Object Oriented Programming with SAS/AF
Using Event Handlers and Extending SAS Objects

Kevin Graham, Montura Inc., Chico, California

ABSTRACT

User-defined events provide Frame/SCL developers a complete toolset that helps streamline the applications development cycle. Combining object-oriented programming with Events is easy to understand and provides an excellent way to prototype application behaviors. By extending SAS objects developers can reprogram inherited methods, events, and attributes to define/redefine object actions and secondary object reactions in a way that closely mimics JavaScript/HTML programming.

INTRODUCTION

This paper explores object programming with SAS/Frame. A series of examples will demonstrate the following:

- How to gain a high measure of control over object behaviors where content in one object is dependent on content in another object.
- How multiple objects can operate on data stored in a single memory location to take advantage of ETL methodologies.

BASIC OOP TERMINOLOGY

SAS provides a wide array of objects that can be dragged-n-dropped directly into a Frame. Many visual objects are referred to as **widgets** in SAS documentation and are also listed as **controls** in the Components window. Non-visual classes include **models,** some of which are used in combination with a visual control to display data.

- **Class:** A complete definition of variables, methods and events. Classes are compiled code and reside on a hard drive.
- **Object:** A class definition that lives in memory during program execution.
- **Instance:** A unique reference to a specific Object. Multiple objects created from a single class can exist in memory at the same time. Each object must have a unique 'name' or instantiation reference.
- **Attributes:** Variables defined within the class. Attributes have similarities to Data Step variables declared using the Length or Attrib statements.
- **Methods:** Named functions within a class. The usage of methods is similar to using the Link statement in a Data Step.
- **Events/Handlers:** Used to create event-based behaviors. Events act as a 'sender' while Event Handlers act as a 'receiver'.

LOCATE READY TO USE SAS/AF CLASSES

Create a new SAS Frame entry in a catalog. Note the **Components Window** becomes available. Each resource component relates directly to a class in SASHELP.CLASSES. (i.e., Combo Box Control relates directly to combobox_c in SASHELP.CLASSES. These components are the basic building blocks needed for object programming using SAS/Frame and SAS Component Language (SCL).

- Components Tab **Figure 1**
  Widgets and models that are available for drag-n-drop use in Frame application development. Method overrides should be used for minor adjustments. Overriding events and methods provides a minimal level of direct control over object behaviors.

- SASHELP.CLASSES **Figure 2**
  The original classes that relate one-to-one to the listing in the Components Tab. Extending classes provides the maximum level of direct control over object behaviors.

