Chapter 5 Combos and Lists

"Why can't somebody give us a list of things that everybody thinks and nobody says, and another list of things that everybody says and nobody thinks."

("The Professor at the Breakfast-Table" by Oliver Wendell Holmes, Sr.)

Combos and lists are two very powerful controls that allow the user to select from a predetermined set of values. Used properly, they provide a valuable means of ensuring data validity. Used improperly, they can be your worst nightmare. If you use a combo box to present the user with thousands of items, you are asking for trouble! In this chapter, we present some handy combo and lists classes that can be used to provide a polished, professional interface while significantly reducing your development time. All the classes presented in this chapter can be found in the CH05 class library.

Combo and list box basics

One look at the properties and methods of combo and list boxes, and you can see they function internally in much the same way. Although a drop-down combo allows you to add items to its *RowSource*, you can't do this with a drop-down or scrolling list – or rather not with native base class controls. With no less than ten possible *RowSourceTypes* and two different ways of handling their internal lists (*ListItemID* and *ListIndex*), these classes provide the developer with almost too much flexibility. Of the ten *RowSourceTypes, 0-None, 1-Value, 2-Alias, 3-SQL Statement, 5-Array and 6-Fields* are the most useful. This chapter contains examples using these six *RowSourceTypes*.

The remaining four, *4-Query (.QPR), 7-Files, 8-Structure* and *9-Popup* are not covered because they are either very specific in their nature (*7-Files* and *8-Structure*) or are included to provide backward compatibility (*4-Query* and *9-Popup*) and do not fit in the context of a Visual FoxPro application.

List and ListItem collections

These two collections allow you to access the items in the control's internal list without having to know anything about its specific *RowSource* or *RowSourceType*. Because of this, these collections and their associated properties and methods can be used to write some very generic code. The *List* collection references the items contained in the list in the same order in which they are displayed. The *ListItem* collection references these same items by their *ItemID's*. The *ItemID* is a unique number, analogous to a primary key that is assigned to items when they are added to the list. Initially, the *Index* and the *ItemID* of a specific item in the list are identical. But as items are sorted, removed and added, these numbers are not necessarily the same anymore.

Table 5.1 *Properties and methods associated with the List collection*

Property or method	**What does it do?**
List	Contains a character string used to access the items in the list by index. Not available at design time. Read only at run time.
ListIndex	Contains the index of the selected item in the list or 0 if nothing is selected
NewIndex	Contains the index of the item most recently added to the list. It is very useful when adding items to a sorted list. Not available at design time. Read only at run time.
TopIndex	Contains the index of the item that appears at the top of the list. Not available at design time. Read only at run time.
AddItem	Adds an item to a list with RowSourceType 0-none or 1-value
IndexToItemID	Returns the ItemID for an item in the list when you know its index
RemoveItem	Removes an item from a list with RowSourceType 0-none or 1-value

Table 5.2 *Properties and methods associated with the ListItem collection*

Property or method	**What does it do?**
ListItem	Contains a character string used to access the items in the list by ItemID. Not available at design time. Read only at run time.
ListItemID	Contains the ItemID of the selected item in the list or -1 if nothing is selected
NewItemID	Contains the ItemID of the item most recently added to the list. It is very useful when adding items to a sorted list. Not available at design time. Read only at run time.
TopItemID	Contains the ItemID of the item that appears at the top of the list. Not available at design time. Read only at run time.
AddListItem	Adds an item to a list with RowSourceType 0-none or 1-value
IItemIDToIndex	Returns the Index for an item in the list when you know its itemID
RemoveListItem	Removes an item from a list with RowSourceType 0-none or 1-value

Gotcha! AddItem

The online help states that the syntax for this command is **`Control.AddItem(cItem [, nIndex] [, nColumn])`**. It goes on to say that when you specify the optional nIndex and nColumn parameters, the new item is added to that row and column in the control. If you specify a row that already exists, the new item is inserted at that row and the remaining items are moved down a row. Sounds good! Unfortunately, it doesn't work quite like that.

The A*ddItem* method really adds an entire row to the list. If the list has multiple columns and you use this syntax to add items to each column, the result is not what you would expect. When using the *AddItem* method to populate a combo or list, add the new item to the first column of each row using the syntax **`Control.AddItem( 'MyNewValue' ).`** Assign values to the remaining columns in that row using the syntax **`Control.List[Control.NewIndex, nColumn] = 'MyOtherNewValue'.`** The *AddListItem* method, however, does work as advertised. This gotcha! is clearly illustrated in the form ListAndListItem, included with the sample code for this chapter.

When do the events fire?

The answer to that, as usual, is "it depends." The events fire a little differently depending on the style of the combo box. The order in which they fire also depends on whether the user is navigating and selecting items with the mouse or with the keyboard. It is a gross understatement to say understanding the event model is important when one is programming in an object-oriented environment. This is absolutely critical when creating reusable classes, especially complex classes like combo and list boxes.

As expected, the first events that fire when the control gets focus are *When* and *GotFocus*. This is true for combo boxes of both styles as well as list boxes. And these events occur in this order whether you tab into the control or click on it with the mouse. Once small peculiarity about the drop-down combo is that the next event to fire is *Combo.Text1.GotFocus*! Text1 is not accessible to the developer. Visual FoxPro responds to any attempt to access it in code with "Unknown member: TEXT1." We assume that text1 is a protected member of Visual FoxPro's base class combo when it's style is set to 0 – DropDown Combo.

The following list contains the events that you will most often be concerned with when dealing with combo and list boxes. It is by no means a comprehensive list, but all the significant events are there. For example, the MouseDown and MouseUp events fire before the object's Click event. For the sake of simplicity and clarity, the mouse events are omitted.

Table 5.3 *Combo and list box event sequence*

Action	DropDown Combo	DropDown List	ListBox
Scroll through list using the down arrow (without dropping the combo's list first)	Not applicable	KeyPress(24, 0) InteractiveChange Click Valid When	KeyPress(24, 0) InteractiveChange Click When
Use the mouse to drop the list	DropDown	DropDown	Not Applicable
Use ALT+DNARROW to drop the list	KeyPress(160, 4) DropDown	KeyPress(160, 4) DropDown	Not Applicable
Scroll through dropped-down list using the down arrow	KeyPress(24, 0) InteractiveChange	KeyPress(24, 0) InteractiveChange	KeyPress(24, 0) InteractiveChange Click When
Select an item in the list using the mouse	InteractiveChange Click Valid When	InteractiveChange Click Valid When	InteractiveChange Click When
Select an item in the list by pressing the <ENTER> key	KeyPress(13, 0) Click Valid	KeyPress(13, 0) Click Valid	KeyPress(13, 0) DblClick Valid
Exit the control by clicking elsewhere with the mouse	Valid LostFocus Text1.LostFocus	LostFocus	LostFocus
Exit the control using the <TAB> key	KeyPress(9, 0) Valid LostFocus Text1.LostFocus	KeyPress(9, 0) LostFocus	KeyPress(9, 0) LostFocus

It is interesting to note that the *Valid* event does **not** fire when the control loses focus for either the DropDown List or the ListBox. One consequence of this behavior is that any code called from the *Valid* method will not get executed when the user merely tabs through a DropDown List or a ListBox. In our opinion, this is a good thing. It means that we can place code that updates underlying data sources in methods called from the *Valid* method of these controls and not worry about dirtying the buffers if the user hasn't changed anything.

It is also worth noting that for ComboBoxes, the *Valid* event fires whenever the user selects an item from the list. (However, if the user selects an item in the list by clicking on it with the mouse, the *When* event also fires.) Conversely for ListBoxes, the *Valid* event only fires when the user selects an item by pressing the **`ENTER`** key or by double clicking on it. It is interesting to note that selecting an item with the **`ENTER`** key also fires the *dblClick* event.

Another anomaly worth noting is that the *Click* event of the ListBox fires inconsistently depending on which key is used to navigate the list! When the user presses the **`up arrow`** and **`down arrow`** keys, the *Click* event fires. When the **`page up`** and **`page down`** keys are used, it doesn't.

How do I bind my combo and list boxes?

Obviously, you bind your combo and list boxes by setting the *ControlSource* property to the name of a field in a table, cursor, or view or to a form property. One gotcha! to be aware of is that you can only bind these controls to character, numeric, or null data sources. If you try to bind a combo or list box to a Date or DateTime field, Visual FoxPro will complain and display the following error at run time:

Error with <ComboName>-Value: Data Type Mismatch.
Unbinding object <ComboName>

If you must bind a combo or list box to a Date or DateTime field, you will have to resort to a little trickery. In this case you cannot use *RowSourceTypes* of 2-Alias or 6-Fields. You can, for example, set the *RowSourceType* to 3-SQL Statement and use a SQL statement similar to this as the *RowSource:*

```
SELECT DTOC( DateField ) AS DisplayDate, yada, nada, blah FROM MyTable ;
       ORDER BY MyTable.DateField INTO CURSOR MyCursor
```

Leave the *ControlSource* blank and add this code to its *Valid* method to update the date field in the underlying table:

```
REPLACE DateField WITH CTOD( This.Value ) IN MyTable
```

You will also need to write some code to manually update the control's *Value* from its *ControlSource* to mimic the behavior of a bound control when you *Refresh* it. This code in the combo or list box's *Refresh* method does the trick:

```
This.Value = DTOC( MyTable.DateField )
```

Another gotcha! that may bite you occurs when the *ControlSource* of your combo box refers to a numeric value that contains negative numbers. This appears to be a problem that only occurs when the *RowSourceType* of the control is 3-SQL Statement and may actually be a bug in Visual FoxPro. The result is that nothing at all is displayed in the text portion of the control for any negative values. However, even though the control's *DisplayValue* is blank, its *Value* is correct. The easy workaround is to use a *RowSourceType*, other than 3-SQL Statement, when the control is bound to a data source that can contain negative values. It's not as if there are a lack of alternatives.

