# “THE AUGSBURG VERSION[1)] OF BSF4REXX”

**Rony G. Flatscher**

University of Augsburg, Germany

„The 2003 International Rexx Symposium“, Raleigh, North Carolina, USA,

May 4th - May 7th, 2003.

## ABSTRACT

"BSF4Rexx", the Bean Scripting Framework for Rexx, allows one to use the Rexx and Object Rexx programming languages with the open source IBM Bean Scripting Framework (BSF) which enables Java programs to easily invoke scripts and programs written in another language than Java. This article introduces the "Augsburg version" of BSF4Rexx which incorporates numerous changes and as a main feature the ability to start Java from Rexx programs. This way all of Java can be viewed as a huge external Rexx function library from the perspective of Rexx, available on any platform Rexx is available. This paper gives a bird eyes view of BSF4Rexx concentrating on this latter ability and introducing Rexx programmers informally to Java and to the most important object-oriented terms such that the unacquainted Rexx and Object Rexx programmer becomes able to read the Java documentation and as a result apply BSF4Rexx to allow (Object) Rexx to use and drive Java.

[1)] This article reflects a major change in BSF4Rexx implemented right after the 2003 International Rexx symposium on the way back to Europe: calling Java from (Object) Rexx does not require (and as a matter of fact even prohibits) the exact indication of the types of Java arguments meant for Java constructors or Java methods! This simplifies the invoking of Java constructors and methods considerably for Rexx and hence is being incorporated into the final version of this article.

# 1 INTRODUCTION

At the 2001 International Rexx symposium the first incarnation of the "BSF4Rexx" - the "Bean Scripting Framework for Rexx" - got introduced to the Rexx community [Flat01]. Taking advantage of the IBM open source project "Bean Scripting Framework" it has become possible to invoke Rexx programs from Java programs in an easy and straight-forward manner. In turn such invoked Rexx programs could call back into Java and take advantage of the wealth of the functionality implemented in Java classes and made available via Java objects.

Should a Rexx programmer be interested in using the Java functionality for solving problems of his own, then it has been mandatory that his Rexx programs be invoked by Java programs in the first place, as the appropriate BSF environment had to be set up prior to Rexx calling (back) into Java.

This article introduces the reader to the "Augsburg version of BSF4Rexx" which for the first time allows Rexx programmers to directly use Java, without the need to invoke his own Rexx programs indirectly via Java.

This way Java becomes the "largest external function package for Rexx in the world"! In addition that particular "external function package" has been already ported to all commercial relevant operating systems and hardware platforms and beyond!

Drawing from comments of some Rexx programmers in the past it has become clear that many of the Rexx programmers have never been directly exposed to Java programs and as a result have been handicapped in taking advantage of BSF4Rexx. Therefore this article concentrates on explaining and demonstrating a Java program, its interface documentation which looks like any Java documentation and the employing of the functionality of such Java programs from Rexx. It is hoped that a reader being a Rexx programmer who has never been exposed to Java will be able to take advantage of this enormeous and rich set of functionality as a result of studying this article.

# 2 BSF - ARCHITECTURE AND APPLICATION

The "Bean Scripting Framework" (BSF) has begun its life as an IBM alphaworks project, which allows IBM employees to make their work available to the world. In the case that a particular alphaworks project [W3Alpha] draws the attention of other developers it may be the case that such a project turns into a so-called "developer work", which usually means that the project gets more attention and resources from IBM ([W3BSF], but may also be employed in other IBM products like "IBM WebSphere" [W3WebSphere].

BSF defines and implements a Java framework which enables Java programmers to invoke programs from Java, which are written in a non-Java programming language like JavaScript or Perl. The initial support for non-Java programming languages by IBM in BSF 2.2 concentrated on Java implemented interpreters, like Mozilla's Rhino [W3Rhino] or Mike Cowlishaw's[2)] NetRexx [W3NetRexx] and in addition to the Microsoft ActiveX scripting languages JScript and VBScript, available on the Windows platform only.

In the fall of 2002 the entire open source project was handed over to the Jakarta project of the Apache organization and got released with the version number "BSF 2.3". This way the technology can be used in additional Apache funded/initiated open source projects like "`ant`" [W3Ant] or "`xerxes`" [W3Xerces].

## 2.1 The Essener Version of BSF4Rexx

The Essener version of BSF4Rexx [Flat01] is rooted in a feasibility study of a student at the University of Essen [Kal01] and extends the set of usable languages for IBM's BSF by Rexx and Object Rexx. Figure 1 depicts the overall architecture.

Steps for invoking a Rexx program using the Essener version of BSF4Rexx:

- The Java program creates an instance of the Java `BSFManager` class, which allows for loading and executing programs written in any of the BSF supported languages.

2) Mike F. Cowlishaw is the original creator of the Rexx programming language and an IBM fellow.

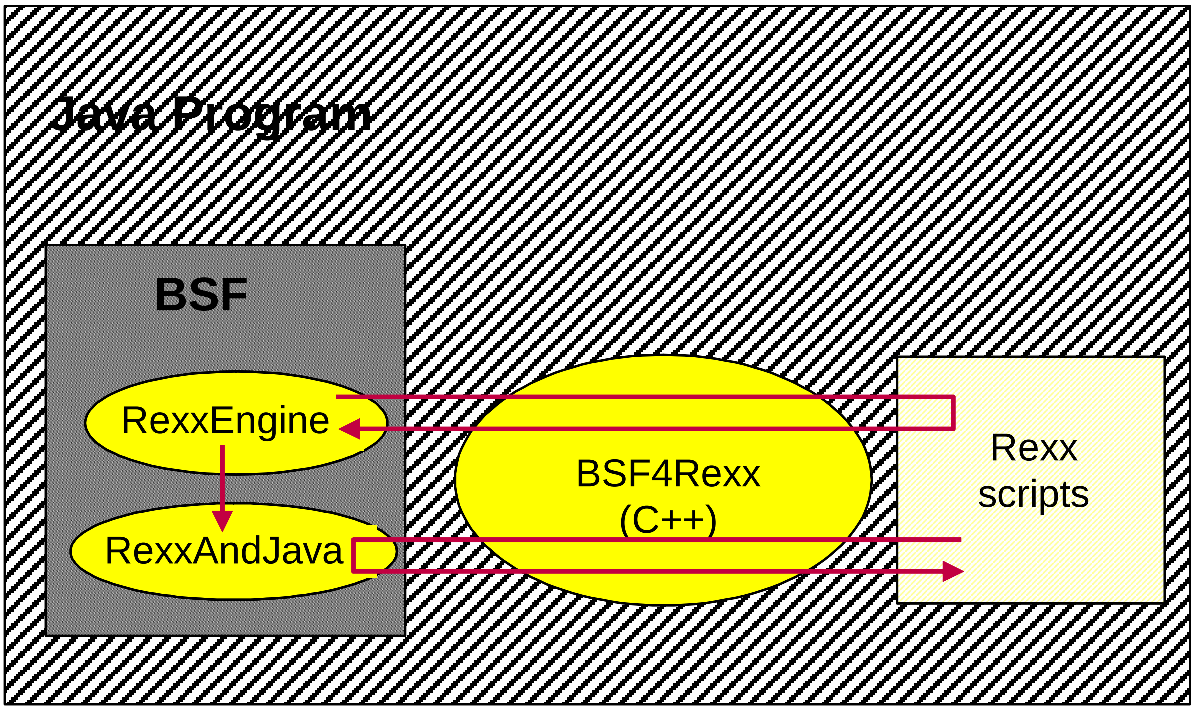


*Figure 1: The BSF Architecture and the Essener Version of BSF4Rexx [Fla01].*

- The Java program uses the `BSFManager` object to load the Rexx engine, which is implemented as a Java program ("`RexxEngine.java`") and which initializes the interface to Rexx, i.e. a compiled C++ program named "`BSF4Rexx`" adhering to the "Java Native Interface" (JNI) specifications.[3] From this moment on it becomes possible for the Rexx engine to invoke Rexx programs by supplying the Rexx code and Rexx arguments for the Rexx interpreter which gets called in "`BSF4Rexx`". Before doing that an external Rexx function named `BSF()` is registered with the Rexx interpreter, which allows the Rexx programs to call back into Java.
- Rexx scripts calling back into Java are able to use a rich set of functions which will get carried out in the Java program "`RexxAndJava.java`" transparently. The scheme is simple: the first argument to the external Rexx function `BSF()` denotes the desired subfunction[4] to be carried out by "`RexxAndJava`". Depending on the subfunction, additional arguments may

[3] "`BSF4Rexx`" is implemented as a DLL under OS/2 resp. eComStation and Windows, and as a shared library under Linux.

[4] E.g. the subfunction to invoke a Java method is called "`invoke`".

```
/* 1 */  import com.ibm.bsf.*;   // BSF support
/* 2 */
/* 3 */  public class TestBSF4Rexx
/* 4 */  {
/* 5 */     public static void main (String[] args)
/* 6 */     {
/* 7 */        try
/* 8 */        {
/* 9 */           BSFManager mgr      = new BSFManager ();
/* 10 */          BSFEngine  rxEngine = mgr.loadScriptingEngine("rexx");
/* 11 */          String     rexxCode = "SAY 'Rexx was here!'";
/* 12 */          rxEngine.exec ("rexx", 0, 0, rexxCode);
/* 13 */       }
/* 14 */       catch (BSFException e)
/* 15 */       {
/* 16 */          e.printStackTrace();
/* 17 */       }
/* 18 */    }
/* 19 */ }
```

*Figure 2: Complete Nutshell Example for a Java Program Invoking a Rexx Program Via BSF.*

have to be supplied by the Rexx program. Upon return from "`RexxAndJava`" a return value will always be supplied to Rexx.[5)]

The Essener version of BSF4Rexx has been allowing Rexx (and Object Rexx for that matter) to be employed in the context of IBM's BSF, making it possible to invoke Rexx programs everywhere where Rhino, VBScript and the like could be used with BSF in the same easy and straight-forward manner by Java programmers.

