The Definitive Guide to the .NET Compact Framework

LARRY ROOF AND DAN FERGUS

Apress™

ISBN (pbk): 1-59059-095-3

Printed and bound in the United States of America 12345678910

Technical Reviewer: Ron Miller

Editorial Board: Dan Appleman, Craig Berry, Gary Cornell, Tony Davis, Steven Rycroft, Julian Skinner, Martin Streicher, Jim Sumser, Karen Watterson, Gavin Wray, John Zukowski

Assistant Publisher: Grace Wong

Project Manager: Tracy Brown Collins

Copy Editor: Ami Knox

Production Manager: Kari Brooks

Compositor, Proofreader, Artist: Kinetic Publishing Services, LLC

Indexer: Ron Strauss

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States: fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, email info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section. You will need to answer questions pertaining to this book in order to successfully download the code.

CHAPTER 18

Working with POOM

Included as part of the software bundled with every Pocket PC is a set of applications that are referred to jointly as *Pocket Outlook*. Individually, these are the Calendar, Contacts, Inbox, Notes, and Tasks applications.

Although functionally these applications operate in a stand-alone fashion, they are designed for use in conjunction with their larger, better-known desktop equivalent, Microsoft Outlook. Between these two platforms of PIM applications resides ActiveSync, which natively handles synchronizing data between Microsoft Outlook and the Pocket Outlook applications.

From the development point of view, Pocket Outlook offers two opportunities. First, it provides a repository of specific data, that being contacts, appointments, and tasks. Second, it comes with automatic synchronization, allowing you to be worry free of how this data moves between a device and its associated PC.

The Pocket Outlook Object Model

Applications access Pocket Outlook data through the Pocket Outlook Object Model, or as it's more commonly referred to, POOM. This COM-based library provides an object hierarchy that simplifies the process of creating, modifying, and displaying appointments, tasks, and contacts.

The problem is that, unlike applications written in eMbedded Visual C++ or eMbedded Visual Basic, those that target the .NET Compact Framework don't natively have access to COM libraries, which means, among other things, you can't directly leverage POOM.

As with all NETCF-to-COM situations, the workaround is to create a non-COM dynamic link library, or DLL. This DLL acts as a go-between for your NETCF application and the COM object. The DLL is written using eMbedded Visual C++, which as I've already mentioned can directly access COM objects, including POOM. The DLL exposes a standard NETCF-friendly interface that in turn can be PInvoked from your application.

Now, as you may have already guessed, this isn't a trivial process. It requires a detailed understanding of eMbedded Visual C++, the Pocket Outlook Object Model, and creating DLLs and PInvoking DLLs from a NETCF application. Given the right amount of time and effort, you probably could struggle through this on your own. Luckily, there are two sources of aid that can significantly shorten this

process: the POOM sample on the GotDotNet Web site and the Pocket Outlook .NET component offered by InTheHand.

GotDotNet's POOM Sample

The POOM sample offered on the GotDotNet Web site demonstrates creating a DLL written in eMbedded Visual C++. You learn how to call this DLL from a NETCF application. While somewhat limited in functionality, offering access to contacts only, it does show the key points and concepts behind leveraging POOM from a .NET Compact Framework application.

One downside of this sample is that it's written completely in eMbedded Visual C++, and converting the NETCF component to Visual Basic .NET isn't a trivial task.

This sample is available for download at `http://www.gotdotnet.com/team/netcf/Samples.aspx`.

InTheHand's Pocket Outlook .NET

The second option available to NETCF developers is the Pocket Outlook .NET component from InTheHand, a software development shop that specializes in NETCF. Pocket Outlook .NET is a set of .NET classes that allow full read/write access to the Appointments, Contacts, and Tasks features of Pocket Outlook.

Unlike the GotDotNet sample application, Pocket Outlook .NET is a library DLL that you can add to your .NET Compact Framework projects to provide a robust object hierarchy. This greatly simplifies incorporating POOM into your applications.

The Pocket Outlook .NET library supports data binding, so you can quickly build PIM-enabled applications using standard NETCF components such as the DataGrid and ComboBox.

From my point of view, Pocket Outlook .NET is the way to go when you need to work with Pocket Outlook data. It's easy to use, offers the right set of features, and is priced right for even the smallest of development shops. This is a cool product. In fact, all of the examples and tutorials in the remainder of this chapter utilize Pocket Outlook .NET.

For more information on this product, see `http://www.inthehand.com`.

Accessing POOM from NETCF

As I've already mentioned, there are two ways to access POOM from the .NET Compact Framework. The first involves creating your own native-code DLL.

Detailing this approach is outside of the scope of this chapter because of its complexity.

The second approach makes use of the Pocket Outlook .NET component from InTheHand. All of the examples and tutorials that follow make use of this component. You'll need to download and install this component to follow along or utilize any of the techniques demonstrated in the remainder of this chapter.

Working with the Pocket Outlook .NET Component

Once you've downloaded and installed the Pocket Outlook .NET component, there are a few steps that you need to perform to make use of this component within your .NET Compact Framework applications:

1. Add a reference to the Pocket Outlook .NET component.

2. Import the **PocketOutlook** namespace.

3. Define the OutlookApplication object.

These steps are detailed in the following sections.

Adding a Reference to the Pocket Outlook .NET Component

The first step that you need to perform is to add a reference to the Pocket Outlook .NET component. To accomplish this, perform the following steps:

1. With your project open, in the Solutions Explorer window right-click the **References** folder.

2. From the pop-up menu, select **Add Reference**.

3. The Add Reference dialog box will display. From the list at the top of this dialog box, select **InTheHand.PocketOutlook**.

4. Click the **Select** button. The InTheHand.PocketOutlook component will be added to the Selected Components window at the bottom of the dialog box as shown in Figure 18-1.

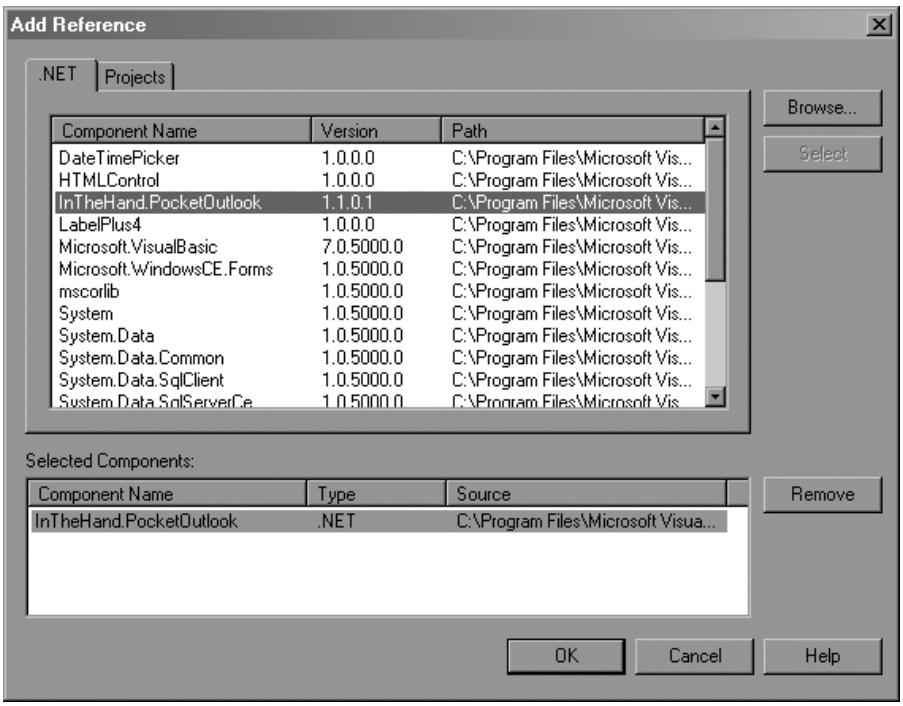

Figure 18-1. Adding the reference to the Pocket Outlook component

5. Click the **OK** button, adding the reference to your project.

Importing the PocketOutlook Namespace

The second step that you need to perform is to add an Imports statement at the top of your form module. To accomplish this, perform the following steps:

1. Open the code window for your form.
2. Navigate to the top of that code window.
3. Add the following line of code to the module:

```
Imports InTheHand.PocketOutlook
```

Defining the OutlookApplication Object

The third step that you need to perform is to declare a variable for the OutlookApplication object. This object is the interface to the Pocket Outlook Object Model. You must always include it in your application.

