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

by Floid R. Gilbert and David B. Dahl

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

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 did not 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** (2017), 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. A second advantage to 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 provide a high-level integration interface that makes Java libraries easier to use from within R. We accomplish this by embedding other programming languages in R that 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.

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 (vectors, n-dimensional arrays, data frames, or lists). The **jsr223** package also converts the most common data structures from Groovy, JavaScript, JRuby, Jython, and Kotlin. 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 loss of floating point precision inherent in JSON data conversion.

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 in R.

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

Document organization

The section "Typical use cases" highlights the **jsr223** package's core functionality. The section "**jsr223** package implementation" contains a brief overview of the package internals. Finally, the "Software review" section puts the **jsr223** project in context with comparisons to other relevant software solutions.

Typical use cases

This section includes introductory examples that demonstrate typical use cases for the **jsr223** package. See the **jsr223** *User Manual* included in the package vignettes for a comprehensive treatment of **jsr223** features and its supported languages.

Using Java libraries

For this demonstration, we use Stanford's Core Natural Language Processing Java libraries (Manning et al., 2014) to identify grammatical parts of speech in a text. Natural language processing (NLP) is a key component in statistical text analysis and artificial intelligence. This example shows how so-called "glue" code can be embedded in R to quickly leverage the Stanford NLP libraries. It also demonstrates how easily **jsr223** converts Java data structures to R objects. The full script is available at `https://github.com/floidgilbert/jsr223/tree/master/examples/JavaScript/stanford-nlp.R`.

The first step: create a **jsr223** `"ScriptEngine"` instance that can dynamically execute source code. In this case, we use a JavaScript engine. The object is created using the `ScriptEngine$new` constructor method. This method takes two arguments: a scripting language's 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 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 do so. In this example, we use the `%@%` operator; it executes a code snippet and discards the return value, if any. The code snippet imports the Stanford NLP `"Document"` class. This syntax is peculiar to the JavaScript dialect. The result, `DocumentClass`, is used to instantiate objects or access static methods.

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

The next code sample defines a JavaScript function named `getPartsOfSpeech`. It tags each element in a text with a grammatical part of speech (e.g., noun, adjective, or verb). The function parses the text using a new instance of the `"Document"` class. The parsing results are transferred to a list of JavaScript objects. Each JavaScript object contains the parsing information for a single sentence.

```
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;
  }
'
```

We use `engine$invokeFunction` to call the JavaScript function `getPartsOfSpeech` from R. The method `invokeFunction` takes the name of the function as the first parameter; any other arguments that follow are automatically converted to Java objects and passed to the JavaScript function. The function's return value is converted to an R object. In this case, **jsr223** intuitively converts the list of JavaScript objects to a list of R data frames as seen in the output below. The parts of speech abbreviations are defined by the Penn Treebank Project (Taylor et al., 2003). A quick reference is available at https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html.

```
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
```

In this example, we effectively used Stanford's Core NLP library with a minimal amount of code. This same functionality can be replicated in any of the **jsr223**-supported programming languages.

Extending existing Java solutions

Many Java libraries are designed to let the developer define custom behaviors by implementing interfaces or extending classes. We illustrate one such solution here in the context of a Bayesian analysis. The library at hand implements a multi-threaded Metropolis sampler (Metropolis et al., 1953). We define a density function for the sampler by extending (i.e., subclassing) an abstract Java class.

We use the Groovy scripting language for this example. Groovy follows the Java programming language syntax very closely; hence, it is a natural choice for Java integrations. Programmers that know Groovy will notice that our code is unnecessarily verbose. We chose to use strict Java coding conventions to make the implementation more familiar to Java programmers.

The Bayesian analysis involves count data $y_1, \ldots, y_n$ that we believe to be independently and identically distributed according to a zero-inflated Poisson sampling model. Given $0 < \pi < 1$ and $\lambda > 0$,

$$\Pr(y_i = 0 \mid \pi, \lambda) = \pi + (1 - \pi)\, e^{-\lambda}$$
$$\Pr(y_i = k \mid \pi, \lambda) = (1 - \pi)\frac{\lambda^k\, e^{-\lambda}}{k!}, \quad \text{for } k = 1, 2, \ldots$$

We choose independent priors $\pi \sim \text{Beta}(\alpha, \beta)$ and $\lambda \sim \text{Gamma}(\theta, \kappa)$. Furthermore, we use independent Gaussian proposal densities for π and λ.