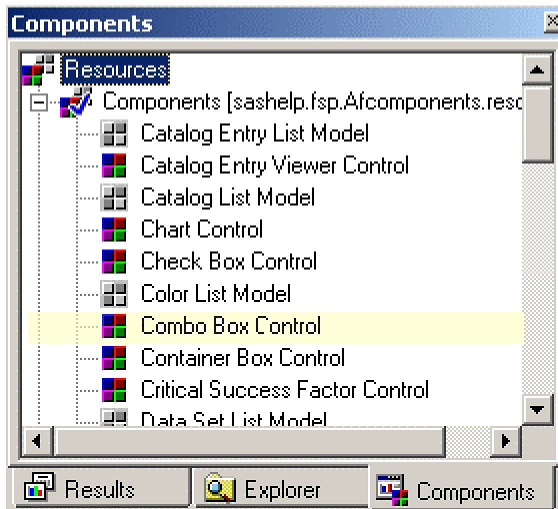


Figure 1

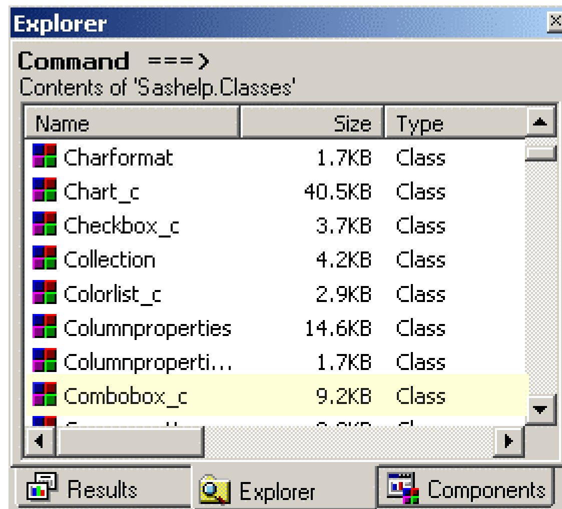


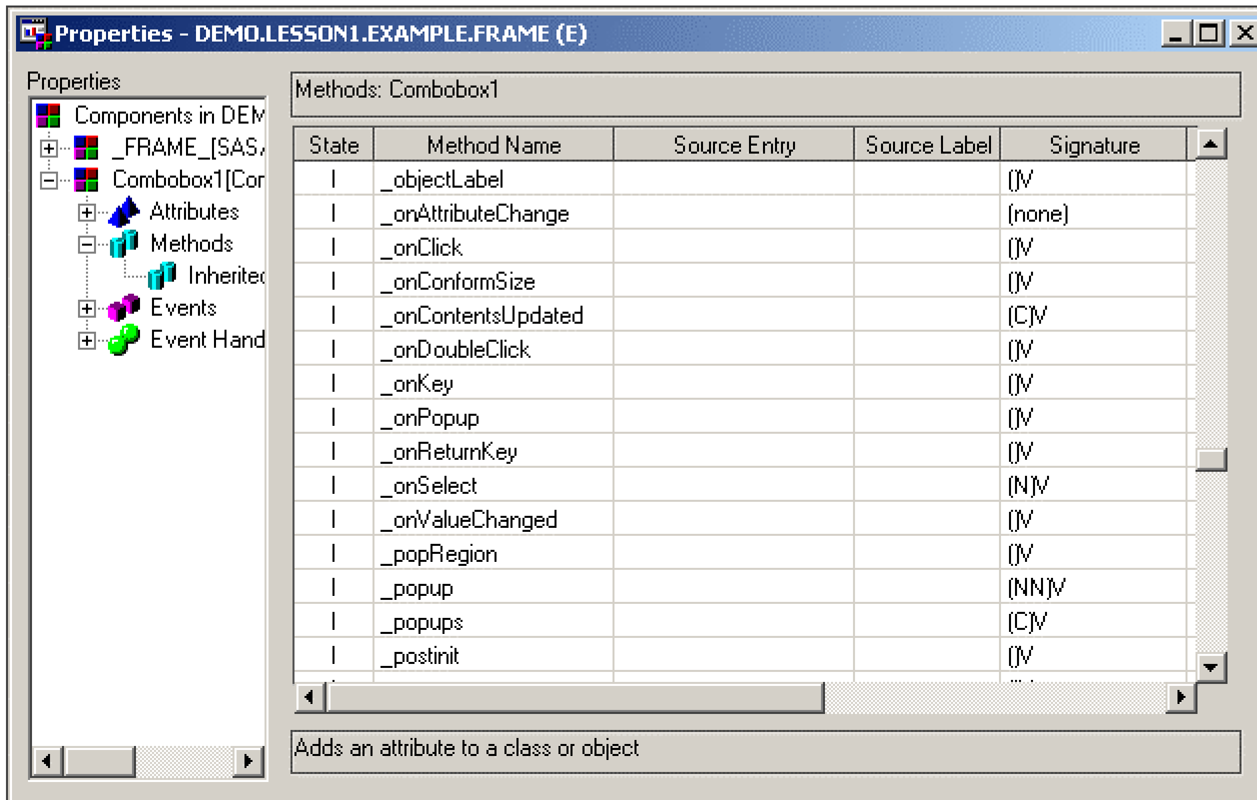
Figure 2

METHODS

Open the Properties window using a Frame with a Combo Box control (right click, select Properties). In the Frame Properties window, locate the ComboBox1 component and open **Methods, Inherited.** A complete listing of all methods associated with an object can be found in here.

SAS Objects contain inherited methods, some of which are directly associated to **events**. The Combo Box Control contains a number of methods that appear as likely candidates for detecting user actions. Some of the methods most commonly associated with user interactions with the Frame are prefixed with '_on', a naming convention also used by JavaScript. Scrolling to the right reveals *Descriptions* for each method.

Method Name and *Signature* are used to code method overrides. Signature indicates any required arguments.



