2017 RexxLA International Rexx Language Symposium Proceedings

René Vincent Jansen (ed.)

THE REXX LANGUAGE ASSOCIATION
RexxLA Symposium Proceedings Series
ISSN 1534-8954

Publication Data

A publication of **RexxLA Press**

The responsible publisher of this edition is identified as *IBizz IT Services and Consultancy*, Amsteldijk 14, 1074 HR Amsterdam, a registered company governed by the laws of the Kingdom of The Netherlands.

The RexxLA Symposium Series is registered under ISSN 1534-8954
The 2017 edition is registered under ISBN 978-94-648-5753-5

ISSN 1534-8954

9 771534 895004 >

ISBN 978-94-648-5753-5

9 789464 857535 >

2023-05-31 First printing

Introduction

History of the International Rexx Language Symposium

In 1990, Cathie Dager of SLAC[1] convened the organizing committee for the first independent Rexx[2] Symposium for Developers and Users. SLAC continued to organize this annual event until the middle of the 1990's when the RexxLA took over that responsibility. Symposia have been held annually since 1990.

About RexxLA

During the 1993 Symposium in La Jolla, California, plans for a Rexx User Group materialized. The Rexx Language Association (RexxLA), as it was called, is an independent, non-profit organization dedicated to promoting the use and understanding of the Rexx programming language. RexxLA manages several open source implementations of Rexx.

The selection procedure

Presentation proposals are solicited yearly using a CFP[3] procedure, after which the RexxLA symposium comittee reviews them and votes which presentations are selected for the symposium. The presentations are peer reviewed before being presented. Presenters are not compensated for their presentations.

Location

The 2017 symposium was held in The Netherlands from 9 Apr 2017 to 12 Apr 2017.

[1] Stanford Linear Accelerator Center, since 2008 SLAC National Accelerator Laboratory

[2] Cowlishaw, M. F., **The REXX Language** (second edition), ISBN 0-13-780651-5, Prentice-Hall, 1990.

[3] Call For Papers.

Contents

1

Open Object Rexx Tutorial – Rony G. Flatscher

Date and Time

9 Apr 2017, 13:30:00 CET

Presenter

Rony G. Flatscher

Presenter Details

Rony works as a professor for Business informatics ("Wirtschaftsinformatik") at the Vienna University of Economics and Business Administration (Wirtschaftsuniversität Wien) and uses Open Object Rexx for teaching Business Administration and MIS students the object-oriented paradigm, as well as remote-controlling (automating) Windows and Windows end-user applications (e.g. MS Office, Open Office) as well as Java and Java applications (he is the author of BSF4ooRexx, the ooRexx-Java bridge, which uses Apache BSF and had Rony invited to become an ASF member). He consults and trains in all of his research fields.

"Leaping from Classic to Object"

2017 International Rexx Symposium

Amsterdam, The Netherlands
(April 2017)

© 2017 Rony G. Flatscher (Rony.Flatscher@wu.ac.at)
Wirtschaftsuniversität Wien, Austria (http://www.wu.ac.at)

Agenda

- History
- Getting Object Rexx
- New procedural features
- New object-oriented features
- Roundup

History, 1

- Begin of the 90s
  - OO-version of Rexx presented to the IBM user group "SHARE"
  - Developed since the beginning of the 90'ies
  - 1997 Introduced with OS/2 Warp 4
    - *Support of SOM and WPS*
  - 1998 Free Linux version, trial version for AIX
  - 1998 Windows 95 and Windows/NT

History, 2

- 2004
  - Spring: RexxLA and IBM join in negotiations about opensourcing Object REXX
  - November: RexxLA gets sources from IBM
  - Opensource developers taking responsibility
    - David Ashley, USA, OS2 guru, Linux freak, ooRexx aficionado
    - Rick McGuire, USA, original lead developer
    - Mark Hessling, Australia, Regina maintainer, author of numerous great, opensource, openplatform Rexx function packages
    - Rony G. Flatscher, Austria (Europe!), author of BSF4Rexx, ooRexx tester of many years
- 2005
  - Spring (March/April): RexxLA makes ooRexx freely available as opensource and openplatform
    - **2005-03-25: ooRexx 3.0**

History, 3

- Summer 2009
  - ooRexx 4.0.0
  - Kernel fully rewritten
    - 32-bit *and* 64-bit versions possible for the first time
    - New OO-APIs into the ooRexx kernel
      - e.g. BSF4ooRexx allows for implementing Java methods in Rexx !
- Latest release as of April 2017
  - ooRexx 4.2, Feb 24, 2014
  - AIX, Linux, MacOSX, Windows
- ooRexx 5.0 in beta

Getting "Open Object Rexx" ("ooRexx") … for Free!

