Paper 1283-2014

I Object: SAS® Does Objects with DS2

Peter Eberhardt, Fernwood Consulting Group Inc.

Xue Yao, Winnipeg Regional Health Authority, Winnipeg, MB

ABSTRACT

The DATA step has served SAS® programmers well over the years, and although it is powerful, it has not fundamentally changed. With DS2, SAS has introduced a significant alternative to the DATA step by introducing an object-oriented programming environment. In this paper, we share our experiences with getting started with DS2 and learning to use it to access, manage, and share data in a scalable, threaded, and standards-based way.

INTRODUCTION

DS2 is a new programming language that is a powerful tool for advanced data manipulation. DS2 is based on object-oriented concepts wherein objects are instances of classes and methods are basic program execution units. It is designed to be simpler to develop programs as well as easier to understand programs to ease maintenance down the road thus, lending itself to more robust programs. As a part of Base SAS, the PROC DS2 enables you to submit DS2 language statements; as a part of SAS High-Performance Analytics environment, the PROC HPDS2 executes DS2 language statements.

We are first going to look at some of the basic components of DS2. This will not be a comprehensive review of the 1,000+ page reference document, rather a review of those parts we feel are salient to getting started. From there we will do a comparison of DS2 and the DATA step followed by a discussion on when you may want to use DS2. We will wrap up with an example DS2 program and compare it to the original DATA step program.

DS2 OVERVIEW

DS2 has many features. Since this can be overwhelming to the SAS programmer getting started with DS2 we will cover some of the features of DS2 we think are critical to know. As your progress with your DS2 programming you will find there are more features of which you can take advantage. Here we will review four components: Methods, DATA program, Package, and Thread. Before we do that we will look at scope and data types.

SCOPE

In programming, ***scope*** is used to mean where a variable is "visible"; put another way, where its value can be accessed. If you write SAS macro functions you will be familiar with ***local*** (%local) and ***global*** (%global) macro variables. A variable declared within a macro function as:

```
%local macVar
```

is only available in the macro function; when the function completes the variables declared as %local are no longer available. On the other hand, macro variables defined as:

```
%global macVar;
```

are available to PROCs, macros (variables or functions) and DATA steps during the entire SAS session. What sometimes will confuse a SAS programmer learning macro programming is the ability to have two macro variables with the same name but with different scope – one global and one local. When a macro variable is defined local in a macro function, its value will take

precedence over a global macro variable with the same name ***while the macro function is executing***. When the macro function completes execution, then the global macro variable is once again available.

The DATA step has no concept of scope; all variables are accessible from in any part of the DATA step (of course if the variable has yet to be assigned a value, accessing the variable will result in a missing value). We could say the variables are global to the DATA step.

In DS2 scope an attribute of identifiers where identifiers refer to program entities:

- method names
- functions
- data names
- labels
- program variables

Although we use program variables as examples, all identifiers are subject to scoping rules.

Scope describes where in a program a variable can be accessed. Global variables have global scope and are accessible from anywhere in the program while local variables have local scope, that is, are accessible only from within the block in which the variable was declared while that block is executing. Each variable in any given scope must have a unique name, but variables in different scopes can have the same name. When scopes are nested, if a variable in an outer scope has the same name as a variable in an inner scope, the variable within the outer scope is hidden by the variable within the inner scope. For example, in the following program two different variables share the same name, x. Global variables ***x and y*** (declared outside all method blocks) have global scope, and local variable ***x*** (declared in INIT) has local scope. Within the local scope of method INIT, local variable ***x*** hides global variable ***x*** therefore the assignment statement assigns 5 to local variable ***x***; since ***y*** is a global variable it is accessible in all methods.

```
proc ds2;
data _null_;
declare int x;  /* global x in global scope */
declare int y;  /* global y in global scope */
method init();
   declare int x;  /* local x in local scope */
   x = 5;         /* local x assigned 5 */
   y = 6;         /* global y assigned 6 */
   put '0. in init() ' x= y=;
end;
method run();
    put '1. in run()  ' x= y=;
   x = 1;
    put '2. in run()  ' x= y=;

end;
```

```
method term();
end;
enddata;
run;
quit;
```

the method INIT first runs, assigns a value (5) to the local variable x and prints x to the log. After method INIT runs, method RUN executes. When we first print the value of ***x*** to the log in method RUN we see it is no longer 5 (actually has no value); the value of 5 belonged to the variable ***x*** which was local to method INIT. Since INIT is no longer active, the variable with the value 5 is no longer available. The LOG:

```
0. in init()  x=5 y=6
1. in run()   x= y=6
2. in run()   x=1 y=6
```

When considering the scope of a variable, a good rule is to keep variables local as much as possible.

