ABSTRACT
The SAS/AF® FRAME entry employs Object Oriented Programming (OOP) techniques to create GUIs. This paper introduces several OOP concepts and how they have been implemented in the FRAME entry. Examples are provided to demonstrate how application developers can:
- create new software units (classes)
- provide default settings for class attributes
- define new class attributes (instance variables)
- write generic SCL programs (methods) for classes

INTRODUCTION
As the two preceding papers have shown, you can quickly prototype GUI applications using the software units like Icon and Push Button provided with the SAS/AF FRAME entry. These GUI building blocks are collectively known as widgets. In OOP terminology each widget type is a class.

When you perform a FILL or MAKE operation while editing a FRAME entry, you choose a widget class from a popup menu (Figure 1) that shows all currently available classes. This menu is called a resource list. The resource list is created from a catalog entry of type RESOURCE. Later examples show how you can create your own resource entries that include classes that you have designed.

Figure 1 Default Resource List

Each class serves as a model for creating a specific software unit known as an object. In OOP an object is an instance of a class.

OOP CONCEPTS
In the FRAME entry shown in Figure 2, there are six objects: one instance of the SAS/GRAPH® Output class, two instances of the Graphics Text class, and three instances of the Icon class.

Figure 2 GRAPH1.FRAME in Execution Mode

Classes are organized into hierarchies so that each object you create inherits all the characteristics of its immediate parent class and any higher level parent classes.

Figure 3 shows the Object class as the root class from which all other classes are derived. The Widget class is the parent class of all classes that appear in the default resource list.

Figure 3 Hierarchy of Supplied Classes

Widgets such as Text Entry, Icon and List Box are specialized subclasses of the Widget class. SAS/AF software stores class definitions in catalog entries of type CLASS (Figure 4). All supplied classes are located in the SASHELP.FSP catalog. The CLASS entry maintains all properties of the class including:
- name of parent class
- name and location of programs (methods) the class can execute
- name and default values of data (instance variables) associated with the class.
Figure 4 CLASS Entry in Browse Mode

Figure 5 illustrates that every class can have instance variables as part of its structure. Some instance variables are the object attributes you edit when you create an object. Other instance variables may be part of the class definition but not immediately apparent as attributes. The DESC variable provides a description of the class (e.g., Text Entry). Every class has this instance variable. Likewise, all widgets have names. The NAME instance variable is defined as part of the Widget class. Unless a widget subclass supplies a default name for a new object, it is OBJn, representing the nth object in the FRAME entry.

Figure 5 Classes as Collections of Data and Methods

Subclasses can have additional instance variables. The Text Entry class has several variables including COLOR and FTYPE (field type). The default value for COLOR is YELLOW and the default for FTYPE is CHAR, denoting a character field. You can reset these values in the attribute window for a Text Entry object.

Classes also have a collection of programs (methods) the class can execute. The Object class supplies several methods. The _IS_A_ method allows application programs to determine whether an object is a instance of a given class. Any subclass can execute this method.

All widgets can perform a common set of methods. These have been abstracted to the Widget Class so each Widget subclass does not have to individually define them. The _HIDE_ and _UNHIDE_ methods were illustrated in an earlier tutorial. Because of the inheritance principle, these methods can be used with all widget subclasses. Widget subclasses have additional specialized methods. The _VALIDATE_ method is defined only for the Text Entry Class. The _SET_COLOR_ method is defined for several (but not all) widget subclasses.

Each subclass inherits all methods and instance variables of its parent classes by default. However, a subclass can disable or modify (override) inherited methods and instance variables as well as introduce new methods and instance variables. Application developers can create their own subclasses of the Object class or widget subclasses. The Widget class is a utility class that cannot be subclassed directly. This following examples show how you can create subclasses of the Widget subclasses like Push Button and Text Entry. Tutorial 4 shows how to subclass the Object class.

CREATING A SIMPLE SUBCLASS

During the design phase of a GUI application, an application developer may notice repeated instances of some software building blocks. A simple example would be a help push button that issues the HELP command. Rather than recreating the same object in different FRAME entries, the developer could create a subclass of the Push Button class and register this subclass in a customized RESOURCE entry. When the developer performs a FILL or MAKE operation using this new resource list, an object created from the Help class will possess all the desired attributes: size, color, button text, and command processing.

This is a very important outcome of OOP — software reusability and consistency. If all programmers use this new class to create Help buttons, GUI design standards are automatically enforced. The easiest way to create a subclass from an existing widget subclass is to use the SAVE AS command bar item in an object's attribute window. This button creates a new subclass based on the current attributes of the object. You are placed in edit mode for defining a new class entry (Figure 6). In the example below, CLASSES.CLASSES.HELP.CLASS is the storage location of the new class. Its description is changed to Help Button. The parent class is automatically filled in.

Figure 6 Creating a New Class with Save As

Once this entry is saved, you can add it to a custom RESOURCE entry. The default resource is stored in the SASHELP.FSP.BUILD.RESOURCE catalog entry. You can copy this entry to your own catalog and make modifications to it. Figure 7 shows the CLASSES.CLASSES.BUILD.RESOURCE in edit mode.
Figure 7 RESOURCE Entry in Edit Mode
Once you add the Help Button to the resource list, you can create new FRAME entries using this resource. BUILD RESOURCE will be used if it is found in any of these catalogs (search sequence):

1. current catalog
2. SASUSER.PROFILE
3. SASHELP.FSP

You can also explicitly declare a resource name with the PROC BUILD RESOURCE=name option or the RESOURCE name command in the BUILD window.

Figure 8 shows the custom resource list that pops up in response to a FILL request in a new FRAME entry.

Figure 8 Filling a Region Using a Custom Resource List

A MORE COMPLEX EXAMPLE

The next example shows how a subclass can define its own instance variables and methods.

Recall from earlier tutorials that a Text Entry object was used to accept a user-specified name of a catalog entry containing SAS/GRAPH output. The catalog name and entry type (GRSEG) was supplied by the application. The same strategy was used to accept the name of a report stored in a catalog entry of type OUTPUT. In each case nearly identical SCL programming was needed to validate user input and provide a selection list of available catalog entries.

Ideally, there would be a Field Type specification for the Text Entry class that would validate that an entry exists in a given catalog, much like how a ONEVAR field type validates a SAS data set variable exists in some specified SAS data set. Rather than submitting an item for the SUGI Software Ballot, we will create our own class called CATENTRY.

Specifications for the CATENTRY Class

The CATENTRY class is a subclass of the Text Entry class. It is designed to accept and validate an 8-character name representing the third-level of a four-level catalog entry name. The entry type and catalog name are supplied by the application. Desired behaviors for class objects:

1. If the user enters a question mark (?) as the object text at any time, a dialog window displaying all valid choices appears. If the user chooses an item from this selection list, appropriate text is automatically assigned to the object.
2. If the user enters other text in the object that does not correspond to an existing catalog entry, an appropriate error message is displayed and the error flag for the object is turned on.

The CATENTRY class has these properties:

Default Attributes
Type=NAME, Length=8, Protect=No, Caps, Noprompt

Instance Variables
All instance variables from parent classes will be inherited. The attributes given above are initial settings for corresponding instance variables. New character variables named CATALOG and ETYPE will be created for the subclass.

Methods
All methods from parent classes will be used unaltered, except for the _VALIDATE_ method of the Text Entry class. This method will be redefined (overidden) by the subclass.

New methods will be added: SET_CATALOG, SET_ENTRY_TYPE, and GET_FULLNAME.

The first step is to create a CLASS entry named CLASSES.CLASSES.CATENTRY.CLASS. You can use the PROC BUILD EDIT command to do this, or SAVE AS technique shown earlier. Set the default attributes as specified.

Next, access Instance Variables attribute and define two new variables CATALOG and ETYPE as automatic character variables. Notice that an initial value of SASHELP.FSP is given for CATALOG (Figure 9). This means that all objects created from this subclass will have this initial value for CATALOG. (We could have left it blank.)

Figure 9 Adding a New Instance Variable

Finally, access the Methods attribute and select the existing _VALIDATE_ method. Specify the name of an SCL entry that will contain the SCL source program for the redefinition of this method (Figure 10). Label represents the SCL labeled section within the
SCL entry.

**Figure 10 Overriding an Existing Class Method**

All other methods are new. Figure 11 shows the specification for the SET_ENTRY_TYPE method. We have elected to store each method in a separate SEL catalog entry. You could store all methods in one SEL entry.

