INTRODUCTION

SAS/AF for version 6.11 brings a wealth of new object oriented programming (OOP) features to SAS applications developers. This paper will focus on some of these enhancements with emphasis on the use of per instance methods and event handling. Please note that this paper presumes a good working knowledge of SAS/AF frame development and a familiarity with basic OOP concepts.

QUICK REVIEW: OOP in SAS/AF

This section reviews OOP concepts with emphasis on their application in SAS/AF. Understanding these concepts is key to fully understanding the example given later in this paper. Some new 6.11 features are mentioned.

It is useful to think of classes as templates. From these templates you create the software objects you work with. In SAS/AF, the list of widget classes available in a frame appears when you click your right mouse button (e.g., text entry class, icon class, push button class, etc.). When you select one of these classes an object of that class is created and placed in your frame. You may then modify this object (its color, length, etc.) to customize it for your current need. Note that non-widget classes are also available.

Each class has attributes (implemented as data values stored in instance variables) and behavior (implemented through programming routines called methods). Very importantly, each object (or instance) of a class incorporates all properties of its class.

For each instance of a class, instance variables are used to store information such as the object’s color, its length, the names of other objects it can send messages to, etc. In SAS/AF, all instance variables are implemented as named items in an SCL list. In other words, each object is associated with an SCL list that contains one named item for each instance variable. Thus if an object has 20 instance variables (color, type, length...) it will be associated with an SCL list containing 20 named items (a named item color, a named item type, a named item length...) and each item will have a value (e.g., color=blue, type=numeric, length=8, etc.). You change an attribute by modifying the object’s instance variable value.

Methods are programming routines that in effect supply an object’s behavior. Classes are associated with many methods (e.g., _SET_COLOR, _HIDE, etc). In SAS/AF, user-created methods are SCL code routines stored in SCL entries. In 6.11, invocation of a method can also trigger display of a catalog entry (e.g., another FRAME, a CBT, etc.).

The totality of SAS/AF classes forms a hierarchy (see fig 1). By default, child classes inherit all attributes of the parent class (also called superclass). In figure 1, the text viewer class inherits from the widget class and (because the widget class inherits from the object class) from the object class.

![Diagram](https://via.placeholder.com/150)

**Fig 1: Partial class hierarchy for SAS/AF 6.11**

When a method is referenced on an instance of a class but the method is not defined for that class a search is made up the class hierarchy through
its parents until the method is found. For version 6.10 and later changes to instance variables can also travel down the hierarchy. For example, if the default value of an instance variable is changed in a parent class, this change will be reflected in all of its subclasses unless the value has been explicitly overridden by a subclass. This behavior can significantly simplify system maintenance and modification.

On occasion you will create your own classes from existing classes through a process called subclassing. With subclassing, you make a copy of a class (which has all of the attributes of its parent class) and then modify or extend its attributes. Prior to SAS 6.11, subclassing through the class editor was the only way to add new instance variables and to add or modify methods of a class. In 6.11 you can do this and other activities “on-the-fly” using the meta object protocol (MOP).

When creating your own classes you will often ‘attach’ new behavior by overriding one or more inherited methods. In overriding, you substitute a custom-written method for an inherited method. For example, if you wish to add new behavior which is exhibited when the user selects an object you can override the _SELECT_method. Essentially, this tells your class to run your custom method instead of its inherited one. However, it is important to note that you will generally still invoke the parent method through use of CALL SUPER (in some cases a SUPER call will be required, because some methods perform vital processing — e.g., _INIT_, which instantiates the object).

In writing your methods, you will often take advantage of several special automatic instance variables which are available to all methods of the class. For example, _SELF_ contains the object ID of the object invoking the method, and _FRAME_ contains the ID of the frame on which the object resides.

Objects communicate with each other through messages. Any object can respond to a message from another object, and each message can drive a method on the receiving object. In SAS/AF, you can send messages using broadcast-and-receive or through event handling. It is important to note that objects should always initiate behavior from other objects through messaging — objects never directly modify each other’s data.

Working in SAS/AF 6.11

This section gives an overview of several important new features for working with classes in version 6.11 of SAS/AF Frame entries.

