Getting Started with the R-Java/Omegahat Interface

Duncan Temple Lang
John Chambers

August 8, 2000

Contents

1. partially qualified class names.

2. .sigs argument

Abstract

This gives a brief outline of how one uses the *.Java()* interface from R/S to Java provided as part of the Omegahat project. We start by providing a little of the philosophy of underlying the interface and how to think about it, specifically comparing it to the *.C()* and *.Call()* functions. Then we illustrate the details. We start with the initialization of the Java Virtual Machine (JVM). Then we move from static (class-specific) methods and how arguments are passed from R to Java. Then we move to creating Java objects and invoking methods in these objects. We then discuss Java arrays. At this point, we focus on how conversion of objects from R to Java and vice-versa is done and how it can be controlled by the user.

This and the other documents discussing the R-Java interface are continually updated. Updates and additional information is available from the Omegahat Web-site `http://www.omegahat.org`. You should check `http://www.omegahat.org/RSJava`.

1 Overview: The Penny Tour

The R-Java interface is an inter-language communication mechanism and is similar to the ability to call *C*, *C++* and Fortran routines from R. One can interactively invoke Java methods from R and pass R objects to these methods. When calling *C* and Fortran, one identifies the (global) routine by name and parameterizes the call by passing different arguments. Unlike *C* and Fortran, Java is an object oriented language. This means that methods are usually not global, but are associated with an object – an instance of a particular class. Thus, one must identify not only the method and its arguments, but also the Java object on which the method is to be invoked. For this R-Java interface to be useful, therefore, we must be able to create Java objects from R and be able to refer to them from R and call their methods from R and have these evaluated within Java. This is a major difference between the Java interface and the corresponding *C*, *C++*and Fortran versions: the Java interface provides a mechanism for creating, managing and identifying persistent objects in the foreign language (Java) from R. It does so by managing these objects in databases similar to the elements of R's variable search path. These *foreign objects* are represented in R as references (also known as pointers or handles).

Java objects are created from R via the *.JavaConstructor()* (and *.JavaArrayConstructor()*) function. The particular constructor method of the Java class being instantiated is identified by the types of the R arguments. One can invoke methods on a Java object via the *.Java()* function. This expects the reference to the Java object, the name of the method and arguments to that method. Again, like *R* but unlike *C*, a Java class can have several different methods with the same name. The types of the R arguments identify which Java method to invoke.

The *.Java()* function returns the value of the Java method call. If this is an object we understand how to convert to an R object, we do so and the value of the object lives in R. Otherwise, it is a reference to a Java object managed by the Omegahat interpreter and is available for future Java method calls either as the target or an argument. The R-Java interface provides a basic set of facilities for converting Java objects to R objects. The obvious conversion of primitives are available and are displayed in table 1. Java scalars are mapped to R vectors of length 1, and arrays of Java primitives are mapped to the appropriate vector of length n, where n is the length of the Java array.

One can use the $ operator to do the equivalent of a call to *.Java()*. Given an R object, *jobj()*, which is a reference to a Java object, the expression `jobj$methodName(arg1, arg2, .....)` is equivalent to `.Java(jobj, "methodName", arg1, arg2, .....)`

The powerful addition the R-Java provides is the ability to add (and remove) converters for different Java classes. One can register different *C* (or *C++*) routines that each take a Java object of particular class associated with that converter and return an R object. (In the next version, we will allow users to specify R functions to perform this conversion.)

The only aspect we are missing in this setup is how R objects are passed to Java. Well, primitive R objects (e.g. the basic vector types: logical, numeric, integer, character) are converted according to table 1. In some circumstances,

Java	R
double	numeric()
float	numeric()
int	integer()
String	character()
char	character()
byte	character()
long	integer()
boolean	logical()

Table 1: Conversion of primitives between R to Java

we will have to specify which of several like-named methods that we wish to invoke due to the method overloading supported by Java. For example, we might wish to call a method that expects an array of `boolean` values rather than one that expects a single `boolean` value. The *.Java()* function allows one to specify target Java types for the different arguments. The *.sigs* influences both how the R objects are converted to Java and also the method selection performed by Omegahat.

