

# Chapter 9

## Form Controls

Every form needs interface controls. Selecting the right control for the job is not always an obvious decision. In this chapter you will examine the various controls you have for your forms, and see what they each do well and what they do poorly. You'll see some of the nuances associated with various controls and learn how to take advantage of them. Through designing your own control classes, you'll find out that there is very little that you cannot accomplish in user-interface design.

### The general use of form controls

First, let's examine each control in the Visual FoxPro toolbox in terms of its purpose and use. **Table 9.1** is a list of the available controls.

**Table 9.1.** The basic form controls available in Visual FoxPro

Control	Description
CheckBox	Used to indicate one of two states: on/off, yes/no, and so on.
ComboBox	A drop-down list. There are two styles: a list that limits the user to selecting items that are part of the available list, and a combo box that allows selection from the list or entry of values not in the list.
CommandButton	An action button. Usually used to initiate an action of some kind when the button is pressed.
CommandGroup	A group of CommandButton controls.
Container	A control that is able to contain other controls.
EditBox	Used for editing character values. Allows for scrolling vertically for editing amounts of text that do not easily fit within the space allotted in the form.
Grid	Multi-record editing control that presents data from many records in a row (record) and column (field) display. The grid is a compound control that is comprised of columns, which in turn contain headers and controls.

Control	Description
HyperLink	A HyperLink object can request that an ActiveX hyperlink-aware container, such as Microsoft Internet Explorer, jump to a particular URL.
Image	Displays an image.
Label	Displays text that cannot be changed interactively by the user.
Line	Displays a horizontal, vertical, or diagonal line.
ListBox	Displays a list of items from which one or more may be selected.
OLEControl	Hosts OLE or ActiveX objects in a Visual FoxPro form.
OLEBoundControl	Displays the contents of a general field in a DBF file.
OptionButton	Can only be added to an OptionGroup.
OptionGroup	A container for OptionButtons. Allows the user to select one from a set of mutually exclusive choices.
PageFrame	A container for holding Pages. A PageFrame is sometimes referred to as a tabbed dialog.
Page	A single tab within a PageFrame.
Shape	A graphical object that can display a box, circle, or ellipse.
Spinner	Allows the setting of a numeric value by “spinning” through a range of acceptable values. Numbers can also be typed into the spinner.
Timer	An invisible control that allows for the execution of code at a certain time interval.
TextBox	A control that allows the editing of data.

The following sections will examine many of the Table 9.1 controls in more detail, focusing on the uses of these controls for user-interface design. I'll discuss when to use which control and how to make the controls exhibit the behavior you want.

## Label

The label is one of the simplest controls at your disposal. It is most often used for displaying prompt information alongside other controls. Because it is “fixed” text, it's well adapted for the prompt job.

The label is often underestimated, though. It can do quite a bit more than prompt the user. Even though it isn't editable by the user, you can change its Caption property at will in your code. You could use a label for displaying non-editable data by simply setting its caption property whenever a new record is encountered.

For example, assume you have a customer table with a field named *balance*. This balance field can't be edited because it is a calculated value from other tables. You could put a label in the form, make its name `lblBalance`, and enter the following in the form's refresh method:

```
THISFORM.lblBalance.Caption = TRANSFORM(Customer.Balance,  
"$999,999,999.99")
```

You need the `TRANSFORM()` in the above code because label captions can only accept character values.

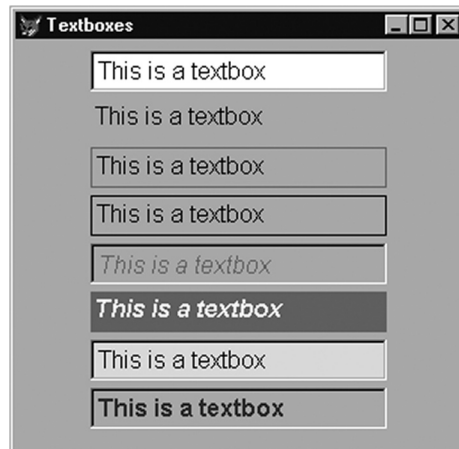
Another strength of the label control is that it's a very lightweight object, meaning that it doesn't take a lot of memory to create. This makes it a candidate for being the base class for many non-visible classes. Add to its light weight the fact that it can have a descriptive caption that is seen in the form designer, and the label becomes an even better candidate for non-visible objects. You make the label non-visible by setting its visible property to `.F`.

When designing a label class, you can add to it any methods or properties that you need, so the fact that it is a label does not limit its functionality.

## TextBox

The `TextBox` is the workhorse control for data entry. It is universal in the types of data it can handle and its user interface is simple. Users naturally seem to know how to use a `TextBox`; they type in it.

You could say that the `TextBox` is also an underestimated control. Its flexibility for data entry is often not fully appreciated. You have virtually unlimited control over the look of a `TextBox`. **Figure 9.1** shows the *Textbox* form from the sample code.

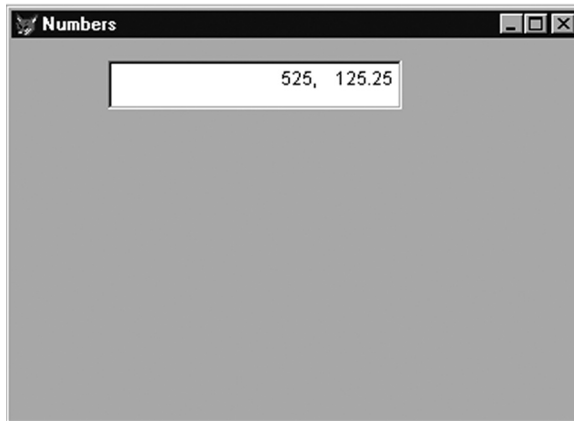


**Figure 9.1.** The Textbox form.

Figure 9.1 is an example of the variety of ways that a TextBox can look. It can have a border or not, the border can be 3-D or plain, and the color schemes and fonts can be modified. All of these features of the TextBox make it a versatile control for your forms. But, I haven't even scratched the surface of it yet.

The TextBox will inherently process the user's keystrokes. In most situations this is exactly what you want, and you don't have to do anything special to handle the user's typing. But there are circumstances where creating a TextBox class and overriding the inherent key processing can be beneficial.

How often have you used a TextBox for entering numeric values? You want the value formatted with commas and a decimal point, so you set the InputMask property to "999,999,999.99" and hope for the best. The problem is that when the user starts typing into the TextBox, the formatting gets messed up until they finish. See **Figure 9.2** for an example.



**Figure 9.2.** Messed up numbers in a TextBox.

Notice the "525" in the TextBox in Figure 9.2. It looks weird. The TextBox started with the value 125.25 in it and I typed 525 at the beginning of the TextBox to get the result shown in Figure 9.2. Users are easily confused by this formatting problem. There is no native property setting of the TextBox that will correct the problem, either.

In this chapter you'll build a TextBox class that will provide for number entry similar to a calculator, where the digits enter the TextBox from the right to the left and are correctly formatted as they are entered. The sample project has a TextBox class named txtNumber in the Controls class library. This class was built to intercept the user's keystrokes and process them in a special fashion. **Listing 9.1** shows the key-press code for this class.

**Listing 9.1.** The keypress event code for the numeric textbox.

```

LPARAMETERS nKeyCode, nShiftAltCtrl
DO CASE
CASE nKeyCode >= 48 AND nKeyCode <= 57
    * Digits 0 to 9
    NODEFAULT
    THIS.cSoFar = THIS.cSoFar + CHR(nKeyCode)
    THIS.SetValue()
CASE nKeyCode = 43
    * + key
    NODEFAULT
    IF LEFT(THIS.cSoFar,1) = "-"
        THIS.cSoFar = SUBSTR(THIS.cSoFar,2)
        THIS.SetValue()
    ENDIF
CASE nKeyCode = 45
    * - key
    NODEFAULT
    IF LEFT(THIS.cSoFar,1) <> "-"
        THIS.cSoFar = "-" + THIS.cSoFar
        THIS.SetValue()
    ENDIF
CASE nKeyCode = 9 OR nKeyCode = 15
    * Tab and Backtab
CASE nKeyCode = 7
    * Del
    NODEFAULT
    THIS.cSoFar = "0"
    THIS.SetValue()
OTHERWISE
    * Ignore all other keys
    NODEFAULT
ENDCASE

```

Notice the use of `NODEFAULT` in the code above. This command prevents Visual FoxPro from executing its default behavior for the `KeyPress` event. You use the `NODEFAULT` so you can process these keystrokes yourself.

The `txtNumber` class has two added properties named `cSoFar` and `nDecimal`. The `cSoFar` property is used to store a character representation of what the user has typed so far, and the `nDecimal` property is used to store a divisor used to position the decimal point. The `cSoFar` is set to "0" in the `GotFocus` so that users always replace the entire current value when they type in the control. `nDecimal` is set in the `Init` for the control by finding the decimal point in the `InputMask` property and doing a calculation. A `SetDecimal` method has been added to the class; it contains the following code:

```

LOCAL lnDec
lnDec = LEN(ALLTRIM(THIS.InputMask)) - ;
        RAT(".",ALLTRIM(THIS.InputMask))
THIS.nDecimal = MAX(1,10 ^ lnDec)

```

The Init event has one line that calls `THIS.SetDecimal()`. The `InputMask` property has been given an assign method so that any time the value of the `InputMask` property is changed, the assign method will run. The code for the assign method is listed below.



Visual FoxPro gives us two special types of methods: assign and access. These methods allow you to trap activities involving properties and write code to manage them. An assign method is executed whenever there is an attempt to assign a value to a property and it is passed the value that is being assigned. The access method is executed whenever there is an attempt to read the value of a property and the `RETURN` value of that method is the value that the code accessing the property will see.

