**Introduction**

In this part of the primer we discuss creating and using custom .jar archives within our R scripts and packages, handling of Java exceptions from R and a quick look at performance comparison between the low and high-level interfaces provided by rJava.

In the [first part](https://jozefhajnala.gitlab.io/r/r901-primer-java-from-r-1/) we talked about using the rJava package to create objects, call methods and work with arrays, we examined the various ways to call Java methods and calling Java code from R directly via execution of shell commands.



R <3 Java, or maybe not?

**Contents**

1. [Using rJava with custom built classes](https://jozefhajnala.gitlab.io/r/r902-primer-java-from-r-2/#using-rjava-with-custom-built-classes)
2. [Very quick look at performace](https://jozefhajnala.gitlab.io/r/r902-primer-java-from-r-2/#very-quick-look-at-performace)
3. [Usage of jars in R packages](https://jozefhajnala.gitlab.io/r/r902-primer-java-from-r-2/#usage-of-jars-in-r-packages)
4. [Handling Java exceptions in R](https://jozefhajnala.gitlab.io/r/r902-primer-java-from-r-2/#handling-java-exceptions-in-r)
5. [References](https://jozefhajnala.gitlab.io/r/r902-primer-java-from-r-2/#references)

**Using rJava with custom built classes**

**Preparing a .jar archive for use**

Getting back to our example with running the main method of our HelloWorldDummy class from the [first part of this primer](https://jozefhajnala.gitlab.io/r/r901-primer-java-from-r-1/), in practice we most likely want to actually create objects and invoke methods for such classes rather than simply call the main method.

For our resources to be available to rJava, we need to create a .jar archive and add it to the class path. An example of the process can be as follows. Compile our code to create the class file, and jar it:

$ javac DummyJavaClassJustForFun/HelloWorldDummy.java

$ cd DummyJavaClassJustForFun/

$ jar cvf HelloWorldDummy.jar HelloWorldDummy.class

**Adding the .jar file to the class path**

Within R, attach rJava, initialize the JVM and investigate our current class path using .jclassPath:

library(rJava)

.jinit()

.jclassPath()

Now, we add our newly created .jar to the class path using .jaddClassPath:

.jaddClassPath(paste0(jardir, "HelloWorldDummy.jar"))

If this worked, we can see the added jar(s) in the class path if we call .jclassPath() again.

**Creating objects, investigating methods and fields**

Now that we have our .jar in the class path, we can create a new Java object from our class:

dummyObj <- .jnew("DummyJavaClassJustForFun/HelloWorldDummy")

str(dummyObj)

## Formal class 'jobjRef' [package "rJava"] with 2 slots

## ..@ jobj :

## ..@ jclass: chr "DummyJavaClassJustForFun/HelloWorldDummy"

We can also investigate the available constructors, methods and fields for our class (or provide the object as argument, then its class will be queried):

* .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
* .DollarNames returns all fields and methods associated with the object. Method names are followed by ( or () depending on arity.

# Requesting vectors of methods, constructors and fields by class

.jmethods("DummyJavaClassJustForFun/HelloWorldDummy")

## [1] "public static void DummyJavaClassJustForFun.HelloWorldDummy.main(java.lang.String[])"

## [2] "public java.lang.String DummyJavaClassJustForFun.HelloWorldDummy.SayMyName()"

## [3] "public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException"

## [4] "public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException"

## [5] "public final void java.lang.Object.wait() throws java.lang.InterruptedException"

## [6] "public boolean java.lang.Object.equals(java.lang.Object)"

## [7] "public java.lang.String java.lang.Object.toString()"

## [8] "public native int java.lang.Object.hashCode()"

## [9] "public final native java.lang.Class java.lang.Object.getClass()"

## [10] "public final native void java.lang.Object.notify()"

## [11] "public final native void java.lang.Object.notifyAll()"

.jconstructors("DummyJavaClassJustForFun/HelloWorldDummy")

## [1] "public DummyJavaClassJustForFun.HelloWorldDummy()"

.jfields("DummyJavaClassJustForFun/HelloWorldDummy")

## NULL

# Requesting vectors of methods, constructors and fields by object

.jmethods(dummyObj)

## [1] "public static void DummyJavaClassJustForFun.HelloWorldDummy.main(java.lang.String[])"

## [2] "public java.lang.String DummyJavaClassJustForFun.HelloWorldDummy.SayMyName()"

## [3] "public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException"

## [4] "public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException"

## [5] "public final void java.lang.Object.wait() throws java.lang.InterruptedException"

## [6] "public boolean java.lang.Object.equals(java.lang.Object)"

## [7] "public java.lang.String java.lang.Object.toString()"

## [8] "public native int java.lang.Object.hashCode()"

## [9] "public final native java.lang.Class java.lang.Object.getClass()"

## [10] "public final native void java.lang.Object.notify()"

## [11] "public final native void java.lang.Object.notifyAll()"

.jconstructors(dummyObj)

## [1] "public DummyJavaClassJustForFun.HelloWorldDummy()"

.jfields(dummyObj)

## NULL

**Calling methods 3 different ways**

We can now invoke our SayMyName method on this object in the three ways as discussed is [the first part of this primer](https://jozefhajnala.gitlab.io/r/r901-primer-java-from-r-1/):

# low level

lres <- .jcall(dummyObj, "Ljava/lang/String;", "SayMyName")

# high level

hres <- J(dummyObj, method = "SayMyName")

# convenient $ shorthand

dres <- dummyObj$SayMyName()

c(lres, hres, dres)

## [1] "My name is DummyJavaClassJustForFun.HelloWorldDummy"

## [2] "My name is DummyJavaClassJustForFun.HelloWorldDummy"

## [3] "My name is DummyJavaClassJustForFun.HelloWorldDummy"

**Very quick look at performace**

The low-level is much faster, since J has to use reflection to find the most suitable method. The $ seems to be the slowest, but also very convenient, as it supports code completion:

microbenchmark::microbenchmark(times = 1000

, .jcall(dummyObj, "Ljava/lang/String;", "SayMyName")

, J(dummyObj, "SayMyName")

, dummyObj$SayMyName()

)

## Unit: microseconds

## expr min lq

## .jcall(dummyObj, "Ljava/lang/String;", "SayMyName") 34.992 48.0385

## J(dummyObj, "SayMyName") 696.860 748.9630

## dummyObj$SayMyName() 894.576 963.1035

## mean median uq max neval

## 62.13235 61.3425 66.6470 723.719 1000

## 903.54615 786.2715 841.2835 66191.562 1000

## 1093.67310 1009.4020 1075.2565 6743.220 1000

**Usage of jars in R packages**

To use rJava within an R package, Simon Urbanek, the author of rJava even provides a convenience function for this purpose which initializes the JVM and registers Java classes and native code contained in the package with it. A quick step by step guide to use .jars within a package is as follows:

1. place our .jars into inst/java/
2. add Depends: rJava and SystemRequirements: Java into our NAMESPACE
3. add a call to .jpackage(pkgname, lib.loc=libname) into our .onLoad.R or .First.lib for example like so:

.onLoad <- function(libname, pkgname) {

.jpackage(pkgname, lib.loc = libname)

}

1. if possible, add .java source files into /java folder of our package

If you are interested in more detail than provided in this super-quick overview, Tobias Verbeke created a [Hello Java World! package](https://cran.r-project.org/web/packages/helloJavaWorld/index.html) with a [vignette](https://cran.r-project.org/web/packages/helloJavaWorld/vignettes/helloJavaWorld.pdf) providing a verbose step-by-step tutorial for interfacing to Java archives inside R packages.

**Setting java.parameters**

The .jpackage function calls .jinit with the default parameters = getOption("java.parameters"), so if we want to set some of the java parameters, we can do it for example like so:

.onLoad <- function(libname, pkgname) {

options(java.parameters = c("-Xmx1000m"))

.jpackage(pkgname, lib.loc = libname)

}

Note that the options call needs to be done before the call to .jpackage, as Java parameters can only be used during JVM initialization. Consequently, this will only work if other package did not intialize the JVM already.

**Handling Java exceptions in R**

rJava maps Java exceptions to R conditions relayed by the stop function, therefore we can use the standard R mechanisms such as tryCatch to handle the exceptions.

The R condition object, assume we call it e for this, is actually an S3 object (a list) that contains:

* call – a language object containing the call resulting in the exception
* jobj – an S4 object containing the actual exception object, so we can for example investigate investigate it’s class: e[["jobj"]]@jclass

tryCatch(

iOne <- .jnew(class = "java/lang/Integer", 1),

error = function(e) {

message("\nLets look at the condition object:")

str(e)

message("\nClass of the jobj item:")

print(e[["jobj"]]@jclass)

message("\nClasses of the condition object: ")

class(e)

}

)

##

## Lets look at the condition object:

## List of 3

## $ message: chr "java.lang.NoSuchMethodError: "

## $ call : language .jnew(class = "java/lang/Integer", 1)

## $ jobj :Formal class 'jobjRef' [package "rJava"] with 2 slots

## .. ..@ jobj :

## .. ..@ jclass: chr "java/lang/NoSuchMethodError"

## - attr(\*, "class")= chr [1:9] "NoSuchMethodError" "IncompatibleClassChangeError" "LinkageError" "Error" ...

##

## Class of the jobj item:

## [1] "java/lang/NoSuchMethodError"

##

## Classes of the condition object:

## [1] "NoSuchMethodError" "IncompatibleClassChangeError"

## [3] "LinkageError" "Error"

## [5] "Throwable" "Object"

## [7] "Exception" "error"

## [9] "condition"

Since class(e) is a vector of simple java class names which allows the R code to use direct handlers, we can handle different such classes differently:

withCallingHandlers(

iOne <- .jnew(class = "java/lang/Integer", 1)

, error = function(e) {

message("Meh, just a boring error")

}

, NoSuchMethodError = function(e) {

message("We have a NoSuchMethodError")

}

, IncompatibleClassChangeError = function(e) {

message("We also have a IncompatibleClassChangeError - lets recover")

recover()

# recovering here and looking at

# 2: .jnew(class = "java/lang/Integer", 1)

# we see that the issue is in

# str(list(...))

# List of 1

# $ : num 1

# We actually passed a numeric, not integer

# To fix it, just do

# .jnew(class = "java/lang/Integer", 1L)

}

, LinkageError = function(e) {

message("Ok, this is getting a bit overwhelming,

lets smile and end here

:o)")

}

)

## Meh, just a boring error

## We have a NoSuchMethodError

## We also have a IncompatibleClassChangeError - lets recover

## recover called non-interactively; frames dumped, use debugger() to view

## Ok, this is getting a bit overwhelming,

## lets smile and end here

## :o)

## Error in .jnew(class = "java/lang/Integer", 1): java.lang.NoSuchMethodError: