

# Benefits of JAVA Within SAS: Exploring the Differences of the new JavaObj and SAS/IML Studio

Diana Shealy, California Polytechnic State University at San Luis Obispo, San Luis Obispo, CA

## Abstract

Released into production in SAS® 9.2, the JavaObj is a new interface to Java programs and classes. Yet, this is not the first and only way to access Java's abilities for SAS. Another way to work with Java using SAS is with the programming language for SAS/IML Studio called IMLPlus. The purpose of this paper is to compare and contrast these two methods for using library-defined or user-defined Java classes. While this paper does not concentrate on Java's massive GUI and graphics packages, it does focus on using Java programs to calculate complex equations and algorithms. There are two main benefits to using Java to write such algorithms: code reuse, if the code is already written in Java, and the flexibility of the Java language. This paper will walk the user through basic Java code and how to access that code through both the JavaObj and through IMLPlus, as well as educate the user on both the advantages and limitations of each method.

## Introduction

The idea of object orientation is one more level of abstraction when it comes to programming, and it allows programmers to imagine problems in more realistic ways because similar ideas can be grouped together. SAS has been interested in adding the principles of object orientation into the language for some time. One way to be able to utilize this methodology in SAS is to be able to use a programming language, like Java, that is object oriented. However, object oriented models not the only benefit to utilizing Java within SAS.

A major benefit is access to the expansive Java libraries. These libraries house code that can be used for GUI production, data structure use, and commonly used functions. The idea for the Java library comes from the idea of code reuse. Why rewrite code again and again? If a complex algorithm is already written and debugged in Java, why try to translate that code into SAS. It is easier to access that Java code and use it in SAS through the methods discussed in this paper.

The flexibility of the Java language also makes it attractive to programmers. While some programming languages have their own niches, Java has the ability to be practical in a variety of uses. It is far easier to write an algorithm for a difficult process in Java than in SAS. Also, in Java, you can create data structures that are useful to you to aid in your programming needs. For the SAS programmer, Java's GUI abilities and its use for complex algorithms are particularly attractive. While this paper concentrates on Java for algorithm use, it should be noted that GUI creation is another practical use.

An untapped, but potentially exciting, use of Java within SAS is to be able to generate simulation data of large, complex processes and then take that simulated data and analyze it using SAS. Areas where this type of simulation may be useful would be high-throughput computing, bioinformatics, and financial engineering.

There are two main ways of using Java for algorithms in SAS. The first is to use it in SAS/IML Studio, here on referred to as IML. The second is the new JavaObj that can be used in the DATA step.

This paper will describe basics of these methods as well as some important key features of the Java language to keep in mind when creating your Java programs. However, the focus of this paper will not be a full description of Java by any means and does not cover any syntax of Java. To describe the Java language in detail is beyond the scope of this paper. Rather, this paper is aimed at people who have basic programming skills, such as logic, loops, and arrays.

Lastly, this paper will highlight some details of both the IML and JavaObj methods to keep in mind before choosing one over the other. The pros and cons of each method were discovered when learning these methods are helpful information to the new Java user.

## Basics of Java

Java programs consist of one or more classes. A **class** is a data type that is defined by the programmer. The class defines what makes up that data type and any **methods**, or functions, which can act on that data type. An **object** is an instance of some class type.

There are only two types of data types in java: **primitives** and objects. Primitives are built into the Java language. Figure 1.1 contains a listing of Java's primitive types, as well as their default initialization value. Unlike the SAS language where there are only numeric and character types, numeric types in Java are defined differently based on their memory capacity. The Java equivalent to SAS's numeric type is the double. Also important to note is the char type in Java refers to a single character and not a string of characters. The Java language does support a String class that handles the string data type, which will be discussed below.

Objects are the other data type in Java. The default initialization for any object is to **null**, a void reference. When an object is created, a reference, or pointer, is created for that object, but that pointer does not refer to anything. To create an instance of an object, you must call one of that object's **constructors**, a special type of method that gives instructions on how to create an instance of the object, as well as using the Java keyword `new`. The following sample code will create an instance of `SomeClass`:

```
SomeClass anInstance = new SomeClass();
```

`SomeClass`' constructor has no parameters, but it is possible to have constructors with parameters to initialize instance fields. Objects can contain:

- **Instance fields**, which is any data that pertains to that object. An instance field can be of primitive or object type.
- **Methods**, the functions of the Java language, these can act on the class object.
- **Constructors**, if no constructor is defined, the objects instance fields will be initialized to their default initialization value, including any object instance fields that will be initialized to null.

