

One [Data] Step Ahead...

An Introduction to SAS DS2

Richard Superfine

UnitedHealth Care - Financial Data Management

Richard Superfine



- Senior Analytics Solutions Architect at UHC
- Extensive background in IT, Software Development and Financial Analytics across many business areas
- “Passion for data”
- Contact: richard_superfine@uhc.com



- Flight Instructor and Commercial Pilot

What we will cover

- Introduction to DS2
- Review of “Traditional” SAS Data Step
- Variables and Scope
- DS2 Data Types
- Missing vs NULL
- Arrays
- Methods
- Threads
- Packages
- In-Database Accelerator
- Summary/DS2 Gotcha’s
- References

What are people saying about DS2?

"DS2 is the intersection of SAS DATA Step and ANSI SQL:1999"



"DS2 is a new SAS proprietary programming language that is appropriate for advanced data manipulation."



"DS2 is included with Base SAS."

"DS2, the last language you will ever learn."

- Paul Kent VP, SAS Platform R&D

The case for DS2

- **Where can I see the most benefit from DS2?**
 - In processes that currently require large data movement.
 - Anywhere you have a complex process that could benefit from structured code.
 - Anywhere you need to maintain consistency with the data types in your source database.
- **Where can I run DS2 programs?**
 - Base SAS
 - SAS High Performance Grid
 - In-Database (SAS In-Database Code Accelerator)
 - High-Performance Analytics Server
 - SAS In-Memory Analytics

The case for DS2

- **What does DS2 look like?**

- It is based on object-oriented concepts from the programming language world

- You are going to notice the word “method” a lot

```
method init();  
method run();  
method term();
```

- You are going to be using objects and setting their properties and calling their methods with DOT NOTATION

- **But we'll get there....let's first review what we already know...**

Data Step: A Review

- Data Step basic format:

```
|data output;  
    set input;  
    /* do some stuff */  
run;
```

- New variables can be introduced to the data set implicitly

```
data output;  
    set input;  
    z=x+y;  
run;
```

The Program Data Vector

- **Compile Phase**
 - Builds PDV from deterministic values
- **Execute Phase**
 - Non-deterministic value execution
 - Example: if $x > 5$ then drop x;
 - Execution of whole data step, one observation at a time
- **All Compile Phase statements are completed before Execute Phase begins, regardless of the order they're written in!**

Name	Type	Length	Retain?	Missing Protect?	Keep?	Value
x	Numeric	8	Yes	No	Yes	1
z	Numeric	8	No	No	Yes	.

Data Step Control Flow

- **Top of Data Step:**
 - For each Variable:
 - IF RETAIN FLAG = “NO” THEN Set Variable = MISSING
- **SET Statement**
 - Any more observations to read?
 - If not then END EXECUTION
 - Otherwise read the next observation
- **Rest of code executes....**
- **Bottom of Data Step:**
 - For each Variable:
 - IF KEEP FLAG = “YES” THEN Write Variable to temporary data set
- **GO TO Top of Data Step**

Issues with the Data Step

- **Variable Scope is global**
- **Limited reusability of code**
- **Limited data types**
- **Single threaded**

DS2 to the rescue!

- **Latest version is SAS DS2 v9.4**
- **DS2 does not replace the “Traditional” Data Step**
 - It *extends* it
 - Leverages your knowledge of Data Step, so you use what you already know
- **Brings the following benefits:**
 - Based on (some) object-oriented principles
 - Easier to understand than “classic” data steps
 - Explicitly declared variables
 - Scoped variables
 - Richer set of Data Types
 - SQL NULL as well as MISSING values
 - Ability to create libraries of code for reuse
 - Mechanisms for multi-threading

Structure of a DS2 Program

Traditional Data Step

- **DATA <output dataset>;**
 - ...Retain/Keep/Drop Commands...
 - ...Data step commands, including SET Statement...
- **RUN;**

DS2

- **PROC DS2;**
 - DATA <output dataset>;
 - ...Variable declarations...
 - ...Retain/Keep/Drop Commands...
 - ...Method definitions...
 - ...Data step commands, including SET Statement...
 - ENDDATA;
 - RUN;
 - ...More data blocks...
- **QUIT;**

Variable Declarations

- **New Variables can be explicitly declared before they are used,**

```
DECLARE integer Account_No;  
DCL char(15) firstname lastname;  
DCL double Premium_Fee HAVING label 'Premium Fee' format 5.1;
```

- **Can still use SET statement,**
 - Reads column information for table and creates a global variable for each one, with the same data type
- **OR you can use variables implicitly.**
 - SAS will try to determine the data type from the value
 - By default a warning is sent to the SAS log
 - This behavior can be controlled by SCOND settings

SCOND Settings

- **DS2SCOND is a system option**
 - e.g. set in SAS Config file
- **SCOND is an option on the DS2 Procedure**
 - e.g. PROC DS2 SCOND=WARNING

Settings to Control Variable Declaration

DS2SCOND/SCOND Setting	Effect on Variable Declaration
WARNING	Declaration by assignment occurs. Warning messages are written to the SAS log. This is the default behavior.
NONE	Declaration by assignment occurs. No messages are written to the SAS log.
NOTE	Declaration by assignment 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.

