"RexxScript" – Rexx Scripts Hosted and Evaluated by Java (Package javax.script)

Flatscher, Rony G.

Published in:
The Proceedings of the Rexx Symposium for Developers and Users

Published: 01/01/2017

Document Version
Publisher's PDF, also known as Version of record

"*RexxScript*" – *Rexx Scripts Hosted and Evaluated by Java (Package* javax.script*)*

Rony G. Flatscher (Rony.Flatscher@wu.ac.at), WU Vienna
"The 2017 International Rexx Symposium", Amsterdam, The Netherlands
April 9th – 12th, 2017

Abstract. The latest version of BSF4ooRexx (a Rexx-Java bridge) implements a Rexx script engine ("RexxScript") according to the specifications laid out in the Java package *javax.script*. This article explains the core concepts of *javax.script* for hosting and evaluating script programs from Java and introduces the new "*RexxScript*" implementation with features that are supposed to ease devising and debugging "Rexx scripts" for Rexx and Java programmers alike. Working stand-alone nutshell examples will demonstrate the new features and will also showcase the available possibilities to interact with the Java supplied *ScriptContext* from the evaluated Rexx scripts hosted by Java programs.

1 Introduction

The Java specification request group 223 ("JSR-223") [1] was formed in 2003 to create a Java package for scripting by eventually defining the Java package *javax.script* in the course of three years.[1] This package was introduced with Java 6 in December 2006 and standardizes how Java interacts with scripting languages of any kind.[2]

The BSF4ooRexx package [4] implements a full functional, bidirectional bridge between ooRexx [5] and Java [6] that allows on the one hand ooRexx to interact with Java objects and on the other hand allows Java to interact with ooRexx objects and run ooRexx programs. BSF4ooRexx is based on the Apache Software Foundation's "Bean Scripting Framework (BSF)" [7] that predates the JSR-223 specifications by almost a decade.

With the advent of ooRexx 4.0[3] in 2009 [8] the scripting language got a new kernel with a comprehensive set of native APIs modeled after Java's JNI [10]. Over the course of the next years BSF4Rexx was rewritten to take advantage of the new kernel and has been renamed to "BSF4*oo*Rexx" to indicate that the new features

1 The author served as an expert in the JSR-223 group. The downloadable JSR-223 specifications can be found at [2].

2 The Java 1.8/8 documentation for the package *javax.script* can be found in [3].

3 At the time of this writing beta versions of ooRexx 5.0 became available for download at [9].

are only available with ooRexx 4 and higher ([11][4], [12][5]). As one of the results this ooRexx to Java bridge has become able to allow for implementing abstract Java classes, Java interface classes and (abstract) Java methods in ooRexx, such that Java method invocations will transparently cause appropriate ooRexx messages to be sent to the proxy ooRexx objects.

In the fall of 2016 work on BSF4ooRexx begun with the goal to make ooRexx available to Java via the *javax.script* package. This would allow Java programmers accustomed to JSR-223 to employ ooRexx for their scripting purpose, without a need to learn the Apache BSF package as is a prerequisite for using BSF4ooRexx from the Java side. In addition any existing Java application that allows the users for identifying a scripting language merely by its name would gain the support for Rexx and ooRexx scripts by merely installing the BSF4ooRexx package![6]

This article will first give an overview of the most important concepts and classes of the *javax.script* package, which then is followed by the introduction of the BSF4ooRexx implementation called "RexxScript" together with the newly introduced "RexxScript annotations".

All the examples in this article will demonstrate and explain how to put the *javax.script* and RexxScript infrastructure to work for the benefits of Java and/or ooRexx programmers.

---

4 [11] discusses some of the shortcomings of BSF4Rexx that were due to the industry standard Rexx SAA (IBM's System Application Architecture) APIs from the 80's. With the new APIs in the ooRexx 4.0 kernel it became possible to implement Rexx proxy objects for Java, real-time handling of Java events, enabling the implementation of abstract Java methods with Rexx methods, communicating Rexx conditions to Java and last, but not least, to allow Rexx to throw specific Java exceptions. As these new features depend on the new ooRexx 4.0, BSF4Rexx from then on was renamed to BSF4ooRexx.

5 [12] documents the new possibilities that BSF4ooRexx introduced by 2012, namely allowing the configuration of Rexx interpreter instances for the first time, including the ability to configure and implement Rexx exit handlers and Rexx command handlers in Java. The appendix takes advantage of BSF4ooRexx "omnipotency" for ooRexx camouflaging Java as ooRexx: it demonstrates how this infrastructure allows the implementation of Rexx exits and Rexx command handlers even in pure Rexx itself!

6 One such example is JavaFX which allows for using any script code in *FXML* files by merely stating with an XML process instruction the name of the script engine to use when code in external files or in event handlers has to be executed from that *FXML* file.

2 The *javax.script* Package

This section introduces briefly the purpose and how the Java classes defined in the *javax.script*[7] package interact in order to become able to understand the Java script framework if one wishes to exploit it. A service provider for a script engine must implement the Java interface classes *javax.script.ScriptEngineFactory*[8] and *javax.script.ScriptEngine*[9] *for evaluating (executing) script code*.

A *ScriptEngine* maintains a *ScriptContext* that manages the environment in which the script gets evaluated and which uses a numerically indexed collection of *Bindings* which each represent a collection of name-value pairs ("attributes") that a script should be able to access. *SimpleScriptContext*[10] implements the Java interface *ScriptContext* which in turn uses the Java class *SimpleBindings* which implements the Java interface *Bindings*.

The *ScriptEngineManager* maintains all available script engines (using the Java service provider mechanism[11]) and allows for maintaining a *Bindings* that is to be used by all script engines it created[12].

Code 1 demonstrates how a Java program uses the *ScriptEngineManager* to load the JavaScript engine and then uses it to evaluate (execute, run) a simple JavaScript program which will output the string: `Hello world from JavaScript!`.

The Java host program is free to add any information to any available *Bindings* of the *ScriptContext* to use for evaluating a script. For each invocation (evaluation) of a script the Java host should supply at least the following entries in the *ScriptContext's ENGINE_SCOPE* (a constant number with the value *100*) *Bindings*, if possible:

7 This article will omit the package name *javax.script* to ease reading.

8 This class holds information about the script engine and with the method *getScriptEngine* returns a *ScriptEngine* implementation that will allow for evaluating (executing) script code.

9 Implementing a *ScriptEngine* is eased considerably, if one merely extends the class *AbstractScriptEngine*.

10 This implementation uses two *Bindings*, one with the numeric index *100* (*ENGINE_SCOPE*), which maintains attributes (name-value pairs) for the current script evaluation, and one with the numeric index *200* (*GLOBAL_SCOPE*), which maintains attributes that are meant to be shared among all scripts that get executed *by a Java host program*. It would be possible to supply an own implementation of the Java *ScriptContext* interface, which might allow for more than the two default Bindings *ENGINE_SCOPE* and *GLOBAL_SCOPE*.

11 A script engine implementation needs to supply the fully qualified name of its *ScriptEngineFactory* in the file *META-INF/services/javax.script.ScriptEngineFactory* of its package.

12 This *Bindings* is indexed with the numeric value *200* (*GLOBAL_SCOPE*) in the *ScriptContext*.

```
import javax.script.*;
public class Test_00_js
{
    public static void main (String args[])
    {
        ScriptEngineManager sem=new ScriptEngineManager();
        ScriptEngine        se =sem.getEngineByName("JavaScript");
        try
        {
            se.eval("print (\"Hello world from JavaScript!\");");
        }
        catch (ScriptException sExc)
        {
              System.err.println(sExc);
        }
    }
}
```

Code 1: A Java program using the JavaScript engine.

- in the case of supplying argument(s) to the script, a Java *Array* object of type *Object* can be created and should be stored under the name "*javax.script.argv*"[13],
- if the script was read from a file name, then the Java host should supply that name (a *String*) with the name "*javax.script.filename*"[14].

The Java scripting framework defines two optional interface classes, *Compilable* and *Invocable*. The optional *Compilable* interface defines two *compile* methods to allow compilation of scripts into *CompiledScript* objects that can be (re-)used to evaluate (run, execute) compiled scripts and to get access to their script engine. The optional *Invocable* interface defines a method *getInterface* which expects the resulting object to implement the methods of the supplied Java class object, a method *invokeFunction* that allows to run top-level routines (procedures, functions) in the *ScriptEngine* and *invokeMethod* that allows to execute methods in a script object.

13 This is a standardized name for which the class *ScriptEngine* constant named *ARGV* defines the *String* value "*javax.script.argv*".

14 This is a standardized name for which the class *ScriptEngine* constant named *FILENAME* defines the *String* value "*javax.script.filename*".

```
import javax.script.*;
public class Test_00_rex
{
    public static void main (String args[])
    {
        ScriptEngineManager sem=new ScriptEngineManager();
        ScriptEngine        se =sem.getEngineByName("Rexx");
        try
        {
            se.eval("say \"Hello world from Rexx!\"");
        }
        catch (ScriptException sExc)
        {
              System.err.println(sExc);
        }
    }
}
```

Code 2: A Java program using the RexxScript engine.

3 The *RexxScript* Implementation

BSF4ooRexx defines a Rexx script package *org.rexxla.bsf.engines.rexx.jsr223*[15] – also known as "*RexxScript*" package" – which contains the implementations of a Rexx script factory named *RexxScriptFactory*, a Rexx script engine named *RexxScriptEngine* and a Rexx specific implementation of *CompiledScript*, the *RexxCompiledScript* class. The *RexxScriptFactory* class is made known via the Java service provider interface conventions[11], such that after BSF4ooRexx got installed the *RexxScriptEngine* can be used transparently by Java programmers. Code 2 demonstrates how a Java host program uses the *ScriptEngineManager* to load the Rexx engine and then uses it to evaluate (execute, run) a simple Rexx program which will output the string: `REXXout>Hello world from Rexx!`[16].

In addition to what the *ScriptEngine* interface defines, the *RexxScriptEngine* implements among other things the following functionality:

- the optional interface *Compilable*, which allows for tokenizing Rexx programs/scripts and reuse them as *CompiledScript*s, the public method *getCurrentScript* returns the latest compiled (tokenized) Rexx script[17],

15 The BSF4ooRexx Javadocs document all BSF4ooRexx Java classes.

16 The prefix "*REXXout>*" is supplied by the *RexxScriptEngine* whenever a Rexx program uses the *SAY* keyword statement in order to distinguish Rexx output from any other output from a Java program. This allows one to distinguish the output from Java and Rexx programs.

17 The *RexxScriptEngine* caches the last evaluated (executed) Rexx program or script.

- the optional interface *Invocable*, which allows for using Rexx objects for carrying out the abstract Java methods defined in Java interface classes[18], running public Rexx routines[19] and sending Rexx objects messages[20] from Java.
- Redirection of the Rexx *.input*, *.output*, *.error* , *.debuginput* and *.traceoutput* monitors to the Java input *Reader*, output *Writer* and error *Writer* objects as supplied via the current *ScriptContext*. This allows Rexx output to be loggable along with the Java output and gets controlled by the static boolean field *bRedirectStandardFiles*.
- In order to ease spotting Rexx input or output the RexxScriptEngine will prefix any Rexx input or output with the strings "*REXXin?>*" (*.input*), "*REXXout>*" (*.output*), "*REXXerr>*" (*.error*), "*REXXdbgIn?>*" (*.debuginput*) and "*REXXtrc>*" (*.traceouptut*) to ease spotting Rexx output and the kind of interaction with the Java *Reader* and *Writer* objects for debugging and analyzing purposes.

The *RexxCompiledScript* class extends *CompiledScript* and implements the *Invocable* interface and in addition adds the following public methods:

- *getFileName*: returns the filename,
- *getScriptSource*: returns the source code as a *String*,
- *getEditedScriptSource*: returns the source code as a *String* that was tokenized and gets actually executed. This String is different to what *getScriptSource* returns, if RexxScript annotations (see below) are

18 Invocable restricts the *getInterface* method to work for a single Java interface class. BSF4ooRexx by default allows any number of Java interface classes to be implemented in a single ooRexx class. Cf. BSF4ooRexx' external Rexx function *BSFCreateRexxProxy*.

19 The RexxScript engine by default behaves differently compared to ooRexx: the RexxEngine will collect the Rexx package of an evaluated Rexx scripts and add that package to the next Rexx script to evaluate (execute). This way a *RexxEngine* – and the scripts it runs – will gain access to all public routines and public classes that have been created in its lifetime when evaluating Rexx scripts and programs. This behavior matches the behavior of quite many script engines, especially those that are deployed in the context of HTML. (ooRexx never adds packages automatically to all Rexx programs, a Rexx programmer must do that by explicitly using the *::REQUIRES* directive.)

20 BSF4ooRexx allows Java programmers to wrap Rexx objects as Java *RexxProxy* objects and send them any Rexx messages they have a need for from Java. If a Rexx script returns Rexx objects as a result to Java, then BSF4ooRexx will wrap it up as a Java *RexxProxy* object (package *org.rexxla.bsf.engines.rexx*). [11]

embedded in the original Rexx script.

3.1 Evaluating Rexx Scripts with *javax.script*

If a Rexx script gets evaluated (executed, run), then BSF4ooRexx will carry out all necessary steps to do so. As the invocation of Rexx scripts occurs from Java, BSF4ooRexx will always supply a trailing slot argument of the Rexx type (class) *Slot.Argument*[21], which subclasses the ooRexx *Directory* class.

In the case of running Rexx scripts via the *RexxScriptEngine*, the slot argument will have an entry named *SCRIPTCONTEXT* which allows one to fetch the current *ScriptContext* Java object from the invoked Rexx executable. This way it becomes possible to interact with the Java *ScriptContext* directly from Rexx, allowing the Rexx programmer to directly fetch attributes from the *ScriptContext Bindings*, change, add or delete them.[22]

In the case that there are many entries in the *ScriptContext* that should be made available as context Rexx variables in the scripts, the *RexxScriptEngine* implementation introduces the possibility of defining "RexxScript annotations". A RexxScript annotation is enclosed in a Rexx block comment which starts and ends in the same line and contains one of the strings "*@get(...)*", "*@se*t(...)" or "*@showsource*". The *@get* and *@set* annotations expect blank delimited[23] names within their parentheses that allow to address attributes in the *ScriptContext*'s *Bindings* that are reflected in the Rexx script. The *@get* annotation will fetch the listed attributes from the current *ScriptContext* and create Rexx context variables of the same name, the *@set* annotation will cause the values of Rexx context variables to be written back into their corresponding attributes in the current

21 A Rexx programmer therefore can always test the last argument whether it got added by BSF4ooRexx as a slot argument by sending it the message *isA(.slot.Argument)*.

22 Quite a few Java *ScriptEngine* implementations automatically push the *ScriptContext* attributes into the scripts by creating local context variables by the same name and push value changes back to their corresponding attributes into the *ScriptContext Bindings*. This may incur very subtle problems, e.g. in cases where scripts use local variable names that over the life time of such scripts may get all of a sudden overwritten by the Java host If additional attributes get added that happen to have the same names as context variables in the evaluated scripts, these may get overwritten without notice, The *RexxScriptEngine* does not automatically create context variables in Rexx, scripts but rather has the Rexx programmer decide which entries to use as context variables in their scripts, which never will allow unexpected name clashes.

23 Should the name of a *ScriptContext Bindings* attribute contain blanks, then one needs to enquote the name in double or single quotes.

ScriptContext Bindings. The *@showsource*[24] annotation will cause the *RexxScriptEngine* to display the edited Rexx script source code to be shown on *.output*, such that a Rexx programmer becomes able to see how the *@get* and *@set* annotations changed the Rexx script source.[25]

Before the Rexx script gets executed the filename of the Rexx script will be set from the entry *javax.script.filename*[26] (*ScriptEngine.FILENAME*) in the *ENGINE_SCOPE Bindings* of the *ScriptContext* and if there is an entry named *javax.script.argv* (*ScriptEngine.ARGV*) in the *ENGINE_SCOPE Bindings* of the *ScriptContext* then this Java array object will be used to directly supply the Rexx scripts the arguments. Therefore Rexx programs invoked this way can directly fetch Java host arguments with the *USE ARG* keyword statement.

3.2 Invoking a Rexx Script from a Java Host

The first Java example evaluates (executes, runs) the Rexx script shown in code 3 below and which gets stored under the name "test_rexx_01.rex" in the same directory as the Java host program. It will show the result of the *PARSE SOURCE* keyword statement which will include the defined filename of the Rexx script. In addition it will iterate over all received arguments and show them. If an argument (always the last one) is of the Rexx type *Slot.Argument*[27], then it will iterate over

```
parse source s
say "parse source: ["s"]"
say

say "received" arg() "arguments:"
do i=1 to arg()
   val=arg(i)      -- get value
   say "   arg #" i": ["val"]"
   if val~isA(.slot.Argument) then -- .slot.Argument is a subclass of .Directory
   do
      say "   a directory with the following entries:"
      loop idx over val~allIndexes~sort
         say "       idx=["idx"] -> item=["val[idx]"]"
      end
   end
end
```

Code 3: "test_rexx_01.rex": Dump received arguments.

24 The complete *RexxScript* annotation to embed into a Rexx script would be: */*@showsource*/*

25 Rexx script annotations depend on the presence of a slot argument.

26 The Rexx *PARSE SOURCE* keyword statement allows for fetching this filename value.

27 The Rexx class *Slot.Argument* gets defined by BSF4ooRexx and subclasses the Rexx class *Directory*. This way a Rexx programmer can always determine whether the last argument is a slot argument (appended by BSF4ooRexx) or not by sending it the message *isA(.slot.Argument)*.

```
import javax.script.*;
import java.io.FileReader;
import org.rexxla.bsf.engines.rexx.jsr223.*;

public class Test_01   // demo evaluating a Rexx script
{
    public static void main (String args[])
    {
        ScriptEngineManager manager = new ScriptEngineManager();
        RexxScriptEngine rse=(RexxScriptEngine) manager.getEngineByName("Rexx");
        try
        {
           String filename="test_rexx_01.rex"; // define the filename
                // add the filename to the engine's SimpleBindings
            ScriptContext sc=rse.getContext();  // get the default ScriptContext
            sc.setAttribute(ScriptEngine.FILENAME,filename,ScriptContext.ENGINE_SCOPE);
            rse.eval(new FileReader(filename)); // now let us execute the Rexx script

            System.out.println("\n... about to reuse the last used Rexx script ...\n");

                // add arguments for the script to the ENGINE_SCOPE bindings
            sc.setAttribute(ScriptEngine.ARGV,
                new Object[] {"one", null, java.util.Calendar.getInstance()},
                ScriptContext.ENGINE_SCOPE);
                // the RexxScriptEngine always compiles the last script and
                // makes it available with the getCurrentScript() method
            rse.getCurrentScript().eval();  // now let us re-execute the Rexx script
        }
        catch (Exception exc)
        {
            System.err.println(exc);
            System.exit(-1);
        }
    }
}
```

Code 4: "Test_01.java": Java host program to run "test_rexx_01.rex" twice.

its content.

The Java program "*Test_01.java*" in code 4 above sets the filename of the Rexx program (script) to "*test_rexx_01.rex*" using the *ScriptContext*'s *ENGINE_SCOPE Bindings*. It then evaluates (executes, runs) the Rexx program without arguments. The first six lines in the output depicted in output 1 below stem from this first evaluation (execution) of the Rexx script.

Then the Java program outputs a blank line, an informative message about reusing the current Rexx script and another blank line. It then defines an entry in the *ScriptContext*'s engine *Bindings* with the name "*javax.script.argv*" for the Java *Array* of type *Object* that gets supplied to the script. The content of this *Array* object will be used by the *RexxScriptEngine* to supply the arguments directly to the invoked Rexx script. The Java program then requests the current Rexx script

```
REXXout>parse source: [WindowsNT SUBROUTINE test_rexx_01.rex]
REXXout>
REXXout>received 1 arguments:
REXXout>   arg # 1: [a Slot.Argument]
REXXout>   a directory with the following entries:
REXXout>       idx=[SCRIPTCONTEXT] -> item=[javax.script.SimpleScriptContext@308db1]

... about to reuse the last used Rexx script ...

REXXout>parse source: [WindowsNT SUBROUTINE test_rexx_01.rex]
REXXout>
REXXout>received 4 arguments:
REXXout>   arg # 1: [one]
REXXout>   arg # 2: [The NIL object]
REXXout>   arg # 3: [java.util.GregorianCalendar@12c9b19]
REXXout>   arg # 4: [a Slot.Argument]
REXXout>   a directory with the following entries:
REXXout>       idx=[SCRIPTCONTEXT] -> item=[javax.script.SimpleScriptContext@308db1]
```

Output 1: Output of executing "java Java_01".

(the last one the *RexxScriptEngine* evaluated) and re-evaluates it.

The last nine lines in the output depicted in output 1 above stem from this second evaluation of the Rexx script.

It is maybe interesting to note that in output 1 above the Rexx output can be easily distinguished from the Java output, as it gets prefixed with the string "*REXXout>*".

3.3 Interacting with a *ScriptContext* from Rexx

The purpose of this example is to demonstrate how a Rexx program, "*test_rexx_02.rex*", can interact with the *ScriptContext* fetched from the slot argument that BSF4ooRexx supplies.

The Java program "*Test_02.java*" is given in code 6 below and comparing it to "*Test_01.java*" in code 4 above the only change lies in the name of the Rexx file to "*test_rexx_02.rex*" evaluate (execute). The logic of the Java host program is unchanged, such that it evaluates the script twice, once without and once with arguments. Hence the explanations of the Java program in section 3.2 above apply accordingly.

"*test_rexx_02.rex*" is depicted in code 5 below and does the following:

- the *PARSE SOURCE* keyword instruction retrieves among other things the filename as set in the Java host program,

```
parse source s
say "parse source: ["s"]"
say
-- demonstrate how to access and use the ScriptContext
slotDir=arg(arg())        -- last argument is a directory containing "SCRIPTCONTEXT"
sc=slotDir~scriptContext -- fetch the ScriptContext object
say "ScriptContext field: ENGINE_SCOPE:" pp(sc~engine_scope)
say "ScriptContext field: GLOBAL_SCOPE:" pp(sc~global_scope)
say

-- import the Java class that defines some Constants like FILENAME, ARGV ...
seClz=bsf.importClass("javax.script.ScriptEngine")
say "ScriptEngine field: FILENAME="pp(seClz~filename)
say "ScriptEngine field: ARGV    ="pp(seClz~argv)
say

key=seClz~FILENAME -- get string value for FILENAME entry in Bindings
say "value of ScriptEngine static field named ""FILENAME"":" pp(key)
say "  fetch filename from ScriptContext                :" pp(sc~getAttribute(key))
say "  fetch scope (engine or global) from Scriptcontext:" pp(sc~getAttributesScope(key))
say

key=seClz~ARGV    -- get string value for ARGV entry in Bindings
say "value of ScriptEngine static field named ""ARGV""    :" pp(key)
say "  fetch filename from ScriptContext                :" pp(sc~getAttribute(key))
say "  fetch scope (engine or global) from Scriptcontext:" pp(sc~getAttributesScope(key))

say "---"
say "received" arg() "arguments:"
do i=1 to arg()
   val=arg(i)
   str="    arg("i")=["
   if val~isA(.bsf) then str=str || val~toString"]"
                    else str=str || val"]"
   say str
   if val~isA(.slot.Argument) then
   do
      say "    a directory with the following entries:"
      loop idx over val~allIndexes~sort
         say "        idx=["idx"] -> item=["val[idx]"]"
      end
   end
end
```

Code 5: "test_rexx_02.rex": Interact with ScriptContext and show arguments.

- it fetches the last argument using the Rexx built-in function *ARG*[28] and retrieves its entry named *SCRIPTCONTEXT*, which then is used to query the values of the *ScriptContext* constant fields named *ENGINE_SCOPE* (the numeric value *100*) and *GLOBAL_SCOPE* (the numeric value *200*),
- it imports the Java class *ScriptEngine* and uses it to query the values of the *ScriptEngine* constant fields *FILENAME* ("*javax.script.filename*") and *ARGV* ("*javax.script.argv*"),
- it then demonstrates how to use the *ScriptContext*'s *getAttribute* and

28 The built-in function *ARG()* returns the number of arguments which is used in the outer *ARG* function to index and retrieve the last argument, which is the slot argument object supplied by BSF4ooRexx, hence the statement: *slotDir=arg(arg())*.

```
import javax.script.*;
import java.io.FileReader;
import org.rexxla.bsf.engines.rexx.jsr223.*;

public class Test_02   // demo evaluating a Rexx script
{
    public static void main (String args[])
    {
        ScriptEngineManager manager = new ScriptEngineManager();
        RexxScriptEngine rse=(RexxScriptEngine) manager.getEngineByName("Rexx");
        try
        {
           String filename="test_rexx_02.rex"; // define the filename
                // add the filename to the engine's SimpleBindings
            ScriptContext sc=rse.getContext();  // get the default ScriptContext
            sc.setAttribute(ScriptEngine.FILENAME,filename,ScriptContext.ENGINE_SCOPE);
            rse.eval(new FileReader(filename)); // now let us execute the Rexx script

            System.out.println("\n... about to reuse the last used Rexx script ...\n");

                // add arguments for the script to the ENGINE_SCOPE bindings
            sc.setAttribute(ScriptEngine.ARGV,
                new Object[] {"one", null, java.util.Calendar.getInstance()},
                ScriptContext.ENGINE_SCOPE);
                // the RexxScriptEngine always compiles the last script and
                // makes it available with the getCurrentScript() method
            rse.getCurrentScript().eval();  // now let us re-execute the Rexx script
        }
        catch (Exception exc)
        {
            System.err.println(exc);
            System.exit(-1);
        }
    }
}
```

Code 6: "Test_02.java": Java host program to run "test_rexx_02.rex" twice.

getAttributesScope using the values "*javax.script.filename*" *and* "*javax.script.argv*":

- *getAttributesScope*("*javax.script.filename*") returns the numeric value *100*, which indicates that it is stored in the *ENGINE_SCOPE Bindings* of the current *ScriptContext*.
- The next statement, *getAttributesScope*("*javax.script.argv*"), will return *-1* in the first run[29] (not present in any *Bindings* of the current *ScriptContext*), but *100* (*ENGINE_SCOPE Bindings* in the *ScriptContext*) in the second run of the script, as the Java host program will have placed this entry in the *ScriptContext ENGINE_SCOPE Bindings* before the second run!

29 Cf. Output 2 below.

```
REXXout>parse source: [WindowsNT SUBROUTINE test_rexx_02.rex]
REXXout>
REXXout>ScriptContext field: ENGINE_SCOPE: [100]
REXXout>ScriptContext field: GLOBAL_SCOPE: [200]
REXXout>
REXXout>ScriptEngine field: FILENAME=[javax.script.filename]
REXXout>ScriptEngine field: ARGV    =[javax.script.argv]
REXXout>
REXXout>value of ScriptEngine static field named "FILENAME": [javax.script.filename]
REXXout>  fetch filename from ScriptContext               : [test_rexx_02.rex]
REXXout>  fetch scope (engine or global) from Scriptcontext: [100]
REXXout>
REXXout>value of ScriptEngine static field named "ARGV"    : [javax.script.argv]
REXXout>  fetch filename from ScriptContext               : [The NIL object]
REXXout>  fetch scope (engine or global) from Scriptcontext: [-1]
REXXout>---
REXXout>received 1 arguments:
REXXout>    arg(1)=[a Slot.Argument]
REXXout>    a directory with the following entries:
REXXout>        idx=[SCRIPTCONTEXT] -> item=[javax.script.SimpleScriptContext@308db1]

... about to reuse the last used Rexx script ...

REXXout>parse source: [WindowsNT SUBROUTINE test_rexx_02.rex]
REXXout>
REXXout>ScriptContext field: ENGINE_SCOPE: [100]
REXXout>ScriptContext field: GLOBAL_SCOPE: [200]
REXXout>
REXXout>ScriptEngine field: FILENAME=[javax.script.filename]
REXXout>ScriptEngine field: ARGV    =[javax.script.argv]
REXXout>
REXXout>value of ScriptEngine static field named "FILENAME": [javax.script.filename]
REXXout>  fetch filename from ScriptContext               : [test_rexx_02.rex]
REXXout>  fetch scope (engine or global) from Scriptcontext: [100]
REXXout>
REXXout>value of ScriptEngine static field named "ARGV"    : [javax.script.argv]
REXXout>  fetch filename from ScriptContext               : [[Ljava.lang.Object;@16f27d]
REXXout>  fetch scope (engine or global) from Scriptcontext: [100]
REXXout>---
REXXout>received 4 arguments:
REXXout>    arg(1)=[one]
REXXout>    arg(2)=[The NIL object]
REXXout>    arg(3)=[java.util.GregorianCalendar[time=1507552521096,areFieldsSet=true,
areAllFieldsSet=true,lenient=true,zone=sun.util.calendar.ZoneInfo[id="Europe/Berlin",
offset=3600000,dstSavings=3600000,useDaylight=true,transitions=143,lastRule=
java.util.SimpleTimeZone[id=Europe/Berlin,offset=3600000,dstSavings=3600000,useDaylight=
true,startYear=0,startMode=2,startMonth=2,startDay=1,startDayOfWeek=1,startTime=3600000,
startTimeMode=2,endMode=2,endMonth=9,endDay=1,endDayOfWeek=1,endTime=3600000,endTimeMode=
2]],firstDayOfWeek=2,minimalDaysInFirstWeek=4,ERA=1,YEAR=2017,MONTH=9,WEEK_OF_YEAR=41,
WEEK_OF_MONTH=2,DAY_OF_MONTH=9,DAY_OF_YEAR=282,DAY_OF_WEEK=2,DAY_OF_WEEK_IN_MONTH=2,
AM_PM=1,HOUR=2, HOUR_OF_DAY=14,MINUTE=35,SECOND=21,MILLISECOND=96,ZONE_OFFSET=3600000,
DST_OFFSET=3600000]]
REXXout>    arg(4)=[a Slot.Argument]
REXXout>    a directory with the following entries:
REXXout>        idx=[SCRIPTCONTEXT] -> item=[javax.script.SimpleScriptContext@308db1]
```

Output 2: Output of running "java Java_02".

- The remaining code lists the received arguments and in the case of a *Directory* object will list all entries ordered by key. The last argument in RexxScript invoked Rexx routines and methods will be the slot argument

Analyzing the output of the two runs in output 2 above, the same Rexx script

"*test_rexx_02.rex*" hosted by the Java program "*Test_02.java*" yields quite different outputs, depending on the presence of arguments supplied by the Java host program using a Java Array attribute named *ScriptEngine.ARGV* and stored in the *ENGINE_SCOPE Bindings* of the *ScriptContext*.

3.4 Rexx Script Annotations

The previous section demonstrated how a Rexx program can interact with the current *ScriptContext* supplied by the Java host program. This knowledge allows a Rexx programmer to fetch any attribute from any *Bindings* of a *ScriptContext* and also change, delete or add attributes to any of the *ScriptContext Bindings*.

As one can expect that Rexx scripts usually will need to use some Java host supplied attributes in one of the *ScriptContext Bindings* and possibly update such attributes, Rexx script annotations[30] have been implemented in *RexxScriptEngine* to make it easy to incorporate such attributes as Rexx context variables and update the *Bindings* attributes directly from Rexx context variables.

The Rexx program "*test_rexx_03.rex*" in code 7 below is structured as follows:

- In the prolog[31] code the attributes named *d1*, *d2* and *sum* get fetched from the *ScriptContext* using its *getAttribute* method and assigned to Rexx variables of the same name, which then get merely displayed.
- In the public routine named *one*, a get RexxScript annotation ("`/*@get(d1 d2 sum)*/`") is used to fetch the attributes *d1*, *d2* and *sum* and create Rexx context variables from them which can be immediately used in the routine. First the values are displayed, then *d1* and *d2* get random values from Rexx and the *sum* variable gets recalculated. All three Rexx variable then get shown again with their new values, however the attributes in the *ScriptContext Bindings* do not get changed. Upon return the Java host would still access the original attribute values from the *ScriptContext Bindings*.
- The public routine named *two* has the same Rexx statements as routine

30 Cf. 3.1 Evaluating Rexx Scripts with javax.script on page 7 above.

31 The "prolog" code of a Rexx program consists of all Rexx statements starting with line one up to, but not including the first directive. If there is no directive the prolog is the same as the entire Rexx program.

```
scriptContext=arg(arg())~scriptContext -- get ScriptContext
d1=scriptContext~getAttribute("d1")    -- get attribute
d2=scriptContext~getAttribute("d2")    -- get attribute
sum=scriptContext~getAttribute("sum")  -- get attribute
say "d1="d1", d2="d2", sum="sum

/* get attributes with Rexx script annotations  */
::routine one public
  /*@get(d1 d2 sum)*/                    -- get attributes
  say "d1="pp(d1)", d2="pp(d2)", sum="pp(sum)
  d1=random()
  d2=random()
  sum=d1+d2
  say "d1="pp(d1)", d2="pp(d2)", sum="pp(sum)

/* get and set attributes with Rexx script annotations  */
::routine two public
  /*@get(d1 d2 sum)*/                    -- get attributes
  say "d1="pp(d1)", d2="pp(d2)", sum="pp(sum)
  d1=random()
  d2=random()
  sum=d1+d2
  say "d1="pp(d1)", d2="pp(d2)", sum="pp(sum)
  say "--> ---> now updating Bindings from Rexx! <--- <--"
  /*@set(d1 d2 sum)*/  -- replace the values in the Bindings

::routine pp   -- "pretty-print": enclose argument in brackets
  return "["arg(1)"]"
```

Code 7: "test_rexx_03.rex": Employing Rexx script annotations.

one, but adds a concluding RexxScript annotation ("`/*@set(d1 d2 sum)*/`") which will update the respective *ScriptContext Bindings* attributes with the values the Rexx context variables hold at that point in time. Upon return the Java host would now access the attribute values from the *ScriptContext Bindings*, i.e. the values as changed by the Rexx program.

The Java program "*Test_03.java*" in code 8 below creates a *RexxScriptEngine* and fetches its default *ScriptContext*, then uses its *ENGINE_SCOPE Bindings* to store the Rexx program's filename and uses its *GLOBAL_SCOPE Bindings* to store the attributes *d1* with the value *1*, *d2* with the value *2* and *sum* with the value *3*. Then the Java static method *showAttributes* is used to lookup the three attributes *d1*, *d2* and *sum* and display their current values.

The Rexx program "*test_rexx_03.rex*" in code 7 above will get evaluated and the Rexx prolog code will lookup the attributes *d1*, *d2* and *sum* in the supplied *ScriptContext* and display them without changing their values in its *GLOBAL_SCOPE Bindings*. Its public routines named *one* and *two* will be made

```
import javax.script.*;
import java.io.FileReader;

public class Test_03   // demo evaluating a Rexx script
{
    public static void main (String args[])
    {
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine            rse = manager.getEngineByName("Rexx");
        try
        {
           String filename="test_rexx_03.rex"; // define the filename
          // add the filename to the engine's SimpleBindings
            ScriptContext sc=rse.getContext();  // get the default ScriptContext
            sc.setAttribute(ScriptEngine.FILENAME,filename,ScriptContext.ENGINE_SCOPE);
            sc.setAttribute("d1",  "1", ScriptContext.GLOBAL_SCOPE);
            sc.setAttribute("d2",  "2", ScriptContext.GLOBAL_SCOPE);
            sc.setAttribute("sum", "3", ScriptContext.GLOBAL_SCOPE);
            showAttributes(sc);
            rse.eval(new FileReader(filename), sc);  // now let us execute the Rexx script

            System.out.println("\n... about to call public Rexx routine 'one':");
            // now let us execute a global Rexx routine, forward slotDir argument!
            rse.eval("call one arg(arg())");
            showAttributes(sc);

            System.out.println("\n... about to call public Rexx routine 'two':");
            // now let us execute a global Rexx routine, forward slotDir argument!
            rse.eval("call two arg(arg())");
            showAttributes(sc);
        }
        catch (Exception exc)
        {
            System.err.println(exc);
            System.exit(-1);
        }
    }
    public static void showAttributes(ScriptContext sc)
    {
        System.out.println("... d1=["+sc.getAttribute("d1")+"]"
                           + ", d2=["+sc.getAttribute("d2")+"]"
                          + ", sum=["+sc.getAttribute("sum")+"] ...");
    }
}
```

Code 8: "Test_03.java": Java host program to host "test_rexx_03.rex".

available to any Rexx program that will be evaluated in that *RexxScriptEngine* instance later, such that any Rexx script can invoke the *one* or *two* routine without a need to require "*test_rexx_03.rex*"[32].

Next, the Java host evaluates (executes) a single line Rexx program that invokes the public Rexx routine *one* from the package "*test_rexx_03.rex*". This single line

32 Please note that this behavior is not the default ooRexx behavior! As noted above *RexxScriptEngine* implements this behavior (adding all prior packages automatically to a new Rexx package, thereby making all public routines or classes available) to match the behavior of other script languages, especially in the context of scripts embedded in HTML. This *RexxScriptEngine* feature can be turned off by the Java host by using its setter method *setAddLatestPackageDynamically(false)*.

```
... d1=[1], d2=[2], sum=[3] ...
REXXout>d1=1, d2=2, sum=3

... about to call public Rexx routine 'one':
REXXout>d1=[1], d2=[2], sum=[3]
REXXout>d1=[591], d2=[762], sum=[1353]
... d1=[1], d2=[2], sum=[3] ...

... about to call public Rexx routine 'two':
REXXout>d1=[1], d2=[2], sum=[3]
REXXout>d1=[31], d2=[262], sum=[293]
REXXout>--> ---> now updating Bindings from Rexx! <--- <--
... d1=[31], d2=[262], sum=[293] …
```

Output 3: Output of running "java Java_03".

Rexx script will receive the BSF4ooRexx slot argument (always the last argument) which will be fetched by this Rexx single line program and supplied as the only argument to the public Rexx routine *one*, such that the contained *ScriptContext* is made available to it as well.[33] Upon return the Java program uses the static method *showAttributes* to display the current (unchanged by the Rexx script) values of the attributes *d1*, *d2* and *sum*.

Lastly, the Java host evaluates (executes) another single line Rexx program that invokes the public Rexx routine *two* from the package "*test_rexx_03.rex*". This single line Rexx script will receive the BSF4ooRexx slot argument (always the last argument) which will be fetched by this Rexx single line program and supplied as the only argument to the public Rexx routine *two*, such that the contained *ScriptContext* is made available to it as well.[33] Upon return the Java program uses the static method *showAttributes* to display the current (this time changed by the Rexx script *@set* RexxScript annotation) values of the attributes *d1*, *d2* and *sum*.

4 Roundup and Outlook

This article introduced a *javax.script* compliant implementation for the ooRexx script language in the BSF4ooRexx function and class package, named "*RexxScript*". It explained the fundamental concepts of the *javax.script* framework which are drove the design of the BSF4ooRexx implementation.

A few nutshell examples introduce and demonstrate various Java host applications

33 As mentioned above, RexxScript annotations are dependent on the existence of the *ScriptContext* object as the last argument of the invoked routine or method.

that evaluate (execute, run) Rexx scripts. The *RexxScriptEngine* implementation introduces "RexxScript annotations" to ease fetching and setting attributes from the current *ScriptContext Bindings*.

By default public routines and classes of evaluated (executed) Rexx scripts will be made available to any Rexx script that gets evaluated later by the same *RexxScriptEngine*, which matches the behavior of other *javax.script* languages, but deviates from the default ooRexx behavior.

As each *RexxScriptEngine* instance will create and use a separate Rexx interpreter instance, thorough testing of both, the *RexxScriptEngine* and of ooRexx will be necessary.[34]

5 References

[1] JSR-223 Homepage (as of 2017-04-01): https://jcp.org/en/jsr/detail?id=223

[2] JSR-223 Specification (as of 2017-04-01): https://jcp.org/aboutJava/communityprocess/edr/jsr223/index.html

[3] Javadocs for the Java package `javax.script` (as of 2017-04-01): https://docs.oracle.com/javase/8/docs/api/javax/script/package-summary.html

[4] Sourceforge homepage BSF4ooRexx (acronym for "bean scripting framework – BSF – for ooRexx"), an ooRexx and Java bridge (as of 2017-04-01): https://sourceforge.net/projects/bsf4oorex

[5] Sourceforge homepage ooRexx ("open object-oriented Rexx"), a dynamically typed scripting language (as of 2017-04-01): https://sourceforge.net/projects/oorexx

[6] Homepage of Java (as of 2017-04-01): http://java.com

[7] Homepage of the Apache Software Foundation (ASF) "Bean Scripting Framework (BSF)" (as of 2017-04-01): https://commons.apache.org/proper/commons-bsf/

[8] Download page for ooRexx 4.0 (as of 2017-04-01):

34 The ooRexx interpreter has a very powerful kernel which allows any number of Rexx interpreter instances to execute in parallel, each capable of running Rexx programs in different threads. This particular feature has been exploited in BSF4ooRexx since the new, powerful kernel got introduced with ooRexx 4.0. [12]

https://sourceforge.net/projects/oorexx/files/oorexx/4.0.0/

[9] Download page for ooRexx 5.0 beta (as of 2017-04-01): https://sourceforge.net/projects/oorexx/files/oorexx/5.0.0beta/

[10] Java Native Interface (JNI) Specifications (as of 2017-04-01): https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html

[11] Flatscher R.G.: "The 2009 Edition of BSF4Rexx", in: Proceedings of the “The 2009 International Rexx Symposium”, Chilworth, England, Great Britain, May 18th – May 21st 2009. URL (as of 2017-04-01): http://wi.wu.ac.at/rgf/rexx/orx20/2009_orx20_BSF4ooRexx-20091031-article.pdf

[12] Flatscher R.G.: "Creating ooRexx Interpreter Instances from Java and NetRexx", in: Proceedings of the "The 2012 International Rexx Symposium", Raleigh, North Carolina, U.S.A., May 14th – 16th, 2012. URL (as of 2017-04-01): http://wi.wu.ac.at/rgf/rexx/orx23/201202-CreatingOoRexxInterpreterInstances-article.pdf