Object-Oriented Graphical User Interface (OOGUI) Using FRAME Entries in SAS/AF * Software
Part IV
David J. Biesack, SAS Institute Inc., Cary, NC

ABSTRACT
This tutorial continues the discussion of object oriented programming when developing SAS/AF® FRAME entries and other object oriented applications with SCL. Examples will show the following features of the software:
- Create custom subclasses of SAS/AF FRAME widget classes that provide custom behavior and reduce programming;
- Write custom attribute screens so that you can modify the BUILD procedure and reduce application programming;
- Send information from one object to another;
- Create utility classes and objects.

INTRODUCTION
As shown in Part III of this series, a common component of many SAS/AF applications is the selection of an entry from a catalog. Rather than repeat code in each screen for processing selection lists and field validation, we introduce two classes of widgets that encapsulate the desired behavior and that we can add to many FRAME entries. We will show how to extend the BUILD procedure by writing a custom attribute screen for each of the new widget classes. These attribute screens provide a way to specify object attributes when you build a FRAME entry, instead of modifying object attributes through SCL programming statements when the user runs the application.

A second example shows how one object, a selectable button, can request another object to perform an action when the button is clicked. This communication link can be established either when you execute a FRAME entry or at the time you build the entry. These classes demonstrate how you can code the generic behavior of a widget one time and reuse that widget without further coding.

Finally, we will show how you can create other reusable classes which do not have a user interface component: they are not widgets, but data objects. Programming with objects will give you the flexibility of changing the interface to your application without having to rewrite it completely.

CREATING A GENERIC SELECTION POPUP WIDGET
The Catalog Entry field CATENTRY.CLASS introduced in Part III allows an advanced user to type the name of a known entry. This section shows how to extend the Catalog Entry class to make it more usable. We also show how to create a new control button class SELARROW.CLASS that allows point and click ability to choose a catalog entry; click on the Selection Arrow and the FRAME entry displays a selection list from which the user can select an entry. The Catalog Entry and Selection Arrow classes will work in concert to provide a simple interface for selecting an existing entry from a catalog.

We want to present the same selection list to the user if they enter a '?' anywhere in the field. An example is shown in Display 1. The field containing 'HBAR2 ?' is a Catalog Entry field; the arrow next to it is a Selection Arrow. A '?' entered in the field causes SAS/AF software to display the selection list. Clicking on the Selection Arrow also results in the selection list appearing.

Display 1
We choose to implement this scenario by providing a 'PROCESS ?' method for the field that handles a '?' value entered by the user or a click on the Selection Arrow. The method will invoke the CATLIST function. The Catalog Entry widget has two character instance variables, CATLIST and ETYPE; that identify the SAS® catalog where entries are found and the entry type. This implementation will allow a Selection Arrow to work with any widget that supports a 'PROCESS ?' method. The 'PROCESS ?' method is shown below. Recall that the CATENTRY class defines two automatic instance variables, ETYPE and CATLIST. The special variable _SELF_ is the identifier of the Catalog Entry widget.

```sas
/* Catalog Entry field 'PROCESS ?' method */
length etype $8 catalog $17; /* Inst. vars */
length catentry entry $17; /* temp vars */
PROC مقابل: method;
   catentry=catlist (catalog, etype, entry);
   if catentry $= "" then return;
   entry = scan (catentry, 1, 'N');
   if catentry= ' ' then return;
   call send (_self_, '_set_text_', entry);
ENDMETHOD;
```

Next, we must invoke this method at the correct time: when the user types a '?' in the field or when the user clicks on the Selection Arrow. Recall that in Part III, we wrote a custom _VALIDATE_ method for our Catalog Entry field. SAS/AF software executes the _VALIDATE_ method each time a user modifies a field, so this is an appropriate place to check for a '?'. We will modify that method as shown below:

```sas
/* Catalog Entry field _VALIDATE_ method */
length catentry entry $17 etype _value_ $8; /* temp vars */
length fullname $30;
VALIDATE: method;
   /* Call the VALIDATE method of the parent class */
   call send (_self_, '_erroroff_');
   _value_ = _value_;  
```

1404
User requests help by entering a '?' in the field.

```sas
if index(_value_, '?') then do;
call send(_self_, 'process ?');
return;
end;
call super (_self_, '_validate_');
call send (_self_, '_in_error_', inError);
if inError then return;
endmethod;
```

/* Validate entered value */

```sas
if _value_ ne _blank_ then do:
call send(_self_, 'get_fullname', fullname);
if cexist(fullname)=O then do;
call send(_self_, '_erroron_');
call send (_frame_, '_set~sg_', 'ERROR: Catalog entry ' || fullname || ' not found.');
return;
end:
end:
end:
endmethod;
```

The Selection Arrow class is defined in CLASSES.SELARROW.SELARROW.CLASS and has the following methods:

<table>
<thead>
<tr>
<th>Method Name</th>
<th>SCL Entry</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>SELECT</em></td>
<td>SELECT.SCL</td>
<td>SELECT</td>
</tr>
<tr>
<td>SET_FIELD</td>
<td>SETFIELD.SCL</td>
<td>SETFIELD</td>
</tr>
</tbody>
</table>

We add each of these class entries to the RESOURCE entry so that they become available when building a FRAME entry. To use the new classes, edit a new FRAME entry GRAPH4A.FRAME, select Make from the popup menu and select Catalog Entry and give the name GRNAME in the attribute screen and select the OK button. Next to the field, use the Make operation to create a Selection Arrow and name it SELARROW. Finally, create a SAS/GRAPH® Output object named GRAPH. Below is the code for entry GRAPH4A that allows users to browse the GRSEG entries in the catalog SUGI.GRAPHS:

```sas
INIT:
call notify('grname', 'set_catalog', 'SUGI.GRAPHS');
call notify('grname', 'set_entry_type', 'GRSEG');
call notify('selarrow', 'set_field', 'grname');
return;
```

The INIT: section specifies the catalog and entry type that the field will allow. The SET_FIELD method links the Selection Arrow to the field GRNAME. Each time the user enters a '?' in the GRNAME field or clicks on the Selection Arrow, the CATLIST function displays a list of GRSEG entries in the SUGI.GRAPHS catalog, and when the user selects one of the graphs, it is displayed in the SAS/GRAPH output area on the screen.

We could create other FRAME entries that used the Catalog Entry field and the Selection Arrow combination to select other catalog entries and perform other actions after the user selects the entry. We do not have to code any of the selection list calls or field validation in any FRAME entry that uses these classes.

**WRITING CUSTOM ATTRIBUTE SCREENS**

The above steps allow us to create a FRAME entry with enough default processing so that the FRAME SCL is quite simple. However, we can simplify it further if there is a way to specify some of the widget attributes at build time rather than through programming statements as shown in the INIT: section.

SAS/AF software allows you to customize the build environment so you can do just that. For widget subclasses you create, you can provide custom attribute screens that are extensions of the default attribute screens, or you can replace the default attribute screen. We will take advantage of this feature to define the catalog name and entry type for the Catalog Entry field, and we will also provide a way to link the Selection Arrow to the field at build time, eliminating the need for the method calls in the INIT: section completely.
By selecting the Set custom attributes... button in the Class Editor, the window in Display 2 appears in which you specify a SAS/AF entry to display from the primary attribute screen for the Catalog Entry field.

Display 2

SAS/AF software passes the identifier of the widget you are editing to your custom attribute screen, which is responsible for modifying the widget. Each widget is represented with an SCL list. Many SCL functions exist for accessing and modifying SCL lists; see SAS Technical Report P-216, SAS/AF Software, SAS/FSF Software, and SAS Screen Control Language: Changes and Enhancements, Release 6.07 for more information.

Since we wish to edit the CATALOG and ETYPE character string instance variables of the Catalog Entry field, our custom attribute screen will have text fields (DEFAULTC and DEFAULTE) for entering these values and each field will have a control widget (ARROW_C and ARROW_E) for displaying a popup list of valid values. The screen will also have several text labels to identify the entry fields. The screen is shown in Display 3.

