ComboBoxes are Looking Up

Doug Hennig

Most applications need a way for a user to select a lookup value for certain fields. In this, the first of a two-part article on reusable tools for managing lookups, Doug presents a combobox class for lookup selection and a form for maintaining lookup values.

Almost every application I've ever written required lookup tables. By "lookup" tables, I mean one or more tables containing the range of values for one or more fields in the "main" tables. For example, you might have a customer table with a TITLE field for the title of the contact. Rather than having the user type the title, you decide to have a table of titles and store the ID for the title of a given contact in the TITLE field. The titles table is related one-to-many to the customer table, with TITLES.ID = CUSTOMER.TITLE. Another simple example is a states lookup table so the user can enter "TX" rather than typing "Texas".

A table of lookup values typically has a simple structure: ID and NAME (or DESCRIPTION or VALUE or something similar). The ID field can be either assigned by the user (for example, the state code in the case of a states table) or a surrogate key assigned by the system (almost always a better idea). Since an application can have dozens or even hundreds of fields that require lookup values, many developers choose to have a single lookup table for an entire application rather than having dozens or hundreds of small, identically structured lookup tables with just a few records each. The way you distinguish which records contain the values for a given field in a given table is with TABLE and FIELD fields. For example, Table 1 shows lookup values for CUSTOMER.TITLE, CUSTOMER.CATEGORY, and EMPLOYEE.TYPE. To present a list of lookups for a single field (such as in a combobox bound to that field), you simply set a filter or perform a SQL SELECT so you have a set of those records for the desired table and field.

Table 1. Records in a single lookup table.

TABLE	FIELD	NAME
CUSTOMER	TITLE	Owner
CUSTOMER	TITLE	President
CUSTOMER	TITLE	Vice-President
CUSTOMER	CATEGORY	Prospect
CUSTOMER	CATEGORY	Dead-beat
CUSTOMER	CATEGORY	Valued client
EMPLOYEE	TYPE	Manager
EMPLOYEE	TYPE	Clerk

I use a table called, strangely enough, LOOKUPS.DBF. It has the following structure: ID I, TABLE C(30), FIELD C(30), and NAME C(40). There's a primary index on ID and a tag called LOOKUPS on upper(TABLE + FIELD).

So, the question comes up: how does the user select a specific lookup value for a given record? Although there may be a lot of approaches to this, I've typically used three of them: a textbox where the user enters a code representing the lookup value (usually used when the ID in the lookup table is assigned by the user or for heads-down data entry where a commonly understood code, such as "MGR" for manager and "CLK" for clerk, is used), a combobox displaying the lookup values (usually used when the ID is system-assigned), and an auto-fill combobox in which the user starts typing the lookup value and the combobox automatically fills with the first value matching what they've typed (discussed in my column in the March 1997 issue of FoxTalk). In this month's column, we'll use a combobox for selecting lookup values; next month, we'll discuss a textbox and the subsequently required picklist.

SFLookupComboBox Class

Because I found myself frequently creating comboboxes bound to lookup values, I created a generic class that does this for me. SFLookupComboBox (defined in SFCCTRLS.VCX) is a subclass of SFComboBox, our combobox base class located in SFCTRLS.VCX. Just to refresh your memory, SFComboBox has a custom aItems array property that's used as the RowSource for the items in the combobox (RowSourceType

is set to 5-Array), so we'll be populating that array with the values we want displayed. I added the new properties shown in Table 2. I also set the BoundColumn property to 2 because we want the first column of aItems to be the display value for each record and the second column to be the primary key of the record.

Table 2. SFLookupComboBox properties.

Property	Purpose
cIDField	The name of the field in the lookup table containing the primary key (default = ID).
cLookupAlias	The alias of the lookup table.
cLookupForm	The name of the form used to maintain the lookup table; can be left blank if it's the same name as the lookup table.
cNameField	The name of the field in the lookup table containing the name or description (default = NAME).

The Init() method calls This.Requery() to populate the combobox. Why not populate the combobox in Init() rather than calling Requery() (which we'll see in a second)? We'll see later that we'll want to update the values in the combobox when the user edits them, so we'll need to repopulate the aItems array, and we don't want to have to write the same code twice.

The Requery() method is where the interesting stuff is located. It does a SQL SELECT from the alias defined in cLookupAlias into the aItems array, grabbing the field specified in cNameField for the first column and the field specified in cIDField for the second column. Notice what it puts into the third column: an upper-case version of the first column. This isn't displayed or used for anything other than for the ORDER BY clause, which makes the combobox display values in a case-insensitive order. Also notice that this class can be used with lookup tables that don't have TABLE and FIELD columns; it selects only the values for the table and field bound to the combobox if a common lookup table is used, or all records if multiple lookup tables are used.

```
local lnPos, ;
  lcTable, ;
  lcField, ;
  lcName, ;
  lcID
with This
  lnPos = at('.', .ControlSource)
  if lnPos = 0 or type(.cLookupAlias + '.TABLE') = 'U'
    store '' to lcTable, lcField
  else
    lcTable = upper(padr(left(.ControlSource, ;
      lnPos - 1), fsize('TABLE', .cLookupAlias)))
    lcField = upper(padr(substr(.ControlSource, ;
      lnPos + 1), fsize('FIELD', .cLookupAlias)))
  endif lnPos = 0 ...
  lcName = .cNameField
  lcID   = .cIDField
  if empty(lcTable)
    select &lcName, ;
        &lcID, ;
        upper(&lcName) ;
      from (.cLookupAlias) ;
      order by 3 ;
      into array .aItems
  else
    select &lcName, ;
        &lcID, ;
        upper(&lcName) ;
      from (.cLookupAlias) ;
      where upper(TABLE + FIELD) = lcTable + lcField ;
      order by 3 ;
      into array .aItems
  endif empty(lcTable)
endwith
```

Since BoundColumn is 2, the combobox will display the description of the lookup values but will be bound to the primary key of each lookup record.

The ShortcutMenu() method is called from the ShowMenu() method, which is itself called from the RightClick() method (this scheme is used in all base classes in SFCTRLS.VCX). When the user right-clicks on the combobox, the ShowMenu() method defines a shortcut menu and passes its name to the ShortcutMenu() method, which populates the menu, after which ShowMenu() then displays the menu and handles the user's selection. The reason I do it this way rather than calling an MPR (shortcut menu program) is because I want the ability to add additional items to the menu in subclasses (the base ShortcutMenu() behavior puts items for Cut, Copy, Paste, Clear, and Select All into the menu). SFLookupComboBox's ShortcutMenu() method adds a bar so the user can edit the list of items in the combobox. Here's the code:

```
lparameters tcMenuName
dodefault(tcMenuName)
lnBars = cntbar(tcMenuName)
define bar lnBars + 1 of (tcMenuName) ;
  prompt '\-'
define bar lnBars + 2 of (tcMenuName) ;
  prompt '\<Edit List'
on selection bar lnBars + 2 of (tcMenuName) ;
  loObject.EditList()
```

In case you're wondering, loObject is defined in ShowMenu() as an object reference to the combobox itself. We can't use code like This.EditList() in menus, so I define loObject as a private variable that contains a reference to the combobox, so menu selections can call methods of the combobox.

The EditList() method is called when the user chooses "Edit List" from the shortcut menu for the combobox. This method calls the lookup maintenance form whose name is defined in the cLookupForm property; if cLookupForm is empty, this method assumes a form of the same name as the lookup table exists. My application class (SFApplication) has an OpenForm method for forms management, so it's called if oApp exists; if not, the form is run directly with DO FORM. In either case, the field this instance of the combobox is bound to is passed to the form as separate field and alias names, along with an object reference to the combobox.

```
local lcTable, ;
  lcField, ;
  lcForm
with This
  lnPos = at('.', .ControlSource)
  if lnPos = 0
    store '' to lcTable, lcField
  else
    lcTable = left(.ControlSource, lnPos - 1)
    lcField = substr(.ControlSource, lnPos + 1)
  endif lnPos = 0
  lcForm = iif(empty(.cLookupForm), .cLookupAlias, ;
    .cLookupForm)
  if type('oApp') = 'O' and not isnull(oApp)
    oApp.OpenForm(lcForm, lcTable, lcField, This)
  else
    do form (lcForm) with lcTable, lcField, This
  endif type('oApp') = 'O' ...
endwith
```

