

## CHAPTER 4

# A Visual Basic .NET Framework

In this chapter, we'll build a simple application framework for Visual Basic .NET forms applications that use SQL Server. You'll see how object-oriented techniques can greatly reduce development time and cost, and simplify your job. If you thought Visual Basic .NET was a lot harder than Visual FoxPro, I think you'll be pleasantly surprised.

A small Visual Basic .NET Windows Application project might consist of

- A Main form to contain the application and the menu
- The Menu control in the Main form
- A few forms to add, edit, and delete records from individual tables

We'll base the Add/Edit/Delete forms on a single inheritable form class, so that we only write the common code once and reuse it as needed.

### Starting the New Windows Application Project

To start a Visual Basic .NET project, open up the IDE and select File, New from the menu. The resulting dialog lets you pick from a wide range of project types. Unlike FoxPro, where there is only one language and one project type, in Visual Studio .NET you pick both the project type and the development language. This selection determines which namespaces are included in the project references. For example, if we pick Visual Basic Projects and click on the Windows Application project type, the `Windows.Forms` namespace (among others)

### IN THIS CHAPTER

- Starting the New Windows Application Project
- Adding a Windows Controls Library
- Building Your First Inheritable Form
- Programming with Class
- Click Event Code for the Form's Buttons

will appear in the list of references. If we instead pick a Smart Devices project, the references for Pocket PC and Windows CE are included.

You first have to pick a name for your new project. When you create this project, Visual Studio adds a new directory under your default projects directory, which is initially `Documents and Settings\MyUserID\Visual Studio Projects\YadaYada`. I changed mine to `C:\VBProjects` and recommend that you do likewise. It creates both a solution (a container for several projects) and a project. As you'll see repeatedly in our examples, Visual Basic .NET assumes a different arrangement for projects than does FoxPro. In FoxPro, we build one project, and may include several class libraries. In Visual Basic .NET, each class library is usually built as its own project, and compiled as a DLL, then included as a reference in other projects that use the classes. It doesn't take long to get used to.

The newly created Visual Basic .NET Windows Application project also includes one form, named `Form1.vb` by default. `Form1` is both the filename and an internal class name. You should change both. In this case, because it's the first form that was created, we'll use it as we used `MAIN.PRG` in our FoxPro project. There's no `_Screen` object in Visual Basic .NET, so this first form will become our background screen. Using F4, open the Properties window, and change the Name property to `AppScreen`. (If you open the code window for the form, you'll see that the class name in the first line has been changed to `AppScreen`.) Open the Solution Explorer with `Ctrl+Alt+L` and select Rename, and change `Form1.vb` to `AppScreen.vb`. Right-click again on the project in the Solution Explorer and select Rebuild.

**NOTE**

A FoxPro form would consist of two files, an SCX and an SCT.

Next, we'll need a menu. Use `Ctrl+Alt+X` to open the Toolbox, select Windows Forms, and drag a `MainMenu` control to anywhere on the `AppScreen` form. When selected, the `MainMenu` control appears in the upper-left corner of the screen. For now, it's the only control, so it will be selected automatically. Later, if you add other controls (for example, a label) to the form, the menu control will disappear when the label is selected. Click on the `MainMenu` control to begin building your menu.

The `MainMenu` control is simple and intuitive. (I believe it was ported directly from Delphi when the lead architect of Delphi was ported over to Microsoft.) As you move down and right, new text boxes appear to let you add menu selections. You should right-click on each of your menu pads and change the name to something meaningful (for example, `mnuExit` for the File, Exit menu pad), so that the `Click` code for the menu option makes sense when you read it. Add a File pad first, and below it add an Exit bar. Right-click on the File pad, select Properties, change the name to `mnuExit`, and then double-click on it and type in the single command **End**. Press F5 to compile and run the application, and you'll see that your form closes when you click on Exit.

## The AppScreen Form Properties

The AppScreen form is the container for the rest of the forms in your application, so let's configure it more to our liking. Use F4 to open the Properties window and make the following property settings:

```
StartPosition - CenterScreen
FormBorderStyle - Fixed3D
Text - My First VB.NET Application
```

Remember that I said that some things are harder in .NET? This is one of them: The drop-shadow trick that we used in Chapter 2, “Building Simple Applications in Visual FoxPro and Visual Basic .NET,” to put a title on the screen with a “drop shadow” turns out to be unusually difficult in Visual Basic .NET because the Label control can't be transparent on a Windows form.

### RANT

<rant>I'm sure it will be changed by the time this book hits the shelves, but at this moment, you can't get there from here. There is a different kind of control that allows drawing text with a shadow, but it's 15 lines of code, doesn't demonstrate inheritance, and is so complicated that it irritates me.</rant>

4

## Adding a Windows Controls Library

The whole idea of object-oriented programming is that you code things once, and then reuse them throughout your application. That applies to everything—even the controls on each of your forms.

For example, I like to make it easy for users to see which control has the focus. So I either change the background color of the active control or enhance its border. It's a small thing, but it wouldn't be a small thing if I had to code each individual control. Believe it or not, kids, back when I was walking six miles to school in a foot of snow, that's what we had to do. We wrote macros to speed up the process, but what a pain!

Now it's much, much easier. Simply right-click on your solution and select Add Class Library, giving it the name MyControls. A new project will be added to your solution, with an empty code window. Name it MyControls.vb. Add the code shown in Listing 4.1. Recompile the project.