DATA TYPES

The SAS data step is a powerful programming language based on two data types: numeric (double precision floating point), and fixed length character. The data type can be explicitly stated using the ***LENGTH***, ***FORMAT***, or ***ATTRIB*** statements, or it can be implicitly determined based on the result of a calculation; implicit determination is very common in SAS programs. Because there is no need to explicitly declare variables, subtle errors can creep into DATA step programs when variables are misspelled. DS2 changes this.

First, DS2 provides a rich array of data types; these data types match the data types found in ANSI standard RDBMS. In addition, there is a new ***DECLARE*** statement to define the variable. Data types available in DS2 are:

CHAR(***n***)	a fixed length character string of maximum n characters. This is the same as using LENGTH $n in the DATA step
NCHAR(***n***)	a fixed-length character string like CHAR but uses a Unicode national character set, here ***n*** is the maximum number of multi-byte characters to store. Depending on the platform, Unicode characters use either two or four bytes per character and support all international characters.
VARCHAR(***n***)	a varying-length character string, where ***n*** is the maximum number of characters to store. The maximum number of characters is not required to store each value. If varchar(10) is specified and the character string is only five characters long, only five characters are stored in the column.

NVARCHAR(***n***)	a varying-length character string like VARCHAR but uses a Unicode national character set, where ***n*** is the maximum number of multi-byte characters to store. Depending on the platform, Unicode characters use either two or four bytes per character and can support all international characters.
BIGINT	a signed, exact whole number, with a precision of 19 digits. The range of integers is -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
INTEGER	a signed, exact whole number, with a precision of ten digits. The range of integers is -2,147,483,648 to 2,147,483,647. Integer data types do not store decimal values; fractional portions are discarded.
SMALLINT	a signed, exact whole number, with a precision of five digits. The range of integers is -32,768 to 32,767.
TINYINT	a very small signed, exact whole number, with a precision of three digits. The range of integers is -128 to 127.
DECIMAL(***p,s***) / NUMERIC(***p,s***)	a signed, exact, fixed-point decimal number, with user specified precision (p) and scale (s). The precision and scale determines the position of the decimal point. The precision is the maximum number of digits that can be stored to the left and right of the decimal point, with a range of 1 to 52. The scale is the maximum number of digits that can be stored following the decimal point. Scale must be less than or equal to the precision. For example, decimal(9,2) stores decimal numbers up to nine digits, with a two-digit fixed-point fractional portion, such as 1234567.89.
DOUBLE	a signed, approximate, double-precision, floating-point number. Allows numbers of large magnitude and permits computations that require many digits of precision to the right of the decimal point. This is the DATA step numeric.
FLOAT(***p***)	a signed, approximate, single-precision or double precision, floating-point number. The user-specified precision determines whether the data type stores a single-precision or double-precision number. If the specified precision is equal to or greater than 25, the value is stored as a double-precision number, which is a DOUBLE. If the specified precision is less than 25, the value is stored as a single-precision number, which is a REAL. For example, float(10) specifies to store up to ten digits, which results in a REAL data type.
REAL	a signed, approximate, single-precision, floating-point number.
BINARY(***n***)	a fixed-length binary data, where n is the maximum number of bytes to store. The maximum number of bytes is required to store each value regardless of the actual size of the value.
VARBINARY(***n***)	stores varying-length binary data, where n is the maximum number of bytes to store. The maximum number of bytes is not required to store each value. If varbinary(10) is specified and the binary string uses only five bytes, only five bytes are stored in the column.

DATE	a calendar date. A date literal is specified in the format yyyy-mm-dd: a four-digit year (0001 to 9999), a two-digit month (01 to 12), and a two-digit day (01 to 31). For example, the date September 24, 1975 is specified as 1975-09-24.
TIME(***p***)	a time value. A time literal is specified in the format hh:mm:ss[.nnnnnnnnn]; a two-digit hour 00 to 23, a two-digit minute 00 to 59, and a two-digit second 00 to 61 (supports leap seconds), with an optional fraction value. For example, the time 6:30 a.m. is specified as 06:30:00. When supported by a data source, the p parameter specifies the seconds precision, which is an optional fraction value that is up to nine digits long.
TIMESTAMP(***p***)	stores both date and time values. A timestamp literal is specified in the format yyyy-mm-dd:hh:mm:ss[.nnnnnnnnn]: a four-digit year 0001 to 9999, a two-digit month 01 to 12, a two-digit day 01 to 31, a two-digit hour 00 to 23, a two-digit minute 00 to 59, and a two-digit second 00 to 61 (supports leap seconds), with an optional fraction value. For example, the date and time September 24, 1975 6:30 a.m. is specified as 1975-09-24:06:30:00. When supported by a data source, the p parameter specifies the seconds precision, which is an optional fraction value that is up to nine digits long.