Meta Object Protocol

Prior to 6.11, to create a class or to add or delete attributes of a class you needed to use the class editor. However, 6.11 changes this with the new Meta Object Protocol (MOP). MOP includes the meta class CLASS.CLASS, a class that works on other classes and allows you to dynamically create other classes, to query the class hierarchy, etc.. MOP also allows you to add new instance variables and to override methods at run time. MOP is particularly useful when using non-widget classes, although it can also be used to create widget classes “on-the-fly” that can appear on your screen almost magically. However, note that any class created on-the-fly cannot be saved as a class entry — creating new classes which persist over time as catalog entries still requires using the class editor.

While MOP lets you do many wonderful things when working with classes, a detailed discussion of this topic is beyond the scope of this paper. We will focus on one aspect of MOP, its per instance methods.

Per Instance Methods

Per instance methods are a major enhancement to 6.11, allowing you to dynamically bind a method to an object at run time (see table 1). Thus, when you want to override a method, you no longer need to create a new subclass. The benefits of this can be considerable, particularly if you are writing large, generic applications. For example, you may want to add some behavior to an instance to handle a user interface event that might not happen on any other frame. For this you would not want to create a whole new class. Rather, you can use the _SET_INSTANCE_METHOD_method to attach your method dynamically through SCL code executing at run time.
It is important to note the order of searching for per instance methods. As noted previously, when a method is invoked the list of methods defined for the current class is searched and, if the method is not defined for the class, the search moves up the class hierarchy through the superclass’ list of methods. With per instance methods, the search always starts in the object’s list of per instance methods. If found there, the search may stop. This has the same implications as overriding a method when you are subclassing -- if you override a method already defined for the object’s class, keep in mind that this method should generally still be invoked through CALL SUPER. In fact, an implicit CALL SUPER can be done through the WHEN parameter of _SET_INSTANCE_METHOD_. This parameter allows you to control the relative execution of the default method through the BEFORE and AFTER options, and even allows you to chain execution of several methods. Refer to the 6.11 Frame Class Dictionary for details.

See SIMPLE EXAMPLE below for sample use of _SET_INSTANCE_METHOD_.

BROADCAST AND RECEIVE

One use of per instance methods could be in creating a broadcast-and-receive (BNR) link between objects. To set up communication between objects prior to 6.11 you needed to subclass in order to override an object’s _RECEIVE_ method (it is the _RECEIVE_ method that responds to the message). In the past this posed a dilemma: while you wished to set up communication between objects, you also wanted to avoid creating many subclasses (often the interobject communication required was specific to a single frame, and thus your subclass might be useful to one frame only). With per instance methods you can now attach a _RECEIVE_ method to an object without subclassing.

However, use of BNR still has some maintenance overhead. Regardless of how you attach a receive method to an object, BNR requires that you write a method which responds to a message passed to the object. If an object is capable of accepting many different messages, then your custom receive method could soon begin looking like a tribute to the SELECT statement (see sample code in fig. 2).

In fig 2, each conditional would invoke an object method, including relaying any parameters that were passed. So, with BNR you must handle all messages through conditional statements in your custom method, even if the method to be invoked is one of the instance’s native methods and requires no parameters (e.g., _GREY_). In addition, BNR allows passing only one character and one numeric parameter, so if you wish to pass several values you must place them in an SCL list before passing and then the receiving object must extract these values. With 6.11, however, this functionality can be achieved with much less effort through the use of event handling.

EVENT HANDLING

Event handling (EH) allows you to send messages to multiple recipients without regard to who these recipients are or how the message will be used. In addition, when generating an event any number of parameters can be passed.

EH is powerful yet simple — it uses only four methods, and you will generally use only two of these. A general description of these methods appears in Table 2. The four parameters for _SET_EVENT_HANDLER_ are sender, event, action, receiver and these are summarized in table 3.
More detailed syntax is not included here for space constraints. Please refer to the OBJECT CLASS chapter of the 6.11 Frame class dictionary.

Event Handling can be used to trigger actions based on a user interface event. That is, you can set up a link between a specific user interface event occurring for an object (e.g., the user clicks on a push button) to drive some action on another object (e.g., in response to the event, a "listening" list box updates its contents). Objects can be both event senders and event "trappers". In general, the sender does not know or care who is receiving an event message.

