Paper 64-28

Java™ Syntax for SAS® Programmers

Don Boudreaux, SAS Institute Inc., Austin, TX

ABSTRACT

As SAS coders are being asked to deploy their results on the web, they are challenged with learning any number of new technologies, scripting languages, and programming languages. In the cases where this effort will involve Applets, Servlets, or JavaServer Pages, the programmer will be faced with a rather complex object-oriented programming language to understand – Java. Obtaining a level of proficiency with Java involves a considerable investment in time and effort. However, a SAS programmer using webAF to create dynamic web applications with JavaServer Pages might only initially need to understand the basic syntax of Java in order to write short embedded code segments (scriptlets). Assuming this specific scenario defines the programming context, this paper will investigate basic Java syntax from a SAS perspective.

INTRODUCTION

An html file is a text file, with an extension of `.html`, that contains text and html tags. It is requested from a web server and is sent directly to the client and rendered by a browser. A JavaServer Page (JSP) is a text file, with an extension of `.jsp`, which contains text, html tags, and JSP tags. It is requested from a web server through a special server-side program called a web container (i.e. Tomcat) that processes the JSP tags. The original text and html along with any text and html that resulted from the processing done by the web container is sent to the client and rendered by a browser (see Figure 1):

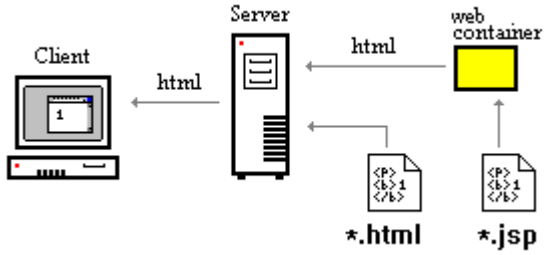

Figure 1. JSP Processing

The web container provides background data objects and services to the JSP – like supporting special tags that allow a programmer to insert Java or SAS processing capabilities directly into the document. Consider the following JSP, besides text and html tags it contains special JSP tags with embedded Java code – a Java scriptlet:

```
   text and html tags  <br />
   <hr />
<%
   // Java code within JSP
   // scriptlet tags.
%>
```

Without getting into the technical specifics or the pros and cons of this type of server-side processing, it is of interest to note that the level of Java coding needed to produce dynamic results with scriptlets can be rather basic. Unlike coding Java Applications, Applets, or Servlets - which all require the programmer to develop Java classes and understand the principles of object-oriented programming, scriptlets may, in many cases, only involve basic Java language syntax and the use of predefined classes. Consequently, a SAS programmer using webAF to create dynamic web applications with JavaServer Pages might initially only need to understand the basic syntax of Java statements, expressions, method calls, and control structures.

BASIC ELEMENTS

Simple Java statements look similar to SAS statements. They typically start with a keyword or an identifier and end with a semicolon. They can contain identifiers, data values, operators, and method (function) calls. There are slight syntax differences between Java and SAS in operators and control structures. And, it is necessary for a SAS programmer to become accustomed to the syntax used in Java to access classes and objects. These topics will be addressed in the following sections.

IDENTIFIERS

Any symbolic name that refers to a class, attribute, method, parameter, or variable in Java is called an identifier. Java identifiers start with a letter, underscore, or currency symbol. They continue with any number of letters, underscores, currency symbols, or numbers. Additionally, identifiers are case sensitive and cannot contain Java keywords or reserved words (for a complete list see reference [3] page 6). But, from the perspective of a SAS programmer, almost any valid SAS name would be a valid identifier in Java.

DATA TYPES AND VALUES

In SAS there are only two data types: character and numeric. In Java there are eight primitive data types. Shown below are their names, value ranges, and their default initial values.

Type	Value Range	Default
boolean	true, false	false
char	one 16-bit character	'\u0000'
byte	8-bit integer: -2^7 to 2^7-1	0
short	16-bit: -2^{15} to $2^{15}-1$	0
int	32-bit: -2^{31} to $2^{31}-1$	0
long	64-bit: -2^{63} to $2^{63}-1$	0L
float	32-bit floating-point	0.0f
double	64-bit floating-point	0.0d

Note the use of the suffix `L`, `f`, and `d` (written in either lower or upper case) to denote long, float, and double values. It is also the case in Java, as a “strongly typed” language, that all identifiers must have their types explicitly declared. In SAS, declaring data types is typically not required. Consider the following scriptlet where four new variables are declared, created, and initialized:

```
<%
   boolean b = true ;
   int     i = 1 ;
   double  x = 1.23 ;
   char    inSingleQuotes = 'a' ;
   String  inDoubleQuotes = "a Java String" ;
%>
```

Note that a Java String was defined in the scriptlet. Strings are not a primitive data type and will be discussed later.

COMMENTS

Besides "normal" statements, Java allows programmers to insert comments within their code. Shown below are the most common comment styles used in Java:

```
   /* C style multiple-line comment */
   // C++ style line comment
```

Note that both SAS and Java use the C style multiple-line comments. The C++ style line comment syntax will initially seem odd to a SAS coder, as it does not seem to have an explicit ending. In fact, everything from the initial double slash through the end of the physical line is the comment.

OPERATORS

Many of the operators used in SAS are also commonly used in Java and have the same meaning. However, there are some differences and additions that a SAS coder will need to understand.

SIMPLE MATH

SAS and Java use standard operators for addition, subtraction, multiplication, and division. Differences show up with modulus and power calculations. For calculating a remainder, SAS uses the `mod()` function while Java uses the `%` operator. For calculating a number to a power, SAS uses the `**` operator and Java has a method (function).

	SAS	Java
add	`+`	`+`
subtract	`-`	`-`
multiply	`*`	`*`
divide	`/`	`/`
remainder	`mod()`	`%`
power	`**`	`Math.pow()`

ASSIGNMENT

For simply assigning values to variables, SAS and Java both use the equals sign. But, Java additionally has a complete set of operators that perform any basic mathematical operation followed with an assignment:

	SAS	Java
assignment	`=`	`=`
add then assign	`+ exp ;`	`+=`
subtract then assign	`+ -exp ;`	`-=`
multiply then assign		`*=`
divide then assign		`/=`
mod then assign		`%=`

With SAS, although there is no operator, it is also possible to use the sum statement to add or subtract the value of an expression into a variable.

INCREMENT AND DECREMENT OPERATORS

Adding and subtracting by one is so common a task that Java uses special operators to accomplish this. Incrementing is done with the `++` operator and decrementing with the `--` operator. These operators can be placed either in front or behind a variable. As a prefix, the incremented (or decremented) value is used in the expression and then saved. As a suffix, the old value is used and then the incremented (or decremented) value is saved. Consider the following scriptlet example:

```
<%
   int x = 0 ;
   int y = 0 ;
   y = x++ ;     // result: y=0, x=1
   y = ++x ;     // result: y=2, x=2
%>
```

Both variables are declared as integers and initially given a value of zero. Then `y` is set to zero using the original value of `x`, after which `x` is given a value of one. Then `x` is again incremented, after which the new value is used to assign two into `y`.

RELATIONAL AND LOGICAL OPERATORS

For most simple relation and logical tests, SAS and Java use the same symbols. However, besides SAS allowing mnemonic text versions of the operators, there are a few differences:

	SAS	SAS	Java
less than	LT	`<`	`<`
less than or equal	LE	`<=`	`<=`
greater than	GT	`>`	`>`
greater than or equal	GE	`>=`	`>=`
logical and	AND	`&`	`&`
logical or	OR	`|`	`|`
equal	EQ	`=`	`==`
not equal	NE	`^=`	`!=`
not	NOT	`^`	`!`
logical xor			`^`

What should initially catch a SAS coder's eye is the use of the double equal signs and the different use of the caret "^" symbol. In Java, the equals sign always and only denotes assignment; the double equal sign denotes a logical test for equivalence (with primitive data types). The caret "^" in SAS is one of the symbols used for NOT (although tilde "~" can also be used). In Java it is used to denote the logical test XOR. There is no equivalent operator in SAS for XOR, although a definition and "work around" would be the SAS expression:

```
((A OR B) AND NOT (A AND B)).
```

One further distinction between SAS and Java is the nature of the expressions that can be used in logical and relational testing and the data type of the results. In SAS, the expression `(x > 1)` will yield a value of 0 for false or 1 for true. In Java, any logical or relational test must evaluate to and result in a boolean value of `true` or `false`.