Converting non-primitive objects pose a potential challenge. What Java class should be used to represent a list? a data frame? a graphics device? The answer is that it depends on the context and the Java method or constructor being invoked. In some cases, it is not desirable to transfer the contents of an R object to a Java. One such reason is that there is no suitable Java class readily available. A second reason is that one wishes the R and Java engines to "share" an object and thus we want to pass a reference to the R object.

In order to support this concept of passing R objects by reference, the R-Java interface provides a convenient mechanism for mapping R objects to a Java class named *RForeignReference*. This provides the basic mechanism by which Java methods can be implemented by calling R functions on that object. These R functions can query and even change the state of the R object (i.e. the objects have mutable state) and then return control to Java. The underlying Omegahat classes even provide a dynamic mechanism for compiling and loading new Java classes that implement any collection of Java interfaces by calling a corresponding R function. This allows an R user to rapidly program Java classes entirely within their familiar R environment.

Finally, since the entire interface mechanism rests on the embedded Omegahat interpreter, one has access from R to all the functionality with an interactive Omegahat session. This means that instead of invoking Java methods, one can create more complex and compound Java expressions and pass these to the Omegahat evaluator via the *.OmegahatExpression()* function. Additionally, this allows one to perform *substitute()*-like tricks to parameterize an expression with Omegahat variables, but from R. This string-based style of programming is convenient, and has proven effective in a variety of settings. The R-Tcl/Tk package is such an example. However, it does complicate debugging and restrict the types of computations that can be performed to those that are expressible via strings. In this way, it does not produce ideal software engineering. Rather than using the *.OmegahatExpression()*, we encourage users to employ the *.Java()* function and avoid static methods where possible. (This allows others to use the same R and Java code but to use derived classes with slight changes in implementation, making use of the power of inheritance.)

The Omegahat interpreter does provide numerous methods which can be used to query the state of the Java environment and the connection to R. For example, one can ask it for a list of all Java classes it considers are convertible to R. Also, one can query the collection of references it currently manages. Like R's and S's capabilities for allowing the user to query available functions and methods for different classes, the Omegahat interpreter provides the `getMethods()` method. This returns a list describing the different methods in a Java class. The R programmer can use this to discover more about a particular object or class. The evaluator also maintains a list of all available classes and can resolve partially qualified class names (see `expandClassName()`).

2 Other Documents

The installation of the R package includes this and other documents and of course, the online per-function documentation.

Examples.pdf	some examples of using the *.Java()* interface, including some of the more advanced topics such as GUIs, callbacks to R functions, dynamic compilation, etc.
Features.pdf	a description of some of the features of this interface and a comparative discussion of other inter-system interface approaches such as CORBA, XML and other S-Java interfaces.
Internals.pdf	an early document that provides some examples of how the interface is used and how the internal mechanisms effect this interface.
Howto.pdf	this document that attempts to give a brief tutorial to get people started and how they might use the R-Java interface.
FAQ.html	a collection of questions and answers that might arise when things haven't been installed correctly, won't run, or give apparently odd answers.
README	Short version of requirements, installation instructions, etc.
examples/	a collection of example R scripts that illustrate different aspects of the interface's features. See the **README** file in that directory for more information.

Table 2: R-Java Documents

You can find where these documents are located via the R expression

```
[R]
  system.file("Docs", pkg="Java")
```

This R-Java interface has been in existence for almost 2 years. Obviously it has evolved since its earliest implementation and the addition of converters, references, dynamic compilation, and other features have been added incrementally. A result of this history is the presence of many different documents that provide some insight into the philosophy and the generality of the the interface.

3 Installation

One installs the R package via the shell command

```
[Install]
 R INSTALL -c Java_1.0.tar.gz
```

Note that the `-c` argument is needed since it creates some extra files (a symbolic link, specifically) after the standard R installation is performed. See the **FAQ.html** file in the **Docs/** directory for more information. Also, details about customizing the installation are given in Section 14 below and in the **README** file in the package's tar file.
The configuration script relies on a recent addition to the R **`INSTALL`** script. The changes makes the location into which the package is being installed available to the configuration script and make procedures. If you have a very recent copy of R (the development version 1.2.0 or higher) the changes will be available and all will work as indicated above. If you are using an older version, you should set the shell variable `R_PACKAGE_DIR` before invoking the **`R INSTALL`**. The value of this variable should be either

- the value specified via the `-l` argument with `/Java` appended, or
- if the `-l` is omitted, `$R_HOME/library/Java`