More detailed syntax is not included here for space constraints. Please refer to the OBJECT CLASS chapter of the 6.11 Frame class dictionary.

Event Handling can be used to trigger actions based on a user interface event. That is, you can set up a link between a specific user interface event occurring for an object (e.g., the user clicks on a push button) to drive some action on another object (e.g., in response to the event, a "listening" list box updates its contents). Objects can be both event senders and event "trappers". In general, the sender does not know or care who is receiving an event message.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET_EVENT_HANDLER</td>
<td>registers to the system interest in an event. This includes specifying the name of the event, the name of sender(s) the receiver is interested in, and an action to take upon occurrence of the event.</td>
</tr>
<tr>
<td>SEND_EVENT</td>
<td>sends out an event and, optionally, parameters to be received by any object that is &quot;listening&quot;.</td>
</tr>
<tr>
<td>DELETE_EVENT_HANDLER</td>
<td>ends a communication you have previously set up.</td>
</tr>
<tr>
<td>PURGE_EVENT_HANDLERS</td>
<td>clears all event handlers defined for an object.</td>
</tr>
</tbody>
</table>

In general, the event drives a method on the receiving object. In other words, the receiving object registers its interest in a specific event and essentially says, "If this event happens, I want this method executed". _SEND_EVENT_ always requires one parm (name-of-event) and any subsequent parm values are passed directly to the receiving method.

<table>
<thead>
<tr>
<th>Parm</th>
<th>Usage and Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>sender</td>
<td>the object sending the event. Valid values: a character (widget name) or num (widget id) or a quoted asterisk (&quot;**&quot; = any sender)</td>
</tr>
<tr>
<td>event</td>
<td>the name of the event that will trigger the event handler execution. A character value (hint: try to keep your event names generic, NEW VALUE not NEW X AXIS VALUE)</td>
</tr>
<tr>
<td>action</td>
<td>the action to be triggered by the event. Value is usually a method name but also can be a 4-level name of a catalog entry (e.g., a frame))</td>
</tr>
<tr>
<td>receiver</td>
<td>the object receiving the event, usually the same object setting the event handler (value is in same format as sender -- but use of asterisk as a parm is NOT recommended)</td>
</tr>
</tbody>
</table>

Table 3 details _SET_EVENT_HANDLER_ parameters. Use caution when using the asterisk parm. It can be used as the sender but should not be used for the receiver parameter.

As noted previously, values can be passed as parameters in the _SEND_EVENT_ method call. By default, all values are passed to the receiving object. However, the receiving object can filter the parameters it receives. This is particularly useful if you are creating a tool kit of classes which will interoperate and must be generic, sending information that may be extraneous to some of the receivers. To filter incoming parameters, you can use the optional parameters parameter of _SET_EVENT_HANDLER_. The general format is CALL SEND (widgetid, _SET_EVENT_HANDLER_, sender, event, action<(parameters)>, receiver). The process of setting communication between two objects is as follows:

**Programmer action at development time:**
1. developer sets up an event-to-action link
   1.1. define a _SEND_EVENT_ call for a sending object
   1.2. define _SET_EVENT_HANDLER_ call for a receiving object, registering its interest in an event. This also identifies the type of action to be taken when the event occurs (usually this is the name of a method to be driven on the object)

**User/Application interaction at run time:**
1. the user performs a user interface event, resulting in a _SEND_EVENT_ method execution
2. the fact that the event occurs is "broadcast" (along with any parameters)
3. the class handler determines if any _SET_EVENT_HANDLERS_ have been set for this event
4. if an event handler(s) has been defined for the event, it is processed. In general the action taken is the execution of a method, although the syntax LIB.CAT.NAME.TYPE in the action parameter field can be used to invoke a catalog entry.

It is important to stress that the sender should never be concerned with what actions will be taken based on the event (i.e., how other objects will "react"). The sender only sends the message. Interpretation of this message is left to the receiving object — or objects, because part
of the power of event handling is its ability to affect several objects at one time in different ways, with each object responding in its own fashion. We will discuss this in somewhat greater detail when considering models/viewers.

Event Handling Between Frames

At the moment event handling between frames and tasks is not available in the current version of 6.11 and will not be addressed in this paper.

SIMPLE EXAMPLE

This example will show the basics of using _SET_INSTANCE_METHOD_ and event handling. Remember that this paper assumes a good working knowledge of frames, so the individual steps for creating objects and setting attributes will not be detailed here.

In this example you will create a frame named SASUSER.FORDEMO.SUG196.FRAME which will contain the 4 objects listed below. Several of these objects will communicate through event handlers. You will also create one SCL entry to contain a method which will be attached to the list box instances at run time using _SET_INSTANCE_METHOD_. Sample screen appears in fig 3, code behind the screen appears in figure 4, and the method is in figure 5.

To build the frame create the following objects:

1. a PUSHBUTTON named DATASET of length 20 with button text set to “Data to Graph”.
2. a LIST BOX named XAXIS with box title “X Axis” and which is populated by an SCL list named ALLVARS.
3. a LIST BOX named YAXIS with box title “Y Axis” which is populated by an SCL list named NUMVARS.
4. a GRAPHICS object named EGRAPH. You may wish to set the region outline to button with a pixel width of about 6 for aesthetics. Leave the graph type set to SCATTER PLOT, which is the default.

The following bullets relate to the code in figure 4 (see next page).

1. the EGRAPH object sets 4 event handlers which drive methods controlling the display of the graph (note that no objects directly modify graph values – they only do so here through events). For example, in the first call the EGRAPH object essentially says “let me know if the object DATASET sends a NEW VALUE event, and if it does I want to run my _SET_DSNAME_ method on myself” (in this case the _SEND_EVENT_ will pass an extra parameter, a dataset name, which is used with _SET_DSNAME_ to the set active dataset for the graph - see bullet 0).

2. here we add our custom method to the list box class instances XAXIS and YAXIS. For example, in the first call we are saying “for the instance identified by the object ID %AXIS_IX> temporarily attach a method named REFRESH_LIST which is stored in the entry ‘SASUSER.FORDEMO.SUG196.SCL’ in the labeled section VARLIST.” Alternatively, an override of _REPOPULATE_ with a CALL SUPER could also have been used here.

Binding this method allows us to use special instance variables like _SELF_ and _FRAME_ in our method code (see below). Note that the object IDs are fetched (the _GET_WIDGET_ calls) so CALL SEND can be used here instead of CALL NOTIFY. This is done here just to point out that CALL SEND rather than NOTIFY is preferable for performance reasons (however, note that use of CALL SEND rather than CALL NOTIFY is required in our method).
when the event occurs a _SEND_EVENT_ is executed. In our example, this drives a method on the listening object EGRAPH. Any parameter values passed to _SEND_EVENT_ are passed through the event handler to the driven method. For example, when DATASET does a _SEND_EVENT_ it passes the new dataset name as a parm (the value stored in the variable temptxt). Thus, each time the DATASET object sends a new event it is sending a new dataset name, and this is used by EGRAPH in a _SET_DSNNAME_ invocation to set the active graph dataset.

```
/* ____________________________ 
  NAME: SUGI96.SCL 
  DESC: A simple demo of _SET_INSTANCE METHOD_ and EVENT 
  HANDLE , where a graph object responds to dataset 
  and variable selection events. 
  NOTE: to simplify this example no event handling was done 
  between the dataset and list box objects. Consider 
  doing this with subclasses of the list box class. */

LENGTH rc & temptxt $20;
INIT:
  call notify('egraph', '_SET_EVENT_HANDLER_', 'dataset', 
    'NEW VALUE', '_SET_DSNNAME', 'egraph');
  call notify('egraph', '_SET_EVENT_HANDLER_', 'dataset', 
    'NEW VALUE', '_SET_BORDER_WIDTH', 'egraph');
  call notify('egraph', '_SET_EVENT_HANDLER_','xaxis', 
    'NEW VALUE', '_SET_INDEX_VAR', 'egraph');
  call notify('egraph', '_SET_EVENT_HANDLER_','yaxis', 
    'NEW VALUE', '_SET_INDEX_VAR', 'egraph');
  allvars = makelist();
  numvars = makelist();
  call notify('xaxis', '_GET_WIDGET_','xaxis',xaxis_id);
  call notify('yaxis', '_GET_WIDGET_','yaxis',yaxis_id);
  call send(xaxis_id, '_SET_INSTANCE METHOD_','REFRESH_LIST', 
     'sasuser.fordemo.sugimeth.scl','varslist');
  call send(yaxis_id, '_SET_INSTANCE METHOD_','REFRESH_LIST', 
     'sasuser.fordemo.sugimeth.scl','varslist');
RETURN;
TERM:
  rc = dellist(allvars);
  rc = dellist(numvars);
RETURN;
DATASET:
  temptxt = dirlist('sasuser','data','l','y'); /* note-using SASUSER lib! */
  call notify('dataset', '_SEND_EVENT_','dataset', 'NEW VALUE', temptxt);
  call notify('xaxis', '_REFRESH_LIST_','tempdata',allvars);
  call notify('yaxis','_REFRESH_LIST_','tempdata',numvars,'N');
RETURN;
XAXIS:
  call notify('xaxis', '_GET_LAST_SEL_','row', issel, temptxt);
  call notify('xaxis', '_SEND_EVENT_','NEW VALUE', temptxt);
RETURN;
YAXIS:
  call notify('yaxis', '_GET_LAST_SEL_','row', issel, temptxt);
  call notify('yaxis', '_SEND_EVENT_','NEW VALUE', temptxt);
RETURN;
```

