CHAPTER 9

Example: Creating An Object-Oriented Application

Introduction

Version 8 SCL provides many object-oriented programming features such as class and useclass syntax, method overloading, and interfaces. This tutorial demonstrates how to use many of these new features by creating a class-based version of a simple data input facility that is based on traditional SCL library functions.

The tutorial is organized in sections by level of difficulty, with Level Zero being the most elementary and Level Five the most challenging. A beginner should work through sections Zero, One, and Two. The remaining sections can then be reviewed as the user progresses. For example, a beginner may want to return to the tutorial after gaining additional experience with the basics. A user who is more experienced in object-oriented programming may want to work through them at once. Level Five should be reviewed by all users because it contains an example that shows how the SCL class syntax can be used with SAS/AF software.

Simple Class Syntax — Level Zero

Before beginning the tutorial, you must have a clear understanding of a simple SCL class. A CLASS statement enables you to use SCL to create a SAS/AF class and to define all the properties for the class, including attributes, methods, events, and interfaces. An SCL class is created with SCL class syntax. The simplest class is an empty class, which is defined using the following code:

```
class x;
endclass;
```

Enter the above code in an SCL entry such as X.SCL and then create the class by using the SAVECLASS command. You should now see a CLASS entry for X in the current catalog.

Note: The name of the entry does not have to match the name of the class, but for beginners this is the easiest way to name an entry. △

To add functionality to this class, you can create a simple method by using the following code:

```
class x;

 m: method;
    put 'Hello';
 endmethod;

endclass;
```

The PUT statement will write **Hello** to the SAS procedure output file or to a file that is specified in the most recent FILE statement. To run this method, you need to create an example of the class and call the method. In an SCL entry called Y.SCL, enter the following code:

```
  init:
    dcl x x = _new_ x();
    x.m();
    return;
```

The _NEW_ operator is used to create an example of the class X and to assign it to the variable *x*. The _NEW_ operator provides a faster and more direct way to create an object by combining the actions of loading a class with LOADCLASS and initializing the object with the _new method, which invokes the object's _init method. You then call method M using the object variable *x* in the dot notation expression x.m().

Note: dot notation provides a shortcut for invoking methods and for setting or querying attribute values. Using dot notation reduces typing and makes SCL programs easier to read. It also enhances run-time performance if you declare the object used in the dot notation as an example of a predefined class instead of a generic object. The object's class definition is then known at compile time. Because dot notation checks the method signature, it is the only way to access an overloaded method as illustrated in "Overloaded Methods — Level Two" on page 145. △

Compile Y.SCL, and then use the TESTAF command to run it. You should see the following output:

```
Hello
```

Creating a Data Set Class — Level One

In this exercise you will read a simple SAS data set that contains two character variables. You will learn how to open the data set, fetch an observation, copy a data set character variable, and close the data set. The SCL functions that will be used are

- □ OPEN
- □ FETCH
- □ GETVARC
- □ CLOSE

You will build a class that encapsulates these operations so that they can be called as methods on any given data set. Because of this, the class itself should represent a data set.

Before you can begin converting these functions to class methods, you must create the class data, as shown in the next section, "Class Data — Level One" on page 139.

Class Data — Level One

Class data is declared with the DCL statement by using the following code:

```
class x;
  dcl num n;
endclass;
```

Here class X contains a numeric variable called *n*.

The DCL statement can be omitted when you provide a variable scope modifier such as public, private or protected. *Scope modifiers* indicate how the variable is to be accessed from locations outside the class. By default, the scope modifier is public, which indicates that anyone can access the variable. Both private and protected scope modifiers restrict access to the variable.

The Data Set Class — Level One

For the data set class, begin with the following class data:

```
DDATA class;
public-string-dname;
public-string-mode;
protected-num-fid;
protected-num-nvars; endclass;
```

where

dname	is the name of the data set
mode	is the access mode
fid	is the file identifier that will be returned from the OPEN call
nvars	is the number of variables in the data set.

In this case, public access is given to *dname* and *mode*, but access is restricted for *fid* and *nvars*.

You will create one method for each of the SCL functions OPEN, FETCH, GETVARC, and CLOSE. The following example shows how to use the FETCH function to take an action that is based on the value of its return code:

```
 read: method return=num;
       dcl num rc;
       rc = fetch(fid);
       return rc;
 endmethod;
```

You can use a function such as GETVARC directly in the IF statement. In this case, the VARTYPE function is executed, and then the IF expression evaluates the return code to determine whether to perform the conditional action. The following method takes a parameter *n*, which represents the variable number, and returns the character variable *c* from GETVARC.

```
cpy: method n: num return=string;
     dcl string c = "";
     if (vartype(fid, n) = 'C') then
        c = getvarc(fid, n);
     return c;
endmethod;
```

CLOSE is used to close a data set as soon as it is no longer needed by the application. The method in this example for CLOSE is

```
_term: method /(state='O');
       if (fid) then close(fid);
       _super();
endmethod;
```

This method closes the data set represented by *fid*. It also contains two items that refer to the parent class of DDATA (State='O' and _super()). A *parent class* is the class from which a particular class was extended or derived. In this case, DDATA was implicitly extended from OBJECT.CLASS. Since OBJECT.CLASS contains a _term method, you must indicate that you are overriding it in DDATA by specifying State='O'. Because OBJECT.CLASS is being overridden, to ensure that the _term method in OBJECT.CLASS is still executed, use the function call _super().

Constructors — Level One

The method that will be used for opening the data set is called a constructor. A *constructor* is a method that is used to instantiate a class and provides a way for initializing class data. In order for a method to be a constructor, it must be a void method (one that does not have a return value), and it must have the same name as the class. Here is the constructor for DDATA:

```
ddata: method n: string m:string nv:num;
        fid = open(n, m);
        dname = n;
        mode = m;
        nvars = nv;
 endmethod;
```

where *n* is a parameter containing the name of the data set, *m* is the input mode, and *nv* is the number of variables.

This constructor method will be called when an example of the DDATA class is created using the _NEW_ operator. For example, the following code creates an example of the DDATA class representing the data set **sasuser.x**. The data set will be opened in input mode and has two variables.

```
init:
 dcl ddata d = _new_ ddata("sasuser.x", "i", 2);
 return;
```

Using the Data Set Class — Level One

The entire data set class is

```
class ddata;

  /* Data */
  public string dname;
  public string mode;
  protected num fid;
  protected num nvars;

  /* Constructor method */
  ddata: method n: string m:string nv:num;
         fid = open(n, m);
         dname = n;
         mode = m;
         nvars = nv;
  endmethod;

  /* FETCH method */
  read: method return=num;
       dcl num rc;
       rc = fetch(fid);
       return rc;
  endmethod;

  /* GETVARC method */
  cpy: method n: num return=string;
      dcl string c = "";
      if (vartype(fid, n) = 'C') then
         c = getvarc(fid, n);
      return c;
  endmethod;

 /* CLOSE method */
  _term: method /(state='O');
        if (fid) then close(fid);
        _super();
  endmethod;
endclass;
```

You can use this class as follows:

```
init:
 dcl ddata d = _new_ ddata("sasuser.x", "i", 2);
 dcl num more = ^d.read();
 do while(more);
    dcl string s s2;
    s = d.cpy(1);
    s2 = d.cpy(2);
    put s s2;
    more = ^d.read();
 end;
 d._term();
```

```
   return;
```

In this example, the data set **sasuser.x** has two character variables, which you read and print until the end of the file is reached.

Now suppose that you create the following data set:

```
data sasuser.x;
 input city $1-14;
 length airport $10;
 if city='San Francisco' then airport='SFO';
 else if city='Honolulu' then airport='HNL';
 else if city='New York' then airport='JFK';
 else if city='Miami' then airport='MIA';
 cards;
 San Francisco
 Honolulu
 New York
 Miami
 ;
```

The output from the program will be

```
San Francisco SFO
Honolulu HNL
New York JFK
Miami MIA
```

Extending Classes — Level Two

While designing the class structure, you might find that some classes share functionality with other classes. In that case, you can extend classes by creating subclasses to prevent duplication of functionality.

In “Constructors — Level One” on page 140, the DDATA class implicitly extended OBJECT.CLASS. In fact, any class without an explicit EXTENDS clause in the CLASS statement extends OBJECT.CLASS. To explicitly extend a class, add the EXTENDS clause shown below:

```
class y extends x;
endclass;
```

In this case, class Y extends the class X. Alternatively, Y is a subclass of X, and X is the parent class of Y.

This enables Y to share X’s functionality. For example, if the class X were

```
class x;
 m: method;
    put 'Hello';
 endmethod;
endclass;
```

and the class y were

```
class y extends x;
endclass;
```

then you could call the method M using an example of the class Y:

```
init:
 dcl y y = _new_ y();
 y.m();
 return;
```

Access Modifiers — Level Two

The access modifiers that we mentioned above – public, private and protected – can now be explained. A variable (or method) that is declared as public can be accessed anywhere. A variable (or method) that is declared as protected can be accessed only by non-proper subclasses of the class in which its declared. Protected variables can be accessed only from the class in which they are declared. This is also true for protected variables that are accessed from the subclasses of those classes.

These modifiers restrict access to certain variables (or methods) that should not be seen outside the class or class hierarchy in which they are declared. For example, there is no need for any class outside the DATA class hierarchy to access the *fid* variable, so it is declared as protected but could also be declared as private.

The DDATA Class as a Subclass — Level Two

To illustrate how subclassing works with the DDATA class, this exercise creates a similar class for external data files. The following SCL functions will be used:

- FOPEN
- FREAD
- FGET
- FCLOSE

These SCL functions will be used to create a class called FDATA to represent an external file. It is important to note similarities to the DDATA class. In particular, each class will have a name, input mode, and file identifier, so a class will be created to store this information. Then the subclasses DDATA and FDATA will be created from the DATA class. The parent data class will be

```
class data;
   public num type;
   public string dname;
   public string mode;
   protected num fid;

data: method f: num n: string m:string;
      fid = f;
      dname = n;
      mode = m;
endmethod;

endclass;
```

In addition to the *name*, *mode* and *file id*, a type variable is stored to indicate whether FDATA is an external file or a SAS data set.

The constructor DATA will be called whenever an example of the DATA class is created. It will also be called automatically whenever any subclasses of data are created if the constructor in the subclass has not be overridden. If the constructor has been overridden, you must use _super to call the parent constructor. You must also use _super if the argument list that is used in the _NEW_ operator does not match the argument list of the parent constructor. This will be the case for DDATA and FDATA.

To extend the DATA class, modify the DDATA CLASS statement, data declarations, and constructor as follows:

```
class ddata extends data;

    /* Class data */
    protected num nvars;

    /* Constructor method */
    ddata: method n: string m:string nv:num;
           fid = open(n, m);
           _super(fid, n, m);
           nvars = nv;
           type = 1;
    endmethod;
```

In this example, the DDATA constructor will call the data constructor via the _super call. This sets the name, mode and file identifier that are stored in the parent class data. The DDATA constructor still sets *nvars* and also sets the type field to indicate that the file is a data set. The rest of the class will remain the same.

The FDATA Class — Level Two

The declaration and constructor of the FDATA class will be similar to those of the DDATA class, as shown in the following:

```
class fdata extends data;

  /* Constructor method */
  fdata: method n: string m: string;
         dcl string ref = "";
         dcl num rc = filename(ref, n);
         fid =  fopen(ref, m);
         _super(fid, n, m);
         type = 2;
  endmethod;

  /* FREAD method */
  read: method return=num;
        dcl num rc = fread(fid);
        return rc;
  endmethod;

  /* FGET method */
  cpy: method n: num return=string;
       dcl string c = "";
       dcl num rc = fget(fid, c);
       return c;
  endmethod;

  /* FCLOSE method */
  _term: method /(state='O');
```

```
          if (fid) then fclose(fid);
          _super();
   endmethod;
 endclass;
```

