Paper 027-2008

SAS® Talking via the Java Object Interface

Magnus Mengelbier, Limelogic Ltd, London, United Kingdom
Jan Skowronski, Limelogic Ltd, London, United Kingdom

ABSTRACT

The introduction of industry standards and the constant optimization of in-house processing are establishing the critical nature of fast, efficient, simple and integrated tools. The Java Object interface in SAS 9.1.3 opens the path for new integration possibilities, which may simplify and provide greater access to integrated systems. We consider examples of how the Java Object interface can be extended beyond simple message traffic to sending and receiving data volumes of varying size. Common Java application features are also considered as we discuss using the Java Object Interface to send and receive streams of data over multiple SAS DATA steps and programs.

INTRODUCTION

The Java Object interface introduced in SAS 9 opens the path for new integration possibilities, which show promise to greatly simplify and provide greater access to integrated systems.

A simple Hello World message is a proof that combination of SAS and Java can work, but for SAS it is more interesting to send and receive information, results, meta-data, regular data or all of the above. The interaction remains the same throughout, but as we shall see, the complexity increases with the volume of information to transfer.

Data comes in all sizes and we will consider the small to large volumes, and briefly discuss any pertinent issues. Sending and receiving data either way is a small step, but the real usefulness of the Java Object shines when SAS can integrate with Java throughout a single program.

The Java Object enhances the SAS capacity to integrate with other systems. Availability of the SAS Java Object as production in SAS for all platforms will unquestionably enhance the integration, even for SAS on a workstation.

MESSAGE

The first small step is to say Hello World, which will present the principle and technique of calling Java from within SAS with the Java object. A message is a classic example of simple communication in one direction, either requesting or sending a message.

HELLO WORLD

The first is to create a small Java class that will return our message.

```
public class helloWorld {

    /* Construct our class helloWorld */
    public helloWorld() {
    }

    public String getMessage() {
        return "Hello World";
    }

}
```

The corresponding SAS program code to access our Java class and retrieve the message waiting is fairly straightforward.

```
data _null_;
  * -- set length of variable to capture message ;
  length message $ 200 ;

  * -- initiate java object ;
  declare javaobj jobj ("helloWorld");

  * -- get message and clean up ;
  jobj.callStringMethod('getMessage', message);
  jobj.delete();

  put message = ;
run;
```

The basic principle is that the SAS variable *jobj* represents our *helloWorld* class within the DATA step. Any access to methods, attributes and information in the Java class is performed when referring to the DATA step variable *jobj*. The Java object then uses the SAS *jobj.callStringMethod()* to call the Java class method *getMessage()* in *helloWorld*. The *getMessage()* method in our Java class returns a string to SAS, which value is captured in the message variable in our DATA step. At the end, we use *jobj.delete()* to disconnect the DATA step variable *jobj* from the Java class and free up memory.

A MESSAGE IN REVERSE

The next logical step is to send a message to Java and request something back. The Java class *reversi* has two methods, *setMessage()* and *getReverseMessage()*. Our returned message echo is simply the original message in verbatim form reversed.

```
public class reversi {
    private givenMessage;

    /** Construct our class reversi */
    public reversi() {
       givenMessage = "Nothing said";
    }

    public void setMessage( String mymessage) {
       /* set message code */
    }

    public String getReverseMessage() {
        /* Method 1. Code to reverse and return stored message */
    }

    public String reverseMessage( String original ) {
        /* Method 2. Code to receive, reverse and return stored message */
    }

}  /* end of class reversi */
```

The SAS code that uses the Java class *reversi* is extremely similar to our initial helloWorld example.

```
data _null_;
  * -- set length of variable to capture message ;
  length message $ 200 ;

  * -- initiate java object ;
  declare javaobj jobj ("reversi");


  * Method 1 -- two step process;
```

```
      * -- save message to java ;
      jobj.callVoidMethod('setMessage', "This is my first message.");

      * -- get reversed message and clean up ;
      jobj.callStringMethod('getReverseMessage', message);
      put message = ;


      * Method 2 -- one step process ;
      jobj.callStringMethod('reverseMessage', "This is my second message.", message);
      put message = ;

      j.delete();

      put message = ;
    run;
```

