CHAPTER 25

Automating Your Application with Visual Basic

Why Aren't We Using Macros? 1583
Assisting Data Entry 1585
Validating Complex Data 1604
Controlling Tabbing on a Multiple-Page Form 1613
Automating Data Selection 1615
Linking to Related Data in Another Form or Report . . 1631
Automating Complex Tasks 1639
Automating Reports 1649
Calling Named Data Macros 1658

Now that you've learned the fundamentals of using Microsoft Visual Basic, it's time to put this knowledge into practice. In this chapter, you'll learn how to create the Visual Basic code you need to automate many common tasks.

You can find dozens of examples of automation in the Conrad Systems Contacts, Housing Reservations, Back Office Software System, and Wedding List sample databases. As you explore the databases, whenever you see something interesting, open the form or report in Design view and take a look at the Visual Basic code behind the form or report. This chapter walks you through a few of the more interesting examples in these databases.

Note

You can find the code explained in this chapter in the Conrad Systems Contacts (Contacts.accdb), Housing Reservations (Housing.accdb), and Wedding List (WeddingList.accdb) sample applications on the companion CD.

Why Aren't We Using Macros?

Although you can certainly use user interface macros to automate applications, macros have certain limitations. For example, as you might have noticed when examining the list of available events in Chapter 19, "Understanding Event Processing," many events require or return parameters that can be passed to or read from a Visual Basic procedure but not a macro. And as you saw in Chapter 20, "Automating a Client Application Using Macros," and Chapter 21, "Automating a Web Application Using Macros," the debugging facilities for macros are not as robust as for Visual Basic.

When to Use Macros

Use macros in your application in any of the following circumstances:

- You are working with a web database.
- Your application consists of only a few forms and reports.
- You need to build a simple application that is automated using only trusted macro actions so that the application can run in an untrusted environment.
- Your application might be used by users unfamiliar with Visual Basic who will want to understand how your application is constructed and possibly modify or enhance it.
- You're developing an application prototype, and you want to rapidly automate a few features to demonstrate your design. However, once you understand Visual Basic, automating a demonstration application is just as easy using event procedures.
- You don't need to evaluate or set parameters passed by certain events, such as AfterDelConfirm, ApplyFilter, BeforeDelConfirm, Error, Filter, KeyDown, KeyPress, KeyUp, MouseDown, MouseMove, MouseUp, NotInList, and Updated.
- You don't need to open and work with recordsets or other objects.

When to Use Visual Basic

Although user interface macros can be useful and are necessary when working with web forms, a number of tasks cannot be carried out with macros, and there are others that are better implemented using a Visual Basic procedure. Use a Visual Basic procedure instead of a macro in any of the following circumstances:

- You need complex error handling in your application.
- You want to define a new function.
- You need to handle events that pass parameters or accept return values (other than Cancel).
- You need to create new objects (tables, queries, forms, or reports) in your database from application code.

- Your application needs to interact with another Windows-based program via ActiveX automation.
- You want to be able to directly call Windows application programming interface (API) functions.
- You want to define application code that is common across several applications in a library.
- You want to be able to open and work with data in a recordset on a record-by-record basis.
- You need to use some of the native facilities of the relational database management system that handles your attached tables (such as Microsoft SQL Server procedures or data definition facilities).
- You want maximum performance in your application. Because modules are compiled, they execute slightly faster than macros. You'll probably notice a difference only on slower processors.
- You are writing a complicated application that will be difficult to debug.

Assisting Data Entry

You can do a lot to help make sure the user of your application enters correct data by using data macros and by defining default values, input masks, and validation rules. But what can you do if the default values come from a related table? How can you assist a user who needs to enter a value that's not in the row source of a combo box? How do you make the display text in a hyperlink more readable? Is there a way you can make it easier for your user to pick dates and times? And how do you help the user edit linked picture files? You can find the answers to these questions in the following sections.

Filling In Related Data

The tblContactProducts table in the Conrad Systems Contacts database (Contacts.accdb) has a SoldPrice field that reflects the actual sales price at the time of a sale. The tblProducts table has a UnitPrice field that contains the normal selling price of the product. When the user is working in the Contacts form (frmContacts) and wants to sell a new product, you don't want the user to have to look up the current product price before entering it into the record.

You learned in Chapter 15, "Advanced Form Design," how to build a form with subforms nested two levels to edit contacts, the default company for each contact, and the products sold to that company and registered to the current contact. However, if you open frmContacts in the Contacts.accdb sample database and click the Products tab, as shown in Figure 25-1, you'll notice that there doesn't appear to be any linking company data between contacts and the products sold. (The subform to display contact products isn't nested inside another subform to show the companies for the current contact.) Again, the user shouldn't have to look up the default company ID for the current contact before selling a product. Note that in Figure 25-1, we navigated to the fourth contact record.

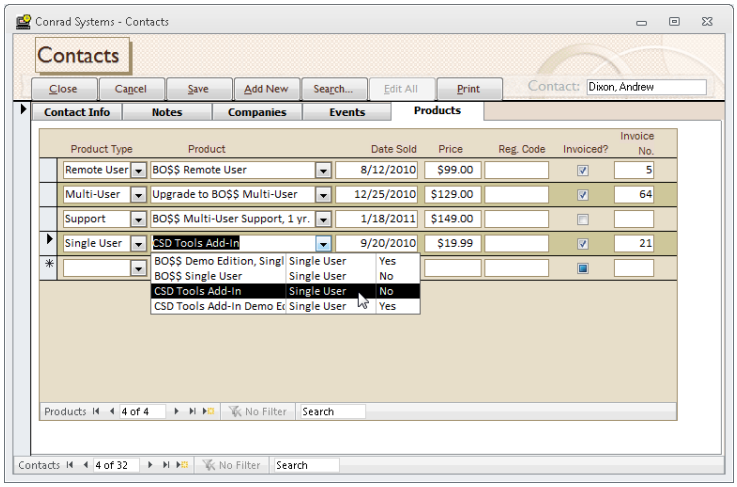

Figure 25-1 Selling a product to a contact involves filling in the price and the default company.

As you can see, a combo box on the subform (fsubContactProducts) helps the user choose the product to sell. Part of the secret to setting the price (the SoldPrice field in tblContactProducts) automatically is in the row source query for the combo box, qlkpProductsForContacts, as shown in Figure 25-2.

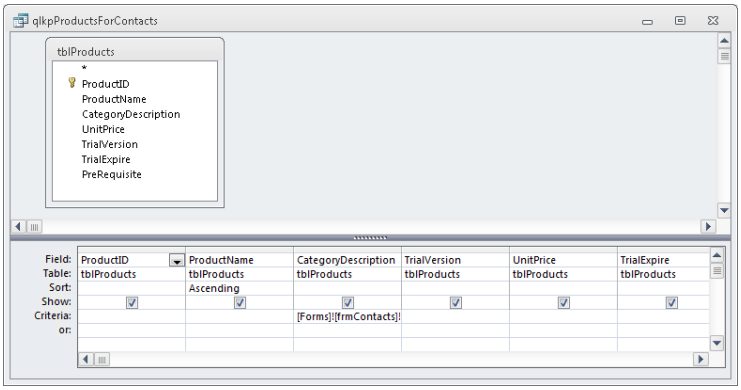

Figure 25-2 The qlkpProductsForContacts query is the row source for the Product combo box on fsubContactProducts.

You certainly need the ProductID field for the new record in tblContactProducts. Displaying the ProductName field in the combo box is more meaningful than showing the ProductID number, and, as you can see in Figure 25-1, the list in the combo box also shows you the CategoryDescription and whether the product is a trial version. But why did we include the UnitPrice, TrialExpire, and PreRequisite columns in the query's design grid?

As it turns out, you can retrieve any of these fields from the current row in the combo box by referencing the combo box Column property. (You'll see later in this chapter, in "Validating Complex Data" on page 1604, that other code behind the form uses the additional fields to make sure the contact already owns any prerequisite product.) You can see the simple line of code that copies the UnitPrice field by opening the Visual Basic module behind the fsubContactProducts form. Go to the Navigation pane, select the fsubContactProducts form, right-click the form and click Design View on the menu, and then click the View Code button in the Tools group on the Design tab. In the Visual Basic Editor (VBE) Code window, scroll down until you find the cmbProductID_AfterUpdate procedure. The code is as follows:

```
Private Sub cmbProductID_AfterUpdate()
  ' Grab the default price from the hidden 5th column
  Me.SoldPrice = Me.cmbProductID.Column(4)
End Sub
```

Notice that you use an index number to fetch the column you want and that the index starts at zero. You can reference the fifth column in the query (UnitPrice) by asking for the Column(4) property of the combo box. Notice also that the code uses the Me shortcut object to reference the form object where this code is running. Therefore, every time you

pick a different product, the After Update event occurs for the ProductID combo box, and this code fills in the related price automatically. Close the fSubContactProducts form before continuing with the next section.

If you open the frmContacts form in Design view, select the fsubContactProducts form on the Products tab, and examine the Link Child Fields and Link Master Fields properties, you'll find that the two forms are linked on ContactID. However, the tblContactProducts table also needs a CompanyID field in its primary key. Code in the module for the fsubContactProducts form handles fetching the default CompanyID for the current contact, so you don't need an intermediary subform that would clutter the form design. If you still have the module for the fsubContactProducts form open in the VBE window, you can find the code in the Form_BeforeInsert procedure. The code is as follows:

```
Private Sub Form_BeforeInsert(Cancel As Integer)
Dim varCompanyID As Variant
  ' First, disallow insert if nothing in outer form
  If IsNothing(Me.Parent.ContactID) Then
    MsgBox "You must define the contact information on a new row before " & _
      "attempting to sell a product", vbCritical, gstrAppTitle
    Cancel = True
    Exit Sub
  End If
  ' Try to lookup this contact's Company ID
  varCompanyID = DLookup("CompanyID", "qryContactDefaultCompany", _
    "(ContactID = " & Me.Parent.ContactID.Value & ")")
    If IsNothing(varCompanyID) Then
    ' If not found, then disallow product sale
    MsgBox "You cannot sell a product to a Contact that does not have a " & _
      "related Company that is marked as the default for this Contact." & _
      "  Press Esc to clear your edits and click on the Companies tab " & _
      "to define the default Company for this Contact.", vbCritical, _
      gstrAppTitle
        Cancel = True
    Else
        ' Assign the company ID behind the scenes
        Me.CompanyID = varCompanyID
    End If
End Sub
```

This procedure executes whenever the user sets any value on a new row in the subform. First, it makes sure that the outer form has a valid ContactID. Next, the code uses the DLookup domain function to attempt to fetch the default company ID for the current contact. The query includes a filter to return only the rows from tblCompanyContacts where the DefaultForContact field is True. If the function returns a valid value, the code sets the required CompanyID field automatically. If it can't find a CompanyID, the code uses the MsgBox statement to tell the user about the error.

Note

The IsNothing function that you see used in code throughout all the sample applications is not a built-in Visual Basic function. This function tests the value you pass to it for "nothing"—Null, zero, or a zero-length string. You can find this function in the modUtility standard module in all the sample databases.

INSIDE OUT Understanding the Useful Domain Functions

Quite frequently in code, in a query, or in the control source of a control on a form or report, you might need to look up a single value from one of the tables or queries in your database. Although you can certainly go to the trouble of defining and opening a recordset in code, Microsoft Access provides a set of functions, called domain functions, that can provide the value you need with a single function call. The available functions are as follows:

Function Name	Description
DFirst, DLast	Return a random value from the specified domain (the table or query that's the record source)
DLookup	Looks up a value in the specified domain
DMax	Returns the highest (Max) value in the specified domain
DMin	Returns the lowest (Min) value in the specified domain
DStDev, DstDevP	Return the standard deviation of a population sample or a population of the specified domain
DSum	Returns the sum of an expression from a domain
DVar, DVarP	Return the variance of a population sample or a population of the specified domain

The syntax to call a domain function is as follows:

```
<function name>(<field expression>, <domain name> [, <criteria> ])
```

where

<*function name*> is the name of one of the functions in the preceding list;

<*field expression*> is a string literal or name of a string variable containing the name of a field or an expression using fields from the specified domain;

<*domain name*> is a string literal or name of a string variable containing the name of a table or query in your database;

<*criteria*> is a string literal or name of a string variable containing a Boolean comparison expression to filter the records in the domain

Note that when a domain function finds no records, the returned value is a Null, so you should always assign the result to a Variant data type variable. When you construct a criteria expression, you must enclose string literals in quotes and date/time literals in the # character. (If you use double quotes to delimit the criteria string literal, then use single quotes around literals inside the string, and vice versa.) For example, to find the lowest work postal code value for all contacts where the contact type is customer and the birth date is before January 1, 1970, enter:

```
DMin("WorkPostalCode", "tblContacts", "[ContactType] = 'customer'
And Format([BirthDate], 'mm/dd/yyyy') < #01/01/1970#")
```

Handling the NotInList Event

In almost every data entry form you'll ever build, you'll need to provide a way for the user to set the foreign key of the edited record on the *many* side of a relationship to point back to the correct one side record—for example, to set the ProductID field in the tblContact-Products table when selling a product on the Products tab of the frmContacts form. But what if the user needs to create a new product? Should the user have to open the form to edit products first to create the new product before selling it? The answer is a resounding no, but you must write code in the NotInList event of the combo box to handle new values and provide a way to create new rows in the tblProducts table.

Figure 25-3 shows you what happens when the user tries to type a product name that's not already in the tblProducts table. In this case, the customer wants to purchase a two-year support contract instead of the already available one-year product. You can see that something has intercepted the new product name to confirm that the user wants to add the new product.

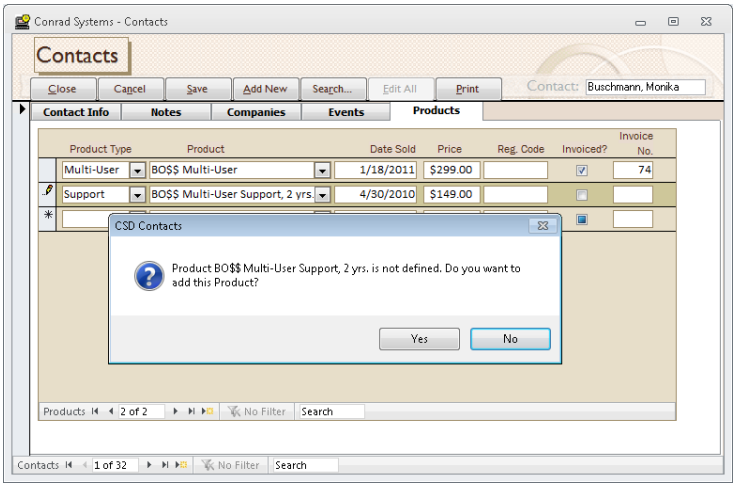

Figure 25-3 When you enter a product that isn't defined in the database, the application asks if you want to add the new product.

First, the combo box has been defined with its Limit To List property set to Yes. Second, there's an event procedure defined to handle the NotInList event of the combo box, and it is this code that's asking whether the user wants to add a product. If the user clicks Yes to confirm adding this product, the event procedure opens the frmProductAdd form in Dialog mode to let the user enter the new data, as shown in Figure 25-4. Opening a form in Dialog mode forces the user to respond before the application resumes execution. The code that opens this form passes the product name entered and the product type that the user selected before entering a new product name. The user can fill in the price and other details. The user can also click Cancel to avoid saving the record and close the form. If the user clicks Save, the form saves the new product record and closes to allow the code in the NotInList event procedure to continue.

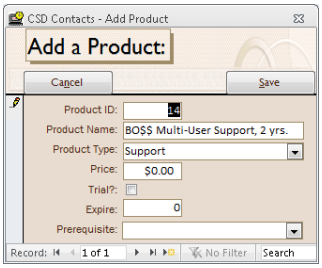

Figure 25-4 The frmProductAdd form lets you define the details for the new product.

To see how this works, open the fsubContactProducts form in Design view, select the cmbProductID combo box control from the Selection Type combo box on the Property Sheet window, find the On Not In List event property in the Properties window, and click Build to open the code. (We had you select the combo box from the Property Sheet window because the ProductName text box control overlays the cmbProductID combo box control on the form design grid.) The code for the procedure is shown here:

```
Private Sub cmbProductID_NotInList(NewData As String, Response As Integer)
Dim strType As String, strWhere As String
  ' User has typed in a product name that doesn't exist
  strType = NewData
  ' Set up the test predicate
  strWhere = "[ProductName] = """ & strType & """"
  ' Ask if they want to add this product
  If vbYes = MsgBox("Product " & NewData & " is not defined. " & _
    "Do you want to add this Product?", vbYesNo + vbQuestion + _
    vbDefaultButton2, gstrAppTitle) Then
    ' Yup.  Open the product add form and pass it the new name
    ' - and the pre-selected Category
    DoCmd.OpenForm "frmProductAdd", DataMode:=acFormAdd, _
      WindowMode:=acDialog, _
      OpenArgs:=strType & ";" & Me.cmbCategoryDescription
    ' Verify that the product really got added
    If IsNull(DLookup("ProductID", "tblProducts", strWhere)) Then
      ' Nope.
      MsgBox "You failed to add a Product that matched what you entered." & _
        "  Please try again.", vbInformation, gstrAppTitle
      ' Tell Access to continue - we trapped the error
      Response = acDataErrContinue
    Else
      ' Product added OK - tell Access so that combo gets requeried
      Response = acDataErrAdded
    End If
  Else
    ' Don't want to add - let Access display normal error
    Response = acDataErrDisplay
  End If
End Sub
```

As you can see, Access passes two parameters to the NotInList event. The first parameter (NewData) contains the string you typed in the combo box. You can set the value of the second parameter (Response) before you exit the sub procedure to tell Access what you want to do. You wouldn't have access to these parameters in a macro, so you can see that this event requires a Visual Basic procedure to handle it properly.

The procedure first creates the criteria string that it uses later to verify that the user saved the product. Next, the procedure uses the MsgBox function to ask whether the user wants to add this product to the database (the result shown in Figure 25-3). If you've ever looked at the MsgBox function Help topic, you know that the second parameter is a number that's

the sum of all the options you want. Fortunately, Visual Basic provides named constants for these options, so you don't have to remember the number codes. In this case, the procedure asks for a question mark icon (vbQuestion) and for the Yes and No buttons (vbYesNo) to be displayed. It also specifies that the default button is the second button (vbDefaultButton2)—the No button—just in case the user quickly presses Enter upon seeing the message.

If the user clicks Yes in the message box, the procedure uses DoCmd.OpenForm to open the frmProductAdd form in Dialog mode and passes it the product name entered and the product type selected by setting the form's OpenArgs property. Note that the use of the named parameter syntax in the call to DoCmd.OpenForm makes it easy to set the parameters you want. You *must* open the form in Dialog mode. If you don't, your code continues to run while the form opens. Whenever a dialog box form is open, Visual Basic code execution stops until the dialog box closes, which is critical in this case because you need the record to be saved or canceled before you can continue with other tests.

After the frmProductAdd form closes, the next statement calls the DLookup function to verify that the product really was added to the database. If the code can't find a new matching product name (the user either changed the product name in the add form or clicked Cancel), it uses the MsgBox statement to inform the user of the problem and sets a return value in the Response parameter to tell Access that the value hasn't been added but that Access can continue without issuing its own error message (acDataErrContinue).

If the matching product name now exists (indicating the user clicked Save on the frmProductAdd form), the code tells Access that the new product now exists (acDataErrAdded). Access re-queries the combo box and attempts a new match. Finally, if the user clicks No in the message box shown in Figure 25-3, the procedure sets Response to acDataErrDisplay to tell Access to display its normal error message.