- **ERROR is the recommended setting, or even better: always explicitly declare your variables using DECLARE/DCL**
 - This is similar to OPTION EXPLICIT in Microsoft VB.NET

Variable Scope

- **“SCOPE” means where a variable’s value is “visible”, or where it can be “accessed”**
 - SCOPE is used in SAS Macros:
 - %local myVar vs %global myVar
 - Local scope “trumps” global scope, in the case of a variable having the same name
- **There is no concept of “SCOPE” in traditional Data Steps**
 - variables are always “global” throughout the entire data step
- **In DS2, “SCOPE” DOES apply to Variables, as well as:**
 - Method names
 - Functions
 - Data Names
 - Labels

DS2 Data Types

- **DS2 supports a much richer set of data types than its traditional data step counterpart. They are ANSI SQL compliant.**
 - **Numeric types:**
 - INTEGER
 - BIGINT
 - SMALLINT
 - TINYINT
 - REAL
 - DOUBLE
 - FLOAT(n)
 - DECIMAL/NUMERIC (p,s)
 - **String types:**
 - CHAR(n)
 - VARCHAR(n)
 - NCHAR(n)
 - NVARCHAR(n)
 - **Other types:**
 - TIME
 - DATE
 - TIMESTAMP
 - BINARY(n)
 - VARBINARY(n)

SAS Missing Values vs NULL

- **DS2 still supports SAS Missing Values, but introduces the concept of a NULL value**
 - NULL comes from the RDBMS world
 - It is important to understand the differences between MISSING and NULL
- **DOUBLE and CHAR data types use MISSING**
 - All others only have NULL
- **Anywhere NULL is used in 3-valued logic, the result is NULL**
 - MISSING Values result in a known BOOLEAN value
- **To test for NULL:**
 - IF NULL(myVar) THEN.....
- **To test for MISSING or NULL:**
 - IF MISSING(myVar) THEN

SAS Missing Values vs NULL

- **Two modes for processing NULL data:**
 - PROC DS2; (SAS Mode – default)
 - PROC DS2 ansimode; (ANSI Mode)

	NUMERIC DATA is missing	STRING DATA is missing	STRING DATA is empty string
SAS MODE	MISSING (.)	MISSING	MISSING
ANSI MODE	NULL	NULL	EMPTY STRING

Example: If input data is CHAR(3) set to ' ' then:

SAS Mode – this would be considered as MISSING (' ')

ANSI Mode – this would be considered as NULL

DS2 Arrays

- An “Array” is a collection of variables, *all of the same data type*, bundled together under a common name, indexed in sequence by an integer, starting from index 1
- Here’s an array of n INTEGERS:

Index 1	Index 2	Index n
22	78	121

- You DECLARE an array in SAS just like variables:

```
DECLARE integer myArray[n];
```

- You can change the lower bound and upper bounds of the array:

```
DECLARE char(10) myStringArray[0:7];
```

- You can specify more than one dimension of an array

```
DECLARE timestamp myTimeArray[5,10];
```

- This is an array of 5 elements, each containing an array of 10 elements (a table. 3 dimensions is a cube, 4 dimensions is a...)

DS2 Arrays

- **Let's get “low level techy” for a moment...**
 - The array we just created is a collection of variables in a block of memory

Index 1	Index 2	Index n
22	78	121

- In SAS, this is called a “Temporary Array”
 - They only exist for the duration of the DS2 program
 - Values are automatically kept across iterations (Keep Flag = True)
 - They can be local or global in scope
 - The values are not in the PDV so won't appear in any result table

DS2 Arrays

- Staying “low level techy” for a moment longer...
 - Now consider an array where the values (elements) are essentially pointers to the variables in the PDV

Index 1	Index 2	Index n
x	z	

Name	Type	Length	Retain?	Missing Protect?	Keep?	Value
x	Numeric	8	Yes	No	Yes	1
z	Numeric	8	No	No	Yes	.

- In SAS this is called a “Variable Array”
 - Declared using the VARARRAY statement
 - VARARRAY double a[3];
 - Changing the value of the array element actually changes the value in the PDV
 - Variable arrays are always global in scope

DS2 Arrays

- **To summarize DS2 arrays:**
 - Homogeneous by type
 - Can be multidimensional
 - Indexed by signed integer values (1-based)
 - Not a variable in the PDV, though the elements can refer to variables in the PDV
 - Will not appear in a result table, though variables in the PDV can appear
 - Two types:
 - Temporary – created with DECLARE statement
 - Variable – created with VARARRAY statement
 - Use the := operator to assign one array to another

```
myArray := myOtherArray;
```

```
myArray := (1 2 3 4 5 6 7);
```
 - Use the PUT statement to output element(s) of an array

