Using context sensitive popmenus to enter values in a SAS/AF Data Table object

Richard A. DeVenezia

Abstract

The SAS/AF Data Table object is a stock component that can be used as a foundational part of a SAS/AF Frame application. The Data Table has many built-in features, but sometimes the defaults are not enough to make an interface truly simple for the end-user. Extending the Data Table to allow new and useful interaction can make an application well received by the user community. Additionally, a developer can reuse the extended object or object extending techniques in other applications.

The use of the right mouse button for displaying a context sensitive menu of choices is a well established convention. This paper describes a technique that overrides a SAS/AF Data Table object's _POPUP_ method to allow column specific context sensitive popmenu data entry. From a user's perspective, when a cell is right clicked a list of allowed values is displayed and the selected value is placed in the cell.

A Sample Scenario

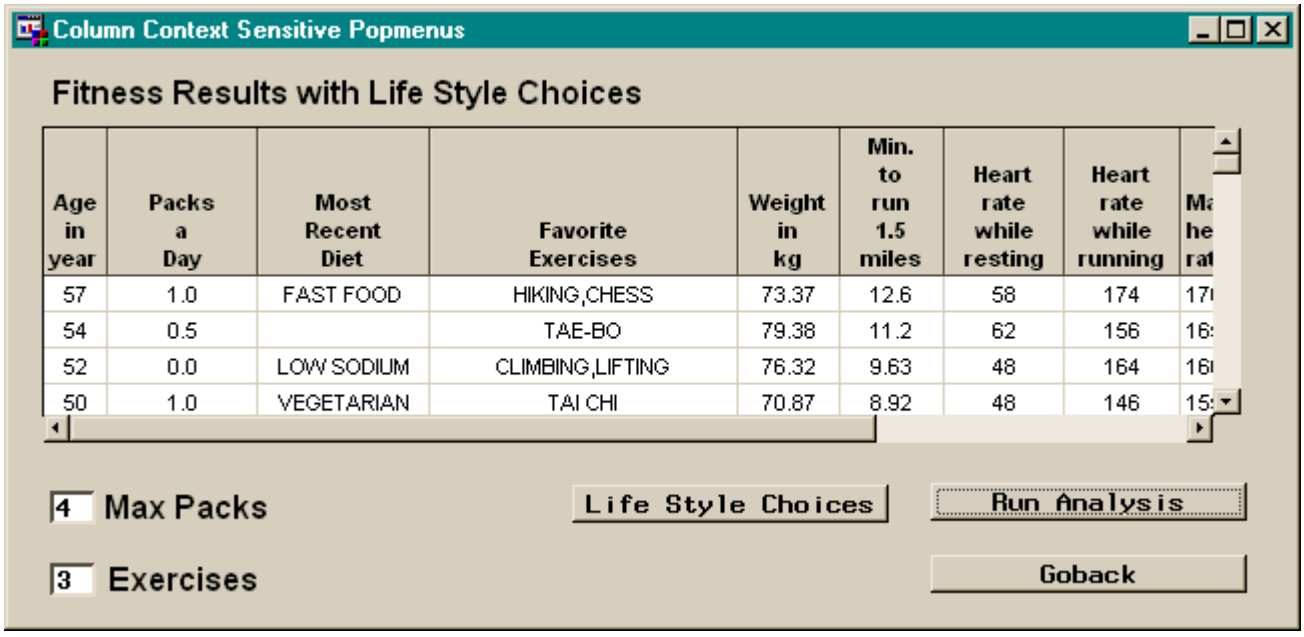
An analyst is doing research on the impact of lifestyle choices, such as smoking and diet, on fitness. She added three columns to the data set SASUSER.FITNESS to record information about her group of subjects. The SMOKES column is for the number of packs smoked a day, the DIET column is for the diet followed at time of fitness measures and EXERCISE is a comma-separated list of types of regular exercises performed.

The allowed values for SMOKES range from zero to the value entered for *Max Packs*. The allowed values for the DIET and EXERCISE variables are stored in a support data set named SASUSER.LIFESTYL, which has columns ASPECT and VALUE. The values of ASPECT correspond to the column names DIET and EXERCISE.

The SASUSER.LIFESTYL data set is displayed when the *Life Style Choices* button is pushed in and SASUSER.FITNESS when the button is pushed out.

The analyst's program is submitted from the Run Analysis button.

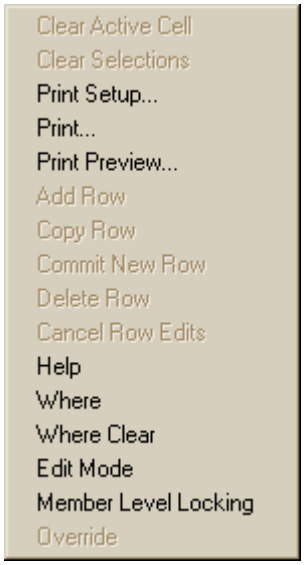
Sample Front End Layout



Default Popmenu

The default popmenu, which appears when the right mouse button is clicked, is designed for a generic form of table editing. There is no knowledge of the life style column contexts.

A context sensitive popmenu will provide selections more relevant to the analyst's task of recording data.



Desired Contexts

When the right mouse button is clicked over a cell in the *Packs a Day* column, a list of values from zero to N by .5 increments should appear. The input field *Max Packs* controls N.

When the right mouse button is clicked over a cell in the *Most Recent Diet* column, a list of different diets should appear.

When the right mouse button is clicked over a cell in the *Favorite Exercises* column, a popmenu of different exercises should appear. Each exercise listed should have the word *add* or *remove* before it. A selected exercise would then be either added to or remove from the comma-separated list of exercise values. Only N exercises are allowed in the comma-separated list of exercises. N is controlled by the input field *Exercises*.

Multiple Row Functionality

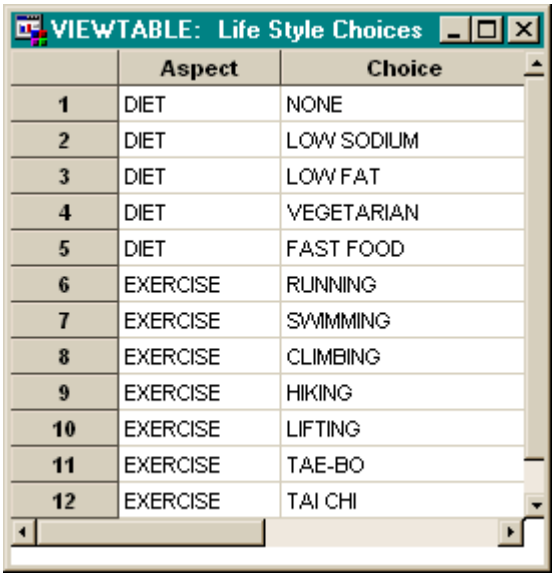
A quote from SAS system help at index 'table editor,selection features':

> "The table editor provides features that enable users to select one or more rows or columns in the table. Selection features determine whether multiple selected rows can be noncontiguous, called multiple selections, or contiguous, called an extended selection.
>
> You enable selection features by setting attributes for the table. In addition to determining whether multiple rows or columns can be selected, other attributes determine whether a row can be selected by a click on any of its columns or a row or column can be selected by a click on its label.
>
> Copyright (c) 1996, SAS Institute Inc., Cary, NC 27513-2414 USA. All rights reserved.

When the right mouse button is clicked over a cell in a selection and a choice is made from a context popmenu, that choice should be placed in each highlighted cell in the column. This functionality will not be provided to the *Favorite Exercises* column.

Support Dataset: SASUSER.LIFESTYL

This dataset is maintained as needed to modify available choices for a context:

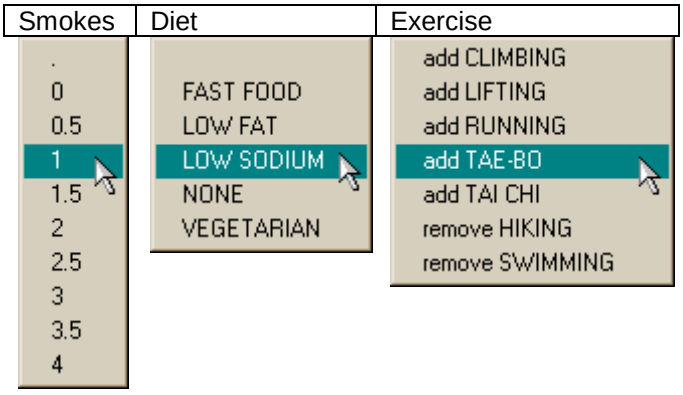


	Aspect	Choice
1	DIET	NONE
2	DIET	LOW SODIUM
3	DIET	LOW FAT
4	DIET	VEGETARIAN
5	DIET	FAST FOOD
6	EXERCISE	RUNNING
7	EXERCISE	SWIMMING
8	EXERCISE	CLIMBING
9	EXERCISE	HIKING
10	EXERCISE	LIFTING
11	EXERCISE	TAE-BO
12	EXERCISE	TAI CHI

The table can be edited 'in frame' by pushing the *Life Style Choices* button in.

Desired Popmenus

Based on the desired contexts and support dataset, the popmenus should appear as follows:



Making it Happen

Getting the front end built and determining the user interface behavior is only half the battle. A reason-

able context handling implementation must be found amid the myriad possibilities available in SAS/AF.

Some questions that arise are:

Q: Should I use data table model SCL?
A: Model SCL is better suited for context sensitive rendering of a value (i.e. foreground and background colors, font, weight and style.) For context popmenus, a label for each column being handled, as well as a check of the _STATUS_ variable, would have to be put in the model SCL. This is not very extensible.

Q: Should I subclass the data table object?
A: Subclass the data table if you wish to reuse features in other frames (or even the same frame) How to sub-class is not covered in this paper.

Q: What events, if any, should be processed in the frames SCL?
A: Frame SCL can be used to process left mouse button events on the data table. These types of events are generally related to selecting a region or the first part of a 'drag and drop.' Doing generic popmenu processing in a frame's SCL is not conducive to extensibility.

Q: Which of the 269 data table methods should I use?
A: Use what ever it takes to get the job done.

Q: Did I call Tech Support too many times today?
A: No ☺

Core Concepts

❶ Override the data table _POPUP_ method. This can be considered as installing an event handler.

❷ Attach items of information to the data table object for use by the _POPUP_ override. (These are *per instance* instance variables)

❸ The type of item attached can be numeric or character.

For numeric items, the _POPUP_ override will assume the value is an *SCL list id* which should be popmenu'ed.

For character items, the _POPUP_ override will assume the value is a *method name*, which can be called. The method name value must have the format *[lib . cat .] entry [. type] : label.* This value is used in a call method statement.

The label's method statement **must** conform to:

```
method _self_ 8 row 8 column_name $32 _pop 8;
```

*_self_* is an input variable whose value is the data table object list id.

row is an input variable whose value is the row number where the popup event occurred. *column_name* is an input variable whose value is the name of the column where the popup event occurred. *_pop* is an input variable whose value is a valid SCL list id. The list is allocated for the method by the _POPUP_ override method. The list should have choice items inserted into it for selection by the user.

❹ The *name* of an item attached to the data table object is of the form **RMB_LIST_FOR_<column_name>**

A properly named and valued item attached to the data table object can be considered as installing an event context handler. The context handler is dispatched by the event handler.

Per instance overrides of data table methods are installed at run time in the frame's INIT section.

Frame Implementation Details

The front end contains these objects:
- Data table named *fitness*.
- Input fields named *maxPacks* and *nExer*.
- Command push buttons named *lifeStyl, run* and *end.*

❶ Override *fitness'* **_POPUP_** method in the INIT section.

```
INIT:
call send(_frame_,'_get_widget_','fitness',tabid);
call send(tabid,'_SET_INSTANCE_METHOD_','_POPUP_',
         'SASUSER.NESUG99.CONTEXT.SCL','POPUPCEL');
```

❷ Attach items of information to the data table.

Column	Attach item of this type
Smokes	Numeric (SCL list)
Diet	Numeric (SCL list)
Exercise	Character (Method name)

The SCL lists need to be created and populated before attaching their list ids to the data table object.

```
INIT:
Link MaxPacks; * creates SCL list _smokes;
Link Diet; * creates SCL list _diet;
* install context handlers;
rc=setnitemn(tabid,_smokes,'RMB_LIST_FOR_SMOKES'
);
rc=setnitemn(tabid,_diet,  'RMB_LIST_FOR_DIET');
rc=setnitemc(tabid,'CONTEXT:EXERCISE',
                        'RMB_LIST_FOR_EXERCISE');
MAXPACKS:
```

```
* if input field is modified this section runs;
* repopulating the list of allowed values;
* clearlist is used because _smokes list id ;
* has been 'installed'.;
if _smokes
    then rc = clearlist (_smokes);
    else _smokes = makelist ();
  rc = insertc (_smokes, '.');
  do i = 0 to max (0, MAXPACKS) by 0.5;
    rc=insertc(_smokes,left(put(i,best4.)),-1);
  end;
return;
```

```
DIET:
* this section is only run once from INIT;
  _diet = makelist ();
  __lifestyl = open
('SASUSER.LIFESTYL(WHERE=(ASPECT="DIET"))');
  if __lifestyl then do;
    n = 0;
    rc=lvarlevel(__lifestyl,'CHOICE',n,_diet);
    __lifestyl = close (__lifestyl);
  end;
  rc = sortlist (_diet);
  rc = insertc (_diet, ' ');
return;
```

POPUP Override Method Details

The override method has to do several things:

- determine cell clicked over
- determine if the column has context handling.
- dispatch the handler
- place a value in the cell (or selection) if a popmenu choice is made.

A cell or selection location within a data table is described in terms of coordinate lists. These SCL lists are documented in the online help at index 'coordinate lists':

> All of the data table viewer methods that operate on a particular row, column, or cell have coordinate arguments. These arguments are SCL lists rather than simple numerics in order to deal with multi-dimensional data, that is, nested rows or columns. The Data Table class does not support multidimensional data. … For the data table, these coordinate lists contain a single integer value which represents a row or column number in the data table.

index '_get_selections_' further elucidates:

> The selections returned by the _GET_SELECTIONS_ method are grouped into individual sublists, one for each selection. Each of the sublists within the outer list contains four named lists, START_ROW, START_COLUMN, END_ROW, and END_COLUMN. These named lists contain the coordinates for the particular point.

When an entire column or row is selected the END_ROW or END_COLUMN lists will contain the value -1.

```
/*****************************************/
/* POPUPCEL: Override the data table _POPUP_ */
/* method,providing column based RMB contexts*/
/*****************************************/

length _method_ colname $40 coltype $1;

POPUPCEL:
  method plist sel 8;

* make lists to contain column coordinate lists;
_poprow = makelist ();
_popcol = makelist ();

* obtain the row and column coordinates of
* the cell where  the last pop-up occured ;
call send(_self_, '_get_popup_cell_' ,
                                  _poprow, _popcol);

* safety check: if coordinate lists are ;
* incomplete then return;
if 0=nameditem (_poprow,'1')
or 0=nameditem (_popcol,'1') then goto byebye;

* get column name cell is in;
row = getnitemn (_poprow, '1');
col = getnitemn (_popcol, '1');

call send(_self_, '_GET_DISPLAYED_COLUMN_NAME_',
                                     col, colname);

* check for column context handler;
handlerItem = nameditem (_self_, 'RMB_LIST_FOR_'
                                        || colname);
if handlerItem = 0 then do;
  * uncomment call super to allow the default
  * method to run for unhandled columns;
  * call super (_self_, _method_, plist, sel);
  goto byebye;
end;

* set active cell to cell popped over to
* ensure user knows where popup is coming from;
call send(_self_, '_set_active_cell_',
                                  _poprow, _popcol);

if row > 0 then
  link dispatch;
else do;
* This is where special handling for popups on;
* column name could occur (i.e. sorting);
* link disptch0;
end;
```

```
byebye:

_poprow = dellist (_poprow);
_popcol = dellist (_popcol);

endmethod;

*-----------------------------------------;
dispatch:
*-----------------------------------------;
* numeric handler items are list ids;
* callers are responsible for ensuring pop list
items are character types;
* character handler items are method names;
* callers are responsible for ensuring method
names are in A.B.C.D:E format;

* determine data set column type, for use in
context handlers;
call send(_self_, '_GET_COLUMN_ATTRIBUTE_',
                      colname, 'TYPE', coltype);

* dispatch based on context handler item type;
select itemtype (_self_, handlerItem);
  when ('N') link popItemN;
  when ('C') link popItemC;
  otherwise;
end;
return;
```

Numeric valued (i.e. SCLI list id) context handlers dispatched by the _POPUP_ event handler are very easy to implement. The list has been setup for us and we only have to check for valid list id's and if a popmenu choice was made.

```
*-----------------------------------------;
popItemN:

* get scl list id, return if an invalid list id;
_pop = getitemn (_self_, handlerItem);
if _pop > 0 and listlen (_pop) = -1 then return;

* popup the list, return if no choice is made;
choice = popmenu (_pop);
if choice > 0 link setCell;
return;
```

Character valued (i.e. method name) context handlers dispatched by the _POPUP_ event handler can be used in many creative ways. The stock behavior of the sample scenario is implemented below. The context method is called assuming the method will populate an SCL list which popItemC will display and handle. A provision is made to not do anything after the context method is called. No post context method handling will occur if the context method passes back an empty list or invalid list id.

Really interesting things can be done in the context method since the method is called passing in all the state information that is available to the event handler. It has enough information to pop a context menu and place values into the table itself.

```
*-----------------------------------------;
popItemC:
* character valued handler items are assumed to
be method names of format ;
* [ lib . cat . ] entry [ . type ] : label ;
* no check is made on the existance of the
method;

length popcol_method $200;

* get context method, return if ill-formed;
popcol_method = getitemc (_self_,handlerItem);
if 0 = index (popcol_method, ':') then return;

* make a list for the method to use, and keep;
* a backup of the list id, just in case the;
* method returns a different list id;
_pop = makelist ();
_pophold = _pop;

* call the method;
call method (scan (popcol_method,1,':'),
             scan (popcol_method,2,':'),
             _self_, row, colname, _pop);

* popup the list if populated by the method;
if listlen (_pop) > 0 then do;
  choice = popmenu (_pop);
  if choice > 0 then link setCell;
end;

* delete the SCL list passed to context method;
if _pophold ne _pop then
  _pophold = dellist (_pophold);
* delete list context method may have made;
if listlen (_pop) ne -1 then
 _pop = dellist (_pop);
return;
```

The setCell section implements the default behavior for which cell(s) receive the value selected by the user.

```
*-----------------------------------------;
setCell:
* put chosen value in pop cell or in each cell;
* in pop cell column in selection;

_select = makelist ();
call send (_self_, '_GET_SELECTIONS_', _select);
```

```
popped_in_selection = 0;

* this loop will only iterate if selections made;

do i = 1 to listlen (_select);
  _rect = getiteml (_select, i);
  sr=getnitemn(getniteml(_rect,'START_ROW'),'1');
  sc=getnitemn(getniteml(_rect,'START_COLUMN'),'1');
  er=getnitemn(getniteml(_rect,'END_ROW'),'1');
  ec=getnitemn(getniteml(_rect,'END_COLUMN'),'1');

  * when pop cell is in a region put chosen value;
  * in all cells in pop cell column in region;

  if  ((sc <= col <= ec) or (ec = -1))
  and ((sr <= row <= er) or (er = -1))
  then do;
    popped_in_selection = 1;

    if sr = 0 then sr = 1; * ignore head row;

    if er = -1 then do;
      * entire column was selected,
      * set er to number of rows in dataset;
      _attr = makelist ();
      rc = setnitemn(_attr,-1,'NUMBER_OF_ROWS');
      call send (_self_,
             '_GET_DATASET_ATTRIBUTES_', _attr);
      er = getitemn (_attr, 1);
      _attr = dellist (_attr);
    end;

    * put chosen value in each cell in pop cell;
    * column in region ;
    do row = sr to er;
      call send (_self_, '_LOCK_ROW_', row);
      select coltype;
        when ('C') link setText;
        when ('N') link setValue;
        otherwise;
      end;
    end;

    call send (_self_, '_clear_select_');
    call send (_self_, '_set_active_cell_',
                              _poprow, _popcol);

  end; /* pop cell in selection region */
end; /* selection loop */

if NOT popped_in_selection then do;
  * put chosen value in popup cell;
  call send(_self_, '_LOCK_ROW_', row);
  select coltype;
    when ('C') link setText;
    when ('N') link setValue;
    otherwise;
  end;
end;

call send(_self_, '_UNLOCK_ROW_');
_select = dellist (_select, 'Y');
return;

*-------------------------------------------;
setText:
call send(_self_, '_SET_COLUMN_TEXT_', colname,
                        getitemc (_pop, choice));
return;
*-------------------------------------------;
setValue:
call send(_self_, '_SET_COLUMN_VALUE_', colname,
          input(getitemc(_pop,choice),best12.));
return;
```

Exercise Context Handler Method

This method is called by the POPUPCEL method dispatching the EXERCISE column handler setup in the frame's INIT section.

The implementation demonstrates some techniques to tightly couple the context with the content of other objects on the frame. In the sample scenario, the object nExer controls the number of comma separate values allowed in a cell.

```
*-------------------------------------------;
* This method is called due to a RMB action on ;
* a table which had its _POPUP_ method         ;
* overridden by POPUPCEL, and an instance      ;
* variable RMB_LIST_FOR_EXERCISE added whose   ;
* value is this method name                    ;
* The frame containing the data table, which   ;
* had its POPUP method overwritten and had the ;
* instance variable RMB_LIST_FOR_<column> added;
* with value CONTEXT:EXERCISE, must also have  ;
* an SCL variable called NEXER                 ;

length exercise $50 anExercise $50;

EXERCISE:
method _self_ 8 row 8 column_name $32 _pop 8;

* this method will CRASH if there is no SCL;
* variable named NEXER in the frame containing;
* the data table,there is no simple way (if any)
* to determine what variables are available in a
* frames scope at run time;

* get number of comma separated exercises that
* are allowed in the EXERCISE column from the;
* variable NEXER in the data tables frame;
call send (getnitemn (_self_, '_FRAME_'),
               '_GET_NUM_VAR_', 'NEXER', nexer);

* get the current EXERCISE value in the popcell;
call send (_self_, '_GET_COLUMN_TEXT_',
```

```
                          column_name, exercise);
exercise = trim (exercise);

* determine if any more exercises can
* be concatenated;
allow_more = (scan(exercise,nexer, ',') eq '');

* separator for when concatenating;
if scan (exercise, 1, ',') ne ''
  then add_sep = ',';
  else add_sep = '';

* put data table exercises in SCL list for
* easy searching;
_dtExercises = makelist ();
i = 1;
do while (scan (exercise, i, ',') ne '');
  rc = insertc (_dtExercises,
                left(scan(exercise,i,',')), -1);
  i + 1;
end;

* put support table choices in an SCL list;
_lsExercises = makelist ();
__lifestyl = open
('SASUSER.LIFESTYL(WHERE=(ASPECT="EXERCISE"))');
if __lifestyl then do;
  n = 0;
  rc = lvarlevel (__lifestyl, 'CHOICE', n,
                                  _lsExercises);
  __lifestyl = close (__lifestyl);
end;
nExerInLifeStyle = listlen (_lsExercises);

* add choices to remove values in _dtExercises
* not found in _lsExercises;
do i = 1 to listlen (_dtExercises);
  anExercise = getitemc (_dtExercises, i);
  if searchC(_lsExercises,anExercise,1,1,'Y')=0
  then do;
      rc = insertc (_lsExercises, 'remove ' ||
                                anExercise, -1);
  end;
end;

* modify the _lsExercise choices to indicate
* which can be added or should be removed;
do i = 1 to nExerInLifeStyle;
  anExercise = trim(getitemc (_lsExercises, i));
  if searchC(_dtExercises,anExercise,1,1,'Y')=0
  then do;
    * the exercise is not in pop cell,
    * therefore it can be added;
    rc = setitemc (_lsExercises, 'add ' ||
                                 anExercise, i);
    if NOT allow_more then
      rc = setlattr (_lsExercises,'INACTIVE',i);
  end;
  else
    rc = setitemc (_lsExercises, 'remove ' ||
                                 anExercise, i);
  end;

* pop the choices;
rc = sortlist (_lsExercises);
choice = popmenu (_lsExercises);

* add to or remove from the pop cell CSV;
if choice > 0 then do;
  anExercise = getitemc (_lsExercises, choice);
  if anExercise =: 'add ' then do;
    exercise = exercise || add_sep ||
                           substr (anExercise,5);
  end;
  else do;
    * remove;
    anExercise = substr (anExercise, 8);
    remove = searchc (_dtExercises, anExercise);
    exercise = '';
    add_sep = '';
    do i = 1 to listlen (_dtExercises);
      if i ne remove then do;
        if add_sep = '' then do;
          exercise = getitemc (_dtExercises,i);
          add_sep = ',';
        end;
        else
          exercise = exercise || add_sep ||
                       getitemc (_dtExercises,i);
      end;
    end;
  end; /* remove */

  * place the new exercise value in the popcell;
  call send (_self_, '_LOCK_ROW_', row);
  call send (_self_, '_SET_COLUMN_TEXT_',
                           column_name, exercise);
  call send (_self_, '_UNLOCK_ROW_');
end;

* clean up;
_lsExercises  = dellist (_lsExercises);
_dtExercises = dellist (_dtExercises);

endmethod;
```