The other critical piece of code is in the Load event for the frmProductAdd form. The code is as follows:

```
Private Sub Form_Load()
Dim intI As Integer
  If Not IsNothing(Me.OpenArgs) Then
    ' If called from "not in list", Openargs should have
    '   Product Name; Category Description
    ' Look for the semi-colon separating the two
    intI = InStr(Me.OpenArgs, ";")
    ' If not found, then all we have is a product name
    If intI = 0 Then
      Me.ProductName = Me.OpenArgs
    Else
      ' If called from fsubContactProducts,
      '  .. have category only
      If intI > 1 Then
          ' Have a product name - grab it
          Me.ProductName = Left(Me.OpenArgs, intI - 1)
```

```
      End If
      Me.CategoryDescription = Mid(Me.OpenArgs, intI + 1)
      ' lock the category
      Me.CategoryDescription.Locked = True
      Me.CategoryDescription.Enabled = False
      ' .. and clear the tool tip
      Me.CategoryDescription.ControlTipText = ""
    End If
  End If
End Sub
```

If you remember, the cmbProductID NotInList event procedure passes the original string that the user entered and selected the product type (the CategoryDescription field) as the OpenArgs parameter to the OpenForm method. This sets the OpenArgs property of the form being opened. The OpenArgs property should contain the new product name, a semicolon, and the selected product type, so the Form_Load procedure parses the product name and product type by using the InStr function to look for the semicolon. (The InStr function returns the offset into the string in the first parameter where it finds the string specified in the second parameter, and it returns 0 if it doesn't find the search string.) The code then uses the two values it finds to set the ProductName and CategoryDescription fields. Also, when the code finds a category description, it locks that combo box so that the user can't change it to something other than what was selected on the new product row in the original form.

Fixing an E-Mail Hyperlink

As you learned in Chapter 9, "Creating and Working with Simple Queries," one of the easiest ways to enter a hyperlink is to use the Insert Hyperlink feature. However, you can also type the hyperlink address directly into the field in a datasheet or form. Remember that a hyperlink field can contain up to four parts: display text, hyperlink address, bookmark, and ScreenTip text. If a user simply enters an e-mail address into a hyperlink field, Access 2010 recognizes the format, adds the *mailto:* protocol, and uses what the user typed as the display text. For example, if the user enters

```
jconrad@proseware.com
```

Access stores in the hyperlink field

```
jconrad@proseware.com#mailto:jconrad@proseware.com#
```

Rather than repeat the e-mail address as the display text, the result might look better if the display text is the person's name rather than a repeat of the e-mail address. One of the forms that has an e-mail address is the frmContacts form in the Conrad Systems Contacts application. You can find the code that examines and attempts to fix the address in the AfterUpdate event procedure for the EmailName text box. (If the user enters some valid protocol other than *http://* or *mailto:*, this code won't fix it.) The code is as follows:

```
Private Sub EmailName_AfterUpdate()
' If you just type in an email name: Somebody@hotmail.com
' Access changes it to: Somebody@hotmail.com#mailto:somebody@hotmail.com#
' This code replaces the display field with the user name
Dim intI As Integer
    ' Don't do anything if email is empty
    If IsNothing(Me.EmailName) Then Exit Sub
    ' Fix up http:// if it's there
    ' This was an old bug in 2003 and earlier, but fixed in Access 2007
    Me.EmailName = Replace(Me.EmailName, "http://", "mailto:")
    ' Now look for the first "#" that delimits the hyperlink display name
    intI = InStr(Me.EmailName, "#")
    ' And put the person name there instead if found
    If intI > 0 Then
      Me.EmailName = (Me.FirstName + " ") & Me.LastName & _
        Mid(Me.EmailName, intI)
    End If
End Sub
```

If the user clears the EmailName text box, the code doesn't do anything. If there's something in the text box, the code uses the Replace function to search for an incorrect *http://* and replace it with the correct *mailto:* protocol identifier. As you know, a hyperlink field can contain text that is displayed instead of the hyperlink, a # character delimiter, and the actual hyperlink address. The code uses the InStr function to check for the presence of the delimiter. (The InStr function returns the offset into the string in the first parameter where it finds the string specified in the second parameter.) If the code finds the delimiter, it replaces the contents of the field with the person's first and last name as display text followed by the text starting with the # delimiter. (The Mid function called with no length specification—the optional third parameter—returns all characters starting at the specified offset.)

> **Note**
>
> **In Access 2003 and earlier, when you typed an e-mail address without the *mailto:* protocol prefix into a hyperlink field, Access would store the hyperlink with the *http://* protocol prefix in error. This bug was fixed in Access 2007, but the preceding code will fix that problem if you use it in the earlier versions of the software.**

Providing a Graphical Calendar

You can always provide an input mask to help a user enter a date and time value correctly, but an input mask can be awkward—always requiring, for example, that the user type a two-digit month. An input mask also can conflict with any default value that you might want to assign. It's much more helpful if the user can choose the date using a graphical calendar.

Access 2010 provides a Show Date Picker property for text boxes. You can set this property to For Dates to instruct Access to display a calendar icon next to the control when it contains a date/time value and has the focus. The user can click the button to pop open a graphical calendar to select a date value. But Show Date Picker isn't available for controls other than the text box control, and the date picker lets the user enter only a date, not a date and time. You also are restricted to moving backward and forward one month at a time using the left and right arrow keys on the date picker, so if you need to enter a date that is many months away from the current date, such as a birth date, you'll have to click the arrow keys many times to reach the correct month and year.

Both the Conrad Systems Contacts and the Housing Reservations sample applications provide sample calendar forms and code you can use to set a date/time value in any control. The two applications have a calendar form—frmCalendar—that uses Visual Basic code to "draw" the calendar on a form using an option group and toggle button controls. This calendar form provides an option to enter a time as well as select a date.

TROUBLESHOOTING

Why isn't Access setting my defined default value for a date/time field?

Did you also define an Input Mask property? If so, then that's your problem. A date/time field is actually a floating-point number, but Access always converts and displays the character value in table and query datasheets and forms and reports. When you define an Input Mask property, any Default Value setting must match the restrictions imposed by the input mask. If the value violates the restrictions, Access won't use the default value. When you assign a default value to a date/time field, you typically use the Date or Now built-in functions. These functions return a valid date/time floating-point value—which probably won't match your input mask restrictions. To have Access use the default value, you must format it to match your input mask. For example, if your input mask is 90/00/0000\ 00:00, then you should set the Default Value property of the field or control to =Format(Now(), "mm/dd/yyyy hh:nn"). This forces Access to return a string value as the default that matches your input mask.

This graphical facility is available in the sample applications wherever you see a small command button next to a control containing a date or date/time field on a form. Click the button to open the calendar and set the value. One control that uses our custom calendar form is the ContactDateTime control on the Events tab of the frmContacts form. You can see the calendar open in Figure 25-5.

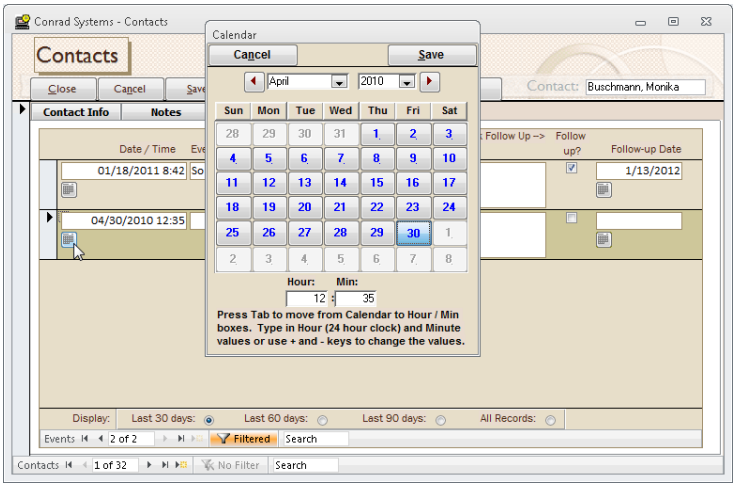

Figure 25-5 Click the calendar command button next to the ContactDateTime control on the Events tab of the frmContacts form to open a graphical form to select the date and enter the time.

The code in the Click event of this command button calls a public function to open the form and pass it the related control that should receive the resulting date value. You can find the code shown here in the module for the fsubContactEvents form:

```
Private Sub cmdContactTimeCal_Click()
Dim varReturn As Variant
  ' Clicked the calendar icon asking for graphical help
  ' Put the focus on the control to be updated
  Me.ContactDateTime.SetFocus
  ' Call the get a date/time function
  varReturn = GetDate(Me.ContactDateTime, False)
End Sub
```

INSIDE OUT

Use Keyboard Shortcuts to Jump to Procedures and Functions

If you highlight the function call, GetDate in our previous example, and then press Shift+F2, Access takes you directly to that function in the modCalendar module.

When the user clicks the command button, Access moves the focus to it. The code moves the focus back to the date field to be edited and calls the public function where the real action happens. You can find the code for the function GetDate in the modCalendar module; the code is also listed here:

```
Option Compare Database
Option Explicit
Public Function GetDate(ctl As control, _
    Optional intDateOnly As Integer = 0) As Integer
'---------------------------------------------------------------
' Inputs: A Control object containing a date/time value
'         Optional "date only" (no time value) flag
' Outputs: Sets the Control to the value returned by frmCalendar
' Created By: JLV 09/05/01
' Last Revised: JLV 09/05/01
'---------------------------------------------------------------
Dim varDateTime As Variant
Dim strDateTime As String
Dim frm As Form
    ' Error trap just in case
    On Error GoTo Error_Date
    ' First, validate the kind of control passed
    Select Case ctl.ControlType
        ' Text box, combo box, and list box are OK
        Case acTextBox, acListBox, acComboBox
        Case Else
            GetDate = False
            Exit Function
    End Select
    ' If the control has no value
    If IsNothing(ctl.Value) Then
        If intDateOnly Then
            ' Set default date
            varDateTime = Date
        Else
            ' .. or default date and time
            varDateTime = Now
        End If
    Else
        ' Otherwise, pick up the current value
        varDateTime = ctl.Value
        ' Make sure it's a date/time
        If vbDate <> varType(varDateTime) Then
            GetDate = False
            Exit Function
        End If
    End If
    ' Turn the date and time into a string
    ' to pass to the form
    strDateTime = Format(varDateTime, "mm/dd/yyyy hh:nn")
    ' Make sure we don't have an old copy of
    ' frmCalendar hanging around
    If IsFormLoaded("frmCalendar") Then
        DoCmd.Close acForm, "frmCalendar"
    End If
    ' Open the calendar as a dialog so this code waits,
    ' and pass the date/time value
```

```
    DoCmd.OpenForm "frmCalendar", WindowMode:=acDialog, _
        OpenArgs:=strDateTime & "," & intDateOnly
    ' If the form is gone, user canceled the update
    If Not IsFormLoaded("frmCalendar") Then Exit Function
    ' Get a pointer to the now-hidden form
    Set frm = Forms!frmCalendar
    ' Grab the date part off the hidden text box
    strDateTime = Format(frm.ctlCalendar.Value, "dd-mmm-yyyy")
    If Not intDateOnly Then
        ' If looking for date and time,
        ' also grab the hour and minute
        strDateTime = strDateTime & " " & frm.txtHour & _
        ":" & frm.txtMinute
    End If
    ' Stuff the returned value back in the caller's control
    ctl.Value = DateValue(strDateTime) + TimeValue(strDateTime)
    ' Close the calendar form to clean up
    DoCmd.Close acForm, "frmCalendar"
    GetDate = True
Exit_Date:
    Exit Function
Error_Date:
    ' This code is pretty simple and does check for
    ' a usable control type,
    '  .. so this should never happen.
    ' But if it does, log it...
    ErrorLog "GetDate", Err, Error
    GetDate = False
    Resume Exit_Date
End Function
```

The function begins by setting an error trap that executes the code at the Error_Date label if anything goes wrong. The function accepts two arguments—ctlToUpdate and intDateOnly. Access uses the ctlToUpdate argument to examine the control type and control value passed into the function. Access uses the optional intDateOnly argument to indicate whether the control needs a date and time or a date only. The function checks to see if the control passed in is a text box, combo box, or list box. If the control matches one of these control types, the function continues; otherwise, the function exists. The function then checks the value of the control passed in. If there is no value in the control, Access assigns the current date to the variant varDateTime or the current date and time, depending upon the value of the intDateOnly argument. If the control has a value, the function assigns the current value of the control to the variant and also verifies the value is a valid date and time. Next, the function converts the variant value to a string and checks to see if the frmCalendar form is open. If the function finds the calendar form currently open, the function closes it, and then reopens the form in Dialog mode. In the OpenForm call to frmCalendar, the function passes in the string value of the date and time and the intDateOnly value as OpenArgs parameters.

After the user clicks either the Save or Cancel button on the frmCalendar form, this function continues executing. If the user cancels the update by clicking the cmdCancel command button on the frmCalendar form, the function exits. If the user clicks the cmdSave command button on the frmCalendar form, the function grabs the selected date and time information off the form, sets the calling control's value to the date and time data, closes the frmCalendar form, and then exits.

The final pieces of code that make all of this work are in the module behind the frmCalendar form. The code in the Load event of the form is listed here:

```
Private Sub Form_Load()
    ' Establish an initial value for the date
    If IsNothing(Me.OpenArgs) Then
        varDate = Date
    Else
        ' Should have date, time, and "DateOnly"
        ' indicator in OpenArgs:
        '   mm/dd/yyyy hh:mm,-1
        varDate = Left(Me.OpenArgs, 10)
        Me.txtHour = Mid(Me.OpenArgs, 12, 2)
        Me.txtMinute = Mid(Me.OpenArgs, 15, 2)
        ' If "date only"
        If Right(Me.OpenArgs, 2) = "-1" Then
            ' Hide some stuff
            Me.txtHour.Visible = False
            Me.txtMinute.Visible = False
            Me.lblColon.Visible = False
            Me.lblTimeInstruct.Visible = False
            Me.SetFocus
            '  .. and resize my window
            DoCmd.MoveSize , , , 4295
        End If
    End If
    ' Initialize the month selector
    Me.cmbMonth = Month(varDate)
    ' Initialize the year selector
    Me.cmbYear = Year(varDate)
    ' Call the common calendar draw routine
    SetDays
    ' Place the date/time value in a hidden control -
    '  The calling routine fetches it from here
    Me.ctlCalendar = varDate
    ' Highlight the correct day box in the calendar
    Me.optCalendar = Day(varDate)
End Sub
```

This code first checks to see if any OpenArgs parameters are passed into the form. If no OpenArgs parameters are passed in, the code assigns the current date to a variant called varDate. If there are OpenArgs parameters passed in, the code parses out the OpenArgs elements for the date and possible time portion. If the optional intDateOnly variable from the GetDate function is True (the control needs only a date value, not a date and time value), the form shrinks to hide those text boxes. Because the event date/time field needs a time value, this parameter is False, so you should be able to see the hour and minute text boxes. The final parts of the code set up the calendar from control elements to match either the value already in the control or the system date and time. (Note that the code calls a SetDays procedure included in the form's class module to set up the various from controls.)

After the setup code for the calendar form controls completes, the form waits until the user enters a value and clicks Save or decides not to change the value by clicking Cancel. The code for the two procedures that respond to the command buttons are as follows:

```
Public Sub cmdCancel_Click()
  ' Closing doesn't pass the value back
  DoCmd.Close acForm, Me.Name
End Sub

Private Sub cmdSave_Click()
  ' Hiding this dialog lets the calling code in GetDate continue
  Me.Visible = False
End Sub
```

Clicking the Cancel button (cmdCancel_Click) simply closes the form without changing any value in the control passed to the form. The code that saves the value that the user selects on the graphical calendar is in the GetDate module. To save the value, the click event for the cmdSave command button simply hides the frmCalendar form.

TROUBLESHOOTING

Why can't I find the ActiveX Calendar Control in Access 2010?

In Access 2010, Microsoft removed the ActiveX Calendar Control that shipped with many past versions of Access. If you want to show a graphical calendar to your users when they need to select a date, you'll need to use the newer, built-in Date Picker control included with Access or create your own calendar form using Visual Basic, as we have in the Conrad Systems Contacts and the Housing Reservations sample applications.

Working with Linked Photos

Although you can certainly store and display photos in an Access application using the OLE Object data type, if your application needs to handle hundreds or thousands of photos, you could easily exceed the 2-GB file size limit for an .accdb file. You can also use the Attachment data type for your photos to store them more efficiently, but you still might run into file size limitations if you need to store many photos. The alternative method is to store the pictures as files and save the picture path as a text field in your tables.

The good news is the image control in Access 2010 lets you specify a Control Source property. When this property points to a field containing a folder and file location as a text string, the image control will load the photo for you from that location. However, you should still provide features in your forms to help users to easily edit the file location information.

The Housing Reservations database (Housing.accdb) is designed to use this functionality. Open the Housing.accdb sample database and then open the frmEmployeesPlain form, as shown in Figure 25-6. The employee picture you see on the frmEmployees and frmEmployeesPlain forms is fetched by the image control from the path stored in the Photo field of the table.

Figure 25-6 The image control loads the photo on the Employees form from a picture path.

Notice that the user cannot see the contents of the Photo field that contains the picture path information. However, we've provided two command buttons to make it easy for the user to edit or delete the photo path information.

Deleting and Updating an Image Path

Clearing the file name saved in the record is the easy part, so let's look at that first. Behind the Delete button that you can see on the frmEmployeesPlain form, you can find the following code:

```
Private Sub cmdDelete_Click()
' User asked to remove the picture
  ' Clear photo
  Me.txtPhoto = Null
  ' Set the message
  Me.lblMsg.Caption = "Click Add to create a photo for this employee."
  ' Put focus in a safe place
  Me.FirstName.SetFocus
End Sub
```

When the user clicks the command button asking to delete the photo, the code sets the photo path to Null and displays the informative label. Setting the Photo field to Null causes the mage control to remove the image. Because the background of the image control is transparent, the label control hidden behind it shows through, displaying an informative message.

The tricky part is to provide the user with a way to enter the picture path to add or update a picture in a record. Although you could certainly use the InputBox function to ask the user for the path, it's much more professional to call the Open File dialog box in Windows so that the user can navigate to the desired picture using familiar tools. The bad news is calling any procedure in Windows is complex and usually involves setting up parameter structures and a special declaration of the external function. The good news is the Microsoft Office 2010 system includes a special FileDialog object that greatly simplifies this process. You need to add a reference to the Microsoft Office library to make it easy to use this object—to do this, from the VBE window, choose References from the Tools menu and be sure the Microsoft Office 14.0 Object Library is selected. After you do this, you can include code using the FileDialog object to load a picture path. You can find the following code behind the Click event of the Add button (cmdAdd) in the frmEmployeesPlain form:

```
Private Sub cmdAdd_Click()
' User asked to add a new photo
Dim strPath As String
  ' Grab a copy of the Office file dialog
  With Application.FileDialog(msoFileDialogFilePicker)
    ' Select only one file
    .AllowMultiSelect = False
    ' Set the dialog title
    .Title = "Locate the Employee picture file"
    ' Set the button caption
    .ButtonName = "Choose"
    ' Make sure the filter list is clear
```

```
    .Filters.Clear
    ' Add two filters
    .Filters.Add "JPEGs", "*.jpg"
    .Filters.Add "Bitmaps", "*.bmp"
    ' Set the filter index to 2
    .FilterIndex = 2
    ' Set the initial path name
    .InitialFileName = CurrentProject.Path & "\Pictures"
    ' Show files as thumbnails
    .InitialView = msoFileDialogViewThumbnail
    ' Show the dialog and test the return
    If .Show = 0 Then
      ' Didn't pick a file - bail
      Exit Sub
    End If
    ' Should be only one filename - grab it
    strPath = Trim(.SelectedItems(1))
    ' Set an error trap
    On Error Resume Next
    ' Set the image
    Me.txtPhoto = strPath
    ' Set the message in case Image control couldn't find it
    Me.lblMsg.Caption = "Failed to load the picture you selected." & _
      "  Click Add to try again."
  End With
  ' Put focus in a safe place
  Me.FirstName.SetFocus
End Sub
```

The code establishes a pointer to the FileDialog object using a With statement, sets the various properties of the object (including the allowed file extensions and the initial path), and then uses the Show method to display the Open File dialog box. Setting the Photo field causes the image control to load the new picture, but the code also sets the message hidden behind the image control just in case the image control had a problem loading the file.

