Creating ooRexx Interpreter Instances from Java and NetRexx.The Proceedings of the Rexx Symposium for Developers and Users, pp. 1-23

Flatscher, Rony G.

Published: 01/01/2012

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

Creating ooRexx Interpreter Instances from Java and NetRexx

Rony G. Flatscher (Rony.Flatscher@wu.ac.at), WU Vienna
"The 2012 International Rexx Symposium", Raleigh, North Carolina, U.S.A.
May 14th – 16th, 2012

Abstract. This article introduces the new BSF4ooRexx 4.1 features that allow Java and NetRexx programmers to fully control the configuration and creation of Rexx interpreter instances using ooRexx 4.1 or newer. The functionality exploits the C++ APIs of ooRexx which get first introduced, followed by an overview of the new Java classes that makes it possible and easy for Java and NetRexx programmers to exploit this interesting and powerful infrastructure. Nutshell examples demonstrate how this new functionality can be easily exploited, including the new ability to implement Rexx handlers ("Rexx callbacks") in Java and NetRexx.

1 Introduction

With the introduction of "Open Object Rexx (ooRexx) 4.0" (summer 2009, [1]) the interpreter was rolled out with a totally new kernel[1], which among many improvements also introduced numerous easy to use application programming interfaces (API) for C++ programmers [3]. Among the new features these APIs make available to C++ programmers there is one that stands out: the ability to create many different Rexx interpreter instances in the same process[2], which can be configured differently from each other, including different additional path lookups, Rexx program extensions or defining Rexx command and exit handlers Rexx interpreter instances should run with.

The new "Bean Scripting Framework for ooRexx (BSF4ooRexx)" 4.1[3] [4] introduces a set of classes which provide a little framework for making it possible (and easy) for Java and NetRexx programmers to exploit this part of the ooRexx APIs.

This article will first introduce the C++ APIs of ooRexx 4.x for configuring and starting Rexx interpreter instances, introducing and discussing at a conceptual level the available startup options. Then the new framework for exploiting these APIs from Java and NetRexx in BSF4ooRexx will be documented, followed by subsections highlighting the different startup options using small nutshell programs given in Java and NetRexx.

---

1 This new kernel was developed over many years by the original project lead and main author of IBM's Object REXX product [the sources of which were handed over to the Rexx Language Association (RexxLA) [2] in the fall of 2004 for opensourcing, which happened in 2005] Rick McGuire, who has been the principal architect and lead of the RexxLA opensource project "ooRexx" since its inception.

2 All Rexx programs that are run by one Rexx interpreter instance can share data using its local environment `.local`, all Rexx programs in all Rexx interpreter instances can share data using the global environment `.environment`.

3 Version 4.1 of BSF4ooRexx is slated for general availability (GA) at the 2012 International Rexx Symposium which is planned for the middle of May 2012.. This version can be used in conjunction with any installed Java that complies to one of the following Java versions: 1.4, 1.5, 1.6/6, or 1.7/7. Notabene: there is no need for the JDK (Java development kit) to fully exploit BSF4ooRexx, an installed Java runtime environment (JRE) suffices!

2 Creation of ooRexx Interpreter Instances

This chapter first introduces briefly all of the C++ Rexx interpreter API options that control the configuration of the Rexx interpreter instance that gets created. Following this brief introduction the new Java framework implemented in BSF4ooRexx gets explained and demonstrated with numerous nutshell examples.

2.1 The C++ Rexx Interpreter API

[3], which may get installed with ooRexx on some operating systems as "rexxpg.pdf", documents the C++ Rexx Interpreter API in the section entitled "9.1 Rexx Interpreter API". Going with it are all the options with which a Rexx interpreter instance can be configured before creating it. In the following subsections the options available to C++ programmers are briefly described, the reader is directed to the abovementioned documentation for more details [3].

Option APPLICATION_DATA. This option allows a C++ program to store a `void *` pointer, such that Rexx interpreter instance related data can be maintained by the C++ program.

Option EXTERNAL_CALL_PATH. This option allows to supply a string which contains additional paths Rexx should search for finding a called Rexx program. This string needs to be formatted like the string of the PATH environment variable of the target operating system. An example for the Windows operating system would be the string `".\anotherpath"`, if a subdirectory named `"anotherpath"` located in the current directory should be searched in addition.

Option EXTERNAL_CALL_EXTENSION. This option allows to define additional Rexx program file suffixes in a comma separated string that the interpreter should search for. An example would be the string `".rxj,.rxo,.rxjo,.jrexx,.abc,.xyz"`.

Option LOAD_REQUIRED_LIBRARY. This option allows to define an array of strings, each denoting the name of a library as one would give with the `::REQUIRES "name_of_library" LIBRARY` directive.

Option REGISTER_LIBRARY. Allows to register Rexx routines and Rexx methods that are implemented in the C++ module.

Option DIRECT_EXITS. Allows to register an array of Rexx exit handlers, which are functions that serve as callbacks for the Rexx interpreter.

Option DIRECT_ENVIRONMENTS. Allows to register an array of Rexx (sub-)command handlers which are functions that serve as callbacks for the Rexx interpreter.

Option INITIAL_ADDRESS_ENVIRONMENT. Allows to denote the default environment name that should be addressed, a string.

2.2 The BSF4ooRexx APIs for the Rexx Interpreter API

The core BSF4ooRexx package consists of a combination of Java classes and a native (C++) "Java native interface (JNI)" binding between Java and the Rexx interpreter APIs.

From the perspective of a Java (NetRexx[4]) programmer a Rexx interpreter instance gets created with the following sequence of Java statements:

- create an instance of `org.apache.bsf.BSFManager`, which is able to manage one Rexx interpreter instance,
- create a `RexxEngine` instance using `BSFManager`'s method `loadScriptingEngine("rexx")`, which is of type `org.rexxla.bsf.engines.rexx.RexxEngine`.

Prior to BSF4ooRexx version 4.1 a Rexx interpreter instance was automatically created when the Rexx engine got created. Starting with BSF4ooRexx version 4.1 this creation of a Rexx interpreter instance is now postponed until the Rexx engine gets used to for the first time to interpret a Rexx program. Until then it is possible to configure the Rexx engine with the Rexx interpreter API options the Java programmer wishes to employ.

The new `RexxEngine` method `getRexxConfiguration()` returns the Rexx engine's Rexx configuration object of type `org.rexxla.bsf.engines.rexx.RexxConfiguration` which possesses the methods to query and set the Rexx interpreter instance options.

A `RexxEngine`'s `RexxConfiguration` object can be changed as long as no Rexx interpreter instance got created. Once a Rexx interpreter instance exists any attempt to change the Rexx configuration illegally will cause an `org.apache.bsf.BSFException` exception to be thrown.

