Package 'jsr223'

December 13, 2018

Type Package

Title A 'Java' Platform Integration for 'R' with Programming Languages 'Groovy', 'JavaScript', 'JRuby' ('Ruby'), 'Jython' ('Python'), and 'Kotlin'

Version 0.3.3

Date 2018-12-12

Description Provides 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'. The 'jsr223'-supported programming languages include 'Groovy', 'JavaScript', 'JRuby' ('Ruby'), 'Jython' ('Python'), and 'Kotlin'. Any of these languages can use and extend 'Java' classes in natural syntax. Furthermore, solutions developed in any of the 'jsr223'-supported languages are also accessible to 'R' developers. The 'jsr223' package also features callbacks, script compiling, and string interpolation. In all, 'jsr223' significantly extends the computing capabilities of the 'R' software environment.

License GPL (>= 2) | BSD_3_clause + file LICENSE

Imports jdx (>= 0.1.0), rJava (>= 0.9-8), R6 (>= 2.2.0), utils (>= 3.3.0), curl (>= 3.0.0)

SystemRequirements Java Runtime Environment (>= 8)

URL https://github.com/floidgilbert/jsr223

BugReports https://github.com/floidgilbert/jsr223/issues

Encoding UTF-8

Suggests testthat, knitr, rmarkdown, pander

VignetteBuilder knitr

NeedsCompilation no

Author Floid R. Gilbert [aut, cre], David B. Dahl [aut]

Maintainer Floid R. Gilbert <floid.r.gilbert@gmail.com>

Repository CRAN

Date/Publication 2018-12-12 23:40:03 UTC

R topics documented:

`jsr223-package` *A Java Platform Integration for R with Programming Languages Groovy, JavaScript, JRuby (Ruby), Jython (Python), and Kotlin*

Description

The **jsr223** package provides a high-level integration for Java that makes Java objects easy to use from within R and simplifies bi-directional data exchange for a wide variety of objects. Furthermore, **jsr223** employs the Java Scripting API to bring several scripting languages to the R software environment: JavaScript, Ruby, Python, Groovy, and Kotlin.

Details

The complete documentation is in the **jsr223** User Manual. It includes in-depth code examples and it covers details, such as data exchange, that cannot be addressed easily in the R documentation.

Author(s)

Floid R. Gilbert <floid.r.gilbert@gmail.com>, David B. Dahl <dahl@stat.byu.edu>

See Also

`ScriptEngine`

Examples

```
# Simple example embedding JavaScript.
library("jsr223")
engine <- ScriptEngine$new("javascript")
engine$radius <- 4
engine %~% "var area = Math.PI * Math.pow(radius, 2)"
cat ("The area of the circle is ", engine$area, ".\n", sep = "")

# Use callbacks to set values, get values, and execute R code
# in the current R session via the global R object.
# Access R from JavaScript.
engine %@% "R.set('a', 12);"
engine %@% "print(\"The value of 'a' is \" + R.get('a') + \".\");"
engine %@% "var randomNormal = R.eval('rnorm(5)');"
```

```
engine$randomNormal

# Use a Java object.
engine$randomNormal <- rnorm(5)
engine$randomNormal
engine %@% "java.util.Arrays.sort(randomNormal)"
engine$randomNormal

# Close the engine and release resources.
engine$terminate()
```

`CompiledScript` *CompiledScript Class*

Description

The `CompiledScript` class represents script compiled by a script engine.

Usage

```
CompiledScript
```

Details

CompiledScript does not have a public constructor. Create an instance of this class with the `ScriptEngine` methods `compile` and `compileSource`.

The complete **jsr223** documentation can be found in the User Manual.

Value

Object of `R6Class` that represents a compiled script.

Methods

`eval(discard.return.value = FALSE, bindings = NULL)` Executes the compiled code referenced by the object. If `discard.return.value = FALSE`, the method returns the result of the last expression in the script, if any, or `NULL` otherwise. The `bindings` argument accepts an R named list. The name/value pairs in the list replace the script engine's global bindings during script execution.

See Also

`ScriptEngine`

Examples

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

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

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

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

## 5

# Supply new bindings...
lst <- list(c = 6, d = 7)
cs$eval(bindings = lst)

## 13

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

## 5

# The following line executes the code but discards the return value.
cs$eval(discard.return.value = TRUE)

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

`getKotlinScriptEngineJars` *Search for Required Kotlin JAR Files*

Description

The `getKotlinScriptEngineJars` function searches a directory recursively for '`kotlin*.jar`' files required to create an instance of the Kotlin script engine. Because Kotlin does not provide a standalone script engine JAR file, we include this convenience function to simplify adding JAR files to the class path.

Usage

```
getKotlinScriptEngineJars(
  directory,
```

```
  minimum = TRUE
)
```

Arguments

directory	A character vector of length one specifying a path that contains Kotlin script engine JAR files.
minimum	A logical vector of length one. When TRUE (the default), the function returns only the minimum required JAR files to instantiate a script engine. If FALSE, all JAR files of the pattern 'kotlin*.jar' will be returned.

Details

The function searches the given directory recursively. If one or more of the required JAR files are not found, the function throws an error with a message listing the required files.

As of this writing, a standalone Kotlin compiler installation does not contain all of the required files to create a script engine instance. See section "Script engine installation and instantiation" in the User Manual for more information.

Value

A character vector containing paths to the JAR files.

See Also

ScriptEngine

Examples

```
## Not run:
library("jsr223")
jars <- getKotlinScriptEngineJars("~/my-path/kotlin")
engine <- ScriptEngine$new("kotlin", jars)
engine %~% "1 + 1"
engine$terminate()

## End(Not run)
```

---

names *Retrieve the Names for a ScriptEngine or CompiledScript class*

Description

These methods retrieve the names associated with objects of class ScriptEngine or CompiledScript.

Usage

```
  ## S3 method for class 'ScriptEngine'
names(x, ...)
  ## S3 method for class 'CompiledScript'
names(x, ...)
```

Arguments

x	An object of class ScriptEngine or CompiledScript.
...	Ignored.

---

print *Print or Return a Character Representation of a Script Engine or Compiled Script Class Object*

Description

These methods print or return a character representation of a script engine.

Usage

```
  ## S3 method for class 'ScriptEngine'
print(x, ...)
  ## S3 method for class 'ScriptEngine'
toString(x, ...)
  ## S3 method for class 'CompiledScript'
print(x, ...)
  ## S3 method for class 'CompiledScript'
toString(x, ...)
```

Arguments

x	An object of class ScriptEngine or CompiledScript.
...	Currently ignored.

`ScriptEngine` *ScriptEngine Class*

Description

The `ScriptEngine` class represents a Java-based script engine. A `ScriptEngine` instance is used to execute arbitrary code and pass data back and forth between the script engine environment and R. The script engine environment contains a global object named `R` that facilitates callbacks into the R environment. Complete documentation is located in the **jsr223** User Manual.

Usage

```
ScriptEngine
```

Details

The complete **jsr223** documentation can be found in the User Manual. It includes more in-depth code examples and it covers details, such as data exchange, that cannot be addressed as easily in the R documentation.

In this document, the section **Constructor Method** details the options required to create a `ScriptEngine` instance. The section **Script Engine Settings** describes class methods pertaining to configurable options. The section **Methods** lists the rest of the class methods. The section **Callbacks** provides an overview of the functionality allowing code in the script engine to access data and execute code in the R environment. Finally, **Script Engines** includes links to the supported script engine providers.

The bridge between R and the script engine is not thread-safe; multiple R threads should not simultaneously access the same engine.

Value

Object of `R6Class` that represents an instance of a Java-based script engine.

Constructor Method

`new(engine.name, class.path = "")` Creates a script engine object.

`engine.name` is a character vector of length one that specifies the type of script engine to create. Valid engine names are '`js`' or '`javascript`' for JavaScript, '`ruby`', '`python`', '`groovy`', and '`kotlin`'. The engine name is case sensitive.

`class.path` is a character vector of paths to any JAR files that are required for the scripting engine and any script dependencies. A *class.path* value is required for all engines except JavaScript. Scripting engine JAR files can be obtained from the language-specific web sites referenced in the section **Script Engines**. Note that class paths accumulate between script engine instances started in the same R session because they all use the same Java Virtual Machine. This is a limitation of **rJava**, the package that **jsr223** builds on.

Script Engine Settings

Several script engine settings are exposed using Java-style getter/setter methods. Other methods are addressed in the **Methods** section.

`getArrayOrder()` Returns a length-one character vector containing the current array order scheme. See `setArrayOrder` for more information.

`setArrayOrder(value)` Sets the current array ordering scheme used for all n-dimensional arrays (such as matrices) converted to and from the script engine. Valid values are `'row-major'` (the default), `'column-major'`, and `'column-minor'`. These indexing schemes are described in the User Manual. This method returns the previous setting invisibly.

`getCoerceFactors()` Returns a length-one logical vector indicating whether the *coerce factors* setting is enabled. See `setCoerceFactors` for more information.

`setCoerceFactors(value)` Enables or disables the *coerce factors* setting. Valid values are `TRUE` (the default) and `FALSE`. When enabled, an attempt is made to coerce R factors to integer, numeric, or logical vectors before converting them to a Java array. If the attempt fails, or if the setting is disabled, the factor is converted to a Java string array. This setting applies to standalone factors as well as factors in data frames. This method returns the previous setting invisibly.

`getDataFrameRowMajor()` Returns a length-one logical vector indicating whether the *data frame row major* setting is enabled. See `setDataFrameRowMajor` for more information.

`setDataFrameRowMajor(value)` Enables or disables the *data frame row major* setting. Valid values are `TRUE` (the default) and `FALSE`. When enabled, data frames are converted to Java objects in row-major fashion. When disabled, column-major ordering is used. See the User Manual for details. This method returns the previous setting invisibly.

`getInterpolate()` Returns a length-one logical vector indicating whether the *string interpolation* setting is enabled. See `setInterpolate` for more information.

`setInterpolate(value)` Enables or disables the *string interpolation* setting. Valid values are `TRUE` (the default) and `FALSE`. When enabled, R code placed between `@{` and `}` in a script is evaluated and replaced by the a string representation of the return value. A script may contain multiple `@{...}` expressions. This method returns the previous setting invisibly.

`getLengthOneVectorAsArray()` Returns a length-one logical vector indicating whether the *length one vector as array* setting is enabled. See `setLengthOneVectorAsArray` for more information.

`setLengthOneVectorAsArray(value)` Enables or disables the *length one vector as array* setting. Valid values are `TRUE` and `FALSE` (the default). When disabled, length-one R vectors and factors are converted to Java scalars. When enabled, length-one R vectors and factors are converted to Java arrays. This latter effect can also be produced by wrapping the vector in the "as-is" function before passing it to the script engine (e.g. `engine$myValue <- I(variable)`). This method returns the previous setting invisibly.

`getStandardOutputMode()` Returns a length-one character vector containing the current standard output mode. See `setStandardOutputMode` for more information.

`setStandardOutputMode(value)` Controls how text written to standard output is handled. The default value, `'console'`, indicates that standard output will be printed in the R console. This output cannot be captured using standard R methods. The `'buffer'` setting indicates that standard output will be saved in an internal buffer. This buffered output can be retrieved and

cleared using the `getStandardOutput` method, or cleared using the `clearStandardOutput` method. Finally, the `'quiet'` setting indicates that standard output will be discarded. This method returns the previous setting invisibly.

`getStringsAsFactors()` Returns a length-one logical vector, or `NULL`, indicating whether the *strings as factors* setting is enabled. See `setStringsAsFactors` for more information.

`setStringsAsFactors(value)` When converting a Java object to a data frame, the *strings as factors* setting controls whether character vectors are converted to factors. The default value of `NULL` indicates that the R system setting `stringsAsFactors` should be used (see `getOption("stringsAsFactors")`). A value of `TRUE` ensures that character vectors are converted to factors. A setting of `FALSE` disables conversion to factors. This method returns the previous setting invisibly.

Methods

This section includes `ScriptEngine` class methods that do not get/set script engine options. See **Script Engine Settings** for information on script engine options.

`$identifier` Retrieves the global variable named `identifier` from the script engine environment. For example, if `engine` is a script engine instance, retrieve the value of a variabled named `myValue` using `engine$myValue`. Quote names that are not valid variable names in R (e.g. engine$`a-3`). This method is equivalent to `get(identifier)`.

`$identifier <- value` Assigns `value` to the global variable named `identifier` in the script engine environment. The R object contained in `value` is converted to a Java object. For example, if `engine` is a script engine instance, set the value of a variabled named `myValue` using `engine$myValue <- 1`. Quote names that are not valid variable names in R (e.g. engine$`a-3` <- 1). This method is equivalent to `set(identifier, value)`.

`%@% script` Evaluates code contained in the `script` character vector and returns `NULL` invisibly. This method is equivalent to using `eval(script, discard.return.value = TRUE)`.

`%~% script` Evaluates code contained in the `script` character vector and returns the result of the last expression in the script, if any, or `NULL` otherwise. This method is equivalent to using `eval(script, discard.return.value = FALSE)`.

`clearStandardOutput()` Empties the script engine's standard output buffer. This method is only useful when the standard output mode has been set to `'buffer'`. See the methods `getStandardOutputMode` and `setStandardOutputMode` in **Script Engine Settings** for more information.

`compile(script)` Compiles code contained in the `script` character vector. Returns a `CompiledScript` object.

`compileSource(file.name)` Compiles code contained in the file specified by the length-one character vector `file.name`. Local file paths or URLs are accepted. Returns a `CompiledScript` object.

`console()` Starts a simple REPL in the current script language. The REPL is useful for quickly setting and inspecting variables in the script engine. Returned values are printed to the console using `base::dput`. Only single-line commands are supported: no line continuations or carriage returns are allowed. Enter '`exit`' to return to the R prompt.

`eval(script, discard.return.value = FALSE, bindings = NULL)` Evaluates code contained in the `script` character vector. If `discard.return.value = FALSE`, the method returns the result of the last expression in the script, if any, or `NULL` otherwise. The `bindings` argument accepts an R named list. The name/value pairs in the list replace the script engine's global bindings during script execution.

`finalize()` This method is called before the object is garbage-collected to release resources. Do not call this method directly. Use the `terminate` method instead.

`get(identifier)` Retrieves the value of a global variable in the script engine environment. The name of the variable is specified in the length-one character vector `identifier`. For example, if `engine` is a script engine instance, retrieve the value of a variabled named `myValue` using `engine$get("myValue")`. This method is equivalent to `$identifier`.

`getBindings()` Lists all of the global variables in the script engine environment. This method returns a named list where the names are the variable names and the values are the respective Java class names.

`getClassPath()` Returns the class path as a character vector. The class path is set in the `ScriptEngine` constructor method. Note that class paths accumulate between script engine instances started in the same R session because they all use the same Java Virtual Machine. This is a limitation of **rJava**, the package that **jsr223** builds on.

`getJavaClassName(identifier)` Retrieves the Java class name of the global variable named `identifier` from the script engine environment. This method is equivalent to using `$identifier`.

`getScriptEngineInformation()` Returns a named list containing information about the script engine including the name, language, and version.

`getStandardOutput()` Returns a character vector of length one containing the contents of the script engine's standard output buffer. The standard output buffer is emptied. This method is only useful when the standard output mode has been set to `'buffer'`. See `setStandardOutputMode` in **Script Engine Settings** for more information.

`initialize()` The constructor for this class. Do not call this method directly. Use `ScriptEngine$new()` instead.

`invokeFunction(function.name, ...)` Invoke a function in the script engine environment. The argument `function.name` is a character vector containing the name of the function to be called. The `...` indicates any number of arguments to be passed to the script function. The return value is the result of the function converted to an R object.

`invokeMethod(object.name, method.name, ...)` Invoke a method of an object in the script engine environment. The arguments `object.name` and `method.name` are character vectors containing the names of the object and method, respectively. The `...` indicates any number of arguments to be passed to the method. The return value is the result of the method converted to an R object. The Groovy, Python, and Kotlin engines can use `invokeMethod` to call methods of Java objects. The JavaScript and Ruby engines only support calling methods of native scripting objects.

`isInitialized()` Returns `TRUE` or `FALSE` indicating whether the script engine instance is active (i.e., it has not been explicitly terminated).

`remove(identifier)` Removes a variable from the script engine environment. The name of the variable is specified in the length-one character vector `identifier`. For example, if `engine` is a script engine instance, remove the variabled named `myValue` using `engine$remove("myValue")`. Returns `TRUE` if the variable exists and `FALSE` otherwise.

`set(identifier, value)` Assigns `value` to a global variable in the script engine environment. The name of the variable is specified in the length-one character vector `identifier`. The R object `value` is converted to a Java object. For example, if `engine` is a script engine instance, set the value of a variabled named `myValue` using `engine$set("myValue", 1)`. This method is equivalent to `$identifier <- value`).

`source(file.name, discard.return.value = FALSE, bindings = NULL)` Evaluates code contained in the file specified by the length-one character vector `file.name`. Local file paths or URLs are accepted. If `discard.return.value = FALSE`, the method returns the result of the last expression in the script, if any, or `NULL` otherwise. The `bindings` argument accepts an R named list. The name/value pairs in the list replace the script engine's global bindings during script execution.

`terminate()` Terminates the script engine instance and releases associated resources. Call this method when the script engine is no longer needed.

Callbacks

Embedded scripts can access the R environment using the **jsr223** callback interface. When a script engine is started, **jsr223** creates a global object named `R` in the script engine's environment. This object is used to execute R code and set/get variables in the R session's global environment. Infinite recursive calls between R and the script engine are supported. The only limitation is available stack space.

To set a variable in the R global environment, use

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

To retrieve a variable from the R global environment, use

```
engine %~% "R.get('a')"
```

Finally, to evaluate R code, use

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

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

Script Engines

The **jsr223** package supports the following Java-based languages. Follow a link below to visit the language's supporting web site and to download script engine JAR files. Detailed instructions are found in the User Manual.

Groovy – A Java-like language enhanced with modern dynamic programming features.

JavaScript (Nashorn) – Nashorn is the JavaScript dialect included in Java 8 and above. No download or *class.path* parameter is required to use JavaScript with **jsr223**.

JRuby – A Java-based implementation of the Ruby programming language.

Jython – A Java-based implementation of the Python programming language.

Kotlin – A statically typed programming language that supports both functional and object-oriented programming paradigms.

See Also

`CompiledScript`

Examples

```
library("jsr223")

# Create an instance of a JavaScript engine. Note that the
# script engine's JAR file is not required for the class.path
# parameter because JavaScript is included with JRE 8.
engine <- ScriptEngine$new("javascript")

# Evaluate arbitrary code. Multiline code is allowed.
engine %~% "var a = Math.PI;"

# Retrieve the value of a global JavaScript variable.
cat("The value of 'a' is ", engine$a, ".", sep = "")

# Set the value of a global variable. If the variable does
# not exist in the engine environment, it will be created.
engine$a <- 10
cat("The value of 'a' is now ", engine$a, ".", sep = "")

# Use callbacks to set values, get values, and execute R code
# in the current R session via the global R object.
# Access R from JavaScript.
engine %@% "R.set('a', 12);"
engine %@% "print(\"The value of 'a' is now \" + R.get('a') + \".\");"
engine %@% "var randomNormal = R.eval('rnorm(5)');"
engine$randomNormal

# Use a Java object.
engine$randomNormal <- rnorm(5)
engine$randomNormal
engine %@% "java.util.Arrays.sort(randomNormal)"
engine$randomNormal

# Enable property to convert length one vectors to arrays
# instead of scalar values.
engine$setLengthOneVectorAsArray(TRUE)
engine$c <- 1
engine %~% "c[0]" # Returns 1
engine$setLengthOneVectorAsArray(FALSE)

# Suppress console output.
engine$setStandardOutputMode("quiet")
engine %~% "print('Hello (1)');"

# Re-enable console output
engine$setStandardOutputMode("console")
engine %~% "print('Hello (2)');"

# Close the engine and release resources.
engine$terminate()

## Not run:
```

```
# Create a JRuby engine by specifying the engine name and
# the class path for the engine JAR. The JAR file path
# will be different on your system.
engine <- ScriptEngine$new(
  engine.name = "jruby"
  , class.path = "../engines/jruby-complete.jar"
)

# Assign a value to a variable. This will create a global
# variable in the Ruby environment.
engine$c <- pi

# Reference the previous value in a code snippet. Note that
# Ruby requires a "$" designator for global variables.
engine %~% "3 * $c"

# Evaluate a script file.
engine$source("./my_script.rb")

# Terminate the engine.
engine$terminate()

## End(Not run)
```

Index