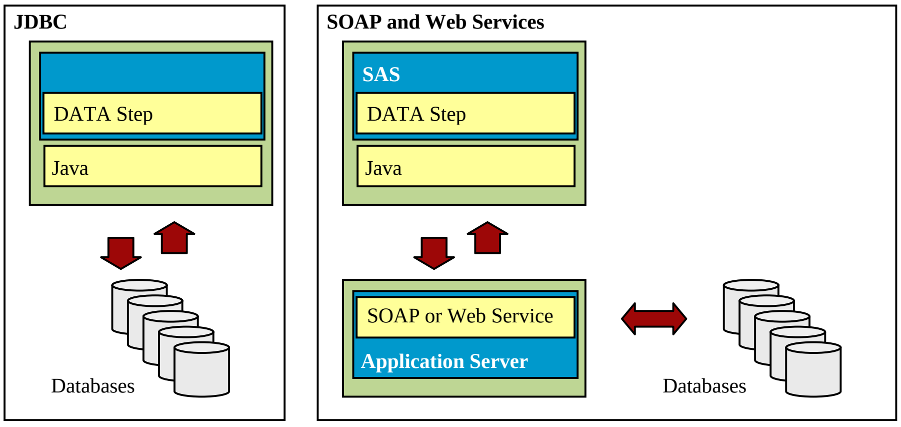
The SAS code example follows and extends the principles introduced in Hello World. Our first approach (Method 1) first saves the message text in our Java class and subsequently requests the reversed message string. Our second approach (Method 2) performs the steps of Method 1 as one single step.

Both are efficient and applicable approaches, although one may be preferred based on the processing and Java code in the background. Consider the change in complexity, if the reversing logic is performed on a server rather on the SAS client. Method 1 may be preferrable as the Java class can be designed to only connect to the server when the Java class method *getReverseMessage()* is called, hence limiting and optimising the connection performance.

The simple message examples have provided the basic principles to connect, set and retrieve a simple message with the SAS Java object, which will be developed further as we consider passing greater amounts of information than just the greeting “Hello World”.

MOVING DATA WITH JAVA

A message is a simple example of communication, but not overly useful in working with data. There are several different approaches to transfer small or big data sets between SAS clients, servers and databases. Consider three common approaches to transfer data, where each is widely used with Java and the possibility to make available the known approaches to a SAS sessions is preferable.



JDBC – Java Database Connectivity allows Java to use a database in a similar way as the SAS libname statement. Provided your Java client is allowed to access the database directly, JDBC enables you to work with the database tables and content. All the larger and smaller database vendors, with rare exceptions, provides the possibility to connect with JDBC.

SOAP – Once known as Simple Object Access Protocol, SOAP is a popular approach to transfer smaller volumes of information. This will allow a simple method of access to information stored in a database or other complex system, without explicit knowledge of how it is organised and stored, or if a direct connection to the database is not provided. This can be very beneficial if you require a specific client and source connection, but want to hide the internal integrated system complexity or protect a sensitive database from direct connections.

Web Services – Web Services are gaining in popularity and becoming more frequent in integrating systems. The Web Services paradigm will allow you to provide a menu of information content that a SAS session may access without explicitly knowing where it is stored. Also most systems or database vendors provide Web Services access as a way to integrate with other systems.

Each of the approaches are applicable with its own limitations to transfer information between systems or a client and server. JDBC is still a preferred method, if the data volume is large and the SAS client will have direct access to the database, such as on the internal network or over VPN. For really sensitive data, even access from a SAS client on the internal network outside the server room is a sensistive issue.

If direct database connections are not possible, simple SOAP or Web Services are a viable alternative. The “Hello World” example would be a good fit to SOAP or a Web Service, as well as smaller data sets. One drawback with SOAP and Web Services is the lack of efficiency in transferring large volumes of information. By design, most implementations of SOAP and Web Services will retain all of the information transferred in memory while processing, which imposes a limit of how large data data sets can be transferred by resource limitations on the client, server, network or all of the above.

Efficiency can be maintained while transferring the larger data in pieces of a prespecified size, also known as chunking. The simple implementation would transfer a data structure in subsets, chunks, instead of one large block in order to remain within the available client, server and network resource restrictions.

SMALL DATA

A small data set, both sending and requesting multiple records, shows a real example of how the SAS Java object interface can be implemented for content such as process meta-data, statistics or summaries.

As an example, select a small data set, SASHELP.CLASS, to transfer from SAS. The aim is to transfer the content of the data set to our Java class, which will further send the content to a database when the Java method *saveData()* is called.

```
data _null_;
  set sashelp.class  end = eof;

  if ( _n_ = 1 ) then do;
     * -- initiate java object .. call this only once ;
     declare javaobj jobj ("smallData");
  end;

  * -- send data to our java class ;
  * -- note all data set variables except rc are input parameters to addData ;
  * -- note this line is called for each record in our input data set ;
  jobj.callIntMethod('addData', name, sex, age, height, weight, rc );

  * -- check for possible problems ;
  * -– rc = 0 is success and rc = 1 is failure ;
  if ( rc eq 1 ) then
     put "ER" "ROR: Failed to send the data to Java for " name ;


  if eof then do;
     * -- save the data ;
     jobj.callIntMethod('saveData', rc );
```

```
         * -- check for possible problems when saving data ;
         * -- rc = 0 is success and rc = 1 is failure ;
         if ( rc eq 1 ) then
            put "ER" "ROR: Failed to save the data.";

         * -- clean up after last use ;
         jobj.delete();
       end;

   run;
```

Similar to the Hello World example, first initiate the SAS Java object *jobj* to refer to the Java class *smallData*. For each record, the content of the variables *name*, *sex*, *age*, *height* and *weight* is transferred to our Java class and await a return code *rc*. The return code *rc* is used by the Java class to notify SAS, if the *addData* and *saveData* Java class methods were successful or failed.

After the last record in our input data set SASHELP.CLASS has been processed by the java method *addData*, the Java class method *saveData* transfers the data to the database, returns an error condition if it failed and finally conclude by freeing up memory with *jobj.delete()*.

CHUNKING

The approach of saving the content to a database after transfering all the input data to our Java class is efficient, if the data set is sufficiently small, A modified approach implementing chunking may be required when the data set is too large compared to the availabe network and client resources. It the data set is too large for the available client resources, a very slow or stalled system is possible.

The below example has the SAS code modified to call the Java class method *saveData()* for every 10 records, at which time the Java class is prompted to save the content to the database. The underlying Java code still remains the same as in the previous small data set example.

```
   data _null_;
     set sashelp.class  end = eof;

     if ( _n_ = 1 ) then do;
        * -- initiate java object .. call this only once ;
        declare javaobj jobj ("smallData");
     end;

     * -- send data to our java class ;
     * -- note all data set variables except rc are input parameters to addData ;
     * -- note this line is called for each record in our input data set ;
     jobj.callIntMethod('addData', name, sex, age, height, weight, rc );

     * -- check for possible problems ;
     * -– rc = 0 is success and rc = 1 is failure ;
     if ( rc eq 1 ) then
        put "ER" "ROR: Failed to send the data to Java for " name ;


     * -- save the data after every 10 records ;
     * -- change 10 in SAS mod function to set records in subset ;
     if ( mod(_n_, 10) eq 0 ) then do;
        jobj.callIntMethod('saveData', rc );

        * -- check for possible problems when saving data;
        * -– rc = 0 is success and rc = 1 is failure ;
        if ( rc eq 1 ) then
           put "ER" "ROR: Failed to save the data.";
     end;
```

```
    if eof then do;
       * -- save any remaining data ;
       jobj.callIntMethod('saveData', rc );

       * -- check for possible problems when saving data;
       * -– rc = 0 is success and rc = 1 is failure ;
       if ( rc eq 1 ) then
          put "ER" "ROR: Failed to save the data.";

       * -- clean up after last use;
       jobj.delete();
     end;

  run;
```

Deciding between the different approaches is dependent on several factors such as memory and connection performance. If the data content is sufficiently small compared to the available SAS session memory, a one time save at the end can be preferred. If there is a lack of performance, saving multiple subsets could possibly alleviate any performance issues.

The size of the subset, being 10, 100, 1000 or more records can be optimised based on available SAS session memory and performance requirements. A block of 1000 could be used as the arbitrary size of a sufficiently large data set, enabling SAS to start chunking only when data set has become sufficiently large. In our example, SAS controls the subset size, but this can easily be managed by the Java code itself.

BIG DATA

Larger data sets compared to our previous examples are very common. A clinical laboratory result data set may include 600 000 records, which is itself considered small in other contexts. It is however sufficiently large to be considered a large data set when considering the different methods of transfer.