So what are BoundTo and BoundColumn used for?

These properties determine how the control gets its value. The value of a combo or list box is taken from the column of its internal list that is specified by its *BoundColumn.* A combo box's *DisplayValue*, the value that is displayed in the text portion of the control, **always comes from column one!** Its value, on the other hand, can be taken from any column of its internal list. This means you can display meaningful text, such as the description from a lookup table, at the same time the control gets its value from the associated key. You do not even have to display this associated key in the list to have access to it.

For example, suppose the user must assign a particular contact type to each contact when it is entered. The appropriate contact type can be selected from a DropDown List with its *RowSourceType* set to 6-Fields, its *RowSource* set to "**`ContactType.CT_Type, CT_Key`**" where **`CT_Type`** is the description and **`CT_Key`** is its associated key in the *ContactType* table. To set up the control, first set the DropDown List's *ColumnCount* to 2 and its *ColumnWidths* to 150, 0. Then set the *BoundColumn* to 2 to update the bound field in the *Contacts* table from the key value instead of the description.

The setting of *BoundTo* specifies whether the value property of a combo or list box is determined by its *List* or its *ListIndex* property. The setting of *BoundTo* only matters when the control is bound to a numeric data source. If the *ControlSource* of the combo or list box refers to numeric data, setting the control's *BoundTo* property to true tells Visual FoxPro to update the *ControlSource* using the data from the bound column of the control's internal list. Leaving *BoundTo* set to false here causes the *ControlSource* to be updated with the control's *ListIndex*, that is, the row number of the currently selected item.

The easiest way to illustrate how this works is by using a little code that simulates how the setting of the *BoundTo* property affects the way in which the control's *Value* property is updated:

```
WITH This
  IF .BoundTo
    .Value = VAL( .List[.ListIndex, .BoundColumn] )
  ELSE
    .Value = .ListIndex
  ENDIF
ENDIF
```

How do I refer to the items in my combo and list boxes?

As discussed earlier, you can use either the *List* or *ListItem* collection to refer to the items in your combo or list box. The big advantage to using these collections to access the items in the control's *RowSource* is that it is not necessary to know anything about that *RowSource*. You can use either *ListIndex* or *ListItemID* property to refer to the currently selected row. If nothing in the list is selected, the control's *ListIndex* property is 0 and its *ListItemID* property is –1. So in the *Valid* method of the combo box or the *LostFocus* method of the list box, you can check to see if the user selected something like this:

```
WITH This
  IF .ListIndex = 0   && You can also use .ListItemID = -1 here
    MESSAGEBOX( 'You must select an item from the list', 16, ;
                'Please make a selection' )
    IF LOWER( .BaseClass ) = 'combobox'
      RETURN 0        && If this code is in the valid of a combo box
    ELSE
      NODEFAULT       && If this code is in the LostFocus of a ListBox
    ENDIF
  ENDIF
ENDWITH
```

There are also several ways to refer to the value of a combo or list box. The simplest of them all is **`Control.Value`**. When nothing is selected, the control's value is empty. This is something to be considered if the *RowSource* for the control permits empty values. It means you cannot determine whether the user has made a selection merely by checking for an empty value property.

Because the *List* and *ListItem* collections are arrays, you can address them as you would any other array. (However, although you can address these collections as arrays, you cannot actually manipulate them directly using the native Visual FoxPro array functions such as **`ASCAN()`**, **`ADEL()`** or **`ALEN()`**.)

To access the selected item in the control's internal list when its *ListIndex* is greater than zero you can use:

```
Control.List[Control.ListIndex, Control.BoundColumn]
```

while this does exactly the same thing when its *ListItemID* is not –1:

```
Control.ListItem[Control.ListItemID, Control.BoundColumn]
```

You can also access the items in the other columns of the selected row of the control by referring to Control.List[Control.ListIndex, 1], Control.List[Control.ListIndex, 2], and so on all the way up to and including Control.List[Control.ListIndex, Control.ColumnCount].

Remember that, when using the control's *List* and *ListItem* collections in this manner, all the items in the control's internal list are stored as character strings. If the control's *RowSource* contains numeric, date, or datetime values, these items will **always** be represented internally as character strings. This means that if you want to perform some "behind the scenes" updates using the items in the currently selected row of the list, you will need to convert these items to the appropriate data type first. Otherwise, Visual FoxPro will complain, giving you a data type mismatch error.

List boxes with *MultiSelect* set to true behave a little bit differently. Its *ListIndex* and *ListItemID* properties point to the last row in the control that was selected. To do something with all of the control's selected items, it is necessary to loop through its internal list and check the *Selected* property of each item like so:

```
WITH Thisform.LstMultiSelect
  FOR lnCnt = 1 TO .ListCount
    IF .Selected[lnCnt]
      *** The item is selected, take the appropriate action
    ELSE
      *** It isn't selected, do something else if necessary
    ENDIF
  ENDFOR
ENDWITH
```

What is the difference between DisplayValue and Value?

Combo boxes are particularly powerful controls because they enable you to display descriptive text from a lookup table while binding the control to its associated key value. This is possible only because the combo box has these two properties. Understanding the role each of them plays can be confusing, to say the least.

DisplayValue is the descriptive text that is displayed in the textbox portion of the control. This is what you see when the combo box is "closed". The combo's *DisplayValue* always comes from the first column of its *RowSource.* On the other hand, the combo's *Value* comes from whichever column is specified as its *BoundColumn*. If the *BoundColumn* of the combo box is column one, its *Value* and *DisplayValue* are the same when the user picks an item from the list. When the control's *BoundColumn* is not column one, these two properties are not the same. See **Table 5.4** below for the differences between these two properties in different situations.

Table 5.4 *Combo and list box event sequence*

Bound Column	Action	DisplayValue	Value
1	Select an item in the list	Column 1 of selected row	Column 1 of selected row
1	Type item not in list	Typed text	Empty
N # 1	Select an item in the list	Column 1 of selected row	Column n of selected row
N # 1	Type item not in list	Typed text	Empty

What’s the difference between RowSourceTypes "alias" and "fields"?

The basic difference is that *RowSourceType* "2-Alias" allows the *RowSource* property to contain just an Alias name. The control fills the number of columns it has available (defined by the *ColumnCount* property) by reading the data from the fields in the specified alias in the order in which they are defined. You may, however, specify a list of fields that are to be used even when the *RowSourceType* is set to "2-Alias." In this case there is no practical difference between the *Fields* and the *Alias* settings.

When using RowSourceType "6-Fields" the RowSource property **must** be filled in using the following format:

```
<Alias Name>.<first field>,<second field>,…….<last field>
```

When using either *RowSourceType* "2-Alias" or "6-Fields," you can still access **any** of the fields in the underlying data source - even if they are not specifically included in the *RowSource*. It is also worth remembering that whenever a selection is made, the record pointer in the underlying data source is automatically moved to the appropriate record. However if no valid selection is made, the record pointer is left at the last record in the data source - not, as you might expect, at **`EOF()`**.

By the way, when using RowSourceType of "3-SQL", always select INTO a destination cursor when specifying the RowSource. Failing to specify a target cursor will result in a browse window being displayed! The behavior of the cursor is the same as when using a table or view directly - all fields in the cursor are available, and selecting an item in the list moves the record pointer in the cursor.

How do I make my combo and list boxes point to a particular item?

When these are bound controls, they automatically display the selection specified by their *ControlSources* so you don't need to do anything at all. But what if the control isn't bound or you want to display some default value when adding a new record? As usual, there is more than one way to skin a fox. Perhaps the easiest way to accomplish this is:

```
Thisform.MyList.ListIndex = 1
```

This statement, either in the form's *Init* method or immediately after adding a new record, selects the first item in the list. You can also initialize a combo or list box by directly setting its value. However, when initializing combos and lists that are bound to buffered data, the act of doing so will dirty the buffers. This means that if you have a routine that checks for changes to the current record using **`GETFLDSTATE(),`** the function will detect the "change" in the current record even though the user hasn't touched a single key. To avoid undesirable side effects, such as the user being prompted to save changes when he thinks he hasn't made any, use **`SETFLDSTATE()`** to reset any affected fields after initializing the value of your bound combo or list box.

One thing that will **not** initialize the value of a combo or list box is setting the selected property of one of its list items to true in the *Init* method of a form. The statement:

```
Thisform.MyList.Selected[1]
```

in the form's *Init* method, does **not** select an item in a combo or list box. This same statement in the form's *Activate* method will, however, achieve the desired result so we suspect the failure of the statement when used in the form's *Init* method might actually be a bug.

Quickfill combos *(Example: CH05.VCX::cboQFill)*

One example of quickfill methodology (and a more detailed explanation of just what "quickfill" is) was presented in Chapter 4 in the incremental search TextBox. This methodology is even easier to implement for the ComboBox class. Quickfill combo boxes give your forms a polished and professional look. They also make the task of selecting an item in the list much easier for the end user. Just type the letter 'S' in the text portion of the combo, and 'Samuel' is displayed. Next type the letter 'm' and the *DisplayValue* changes to 'Smith.' Very cool stuff!

Our quickfill combo can be used no matter what the *RowSource* of the ComboBox happens to be because it operates on the control's internal list. For this reason, it should only be used for controls that display, at most, a few dozen items. If you require this functionality, but need to display hundreds of items, we refer you to Tamar Granor's article on the subject in the September 1998 issue of *FoxPro Advisor*.

The quickfill combo has one custom property called *cOldExact*. Because we are looking for the first item that matches what has been typed in so far, we want to `SET EXACT OFF.` In the control's *GotFocus* method, the original value of `SET( 'EXACT' )`is saved so it can be restored when the control loses focus. When `SET( 'EXACT' ) = 'OFF',` there is no need to use the *LEFT()* function to compare what the user has typed so far to find the closest match in the list. This improves the performance of the search.

Just as in the incremental search TextBox described earlier, the *HandleKey* method is invoked from the control's *InteractiveChange* method after the keystrokes have already been processed. In fact, there is no code at all in the ComboBox's *KeyPress* method. The *InteractiveChange* method contains only the code necessary to determine if the key must be handled:

```
IF This.SelStart > 0
  *** Handle printable character, backspace, and delete keys
  IF ( LASTKEY() > 31 AND LASTKEY() < 128 ) OR ( LASTKEY() = 7 )
    This.HandleKey()
  ENDIF
ENDIF
```

Most of the work is accomplished in the *HandleKey* method. It iterates through the combo's internal list to find a match for what the user has typed in so far:

```
LOCAL lcSofar, lnSelStart, lnSelLength, lnRow

WITH This
  *** Handle backspace key
  IF LASTKEY() = 127
        .SelStart = .SelStart - 1
  ENDIF

  *** Save the insertion point and extract what the user has typed so far
  lnSelStart = .SelStart
  lcSofar =  LEFT( .DisplayValue, lnSelStart )

  *** Find a match in the first column of the combo's internal list
  FOR lnRow = 1 TO .ListCount
        IF UPPER( .List[ lnRow, 1 ] ) = UPPER( lcSoFar )
            .ListIndex = lnRow
            EXIT
        ENDIF
  ENDFOR

  *** Highlight the portion of the value after the insertion point
  .SelStart = lnSelStart
  lnSelLength = LEN( ALLTRIM( .DisplayValue ) ) - lnSelStart
  IF lnSelLength > 0
        .SelLength =  lnSelLength
  ENDIF
ENDWITH
```

This is all that is required for a quickfill combo that works with any *RowSourceType.* Just drop it on a form and set its *RowSourceType*, *RowSource* and *ControlSource* (if it is to be a bound control). Nothing could be easier. The form Quickfill.scx, provided with the sample code for this chapter, illustrates the use of this class with several different *RowSourceTypes.*

How do I add new items to my combo and list boxes?

(Example: CH05.VCX::cboAddNew and lstAddNew)

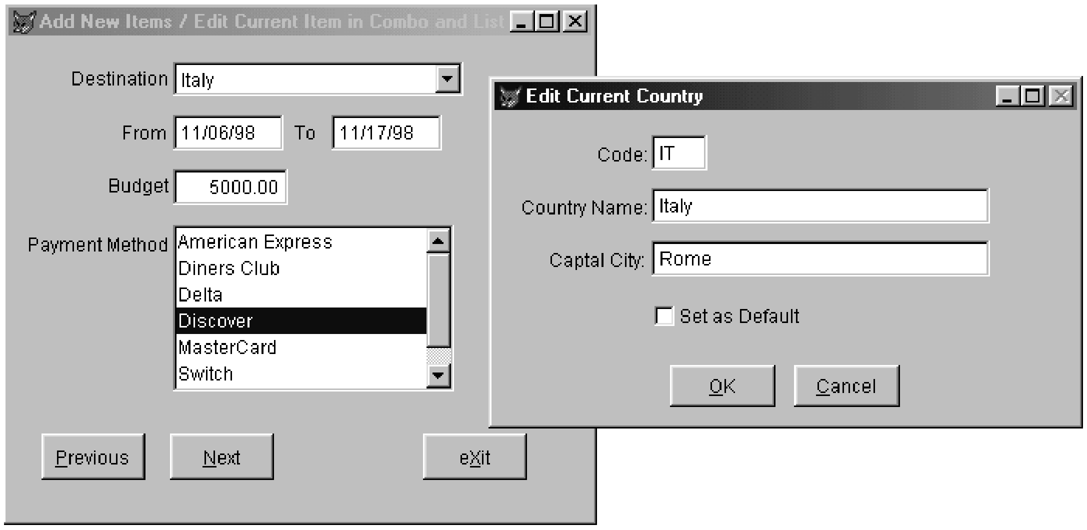


Figure 5.1 *Add new items and edit existing items in combo and list boxes*

Adding a new item to a ComboBox with style = 0-DropDown Combo is a pretty straightforward process because the control's *Valid* event fires whenever a selection is made from the list and again before it loses focus. When a user has typed something that is not in the current list, the control's *DisplayValue* property will hold the newly entered data but the *Value* property will be empty. A little code in the *Valid* method of the control allows you to determine whether the user selected an item in the list or typed a value not in the list. For example the following code could be used:

```
IF  NOT( EMPTY( This.DisplayValue ) ) AND EMPTY( This.Value )
  *** The user has typed in a value not in the list
```

However, this will not be reliable if the *RowSource* allows for empty values so a better solution is to use either:

```
IF  NOT( EMPTY( This.DisplayValue ) ) AND This.ListIndex = 0
```

OR

```
If  NOT( EMPTY( This.DisplayValue ) ) AND This.ListItemID = -1
```

You must then take action to add the new item to the control's *RowSource*. The code used to do this will be instance specific, depending on how the control is populated. If the combo's *RowSourceType* is "0-None" or "1-Value," use the *AddItem* or *AddListItem* method to add the new value to the list. If the *RowSourceType* is "2-Alias," "3-SQL Statement" or "6-Fields," the new item must be added to the underlying table and the combo or list box requeried to refresh its internal list. For *RowSourceType* "5-Array," add the item to the array and requery the control.

Although it is simple enough to add a new item to a DropDown Combo, this simplistic solution may not be adequate. If the only requirement is to add a new description along with its primary key to a lookup table, the methodology discussed above is up to the task. Much of the time, however, a lookup table contains more than two columns. (For example, the lookup table provided with the sample code for this chapter has a column for a user-defined code.) Additional fields may also need to be populated when a new item is added to the combo box.

We must also consider the fact that there is no quick and easy way to add new items to a ListBox. Considering how similar the ComboBox and ListBox classes are, we think it is appropriate that they share a common interface for adding new items. If the end-users add new items in the same manner, they have one thing to remember instead of two. The *cboAddNew* and *lstAddNew* classes provide this functionality through a shortcut menu invoked by right clicking the control. This shortcut menu also provides edit functionality. More often than not, if an item in a combo or list is misspelled, the user will realize it when he is selecting an item from the list. It is much more convenient to fix the mistake at this point, than having to use a separate maintenance form.

We created *cboAddNew* as a subclass of *cboQuickfill* to achieve a more consistent user interface. All of our custom combo box classes inherit from our quickfill combo class, so all behave in a similar manner. This type of consistency helps make an application feel intuitive to end users.

The "Add New" combo and list box classes have three additional properties. The *cForm2Call* property contains the name of the maintenance form to instantiate when the user wants to add or edit an item. The settings of the *lAllowNew* and *lAllowEdit* properties determine whether new items can be added or existing items edited. They are, by default, set to true because the object of this exercise is to allow new items to be added and current items to be edited. However, when designing the class we did our best to build in maximum flexibility, so these properties can be set to override this behavior at the instance level.

The code that does most of the work resides in the custom *ShowMenu* method and is called from the *RightClick* method of both. In this example, the combo's *BoundColumn* contains the primary key associated with the underlying data. It is assumed that the maintenance form will return the primary key after it adds a new item (If you need different functionality, code it accordingly.):

```
LOCAL lnRetVal, loparameters
PRIVATE pnMenuChoice
WITH This
  *** Don't display the menu if we can't add or edit
  IF .lAllowNew OR .lAllowEdit
    *** Display the shortcut menu
    pnMenuChoice = 0
    DO mnuCombo.mpr
    IF pnMenuChoice > 0
      *** Create the parameter object and populate it
      loParameters = CREATEOBJECT( 'Line' )
      loParameters.AddProperty('cAction', IIF( pnMenuChoice = 1, 'ADD', 'EDIT'
) )
      loParameters.AddProperty('uValue', .Value )
      *** Add any optional parameters if needed
      .AddOptionalParameters( @loParameters )
      *** Now call the maintenance form
      DO FORM ( .cForm2Call ) WITH loParameters TO lnRetVal
      lnValue = IIF(lnRetVal = 0, This.Value, lnRetVal )
      .Requery()
      .Value = lnValue
    ENDIF
  ENDIF
ENDWITH
```

The specifics of the maintenance form obviously depend upon the table being updated. However, any maintenance form called from the combo or list box's *ShowMenu* method will need to accept the parameter object passed to its *Init* method and use the passed information to do its job. It will also need to return to the required primary key after it has successfully added a new entry. While the specific fields to be updated by this process will vary depending on the table being updated, the process itself is fairly generic. All the forms used to add and edit entries are based on the *frmAddOrEdit* form class provided with the sample code for this chapter. This class clearly illustrates how the process works, and you can check out *Itineraries.scx* to see how this maintenance form is called by the *lstAddNew* and *cboAddNew* objects.

How do I filter the items displayed in a second combo or list box based on the selection made in the first? *(Example: FilterList.SCX)*

This is a lot easier than you may think. FilterList.scx, in the sample code for this chapter, not only filters a ListBox depending on what is selected in a ComboBox, it also filters the ComboBox depending on what is selected in the OptionGroup.

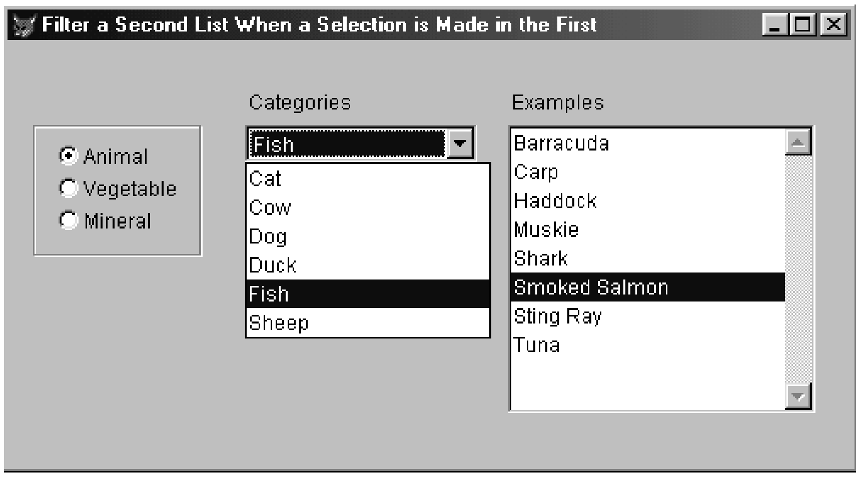


Figure 5.2 *Dynamically filter the contents of a combo or list box*

Our preferred *RowSourceType* for filtered controls is "3-SQL Statement" because it makes setting up the dependency easy. We just specify **`WHERE Somefield = ( Thisform.MasterControl.Value )`** in the *WHERE* clause of the dependent control's *RowSource.* Then, each time the dependent control is requeried, it gets populated with the appropriate values.

There are two little snags here. First, if we use the expression *ThisForm* in the dependent control's *RowSource* directly in the property sheet, Visual FoxPro kicks up a fuss at run time. It tells us that ThisForm can only be used within a method. Secondly, although we could set the dependent control's *RowSource* in its *Init* method, this may also result in some rather unpleasant run-time surprises. If the dependent control is instantiated before the master control, Visual FoxPro will complain that ThisForm.MasterControl is not an object.

The trick to making this work properly is to put the code initializing the *RowSources* of the dependent controls in the right place. Since the form's controls are instantiated before the form's *Init* fires, a custom method called from the form's *Init* method is a **good** place to put this sort of code. This is exactly what we have done in our sample form's *SetForm* method:

```
LOCAL lcRowSource

*** Select only the items that have a Cat_No equal to the option selected
*** in the option group
```

All the controls on the sample form are bound to form properties. The OptionGroup is bound to Thisform.nType. Because this property is initialized to 1 in the property sheet, all the controls contain a value when the form displays for the first time:

```
lcRowSource = 'SELECT Cat_Desc, Cat_Key, UPPER( Categories.Cat_Desc ) AS '
lcRowSource = lcRowSource + 'UpperDesc FROM Categories '
lcRowSOurce = lcRowSOurce + 'WHERE Categories.Cat_No = ( Thisform.nType ) '
lcRowSource = lcRowSource + 'INTO CURSOR csrCategories ORDER BY UpperDesc'
```

```
*** Now set up the combo's properties
WITH Thisform.cboCategories
  .RowSourceType = 3
  .RowSource = lcRowSource
  *** Don't forget to repopulate the control's internal list
  .Requery()
  *** Inialize it to display the first item
  .ListIndex = 1
ENDWITH
```

Now that we have initialized the categories combo box, we can set up the SQL statement to use as the *RowSource* for the detail list box. We want to select only the items that match the Item selected in the combo box:

```
lcRowSource = 'SELECT Det_Desc, Det_Key, UPPER( Details.Det_Desc ) AS '
lcRowSource = lcRowSOurce + 'UpperDesc FROM Details '
lcRowSource = lcRowSource + 'WHERE Details.De_Cat_Key = ( Thisform.nCategory )
'
lcRowSOurce = lcRowSOurce + 'INTO CURSOR csrDetails ORDER BY UpperDesc'

*** Now set up the list box's properties
WITH Thisform.lstDetails
  .RowSourceType = 3
  .RowSource = lcRowSource
  *** Don't forget to repopulate the control's internal list
  .Requery()
  *** Initialize it to display the first item
  .ListIndex = 1
ENDWITH
```

This code, in the *Valid* method of the OptionGroup, updates the contents of the ComboBox when a new selection is made. It also updates the contents of the ListBox so all three controls stay in synch:

```
WITH Thisform
  .cboCategories.Requery()
  .cboCategories.ListIndex = 1
  .lstDetails.Requery()
  .lstDetails.ListIndex = 1
ENDWITH
```

Finally, this code in the ComboBox's *Valid* method updates the contents of the ListBox each time a selection is made from the combo. This code will work just as well if placed in the ComboBox's *InteractiveChange* method. The choice of method, in this case, is a matter of personal preference:

```
WITH Thisform
  .lstDetails.Requery()
  .lstDetails.ListIndex = 1
ENDWITH
```

A word about lookup tables

It is inevitable that a discussion about combo and list boxes should turn to the subject of lookup tables. After all, combos and lists are most commonly used to allow the user to select among a set of values kept in such a table. Generally speaking, the descriptive text from the lookup table is displayed in a control that is bound to the foreign key value in a data file. But what is the best way to structure a lookup table? Should there be one, all-purpose lookup table? Or should the application use many specialized lookup tables? Once again, the answer is "It depends." You need to pick the most appropriate solution for your particular application. Of course to make an informed decision, it helps to know the advantages and disadvantages of each approach. So here we go…

There are two major advantages to using a single, consolidated lookup table. The first is that by using a single structure, you can create generic, reusable combo and list box lookup classes that are capable of populating themselves. This minimizes the amount of code needed at the instance level, and less code means less debugging. The second advantage to this approach is that your application requires only one data entry form for maintaining the various lookups. A two-page lookup maintenance form accomplishes this task quite nicely. The user can select the lookup category from a list on the first page. The second page then displays the appropriate items for the selected category. An edit button on page two can then be used to launch a modal form for editing the current item. The big disadvantage to this approach is that all lookups share a common structure. This means that if you have a lookup category requiring more information for each item, you must either create a separate lookup table for that category or add extra columns to your consolidated table that will seldom be used.

Using a separate table for each type of lookup in your application provides more flexibility than the previous approach. Increased flexibility also brings increased overhead. It is more difficult to create generic, reusable lookup classes. This solution also requires multiple lookup table maintenance forms.

We use a combination of the two approaches. A single, all-purpose lookup table works for simple items that are likely to be reused across applications. We use separate tables for specialized lookups that are likely to have a unique structure. Our standard lookup table is actually two tables: a lookup header table containing the lookup categories and an associated lookup detail table that holds the items for each category.

