Package 'rJava'

September 24, 2024

Version 1.0-11

Title Low-Level R to Java Interface

Author Simon Urbanek <simon.urbanek@r-project.org>

Maintainer Simon Urbanek <simon.urbanek@r-project.org>

Depends R (>= 3.6.0), methods

Description Low-level interface to Java VM very much like .C/.Call and friends. Allows creation of objects, calling methods and accessing fields.

License LGPL-2.1

URL http://www.rforge.net/rJava/

SystemRequirements Java JDK 1.2 or higher (for JRI/REngine JDK 1.4 or higher), GNU make

BugReports https://github.com/s-u/rJava/issues

NeedsCompilation yes

Contents

`.jgc` *Invoke Java Garbage Collection*

Description

`.jgc` invokes the R and Java garbage collectors.

Usage

```
.jgc(R.gc = TRUE, ...)
```

Arguments

`R.gc`	logical, if `TRUE` then `gc(...)` is called first, if `FALSE` only Java garbage collector is called
`...`	any additional parameters passed to `gc()`

Details

`.jgc` invokes the R garbage collector (unless `R.gc=FALSE`) which removes any unused Java references and then invokes the Java garbage collector to reclaim Java heap space.

Author(s)

Simon Urbanek

`.jinstanceof` *Is a java object an instance of a given java class*

Description

Is a java object an instance of a given java class

Usage

```
o %instanceof% cl
.jinstanceof( o, cl )
```

Arguments

`o`	java object reference
`cl`	java class. This can be a character vector of length one giving the name of the class, or another java object, or an instance of the Class class, or a object of class `jclassName`.

Value

TRUE if o is an instance of cl

Author(s)

Romain Francois <francoisromain@free.fr>

Examples

```
Double <- J("java.lang.Double")
d <- new( Double, "10.2" )

# character
d %instanceof% "java.lang.Double"
d %instanceof% "java.lang.Number"

# jclassName
d %instanceof% Double

# instance of Class
Double.class <- Double@jobj
d %instanceof% Double.class

# other object
other.double <- new( Double, 10.2 )
d %instanceof% other.double
```

`aslist` *Converts java objects or arrays to R lists*

Description

`as.list` is implemented for java objects and java arrays to facilitate using `lapply` calls over elements of a java array or items of an Iterator associated with an Iterable object

For java array references, `as.list` is mapped to `.jevalArray`

For java objects that implement the Iterable interface, the list is created by iterating over the associated iterator

Usage

```
## S3 method for class 'jobjRef'
as.list(x, ...)
## S3 method for class 'jarrayRef'
as.list(x, ...)
```

Arguments

`x`	java array or Iterable java object
`...`	ignored

Value

An R list, or vector.

Note

The function is not intended to be called directly. It is implemented so that java arrays or Iterable java objects can be used as the first argument of `lapply`

See Also

`.jevalArray`, `lapply`

Examples

```
  # lapplying over a java array
  a <- .jarray( list(
.jnew( "java/awt/Point", 10L, 10L ),
.jnew( "java/awt/Point", 30L, 30L )
  ) )
  lapply( a, function(point){
with(point, {
(x + y ) ^ 2
} )
  } )

# lapply over a Vector (implements Iterable)
v <- .jnew("java/util/Vector")
v$add( "foo" )
```

```
v$add( .jnew("java/lang/Double", 10.2 ) )
sapply( v, function(item) item$getClass()$getName() )
```

`clone` *Object cloner*

Description

Generic function to clone objects

Usage

```
clone(x, ...)
```

Arguments

`x`	An object to clone
`...`	Further arguments, ignored

Value

A clone of the object

Methods

clone `signature(x = "jobjRef")`: clone a java object reference (must implement Cloneable)

clone `signature(x = "jarrayRef")`: clone a java rugged array (not yet implemented)

clone `signature(x = "jrectRef")`: clone a java rectangular array (not yet implemented)

Warning

The implementation of clone for java object references uses the clone method of the Object class. The reading of its description in the java help page is *strongly* recommended.

Examples

```
p1 <- .jnew("java/awt/Point" )
p2 <- clone( p1 )
p2$move( 10L, 10L )
p1$getX()

# check that p1 and p2 are not references to the same java object
stopifnot( p1$getX() == 0 )
stopifnot( p2$getX() == 10 )
```

`Exceptions` *Exception handling*

Description

R handling of java exception

Usage

```
## S3 method for class 'Throwable'
x$name
## S3 replacement method for class 'Throwable'
x$name  <- value
```

Arguments

`x`	condition
`name`	...
`value`	...

Details

Java exceptions are mapped to R conditions that are relayed by the `stop` function.

The R condition contains the actual exception object as the `jobj` item.

The class name of the R condition is made of a vector of simple java class names, the class names without their package path. This allows the R code to use direct handlers similar to direct exception handlers in java. See the example below.

Examples

```
Integer <- J("java.lang.Integer")
tryCatch( Integer$parseInt( "10.." ), NumberFormatException = function(e){
e$jobj$printStackTrace()
} )

# the dollar method is also implemented for Throwable conditions,
# so that syntactic sugar can be used on condition objects
# however, in the example below e is __not__ a jobjRef object reference
tryCatch( Integer$parseInt( "10.." ), NumberFormatException = function(e){
e$printStackTrace()
} )
```

J *High level API for accessing Java*

Description

J creates a Java class reference or calls a Java method

Usage

```
J(class, method, ..., class.loader=.rJava.class.loader)
```

Arguments

class	java object reference or fully qualified class name in JNI notation (e.g "java/lang/String") or standard java notation (e.g "java.lang.String")
method	if present then J results in a method call, otherwise it just creates a class name reference.
...	optional parameters that will be passed to the method (if the method argument is present)
class.loader	optional, custom loader to use if a class look-up is necessary (i.e., if class is a string)

Details

J is the high-level access to Java.

If the method argument is missing then code must be a class name and J creates a class name reference that can be used either in a call to new to create a new Java object (e.g. new(J("java.lang.String"), "foo")) or with $ operator to call a static method (e.g. J("java.lang.Double")$parseDouble("10.2").)

If the method argument is present then it must be a string vector of length one which defines the method to be called on the object.

Value

If method is missing the the returned value is an object of the class jclassName. Otherwise the value is the result of the method invocation. In the latter case Java exceptions may be thrown and the function doesn't return.

Note

J is a high-level API which is slower than .jnew or .jcall since it has to use reflection to find the most suitable method.

See Also

.jcall, .jnew

Examples

```
if (!nzchar(Sys.getenv("NOAWT"))) {
  f <- new(J("java.awt.Frame"), "Hello")
  f$setVisible(TRUE)
}

J("java.lang.Double")$parseDouble("10.2")
J("java.lang.Double", "parseDouble", "10.2" )

Double <- J("java.lang.Double")
Double$parseDouble( "10.2")

# String[] strings = new String[]{ "string", "array" } ;
  strings <- .jarray( c("string", "array") )
# this uses the JList( Object[] ) constructor
# even though the "strings" parameter is a String[]
  l <- new( J("javax.swing.JList"), strings)
```

---

jarray *Java array handling functions*

Description

`.jarray` takes a vector (or a list of Java references) as its argument, creates a Java array containing the elements of the vector (or list) and returns a reference to such newly created array.

`.jevalArray` takes a reference to a Java array and returns its contents (if possible).

