Paper 7280-2016

Tips and Techniques for User-Defined Packages in SAS® DS2

Chris Brooks, Melrose Analytics Ltd

ABSTRACT

SAS® DS2 is a powerful new object-oriented programming language that was introduced with SAS® 9.4. Having been designed for advanced data manipulation, it enables the programmer not only to use a much wider range of data types than with the traditional Base SAS® language, but it also allows the creation of custom methods and packages. These packages are analogous to classes in other object-oriented languages such as C# or Ruby and can be used to massively improve your programming effectiveness. This paper demonstrates a number of techniques that can be used to take full advantage of DS2's object-oriented, user-defined package capabilities. Examples include creating new data types, storing and reusing packages, using simple packages as building blocks in the creation of more complex packages, a technique to overcome DS2's lack of support for inheritance, and building lightweight packages to facilitate method overloading and make parameter passing simpler.

INTRODUCTION

SAS DS2 is very much the "new kid on the block" in terms of SAS programming languages. It was released as a production product in SAS 9.4 and takes a totally different (object-oriented) view of how to approach programming than the original Data Step language or SAS Macro language.

In these traditional programming languages programs are seen as a series of logical procedures that take input data, process it and produce output data. In object-oriented programming languages (such as DS2) the emphasis is on the objects (data) that we wish to manipulate and the methods we wish to use to carry out that manipulation. In the course of working through this paper we will see how the SAS programmer can start to take advantage of these exciting capabilities to bring the world of object-oriented programming to their SAS applications.

This paper assumes a basic knowledge of DS2 syntax and concentrates on some techniques a programmer can use to utilize the full power of its user-defined packages.

THE BASICS – PACKAGES AND METHODS: WHAT ARE THEY?

In its simplest terms a package is merely a collection of variables and methods (functions) that can be used in a DS2 program. However, if we want to take full advantage of DS2's capabilities we need to treat them as being analogous to classes in other object-oriented languages such as C++, Ruby or C#. That is that the package contains both the attributes for the class and that classes methods. Methods are discrete functions which can be called to access the variables (or attributes in Object-oriented terminology) belonging to a Package (or Class) instance and perform operations on those variables.

As part of the DS2 installation SAS provides a number of pre-defined packages such as the Hash, JSON and Logger packages but it also allows you to create your own – these user-defined packages are the focus of this paper.

DEFINITIONS

It may be helpful at this point to provide some definitions for terms used in the paper which may be unfamiliar to those who haven't used an Object-Oriented language before:

Object – an object is a self-contained component which is used to represent a real-life object and which has attributes holding information about that object (also called properties or variables) and methods which perform actions on that object.

Class – A class is a blueprint or template used to build a specific type of object. An instance is a specific object built from a class and will be referenced as an object variable (or in SAS DS2 as a package variable). Examples of Classes would be Products, Stores, Orders and Machines.

Instance – A specific object created from a class; the process of creating this object is called instantiation.

Attribute – An item of information about an instance of a class, similar to a Data Set variable.

Method – A procedure or function associated with an object which performs operations on it.

Inheritance - A feature which allows new classes to take on the properties and methods of an existing class; for example, if we had a Person Class it would have attributes called name and age, these could be inherited by a Class called Customer which would inherit both the name and age attributes of the Person Class and have it's own attributes such as credit card number.

CREATE YOUR OWN DATA TYPES

In the traditional SAS Data Step language the programmer has been restricted to two data types – numeric and character. In DS2 SAS have widened this to include data types more usually associated with relational databases. We can, therefore, declare variables as floats, doubles or varchars as well as explicit date, time and timestamp variables. These and many others are available to facilitate DS2's ability to interact with relational databases systems such as Oracle and Teradata. What, however, if you are not satisfied with these inbuilt types and want to create one of your own to use in your code?

