A Brief Introduction To Some Object-Oriented Programming (OOP) Concepts For SAS Programmers
Andra Northup, Advanced Analytic Designs, Inc., Davis, California

Abstract
DS2, a significant alternative to the DATA Step, introduces an object-oriented programming environment. Many capable, experienced SAS programmers have not had the opportunity to learn and use object-oriented programming which may seem completely foreign, both conceptually and in terminology. This paper introduces and provides DS2 examples of some basic OOP concepts such as Encapsulation, Method, Packages, Object, Block, Overloading, and Instantiation, to provide grounding for further exploration of DS2.

Introduction
The focus of this paper is on concepts essential to a basic understanding of DS2, particularly those that are unfamiliar even to experienced SAS programmers. Many of these are components of object-oriented Programming (OOP).

Why Become Familiar with OOP?
Procedural languages, such as FORTRAN, Cobol, and C, use a “Top Down” or functional decomposition design approach, similar to Base SAS, focusing on procedures that operate on data. This approach has been described as “task-centric” analogous to focusing on the linguistic component of verbs.

In object-oriented languages, such as java, perl and C#, data and related procedures are bundled together into “objects”. This approach has been described as “data-centric” and analogous to focusing on the linguistic component of nouns.

Modularity, code reuse and ease of debugging are some of the benefits recounted for OOP. Also, object-oriented programming allows multiple teams of developers to work on the same project easily, and object-oriented languages can help the developer manage the code.

OOP has been criticized as not meeting its stated goals of reusability and modularity, and overemphasizing one aspect of software design and modeling (data/objects) at the expense of other important aspects (computation/algorithms). Additional complaints include thickly layered programs that destroy transparency, difficulty following execution flow, and the need to have packages and libraries installed for proper functioning. There is recognition, however, that in large, complex systems OOP can provide advantages including increased efficiency.

Regardless of one’s position on the question, there is no doubt that basic knowledge of OOP serves one well in understanding the modern information landscape and languages in current use.

Why Use DS2?
The core features of the DATA Step include the implicit loop of the SET statement, reading and writing data set observations, implicit global variable declaration, access to a large library of SAS functions, and the ability to use system or user-defined formats. DS2 shares the core features of the DATA step and in addition offers variable scoping, user-defined methods, ANSI SQL data types, user-defined packages, programming structure elements, and the ability to insert SQL directly into the SET statement.

DS2 was designed for data manipulation and data modeling applications that can achieve increased efficiency by running code in threads. One of the key principles of performing speedy analytics on big data is to split the data across multiple processors and disks, to send the code to the distributed processors and disks, have the code run on each processor against its sub-set of data, and to collate the results back at the point from which the request was originally made. This approach has been described as sending code to the data rather than pulling the data to the code to utilize the speed of sending a few dozen lines of code to many processors rather than pulling many millions of rows of data to one (big) processor. Of course, performance is also dependent on hardware architecture and the amount of effort you put into the tuning of your architecture and code.

Although with DS2 there are many potential benefits, inevitably there is some downside to any tool. For example, DS2 will still perform type conversions but the rules are more complicated because DS2 introduces
so many different types. Also, DS2 does not respect the SASHELP library. If you reference SASHELP (on a SET statement, for example) there will be an error message that the "schema name SASHELP was not found". The current implementation of DS2 cannot be used to read raw data and create data tables.

There are differences in DATA step and DS2 data-handling that could influence your choice of environment. For example, the DATA step supports only missing values, and has no concept of a null value. In contrast, DS2 supports both missing and null values. Nulls from a database can be processed in ANSI mode or in SAS mode.

DS2 supports the SQL style date and time conventions that are used in other data sources. Date and time values with a data type of DATE, TIME, and TIMESTAMP can be converted to a SAS date, time, or datetime value, but DS2 cannot convert a SAS date, time, or datetime value to a value having a DATE, TIME, or TIMESTAMP data type.

DS2 is particularly suited for the programs/applications that:

- require the precision that new supported data types offer
- benefit from using the new expressions, or write methods or packages
- can capitalize on the ability to use SQL within a SET statement
- can take advantage of the large overlaps with the abilities of the macro language, but with the advantage of using one coherent language, with many different types of data available (not just character).
- need to execute SAS FedSQL from within the DS2 program (SAS FedSQL is a SAS proprietary implementation of ANSI SQL:1999 core standard. FedSQL is a vendor-neutral SQL dialect that provides a common SQL syntax across all data sources. You can embed and execute FedSQL statements from within your DS2 programs. Proc FEDSQL enables you to submit FedSQL language statements from a Base SAS session.)
- execute outside a SAS session, e.g. on High-Performance Analytics Server or the SAS Federation Server
- take advantage of threaded processing in products such as the SAS In-Database Code Accelerator, SAS High-Performance Analytics Server, and SAS Enterprise Miner
- profit from increased efficiency by defining threads to use the processing power of a Massively Parallel Processing (MPP) environment.
- can use SAS in-Database Code Accelerator if Greenblum or Teradata available

In determining whether to use DATA Step or DS2 to develop a program/application, weigh the advantages of features offered by DS2 against the additional complexity of creating and maintaining DS2 programs.

A word on rules and terminology...

DS2 uses the terms “row”, “column”, and “table”, which correspond to the SAS DATA step terminology “observation”, “variable”, and “data set”.

Variables in DS2 are 1-256 characters in length and follow the naming convention similar to DATA step variables. The properties of DS2 variables are name, scope and data type. Variable names are called “identifiers” in DS2, as are the names of other DS2 programming language entities, such as methods, packages, and arrays, as well as the names of tables and columns.

A variable declaration, either explicit or implicit, allocates memory for the variable, identifies that memory with an identifier, and designates the type of data that can be saved at that memory location. The DECLARE statement can be used to specify scalar variables (numeric, character, date, or time data types) and temporary arrays. In DS2, the DECLARE statement is also used for package and thread declarations.

More than one variable and/or array can be specified in a DECLARE statement. For example, the following DECLARE statement specifies two scalar variables named x and y and two temporary arrays named a and b, all having a data type of DOUBLE.

```
declare double a[10] x y b[20];
```

DECLARE and DCL are equivalent. Thus, the above statement could also be coded as
If you use a variable without declaring it, DS2 assigns the variable a data type (implicit declaration). The data type for an undeclared variable on the left side of an assignment statement is determined by the data type of the value on the right side of the assignment statement.

The myriad rules and exceptions of DS2, important though they are, are beyond the scope of this paper and focusing on them is potentially counterproductive to acquiring a conceptual overview. The reader is encouraged to use the information here as a jumping off point providing a groundwork for exploration of the power and complexity of DS2.

And now for some basic concepts...

What Is an Object?
Objects are structures that contain both data (state, attributes) and procedures (behavior, methods).

Software objects are like real-world objects which also have state (data) and behavior (procedures). Cats have state (name, color, breed, hungry) and behavior (purring, eating, playing with yarn). Cars also have state (type of transmission, mileage, current speed) and behavior (increasing speed, turning, applying brakes). Identifying the state and behavior for real-world objects is a way to begin thinking in terms of object-oriented programming.

Each object is said to be an instance of a particular template called a package (for example, an object with the variable name set to "Mary" might be an instance of the package "Employees").

Objects are created by calling a special type of code (method) known as a constructor. A program may create many instances of the same package as it runs.

After you create an instance of a package, dot notation is used to access a method of the package instance, as the following example shows.

All in a cat's day
Fluffy is a cat. During a typical day, he does various actions: he eats, sleeps, etc. Here's how some object-oriented code might look.

```
Package Cat;
Fluffy = _NEW_ Cat();
Fluffy.eats();
Fluffy.runs();
Fluffy.sleeps();
```

A package can be thought of as a special function which creates instances of an object, as well as the template for the object.

The connection between the methods with the object is indicated by dot notation, i.e. a "dot" ("."), written between them.

What Does Instantiate Mean?
In object-oriented programming (OOP) language to instantiate an object is to create an instance or occurrence of the object. An instantiated object is given a name and is constructed using the structure described within a package. An object can be instantiated in a package, a thread program or a data program. As noted above, the constructor is the code used to instantiate an object. It looks like a method. You call the constructor by using the keyword _NEW_ followed by the name of the class and any necessary parameters. Examples of instantiation are included in the discussion of the concept of package.

What Is Scope?
The concept of scope defines where in a program a variable can be accessed. The DATA step does not have a concept of scope. All variables are global, i.e. known to all of the code within the DATA step.
In DS2, a variable can be “global” - known to all of the code within the DS2 program, or “local” to a particular program structure. (Peter Eberhardt and Xue Yao in their 2015 paper point out the analogous use of %local and % global variables in SAS macro functions.) As the program structures of Blocks, Methods, Packages, and Threads are discussed below, scope will be addressed for each.

Although sometimes confusing, it is possible for variables within the same program to have the same name and data type, as long as they have different scope. Examples of this are shown below in the discussion of method scope.

What Is a Block?

A block is a group of program statements enclosed between a DATA, PACKAGE, or THREAD statement and its concluding END statement:

- DATA...ENDDATA
- PACKAGE...ENDPACKAGE
- THREAD...ENDTHREAD

Each DS2 program must have one and only one program block statement. The program block can contain other statements, and defines the scope of identifiers within that block.

The general structure of a DS2 data program is created by the DATA...ENDDATA statements containing a global declaration list and a METHOD statement list.

Similarly, a thread program would consist of a global declaration list and a METHOD statement list contained between the THREAD...ENDTHREAD statements. The structure of a thread program is essentially the same as that of a data program, but is used to execute several threads in parallel.

A package also consists of a global declaration list and a METHOD statement list contained within a programming block created by the PACKAGE…ENDPACKAGE statements. A package is compiled and stored for later use by a data program, a thread program, or another package. When you declare the package in a DS2 data program, thread program or in another package, the stored package is loaded into memory. You can then access the methods and variables in the package.

<table>
<thead>
<tr>
<th>Keywords</th>
<th>Creates</th>
<th>Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA...ENDDATA</td>
<td>data program</td>
<td>RUN()</td>
</tr>
<tr>
<td>THREAD...ENDTHREAD</td>
<td>thread program</td>
<td>Loaded into memory when referenced in a DECLARE statement in another data program or package. Used to execute threads in parallel in one or more operating system threads when referenced in SET FROM statement in a subsequent data program</td>
</tr>
<tr>
<td>PACKAGE…ENDPACKAGE</td>
<td>a collection of variables and methods that can be called by a data program, a thread program, or another package</td>
<td>Compiled and stored for later use. Loaded into memory when referenced in a DECLARE statement in a data program, thread program or another package, and the methods and variables in the loaded package are then accessible.</td>
</tr>
</tbody>
</table>

Table 1 - Comparison of Programming Blocks

Program Subblock Statements

There are two statements that create program subblocks:

- DO...END
- METHOD...END

A DS2 program normally contains several subblocks of programming statements. Each subblock contains two sections: a section of global declaration statements followed by a section of other local statements.
What Is a Method?

In general, DS2 methods are like functions, procedures, or subroutines. DS2 methods can exist only as a named subblock of code within a data program, thread program, or package. Because executable code can reside only in methods, methods are the structural building blocks of DS2 programs. Method names have global scope within the enclosing program block.

As noted above, a method programming subblock begins with the METHOD keyword and methodname and ends with the END keyword, as the following generalized example shows.

```
METHOD methodname();
    ...DS2 variables and statements...
END;
```

There are two types of methods:

- **System Methods** -
  - INIT( ) - Automatically executes one time, as the first method of a program.
  - RUN( ) - Automatically executes after INIT( ) completes; functional equivalent of DATA step.
  - TERM( ) - Automatically executes one time, as the last method of a program.

  System methods cannot be overloaded. There is also one optional system method that is used only with threads: SETPARMS.

- **User-defined Methods** are the methods that you create or that someone else created for reuse. Unlike system methods, which automatically run, user-defined methods must be called. You can call a user-defined method from wherever the method is in scope, and you can do it as many times as needed.

Every DS2 program will contain, either implicitly or explicitly, the three system methods above. If you do not define any one of these methods in a DS2 program, the DS2 compiler will create an empty version of it (like those below). These methods are meant to provide a more structured framework than the SAS DATA Step implicit loop concept. In Base SAS, the entire DATA Step program is included in the implicit loop. In DS2, the implicit loop is represented by the RUN method, with the INIT and TERM methods providing initialization and finalization code, respectively.