So for example, if you would have issued the command

```
[]
  R INSTALL -c -l ${HOME}/Rpackages Java_1.0.tar.gz
```

then set the value of `R_PACKAGE_DIR` to `$HOME/Rpackages/Java`.
Given that the package has been installed, one can make the functions available to the R session via a call to the function *library()*.

```
[Library]
 library(Java)
```

4 Initializing the Java Virtual Machine

In order to execute Java commands, one must first create a Java Virtual Machine in which these commands are interpreted. This is done via the *.JavaInit()* function. This not only initializes the JVM, but also the necessary support for executing Java expressions from R/S. This is done by creating an embedded Omegahat interpreter.
In many cases, no arguments need be supplied to *.JavaInit()*. A simple invocation such as

```
[]
.JavaInit()
```

is sufficient to gain access to all of the core Java and Omegahat classes.
Just as when running the regular **`java`** command, we can specify arguments that customize the specific instance of the JVM. These include the classpath and system properties (usually passed as `-Dname=value`). The *.JavaInit()* allows the caller to specify a list of JVM parameters via its *config* argument. The default version of this list is stored in the R object *.javaConfig()*. The list is expected to have any or all of the elements named `classPath`, `properties` and `libraryPath`. (The library path is for use when loading JNI code from Java classes via the `loadLibrary()` method of the `System` class. The default will suffice unless you are using JNI and if that is the case, you know what this means!) Unless one specifies a second argument (*default*), the values specified in the *config* list are prefixed to the defaults in *.javaConfig()*.
The `classpath` element in the *config* argument should be a character vector in which each element identifies a directory, a jar file or a URL. In other words, it is *not* a colon separated string that one usually provides to Java via the `-classpath` argument or the `CLASSPATH` variable.

```
[]
.JavaInit(list(classPath=c("/home/duncan/MyJavaClasses",
                           "/home/duncan/Java/colt.jar")))
```

If no `classPath` entry is passed to *.JavaInit()*, we use the value in the `CLASSPATH` environment variable and split it into the individual components.
The default values in *.javaConfig()* are computed and fixed when the R library is installed These contain the entries of the default classpath necessary to locate the classes used to run the embedded Omegahat interpreter and also the system properties that control how that interpreter is created and behaves (e.g. class names for components, etc.).
One can specify additional system properties using this mechanism. The properties are given via named character vectors.

```
[]
.JavaInit(list(properties=c(RVersion=paste(version$major, version$minor,sep="."),
                            myProperty="understood by some class")))
```

One can control how the Omegahat interpreter behaves by specifying values for properties it uses. For example, the interpreter looks for Omegahat scripts by searching in elements of a path given by the property `OmegahatScript-SearchPath`. One can specify this as a colon-separated (actually this is platform dependent) list of directories, jar files, URLs, etc.

```
[]
.JavaInit(list(properties=
               c(OmegahatScriptSearchPath="/home/duncan:/tmp/scripts.jar")))
```

Properties that are understood by the JVM's initialization can also be supplied.

```
[]
.JavaInit(list(properties=c("JAVA_COMPILER"="NONE")))
```

Note that to satisfy R/S, we must quote the name of the property in this example to "escape" the underscore in the property name.
Obviously, we can specify both properties and classPath elements simultaneously, e.g.

```
[]
 .JavaInit(list(properties=c("JAVA_COMPILER"="NONE"),
            classPath="/home/duncan/Java/colt.jar"))
```

5 Executing Java Commands/Expressions

Now that the JVM is running and the Omegahat interpreter is available, we can invoke Java commands. We start with simple ones and gradually increase the utility and complexity, illustrating the different aspects of the interface.
First, we'll invoke a static method. The Java class `System` provides a method `getProperty()`. This provides access to a collection of global name-value pairs. Both the names and the corresponding values are strings. These properties provide information about the classpath in effect, the version of the JVM, Java language, the directory separator, the current directory, and so on.
To call this method, we use the *.Java()* function. This expects an object on which to invoke a method and the name of the method. In this case, the object is the class `System`. Because of the way the Omegahat interpreter works, we can specify just the name of the class.
The name of the method is `"getProperty"`.
The next arguments are the ones that are to be passed to the Java method. In this case, it is a single string naming the property to be retrieved. Let's get the value of the `java.class.path` property.
At this point, we have all the necessary components of the call and issue it with the following R expression:

```
[]
.Java("System", "getProperty", "java.class.path")
```

The result is an R character vector of length 1 - a string. It will probably be something like (but with different directories!)

```
[]
[1] "/home/duncan/bode/Rpackages/Java/org/omegahat/Jars/Environment.jar:
/home/duncan/bode/Rpackages/Java/org/..:
/home/duncan/bod\e/Rpackages/Java/org/omegahat/Jars/antlr.jar:
/home/duncan/bode/Rpackages/Java/org/omegahat/Jars/jas.jar:
/home/duncan/bode/Rpackages/Java/org/omegahat/Jars/jhall.jar"
```

How does this command work? Well, the first argument is the object on which to invoke the method. In this case, it is a static method and so we pass the name of the class. The full name of the class is `java.lang.System`, but we can specify a partially qualified class name thanks to the Omegahat interpreter. (More on partial class names in a moment.) The second argument is the name of the method to be invoked. And finally, we give a list of arguments via

the ··· argument to *.Java()*. In this example, there is only one argument – the name of the property whose value we want.
The Omegahat interpreter resolves the object whose method is to be invoked, in this case the class `System()`. Not finding an object/variable in its databases (like the R/S search path) named *System()*, it looks for a class with that name and finds `java.lang.System`.
When the arguments are passed to Java from R via the internal C code, they are converted to Java objects. Table 1 shows how the basic types are converted from R to Java values. In this example, the character vector (`"java.class.path"`) is converted to a Java `String`.
At this point, Omegahat has control and performs the complex task of finding the most appropriate method named `getProperty()` in the class `System` that takes a single argument of class `String`. There are in fact only two `getProperty()` methods in this class

```
[]
public static java.lang.String java.lang.System.getProperty(java.lang.String)
public static java.lang.String java.lang.System.getProperty(java.lang.String,
                                                             java.lang.String)
```

and only one takes a single argument. Hence, the match is done relatively easily.
Finally, Omegahat decides how to return the value of this method to the R engine. Again, in this case, it is simple since the object is a `String` and is converted to an R character vector of length 1.
The first time you call the *.Java()* function, you may notice that things are very slow. This is because the Omegahat has to typically resolve a class (e.g. `System` in our example) and to do this correctly it must construct a list of all possible classes. On subsequent calls this list has already been computed and these will execute quickly.
You can avoid the construction of these class lists (and the delay) by giving the fully qualified class name (`java.lang.System`).
To illustrate using different types, but still in the context of static methods, we can use the Java's random number. The class `Math` provides a method `random()`. We call it with no arguments and it returns a numeric vector of length 1 corresponding to the `double` return type of that method.

```
[]
.Java("Math", "random")
[1] 0.1194042
```

While we have concentrated on invoking methods, we can also access fields in a Java class (or object). The Omegahat interpreter determines whether the name given in the second argument of the *.Java()* function call identifies a method or a field. In the case of a field, the Omegahat interpreter returns its value. (It checks whether the fields is publically accessible.)
The following example illustrates how we can access the (static) field `PI` in the class `Math`. (Of course, we could have asked R, but where would the fun be in that!)

```
[]
.Java("Math", "PI") [1]
3.141593
```

6 Calling Omegahat Evaluator Methods

Before we turn our attention to creating new Java objects and invoking their methods, we can discuss invoking non-static methods without having to create new objects. In calling *.JavaInit()*, one causes the Omegahat interpreter to be instantiated. The evaluator can be referenced in a call to *.JavaInit()* by the string `"__Evaluator"` (with *two* underscores). The evaluator itself provides many utilities and is a specialized version of the basic interactive Omegahat evaluator tailored for this inter-system communication. It has many methods/functions that make managing the session easy as well as being useful in their own right. We do not want to change the focus of this document to explain the Omegahat interpreter. Instead, see the different online tutorials, API documents, etc. at `http://www.omegahat.`

