jsr223: A Java Platform Integration for R with Programming Languages Groovy, JavaScript, JRuby, Jython, and Kotlin

Floid R. Gilbert & David B. Dahl

Abstract

The R package **jsr223** is a high-level integration for the Java platform. It makes Java objects easy to use from within R; it provides a unified interface to integrate R with several programming languages; and it features extensive data exchange between R and Java. In all, **jsr223** significantly extends the computing capabilities of the R software environment.

Contents

1 Introduction

About the same time Ross Ihaka and Robert Gentleman began developing R at the University of Auckland in the early 1990s, James Gosling and the so-called Green Project Team was working on a new programming language at Sun Microsystems in California. The Green Team didn't set out to make a new language; rather, they were trying to move platform-independent, distributed computing into the consumer electronics marketplace. As Gosling explained, "All along, the language was a tool, not the end" (O'Connell, 1995). Unexpectedly, the programming language outlived the Green Project and flourished into one of the most popular development platforms in computing history. That platform, Java, now powers applications ranging from the enterprise (GMail), to games (Minecraft), to interactive media (Blu-ray), to mobile devices (Android).

In 2003, Simon Urbanek released **rJava**, an integration package designed to avail R of the burgeoning development surrounding Java. The package has been very successful to this end. Today, it is one of the top-ranked solutions for R as measured by monthly downloads.[1] **rJava** is described by Urbanek as a low-level R to Java interface analogous to `.C` and `.Call`, the built-in R functions for calling compiled C code. Like R's integration for C, **rJava** loads compiled code into an R process's memory space where it can be accessed via various R functions. Urbanek achieves this feat using the Java Native Interface (JNI), a standard framework that enables native (i.e. platform-dependent) code to access and use compiled Java code. The **rJava** API requires users to specify classes and data types in JNI syntax. One advantage to this approach is that it gives the user granular, direct access to Java classes. However, as with any low-level interface, the learning curve is relatively high and implementation requires verbose coding. Another advantage of using JNI is that it avoids the difficult task of dynamically interpreting or compiling source code. Of course, this is also a disadvantage: it limits **rJava** to using compiled code as opposed to embedding source code directly within R script.

Our **jsr223** package builds on **rJava** to make Java libraries easier to use from within R. We achieve this by embedding other programming languages in R that can, in turn, use Java objects in natural syntax. As we show in the **rJava** software review, this approach is generally simpler and more intuitive than **rJava**'s low-level JNI interface. To date, **jsr223** supports embedding five programming languages that target the Java platform: Groovy, JavaScript, JRuby, Jython, and Kotlin. (JRuby and Jython are Java implementations of the Ruby and Python languages, respectively.) Besides providing simplified access to Java libraries, **jsr223** also makes countless existing solutions in these supported programming languages immediately available to R developers. For example, Ruby gems can be embedded in R and popular JavaScript libraries can be sourced directly from the Internet.

The **jsr223** package features extensive, configurable data exchange between R and Java. R vectors, factors, n-dimensional arrays, data frames, lists, and environments are converted to standard Java objects. Java scalars, n-dimensional arrays, maps, and collections are inspected for content and converted to the most appropriate R structure (i.e., vectors, n-dimensional arrays, data frames, or lists). Several data exchange options are available, including row-major and column-major ordering schemes for data frames and n-dimensional arrays. Many language integrations for R provide a comparable feature set by using JSON (JavaScript Object Notation) libraries. In contrast, the **jsr223** package implements data exchange using custom Java routines to avoid the serialization overhead and the loss of floating point precision inherent in JSON data conversion.

1 The **rJava** package ranks in the 95th percentile for R package downloads according to http://rdocumentation.org.

Other distinguishing features of the **jsr223** package include a simple callback interface to access the R environment from embedded scripts, script compiling, and string interpolation. In all, **jsr223** lowers the barrier to using sophisticated Java solutions from within R.

Package script engine installation instructions and complete feature documentation are available in the **jsr223** package vignette. All code examples related to this document are available on GitHub at `https://github.com/floidgilbert/jsr223/tree/master/examples`.

1.1 The jsr223 package implementation

The **jsr223** package is built on **rJava**. The design of **jsr223** follows cues from **rJava**, **rscala** (Dahl, 2016), and **V8** (Ooms, 2016).

The **jsr223** package uses the Java Scripting API (Oracle, 2016a) as defined by the specification "JSR-223: Scripting for the Java Platform" (Sun Microsystems, Inc., 2006). The JSR-223 specification includes two crucial elements: an interface for Java applications to execute code written in scripting languages, and a guide for scripting languages to create Java objects in their own syntax. Hence, the JSR-223 specification is the basis for our integration.

Many programming language projects have implemented JSR-223. Our **jsr223** package officially supports most of them. Technically, any JSR-223 implementation will work with **jsr223**, but we may not officially support some languages for various reasons. The most notable exclusion is Scala; we don't support it simply because the JSR-223 implementation is incomplete. (For a very thorough Scala integration for R, consider the **rscala** package.) We also exclude languages that are no longer being actively developed, like BeanShell.

Data exchange for **jsr223** is provided by **jdx**: Java Data Exchange for R and **rJava**. The **jdx** package's functionality was originally part of **jsr223**, but we broke it out into a separate package to simplify maintenance and to make its features available to other developers. For more information, see Data exchange details.

Callbacks allow embedded scripts to call back into the same R session. In **jsr223**, callbacks are implemented via multi-threading and a custom messaging protocol. This implementation is lightweight, does not require any special configuration, and supports infinite recursion between R and the script engine (limited only by stack space).

1.2 Document organization

This is how it's going to go...

2 Helpful terminology and concepts

Java programs are compiled to Java bytecode that can be executed by an instance of a Java Virtual Machine (JVM). A JVM is an abstraction layer that provides a platform-independent execution environment for Java programs. A JVM interprets Java bytecode to machine code (i.e., processor-specific instructions). JVMs are available for a wide variety of hardware and software platforms. In principle, the same Java program will run on any platform that supports a JVM. The Java paradigm contrasts with traditional compiled languages, such as C, that are compiled directly to processor-dependent machine code, and therefore must be recompiled for every targeted architecture. Often, changes in the source code are also required to support different platforms.

Today, there are several programming languages that compile down to Java bytecode including all of the languages currently supported by **jsr223**. This may be surprising to some readers because languages like JavaScript are traditionally interpreted only, not compiled. In fact, the **jsr223** languages blur the line between scripting languages (those that are interpret-only) and traditional compiled languages. Nevertheless, we generally refer to the languages supported by **jsr223** as scripting languages in this document because, as far as the user is aware, source code is interpreted and executed (i.e., evaluated) in one step. Even so, this implementation benefits from the significant performance gains of compiled code.

A *scripting engine* (usually shortened to *script engine*) is software that enables a scripting language to be embedded in an application. Internally, a script engine uses an *interpreter* to parse and execute source code. The terms *script engine* and *interpreter* are often used interchangeably. In this document, *script engine* refers to the software component, not the interpreter. A *script engine instance* denotes an instantiated script engine. Finally, a *script engine environment* refers to the state (i.e., the variables and settings) of a given instance.

Bindings refers to the name/value pairs associated with variables in a given scope. Conceptually, a variable's name is bound to its value. The variable names and values in R's global environment are examples of bindings.

3 Typical use cases

This section includes introductory examples that demonstrate the core functionality of the **jsr223** package. For a complete overview of **jsr223** features, see the Feature documentation.

3.1 Using Java libraries

This example shows how so-called "glue" code can be embedded in R to quickly leverage Java solutions. It also demonstrates how easily **jsr223** converts Java data structures to R objects.

Specifically, we show how to use Stanford's Natural Language Processing (NLP) libraries to identify grammatical parts of speech in a text. NLP is a key component in statistical text analysis and artificial intelligence. The NLP Java libraries required for this example can be obtained from the Stanford CoreNLP.

The first step: create a **jsr223** `ScriptEngine` object that can dynamically execute source code. In this case, we use a JavaScript engine. The object is created using the `ScriptEngine$new` constructor method. The method takes two arguments: the scripting language name and a character vector containing paths to the required Java libraries. In the code below, the `class.path` variable contains the Java library paths. The new `ScriptEngine` object instance is assigned to the variable `engine`.

```
class.path <- c(
  "./protobuf.jar",
  "./stanford-corenlp-3.9.0.jar",
  "./stanford-corenlp-3.9.0-models.jar"
)

library("jsr223")
engine <- ScriptEngine$new("JavaScript", class.path)
```

Now we can execute JavaScript source code. The **jsr223** interface provides several methods to execute code. In this example, we use the `%@%` operator which executes a code snippet and

discards the return value, if any, instead of returning it to R. This code snippet, supplied as a character constant, imports the Stanford NLP `Document` class. This syntax is peculiar to the JavaScript language integration with Java. The resulting object, `DocumentClass` is a `///`. It can be used to instantiate objects or access static methods.

```
engine %@% 'var DocumentClass = Java.type("edu.stanford.nlp.simple.Document");'
```

and defines a JavaScript function named `getPartsOfSpeech`. The function tags each element in a text with the corresponding grammatical part of speech. ///describe the return value being built.

```
engine %@% '
  function getPartsOfSpeech(text) {
    var doc = new DocumentClass(text);
    var list = [];
    for (i = 0; i < doc.sentences().size(); i++) {
      var sentence = doc.sentences().get(i);
      var o = {
        "words":sentence.words(),
        "pos.tag":sentence.posTags(),
        "offset.begin":sentence.characterOffsetBegin(),
        "offset.end":sentence.characterOffsetEnd()
      }
      list.push(o);
    }
    return list;
  }
'
```

Finally, we use `engine$invokeFunction` to call `getPartsOfSpeech` from R. The `invokeFunction` method takes the name of the function as the first parameter and any number of function parameters. The **jsr223** package automatically converts the function's return value to R objects. In this case, it intuitively converts the list of JavaScript objects to a list of R data frames. The output is printed below. The parts of speech abbreviations are defined by the Penn Treebank Project.

```
engine$invokeFunction(
  "getPartsOfSpeech",
  "The jsr223 package makes Java objects easy to use. Download it from CRAN."
)
```

```
## [[1]]
##      words pos.tag offset.begin offset.end
## 1      The      DT            0          3
## 2   jsr223      NN            4         10
## 3  package      NN           11         18
## 4    makes     VBZ           19         24
## 5     Java     NNP           25         29
## 6  objects     NNS           30         37
## 7     easy      JJ           38         42
```

```
## 8       to      TO          43         45
## 9      use      VB          46         49
## 10       .       .          49         50
##
## [[2]]
##      words pos.tag offset.begin offset.end
## 1 Download      VB          51         59
## 2       it     PRP          60         62
## 3     from      IN          63         67
## 4     CRAN     NNP          68         72
## 5        .       .          72         73
```

3.2 Extending Java classes

///it is possible to use Groovy closures or anonymous classes to do the same thing with less code (no need to extend a class...pass the method as a function to sample, but it's twice as slow in our tests). so far, formally extending an anonymous class is the fastest under these circumstances. With four threads using Intel i7-5500U @

This example demonstrates using Java libraries from R and converting Java data structures to R objects. Specifically, we show how to use Stanford's Natural Language Processing (NLP) libraries to identify grammatical parts of speech in a text. NLP is a key component in statistical text analysis and artificial intelligence. We create and use the objects in the Groovy language. Groovy syntax is very similar to the Java programming language; hence, it is a natural choice for Java integrations.

