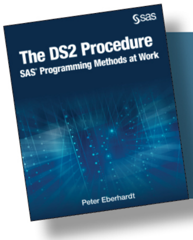
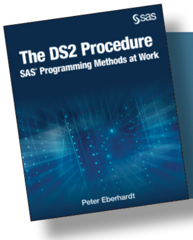
The DS2 Procedure

SAS® Programming Methods at Work

Peter Eberhardt


sas
The DS2 Procedure
SAS Programming Methods at Work
Peter Eberhardt

Contents



Chapter 1: Solving Bigger Problems

big data. Big data. BIG DATA.

It seems that not a day goes by that we do not hear a familiar chant; even the most techno-Luddites chant it—"big data. Big data. BIG DATA." Although there is no doubt that the volumes of data are growing, big data is the smaller of our problems. Yes, data are big, but *how we handle* that big data is an even bigger problem. If the problems that we have today were the same as the ones that we had 10 or even five years ago, our bigger and better hardware could easily handle them.

Today, we have far more complex problems. Today, the mega-retailer is no longer happy with data about the profitability of a product by store. It wants to know who is buying what, when and where are they are buying it, in what combinations are they buying it, and what can be offered at check-out to increase the basket value. This is a complex problem, and bigger and better hardware does not solve it. The complex and mercurial nature of today's problems means that we have to develop complex yet flexible solutions. How can we, as SAS developers, develop more complex and flexible solutions? One way is to use PROC DS2.

PROC DS2

The DATA step has served SAS programmers well over the years. Although it is powerful, it has not fundamentally changed since its inception. SAS has introduced a significant programming alternative to the DATA step—PROC DS2—a new procedure for your object-oriented programming environment. PROC DS2 is basically a new programming language based on the DATA step language. It is a powerful tool for advanced problem solving and advanced data manipulation. PROC DS2 makes it easier to develop complex and flexible programs for complex and flexible solutions. These programs are robust and easier to understand, which eases maintenance down the road.

Starting with SAS 9.4, PROC DS2 is part of the Base SAS package. For users in a high-performance analytics environment, there is PROC HPDS2. However, in this book, only PROC DS2 is discussed.

Problem Space

PROC DS2 deals with this more complex problem space by using many object-oriented programming (OOP) constructs. With OOP constructs, SAS programmers can develop more robust and flexible programs using the following:

- clarity
- scope
- modularity and encapsulation
- data types

Clarity

In DS2, you must be clear with each identifier that you are using. An identifier is one or more tokens or symbols that name programming language entities such as variables, labels, method names, package names, and arrays, as well as data source objects such as table names and column names. To ensure clarity, in DS2, identifiers are declared using a DECLARE statement. The DECLARE statement clearly states both the name and data type of the identifier. Before you can use an element in a DS2 program, you must tell DS2 the name and data type of the element. The benefit (besides making the programmer think more clearly about the nature of the program!) is that because the program does not compile if an invalid identifier is used, misspellings and other hard-to-detect errors can be addressed and corrected at the beginning.

Scope

In programming, scope is the area in which a variable is visible. In other words, scope lets you know where a variable can be accessed. In DS2, there are two levels of scope:

- global
- local

Global variables have global scope. That is, they are accessible from anywhere in the program. Local variables have local scope. That is, they are accessible only from within the block in which the variable was declared and only while that block is executing. Each variable in any scope must have a unique name, but variables in different scopes can have the same name. This enables you to use consistent and meaningful variable names in different parts (or methods) of your program without overwriting values. The benefit is that you can more easily isolate worker variables (e.g., a DO loop variable, an intermediate calculation, etc.) from variables that will ultimately be written out to result sets.

Modularity and Encapsulation

A programming block is a section of a DS2 program that encapsulates variables and code. Programming blocks enable modularity and encapsulation by using modular and reusable code to perform specific tasks. This, in turn, can lead to shorter development time and the standardization of often-repeated or business-specific programming tasks. Layered programming blocks enable advanced encapsulation and abstraction of behavior, which enhances the readability and understandability of a program.