CONDITIONAL OPERATORS

Besides having AND and OR operators, Java also has conditional AND (using `&&`) and conditional OR (using `||`) operators. These operators allow for sequential processing of the expressions within a test. Consider:

```
<%
   int x = 0 ;
   int y = 0 ;
   boolean b = ((x==1)&&(++y > 1)) ;
   // result: x=0, y=0, b=false
%>
```

In this example, the term `(x==1)` is `false`. The Boolean variable `b` is set to `false` and the expression `(++y > 1)` is ignored. Only if `(x==1)` is true would the expression `(++y > 1)` be evaluated. Conversely, with conditional OR tests, if the first expression is true, the second expression is ignored. This behavior can be used to save on processing or to "shield" the second term from evaluation. There are no comparable operators to do this in SAS, but the same result can be generated with nested `if` logic:

```
data _null_ ;
   x = 0 ;
   y = 0 ;
   b = 0 ;
   if x=1 then do ;
      y = y + 1 ;
      if y > 1 then b = 1 ;
   end ;
run ;
* result: x=0, y=0, b=0 (false) ;
```

JAVA METHODS

Java methods are similar to SAS functions in their use, but they involve a distinctly different syntax and require the programmer to have a basic understanding of classes and objects. We'll start with a brief discussion of classes and objects.

JAVA CLASSES AND OBJECTS

Java, as a programming language, is actually considered a "small" language. It has relatively few basic language elements. However, it does utilize a large number of predefined classes and it allows a programmer to develop their own classes. Even when our need is only to develop short Java code segments using preexisting classes, it is essential to have a basic understanding of Java classes. To this end, consider the following "over-simplified" definition: a class is a "bundled" collection of variables (data /attributes) and methods (functions). Note the following class definition:

```
class ExClass {
     int x = 1 ;
     int y = 2 ;

     int getSum() {
          return x+y ;
     }
}
```

The code defines a class called Ex`Class`. It contains two variables or attributes (`x` and `y`) and it defines one function or method (`getSum()`). The attributes contain integer values that the `getSum()` method, when called, would add together. To use this class it would be necessary to first make an object or instance of this class. This could be done with the following Java statement:

```
ExClass obj = new ExClass() ;
```

It defines an object called `obj` that will contain a copy of `x`, a copy of `y`, and have access to the method `getSum()`. In a way, we can consider a class as a complex or non-primitive data type. The syntax to access the parts of the object would look like the following:

```
obj.x            to access x: 1
obj.y            to access y: 2
obj.getSum()     to calculate x+y: 3
```

These "dot notation" expressions use the name of the object prefixed to the variable or method to access the elements inside of the object. On a terminology note, the variables `x` and `y` would specifically be called instance variables. Each identifier of this complex type (instance of the class) would get its own copy of the data.

It is also possible, with the keyword `static`, to define class level variables that hold a single "global" copy of the data. The class level variable values are shared by all the instances of the class and can be used independently of any instance of the class. In `ExClass`, changing the definition of the variable `x` to begin with the keyword `static` would define it as a class level variable. Then, either the class name `ExClass` or the object name `obj` could be used with the variable name x to reference the value of one - using the syntax `ExClass.x` or `obj.x` .

A NUMERIC METHODS EXAMPLE

One key to working with a language like Java is to become familiar with the pre-canned class libraries. These libraries consist of a large number of predefined classes that have many class level variables and methods. It is outside of the scope of this paper to even begin to list them. But, in order to get used to the idea of using them and see the syntax involved, let's examine a simple example. Using SAS functions and comparable Java methods, let's find a random observation number in the range of 1 to 100. The SAS code and its result (from the log) would look like this:

```
data _null_ ;
   rn1 = ranuni(0) ;
   rn2 = rn1*100 ;
   obs = ceil(rn2) ;
   put "   0-1 : " rn1 ;
   put " 0-100 : " rn2 ;
   put " Obs. Number : " obs ;
run ;

  0-1 : 0.7932329135
0-100 : 79.32329135
Obs. Number : 80
```

Note that we do not have to declare data types in the SAS code and we only need to know the function names and the required parameters involved. A Java scriptlet to accomplish the same task and its output (to the browser) would look like:

```
<%
   double rn1 = Math.random() ;
   double rn2 = rn1*100.0 ;
   int obs = (int)Math.ceil(rn2) ;
   out.println("  0-1 : "+ rn1 + "<br />") ;
   out.println("0-100 : "+ rn2 + "<br />") ;
   out.println("Obs. Number : "+ obs) ;
%>

   0-1 : 0.666922275367492
   0-100 : 66.6922275367492
   Obs. Number : 67
```

In this scriptlet, all of the variables must have their types explicitly declared, the static methods `Math.random()` and `Math.ceil()` must be invoked using the class name, and the type double results from `Math.ceil()` must be explicitly converted (cast) to an integer. The cast is done by enclosing the type name in parentheses and appending it to the front of the expression to convert. Also, note the `println()` method used for writing the results. It has been called using the object `out`. This object, along with a number of others, is automatically defined for the programmer within the JSP.

