Object-Oriented Programming With SAS® Via PROC DS2

Author: Dari Mazloom, USAA

ABSTRACT

PROC DS2 is a new programming feature of SAS® programming language which introduces concepts of Object-Oriented principles to SAS programming. DS2 is included with BASE SAS® language 9.4. DS2 provides new data types which make possible to have better precision for data and better interaction with external databases.
This paper concentrates on how SAS® programmers can use PROC DS2 to write Object-Oriented programs; it also illustrates what PROC DS2 is, why use PROC DS2, and basic programming instructions(syntax) associated with the construction and use of PROC DS2.

INTRODUCTION

SAS® has introduced new capabilities via PROC DS2 to write programs employing Object-Oriented programming concepts. The OO techniques can be applied to programs via a variable named package which is built in PROC DS2.
In the following topics, we will discuss the basic constructs of PROC DS2 and how SAS® programmers can write Object-Oriented programs using packages.
Please note:

- Package and class are used interchangeably.
- An object refers to an instance of package or class.
- DS2 does NOT support inheritance.

This paper covers the following topics:

- Why use PROC DS2?
- Data types in PROC DS2
- The constructs of PROC DS2
- User-defined methods in a DS2 program
- What is Object-oriented programming?
- What are the principles of Object-oriented programming?
- What is a package / class in Object-oriented programming?
- What is an object in Object-oriented programming?
- How to define a package / class in PROC DS2
- Accomplishing Inheritance by embedding the super class inside sub-classes
- Variable Scope

This paper does NOT cover the following topic(s):

- Interaction with external databases
- Threads

TOPIC 1: WHY USE PROC DS2?

PROC DS2 extends the power of DATA step into OO programming. It provides an extensive list of data types which makes the SAS® data compatible with other databases.

TOPIC 2: DATA TYPES IN PROC DS2

PROC DS2 provides an extensive list of data types which extends the current SAS® data types.

The new data types facilitate a seamless interaction among SAS® and external databases.

The new data types are: BIGINT, BINARY(n), CHAR(n), DATE, DECIMAL|NUMERIC(p,s), DOUBLE, FLOAT, INTEGER, NCHAR(n), NVARCHAR(n), REAL, SMALLINT, TIME(p), TIMESTAMP(p), TINYINT, VARBINARY(n), VARCHAR(n).

For details of the new data types, please consult: What Are the Data Types?

http://support.sas.com/documentation/cdl/en/ds2ref/69739/HTML/default/viewer.htm#n0v130wmh3hmuzn1t7y5y4pgxa69.htm

TOPIC 3: THE CONSTRUCTS OF A DS2 PROGRAM

A DS2 program is comprised of a list of variable declarations together with a list of methods.

Example 1: Let's look at and analyze a simple DS2 program.

```
PROC DS2 LIBS=WORK;
     DATA WRK_T1 (OVERWRITE=YES);
        /*The scope of the following data elements is the current*/
        /*PROC DS2*/
           DCL VARCHAR(20) MAKE_NAME;
           DCL VARCHAR(20) MODEL_NAME;
           DCL INT COUNTER;

           /*The INIT method is run automatically once at the*/
           /*beginning of the program to perform all initialization*/
           METHOD INIT();
                PUT 'METHOD=INIT';
                COUNTER = 0;
                MAKE_NAME = 'FORD';
                MODEL_NAME = 'F-150';
           END;

           /*The RUN method is run automatically after the INIT*/
           /*method.*/
           METHOD RUN();
```

```
            PUT 'METHOD=RUN';
            COUNTER = COUNTER + 1;
            PUT MAKE_NAME=;
            PUT MODEL_NAME=;
            OUTPUT WRK_T1;
        END;

        /*The TERM method is run automatically once at the end of*/
        /*the process and it can perform house cleaning tasks. */

        METHOD TERM();
            PUT 'METHOD=TERM';
        END;
    ENDDATA;
RUN;
QUIT;
```

TOPIC 4: USER-DEFINED METHODS IN A DS2 PROGRAM

In addition to the three DS2 system-defined methods (INIT, RUN, TERM), DS2 allows the Programmers to define their own methods called User-Defined methods.

Important note: All user-defined methods need to be defined and placed prior to the placement of methods, INIT, RUN, and TERM.

Example 1: The following is a DS2 program that contains user-defined methods.

```
PROC DS2 LIBS=WORK;
    DATA WRK_T1 (OVERWRITE=YES);
        DCL VARCHAR(20) MAKE_NAME;
        DCL VARCHAR(20) MODEL_NAME;
        DCL INT COUNTER;

        /*User-defined method: Assigns a value to MAKE_NAME*/
        METHOD SetMakeName(VARCHAR(20) MakeName);
            MAKE_NAME = MakeName;
        End;

        /*User-defined method: Returns the value of MAKE_NAME to
        /the calling section*/
        METHOD GetMakeName() RETURNS VARCHAR(20);
            RETURN MAKE_NAME;
        End;

        /*User-defined method: Assigns a value to MODEL_NAME*/
        METHOD SetModelName(VARCHAR(20) ModelName);
            MODEL_NAME = ModelName;
        End;

        /*User-defined method: Returns the value of MODEL_NAME to*/
        /*the calling section*/
```

```
            METHOD GetModelName() RETURNS VARCHAR(20);
                  RETURN MODEL_NAME;
            End;

            METHOD INIT();
                  PUT 'METHOD=INIT';
                  COUNTER = 0;
                  MAKE_NAME='';
                  MODEL_NAME='';
            END;

            METHOD RUN ();
                  DCL VARCHAR(20) TempMakeName;
                  DCL VARCHAR(20) TempModelName;

                  PUT 'METHOD=RUN';

                  SetMakeName('FORD');
                  SetModelName('F-150');
                  TempMakeName = GetMakeName();
                  TempModelName = GetModelName();
                  PUT TempMakeName=;
                  PUT TempModelName=;
                  COUNTER = COUNTER + 1;
                  PUT COUNTER=;
                  OUTPUT WRK_T1;
            END;

            METHOD TERM();
                  PUT 'METHOD=TERM';
            END;
      ENDDATA;
RUN;
QUIT;
```

TOPIC 5: WHAT IS OBJECT-ORIENTED PROGRAMMING?

A type of program design in which both data and methods of a data structure are defined. The data is the common information that all instances of the structure have in common; such as name, address, etc. The methods are actions which can be applied to the data; such as compute taxes, compute discounts, etc.

TOPIC 6: WHAT IS A CLASS IN OBJECT-ORIENTED PROGRAMMING?

In OO programming language, a class is a “blue print” which holds the relative information or detail of all the objects or instances of the class. A class defines all data and Methods or functions which can act upon the data. Example: VEHICLE class.

TOPIC 7: WHAT IS AN OBJECT IN OBJECT-ORIENTED PROGRAMMING?

In OO programming language, an object is a physical manifestation or instance of a class.
Example: A Car; A table.

TOPIC 8: WHAT ARE THE PRINCIPLES OF OBJECT-ORIENTED PROGRAMMING?

There ae four principles in Object-Oriented programming:

Encapsulation – Hides the internal specifications of an object. The consumers of the object are prohibited from directly accessing or modifying the internal data of the object. The object provides public accessors(getters) to expose its data and it also provides public modifiers(setters) to modify its data.

Abstraction – Hides the detail implementation of an object and it only exposes essential properties and methods of an object. One example of abstraction is an interface or an API.

Polymorphism – Indicates same name but different behavior. Multiple methods or functions can have the same name, but they can have different implementations. The act of redefining a method is called method overload.
Example: In the animal kingdom, all animals move. However, their movement is accomplished in different manners. A pony hops, a fish swims, and a cat walks.

Inheritance – A way of re-using code of existing objects by creating sub-classes of an existing class. DS2 does NOT support inheritance. However, the concept of inheritance can be “sort of” accomplished By embedding the super class in the sub-classes.

TOPIC 9: HOW TO DEFINE A PACKAGE IN PROC DS2?

A user-defined package provides a collection of methods and variables that belong to a common object. A package is comprised of data elements, system-defined methods, and user-defined methods. One user-defined method that must be created is called a CONSTRUCTOR.
What is a constructor? A constructor is the first method called to instantiate a package.
The name of the constructor is the same as the name of the package. A constructor can be overloaded multiple ways. However, all versions of a constructor have the same name but different signatures.

What is a method signature? A method’s signature is comprised of the name of the method together with its parameters and its return type (if any).

The following is a list of two methods that share the same name, but different signatures. The method Salary is now overloaded. This process makes the method ComputeDealerCost a polymorphic method.

```
METHOD ComputeDealerCost () RETURNS DOUBLE;
      RETURN (MSRP – (MSRP * 0.30));
End;

METHOD ComputeDealerCost (DOUBLE CostFactor) RETURNS DOUBLE;
      RETURN (MSRP – (MSRP  *  CostFactor));
End;
```

Example: The following creates a package named VEHICLE with its methods and variables.

```
PROC DS2 LIBS=WORK;
      PACKAGE VEHICLE / OVERWRITE=YES;
            DCL VARCHAR(20) MAKE_NAME;
            DCL VARCHAR(20) MODEL_NAME;
            DCL CHAR(20) ENGINE_SIZE;
            DCL DOUBLE MSRP;

            METHOD SetMakeName(VARCHAR(20) MakeNm);
                  MAKE_NAME = MakeNm;
            END;

            METHOD GetMakeName() RETURNS VARCHAR;
                  RETURN (MAKE_NAME);
            END;

            METHOD SetModelName(VARCHAR(20) ModelNm);
                  MODEL_NAME = ModelNm;
            END;

            METHOD GetModelName() RETURNS VARCHAR;
                  RETURN (MODEL_NAME);
            END;

            METHOD SetEngineSize(CHAR(20) EngineSize);
                  ENGINE_SIZE = EngineSize;
            END;

            METHOD GetEngineSize() RETURNS CHAR(20);
                  RETURN (ENGINE_SIZE);
            END;

            METHOD SetMSRP(DOUBLE MSRPAmt);
                  MSRP = MSRPAmt;
            END;

            METHOD GetMSRP() RETURNS DOUBLE;
                  RETURN (MSRP);
```

```
            END;

            METHOD CalcDealerCost() RETURNS DOUBLE;
                  DCL Double DealerCost;
                  DealerCost = GetMSRP() - (GetMSRP() * 0.30);
                  RETURN (DealerCost);
            END;

            METHOD CalcDealerCost(DOUBLE CostFactor) RETURNS DOUBLE;
                  RETURN (GetMSRP() - (GetMSRP() * CostFactor));
            END;

            /*Default Constructor*/
            METHOD VEHICLE();
                  PUT 'VEHICLE - Default Constructor.';
                  MSRP = 50000.00;
            END;

            /*Overloaded constructor*/
            METHOD VEHICLE(VARCHAR(20) MakeNm, VARCHAR(20) ModelNm,
                        CHAR(20) EngineSize);
                  PUT 'VEHICLE - Constructor Overloaded.';

            /*Calling the default constructor to instantiate the
            /*package*/
                  VEHICLE();
                  SetMakeName(MakeNm);
                  SetModelName(ModelNm);
                  SetEngineSize(EngineSize);

            END;

            /*The DELETE method is NOT required*/
            METHOD DELETE();
                  PUT 'VEHICLE - DELETE EXECUTED';
            END;

      ENDPACKAGE;      /*End of VEHICLE*/

RUN;
QUIT;

/*Let's create an instance of the package VEHICLE*/
PROC DS2 LIBS=WORK;
      DATA WORK.DARI_DS2_1 (OVERWRITE=YES);

            DCL PACKAGE VEHICLE VehicleObj('FORD', 'F-150',
                                          '600 Cubic Inches');

            /*The following variables have a local scope.*/
            DCL VARCHAR(20) MakeNm;
            DCL VARCHAR(20) ModelNm;
```

```
            DCL CHAR(20) EngineSize;
            DCL DOUBLE MSRP;
            DCL DOUBLE DealerCost1;
            DCL DOUBLE DealerCost2;

            METHOD RUN();
                  MakeNm = VehicleObj.GetMakeName();
                  ModelNm = VehicleObj.GetModelName();
                  EngineSize = VehicleObj.GetEngineSize();
                  MSRP = VehicleObj.GetMSRP();
                  DealerCost1 = VehicleObj.CalcDealerCost();
                  DealerCost2 = VehicleObj.CalcDealerCost(0.20);

                  PUT MakeNm=;
                  PUT ModelNm=;
                  PUT EngineSize=;
                  PUT MSRP=;
                  PUT DealerCost1=;
                  PUT DealerCost2=;
                  PUT;
            END;
      ENDDATA;
RUN;
QUIT;

The following is the result:

VEHICLE - Constructor Overloaded.
VEHICLE - Default Constructor.
MakeNm=FORD
ModelNm=F-150
EngineSize=600 Cubic Inches
MSRP=50000
DealerCost1=35000
DealerCost2=40000
VEHICLE - DELETE EXECUTED
```

TOPIC 10: ACCOMPLISHING INHERITANCE BY EMBEDDING THE SUPER CLASS INSIDE SUB-CLASSES

DS2 does not implement the concept of inheritance. However, inheritance can be accomplished by embedding the super class as a data member inside the sub-classes.

Example: The following defines three packages, VEHICLE, TRUCK, and CAR. All three packages have make, model, Engine size, and MSRP in common. Therefore, these variables will be defined in the super class VEHICLE..
The sub-classes will instantiate the super class as a data member. All common methods in sub-classes will refer to the corresponding methods in the super class.

Example: The following defines three packages, VEHICLE, TRUCK, and CAR.
The TRUCK and CAR packages each have a data member whose type is VEHICLE.

```
PROC DS2 LIBS=WORK;
      PACKAGE VEHICLE / OVERWRITE=YES;
```

```
DCL VARCHAR(20) MAKE_NAME;
DCL VARCHAR(20) MODEL_NAME;
DCL CHAR(20) ENGINE_SIZE;
DCL DOUBLE MSRP;

METHOD SetMakeName(VARCHAR(20) MakeNm);
     MAKE_NAME = MakeNm;
END;

METHOD GetMakeName() RETURNS VARCHAR;
     RETURN (MAKE_NAME);
END;

METHOD SetModelName(VARCHAR(20) ModelNm);
     MODEL_NAME = ModelNm;
END;

METHOD GetModelName() RETURNS VARCHAR;
     RETURN (MODEL_NAME);
END;

METHOD SetEngineSize(CHAR(20) EngineSize);
     ENGINE_SIZE = EngineSize;
END;

METHOD GetEngineSize() RETURNS CHAR(20);
     RETURN (ENGINE_SIZE);
END;

METHOD SetMSRP(DOUBLE MSRPAmt);
     MSRP = MSRPAmt;
END;

METHOD GetMSRP() RETURNS DOUBLE;
     RETURN (MSRP);
END;

METHOD CalcDealerCost() RETURNS DOUBLE;
     DCL Double DealerCost;
     DealerCost = GetMSRP() - (GetMSRP() * 0.30);
     RETURN (DealerCost);
END;

METHOD CalcDealerCost(DOUBLE CostFactor) RETURNS DOUBLE;
     RETURN (GetMSRP() - (GetMSRP() * CostFactor));
END;

/*Default Constructor*/
METHOD VEHICLE();
     PUT 'VEHICLE - Default Constructor.';
     MSRP = 50000.00;
END;
```

```
      /*Overloaded constructor*/
      METHOD VEHICLE(VARCHAR(20) MakeNm, VARCHAR(20) ModelNm,
                     CHAR(20) EngineSize);
            PUT 'VEHICLE - Constructor Overloaded.';

            /*Calling the default constructor to instantiate the*/
            /*package*/
            VEHICLE();
            SetMakeName(MakeNm);
            SetModelName(ModelNm);
            SetEngineSize(EngineSize);

      END;

      /*The DELETE method is NOT required*/
      METHOD DELETE();
            PUT 'VEHICLE - DELETE EXECUTED';
      END;

ENDPACKAGE;      /*End of VEHICLE*/

PACKAGE TRUCK / OVERWRITE=YES;
      DCL PACKAGE VEHICLE VehicleObj;
      DCL VARCHAR(20) BED_SIZE;

      METHOD SetMakeName(VARCHAR(20) MakeNm);
            VehicleObj.SetMakeName(MakeNm);
      END;

      METHOD GetMakeName() RETURNS VARCHAR;
            RETURN (VehicleObj.GetMakeName());
      END;

      METHOD SetModelName(VARCHAR(20) ModelNm);
            VehicleObj.SetModelName(ModelNm);
      END;

      METHOD GetModelName() RETURNS VARCHAR;
            RETURN (VehicleObj.GetModelName());
      END;

      METHOD SetEngineSize(CHAR(20) EngineSize);
            VehicleObj.SetEngineSize(EngineSize);
      END;

      METHOD GetEngineSize() RETURNS CHAR(20);
            RETURN (VehicleObj.GetEngineSize());
      END;

      METHOD SetMSRP(DOUBLE MSRPAmt);
            VehicleObj.SetMSRP(MSRPAmt);
```

```
      END;

      METHOD GetMSRP() RETURNS DOUBLE;
            RETURN (VehicleObj.GetMSRP());
      END;

      METHOD SetBedSize(VARCHAR(20) BedSize);
            BED_SIZE = BedSize;
      END;

      METHOD GetBedSize() RETURNS VARCHAR(20);
            RETURN (BED_SIZE);
      END;

      METHOD CalcDealerCost() RETURNS DOUBLE;
            RETURN (VehicleObj.CalcDealerCost());
      END;

      METHOD CalcDealerCost(DOUBLE CostFactor) RETURNS DOUBLE;
            RETURN (VehicleObj.CalcDealerCost(CostFactor));
      END;

      /*TRUCK Default Constructor*/
      METHOD TRUCK();
            PUT 'Truck - Default Constructor.';
            VehicleObj = _NEW_ [this] VEHICLE();
            VehicleObj.SetMSRP(50000.00);
            BED_SIZE = '6 FEET';
      END;

      /*TRUCK Overloaded constructor*/
      METHOD TRUCK(VARCHAR(20) MakeNm, VARCHAR(20) ModelNm,
                   CHAR(20) EngineSize);
            PUT 'TRUCK - Constructor Overloaded.';

            /*Calling the default constructor to instantiate the*/
           /*package*/
            Truck();
            SetMakeName(MakeNm);
            SetModelName(ModelNm);
            SetEngineSize(EngineSize);

      END;

      /*The DELETE method is NOT required*/
      METHOD DELETE();
            PUT 'TRUCK DELETE EXECUTED';
      END;

ENDPACKAGE;      /*End of TRUCK*/

PACKAGE CAR / OVERWRITE=YES;
```

```
DCL PACKAGE VEHICLE VehicleObj;
DCL VARCHAR(20) TRUNK_SIZE;

METHOD SetMakeName(VARCHAR(20) MakeNm);
     VehicleObj.SetMakeName(MakeNm);
END;

METHOD GetMakeName() RETURNS VARCHAR;
     RETURN (VehicleObj.GetMakeName());
END;

METHOD SetModelName(VARCHAR(20) ModelNm);
     VehicleObj.SetModelName(ModelNm);
END;

METHOD GetModelName() RETURNS VARCHAR;
     RETURN (VehicleObj.GetModelName());
END;

METHOD SetEngineSize(CHAR(20) EngineSize);
     VehicleObj.SetEngineSize(EngineSize);
END;

METHOD GetEngineSize() RETURNS CHAR(20);
     RETURN (VehicleObj.GetEngineSize());
END;

METHOD SetMSRP(DOUBLE MSRPAmt);
     VehicleObj.SetMSRP(MSRPAmt);
END;

METHOD GetMSRP() RETURNS DOUBLE;
     RETURN (VehicleObj.GetMSRP());
END;

METHOD SetTrunkSize(VARCHAR(20) TrunkSize);
     TRUNK_SIZE = TrunkSize;
END;

METHOD GetTrunkSize() RETURNS VARCHAR(20);
     RETURN (TRUNK_SIZE);
END;

METHOD CalcDealerCost() RETURNS DOUBLE;
     RETURN (VehicleObj.CalcDealerCost());
END;

METHOD CalcDealerCost(DOUBLE CostFactor) RETURNS DOUBLE;
     RETURN (VehicleObj.CalcDealerCost(CostFactor));
END;
```

```
            /*CAR Default Constructor*/
            METHOD CAR();
