Hello Java World! A Tutorial for Interfacing to Java Archives inside R Packages.

Tobias Verbeke

2014-09-03

Contents

1 Introduction

This document provides detailed guidance on interfacing R to Java archives inside an R package. The package we will create in this tutorial, the `helloJavaWorld` package, will invoke a very simple Java class, the `HelloJavaWorld` class, from inside an R function of the package, the `helloJavaWorld` function. The objective is to help other people to make available Java algorithms to the R world, be it to compare results, or for their own sake.

2 Structure of the package

We create a folder, `helloJavaWorld`, which will be the root folder of our `helloJavaWorld` package. For detailed information on R packages and their structure, the reader is referred to the Writing R Extensions manual. This section only highlights elements that are relevant to our situation. Before detailing the contents of the individual files and folders, we provide a summary overview in Figure 1.

`DESCRIPTION` Inside this folder, we create `DESCRIPTION` file, which can be considered to be the 'identity card' of the package. The most important changes when comparing to regular `DESCRIPTION` files are that

```
helloJavaWorld
    `- inst
        `- doc
            `- helloJavaWorld.Rnw       (this document)
        `- java
            `- hellojavaworld.jar
    `- java
        `- HelloJavaWorld.java
    `- man
        `- helloJavaWorld.Rd
    `- R
        `- helloJavaWorld.R
        `- onLoad.R
    `- DESCRIPTION
    `- NAMESPACE
```

Figure 1: Overview of package contents for the `helloJavaWorld` package.

- there must be a package dependency on the `rJava` package (`Depends` field) as well as
- a system dependency on Java (`SystemRequirements` field).

`inst/java` We create the folder `inst/java` to host our JAR file. This JAR file contains a single `HelloJavaWorld.class` file, generated from the following `HelloJavaWorld.java` file.

```
public class HelloJavaWorld {

  public String sayHello() {
    String result = new String("Hello Java World!");
    return result;
  }

  public static void main(String[] args) {
  }
}
```

As one can see, this is not the typical example of a Hello World application. The string `Hello Java World!` is namely not printed to the console, but returned by the `sayHello` method. The reason to deviate from this tradition is that in practice the interfacing to Java classes will nearly always result in return values being returned to R by the methods of the classes in the JAR file(s).

`java/` The Writing R Extensions Manual recommends to make Java source files available in a top-level `java/` directory inside the package. This will ensure source files are not installed, but makes these files available as required when the package is published under an open source license.

`R/` Two functions are contained in the `R/` subfolder of the package. The first function is the function that will assure that the Java code is made available. The second function will be the R wrapper to execute the Java HelloWorld class.

Namespaces are recommended in R packages, so we will only detail how to include the first function when the package has a namespace. We include a file `onLoad.R` with the following content :

```
.onLoad <- function(libname, pkgname) {
  .jpackage(pkgname, lib.loc = libname)
}
```

The `.onLoad` function is a hook function that will be run immediately after loading the package. The function `.jpackage` that is called inside the `.onLoad` function takes care to initialize the JVM and to add the `java/` folder of the package to the class path. If users make use of custom R library paths, they can rely on the `libname` argument. Note that when building the package the `inst` level is 'taken out', so that in the built package the `java/` subfolder will be directly in the root folder.

The second function we included in the `R/` folder is responsible for making the Java class available to the user and is a simple wrapper:

```
helloJavaWorld <- function(){
  hjw <- .jnew("HelloJavaWorld")     # create instance of HelloJavaWorld class
  out <- .jcall(hjw, "S", "sayHello")  # invoke sayHello method
  return(out)
}
```

The `.j*` functions such as `.jnew` and `.jcall` are part of the `rJava` package and documented in their respective documentation files[1]. For convenience, we provide some minimal explanation here as well.

The `.jnew` function creates a new instance of a Java object. The first argument (here `"HelloJavaWorld"`) indicates the class of the object to be created. If other arguments are needed to construct the Java object, these can be passed as R arguments to the `.jnew` function as well. In this very simple case, however, no supplementary arguments are required to create a `"HelloJavaWorld"` object. By assigning the result of `.jnew` to a variable,

[1] The `rJava` package can be obtained from CRAN; more information on the package can be found at the package home page `http://www.rforge.net/rJava/`

Abbreviation	Type
`"V"`	void
`"I"`	integer
`"D"`	double (numeric)
`"J"`	long
`"F"`	float
`"Z"`	boolean
`"C"`	char (integer)
`"S"`	String
`"B"`	byte (raw)
`"L<class>"`	Java object of class `<class>` (e.g. `"Ljava/lang/Object"`)
`"[<type>"`	Array of objects of type `<type>` (e.g. `"[D"` for an array of doubles)

Table 1: Overview of the abbreviations that can be used as a shortcut to indicate the return type of a Java method in the `.jcall` function of the `rJava` package.

we obtain a reference to the object (here `hjw`) that we can use to further work with the object, or to call Java methods on the newly created object.

The `.jcall` function allows to call a method (`"sayHello"`) on a Java object. The first argument will be a reference to the object, which we stored in the `hjw` variable in the previous statement. The second argument contains a character indicating the return type of the method we call. In this example, the return type will be a String, which has been mapped to the `"S"` character. An overview of all return types is offered in Table 1.

It is important to note that the `out` object is no longer a reference to the resulting Java object, but has been evaluated directly to a string, i.e. an R character vector of length one. The explanation is that the `.jcall` method has an argument `evalString` that is set to `TRUE` by default. If we would have wanted to obtain the reference to the Java object to further work with it, we would have explicitly set `evalString` to `FALSE` as in

```
outRef <- .jcall(hjw, "S", "sayHello", evalString = FALSE)
```

To explicitly obtain a string from a Java object reference, we could then use the function.jstrVal, as in

```
.jstrVal(outRef)
```

The string returned would, of course, be exactly the same

```
"Hello Java World!"
```

NAMESPACE In the R functions of our package, some functions were used from the `rJava` package. As this package has a namespace as well, it is necessary to import the relevant functions or the whole package into the package namespace to ensure that the `rJava` package will be loaded as well. Importing the whole package can be done using the `import` directive.

If, on the other hand, we want to make our functions (or other objects) available to the package user, we need to export these objects using the `export` directive. We will of course not export the `.onLoad` function as it is not meant to be used by the package user. The only function exported, therefore, is the `helloJavaWorld` function. The `NAMESPACE` file will thus have the following contents[2] :

```
import("rJava")
export("helloJavaWorld")
```

3 Use of the package

Once the package is checked, built and installed, we can load the package with

```
library(helloJavaWorld)
```

and demonstrate it by simply calling

```
> helloJavaWorld()
[1] "Hello Java World!"
```

4 Acknowledgements

We would like to thank Simon Urbanek for his continuous efforts on the `rJava` package, without which it would not have been possible to say Hello from Java to R. The following persons provided feedback and suggestions that helped improve the document: Duncan Murdoch, Prof. Ripley, Simon Urbanek, Hadley Wickham.

[2] For more information on package namespaces, please consult Section 1.6 from the Writing R Extensions manual.