**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. The First Part is described below.

# A-primer-in-using-Java-from-R-part-1

**Introduction**

This primer shall consist of two parts and its goal is to provide a walk-through of using resources developed in Java from R. It is structured as more of a “note-to-future-self” rather than a proper educational article, I however hope that some readers may still find it useful.

The primer is split into 2 posts:

1. In this first one we talk about using of the rJava package to create objects, call methods and work with arrays, we examine 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?

**Calling Java from R directly**

Calling Java resources from R directly can be achieved using R’s system() function, which invokes the specified OS command. We can either use an already compiled java class, or invoke the compilation also via a system() call from R. Of course for any real world practical uses, we will probably do the Java coding, compilation and jaring in a Java IDE and provide R with just the final .jar file(s), I however find it helpful to have a small example of the simplest complete case, for which even the following is sufficient.

Let us show that by writing a very silly dummy class with just 2 methods:

* main, that prints “Hello World!” + an optional suffix, if provided as argument
* SayMyName method, that returns a string constructed from “My name is” and getClass().getName()

This HelloWorldDummy.java file can look as follows:

package DummyJavaClassJustForFun;

public class **HelloWorldDummy** {

public String **SayMyName**() {

return("My name is " + getClass().getName());

}

public static void **main**(String[] args) {

String stringArg = "And that is it.";

if (args.length > 0) {

stringArg = args[0];

}

System.out.println("Hello, World. " + stringArg);

}

}

**Compilation and execution via bash commands**

Now that we have our dummy class ready, we can put together the commands and test them by just executing via a shell, or for RStudio fans, we can test the commands via RStudio’s cool Terminal feature. First, the compilation command, which may look something like the following, assuming that we are in the correct working directory:

$ javac DummyJavaClassJustForFun/HelloWorldDummy.java

Now that we have the class compiled, we can execute the main method, with and without the argument provided:

$ java DummyJavaClassJustForFun/HelloWorldDummy

$ java DummyJavaClassJustForFun/HelloWorldDummy "I like winter"

In case we need to compile and run with more .jars that are in folder jars/, we specify the folder using -cp (class path):

$ javac -cp "jars/\*" DummyJavaClassJustForFun/HelloWorldDummy.java

$ java -cp "jars/\*:compile/src" DummyJavaClassJustForFun/HelloWorldDummy

**Compilation and execution of Java code from R**

Now that we have tested our commands, we can use R to do the compilation via the system function. Do not forget to cd into the correct directory within a single system call if needed:

system('cd data/; javac DummyJavaClassJustForFun/HelloWorldDummy.java')

After that we can also execute the main method, and the main method with one argument specified, just like we did it outside of R, once again using cd to enter the proper working directory if needed:

system('cd data/; java DummyJavaClassJustForFun/HelloWorldDummy')

system('cd data/; java DummyJavaClassJustForFun/HelloWorldDummy "Also I like winter"')

**The rJava package - an R to Java interface**

The rJava package provides a low-level interface to Java virtual machine. It allows creation of objects, calling methods and accessing fields of the objects. It also provides functionality to include our java resources into R packages easily.

We can install it with the classic:

install.packages("rJava")

Note the system requirement Java JDK 1.2 or higher and for JRI/REngine JDK 1.4 or higher. After attaching the package, we also need to initialize a Java Virtual Machine (JVM):

## Attach rJava and Init a JVM

library(rJava)

.jinit()

**Creating Java objects with rJava**

We will now very quickly go through the basic uses of the package. The .jnew function is used to create a new Java object. Note that the class argument requires a fully qualified class name in Java Native Interface notation.

# Creating a new object of java.lang class String

sHello <- .jnew(class = "java/lang/String", "Hello World!")

# Creating a new object of java.lang class Integer

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

**Working with arrays via rJava**

# Creating new arrays

iArray <- .jarray(1L:2L)

.jevalArray(iArray)

## [1] 1 2

# Using a list of 2 and lapply

# Integer Matrix int[2][2]

iMatrix <- .jarray(list(iArray, iArray), contents.class = "[I")

lapply(iMatrix, .jevalArray)

## [[1]]

## [1] 1 2

##

## [[2]]

## [1] 1 2

# Integer Matrix int[2][2]

square <- array(1:4, dim = c(2, 2))

square

## [,1] [,2]

## [1,] 1 3

## [2,] 2 4

# Using dispatch = TRUE to create the array

# Using simplify = TRUE to return a nice R array

dSquare <- .jarray(square, dispatch = TRUE)

.jevalArray(dSquare, simplify = TRUE)

## [,1] [,2]

## [1,] 1 3

## [2,] 2 4

# Integer Tesseract int[2][2][2][2]

tesseract <- array(1L:16L, dim = c(2, 2, 2, 2))

tesseract

## , , 1, 1

##

## [,1] [,2]

## [1,] 1 3

## [2,] 2 4

##

## , , 2, 1

##

## [,1] [,2]

## [1,] 5 7

## [2,] 6 8

##

## , , 1, 2

##

## [,1] [,2]

## [1,] 9 11

## [2,] 10 12

##

## , , 2, 2

##

## [,1] [,2]

## [1,] 13 15

## [2,] 14 16

# Use dispatch = TRUE to create the array

# Use simplify = TRUE to return a nice R array

# Interestingly, this seems weird

dTesseract <- .jarray(tesseract, dispatch = TRUE)

.jevalArray(dTesseract, simplify = TRUE)

## , , 1, 1

##

## [,1] [,2]

## [1,] 1 0

## [2,] 0 0

##

## , , 2, 1

##

## [,1] [,2]

## [1,] 0 0

## [2,] 0 8

##

## , , 1, 2

##

## [,1] [,2]

## [1,] 9 0

## [2,] 0 0

##

## , , 2, 2

##

## [,1] [,2]

## [1,] 0 0

## [2,] 0 16

**Calling Java methods using the rJava package**

rJava provides two levels of API:

* fast, but inflexible low-level JNI-API in the form of the .jcall function
* convenient (at the cost of performance) high-level reflection API based on the $ operator.

In practice, there are three ways available to us from the rJava package enabling us to call Java methods, each of them with their positives and negatives.

**The low-level way - .jcall()**

.jcall(obj, returnSig = "V", method, ...) calls a Java method with the supplied arguments the “low-level” way. A few important notes regarding the usage, for more refer to the R help on .jcall:

* requires exact match of argument and return types, doesn’t perform any lookup in the reflection tables
* passing sub-classes of the classes present in the method definition requires explicit casting using .jcast
* passing null arguments needs a proper class specification with .jnull
* vector of length 1 corresponding to a native Java type is considered a scalar, use .jarray to pass a vector as array for safety

# Calling a Java method length on the object low-level way

.jcall(sHello, returnSig = "I", "length")

## [1] 12

# Also we must be careful with the data types:

# This works

.jcall(sHello, returnSig = "C", "charAt", 5L)

## [1] 32

# This does not

.jcall(sHello, returnSig = "C", "charAt", 5)

## Error in .jcall(sHello, returnSig = "C", "charAt", 5): method charAt with signature (D)C not found

**The high-level way - J()**

J(class, method, ...) is the high level API for accessing Java, it is slower than .jnew or .jcall since it has to use reflection to find the most suitable method.

* to call a method, the method argument must be present as a character vector of length 1
* if method is missing, J creates a class name reference

# Calling a Java method length on the object high-level way

J(sHello, "length")

## [1] 12

# Also, the high-level will not help here this way

J(sHello, "charAt", 5L)

## Error in .jcall(o, "I", "intValue"): method intValue with signature ()I not found

J(sHello, "charAt", 5)

## Error in .jcall("RJavaTools", "Ljava/lang/Object;", "invokeMethod", cl, : java.lang.NoSuchMethodException: No suitable method for the given parameters

**The high-level way with convenience - $**

Closely connected to the J function, the $ operator for jobjRef Java object references provides convenience access to object attributes and calling Java methods by implementing relevant methods for the completion generator for R.

* $ 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

# And via the $ operator

sHello$length()

## [1] 12

# But these still do not work

sHello$charAt(5L)

## Error in .jcall(o, "I", "intValue"): method intValue with signature ()I not found

sHello$charAt(5)

## Error in .jcall("RJavaTools", "Ljava/lang/Object;", "invokeMethod", cl, : java.lang.NoSuchMethodException: No suitable method for the given parameters

**Examining methods and fields**

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

# vector of all fields and methods associated with sHello

.DollarNames(sHello)

## [1] "CASE\_INSENSITIVE\_ORDER" "equals("

## [3] "toString()" "hashCode()"

## [5] "compareTo(" "compareTo("

## [7] "indexOf(" "indexOf("

## [9] "indexOf(" "indexOf("

## [11] "valueOf(" "valueOf("

## [13] "valueOf(" "valueOf("

## [15] "valueOf(" "valueOf("

## [17] "valueOf(" "valueOf("

## [19] "valueOf(" "length()"

## [21] "isEmpty()" "charAt("

## [23] "codePointAt(" "codePointBefore("

## [25] "codePointCount(" "offsetByCodePoints("

## [27] "getChars(" "getBytes()"

## [29] "getBytes(" "getBytes("

## [31] "getBytes(" "contentEquals("

## [33] "contentEquals(" "equalsIgnoreCase("

## [35] "compareToIgnoreCase(" "regionMatches("

## [37] "regionMatches(" "startsWith("

## [39] "startsWith(" "endsWith("

## [41] "lastIndexOf(" "lastIndexOf("

## [43] "lastIndexOf(" "lastIndexOf("

## [45] "substring(" "substring("

## [47] "subSequence(" "concat("

## [49] "replace(" "replace("

## [51] "matches(" "contains("

## [53] "replaceFirst(" "replaceAll("

## [55] "split(" "split("

## [57] "join(" "join("

## [59] "toLowerCase(" "toLowerCase()"

## [61] "toUpperCase()" "toUpperCase("

## [63] "trim()" "toCharArray()"

## [65] "format(" "format("

## [67] "copyValueOf(" "copyValueOf("

## [69] "intern()" "wait("

## [71] "wait(" "wait()"

## [73] "getClass()" "notify()"

## [75] "notifyAll()" "chars()"

## [77] "codePoints()"

**Signatures in JNI notation**

| **Java Type** | **Signature** |
| --- | --- |
| boolean | Z |
| byte | B |
| char | C |
| short | S |
| int | I |
| long | J |
| float | F |
| double | D |
| type[] | [ type |
| method type | ( arg-types ) ret-type |
| fully-qualified-class | Lfully-qualified-class ; |

**In the fully-qualified-class row of the table above note the**

* L prefix
* ; suffix

**For example**

* the Java method: long f (int n, String s, int[] arr);
* has type signature: (ILjava/lang/String;[I)J



R <3 Java, or maybe not?

# A-primer-in-using-Java-from-R-part-2

**Using rJava with custom built classes**

**Preparing a .jar archive for use**

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.

# 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

**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: