

Taking Advantage of Inheritance in SAS/AF® Applications

Carl R. Haske, Ph.D., STATPROBE, Inc., Ann Arbor, MI

ABSTRACT

SAS/AF supplies many predefined classes for use in application development. When developing applications with a high level of complexity, it is inevitable that the application will contain multiple objects with indistinguishable functionality. SAS/AF provides a technique to define new classes that inherit methods from a parent class along with the flexibility to override inherited methods and define new methods. This flexibility provides many advantages to the developer. Class libraries can be developed containing reusable objects that are utilized in many different types of applications. This leads to a higher degree of standardization in applications and frees the developer to concentrate on the higher level functions of an application.

This paper discusses techniques for deriving classes in SAS/AF. Details are provided on defining new methods and overriding methods, thereby extending class functionality. In addition, examples describe how to implement SCL programs that call inherited methods and new methods. Examples also demonstrate the concept of reusability and object-oriented programming methodology. These techniques allow the developer to rapidly prototype and complete software applications. Examples show how to make derived classes available for creating objects. In this paper, we will cover the following topics.

- Object-oriented programming with SAS/AF frames
- SAS/AF objects
- Deriving a subclass
- Overriding methods
- Defining a new method
- Design a custom attribute frame
- Resourcing a new class and creating an object
- An example application with a new class

OBJECT-ORIENTED PROGRAMMING WITH SAS/AF FRAMES

A SAS/AF application consists of a series of frames that interact with the user and perform actions based on user input. In an application, it is very common to use similar objects on different frames to perform similar actions. For example, an application might have the user enter a date field on several different frames. The object that collects the date field would be a special kind of text entry box that does special processing to make sure that a valid date has been entered. Rather than recreate this special object on each frame, it is better to define a class of date entry boxes and instantiate an object of this class whenever it is needed. During the design phase of an application, objects and their purposes are defined. Objects of similar purpose should be organized into classes and act as the basis for a class library.

Classes are a fundamental concept in object-oriented programming. Classes are a programming construct that embody both data and methods. The data contain information about the class and the methods are procedures to access the data, modify the data, and send messages to other objects. SAS/AF includes several classes, including buttons, icons, list boxes, radio boxes, and graphic displays, with which to build a user interface for applications. The data for these classes

are referred to as attributes and are stored in instance variables. Subclasses are derived from a parent class and they inherit all of the parent's data and methods. The methods of the parent class can be overridden, replaced, or supplemented for use by the sub-class. In addition, new attributes and methods can be defined for sub-classes.

The steps for developing a sub-class in SAS/AF are as follows.

- Derive the sub-class
- Identify new instance variables
- Identify new methods and methods to be overridden
- Program SCL entries for new methods and overridden methods
- Design a custom attribute window to initialize instance variables
- Add the new class to the SAS/AF resource list

We will illustrate each of these steps by systematically developing a subclass of the list box class. First we will develop a list box class that lists data set labels for all data sets in a data library. This subclass will be modified further so that the data library to be listed can be dynamically assigned by an application. Finally, the subclass will be modified so that an application can access the name of the data set corresponding to the selection on a list box object. This behavior will be exhibited in a simple application that uses the new subclass.

SAS/AF OBJECTS

The data structure for a SAS/AF object is a named SCL list. The SCL list stores all of the instance variables for an object. Diagram 1 shows some of the list entries that store the instance variables for a text entry object. For the sake of brevity, not all the instance variables for the text entry object are listed in Diagram 1.

```
( USERATTR=() [39]
  ROW=5
  COL=59
  LEN=14
  NAME='TEXTBOX'
  LABEL='TEXTBOX'
  LENGTH=14
  DESC='Special Text Entry'
  COLOR=28
  _PAD_CHAR_=' '
  _PROTECT_='N'
  _VALUE_='Some text info'
) [37]
```

Diagram 1. Partial SCL list for text entry object

The instance variables can be modified either through an attributes window or with a SCL program. A program can access instance variables through methods or by using SCL list functions. For example, to obtain the text of the text entry object and assign it to the SCL variable *textval* in the SCL code for a method, use the program line:

```
textval=getnitemc(_self, '_VALUE_');
```

Alternately, use the program line:

```
call send(_self, '_GET_TEXT_', textval);
```

To access the text in the SCL code for the frame where the text entry object is instantiated, use the program line:

```
call notify('textbox', '_GET_TEXT_',textval);
```

The `_self_` expression is used for a class to reference itself in methods as in the 'call send' example. On the other hand, an object can be referenced by name, as in the 'call notify' example.

DERIVING A SUBCLASS

A subclass should be derived when a new attribute or behavior is consistently needed for a set of objects in an application. When developing a class library, it is a good idea to make a catalog to store all the class entries, SCL code for methods, and custom attribute frames. We will refer to this catalog as the *class library catalog*. Figure 1 displays a class library catalog with the `datalist` entry that is developed in this paper.

Entries		
DATALIST CLASS	List of data set labels	06/11/95
DLSTSEND CLASS	List of data set labels - BROADCAST	06/11/95
STEXTBOX CLASS	Special Text Entry	06/12/95
VARLIST CLASS	Variable label list	06/11/95
VLSTRECV CLASS	Variable label list - RECEIVE	06/11/95
DLATTR FRAME	Custom attributes for datalist.class	06/11/95
VLATTR FRAME	Custom attributes for varlist.class	06/11/95
DATALIST SCL	Methods for datalist.class	06/11/95
DLATTR SCL	SCL for DLATTR.FRAME	06/11/95
DLSTSEND SCL	Methods for dlstsend.class	06/11/95
STEXTBOX SCL	STEXTBOX.SCL	06/12/95
VARLIST SCL	Methods for varlist.class	06/11/95
VLATTR SCL	SCL for VLATTR.FRAME	06/11/95
VLSTRECV SCL	Methods for vlstrecv.class	06/11/95

Figure 1. Example class library catalog

For our example, we use a list box that displays a list of data set labels in a SAS data library. The list box should have a text attribute that identifies a SAS data library as the source of the data set labels to be listed. Thus, we need to create a new instance variable that we will call *source*.

To begin this process, start a build session and open the class library catalog. Select `FILE-NEW-ENTRY` and specify the entry name as *datalist* and the entry type as *class*. This will open the class editor window shown in figure 2. The 'Class Entry' and 'Description' fields will be filled with the new class name and a default description that can be modified. Select the control arrow at the right of the text entry object labeled 'Parent Class' to open a select window, then select `SASHELP.FSP.LISTBOX.CLASS` by selecting appropriate entries in the *Library*, *Catalog*, and *Entries* list boxes.

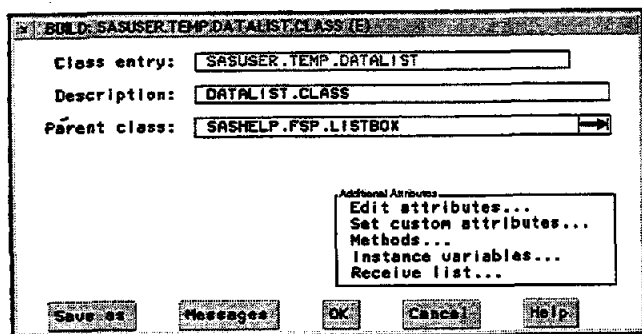


Figure 2. Class Editor

An alternate way to define a new class is to open a new frame and make a list box object by clicking the right mouse button, selecting *make*, then selecting *List Box*. This will

open the List Box attribute frame shown in figure 3. Selecting the 'Save as' button will start the class editor. With this process it is not necessary to specify the 'Parent class' field, however the 'Class Entry' field must be supplied with a three level SAS name that identifies the libname, catalog, and catalog entry for the class. It is also a good idea to modify the class description at this time.