Usage

```
.jarray(x, contents.class = NULL, dispatch = FALSE)
.jevalArray(obj, rawJNIRefSignature = NULL, silent = FALSE, simplify = FALSE)
```

Arguments

`x`	vector or a list of Java references
`contents.class`	common class of the contained objects, see details
`obj`	Java object reference to an array that is to be evaluated
`rawJNIRefSignature`	JNI signature that would be used for conversion. If set to `NULL`, the signature is detected automatically.
`silent`	if set to true, warnings are suppressed
`dispatch`	logical. If `TRUE` the code attempts to dispatch to either a `jarrayRef` object for rugged arrays and `jrectRef` objects for rectangular arrays, creating possibly a multi-dimensional object in Java (e.g., when used with a matrix).
`simplify`	if set to `TRUE` more than two-dimensional arrays are converted to native objects (e.g., matrices) if their type and size matches (essentially the inverse for objects created with `dispatch=TRUE`).

Details

`.jarray`: The input can be either a vector of some sort (such as numeric, integer, logical, ...) or a list of Java references. The contents is pushed to the Java side and a corresponding array is created. The type of the array depends on the input vector type. For example numeric vector creates `double[]` array, integer vector creates `int[]` array, character vector `String[]` array and so on. If `x` is a list, it must contain Java references only (or `NULL`s which will be treated as `NULL` references).

The `contents.class` parameter is used only if `x` is a list of Java object references and it can specify the class that will be used for all objects in the array. If set to `NULL` no assumption is made and `java/lang/Object` will be used. Use with care and only if you know what you're doing - you can always use `.jcast` to cast the entire array to another type even if you use a more general object type. One typical use is to construct multi-dimensional arrays which mandates passing the array type as `contents.class`.

The result is a reference to the newly created array.

The inverse function which fetches the elements of an array reference is `.jevalArray`.

`.jevalArray` currently supports only a subset of all possible array types. Recursive arrays are handled by returning a list of references which can then be evaluated separately. The only exception is `simplify=TRUE` in which case `.jevalArray` attempts to convert multi-dimensional arrays into native R type if there is a such. This only works for rectangular arrays of the same basic type (i.e. the length and type of each referenced array is the same - sometimes matrices are represented that way in Java).

Value

`.jarray` returns a Java array reference (`jarrayRef` or `jrectRef`) to an array created with the supplied contents.

`.jevalArray` returns the contents of the array object.

Examples

```
a <- .jarray(1:10)
print(a)
.jevalArray(a)
b <- .jarray(c("hello","world"))
print(b)
c <- .jarray(list(a,b))
print(c)
# simple .jevalArray will return a list of references
print(l <- .jevalArray(c))
# to convert it back, use lapply
lapply(l, .jevalArray)

# two-dimensional array resulting in int[2][10]
d <- .jarray(list(a,a),"[I")
print(d)
# use dispatch to convert a matrix to [[D
e <- .jarray(matrix(1:12/2, 3), dispatch=TRUE)
print(e)
# simplify it back to a matrix
.jevalArray(e, simplify=TRUE)
```

`jarrayRef-class` *Class "jarrayRef" Reference to an array Java object*

Description

This class is a subclass of jobjRef-class and represents a reference to an array Java object.

Objects from the Class

Objects cannot be created directly, but only as the return value of `.jcall` function.

Slots

`jsig`**:** JNI signature of the array type

`jobj`**:** Internal identifier of the object

`jclass`**:** Inherited from `jobjRef`, but unspecified

Methods

[`signature(x = "jarrayRef")`: *not yet implemented*

[[`signature(x = "jarrayRef")`: R indexing of java arrays

[[<- `signature(x = "jarrayRef")`: replacement method

`head signature(x = "jarrayRef")`: head of the java array

`tail signature(x = "jarrayRef")`: tail of the java array

length `signature(object = "jarrayRef")`: Number of java objects in the java array

str `signature(object = "jarrayRef")`: ...

unique `signature(x = "jarrayRef")`: *not yet implemented*

duplicated `signature(x = "jarrayRef")`: *not yet implemented*

anyDuplicated `signature(x = "jarrayRef")`: *not yet implemented*

sort `signature(x = "jarrayRef")`: *not yet implemented*

rev `signature(x = "jarrayRef")`: *not yet implemented*

min `signature(x = "jarrayRef")`: *not yet implemented*

max `signature(x = "jarrayRef")`: *not yet implemented*

range `signature(x = "jarrayRef")`: *not yet implemented*

Extends

Class "`jobjRef`", directly.

Author(s)

Simon Urbanek

See Also

`.jcall` or `jobjRef` `jrectRef` for rectangular arrays

java-tools *java tools used internally in rJava*

Description

java tools used internally in rJava

Examples

JavaAccess *Field/method operator for Java objects*

Description

The $ operator for jobjRef Java object references provides convenience access to object attributes and calling Java methods.

Usage

```
## S3 method for class 'jobjRef'
   .DollarNames(x, pattern = "" )
## S3 method for class 'jarrayRef'
 .DollarNames(x, pattern = "" )
## S3 method for class 'jrectRef'
  .DollarNames(x, pattern = "" )
## S3 method for class 'jclassName'
.DollarNames(x, pattern = "" )
```

Arguments

x	object to complete
pattern	pattern

Details

rJava provides two levels of API: low-level JNI-API in the form of .jcall function and high-level reflection API based on the $ operator. The former is very fast, but inflexible. The latter is a convenient way to use Java-like programming at the cost of performance. The reflection API is build around the $ operator on jobjRef-class objects that allows to access Java attributes and call object methods.

$ returns either the value of the attribute or calls a method, depending on which name matches first.

$<- assigns a value to the corresponding Java attribute.

names and .DollarNames returns all fields and methods associated with the object. Method names are followed by (or () depending on arity. This use of names is mainly useful for code completion, it is not intended to be used programmatically.

This is just a convenience API. Internally all calls are mapped into .jcall calls, therefore the calling conventions and returning objects use the same rules. For time-critical Java calls .jcall should be used directly.

Methods

`$ signature(x = "jobjRef"): ...`

`$ signature(x = "jclassName"): ...`

`$<- signature(x = "jobjRef"): ...`

`$<- signature(x = "jclassName"): ...`

`names signature(x = "jobjRef"): ...`

`names signature(x = "jarrayRef"): ...`

`names signature(x = "jrectRef"): ...`

`names signature(x = "jclassName"): ...`

See Also

J, .jcall, .jnew, jobjRef-class

Examples

```
v <- new(J("java.lang.String"), "Hello World!")
v$length()
v$indexOf("World")
names(v)



J("java.lang.String")$valueOf(10)

Double <- J("java.lang.Double")
# the class pseudo field - instance of Class for the associated class
# similar to java Double.class
Double$class
```

---

javaImport *Attach mechanism for java packages*

Description

The javaImport function creates an item on R's search that maps names to class names references found in one or several "imported" java packages.

Usage

```
javaImport(packages = "java.lang")
```

Arguments

packages	character vector containing java package paths

Value

An external pointer to a java specific `UserDefinedDatabase` object

Warning

This feature is experimental. Use with caution, and don't forget to detach.

Note

Currently the list of objects in the imported package is populated as new objects are found, *not* at creation time.

Author(s)

Romain Francois <francoisromain@free.fr>

References

User-Defined Tables in the R Search Path. Duncan Temple Lang. December 4, 2001 https://www.omegahat.net/RObjectTables/

See Also

`attach`

Examples

```
## Not run:
attach( javaImport( "java.util" ), pos = 2 , name = "java:java.util" )

# now we can just do something like this
v <- new( Vector )
v$add( "foobar" )
ls( pos = 2 )

# or this
m <- new( HashMap )
m$put( "foo", "bar" )
ls( pos = 2 )

# or even this :
Collections$EMPTY_MAP

## End(Not run)
```

`jcall` *Call a Java method*

Description

`.jcall` calls a Java method with the supplied arguments.

Usage

```
.jcall(obj, returnSig = "V", method, ..., evalArray = TRUE,
    evalString = TRUE, check = TRUE, interface = "RcallMethod",
    simplify = FALSE, use.true.class = FALSE)
```

Arguments

obj	Java object (jobjRef as returned by .jcall or .jnew) or fully qualified class name in JNI notation (e.g. "java/lang/String").
returnSig	Return signature in JNI notation (e.g. "V" for void, "[I" for int[] etc.). For convenience additional type "S" is supported and expanded to "Ljava/lang/String;", re-mapping "T" to represent the type short.
method	The name of the method to be called
...	Any parameters that will be passed to the Java method. The parameter types are determined automatically and/or taken from the jobjRef object. All named parameters are discarded.
evalArray	This flag determines whether the array return value is evaluated (TRUE) or passed back as Java object reference (FALSE).
simplify	If evalArray is TRUE then this argument is passed to .jevalArray().
evalString	This flag determines whether string result is returned as characters or as Java object reference.
check	If set to TRUE then checks for exceptions are performed before and after the call using .jcheck(silent=FALSE). This is usually the desired behavior, because all calls fail until an exception is cleared.
interface	This option is experimental and specifies the interface used for calling the Java method; the current implementation supports two interfaces: "RcallMethod" the default interface. "RcallSyncMethod" synchronized call of a method. This has similar effect as using synchronize in Java.
use.true.class	logical. If set to TRUE, the true class of the returned object will be used instead of the declared signature. TRUE allows for example to grab the actual class of an object when the return type is an interface, or allows to grab an array when the declared type is Object and the returned object is an array. Use FALSE for efficiency when you are sure about the return type.

Details

.jcall requires exact match of argument and return types. For higher efficiency .jcall doesn't perform any lookup in the reflection tables. This means that passing subclasses of the classes present in the method definition requires explicit casting using .jcast. Passing null arguments also needs a proper class specification with .jnull.

Java types long and float have no corresponding types in R and therefore any such parameters must be flagged as such using .jfloat and .jlong functions respectively.

Java also distinguishes scalar and array types whereas R doesn't have the concept of a scalar. In R a scalar is basically a vector (called array in Java-speak) of the length 1. Therefore passing vectors of the length 1 is ambiguous. .jcall assumes that any vector of the length 1 that corresponds to a native Java type is a scalar. All other vectors are passed as arrays. Therefore it is important

to use `.jarray` if an arbitrary vector (including those of the length 1) is to be passed as an array parameter.

Important note about encoding of character vectors: Java interface always works with strings in UTF-8 encoding, therefore the safest way is to run R in a UTF-8 locale. If that is not possible for some reason, rJava can be used in non-UTF-8 locales, but care must be taken. Since R 2.7.0 it is possible to associate encoding with strings and rJava will flag all strings it produces with the appropriate UTF-8 tag. R will then perform corresponding appropriate conversions where possible (at a cost of speed and memory usage), but 3rd party code may not (e.g. older packages). Also rJava relies on correct encoding flags for strings passed to it and will attempt to perform conversions where necessary. If some 3rd party code produces strings incorrectly flagged, all bets are off.

Finally, for performance reasons class, method and field names as well as signatures are not always converted and should not contain non-ASCII characters.

Value

Returns the result of the method.

See Also

`.jnew`, `.jcast`, `.jnull`, `.jarray`

Examples

```
.jcall("java/lang/System","S","getProperty","os.name")
if (!nzchar(Sys.getenv("NOAWT"))) {
  f <- .jnew("java/awt/Frame","Hello")
  .jcall(f,,"setVisible",TRUE)
}
```

---

jcast *Cast a Java object to another class*

Description

`.jcast` returns a Java object reference cast to another Java class.

Usage

```
.jcast(obj, new.class = "java/lang/Object", check = FALSE, convert.array = FALSE
```

Arguments

`obj`	a Java object reference
`new.class`	fully qualified class name in JNI notation (e.g. `"java/lang/String"`).
`check`	logical. If `TRUE`, it is checked that the object effectively is an instance of the new class. See `%instanceof%`. Using FALSE (the default) for this argument, rJava does not perform type check and this will cause an error on the first use if the cast is illegal.
`convert.array`	logical. If `TRUE` and the object is an array, it is converted into a `jarrayRef` reference.

Details

This function is necessary if a argument of `.jcall` or `.jnew` is defined as the superclass of the object to be passed (see `.jcall`). The original object is not modified.

The default values for the arguments `check` and `convert.array` is `FALSE` in order to guarantee backwards compatibility, but it is recommended to set the arguments to `TRUE`

Value

Returns a Java object reference (`jobjRef`) to the object `obj`, changing the object class.

See Also

`.jcall`

Examples

```
## Not run:
v <- .jnew("java/util/Vector")
.jcall("java/lang/System","I","identityHashCode",.jcast(v, "java/lang/Object"))

## End(Not run)
```

---

jcastToArray *Ensures that a given object is an array reference*

Description

`.jcastToArray` takes a Java object reference of any kind and returns Java array reference if the given object is a reference to an array.

Usage

```
.jcastToArray(obj, signature=NULL, class="", quiet=FALSE)
```

Arguments

`obj`	Java object reference to cast or a scalar vector
`signature`	array signature in JNI notation (e.g. `"[I"` for an array of integers). If set to `NULL` (the default), the signature is automatically determined from the object's class.
`class`	force the result to pose as a particular Java class. This has the same effect as using `.jcast` on the result and is provided for convenience only.
`quiet`	if set to `TRUE`, no failures are reported and the original object is returned unmodified.

Details

Sometimes a result of a method is by definition of the class `java.lang.Object`, but the actual referenced object may be an array. In that case the method returns a Java object reference instead of an array reference. In order to obtain an array reference, it is necessary to cast such an object to an array reference - this is done using the above `.jcastToArray` function.

The input is an object reference that points to an array. Usually the signature should be left at `NULL` such that it is determined from the object's class. This is also a check, because if the object's class is not an array, then the functions fails either with an error (when `quiet=FALSE`) or by returning the original object (when `quiet=TRUE`). If the signature is set to anything else, it is not verified and the array reference is always created, even if it may be invalid and unusable.

For convenience `.jcastToArray` also accepts non-references in which case it simply calls `.jarray`, ignoring all other parameters.

Value

Returns a Java array reference (`jarrayRef`) on success. If `quiet` is `TRUE` then the result can also be the original object in the case of failure.

Examples

```
## Not run:
a <- .jarray(1:10)
print(a)
# let's create an array containing the array
aa <- .jarray(list(a))
print(aa)
ba <- .jevalArray(aa)[[1]]
# it is NOT the inverse, because .jarray works on a list of objects
print(ba)
# so we need to cast the object into an array
b <- .jcastToArray(ba)
# only now a and b are the same array reference
print(b)
# for convenience .jcastToArray behaves like .jarray for non-references
print(.jcastToArray(1:10/2))

## End(Not run)
```

`jcheck` *Java exception handling*

Description

`.jcheck` checks the Java VM for any pending exceptions and clears them.

`.jthrow` throws a Java exception.

`.jgetEx` polls for any pending exceptions and returns the exception object.

`.jclear` clears a pending exception.

Usage

```
.jcheck(silent = FALSE)

.jthrow(exception, message = NULL)
.jgetEx(clear = FALSE)
.jclear()
```

Arguments

silent	If set to FALSE then Java is instructed to print the exception on stderr. Note that Windows Rgui doesn't show stderr so it will not appear there (as of rJava 0.5-1 some errors that the JVM prints using the vfprintf callback are passed to R. However, some parts are printed using System.err in which case the usual redirection using the System class can be used by the user).
exception	is either a class name of an exception to create or a throwable object reference that is to be thrown.
message	if exception is a class name then this parameter specifies the string to be used as the message of the exception. This parameter is ignored if exception is a reference.
clear	if set to TRUE then the returned exception is also cleared, otherwise the throwable is returned without clearing the cause.

Details

Please note that some functions (such as .jnew or .jcall) call .jcheck implicitly unless instructed to not do so. If you want to handle Java exceptions, you should make sure that those function don't clear the exception you may want to catch.

The exception handling is still as a very low-level and experimental, because it requires polling of exceptions. A more elaborate system using constructs similar to try ... catch is planned for next major version of rJava.

Warning: When requesting exceptions to not be cleared automatically, please note that the show method (which is called by print) has a side-effect of making a Java call to get the string representation of a Java object. This implies that it will be impeded by any pending exceptions. Therefore exceptions obtained through .jgetEx can be stored, but should not be printed (or otherwise used in Java calls) until after the exception is cleared. In general, all Java calls will fail (possibly silently) until the exception is cleared.

Value

.jcheck returns TRUE if an exception occurred or FALSE otherwise.

.jgetEx returns NULL if there are no pending exceptions or an object of the class "java.lang.Throwable" representing the current exception.

See Also

.jcall, .jnew

Examples

```
# we try to create a bogus object and
# instruct .jnew to not clear the exception
# this will raise an exception
v <- .jnew("foo/bar", check=FALSE)

# you can poll for the exception, but don't try to print it
# (see details above)
if (!is.null(e<-.jgetEx())) print("Java exception was raised")

# expect TRUE result here because the exception was still not cleared
print(.jcheck(silent=TRUE))
# next invocation will be FALSE because the exception is now cleared
print(.jcheck(silent=TRUE))

# now you can print the actual expection (even after it was cleared)
print(e)
```

---

jclassName-class *Class "jclassName" - a representation of a Java class name*

Description

This class holds a name of a class in Java.

Objects from the Class

Objects of this class should *not* be created directly. Instead, the function `J` should be used to create new objects of this class.

Slots

`name`: Name of the class (in source code notation)

`jobj`: Object representing the class in Java

Methods

The objects of class `jclassName` are used indirectly to be able to create new Java objects via `new` such as `new(J("java.lang.String"), "foo")` or to use the `$` convenience operator on static classes, such as `J("java.lang.Double")$parseDouble("10.2")`.

`as.character signature(x = "jclassName")`: returns the class name as a string vector of length one.

Author(s)

Simon Urbanek

See Also

`J`, `new`

jengine *Java callback engineCast a Java object to another class*

Description

.jengine obtains the current callback engine or starts it.

Usage

```
.jengine(start=FALSE, silent=FALSE)
```

Arguments

start	if set to TRUE then the callback engine is started if it is not yet active
silent	if set to TRUE then NULL is returned if there is no engine available. Otherwise an error is raised

Details

.jengine can be used to detect whether the engine was started or to start the engine.

Before any callbacks from Java into R can be performed, the Java callback engine must be initialized, loading Java/R Interface (JRI). If JRI was not started and start is set to TRUE then .jengine will load necessary classes and start it.

Note that JRI is an optional part of rJava and requires R shared library at the moment. By default rJava will continue with installation even if JRI cannot be built.

Value

Returns a Java object reference (jobjRef) to the current Java callback engine.

See Also

.jcall

Examples

```
## Not run:
.jengine(TRUE)

## End(Not run)
```

jequals *Comparing Java References*

Description

.jequals function can be used to determine whether two objects are equal. In addition, it allows mixed comparison of non-Java object for convenience, unless strict comparison is desired.

The binary operators == and != are mapped to (non-strict) call to .jequals for convenience.

.jcompare compares two objects in the sense of the java.lang.Comparable interface.

The binary operators <, >, <=, >= are mapped to calls to .jcompare for convenience

Usage

```
.jequals(a, b, strict = FALSE)
.jcompare( a, b )
```

Arguments

a	first object
b	second object
strict	when set to TRUE then non-references save for NULL are always treated as different, see details.

Details

.jequals compares two Java objects by calling equals method of one of the objects and passing the other object as its argument. This allows Java objects to define the 'equality' in object-dependent way.

In addition, .jequals allows the comparison of Java object to other scalar R objects. This is done by creating a temporary Java object that corresponds to the R object and using it for a call to the equals method. If such conversion is not possible a warning is produced and the result it FALSE. The automatic conversion will be avoided if strict parameter is set to TRUE.

NULL values in a or b are replaced by Java null-references and thus .jequals(NULL,NULL) is TRUE.

If neither a and b are Java objects (with the exception of both being NULL) then the result is identical to that of all.equal(a,b).

Neither comparison operators nor .jequals supports vectors and returns FALSE in that case. A warning is also issued unless strict comparison was requested.

Value

.jequals returns TRUE if both object are considered equal, FALSE otherwise.

.jcompare returns the result of the compareTo java method of the object a applied to b

Methods

!= signature(e1 = "ANY", e2 = "jobjRef"): ...

!= signature(e1 = "jobjRef", e2 = "jobjRef"): ...

!= signature(e1 = "jobjRef", e2 = "ANY"): ...

== signature(e1 = "ANY", e2 = "jobjRef"): ...

== signature(e1 = "jobjRef", e2 = "jobjRef"): ...

== signature(e1 = "jobjRef", e2 = "ANY"): ...

< signature(e1 = "ANY", e2 = "jobjRef"): ...

< signature(e1 = "jobjRef", e2 = "jobjRef"): ...

< signature(e1 = "jobjRef", e2 = "ANY"): ...

> signature(e1 = "ANY", e2 = "jobjRef"): ...

> signature(e1 = "jobjRef", e2 = "jobjRef"): ...

> signature(e1 = "jobjRef", e2 = "ANY"): ...

>= signature(e1 = "ANY", e2 = "jobjRef"): ...

>= signature(e1 = "jobjRef", e2 = "jobjRef"): ...

>= signature(e1 = "jobjRef", e2 = "ANY"): ...

<= signature(e1 = "ANY", e2 = "jobjRef"): ...

<= signature(e1 = "jobjRef", e2 = "jobjRef"): ...

<= signature(e1 = "jobjRef", e2 = "ANY"): ...

Note

Don't use `x == NULL` to check for `null`-references, because `x` could be `NULL` and thus the result would be an empty vector. Use `is.jnull` instead. (In theory `is.jnull` and `x == .jnull()` are the the same, but `is.jnull` is more efficient.)

See Also

`is.jnull`

Examples

```
s <- .jnew("java/lang/String", "foo")
.jequals(s, "foo") # TRUE
.jequals(s, "foo", strict=TRUE) # FALSE - "foo" is not a Java object
t <- s
.jequals(s, t, strict=TRUE) # TRUE

s=="foo" # TRUE



Double <- J("java.lang.Double")
d1 <- new( Double, 0.0 )
d2 <- new( Double, 1.0 )
d3 <- new( Double, 0.0 )

d1 < d2
d1 <= d3
```

```
d1 >= d3
d1 > d2

# cannot compare a Double and a String
try( d1 < "foo" )

# but can compare a Double and an Integer
d1 < 10L
```

`jfield` *Obtains the value of a field*

Description

`.jfield` returns the value of the specified field on an object.

Usage

```
.jfield(o, sig = NULL, name, true.class = is.null(sig), convert = TRUE)
`.jfield<-`(o, name, value)
```

Arguments

`o`	Class name or object (Java reference) whose field is to be accessed. Static fields are supported both by specifying the class name or using an instance.
`sig`	signature (JNI type) of the field. If set to `NULL` rJava attempts to determine the signature using reflection. For efficiency it is recommended to specify the signature, because the reflection lookup is quite expensive.
`name`	name of the field to access
`true.class`	by default the class of the resulting object matches the signature of the field. Setting this flag to `TRUE` causes `.jfield` to use true class name of the resulting object instead. (this flag has no effect on scalar fields)
`convert`	when set to `TRUE` all references are converted to native types (where possible). Otherwise Java references are returned directly.
`value`	value to assign into the field. The field signature is determined from the value in the same way that parameter signatures are determined in `.jcall` - be sure to cast the value as necessary, no automatic conversion is done.

Details

The detection of a field signature in `.jfield` using reflection is considerably expensive (more than 3 additional method calls have to be performed), therefore it is recommended for time-critical code to specify the field signature beforehand.

NOTE: The sequence of arguments in `.jfield` has been changed since rJava 0.5 to be more consistent and match the sequence in `.jcall`. Also `.jsimplify` is no longer needed as primitive types are obtained directly.

Value

`.jfield`: contents of the field, `.jfield<-`: modified object.

See Also

`.jcall`

Examples

```
## Not run:
.jfield("java/lang/Boolean",, "TYPE")

## End(Not run)
```

---

jfloat *Wrap numeric vector as flat Java parameter*

Description

`.jfloat` marks a numeric vector as an object that can be used as parameter to Java calls that require `float` parameters. Similarly, `.jlong` marks a numeric vector as `long` parameter, `.jshort` as `short` and `.jbyte` as `byte`.

Usage

```
.jfloat(x)
.jlong(x)
.jbyte(x)
.jchar(x)
.jshort(x)
```

Arguments

`x`	numeric vector

Details

R has no native `float` or `long` type. Numeric vectors are stored as `doubles`, hence there is no native way to pass float numbers to Java methods. The `.jfloat` call marks a numeric vector as having the Java type `float` by wrapping it in the `jfloat` class. The class is still a subclass of `numeric`, therefore all regular R operations are unaffected by this.

Similarly, `.jlong` is used to mark a numeric vector as a parameter of the `long` Java type. Please note that in general R has no native type that will hold a `long` value, so conversion between Java's `long` type and R's numeric is potentially lossy.

`.jbyte` is used when a scalar byte is to be passed to Java. Note that byte arrays are natively passed as raw vectors, not as `.jbyte` arrays, although non-scalar `.jbyte` is equivalent except for using four-times as much memory.

`.jchar` is strictly experimental and uses integer vector as storage class. The type `char` in Java represents 16-bit Unicode code points (not to be confused with `char` in C which is `byte` in Java!), see Java documentation for details. `x` can also be a non-`NA` string in which case `.jchar(x)` is just a shorthand for `.jnew("java.lang.String", x)$toCharArray()` and thus performs a Java call (unlike all other functions mentioned here).

Value

Returns a numeric vector of the class `jfloat`, `jlong`, `jbyte`, `jshort` or `jchar` that can be used as parameter to Java calls that require `float`, `long`, `byte`, `short` or `char` parameters respectively.

See Also

`.jcall`, `jfloat-class`

`jfloat-class` *Classes "jfloat", "jlong", "jbyte" and "jchar" specify Java native types that are not native in R*

Description

These classes wrap a numeric vector to be treated as `float` or `long` argument when passed to Java and an integer vector to be treated as `byte` or `char`. R doesn't distinguish between `double` and `float`, but Java does. In order to satisfy object types, numeric vectors that should be converted to floats or long on the Java side must be wrapped in this class. In addition `jbyte` must be used when passing scalar byte (but not byte arrays, those are mapped into RAW vectors). Finally `jchar` it used when mapping integer vectors into unicode Java character vectors.

Objects from the Class

Objects can be created by calling `.jfloat`, `.jlong`, `.jbyte` or `.jchar` respectively.

Slots

`.Data`**:** Payload

Extends

"jfloat" and "jlong": Class `"numeric"`, from data part. Class `"vector"`, by class `"numeric"`.

"jbyte" and "jchar": Class `"integer"`, from data part. Class `"vector"`, by class `"integer"`.

Methods

"jfloat" and "jlong" have no methods other than those inherited from "numeric". "jbyte" and "jchar" have no methods other than those inherited from "integer".

Author(s)

Simon Urbanek

See Also

`.jfloat`, `.jlong`, `.jbyte`, `.jchar` and `.jcall`

`jinit` *Initialize Java VM*

Description

`.jinit` initializes the Java Virtual Machine (JVM). This function must be called before any rJava functions can be used.

`.jvmState()` returns the state of the current JVM.

Usage

```
.jinit(classpath = NULL, parameters = getOption("java.parameters"), ...,
silent = FALSE, force.init = FALSE)
.jvmState()
```

Arguments

`classpath`	Any additional classes to include in the Java class paths (i.e. locations of Java classes to use). This path will be prepended to paths specified in the `CLASSPATH` environment variable. Do NOT set this system class path initializing a package, use `.jpackage` instead, see details.
`parameters`	character vector of parameters to be passed to the virtual machine. They are implementation dependent and apply to JDK version 1.2 or higher only. Please note that each parameter must be in a separate element of the array, you cannot use a space-separated string with multiple parameters.
`...`	Other optional Java initialization parameters (implementation-dependent).
`silent`	If set to `TRUE` no warnings are issued.
`force.init`	If set to `TRUE` JVM is re-initialized even if it is already running.

Details

Starting with version 0.5 rJava provides a custom class loader that can automatically track classes and native libraries that are provided in R packages. Therefore R packages should NOT use `.jinit`, but call `.jpackage` instead. In addition this allows the use of class path modifying function `.jaddClassPath`.

Important note: if a class is found on the system class path (i.e. on the `classpath` specified to `.jinit`) then the system class loader is used instead of the rJava loader, which can lead to problems with reflection and native library support is not enabled. Therefore it is highly recommended to use `.jpackage` or `.jaddClassPath` instead of `classpath` (save for system classes).

Stating with version 0.3-8 rJava is now capable of modifying the class path on the fly for certain Sun-based Java virtual machines, even when attaching to an existing VM. However, this is done by exploiting the way ClassLoader is implemented and may fail in the future. In general it is officially not possible to change the class path of a running VM.

At any rate, it is impossible to change any other VM parameters of a running VM, so when using `.jinit` in a package, be generous with limits and don't use VM parameters to unnecessarily restrict resources (or preferably use `.jpackage` instead). JVM parameters can only be set if the initial state of the JVM is `"none"`.

There is a subtle difference between `"initialized"` and the JVM state. It is in theory possible for `"initialized"` to be `FALSE` and still `"state"` to be `"created"` or `"attached"` in

case where JVM was created but rJava has not been able to initialize for other reasons, although such state should be rare and problematic in either case. Behavior of rJava functions other than `.jinit` and `.jvmState` is undefined unless `.jvmState()$initialized` is `TRUE`.

Value

The return value is an integer specifying whether and how the VM was initialized. Negative values indicate failure, zero denotes successful initialization and positive values signify partially successful initilization (i.e. the VM is up, but parameters or class path could not be set due to an existing or incompatible VM).

`.jvmState` returns a named list with at least the following elements:

`initialized`	`TRUE` if rJava is initialized and has a runing JVM, `FALSE` otherwise.
`state`	string representing the current state of the JVM. One of the following values: `"none"` if there is no JVM, `"created"` if the current JVM has been created by rJava, `"attached"` if rJava attached into an existing JVM (typically when R is embedded into a running JVM via JRI), `"detached"` if there is a JVM (such as embedded R), but rJava has not been initialized to use it, `"dead"` if the process is about to die due to the JVM forcing en exit or `"destroyed"` if a JVM existed before, but was destroyed.

See Also

`.jpackage`

Examples

```
## Not run:
## set heap size limit to 512MB (see java -X) and
## use "myClasses.jar" as the class path
.jinit(classpath="myClasses.jar", parameters="-Xmx512m")
.jvmState()

## End(Not run)
```

---

jmemprof *rJava memory profiler*

Description

`.jmemprof` enables or disables rJava memory profiling. If rJava was compiled without memory profiling support, then a call to this function always causes an error.