Of all of the C++ interpreter APIs the following are available to Java programmers and introduced one by one in the following subsections:

- EXTERNAL_CALL_PATH,
- EXTERNAL_CALL_EXTENSION,
- LOAD_REQUIRED_LIBRARY,

4 In this article any reference to Java or Java programmers is also meant to adhere to NetRexx and NetRexx programmers.

- DIRECT_EXITS,
- DIRECT_ENVIRONMENTS,
- INITIAL_ADDRESS_ENVIRONMENT.

The options APPLICATION_DATA, REGISTER_LIBRARY, REGISTERED_EXITS and REGISTERED_ENVIRONMENTS are C++ or C++ module specific and therefore do not apply to Java and are not supported.

2.2.1 Option EXTERNAL_CALL_PATH

The option `EXTERNAL_CALL_PATH` allows to set an additional search path to be used when the Rexx interpreter searches for Rexx programs. It needs to be formed according to the operating system's convention.

According to the Rexx reference book [5], chapter "7.2.1.1. Locating External Rexx Files" the following search sequence is carried out by the interpreter:

1. the same directory from which the program was loaded that calls an external program,
2. the current directory,
3. the directories specified in the extension path defined with the option `EXTERNAL_CALL_PATH`,
4. the directories specified with the `REXX_PATH` environment variable, and finally
5. the directories specified with the `PATH` environment variable.

Java programmers can fetch the operating system's file separator with `java.lang.System.getProperty("file.separator")` and the operating system's path separator with

```
import org.apache.bsf.*;
import org.rexxla.bsf.engines.rexx.*;

public class SamplePathExtension
{
    public static void main (String args[]) throws BSFException
    {
        BSFManager mgr       =new BSFManager();     // create an instance of BSFManager
        RexxEngine rexxEngine=(RexxEngine) mgr.loadScriptingEngine("rexx");
            // Configure the RexxEngine
        RexxConfiguration rexxconf=rexxEngine.getRexxConfiguration();
            // add ".\anotherpath" (Windows), "./anotherpath" (Unix)
        String externalPath="."+System.getProperty("file.separator")+"anotherpath";
        rexxconf.setExternalCallPath(externalPath);
            // invoke the interpreter and run the Rexx program
        String rexxCode= "call 'testSamplePathExtension.rex' ";
        rexxEngine.apply ("SamplePathExtension.rex", 0, 0, rexxCode, null, null);
        rexxEngine.terminate();     // terminate Rexx engine (Rexx interpreter instance)
    }
}
```

Code 1: A Java Program Adding an Additional Search Path to the Rexx interpreter instance.

```
mgr        = org.apache.bsf.BSFManager()      -- create an instance of BSFManager
rexxEngine = org.rexxla.bsf.engines.rexx.RexxEngine mgr.loadScriptingEngine("rexx")

    -- Configure the RexxEngine
rexxconf=rexxEngine.getRexxConfiguration()
System.err.println("SamplePathExtension.java, Rexx configuration:\n\n"rexxconf"\n")
    -- add ".anotherpath" (Windows), "./anotherpath" (Unix)
rexxconf.setExternalCallPath("."System.getProperty("file.separator")"anotherpath");

    -- Rexx code to run (quote filename for Unix filesystems)
rexxCode= "call 'testSamplePathExtension.rex' "
    -- invoke the interpreter and run the Rexx program
rexxEngine.apply("nrxSamplePathExtension.rex", 0, 0, rexxCode, null, null)
rexxEngine.terminate()     -- terminate Rexx engine (Rexx interpreter instance)
```

Code 2: A NetRexx Program Adding an Additional Search Path to the Rexx interpreter instance.

`java.lang.System.getProperty("path.separator")`, such that it becomes possible for them to create the extension path according to the rules of the host operating system.

The `RexxConfiguration` class supplies the following two methods for setting and getting this option's value:

- `void setExternalCallPath(String path) throws BSFException`: sets or replaces the additional search path. Must not be used, once the Rexx interpreter instance got created.
- `String getExternalCallPath()`: returns the current additional search path.

The Java program in Code 1 above and the NetRexx program in Code 2 above configure a `RexxEngine` and execute a Rexx program which calls another (external) Rexx program `'testSamplePathExtension.rex'`. When searching for this external Rexx program the interpreter will look in step 3 in the subdirectory named `"anotherpath"` in the current directory.

2.2.2 Option EXTERNAL_CALL_EXTENSION

The option `EXTERNAL_CALL_EXTENSION` allows to set a comma separated list of additional file extensions to be used when the Rexx interpreter searches for external Rexx programs that have no file extension[5] defined.

According to the Rexx reference book [5], chapter "7.2.1.1. Locating External Rexx Files" the following search sequence is carried out by the interpreter for names without file extensions:[6]

1. the file extension of the calling Rexx program is appended, and the resulting Rexx program is searched in the usual directories,
2. the file extensions as given in the option EXTERNAL_CALL_EXTENSION are appended

5 A file extension is led in by a dot "." and followed by one or more characters.

6 On case sensitive file systems Rexx searches first the files using the supplied (exact) case and if not found the search is repeated with the filename all in lowercase.

```
import org.apache.bsf.*;
import org.rexxla.bsf.engines.rexx.*;

public class SamplePathExtension
{
    public static void main (String args[]) throws BSFException
    {
        BSFManager mgr       =new BSFManager();     // create an instance of BSFManager
        RexxEngine rexxEngine=(RexxEngine) mgr.loadScriptingEngine("rexx");
            // Configure the RexxEngine
        RexxConfiguration rexxconf=rexxEngine.getRexxConfiguration();
            // add ".abc,.xyz" to default call extensions
        rexxconf.setExternalCallExtensions(rexxconf.getExternalCallExtensions()+",.abc,.xyz");
            // invoke the interpreter and run the Rexx program
        String rexxCode= "call 'testSampleFileExtension' ";
        rexxEngine.apply ("SamplePathExtension.rex", 0, 0, rexxCode, null, null);
        rexxEngine.terminate();     // terminate Rexx engine (Rexx interpreter instance)
    }
}
```

Code 3: A Java Program Defining Additional File Extensions for the Rexx interpreter instance.

one after each other and for each resulting Rexx program name the usual directories are searched,

3. the default system file extension `".REX"` is appended and the resulting Rexx program is searched in the usual directories,
4. finally, the Rexx program is searched as is in the usual directories.

The `RexxConfiguration` class supplies the following two methods for setting and getting this option's value:

- `void setExternalCallExtension(String fileExtension) throws BSFException`: sets or replaces the additional file extensions, a comma separated list of file extensions. Must not be used, once the Rexx interpreter instance got created.
- `String getExternalCallExtension()`: returns the current list of additional file extensions.