If we recall that packages are analogous to classes and objects are instances of classes and realize that a data type is in itself a class then we can do this quite simply. For instance, in SAS Data Step it is quite common to create a numeric variable to act as a counter during the implicit loop within the step. For example:

```
data _null_;
  set sashelp.class end=last;
    if _n_=1 then boycounter=0; /* 1 reference to counter */
    if sex="M" then boycounter=boycounter+1; /* 2 references to counter */
    retain boycounter; /* 1 reference to counter */
    if last then put boycounter=; /* 1 reference to counter */
run;
```

In the preceding code the counter, called boycounter, is referenced five times:

1. Once in line 3 where boycounter is initialized to zero in the first iteration of the loop.
2. Twice in line 4 where boycounter is incremented if the variable sex = "M".
3. Once in line 5 where the value of boycounter is retained to the next iteration.
4. Once in line 6 where the value of boycounter is output on the final iteration.

It is common for programmers to forget to either initialize the counter variable (line 3) or retain it (line 5) and so it would be helpful if we could find a way to remove this responsibility from the programmer. We can do this by creating a Package to manage a new data type – the Counter.

Firstly, we need to start the DS2 Procedure and issue the Package statement with the option overwrite=yes. If we do not use the overwrite option an error will be triggered the next time we run the code to generate the package:

```
Proc ds2;
  package counter/ overwrite=yes;
```

Now we can declare the package variables (attributes). We only need one for our counter and it should be an integer (we could create a counter that we could increment in fractions in which case the variable would need to be a float or a double):

```
    dcl int value;
```

We can now start to declare the methods that will operate on our counter package instance. The first is a method that has the same name as the package and will therefore automatically run whenever an instance of the counter package is declared. This is known as a constructor method. The only thing it does is initialise the value of the value variable to zero:

```
      method counter();
        value=0;
      end;
```

Our next and final method is one that is called whenever we want to increment the value of the counter. We shall call this the Add method:

```
      method add();
        value=value+1;
      end;

    endpackage;
  run;
  quit;
```

Note that at first sight there appears to be no way to retrieve the value of the counter from the package, nor is there a way to set it to a value other than zero. However, for each globally declared variable in the package there are automatically two methods that perform this function. Simply prepend get_ to the name of the variable to retrieve the value of that variable or set_ to set the value of the variable in any DS2 code outside the package. These appear to be undocumented at the time of writing but as it is not considered good practice in object oriented programming to directly access an objects attributes outside that object code then these “getter and setter” methods should be used.

Now that we have our package this is how we will use it:

```
  data class;
    set sashelp.class;
  run;

  Proc ds2;
    package counter/ overwrite=yes;
      dcl int value;

      method counter();
        value=0;
      end;

      method add();
        value=value+1;
      end;

    endpackage;
  run;

  data _null_;
    dcl package counter boycounter();
    dcl int nummales;

      method run();
        set class;
        if sex='M' then boycounter.add();
      end;

      method term();
        nummales=boycounter.get_value();
```

```
      put nummales=;
    end;
enddata;
run;
quit;
```

Firstly you will see that we have copied the file SASHELP.CLASS to the work library – this is done because DS2 doesn't recognize the SASHELP library. Having done that we create our counter package and then, in the Data Step declare an instance of it called boycounter; in the Run method we then iterate through the Class Data Set until we reach an observation where the variable Sex is equal to "M" and call the add method on the counter. This increments the value of boycounter by one and finally when all observations have been read we use the get_value method to return the value of the counter. Note that we don't need to initialise the counter nor retain it as the package automatically handles that.

Now that we have created one new data type we can use it as the foundation for others. Another common thing to want to do is create a running total of the value of a variable. We can achieve that by creating another data type very similar to the counter:

```
Proc DS2

  package summation/ overwrite=yes;

    dcl float value;


    method summation();
      value=0;
    end;

    method add(float inval);
      value=value+inval;
    end;

  endpackage;
run;
quit;
```

The only differences between the summation package and the counter package are that the running total is held as a floating point number instead of an integer and when the add method is called a value is passed to it which is added to the running total instead of incrementing it by one.

We can use this package as follows:

```
Data class;
  set sashelp.class;
run;

Proc ds2;
  package summation/ overwrite=yes;
    dcl float value;

    method summation();
      value=0;
    end;

    method add(float inval);
      value=value+inval;
    end;

  endpackage;
run;
```

```
  Data out(overwrite=yes);
    dcl package summation k();
    dcl float totweigh;

    method run();
      set class;
      k.add(weight);
    end;

    method term();
      totweigh=k.get_value();
      put totweigh=;
    end;
  enddata;
run;
quit;
```

STORE YOUR PACKAGE CODE FOR EASY RE-USE

So far we have been hard coding all of our packages within the DS2 procedure code itself. This could lead to problems over time as programs become larger and more unwieldy to use with more and more packages added to them. Also if we changed any package code we would have to go through all our programs changing the code in each of them individually. It is possible to store a package in a metadata server but if we do not have access to one we can store our code in a data set held within a SAS base library. To store the code for the counter package you would simply submit the following code:

```
libname sgfpkgs "/folders/myfolders/SASGF2016/DS2/Packages";

Proc ds2;
  package sgfpkgs.counter /overwrite=yes;
    dcl int value;

    method counter();
      value=0;
    end;

    method add();
      value=value+1;
    end;
  endpackage;
run;
quit;
```

The first of our programs using the counter package would them look like this:

```
data class;
  set sashelp.class;
run;

Proc ds2;
  dcl package sgfpkgs.counter boycounter();

  data _null_;
    dcl package counter boycounter();
    dcl int nummales;

    method run();
      set class;
```

```
      if sex='M' then boycounter.add();
    end;

    method term();
      nummales=boycounter.get_value();
      put nummales=;
    end;
  enddata;
run;
quit;
```

NO INHERITANCE - NO PROBLEM

Most object oriented languages support a concept called inheritance. This means that classes can be based on other classes in a parent-child relationship with the child class inheriting all of the attributes and methods of it's parent or parents. SAS DS2 does not support inheritance but this is not an insurmountable obstacle to using packages as inheritable objects as we can use a technique known as "composition". We use composition by embedding packages within other packages in a "has a" relationship. The package which has another package embedded into it has access to all the attributes (variables) and methods of the embedded package. Where the programmer needs to call the method of the embedded package they will create a forwarding method to enable that method to be called. The next example shows Person, Teacher and Student packages that could be used in processing school or college data.

Firstly, we will create a generic person package – all persons in any organisation will have certain basic attributes but for our purposes we only need to show three – name, age and sex:

```
Proc ds2;
  package person / overwrite=yes;
    dcl varchar(50) name;
    dcl int age;
    dcl char(1) sex;

    method person(varchar(50) iname, int iage, char(1) isex);
      name=iname;
      age=iage;
      sex=isex;
    end;
  endpackage;
run;
quit;
```

We will then create a package for teachers – in addition to name, age and sex we may want to hold data about which department they work in – this would not be needed for pupils or administrative staff. In order to make use of the variables and methods of the Person package we will embed an instance of the person package inside the teacher package but in this case we will delay its instantiation (creation):

```
Proc ds2;

     package teacher / overwrite=yes;

       dcl package person per;

       dcl varchar(25) dept;


       method teacher(varchar(50) jname, int jage, char(1) jsex, varchar(50)
jdept);

         per = _new_ [this] person(jname, jage, jsex);

         dept=jdept;

       end;
```

```
        method get_name() returns varchar(50);
          dcl varchar(50) tname;

          tname=per.get_pname();
          return tname;
        end;
/* Followed by similar forwarding methods for age and sex */
     endpackage;
run;
quit;
```

You will note that when we create an instance of the teacher package via the teacher() constructor method an instance of the person package is created via the _new_ statement, passing the creation parameters straight to the person instance which is responsible for their maintenance. Only the dept attribute is the responsibility of the teacher package. This means that when we wish to retrieve the teacher's name via the get_name method we must call the person's get_pname method to retrieve it before passing it back to the caller. We can use the package like so:

```
data _null_;
  dcl package teacher t('John Brown', 28,'M','History');
  dcl varchar(50) xname;
  dcl varchar(25) xdept;

  method run();
    xname = t.get_name();
    xdept = t.get_tdept();
    put xname=;
    put xdept=;
  end;
enddata;
```

Now that we have our Teacher package the next step is to create a Student package:

```
package student / overwrite=yes;
  dcl package person per;
  dcl int grade;

  method student(varchar(50) iname, int iage, char(1) isex, int igrade);
    per = _new_ [this] person(iname, iage, isex);
    grade=igrade;
  end;

  method get_name() returns varchar(50);
    dcl varchar(50) sname;
```

```
      sname=per.get_pname();
      return sname;
    end;
/* Followed by similar forwarding methods for age and sex */
  endpackage;
```

This time only the Grade variable is the responsibility of the Student package – everything else is handled by the Person package.

In creating an "inheritable" package we have eliminated code duplication, making program maintenance easier and more reliable.

HANDLING DATA THROUGH PACKAGES

In a real-world scenario you will almost inevitably need to read and write data to permanent files in your applications. The question we will next address is "how do we do this through user-defined packages?"

In order to answer that question, we will create a much-simplified version of a bank account handling system. The diagram below shows the package hierarchy we will use.

Figure 1 Package Hierarchy for Bank Account Example

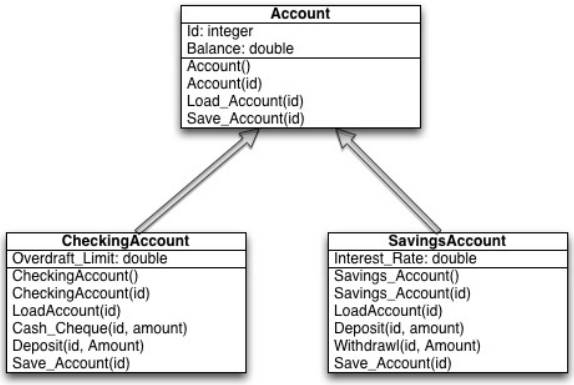

Figure 1 Package Hierarchy for Bank Account Example

The parent package is the Account package which has two child packages – CheckingAccount and SavingsAccount. The parent package has two attributes both of which are common to all types of account (Id and Balance) and four methods (two constructors, a Load_Account method and a Save_Account method). The two child packages (CheckingAccount and SavingsAccount) each have one additional attribute unique to their account type (Overdraft_Limit and Interest_Rate). In addition the child packages have methods of the same name as the parent package plus methods unique to their account type.

In order to test our packages, we will first need some data. You will often (though not necessarily) find that you have one table or Data Set per package and that is the design we will use for this example:

```
libname bankdata '/folders/myfolders/SASGF2016/data';
data bankdata.account;
  infile datalines delimiter=',';
  input id  balance;
datalines;
1,3375.55
2,3913.57
```

```
3,3843.27
4,3736.01
5,3221.02
6,3854.22
7,1846.07
8,768.37
9,3476.4
10,3981.64
;
run;

data bankdata.checkingaccount;
  infile datalines delimiter=',';
  input id  overdraft_limit;
datalines;
1,800
2,1000
3,650
4,675
5,1500
8,550
9,500
10,400
;
run;

data bankdata.savingsaccount;
  infile datalines delimiter=',';
  input id  interest_rate;
datalines;
6,2
7,2
;
run;
```

We will also need a Data Set to hold some transaction information to process:

```
data bankdata.cheques;
  infile datalines delimiter=',';
  input id amount;
datalines;
1,25
```

```
1,45.50
1,50.25
5,75
8,200.00
;
run;
```

We can now start to build our packages, starting with the Account package:

```
libname bankdata '/folders/myfolders/SASGF2016/data';
libname sgfpkgs "/folders/myfolders/SASGF2016/DS2/Packages";

Proc ds2;
  package sgfpkgs.Account / overwrite=yes;
    dcl int id;
    dcl double balance;
    forward load_account;

    method Account();

    end;

    method Account (int newid);
      id=newid;
      load_account(newid);
    end;

    method load_account(int iid);
      dcl package sqlstmt stmt1('select balance from bankdata.account where
id=?',[myid]);

      this.myid=iid;
      stmt1.execute();
      stmt1.bindresults([balance]);
      stmt1.fetch();
    end;

    method save_account(int iid);
      dcl char(100) sqltxt;

      sqltxt='update bankdata.account set balance='||balance||' where id='||iid;
      sqlexec(sqltxt);
```

```
    end;
  endpackage;
run;
quit;
```

There are several things to note in this package code:

1. It contains two constructors – an empty default constructor and a constructor where the programmer will supply an id value parameter.
2. The FORWARD statement tells the compiler that we will be referencing a method called load_account before it is defined. If we do not do this the package will fail to compile.
3. The load_account method retrieves data from the account Data Set by using the SAS supplied SQLSTMT package. The term [myid] is used to define a parameter which will be supplied prior to the SQLSTMT object's EXECUTE method.
4. The term this.myid=iid create myid as a global variable and sets it to the value of the parameter supplied to the load_account method.
5. The SQLSTMT object's EXECUTE method executes the SQL statement and returns a result set.
6. The BINDRESULTS([balance]) statement binds the package variable balance to the result set variable of the same name.
7. The SQLSTMT FETCH method then retrieves a row from the result set and completes the loading of the balance variables value.
8. The save_account method saves the current values of the id and balance variables back to the Data Set by creating an SQL string which is then executed using the SQLEXEC function.

Next we create the CheckingAccount Package:

```
Proc ds2;
 package sgfpkgs.CheckingAccount / overwrite=yes;
  dcl package sgfpkgs.Account parent_account();
  dcl double overdraft_limit;
  forward load_account;


  method CheckingAccount();


  end;


  method CheckingAccount (int newid);
   parent_account = _new_ [this] sgfpkgs.account(newid);
   load_account(newid);
  end;


  method load_account(int iid);
   dcl package sqlstmt stmt3('select overdraft_limit from Bankdata.checkingaccount
where id=?',[myid]);
```

```
    parent_account.load_account(iid);
    this.myid=iid;
    stmt3.execute();
    stmt3.bindresults([overdraft_limit]);
    stmt3.fetch();
  end;

  method cash_cheque(int iid, double iamount);
      dcl double newbalance;
      dcl double pre_balance;

      pre_balance = parent_account.get_balance();
      newbalance=pre_balance - iamount;
      if newbalance > overdraft_limit then parent_account.set_balance(newbalance);
      else put 'reject';
    end;

    method deposit(int iid, double iamount);
      dcl double newbalance;
      dcl double pre_balance;

      pre_balance=parent_account.get_balance();
      newbalance=pre_balance + iamount;
      parent_account.set_balance(newbalance);
    end;

    method save_account(int iid);
      parent_account.save_account(iid);
    end;

  endpackage;
run;
quit;
```

This package has many of the same features of the preceding Account package with the following additions:

1. The second constructor creates an instance of the Account package object – this will be the “parent” object. The load_account method is then called for the Checking Account object itself.
2. The save_account method is simply a forwarder method to ensure that the balance variable (which is the responsibility of the Account package) is maintained by that package.

Finally, we create the SavingsAccount package:

```
Proc ds2;
  package sgfpkgs.SavingsAccount / overwrite=yes;
    dcl package sgfpkgs.Account parent_account();
    dcl double interest_rate;
    forward load_account;
    method SavingsAccount();

    end;

    method SavingsAccount (int newid);
      parent_account = _new_ [this] sgfpkgs.account(newid);
      load_account(newid);
    end;

    method load_account(int iid);
      dcl package sqlstmt stmt2('select interest_rate from bankdata.savingsaccount
where id=?',[myid]);

      parent_account.load_account(iid);
      this.myid=iid;
      stmt2.execute();
      stmt2.bindresults([interest_rate]);
      stmt2.fetch();
    end;

    method deposit(int iid, double iamount);
      dcl double newbalance;
      dcl double pre_balance;

      pre_balance=parent_account.get_balance();
      newbalance=pre_balance + iamount;
      parent_account.set_balance(newbalance);
    end;

    method withdraw(int iid, double iamount);
      dcl double newbalance;
      dcl double pre_balance;
```

```
      pre_balance = parent_account.get_balance();
      newbalance=pre_balance - iamount;
      if newbalance >= 0 then parent_account.set_balance(newbalance);
      else put 'reject';
    end;

    method save_account(int iid);
      parent_account.save_account(iid);
    end;
  endpackage;
run;
quit;
```

In order to test our new package hierarchy we will use the bankdata.cheques file described earlier. We will write a SAS DS2 program to process the cheques by using the methods defined in the packages:

```
Proc ds2;
  data _null_;
    dcl package sgfpkgs.checkingaccount c();

    method run();
      set bankdata.cheques;
      c.load_account(id);
      c.cash_cheque(id,amount);
      c.save_account(id);
    end;
  enddata;
run;
quit;
```

This program simply declares an instance of the CheckingAccount package, reads the transaction file called cheques using the SET statement and then calls methods associated with the CheckingAccount package to carry out the process of updating the account to reflect the transactions. There are two things to note here:

1. At no point is the Account package instance created by the CheckingAccount directly accessed by the program.
2. The CheckingAccount is only accessed via it's methods – it's variables are never directly accessed by the program.

This approach has a number of advantages over the traditional Data Step:

1. The final program is much simpler involving only method calls after the SET statement. The

programmer needs to know nothing about the internal business logic of the system in order to write an effective program.

2. Because all of the business logic is held inside the packages if we need to change it e.g. to stipulate a minimum balance which must be held in the Savings Account we only need to change the code in that one package instead of all the programs which process that account's data.
3. Should we wish to add further account types e.g. an investment account it can similarly "inherit" much of it's functionality from the Account Package reducing development effort and facilitating code re-use.

USE LIGHTWEIGHT PACKAGES TO MAKE CODING EASIER

Most object-oriented languages have a data type called a struct which groups together a number of logically connected variables for use as a single complex data type. Normally a struct will only contain constructor, accessor and destructor methods. If any other methods are required then a full class is usually more appropriate. Although SAS DS2 does not contain an explicit struct we can mimic it by creating lightweight packages which adhere to these rules.

To demonstrate how we can use these lightweight packages we will take the case of an application designed to calculate the area of a two dimensional object. If we run the following piece of code we will have a data set containing the required dimensions for a variety of shapes:

```
data shapes;
  length shape $10;
  Input shape radius1 radius2 side;
datalines;
circle 10 . .
square . . 5
circle 15 . .
circle 20 . .
ellipse 20 15 .
circle 20 . .
circle 9 . .
;
run;
```

Figure 2 Data for Lightweight Packages Example

List Data for WORK.SHAPES

Obs	shape	radius1	radius2	side
1	circle	10	.	.
2	square	.	.	5
3	circle	15	.	.
4	circle	20	.	.
5	ellipse	20	15	.
6	circle	20	.	.
7	circle	9	.	.

Figure 2 Data for Lightweight Packages Example

In order to calculate the area of a circle we need one dimension (the radius), for an ellipse we need two dimensions (two radiuses which will be of different values) and for a square we need only one dimension (the side) as all sides will be of the same length. Each of the shape types has a different formula for calculating it's area so we could create full-scale packages for each type all containing it's own get_area method. However over time this may become burdensome as other shapes (triangles, hexagons etc) are added to the file. If we start to add three-dimensional objects (cones, cubes etc) and add other measurements (perimeter, volume etc) we may find ourselves maintaining a large number of separate packages. In order to remove this danger we decide to create a shape_calcer package to handle calculation of all measurements for all shapes.