`org`. Additionally, it is convenient to experiment either in an interactive Omegahat session or from R. (These are almost the same.) One can inquire what methods a class has by using the evaluator's own `getMethods()` method. This takes the name of a class and returns a list containing a description of the different method supported by that class.
How do we invoke a method on the evaluator itself? We use the *.Java()* function and specify the string `"__Evaluator"` (or alternatively `NULL`)
We can ask the evaluator to return the fully qualified name of a class that we specify by name. The method `expandClassName()` provided by the evaluator does this. Again, it is a simple method invocation. We pass it a string identifying the class of interest.

```
[]
.Java("__Evaluator","expandClassName", "JFrame")
[1] "javax.swing.JFrame"
```

6.1 Discovering Java Methods

We can also ask what methods a class has. The evaluator's `getMethods()` method returns a list of each of whose elements contain a description of a Java method, providing its name, the number and type of its arguments and its return type. (We will talk more about reflectance later. See Section **??**.)

```
[*]
m <- .Java("__Evaluator","getMethods", "OmegaInterfaceManager")
```

We can find the names of the different methods by extracting the `name` element within each element of the list.

```
[*]
nms <- sapply(m, function(x) x$name)
nms
```

Let's look at the method we just called. We extract all the elements whose `name` element is `"getMethods"`. This returns a list of length two corresponding to the overloaded methods with the same name in the evaluator's class. Note that we called the second one (in the output below). We know this by looking at the `Parameters` element of each and noting that the first expects a `Class` object, while the second expects a `String`.

```
[]
 m[nms == "getMethods"]

[[1]]
[[1]]$name
[1] "getMethods"

[[1]]$"Declaring class"
[1] "org.omegahat.Interfaces.NativeInterface.OmegaInterfaceManager"

[[1]]$Parameters
[1] "java.lang.Class"

[[1]]$Modifiers
public
     1
```

```
[[1]]$Exceptions
character(0)

[[1]]$"Return type"
[1] "[Ljava.lang.reflect.Method;"


[[2]]
[[2]]$name
[1] "getMethods"

[[2]]$"Declaring class"
[1] "org.omegahat.Interfaces.NativeInterface.OmegaInterfaceManager"

[[2]]$Parameters
[1] "java.lang.String"

[[2]]$Modifiers
public
     1

[[2]]$Exceptions
[1] "java.lang.ClassNotFoundException"

[[2]]$"Return type"
[1] "[Ljava.lang.reflect.Method;"
```

7 Basic Non-Primitive Conversion

As a last detoure before we turn to creating Java objects, we discuss how non-primitive objects are returned to R. The Omegahat evaluator offers a mechanism for finding files by searching different directories and append different extensions. (This is implemented in the *FileLocator* class.) This method is `findFile()` and it is a reasonably intelligent facility that can look inside jar files as well as directories. We will ask it to find the file OmegaInit in the **Environment.jar** file and assign the result to an R variable, *f()*.

```
[]
f <- .Java("__Evaluator", "findFile", "OmegaInit")
```

What should the contents of *f()* be? A file name? the jar file name and the expanded name of the entry within the jar file? If it is either of these, how can we use this, e.g to read the contents of the file, find its date, etc. without being forced to re-locate it? There is no clear answer to these questions, but a general mechanism allows us to give a clear and unambiguous approach.
The object returned by this particular method call is a Java object of class `org.omegahat.Environment.IO.ArchiveEntry`. Since there is no clear way to convert it to an R object, the Omegahat interpreter stores the value in one of its databases. It then creates a proxy object that contains sufficient information to identify the real object now stored away and returns that proxy. The low-level C code that implements the R-Java bridge knows how to convert these special Java proxy objects and create R objects from them.
So now we know what the R object *f()* should look like. It is a reference to a Java object and looks like the following.

```
[]
f
```

```
$key
[1] "2"

$className
[1] "org.omegahat.Environment.IO.ArchiveEntry"

attr(,"class")
[1] "AnonymousOmegahatReference"
```

It contains information about the class of the Java object to which it refers. This is in the `className` field. It also contains the key or name by which Omegahat knows it. This is the name used to store the real object to which this proxy refers (e.g. the *ArchiveEntry* from the earlier `find()` method call).
The class of this variable is `AnonymousOmegahatReference`. This indicates that not only is it a reference to an object managed by Omegahat, but also that it was not given a name by the R call that created it. Instead, Omegahat has generated a unique name and stored it in a special database – the anonymous database.
We could have forced Omegahat to use a particular name for the result of an object. We do this by providing a value for the argument *.name*. In this case, Omegahat stores the resulting object in its default database using the name specified by the R call and returns an object of class `NamedOmegahatReference`.

```
[*]
f <- .Java("__Evaluator", "findFile", "OmegaInit", .name="myFile")
```

Is there an advantage to using named references rather than anonymous ones? There is little or no difference in speed. One benefit is that you can then use the name directly as the first argument in a call to *.Java()*, such as

```
[]
 .Java("myFile", "size")
[1] 369
```

Additionally, when using *.OmegahatExpression()* which allows one to evaluate an Omegahat or Java expression one can refer to the variable by name rather than having to substitute it on the R side. (See Section 12 for more details.)
Perhaps the most important use of named references is that by specifying a name, Omegahat will not attempt to convert the resulting object. This is useful when we want to avoid the conversion of an object that we will use in a subsequent Java call. For example, we may want to display the names of all the objects stored in the default Omegahat database (the named objects themselves) in a Swing `JList`. The array of names is retrieved by calling the evaluator's `objects()` method. Then pass this to the `JList` constructor. We can avoid the conversion from Java to R and then again from R to Java by specifying any name as the value of the *.name* argument when calling the `objects()` method.

```
[]
 x <- .Java("__Evaluator", "objects", .name="anyName")
 .JavaConstructor("JList", x)
```

To make certain you understand the conversion mechanism, take a moment to consider what the same code would do if the *.name* argument was not specified.
We should note that one can obtain the same effect using the *.convert* argument without specifying a unique name that doesn't conflict with an existing entry. See Section 11.5.
Another important use of the *.name* argument is when we must guarantee that we are using the same Java object in different calls, rather than two different Java objects that have the same values. For example, suppose we wish to create two new Java objects and have them share a single array of `String` objects. If we convert the same R object on two occasions, we will not obtain the *same* Java object, but just a duplicate. In this way, we will not be (easily) able

to pass the same Java object to the two new objects. Storing an object in the Omegahat database and referring to it in subsequent calls will guarantee that it is the same object.
Perhaps one of the major disadvantages of using the *.name* argument is the potential to conflict with another name. For example, suppose you use a function which decides to store a Java object using the name `"x"` and then call another function that also uses this name. In this case, the second assignment will overwrite the earlier one and the Omegahat database will contain the second object. The first will have disappeared. This is very similar to the problems that are encountered in *S* when people use frame 1 to store objects created in one function call that are needed in other function calls (e.g. the model and trellis code).
In summary, use the *.convert* argument to suppress Omegahat's conversion to an R object unless you are using the *.OmegahatExpression()* and need to refer to the object as an Omegahat variable in a string version of the Omegahat expression. Even in this case, you can use the *substitute()*-like functionality of *.OmegahatExpression()*. Basically, don't use the *.name()* argument from within a function. Use it only as part of a user-level interactive command where the user is responsible for the entire management and selection of Omegahat names.

7.1 Named Arguments

As we have seen, we can avoid unnecessary conversion of by explicitly storing the return value in the named Omegahat database. This also allows the use of unique Java references within Omega where necessary. The same logic applies to arguments passed from R to Java methods and constructors via the *.Java()* and *.JavaConstructor()* functions. Again, suppose we want to create two separate lists but with the same `String` array as the argument for both. We could transfer the object from R to Java in one step and the call *.JavaConstructor()* to create the two lists and refer to the previously created `String` array in both. However, the R-Java interface provides a simpler mechanism. One can optionally provide names for the arguments given in the $\cdots$ argument of *.Java()* (and *.JavaConstructor()*). For each of these named arguments, the Omegahat assigns the resulting Java object to the named database and makes it available in exactly the same way as it does named return values.
In our example, we could then achieve the same result with just two inter-system calls. Suppose that *x()* is the character vector that we want to convert to a Java `String` array and share between both lists. Then we pass this as the argument to the first constructor of the `JList` and give it a name – `myName`. Then, entirely within R, we create an object representing a named Omegahat reference with `myName` as the key. We use the function *omegahatReference()* to create this locally generated version of the reference. Then, we pass this reference as the argument to the second constructor.

```
[]
 x <- letters
 list1 <- .JavaConstructor("JList", myName = x)
 r <- omegahatReference("myName")
 list2 <- .JavaConstructor("JList", r)
```

Currently, there is no direct way in R to effect the other approach whereby we convert the object, get a reference to it and then create two `JList` objects. Instead, we use the *.Java()* method and explicitly assign the object to the Omegahat database. This can be done with the following command:

```
[]
 ref <- .Java("__Evaluator", "identity", x, .name="myName")
```

Then we can use this reference as the argument to each of the constructor methods.

7.2 Garbage Collection & Querying the Omegahat References

We know about the way Omegahat stores objects in its local databases and the resulting named and anonymous references in R. One difficulty with this reference approach is that Omegahat cannot determine when the R session has no use for an object. (There are some exceptions, but this is true in general.) As a result, the R user is responsible for releasing the objects he or she no longer needs. This is similar to R objects that are assigned and no longer used.

The Omegahat evaluator allows one to obtain the names of the different references it is managing. One of the `objects()` methods takes a single boolean argument which indicates that it should return the names from the anonymous reference database (`false`) or the regular named database (`true`).

```
[*]
   # the anonymous reference database
 .Java("__Evaluator", "objects", T)

   # the named reference database
 .Java("__Evaluator", "objects", F)
```

Additionally, one can obtain a complete listing of the references in each of these two databases. These listings include the class of the object to help one identify it. The evaluator method that gives these listings is `getReferences()`. Again, it expects a boolean value as its only argument - `true` for named references and `false` for anonymous references.

```
[*]
 .Java("__Evaluator", "objects", F)
```

8 Creating Java Objects

At this point, we have discussed how to invoke methods and access fields. We have even had a brief example of how to find out about the available methods. But now we turn our attention to the power of the R-Java interface and the ability to create instances of arbitrary Java classes. We first focus on Java classes and then discuss Java arrays. The two functions of interest are *.JavaConstructor()* and *.JavaArrayConstructor()*.
The *.JavaConstructor()* takes a class identifier and a collection arguments that are passed to an appropriate constructor of that class. This is very similar to the *.Java()* function except that there is no need to specify the method name. This is because the constructors have an implied name. The class identifier given as the first argument to *.JavaConstructor()* can be either of the following:

- a string giving the (potentially partially qualified) name of a class. The class names is resolved and expanded by the Omegahat evaluator.
- a Java object returned from a previous call to *.Java()* that returned a `Class` object.

The arguments passed via the $\cdots$ argument of the *.JavaConstructor()* are handled exactly as they are in the *.Java()* function. They are converted to Java objects and the passed to the Omegahat interpreter along with the class identifier. The appropriate constructor is identified and the newly created object returned. If no converter is found to convert this new object to an *R* object, an reference to Java objects is returned to the R engine. As with the *.Java()* function, if a value is passed for the *.name* argument, a named reference is returned and one can refer to that object in subsequent *.Java()* and *.JavaConstructor()* calls with that same string. If no *.name* argument is specified, an anonymous reference is returned and one should use the resulting R object to identify the object in future calls.
The following commands illustrate many of the different aspects of using *.JavaConstructor()*. We start by creating a Swing button (`JButton`). We invoke that class' constructor that expects a string giving the text to display within in the button. The result is an object that cannot be meaningfully converted by the Omegahat evaluator to an R object. Accordingly, it is returned as a reference. Since there is no *.name* argument, we get a anonymous reference to a Java object and assign that reference to the R variable *b()*.
Next, we want to display that button in a window. We can use the Omegahat convenience class *GenericFrame*. We create an instance of this class and give it a reference to the button and a logical value (**F**) that indicates that we do not want the window to be immediately displayed when it is created. These are the arguments to the constructor. The *.name* means that the Omegahat evaluator assigns the resulting *GenericFrame* instance to its regular/default database and returns a reference to it.

