The Power of CALL EXECUTE

Transcript

The Power of CALL EXECUTE Transcript was developed by Kenny Sucher. Additional contributions were made by Cynthia Johnson, Warren Repole, and Russ Tyndall. Editing and production support was provided by the Curriculum Development and Support Department.

The Power of CALL EXECUTE Transcript

Book code E1513, course code RLSPCLEX, prepared date 16Dec2009. RLSPCLEX_001

ISBN 978-1-60764-380-7

Table of Contents

Lecture Description iv

Prerequisites v

Accessibility Tips v

The Power of CALL EXECUTE 1

1. The Basics of CALL EXECUTE 7
2. Generating Code Based on Data Set Values 16
3. Repetitive Operations 21
4. Passing Variable Values to a Macro Call 29
5. Inappropriate Uses of CALL EXECUTE 34

Appendix A Demonstration Programs A-1

1. Generating Data-Dependent Code A-3
2. Repetitive Operations A-4
3. Using CALL EXECUTE to Create Macro Calls A-5
4. Inappropriate Uses of CALL EXECUTE A-6

Lecture Description

This e-lecture shows how to use a powerful macro tool: CALL EXECUTE. The lecture reviews the basics of CALL EXECUTE and discusses some of the reasons to use it, as well as demonstrates situations when it isn't appropriate and why.

To learn more…

Prerequisites

Before viewing this lecture, learners should have completed the *SAS® Macro Language 2: Developing Macro Applications* course or have equivalent knowledge.

Accessibility Tips

If you are using a screen reader, such as Freedom Scientific's JAWS, you may want to configure your punctuation settings so that characters used in code samples (comma, ampersand, semicolon, percent) are announced. Typically, the screen reader default for the character & is to read "and." For clarity in code samples, you may want to configure your screen reader to read & as "ampersand." In addition, depending on your verbosity options, the character & might be omitted. The same is true for some commas before a code variable. To confirm code lines, you may choose to read some lines character by character. When testing this scenario with Adobe Acrobat Reader 9.1 and JAWS 10, ampersands before SAS macro names were announced only when in character-reading mode.

The Power of CALL EXECUTE

1. The Basics of CALL EXECUTE 7

2. Generating Code Based on Data Set Values 16

3. Repetitive Operations 21

4. Passing Variable Values to a Macro Call 29

5. Inappropriate Uses of CALL EXECUTE 34

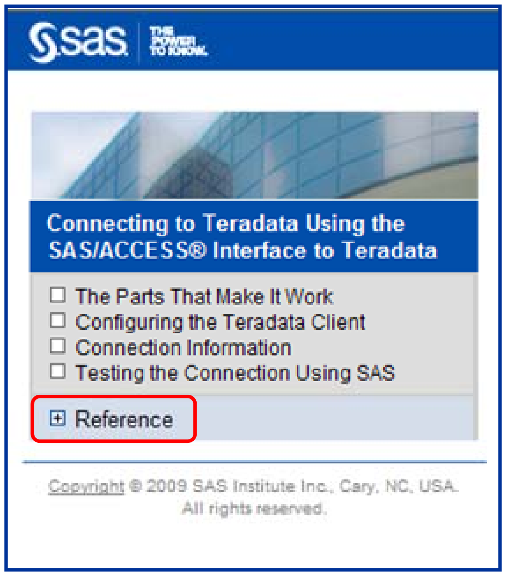
The Power of CALL EXECUTE

Hi and welcome to the SAS E-Lecture entitled *The Power of CALL EXECUTE*. My name is Kenneth, and I’m an instructor for SAS Education. I teach courses in many areas, including ones on macros and advanced macro topics. This lecture was developed for users with advanced macro programming knowledge to give them additional information about a powerful macro tool: CALL EXECUTE. This e-lecture will review the basics of CALL EXECUTE and discuss some of the reasons you’d want to use it, as well as demonstrate situations when it isn’t appropriate and why.

Reference Materials

To access the transcript for this lecture:

1. Go to the table of contents on the left side of the viewer.
2. Select **Reference**.
3. Select **Transcript**.


sas THE POWER TO KNOW.
Connecting to Teradata Using the SAS/ACCESS® Interface to Teradata
The Parts That Make It Work
Configuring the Teradata Client
Connection Information
Testing the Connection Using SAS
Reference
Copyright © 2009 SAS Institute Inc., Cary, NC, USA. All rights reserved.

2

Before we begin the lecture, let me mention that we have included a transcript so that you can print all of the technical information provided in this lecture. To access the transcript, select **Reference** and then **Transcript** in the table of contents on the left side of the viewer. You can print this transcript now for use when viewing the lecture or print it later to keep as a reference. Also, note that Appendix A in the transcript contains copies of the programs used for the demos in this lecture.

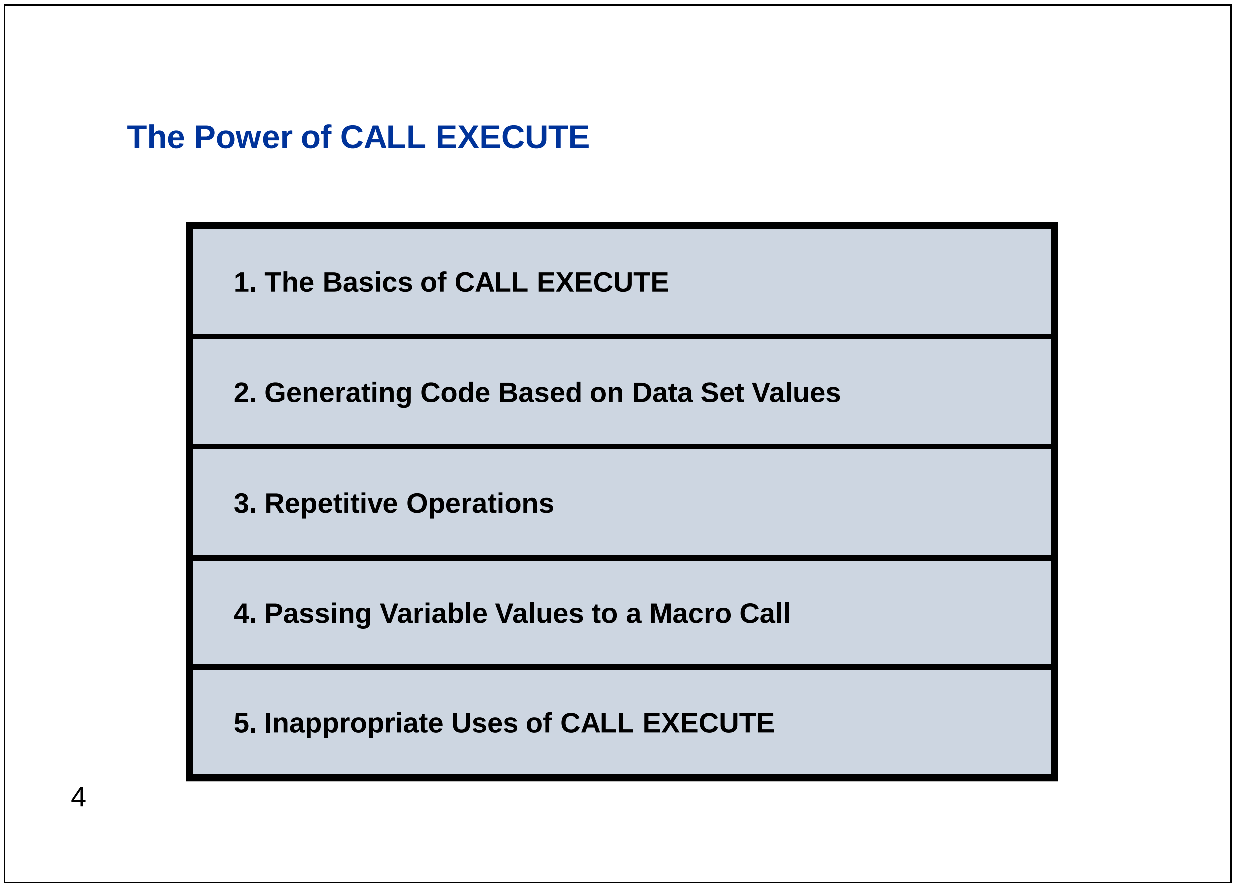
Navigation Help

For information on how to navigate this lecture:

1. Go to the upper-right corner of the browser.
2. Select **Help**.

3

If you need help with the navigation of this lecture, please select **Help** in the upper-right corner of the browser.



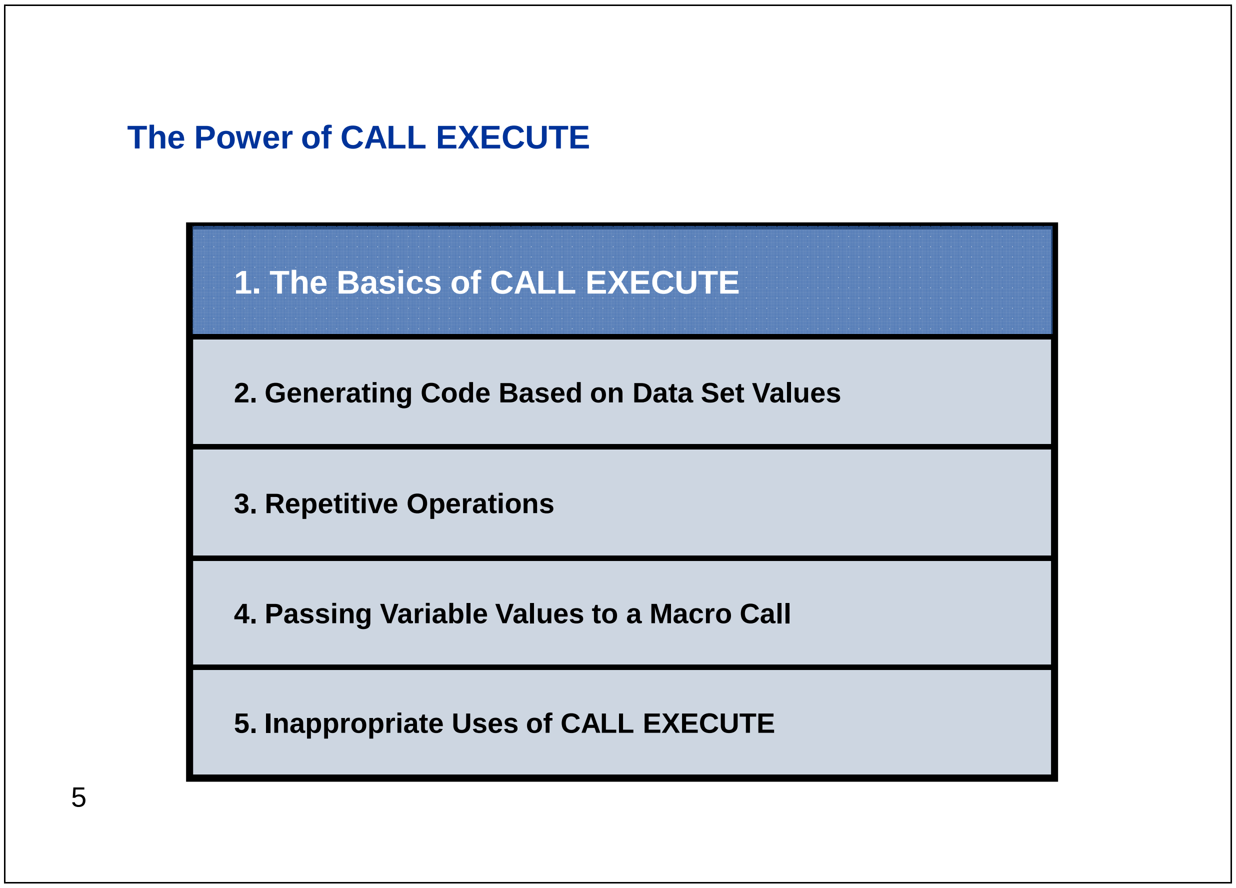
The Power of CALL EXECUTE

1. The Basics of CALL EXECUTE
2. Generating Code Based on Data Set Values
3. Repetitive Operations
4. Passing Variable Values to a Macro Call
5. Inappropriate Uses of CALL EXECUTE

4

In this e-lecture I intend to review the basics of CALL EXECUTE and expand on the routine. After that, we’ll see some reasons to use CALL EXECUTE, which include generating code based on data set values, taking advantage of repetitive operations, and passing DATA step variable values to a macro call, and we’ll see some examples. Lastly, I’ll discuss some of the pitfalls of CALL EXECUTE and show an alternative method.

1. The Basics of CALL EXECUTE

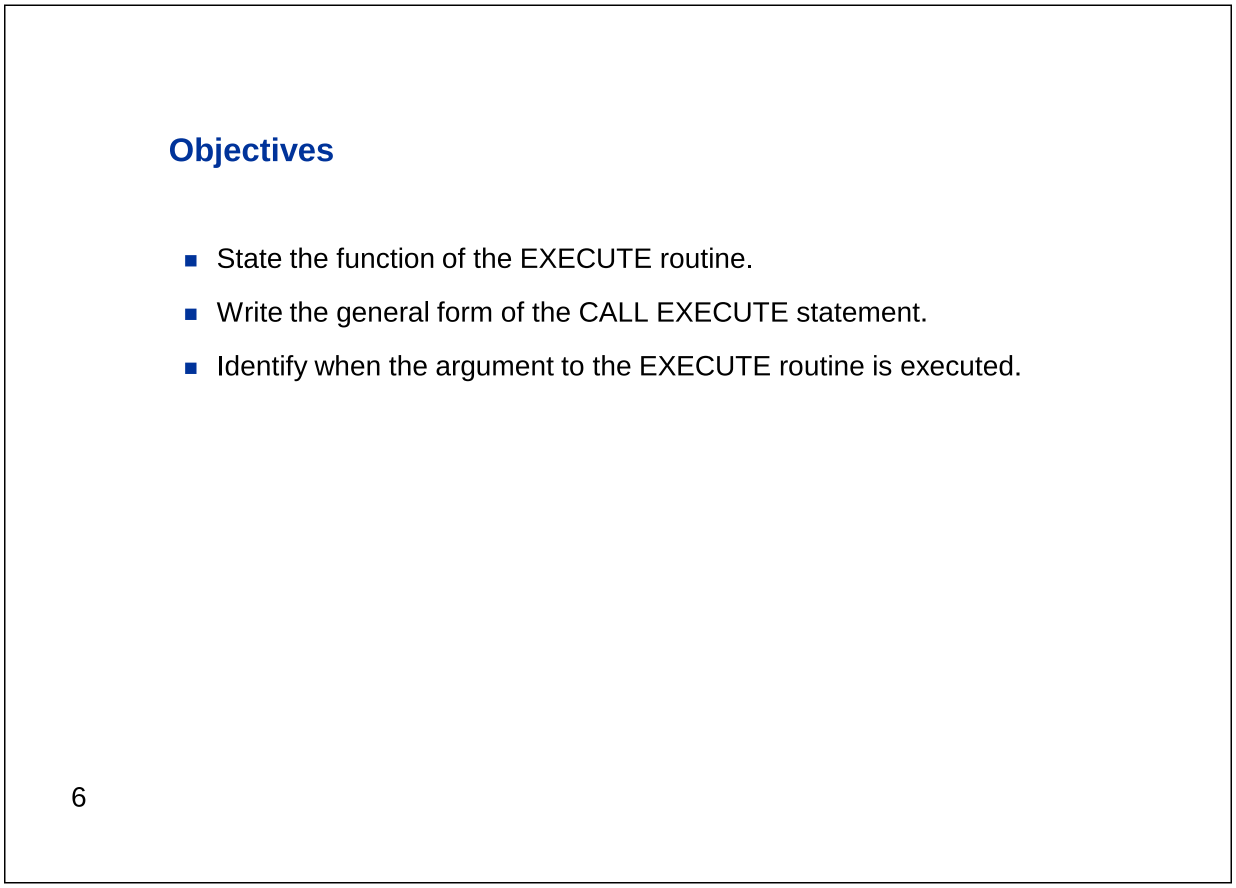


The Power of CALL EXECUTE

1. The Basics of CALL EXECUTE
2. Generating Code Based on Data Set Values
3. Repetitive Operations
4. Passing Variable Values to a Macro Call
5. Inappropriate Uses of CALL EXECUTE

5

This first section of the e-lecture is entitled “The Basics of CALL EXECUTE.”

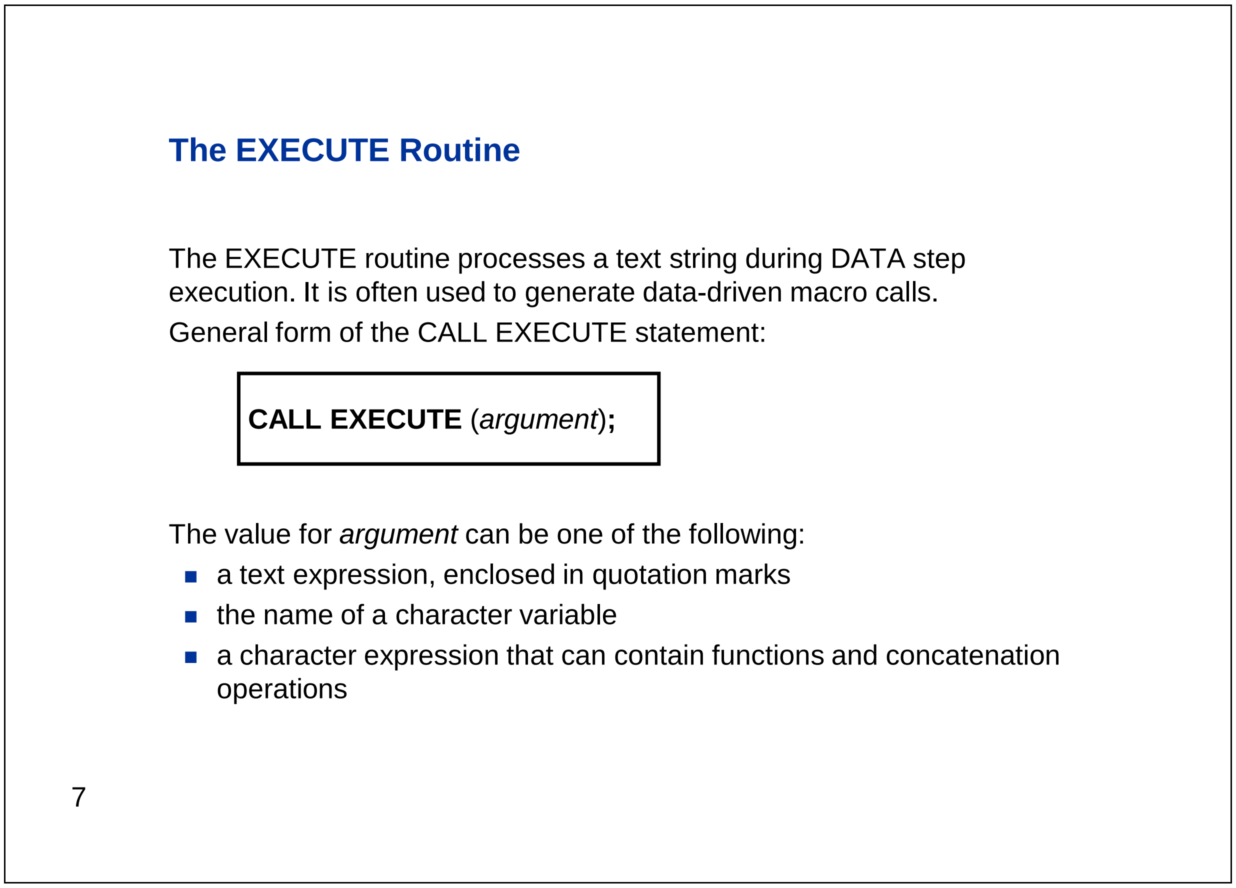


Objectives

- State the function of the EXECUTE routine.
- Write the general form of the CALL EXECUTE statement.
- Identify when the argument to the EXECUTE routine is executed.

6

In this first section, we will review the basics of CALL EXECUTE and show a simple example of the routine.



The EXECUTE Routine

The EXECUTE routine processes a text string during DATA step execution. It is often used to generate data-driven macro calls.

General form of the CALL EXECUTE statement:

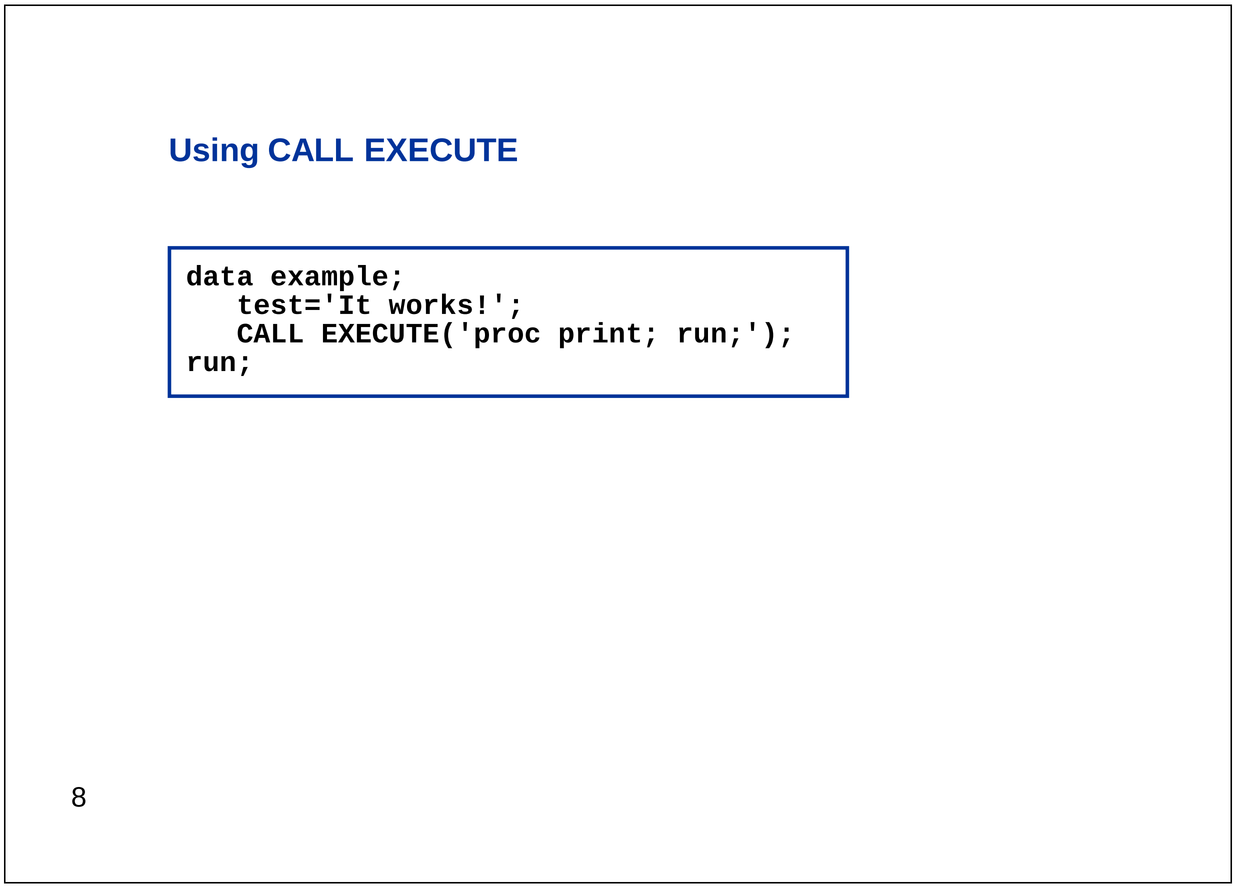
$\boxed{\textbf{CALL EXECUTE } (\textit{argument})\textbf{;}}$

The value for *argument* can be one of the following:

- a text expression, enclosed in quotation marks
- the name of a character variable
- a character expression that can contain functions and concatenation operations

7

The EXECUTE routine processes a text string during DATA step execution and can be used to generate data-driven macro calls. The general form of the CALL EXECUTE statement is shown here. The argument can be a character literal, a character variable, or a character expression.

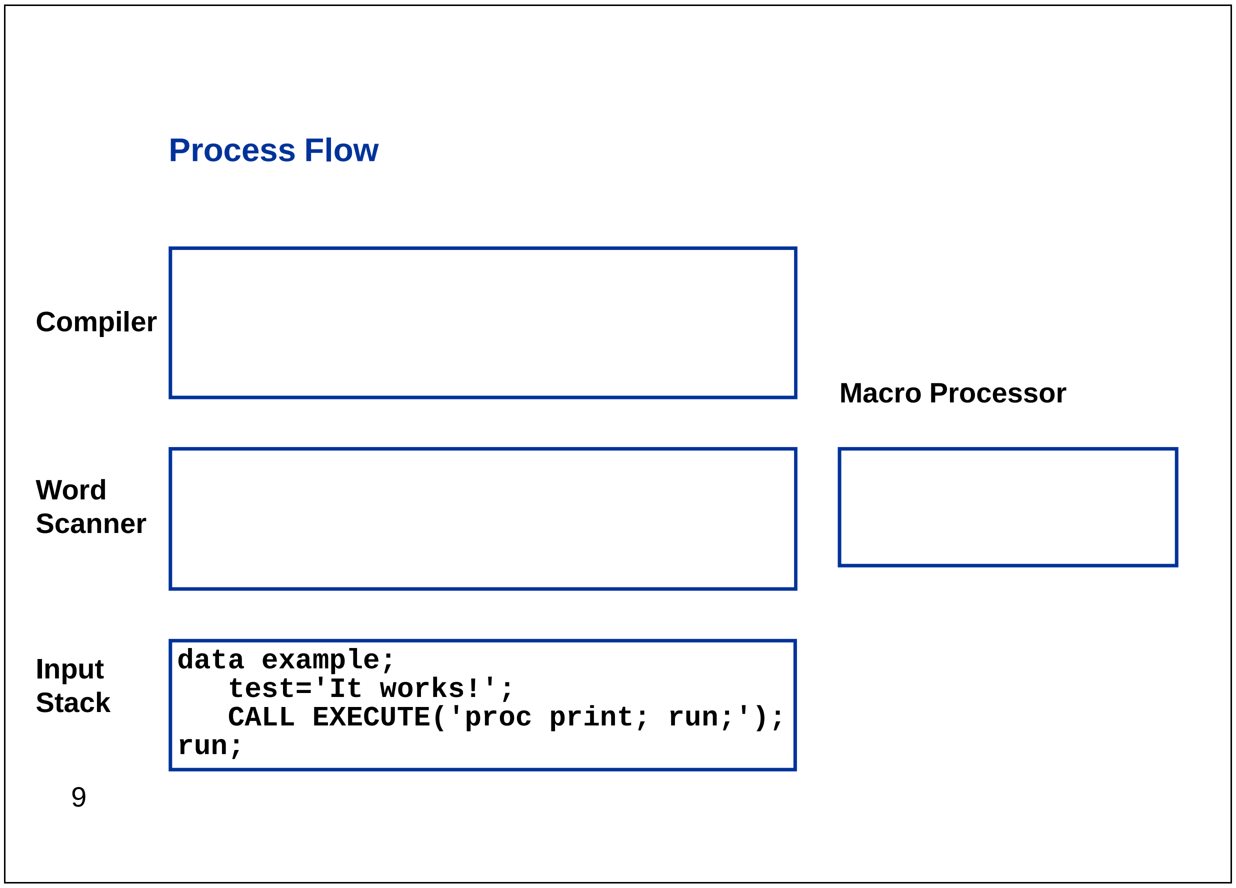


Using CALL EXECUTE

```
data example;
   test='It works!';
   CALL EXECUTE('proc print; run;');
run;
```

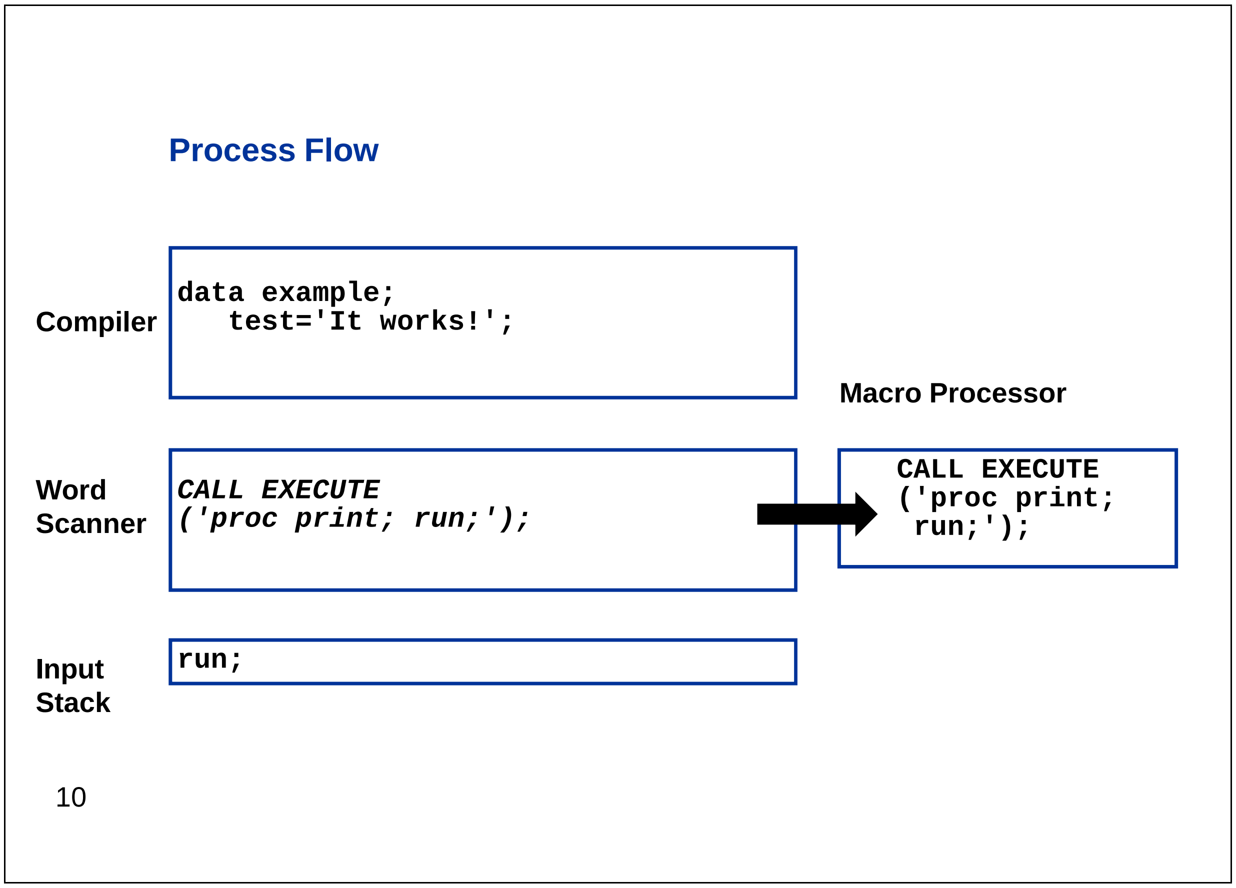
8

If the argument to the EXECUTE routine does not contain macro triggers, the text is inserted into the input stack as additional program code that will execute **after** the current DATA step. Such is the case in the example here: the PROC PRINT step would be executed **after** the DATA step finishes executing. Let’s see what this looks like behind the scenes.



```
data example;
   test='It works!';
   CALL EXECUTE('proc print; run;');
run;
```

When this DATA step is submitted, the code enters the input stack.



Process Flow

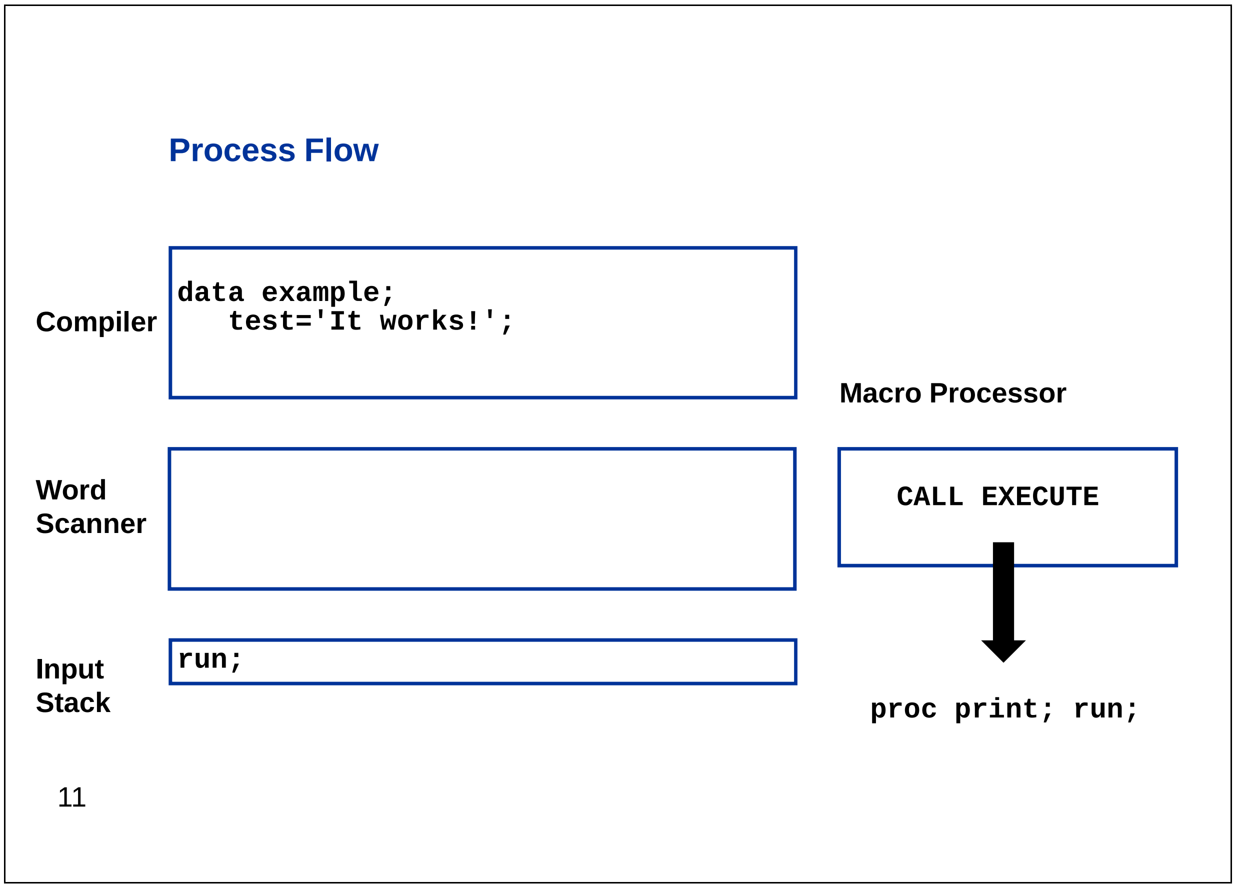
```
data example;
   test='It works!';
```

```
CALL EXECUTE
('proc print; run;');
```

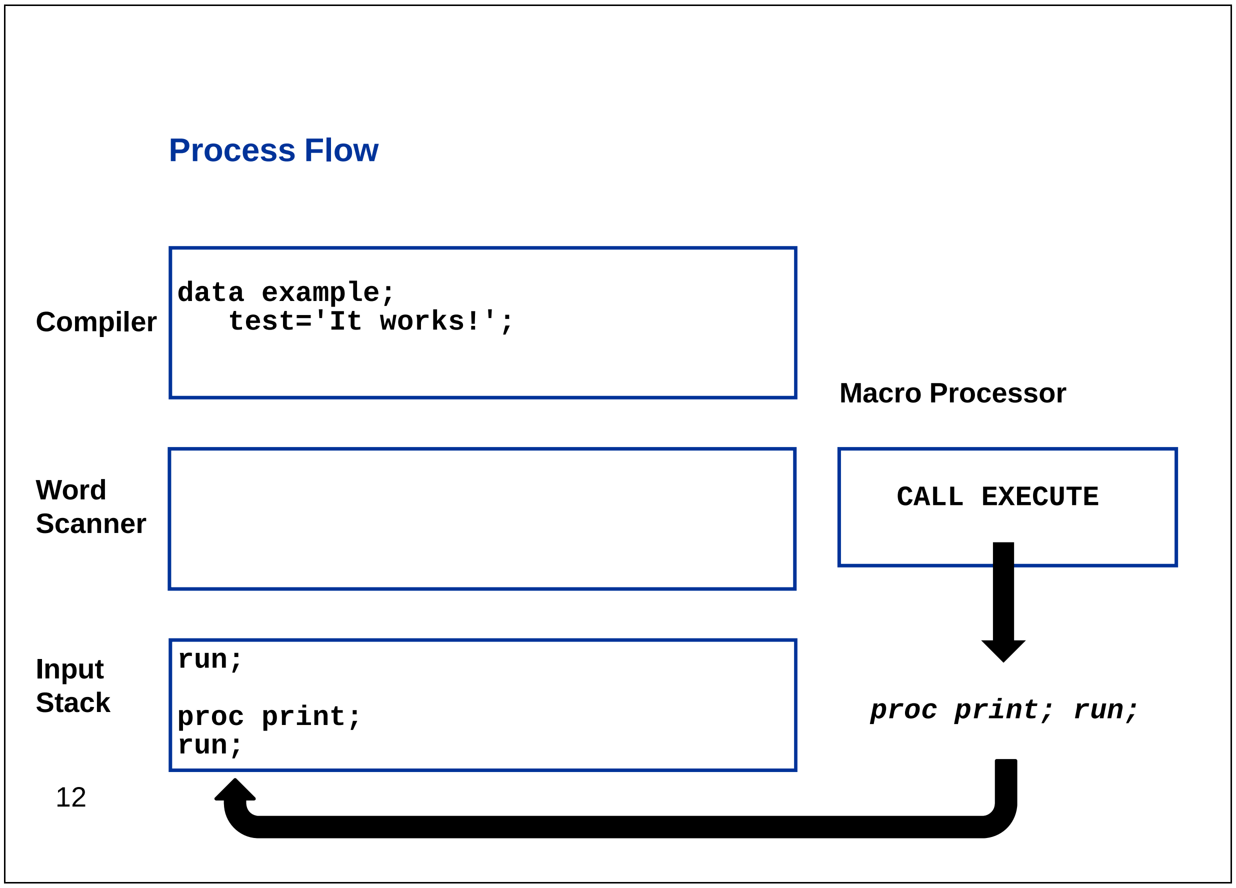
```
run;
```

```
CALL EXECUTE
('proc print;
 run;');
```

As the code gets tokenized in the word scanner and sent to the compiler, the word scanner encounters the CALL EXECUTE statement. It then sends the request to the macro processor.