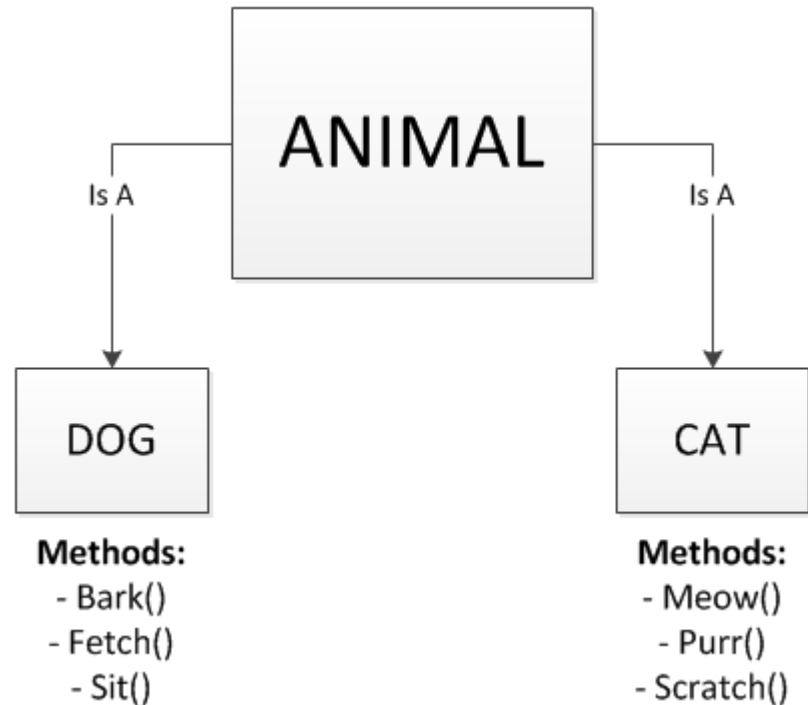
```
PUT myArray[7];
```

```
PUT myArray[*];
```

Methods

- **Come from Object Oriented principles**

- Objects store *state*
 - Encapsulation
- Objects inherit *behaviors* from other classes
 - Levels of *abstraction*
- *Methods*
 - Describe behaviors
 - Repeated invocations
- *Polymorphism, abstract classes, interfaces, inheritance*
 - Not supported, nor needed in SAS (yet!)



Methods

- **Methods are the building blocks of DS2 programs**
 - Akin to functions/procedures/subroutines in other languages
 - Where behaviors are defined and can be repeatedly invoked (executed)
 - Think of it as a module that contains a sequence of instructions to perform a specific task
 - Makes easier work of design, implementation and testing of code
 - Code is more readable, so easier to understand by others
 - Methods are global in scope
 - Each method creates its own local scope

User-Defined Methods

- General format of a SAS DS2 method:

```
method myMethod();  
  
    /* DS2 statements ... */  
  
end;
```

To call/invoke a method:

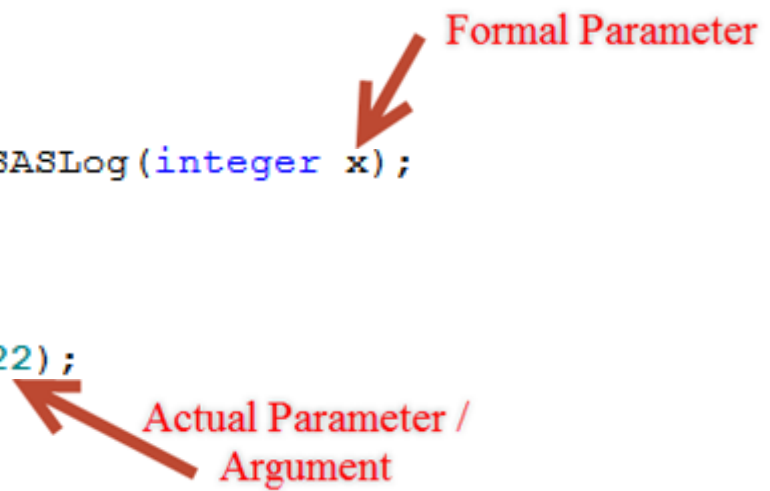
- Simply use the name of the method

```
proc ds2;  
    method doStuff();  
        declare integer x;  
        x = y + 1;  
    end;  
    .....  
    doStuff();  
quit;
```

User-Defined Methods

- The real power of methods come from passing in parameters

```
proc ds2;  
  method writeMsgToSASLog(integer x);  
    PUT x;  
  end;  
  .....  
  writeMsgToSASLog(22);  
quit;
```



The diagram illustrates the relationship between formal and actual parameters in the provided SAS code. A red arrow points from the text "Formal Parameter" to the variable `x` in the method signature `integer x`. Another red arrow points from the text "Actual Parameter / Argument" to the value `22` in the method call `writeMsgToSASLog(22)`.