Display 3

The source for the attribute screen is shown below.

```
entry optional= _widget_ 8 _uattr_ $ _class_ 8;

INIT:
/* init fields with widget's instance vars */
defaultc=getnitemc(_widget_, 'CATALOG');
defaulte=getnitemc(_widget_, 'ETYPE');
control error;
clist=makelist();
rc=fillist('catalog',
'classes.catentry.etypes.list',clist);
return;

ARROW_C:
/* Choose an existing catalog */
defaultc=dirlist('!', 'CATALOG', 1, 'Y');
return;

ARROW_E:
item=popmenu(clist, 6, 3, 50);
if item then do;
defaulte=getnitemc(clist, item);
erroroff defaulte;
end;
return;

MAIN:
if error(defaultc) then
_msg_='Not a valid catalog name.';
return;

TERM:
rc=dellist(clist);
if _status_='C' then return;
_widget_ =setnitemc(_widget_, defaultc, 'CATALOG');
_widget_ =setnitemc(_widget_, defaulte, 'ETYPE');
return;
```

Similarly, we could write a custom attribute screen for the Selection Arrow class in which you specify the name of the field. By initializing this instance variable at build time, there is no need to include a method call like the following

```
call notify('selarrow', 'set_field', 'name');
```

in each FRAME entry that uses a Selection Arrow. This initialization can be done at BUILD time without requiring programming statements.

This attribute screen, ATTRSELFRAME, contains a text entry field named FIELD and a control named ARROW.

The source is listed below:

```
entry optional= _widget_ 8 _uattr_ $ _class_ 8;

INIT:
field = getnitemc(_widget_, 'FIELD');
aname = getnitemc(_widget_, 'NAME');
fieldNames = makelist();
/* Build a list of widget names in the FRAME */
frame = getnitemc(_widget_, '_frame_');
objects = getniem(frame, 'OBJECTS');
do i = 1 to listlen(objects);
    name = nameitem(objects, i);
    if name not eq name then
        rc = insertc(fieldNames, name, -1, name);
end;
return;

TERM:
rc = dellist(fieldNames);
if _status_='C' then return;
_widget_ =setnitemc(_widget_, field, 'FIELD');
return;

ARROW:
choice = popmenu(fieldNames);
if choice then do;
    erroroff field;
```
This program takes advantage of the fact that a widget has the identifier of the FRAME entry in the instance variable _FRAME_ and each FRAME object has a sublist of widgets named OBJECTS. The names of the widgets, with the exception of the name of the Selection Arrow widget, are placed in the SCL list FIELDNAMES. When the ARROW control is selected, the POPMENU function displays this list and the user selects a field name. The selected name is stored in FIELD. If the user modifies FIELD directly, the new name must exist in the list FIELDNAMES, or there is an error.

To change the class so that it uses the new attribute screen, edit the BUILD RESOURCE list and select the Selection Arrow in the list box, then choose Edit from the pop menu to edit the class SELARROW.CLASS. Select Set Custom Attributes, and enter the name CLASSES SELARROW ATTRSEL.FRAME. When you OK from the Class Editor, your changes will be reflected in the resource list. Then, OK from the Resource Editor to save the changes to the resource list.

Once our classes are in the resource list, we can edit a second FRAME entry, GRAPH4B.FRAME, which uses them. Create a Catalog Entry widget named GNOME, select the Custom attributes screen, and set its catalog to SUGI GRAFHS and the entry type to GRSEG. OK from the custom attributes screen and from the primary Text Entry attributes screen. Make a Selection Arrow named SELARROW and invoke its custom attribute screen. Click in the control arrow and a list containing GNOME appears; select GNOME. This will cause the Selection Arrow SELARROW to send the 'PROCESS?' method to the widget named GNOME when the user clicks on the Selection Arrow. OK from the two attribute screens. Finally, create a SUGI/GRAPH Output object called GRAPH. When you are done, GRAPH4B.FRAME will look just like GRAPH4A.FRAME. Enter the following SCL program for GRAPH4B.FRAME:

```
MAIN:
  if gname = blank then return;
  call notify('gnome', 'get_fullname', graph);
  return;
```

Looking at the implementation of the Selection Arrow class, you may notice that there is a change we could make that will make it even more reusable and generic. Instead of unconditionally sending the 'PROCESS?' method, we could make the method name an instance variable of the class and allow you to set the method name instance variable in the the custom attribute screen ATTRSEL.FRAME. Then the same class can be used in several different programs, even if the objects they link to do not all have a 'PROCESS?' method. This implementation decouples the two classes completely, resulting in a more reusable set of classes.

**UTILITY CLASSES**

Not only can you create custom classes and objects based on the many widgets provided with SAS/AF Software FRAME entries, you can also create classes which have no visual representation at all. Such classes are useful as programming data structures with specific functions (methods) which operate on the data structure. Or, your class may model some real world object for your application, such as a checking account, a printer, or a stock portfolio. Once you define a data structure class, you can create further subclasses of it if you need to specialize its behavior. For example, you may create a checking account with interest which has additional behavior: 'POST INTEREST' and additional instance variables, such as INT RATE.

We will show how to define simple data structure classes to model a simple appointment calendar. A calendar contains a list of appointments, each of which has a time, date or day, a duration, and brief and long notes describing the appointment. We want methods for the calendar to schedule new appointments, remove appointments, list appointments for a day of the week, date, or range of days.

Upon closer inspection of the problem, we decide there are two different data types: appointments and calendars. A calendar contains one or more appointments. We will create two catalogs in the CLASSES library: APPOINT and CALENDAR, and store the CLASS and SCL entries for each class in their respective catalog.

**The APPOINT Class**

The APPOINT class has the following methods:

- **SET_TIME**(day, date, start_time, end_time) sets the day or date of the appointment and the start and end times.
- **GET_TIME**(day, date, start_time, end_time) Retrieves the day, date, and start and end times.
- **SET_DESCRIPTION**(desc) Sets the description of the appointment
- **GET_DESCRIPTION**(desc) Retrieves the description of the appointment.
- **PRINT**() Prints an appointment
- **REFERENCE()** increments a usage count
- **DEREFERENCE()** decrements a usage count. If the usage count falls to 0, the appointment is deleted.

An appointment will have the following instance variables:

- **DAYOFWEEK** the day of the appointment, "SUNDAY" through "SATURDAY", or "" if it is a specific date.
- **DATE** the date of the appointment (a SAS date value), or missing if the appointment is for a specific day of the week
- **START_TIME** the start time, a SAS time value
- **END_TIME** the end time, a SAS time value
- **DESC** the description of the appointment.
- **REFCOUNT** count of the number of calendars this appointment appears in.

None of the APPOINT class' instance variables are automatic, which means that their values must be explicitly fetched from the object using SCL list functions. To update the instance variable, another SCL list function must be used. Automatic instance variables make coding a method easier, but their use must be documented clearly since it may become easy to confuse automatic instance variables from other variables used in an SCL method.

By listing the instance variables for these classes, we are merely specifying an implementation. An application which uses these objects will not use the instance variables at all – the application only interacts with the objects through their methods.

To create the APPOINT class, we first invoke the BUILD procedure on the catalog CLASSES.APPOINT, then edit the entry
APPOINT.CLASS. We specify the parent class of this new class as SASHELP.FSP.OBJECT, as shown in Display 4.

**Display 4**

Next, we select Instance Variables and enter the list of instance variables, as shown in Display 5. Here, we have selected the instance variable END_TIME and can see that it is Numeric and it is New for this class (it was not inherited from the parent class). We reuse the DESC instance variable inherited from the OBJECT class to hold the appointment description.

**Display 5**

After we finish editing the instance variables, we define the methods for the class. All the methods for the APPOINT class will be located in SCL entries in the CLASSES.APPOINT catalog. Display 6 shows the screen in which you edit the methods for a class; we are specifying the new method SET_TIME.

**Display 6**

The methods for the appointment class are shown below:

```plaintext
length _self_ 4;
/* call send(appt, 'get_description', desc); */
GET_DESC:
method desc $;
desc = getnitemc(_self_, 'desc');
endmethod;
/* call send(appt, 'set_description', desc); */
SET_DESC:
method desc $;
_self_ = setnitemc(_self_, desc, 'desc');
endmethod;
/* call send(appt, 'get_time', day, date, start_time, end_time); */
GET_TIME:
method day $ date start_time end_time 4;
    day = getnitemc(_self_, 'DAYOFWEEK');
    date = getnitemc(_self_, 'DATE');
    start_time = getnitemn(_self_, 'START_TIME');
    end_time = getnitemn(_self_, 'END_TIME');
endmethod;
/* call send(appt, 'set_time', day, date, start_time, end_time); */
SET_TIME:
method day $ date start_time 4 optional= end_time 4;
    if not end_time then end_time = start_time;
    _self_ = setnitemc(upcase(trim(left(day))),
                      _self_, 'DAYOFWEEK');
    _self_ = setnitemn(_self_, date, 'DATE');
    _self_ = setnitemn(_self_, start_time, 'START_TIME');
    _self_ = setnitemn(_self_, end_time, 'END_TIME');
endmethod;
/* call send(appt, 'print'); */
length text 200;
print:
method;
text = '';
call send(_self_, 'get_time', day, date, start_time, end_time);
if date then
    text = put(date, date.); 
else
    text = day;
text = substr(text || put(start_time, time.); '' to '' put(end_time, time.));
prompt;
text = ''
    getnitemc(_self_, 'DESC');
    put text;
endmethod;
ADDREF:
method;
delta = 1;
link ref;
endmethod;
```
The calendar class methods are shown here. Note that it uses the methods of the appointment object to manipulate the appointments that it contains.

The CALENDAR Class

The list below shows the methods for the CALENDAR class:

- **NUM_APPOINTMENTS(num)** returns in num the number of appointments in the calendar
- **PRINT()** prints the contents of the calendar
- **ADD_APPOINTMENT(appt)** adds an appointment to the calendar. The calendar allows conflicting appointments.
- **DELETE_APPOINTMENT(appt, i)** deletes an appointment or the ith appointment if i is specified.
- **CLEAR(day, date)** clears all appointments for a day or date (either or both may be missing). When clearing a date, weekly appointments which happen to fall on that date are not cleared; when clearing a day, date appointments which occur on that day are not cleared. For example, clearing appointments for May 12, 1993 should not clear all Wednesday appointments.
- **LIST(day, start_date, end_date, start_time, end_time, cal)** Adds to the second calendar cal the set of appointments for the specified day or range of dates. Only appointments which overlap the start_time to end_time are listed, but if start_time and end_time are missing, all appointments for the matching days are added.
- **GET(i, appt)** returns the ith appointment in the calendar, or 0 if i is out of range.

Note that the LIST method extracts a subset of a calendar’s appointments and adds them to a secondary calendar CAL. Recall that the calendar object on which the LIST method is invoked is identified by the special variable _SELF_ in the SCL source code for the method.

In addition to these new classes, the CALENDAR class will override the _INIT_ and _TERM_ methods. The _INIT_ method runs automatically when an instance of the class is created. Its purpose is to initialize the object. For the CALENDAR class, the _INIT_ method will create a new SCL list APPOINT to store the appointments. The _TERM_ method is used to destroy an object. We override the _TERM_ method for the CALENDAR class so that we can delete the APPOINT list and also dereference any appointments stored in the calendar. When you override the _INIT_ or _TERM_ methods, you must invoke the inherited method with CALL SUPER; see the INIT and TERM labels in the source.

An appointment calendar should have other methods, such as methods for checking for conflicts, searching for appointments by text substrings, reading and writing the calendar to a file, and so on. We have omitted such methods for the sake of simplicity.

Calendar has one instance variable, APPOINT, which is an SCL list of appointments. The APPOINT instance variable is marked as Automatic which means that, when a CALENDAR method is executing, the APPOINT variable in the SCL program is automatically initialized with the value of that instance variable.

Also, assignments to the corresponding SCL variable will be copied back into the object when the method finishes.

length _self_ 4;
length date start_time end_time 4;
length appointments appt 4;
length day $9 astart send date 4;
length days $70;

/* Override _INIT_ method: initialize lists */
INIT:
method optional= arg 4;
call super(_self_, '_init_', arg);
apoint = makelist();
endmethod;

TERM: /* Override _TERM_ method */
method;
do i = listlen(appoint) to 1 by -1;
appt = getitem(appoint, i);
appointements = delitem(appoint, i);
call send(appt, 'dereference');
end;
rc = dellist(appoint);
call super(_self_, '_term_');
endmethod;

/* call send(calendar, 'print'); */
PRINT:
method;
do i = 1 to listlen(appoint);
appt = getitem(appoint, i);
call send(appt, 'print');
end;
endmethod;

/* call send(calendar, 'add', appt); */
ADD:
method appt 4;
appoint = insertn(appoint, appt, -1);
call send(appt, 'reference');
endmethod;

/* call send(calendar, 'delete', appt, index); */
DELETE:
method appt 4 optional= index 4;
if not index then
    index = searchn(appoint, appt);
if index $= 1
    and index $= listlen(appoint) then do;
        rc = delitem(appoint, index);
    end;
endmethod;