The macro processor then sends the generated code to the input stack.

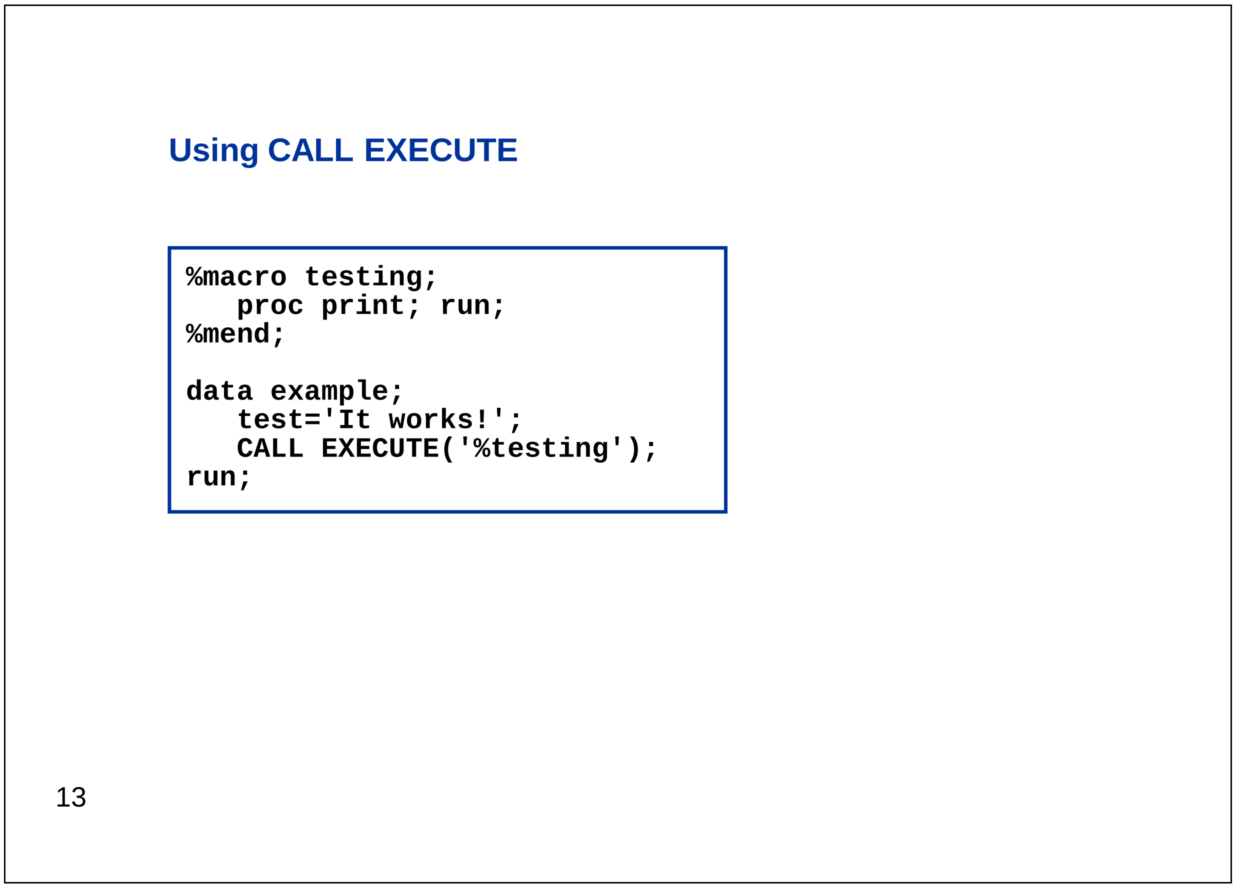

Process Flow

```
data example;
   test='It works!';
```

```
run;

proc print;
run;
```

```
proc print; run;
```

Notice that the generated code gets placed after the end of the DATA step in the input stack. So, it will get executed after the DATA step finishes.



Using CALL EXECUTE

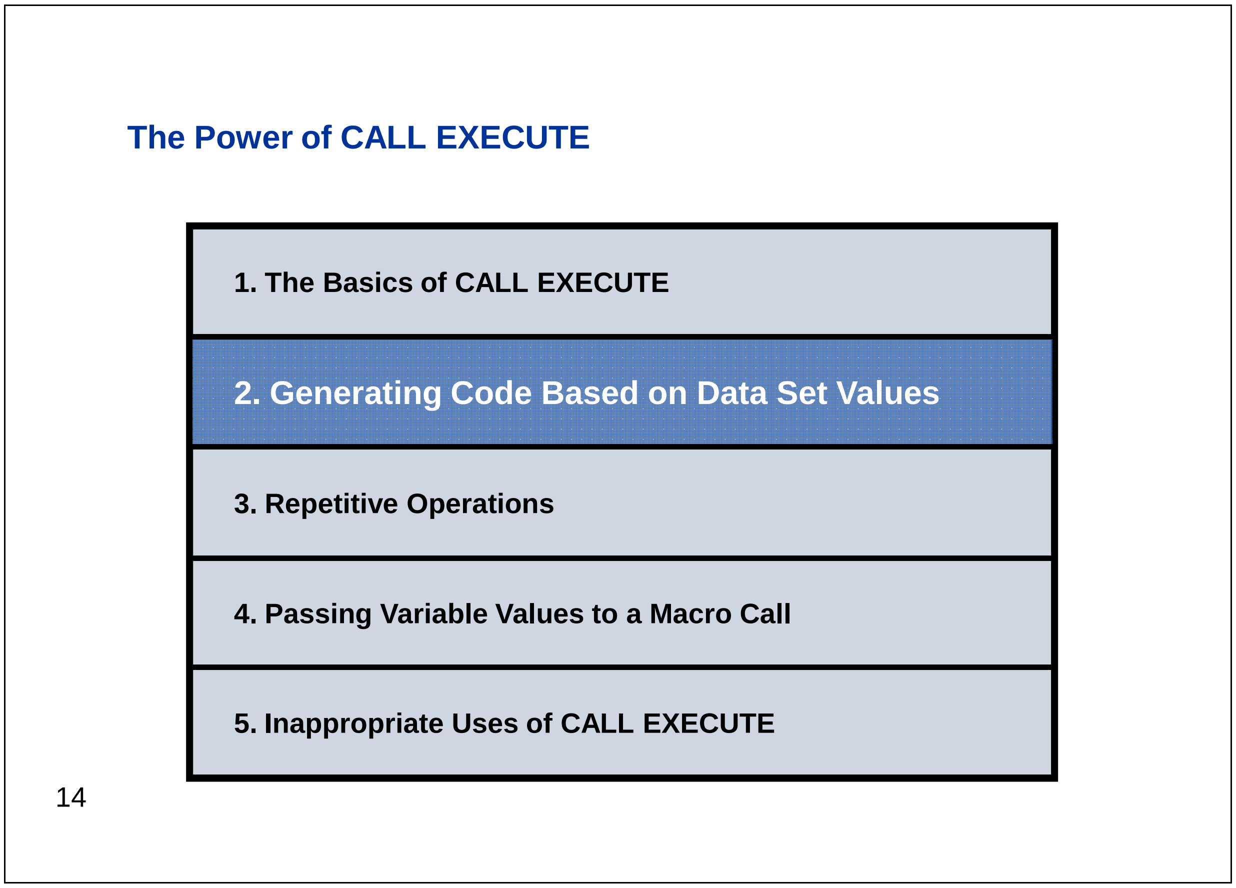
```
%macro testing;
   proc print; run;
%mend;

data example;
   test='It works!';
   CALL EXECUTE('%testing');
run;
```

13

Now if the argument to the EXECUTE routine resolves to a macro call, the macro executes immediately and DATA step execution pauses while the macro executes. Macro language statements within the macro definition are executed, and all other SAS code generated by the macro is inserted into the input stack to execute after the current DATA step. If the macro “testing” in the example had any macro language statements in it, they would get immediately executed. Because the macro definition contains no macro triggers, the code gets sent to the input stack after the DATA step, so we get the exact same results with both of these examples. So, the %TESTING macro executes immediately here, but the SAS code generated by the macro is submitted after the DATA step finishes.

The rest of this e-lecture will cover more useful examples of using CALL EXECUTE, as well as some pitfalls.

2. Generating Code Based on Data Set Values



The Power of CALL EXECUTE

1. The Basics of CALL EXECUTE
2. Generating Code Based on Data Set Values
3. Repetitive Operations
4. Passing Variable Values to a Macro Call
5. Inappropriate Uses of CALL EXECUTE

14

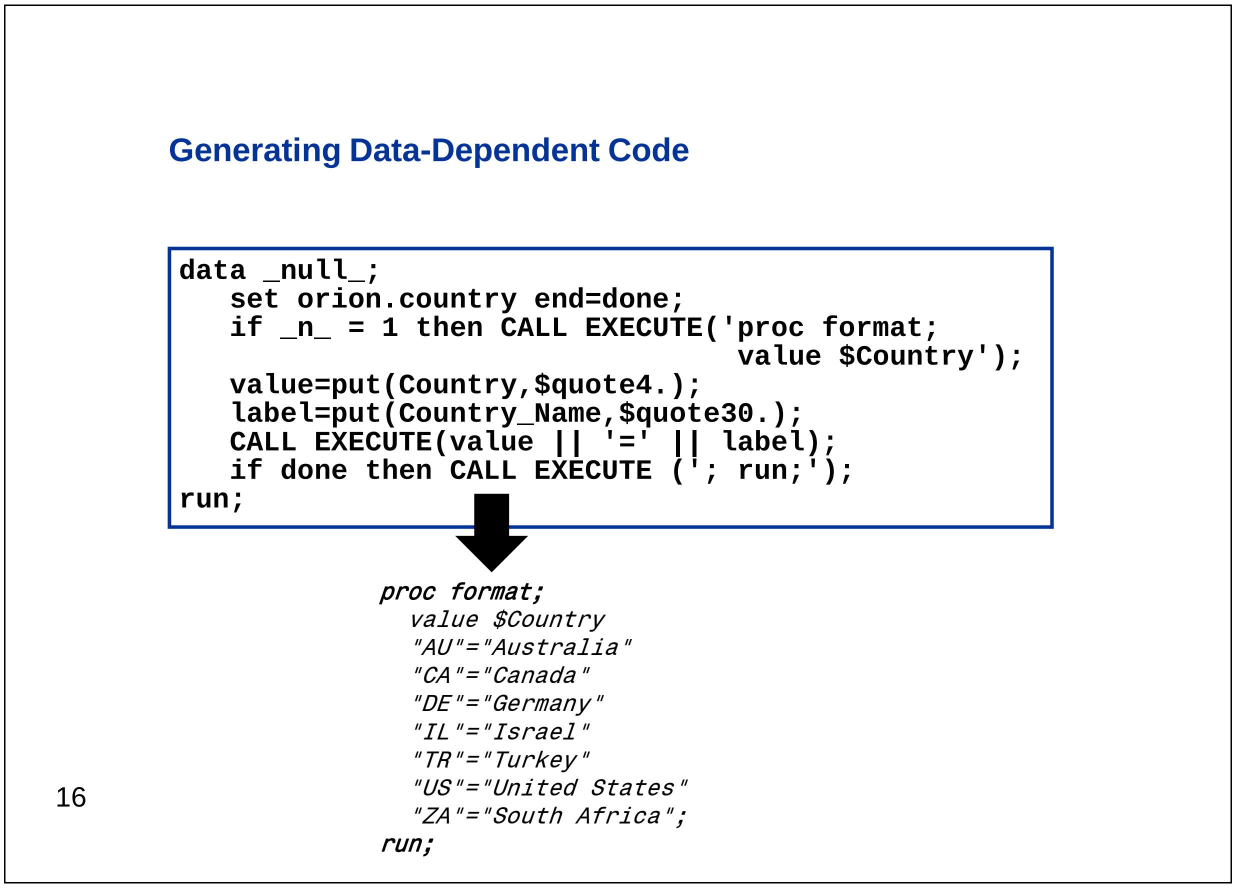
The next section will focus on using CALL EXECUTE to generate code based on data set values.

Objectives

- Generate code based on data set values.

15

In this section, I will demonstrate and explain an example of generating code based on variable values in a data set using CALL EXECUTE.



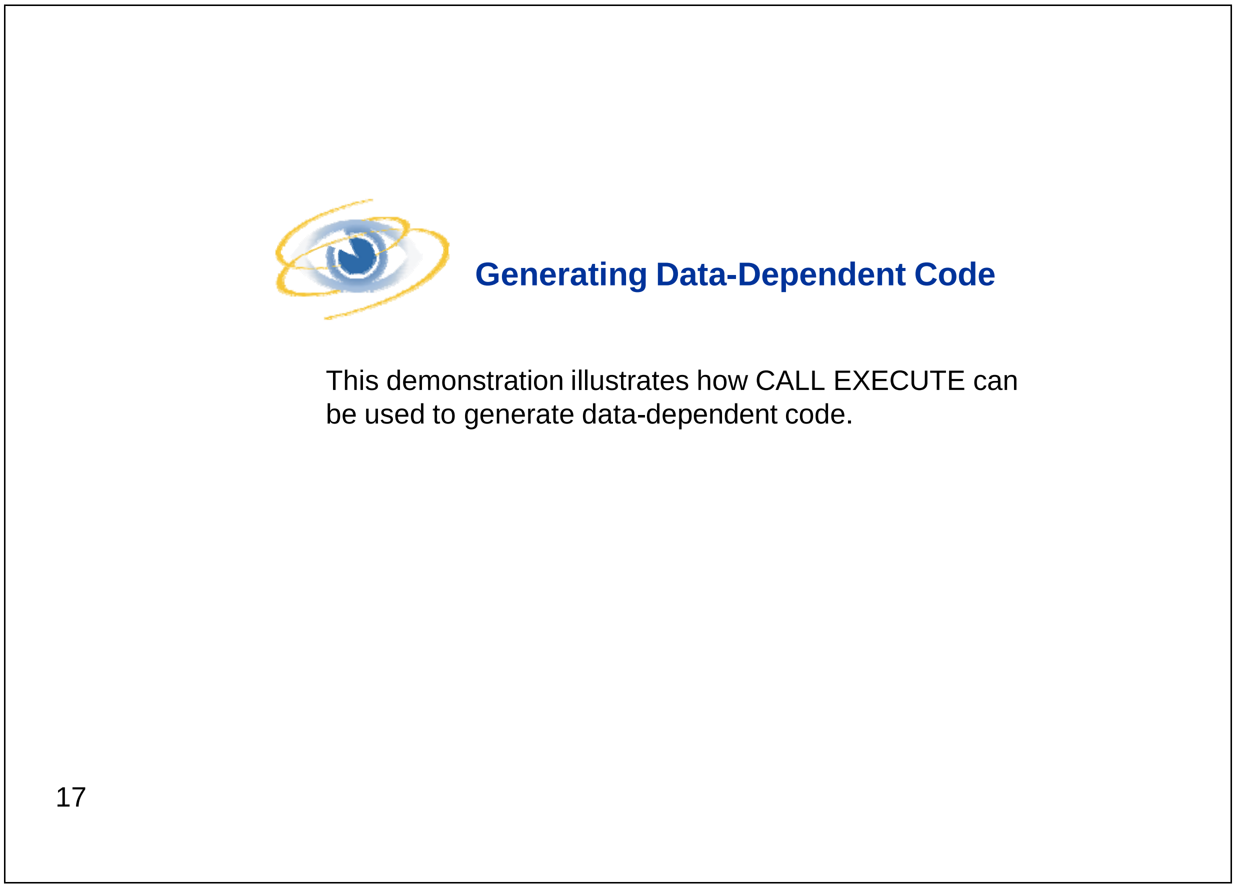
Generating Data-Dependent Code

```
data _null_;
   set orion.country end=done;
   if _n_ = 1 then CALL EXECUTE('proc format;
                                 value $Country');
   value=put(Country,$quote4.);
   label=put(Country_Name,$quote30.);
   CALL EXECUTE(value || '=' || label);
   if done then CALL EXECUTE ('; run;');
run;
```

```
proc format;
  value $Country
  "AU"="Australia"
  "CA"="Canada"
  "DE"="Germany"
  "IL"="Israel"
  "TR"="Turkey"
  "US"="United States"
  "ZA"="South Africa";
run;
```

16

One task you might need to perform is using the values of a DATA step variable or variables to generate further SAS code – for example, if you need to create a data-dependent PROC FORMAT step that includes values and labels based on a reference table. The following example does just that. Let’s jump into SAS and take a look.



Generating Data-Dependent Code

This demonstration illustrates how CALL EXECUTE can be used to generate data-dependent code.

17

Just a quick word about the data used in this e-lecture…

The examples you'll see will use a few tables from a relational database of a fictitious sports and outdoor retailer named Orion Star Sports & Outdoors.

Here we create a simple look-up application using PROC FORMAT to assign full country names to country abbreviations. The following DATA step uses the EXECUTE routine to generate a PROC FORMAT step directly from the **`Country`** table (which includes both the country abbreviations and their full names – the rest of the data sets in the database only include the abbreviated information).

We can apply this format later on; let's focus now on creating it.