User-Defined Methods

- **Parameters, by default, are passed by VALUE**
 - They are only in scope within the method
 - Their value is not “visible” outside of the method

```
method swap_vars(double a, double b);  
    declare double temp;  
    temp=a;  
    a=b;  
    b=temp;  
    PUT 'Inside Method....' a= b=;  
end;  
.....  
a=10; b=55;  
PUT 'Before Method Call....' a= b=;  
swap_vars(a,b);  
PUT 'After Method Call....' a= b=;
```

Before Method Call.... a=10 b=55

Inside Method.... a=55 b=10

After Method Call.... a=10 b=55

User-Defined Methods

- Passing parameters by REFERENCE solves this issue
 - Use the key word IN_OUT in the FORMAL PARAMETER list

```
method swap_vars(IN_OUT double a, IN_OUT double b);  
    declare double temp;  
    temp=a;  
    a=b;  
    b=temp;  
    PUT 'Inside Method....' a= b=;  
end;  
.....  
a=10; b=55;  
PUT 'Before Method Call....' a= b=;  
swap_vars(a,b);  
PUT 'After Method Call....' a= b=;
```

Before Method Call.... a=10 b=55

Inside Method.... a=55 b=10

After Method Call.... a=55 b=10

User-Defined Methods

- **Functions**

- Return a value in lieu of using IN_OUT parameters
 - Useful if you want to assign a value to a variable

- Compare:

```
method square_value(integer n) returns integer;  
    return n*n;  
end;  
.....  
a=10;  
sq = square_value(a);
```

```
-----  
  
method square_value(integer n, IN_OUT integer sq);  
    sq = n*n;  
end;  
.....  
a=10;  
square_value(a,sq);
```

User-Defined Methods

- **Overloaded Methods**

- Can have multiple methods with same name but with different signatures (data types and/or number of formal parameters)

```
method my_method(integer x);  
    PUT x;  
end;
```

```
method my_method(integer x, integer y) returns integer;  
    RETURN x + y;  
end;
```

```
method my_method(char(100) x, IN_OUT char(100) y);  
    y = TRIM(x);  
end;
```

User-Defined Methods

- **Method scope**

- Variables declared within a method are local in scope to that method, just like parameters that are passed in by value

```
method my_method();  
    declare double x;    /* local scope */  
    x=10.2;  
end;
```

.....

```
declare double x;        /* global scope */  
x=22.8;  
my_method();
```

User-Defined Methods

- **Method scope**

- You can access global variables of the same name as a local variable within a method by using “THIS”

```
method my_method();  
    declare double x;    /* local scope */  
    x=10.2;  
    PUT x;                /* 10.2 */  
    PUT this.x;           /* 22.8 */  
    this.x=99.9;  
end;  
  
.....  
  
declare double x;        /* global scope */  
x=22.8;  
my_method();  
PUT x;                   /* 99.9 */
```


Summary of Methods

- **Methods**
 - Use object oriented principles from other programming languages
 - Allow you to abstract code into one place, for multiple invocations, so you don't have to "reinvent the wheel"
 - Reduce code generation
 - Compare to Macros
 - Have their own scoping rules
 - May be overloaded
 - May be written as subroutines (by value or by reference) or as functions
- **But wait....there's more!**
 - We just looked at USER DEFINED methods
 - There are SYSTEM methods too!...

System Methods

- **System Methods**

- Predefined by the SAS DS2 system
 - **INIT()**
 - **RUN()**
 - **TERM()**
- If you **do not** explicitly define them in your DS2 program, then the SAS compiler will automatically create default versions of them for you
- If you **do** explicitly define them then they must be “void”
 - Take no parameters and returns no value
 - Otherwise a compiler error will occur
- The purpose of the System Methods is to provide a more structured code framework than the traditional SAS Data Step’s implicit loop provides.
 - They are automatically called; you can not explicitly call system methods

System Methods

- **The traditional SAS Data Step runs as an implicit loop if the SET statement is used**
 - Can get complicated with initiation and clean-up of variables
 - KEEP/DROP/RETAIN statements are hard to read and subject to error
- **The System Methods provide a cleaner way to initiate and clean-up variables**
- **INIT() is called at the start of a DS2 Data program**
 - Only called once – not part of the implicit loop
 - Good place to initialize your variables
- **TERM() is called at the end of a DS2 Data program**
 - Only called once – not part of the implicit loop
 - Good place to finalize code (e.g. write data to SAS log)

System Methods

- **RUN() is automatically executed as soon as INIT() completes**
 - Functional equivalent of a traditional data step
 - If it contains a SET statement, then will run as an implicit loop
 - Local variables' values are lost on each iteration
 - Global variables are set to MISSING (SAS Mode) or NULL (ANSI Mode) on each iteration
 - Use RETAIN to prevent this
- **You are NOT REQUIRED to make use of the implicit loop**
 - You *could* code your entire program in INIT() or TERM()

System Methods

- **A closer look at INIT() ...**
 - Initializes system predefined variables
 - e.g. `_N_` and `_N` (1 and 0 respectively)
 - Great place to initialize variables or create data
 - Regular scoping rules apply.
 - Recall, we use “THIS” to set global variables
- Let's work through some examples...

System Methods

Scope when initializing variables...

```
data;  
  dcl int a; 1 /* A is a global variable */  
  method init();  
    dcl int c; 2 /* C is a local variable */  
    a = 1; 3  
    b = 2; 4 /* B is undeclared so it is global */  
    c = a + b;  
    this.total = a + b + c; 5  
  end;  
enddata;  
run;
```

- 1** A is a global variable because it is declared in the outermost DS2 program.
- 2** C is a local variable because it is declared inside the method block, METHOD INIT().
- 3** Because A is a global variable, it can be referenced within the method block, METHOD INIT().
- 4** Because B is not declared in METHOD INIT(), it defaults to being a global variable. DS2 assigns B a data type of DOUBLE. B appears in the PDV and the output data set.
- 5** THIS.TOTAL simultaneously declares the variable TOTAL as a global variable with the data type of DOUBLE and assigns a value to it based on the values of A, B, and C.