**Figure 1.1 – Primitive Types**

Type	Definition	Initialization value
Byte	8-bit signed two's complement integer	0
Short	16-bit signed two's complement integer	0
Int	32-bit signed two's complement integer	0
Long	64-bit signed two's complement integer	0
Float	single-precision 32-bit IEEE 754 floating point	0
Double	double-precision 64-bit IEEE 754 floating point	0
Boolean	Has only two possible values: true or false. Used as a flag to track true/false conditions	False
Char	single 16-bit Unicode character	'\u0000'

## Object Orientation

The basis of Java is the idea of object-orientation. Classes in Java can be based and defined off of other already defined classes. This is the idea of **inheritance** in Java. Inheritance is a mechanism for adding more functionality to new classes by adding methods and fields to other classes. The more specialized class that inherits the more general class' contents is called the **subclass**, while the more general class that is extended from is called the **superclass**. This is done with the Java key word `extends` when declaring the class. For example:

```
public class SubClass extends SuperClass
```

would create a class called `SubClass` that would inherit the instance fields and methods of `SuperClass`, as well as contain any newly defined instance fields and methods. All classes in Java extend the `Object` class either directly or through its defined superclass. See Figure 1.2 for more information on the `Object` Class.

## Methods: Useful Information

Creating methods in the Java language are fairly straight forward and follow the below basic formula:

```
[return type] methodName([parameter type] paramname, ...)
```

The above example is called the method's signature. If the method does not return any type, you use the Java keyword `void`. If the method does not need any parameters, then the inside of the parentheses are left empty. You can define a method's accessibility within the method signature. See table 1.3 for the possible access types. A method defined as `public` allows objects outside the class to call that function, while `private` methods can only be used by that instance of that class. Another possible option is to declare the method `static`. `Static` is a Java keyword that when used with a method means that the method does not, and cannot access, that object's instance variables. The appropriate time to declare a static method is when that method "stands alone" within that object, meaning that all the information that method needs to perform the operation is given through that method's parameters. The `Math` class gives many examples of static methods. Static methods are methods that belong to that class, where non-static methods belong to an instance of that class.

## Instance Fields: Useful Information

Instance fields are analogous to an object's "guts". They are what define that particular instance of a class. Most of the time, once you create an object and initialize it's fields, you want to be very careful who has the ability to get and change those fields. Like methods, instance fields can have access privileges and can also be declared static.

If a field is declared `private`, you must use **accessor** and **mutator** methods to set and retrieve an instance field. In general, an accessor method retrieves an instance field's information, while a mutator method changes an instance field's information. These programming strategies helps protect the instance fields by only allowing access though these types of methods.

Like methods, fields that are declared static belong to a Class rather than individual objects.

Figure 1.2

### The Object Class

- The root of all class hierarchy
- Every class has the Object class as a superclass, including arrays
- A subclass of Object can rewrite any method of the Object class
- Important methods inherited from the Object Class:
  - `clone` – creates and returns a copy of this object
  - `equals` – returns whether two object are equal to one another.
  - `getClass` – returns the class type of an object
  - `hashCode` – returns a hash code value for an object
  - `toString` – returns a string representation of an object

Figure 1.3 – Java Access \*

	Class	Package	Subclass	World
Public	Yes	Yes	Yes	Yes
Protected	Yes	Yes	No	No
Private	Yes	No	No	No
No modifier	Yes	No	No	No

\*Controlling access to the members of a class. (n.d.). Retrieved from <http://download.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html>

## Using Java within SAS/IML Studio

### Declaring Primitives and Objects

To declare either a primitive or an object within IML Studio using the IMLPlus language, you use the keyword `declare`. The below statement shows the basic syntax for declaring variables:

```
declare [type/object] [name];
```

An important thing to note is that any variable that is not initiated with a declare statement is of a matrix data type.

### Initializing a Variable

To initialize a primitive, you can either include the initialization within the declare statement, or separately as seen below:

```
declare int a = 10;
/** OR **/
declare int a;
    a = 10;
```

Initializing an object is similar, except that you use the new operator. As defined above, the new operator allows you to create a new instance of a class. If declaring an object, you would then call the appropriate constructor. Below is an example of the initialization of an instance of the Point class:

```
declare Point pt = new Point(1.4, 4.5);
```

Special care needs to be taken with arrays. Remember, in Java, arrays are considered objects and must be created using the new keyword. The code below all creates the same integer array but there are important differences between the three:

```
/* Method One */
declare int[] a;
a = {1, 4, 6};
/* Method Two */
declare int[] a = new int[3];
a[0] = 1;
a[1] = 4;
a[2] = 6;
/* Method Three */
Declare int[] a = {1, 4, 6};
```

In method one, a is first created as a matrix type and is converted to the Java int array type in the following line. This can cause efficiency issues due to the necessary conversion, which makes method one not preferred. There is only a slight difference between methods two and three. Method two creates an instance of a Java integer array and then initializes each index as necessary, whereas method three creates and initializes the array within the declare statement. The use between these two is that one would generally prefer method two when initializing a large array, since a loop can be utilized to get to each index. However, it can be easier to use method three to declare and initialize small arrays with less code.

Strings can also be treated similarly to arrays as they can be initialized either by using the new keyword and a constructor or by setting the string to some value as per method three. For example:

```
/* Method Two for Strings */
declare String str1 = new String("Hello, World");
/* Method Three for Strings*/
Declare String str1 = "Hello, World";
```

### Importing Types and Packages

One of the main benefits of using Java within SAS is having the ability to access and use the massive Java library as well as be able to use your own Java programs. To be able to use those classes within your IML program, you need to use the import statement:

```
/* To import a Java Library */
Import java.[java package].[class name];
```

```
/* To import your own package */;
Import [Pathway to your package's location];
```

Packages only need to be declared once. A helpful shortcut to use if you are going to need access to multiple classes in the same package is to use following syntax:

```
Import java.util.*;
```

This line of code will import all classes that fall within the util package.

## Accessing Methods

To access a class' methods, the IMLPlus language uses the dot notation exactly like Java. For example, if you have an instance of SomeClass and would like to use SomeClass' methods, you would use:

```
x = instanceOfSomeClass.method();
y = instanceOfSomeClass.method(param1, param2);
```

IMLPlus does allow you to use methods that have multiple signatures, meaning methods that have the same name but a different number of parameters.

For static methods, instead of using the name of the instance of an object, you simply refer to the class name itself. For example, the Math class contains a large amount of static methods including an absolute value method. To use that method you would:

```
y = Math.abs(-19);
```

One important note about accessing methods in IMLPlus is that you can only access methods that are declared `public`. Methods with other modifiers, such as `private` and `protected`, cannot be accessed.

## Using Java through the JavaObj

### Declaring the JavaObj

Similar to IMLPlus, to create a JavaObj in the DATA step, you must use the declare statement:

```
declare javaobj x ("Appropriate Pathway to Java Class");
/* OR */
declare javaobj x;
x = _NEW_ javaobj("Appropriate Pathway to Java Class");
```

A critical note to the declaration of the JavaObj is that you must include the entire pathway to the class as well as the class' name surrounded by quotes. For example, if you need to use the Point class, you would input "java/awt/Point".

If a class has a constructor that contains parameter, you would put the appropriate number of parameters inside the parentheses separated by commas as such:

```
declare javaobj pt("java/awt/Point", 3.6, 8.9)
```

### Calling Methods

To access a class' method through the corresponding JavaObj, you must use the JavaObj methods. You use the appropriate JavaObj method depending on the class' method's return type. For example, if you had a method that returned an integer, you would use:

```
[javaobj].callIntMethod("nameOfMethod").
```

Like with declaring a JavaObj, you must wrap the class' method's name with quotes and pass it as a parameter to the JavaObj method. If the method contains parameters, they follow the name of the method, separated by commas. The last parameter always contains the name of the variable to place the return value. If the method returns nothing, a void method, then no return variable is necessary. Below are some possible examples.

```
/* Call the sum method of some class and store the answer into ans1 */
```

```

x.callIntMethod("sum", ans1);
/* Call the area method of some class with a parameter of 18 and store into
ans2*/
y.callDoubleMethod("area", 18, ans2);
/* Call the add method of some class and input 39*/
z.callVoidMethod("add", 39);

```

If a method is static, you use the corresponding static JavaObj methods. Their use is similar to those explained above. For example:

```
a.callStaticDoubleMethod("sqrt", 5, ans3);
```

With the JavaObj, it is important to note that all Java primitives, except char, are appropriately mapped to the SAS numeric data type. The Java char data type is not supported. Java methods that return a String object are supported and map to the SAS character data type. At this time, besides the String class, you cannot call Java methods that return objects.