EVENTS

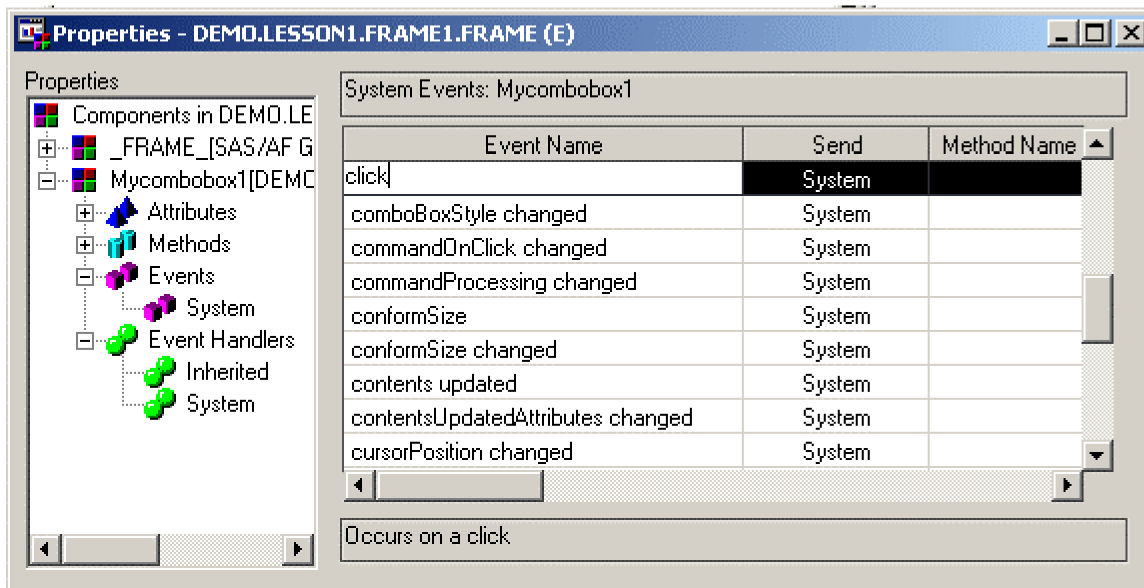
Event handling with SAS is similar to event trapping on HTML web pages, where JavaScript code embedded in HTML creates the event trigger and another JavaScript code executes as the event handler.

A common event/handler combination is activated when web surfers press a Submit button during a purchase and JavaScript code runs a series to verify charge card numbers and expiration date were properly entered.

Event based programming with SAS offers key advantages over top-down programming, where the advantages usually translate into reduced development time:

- SAS automatically notifies every object in the Frame of each Event.
- Each object can be programmed to execute a particular method in response to an Event.
- Objects can be dragged-n-dropped from Explorer window directly to Frame(s).

Right click on Frame1 in the Build window, select Properties from the pop-up menu.
Drill down to Event Handlers, System.

