CHAPTER

6

Controlling Program Flow

Introduction

You can control the flow of execution of your SCL application by

- using any of several programming constructs such as DO loops and IF/THEN-ELSE statements
- branching to labeled sections with the LINK statement
- branching to PROGRAM, FRAME, MENU, CBT, or HELP entries with the GOTO statement
- branching to another SCL entry with CALL DISPLAY
- executing a method that is stored in a separate SCL entry with CALL METHOD
- executing an object method by using dot notation
- sending a method to an object with CALL SEND
- sending a method to a FRAME entry control with CALL NOTIFY
- specifying how labeled sections are executed, when and where submit blocks are executed, and whether execution halts when errors are encountered in dot notation with the CONTROL statement
- creating a program halt handler to control how run-time errors are processed.

For more information about controlling the flow of execution in applications that use frames, refer to *SAS Guide to Applications Development*.

Using DO Loops

There are four forms of the DO statement:

- The DO statement designates a group of statements that are to be executed as a unit, usually as a part of IF-THEN/ELSE statements.
- The iterative DO statement executes a group of statements repetitively based on the value of an index variable. If you specify an UNTIL clause or a WHILE clause, then the execution of the statements is also based on the condition that you specify in the clause.
- The DO UNTIL statement executes a group of statements repetitively until the condition that you specify is true. The condition is checked *after* each iteration of the loop.
- The DO WHILE statement executes a group of statements repetitively as long as the condition that you specify remains true. The condition is checked *before* each iteration of the loop.

For more information about DO statements, in addition to the information in this documentation, refer to *SAS Language Reference: Dictionary*.

DO Statement

The DO statement designates a group of statements that are to be executed as a unit. The simplest form of the DO loop is

DO;

. . .*SAS statements*. . .

END;

This simple DO statement is often used within IF-THEN/ELSE statements to designate a group of statements to be executed if the IF condition is true. For example, in the following code, the statements between DO and END are performed only when YEARS is greater than 5.

```
if years>5 then
  do;
    months=years*12;
    put years= months=;
  end;
```

Iterative DO Loops

The iterative DO loop executes the statements between DO and END repetitively based on the value of an index variable.

DO *index-variable* = *start* TO *stop* <BY *increment*>;

Note: In SCL applications, both *start* and *stop* are required, and *start*, *stop*, and *increment* must be numbers or expressions that yield a number. The TO and BY clauses cannot be reversed, and *start* cannot be a series of items separated by commas. You can use only one *start* TO *stop* specification (with or without the BY clause) in a DO loop. △

If *increment* is not specified, then *index-variable* is increased by 1. If *increment* is positive, then *start* must be the lower bound and *stop* must the be upper bound for the

loop. If *increment* is negative, then *start* must be the upper bound and *stop* must be the lower bound for the loop.

The values of *start*, *stop*, and *increment* are evaluated before the first execution of the loop. Any changes made to *stop* or *increment* within the DO group do not affect the number of times that the loop executes.

CAUTION:
Changing the value of *index-variable* within the DO group may produce an infinite loop. If you change the value of *index-variable* inside of the DO group, then *index-variable* may never become equal to the value of *stop*, and the loop will not stop executing. △

For example, the following code prints the numbers 20, 18, 16, 14, 12, and 10.

```
dcl num k=18 n=11;
do i=k+2 to n-1 by -2;
   put i;
end;
```

The following code uses the DOPEN and DNUM functions to execute SAS statements once for each file in the current directory:

```
rc=filename('mydir','.');
dirid=dopen('mydir');
do i=1 to dnum(dirid);
   ...SAS statements...
end;
rc=dclose(dirid);
```

Using UNTIL and WHILE Clauses

You can add either an UNTIL clause or a WHILE clause to your DO statements.

DO *index-variable* = *start* TO *stop* <BY *increment*>

<WHILE (*expression*)> | <UNTIL (*expression*)>;

The UNTIL expression is evaluated *after* the statements in the DO loop have executed, and the WHILE expression is evaluated *before* the statements in the DO loop have executed. The statements in a DO UNTIL loop are always executed at least once, but the statements in a DO WHILE loop will not execute even once if the DO WHILE expression is false.

If *index-variable* is still in the range between *start* and *stop*, then if you specify an UNTIL clause, the DO group will execute until the UNTIL expression is true. If you specify a WHILE clause, the loop will execute as long as the WHILE expression is true.

The following example uses an UNTIL clause to set a flag, and then it checks the flag during each iteration of the loop:

```
flag=0;
do i=1 to 10 until(flag);
  ...SAS statements...
  if expression then flag=1;
end;
```

The following loop executes as long as I is within the range of 10 to 0 and MONTH is equal to **JAN**.

```
do i=10 to 0 by -1 while(month='JAN');
   ...SAS statements...
end;
```

DO WHILE Statement

The DO WHILE statement works like the iterative DO statement with a WHILE clause, except that you do not specify an *index-variable* or *start*, *stop*, or *increment*.

DO WHILE (*expression*);

. . .*SAS statements*. . .

END;

Whether the loop executes is based solely on whether the expression that you specify evaluates to true or false. The expression is evaluated before the loop executes, and if the expression is false, then the loop is not executed. If the expression is false the first time it is evaluated, then the loop will not execute at all.

For example, the following DO loop is executed once for each value of N: 0, 1, 2, 3, and 4.

```
n=0;
do while(n<5);
  put n=;
  n+1;
end;
```

DO UNTIL Statement

The DO UNTIL statement works like the iterative DO statement with an UNTIL clause, except that you do not specify an index variable nor *start*, *stop*, or *increment*.

DO UNTIL (*expression*);

. . .*SAS statements*. . .

END;

Whether the loop executes is based solely on whether the expression that you specify evaluates to true or false. The loop is always executed at least once, and the expression is evaluated after the loop executes.

For example, the following DO loop is executed once for each value of N: 0, 1, 2, 3, and 4.

```
n=0;
do until(n>=5);
  put n=;
  n+1;
end;
```

Controlling DO Loops (CONTINUE and LEAVE)

You can use the CONTINUE and LEAVE statements to control the flow of execution through DO loops.

The CONTINUE statement stops the processing of the current DO loop iteration and resumes with the next iteration of the loop. For example, the following code reads each row in the DEPT table, and if the status is not **PT**, it displays a frame that enables the user to update the full-time employee's salary.

```
deptid=open('dept');
call set(deptid);
```

```
do while (fetch(deptid) ne -1);
  if (status='PT') then continue;
  newsal=display('fulltime.frame');
end;
```

The LEAVE statement stops processing the current DO loop and resumes with the next statement after the DO loop. With the LEAVE statement, you have the option of specifying a label for the DO statement:

LEAVE <*label*>;

If you have nested DO loops and you want to skip out of more than one loop, you can specify the label of the loop that you want to leave. For example, the following LEAVE statement causes execution to skip to the last PUT statement:

```
myloop:
do i=1 to 10;
  do j=1 to 10;
    if j=5 then leave myloop;
    put i= j=;
  end;
end;
put 'this statement executes next';
return;
```

In SCL applications, the LEAVE statement can be used only within DO loops, not in SELECT statements (unless it is enclosed in a DO statement).

For more information, refer to "CONTINUE" on page 300, "LEAVE" on page 504, and *SAS Language Reference: Dictionary*.

Using SELECT-WHEN/OTHERWISE Conditions

The SELECT statement executes one of several statements or groups of statements based on the value of the expression that you specify.

SELECT<(*select-expression*)>;

WHEN-1 (*when-expression-1*) *statement(s)*;

<WHEN-*n* (*when-expression-n*) *statement(s)*;>

<OTHERWISE *statement*;>

END;

SAS evaluates *select-expression*, if present, as well as *when–expression-1*. If the values of both expressions are equal, then SAS executes the statements associated with *when-expression-1*. If the values are *not* equal, then SAS evaluates *when-expression-n*, and if the values of *select–expression-1* and *when-expression-1 are* equal, SAS executes the statements associated with *when-expression-n*. SAS evaluates each when expression until it finds a match or until it has evaluated all of the when expressions without finding a match. If you do not specify a select expression, then SAS evaluates each when expression and executes only the statements associated with the first when expression that evaluates to true.

If the value of none of the when expressions matches the value of the select expression, or if you do not specify a select expression and all of the when expressions are false, then SAS executes the statements associated with the OTHERWISE statement. If you do not specify an OTHERWISE statement, the program halts.

In SCL applications, you cannot specify a series of when expressions separated by commas in the same WHEN statement. However, separating multiple WHEN statements with a comma is equivalent to separating them with the logical operator OR, which is acceptable in SCL applications.

The statements associated with a when expression can be any executable SAS statement, including SELECT and null statements. A null statement in a WHEN statement causes SAS to recognize a condition as true and to take no additional action. A null statement in an OTHERWISE statement prevents SAS from issuing an error message when all of the when expressions are false.

Each WHEN statement implies a DO group of all statements until the next WHEN or OTHERWISE statement. Therefore the following program is valid:

```
select (paycat);
   when ('monthly')
     amt=salary;
   when ('hourly')
     amt=hrlywage*min(hrs,40);
     if hrs>40 then put 'Check timecard.';
   otherwise put 'problem observation';
end;
```

However, if you need to include a LEAVE statement as part of your WHEN statement, then you must explicitly specify the DO statement in your WHEN statement.

You can specify expressions and their possible values in either of the following ways:

1

SELECT;

WHEN (*variable operator value*) *statement(s)*;

END;

2

SELECT (*variable*);

WHEN (*value*) *statement(s)*;

END;

For example, both of the following SELECT statements are correct:

```
select;
  when (x<=5) put '1 to 5';
  when (x>=6) put '6 to 10';
end;

select (x);
  when (1) put 'one';
  when (2) put 'two';
end;
```

The following code is incorrect because it compares the value of the expression X with the value of the expression X=1. As described in "Boolean Numeric Expressions" on page 31, in Boolean expressions, a value of 0 is false and a value of 1 is true. Therefore, the expression X is false and the expression X=1 is false, so the program prints **x is 1**.

```
x=0;
select (x);
  when (x=0) put 'x is 0';
```

```
  when (x=1) put 'x is 1';
  otherwise put x=;
end;
```

For more information about the SELECT statement, refer to "SELECT" on page 640 and to *SAS Language Reference: Dictionary*.

Using IF-THEN/ELSE Conditions

The IF-THEN/ELSE statement executes a statement or group of statements based on a condition that you specify.

IF *expression* THEN *statement*;

<ELSE *statement*;>

If *expression* is true, then SAS executes the statement in the THEN clause. If the *expression* is false and if an ELSE statement is present, then SAS executes the ELSE statement. The statement following THEN and ELSE can be either a single SAS statement (including an IF-THEN/ELSE statement) or a DO group.

For example:

```
if (exist(table)) then
  _msg_='SAS table already exists.';
else do;
  call new(table,'',1,'y');
  _msg_='Table has been created.';
end;
```

Suppose your application is designed to run in batch mode and you do not want to generate any messages. You could use a null statement after THEN:

```
if (exist(table)) then;
  else call new(table,'',1,'y');
```

For more information, refer to *SAS Language Reference: Dictionary*.

Using the RETURN Statement

The RETURN statement stops the execution of the program section that is currently executing.

RETURN <*value*>;

The RETURN statement at the end of a reserved program section (FSEINIT ,INIT, MAIN, TERM, and FSETERM) sends control to the next program section in the sequence.

The first RETURN statement after a LINK statement returns control to the statement that immediately follows the LINK statement.

When the RETURN statement is encountered at the end of a window variable section, control returns to the next section in the program execution cycle. That next section may be another window variable section or it may be the MAIN section. When the current program execution cycle finishes, control returns to the application window.

The RETURN statement at the end of a method returns control to the calling program.

The RETURN statement for an ENTRY or METHOD block can return *value* if the ENTRY or METHOD statement contains RETURN=*data-type*. The returned value has no effect if control does not immediately return to the calling program.

For an example of the RETURN statement, see the example in "Branching to Another Entry (GOTO)" on page 72. For more explanation and an additional example, see "RETURN" on page 615.

Branching to a Labeled Section (LINK)

The LINK statement tells SCL to jump immediately to the specified statement label.

LINK *label*;

SCL then executes the statements from the statement *label* up to the next RETURN statement. The RETURN statement sends program control to the statement that immediately follows the LINK statement. The LINK statement and the *label* must be in the same entry.

The LINK statement can branch to a group of statements that contains another LINK statement; that is, you can nest LINK statements. You can have up to ten LINK statements with no intervening RETURN statements.

See "Branching to Another Entry (GOTO)" on page 72 for an example that includes LINK statements.

For more information, refer to *SAS Language Reference: Dictionary*.

Branching to Another Entry (GOTO)

You can use the GOTO statement to transfer control to another SAS/AF entry.

CALL GOTO (*entry*<, *action*<, *frame*>>);

Entry specifies a FRAME, PROGRAM, MENU, CBT, or HELP entry. By default, when the entry ends, control returns to the parent entry that was specified in *entry*. If a parent entry is not specified, then the window exits.

For example, suppose WORK.A.A.SCL contains the following code:

```
INIT:
  link SECTONE;
  put 'in INIT after link to SECTONE';
return;

SECTONE:
  put 'in SECTONE before link to TWO';
  link TWO;
  put 'in SECTONE before goto';
  call goto('work.a.b.frame');
  put 'in SECTONE after goto to frame';
return;

TWO:
  put 'in TWO';
return;
```

WORK.A.B.SCL contains the following code:

```
INIT:
  put 'in WORK.A.B.FRAME';
return;
```

If you compile WORK.A.B.FRAME and WORK.A.A.SCL, and then test WORK.A.A.SCL, you will see the following output:

```
in SECTONE before link to TWO
in TWO
in SECTONE before goto
in WORK.A.B.FRAME
```

The PUT statement in the INIT section of A.SCL and the last PUT statement in SECTONE are never executed. After WORK.A.B.FRAME is displayed and the user exits from the window, the program ends.