Connection performance will become more apparent as the data size increases. Of our mentioned approaches, each will have different performance limitations. JDBC database connections are simple and provide reasonable performance. The large data issues with SOAP and Web Services are now apparent and have to be resolved.

Transfering large data volumes to our database follows the last example under Small Data. The complexities of implementing chunking can be hidden within the underlying Java code and our SAS code can be simplified to only prompt for or save the data.

A simple example is to retrieve a larger set of clincal laboratory results with our Java class bigData. The Java class is prompted to retrieve “labs” by the SAS code and create the output data set WORK.CLINLAB.

```
  data work.clinlab ;

     * -- define data set variable attributes ;
     attrib    subject    format = z10.
               visit      length = $35
               param      length = $35
               result     length = $35
     ;

    * -- initiate java object ;
    declare javaobj jobj ("bigData");

    * -- get data from our java class ;
    * -- select lab and return the number of records in numRecords ;
    jobj.callIntMethod('getDataFromSomewhere', "lab", numRecords );

    * -- check for possible problems retrieving data ;
```

```
      * -- numRecords lt 0 is failure and numRecords ge 0 is success ;
      if ( numRecords lt 0 ) then do;
         put "ER" "ROR: Failed to get data from Java.";
         stop ;
      end;

      * -- first call to nextRecord gets first record in sequence ;
      * -- hasMore = 0 is no records and hasMore gt 0 is next record found;
      jobj.callIntMethod('nextRecord', hasMore) ;

      do while (hasMore gt 0) ;

         * -- copy data from Java to SAS ;
         jobj.callIntMethod('getSubject', subject);
         jobj.callStringMethod('getVisit', visit);
         jobj.callStringMethod('getLaboratoryParameter', param);
         jobj.callStringMethod('getResult', result);

         * -- create record in output data set ;
         output ;

         * -- after first call to nextRecrod gets next record in sequence ;
         jobj.callIntMethod('nextRecord', hasMore) ;
      end;

      * -- clean up after last use;
      jobj.delete();

    run;
```

The approach is to first initiate the Java object in our DATA step, as in previous examples. Next, identify the information to be retrieved from the database, *lab*, and verify that the Java class method *getDataFromSomewhere()* was successful. The returned value of *numRecords* less than 0 is the Java class indicating failure and not a value dictated by the Java object in SAS.

The Java class method *nextRecord()* tests if there are any records retrieved. The *nextRecord()* method will by design return 0, if no records where collected from our connection or there are no more records to return to SAS.

If the method *nextRecord()* indicates more records to be transferred from the Java class to SAS, the DATA step populates the SAS variables *subject*, *visit*, *param* and *result* using the SAS Java object *callIntMethod()* and *callStringMethod()*, as applicable. A simple output statement makes sure that the data retrieved from our Java class is stored in the output data set WORK.CLINLAB.

Our Java class *bigData* is designed to hide the complexities of large data sets from the SAS code. If the data set is sufficiently large, an initial subset of data is transferred from the database to the Java class when SAS calls the Java method *getDataFromSomewhere()*. The database connection will also pass information to our Java class if there are more records in the database that our Java class has not yet received. When the Java class method *nextRecord()* is called, it will check if the local cache of subset records is empty and, if there are more records to receive, request the next subset of records from the database.

The Java class *nextRecord()* will return a value of 0 indicating that the transfer is complete, when all records have been received from the database and transferred to the data set WORK.CLINLAB.

The performance of a large data set transfer is dependent on several factors with no easy set of metrics.

- Time to create initial connection
- Transfer time of data subset, which may vary depending on network configuration. A network may restrict performance for large transfers while small are prioritised to preserve network performance for Office applications.
- Time to re-establish a connection or transfer session, if more than one subset is required.

- Client and server configuration of processing and memory allocation.
- Encryption. Clear text transfer, for example HTTP, is much faster and requires less client and server performance than encrypted transfers, such as SSL, SFTP, HTTPS, etc.
- Other client applications and network traffic.

A simple approach is to balance the size of subsets to transfer in relation to available client processing power and memory, connection times and overall network performance. Small amount of client memory will require more and smaller data subsets to attain an acceptable client system performance. If the server is too small, it is usually easier to get a bigger one if the application and its performance are critical.

BI-DIRECTIONAL

The ability to easily read and write across multiple requests and responses is key for integration and SAS to become more than just another client. All of the previous topics have discussed a reading or writing content within one DATA step. The examples presented previously can be extended to retrieve a data set from one database, calculate statistics locally and return the statistics to the same or a different central database. A simple way to store connection information, retain connections between DATA steps and other housekeeping could greatly increase performance and in some ways benefit maintainability.

A simple example of using Java to retain information between DATA steps is provided below. The first data step calls our Java class *firstClass* and saves a message with the Java class method *setMessage()*. A return code *rc* of 0 indicates success. The second data step uses Java class *secondClass* and the Java class method *getMessage()* to retrieve the message stored by the first data step.

```
data _null_;
  * -- initialise return code ;
  rc = -1;

  * -- initiate java object ;
  declare javaobj j ("firstClass");

  j.callIntMethod('setMessage', "This is the first class message.", rc);
  put rc = ;

  j.delete();

run;

/* -- more SAS here -- */

data _null_;
  * -- initialise return code ;
  length message $ 100;

  * -- initiate java object ;
  declare javaobj j ("secondClass");

  * -- send message to java ;
  j.callStringMethod('getMessage', message);
  put message = ;

  j.delete();

run;
```

The above passing of information is possible with the use of a Singleton class in the underlying Java code. In simple terms, a Singleton class exists once instead of ususal one instance for evey call, enabling our DATA steps to share the message. The Singleton class is never called directly from the SAS code, but used by both the Java classes *firstClass* and *secondClass* in the background. In-depth review of Singleton classes and other approaches are beyond the scope of this paper.

The Singleton Java class

```
public class mySingleton {

    ...

    public int setMessage( String new_message ) {
       /* Code to store message and return success of failure */
    }

    public String getMessage() {
       /* Code to retrive message set by setMessage() */
    }

}
```

The Java class *firstClass* contains the Java class method *setMessage()* called by our first DATA step. It stores the passed message text mymessage in the Singleton Java class for later retrieval.

```
public class firstClass {

    ...

    public int setMessage(String mymessage ) {
        /* code to store the passed message in my Singleton class and
           and return success or failure                                */
    }

}
```

Our second Java class *secondClass* contains the Java class method *getMessage()* called by our second DATA step. It returns the passed message text that was stored by the Java class *firstClass* in the first DATA step.

```
public class secondClass {

    ...

    public String getMessage() {
        /* code to return the message stored in my Singleton class */
    }

}
```

The Singleton class is an in memory storage and should be accounted for when considering the performance balance of available and required SAS client memory. A possible convention of Singleton use is to retain connection information, authentication details, simple logs and other useful housekeeping details, while defering storage of an entire data set to the approaches discussed for small and big data sets alike.

The Singleton class does not retain information between SAS programs, and any such implementation will have to use other means, such as physical storage of cache files, session files, etc. The capability of the Singleton class is still much appreciated.

The housekeeping of the hidden Java class details provides the tool for an efficient and modular application for use within one SAS program session.

```
%connect( server = myserver, user = myusername, password = xyz );
```

```
%getdata( select = demog, out = work.demog );

/*  Generate counts of gender Male and Female  */
proc freq data = work.demog ;
  tables gender  / out = work.freqs ;
run;

%poststatistics( data = work.fregs,
                 variable = gender,
                 statistics = count percent );

%close;
```

Each of the macros in the above example uses the SAS Java object and connects with a server to either recieve or send information. Internally, the Singleton class manages the authentication details, connection information and other application details, greatly simplifying the SAS code required.

CONCLUSION

The simple Hello World message is provides the basic principles of SAS interacting with Java through the SAS Java object. The complexity does increase when small and large data sets are considered, but the basic approach remains unchanged.

There are several information access methods available to Java, where the three JDBC, SOAP and Web Services are considered. Performance constraints and accessibility issues with each are topics to consider as the data volumes increase.

The transfer of small and large data sets, and a clear optimisation strategy is important to maintain efficency.

Transfer of data in any direction, sending and receiving, from within a single DATA step is a small step, but the real usefulness of the Java object is tangible when SAS can integrate with Java classes throughout a single program and retain program session information. The Singleton class is a very useful tool.

CONTACT INFORMATION

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

Magnus Mengelbier or Jan Skowronski
Limelogic Ltd
London, United Kingdom

E-mail: papers@limelogic.com
Web: www.limelogic.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.