Validating Complex Data

Although you can certainly take advantage of the Input Mask property and the field and table Validation Rule properties, your application often has additional business rules that you can enforce only by adding code behind the forms you provide to edit the data or by using data macros at the data level. The following examples show you how several of the business rules in the Conrad Systems Contacts and Housing Reservations applications are enforced with Visual Basic code.

Checking for Possible Duplicate Names

When you design a table, you should attempt toone identify some combination of fields that will be unique across all records to use as your primary key. However, when you create a table to store information about people, you usually create an artificial number as the primary key of the table because you would need to combine many fields to ensure a unique value. Even when you attempt to construct a primary key from first name, last name, address, postal code, and phone number, you still can't guarantee a unique value across all rows.

Using an artificial primary key doesn't mean you should abandon all efforts to identify potentially duplicate rows. Code in the frmContacts form in the Conrad Systems Contacts application (Contacts.accdb) checks the last name the user enters for a new record and issues a warning message if it finds any close names. For example, if the user creates a new record and enters a last name like "Viscas" (assuming John's record is still in the table), code behind the form's BeforeUpdate event detects the similar name and issues the warning shown in Figure 25-7.

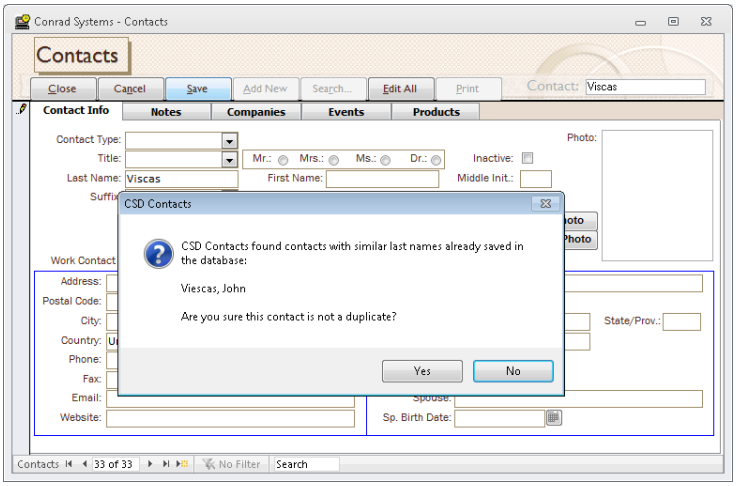

Figure 25-7 The application warns you about a potential duplicate name in the contacts table.

The code searches for potential duplicates by comparing the Soundex codes of the last names. The formula for generating a Soundex code for a name was created by the U.S. National Archives and Records Administration (NARA). Soundex examines the letters by sound and produces a four-character code. When the codes for two names match, it's

likely that the names are very similar and sound alike. Therefore, by using Soundex, the error-checking code not only finds existing contacts with exactly the same last name but also other contacts whose name might be the same but one or both might be slightly misspelled.

Access 2010 doesn't provide a built-in Soundex function (SQL Server does), but it's easy to create a simple Visual Basic procedure to generate the code for a name. You can find a Soundex function in the modUtility module in both the Conrad Systems Contacts and Housing Reservations sample databases. You can find the code that checks for a potential duplicate name in the BeforeUpdate event procedure of the frmContacts form. The code is as follows:

```
Private Sub Form_BeforeUpdate(Cancel As Integer)
Dim rst As DAO.Recordset, strNames As String
  ' If on a new row,
  If (Me.NewRecord = True) Then
    ' Check for similar name
    If Not IsNothing(Me.LastName) Then
      ' Open a recordset to look for similar names
      Set rst = CurrentDb.OpenRecordset("SELECT LastName, FirstName FROM " & _
        "tblContacts WHERE Soundex([LastName]) = '" & _
        Soundex(Me.LastName) & "'")
      ' If got some similar names, collect them for the message
      Do Until rst.EOF
        strNames = strNames & rst!LastName & ", " & rst!FirstName & vbCrLf
        rst.MoveNext
      Loop
      ' Done with the recordset
      rst.Close
      Set rst = Nothing
      ' See if we got some similar names
      If Len(strNames) > 0 Then
        ' Yup, issue warning
        If vbNo = MsgBox("CSD Contacts found contacts with similar " & _
         "last names already saved in the database: " & vbCrLf & vbCrLf & _
          strNames & vbCrLf & _
          "Are you sure this contact is not a duplicate?", _
          vbQuestion + vbYesNo + vbDefaultButton2, gstrAppTitle) Then
            ' Cancel the save
            Cancel = True
        End If
      End If
    End If
  End If
End Sub
```

The code checks only when the user is about to save a new row. It opens a recordset to fetch any other contact records where the Soundex code of the last name matches the last name about to be saved. It includes all names it finds in the warning message so that

the user can verify that the new contact is not a duplicate. If the user decides not to save the record, the code sets the Cancel parameter to True to tell Access not to save the new contact.

Testing for Related Records When Deleting a Record

You certainly can and should define relationships between your tables and ask Access to enforce referential integrity to prevent saving unrelated records or deleting a record that still has related records in other tables. In most cases, you do not want to activate the cascade delete feature to automatically delete related records. However, Access displays a message "The record cannot be deleted or changed because 'tblXYZ' contains related records" whenever the user tries to delete a record that has dependent records in other tables.

You can do your own testing in code behind your forms in the Delete event and give the user a message that more clearly identifies the problem. For example, here's the code in the Delete event procedure of the frmContacts form in the Conrad Systems Contacts application:

```
Private Sub Form_Delete(Cancel As Integer)
Dim db As DAO.Database, qd As DAO.QueryDef, rst As DAO.Recordset
Dim varRelate As Variant
  ' Check for related child rows
  ' Get a pointer to this database
  Set db = CurrentDb
  ' Open the test query
  Set qd = db.QueryDefs("qryCheckRelateContact")
  ' Set the contact parameter
  qd!ContactNo = Me.ContactID
  ' Open a recordset on the related rows
  Set rst = qd.OpenRecordset()
  ' If we got rows, then can't delete
  If Not rst.EOF Then
    varRelate = Null
    ' Loop to build the informative error message
    rst.MoveFirst
    Do Until rst.EOF
      ' Grab all the table names
      varRelate = (varRelate + ", ") & rst!TableName
      rst.MoveNext
    Loop
    MsgBox "You cannot delete this Contact because you have " & _
      "related rows in " & _
      varRelate & _
      ".  Delete these records first, and then delete the Contact.", _
      vbOKOnly + vbCritical, gstrAppTitle
    ' close all objects
    rst.Close
    qd.Close
    Set rst = Nothing
```

```
    Set qd = Nothing
    Set db = Nothing
    ' Cancel the delete
    Cancel = True
    Exit Sub
  End If
  ' No related rows - clean up objects
  rst.Close
  qd.Close
  Set rst = Nothing
  Set qd = Nothing
  Set db = Nothing
  ' No related rows, so OK to ask if they want to delete!
  If vbNo = MsgBox("Are you sure you want to delete Contact " & _
    Me.txtFullName & "?", _
    vbQuestion + vbYesNo + vbDefaultButton2, gstrAppTitle) Then
    Cancel = True
  End If
End Sub
```

The code uses a special UNION parameter query, qryCheckRelateContact, that attempts to fetch related rows from tblCompanyContacts, tblCompanies (the ReferredBy field), tblContactEvents, and tblContactProducts, and returns the name(s) of the table(s) that have any related rows. When the code finds rows returned by the query, it formats a message containing names more meaningful to the user, and it includes all the tables that the user must clear to be able to delete the contact. The standard Access error message lists only the first related table that Access finds. Even when the check for related records finds no problems, the code also gives the user a chance to decide not to delete the contact after all.

Verifying a Prerequisite

In some applications, it makes sense to save a certain type of record only if prerequisite records exist. For example, in a school or seminar registration application, the user might need to verify that the person enrolling has successfully completed prerequisite courses. In the Conrad Systems Contacts application, it doesn't make sense to sell support for a product that the contact doesn't own. It's not possible to ask Access to perform this sort of test in a validation rule, so you must write code or create a data macro to enforce this business rule.

Figure 25-8 shows you the message the user sees when trying to sell support for a product that the contact doesn't own. This message also appears if the user attempts to sell the special upgrade to a multi-user product and the contact doesn't already own the prerequisite single-user product.

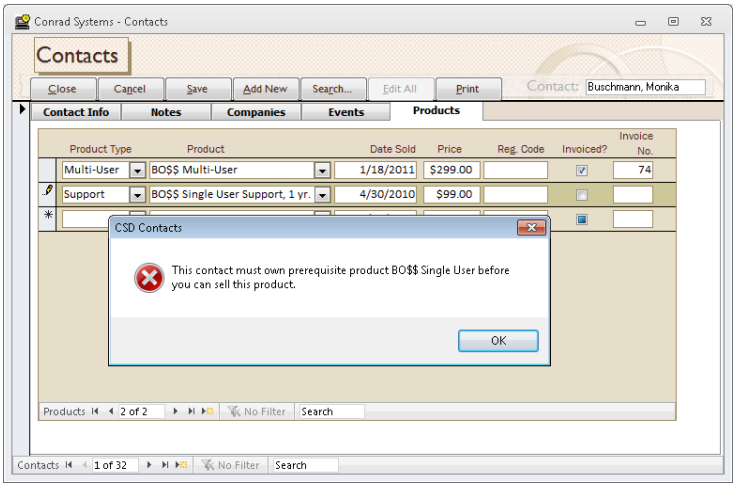

Figure 25-8 Special business rule code won't let you sell a product with a missing prerequisite.

The code that enforces this business rule is in the BeforeUpdate event procedure of the fsubContactProducts form. The code is as follows:

```
Private Sub Form_BeforeUpdate(Cancel As Integer)
Dim lngPreReq As Long, strPreReqName As String
  ' Check for prerequisite
  If Not IsNothing(Me.cmbProductID.Column(6)) Then
    ' Try to lookup the prerequisite for the contact
    lngPreReq = CLng(Me.cmbProductID.Column(6))
    If IsNull(DLookup("ProductID", "tblContactProducts", _
      "ProductID = " & lngPreReq & " And ContactID = " & _
      Me.Parent.ContactID)) Then
      ' Get the name of the prerequisite
      strPreReqName = DLookup("ProductName", "tblProducts", _
        "ProductID = " & lngPreReq)
      ' Display error
      MsgBox "This contact must own prerequisite product " & strPreReqName & _
        " before you can sell this product.", vbCritical, gstrAppTitle
      ' Cancel the edit
      Cancel = True
    End If
  End If
End Sub
```

Remember, from Figure 25-2, that the query providing the row source for the cmbProductID combo box includes any prerequisite product ID in its seventh column. When the code finds a prerequisite, it uses the DLookup function to verify that the current contact already owns the required product. If not, then the code looks up the name of the product,

includes it in an error message displayed to the user, and disallows saving the product by setting the Cancel parameter to True. This enforces the business rule and makes it crystal clear to the user what corrective action is necessary.

Maintaining a Special Unique Value

When two subjects have a many-to-many relationship in your database, you must define a linking table to create the relationship. (See Article 1, "Designing Your Database Application," on the companion CD, for details about designing tables to support a many-to-many relationship.) You will often add fields in the linking table to further clarify the relationship between a row in one of the related tables and the matching row in another. Figure 25-9 shows you the table in the Conrad Systems Contacts application that defines the link between companies and contacts.

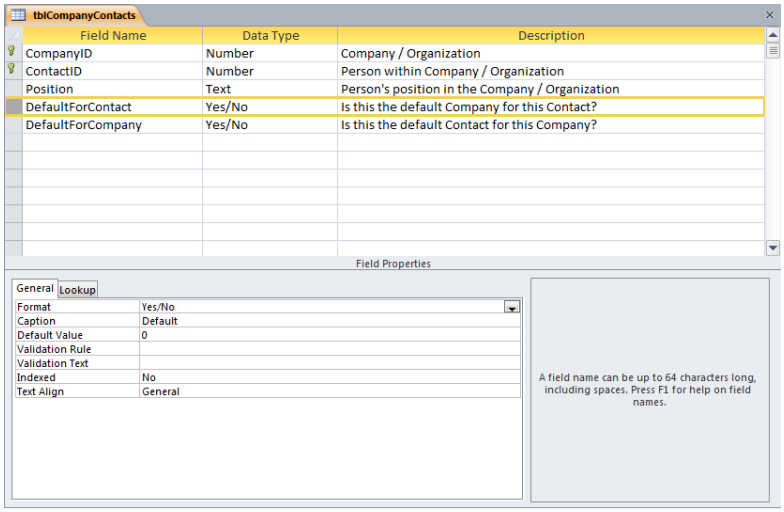

Figure 25-9 The tblCompanyContacts table defines the many-to-many relationship between companies and contacts.

Two special yes/no fields in this table identify which company is the default for a contact and which contact is the default for a company. However, a contact can't have two or more default companies. Likewise, it doesn't make sense for a company to have more than one default contact. To verify this type of special unique value constraint, you can add business rules in code behind the forms you provide to the user to edit this data.

You can find the code that ensures that there is only one default company for each contact in code behind the fsubContactCompanies form in the Conrad Systems Contacts sample application (Contacts.accdb). The code is in the BeforeUpdate event procedure for the DefaultForContact control on the form. The code is as follows:

```
Private Sub DefaultForContact_BeforeUpdate(Cancel As Integer)
  ' Disallow update if there's no Company ID yet
  If IsNothing(Me.CompanyID) Then
    MsgBox "You must select a Company / Organization before" & _
      " you can set Default.", _
      vbCritical, gstrAppTitle
    Cancel = True
    Exit Sub
  End If
  ' Make sure there's only one default
  ' Check only if setting Default = True
  If (Me.DefaultForContact = True) Then
    ' Try to lookup another contact set Default
    If Not IsNothing(DLookup("ContactID", "tblCompanyContacts", _
      "ContactID = " & Me.Parent.ContactID & _
      " AND CompanyID <> " & Me.CompanyID & _
      " AND DefaultForContact = True")) Then
      ' ooops...
      MsgBox "You have designated another Company as the" & _
        " Default for this Contact." & _
        "  You must remove that designation before you" & _
        " can mark this Company as the Default.", _
        vbCritical, gstrAppTitle
      Cancel = True
    End If
  End If
End Sub
```

First, the code verifies that the user has chosen a company for this record. (The Link Child Fields and Link Master Fields properties of the subform control provide the ContactID.) Next, if the user is attempting to mark this company as the default for the contact, the code uses the DLookup function to see if any other record exists (in the tblCompany-Contacts table for the current contact) that is also marked as the default. If it finds such a duplicate record, it warns the user and sets the Cancel parameter to True to prevent saving the change to the control. You'll find similar code in the fsubCompanyContacts form that makes sure only one contact is the primary for any company.

Checking for Overlapping Data

When you build an application that tracks the scheduling of events or reservations that can span a period of time, you most likely need to make sure that a new event or reservation doesn't overlap with an existing one. This can be a bit tricky, especially when the records you're checking have start and end dates or times.

Of course, the Housing Reservations application (Housing.accdb) must make sure that an employee doesn't enter an overlapping reservation request. To see how this works, open the sample database and then open the frmSplash form to start the application. Choose any employee name you like from the combo box in the sign-on dialog box (Jack Richins is a good choice), type **password** as the password, and click Sign On. On the main switchboard, click Reservation Requests. If you see the Edit Reservation Requests dialog box (because you happened to sign on as a manager), click Edit All.

The Reservation Requests form won't let you enter a reservation start date in the past. Click in the blank new row in the list of reservation requests, enter a reservation request for next week for a span of several days, and save the row. (Remember, you can click the Calendar buttons that appear next to the date fields when the focus is on the field to help you choose dates.) Enter another request that overlaps the reservation you just created either at the beginning, the end, or across the middle of the reservation you just entered. Try to save the row, and you should see a warning message similar to the one in Figure 25-10.

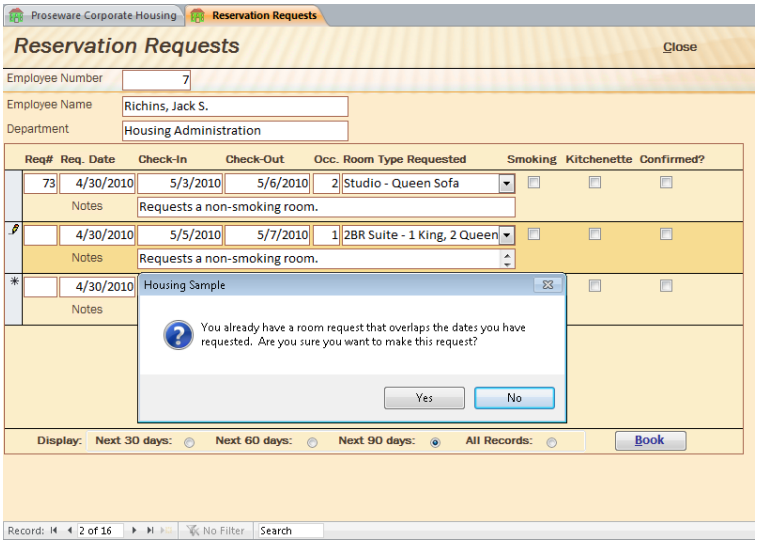

Figure 25-10 The Housing Reservations application displays a warning when you attempt to save an overlapping reservation request.

If you click No, the code cancels your save and returns you to the record to fix it. Notice that you can click Yes to save the duplicate—the application allows this because an employee might intend to reserve two or more rooms on the same or overlapping dates. The code that performs this check in the BeforeUpdate event of the fsubReservationRequests form is as follows (note that this code is near the end of the BeforeUpdate event code):

```
Dim varNum As Variant
  ' Check for overlap with existing request
  ' Try to grab RequestID - will be Null on unsaved row
  varNum = Me.RequestID
  If IsNull(varNum) Then varNum = 0 ' Set dummy value
  If Not IsNull(DLookup("RequestID", "tblReservationRequests", _
    "(EmployeeNumber = " & _
    Me.Parent.EmployeeNumber & ") AND (CheckInDate < #" & _
    Format(Me.CheckOutDate, "mm/dd/yyyy") & _
    "#) AND (CheckOutDate > #" & _
    Format(Me.CheckInDate, "mm/dd/yyyy") & "#) AND (RequestID <> " & _
    varNum & ")")) Then
    If vbNo = MsgBox("You already have a room request " & _
      "that overlaps the dates you have " & _
      "requested.  Are you sure you want to make this request?", _
      vbQuestion + vbYesNo + vbDefaultButton2, gstrAppTitle) Then
      Cancel = True
      Exit Sub
    End If
  End If
```

The code uses the DLookup function to see if another reservation exists (but a different request ID) for the same employee with dates that overlap. The criteria asks for any record that has a check-in date earlier than the requested checkout date (an employee can legitimately check out and then check back in on the same date) and a checkout date that is later than the requested check-in date. You might be tempted to build more complex criteria that checks all combinations of reservations that overlap into the start of the requested period, overlap into the end of the requested period, span the entire requested period, or are contained wholly within the requested period, but the two simple tests are all you need.

Controlling Tabbing on a Multiple-Page Form

In Chapter 15, you learned how to create a multiple-page form as one way to handle displaying more data than will fit on one page of a form on your computer screen. You also learned how to control simple tabbing on the form by setting the form's Cycle property to Current Page. One disadvantage of this approach is you can no longer use Tab or Shift+Tab to move to other pages or other records. You must use the Page Up and Page Down keys or the record selector buttons to do that. You can set the Cycle property to All Records to restore this capability, but some strange things happen if you don't add code to handle page alignment.

To see what happens, open the frmXmplContactsPages form in the Conrad Systems Contacts sample database (Contacts.accdb) from the Navigation pane. Press Page Down to move to the Home Address field for the first contact. Next, press Shift+Tab once (back tab). Your screen should look something like Figure 25-11.

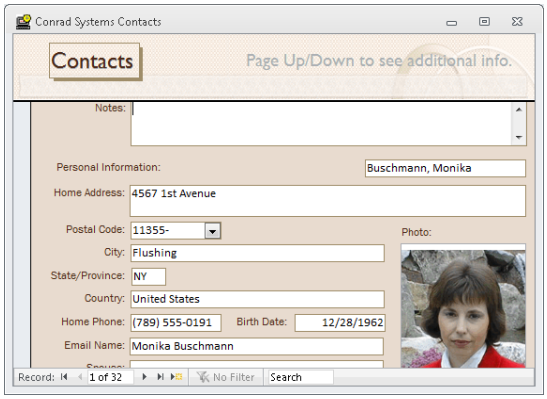

Figure 25-11 The form page doesn't align correctly when you back-tab from the Home Address field in frmXmplContactsPages.

If you leave the Cycle property set to All Records or Current Record, tabbing across page boundaries causes misalignment unless you add some code to fix it. What happens is that Access moves the form display only far enough to show the control you just tabbed to. (In this example, you're tabbing to the Notes text box control.) Open the sample frmContactsPages form that has the code to fix this problem and try the same exercise. You should discover that Shift+Tab places you in the Notes field, but the form scrolls up to show you the entire first page.

To allow tabbing across a page boundary while providing correct page alignment, you need event procedures in the Enter event for the first and last controls that can receive the focus on each page. If you examine the code behind the frmContactsPages form, you'll find these four procedures:

```
Private Sub ContactID_Enter()
    ' If tabbing forward into this field from previous record
    '  align page 1
    Me.GoToPage 1
End Sub

Private Sub HomeAddress_Enter()
    ' If tabbing forward into this field, align page 2
    Me.GoToPage 2
End Sub

Private Sub Notes_Enter()
    On Error Resume Next
    ' If tabbing backward into the last control on page 1, align it
    Me.GoToPage 1
End Sub
```

```
Private Sub Photo_Enter()
    On Error Resume Next
    ' If tabbing backward into the last control on page 2, align it
    Me.GoToPage 2
End Sub
```

This is arguably some of the simplest example code in any of the sample databases, but this attention to detail will make the users of your application very happy.

> **Note**
>
> **The code also executes when you tab backward into the ContactID or HomeAddress controls or forward into the Notes or Photo control, or you can click in any of the controls. Access realizes that the form is already on the page requested in each case, so it does nothing.**

Automating Data Selection

One of the most common tasks to automate in a database application is filtering data. Particularly when a database contains thousands of records, users will rarely need to work with more than a few records at a time. If your edit forms always display all the records, performance can suffer greatly. So it's a good idea to enable the user to easily specify a subset of records. This section examines four ways to do this.

Working with a Multiple-Selection List Box

You work with list boxes all the time in Windows and in Access. For example, the file list in Windows Explorer is a list box, the Access 2010 Navigation pane is a list box, and the list of properties on any tab in the property sheet is a list box. In the property list box, you can select only one property from the list at a time. If you click a different property, the previous object is no longer selected—this is a simple list box. In Windows Explorer, you can select one file, select multiple noncontiguous files by holding down the Ctrl key and clicking, or select a range of files by holding down the Shift key and clicking—this is a multiple-selection list box.

Suppose you're using the Conrad Systems Contacts application (Contacts.accdb) and you're interested in looking at the details for several contacts at one time but will rarely want to look at the entire list. Start the application by opening the frmSplash form, select John Viescas as the User Name, and click Sign On (no password required). Click the Contacts button on the main switchboard form, and the application opens the Select Contacts form (frmContactList). As shown in Figure 25-12, the frmContactList form contains a multiple-selection list box.

Note

You won't see the Select Contacts dialog box if the Don't Show Contact List option is selected in John's user profile. If the Contacts form opens when you click the Contacts button on the main switchboard, close the form and click Users. Clear the Don't Show Contact List option in John's profile, save the record, and close the form. You should now see the Select Contacts dialog box when you click the Contacts button on the main switchboard.

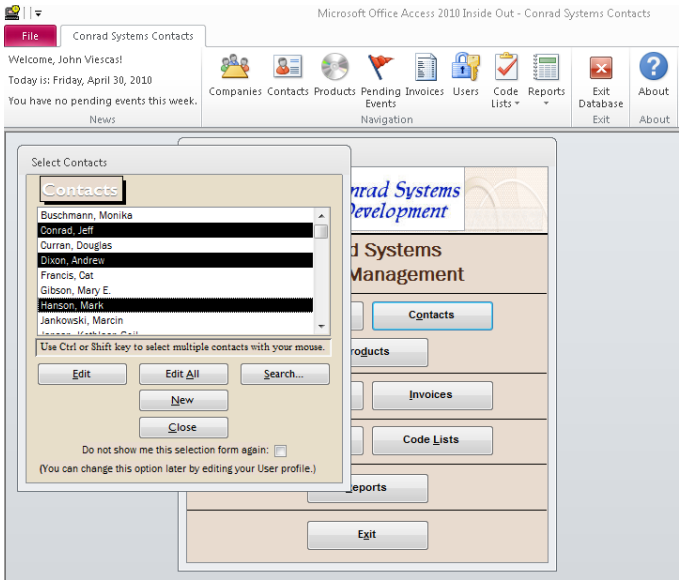

Figure 25-12 You can select multiple contact records to edit in the frmContactList form.

In this list box, the contacts are shown in alphabetic order by last name, and the list is bound to the ContactID field in the underlying table. You can edit any single contact by simply double-clicking the person's name. You can move the highlight up or down by using the arrow keys. You can also type the first letter of a contact's last name to jump to the next contact whose last name begins with that letter. You can hold down the Shift key and use the arrow keys to extend the selection to multiple names. Finally, you can hold down either the Shift key or the Ctrl key and use the mouse to select multiple names.

Figure 25-12 shows three contacts selected using the Ctrl key and the mouse. When you click Edit, the application opens the frmContacts form with only the records you selected. As shown in Figure 25-13, the caption to the right of the Record Number box indicates that there are three available records and that the recordset is filtered.

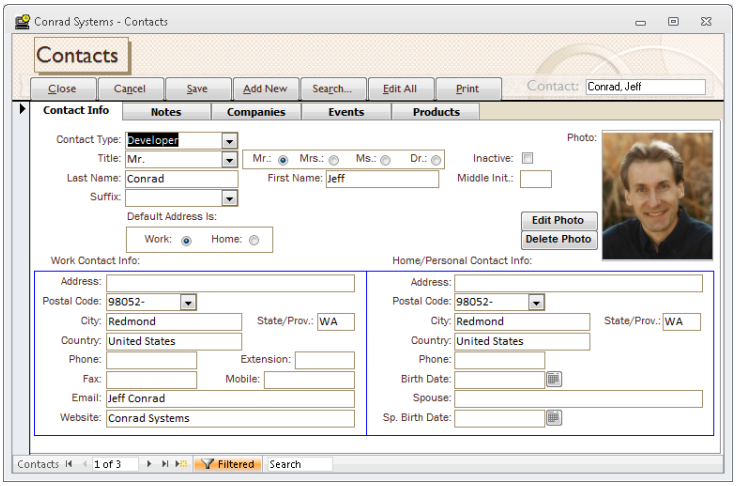

Figure 25-13 After you select the records you want to edit in the frmContactList, the application opens the frmContacts form displaying only those records.

To see how this works, you need to go behind the scenes of the frmContactList form. Click Exit on the main switchboard form to return to the Navigation pane. (Click Yes in the message box that asks "Are you sure you want to exit?" and click No if the application offers to create a backup for you.) Select frmContactList, and open the form in Design view, as shown in Figure 25-14. Click the list box control, and open its property sheet to see how the list box is defined. The list box uses two columns from the qlkpContacts query, hiding the ContactID (the primary key that will provide a fast lookup) in the first column and displaying the contact name in the second column. The key to this list box is that its Multi Select property is set to Extended. Using the Extended setting gives you the full Ctrl+click or Shift+click features that you see in most list boxes in Windows. The default for this property is None, which lets you select only one value at a time. You can set it to Simple if you want to select or clear multiple values using the mouse or the spacebar.

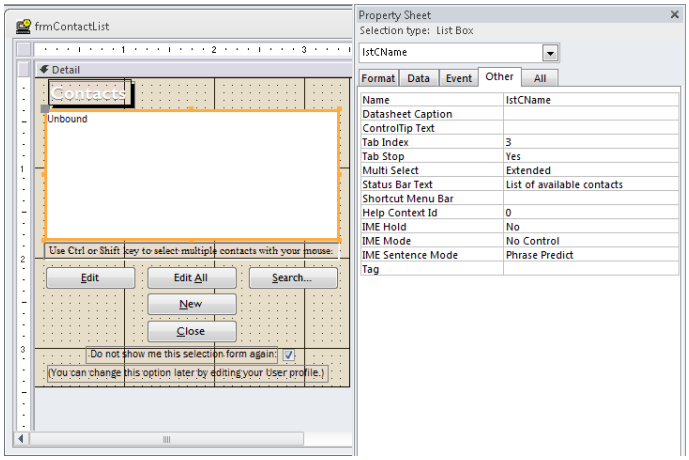

Figure 25-14 The multiple-selection list box on the frmContactList form has its Multi Select property set to Extended.

If you scroll down to the Event properties, you'll find an event procedure defined for On Dbl Click. The code for this event procedure (which is called when you double-click an item in the list box) runs only the cmdSome_Click procedure. Right-click the cmdSome command button (the one whose caption says *Edit*), and choose Build Event from the shortcut menu to jump to the cmdSome_Click procedure that does all the work, as shown here:

```
Private Sub cmdSome_Click()
Dim strWhere As String, varItem As Variant
  ' Request to edit items selected in the list box
  ' If no items selected, then nothing to do
  If Me!lstCName.ItemsSelected.Count = 0 Then Exit Sub
  ' Loop through the items selected collection
  For Each varItem In Me!lstCName.ItemsSelected
    ' Grab the ContactID column for each selected item
    strWhere = strWhere & Me!lstCName.Column(0, varItem) & ","
  Next varItem
  ' Throw away the extra comma on the "IN" string
  strWhere = Left$(strWhere, Len(strWhere) - 1)
  ' Open the contacts form filtered on the selected contacts
  strWhere = "[ContactID] IN (" & strWhere & ") And (Inactive = False)"
  DoCmd.OpenForm FormName:="frmContacts", WhereCondition:=strWhere
  DoCmd.Close acForm, Me.Name
End Sub
```

When you set the Multi Select property of a list box to something other than None, you can examine the control's ItemsSelected collection to determine what (if anything) is selected. In the cmdSome_Click procedure, the Visual Basic code first checks the Count property of the control's ItemsSelected collection to determine whether anything is selected. If the Count is 0, there's nothing to do, so the procedure exits.

The ItemsSelected collection is composed of variant values, each of which provides an index to a highlighted item in the list box. The For Each loop asks Visual Basic to loop through all the available variant values in the collection, one at a time. Within the loop, the code uses the value of the variant to retrieve the Contact ID from the list. List boxes also have a Column property, and you can reference all the values in the list by using a statement such as

```
Me.ListBoxName.Column(ColumnNum, RowNum)
```

where *ListBoxName* is the name of your list box control, *ColumnNum* is the relative column number (the first column is 0, the second is 1, and so on), and *RowNum* is the relative row number (also starting at 0). The variant values in the ItemsSelected collection return the relative row number. This Visual Basic code uses column 0 and the values in the ItemsSelected collection to append each selected ContactID to a string variable, separated by commas. You'll recall from studying the IN predicate in Chapter 9 that a list of values separated by commas is ideal for an IN clause.

After retrieving all the ContactID numbers, the next statement removes the trailing comma from the string. The final Where clause includes an additional criterion to display only active contacts. The DoCmd.OpenForm command uses the resulting string to create a filter clause as it opens the form. Finally, the code closes the frmContactList form. (Me.Name is the name of the current form.)

Providing a Custom Query By Form

Suppose you want to do a more complex search on the frmContacts form—using criteria such as contact type, company, or products owned rather than simply using contact name. You could teach your users how to use the Filter By Form features to build the search, or you could use Filter By Form to easily construct multiple OR criteria on simple tests. But if you want to find, for example, all contacts who own the Single User edition or whom you contacted between certain dates, there's no way to construct this request using standard filtering features. The reason for this is that when you define a filter for a subform (such as the Events subform in frmContacts) using Filter By Form, you're filtering only the subform rows. You're not finding contacts who have only a matching subform row.

The only solution, then, is to provide a custom Query By Form that provides options to search on all the important fields and then build the Where clause to solve the search problem using Visual Basic code. To start, open the Conrad Systems Contacts application. (If you have exited to the Navigation pane, you can start the application by opening frmSplash.) Sign on, click the Contacts button on the main switchboard form, and then click the Search button in the Select Contacts dialog box. You should see the fdlgContactSearch form, as shown in Figure 25-15.

Figure 25-15 You can design a custom Query By Form to perform a complex search.

Try selecting contacts whose last name begins with the letter *M*, whom you contacted between September 1, 2010, and December 15, 2010, and who own the BO$$ Single User product (from the Owns Product drop-down list). When you click Search, you should see the frmContacts form open and display two contacts.

To see how this works, you need to explore the design of the fdlgContactSearch form. Switch to the Navigation pane (by pressing F11), and open the form in Design view. You should see a window like Figure 25-16. Notice that the form is not bound to any record source. The controls must be unbound so they can accept any criteria values that a user might enter.

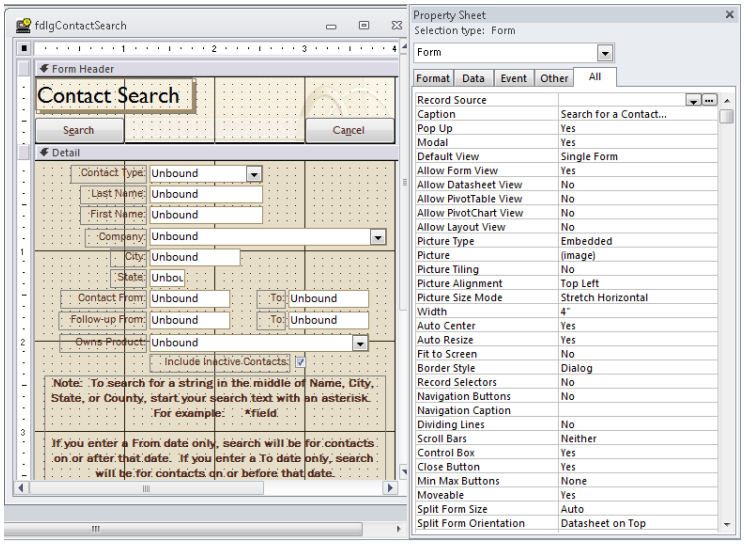

Figure 25-16 When you look at the fdlgContactSearch form in Design view, you can see that it has no record source.

The bulk of the work happens when you click Search. The code for the event procedure for the Click event of the Search button is shown here:

```
Private Sub cmdSearch_Click()
Dim varWhere As Variant, varDateSearch As Variant
Dim rst As DAO.Recordset
  ' Initialize to Null
  varWhere = Null
  varDateSearch = Null
  ' First, validate the dates
  ' If there's something in Contact Date From
  If Not IsNothing(Me.txtContactFrom) Then
    ' First, make sure it's a valid date
    If Not IsDate(Format(Me.txtContactFrom, "mm/dd/yyyy")) Then
      ' Nope, warn them and bail
      MsgBox "The value in Contact From is not a valid date.", _
        vbCritical, gstrAppTitle
      Exit Sub
    End If
    ' Now see if they specified a "to" date
    If Not IsNothing(Me.txtContactTo) Then
      ' First, make sure it's a valid date
      If Not IsDate(Format(Me.txtContactTo, "mm/dd/yyyy")) Then
        ' Nope, warn them and bail
        MsgBox "The value in Contact To is not a valid date.", _
          vbCritical, gstrAppTitle
        Exit Sub
```

```
    End If
    ' Got two dates, now make sure "to" is >= "from"
    If Format(Me.txtContactTo, "mm/dd/yyyy") < _
      Format(Me.txtContactFrom, "mm/dd/yyyy") Then
      MsgBox "Contact To date must be greater than " & _
        "or equal to Contact From date.", _
        vbCritical, gstrAppTitle
      Exit Sub
    End If
  End If
Else
  ' No "from" but did they specify a "to"?
  If Not IsNothing(Me.txtContactTo) Then
    ' Make sure it's a valid date
    If Not IsDate(Format(Me.txtContactTo, "mm/dd/yyyy")) Then
      ' Nope, warn them and bail
      MsgBox "The value in Contact To is not a valid date.", _
        vbCritical, gstrAppTitle
      Exit Sub
    End If
  End If
End If
' If there's something in Follow-up Date From
If Not IsNothing(Me.txtFollowUpFrom) Then
  ' First, make sure it's a valid date
  If Not IsDate(Format(Me.txtFollowUpFrom, "mm/dd/yyyy")) Then
    ' Nope, warn them and bail
    MsgBox "The value in Follow-up From is not a valid date.", _
      vbCritical, gstrAppTitle
    Exit Sub
  End If
  ' Now see if they specified a "to" date
  If Not IsNothing(Me.txtFollowUpTo) Then
    ' First, make sure it's a valid date
    If Not IsDate(Format(Me.txtFollowUpTo, "mm/dd/yyyy")) Then
      ' Nope, warn them and bail
      MsgBox "The value in Follow-up To is not a valid date.", _
        vbCritical, gstrAppTitle
      Exit Sub
    End If
    ' Got two dates, now make sure "to" is >= "from"
    If Format(Me.txtFollowUpTo, "mm/dd/yyyy") < _
      Format(Me.txtFollowUpFrom, "mm/dd/yyyy") Then
      MsgBox "Follow-up To date must be greater than " & _
        "or equal to Follow-up From date.", _
        vbCritical, gstrAppTitle
      Exit Sub
    End If
  End If
Else
  ' No "from" but did they specify a "to"?
  If Not IsNothing(Me.txtFollowUpTo) Then
    ' Make sure it's a valid date
```

```
    If Not IsDate(Format(Me.txtFollowUpTo, "mm/dd/yyyy")) Then
      ' Nope, warn them and bail
      MsgBox "The value in Follow-up To is not a valid date.", _
        vbCritical, gstrAppTitle
      Exit Sub
    End If
  End If
End If
' OK, start building the filter
' If specified a contact type value
If Not IsNothing(Me.cmbContactType) Then
  ' .. build the predicate
  varWhere = "(ContactType.Value = '" & Me.cmbContactType & "')"
End If
' Do Last Name next
If Not IsNothing(Me.txtLastName) Then
  ' .. build the predicate
  ' Note: taking advantage of Null propagation
  '  so we don't have to test for any previous predicate
  varWhere = (varWhere + " AND ") & "([LastName] LIKE '" & _
    Me.txtLastName & "*')"
End If
' Do First Name next
If Not IsNothing(Me.txtFirstName) Then
  ' .. build the predicate
  varWhere = (varWhere + " AND ") & "([FirstName] LIKE '" & _
    Me.txtFirstName & "*')"
End If
' Do Company next
If Not IsNothing(Me.cmbCompanyID) Then
  ' .. build the predicate
  ' Must use a subquery here because the value is in a linking table...
  varWhere = (varWhere + " AND ") & _
    "([ContactID] IN (SELECT ContactID FROM tblCompanyContacts " & _
    "WHERE tblCompanyContacts.CompanyID = " & Me.cmbCompanyID & "))"
End If
' Do City next
If Not IsNothing(Me.txtCity) Then
  ' .. build the predicate
  ' Test for both Work and Home city
  varWhere = (varWhere + " AND ") & "(([WorkCity] LIKE '" & _
    Me.txtCity & "*')" & _
    " OR ([HomeCity] LIKE '" & Me.txtCity & "*'))"
End If
' Do State next
If Not IsNothing(Me.txtState) Then
  ' .. build the predicate
  ' Test for both Work and Home state
  varWhere = (varWhere + " AND ") & "(([WorkStateOrProvince] LIKE '" & _
    Me.txtState & "*')" & _
    " OR ([HomeStateOrProvince] LIKE '" & Me.txtState & "*'))"
End If
```

```
' Do Contact date(s) next -- this is a toughie
'   because we want to end up with one filter on the subquery table
'   for both Contact Date range and FollowUp Date range
' Check Contact From first
If Not IsNothing(Me.txtContactFrom) Then
  ' .. build the predicate
  varDateSearch = "tblContactEvents.ContactDateTime >= #" & _
    Format(Me.txtContactFrom, "mm/dd/yyyy") & "#"
End If
' Now do Contact To
If Not IsNothing(Me.txtContactTo) Then
  ' .. add to the predicate, but add one because ContactDateTime includes
  '  a date AND a time
  varDateSearch = (varDateSearch + " AND ") & _
    "tblContactEvents.ContactDateTime < #" & _
    CDate(Format(Me.txtContactTo, "mm/dd/yyyy")) + 1 & "#"
End If
' Now do Follow-up From
If Not IsNothing(Me.txtFollowUpFrom) Then
  ' .. add to the predicate
  varDateSearch = (varDateSearch + " AND ") & _
    "tblContactEvents.ContactFollowUpDate >= #" & _
    Format(Me.txtFollowUpFrom, "mm/dd/yyyy") & "#"
End If
' Finally, do Follow-up To
If Not IsNothing(Me.txtFollowUpTo) Then
  ' .. add to the predicate
  varDateSearch = (varDateSearch + " AND ") & _
    "tblContactEvents.ContactFollowUpDate <= #" & _
    Format(Me.txtFollowUpTo, "mm/dd/yyyy") & "#"
End If
' Did we build any date filter?
If Not IsNothing(varDateSearch) Then
  ' OK, add to the overall filter
  ' Must use a subquery here because the value is in a linking table...
  varWhere = (varWhere + " AND ") & _
    "([ContactID] IN (SELECT ContactID FROM tblContactEvents " & _
    "WHERE " & varDateSearch & "))"
End If
' Do Product
If Not IsNothing(Me.cmbProductID) Then
  ' .. build the predicate
  ' Must use a subquery here because the value is in a linking table...
  varWhere = (varWhere + " AND ") & _
    "([ContactID] IN (SELECT ContactID FROM tblContactProducts " & _
    "WHERE tblContactProducts.ProductID = " & Me.cmbProductID & "))"
End If
' Finally, do the Inactive check box
If (Me.chkInactive = False) Then
  ' Build a filter to exclude inactive contacts
  varWhere = (varWhere + " AND ") & "(Inactive = False)"
End If
```

```
  ' Check to see that we built a filter
  If IsNothing(varWhere) Then
    MsgBox "You must enter at least one search criteria.", _
      vbInformation, gstrAppTitle
    Exit Sub
  End If
  ' Open a recordset to see if any rows returned with this filter
  Set rst = CurrentDb.OpenRecordset("SELECT * FROM tblContacts " & _
    "WHERE " & varWhere)
  ' See if found none
  If rst.RecordCount = 0 Then
    MsgBox "No Contacts meet your criteria.", vbInformation, gstrAppTitle
    ' Clean up recordset
    rst.Close
    Set rst = Nothing
    Exit Sub
  End If
  ' Hide me to fix later focus problems
  Me.Visible = False
  ' Move to last to find out how many
  rst.MoveLast
  ' If 5 or less or frmContacts already open,
  If (rst.RecordCount < 6) Or IsFormLoaded("frmContacts") Then
    ' Open Contacts filtered
    ' Note: if form already open, this just applies the filter
    DoCmd.OpenForm "frmContacts", WhereCondition:=varWhere
    ' Make sure focus is on contacts
    Forms!frmContacts.SetFocus
  Else
    ' Ask if they want to see a summary list first
    If vbYes = MsgBox("Your search found " & rst.RecordCount & _
      " contacts.  " & _
      "Do you want to see a summary list first?", _
      vbQuestion + vbYesNo, gstrAppTitle) Then
      ' Show the summary
      DoCmd.OpenForm "frmContactSummary", WhereCondition:=varWhere
      ' Make sure focus is on contact summary
      Forms!frmContactSummary.SetFocus
    Else
      ' Show the full contacts info filtered
      DoCmd.OpenForm "frmContacts", WhereCondition:=varWhere
      ' Make sure focus is on contacts
      Forms!frmContacts.SetFocus
    End If
  End If
  ' Done
  DoCmd.Close acForm, Me.Name
  ' Clean up recordset
  rst.Close
  Set rst = Nothing
End Sub
```

The first part of the procedure validates the contact date's from and to values and the follow up date's from and to values. If any are not valid dates, or the from date is later than the to date, the code issues an appropriate warning message and exits.

The next several segments of code build up a WHERE string by looking at the unbound controls one at a time. If the corresponding field is a string, the code builds a test using the LIKE predicate so that whatever the user enters can match any part of the field in the underlying table, but not all the fields are strings. When the function adds a clause as it builds the WHERE string, it inserts the AND keyword between clauses if other clauses already exist. Because the variable containing the WHERE clause is a Variant data type initialized to Null, the code can use a + concatenation to optionally add the AND keyword. Note that because the ContactType field is a multi-value field, the code specifically searches the Value property of the field.

The underlying record source for the frmContacts form does not include either contact event or product information directly, so the procedure has to build a predicate using a subquery if you ask for a search by contact date, follow-up date, or product. In the case of contact date or follow-up date, the code builds a separate filter string (varDateSearch) because both fields are in the same table (tblContactEvents). If you ask for any date range check, the code builds criteria using a subquery that finds the ContactID from records in the tblContactEvents table that fall within the date range. For a search by product, the code builds criteria using a subquery that finds the ContactID from records in the tblContactProducts table that match the product you selected. Finally, if you leave the Include Inactive Contacts check box cleared, the code adds a test to include only records that are active.

After examining all the possible filter values the user could have entered, the code checks to see if there's anything in the filter string (varWhere). There's no point in opening the form without a filter, so the code displays a message and exits, leaving the form open to allow the user to try again.

The final part of the procedure builds a simple recordset on the tblContacts table used in both the frmContacts and frmContactSummary forms, applying the WHERE clause built by the code in the first part of the procedure. If it finds no records, it uses the MsgBox function to inform the user and then gives the user a chance to try again.

When you first open a Recordset object in code, its RecordCount property is 0 if the recordset is empty and is some value greater than 0 if the recordset contains some records. The RecordCount property of a Recordset object contains only a count of the number of rows visited and not the number of rows in the recordset. Therefore, if it finds some rows, the procedure moves to the last row in the temporary recordset to get an accurate count. When the record count is greater than 5 and the frmContacts form is not already open, the procedure uses the MsgBox function to give the user the option to view a summary of the records found in the frmContactSummary form or to display the records found directly in

the frmContacts form. (As noted earlier, both forms use the same record source, so the code can apply the filter it built as it opens either form.) We'll examine how the frmContactSummary form works in the next section.

Selecting from a Summary List

As you saw in the cmdSearch_Click procedure in the previous section, the user gets to make a choice if more than five rows meet the entered criteria. To examine this feature in more detail, make sure the frmContacts form is not open, and ask for a search of contacts with a Contact Type of Customer in the fdlgContactSearch form. The result should look like Figure 25-17, in which 30 contacts are categorized as customers.

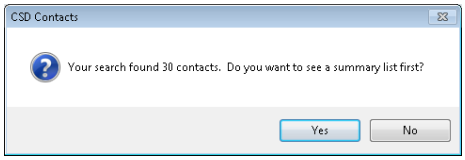

Figure 25-17 This message box appears when the cmdSearch_Click procedure returns more than five rows.

If you click Yes, the cmdSearch_Click procedure opens the Contact Search Summary form (frmContactSummary), as shown in Figure 25-18. You can scroll down to any row, put the focus on that row (be sure the row selector indicator is pointing to that row), and then click View Details to open the frmContacts form and view the details for the one contact you selected. You can see that this is a very efficient way to help the user narrow down a search to one particular contact.

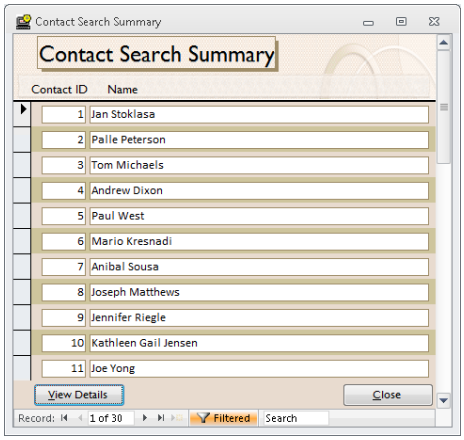

Figure 25-18 You can select a specific contact from the search summary form.

You can also double-click either the Contact ID or the Name field to see the details for that contact. Because this list is already filtered using the criteria you specified in the fdlgContactSearch form, the code that responds to your request builds a simple filter on Contact ID to make opening the frmContacts form most efficient. The code behind this form, which responds to your request in the Click event of the Details command button, is as follows:

```
Private Sub Details_Click()
Dim strFilter As String
  ' They asked for details (or double-clicked one of the controls)
  ' Set up the filter
  strFilter = "(ContactID = " & Me.ContactID & ")"
  ' Open contacts filtered on the current row
  DoCmd.OpenForm FormName:="frmContacts", WhereCondition:=strFilter
  ' Close me
  DoCmd.Close acForm, Me.Name
  ' Put focus on contacts
  Forms!frmContacts.SetFocus
End Sub
```

Filtering One List with Another

You might have noticed when editing products on the Products tab in the frmContacts form (see Figure 25-1) that you can first choose a product type to narrow down the list of products and then choose the product you want. There are only 11 products in the sample application, so being able to narrow down the product selection first isn't all that useful, but you can imagine how a feature like this would be absolutely necessary in an application that had thousands of products available for sale.

The secret is that the row source for the Product combo box is a parameter query that filters the products based on the product type you chose. When you use this technique in a form in Single Form view, all you need to do is requery the filtered combo box (in this case, the Product combo box) when the user moves to a new record (in the Current event of the form) and requery when the user chooses a different value in the combo box that provides the filter value (in the AfterUpdate event of the combo box providing the filter value).

However, using this technique on a form in Continuous Forms view is considerably more complex. Even though you can see multiple rows in Continuous Forms view, there is actually only one copy of each control on the form. If you always requery the Product combo box each time you move to a new row, the product name displayed in other rows that have a different product type will appear blank. When the value in a row doesn't match a value in the list, you get a blank result, not the actual value of the field.

The way to solve this problem is to include the display name in the recordset for the form and carefully overlay each combo box with a text box that always displays the correct value regardless of the filter. You can open the fsubContactProducts form in Design view to see

how we did this. Figure 25-19 shows you the form with the two overlay text boxes (CategoryDescription and ProductName) pulled down from the underlying combo boxes (Unbound and ProductID).

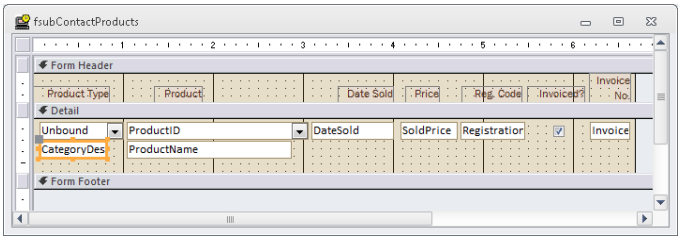

Figure 25-19 You can solve a filtered combo box display problem by overlaying text boxes.

Notice that the control source of the Product combo box is actually the ProductID field, but the combo box displays the ProductName field. Also, the Product Type combo box isn't bound to any field at all—there is no CategoryDescription field in tblContactProducts—but it does display the CategoryDescription field from the lookup table. To make this work, you need to include the ProductName and CategoryDescription fields in the record source for this form. You don't want the user to update these values, but you need them to provide the overlay display. These two text boxes have their Locked property set to Yes to prevent updating and their Tab Stop property set to No so that the user will tab into the underlying combo boxes and not these text boxes. Figure 25-20 shows you the qryContactProducts query that's the row source for this form.

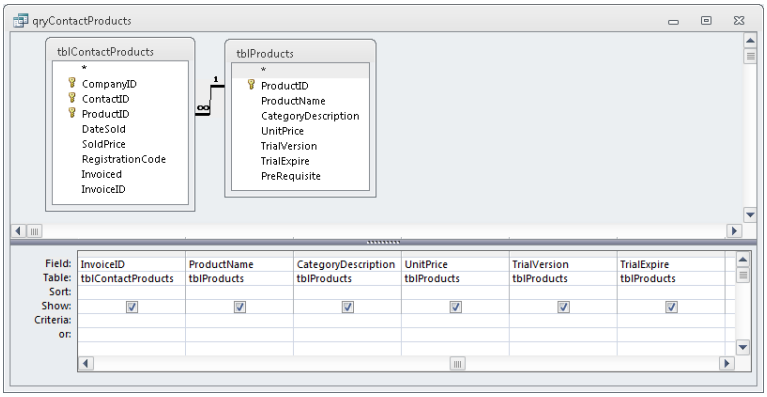

Figure 25-20 The qryContactProducts query provides the necessary ProductName and CategoryDescription fields from a related table so that you can display the values.

To make it all work correctly, several event procedures make sure that the focus goes where necessary and that the filtered Product combo box gets re-queried correctly. The code behind the fsubContactProducts form that does this is as follows:

```
Private Sub CategoryDescription_GotFocus()
  ' We have some tricky "overlay" text boxes here that
  ' shouldn't get the focus.  Move focus to the underlying
  ' combo box if that happens.
  Me.cmbCategoryDescription.SetFocus
End Sub

Private Sub cmbCategoryDescription_AfterUpdate()
  ' If they pick a new Category, then requery the
  '  product list that's filtered on category
  Me.cmbProductID.Requery
  ' Set the Product to the first row in the new list
  Me.cmbProductID = Me.cmbProductID.ItemData(0)
  ' .. and signal Product after update.
  cmbProductID_AfterUpdate
End Sub

Private Sub Form_Current()
  ' If we have a valid Category Description on this row...
  If Not IsNothing(Me.CategoryDescription) Then
    ' Then make sure the unbound combo is in sync.
    Me.cmbCategoryDescription = Me.CategoryDescription
  End If
  ' Requery the product list to match the current category
  Me.cmbProductID.Requery
  If (Me.Invoiced = True) Then
      Me.cmbProductID.Locked = True
      Me.cmbCategoryDescription.Locked = True
      Me.DateSold.Locked = True
      Me.SoldPrice.Locked = True
      Me.RegistrationCode.Locked = True
  Else
      Me.cmbProductID.Locked = False
      Me.cmbCategoryDescription.Locked = False
      Me.DateSold.Locked = False
      Me.SoldPrice.Locked = False
      Me.RegistrationCode.Locked = False
  End If
End Sub

Private Sub ProductName_GotFocus()
    ' We have some tricky "overlay" text boxes here that
    ' shouldn't get the focus.  Move focus to the underlying
    ' combo box if that happens.
    Me.cmbProductID.SetFocus
End Sub
```

As expected, the code re-queries the Product combo box whenever you pick a new category (cmbCategoryDescription_AfterUpdate) or when you move to a new row (Form_Current). It also keeps the unbound combo box in sync as you move from row to row so long as the underlying record has a valid category. (A new record won't have a related CategoryDescription until you choose a Product ID, so the code doesn't update the unbound combo box on a new record.) Finally, if you try to click in CategoryDescription or ProductName, the GotFocus code moves you to the underlying combo box where you belong. Why didn't we simply set the Enabled property for CategoryDescription and ProductName to No? If you do that, then you can't ever click into the category or product combo boxes because the disabled text box overlaid on top would block you.

> **Note**
>
> **If you want to see what the filtered combo box looks like without the overlay, make a backup copy of Contacts.accdb, open the fsubContactProducts form in Design view, move the Category Description and Product Name text boxes down similar to Figure 25-19, and save the form. Now, open the frmContacts form and click the Products tab.**

Linking to Related Data in Another Form or Report

Now that you know how to build a filter to limit what the user sees, you can probably surmise that using a filter is a good way to open another form or report that displays information related to the current record or set of filtered records in the current form. This section shows you how to do this for both forms and reports. Later in this section, you will learn how to use events in class modules to build sophisticated links.

Linking Forms Using a Filter

You've already seen the frmContactSummary form (see Figure 25-18) that uses a simple filter to link from the record selected in that form to full details in the frmContacts form. You can find similar code behind the fsubCompanyContacts form used as a subform in the frmCompanies form. Figure 25-21 shows you the frmCompanies form and the Edit This buttons we provided on the subform.

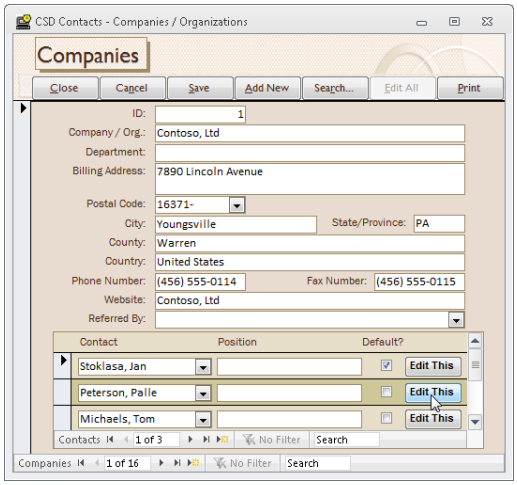

Figure 25-21 You can provide a link from the Companies / Organizations form to details about a particular contact.

To see the details for a particular contact, the user clicks the Edit This button on the chosen contact record, and code opens the frmContacts form with that contact displayed. The code behind the button is as follows:

```
Private Sub cmdEdit_Click()
    ' Open Contacts on the related record
    DoCmd.OpenForm "frmContacts", WhereCondition:="ContactID = " & Me.ContactID
End Sub
```

And code in the form's Current event prevents the user from clicking the button when in a new record that doesn't have a contact ID, as shown here:

```
Private Sub Form_Current()
    ' Disable "edit this" if on a new row
    Me.cmdEdit.Enabled = Not (Me.NewRecord)
End Sub
```

Setting the button's Enabled property to False causes it to appear dimmed, and the user cannot click the button.

Linking to a Report Using a Filter

Now, let's look at using the Filter technique to link to related information in a report. Open the frmInvoices form in the Conrad Systems Contacts application (Contacts.accdb) and move to an invoice that looks interesting (we used invoice number 50). Click Print to open

the Print Invoices form (fdlgInvoicePrintOptions) that gives you the option to see the current invoice formatted in a report, display all unprinted invoices in a report, display only unprinted invoices for the current customer, or print all invoices currently shown in the frmInvoices form. (You can use Search to filter the displayed invoices to the ones you want.) Select the Current Invoice Only option and click Print again to see the invoice in a report, as shown in Figure 25-22. (The figure shows you the sequence you see after clicking the Print button on the frmInvoices form. The Print Invoices dialog box closes after opening the report.)

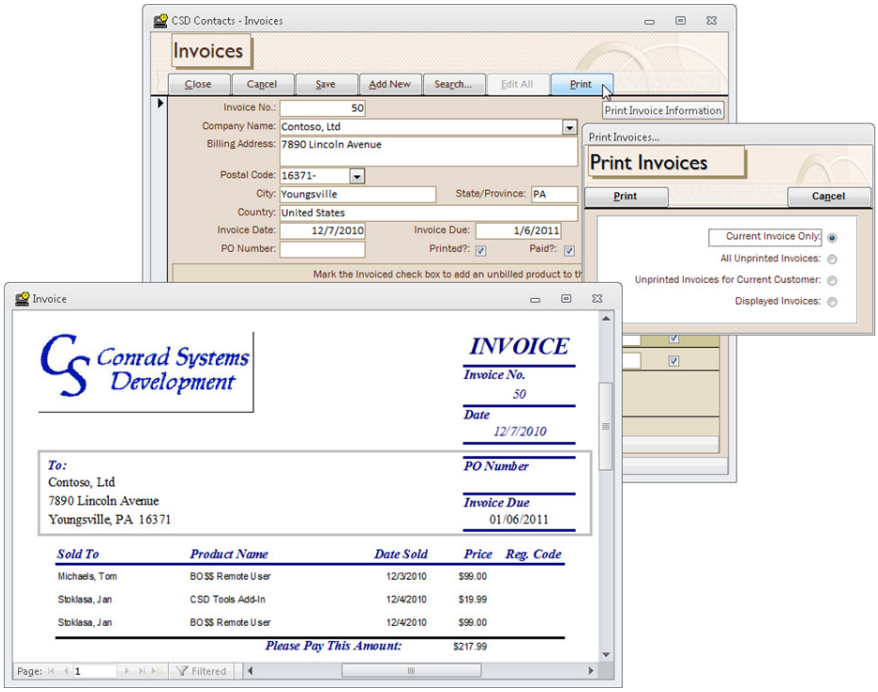

Figure 25-22 You can ask to print only the current invoice in the Conrad Systems Contacts database.

The code from the Click event of the Print button in the fdlgInvoicePrintOptions form is as follows:

```
Private Sub cmdPrint_Click()
Dim strFilter As String, frm As Form
  ' Set an error trap
  On Error GoTo cmdPrint_Error
  ' Get a pointer to the Invoices form
  Set frm = Forms!frmInvoices
  Select Case Me.optFilterType.Value
```

```
      ' Current Invoice
      Case 1
        ' Set filter to open Invoice report for current invoice only
        strFilter = "[InvoiceID] = " & frm!InvoiceID
      ' All unprinted invoices
      Case 2
        ' Set filter to open all unprinted invoices
        strFilter = "[InvoicePrinted] = 0"
      ' Unprinted invoices for current company
      Case 3
        ' Set filter to open unprinted invoices for current company
        strFilter = "[CompanyID] = " & frm!cmbCompanyID & _
          " AND [InvoicePrinted] = 0"
      ' Displayed invoices (if filter set on form)
      Case 4
        ' Check for a filter on the form
        If IsNothing(frm.Filter) Then
          ' Make sure they want to print all!
          If vbNo = MsgBox("Your selection will print all " & _
            "Invoices currently in the " & _
            "database.  Are you sure you want to do this?", _
            vbQuestion + vbYesNo + vbDefaultButton2, _
            gstrAppTitle) Then
            Exit Sub
          End If
          ' Set "do them all" filter
          strFilter = "1 = 1"
        Else
          strFilter = frm.Filter
        End If
    End Select
    ' Hide me
    Me.Visible = False
    ' Have a filter now.  Open the report on that filter
    DoCmd.OpenReport "rptInvoices", acViewPreview, , strFilter
    ' Update the Print flag for selected invoices
    CurrentDb.Execute "UPDATE tblInvoices SET InvoicePrinted = -1 WHERE " & _
      strFilter
    ' Refresh the form to show updated Printed status
    frm.Refresh
    ' Execute the Current event on the form to make sure it is locked correctly
    frm.Form_Current
cmdPrint_Exit:
    ' Clear the form object
    Set frm = Nothing
    ' Done
    DoCmd.Close acForm, Me.Name
    Exit Sub
cmdPrint_Error:
    ' Got an error
    ' If Cancel, that means the filter produced no Invoices
    If Err = errCancel Then
```

```
    ' Exit - report will display "no records" message
    Resume cmdPrint_Exit
  End If
  ' Got unknown error - display and log
  MsgBox "Unexpected error while printing and updating print flags: " & _
    Err & ", " & _
    Error, vbCritical, gstrAppTitle
  ErrorLog Me.Name & "_Print", Err, Error
  Resume cmdPrint_Exit
End Sub
```

This first part of this procedure sets an object reference to the frmInvoices form to make it easy to grab either the InvoiceID or the CompanyID and to reference properties and methods of the form's object. The Select Case statement examines which option button the user selected on fdlgInvoicePrintOption and builds the appropriate filter for the report. Notice that if the user asks to print all the invoices currently displayed on the form, the code first looks for a user-applied filter on the frmInvoices form. If the code finds no filter, it asks if the user wants to print all invoices. The code uses the filter it built (or the current filter on the frmInvoices form) to open the rptInvoices report in Print Preview. It also executes a Structured Query Language (SQL) UPDATE statement to flag all the invoices the user printed. If you look at code in the Current event of the frmInvoices form, you'll find that it locks all controls so that the user can't update an invoice that has been printed.

Synchronizing Two Forms Using a Class Event

Sometimes it's useful to give the user an option to open a pop-up form that displays additional details about some information displayed on another form. As you move from one row to another in the main form, it would be nice if the form that displayed the additional information stayed in sync.

Of course, the Current event of a form lets you know when you move to a new row. In the Wedding List sample database built with macros (WeddingListMC.accdb), the macros do some elaborate filtering to keep a pop-up form with additional city information in sync with the main form. However, doing it with macros is the hard way!

The primary Wedding List sample application is in WeddingList.accdb, and it uses Visual Basic to provide all the automation. With Visual Basic, we were able to declare and use a custom event in the WeddingList form to signal the CityInformation form if it's open and responding to the events. In the Current event of the WeddingList form, we don't have to worry about whether the companion form is open. The code simply signals the event and lets the City Information form worry about keeping in sync with the main form. (The user can open the City Information form at any time by clicking the City Info button on the Wedding List form.) You can see these two forms in action in Figure 25-23.

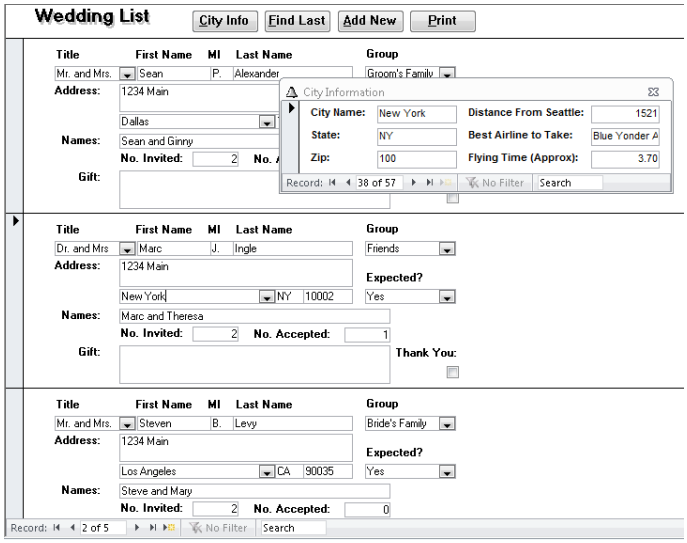

Figure 25-23 The CityInformation form pops open over the main WeddingList form to display additional information about the invitee's home city.

Here's the code from the WeddingList form class module that makes an event available to signal the CityInformation form:

```
Option Compare Database
Option Explicit
' Event to signal we've moved to a new city
Public Event NewCity(varCityName As Variant)
' End of Declarations Section

Private Sub Form_Current()
On Error GoTo Form_Current_Err
  ' Signal the city form to move to this city
  ' and pass the city name to the event
  RaiseEvent NewCity(Me!City)
Form_Current_Exit:
  Exit Sub
Form_Current_Err:
  MsgBox Error$
  Resume Form_Current_Exit
End Sub

Private Sub cmdCity_Click()
On Error GoTo cmdCity_Click_Err
  ' If the city form is not open, open it
  If Not IsFormLoaded("CityInformation") Then
    DoCmd.OpenForm "CityInformation", acNormal, , , acFormReadOnly, acHidden
    ' Give the other form a chance to "hook" our event
```

```
    DoEvents
  End If
  ' Signal the form we just opened
  RaiseEvent NewCity(Me!City)
cmdCity_Click_Exit:
  Exit Sub
cmdCity_Click_Err:
  MsgBox Error$
  Resume cmdCity_Click_Exit
End Sub
```

In the Declarations section of the module, we declared an event variable and indicated that we're going to pass a parameter (the city name) in the event. In the Form_Current event procedure, the code uses RaiseEvent to pass the current city name to any other module that's listening. The code doesn't have to worry about whether any other module is interested in this event—it just signals the event when appropriate and then ends. (This is not unlike how Access works. When a form moves to a new record, Access signals the Form_Current event, but nothing happens unless you have written code to respond to the event.) Note that the variable passed is declared as a Variant to handle the case when the user moves to the new row at the end—the City control will be Null in that circumstance. A command button (cmdCity) on the WeddingList form allows the user to open the CityInformation form. The Click event of that button opens the form hidden and uses the DoEvents function to give the CityInformation form a chance to open and indicate that it wants to listen to the NewCity event on the WeddingList form. After waiting for the CityInformation form to finish processing, the code raises the event to notify that form about the city in the current row.

The CityInformation form does all the work (when it's open) to respond to the event signaled by the WeddingList form and move to the correct row. The code is shown here:

```
Option Compare Database
Option Explicit
Dim WithEvents frmWedding As Form_WeddingList
' End of the Declarations Section

Private Sub Form_Load()
On Error GoTo Form_Load_Err
  ' If the wedding list form is open
  If IsLoaded("WeddingList") Then
    ' Then set to respond to the NewCity event
    Set frmWedding = Forms!WeddingList
  End If
Form_Load_Exit:
  Exit Sub
Form_Load_Err:
  MsgBox Error$
  Resume Form_Load_Exit
End Sub
```

```
Private Sub frmWedding_NewCity(varCityName As Variant)
  ' The Wedding List form has asked us to move to a
  ' new city via the NewCity event
  On Error Resume Next
  If IsNothing(varCityName) Then
    ' Hide me if city name is empty
    Me.Visible = False
  Else
    ' Reveal me if there's a city name, and go
    ' find it
    Me.Visible = True
    Me.Recordset.FindFirst "[CityName] = """ & _
      varCityName & """"
  End If
End Sub
```

In the Declarations section, you can find an object variable called frmWedding that has a data type equal to the class module name of the WeddingList form. The WithEvents keyword indicates that code in this class module will respond to events signaled by any object assigned to this variable. When the form opens, the Form_Load procedure checks to see that the WeddingList form is open (just in case you opened this form by itself from the Navigation pane). If the WeddingList form is open, it "hooks" the NewCity event in that form by assigning it to the frmWedding variable.

The frmWedding_NewCity procedure responds to the NewCity event of the frmWedding object. Once the Load event code establishes frmWedding as a pointer to the WeddingList form, this procedure runs whenever code in the class module for that form signals the NewCity event with RaiseEvent.

The code in the event procedure is pretty simple. If the CityName parameter passed by the event is "nothing" (Null or a zero-length string), the procedure hides the form because there's nothing to display. If the event passes a valid city name, the procedure uses the FindFirst method of the Recordset object of this form to move to the correct city.

> **Note**
>
> **The Recordset property of a form in an Access database (.accdb file) returns a Data Access Objects (DAO) recordset in Access 2010. For this reason, you should use a DAO FindFirst method, not an ADO Find method, to locate rows in a form recordset.**

Automating Complex Tasks

The most complex Visual Basic code we've examined thus far in this chapter is the procedure to build a search clause from the data you enter in the fdlgContactSearch form. Trust us—we've only started to scratch the surface.

Triggering a Data Task from a Related Form

One of the more complex pieces of code in the Conrad Systems Contacts sample database is triggered from the fsubContactEvents form that's part of the frmContacts form. After signing in correctly to the application, the user can open the frmContacts form, click the Events tab, and add an event indicating the sale of a product. As soon as the user saves the record, code behind the subform automatically adds the product to the contact, as shown in Figure 25-24.

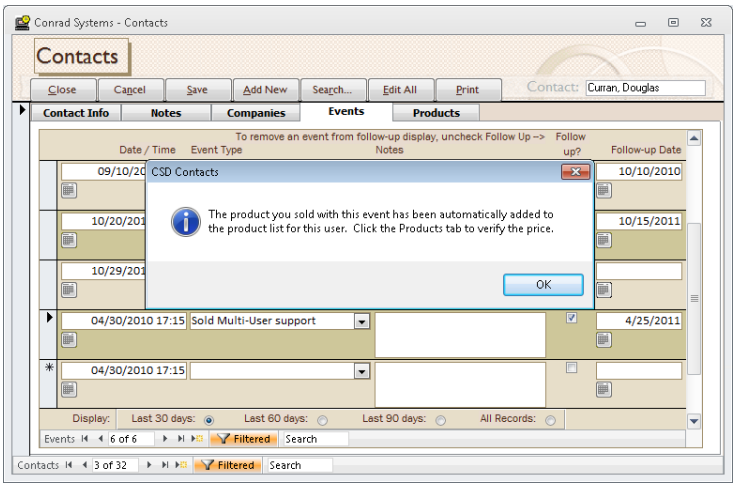

Figure 25-24 Logging a product sale event on the Events tab automatically sells the product to the contact.

If you look behind the fsubContactEvents form, you'll find event procedures that detect when the user has created a sale event and execute an SQL INSERT command to create the related product row. The code is as follows:

```
Option Compare Database
Option Explicit
' Flag to indicate auto-add of a product if new event requires it
Dim intProductAdd As Integer
' Place to store Company Name on a product add
Dim varCoName As Variant
```

```
' End of the Declarations Section

Private Sub ContactEventTypeID_BeforeUpdate(Cancel As Integer)
  ' Did they pick an event that involves a software sale?
  ' NOTE: All columns in a combo box are TEXT
  If Me.ContactEventTypeID.Column(4) = "-1" Then
    ' Try to lookup this contact's Company Name
    varCoName = DLookup("CompanyName", "qryContactDefaultCompany", _
      "ContactID = " & Me.Parent.ContactID.Value)
    ' If not found, then disallow product sale
    If IsNothing(varCoName) Then
      MsgBox "You cannot sell a product to a Contact " & _
        "that does not have a " & _
        "related Company that is marked as the default for this Contact." & _
        "  Press Esc to clear your edits and click on the Companies tab " & _
        "to define the default Company for this Contact.", _
        vbCritical, gstrAppTitle
      Cancel = True
    End If
  End If
End Sub

Private Sub Form_BeforeUpdate(Cancel As Integer)
  ' Did they pick an event that involves a software sale?
  ' NOTE: All columns in a combo box are TEXT
  If Me.ContactEventTypeID.Column(4) = "-1" Then
    ' Do this only if on a new record or they changed the EventID value
    If (Me.NewRecord) Or (Me.ContactEventTypeID <> _
      Me.ContactEventTypeID.OldValue) Then
      ' Set the add product flag
      '- product added by AfterUpdate code for safety
      intProductAdd = True
    End If
  End If
End Sub

Private Sub Form_AfterUpdate()
Dim strSQL As String, curPrice As Currency,
Dim lngProduct As Long, varCoID As Variant
Dim rst As DAO.Recordset, strPreReqName As String
  ' See if we need to auto-add a product
  If (intProductAdd = True) Then
    ' Reset so we only do this once
    intProductAdd = False
    ' Set an error trap
    On Error GoTo Insert_Err
    ' Save the Product ID
    lngProduct = Me.ContactEventTypeID.Column(5)
    ' Fetch the product record
    Set rst = CurrentDb.OpenRecordset("SELECT * FROM tblProducts " & _
      "WHERE ProductID = " & lngProduct)
    ' Make sure we got a record
```

```
If rst.EOF Then
  MsgBox "Could not find the product record for this sales event." & _
    "  Auto-create of " & _
    "product record for this contact has failed.", _
    vbCritical, gstrAppTitle
  rst.Close
  Set rst = Nothing
  GoTo Insert_Exit
End If
' Check for prerequisite product
If Not IsNull(rst!PreRequisite) Then
  ' Make sure contact owns the prerequisite product
  If IsNull(DLookup("ProductID", "tblContactProducts", _
    "ProductID = " & rst!PreRequisite & " And ContactID = " & _
    Me.Parent.ContactID)) Then
    ' Get the name of the prerequisite
    strPreReqName = DLookup("ProductName", "tblProducts", _
      "ProductID = " & rst!PreRequisite)
    ' Display error
    MsgBox "This contact must own prerequisite product " & _
      strPreReqName & " before you can sell this product." & _
      vbCrLf & vbCrLf & _
      "Auto-create of product record for this contact has failed", _
      vbCritical, gstrAppTitle
    ' Bail
    rst.Close
    Set rst = Nothing
    GoTo Insert_Exit
  End If
End If
' Save the price
curPrice = rst!UnitPrice
' Done with the record - close it
rst.Close
Set rst = Nothing
' Now, find the default company for this contact
varCoID = DLookup("CompanyID", "qryContactDefaultCompany", _
  "ContactID = " & Me.Parent.ContactID.Value)
' If not found, then disallow product sale
If IsNothing(varCoID) Then
  MsgBox "You cannot sell a product to a Contact who does not have a " & _
    "related Company that is marked as the default for this Contact.", _
    vbCritical, gstrAppTitle
  GoTo Insert_Exit
End If
' Set up the INSERT command
strSQL = "INSERT INTO tblContactProducts " & _
  "(CompanyID, ContactID, ProductID, DateSold, SoldPrice) " & _
  "VALUES(" & varCoID & ", " & Me.Parent.ContactID & ", " & _
  lngProduct & ", #" & _
  DateValue(Format(Me.ContactDateTime, "mm/dd/yyyy")) & "#, " & _
  curPrice & ")"
```

```
    ' Attempt to insert the Product row
    CurrentDb.Execute strSQL, dbFailOnError
    ' Got a good add - inform the user
    MsgBox "The product you sold with this event " & _
      "has been automatically added " & _
      "to the product list for this user.  " & _
      "Click the Products tab to verify the price.", _
      vbInformation, gstrAppTitle
    ' Requery the other subform to get the new row there
    Me.Parent.fsubContactProducts.Requery
  End If
Insert_Exit:
  Exit Sub
Insert_Err:
  ' Was error a duplicate row?
  If Err = errDuplicate Then
    MsgBox "CSD Contacts attempted to auto-add " & _
      "the product that you just indicated " & _
      "that you sold, but the Contact appears " & _
      "to already own this product.  Be sure " & _
      "to verify that you haven't tried to sell the same product twice.", _
      vbCritical, gstrAppTitle
  Else
    MsgBox "There was an error attempting to auto-add " & _
      "the product you just sold: " & _
      Err & ", " & Error, vbCritical, gstrAppTitle
    ' Log the error
    ErrorLog Me.Name & "_FormAfterUpdate", Err, Error
  End If
  Resume Insert_Exit
End Sub
```

In the Declarations section of the module, you can find two variables that the event procedures use to pass information between events. (If you declare the variables inside one of the procedures, only that procedure can use the variables.) The BeforeUpdate event procedure for the contact event type checks to see if the event is a product sale (by examining one of the hidden columns in the combo box row source). If the user is trying to log a product sale and this particular contact doesn't have a default company defined, the code displays an error message and won't let the user save that event type. Remember, a record in the tblContactProducts table must have a CompanyID as well as a ContactID.

When the user attempts to save a new or changed event record, Access runs the form's BeforeUpdate event procedure. This code again checks to see if the record about to be saved is for a product sale. However, if this isn't a new record or the user is saving an old event record but didn't change the event type, the code exits because it doesn't want to add a product record twice. (If this is an existing record and the event type didn't change, this code probably created the companion contact product record the first time the user saved the record.) The code could insert the record into tblContactProducts at this point,

but, as you learned in Chapter 19, the record isn't really saved until after the BeforeUpdate event completes. Therefore, this code sets the module variable to tell the form's AfterUpdate event procedure to perform that task after Access has saved the changed record.

After Access saves the new or changed event record, it runs the form's AfterUpdate event procedure. If the code in BeforeUpdate indicated that a product insert is required by setting the module intProductAdd variable to True, this code sets up to add the new record. It opens a recordset on the tblProducts table for the product that was just sold so that it can get the product price and check for any prerequisite product. If the product has a prerequisite but this contact doesn't own the prerequisite, the code displays an error message and exits.

Although previous code checked to see that this contact has a default CompanyID, this code checks again and exits if it can't find one. After the code has completed all checks and has the price and company ID information it needs, it inserts the new record into the tblContactProducts table using SQL. Notice that at the bottom of the procedure, you can find error-trapping code that tests to see if the insert caused a duplicate record error.

Linking to a Related Task

Let's switch to the Housing Reservations application (Housing.accdb) and take a look at the process for confirming a room for a reservation request. To see this in action, you must start the application by opening the frmSplash form, and then sign on as an administrator (Conrad, Jeff; Richins, Jack S.; Schare, Gary; or Viescas, John L.) using **password** as the password. On the main switchboard, click Reservation Requests, and then click View Unbooked in the Edit Reservation Requests dialog box. You'll see the Unbooked Requests form (frmUnbookedRequests), as shown in Figure 25-25.

Note

The query that provides the records displayed in the frmUnbookedRequests form includes criteria to exclude any requests that have a check-in date earlier than today's date. (It doesn't make sense to confirm a reservation request for a date in the past.) The latest requested check-in date in the original database is March 11, 2011, so you will probably see an error message when you attempt to look at unbooked requests. You can use the zfrmLoadData form to load new reservations and requests that are more current into the qryUnbookedRequests query not to eliminate old requests, to be able to see how the frmUnbookedRequests form works.

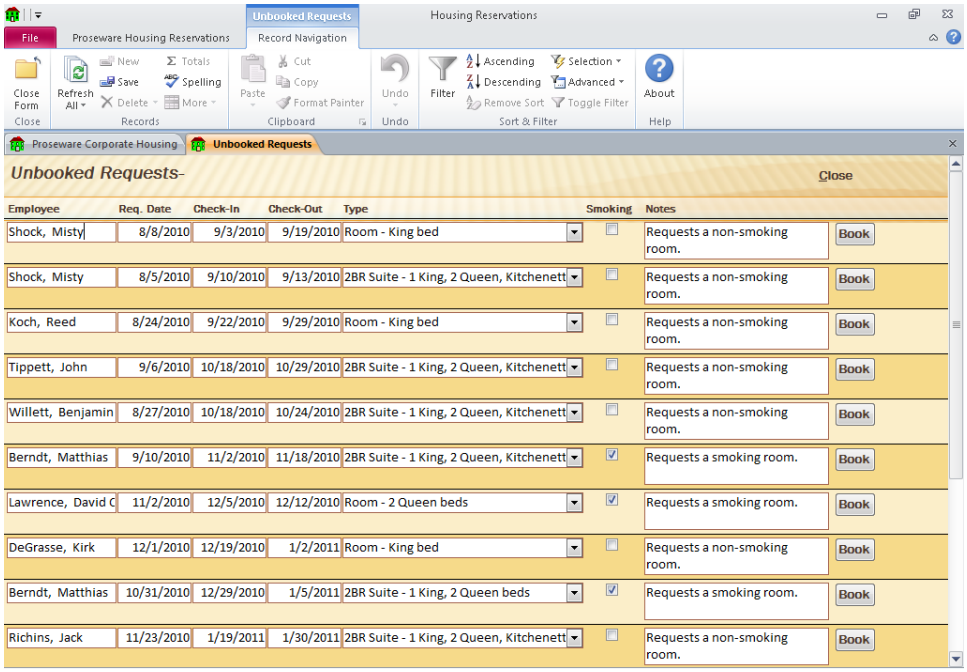

Figure 25-25 The Unbooked Requests form lets administrators view pending requests and start the booking process.

Earlier in this chapter, in "Linking to Related Data in Another Form or Report" on page 1631, you learned one technique for using a command button to link to a related task. The key task in the Housing Reservations application for the housing manager (or any administrator) is to assign a room and book a reservation for pending requests. When you click one of the Book buttons on the Unbooked Requests form, code behind the form opens a form to show the manager the rooms that match the request and aren't booked for the time span requested. If you click the request from Kirk DeGrasse for a room with a king bed from December 1, 2010, to December 19, 2010, you'll see the list of available rooms in the fdlgAvailableRooms form, as shown in Figure 25-26.

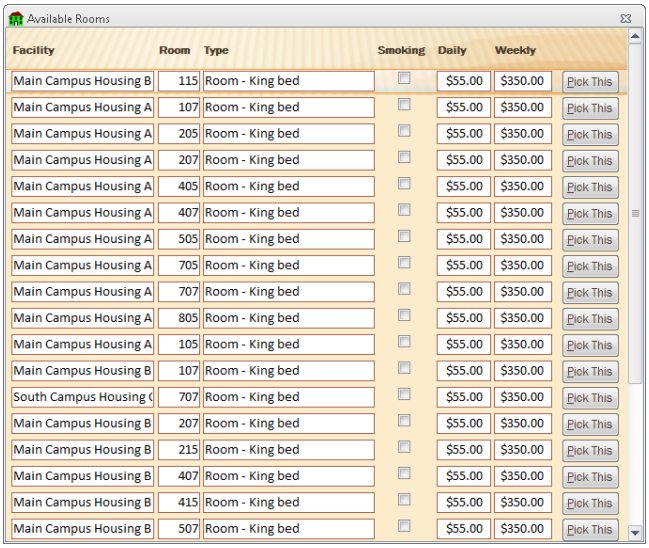

Figure 25-26 The fdlgAvailableRooms form shows a list of available rooms matching the selected reservation request.

The code behind the Book button on the frmUnbookedRequests form is as follows:

```
Private Sub cmdBook_Click()
  ' Make sure no changes are pending
  If Me.Dirty Then Me.Dirty = False
  ' Open the available rooms form - hidden, dialog
  '  and check if any available
  DoCmd.OpenForm "fdlgAvailableRooms", _
    WhereCondition:="Smoking = " & Me.Smoking, _
    WindowMode:=acHidden
  If Forms!fdlgAvailableRooms.RecordsetClone.RecordCount = 0 Then
    MsgBox "There are no available rooms of this " & _
      "type for the dates requested." & _
      vbCrLf & vbCrLf & _
      "You can change the Room Type or dates and try again.", _
      vbInformation, gstrAppTitle
    DoCmd.Close acForm, "fdlgAvailableRooms"
    Exit Sub
  End If
  ' Show the available rooms
  ' - form will call our public sub to create the res.
  Forms!fdlgAvailableRooms.Visible = True
End Sub
```

The record source of the fdlgAvailableRooms form is a parameter query that filters out rooms already booked for the specified dates and includes the remaining rooms that match the requested room type. The code behind the Book button adds a filter for the smoking or nonsmoking request because the room type doesn't include this information, but each specific available room does. Behind the Pick This button on the fdlgAvailableRooms form, you can find the following code:

```
Private Sub cmdPick_Click()
Dim intReturn As Integer
  ' Call the build a reservation proc in the calling form
  intReturn = Form_frmUnbookedRequests.Bookit(Me.FacilityID, Me.RoomNumber, _
    Me.DailyRate, Me.WeeklyRate)
  If (intReturn = True) Then
    MsgBox "Booked!", vbExclamation, gstrAppTitle
  Else
    MsgBox "Room booking failed.  Please try again.", _
      vbCritical, gstrAppTitle
  End If
  DoCmd.Close acForm, Me.Name
End Sub
```

Can you figure out what's happening? Back in frmUnbookedRequests, there's a public function called Bookit that this code calls as a method of that form. It passes the critical FacilityID, RoomNumber, DailyRate, and WeeklyRate fields to complete the booking. Back in frmUnbookedRequests, the code in the public function is as follows:

```
Public Function Bookit(lngFacility As Long, lngRoom As Long, _
  curDaily As Currency, curWeekly As Currency) As Integer
' Sub called as a method by fdlgAvailableRooms to book the selected room
' Caller passes in selected Facility, Room number, and rates
Dim db As DAO.Database, rstRes As DAO.Recordset
Dim varResNum As Variant, strSQL As String, intTrans As Integer
  ' Set error trap
  On Error GoTo BookIt_Err
  ' Get a pointer to this database
  Set db = CurrentDb
  ' Open the reservations table for insert
  Set rstRes = db.OpenRecordset("tblReservations", _
    dbOpenDynaset, dbAppendOnly)
  ' Start a transaction
  BeginTrans
  intTrans = True
  ' Get the next available reservation number
  varResNum = DMax("ReservationID", "tblReservations")
  If IsNull(varResNum) Then varResNum = 0
  varResNum = varResNum + 1
  ' Update the current row
  strSQL = "UPDATE tblReservationRequests SET ReservationID = " & _
    varResNum & " WHERE RequestID = " & Me.RequestID
  db.Execute strSQL, dbFailOnError
```

```
  ' Book it!
  rstRes.AddNew
  ' Copy reservation ID
  rstRes!ReservationID = varResNum
  ' Copy employee number
  rstRes!EmployeeNumber = Me.EmployeeNumber
  ' Copy facility ID from the room we picked
  rstRes!FacilityID = lngFacility
  ' .. and room number
  rstRes!RoomNumber = lngRoom
  ' Set reservation date = today
  rstRes!ReservationDate = Date
  ' Copy check-in, check-out, and notes
  rstRes!CheckInDate = Me.CheckInDate
  rstRes!CheckOutDate = Me.CheckOutDate
  rstRes!Notes = Me.Notes
  ' Copy daily and weekly rates
  rstRes!DailyRate = curDaily
  rstRes!WeeklyRate = curWeekly
  ' Calculate the total charge
  rstRes!TotalCharge = ((Int(Me.CheckOutDate - Me.CheckInDate) \ 7) * _
    curWeekly) + _
    ((Int(Me.CheckOutDate - Me.CheckInDate) Mod 7) * _
    curDaily)
  ' Save the Reservation Row
  rstRes.Update
  ' Commit the transaction
  CommitTrans
  intTrans = False
  ' Clean up
  rstRes.Close
  Set rstRes = Nothing
  Set db = Nothing
  ' Requery this form to remove the booked row
  Me.Requery
  ' Return success
  Bookit = True
BookIt_Exit:
  Exit Function
BookIt_Err:
  MsgBox "Unexpected Error: " & Err & ", " & Error, vbCritical, gstrAppTitle
  ErrorLog Me.Name & "_Bookit", Err, Error
  Bookit = False
  If (intTrans = True) Then Rollback
  Resume BookIt_Exit
End Function
```

It makes sense to have the actual booking code back in the frmUnbookedRequests form because the row the code needs to insert into tblReservations needs several fields from the current request record (EmployeeNumber, CheckInDate, CheckOutDate, and Notes). The code starts a transaction because it must simultaneously enter a ReservationID in both the

tblReservationRequests table and the tblReservations table. If either fails, the error-trapping code rolls back both updates. Notice that the code opens the tblReservations table for append only to make the insert of the new reservation more efficient.

Calculating a Stored Value

If you follow the rules of good table design (see Article 1 on the companion CD), you know that storing a calculated value in a table isn't usually a good idea because you must write code to maintain the value. But sometimes, in a very large database, you need to calculate and save a value to improve performance for searching and reporting. The Housing Reservations application isn't all that large—but it could be in real life. We chose to store the calculated total charge for each reservation to show you some of the steps you must take to maintain a value like this.

Users can create and edit reservation requests, but the creation of the reservation records that contain the calculated value is controlled entirely by code, so maintaining the calculated TotalCharge value in this application is simple. You've already seen the one place where a new reservation record is created—in the public Bookit function in the frmUnbookedRequests form. The little piece of code that calculates the value is as follows:

```
' Calculate the total charge
rstRes!TotalCharge = ((Int(Me.CheckOutDate - Me.CheckInDate) \ 7) * _
  curWeekly) + _
  ((Int(Me.CheckOutDate - Me.CheckInDate) Mod 7) * _
  curDaily)
```

However, in many applications, you may not be able to control the editing of a calculated value this closely. You need to carefully consider the ramifications of saving a calculated value in your table and perhaps write code that an administrator can run to periodically verify that any saved calculated value hasn't become out of sync with the other fields used to perform the calculation.

> **Note**
>
> **Access 2010 includes a new Calculated data type that you can use for storing calculated values, but we wanted to show you in this section how you can achieve this functionality through Visual Basic. You need to realize, however, that if you use Visual Basic to manage storing a calculated value, you cannot enforce the calculation if a user changes the data directly in a table or query datasheet. If you use a Calculated data type, Access controls the calculated value at the engine level.**

Automating Reports

In a typical application, you'll probably spend 80 to 90 percent of your coding effort in event procedures for your *forms*. That doesn't mean that there aren't many tasks that you can automate on *reports*. This next section shows you just a few of the possibilities.

Allowing for Used Mailing Labels

Have you ever wanted to create a mailing label report and come up with a way to use up the remaining labels on a partially used page? You can find the answer in the Conrad Systems Contacts sample application (Contacts.accdb). Let's say you want to send a promotional mailing to all contacts who own the Single User product, offering them an upgrade to Multi-User. Open the main switchboard form (frmMain), click Contacts, and then click Search in the Select Contacts pop-up window. Perform a search for all contacts who own the BO$$ Single User product—you should find eight records in the original sample data. (Click No when the application asks you if you want to see a summary list first.) Click the Print button on the frmContacts form, select Avery 5163 Labels (2" × 4"), ask for the report to include the Displayed Contacts, and specify that your first page of labels is missing three used ones. Your screen should look like Figure 25-27 at this point.

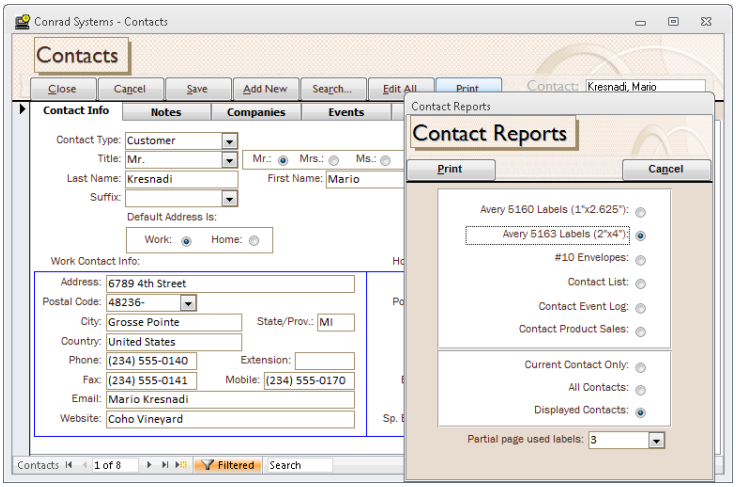

Figure 25-27 You can request mailing labels and specify that some labels have already been used on the first page.

Click the Print button in the dialog box, and you should see the labels print—but with three blank spaces first to avoid the used ones—as shown in Figure 25-28.

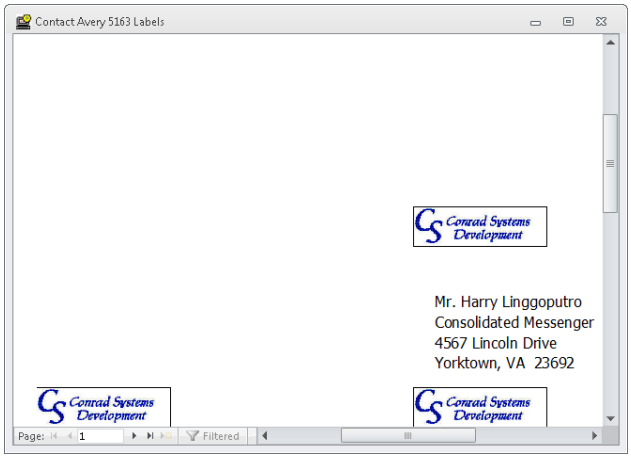

Figure 25-28 The labels print and avoid the used ones.

You can find some interesting code in the AfterUpdate event of the option group to choose the report type in the fdlgContactPrintOptions form. The code is as follows:

```
Private Sub optReportType_AfterUpdate()
  ' Figure out whether to show the "used labels" combo
  Select Case Me.optReportType
    Case 1
      ' Show the used labels combo
      Me.cmbUsedLabels.Visible = True
      ' Hide the number of days option group
      Me.optDisplay.Visible = False
      ' up to 29 used labels on 5160
      Me.cmbUsedLabels.RowSource = "0;1;2;3;4;5;6;7;8;9;10;11;12;13;" & _
        "14;15;16;17;18;19;20;21;22;23;24;25;26;27;28;29"
    Case 2
      ' Show the used labels combo
      Me.cmbUsedLabels.Visible = True
      ' Hide the number of days option group
      Me.optDisplay.Visible = False
      ' up to 9 used labels on 5163
      Me.cmbUsedLabels.RowSource = "0;1;2;3;4;5;6;7;8;9"
    Case 3, 4
      ' Don't need the combo for Envelopes and contact list
      Me.cmbUsedLabels.Visible = False
      ' .. or the number of days filter
      Me.optDisplay.Visible = False
```

```
    Case 5, 6
      ' Don't need the used labels combo for contact events or products
      Me.cmbUsedLabels.Visible = False
      ' Do need the day filter
      Me.optDisplay.Visible = True
  End Select
End Sub
```

You can have up to 29 used labels when printing on Avery 5160 (1" × 2.625") label paper. You can have up to 9 used labels when printing on Avery 5163 (2" × 4") label paper. The combo box that you can use to indicate the number of used labels has a Value List as its row source type, so the code sets up the appropriate list based on the label type you choose.

However, the real trick to leaving blank spaces on the report is in the query that is the record source for the rptContactLabels5163 report—qryRptContactLabels. In the sample database, you can find a table, ztblLabelSpace, that has 30 records, and each record has one field containing the values 1 through 30. The SQL for the qryRptContactLabels query is as follows:

```
PARAMETERS [Forms]![fdlgContactPrintOptions]![cmbUsedLabels] Long;
SELECT "" As Contact, "" As CompanyName, "" As Address, "" As CSZ,
Null As ContactID, "" As Zip, "" As LastName, "" As FirstName,
"" As ContactType, "" As WorkCity, "" As WorkStateOrProvince,
"" As HomeCity, "" As HomeStateOrProvince, 0 As Inactive
FROM ztblLabelSpace
WHERE ID <= [Forms]![fdlgContactPrintOptions]![cmbUsedLabels]
UNION ALL
SELECT ([tblContacts].[Title]+" ") & [tblContacts].[FirstName] & " " &
([tblContacts].[MiddleInit]+". ") & [tblContacts].[LastName] &
(", "+[tblContacts].[Suffix]) AS Contact,
Choose([tblContacts].[DefaultAddress], qryContactDefaultCompany.CompanyName,
Null) As CompanyName,
Choose([tblContacts].[DefaultAddress],[tblContacts].[WorkAddress],
[tblContacts].[HomeAddress]) AS Address,
Choose([tblContacts].[DefaultAddress],[tblContacts].[WorkCity] & ", " &
[tblContacts].[WorkStateOrProvince] & "  " & [tblContacts].[WorkPostalCode],
[tblContacts].[HomeCity] & ", " & [tblContacts].[HomeStateOrProvince]
& "  " & [tblContacts].[HomePostalCode]) AS CSZ,
tblContacts.ContactID,
Choose([tblContacts].[DefaultAddress],[tblContacts].[WorkPostalCode],
[tblContacts].[HomePostalCode]) AS Zip,
tblContacts.LastName, tblContacts.FirstName, tblContacts.ContactType,
tblContacts.WorkCity, tblContacts.WorkStateOrProvince, tblContacts.HomeCity,
tblContacts.HomeStateOrProvince, tblContacts.Inactive
FROM tblContacts
LEFT JOIN qryContactDefaultCompany
ON tblContacts.ContactID = qryContactDefaultCompany.ContactID;
```

The first SELECT statement (up to the UNION ALL) creates dummy blank columns for each field used by the report and uses the ztblLabelSpace table and a filter on the combo box in the fdlgContactPrintOptions form (see Figure 25-27) to return the correct number of blank rows. The query uses a UNION with the actual query that returns contact data to display information on the report.