Second, DS2 provides an option, ***SCOND=,*** which controls how undeclared variables are handled. The options are:

WARNING	Implicit declaration of variables occurs. A WARNING: message is written to the SAS log. This is the default behavior.
NONE	Implicit declaration of variables occurs. No messages are written to the SAS log. This is how the DATA step works.
NOTE	Implicit declaration of variables occurs. A NOTE: is written to the SAS log.
ERROR	Declaration by assignment does not occur. An ERROR: message is written to the SAS log. This is also known as variable declaration strict mode. This is the behavior of most programming languages.

Setting SCOND=ERROR is a good practice. Also note there is a new system option DS2COND which has the same effect. If both the system option and the PROC option are set, the PROC option takes precedence.

Variables are defined using the DECLARE statement. Variables declared within a method are local to that method while variables declared outside all methods are global; this makes them available to all methods. To explicitly declare variable use the DECLARE statement, The DECLARE statement takes the form:

```
DECLARE dataType varName HAVING label ‘label text’ FORMAT sasfmt.
```

For example:

```
DECLARE DOUBLE           feeAgePrem65   HAVING label  'Age Premium (65+)' format
comma6.2;
DECLARE DECIMAL(10,2)    feeAgePrem65X  HAVING label  'Age Premium (65+) * services';
DECLARE INTEGER i j; /* for DO loops */
```

We will touch on data types again later.

METHOD

Methods are the building blocks of DS2 programs, threads, and packages. A Method is a named unit consisting of related program statements; in DS2, all executable code must reside in methods. A method can be invoked multiple times by name. In its simplest a method has following form:

```
method method_name();
...
     DS2 statements
...
end;
```

To be more useful, methods can take arguments and also return a value; in this way they act the same as a SAS function. The following example shows a method that converts temperature from degrees Fahrenheit to degrees Celsius:

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

This method takes one argument, a DOUBLE, representing the temperature measured in Fahrenheit; it returns a DOUBLE, the equivalent temperature in Celsius. To invoke the method, you call it as you would a SAS function:

```
DECLARE float c having label ‘Celsius degrees’ format 5.1;
c = F_to_C(212.);
```

Methods are global in scope . There are two types of methods: predefined and user-defined. First, there are four predefined methods: ***INIT***, ***RUN***, ***TERM***, and ***SETPARMS***. Every DS2 program will contain, either implicitly or explicitly, the three methods ***INIT***, ***RUN*** and ***TERM***; if you do not explicitly define them the DS2 compiler will generate them. The ***INIT*** method is automatically called once at the beginning of the program and the ***TERM*** method is automatically called once at the end of the program; this provides an efficient and structured mechanism to invoke start-up and clean-up code. After the ***INIT*** method runs, the ***RUN*** is called; by default, this method will iterate once for each input row. The ***SETPARMS*** method is used in a thread to initialize parameters.

When the built-in methods ***INIT***, ***RUN***, ***TERM*** are explicitly defined, they must be defined without any parameters and without a return value; adding parameters and/or returning a value will result in a compile error. These three methods provide a more structured framework than the SAS DATA Step implicit loop concept.

As noted above, user-defined methods can take parameters and return values; each parameter must have a data type. DS2 allows for method overloading, that is using the same method name but different parameter lists (the method signature). When

the method is called, DS2 determines which instance to use based on the arguments passed in. The best match will be the one for which the number of method parameters is equal to the number of arguments, and such that no other method signature has as many exact parameter type matches for the given argument list. If a best match is not found, an error occurs. The following example shows the F_to_C method with two signatures: one for a double precision argument and one for an integer:

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

The RETURN statement is used to return a value. And each RETURN statement that appears in the method must have an associated return expression.

Methods can also return values in through the parameter list; in this case there can be no RETURN statement in the method. To return values through the parameter list use the IN_OUT modifier:

```
method F_to_C(IN_OUT INTEGER F, IN_OUT DOUBLE C);
  /* convert degrees fahrenheit to degrees celsius */
  C = (F - 32) * (5. / 9.);
 end;
```

Methods can have the same name as a SAS function; if this is the case the SAS function is hidden. To access the SAS function you need to use the SYSTEM modifier:

```
td = SYSTEM.date();
```

PACKAGES

DS2 packages are collections of logically related methods and variables that can be shared and re-used in DS2 programs and threads; for example, you may have a set of measurement conversion functions tested and validated that can then be used through- out the organization. These packages can be thought of as a library that contains the methods. The method stored in a package can be accessed by creating an instance of the package and then using dot notation to call the method. SAS also provides packages for your use; these include: FCMP, Hash, Logger, Matrix and SQLSTMT.

- The FCMP package supports calls to FCMP functions and subroutines from within the DS2 language.
- The Hash package enables you to quickly and efficiently store, search, and retrieve data based on unique lookup keys .
- The Logger package provides a basic interface to the SAS logging facility.
- The Matrix package provides a powerful and flexible matrix programming capability.

- SQLSTMT provides a way to pass FedSQL statements to a DBMS for execution and to access the result set returned by the DBMS.

The PACKAGE statement defines a package, and the DECLARE PACKAGE statement constructs an instance of the package. The following example shows a package definition as well as use of the package:

```
1 PACKAGE convert / overwrite = yes;
method C_to_F(INTEGER C) returns DOUBLE;
  /* convert degrees fahrenheit to degrees celsius */
  return 32 + (C *  (9. / 5.));
end;
method IN_to_CM(DOUBLE inch) returns double;
  return inch * 2.54;
end;
...
ENDPACKAGE;
2 ATA measures (overwrite=YES);
3 eclare package convert cnv();
  method init();
    type = 'C';
    do C = 0. to 100.;
4      F = cnv.C_to_F(C);
       output;
end;
end;
ENDDATA;
```

1. PACKAGE convert / overwrite = yes;
   All methods between the PACKAGE/ENDPACKAGE are included. ***overwrite=yes*** will recreate the package if it exists; DS2 will not automatically overwrite packages
2. DATA measures (overwrite=YES);
   Starts the DATA program block. ***overwrite=yes*** will recreate the package if it exists; DS2 will not automatically overwrite data tables
3. DECLARE PACKAGE convert cnv();
   Instantiates an instance of the package convert. The instance is called cnv. Since this is declared outside all methods it has global scope
4. cnv.C_to_F(C);
   Calls the C_to_F() method through the package instance cnv.

A package instance can be created two ways. First, as shown above, the instance variable has parenthesis; the parenthesis tells DS2 to automatically instantiate the package instance:

```
DECLARE PACKAGE convert cnv();
```

Second, the instance variable is declared without parenthesis and later it the program it is instantiated with the _NEW_ operator.

```
DECLARE PACKAGE convert cnv;
  ...
cnv = _NEW_ [THIS] convert();
```

The scope of a package instance follows the same rules as the scope for variables; that is, if it declared within a method it is local to that method and if it is declared outside of a method it has global scope. The scope of the package instance can also be set with the _NEW_ operator. In the example above, the keyword ***THIS*** tells DS2 to make cnv global in scope. In addition to the THIS keyword, you can also supply the name of another variable and the new package instance will have the same scope as that variable.

THREAD

A DS2 program can run in two different ways: as a program and as a thread. Using threaded processing, sections of the code can run concurrently; each section that executes concurrently is said to be running in a thread. When running in a thread, input data can only come from database tables, not other threads, and output data are returned to the DS2 program that started the thread. If running as a program, input data can include both rows from database tables and rows from DS2 program threads and output data can be both database tables and rows returned to the program. A thread is defined by the THREAD statement and ended by ENDTHREAD statement. The instance of a thread is created by using the DECLARE THREAD statement. To execute the thread, use the SET FROM statement. In the following example, a simple thread t is created. Then the instance t_instance is declared, and two threads are executed, using the SET FROM statement in the RUN method.

```
thread t;
 dcl int x;
 method init();
    dcl int i;
    do i=1 to 3;
       x=i;
       output;
    end;
 end;
endthread;
data;..
 dcl thread t t_instance;
 method run();
    set from t_instance threads=2;
    put ‘x= ’ x;
```

```
 end;
enddata;
```

Threads are most effective when used in a threaded multi-processing environment.

PROGRAMMING BLOCKS

We have seen all the main programming blocks above. To reiterate, the main programming blocks in DS2 include with the block delimiters are:

Block	Delimiters
DS2 program	PROC DS2 … QUIT
DATA program	DATA … ENDATA Only one DATA program is allowed
Method	METHOD … RUN Methods must be defined before they are called either explicitly or with a FORWARD reference
Package	PACKAGE … ENDPACKAGE
Thread	THREAD … ENDTHREAD

Using DS2, new variables are expected to be declared by using DECLARE statement. The DECLARE statement assigns a name, designates the type of data and allocates a specified amount of memory to the variable. More than one variable can be declared in one DECLARE statement. A variable declared in the outermost programming block has global scope in the block. Scope describes where in a program a variable can be accessed. Global variables have global scope and can be accessed anywhere in the programming block; by default, in a DATA program all global variables are sent to the data table being created. DS2 supports three types of global scope: program, package, and thread. A variable declared in any method has local scope and are accessible only from within the program or block in which the variable was declared. Implicitly declared variables are added to the global set of a DS2 program and thus also sent to the data table being created, variables declared in method, package or thread are not. Each variable in any given scope must have a unique name, but variables in different scopes can have the same name. In this example, any reference to X in the INIT method will refer to the local declaration of X, but any reference to X in the RUN method will refer to the global declaration of X.

```
declare int x;
method init();
 declare int x;
end;
method run();
end;
```

In the following example, the DECLARE statement out of the method assigns the datatype CHAR which has 15 characters to variable FIRSTNAME and LASTNAME. In the first DATA statement, the table example1 is created with specified values for two variables and the OVERWRITE=YES option enables you to re-create the table at each execution. In the second DATA statement, the table example1 is used as input for the table names.

```
proc ds2;
data example1 (OVERWRITE=YES);
dcl char(15) firstname lastname;
 method init();
    firstname='Stephanie'; lastname='Oraternisky'; output;
    firstname='Chunjiao'; lastname='Zhang'; output;
    firstname='William'; lastname='Green'; output;
 end;
enddata;
run;
data names (OVERWRITE=YES);
 method run();
    set example1;
    if lastname='Green';
 end;
enddata;
run;
quit;
```

A DS2 program can have multiple subprograms followed by an optional data program, but there can be only one data program and the data program must be the last subprogram.

Earlier we noted that you can access a hidden SAS function by using the SYSTEM modifier; we can do something similar with global variables. If a variable is named as both global and local then when the block where the variable is local is executing, accessing the variable name means you are accessing the local instance. If you want to access the global instance you use the THIS modifier:

```
data x;
1 eclare double x;
 method run();
2    declare double x;
    this.x=1.0; /*assign 1.0 to global x*/
    x=0.0;   /*assign 0.0 to local x*/
    output;
 end;
enddata;
run;
```

In this code block we have a variable called x defined as global (1) and local (2) and the use of this.x to access the global instance.

WHEN TO USE DS2

There are many technical reasons that would compel you to learn and use DS2, but possibly the most important reason is DS2 allows you to modular, robust, and reusable code.

Of course there are technical features in DS2 that will compel you to start implementing your solutions in DS2, most important among these are:

- precision
    - DS2 has several new data types that will allow greater precision in calculation.
- reusable methods
    - PROC FCMP brought the ability to write our own functions in DATA step language. These functions are still available to us.
    - Packages and methods extend the ability of FCMP functions into the object oriented realm,
- SQL
    - SQL code can now be submitted directly
- offload processing
    - When you have access to High-Performance Analytics technologies such as Grid Manager, SAS In Database, and SAS Embedded Process capable databases
- threading
    - if you work in threaded processing environments that are configured either as a Symmetric Multiprocessing (SMP) environment, or a Massively Parallel Processing (MPP) environment engineered specifically to support high-volume parallel processing and high-performance computing.

As noted above, even if no technical reason will compel you to move to DS2, the ability to write more robust programs should compel you.

DS2 EXPERIENCES

To look at how some of the features of DS2 work, we converted an existing DATA step program to DS2; this program was chosen because its structure suited the use of DS2 method. In addition, this is one of a set of programs that have a common set of programming blocks so the re-use of packages can be evaluated. The original program is in Appendix A and the converted program is in Appendix B.

WHAT'S MISSING

The current implementation of DS2 does not support the input statement; this means programs which read raw data and create data tables are not candidates for DS2. In our case this is a disappointment since we have several programs which create tables from raw data files and many these programs would benefit from DS2. We tried tricking DS2 by creating a data set view, but DS2 was not tricked.

DS2 has a rich variety of data types. Base SAS tables still only support fixed length character and double precision numeric variables. If your DS2 programs will be accessing RDBMS data you will be better able to match the data types in your DS2

programs. If you are using SAS data tables, you can use the new data types in your programs, but when the tables are saved the variables are converted to fixed length character or double precision numeric as appropriate. Be sure you read and understand the SAS rules for data type conversion.

MERGE. We so have the SET statement but MERGE is not available (although MERGE is listed as one of the reserved DS2 words). The basic functionality of a merger statement can be accomplished in SQL, however a common validation/cleaning process we use looks something like:

```
data inBoth inOne inTwo;
   merge  one (in=in1) two (in=in2);
   by j;
   if      in1 and in2 then do; /* more statements */ output inBoth; end;
   else if in1         then  do; /* more statements */ output inOne; end;
   else                      do; /* more statements */ output inTwo; end;
  end;
run;
```

this allows us to process the incoming data once and create several output tables.

MISSING VALUES

DS2 still supports SAS missing values. The SQL concept of null has been added to DS2 in addition to the usual SAS missing values. If you deal with missing values you will have to understand how DS2 treats SAS missing values and null values. In DS2, only the two base SAS data types (CHAR, DOUBLE) can have missing values while all the new data types can have null values. DS2 provides a NULL() function to test for a null value and a MISSING() function to test for a SAS missing value or a null value. In addition, DS2 provides two modes for processing null data: ANSI mode and SAS mode. In SAS mode missing values read from a table are handled as SAS missing values; in ANSI mode all SAS numeric missing values are converted to null. If you make use of SAS special missing values (._, .A - .Z) then you will need to process in SAS mode otherwise they will all be converted to null and you will lose information. SAS character variables are considered missing if the character string contains blanks; in ANSI mode this would not be converted to null, rather it is just left as a blank string. The reference manual has six pages dealing with the null/missing values so the description here is simplified but sufficient for most uses.

