February, 1997

Advisor Answers

Q: I haven't found a good way of subclassing grid columns. I tried defining my main grid class with a pair of columns, customizing the header and text controls inside. The problem is that when I add new columns all the PEMs default to the VFP standards.

Is there some silly way to do this that I'm missing, or even a hard one? In any case, I'd appreciate your help.

–Martin Salias (via the Internet)

A: In fact, subclassing grid columns (and pages in a pageframe and option buttons in an option group) is difficult. The Class Designer doesn't let you create subclasses of these base classes and there's no way to specify that a particular subclass should be used.

There are a couple of solutions, however. They differ mostly in how far you go towards subclassing. One solution is to do the work in the code of your grid class. In the grid's Init method, use its SetAll method to set column properties the way you want them. For example:

```
* Show controls in all rows
THIS.SetAll("Sparse",.F.,"Column")
* Ledger-style coloring
THIS.SetAll("DynamicBackColor", ;
   "IIF(MOD(RECNO(),2)=0,RGB(0,255,0),RGB(255,255,255))")
* And so forth
```

This approach, while simple, has some serious limitations. First, you can only change the column properties, not the methods. Second, if you want to add columns dynamically at run-time, you have to set all the same properties again.

An alternate approach is to create a column subclass in code. Although the Class Designer doesn't allow columns to be subclassed, you can do it using the DEFINE CLASS command. (This applies to most of the classes that can't be subclassed visually - I like to think of them as "half-classed.") For example, the following code creates a column with the same characteristics set above.

```
DEFINE CLASS colColumn AS Column

   Sparse = .F.
   DynamicBackColor = ;
      "IIF(MOD(RECNO(),2=0,RGB(0,255,0),RGB(255,255,255))"

ENDDEFINE
```

Once you define a column class, how do you get your grids to use it? You have to forget about creating grid columns at design-time and instead set them up on the fly at runtime. Set ColumnCount to 0. In the Grid's Init method (or any other), use the AddObject method to add columns based on your subclass. The minimal set of properties you need to set to get your column working is Visible (to .T.) and

ControlSource (to link it to a field of the source table.) If you need to add columns dynamically, again use AddObject.

Here's code that can go in the grid's Init. It adds a column for each field of the table, setting the header caption to the field's stored caption if it has one and to the field name, if not. It assumes you've already SET PROCEDURE TO the column class definition. This code also assumes you've set the grid's RecordSource in the property sheet or in some piece of code that executes before the grid's Init.

```
LOCAL nFieldCount,cNewColumn,nCnt,nOldArea,cCaption
LOCAL cDatabase, cOldData

nOldArea = SELECT()

* Select source of grid
SELECT (THIS.RecordSource)
nFieldCount = AFIELDS(aFieldList)

* Check if free or contained table and get database name
cDataBase = CURSORGETPROP("Database")
IF NOT EMPTY(cDatabase)
   cOldData=SET("DATABASE")
   SET DATABASE TO (cDatabase)
ENDIF

FOR nCnt = 1 TO nFieldCount
   * Figure out name for new column - since this
   * is Init, we don't need to worry about a column
   * name already being in use.
   cNewColumn = "Column"+ ;
      LTRIM(STR(THIS.ColumnCount+1))

   * Add the column and set properties
   THIS.AddObject(cNewColumn,"ColColumn")
   WITH THIS.Columns[THIS.ColumnCount]
      .Visible=.T.
      .ControlSource = FIELD(nCnt)
      * Get Caption from database
      cCaption = ""
      IF NOT EMPTY(cDatabase)
         cCaption = DBGETPROP(ALIAS()+"."+FIELD(nCnt), ;
            "FIELD","Caption")
      ENDIF
      IF EMPTY(cCaption)
         cCaption = PROPER(FIELD(nCnt))
      ENDIF
      .Header1.Caption = cCaption
   ENDWITH
ENDFOR

SELECT (nOldArea)
IF NOT EMPTY(cOldData)
   SET DATABASE TO (cOldData)
ENDIF
```

This month's Companion Resource Disk contains a form (GridFill.SCX) that uses this technique to populate a grid with custom columns linked to the TasTrade Customer table.

