ARTICLE 1

Programming Forms with VBScript

You can often develop a reasonably powerful custom solution in Microsoft® Office Outlook® 2007 simply by creating and using Outlook forms with standard and custom controls. However, this approach has limits. For example, what if you want a new task to be created every time a user sends your custom form? What if you want to send a meeting request when your form is submitted? What if you want to connect to a database and fill in a drop-down list on your form with values from a specific database table? Although it isn't practical to explain how to code for all these problems in this introductory chapter, the preceding list offers good examples of some of the tasks that are difficult, if not impossible, to accomplish using only custom forms that have no underlying custom program code. This chapter looks at developing custom form solutions with computer code and custom fields. You'll strengthen your understanding of programming basics and learn how you can begin to apply this understanding to developing a custom Outlook solution.

To create a solution that involves working with dynamic data, launching new items, and animating other programs, you could search for a control that already has these behaviors built in. However, don't spend too much time looking, because you're unlikely to find a control with functionality so specifically tailored to your needs. The way to create powerful forms is by writing code.

For some people, writing code in an Outlook solution simply means learning the syntax of Microsoft Visual Basic Scripting Edition (VBScript) and some of the properties, methods, and events of Outlook and Outlook forms. For other people, writing code might seem much more daunting. No matter what your level of programming experience is, this chapter provides some basic knowledge you need to begin programming Outlook forms.

Understanding Scripting

Programming languages such as C++, Microsoft Visual Basic, VBScript, and C# (pronounced "C sharp") provide access to computer resources at different levels. The basic rule is that the "lower" you go, the more you need to know. Think of it as similar to programming a microwave: when you enter a combination of settings to defrost a chicken that weighs 1.3 pounds, you're programming the oven to do certain things. At a lower level, a program inside the microwave takes the settings you provide and translates them into a cooking plan that the machine can execute. Most often, lower level programming languages, which are usually designed for speed and scalability, are used to create sophisticated, complex applications.

However, the learning curve for these languages can be prohibitive for someone who wants to simply use some controls to create a form that sends an e-mail message. This is where a higher level language such as VBScript steps in, giving the power of programming to users who don't have extensive backgrounds in computer science.

As the name implies, VBScript is a scripting language. A scripting language can create an unlimited number of programmatic solutions relatively quickly. Always remember, however, that you are trading true power and performance for simplicity and speed of development. Unlike a compiled programming language, a scripted language is interpreted—that is, the statements of instruction are reduced to a simple sequence of statements that are then executed by what is called a command interpreter. For each scripting language, a corresponding scripting engine provides this interpreter, along with other basic services to execute the script. When you program Outlook forms, you don't need to do anything to install this engine or ensure that your scripts will actually be executed. Outlook has this engine built in; all you need to do is open the Script Editor and begin writing code.

Because Outlook uses VBScript as the language for form development, you need to know the syntax of VBScript. You also need to know which interfaces are available for you to manipulate with lines of code. The next section provides this information.

A Brief Overview of VBScript

VBScript is a subset of the Microsoft Visual Basic programming language. The VBScript engine is portable, which means the engine can be embedded in or used by many different programs, such as Microsoft Internet Explorer or Outlook. As mentioned earlier, VBScript is already embedded in Outlook. You can begin writing form-based code, and those who use your forms will benefit from the code behind the forms without installing anything special to make the forms work. Because the VBScript engine is a true subset of the complete Visual Basic language, it is less complex and therefore easier to learn.

Understanding Objects and Properties

Object is probably one of the most overused words in the world of computer programming. Over time, the word has come to represent many different things to people in the IT field. Because the focus of this book is not hard-core computer science, this chapter is relatively introductory and covers only what you need to get your Outlook solution moving along. In this context, the following section discusses the notion of an object and what it means to your Outlook solution.

Objects

At the simplest level, an object in your Outlook development environment represents Outlook, the program itself, and other elements in Outlook (such as a text box) that expose properties, methods, or events with which you can interact programmatically. Outside the world of computing, the definition of an object is widely understood. For example, you know that a book is an object. It has certain properties such as height, weight, color, number of pages, title, author, and so on. You can also perform certain actions with a book: you can open it, close it, turn pages one at a time, flip to the end, return to the beginning, and fold the corners to mark your location.