### LISTING 4.1 The MyControls Class

```
Imports System.Windows.Forms
Imports System.Drawing

Public Class MyControls
```

**LISTING 4.1** Continued

```
Public Class MyText
    Inherits TextBox
    Public Sub New()
        MyBase.new()
        Text = ""
        Width = 200
        Enabled = False
        BackColor = System.Drawing.SystemColors.ControlLight
    End Sub
    Public Sub EnterHandler( _
        ByVal Sender As Object, ByVal e As EventArgs) _
        Handles MyBase.Enter
        ForeColor = ForeColor.White
        BackColor = BackColor.Blue
    End Sub
    Public Sub LeaveHandler( _
        ByVal Sender As Object, ByVal e As EventArgs) _
        Handles MyBase.Leave
        ForeColor = ForeColor.Black
        BackColor = BackColor.White
    End Sub
    Public Sub DisableHandler( _
        ByVal Sender As Object, ByVal e As EventArgs) _
        Handles MyBase.EnabledChanged
        Sender.BackColor = _
            IIf(Sender.enabled, BackColor.White, _
                System.Drawing.SystemColors.ControlLight)
    End Sub
End Class

Public Class MyEdit
    Inherits TextBox
    Public Sub New()
        MyBase.new()
        Text = ""
        Width = 200
        Multiline = True
        Enabled = False
        BackColor = System.Drawing.SystemColors.ControlLight
    End Sub
    Public Sub EnterHandler( _
        ByVal Sender As Object, ByVal e As EventArgs) _
```

**LISTING 4.1** Continued

```
Handles MyBase.Enter
    ForeColor = ForeColor.White
    BackColor = BackColor.Blue
End Sub
Public Sub LeaveHandler( _
    ByVal Sender As Object, ByVal e As EventArgs) _
    Handles MyBase.Leave
    ForeColor = ForeColor.Black
    BackColor = BackColor.White
End Sub
Public Sub DisableHandler( _
    ByVal Sender As Object, ByVal e As EventArgs) _
    Handles MyBase.EnabledChanged
    Sender.BackColor = _
    IIf(Sender.enabled, BackColor.White, _
        System.Drawing.SystemColors.ControlLight)
End Sub
End Class

Public Class MyCombo
    Inherits ComboBox
    Public Sub New()
        MyBase.new()
        Text = ""
        Width = 200
        Enabled = False
        BackColor = System.Drawing.SystemColors.ControlLight
    End Sub

    Public Sub DisableHandler( _
        ByVal Sender As Object, ByVal e As EventArgs) _
        Handles MyBase.EnabledChanged
        Sender.BackColor = _
        IIf(Sender.enabled, BackColor.White, _
            System.Drawing.SystemColors.ControlLight)
    End Sub
End Class

Public Class MyCheck
    Inherits CheckBox
    Public Sub New()
        MyBase.new()
```

**LISTING 4.1** Continued

---

```
        Text = ""
        Width = 200
        Enabled = False
    End Sub
End Class

Public Class MyRadio
    Inherits RadioButton
    Public Sub New()
        MyBase.new()
        Text = ""
        Width = 200
        Enabled = False
    End Sub
End Class

Public Class MyLabel
    Inherits Label
    Public Sub New()
        MyBase.new()
        Text = ""
        TextAlign = ContentAlignment.MiddleRight
        Height = 12
    End Sub
End Class

End Class
```

---

Next, open the Toolbox using Ctrl+Alt+X, right-click, and add a new tab called My User Controls. Open the tab, right-click anywhere under it, and select Add/Remove Items. When the Customize Toolbox dialog appears, click on the Browse button, and add the new MyControls.dll component from the MyControls\bin directory.

Now when you open the Toolbox and select the MyControls tab, you'll see the MyText, MyCombo, MyCheck, MyRadio, MyLabel, and MyEdit components.

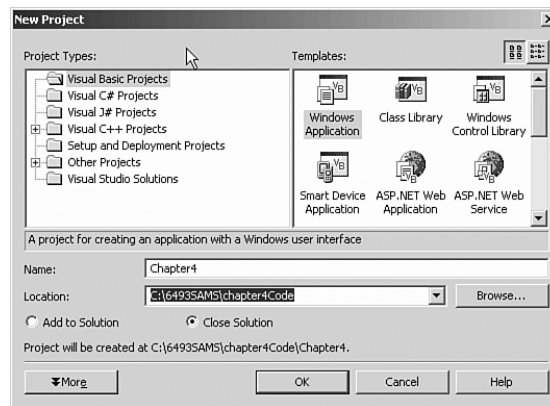
From now on you'll use these controls on your forms, rather than the VB standard controls. And if you change MyLabel's font to Tahoma Bold and its color to purple, it will change on every single one of your forms. That's a lot of benefit for 15 seconds of work.

The same holds true for forms. In FoxPro, it's common to build a form to add, edit, and delete records in a single table. Most applications have several such tables, and the logic is the same for all of them. In FoxPro, they're called form templates, but they're just form

classes. In Visual Basic .NET, they're called inheritable forms. So let's build one and see if the benefits that accrue are as considerable as they are in FoxPro.

## Building Your First Inheritable Form

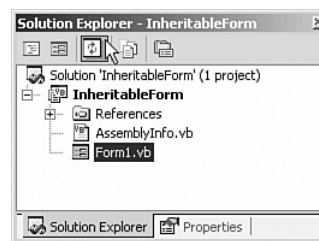
To create an inheritable form class, open the VS IDE and select New Visual Basic .NET Windows Application project. Enter **InheritedForm** as the project name. (I've changed the location where the solution and project will be created to C:\VBProjects so as to make my Visual Basic .NET code easy to find.) Fill in the screen as shown in Figure 4.1.