/* call send(calendar, 'clear', day, date); */
CLEAR:
method optional= day $ date 4;
day = upcase(day);
do i = listlen(appoint) to 1 by -1;
appt = getitem(appoint, i);
call send(appt, 'get_time',
day, aday, start_time, end_time);
if (day = " " and not date) then
del = 1;
else if (date and adate = date) then
del = 1;
else if (not date and aday = day) then
del = 1;
if del then do:
  rc = delitem(appoint, i);
call send(appt, 'dereference');
end;
day = upcase(aday);
if adate >= start_date
  and adate <= end_date then
  link chckttime;
else if index(days, aday) then
  link chckttime:
end;
endmethod;
/* call send(calendar, 'get', index, appt); */
GET:
method index appt 4:
  if abs(index) < listlen(appoint) then
    appt = getitem(appoint, index);
  else appt = 0;
endmethod;
LIST:
method day $ start_date end_date
  start_time end_time cal 4:
  /* call send(calendar, 'get', index, appt); */
  GET:
  method index appt 4:
    if abs(index) < listlen(appoint) then
      appt = getitem(appoint, index);
    else appt = 0;
  endmethod;
  return:
A Sample Application
To write an application which uses these classes, we use the
LOADCLASS function which returns a class identifier
given a class name. We can then use the class identifier to create several
instances of the class.
For example, the following code fragment creates an empty
appointment calendar:
cal_class = loadclass('classes.calendar.calendar');
cal = instance(cal_class);
Next, we load the APPOINT class and create several appointment
objects. Each of the objects APPT1, APPT2, and APPT3 are
separate objects, but they are all instances of the same class. All
of the methods of the APPOINT class apply to all three objects.
appt_class = loadclass('classes.appoint.appoint');
appt1 instance(appt_class);
appt2 instance(appt_class);
appt3 : instance (appt_class) ;

After we have created the objects, we can define the day, date, and
time of each appointment, and set the description string. After this
is done, we can add each appointment to the calendar with the
calendar's ADD_APPPOINT method:
call send(appt1, 'set_time', "Monday", ..
'13:00'T, '14:00'T);
call send(appt1, 'set_description',
'Staff meeting');
call send(cal, 'add_appointment', appt1);
call send(cal, 'set_time', "Wednesday", ..
'10:00'T, '11:00'T);
call send(cal, 'set_description',
'Project status meeting');
call send(cal, 'add_appointment', appt2);
call send(appt3, 'set_time', "", '09Jun93'D, '16:00'T, '17:00'T);
call send(appt3, 'set_description', "Dr. appointment: Dr. Howser");
call send(cal, 'add_appointment', appt3);

Finally, we can try some of the other methods of the calendar class.
We will print the initial calendar, then delete the third appointment, then clear MONDAY.

put '========== Initial calendar:
put '========== Delete appointment 3
put '========== Clear MONDAY
put '========== Clear MONDAY

The results are printed to the LOG, shown below:

========== Initial calendar:
MONDAY From 13:00 to 14:00
  Staff meeting
WEDNESDAY From 10:00 to 11:00
  Project status meeting
09JUN93 From 16:00 to 17:00
  Dr. appointment: Dr. Howser
========== Delete appointment 3
MONDAY From 13:00 to 14:00
  Staff meeting
WEDNESDAY From 10:00 to 11:00
  Project status meeting
========== Clear MONDAY
WEDNESDAY From 10:00 to 11:00
  Project status meeting

SUMMARY
We have seen how object oriented programming greatly facilitates application development with SAS/AF software. You can extend run-time behavior of widgets by overriding such methods as the _VALIDATE_ method of a text entry field to allow custom field validation. You can override the _SELECT_ method of a control object so that it notifies another object of a selection. We have also shown how you can extend the BUILD procedure environment by providing custom attribute screens which allow you to specify many default values for object attributes when you build a FRAME application. This reduces the amount of programming required to use the objects, thereby reducing development time and execution time of an application. Finally, we saw how you can create utility classes that, as subclasses of the OBJECT class, provide modular data structures with a good level of abstraction and information hiding combined with inheritance and reusability. All of these features result in a very flexible and powerful tool for creating information delivery applications with SAS software.

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