                  PUT 'CAR - Default Constructor.';
                  VehicleObj = _NEW_ [this] VEHICLE();
                  VehicleObj.SetMSRP(40000.00);
                  TRUNK_SIZE = '10 Cubic Ft';
            END;

            /*CAR Overloaded constructor*/
            METHOD CAR(VARCHAR(20) MakeNm, VARCHAR(20) ModelNm,
                        CHAR(20) EngineSize);
                  PUT 'CAR - Constructor Overloaded.';

                  /*Calling the default constructor to instantiate the*/
                  /*package*/
                  CAR();
                  SetMakeName(MakeNm);
                  SetModelName(ModelNm);
                  SetEngineSize(EngineSize);

            END;

            /*The DELETE method is NOT required*/
            METHOD DELETE();
                  PUT 'CAR DELETE EXECUTED';
            END;

      ENDPACKAGE;      /*End of CAR*/

RUN;
QUIT;

/*Let's create an instance of the package Vehicle*/
PROC DS2 LIBS=WORK;
      DATA WORK.DARI_DS2_1 (OVERWRITE=YES);
            DCL PACKAGE TRUCK TruckObj('FORD', 'F-150',
                                       '600 Cubic Inches');

            DCL PACKAGE CAR CarObj('Toyota', 'Camry', '218 Cubic Ft');

            /*The following variables have a local scope.*/
            DCL VARCHAR(20) MakeNm;
            DCL VARCHAR(20) ModelNm;
            DCL CHAR(20) EngineSize;
            DCL DOUBLE MSRP;
            DCL VARCHAR(20) BedSize;
            DCL VARCHAR(20) TrunkSize;
            DCL DOUBLE DealerCost1;
            DCL DOUBLE DealerCost2;

            METHOD RUN();
                  MakeNm = TruckObj.GetMakeName();
```

```
                    ModelNm = TruckObj.GetModelName();
                    EngineSize = TruckObj.GetEngineSize();
                    MSRP = TruckObj.GetMSRP();
                    BedSize = TruckObj.GetBedSize();
                    DealerCost1 = TruckObj.CalcDealerCost();
                    DealerCost2 = TruckObj.CalcDealerCost(0.20);

                    PUT MakeNm=;
                    PUT ModelNm=;
                    PUT EngineSize=;
                    PUT MSRP=;
                    PUT BedSize=;
                    PUT DealerCost1=;
                    PUT DealerCost2=;
                    PUT;

                    MakeNm = CarObj.GetMakeName();
                    ModelNm = CarObj.GetModelName();
                    EngineSize = CarObj.GetEngineSize();
                    MSRP = CarObj.GetMSRP();
                    TrunkSize = CarObj.GetTrunkSize();
                    DealerCost1 = CarObj.CalcDealerCost();
                    DealerCost2 = CarObj.CalcDealerCost(0.20);

                    PUT MakeNm=;
                    PUT ModelNm=;
                    PUT EngineSize=;
                    PUT MSRP=;
                    PUT TrunkSize=;
                    PUT DealerCost1=;
                    PUT DealerCost2=;
                    PUT;
               END;
          ENDDATA;
RUN;
QUIT;
```