4

**FIGURE 4.1** Starting a new solution and Windows Application project.

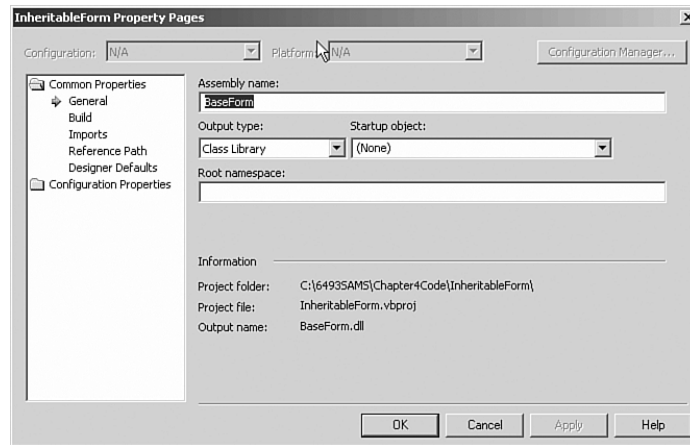
Visual Studio will create a solution file named `InheritableForm`, as well as a project file named `InheritableForm` containing a form named `Form1.vb`, as shown in Figure 4.2.



**FIGURE 4.2** The solution and project in the Solution Explorer window.

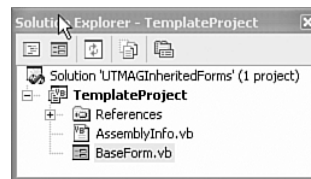
Because we want to create an inheritable form, we need to change the Output type from Windows Application to Class Library. First close the Form Designer because you can't change project properties while the Form Designer has one of its forms open. Make sure the Solution Explorer window (Ctrl+Alt+L) is visible and selected, then right-click on the

project name and select Properties. Select Class Library from the Output Type, combo box, change the name to `BaseForm`, and blank out the Root Namespace text box as shown in Figure 4.3.



**FIGURE 4.3** Changing the inheritable form name in the Inheritable Form Property Page.

Close the Properties page. Finally, right-click on `Form1.vb` in the Solution Explorer and change the name to `BaseForm.vb`. The result should look like what's shown in Figure 4.4.



**FIGURE 4.4** The renamed form class in the Solution Explorer.

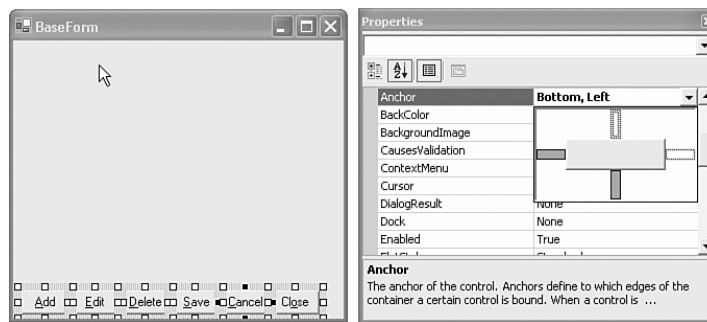
Double-click on `baseform.vb` to bring up the designer, and then use `Ctrl+Alt+X` (or select View, Toolbox from the IDE menu) to display the Toolbox. Select Windows Forms. (You can only select it if the Designer is open, so if you don't see the Toolbox tab, that's probably the problem.) I want to have buttons to add, edit, delete, save, cancel, and close the form, so double-click six times on the button control. You can then use the Layout tools to line them up. Select View, Toolbars from the IDE menu and make sure Layout is checked.

Next, change the buttons' Name properties to `btnAdd`, `btnEdit`, `btnDelete`, `btnSave`, `btnCancel`, and `btnClose`, and then change their respective Text values to Add, Edit, Delete, Save, Cancel, and Close. I like to put an ampersand (&) in front of the appropriate capitalized letter of each text caption to enable a hotkey. I use `C1&ose` because `&Cancel` is



taken, even though it's not really a conflict because both are never enabled at the same time. Speaking of enabling, I initially disable all buttons except Close, for reasons that will become clear presently. I also changed the form's Text property to Change me! as a reminder to the programmer.

Finally, position all six of the buttons near the bottom left of the screen, and then select all of them and set the Anchor property to Bottom, Left, as shown in Figure 4.5.



4

**FIGURE 4.5** Anchoring the buttons to the bottom left of the form.

Although some “flat files” are very large, most applications use lots of little tables. So I’m going to make a simplifying assumption that a single level of filtering is sufficient to show a subset of matching records from which a single record can be chosen for viewing or editing. For this purpose, I’ll put a text box at the top of the screen so that the user can enter a filtering value for a designated search field, and a button and a list box to show the matching values of said search field. It’s not going to work in every case, but if it meets 90% of your needs, it’s a good start. Add two labels, a text box, a button, and a list box to make the form look like Figure 4.6.



**FIGURE 4.6** Adding a search capability to the form.

I anchored the Show Matching Records button at the top and right, and the list box is anchored at the top, right, and bottom. As a result, automatic resizing performs exactly as you would expect. That's better than writing all of that resizing code that used to be required, and it works the same way in the IDE as it does in the final program.

### Coding the Form Class

The first thing I code is always the Close button because I'm anxious to see it work. Double-click on the Close button and you'll see that the IDE generates two lines of code. Listing 4.2 shows the code for the Click event of the Close button. Note that the Handles clause determines which event the routine responds to, not the routine name as is the case in Visual FoxPro.