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

**Method Scope**

Each method creates a method scope. Variables that are declared at the top of the programming subblock have local scope. Local variables are not included in the Program Data Vector and exist only for the duration of the method call.

Method parameters are only in scope within the method. Their value is not “visible” outside of the method. This is an example of variables within the same program having the same name and data type but different scope. Global variables a and b have the values specified in the system method RUN(). The value of method parameters (local variables) a and b within the method swap_vars is different and not “visible” outside the method as shown by the log output.

```
proc ds2;
    /* data program */
data;
    declare double a b;
    method swap_vars(double a, double b);
        dcl double temp;
        temp=a;
        a=b;
        b=temp;
        put 'Inside Method: a= ' a ' b= ' b;
    end;
```
A Brief Introduction To Some Object-Oriented Programming (OOP) Concepts For SAS Programmers, continued

```sas
method run();
    a=1;
    b=25;
    put 'Before Method Call: a= ' a ' b= ' b;
    swap_vars(a,b);
    put 'After Method Call: a= ' a ' b= ' b;
end;
enddata;
run;
quit;
```

Output 1. Output written to log from execution of Method swap_vars

Variables declared within a method are local in scope to that method, just like parameters passed in by value. This is another example of variables within the same program having the same name and data type but different scope. As the log output indicates, the value of global variable a is unchanged despite assignment of a different value to local variable a.

```sas
proc ds2;
/* data program */
data;
    declare double a;     /* global scope */
    method my_method();
        dcl double a;    /* local scope */
        a=25;
        put 'Inside Method: a= ' a;
    end;
    method run();
        a=1;
        put 'Before Method Call: a= ' a;
        my_method();
        put 'After Method Call: a= ' a;
    end;
enddata;
run;
quit;
```

Output 2. Output written to log from execution of Method my_method

You can access global variables of the same name as a local variable within a method by using the keyword "THIS". Note how in the example below the value of the global variable is accessed and changed.

```sas
proc ds2;
/* data program */
data;
    declare double a;     /* global scope */
    method my_method();
        dcl double a;    /* local scope */
        a=77.7;
        put 'Local Scope: a = ' a;    /* 77.7 */
    ```
A Brief Introduction To Some Object-Oriented Programming (OOP) Concepts For SAS Programmers, continued

```
put 'Global Scope: a = ' this.a; /* 100.1 */
this.a = 98.6; /* new value assigned to global */
end;
method run();
a=100.1;
my_method();
put 'New Global Scope: a = ' a; /* 98.6 */
end;
enddata;
run;
quit;
```

Output 3. Output written to log from execution of Method my_method

Below is an example of the scope of variables when initialized by the system method INIT().

```
proc ds2;
/* data program */
data;
    declare int x; /* x is a global variable */
    method init();
        dcl int z; /* z is a local variable */
        x = 1;
        y = 2; /* y is undeclared so a global variable */
        z = x + y;
        this.total = x + y + z; /* using "this" declares total as global */
    end;
enddata;
run;
quit;
```

- The global declaration list at the top of the data program contains variable x. Because x is a global variable, it can be referenced within the method block.
- Because it is declared within the method block, z is a local variable.
- Though variable y is in the method subblock, it defaults to a global variable because it is undeclared (implicitly declaration).
- The variable total is a global variable because of the use of the keyword “THIS”.

What Is Overloading a Method?

Using more than one version of a method with the same name is overloading a method. Where there are multiple methods with the same name, the method used is the one having the same type and number of arguments passed in. Thus, methods within a program or package can have the same name if they have different parameter lists. In contrast, you cannot declare more than one method with the same name having the same number and type of arguments, because the compiler cannot tell them apart.

The following example shows an overloaded method, d, with unique argument signatures:

```
proc ds2;
data _null_;  
    method d(double x, double y, double z) returns double;  
        return x + y + z;
    end;
```
A Brief Introduction To Some Object-Oriented Programming (OOP) Concepts For SAS Programmers, continued

method d(int x, int y, int z) returns int;
    return x + y + z + 500;
end;

method run();
    dcl double a;
    dcl int b;
    a = d(100.1, 100.2, 100.3);
    put 'a = ' a;
    b = d(1, 2, 3);
    put 'b = ' b;
end;
enddata;
run;
quit;

When input of d(1,2,3) is specified, the second method whose signature requires values with INTEGER data types is utilized and the following line is written to the SAS log:

b=  506

Output 2. Output written to log from execution of Method d utilizing argument of 3 integers

When input of d(100.1, 100.2, 100.3) is specified, the first method whose signature requires values with DOUBLE data types is utilized and the following line is written to the SAS log:

a=  300.6

Output 3. Output written to log from execution of Method d utilizing argument of 3 double-precision, floating-point numbers

In the example below, DS2 cannot determine whether the values in the method expression have a data type of INTEGER or DOUBLE. Because it is ambiguous, DS2 issues an error:

proc ds2;
data _null_;  

    method d(double x, int y) returns double;
        return x + y;
    end;

    method d(int x, double y) returns double;
        return x + y;
    end;

    method run();
        z = d(100, 102);
        put 'z = ' z;
    end;
enddata;
run;
quit;

ERROR: Compilation error.
ERROR: Ambiguous method call for method d. Multiple method declarations match the given argument list.

Output 3. Error message written to log from ambiguous method overloading
What Is a Package?

A DS2 package is not a program, but rather a template/pattern used to define named objects, their variables and methods, that can be stored and reused by other DS2 programs. In essence the package is creating a data type. (A package is similar to a “class” in other OOP languages, although some OOP languages, such as Java utilize a “package” concept in addition to “class”.)

The main benefit of a package derives from the reusability of a set of useful methods. There are two types of packages:

- Predefined Packages
  - FCMP - supports calls to FCMP functions and subroutines
  - Hash and hash iterator - enables storage, search & retrieval of data based on unique lookup keys
  - HTTP - constructs an HTTP client to access HTTP web services
  - Logger - provides a basic interface (open, write, and level query) to the SAS logging facility
  - Matrix - provides matrix programming capability, an implementation of SAS/IML functionality
  - SQLSTMT - can pass FedSQL statements to DBMS for execution & access the result set returned

- User-defined Packages you create or that someone else created for reuse.

A user-defined package must be compiled and stored before it can be used. It is defined by a programming block in a DS2 procedure starting with the PACKAGE keyword and packagename and ending with the ENDPACKAGE keyword, as the following generalized example shows.

```
PACKAGE packagename;
...DS2 variables, statements, and methods...
ENDPACKAGE;
```

Once a package has been compiled and stored it can be used in a DS2 program, another package, or a thread to construct instances of the object(s) it defined. The program, package or thread can then access the object's methods. A package variable is used to reference a particular instance of the package.

Package Constructors and Destructors

Constructors and destructors are special package methods that are used during construction and destruction of package instances.

The constructor method

- initializes a newly constructed package instance
- has the same name as the package
- does not have return types and does not return values.

The destructor method

- performs cleanup, releases resources held by the package instance before the package instance is destroyed, or both.
- is the DELETE method.
- does not have return types and does not return values.

You can create an instance of a package with a DECLARE statement and a statement using the _NEW_ operator:

```
declare package package-name instance-name;
instance-name = _new_ package-name();
```

Alternatively, you can use the condensed constructor syntax of a DECLARE statement with constructor arguments:

```
declare package package-name instance-name();
```
When a package is declared, a variable is created that can reference an instance of the package. If constructor arguments are provided with the package variable declaration, then a package instance is constructed and the package variable is set to reference the constructed package instance.

Multiple package variables can be created and multiple package instances can be constructed with a single DECLARE PACKAGE statement, and each package instance represents a completely separate copy of the package.

**Note:** If you declare a package variable without simultaneously declaring an instance of the package, the value of the package variable is NULL.

DS2 automatically calls a package’s constructor when an instance of the package is constructed. As noted above, an instance of a package can be constructed either with
- the DECLARE PACKAGE statement with constructor arguments
- the DECLARE PACKAGE statement without constructor arguments and the _NEW_ operator

Thus, the following statement

```
declare package complex c();
```

is equivalent to the following two statements where the DECLARE PACKAGE statement tells SAS that c is a variable of type COMPLEX package. The _NEW_ operator constructs an instance of the COMPLEX package and assigns it to the package variable c.