FIG 4. The code behind the frame (entry SUGI96.SCL)
/* NAME: SUGIMETH.SCL */
/* DESCR: Just a routine which returns an SCL list containing the names of variables for the passed dataset. Valid*/
/* TYPEVARS parameter values are C=charvars, N=numvars */

length type $#;

VARSLIST:
method dataset $1 varlist $8 optional=typevars $1;

if typevars "_blank_" then type = "TYPE=" | typevars | ""; 
if exist(dataset) then do;
   entries = makelist();
   rc = clearlist(varlist);

dataclass=instance(loadclass('SASHHELP.FSP.DATASETS'));
call send(dataclass, '_SETUP_', dataset);
call send(dataclass, '_GET_MEMBERS_', entries, type, 'NAME');
/* fill our list with the names of the variables */
do i=1 to listlen(entries);
   name = getitem(getitem(entries, i), 'name');
   rc = insertc(varlist, name, -1);
end;
call send(_self_, '_REPOPULATE_');
call send(dataclass, '_TERM_');
rc = dellist(entries);
end;
else call send(_frame_, '_SET_MSG_','ERROR: dataset does not exist');
endmethod;
RETURN;

FIG 5. The method (entry SUGIMETH.SCL)

the TYPEVARS parameter can be used to filter the variables collected from the SAS Data File class (e.g., if type="" then only numeric variables will be retrieved). The SAS Data File class is one of the new family of classes used to collect information on SAS environment objects (also see SAS Library class, SAS Metailib class, SAS External file class, SAS Variable class, etc.). The SAS Data File class provides an OOPS interface which allows you to query many data file attributes and to manage datasets (copy, delete, rename, etc.)

because our REFRESH_LIST method now is bound to our object (in this case, as a per instance method) we can access special automatic variables like FRAME and SELF. SELF refers to the object that has called the currently executing method (see FRAME manual, esp. pg 114). FRAME refers to the frame on which the calling object resides (see FRAME manual pgs 213, 239).

After creating the frame, type in the code shown in figure 4 into its SCL entry (SUGIB96.SCL). Finally, type in the method code shown in fig. 5 into an SCL entry named SUGIMETH. Now TESTAP the frame (note that this looks to the SASUSER library for data, so be sure you have at least one dataset in your SASUSER library, or substitute another library in the DIRLIST function call).

In TESTAF mode, when you select a dataset the list boxes should fill with variables and the graph will set its active dataset and border text to the dataset you have selected. When you have selected both an X and Y axis variable the graph will respond to these events, creating a scatter plot of the selected variables. Note that our example communicates with the EGRAPH object only through the event handlers.
In this example we have used per instance methods to essentially create a list box subclass "on-the-fly", and the example is useful at least to show how _SET_INSTANCE_METHOD_ can be used. However, this simplified approach is probably not the best for this situation.

