arji – another R Java interface

VJ Carey

April 10, 2006

Contents

1 Introduction

SJava (Temple Lang, 2000) and *rJava* (Urbanek, 2001) are R packages defining interfaces between R and Java. *arji* is a new approach to a one-way interface with the following features

- developed and tested simultaneously on Unix, Windows and Mac platforms
- designed using S4 classes and methods and external pointer support
- exposes JNI facilities to R so that the interface can be extended using R if desired
- attempts to provide compatible APIs with other interfaces

This work would not exist without the excellent original work by Temple Lang and Urbanek.

2 Scope of functionality

2.1 Virtual machine control

The current approach is quite crude. It seems that JDK 1.5 does not allow more than 1 JVM associated with a process through the JNI. We will, by default, use a global variable in R to represent the JVM reference; however, an option to the JVM invocation function (`createJVM`) allows one to handle the JVM reference like any other (non-global) R object.

```
> library(arji)
> args(createJVM)

function (optstr, globalize = TRUE, RobjName = "..arjiJVM")
NULL
```

If `globalize` is `FALSE`, the `createJVM` function returns an instance of *JavaVMRef*; otherwise the instance is assigned to the symbol named by parameter `RobjName`. Currently nothing is done to pick up the current environment classpath setting; you must set the class path explicitly.

```
> jvm <- createJVM(opts = paste("-Djava.class.path=",
+     system.file("java", package = "arji"), sep = ""),
+     globalize = FALSE)
> jvm

arji package JavaVM reference, invoked with options:
  -Djava.class.path=/tmp/Rinst1578578897/arji/java
address: <pointer: 0x2a974f10c0>
```

To obtain the default behavior, I will now assign this to the default variable for holding the JVM reference:

```
> ..arjiJVM <- jvm
```

The *JavaVMRef* class contains the option string and a pointer to the JNI JVM reference. Additional data concerning, e.g., versions, could be held by this object.

```
> getSlots(class(jvm))

       jvmptr     optString
"externalptr"   "character"
```

The `destroyJVM` function uses the JNI to unload the JVM. We will likely provide a `.onUnload` function to ensure that this occurs on termination of package use.

At present a global variable `..JVMAlive` is a logical scalar with obvious interpretation. When `destroyJVM` is used , this variable is set to `FALSE`. In order for `.onUnload` to know what object `destroyJVM` should be applied to, the `..JVMAlive` may be extended to include the variable name used for the JavaVMRef object.

A `checkJVM` function is run by most functions that explicitly use the JVM to see if a live JVM is available. This is inadequate. A preferred functionality is to have a way of determining whether a given Java object reference in R is associated with the *existing* JVM. As it stands one can get extremely unpleasant behavior if one tries to invoke a method that was defined in a class instantiated from a defunct JVM.

2.2 Class extraction and inspection

The package ships with some compiled java classes under `inst/java`.

```
> cc <- getJavaClass("demo")
> cc
```

```
arji package Java Class reference, classname  demo
```

2.3 Object construction and inspection

```
> demo <- .arjiNew("demo")
> demo
```

```
arji package Java Object reference, instance of class demo
```

```
> str <- .arjiNew("java/lang/String", "abc")
> str
```

```
arji package Java Object reference, instance of class java/lang/String
```

2.4 Field extraction

In these examples we extract values of two fields in the demo class:

```
> fid <- getJavaStaticFieldID("demoSField", "I",
+     cc)
> fid2 <- getJavaStaticFieldID("demoSFieldD", "D",
+     cc)
> getJavaStaticField(fid, cc)
```

```
[1] 5
```

```
> getJavaStaticField(fid2, cc)
```

```
[1] 5.15515
```

2.5 Method extraction, inspection, and invocation (static and nonvirtual)

We can obtain a static method ID given the method name, its signature, and the defining class.

```
> mm <- getJavaStaticMethodID("main", "([Ljava/lang/String;)V",
+     cc)
```

We can then invoke this using a very general interface, `callJNIArgable`. Given a method ID, `callJNIArgable` will set up C-level structures to invoke the method with arguments.

```
> callJNIArgable(mm, .jtask("CallStaticMethod"),
+     .jtype("void"))
```

```
An object of class "arjiArgableOut"
Slot "outObjRef":
arji package Java Object reference, instance of class java/lang/Object

Slot "rettype":
[1] 1

Slot "midobj":
arji package Java static MethodID reference, methodname  main
declared in class  demo
```

The utility of this somewhat clumsy approach to invocation is illustrated here – we want a single interface that can invoke the appropriate method on the basis of the argument types.

```
> m4 <- getJavaStaticMethodID("doit3", "([DLjava/lang/String;)Ljava/lang/String;",
+     cc)
> callJNIArgable(m4, .jtask("CallStaticMethod"),
+     .jtype("void"), as.double(c(213.45, 22.23)),
+     "newth")
```

```
An object of class "arjiArgableOut"
Slot "outObjRef":
arji package Java Object reference, instance of class java/lang/String

Slot "rettype":
[1] 1

Slot "midobj":
arji package Java static MethodID reference, methodname  doit3
declared in class  demo
```

Passage of an object to a static method is illustrated here.

```
> d2 <- .arjiNew("demo2")
> md2 <- getJavaStaticMethodID("bing", "(Ldemo;)V",
+     d2@classObj)
> callJNIArgable(md2, .jtask("CallStaticMethod"),
+     .jtype("void"), demo@objref)

An object of class "arjiArgableOut"
Slot "outObjRef":
arji package Java Object reference, instance of class java/lang/Object

Slot "rettype":
[1] 1

Slot "midobj":
arji package Java static MethodID reference, methodname  bing
declared in class  demo2
```