Paper 39-25

Changes and Enhancements to the SAS® Component Language (SCL)

For Component Development

Carl LaChapelle, SAS Institute Inc., Cary, NC
Tammy L. Gagliano, SAS Institute Inc, Chicago, IL

ABSTRACT

SAS Component Language (SCL), formerly referred to as Screen Control Language, is the object-oriented programming language designed to facilitate the development of interactive SAS applications with SAS/AF® software. A lot has changed since Version 6, with respect to the features SCL has to offer. This paper will discuss some of the most significant enhancements and how your application can benefit by exploiting the newest technology. This is not an introductory paper but rather it assumes previous experience in writing SCL and FRAME-based applications using SAS/AF software. Knowledge of creating subclasses and/or overriding methods would also be helpful.

INTRODUCTION

The intent of this paper is to provide an overview of new features that have been added to SAS/AF software. Although there have been many changes made to the build-time environment and available components, the paper will focus primarily on SCL language enhancements.

The topics we will discuss are:

- Using dot notation when writing SCL programs
- Declaring SCL variables, including new list, array and object types
- Better error handling control through trapping and/or stopping program halts altogether
- Changes made to SCL methods such as method signatures and USECLASS syntax when implementing methods. Also, how you can take advantage of compile-time binding for improved performance and better compile-time validation
- A simple programmatic approach to building and maintaining your CLASS entries
- Miscellaneous features such as new syntax for creating SCL lists along with several new functions for SCL list management, host dialog windows useful when performing basic file management tasks and more.

Throughout the paper, references will be made to Version 8 since this is the most current release. However, most of the examples shown will work with either Version 7 or Version 8 of the SAS System unless otherwise indicated as a Version 8.1 only enhancement.

DOT NOTATION

In previous releases of SAS/AF software, to query an object for information about its current state or attribute settings, you'd find yourself writing a mixture of SCL statements -- maybe some method calls, maybe some SCL list function calls, etc. For example:

```
list = makelist();
call notify('obj1','_get_region_', list);
title = getnitemc(list, 'border_title');
style = getnitemc(list, 'border_style);
color = getnitemc(list, 'border_color');
rc =  dellist(list);
put title= style= color=;
```

In Version 8, the previous code would look like:

```
put graphOutput1.borderTitle=
    graphOutput1.borderStyle=
    graphOutput1.borderColor=;
```

Similarly, setting properties in Version 6 would also include a mixture of method calls and SCL list functions:

```
call notify('obj1',
     '_set_border_title_', 'Sample title');
call notify('obj1',
     '_set_border_style_', 'embossed');
call notify('obj1',
     '_set_border_color_', 'black');
call notify('obj1', '_set_graph_',
     'lib.cat.entry.grseg');
```

The above code in Version 8 would look like

```
graphOutput1.title = 'Sample title';
graphOutput1.borderStyle = 'embossed';
graphOutput1.borderColor = 'black';
graphOutput1.graph = 'lib.cat.entry.grseg';
```

Notice in both examples, that not only have the number of lines been reduced, but the code is more consistent and easier to read. Using dot notation, the syntax is exactly the same across all components. You don't have to remember what method to call or what arguments it supports. Plus, there's less room for syntax errors because you're not typing as many underscores, commas and quotes!

Also, you might have noticed from the examples that object names are no longer limited to 8 characters. They can now be up to 32 characters and the default object names supplied by new SAS/AF components are more descriptive than `objn` used previously.

In addition to the examples shown above, you can use dot notation in your SCL programs in any of the following forms.

```
/* invoking a method */
graphOutput1.method(<arguments>);
/* invoking a method that returns a value */
returnValue =
     graphOutput1.method(<arguments>);
/* setting an attribute value */
graphOutput1.attribute = value;
/* querying an attribute value */
value = graphOutput1.attribute;
/* comparing the return value of a method */
if (graphOutput1.method(<arguments>)
     = some-value) then…;
/* comparing the value of an attribute */
if (graphOutput1.attribute = some-value)
     then…;
```

In dot notation, the first-level qualifier is always the name of an object, which in the above examples is `graphOutput1`. In order for this to work the object name must reference the object's ID, not its value. This is not the default behavior for legacy objects but is the default for new SCOM components.

Legacy Objects Versus SCOM Components

SAS Component Object Model (SCOM) is an object-oriented programming model that provides a flexible framework for SAS/AF component developers. A component in SCOM is a self-contained, reusable object with specific properties including attributes, methods, events, event handlers and a defined set of interfaces. SCOM components provide more functionality than the existing set of legacy objects – a legacy object being any object created from a CLASS entry available prior to Version 7. For more details on the SCOM architecture and how you can use components available with this new technology, refer to the SUGI24 paper *SAS Component Object Model (SCOM) in Version 7 of SAS/AF Software.*

Beginning with Version 7, the class library has grown to include many new classes all designed under SCOM. You can create these components on your frame by dragging them from the new Components window (available at build-time) and dropping them onto the frame. Because they are SCOM components, their object name references the object's ID by default which means you can immediately begin accessing any of the component's attributes or methods using dot notation in your FRAME SCL.

For legacy objects, the object's name does not reference the object ID by default. If you use a PUT statement to print the contents of a legacy object

```
put obj1=;
```

it will print the current value of the object

```
obj1='sasuser.sugi.sample.grseg'
```

which differs depending on what object you are using. In the example above, `obj1` is the name given to a SAS/Graph Output object in a frame. The value of a SAS/Graph Output object is the four-level name of the GRSEG entry currently being displayed.

You can use dot notation with a legacy object if you do one of the following:

- specify "ID" as the value for the legacy object's `objectNameUsage` attribute. You can do this through an option labeled 'Use object name as ID in SCL', which is available in the new Properties Window at build-time. Or, you can use the Class Editor to assign the attribute's value on the class itself.
- programmatically declare a local variable that contains the object identifier of the legacy object as an object in your SCL code. You can retrieve the object identifier for the legacy object using the frame's _getWidget method in your SCL. To find out more on how to declare this variable as an object, read the next section!

Since you can build SAS/AF applications using both legacy objects and new SCOM components, you might want to be consistent with your programming style. This enables you to use dot notation in your SCL programs regardless of whether you are working with a new component or a legacy object.

DECLARING OBJECTS AND OTHER SCL VARIABLE TYPES

To declare SCL variables in your program, you use the new DECLARE (or DCL) statement. Using the DCL statement, you can declare all types of variables used in your SCL program. Valid types are character, numeric, list, objects and arrays. The DCL statement essentially replaces the LENGTH statement. The LENGTH statement only let you declare variables of type character and numeric.

There is a set of reserved keywords that the DCL statement uses to recognize different type declarations: NUM, CHAR, LIST and OBJECT. Examples:

```
/* declare a numeric variable AGE */
declare num age;

/*declare a character variable LASTNAME and
assign the value 'Smith' to it */
dcl  char lastName = 'Smith';
/* declare an SCL list MYLIST */
dcl list mylist;
/* declare numeric variables AGE and WEIGHT
   and a character variable LASTNAME with
   length 20 */
dcl num age weight, char(20) lastName;
```

Blanks are used to separate multiple variables being declared with the same type as with `age` and `weight` in the last example. The last example also illustrates how commas are used to separate disparate types (i.e. declaring two numeric variables followed by a character variable named `lastName`). A length of 20 is specified for `lastName` using parentheses immediately following the CHAR keyword: `char(20)`.

As mentioned earlier, one way to use dot notation with a legacy object in your SCL is to programmatically declare a local variable that contains its identifier as an object. When declaring objects, you have two choices:

- assign it as a generic object
- specify the specific class name that the object identifier represents

The OBJECT keyword is used to indicate a generic object for situations where the object's type is not guaranteed. When an object's type is guaranteed, you can declare it by its specific class name as illustrated in the next example.

```
declare sashelp.fsp.image.class imageObj;
init:
  _frame_._getWidget('obj1', imageObj);
  imageObj._hide();
return;
```

In the above program, the _getWidget method is used first to retrieve the object identifier of a Version 6 Image Object on the frame. Its identifier is then stored in the local SCL variable named `imageObj`. Using the DECLARE statement, the `imageObj` variable is declared as an object of a specific type. You are stating that it will be an instance of the `sashelp.fsp.image.class`. Instead of using one of the DCL keywords (like OBJECT, CHAR, LIST or NUM) you use the three- or four-level name of the class to define its type followed by the variable name. Using the above DCL statement, you can now use dot notation to manipulate the properties of that object with `imageObj` as the first-level qualifier.

You should always declare your objects using this approach when possible. Doing so improves performance and enables the compiler to validate all tasks performed on that object. For example, if you are trying to access an attribute that does not exist on the object or if the method arguments used are incorrect, the compiler can catch these errors. When using generic objects, this validation will be performed at run-time instead, which only slows down your application.

In one of the examples shown earlier

```
dcl  char lastName = 'Smith';
```

the variable `lastName` was not only declared as a character variable using the CHAR keyword but it was also initialized on the DCL statement to the value 'Smith'. All variable types can be created and initialized on the DCL statement including objects, lists and arrays.

Here are a few more examples illustrating this:

```
/* declare 3 character elements in an
   array each with a length of 10 */
dcl char(10) colors(3);
/* declare a numeric array AGE with 3
   elements and initialize each element with
   a value */
dcl num age(3) = (21, 40, 65);
/* declare and instantiate an object of type
sashelp.classes.dataSetList_c.class */
dcl sashelp.classes.dataSetList_c
     myobj =instance(loadclass
     ('sashelp.classes.dataSetList_c'));
/* declare and create an empty list MYLIST */
dcl list mylist=makelist();
/* declare, create and initialize the list
     COLORLIST with three items */
dcl list colorList = {'red', 'blue',
     'green'};
```

When instantiating objects or creating lists using the DCL statement, it is still your responsibility to perform the appropriate clean-up in your SCL program using DELLIST to free the list or _TERM to terminate the object otherwise memory leaks will occur.

NEW SCL LIST SYNTAX AND SCL LIST FUNCTIONS

In the last example above, you might have noticed the following syntax was used to allocate an SCL list and fill it with three character items:

```
colorList = {'red', 'blue', 'green'};
```

This new SCL list syntax can be used anywhere in your SCL program. It is not limited to the DCL statement. The syntax rules for this new style are as follows:

- use the left bracket ({) to begin the definition of a list
- use the right bracket (}) to end the definition of a list
- use *name*= to proceed the item's value if you want a named item in the list where the string proceeding the equal sign is the name of the item

```
infoList = { fname='Carl',
             lname = 'LaChapelle',
             dept = 'DPD'};
```

- separate all items in the list with commas
- embed sublists using additional pairs of brackets around the items that make up the sublist. Sublists can be named items in the list as well.

```
infoList = { employee = {lname = 'Gagliano',
                         fname = 'Tammy'},
             dept = 'IDD'};
```

The only restrictions in using this new syntax are illustrated in the following example, which would fail to compile.

```
dcl list msgList, num rc, char(41) dsname;
INIT:
  if dsname = ' ' then
    ❶ msgList = {'Save changes to ',
                                dsname,'?'};
  else ❷ msgList = {'No table'||'specified'};
return;
TERM:
   rc = dellist(msgList);
return;
```

You cannot use variable substitution[1] or string concatenation[2] to define an item in the list using this syntax. Also, all lists created with this syntax need to be cleaned up using the DELLIST function as shown above. SAS/AF does not perform this clean up for you automatically.

Along with the new SCL list syntax, additional functions have been added that operate specifically on SCL lists.

COMPARELIST	compares two lists
GETITEMO, GETNITEMO, INSERTO, SEARCHO, SETITEMO, SETNITEMO	queries and retrieves items from a list of type object

HANDLING ERRORS IN YOUR SCL PROGRAM

No matter how much SAS/AF software helps us out with its improved syntax checking and compile-time validation using dot notation, there is still the chance of a program halt occurring at run-time. Obviously, this would not be through fault of ours as developers of course! To eliminate these nasty program halts and even build in some recovery, a new non-visual component has been added to trap and eliminate program halts from printing to the log.

By default, when a program halt occurs in your SCL program, information about the error is printed to the LOG and execution of your program stops. The statements following the one that caused the error are not executed. Using the Program Halt class (sashelp.classes.programHalt.class) you can trap program halts at run-time, perform some kind of additional validation as well as conditionally tell SAS/AF to continue execution of the application.

To use this class, you need to override the appropriate method. The following methods automatically execute when specific error conditions occur:

_onAttributeError	an error occurs from setting or querying an attribute through dot notation
_onGeneric	a generic error occurs or if no other methods in this table has been overridden.
_onGenericMath	the program halt occurs because of a genericMath error
_onOverFlow	the program halt occurs because of an overflow error
_onUnderFlow	the program halt occurs because of an underflow error
_onZeroDivide	the program halt occurs from a zeroDivide error

Once an error occurs, depending on the type of error, the corresponding method is automatically invoked on the program halt object. For example, if you accidentally divide a number by zero, the _onZeroDivide method is automatically invoked. You can override this method and place any logic you desire to control the behavior of your application when this error occurs.

The most common type of error is the generic error. For example if you use a GETITEMC function to retrieve an item in a list but that item's type is numeric, the program halt that occurs as a result falls under this category. When this occurs, the program halt object executes its _onGeneric method.

If the _onGeneric method is the only method you override on the program halt object, this method will get called for all errors that occur. Thus, you can place all of your error handling logic in a single method and use the attributes available to you to find out more about where and what type of error has occurred.

The attributes that supply information about the program halt are:

Attribute	Description
dump	A list containing the actual program halt
entry	The four-level name of the SAS/AF entry where the program halt occurred
keyword	The name of the method or function on which the program halt occurred.
keywordType	Indicates whether the keyword contains a method name or a function name. Valid values are METHOD, FUNCTION or a blank. If the value is blank, the program halt occurred on some other statement.
lineNumber	The line number in the entry where the program halt occurred
traceback	A list that contains the same information that the SCL traceback function provides
type	The type of error that occurred: GENERIC, GENERICMATH, OVERFLOW, UNDERFLOW, ZERODIVIDE, ATTRIBUTEERROR

These attributes are query only, meaning you can only retrieve their values. You cannot set them. One additional attribute that you will find extremely valuable is stopExecution. This attribute controls whether or not your program continues to execute after the error has occurred. By default, its value is 'Yes' which means execution will stop; however, unlike the other attributes mentioned, you can set its value in your method override to control this behavior.

Another approach to controlling some of the program halts in your SCL program is through a new option on the CONTROL statement named noHaltOnDotAttribute. By default, when dot notation is used to set or query an attribute's value and an error condition occurs, the result is a program halt. The error is printed in the LOG and execution of the program stops at this point. Using this option, you can tell SAS/AF to continue executing the statements following the error as if the error did not occur. This option is more limited than the Program Halt class described earlier for the following reasons:

- It only controls program halts that occur due to dot notation used when setting or querying attribute values. It would not, for example, trap a program halt due to an invalid SCL list function call or bad method invocation.
- You have no real way of knowing in your SCL whether or not a program halt has occurred unless you programmatically check the value of the object's errorMessage attribute after every dot notation call to see if it contains a value. Even then, this is not a foolproof approach because the errorMessage attribute can contain other messages besides errors such as warnings or general notes.

METHODS AND HOW THEY'VE CHANGED

New Naming Conventions

The first two things you might notice about method names when you edit a class in Version 8 using the Class editor are

- fewer underscores
- mixed casing

In Version 6, SAS/AF used underscores to separate words in method names (e.g. `_set_text_color_`). In Version 8, the new convention is to use a lowercase letter for the first letter and subsequent uppercasing of any joined word (such as `_setTextColor`).

The embedded underscores have been removed to promote readability. However, for compatibility purposes, `_setTextColor` is equivalent to `_set_text_color_`. All existing Version 6 code that uses the old style naming conventions along with CALL SEND and NOTIFY routines will still function with no modification.

Also, it is now possible in Version 8 for you to name new methods using a leading underscore; however, you should use caution when doing so. Methods with leading underscores typically imply they are SAS-supplied methods. Your methods named this way may conflict with new ones added to the parent classes in future releases of SAS/AF software.

Method Signatures

As shown in the beginning of this paper, you can use dot notation to invoke methods on an object. Aside from this programming style being easier to use, improved performance was also alluded to as a benefit of using this new syntax. However, using dot notation alone does not improve your application's run-time performance. You must declare your object as a specific class name type (also discussed earlier in the section labeled *Declaring Objects and Other SCL Variable Types*) and you must also define method signatures for all of the methods.