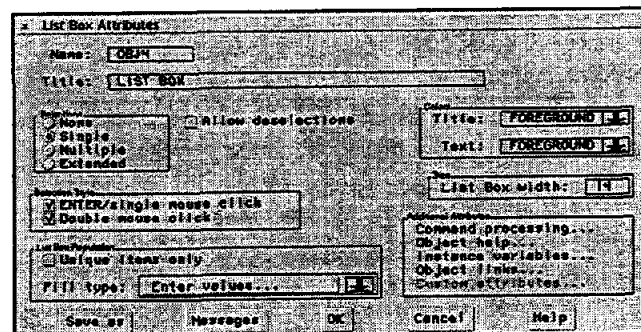


Figure 3. List Box Attribute frame

Next, we need to define the source instance variable. In the class editor, select the 'Instance variables...' option to start the Instance variables window displayed in figure 4. Choose 'Actions', then 'Add mode on' and enter *source* in the 'Name' field. This completes the process to define a subclass and specify a new instance variable.

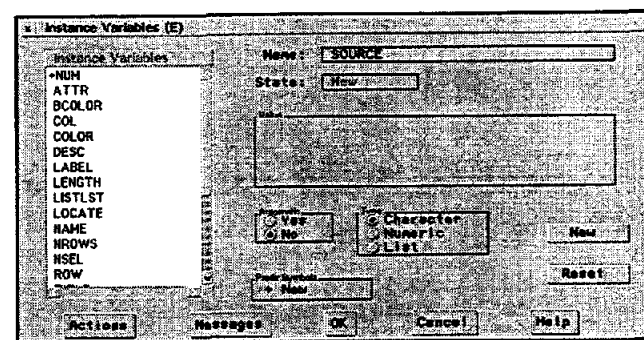


Figure 4. Instance variable window

The new list box subclass *datalist* inherits all attributes and methods from the list box class. The next section describes how to override the `_REPOPULATE_` method. This method refills a list box with its items.

OVERRIDING METHODS

When a new subclass is created that exhibits behavior different from the parent class, it is necessary to override the inherited method that controls the behavior. Usually it is a good idea to execute the inherited method by using the call super function either before, after, or during the custom processing in the override method. Many times an error will result if the inherited method is not executed because the inherited method performs processing that is critical for the class.

In our list box example, we want to populate the list box in a special way with a set of data set labels. To accomplish this we will override the `_REPOPULATE_` method. We can override the method by first executing the inherited method then carrying out custom processing to access data set labels and adding the labels as items in the list box. The SCL code to achieve this is in diagram 2. This code is stored in the class

library catalog as the entry DATALIST.SCL. In general, it is a good idea to store methods for a subclass in an SCL entry of the same name.

```

*****;
*** Module: Datalist.scl ***;
*** Purpose: Methods for Datalist class ***;
*****;
_REPOP_:
Method;
call super(_self_,'DELETE_ALL');
call super(_self_,'REPOPULATE');

*** Do nothing if source not defined ***;
If getnitemc(_self_,'source')='' then return;

*** Do nothing if libref for source not defined ***;
If libref(getnitemc(_self_,'source')) then return;

***Assign a fileref to the physical location of the db ***;
*** Do nothing if fileref assignment fails ***;
If filename('DATABASE',
            pathname(getnitemc(_self_,'source')))
then return;

*** Do nothing if directory does not exist ***;
If not fexist('DATABASE') then return;

*** Make a list to store the data set labels ***;
labllst = makelist();

*** Open the db directory ***;
did = dopen('DATABASE');

If did then do;
  *** Get the number of db members ***;
  memcount = dnum(did);
  Do ddidx = 1 to memcount;
    *** Open member and add member label to list ***;
    dsname = substr(dread(did,ddidx),1,
                    index(dread(did,ddidx),'.')-1);
    dsid = open(getnitemc(_self_,'source')||'|'||dsname,'i');
    If dsid then do;
      dslabel = attrc(dsid,'LABEL');
      If dslabel='' then
        dslabel = dsname;
      labllst = setitemc(labllst,dslabel,ddidx,'Y');
      rc = close(dsid);
    End;
  End;
  rc = dclose(did);
End;

*** Sort list ***;
labllst = sortlist(labllst,'VALUE');
*** Add data set labels to list box ***;
Do ddidx = 1 to listlen(labllst);
  call super(_self_,'ADD',getitemc(labllst,ddidx),ddidx);
End;

*** Clean up ***;
rc = dellist(labllst);
rc = filename('DATABASE','');
Endmethod;
Return;