#### LISTING 4.2 The Close Button Click Event Code

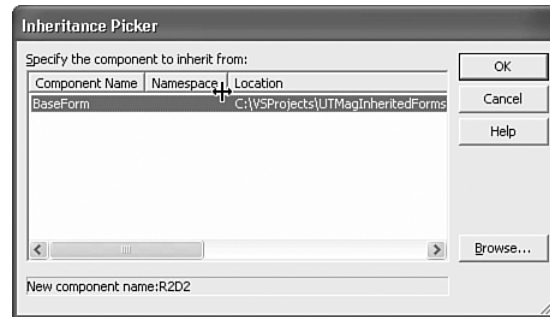
```
Private Sub btnClose_Click( _  
    ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) _  
    Handles btnClose.Click  
  
    Close()  
End Sub
```

The Private Sub and End Sub lines were written by the IDE's code generator. I added the Close command. (The ending parentheses were added by the IDE.)

The form displays one record at a time, which makes saving changes simple. That's the reason for this design. But that means we have to have a mechanism for selecting a record for viewing or editing. That's what the text box, the list box, and the Show Matching Record button are for. The user enters a string—one or more letters—into the text box and clicks on the button, and the list is populated with all of the records that match. But match on what? I've decided to specify a single search field, presumably the most important field in the table, as the target for the search. I've also decided to show all records that start with the string entered by the user. As you'll see shortly, matching any part of the expression is equally easy. But it's my design, so I'll do it my way.

To see how this works, compile the project. This creates a DLL named BookInheritedForm.dll, which can contain several inheritable items. Next, add a project called UseTheFormLuke to your current solution. (I did my first FoxBASE project for George Lucas at the Skywalker Ranch.) It will add a form called Form1, which we'll ignore for now.

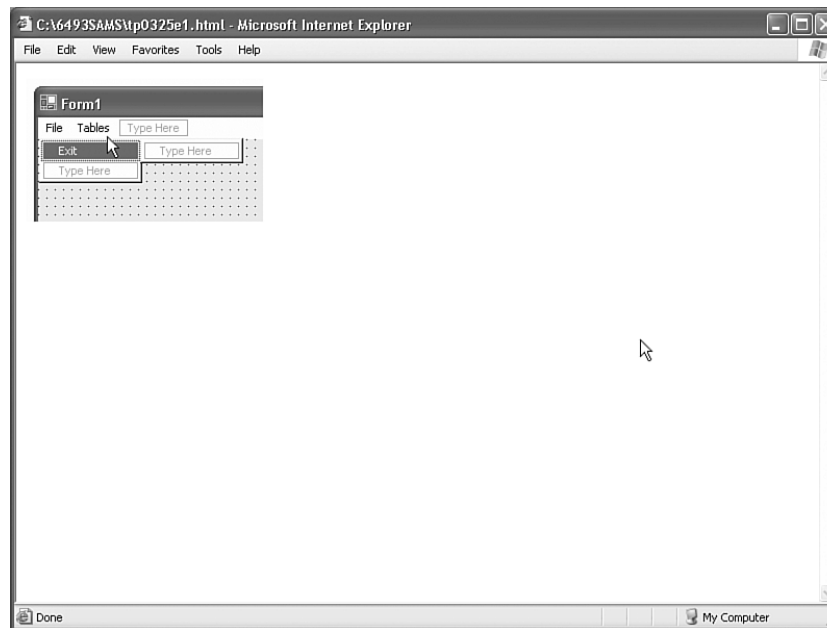
Next, right-click on the new project and select Add, Add Inherited Form from the context menu. The resulting dialog will first ask for a name for the new form (call it Test), and then it will ask you to select from the available inheritable classes, as shown in Figure 4.7. Select BaseForm, the only one on the list. The resulting form will look just like BaseForm because it inherits from BaseForm.



**FIGURE 4.7** Selecting an inheritable class for an inherited form.

We'll need a Main Form in order to test our inherited form. You can use Form1, which was created automatically when you added the Windows Application project. Change the filename to MainForm.vb by right-clicking on the filename and selecting Rename, and then open the form's code and change `Class Form1`, the first line in the file, to `Class MainForm`. (You can also open the form in the Form Designer and change the Name property.)

Next, drag a MainMenu control from the Windows Form toolbox to the form's design surface. Type **File** in the top left cell, and **Exit** just below it, as shown in Figure 4.8.



**FIGURE 4.8** Adding menu items to the form.

Double-click on **Exit** and enter the single command **End**. Go up and to the right of **File** and type **Tables**, and then go down and type **Test Form**. Double-click on **Test Form** and enter the following three lines of code:

```
Dim frm as Test
frm = New Test
frm.Show()
```

Press F5 to compile and run your application. Then select **Tables, Test** from the menu, and you'll see your first inherited form. It doesn't do much yet, but it will.

## Programming with Class

We want to allow programmers to use this inheritable form simply by filling in some properties. What are properties? In Visual FoxPro, they're something like public variables at the class level. In Visual Basic .NET you can enter a `Public As String` statement in the declarations at the top of a class, and the resulting element (called a **field**) is accessible to classes subclassed from the class. For example, if a class contains `Public MyField as String` in its declarations, then in a subclass of the class, IntelliSense will expose `Me.MyField` (Me is like `THISFORM` in FoxPro). But it's not visible in the class's property sheet, nor is it visible in the property sheet of a subclass of the class.