Table 5.5 *Structure of the Lookup Header table*

Field Name	Data Type	Field Length	Purpose
Lh_Key	Integer		Primary Key – uniquely identify record
Lh_Desc	Character	30	Lookup Category Description
Lh_Default	Integer		Default Value (if any) to use from Lookup Details Table

Table 5.6 *Data contained in the Lookup Header table*

Lh_Key	Lh_Desc	Lh_Default
1	Contact Types	1
2	Telephone Types	5
3	Countries	10
4	Business Types	23
5	Relationships	63
6	Colors	

Table 5.7 *Structure of the Lookup Details table*

Field Name	Data Type	Field Length	Purpose
Ld_Lh_Key	Integer		Foreign key from Lookup Header Table
Ld_Key	Integer		Primary Key – uniquely identify record
Ld_Code	Character	3	User defined code (if any) for item
Ld_Desc	Character	30	Lookup detail item description

Table 5.8 *Partial listing of data contained in the Lookup Details table*

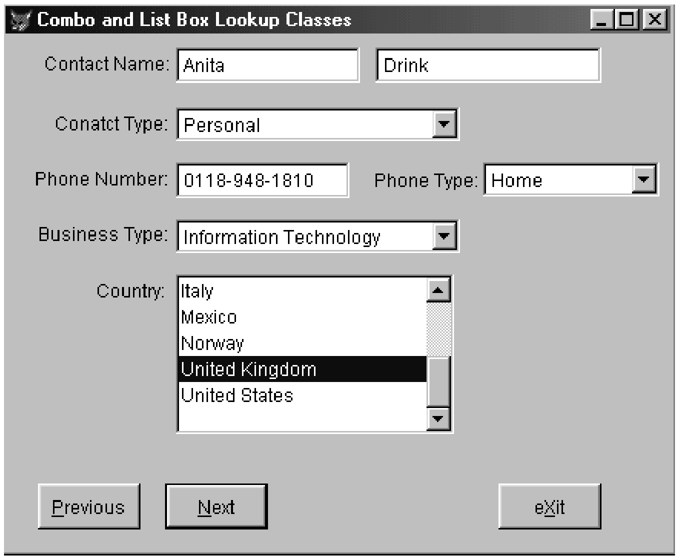
Ld_lh_key	Ld_Key	Ld_Code	Ld_Desc
1	1		Client
1	2		Prospect
1	3		Competitor
1	4		Personal
2	5		Home
2	6		Business
2	7		Fax
2	8		Cellular
2	9		Pager
3	10	USA	United States
3	11	UK	United Kingdom
3	12	CAN	Canada
3	13	GER	Germany

As you can see from the listings, it is quite easy to extract data from the detail table for any category. Since each item in the detail table has its own unique key, there is no ambiguity even if the same description is used in different "categories."

Generic lookup combos and lists *(Example: CH05.VCX::cboLookUp and lstLookUp)*

Generic, reusable combo and list box lookup classes allow you to implement your all-purpose lookup table with very little effort. Because of the way the lookup table is structured, it lends itself very well to *RowSourceType* = "3-SQL Statement." All that's required is a couple custom properties and a little code to initialize the control.

The *cCursorName* property is used to specify the cursor that holds the result of the SQL select. This is required in case there are multiple instances of the control on a single form. Each instance requires its own cursor to hold the result of the SQL statement in its *RowSource*. If the same name is used in every case, the cursor will be overwritten as each control instantiates. You will end up with all controls referring to the version of the cursor created by the control instantiated last.



Contact Name: Anita Drink

Conatct Type: Personal

Phone Number: 0118-948-1810 Phone Type: Home

Business Type: Information Technology

Country: Italy, Mexico, Norway, United Kingdom, United States

Figure 5.3 *Generic lookup combo and list box classes in action*

The *nHeaderKey* property is used to limit the contents of the control's list to a single category in the Lookup Header table. It contains the value of the primary key of the desired category and is used to construct the *WHERE* clause for the control's *RowSource*.

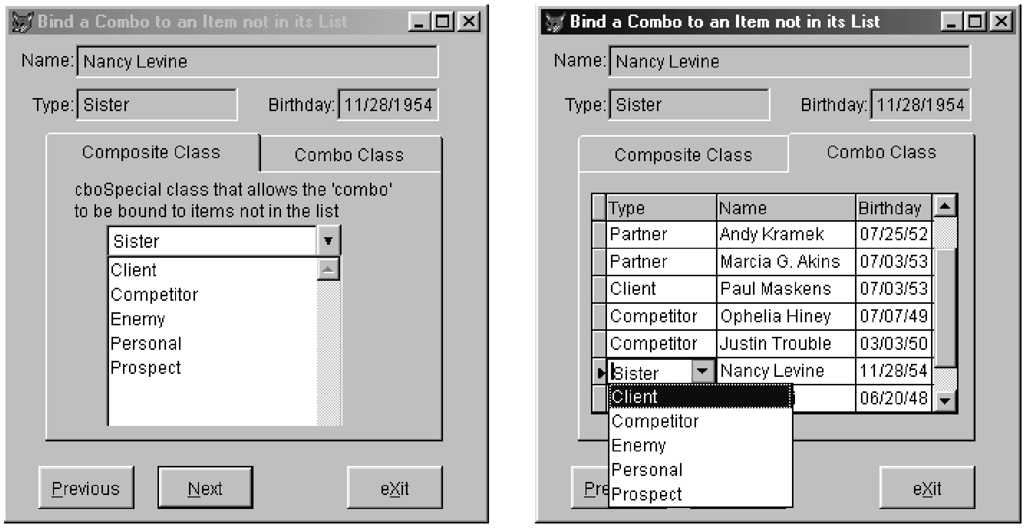
The control's *Setup* method, invoked upon instantiation, populates its *RowSource* using the properties specified above:

```
LOCAL lcRowSource

*** Make sure the developer set up the required properties
ASSERT !EMPTY( This.cCursorName ) MESSAGE ;
  'cCursorName MUST contain the name of the result cursor for the SQL SELECT!'
ASSERT !EMPTY( This.nHeaderKey ) MESSAGE ;
  'nHeaderKey MUST contain the PK of an item in LookupHeader.dbf!'

*** Set up the combo's RowSource
lcRowSource = 'SELECT ld_Desc, ld_Key FROM LookUpDetail WHERE '
lcRowSource = lcRowSource + 'LookUpDetail.ld_lh_key = ( This.nHeaderKey ) '
lcRowSource = lcRowSource + 'INTO CURSOR ( This.cCursorName ) '
lcRowSource = lcRowSource + 'ORDER BY ld_Desc'

*** Set up the combo's properties
WITH This
  .RowSourceType = 3
  .RowSource = lcRowSource
  .ColumnWidths = ALLTRIM( STR( .Width ) ) + ',0'
  .Requery()
ENDWITH
```

Using the lookup combo is very easy. Just drop it on a form, set its *ControlSource*, and fill in the *cCursorName* and *nHeaderKey* properties. Since it inherits from the cboAddNew class, it is also possible to add new entries to the lookup table and edit existing items on the fly. Since all instances of our generic lookup combo and list boxes populate their lists from the same generic lookup table, we can even put the name of the lookup maintenance form in the class's *cForm2Call* property. What could be easier? Lookups.scx, which is included with the sample code for this chapter, illustrates just how easy it is.

So what if I want to bind my combo to a value that isn't in the list? *(Example: CH05.VCX::cboSpecial and CH05.VCX::cboNotInList)*

The first question that leaps to mind here is "Then why are you using a combo box?" Combo and list boxes are used to limit data entry to a set of predefined selections. Permitting the user to enter an item that isn't in the list defeats the purpose of using a combo box in the first place. Having said that, we realize there may be occasions where this sort of functionality is required. For example, let's suppose a particular field in a table is usually populated from a set of standard selections. However, occasionally none of the standard selections are suitable and the end user needs to enter something that is not in the list. Clearly, if the non-standard items were regularly added to the underlying lookup table, the table would grow quickly with seldom used entries. In this instance, the field bound to the combo box must contain the description of the item in the lookup table and not its key value. Obviously, if we allow the field to be bound to items that are not in the list, we must store the information in this non-normalized manner. The only place to "look up" such ad hoc items is in the bound field itself!



Name: Nancy Levine
Type: Sister
Birthday: 11/28/1954

cboSpecial class that allows the 'combo' to be bound to items not in the list

Sister
Client
Competitor
Enemy
Personal
Prospect

Type	Name	Birthday
Partner	Andy Kramek	07/25/52
Partner	Marcia G. Akins	07/03/53
Client	Paul Maskens	07/03/53
Competitor	Ophelia Hiney	07/07/49
Competitor	Justin Trouble	03/03/50
Sister	Nancy Levine	11/28/54
Client		06/20/48

Client
Competitor
Enemy
Personal
Prospect

Figure 5.4 *Special combo class that binds to items not in the list*

Since such a combo can only be used when the table to which it is bound is not normalized, we are tempted to use a combo box with RowSourceType = 1-None and its *Sorted* property set to true. This code, in the combo's *Init* method, populates it with the types that currently exist in our "People" table:

```
LOCAL lnSelect
LnSelect = SELECT()
SELECT DISTINCT cType FROM People ORDER BY cType INTO CURSOR csrTypes
IF _TALLY > 0
  SELECT csrTypes
  SCAN
    This.AddItem( csrTypes.cType )
  ENDSCAN
ENDIF
SELECT ( lnSelect )
```

Then in the combo's *Valid* method, we could check to see if the user typed a value not in the list and add it:

```
WITH This
  IF !( UPPER( ALLTRIM( .DisplayValue ) ) == UPPER( ALLTRIM( .Value ) ) )
    .AddItem ( .DisplayValue )
  ENDIF
ENDWITH
```

While it's true this code works, there are a few fundamental problems. First, if the source table has a large number of records, the initial query could cause significant delay when the form is instantiated. Second, this will not solve the original problem. If new items are always added to the list, the list will continue to grow, making it difficult for the user to select an entry.

This illustration also does not allow the user to distinguish which items are the "standard" selections in the list and which are merely ad hoc entries.

Our cboSpecial class, consisting of a text box, list box, and command button inside a container, solves the problem. The text box is the bound control. The list box is left unbound and its *RowSource* and *RowSourceType* are populated to display the standard selections. The class contains code to provide the text box with "quick fill" functionality and to synchronize the display in its contained controls.