A method signature is a set of parameters that uniquely identifies a method to the SCL compiler. A method signature in SAS/AF software is usually represented by a shorthand notation called a *sigstring* (for signature string). This sigstring is displayed in the Signature column of the Methods table in the Class Editor. For example,

```
objectID._setTextColor('red');
```

would have a sigstring of (C)V. The method takes a single character argument. If the method took two arguments, the first being a numeric argument and the second being a character argument, the sigstring would be (NC)V.

The parentheses group the arguments and indicate the type of each argument, which can be numeric, character, list, object, the four-level name of a specific class, or an array. The value outside of the parentheses represents the return argument.

What's a return argument you ask? In Version 7, methods support return arguments. For example, the method used to retrieve a color might be invoked as follows:

```
dcl char(15) color;
color = objectID._getTextColor();
```

The sigstring for the `_getTextColor` method would be ()C. There are no arguments passed in to the method when it is invoked (i.e., no arguments listed inside the parentheses). The method, instead, returns a character string (represented by the value outside of the parentheses, which indicates it returns a character value). When a sigstring shows the character "V" (for "void") it indicates that no return argument is used.

When compiling a program, SAS/AF software uses the signature metadata and displays compile errors in situations where

- the method you are trying to invoke does not exist on the class
- the arguments passed in are too many, too few or of the wrong type

It's important to note that the compile-time validation and run-time performance gains adhere to the following rules:

- These features are only supported through dot notation, not through the old-style CALL SEND or NOTIFY routines. And *only* if the compiler knows the type of the object (i.e., declaration of the object as a specific class name type or within a USECLASS/ENDUSECLASS statement block, which is discussed later).
- These improvements are also not recognized for methods that display a signature of "(None)", which is supported primarily for compatibility purposes since classes prior to Version 7 contained no signature metadata. "(None)" does

not mean the method has no arguments but rather the signature for the method has not been defined in the class metadata.

Overriding versus Overloading Methods

If the object you are using does not have a method that provides the necessary behavior that your application needs, it is possible to

- implement a new method with this functionality
- override an existing method to add or modify its behavior.

When overriding a method, you should never alter the method's signature. This is generally considered poor object-oriented design and would break existing applications built that are depending on the previous method signature. Until now, there were no check points built into SAS/AF software to stop you from doing this; however, in Version 8 we provide better compile-time validation and error checking so that this situation cannot occur.

If you need to change the method signature of an existing object, instead of overriding the method you can use a new feature referred to as method overloading. Through method overloading, a component is allowed to have more than one method by the same name as long as their arguments differ in number, order and/or type. When you invoke an overloaded method (using dot notation), SAS/AF checks the method arguments, scans the signatures for a match, and executes the appropriate code.

For example, if you had an overloaded setColor method on your class, it would be possible for you to do the following in an SCL program using those methods:

```
dcl char color,
    num r g b;
myobj.setColor(color);
myobj.setColor(r,g,b);
```

In the example above, the first method's sigstring would be (C)V whereas the second method's sigstring would be (NNN)V. Both methods change the object's color. The first method takes a color name as input and the second method takes numerical RGB values. The advantage of overloaded methods is that they require programmers to remember only one method instead of several different methods that perform the same function but with different data.

There are two rules to remember when overloading methods:

1. A method with a signature of "(None)" cannot be overloaded.
2. Each signature in an overloaded method can have a different return argument type, but the arguments inside the parentheses must be different for each signature. For example, the following signatures would be allowed for an overloaded method:

   sigstring1 = (NN)V
   sigstring2 = (CC)N

   But the method could not be overloaded as follows:

   sigstring1 = (NC)V
   sigstring2 = (NC)N

   In the latter example, the signatures differ only by their return type. If someone tried to invoke the method as `object.method(3,'red')`, the compiler would have no way of knowing which method implementation to invoke.

Implementing Methods (New or Overridden)

When adding new methods or overriding existing ones on your class, you will still use the METHOD/ENDMETHOD statement block in your SCL entry to define each method.

On the method statement, you must declare each argument type. You can declare arguments using the same syntax as introduced with Version 6 or you can use the new DCL statement keywords introduced earlier in the section labeled *Declaring Objects and Other SCL Variable Types*. Using the new keywords, the syntax is slightly different than when used on the DCL statement. This is because there are additional pieces of information you can define about an argument such as its scope and whether or not the argument is to be treated as an input, output or update argument. For example:

```
label: method public
       Arg1:output:lib.cat.entry.class
       Arg2[3]:output:num  /*numeric array*/
       Arg2:update:list
       Arg3:input:char(32767)
       Arg4:num;  /*default is update*/
endmethod;
```

The keyword immediately following the `method` keyword (`public` in this example) indicates the scope of the method. This is new and controls the permission level for accessing the method. Valid keywords are PUBLIC (default), PROTECTED or PRIVATE.

This is followed by the list of arguments the method supports. For each argument, you can specify the argument name, its usage and its type all separated by colons. Valid usage keywords are INPUT, OUTPUT or UPDATE (default). Only the argument name and type are required. Method scope and argument usage information are optional, however, it is always good programming practice to include them so that others reading your code can see the complete method definition.

New to Version 7 is the option of implementing a method with a return argument.

```
label: method protected
             arg1:input:char(41)
             return=num;
   if arg1 ne ' ' then return 1;
   else return 0;
endmethod;
```

In the above example, depending on the value of arg1 that gets passed into the method, the method returns a 1 or a 0. Specifying `return=` on the METHOD statement tells SAS/AF that this method supports a return argument. The value for this option indicates the argument type, which in this case is numeric. Within the method block itself, when a RETURN statement is executed, the method is exited immediately and the value specified on the RETURN statement is returned to the calling program.