Following is an example of declaring a variable for this object. Typically, this declaration will occur at the module or project level of your application, so that the OutlookApplication is available across a broad scope.

```
 [at the module level]
Dim poApplication As New OutlookApplication
```

NOTE *When this line of code executes, your application automatically logs in to the Pocket Outlook data source. Unlike working with POOM directly, where you need to log in before performing any task, the InTheHand Pocket Outlook .NET component performs this step for you.*

Obtaining Help on the Pocket Outlook .NET Component

The installation package for the Pocket Outlook .NET component includes a help file titled Object Model Documentation. The installation process creates a shortcut to this document. To access this documentation, perform the following steps:

1. Click the **Start** menu.
2. From the Start menu, select **Programs**.
3. From Programs, select **InTheHand**.
4. From InTheHand, select **Pocket Outlook .NET Wrapper**.
5. From Pocket Outlook .NET Wrapper, select the shortcut to **Object Model Documentation**.

This help system provides an overview of the object hierarchy provided through the Pocket Outlook .NET component. It's not intended as a how-to guide, but rather as a reference. If you're looking for how-to material, then the rest of this chapter is for you, starting with how to work with Pocket Outlook tasks.

An Important Consideration for Developers Using POOM

When developing an application that uses POOM to modify and query data within Pocket Outlook, you must always keep in mind that the data you are working with is also accessible directly by the user from the native applications themselves—Contacts, Appointments, and Tasks. What that means is the data you are incorporating into your application may be altered outside of your application in a way that may adversely affect your programming scheme.

Working with Tasks

The Pocket Outlook .NET component provides the ability to work with three facets of the Pocket Outlook data: tasks, contacts, and appointments. I'll start the discussion of this component with tasks.

From the developer's point of view, tasks can be used for a variety of purposes. For example, they could be used within a maintenance application to record follow-up work to be completed. In a delivery system, tasks could be used to note items for the next delivery. A CRM application could record steps to perform for a particular customer.

At the core of the task functionality provided through the Pocket Outlook .NET component from InTheHand is the **Tasks** property and the Task object. We'll look at each of these items in greater detail next.

The Tasks Property

The **Tasks** property of the OutlookApplication object provides access to a collection of the tasks that reside on a device. In actuality, the **Tasks** property furnishes an interface to the **Tasks** folder, provided through POOM. This property links through to an Items collection that contains all of the tasks.

We'll see several examples later in this section in which the **Tasks** property along with its Items collection is used to retrieve tasks.

The Task Object

You work with individual tasks through the Task object. Commonly used properties of this object are shown in Table 18-1. Frequently used methods of the Task object are shown in Table 18-2.

Table 18-1. Commonly Used Properties of the Task Object

PROPERTY	DESCRIPTION
Body	Defines the text of the notes accompanying a task
Categories	Indicates the categories for which a task is assigned
Complete	Indicates whether the task is complete
DateCompleted	Specifies the date that a task was completed
DueDate	Specifies the date that a task is due
Importance	Dictates the importance of a task
IsRecurring	Indicates whether a task is a single instance or recurs
Oid	Defines the unique identifier for a task
ReminderSet	Indicates whether to remind the user of the task
StartDate	Specifies the date that a task starts
Subject	Indicates the subject for a task

Table 18-2. Commonly Used Methods of the Task Object

METHOD	DESCRIPTION
Copy	Creates a copy of an existing task
Delete	Deletes a task
Display	Displays a task using the native Task interface
Save	Saves modifications to a task

While these tables might serve you well for reference, a set of practical examples follows that demonstrates commonly performed task-related operations.

Retrieving All Tasks

One of the most common functions involving tasks is the retrieving of a task(s). You can retrieve tasks in several ways:

- As a collection of all tasks
- As a subset of all tasks
- As a single task

We'll start by looking at how to retrieve all tasks, as it's a frequently used approach and the easiest method for retrieving tasks. Listing 18-1 provides a simple example of retrieving all of the tasks that are resident on a device.

Listing 18-1. Retrieving All of the Tasks

```
Imports InTheHand.PocketOutlook

[at the module level]
Dim poApplication As New OutlookApplication
Dim myTasks As OutlookItemCollection

Private Sub btnLoad_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles btnLoad.Click

' Retrieve all of the tasks.
  myTasks = poApplication.Tasks.Items

End Sub
```

The key parts to this example are

- The Imports statement, which must be located at the top of a module.
- The declaration statement for the OutlookApplication object.
- The declaration of the OutlookItemCollection object, which can hold a collection of any Outlook items. In our case, it will hold a collection of tasks.
- Loading the collection of tasks from the OutlookApplication object, poApplication, into the OutlookItemCollection, myTasks.

At this point, the collection myTasks contains a set of task objects, one for each task that is resident on your test device. You can loop through this collection to propagate a ListBox or ComboBox, view specific task information, or reference a specific task within the collection.

Retrieving Select Tasks

There are times when you only want to retrieve specific tasks—either a single task, or perhaps a subset of tasks. Happily, it's easy to do using the **Restrict** method of the Items collection. Listing 18-2 demonstrates retrieving a subset of tasks. This example retrieves only those tasks with the category "demo".

Listing 18-2. Retrieving Select Tasks

```
Imports InTheHand.PocketOutlook

[at the module level]
Dim poApplication As New OutlookApplication
Dim myTasks As OutlookItemCollection

Private Sub btnSelect_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles btnSelect.Click

  Dim strCategory As String
  Dim strQuery As String

' Retrieve the selected tasks.
  strCategory = "demo"
  strQuery = "[Categories] = " & ControlChars.Quote & strCategory & _
    ControlChars.Quote
  myTasks = poApplication.Tasks.Items.Restrict(strQuery)

End Sub
```

Much of the preparatory work is identical to that required to retrieve all of the tasks. You still need to add the Imports statement, declare the OutlookApplication object variable, and declare the OutlookItemCollection object variable.

The code used to select only the tasks you're interested in is located near the bottom of Listing 18-2. There you build a query string, which is nothing more than the WHERE part of a SQL SELECT statement. This query string is subsequently used with the **Restrict** method to retrieve only those tasks in which you're interested.

NOTE *You can restrict tasks based upon* any *property provided through the Task object.*

Displaying a Task

One of the cool features provided through POOM is the ability to display Pocket Outlook data in its native application interface—that's to say, just like it would appear to users had they gone into the Tasks application and selected to view a particular task.

POOM includes a method, **Display**, which is provided through the Pocket Outlook .NET component. Listing 18-3 demonstrates working with this method.

Listing 18-3. Displaying a Task

```
Imports InTheHand.PocketOutlook

[at the module level]
Dim poApplication As New OutlookApplication
Dim myTasks As OutlookItemCollection

Private Sub btnDisplay_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles btnDisplay.Click

  Dim myTask As Task

' Retrieve all of the tasks.
  myTasks = poApplication.Tasks.Items

' Display the first task.
  myTask = myTasks.Item(0)
  myTask.Display()

End Sub
```

As with the two previous examples, you start by adding the Imports statement and declaring the variables for OutlookApplication and OutlookItemCollection.

At the bottom of Listing 18-3 you'll see that all of the tasks are first retrieved. From this collection of tasks, you extract a single task, the first, into a Task object variable. This object provides the **Display** method, which in turn is called to display the selected task.

Adding a Task

Adding a new task is a three-step process. First, you need to create a new task. Second, you need to configure the task. Third, you need to save the task. Listing 18-4 demonstrates adding a task.

Listing 18-4. Adding a Task

```
Imports InTheHand.PocketOutlook

[at the module level]
Dim poApplication As New OutlookApplication

Private Sub btnAdd_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles btnAdd.Click

  Dim myTask As Task

' Create a new task.
  myTask = poApplication.CreateTask

' Configure the task.
  With myTask
    .Body = "This is a sample task."
    .Categories = "demo"
    .DueDate = Today.Date
    .Importance = Importance.High
    .ReminderOptions = ReminderOptions.Dialog
    .ReminderSet = True
    .StartDate = Today.Date
    .Subject = "sample task"

' Finally, save the task.
    .Save()
  End With

End Sub
```

As with all of the previous examples, you start by adding the Imports statement and declaring the variables for OutlookApplication and OutlookItemCollection.

At the bottom of Listing 18-4 you'll see the three steps that I described. First, the **CreateTask** method of the OutlookApplication object is used to create your new task. Second, properties of the new task are loaded. Third, the Task object's **Save** method is called to save the task.

Modifying a Task

Modifying a task is similar to adding a task, only instead of creating a new task, you load a Task object with an existing task. Listing 18-5 shows how to modify a task.