DATES

For many SAS programmers, and particularly new programmers, SAS dates are usually a source of confusion. With DS2 we now have date data types and these data types are not compatible with SAS dates. To use one of the new date data types with the SAS date functions you need to cast the date variable to a double using the to_double() function:

```
ageDays = intck('day', to_double(patientDOB), to_double(servdate) );
```

When DS2 reads a variable from a SAS dataset that has a date format, it converts it to DS2 date variable thus making it necessary to cast the variables.

CONCLUSION

This has been a quick overview of the DS2 procedure and its main components. Our experience has shown that the learning curve for DS2 is not steep, however there are two areas in which you initially need attention: data type conversion, and missing values. The use of methods and packages will make it easier to build more robust code. For programmers working in a threaded multi-processing environment and/or using SAS High-Performance Analytics technologies can take advantage of threaded processing. For many programmers there is not 'need' to move to DS2 programming, but the move will be worthwhile.

REFERENCES

SAS® 9.4 DS2 Language Reference. SAS Institute Inc. 2013. Cary, NC: SAS Institute Inc.

Your comments and questions are valued and encouraged. Contact the authors at:

Peter Eberhardt
Fernwood Consulting Group
Toronto, ON, Canada
peter@fernwood.ca
www.fernwood.ca
Twitter: rkinRobin

Xue Yao
Winnipeg Regional Health Authority
Winnipeg, MB, Canada
xueyao.statistic@gmail.com

APPENDIX A – THE SAS DATA STEP PROGRAM

```
data flow.adjustments;
       retain gnum     phynum    hnum      servdate
            feecode   feesufix   diagcode
            feepaid   feebilled  feeapproved
            services  patientDOB;
       set source (where=(fiscalyear='2014' AND TYPE = 'SB'));
       drop x;
       length codeSuffix $5.;
       drop codeSuffix;
       ageDays = servdate - patientDOB;
       ageCatYoung = input(put(ageDays, kage.), 1.);
       ageCatOld   = input(put(ageDays, aage.), 1.);
       flowType    = 0;
       format CompareFeeTotal comma9.2;
* SA and AN regular rate increase amount;
* SA ;
       if  feesufix = 'B'
       then
         do;
            x = input(put(feecode, $SACD.), 8.2);
            if x NE .
            then
              do;
                feeSARate  = input(put(feesufix, $SART.), 8.2); /* * services */
                feeSARate = feeSARate * services;
                feeSAExtra = x;
                flowType = flowType + 4;
              end;
         end;
* AN ;
       else if feesufix = 'C'
       then
         do;
            x  = input(put(feecode, $ANCD.), 8.2);
            if x NE .
            then
              do;
                feeANRate = input(put(feesufix, $AURT.), 8.2);
```

```
                feeAnRate = feeANRate * services;
                feeANExtra = x;
                flowType = flowType + 8;
              end;
         end;

* SA and AN regular increase amount;
* SA ;
/*************************************************************************
       if  feesufix = 'B'
       then
         do;
            x = input(put(feecode, $SACD.), 8.2);
            if x NE .
            then
              do;
                feeSARate = input(put(feesufix, $SART.), 8.2); /* * services
                flowType = flowType + 4;
              end;
         end;
* AN ;
       else if feesufix = 'C'
       then
         do;
            x  = input(put(feecode, $ANCD.), 8.2);
            if x NE .
            then
              do;
                feeANRate = input(put(feesufix, $AURT.), 8.2);
                flowType = flowType + 8;
              end;
         end;
*************************************************************************/

* the regular premium increases;
     codeSuffix = feecode || feesufix;
       if flowType = 0
       then
         do;
```

```
          feePREM  = input(put(codeSuffix, $regprem.), 8.2);
          if feePREM ne .
          then
            do;
              feePREMX = feePrem * feeBilled;
              flowType = flowType + 2;
            end;
        end;

* the regular increases;
       if flowType = 0
       then
         do;
           feeREG   = input(put(feecode, $reg.),  8.);
           if feeREG ne .
           then
             do;
               feeREGX  = feeREG * services;
               flowType = flowtype + 1;
             end;
       end;


    x = input(put(diagcode, $cdmDiag.), 8.2);
    if x ne .
    then
      do;
        feeDiagExtra = input(put(feecode, $cdmDgRT.), 8.2);
      end;

   if (feesufix = 'A') and (ageCatYoung < 6) /* under 18 */
   then
     do;
       x = input(put(feecode, $cdmPSB.), 8.2); /* SOB */
       if x NE .
       then
         do;
           feePaedSOB  = input(put(ageCatYoung, cdmKage.), 8.2);
           feePaedSOBX = feePaedSOB * feeBilled;
```

```
          end;
        x = input(put(feecode, $cdmPS.), 8.2);  /* SEL */
        if x NE .
        then
          do;
            feePaedSEL  = input(put(ageCatYoung, cdmKage.), 8.2);
            feePaedSELX = feePaedSEL * feeBilled;
          end;
      end;

      if feesufix = 'A'
      then
        do;
          feeEPrem  = input(put(feecode, $cdmeye.), 8.2);
          feeEPremX = (feeEPrem) * feeBilled;
        end;

/* age premiums */
      if ageCatOld NE .
      then
        do;
          x = input(put(feecode, $ageCode.), 1.);
          if x = 1
          then
            do;

              if ageCatOld <= 2 /* (was agecat = 1) 65+ */
              then
                do;
                  if feecode = 'Axxx'
                  then
                    do;
                      feeAgePrem65  = input(put('Exxx', $cdmAAGE.), 8.);
                      feeAgePrem65X = feeAgePrem65 * services;
                    end;
                  if feecode = 'Axxx'
                  then
                    do;
                      feeAgePrem66  = input(put('Exxx', $cdmAAGE.), 8.);
```

```
                feeAgePrem66X = feeAgePrem71 * services;
              end;
          end;

        if ageCatOld = 2 /* 70+ */
        then
          do;
            if feecode = 'Axxx'
            then
              do;
                 feeAgePrem70  = input(put('Exxx', $cdmAAGE.), 8.);
                 feeAgePrem70X = feeAgePrem70 * services;
              end;
            if feecode = 'Axxx'
            then
              do;
                feeAgePrem71  = input(put('Exxx', $cdmAAGE.), 8.);
                feeAgePrem71X = feeAgePrem71 * services;
              end;
          end;
     end;
    feeAgePremTot = sum(0, feeAgePrem65X,
                        feeAgePrem66X,
                        feeAgePrem70X,
                        feeAgePrem71X
                       );

  end;
CompareFeeTotal = sum(0, feeREGX,
                      feePREMX,
                      feeSARate,
                      feeANRate,
                      feeSAExtra,
                      feeANExtra
                     );
FeeTotal = sum(0, CompareFeeTotal,
                  feeAgePremTot,
                  feeDiagExtra,
                  feeEPremX,
```

```
                        feeE411PremX,
                        feeE411Amt,
                        feePaedSOBX,
                        feePaedSELX
                      );
run;
```