```
declare package complex c;
c = _new_ complex();
```

DS2 automatically calls a package’s destructor when a package instance goes out of scope and is destroyed.

Note that creating a null package variable with a DECLARE PACKAGE statement does not result in DS2 calling the package’s constructor. A package’s constructor is called only when a package instance is constructed with either a DECLARE PACKAGE statement with constructor arguments or with a _NEW_ operator.

When you create methods in a user-defined package those methods are stored in the package. These packages can be thought of as libraries of your methods. Any type of method can be saved in a package. Once you have stored methods in a package (within the PACKAGE ... ENDPACKAGE statements), you can access them by creating an instance of the package with either a DECLARE statement with constructor arguments or with the _NEW_ operator.

Here is an example of a very simple user-defined package called MATH. It contains a method that adds two numbers.

```
proc ds2;
/* MATH User-defined package that sums two floating-point doubles */
package sas_ds2.math/overwrite=yes;
    method add(double x, double y) returns double;
        return x+y;
    end;
endpackage;
run;
quit;
```

**NOTE:** Created package math in data set sas_ds2.math.

**NOTE:** Execution succeeded. No rows affected.

Output 4. Log output confirming package created and saved

In the next example, two numbers are added by using the ADD method in the MATH package that was created in the previous example. First, the MATH package is declared and instantiated. Then the ADD method
is called and the result is assigned to SUM. Note the use of dot notation to access a method of the package instance (highlighted).

```sas
proc ds2;
/* data program */
data _null_;  
dcl double sum;  
method init();  
/* declare instance 'f' of package MATH and use package method ADD */
dcl package sas_ds2.math f();  
sum = f.add(2,3);  
put 'sum= ' sum;  
end;  
enddata;  
run;  
quid;
```

Output 5. Log output when saved package declared and used in data program

Because packages are not programs, they do not have system methods that run automatically. Thus, to execute a package method, your program must call it.

**Protection from Overwriting**

Unlike Base SAS, DS2 protects existing tables from being overwritten. However, if you are developing or changing a package, you need the ability to overwrite the package. To overwrite an existing package, use the OVERWRITE=YES table option (highlighted), as seen in a previous example.

```sas
package sas_ds2.math/overwrite=yes;
...
endpackage;
```

**Packages and Scope**

The scope of packages follows the same rules as the scope of variables:

- If declared within a method, it is local to the method
- If declared outside a method, it is global to the data program/package/thread block containing it

A package instance is deleted automatically when execution exits the scope in which it was created. Thus, the lifetime of a package instance depends on the scope in which it is created. An instance created in a method is created in the local scope of that method. Since local to a method, the instance is deleted when execution exits the method scope in which the instance was created.

If a method needs to return a package instance, then the instance needs to be created outside the scope of the method. Otherwise, it would be deleted prematurely when the method returns.

**Global Scope using keyword THIS**

Alternatively, using the THIS keyword as an argument in the new operator (_NEW_) allows you to create the instance in the method subblock in global scope rather than the default method scope. (Note that brackets are required in the DS2 syntax for use of keyword [THIS] in this context.)

- A package instance created in a data program with the _NEW_ operator and the THIS scope keyword is created in program global scope. The lifetime of the package instance is the entire program.
- A package instance created in a package block with the _NEW_ operator and the THIS scope keyword is created in package global scope. Package global scope is limited to the lifetime of a specific instance of the package. When the package instance is deleted, all package instances in the instance’s global scope are also deleted.
• The third type of global scope is thread global scope. A package instance created with the THIS scope keyword in a thread program is in thread global scope. Thread global scope is limited to the lifetime of a specific thread. When the thread exits, all package instances in the thread's global scope are automatically deleted.

This example uses the _NEW_ operator with the keyword [THIS] to construct an instance of the package mypkg in the global scope of the data program.

```
package mypkg/overwrite=yes;
   method m(double x) returns double;
      return x+99;
   end;
endpackage;

data _null_;
   dcl package mypkg p;
   method m();
      p = _new_ [this] mypkg();
   end;
enddata;
```

It is important to note the difference between the scope of the package variable and the scope of the package instance. In the following example, the variable p is declared local to the method m and the instance is global to the entire program.