The Java program in Code 3 above and the NetRexx program in Code 4 below configure a `RexxEngine` and execute a Rexx program which calls another (external) Rexx program

```
mgr       = org.apache.bsf.BSFManager()      -- create an instance of BSFManager
rexxEngine = org.rexxla.bsf.engines.rexx.RexxEngine mgr.loadScriptingEngine("rexx")
    -- Configure the RexxEngine
rexxconf=rexxEngine.getRexxConfiguration()
    -- add ".abc,.xyz" to default call extensions
rexxconf.setExternalCallExtensions(rexxconf.getExternalCallExtensions()",.abc,.xyz")

    -- Rexx code to run (quote filename for Unix filesystems)
rexxCode= "call 'testSampleFileExtension.rex' "
    -- invoke the interpreter and run the Rexx program
rexxEngine.apply("SampleFileExtension.rex", 0, 0, rexxCode, null, null)
rexxEngine.terminate()     -- terminate Rexx engine (Rexx interpreter instance)
```

Code 4: A NetRexx Program Defining Additional File Extensions for the Rexx interpreter instance.

'testSampleFileExtension'. When searching for this external Rexx program the interpreter will look in step 2 above for files named[7]:

- 'testSampleFileExtension***.rxj***'
- 'testSampleFileExtension***.rxo***'
- 'testSampleFileExtension***.rxjo***'
- 'testSampleFileExtension***.jrexx***'
- 'testSampleFileExtension***.abc***'
- 'testSampleFileExtension***.xyz***'

2.2.3 Option LOAD_REQUIRED_LIBRARY

The option LOAD_REQUIRED_LIBRARY enables one to have multiple external Rexx function libraries loaded, before any Rexx program gets executed. This way one can ensure that Rexx programs do not need to require/call these libraries themselves.

The RexxConfiguration class supplies the following two methods for adding and getting this option's value:

- void addRequiredLibrary(String libraryName) throws BSFException: adds libraryName to load. Must not be used, once the Rexx interpreter instance got created.
- String[] getRequiredLibraries(): returns a String array of the libraries to load.

```
import org.apache.bsf.*;
import org.rexxla.bsf.engines.rexx.*;

public class SampleRequiredLibrary
{
    public static void main (String args[]) throws BSFException
    {
        BSFManager mgr       =new BSFManager();     // create an instance of BSFManager
        RexxEngine rexxEngine=(RexxEngine) mgr.loadScriptingEngine("rexx");  // load the Rexx engine
        // Configure the RexxEngine
        RexxConfiguration rexxconf=rexxEngine.getRexxConfiguration();
        rexxconf.addRequiredLibrary("rxmath");  // add "rxmath" library, distributed with ooRexx
        // Rexx code to run
        String rexxCode= "parse source . . s  \n" +
                         "say s               \n" +
                         "numeric digits 16   \n" +
                         "say RxCalcPi()        ";
        // invoke the interpreter and run the Rexx program
        rexxEngine.apply ("SampleRequiredLibrary.rex", 0, 0, rexxCode, null, null);
        rexxEngine.terminate();     // terminate Rexx interpreter instance
    }
}
```

Code 5: A Java Program Requiring the ooRexx Math-Library for the Rexx interpreter instance.

7 BSF4ooRexx by default defines the extensions: ".rxj,.rxo,.rxjo,.jrexx".

```
mgr      =org.apache.bsf.BSFManager()      -- create an instance of BSFManager
rexxEngine=org.rexxla.bsf.engines.rexx.RexxEngine mgr.loadScriptingEngine("rexx")

-- Configure the RexxEngine
rexxconf=rexxEngine.getRexxConfiguration()
rexxconf.addRequiredLibrary("rxmath")  -- add "rxmath" library, distributed with ooRexx

-- Rexx code to run
rexxCode= "parse source . . s; say s\n" -
          "numeric digits 16        \n" -
          "say RxCalcPi()             "

-- invoke the interpreter and run the Rexx program
rexxEngine.apply("nrxSampleRequiredLibrary.rex", 0, 0, rexxCode, null, null)
rexxEngine.terminate()      -- terminate Rexx interpreter instance
```

Code 6: A NetRexx Program Requiring the ooRexx Math-Library for the Rexx interpreter instance.

Code 5 above and Code 6 require the ooRexx math library `"rxmath"` to be required, such that the Rexx program that will get executed is able to use all of that library's public Rexx functions/routines.

Running the program as depicted in Code 5 and Code 6 yields the following output, using successfully the external public routine named `RxCalcPi()` from the `"rxmath"` external library:

```
SampleRequiredLibrary.rex [8]
3.141592653589793
```

2.2.4 Defining Rexx Exit and Command Handlers

Rexx Exit and command handlers are Rexx callbacks to C++, which have been extended to Java or NetRexx with the new BSF4ooRexx 4.1. Callback code is usually able to query, set or drop context variables from the Rexx program that caused the callback.

Java Rexx exit and command handlers need to implement the interfaces `org.rexxla.bsf.engines.rexx.RexxExitHandler` and `org.rexxla.bsf.engines.rexx.RexxCommandHandler`, respectively. The first argument passed to the interface methods is of type `Object` and dubbed `slot` . It is an opaque argument that must not be changed in any way and is only intended to be passed on as the first argument to the static methods[9] the abstract class `org.rexxla.bsf.engines.rexx.RexxHandler` defines, if there is a need for the Java or NetRexx Rexx handlers to interface with the ooRexx C++ APIs while processing the callback.

2.2.4.1 Option DIRECT_EXITS

Rexx exits are callbacks for certain processing steps that the Rexx interpreter defines. Table 1

8 The name of the executed Rexx code got set by the first argument of the `RexxEngine.apply(...)` method.

9 Please consult the BSF4ooRexx JavaDocs for the `RexxHandler` class for the available static methods that make some of the ooRexx C++ APIs available to the Rexx Java or NetRexx handlers..

below gives a brief overview of the exits, which are defined in the ooRexx C++ API documentation in [3], "9.12 Rexx Exits Interface".

The interface `org.rexxla.bsf.engines.rexx.RexxExitHandler` defines `final int static` fields for the mnemonics that get used in the ooRexx C++ API and which are also used in Table 1 below. In addition this interface also defines the static fields `RXEXIT_HANDLED` / `0`, `RXEXIT_NOT_HANDLED` / `1`, and `RXEXIT_RAISE_ERROR` / `-1`, which are all values a Rexx exit handler may return. Any Java class that should serve as a Rexx exit callback, needs to implement this interface and its single method:

- `public int handleExit(Object slot, int exitNumber, int subFunction, Object[] parmBlock)`

The argument `slot` is an opaque argument and only needed, if the static methods of the abstract class `org.rexxla.bsf.engines.rexx.RexxHandler` get used, which expect this argument to be passed on. The arguments `exitNumber` and `subFunction` have one of the values that are documented in Table 1 below. The last argument `parmBlock` is an array which matches the respective C++ structures which are documented in the ooRexx C++ API documentation in [3], "9.12 Rexx Exits Interface". If an element of the structure is a structure itself, another array will be created and used in its place on the Java side. If a return value is to be supplied by the Java exit handler, then one needs to place that value into the appropriate index of the `parmBlock` array.

All of the exits with their mnmeonics and integer values, as well as the layout of the `parmBlock` array, which depends on the exit number and its subfunction, is documented in the BSF4ooRexx JavaDocs for the `RexxExitHandler` interface.

The option `DIRECT_EXITS` allows one to set a Java Rexx exit handler for one or more of the Rexx exit numbers. At runtime it is possible to (temporarily) nullify a Java Rexx exit handler by setting the appropriate exit number to `null` or to replace an assigned exit handler with a different one. Note that after a Rexx interpreter instance got created, it is not possible anymore to define exit handlers for new (unhandled) exit numbers.

Exit Number	Subfunction	Purpose
RXFNC / 2		**Process external functions.**
	RXFNCCAL / 1	Processes calls to external functions.
RXCMD / 3		**Process host commands.**
	RXCMDHST / 1	Calls a named subcommand handler.
RXMSQ / 4		**Manipulate queue.**
	RXMSQPLL / 1	Pull a line from queue.
	RXMSQPSH / 2	Place a line on queue.

Exit Number	Subfunction	Purpose
	RXMSQSIZ / 3	Return num of lines on queue.
	RXMSQNAM / 20	Set active queue name.
RXSIO / 5		**Session I/O.**
	RXSIOSAY / 1	SAY a line to STDOUT.
	RXSIOTRC / 2	Trace output.
	RXSIOTRD / 3	Read from char stream.
	RXSIODTR / 4	DEBUG read from char stream.
	RXSIOTLL / 5	Return linelength (OS/2 only).
RXHLT / 7		**Halt processing.**
	RXHLTCLR / 1	Clear HALT indicator.
	RXHLTTST / 2	Test HALT indicator.
RXTRC / 8		**Test ext trace indicator.**
	RXTRCTST / 1	Tests the external trace indicator.
RXINI / 9		**Initialization processing.**
	RXINIEXT / 1	Initialization exit.
RXTER / 10		**Termination processing.**
	RXTEREXT / 1	Termination exit.
RXEXF / 12		**Scripting function call.**
	RXEXFCAL / 1	Processes calls to external functions.
RXNOVAL / 13		**NOVALUE exit.**
	RXNOVALCALL / 1	Processes a Rexx NOVALUE condition.
RXVALUE / 14		**VALUE function exit.**
	RXVALUECALL / 1	Processes an extended call to the VALUE built-in function.
RXOFNC / 15		**Process external functions using object values.**
	RXOFNCCAL / 1	Processes calls to external functions.

Table 1: Rexx Exit and Subfunction Numbers

The `RexxConfiguration` class supplies the following methods for setting and getting this option's values:

- `void addExitHandler(int function, RexxExitHandler exitHandler) throws BSFException`: defines the Rexx exit `function` to be served by the supplied `exitHandler`. Must not be used, once the Rexx interpreter instance got created.
- `RexxExitHandler setExitHandler(int function, RexxExitHandler exitHandler) throws BSFException`: replaces and returns the Rexx exit handler for the given Rexx exit `function`

```
import org.apache.bsf.*;
import org.rexxla.bsf.engines.rexx.*;

public class SampleExitHandler implements RexxExitHandler
{
    public static void main (String args[]) throws BSFException
    {
        BSFManager mgr       =new BSFManager();     // create an instance of BSFManager
        RexxEngine rexxEngine=(RexxEngine) mgr.loadScriptingEngine("rexx");
         // Rexx code to run
        String rexxCode= "call 'testSampleExitHandler.rex' " ;
         // Configure the RexxEngine
        RexxConfiguration rexxconf=rexxEngine.getRexxConfiguration();
        System.err.println("default rexxconf=["+rexxconf+"]\n");
         // add system exits
        rexxconf.addExitHandler(RexxExitHandler.RXVALUE, new SampleExitHandler() );
        System.err.println("edited  rexxconf=["+rexxconf+"]\n");
        // invoke the interpreter and run the Rexx program
        rexxEngine.apply ("SampleExitHandler.rex", 0, 0, rexxCode, null, null);
        rexxEngine.terminate();     // terminate Rexx engine instance
    }
     // implementation of a RXVALUE exit handler
    public int handleExit(Object slot, int exitNumber, int subFunction, Object[] parmBlock)
    {
        System.err.println("(Java side) -> selector=["+parmBlock[0]+"], varName=["+parmBlock[1]+"]"+
                           ", value=["+parmBlock[2]+"]");
        String selector=(String) parmBlock[0];
        if (selector.compareToIgnoreCase("RGF")==0) // o.k., addressed to us, handle it
        {
            if (parmBlock[2]==null) // if value is null, give it some value
            {
                parmBlock[2]="value for variable name ["+parmBlock[1]+"]"+
                             " by a Java Rexx exit handler";
            }
            return RexxExitHandler.RXEXIT_HANDLED;
        }
        return RexxExitHandler.RXEXIT_NOT_HANDLED;
    }
}
```

Code 7: A Java Program Implementing a RXVALUE Rexx Exit Handler.

with the supplied `exitHandler`. Once the Rexx interpreter instance got created this method can be used to nullify the exit handler by supplying `null`, which causes the Rexx interpreter to take on the responsibility of handling the exit, or to change the `exitHandler` that should serve the Rexx exit callbacks from then on.

- `RexxExitHandler getExitHandler(int function)`: returns the exit handler object that serves the Rexx exit with the supplied `function` or `null`, if none is defined at the time of invocation of this method.
- `Object [] getExitHandlers()`: returns an array of type `Object`, which contains two entries: the first entry is an array of type `int` denoting the Rexx exit number for which exit handlers got defined, and the second entry is an array of type `RexxExitHandler` storing the matching Java handler.
- `BitSet getDefinedExits()`: returns a `BitSet` object that indicates for which Rexx exit numbers a Rexx exit handler got defined for.

```
import org.rexxla.bsf.engines.rexx.RexxExitHandler
class nrxSampleExitHandler public implements RexxExitHandler

method main(s=String[]) static
    mgr        = org.apache.bsf.BSFManager()      -- create an instance of BSFManager
    rexxEngine = org.rexxla.bsf.engines.rexx.RexxEngine mgr.loadScriptingEngine("rexx")
        -- Configure the RexxEngine
    rexxconf=rexxEngine.getRexxConfiguration()
        -- add exit handler
    rexxconf.addExitHandler(RexxExitHandler.RXVALUE, nrxSampleExitHandler() );
    say "nrxSampleExitHandler.nrx, edited rexxonf=["rexxconf"]\n"
        -- Rexx code to run (quote filename for Unix filesystems)
    rexxCode= "call 'testSampleExitHandler.rex' "
        -- invoke the interpreter and run the Rexx program
    rexxEngine.apply("nrxSampleExitHandler.rex", 0, 0, rexxCode, null, null)
    rexxEngine.terminate()     -- terminate Rexx engine (Rexx interpreter instance)

method handleExit(slot=Object, exitNumber=int, subFunction=int, parmBlock=Object[]) returns int
    if parmBlock[2]=null then
        say "(NetRexx side) -> selector=["parmBlock[0]"], varName=["parmBlock[1]"]"
    else
        say "(NetRexx side) -> selector=["parmBlock[0]"], varName=["parmBlock[1] -
            "], value=["parmBlock[2]"]"

    selector=String parmBlock[0]
    if selector="RGF" then      -- o.k., addressed to us, handle it
    do
        if (parmBlock[2]==null) then    -- if value is null, give it some value
        do
            parmBlock[2]="value for variable name ["parmBlock[1]"] by a Java Rexx exit handler"
        end
        return RexxExitHandler.RXEXIT_HANDLED
    end
    return RexxExitHandler.RXEXIT_NOT_HANDLED
```

Code 8: A NetRexx Program Implementing a RXVALUE Rexx Exit Handler.

```
say "(Rexx side) value('abc',,'RGF')           ="pp(value('abc',,'RGF'))
say

say "(Rexx side) value('def',,'rGf')           ="pp(value('def',,'rGf'))
say

say "(Rexx side) value('ghi','na,sowas!','RGF')="pp(value('ghi','na,sowas!','RGF'))

::requires "BSF.CLS"    -- get access to the public routine pp()
```

Code 9: A Rexx Program Using the VALUE Built-in Function (BIF).

```
E:\exitHandler>java SampleExitHandler
default rexxconf=[org.rexxla.bsf.engines.rexx.RexxConfiguration[initialAddressEnvironment=[null],
externalCallPath=[null],externalCallExtensions=[.rxj,.rxo,.rxjo,.jrexx],loadRequiredLibrary={},
exitHandlers={},commandHandlers={}]]

edited  rexxconf=[org.rexxla.bsf.engines.rexx.RexxConfiguration[initialAddressEnvironment=[null],
externalCallPath=[null],externalCallExtensions=[.rxj,.rxo,.rxjo,.jrexx],loadRequiredLibrary={},
exitHandlers={RXVALUE/14/SampleExitHandler@1a679b7},commandHandlers={}]]

(Java side) -> selector=[RGF], varName=[abc], value=[null]
(Rexx side) value('abc',,'RGF')           =[value for variable name [abc] by a Java Rexx exit handler]

(Java side) -> selector=[rGf], varName=[def], value=[null]
(Rexx side) value('def',,'rGf')           =[value for variable name [def] by a Java Rexx exit handler]

(Java side) -> selector=[RGF], varName=[ghi], value=[na,sowas!]
(Rexx side) value('ghi','na,sowas!','RGF')=[na,sowas!]
```

Code 10: Output of Running Code 7 above.

The Java program in Code 7 above and its NetRexx counterpart in Code 8 above implement a Rexx exit handler for the `RXVALUE` exit and run the Rexx program in Code 9 above, which employs the Rexx `VALUE` built-in function causing the implemented Rexx exit to be invoked, each time an unknown value is supplied for the third argument `"selector"`. The exit handler's implementation checks caselessly whether the selector is the value the exit was written for, and if so, handles the callback: if the second argument to the `VALUE` BIF is `null`, then a new value is created for it. Running the Java program from the command line yields the output that is shown in Code 10 above.

Employing BSF4ooRexx makes it even possible to implement a Rexx exit handler in Rexx itself, cf. Code 18, page 22!

The installation of BSF4ooRexx 4.1 will create a `"samples/Java/handlers/exitHandlers"` subdirectory which contains additonal examples of Java[10] implemented Rexx exit handlers for each Rexx exit ooRexx 4.1.1 defines.

2.2.4.2 Options DIRECT_ENVIRONMENTS and INITIAL_ADDRESS_ENVIRONMENT

Named Rexx environments are callbacks that Rexx programs can address commands to by using the Rexx `ADDRESS` keyword instruction. These commands are plain strings for the command handler. "Command handlers" are sometimes also named "Subcommand handlers", which in this article is therefore used as a synonym.

The interface `org.rexxla.bsf.engines.rexx.RexxCommandHandler` must be implemented for Rexx environment handlers written in Java or NetRexx, which in effect means that one needs to code its single method:

- `public Object handleCommand(Object slot, String address, String command)`

The argument `slot` is an opaque argument and only needed, if the static methods of the abstract class `org.rexxla.bsf.engines.rexx.RexxHandler` get used, which expect this argument to be passed on. The argument `address` contains the name of the addressed subcommand handler, which makes it possible to implement different subcommand handlers in this method, such that the handling of the command is dependent on the actual addressed environment. The last argument `command` is a string containing the command to handle.

If a return value is to be supplied by the Java or NetRexx command handler, then one needs to merely return that value, which may be any object. The Rexx program will be able to fetch this

10 It is left as an exercise of NetRexx programmers to implement the same logic in NetRexx by using the transcriptions from Java to NetRexx of this article.

```
import org.apache.bsf.*;
import org.rexxla.bsf.engines.rexx.*;

public class SampleCommandHandler implements RexxCommandHandler
{
    public static void main (String args[]) throws BSFException
    {
        BSFManager mgr       =new BSFManager();     // create an instance of BSFManager
        RexxEngine rexxEngine=(RexxEngine) mgr.loadScriptingEngine("rexx");  // load the Rexx engine
        // Configure the RexxEngine
        RexxConfiguration rexxconf=rexxEngine.getRexxConfiguration();
        // add command handler
        rexxconf.addCommandHandler("TEST1", new SampleCommandHandler());
        System.err.println("edited rexxconf=["+rexxconf+"]\n");
        // Rexx code to run
        String rexxCode= "call 'testSampleCommandHandler.rex' ";
        // invoke the interpreter and run the Rexx program
        rexxEngine.apply ("SampleCommandHandler.rex", 0, 0, rexxCode, null, null);
        rexxEngine.terminate();     // terminate Rexx interpreter instance
    }

    int counter=0;  // count # of undefined commands
        // implementation of the Rexx command handler:
    public Object handleCommand(Object slot, String address, String command)
    {
    System.err.println("address=["+address+"], command=["+command+"]");
        if (command.compareToIgnoreCase("Hi")==0) {return "Hi, who are you?";}
        else if (command.compareToIgnoreCase("one plus two")==0) {return "3";}
        else if (command.compareToIgnoreCase("please panic a little bit")==0)
        {
            RexxHandler.raiseException1(slot, 35900, this+": You asked for this exception!");
            return null;
        }
            // undefined command
        counter++;
        return "Undefined command # "+counter+": ["+command+"]";
    }
}
```

Code 11: A Java Program Implementing a Rexx Command Handler.

return value after having addressed the subcommand handler by accessing the Rexx variable named `RC`, which will be set by the Rexx interpreter to this returned value. If no return value is to be returned then Java or NetRexx should return `null`, which will cause the number `0` to be assigned to the Rexx variable `RC` by the Rexx interpreter upon return.

In order to register one or more Rexx command handlers the Rexx interpreter option named `DIRECT_ENVIRONMENTS` needs to be exploited.

The `RexxConfiguration` class supplies the following methods for setting and getting this option's values:

- `void addCommandHandler(String name, RexxCommandHandler commandHandler) throws BSFException`: defines the `name` of a Rexx environment and the `commandHandler` to serve commands sent to it. Must not be used, once the Rexx interpreter instance got created.
- `RexxCommandHandler setCommandHandler(String name, RexxCommandHandler commandHandler) throws BSFException`: replaces and returns the Rexx command handler

```
class nrxSampleCommandHandler public implements org.rexxla.bsf.engines.rexx.RexxCommandHandler

properties private
    counter=0   -- count # of undefined commands

method main(s=String[]) static
    mgr        = org.apache.bsf.BSFManager()      -- create an instance of BSFManager
    rexxEngine = org.rexxla.bsf.engines.rexx.RexxEngine mgr.loadScriptingEngine("rexx")
        -- Configure the RexxEngine
    rexxconf=rexxEngine.getRexxConfiguration()
        -- add command handler
    rexxconf.addCommandHandler("TEST1", nrxSampleCommandHandler())
    say "nrxSampleCommandHandler.nrx, edited rexxonf=["rexxconf"]\n"
        -- Rexx code to run (quote filename for Unix filesystems)
    rexxCode= "call 'testSampleCommandHandler.rex' "
        -- invoke the interpreter and run the Rexx program
    rexxEngine.apply("nrxSampleCommandHandler.rex", 0, 0, rexxCode, null, null)
    rexxEngine.terminate()     -- terminate Rexx engine (Rexx interpreter instance)

method handleCommand(slot=Object, address=String, command=String) returns Object
    say "address=["address"], command=["command"]"
    if command="Hi"               then return "Hi, who are you?"
    else if command="one plus two" then return String("3")
    else if command="please panic a little bit" then
    do
        org.rexxla.bsf.engines.rexx.RexxHandler.raiseException1(slot, 35900, -
                this": You asked for this exception!")
        return null
    end
        -- undefined command
    counter=counter+1
    return "Undefined command #" counter": ["command"]"
```

Code 12: A NetRexx Program Implementing a Rexx Command Handler.

for the Rexx command environment with the given `name` and with the supplied `commandHandler`. Once the Rexx interpreter instance got created this method can be used to change a command handler serving the environment with the given `name` at runtime. Supplying `null` is not allowed for command handlers and would cause an appropriate `BSFException` to be thrown.

- `RexxCommandHandler getCommandHandler(String name)`: returns the command handler object that serves the Rexx environment with the given `name`.
- `Object [] getCommandHandlers()`: returns an array of type `Object`, which contains two entries: the first entry is an array of type `String` denoting the names of the defined command handlers, and the second entry is an array of type `RexxCommandHandler` storing the matching Java handlers.

In the context of command handlers it is possible to configure a Rexx interpreter instance to start out with a default environment that should be addressed, if the Rexx programmer does not explicitly use the `ADDRESS` keyword instruction. By default the Rexx interpreter addresses the current shell as the subcommand environment. The `RexxConfiguration` class supplies the following methods for

```
address test1 "hi"
say "rc="pp2(rc)
say

address test1  -- change address permanently
one  plus  two
say "rc="pp2(rc)
say

call testException

"nothing to do?"
say "rc="pp2(rc)
say

::requires "rgf_util2.rex" -- get public routines pp2(), ppCondition2()

::routine testException -- send the command that raises an exception
  signal on any
  address test1 "please panic a little bit"
  return
any:
  say ppCondition2(condition('Object'))
  say
  return
```

Code 13: A Rexx Program Using the "TEST1" environment.

setting and getting this option's values:

- `void setInitialAddressEnvironment(String name) throws BSFException`: defines the `name` environment to be used as the default environment for this Rexx interpreter instance. Must not be used, once the Rexx interpreter instance got created.
- `String getInitialAddressEnvironment()`: returns the name of the initial Rexx environment

```
E:\commandHandler>java SampleCommandHandler
edited rexxconf=[org.rexxla.bsf.engines.rexx.RexxConfiguration[initialAddressEnvironment=[null],
externalCallPath=[null],externalCallExtensions=[.rxj,.rxo,.rxjo,.jrexx],loadRequiredLibrary={},
exitHandlers={},commandHandlers={TEST1=SampleCommandHandler@1a679b7}]]

address=[TEST1], command=[hi]
rc=[Hi, who are you?]

address=[TEST1], command=[ONE PLUS TWO]
rc=[3]

address=[TEST1], command=[please panic a little bit]
        [ADDITIONAL] =[an Array (1 items) id#_266374438]
                [SampleCommandHandler@1a679b7: You asked for this exception!]
        [CODE]       =[35.900]
        [CONDITION]  =[SYNTAX]
        [DESCRIPTION]=[]
        [ERRORTEXT]  =[Invalid expression]
        [INSTRUCTION]=[SIGNAL]
        [MESSAGE]    =[SampleCommandHandler@1a679b7: You asked for this exception!]
        [PACKAGE]    =[a Package id#_266374529]
        [POSITION]   =[20]
        [PROGRAM]    =[E:\commandHandler\testSampleCommandHandler.rex]
        [PROPAGATED] =[1]
        [RC]         =[35]
        [TRACEBACK]  =[a List (0 items) id#_266374513]


address=[TEST1], command=[nothing to do?]
rc=[Undefined command # 1: [nothing to do?]]
```

Code 14: Output of Running Code 11 above.

or `null`, if none was set.

The Java program in Code 11 above and its NetRexx counterpart in Code 12 above implement a Rexx command handler and run the Rexx program in Code 13 above, which sends commands to the subcommand handler. The command handler's implementation checks caselessly all commands and acts appropriately. In the case that commands are not defined to be handled, a counter gets increased for each such unknown command and the supplied command gets returned, enclosed in square brackets and prepended with the string `"Undefined command # "` followed by the current value of the `counter`. Running the Java program from the command line yields the output that is shown in Code 14 above.

Employing BSF4ooRexx makes it even possible to implement a Rexx command handler in Rexx itself, cf. Code 19, page 23!

For further examples of Java implemented Rexx command handlers please consult the subdirecotry `"samples/Java/handlers/commandHandlers"` which gets created by BSF4ooRexx 4.1 and which contains additional samples of Java[11] implemented Rexx command handlers.

3 Roundup and Outlook

BSF4ooRexx 4.1, with a planned release of spring 2012, extends its functionality for Java and/or NetRexx programmers allowing them to fully control the options with which Rexx interpreter instances can be created. The following Rexx interpreter instance options can be easily used by Java and/or NetRexx programmers using the appropriate methods of the new class `org.rexxla.bsf.engines.rexx.RexxConfiguration` of which an instance gets stored with each `RexxEngine` object. The creation of a Rexx interpreter instance gets delayed as long as possible, ie. only when a Rexx program is executed via the `RexxEngine` methods `apply(…)`, `call(…)`, or `eval(…)`. The following configuration methods are available for setting (and querying) the options for a Rexx interpreter instance:

- `EXTERNAL_CALL_PATH`, `RexxConfiguration` methods: `setExternalCallPath(…)`, `getExternalCallPath()`,
- `EXTERNAL_CALL_EXTENSION`, `RexxConfiguration` methods: `setExternalCallExtensions(…)`, `getExternalCallExtensions()`,
- `LOAD_REQUIRED_LIBRARY`, `RexxConfiguration` methods: `addRequiredLibrary(…)`,

11 It is left as an exercise of NetRexx programmers to implement the same logic in NetRexx by using the transcriptions from Java to NetRexx of this article.

getRequiredLibraries(),

- DIRECT_EXITS, RexxConfiguration methods: addExitHandler(…), setExitHandler(…), getExitHandler(…), getExitHandlers(), getDefinedExitHandlers(),
- DIRECT_ENVIRONMENTS, RexxConfiguration methods: addCommandHandler(…), setCommandHandler(…), getCommandHandler(…), getCommandHandlers(),
- INITIAL_ADDRESS_ENVIRONMENT, RexxConfiguration methods: setInitialAddressEnvironment(…), getInitialAddressEnvironment(…).

There have been Java interfaces defined for Rexx exit handlers (`org.rexxla.bsf.engines.rexx.RexxExitHandler`) and command handlers (`org.rexxla.bsf.engines.rexx. RexxCommandHandler`), which Java handlers need to implement. The `RexxExitHandler` interface class in addition defines Java constant values that exactly match the ooRexx C++ API constants and its JavaDocs define exactly the structure of the `parmBlock` argument with which the exit handlers need to interact with.

Rexx exit and command handlers implemented in Java or NetRexx can take advantage of the abstract class `RexxHandler`, which supplies implemented static methods for direct interaction with specific C++ APIs in the same thread that invoked the handlers. To do so, the Java handlers must supply the received argument `slot` as the first argument to those static methods as documented in the JavaDocs and further in the ooRexx C++ APIs in [3], "9.14 Rexx Interface Methods Listing". Here is an overview of the static methods the abstract class `RexxHandler` implements:

- getContextVariable(…), setContextVariable(…), setContextVariableToNil(…), dropContextVariable(…), getAllContextVariables(…),
- raiseCondition(…), raiseException(…), checkCondition(…), clearCondition(…), getConditionInfo(…),
- getCallerContext(…), getLocalEnvironment(…), getGlobalEnvironment(…), getNil(…),
- haltThread(…), setThreadTrace(…),
- getLanguageLevel(…), getInterpreterVersion(…).

The BSF4ooRexx 4.1 installation creates a subdirectory named `"samples/Java/handlers"` underneath its home directory which demonstrate with numerous examples the implementation of Rexx handlers in Java. These nutshell examples serve as teasers, but also as complete, self-contained Java handlers, and can be easily transcribed to NetRexx, if the reader studies the

differences between Java and NetRexx programs as demonstrated by the code examples in this very article.

As BSF4ooRexx allows ooRexx to use all of Java as if it was ooRexx (e.g. using the camouflaging support of the ooRexx package named `"BSF.CLS"`), without using any Java code directly, it would be an interesting experiment to create Rexx programs that can be used instead of the Java and the NetRexx programs depicted in this article. As BSF4ooRexx is intended to allow exactly for replacing Java by pure ooRexx and allowing Rexx to create instances of Java classes and interact with Java objects by sending pure ooRexx messages, such an experiment can be carried out. Chapter 5 "Appendix: Using ooRexx instead of Java to Configure another Rexx Interpreter Instance", starting on page 20 gives for each Java program of the article the matching "pure ooRexx" counterpart, exploiting the camouflaging support available from the BSF4ooRexx package.

Acknowledgements

The author wishes to thank DI Walter Pachl for his suggestions and proof-reading.

4 References

[1] Homepage of “Open Object Rexx (ooRexx)” (as of 2012-03-17): http://www.ooRexx.org

[2] Homepage of the “Rexx Language Association (RexxLA)” (as of 2012-03-17): http://www.RexxLA.org

[3] "Open Object Rexx - Programming Guide" (as of 2012-03-17): http://www.oorexx.org/docs/rexxpg/rexxpg.pdf

[4] Homepage of the Java language binding for ooRexx (“BSF4ooRexx”, as of 2012-03-17): http://sourceforge.net/projects/bsf4oorexx/

[5] "Open Object Rexx – Reference" (as of 2012-03-17): http://www.oorexx.org/docs/rexxref/rexxref.pdf

5 Appendix: Using ooRexx instead of Java to Configure another Rexx Interpreter Instance

This chapter documents ooRexx programs that can be used as full replacement of the Java code examples of this article.

5.1 ooRexx Program "Option EXTERNAL_CALL_PATH"

The ooRexx Program in Code 15 below implements the same functionality as the Java program in Code 1 on page 4.

```
rexxEngine=.bsf~new("org.apache.bsf.BSFManager")~loadScriptingEngine("rexx")

    -- Configure the RexxEngine
rexxconf=rexxEngine~getRexxConfiguration
    -- add ".anotherpath" (Windows), "./anotherpath" (Unix)
fs=.java.lang.System~getProperty("file.separator")
rexxconf~setExternalCallPath("."fs"anotherpath")

    -- Rexx code to run (quote filename for Unix filesystems)
rexxCode= "call 'testSamplePathExtension.rex' "
    -- invoke the interpreter and run the Rexx program
rexxEngine~apply("nrxSamplePathExtension.rex", 0, 0, rexxCode, .nil, .nil)
rexxEngine~terminate    -- terminate Rexx engine (Rexx interpreter instance)

::requires BSF.CLS       -- get Java support
```

Code 15: An ooRexx Program Adding an Additional Search Path to the Rexx interpreter instance.