APPENDIX B – THE DS2 PROGRAM

Note the use of FORWARD declaration of the methods. This was done so the run() method is quickly spotted near the top.

```
proc ds2 SCOND=ERROR ;
data adjustments (overwrite = yes);
/* FORWARD declare the methods - they follow method run() in the code */
     FORWARD regularSA_AN;
     FORWARD regularIncrease;
     FORWARD regularPremium;
     FORWARD diagnosticExtra;
     FORWARD PPremium;
     FORWARD EPremium;
     FORWARD APremium;
     FORWARD agePremiums;

/* these were initially declared INT, but this lead to different results */
/* because of the way SAS missing values are handled in INT           */
  DECLARE DOUBLE  ageDays ageCatYoung ageCatOld flowType;

  DECLARE DOUBLE  CompareFeeTotal FlowFeeTotal feeREG  feeREGX feePREM feePREMX feeSARate ;
  DECLARE DOUBLE  feeANRate   feeSAExtra  feeANExtra   feeSARate    feeANRate    feeDiagExtra ;
  DECLARE DOUBLE  feePSOB     feePSOBX    feePSEL     feePSELX      feeEPrem     feeEPremX   ;
  DECLARE DOUBLE  feeEAmt     feeAPrem    feeAPremX   feeAgePrem65  feeAgePrem65X feeAgePrem66;
  DECLARE DOUBLE  feeAgePrem66X feeAgePrem70  feeAgePrem70X feeAgePrem71  feeAgePrem71X feeAgePremTot;

  DECLARE CHAR(5) codeSuffix ;
  DROP  codeSuffix;
  DECLARE DOUBLE x;
  DROP x;
    method run();
        set {select * from source where fiscalyear='2014' AND TYPE = 'SB'};
        flowType = 0;
        ageDays = intck('day', to_double(patientDOB), to_double(servdate) );
        codeSuffix = put(ageDays, kage.);
        ageCatYoung = inputn(put(ageDays, kage.), '3.');
        x  = inputn(put(ageDays, kage.), '3.');
        ageCatOld   = inputn(put(ageDays, aage.), '3.');
        flowType    = 0;
```

```
        regularSA_AN();
        regularPremium();
        regularIncrease();
        diagnosticExtra();
        PPremium();
        EPremium();
        EPremium();
        agePremiums();

        CompareFeeTotal = sum(0, feeREGX,
                              feePREMX,
                              feeSARate,
                              feeANRate,
                              feeSAExtra,
                              feeANExtra
                             );
        FeeTotal = sum(0, CompareFeeTotal,
                        feeAgePremTot,
                        feeDiagExtra,
                        feeAPremX,
                        feeEPremX,
                        feeEAmt,
                        feePSOBX,
                        feePSELX
                       );
    end;

/*************************************************************************/
    method regularSA_AN();
        if  feesufix = 'B'
        then
          do;
            x = inputn(put(feecode, $SACD.), '8.2');
            if x NE .
            then
              do;
                feeSARate  = inputn(put(feesufix, $SART.), '8.2'); /* * services */
                feeSARate = feeSARate * services;
                feeSAExtra = x;
```

```
                flowType = flowType + 4;
              end;
          end;
* AN ;
        else if feesufix = 'C'
        then
          do;
             x  = inputn(put(feecode, $ANCD.), '8.2');
             if x NE .
             then
               do;
                 feeANRate = inputn(put(feesufix, $AURT.), '8.2');
                 feeAnRate = feeANRate * services;
                 feeANExtra = x;
                 flowType = flowType + 8;
               end;
          end;

    end;

/*************************************************************************/
    method regularPremium();
* the regular premium increases;
        codeSuffix = feecode || feesufix;
        if flowType = 0
        then
          do;
            feePREM  = inputn(put(codeSuffix, $regprem.), '8.2');
            if feePREM ne .
            then
              do;
                feePREMX = feePrem * feeBilled;