```
data _null_;  
   method m();
      dcl package mypkg p;
      p = _new_ [this] mypkg();
   end;

   method init();
      m();
      /* instance of mypkg created in global */
      /* scope and therefore still exists. It is */
      /* a dead reference because the instance */
      /* is not referenced by any package variable */
      /* and therefore is inaccessible. */
   end;
enddata;
run;
```

If the instance is not explicitly deleted or reassigned to a variable with global scope, then the instance becomes a dead reference after the method m completes.

**What Is a Thread?**

A thread is a program block enclosed between a THREAD statement and its concluding ENDTHREAD statement.

Typically, DS2 code runs sequentially. That is, one process runs to completion before the next process begins. However, it is possible to run more than one process concurrently, using threaded processing. In threaded processing, each concurrently executing section of code is said to be running in a thread. DS2 threading works well both on a machine with multiple cores and within a massively parallel processing (MPP) database.

Running as a program versus running as a thread impacts input and output:

When a DS2 program runs as a sequentially processed program,

- Input data can include both rows from database tables and rows from DS2 program threads.
- Output data can be both database tables and rows that are returned to the client application.
When a DS2 program runs as a thread,

- Input data can include only rows from database tables, not other threads.
- Output data includes the rows that are returned to the DS2 program that started the thread.

To enable DS2 code to run in threads:
1. Create the thread by enclosing your DS2 code between THREAD...ENDTHREAD statements.
2. Create one or more instances of the thread in a DS2 program by using a DECLARE THREAD statement.
3. Execute the thread or threads by using a SET FROM statement.

In this example, a very simple thread, t, is created by using the THREAD statement.

```
proc ds2;
    thread t;
    dcl int x;
    method init();
        dcl int i;
        do i = 1 to 3;
            x = i;
            output;
        end;
    end;
    endthread;
run;
quit;
```

**NOTE:** Created thread t in data set work.t.
**NOTE:** Execution succeeded. No rows affected.

Output 6. Log output confirming thread created and saved

In this DS2 program, an instance of t is declared, and two threads are executed, using the SET FROM statement in the RUN method. Each of the two threads generates three observations for x for a total of six observations in the output table.

```
data;
    dcl thread t t_instance;
    method run();
        set from t_instance threads=2;
        put 'x= ' x ;
    end;
enddata;
```

```
x=  1
x=  2
x=  3
x=  1
x=  2
x=  3
```

Output 7. Log Output from DS2 program where two threads executed using SET FROM statement in RUN method.

When running DS2 programs utilizing thread processing, the order of SAS log output might vary due to the order in which threads are processed.

**What Is Encapsulation?**

Encapsulation is:
- the packing of data and functions into a single component.
- the implementation details are hidden from users to prevent external interference.
DS2 methods and packages allow you to write packages that approximate the encapsulation and abstraction of object-oriented classes.

The power steering mechanism of a car provides an analogy. Power steering is a complex system, with internal components tightly coupled together working synchronously to turn the car in the desired direction, even controlling the power delivered by the engine to the steering wheel. Externally, though, there is only one interface available, and the rest of the complexity is hidden. Moreover, the steering unit in itself is complete and independent. It does not affect the functioning of any other mechanism.

Similarly, the concept of encapsulation can be applied to code. Encapsulated code should have following characteristics:

- Everyone knows how to access it.
- It can be easily used regardless of implementation details.
- There shouldn’t any side effects of the code, to the rest of the application.

The Overall Flow of a DS2 (OOP) Program

The DS2 procedure enables you to submit DS2 language statements from a Base SAS session.

1. A PROC DS2 statement specifies that the subsequent input is DS2 language statements
2. Program Block opening statement: DATA, PACKAGE or THREAD.
3. Global variables, if any, are declared
4. The INIT method runs (explicitly or implicitly)
5. Variables in the program data vector which have not been retained will be set to the appropriate missing values.
6. The RUN method executes (implicitly or explicitly). The RUN method is where the implicit loop exists. RUN executes until all input tables are completely read.
7. Execution control then depends on the status of any input statement in the RUN method. Currently, the only input statement in DS2 is the SET statement. If the RUN method meets one of these conditions,
   a. No input statements
   b. An input statement that has completed execution

    then processing proceeds to Step 8. Otherwise, processing proceeds to Step 5 so that the RUN method can execute again.
8. The TERM method executes (implicitly or explicitly). Final processing is performed.
9. The RUN statement following submits the DS2 statements. The RUN statement is required. SAS reads the program statements that are associated with one task until it reaches a RUN statement.
10. The QUIT statement stops the procedure

Examples

Example 1: Writing a Message to the Log with DATA ... ENDDATA