A significant aspect of programming a custom solution using VBScript or Microsoft Visual Basic for Applications (VBA) is manipulating Outlook items. For example, you add a text box to a form and then add or change the text it displays by setting a property of that text box object. You display or close a form by using a Show or Hide method on the form, treating the form as a complete object rather than showing or hiding the individual controls on the form. When a user clicks a button on a form, that button object generates a Click event. You add the code you want Outlook to execute when the button is clicked.

Thinking of Outlook items as programmable objects will help you begin to understand how you can create custom solutions with VBScript and VBA. Don't think of your custom program as pages and pages of program code. Instead, think of it as a collection of forms, controls, and other objects with properties you can set, either at design time or through program code when the form runs, to make it perform the tasks you intend.

Properties

Properties are attributes of an object, describing some aspect of how an object behaves. To return to an earlier analogy, an object in Outlook can have some of the same properties as a book. For example, a form has height, width, and other basic properties that describe appearance.

A command button has properties such as height, width, position, caption, name, and color. The more you know about the properties of the objects you use, the more you can do with those objects. To set a property of an object, you must reference the object and

follow it with a period and then the name of the property. The following code sample shows how to set the property of an object:

```
cmdAddAppointment.Caption = "Add Appt."
```

In this example, assume that your form contains a command button named cmdAddAppointment. You manipulate the Caption property of that button by setting it to some new value. How do you know that the button has this property? For any given object, you must read documentation to know the available properties and other behaviors for that object. For example, a text box does not have a Caption property, but a label does; both these controls have a Font property.

Notice that in the preceding example the Caption property required a string of characters between quotes. This is because the Caption property is predefined to accept characters strung together and not numbers, bitmaps, or other kinds of data. When you work with properties, you need to learn which types of data each property accepts to accurately apply property settings for your objects.

Understanding Events and Methods

Events and methods are two other powerful behaviors of a form or other object in Outlook. To continue with the previous analogy, you can do a number of things with a book: you can open it, close it, and so on. Put another way, the book possesses a number of methods, such as OpenBook, CloseBook, TurnPage, GoToEnd, GoToBeginning, and AddBookMark. These method names are intuitive. The creators of any object model usually give intuitive names to the methods that their object exposes.

You can also associate certain events with a book. For example, when you finish reading the book, you might have to return it to the library. When you open the book for the first time, you might have to remove the book's dust jacket. To return to the microwave analogy, when the microwave completes the cooking plan you've provided, this event causes an alarm to sound. In other words, the microwave has a CookingPlanCompleted event that fires when the time runs out. When this event occurs, the alarm sounds.

In the world of Outlook development, you'll interact with many objects, some of which expose certain events. Arguably, the most common event to fire is some kind of Click event, which fires when a user clicks a command button or other control. Notice the terminology: an event fires when something else happens. This is important to remember, because it is easy to get methods and events confused. Just remember that methods are actions you can take with an object, whereas events fire in response to an action—perhaps in response to a method or the changing of a property setting.

Creating Scripts and Using the Editor

Now that you have a little background information about the properties, methods, and events of the objects you can include in your solution, you're ready to begin writing code.

Where do you write it? Theoretically, you can write the code in any program you want—in Microsoft Word, Notepad, or any other text editor. However, where you write the code is not as important as where it must reside to be able to run. Outlook has a window, or code editor, where all code must be placed before it can run. To access this code editor in Outlook, open a form for editing and click View Code in the Form group of the form's ribbon. In either case, the window shown in Figure A1-1 appears. Any code that you want to run for your Outlook form must be contained in this window.

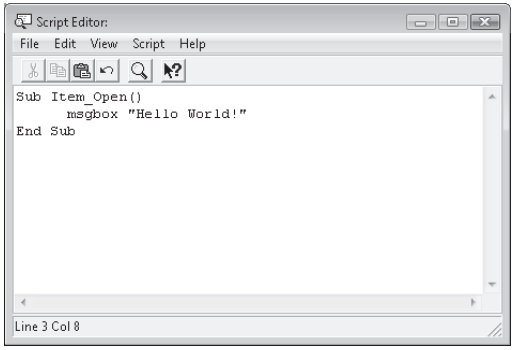

Figure A1-1. You can write and run the code for your form in the Script Editor window.

Object Library Help

The Outlook Script Editor provides roughly the same level of functionality as Notepad or another basic text editor. However, the menu bar of this editor does contain the Outlook Object Library Help, shown in Figure A1-2. The Object Library will come in handy as you develop your programming skills in Outlook. To access this help, choose Help, Microsoft Office Outlook Object Library Help from the editor's command bar. It's helpful to take some time to understand the Outlook object model.