In order to view and set the property in a subclass of the class, you have to create a private variable and a property procedure to save and retrieve it. And what's exposed is not the private variable, but rather the property procedure containing the Getter and Setter routines. For example, to provide a settable `MainTable` property, you add the code shown in Listing 4.3 to the top of your form class's code, just below the declarations.

### LISTING 4.3 Declaring A Property Procedure

```
Private _MainTable As String
Public Property MainTable() As String
    Get
        Return _MainTable
    End Get
    Set(ByVal Value As String)
        _MainTable = Value
    End Set
End Property
```

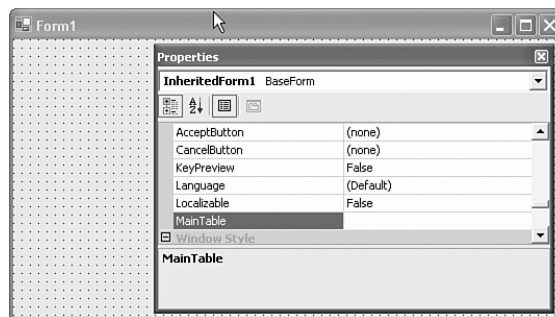
You really only have to enter four of these lines of code: When you type `Public Property As` and press Enter, the IDE writes all of the code except `Return _MainTable` and `_MainTable = Value`. So you have to create the private variable (by convention the name of the property procedure preceded by an underscore), the procedure name, and the

Return and Value assignment statements. Then, when you subclass the class, `MainTable` (not `_MainTable`) appears in the property sheet. It looks unnecessarily complicated, especially to FoxPro developers; but after you do it a few hundred times, you won't even notice it.

### Automating Data Access

ADO.NET is the data engine used in .NET applications. It's a disconnected methodology; you request data and close the door. When you want to save it, you reconnect and send the changes. Inside the application, data is stored in a dataset—sort of a miniature data environment, which can contain tables, relations, and some other elements. For our simple application, it will contain a single table, and the programmer will know the table's name. So we'll need a `MainTable` property. Open up the `BaseForm.vb` code module and enter the code shown in Listing 4.3. Rebuild the project and then rebuild the solution. Now open the inherited form in the designer, press F4, and look under Misc (see Figure 4.9): Voilà, there's your new `MainTable` property!

4



**FIGURE 4.9** Exposing a property in a subclass.

How do we use this property? We use it just exactly as we use properties in FoxPro; they're variables that the programmer can set while designing the form. I need two more properties and a constant before I can do what needs to be done, so I'll just fast-forward and list the entire declarations and property procedures code in one fell swoop, as shown in Listing 4.4.

#### LISTING 4.4 The BaseForm Inheritable Form Class

```
Public Class BaseForm

    Inherits System.Windows.Forms.Form

    #Region " My declarations "
        Public Const TurnOn As Boolean = True
```

**LISTING 4.4** Contintued

---

```
Public Const TurnOff As Boolean = False

Public ConnStr As String = _
    "Provider=SQLOLEDB;server=(local);database=Northwind;uid=sa;pwd="
Public dc As OleDb.OleDbConnection

Public daFiltered As OleDb.OleDbDataAdapter
Public dsFiltered As DataSet

Public daOneRecord As OleDb.OleDbDataAdapter
Public dsOneRecord As DataSet

Public _MainTable As String
Public _keyfield As String
Public _searchfield As String

Public spacer As String
#End Region

#Region " My Property procedures "
Public Property MainTable() As String
    Get
        Return _MainTable
    End Get
    Set(ByVal Value As String)
        _MainTable = Value
    End Set
End Property

Public Property KeyField() As String
    Get
        Return _keyfield
    End Get
    Set(ByVal Value As String)
        _keyfield = Value
    End Set
End Property

Public Property SearchField() As String
    Get
        Return _searchfield
    End Get
```

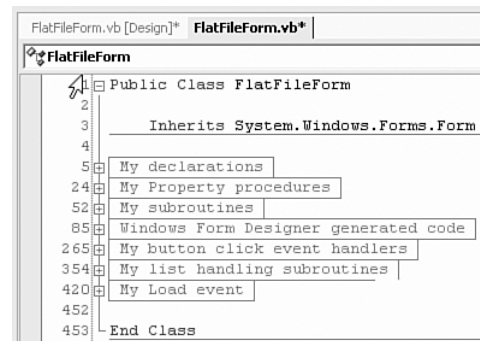
**LISTING 4.4** Contintued

```

        Set(ByVal Value As String)
            _searchfield = Value
        End Set
    End Property
#End Region

```

The #Region directives allow you to hide chunks of code. For example, when I collapse all of my code regions, this is what I see in the code editor for the BaseForm (see Figure 4.10). Needless to say, this is a lot easier to navigate than 453 lines of code.



4

**FIGURE 4.10** Collapsed code using #Region directives.

The additional public variables declared in the preceding code include constants to provide more meaningful symbols than True and False; a connection string and DataConnection to hook up to SQL Server; a couple of DataAdapters and datasets to get a list of candidate records and the single record the user selected, respectively; and public properties for the names of the Main Table, the key field (for retrieving the selected record), and the name of the searchable field to display in the ListBox.

The reason that everything has to be declared up front is Option Strict. The code won't compile unless we use DIM, PUBLIC, or PRIVATE (or FRIEND or whatever) to declare every single variable that we use in the code. IntelliSense uses these declarations to know what to show us when we hit that first period, and the compiler uses them to set aside storage.