JAVA STRINGS

Java has a primitive data type called `char` for working with single characters. Variables of this type hold a single 16-bit character and can be loaded with a character constant enclosed in single quotes. But, Java also has a String data type for working with any amount of text - whose values are enclosed in double quotes. Unfortunately, both the terminology and the use of specific types of quotes can sometimes “catch” a SAS programmer.

There are also two other major syntax issues that involve String variables. First, Java overloads one of its operators to provide for String concatenation - the plus sign. Second, a String can be created, used, and act like an object or it can be created as a literal value and in some cases act like a primitive data value. Both of these issues will be investigated next.

STRINGS AND PLUS SIGNS

An overloaded operator is an operator that has more than one use. For example, in SAS, an equals sign can be either an assignment or a logical test. The only situation where operator overloading is supported in the Java language is a plus sign. With numbers it means to add and with Strings it means to concatenate. When Strings and numbers are used together it will mean concatenation unless the numbers and the plus signs are separated with parentheses. To illustrate, consider the following

scriptlet:

```
<%
   int x = 1 ;
   int y = 2 ;
   out.println("NOTE_"+ x + y) ;    // NOTE_12
   out.println("NOTE_"+ (x + y)) ; // NOTE_3
   out.println( x + y ) ;          // 3
%>
```

In the first `println()`, the text in the argument dictates that all of the plus signs here mean text concatenation. The integer information would be converted to simple text and the output would be: `NOTE_12`. In the second `println()`, the inner parentheses would force Java to see the integers as numbers and the plus would result in addition, then the resulting sum and the remaining text would be concatenated. The output would be: `NOTE_3`. In the last `println()`, there is no text and simple integer addition would results in a value of: `3`.

STRINGS LITERALS AND STRING OBJECTS

Strings in Java can be initially created two different ways. They can be defined as String literals within a simple assignment statement or they can be created using the `new` operator as a String object.

Like primitive data types, the identifiers of String literals are directly associated with their text values. The identifiers of the String objects are associated with the memory locations where they are built. Figure 2 graphically shows the difference:

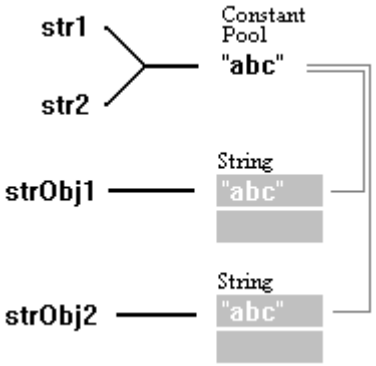

Figure 2. String Literals and String Objects

Unfortunately, this difference can lead to problems. Note the tests shown below and the resulting values:

```
<%
   String str1 = "abc" ;
   String str2 = "abc" ;
   String strObj1 = new String("abc") ;
   String strObj2 = new String("abc") ;
   boolean b = (str1==str2) ;       // true
   b = (str1==strObj1) ;            // false
   b = (strObj1==strObj2) ;         // false
   b = (str1.equals(str2)) ;        // true
   b = (str1.equals(strObj1)) ;     // true
   b = (strObj1.equals(strObj2)) ;  // true
%>
```

In the example, testing the two string literals with the `==` operator does give the expected result of true, but a false is generated with any literal vs. object or object vs. object test. Remembering that the operator `==` is designed to work on primitive data types; it should not be used with object identifiers. In the example, this is remedied by using the `equals()` method to test for equality of value. It returns the expected result value of `true` regardless of how the String was created.

CONDITIONAL PROCESSING

The Java conditional processing and looping control structures are similar to C and C++, but they are slightly different from SAS.

IF PROCESSING

A SAS programmer needs to note two major differences when coding Java `if` statements. First, Java does not use the `then` keyword. Second, the logical condition, which must evaluate to a boolean value, is required to be enclosed in parentheses. An example:

```
<%
   int x = 1 ;
   int y = 0 ;
   if (x == 1) {
        y = 1 ;
   }
   else {
        y = 2 ;
   }
%>
```

We can also see that, like SAS, the `else` is a separate and optional statement. And, like SAS, code blocks are allowed - but denoted by curly braces in Java.

TERNARY OPERATOR

Java also has a ternary operator, which allows conditional processing within an expression. The syntax looks like the following: `(test)?value_1 :value_2 .` The return value of this expression is `value_1` if the test is true and is `value_2` if the test is false. Using this operator, it would be very easy to recode the conditional processing done in the last scriptlet example into a single statement: `y = (x == 1)? 1: 2 ;`

SWITCH

In the case where multiple conditions need to be checked, SAS has a `select` control structure and Java has a `switch`. Programmers using C or C++ will recognize the `switch`. The statements used inside the two structures are noted below:

	SAS	Java
begin structure	`select(x) ;`	`switch(x) {`
process condition	`when(value)`	`case value:`
exit section		`break ;`
process default	`otherwise;`	`default:`
end structure	`end ;`	`}`

Unlike the SAS `select`, which can be used with any type of data and allows any type of relational or logical test, the `switch` is rather restricted. The `switch` will not compile for non-integers and cannot be used to test for any relationship other than equality. A simple example is shown below in order to see all of the statements within this control structure:

```
<%
   int x = 1 ;
   int y = 0 ;
   switch (x) {
        case 1:
             y += 10 ;
             break ;
        case 2:
        case 3:
             y += 20 ;
             break ;
        default:
             y += 30 ;
             break ;
```

```
    }
%>
```

This `switch` is coded so that ten is added to `y` if `x` is one. Or, if `x` is either two or three then twenty is added to `y`. And, if `x` is not one or two or three then thirty is added to the value of `y`. The result in this example will be that `y` ends up with a value of ten.

Note that the `break` in a `switch` is optional. It is typically used to exit any of the labeled sections, but the `break` can be removed to allow all subsequent statements to process. If the example shown above had all of its breaks removed then the final value of `y` would be sixty. It would start as zero, have ten added to it, have twenty added to it, and then finally have thirty added on.

Also note that the `default` is not required in Java. This is different from SAS where an `otherwise` statement is required if the `when` statements do not cover all of the possible values that might be encountered.

JAVA LOOPS

For simple iterative processing; SAS uses a `do` and Java a `for` loop. The basic syntax of the SAS `do` loop allows the use of any range or list of values. The Java `for` loop allows any range of numeric or character values. Java does not process lists within the syntax of a `for` loop, but uses special classes instead. Both languages also support conditional looping. The differences between SAS and Java loops are shown below:

loop	**SAS**	`do i=1 to n ; ... end ;`
	Java	`for (int i=1; i<=n; i++) { }`
loop - list	**SAS**	`do i=1,2, n ; ... end ;`
	Java	`Iterator Objects`
while loop	**SAS**	`do while(x) ; ... end ;`
	Java	`while(x) { }`
until loop	**SAS**	`do until(x) ; ... end ;`
	Java	`do { } while(x) ;`

FOR LOOP

The basic syntax of the Java loop involves three terms. An initialization term (that typically defines a new variable to be used to control the loop processing), a logical condition to determine when the loop will process, and a term that will process upon each iteration of the loop. These terms are separated by semicolons within a set of parentheses following the `for`. The main body of the loop follows this enclosed inside of a set of curly braces. Consider the following example:

```
<%
   double x = 0.0 ;
   for (int i = 1 ; i < 3 ; i++) {
        x += Math.random() ;
   }
%>
```

As Java begins to processes this loop, it will initially setup a new integer variable `i` set to one, as defined in the first term. It will process through the loop as long as the expression `i < 3` is true, as defined in the second term. And, each time Java iterates through the loop, one will be added to `i`, as defined in the third term.

It should be noted, that is a very basic example. In fact, the first and third terms in a `for` specification can involve multiple variables separated by commas. Although, this does not extend to the second term; it still only takes a single logical expression.

WHILE AND DO WHILE LOOPS

In SAS, besides `do` loops that process through a range of values or a list of values, there are also `do while` and `do until` loops. In Java there is a `while` loop and a `do while` loop:

```
<%
   int i = 0 ;
   while (i < 3) {      // checked at top
        i++ ;           // like SAS do while
   }
   i = 0 ;
   do {                 // checked at bottom
        i++ ;           // like SAS do until
   } while (i < 3) ;
%>
```

The only major difference, other than the keyword difference, is that a true condition in the Java `do while` continues the looping, unlike a SAS `do until` which stops processing if the condition is true.