The text box portion of the class uses this code in its *KeyPress* method to make the list portion visible when the user presses **`ALT+DNARROW`** or **`F4`**. It also makes sure the list becomes invisible when the **`TAB`** `or` **`ESC`** keys are pressed:

```
*** <ALT>+<DNARROW> OR <F4> were pressed
IF nKeyCode = 160 OR nKeyCode = -3
  This.Parent.DropList()
  NODEFAULT
ENDIF

*** <TAB> or <ESC> were pressed
IF nKeyCode = 9 OR nKeyCode = 27
  This.Parent.lstSearch.Visible = .F.
ENDIF
```

The only other code in the text box resides in its *InteractiveChange* method and its only purpose is to invoke the container's *Search* method:

```
*** If a valid character was entered, let the parent's search method handle it
IF This.SelStart > 0
  IF ( LASTKEY() > 31 AND LASTKEY() < 128 ) OR ( LASTKEY() = 7 )
    This.Parent.Search()
  ENDIF
ENDIF
```

The container's *Search* method then does the required work:

```
LOCAL lcSofar, lnSelStart, lnSelLength, lnRow

WITH This
  WITH .txtqFill
    *** Handle backspace key
    IF LASTKEY() = 127
      .SelStart = .SelStart - 1
    ENDIF
    *** Get the value typed in so far
    lnSelStart = .SelStart
    lcSofar =  LEFT( .Value, lnSelStart )
  ENDWITH
  *** Find a match in column #1 of the list portion of this control
  WITH .lstSearch
    *** Reset the list index in case we have type ion something that is not
    *** in the list
    .ListIndex = 0
    FOR lnRow = 1 TO .ListCount
      IF UPPER( .List[ lnRow, 1 ] ) = UPPER( lcSoFar )
```

```
        .ListIndex = lnRow
        *** Synchronize the contents of the textbox with what is selected
        *** in the list
        This.txtQfill.Value = .Value
        EXIT
      ENDIF
    ENDFOR
  ENDWITH

  WITH .txtqFill
    *** Highlight the portion of the value after the insertion point
    .SelStart = lnSelStart
    lnSelLength = LEN( ALLTRIM( .Value ) ) - lnSelStart
    IF lnSelLength > 0
      .SelLength =  lnSelLength
    ENDIF
  ENDWITH
ENDWITH
```

The control's list box portion contains code in its *KeyPress* and *InteractiveChange* methods to update the text box's value with its value when the user selects from the list. This code is required in both methods because pressing either the **ENTER** key or the **SPACE BAR** to select a list item will cause its *KeyPress* event to fire but will not fire its *InteractiveChange* event. Selecting an item with the mouse fires the *InteractiveChange* event but does not fire the *KeyPress*, even though **LASTKEY()** returns the value 13 whether the mouse or the **ENTER** key is used. This code, from the list box's *InteractiveChange* method, is similar but not identical to the code in its *KeyPress* method:

```
IF LASTKEY() # 27
  This.Parent.TxtQFill.Value = This.Value
  *** a mouse click gives a LASTKEY() value of 13 (just like pressing enter)
  IF LASTKEY() = 13
    This.Visible = .F.
    This.Parent.txtQfill.SetFocus()
  ENDIF
ENDIF
```

The only other code in the list box portion of the class resides in its *GotFocus* method. This code simply invokes the container's *RefreshList* method to ensure the selected list box item is synchronized with the text box's value when the list box is made visible:

```
LOCAL lnRow
WITH This.lstSearch
  .ListIndex = 0
  FOR lnRow = 1 to .Listcount
    IF UPPER( ALLTRIM( .List[ lnRow ] ) ) = ;
       UPPER( ALLTRIM( This.txtQFill.Value ) )
      .ListIndex = lnRow
      EXIT
    ENDIF
  ENDFOR
ENDWITH
```

One of the shortcomings of the container class is that is cannot easily be used in a grid. To fulfill this requirement, we created the cboNotInList combo class. As you can see in **Figure 5.4**, when the list in the grid is dropped, the first item is highlighted when the *DisplayValue* is not in the list. That is why we prefer to use the cboSpecial class when it does not need to be placed in a grid. When *DisplayValue* is not in the list, notice no selection of the composite class is highlighted.

The cboNotInList combo class uses the same trick that we used when constructing the numeric textbox in Chapter 4. Although it may be a bound control, we unbind it behind the scenes in its *Init* method and save its *ControlSource* to the custom *cControlSource* property:

```
IF DODEFAULT()
  WITH This
    .cControlSource = .ControlSource
    .ControlSource = ''
  ENDWITH
ENDIF
```

The combo's *RefreshDisplayValue* method is called from both its *Refresh* and *GotFocus* methods to update its *DisplayValue* with the value of the field to which it is bound. It is interesting to note that it is only necessary to invoke this method from the combo's *GotFocus* when the combo is in a grid:

```
LOCAL lcControlSource
WITH This
  IF ! EMPTY( .cControlSource )
    lcControlSource = .cControlSource
      .DisplayValue = &lcControlSource
  ENDIF
ENDWITH
```

Finally, the combo's *UpdateControlSource* method is called from its *Valid* to (you guessed it) update its control source from its *DisplayValue:*

```
LOCAL lcAlias, lcControlSource
WITH This
  IF ! EMPTY( .cControlSource )
    lcAlias = JUSTSTEM( .cControlSource )
    IF UPPER( ALLTRIM( lcAlias ) ) = 'THISFORM'
      lcControlSource = .cControlSource
      STORE .DisplayValue TO &lcControlSource
    ELSE
      REPLACE ( .cControlSource ) WITH .DisplayValue IN ( lcAlias )
    ENDIF
  ENDIF
ENDWITH
```

See NotInList.scx in the sample code for this chapter to see both these classes in action.

How do I disable individual items in a combo or list?

Our first reaction is "Why are you displaying items in a combo box that the user is not allowed to select?" It certainly seems to be at odds with the basic reasoning for using a combo box in the first place: to allow the user to choose from a pre-defined list of allowable entries. If the user is not allowed to select an item, what the heck is it doing in the combo or list box to begin with? Again we realize there may be valid scenarios that require such functionality. For example, the CPT codes used in medical applications to define various transactions and the ICD-9 code used to define standard diagnoses may change over time. One cannot merely delete codes that are no longer used because historical detail may be linked to these obsolete codes. When displaying this historical data, it would be useful to display the inactive code as disabled. This way, a combo box bound to the ICD-9 code would not "lose" its display value because the code could not be found in the control's list. Nor would the user be permitted to select it for a current entry since it would be disabled.

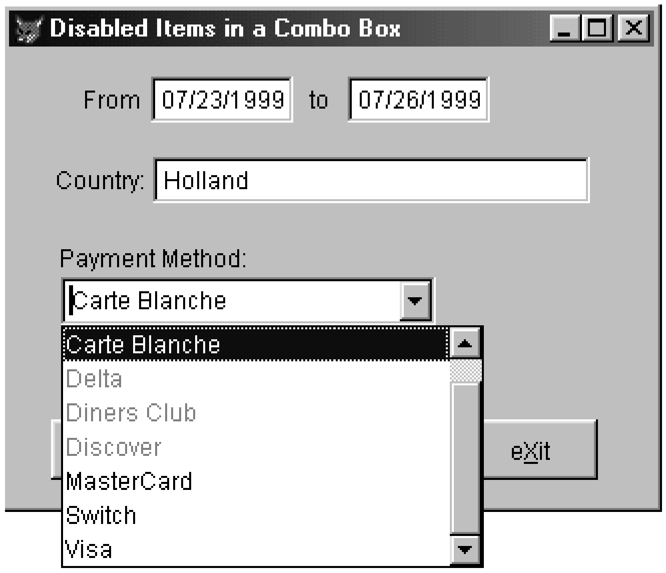


Figure 5.5 *Disabled items in a combo box*

To disable an item in a combo or list box, just add a back slash to the beginning of the string that defines the content of the first column in the control's *RowSource*. However, this only works for *RowSourceTypes* "0-None", "1-Value", and "5-Array". The form DisabledItems.scx, provided with the sample code for this chapter, provides an example for a combo with *RowSourceType* "5-Array". This code in the combo box's *Init* method populates the array and disables items only if the date in the underlying table indicates this item is inactive. This type of logic can be also used to disable specific items based on the value of a logical field that determines if the item is still active:

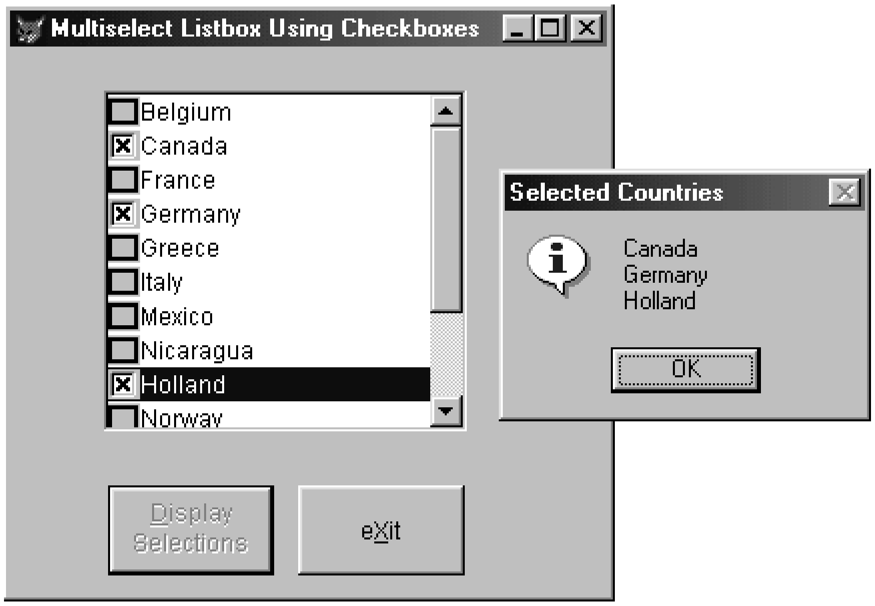
```
SELECT IIF( !EMPTY( PmtMethods.pm_dStop ), '\'+pm_Desc, pm_Desc ) AS pm_Desc, ;
   pm_Key FROM PmtMethods ORDER BY pm_Desc INTO ARRAY This.aContents
This.Requery()
```

We could just as easily set the combo box up with a *RowSourceType* of "0-None", set its *Sorted* property to true and populated it like this instead:

```
LOCAL lcItem

SELECT PmtMethods
SCAN
  lcItem = IIF( EMPTY( dStop ), dStop, '\' + dStop )
  This.AddItem( lcItem )
ENDSCAN
```

How do I create a list box with check boxes like the one displayed by Visual FoxPro when I select "View Toolbars" from the menu? *(Example: CH05.VCX::lstChkBox)*

Did you ever notice that list boxes have a *Picture* property? You can find this property listed under the layout tab of the property sheet. Technically speaking, the picture property really belongs to the list box's *items* rather than to the list box as an object. Essentially, you can manipulate the *Picture* property of the current *ListItem* and set it to a checked box if it is selected and to a plain box if it is not.



Canada
Germany
Holland

Figure 5.6 *Multiselect list box using check boxes*

List boxes of this type exhibit other nonstandard behavior that requires special programming. For example, clicking on a selected item in this list box de-selects it. Pressing the space bar also acts like a toggle for selecting and de-selecting items. This is much more convenient than the standard **CTRL+CLICK** and **CTRL+SPACE BAR** combinations normally used for selecting multiple items in a list box.

This list box class requires a custom array property to track the selected items. We can't just use the native *Selected* property because it is not always set in the expected way. For example, pressing the space bar selects the current item. But the item's *Selected* property isn't set to true until after the *KeyPress* method completes. Suppose we want to manipulate this property in the list box's *KeyPress* method. `This.Selected[ This.ListIndex ]` would return false even if we had just pressed the space bar to select the current item! You can see the dilemma. And it becomes even more complicated when trying to use the key to toggle the item's *Selected* property. The simple and straightforward solution is to maintain our own list of current selections over which we have total control. This is implemented using a custom array property (*aSelected*), initialized in the control's *Reset* method, which is called from its *Init:*

```
WITH This
  *** clear all selections
  *** If other behavior is required by default, put it here and call this
  *** method from any place that the list box's contents must be reset
  .ListIndex = 0
  DIMENSION .aSelected[.ListCount]
  .aSelected = .F.
ENDWITH
```

The *SetListItem* method is called from the list box's *Click* and *KeyPress* methods. Since the *Click* event fires so frequently (every time the user scrolls through the list using the cursor keys) we must make sure we only call the *SetListItem* method when the user actually clicks on an item. We do this by checking for `LASTKEY() = 13`. Likewise, it is only invoked from the *KeyPress* method when the user presses either the **ENTER** key or the **SPACE BAR**. This method sets the specified row in the custom array that tracks the list box selections and sets the item's *Picture* property to the appropriate picture for its current state:

```
WITH This
  IF .ListIndex > 0
    *** The item is currently selected so de-select it
    IF .aSelected[.ListIndex]
      .Picture[.ListIndex] = 'Box.bmp'
      .aSelected[.ListIndex] = .F.
    ELSE
      *** The item is not selected yet, so select it
      .Picture[.ListIndex] = 'CheckBx.bmp'
      .aSelected[.ListIndex] = .T.
    ENDIF
  ENDIF
ENDWITH
```

Visual FoxPro's standard behavior is to de-select all list box items when it gets focus. It seems that this default behavior also resets all of the pictures in the list. Because we do not want our multi-select list box to exhibit this type of amnesia, we call our its custom *RefreshList* method from its *GotFocus* method:

```
LOCAL lnItem
WITH This
  FOR lnItem = 1 TO .ListCount
    .Picture[ lnItem ] = IIF( .aSelected[ lnItem ], 'CheckBx.bmp', 'Box.bmp' )
  ENDFOR
ENDWITH
```

A mover list class *(Example: CH05.VCX::cntMover)*

Mover lists, like the "check box" list box class discussed above, are more user-friendly than the multi-select list box. Although they appear to be complex controls, it is relatively simple to create a generic, reusable, mover list class. Our mover list class consists of a container with one list box to hold the set of items from which the user may choose and another that will be populated with the selected items. It also contains four command buttons to move items between the two lists. The mover bars on the destination list are enabled to provide maximum flexibility. If the order of the selected items is critical to the mover list's functionality, this provides a mechanism for the user to order his selections. The container's *ResetList* method, which initially populates the lists, is the only method that is instance specific. The form MoverList.scx, provided with the sample code for this chapter, uses this method to populate the mover's source list from our "PmtMethods" table:

```
WITH This
  .lstSource.Clear()
  .lstDestination.Clear()
  SELECT Countries
  SCAN
    .lstSource.AddItem( c_Desc )
  ENDSCAN
ENDWITH
```

Our mover list class is by no means the last word in movers, but it will give you something to build upon.

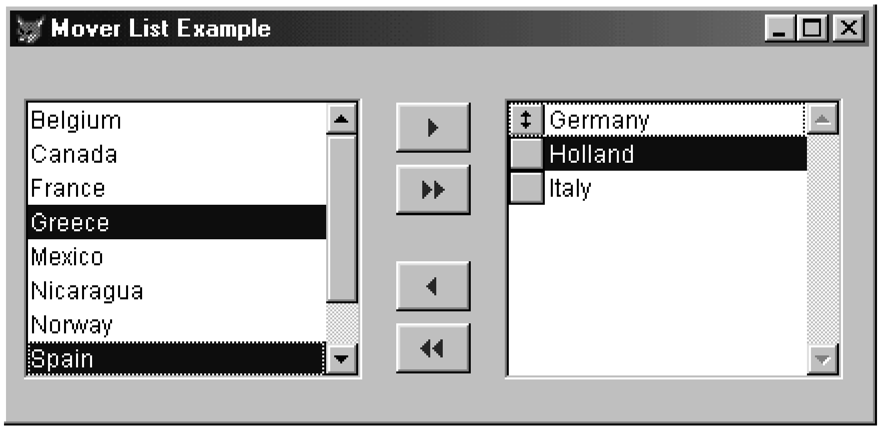


Figure 5.7 *Simple mover list*

The command buttons are not the only way to move items between the two lists. Double clicking on an item removes it from the current list and adds it to the other. Selected items may also be dragged from one list and dropped into the other. In fact, it doesn't matter how the items are moved between the lists. All movement is accomplished by invoking the container's *MoveItem* method and passing it a reference to the source list for the move. There is a call to this method from the *DblClick* and *DragDrop* methods of each of the list boxes as well as in the *OnClick* method of the two command buttons, *cmdMove* and *cmdRemove*:

```
LPARAMETERS toSource
LOCAL lnItem, toDestination
```

Since this method is passed an object reference to the source list, we can use this reference to determine which is the destination list:

```
WITH This
  IF toSource = .lstSource
    toDestination = .lstDestination
  ELSE
    toDestination = .lstSource
  ENDIF
ENDWITH

*** Lock the screen so to avoid the distracting visual side-effects
*** that result from moving the item(s)
THISFORM.LockScreen = .T.
```

We now loop through the source list to add each selected item to the destination list and remove it from the source list afterward. As items are removed from the source list, their *ListCount* properties are decreased by one. So we use a `DO WHILE` loop instead of a `FOR` loop to have more control over when the index into the source list is incremented. We increment it only when the current item in the list is not selected to move on to the next:

```
lnItem = 1
WITH toSource
  DO WHILE lnItem <= .ListCount
    IF .Selected[ lnItem ]
      toDestination.AddItem( .List[ lnItem ] )
      .RemoveItem( lnItem )
    ELSE
      lnItem = lnItem + 1
    ENDIF
  ENDDO
ENDWITH

*** Don't forget to unlock the screen!
THISFORM.LockScreen = .F.
```