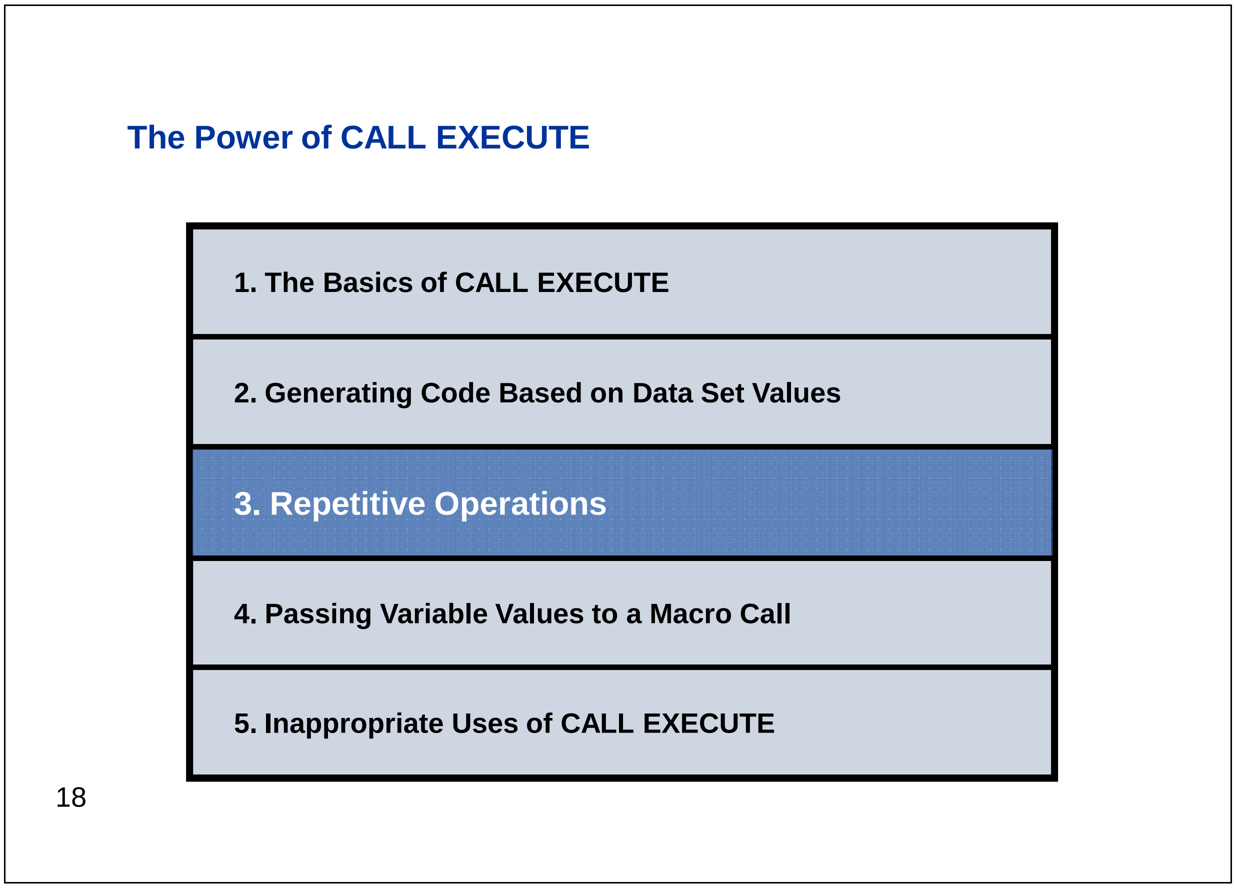
1. Notice first we're using a DATA _NULL_ statement here. The sole intent is to submit a data-dependent PROC FORMAT step after the DATA step executes, so no new data set is needed.
2. In our SET statement, we've included an END= option to mark the end of file. We need this to tell SAS when to include the final closing syntax of the PROC FORMAT step we're generating.
3. The beginning of the PROC FORMAT step is generated just once during the first iteration of the DATA step by using _N_=1. Here we build the literal PROC FORMAT statement and the beginning of the VALUE statement with the name of our user-defined format: $COUNTRY.
4. We then create two variables, **`value`** and **`label`**. **`value`** is a character variable created using the PUT function to grab all the different **`Country`** values and surround them with double quotes using the $QUOTE format. The **`label`** variable is also created using the PUT function to grab the various

`Country_Name` values that correspond to the **`Country`** abbreviations. CALL EXECUTE is used to generate the various **`Country`** values and **`Country _Name`** labels for the PROC FORMAT syntax.

5. Lastly, after the final iteration of the DATA step, CALL EXECUTE generates a final semicolon to end the VALUE statement started at the beginning of the DATA step and a RUN statement to end the step. Remember the code generated by the EXECUTE routine does **not** get executed until **after** the DATA step is finished.

At this point the generated PROC FORMAT code is run, and the supplier format is created as you can see in the log. We'll apply this format in a later example. Let's jump back into the slides and get into Section 3.

3. Repetitive Operations

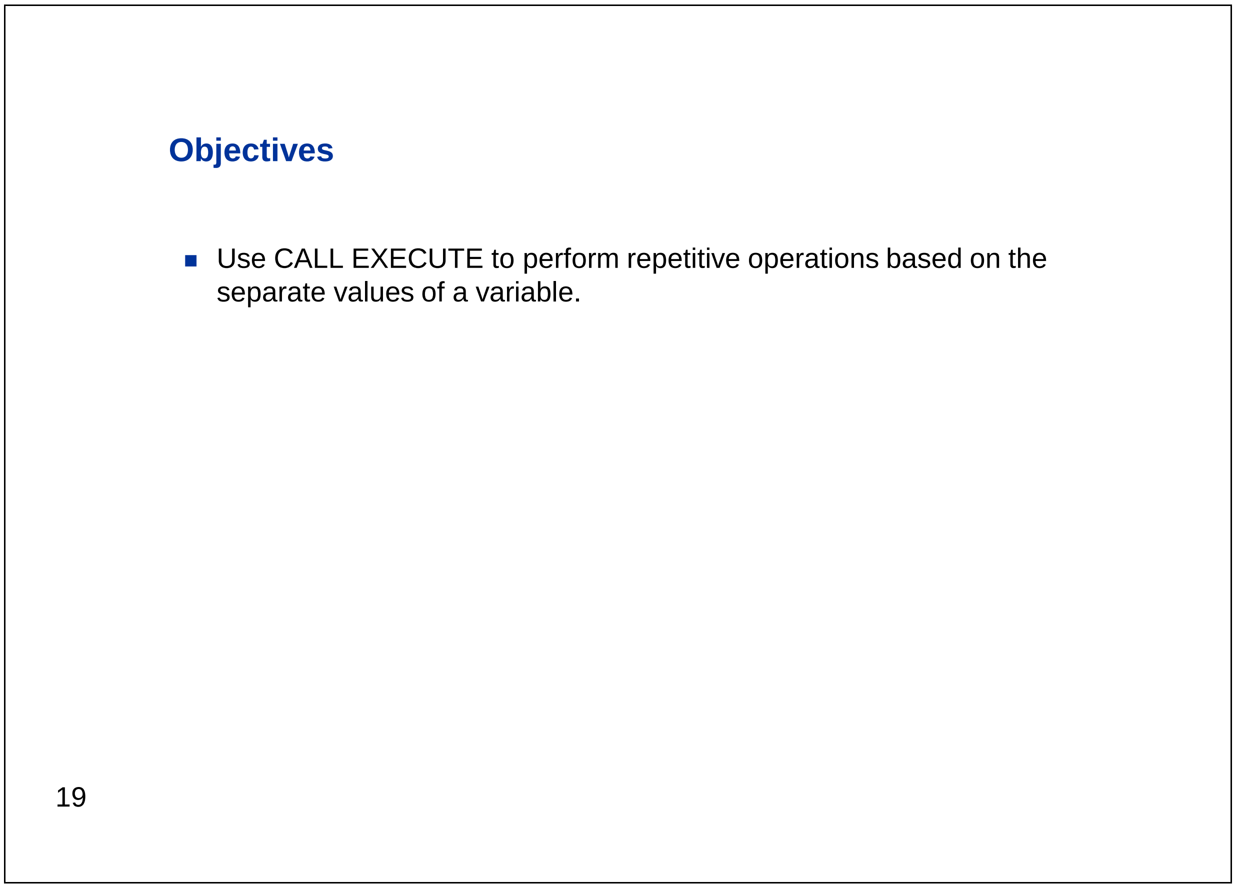


The Power of CALL EXECUTE

1. The Basics of CALL EXECUTE
2. Generating Code Based on Data Set Values
3. Repetitive Operations
4. Passing Variable Values to a Macro Call
5. Inappropriate Uses of CALL EXECUTE

18

This third section will cover another useful reason for using CALL EXECUTE: repetitive operations.

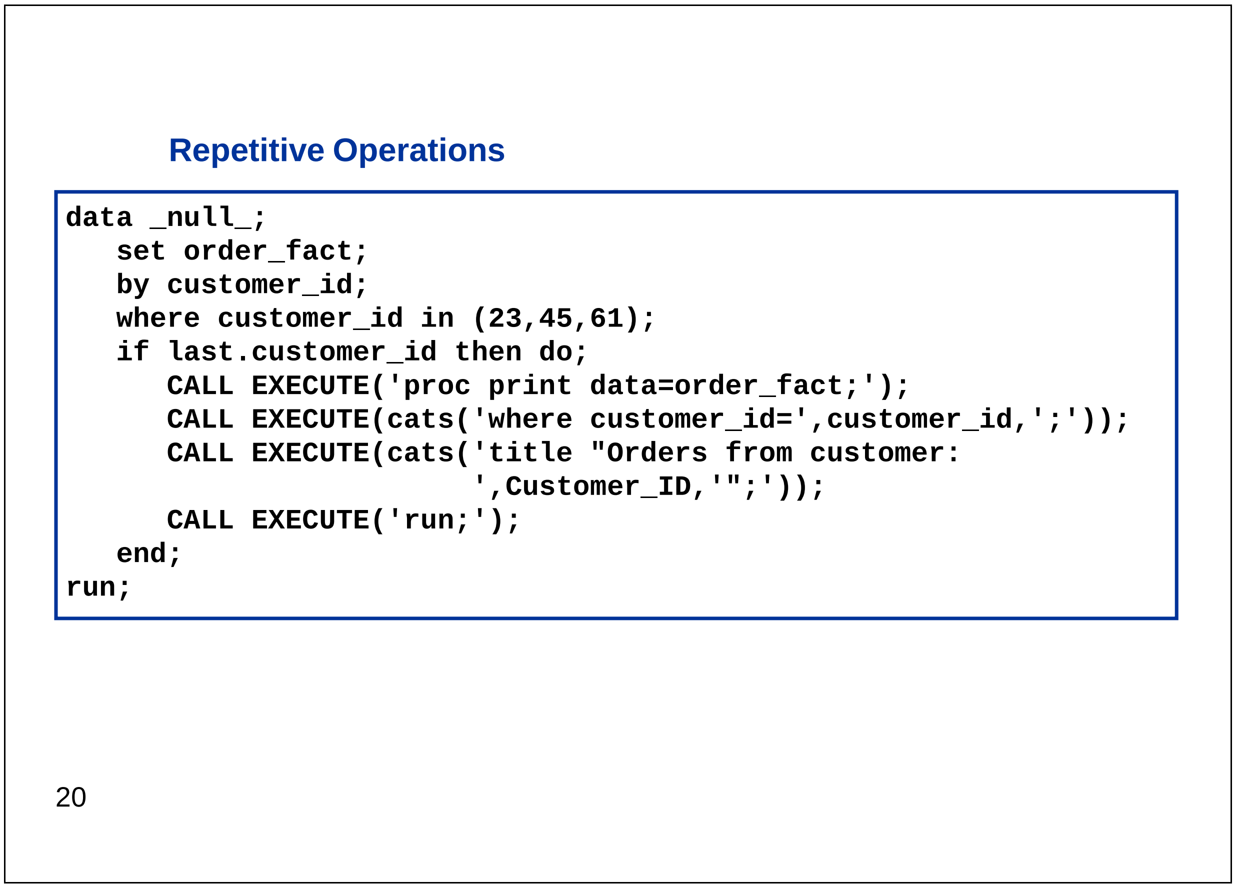


Objectives

- Use CALL EXECUTE to perform repetitive operations based on the separate values of a variable.

19

In this section, I will demonstrate the use of CALL EXECUTE to repetitively perform the same operation based on the separate values of a variable.

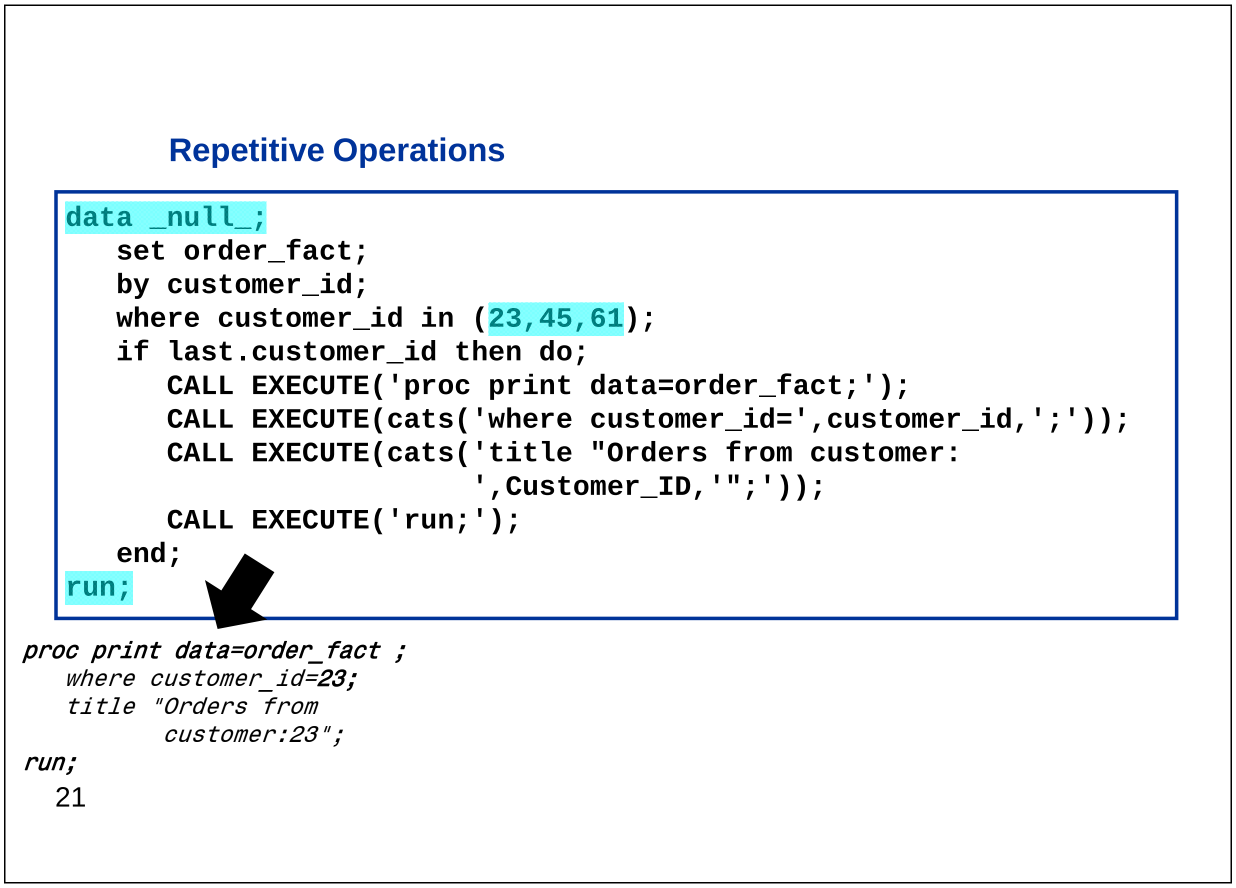


Repetitive Operations

```
data _null_;
   set order_fact;
   by customer_id;
   where customer_id in (23,45,61);
   if last.customer_id then do;
      CALL EXECUTE('proc print data=order_fact;');
      CALL EXECUTE(cats('where customer_id=',customer_id,';'));
      CALL EXECUTE(cats('title "Orders from customer:
                        ',Customer_ID,'";'));
      CALL EXECUTE('run;');
   end;
run;
```

20

Using CALL EXECUTE, we can repeatedly execute the same step, replacing a key part or parts of each step it creates. It’s sort of like placing your code in a loop. Actually, it’s exactly like placing your code in a loop. You’re placing it in the DATA step loop and using variable values to change what each repetitive operation does. For example, the above code replaces the need for BY-group processing in PROC PRINT. It generates a separate PROC PRINT step for each unique value of a variable – or, in this case, each customer based on the **`last.Customer_ID`** variable. The simplified program is being shown here solely to demonstrate the CALL EXECUTE process. The power of CALL EXECUTE will be illustrated in an upcoming demonstration. In this example code, SAS will generate a separate PROC PRINT when the DATA step reaches the last record for the three selected customers.



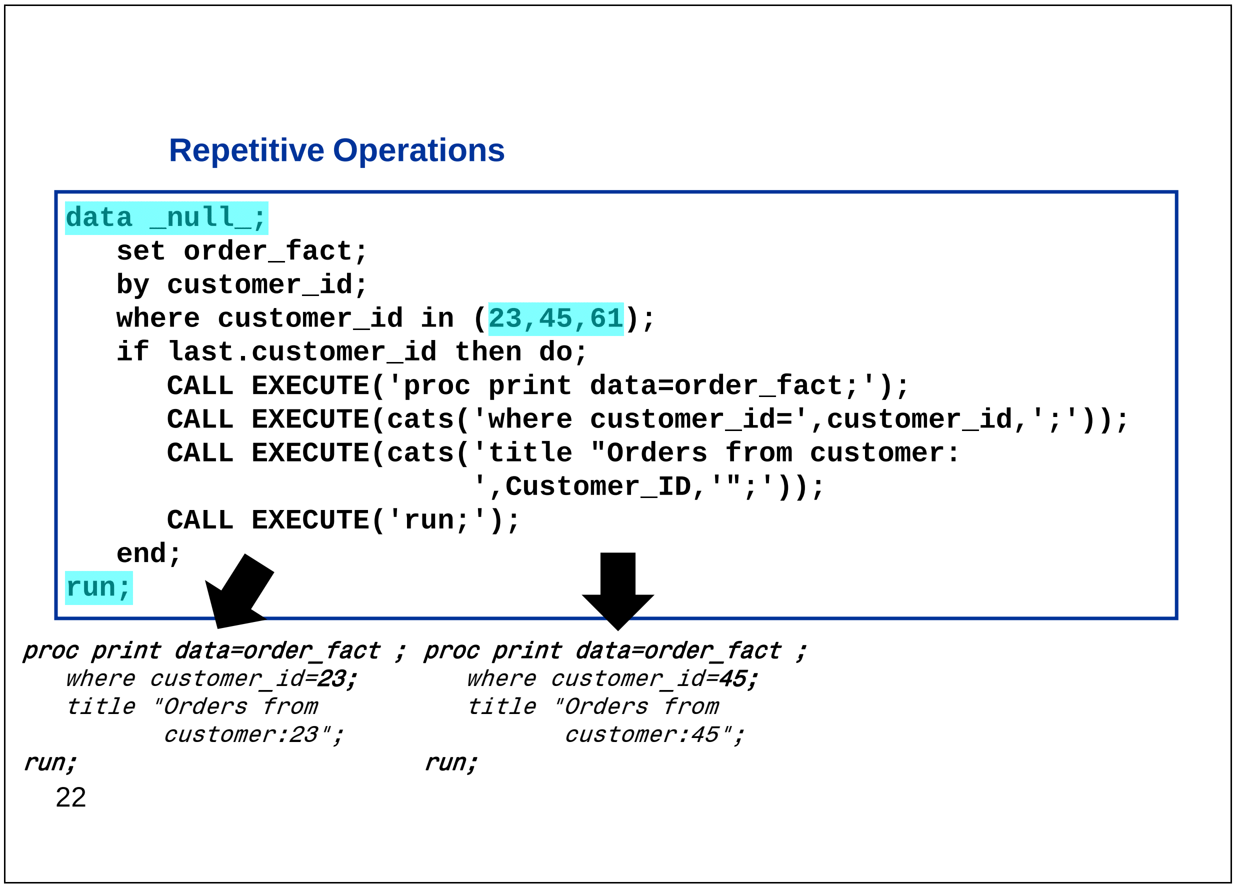
Repetitive Operations

```
data _null_;
   set order_fact;
   by customer_id;
   where customer_id in (23,45,61);
   if last.customer_id then do;
      CALL EXECUTE('proc print data=order_fact;');
      CALL EXECUTE(cats('where customer_id=',customer_id,';'));
      CALL EXECUTE(cats('title "Orders from customer:
                        ',Customer_ID,'";'));
      CALL EXECUTE('run;');
   end;
run;
```

```
proc print data=order_fact ;
   where customer_id=23;
   title "Orders from
         customer:23";
run;
```

21

A PROC PRINT for customer 23 …



Repetitive Operations

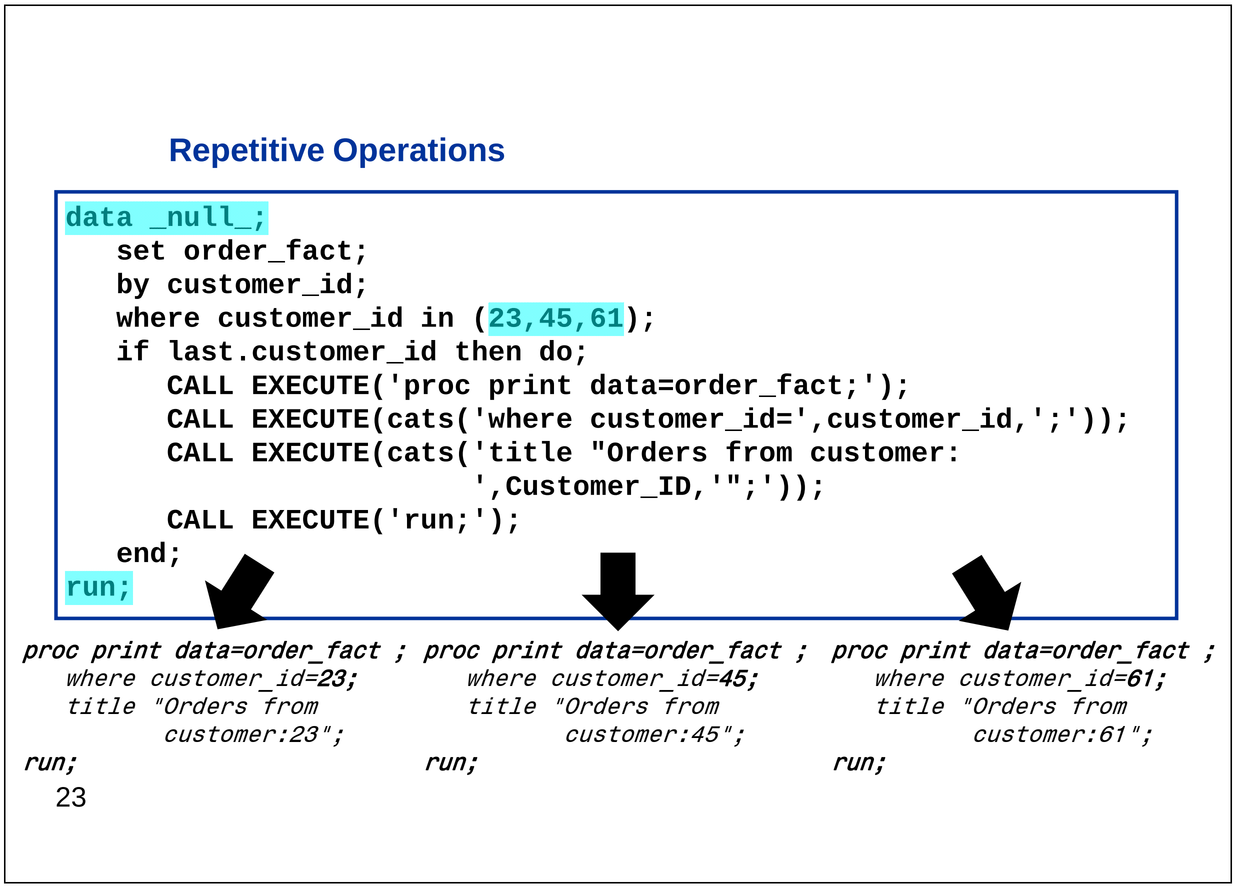
```
data _null_;
   set order_fact;
   by customer_id;
   where customer_id in (23,45,61);
   if last.customer_id then do;
      CALL EXECUTE('proc print data=order_fact;');
      CALL EXECUTE(cats('where customer_id=',customer_id,';'));
      CALL EXECUTE(cats('title "Orders from customer:
                  ',Customer_ID,'";'));
      CALL EXECUTE('run;');
   end;
run;
```

```
proc print data=order_fact ;
   where customer_id=23;
   title "Orders from
         customer:23";
run;
```

```
proc print data=order_fact ;
   where customer_id=45;
   title "Orders from
         customer:45";
run;
```

22

A PROC PRINT for customer 45 …



Repetitive Operations

```
data _null_;
   set order_fact;
   by customer_id;
   where customer_id in (23,45,61);
   if last.customer_id then do;
      CALL EXECUTE('proc print data=order_fact;');
      CALL EXECUTE(cats('where customer_id=',customer_id,';'));
      CALL EXECUTE(cats('title "Orders from customer:
                        ',Customer_ID,'";'));
      CALL EXECUTE('run;');
   end;
run;
```

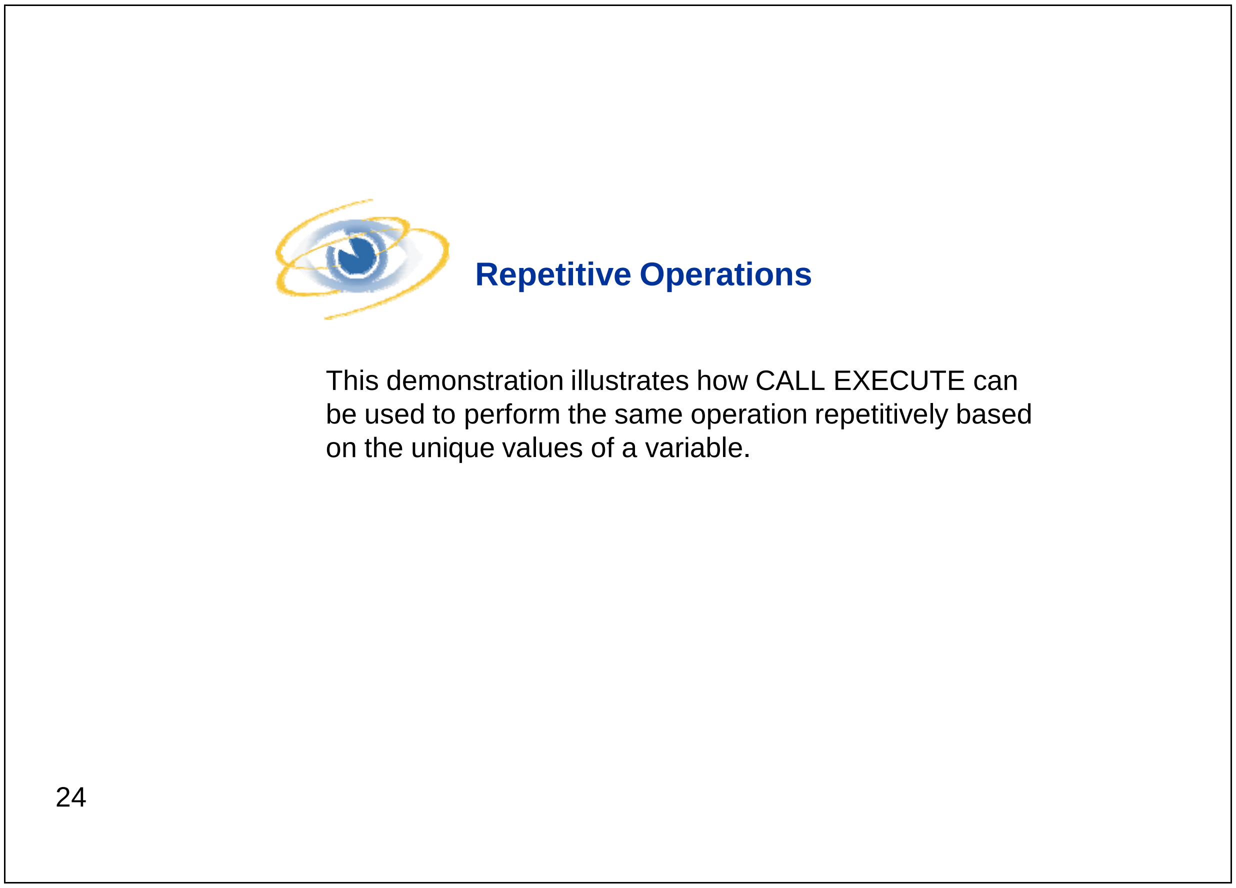
```
proc print data=order_fact ;
   where customer_id=23;
   title "Orders from
          customer:23";
run;
```

```
proc print data=order_fact ;
   where customer_id=45;
   title "Orders from
          customer:45";
run;
```

```
proc print data=order_fact ;
   where customer_id=61;
   title "Orders from
          customer:61";
run;
```

23

And a PROC PRINT for customer 61. Using this method, we can include additional information about an individual customer for the various PROC PRINT steps (such as order totals), which we will do in the demonstration. Let’s jump into SAS and take a look at some code similar to this – but a bit more elaborate.



Repetitive Operations

This demonstration illustrates how CALL EXECUTE can be used to perform the same operation repetitively based on the unique values of a variable.

24

The example here takes a selection of customers who have placed orders (maybe customers who have inquired about their orders) and places them in a sorted subset. This subset is then read into a DATA step and uses both FIRST./LAST. group processing to compute order information about the customers and then CALL EXECUTE to generate a separate PROC PRINT for each customer with the calculated information included in the different TITLE statements. This approach avoids the need of having to sort and perform BY-group processing on a large amount of data and also avoids using a lot of macros in the code to produce customer-specific TITLE statements and so on. Let’s go through this program in more detail.

1. First, we create a simple macro variable to represent the subset of customers whose orders we are interested in.
2. We then resolve this macro variable in a WHERE statement in a PROC SORT to both subset and group these customers’ orders by their IDs.
3. This sorted subset is then read into a DATA step.
    a. The DATA statement uses the _NULL_ option because we don’t need an output SAS data set. We’re using the DATA step to simply calculate customer information and produce customer-specific PROC PRINTs.
    b. When we set by **`customer_id`**, we create our FIRST./LAST. variables that enable us to perform BY-group processing.
    c. When we get to the first order by a customer, we reset our order count and order total variables and then use sum statements to add up this information.

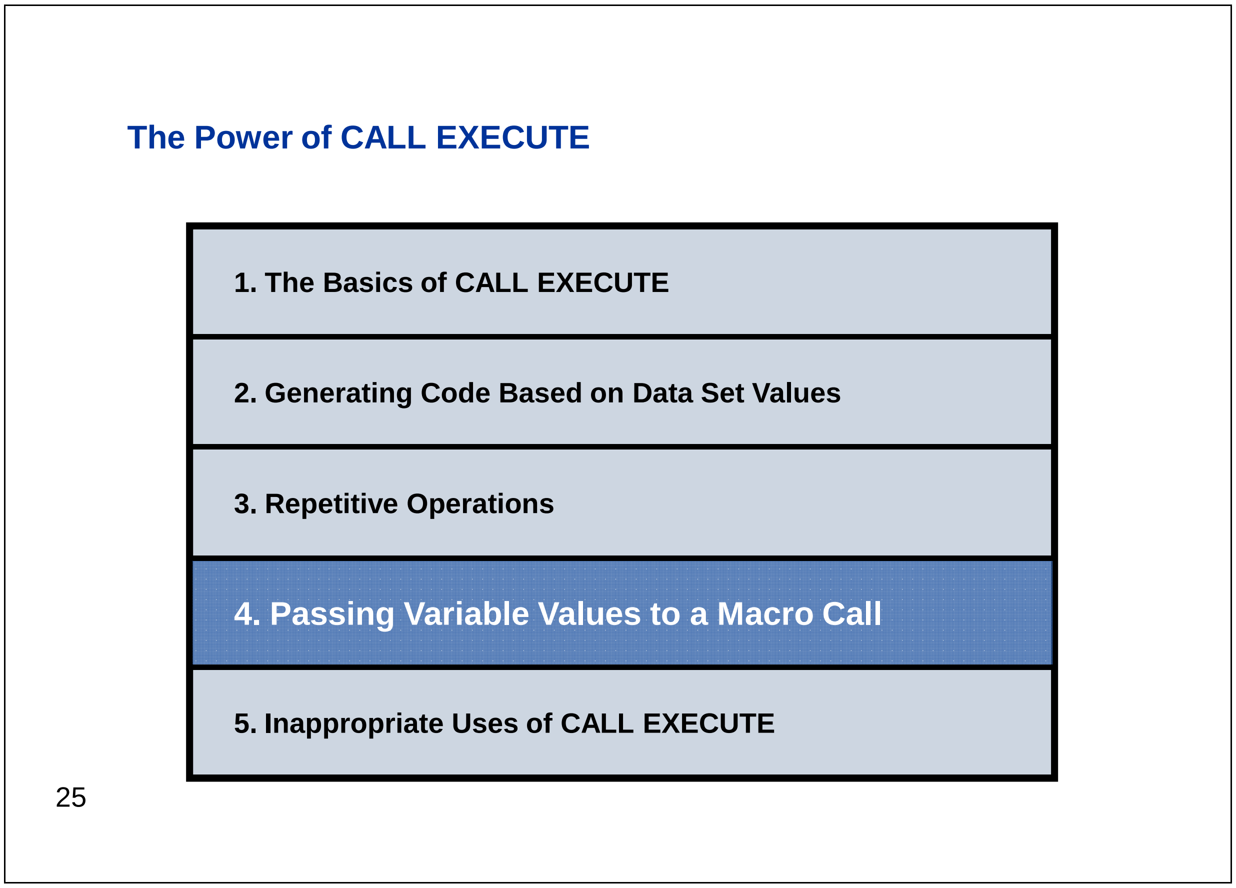
d. When we get to the last order for each customer, we generate the various CALL EXECUTE statements to build a unique PROC PRINT step for each customer. These PROC PRINT steps use the variables that are coming in from our input data set, as well as our order count and total variables we’re generating during DATA step execution to create customer-specific TITLE and WHERE statements.

After the DATA step is finished executing, there will be five PROC PRINT steps submitted and executed for the five customers added to our LIST macro variable. Each PROC PRINT will display a separate customer-specific title that includes the customer ID, order count, and order total.

Let’s look at the log to see the various PROC PRINTs that were generated. And here’s the output for these PROC PRINTs….

We could do something similar to this application using a series of macro variables and macro DO loops, but the nice thing about this method is it doesn’t require any advanced macro knowledge. Let’s return now to the slides and into the next section.

4. Passing Variable Values to a Macro Call

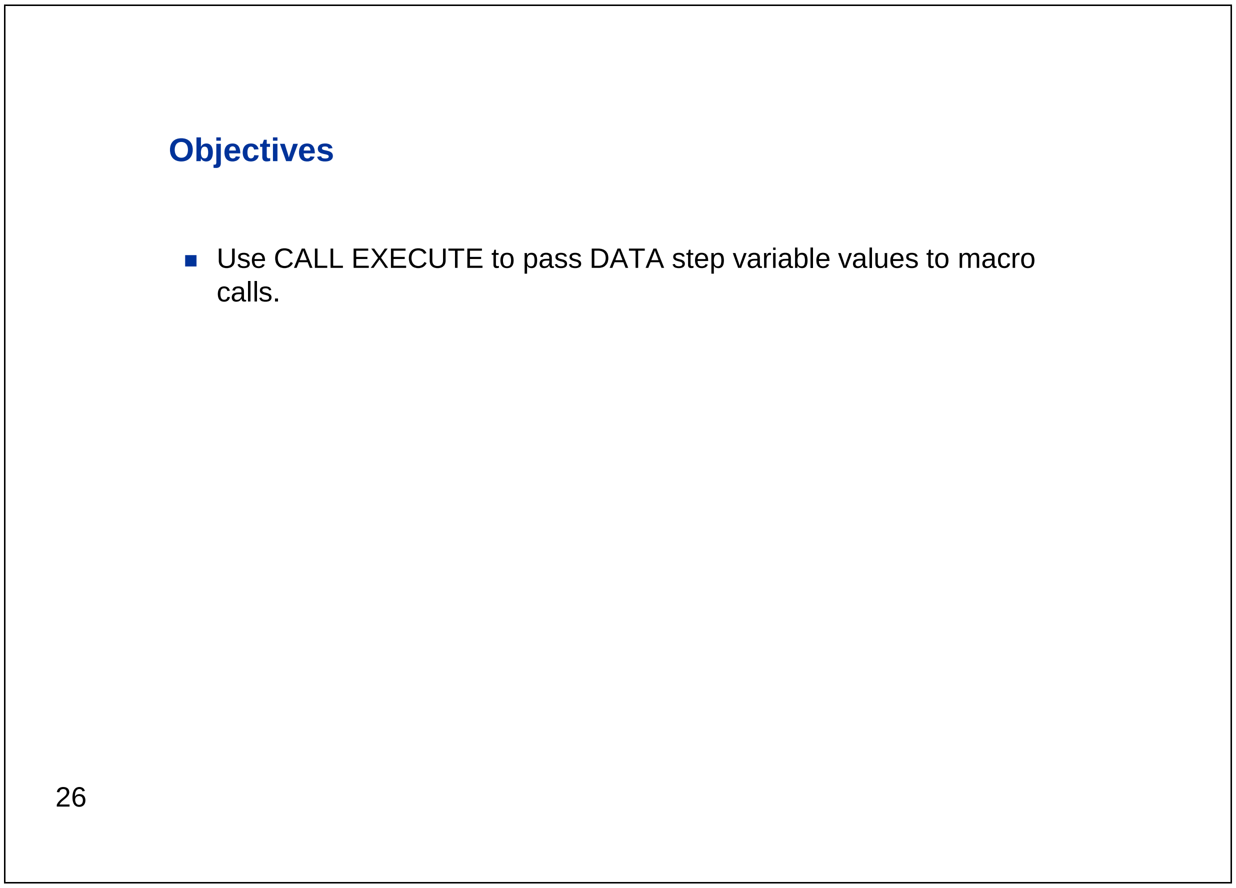


The Power of CALL EXECUTE

1. The Basics of CALL EXECUTE
2. Generating Code Based on Data Set Values
3. Repetitive Operations
4. Passing Variable Values to a Macro Call
5. Inappropriate Uses of CALL EXECUTE

25

The fourth section of this e-lecture is entitled “Passing Variable Values to a Macro Call.”

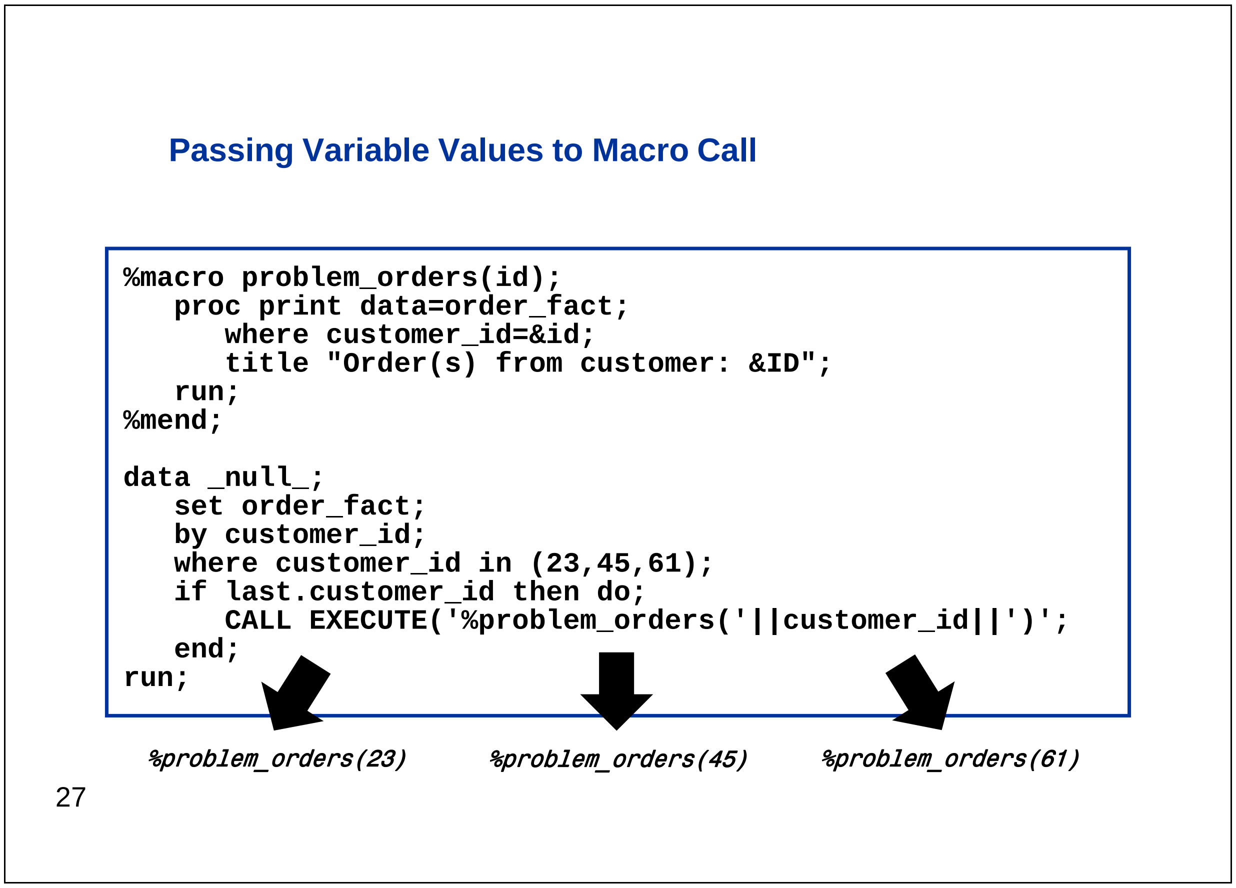


Objectives

- Use CALL EXECUTE to pass DATA step variable values to macro calls.

26

In this section, I will demonstrate the method of passing DATA step variable values to macro calls using CALL EXECUTE.



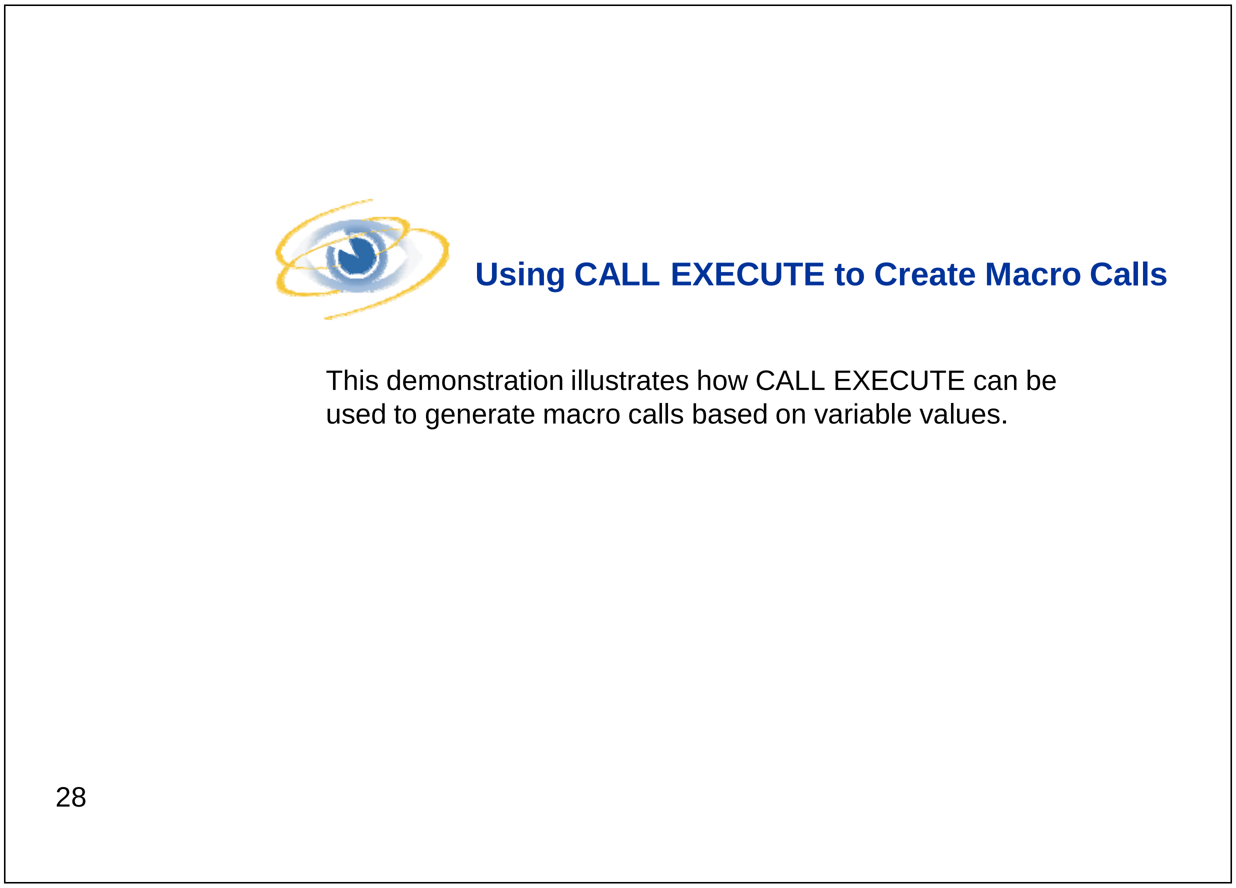
Passing Variable Values to Macro Call

```
%macro problem_orders(id);
   proc print data=order_fact;
      where customer_id=&id;
      title "Order(s) from customer: &ID";
   run;
%mend;

data _null_;
   set order_fact;
   by customer_id;
   where customer_id in (23,45,61);
   if last.customer_id then do;
      CALL EXECUTE('%problem_orders('||customer_id||')');
   end;
run;
```

```
%problem_orders(23)      %problem_orders(45)      %problem_orders(61)
```

27

The code here does exactly what we needed it to do in the previous example. It generates a separate PROC PRINT for each unique value of a variable during each pass of the DATA step. Or in this case, a separate macro call – each calling a macro with a parameter value that represents a unique customer ID. The macro in this case stores the PROC PRINT (that we previously used the EXECUTE routine to generate). Let’s jump into SAS and take a look at a similar but more developed program.



Using CALL EXECUTE to Create Macro Calls

This demonstration illustrates how CALL EXECUTE can be used to generate macro calls based on variable values.

28

This demonstration illustrates how CALL EXECUTE can be used to generate macro calls based on variable values.

So again, the following code has the same setup as the previous demonstration. The big difference here is that instead of using CALL EXECUTE to build the PROC PRINT code, we use it to generate macro calls to a macro that includes the PROC PRINT code.

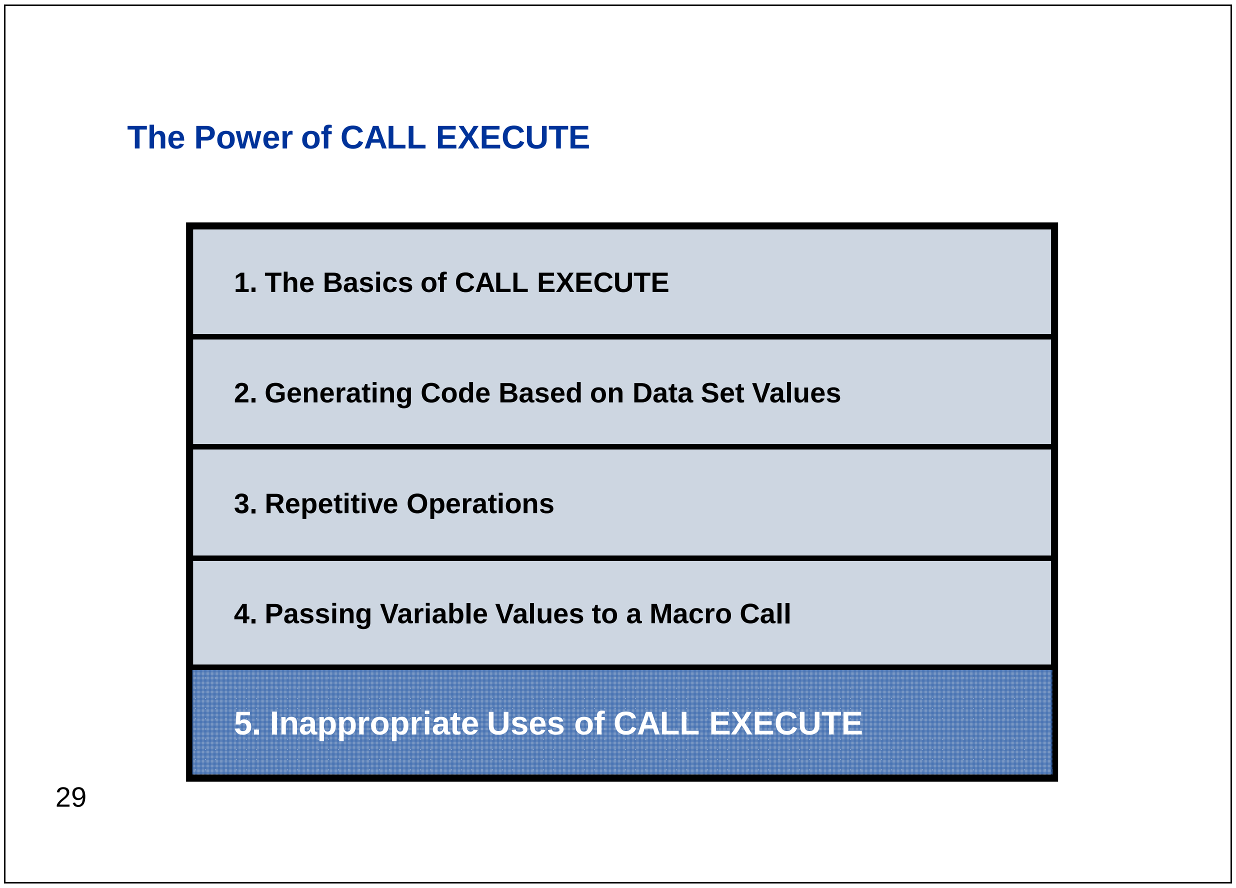
1. Notice the PROBLEM_ORDERS macro here. It has three parameters for the customer information we need to include in our reports: customer IDs, order counts, and order totals. The PROC PRINT then subsets based on the customer ID and includes the additional information in the TITLE statement.
2. We use the DATA step shown here to call this macro with the appropriate customer-specific information. As in the previous example, we use FIRST./LAST. processing to calculate the number of orders and the total amount spent. We then place this information, along with **`Customer_ID`**, in the CALL EXECUTE statement. The macro invocation gets immediately executed here instead of waiting until the DATA step finishes executing; this doesn't cause a problem in this example but could if the macro that was called included CALL SYMPUT statements. We'll see an example of that in a bit. I've actually included two CALL EXECUTE statements in this solution; the second one is the correct form and I'll explain why.

Notice that we're using the CATS function in the example. This strings together the various parts of our macro call. The CATS function concatenates the arguments separated by commas and removes any leading or trailing blanks. Let’s run this program with the second (correct) CALL EXECUTE commented out. It generates **some** of the reports, and we get a few errors in the log that “More positional parameters [were] found than defined.” So what’s happening here?

If you remember the results from the last example, some of the customers we were looking at had spent $1,000 or over. The PUT function in the CALL EXECUTE statement formats the amount and adds dollar signs and commas. The commas that are added for some of the customers are being included in the macro invocation, which looks like an additional argument is being submitted to the macro call and throws everything off. The fix here is using the %STR macro function, and the trick is to make sure the %STR is part of the literal strings (in quotes) so that they’re not executed until the macro is called (not when it’s being created by the EXECUTE routine). Put simply: be careful of commas in your variable values.

3. Let’s delete the first CALL EXECUTE and use the second (correct) form. Notice now that we get all the reports and we don’t see any errors in the log. Now that we’ve seen some reasons to use CALL EXECUTE, let’s go back into the slides and into the next section to see an example of when it isn’t appropriate to use it.

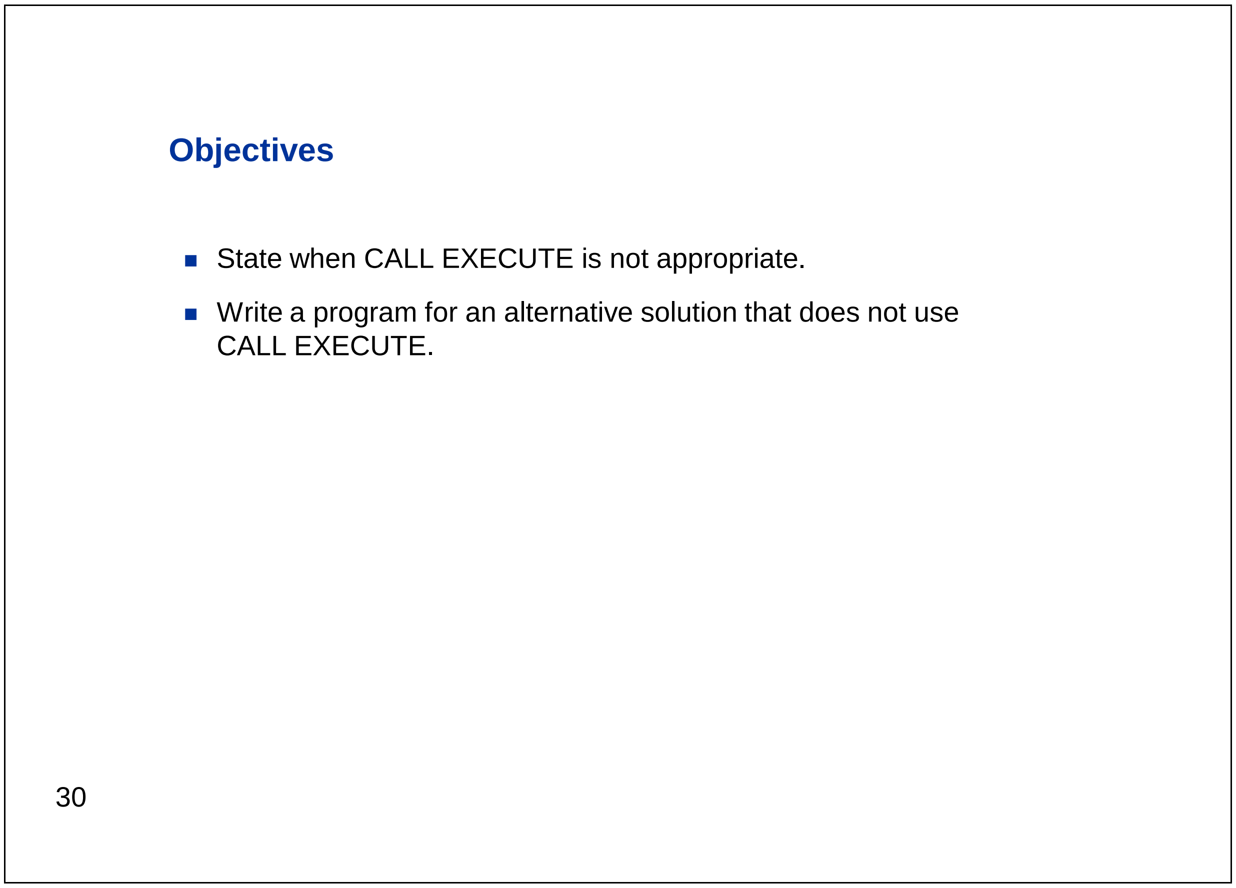
5. Inappropriate Uses of CALL EXECUTE


The Power of CALL EXECUTE

1. The Basics of CALL EXECUTE
2. Generating Code Based on Data Set Values
3. Repetitive Operations
4. Passing Variable Values to a Macro Call
5. Inappropriate Uses of CALL EXECUTE

29

The fifth and final section of this e-lecture is titled “Inappropriate Uses of CALL EXECUTE.”

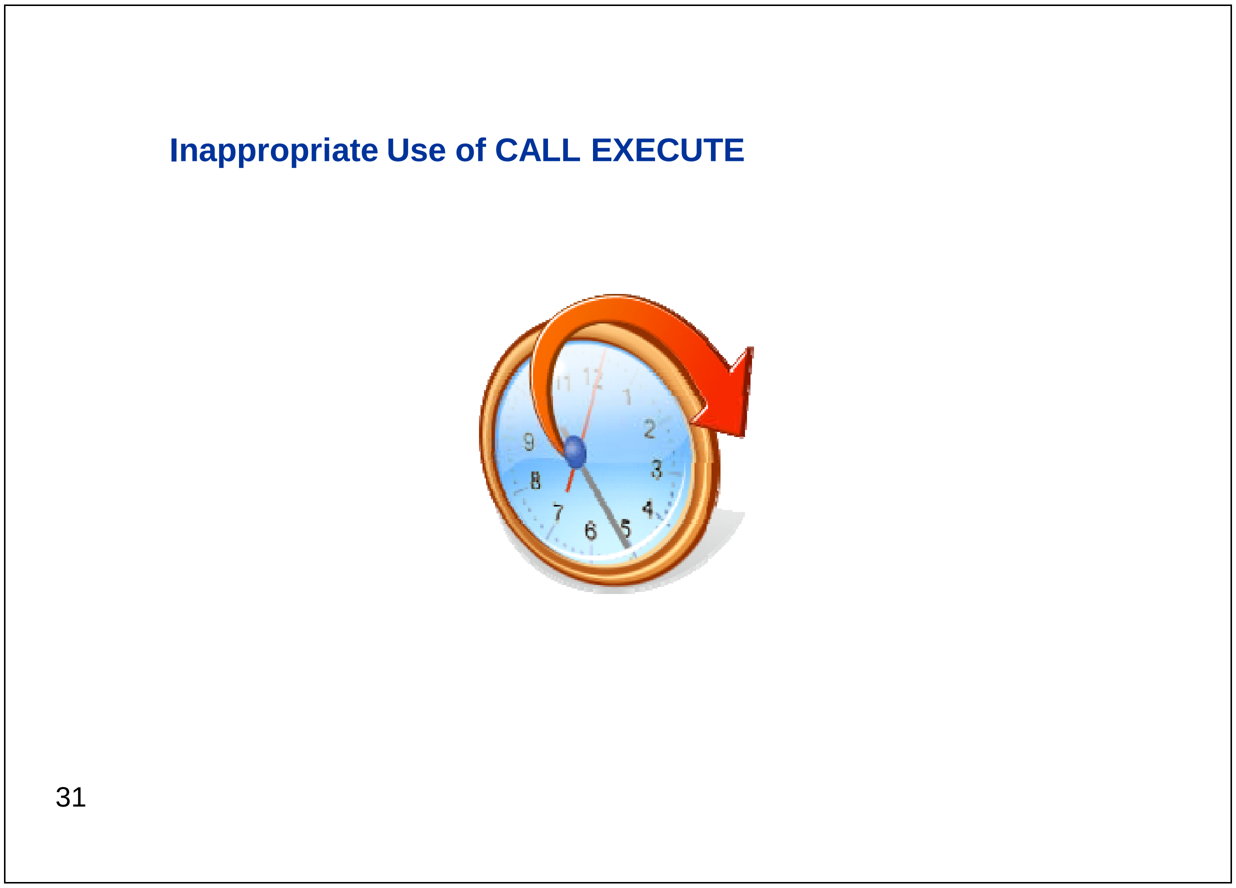


Objectives

- State when CALL EXECUTE is not appropriate.
- Write a program for an alternative solution that does not use CALL EXECUTE.

30

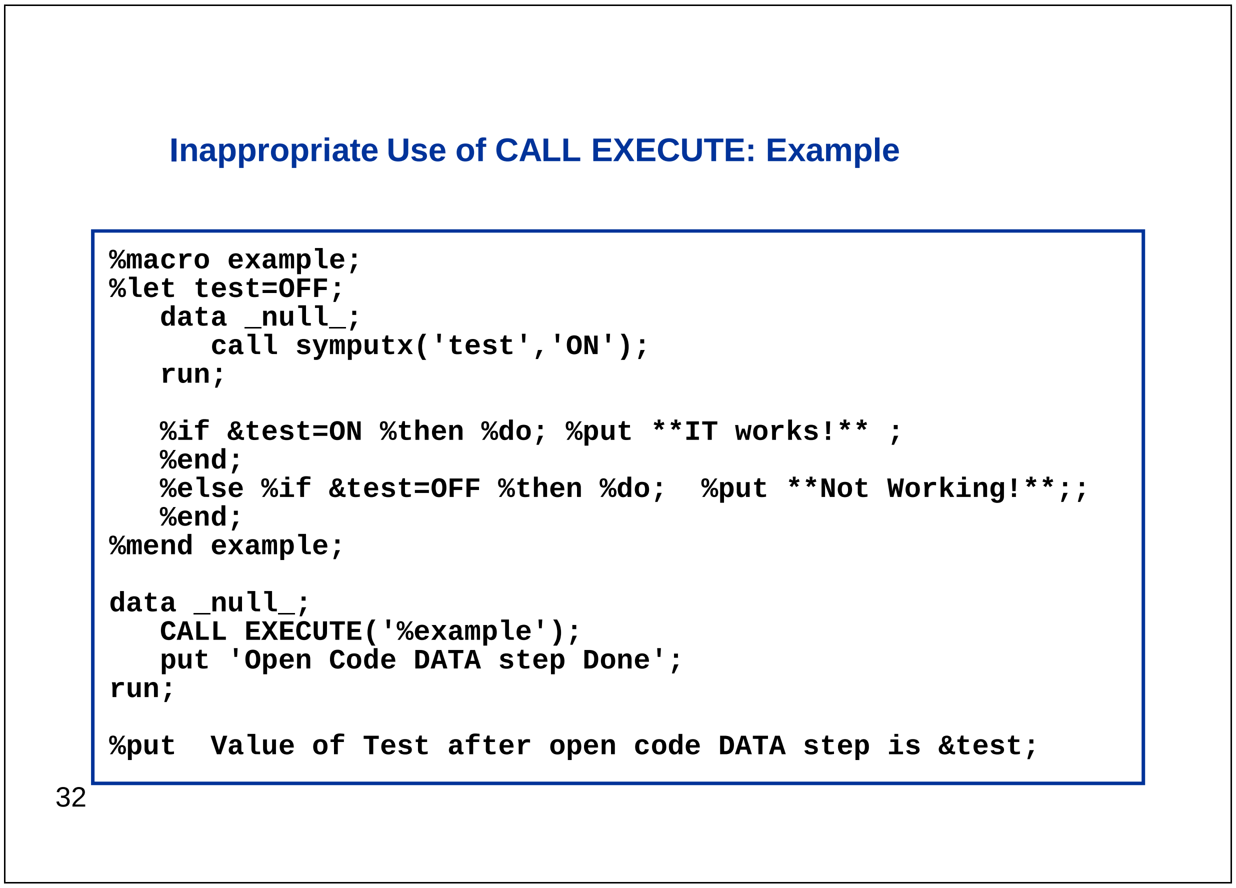
In this section, we will discuss when CALL EXECUTE is inappropriate and show an alternative solution to get the results we need.



Inappropriate Use of CALL EXECUTE

31

Remember when Marty McFly went back in time in the film *Back to the Future*? Well, he couldn't very well have called anyone in the future because it hadn't happened yet. He had to wait to get back to the future to do that. Trying to use a CALL SYMPUT statement inside a macro definition that's called through CALL EXECUTE is sort of like that. It doesn't work until you get back – or until the original DATA step finishes executing.

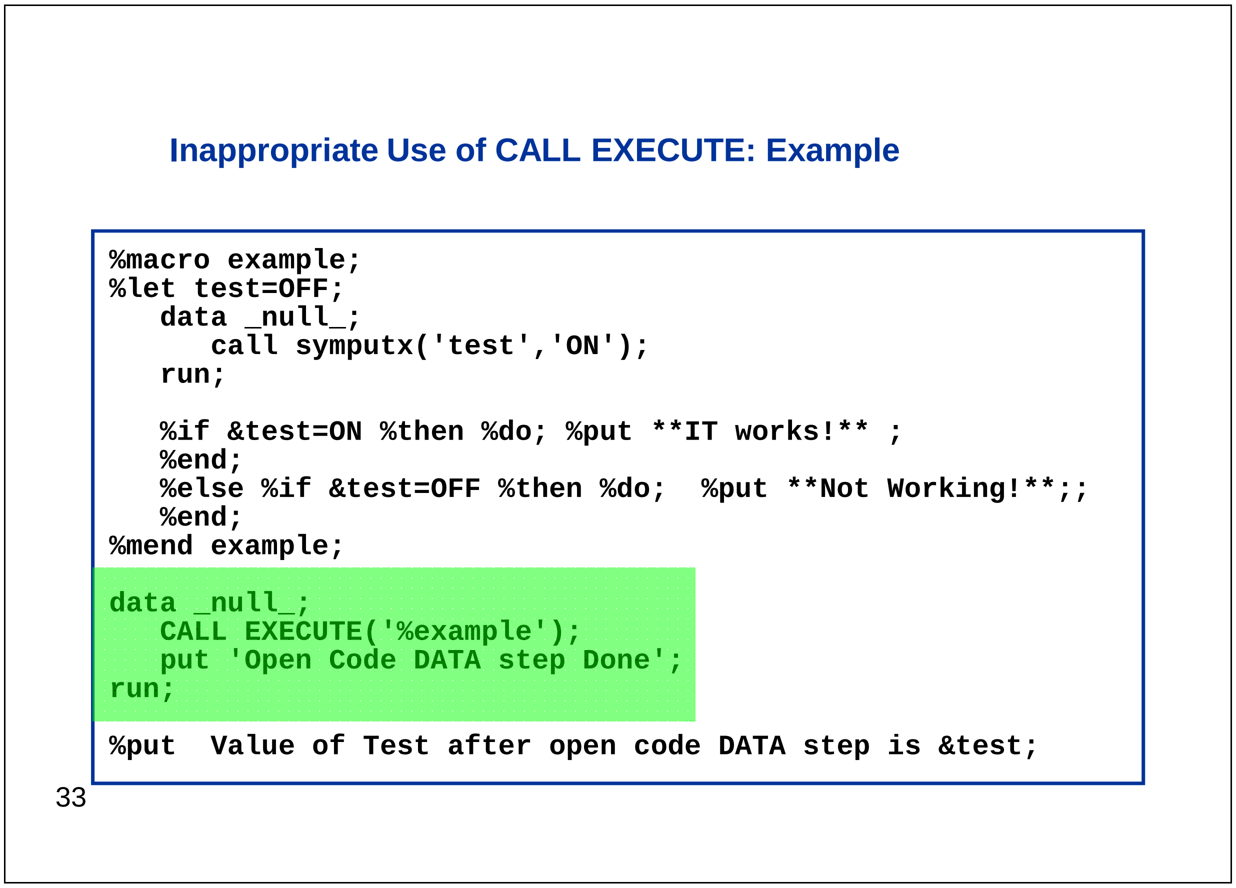


Inappropriate Use of CALL EXECUTE: Example

```
%macro example;
%let test=OFF;
   data _null_;
      call symputx('test','ON');
   run;

   %if &test=ON %then %do; %put **IT works!** ;
   %end;
   %else %if &test=OFF %then %do;  %put **Not Working!**;;
   %end;
%mend example;

data _null_;
   CALL EXECUTE('%example');
   put 'Open Code DATA step Done';
run;

%put  Value of Test after open code DATA step is &test;
```

32

Most syntax generated by CALL EXECUTE will be submitted **after** the DATA step from which the syntax was generated finishes completion, with the exception of macro statements, which are executed immediately. This normally isn't an issue as we've seen in some previous examples. However, if you're using CALL EXECUTE to generate macro calls to macro definitions containing CALL SYMPUTX statements, it **will** be a problem. This is because the CALL SYMPUTX statements won't get executed until **after** the original DATA step, …

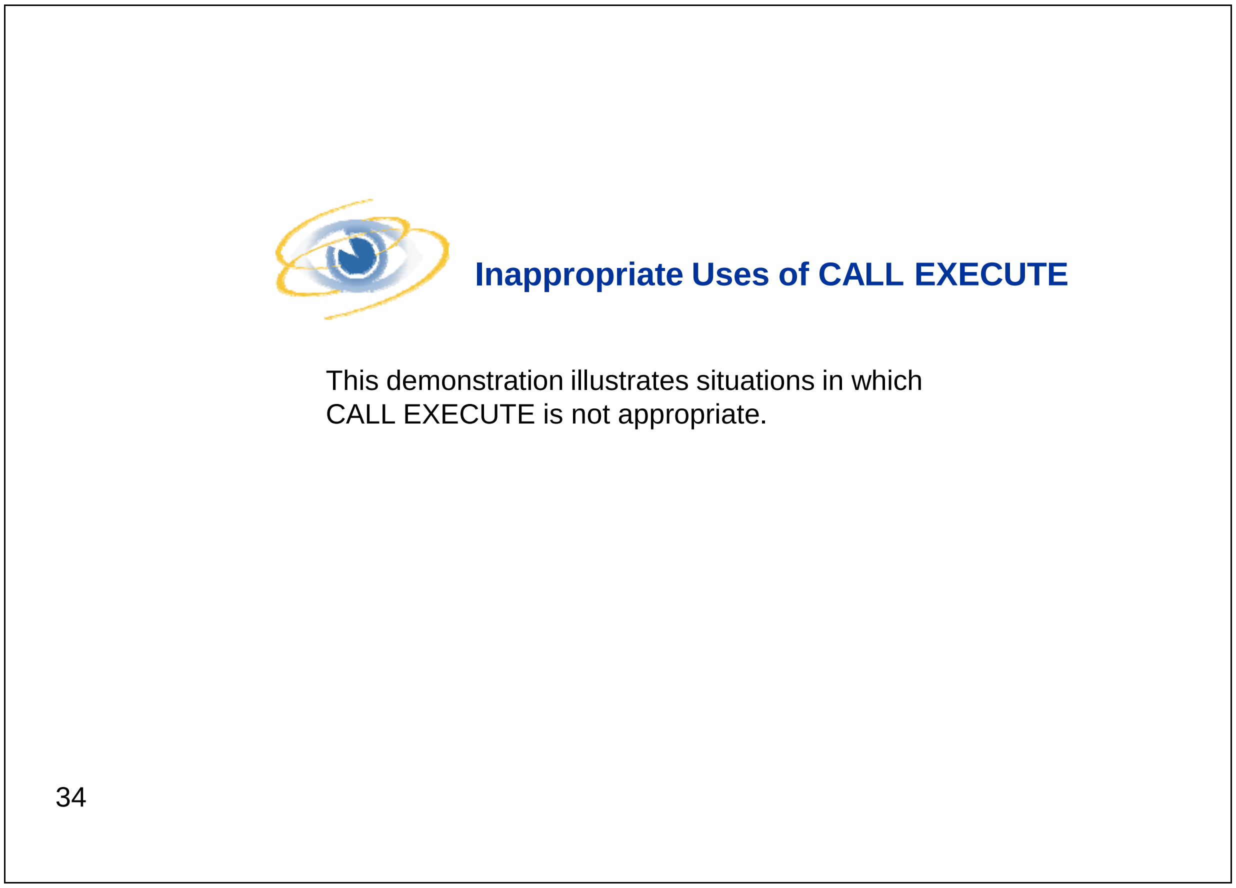


Inappropriate Use of CALL EXECUTE: Example

```
%macro example;
%let test=OFF;
   data _null_;
      call symputx('test','ON');
   run;

   %if &test=ON %then %do; %put **IT works!** ;
   %end;
   %else %if &test=OFF %then %do;  %put **Not Working!**;;
   %end;
%mend example;

data _null_;
   CALL EXECUTE('%example');
   put 'Open Code DATA step Done';
run;

%put  Value of Test after open code DATA step is &test;
```

33

… the one that contains the CALL EXECUTE statement, finishes. This means that the macro executes before the SYMPUT routine has stored the macro variable value. So if part of your macro definition relies on the value of your CALL SYMPUTX, it won't work correctly. As we can see in the basic example shown here in the slide, the DATA step in open code uses a CALL EXECUTE to invoke the example macro. The example macro has another DATA step with a CALL SYMPUTX statement. Let's jump into SAS and run this and look at the log. We'll also look at a more practical example while in SAS.



Inappropriate Uses of CALL EXECUTE

This demonstration illustrates situations in which CALL EXECUTE is not appropriate.

34

This demonstration illustrates situations in which CALL EXECUTE is not appropriate.

1. Let’s run the code we just saw on the slide and look at the log. We can see we get the value “**Not Working!**” because the test macro doesn’t get assigned the value of ON until the open code DATA step finishes executing. Notice the SYMBOLGEN results here as well. When the conditions are checked, the test macro still is OFF. When the DATA step that generated the CALL EXECUTE finished, the test macro is ON. Also, notice in the log that we don’t get notes about the completion of the DATA step inside the macro definition until the very end.

   Let’s take a look now at a more practical example.

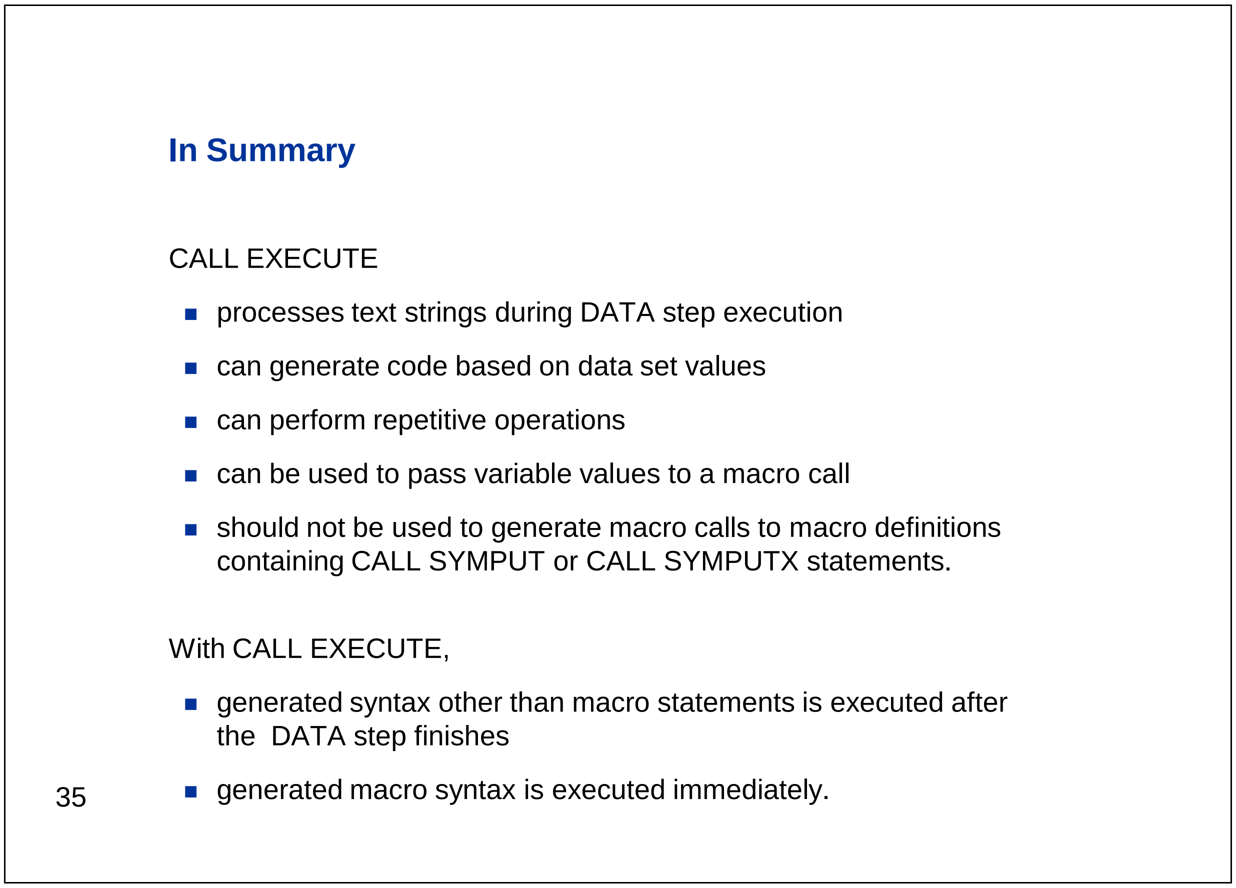
2. The setup is the same as in the previous examples:
   a. Create a simple macro variable named LIST to store a selection of customers and then create a customer ID-sorted subset of orders placed by these customers.
   b. The open code DATA step at the end of this program again uses CALL EXECUTE to generate a macro call to the PROBLEM_ORDERS macro.
   c. Notice this macro program contains a DATA step with a CALL SYMPUTX, which is supposed to create a macro variable named CountryName that contains the full name of the country for the customer ID passed to the macro. I also added the $COUNTRY user format we created earlier to this program. Make sure to run the code from the first demo before attempting this one.

d. The PROC PRINT in this macro definition should only be executed for customers from the United States.

When we run this, we'll see we won't get any PROC PRINTs. Remember, the CALL SYMPUTX doesn't get submitted until after the DATA step that's generating the CALL EXECUTES finishes. Let's take a look at the log. Among other errors, we see a bunch of warnings that &CountryName doesn't have a value. Remember this is because at the time the macro processor saw the macro references it hadn't yet executed the CALL SYMPUTXs. Notice all the DATA step results at the end of the log.

Now many users at this point might try running this program a second time. If you do this – which we will do now – it looks like it might have worked. Notice the lack of red errors in the log. This is because the &CountryName macro variable now has a value; its value is wherever the last customer processed was from – Israel in this case, as we can see if we run the following %PUT and PROC PRINT.

3. Here's one way to fix this problem. Notice we've taken out all the CALL EXECUTEs and instead used a few CALL SYMPUTXs to build a series of macros variables representing the needed information for our select customers. We then call the PROBLEM_ORDERS macro (that includes the same CALL SYMPUTX from before) using a macro loop inside the driver macro. The main point here is: Don't use CALL EXECUTE to build macro calls to macros that include CALL SYMPUTXs!



In Summary

CALL EXECUTE

- processes text strings during DATA step execution
- can generate code based on data set values
- can perform repetitive operations
- can be used to pass variable values to a macro call
- should not be used to generate macro calls to macro definitions containing CALL SYMPUT or CALL SYMPUTX statements.

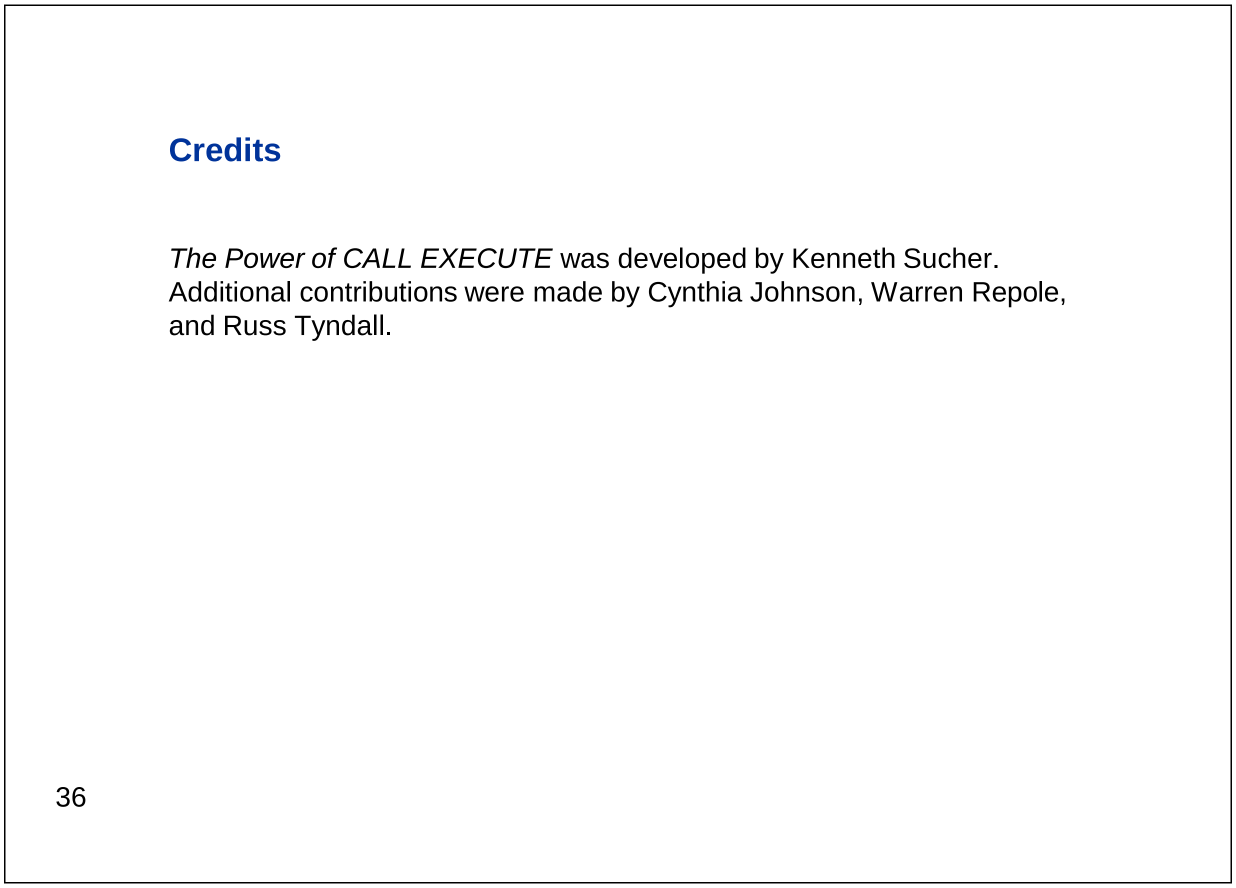
With CALL EXECUTE,

- generated syntax other than macro statements is executed after the DATA step finishes
- generated macro syntax is executed immediately.

35

In summary, CALL EXECUTE processes text strings during DATA step execution. It can generate code based on data set variable values and perform repetitive operations. It can also be used to pass variable values to a macro call. Syntax generated by CALL EXECUTE that does not contain macro triggers will be submitted after the DATA step finishes processing, whereas macro triggers are resolved immediately. CALL EXECUTE should not be used to generate macro calls to macro definitions containing CALL SYMPUT or CALL SYMPUTX statements.

So, you now know what CALL EXECUTE is. More specifically, you've learned when and how to use the routine. In addition, you know when the code that CALL EXECUTE generates is executed: macro syntax is immediate, and everything else is submitted after the DATA step completes. You've also seen some reasons why to use it, as well as some situations where CALL EXECUTE isn't appropriate.

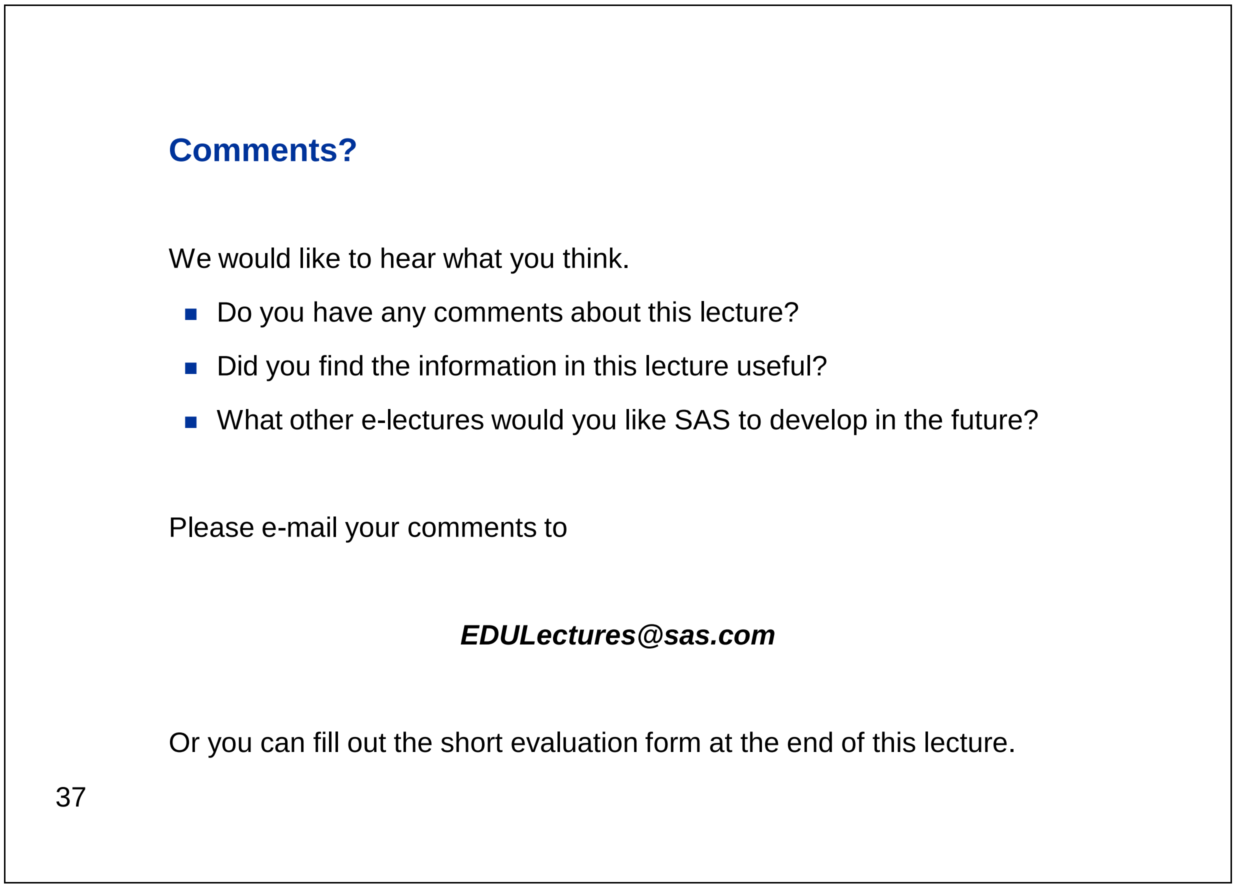


Credits

The Power of CALL EXECUTE was developed by Kenneth Sucher. Additional contributions were made by Cynthia Johnson, Warren Repole, and Russ Tyndall.

36

This concludes the SAS e-Lecture *The Power of CALL EXECUTE*. I hope you have found the material in this lecture to be helpful to your work tasks.



Comments?

We would like to hear what you think.

- Do you have any comments about this lecture?
- Did you find the information in this lecture useful?
- What other e-lectures would you like SAS to develop in the future?

Please e-mail your comments to

EDULectures@sas.com

Or you can fill out the short evaluation form at the end of this lecture.

37

If you have any comments about this lecture or e-lectures in general, we would appreciate receiving your input. You can use the e-mail address listed here to provide that feedback or you can fill out the short evaluation coming up at the end of the lecture.

Copyright

38

Thank you for your time.

Appendix A Demonstration Programs

1. **Generating Data-Dependent Code A-3**

2. **Repetitive Operations A-4**

3. **Using CALL EXECUTE to Create Macro Calls A-5**

4. **Inappropriate Uses of CALL EXECUTE A-6**

1. Generating Data-Dependent Code

Section 2, Slide 17

```
libname orion '.';

data _null_;
   set orion.country end=done;
   if _n_ = 1 then CALL EXECUTE('proc format;
      value $Country');
   value=put(Country,$quote4.);
   label=put(Country_Name,$quote30.);
   CALL EXECUTE(value || '=' || label);
   if done then CALL EXECUTE ('; run;');
run;
```

2. Repetitive Operations

Section 3, Slide 24

```
libname orion '.';

%let list=23,45,61,75,14703;

proc sort data=orion.order_fact
          out=order_fact (keep=Customer_ID Product_ID Quantity
                          Total_Retail_Price);
   by customer_id;
   where Customer_ID in (&list);
run;

data _null_;
   set order_fact;
   by customer_id;
   if first.customer_id then do;
       order_count=0;
       order_total=0;
   end;
   order_count+1;
   order_total+total_retail_price;
   if last.customer_id then do;
      CALL EXECUTE('proc print data=order_fact noobs;');
      CALL EXECUTE(cats('where customer_id=',customer_id,';'));
      CALL EXECUTE(catx(' ','title "',order_count,
           'Order(s) from customer',
           Customer_ID,'totaling: ',put(order_total,dollar9.2),'";'));
      CALL EXECUTE('run;');
   end;
run;
```

3. Using CALL EXECUTE to Create Macro Calls

Section 4, Slide 28

```
libname ORION '.';
%let list=23,45,61,75,14703;

proc sort data=orion.order_fact
          out=order_fact (keep=Customer_ID Product_ID Quantity
                          Total_Retail_Price);
   by customer_id;
   where Customer_ID in (&list);
run;

%macro problem_orders(id,count,total);
   proc print data=order_fact;
      where customer_id=&id;
      title "&count Order(s) from customer &ID totaling: &total";
   run;
%mend;

data _null_;
   set order_fact;
   by customer_id;
   if first.customer_id then do;
       order_count=0;
       order_total=0;
   end;
   order_count+1;
   order_total+total_retail_price;
   if last.customer_id then do;
    CALL EXECUTE(cats('%problem_orders(',customer_id,',',order_count,
                      ',',put(order_total,dollar9.2),')'));
   end;
run;

data _null_;
   set order_fact;
   by customer_id;
   if first.customer_id then do;
       order_count=0;
       order_total=0;
   end;
   order_count+1;
   order_total+total_retail_price;
   if last.customer_id then do;
    CALL EXECUTE(cats('%problem_orders(',customer_id,',',order_count,
                      ',',' %str(',put(order_total,dollar9.2),'))'));
   end;
run;
```

4. Inappropriate Uses of CALL EXECUTE

Section 5, Slide 34

Demonstration Code 1

```
options symbolgen mlogic;

%macro example;
%let test=OFF;
   data _null_;
      call symputx('test','ON');
   run;

   %if &test=ON %then %do; %put **IT works!** ;
   %end;
   %else %if &test=OFF %then %do; %put **Not Working!**;
   %end;
%mend example;

data _null_;
   CALL EXECUTE('%example');
run;
```

Demonstration Code 2

Incorrect Code

```
libname ORION '.';
options nosymbolgen nomlogic;
%let list=23,45,61,75,14703;

proc sort data=orion.order_fact
          out=order_fact (keep=Customer_ID Product_ID Quantity
                          Total_Retail_Price);
   by customer_id;
   where Customer_ID in (&list);
run;

%macro problem_orders(id,count,total);

   data _null_;
      set orion.customer_dim (where=(Customer_ID=&id));
       call symputx('CountryName',put(Customer_Country,$Country.));
   run;
```

(Continued on next page.)

```
%if &CountryName=United States %then %do;

   proc print data=order_fact;
      where customer_id=&id;
      title "&count Order(s) from customer &ID totaling: &total";
        title2 "Customer is from &CountryName";
  run;

%end;

%mend;

data _null_;
   set order_fact;
   by customer_id;
   if first.customer_id then do;
       order_count=0;
       order_total=0;
   end;
   order_count+1;
   order_total+total_retail_price;
   if last.customer_id then do;
    CALL EXECUTE('%problem_orders('||customer_id||','||order_count||
                ', %str('||put(order_total,dollar9.2)||'))' );
   end;
run;
```

To Check Results:

```
%put CountryName=&CountryName;

proc print data=orion.customer_dim noobs;
   title;
   var Customer_ID Customer_Country;
   where Customer_Id in (&list);
   format Customer_Country $Country.;
run;
```

A Fix:

```
%macro problem_orders(id,count,total);

  data _null_;
     set orion.customer_dim (where=(Customer_ID=&id));
      call symputx('CountryName',put(Customer_Country,$Country.));
  run;

%if &CountryName=United States %then %do;

  proc print data=order_fact;
```

```
     where customer_id=&id;
     title "&count Order(s) from customer &ID totaling: &total";
      title2 "Customer is from &CountryName";
  run;

%end;

%mend;


data _null_;
   set order_fact end=last;
   by customer_id;
   if first.customer_id then do;
       order_count=0;
       order_total=0;
   end;
   order_count+1;
   order_total+total_retail_price;
   if last.customer_id then do;
      counter+1;
      call symputx('ID'||left(counter),customer_id);
      call symputx('Count'||left(counter),order_count);
     call symputx('Total'||left(counter),put(order_total,dollar9.2));
     if last then call symputx('n',counter);
   end;
run;

%macro driver;
   %do i=1 %to &n;
   %problem_orders(&&ID&i,&&Count&i,%bquote(&&Total&i))
   %end;
%mend;

%driver
```