SCOPE

In Java the location of a variable definition is highly significant in relation to where the variable can be used - it's scope. Variables defined within any type of code block (like the code blocks seen earlier with conditional processing and loops) are restricted to be local within that code block. Consider the following example of code within a labeled code block:

```
<%
   int x = 0 ;
   cbk: {
        int y = x ;
   }
   x = 1 ;
%>
```

The variable `x` is available anywhere either inside or outside of the code block labeled `cbk`. But, the variable `y` was created inside of the block and is not available outside of it. Note that a code block does not have to be labeled - any set of matching curly braces would define a code block.

MISSING

In SAS the concept of a missing value is rather straightforward. Reading invalid data, using illegal arguments in functions, and requesting impossible mathematical operations will all generate a SAS missing value. In the Java language, data problems can result in compile errors, NaN ("Not a Number") values, positive infinity, negative infinity, and exceptions. Consider the following calculations in Java that in SAS would all generate missing values:

Calculation	**Resulting Value**
`0.0/0.0`	`NaN`
`1.0/0.0`	`Infinity`
`-1.0/0.0`	`-Infinity`

NaN calculations, like missing values in SAS, propagate NaN values. But, NaN is not ordinal. NaN cannot be meaningfully tested with logical operators - all tests will return a false value. Even testing for equality will return a false unless the Double.isNaN() method is used.

Infinity, unlike missing, can be used in meaningful calculations. But, unlike NaN and like missing values in SAS, Infinity is ordinal. It can be tested with logical operators or the `==` operator.

Another possible value that will show up in Java code is a null value. In SAS the keyword `null` can be used in where processing with the `is` keyword as an alternative to the keyword `missing`. However, in Java, the standard default value of an object reference is `null`. This indicates that no instance of a class is assigned to an object identifier.

TRY CATCH BLOCKS

As mentioned in the section on missing values, data problems can result in exceptions. Unlike NaN or Infinity, this is not a value that is assigned to an identifier. Instead, it is a condition that is generated to identify a severe problem and it will terminate the processing of the code if it is not "handled". Consider the example where a scriptlet had an integer calculation with division by zero. Unlike a double calculation with division by zero, this is not allowed and it will generate or "throw" an ArithmeticException. This exception will terminate the processing of the code - unless the calculation is performed inside of a `try` block where the exception can be detected and subsequently handled in a `catch` block. The syntax for this type of structure is displayed in the following example:

```
<%
   try {
        out.print(" 10 / 0 ") ;
        int x = 10 / 0 ;
        out.println(" = "+ x ) ;
   }
   catch (java.lang.ArithmeticException e) {
        out.println(" _error_: div by zero") ;
   }
%>
```

Note that these are not a labeled sections, they are supposed to start with the keywords `try` and `catch`. Within the initial block, the `out.println()` method would write the "`10 / 0`" text, then the code would create a new integer called `x` and attempt to load it with the results of the calculation. The calculation will throw an ArithmeticException error and immediately transfer control to the `catch` block. In this block an additional piece of text with an error message will be written to give the final result of:

```
10 / 0 _error_: div by zero .
```

The line in the try block with the instructions to write out the equals sign and the value of `x` is completely bypassed.

Additionally, an optional `finally` block can be coded to execute a code segment regardless of the catch. Note that a `try` must have a `catch` or a `finally` associated with it, but both are not required. Also, multiple `catch` blocks can be associated with a single `try`.

CONCLUSION

This paper attempts to provide a quick introduction to the basic syntax of the Java programming language. It points out a number of similarities and differences between SAS and Java syntax. Hopefully this will benefit SAS programmers who are beginning to investigate Java. But, aside from the basic syntax elements, Java is a complex object-oriented programming language. To effectively work with Java, it would be advisable to consider a good reference, one or more instructor-based classes, and several months of solid experience.

REFERENCES

[1] SAS Institute Inc. (1999), SAS Language Reference: Concepts, Version 8, Cary, NC: SAS Institute Inc.

[2] SAS Institute Inc. (1999), SAS Language Reference: Dictionary, Version 8, Cary, NC: SAS Institute Inc.

[3] Roberts, S., Heller, P., Ernest, M. (2000), Complete Java 2 Certification Study Guide, Second Edition, SYBEX Inc.

[4] Whitehead, Paul (2001), JavaServer Pages: Your visual blueprint for designing dynamic content with JSP, Hungry Minds, Inc.

CONTACT INFORMATION

Please forward comments and questions to:

Don Boudreaux, Ph.D.
SAS Institute Inc.
11920 Wilson Parke Ave
Austin, TX. 78726

Work: 512-258-5171 ext. 3335
E-mail: don.boudreaux@sas.com