```

Diagram 2. Method to override _REPOPULATE_

The code section labeled _REPOP_ contains the method that overrides _REPOPULATE_. The first step in the overriding method is to execute the inherited _DELETE_ALL_ method in order to remove all the items from the list box. This prepares the list box to be repopulated. The next step is to execute the inherited _REPOPULATE_ of the parent class. The next four steps perform some error checking. The error checking performed includes determining if the instance variable *source* has been assigned, determining if the value stored in *source* is a valid logical, assigning a filename to the physical path corresponding to the logical stored in *source*, and, finally, determining if the physical path exists. If a failure occurs at any point, the method will end processing.

The remainder of the method opens the directory where the data are stored and successively opens each data set and stores the data set label in a local SCL list. After all data set labels have been stored in the SCL list, the SCL list is sorted, all items are deleted from the list box *_self_*, and then each label in the SCL list is added to the list box *_self_*.

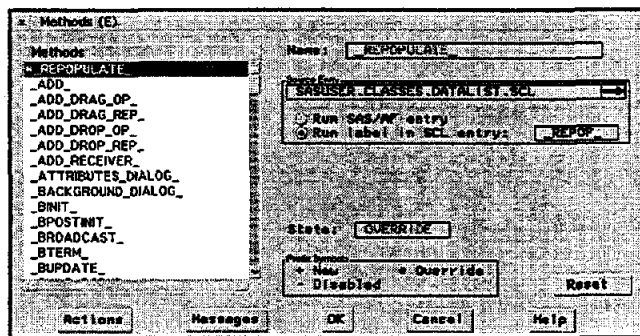


Figure 5. Methods Window

After the SCL program has been written to override an inherited method, the new method must be added to the subclass. To do this open the class editor window and select the 'Methods...' option (refer to figure 2). This will open the methods window displayed in figure 5. From this window, select the name for the inherited method in the Methods list box. In the 'Source Entry' field enter the four level SAS name for the SCL entry that contains the new method or select the control to the right of the field to pop-up a select window and select the SCL entry. Select the option to 'Run label in SCL entry' to complete the process to override the inherited method.

DEFINING A NEW METHOD

When a new subclass is created, many times it is necessary to program new methods for the subclass. Especially if new instance variables are created that need to be accessible and modifiable by the application program. In these cases, methods need to be programmed to make the instance variables available.