```sas
proc ds2;
data _null_;        /*_NULL_ on the DS2 DATA statement indicates that there is no automatic output generated*/
   method init();
   dcl varchar(16) str;
   str = 'Hello World!';
   put str;         /*The DS2 PUT statement writes to the SAS log*/
end;
enddata;
run;
quit;             /*The QUIT statement stops the procedure*/
```
A Brief Introduction To Some Object-Oriented Programming (OOP) Concepts For SAS Programmers, continued

Example 2: Creating an Output Table with DATA ... ENDDATA.

Example 3: Using DATA ... ENDDATA When Not Creating a Table

Using the keyword _NULL_ as the table name causes the DS2 program to execute without writing rows to a table. This example writes to the log the value of IDnumber and SATscore for each row. An output table is not created.
run;
quit;

Output 10. Output written to log. No table written

Example 4: Using DATA ... ENDDATA to Create a SAS Data Set

```sas
libname MyFiles base 'C:\Myfiles';
proc ds2;
data MyFiles.BaseTable;
    declare double j j2;
    method run();
    do j = 1 to 1000;
        j2 = 2*j;
        output;
    end;
end;
enddata;
run;
quit;
```


```sas
proc print data=myfiles.basetable (obs=10);
run;
```
Example 5: Using Methods

In the following example, methods CONCAT and ADD are defined and then invoked in the INIT() method.

```sas
proc ds2;
  data;
    dcl char(200) r;
    method concat(char(100) x, char(100) y) returns char(200);
      return trim(x)|| y;
    end;
    method add(double x, double y) returns double;
      return x + y;
    end;
    method init();
      r = concat('abc', 'def');
      d = add(100,101);
      output;
      put 'r = ' r;
      put 'd = ' d;
    end;
  enddata;
run;
quit;
```

Using OUTPUT without arguments causes the current row to be written to all tables that are named in the DATA statement or thread program. If no tables are specified in the DATA statement or thread program, then the row is written to the client application.

Output 12. Output written to client application.

```
  r       d
  abcdef  201
```

Output 13. Output written to log.

Hash Objects

The SAS hash object, part of version 9, first introduced the object-oriented programming paradigm into the SAS DATA step language. Rather than perform operations directly on data set variables, or utilize functions and call routines, the hash object, a complex data type, is operated on via its attributes and methods accessed using dot notation.

If you are not familiar with the hash object yet, this might be a good place to get your first taste of OOP. Listed in the Recommended Reading section below are resources discussing the use of hash objects.

CONCLUSION

As the amount of data collected and the speed and power with which it can be processed rapidly increase, new tools and environments are evolving. A high-level overview of OOP concepts used in SAS DS2 is provided here. The reader is encouraged to explore further the new tools and environments available.
REFERENCES

A Brief Introduction To Some Object-Oriented Programming (OOP) Concepts For SAS Programmers, continued

- SAS® 9.4 DS2 Language Reference, Fourth Edition
- SAS(R) 9.4 FedSQL Language Reference, Fourth Edition

ACKNOWLEDGMENTS
Many thanks to Kirk Lafler, Susan Slaughter and Mary Northup for their feedback and support in the writing of this paper.

RECOMMENDED READING ABOUT SAS HASH OBJECTS


CONTACT INFORMATION
Your comments and questions are valued and encouraged. Contact the author at:

Andra Northup
Advanced Analytic Designs, Inc.
P.O. Box 72892, Davis, CA 95617
530.400.0554
andra@advancedanalyticdesigns.com
www.advancedanalyticdesigns.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.