Paper 241-30

Java in SAS®

JavaObj, a DATA Step Component Object

Richard A. DeVenezia, Independent Consultant

ABSTRACT

The experimental JavaObj gives DATA Step programmers the ability to reach into the world of Java. This is a powerful new feature for solving real problems. Imagine, all the Java application programming interfaces (API) and class libraries, placed at your disposal. Several JavaObj examples that interact with the Java 2D^{TM} API are presented to demonstrate the syntax of the DECLARE statement and the methods of JavaObj.

The design of JavaObj creates many situations that require an *adapter* Java class to fully exploit the features of a library. Programming concepts needed to write adapters will be discussed. SAS Programs that heavily use an adapter can suffer from syntax clutter. Using the SAS Macro Language as a remedy is examined.

INTRODUCTION

The Java programming language is arguably one of the most popular and widespread today. The Java 2 Platform, Standard Edition (J2SE) from Sun provides free tools for development and deployment of Java applications. J2SE includes a compiler, **javac**, which creates class files. Class files contain the instructions that a Java Virtual Machine (JVM) uses to perform whatever programming task was coded. A JVM exists when you use the J2SE runner, **java** or **javaw**. A JVM can also be brought into existence by a host application, such as a Version 9 SAS Session. The DATA Step JavaObj is the conduit into this hosted JVM.

The J2SE libraries address a wide variety of topics, such as security, remote processing, input/output, databases, internet and multimedia. Vast numbers of third party libraries cover almost any conceivable topic. Examples using Java 2D API will demonstrate the features of JavaObj and convey a glimmer of how far you can now reach.

COMPONENT INTERFACE

A new feature in SAS version 9 is "The DATA Step Component Interface and Object Dot Syntax"[1]. Presently three objects are available; the experimental `JavaObj` and the production `Hash` and `HashIter`. No facility exists for creating your own component objects.

A component object is dynamically allocated at DATA Step runtime and exists until the step ends or the object is explicitly deleted. The component object is referenced through a variable in the DATA Step program data vector (PDV). The variable is always temporary and retained. Object variables cannot be saved in a data set.

DECLARE STATEMENT

A component object enters the PDV by using the DECLARE statement.

```
DECLARE ComponentObject variable;
```

Object instantiation will occur at a declaration statement having constructor arguments.

```
DECLARE ComponentObject variable ( constructor arguments );
```

1 http://support.sas.com/rnd/base/topics/datastep/dot/javaobj.html

NEW STATEMENT

Use the _NEW_ statement to instantiate a component object and assign its reference to a declared variable.

```
DECLARE ComponentObject variable;
variable = _new_ ComponentObject ( constructor arguments );
```

The number, structure and meaning of the constructor arguments are different for each specific component object. The JavaObj argument types can be Character, Numeric or JavaObj. The pattern of the types is known as the constructors *signature*.

A guard block is usually coded to prevent a extra instances when the DATA Step loops.

```
if _N_ = 1 then myVar = _new_ ComponentObject ( constructor arguments );
```

ASSIGNMENT

Component object variables are references that can be assigned.

```
DECLARE ComponentObject co_one(); * causes an object instantiation;
DECLARE ComponentObject co_two;   * declares an object;
co_two = co_one;  * co_one and co_two now refer to the same object;
```

Assigning an object to another object can cause loss of reference and memory leaks. There is <u>no</u> garbage collection system for deleting unreferenced component objects.

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

OBJECT DOT SYNTAX

Each component object provided by SAS has set of documented methods. The syntax for invoking an object's method is to place a dot between the object variable and the method name. If the method accepts arguments, they are passed in a parenthetical comma separated list.

```
ComponentObject . method ( arguments );
```

JavaObj method argument types can be Character, Numeric or JavaObj. The pattern of the argument types is known as the methods *signature*.

JAVAOBJ

All further discussion will focus on the JavaObj component object.

```
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.*

Examples:

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

In general, the dots (.) of a hierarchy are replaced with slashes (/) and the dots (.) of a nesting are replaced 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
```

CLASSPATH

The environment variable CLASSPATH lists the places that should be searched when a java class is to be loaded into the JVM. The value is a semi-colon (;) separated list of locations. A location can be a relative or absolute pathname of a folder or jar file. A jar file is zip archive that contains other hierarchies of files.

CLASSPATH can be set in wide variety of ways. This lists the most common in order of precedence (lowest number is highest precedence):

1. SAS session command line, -SET CLASSPATH *classpath*
2. SAS session configuration file, -SET CLASSPATH *classpath*
3. Batch file environment variable
4. User environment (stored under HKEY_USERS)
5. System environment (stored under HKEY_LOCAL_MACHINE)

CLASSPATH is examined only at startup; it cannot be changed during the SAS session.

JVM

The JVM that is used within a SAS session is specifed by a system property set within the SAS invocation option JREOPTIONS.

The default JVM is specified in configuration file `...\nls\en\SASV9.CFG`, at the line

```
-JREOPTIONS (-Dsas.jre.home=c:\opt\sas\shared\JRE\1.4.1)
```

Q: Why would you want to change the JVM?

A: Some Java libraries only work for specific versions of a JVM, or perhaps you want to write and run Java code using the most recently released development kit (J2SE5.0)

You can change a system property value by changing the configuration file, or specifying the property on the SAS command line.

```
 -JREOPTIONS (-Dsas.jre.home=path).
```

The jre.home is only examined at startup, it cannot be changed during a SAS session.

PROC JAVAINFO

The java system properties that are affecting your SAS session can be examined by running the JAVAINFO procedure.

```
proc javainfo;
run;
```

```
java.version = 1.4.1
java.vm.version = 1.4.1-b21
...
```

Note: Experimental procedures may have unknown side effects impacting JVM selection.

PRIMITIVE TYPES

JAVA

There are nine primitive types in Java *

Primitive type	***Size***	***Minimum***	***Maximum***	***Wrapper type***
boolean	1-bit	–	–	Boolean
char	16-bit	Unicode 0	Unicode 216-1	Character
byte	8-bit	-2^{7}	$+2^{7} -1$	Byte
short	16-bit	-2^{15}	$+2^{15} -1$	Short
int	32-bit	-2^{31}	$+2^{31} -1$	Integer
long	64-bit	-2^{63}	$+2^{63} -1$	Long
float	32-bit	IEEE754	IEEE754	Float
double	64-bit	IEEE754	IEEE754	Double
void	–	–	–	Void

* Table is from "Thinking In Java" by Bruce Eckel. The numeric types are signed.

DATA STEP

There are two primitive types in DATA Step

Primitive type	***Default Size***	***Minimum***	***Maximum***
number	8-byte		
character (fixed length)	8-char	1-char	32767-char

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.

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.

JAVAOBJ METHODS

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
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 JavaObj will also provide access to `Boolean` and `Char` fields.

Example:

```
j.getDoubleField ('x', x);
j.setDoubleField ('y', 75);
```

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 (see Appendix). Version 9.2 will report a normal error.

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
```

Other

- A Java class may have getter and setter methods; these are separate and distinct from the JavaObj getter and setter methods.
- An adapter method is one that takes its arguments and uses them to call similar or like named methods with a different signature.

SUGI 30 GRAPHICS

This Java class can be used to read an image file from local disk or the internet and display the image in a dialog box.

```
package Sugi30;

import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.image.*;
import javax.imageio.*;
import javax.swing.*;
```

```
public class Graphics
{
   Graphics ()
   {      System.out.println ("Welcome to Graphics"); }

   private String readinSource;
   private BufferedImage readin;

   void readImage (String source) throws Exception
   {
          System.out.println ("Reading "+source);
          readin = null;
          readinSource = null;

          try
          {
                 readin = ImageIO.read (new File(source));
                 readinSource = source;
                 System.out.println (source+" read as file.");
          }
          catch (Exception e1)
          {
                 try
                 {
                        readin = ImageIO.read (new URL(source));
                        readinSource = source;
                        System.out.println (source+" read as URL.");
                 }
                 catch (Exception e2)
                 {
                        throw new Exception (
                           source+" could not be read as a File or URL");
                 }
          }
   }

   void showImage ()
   {      show (readin, "Readin ImageBuffer - "+readinSource); }

   void show (BufferedImage img, String title)
   {
          JOptionPane.showMessageDialog(null, null, title,
                         JOptionPane.PLAIN_MESSAGE, new ImageIcon(img));
   }
}
```

DATA Step uses an instance of Graphics to read an image file and display it in a dialog box.

```
data _null_;
  declare javaobj _g ('Sugi30/Graphics');

  _g.callVoidMethod ('readImage', 'D:\Images\sugi30_logo.png');
  _g.callVoidMethod ('showImage');

  _g.delete();
run;
```

Any output generated by `System.out.println` will appear on the SAS session standard out.