In our example, we have an instance variable for a list box that stores a libref to a SAS database. An application may want to modify the libref in order to list the data for several libraries different at various times. In this case, the list box needs a method to modify the source instance variable. Another case is for the list box to provide the two-level SAS data set name for a selected label. A method needs to be programmed to provide this information. Diagram 3 shows methods that accomplish these tasks. After a new method is programmed, it needs to be added to the class using the class editor (refer to figure 5).

```

length rowtext $40;

CHG_SRC:
  Method src $;
    _self = setnitemc(_self,_src,'source');
    call send(_self, '_REPOPULATE_');
  Endmethod;
Return;

GET_SRC:
  Method dsname $;
    *** Get selection ***;
    call send(_self, '_GET_LAST_SEL_',row,issel,rowtext);
    If not issel then return;

    *** Do nothing if source not defined ***;
    If getnitemc(_self,'source')='' then return;

    *** Do nothing if libref for source not defined ***;
    If libref(getnitemc(_self,'source')) then return;

    *** Assign a fileref to the physical location of the db ***;
    *** Do nothing if fileref assignment fails ***;
    If filename('DATABASE',
      pathname(getnitemc(_self,'source')))
      then return;

    *** Do nothing if directory does not exist ***;
    If not fexist('DATABASE') then return;

    *** Open the db directory ***;
    did = dopen('DATABASE');

    If did then do;
      *** Get the number of db members ***;
      memcount = dnum(did);

      dsname = '';
      Do ddidx = 1 to memcount;
        *** Open member and add member label to list ***;
        dsname = substr(dread(did,ddidx),1,
          index(dread(did,ddidx),'.')-1);
        dsid = open(getnitemc(_self,
          'source')||'.'||dsname,'i');
        If dsid then do;
          dslabel = attrc(dsid,'LABEL');
          If dslabel = '' then
            dslabel = dsname;
          rc = close(dsid);
        End;
        If dslabel = rowtext then leave;
        else dsname = '';
      End;
      rc = dclose(did);
    End;

    *** Clean up ***;
    rc = filename('DATABASE','');

    If dsname > '' then
      dsname = getnitemc(_self,'source')||'.'||dsname;
  Endmethod;
Return;

```

Diagram 3. New methods for instance variable access

The CHG_SRC method provides a way to change the value stored in the source instance variable. The GET_SRC method

retrieves the two level SAS data set name corresponding to the data set label in the list that is currently selected. This method provides a way for an application to access the SAS data set that a user selects. When the CHG_SRC method runs, it calls the _REPOPULATE_ method after changing the value of the source instance variable to repopulate the list with new data set labels. The GET_SRC method scans the database for the data set that has a label equal to the text that has been selected on the list box. It returns a blank string if the corresponding data set is not found, otherwise it returns the two-level SAS data set name that coincides with the selected label.

DESIGNING A CUSTOM ATTRIBUTE FRAME

When a subclass is created that contains additional attributes, it is a good idea to design a frame to assign initial values for these attributes. In our example of the derived list box subclass datalist, we have no way to assign a libref for the source when an object of this class is instantiated unless we

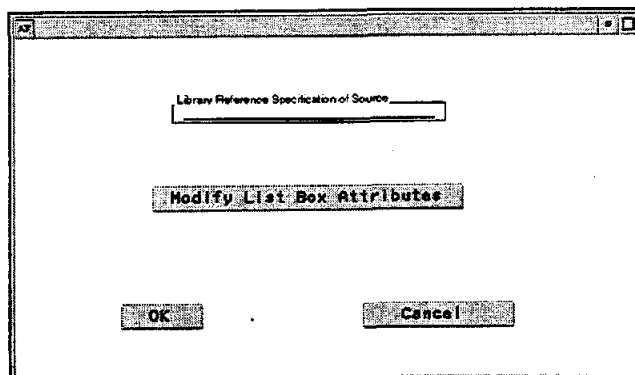


Figure 6. Attribute frame for datalist subclass

build an attribute frame to collect this information. Figure 6 shows a simple attribute frame for the datalist subclass called dlattr.frame stored in the class library catalog.

The frame dlattr.frame has a text entry object named *source* to collect the libref to be stored in the source instance variable, a button to invoke the standard list box attribute window named *lattr* for modifying all inherited attributes, an 'OK' button, and a 'Cancel' button. The processing for this frame is handled by the class library entry *dlattr.scl* detailed in diagram 2.

```

entry optional = _widget_8_uattr_ $ _class_ 8;

INIT:
  source = getnitemc(_widget_,'source',1,1,'');
Return;

LATTR:
  call display('sashelp.fsp.attrbox.frame'
    ,_widget_,_uattr_,_class_);
Return;

TERM:
  _widget_ = setnitemc(_widget_,source,'source');
Return;

```

Diagram 2. SCL code for DLATTR.SCL

Every custom attribute SCL program must accept three optional parameters. The *_widget_* parameter is the list handle for the SCL list that contains the objects instance

variables. The `_uattr_` is the entry name of the custom attribute window and `_class_` contains the list handle for the class to which the object belongs.

In the example, the INIT section assigns the value in the object's source instance variable to the local source variable so that the current value will display when the frame starts. The LATTR section executes when the button to modify list box attributes is activated. This section starts the standard list box attribute frame stored in the SASHELP.FSP catalog. The processing of the 'Ok' button is equivalent to issuing the SAS 'End' command and the processing of the 'Cancel' button is equivalent to issuing the SAS 'Cancel' command. When the frame terminates, any value that has been entered or modified in the text entry box is assigned to the objects source instance variable using the `setnitemc` function.

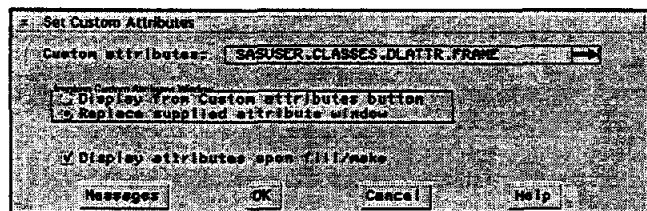


Figure 7. Set Custom Attributes window

To assign the custom attribute frame to the class, open the class editor and select the 'Set custom attributes ...' option (refer to figure 2). The set custom attributes window displayed in figure 7 will open. Either enter the four level SAS name for the custom attribute frame or select the control object at the right of the entry field to start a selection window for selecting the frame entry.

Selecting the 'Replace supplied attribute window' option, as we have done in this example, will cause the custom attribute window to execute each time a new object for the new class is instantiated. Selecting the 'Display from Custom attributes button' option will cause the custom attribute window to display when the 'Custom attributes...' options is selected in the parent attribute window (see figure 3).

RESOURCING A NEW CLASS AND CREATING AN OBJECT

The resource list is the list of classes available to the developer for designing an application. When a new subclass is created it should be added to the resource list so that objects in the subclass can be created for an application. To do this, open the application catalog and copy the resource entry `sasuser.fsp.build.resource`. This copies the entry `resource.build` that contains most of the standard classes available in SAS/AF.

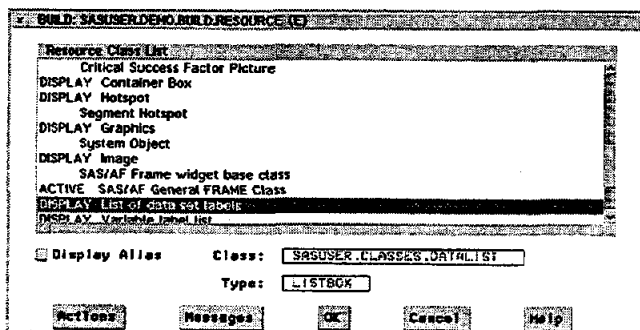


Figure 8. Resource Class List

Edit the `resource.build` entry by double clicking on it in a build window or by issuing the command

`edit build.resource`

in a catalog window. This will open the Resource Class list window displayed in figure 8.

Activate the 'Actions' button and select 'Add' from the resulting pop-up menu. From the select window, choose the three level name to the new subclass. The subclass is now available for use in your application.

EXAMPLE APPLICATION

This section presents a simple example application that uses the `datalist` class. The application consists of one frame that has four *library* push buttons, a *datalist* object, a text entry object, and a "Done" push button. The library push buttons select a SAS database for the *datalist* object to list. The text entry object displays the two-level SAS name corresponding to the selected data set label. The "Done" button terminates the frame. Figure 9 shows the application frame.

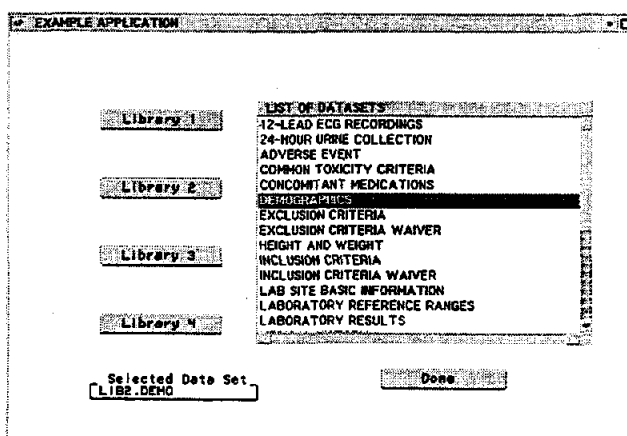


Figure 9. Frame from example application

The library buttons are on the left side of the frame, the *datalist* object occupies the right side of the frame, and the text entry box displaying the selected data set is directly below the library buttons.

Diagram 4 shows the SCL code to control this frame. When the frame initializes, the window is named 'EXAMPLE APPLICATION' using the `wname` function. The application assumes that library references `lib1`, `lib2`, `lib3`, and `lib4` are assigned. Each of the library buttons, when activated, sends the message 'CHANGE_SOURCE' with a text string for the `libref` as a parameter, to the *datalist* object for changing the value of the source instance variable. The `CHG_SRC` method executes changing the source instance variable and repopulating the list.

When the *datalist* object is selected the code block labeled `DATALIST` executes. First the number of items selected is retrieved. If the number of items selected is zero, the application sends a blank string to be displayed in the text entry box. Otherwise, the application sends the message 'GET_SOURCE' to the *datalist* object. The `GET_SRC` method executes and returns the two-level data set name in the SCL variable `ds_name`. The application then sends a message to the text entry box to display the data set name. Activating the 'Done' button quits the application by issuing the SAS command 'End'.

```

Length ds_name $15;

INIT:
  call wname('EXAMPLE APPLICATION');
Return;

LIB1:
  call notify('datalist','CHANGE_SOURCE','LIB1');
Return;

LIB2:
  call notify('datalist','CHANGE_SOURCE','LIB2');
Return;

LIB3:
  call notify('datalist','CHANGE_SOURCE','LIB3');
Return;

LIB4:
  call notify('datalist','CHANGE_SOURCE','LIB4');
Return;

DATALIST:
  call notify('datalist','_GET_NSELECT_',numsel);
  If not numsel then
    call notify('textbox','_set_text_', '');
  Else do;
    call notify('datalist','GET_SOURCE',ds_name);
    call notify('textbox','_set_text_',ds_name);
  End;
Return;

```

AUTHORS ADDRESS

Carl Haske, Ph.D.
 STATPROBE, Inc.
 3885 Research Park Drive
 Ann Arbor, MI 48108
 (313) 769-5000 x115
 E-Mail:
 • 72700,1075@compuserve.com
 • statprob@oeonline.com

Diagram 4. SCL code for example application

This example application is simple to build and illustrates the power available to the application developer that uses subclassing and the object-oriented concepts of inheritance.

CONCLUSION

Deriving a class library is very advantageous for application development. Classes are a means for producing reusable and maintainable programming. Classes can be reused repeatedly in multiple applications. Maintaining a class library is very efficient since new subclasses can be derived from existing classes to add functionality while preserving backward compatibility in applications that use the class library.

REFERENCES

SAS Institute, Inc. (1993), *SAS/AF Software: FRAME Entry, Usage and Reference, Version 6, First Edition*, Cary, NC: SAS Institute Inc.

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

ACKNOWLEDGMENTS

SAS and SAS/AF are registered trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.