Our analysis involves two scripts: (i) a Groovy script to extend and execute the Metropolis sampler class; and (ii) an R script to prepare the data and parameters, execute the Groovy script, and summarize the results. The scripts, named 'metropolis.groovy' and 'metropolis.R', are located at https://github.com/floidgilbert/jsr223/tree/master/examples/Groovy. The required Java files can be downloaded from the 'lib' subfolder.

The Groovy script To begin, we import the necessary classes. The first line below imports all of the static methods of the "Math" class. The second line imports the abstract class for the Metropolis sampler. The last line imports a univariate normal proposal class. (This latter class implements the interface "ProposalDistributionUnivariate". If we wanted to use a custom proposal distribution, we could do so by creating a class that implements this interface in script.)

```
import static java.lang.Math.*;
import org.fgilbert.jsr223.examples.MetropolisSamplerUnivariateProposal;
import org.fgilbert.jsr223.examples.ProposalDistributionUnivariateNormal;
```

The code that follows is the key element of this example; it defines the behavior of a Java class in script. Specifically, we define a class named "Sampler" that extends the abstract class "MetropolisSamplerUnivariateProposal". The "Sampler" class has just two members: the constructor method and the logPosterior method. The constructor method takes the parameter values for the prior distributions and computes statistics used in the posterior function. The logPosterior method implements the log of the posterior function that is called by the sampler.

```
public class Sampler extends MetropolisSamplerUnivariateProposal {
  private double alpha, beta, theta, kappa;
  private double dataLength, dataSum, dataZeroCount, dataPositiveCount;

  public Sampler(double alpha, double beta, double theta, double kappa, int[] data) {
    this.alpha = alpha; this.beta = beta;
    this.theta = theta; this.kappa = kappa;

    dataLength = data.length;
    for (int i = 0; i < dataLength; i++) {
      dataSum += data[i];
      if (data[i] == 0)
        dataZeroCount++;
    }
    dataPositiveCount = dataLength - dataZeroCount;
  }

  @Override
  public double logPosterior(double[] values) {
    double pi = values[0];
    double lambda = values[1];
    if (pi <= 0 || pi >= 1 || lambda < 0)
      return Double.NEGATIVE_INFINITY;
    return (alpha - 1) * log(pi) + (beta - 1) * log(1 - pi) +
      (theta - 1) * log(lambda) - kappa * lambda +
      dataZeroCount * log(pi + (1 - pi) * exp(-lambda)) +
      dataPositiveCount * log((1 - pi) * exp(-lambda)) +
      dataSum * log(lambda);
  }
}
```

The next lines initialize an array of normal proposal distribution objects that will be used by the sampler. Each distribution object is initialized with a variance. The proposalVariances array is not defined here; it is supplied by the R script.

```
ProposalDistributionUnivariateNormal[] pd =
  new ProposalDistributionUnivariateNormal[proposalVariances.length];
for (int i = 0; i < proposalVariances.length; i++)
  pd[i]        = new ProposalDistributionUnivariateNormal(proposalVariances[i]);
```

Finally, we create a "Sampler" instance and assign it to the variable sampler. The last line runs the sampler. Because it is the last line in the script, its return value will be automatically returned to the R environment. Notice that all but one of the variables passed as method arguments have not been defined yet. They will be provided by the R script.

```
Sampler sampler = new Sampler(alpha, beta, theta, kappa, data);
sampler.sample(startingValues, pd, iterations, discard, threads);
```

The R script First, we instantiate a Groovy script engine. The paths to the required Java libraries are defined in `class.path`. The first file is the Groovy script engine; the second file contains the Metropolis sampler; and the last file is the Apache Commons Mathematics Library (http://commons.apache.org/proper/commons-math).

```
library("jsr223")

class.path <- c(
  "lib/groovy-all-2.4.7.jar",
  "lib/org.fgilbert.jsr223.examples-0.3.0.jar",
  "lib/commons-math3-3.6.1.jar"
)
engine <- ScriptEngine$new("Groovy", class.path)
```