## Setting and Retrieving Instance Fields

Proper Java etiquette states that the instance fields of a class should be set to private to avoid any possible contamination. However, this makes setting and retrieving instances fields slightly more difficult. You would need to use accessor and mutator methods to do change and get instance fields; these two types of methods were explained briefly in the above Java basics section of this paper.

To set an instance field, you would call the appropriate JavaObj method and pass the mutator method name and parameters as described in the Calling Methods section above.

To retrieve an instance fields, you would call a JavaObj method, pass the accessor method name and the name of the variable to store the field.

However, if you do decide to allow instance fields to be set to public, it is much simpler to set and get an instance field by using the JavaObj instance fields methods. The syntax is similar to calling a class' methods, except you pass the instance field's name as the first parameter. The following parameter depends on whether you are setting or retrieving an instance field. For retrieving an instance field, the second parameter is the return variable, whereas the second parameter of a setting an instance field is the value you wish to set the field to. For example:

```

/* To get an instance field */
x.getIntField("x", xVal);
/* To set an instance field */
x.setStringField("str", "Hello, World");

```

## Benefits and Limitations of the Two Methods

The two methods share their own sets of benefits and limitations. Depending on what SAS products you use, and your familiarity with Java and SAS will dictate with method will be most appropriate.

The largest benefit of accessing Java through IML is how similar the syntax is to Java. It has similar touch and feel when writing code and does not feel like you are programming two completely different languages. In fact, much of the syntax in IML is exactly the same as in Java. For the SAS user who is a novice Java programmer this is a huge benefit.

As for the JavaObj, it seems that the syntax of the JavaObj is its worst limitation. While certain aspects are the same, such as the use of dot notation, it is cumbersome to have to remember all of the JavaObj's methods and use of those methods as well as the methods of the Java class you are using. While the syntax is not difficult, there are a lot of little details that must be remembered, like wrapping class or method names in quotes or that the return variable must be the last parameter. For newcomers, all of these little details can be frustrating.

That said, what the JavaObj does well is its ample and simple documentation. When learning how to use the JavaObj, the documentation was easy to find, follow and read. Sample code was clear, and possible limitations were well described. The same cannot be said about the IML method. In fact, it was quite difficult to find any documentation on how to precede using Java in IML. The only documentation available was within IML Studio's help pages. While the IML Studio's documentation was very clear, it was nearly impossible to find.

Both methods have their own quirks that limit their usefulness. In IML, one must be careful to initialize variables properly or you can have costly efficiency errors. In the JavaObj, you cannot return objects from method calls. It is also unclear in both methods how to handle generic classes, like LinkedList or Vector. Since one possible use of Java within SAS is to be able to use data structures, we need to be able to use these generic classes. Formal documentation on the use of Generics in both IML and JavaObj would be incredibly helpful.

## Conclusion

Object orientation brings a new light to SAS, and can help organize data in new ways that will make analysis more interesting and faster. Using Java in SAS, algorithms that could take hundreds of lines within SAS, are far simpler and are still able to be utilized within SAS. The two methods described in this paper are just an introduction of how to use Java within SAS. Additionally, the small section on the basics of Java is just a grain of sand compared to all there is to know. The JavaObj seems most useful for use with data structures while the IML method seems easier to use for those hoping to use Java for algorithms. Hopefully, by listing some small benefits of each method, programmers can choose the best fit for themselves. For the SAS programmer interested in adding Java to their tool box, the possibilities are endless.

## References and Recommended Reading

- Horstmann, CS. (2001). *Big java: programming and practice*. Wiley.
- *The java tutorials*. (2011, July 20). Retrieved from <http://download.oracle.com/javase/tutorial/>
- *The java object and the data step component interface*. (2011). Retrieved from <http://support.sas.com/rnd/base/datastep/dot/javaobj.html#dcl>
- DeVenezia, RA. (2005). Java in sas@ javaobj, a data step component object. *Proceedings of the SUGI 30*, <http://www2.sas.com/proceedings/sugi30/241-30.pdf>

## Contact Information

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

Diana Shealy

Student- California Polytechnic State University at San Luis Obispo

San Luis Obispo, CA 93407

Phone: (805) 423 - 4560

E-mail: [dianashealy@gmail.com](mailto:dianashealy@gmail.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.