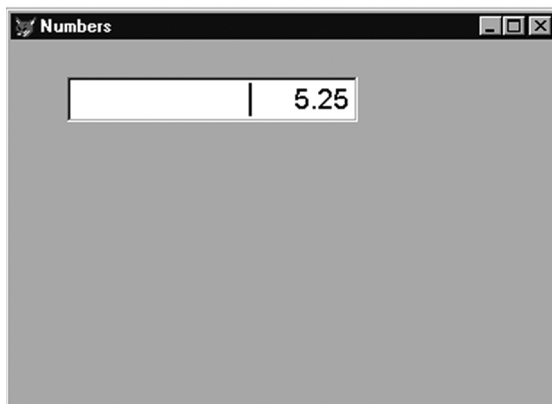
These methods are named `<PropertyName>_access` and `<PropertyName>_assign`. In the numeric `TextBox`, I used a method named `InputMask_assign` so it will react to any changes made to the `InputMask` of the `TextBox` control at runtime.

```

LPARAMETERS vNewVal
THIS.InputMask = m.vNewVal
THIS.SetDecimal()
THIS.SetValue()

```

**Figure 9.3** shows a form using the `txtNumber` class. In this form I typed the same 525 that I typed earlier.



**Figure 9.3.** The `txtNumber` class in use.

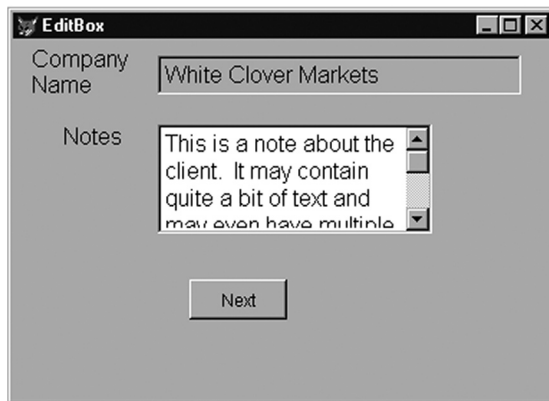
Notice that the display now shows 5.25. This is because the `InputMask` of 999,999.99 is setting the decimal point. To get 525.00, I would type 52500 in the `TextBox`.

The key point of this class is to demonstrate the fact that you are not limited in the usage of any control in Visual FoxPro. You can override the default behavior and provide your own behavior, thereby creating a new control. When you feel frustrated by the way things are happening, stop and ask yourself, “How can I make it happen the way I want it to happen?”

## EditBox

Similar to the TextBox, the EditBox is a fairly simple editing control: It can be used to present and edit character data. The EditBox has the additional functionality of being able to present data that is too large to fit neatly inside the control. With vertical scrollbars on the right side, this control allows the user to scroll through large amounts of character data.

When you need to present free-form character data, such as the contents of a memo field, the EditBox really shines. **Figure 9.4** shows a form with an EditBox being used to edit the mNotes field of the Clients table in the Time and Billing sample data.



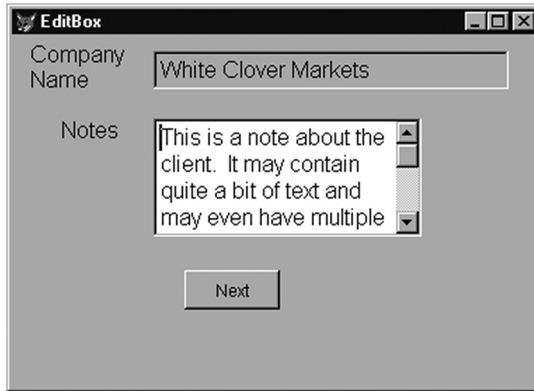
**Figure 9.4.** An editbox used for a memo field.

The scroll bars on the right are enabled, indicating that more data follows what you can see. Note the way the last visible line is cut in half. This occurs because of the height of the EditBox and the FontSize used. The spacing of the lines causes a partial display of the last line.

You can fix this partial display by making the height of the EditBox neatly fit the line height for the font and FontSize being used. You may see that as a painful process, calculating the height of the font and then figuring out the correct control height. Lucky for us, Microsoft gave us a property that handles this specific problem: the `IntegralHeight` property.

When `IntegralHeight` is set to `.F.` (the default), the results are similar to Figure 9.4. However, setting `IntegralHeight` to `.T.` will resize the control to fit an exact number

of lines. **Figure 9.5** shows the same form, only this time the `IntegralHeight` for the `EditText` is set to `.T.`



**Figure 9.5.** Using `IntegralHeight` on an `EditText` control.

The `EditText` allows you to control just about every detail of its appearance. You can make the scrollbar go away by setting the `ScrollBars` property to `0—None`. You can affect the appearance of the border around the `EditText` using the `BorderStyle`, `BorderColor`, and `SpecialEffect` properties. A full complement of events and methods are available in the `EditText` for controlling the behavior of the control.

To view the `EditText` as a tool only for editing memo fields would greatly limit the utility of this control. A little imagination will reveal some uses for this control that go well beyond its obvious utility for editing memos. **Figure 9.6** shows an `EditText` being used in quite a different way.



**Figure 9.6.** Using an `EditText` for a unique purpose.

The text in **Figure 9.6** is in an `EditText`. The form is in the sample project on your CD and is named `EditBox2.scx`. The code in the `Start` button's click method is a very simple simulation of some processing. That code is listed here:



```
THISFORM.Edit1.Value = "Starting"
LOCAL lnCnt
FOR lnCnt = 1 TO 5
    INKEY(1.0)
    THISFORM.Edit1.Value = THISFORM.Edit1.Value + CHR(13) + ;
        "Process " + LTRIM(STR(lnCnt)) + " Completed"
ENDFOR
THISFORM.Edit1.Value = THISFORM.Edit1.Value + CHR(13) + "Finished"
```

**Table 9.2** shows the non-default property setting for the `EditText` in Figure 9.6.

**Table 9.2.** The property settings for the `EditText` in the `EditText2` form.

Property	Setting
BackStyle	0 – Transparent
BorderStyle	0 – None
FontBold	.T.
FontSize	12
ForeColor	255,0,0
IntegralHeight	.T.
ReadOnly	.T.
ScrollBars	0 – None

The `EditText` control, like all of the other controls, has its obvious use and also some not-so-obvious uses. “Thinking outside the box” will allow you to exploit the unique qualities of any control to suit your requirements, and the `EditText` is no different in this respect.

## Spinner

The `Spinner` is designed for numeric data types. It allows the user to input or change its value in two ways: by typing directly into the control or by using the up- and down-arrow buttons at the right side of the control. **Figure 9.7** shows a form with a `Spinner` control.



**Figure 9.7.** The `Spinner` control.

The `Spinner` in Figure 9.7 has some property settings that are worth mentioning. Table 9.3 lists the non-default properties and their settings.

**Table 9.3.** Property settings for the Spinner seen in Figure 7.

Property	Setting
Increment	0.25
InputMask	999.99
KeyboardHighValue	100
KeyboardLowValue	0
SpinnerHighValue	100
SpinnerLowValue	0

Some other properties are set to values other than their defaults, like Top, Height, Name, and so on; however, those properties are not important to the behavior of the control. Table 9.3 lists the important properties.

The Increment property controls how much the value of the control will be increased or decreased when the spin buttons are used. The up- and down-arrow keys have the same effect as the respective spin buttons when the Spinner has the focus.

Inputmask has the same effect on a Spinner as it does on any other control: it defines the display mask for the data. In the Spinner above, the Inputmask defines a display of three digits, a decimal point, and two digits.

The next four properties seem to be two sets of two properties, and in fact they are. Why two sets? Users can alter the value of the Spinner in two distinctly different ways. They can use the keyboard to type into the control, or they can use the Spinner buttons. The Keyboard properties control the range of values that can be typed in, and the Spinner properties affect the range of values for the spin buttons.

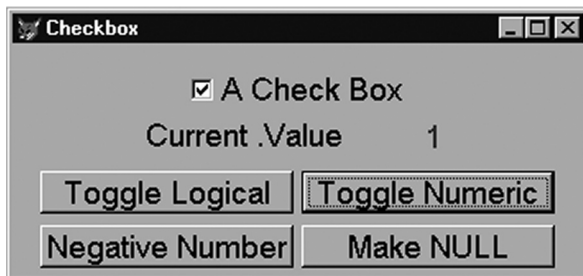
Hmm, now wait one minute ... why would these two sets of properties be set differently from one another? It seems unlikely that you would want them to be different, but “unlikely” is not the same as “never.”

Here’s one example where you might want these properties to have different ranges. Suppose you are using a Spinner to present the data for a bowling alley’s number of strings for a league game. The business rules say that the range for the number of strings is always between 1 and 3. An additional rule says that in rare situations the number of strings may be more or less than that range. Your analysis finds that 97 percent of the leagues have a number of strings within the range. So you decide to use a Spinner and set the SpinnerHighValue to 3, the SpinnerLowValue to 1, the KeyboardHighValue to 15 and the KeyboardLowValue to 1. This allows the user to spin within the common range but prevents an unusually high number from being entered without the user expressly typing that number into the control.

## CheckBox

This is an easy one. The CheckBox is either checked or not checked. It is a two-state control that can be used for such data entry tasks as true/false, on/off, or yes/no. Seems simple enough.

The CheckBox supports two data types: logical and numeric. When using a logical data type, true is represented by a checked state of the control, and false is unchecked. For numeric data types, 0 represents unchecked and non-zero (any value other than 0) represents the checked state. **Figure 9.8** shows the form in the sample project named Check.scx.



**Figure 9.8.** A CheckBox in the Check form.

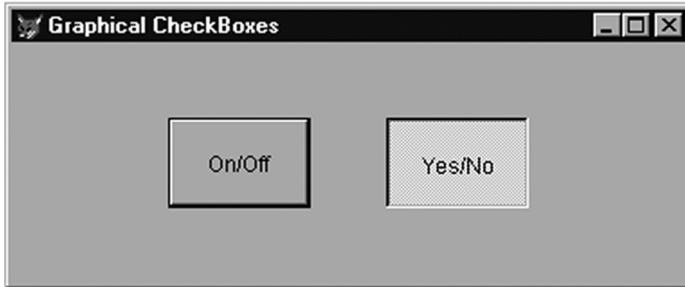
The form in Figure 9.8 was built to demonstrate the possible values for a CheckBox control. The Toggle Logical and Toggle Numeric buttons alter the values between .T./.F. and 0/1, respectively. By running the form and clicking the buttons, you can see the checked state of the CheckBox change. The Current Value line in the form shows you the current value for the CheckBox. Clicking on the CheckBox will toggle the value just like the buttons do. However, note that clicking the CheckBox toggles the value within the current data type for the value property of the CheckBox, while using the buttons will change the data type.

The Negative Number button assigns a value of  $-1$  to the CheckBox. When you click that button, the CheckBox's state becomes checked. But you'll see that the CheckBox takes on the disabled colors. Clicking the CheckBox will show you that the control is not disabled. You can only assume that the disabled colors are used to visually show negative numbers differently than positive numbers. Setting the value to a negative number reinforces the fact that a value of 0 will be unchecked and any value other than 0 will be checked.

Clicking the Make NULL button assigns a value of NULL to the CheckBox. The value of NULL displays the same as a negative number; checked with disabled colors. The NULL-valued CheckBox isn't really disabled, though. Clicking the NULL-valued CheckBox toggles it to the unchecked value for the data type, a logical .F. or numeric 0.

## Graphical CheckBox

A graphical CheckBox is not another control; it is just a CheckBox with the Style property set to 1—Graphical instead of 0—Standard. So what is a graphical CheckBox? **Figure 9.9** shows a pair of graphical CheckBoxes, with the left one unchecked and the right one checked.



**Figure 9.9.** Graphical CheckBoxes in the CheckG form.

The Figure 9.9 form is the CheckG form in the sample project on the CD. Graphical CheckBoxes are those buttons that “stay pushed” when you click them. Other than their appearance, they are no different from the standard CheckBoxes.



One issue that relates to both the CheckBox just discussed and the Option-Group discussed later in this chapter is that they act differently when the user presses the Enter key to move to the next control. Both the CheckBox and the OptionButton change the selection status of the control before moving on when the Enter key is pressed. This often causes a user to grab the mouse to navigate through these controls and can therefore slow down data entry. It is wise to avoid both of these controls in a “heads-down” data-entry form.

## ComboBox

The ComboBox is the first of the *complex controls*, those that have more than one part to their interfaces. The ComboBox has the data-entry box and the drop-down list as its two parts.

The ComboBox is a very flexible and useful control with two distinct personalities that are controlled by its Style property. One of these personalities is the *drop-down combo* and the other is the *drop-down list*. Both styles provide a list of values in the ComboBox from which the user can make a selection. The drop-down combo style also allows the user to type a value into the control that is not included in the list. The drop-down list restricts the user’s selection to only those items in the list. In a later chapter there is a detailed discussion of the ComboBox, along with other complex controls. Here I will focus on the general aspects of the control and the “when and where” of using it.

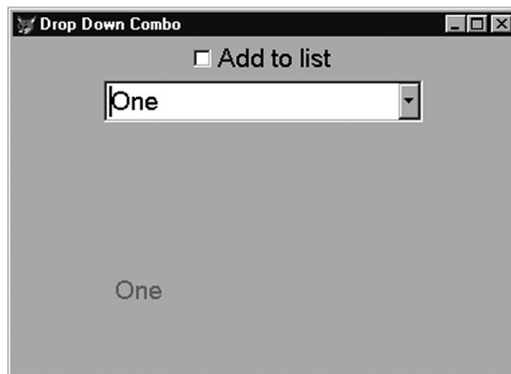
## Drop-down combo

The „drop-down combo” style ComboBox allows the user the ease of selecting an entry from a list of possible values but doesn't limit him to selections from that list. That is the party line on the drop-down combo, but in truth things won't work that way without some coding on your part.

Both combo styles have two properties that hold values. The Value property is connected to the ControlSource for the combo and the DisplayValue is the value that is visible when the combo's list is not dropped down. The DisplayValue and the Value will only be the same when the DisplayValue is found in the list. If the DisplayValue is not found in the list, then the Value will be blank.

If you choose the drop-down combo style, thinking that the user can type in any value and that value will be saved to the ControlSource, you are mistaken. Unless you have written some code to add the DisplayValue to the combo's list when it isn't found in the list, the typed value won't be stored in the ControlSource.

In the sample project, the form named *DDCombo.sxc* was built to demonstrate the effects just described. **Figure 9.10** illustrates this form.



**Figure 9.10.** Drop-down combo demonstration form.

The form has three controls in it: a CheckBox that is used to turn on and off the ability to add a value to the combo's list, the combo itself, and a TextBox at the bottom that shows the value of the ControlSource for the combo. The combo has its ControlSource set to a property (cValue) that has been added to the form, so that changing the value of the combo will change the value of the form's cValue property. The TextBox has its ControlSource set to the same form property as the combo.

Here is the code in the combo's Init that populates the list with values:

```
THIS.AddListItem("One",this.NewItemId+1)
THIS.AddListItem("Two",this.NewItemId+1)
THIS.AddListItem("Three",this.NewItemId+1)
```

This combo has its `RowSourceType` set to *none* (there will be more discussion of `RowSourceType` later) and uses the `AddListItem` method to add items to the list. The `When` event of the combo contains the following code:

```
THISFORM.txtcValue.Refresh()
```

This line simply refreshes the display of the `TextBox` each time the selected item in the combo is changed. Finally, there is the code in the `Valid` event:

```
IF THISFORM.chkAdd.Value
    * If the checkbox is checked
    IF NOT ALLTRIM(THIS.DisplayValue) == ALLTRIM(THIS.Value)
        * Declare variable to temporarily hold the DisplayValue
        LOCAL lcValue
        * Save the DisplayValue
        lcValue = THIS.DisplayValue
        * Add the DisplayValue to the list
        THIS.AddListItem(THIS.DisplayValue,THIS.NewItemId + 1)
        * Requery the combo's list
        THIS.Requery()
        * Set the combo's Value to be the same as the DisplayVaue
        THIS.Value = lcValue
        * Now refresh the textbox
        THISFORM.txtcValue.Refresh()
    ENDIF
ENDIF
```

This code first checks to see if the `CheckBox` on the form is checked; if not, the code does nothing. If the `CheckBox` is checked, the code then looks to see if the `DisplayValue` and the `Value` properties have different values. If these two properties have the same value, then you don't need to do anything. The `DisplayValue` is different from the `Value` when the user types a string that isn't in the list. In that situation, you execute the code inside the `IF` statement in order to add the item to the combo's list, requery the combo's list, and then set the `Value` property to the same value that the `DisplayValue` property had. Finally, the `TextBox` is refreshed.

If you run this form and try using the `ComboBox` to enter the value of "Four" with and without checking the `CheckBox`, you'll see for yourself the difference in behavior. If the `CheckBox` is checked, the value of "Four" will be put into the form's `cValue` property by the `ComboBox`. When the `CheckBox` is not checked, the value of "Four" appears when you type it in, but that value is lost once you try to leave the `ComboBox`. Both the `ComboBox` and the `TextBox` show a value that is blank.

In the next section, you'll see the various types of row sources that a combo can have. The code in the above example works well when the `ComboBox` has a `RowSourceType` of `None`, meaning that the combo manages its own list internally. In the example, you used the `AddListItem` method to populate the list. If you had used one of the other `RowSourceTypes` available, you would have then needed to add the typed value to the `RowSource` for the combo, rather than using the `AddListItem`.

## Drop-down list

The other style for a ComboBox is „drop-down list.” The drop-down list does not allow the user to type values that are not in the list, but forces the user to make a selection from the list. This style is useful in situations where you know the available options ahead of time, so that only the valid choices can be included in the list.

You can obtain the available choices from a number of different places. They might be in a table, or obtained through a SQL SELECT command. The RowSource property controls the source for the list items. The RowSourceType setting dictates the acceptable values for the RowSource property. The possible values for the RowSourceType property are shown in **Table 9.4**.

**Table 9.4.** RowSourceType settings for Lists and ComboBoxes.

RowSource-Type	RowSource
0 – None	There is no RowSource; the list is filled using the AddItem and/or the AddListItem methods.
1 – Value	The RowSource will be a comma-delimited list of values, such as “Apples, Oranges, Bananas.”
2 – Alias	RowSource will be an alias name for an open table or cursor. The ColumnCount and ColumnWidth properties can be used to control the fields that are displayed.
3 – SQL Select	The RowSource will contain a SQL SELECT command that produces a table or cursor—perhaps something like “SELECT Name FROM Customer WHERE State = ‘NY’ INTO CURSOR MyList.”
4 – Query (QPR)	RowSource will contain the name of a .QPR file. QPR files are produced from the Query wizard and contain a SQL SELECT command.
5 – Array	RowSource will contain the name of a memory variable array (or an array property of an object). The ColumnCount and ColumnWidth properties can be used to control the display of multiple columns from the array.
6 – Fields	RowSource will contain a comma-delimited list of field names, which may or may not include the alias and period. Example: “Name, Address, State” or “Customer.Name, Address, State”. Notice that when using the alias name, only the first field in the list gets it. Repeating the alias name on the other field names will cause an error at runtime. The list will be populated from the values of the fields in the list.
7 – Files	The ComboBox will be populated with a list of files in the current directory. The RowSource property can contain a filename mask such as “*.DBF” or “*.TXT” to limit the filenames shown in the list.