The NLP Java libraries required for this example can be obtained from the Stanford CoreNLP. The Groovy script engine file can be obtained by following the instructions under Script engine installation and instantiation in this document.

The first step: create a **jsr223** `ScriptEngine` object that can dynamically execute source code. We create the object by using the `ScriptEngine$new` constructor method. The method takes two arguments: the scripting language name and a character vector containing paths to the required Java libraries. In the code below, the `class.path` variable contains the Java library paths. The new `ScriptEngine` object instance is assigned to the variable `engine`.

Mention performance and how all data was transformed back in less than a second.

Metropolis sampler

This example uses a generic java class with dynamic bits provided by javascript. maybe other languages.

Markov chain Monte Carlo ///implementing interfaces ///returning data fast ///performance

3.3 Using other solutions

In addition to using Java libraries, **jsr223** can easily take advantage of solutions written in other languages. In some cases, integration is as simple as sourcing a script file. For example, many of the ///

```
engine$source(
  "https://raw.githubusercontent.com/panzerdp/voca/master/dist/voca.min.js",
  discard.return.value = TRUE
)
```

Table 1: Results of MCMC. Quantiles, acceptance ratio, and effective sample size (ESS).

Parm	Chain	2.5%	25%	50%	75%	97.5%	Acc. Ratio	ESS
π	1	0.294	0.463	0.543	0.611	0.714	0.235	1006
π	2	0.292	0.461	0.536	0.607	0.715	0.229	805
π	3	0.294	0.459	0.534	0.603	0.710	0.232	929
π	4	0.311	0.458	0.532	0.598	0.714	0.242	987
λ	1	0.955	1.331	1.571	1.843	2.422	0.264	1007
λ	2	0.937	1.340	1.563	1.820	2.409	0.265	1068
λ	3	0.939	1.321	1.560	1.808	2.395	0.264	1119
λ	4	0.953	1.325	1.546	1.823	2.359	0.268	1187

4 Installation

4.1 Package installation

The **jsr223** package requires Java 8 Standard Edition or above. The current version of the Java Runtime Environment (JRE) can be determined by executing '`java -version`' from a system command prompt. See the example output below. Java 8 is denoted by version 1.8.x_xx.

```
java version "1.8.0_144"
Java(TM) SE Runtime Environment (build 1.8.0_144-b01)
Java HotSpot(TM) 64-Bit Server VM (build 25.144-b01, mixed mode)
```

The JRE can be obtained from Oracle's web site. Select the architecture (32 or 64 bit) that matches your R installation.

jsr223 runs on a standard installation of R (e.g., the R build option `--enable-R-shlib` is not required). **jsr223** is available on CRAN and can be installed with the usual command:

```
install.packages("jsr223")
```

This command will also download and install the **rJava** dependency necessary. However, the **rJava** installation will fail if R is not yet configured to use Java on Unix, Linux, or OSX. To configure R for Java, execute '`sudo R CMD javareconf`' in a terminal. This command is not required for Windows systems. If the Java reconfiguration command generates errors, address the errors and execute the command again. One common error can be resolved by determining whether the GNU Compiler Collection (GCC) is accessible. To check for GCC, execute 'gcc –help' from a terminal. This command will fail if GCC is not installed or if the license agreement has not been accepted.

4.2 Script engine installation and instantiation

To create an instance of a language's script engine, **jsr223** requires access to the associated Java Archive (JAR) files. These instructions will help you obtain the required files and create a script engine instance.

4.2.1 *Groovy*

Groovy is a Java-like scripting language. Java code can often be executed by the Groovy engine with little modification. Hence, this Groovy integration essentially brings the Java language to R.

To obtain the standalone Groovy engine, go to http://groovy-lang.org and click the 'Download' link. Locate the current binary distribution. Download and extract the archive to a temporary folder. Locate the 'embeddable' subfolder. Copy the file named 'groovy-all-x.x.x.jar' to a convenient location and make note of the path. Specify this path in the class.path parameter of the ScriptEngine$new constructor to create a Groovy script engine instance:

```
library("jsr223")
engine <- ScriptEngine$new("groovy", class.path = "~/your-path/groovy-all.jar")
```

4.2.2 *JavaScript (Nashorn)*

Nashorn is the JavaScript dialect included in Java 8. Nashorn implements ECMAScript 5.1. Because Nashorn is included in the JRE, no downloads are necessary to use JavaScript with **jsr223**. Technical documentation and examples for Nashorn are available at Oracle's web site. Create a JavaScript instance using

```
library("jsr223")
engine <- ScriptEngine$new("javascript")
```

4.2.3 *JRuby*

JRuby is a Java-based implementation of the Ruby programming language. Obtain the standalone JRuby engine by clicking the 'Downloads' link at at http://jruby.org. Find 'JRuby x.x.x.x Complete.jar' and save it to a convenient location. Specify the path to the JAR file in the class.path parameter of the ScriptEngine$new constructor to create a JRuby script engine instance.

```
library("jsr223")
engine <- ScriptEngine$new("ruby", class.path = "~/your-path/jruby-complete.jar")
```

4.2.4 *Jython*

Jython is a Java-based implementation of the Python programming language. The standalone Jython engine is available at http://www.jython.org. Follow the 'Download' link. Click 'Download Jython x.x.x - Standalone Jar' to start the download. Save the JAR file to a convenient location and remember the path. This path will be used by **jsr223** to load the script engine as in the following code.

```
library("jsr223")
engine <- ScriptEngine$new("python", class.path = "~/your-path/jython-standalone.jar")
```

4.2.5 *Kotlin*

Kotlin is a relatively new programming language that is interoperable with Java. As of this writing, a standalone JAR file is not available for the script engine. The most straight-forward way to obtain the files is to use selected files from the Community Edition of the JetBrains IntelliJ Idea integrated development environment (IDE). The IDE doesn't need to be installed. Download the IDE's archive file (e.g., a zip file, not the executable installer package). Create an empty target folder on your system for the Kotlin files. Extract the 'bin' and 'plugins/Kotlin' folders to the target folder preserving the original folder structures. **Note:** The 'bin' folder isn't strictly required, but it will eliminate warnings on some systems. Make note of the

fully-qualified path to the 'plugins/Kotlin' folder; it will be used by **jsr223** to load the script engine.

If you are already using a current version of IntelliJ Idea, or if you decide to install the IDE, locate the path to the 'plugins/Kotlin' subfolder of the IDE's installation path. This folder will be used to load the script engine.

Because Kotlin does not provide a standalone script engine JAR file, **jsr223** includes a convenience function `getKotlinScriptEngineJars` to simplify adding JAR files to the class path. The following code demonstrates creating a Kotlin script engine instance using only the minimum required JAR files. The `kotlin.path` variable contains the path to the 'plugins/Kotlin' folder on your system.

```
library("jsr223")
engine <- ScriptEngine$new(
  "kotlin",
  class.path = getKotlinScriptEngineJars(kotlin.path)
)
```

To include all Kotlin system JAR files in the class path, use this example instead.

```
library("jsr223")
engine <- ScriptEngine$new(
  "kotlin",
  class.path = getKotlinScriptEngineJars(kotlin.path, minimum = FALSE)
)
```

5 Feature documentation

The primary features of **jsr223** are designed to be accessible to R programmers of all experience levels. This quick start guide illustrates these features with simple code examples. In general, the code samples work with all supported script engines with two exceptions.

1. Global variables in Ruby script must be prefixed with a dollar sign.
2. Kotlin script engine bindings are not created as global variables. See Kotlin idiosyncrasies.

5.1 Hello world

The R code snippet below demonstrates the basic elements required to embed a scripting language: start a script engine, optionally pass data to the script engine environment, execute a script, and terminate the script engine when it is no longer needed.

```
library("jsr223")
engine <- ScriptEngine$new("javascript")
engine$message <- "Hello world"
engine %~% "print(message);"

## Hello world

engine$terminate()
```

The `ScriptEngine$new` constructor method creates a script engine instance. In the preceding example, we assign the new script engine object to the variable `engine`. The first argument of `ScriptEngine$new` specifies the type of script engine to create. In this case, we create a JavaScript engine. The third line assigns the value `"Hello world"` to a global variable named `message` in the script engine environment. The next line executes a JavaScript code snippet using the `%~%` operator. The snippet uses the JavaScript `print` method to write the message to the console. The last line in the example terminates the script engine and releases the associated resources.

To create a script engine other than JavaScript, specify a different script engine name and a character vector containing the required script engine JAR files. (See Script engine installation for instructions to obtain script engines.) The supported script engine names are listed in Table 2. These names are defined by the script engine provider. **Note:** Script engine names are case sensitive.

The next example reproduces the "Hello world" example in Ruby script.

```
library("jsr223")
engine <- ScriptEngine$new(
  engine.name = "ruby",
  class.path = "~/your-path/jruby-complete.jar"
)
engine$message <- "Hello world"
engine %~% "puts $message"

## Hello world

engine$terminate()
```

In this case, two parameters are passed to the `ScriptEngine$new` method: the script engine name `"ruby"`, and the path to the JRuby script engine JAR file. As before, we assign the value `"Hello world"` to a global variable named `message` and print it to the console. Notice that we prefix the global variable with a dollar sign: `$message`. This syntax is peculiar to global variables in the Ruby language.

Language	Script engine names
Groovy	`groovy`, `Groovy`
JavaScript (Nashorn)	`js`, `JS`, `JavaScript`, `javascript`, nashorn, `Nashorn`, `ECMAScript`, `ecmascript`
JRuby (Ruby)	`jruby`, `ruby`
Jython (Python)	`jython`, `python`
Kotlin	`kotlin`

Table 2: The `ScriptEngine$new` constructor method creates a new script engine instance for a given language using the associated names in this table. Script engine names are case sensitive.

5.2 Executing script

jsr223 provides several methods to execute script. The lines

```
return.value <- engine %~% script
return.value <- engine$eval(script)
```

both evaluate the expression in the character vector `script`. The return value is the result of the last expression in the script, if any, or `NULL` otherwise. Text written to standard output by the script engine is printed to the R console. The following line executes JavaScript code and assigns the result to an R variable.

```
result <- engine %~% "isFinite(1);"
```

The following lines also execute script, but there are no return values. This notation is convenient if the last expression in the snippet returns unneeded data or an unsupported type (like a function).

```
engine %@% script
engine$eval(script, discard.return.value = TRUE)
```

To execute a script file, use either of the following lines where `file.name` is the path or URL to the script file.

```
engine$source(file.name)
engine$source(file.name, discard.return.value = TRUE)
```

The methods `eval` and `source` take an argument named `bindings` that accepts an R named list. The name/value pairs in the list replace the script engine's global bindings during script execution. The following JavaScript example demonstrates this functionality. Notice that the result of `a + b` changes when bindings are specified.

```
engine$a <- 2
engine$b <- 3
engine$eval("a + b")

## 5

lst1 <- list(a = 6, b = 7)
engine$eval("a + b", bindings = lst1)

## 13
```

This script would throw an error because 'b' is not defined in the list.

```
lst2 <- list(a = 6)
engine$eval("a + b", bindings = lst2)
```

When the `bindings` parameter is not specified, the script engine reverts to the default global bindings.

```
engine$eval("a + b")

## 5
```

5.3 Sharing data between language environments

The following two lines of R code are equivalent: they convert an R object to a Java object and assign the new object to a variable `myValue` in the script engine's environment. This syntax is the same for all supported R data structures.

```
engine$myValue <- iris
engine$set("myValue", iris)
```

To retrieve `myValue` from the script engine (i.e., to convert a Java object to an R object), use either of the following lines.

```
engine$myValue
engine$get("myValue")
```

Remove the `myValue` variable with `engine$remove("myValue")`. List all bindings in the script engine's environment with `engine$getBindings()`.

Bindings are synonymous with global variables in most script engine environments. For example, the following sample creates a binding using the R interface and retrieves the value through JavaScript.

```
engine$myValue1 <- 5
engine %~% "myValue1;"
```

```
## [1] 5
```

This example does the opposite; it creates a new global variable in JavaScript and returns its value through the **jsr223** binding interface.

```
engine %@% "var myValue2 = 6;"
engine$myValue2
```

```
## [1] 6
```

The Kotlin language is an exception to this behavior. It handles bindings through a the global object `jsr223Bindings` as follows. See Kotlin idiosyncrasies for more information.

```
engine$myValue1 <- 5
engine %~% 'jsr223Bindings["myValue1"]'
```

```
## [1] 5
```

```
engine %@% 'jsr223Bindings["myValue2"] = 6'
engine$myValue2
```

```
## [1] 6
```

All data structures in Java-based scripting languages are backed by Java objects. Discover the Java class for any global variable using `engine$getJavaClassName("identifier")` where `identifier` is the variable's name.

Behind the scenes, **jsr223**'s simplified data exchange is provided by **jdx**: Java Data Exchange for R and **rJava**. The **jdx** package's functionality was originally part of **jsr223**, but it was

broken out into a separate package to simplify maintenance and to make its features available to other developers.

The **jdx** package (and hence **jsr223**) supports converting R vectors, factors, n-dimensional arrays, data frames, named lists, unnamed lists, nested lists (i.e., lists containing lists), and environments to generic Java objects. Row-major and column-major ordering options are available for arrays and data frames. R data types numeric, integer, character, raw, and logical are supported. Complex types and date/time classes are not supported.

Java scalars, n-dimensional arrays, collections, and maps can be converted to standard objects in the R environment. These structures cover all of the primary data types in the supported scripting languages. Moreover, collections and maps are ubiquitous in Java APIs; providing support for these structures gives R developers easy access to a vast number of data structures available on the Java platform. This includes most scripting language-specific structures such as Python dictionaries and native JavaScript objects.

All **jdx** data conversion options are mirrored by settings in **jsr223**. The most pertinent details are discussed in the following sections. For a more thorough discussion, see the vignette included with the **jdx** package.

5.4 Setting and getting script engine options

The **jsr223** `ScriptEngine` class exposes several methods that control settings for a script engine instance. These methods are named using the Java getter/setter convention: methods that set values are prefixed with "set" and methods that retrieve values begin with "get". For example, if `engine` is a script engine object, `engine$setArrayOrder('column-major')` will change the *array order* setting. The code `engine$getArrayOrder()` will retrieve the current *array order* setting.

5.5 Handling R vectors

By default, length-one R vectors are converted to Java scalars when passed to the script engine environment. If a Java length-one array is desired, wrap the value in the R "as-is" function (e.g., `I(myValue)`), or set the *length one vector as array* setting to `TRUE` using the `setLengthOneVectorAsArray` method. By default, length-one vectors are converted to Java scalars as demonstrated here.

```
engine$setLengthOneVectorAsArray(FALSE)
engine$myScalar <- 1
engine$getJavaClassName("myScalar")
```

```
## [1] "java.lang.Double"
```

Wrap a length-one vector with `I()` to indicate that an array should be created instead. In this case, the resulting Java class name is `"[D"` which denotes a primitive, double one-dimensional array.

To change the conversion behavior for all length-one vectors, set the *length one vector as array* setting to `TRUE`.

```
engine$setLengthOneVectorAsArray(TRUE)
engine$myArray <- 1
engine$getJavaClassName("myArray")
```

```
## [1] "[D"
```

Vectors of any length other than one are always converted to primitive Java arrays. The following code passes a vector of ten random normal deviates to the script engine environment. The first element of the resulting array is returned. **Note:** Java arrays use zero-based indexes.

```
set.seed(10)
engine$norms <- rnorm(10)
engine %~% "norms[0]"
```

```
## [1] 0.01874617
```

5.6 Handling R matrices and other n-dimensional arrays

By default, n-dimensional arrays are copied in row-major order. The following example demonstrates converting a simple 2 x 2 R matrix. Because the order is row-major, the last line of code returns the element in the first row, second column. Remember, Java arrays use zero-based indexes.

```
m <- matrix(1:4, 2, 2)
m
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```
engine$m <- m
engine %~% "m[0][1]"
```

```
## [1] 3
```

The `setArrayOrder` script engine method controls ordering for arrays converted from R to Java, and vice versa. Three array index ordering schemes are available: `'row-major'`, `'column-major'`, and `'column-minor'`. These settings control how the destination Java array is constructed.

Before describing the ordering schemes, it is helpful to think of n-dimensional arrays as collections of smaller structures. A one-dimensional array (a vector) is a collection of scalars. A two-dimensional array (a matrix) is a collection of one-dimensional arrays representing either rows or columns of the matrix. A three-dimensional array (a rectangular prism or cube) is a collection of matrices. A four-dimensional array is a collection of cubes, and so forth.

Now we describe the each of the *array order* settings.

- `'row-major'` – The data of the resulting Java n-dimensional array are ordered `[row][column][matrix]...[n]`. The **jsr223** package defaults to `'row-major'` because R syntax uses this indexing scheme (though R stores the array in memory using column-major order). This row-major scheme is not intuitive for Java programmers when n > 2 because Java n-dimensional arrays are constructed as high-order objects containing low-order objects.

- `'column-major'` – The data of the resulting Java n-dimensional array are ordered `[n]...[matrix][column][row]`. This ordering scheme is natural for Java programmers: the data contained in the one-dimensional arrays represent columns of the parent matrix.
- `'column-minor'` – The data of the resulting Java n-dimensional array are ordered `[n]...[matrix][row][column]`. This provides Java programmers with a natural ordering scheme where the arrays at the one-dimensional level represent rows of the parent matrix. For matrices, `'column-minor'` and `'row-major'` are equivalent.

Note: If an R array is converted to Java using a particular array order, use the same array order when converting it back from Java to R. Otherwise, the data will be in the wrong order.

In the following JavaScript example, a three-dimensional array is copied to the script engine using each of the three indexing options. We use the Java static method `deepToString` to create a string representation of the array that shows the resulting order of the data in the script engine.

```
a <- array(1:8, c(2, 2, 2))
a

## , , 1
##
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## , , 2
##
##      [,1] [,2]
## [1,]    5    7
## [2,]    6    8

engine$setArrayOrder("row-major")
engine$a <- a
engine %~% "java.util.Arrays.deepToString(a);"

## [1] "[[[1, 5], [3, 7]], [[2, 6], [4, 8]]]"

engine$setArrayOrder("column-major")
engine$a <- a
engine %~% "java.util.Arrays.deepToString(a);"

## [1] "[[[1, 2], [3, 4]], [[5, 6], [7, 8]]]"

engine$setArrayOrder("column-minor")
engine$a <- a
engine %~% "java.util.Arrays.deepToString(a);"

## [1] "[[[1, 3], [2, 4]], [[5, 7], [6, 8]]]"
```

5.7 Handling R data frames

R data frames can be converted to the script engine using either row-major or column-major order. Row-major order (the default) creates a list of records. This representation is perhaps the most common in programming for tabular data. Column-major order, on the other hand, creates a list of columns. Column-major structures are faster to create and are generally preferred for aggregate column calculations. Change the *data frame order* setting with the `setDataFrameRowMajor` method.

When the row-major setting is selected (i.e., `engine$setDataFrameRowMajor(TRUE)`), an R data frame is converted to a `java.util.ArrayList` object. The list contains `java.util.LinkedHashMap` objects that represent the rows of the data frame. Each member of the hash map is a name/value pair of a single field in the data frame. The name of the field is the corresponding column's name. The following example uses R's built-in `iris` data set to illustrate using row-major data frames in the script environment.

```
engine$setDataFrameRowMajor(TRUE)
engine$iris <- iris

# Return the number of rows.
engine %~% "iris.size()"

## [1] 150

# Retrieve the sepal length in the first row.
engine %~% "iris[0].get('Sepal.Length')"

## [1] 5.1

# Retrieve the second row as a list.
engine %~% "iris[1]"

## $`Sepal.Length`
## [1] 4.9
##
## $Sepal.Width
## [1] 3
##
## $Petal.Length
## [1] 1.4
##
## $Petal.Width
## [1] 0.2
##
## $Species
## [1] "setosa"
```

When the column-major setting is selected (i.e., `engine$setDataFrameRowMajor(FALSE)`), an R data frame is converted to a `java.util.LinkedHashMap` object. The map members are arrays representing the columns in the data frame.

Row names for data frames are not preserved during conversion. To include row names in the conversion, simply add them as a column in your data frame. We do not automatically include row names in conversion because it would require us to create an additional element in the Java map with a reserved key value such as `_row`. Instead, we leave the decision of how to handle row names to the developer.

The following commented example uses R's built-in `mtcars` data set to illustrate basic functionality.

```
engine$setDataFrameRowMajor(FALSE)

# 'mtcars' is an R data frame containing information for 32 cars. 'mtcars'
# stores vehicle names as row names. Row names are not preserved during
# conversion. This line creates a new R data frame with the vehicle names as
# a new column 'name'.
df <- data.frame(name = row.names(mtcars), mtcars)

# This line converts the new data frame to a Java map named 'mtcars'.
engine$mtcars <- df

# Return the number of columns in the map.
engine %~% "mtcars.size()"

## [1] 12

# Access each column using the map's 'get' method and the column's name. This
# line returns the first element of the column 'name'.
engine %~% "mtcars.get('name')[0]"

## [1] "Mazda RX4"

# Add a new column named 'cylsize' representing the size of a single cylinder.
engine$cylsize <- mtcars[, "disp"] / mtcars[, "cyl"]
engine %@% "mtcars.put('cylsize', cylsize)"

# Remove the columns 'name' and 'cylsize'.
engine %@% "mtcars.remove('name')"
engine %@% "mtcars.remove('cylsize')"

# Compare the contents of the map to the original data frame in R.
all.equal(mtcars, engine$mtcars, check.attributes = FALSE)

## [1] TRUE
```

Groovy and JavaScript support an additional syntax that allows map elements to be accessed like object properties instead of using the `get` and `put` methods.

```
# The following two lines are equivalent in Groovy and JavaScript.
engine %~% "mtcars.cyl[0];"
engine %~% "mtcars.get('cyl')[0];"
```

5.8 Handling R factors

R factors are comprised of a character vector of levels and an integer vector of indexes that reference the levels. For example, if the integer vector `5:7` is converted to a factor, the levels will be `c("5", "6", "7")` and the indexes will be `c(1L, 2L, 3L)`. The script engine *coerce factors* setting determines how the factor levels are handled when converting the factor to a Java array. When this setting is enabled (e.g., `engine$setCoerceFactors(TRUE)`), an attempt is made to coerce the factor levels to integer, numeric, or logical values. If coercion fails, the character levels are used. When *coerce factors* is disabled, the factor is always converted to a string array. The *coerce factors* setting applies to standalone factors as well as factors in data frames.