ADO.NET uses a connection to build a DataAdapter. The DataAdapter contains Select, Insert, Update, and Delete logic to get the data to and from the data source specified by the connection. The data is stored inside your form in a DataSet object, which is like a data environment built of XML. It contains tables, relations, and other stuff that this exercise won't need. For our purposes, it contains a table, and its name is contained in the MainTable property. The KeyField, the one we'll use to retrieve a single record and to post

updates, is another named property, and `SearchField`, the field to search and to display in the list box, is named in the third property. Our code will refer to these three properties, trusting that the programmer has filled them in correctly.

Now we're ready to write some code. The `Load` event fires first in Visual Basic .NET, just as it does in FoxPro. We'll use the connection string to open the connection; then construct a `SELECT` statement and create a `DataAdapter`; then use the `DataAdapter`'s `Fill` method to fill a `DataSet` with a table named using the contents of the `MainTable` property. Listing 4.5 shows the code for the `Load` event.

---

**LISTING 4.5** The `BaseForm` `Load` Event Code

---

```
Private Sub BaseForm_Load( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles MyBase.Load
    Try
        Label2.Text = "in the " + SearchField + " field"
        dc = New OleDb.OleDbConnection
        dc.ConnectionString = ConnStr
        dc.Open()
    Catch oEx As Exception
        MsgBox("Connection failed: " + oEx.Message)
        Close()
    End Try
End Sub
```

---

**How `BaseForm` `Load` Works**

The program puts the name of the search field into the label at the top of the screen so that the display makes sense. Then it uses the connection string from the template form to open a connection to the data source, which could be SQL, ODBC, or anything else.

**Loading the List Box and Displaying a Record When Clicked**

The user gets a chance to filter the data in the table; if the table contains only a dozen or two records, it may not even be necessary. When the `Show Matching Records` button is clicked, the list is loaded and the first record in the list is displayed. Subsequently, clicking on any item in the list causes its record to be displayed. The code is shown in Listing 4.6.

---

**LISTING 4.6** The `LoadList` Button `Click` Event Code

---

```
Private Sub LoadList_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles btnLoadList.Click
    LoadTheList()
End Sub
```



**LISTING 4.6** Continued

```
Public Sub LoadTheList()  
    Dim I As Integer  
    Dim NumFound As Integer  
    Dim Str As String  
    Str = "SELECT * FROM " + MainTable _  
        + " WHERE UPPER(" + SearchField + ") LIKE '" _  
        + SearchValue.Text.ToUpper.Trim + "%'"  
    daFiltered = New OleDb.OleDbDataAdapter(Str, dc)  
    dsFiltered = New DataSet  
    daFiltered.Fill(dsFiltered, MainTable)  
    'Clear the listbox and load it  
    With ListBox1  
        .Items.Clear()  
        NumFound = dsFiltered.Tables(MainTable).Rows.Count - 1  
        Dim dr As DataRow  
        For I = 0 To NumFound  
            dr = dsFiltered.Tables(MainTable).Rows(I)  
            Str = dr(SearchField)  
            Str = Str.PadRight(40)  
            Str = Str + CStr(dr(KeyField))  
            .Items.Add(Str)  
        Next  
    End With  
    ListBox1.SelectedIndex = 0  
    LoadaRecord()  
    Buttons(TurnOn)  
End Sub  
  
Private Sub ListBox1_SelectedIndexChanged( _  
    ByVal sender As System.Object, ByVal e As System.EventArgs) _  
    Handles ListBox1.SelectedIndexChanged  
    LoadaRecord()  
End Sub  
  
Public Sub LoadaRecord()  
    Dim Kv As String  
    Kv = ListBox1.SelectedItem  
    Kv = Kv.Substring(40)  
    Kv = Kv.Trim  
    Dim str As String  
    str = "SELECT * FROM " + MainTable _
```

**LISTING 4.6** Continued

---

```

        + " WHERE " + KeyField + " = '" + Kv + "'"
daOneRecord = New OleDb.OleDbDataAdapter(str, dc)
dsOneRecord = New DataSet
dsOneRecord.Clear()
daOneRecord.Fill(dsOneRecord, MainTable)
Dim dr As DataRow
dr = dsOneRecord.Tables(MainTable).Rows(0)
' Load on-screen controls' text properties
Dim fldName As String
Dim Ctrl As Control
For Each Ctrl In Controls
    Try
        If TypeOf Ctrl Is TextBox Or TypeOf Ctrl Is ComboBox Then
            Ctrl.DataBindings.Clear()
            fldName = Ctrl.Name.Substring(3)
            ' skip characters "0-2"
            Ctrl.Text = dr(fldName)
        End If
    Catch ' ignore fields that don't have a column to bind to
    End Try
Next
End Sub

```

---

**How LoadList Click Works**

Clicking on the Show Matching Records button is handled by the `LoadList_Click` routine, which simply calls `LoadList()`. The routine creates a `SELECT` statement ending in a `LIKE` condition that matches any string starting with the letters the user typed in (try it with a single letter to start). The field that's searched is the one named in the `SearchField` property, which is also the field that's loaded into the `TextBox` item list. The key value for each record, `KeyField`, is appended to the end of the 40-character `SearchField` string, so that it's not visible. When the user clicks on the list, the key is extracted from position 41 (40 in VBSpeak) of the selected item and used to return a single record into the `dsOneRecord` dataset.

The challenge here was to bind the data to the fields on the screen. I used a little trick here; I assume that each field starts with a three-character mnemonic for the control type (txt for text box, cmb for combo box, and so on—a mechanism we're all pretty much used to anyway), and that the remaining characters are precisely the name of a field in the dataset. Datasets don't have field names, but the rows in the tables that they contain do; so I reference a row in the `Tables(MainTable)` collection, and then use `dr(FieldName)`—

where I just inferred *FieldName* from the control name—to find the data and assign it to the control's *Text* property. The *Try...Catch...End Try* with no code after the *Catch* is a neat trick; if the control doesn't have a matching field in the data row, it's an error, which I throw away.