Usage

```
.jmemprof(file = "-")
```

Arguments

`file`	file to write profiling information to or `NULL` to disable profiling

Details

The `file` parameter must be either a filename (which will be opened in append-mode) or "-" to use standard output or `NULL` to disable profiling. An empty string "" is equivalent to `NULL` in this context.

Note that lots of finalizers are run only when R exists, so usually you want to enable profiling early and let R exit to get a sensible profile. Running gc may be helpful to get rid of references that can be collected in R.

A simple perl script is provided to analyze the result of the profiler. Due to its simple text format, it is possible to capture entire stdout including the profiler information to have both the console context for the allocations and the profile. Memory profiling is also helpful if rJava debug is enabled.

Note that memory profiling support must be compiled in rJava and it is by default compiled only if debug mode is enabled (which is not the case by default).

Value

Returns `NULL`.

Examples

```
## memory profiling support is optional so only works when enabled
tryCatch(
.jmemprof("rJava.mem.profile.txt"),
error=function(e) message(e))
```

---

jnew *Create a Java object*

Description

`.jnew` create a new Java object.

Usage

```
.jnew(class, ..., check=TRUE, silent=!check, class.loader=NULL)
```

Arguments

`class`	fully qualified class name in JNI notation (e.g. `"java/lang/String"`).
`...`	Any parameters that will be passed to the corresponding constructor. The parameter types are determined automatically and/or taken from the `jobjRef` object. For details see `.jcall`. Note that all named parameters are discarded.
`check`	If set to `TRUE` then `.jcheck` is invoked before and after the call to the constructor to clear any pending Java exceptions.
`silent`	If set to `FALSE` then `.jnew` will fail with an error if the object cannot be created, otherwise a null-reference is returned instead. In addition, this flag is also passed to final `.jcheck` if `check` above is set to `TRUE`. Note that the error handling also clears exceptions, so `check=FALSE, silent=FALSE` is usually not a meaningful combination.

`class.loader` optional class loader to force for loading the class. If not set, the rJava class loader is used first. The default Java class loader is always used as a last resort. Set to `.rJava.class.loader` inside a package if it uses its own class loader (see `.jpackage` for details).

Value

Returns the reference (`jobjRef`) to the newly created object or `null`-reference (see `.jnull`) if something went wrong.

See Also

`.jcall`, `.jnull`

Examples

```
## Not run:
f <- .jnew("java/awt/Frame","Hello")
.jcall(f,,"setVisible",TRUE)

## End(Not run)
```

---

jnull *Java null object reference*

Description

`.jnull` returns a `null` reference of a specified class type. The resulting object is of the class `jobjRef`.

`is.jnull` is an extension of `is.null` that also returns `TRUE` if the supplied object is a `null` Java reference.

Usage

```
.jnull(class = "java/lang/Object")
is.jnull(x)
```

Arguments

`class`	fully qualified target class name in JNI notation (e.g. `"java/lang/String"`).
`x`	object to check

Details

`.jnull` is necessary if `null` is to be passed as an argument of `.jcall` or `.jnew`, in order to be able to find the correct method/constructor.

Example: given the following method definitions of the class `A`:

- `public static void run(String a);`
- `public static void run(Double n);`

Calling `.jcall("A",,"run",NULL)` is ambiguous, because it is unclear which method is to be used. Therefore rJava requires class information with each argument to `.jcall`. If we wanted to run the String-version, we could use `.jcall("A",,"run",.jnull("java/lang/String"))`.

`is.jnull` is a test that should be used to determine whether a given Java reference is a `null` reference.

Value

`.jnull` returns a Java object reference (`jobjRef`) of a `null` object having the specified object class.

`is.jnull` returns `TRUE` if `is.null(x)` is `TRUE` or if `x` is a Java `null` reference.

See Also

`.jcall`, `.jcast`

Examples

```
## Not run:
.jcall("java/lang/System","I","identityHashCode",.jnull())

## End(Not run)
```

---

jobjRef-class *Class "jobjRef" - Reference to a Java object*

Description

This class describes a reference to an object held in a JavaVM.

Objects from the Class

Objects of this class should *not* be created directly. Instead, the function `.jnew` should be use to create new Java objects. They can also be created as results of the `.jcall` function.

Slots

`jobj`**:** Internal identifier of the object (external pointer to be precise)

`jclass`**:** Java class name of the object (in JNI notation)

Java-side attributes are not accessed via slots, but the `$` operator instead.

Methods

This object's Java methods are not accessed directly. Instead, `.jcall` JNI-API should be used for invoking Java methods. For convenience the `$` operator can be used to call methods via reflection API.

Author(s)

Simon Urbanek

See Also

.jnew, .jcall or jarrayRef-class

---

jpackage *Initialize an R package containing Java code*

Description

.jpackage initializes the Java Virtual Machine (JVM) for an R package. In addition to starting the JVM it also registers Java classes and native code contained in the package with the JVM. function must be called before any rJava functions can be used.

Usage

```
.jpackage(name, jars='*', morePaths='', nativeLibrary=FALSE,
          lib.loc=NULL, parameters = getOption("java.parameters"),
          own.loader = FALSE)
```

Arguments

name	name of the package. It should correspond to the pkgname parameter of .onLoad or .First.lib function.
jars	Java archives in the java directory of the package that should be added to the class path. The paths must be relative to package's java directory. A special value of '*' adds all .jar files from the java the directory.
morePaths	vector listing any additional entries that should be added to the class path.
nativeLibrary	a logical determining whether rJava should look for native code in the R package's shared object or not.
lib.loc	a character vector with path names of R libraries, or NULL (see system.file and examples below).
parameters	optional JVM initialization parameters which will be used if JVM is not initilized yet (see .jinit).
own.loader	if TRUE then a new, separate class loader will be initilized for the package and assigned to the .pkg.class.loader variable in the package namespace. New packages should make use of this feature.

Details

.jpackage initializes a Java R package as follows: first the JVM is initialized via .jinit (if it is not running already). Then the java directory of the package is added to the class path. Then .jpackage prepends jars with the path to the java directory of the package and adds them to the class path (or all .jar files if '*' was specified). Finally the morePaths parameter (if set) is passed to a call to .jaddClassPath.

Therefore the easiest way to create a Java package is to add .jpackage(pkgname, lib.loc=libname) in .onLoad or .First.lib, and copy all necessary classes to a JAR file(s) which is placed in the inst/java/ directory of the source package.

If a package needs special Java parameters, "java.parameters" option can be used to set them on initialization. Note, however, that Java parameters can only be used during JVM initialization and other package may have intialized JVM already.

Since rJava 0.9-14 there is support of package-specific class loaders using the own.loader=TRUE option. This is important for packages that may be using classes that conflict with other packages are therefore is highly recommended for new packages. Before this feature, there was only one global class loader which means that the class path was shared for all class look ups. If two packages use the same (fully qualified) class name, even in a dependency, they are likely to clash with each if they don't use exactly the same version. Therefore it is safer for each package use use a private class loader for its classes to guarantee that the only the classes supplied with the package will be used. To do that, a package will set own.loader=TRUE which instructs rJava to not change the global loader, but instead create a separate one for the package and assign it to .rJava.class.loader in the package namespace. Then if package wants to instantiate a new class, it would use .jnew("myClass", class.loader=.rJava.class.loader) to use its own loader instead of the global one. The global loader's class path won't be touched, so it won't find the package's classes. It is possible to get the loader used in a package using .jclassLoader(package="foo") which will return the global one if the package has not registered its own. Similarly, to retrieve the class path used by a package, one would use .jclassPath(.jclassLoader(package="foo")).

Note that with the advent of multiple class loaders the value of the java.class.path property is no longer meaningful as it can reflect only one of the loaders.