                flowType = flowType + 2;
              end;
          end;
    end;

/*************************************************************************/
    method regularIncrease();
```

```
* the regular increases;
       if flowType = 0
       then
         do;
           feeREG   = inputn(put(feecode, $reg.),  '8.');
           if feeREG ne .
           then
             do;
               feeREGX  = feeREG * services;
               flowType = flowtype + 1;
             end;
         end;
    end;

/***************************************************************************/
    method diagnosticExtra();
      x = inputn(put(diagcode, $cdmDiag.), '8.2');
      if x ne .
      then
        do;
          feeDiagExtra = inputn(put(feecode, $cdmDgRT.), '8.2');
        end;
    end;

/***************************************************************************/
    method PPremium();
       if (feesufix = 'A') and (ageCatYoung < 6)  /* under 18 */
       then
         do;
           x = inputn(put(feecode, $cdmPSB.), '8.2'); /* SOB */
           if x NE .
           then
             do;
               feePSOB  = inputn(put(ageCatYoung, cdmKage.), '8.2');
               feePSOBX = feePSOB * feeBilled;
             end;
           x = inputn(put(feecode, $cdmPS.), '8.2');  /* SEL */
           if x NE .
           then
```

```
            do;
              feePSEL  = inputn(put(ageCatYoung, cdmKage.), '8.2');
              feePSELX = feePSEL * feeBilled;
            end;
         end;
    end;

/*****************************************************************************/
    method EPremium();
     feeEPrem  = inputn(put(feecode, $cdm411P.), '8.2');
     feeEPremX = (feeEPrem) * feeBilled;
     feeEAmt  = inputn(put(feecode, $cdm411A.), '8.2');
    end;

/*****************************************************************************/
    method APremium();
     if feesufix = 'A'
     then
       do;
         feeAPrem  = inputn(put(feecode, $cdmeye.), '8.2');
         feeAPremX = (feeAPrem) * feeBilled;
       end;
    end;

/*****************************************************************************/
   method agePremiums();
/* age premiums */
     if ageCatOld NE .
     then
       do;
         x = inputn(put(feecode, $ageCode.), '1.');
         if x = 1
         then
           do;

            if ageCatOld <= 2 /* (was agecat = 1) 65+ */
            then
              do;
                if feecode = 'Axxx'
```

```
                then
                  do;
                    feeAgePrem65  = inputn(put('Exxx', $cdmAAGE.), '8.');
                    feeAgePrem65X = feeAgePrem65 * services;
                  end;
                if feecode = 'Ayyy'
                then
                  do;
                    feeAgePrem66  = inputn(put('Eyyy', $cdmAAGE.), '8.');
                    feeAgePrem66X = feeAgePrem71 * services;
                  end;
              end;
              if ageCatOld = 2 /* 70+ */
              then
                do;
                  if feecode = 'Axxx'
                  then
                    do;
                      feeAgePrem70  = inputn(put('Ezzz', $cdmAAGE.), '8.');
                      feeAgePrem70X = feeAgePrem70 * services;
                    end;
                  if feecode = 'Ayyy'
                  then
                    do;
                      feeAgePrem71  = inputn(put('Exxx', $cdmAAGE.), '8.');
                      feeAgePrem71X = feeAgePrem71 * services;
                    end;
                end;
            end;
            feeAgePremTot = sum(0, feeAgePrem65X,
                                   feeAgePrem66X,
                                   feeAgePrem70X,
                                   feeAgePrem71X
                               );
        end;
    end;
enddata;
run;
quit;
```