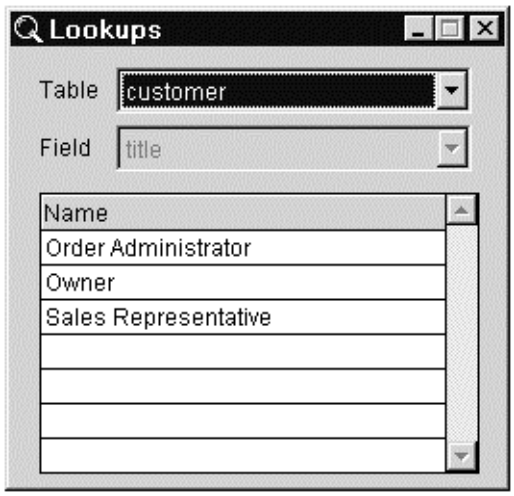
SFLookupComboBox can be used by simply dropping it on a form, binding it to a field in a table, and setting a few properties (mainly cLookupAlias, but possibly others if the structure of the lookup table is different than the LOOKUPS.DBF I discussed earlier). However, it might be wise to create subclasses of SFLookupComboBox with these properties filled in, especially if you use separate lookup tables rather than one combined one.

Lookup Maintenance Form

The lookup maintenance form is LOOKUPS.SCX. It's based on SFMaintForm, the table maintenance form class defined in SFFORMS.VCX. There are three controls on the form (not counting labels): a combobox of tables that use lookup values, a second combobox of fields that use lookup values from the table selected in

the first combobox, and a grid showing the list of lookup values for the selected table and field. The form is shown in Figure 1.

Figure 1. LOOKUPS.SCX, the lookup maintenance form.



The aItems array of the tables combobox is populated with unique table names from the LOOKUPS table using the following code in its Init() method:

```
select TABLE ;
  from LOOKUPS ;
  group by 1 ;
  into array This.aItems
```

Its AnyChange() method, which is called whenever an interactive or programmatic change is made to the value of the combobox, calls Thisform.SelectTable() to handle the table selection. That method populates the aItems array of the fields combobox with unique field names for the specified table from LOOKUPS. Thus, when a table is selected, the field combobox shows only fields from that table. If there's only one field from that table, the combobox is disabled.

```
local lnIndex, ;
  lcTable
with This.cboField
  lnIndex = This.cboTable.ListIndex
  if lnIndex > 0
    lcTable = This.cboTable.aItems[lnIndex]
    select FIELD ;
      from LOOKUPS ;
      where TABLE = lcTable ;
      group by 1 ;
      into array .aItems
    .Enabled = _tally > 1
    .Requery()
    .ListIndex = 1
  endif lnIndex > 0
endwith
```

The AnyChange() method of the fields combobox needs to display the list of values for the selected field in the selected table in the grid. It does this by calling a custom GetLookups() method of the form, and then refreshes the form.

```
with Thisform
  .GetLookups()
  .RefreshForm()
endwith
```

The grid has a parameterized view, LV_LOOKUPS_FOR_TABLE_FIELD, as its RecordSource. This simple view contains only those records from the LOOKUPS tables that match a specific table and field; its definition is as follows (see MAKEVIEW.PRG for which properties of the view are set to make the view updatable):

```
select *;
  from TESTDATA!LOOKUPS;
  where LOOKUPS.TABLE = ?vpTable and ;
    LOOKUPS.FIELD = ?vpField ;
  order by LOOKUPS.NAME
```

The vpTable and vpField parameters contain the desired table and field names, respectively. The GetLookups() method simply sets these parameters to the selected table and field and then requeries the view.

```
local lnTable, ;
  lnField, ;
  vpTable, ;
  vpField, ;
  lcAlias
with This
  lnTable = .cboTable.ListIndex
  lnField = .cboField.ListIndex
  if lnTable > 0 and lnField > 0
    vpTable = .cboTable.aItems[lnTable]
    vpField = .cboField.aItems[lnField]
    lcAlias = .grdLookups.RecordSource
    if cursorgetprop('SourceType', lcAlias) <> 3
      requery(lcAlias)
    endif cursorgetprop('SourceType', lcAlias) <> 3
  endif lnTable > 0 ...
endwith
```