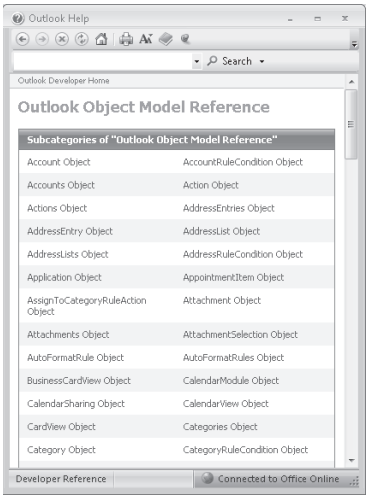

Figure A1-2. Use Outlook Object Library Help when writing code for Outlook.

The other features of the Script Editor—an Event Handler window and an Object Browser window—are useful for entering text as well as viewing available events and objects. Both windows, which are accessible from the Script menu, open simple dialog boxes that allow you to insert small bits of text into the Script Editor window.

The problem is that the text excerpts provided in these windows are not displayed in any contextual way, so unless you already know the full object model, the textual excisions don't make much sense.

When you've finished writing your code sample, all you need to do is close the Script Editor window. Choose File, Close or close the window by clicking the X in the upper right corner. You don't need to do anything special to save the script; it is automatically stored with the form when you publish or save the form.

For information about publishing and saving forms, see "Publishing and Sharing Forms," on page 713 in the book.

Referencing Controls

One of the most common problems faced by Outlook forms programmers is trying to access the properties and methods of controls. This is because Outlook requires you to perform an extra step before you can acquire a reference to a control on a form. In other languages, such as Visual Basic or VBA, you can begin to use methods for a control as soon as you place that control on a form, simply by using its name. For example, if you draw a combo list box control on a form in Visual Basic and name it cboEmployees, you can use the following code to add items to the list in the control:

```
cboEmployees.AddItem "Employee One"
```

> **Note**
>
> The name of the combo box object in this example is prefixed with cbo to indicate that it is a combo box. This type of prefix, called Hungarian notation, helps you identify the type and function of a control when you're browsing or editing your script code. For more information on Hungarian notation and a list of prefixes, see Microsoft Knowledge Base article 173738.

In Outlook's VBScript environment, it is not possible to add items to the list the way you do in Visual Basic (for reasons that are beyond the scope of this book). To begin adding items to cboEmployees with VBScript, you first need to declare a variable and set cboEmployees equal to that new variable. You then use the new variable to execute methods and set properties.

The following code sample illustrates how to get a reference to a control named cboEmployees on a form and then add an item to that control programmatically:

```
Dim cboEmployeesRef
Set cboEmployeesRef = Item.GetInspector._
ModifiedFormPages("P.2").Controls("cboEmployees")
cboEmployeesRef.AddItem "Employee One"
```

Notice the new variable in play here, cboEmployeesRef. This is actually just an empty container that you use to fill with a pointer to the real combo box control, cboEmployees. Figure A1-3 shows the form.

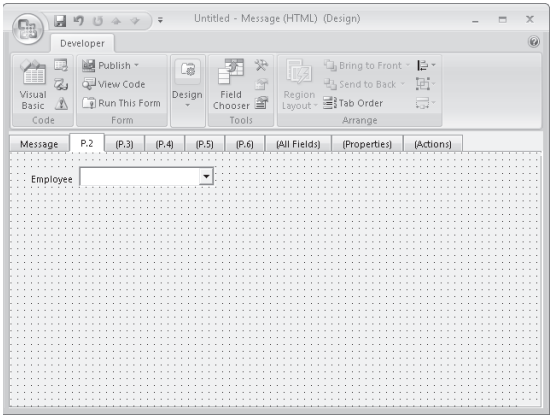

Figure A1-3. This example places a combo box control on an Outlook form.

The control is on the second page of the form; the caption for this page is P.2. You use this name in your code to find out which page the control was placed on. Then you sift through the collection of controls on this page of the form to find the one that interests you—in this case, cboEmployees. You next set your variable equal to the control you've found on this form page. With the variable now acting on behalf of the real control, you

can use the properties and methods of that control by referencing the variable, in this case, cboEmployeesRef.

Special reference names

Although you need the special reference to the control to be able to work with properties and methods on an Outlook form, you don't use the special reference name when dealing with events. In other words, although you must write *cboEmployeesRef.AddItem* to add an item, the events associated with *cboEmployees* are listed under that name and not under the name *cboEmployeesRef*. Confused? Don't worry—you aren't the first to find it a little befuddling. After you work with forms for a short while, you'll find it a little more comprehensible.

Using Methods

Methods are invoked by referencing the programmable object followed by a period and the method name. Let's go back to the form with cboEmployees. Suppose that you want to execute the AddItem method of the cboEmployees combo box. First you must build and use a form on which you can place the combo box.

Follow these steps to create the form:

1. While in Outlook, open a new standard message form in design mode.
2. Leave the Message form page alone for now. To make the second page visible, click its tab, click Page in the Design group on the Ribbon, and choose Display This Page.
3. Add a label control to the form and position it in the upper left corner of the form.

   For details on adding controls such as labels to a form, see "Adding and Arranging Controls," page 699.
4. Add a combo box control to the form and position it just to the right of the label control.
5. Right-click the label, choose Properties from the shortcut menu, and set the label's caption equal to Employees:.
6. Right-click the combo box, choose Properties from the shortcut menu, and set its name to cboEmployees.
7. Verify that your form now looks like the preceding one shown in Figure 40-3.

Now you must prepopulate this combo box with a few values so that when the user opens the form, the combo box provides a list of employees. To do this, you need to write code in an event handler for the item itself. To simplify things for this example,

assume that you'll place explicit values in the control's list. (In the real world, you'd use a few more lines of code to dynamically populate the control with an available list of employees or e-mail addresses from a database.) This code goes in the Item_Open event, which fires when a user opens the form. (Event handling is discussed in the following section.) For now, place the following code in the Script Editor:

```
Private cboEmployeesRef

Sub Item_Open()
   Set cboEmployeesRef = Item.GetInspector._ ModifiedFormPages("P.2").Controls_
        ("cboEmployees")
   cboEmployeesRef.AddItem "Employee One"
   cboEmployeesRef.AddItem "Employee Two"
   cboEmployeesRef.AddItem "Employee Three"
   cboEmployeesRef.AddItem "Administrator"
   cboEmployeesRef.ListIndex = 0
End Sub
```

If this seems confusing, it's because of the assumption that you know what you're supposed to type after the AddItem method is invoked. The only way you could know this is by reading documentation to learn how the combo box in this example works. Reading code others have written can also help you learn the properties, methods, and events of the objects you use. If you fail to type the correct syntax or you try to execute a method that doesn't exist, Outlook presents an error message. As you fix problems in your code, pay careful attention to the messages you receive.

Event Handling

In the code sample used in the preceding section, you wrote code for adding employees to the combo box between the following lines:

```
Sub Item_Open()
End Sub
```

These lines bracket what is known as a procedure. Think of a procedure as a small program that runs inside your form. In this example, the procedure runs in correspondence with a specific event. The code between Sub Item_Open( ) and End Sub runs whenever the form is opened. Because this procedure is tied to an event, it is called an event handler. You place the code to populate the combo box in this event handler because you want the combo box to be filled every time the form is opened.

Other common event handlers are Item_Close, Item_Send, and CommandButton_Click.

Using Variables

You'll find variables useful when you develop an Outlook form solution; they are useful in other programming environments as well. If you ever took algebra, the notion of a

variable will be familiar to you. If you've never heard of or used variables before, don't worry; they're easily understood. This section takes a look at an example of variables outside the computing world and also discusses scope.

Scope

In VBScript (as in any programming language), there exists the notion of scope. Scope refers to the context in which a variable is active. Say that you're going to the movies with friends. Two friends stand in line for you while you park the car. While one friend (Friend1) waits for you at one movie theater, another friend (Friend2) waits for you at a different movie theater. Friend1's scope is the line where she is waiting. The same is true of Friend2. Just because Friend1 is first in line at one theater doesn't mean that your place is first in line at the other theater. The scopes are separate.

Pay attention to where you declare variables and to the words you use to declare them, because these factors determine the scope of your variable. The words, or statements, for declaring a variable in VBScript are Dim, Public, or Private. Use Dim only when you declare a variable within a procedure or function. Public and Private are used to declare variables at the beginning of a code module. Private is the most commonly used statement; use it when coding Outlook forms. You use Public to make variables or procedures in the module available to other code modules. Because this isn't possible in Outlook form development, you don't need to worry about the Public statement.

Variables

Think of a variable as a temporary container for something else. In the movie example, you asked two friends to wait in line for you while you parked the car. Your friends were actually the variables—they represented you. In computing terms, a variable is a named place in computer memory. You can give a variable just about any name you want. This variable will hold a place in memory and store some other data, such as a number or a string of characters. Whatever is contained in the variable is the value of that variable, and you can set, change, and read the value when needed.

Declaration

Before you can start using a variable, you should declare that variable. In VBScript, you aren't actually obligated to declare variables, but doing so is always a good idea. It's also a good idea to add comments to variable declarations that are obscure or less easily understood, thereby allowing readers of your code to understand what the variable represents. When you declare a variable, you're telling the scripting engine that you have a variable of a specific name and that any time this name is used, the variable is being used. Again, declare variables by using a Dim or Private statement.

In the code sample on page A8, you used the Private statement to declare a variable called cboEmployeesRef. This variable held a pointer, or reference, to the actual combo box control. The scope of this variable is global because you made the declaration at the beginning of the module and not in a specific procedure or function. As you become more comfortable with programming, you'll find that you might make many declara-

tions, both within and outside procedures. The following code sample is taken from a rather sophisticated form:

```
Private mobjSession                  ' CDO Session
Private mobjNS                       ' Namespace
Private mstrCurrentUserName          ' Name of current form user form user
Private mstrCustomerID               ' CustomerID retrieved from database
Sub cmdModifyCustomer_Click()
   On Error Resume Next
   Dim objMessage ' Temporary message holder
   Dim objRecipient ' Recipient object that will receive message
   Dim strEmployeeID ' The assigned Customer Agent's ID
End Sub
```

Notice how the variables were declared and commented. Notice also that the subprocedure contains its own variable declarations. These are usually made at the beginning of a procedure and not throughout, making the code more readable. Remember that other people will inevitably need to read, understand, and support your code, so it is more than just a professional courtesy to write, organize, format, and comment your code carefully.

Using Constants

Like variables, constants are containers for data. You can place a text string in a variable or in a constant and then reference and use both in much the same way. The essential difference between variables and constants is this: after you set the values of constants at design time, the program can only read these values at run time. Variables, however, allow the program to change their value when appropriate. Given this difference, constants are declared and used to hold information that you don't want to change while the program is being executed. For example, suppose that your code contains some procedures that all need to use the following string: "Region: 2330n; Section: 4; District: 5." Typing this in over and over in each procedure would be tedious. In addition, what if the company reorganized Section 4 into Sections 4A and 4B and also renamed the region? You would then have to go into your code and replace the string in each procedure. The risk for error and the amount of effort required (even with the magic of Copy/Paste) is higher than if you were simply to declare a constant in one part of your code and use that constant in all the procedures. Your constant declaration would look like the code shown here:

```
Const AGENT_LOCATION = "Region: 2330n; Section: 4; District: 5."
Const PO_AUTHCENTER = &HA02D001E ' Hex value for PO authorization
```

This code contains another constant (PO_AUTHCENTER) that illustrates an especially useful aspect of constants: they can make esoteric values readable. The code contains a hexadecimal value that corresponds to an authorization number for purchase orders. The hexadecimal value won't make much sense to other people who have to read the code. Furthermore, if your code contains several of these cryptic values, you'll probably become confused as well. Using constants allows you to assign this enigmatic value a user-friendly name that will make sense.

Declare constants at the beginning of your code module, and use the Const keyword, followed by the name of the constant in uppercase characters. Using all uppercase letters is a common practice that allows readers of your code to easily determine whether the name they are about to use is a variable or a constant. Make sure your constant name is user-friendly and comprehensible within the context of your business solution. Using names that are too short or known only to you will diminish some of the advantages that constants provide. After you provide a name for your constant, type an equal sign followed by the value of the constant.

Retrieving and Setting Field Values

Outlook has a number of item types that form the basis of practically any collaborative Outlook solution. Each item type has a number of fields or properties that you can set and read in your code. You can easily set these fields or properties by referencing the Item object and the property in question.

The following code sets the Subject field of a MailItem:

```
Item.Subject = cboEmployeesRef.List(cboEmployeesRef.ListIndex)
```

This code sets the Subject property of the Item object to the text value of the currently selected item in the combo box.