Implementing drag and drop functionality requires a little more code. To accomplish this, we have added the n*MouseX*, n*MouseY* and *nDragThreshold* properties to the container. The *MouseDown* method of the contained list boxes sets the *nMouseX* and *nMouseY* properties to the current coordinates when the left mouse button is depressed. This is done to prevent dragging an item when the user has just clicked to select it. The drag operation does not begin unless the mouse has moved at least the number of pixels specified by the *nDragTheshold* property. The container's *StartDrag* method, called from the list boxes' *MouseMove* method, checks to see if the mouse has been moved sufficiently to start the *Drag* operation. The calling list passes an object reference to itself and the current coordinates of the mouse to this method:

```
LPARAMETERS toList, tnX, tnY

WITH This
  *** Only begin the drag operation if the mouse has moved
  *** at least the minumun number pixels
  IF ABS( tnX - .nMouseX ) > .nDragThreshold OR ;
    ABS( tnY - .nMouseY ) > .nDragThreshold
    toList.Drag()
  ENDIF
ENDWITH
```

Finally, the list's *DragIcon* property from which the drag operation originated, needs to be changed to the appropriate icon. When the cursor is on a portion of the screen that does not permit the item to be dropped, we want to display the familiar circle with a slash inside it that universally means "NO!" We do this by calling the container's *ChangeIcon* method from the list's *DragOver* method. This method, as the name implies, merely changes the drag icon to the appropriate icon:

```
LPARAMETERS toSource, tnState

IF tnState = 0
  *** allowed to drop
  toSource.DragIcon = THIS.cDropIcon
ELSE
  IF tnState = 1
    *** not allowed to drop
    toSource.DragIcon = THIS.cNoDropIcon
  ENDIF
ENDIF
```

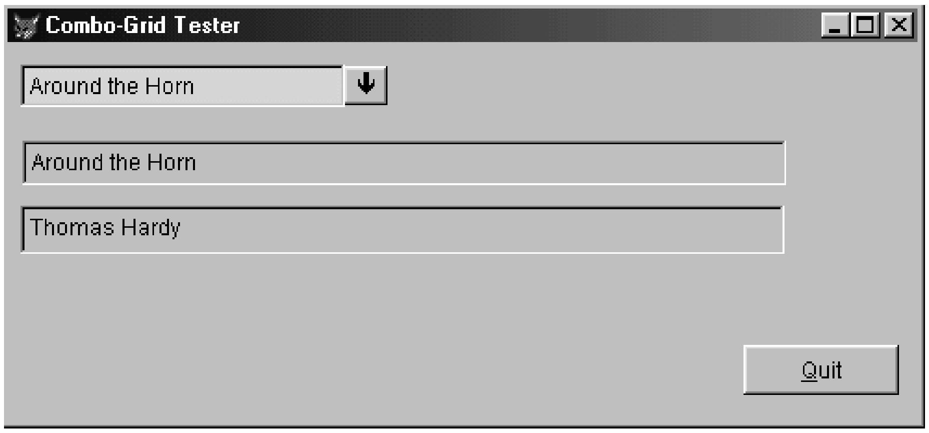
Not only is the mover list a lot easier to use than the multi-select list box, it also gives the application a more professional look and feel – without a lot of extra effort.

What if I need to display hundreds of items in my combo box?

In this case, a combo box is too slow because it must always populate its internal list with all items from the underlying data source. If we were able to combine the efficiency of a grid (which can load data as needed) with the incremental search capability of our quickfill combo box, we would have the prefect solution. Fortunately, using composition, we can do exactly that and we created a class for this "*ComboGrid*" control that uses a textbox, a command button and a grid as its main components.

The construction is similar to that described for our "*cboSpecial*" class - used to permit binding a control to a value not contained in the internal list. The main difference is that instead of the list box used in the earlier control, we now use a grid for the drop-down functionality. Some special handling is required at the container level to ensure that everything operates smoothly but the techniques are the same.

This control was originally written and posted as a "*FreeHelp*" contribution to the CompuServe Visual FoxPro forum, and later was used as the basis for an article in *FoxTalk* magazine. For a full discussion of the design and implementation of this control see "*Now you see it, now you don't*" an article in the July 1999 edition of *FoxTalk*. Figures 5.8 and 5.9 show the two incarnations of the combo grid:

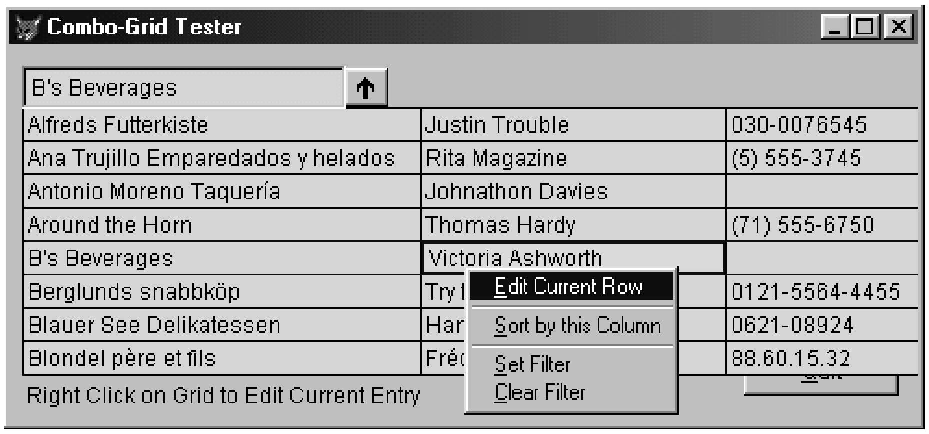


Around the Horn

Around the Horn

Thomas Hardy

Figure 5.8 *The dormant ComboGrid*



B's Beverages

Alfreds Futterkiste	Justin Trouble	030-0076545
Ana Trujillo Emparedados y helados	Rita Magazine	(5) 555-3745
Antonio Moreno Taquería	Johnathon Davies	
Around the Horn	Thomas Hardy	(71) 555-6750
B's Beverages	Victoria Ashworth	
Berglunds snabbköp	Try t	0121-5564-4455
Blauer See Delikatessen	Har	0621-08924
Blondel père et fils	Fréc	88.60.15.32

Right Click on Grid to Edit Current Entry

Figure 5.9 *The active ComboGrid*

Release Notes for the ComboGrid Class (*Class: CboGrid.vcx, Example Form: FrmCGrd.scx*)
Created By: Marcia G. Akins and Andy Kramek, and placed in the Public Domain in February 1999.

This class is designed to use a grid in place of a standard VFP Combobox for a large data volume or to enter data into more than one lookup table field, used to populate the dropdown. The class consists of a Textbox, a Command Button and a Grid inside a Container. The grid is normally invisible and the class looks and behaves just like a standard VFP combo box.

The class is controlled by 7 properties, as follows:

***Table 5.9** ComboGrid Custom Properties*

Property	**Function**
CAlias	Name of the Table to use as the RecordSource for the Grid
CColSource	Comma separated list of field names for columns in the Grid
CControlSource	Data Source for the Textbox (if required to be bound)
CkeyField	Field to use to set the Value of the TextBox
CtagName	Tag to use in cAlias
LAddNewEntry	Determines whether new entries are added to the Grid Table as well as to the textbox controlsource
NColCount	Number of columns to display in the grid

A single Instance level method is available so data can be read from the grid after a selection is made, for example, to populate other fields on the form:

***Table 5.10** ComboGrid custom methods*

Method	**Function**
RefreshControls	Instance Level Method called by Grid AfterRowColChange() and by KeyPress in the QuickFill TextBox

There are four ways of using this control depending on the setting of the cControlSource and lAddNewEntry properties as follows:

- cControlSource:= “” and lAddNewEntry = .F. The text box is NOT bound and although the incremental search will operate, and new data may be entered into the text box, new records will not be added to the grid’s underlying table.
- cControlSource:= “” and lAddNewEntry = .T. The text box is NOT bound and new data may be entered into the text box, new records will be added to the grid’s underlying table. Additional columns may have data entered by right clicking in the appropriate cell.
- cControlSource:= <Specified> and lAddNewEntry = .F. The text box is bound and will read its initial value from the specified table and will write changes to that field back to the table. New records will not be added to the grid’s underlying table.
- cControlSource:= <Specified> and lAddNewEntry = .T. The text box is bound and will read its initial value from the specified table and will write changes to that field back to the table. Additional columns may have data entered by right clicking in the appropriate cell.

Acknowledgements
Thanks are due to Tamar Granor for the QuickFill methodology that she published in the September 1998 issue of *FoxPro Advisor* and to Paul Maskens who provided the inspiration for the self-sizing container and grid.

Following a suggestion by Dick Beebe, the ComboGrid was enhanced to incorporate a right-click menu in the drop-down grid. Depending on the settings, you can either sort by the selected column or edit the currently selected row. If neither is appropriate the menu does not appear.

To enable editing, the Container's lAddNewEntry property must be set. To enable sorting the grid's data source, you must have an index tag named exactly the same as the field used as the controlsource for the column. If this is not your normal practice, you should probably consider it anyway.

Author's disclaimer
As always we have tried to make the ComboGrid foolproof, but we have placed this control class in the Public Domain "as is," with no warranty implied or given. We cannot accept liability for any loss or damage resulting from its use (proper or improper). All source code is included in the CboGrid sub-directory of this chapter's sample code, so feel free to modify, change or enhance it to meet your specific needs.