If the lookup maintenance form is called from a menu item, we want the user to be able to select a table and field and then enter lookup values for that field. However, if the form is called from the EditList() method of an SFLookupComboBox, we don't want the user to see the tables and fields comboboxes and we want the grid to automatically show the lookup values for the field the SFLookupComboBox was bound to (Figure 2 shows what the form looks like under these conditions). That's handled by passing the table and field name to the Init() method of the form, along with an object reference to the SFLookupComboBox. The Init() method hides the tables and fields combobox and moves the grid up and resizes it to cover the empty space. It also sets the ListIndex of hidden comboboxes to the desired table and field, which causes the AnyChange() method of each to fire, which does the tasks outlined above to ensure the grid displays the proper records (isn't it fun how all this stuff ties together when you do things right? <g>). Here's the code for the Init() method:

```
lparameters tcTable, ;
  tcField, ;
  toCaller
local lnTop, ;
  lnIndex
dodefault()
with This
  if empty(tcTable)
    .cboTable.ListIndex = 1
  else
    store .F. to .cboTable.Visible, ;
      .lblTable.Visible, .cboField.Visible, ;
      .lblField.Visible
    lnTop = .grdLookups.Top
    .grdLookups.Top    = .cboTable.Top
    .grdLookups.Height = .grdLookups.Height + ;
      lnTop - .grdLookups.Top
    lnIndex = ascan(.cboTable.aItems, tcTable)
```

```
    .cboTable.ListIndex = lnIndex
    lnIndex = ascan(.cboField.aItems, tcField)
    .cboField.ListIndex = lnIndex
    .Caption = 'Edit ' + ;
      proper(alltrim(.cboField.aItems[lnIndex])) + ;
      ' List'
    .oCaller = toCaller
  endif empty(tcTable)
endwith
```

Figure 2. The lookup maintenance form when called with a table and field name.

Since the lookup maintenance form is modeless, there's no return value to send back to the SFLookupComboBox telling it that the user added, edited, or deleted lookup values. That's why we passed an object reference to the SFLookupComboBox to the form, which is stored in the oCaller property; the Destroy() method of the form tells the SFLookupComboBox to requery itself. Here's the code:

```
with This
  if type('.oCaller') = 'O' and ;
    not isnull(.oCaller) and ;
    pemstatus(.oCaller, 'Requery', 5)
    .oCaller.Requery()
  endif type('.oCaller') = 'O' ...
endwith
```

The only other custom code in LOOKUPS.SCX is the AddRecord method, which is called when a new record is added. We need to override the normal behavior of adding a blank record because we want the ID, TABLE, and FIELD fields filled in with the next available ID value (NEXTID.PRG will do that for us) and the values of the table and field comboboxes.

```
with Thisform
  lcAlias = .grdLookups.RecordSource
  insert into (lcAlias) ;
      (ID, ;
      Table, ;
      Field) ;
    values ;
      (NextID('LOOKUPS'), ;
      .cboTable.aItems[.cboTable.ListIndex], ;
      .cboField.aItems[.cboField.ListIndex])
  .RefreshForm()
endwith
```

Putting it All Together

As an example of how SFLookupComboBox is used, look at the CUSTOMER.SCX sample form available from the Subscriber Downloads site. This form, which is based on SFMaintForm, has no custom code in it. It simply has the CUSTOMER and LOOKUPS tables included in its DataEnvironment, has textboxes bound to the COMPANY and CONTACT fields, and an SFLookupComboBox bound to TITLE. The only properties set for the combobox are Width and cLookupAlias, which is set to LOOKUPS. Run this form and notice that the title for each contact shows correctly as you move through the records using the simple toolbar that appears. You can choose another title from the combobox, or right-click on the combobox and bring up the lookup maintenance form by choosing Edit List. Add a few new titles, then close the maintenance form and notice the new titles are available in the combobox.

Conclusion

This month, we looked at a generic combobox class for choosing lookup values and a form for maintaining the lookup values in an application. Next month, we'll examine a container class consisting of a textbox into which a lookup code is entered and a button to display a picklist of values. As usual, both the container and picklist classes will be reusable generic classes.

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He is the author of Stonefield's add-on tools for FoxPro developers, including Stonefield Database Toolkit and Stonefield Query. He is also the author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's "The Pros Talk Visual FoxPro" series. Doug has spoken at the 1997 and 1998 Microsoft FoxPro Developers Conferences (DevCon) as well as user groups and regional conferences all over North America. He is a Microsoft Most Valuable Professional (MVP). CompuServe 75156,2326 or dhennig@stonefield.com.