Use FDATA to read an external class by instantiating it and looping through the data:

```
init:
dcl fdata f = _new_ fdata("some_file", "i");
dcl num more = ^f.read();
do while(more);
   dcl string s s2;
   s = f.cpy(1);
   s2 = f.cpy(2);
   put s s2;
   more = ^f.read();
end;
f._term();
return;
```

This code assumes that the external file is formatted with each line containing two character variables separated by a blank. For example:

```
Geoffrey Chaucer
Samuel Johnson
Henry Thoreau
George Eliot
Leo Tolstoy
```

Overloaded Methods — Level Two

Method overloading is the process of defining multiple methods that have the same name, but which differ in parameter number, type, or both. Method overloading lets you use the same name for methods that are related conceptually but take different types or numbers of parameters.

For example, you may have noticed that the CPY method in FDATA has a numeric parameter that apparently serves no useful purpose. You do not need to specify a variable number for an external file. This parameter is used so that in the future when you use interfaces, the CPY method in FDATA matches the one in DDATA. For now, the parameter is not needed. One way of resolving this is to overload the CPY method by creating another CPY method with a different parameter list, as shown in the following code:

```
cpy: method return=string;
     dcl string c="";
     dcl num rc = fget(fid, c);
     return c;
endmethod;

cpy: method n: num return=string;
     return cpy();
endmethod;
```

In this example, the original CPY method ignores the parameter and calls a CPY method that returns the character value. By doing this, you have defined two methods that have the same name but different parameter types. With this simple change, you do not have to worry about which method to call.

The CPY method can be used as follows:

```
 s = f.cpy();
```

Overloaded methods can be used any time you need to have multiple methods with the same name but different parameter lists. For example, you may have several methods that are conceptually related but which operate on different types of data, or you may want to create a method with an optional parameter, as in the CPY example.

To differentiate between overloaded methods, the compiler refers to the *method signature*, which is a list of the method's parameter types. A method signature provides a means of extending a method name, so that the same name can be combined with multiple different signatures to produce multiple different actions. Method signatures are created automatically when a method is added to a class and when the compiler is parsing a method call. Method signatures appear as part of the information that the Class Editor displays about a method.

Interfaces and Higher Levels of Abstraction — Level Three

The routines that use DDATA and FDATA are very similar. In fact, the set of methods for each class is similar by design. The actual implementations of the methods differ. For example, the CPY method in DDATA is different from the CPY method in FDATA, but the basic concept of reading character data is the same. In effect, the interface for both classes is essentially the same.

You can exploit this similarity to make it easier to use the two classes. In fact, you can have one data loop that handles both types of classes by defining an SCL interface for both classes. To define the interface, you generalize the functionality and create the following SCL entry:

```
interface reader;
  read: method return=num;
  cpy: method n: num return=string;
 endinterface;
```

Use SAVECLASS to create an interface entry with two abstract methods, READ and CPY. The abstract methods are by definition the interface itself. Any class that supports this interface will need to supply the implementations for these methods.

When you use the SAVECLASS command in an SCL entry that contains a CLASS block, the class is generated and its CLASS entry is created. This is the equivalent of using the Class Editor to interactively create a CLASS entry.

Once you have created the interface, you must modify the DDATA and FDATA classes to support it. To do that, change the CLASS statements in each class as follows:

```
 class ddata extends data supports reader;
```

and

```
 class fdata extends data supports reader;
```

Since DDATA and FDATA contain READ and CPY methods, no other changes are needed in the classes.

To use the new interface, you will create two helper classes. One is an iterator class that will be used to abstract the looping process over both DDATA and FDATA. Use the following code to create the two helper classes:

```
class iter;

  private num varn nvars;
  public reader r /(autocreate='n');

  /* Constructor */
  iter: method rdr: reader n: num;
        varn = 1;
        nvars = n;
        r = rdr;
  endmethod;

  /* Check if there are more elements to iterate over */
  more: method return=num;
        dcl num more = ^r.read();
        varn = 1;
        return more;
  endmethod;

  /* Return the next element */
  next: method return=string;
        dcl string c = "";
        c =  r.cpy(varn);
        varn + 1;
        return c;
  endmethod;

endclass;
```

Several things require explanation for this class. First, note that it has two private variables, *varn* and *nvars*, to keep track of where it is in the iteration process.

It also has a variable *r* which is an interface type. Since we cannot create the interface automatically when an example of ITER is created, we specify the AUTOCREATE='N' option.

The iterator has three methods. In the first method, the constructor stores the reader variable and the number of variables in the reader. A *reader variable* is any class that supports the READER interface. The MORE method reads from the reader to check whether there are any more elements. The NEXT method returns the next element.

The other helper class uses the iterator to loop over the data in a reader.

```
class read;

 private iter i;

 read: method r: reader;
       i = _new_ iter(r, 2);
 endmethod;

 loop:method;
     do while(i.more());
     dcl string s s2;
     s = i.next();
     s2 = i.next();
     put s s2;
   end;
   endmethod;

   _term: method /(state='O');
     i._term();
     _super();
   endmethod;

 endclass;
```

The constructor will create a new iterator, and the LOOP method will use it to loop over the data.

The SCL to use these classes is

```
 init:
 dcl string filename;
 dcl fdata f;
 dcl ddata d;
 dcl read r;

 /* Read an external file */
 filename = "some_file";
 f = _new_ fdata(filename, "i");
 r = _new_ read(f);
 r.loop();
 r._term();

 /* Read a dataset */
 filename = "sasuser.x";
 d = _new_ ddata(filename, "i", 2);
 r = _new_ read(d);
 r.loop();
 r._term();
 return;
```

This code will successively read an external file and a data set.

Other Classes and Further Abstraction — Level Four

Given the reader interface, you can now use other classes – even ones outside the data class hierarchy – as readers, as long as they support the reader interface. Using the abstract reader interface enables you to read from many different types of objects as well.

For example, consider the following class, which uses SCL lists to maintain data:

```
class lst supports reader;
  private list l;
  private num nvars;
  private num nelmts;
  private num cur;

  /* Constructor */
  lst: method n:num;
       l = makelist();
       nvars = n;
       nelmts = 0;
```

```
          cur = 1;
     endmethod;

     /* Copy method */
     cpy: method n: num return=string;
            dcl string c = "";
            if (cur <= nelmts) then do;
              c = getitemc(l, cur);
              cur + 1;
            end;
            return c;
     endmethod;

    /* Read method */
     read: method return=num;
        if (cur > nelmts) then
            return 1;
        else
            return 0;
     endmethod;

     /* Add an element to the list */
     add: method c:string;
          nelmts + 1;
          setitemc(l, c, nelmts, 'Y');
     endmethod;

     /* Add two elements to the list */
     add: method c1:string c2:string;
            add(c1);
            add(c2);
     endmethod;

     /* Terminate the list */
     _term: method /(state='O');
         if (l) then dellist(l);
         _super();
     endmethod;
   endclass;
```

This class represents a list, and because it supports the READER interface, it can be read in the same way as the DDATA and FDATA classes.

The SCL for reading from the list is

```
init:
 dcl lst l;
 dcl read r;
 l = _new_ lst(2);

 /* Notice the overloaded add method */
 l.add("123", "456");
 l.add("789", "012");
 l.add("345", "678");

 /* Create a read class and loop over the data */
```

```
   r = _new_ read(1);
   r.loop();

   r._term();
   return;
```

The output for this program will be

```
123 456
789 012
345 678
```

USECLASS — Level One

This section presents the USECLASS statement and is intended for those users who are unfamiliar with USECLASS. It is not required for the remainder of the tutorial.

A USECLASS statement binds methods that are implemented within it to the specified class definition. USECLASS allows a class's method implementations to reside in different entries other than the class declaration's entry. This is helpful if a class is complex enough to require several developers to write its methods.