**Figure 11 Adding a New Class Method**

Before we show the SEL programs that make up these methods, we will discuss how classes structures and objects have been implemented in SEL.

**IMPLEMENTATION OF OOP IN SEL**

When you create objects in a frame, the frame records information about

<table>
<thead>
<tr>
<th>Object Attributes</th>
<th>for each object: name, position, size, class and all other attributes you designate</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Frame Attributes</td>
<td>name, window size, window colors, menu, keys, help, etc.</td>
</tr>
<tr>
<td>Resource Info</td>
<td>location and names of all available classes and methods</td>
</tr>
</tbody>
</table>

At run time, all this information is loaded into memory in the form of SEL lists. SEL lists were introduced in Release 6.07.3. They are documented in SAS Technical Report P-216, SAS/AF Software, SAS/FSP Software, and SAS Screen Control Language: Changes and Enhancements, Release 6.07. If you have not programmed with SEL lists, you can think of them as linked list structures or dynamic arrays that can grow or shrink during execution. Every SEL list has a unique list identifier assigned to it, much like how the OPEN function assigns a unique identifier to a SAS data set. SEL supports about 60 functions that manipulate SEL lists. Writing class methods involves reading information from (and possibly updating) SEL lists associated with classes, objects and the frame.

To make programming easier, several SEL features specific to OOP have been implemented:

**Automatic SEL Variables**

If a FRAME entry has an associated SEL entry, the automatic SEL variable _SELF_ is present in the SEL vector. It contains the list identifier for all frame information. An SEL variable is also created for each object in the frame using the object name. The type and contents of this SEL variable depend on the object class. In general, the associated SEL variable does not contain the list identifier for the object but some instance variable value.

**Functions and Routines**

The SEND routine sends a method to an object via its object identifier.

The NOTIFY routine is a special case of SEND that can be used in a frame. It allows you to send a method to a widget via the object's name. This routine must translate the object name into an object identifier in order to execute the method.

**Automatic Execution Sequence of Methods**

Some methods are executed automatically, without an application program calling them explicitly. The table below shows some of these methods. Notice that, in our example, the _VALIDATE_ method for a Text Entry object is called before any programming code you might specify in the object-labeled section of the frame's SEL program.

<table>
<thead>
<tr>
<th>Method</th>
<th>Class</th>
<th>When Executed</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>INIT</em></td>
<td>Object</td>
<td>before INIT section</td>
</tr>
<tr>
<td><em>POSTINIT</em></td>
<td>Widget</td>
<td>after INIT section</td>
</tr>
<tr>
<td><em>VALIDATE</em></td>
<td>Text Entry</td>
<td>before object-labeled section</td>
</tr>
<tr>
<td><em>SELECT</em></td>
<td>Widget</td>
<td>after object-labeled section</td>
</tr>
<tr>
<td><em>PRETERM</em></td>
<td>Widget</td>
<td>before TERM section</td>
</tr>
<tr>
<td><em>TERM</em></td>
<td>Object</td>
<td>after TERM section</td>
</tr>
</tbody>
</table>

**WRITING METHODS**

SCL entries allow you to store compiled SEL programs (methods) that are capable of execution independent of a windowing environment. Here is the complete source listing for the method SET_CATALOG stored in CLASSES.CATENTRY.SET_CAT.SEL:

```plaintext
length catalog $ 17;
SET_CAT: /* Set catalog name */
method newcat $17;
    catalog=upcase(newcat);
endmethod;
```

The label SET_CAT serves as the entry point during execution. NEWCAT is a parameter passed in by a CALL NOTIFY or CALL SEND routine. Because CATALOG was defined as an automatic
Instance variable, it can be referenced directly in the method. The value of the SCL variable CATALOG automatically updates the object's list item (instance variable) named CATALOG. If CATALOG had not been an automatic instance variable, this method would have been written as follows:

```sas
length catalog $17;
_SET_CAT: /* Set catalog name */
method newcat $17;
_self_.setitemc(_self_.upcase(newcat), 'catalog');
endmethod;
```

SETNITEMC is a list function that directly updates the object's list item named CATALOG with the value of NEWCAT.

Within methods, two automatic SCL variables are available, _FRAME_ and _SELF_. _SELF_ represents the list identifier of the object which called the method. _FRAME_ is the list identifier for the FRAME entry containing this object. These automatic variables allow class methods to be generic for any object of that class (or subclass) in any FRAME entry.

The _VALIDATE_ method is similar:

```sas
VALIDATE:
method;
/* Turn off error flag and obtain value */
call send(_self_,'_erroroff_');
call send(_self_,'-9'et_text_,'catentry');
if catentry=_blank_ then return;
/* User requests help by entering a ? */
if index(catentry, '?') then do;
catentry=catlist(catalog,etype,l,'N');
catentry=scan(catentry,l,' ');;
call send(_self_,'_set_text_','catentry');
end;
/* Call the parent VALIDATE method */
call super(_self_,'_validate_');
call send(_frame_,'_in_error_','inerror');
if inerror=1 then return;
/* Validate entered value */
fullname=trim(catalog)|'|'.|'|'|etype;
if exist(fullname)=0 then do;
call send(_self_,'_erroron_');
message='ERROR: Catalog entry |'| fullname|'| not found.';
call send(_frame_,'_set_msg_','message');
return;
end;
endmethod;
```

As an example of how to use these methods, suppose you create a FRAME entry that has a CATENTRY object named GRAFNAME and a SAS/GRAPH Output object named GRAPH. You want the user to enter the name of a GRSEG entry stored in the SUGI.GRAPH.S catalog. After validating this name, your SCL program should display the corresponding stored graph.

The SCL program for the frame:

```sas
INIT:
call notify('grafname','set_catalog', sugi.graphs');
call notify('grafname','set_entry_type', 'grseg');
return;

GRAFNAME:
if grafname _blank_ then
call notify('grafname','get_fullname',graph);
return;
```

The _VALIDATE_ method of the Text Entry class is automatically executed whenever an object of this class (or subclass) is modified. A method labeled with the keyword _validate_ executes after execution of the _VALIDATE_ method. Since our CATENTRY subclass needs to perform specialized validation, we have overridden the parent _VALIDATE_ method with our own method called VALIDATE.

```sas
VALIDATE:
method;
/* Turn off error flag and obtain value */
call send(_self_,'_erroroff_');
call send(_self_,'-9'et_text_,'catentry');
if catentry=_blank_ then return;
/* User requests help by entering a ? */
if index(catentry, '?') then do;
catentry=_blank_;
catentry=catlist(catalog,etype,l,'N');
catentry=scan(catentry,l,' ');;
call send(_self_,'_set_text_','catentry');
end;
/* Call the parent VALIDATE method */
call super(_self_,'_validate_');
call send(_self_,'_in_error_','inerror');
if inerror=1 then return;
/* Validate entered value */
fullname=trim(catalog)|'|'.|'|'|etype;
if exist(fullname)=0 then do;
call send(_self_,'_erroron_');
message='ERROR: Catalog entry |'| fullname|'| not found.';
call send(_frame_,'_set_msg_','message');
return;
end;
endmethod;
```

When you write a method that overrides a parent method, you may want one of these behaviors:

- **ignore the parent method, only execute the code of the overriding method**
- **execute the parent method within the overriding method to take advantage of what the parent method does.**

The SUPER routine is used to execute the inherited parent method within your method. In our example we want to execute the
_VALIDATE_ method of the Text Entry class to perform preliminary validation. Since the default type (FTYPE) for our class is NAME, the parent _VALIDATE_ method can guarantee that any entered value follows SAS naming convention. The parent method is responsible for generating any error messages and setting the error flag. If an entered value passes this test, additional validation is performed to guarantee it is also a valid catalog entry name.

Once the methods for the CATENTRY class are created, you can create CATENTRY objects in frame applications that employ the custom resource entry CLASSES.CLASSES.BUILD RESOURCE. The payoff for creating this subclass is that any application developer can use this software tool with the assurance that it will perform correctly. Because the instance variables and methods have been encapsulated in the class definition, no application specific code is needed to perform these actions. This reduces redundant coding in applications that perform variations on the same task. Application developers can also further subclass the CATENTRY class if more specialization is required.

SUMMARY
The SAS/AF Frame entry supports Object Oriented Programming techniques. You can create your own subclasses of the supplied widget subclasses to extend and refine the library of software tools used to create GUIs.

The next tutorial will show how you can

- create FRAME entries that initialize the instance variables of your subclasses
- write methods that enable objects to send messages directly to other objects
- create subclasses of the Object class

REFERENCES

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