A significant change to the SCL language that should be used in all new development for method implementation is the new USECLASS/ENDUSECLASS statement block.

Syntax:

```
USECLASS four-level-class-name;
/* one or more method blocks go inside */
ENDUSECLASS;
```

The USECLASS keyword begins the block. The four-level-class-name specified is the name of the class entry that owns the methods you are implementing in this SCL entry. The ENDUSECLASS is used at the very end of the SCL entry.
With USECLASS, the compiler can validate information that is provided within the SCL and on the METHOD statement against the properties and metadata that are defined in the CLASS entry itself.

Using USECLASS, compile time errors result if it finds

- the implemented method is not overridden or new in the specified class entry

- an invalid label is being used
- a difference in SCOPE
- the wrong number of arguments are declared on the METHOD statement
- invalid argument types (including the validation of the argument as an input, output or update)

Within the SCL, methods and attributes for the specified class can be referenced without repeating the class identifier (that is, it eliminates the use of the _self_ qualifier). The compiler can

- distinguish between local variables and class attributes
- detect invalid method invocations

Consider the following guidelines when implementing your methods:

- Always define a method signature for a new method.
- Place all method implementation in the same SCL entry.
- Use the USECLASS/ENDUSECLASS statement block in the SCL entry.
- Implement everything in the SCL entry as a method, not as a linked label. Links are not allowed within USECLASS.
- Declare any local variables that you use within a METHOD block with a DCL statement or store the variable as a private attribute on the class. USECLASS syntax is not as forgiving in that it will require all variables to be declared, including numerics.
- Since SCL labels can be up to 32 characters, for readability purposes, use the name of your method as the label for the method's implementation.

For more detail on methods, their metadata, signature rules and USECLASS syntax, see *SAS Guide to Applications Development, First Edition*.

CREATING AND MAINTAINING CLASS ENTRIES USING CLASS SYNTAX

While the Class Editor offers a friendly user-interface to managing your class entries, you can create and manage a class in SCL using what's referred to as class syntax. This new syntax provides a nice alternative – especially if you need to make lengthy, repetitive changes. It might be quicker to make these changes in a text editor rather than working through a GUI environment.

The basic building blocks of this new syntax revolve around the CLASS/ENDCLASS statement block. Using this syntax, you can define any kind of property and associated metadata: attributes, methods, events, event handlers and interfaces. The method implementation can even be included within the same SCL entry. As a result, you get the same compile-time validation features discussed with USECLASS/ENDUSECLASS statement blocks.

Converting a CLASS entry to an SCL entry

From an existing class, class syntax can be generated using one of two approaches:

- interactively through the Class Editor
- programmatically using the new CREATESCL function.

To interactively generate the class syntax for a specific class, edit the CLASS entry using the Class Editor and from within the Class Editor,

1. Select **File ➔ Save As…** from the menu.
2. In the Save As dialog that displays, set the **Entry Type** to SCL.
3. Specify the **Entry Name** to store the SCL.
4. Select the **Save** button to close the dialog.

Programmatically, you can convert classes to SCL using the CREATESCL function.

Syntax:

```
rc=createSCL(className, SCLentryName,
     entryDescription);
```

Where…	is type…	and represents…
className	C	the four-level name of the CLASS entry from which the SCL class syntax will be generated
SCLentryName	C	the four-level name of the SCL entry used to store the generated class syntax
entryDescription	C	the description used for the SCL entry

Using either of the above approaches, the SCL entry will be created for you and will contain the class syntax that can be modified and used to regenerate the class entry.

.

Class syntax language elements

The class syntax only contains new or overridden properties for a class. It does not contain inherited properties. Properties include attributes, methods, events, event handlers, and interfaces. The metadata defined for each property is included as well.

For more detail on class properties and their metadata, see *SAS Guide to Applications Development, First Edition*.

The basic language elements for class syntax are as follows:

```
<ABSTRACT> CLASS class-name
<EXTENDS parent-class-name>
<SUPPORTED supported-interface>
<REQUIRED required-interface>
< / (class-metadata)>
<(attribute-statements)>
<(method-declaration-statements)>
<(method-implementation-blocks)>
<(event-declaration-statements)>
<(eventhandler-declaration-statements)>
ENDCLASS;
```

With the exception of the optional `ABSTRACT` keyword at the beginning of the file, all statements within the SCL entry must be contained within the `CLASS` and `ENDCLASS` statements. The entry begins with the `CLASS` keyword. It ends with the `ENDCLASS` statement.

The `ABSTRACT` keyword designates an abstract class, or a class that cannot be instantiated but is used as a common definition for other classes.

`class-name` is the one- to four- level catalog entry name of the class.

`EXTENDS parent-class-name` defines the parent class. If you omit the `EXTENDS` clause, the class is a subclass of sashelp.fsp.object.class by default.

`class-metadata` define options for the class and are typically managed by SAS/AF. They follow a forward slash (/) and are contained within parentheses. Commas separate multiple items. For example:

```
CLASS sasuser.test.tableSelectorButton.class
 extends sashelp.classes.pushbutton_c.class
 / (Type='AFCNTRL',
    Module='SASOWID',
    Description='Table Selector Button');
```

defines the CLASS entry sasuser.test.tableSelectorButton, which is a subclass of sashelp.classes.pushbutton_c.class. All other options that follow the forward slash are not typically items you will modify. Use caution when modifying or deleting any option on this statement as these are primarily managed internally by SAS/AF software.

The following sections describe the parts of the class syntax that you will most often be maintaining, which are the property definitions and their metadata.

Defining Attributes

To add a new attribute you would include the following property statement:

```
Public char backgroundColor
   /(Category = 'Appearance',
     Description= 'Returns or sets the
          background color');
```

Where…	*and represents…*
Public	the attribute's scope as public
char	the type of the attribute as character
backgroundColor	the name of the attribute
Category	the category used by the Class Editor and Properties window to group the attribute with others in the same category
Description	a short description for the attribute

The required elements on this statement are the attribute's scope followed by its type and name. These items are separated by blanks. All remaining attribute metadata (e.g., initialValue, validvalues, setCAM, editor) are optional but when specified,

- follow the forward slash (/) on the statement
- are included within parentheses
- separated by commas (,)

The property statement ends with a semicolon (;).

When overriding an attribute you must also include `State='O'` as part of its metadata.

```
Public num height
   /(State='O',
     InitialValue=21);
```

The above example defines height as an overridden attribute with an initial value of 2.

Defining Methods

To add a new method, you include the following property statement:

```
setcamTable: public method
     attrval:update:char return=num;
     / (SCL='sasuser.test.catalogModel.scl',
         Label='setcamTable',
         Description='The method executed when
             the value of the table attribute
             changes');
```

Where…	*and represents…*
setcamTable	the name of the method
public	the scope of the method as public. Public is the default if not specified
method	a required keyword for method statements
attrval:update:char	the definition for the method's first argument as a character string that behaves as an update argument. Update is the default if not specified. The argument's name is 'attrval'
return=num	a numeric return argument
SCL	the location of the method's implementation
Label	the labeled section within the SCL entry where the method implementation begins
Description	a short description for the method

When defining method properties, the required elements on the property statement are the label, scope, and its method arguments. The name of the entry where the method is implemented (if it is not implemented within the class syntax) should also be included as one of its metadata options.

You can optionally specify descriptions for each argument on the method statement using the argDesc*n* metadata item. The *n* is a number that represents the position of the argument on the method statement. In the example above, you would define a description for the attrval argument using

```
ArgDesc1 = 'the value of the attribute
           currently being changed',
```

For overridden methods, just like with attributes, you need to include state='O' as a metadata item. Also, if you are overriding a method that has no signature defined (i.e., signature='[none]'), you need to include signature='N' as an option on the METHOD statement. The compiler will generate an error without it.

```
foo: method arg1:num / (signature = 'n');
     / (State='O',
        ArgDesc1 = 'description for arg1',
        SCL = 'lib.cat.entry.scl',
         Label = 'foo',
        Description = 'sample description);
```

You can optionally include the method implementation within this same entry for new and overridden methods. For example,

```
_onClick: public method;
     / (State='O');
     dcl char(41) tableName;
     tableName = openSASFileDialog('DATA');
endmethod;
```

In the above method definition, the SCL and LABEL metadata items were removed and the ENDMETHOD statement was added. All SCL statements between the METHOD and ENDMETHOD statement block are the method's implementation.

Defining Events

To add event properties, use

```
EVENT event-name </(event-options)>;
```

where `event-options` are event metadata items.

Defining Event Handlers

To add event handlers, use

```
EVENTHANDLER handler-name < / (handler-
options)>;
```

where `handler-options` are metadata items.

Defining Supported or Required Interfaces

Interfaces are new to SAS/AF software and are used to establish Model/View communication between components in your application.

For more detail on this topic, refer to *SAS Guide to Applications*

Development, First Edition.

Using CLASS syntax, the interfaces that a class supports or requires is defined at the beginning of the class definition. Use `SUPPORTED interface-name` to indicate that the class supports a given interface. Use `REQUIRED interface-name` to indicate that the class requires a given interface.

For example, the List Box control requires the staticStringList interface. The beginning of its class syntax looks like this:

```
Class SASHELP.CLASSES.LISTBOX_C.CLASS
   Extends SASHELP.CLASSES.AFCONTROL
   Required
     SASHELP.CLASSES.STATICSTRINGLIST.INTRFACE
   / (Type="AFCNTRL",
      Module="SASOWID",
      Description="List Box Control",
      allowUnderscore="Y");
```

Saving the SCL class definition as a CLASS entry

To save an SCL entry that contains a single CLASS definition as a CLASS entry:

- From the Source window, select **File ➔ Save As Class**
- Or, you can enter the SAVECLASS command.

Saving an SCL program as a class is equivalent to saving a class that you created interactively with the Class Editor.

Tips on how to effectively use class syntax

Even though it is possible for you to include your method implementation within the CLASS/ENDCLASS statement block, you should use caution when doing so. This approach is straightforward if you only use class syntax to maintain your class entries. If you alternate between this approach and the Class Editor, it is possible that you will accidentally delete the method implementation.

For example, you have generated a class using class syntax, which includes the method implementation for its methods. You then decide to add a new attribute and use the Class Editor to perform this one task. At this point, the class syntax is out of sync with the properties that are actually stored in the CLASS entry. You decide to regenerate the class syntax using the CREATESCL function or through the Class Editor. If you perform this task and do not remember to store the class syntax in a different SCL entry than the one previously used, the existing SCL will be replaced, which means the method implementation in that entry is lost. Even if you store the updated version of the class syntax in a different SCL entry, you now need to transfer the method implementation from one SCL entry to the other to make it complete.

Because of the confusion this can cause, you should consider storing your method implementation in a separate SCL entry from the class syntax used to generate your class. You can still achieve the same performance gains as CLASS/ENDCLASS through compile-time binding if you implement your methods within a USECLASS/ENDUSECLASS statement block.

