Examples of the Omegahat R-Java Interface

Duncan Temple Lang

July 25, 2000

Contents

A simple use of the *.Java()* mechanism is to present the user with a graphical interface for selecting a file, for example, when no argument is provided to *source()*. In *Java*^{TM} we can use the class `JFileChooser` to provide the display of a directory and allow the user to select a file or investigate other directories associated with the "current" directory. Similarly, we may have a menu for selecting one of many options (a graphical *menu()*), such as a a type of algorithm, smoother, etc. Also we may allow the user specify a parameter (e.g. bandwidth) using a slider before the computations are initialized. These are all characterized by blocking calls from *S*/*R*, waiting for user input.

It cannot be overemphasized that these are simple examples of the mechanism and careful GUI design is needed for each scenario to be useful. One of the uses for having the *.Java()* interface and the $\widehat{\Omega}$ language is to allow people to quickly experiment with GUIs and see what works.

We focus on the first of these examples. The others are similar in spirit on the *S*/*R* details. The basic steps are to create the `JFileChooser`, specifying the directory of interest, and then to display it in a window. Finally, we wait for the user to interact with the dialog and either dismiss it or select a file by clicking on the *Open* button. We test for the latter case and retrieve the selected file from the `JFileChooser`.

Since there are several steps, we can turn this into an $\widehat{\Omega}$ function. See below.

```
[*]
 dir = "";
 chooser = new JFileChooser(dir);
 f = new JFrame();
 ok = chooser.showOpenDialog(f);
 value = null;
 if(ok == JFileChooser.APPROVE_OPTION) {
   value = chooser.getSelectedFile();
 }
```

To obtain the same results in *S*/*R*, we can use the following calls. We can ignore the test for whether selected *Open* or not since calling `getSelectedFile()` simply returns `null` if the user selected *Cancel*.

```
[getFile.s]
 chooser <- .JavaConstructor("JFileChooser")
 f       <- .JavaConstructor("JFrame")
 .Java(chooser,"showOpenDialog", f)
 file <- .Java(chooser,"getSelectedFile")
 .Java(file,"toString")

 source(file);
```

Since there are several steps in the `JFileChooser`, we can turn this into an $\widehat{\Omega}$ function.

```
[getFile.omg]
getFile = function(dir = "") {
 chooser = new JFileChooser(dir);
 f = new JFrame();
 ok = chooser.showOpenDialog(f);
 value = null;
 if(ok == JFileChooser.APPROVE_OPTION) {
   value = chooser.getSelectedFile();
 }
}
```

Of course, we can also create an *R*/*S* function. See **file.choose**.

1 Filename Filters

We can extend the example above by specifying a file filter to focus on files which match a particular characteristic. The most obvious of these are extensions, but we can have more elaborate ones such as those that only identify large files, or files whose first starts with "#"!.
As usual, with

2 Network Access

Java^{TM} provides an extensive array of network access classes. These allow us to add URL connections, sockets, etc. to *S*[1] Suppose we want to read the contents of a URL which we know to be pure text (as opposed to an image, etc.). There are several ways to do this. The simplest is to find a *Java*^{TM} class that does this and returns an array of `String` objects. We have one of these in **StatDataURL.java**.

```
[url.s]
 u <- .JavaConstructor("StatDataURL", "http://www.omegahat.org/Scripts/AwtButton")
 els <- paste(.Java(u, "getContents"),collapse="\n")
```

The code underlying this can be done directly from S but involves significantly more expressions.
This is an example of where it is more convenient to do a little Java programming and call the higher level method from *S*/*R* rather than performing all the calls via the *.Java()*.

[1] *R* already has some of these features.

3 Data Management & Garbage Collection

One problem with remote references is that one side of the interface cannot really know about the continued use of an object and so values must be stored until explicitly released by the "owner". We can definitely help in this regard with some heuristics, but it is not possible to be absolutely certain that an object will not be used in the future in general contexts.
In some situations it is easy to identify temporary values. From *S*/*R*, we can examing the arguments to *.Java()* calls and determine if those that are themselves *.Java()* or *.JavaConstructor()* calls are named arguments. If they are, they will be assigned into the permanent "*Java*^{TM}" database and are explicitly being assigned for future use.
For example, in the call below, the *.JavaConstructor()* of the `JFrame` is not permanently assigned and so there is no way to refer to it in future calls.

```
[*]
.Java(chooser,"showOpenDialog", .JavaConstructor("JFrame"))
```

In such cases, we can diagnose this at the *S*/*R* call and release the anonymous reference explicitly within the *.Java()*.

4 Polymorphic (Overloaded) Functions

Different from IDL and the CORBA interfaces, *Java*^{TM} allows method overloading, meaning that we may have more than one object
Having $\widehat{\Omega}$ handle this by having fuzzier matching does not help. Almost immediately this breaks. Instead, in the case that there is more than one possible method, we can have $\widehat{\Omega}$ identify all the matching methods and return these or popup a selection dialog allowing the user to select the most appropriate one.
Alternatively, good code writers will remove ambiguity in the *S*/*R* calls by specifying the *.sig* argument. This is a list of strings specifying the *Java*^{TM} type to use for the different arguments. This aids the conversion and also identifies the *Java*^{TM} method directly.
This needs to be reimplemented to support lists, . . .

5 Callbacks

As discussed in Dynamic Class Generation, it is very convenient to have seamless access to arbitrary *Java*^{TM} classes and methods, but there is a potential asymmetry. We cannot use non-primitive local objects - *S* and *R* values - as arguments in *Java*^{TM} method calls. If this were not surmountable, this would definitely be a significant drawback. We would need to have conversion methods for each *S* and *Java*^{TM} type pairs. These would not have to be implemented in *C*, but we can use the *.JavaConstructor()* and *.Java()* functions themselves to transfer the sub-elements of an *S* object. However, the two languages are very different and it is not possible in all cases to transfer data and preserve its semantic meaning. Instead, *Java*^{TM} requires methods to be associated with data values in a class . . ..
An alternative is to use the same mechanism as we use in the CORBA interface.
This risks multiple communications across the interface between the two systems. For example, in *S*, we may create a "variable" named *x* by generating 10 random numbers. We then add this to a *Java*^{TM} *DataFrame*. This should be an object of class (actually, interface) *VariableInt*.

```
[*]
 x = rnorm(10)
 .JavaConstructor("DataFrame, x, .sig=c"VariableInt")
```

This signals the *Java*^{TM} conversion mechanism that it should look for a method that takes a *numeric* and generates a *VariableInt*. If no explicit converter exists, we create a *proxy* or remote reference from the *S* side and pass it to the *Java*^{TM} side of the interface (along with the target *Java*^{TM} class, if specified). This is the same thing as being done in the $\widehat{\Omega}$ databases when returning non-primitive objects. Basically, we store the *S*/*R* object in a special place with a unique name and send across the name and other information used to identify this object. The $\widehat{\Omega}$ code then converts

this reference into an object appropriate for the specified call. In our example, this is *VariableInt*. It does so by generating a new class which implements the methods in this interface. The body of each method simply gathers up the *Java*^{TM} arguments into a *List* and passes them to an inherited `eval()` method along with the name of the method. This converts these arguments to *S* objects via native methods and arranges to invoke an *S*/*R* call to a function with the same name as the original *Java*^{TM} method being invoked with these arguments.
This is identical to the *FunctionListener* class mechanism and the dynamic class generation.
We can use the example above to more concrete about this. The call to *.JavaConstructor()* above creates an *S* reference and stores it so that it won't disappear after the *S* call is complete. The $\widehat{\Omega}$ evaluator receives this and the specification that it should be a *VariableInt*. So it performs the following calls to

- create a new class that is both an *S* remote reference and implements the specified interface
- instantiates an object of this new class with the specified reference.

```
[*]
Object call(SRemoteReference ref)
{
  gen = new EvaluableInterfaceGenerator("VariableInt", "SReferenceVariableInt");
  dynamicClassLoader().defineClass(gen);
  obj = new SReferenceVariableInt(ref);
 return(obj);
}
```

The *VariableInt* interface has a method `value(int)`. The newly generated class *SReferenceVariableInt* (which is not an interface) then implements this in the following manner.

```
[*]
public Object value(long which) {
 FunctionCallArguments args = new FunctionCallArguments(1);
 args.addArgument(which);
 Object obj = eval(args, "value");
 return(obj);
}
```

The `eval()` method is a native method. It passes the identifying information of the remote reference stored in the object (the owner of the `eval()` method), the second argument identifying the method (`value`) being called and the arguments to *C* code that invokes the appropriate expression in *S*/*R*.
It is simplest to think about this in *R*, I believe, and to provide the complete picture. While we can perform most of the conversions automatically, we will explain things describing the manual steps. Rather than passing the *S*/*R* object *x* to the call to *.JavaConstructor()*, we actually create a function closure that provides the methods for VariableInt but in *R* and has access to the actual data.

```
[VariableInt.r]
VariableInt <- function(data) {
 value <- function(which) {
   data[which]
 }
 add <- function(obj, which) {
   which <- which + 1
   if(length(data) >= which) {
       x <-data
       x[which] <- obj
      data
<<- x
```

```
   } else
     data
<<- c(data,obj)
  obj
 }

 return(list(value=value, add=add, data=data))
}
```

6 Data Frames

Suppose we have a data frame in R and the we wish to make it available to an Java method. As usual, we have two choices: a) to transfer the contents of the data frame to an appropriate Java object, or b) pass a reference to the R object in the form of a Java interface which implements its methods via R function calls.
Of these two, a) may appear simpler. For example, suppose we chose to create an instance of the *DataFrame* class in Omegahat. We can then use the *.JavaConstructor()* and *.Java()* methods to create such an instance and transfer the elements of the R data frame. This is done in the following code.

```
[*]
data(mtcars)
jdata <- .JavaConstructor("DataFrame")
for( i in 1:ncol(mtcars))
  .Java(jdata,"addVariable", names(mtcars)[i],
         .JavaConstructor("RealVariable", mtcars[,i]))
```

We can then use this Java *DataFrame* in whatever manner we wish. This is a separate copy of the data and any changes to the Java object will not be seen by the R object. One thing we might do is display the data frame in an interactive table/grid. Omegahat has such a class – *DataFrameViewer* (and the utility class *DataFrameViewerWindow*).
Note that this doesn't work now. The problem seems to lie with threads and the result is a major crash of the JVM and hence R.

```
[*]
w <- .JavaConstructor("DataFrameViewerWindow", jdata)
```

We can at any time retrieve the values from the *jdata()* object from within R. We use the methods of *DataFrame* via the *.Java()* function.

```
[*]
v <- .Java(jdata, "get", "mpg")
.Java(v, "getValues")
```

And we can update individual values.

```
[*]
v <- .Java(jdata, "get", "mpg")
.Java(v, "add", pi, as.integer(3))
```

The second approach (b)) involves creating a Java class that implements the *DataFrameInt* interface and calls R functions when any of its methods are called. We use the dynamic compiler (see the Howto document) to do this. Then we create the foreign reference in R representing the local data frame and pass this as the argument to the constructor.

