CHAPTER 2

The Structure of SCL Programs

Introduction

An SCL application consists of one or more SCL entries. These SCL entries can contain the following types of modules:

- labeled sections
- CLASS blocks
- METHOD blocks
- USECLASS blocks
- INTERFACE blocks
- macros.

For example, an SCL program may consist of only one labeled section, or it may contain two labeled sections, a METHOD block, and a couple of macros. A complex program may contain several CLASS, METHOD, USECLASS, and INTERFACE blocks.

Some types of modules can be stored together in one SCL entry, but others cannot. One SCL entry may contain any one of the following:

- one or more labeled sections and/or one or more macros
- one CLASS block, which may also contain one or more METHOD blocks
- one INTERFACE block
- one or more USECLASS blocks, each of which may contain one or more METHOD blocks
- one or more METHOD blocks (that are not contained within a CLASS or USECLASS block).

In strictly object-oriented applications, METHOD blocks are contained within CLASS or USECLASS blocks. If your application is not a strictly object-oriented application, you can save METHOD blocks by themselves in SCL entries.

Object-oriented applications use CLASS, METHOD, USECLASS, and INTERFACE blocks extensively. For information about designing and implementing object-oriented applications see *SAS Guide to Applications Development* in addition to the information contained in this documentation.

Using Labeled Sections

SCL programs execute in phases, such as the initialization phase and the termination phase. During each phase, control of the entry can pass to a different segment of an SCL program. The segments of the program are identified by labels; that is, the SCL program is divided into labeled sections.

Labeled sections are program segments that are designed to be called only within the program in which they are defined. Labeled sections begin with a label and end with a RETURN statement. A label may be one of the reserved labels such as INIT or MAIN; it may correspond to a field name, window-variable name, or object name in your application; or it may simply identify a module that is called repetitively, such as a module that sorts data.

Labeled sections are a good solution when you are working within one SCL program because they can help divide the program into smaller, simpler pieces.

For example, the following SCL program contains an INIT section that is executed before the application window is displayed; two sections that correspond to window variables, NAME and ADDRESS; and a MAIN section that is executed each time the user updates a field in the window.

```
init:
   ...some SCL statements...
   return;

name:
   ...some SCL statements...
   return;

address:
   ...some SCL statements...
   return;

main:
   ...some SCL statements...
   return;
```

Reserved Labels

There are five reserved labels:

FSEINIT
: An FSEINIT section, which is valid in FSEDIT and FSBROWSE applications only, contains any statements that are needed to initialize the application. These statements are executed one time only when the user invokes the FSEDIT or FSBROWSE procedure and before the first row is displayed.

INIT
: The INIT section is executed before the application window is displayed to the user. Typically, you use the INIT section to initialize variables, to import values through macro variables, to open tables, and to define initial messages to display when the window is opened. In FSEDIT and FSBROWSE applications, as well as Data Table and Data Form controls, the INIT section is executed before each SAS table row is displayed.

MAIN
: The MAIN section is executed each time the user modifies a field in the window and presses ENTER.

TERM
: The TERM section is executed when the user issues the END command. You might use the TERM section to close tables, to export values of macro variables, and to construct and submit statements for execution. In FSEDIT applications, Data Table controls, and Data Form controls, the TERM section is executed after each SAS table row is displayed, provided that the MAIN section has been executed for the row.

FSETERM
: The FSETERM section is valid in FSEDIT applications only. This section executes once after the user issues the END command and terminates the FSEDIT procedure.

In FSVIEW applications, you write individual formulas consisting of one or more lines of SCL code for each computed variable, rather than complete SCL programs. These formulas are stored in FORMULA entries. The FSVIEW procedure automatically adds a label before the formula and a RETURN statement after the formula. The label is the same as the name of the variable whose value you are calculating.

An SCL program for FSEDIT or FSBROWSE applications must contain at least one of the following reserved labels: INIT, MAIN, or TERM. If a program does not include at least one of these three reserved labels, the procedure never passes control to your SCL program. If a program does not contain all three of these reserved labels, you get one or more warning messages when you compile the program.

The FSEINIT and FSETERM labels are optional. The compiler does not issue any warnings if these labels are not present.

Neither SCL programs for FRAME entries nor programs in SCL entries that contain method block definitions require any reserved sections.

For more information about the FSVIEW, FSEDIT, and FSBROWSE procedures, see*SAS/FSP Software Procedures Guide*. For more information about FRAME applications, see *SAS Guide to Applications Development*.

Window Variable Sections

SCL provides a special type of labeled section, called a window variable section, that automatically executes when a user takes an action in a particular control or field. For example, window variable sections might be used to verify values that users enter in controls or fields. An SCL program can include labeled sections for any number of window variables.

The sequence for executing window variable sections is determined by the physical position of the window element. Window variable sections execute sequentially for each window element, from left to right and from top to bottom. A window variable section must be labeled with the name of the associated window variable. For more information about window variables, see *SAS Guide to Applications Development*.

Correcting Errors in Window Variables

To correct an error in a window variable, you can allow users to correct the error in the window, or you can include a CONTROL ERROR statement along with statements in the window variable section that make a correction, as shown in the following example:

```
INIT:
    control error;
return;

Name:
   if error(Name) then do;
      erroroff Name;
      Name=default-value-assigned-elsewhere;
      _msg_=
          'Value was invalid and has been reset.';
   end;
return;
```

Using a window variable section in this manner reduces overhead because the program's MAIN section executes only after the window variable sections for all modified window variables have executed correctly.

If a program also uses CONTROL ERROR, CONTROL ALWAYS, or CONTROL ALLCMDS, then the MAIN section executes after the window variable sections even if an error has been introduced. For more information about the CONTROL statement, see "CONTROL" on page 302.

Defining Classes

Classes define data and the operations that you can perform on that data. You define classes with the CLASS statement. For example, the following code defines a class, Simple, that defines an attribute named **total** and implements a method named addNums:

Example Code 2.1 Example Class Definition

```
class Simple;
  public num total;

  addNums: public method n1:num n2:num return=num;
      total=n1+n2;
      return(total);
  endmethod;

endclass;
```

The CLASS statement does not have to implement the methods. The methods may be only declared. For example:

```
class Simple;
  public num total;

  addNums: public method n1:num n2:num return=num
       / (scl=work.a.simMeth.scl');
endclass;
```

The code to implement the method is contained in **`work.a.simMeth.scl`** (see Example Code 2.2 on page 14).

For more information about defining and using classes, see Chapter 8, “SAS Object-Oriented Programming Concepts,” on page 93; Chapter 9, “Example: Creating An Object-Oriented Application,” on page 137; and “CLASS” on page 277.

Defining and Using Methods

Methods define operations that you can perform on data. Methods are defined with the METHOD statement. Methods can be implemented in CLASS blocks or in USECLASS blocks. In addition, if you are not designing a strictly object-oriented application, they can be stored in separate SCL entries.

Storing method implementations in SCL entries enables you to write methods that perform operations that are common to or shared by other applications. You can call the methods from any SAS/AF application.

For more information about defining and using methods in CLASS and USECLASS blocks, see Chapter 8, “SAS Object-Oriented Programming Concepts,” on page 93 and Chapter 9, “Example: Creating An Object-Oriented Application,” on page 137.

Defining Method Blocks

To define a method, use the METHOD statement. For example, the METHOD statement in Example Code 2.1 on page 12 defines the public method addNums, which takes two numeric parameters, adds them, and returns the total.

Note: Do not include window-specific statements or functions (for example, the PROTECT and CURSOR statements and the FIELD and MODIFIED functions) in method blocks that are stored in independent SCL entries. SCL entries that contain window-specific statements or functions will not compile independently. They must be compiled in conjunction with the associated FRAME entry. △

When you want to pass parameters between an SCL program and a method block, you use the same principles as when you are passing parameters between a CALL DISPLAY statement and an ENTRY statement. Unless the REST=, ARGLIST=, or OPTIONAL= option is used in the METHOD statement, the parameter list for the METHOD statement and the argument list for the associated ENTRY statement must agree in the following respects:

- The number of parameters passed by the METHOD statement must equal the number of arguments received by the ENTRY statement.
- The position of each parameter in the METHOD statement must be the same as the position of the corresponding argument in the ENTRY statement.
- Each parameter in the METHOD statement and its corresponding argument in the ENTRY statement must have the same data type.

The parameters and arguments do not have to agree in name. For more information, see “METHOD” on page 540 and “ENTRY” on page 369.

Calling a Method That Is Stored in an SCL Entry

If the method module is stored in a PROGRAM entry or SCREEN entry, then you must use a LINK or GOTO statement to call it. Although parameters cannot be passed with LINK or GOTO statements, you can reference global values in those statements.