After **jsr223** converts an R factor to a Java array, there is no consistent way to determine whether the array was originally created from an R factor. Therefore, if an R factor is copied to the script engine, and then the resulting array is returned to R, it will be converted to an R vector, not a factor.

When creating a data frame in R, character vectors are converted to factors by default. The **jsr223** package follows this standard when a qualifying Java object is converted to an R data frame. The `setStringsAsFactors` method modifies this behavior. The method takes one of three values: `NULL`, `TRUE`, and `FALSE`. If `NULL` is specified (the default), the R system setting is used (see
`getOption("stringsAsFactors")`). A value of `TRUE` ensures that character vectors are always converted to factors for new data frames. Finally, a setting of `FALSE` disables conversion to factors.

5.9 Handling R lists and environments

The **jsr223** package converts R lists and environments to Java objects. List elements may be any R data structure supported by **jsr223**, including other lists (i.e., nested lists). There is no limitation to the levels of nesting.

R named lists and environments are converted to Java `java.util.HashMap` objects. See Handling R data frames for map code examples. The only difference is that a data frame's contents are always converted to a map of arrays. For lists, the map elements may be any data structure.

R unnamed lists are converted to Java objects implementing the `java.util.ArrayList` interface. The following code demonstrates basic `java.util.ArrayList` functionality.

```
# Create an unnamed list with three elements.
engine$list <- list(c("a", "b", "c"), TRUE, pi)

# Members in the list are accessed by index. This line returns the first element.
engine %~% "list[0]"

## [1] "a" "b" "c"

# Replace an element in the list.
engine %@% "list[0] = 'replaced'"

# Add a new element to the end of the list.
engine %@% "list.add('last item')"
```

```
# Insert a new item before the first item.
engine %@% "list.add(0, 'first item')"

# Remove the last item.
engine %@% "list.remove(list.size() - 1)"

# Return the number of elements
engine %~% "list.size()"

## [1] 4
```

5.10 Data exchange details

So far, we have discussed all of the basic functionality and settings related to data exchange. This section includes a few additional notes for data exchange. A comprehensive guide, including details for unexpected conversion behaviors, is included in the **jdx** package vignette.

R reserves special `NA` values to indicate missing types. Table 3 outlines how `NA` values are handled for different R data types. Table 4, in turn, describes how Java null values are interpreted when converting Java objects to R.

Because **jsr223** converts data to generic Java data structures, R attributes such as names cannot always be included in conversion. For example, R vectors are converted to native Java arrays, therefore names associated with vector elements must be discarded. Likewise, dimension names are not preserved for n-dimensional structures. Column names for data frames are preserved, but row names are not. To preserve data frame row names, simply copy the names to a new column before converting the data frame.

The **jsr223** package always converts R vectors and arrays to Java arrays. Java arrays are intuitive to use in all of the supported scripting environments. However, the supported scripting languages can also create array structures that are not native Java arrays. **jsr223** also supports converting these language-specific array and collection structures to R vectors and arrays.

Java n-dimensional arrays whose subarrays of a given dimension are not the same dimension are known as *ragged arrays*. Ragged arrays cannot be converted to R arrays. Instead, **jsr223** translates ragged arrays to lists of the appropriate object. For example, a matrix containing subarrays of different lengths will be converted to an R list of vectors. Likewise, a three-dimensional array containing two matrices of different dimensions will be converted to an R list of matrices.

R Structure	NA Behavior
`numeric`	`NA_real_` maps to a reserved value.
`integer`	`NA_integer_` maps to a reserved value.
`character`	`NA_character_` maps to Java `null`.
`logical`	`NA` maps to Java `false` with a warning.

Table 3: R reserves special `NA` values to indicate missing types. This table outlines how `NA` values are converted to Java values.

Java Structure	Java null Conversion
`Boolean[]..[]`	`null` maps to `FALSE` with a warning.
`Byte[]..[]`	`null` maps to `raw 0x00` with a warning.
`Character[]..[]`	`null` maps to `NA_character_`.
`Double[]..[]`	`null` maps to `NA_real_`.
`Float[]..[]`	`null` maps to `NA_real_`.
`Integer[]..[]`	`null` maps to `NA_integer_`.
`java.math.BigDecimal[]..[]`	`null` maps to `NA_real_`.
`java.math.BigInteger[]..[]`	`null` maps to `NA_real_`.
`Long[]..[]`	`null` maps to `NA_real_`.
`Object[]..[]`	`null` maps to `NULL`.
`Short[]..[]`	`null` maps to `NA_integer_`.
`java.lang.String[]..[]`	`null` maps to `NA_character_`.

Table 4: Java `null` indicates missing or uninitialized values. This table outlines how `null` is interpreted when converting Java objects to R. The syntax `[]..[]` is used to indicate an array of one or more dimensions.

As described earlier, R unnamed lists are converted to `java.util.ArrayList` objects. The `ArrayList` class implements the `java.util.Collection` interface. This is one of the most basic interfaces in Java and it is common to a large number of structures. **jsr223** converts Java objects implementing the `java.util.Collection` interface to vectors, n-dimensional arrays, data frames, and unnamed lists, depending on the structure's content. In some cases an R list converted to a Java object, and then converted back to an R object, may not produce an R list. See the sections "Java Collections" and "Conversion Issues" in the **jdx** package vignette for conversion rules and in-depth explanations.

The jdx package converts R raw values to Java byte values and vice versa. R raw values and Java byte values are both 8 bits, but they are interpreted differently. R raw values range from 0 to 255 (i.e., unsigned bytes). Java byte values range from -128 to 127 (i.e., signed bytes). The 8-bit value 0xff represents 255 in R, but is -1 in Java. Usually this discrepancy is not an issue because raw and byte values are used to store and transfer binary data such as images. If human-readable values are important, use integer vectors instead.

5.11 Calling script functions and methods

Functions and methods defined in script can be called directly from R via the `invokeFunction` and `invokeMethod` script engine methods. Any number of supported R structures can be passed as parameter values.

Note: The Groovy, Python, and Kotlin engines can use `invokeMethod` to call methods of Java objects. The JavaScript and Ruby engines only support calling methods of native scripting objects. For the latter two engines, we recommend wrapping Java objects in native functions or methods to facilitate their use from R.

As described in Handling R vectors, length-one vectors are converted to Java scalars by default. One way to ensure that a vector is always converted to a Java array is by wrapping it in the "as-is" function `I()`. This feature is particularly useful when passing multiple parameters to a script function. In the same function, some parameters may require scalars while others require arrays. Simply use `I()` to indicate which vectors should be converted to arrays.

The following example demonstrates calling a simple JavaScript function, `sumThis`, that sums the elements of an array. If the first parameter is not an array, the function throws an error.

```
# Define a simple global function 'sumThis'.
engine %@% "
function sumThis(a) {
  if (!a.getClass().isArray())
    throw 'Not an array.';
  sum = 0;
  for (i = 0; i < a.length; i++) {
      sum += a[i];
  }
  return sum;
}
"

# Set the default length-one vectors setting so the example works as intended.
engine$setLengthOneVectorAsArray(FALSE)

# Call the function with a vector with length > 1.
vector <- c(1, 2, 3)
engine$invokeFunction("sumThis", vector)

## [1] 6

# If the vector is length-one, an error is thrown because an array parameter
# is expected.
vector <- 1
engine$invokeFunction("sumThis", vector)

## javax.script.ScriptException: Not an array. in <eval> at line number 4 at
## column number 4

# Try again, this time marking the vector as-is, meaning that it should
# always be converted to an array.
vector <- 1
engine$invokeFunction("sumThis", I(vector))

## [1] 1
```

The next example demonstrates using `invokeMethod`. It is essentially the same as `invokeFunction` except that the first two parameters require the object's name and method, respectively.

```
# Invoke the 'abs' (absolute value) method of the JavaScript 'Math' object.
```

```
engine$invokeMethod("Math", "abs", -3)
```

```
## [1] 3
```

5.12 String interpolation

jsr223 features string interpolation before code evaluation. R code placed between `@{` and `}` in a code snippet is evaluated and replaced by the a string representation of the return value before the snippet is executed by the script engine. A script may contain multiple `@{...}` expressions. String interpolation is enabled by default. It can be disabled using `engine$setInterpolation(FALSE)`.

Note: Interpolated decimal values may lose precision when coerced to a string.

This example simply sums two numbers. The section Callbacks includes a more interesting interpolation example involving recursion.

```
a <- 1; b <- 2
engine %~% "@{a} + @{b}"
```

```
## 3
```

Interpolation expressions are evaluated in the current scope. The following example shows that interpolation locates the value defined in the function's scope before the global variable of the same name.

```
a <- 1

constantFunction <- function() {
  a <- 3
  engine %~% "@{a}"
}

constantFunction()
```

```
## [1] 3
```

5.13 Callbacks

Embedded scripts can access the R environment using the **jsr223** callback interface. When a script engine is started, **jsr223** creates a global object named `R` in the script engine's environment. This object is used to execute R code and set/get variables in the R session's global environment.

This code example demonstrates setting and getting a variable in the R environment. For Ruby, remember to prefix the global variable `R` with a dollar sign.

```
engine %@% "R.set('a', [1, 2, 3])"
engine %~% "R.get('a')"
```

```
## [1] 1 2 3
```

Note: Changing any of the data exchange settings will affect the behavior of the callback interface. For example, using `engine$setLengthOneVectorAsArray(TRUE)` will cause `R.get("pi")` to return an array with a single element instead of a scalar value.

Execute R script with `R.eval(script)` where `script` is a string containing R code. This example returns a single random normal draw from R.

```
set.seed(10)
engine %~% "R.eval('rnorm(1)')"

## [1] 0.01874617
```

Infinite recursive calls between R and the script engine are supported. The only limitation is available stack space. The following code demonstrates recursive calls and string interpolation with a countdown.

```
recursiveCountdown <- function(start.value) {
  cat("T minus ", start.value, "\n", sep = "")
  if (start.value > 0)
    engine %~% "R.eval('recursiveCountdown(@{start.value - 1})');"
}

engine %~% "R.eval('recursiveCountdown(3)')"

## T minus 3
## T minus 2
## T minus 1
## T minus 0
```

5.14 Embedding R in another scripting language

It is often desirable to use R as an embedded language. The **jsr223** interface does not provide a standalone interface to call into R. However, the same functionality can be achieved with the `RScript` command line executable, a simple launch script, and the **jsr223** callback interface. The following R script is an example of a launch script for Groovy. It executes any Groovy script file provided as a command line parameter.

```
library("jsr223")
engine <- ScriptEngine$new("groovy", "~/my-path/groovy-all.jar")
tryCatch (
  engine$source(commandArgs(TRUE)[1], discard.return.value = TRUE),
  error = function(e) { cat(e$message, "\n", sep = "") },
  finally = { engine$terminate() }
)
```

The following command line uses the launch script to execute a Groovy script. The launch script is named '`groovy-launcher.R`' and '`source.groovy`' is an arbitrary Groovy source file.

```
RScript groovy-launcher.R source.groovy
```

With this setup, a developer can author a Groovy script in a dedicated script editor. The Groovy script can embed R using the **jsr223** callback interface as if it were a standalone interface. The command line above can be provided to a code editor to execute the Groovy script on demand. The Groovy code below is an example of embedding R.

```
// Set a variable named 'probabilities' in the R global environment.
R.set('probabilities', [0.25, 0.5, 0.20, 0.05]);

// Take a random draw of size two using the given probabilities.
draws = R.eval('sample(4, 2, prob = probabilities)');
```

5.15 Compiling script

The Java Scripting API supports compiling script to Java bytecode before evaluation. If unstructured code (i.e., code not encapsulated in methods or functions) is to be executed repeatedly, compiling it will improve performance. This feature does not apply to methods and functions as they are compiled on demand.

The following two lines show how to compile code snippets and source files, respectively. For the latter, local disk files or URLs can be specified. In both cases, a compiled script object is returned.

```
cs <- engine$compile(script)
cs <- engine$compileSource(file.name)
```

The compiled script object has a single method, `eval`, that is used to execute the compiled code. It can be argued that the method should be called `exec` in this case, but our interface follows the Java Scripting API naming scheme. The following trivial example demonstrates the compiled script interface.

```
# Compile a code snippet.
cs <- engine$compile("c + d")

# This line would throw an error because 'c' and 'd' have not yet been declared.
## cs$eval()

engine$c <- 2
engine$d <- 3
cs$eval()

## 5
```

The `eval` method takes an argument named `bindings` that accepts an R named list. The name/value pairs in the list replace the script engine's global bindings during script execution as shown in this code sample.

```
lst <- list(c = 6, d = 7)
cs$eval(bindings = lst)

## 13
```

```
# When 'bindings' is not specified, the script engine reverts to the original
# environment.
cs$eval()

## 5
```

The `discard.return.value` argument of the `eval` method determines whether the return value of a script is discarded. The default is `FALSE`. The following line executes code but does not return a value.

```
cs$eval(discard.return.value = TRUE)
```

5.16 Handling console output

When script is evaluated, any text printed to standard output appears in the R console by default. Console output can be disabled entirely with `engine$setStandardOutputMode('quiet')`. To resume printing output to the console, use `engine$setStandardOutputMode('console')`.

Text printed to the console by a script engine cannot be captured using R's `sink` or `capture.output` methods. To capture output, set the *standard output mode* setting to `'buffer'`. In this JavaScript example, the `print` method output will not appear in the R console; it will be stored in an internal buffer. The contents of the buffer can be retrieved and cleared using the `getStandardOutput` method.

```
engine$setStandardOutputMode("buffer")
engine %@% ("print('abc');")
engine$getStandardOutput()

## [1] "abc\n"
```

Alternatively, the buffer can be discarded using the `clearStandardOutput` method.

```
engine %@% ("print('abc');")
engine$clearStandardOutput()
```

5.17 Console mode: a simple REPL

jsr223 provides a simple read-evaluate-print-loop (REPL) for interactive code execution. This feature is inspired by Jeroen Ooms's **V8** package. The REPL is useful for quickly setting and inspecting variables in the script engine. Returned values are printed to the console using `base::dput`. The `base::cat` function is not used because it does not handle complex data structures.

Use `engine$console()` to enter the REPL. Enter 'exit' to return to the R prompt. The REPL supports only single line entry: no line continuations or carriage returns are allowed. This limitation arises from the fact that the Java Scripting API does not support code validation.

The following output was produced by a Python REPL session. The code creates a Python dictionary object and accesses the elements. The tilde character ('~') indicates a prompt.

```
python console. Press ESC, CTRL + C, or enter 'exit' to exit the console.
~ dict = {"first": 1, "second": 2}
```

```
~ dict["first"]
1
~ dict["second"]
2
~ exit
Exiting console.
```

Most developers are familiar with the command history in the R REPL. Unfortunately, command history for the **jsr223** REPL is unreliable or non-existent because there is no functional standard for saving and restoring commands in R consoles.

6 R with Groovy

Groovy is a dynamically typed programming language that closely follows Java syntax. Hence, the **jsr223** integration for Groovy enables developers to essentially embed Java language solutions in R. There are some minor language differences between Groovy and Java; they are described in the online guide Differences with Java.

6.1 Groovy idiosyncrasies

Top-level (i.e., global) variables created in Groovy script will be discarded after script evaluation unless the variables are declared using specific syntax. To create a binding that persists in the script engine environment, declare a top-level variable omitting the type definition and Groovy's `def` keyword. For example `myValue = 42` will create a global variable. The `@myValue` notation cannot be used. To specify a data type for a global variable, use a constructor (`myVar = new Integer(42)`) or a type suffix (`myVar = 42L`).

6.2 Groovy and Java classes

If you already know Java, using Java classes in Groovy will be very familiar. Java package members are imported (i.e., made accessible to the script) using the `import` statement. Groovy automatically imports many common Java packages by default such as `java.io.*`, `java.lang.*`, `java.net.*`, and `java.util.*`. If the package is not part of the JRE, add the package's JAR file to the `class.path` parameter of the `ScriptEngine$new` constructor when creating the script engine.

Tip: Supply class paths as separate elements of a vector instead of concatenating the paths with the usual path delimiters (";" for Windows, and ":" for all others). This will make your code platform-independent and easier to read.

This example demonstrates using Java objects in R. We use the Apache Commons Mathematics Library to sample from a bivariate normal distribution.

```
library("jsr223")

# Include both the Groovy script engine and the Apache Commons Mathematics
# libraries in the class path. Specify the paths seperately in a character
# vector.
engine <- ScriptEngine$new(
  engine.name = "groovy",
```

```
  class.path = c("~/my-path/groovy-all.jar", "~/my-path/commons-math3-3.6.1.jar")
)

# The getClassPath method displays the current class path.
engine$getClassPath()

# Define the means vector and covariance matrix that will be used to create the
# bivariate normal distribution.
engine$means <- c(0, 2)
engine$covariances <- diag(1, nrow = 2)

# Import the package member and instantiate a new class. For Groovy, excluding
# the type and 'def' keyword will make 'mvn' a global variable.
engine %@% "
  import org.apache.commons.math3.distribution.MultivariateNormalDistribution;
  mvn = new MultivariateNormalDistribution(means, covariances);
"

# Take a sample.
engine$invokeMethod("mvn", "sample")

## [1] 0.3279374 0.8652296

# Take three samples.
replicate(3, engine$invokeMethod("mvn", "sample"))

##           [,1]      [,2]      [,3]
## [1,] 0.9924368 -1.295875 0.2025815
## [2,] 2.5145855  2.128243 1.1666272

engine$terminate()
```

7 R with JavaScript

The popularity of JavaScript has overflowed the arena of web development into standalone solutions involving databases, charting, machine learning, and network-enabled utilities, to name just a few. Many of these solutions can be harnessed by R with the help of **jsr223**. Even browser-based scripts that require a document object model (DOM) can be executed using Java's `WebView` browser. Popular JavaScript solutions can be found at JavaScripting, an online database of JavaScript solutions. Github also lists trending solutions for JavaScript, as well as other languages.

Nashorn is the JavaScript dialect included in Java 8. Nashorn implements ECMAScript 5.1. No download is required to use JavaScript with **jsr223**. JavaScript Nashorn provides wide support for Java classes, including the ability to extend classes and implement interfaces. For details, see the official Nashorn documentation.

Data in JavaScript objects be converted to R named lists or data frames, depending on content. The following converts a simple JavaScript object to an R named list.

```
engine %@% 'var person = {fname:"Jim", lname:"Hyatt", title:"Principal"};'
engine$person
```

```
## $`fname`
## [1] "Jim"
##
## $lname
## [1] "Hyatt"
##
## $title
## [1] "Principal"
```

7.1 JavaScript and Java classes

Nashorn provides several methods to reference JavaScript classes. We demonstrate the two most common methods. The first approach is the one recommended in the Nashorn documentation; it uses the built-in `Java.type` method to create a JavaScript reference to the class. This reference can be used to access static members or to create instances. In this example, we use a static method of the `java.util.Arrays` class to sort a vector of integers.

```
engine %~% "
  var Arrays = Java.type('java.util.Arrays');
  var random = R.eval('sample(5)');
  Arrays.sort(random);
  random;
"
```

```
## [1] 1 2 3 4 5
```

A second approach involves accessing the target class using its fully-qualified name. This approach requires more overhead per call, but it is more convenient than using `Java.type`. The following code is functionally equivalent to the previous example.

```
engine %~% "
  var random = R.eval('sample(5)');
  java.util.Arrays.sort(random);
  random;
"
```

```
## [1] 1 2 3 4 5
```

The `Java.type` method is required to create Java primitives. In this example, we create a Java integer array with five elements.

```
engine %~% "
  var IntegerArrayType = Java.type('int[]');
  var myArray = new IntegerArrayType(5);
  myArray;
"
```

```
## [1] 0 0 0 0 0
```

Next, we reproduce the Groovy bivariate normal example in JavaScript. The code demonstrates importing an external library and highlights an important limitation in Nashorn regarding `invokeMethod`.

```
library("jsr223")

# Include the Apache Commons Mathematics library in class.path.
engine <- ScriptEngine$new(
  engine.name = "js",
  class.path = "~/my-path/commons-math3-3.6.1.jar"
)

# Define the means vector and covariance matrix that will be used to create the
# bivariate normal distribution.
engine$means <- c(0, 2)
engine$covariances <- diag(1, nrow = 2)

# Import the package member and instantiate a new class.
engine %@% "
  var MultivariateNormalDistributionClass = Java.type(
    'org.apache.commons.math3.distribution.MultivariateNormalDistribution'
  );
  var mvn = new MultivariateNormalDistributionClass(means, covariances);
"

# This line would throw an error. Nashorn JavaScript supports 'invokeMethod' for
# native JavaScript objects, but not for Java objects.
#
## engine$invokeMethod("mvn", "sample")

# Instead, use script...
engine %~% "mvn.sample();"

## [1] 0.3279374 0.8652296

# ...or wrap the method in a JavaScript function.
engine %@% "function sample() {return mvn.sample();}"
engine$invokeFunction("sample")

## [1] 0.2527757 1.1942332

# Take three samples.
replicate(3, engine$invokeFunction("sample"))

##           [,1]      [,2]      [,3]
## [1,] 0.9924368 -1.295875 0.2025815
## [2,] 2.5145855  2.128243 1.1666272

engine$terminate()
```

7.2 Using JavaScript solutions - Voca

The **jsr223** package enables developers to access solutions developed in other languages by simply sourcing a script file. For example, Voca is a popular string manipulation library that simplifies many difficult tasks such as word wrapping and diacritic detection (e.g., the "é" café). Using Voca with **jsr223** is simply a matter of sourcing a single script file. This sample script loads Voca and demonstrates its functionality.

```
# Source the Voca library. This creates a utility object named 'v'.
engine$source(
  "https://raw.githubusercontent.com/panzerdp/voca/master/dist/voca.min.js",
  discard.return.value = TRUE
)

# 'prune' truncates string, without break words, ensuring the given length, including
# a trailing "..."
engine %~% "v.prune('A long string to prune.', 12);"

## [1] "A long..."

# Provide a different suffix to 'prune'.
engine %~% "v.prune('A long string to prune.', 16, ' (more)');"

## [1] "A long (more)"

# Voca supports method chaining.
engine %~% "
v('Voca chaining example')
  .lowerCase()
  .words()
"

## [1] "voca"     "chaining" "example"

# Split graphemes.
engine %~% "v.graphemes('cafe\u0301');"

## [1] "c" "a" "f" "é"

# Word wrapping.
engine %~% "v.wordWrap('A long string to wrap', {width: 10});"

## [1] "A long\nstring to\nwrap"

# Word wrapping with custom delimiters.
engine %~% "
v.wordWrap(
  'A long string to wrap',
  {
    width: 10,
```

```
    newLine: '<br/>',
    indent: '__'
  }
);
"

## [1] "__A long<br/>__string to<br/>__wrap"
```

8 R with Python

Like R, the Python programming language is used widely in science and analytics. Python has many powerful language features, yet it is known for being concise and easy to read. The Jython project has migrated core Python to the Java platform. This implementation does not include popular libraries such as NumPy and SciPy. These libraries compile to machine code and, as such, they are not compatible with the JVM. However, JVM implementations of some Python native libraries are being developed in a related project, JyNI (the Jython Native Interface). To include these libraries in a **jsr223** solution, download the JyNI JAR file and include it in the class path when instantiating a Jython script engine.

8.1 Python idiosyncrasies

Leading white space is significant in Python; it is used to delimit code blocks. Avoid syntax errors by left-aligning code in multi-line string snippets as shown in the examples.

8.2 Python and Java classes

To create a Java object in Python, simply import the associated package and call the class constructor. The Jython User Guide provides further details for using Java classes. This example generates a random number using the `java.util.Random` class. Notice that the Python code is not indented; leading white space is significant.

```
# Create an object from the java.util.Random class.
engine %~% "
from java.util import Random
r = Random(10)
"


# Jython supports invoking Java methods.
engine$invokeMethod("r", "nextDouble")

## [1] 0.7304303
```

Jython's `jarray` module is required to create native Java arrays. The `array` method copies a Python sequence to a Java array of the given type. The `zeros` method initializes a Java array of the requested type with zero or `null`. This code snippet demonstrates both methods.

```
# Use 'jarray.array' to copy a sequence to a Java array of the requested type.
engine %~% "
```

```
from jarray import *
myArray = array([3, 2, 1], 'i')
"
engine$myArray

## [1] 3 2 1

# Alternatively, use zeros to initialize an array with zeros or null. This
# example allocates an array and udpates the values with a loop.
engine %~% "
myArray = zeros(5, 'i')
for i in range(myArray.__len__()):
  myArray[i] = i
"
engine$myArray

## [1] 0 1 2 3 4
```

8.3 A simple Python HTTP server

This code sample creates a simple HTTP server. It demonstrates calling Python class members from R and calling R code from Python. The Python script below defines two classes: the `MyHandler` class processes HEAD and GET requests for the server; and the `MyServer` class is used from an R script to start and stop the web server. The Python code is adapted from the Python Wiki.

```
import time
import BaseHTTPServer

# HTTP request handler class
class MyHandler(BaseHTTPServer.BaseHTTPRequestHandler):
    def do_HEAD(s):
        s.send_response(200)
        s.send_header("Content-type", "text/html")
        s.end_headers()
    def do_GET(s):
        print time.asctime(), "Received request"
        s.send_response(200)
        s.send_header("Content-type", "text/html")
        s.end_headers()
        s.wfile.write("<html><head><title>R/Python HTTP Server</title></head>")
        html = R.eval('getHtmlTable()') # Get HTML table from R.
        s.wfile.write(html)
        s.wfile.write("</body></html>")

class MyServer:
    def __init__(self, host_name, port_number, timeout):
        self.host_name = host_name
        self.port_number = port_number
```

```
        server_class = BaseHTTPServer.HTTPServer
        self.httpd = server_class((self.host_name, self.port_number), MyHandler)
        self.httpd.timeout = timeout
        print time.asctime(), "Server Started - %s:%s" % (self.host_name, self.port_number)
    def handle_request(self):
        # This method exists only for demonstration purposes. For a more robust
        # implementation, see 'SocketServer.serve_forever()'.
        self.httpd.handle_request()
    def close(self):
        self.httpd.server_close()
        print time.asctime(), "Server Stopped - %s:%s" % (self.host_name, self.port_number)
```

The R script here sources the Python script and starts the web server. It also defines `getHtmlTable`: a function that generates HTML content for the web server. Run the R script and point a web browser to `http://localhost:8080` to see the result. For demonstration purposes, the R script shuts down the Python web server automatically after 60 seconds.

```
library("xtable")
library("jsr223")

# Format the iris data set as an HTML table. This function will be called from
# the Python web server in response to an HTTP GET request.
getHtmlTable <- function() {
  t <- xtable(iris, "Iris Data")
  html <- capture.output(print(t, type = "html", caption.placement = "top"))
  paste0(html, collapse = "\n")
}

# Start the python engine.
engine <- ScriptEngine$new(
  engine.name = "python",
  class.path = "~/my-path/jython-standalone.jar"
)

# Source the Python script.
engine$source("./python-http-server.py", discard.return.value = TRUE)

runServer <- function(server.runtime = 60) {
  # Automatically shut down server when this function exits.
  on.exit(
    {
      engine$invokeMethod("server", "close")
      engine$terminate()
    }
  )

  # Create an instance of Python 'MyServer' class which starts the server at the
  # specified port with the given request timeout in seconds. A timeout would
  # not be used in a production scenario.
  engine %@% "server = MyServer('localhost', 8080, 2)"
```

```
  # Handle requests for 'server.runtime' seconds before shutting down. The
  # 'handle_request' method waits for the timeout specified in the 'MyServer'
  # constructor before returning to the event loop to allow interruptions. In a
  # true web service, the R side would not be involved in monitoring requests.
  # See Python's 'SocketServer.serve_forever()' for more information.
  started <- as.numeric(Sys.time())
  while(as.numeric(Sys.time()) - started < server.runtime)
    engine$invokeMethod("server", "handle_request")
}

runServer(60)
```

9 R with Ruby

The Ruby programming language is a general-purpose, object-oriented programming language invented by Yukihiro Matsumoto. According to Matsumoto, he designed the language to "help every programmer in the world to be productive, and to enjoy programming, and to be happy" (Matsumoto, 2008). JRuby is a Java implementation of the Ruby language. It is compatible with the popular web application framework Ruby on Rails.

9.1 Ruby idiosyncrasies

Global variables in Ruby script must be prefixed with a dollar sign. Hence, if we create a variable `myValue` using a **jsr223** assignment (e.g., `engine$myValue <- 10`), it is accessed in Ruby script as `$myValue`. Do not use the dollar sign prefix when accessing global variables via **jsr223** methods (e.g., `engine$get("myValue")`).

We have observed a bug in JRuby's exception handling: when JRuby encounters an error, the engine may continue to throw errors erroneously in subsequent evaluation requests. If this happens, restart the script engine.

9.2 Ruby and Java classes

JRuby implements several methods to access Java classes in Ruby syntax. For a comprehensive guide, see Calling Java from JRuby. We demonstrate the most intuitive syntax using the multivariate normal random sampler.

```
library("jsr223")

# Include both the JRuby script engine and the Apache Commons Mathematics
# libraries in the class path. Specify the paths seperately in a character
# vector.
engine <- ScriptEngine$new(
  engine.name = "ruby",
  class.path = c(
    "~/my-path/jruby-complete.jar",
    "~/my-path/commons-math3-3.6.1.jar"
  )
```

```
)

# Define the means vector and covariance matrix that will be used to create the
# bivariate normal distribution.
engine$means <- c(0, 2)
engine$covariances <- diag(1, nrow = 2)

# Import the class and create a new object from the class.
engine %@% "
java_import org.apache.commons.math3.distribution.MultivariateNormalDistribution
$mvn = MultivariateNormalDistribution.new($means, $covariances)
"

# This line would throw an error. JRuby supports 'invokeMethod' for
# native Ruby objects, but not for Java objects.
#
## engine$invokeMethod("mvn", "sample")

# Instead, use script...
engine %~% "$mvn.sample()"

## [1] 0.3279374 0.8652296

# ...or wrap the method in a function.
engine %@% "
def sample()
  return $mvn.sample()
end
"
engine$invokeFunction("sample")

## [1] 0.2527757 1.1942332

# Take three samples.
replicate(3, engine$invokeFunction("sample"))

##           [,1]      [,2]      [,3]
## [1,] 0.9924368 -1.295875 0.2025815
## [2,] 2.5145855  2.128243 1.1666272

engine$terminate()
```

9.3 Ruby gems

Ruby libraries and programs are distributed in a standardized package format called a *gem*. We demonstrate using gems in **jsr223** with Benjamin Curtis's `faker`: a library used to produce fake records for data sets.

A full installation of JRuby is required to use gems. Install JRuby and using the instructions found in Getting Started with JRuby. Install the `faker` gem using '`gem install faker`' in a terminal.

To access `faker` with **jsr223**, the paths to the gem and its dependencies must be added to the `ScriptEngine$new` class path. These paths can be discovered using the JRuby REPL, `jirb`, in a terminal session as shown here.

```
me@ubuntu:~$ jirb
irb(main):001:0> require 'faker'
=> true
irb(main):002:0> puts $LOAD_PATH
~/jruby-9.1.15.0/lib/ruby/gems/shared/gems/concurrent-ruby-1.0.5-java/lib
~/jruby-9.1.15.0/lib/ruby/gems/shared/gems/i18n-0.9.3/lib
~/jruby-9.1.15.0/lib/ruby/gems/shared/gems/faker-1.8.7/lib
~/jruby-9.1.15.0/lib/ruby/2.3/site_ruby
~/jruby-9.1.15.0/lib/ruby/stdlib
=> nil
irb(main):003:0> exit
```

These resulting paths will be required along with the path to 'jruby.jar' (the latter is in the 'lib' subfolder of the JRuby installation). Supply these paths to the `class.path` parameter of the **jsr223** `ScriptEngine$new` method when creating the script engine instance. In our experience, the '`site_ruby`' path did not exist. If `ScriptEngine$new` throws an error indicating a path does not exist, simply exclude it from the class path.

The code below uses the `faker` gem to generate a data frame containing fake names and titles.

```
library("jsr223")

class.path <- "
~/jruby-9.1.12.0/lib/jruby.jar
~/jruby-9.1.12.0/lib/ruby/gems/shared/gems/i18n-0.8.6/lib
~/jruby-9.1.12.0/lib/ruby/gems/shared/gems/faker-1.8.4/lib
~/jruby-9.1.12.0/lib/ruby/stdlib
"
class.path <- unlist(strsplit(class.path, "\n", fixed = TRUE))

engine <- ScriptEngine$new(
  engine.name = "jruby",
  class.path = class.path
)

# Import the required Ruby libraries.
engine %@% "require 'faker'"

# To create data deterministically, set a seed.
engine %@% "Faker::Config.random = Random.new(10)"

# Demonstrate unique, fake name.
engine %~% "Faker::Name.unique.name"
```

```
## [1] "Ms. Adrain Torphy"

# Define a Ruby function to return a given number of fake profiles.
engine %@% "
def random_profile(n = 1)
  fname = Array.new(n)
  lname = Array.new(n)
  title = Array.new(n)
  for i in 0..(n - 1)
    fname[i] = Faker::Name.unique.first_name
    lname[i] = Faker::Name.unique.last_name
    title[i] = Faker::Name.unique.title
  end
  return {'fname' => fname, 'lname' => lname, 'title' => title}
end
"

# Retrieve 5 fake profiles. The Ruby hash of same-length arrays will be
# automatically converted to a dataframe.
engine$invokeFunction("random_profile", 5)

##     fname      lname                           title
## 1 Quentin     Barton           Dynamic Paradigm Agent
## 2   Claud    Bernier         Regional Metrics Planner
## 3   Kevin Hodkiewicz      Investor Marketing Designer
## 4    Toni    Stracke Legacy Implementation Strategist
## 5  Jannie       Haag Dynamic Implementation Architect

engine$terminate()
```

10 R with Kotlin

Kotlin is a statically typed programming language that supports both functional and object-oriented programming paradigms. Kotlin is concise and pragmatic; in many cases, it requires less code than Java to accomplish the same task. Kotlin version 1.0 was released in 2016 (kot) making it the newest of the **jsr223**-supported languages.

Kotlin's JSR-223 implementation is progressing quickly though it is not complete. We will not list the deficiencies here as they will probably be resolved soon. See the **jsr223** issue tracker to review pending issues and workarounds. In the issue tracker search dialog, select the "Kotlin issues" label and include both open and closed issues.

10.1 Kotlin idiosyncrasies

The Kotlin script engine handles bindings through a global map object instead of creating global variables in the script engine environment. The best way to illustrate this behavior is by example. The following code creates and retrieves a binding `myValue` as you would expect.

```
engine$myValue <- 4
engine$myValue
```

```
## [1] 4
```

However, `myValue` will not be available as a global variable in Kotlin script environment. Instead, it must be accessed and updated via the `jsr223Bindings` object as follows.

```
engine %@% 'jsr223Bindings.put("myValue", 5)'
engine %~% 'jsr223Bindings.get("myValue")'
```

```
## [1] 5
```

Kotlin documentation demonstrates managing bindings through an object named `bindings`. However, the `bindings` object is read-only as of this writing. This is a reported bug. The accepted workaround is to use `jsr223Bindings`.

In Callbacks, we explain how a global `R` object is added to the script engine environment to enable callbacks into the R environment. This `R` object is necessarily present in `jsr223Bindings`, but we do not recommend accessing it from that structure. Instead, use the global `R` variable as demonstrated in the code here.

```
# jsr223 automatically creates a variable R in the global scope of the Kotlin
# environment to facilitate callbacks.
engine %@% 'R.set("c", 4)'

# The R object in 'jsr223Bindings' is inconvenient to use because it must be
# cast to an explicit type.
engine %@% '(jsr223Bindings["R"] as org.fgilbert.jsr223.RClient).set("c", 3)'
```

10.2 Kotlin and Java classes

Kotlin is designed to be interoperable with Java. This example uses the Apache Commons Mathematics Library to sample from a bivariate normal distribution.

```
library("jsr223")

# Change this path to the installation directory of the Kotlin compiler.
kotlin.directory <- Sys.getenv("KOTLIN_HOME")

# Include both the Kotlin script engine jars and the Apache Commons Mathematics
# libraries in the class path.
engine <- ScriptEngine$new(
  engine.name = "kotlin"
  , class.path = c(
    getKotlinScriptEngineJars(kotlin.directory),
    "~/my-path/commons-math3-3.6.1.jar"
  )
)

# Define the means vector and covariance matrix that will be used to create the
```

```
# bivariate normal distribution.
engine$means <- c(0, 2)
engine$covariances <- diag(1, nrow = 2)

# Import the package member and instantiate a new class.
engine %@% '
import org.apache.commons.math3.distribution.MultivariateNormalDistribution
val mvn = MultivariateNormalDistribution(
  jsr223Bindings["means"] as DoubleArray,
  jsr223Bindings["covariances"] as Array<DoubleArray>
)
'

# This line is a workaround for a Kotlin bug involving `invokeMethod`.
# https://github.com/floidgilbert/jsr223/issues/1
engine %@% 'jsr223Bindings["mvn"] = mvn'

# Take a multivariate sample.
engine$invokeMethod("mvn", "sample")

## [1] -2.286145  2.016230

# Take three samples.
replicate(3, engine$invokeMethod("mvn", "sample"))

##           [,1]      [,2]      [,3]
## [1,] 0.9924368 -1.295875 0.2025815
## [2,] 2.5145855  2.128243 1.1666272

# Terminate the script engine.
engine$terminate()
```

11 Software review

There are many integrations that combine the strengths of R with other programming languages. These language integrations can generally be classified as either *R-major* or *R-minor*. R-major integrations use R as the primary environment to control some other embedded language environment. R-minor integrations are the inverse of R-major integrations. For example, **rJava** is an R-major integration that allows Java objects to be used within an R session. The Java/R Interface (**JRI**), in contrast, is an R-minor integration that enables Java applications to embed R.

The **jsr223** package provides an R-major integration for the Java platform and the **jsr223** programming languages. In this software review, we provide context for the **jsr223** project through comparisons with other R-major integrations. Popular R-minor language integrations such as **Rserve** (Urbanek, 2016c) and **OpenCPU** (Ooms, 2013) are not included in this discussion because their objectives and features do not necessarily align with those of **jsr223**. We do, however, include a brief discussion of R language implementations for the JVM.

Before we compare **jsr223** to other R packages, we point out one unique feature that contrasts **jsr223** with all other integrations in this discussion: **jsr223** is the only package that provides a standard interface to integrate R with multiple programming languages. This key feature enables developers to take advantage of solutions and features in several languages without the need to learn multiple integration packages.

Our software review does not include integrations for Ruby and Kotlin because **jsr223** is the only R-major integration for those languages on CRAN.

11.1 rJava software review

As noted in the introduction, **rJava** is the preeminent Java integration for R. It provides a low-level interface to compiled Java classes via the JNI. The **jsr223** package uses **rJava** together with the Java Scripting API to create a simple, multi-language integration for R and the Java platform.

The following code example is taken from **rJava**'s web site (Urbanek, 2016b). It demonstrates the essential functions of the **rJava** API with code to create and display a window with a single button. The first two lines are required to initialize **rJava**. The next lines use the `.jnew` function to create two Java objects: a GUI frame and a button. The associated class names are denoted in JNI syntax. Of particular note is the first invocation of `.jcall`, the function used to call object methods. In this case, the `add` method of the frame object is invoked. For **rJava** to identify the appropriate method, an explicit return type must be specified in JNI notation as the second parameter to `.jcall` (unless the return value is `void`). The last parameter to `.jcall` specifies the object to be added to the frame object. It must be explicitly cast to the correct interface for the call to be successful.

```
library("rJava")
.jinit()
f <- .jnew("java/awt/Frame", "Hello")
b <- .jnew("java/awt/Button", "OK")
.jcall(f, "Ljava/awt/Component;", "add", .jcast(b, "java/awt/Component"))
.jcall(f, , "pack")
# Show the window.
.jcall(f, , "setVisible", TRUE)
# Close the window.
.jcall(f, , "dispose")
```

The snippet below reproduces the **rJava** example using JavaScript. In comparison, the JavaScript code is more natural for most programmers to write and maintain. The fine details of method lookups and invocation are handled automatically: no explicit class names or type casts are required. This same example can be reproduced in any of the five other **jsr223**-supported programming languages.

```
var f = new java.awt.Frame('Hello');
f.add(new java.awt.Button('OK'));
f.pack();
// Show the window.
f.setVisible(true);
// Close the window.
f.dispose();
```

Using **jsr223**, the preceding code snippet can be embedded in an R script. The first step is to create an instance of a script engine. A JavaScript engine is created as follows:

```
library(jsr223)
engine <- ScriptEngine$new("JavaScript")
```

This engine object is now ready to evaluate script on demand. Source code can be passed to the engine using character vectors or files. The sample below demonstrates embedding JavaScript code in-line with character vectors. This method is appropriate for small snippets of code. (Note: If you try this example the window may appear in the background. Also, the window must be closed using the last line of code. These are limitations of the code example, not **jsr223**.)

```
# Execute code inline to create and show the window.
engine %@% "
  var f = new java.awt.Frame('Hello');
  f.add(new java.awt.Button('OK'));
  f.pack();
  f.setVisible(true);
"


# Close the window
engine %@% "f.dispose();"
```

To execute source code in a file, use the script engine object's `source` method: `engine$source(file.name)`. The variable `file.name` may specify a local file path or a URL. Whether evaluating small code snippets or sourcing script files, embedding source code using **jsr223** is straightforward. The ability to embed source code in R facilitates prototyping and promotes rapid application development.

In comparison to **rJava**'s low-level interface, **jsr223** allows developers to use Java objects without knowing the details of JNI and method lookups. However, it is important to note that **rJava** does include a high-level interface for invoking object methods. It uses the Java reflection API to automatically locate the correct method signature. This is an impressive feature, but according to the **rJava** web site, its high-level interface is experimental and does not work correctly for all scenarios. There are no such limitations for the **jsr223** scripting languages.

The **jsr223** languages also feature support for advanced object-oriented constructs. For example, classes can be extended and interfaces can be implemented using any language. These features allow developers to quickly implement sophisticated solutions in R without developing, compiling, and distributing custom Java classes.

The **rJava** package supports exchanging scalars, arrays, and matrices between R and Java. The following R code demonstrates converting an R matrix to a Java object, and vice versa, using **rJava**.

```
a <- matrix(rnorm(10), 5, 2)
# Copy matrix to a Java object with rJava
o <- .jarray(a, dispatch = TRUE)
# Convert it back to an R matrix.
b <- .jevalArray(o, simplify = TRUE)
```

Again, the **jsr223** package builds on **rJava** functionality by extending data exchange. Our package converts R vectors, factors, n-dimensional arrays, data frames, lists, and environments to generic Java objects.[2] In addition, **jsr223** can convert Java scalars, n-dimensional arrays, maps, and collections to base R objects. Several data exchange options are available, including row-major and column-major ordering schemes for data frames and n-dimensional arrays.

This code snippet demonstrates data exchange using **jsr223**. The variable `engine` is a **jsr223** ScriptEngine object. Similar to the preceding **rJava** example, this code copies a matrix to the Java environment and back again. The same syntax is used for all supported data types and structures.

```
a <- matrix(rnorm(10), 5, 2)
# Copy an R object to Java using jsr223.
engine$a <- a
# Retrieve the object.
engine$a
```

The **rJava** package does not directly support callbacks into R. Instead, callbacks are implemented through **JRI**: the Java/R Interface. To use **JRI**, R must be compiled with the shared library option '`--enable-R-shlib`'. The **JRI** interface is technical and extensive. In contrast, **jsr223** supports callbacks into R using a lightweight interface that provides just three methods to execute R code, set variable values, and retrieve variable values. The **jsr223** package does not use **JRI**, so there is no requirement for R to be compiled as a shared library.

In conclusion, **jsr223** provides an alternative integration for the Java platform that is easy to learn and use.

11.2 Groovy integrations software review

The only other Groovy language integration on CRAN is **rGroovy**. It is a simple integration that uses **rJava** to instantiate `groovy.lang.GroovyShell` and pass code snippets to its `evaluate` method. We outline the typical integration approach using **rGroovy**.

To create a Groovy script engine, the necessary Groovy JAR files must be specified in a class path. Furthermore, if a solution requires additional external Java libraries, these must also be included in the class path. Using **rGroovy**, the class path must be specified as an unnamed list in the global R option `GROOVY_JARS` *before* loading the **rGroovy** package. The class path cannot be modified after the package is loaded.

```
options(GROOVY_JARS = list("~/my-path/groovy-all.jar", ...))
library("rGroovy")
```

After the package is loaded, the `Initialize` function is called to instantiate an instance of the Groovy script engine that will be used to handle script evaluation. The `Initialize` function has one optional argument named `binding`. This argument accepts an **rJava** object reference to a `groovy.lang.Binding` object that represents the bindings available to Groovy script engine. Hence, **rJava** must be used to create, set, and retrieve values from this object. The following code example demonstrates initializing the Groovy script engine. We

2 **rJava**'s interface can theoretically support n-dimensional arrays, but currently the feature does not produce correct results for n > 2. See the related issue at the **rJava** Github repository: ".jarray(..., dispatch=T) on multi-dimensional arrays creates Java objects with wrong content."

initialize the script engine bindings with a variable named `myValue` that contains a vector of integers. Notice that knowledge of **rJava** and JNI notation is required to create an instance of the `bindings` object, convert the vector to a Java array, cast the resulting Java array to the appropriate interface, and finally, call the `setVariable` method of the `bindings` object.

```
bindings <- rJava::.jnew("groovy/lang/Binding")
Initialize(bindings)
myValue <- rJava::.jarray(1:3)
myValue <- rJava::.jcast(myValue, "java/lang/Object")
rJava::.jcall(bindings, "V", method = "setVariable", "myValue", myValue)
```

Finally, Groovy code can be executed using the `Evaluate` method. The evaluate method returns the value of the last statement, if any. In this example, we modify the last element of our `myValue` array, and return the contents of the array.

```
script <- "
  myValue[2] = 5;
  myValue;
"
Evaluate(groovyScript = script)

## [1] 1 2 5
```

The **rGroovy** package includes another function, `Execute`, that allows developers to evaluate Groovy code without using **rJava**. However, this interface creates a new Groovy script engine instance each time it is called. In other words, it does not allow the developer to preserve state between each script evaluation.

In this code example, we show Groovy integration with **jsr223**. After the library is loaded, an instance of a Groovy script engine is created. The class path is defined at the same time a script engine is created. The variable `engine` represents the script engine instance; it exposes several methods and properties that control data exchange behavior and code evaluation. The third line creates a binding named `myValue` in the script engine's environment; the R vector is automatically converted to a Java array. The fourth line executes Groovy code that changes the last element of the `myValue` Java array before returning it to the R environment.

```
library("jsr223")
engine <- ScriptEngine$new("groovy", "~/my-path/groovy-all.jar")
engine$myValue <- 1:3
engine %~% "
  myValue[2] = 5;
  myValue;
"

## [1] 1 2 5
```

In comparison to **rGroovy**, the **jsr223** implementation is more concise and requires no knowledge of **rJava** or required Java classes. Though not illustrated in this example, **jsr223** can invoke Groovy functions and methods from within R, it supports callbacks from Groovy into R, and it provides extensive and configurable data exchange between Groovy and R. These features are not available in **rGroovy**.

In summary, **rGroovy** exposes a simple interface for executing Groovy code and returning a result. Data exchange is primarily handled through **rJava**, and therefore requires knowledge of **rJava** and JNI. The **jsr223** integration is more comprehensive and does not require any knowledge of **rJava**.

11.3 JavaScript integrations software review

The most prominent JavaScript integration for R is Jeroen Ooms' **V8** package. It uses the open source V8 JavaScript engine featured in Google's Chrome browser. We discuss the three primary differences between **V8** and **jsr223**.

First, the JavaScript engine included with **V8** provides only essential ECMAscript functionality. For example, **V8** does not include even basic file and network operations. In contrast, **jsr223** provides access to the entire JVM which includes a vast array of libraries and computing functionality.

Second, all data exchanged between **V8** and R is serialized using JSON via the **jsonlite** package. JSON is very flexible; it can represent virtually any data structure. However, JSON converts all values to/from string representations which adds overhead and imposes round-off error for floating point values. The **jsr223** package handles all data using native values which reduces overhead and preserves maximum precision. In many applications, the loss of precision is not critical as far as the final numeric results are concerned, but it does require defensive programming when checking the equality of values. For example, an application using **V8** must round two values to a given decimal place before checking if they are equal. The following code example demonstrates the precision issue using the R constant `pi`. The JSON conversion is handled via **jsonlite**, just as in the **V8** package. We see that after JSON conversion the value of `pi` is not identical to the original value. In the **jsr223** code, the result is identical to the original value.

```
# 'digits = NA' requests maximum precision.
library("jsonlite")
identical(pi, fromJSON(toJSON(pi, digits = NA)))
```

```
## [1] FALSE
```

```
library("jsonlite")
engine <- ScriptEngine$new("js")
engine$pi <- pi
identical(engine$pi, pi)
```

```
## [1] TRUE
```

The third significant difference between **V8** and **jsr223** is code validation. **V8** includes an interface to validate JavaScript code. The Java Scripting API does not provide an interface for code validation, hence, **jsr223** does not provide this feature. We have investigated other avenues to validate code, but none are uniformly reliable across all of the **jsr223** supported languages. Moreover, this feature is generally not critical for most integration applications; code validation is most useful for interactive code entry.

11.4 Python integrations software review

In this section, we compare **jsr223** with two Python integrations for R: **reticulate** and **rJython**. Of the many Python integrations available for R on CRAN, **reticulate** is the most popular as measured by monthly downloads.[3] We also discuss **rJython** because, like **jsr223**, it targets Python on the JVM.

The **reticulate** package is a very thorough Python integration for R. It includes some refined interface features that are not available in **jsr223**. For example, **reticulate** enables Python objects to be manipulated in R script using list-like syntax. One major **jsr223** feature that **reticulate** does not support is callbacks (i.e., calling R from Python). Though there are many interface differences between **jsr223** and **reticulate** (too many to list here), the most practical difference arises from their respective Python implementations. The **reticulate** package targets CPython, the reference implementation of the Python script engine. As such, **reticulate** can take advantage of the many Python libraries compiled to machine code such as Pandas. The **jsr223** package targets the JVM via Jython, and therefore supports accessing Java objects from Python script. It cannot, however, access the Python libraries compiled to machine code because they cannot be executed by the JVM. One project, JyNI (the Jython Native Interface), is working to make the important extensions NumPy and SciPy available to the JVM.

The **rJython** package is similar to **jsr223** in that it employs Jython. Both **jsr223** and **rJython** can execute arbitrary Python code, call Python functions and methods directly from R, use Java objects, and copy data between environments. However, there are several important differences between the packages. They are detailed in the list below.

- Data exchange for **rJython** can be handled via JSON or direct calls to the Jython interpreter object via **rJava**. **jsr223** uses a single, unified approach to data conversion via custom data conversion routines.
- When using **rJava**, **rJython** data exchange is essentially limited to vectors and matrices. **jsr223** includes data exchange capabilities for all major R data structures.
- When using JSON, **rJython** converts R objects to generic Python structures. **jsr223** converts R objects to generic Java structures.
- **rJython**'s JSON features are provided by the **rjson** package on the R side. As we describe in JavaScript integrations, JSON implementations incur overhead and loss of numeric precision. Furthermore, **rjson** does not handle some R structures very well. For example, n-dimensional arrays and unnamed lists are both converted to one-dimensional JSON arrays. Lastly, **rJython** converts data frames to Python dictionaries, but Python dictionaries are always returned to R as named lists. **jsr223** exhibits none of these limitations.
- **rJython** does not provide configurable data exchange options. **jsr223** provides several options for converting factors, data frames, and n-dimensional arrays.
- **rJython** does not return the value of the last expression when executing Python code. Instead, scripts must assign a value to a global Python variable which can be fetched from R using another **rJython** method. This does not promote fast code exploration and prototyping.
- **rJython** does not support callbacks.

3 The **reticulate** package has 3,681 downloads per month according to http://rdocumentation.org. The next most popular Python integration is **PythonInR** with 322 monthly downloads.

- **rJython** does not supply interfaces for script compiling or capturing console output.

In all, **jsr223** provides a more complete integration for Jython than **rJython**.

11.5 Renjin and FastR software review

12 Conclusion

///

13 Package version history

///

14 Document version history

///

References

Kotlin 1.0 released: Pragmatic language for jvm and android. URL https://blog.jetbrains.com/kotlin/2016/02/kotlin-1-0-released-pragmatic-language-for-jvm-and-android/.

D. B. Dahl. *rscala: Bi-Directional Interface Between R and Scala with Callbacks*, 2016. URL http://CRAN.R-project.org/package=rscala. R package version 1.0.6.

Google developers. *Chrome V8*, 2016. URL https://developers.google.com/v8/.

JRuby team. *JRuby: The Ruby Programming Language on the JVM*, 2016. URL http://jruby.org/.

Jython developers. *Jython: Python for the Java Platform*, 2016. URL http://www.jython.org/.

Kotlin developers. *Kotlin: Kotlin Programming Language*, 2018. URL http://www.jython.org/.

Y. Matsumoto. *Ruby 1.9*, 2008. URL https://www.youtube.com/watch?v=oEkJvvGEtB4.

Y. Matsumoto. *Ruby Programming Language*, 2016. URL http://www.ruby-lang.org.

Mozilla Developer Network. *Rhino*, 2016. URL https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino.

M. O'Connell. Java: The inside story. *SunWorld Online*, 1995. URL http://tech-insider.org/java/research/1995/07.html.

J. Ooms. *OpenCPU*, 2013. URL https://www.opencpu.org/.

J. Ooms. The jsonlite package: A practical and consistent mapping between json data and r objects. *arXiv:1403.2805 [stat.CO]*, 2014. URL https://arxiv.org/abs/1403.2805.

J. Ooms. *V8: Embedded JavaScript Engine for R*, 2016. URL https://CRAN.R-project.org/package=V8. R package version 1.2.

Oracle. *Java Scripting API*, 2016a. URL https://docs.oracle.com/javase/8/docs/technotes/guides/scripting/prog_guide/toc.html.

Oracle. *Java Platform, Standard Edition Nashorn User's Guide, Release 8*, 2016b. URL https://docs.oracle.com/javase/8/docs/technotes/guides/scripting/nashorn/.

Python developers. *Python*, 2016. URL https://www.python.org/.

Scala developers. *The Scala Programming Language*, 2016. URL https://www.scala-lang.org/.

J. Strachan. *Groovy*, 2016. URL http://www.groovy-lang.org.

Sun Microsystems, Inc. *JSR-223: Scripting for the Java Platform*, 2006. URL https://jcp.org/en/jsr/detail?id=223.

S. Urbanek. *rJava: Low-Level R to Java Interface*, 2016a. URL http://CRAN.R-project.org/package=rJava. R package version 0.9-8.

S. Urbanek. *rJava: Low-Level R to Java Interface*, 2016b. URL http://www.rforge.net/rJava/.

S. Urbanek. *Rserve: Binary R server*, 2016c. URL http://CRAN.R-project.org/package=Rserve. R package version 0.6-8.1.