Value

The return value is an invisible TRUE if the initialization was successful.

See Also

.jinit

Examples

```
## Not run:
.onLoad <- function(libname, pkgname) {
  .jpackage(pkgname, lib.loc=libname, own.loader=TRUE)
  ## do not use, just an illustration of the concept:
  cat("my Java class path: ")
  print(.jclassPath(.jclassLoader(package=pkgname)))
}

## End(Not run)
```

---

jrectRef-class *Rectangular java arrays*

Description

References to java arrays that are guaranteed to be rectangular, i.e similar to R arrays

Objects from the Class

Objects of this class should *not* be created directly. Instead, they usually come as a result of a java method call.

Slots

`jsig`: JNI signature of the array type

`jobj`: Internal identifier of the object

`jclass`: Inherited from `jobjRef`, but unspecified

`dimension`: dimension vector of the array

Extends

Class "`jarrayRef`", directly. Class "`jobjRef`", by class "jarrayRef", distance 2.

Methods

length `signature(x = "jrectRef")`: The number of elements in the array. Note that if the array has more than one dimension, it gives the number of arrays in the first dimension, and not the total number of atomic objects in the array (like R does). This gives what would be returned by `array.length` in java.

str `signature(object = "jrectRef")`: ...

[`signature(x = "jrectRef")`: R indexing of rectangular java arrays

dim `signature(x = "jrectRef")`: extracts the dimensions of the array

dim<- `signature(x = "jrectRef")`: sets the dimensions of the array

unique `signature(x = "jrectRef")`: unique objects in the array

duplicated `signature(x = "jrectRef")`: see `duplicated`

anyDuplicated `signature(x = "jrectRef")`: see `anyDuplicated`

sort `signature(x = "jrectRef")`: returns a *new* array with elements from x in order

rev `signature(x = "jrectRef")`: returns a *new* array with elements from x reversed

min `signature(x = "jrectRef")`: the smallest object in the array (in the sense of the Comparable interface)

max `signature(x = "jrectRef")`: the biggest object in the array (in the sense of the Comparable interface)

range `signature(x = "jrectRef")`: the range of the array (in the sense of the Comparable interface)

Examples

```
v <- new( J("java.util.Vector") )
v$add( "hello" )
v$add( "world" )
v$add( new( J("java.lang.Double"), "10.2" ) )
array <- v$toArray()

array[ c(TRUE,FALSE,TRUE) ]
array[ 1:2 ]
array[ -3 ]

# length
length( array )


# also works as a pseudo field as in java
```

```
array$length
```

`jreflection` *Simple helper functions for Java reflection*

Description

`.jconstructors` returns a character vector with all constructors for a given class or object. `.jmethods` returns a character vector with all methods for a given class or object. `.jfields` returns a character vector with all fields (aka attributes) for a given class or object.

Usage

```
.jconstructors(o, as.obj = FALSE, class.loader=.rJava.class.loader)
.jmethods(o, name = NULL, as.obj = FALSE, class.loader=.rJava.class.loader)
.jfields(o, name = NULL, as.obj = FALSE, class.loader=.rJava.class.loader)
```

Arguments

`o`	Name of a class (either notation is fine) or an object whose class will be queried
`name`	string, regular expression of the method/field to look for
`as.obj`	if `TRUE` then a list of Java objects is returned, otherwise a character vector (obtained by calling `toString()` on each entry).
`class.loader`	optional, class loader to use for class look up if needed (i.e., if `o` is a string)

Details

There first two functions are intended to help with finding correct signatures for methods and constructors. Since the low-level API in rJava doesn't use reflection automatically, it is necessary to provide a proper signature. That is somewhat easier using the above methods.

Value

Returns a character vector (if `as.obj` is `FALSE`) or a list of Java objects. Each entry corresponds to the `Constructor` resp. `Method` resp. `Field` object. The string result is constructed by calling `toString()` on the objects.

See Also

`.jcall`, `.jnew`, `.jcast` or `$,jobjRef-method`

Examples

```
## Not run:
.jconstructors("java.util.Vector")
v <- .jnew("java.util.Vector")
.jmethods(v, "add")

## End(Not run)
```

`jserialize` *Java object serialization*

Description

`.jserialize` serializes a Java object into raw vector using Java serialization.

`.junserialize` re-constructs a Java object from its serialized (raw-vector) form.

`.jcache` updates, retrieves or removes R-side object cache which can be used for persistent storage of Java objects across sessions.

Usage

```
.jserialize(o)
.junserialize(data)
.jcache(o, update=TRUE)
```

Arguments

`o`	Java object
`data`	serialized Java object as a raw vector
`update`	must be `TRUE` (cache is updated), `FALSE` (cache is retrieved) or `NULL` (cache is deleted).

Details

Not all Java objects support serialization, see Java documentation for details. Note that Java serialization and serialization of R objects are two entirely different mechanisms that cannot be interchanged. `.jserialize` and `.junserialize` can be used to access Java serialization facilities.

`.jcache` manipulates the R-side Java object cache associated with a given Java reference:

Java objects do not persist across sessions, because the Java Virtual Machine (JVM) is destroyed when R is closed. All saved Java object references will be restored as `null` references, since the corresponding objects no longer exist (see R documentation on serialization). However, it is possible to serialize a Java object (if supported by the object) and store its serialized form in R. This allows for the object to be deserialized when loaded into another active session (but see notes below!)

R-side cache consists of a serialized form of the object as raw vector. This cache is attached to the Java object and thus will be saved when the Java object is saved. rJava provides an automated way of deserializing Java references if they are `null` references and have a cache attached. This is done on-demand basis whenever a reference to a Java object is required.

Therefore packages can use `.jcache` to provide a way of creating Java references that persist across sessions. However, they must be very cautious in doing so. First, make sure the serialized form is not too big. Storing whole datasets in Java serialized form will hog immense amounts of memory on the R side and should be avoided. In addition, be aware that the cache is just a snapshot, it doesn't change when the referenced Java object is modified. Hence it is most useful only for references that are not modified outside R. Finally, internal references to other Java objects accessible from R are not retained (see below). Most common use of `.jcache` is with Java references that point to definitions of methods (e.g., models) and other descriptive objects which are then used by other, active Java classes to act upon. Caching of such active objects is not a good idea, they should be instantiated by functions that operate on the descriptive references instead.

Important note: the serialization of Java references does NOT take into account any dependencies on the R side. Therefore if you hold a reference to a Java object in R that is also referenced by the serialized Java object on the Java side, then this relationship cannot be retained upon restore. Instead, two copies of disjoint objects will be created which can cause confusion and erroneous behavior.

The cache is attached to the reference external pointer and thus it is shared with all copies of the same reference (even when changed via `.jcast` etc.), but it is independent of other references to the object obtained separately (e.g., via `.jcall` or `.jfield`).

Also note that deserialization (even automated one) requires a running virtual machine. Therefore you must make sure that either `.jinit` or `.jpackage` is used before any Java references are accessed.

Value

`.jserialize` returns a raw vector

`.junserialize` returns a Java object or `NULL` if an error occurred (currently you may use `.jcheck()` to further investigate the error)

`.jcache` returns the current cache (usually a raw vector) or `NULL` if there is no cache.

`jsimplify` *Converts Java object to a simple scalar if possible*

Description

`.jsimplify` attempts to convert Java objects that represent simple scalars into corresponding scalar representation in R.

Usage

```
.jsimplify(o, promote=FALSE)
```

Arguments

`o`	arbitrary object
`promote`	logical, if `TRUE` then an ambiguous conversion where the native type value would map to `NA` (e.g., Java `int` type with value -2147483648) will be taken to represent an actual value and will be promoted to a larger type that can represent the value (in case of `int` promoted to `double`). If `FALSE` then such values are assumed to represent `NA`s.

Details

If `o` is not a Java object reference, `o` is returned as-is. If `o` is a reference to a scalar object (such as single integer, number, string or boolean) then the value of that object is returned as R vector of the corresponding type and length one.

This function is used by `.jfield` to simplify the results of field access if required.

Currently there is no function inverse to this, the usual way to wrap scalar values in Java references is to use `.jnew` as the corresponding constructor.

Value

Simple scalar or `o` unchanged.

See Also

`.jfield`

Examples

```
## Not run:
i <- .jnew("java/lang/Integer", as.integer(10))
print(i)
print(.jsimplify(i))

## End(Not run)
```

`loader` *Java Class Loader*

Description

`.jaddClassPath` adds directories or JAR files to the class path.

`.jclassPath` returns a vector containing the current entries in the class path

Usage

```
.jaddClassPath(path, class.loader=.rJava.class.loader)
.jclassPath(class.loader=.rJava.class.loader)
.jclassLoader(package=NULL)
```

Arguments

`path`	character string vector listing the paths to add to the class path
`class.loader`	Java class loader to use for the query of madification. Defaults to global class loader.
`package`	string, name of a package or `NULL` for the global class loader

Details

Whenever a class needs to be instantiated in Java it is referred by name which is used to locate a file with the bytecode for the class. The mechanism to map a name to an actual bytecode to load ind instantiate is habdled by the Java class loader. It typically keeps a list of directories and JAR files to search for the class names.

The `.jaddClassPath()` function allows the user to append new locations to the list of places which will be searched. The function `.jclassPath` retrieves the current sarch list from the loader.

When rJava is initialized, it instantiates the global class loader which is responsible for finding classes in functions such as `.jnew()`. In addition to the global class loader, R packages can create their own class loaders to avoid conflicts between packages such that they can be sure to use their own files to look for classes. See `.jpackage` for details on how that works. If the `package`

argument is supplied `.jclassLoader` will look in that package to see if it has a custom loader and will return it, otherwise it returns the global loader. Note that is will fail with an error when supplied a non-existing package name.

If you want to trace issues related to missing classes, you can enable debugging in the class loader by using the `setDebug` method, for example: `.jclassLoader()$setDebug(1L)`

Value

`.jclassPath` returns a character vector listing the class path sequence.

See Also

`.jpackage`

Examples

```
## Not run:
.jaddClassPath("/my/jars/foo.jar","/my/classes/")
print(.jclassPath())

## End(Not run)
```

---

new *Create a new Java object*

Description

Creates a new Java object and invokes the constructor with given arguments.

Details

The `new` method is used as the high-level API to create new Java objects (for low-level access see `.jnew`). It returns the newly created Java object.

`...` arguments are passed to the constructor of the class specified as `J("class.name")`.

Methods

`new signature(Class = "jclassName"): ...`

See Also

`.jnew`, `jclassName-class`

Examples

```
## Not run:
v <- new(J("java.lang.String"), "Hello World!")
v$length()
v$indexOf("World")
names(v)

## End(Not run)
```

rep *Creates java arrays by cloning*

Description

Creates a java array by cloning a reference several times

Methods

rep `signature(object = "jobjRef")`: ...

rep `signature(object = "jarrayRef")`: ...

rep `signature(object = "jrectRef")`: ...

See Also

`rep` or `.jarray`

Examples

```
if (!nzchar(Sys.getenv("NOAWT"))) {
  p <- .jnew( "java.awt.Point" )
  a <- rep( p, 10 )

  stopifnot( dim(a) == c(10L ) )
  a[[1]]$move( 10L, 50L )
  stopifnot( a[[2]]$getX() == 0.0 )
}
```

show *Show a Java Object Reference*

Description

Display a Java object reference in a descriptive, textual form. The default implementation calls `toString` Java method to obtain object's printable value and uses calls `show` on the resulting string garnished with additional details.

Methods

show `signature(object = "jobjRef")`: ...

show `signature(object = "jarrayRef")`: ...

show `signature(object = "jclassName")`: ...

str `signature(object = "jobjRef")`: currently identical to show

toJava *Convert R objects to REXP references in Java*

Description

toJava takes an R object and creates a reference to that object in Java. This reference can then be passed to Java methods such that they can refer to it back in R. This is commonly used to pass functions to Java such that Java code can call those functions later.

Usage

```
toJava(x, engine = NULL)
```

Arguments

x	R object to reference. It can be any R object and it will be retained at least for the duration of the reference on the Java side.
engine	REngine in which the reference is to be created. If <code>null</code> then the last created engine is used. This must be a Java object and a subclass of org.rosuda.REngine (and NOT the old org.rosuda.JRI.Rengine!).

Value

There result is a Java reference (jobjRef) of the Java class REXPReference.

Examples

```
## Not run:
  .jinit()
  # requires JRI and REngine classes
  .jengine(TRUE)
  f <- function() { cat("Hello!\n"); 1 }
  fref <- toJava(f)
  # to use this in Java you would use something like:
  # public static REXP call(REXPReference fn) throws REngineException, REXPMismatchExcept
  #  return fn.getEngine().eval(new REXPLanguage(new RList(new REXP[] { fn })), null, fal
  # }
  # .jcall("Call","Lorg/rosuda/REngine/REXP;","call", fref)

## End(Not run)
```

with.jobjRef *with and within methods for Java objects and class names*

Description

Convenience wrapper that allow calling methods of Java object and classes from within the object (or class).

Usage

```
## S3 method for class 'jobjRef'
with(data, expr, ...)
## S3 method for class 'jobjRef'
within(data, expr, ...)

## S3 method for class 'jarrayRef'
with(data, expr, ...)
## S3 method for class 'jarrayRef'
within(data, expr, ...)

## S3 method for class 'jclassName'
with(data, expr, ...)
## S3 method for class 'jclassName'
within(data, expr, ...)
```

Arguments

`data`	A Java object reference or a java class name. See `J`
`expr`	R expression to evaluate
`...`	ignored

Details

The expression is evaluated in an environment that contains a mapping between the public fields and methods of the object.

The methods of the object are mapped to standard R functions in the environment. In case of classes, only static methods are used.

The fields of the object are mapped to active bindings (see makeActiveBinding) so that they can be accessed and modified from within the environment. For classes, only static fields are used.

Value

`with` returns the value of the expression and `within` returns the `data` argument

Author(s)

Romain Francois <francoisromain@free.fr>

References

the `java.lang.reflect` package: https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/package-summary.html

Examples

```
if (!nzchar(Sys.getenv("NOAWT"))) {
  p <- .jnew( "java/awt/Point", 0L, 0L )
  with( p, {
# x and y and now 0
move( 10L, 10L )
# x and y are now 10
```

```
x <- x + y
  } )

  f <- within( .jnew( "javax/swing/JFrame" ) , {
layout <- .jnew( "java/awt/BorderLayout" )
setLayout( layout )
add( .jnew( "javax/swing/JLabel", "north" ), layout$NORTH )
add( .jnew( "javax/swing/JLabel", "south" ), layout$SOUTH )
add( .jnew( "javax/swing/JLabel", "west" ), layout$WEST )
add( .jnew( "javax/swing/JLabel", "east" ), layout$EAST )
setSize( .jnew( "java/awt/Dimension", 400L, 400L ) )
setVisible( TRUE )
  } )
}

Double <- J("java.lang.Double")
with( Double, MIN_VALUE )
with( Double, parseDouble( "10.2" ) )

## Not run:
# inner class example
HashMap <- J("java.util.HashMap")
with( HashMap, new( SimpleEntry, "key", "value" ) )
with( HashMap, SimpleEntry )

## End(Not run)

with( J("java.lang.System"), getProperty("java.home") )
```

Index