- http://www.RexxLA.org
  - Choose the link to "ooRexx"
- http://www.ooRexx.org
  - Homepage for ooRexx
  - Links to Sourceforge
    - Source
    - Precompiled versions for AIX, Linux (Debian, K/Ubuntu, Red Hat, Suse,), MacOSX, Solaris, Windows
    - Consolidated (great!) PDF- and HTML-rendered documentation!

New Procedural Features, 1

- Fully compatible with classic Rexx, TRL 2
  - New: execution of a Rexx program
    - *Full syntax check of the Rexx program*
    - *Interpreter carries out all directives (leadin with "::")*
    - *Start of program*
- "rexxc.exe": explicit tokenization of Rexx programs
- **USE** ARG in addition to PARSE ARG
  - among other things allows for retrieving stems by reference (!)

Example (ex_stem.rex) "USE ARG" with a Stem

```
/* ex_stem.rex: demonstrating USE ARG  */

info.1 = "Hi, I am a stem which could not get altered in a procedure!"
info.0 = 1              /* indicate one element in stem                         */
call work info.         /* call procedure which adds another element (entry)    */
do i=1 to info.0        /* loop over stem                                       */
   say info.i           /* show content of stem.i                               */
end
exit

work: procedure
   use arg great.       /* note the usage of "USE ARG" instead of "PARSE ARG"   */
   idx = great.0 + 1    /* get number of elements in stem, enlarge it by 1      */
   great.idx = "Object Rexx allows to directly access and manipulate a stem!"
   great.0 = idx        /* indicate new number of elements in stem              */
   return

/* yields:

   Hi, I am a stem which could not get altered in a procedure!
   Object Rexx allows to directly access and manipulate a stem!
*/
```

New Procedural Features, 2

- Routine-directive
  - same as a function/procedure
  - if public, then even callable from another (!) program
- Requires-directive
  - allows for loading programs ("modules") with public routines and public classes one needs
- User definable exceptions

OO-Features Simply Usable by Classic Rexx Programs

- "Environment"
  - a directory object
    - *allows to store data with a key (a string)*
    - *sharing information (coupling of) among different Rexx programs*
  - "**.local**"
    - *available to all Rexx programs within the same Rexx interpreter instance in a process*
  - "**.environment**"
    - *available to all Rexx programs running under all Rexx interpreter instances within the same process*
    - *gets searched after* ***.local***

Example (dec2roman.rex) Classic Style

```
/* dec2roman.rex: turn decimal number into Roman style   */
Do forever
   call charout "STDOUT:", "Enter a number in the range 1-3999: "; PARSE PULL number
   If number = 0 then exit
   say "   --->" number "=" dec2rom(number)
End

dec2rom: procedure
  PARSE ARG num, bLowerCase     /* mandatory argument: decimal whole number              */
  a.        = ""
        /* 1-9 */      /* 10-90 */    /* 100-900 */   /* 1000-3000 */
  a.1.1  = "i"   ; a.2.1  = "x"   ; a.3.1  = "c"   ; a.4.1  = "m"   ;
  a.1.2  = "ii"  ; a.2.2  = "xx"  ; a.3.2  = "cc"  ; a.4.2  = "mm"  ;
  a.1.3  = "iii" ; a.2.3  = "xxx" ; a.3.3  = "ccc" ; a.4.3  = "mmm" ;
  a.1.4  = "iv"  ; a.2.4  = "xl"  ; a.3.4  = "cd"  ;
  a.1.5  = "v"   ; a.2.5  = "l"   ; a.3.5  = "d"   ;
  a.1.6  = "vi"  ; a.2.6  = "lx"  ; a.3.6  = "dc"  ;
  a.1.7  = "vii" ; a.2.7  = "lxx" ; a.3.7  = "dcc" ;
  a.1.8  = "viii"; a.2.8  = "lxxx"; a.3.8  = "dccc";
  a.1.9  = "ix"  ; a.2.9  = "xc"  ; a.3.9  = "cm"  ;
  IF num < 1 | num > 3999 | \DATATYPE(num, "W") THEN
  DO
     SAY num": not in the range of 1-3999, aborting ..."
     EXIT -1
  END

  num = reverse(strip(num))     /* strip & reverse number to make it easier to loop      */
  tmpString = ""
  DO i = 1 TO LENGTH(num)
     idx = SUBSTR(num,i,1)
     tmpString = a.i.idx || tmpString
  END

  bLowerCase = (translate(left(strip(bLowerCase),1)) = "L")    /* default to uppercase   */
  IF bLowerCase THEN RETURN           tmpString
                ELSE RETURN TRANSLATE(tmpSTring)               /* x-late to uppercase    */
```

Example (routine1_dec2roman.rex)

```
/* routine1_dec2roman.rex: initialization */
  a.        = ""
        /* 1-9 */      /* 10-90 */    /* 100-900 */   /* 1000-3000 */
  a.1.1  = "i"   ; a.2.1  = "x"   ; a.3.1  = "c"   ; a.4.1  = "m"   ;
  a.1.2  = "ii"  ; a.2.2  = "xx"  ; a.3.2  = "cc"  ; a.4.2  = "mm"  ;
  a.1.3  = "iii" ; a.2.3  = "xxx" ; a.3.3  = "ccc" ; a.4.3  = "mmm" ;
  a.1.4  = "iv"  ; a.2.4  = "xl"  ; a.3.4  = "cd"  ;
  a.1.5  = "v"   ; a.2.5  = "l"   ; a.3.5  = "d"   ;
  a.1.6  = "vi"  ; a.2.6  = "lx"  ; a.3.6  = "dc"  ;
  a.1.7  = "vii" ; a.2.7  = "lxx" ; a.3.7  = "dcc" ;
  a.1.8  = "viii"; a.2.8  = "lxxx"; a.3.8  = "dccc";
  a.1.9  = "ix"  ; a.2.9  = "xc"  ; a.3.9  = "cm"  ;
.local~dec.2.rom = a.                  /* save in .local-environment for future use      */

::routine dec2roman public
  PARSE ARG num, bLowerCase            /* mandatory argument: decimal whole number       */

  a. = .local~dec.2.rom                /* retrieve stem from .local-environment          */
  IF num < 1 | num > 3999 | \DATATYPE(num, "W")THEN
  DO
     SAY num": not in the range of 1-3999, aborting ..."
     EXIT -1
  END

  num = reverse(strip(num))     /* strip & reverse number to make it easier to loop      */
  tmpString = ""
  DO i = 1 TO LENGTH(num)
     idx = SUBSTR(num,i,1)
     tmpString = a.i.idx || tmpString
  END

  bLowerCase = (translate(left(strip(bLowerCase),1)) = "L")    /* default to uppercase   */
  IF bLowerCase THEN RETURN           tmpString
                ELSE RETURN TRANSLATE(tmpSTring)               /* x-late to uppercase    */
```

Example (use_routine1_dec2roman.rex)

```
/* use_routine1_dec2roman.rex */
Do forever
   call charout "STDOUT:", "Enter a number in the range 1-3999: "
   PARSE PULL number
   If number = 0 then exit
   say "   --->" number "=" dec2roman(number)
End

::requires "routine1_dec2roman.rex" /* directive to load module with public routine  */
```

Example (routine2_dec2roman.rex)

```
/* routine2_dec2roman.rex: Initialization code  */
 d1    = .array~of( "", "i", "ii", "iii", "iv", "v", "vi", "vii", "viii", "ix" )
 d10   = .array~of( "", "x", "xx", "xxx", "xl", "l", "lx", "lxx", "lxxx", "xc" )
 d100  = .array~of( "", "c", "cc", "ccc", "cd", "d", "dc", "dcc", "dccc", "cm" )
 d1000 = .array~of( "", "m", "mm", "mmm"                                       )
.local~roman.arr = .array~of( d1, d10, d100, d1000 )    /* save in local environment    */

::ROUTINE dec2roman PUBLIC              /* public routine to translate number into Roman*/
  USE ARG num, bLowerCase               /* mandatory argument: decimal whole number     */

  IF num < 1 | num > 3999 | \DATATYPE(num, "W") THEN
     RAISE USER NOT_A_VALID_NUMBER      /* raise user exception                          */

  num = num~strip~reverse       /* strip & reverse number to make it easier to loop     */
  tmpString = ""
  DO i = 1 TO LENGTH(num)
     tmpString = .roman.arr[i] ~at(SUBSTR(num,i,1)+1) || tmpString
  END

  bLowerCase = (bLowerCase~strip~left(1)~translate = "L")       /* default to uppercase */
  IF bLowerCase THEN RETURN           tmpString
                ELSE RETURN TRANSLATE(tmpSTring)                 /* x-late to uppercase  */
```

Example (use_routine2_dec2roman.rex)

```
/* use_routine2_dec2roman.rex */
Do forever
   call charout "STDOUT:", "Enter a number in the range 1-3999: "
   PARSE PULL number
   If number = 0 then exit
   say "   --->" number "=" dec2roman(number)
End

::requires "routine2_dec2roman.rex" /* directive to load module with public routine  */
```

New Object-oriented Features, 1

- Allows for implementing abstract data types (ADT)
  - "Data Type" (DT)
    - *a data type defines the set of valid values*
    - *a data type defines the set of valid operations for it*
    - *examples*
      - *numbers: adding, multiplying, etc*
      - *strings: translating case, concatenating, etc.*
  - "Abstract Data Type" (ADT)
    - *a generic schema defining a data type with*
      - *attributes*
      - *operations on attributes*

New Object-oriented Features, 2

- Object-oriented features of Rexx
  - allow for implementing an ADT
  - a predefined classification tree
  - allow for (multiple) inheritance
  - explicit use of metaclasses
  - tight security manager (!)
    - *allows for implementing any security policy w.r.t. Rexx programs*
      - *untrusted programs from the net*
      - *roaming agents*
      - *company policy w.r.t. executing code in secured environment*

About Implementing ADTs, 1

- Rexx and ADTs
  - Cannot define routines confined to a datatype!
  - Attributes can be encoded as
    - Rexx strings, e.g.

      ```
      birthday="19590520 13:01"
      ```
    - Rexx stems, e.g.

      ```
      birthday.date="19590520"
      Birthday.time="13:01"
      ```
  - Quite complicated and can be error prone
    - Rexx programmers must know exactly the structure and all operations to implement!

About Implementing ADTs, 2

- ooRexx
  - – Designed to easily implement ADTs
  - – Directives

```
::CLASS adt_name
::ATTRIBUTE attr_name
::METHOD meth_name
```

  - – An implemented ADT is sometimes termed "class", sometimes "type", sometimes "structure"
  - – "Black box"
    - Rexx users do not need to know any implementation details in order to use classes/types/structures !

About Objects and Messages

- "object"
  - – A synonym for "value of a specific type", "instance"
  - – Possesses all attributes and methods of its class
  - – Only reacts upon receiving messages
    - Message operator ~ (tilde, dubbed "twiddle")
    - Followed by a *message name*, optionally with arguments in parenthesis
    - Searches and invokes the method with the same name as the message name and returns any return values from the method

Example (dog.rex)
Defining Dogs ...

```
/* dog.rex: a program for dogs ...  */

myDog = .Dog~new        /* create a dog from the class          */
myDog~name = "Sweety"   /* tell the dog its name                */
say "My name is:" myDog~name    /* now ask the dog for its name */
myDog~bark              /* come on show them who you are!       */

::class  Dog            /* name of the implemented ADT          */
::attribute name        /* let it have an attribute             */
::method bark           /* let it be able to bark               */
  say "Woof! Woof! Woof!"

/* yields:

   My name is: Sweety
   Woof! Woof! Woof!

*/
```

Example (bigdog.rex)
Defining BIG Dogs ...

```
/* bgdoc.rex: a program for BIG dogs ...  */

myDog = .BigDog~new     /* create a BIG dog from the class      */
myDog~name = "Arnie"    /* tell the dog its name                */
say "My name is:" myDog~name    /* now ask the dog for its name */
myDog~Bark              /* come on show them who you are!       */

::class  Dog            /* define the class "Dog"               */
::attribute name        /* let it have an attribute             */
::method bark           /* let it be able to bark               */
  say "Woof! Woof! Woof!"


  /* the following class reuses most of what is already
     defined for the class "Dog" via inheritance; it overrides
     the way a big dog barks                                    */
::class  BigDog subclass Dog    /* define the class "BigDog"    */
::method bark           /* let it be able to bark like big dogs
                          do, all in uppercase! :)  */
  say "WOOF! WOOF! WOOF!"


/* yields:

   My name is: Arnie
   WOOF! WOOF! WOOF!

*/
```

New Object-oriented Features, 3

- Object Rexx' classification tree
  - Fundamental classes
    - *Object, Class, Method, Message*
  - Classic Rexx classes
    - *String, Stem, Stream*
  - Collection classes
    - *Array, CircularQueue, List, Queue, Supplier*
    - *Directory, Properties, Relation and Bag, Table, Set*
      - *index is set explicitly by programs*
  - Miscellaneous classes
    - *Alarm, Monitor, ...*

Example (fruit.rex) A Bag Full of Fruits ...

```
/* fruit.rex: a bag, full of fruits ...   */

Fruit_Bag = .bag~of( "apple", "apple", "pear", "cherry", "apple", "banana",         ,
                     "plum", "plum", "banana", "apple", "pear", "papaya",           ,
                     "peanut", "peanut", "peanut", "peanut", "peanut", "apple",     ,
                     "peanut", "pineapple", "banana", "plum", "pear", "pear",       ,
                     "plum", "plum", "banana", "apple", "pear", "papaya",           ,
                     "peanut", "peanut", "peanut", "apple", "peanut", "pineapple", ,
                     "banana", "peanut", "peanut", "peanut", "peanut", "peanut",    ,
                     "apple", "peanut", "pineapple", "banana", "peanut", "papaya", ,
                     "mango", "peanut", "peanut", "apple", "peanut", "pineapple",  ,
                     "banana", "pear" )

SAY "Total of fruits in bag:" Fruit_Bag~items
SAY

Fruit_Set = .set~new~union(Fruit_Bag)
SAY "consisting of:"
DO fruit OVER Fruit_Set
   SAY right(fruit, 21) || ":" RIGHT( Fruit_Bag~allat(fruit)~items, 3 )
END
```

Example (fruit.rex) Output

```
Total of fruits in bag: 56

consisting of:
                 plum:   5
               cherry:   1
                 pear:   6
                mango:   1
               banana:   7
               peanut:  20
            pineapple:   4
               papaya:   3
                apple:   9
```

Open Object Rexx ("ooRexx") Roundup

- Adds features, long asked for, e.g.
  - Variables (stems) by reference (USE ARG)
  - Public routines available to other programs (concept of modules)
  - Very powerful and complete implementation of the OO-paradigm
- Availability
  - Free
  - Opensource
  - Openplatform
    - Precompiled versions for: AIX, Linux (rpm, deb), MacOSX, Solaris, Windows 98/NT/2000/XP/Vista/W7/W8
- Rony G. Flatscher, *„Introduction to Rexx and ooRexx“,* order form: *http://www.facultas.at/flatscher*
- TBD*: http://www.RonyRexx.net*

2

Rexx Scripts Hosted and Evaluated by Java – Rony G. Flatscher

Date and Time

10 Apr 2017, 08:00:00 CET

Presenter

Rony G. Flatscher

Presenter Details

Rony works as a professor for Business informatics (”Wirtschaftsinformatik”) at the Vienna University of Economics and Business Administration (Wirtschaftsuniversität Wien) and uses Open Object Rexx for teaching Business Administration and MIS students the object-oriented paradigm, as well as remote-controlling (automating) Windows and Windows end-user applications (e.g. MS Office, Open Office) as well as Java and Java applications (he is the author of BSF4ooRexx, the ooRexx-Java bridge, which uses Apache BSF and had Rony invited to become an ASF member). He consults and trains in all of his research fields.

"*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]