For more information, see "GOTO" on page 455.

Calling SCL Entries

SAS/AF software provides SCL entries for storing program modules. SCL programs can access a module that is stored in another SCL entry. They can pass parameters to the module and can receive values from the module. An SCL module can be used by any other SCL program.

You call an SCL module with a CALL DISPLAY routine that passes parameters to it and receives values that are returned by the SCL entry. The module's ENTRY statement receives parameters and returns values to the calling program.

For example, if you were creating an SCL module to validate amounts and rates that are entered by users, you could store the labeled sections in separate SCL entries named AMOUNT.SCL and RATE.SCL. Then, you could call either of them with a CALL DISPLAY statement like the following:

```
call display('methdlib.validate.amount.scl',amount,error);
```

For more information, see "DISPLAY" on page 350.

Stopping Execution of the Current Section

The STOP statement stops the execution of the current section. If a MAIN or TERM section is present, control passes to MAIN or TERM. For example, in the following program, control passes from INIT to SECTONE. Since X=1 is true, the STOP statement is executed, so control never passes to TWO. Control passes directly from the STOP statement in SECTONE to MAIN. The STOP statement at the end of MAIN has no effect, and control passes to TERM.

```
INIT:
  put 'beginning INIT';
  x=1;
  link SECTONE;
  put 'in INIT after link';
stop;

MAIN:
  put 'in MAIN';
stop;
```

```
SECTONE:
  put 'in SECTONE';
  if x=1 then stop;
  link TWO;
return;

TWO:
  put 'in TWO';
return;

TERM:
  put 'in TERM';
return;
```

This program produces the following output:

```
beginning INIT
in SECTONE
in MAIN
in TERM
```

For more information, see "STOP" on page 674.

Executing Methods

In object-oriented applications, methods are implemented in CLASS blocks or USECLASS blocks. These methods are usually invoked with dot notation. See "Accessing Object Attributes and Methods With Dot Notation" on page 119 for information about dot notation.

You can also send methods to an object by using CALL SEND, and you can send a method to a control in a FRAME entry by using CALL NOTIFY. See "SEND" on page 644 and "NOTIFY" on page 572 for more information.

Methods may also be stored in SCL, PROGRAM, or SCREEN entries. If the method is stored in an SCL entry, then call the method with the CALL METHOD routine. If the method is stored in a PROGRAM or SCREEN entry, you can use the LINK or GOTO statements to call it. See "Calling a Method That Is Stored in an SCL Entry" on page 13, "Branching to a Labeled Section (LINK)" on page 72, and "Branching to Another Entry (GOTO)" on page 72 for more information.

Using the CONTROL Statement

The CONTROL statement enables you to specify options that control the execution of labeled sections, the formatting of submit blocks, and whether an error in dot notation causes a program halt.

CONTROL *options*;

You can specify the following options with the CONTROL statement:

ALLCMDS | NOALLCMDS
: determines whether SCL can intercept procedure-specific or custom commands that are issued in the application. This option also determines if and when the MAIN section executes.

ALWAYS | NOALWAYS
determines whether the MAIN section executes if the user enters a command that SCL does not recognize.

ASIS NOASIS
determines whether SCL eliminates unnecessary spaces and line breaks before submit blocks are submitted.

BREAK *label* | NOBREAK
enables you to specify a labeled program section that will be executed if an interrupt or break occurs while your program is executing.

HALTONDOATTRIBUTE | NOHALTONDOTATTRIBUTE
determines whether execution halts if SCL finds an error in the dot notation that is used in your program.

ENDSAS | NOENDSAS
determines whether the TERM section executes when the user enters the ENDSAS or BYE commands.

ENDAWS | NOENDAWS
determines whether the TERM section executes when a user ends a SAS session by selecting the system closure menu in a FRAME entry that is running within the SAS application workspace.

ENTER | NOENTER
determines whether the MAIN section executes when the user presses the ENTER key or a function key without modifying a window variable.

ERROR | NOERROR
determines whether the MAIN section executes if a control or field contains a value that causes an attribute error.

LABEL | NOLABEL
determines whether the MAIN section executes before or after the window variable sections.

TERM | NOTERM
determines whether the TERM section executes even if a user does not modify any columns in the current row of the SAS table.

For more information, see “CONTROL” on page 302.

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

ISBN 1–58025–495–0

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

1st printing, October 1999