Another approach to adding columns on the fly is to redefine the grid's AddColumn method. By default, AddColumn adds a new column of the base column class. But a little code in this method can change its behavior so that it adds a column from your custom class instead. The code is pretty much the same as you'd use elsewhere, except you also need the NODEFAULT command to tell AddColumn not to do its usual thing. Here's code that adds the next column in sequence:

```
LPARAMETERS nIndex
LOCAL nColIndex, cNewColumn

* Turn off default behavior of AddColumn
NODEFAULT

* Get a unique name for the column
nColIndex  = THIS.ColumnCount+1
cNewColumn = "Column"+LTRIM(STR(nColIndex))
DO WHILE TYPE("THIS."+cNewColumn)<>"U"
   nColIndex = nColIndex+1
   cNewColumn = "Column"+LTRIM(STR(nColIndex))
ENDDO

THIS.AddObject(cNewColumn,"ColColumn")
THIS.Columns[THIS.ColumnCount].Visible=.T.
```

Since AddColumn doesn't normally do any more than add the new column and make it visible, that's all this code does. After calling it, you can set up the new column as you wish - at a minimum, ControlSource needs to point to the field to be displayed.

The code above assumes that the custom column class is called "ColColumn" - a generic approach would be better. We can add custom properties to the grid class to specify the name of the column class and the class library. In the grid's Init, we SET PROCEDURE to the class library, as follows:

```
* The custom grid property cClassLib contains the name of
* the .PRG file containing our column class
SET PROCEDURE TO (THIS.cClassLib) ADDITIVE
```

AddObject expects a parameter to indicate where the new column should be added in the grid - in the code above, we ignored it. It's fairly simple to honor the parameter, since it affects only ColumnOrder, not the actual order of the columns in the Columns collection. Here's the updated code for AddColumn to add a custom column class:

```
LPARAMETERS nIndex
LOCAL nColIndex, cNewColumn

* Don't add a base column object. This line can go
* anywhere in this method - it's at the top so it's
* obvious from the beginning that we're replacing,
* not supplementing the base behavior.
NODEFAULT

* Figure out name for new column. We can't assume
* that the next column number is available.
nColIndex  = THIS.ColumnCount+1
cNewColumn = "Column"+LTRIM(STR(nColIndex))
DO WHILE TYPE("THIS."+cNewColumn)<>"U"
   nColIndex = nColIndex+1
```

```
   cNewColumn = "Column"+LTRIM(STR(nColIndex))
ENDDO

* Add the new column and set properties.
* cColumnClass is a grid property containing the name of
* the custom column class.
THIS.AddObject(cNewColumn,THIS.cColumnClass)
WITH THIS.Columns[THIS.ColumnCount]
   .Visible=.T.
   * If position is specified, update it. Otherwise leave it
   * at the default
   IF TYPE("nIndex")="N"
      .ColumnOrder = nIndex
   ENDIF
ENDWITH
```

With this code in AddColumn, adding a new column is no different than in any other grid—just call AddColumn, like this:

```
THIS.AddColumn()
```

Then, set the properties of the new column. For example:

```
THIS.Columns[THIS.ColumnCount].Header1.Caption = "Company"
THIS.Columns[THIS.ColumnCount].ControlSource = ;
   "Customer.Company"
```

Unfortunately, changing a grid's ColumnCount doesn't seem to call on the grid's AddColumn method - it still adds a column from the column base class. So, it's necessary to explicitly call AddColumn. This month's Companion Resource Disk contains a class called grdAddCol which implements this technique in the AddColumn method. ColumnCount for the grid is set to 0.

Similar approaches to using custom classes for contained objects work with the other container objects - you can build pageframes up out of custom pages, option groups from custom option buttons and so forth (though none of the others have a specialized method like AddColumn). Some of them even allow you to do the AddObject at design-time, which means it's possible to write a builder to set up the container with the contents you want at design-time.

–Tamar