PhUSE 2016

Paper CS02

SAS, Java & Excel: A ménage à trois

Dana Vinzelberg, Bayer AG, Berlin, Germany

ABSTRACT

Through the communication between SAS® and Java arise undreamt-of possibilities. Since Java is widely used and has access to a large amount of packages, which are free to use, the interface to Java expands the power of SAS. For instance a Java program which reads Excel files and passes their contents to SAS provides access at a more granular level, e.g. by enabling a sheet's input range to be defined. The data step component JavaObj utilizes the Declare statement to instantiate an object of a Java class and is able to call implemented methods. There are several obstacles to overcome, for instance the different number of data types in SAS and Java and therefore the mapping of the transfer parameters. This paper will reveal the endless opportunities of Java being invoked by SAS.

INTRODUCTION

Since nowadays Excel is a widely used tool for storing and exchanging data, it is desirable that this Excel data also be accessible in SAS. Since the methods implemented in SAS to import Excel data have some disadvantages regarding the operating system and file conversion, a more suitable way needs to be investigated.

COMMUNICATION

SAS AND EXCEL

There are several different ways to get Excel data into SAS and save modifications or new data back into an Excel file. If you do not want to use point-and-click, the most common ways are via PROC IMPORT/EXPORT and through the output delivery system (ODS). With SAS 9.3 comes additional functionality, namely LIBNAME EXCEL.

Several SAS options can be applied using PROC IMPORT or PROC EXPORT. The import of selected cells for example may be controlled with a WHERE statement. The file and the dataset need to be provided to both procedures and an identifier for the type of data to be imported needs to be specified with the option DBMS. There are several identifiers for reading Excel data. With DBMS=EXCEL data from Excel 97 up to Excel 2007 can be imported. This option is only applicable for Windows and not for Unix/Linux. The XLS option, which is also available for Unix/Linux, works differently from EXCEL and sometimes results in incorrect data.

The Output Delivery System (ODS) can be used with the ExcelXP tagset. This is applied like any other ODS destination: the actual code is enclosed between two ODS statements. One opens the ODS channel and the other closes it. The ExcelXP tagset does not operate on the native Excel file but on its XML version, which is also Excel specific. This leads to some problems while using VBA but allows the use of Excel formats and formulas.

The LIBNAME statement with the EXCEL tag is new in SAS 9.3 and provides access to an Excel file and its sheets through a library. The sheets are accessed via a two level name, similar to the <library>.<dataset> syntax but instead <library>.<sheet>. If the specified file does not exist, the file will be created. The disadvantage of this method is the lock on the Excel file whilst the library reference is active.

SAS AND JAVA

With SAS 9.2 an interface to Java was released, which combines the advantages of the SAS language and the object-orientated Java. The interface can be called within a data step as a component named JavaObj. It utilizes the DECLARE statement to instantiate the Java objects. Dot syntax is used to call methods from the instantiated object. This is unusual for SAS, but typical for Java.

The following code shows a minimalistic application example of the Java interface.
Java code:

```
public class Speak2Me{
    public Speak2Me(){}
    public String repeat(String sentence){
           System.out.println(sentence+"! - Java");
    }
}
```

SAS code:

```
DATA _null_;
    LENGTH said $100;
    DECLARE JavaObj speaker ('Speak2Me');
    speaker.callStringMethod('repeat, 'Hello SAS', said);
    speaker.delete();
    PUT said=;
RUN;
```

LOG output:

```
said=Hello SAS! - Java
```

This is how data in Java can be exchanged with data in SAS and vice versa within a data step.

To make the classes available for SAS, the Java code needs to be available in the form of a runnable JAR (**J**ava **Ar**chive) file. Access in SAS is realized by setting a CLASSPATH to the JAR. Since the interface can be used repeatedly in macros, the SET CLASSPATH statement can be implemented within a small macro, which needs just the path of the file as a parameter. Alternatively the Java Virtual Machine (JVM) to be applied can be set in SAS with the JREOPTIONS option. JREOPTIONS can only be executed on SAS startup whereas the CLASSPATH can be changed during a SAS session. SAS provides small macros to initialize and set the CLASSPATH (see reference [17] http://support.sas.com/kb/38/518.html). The following code example shows the setting of the class path with these macros.

```
%init_classpath_update;
%add_to_classpath(A:\path\to\the.jar);
```

The Java classes collected in the JAR are now available in SAS.

By instantiating an object with the statement DECLARE JAVAOBJ, the object provides the methods of its class to SAS. Calling a method is realized by using a modifier on the object and specifying the name of the method as a parameter. Since Java methods may return values, for each Java return type a different modifier is called.
Table 1 shows all types with their modifier for non-static and static methods.

TYPE	JAVAOBJ METHODS	
void	call**Void**Method	call**Static**VoidMethod
double	call**Double**Method	call**Static**DoubleMethod
String	call**String**Method	call**Static**StringMethod
char	call**Char**Method	call**Static**CharMethod
Boolean	call**Boolean**Method	call**Static**BooleanMethod
short	call**Short**Method	call**Static**ShortMethod
byte	call**Byte**Method	call**Static**ByteMethod
long	call**Long**Method	call**Static**LongMethod
float	call**Float**Method	call**Static**FloatMethod
int	call**Int**Method	call**Static**IntMethod

Table 1:
JavaObj methods that invoke Java methods
[http://www2.sas.com/proceedings/sugi30/241-30.pdf]

The syntax of all JavaObj methods is similar and can be broken down into the following syntax:

```
returnCode = JavaObj.call{Modifier}{Type}Method(  "JavaMethodName",
                                                   argument(s),
                                                   SASReturnValue
                                                 )
```

The `returnCode` is a numeric value, which is 0 if the call was successful and non 0 otherwise, to indicate the performance of the call. The `returnCode` is not the return value, which is passed to SAS by the Java method. The actual return value from the Java method is caught within a SAS variable which is given as parameter in the JavaObj component. The `{Modifier}` tag is optional but should be Static for static Java methods. In contrast the tag for `{Type}` is required and represents the primitive Java types. Also mandatory is the `JavaMethodName`, which is the name of the Java method to be invoked and which is case sensitive. Applicable `argument(s)` and a variable for a return value if necessary must be provided in accordance with the requirements of the Java method being invoked. The variable for the return value can be of type numeric, character or even JavaObj to pass object references.

Everything written to the console in Java results in a message written to the log in SAS. If something goes wrong in Java, an exception is thrown and converted to an error message which is written to the SAS log. Recommended however is catching the exceptions in Java and writing custom error messages for the SAS log.

To import and export data to and from SAS, the entire interaction with Excel can be entrusted to Java.

JAVA AND EXCEL

Java is widely used and therefore has access to many packages many of which are free to use. A well-known source for free packages is Apache. Apache provides open-source packages for many different fields of application. One of its packages is utilized to deal with Microsoft documents and is called Apache POI.

Apache POI HSSF is a package which reads and writes Excel files with the extension XLS (versions ’97-2007), and Apache POI XSSF is for Excel 2007-2013 with the extension XLSX. Excel files can be converted to a Java object with these packages. This Java object consists of several subclasses which represent the sheets, rows and cells of the Excel file. There are also classes which the HSSF and XSSF classes originate from. These classes are called superclasses. Superclasses can be derived to the HSSF version as well as the XSSF version of the class. Therefore the superclasses can be used as wildcards for the corresponding HSSF or XSSF classes. These superclasses are Workbook (which represents the file), Sheet, Row and Cell. They have a hierarchy as listed, which means that the Workbook class includes objects from the Sheet class, which includes objects from the Row class, which includes objects from the Cell class. Therefore Cell is a subclass of Row and so on.

The classes contain several built-in functions, which are called *methods* in Java. These provide access to the subclasses and the information within. The meaning of the methods is not hard to understand, but since the package provides a huge variety of methods, it is sometimes difficult to keep the general overview.

TEAM PLAY

%EXCEL2SAS

We have implemented two SAS macros, one to act as an Excel reader and the other as an Excel writer. Both use Java.

To ensure a flexible and robust SAS macro, a single data step for calling the Java interface is not sufficient. The only necessary parameter is the name of the Excel file to be read in. Firstly in the code there are common checks for correct parameter input, such as whether the given file exists. After that an automated investigation for the names of the sheets is carried out, which is also realized through the JavaObj interface within a data step. The names of the resulting dataset(s) are determined by the sheet names. Via a loop all sheets are processed and their corresponding datasets created. First the variable names are determined and therefore the number of columns in the table in Excel is needed. It is also possible to apply a range to the Excel table to specify which values to read in. A Java method is used to pass each row of the Excel sheet to SAS cell by cell, with each variable receiving a cell value to obtain an observation. This is done for all sheets. The formats are automatically derived from the cell values. If there is a string in one of the cells in a given column, then the corresponding variable format is character. Otherwise, if there are only numbers in the cells, the variable is given a numeric format (namely `BEST.`). The format derivation is also carried out by Java. The interaction between the SAS macro, the Java classes and an Excel file is shown in *Figure 1*.

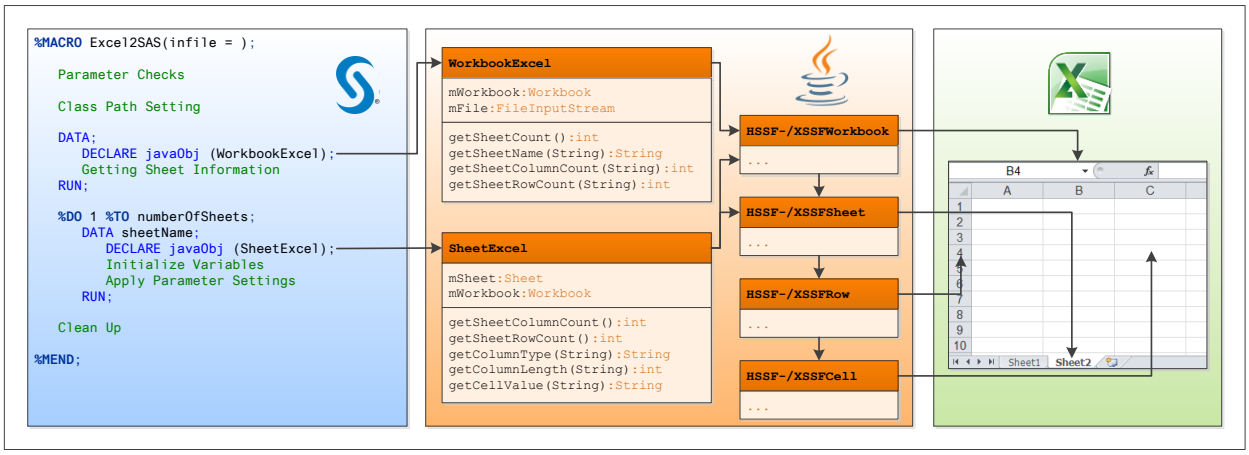

Figure 1:
Overview of Macro Excel2SAS, Java Classes and Access to Excel

Adapted Java classes are implemented by using Apache POI HSSF and XSSF to pass the Excel content to SAS. These classes hold methods which manage the communication with SAS. The included package enables the content of an input Excel file to be processed and adapted for output to SAS. The first class (`WorkbookExcel`) reads the Excel file and represents the file itself. The class holds methods which, for example, count the number of sheets, get the sheet names and clean-up the memory after finishing the export. With the next class (`SheetExcel`), which represents a single sheet, a sheet can be processed. The class holds methods which count the rows and columns, determining if there are only numbers in a column and returning the value of a single cell. There is also a method which gets the length of the longest string in a column, to determine which format length to use within SAS.

%SAS2EXCEL

The process for writing an Excel file is nearly the same. The datasets to be written are required as a parameter as are the location and name of the Excel file to write to. First the sheet names are determined from the given dataset names. The tables are passed to Java value by value. There is only one class in Java for writing. The class holds methods for creating sheets and rows and cells within. The process flow is shown as pseudo code and Java diagrams in *Figure 2*.

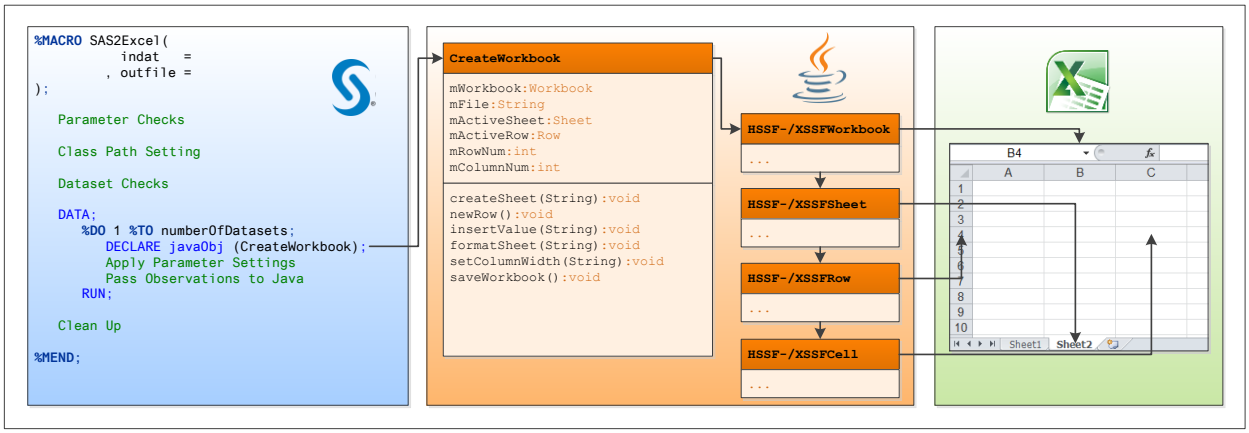

Figure 2:
Overview of Macro SAS2Excel, Java Classes and Access to Excel

OBSTACLES

DATA TYPES

There are several obstacles to overcome. The most obvious one is the different data types in SAS and Java. While SAS provides only two primitive data types, Java has nine. As a consequence another tricky obstacle appears, which is the parameter passing, more precisely the type of the passed parameter. The arguments passed from SAS to Java need to be string or double for Java. This is important to know for implementing the Java methods. A string may be converted to another data type which may be more suitable in Java. The return value of a method in Java may also be of other types. All return values may be truncated, if the target variable in SAS does not have the appropriate

format. Strings are mapped to character formats and numbers (double, float, int, byte, short, long) are mapped to numeric SAS formats. Character and numeric formats are mapped to string and double, which explains the mentioned restriction for the argument type in Java. Boolean and char need to be passed in another way (e.g. as strings). This mapping of the data types is shown in *Table 2*.

JAVA	SAS
string	character
double	numeric
float	
int	
byte	
short	
long	
boolean	
char	

Table 2:
Data types in Java and SAS and their mapping

If a floating point number is passed from Java to SAS, it is likely that there will be precision errors, which are caused by the different machine precision used by the Java Virtual Machine and SAS.

LARGE FILES

Processing large files leads to high memory consumption when using the Apache HSSF or XSSF package. Consequently an out-of-memory-error is thrown if the heap space is consumed. Usually the heap space is set to 256 MB. It is possible to set the heap space prior to the execution according to the RAM on the computer. But this is not user friendly and cannot easily be done automatically.

For writing an Excel file Apache also provides SXSSF, which is a streaming version of XSSF and is less memory intensive as it writes the Excel file in parts. Since XSSF is for XLSX files only, XLS files cannot be processed with SXSSF. SXSSF uses a window covering about 100 lines to write the values to an Excel file. If the end of the window is reached, the values are written to the file after which there is no way to have access to these values. This is why this is a good solution for writing but not for reading. Whole columns need to be accessible e.g. to get sufficient information to determine the format.

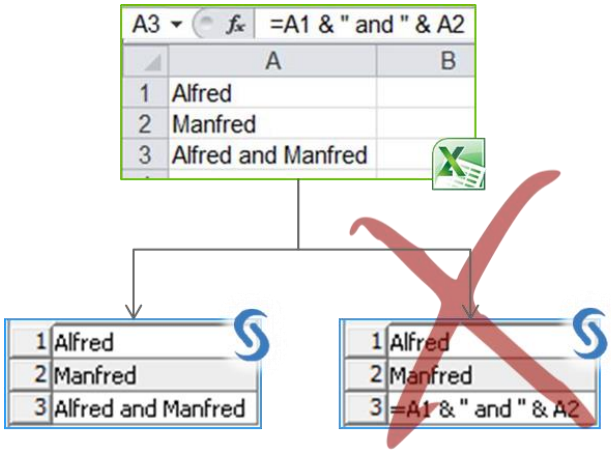

Figure 3:
Correct Handling of Excel Formulas

Reading an Excel file is realized by parsing its XML version. The SAX parser from JavaX does not work on the native Excel file, but it is able to handle Excel formulas and formats correctly. If formulas are applied in a cell, the value passed to SAS should not be the formula but the evaluated value. This is shown in *Figure 3*. The evaluated values are stored in a simple object, which differentiates between numeric and character values. Each row is represented as an array of these objects. An array of rows stands for a sheet and an array of sheets for a workbook. The methods of the corresponding classes for large workbooks and sheets are the same as for the adapted HSSF and XSSF ones. That makes the implementation much easier. Only the instantiation of the required object differs and the rest of the code is the same. To check whether a file is too large to be read in with HSSF or XSSF, a check class is implemented in Java. This class tries to read the file with HSSF or XSSF respectively and contains a method `isFileLarge` which

returns “No” if the file caused no problems with the HSSF or XSSF, and “Yes” otherwise. With this return value, it can be decided within SAS which class to instantiate. The SAX parser is only able to parse XLSX files. The size of an XLS file is most often smaller than XLSX, because of its lower row and line limitations. Therefore an Excel XLS file can usually be processed without an out-of-memory-error, however if this occurs, an error message is written and the file can be converted to an XLSX file within Excel in order that it can be processed.

SPECIALITIES OF THE LIAISON

For each character variable in SAS it is advisable to define the length beforehand. To enable this, all cell values in Excel need to be examined for being character and for their length. The maximum character length of all cells in a column is then used as the format length for the variable. If this length is smaller than 200, it is not used and a `$200.` format is used instead. If the Excel file is processed in its XML version for large files, an additional problem occurs. Each filled cell is read into the array. Therefore if tables are filled unevenly and the variable names are not all present in the first line, the arrays representing the rows differ in their length. This needs to be fixed in Java so that the correct values can be passed to SAS. For empty cells at the end of a row, the corresponding arrays are filled up with empty values. Empty cells between filled ones can be determined by checking the index of the input cells. The index is the same as the cell reference in Excel (e.g. A7). When writing an Excel file it is important to know the limitations of an Excel file. If an XLS file is requested, the number of columns is limited to 256 and the number of rows to 65536.

IMPROVEMENTS

ENHANCED FUNCTIONALITY

Additional parameters are integrated to advance the functionality of the macro itself.

The location of the imported datasets can be defined by setting a library. Providing additionally a dataset name results in the dataset being given that name instead of the sheet name. Another parameter defines whether the second row should be used as labels for the variables. By default, the variable names are taken from the first row in the sheet, but if a name is invalid for SAS or duplicated, it is replaced by the tag “col” together with its column number. It is also possible to specify which part of a sheet should be imported by defining a range. In this case all variables are named as “col” together with their column number. If a column contains only empty cells (except for the variable name and if required the label) it can be specified whether the resulting variable is numeric or character; by default it is numeric. Variable attributes (e.g. formats) for an output dataset can be set by defining a reference dataset which is used as a template for the output dataset. The reference dataset does not have to contain data as it is only the variable attributes which are used. An Example is shown in *Figure 4*. On the left hand side is the reference dataset whose formats are applied after reading in the Excel sheet in the middle. The output dataset with the formats from the reference dataset is displayed on the right hand side.

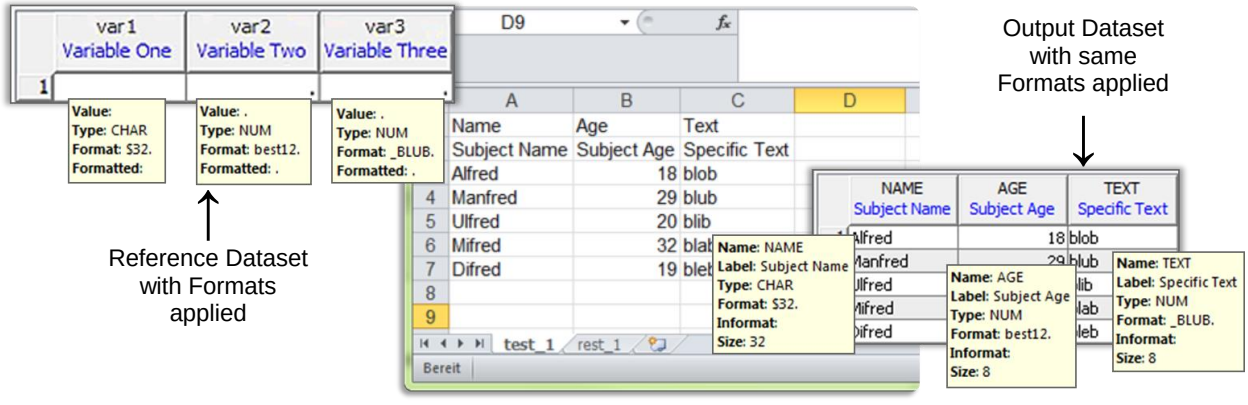

Figure 4:
Example for Usage of a Reference Dataset

When exporting datasets into Excel, the sheet names are definable through a parameter. The dataset names are used as sheet names unless these are explicitly specified. It is possible to write the variable labels to the second row and start with the values at row three. An auto filter can be applied to the Excel file through a parameter, to have a filter option for every column available. If there are any variables which should not be exported, these can be excluded. Since these need to be listed, there is an option to include only specified variables, say if the list of variables to exclude is longer than the list to include. The formats of the SAS variables need to be fully considered if they affect the output. It should be specified too whether the raw or formatted values be written to the file, and a parameter for this is provided. In any case the applied formats themselves are lost but they may be saved and reapplied by using a reference dataset.

THE POWER OF THREE

The most powerful advantage of Java is its platform independence. Because of that Java is widely used and very popular. Once a program is written, it will run on any platform which supports Java, such as Windows, Linux or Mac, and it can therefore be used for exchange between systems as well as running web applications, games, mathematical programs, GUIs and much more. This means it is very flexible, especially with its huge amount of freely available third-party libraries and code, which are most often very well written. Additionally Java is well structured due to its object oriented style. Since Java is a programming language with a general purpose, it is a great support language for SAS. SAS is a more specific system for statistical analysis, but one which is able to do a lot more than just statistics, and one which can handle huge amounts of data very efficiently. In contrast to SAS and Java, Excel is the easiest way to collect and distribute data. Since most people are able to input data into Excel, it is widely used to exchange data. This is why a combination of the three methods is a powerful way to collect, process and exchange clinical data.

CONCLUSION

The JavaObj interface raises the power of SAS to a new level for SAS programmers with skills in Java. The best of both programming languages can be combined to create a powerful interaction between the two. The ménage à trois with Excel is just a small example of what is possible with this liaison of SAS and Java.

REFERENCES

[1] "*Reading Data into SAS*", http://www.ats.ucla.edu/stat/sas/library/SASRead_os.htm#ReadingExcelFilesIntoSAS, 16.08.2016

[2] "*Using ODS to Generate Excel Files*", Chevell Parker, https://support.sas.com/rnd/base/ods/templateFAQ/Excel1.pdf, 16.08.2016

[3] "*ODS Excel Statement*", http://support.sas.com/documentation/cdl/en/odsug/67921/HTML/default/viewer.htm#p09n5pw9ol0897n1qe04zeur27rv.htm, 18.08.2016

[4] "*Try This Demo: The ExcelXP Tagset and Microsoft Excel*", https://support.sas.com/rnd/base/ods/odsmarkup/excelxp_demo.html, 22.08.2016

[5] "*ODS Tagset ExcelXP*", http://de.saswiki.org/wiki/ODS_Tagset_ExcelXP, 22.08.2016

[6] "*How do I read/write Excel files in SAS?*", http://www.ats.ucla.edu/stat/sas/faq/rwxls8.htm, 16.08.2016

[7] "*PROC IMPORT Statement*", http://support.sas.com/documentation/cdl/en/acpcref/63184/HTML/default/viewer.htm#a003102096.htm, 16.08.2016

[8] "*LIBNAME Statement Syntax*", http://support.sas.com/documentation/cdl/en/acpcref/63181/HTML/default/viewer.htm#n1wvmggexroxgyn17rp61jml3cvn.htm, 16.08.2016

[9] "*How to Excel with SAS®*", Keith Cranford, Office of the Attorney General of Texas, Child Support Division, http://www.lexjansen.com/scsug/2007/how/HOW-Cranford.pdf, 16.08.2016

[10] "*Java in SAS®: JavaObj, a DATA Step Component*" Object, Richard A. DeVenezia, Independent Consultant, http://www2.sas.com/proceedings/sugi30/241-30.pdf, 18.08.2016

[11] "*Greetings from the Edge: Using JavaObj in DATA Step*", Richard A. DeVenezia, Remsen, NY, http://www2.sas.com/proceedings/sugi29/033-29.pdf, 18.08.2016

[12] "*Using the Java Object*", http://support.sas.com/documentation/cdl/en/lrcon/68089/HTML/default/viewer.htm#n0swy2q7eouj2fn11g1o28q57v4u.htm, 18.08.2016

[13] "*Apache POI - the Java API for Microsoft Documents*", https://poi.apache.org/index.html, 19.08.2016

[14] "*POI-HSSF and POI-XSSF - Java API To Access Microsoft Excel Format Files*", https://poi.apache.org/spreadsheet/, 25.08.2016

[15] "*What is Java technology and why do I need it?*", https://java.com/en/download/faq/whatis_java.xml, 29.08.2016

[16] "*Java Pros And Cons*", http://c2.com/cgi/wiki?JavaProsAndCons, 29.08.2016

[17] "*Dynamically setting the CLASSPATH for use in a DATA step that uses the Java object component*", http://support.sas.com/kb/38/518.html, 31.08.2016

[18] "Apache POI HSSF Class Diagram", http://poi.apache.org/spreadsheet/diagram1.html, 31.08.2016

CONTACT INFORMATION

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

Dana Vinzelberg
Bayer AG
Muellerstr. 178
Berlin / 13353
Work Phone: +49 30 468 196998
Email: dana.vinzelberg@bayer.com