This however presents us with an immediate problem – we would want to overload the get_area method for each shape type; however the methods for both squares and circles would both take only one parameter and it would be a float in both cases so we would have a conflict. The way to overcome this is by creating a lightweight package for each shape; and as a package is itself a data type which can be passed to a method we would be able to overload methods which have these packages as parameters. Firstly, we create the packages for the three shape types.

```
libname sgfpkgs "/folders/myfolders/SASGF2016/DS2/Packages";
Proc ds2;
  package sgfpkgs.circle / overwrite=yes;
    dcl float radius;

    method circle(float dim1);
      radius=dim1;
    end;
  endpackage;run;
```

```
quit;

Proc ds2;
  package sgfpkgs.square / overwrite=yes;
    dcl float side;

    method square(float dim3);
      side=dim3;
    end;
  endpackage;
run;
quit;

Proc ds2;
  package sgfpkgs.ellipse / overwrite=yes;
    dcl float radius1;
    dcl float radius2;

    method ellipse(float dim1, float dim2);
      radius1=dim1;
      radius2=dim2;
    end;
  endpackage;
run;
quit;
```

Notice that each package only has one explicit method – a simple constructor. We next create the shape_calcer package like so:

```
Proc ds2;
  package sgfpkgs.shape_calcer / overwrite=yes;
    dcl float area;

    method get_area(package sgfpkgs.circle c) returns float;
      area = constant('PI')*(c.radius**2);
      return area;
    end;

    method get_area(package sgfpkgs.square s) returns float;
      area=s.side**2;
      return area;
    end;
```

```
    method get_area(package sgfpkgs.ellipse e) returns float;
      area=constant('PI')*e.radius1*e.radius2;
      return area;
    end;
  endpackage;
run;
quit;
```

You will notice that the package consists of three versions of the get_area method, one for each shape type. In SAS DS2 it is possible to have multiple methods of the same name in the same package as long as their signature varies (this is called method overloading). A method's signature consists of it's list of input parameters. In the case of our shape_calcer package we will pass an instance of the shape to the method as it's only parameter so that when get_area is called the package will automatically use the correct method.

Once we have created our lightweight packages and shape_calcer package we can use it to calculate the areas of all the shapes in our file like this:

```
Proc ds2;


  data out(overwrite=yes);
  /* Declare a variable of each package type but delay creation of the instance */
    dcl package sgfpkgs.circle c;
    dcl package sgfpkgs.square s;
    dcl package sgfpkgs.ellipse e;
    dcl package sgfpkgs.shape_calcer sc();
  /* Declare a float variable to hold the calculated area */
    dcl float area;


    method run();
      set shapes;
  /* Use the shape variable to decide which package instance to create */
  /* Then call the get_area method */
      if shape='circle' then do;
        c = _new_ [this] sgfpkgs.circle(radius1);
        area=sc.get_area(c);
      end;
      else if shape='square' then do;
        s = _new_ [this] sgfpkgs.square(side);
        area=sc.get_area(s);
      end;
      else if shape='ellipse' then do;
        e = _new_ [this] sgfpkgs.ellipse(radius1, radius2);
```

```
        area=sc.get_area(e);
      end;
    end;
  enddata;
run;
quit;
```

Figure 3 Output of Lightweight Packages Example

List Data for WORK.OUT

Obs	area	shape	radius1	radius2	side
1	314.16	circle	10	.	.
2	25.00	square	.	.	5
3	706.86	circle	15	.	.
4	1256.64	circle	20	.	.
5	942.48	ellipse	20	15	.
6	1256.64	circle	20	.	.
7	254.47	circle	9	.	.

Figure 3 Output of Lightweight Packages Example

CONCLUSION

With the addition of user-defined packages to the SAS DS2 programmer's toolbox SAS have opened up a new way of designing and programming our SAS applications. If we, as SAS developers, use this facility in a true object-oriented way we can significantly improve code re-use, encourage standardization of techniques and remove much of the routine coding which is part and parcel of the traditional procedural programming paradigm.

This paper has sought to demonstrate some ways in which you can use user-defined packages but there are many others and it is hoped that it will encourage you to explore to the full all the possibilities made available by this exciting new facility.

RECOMMENDED READING

- *SAS® 9.4 DS2 Language Reference*

CONTACT INFORMATION

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

Chris Brooks
Melrose Analytics Ltd
chrisbrooks@melroseanalytics.co.uk
www.melroseanalytics.co.uk