Paper SBC-132

JavaObj, the newest production component object

Richard A. DeVenezia, Independent Consultant, Remsen, NY

ABSTRACT

The DATA Step component object repertoire continues to grow in each new major release of SAS®. HASH and HITER were the first objects supported in release 9. In release 9.2 JAVAOBJ has gone production and is fully supported. This paper will show some examples incorporating JAVAOBJ.

Keywords: JavaObj, Java Object, Component Object, Interface, DATA Step, SAS9.2, instantiation, custom classes

INTRODUCTION

The first example will show how to obtain a folder listing. The second will show how to input data from text files contained in a zip archive. Before the examples are shown a brief review of component objects and Java classes is in order.

COMPONENT OBJECTS

A component object is a SAS developed code object that is external to the DATA Step compiler. Each component object can be thought of as a new type, extending your repertoire beyond the standard numeric and character types. The DATA Step contains syntax that allows a SAS programmer to activate and access the component object during the runtime of a step. An object cannot be stored in a SAS table. A component object has methods and properties. The SAS documentation describes the names and details of the methods and properties.

DECLARATION

The DECLARE statement is used to identify the variables that are objects. An object variable is instantiated using DECLARE or the _NEW_ operator. Object variables are coded with dot notation in order to invoke a method and access a property.

```
DECLARE component-object variable-name (object-constructor-arguments);
```

or

```
DECLARE component-object variable-name;
variable-name = _NEW_ ( object-constructor-arguments );
```

The constructor arguments are different for each component object and are stated in SAS documentation. An object that has been *newed* is also called an **instance** of the object type. Instances can not be stored in an output table. The DATA Step compiler is only aware of known component objects, but not their calling details. The arguments specified in your code are evaluated by the internal component object handling subsystem at run-time. Any problems with the arguments (the part between the parenthesis) are reported as errors in the log window.

DOT NOTATION

Object variables are coded with dot notation in order to invoke a method and access a property.

Read a property value

```
X = variable-name.property-name;
```

Set a property value. Note that not all properties are settable. The expression resultant type must match the property type.

```
variable-name.property-name = expression;
```

Invoke a method

```
variable-name.method-name ( method-arguments );
```

ASSIGNMENT

Component object variables are references. Object variables of the same type can be assigned to one another.

```
DECLARE ComponentObject co_one(); * causes an object instantiation;
DECLARE ComponentObject co_two;   * declares an object variable only, no instantiation;
co_two = co_one;  * co_one and co_two now refer to the same object;
```

Assignment can cause loss of reference and memory leaks. There is no garbage collection system for deleting unreferenced component objects.

```
DECLARE ComponentObject foo(); * causes an object instantiation;
DECLARE ComponentObject bar(); * causes an object instantiation;
foo = bar;  * instance originally referenced by foo is now unreachable;
```

At present there is no syntax for declaring an ARRAY of objects.

REPERTOIRE

The following are the component objects that are supported at the production level in version 9.2; Hash, HashIter, JavaObj. The OdsOut object is available at the preproduction level.

JAVAOBJ READINESS

JavaObj went to production level in 9.2. The means there is SAS Technical Support staff and apparatus in place ready to answer questions and respond to bug issues. There are several items to be aware of when using JavaObj.

CLASS LOCATIONS

The CLASSPATH environment variable is used to specify the location of custom developed classes you have purchased, created or been given. The value can be set as a system wide variable using (Control Panel ▸ System ▸ Advanced ▸ Environment Variables), or a session specific variable using -SET CLASSPATH in the command line or your SASV9.CFG file. In SAS V9.2 TS1M0 the CLASSPATH must point to a folder containing a Java run-time jar file (rt.jar).

JAVA VIRTUAL MACHINE (JVM)

The SAS session startup option -JREOPTIONS is used to alter the default JVM used during the SAS session. The new JVM is specified by defining the name value pair -Dsas.jre.home=*path.* Changing this setting to the newest available JVM can cause failure in other parts of your SAS session that rely on the default installation settings; specifically ODS statistical graphics.

Proc JAVAINFO can be used to examine important details about the current Java environment being hosted within the SAS session.

JAVA CLASSES

There are numerous tools for developing Java classes that you can use from the DATA Step. The simplest to find and most common is the Sun Java Development Kit Standard Edition (JDK SE) found at developers.sun.com. Some developers prefer an integrated development environment (IDE) as presented in the NetBeans (Sun), Eclipse (Open-source) or JBuilder (Codegear) applications.

The entire breadth and width of the standard Java libraries are available to you. Additional libraries added to the CLASSPATH also become available. Security and networking are part of the fundamental considerations within Java, and you can leverage that to your advantage in your SAS programs.

PRIMITIVE TYPES

A Java class may contain public fields that can be set or queried. A field is a variable of a primitive type or of an object type. There are nine primitive types in Java; boolean, char, byte, short, int, long, float, double, and void.

METHOD SIGNATURES

The method name and pattern of argument types that it accepts is called a signature. This is an important concept when invoking a Java method from a JavaObj. When signatures do not match an error will occur and the desired method will not run. Additionally, the signatures of non-deprecated methods of a library can be considered a contract with the class developers. The contract means the signature will not change from release to release of the library. A stable contract is important in making your code-base consistent and maintainable for many years.

JAVAOBJ DETAILS

The declarative syntax for JavaObj is as follows:

```
DECLARE JavaObj variable;
variable = _new_ JavaObj ( class, constructor arguments );
```

or

```
DECLARE JavaObj variable ( class, constructor arguments );
```

variable: A SAS Name

class: Required. Character (variable or literal) specifying a class name that should be found in the JVM's classpath. Package or hierarchy is indicated with a slash (/), not with a dot (.).

arguments: Depend on *class*. Constructor arguments are of type Character, Numeric and JavaObj. The pattern of the argument types is called the *signature.*

EXAMPLE

Note that a Java programmer would code the full path class names as java.lang.String and java.awt.geom.Point2D.Double:

```
declare JavaObj j ('java/lang/String');
declare JavaObj j ('java/awt/geom/Point2D$Double');
```

In SAS the constructor of JavaObj expects dots (.) of a hierarchy to be replaced with slashes (/) and the dots (.) of a nesting with dollar signs ($).

The SAS log will show errors if the class is not found, or if the SAS signature does not match a Java constructor signature. Additionally, when a class is not found, a Java exception message is written to the SAS session standard error.

```
java.lang.NoClassDefFoundError: xyzzy
```

DATA STEP TO JAVA

DATA Step values are delivered to Java objects by using the ***set*★** (see Table 1) and ***call*★** (see Table 2) methods of JavaObj.

In the case of ***set**** methods, JavaObj maps SAS numeric values to Java types byte, short, int, long, float or double. If the SAS value is outside the range of the destination Java type you might experience truncation or miscasting. SAS character values are mapped to the Java class `String`. SAS JavaObj references are mapped to the class specified at declaration or _new_; thus you can pass objects from DATA Step to Java.

JAVA TO DATA STEP

DATA Step obtains values from Java objects by using the ***get*★** (see Table 1) and ***call*★** (see Table 2) methods of JavaObj.

JavaObj maps Java types byte, short, int, long, float and double to SAS numeric. The Java class `String` is mapped to SAS character. No other Java class is mapped to a SAS type. This means object references cannot be returned to DATA Step from Java.

ACCESSING FIELDS

The value of a public primitive Java class field can be retrieved or assigned through JavaObj methods. Methods that retrieve a value are known as *getters*. Methods that assign a value are called *setters*.

*rc = javaobj . {**access**} {**modifier**} {**type**}*`Field` *(field, argument)*

rc: return code - 0 if successful, not 0 otherwise.

access: `get` or `set`

modifier: Optional. If present it should be `Static`.

type: A Java primitive type or `String`.

field: Character. Name of Java class field.

argument: Character or Numeric. For setters, argument is a variable or a literal.
For getters, argument must be a variable (that receives the value).

	access			
type	***getters***		***setters***	
String	`getStringField`	`getStaticStringField`	`setStringField`	`setStaticStringField`
Double	`getDoubleField`	`getStaticDoubleField`	`setDoubleField`	`setStaticDoubleField`
Int	`getIntField`	`getStaticIntField`	`setIntField`	`setStaticIntField`
Short	`getShortField`	`getStaticShortField`	`setShortField`	`setStaticShortField`

	access			
type	**getters**		**setters**	
Byte	`getByteField`	`getStaticByteField`	`setByteField`	`setStaticByteField`
Long	`getLongField`	`getStaticLongField`	`setLongField`	`setStaticLongField`
Float	`getFloatField`	`getStaticFloatField`	`setFloatField`	`setStaticFloatField`

Table 1 JavaObj methods to access Java fields.
Note: Version 9.2 also provides access to `Boolean` and `Char` fields.