System Methods

- Here's an example where we supply INIT() and RUN() but not TERM()
 - We define data in INIT() and invoke the implicit loop in RUN()
- Do not confuse the method run() with the SAS statement run;

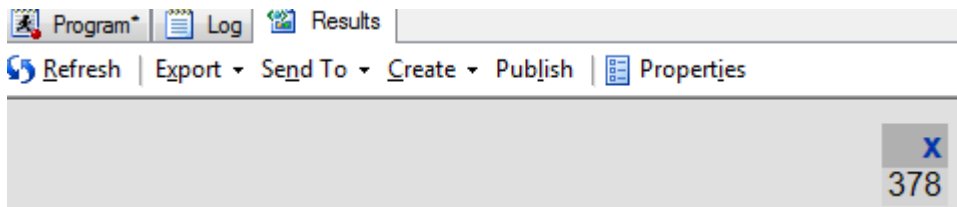
```
proc ds2;
  data example1 (OVERWRITE=YES);
    dcl char(15) firstname lastname;
    method init();
      firstname='Jeremy'; lastname='Rose'; output;
      firstname='Maria'; lastname='Hughes'; output;
      firstname='Bill'; lastname='Green'; output;
    end;
  enddata;
run;

  data names (OVERWRITE=YES);
    method run();
      set example1;
      if lastname='Green';
    end;
  enddata;
run;
quit;
```

System Methods

- Methods can be recursive
 - Given x, calculate $x + (x-1) + (x-2) + \dots 0$

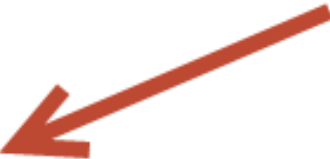
```
|proc ds2;  
  data;  
    DECLARE integer x;  
  
    method sum_downwards(integer x) returns integer;  
      if x <= 0 then  
        return 0;  
      else  
        return x + sum_downwards(x-1);  
      end;  
  
    method run();  
      x = sum_downwards(27);  
    end;  
  enddata;  
  run;  
quit;
```



System Methods

- Methods should be declared before used, otherwise need a FORWARD statement (2 stage parsing)

```
proc ds2;  
  data;  
    DECLARE integer x;  
  
    FORWARD sum_downwards;  
  
    method run();  
      x = sum_downwards(27);  
    end;  
  
    method sum_downwards(integer x) returns integer;  
      if x <= 0 then  
        return 0;  
      else  
        return x + sum_downwards(x-1);  
      end;  
  
  enddata;  
  run;  
quit;
```



System Methods

- In the RUN() method, you can source from the results of an SQL statement instead of a data set

```
data;  
  method run();  
    set {select * from investment where investment_year > 2010};  
  end;  
enddata;  
run;
```

A real life example

- Given a table of attributes, TAG_INPUT

	Filter and Sort	Query Builder	Data ▾	Describe ▾	Graph ▾	Analyze ▾	Export ▾	Send T
	123	⚠	⚠	⚠	⚠	⚠		
	TagID	Attr1	Attr2	Attr3	Tag			
1	1	X	44	Bye				
2	2	1	2	Hello				
3	3	X	44	Hello				

- And a table of rules, TAGGING_RULES

	Filter and Sort	Query Builder	Data ▾	Describe ▾	Graph ▾	Analyze ▾	Export ▾	Send T
	123	⚠	⚠	⚠	⚠	⚠		
	RuleID	Attr1	Attr2	Attr3	Tag			
1	1	X	<ANY>	Bye	A			
2	2	<ANY>	2	Hello	B			
3	3	Y	<ANY>	Bye	C			
4	4	X	44	<ANY>	D			
5	5	X	44	Hello	E			
6	6	<ANY>	<ANY>	<ANY>	F			

- Apply first appropriate rule to populate “Tag”

A real life example

- Traditional Data Step...
 - Implicit loop is useful
 - iterate through each row in the TAG_INPUT table and look up appropriate rule
 - Challenging to do rule application
 - <ANY>
 - Only apply one rule per tag
- PROC SQL...
 - Rule application a lot easier in SQL
 - No easy way to loop – have to use a macro
 - The code would look something like this...

```

PROC SQL NOPRINT;
    SELECT MAX(monotonic()) INTO :RowCount FROM MyLib.Tag;
RUN;

%MACRO AssignTags;
    %DO i=1 %TO &RowCount;
        PROC SQL NOPRINT;

            SELECT  T.Attr1, T.Attr2, T.Attr3
            INTO    :Attr1, :Attr2, :Attr3
            FROM    MyLib.Tag AS T
            WHERE   monotonic() = &i;

            SELECT  R.Tag
            INTO    :Tag
            FROM    MyLib.Rules AS R
            WHERE
                R.Attr1 IN ("&Attr1", '<ANY>')
            AND
                R.Attr2 IN ("&Attr2", '<ANY>')
            AND
                R.Attr3 IN ("&Attr3", '<ANY>')
            ORDER BY
                RuleID;

            UPDATE  MyLib.Tag
            SET     Tag = "&Tag"
            WHERE   monotonic() = &i;

        RUN;
    %END;
%MEND AssignTags;

%AssignTags

```

```

]proc ds2;

data MyLib.Tag_Output (overwrite=yes);

    dcl integer CurrentTag;
    retain CurrentTag;
    drop CurrentTag, RuleID;

method init();
    this.CurrentTag = 0;
end;

```

```
method run();
```

```

SET
{
    SELECT
        T.TagID,
        R.RuleID,
        T.Attr1,
        T.Attr2,
        T.Attr3,
        R.Tag
    FROM
        MyLib.Tag_Input AS T
    INNER JOIN
        MyLib.Tagging_Rules AS R ON
        R.Attr1 IN (T.Attr1, '<ANY>')
    AND
        R.Attr2 IN (T.Attr2, '<ANY>')
    AND
        R.Attr3 IN (T.Attr3, '<ANY>')
    ORDER BY
        T.TagID,
        R.RuleID
};

```

```

if this.CurrentTag ~= TagID then do;
    this.CurrentTag = TagID;
    output;
end;

```

```
end;
```

```
enddata;
```

```
run;
```

```
quit;
```

TagID Attr1 Attr2 Attr3 Tag

1	X	44	Bye
2	1	2	Hello
3	X	44	Hello

RuleID Attr1 Attr2 Attr3 Tag

1	X	<ANY>	Bye	A	← 1
2	<ANY>	2	Hello	B	← 2
3	Y	<ANY>	Bye	C	
4	X	44	<ANY>	D	← 3
5	X	44	Hello	E	
6	<ANY>	<ANY>	<ANY>	F	

TAGID ATTR1 ATTR2 ATTR3 TAG

1	X	44	Bye	A	←
1	X	44	Bye	D	
1	X	44	Bye	F	
2	1	2	Hello	B	←
2	1	2	Hello	F	
3	X	44	Hello	D	←
3	X	44	Hello	E	
3	X	44	Hello	F	

TAGID ATTR1 ATTR2 ATTR3 TAG

1	X	44	Bye	A
2	1	2	Hello	B
3	X	44	Hello	D

Threads

- **So far we have looked at DS2 running as a PROGRAM**
 - A program runs on a single thread; processing must finish before a new program can start
- **DS2 supports *THREADS* which allow simultaneous execution of code**
- **To create a multi-threaded DS2 program:**
 - 1) Write the code that you want to run in multi-threads
 - 2) Create an instance of that code
 - 3) Execute the thread instance

Threads

```
proc ds2;
```

```
  thread t /overwrite=yes;
```

```
    dcl int x;
```

```
    method init();
```

```
      dcl int i;
```

```
      do i = 1 to 3;
```

```
        x = i;
```

```
        output;
```

```
      end;
```

```
    end;
```

```
  endthread;
```

```
run;
```

```
data ;
```

```
  dcl thread t t_instance;
```

```
  method run();
```

```
    set from t_instance threads=3;
```

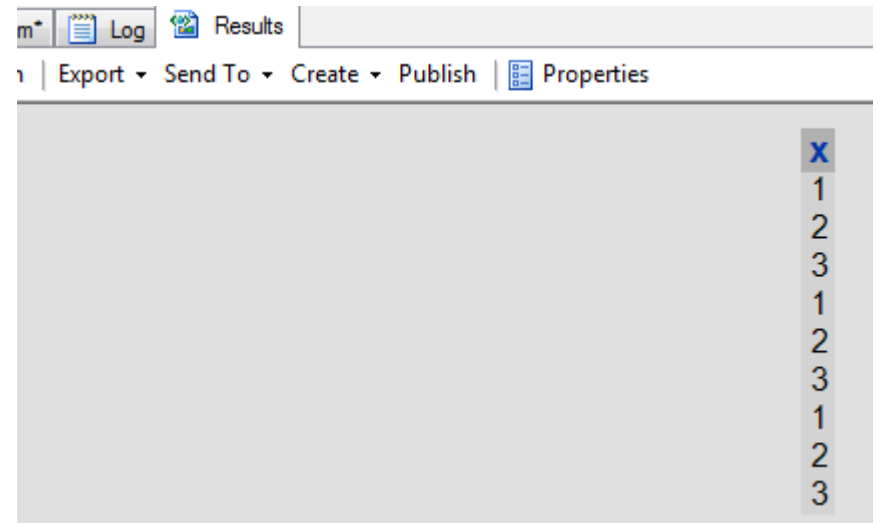
```
    put 'x= ' x ;
```

```
  end;
```

```
enddata;
```

```
run;
```

```
quit;
```