Using this approach, it is recommended that you establish a standard naming convention for your SCL entries that contain class syntax so as not to confuse them with SCL entries that contain the implementation for the class methods. For example,

- the SCL entry that defines the class using CLASS/ENDCLASS (tableSelectorButtonClass.scl)
- the SCL entry that implements the methods of the class using USECLASS/ENDUSECLASS (tableselectorButton.scl)
- the CLASS entry itself (tableSelectorButton.class)

MISCELLANEOUS SCL ENHANCEMENTS

Array Support

Previously, all SCL arrays were static. A static array is one in which the size of the array is set when you declare the array and cannot be changed at run-time. With dynamic arrays, you do not specify the size when you declare it. Instead, you can later use one of several different SCL functions to allocate and resize the array as needed in your SCL program.

As shown earlier, you can declare arrays using the new DCL statement. For example:

```
dcl num depts[5] employees[*];
```

In the above example, both arrays have been defined as numeric arrays. However, the first array `depts` is static with five elements. The second array `employees` is dynamic as indicated by the asterisk (*) inside the brackets. The size of the dynamic array must be allocated before the array's elements can be referenced or assigned.

There are five new SCL functions in Version 8 that can be used with dynamic arrays:

makeArray
: used to allocate the array.

reDim
: used to change the high bound of any dimension of a dynamic array. The data will be preserved if possible (unless going from a larger to a smaller array).

copyArray
: allows you to copy all elements from one array to another. By default, both arrays must have the same type, dimension and size; however, there is an optional parameter on this function that tells SAS/AF to allow the copy for different sized arrays. When copying from a smaller array to a larger array, elements with no values to copy from are assigned missing values. When copying from a larger array to a smaller array, the data is lost which is similar to how the REDIM function operates.

compareArray
: allows you to compare two arrays for size and data equality. To be considered equal, the array must have the same number of dimensions, the same type, the same bounds, and the same values in each element.

delArray
: used to delete dynamic arrays that have previously been allocated. It is always good programming practice to perform your own clean-up for things like arrays, lists and objects when you no longer use them. However, in the case of arrays, SAS/AF will eventually clean-up the memory used if you have not physically deleted them in your SCL (except in situations similar to the first two examples below where the memory is truly leaked and cannot be recovered, even by SAS/AF). It is also important to note that if your array elements contain lists or objects, the DELARRAY function does not delete these. You must still delete the actual lists and objects yourself.

reDimOpt
: marks an array as being implicitly resizable. Using this

feature, the array can grow in size when the value for an out-of-bound element is set without the user having to use the REDIM function first to specify the new size of the array. (Version 8.1 feature)

After you have declared a dynamic array, you can allocate the array with the MAKEARRAY function. This function allocates the array with a given size and initializes all elements in the array to missing for numeric data and blank for character data. The number of dimensions must be the same as what was specified in the declaration (each asterisk represents one dimension). The low bound for all dynamic arrays is one with the high bound being determined at run-time by the values that you specify with the MAKEARRAY function.

The following statements create a one-dimensional dynamic array named `students` that has three elements which are then subsequently initialized.

Array Example 1:

```
dcl char students[*];
students = MAKEARRAY(3);
students[1] = 'Mary';
students[2] = 'Johnny';
students[3] = 'Bobby';
```

If you use MAKEARRAY to resize the array, all data in the array will be lost. For example, if you add the following statements to the above program:

Array Example 2:

```
students = MAKEARRAY(5);
put students;
```

The results would be:

```
students[1]=''
students[2]=''
students[3]=''
students[4]=''
students[5]=''
```

Aside from having lost the data stored in the first three elements of the array that were previously set, you have created a memory leak. In the above example, you should insert a DELARRAY function call to free the memory from the first allocation.

To resize the array, use the REDIM function. With this function, you cannot change the number of dimensions or type of the array, only the bounds. The REDIM function will also preserve the data, unlike the MAKEARRAY function, unless you resize the array to a smaller size. In that case, you will lose the data in the eliminated elements.

Array Example 3 (adding to Example 1):

```
dcl num rc;
rc = REDIM(students, DIM(students)+1);
students[DIM(students)] = 'Alice';
put students;
```

The output would be:

```
students[1]='Mary'
students[2]='Johnny'
students[3]='Bobby'
students[4]='Alice'
```

You can now use SCL arrays in assignment statements just as you can any other SCL variable. The behavior is basically the same as performing a COPYARRAY function except that on assignment statements, the arrays must have the same type, dimensions, and size or an error condition will occur.

Array Example 4:

```
dcl num to[3] from[3];
to = from;
```

The assignment statement above would copy all elements from the 'from' array into the 'to' array.

Arrays can be passed as an argument between entries using the CALL DISPLAY routine. They can also be specified as arguments for a method's signature. This functionality has always been available, however, the behavior of arrays used in this manner has changed somewhat. Previously, an array passed to a method or to another program had to be defined in both the calling and the receiving program. In Version 8, arrays used in either of these situations are defined as reference arrays. A reference array is a pointer to another defined array. This means that the array in the receiving program points to the actual array defined in the calling program, using the same memory. Once a reference array has been created by a call to another program, it can be used in any way that a regular array can be used.

An array can also be assigned as the return argument for a method. The array to which the values are being assigned must have the same type, dimensions, and size as the array returned from the method. Otherwise, an error condition will occur.

And finally, both static and dynamic arrays can be defined as attributes on your class. The use of attribute arrays is the same as any declared array variable. You can pass an array attribute to a function or a method or you can use it in regular assignment statements. Just as with all other uses of dynamic arrays, it is your responsibility to both allocate and delete the array attribute in your SCL program.