Consider: how often do you find yourself needing a list box to display a list of datasets, or the variables for a dataset? Would it not be useful to have subclasses available that would provide this functionality for you? Classes that would interoperate, sending messages to each other when their status changed? To do this you could create subclasses of the list box class which create, fill, and maintain their own SCL lists based on messages sent to them, and which always send NEW VALUE events whenever they themselves are modified. And, of course, as with all subclasses you have the ability to build in your business rules or other functionality (e.g., filter the list of datasets displayed to the user based on the user's access rights).

Classes such as these could be very handy tools in your developer's tool kit. If well designed, such a tool kit can significantly reduce the development and maintenance costs of your SAS/AFC applications. Through this approach our "simple example" could be made considerably simpler, and in fact by using classes which incorporate event handling we could provide all the functionality of our example with just four _SET_EVENT_HANDLER_ calls in the INIT section of SUGI96.SCL (this will be demonstrated in the SUGI96 presentation).

What are some of the potential advantages of using event handling in conjunction with subclassing? For one, ease of code modification. Try this: replace the push button widget with a text entry field named DATASET, then comment out the DIRLIST call in the DATASET section of your SCL code (for your new DATASET object be sure to set SELECTION STYLE to "ENTER/Single Mouse click"). Now try recompiling the frame and typing a dataset name into the field. Why does our example still work? Because you are broadcasting the value of any widget called DATASET whenever its value changes, and because changing the type of widget makes no difference to the recipients of the message -- they care only about the event, not the type of object sending the event. Thus, use of event handling in combination with a library of classes which work together like this can let you more easily modify your application (and more quickly create applications in the first place).

Event handling is particularly useful when using the model/viewer/controller paradigm. This is briefly discussed in the next section.

**Event Handling with Models and Viewers**

It is important to note here that a major benefit of event handling is that it allows you to easily attach many different viewers to a data model when using the model/viewer/controller paradigm (MVC). MVC is a major enhancement for SAS 6.11.

MVC lets you attach one or more viewers to a single model (see figure 6). In MVC the model notifies viewers of any changes to its state without regard to who is "listening" or how its information will be used. As you might guess, event handling is very useful here, allowing the model to do a _SEND_EVENT_ whenever its state changes so that any viewers attached to it can automatically be refreshed to reflect these changes. In fact, models should never communicate directly to any viewer/controller via methods (note the direction of the arrows in figure 6) but should always do so through events. A viewer/controller, on the other hand, often sends methods to a model based on these events (e.g., to get more information on how the model's state has changed).
APPLICATIONS DEVELOPMENT

CONCLUSION

There is an almost dizzying number of new features for the SAS/AF Frame developer working in SAS version 6.11. SAS/AF is more OOPs than ever, particularly with its meta object protocol. Per instance methods can allow you to customize a class for a specific task without requiring subclassing. This can reduce the number of classes you must create and maintain, potentially a significant simplification and reduction of work. Event handling provides a sophisticated yet simple mechanism for messaging between objects, and combined with a library of classes customized for your own needs can have a significant impact on the ease of system development and maintenance.

For further information feel free to contact the author at GRONELLD@PIPELINE.COM.

ACKNOWLEDGEMENTS

Many thanks to SAS Institute staff Julie Platt, Linda Partenheimer, Susan Carroll, Jay Wright and especially to Randy Pierce.

REFERENCES


SAS Screen Control Language Reference, Version 6, second edition, SAS Institute, Cary, N.C.

SAS/AF Software: FRAME Entry, Usage and Reference, Version 6, SAS Institute, Cary, N.C.


Pierce, Randy, The Benefits of Object-Oriented Application Development Using the SAS System, SUGI 19 Proceedings

Norton, Andrew A., Designing Complex Systems Using SCL Data Objects, SUGI 20 Proceedings

Gronell, Daniel, Working with Class in SAS/AF Frame Entries, SUGI 20 Proceedings