NOTE: PROCEDURE DS2 used (Total process time):

real time 1.23 seconds

The SAS System

user cpu time 0.05 seconds

system cpu time 0.05 seconds

memory 8374.68k

OS Memory 27040.00k

Timestamp 05/07/2015 04:07:20 PM

Step Count 13 Switch Count 131

Page Faults 0

Page Reclaims 3077

Page Swaps 0

Voluntary Context Switches 7626

Involuntary Context Switches 61

Block Input Operations 0

Block Output Operations 8

Single Threaded program log



3 Threads log →

NOTE: PROCEDURE DS2 used (Total process time):

real time 0.53 seconds

user cpu time 0.05 seconds

system cpu time 0.05 seconds

memory 9033.00k

OS Memory 27824.00k

3

The SAS System

Timestamp 05/07/2015 04:07:20 PM

Step Count 14 Switch Count 140

Page Faults 0

Page Reclaims 3603

Page Swaps 0

Voluntary Context Switches 7412

Involuntary Context Switches 12

Block Input Operations 0

Block Output Operations 144

Threads

- **Threads can take parameters, just like user-defined methods**
 - IN_OUT and RETURNS not allowed – all params must be by value

```
thread work.t (integer n, char(100) name) /overwrite=yes;
    decl char(110) str;
    keep str;

    method init();
        do i = 1 to n;
            str = i || name;
            output;
        end;
    end;
endthread;
run;
```

- Use “SETPARMS” to pass parameters into the THREAD
 - Must be done before the SET FROM statement

Threads

```
proc ds2;  
  thread work.t (integer n, char(100) name) /overwrite=yes;  
    dcl char(110) str;  
    keep str;  
  
    method init();  
      do i = 1 to n;  
        str = i || name;  
        output;  
      end;  
    end;  
  endthread;  
run;  
  
data;  
  dcl thread work.t t;  
  
  method init();  
    t.setparms(4, 'Richard');  
  end;  
  
  method run();  
    set from t threads=2;  
  end;  
enddata;  
run;
```

```
quit;
```

Program* Log Results

Refresh Export Send To Create Publish Properties

str

- 1Richard
- 2Richard
- 3Richard
- 4Richard
- 1Richard
- 2Richard
- 3Richard
- 4Richard

Packages

- **Traditional programming languages have “libraries” to encourage code re-use**
 - Can be user-defined (e.g. calculator) or predefined (e.g. string manipulation)
 - Encapsulate tested business rules and functionality in one place
- **The SAS DS2 equivalent of a library is the PACKAGE**
 - Logically related methods and variables that can be shared and re-used in DS2 programs and threads.
 - To use objects inside the package, you create an INSTANCE of the package, then use DOT NOTATION to call the method
 - SAS DS2 also has the concept of predefined and user-defined packages

Predefined Packages

- **FCMP (SAS Function Compiler) Package**
 - Supports calls to FCMP functions and subroutines from within DS2
- **Hash Package**
 - Provides standard functionality for quick data storage and retrieval, based on unique lookup keys
- **Hash Iterator Package**
 - For searching data based on unique lookup keys
- **HTTP Package**
 - Constructs HTTP clients to access web servers/REST
- **Logger Package**
 - Standard methods for logging to files. Configured using XML.
- **Matrix Package**
 - To use the matrix data structure (2 dimensional array)
- **SQLSTMT Package**
 - For using FedSQL (SAS Federated Query Language, ANSI SQL 1999)

Packages

- To use a package in your DS2 program, you must create an instance of the package
 - Two ways: Immediate and Deferred

- **Immediate instantiation**

```
DECLARE PACKAGE hash h();
```

- “hash” is the name of the package.
- “h” is the instance of the package
- The parentheses indicate immediate instantiation
- We can now use dot notation to call methods inside h
 - e.g. h.find([k1]) – find key k1 in the hash table h

- **Deferred instantiation**

```
DECLARE PACKAGE hash h();
```

```
.....  
h = _NEW_ hash();
```

Packages

- **Scope of packages – same rules as variables**
 - If the package is declared within a method, then it is local to that method
 - If declared outside a method, it is global to the program/package/thread
- Can use `_THIS_` keyword to change scope

```
DECLARE PACKAGE hash h();
```

```
.....
```

```
h = _NEW_ hash();    /* h is local */
```

```
h = _NEW_ THIS hash(); /* h is global */
```


Packages

- **Example of using the MATRIX pre-defined package**
 - Reads table x using SET statement into variable array a
 - The “IN” method in the MATRIX package called to load matrix with array

```
proc ds2;
    vararray double a[4];
    dcl package matrix m;

    /* Create an empty matrix to hold the input values */
    method init();
        m = _new_ [this] matrix(4, 4);
        i = 1;
    end;

    /* Read and initialize each row of matrix from vararray a */
    method run();
        set x;
        m.in(a, i);
        i + 1;
    end;
quit;
```

Packages

- **Example of using the SQLSTMT pre-defined package**
 - Can provide a template SQL statement which is resolved dynamically at run-time

```
proc ds2;  
  dcl package sqlstmt s('insert into td.testdata (x, y, z) values (?, ?, ?)', [x y z]);  
  
  do i=1 to 5;  
    x=i;  
    y=i*1.1;  
    z=i*10.01;  
    s.execute();  
  end;  
end;  
quit;
```