<u>New Host Dialog Functions</u>

The following SCL functions have been added which display host dialog windows to perform the specific task. These windows will look and behave similar to windows found in other applications on the same host.

messageBox
: displays a host message window.

openEntryDialog
: displays a dialog that allows the user to navigate through an Explorer environment to choose a SAS data library, then catalog and see its contents. The selected catalog entry is then returned.

saveEntryDialog
: is the counterpart to the openEntryDialog with the same behavior except this window would be used when performing some kind of task that saves a catalog entry to another source.

openSASFileDialog
: displays a dialog that allows the user to navigate through an Explorer environment to choose a SAS data library, then a specific SAS file. The user's selection is returned.

saveSASFileDialog
: is the counterpart to the above function except that this window would be used when performing some kind of task that saves a SAS file to another source.

Along with these new functions, the user interface for many of the existing SCL functions that display dialog windows has been improved. The CATLIST, DIRLIST, FILELIST and LIBLIST windows have all been replaced with new host dialog windows. New frames, which offer a more graphical look, were also designed for the following functions: LISTC, LISTN, SHOWLIST, VARLIST, DATALISTC, DATALISTN.

OTHER SAS/AF ENHANCEMENTS WORTHY OF NOTING

Although the features mentioned in this section are not SCL specific, they are significant to the overall usability of SAS/AF software and to the end results that you can deliver in your finished application.

Some of the SAS/AF highlights that began with Version 7 include

- A new set of visual components, such as the Combo Box, Spin Box, and Text Pad Controls just to name a few, can be used to build FRAME applications. They offer a native look and feel as they are true host controls. Your SAS applications will look and behave just as any other native application would on the targeted host.
- New non-visual components, referred to as models, can be used to establish model/view communication between two components. For example, you can drag a Data Set List Model on top of a list box in a frame and have the items automatically display a list of data sets from a specific SAS data library. Model/view communication requires absolutely no programming on the part of the frame developer.
- The ability to easily create objects in your frame, including non-visual components, using drag and drop through the new Components window.
- A new Properties window that is available for all objects in the frame, including non-visual components. Using this window, a component's properties including attributes, methods, events and event handlers can be easily reviewed and managed. Common attributes across multiple objects in a frame can be set all in a single step.
- Communication between components can be established easily through attribute linking available automatically with all SCOM components. Using attribute linking, you can create a frame w/ a list of graphs and when the user selects a graph, have that graph automatically displayed in a graph output control on the same frame. Again, this entire frame can be built with no SCL programming.
- Miscellaneous build-time enhancements which include
  - standard multi-select of visual controls in your frame so that you can quickly set common attributes or perform cut/copy/paste actions within the same or to a different frame
  - control tab order using a new Tab Order dialog
  - use one of several region alignment tools to quickly position and align your objects.
- A greatly enhanced Class Editor that enables you to interactively manage all of the properties of your class.

Refer to the documentation and additional SUGI papers mentioned at the end of this paper under REFERENCES for more detail on the above topics.

EXCITING ATTRACTIONS TO COME WITH VERSION 8.1

Soon, Version 8.1 will be here and with this release some powerful new components will be available.

- Table Viewer Control
- Form Viewer Control
- SAS Data Set Model
- SCL List Model

These models and viewers have been designed under the SCOM umbrella and offer a full suite of attributes for customizing their appearance and behavior. They have also been designed to communicate with each other through model/view, which will allow you to build sophisticated data browsing and editing applications with little to no SCL code! These components basically replace the need to use legacy objects such as the Data Table and Data Form objects.

The Progress Bar control is new and draws a visual indicator on the frame that can be used to show the progress of some process that is taking place. Other new components that you will see are the Catalog Entry Viewer, External File Viewer and Image Viewer controls. All were designed under the SCOM architecture, which means they too have more to offer than their Version 6 counterparts. You'll want to check them out for sure!

CONCLUSION

As you can see, SAS/AF has gone through some significant changes -- specifically in the area of the SAS Component Language.

It is important to note, however, that a tremendous amount of effort has been made on the part of SAS Institute to ensure that your existing applications will run smoothly as you move between releases of the software. The features mentioned in this paper are not required changes that you have to make but rather changes that we hope you will choose to make which will bring you the following benefits:

- a simpler coding style through dot notation which reduces the amount of typing you have to do and makes your code easier to read and maintain
- stronger object-oriented design and techniques through the new SAS Component Object Model (SCOM) which encourages code reuse
- improved compile-time validation, error handling for run-time failures and overall application performance.

REFERENCES

SAS Institute Inc. (1999), SAS Guide to Applications Development, First Edition, Cary, NC: SAS Institute Inc.

SAS Component Object Model (SCOM) in Version 7 of SAS/AF Software written by Tammy Gagliano and Sue Her for SUGI24.

Version 7 SAS/AF Software - The New Component Technology written by Glen Walker and Tammy Gagliano for SUGI23.

Dressing Up Your Version 6 Objects to be Version 7 Components, written by Glen Walker and Tammy Gagliano for SUGI23.

ACKNOWLEDGMENTS

Many thanks go out to the folks that helped review and provide content for this paper: Glen Walker, Corey Benson, Chad Ferguson, and Sue Her.

CONTACT INFORMATION

If you have any questions or comments, please feel free to contact us.

Carl LaChapelle
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
Work Phone: (919) 677-8000, extension 7712
Email: Carl.LaChapelle@sas.com

Tammy Gagliano
SAS Institute Inc.
Work Phone: (630) 724-9496
Email: Tammy.Gagliano@sas.com