## Utility Routines

There are a few routines that are used by several of the buttons' *Click* events, so I'll show them first. Listing 4.7 shows the code for the *Inputs* subroutine.

### LISTING 4.7 The Inputs Subroutine Code

```
Public Sub Inputs(ByVal onoff As Boolean)
    Dim Ctrl As Control
    For Each Ctrl In Controls
        If TypeOf Ctrl Is TextBox _
            Or TypeOf Ctrl Is ComboBox Then
            Ctrl.Enabled = onoff
        End If
    Next
    SearchValue.Enabled = Not onoff
    btnLoadList.Enabled = Not onoff
    Buttons(Not onoff)
End Sub

Public Sub Buttons(ByVal onoff As Boolean)
    btnAdd.Enabled = onoff
    btnEdit.Enabled = onoff
    btnDelete.Enabled = onoff
    btnClose.Enabled = onoff
    btnSave.Enabled = Not onoff
    btnCancel.Enabled = Not onoff
End Sub

Public Sub ClearFields()
    Dim Ctrl As Control
    For Each Ctrl In Controls
        If TypeOf Ctrl Is TextBox _
            Or TypeOf Ctrl Is ComboBox Then
            Ctrl.Text = ""
        End If
    Next
End Sub
```

### How the Inputs Subroutine Works

Inputs turns the text boxes, combo boxes, and other controls on and off as needed. I can pass it the constant TurnOn (True) to enable them or TurnOff (False) to disable them. Buttons ensures that, when the input fields are enabled, all of the buttons except Save and Cancel are disabled and vice versa. ClearFields blanks the input fields before adding a record.

## Click Event Code for the Form's Buttons

The buttons on the form give the users the options they need. We'll discuss the code one routine at a time. Listing 4.8 shows the code for the Click event of the Add button.

### LISTING 4.8 The Add Button Click Event Code

---

```
Private Sub btnAdd_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles btnAdd.Click
    Try
        dsOneRecord.Clear()
        BindingContext(dsOneRecord, MainTable).AddNew()
        ClearFields()
        Inputs(TurnOn)
    Catch oEx As Exception
        MsgBox("Error: " + oEx.Message)
    End Try
End Sub

Private Sub btnEdit_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles btnEdit.Click
    Inputs(TurnOn)
End Sub

Private Sub btnDelete_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles btnDelete.Click
    Try
        BindingContext(dsOneRecord, MainTable).RemoveAt(0)
        Dim cb As OleDb.OleDbCommandBuilder
        cb = New OleDb.OleDbCommandBuilder
        cb.DataAdapter = daOneRecord
        daOneRecord.UpdateCommand = cb.GetUpdateCommand()
    End Try
End Sub
```

**LISTING 4.8** Continued

---

```

    daOneRecord.Update(dsOneRecord, MainTable)
    dsOneRecord.Tables(MainTable).AcceptChanges()
    LoadTheList()
    MsgBox("Record deleted", MsgBoxStyle.Information, "My app")
Catch oEx As Exception
    MsgBox("Error: " + oEx.Message)
End Try
End Sub

```

---

**How the Button Code Works**

The Add button clears the text boxes and uses a `BindingContext` object to do the FoxPro equivalent of APPEND BLANK. Because the fields aren't bound to the data row in this exercise, I have to manually blank the onscreen controls, and finally I have to enable them. Edit is much simpler, of course. Listing 4.9 shows the code for the Click event of the Delete button.

4

**LISTING 4.9** The Delete Button Click Event Code

---

```

Private Sub btnDelete_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles btnDelete.Click
    Try
        BindingContext(dsOneRecord, MainTable).RemoveAt(0)
        Dim cb As OleDb.OleDbCommandBuilder
        cb = New OleDb.OleDbCommandBuilder
        cb.DataAdapter = daOneRecord
        daOneRecord.UpdateCommand = cb.GetUpdateCommand()
        daOneRecord.Update(dsOneRecord, MainTable)
        dsOneRecord.Tables(MainTable).AcceptChanges()
        LoadTheList()
        MsgBox("Record deleted", MsgBoxStyle.Information, "My app")
    Catch oEx As Exception
        MsgBox("Error: " + oEx.Message)
    End Try
End Sub

```

---

**How btnDelete Click Works**

Because the recordset is disconnected from the data source, I have two separate tasks: First, I mark the record deleted using the `BindingContext` object; then I use a `CommandBuilder` object to construct a Delete command object and use the Update method of the

DataAdapter object to pass it back to the data source. After updating the data source, I accept the changes to the dataset, clearing it, and reload the list using a call to `LoadTheList()`. You can't call a `.Click` method directly in Visual Basic .NET as you can in FoxPro, so that's why `LoadTheList` is a separate routine called both here and in the `LoadList_Click` method. Listing 4.10 shows the code for the `Click` event of the `Save` button.