The method was able to place a value into the data table by itself. The method only 'needed' the event handler (POPUPCEL, the _POPUP_ override) to make sure it was called. The value of the *_pop* argument was left unchanged to inform POPUPCEL that it should do nothing more.

Back to the Frame

The 'in frame' editing of the LIFESTYL data set is accomplished by using a command push button.

The region attributes, button behavior, are set to check box. This means a mouse click pushes the button in and a second click is used to pop the button out.

```
*********************************************;
LIFESTYL:
length state $3;
call notify ('LIFESTYL', '_GET_BORDER_STATE_',
                                          state);
if state = 'ON' then
  call notify ('FITNESS', '_SET_DATASET_',
          'SASUSER.LIFESTYL', 'EDIT', 'MEMBER');
else do;
  call notify ('FITNESS', '_SET_DATASET_',
           'SASUSER.FITNESS', 'EDIT', 'RECORD');
  call notify ('FITNESS','_display_column_label_',
                                        '_all_');
end;
return;
```

Other Possibilities

Instead of adding one item per column, a more generic handler could be installed. Consider an item named 'GENERIC_HANDLER' of itemtype list, which contains items such as the method to call and the name of the dataset which contains the lookups. The lookup table would be more abstract than the sample LIFESTYL. It would have columns such as VARNAME, VARVALUE and VARTYPE. If the cell selected matches in regard to variable name and type, then the values would be popped.

Summary

A well thought out application is only half the project. Be assured SAS/AF provides all the tools necessary to implement your user interface design. The breadth and depth of SAS/AF can be daunting when trying to find very specific information about classes and method. SAS Institute Technical Support personnel are among the best and can find your needle in the haystack.

Recommended Reading

SAS Institute Inc., *SAS/AF Software: FRAME Entry, Version 6, First Edition,* Cary NC, SAS Institute Inc., 1993.

SAS Institute Inc., *SAS Screen Control Language, Version 6, Second Edition,* Cary NC, SAS Institute Inc., 1994.

SAS Institute Inc., *SAS/AF Software: FRAME Application Development Concepts, Version 6, First Edition,* Cary NC, SAS Institute Inc., 1995.

SAS Institute Inc., *SAS/AF Software: FRAME Class Dictionary,* Cary NC, SAS Institute Inc., 1995.

SAS Institute Inc., *Getting Started with the FRAME Entry, Second Edition: Developing Object-Oriented Applications,* Cary NC, SAS Institute Inc., 1997.

SAS Institute Inc., *Course Notes: Building SCL Applications Using FRAME Entries,* Cary NC, SAS Institute Inc., 1996.

SAS Institute Inc., *Course Notes: Application Development with The SAS System, Release 6.12,* Cary NC, SAS Institute Inc., 1996.

SAS Institute Inc., *Course Notes: Object-Oriented Programming Using SAS/AF Software,* Cary NC, SAS Institute Inc., 1996.

SAS Institute Inc., *Course Notes: Using SAS/AF Data Table and Data Form Objects,* Cary NC, SAS Institute Inc., 1996.

Acknowledgements

Author Information

Richard A. DeVenezia
9949 East Steuben Road
Remsen, NY 13438
radevenz@ix.netcom.com

The sample application is available in CPORT format upon request.