In addition, a programming block defines the scope of identifiers within that block. An identifier declared in the outermost programming block has global scope. An identifier declared in a nested block has local scope.

Table 1.1[1] lists some of the most common programming blocks, adapted from the *SAS 9.4 DS2 Language Reference Manual*.

Table 1.1: Common Programming Blocks

Block	Delimiters	Notes
Procedure	PROC DS2…QUIT	
Data program	DATA…ENDDATA	Variables that are declared at the top of a data program have global scope within the data program. In addition, variables that the SET statement references have global scope. Unless you explicitly drop them, global variables in the data program are included in the program data vector (PDV). **Note:** Global variables exist for the duration of the data program.
Method	METHOD…END	A method is a sub-block of a data program, package, or thread program. Method names have global scope within the enclosing programming block. Methods contain all of the executable code. PROC DS2 has three system-defined methods: INIT(), RUN(), and TERM(). Variables that are declared at the top of a method have local scope. Local variables in the method are not included in the PDV. **Note:** Local variables exist for the duration of the method call.
Package	PACKAGE…ENDPACKAGE	Variables that are declared at the top of a package have global scope within the package. Package

Block	Delimiters	Notes
		variables are not included in the PDV of a data program that is using an instance of the package. **Note:** Package variables exist for the duration of the package instance.
Thread	THREAD…ENDTHREAD	Variables that are declared at the top of a thread have global scope within the thread program. In addition, variables that the SET statement references have global scope. Unless you explicitly drop them, global variables in the thread program are included in the thread output set. **Note:** Thread variables exist for the duration of the thread program instance. They can be passed to the data program using the SET FROM statement.

Data Types

Unlike the DATA step, which has two data types—numeric (double-precision floating-point) and fixed-length character—DS2 has many data types. This allows DS2 programs to interact better with external databases.

Data Space

No surprise here, you have to deal with a big data space. DS2 helps you by providing three major features:

- embedded SQL
- threaded data access
- in-database processing

Embedded SQL

DS2 can access data through a SET statement just like the DATA step. In addition, data can be accessed through embedded SQL statements.

Threaded Data Access

DS2 can access data through a SET statement or through embedded SQL statements. DS2 also has threaded access to the data. The effectiveness of threaded access is determined, to a large extent, by how the back-end database manages threads.

In-Database Processing

If your data is in one of the supported databases, DS2 can process inside the database. This topic is not covered in this book.

Our First DS2 Programs

It seem de rigueur to start all programming language tutorials with a “Hello World” example. Because SAS developers are focused on real world problems and getting accurate results, let’s fast-forward and say “hello” to some simple data conversions.

PROC DS2 as a DATA Step Equivalent

Before you really take advantage of DS2, let’s look at a simple DATA step that creates a table, and then let’s look at the equivalent in DS2. The example data and program creates a SAS data set with data points representing temperatures in degrees Celsius. The following DATA step creates a SAS data set named dsDegC and uses parameters defined in the macro variables. One thousand observations (&NObs) are generated between -40 (&min) and 40 (&max). To verify that the DATA step and DS2 both create the same data, the seed value (&seed) is set to be passed into a random number generator.

Parameters

```
%let NObs = 1000;
%let min  = -40;
%let max  =  40;
%let seed = 123456;
```

DATA Step

```
data dsDegC (keep=degC)
     dsAvgC (keep=avgC)
     ;
       label degC = 'Temp in Celsius';
       label avgC = 'Average Temp in Celsius';
       format degC F3.;
       format avgC F5.2;
       call streaminit(&seed);
       Min = &min; Max = &max;
       sum = 0;
       do obs = 1 to &NObs;
          u = rand("Uniform");               /* U[0,1] */
          degC = min + floor((1+Max-Min)*u); /* uniform integer in Min..Max */
          output dsDegC;
          sum = sum + degC;
       end;
       avgC = sum / (obs-1);
       output dsAvgC;
run;
```

DS2

```
proc DS2 scond=error; ❶
data ds2DegC_1 (keep=(degC) overwrite=YES) ❷
     ds2AvgC_1 (keep=(avgC) overwrite=YES)
     ;
   declare integer degC having label 'Temp in Celsius' format F3.;  ❸
   declare double avgC having label 'Average Temp in Celsius' format F5.2;
   method run();  ❹
        declare int min max obs;  ❺
        declare double u sum;
        streaminit(&seed);
        Min = &min; Max = &max;
        sum = 0;
        do obs = 1 to &NObs;
           u = rand('UNIFORM');
          degC = min + floor((1+Max-Min)*u); /* uniform integer in Min..Max */
           output ds2DegC_1;
           sum = sum + degC;
        end;
        avgC = sum / (obs-1);
        output ds2AvgC_1;
   end;
enddata;
run;
quit;
```

The heart of the program, with the exception of the output data set name, is the same in both the DATA step and DS2.

```
do obs = 1 to &NObs;
   u = rand("Uniform");                  /* U[0,1] */
   degC = min + floor((1+Max-Min)*u); /* uniform integer in Min..Max
*/
   output dsDegC;
   sum = sum + degC;
end;
```

However, the DS2 program appears to be more complex, requiring more statements to get to the heart of the program.

❶ DS2 is a new procedure in SAS 9.4 terminated by the QUIT statement. The `scond=error` option specifies that any undeclared identifiers should cause an error. There is also a new SAS option called DS2COND that can be set to ERROR. A best practice is to set DS2COND=ERROR in the configuration file so that it is always set.

❷ Unlike the DATA step, DS2 does not automatically overwrite existing tables. The `overwrite=YES` data set option tells DS2 to drop the data set if it exists before creating it. This is standard in SQL.

❸ All identifiers must be declared with a name and data type. The label and format are optional. The variables `degC` and `avgC` are declared outside of the method so they are global in scope. Only global variables can be written to the output tables.

❹ All executable code must reside in a method. `method run()` is one of the system-defined DS2 methods.

❺ `min`, `max`, and `obs` are integer variables. Because they are declared inside `method run()`, they are local in scope. Local variables are not written to the output tables.

The original DATA step has three distinct phases:

The first phase is initialization (setting the starting values):

```
call streaminit(&seed);
Min = &min; Max = &max;
sum = 0;
```

The second phase is processing (executing the DO loop):

```
do obs = 1 to &NObs;
   u = rand("Uniform");
   degC = min + floor((1+Max-Min)*u);
   output dsDegC;
   sum = sum + degC;
end;
```

The third phase is termination (calculating the average):

```
avgC = sum / (obs-1);
output dsAvgC;
```

In this simple DATA step, it is easy to enforce the one-time nature of the initialization and termination phases of the program. However, in many DATA steps, you must add programming logic to enforce these phases. DS2 simplifies and clarifies these phases.

Initialization, Processing, and Termination

DS2 simplifies and clarifies the three phases (initialization, processing, and termination) using three system-defined methods INIT(), RUN(), and TERM(). The first refinement of the DS2 program demonstrates this:

```
proc DS2 scond=error;
data ds2DegC_2 (keep=(degC) overwrite=YES)
     ds2AvgC_2 (keep=(avgC) overwrite=YES)
     ;
declare integer degC having label 'Temp in Celsius' format F3.;
declare double avgC having label 'Average Temp in Celsius' format F5.2;
declare int min max NObs;  ❶
declare double sum;
retain  sum nobs;
```

```
method init();    ❷
    streaminit(&seed);
    Min = &min; Max = &max;
    nobs = &NObs;
    sum = 0;
end;

method run();
    declare double u;
    declare int obs;
    do obs = 1 to NObs;
       u = rand('UNIFORM');
       degC = min + floor((1+Max-Min)*u);
       output ds2DegC_2;
       sum = sum + degC;
    end;
end;

method term();    ❸
    avgC = sum / nobs;
    output ds2AvgC_2;
end;

enddata;
run;
quit;
```

❶ More variables now have global scope. They are no longer just inside a method and have only local scope. All three methods use global variables.

❷ `method init()` is a system-defined method. It is automatically called at the start of the program. This replaces the `if _n_ = 1` block that is common in many DATA steps. This method can be used to initialize variables and invoke processing.

❸ `method term()` is a system-defined method. It is automatically called after `method run()` completes. It can be used to perform any wrap-up processing (in this case, calculating the average).

User-Defined Method

DS2 enables you to create your own methods to encapsulate logic. In the DS2 program, there is a formula (`min + floor((1+Max-Min)*u)`) that is used in more than one place. You can simply repeat the calculation. Or, even better, you can encapsulate the logic in a method. In this way, if you want to change the formula, you change it only once, as seen in the following example:

```
proc DS2 scond=error;
data ds2DegC_3 (keep=(degC) overwrite=YES)
     ds2AvgC_3 (keep=(avgC) overwrite=YES)
     ;
```

```
declare integer degC having label 'Temp in Celsius' format F3.;
declare double avgC having label 'Average Temp in Celsius' format
F5.2;
declare integer min max NObs;
declare double sum;
retain  sum nobs;

method getRange(integer min, integer max, double u) returns integer; ❶
       return(min + floor((1+Max-Min)*u)); ❷
end;

method init();
    streaminit(&seed);
    Min = &min; Max = &max;
    nobs = &NObs;
    sum = 0;
end;

method run();
    declare double u;
    declare int obs;
    do obs = 1 to nobs;
       u = rand('UNIFORM');
       degC = getRange(min, max, u);  ❸
       output ds2DegC_3;
       sum = sum + degC;
    end;
end;

method term();
    avgC = sum / nobs;
    output ds2AvgC_3;
end;

enddata;
run;
quit;
```

❶ `getRange` takes three positional arguments—two integers (`min` and `max`) and `double u`. It returns an integer value.

❷ The `return` statement sends the `getRange` method's result to the caller. The formula is embedded in the `return` statement.

❸ The `getRange` method is invoked to calculate the `degC` value rather than using the formula directly.

Packages Make Methods Reusable

In the previous example, you saw how a method can be defined to replace a formula or algorithm that occurs in many places in a program. You can also define a method that can be invoked in many DS2 programs—this is called a package. In its simplest form, a package is a collection of related methods that is saved to a table that can be accessed by other DS2 programs.

```
proc DS2 scond=error;
package range /overwrite=YES;   ❶
   method getRange(integer min, integer max, double u) returns
integer;
          return(min + floor((1+Max-Min)*u));
   end;
endpackage;
run;
quit;

proc DS2 scond=error;  ❷
data ds2DegC_4 (keep=(degC) overwrite=YES)
     ds2AvgC_4 (keep=(avgC) overwrite=YES)
     ;
declare integer degC  having label 'Temp in Celsius' format F3.;
declare double avgC having label 'Average Temp in Celsius' format
F5.2;
declare integer min max nobs;
declare double sum;
retain sum nobs;

declare package range range();   ❸

method init();
    streaminit(&seed);
    Min = &min; Max = &max;
    nobs = &NObs;
    sum = 0;
end;

method run();
    declare double u;
    declare int obs;
    do obs = 1 to nobs;
       u = rand('UNIFORM');
       degC = range.getRange(min, max, u);   ❹
       output ds2DegC_4;
       sum = sum + degC;
    end;
end;

method term();
    avgC = sum / nobs;
    output ds2AvgC_4;
end;

enddata;
run;
quit;
```

❶ A package is a collection of methods. Typically, the methods are logically related (for example, all of the methods are used to calculate a range of values). The package is saved to a table so that it can be used by other DS2 programs. In this example, the package is saved in the Work library. Once a package is tested and debugged, it is saved to a permanent library.

❷ PROC DS2 is invoked a second time to demonstrate the use of packages defined outside the PROC.

❸ All identifiers in a DS2 program need to be declared. In this line, an entity (variable) called `range` is declared. The `range` variable initiates an instance of a `range` package that was defined in a previous DS2 program. Although the variable `range` and the package `range` have the same name, it is not required.

❹ The `getRange()` method is called. It is in the `range` package referenced by the `range` variable.

The previous examples demonstrate clarity, specifically because they separate processing steps into different methods—init(), term(), and getRange(). Furthermore, encapsulation is used; first, computational formulas are moved into methods. Second, methods are moved into a package that can be accessed by other DS2 programs.

Accessing Data—SET statement

In the following example, the table that was created in the previous example is read and a new data set is created. Temperatures are in degrees Fahrenheit.

```
proc DS2 scond=error;
package conv /overwrite=yes;    ❶

 method C_to_F(integer C) returns double;
  /* convert degrees fahrenheit to degrees celsius */
  return 32. + (C * (9. / 5.));
 end;

 method F_to_C(double F) returns double;
  /* convert degrees fahrenheit to degrees celsius */
  return (F - 32.) * (5. / 9.);
 end;

endpackage;
run;
quit;

proc DS2 scond=error;
data ds2DegF_5 (keep=(degF) overwrite=YES)
     ds2AvgF_5 (keep=(avgF) overwrite=YES)
     ;
declare double degF having label 'Temp in Fahrenheit' ' format F6.1;
declare double avgF having label 'Avg Temp in Fahrenheit' format F6.1;
declare double sum;
declare integer cnt;
declare package conv cnv();    ❷
retain sum cnt;
```

```
method init();
    sum = 0;
    cnt = 0;
end;

method run();
    set ds2DegC_1;    ❸
    degF = cnv.C_to_F(degC);  ❹
    sum = sum + degF;
    cnt = cnt + 1;
    output ds2DegF_5;
end;

method term();
    avgF = sum / cnt;
    output ds2AvgF_5;
end;

enddata;
run;
quit;
```

❶ A new package is created with temperature-conversion methods.

❷ A new instance of the package is created and called `cnv`.

❸ The table created in the previous example is read. The `run()` method iterates over all of the rows in the table.

❹ The C_to_F() method is invoked.

Accessing Data—Threads

The last enhancement to this example shows how processing goes from sequential using the SET statement to concurrent using threads. You can use threaded processing on a single machine with multiple cores or parallel processing on back-end databases.

```
proc ds2;
thread temps /overwrite=yes;  ❶
   method run();  ❷
       set ds2DegC_1;
   end;

endthread;
run;
quit;

proc DS2 scond=error;
data ds2DegF_6 (keep=(degF) overwrite=YES)
     ds2AvgF_6 (keep=(avgF) overwrite=YES)
     ;
declare double degF having label 'Temp in Fahrenheit' format F6.1;
declare double avgF having label 'Avg Temp in Fahrenheit' format F6.1;
declare double sum;
```

```
declare integer cnt;
declare package conv cnv();
declare thread temps temps;  ❸
retain sum cnt;

method init();
    sum = 0;
    cnt = 0;
end;

method run();
    set from temps threads=4;  ❹
    degF = cnv.C_to_F(degC);
    sum = sum + degF;
    cnt = cnt + 1;
    output ds2DegF_6;
end;

method term();
    avgF = sum / cnt;
    output ds2AvgF_6;
end;

enddata;
run;
quit;
```

❶ A thread is created in the Work library. The `overwrite=yes` option deletes an existing thread of the same name if one exists.

❷ The method `run()` iterates on the input data.

❸ The thread must be declared before it is used.

❹ DS2 launches four threads to read the data.

---

[1] SAS Institute Inc. 2015. *SAS® 9.4 DS2 Language Reference, Fifth Edition*. Cary, NC: SAS Institute Inc.

Index

About This Book

Purpose

PROC DS2: SAS® Programming Methods at Work introduces a new SAS procedure. New in SAS 9.4, PROC DS2 provides programming concepts that can be used for the first time by many SAS programmers. Through examples, the book helps SAS DATA step programmers learn about PROC DS2 features such as data types, methods, packages, and threads.

Is This Book for You?

PROC DS2: SAS® Programming Methods at Work is for anyone interested in learning this new SAS procedure. This book explains the basic concepts from the ground up.

Prerequisites

PROC DS2: SAS® Programming Methods at Work assumes that the reader has at least a basic understanding of SAS DATA step programming.

Scope of This Book

PROC DS2: SAS® Programming Methods at Work covers the basic concepts of PROC DS2 so that DATA step programmers can quickly take advantage of the new procedure.

It discusses more advanced uses of PROC DS2 for high-performance procedures or in-database computing.

About the Examples

Software Used to Develop the Book's Content

PROC DS2: SAS® Programming Methods at Work was developed using the third maintenance release for SAS 9.4.

Example Code and Data

All code in this book was created using the SAS program files that are provided. No other external data are needed.

You can access the example code and data for this book by accessing the author's page at http://support.sas.com/publishing/authors. Click the name of the author. Then, look for the thumbnail image of this book. Select **Example Code and Data** to display the SAS program files that are included in this book.

If you are unable to access the example code and data, send email to saspress@sas.com.

SAS University Edition

If you are using SAS University Edition to access data and run your programs, then check the SAS University Edition page at http://support.sas.com/software/products/university-edition/index.html to ensure that you have the products that you need to run the example code.

Output and Graphics Used in This Book

In the book, all output goes to the default Results window. No special drivers are required.

Additional Help

Although this book illustrates many analyses that are regularly performed in businesses across industries, you might have specific questions. To fully support you, SAS and SAS Press offer you the following resources:

- For questions about topics covered in this book, contact the author through SAS Press.
  - Send questions by email to saspress@sas.com. Include the book title in your correspondence.
  - Submit feedback on the author's page at http://support.sas.com/author_feedback.
- For questions about topics beyond the scope of this book, post questions to the relevant SAS Support Communities at https://communities.sas.com/welcome.
- SAS maintains a comprehensive website with up-to-date information. One page that is particularly useful to both the novice and the seasoned SAS user is the Knowledge Base. Search for relevant notes in the "Samples & SAS Notes" section of the Knowledge Base at http://support.sas.com/resources.
- Registered SAS users or their organizations can access SAS Customer Support at http://support.sas.com. Here, you can pose specific questions to SAS Customer Support; under **SUPPORT**, click **Submit a Problem**. Provide an email address to which replies can be sent, identify your organization, and provide a customer site number or license information. This information can be found in your SAS logs.

Keep in Touch

We look forward to hearing from you. We invite questions, comments, and concerns. If you want to contact us about a specific book, include the book title in your correspondence.

Contact the Author through SAS Press

- By email: saspress@sas.com
- Via the web: http://support.sas.com/author_feedback

Purchase SAS Books

For a complete list of books available through SAS, visit sas.com/store/books.

- Phone: 1-800-727-0025
- Email: sasbook@sas.com

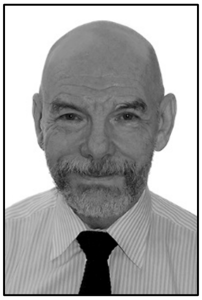
Subscribe to the SAS Training and Book Report

Receive up-to-date information about SAS training, certification, and publications via email by subscribing to the SAS Learning Report monthly e-newsletter. Read the archives and subscribe today at http://support.sas.com/community/newsletters/training.

Publish with SAS

SAS is recruiting authors! Are you interested in writing a book? Visit http://support.sas.com/saspress for more information.

About The Author



Peter Eberhardt has been a SAS consultant since 1982; his company, Fernwood Consulting Group, Inc., is a SAS Alliance Partner. In addition to providing consulting services to the Financial and Health Care sectors, Peter has been an active contributor to the SAS user community as a speaker at user groups across Canada, the United States, the Caribbean, and China. He has also served on the Content Advisory Team for SAS Global Forum and as the Academic Chair for SESUG (2008, 2012) and PharmaSUG China (2015).

Learn more about this author by visiting his author page at http://support.sas.com/eberhardt. There you can download free book excerpts, access example code and data, read the latest reviews, get updates, and more.