The value of the Subject property is also accessible for reading purposes. For example, you might want to put code in the Item_Send event to first check to see whether the Subject field is blank. If it is blank, set the value equal to the text shown in the combo box; otherwise, leave the value as the user typed it. The code would look like this:

```
If Len(Item.Subject) = 0 Then
   Item.Subject = cboEmployeesRef.List(cboEmployeesRef.ListIndex)
End If
```

The preceding code checks the length of the Subject property. If the length of that property is 0, you know that the user has not typed any text into the Subject field. You can then let the code do a little work for the user by automatically adding a value to the field.

Retrieving User-Defined and Other Properties

You can create your own custom properties. These are properties that your solution requires but are not provided by the item types available to you. For example, if you deal with employees, you might want to have a property that refers to an ID for the division in which an employee works. None of the item types that Outlook provides has a DivisionID property. The good news is that you can add a new property for this data. These user-defined properties, sometimes called user-defined fields, can also be bound to controls. In Chapter 28, “Designing and Using Forms,” you learned to bind controls to item

properties. You can bind these same controls to a user-defined property the same way you bind them to a regular, built-in property.

The sample form in this chapter contains a list of employees. A senior manager selects an employee and sends the message to someone who is supposed to interview the employee. When the user receives the message and opens it, she or he is given the opportunity to click a button and automatically add a task to a private task list. This task will already contain the name of the person to be interviewed in the subject line.

This form will be defined with two pages, both with Compose and Read areas. The Compose area of the second page includes a combo box that will be populated with employee names. The Read area of the second page contains a couple of labels, one of which will be bound to a user-defined property. This page should look like the one shown in Figure A1-4. To get to this stage, you need to create a new property for the form, named EmployeeToInterview.

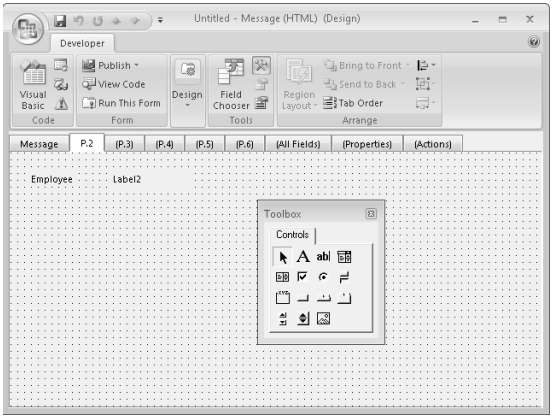

Figure A1-4. The Read area of the second form page contains two label controls.

Follow these steps to create the new user-defined property:

1. Activate the separate Read and Compose areas by choosing Form, Separate Read Layout.
2. With the form selected, choose the All Fields tab in design mode. You should see the pane shown in Figure A1-5.

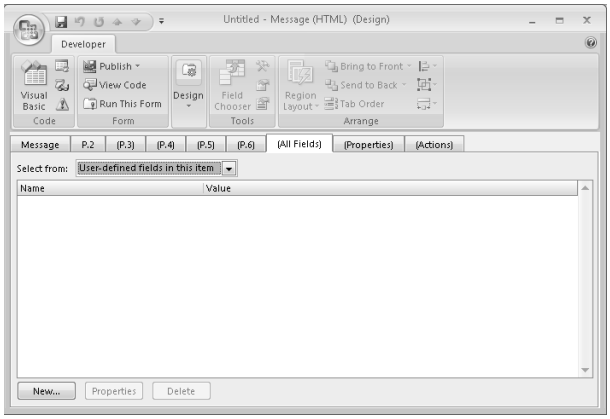

Figure A1-5. Use the All Fields tab to create a new user-defined property.

3. Click New and type the name of the new property (EmployeeToInterview). Click OK. Optionally, you can type a default value for this user-defined property in the Value column.

Notice that this user-defined property stores text. When you create a user-defined property, you can decide what type of data the property will hold. You can also prescribe the format of the data the property holds. For example, if you define a new property called AmountSaved and choose the Currency data type, you could choose how to display the currency values—for example, you might want to omit decimal values and keep the figures at the dollar level only.