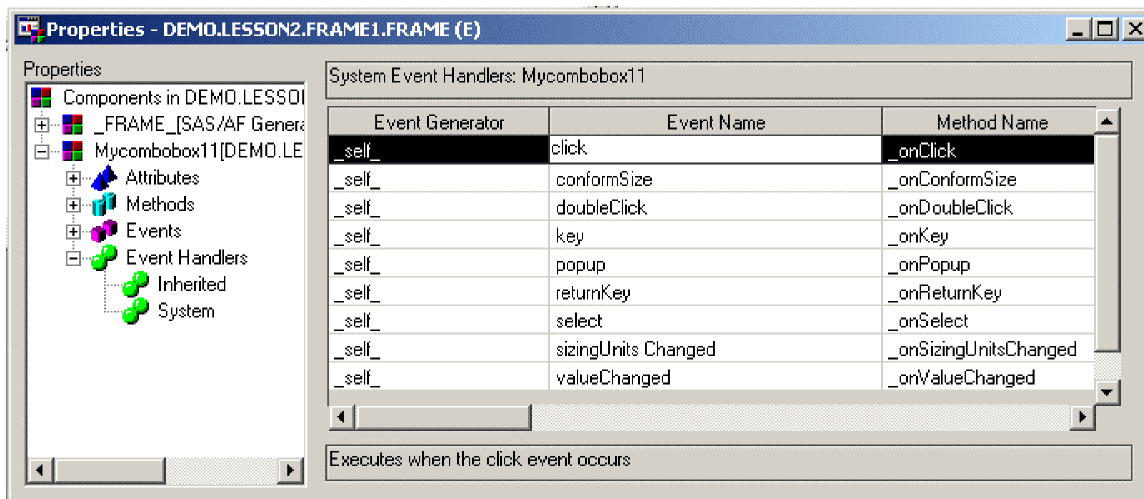


Events Definition

Events alert applications when there is a change of state to an object. Events occur when a user action takes place, when an attribute value is changed, or when a user-defined condition occurs. Events are messages sent to objects.

e.g. A user action is a mouse click

- the event name is **click**:
- the method assigned is _onClick



Event Handlers Definition

Event handlers are methods that listen and respond to Event messages. Event handlers are directives that call one method to be executed.

EVENT PROGRAMMING

Identify which methods execute when a selection is made from a Combo Box.

```
class myCombobox extends sashelp.classes.combobox_c.class;
   _init: method / (state='o');
       _self_.items={'1','2','3'};
       put 'object  initialization';
   endmethod;

   myCombobox / (state='o');
      put 'constructor method in extended classes never execute';
   endmethod;

   _onSelect: method arg:num / (state='o');
       put 'onSelect';
   endmethod;

   _onClick: method / (state='o');
        put '_onClick';
   endmethod;

   _onContentsUpdated: method arg:char / (state='o');
       put '_onContentsUpdated';
   endmethod;

   _onValueChanged: method / (state='o');
       put '_onValueChanged';
   endmethod;
endclass;
```

Testing Expected Results

1. Create a new catalog and then create a new scl program called myComboBox.scl.

2. Save and compile myCombobox.scl: ====> save;saveclass. Notice a class entry appears in the Explorer window.

3. Create a new frame called frame1. Edit frame1.scl and code the following, and then compile ====> save;comp
   init:
   return;

4. Drag and drop myCombobox.class in the Explorer window onto Frame1.frame in the Build window. Drag-n-drop using frame1.scl will have no effect.

5. Right click on Frame1 in the Explorer window, select **run** from the pull down menu that appears. Click on the Combo Box and select one item from the list.

6. From the Log Window, identify which methods were executed, and when.

7. Remove methods not needed for your application.

DATA VISIBILTY AND OBJECT VISIBILITY

The %global statement creates easy to access and update macro variables during a SAS session. In the following example, **global_protocol_id** is the Object Identifier for one macro value of unknown length.

```
init:
    dcl char protocol_id=symget('global_protocol_id');
return;
```

In the following example, **clinprog** is the Object Identifier for an abstract class with an unknown number of variables.

```
init:
    dcl clinprog clinprog =_new_ clinprog ();
return;
```

Please note the previous example is the **short** form of instantiation. The following is the explicit fully qualified example. Now it's easier to see which clinprog is the Object Identifier.

```
init:
    dcl demo.intro.clinprog.class clinprog=_new_ demo.intro.clinprog.class();
return;
```

Automatic Loading

SAS/Frame automatically performs class instantiation for **all components** listed in the Properties window **all abstract classes** attached as attributes to the Frame.

- Pro: fewer lines of code.
- Con: makes it appear as if no Object Identifiers are available.

Inter-Object Communications

Components listed in the Properties windows of a Frame behave a lot like ornery little children. They do not share (access to information). Strangely enough, it's the ability to separate program code into little segments capable of communicating with each other which is the main strength of OOP programming.

The Object Identifier for each component functions in the same manner as a Post Office Box address. Sending or receiving information to a *specific address* requires the Object Identifier, which is maintained by the Post Office.

The component attribute object is created at runtime. It cannot be found in the Frame Attributes list or the components attributes list. In fact – it's not readily found in SAS documentation. Execute **call putlist(componentObjectID)** in the code below to see its metadata contents. Substitute any component in your Properties window for "frame_component_name".

```
dcl object componentObjectID;
_frame_._getWidget('frame_component_name', componentObjectID);
```

- Object Identifier = "mail address" for an object, allows direct dot-notation access to attributes and methods in other Frame objects.