**LISTING 4.10** The Save Button Click Event Code

```
Private Sub btnSave_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles btnSave.Click
    Try
        BindingContext(dsOneRecord, MainTable).EndCurrentEdit()
        Dim fldName As String
        Dim NewKey As String
        Dim Ctrl As Control
        For Each Ctrl In Controls
            If TypeOf Ctrl Is TextBox And Ctrl.Name <> "SearchValue" Then
                fldName = Ctrl.Name.Substring(3)
                'skip characters 0-2 - thanks, Bill..
                dsOneRecord.Tables(0).Rows(0).Item(fldName) = Ctrl.Text
                If fldName = KeyField Then
                    NewKey = Ctrl.Text
                End If
            End If
        Next
        Dim cb As OleDb.OleDbCommandBuilder
        cb = New OleDb.OleDbCommandBuilder
        cb.DataAdapter = daOneRecord
        daOneRecord.UpdateCommand = cb.GetUpdateCommand()
        daOneRecord.Update(dsOneRecord, MainTable)
        dsOneRecord.Tables(MainTable).AcceptChanges()
        ' Load the list so as to include the new record
        LoadTheList()
        ' Find the new key in the list
        Dim str As String
        Dim I As Integer
        For I = 0 To ListBox1.Items.Count - 1
            str = ListBox1.Items(I)
            If str.ToUpper.Substring(1).IndexOf(NewKey.ToUpper) > 0 Then
                ListBox1.SelectedIndex = I
            Exit For
        End For
    End Try
End Sub
```

**LISTING 4.10** Continued

```
        End If
    Next
    LoadaRecord()
    Inputs(TurnOff)
    Catch oEx As Exception
        MsgBox("Error: " + oEx.Message)
    End Try
End Sub
```

**How btnSave Click Works**

Saving the changes was the biggest challenge. First, I must end the current edit using the `BindingContext` object. Then, I have to put the values stored in the text properties of the onscreen controls back into the dataset. (I've fully qualified the row reference here so that you can see where it's going, but you can also use a `DataRow` object or even a `DataTable` object.)

Next I build an `Update` command and execute it using the `Update` method. Again, I reload the list to reflect any changes or an added record. Finally, I want the record that I was just editing or adding to be selected when the save is completed, so that's what the last `For . . . Next` loop is about. When I find the `SelectedIndex`, I call `LoadaRecord` and disable all of the input controls. Listing 4.11 shows the code for the `Click` event of the `Cancel` button.

**LISTING 4.11** The Cancel Button Click Event Code

```
Private Sub btnCancel_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles btnCancel.Click
    Try
        BindingContext(dsOneRecord, MainTable).CancelCurrentEdit()
        LoadaRecord()
        Inputs(TurnOff)
    Catch oEx As Exception
        MsgBox("Error: " + oEx.Message)
    End Try
End Sub
```

**How btnCancel Click Works**

The `Cancel` command only requires that I cancel the current edit using the `BindingContext` object (which, by the way, also encapsulates the functionality of

TableUpdate(), TableRevert(), DELETE, APPEND BLANK, TOP, BOTTOM, and SKIP), reload the record I was editing, and disable the input fields.

The last bit of code, which we saw at the beginning of this exercise, is shown in Listing 4.12.

---

**LISTING 4.12** The Close Button Click Event Code

---

```
Private Sub btnClose_Click( _  
    ByVal sender As System.Object, ByVal e As System.EventArgs) _  
    Handles btnClose.Click  
    Close()  
End Sub
```

---

### How to Use This Template


To use this template in your projects, either copy it to a directory where you've started a solution, or use Add an Existing Project by right-clicking on the solution in the Solution Explorer and adding it to the solution. All you really need is `InheritedForm.dll`, but I recommend you copy the code in case you want to make some improvements.

For each inherited form that you want to build, right-click on a project in the Solution Explorer and provide a name (usually the name of one of your data tables), then point to `InheritedForm.dll` (you may have to browse to it) and click on `BaseForm`. Fill in the `MainTable`, `KeyField`, and `SearchField` properties with the names of your main data table, key field, and the field you want users to search.

Finally, drag text boxes and/or combo boxes onto the form and name them with three-letter prefixes denoting the control type (for example, `txt`, `cmb`, `chk`) followed by the field names. Be sure to set the tab order using the View, Tab Order menu selection. (Hint: Click the same menu selection again when you're finished. That only took me an hour to discover.) Finally, disable all of the input fields. The user has to click on Add or Edit to change them.

When you've finished your new form, add a menu selection for the table in your Main Form `MainMenu` control using the pattern described earlier in the chapter, and rebuild the project, and it oughta work. The Customer form shown in Figure 4.11 took me two minutes and 15 seconds to build, from start to finish.





The screenshot shows a window titled "Customers". At the top, there is a search bar with the text "Show record containing" followed by a dropdown menu showing "0", and "in the CompanyName field". To the right of the search bar is a button labeled "Show matching records". Below the search bar, the form contains several fields: Customer ID (BLONP), Company Name (Blondesddel père et fils), Contact Name (Marketing Manager), Contact Title (Marketing Manager), Address (24, place Kléber), City (Strasbourg), State, Zip (67000), and Country (France). At the bottom of the form are buttons for "Add", "Edit", "Delete", "Save", "Cancel", and "Close". To the right of the form is a list of matching records: Berglunds snabbköp, Blauer See Delikatessen, Blondesddel père et fils, Bóldo Comidas preparada, Bon app', Bottom-Dollar Markets, and B's Beverages.

FIGURE 4.11 An Add/Edit/Delete form finished in less than three minutes.

## Summary

In this chapter, you saw how you can build inheritable forms, Visual Basic .NET's equivalent of template classes in FoxPro, thereby achieving the same rapid prototyping capability. You've seen how you can support both local tables and SQL Server with no change in the form code.

In Chapter 5, "Adding Internet Access," we'll extend these two models to include support for XML Web Services so that you can offer your clients the ability to run their rich client applications over the Internet.