When you create user-defined properties, you can bind controls to them. In this case, you bind the second label (lblEmployee) on the Read area of the second page to the user-defined property EmployeeToInterview. You also add code so that when the user sends the form, the value of the new property is set equal to the value shown in the combo box. Without this additional code, the value shown in the label bound to the user-defined property would not be the dynamic value the user has chosen in the combo box. In other words, you have created a user-defined property for this item. When the user chooses an employee from the combo box and sends the message, the value of the combo box is stored in the user-defined property. When the recipient reads the message, the label on the second page is already populated with the value of the user-defined property. The following code is necessary to make the form function properly:

```
Private cboEmployeesRef

Sub Item_Open()
   If Item.Size = 0 Then
       Item.To = "Administrator"
   Set cboEmployeesRef = Item.GetInspector._
       ModifiedFormPages("P.2").Controls("cboEmployees")
   cboEmployeesRef.AddItem "Employee One"
   cboEmployeesRef.AddItem "Employee Two"
   cboEmployeesRef.AddItem "Employee Three"
```

```
   cboEmployeesRef.AddItem "Administrator"
   cboEmployeesRef.ListIndex = 0
   End If
End Sub

Sub Item_Send()
   If Len(Item.Subject) = 0 Then
        Item.Subject = cboEmployeesRef.List(cboEmployeesRef.ListIndex)
   End If
   Item.UserProperties.Item("EmployeeToInterview").Value = _
   cboEmployeesRef.List(cboEmployeesRef.ListIndex)
End Sub
```

In this simple example, you both set and retrieve the values of the user-defined property.

However, there's a little more to be said about user-defined properties. After you create a user-defined property, how can you be sure that recipients and users of your form will have access to it? What do you need to be able to manage these properties and ensure their integration in a solution? The short answer to these questions is that user-defined properties exist in the place where you publish your form.

For information on publishing forms, see "Publishing and Sharing Forms," page 653.

For example, assume that you create a form to specify travel preferences, such as airplane seating choice and meal types. You create new user-defined properties for each piece of information and publish the form to your Inbox. Those properties are now available in your Inbox; you'll find them listed when you use the Field Chooser dialog box. As it happens, your form is so successful that you need to make it available to others, and you choose to publish the form in a public folder called Travel Planning. When you do this, the user-defined properties you created become available to anyone who opens your posts in that folder.

For information on using the Field Chooser, see "Outlook Fields," page 696.

Custom Formula and Combination Fields

Formula fields and combination fields allow you to join together multiple fields and manipulate their data. In some cases, the piece of information you want users to view in a list of sent items will be a field value that has been altered by a function, such as one that makes all characters in a field expressed in uppercase. In another example, think of an instance in which two fields contain important values for your solution, but users are accustomed to seeing the two values placed together, such as a given name and a surname. Rather than forcing users to enter three fields—one for the given name, one for the last name, and one for the union of the two—you can create a combination field that automatically displays the combination of the two other fields. Both formula and combination fields are constructed in much the same way as the other user-defined properties you've learned to create. However, instead of choosing a data type such as Text or Number, you choose Formula or Combination.

To create a combination field, follow these steps:

1. Open the form you're modifying in design mode.
2. Select the All Fields tab and click New.
3. Type a field name and select Combination for the type.
4. Click Edit.
5. Select the first option in the Combine Field Values By frame.
6. Click Field to add a field. Repeat this step for as many fields as you want to insert.
7. Click OK twice to complete the field addition.

Before you click OK, your New Field dialog box should look like Figure A1-6.

Figure A1-6. Create a combination field to simplify the way users view information.

In step 5, you chose the first option. The two options have to do with how empty field values are displayed. For example, in the case of combination fields, one or more of the fields might be empty, whereas others will contain complete values. You can have Outlook display all the complete fields by selecting the first option. On the other hand, if you combine two or more field values and you want only the first nonempty field to display, choose the second option.

For example, you might choose this option when you want to use either a person's first name (if it has a value) or the person's nickname (if it has a value), but not both. In this case, you would include both fields in the combination but select the second option. Only the first field with a value is shown.

You create a formula field in much the same way you create a combination field, except that you choose the Formula type instead of the Combination type. When you click Edit, you see a Field button and a Function button in the dialog box, allowing you to insert functions as well as field names. The Function button greatly enhances the power of the field you have created by making available a large number of functions related to time, date, financial formulas, textual operations, and mathematical formulas.

The ability to employ user-defined properties expands what you can do with an Outlook solution. User-defined properties allow you to extend Outlook forms and item types to include information tailored to your organization and operations. Creating user-defined properties is a common task for building customized form-based solutions, and the properties should correspond to data elements in the environment where they will be used.