7 Mouse Events

An example of tracking the mouse was raised in connection with the Tcl/Tk package in R. Here is a mechanism for doing this in Java. We start by constructing a very simple window and canvas whose mouse motion events we can monitor.

```
[*]
 comp <- .JavaConstructor("JCanvas")
      # change the color to red.
 .JavaConstructor("GenericFrame", comp, T)

 .Java(comp,"setBackground",.Java("Color", "red"))
```

Next, we create and load a new class which implements the `MouseMotionListener` methods by calling R functions.

```
[*]
 compiler <- .JavaConstructor("ForeignReferenceClassGenerator",
                              "java.awt.event.MouseMotionListener",
                               "RMouseListener")

 .Java(.Java("__Evaluator","dynamicClassLoader"), "defineClass", compiler)
```

Now we are ready to implement the R side of the listener. We define a closure function (*mouseListener()*) in which we have the x and y coordinates of the last reported mouse position and the function *mouseMoved()* which is called to update these values. This function is called by the Java event notification mechanism and passes it a Java `MouseEvent` object as the only argument. We use the *.Java()* to get the X and Y coordinates from the event.

```
[*]
 mouseListener <- function() {
    x <- 1
    y <- 1
   mouseMoved <- function(ev) {
     x
<<- .Java(ev, "getX")
     y
<<- .Java(ev, "getY")
   }
   return(list(mouseMoved = mouseMoved,
               where=function(){c(x,y)}))
 }
```

Now we create an instance of this closure definition and export it by registering it with the foreign reference manager. (This is done implicitly as part of the call to *foreignReference()*.) And lastly, we register it as a Java listener by creating an instance of the newly created class (*RMouseListener*) and call the `addMouseMotionListener()` on the canvas.

```
[*]
 l <- mouseListener()
 r <- foreignReference(l)
 .Java(comp,"addMouseMotionListener", .JavaConstructor("RMouseListener",r))
```

As it stands, this example is fine. However, it illustrates one of the difficulties that comes from integrating event-driven, GUI tools into a console-based, read-eval-print program like R. Where and when do we process the mouse location? Let's extend the example and have the R listener function (*mouseMoved()* above) display the values in a label. (See **mouseMotion.R** in the examples directory.)
First, we create the graphical interface slightly different.

```
[*]
 panel <- .JavaConstructor("JPanel")
 .Java(panel,"setLayout", .JavaConstructor("BorderLayout"))
 label <- .JavaConstructor("JLabel", "(   ,   )")
 comp <- .JavaConstructor("JPanel")
 .Java(panel,"add","North", label)
 .Java(panel,"add","Center", comp)
 .JavaConstructor("GenericFrame", panel, T)
```

Next we define the closure to accept the `JLabel` reference as an argument. In this way, it is available to the inner functions. We add a command to the end of that function that calls the `setText()` method of that `JLabel` object, displaying the current coordinates of the mouse.

```
[*]
 mouseListener <- function(jlabel) {
    x <- 1
    y <- 1
   mouseMoved <- function(ev) {
     x
<<- .Java(ev, "getX")
     y
<<- .Java(ev, "getY")

     .Java(jlabel, "setText", paste("(",x,", ",y,")", sep=""))
   }
   return(list(mouseMoved = mouseMoved,
               where=function(){c(x,y)}))
 }
 l <- mouseListener(label)
```

The remainder of the script is the same as the earlier one. When we run this, the label is updated each time and shows the (X, Y) pair.

8 Graphics & Graphics Devices

Given a Swing component, it is reasonably easy to draw on it. The full range of graphics functionality in Java is available, including the 2D and 3D API.

```
[*]
canvas <- .JavaConstructor("JPanel")
.JavaConstructor("GenericFrame",canvas,T)
```

```
g <- .Java(canvas,"getGraphics")
.Java(g,"drawLine", 10,10, 100, 100, .sigs=rep("I", 4))
.Java(g,"drawArc",  10,10, 100, 100, 0, 180, .sigs=rep("I", 6))
.Java(g,"setColor",  .Java("Color", "red"))
```

The book *Java 2D Graphics* illustrates many different things one can do with the 2D Graphics API, including image manipulation, rotation, etc.