GETTERS

The typecasting of a Java primitive to SAS numeric that occurs inside the getter methods is not a problem. You must be careful though, some `long` values of magnitude $>2^{53}$ cannot be represented in SAS numerics (64-bit IEEE-754 format).

SETTERS

You should only set Java fields with values that fit the range they were designed to hold. What happens when a value is outside the range? In version 9.1 you may encounter typecasting problems or experience abnormal errors. Version 9.2 will report a normal error in the log.

INVOKING METHODS

The methods of a Java class are invoked through JavaObj methods.

rc = *javaobj* . `call`{***`modifier`***}{***`type`***}`Method` *(method, argument(s), return)*

rc: return code - 0 if successful, not 0 otherwise.

modifier: Optional. If present it should be `Static`.

type: A Java primitive type or `String`.

method: Required. Character. Name of Java class method.

argument(s): Depends on *method*. Character, Numeric or JavaObj. Signature must match that of *method*.

return: Character or Numeric variable that receives value returned by *method*. Not present if *method* is void.

type	**modifier**	
Void	`callVoidMethod`	`callStaticVoidMethod`
Double	`callDoubleMethod`	`callStaticDoubleMethod`
String	`callStringMethod`	`callStaticStringMethod`
Boolean	`callBooleanMethod`	`callStaticBooleanMethod`
Short	`callShortMethod`	`callStaticShortMethod`
Byte	`callByteMethod`	`callStaticByteMethod`
Long	`callLongMethod`	`callStaticLongMethod`
Float	`callFloatMethod`	`callStaticFloatMethod`
Int	`callIntMethod`	`callStaticIntMethod`

Table 2 JavaObj methods that invoke Java methods.
Note: Version 9.2 JavaObj will also invoke `Boolean` and `Char` methods.

In Java a class method has a *type* and *signature*. The *type* is the type of the value returned by the method; nothing (void), a primitive type or a class. The *signature* is the pattern of argument types the method accepts; nothing, primitives and classes. Note: two methods of a class can have the same name, but only if their signatures are different.

There are restrictions regarding the Java methods that JavaObj can invoke.

TYPE
JavaObj will only invoke Java methods of *type* void, primitive or `String`.

SIGNATURE
A JavaObj will only invoke Java methods whose signature matches the signature of the arguments passed to the JavaObj method. DATA Step can pass JavaObj methods three types; `Numeric`, `Character` and `JavaObj`.

DOUBLE
SAS numeric values are always presented to Java as `double`. If a method is expecting a different primitive type you will not be able to invoke it from SAS. An adapter Java class will have to be created to typecast doubles coming from SAS to the primitive type needed by the method.

STRING
SAS character values are always presented to Java as `String`.

CLASS
SAS JavaObj variables are always presented to Java as the exact class instantiated at `declare` or `_new_`. If the JavaObj is of a subclass of the class expected by the Java method you will not be able to invoke the method from SAS. An adapter Java class will have to be created to classcast objects coming from SAS to the class needed by the method.

EXAMPLE:
Passing a `JavaObj` and getting a `String` in return. The Javadoc for Class String, Method concat is as follows:

`String`	**`concat`**`(String str)` Concatenates the specified string to the end of this string.

```
data _null_;
  length s_out $200;
  declare JavaObj j1 ('java/lang/String','ABCDE');
  declare JavaObj j2 ('java/lang/String','FGHIJ');
  j1.callStringMethod ('concat', j2, s_out);
  put s_out=;
  j1.delete(); j2.delete();
run;
```

```
s_out=ABCDEFGHIJ
```

ADAPTERS
It is unlikely that the parameter types that SAS can pass to Java will satisfy the signatures surfaced by the objects in a Java library. An adapter is a Java code that must be written to facilitate the interfacing between JavaObj and functionality to be implemented in Java. The JavaObj instantiates the adapter which has methods to type cast incoming arguments for operation within the JVM.

GETTING A LIST OF FILES
Java has a File object for enumerating and obtaining information about system files. The File listFiles method returns an array of File objects that cannot be passed back to a DATA Step JavaObj. Thus, a wrapper class with a small set of interfacing methods must be written to surface the desired functionality.

ListFiles.sas

This example supposes the wrapper class named ListFiles is in the CLASSPATH. The class constructor is used to pass the folder of which a file listing is desired. The getNext method advances the wrapper to the next file in the folder. There are support methods (getName, getType, etc...) for obtaining the information about the current file. The support methods are invoked using the JavaObj call*Method methods. Note that this example does not use public fields for this data. If fields had been used the JavaObj methods get*Field would have been coded.

```
data files;
```

```
  length name $200 type $4 len mod b 8; drop b;
  format mod datetime19. len comma9.;
  dcl javaobj jlf ("ListFiles", "C:\WINNT");
  jlf.callBooleanMethod ("getNext", b);
  do while (b);
   jlf.callStringMethod("getName", name);
   jlf.callStringMethod("getType", type);
   jlf.callLongMethod  ("getLength", len);
   jlf.callDoubleMethod("getModifiedAsSasDateTime",mod);
   output;
   jlf.callBooleanMethod ("getNext", b);
  end;
  jlf.delete();
run;
```

ListFiles.java

An adapter often follows different coding patterns than typically found in Java programming books. This class has fields named **files** and **i** so that state is maintained between invocations of its methods. The arithmetic needed to convert a files timestamp into a SAS datetime value is in the Java code and thus not cluttering up the DATA Step.

```
import java.io.*;
import java.util.*;

public class ListFiles  {

  File[] files;
  int i = -1;

  public ListFiles () {
      ListFiles(".");
  }

  public ListFiles ( String path ) {
      File dir = new File(path);
      files = dir.listFiles();
  }

  public boolean getNext () {
    i++;
    return i<files.length;
  }

  public String getName () {
    if (i<files.length)
      return files[i].getName();
    else
      return "";
  }

  public String getType () {
    if (i<files.length)
      if (files[i].isDirectory())
         return "Dir";
      else
      if (files[i].isFile())
        return "File";
      else
        return "?";
    else
      return "";
  }

  public long getLength () {
    if (i<files.length)
      return files[i].length();
    else
      return Double.NaN;
  }

  public double getModifiedAsSasDateTime () {
    if (i<files.length) {
      Calendar saszero = Calendar.getInstance();
      saszero.set(1960,Calendar.JANUARY,1,0,0,0);
      long sasmillis = files[i].lastModified()
                     - saszero.getTimeInMillis();
      return sasmillis / 1000;
    }
    else
      return Double.NaN;
  }
}
```

READING DATA ARCHIVED IN A REMOTE ZIP FILE

Consider the situation of needing to read text data from files stored in a zip archived reposited on a web server. The archive is named “alarms-2000.zip” and contains text files, one file per month, named “*yyyy-mm*-alarms.log”. Each line of data in these files is formatted as *day-of-month time-alarm-started duration-of-alarm alarm-type*.

ReadZippedFiles.sas

A DATA Step uses “ReadZippedFiles” to iterate through the files in the archive and to read the text lines from each file. The lines are parsed with an input statement. Replacing the _infile_ buffer prior to the input statement is a 'trick' technique.

```
filename buffer catalog 'work.foo.bar.source';
* create dummy entry that infile will use;
data _null_;
  file buffer;
  put ' ';
run;

data alarms (keep=when duration type);
```

```
  format when datetime16.;
  format time duration time8.;
  length type $10;

  length archive entryname $200;
  length line $256;

  * would also work if archive is a local file;
  archive = 'http://www.devenezia.com/papers/sugi-29/examples/alarms-2000.zip';

  * prep _infile_ buffer;
  infile buffer ;
  input @;

  declare javaobj jz ('ReadZippedFiles', archive);

  jz.callBooleanMethod ("getNextEntry", entryFound);

  do while (entryFound);
    jz.callStringMethod ("getEntryName", entryname);
    yearmo = substr(entryname,1,8);

    jz.callStringMethod ("readLine", line);
    jz.callBooleanMethod ("endOfEntry", eoe);

    do while (not eoe);
      * parse line of text with input statement;
      _infile_ = trim(yearmo) || line;
      input @1 date yymmdd10. time: time8. duration: time8. type: @ ;
      put _infile_;
      when = dhms(date,0,0,0)+time;
      OUTPUT;
      jz.callStringMethod ("readLine", line);
      jz.callBooleanMethod ("endOfEntry", eoe);
    end;

    jz.callBooleanMethod ("getNextEntry", entryFound);
  end;

  jz.delete();
  stop;
run ;

filename buffer;
```

ReadZippedFiles.java

This adapter class serves multiple purposes. The first is to list names of the files in the archive, and the second is to deliver the textual content of each file in the archive. The highly capable URL object is used when the archive name is an Internet address. The structured hierarchy of the java.io library lets the java programmer deal with the concept of an input stream. The gory details of a dealing with the contents of a .zip file do not have to be understood; only the methods of the Java provided ZipInputStream class.

```
import java.io.*;
import java.net.*;
import java.util.*;
import java.util.zip.*;

public class ReadZippedFiles {

  ZipInputStream zis = null;
  Enumeration e = null;
  ZipEntry ze = null;

  boolean zeLoopStarted = false;
  boolean zerLoopStarted = false;
  boolean zerEndOfFile = false;

  BufferedReader zer;

  public ReadZippedFiles ( String aZipFile ) {
    URL url = null;
    InputStream is = null;

    try
    { // is the string a URL ?
      url = new URL ( aZipFile );
      is = url.openStream();
    }
    catch (Exception e1)  {}

    if (url == null) {
      try { // is the string a filename ?
        is = new FileInputStream ( aZipFile );
      }
      catch (IOException e2)
      { }
    }

    if (is == null) return;
```

```
    zis = new ZipInputStream ( is  );
  }

  public boolean getNextEntry () {
    if (zis != null)
    try
    {
      if (! zeLoopStarted) {
        ze = zis.getNextEntry();
        zeLoopStarted = true;
      }
      else
      if (ze != null) {
        ze = zis.getNextEntry();
      }

      if (ze == null) {
        zis.close();
      }
    }
    catch (java.io.IOException ioe)
    { ze = null; }

    zerLoopStarted = false;
    zerEndOfFile = false;

    return (ze != null);
  }

  public String getEntryName () {
    if (ze != null)
      return ze.getName() ;
    else
      return "";
  }

  public String readLine () {
    try
    {
      if (! zerLoopStarted) {
        zer = new BufferedReader
        (
           new InputStreamReader ( zis )
        ) ;
        zerLoopStarted = true;
      }
      String s = zer.readLine ();
      if (s == null) {
        zerEndOfFile = true;
        zis.closeEntry();
      }

      return s;
    }
    catch (java.io.IOException ioe)
    { return ""; }
  }

  public boolean endOfEntry () {
    return zerEndOfFile;
  }
}
```

CONCLUSION

JavaObj opens new horizons for DATA Step programmers. Using JavaObj is easy, but in some cases designing Java classes for use by way of JavaObj requires creativity beyond normal Java training. A developer fluent in both sides of JavaObj can develop remarkable solutions that benefit from the Power of Java and the Power to Know.

RECOMMENDED READING

"Thinking in Java" - Bruce Eckel, Prentice Hall PTR, ISBN 0-13-659723-8.

"Design Patterns" - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley, ISBN 0-20-163361-2

"Dot Notation and DATA Step Component Objects" - http://support.sas.com/documentation/cdl/en/lrdict/59540/HTML/default/a002587970.htm

"The Java Object and the DATA Step Component Interface" - http://support.sas.com/rnd/base/datastep/dot/javaobj.html

CONTACT INFORMATION

Richard A. DeVenezia
9949 East Steuben Road
Remsen, NY 13438
(315) 831-8802
http://www.devenezia.com/contact.php

Richard is an independent consultant who has worked extensively with SAS products for over fifteen years. He has presented at previous SUGI, NESUG and SESUG conferences. Richard is interested in learning and applying new technologies. He is a SAS-L Hall of Famer and remains an active contributor to SAS-L.