The matrix `starting.values` defined here contains initial values for the Metropolis sampler. Each row is a (π, λ) value pair that will be used to initialize a random walk. Hence, we will have four MCMC chains for each parameter. The multi-threaded sampler computes these chains in parallel.

```
starting.values <- rbind(
  c(0.999, 0.001),
  c(0.001, 0.001),
  c(0.001, 30),
  c(0.999, 30)
)
```

In the next step, we initialize global variables that are used by the Groovy script. The **jsr223** package provides a few ways to do this, but the most convenient approach is the list-like assignment syntax shown below. The first four assignments (alpha, beta, theta, and kappa) correspond to the parameter values for the prior densities; the variable `data` contains the counts $y_1, \ldots, y_n$; `proposalVariances` is an array of variances for the Gaussian proposal distributions; `startingValues` contains the initial (π, λ) parameter values for four random walks; `iterations` indicates the number of MCMC iterations per random walk; `discard` is the number of initial draws to ignore; and `threads` defines the size of the parallel processing thread pool.

```
engine$alpha <- 1
engine$beta <- 1
engine$theta <- 2
engine$kappa <- 1
engine$data <- as.integer(c(rep(0, 25), rep(1, 6), rep(2, 4), rep(3, 3), 5))
engine$proposalVariances <- c(0.3^2, 1.2^2)
engine$startingValues <- starting.values
engine$iterations <- 10000L
engine$discard <- as.integer(engine$iterations * 0.20)
engine$threads <- parallel::detectCores()
```

The Metropolis sampler will return two multi-dimensional arrays. We prefer the arrays to be structured in a specific order, so we change the default ordering using the code here. For the sake of brevity, we refer the reader to the **jsr223** *User Manual* for details regarding array order settings.

```
engine$setArrayOrder("column-minor")
```

Next, we compile and execute the Groovy script. Compiling the script is optional; we could evaluate the code in one step using the **jsr223** `source` method. However, we intend to execute the script more than once, so we compile it for efficiency. The first line below compiles the script and assigns the resulting object to the variable `cs`. The second line executes the compiled code and assigns the output to the variable `r`. The result is the return value of the last line in the script. In our Groovy script, the last line is a call to the method `sample` of the `"Sampler"` class.

```
cs <- engine$compileSource("metropolis.groovy")
r <- cs$eval()
```

The `sample` method returns a Java map (i.e., an associative array or dictionary) with two members: `"acceptance_rates"` and `"chains"`. The **jsr223** package automatically converts the map and its contents to an R named list with the same member names. Define k as the number of iterations minus the number discarded, $p = 2$ as the number of parameters, and $w = 4$ as the number of random walks. Then the `"chains"` member is a 3-dimensional $k \times p \times w$ matrix containing the MCMC results. Here, we output the dimensions of the 3-dimensional array and the top six rows of first random walk.

```
dim(r$chains)

## [1] 8000    2    4

parameter.names <- c("pi", "lambda")
dimnames(r$chains) <- list(NULL, parameter.names, NULL)
head(r$chains[, , 1])

##             pi   lambda
## [1,] 0.5225463 1.647577
## [2,] 0.4613551 1.647577
## [3,] 0.4613551 1.647577
## [4,] 0.6012411 1.647577
## [5,] 0.6012411 1.647577
## [6,] 0.4965568 1.647577
```

The "acceptance_rates" member is a $w \times p$ matrix containing the acceptance rates for each parameter and random walk. We output those values below.

```
colnames(r$acceptance_rates) <- parameter.names
r$acceptance_rates

##          pi lambda
## [1,] 0.3695 0.3143
## [2,] 0.3655 0.3209
## [3,] 0.3638 0.3175
## [4,] 0.3708 0.3148
```

For the sake of demonstration, let's say that the acceptance rates are too high. We need to widen the variances for the proposal distributions and re-run the sampler. There is no need to recompile the script; instead, just update the corresponding global variable and execute the compiled script again. We output the acceptance rates here for comparison.

```
engine$proposalVariances <- c(0.5^2, 1.7^2)
r <- cs$eval()
colnames(r$acceptance_rates) <- parameter.names
r$acceptance_rates

##          pi lambda
## [1,] 0.2361 0.2377
## [2,] 0.2354 0.2383
## [3,] 0.2340 0.2305
## [4,] 0.2418 0.2304
```

Table 1 contains a summary for each parameter by individual chain. The code used to summarize the results can be found in the R script 'metropolis.R' on GitHub.