Listing 18-5. Modifying a Task

```
Imports InTheHand.PocketOutlook

[at the module level]
Dim poApplication As New OutlookApplication
Dim myContacts As OutlookItemCollection

Private Sub btnModify_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles btnModify.Click

  Dim myTask As Task

' Retrieve all of the tasks.
  myContacts = poApplication.Tasks.Items

' Modify the first task.
  myTask = myTasks.Item(0)
  With myTask
    .Body = "This is updated content."
    .Save()
  End With

End Sub
```

Once again, you start by adding the Imports statement, declaring the variables for OutlookApplication and OutlookItemCollection. At the bottom of Listing 18-5 you'll see where the task is first retrieved (along with all other tasks) and then loaded into the Task object. At this point, you can modify any of the Task object's properties. Finish the modification process by calling the Task object's **Save** method.

Now that you've seen the basics of working with tasks, let's pull it all together in a comprehensive example.

Step-by-Step Tutorial: Working with Tasks

In this step-by-step exercise, you will create an application that demonstrates working with tasks within Pocket Outlook. As part of this exercise, you'll

- Connect to Pocket Outlook.
- Retrieve a list of all tasks.
- Retrieve a list of tasks based upon their **Category** property.
- Display a task.
- Add a task.

This application provides all of the fundamentals of working with tasks within Pocket Outlook.

NOTE *To complete this tutorial, you will need the PocketOutlook component from InTheHand, available at* `http://www.inthehand.com`.

NOTE *I provide a completed version of this application, titled Tasks - Complete, under the* ***Chapter 18*** *folder of the* ***Samples*** *folder for this book. See Appendix D for more information on accessing and loading the sample applications.*

Step 1: Opening the Project

To simplify this tutorial, I've already created the project and the user interface for the Tasks Tutorial application. This template project is included under the **Chapter 18** folder of the **Samples** folder. To load this project, follow these steps:

1. From the VS .NET IDE Start Page, select to open a project. The Open Project dialog box will be displayed.
2. Use this dialog box to navigate to the **Chapter 18** folder under the **Samples** folder for this book.
3. Select and open the project **Tasks**. The project will be loaded onto your computer.

Step 2: Examining the User Interface

The user interface for the Tasks Tutorial application comprises several controls: a ListBox, four Buttons, and a TextBox. Figure 18-2 shows the application's interface.

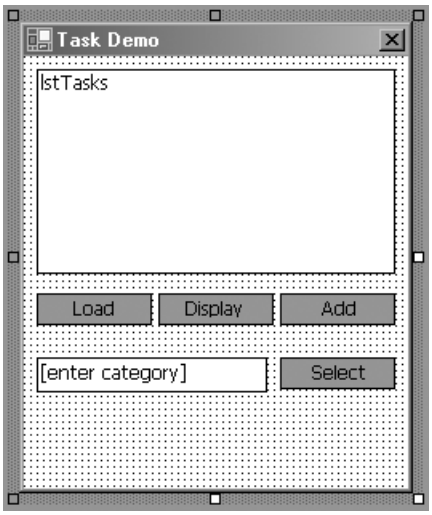

Figure 18-2. The Tasks application interface

The ListBox will display available tasks.
The four buttons for the Tasks Tutorial application function as follows:

- The **Load** button triggers loading a list of all tasks into the ListBox.
- The **Display** button triggers displaying the selected task.
- The **Add** button adds a new task with predefined values.
- The **Select** button retrieves a list of tasks for a select category.

The TextBox allows the user to specify the name of a category.

Step 3: Adding a Reference to the PocketOutlook Component

You need to add a single reference to your application to enable you to work with the Pocket Outlook Object Model from a .NET Compact Framework application.

To add this reference, perform the following steps:

1. In the Solution Explorer window, right-click the **References** folder. A pop-up menu displays.

2. From the menu, select **Add Reference**.

3. The Add Reference dialog box displays. Select the **InTheHand.PocketOutlook** component.

4. Click the **OK** button to add the selected component to your project.

Step 4: Adding the Imports Statement

The first line of code you'll be adding imports the **PocketOutlook** namespace. This allows you to reference and work with the PocketOutlook elements within your code without having to fully qualify each element.

To import the **PocketOutlook** namespace, perform the following steps:

1. Open the code window for the form.

2. At the top of the module, above the line that declares the **Form** class, add the following line of code:

```
Imports InTheHand.PocketOutlook
```

Step 5: Declaring Module-Level Objects

This tutorial uses two module-level object variables. These variables hold instances of the PocketOutlook OutlookApplication and OutlookItemCollection objects.

To define these variables, add the following code to the module level of the form:

```
Dim poApplication As New OutlookApplication
Dim myTasks As OutlookItemCollection
```

Step 6: Loading a List of All Tasks

The first functionality that you'll add to the application is to display a list of all tasks. Obtaining this list is simple. Loading the list into your ListBox requires nothing more than a For loop for running through the collection.

The first part of this step obtains a list of tasks. While you could simply reference the Tasks collection of the OutlookApplication object, you are instead going to retrieve a copy of the Tasks collection into a collection of its own. Having a collection of tasks that matches those tasks displayed in your ListBox makes it easier to display the details of an individual task. You'll see more on this later in the tutorial.

The code required to retrieve the Tasks collection is shown in Listing 18-6. Add this code to the **Click** event procedure of the **Load** button. In this procedure, you create a copy of the Tasks collection through the OutlookApplication.Contacts.Items collection.

Listing 18-6. Retrieving a List of All Tasks

```
Private Sub btnLoad_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles btnLoad.Click

' Store the collection of tasks for future use.
  myTasks = poApplication.Tasks.Items

' Load the list of tasks.
  LoadTasks()

End Sub
```

Displaying the list of tasks is triggered at the bottom of the procedure shown in Listing 18-6. The **LoadTasks** procedure is a general-purpose procedure that displays the contents of the myTasks collection in a ListBox.

To define the **LoadTasks** procedure, add the code shown in Listing 18-7 to your form module. The heart of this procedure is the For loop located near its bottom. This loop iterates through all of the tasks stored within the myTasks collection, adding the **Subject** property of each task to the ListBox. Remember, myTasks is a collection of tasks. Each item in this collection is a Task object, with all of its properties and methods.

Listing 18-7. The LoadContacts Procedure

```
Sub LoadTasks()
  Dim intCount As Integer
  Dim myTask As Task

' First, make sure that the list box is empty.
  lstTasks.Items.Clear()
```

```
' Next, load the tasks into the list box.
  For intCount = 0 To myTasks.Count - 1
    myTask = myTasks.Item(intCount)
    lstTasks.Items.Add(myTask.Subject)
  Next

End Sub
```

Step 7: Displaying a Task

The process of displaying a task is easy because of the functionality provided through the Pocket Outlook Object Model. Calling the **Display** method of the Task object results in the task being displayed using the default task interface.

To display a task, add the code shown in Listing 18-8 to the **Click** event procedure of the **Display** button.

Listing 18-8. Displaying a Task

```
Private Sub btnDisplay_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles btnDisplay.Click

  Dim myTask As Task

' Display the selected task.
  If (lstTasks.SelectedIndex <> -1) Then
    myTask = myTasks.Item(lstTasks.SelectedIndex)
    myTask.Display()
  End If

End Sub
```

Earlier in this tutorial, I mentioned the use of a collection to hold a list of Task objects. This is where that approach pays off. Since the collection myTasks matches the tasks displayed in the ListBox in a one-to-one relationship, it's easy to display a single task. All that you need to do is to create an instance of the task and then call the **Display** method of that task.

Step 8: Adding a Task

Adding a task is a three-step process. First, you need to create the task. Second, you configure the properties of the task. Third, you save the task.

In this tutorial, the task that is added is predefined, which is to say that the user has no input in the matter. Insert the code shown in Listing 18-9 into the **Click** event of the **Add** button.

Listing 18-9. Adding a Task

```
Private Sub btnAdd_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles btnAdd.Click

  Dim myTask As Task

' Create a new task.
  myTask = poApplication.CreateTask

' Configure the task.
  With myTask
    .Body = "This is a sample task."
    .Categories = "demo"
    .DueDate = Today.Date
    .Importance = Importance.High
    .ReminderOptions = ReminderOptions.Dialog
    .ReminderSet = True
    .StartDate = Today.Date
    .Subject = "sample task"

' Finally, save the task.
    .Save()
  End With

' Let the user know that the task was added.
  MessageBox.Show("task added...")

End Sub
```

Step 9: Loading a List of Select Tasks

The last feature that you're going to add to this application is the ability to select a subset of the tasks list—in this case, all of the tasks from a specific category.