The following is the list of calls made in order of their execution:

```
TRUCK - Constructor Overloaded.
Truck - Default Constructor.
VEHICLE - Default Constructor.
CAR - Constructor Overloaded.
CAR - Default Constructor.
VEHICLE - Default Constructor.
MakeNm=FORD
ModelNm=F-150
EngineSize=600 Cubic Inches
MSRP=50000
BedSize=6 FEET
DealerCost1=35000
DealerCost2=40000
```

```
MakeNm=Toyota
ModelNm=Camry
EngineSize=218 Cubic Ft
MSRP=40000
TrunkSize=10 Cubic Ft
DealerCost1=28000
DealerCost2=32000

TRUCK DELETE EXECUTED
VEHICLE - DELETE EXECUTED
CAR DELETE EXECUTED
VEHICLE - DELETE EXECUTED

Name of the Vehicle: Tesla Model X

DEALERSHIP Destructor is called.

Vehicle Destructor is called.
```

TOPIC 11: VARIABLE SCOPE

The following was copied from: SAS® 9.4 DS2 Language Reference, Sixth Edition - SAS Support

Global Variables

A variable with global scope, a global variable, is declared in one of three ways: in the outermost programming block of a DS2 program, using a DECLARE statement, implicitly declared inside a programming block using a SET statement, or implicitly declared inside a programming block by using an undeclared variable. Variables with global scope can be accessed from anywhere in the program and exist for the duration of the program. Global variables can be used in a THIS expression in any program block.

Local Variables

A variable with local scope, a local variable, is declared within an inner programming block, such as a method or package, by using a DECLARE statement. Variables with local scope are known only within the inner programming block where they are declared.

Global and Local Variables in DS2 Output

Only global variables, by default, are included in the output. Local variables that are used for program loops and indexes do not need to be explicitly dropped from the output. Local variables are always created at the start of a method invocation, such as an iteration of the implicit loop of the RUN method, and are destroyed at the end of each invocation. Therefore, it is not recommended to use local variables as accumulator variables in the RUN method.

All global variables are named in the program data vector (PDV). The PDV is the set of values that are written to an output table when DS2 writes a row.

Example of Global and Local Variables

The following program shows both global (A, B, and TOTAL) and local variables (C):

```
PROC DS2;
     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;
QUIT;
```

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 table.

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.

CONCLUSION

DS2 language provides a powerful set of techniques to write Object-Oriented Programs.

This paper is providing just a brief introduction to the capabilities of DS2. For more details and techniques, the practitioners should consult the following references.

REFERENCES

SAS® 9.4 DS2 Language Reference, Sixth Edition - SAS Support

ABOUT THE AUTHOR

Dari Mazloom is a lead business intelligence advisor with over thirty years of practical experience in a number of programming languages such as SAS®, C#, C++, JAVA, VB, VBA, Smalltalk, COBOL, Easytrieve, JCL.

CONTACT INFORMATION

Your comments and questions are valued and encouraged.

Contact the author at:

Dari Mazloom

Dariush.Mazloom@USAA.Com

APPENDIX A: LIST OF SAS DS2 DATA TYPES

The following was copied from: SAS® 9.4 DS2 Language Reference, Sixth Edition - SAS Support

DS2 Data Types

Data Type	Description
BIGINT	stores a large signed, exact whole number, with a precision of 19 digits. The range of integers is -9,223,372,036,854,775,80 Integer data types do not store decimal values; fractional portions are discarded.
BINARY(*n*)	stores fixed-length binary data, where *n* is the maximum number of bytes to store. The maximum number of bytes is required regardless of the actual size of the value.
CHAR(*n*)	stores a fixed-length character string, where *n* is the maximum number of characters to store. The maximum number of char each value regardless of the actual size of the value. If **`char(10)`** is specified and the character string is only five characters the value is right padded with spaces.
DATE	stores 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 and a two-digit day (01 to 31). For example, the date September 24, 1975 is specified as **`1975-09-24`**. DS2 complies with ANSI SQL:1999 standards regarding dates. However, not all data sources support the full range of dates For example, dates between 0001-01-01 and 1582-12-31 are not valid dates for a SAS data set or an SPD data set.
DECIMAL|NUMERIC(*p*,*s*)	stores a signed, exact, fixed-point decimal number, with user-specified precision and scale. The precision and scale determi 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 The scale is the maximum number of digits that can be stored following the decimal point. Scale must be less than or equal For example, **`decimal(9,2)`** stores decimal numbers up to nine digits, with a two-digit fixed-point fractional portion, such a
DOUBLE	stores a signed, approximate, double-precision, floating-point number. Allows numbers of large magnitude and permits comp many digits of precision to the right of the decimal point.
FLOAT	stores a signed, approximate, double-precision, floating-point number. Data defined as FLOAT is treated the same as DOUB
INTEGER	stores a regular size signed, exact whole number, with a precision of ten digits. The range of integers is -2,147,483,648 to 2, Integer data types do not store decimal values; fractional portions are discarded. Note: Integer division by zero does not produce the same result on all operating systems. It is recommended that you avoid i
NCHAR(*n*)	stores a fixed-length character string like CHAR but uses a Unicode national character set, where *n* is the maximum number Depending on the platform, Unicode characters use either two or four bytes per character and support all international chara
NVARCHAR(*n*)	stores a varying-length character string like VARCHAR but uses a Unicode national character set, where *n* is the maximum n Depending on the platform, Unicode characters use either two or four bytes per character and can support all international c
REAL	stores a signed, approximate, single-precision, floating-point number.
SMALLINT	stores a small signed, exact whole number, with a precision of five digits. The range of integers is -32,768 to 32,767. Integer fractional portions are discarded.
TIME(*p*)	stores 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 minu second 00 to 61 (supports leap seconds), with an optional fraction value. For example, the time 6:30 a.m. is specified as **`06`** When supported by a data source, the *p* parameter specifies the seconds precision, which is an optional fraction value that i
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-di 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 sec with an optional fraction value. For example, the date and time September 24, 1975 6:30 a.m. is specified as **`1975-09-24`** When supported by a data source, the *p* parameter specifies the seconds precision, which is an optional fraction value that i
TINYINT	stores a very small signed, exact whole number, with a precision of three digits. The range of integers is -128 to 127. Integer data types do not store decimal values; fractional portions are discarded.
VARBINARY(*n*)	stores varying-length binary data, where *n* is the maximum number of bytes to store. The maximum number of bytes is not re If **`varbinary(10)`** is specified and the binary string uses only five bytes, only five bytes are stored in the column.
VARCHAR(*n*)	stores a varying-length character string, where *n* is the maximum number of characters to store. The maximum number of ch If **`varchar(10)`** is specified and the character string is only five characters long, only five characters are stored in the colum