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 five programming languages in the Java platform: Groovy, JavaScript, JRuby, Jython, and Kotlin. Each of these languages can use Java objects in their own syntax. Hence, **jsr223** is also an integration for R and the Java platform. It enables developers to leverage Java solutions from within R by embedding code snippets or evaluating script files. This approach is generally easier than **rJava**'s low-level approach that employs the Java Native Interface. **jsr223**'s multi-language support is dependent on the Java Scripting API: an implementation of "JSR-223: Scripting for the Java Platform" that defines a framework to embed scripts in Java applications. The **jsr223** package also features extensive data exchange capabilities and a callback interface that allows embedded scripts to access the current R session. In all, **jsr223** makes solutions developed in Java or any of the **jsr223**-supported languages easier to use in R.

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 sparked one of the most successful development platforms in computing history: Java. According to the TIOBE index, Java has been the most popular programming language, on average, over the last sixteen years. Java's success can be attributed to several factors. Perhaps the most important factor is platform-independence: the same Java program can run on several operating systems and hardware devices. Another important factor is that memory management is handled automatically for the programmer. Consequently, Java programs are easier to write and have fewer memory-related bugs than programs written in C/C++. These and other factors accelerated Java's adoption in enterprise systems which, in turn, established a thriving developer community that has created production-quality frameworks, libraries, and programming languages for the Java platform. Many successful Java solutions are relevant to data science today such as Hadoop, Hive, Spark, Cassandra, HBase, Mahout, Deeplearning4j, Stanford CoreNLP, and others.

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 users 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 interface to the Java platform. 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: Groovy, JavaScript, JRuby, Jython, and Kotlin. (JRuby and Jython are Java platform implementations of the Ruby and Python languages, respectively.) See Table 1 for a brief description of each language.

The **jsr223** multi-language integration is made possible by the Java Scripting API (Oracle, 2016), an

[1] **rJava** ranks in the 99th percentile for R package downloads according to `http://rdocumentation.org`.

Table 1: The five programming languages supported by **jsr223**. See the **jsr223** *User Manual* for code examples and language details.

Language	Description
Groovy	Groovy is a scripting language that follows Java syntax very closely. Hence, **jsr223** enables developers to embed Java source code directly in R script. Groovy also supports an optionally typed, functional paradigm with relaxed syntax for less verbose code.
JavaScript	JavaScript is well known for its use in web applications. However, its popularity has overflowed into standalone solutions involving databases, plotting, machine learning, and network-enabled utilities, to name just a few. **jsr223** uses Nashorn, the ECMA-compliant JavaScript implementation for the Java platform.
JRuby	JRuby is the Ruby implementation for the Java platform. Ruby is a general-purpose, object-oriented language with unique syntax. It is often used with the web application framework Ruby on Rails. Ruby libraries, called *gems*, can be accessed via **jsr223**.
Jython	Jython is the Python implementation for the Java platform. 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. Popular libraries SciPy and NumPy are available for the Java platform through JyNI (the Jython Native Interface).
Kotlin	Kotlin version 1.0 was released in 2016 making it the newest **jsr223**-supported language. It is a statically typed language that supports both functional and object-oriented programming paradigms. Kotlin has similarities to Java, but it often requires less code than Java to accomplish the same task. Kotlin and Java are the only languages officially supported by Google for Android application development.

implementation of 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, JSR-223 is the basis for our package. However, no knowledge of JSR-223 or the Java Scripting API is necessary to use **jsr223**. Figures 1 and 2 show how **rJava** and **jsr223** facilitate access to the Java platform. Where **rJava** uses JNI, **jsr223** uses the Java Scripting API and embeddable programming languages.

The primary goal of **jsr223** is to enable R developers to leverage existing Java solutions with relative ease. We demonstrate two typical use cases in this document with subjects that are of particular interest to many data scientists: a natural language processor, and a neural network classifier. In addition to Java solutions, R developers can use projects developed in any of the five **jsr223**-supported programming languages. In essence, **jsr223** opens R to a broader ecosystem.

For Java developers, **jsr223** facilitates writing high-performance, cross-platform R extensions using their preferred platform. **jsr223** also allows organizations that run enterprise Java applications to more readily develop dashboards and other business intelligence tools. Instead of writing R code to query raw data from a database, **jsr223** enables R packages to consume data directly from their application's Java object model where the data has been coalesced according to business rules. Java developers will also be interested to know that the **jsr223**-supported programming languages can implement interfaces and extend classes, just like the Java programming language. See "Extending existing Java solutions" in the **jsr223** *User Manual* for an in-depth code example that demonstrates extending Java classes and several other features.

In this paper, we present an introduction to the **jsr223** package. The section "**jsr223** package implementation and features overview" contains a brief description of the package's primary features and internals. The section "Typical use cases" provides code examples that highlight the **jsr223** package's core functionality. Finally, the "Software review" section puts the **jsr223** project in context with comparisons to other relevant software solutions.

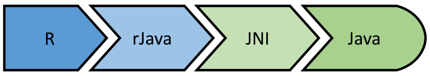

Figure 1: The **rJava** package facilitates low-level access to the Java platform through the Java Native Interface (JNI). Some knowledge of JNI is required.

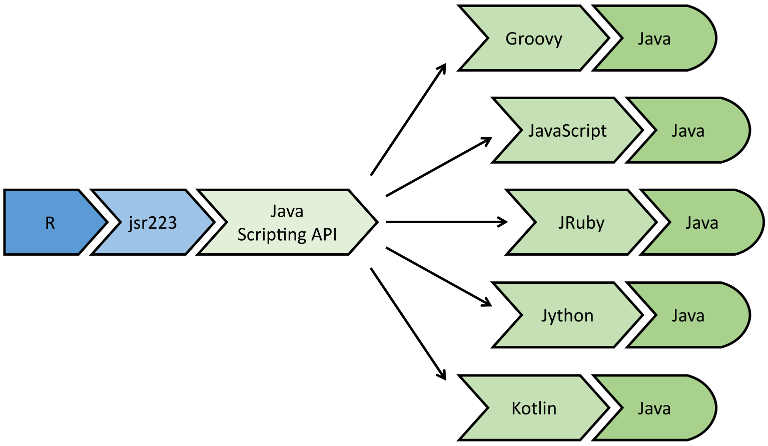

Figure 2: The **jsr223** package provides high-level access to the Java platform through five programming languages. Although **jrs223** uses the Java Scripting API in its implementation, users do not need to learn the API.

jsr223 package implementation and features overview

The **jsr223** package supports most of the major programming languages that implement JSR-223. Technically, any JSR-223 implementation will work with our package, but we may not officially support some languages. 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 (Dahl, 2018).) We also exclude languages that are not actively developed, such as BeanShell.

The **jsr223** package features extensive, configurable data exchange between R and Java via **jsr223**'s companion package **jdx** (Gilbert and Dahl, 2018). 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). 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.

The **jsr223** package also supports converting the most common data structures from the **jsr223**-supported languages. For example, **jsr223** can convert Jython dictionaries and user-defined JavaScript objects to R objects. Behind the scenes, every Java-based programming language uses Java objects. For example, a Jython dictionary is backed by a Java object that defines the dictionary's behavior. The **jsr223** package uses **jdx** to inspect these Java objects for data and convert them to an appropriate R object. In most cases, the default conversion rules are intuitive and seamless. Details covering data exchange features and behavior can be found in the **jsr223** *User Manual* and the **jdx** package vignette.

The **jsr223** programming interface follows design cues from **rscala**, and **V8** (Ooms, 2017b). The application programming interface is implemented using **R6** (Chang, 2017) classes for a traditional object-oriented style of programming. **R6** objects wrap methods in an R environment making them accessible from the associated variable using list-like syntax (e.g., `myObject$myMethod()`).

jsr223 uses **rJava** to load and communicate with the Java Virtual Machine (JVM): the abstract computing environment that executes compiled Java code. **jsr223** employs a client-server architecture and a custom multi-threaded messaging protocol to exchange data and handle script execution. This protocol optimizes performance by eliminating **rJava** calls that inspect generic return values and transform data, both which incur significant overhead. The protocol also facilitates callbacks that allow embedded scripts to manipulate variables and evaluate R code in the current R session. This callback implementation is lightweight, does not require any special R software configuration, and supports infinite callback recursion between R and the script engine (limited only by stack space).

Other distinguishing **jsr223** features include script compiling and string interpolation. Complete feature documentation is available in the **jsr223** *User Manual* vignette.

Typical use cases

This section includes examples that demonstrate typical use cases for the **jsr223** package. More code examples are available in the **jsr223** *User Manual*.

Using Java libraries

For this introductory example, 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 required 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. The import 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 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.

Using Java libraries with complex dependencies

In this example we use Deeplearning4j (DL4J) (Eclipse Deeplearning4j Development Team, 2018) to build a neural network. DL4J is an open-source deep learning solution for the Java platform. It is notable both for its scalability and performance. DL4J can run on a local computer with a standard CPU, or it can use Spark for distributed computing and GPUs for massively parallel processing. DL4J is modular in design and it has a large number of dependencies. As with many other Java solutions, it is designed to be installed using a software project management utility like Apache Maven, Gradle, or sbt. These utilities feature dependency managers that automatically download a library's dependencies from a central repository and make them accessible to your project. This is similar to installing an R package from CRAN using `install.packages`; by default, any referenced packages are also downloaded and installed.

The primary goal of this example is to show how **jsr223** can easily leverage complex Java solutions with the help of a project management utility. We will install both Groovy and DL4J using Apache Maven. We will then integrate Groovy script with R to create a simple neural network. The process is straightforward: i.) create a skeleton Java project; ii.) add dependencies to the project; iii.) build a class path referencing all of the dependencies; and iv.) pass the class path to **jsr223**. Though we use Maven here, the same concepts apply to any project management utility that supports Java.

To begin, visit the Maven web site (https://maven.apache.org/) and follow the installation instructions for your operating system. Next, create an empty folder for this sample project. Open a terminal (a system command prompt) and change the current directory to the project folder. Execute the following Maven command. It will create a skeleton Java project named 'stub' in a subfolder by the same name. The Java project is used only to retrieve dependencies; it is not required for the R project. If this is the first time Maven has been executed on your computer, several files will be downloaded to the local Maven repository cache on your computer.

```
mvn archetype:generate -DgroupId=none -DartifactId=stub -DinteractiveMode=false
```

Open the Maven project object model file, 'stub/pom.xml', in a plain text editor or an XML editor. Locate the XML element `<dependencies>`. It will be similar to the example displayed below. A `<dependency>` child element defines a single project dependency that will be retrieved from the Maven repository. Notice that a dependency has a group ID, an artifact ID, and a version. (*Artifact* is the general term for any file residing in a repository.) How do you know which dependencies are required for your project? They are often provided in installation documentation. Or, if you are starting from a code example, dependencies can be located in a Maven repository using fully-qualified Java class names.

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Maven dependency definitions can be located at `https://search.maven.org`. We will search for dependencies using the syntax '`g:<group-id> a:<artifact-id>`'. This avoids erroneous results and near-matches. A search string for each dependency in our demonstration is provided in the bullet list below. Perform a search using the first bullet item. In the search results, click the version number under the column heading "Latest Version." On the right-hand side of the page that follows you will see an XML Maven dependency definition for the artifact. Copy the XML and insert it after the last `</dependency>` end tag in your 'pom.xml' file. It is not necessary to preserve indentations or other white space. Repeat this process for each of the remaining search strings below.

- g:org.apache.logging.log4j a:log4j-core
- g:org.slf4j a:slf4j-log4j12
- g:org.deeplearning4j a:deeplearning4j-core
- g:org.nd4j a:nd4j-native-platform
- g:org.datavec a:datavec-api
- g:org.codehaus.groovy a:groovy-all

Save the 'pom.xml' file. In your terminal window, change directories to the Java project folder ('stub') and execute the following Maven command. This will download all of the dependencies to a local repository cache on your computer. It will also create a file named 'jsr223.classpath' in the parent folder. It contains a class path referencing all of the dependencies that will be used by **jsr223**.

```
mvn dependency:build-classpath -Dmdep.outputFile="../jsr223.classpath"
```

Now everything is in place to create a neural network using Groovy and DL4J. To keep the example simple, we use a feedforward neural network to classify species in the iris data set. The example involves an R script ('dl4j.R') and a Groovy script ('dl4j.groovy'). Both scripts can be downloaded from `https://github.com/floidgilbert/jsr223/tree/master/examples/Groovy/dl4j`. Save both scripts in the same folder as 'jsr223.classpath'.

The R script First, we read in the class path created by Maven and create the Groovy script engine.

```
library(jsr223)

file.name <- "jsr223.classpath"
class.path <- readChar(file.name, file.info(file.name)$size)
engine <- ScriptEngine$new("groovy", class.path)
```

Next, we set a seed for reproducible results. The value is saved in a variable that will be retrieved by the Groovy script.

```
seed <- 10
set.seed(seed)
```

The code that follows splits the iris data into train and test matrices. The inputs are centered and scaled. The labels are converted to a binary matrix format: for each record, the number 1 is placed in the column corresponding to the correct label.

```
train.idx <- sample(nrow(iris), nrow(iris) * 0.65)
train <- scale(as.matrix(iris[train.idx, 1:4]))
train.labels <- model.matrix(~ -1 + Species, iris[train.idx, ])
test <- scale(as.matrix(iris[-train.idx, 1:4]))
test.labels <- model.matrix(~ -1 + Species, iris[-train.idx, ])
```

Finally, we execute the Groovy script. The results will be printed to the console.

```
result <- engine$source("dl4j.groovy")
cat(result)
```

The Groovy script The Groovy script here follows Java syntax with one exception: we provide no class. Instead, we place all of the code at the top level to be executed at once. This is merely a style choice to keep the code samples easy to follow. The script begins by importing the necessary classes.

```
import org.deeplearning4j.eval.Evaluation;
import org.deeplearning4j.nn.conf.MultiLayerConfiguration;
import org.deeplearning4j.nn.conf.NeuralNetConfiguration;
import org.deeplearning4j.nn.conf.layers.DenseLayer;
import org.deeplearning4j.nn.conf.layers.OutputLayer;
import org.deeplearning4j.nn.multilayer.MultiLayerNetwork;
import org.deeplearning4j.nn.weights.WeightInit;
import org.nd4j.linalg.activations.Activation;
import org.nd4j.linalg.api.ndarray.INDArray;
import org.nd4j.linalg.cpu.nativecpu.NDArray;
import org.nd4j.linalg.dataset.DataSet;
import org.nd4j.linalg.learning.config.Sgd;
import org.nd4j.linalg.lossfunctions.LossFunctions;
```

Next, we convert the train and test data from R objects to the `DataSet` objects consumed by DL4J. We retrieve the data from the R environment using the `get` method of **jsr223**'s built-in `R` object. The R matrices are automatically converted to multi-dimensional Java arrays. These arrays are used to instantiate the `NDArray` objects which, in turn, are used to instantiate the `DataSet` objects.

```
DataSet train = new DataSet(
    new NDArray(R.get("train")),
    new NDArray(R.get("train.labels"))
);
DataSet test = new DataSet(
    new NDArray(R.get("test")),
    new NDArray(R.get("test.labels"))
);
```

Pulling the data from the R environment using the `R` object is just one convenient way to share data between R and the Java environment. It is also possible to push data from R to the Groovy environment, or to pass the data as function parameters. Note: for very large data sets it is impractical to exchange data between R and Java using **jsr223** methods. Instead, load the data on the Java side for processing using DL4J classes optimized for big data.

Here we configure a feedforward neural network with backpropagation. The network consists of four inputs, a seven node hidden layer, a three node hidden layer, and a three node output layer. An explanation of the network's hyperparameters is beyond the scope of this discussion. See https://deeplearning4j.org/docs/latest/deeplearning4j-troubleshooting-training for a DL4J hyperameter reference.

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .seed(R.get("seed").intValue())
    .activation(Activation.TANH)
    .weightInit(WeightInit.XAVIER)
    .updater(new Sgd(0.1)) // Learning rate.
    .list()
    .layer(new DenseLayer.Builder().nIn(4).nOut(7).build())
    .layer(new DenseLayer.Builder().nIn(7).nOut(3).build())
    .layer(
        new OutputLayer.Builder(LossFunctions.LossFunction.NEGATIVELOGLIKELIHOOD)
            .activation(Activation.SOFTMAX)
            .nIn(3)
            .nOut(3)
            .build()
      )
    .backprop(true)
    .build();
```

We use the network configuration to initialize a model which is then trained over 200 epochs.

```
MultiLayerNetwork model = new MultiLayerNetwork(conf);
model.init();

for (int i = 0; i < 200; i++) {
    model.fit(train);
}
```

At last, the trained model is evaluated using the test data. The last line produces a text report including classification metrics and a confusion matrix.

```
Evaluation eval = new Evaluation(3); // 3 is the number of possible classes
INDArray output = model.output(test.getFeatures());
eval.eval(test.getLabels(), output);
eval.stats();
```

Results Executing the R script will produce the following console output. Our simple model performs reasonably well in this case, misclassifying two out of 53 observations.

```
## ========================Evaluation Metrics========================
##  # of classes:    3
##  Accuracy:        0.9623
##  Precision:       0.9628
##  Recall:          0.9628
##  F1 Score:        0.9628
## Precision, recall & F1: macro-averaged (equally weighted avg. of 3 classes)
##
##
## =========================Confusion Matrix=========================
##   0  1  2
## ----------
##  17  0  0 | 0 = 0
##   0 16  1 | 1 = 1
##   0  1 18 | 2 = 2
##
## Confusion matrix format: Actual (rowClass) predicted as (columnClass) N times
## ==================================================================
```

This example demonstrated that complex Java solutions can be integrated with R using **jsr223** and standard dependency management practices.

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 a gem 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 some Python libraries is limited on the Java platform. Please see "Python integrations software review" for more information.

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.

[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."

```
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
```

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 syntax checking. **V8** includes an interface to check JavaScript code syntax. The Java Scripting API does not provide an interface for syntax checking, hence, **jsr223** does not provide this feature. We have investigated other avenues to check syntax, but none are uniformly reliable across all of the **jsr223**-supported languages. Moreover, this feature is not critical for most integration scenarios; syntax 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 plotting; 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 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

[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.

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. [p12]

W. Chang. *R6: Classes with Reference Semantics*, 2017. URL https://CRAN.R-project.org/package=R6. R package version 2.2.2. [p3]

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. [p3]

Eclipse Deeplearning4j Development Team. *Deeplearning4j: Open-Source Distributed Deep Learning for the JVM*, 2018. URL http://deeplearning4j.org. [p5]

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. [p3]

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. [p12]

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. [p4]

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]

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. [p3, 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. [p1]

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. [p2]

A. Taylor, M. Marcus, and B. Santorini. The penn treebank: An overview. In N. Ide, editor, *Text, Speech and Language Technology*, volume 20. Springer-Verlag, 2003. URL https://doi.org/10.1007/978-94-010-0201-1_1. [p5]

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