- The following rows are inserted into the table td.testdata:

1	1.1	10.01
2	2.2	20.02
3	3.3	30.03
4	4.4	40.04
5	5.5	50.05

Packages

- Comparison of the SQLSTMT package with the SQLEXEC function

SQLSTMT Package	SQLEXEC Function
applicable when FedSQL statements are executed multiple times	applicable when a FedSQL statement is executed only once
allocates, prepares, executes, and frees a FedSQL statement dynamically at run time	allocates, prepares, executes, and frees a FedSQL statement dynamically at run time
supports the passing of parameters	does not support the passing of parameters
produces a result set	does not produce a result set
supports run-time SELECT query generation	cannot be used with a SELECT statement
similar to the Java Database Connectivity (JDBC) PreparedStatement class	similar to the SQL EXECUTE IMMEDIATE statement or the JDBC Statement.executeUpdate(String) method

User-defined Packages

- To create your own (user-defined) packages, you use the **PACKAGE...ENDPACKAGE** block construct

```
proc ds2;
  PACKAGE convert / overwrite = yes;
    method C_to_F(INTEGER C) returns DOUBLE;
      /* convert degrees fahrenheit to degrees celsius */
      return 32 + (C * (9. / 5.));
    end;
  ENDPACKAGE;

  DATA measures (overwrite=YES);
    declare package convert cnv();
    method init();
      type = 'C';
      do C = 0. to 100.;
        F = cnv.C_to_F(C);
        output;
      end;
    end;
  ENDDATA;
quit;
```

User-defined Packages

- **Constructor**
 - Method with same name as package
 - Executes once on instantiation of the package by either:
 - `DECLARE PACKAGE`
 - `_NEW_` keyword
 - Can pass parameters
 - Can not return any value
- **Destructor**
 - Method with the name “DELETE”
 - Executes once when package instantiation is going out of scope
 - Does not take parameters
 - Can not return any value

User-defined Packages

```
|proc ds2;
  PACKAGE test / overwrite = yes;
  dcl integer x;

  method test(char(10) name, integer reps);
    this.x = reps;
    put 'Constructor - hello' name;
  end;

  method write(char(100) msg);
    dcl integer i;
    do i=1 to this.x;
      put i ':' msg;
    end;
  end;

  method delete();
    put 'Destructor';
  end;
ENDPACKAGE;

data _null_;
  method run();
    declare package test tst('Richard', 5);
    tst.write('Hello World');
  end;
enddata;
run;
quit;
```

```
49          enddata;
50          run;
Constructor - hello Richard
1 : Hello World
2 : Hello World
3 : Hello World
4 : Hello World
5 : Hello World
Destructor
```

2

NOTE: Created package test in data

NOTE: Execution succeeded. No rows

SAS In-Database Code Accelerator

- Enables you to run a DS2 thread in parallel on a database
- The following databases are supported, and you must have configured SAS/ACCESS interface:
 - Greenplum
 - Hadoop (MapReduce/Hive)
 - Teradata
- To use, must specify one of the following
 - PROC DS2 DS2ACCEL=YES;
 - DS2ACCEL system option = ANY:

```
options ds2accel=any;
```

```
libname teralib teradata server=terapin database=xxxxxx  
user=xxxxxx password=xxxxxx;
```

- Can then access libname using DOT notation

```
data teralib.indata;  
  do i = 1 to 10;  
    output;  
  end;  
run;
```

Summary of DS2

- **It's not a huge leap from traditional Data Step to DS2**
 - Much of DS2 is very similar to DATA Step, so you can leverage what you already know.
 - Some of the core features that DS2 shares with the DATA step include:
 - SAS statements such as
 - SET, BY, RETAIN, PUT, OUTPUT, DO, IF/THEN/ELSE
 - the implicit loop of the SET statement
 - the implicit declaration of global variables
 - ability to call SAS functions and formats
- **DS2 extends the traditional Data Step**
 - support for ANSI SQL data types such as TIMESTAMP and VARCHAR
 - provides variable scoping
 - variable arrays
 - enabling parallel execution of entire programs through threading
 - adding user-defined methods and packages
 - NULL

DS2 Gotcha's

- **INPUT Statement – Not supported**

- Programs that read raw data (e.g. from a file) and create data tables are not candidates for DS2

- **MERGE Statement – Not supported**

- MERGE is still a reserved word in DS2 but its functionality is not available
- Used to merge more than one dataset into one
 - Will have to “hand-code” this functionality

- **Data Type conversions**

- Base SAS tables only support fixed length character and double precision numeric variables
- When moving between tables that can use ANSI 1999 data types and Base SAS tables, data type conversion will occur
 - There are many conversion rules – see SAS documentation

References

- **An Introduction to SAS Data Steps**
Social Science Computing Cooperative
- **An Introduction to DS2**
Shaun Kaufmann
- **I Object: SAS Does Objects with DS2**
Peter Eberhardt
- **SAS® 9.4 DS2 Language Reference**
SAS Institute, Cary NC.
- **Contact**
 - Richard Superfine (richard_superfine@uhc.com)