If the module is an SCL entry, then call the method module with a CALL METHOD routine. The CALL METHOD routine enables you to pass parameters to the method. For example, a program that needs to validate an amount can call the AMOUNT method, which is stored in METHDLIB.METHODS.VALIDATE.SCL:

```
call method('methdlib.methods.validate.scl',
             'amount',amount,controlid);
```

After the method module performs its operations, it can return modified values to the calling routine.

If the method module is stored in the SCL entry that is associated with the FRAME entry, then you must compile the SCL entry as a separate entity from the FRAME entry in addition to compiling the FRAME entry.

For more information about the CALL METHOD routine, see “METHOD” on page 539.

USECLASS Blocks

USECLASS blocks contain method blocks. A USECLASS block binds the methods that are implemented within it to a class definition. This binding enables you to use the attributes and methods of the class within the methods that are implemented in your USECLASS block. However, your USECLASS block does not have to implement all of the methods defined in the class.

For example, the USECLASS block for the class defined in “Defining Classes” on page 12 would be stored in **work.a.simMeth.scl** and would contain the following code:

Example Code 2.2 USECLASS Block for the addNums Method

```
useclass simple.class;
  addNums: public method n1:num n2:num return=num;
       total=n1+n2;
       return(total);
  endmethod;
enduseclass;
```

Using USECLASS blocks to separate the class definition from the method implementations enables multiple programmers to work on method implementations simultaneously.

For more information, see Chapter 8, “SAS Object-Oriented Programming Concepts,” on page 93; “USECLASS — Level One” on page 150; and “USECLASS” on page 698.

Defining Interfaces

Interfaces are groups of method declarations that enable classes to possess a common set of methods even if the classes are not related hierarchically. Interfaces are especially useful when you have several unrelated classes that perform a similar set of actions. These actions can be declared as methods in an interface, and each associated class can provide its own implementation for each of the methods. In this way, interfaces provide a form of multiple inheritance.

For more information about defining and using interfaces, see Chapter 8, “SAS Object-Oriented Programming Concepts,” on page 93; Chapter 9, “Example: Creating An Object-Oriented Application,” on page 137; and “INTERFACE” on page 486.

Using Macros

You can use the SAS macro facility to define macros and macro variables for your SCL program. That is, you can use SAS macros and SAS macro variables that have been defined elsewhere in the SAS session or in autocall libraries. You can then pass parameters between macros and the rest of your program. In addition, macros can be used by more than one program. However, macros can be more complicated to maintain than the original program segment because of the symbols and quoting that are required.

If a macro is used by more than one program, you must keep track of all the programs that use the macro so that you can recompile all of them each time the macro is updated. Because SCL is compiled (rather than interpreted like the SAS language), each SCL program that calls a macro must be recompiled whenever that macro is updated in order to update the program with the new macro code.

Macros and macro variables in SCL programs are resolved when you compile the SCL program, not when a user executes the application. However, you can use the SYMGET and SYMGETN functions to retrieve the value of a macro variable or to store a value in a macro variable at execution time, and you can use the SYMPUT and SYMPUTN functions to create a macro variable at execution time. For more information, see “Using Macro Variables” on page 87.

Note: Macros and macro variables within submit blocks are not resolved when you compile the SCL program. Instead, they are passed with the rest of the submit block to SAS software when the block is submitted. For more information about submit blocks, see “Submitting SAS Statements and SQL Statements” on page 80. △

Note: Using macros does not reduce the size of the compiled SCL code. Program statements that are generated by a macro are added to the compiled code as if those lines existed at that location in the program. △

Example

The following SCL program uses macros to validate an amount and a rate:

```
%macro valamnt;
  if amount < 0 or amount > 500 then do;
    erroron amount;
    _msg_='Amount must be between $0 and $500.';
    stop;
  end;
  else erroroff amount;
%mend;
%macro rateamnt;
  if rate<0 or rate>1 then do;
    erroron rate;
    _msg_='Rate must be between 0 and 1.';
    stop;
  end;
  else erroroff rate;
%mend;

INIT:
  control error;
  amount=0;
  rate=.5;
return;

MAIN:
  payment=amount*rate;
return;

TERM:
return;

AMOUNT:
  %valamnt
return;

RATE:
  %rateamnt
return;
```

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

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

1st printing, October 1999