5.2 ooRexx Program "Option EXTERNAL_CALL_EXTENSION"

The ooRexx Program in Code 16 below implements the same functionality as the Java program in Code 3 on page 6.

```
rexxEngine=.bsf~new("org.apache.bsf.BSFManager")~loadScriptingEngine("rexx")

    -- Configure the RexxEngine
rexxconf=rexxEngine~getRexxConfiguration
    -- add ".abc,.xyz" to default call extensions
rexxconf~setExternalCallExtensions(rexxconf~getExternalCallExtensions()",.abc,.xyz")

    -- Rexx code to run (quote filename for Unix filesystems)
rexxCode= "call 'testSampleFileExtension' "
    -- invoke the interpreter and run the Rexx program
rexxEngine~apply("SampleFileExtension.rex", 0, 0, rexxCode, .nil, .nil)
rexxEngine~terminate    -- terminate Rexx engine (Rexx interpreter instance)

::requires BSF.CLS       -- get Java support
```

Code 16: An ooRexx Program Defining Additional File Extensions for the Rexx interpreter instance.

5.3 ooRexx Program "Option LOAD_REQUIRED_LIBRARY"

The ooRexx Program in Code 17 below implements the same functionality as the Java program in Code 5 on page 7.

```
rexxEngine=.bsf~new("org.apache.bsf.BSFManager")~loadScriptingEngine("rexx")

-- Configure the RexxEngine
rexxconf=rexxEngine~getRexxConfiguration
rexxconf~addRequiredLibrary("rxmath")   -- add "rxmath" library, distributed with ooRexx

-- Rexx code to run
rexxCode= "parse source . . s; say s;" -
          "numeric digits 16        ;" -
          "say RxCalcPi()            "

-- invoke the interpreter and run the Rexx program
rexxEngine~apply("rexSampleRequiredLibrary.rex", 0, 0, rexxCode, .nil, .nil)
rexxEngine~terminate()      -- terminate Rexx interpreter instance

::requires BSF.CLS       -- get Java support
```

Code 17: A Java Program Requiring the ooRexx Math-Library for the Rexx interpreter instance.

5.4 ooRexx Program "Option DIRECT_EXITS"

The ooRexx Program in Code 18 below implements the same functionality as the Java program in Code 7 on page 11. Please note that because of the ooRexx camouflaging support, the Java array `parmBlock` is indexed starting with `1` as if it was an ooRexx array!

```
   -- prepare another Rexx interpreter instance besides the current one for this Rexx program
clzName="org.rexxla.bsf.engines.rexx.RexxExitHandler"
clz=bsf.loadClass(clzName)
.local~RXEXIT_HANDLED    =clz~RXEXIT_HANDLED
.local~RXEXIT_NOT_HANDLED=clz~RXEXIT_NOT_HANDLED

rexxEngine=.bsf~new("org.apache.bsf.BSFManager")~loadScriptingEngine("rexx")
rexxconf=rexxEngine~getRexxConfiguration
proxy=BsfCreateRexxProxy(.sampleExitHandler~new, ,clzName)
rexxconf~addExitHandler(clz~RXVALUE, proxy)
say "rexSampleExitHandler.rxj, edited rexxconf="pp(rexxconf~toString)
say

rexxCode= "call 'testSampleExitHandler.rex' "
   -- invoke the interpreter and run the Rexx program
rexxEngine~apply("from_rexSampleExitHandler.rex", 0, 0, rexxCode, .nil, .nil)
rexxEngine~terminate       -- terminate Rexx engine (Rexx interpreter instance)


::requires BSF.CLS   -- get Java support
-----------------------------------------------------------------------------------------
::class SampleExitHandler  -- Rexx class implementing the "handleCommand" interface

::method handleExit     -- Rexx exit handler implemented in Rexx!
  use arg slot, exitNumber, subFunction, parmBlock

  say "(RexxExitHandler side) -> selector=["parmBlock[1] -
      "], varName=["parmBlock[2]"], value=["parmBlock[3]"]"

  if parmBlock[1]~caselessEquals("RGF") then    -- o.k., addressed to us, handle it
  do
      if (parmBlock[3]==.nil) then  -- if value is null, give it some value
          parmBlock[3]="value for variable name ["parmBlock[2]"] by a Java Rexx exit handler"

      return .RXEXIT_HANDLED
  end
  return .RXEXIT_NOT_HANDLED
-----------------------------------------------------------------------------------------
```

Code 18: An ooRexx Program Implementing a RXVALUE Rexx Exit Handler.

5.5 ooRexx Program "Option DIRECT_ENVIRONMENTS"

The ooRexx Program in Code 19 below implements the same functionality as the Java program in Code 11 on page 14.

```
   -- prepare another Rexx interpreter instance besides the current one for this Rexx program
rexxEngine=.bsf~new("org.apache.bsf.BSFManager")~loadScriptingEngine("rexx")
rexxconf=rexxEngine~getRexxConfiguration
proxy=BsfCreateRexxProxy(.sampleCommandHandler~new,,"org.rexxla.bsf.engines.rexx.RexxCommandHandler")
rexxconf~addCommandHandler("TEST1", proxy)
say "rexSampleCommandHandler.rxj, edited rexxconf="pp(rexxconf~toString)
say

rexxCode= "call 'testSampleCommandHandler.rex' "
   -- invoke the interpreter and run the Rexx program
rexxEngine~apply("from_rexSampleCommandHandler.rex", 0, 0, rexxCode, .nil, .nil)
rexxEngine~terminate       -- terminate Rexx engine (Rexx interpreter instance)


::requires BSF.CLS   -- get Java support

::class SampleCommandHandler  -- Rexx class implementing the "handleCommand" interface

::method init        -- needed to define the "counter" attribute and set it to "0"
  expose counter
  counter=0

::method handleCommand  -- Rexx command handler implemented in Rexx!
  expose counter
  use arg slot, address, command

  say "address=["address"], command=["command"]"
  if command~caselessEquals("Hi")               then return "Hi, who are you?"
  else if command~caselessEquals("one plus two") then return 3
  else if command~caselessEquals("please panic a little bit") then
     raise syntax 35.900 array (self": You asked for this exception!")

     -- undefined command
  counter=counter+1
  return "Undefined command #" counter": ["command"]"
```

Code 19: An ooRexx Program Implementing a Rexx Command Handler.