Because this label report prints a logo, there's one final bit of code that keeps this from appearing on the blank labels in the rptContactLabels5163 report. The code is as follows:

```
Private Sub Detail_Format(Cancel As Integer, FormatCount As Integer)
  ' Don't print the return logo if this is a "spacer" record
  If IsNull(Me.ContactID) Then
    Me.imgCSD.Visible = False
  Else
    Me.imgCSD.Visible = True
  End If
End Sub
```

The Format event of the Detail section depends on the fact that the ContactID in the "spacer" rows is Null. When printing a blank row for spacing, the code hides the logo.

Drawing on a Report

When you want to draw a border around a report print area, sometimes you'll need to write some code to ask Access to draw lines or a border after placing the data on the page. This is especially true if one or more controls on the report can grow to accommodate a large amount of data.

We used the Report Wizard to create the basic rptContacts report using the Justified format. (We customized the report after the wizard finished.) The wizard created a fairly decent layout with a border around all the fields, but it didn't make the text box to display notes large enough to display the text for all contacts. Figure 25-29 shows you the report displaying John's contact record from the database. You can see that the notes about John are cut off at the bottom.

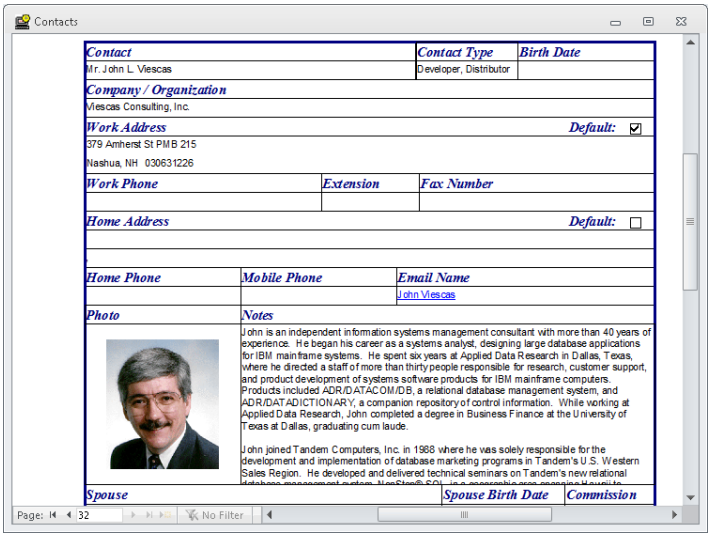

Figure 25-29 This report uses a border around the data, but one of the text boxes isn't large enough to display all the text.

It's simple enough to change the Can Grow property of the text box to Yes to allow it to expand, but the rectangle control used to draw the border around all the text doesn't also have a Can Grow property. The solution is to remove the rectangle and use the Line method of the Report object in the report's Format event of the Detail section to get the job done. Here is the code that you can find in this event procedure in the rptContactsExpandNotes report:

```
Private Sub Detail_Print(Cancel As Integer, PrintCount As Integer)
Dim sngX1 As Single, sngY1 As Single
Dim sngX2 As Single, sngY2 As Single, lngColor As Long
  ' Set coordinates
  sngX1 = 120
  sngY1 = 120
  sngX2 = 9120
  ' Adjust the height if Notes has expanded
  sngY2 = 7680 + (Me.Notes.Height - 2565)
  ' Draw the big box around the data
  ' Set width of the line to 8 pixels
  Me.DrawWidth = 8
  ' Draw the rectangle around the expanded fields
  Me.Line Step(sngX1, sngY1)-Step(sngX2, sngY2), RGB(0, 0, 197), B
End Sub
```

The Line method accepts three parameters:

1. The upper-left and lower-right corners of the line or box you want to draw, expressed in *twips*. (There are 1440 twips per inch.) Include the Step keyword to indicate that the coordinates are relative to the current graphics position, which always starts at 0, 0. When you use Step for the second coordinate, you provide values relative to those you specified in the first set of coordinates.
2. The color you want to draw the line or box, expressed as a red-green-blue (RGB) value. (The RGB function is handy for this.)
3. An indicator to ask for a line, a rectangle, or a filled rectangle. No indicator draws a line. Include the letter B to ask for a rectangle. Add the letter F to ask for a filled rectangle.

Before you call the Line method, you can set the DrawWidth property of the report to set the width of the line. (The default width is in pixels.)

The only tricky part is figuring out the coordinates. On the original report, the rectangle starts at 0.0833 inches in from the left and down from the top, so multiplying that value by 1440 twips per inch gave us the starting values of 120 down from the top and 120 in from the left edge. The width of the rectangle needs to be about 6.3333 inches, so the relative coordinate for the upper-right corner is 6.3333 × 1440, or about 9,120 twips. The height of the rectangle needs to be at least 5.3333 inches, or about 7,680 twips, and the height needs to be adjusted for the amount that the Notes text box expands. The Notes text box is designed to be a minimum of 1.7813 inches high, or 2,565 twips, so subtracting 2,565 from the actual height of the Notes text box when it's formatted (the Height property is also in twips) gives you the amount you need to add to the original height of the rectangle.

If you open the rptContactsExpandNotes report and move to John's record on page 32, you'll see that the rectangle now expands nicely to fit around the Notes text box that grew to display all the text in John's record. Figure 25-30 shows you the report with the rectangle drawn by the code behind the report.

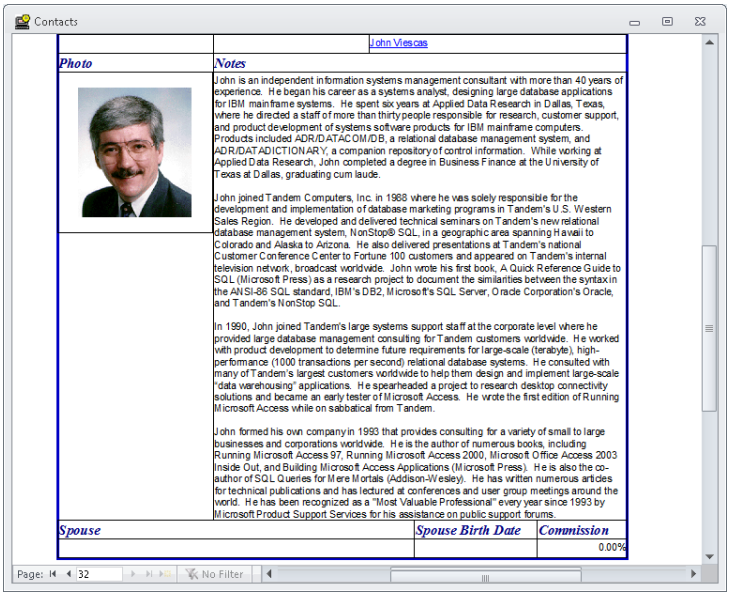

Figure 25-30 Code in the rptContactsExpandNotes report draws a custom rectangle around expanded text.

INSIDE OUT

Exploring Other Reports That Use the Line Method

To see more interesting examples of using the Line method to dynamically draw lines on a report, you can also explore the rptLaborPlan, rptLaborPlanIndividual, and rptWeeklyPostedSchedule client reports in the Back Office Software System web database. You might find it easier to explore these reports if you use the BOSSDataCopy.accdb database because the main BOSS.accdb database uses custom ribbons.

Dynamically Filtering a Report When It Opens

The two most common ways to open a report filtered to print specific records are:

- Use the WhereCondition parameter with the DoCmd.OpenReport method (usually in code in an event procedure behind a form) to specify a filter.
- Base the report on a parameter query that prompts the user for the filter values or reference control values on an open form.

In some cases, you might design a report that you intend to open from several locations in your application, and you can't guarantee that the form to provide filter values will always be open. Alternatively, you might have multiple reports that need the same filter criteria, and you don't want to have to design a separate filter form for each report. To solve these problems, you can add code to the report to have it open its own filter dialog box from the report's Open event procedure. Let's go back to the Housing Reservations application (Housing.accdb) to take a look at a report that uses this technique.

In the Housing Reservations application, both the rptFacilityOccupancy report and the rptFacilityRevenueChart report depend on a single form, fdlgReportDateRange, to provide a starting and ending date for the report. To see the rptFacilityOccupancy report, you can start the application by opening the frmSplash form, sign in as an administrator (Conrad, Jeff; Richins, Jack S.; Schare, Gary; or Viescas, John L.), click the Reports button on the main switchboard, and then click the Reservations button in the Facilities category on the Reports switchboard. (You can also simply open the report directly from the Navigation pane.) When you open the report, you'll see a dialog box prompting you for the dates you want, as shown in Figure 25-31.

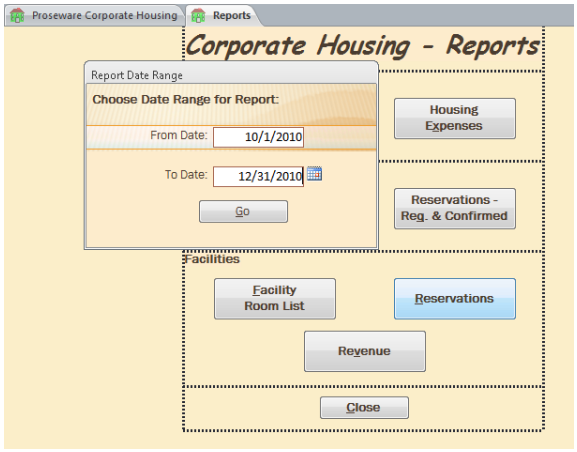

Figure 25-31 A parameter dialog box opens from the report that you asked to view.

Unless you've reloaded the sample data, the database contains reservations from August 24, 2010 through March 22, 2011, so asking for a report for October, November, and December should work nicely. Enter the dates you want and click Go to see the report, as shown in Figure 25-32.

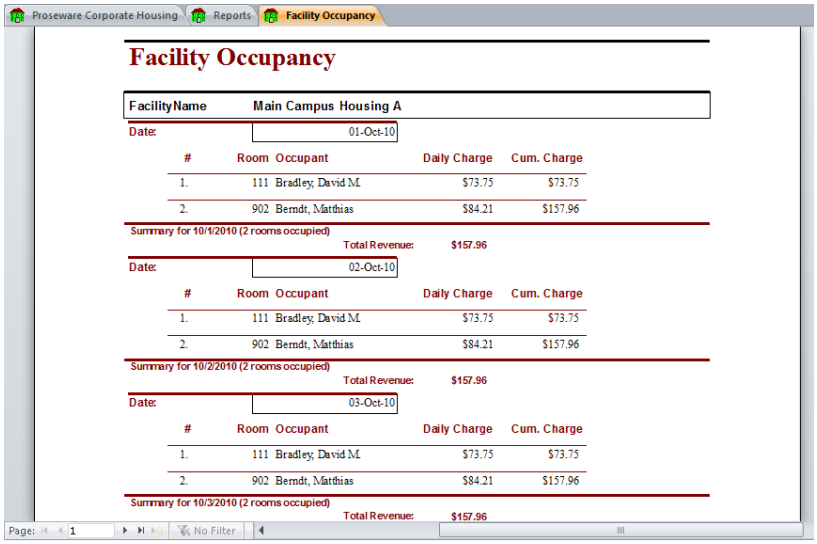

Figure 25-32 The Facility Occupancy report uses a shared filter dialog box to let you specify a date range.

The report has a parameter query in its record source, and the parameters point to the *from* and *to* dates on the fdlgReportDateRange form shown in Figure 25-31. However, the code behind the Reservations button on the Reports switchboard opens the report unfiltered. It's the code in the report's Open event procedure that opens the dialog box so that the query in the record source can find the parameters it needs. The code is as follows:

```
Private Sub Report_Open(Cancel As Integer)
  ' Open the date range dialog
  ' .. report record source is filtered on this!
  DoCmd.OpenForm "fdlgReportDateRange", WindowMode:=acDialog
End Sub
```

This works because a Report object doesn't attempt to open the record source for the report until after the code in the Open event completes. Therefore, you can place code in the Open event to dynamically change the record source of the report or, as in this example, open a form in Dialog mode to wait until that form closes or hides itself. The code behind the dialog form, fdlgReportDateRange, is as follows:

```
Private Sub Form_Load()
  ' If passed a parameter, reset defaults to last quarter
  If Not IsNothing(Me.OpenArgs) Then
    ' Set the start default to first day of previous quarter
    Me.txtFromDate.DefaultValue = "#" & _
      DateSerial(Year(Date), ((Month(Date) - 1) \ 3) * 3 - 2, 1) & "#"
    ' Set the end default to last day of previous quarter
```

```
    Me.txtToDate.DefaultValue = "#" & _
      DateSerial(Year(Date), ((Month(Date) - 1) \ 3) * 3 + 1, 1) & "#"
  End If
End Sub

Private Sub cmdGo_Click()
  ' Hide me so report can continue
  Me.Visible = False
End Sub
```

The code in the form's Load event checks to see if the report that is opening the form has passed a parameter in the OpenArgs property. If so, the code resets the default values for the two date text boxes to the start and end dates of the previous quarter. If you look at the code behind the rptFacilityRevenueChart report, you'll find that this report asks for the different default values, but it's the code in the Click event of the Go command button that gets things rolling. The code behind the form responds to your clicking the Go button to hide itself so that the report can continue. It can't close because the record source of the report depends on the two date parameters. As noted earlier, hiding this form opened in Dialog mode allows the code in the Open event of the report to finish, which lets the report finally load its record source. As you might suspect, there's code in the Close event of the report to close the parameter form when you close the report or the report finishes printing.

Calling Named Data Macros

As you learned in Chapter 7, "Creating Table Data Macros," Access 2010 now includes data macros that you can use to attach logic at the data layer. When you attach data macros to table events, Access only fires those data macros in response to data events, such as inserting a new record, updating values in an existing record, or deleting records. You can also create a named data macro attached to the table itself. A named data macro, in many ways, is similar to a public procedure or function in a standard code module—you can call a named data macro from macros and Visual Basic to execute the actions defined in the data macro and return results to the caller if you want.

In the Back Office Software System web database, we defined many named data macros attached to the web tables to help automate the application. In the tblErrorLog table included in this web database, we log application errors that occur when users are working with the database in Access client. Attached to this table is a named data macro, called LogError, that you studied in Chapter 7. As you might recall, this named data macro includes six parameters that we use to fill in the data we want to record about the error in the table. We use the CreateRecord data block to create a new record in the tblErrorLog table and then use the SetField action and pass in the data from the parameters into the appropriate fields.

Open the BOSSDataCopy.accdb web database and we'll show you how we call this named data macro from Visual Basic. If you open any client form or client report in this web database, you'll notice that each Visual Basic procedure or function includes an error handler label. If the procedure or function encounters an error, we call a public procedure in a standard module to log the error details. A code example of this call at the form and report level is as follows:

```
ErrorHandler "frmDailyLaborPlans", "cmdClearList_Click", Err.Number, Err.Description
```

As you can see, we call a public procedure called ErrorHandler that includes four parameters—the name of the object where the error occurred, the specific procedure that generated the error, the error number that Access returned, and the description of the application error. Open the modErrorTrap module in Design view from the Navigation pane and let's take a look at this procedure. The code in the public ErrorHandler procedure in modErrorTrap is as follows:

```
Public Sub ErrorHandler(strModule As String, _
    strProcedure As String, _
    lngErrorNumber As Long, _
    strErrorString As String)
' For this procedure, continue on if any errors occur
On Error Resume Next
    Dim strMessage As String
    Dim datCurrentTime As Date
    Dim strUser As String
    ' Current date and time
    datCurrentTime = Now()
    ' If we can't determine the current web user who
    ' triggered the error, then use the legacy CurrentUser
    ' function to log user name
    If IsNull(CurrentWebUser(acWebUserName)) Then
        strUser = CurrentUser()
    Else
        strUser = CurrentWebUser(acWebUserName)
    End If
    ' The second action argument of DoCmd.SetParameter is an expression.
    ' If the second argument is a string,
    ' you need to escape the string in double quotes.
    DoCmd.SetParameter "ParamModule", "'" & strModule & "'"
    DoCmd.SetParameter "ParamProcedure", "'" & strProcedure & "'"
    DoCmd.SetParameter "ParamErrorNumber", lngErrorNumber
    DoCmd.SetParameter "ParamErrorString", "'" & strErrorString & "'"
    DoCmd.SetParameter "ParamUser", "'" & strUser & "'"
    DoCmd.SetParameter "ParamTime", "#" & datCurrentTime & "#"
    ' Run the named data macro to log this error event to the error table
    DoCmd.RunDataMacro "tblErrorLog.LogError"
    ' Build up a custom message to display to the user
    strMessage = "There was an unexpected error in your application." & vbCr & vbCr
    strMessage = strMessage & "Error Details:" & vbCr
    strMessage = strMessage & "Module : " & strModule & vbCr
```

```
    strMessage = strMessage & "Procedure : " & strProcedure & vbCr
    strMessage = strMessage & "Error : " & lngErrorNumber & vbCr & vbCr
    strMessage = strMessage & "Please notify Technical Support of this error." _
        & vbCr & vbCr
    strMessage = strMessage & "Would you like to view this report in order to print?"
    ' View report if requested by user
    If MsgBox(strMessage, vbCritical + vbYesNo + vbDefaultButton1, _
        "BOSS - Unexpected Error") = vbYes Then
        DoCmd.OpenReport "rptErrorLog", acViewPreview, , _
        "ErrorTime=#" & datCurrentTime & "#"
    End If
End Sub
```

In the first part of this procedure, the code fetches the current system date and time and currently logged in user name to record into the tblErrorLog table. Next, the code needs to run a DoCmd.SetParameter method for each parameter we defined in the data macro. The first action argument of the SetParameter action is the name of the parameter defined in the named data macro and the second action argument is the data to pass into the parameter. As you can see in the code, we list each parameter name and the variable that holds the data we want to record. Note that if the data you are passing in is a string, you'll need to escape the string in quote marks. After the code sets all the parameters, we're ready to call the named data macro by using the DoCmd.RunDataMacro method. In the argument for the RunDataMacro method, you need to supply the name of the table, a period, and then the name of the named data macro attached to the table. You also need to enclose the argument in double quotation marks even if your table and named data macro does not have any spaces. The last part of this code creates a custom message to display to the user and asks if the user wants to print out the error report. The completed code procedure logs any application errors encountered in Access client by calling our named data macro.

If you call a named data macro that returns data to the caller using a ReturnVar, you can use the value in the ReturnVar variable in your Visual Basic code. For example, if you wanted to use Visual Basic to get the current version number of the BOSS application and display the value in a message box, you could use the sample TestReturnVar procedure in the modErrorTrap module:

```
Public Sub TestReturnVar()
  ' Set the parameter of ParamValue to Version
  DoCmd.SetParameter "ParamValue", """Version"""
  ' Run the named data macro to get the current version of BOSS
  DoCmd.RunDataMacro "tblSettings.GetCurrentValue"
  ' Display the current version of BOSS in a message box
  ' Use the ReturnVar called RVVersion in the message box
  MsgBox "The current version number of BOSS is: " & _
    ReturnVars!RVVersion, vbInformation, "Version Number"
End Sub
```

If you run this sample procedure, the code executes the named data macro GetCurrentValue attached to the tblSettings web table, passes in the appropriate parameter value we want to use, and then displays the value received from the data macro in the message box using the ReturnVars collection. Note that unlike the TempVars collection, the ReturnVars collection does not support a Remove or RemoveAll method to delete or clear the values stored in the ReturnVars collection. You also cannot set a return variable using Visual Basic; you must always use the SetReturnVar data action in a named data macro attached to a table to set return variables. If you set a ReturnVar in a named data macro, Access keeps that value in the ReturnVars collection until you run another named data macro that sets another ReturnVar or you close the database. You can set up to 255 return variables in the ReturnVars collection, but you would have to set all those return variables in one named data macro call.

As you've seen in this chapter, Visual Basic is an incredibly powerful language, and the tasks you can accomplish with it are limited only by your imagination. In Chapter 26, "The Finishing Touches," the next chapter of this book on the companion CD, you'll learn how to set startup properties and create custom ribbons.