RowSource-Type	RowSource
8 – Structure	The list will contain the field names for the table whose alias name is in the RowSource. If the RowSource is left blank, then the currently selected alias will be used to obtain the field names for the list.
9 – Popup	The items in the list will be obtained from a popup menu's bars as defined with the DEFINE POPUP and DEFINE BAR commands. This option for RowSourceType is present in the product for backward compatibility and should be avoided.

With all these choices, how do you decide what to use? The most common four choices by far are None, Alias, SQL Select, and Array. The others may be useful in certain situations; when you're faced with one of those situations, you'll know it.

The Alias, SQL Select, and Array settings are self-explanatory, but where does the setting of None come in? To be truthful, the None setting is probably the most flexible of all the settings. It requires that the AddListItem or AddItem methods be used to populate the list of the control. This may initially seem to be a hindrance, but although it requires you to populate the list, it also gives you complete control over what gets into the list.

For example, what if a certain alias has all of the possible choices in it? It might be easier to just use the Alias RowSourceType and be done with it. But what if not all of the values in the alias are valid in all situations? This would require using the Valid method of the control to disallow making certain choices from the list. The problem with this technique is that a list control shouldn't include items that the user cannot choose. By using the None setting, you could write code in the Requery method of the control that would scan the alias and call the list's AddListItem method to add only those values that were valid choices. That way, the user could never choose an invalid value from the list.

There are a few other benefits to the None RowSourceType. Along with the Value RowSourceType, None can have its Sorted property set to .T. to provide a sorting of the items in the list. In a List control, the MoverBars can be used to rearrange the order of the items in the control if the RowSourceType is set to None or Value.

## ListBox

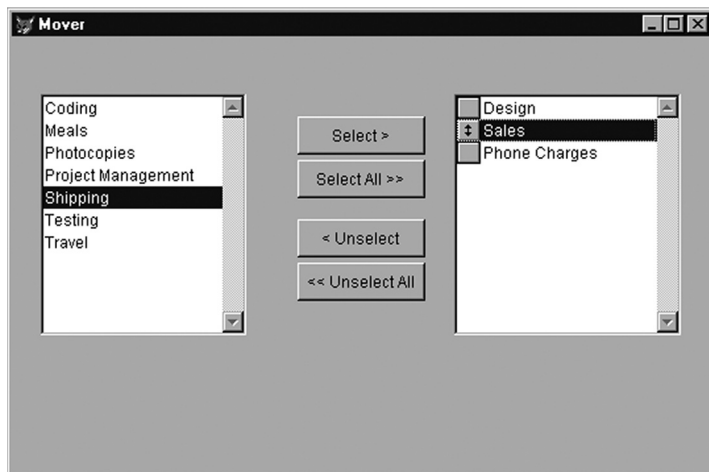
The ListBox control is similar to the ComboBox in that it presents a list of values from which the user can choose. The difference is that with the ListBox, more than one item from the list is visible all the time, and the selected item(s) is(are) highlighted. The ComboBox only shows the list when the list has been "dropped down;" otherwise it shows only the currently selected value. This means the ListBox will take up more real estate in the form. Besides the space requirement, the ListBox may be



inappropriate because it can cause confusion for the user by showing multiple items. These are considerations for choosing a `ListBox` over a `ComboBox` in your interface. Keep in mind, though, that what is a problem in one situation may be exactly what is needed in another situation. One feature of a `ListBox` is the ability to allow multi-selection; that is, to allow the user to select more than one item in the list. In this situation, it's important to make it clear to the user what items are currently selected. The `ListBox` can do this by highlighting multiple items in the list. Later in this chapter I'll show you a better method of multi-selection than just the `ListBox`.

The `ListBox` and `ComboBox` controls are both valuable when you need to present a list of possible choices to the user. Their limitation is that they must keep their entire list in memory; therefore, they aren't very good candidate controls when the possible list of choices is greater than between 100 and 500. Of course, if you think about it from the user's perspective, a `ListBox` or `ComboBox` is not an easy way to make a choice from a large number of possibilities. I try to limit my lists to fewer than 100 items, believing that any more than 100 choices would be better presented to the user in a different way.

The `ListBox` has the same choices for `RowSourceType` as the `ComboBox`: `Alias`, `Array`, `SQL SELECT Statement`, and `None`. The sample code contains a list form named *MoverJB.scx*. **Figure 9.11** shows this form with both lists populated.



**Figure 9.11.** A mover dialog using two `ListBox`s.

The interesting thing about this mover dialog is that both of the `ListBox`s have `None` as their `RowSourceType`. The population of the lists is done through program code. Selecting and unselecting are handled in methods of the form. Here is the code that originally populates the `ListBox` on the left:

```

LOCAL lcAlias, lnNextItem
lcAlias = ALIAS()
SELECT SystemCodes
WITH THISFORM
    WITH .lstSource
        .Sorted = .F.
        .Clear()
        SCAN
            lnNextItem = .NewItemId + 1
            .AddListItem( cDescription, lnNextItem,1)
            .AddListItem( cType, lnNextItem,2)
            .AddListItem( STR(icode_id,4,0), lnNextItem,3)
        ENDSCAN
        .Sorted = .T.
    ENDWITH
    .ResetButtons()
ENDWITH

```

Notice the use of the `NewItemId` property of the `ListBox` to get an `ItemId` for adding an item. Using `AddListItem` rather than `AddItem` allows you to add multiple columns of information. Also note the use of the `Sorted` property to sort the items in the list.

The code below will select one item from the list on the left and move it to the list on the right. This method of the form is named `SelectOne`. It is called from the `Select One` button as well as the `DbClick` event of the left list. Putting this code to a form method allows us to call it from two different places providing the user with more than one way to move an item to the selected list.

#### **Listing 9.2.** The `SelectOne` method of the form.

```

LOCAL lcCol1, lcCol2, lcCol3, lnSourceItem
WITH THIS
    * Shut off sorting to speed things up
    .lstSource.Sorted = .F.
    WITH .lstSource
        * Pick up values from source list
        lcCol1 = .ListItem( .ListItemId, 1 )
        lcCol2 = .ListItem( .ListItemId, 2 )
        lcCol3 = .ListItem( .ListItemId, 3 )
        .RemoveListItem( .ListItemId )
    ENDWITH
    WITH .lstTarget
        * Add the item to the target list
        lnItemId = .NewItemId + 1
        .AddListItem( lcCol1, lnItemId, 1 )
        .AddListItem( lcCol2, lnItemId, 2 )
        .AddListItem( lcCol3, lnItemId, 3 )
    ENDWITH
    * Set focus to the target list
    .lstTarget.SetFocus()
    * Sort the source list

```

```
.lstSource.Sorted = .T.  
* Process all events  
DOEVENTS  
* Now reset the buttons  
.ResetButtons()  
ENDWITH
```

This code first saves the three columns of information from the source list to variables. It then removes the item from the source list. Next it adds an item to the target list with the variables as the data.

Notice that the source list's sorted property is first set to .F. and then reset to .T. after everything is done. This is because if the list is sorted while you manipulate its contents, everything slows down while Visual FoxPro tries to keep the list items sorted.

The SelectAll method of this form is essentially the same as the SelectOne method, except that it moves all items in the source list to the target list. The Unselect methods are also similar, except they move the data the other way.

Next, there is the DoEvents command, which tells Visual FoxPro to process all pending events in the event queue. This command is needed here to deal with a timing problem related to the ResetButtons method that is called next. As you'll see in the ResetButtons code, you will refer to the items in the two lists to determine which buttons should be enabled and disabled. Calling the DoEvents command will ensure that the proper settings have affected the lists before you run the ResetButtons method.

The ResetButtons method is used to enable and disable the buttons for selecting and unselecting items from the lists. Here is the code for this method:

```
WITH THIS  
* Enable the Unselect buttons only if there is  
* at least one item selected  
.cmdUnselectOne.Enabled = (.lstTarget.ListItemId <> 0)  
.cmdUnselectAll.Enabled = (.lstTarget.ListCount > 0)  
  
* Enable the Select buttons if there is  
* at least one item in the source list  
.cmdSelectOne.Enabled = (.lstSource.ListItemId <> 0)  
.cmdSelectAll.Enabled = (.lstSource.ListCount > 0)  
  
ENDWITH
```

One last note about the mover dialog: notice the button-like objects to the left of each item in the selected list on the right. These are MoverBars, which allow the user to rearrange the order of the list items. You could use code to read the items in the selected list from top to bottom and do something with the data. These MoverBars allow the user to determine in which order those things will be done. MoverBars are available only when the RowSourceType for the list is either None or Value. Any other choice of RowSourceType will force the list's MoverBars property to be .F.

## OptionGroup

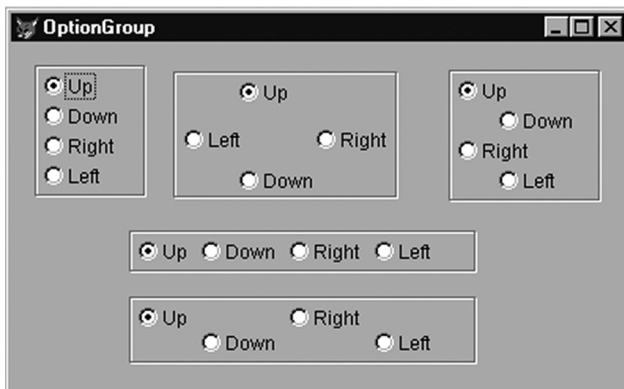
The OptionGroup used to be called radio buttons, which were named after the '50s and '60s car radios that had a front panel of buttons with which to select stations. The OptionGroup is used when there is a short list of options that are mutually exclusive. The OptionGroup can contain any number of OptionButtons. Each OptionButton represents a single choice. Only one OptionButton can be “selected” at any one time; selecting a different OptionButton will deselect the previously selected one.

Where do these OptionGroups fit into user-interface design? Not very many places, because they tend to slow down the data-entry person rather than speed him up. The OptionGroup has to be manipulated differently from the other controls, and that alone will slow down a user. I find myself using OptionGroups in some of the forms I design for report criteria setup, because in those forms there are often opportunities to use the OptionGroup effectively. However, a TextBox or short list is more effective in common data-entry situations, such as *marital status*, which could have Single, Married, Widowed, or Divorced as choices.

Okay, so how do they work? The OptionGroup is a specialized container that is limited to containing only OptionButtons. By default the OptionGroup has two buttons in it, and you can vary the number of buttons by adjusting the OptionGroup's ButtonCount property. The OptionGroup has a Value property that contains the number of the currently selected button.

Another useful property of the OptionGroup is Autosize. Setting this property to .T. will size the OptionGroup box to surround the contained buttons perfectly. If you change the ButtonCount, the OptionGroup will resize itself.

The buttons inside the OptionGroup do not need to be placed vertically in a column, although that is their default positioning. **Figure 9.12** shows a number of different arrangements of the OptionButtons in an OptionGroup.



**Figure 9.12.** Various configurations of OptionGroups.

## CommandButton and CommandGroup

I'm going to address the `CommandButton` and `CommandGroup` controls at the same time because they relate so closely to each other. The `CommandButton` is used for starting an action like saving the user's work or closing a form. The `CommandGroup` is a special container that, similarly to an `OptionGroup`, can only contain `CommandButtons`, which allows you to group buttons into inseparable units.

Whether it's a good or bad idea to combine `CommandButtons` into `CommandGroups` is a matter of opinion. I can say that when `CommandButtons` are combined into `CommandGroups`, the buttons should be fully defined before they are added to the group. This is the issue of "early vs. late" composition. The later the composition (that is, building the whole from its parts), the better the design.

Consider a set of four buttons that navigate in a table: forward, backward, to the first record and to the last record. You could design this `CommandGroup` in two different ways. You could create a `CommandGroup` class, add four buttons to it, and write the necessary code in each button's click event.

The second choice would be to define a command button class named `cmdNavigate`. Then you define each of the navigation command buttons as a subclass of `cmdNavigate` with all of the code that provides each button with its behavior. You would then have four command button classes, one for each button. You could then define a container class and place one of each of the command button classes into the container.

Which way is better? The latter method is preferred. But why? Because the latter method, defining each button as part of a class of buttons and then combining them into the group, provides the most flexibility in evolving the construct in the future. You can change the behavior of any one button by altering its class code, but you can also affect the entire group by altering the `cmdNavigate` class. You don't need to find the `CommandGroup` that contains those buttons to make the changes.

On an even more important level, what if your design required you to have a set of all four navigation buttons in some forms, but only two buttons (next and previous) in other forms? Then you would have two `CommandGroups`: one with four buttons and one with two buttons. If you followed the first methodology of constructing the `CommandGroups`, there would be no `cmdNext` class because you defined the button inside its container. If you wanted to improve the functionality of `cmdNext`, you'd have to edit two class definitions: the `CommandGroup` with four buttons and the one with two buttons. If you followed the concept of late composition, you'd have a class for `cmdNext` that was used in both `CommandGroups`, and therefore you'd have only one place to alter your code.

This discussion of `CommandGroups` and `CommandButtons` is equally true for all composition situations. Build the pieces completely before you try to assemble the whole.

## Timer

The timer is at once the simplest and most complex control in Visual FoxPro. It's simple in its elegant design and ease of use. It's complex in the types of problems it can solve.

The timer has three properties and methods of concern. The Enabled property turns the timer on and off, the Interval property determines how often the timer will “fire,” and the Timer method holds the code that runs when the timer “fires.”

The Enabled property can be set to .T. to turn the timer on, and .F. to turn it off. The Interval property sets the time between firings of the timer event when the timer is turned on. The Interval is set in milliseconds, so a one-second interval has a value of 1000.



The interval of a timer is set in milliseconds. It's easy to bring an application to its knees by setting the interval thinking in seconds rather than milliseconds.

Finally, the code written in the timer event will be executed every time the interval expires. One caution: Be sure that the code runs in less time than the interval for the timer.

It is also important to note that the timer event will *not* fire if other Visual FoxPro code is running. In this case the timer will wait until the other code finishes and then it will run its timer event code.

So what can a timer be used for? The answer to that question is in your own imagination. You could use a timer to check for a condition after each interval, or to cause a delayed action to occur. The possibilities are limited only by your own imagination.

## PageFrames and Pages

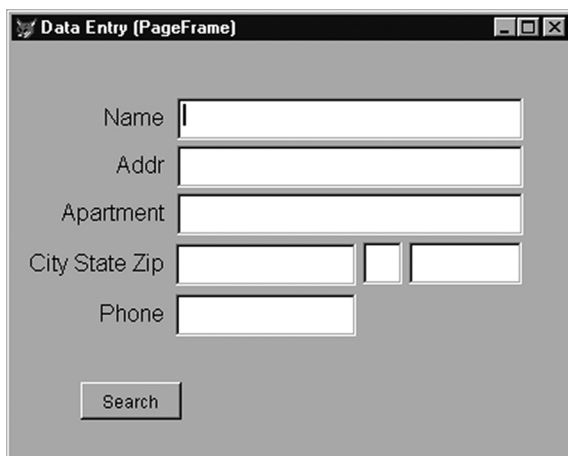
A tabbed dialog is a control with tabs across the top. When clicked, each tab presents a different set of controls. To produce a tabbed dialog in Visual FoxPro, use a PageFrame, which is a special container that can contain only Pages. Pages are containers that can contain any object except a form or formset. Pages can actually contain other PageFrames, although the interface design might become confusing to the user.

One of the properties available on the PageFrame is the Tabs property. When set to .T. (the default) each page has a tab at the top of the PageFrame that can be clicked to select that page. When the Tabs property is set to .F., there are no tabs at the top and the only way to change pages is programmatically. Imagining a PageFrame without tabs may seem silly at first, but consider a simple data-entry form. You might want to provide a searching capability for this form where the user can specify a number of different values to assist in the search.

Your first thought might be to bring up another form when the user clicks the Search button. But what if your PageFrame in the data-entry form had all data-entry controls on page 1 and the searching controls on page 2? If that PageFrame had no tabs

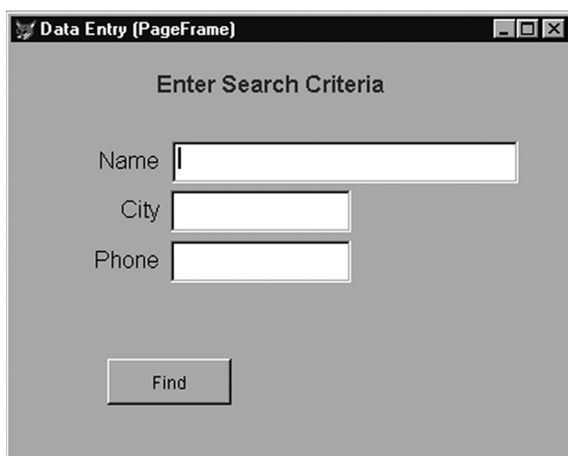
and it filled the form, the user would have no idea the PageFrame was even there.

When the user clicked the Search button, you'd simply set the ActivePage property of the PageFrame to 2, which would display the search controls. When the user finished the search, your code would reset the ActivePage of the PageFrame to 1, again showing the edit controls. This provides search functionality without throwing another form in the user's face. **Figure 9.13** shows such a form.



**Figure 9.13.** An edit form with a PageFrame that has no tabs.

**Figure 9.14** shows the same form after the user has clicked the Search button.



**Figure 9.14.** The same form as Figure 13 after the user has clicked the Search button.

The form in Figures 9.13 and 9.14 is included on the CD in the sample project and is named *PgfSrch.scx*.

Another use for PageFrames is to simplify the design of a data-entry form. Often, the amount of data required to describe a certain entity can easily make any data-entry form appear crowded and cluttered. A PageFrame can reduce the clutter and confusion. *PageFram.scx*, also included in the sample project, demonstrates the use of a PageFrame to accomplish this goal of simplifying the interface for the user. **Figure 9.15** shows the first page of the form.

**Figure 9.15.** The demographic page of a customer edit form.

**Figure 9.16** shows the financial page of the same form.

**Figure 9.16.** The financial page of the customer edit form in Figure 15.



Without viewing every one of the five pages here, you can imagine that if a PageFrame were not used, the amount of information required for one entity would crowd and clutter a single form. The user would have to work harder to find what he was looking for. With the PageFrame tool you can separate the information into groups of related data and thus make the user's job easier.

## What about speed?

From the previous example, you can readily see that by using PageFrames a form could become quite complex with a large number of controls in it. This could have an impact on the amount of time it takes for that form to be created in memory. Is there a way to have the best of both worlds, that is, the quick response of a simple, single-page form and the simplicity of the interface of a multi-page form? Yes, and it's called *delayed instantiation*. With delayed instantiation you only put the controls on the page of the form that the user can see. The other pages have a single control on them called a proxy, a stand-in for the real controls. The real controls for the page are defined in a container-class definition and are instantiated when the user brings that page to the front.

Try using *PageFram.scx* to create a delayed-instantiation situation. **Figure 9.17** shows page 4, the financial page, of the form.

The screenshot shows a software window titled "Form Designer - pagefram.scx". Inside, there's a form titled "Customer Information" with five tabs: "Demographics", "Locations", "Contacts", "Financial" (which is selected), and "Notes".

Under the "Financial" tab, there are three text input fields: "Balance" followed by a text box labeled "Text1", "Last Payment" followed by a text box labeled "Text2", and "Date" followed by a text box labeled "Text3".

Below these fields is a section titled "Invoices" containing a table with four columns: "Invoice", "Date", "Amount", and "Balance Due". The first row of the table contains the text "abl" in each of the four columns. There are three empty rows below it.

Below the "Invoices" table is a section titled "Payments" containing a table with three columns: "Check #", "Date", and "Amount". The first row of the table contains the text "abl" in each of the three columns. There are three empty rows below it.

**Figure 9.17.** The financial page of the customer edit form.

In order to use delayed instantiation, first save the controls in question as a class, and then instantiate that class at runtime. You can select all controls on the page (use Shift-Click to select each control), then choose *Save As Class* from the *File* menu as shown in **Figure 9.18**.

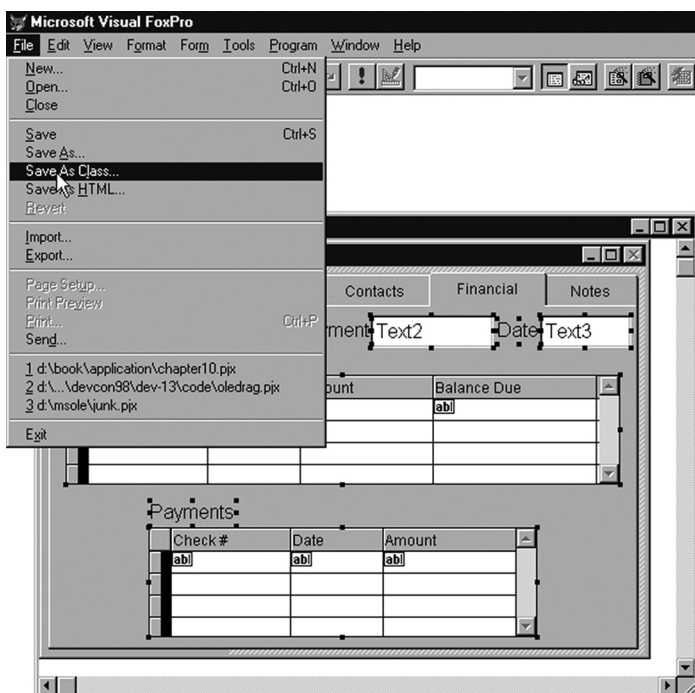


Figure 9.18. The *Save As Class* option of the *File* menu.

Once you click *Save As Class*, you can fill in the dialog as shown in Figure 9.19.

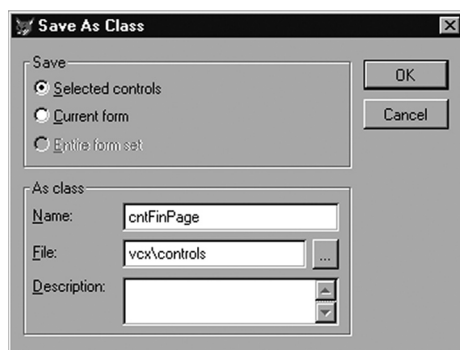
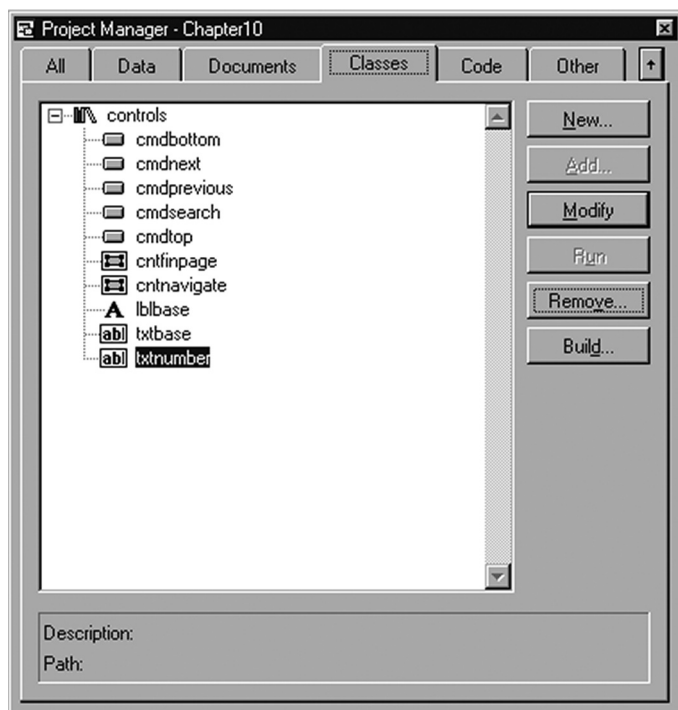


Figure 9.19. The *Save As Class* dialog completed for your class.

The controls you selected will appear in a container class, so you can delete those controls from the page in the form designer.

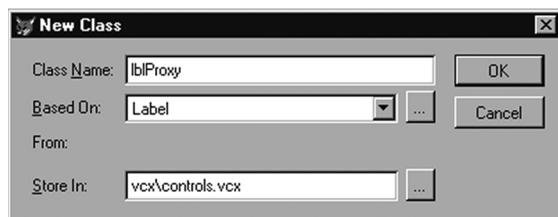
Next, in the Project Manager, select the *Classes* tab as shown in Figure 9.20.



**Figure 9.20.** The *Classes* tab of the Project Manager.

Notice that your `cntFinPage` class is already in the `Controls.vcx` class library. Edit the `cntFinPage` class and make one minor change. Change the `BorderWidth` property to 0 so the container has no border. Highlight the `cntFinPage` class and then click the `Modify` button. This will open that class in the Visual Class Designer. Right-click the class and choose *Properties*, find the `BorderWidth` property and change its value to 0. Then click the `Close` button in the class designer and click `Yes` to save the changes.

Finally, create the proxy class. Highlight any class in the controls library and then click the `New` button. **Figure 9.21** shows the resulting *New Class* dialog, filled in the way you want it to be.



**Figure 9.21.** The *New Class* dialog filled in for your proxy class.

**Figure 9.22** shows the lblProxy class after a few properties have been changed. Set the BackColor to red, the ForeColor to yellow, the Caption to “Proxy for actual controls,” and the Visible property to .F.



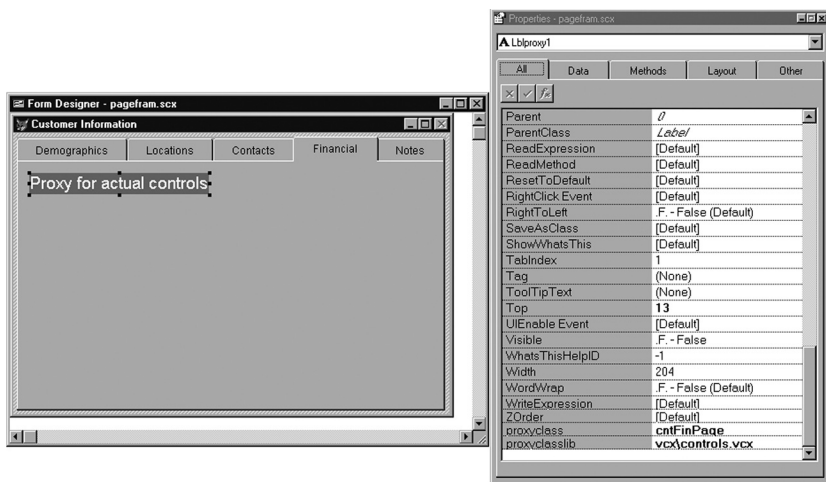
**Figure 9.22.** The lblProxy class with your property settings.

From the *Class* menu select *New Property* and add two properties named ProxyClassLib and ProxyClass. You’ll notice that these properties are added to the end of the property list in the property sheet. Go to each of these properties and give them each a value of a single space to set their data type to *character*. Now add some code to the UIEnable event of your class. The UIEnable event is fired for all objects in a page whenever that page is activated or deactivated. Because you want your proxy object to do its thing when the page is activated, the UIEnable event is the correct one to choose. Put this code in the UIEnable event:

```
LOCAL lcVisible
IF TYPE("THIS.Parent."+THIS.ProxyClass+".Name") <> "C"
    SET CLASSLIB TO (THIS.ProxyClassLib) ADDITIVE
    THIS.Parent.AddObject(THIS.ProxyClass,THIS.ProxyClass)
    lcVisible = "THIS.Parent." + THIS.ProxyClass + ".Visible"
    &lcVisible = .T.
ENDIF
```

This code checks to see if the container is there already. If it is, it does nothing. If the container isn’t there, the code opens the class library and then adds that container object to the page.

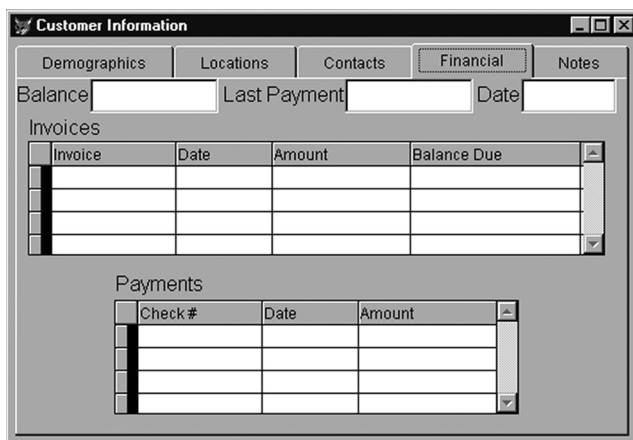
Now return to your PageFram.scx form, open the financial page, and put an lblProxy object on that page. **Figure 9.23** shows this form designer for your PageFrame form.



**Figure 9.23.** The lblProxy class added to the financial page of your form.

Notice in the properties sheet that you've set the ProxyClass property to cntFinPage, the name of your controls container for this page and the ProxyClassLib property to VCX\Controls.vcx, the class library that holds your container class. The *label* base class was used so you could give it a caption that is visible in the form designer, reminding you of its purpose.

At runtime the label can't be seen because its Visible property is set to .F., but once that page is brought to the front the controls are present as shown in **Figure 9.24**.



**Figure 9.24.** The PageFrame form at runtime after the lblProxy class has added the container to the page.

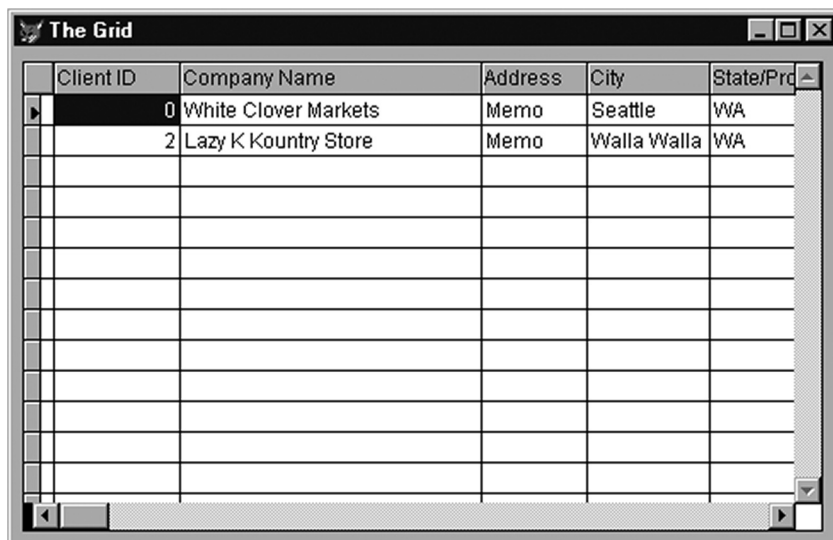
By using this proxy technique for every page in the PageFrame, except for page 1, you can greatly reduce the amount of time it takes the form to initially be created by reducing the number of controls that need to be instantiated.

The actual gain in performance achieved depends on how many pages are involved and how many controls are on each of those pages. The downside of this proxy approach is that you must edit the controls in the class designer instead of in the form designer.

## Grid

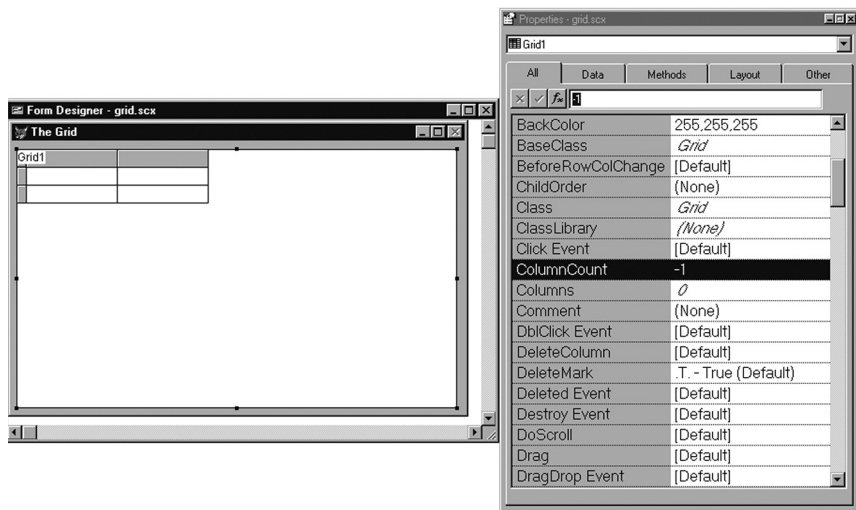
The next control is the one most often requested by FoxPro 2.x developers. The Grid is not unlike the FoxPro 2.x Browse control because it displays multiple records in a row-and-column orientation, and allows the user to navigate in both row (record) and column (field) directions. The major weakness of Browse in earlier versions of FoxPro was that it wouldn't coordinate well with a data-entry screen or form.

The Grid control in Visual FoxPro is, in fact, a control that can be placed inside a form, and it acts just like any other control in the form. **Figure 9.25** shows a Visual FoxPro form with a Grid as the only control.



**Figure 9.25.** A form with a simple Grid control showing the Clients table in the Time and Billing database.

Figure 9.26 shows this form in the form designer.

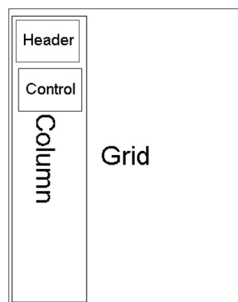


**Figure 9.26.** The Grid form in the form designer.

Notice that the ColumnCount property for the grid is set to -1. This is a special ColumnCount that indicates that the Grid should have one column for each field in its record source. The actual count of columns will be determined at runtime.

This format for a Grid can be used to get the simplest Grid into a form. Of course, you can set the ColumnCount property to any number you choose and fully control the number of columns the Grid will have. Let's step back a moment before you dig into this, and take a more general look at the Grid itself and its construction.

The Grid is a container that can contain only one object class: a column. A column, also a container, can contain one header object and one or more controls. The default control in a column is a TextBox. **Figure 9.27** shows this containership relationship.

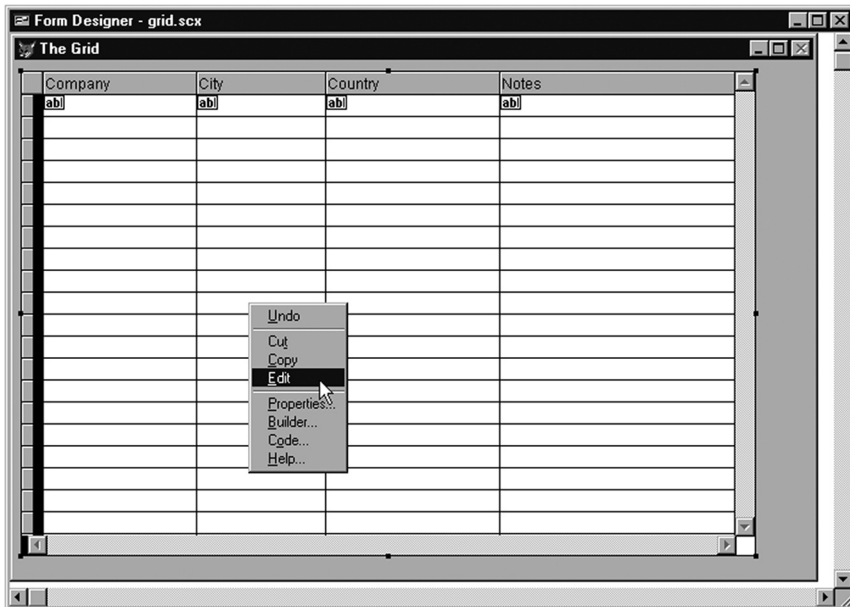


**Figure 9.27.** The Visual FoxPro Grid containership structure.

In Figure 9.27 the outer box is the Grid, which can contain only columns. The columns in turn can contain one header and one or more controls.

As stated earlier, the default control for a column is a TextBox. However, you can actually put any control you like into the column of a Grid. Right-clicking the Grid and choosing the *Edit* option will allow you to add a different control to a column. When you choose *Edit* from the right-click menu, the Grid will get a colored box around it, indicating that you are now editing the Grid and its contents.

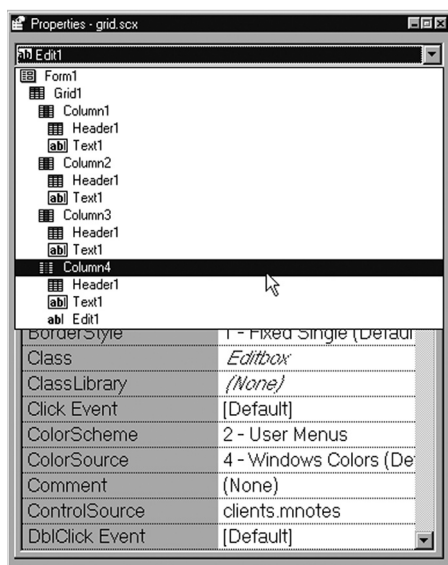
Once the Grid is in edit mode, you can choose a control from the form controls toolbar and simply drop it on one of the columns in the Grid. Note that this can only be done on Grids with a ColumnCount other than -1. **Figure 9.28** shows your Grid form after you have changed the ColumnCount to 4 and you are about to select *Edit*.



**Figure 9.28.** The shortcut menu for your Grid, ready to select the Edit option.

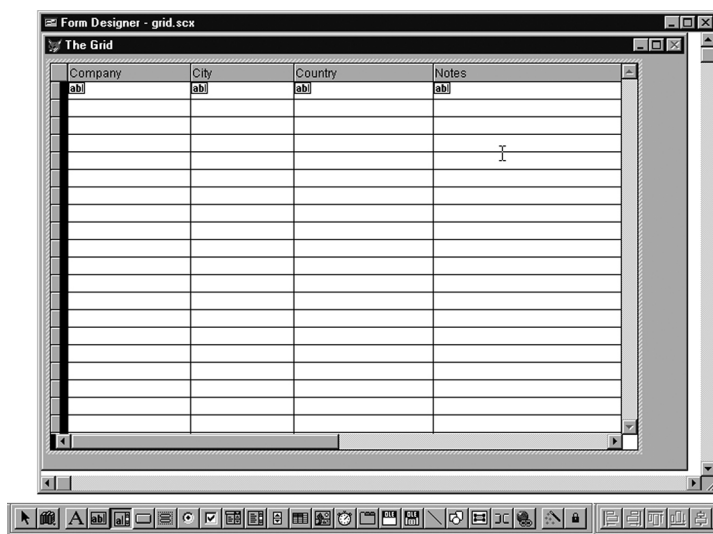
Next, click the *EditText* control in the form controls toolbar and drop that control over the fourth column of the Grid. See **Figure 9.29**.





**Figure 9.29.** Dropping the EditBox control in column 4 of the Grid.

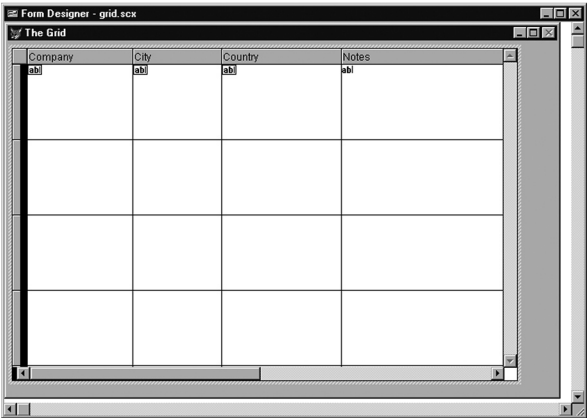
To confirm that the EditBox has actually been added to column 4, you can use the ComboBox at the top of the property sheet as shown in **Figure 9.30**.



**Figure 9.30.** The Grid form in the form designer.

The listing under column 4 contains a Header, a TextBox, and an EditText. Consider that any column can contain more than one control, but at any time only one of those controls is the current control, the one that the user will see. You can change the CurrentControl property for column 4 to be the Edit1 EditText.

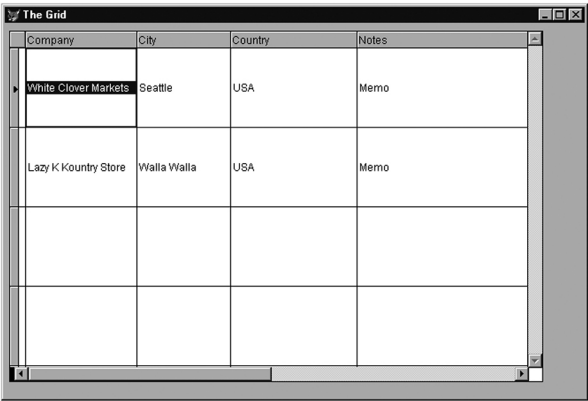
While you're editing the Grid, you can also increase its row height by dragging the line between rows. Make your Grid look like the one in **Figure 9.31**.



**Figure 9.31.** The Grid form with a larger row height.

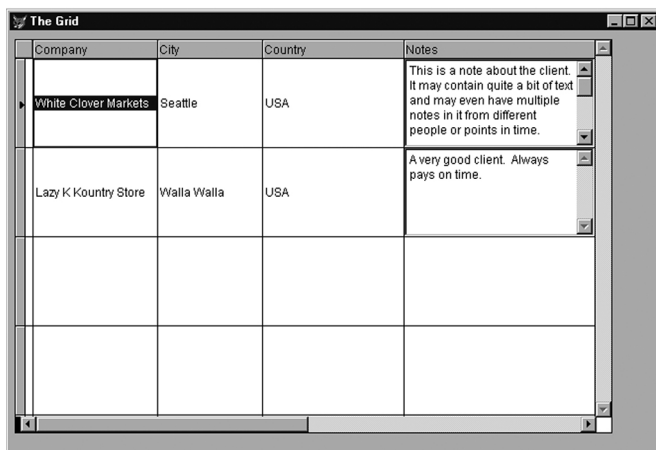
You can also change the RowHeight of the Grid by typing a new height in the property sheet for the RowHeight property.

Now run the form and see what it looks like. **Figure 9.32** shows the Grid form running.



**Figure 9.32.** The Grid form as it's running.

Notice that column 4 contains the word “Memo” rather than the contents of the memo field. This is because of one column property that is very important to understand: Sparse. This property tells the column whether to show the control in the column, or just the data, when a cell doesn’t have focus. By default the Sparse property is .T., which only shows the data and does not draw the CurrentControl in all cells of the column except the cell that has focus. That is why you see the word “Memo,” the data when an EditBox is not used for a memo field. If you simply change the Sparse property of column 4 to .F. and then run the form, you’ll see something similar to **Figure 9.33**.



**Figure 9.33.** Column 4 with the Sparse property set to .F.

I’ll discuss more about the Grid control in the next chapter, “Advanced Form Controls.”

## Container

The Container control is simply a holder for other controls. In the PageFrame section of this chapter, we presented the idea of delayed instantiation, you saved all controls on a page into a class library. Visual FoxPro automatically placed these controls inside a container for you.

You can use containers in your class designs as well. Whenever you need more than one control to perform a function, there is a possibility that a container might make the job easier.

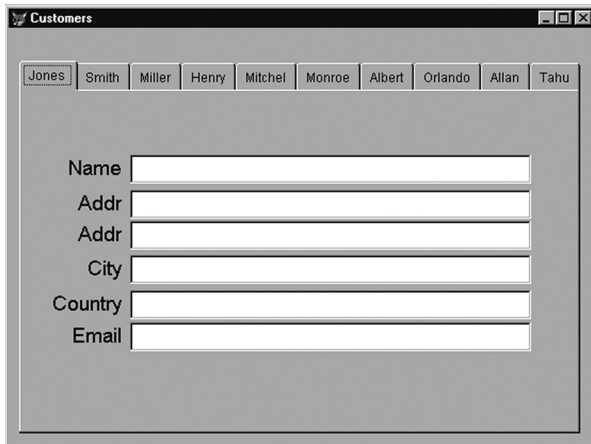
You’ll still have to deal with the issue of “early vs. late” composition when using containers. It’s always a good idea to fully define each control as a class of its own before adding it to the container. The next, “Advanced Form Controls,” will discuss the use of containers in more depth.

## Misusing and abusing the controls

Ah, the misuse and abuse of controls. You see it everywhere. One issue that comes up quite often is the question of how long it takes a ListBox to load a list of 20,000 items. Think about it. Would you want to scroll through a ListBox of 20,000 items looking for the one you wanted? ListBox and ComboBox are powerful controls for presenting a relatively small number of choices to the user. When the number of options grows too large, then it's time to find another way to present the choices.

How about the form with nothing in it but a Grid that has ComboBoxes in some columns, EditBoxes in others, and even a Grid in one of the columns? Wouldn't this interface be better presented to the user by diverging and using more than one control?

Take a look at **Figure 9.34**. Can you see anything wrong with this interface design?



**Figure 9.34.** A PageFrame gone wild.

This is obviously a poor use of a PageFrame. What do you do when you have more customers than you can fit into pages?

Another abuse of PageFrames is using one when the interface should actually use multiple forms. Another is where the PageFrame is unnecessary (all of the controls could fit in one form without a PageFrame).

An OptionGroup should never have more than four or five options in it. More choices than that should use a ComboBox or ListBox control. Using an OptionGroup for more than four or five choices is another example of abuse of a control.

I can go on and on listing the abuses of various controls; the point is that each control has its strengths and weaknesses. The design of an interface should take advantage of the strengths of a control and avoid its weaknesses.

As a developer, you need to understand the purpose for the form and then build its contents to meet that purpose. You need to know the strengths and weaknesses for each of the controls available and choose the control that best meets the requirements at hand.

The misuse of containers isn't uncommon. You should use a container only when it adds to the functionality or reusability of the controls it contains. Do not use containers simply to "Lasso" a bunch of controls together.

## Summary

The control is the object closest to the user. Although the user sees and interacts with forms, the controls provide a message path between the user and the application. The selection of controls is not always a simple process. Many times you will need to change your initial choices to better meet the user's needs.

Visual FoxPro not only provides a variety of base controls for you to choose from, but it provides you the ability to create your own controls. Using the class designer, you can modify the behavior of the Visual FoxPro base classes and create new controls that are specialized for whatever purpose you have.

You can combine multiple controls, using containers, into more complex interface objects that meet the needs of the user. These containers can also be used to reduce your own development time. By using containers to combine the controls that are "always" found together, you reduce the time it takes to build the forms. For example, building an address block container as a class would allow you to place that address block in the customer form, the employee form, the vendor form, etc.

Choosing the right control is critically important to building a good user interface. While it is possible to misuse any control discussed in this chapter, each one deserves a place in your toolbox.