Add the code shown in Listing 18-10 to the **Click** event of the **Select** button. At the heart of this procedure are two steps—the building of the selection string and then the use of this string with the **Restrict** method. The result is the creation of a collection of tasks that match the desired criteria.

Listing 18-10. Selecting a Subset of the Tasks

```
Private Sub btnSelect_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles btnSelect.Click

  Dim strQuery As String

' Retrieve the selected tasks.
  strQuery = "[Categories] = " & ControlChars.Quote & txtCategory.Text & _
    ControlChars.Quote
  myTasks = poApplication.Tasks.Items.Restrict(strQuery)

' Load the list of tasks.
  LoadTasks()

End Sub
```

Step 10: Testing the Application

Finally, you're ready to test your application. To begin, you need to copy the application to your target device by performing the following steps:

1. Select either **Pocket PC Device** or **Pocket PC Emulator** from the Deployment Device combo box.

2. Select **Release** from the Solution Configurations combo box.

3. Click the **Start** button.

Your application copies to the target device along with the PocketOutlook component. Upon completion, the application starts.

To verify the functionality of your application, perform the following steps:

1. Tap the **Load** button. After a brief pause, the ListBox is loaded with all of the tasks that are resident on your device.

2. Select a task from the ListBox. Tap the **Display** button. The selected task displays in the default Tasks interface. Close the selected task.

3. Tap the **Add** button. A message box displays confirming the addition of the task. To verify this addition, tap the **Load** button. The task named "sample task" should appear in the ListBox.

4. Enter a category into the TextBox (you can always use the "demo" category, which was added in the previous step). Tap the **Select** button. All of the tasks that are within this category are loaded into the ListBox.

NOTE *The availability of tasks for a specific category is dependent upon the tasks resident on your test device.*

HOMEWORK *Add a button to this application that will modify an existing task. The task to modify is the one selected within the ListBox.*

Next, we'll turn our attention to accessing, creating, and modifying contacts through the Pocket Outlook Object Model.

Working with Contacts

The second of the trifecta of Pocket Outlook data sources is contacts. Contacts offer developers a variety of application opportunities. They provide an easy way to incorporate contact data into a mobile application without all of the overhead of having to roll your own contact code.

As with tasks, contacts provide a **Categories** property, which is useful in identifying contacts that apply specifically to your application. Contacts are accessed through the **Contacts** property and the Contact object of the Pocket Outlook .NET component. Each of these items is covered in the following section.

The Contacts Property

The **Contacts** property of the OutlookApplication object provides access to a collection of contacts that are resident on a device. This property provides an interface to the **Contacts** folder within POOM, within which resides the contacts.

Several of the examples presented in this section demonstrate the use of the **Contacts** property along with its Items collection in working with contacts.

The Contact Object

All work with contacts themselves is handled through the Contact object. Commonly used properties of this object are shown in Table 18-3. Frequently used methods of the Contact object are shown in Table 18-4.

Table 18-3. Commonly Used Properties of the Contact Object

PROPERTY	DESCRIPTION
Body	The notes accompanying a contact
BusinessAddressCity	The city portion of the contact's address
BusinessAddressState	The state portion of the contact's address
BusinessAddressStreet	The street portion of the contact's address
BusinessFaxNumber	The fax number for the contact
BusinessTelephoneNumber	The business number for the contact
Categories	The categories for which the contact is assigned
CompanyName	The name of the contact's company
EmailAddress1	The e-mail address for the contact
FileAs	How the contact is filed (typically last name first)
MobileTelephoneNumber	The contact's cell phone number

NOTE *This is only a small subset of the properties provided by the Contact object. The properties detailed in Table 18-3 are the ones that I've found to be the most useful within a normal mobile business application. Your applications may benefit from the use of the remaining properties and as such, you should familiarize yourself with the complete property list through the Pocket Outlook .NET documentation.*

Table 18-4. Frequently Used Methods of the Contact Object

METHOD	DESCRIPTION
Copy	Creates a copy of an existing contact
Delete	Deletes a contact
Display	Displays a contact using the native Contact interface
Save	Saves modifications to a contact

While you will find both of these tables are useful for reference, the following sections, along with the examples, provide a quick-start approach to incorporating contact data into your applications.

Retrieving All Contacts

While it's unusual that your application will want to retrieve a list of all contacts (that is, unless you're creating a Contacts application knockoff), I'm still going to start this section by showing you how this is accomplished. As with tasks, you can retrieve contacts in several ways:

- As a collection of all contacts
- As a subset of all contacts
- As a single contact

Listing 18-11 demonstrates how to retrieve all contacts. Please note that, depending upon the number of contacts you have resident on a device, this could result in a sizeable collection.

Listing 18-11. Retrieving All of the Contacts

```
Imports InTheHand.PocketOutlook

[at the module level]
Dim poApplication As New OutlookApplication
Dim myContacts As OutlookItemCollection

Private Sub btnLoad_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles btnLoad.Click

' Retrieve all of the contacts.
  myContacts = poApplication.Contacts.Items

End Sub
```

The key parts of this example are as follows:

- The Imports statement, which must be located at the top of a module.
- The declaration statement for the OutlookApplication object.
- The declaration of the OutlookItemCollection object, which holds a collection of any Outlook items. In this example, it's a collection of contacts.
- Loading the collection of contacts from the OutlookApplication object, poApplication, into the OutlookItemCollection, myContacts.

At this point, the collection myContacts contains a set of contact objects, one for each contact that resides on the test device. As with any collection, you can loop through the collection to access individual contacts and view specific contact information.

Retrieving a Range of Contacts

With contacts, it is more common that you will need to retrieve a subset of all contacts, rather than retrieve all contacts. This subset might be comprised of only those contacts that begin with the letter *A,* or even as restrictive as a single contact.

Retrieving a subset of contacts is easily accomplished with the use of the **Restrict** method of the Items collection. Listing 18-12 demonstrates the use of this method. In this example, only contacts that begin with the letter *A* are returned.

Listing 18-12. Retrieving a Range of Contacts

```
Imports InTheHand.PocketOutlook

[at the module level]
Dim poApplication As New OutlookApplication
Dim myContacts As OutlookItemCollection

Private Sub btnSelect_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles btnSelect.Click

  Dim strA As String = "A"
  Dim strB As String = "B"
  Dim strQuery As String

' Retrieve contacts that begin with the letter A.
  strQuery = "[FileAs] >= " & ControlChars.Quote & strA & _
    ControlChars.Quote
  strQuery = strQuery & " AND [FileAs] < " & strB & _
     cmbPrefix.Items(cmbPrefix.SelectedIndex + 1) & ControlChars.Quote

  myContacts = poApplication.Contacts.Items.Restrict(strQuery)

End Sub
```

Much of the preparatory work is identical to that required to retrieve all contacts. You still need the Imports statement and to declare both the OutlookApplication and OutlookItemCollection objects.

The interesting code appears at the bottom of Listing 18-12. Here you build a query string, which restricts the contacts returned to only those that begin with the letter *A*. Applying this string to the **Restrict** method provides just the contacts that you're interested in.

NOTE *You can restrict contacts using any of the properties provided through the Contact object. While typically this will be the* ***FileAs*** *property, you are by no means limited to it alone.*

Displaying a Contact

As you have already seen with tasks, POOM provides the ability to easily access and leverage the native application interface of a data type through your application. What users see is a contact, displayed in the typical contact interface.

This functionality is provided through the **Display** method of the Contact object. Listing 18-13 shows an example of this technique.

Listing 18-13. Displaying a Contact

```
Imports InTheHand.PocketOutlook

[at the module level]
Dim poApplication As New OutlookApplication
Dim myContacts As OutlookItemCollection

Private Sub btnDisplay_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles btnDisplay.Click

  Dim myContact As Contact

' Retrieve all of the contacts.
  myContacts = poApplication.Contacts.Items

' Display the first contact.
  myContact = myContacts.Item(0)
  myContact.Display()

End Sub
```

As with the two previous contact examples, you need to add the Imports statement and declaration of the variables for both OutlookApplication and OutlookItemCollection to your module.

At the bottom of Listing 18-13 you'll see that all of the contacts are first retrieved. You could use a subset of the contacts or even a single contact as the starting point. I've chosen to demonstrate all contacts here because it's the easiest to understand.

From this collection of contacts, you extract a single contact, the first, into a Contact object of its own. It's through this Contact object that you gain access to the **Display** method, which in turn is called to display the selected contact.

Adding a Contact

Adding a contact is a three-step process. First, you need to create a new contact. Second, you need to configure the contact. Third, you need to save the contact. Listing 18-14 demonstrates adding a contact.

Listing 18-14. Adding a Contact

```
Imports InTheHand.PocketOutlook

[at the module level]
Dim poApplication As New OutlookApplication

Private Sub btnAdd_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles btnAdd.Click

  Dim myContact As Contact

' Create a new contact.
  myContact = poApplication.CreateContact

' Configure the contact.
  With myContact
    .Birthday = CDate("01/01/1960")
    .Body = "This is a sample contact."
    .BusinessTelephoneNumber = "888-555-0001"
    .Categories = "demo"
    .CompanyName = "Acme"
    .Email1Address = "joe.acme@acme.com"
    .FileAs = "Acme, Joe"
    .FirstName = "Joe"
    .LastName = "Acme"
    .Title = "President"
```

```
' Finally, save the contact.
    .Save()
  End With

End Sub
```

As with all of the previous contact examples, you need to start by adding the Imports statement and declaring the variables for OutlookApplication and OutlookItemCollection.

At the bottom of Listing 18-14 you'll see the three steps I described. First, the **CreateContact** method of the OutlookApplication object is called to create your new contact. Second, you configure the properties of the new contact. Third, the Contact object's **Save** method is called to save the contact.

Modifying a Contact

Modifying a contact is similar to adding a contact, only instead of creating a new contact you load a Contact object from an existing contact. Listing 18-15 demonstrates this process.

Listing 18-15. Modifying an Existing Contact

```
Imports InTheHand.PocketOutlook

[at the module level]
Dim poApplication As New OutlookApplication
Dim myContacts As OutlookItemCollection

Private Sub btnModify_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles btnModify.Click

  Dim myContact As Contact

' Retrieve all of the contacts.
  myContacts = poApplication.Contacts.Items

' Modify the first contact.
  myContact = myContacts.Item(0)
  With myContact
    .BusinessTelephoneNumber = "888-555-0001"
    .Save()
  End With

End Sub
```

As with all of the previous contact examples, you start by adding an Imports statement and declaring the variables for OutlookApplication and OutlookItemCollection. At the bottom of Listing 18-15, you'll find where the contact is first retrieved. In this example, all of the contacts are retrieved, but only the first contact is loaded into the Contact object. From this point, you are free to modify any of the Contact object's properties. You finish the modification process by calling the Contact object's **Save** method.

Now that you've seen the basics of working with contacts, let's put all of these techniques together into a comprehensive example.

Step-by-Step Tutorial: Working with Contacts

In this step-by-step exercise, you'll create an application that demonstrates working with contacts within Pocket Outlook. As part of this exercise, you'll

- Connect to Pocket Outlook.
- Retrieve a list of all contacts.
- Retrieve a list of contacts based upon the first letter of their name.
- Display a contact.
- Add a contact.

This application provides all of the fundamentals of working with contacts within Pocket Outlook.

NOTE *To complete this tutorial you'll need the PocketOutlook component from InTheHand at* `http://www.inthehand.com`.

NOTE *I provide a completed version of this application, titled Contacts - Complete, under the* ***Chapter 18*** *folder of the* ***Samples*** *folder for this book. See Appendix D for more information on accessing and loading the sample applications.*

Step 1: Opening the Project

To simplify this tutorial, I've already created the project and the user interface for the Contacts Tutorial application. This template project is included under the **Chapter 18** folder in the **Samples** folder. To load this project, follow these steps:

1. From the VS .NET IDE Start Page, select to open a project. The Open Project dialog box will be displayed.

2. Use this dialog box to navigate to the **Chapter 18** folder under the **Samples** folder for this book.

3. Select and open the project **Contacts**. The project will be loaded onto your computer.

Step 2: Examining the User Interface

The user interface for the Contacts Tutorial application is comprised of several controls: a ListBox, four Buttons, and a ComboBox. Figure 18-3 shows the application's interface.

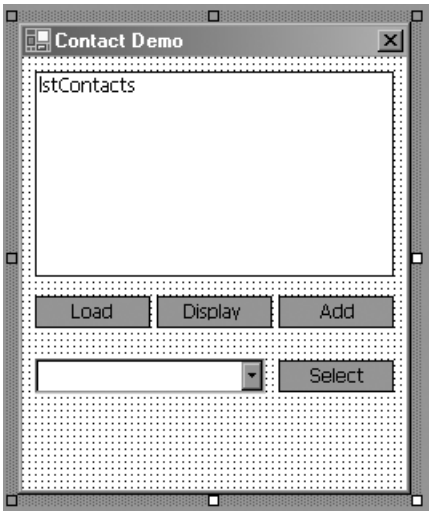

Figure 18-3. The Contacts application interface

The ListBox will display available contacts.

The four buttons for the Contacts Tutorial application function as follows:

- The **Load** button triggers loading a list of all contacts into the ListBox.
- The **Display** button triggers displaying the selected contact.
- The **Add** button adds a new contact with predefined values.
- The **Select** button retrieves a list of contacts that begin with a specific letter.

The ComboBox provides a list from which the user may select an alphabetical subset of the contacts. The ComboBox has been preloaded with all of the letters of the alphabet.

Step 3: Adding a Reference to PocketOutlook

You need to add a single reference to your application to enable you to work with the Pocket Outlook Object Model from a .NET Compact Framework application.

To add this reference, perform the following steps:

1. In the Solution Explorer window, right-click the **References** folder. A pop-up menu displays.
2. From the menu, select **Add Reference**.
3. The Add Reference dialog box displays. Select the **InTheHand. PocketOutlook** component.
4. Click the **OK** button to add the selected component to your project.

Step 4: Adding the Imports Statement

The first line of code you'll be adding imports the **PocketOutlook** namespace. This allows you to reference and work with the PocketOutlook elements within your code without having to fully qualify each element.

To import the **PocketOutlook** namespace, perform the following steps:

1. Open the code window for the form.
2. At the top of the module, above the line that declares the **Form** class, add the following line of code:

```
Imports InTheHand.PocketOutlook
```

Step 5: Declaring Module-Level Objects

This tutorial uses two module-level object variables. These variables hold instances of the PocketOutlook OutlookApplication and OutlookItemCollection objects.

To define these variables, add the following code to the module level of the form:

```
Dim poApplication As New OutlookApplication
Dim myContacts As OutlookItemCollection
```

Step 6: Loading a List of All Contacts

The first functionality that you'll add to the application is to display a list of all contacts. Obtaining this list is simple. Loading the list into your ListBox requires nothing more than a For loop for running through the collection.

The first part of this step obtains a list of contacts. While you could simply reference the Contacts collection of the OutlookApplication object, you are instead going to retrieve a copy of the Contacts collection into a collection of its own. Having a collection of contacts that matches those contacts displayed in your ListBox makes it easier to display the details of an individual contact. You'll see more on this later in the tutorial.

The code required to retrieve the Contacts collection is shown in Listing 18-16. Add this code to the **Click** event procedure of the **Load** button. In this procedure, you create a copy of the Contacts collection through the OutlookApplication. Contacts.Items collection.

Listing 18-16. Retrieving a List of All Contacts

```
Private Sub btnLoad_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles btnLoad.Click

' Store the collection of contacts for future use.
  myContacts = poApplication.Contacts.Items

' Load the list of contacts.
  LoadContacts()

End Sub
```

Displaying the list of contacts is triggered at the bottom of the procedure shown in Listing 18-16. The **LoadContacts** procedure is a general-purpose procedure that displays the contents of the myContacts collection in a ListBox.

To define the **LoadContacts** procedure, add the code shown in Listing 18-17 to your form module. The heart of this procedure is the For loop located near its bottom. This loop iterates through all of the contacts stored within the myContacts collection, adding the **FileAs** property of each contact to the ListBox. Remember, myContacts is a collection of contacts. Each item in this collection is a Contact object, with all of its properties and methods.

Listing 18-17. The LoadContacts Procedure

```
Sub LoadContacts()
  Dim intCount As Integer
  Dim myContact As Contact

' First, make sure that the list box is empty.
  lstContacts.Items.Clear()

' Next, load the contacts into the list box.
  For intCount = 0 To myContacts.Count - 1
    myContact = myContacts.Item(intCount)
    lstContacts.Items.Add(myContact.FileAs)
  Next

End Sub
```

Step 7: Displaying a Contact

The process of displaying a contact is easy because of the functionality provided through the Pocket Outlook Object Model. Calling the **Display** method of the Contact object results in the contact being displayed using the default contact interface.

To display a contact, add the code shown in Listing 18-18 to the **Click** event procedure of the **Display** button.

Listing 18-18. Displaying a Contact

```
Private Sub btnDisplay_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles btnDisplay.Click

  Dim myContact As Contact

' Display the selected contact.
```

```
  If (lstContacts.SelectedIndex <> -1) Then
    myContact = myContacts.Item(lstContacts.SelectedIndex)
    myContact.Display()
  End If

End Sub
```

Earlier in this tutorial, I mentioned using a collection to hold a list of Contact objects. This is where that approach pays off. Since your collection myContacts matches the contacts displayed in the ListBox in a one-to-one relationship, it's easy to display a single contact. All that you need to do is to create an instance of the contact and then call the **Display** method of that contact.

Step 8: Adding a Contact

Adding a contact is a three-step process. First, you need to create the contact. Second, you configure the properties of the contact. Third, you save the contact.

In this tutorial, the contact that is added is predefined, which is to say that the user has no input in the matter. Insert the code shown in Listing 18-19 into the **Click** event of the **Add** button.

Listing 18-19. Adding a Contact

```
Private Sub btnAdd_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles btnAdd.Click

  Dim myContact As Contact

' Create a new contact.
  myContact = poApplication.CreateContact

' Configure the contact.
  With myContact
    .Birthday = CDate("01/01/1960")
    .Body = "This is a sample contact."
    .BusinessTelephoneNumber = "888-555-0001"
    .Categories = "demo"
    .CompanyName = "Acme"
    .Email1Address = "joe.acme@acme.com"
    .FileAs = "Acme, Joe"
```

```
    .FirstName = "Joe"
    .LastName = "Acme"
    .Title = "President"

' Finally, save the contact.
    .Save()
  End With

' Let the user know that the contact was added.
  MessageBox.Show("contact added...")

End Sub
```

Step 9: Loading a List of Select Contacts

The last feature that you're going to add to this application is the ability to select a subset of the contacts list. In this case, you're going to select all of the contacts that begin with a specific letter—for example, only those contacts that begin with the letter *A*.

Add the code shown in Listing 18-20 to the **Click** event of the **Select** button. At the heart of this procedure are two steps—the building of the selection string and then the use of this string with the **Restrict** method. The result is the creation of a collection of contacts that match the desired criteria.

Listing 18-20. Selecting a Subset of the Contacts

```
Private Sub btnSelect_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles btnSelect.Click

  Dim strQuery As String

' Retrieve the selected contacts.
  strQuery = "[FileAs] >= " & ControlChars.Quote & cmbPrefix.Text & _
    ControlChars.Quote

' Use a range for anything other than contacts beginning with the letter "Z".
  If (cmbPrefix.Text < "Z") Then
    strQuery = strQuery & " AND [FileAs] < " & ControlChars.Quote & _
      cmbPrefix.Items(cmbPrefix.SelectedIndex + 1) & ControlChars.Quote
  End If
```

```
    myContacts = poApplication.Contacts.Items.Restrict(strQuery)

' Load the list of contacts.
    LoadContacts()

End Sub
```

Step 10: Testing the Application

Finally, you're ready to test your application. To begin, you need to copy the application to your target device by performing the following steps:

1. Select either **Pocket PC Device** or **Pocket PC Emulator** from the Deployment Device combo box.

2. Select **Release** from the Solution Configurations combo box.

3. Click the **Start** button.

Your application copies to the target device along with the PocketOutlook component. Upon completion the application starts.

To verify the functionality of your application, perform the following steps:

1. Tap the **Load** button. After a brief pause, the ListBox is loaded with all of the contacts that are resident on your device.

2. Select a contact from the ListBox. Tap the **Display** button. The selected contact displays in the default Contacts interface. Close the selected contact.

3. Tap the **Add** button. A message box displays, confirming the addition of the contact. To verify this addition, tap the **Load** button. The contact for "Acme, Joe" should appear in the ListBox.

4. Select a letter from the ComboBox. Tap the **Select** button. All of the contacts that begin with the selected letter are loaded into the ListBox.

NOTE *The availability of contacts that begin with a specific letter is dependent upon the contacts resident on your test device.*

HOMEWORK *Add a button to this application that will modify an existing contact. The contact to modify is the one selected within the ListBox.*

Next, we'll turn our attention to accessing, creating, and modifying appointments through the Pocket Outlook Object Model.

Working with Appointments

The last piece of Pocket Outlook data we have to examine is appointments. As with tasks and contacts, the Pocket Outlook .NET component makes working with appointments an easy task.

From the development standpoint, appointments offer a powerful way to integrate mobile applications with back-end scheduling systems. From your application, through Pocket Outlook, ActiveSync, Outlook, and finally an enterprise server, you have a seamless path for the delivery of appointment-related data. This functionality can be used to schedule maintenance, meetings, follow-ups, or any task that is date and time specific.

Appointment functionality is exposed to the .NET Compact Framework through the Pocket Outlook .NET component from InTheHand. This component's **Appointments** property and Appointment object are the subjects to this section.

The Appointments Property

The **Appointments** property of the OutlookApplication object provides access to the collection of appointments resident on a device. In actuality, the **Appointments** property provides an interface to the **Appointments** folder. The underlying Pocket Outlook Object Model supplies access to this folder. This property links through to an Items collection that contains the actual appointments. This section includes several examples of the **Appointments** property along with its Items collection.

The Appointment Object

Individual appointments are created and modified through the Appointment object. Commonly used properties and methods for this object are shown in Tables 18-5 and 18-6 respectively.

Table 18-5. Commonly Used Properties of the Appointment Object

PROPERTY	DESCRIPTION
AllDayEvent	Specifies whether this is an all-day event
Body	Defines the notes accompanying an appointment
Categories	Specifies the categories for which this appointment is assigned
Duration	Indicates the length of the appointment
End	Specifies when the appointment ends
Location	Indicates where the appointment takes place
ReminderSet	Specifies whether a reminder is configured for an appointment
Start	Indicates when the appointment starts
Subject	Indicates the subject of an appointment

Table 18-6. Commonly Used Methods of the Appointment Object

METHOD	DESCRIPTION
Cancel	Sends a cancellation of a meeting request
Copy	Creates a copy of an existing appointment
Delete	Deletes an appointment
Display	Displays an appointment using the native Appointment interface
Save	Saves modifications made to an appointment
Send	Sends a meeting request to the recipient list for an appointment

While these tables serve well for reference purposes, a set of practical demonstrations follow that provide detailed examples of commonly performed appointment-related operations.

Retrieving All Appointments

Whether or not you retrieve all appointments is going to be dependent upon your applications. Typically, you won't, instead opting for a range of dates under which the appointments fall. As with tasks and contacts, there are several ways that you can retrieve appointments:

- As a collection of all appointments
- As a subset of all appointments
- As an individual appointment

We'll start by looking at how to retrieve all appointments. It's the simplest method, and while not the most commonly used approach, it's still utilized within mobile applications. Listing 18-21 demonstrates this process.

Listing 18-21. Retrieving All of the Appointments

```
Imports InTheHand.PocketOutlook

[at the module level]
Dim poApplication As New OutlookApplication
Dim myAppointments As OutlookItemCollection

Private Sub btnLoad_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles btnLoad.Click

' Retrieve all of the appointments.
  myAppointments = poApplication.Appointments.Items

End Sub
```

The keys to this example are as follows:

- The Imports statement, which must be located at the top of a module.
- The declaration statement for the OutlookApplication object.
- The declaration of the OutlookItemCollection object, which can hold a collection of any Outlook items. In this case, it will hold a collection of appointments.
- Loading the collection of appointments from the OutlookApplication object, poApplication, into the OutlookItemCollection, myAppointments.

At this stage, the collection myAppointments contains a set of appointment objects, one for each appointment that resides on the device. You can loop through this collection to view information on specific appointments.

Retrieving Appointments for a Given Date

While you may at times retrieve a list of all appointments, more commonly your application will want only those appointments that fall within a certain range of dates.

Listing 18-22 demonstrates retrieving a subset of appointments. First, appointments for today are grabbed. Next, appointments for tomorrow are retrieved. Finally, how to select appointments for the next week is shown.

Listing 18-22. Retrieving Appointments for Specific Dates

```
Imports InTheHand.PocketOutlook

[at the module level]
Dim poApplication As New OutlookApplication
Dim myAppointmens As OutlookItemCollection

Private Sub btnSelect_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles btnSelect.Click

  Dim strQuery As String
  Dim tmpDate As Date

' Retrieve appointments for today.
  strQuery = "[Start] = " & _
    ControlChars.Quote & Date.Today.ToShortDateString & _
    ControlChars.Quote
  myAppointments = poApplication.Calendar.Items.Restrict(strQuery)

' Retrieve appointments for tomorrow.
  tmpDate = Date.Today.AddDays(1)
  strQuery = "[Start] = " & _
    ControlChars.Quote & tmpDate.Date.ToShortDateString & ControlChars.Quote
  myAppointments = poApplication.Calendar.Items.Restrict(strQuery)

' Retrieve appointments for this next week.
  tmpDate = Date.Today.AddDays(7)
  strQuery = "[Start] >= " & _
    ControlChars.Quote & Date.Today.ToShortDateString & ControlChars.Quote
  strQuery = strQuery & " AND [Start] < " & ControlChars.Quote & _
    tmpDate.Date.ToShortDateString & ControlChars.Quote
  myAppointments = poApplication.Calendar.Items.Restrict(strQuery)

End Sub
```

The preparatory work for this example is identical to that required by the previous example. You have to include the Imports statement and the declaration of both the variables for OutlookApplication and OutlookItemCollection.

Each of the sections of code that select appointments for today, tomorrow, and the next week make use of the **Restrict** method to retrieve the appropriate data. As you can see, the key to these Restrict statements is some creative date manipulations.

NOTE *While this example focuses on restricting appointments by their start date, you can in fact restrict appointments based upon the values of any of their properties.*

Displaying an Appointment

As you've already seen with tasks and contacts, POOM provides you with the ability to access the native application interface of its data types through your application. What users will see is an appointment, displayed in the Pocket PC appointment interface.

This functionality is provided by the **Display** method of the Appointment object. Listing 18-23 shows an example of this technique.

Listing 18-23. Displaying an Appointment

```
Imports InTheHand.PocketOutlook

[at the module level]
Dim poApplication As New OutlookApplication
Dim myAppointments As OutlookItemCollection

Private Sub btnDisplay_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles btnDisplay.Click

  Dim myAppointment As Appointment

' Retrieve all of the appointments.
  myAppointments = poApplication.Appointments.Items

' Display the first appointment.
  myAppointment = myAppointments.Item(0)
  myAppointment.Display()

End Sub
```

As with both of the examples involving retrieving appointments, you need to add to your application declarations for the Imports statement and the variables for both OutlookApplication and OutlookItemCollection.

At the bottom of Listing 18-23 you'll see the code used to display an appointment. First, a collection of all appointments is retrieved. Next, a single appointment is selected, in this case the first appointment, and this appointment is used to create an Appointment object. It's through this Appointment object that you gain access to the **Display** method, which causes the appointment to be displayed.

Adding an Appointment

Adding an appointment is a three-step process. First, you need to create a new appointment. Second, you need to configure the appointment. Third, you need to save the appointment. Listing 18-24 demonstrates this process.

Listing 18-24. Adding an Appointment

```
Imports InTheHand.PocketOutlook

[at the module level]
Dim poApplication As New OutlookApplication

Private Sub btnAdd_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles btnAdd.Click

  Dim myAppointment As Appointment

' Create a new appointment.
  myAppointment = poApplication.CreateAppointment

' Configure the appointment.
  With myAppointment
    .Body = "This is a sample appointment."
    .Categories = "demo"
    .End = DateAdd(DateInterval.Hour, 1, Now)
    .Location = "New York"
    .Start = Now
    .Subject = "demo appointment"

' Finally, save the appointment.
    .Save()
  End With

End Sub
```

As with the previous appointment examples, you start by adding the Imports statement and declaring the variables for OutlookApplication and OutlookItemCollection.

At the bottom of Listing 18-24 is the code that performs the three steps required to add an appointment. First, a call is made to the **CreateAppointment** method of the OutlookApplication object. Second, the properties of the new appointment are configured. Finally, the Appointment object's **Save** method is called to save the appointment.

Modifying an Appointment

The process of modifying an appointment is similar to adding an appointment, only instead of creating a new appointment, you load an existing appointment into an Appointment object. Listing 18-25 demonstrates this process.

Listing 18-25. Modifying an Existing Appointment

```
Imports InTheHand.PocketOutlook

[at the module level]
Dim poApplication As New OutlookApplication
Dim myAppointments As OutlookItemCollection

Private Sub btnModify_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles btnModify.Click

  Dim myAppointment As Appointment

' Retrieve all of the appointment.
  myAppointments = poApplication.Appointments.Items

' Modify the first appointment.
  myAppointment = myAppointments.Item(0)
  With myAppointment
    .Body = "This is updated content."
    .Save()
  End With

End Sub
```

As with all of the previous appointment examples, you start by adding an Imports statement and declaring the variables for OutlookApplication and OutlookItemCollection. At the bottom of Listing 18-25, you'll find where the

appointment is first retrieved. In this example, all of the appointments are retrieved, but only the first appointment is loaded into the Appointment object. From this point, you're free to modify any of the Appointment object's properties. You finish the modification process by calling the Appointment object's **Save** method.

Now that you've seen the basics of working with appointments, let's put all of these techniques together into a comprehensive example.

Step-by-Step Tutorial: Working with Appointments

In this step-by-step exercise, you'll create an application that demonstrates working with appointments within Pocket Outlook. As part of this exercise, you'll

- Connect to Pocket Outlook.
- Retrieve a list of all appointments.
- Retrieve a list of appointments for today, tomorrow, and the next week.
- Display an appointment.
- Add an appointment.

This application provides all of the fundamentals of working with appointments within Pocket Outlook.

NOTE *To complete this tutorial, you'll need the PocketOutlook component from InTheHand, available at* `http://www.inthehand.com`.

NOTE *I provide a completed version of this application, titled Appointments - Complete, under the* ***Chapter 18*** *folder of the* ***Samples*** *folder for this book. See Appendix D for more information on accessing and loading the sample applications.*

Step 1: Opening the Project

To simplify this tutorial, I've already created the project and the user interface for the Appointments Tutorial application. This template project is included

under the **Chapter 18** folder in the **Samples** folder. To load this project, follow these steps:

1. From the VS .NET IDE Start Page, select to open a project. The Open Project dialog box will be displayed.

2. Use this dialog box to navigate to the **Chapter 18** folder under the **Samples** folder for this book.

3. Select and open the project **Appointments**. The project will be loaded onto your computer.

Step 2: Examining the User Interface

The user interface for the Appointments Tutorial application is comprised of several controls: a ListBox, four Buttons, and a ComboBox. Figure 18-4 shows the application's interface.

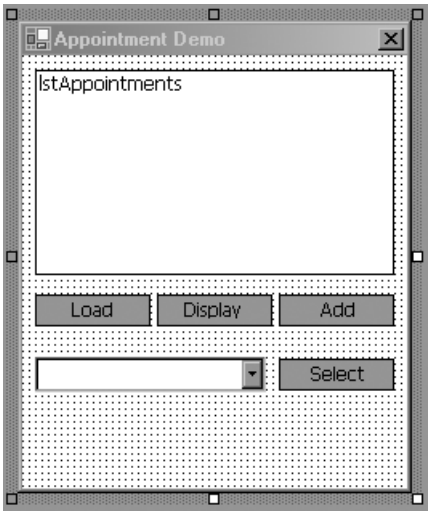

Figure 18-4. The Appointments application interface

The ListBox will display available appointments.

The four buttons for the Appointments Tutorial application function as follows:

- The **Load** button triggers loading a list of all appointments into the ListBox.
- The **Display** button triggers displaying the selected appointment.
- The **Add** button adds a new appointment with predefined values.
- The **Select** button retrieves a list of appointments for today, tomorrow, or the next week.

The ComboBox provides three time-range options from which to choose: today, tomorrow, and next week. These values have been preloaded.

Step 3: Adding a Reference to PocketOutlook

You need to add a single reference to your application to enable you to work with the Pocket Outlook Object Model from a .NET Compact Framework application.

To add this reference, perform the following steps:

1. In the Solution Explorer window, right-click the **References** folder. A pop-up menu displays.
2. From the menu, select **Add Reference**.
3. The Add Reference dialog box displays. Select the **InTheHand.PocketOutlook** component.
4. Click the **OK** button to add the selected component to your project.

Step 4: Adding the Imports Statement

The first line of code you'll be adding imports the **PocketOutlook** namespace. This allows you to reference and work with the PocketOutlook elements within your code without having to fully qualify each element.