```
Welcome to Graphics
Reading D:\Images\sugi30_logo.png
D:\Images\sugi30_logo.png read as file.
```

MORE JAVAOBJ METHODS - EXCEPTIONS

There is always the possibility that a Java method will throw an exception. There are three JavaObj methods that deal with exceptions in Java.

rc = *javaobj* . `exceptionDescribe` (*state*)

rc	return code - 0 if successful, not 0 otherwise
state	numeric expression. not 0 – report exception messages to the SAS session standard error 0 – do not report exception messages when they occur.

rc = *javaobj* . `exceptionCheck` (*check_var*)

rc	return code - 0 if successful, not 0 otherwise
check_var	numeric variable. contains 1 if prior call★Method caused an exception, 0 otherwise.

rc = *javaobj* . `exceptionClear` ( )

rc	return code - 0 if successful, not 0 otherwise

Clear the exceptionCheck state in the JavaObj. After `exceptionClear` a call to `exceptionCheck` will return with *check_var* set to 0.

Sample – Attempt to show a non-existent image file. An exception check is made after invoking readImage:

```
data _null_;
  declare javaobj _g ('Sugi30/Graphics');
  _g.exceptionDescribe(1);

  source = "D:\Images\sugi30_logo.pix";
  _g.callVoidMethod ('readImage', source);
  _g.exceptionCheck (e);
  if e then do;
    _g.exceptionClear();  * always clear the exception flag;
    put "WARNING: " source "could not be read.";
  end;
  else
    _g.callVoidMethod ('showImage');

  _g.delete();
run;
```

The SAS log will show

```
WARNING: D:\Images\sugi30_logo.pix could not be read.
```

exceptionDescribe was turned on, so the SAS session standard error will show

```
java.lang.Exception: D:\Images\sugi30_logo.pix could not be read as a File or URL
      at Sugi30.Graphics.readImage(Graphics.java:39)
```

JAVA CODING PATTERNS

There are many situations where an iterable Java construct is not directly accessible to JavaObj. A useful pattern for adapting the iteration process is demonstrated here. A private class field tracks the current index. Each call to the get method increments the index and returns a String representation of the element. When the last element is reached, a blank String is returned; signaling the end of the loop.

```
    // An adapter for delivering all the method names of a class
    private int mni = -1;
    String getMethodNames ()
    {
        mni ++;

        Class c = this.getClass();
        if (mni >= c.getDeclaredMethods().length)
        {
            mni =- 1;  // reset private index in preparation of future calls
            return "";
        }
        else
          return c.getDeclaredMethods()[mni].toString();
    }
```

A call to the get method, after the end of loop signal, will cause the iteration process to restart.

JAVA2D

The Java2D library is a rich set of classes for creating and manipulating images. Of special interest is the ability to draw anti-aliased lines, arcs and shapes into an image buffer.

Java2D is class-centric and thus not directly accessible to DATA Step through JavaObj. The adapter **Graphics**, shown earlier, can have many task-centric methods added to it; in essence creating an API that connects DATA Step to Java2D.

Q: SAS/Graph DSGI functions and macros are sufficient for creating arbitrary images.
Why go through the trouble of creating an API style adapter?

A: SAS/Graph is limited to 256 colors and DSGI does not do anti-aliased operations.

A few fields and methods are added to `Graphics` so that an image can be created and saved to disk . The private field `buffer` contains the image state and `g` is the context for affecting the image state. Both of these fields are utilized to make Java functionality accessible to JavaObj. Examine the adapter methods `fillRect` and `drawPolygon`. Since they have double arguments JavaObj will be able to accessible them. The doubles are cast to integers as they are forwarded to a like named Graphics2D method.

```
   private BufferedImage buffer;
   private Graphics2D g;
   void createBuffer (double width, double height)
   {
         buffer = new BufferedImage((int)width, (int)height,
                                           BufferedImage.TYPE_INT_RGB);
         g = buffer.createGraphics();
         setAntiAlias ( 1d );
   }

   void setAntiAlias ( double onoff )
   {
         Object value = ( onoff != 0 )
                        ? RenderingHints.VALUE_ANTIALIAS_ON
                        : RenderingHints.VALUE_ANTIALIAS_OFF;

         g.setRenderingHint ( RenderingHints.KEY_ANTIALIASING, value );
   }

   void setColor (String color /* rgb hex form such as "0000ff" */)
   {
         Color c = new Color ( Integer.parseInt(color,16) );
         g.setColor (c);
   }

   void fillRect ( double x, double y, double width, double height )
   {  g.fillRect ( (int) x, (int) y, (int) width, (int) height ); }

   void drawPolygon ( double x[], double y[] )
   {
         int[] xi = new int[x.length];
         int[] yi = new int[y.length];

         // copy double arrays into int arrays
         for (int i=0; i<x.length; i++) { xi[i] = (int) x[i]; }
         for (int i=0; i<y.length; i++) { yi[i] = (int) y[i]; }

         g.drawPolygon ( xi, yi, Math.min(xi.length,yi.length) );
   }

   String saveAs ( String type, String name ) throws Exception
   {
         File file = new File(name);
         String savedAs = file.getCanonicalPath();

         ImageIO.write (buffer, type, new File(savedAs));

         return savedAs;
   }

   void showBuffer ()
   {
         show (buffer, "Active ImageBuffer");
   }
```

The DATA Step creates a 400x300 pixel image depicting two triangles on a baby blue background. One of the triangles is antialiased; the other is not.

```
data _null_;
  declare javaobj _g ('Sugi30/Graphics');
  gfile = 'D:\Images\redRect';
  gtype = 'png';
  gout = gfile || '.' || gtype;

  length savedAs $200;

  array x1[3] ( 50, 350, 350 );
  array y1[3] (260,  60, 260 );

  array x2[3] ( 50, 350,  50 );
  array y2[3] (240,  40,  40 );

  _g.exceptionDescribe (1);
  _g.callVoidMethod ('createBuffer', 400,300);
  _g.callVoidMethod ('setColor', 'eeeeff');     * baby blue;
  _g.callVoidMethod ('fillRect', 0,0,400,300);
  _g.callVoidMethod ('setColor', '000000');
  _g.callVoidMethod ('drawPolygon', x1, y1);
  _g.callVoidMethod ('setAntiAlias', 0);
  _g.callVoidMethod ('drawPolygon', x2, y2);
  _g.callVoidMethod ('showBuffer');
  _g.callStringMethod ('saveAs', gtype, gout, savedAs);
  _g.callVoidMethod ('readImage', gout);
  _g.callVoidMethod ('showImage');

  _g.delete();
run;
```

BUILDING AN API

The code patterns in the sample indicate a graphic heavy DATA Step will be dominated by the wallpapering text `_g.callVoidMethod ()`. The highly repetitive nature of this syntax in the code is white noise that distracts from the logic that is being applied to create some visualization. The noise can be removed by creating macros that generate the source code that invoke the JavaObj methods that invoke the SUGI30/Graphics methods that invoke Graphics2D methods.

Consider the process of drawing into some buffer. From an artisans perspective the buffer is a canvas. This perspective can be used when creating macros. The canvas metaphor in DATA Step is similar to the graphics context while working with DSGI. Since these macros are quite similar to DSGI functions and macros, I named them jDSGI.

If you accept that only one canvas can be worked on at a time, the macros can rely on a global macro variable `jdsgi` for containing the name of a DATA Step variable that should be used in all macros subsequent to `canvas_create`. The final statement in each macro is lacking a semi-colon. This design feature forces the macro user to end each macro call with a semi-colon, ensuring source code that appears statement-centric.

```
%macro canvas_create (javaobj, width, height, color);
  %global jdsgi;

  %let jdsgi = &javaobj;

  declare javaobj &jdsgi ('SUGI30/Graphics');

  &jdsgi..exceptionDescribe(1);

  %createBuffer (&width, &height);
  %setColor (&color);
  %fillRect (0,0,&width,&height);
  %setColor ('000000')
%mend;

%macro canvas_delete();
  &jdsgi..delete();

  %symdel jdsgi;
%mend;

%macro canvas_show();
  &jdsgi..callVoidMethod ('showBuffer')
%mend;

%macro createBuffer (width, height);
  &jdsgi..callVoidMethod ('createBuffer', &width, &height)
%mend;

%macro setColor (color);
  &jdsgi..callVoidMethod ('setColor', &color)
%mend;

%macro fillRect (x,y,width,height);
  &jdsgi..callVoidMethod ('fillRect', &x, &y, &width, &height)
%mend;

%macro setAntiAlias ( state );
  &jdsgi..callVoidMethod ('setAntiAlias', &state)
%mend;
```

```
%macro drawPolygon (xpoints, ypoints);
  &jdsgi..callVoidMethod ('drawPolygon', &xpoints, &ypoints)
%mend;

%macro canvas_saveAs (basename, type, savedAs);
  &savedAs = repeat(' ', 300);
  &jdsgi..callStringMethod ('saveAs', trim(&type)
                                    , catx('.',&basename,&type), &savedAs);
  put &savedAs=
%mend;

%macro readImage (source);
  &jdsgi..callVoidMethod ('readImage', &source)
%mend;

%macro image_show();
  &jdsgi..callVoidMethod ('showImage')
%mend;
```

USING THE MACROS

The SASAUTOS option is the recommended technique for making macros automatically available to a SAS session. Define all the macros of the API in a single source file `jsdgimac.sas`. The first invocation `%jsdgimac` will cause the SASAUTOS system to automatically submit `jsdimac.sas`, causing all the API macros to be compiled as a side effect. Note: this is same technique used to prepare DSGI macros via `%annomac`.

The earlier sample, recoded using jDSGI macros has a much cleaner look:

```
data _null_;
  gfile = 'D:\Images\sample';
  gtype = 'png';

  length savedAs $200;

  array x1[3] ( 50, 350, 350 );
  array y1[3] (260,  60, 260 );
  array x2[3] ( 50, 350,  50 );
  array y2[3] (240,  40,  40 );

  %canvas_create ( _g, 400, 300, 'eeeeff' );
  %drawPolygon (x1, y1);
  %setAntiAlias (0);
  %drawPolygon (x2, y2);
  %canvas_show ();
  %canvas_saveAs (gfile, gtype, savedAs);
  %readImage (savedAs);
  %image_show ();

  %canvas_delete ();
run;
```

There are many other Graphics2D methods that can be surfaced to JavaObj by an adapter class such as `SUGI30.Graphics`. A robust set of methods and macros can be found at the authors website[2].

CONCLUSION

JavaObj opens new horizons for the DATA Step programmer. Using JavaObj is easy, but in some cases designing Java classes for use by way of JavaObj requires creativity beyond normal Java training. Development of an API can be time consuming, but the effort is justified when the API is accepted by a programming community as a stepping stone to greater things.

JavaObj is experimental and subject to improvement. Learn about this new technology and plan for the future. Give it a try and send your impressions and ideas to the technical support team at support@sas.com, subject: JavaObj.

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

ABOUT THE AUTHOR

Richard A. DeVenezia is an independent consultant and has worked extensively with SAS products for over ten years. He has presented at previous SUGI, NESUG and SESUG conferences. Richard has an active interest in learning and applying new technologies. He is an active contributor on SAS-L.

SAMPLE APPLICATION

A robust library of jDSGI macros can be downloaded from the authors website.
http://www.devenezia.com/downloads/sas/macros/?m=jdsgimac

Java source code and sample SAS usage of a the macros can be found at
http://www.devenezia.com/downloads/sas/samples/#jdsgi

2 http://www.devenezia.com/downloads/sas/macros/?m=jdsgimac

APPENDIX

V9.1 SETTER TYPECASTING

SAS numbers are eight byte double precision. The internals of JavaObj setters must **cast** the SAS value passed to the specified Java type. Casting can cause truncation and unexpected values.

Consider this class:

```
class Sample
{
   byte index = 0;
}
```

and this DATA Step

```
data _null_;
  declare JavaObj j ('Sample');
  indexIn = 128;
  j.setByteField ('index', indexIn);
  j.getByteField ('index', indexOut);
  put 'NOTE:' (index:) (=);
run;
```

```
NOTE: indexIn=128 indexOut=-128
```

WHAT HAPPENED!

A JavaObj setter method was passed a SAS 8 byte numeric with value 128 which is outside the range of Java primitive byte (-128..127). The JavaObj component subsystem typecast the 128 value to an 1 byte signed integer, whose eight bits are `10000000`. This value is assigned to the Java field `index`. Unfortunately, the two's complement interpretation of the 8 bits is the value -128.

When the JavaObj getter method was called, the signed byte value of field `index` was obtained by the JavaObj subsystem, which cast it to a SAS numeric type and that value was placed in the DATA Step variable `index`.

Thus 128 in and -128 out.

There are no warnings when oversized typecasting occurs. In addition to typecasting concerns, passing SAS values of overly large magnitude to JavaObj setters can lead to errors such as:

```
ERROR: An unknown, abnormal error has occurred during execution
ERROR: Termination due to Floating Point Exception
```

This error will occur in Windows platforms if you pass `setByteField`, `setShortField`, or `setIntField` a SAS value $< -2^{31}$ or $> 2^{31}-1$. `setLongField` will error when the SAS value is $< -2^{63}-1024$ or $> 2^{63}-513$. The -1024 and -513 offsets are due to limitations of integer representation in the SAS numeric type. An eight byte numeric can only represent exactly every integer upto 2^{53} (which is smaller than the largest Java long)
