June, 2006

Advisor Answers

Cascading combos and lists

VFP 9/8/7

Q: I want to put three comboboxes on a form, where the user's choice in the first combo determines the contents of the second combo and the user's choice in the second combo determines the contents of the third combo. For example, when the user chooses a state from the first combo, the second combo should show the list of cities in that state. When the user chooses a city, the third combo should show customers in that city. How can I set this up?

A: This is a question I've been seeing a lot lately, but it's certainly not new. I answered it in the September, 1996 Advisor Answers column. But I've learned a lot since then and while the technique I suggested is still valid, it's not exactly the way I'd approach this problem today. In particular, because this is a common need, building combobox and listbox classes that contain as much of the necessary code as possible makes sense.

First, a note about combos and lists. Although their physical appearance is different, under the hood, they're quite similar. Both have a RowSource that supplies their content and a variety of RowSourceTypes to choose from. Over the years, I've found that using a RowSourceType of 5-Array tends to provide the easiest handling.

Combos and lists also have the same internal mechanisms. Each has a ListIndex property pointing to the current item, a Requery method to call when the underlying list is updated, and a BoundTo property that lets you bind numeric variables, fields and properties to the control. There's a lot more in common than this, but those are all the significant items for this problem.

To start, I subclassed both the ComboBox class and the ListBox class. In each, I set the following properties:

```
BoundTo = .T.
RowSourceType = 5-Array
RowSource  = This.aItems
```

I added two custom properties and a custom method to each of my new classes. The properties are aItems, an array to hold the list of

items and cSubCombo, which holds the name of a combo or list in the same container that depends on the value of this control. I set cSubCombo to the empty string. (Remember that to add an array property in the Add Property dialog, you need to include the brackets and a size, like this: aItems[1].) The custom method is GetItems, which will be used in instances to populate the aItems array.

I then added code to the Init method to fill the array and choose the first item initially:

```
DODEFAULT()

This.aItems[1] = "" && in case no matches
This.GetItems()

IF EMPTY(This.ControlSource)
   This.ListIndex = 1
ELSE
   This.Refresh()
ENDIF
```

There's a little more code in the Requery method, because it also ensures that any combo or list depending on this control gets updated when this one's underlying list changes:

```
This.GetItems()

DODEFAULT()

IF EMPTY(This.ControlSource)
   This.ListIndex = 1
ELSE
   This.Refresh()
ENDIF

IF NOT EMPTY(This.cSubCombo)
   oCombo = EVALUATE("This.Parent." + This.cSubCombo)
   oCombo.Requery()
ENDIF
```

Finally, I added code to the Valid method, so that whenever the user changes the value of the combo or list, the dependent control is updated.

```
DODEFAULT()
IF NOT EMPTY(This.cSubCombo)
   oCombo = EVALUATE("This.Parent." + This.cSubCombo)
   oCombo.Requery()
ENDIF
```

That's all it takes to set up this mechanism generally. To use these controls, drop one onto a form, add code to the GetItems method to fill the array, and if another combo or list depends on this one, set cSubCombo to the name of that control.

To demonstrate, I created a form (shown in Figure 1) using the Northwind database. Choosing a Customer from the first combo updates the list of orders in the Order combo. Choosing an order updates the list of products (all the products included in that order) in the listbox. The form and the combo and list subclasses are included on this month's Professional Resource CD.

Figure 1. Cascading combos and lists—It only takes a little code to set up combos and lists to update when a selection is made in another list or combo.

While the key points here (using Requery to repopulate the control, calling one control's Requery method from the other control's Valid) are the same as what I suggested in 1996, this solution is more self-contained and easier to use than my original ideas.

–Tamar