Performance So far we have shown that we can extend compiled Java classes from within R. But, is the runtime performance acceptable? We report some performance metrics here. The scripts were run on a typical notebook computer with an Intel i7-5500U, 2.40Ghz processor and 8GB RAM. The processor can execute four threads in parallel. We allocated all four threads to the Metropolis sampler to execute four walks in parallel. We ran the simulation for 10,000, 100,000, and 1,000,000 MCMC iterations per random walk. No values were discarded. Timings were recorded using the **microbenchmark** package (Mersmann, 2018). We report the mean run time over 40 simulations in Table 2. The first column contains the number of MCMC iterations per random walk. The second column contains the mean run times for the expression `cs$eval()` as in our preceding example. The third column contains the mean run times for the expression `cs$eval(discard.return.value = TRUE)`. This expression executes the Groovy script but discards the results instead of converting them to R objects. The last column contains the difference of columns two and three; hence, it roughly represents the time required to convert the Java results to R. The number of values returned to R for each simulation is $p \times w + iterations \times p \times w$. For example, when one million MCMC iterations are requested, a total of 8,000,008 numeric values are returned. The size of the resulting R object is about 61MB (see `object.size(r)`).

To put these results in context, we ran another simulation where the Metropolis abstract class is extended in a compiled Java class instead of Groovy. Otherwise, the simulation is identical to the

foregoing example. Table 3 summarizes those results. In the maximum case (one million MCMC iterations), the difference between the implementations is well under two seconds. We find this performance to be acceptable. Hence, the Groovy solution is a good balance of scripting convenience and compiled performance. These results may vary depending on the scripting language used.

What we have shown here is only one approach to this implementation. The total runtime could be reduced by using simpler data structures or summarizing the data on the Java side. However, R is designed for summarizing data. Therefore, we chose to transfer the full results to R in a convenient format because we believe it represents the most typical use case. Furthermore, we could have implemented this solution using anonymous classes, lambda functions, or closures. These constructs usually require less code, and they are supported within the various **jsr223**-compatible programming languages. However, we found that they did not execute as quickly.

Table 1: A summary of the MCMC chains generated by the Metropolis sampler. Each parameter and chain is listed with quantiles, acceptance rate, and effective sample size (ESS).

Parameter	Chain	2.5%	25%	50%	75%	97.5%	Acc. Rate	ESS
π	1	0.288	0.460	0.537	0.606	0.724	0.236	764
π	2	0.288	0.467	0.540	0.604	0.724	0.235	864
π	3	0.298	0.459	0.529	0.603	0.722	0.234	989
π	4	0.268	0.456	0.528	0.598	0.713	0.242	702
λ	1	0.933	1.313	1.563	1.814	2.377	0.238	1023
λ	2	0.968	1.324	1.563	1.839	2.412	0.238	805
λ	3	0.935	1.356	1.579	1.816	2.344	0.231	785
λ	4	0.875	1.303	1.532	1.803	2.364	0.230	813

Table 2: Benchmark timings for the Java Metropolis sampler extended in Groovy. All times are in milliseconds. The first column indicates the number of MCMC iterations computed for each of the four random walks run in parallel. The second column contains the mean runtime for the expression `cs$eval()` over 40 simulations. The third column is the mean runtime for `cs$eval(TRUE)` (i.e., the return value is discarded). The last column is the difference between columns two and three; hence, it roughly represents the time required to convert the Java results to R objects.

	With Return Values		
Iterations	Yes	No	Difference
10,000	43.6	28.5	15.2
100,000	426.2	294.0	132.2
1,000,000	6,004.1	3,783.7	2,220.4

Table 3: Benchmark timings for the Metropolis sampler extended in Java. All times are in milliseconds. The columns are as in Table 2.

	With Return Values		
Iterations	Yes	No	Difference
10,000	33.6	17.9	15.8
100,000	293.2	171.1	122.2
1,000,000	5,348.2	2,391.5	2,956.7

Table 4: Performance comparison for the Metropolis samplers. The values are the mean run-times over 40 iterations reported in milliseconds.

Iterations	R	Groovy Class	Java Class
10,000	1,185.3	43.6	33.6
100,000	4,174.6	426.2	293.2
1,000,000	34,202.5	6,004.1	5,348.2

Finally, how does the performance compare to a base R implementation? We created a parallel Metropolis sampler in R that is very similar to the Java sampler (see 'metropolis-base-r.R' on GitHub). Table 4 reports the mean run times over 40 simulations. For comparison, the table includes the timings for the implementations featuring the Groovy class and the Java class. For 10,000 iterations, the Groovy class implementation is 27.2 times faster than the R implementation. The Java class implementation is 35.3 times faster. Part of the performance difference can be attributed to how parallelization is implemented in Java vs. R. The Java sampler uses a thread pool whereas the R approach uses a process pool. The latter requires significantly more overhead. When we consider one million iterations, the difference in parallelization implementations becomes insignificant and the Groovy implementation is only 5.7 times faster than the R implementation. However, the reduction in performance ratio is due, in part, to the conversion of large Java objects to R objects. If we exclude the data conversion time we have a more direct comparison of code execution performance. Using the run times excluding data conversion in Table 2 and Table 3, we find that the Groovy implementation executes 9.0 times faster than the R implementation and the Java implementation executes 14.3 times faster. This comparison is not comprehensive; we report these numbers only to give the reader some basic expectation of performance benefits.

Conclusion This example illustrated how **jsr223** facilitates the development of advanced Java solutions. Java interfaces can be implemented and classes extended within script promoting rapid application development and quick execution times.

Using other language libraries

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 common JavaScript libraries like Underscore (http://underscorejs.org) and Voca (https://vocajs.com) can be sourced using a URL. The following example sources Voca and word-wraps a string.

```
engine$source(
  "https://raw.githubusercontent.com/panzerdp/voca/master/dist/voca.min.js",
  discard.return.value = TRUE
)
engine$invokeMethod(
  "v",
  "wordWrap",
  "A long sentence to wrap using Voca methods.",
  list(width = 20)
)

## [1] "A long sentence to\nwrap using Voca\nmethods."
```

Compiled Groovy and Kotlin libraries are accessed in the same way as Java libraries: simply include the relevant class or JAR files when instantiating a script engine.

Ruby gems (i.e., libraries) can also be used with **jsr223**. The **jsr223** *User Manual* provides instructions and a code example that uses Ruby gems to generate fake entities for sample data sets.

Core Python language features are fully accessible via **jsr223**. The **jsr223** *User Manual* contains a code example that implements a simple HTTP server in Python. The Python server calls back to R to retrieve HTML content. Compatibility with many common Python libraries is limited on the Java platform. Please see "Python integrations software review" for more information.

The jsr223 package implementation

The **jsr223** package is built on **rJava**. The design of **jsr223** follows cues from **rJava**, **rscala** (Dahl, 2018), and **V8** (Ooms, 2017b).

The **jsr223** package uses the Java Scripting API (Oracle, 2016) 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 package.

The **jsr223** package supports most of the programming languages that have implemented JSR-223. Technically, any JSR-223 implementation will work with our package, 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 not complete. (Consider, instead, the **rscala** package for a Scala/R integration.) We also exclude languages that are no longer being actively developed, such as BeanShell.

Data exchange for **jsr223** is provided by the R package **jdx**: Java Data Exchange for R and **rJava** (Gilbert and Dahl, 2018). 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.

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).

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 several 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, 2013) and **opencpu** (Ooms, 2017a) 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 an R language implementation 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.

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 user-friendly, multi-language integration for R and the Java platform.

The following code example is taken from **rJava**'s web site `http://www.rforge.net/rJava`. It demonstrates the essential functions of the **rJava** API by way of creating and displaying a GUI 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 above 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.

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 much slower than the low-level interface and it does not work correctly for all scenarios.

The **jsr223**-compatible programming 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. This can speed development and deployment significantly.

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. The **JRI** interface is included with **rJava**. However, 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.

Groovy integrations software review

Besides **jsr223**, the only other Groovy language integration available on CRAN is **rGroovy** (Fuller, 2018). 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**.

Class paths must set in the global option `GROOVY_JARS` *before* loading the **rGroovy** package.

```
options(GROOVY_JARS = list("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 the Groovy script engine. Hence, **rJava** must be used to create, set, and retrieve values in the `bindings` object. The following code example demonstrates instantiating the Groovy script engine. We 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; it 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
```

[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."

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 demonstrate 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 the 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$new("Groovy", "groovy-all.jar")
engine$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 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**.

JavaScript integrations software review

The most prominent JavaScript integration for R is Jeroen Ooms' **V8** package (2017b). It uses the open source V8 JavaScript engine (Google developers, 2018) 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 (Ooms et al., 2017). 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 for equality. 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 contrast, the **jsr223** conversion result is identical to the original value.

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

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

```
library("jsr223")
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 not critical for most integration scenarios. Code validation is more common in applications that involve interactive code editing.

Python integrations software review

In this section, we compare **jsr223** with two Python integrations for R: **reticulate** (Allaire et al., 2018) and **rJython** (Grothendieck and Bellosta, 2012). 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 (McKinney, 2010). 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. This isn't a complete dead-end for Jython; many important Python extensions are being migrated to the JVM by the Jython Native Interface project (`http://www.jyni.org`). These extensions can easily be accessed through **jsr223**.

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 also several important differences.

Data exchange for **rJython** can be handled via JSON or direct calls to the Jython interpreter object via **rJava**. When using **rJava** for data exchange, **rJython** is essentially limited to vectors and matrices. When using JSON for data exchange, **rJython** converts R objects to Jython structures. In contrast, the **jsr223** supports a single data exchange interface that supports all major R data structures. It uses custom Java routines that avoid the overhead and roundoff error associated with JSON conversion. Finally, **jsr223** converts R objects to generic Java structures instead of Jython objects.

JSON data exchange for **rJython** is handled by the **rjson** (Couture-Beil, 2014) package. It does not handle some R structures as one would expect. For example, n-dimensional arrays and unnamed lists are both converted to one-dimensional JSON arrays. Furthermore, **rJython** converts data frames to Jython dictionaries, but dictionaries are always returned to R as named lists. The **jsr223** package does not exhibit these limitations; it provides predictable data exchange for all major R data structures.

Unlike **jsr223**, the **rJython** package does not return the value of the last expression when executing Python code. Instead, scripts must assign a value to a global Python variable to be fetched by another **rJython** method. This does not promote fast code exploration and prototyping. In addition, **rJython** does not supply interfaces for callbacks, script compiling, or capturing console output.

In essence, **rJython** implements a basic interface to the Jython language. The **jsr223** package, in comparison, provides a more developed feature set.

Renjin software review

Renjin (Renjin developers, 2018) is an ambitious project whose primary goal is to create a drop-in replacement for the R language on the Java platform. The Renjin solution features R syntax extensions that allow Java classes to created and used naturally within R script. The Renjin language implementation has two important limitations: (i) it does not support graphical methods; and (ii) it can't use R packages that contain native libraries (like C). The **jsr223** package, in contrast, is designed for the reference distribution of R. As such, it can be used in concert with any R package.

Renjin also distributes an R package called **renjin**. It is not available from CRAN. (Find the installation instructions at `http://www.renjin.org`.) The **renjin** package exports a single method that evaluates an R expression. It is designed only to improve execution performance for R expressions; it

[3] The **reticulate** package has 3,681 downloads per month according to `http://rdocumentation.org`. The next most popular Python integration is **PythonInR** (Schwendinger, 2018) with 322 monthly downloads.

does not allow Java classes to be used in R script. Hence, the **renjin** package is not a Java platform integration.

Overall, Renjin is a promising Java solution for R, but it is not yet feature-complete. In comparison, **jsr223** presents a viable Java solution for R today.

Summary

Java is one of the most successful development platforms in computing history. Its popularity continues as more programming languages, tools, and technologies target the JVM. The **jsr223** package provides a high-level, user-friendly interface that enables R developers to take advantage of the flourishing Java ecosystem. In addition, **jsr223**'s unified integration interface for Groovy, JavaScript, Python, Ruby, and Kotlin also facilitates access to solutions developed in these languages. In all, **jsr223** significantly extends the computing capabilities of the R software environment.

In this paper, we provided an introduction to the main features and advantages of the **jsr223** package. For more language-specific examples and a full treatment of software features, see the **jsr223** *User Manual* included in the package vignettes.

Bibliography

J. Allaire, Y. Tang, and M. Geelnard. *reticulate: Interface to 'Python'*, 2018. URL https://CRAN.R-project.org/package=reticulate. R package version 1.5. [p13]

A. Couture-Beil. *rjson: JSON for R*, 2014. URL https://CRAN.R-project.org/package=rjson. R package version 0.2.15. [p13]

D. B. Dahl. *rscala: Bi-Directional Interface Between R and Scala with Callbacks*, 2018. URL http://CRAN.R-project.org/package=rscala. R package version 2.5.1. [p8]

T. P. Fuller. *rGroovy: Groovy Language Integration*, 2018. URL https://CRAN.R-project.org/package=rGroovy. R package version 1.2. [p11]

F. R. Gilbert and D. B. Dahl. *jdx: 'Java' Data Exchange for 'R' and 'rJava'*, 2018. URL https://CRAN.R-project.org/package=jdx. R package version 0.1.2. [p9]

Google developers. *Chrome V8*, 2018. URL https://developers.google.com/v8/. [p12]

G. Grothendieck and C. J. G. Bellosta. *rJython: R interface to Python via Jython*, 2012. URL https://CRAN.R-project.org/package=rJython. R package version 0.0-4. [p13]

C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60, 2014. URL http://www.aclweb.org/anthology/P/P14/P14-5010. [p2]

W. McKinney. Data structures for statistical computing in python. In S. van der Walt and J. Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56, 2010. [p13]

O. Mersmann. *microbenchmark: Accurate Timing Functions*, 2018. URL https://CRAN.R-project.org/package=microbenchmark. R package version 1.4-4. [p6]

N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of State Calculations by Fast Computing Machines. *The Journal of Chemical Physics*, 21:1087–1092, June 1953. doi: 10.1063/1.1699114. [p3]

M. O'Connell. Java: The inside story. *SunWorld Online*, 1995. URL http://tech-insider.org/java/research/1995/07.html. [p1]

J. Ooms. *opencpu: Producing and Reproducing Results*, 2017a. URL https://CRAN.R-project.org/package=opencpu. R package version 2.0.5. [p9]

J. Ooms. *V8: Embedded JavaScript Engine for R*, 2017b. URL https://CRAN.R-project.org/package=V8. R package version 1.5. [p8, 12]

J. Ooms, D. T. Lang, and L. Hilaiel. *jsonlite: A Robust, High Performance JSON Parser and Generator for R*, 2017. URL https://CRAN.R-project.org/package=jsonlite. R package version 1.5. [p12]

Oracle. *Java Scripting API*, 2016. URL https://docs.oracle.com/javase/8/docs/technotes/guides/scripting/prog_guide/toc.html. [p8]

Renjin developers. *Renjin*, 2018. URL http://www.renjin.org. [p13]

F. Schwendinger. *PythonInR: Use 'Python' from Within 'R'*, 2018. URL https://CRAN.R-project.org/package=PythonInR. R package version 0.1-4. [p13]

Sun Microsystems, Inc. *JSR-223: Scripting for the Java Platform*, 2006. URL https://jcp.org/en/jsr/detail?id=223. [p8]

A. Taylor, M. Marcus, and B. Santorini. The penn treebank: An overview, 2003. [p3]

S. Urbanek. *Rserve: Binary R server*, 2013. URL http://CRAN.R-project.org/package=Rserve. R package version 1.7-3. [p9]

S. Urbanek. *rJava: Low-Level R to Java Interface*, 2017. URL https://CRAN.R-project.org/package=rJava. R package version 0.9-9. [p1]

Floid R. Gilbert
Master's Student
Department of Statistics
Brigham Young University
Provo, UT 84602
USA
floid.r.gilbert@gmail.com

David B. Dahl
Professor, Graduate Coordinator, and Associate Chair
Department of Statistics
Brigham Young University
Provo, UT 84602
USA
dahl@stat.byu.edu