Figure 2 shows a minimal Java program using BSF4Rexx to create a BSFManager instance, which then is used to load the Rexx scripting engine containing the necessary statements in order to call Rexx from Java using BSF4Rexx:

- Line # 1: the BSF Java package contains all the BSF classes, among them the Java classes `BSFManager` and `BSFEngine`.[6)]
- Line # 9: an instance named "`mgr`" of the `BSFManager` class is created.

5) In the case that Java does not return a value (or a Java "`null`" indicating that no value is available), then the character string "`.NIL`" is returned. One can use this very same string to indicate from Rexx that one does not supply a value for a particular argument; the same effect can be achieved by omitting the argument altogether from the Rexx side.

6) This package refers to IBM's version of BSF, a.k.a. "BSF 2.2". The package name for the Apache version of BSF is "`org.apache.bsf`" and can only be used, if the JNI (Java native interface) C++ program "`BSF4Rexx.cc`" is compiled with an Apache-specific switch.

- Line # 10: using "`mgr`" an instance of `BSFEngine` is created and assigned to a variable named "`rxEngine`".
- Line # 11: a String named "`rexxCode`" is defined containing a Rexx program.[7)]
- Line # 12: the Rexx engine is used to execute the Rexx program contained in the String variable "`rexxCode`".

For a Java programmer the above program is extremely simple and should be very easy to comprehend.

However, Rexx programmers who have never been exposed to Java need to learn quite a few Java fundamentals before being able to take advantage of the Essener version of BSF4Rexx. Besides, if interested in using rather the Java functionality for the purpose of solving problems in Rexx, it has been a little bit awkward that for that purpose alone, one would need to write a Java program to invoke the respective Rexx program which in turn would call back into Java.

It would be simpler and hence more "Rexx-like", if it was possible to control everything from Rexx and if the Rexx programmer needed no Java programming skills at all.

## 2.2 The Augsburg Version of BSF4Rexx

The Augsburg version of BSF4Rexx was developed while at the University of Augsburg and extends the Essener version in three main areas:

- it allows to invoke Java (the Java runtime environment with all of its classes) from Rexx, if needed,
- it removes all restrictions of the Essener version with respect to using Java arrays, hence arbitrarily dimensioned Java arrays can be used,

[7)] Please note that in this nutshell example the Rexx code in line # 11 is explicitly given. Instead, one could read the contents of a file into the variable "`rexxCode`" and pass it to the Rexx engine for execution in line # 12.

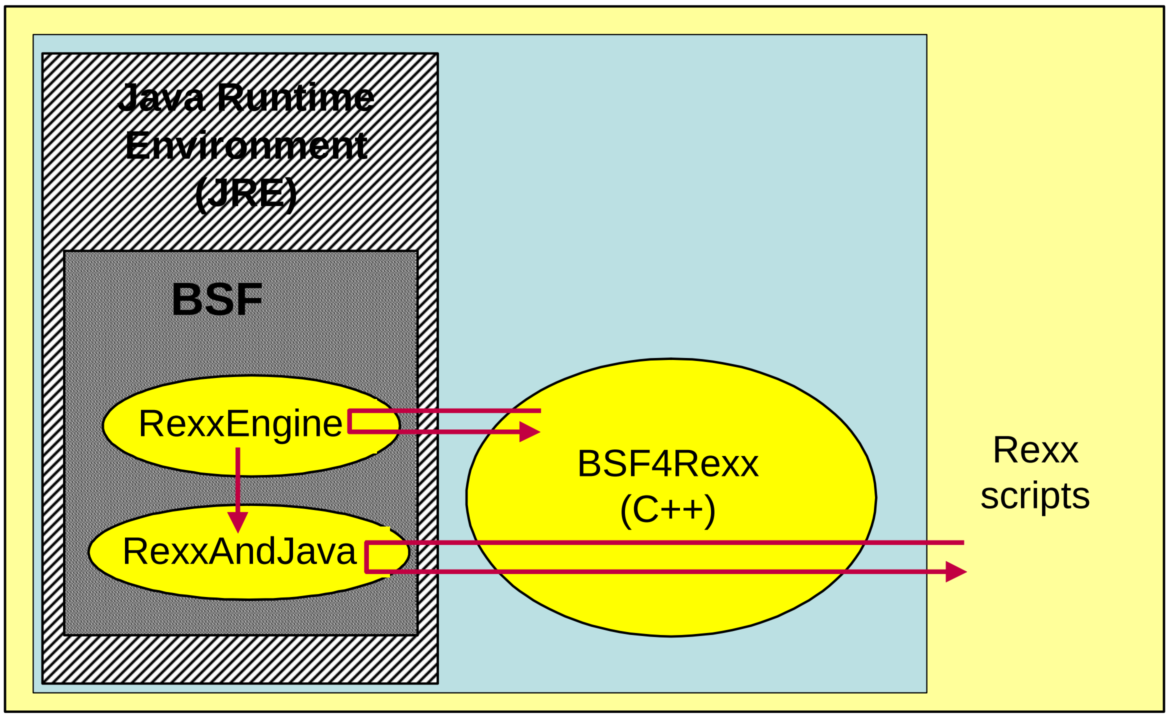


*Figure 3: The BSF Architecture and the Augsburg Version of BSF4Rexx [Fla01].*

- it makes sure, that Java programs not adhering to the BSF-specifications[8], and passing "pure" Java objects directly to Rexx, still work by registering such Java objects with the BSF registry and supplying the (string) key as an argument instead.[9]

Figure 3 depicts the Augsburg version of BSF4Rexx, stressing the point that now it has become possible for Rexx programs to load Java, if necessary, hence foregoing the necessity to invoke Rexx programs via Java programs in order to be able to call back into Java from Rexx. Nevertheless, comparing figure 3 with figure 1 it becomes clear, that the basic architecture of BSF4Rexx remained intact. And indeed, the information given in [Fla01] for the Essener version of BSF4Rexx remains appliccable to the Augsburg version!

[8] Java programs which pass Java objects to non-Java programs via BSF as arguments must register those Java objects with the BSF registry and supply the (string) key as an argument instead. Referring to such Java objects can then be achieved by supplying that very same (string) key as an argument, if calling into Java. In the case of Rexx "`RexxAndJava`" will carry out this lookup in the BSF registry at the Java side in order to get access to the indicated Java object.

[9] Technically, this is implemented in "`RexxEngine.java`".

### 2.2.1 The "BSF" External Rexx Functions

Starting with the Augsburg version of BSF4Rexx there are new Rexx external functions implemented in the dynamic link/shared library "`BSF4Rexx`"10) for allowing Rexx to demand load Java among other things. Figure 4 shows all of the available external Rexx functions and indicates which ones are preloaded by Java, if Rexx was invoked by a Java program. It also depicts which external Rexx functions get loaded, if using the loader function "`BSFLoadFuncs()`" from Rexx.

The following external Rexx functions from figure 4 accept arguments:

- `BSF( arguments... )`: the arguments depend on the "`RexxAndJava`" function addressed and are documented in [Flat01], a synopsis is given in "Addendum B" at the end of this article.

| External Function Name | Loaded by Java's RexxEngine | Loaded by BsfLoadFuncs | Synopsis |
|---|---|---|---|
| `BSF()` | yes | yes | Allows Rexx to interface with Java. |
| `BsfDropFuncs()` | - | yes | New function, deregisters all external Rexx functions. |
| `BsfInvokedBy()` | yes | yes | New function, returns '1' if Rexx was invoked by Java, '2' if Java was loaded by Rexx, '0' if no Java is present. |
| `BsfLoadFuncs()` | - | - | New function, BSF Loader function. |
| `BsfLoadJava()` | - | yes | New, loads the Java runtime environment. |
| `BsfQueryAllFunctions()` | yes | yes | New, returns a stem array with all defined external Rexx functions. |
| `BsfQueryRegisteredFunctions()` | yes | yes | New, returns a stem array with all presently registered function. |
| `BsfUnloadJava()` | - | yes | New, unloads Java. |
| `BsfVersion()` | yes | yes | New, returns the version of the DLL/shared library "`BSF4Rexx`". |

*Figure 4: External Rexx Functions Defined in the Dynamic/Shared library.*

10) The source file from which the dynamic link/shared library "`BSF4Rexx`" gets created is named "`BSF4Rexx.cc`".

- `BsfLoadJava( [add2classpath] [, {Ap|Pre}pend])`: the optional `add2classpath` argument denotes a path to be added to Java's `CLASSPATH`. If using Java 1.1 the second argument governs whether `add2classpath` should be appended (default) or prepended to the existing `CLASSPATH` environment variable.
- `BsfQueryAllFunctions([stemName])`: returns a stem array containing a list of all Rexx external functions defined in "`BSF4Rexx.cc`". Optionally, one can supply the stem name to be used (including the trailing dot!), otherwise the stem name "`BSFQUERYALLFUNCTIONS.`" is used (named after the external Rexx function).
- `BsfQueryRegisteredFunctions([stemName])`: returns a stem array containing a list of all Rexx external functions defined in "`BSF4Rexx.cc`". Optionally, one can supply the stem name to be used (including the trailing dot!), otherwise the stem name "`BSFQUERYREGISTEREDFUNCTIONS.`" is used (named after the external Rexx function).
- `BsfVersion()`: returns the version in a number formatted as: `major number *100+minor number`, decimal dot, date of this particular version in the format `YYYYMMDD`, where `YYYY` represents the four-digit year, `MM` represents the month and `DD` represents the day, followed by a space, followed by the BSF Java package name with names delimited by a slash "`/`" e.g.: "`200.20030430 com/ibm/bsf/engines/rexx`"[11].

### 2.2.2 Preregistered Java Class Objects in the BSF Registry

With the Augsburg version of BSF4Rexx some often used Java class objects are preregistered in the BSF registry. This allows Rexx programmers to refer to them merely by denoting the names in figure 5 wherever a Java class object is expected, e.g. in creating a Java array of a certain type.

Please note that for every primitive Java datatype there is also its object-oriented counterpart defined, which starts its name with a capital letter in figure 5!

[11] The Apache variant of "`BSF4Rexx.cc`" of the Augsburg version of BSF4Rexx returns as its package name "`org/apache/bsf/engines/rexx`".

| Java class object | Name in BSF-Registry | Name in ".bsf4rexx" directory |
|---|---|---|
| ***Fundamental Java class objects*** | | |
| java.lang.**Object** | Object.class | Object.class |
| java.lang.**Class** | Class.class | Class.class |
| ***Additional, useful Java class objects*** | | |
| java.lang.reflect.**Array** | Array.class | Array.class |
| java.lang.**String** | String.class | String.class |
| java.lang.**System** | System.class | System.class |
| ***Primitive Java datatypes (pseudo class objects) and their object-oriented Java class objects*** | | |
| ***boolean***.*class* | *boolean.class* | *boolean* |
| java.lang.**Boolean** | Boolean.class | Boolean.class |
| ***byte***.*class* | *byte.class* | *byte* |
| java.lang.**Byte** | Byte.class | Byte.class |
| ***char***.*class* | *char.class* | *char* |
| java.lang.**Character** | Character.class | Character.class |
| ***double***.*class* | *double.class* | *double* |
| java.lang.**Double** | Double.class | Double.class |
| ***float***.*class* | *float.class* | *float* |
| java.lang.**Float** | Float.class | Float.class |
| ***int***.*class* | *int.class* | *int* |
| java.lang.**Integer** | Integer.class | Integer.class |
| ***long***.*class* | *long.class* | *long* |
| java.lang.**Long** | Long.class | Long.class |
| ***short***.*class* | *short.class* | *short* |
| java.lang.**Short** | Short.class | Short.class |
| ***void***.*class* | *void.class* | *void* |
| java.lang.**Void** | Void.class | Void.class |

*Figure 5: Java Class Objects, Preregistered in the BSF Directory.*

The following example demonstrates how to take advantage of this service from Rexx, e.g. querying the Java System class object for its property named "`java.version`":

```
/* load the BSF4Rexx functions and start a JVM, if necessary */
if rxFuncQuery("BSF") = 1 then   /* BSF() support not loaded yet ? */
do
   call rxFuncAdd "BsfLoadFuncs", "BSF4Rexx", "BsfLoadFuncs"
   call BsfLoadFuncs   /* load all external Rexx functions from BSF4Rexx */
   call BsfLoadJava    /* load the Java runtime environment              */
end

  /* depending on your Java version "1.4.1_01" may be displayed: */
say bsf('invoke', 'System.class', 'getProperty', 'java.version')
```

### *Object Rexx*

The Object Rexx support for BSF4Rexx has been adjusted such, that for all of these Java class objects in "`BSF.cls`" proxy Object Rexx objects are created upon loading this Object Rexx support program. These proxy Object Rexx objects representing the Java class objects are stored in a directory object which is stored in the local Object Rexx environment [Flat96a] by the name "`bsf4rexx`", in effect allowing to refer to it with its environment symbol "`.bsf4rexx`" (note the preceding dot).[12] This makes referral to those preregistered Java class objects by Object Rexx easy, e.g. accessing the Java class object "`java.lang.System`"[13] and querying it for the value of the Java runtime property "`java.version`" could be expressed as:[14]

```
 /* depending on your Java version "1.4.1_01" may be displayed: */
say .bsf4rexx~System.class ~getProperty("java.version")

::requires "BSF.cls"  -- load the Object Rexx support for BSF4Rexx
```

### 2.2.3 BSF()-Subfunction "wrapEnumeration"

While testing the Augsburg version of BSF4Rexx under the different available versions of Java[15] a bug[16] with Java's reflection mechanism surfaced which prohibited the access to objects of type "`Enumeration`", if these were created with public inner classes. A possible solution[17] for this particular nasty problem available to all Java versions has been created by using a new class named "`EnumerationWrapper`"[18] which wraps such objects and makes the Enumeration

[12] Please note that the names of the primitive datatypes in the directory with "`.bsf4rexx`" do not contain the trailing characters "`.class`", hence allowing to distinguish the primitive datatypes from their object-oriented representations as Java classes.

[13] According to figure 5 the class object for "`java.lang.System`" is pre-registered in the BSF registry under the name "`System.class`".

[14] The tilde (~) is used as the message operator in Object Rexx.

[15] In the beginning of 2003 the Java versions used for testing were: 1.1, 1.2, 1.3 and 1.4.

[16] An according bug report submitted to Sun was approved by Sun on February 14, 2003 (Bug id: 4819108).

[17] A general solution for all Java versions starting with 1.2 has been created for the Augsburg version of BSF4Rexx for IBM and Apache by patching the BSF code of IBM and Apache to allow explicit access (using "`java.lang.reflect.AccessibleObject"`) in cases where objects stem from public inner classes.
However, Java 1.1 users *must* use the subfunction "`wrapEnumeration`" should they wish to access Enumeration objects stemming from public inner classes as the "`AccessibleObject`" class is not available to this older version of Java.

[18] The fully qualified name is "`com.ibm.bsf.engines.rexx.EnumerationWrapper`" for the IBM version, the Apache version starts with "`org.apache.`" instead of "`com.ibm.`".

interface available to Rexx in the case it stems from an object returned from any public class[19].

The only two methods of this class are the ones of the Java "`Enumeration`" interface:

- "`hasMoreElements()`": returns "`1`", if an element is available, "`0`" else.
- "`nextElement()`": returns the next element of the enumeration object.

### 2.2.4 Java Arrays

The Essener version of BSF4Rexx [Fla01] restricted the number of dimensions for Java arrays arbitrarily to a maximum of five. Starting with the Augsburg version, all restrictions in the context of Java arrays have been lifted. If creating Java arrays for primitive Java datatypes from Rexx use the appropriate pre-registered Java pseudo class objects[20] to indicate to Java for which datatype the Java array should be created for.[21]

#### 2.2.4.1 New BSF()-Subfunction "wrapArray"

The new `BSF()` subfunction "`wrapArray`" returns a Java object which allows to conveniently interact with the Java array object supplied. The Augsburg version of BSF4Rexx supplies a public Java class named "`ArrayWrapper`"[22], which offers public access to the fields and methods as depicted in figure 6.[23]

---

[19] The described solution would work with all BSF scripting languages.

[20] Cf. figure 5 above.

[21] Please note, that the first element of a Java array has to be indexed with the value "`0`".
The Object Rexx support makes Java arrays look like Object Rexx arrays, including the first element of an array to be indexed with the value "`1`", the methods "`items`" and the ability to enumerate all Java array entries with a "`do...over`" loop, as was the case with the Essener version already. This translation occurs transparently through the Object Rexx proxy for Java arrays. If accessing Java array objects without the Object Rexx support, then one needs to use the Java indexing, in which the first element of an array starts at (offset) "`0`".

[22] The fully qualified name is "`com.ibm.bsf.engines.rexx.ArrayWrapper`" for the IBM version, the Apache version starts with "`org.apache.`" instead of "`com.ibm.`".

[23] The methods "`supplier`" and "`makearray`" have been created to match the appropriate Object Rexx methods for Object Rexx Array classes. Cf. subsection '2.2.5 The Java Class "Supplier"' which documents the appropriate Java class allowing for the "Supplier" functionality of Object

| **Member Name** | **Synopsis** |
|---|---|
| `dimensions` | Field, stores the number of dimensions. |
| `dimension(int i)` | Method, returns the size of the given dimension (numbering starts with '0'!) |
| `items` | Field, stores the total number of entries of the array object. |
| `componentType` | Field, stores the Java class object indicating the type of the array object |
| `supplier( [0 \| 1])` | Method, returns a supplier object, allowing to enumerate all array entries, index being a String representing the needed indexing to arrive at the according entry. If no argument is given or the argument is "0", then the String representing the index part starts indexing with "0" (Java-style, index parts are enclosed in square brackets), else with "1" (Object Rexx style, index parts are separated by a comma and enclosed within rounded parenthesis). |
| `makearray({0\|1},{0\|1})` | Method, returns an array with possibly two single dimensioned array rendering of the array object. If first argument is set to "1" (`true`) then two matching array objects are returned, one containing the array entries and one the index in form of a string where the according entry is stored. If the second argument is "1" (`true`) then the index part is formatted in the Object Rexx style (see above).<br>Entries with "`null`" are ignored and therefore skipped. |

*Figure 6: Public Members of the Java Support Class "ArrayWrapper".*

The syntax is:

```
wa=BSF("wrapArray", someJavaArray)
```

***Object Rexx***

Under Object Rexx the class method "`wrapArray`" of the Object Rexx BSF4Rexx support class "BSF" can be used instead, e.g.:

```
waObject=.BSF~wrapArray( someJavaArray )
```

### 2.2.4.2 New BSF()-Subfunction "createArray"

Creating an array of a certain ("component") type can be achieved with the subfunction "`createArray`" of the `BSF()`-function. With the Augsburg version, as mentioned above, it has become possible to create arrays of arbitrary dimensions, though the programmer needs to indicate the maximum number of entries per dimension.

Rexx.

The Java type is indicated by referring to the appropriate Java class object. For this purpose you can also directly use the BSF registry names of the predefined Java class objects as depicted in figure 5 above.

The syntax is:

```
array=BSF("createArray", javaClass, max_elements1 [, max_elementsX]...)
```

The following example defines a three dimensional Java array of the primitive datatype "int":

```
array=BSF("createArray", "int.class", 3, 5, 4)
```

***Object Rexx***

Under Object Rexx the class method "`createArray`" of the Object Rexx BSF4Rexx support class "BSF" can be used instead, e.g.:

```
arrayObject=.BSF~createArray( javaClass, max_elements1 [, max_elementsX]...)
```

The latter returns an Object Rexx array proxy which behaves like an Object Rexx array.

The following example defines a three dimensional Java array of the primitive datatype "`int`", hence accepting integers only:

```
arrayObject=.BSF~createArray(.bsf4rexx~int, 3, 5, 4)
```

### 2.2.5 The Java Class "Supplier"

Borrowing from the Object Rexx class "`Supplier`"[24] an equivalent Java class has been created for the Augsburg version of BSF4Rexx, allowing to iterate over all elements of an array via the methods known from Object Rexx (cf. figure 7). In addition the Java "`Enumeration`" interface[25] is implemented.

[24] The fully qualified name is "`com.ibm.bsf.engines.rexx.Supplier`" for the IBM version, the Apache version starts with "`org.apache.`" instead of "`com.ibm.`".

[25] Cf. subsection entitled '2.2.3 New BSF()-Subfunction "`wrapEnumeration`"' above.

| **Member Name** | **Synopsis** |
|---|---|
| Methods of the Java Enumeration interface | |
| hasMoreElements | Method, returns "`1`" if element is available, "`0`" else |
| nextElement | Method, returns the next element of the Enumeration |
| Methods of the Object Rexx Supplier class | |
| available | Method, returns "`1`" if element is available, "`0`" else |
| next | Method, positions on next element |
| item | Method, returns the element |
| index | Method, returns a string representing some index |

*Figure 7: Public Members of the Java Support Class "Supplier".*

Instances of this class are returned by the method "`supplier()`" for wrapped array objects.[26)]

## 2.2.6 Need for Strong Typing Removed

As mentioned in the introduction the author removed the need for strong typing with the planned release of the Augsburg version of BSF4Rexx. This means, that the explicit supply of datatype indicators preceeding each argument meant for Java has been lifted. Instead BSF4Rexx takes the arguments and looks up the appropriate Java method on the Java side. Hence, starting with the Augsburg version it is *forbidden* to supply type information for the following subfunctions[27)] in `BSF()`-calls:

- "`arrayPut`",
- "`invoke`",
- "`registerBean`",
- "`setFieldValue`",
- "`setPropertyValue`".

26) Cf. subsection entitled '2.2.4.1 New BSF()-Subfunction "wrapArray"' above.

27) For a full list and description of all subfunctions available in `BSF()`-calls cf. [Fla01], a synopsis is given in Addendum B at the end of this article.

In the case that programmers still wish to be able to explicitly state the Java types of the arguments[28)], then they need to use the above subfunctions with the word "`strict`" appended to their names[29)].

---

[28)] There may be rare situations where it may be advisable to use strong typing in the context of invoking a Java method (subfunction "`invoke`") or constructor (subfunction "`bsfRegister`"), e.g. if a value could be interpreted to be a number or a string and that there are Java methods with a number and a string signature.

[29)] Hence the subfunction names which allow for supplying the Java datatypes in front of the individual arguments as was the default for the Essener versions need to be called: "`arrayPutStrict`", "`invokeStrict`", "`registerBeanStrict`", "`setFieldValueStrict`" and "`setPropertyValueStrict`".
Under Object Rexx all of these subfunctions, except for "`arrayPutStrict`" and "`registerBeanStrict`" are made available in the form of methods with the prefix "`BSF.`", i.e. "`bsf.invokeStrict`", "`bsf.setFieldValueStrict`", "`bsf.setPropertyValueStrict`". Should it become necessary to create an instance of a Java class with strong typing, then one would need to proceed as follows:

```
  -- create the Java object as with classic Rexx
tmp=BSF('registerBeanStrict', java_class [, "argType", "arg"]...)

  -- now turn the reference into an Object Rexx proxy object, so one
  -- is able to send it Java messages with the twiddle (~)
proxy=.bsf_proxy~new(tmp)   -- creates an Object Rexx proxy object
```

# 3 USING JAVA CLASSES FROM REXX

This section briefly introduces the reader to Java classes and their documentation, trying to explain the most important terms and concepts with respect to Java. It then demonstrates the usage of an example Java class from Rexx. In principle, all Java classes and interfaces can be used with exactly the same approach.

## 3.1 Java Classes

A Java class is the implementation of an abstract data type. An abstract data type (ADT) usually defines in general terms the properties and behaviours of data of a certain kind[30)]. Using an object-oriented language like Java it is rather easy to implement an ADT using a C/C++ like syntax.[31)]

### 3.1.1 Defining Properties and Behaviour Using a Java Class Definition

Java classes consist of fields[32)] (representing properties) and methods[33)] (implementing the behaviour). Fields and methods are called "members" of the class (definition).

If one wishes to use the infrastructure of a specific class, one needs to create an "`instance`" of that particular class. Such instances are synonymeously called "`object`"s in object-oriented terms and supply access to all fields and methods, which are declared to be "`public`"ally accessible and which are laid out in the class. Hence, the Java class serves as a blue-print for the properties and the behaviour its instances should expose.

For each instance of a Java class an individual set of fields as defined in the Java class is created, allowing the storage of values pertaining to that particular instance only. In addition all methods defined in the class are part of the instances of that

30) In the context of this article "kind" is a synonym for "type" which is a synonym for "class", hence all three terms are used interchangeably.

31) Object Rexx is another object oriented programming language which allows for an easy implementation of an ADT.

32) Sometimes the term "`attribute`" is used instead of "`field`".

33) Loosely speaking, a "`method`" is a procedure/function of an object.

class, hence all instances (objects) "behave" the same as they all share the same methods.

In the case that particular fields should be shared among all objects of a class, the keyword "`static`" needs to be supplied with the field. If there are methods which should be restricted to access static fields only, then these methods need to be decorated with the keyword "`static`" as well.[34] [35]

### 3.1.2 About Class Hierarchies

In object-oriented systems the concept of a hierarchy of classes is employed to allow programmers to reuse as much of pre-defined functionality as possible, by denoting that a particular class is to be regarded as a specialization ("subclass") of an existing class ("superclass"). Doing so allows the programmer to use all the methods of the superclass for his/her own new class, allowing to define and implement only those methods which add "specialized" behaviour, making the process of software engineering considerably easier. Being able to get access to the methods (and fields) of superclasses is often pictured with the term "inheritance". E.g. the fact that Java allows to specialize one single class at one time is dubbed "single inheritance".[36]

Java comes with an incredible rich set of pre-defined (pre-programmed and pre-tested) classes, organized in the form of a class hierarchy.[37] The root class of the Java class hierarchy is called "`Object`". The public members of the class

---

[34] Fields and methods decorated with the keyword "`static`" are also called "class fields" and "class methods". (In Object Rexx the keyword "class" needs to be used with the method directive in order to achieve the same results.)

[35] One is able to access public static fields and public static methods also by merely referring to the class itself. Or with other words, it is not necessary to create an instance of a Java class, if one wants to access public static fields and public static methods.

[36] In Object Rexx or C++ it is possible to specialize more than one superclass at the same time, which is called "multiple inheritance".

[37] The full documentation of all of Sun's Java classes in HTML (see below) is available from [W3JavaDoc].

definition of the Java class "`Object`" are therefore available to all of its direct or indirect subclasses.[38)]

### 3.1.3 More on Methods

In Java the definition of a method consists of optional decoration keywords, a return value type[39)], the name of the method, an opening and closing bracket, which *may* contain a list of arguments with explicit declaration of their types.

The name of a method together with its argument list is called "signature" and must be distinct to any other signature in the same class.[40)] The code of a Java method follows the signature and is enclosed within curly brackets.

In many object-oriented programming languages one can define methods which should get automatically invoked, if an instance of a class is to be built. This way it becomes possible to receive control at creation ("construction") time of an object, allowing e.g. for initializing each object before it gets used. Such methods are called "constructors" and in Java they are named exactly as the class itself. One may define different constructors for a Java class, each having a different signature.[41)]

### 3.1.4 Invoking a Method

In object-oriented systems one needs to "tell" the "object" which "method" it should activate. Conceptually, this is done by "sending a message to an object", which then takes on the responsibility to find a method with the same signature (same name and same argument types of the message). In Java this is realized by appending the method name with a dot to the object (the variable containing the reference to an

38) The same holds for Object Rexx, which out of the box possesses a minimal but very functional set of classes and the root of its class hierarchy is constituted by the Object Rexx class named "`Object`".

39) If a method does not return a value, then the special name "`void`" is used to indicate this fact.

40) In Object Rexx there must not be multiple methods by the same name defined in a class. Rather, one needs to use the "`arg()`"-function to determine if and what kind of arguments were supplied for the method, very much the same concept as with procedures/functions in classic (procedural) Rexx.

41) In Object Rexx a method named "`INIT`" serves as the constructor of that class, i.e. gets automatically invoked, if an instance (object) of an Object Rexx class is created.

instance of a Java class), followed by brackets, which may contain the argument values.[42)]

If such a method cannot be found in the class from which the object was created, the object conceptually then searches its superclass for a matching method up the class hierarchy until it arrives at the root class "`Object`". If a matching method cannot be found an error condition is raised.[43)]

## 3.2 Example: The Java Class "XyzType"

Figure 8 depicts the Java source code of the Java class named "`XyzType`" implementing some ADT. The members of this class are as follows:

- a `public` (constructor) method named "`XyzType`" with a signature indicating, that it gets invoked if there are no arguments supplied at creation time,
- a `public` (constructor) method named "`XyzType`" with a signature indicating, that it gets invoked if there is exactly one argument (of type String) supplied,
- a `public` static field named "`counter`" of type "`int`", i.e. a 32-Bit signed integer type, shared among all instances because of the keyword "`static`",
- a `private` field named "`info`" of type "`String`", only accessible from members of the class itself due to the keyword "`private`",
- a `public` method named "`getInfo`" with a signature indicating, that it gets invoked if there are no arguments supplied, returning a value of type "`String`",
- a `public` method named "setInfo" with a signature indicating, that it gets invoked, if there is exactly one argument of type "`String`" supplied, not

42) In Object Rexx one uses a message operator, which is dubbed "twiddle" and represented by the tilde (~) character, followed by the name of the message. Unlike Java, Object Rexx allows to omit the brackets from a message, if no arguments are supplied.

43) In a compiled language like Java the search for methods is usually carried out at compile time by the compiler. In an interpreted object-oriented language like Object Rexx, this search is carried out at runtime. If the method cannot be found, Object Rexx raises the same message as one of its conceptual ancestors, Smalltalk, namely "object cannot understand message".

```
public class XyzType    // example class for demonstrating BSF4Rexx
{
        // constructors of this class (same name as class!)
    public XyzType () {       // constructor without arguments
        counter=counter+1;    // increase counter
    }

    public XyzType (String initialValue) {   // constructor with argument
        this();               // invoke constructor above (no argument)
        info=initialValue;    // save initial value
    }
        // keyword "static": class fields (attributes) and class methods
    static public int counter=0;    // field: will count # of instances


        // instance fields (attributes) and instance methods
    private String info = null; // field: no initial value per default

    public String getInfo () {  // accessor (getter) method (function)
        return info;            // return whatever "info" points to
    }

    public void setInfo (String aValue) {   // setter method (function)
        info=aValue;            // save received value with "info"
    }
}
```

*Figure 8: The Java Class "`XyzType.java`".*

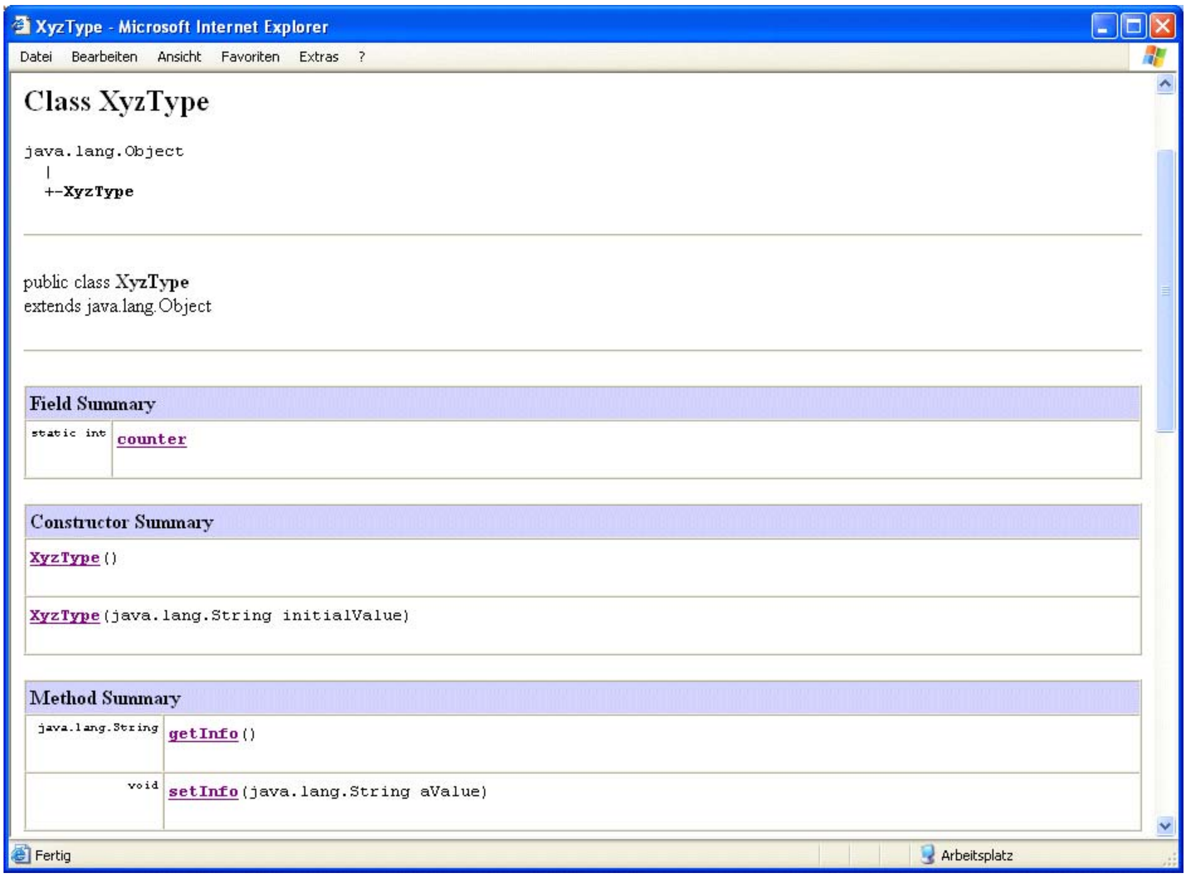
returning anything, because of the special keyword "`void`" in front of the method name.

Studying the methods the following behaviour can be deduced:

- Each time an instance is created the static field "`counter`" gets its value increased in the constructor method, and as a result representing the number of instances (objects) which got created from this particular class.
- If a string is supplied at creation time that value gets stored in the field "`info`" and can be retrieved by the public method "`getInfo()`".
- One is able to change the value of the field "`info`" by using the public method "`setInfo()`".

The Java program in figure 8 can be compiled by issuing the Java compile command "`javac XyzType.java`", resulting in the class file "`XyzType.class`", which from this moment on can be used to create instances of that type.

In the programming language Java the creation of an instance is carried out with the statement "`new XyzType()`", which returns a reference to the just created object,


XyzType - Microsoft Internet Explorer

Datei Bearbeiten Ansicht Favoriten Extras ?

Class XyzType

```
java.lang.Object
  |
  +--XyzType
```

public class XyzType
extends java.lang.Object

| Field Summary | |
|---|---|
| static int | counter |

| Constructor Summary |
|---|
| XyzType() |
| XyzType(java.lang.String initialValue) |

| Method Summary | |
|---|---|
| java.lang.String | getInfo() |
| void | setInfo(java.lang.String aValue) |

Fertig Arbeitsplatz

*Figure 9: The Java help file for the Java class "XyzType".*

after the constructors finished their work. In "BSF4Rexx" the BSF()-subfunction "registerBean" is used for this very same purpose.

A Java programmer wishing that his Java classes are documented in the Java standard, i.e. in the form of intertwined HTML-files which can be viewed and navigated by any stock WWW browser on any operating system, can do so by using the "javadoc" program which comes with the Java development kit (JDK). Creating the HTML-helpfile for the Java program in figure 8 one merely needs to issue the command "javadoc XyzType.java". The resulting helpfile containing the accessible fields and methods of the Java class "XyzType" is depicted in figure 9.[44)]

For Java programmers it is possible to add additional comments right into their programs which then get extracted by "javadoc" and stored in the Java HTML help

[44)] Please note, that by default only the public members (fields and methods) of a Java class are documented. Therefore the unaccessible private field "info" does not get documented in figure 9.

files. As a matter of fact, all of Sun's documentation about their Java classes has been created by this very same means. Hence anyone understanding the information in figure 9 will be able to study each single Java class, all of its public members (fields, methods).[45)]

### 3.2.1 Using the Java type "XyzType" from Rexx

Figure 10 depicts a Rexx program which employs "`BSF4Rexx`" in order to get access and use the Java class (type) "`XyzType`". It accesses the static field "`counter`" by referring the class "`XyzType`" directly and it creates instances of that type and demonstrating the methods "`getInfo()`" and "`setInfo()`". Here are brief comments about this Rexx program:

- Line # 1 through line # 6: if this Rexx program is invoked directly by Rexx, the external Rexx function `BSF()` is not registered yet; therefore the loader function of the external Rexx function package named "`BsfLoadFuncs()`" is registered and thereafter executed, registering all other external Rexx functions from the "`BSF4Rexx`" dynamic link/shared library. Therafter Java gets loaded in line # 5 in order to allow access to Java classes from Rexx.
- Line # 8: a variable gets the string value "`.NIL`" assigned which allows Rexx to indicate to Java the value "`null`".
- Line # 9: the name of the Java class gets stored in a variable.
- Line # 12: using the name of the Java class and the BSF subfunction "`getStaticValue`" the static field "`counter`" gets accessed and its present value is returned.
- Line # 16: the `BSF()` subfunction "`registerBean`" allows for creating an instance (object) of a Java class and returns the string value used to store

45) Usually, there is much more information supplied with the individual members, explaining the purpose and usage. Sometimes, programmers even give little examples of applying the methods of a particular class right at the top of the documentation. Also, "`javadoc`" creates navigation frames, indexes all fields and methods of all Java classes and presents the results in alphabetic order. Furthermore, it is possible to navigate individual packages full of Java classes, which are related with each other, e.g. all Java classes of the "`awt.`" package, constituting the portable, simple graphical user interface means of the "abstract window toolkit".

```
/* 1 */  if rxFuncQuery("BSF") = 1 then   /* BSF() support not loaded yet ? */
/* 2 */  do
/* 3 */     call rxFuncAdd "BsfLoadFuncs", "BSF4Rexx", "BsfLoadFuncs"
/* 4 */     call BsfLoadFuncs    /* register the BSF* external functions   */
/* 5 */     call BsfLoadJava     /* load Java, we need it!                 */
/* 6 */  end
/* 7 */
/* 8 */  null=".NIL"             /* representation for Java's "null"       */
/* 9 */  javaClass = "XyzType"   /* determine Java class to use            */
/* 10 */
/* 11 */  /* query value of static field (attribute) "counter" via class itself */
/* 12 */ say "value of static field 'counter'=" || bsf("getStaticValue", javaClass, "counter")
/* 13 */ say
/* 14 */
/* 15 */    /* creating an instance of the Java class "XyzType"   */
/* 16 */ o=BSF("registerBean", null, javaClass)    /* create an instance of "XyzType"     */
/* 17 */ say "o:" o
/* 18 */ say "# 1:" bsf("invoke", o, "getInfo")    /* get the value via the getter method */
/* 19 */    /* use the object's setter method to define a string value     */
/* 20 */ call bsf "invoke", o, "setInfo", "Hello, from Rexx..."
/* 21 */ say "# 2:" bsf("invoke", o, "getInfo")    /* get the value via the getter method */
/* 22 */    /* query value of static field (attribute) "counter"  */
/* 23 */ say "value of static field 'counter'=" || bsf("getFieldValue", o, "counter")
/* 24 */ say
/* 25 */
/* 26 */    /* release (unregister) reference to the Java object  */
/* 27 */ call BSF "unregisterBean", o              /* remove register entry from Java     */
/* 28 */
/* 29 */   /* create a second Java object      */
/* 30 */ say "creating another instance of XyzType, this time with an initial value..."
/* 31 */
/* 32 */          /* create an instance of "XyzType" and supply a string value   */
/* 33 */ o=BSF("registerBean", "otl", javaClass, "Hi, RexxLA!")
/* 34 */ say "o:" o
/* 35 */ say "# 3:" bsf("invoke", o, "getInfo")    /* get the value via the getter method */
/* 36 */
/* 37 */    /* query value of static field (attribute) "counter" via object  */
/* 38 */ say "value of static field 'counter'=" || bsf("getFieldValue", o, "counter")
```

*Figure 10: A Rexx Program Using the Java Class "XyzType".*

the just created Java object in the BSF registry.[46] No argument is passed to Java, hence the private field "`info`" in this newly created Java object will have no value which is indicated by the Java value "`null`".

- Line # 17: displays the string value under which the Java object got registered in the BSF registry and which one needs to use from Rexx if referring to it.
- Line # 18: the method "`getInfo()`" is invoked upon the Java object and returns "`.NIL`" as its value, indicating that no value is present in its private field "`info`".

[46] It is possible to force the `BSF()` subfunction "`registerBean`" to use a Rexx supplied string value as the key to be used to store the Java instance in the BSF registry, by supplying that value as an argument to "`registerBean`" instead of "`null`".

```
value of static field 'counter'=0

o: XyzType@15f5897
# 1: .NIL
# 2: Hello, from Rexx...
value of static field 'counter'=1

creating another instance of XyzType, this time with an initial value...
o: otl
# 3: Hi, RexxLA!
value of static field 'counter'=2
```

*Figure 11: A Possible Output of the Rexx Program of Figure 10.*

- Line # 20, # 21: with the help of the setter method "`setInfo()`" the string value "`Hello, from Rexx...`" will be stored in the private field "`info`" of the Java object. The Rexx statement in line # 21 uses the getter method "`getInfo()`" to retrieve the value of the private field "`info`" and to display it.
- Line # 23: this statement uses the `BSF()` subfunction "`getFieldValue`" to retrieve the value of the static field "`counter`" using the "`XyzType`" object, returning the value "`1`" as one object got created from this class so far.
- Line # 27: this statement causes the Java object in the BSF registry to be removed and makes it such inaccessible from Rexx.[47)]
- Line # 33: another instance of class "`XyzType`" gets created with the `BSF()` subfunction "`registerBean`", this time supplying an argument ("`Hi, RexxLA`!") which gets stored with the private field "`info`" in the newly created Java object. The value "`otl`" is to be used as the key for storing the newly created Java object in the BSF registry.
- Line # 35: this statement uses the Java getter method "`getInfo()`" to retrieve the value stored with the private field "`info`", displaying "`Hi, RexxLA!`" on the screen.
- Line # 38: this statement retrieves the value of the static field "`counter`" and displays the value "2" as there have been two objects created from the class so far.

47) Unregistering a Java object from the BSF registry makes it also garbage collectible by Java. Therefore, long running Rexx programs should always unregister Java objects which they do not need anymore.

Figure 11 shows a possible output generated by running the Rexx program of figure 10.

### 3.2.2 Using the Java type "XyzType" from Object Rexx

Figure 12 shows an Object Rexx program[48] using the Java class "`XyzType`" with the Object Rexx support of BSF4Rexx, which gets included in the requires directive, which loads the Object Rexx support program "`BSF.cls`".

- Line # 24: the requires directive[49] causes the Object Rexx interpreter to load and execute the Object Rexx program "`BSF.cls`", which checks whether Java is loaded and loads it if not and which sets up Object Rexx proxy classes to ease the interaction with Java, attempting to make most of Java appear to be Object Rexx.

```
/* 1 */  javaClass = "XyzType"   /* determine Java class to use            */
/* 2 */  say "value of static field 'counter'=" || .bsf~getStaticValue(javaClass, "counter")
/* 3 */  say
/* 4 */
/* 5 */  o=.BSF~new(javaClass)   /* create an instance of "XyzType"     */
/* 6 */  say "o:" o
/* 7 */  say "# 1:" o~getInfo    /* get the value via the getter method */
/* 8 */  o~setInfo("Hello, from Rexx...")
/* 9 */  say "# 2:" o~getInfo    /* get the value via the getter method */
/* 10 */ say "value of static field 'counter'=" || o~bsf.getFieldValue("counter")
/* 11 */ say
/* 12 */    /* release (unregister) reference to the Java object  */
/* 13 */  -- not necessary for Object Rexx: garbage collection will take care of this !!!
/* 14 */
/* 15 */    /* create a second Java object      */
/* 16 */ say "creating another instance of XyzType, this time with an initial value..."
/* 17 */          /* create an instance of "XyzType" and supply a string value   */
/* 18 */ o=.BSF~new(javaClass, "Hi, RexxLA!")
/* 19 */ say "o:" o
/* 20 */ say "# 3:" o~getInfo    /* get the value via the getter method */
/* 21 */
/* 22 */ say "value of static field 'counter'=" || o~bsf.getFieldValue("counter")
/* 23 */
/* 24 */ ::requires "BSF.cls"    -- get Object Rexx support
```

*Figure 12: An Object Rexx Program Using the Java Class "XyzType".*

[48] Object Rexx uses the tilde (~) character as its message operator and is called "twiddle" in the Object Rexx documentation. It also adds line comments which are led in with two dashes (`--`).

[49] Directives are led in by two colons (::), appear at the end of Object Rexx programs and will be carried out by the Object Rexx interpreter, before the Object Rexx program gets executed with its statements starting at line # 1. This way it is possible to set up or enhance the environment for the Object Rexx program before it gets executed. In this case the BSF4Rexx support for Object Rexx gets initialized and its classes are made available to the Object Rexx program.

```
value of static field 'counter'=0

o: XyzType@15f5897
# 1: The NIL object
# 2: Hello, from Rexx...
value of static field 'counter'=1

creating another instance of XyzType, this time with an initial value...
o: XyzType@f9f9d8
# 3: Hi, RexxLA!
value of static field 'counter'=2
```

*Figure 13: A Possible Output of the Object Rexx Program of Figure 12.*

- Line # 2: using a class method of the Object Rexx class "`.bsf`" it is possible to query public static Java fields, in this case the value of "`counter`" of the Java class "`XyzType`", which returns the value "`0`" as so far no instances of that class have been created.
- Line # 5: using the Object Rexx class "`.bsf`" as a wrapper for any Java class it is possible to create Java objects from Java classes as if the Java classes were Object Rexx classes. The reference to the Java objects is made available via an Object Rexx proxy object, hence it is possible to activate Java methods by sending Object Rexx messages by the desired name to the Object Rexx proxy object.
- Line # 6: this statement shows the string name used as a key to store the Java object in the BSF registry and which this Object Rexx proxy object will use to refer to it.
- Line # 7: the Java method "`getInfo()`" gets invoked via sending the "`getInfo`"[50)] message to the Object Rexx proxy object, returning the value "`.nil`", the Object Rexx counterpart to Java's "`null`" value. The "`.nil`" object renders itself to the string "`The NIL object`" if outputted to the standard out stream, as is the case with the "SAY" statement.
- Line # 8 to # 9: the Java method "`setInfo()`" with the argument "`Hello, from Rexx...`" gets invoked, storing the value with its private field "`info`". In line # 9 the getter method "`getInfo()`" is used to query the actual value of its private field "`info`", returning the just set value "`Hello, from Rexx...`".

50) In Object Rexx one may omit the parenthesis after the message name, if no arguments are supplied.

- Line # 10: the Object Rexx support for BSF4Rexx adds a set of methods to the proxy objects starting out with the string "`bsf.`" and appends it with the names of those `BSF()` subfunctions, which may make sense in the context of interacting with a Java proxy object from Object Rexx. This particular feature is employed here by using the Object Rexx proxy object instance method "`bsf.getFieldValue()`" to retrieve the actual value of the (static) field "`counter`", which now returns a value of "`1`", as one
- Line # 13: this line comment informs the reader that the Object Rexx support for BSF4Rexx will take care of unregistering the Java object from the BSF registry, if it is not needed anymore. This is realized via the Object Rexx destructor method "`UNINIT`" which gets run, once the Object Rexx garbage collector frees the proxy object, because no references from the Object Rexx program exist to it anymore.
- Line # 18 and # 20: another instance of the Java class "`XyzType`", this time supplying an initial value of "`Hi, RexxLA!".` Querying the object in line # 20 for the value of its private field via the getter method "`getInfo()`" yields this very same string: "`Hi, RexxLA!`".
- Line # 22: this is the last statement of the Object Rexx program asking for the present value of the public static field "`counter`", which yields a value of "2" as a total of two instances of the Java class "`XyzType`" have been created up to now.

Figure 13 shows the output generated by running the Object Rexx program of figure 12.

# 4 SUMMARY AND OUTLOOK

This article introduced the reader to the Augsburg version of BSF4Rexx, consisting of a Rexx engine written in Java, a Java interface program to bridge Rexx with Java, three new Java utility classes, a Rexx Java native interface library realized as a Rexx external function package[51)], and an Object Rexx support program which makes Java appear to be Object Rexx, for IBM's BSF 2.2 and Apache's BSF 2.3. The Augsburg version of BSF4Rexx builds directly on the Essener version [W3BSF4RO] and hence the reader is guided to [Flat01], which introduces and describes the architecture and the available functions and methods in more detail.

The two most important new features introduced with the Augsburg version of BSF4Rexx are the ability to invoke Java from Rexx and thereafter using all of Java with the BSF4Rexx infrastructure from Rexx and the dropping of the need of declaring the types of arguments for Java, which makes the interface simpler to use for Rexx programmers and appears to be much more "Rexx-like".

For (classic procedural) Rexx programmers an informal introduction to Java and the object oriented paradigm is attempted, which should allow them to understand the Java documentation and as a result using all functionality of the Java runtime environment (JRE) as a huge, ported set of external Rexx functions available on any platform Rexx runs on. The reader is then presented with a simple Java class and a Rexx and an Object Rexx program using that class for creating Java objects from it and interacting with such Java objects with the documented set of public accessible Java methods and Java fields.

At the time of this writing the Augsburg version was in the gamma state and can be regarded to be feature complete. It is planned to release the final version via Sourceforge sometimes in the early summer of 2003 to [W3BSF4RP].

Further discussions are delegated to and expected to take place in the Internet newsgroup <`news:comp.lang.rexx`>.

---

51) This library is available pre-compiled for Linux, OS/2 (eComStation) and Windows for the open source and free Regina Rexx interpreter, IBM's Object Rexx interpreter and a version allowing to use any Rexx interpreter which is supported by Mark Hessling's RexxTrans-library [W3RxTrans].

## 5 REFERENCES

[Ende97] Ender T.: "Object-Oriented Programming with REXX", John Wiley & Sons, New York et.al. 1997.

[Flat96a] Flatscher R.G.: "Local Environment and Scopes in Object REXX", in: Proceedings of the "7th International REXX Symposium, May 12-15, Texas/Austin 1996", The Rexx Language Association, Raleigh N.C. 1996.

[Flat96b] Flatscher R.G.: "Object Classes, Meta Classes and Method Resolution in Object REXX", in: Proceedings of the "7th International REXX Symposium, May 12-15, Texas/Austin 1996", The Rexx Language Association, Raleigh N.C. 1996.

[Flat01] Flatscher R.G.: "Java Bean Scripting with Rexx", in: Proceedings of the „12th International Rexx Symposium“, Raleigh, North Carolina, USA, April 30th - May 2nd, 2001.

[Kal01] Kalender P.: “A Concept for and an Implementation of the Bean Scripting Framework for Rexx”, Seminar paper, University of Essen, MIS and Software Engineering Department, February 2001. URL (2003-05-29): `http://nestroy.wi-inf.uni-essen.de/Lv/seminare/ws0001/PKalender/Seminararbeit.pdf`

[VeTrUr96] Veneskey G., Trosky W., Urbaniak J.: "Object Rexx by Example", Aviar, Pittsburgh 1996.

[W3Alpha]Homepage of IBM’s alphaworks projects, URL (2003-05-29): `http://www.alphaworks.ibm.com/`

[W3Ant] Homepage of the open source Apache project "ant", URL (2003-05-29): `http://ant.apache.org/`

[W3Apa] Homepage of the open source Apache organization, URL (2003-05-29): `http://www.apache.org/`

[W3BSF] Homepage of IBM’s “Bean Scripting Framework” (BSF), version 2.2, released 2001-01-31, URL (2003-05-29): `http://oss.software.ibm.com/developerworks/projects/bsf`

[W3BSF4RG] Gamma test (release candidate) site of the Augsburg version of the “BSF4Rexx” package, URL(2003-05-29): `http://wi.wu-wien.ac.at/rgf/rexx/bsf4rexx/`

[W3BSF4RO] Homepage of the original (Essener version) “BSF4Rexx” package, URL(2003-05-29): `http://nestroy.wi-inf.uni-essen.de/Forschung/rgf/Entwicklung.html`

[W3BSF4RP] Planned homepage of the Augsburg version of the “BSF4Rexx” package, URL(2003-05-29): `http://sourceforge.net/projects/bsf4rexx`

[W3Jakarta] Homepage of the open source Apache organization, URL (2003-05-29): `http://jakarta.apache.org/`

[W3Java] Java homepage, URL (2003-05-29): `http://java.sun.com/`

[W3JavaDoc] Java documentation homepage, URL (2003-05-29): `http://java.sun.com/docs`/

[W3NetRexx]NetRexx homepage of the creator of the language, the IBM fellow Mike Cowlishaw, URL (2003-05-29): `http://www2.hursley.ibm.com/netrexx/`

[W3ObjRexx]Object Rexx homepage of IBM, URL (2003-05-29): `http://www.ibm.com/software/ad/obj-rexx/`

[W3Rexx] Rexx homepage of the creator of the language, the IBM fellow Mike Cowlishaw, URL (2003-05-29): `http://www2.hursley.ibm.com/rexx/`

[W3RexxLA] Rexx homepage of the “Rexx Language Association”, URL (2003-05-29): `http://www.RexxLA.org`

[W3RxTrans]Homepage of Mark Hessling's "RexxTrans", URL (2003-05-29): `http://rexxtrans.sourceforge.net/index.html`

[W3Rhino] Rhino homepage, URL (2003-05-29): `http://www.mozilla.org/rhino`

[W3WebSphere] Homepage of IBM's WebSphere product, URL (2003-05-29): `http://www.ibm.com/software/info1/websphere/index.jsp`

[W3Xerces] Homepage of the open source Apache project "Xerces", URL (2003-05-29): `http://xml.apache.org/xerces2-j/index.html`

## ADDENDUM A: EXAMPLE PROGRAMS FROM [FLAT01] ADAPTED FOR THE AUGSBURG VERSION

In the article about the Essener version of BSF4Rexx [Flat01] there was a little Rexx and Object Rexx program depicted, which demonstrated the usage of the Java "awt" (abstract window toolkit) classes to create and use a platform independent graphical user interface. Due to the Essener version all arguments were strongly typed.

In this section the very same programs are shown with the type information removed, so they can be run under the Augsburg version of BSF4Rexx. In addition the frame window is changed to a larger size in order to show the full text of its title and is shown in figure 16. The upper screenshot stems from Windows XP running the programs of figure 14 and 15 under Object Rexx, the bottom screenshot was taken from a Linux Red Hat 7.3 system running the open source Rexx interpreter

```
/* "ShowCount.rex" - a Rexx program to count number of button presses   */
call BSF 'registerBean',     'win', 'java.awt.Frame', 'Show count'
call BSF 'addEventListener', 'win', 'window', 'windowClosing', 'call BSF "exit"'

call BSF 'registerBean',     'but', 'java.awt.Button', 'Press me!'
call BSF 'addEventListener', 'but', 'action', '', 'call ShowSize'

call BSF 'registerBean',     'lab', 'java.awt.Label'
call BSF 'invoke',           'lab', 'setAlignment', 1

call BSF 'invoke', 'win', 'add', 'Center', 'lab'
call BSF 'invoke', 'win', 'add', 'South',  'but'
call BSF 'invoke', 'win', 'pack'
call BSF 'invoke', 'win', 'setSize', 300, 90
call BSF 'invoke', 'win', 'show'
call BSF 'invoke', 'win', 'toFront'

i=0             /* set counter to 0     */

do forever
   a = bsf("pollEventText")      /* wait for an eventText to be sent      */
   interpret a                   /* execute as a Rexx program     */
   if result= "SHUTDOWN, REXX !" then leave /* JVM will be shutdown in 0.1sec */
end

exit

        /* show the actual number of times, you pressed the button       */
ShowSize:
   i=i+1
   call BSF 'invoke', 'lab', 'setText', "Press #" i
   return
```

*Figure 14: A Rexx Program which Uses Java's AWT for a GUI Interface, Adaptation (removing type information from the arguments) of Figure 20 in [Flat01].*

```
/* "ShowCount.rex" - an Object Rexx program to count number of button presses  */
.bsf~import("awtFrame",  "java.awt.Frame")
.bsf~import("awtButton", "java.awt.Button")
.bsf~import("awtLabel",  "java.awt.Label")

win=.awtFrame~new("Show Count")
win~bsf.addEventListener('window', 'windowClosing', '.bsf~exit')

but=.awtButton~new("Press me!")
but~bsf.addEventListener('action', '', 'call ShowSize')

lab=.awtLabel~new ~~setAlignment(1)

win ~~add("Center", lab) ~~add("South", but) ~~pack ~~setSize(300,90) ~~show ~~toFront

i=0             /* set counter to 0     */

do forever
   a = bsf("pollEventText")     /* wait for an eventText to be sent     */
   interpret a                  /* execute as a Rexx program    */
   if result= "SHUTDOWN, REXX !" then leave /* JVM will be shutdown in 0.1sec */
end

exit

        /* show the actual number of times, you pressed the button      */
ShowSize:
   i=i+1
   lab~setText("Press #" i)
   return

::requires "BSF.cls"    -- get access to the Object Rexx support enhancement
```

*Figure 15: An Object Rexx Program which Uses Java's AWT for a GUI Interface, Adaptation (removing type information from the arguments) of Figure 23 in [Flat01].*

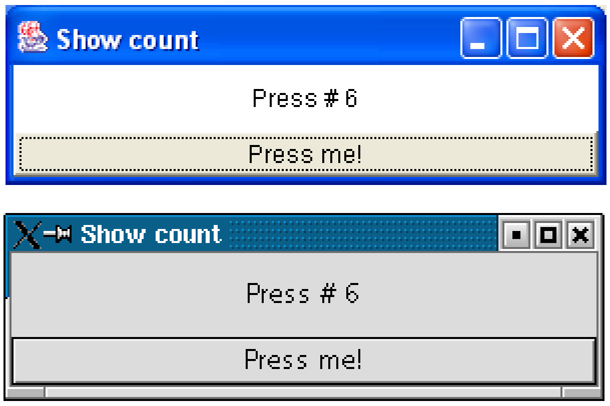


*Figure 16: Platform Independent GUI Created with Java's AWT by the Rexx Program of Figure 14 and the Object Rexx Program of Figure 15 running under Windows XP and Linux Red Hat 7.3.*

Regina to execute the Rexx code of figure 14 and the free evaluation copy of Object Rexx for Linux was used to run the Object Rexx code of figure 15.

## ADDENDUM B: BRIEF OVERVIEW OF THE BSF()-SUBFUNCTIONS

The external Rexx function "BSF()" allows Rexx programmers to call into Java, where with the help of a Java program ("RexxAndJava.java") the desired subfunction gets carried out. The call syntax from Rexx looks like:

```
    call BSF "SubFunction" [, "argument1"]...
```

or:

```
    a=BSF("SubFunction" [, "argument"]...)
```

The table in figure 17 lists and briefly describes all 27 "BSF()" subfunctions, which are implemented in "RexxAndJava.java" and documented with "javadoc". Hence, one could look up the HTML-help for "RexxAndJava" and study the documentation of the method "javaCallBSF" to learn about these subfunctions as well.

The subfunctions which deal directly with a Java class ("registerBean" and "registerBeanStrict" to create an instance from it, "getStaticField" to get the value of a public static field of a Java class) expect a string denoting the fully qualified name of the Java class.

All subfunctions interacting with a Java object need to use the string (key) used to store that Java object in the BSF registry on the Java side to uniquely address the Java object, e.g. subfunctions: "invoke", "getFieldValue", "setFieldValue".

All subfunctions containing the string "Strict" need their arguments to be "strongly typed", i.e. the Rexx programmer needs to indicate before each argument of which Java type it is. These strings denominating the Java types are depicted in figure 18 and are sometimes called "argTypes" or "typeIndicator".

| SubFunction | Brief description |
|---|---|
| `"addEventListener", beanName, eventSetName, filter, eventText` | Allows to add an event listener and tell it what event string to send to the Rexx program. This could be Rexx code to be interpreted upon receipt. |
| `"arrayAt", arrayObject, index1 [, indexn]...`<br>`"arrayAt", arrayObject, intArray` | Returns the object stored in the `arrayObject` at the given index/indices. |

Optionally the given index/indices may be supplied with an `intArray`.

`"arrayLength", arrayObject`

Returns the capacity of this particular `arrayObject`.

`"arrayPut", arrayObject, newValue, index1 [, indexn]...`
`"arrayPut", arrayObject, newValue, intArray`

Stores `newValue` in the `arrayObject` at the given index/indices. Optionally the given index/indices may be supplied with an `intArray`.

`"arrayPutStrict", arrayObject, typeIndicator, newValue, index1 [, indexn]...`
`"arrayPutStrict", arrayObject, typeIndicator, newValue, intArray`

Stores `newValue` in the `arrayObject` at the given index/indices. Optionally the given index/indices may be supplied with an `intArray`. (Deprecated.)

`"createArray", componentType, capacity1 [, capacityn]...`
`"createArray", componentType, intArray`

Creates a Java array of the given `componentType` (a Java class object), determining the `capacity` in each dimension. Alternatively, an `intArray` can be given which is used to store the capacity of each dimension.

`"exit"[, [retVal] [, time2wait]]`

Terminates the Java virtual machine with a return code of `retVal` after `time2wait` milliseconds.

`"getFieldValue", javaObject, fieldName`

Looks up and returns the value of `fieldName` in the given `javaObject`.

`"getPropertyValue", javaObject, propertyName, index`

Looks up and returns the value of `propertyName` in the given `javaObject` at the given `index` (set to `null`, if not an indexed JavaBean property).

`"getStaticValue", className, fieldName`

Looks up and returns the value of the public static `fieldName` in the given Java class of `className` (can be a Java `interface` as well).

`"invoke", javaObject, method [, arg1] ...`

Invokes the `method` on the `javaObject` supplying the arguments, if any. Returns whatever the method returns or `null`.

`"invokeStrict", javaObject, method [, typeIndicator1, arg1] ...`

Invokes the `method` on the `javaObject` supplying the type of each argument before the argument, if any. Returns whatever the method returns or `null`.

`"lookupBean", beanName`

Returns `beanName`, if there is a Java object registered in the BSF registry under the name `beanName`, `null` else.

`"pollEventText" [, timeout]`

Returns the eventText, if available, else waits `timeout` milliseconds. If no timeout is given, this subfunctions waits until an eventText becomes available.

`"postEventText", eventText [, priority]`

Posts the `eventText` at the given `priority` (1=highest, 2=default or 3=lowest).

`"registerBean", [beanName], className [, arg1]...`

Creates an instance of the Java class named `className`, supplying arguments if

| | |
|---|---|
| | available. The Java object will get stored in the BSF registry under `beanName` (creating a unique name, if `beanName` was omitted by Rexx), which gets returned. |
| `"registerBeanStrict", [beanName], className [, typeIndicator1, arg1]...` | |
| | Creates an instance of the Java class named `className`, supplying arguments if available. The Java object will get stored in the BSF registry under `beanName` (creating a unique name, if `beanName` was omitted by Rexx), which gets returned. If arguments are supplied, each will be preceded by its type. |
| `"setFieldValue", javaObject, fieldName, newValue` | |
| | Looks up and sets the value of `fieldName` in the given `javaObject` to `newValue`. |
| `"setFieldValueStrict", javaObject, fieldName, typeIndicator, newValue` | |
| | Looks up and sets the value of `fieldName` in the given `javaObject` using the type of the `typeIndicator` for the `newValue`. (Deprecated.) |
| `"setPropertyValue", javaObject, propertyName, index, newValue` | |
| | Looks up and sets the value of `propertyName` at the given `index` (`null`, if not an indexed JavaBean property) in the given `javaObject` to `newValue`. |
| `"setPropertyValue", javaObject, propertyName, index, typeIndicator, newValue` | |
| | Looks up and sets the value of `propertyName` at the given `index` (`null`, if not an indexed JavaBean property) in the given `javaObject` using the type of the `typeIndicator` for the `newValue`. |
| `"setRexxNullString", newString` | |
| | Sets the textual representation of a Java `null` value to `newString`. This allows Rexx to determine and to indicate the Java `null` value and is preset to "`.NIL`". |
| `"sleep", time2sleep` | |
| | Sleeps `time2sleep` seconds (can be a fraction as well) before returning to Rexx. |
| `"unregisterBean", beanName` | |
| | Removes the Java object registered with `beanName` from the BSF registry, making it unavailable to Rexx. This allows the Java object to be garbage collected from Java. |
| `"wrapArray", arrayObject` | |
| | Wraps the `arrayObject` which then can be accessed and analyzed via the methods of the Java class "`ArrayWrapper`". |
| `"wrapEnumeration", enumObject` | |
| | Wraps an enumerable Java object which one wishes to enumerate. (This is meant for programs running under Java 1.1 *only* to overcome an access violation bug. Starting with Java 1.2 this Java reflection bug is circumvented by the Augsburg version of BSF4Rexx taking advantage of the Java class "`AccessibleObject`".) |
| `"version"` | |
| | Returns the version string of the program "`RexxAndJava`", e.g. `"200.20030416 com.ibm.bsf.engines.rexx"` or `"200.20030416 org.apache.bsf.engines.rexx"`. |

Figure 17: The "`BSF()`" Subfunctions of the Augsburg Version of BSF4Rexx.

| Indicator | Datatype |
|---|---|
| `"`**`bo`**`olean"` | the value `0` (`false`) or `1` (`true`) |
| `"`**`by`**`te"` | a byte value |
| `"`**`c`**`har"` | a single (UTF8) character |
| `"`**`d`**`ouble"` | a double value |
| `"`**`f`**`loat"` | a float value |
| `"`**`i`**`nt"` | an integer value |
| `"`**`l`**`ong"` | a long value |
| `"object"` | a Java object which is registered with the BSF registry (the immediately following argument is the string serving as the key for retrieving the desired Java object from the BSF registry). |
| `"`**`sh`**`ort"` | a short value |
| `"`**`st`**`ring"` | a string value (UTF8) |

*Figure 18: The Java Type Indicator Strings for the "Strict"-Subfunctions of figure 17 above, which must precede each individual argument. Only the bold  letters need to be given.*

Date of Article: 2003-06-01.

Published in: Proceedings of the „2003 International Rexx Symposium“, Triangle Research Park, North Carolina, USA, May 4th - May 7th, 2003, The Rexx Language Association, Raleigh N.C. 2003.

Presented at: „2003 International Rexx Symposium“, Raleigh, North Carolina, USA, May 5th, 2003.