The DDATA class will be modified to use USECLASS. Although this class is certainly not complex enough to require USECLASS, it illustrates its use.

First, rewrite the class specification, using the following code:

```
class ddata;

  /* Data */
  public string dname;
  public string mode;
  protected num fid;
  protected num nvars;

  /* Constructor method */
  ddata: method n: string m:string nv:num /
 (scl='sasuser.a.constr.scl');

  /* FETCH method */
  read: method return=num                  /
 (scl='sasuser.a.read.scl');

  /* GETVARC method */
  cpy: method n: num return=string         /
 (scl='sasuser.a.cpy.scl');

 /* CLOSE method */
  _term: method                            /
 (state='O', scl='sasuser.a.trm.scl');
endclass;
```

The method implementations are removed, and the method declaration statements are modified to indicate which SCL entry contains each method implementation. This new class specification should be compiled with the SAVECLASS command.

Next, create the method implementations in each entry. These should be compiled with the COMPILE command, not with SAVECLASS. SASUSER.A.CONSTR.SCL should contain

```
useclass ddata;

  /* Constructor method */
  ddata: method n: string m:string nv:num;
         fid = open(n, m);
         dname = n;
         mode = m;
         nvars = nv;
  endmethod;

enduseclass;
```

SASUSER.A.READ.SCL should contain

```
useclass ddata;

  /* FETCH method */
  read: method return=num;
       dcl num rc;
       rc = fetch(fid);
       return rc;
  endmethod;

enduseclass;
```

SASUSER.A.CPY.SCL should contain

```
useclass ddata;

  /* GETVARC method */
  cpy: method n: num return=string;
      dcl string c = "";
      if (vartype(fid, n) = 'C') then
         c = getvarc(fid, n);
      return c;
  endmethod;

enduseclass;
```

SASUSER.A.TRM.SCL should contain

```
useclass ddata;

 /* CLOSE method */
  _term: method /(state='O');
        if (fid) then close(fid);
        _super();
  endmethod;
enduseclass;
```

Using SCL Class Syntax with SAS/AF — Level Five

So far you have created stand-alone SCL classes. SCL class syntax can be used to create SAS/AF visual objects.

This exercise will extend the SAS/AF List Box class. The readers that were previously developed in this tutorial will be used to read data that will be used to initialize the items in the List Box.

To extend the List Box class, you must write a class to extend List Box and modify the READ class that was created in "Other Classes and Further Abstraction — Level Four" on page 148. The READ class will then be able to pass the new class to the list box to use for initializing the item list instead of having it simply print the character data after reading it.

You must create a new interface and make a minor modification to the LOOP method in READ.

The interface has a single method, CALLBACK:

```
interface call;
 callback: method s:string;
endinterface;
```

The modified LOOP method in READ is

```
loop: method caller:call;
       do while(i.more());
         caller.callback(i.next());
       end;
   endmethod;
```

The method now takes a CALL interface parameter and calls its CALLBACK method.

Note: You do not specify the implementation of a method in an interface; you simply supply the name and parameter list. △

It is not necessary to know what the implementation for CALLBACK is at this point, only that you call it in the read loop and pass a character value to it. Whatever class supports the interface will supply the implementation for CALLBACK.

Now, create the extended List Box class (depending on which version of SAS you have, you may need to create an empty MLIST class first in order for the following to work).

```
import sashelp.classes;
class mlist extends listbox_c supports call;

   /* Local item list */
   private list l;

   /* Set method */
   set: method r: read;
        l = makelist();
        r.loop(_self_);
   endmethod;

   /* Store the character value in the local list */
   callback: method s:string;
             insertc(l, s, -1);
   endmethod;

   /* Set the items attribute */
   setattr: method;
            _self_.items = l;
   endmethod;

endclass;
```

Note how the IMPORT statement and the LOOP, SET, SETATTR, and CALLBACK methods will be used:

- The IMPORT statement defines a search path for CLASS entry references in an SCL program so that you can refer to a class by its two-level name instead of having to specify the four-level name each time. It is used to specify a catalog to search for abbreviated class names. For example, the MLIST class extends LISTBOX_C, but if LISTBOX_C is not in the current catalog, the compiler will not know where to find it. The IMPORT statement tells the compiler to search the SASHELP.CLASSES catalog for any classes it cannot find in the current catalog.
- The SET method is used to set up a local list that will hold the new set of items for the list box. It will also call the LOOP method in READ, with MLIST's object as a parameter. Recall that MLIST supports the CALL interface, so this will work with the new LOOP method that was created above.
- As the LOOP method executes, it will call the CALLBACK method for each character variable that it reads. The CALLBACK method will store the variable in the local list that was created in the SET method.
- Finally, the SETATTR method will assign the local list to MLIST's item list, thus changing the list of items seen when the List Box, which is actually MLIST, is displayed.

To see how this works, create the CALL interface, as well as the classes READ and MLIST, by using the SAVECLASS command. Then edit a frame. In the Components window, add the MLIST class to the class list (via **AddClasses** on the pop-up menu). After it appears on the list, drag and drop MLIST to the frame. In the frame's source, enter

```
init:
 dcl ddata d;
 dcl read r;
 dcl string filename = "sasuser.x";
 d = _new_ ddata(filename, "i", 2);
 r = _new_ read(d);
 mlist1.set(r);
 mlist1.setattr();
 return;
```

This will create a DDATA reader with an associated READ class. Now call the SET and SETATTR methods on the new List Box class (MLIST).

Compile and use the TESTAF command on the frame. The initial list of items will be

```
San Francisco
Honolulu
New York
Miami
```

Flexibility — Level Five

Using the CALL interface in the above exercise allows a great deal of flexibility in modifying the MLIST and READ classes.

For example, to process numeric data instead of character data, you could simply overload the CALLBACK method in the interface

```
interface call;
 callback: method s:string;
```

```
   callback: method n:num;
  endinterface;
```

and support it in the MLIST class

```
   callback: method n:num;
              /* process numeric value */
   endmethod;
```

Now, the READ class – or any class that supports CALL – can call the CALLBACK method with a numeric parameter. Clearly, this process can be generalized to make use of any possible parameter lists that are needed for CALLBACK.

Another feature is that any class that supports the READER interface can be used to read the data into the list box. For example, to use an external file, change the frame's SCL to

```
  init:
   dcl fdata f;
   dcl read r;
   dcl string filename = "some_file";
   f = _new_ fdata(filename, "i");
   r = _new_ read(f);
   mlist1.set(r);
   mlist1.setattr();
   return;
```

We can consolidate the code further by creating another class to set up the reader:

```
  init:
   dcl SetReader g = _new_ SetReader();
   dcl read r = g.get();
   mlist1.set(r);
   mlist1.setattr();
   return;
```

SetReader sets up whatever reader is necessary even if your program is using external data. Then, at the frame level, you can read from any type of data source, such as a data set, an external file, an SCL list, or any other user-defined data source. The only requirement is that SetReader support the reader interface.

The correct bibliographic citation for this manual is as follows: SAS Institute Inc., *SAS® Component Language: Reference, Version 8*, Cary, NC: SAS Institute Inc., 1999.

SAS® Component Language: Reference, Version 8

Copyright © 1999 by SAS Institute Inc., Cary, NC, USA.

ISBN 1–58025–495–0

All rights reserved. Produced in the United States of America. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, or otherwise, without the prior written permission of the publisher, SAS Institute Inc.

U.S. Government Restricted Rights Notice. Use, duplication, or disclosure of the software and related documentation by the U.S. government is subject to the Agreement with SAS Institute and the restrictions set forth in FAR 52.227–19 Commercial Computer Software-Restricted Rights (June 1987).

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513.

1st printing, October 1999

SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are registered trademarks or trademarks of their respective companies.

The Institute is a private company devoted to the support and further development of its software and related services.