To import the **PocketOutlook** namespace, perform the following steps:

1. Open the code window for the form.
2. At the top of the module, above the line that declares the **Form** class, add the following line of code:

```
Imports InTheHand.PocketOutlook
```

Step 5: Declaring Module-Level Objects

This tutorial uses two module-level object variables. These variables hold instances of the PocketOutlook OutlookApplication and OutlookItemCollection objects.

To define these variables, add the following code to the module level of the form:

```
Dim poApplication As New OutlookApplication
Dim myAppointments As OutlookItemCollection
```

Step 6: Loading a List of All Appointments

The first functionality that you'll add to the application is to display a list of all appointments. Obtaining this list is simple. Loading the list into your ListBox requires nothing more than a For loop for running through the collection.

The first part of this step obtains a list of appointments. While you could simply reference the Appointments collection of the OutlookApplication object, you are instead going to retrieve a copy of the Appointments collection into a collection of its own. Having a collection of appointments that matches those appointments displayed in your ListBox makes it easier to display the details of an individual appointment. You'll see more on this later in the tutorial.

The code required to retrieve the Appointments collection is shown in Listing 18-26. Add this code to the **Click** event procedure of the **Load** button. In this procedure, you create a copy of the Appointments collection through the OutlookApplication.Appointments.Items collection.

Listing 18-26. Retrieving a List of All Appointments

```
Private Sub btnLoad_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles btnLoad.Click

' Store the collection of appointments for future use.
  myAppointments = poApplication.Calendar.Items

' Load the list of appointments.
  LoadAppointments()

End Sub
```

Displaying the list of appointments is triggered at the bottom of the procedure shown in Listing 18-26. The **LoadAppointments** procedure is a general-purpose

procedure that displays the contents of the myAppointments collection in a ListBox.

To define the **LoadAppointments** procedure, add the code shown in Listing 18-27 to your form module. The heart of this procedure is the For loop located near its bottom. This loop iterates through all of the appointments stored within the myAppointments collection, adding the **Subject** property of each appointment to the ListBox. Remember, myAppointments is a collection of appointments. Each item in this collection is an Appointment object, with all of its properties and methods.

Listing 18-27. The LoadAppointments Procedure

```
Sub LoadAppointments()
  Dim intCount As Integer
  Dim myAppointment As Appointment

' First, make sure that the list box is empty.
  lstAppointments.Items.Clear()

' Next, load the appointments into the list box.
  For intCount = 0 To myAppointments.Count - 1
    myAppointment = myAppointments.Item(intCount)
    lstAppointments.Items.Add(myAppointment.Subject)
  Next

End Sub
```

Step 7: Displaying an Appointment

The process of displaying an appointment is easy because of the functionality provided through the Pocket Outlook Object Model. Calling the **Display** method of the Appointment object results in the appointment being displayed using the default appointment interface.

To display an appointment, add the code shown in Listing 18-28 to the **Click** event procedure of the **Display** button.

Listing 18-28. Displaying an Appointment

```
Private Sub btnDisplay_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles btnDisplay.Click

  Dim myAppointment As Appointment
```

```
' Display the selected appointment.
  If (lstAppointments.SelectedIndex <> -1) Then
    myAppointment = myAppointments.Item(lstAppointments.SelectedIndex)
    myAppointment.Display()
  End If

End Sub
```

Earlier in this tutorial, I mentioned using a collection to hold a list of Appointment objects. This is where that approach pays off. Since your collection myAppointments matches the appointments displayed in the ListBox in a one-to-one relationship, it's easy to display a single appointment. All that you need to do is to create an instance of the appointment and then call the **Display** method of that appointment.

Step 8: Adding an Appointment

Adding an appointment is a three-step process. First, you need to create the appointment. Second, you configure the properties of the appointment. Third, you save the appointment.

In this tutorial, the appointment that is added is predefined, which is to say that the user has no input in the matter. Insert the code shown in Listing 18-29 into the **Click** event of the **Add** button.

Listing 18-29. Adding an Appointment

```
Private Sub btnAdd_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles btnAdd.Click

  Dim myAppointment As Appointment

' Create a new appointment.
  myAppointment = poApplication.CreateAppointment

' Configure the appointment.
  With myAppointment
    .Body = "This is a sample appointment."
    .Categories = "demo"
    .End = DateAdd(DateInterval.Hour, 1, Now)
    .Location = "New York"
    .Start = Now
    .Subject = "demo appointment"
```

```
    ' Finally, save the appointment.
      .Save()
    End With

' Let the user know that the appointment was added.
    MessageBox.Show("appointment added...")

End Sub
```

The key point to bring away from this sample is the configuration of two properties, **Start** and **End**. In the case of this sample, you set the start of the appointment to the present time. The end of the appointment is set to 1 hour from now. Obviously, this is an impractical example. Why would anyone want to set an appointment to start right now? Still, you get the idea. The **Start** and **End** properties are pivotal items when it comes to appointments, and the date-related functionality provided through the .NET Compact Framework makes working with dates easy.

Step 9: Loading a List of Select Appointments

The last feature that you're going to add to this application is the ability to select appointments for either today, tomorrow, or the next week.

Add the code shown in Listing 18-30 to the **Click** event of the **Select** button. At the heart of this procedure are two steps—the building of the selection string and then the use of this string with the **Restrict** method. The result is the creation of a collection of appointments that match the desired criteria.

Listing 18-30. Selecting a Subset of the Appointments

```
Private Sub btnSelect_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles btnSelect.Click

  Dim strQuery As String
  Dim tmpDate As Date

' Retrieve the selected appointments.
  Select Case cmbDates.Text
    Case "today"
      strQuery = "[Start] = " & ControlChars.Quote & _
        Date.Today.ToShortDateString & ControlChars.Quote
```

```
    Case "tomorrow"
      tmpDate = Date.Today.AddDays(1)
      strQuery = "[Start] = " & ControlChars.Quote & _
        tmpDate.Date.ToShortDateString & ControlChars.Quote
    Case "next week"
      tmpDate = Date.Today.AddDays(7)
      strQuery = "[Start] >= " & ControlChars.Quote & _
        Date.Today.ToShortDateString & ControlChars.Quote
      strQuery = strQuery & " AND [Start] < " & ControlChars.Quote & _
        tmpDate.Date.ToShortDateString & ControlChars.Quote
  End Select

  myAppointments = poApplication.Calendar.Items.Restrict(strQuery)

' Load the list of appointments.
  LoadAppointments()

End Sub
```

Selecting appointments for today is the easiest. You simply need to set the criteria to the present date. Selecting appointments for tomorrow is only slightly more complicated. Some simple date math is used to add one day to the present date before performing the selection. Selecting the appointments for the next week is by far the most complicated of the three, and even then it's not rocket science. Here a compound select statement along with some date math is used to specify a range of dates for the selection.

Step 10: Testing the Application

Finally, you're ready to test your application. To begin, you need to copy the application to your target device by performing the following steps:

1. Select either **Pocket PC Device** or **Pocket PC Emulator** from the Deployment Device combo box.

2. Select **Release** from the Solution Configurations combo box.

3. Click the **Start** button.

Your application copies to the target device along with the PocketOutlook component. Upon completion the application starts.

To verify the functionality of your application, perform the following steps:

1. Tap the **Load** button. After a brief pause, the ListBox is loaded with all of the appointments that are resident on your device.

2. Select an appointment from the ListBox. Tap the **Display** button. The selected appointment displays in the default Appointment interface. Close the selected appointment.

3. Tap the **Add** button. A message box displays confirming the addition of the appointment. To verify this addition, tap the **Load** button. The appointment named "This is a sample appointment." should appear in the ListBox.

4. Select a date range option from the ComboBox. Tap the **Select** button. All of the appointments for that range are loaded into the ListBox.

NOTE *The availability of appointments that fall within a specific timeframe is dependent upon the appointments resident on your test device.*

HOMEWORK *Add a button to this application that will modify an existing appointment. The appointment to modify is the one selected within the ListBox.*

Summary

The Pocket Outlook Object Model is the gateway to Outlook-specific data. POOM is not only a fun word to say, but also a powerful resource for developers. POOM allows you to integrate seamlessly your mobile applications into the standard core applications found on every Pocket PC.

The Pocket Outlook .NET component from InTheHand makes working with POOM easy from within your NETCF applications. It offers a solid set of tools to deal with the most useful of Pocket Outlook data: tasks, contacts, and appointments. Rather than having to write your own native-language DLL, and spending months of development time, you can be up and working with Pocket Outlook data in a few hours.