A Frame assigns every object under its control an objectid, which provides the most direct way to interact with objects. It works a lot like calling a child by name. Use a child's correct name (with milk and cookie in hand) and a response is almost guaranteed. Use an objectid (milk and cookies are all yours now) and gain guaranteed access/responses with any object.

The use of Events and objectids allow OOP style coding practices where ETL classes (programs perform database I/O) can be separated from Frame objects (programs that control screen behaviors).

CODE EXAMPLE #1

- Create a new Frame
- Copy and paste the following code into obj_combobox.scl
- Save and SaveClass

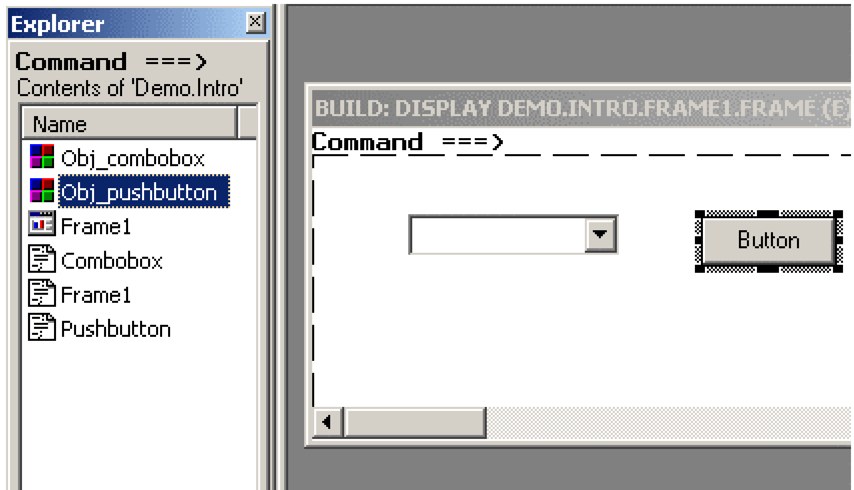
```
class obj_combobox extends sashelp.classes.combobox_c.class;
    eventhandler lock       / (sender='*', event='lockProtocol');
    eventhandler unlock     / (sender='*', event='unlockProtocol');

    _init: method / (state='o');
        _super();
        items={'1','2'};
    endmethod;

    lock: method;
        enabled='No';
    endmethod;

    unlock: method;
        enabled='Yes';
    endmethod;
endclass;
```

- Copy and paste the following code into obj_pushbutton.scl
- Save and SaveClass

```
class obj_pushbutton extends sashelp.classes.pushbutton_c.class;
    _init: method / (state='o');
        _super();

        label='Select';
        enabled='No';

        dcl object componentObjectID;
        _frame_._getWidget('obj_combobox1', componentObjectID);
        dcl num rc=_self_._addEventHandler(componentObjectID,'selectedIndex changed' ,'enabledStatus');
    endmethod;

    enabledStatus: method metadata:object;
        dcl object componentObjectID=metadata.objectid;
        if componentObjectID.selectedIndex=0 then enabled='No';
                                            else enabled='Yes';
    endmethod;

    _onclick: method / (state='o');
        select(label);
            when('Select') do;
                _sendEvent('lockProtocol');
                label='Reset';
```

```
            end;
            when('Reset') do;
                _sendEvent('unlockProtocol');
                label='Select';
            end;
        end;
    endmethod;
endclass;
```

- drag-n-drop obj_combobox
- drag-n-drop obj_pushbutton



- Copy and paste the following into the Frame SCL
- Save, Compile, and exit the Frame

```
init:
return;
```

Program Objectives

Combobox Behaviors

- Initialize the visual control, populated with clinical protocol numbers.
- Disable interactions/enable interactions with the visual control.
- Detect events originating in other objects.

PushButton Behaviors

- Initialize the visual control in the disabled state with display text ‘Select’.
- Dynamically create an Event Handler.
- Disable interactions/enable interactions with the visual control.
- Toggle display text between ‘Select’ and ‘Reset’.
- Issue two events: ‘lockProtocol’ and ‘unlockProtocol’.

Key OOP Objectives

Instantiation

- Components (programs) are automatically loaded into memory, ready for execution.

Object Interaction and Identification

- Programs are able to identify and reference attributes/values in another specified object.
- Ability to discriminately accept Events, narrowing method execution to events triggered by one specific object.

- Object Interaction: _getWidget()

  The _getWidget() method returns the objectid for one specified object. The specified object must be a component of the Frame. Object can be referenced by name (displayed in Properties Window) or by objectid (created at runtime by the Frame).

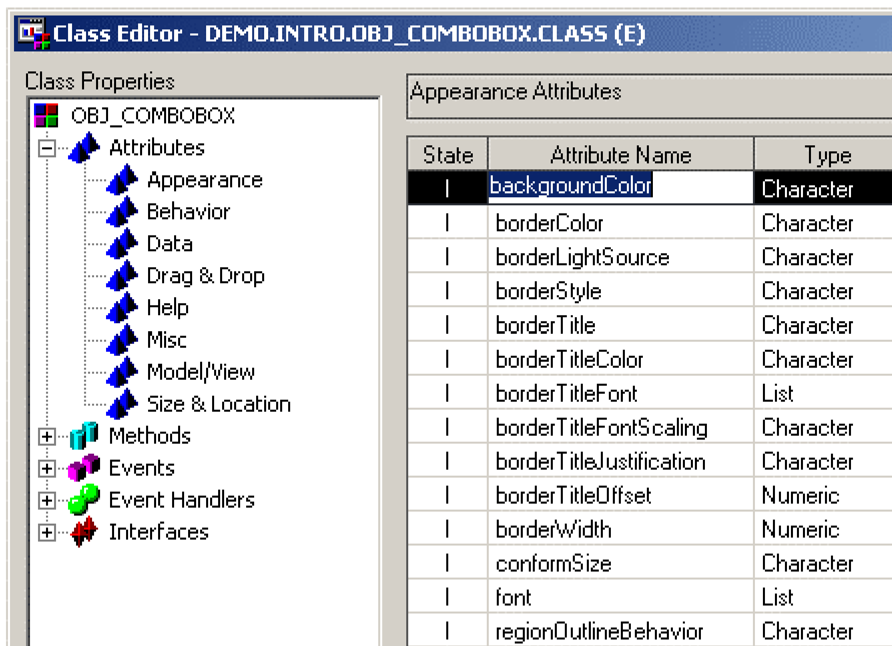
- Object Identification: metadata typecast as an object

  Metadata-objects function as pointers to original data as the original data itself is not transferred. A metadata pointer is typecast as an **object** and contains a lot more information than a pointer used in C-Language. Use **call putlist()** to see a complete display of object content in the Log Window.

  The **objectid** attribute may accessed through the typecast object, but is not located in the metadata. This value is referenced as an **inherited** attribute of the Frame. Remember, the Frame creates the objectid at runtime during object instantiation.

RESERVED ATTRIBUTE NAMES

A complete list of reserved names can be located using the Properties window. Double-click any class object and access Attributes. SAS allows categorization of Attributes (Appearance, Behaviors, Misc., etc.) This visual organization has no effect on how variables are accessed.



GENERAL METACLASS COMPONENTS

SAS allows general metaclass objects to be added as Frame components. Simply drag-n-drop the class from the Explorer window to the Build Window. Open the Frame Properties Window and note the icon associated with this type of class is the grayed-icon normally associated with **model** components.

Advantage

A general metaclass can function as a general workspace, which is where methods/data affecting the entire application can be located. Extract Transform and Load (ETL) logic coded in this component can be made available as a "universal workspace" with full read/write access for every extended object in the Frame. The following example is a general metaclass familiar to most programmers.

```
class helloworld;
    public char textVariable / (initialValue='Hello World');

    _init: method / (state='o');
        _super();
        put textVariable;
    endmethod;
endclass;
```

CODE EXAMPLE #2

Save and compile the Frame and each class, beginning with workspace. Drag-n-drop classes over the Build Window containing the Frame.

Workspace Component: This component demonstrates reading one row of data from a SAS table. Each variable read from the table is loaded into a SCL named list, which will contain every numeric and character variable found in the SAS table.

```
class workspace;
    public list databaseInfo;

    _init: method / (state='o');
        dcl num dset i rc numberCols;

        dset=open('work.protocol', 'i');
        numberCols=attrn(dset, 'nvars');
        rc=fetch(dset);
        do i=1 to numberCols;
            if vartype(dset, i)='C'
              then rc=setnitemc(databaseInfo, getvarc(dset, i), varname(dset, i));
              else rc=setnitemn(databaseInfo, getvarn(dset, i), varname(dset, i));
        end;
        rc=close(dset);
    endmethod;
endclass;
```

Text Entry Component: This component demonstrates how to dynamically create event handlers. The inherited method **_init** creates a 'listener' or event handler during object initialization using the _addEventHandler method with a signature requiring an Object Identifier. Programs dynamically create 'listeners' where each listener is restricted to events generated from one specified component.

The _getWidget method extracts Object Identifiers necessary to create event handlers and also shows this implementation has one drawback -- the instantiation name must be hard-coded.

```
class obj_textentry extends sashelp.classes.textentry_c.class;
    eventhandler workpointer / (sender='*', event='workpointer');
    public object workspace;

    _init: method / (state='o');
        _super();

        dcl object componentObjectID;
        _frame_._getWidget('obj_combobox1', componentObjectID);
        dcl num rc=_self_._addEventHandler(componentObjectID,'selectedIndex changed','onProtocolChange');
    endmethod;

    onProtocolChange: method metadata:object;
        dcl object componentObjectID=metadata.objectid;

        text=workspace.clinicalStudyFolder||'\'||
             getnitemc(componentObjectID.databaseInfo, 'reportingpath');
    endmethod;

    workpointer: method metadata:object;
        workspace=metadata;
    endmethod;
endclass;
```

Combo Box Component: This component demonstrates how visual components are populated with data shared in a metaclass object. The shared data is accessed using dot-notation though an object called workspace, which was initialized through the workpointer method.

```
class obj_combobox extends sashelp.classes.combobox_c.class;
    eventhandler workpointer / (sender='*', event='workpointer');
    public object workspace;

    _init: method / (state='o');
        _super();

        items=copylist(workspace.protocols);
    endmethod;

    workpointer: method metadata:object;
        workspace=metadata;
    endmethod;
endclass;
```

Frame SCL Component: This component demonstrates how the reference for an instantiated object is sent to other objects. The reference to *workspace1* may appear to be a numeric value, commonly called a pointer in C-Language. Event based programs rarely need SCL code located in the Frame's SCL.

```
init:
    _frame_._sendEvent('pointers', workspace1);
return;
```

Program Objectives

Combobox Behaviors

- Initialize the visual control with a list of clinical protocol numbers from a central repository of data.

TextBox Behaviors

- Detect one specific Event in another object and reference its attributes and values. Update the visual control with selected data to form an operating system path.

Key OOP Objectives

Pointers — Use a metaclass component to keep a **single copy** of data in memory instead of passing parameters.

Access to any data stored in memory remains available using standard dot-notation.

- The original object containing the shared data will appear to be passed as a parameter to other components in the Frame using _sendEvent(). In reality, only a metadata pointer is passed -- and only one copy of the data exists in memory.

- Pointers to Frame components are typecast as an object (or metadata pointer) and access to attributes and methods in that component is available by dot-notation or _sendEvent().

CONCLUSION

The SAS implementation of OOP offers an easy transition to structured programming. Interactions between objects are created easily with objects as parameters and event messages, system defined events use names that strongly resemble JavaScript functions used in web pages, and most of the functions developers learned coding Data Steps exist in SCL with the same name, or may exist with similar functionality using an improved implementation.

- Frame offers GUI developers a built-in relational database accessible by compiled SCL code, legacy SAS/Base code, and SQL.

- OOP developers reduce complex code and create prototypes faster, leaving more time to focus on business solutions.

- Events/Event Handlers function in the same manner as JavaScript embedded in HTML, providing a commonality between languages.

- Objects allow data and program code to live in memory for the life of the object. Removing critical data and program code from general-user access may bring applications into compliance with some portions of 21 CFR Part 11.

REFERENCES

SAS Institute Inc (1999) SAS Component Language: Reference, Version 8.
SAS Institute Inc (1999) SAS Online Doc, Version 8.

AUTHOR CONTACT INFORMATION

Kevin Graham
Montura Inc.

(510) 798-8367
Kevin@montura.com

400 Mission Ranch Blvd. #139
Chico, CA 95926