Note how the R variable *b()* is passed from R to Omegahat in this second constructor. Omegahat resolves this reference using the name and database identifies stored in it.
The second pair of expressions access the newly created object. The first invokes a method in the button, changing its background color. (Note how we access the static field in the class `Color`.) The second expression illustrates how we can use the name `"myWindow"` to refer to an object create previously with the *.name* argument.

```
[*]
 b <- .JavaConstructor("JButton", "A button")
 .JavaConstructor("GenericFrame", b, F, .name="myWindow")

 .Java(b, "setBackground", .Java("Color", "red"))
  .Java("myWindow", "setVisible", T)
```

9 Creating Arrays

One can create arrays of objects, including arrays of arrays or multi-dimensional arrays. We currently use a different function – *.JavaArrayConstructor()* – for this purpose. This takes the name of the class of the element type for the array and the length of each dimension. For example, we can create an array of `String` objects with length 3 as follows.

```
[]
.JavaArrayConstructor("String", 3)
```

We can create a two-dimensional array consisting of 4 arrays, each of length 3 to contain `Class` objects.

```
[]
.JavaArrayConstructor("Class", c(4, 3))
```

Ragged arrays and arrays all of whose dimensions are not known at creation time can be created by specifying the length of a particular dimension as 0. (Unfortunately, omitting a value as in `c(4,)` does not work due to the definition of the *c()* function.) So the following commands create an array of String arrays in which the first element has 3 entries and the second has length 4.

```
[]
r <- .JavaArrayConstructor("String", dim=c(2,0))
.JavaSetArrayElement(r, .JavaArrayConstructor("String", dim=3),1)
.JavaSetArrayElement(r, .JavaArrayConstructor("String", dim=4),2)
```

Unfortunately, in this release, there is no simple way to initialize the contents of a nested array. This is because of the ambiguity of conversion.
We can access the elements of the two-way array above using *.JavaSetArrayElement()* and *.JavaGetArrayElement()*. First we set the second value of the first array (i.e. 1,2). Then we retrieve the value of that same element. And then we get the top-level elements of the two-dimensional array. Each of these are arrays themselves and automatically converted to character vectors.

```
[]
.JavaSetArrayElement(r, "A test", 1,2)
.JavaGetArrayElement(r, 1, 2)
.JavaGetArrayElement(r, 1)
.JavaGetArrayElement(r, 2)
```

10 Advanced Converters

The R-Java/Omegahat interface understands how to convert certain basic R types to Java values and vice-versa. Table 1 shows the relationships between the different R and Java types. But we have also seen an example of how we converted Java `Method` information to R when we invoked the `getMethods()` method of the evaluator (see 6.1). How does R know how to convert Java `Method` objects? The answer is, of course, we told it? Specifically, we wrote a C routine that takes a Java `Method()` object and converts generates an appropriate R object. The R-Java interface allows the R user to manage the list of such converters. The function *getJavaConverterDescriptions()* allows the user to see both how many converters are currently registered and also obtain a description of each of these.
There are two sets of converters: from Java to R and from R to Java. The *getJavaConverterDescriptions()* by default returns descriptions of both sets of converters. The default set of converters is given below.

```
[]
getJavaConverterDescriptions()
$fromJava
[1] "Converts any Java InterfaceReference"
[2] "class == java.lang.reflect.Method"
[3] "class == java.lang.reflect.Constructor"
[4] "instanceof java.util.Properties"

$toJava
[1] "RFunctionListener"
```

The descriptions attempt to indicate on what type of object they operate. Consider the `fromJava` list. The first entry indicates that the associated converter will handle an object that is derived from the Omegahat reference class *InterfaceReference*. This uses the equivalent of the `isAssignableFrom()` method in Java to determine whether an object can be assigned to a variable of a particular class - in this case *InterfaceReference*.
The next two descriptions indicate that they are prepared process objects that are of class `Method` and `Constructor`, respectively. They will not process objects of classes derived from these classes. (This is not an issue since these are **final** classes and so cannot be extended.)
The final description indicates that the converter will process any object for which the Java expression

```
[]
    obj instanceof Properties
```

returns true. For the class `Properties`, this is the same as the `isAssignableFrom()` comparison. However, if the class were an interface rather than `Properties`, it would match any class that implemented that interface.
These descriptions indicate that there are two aspects to conversion. First, a converter must determine whether it is capable of processing a particular object. If it is, then it must perform the actual conversion. We separate these two actions into different routines: the matching routine and the conversion routine. Additionally, we provide implementations of the 3 basic types of conversion: instance of, assignable from an class equality.
When an object is being returned from Omegahat, that system determines whether the object is convertible or not. and passes it the internal routines to convert. It then iterates over the list of converters until the matching routine of one of them indicates that the associated converter will perform the conversion. Then the basic converter engine calls that converter and the result is an appropriate R object.
Conversion involves both sides of the interface knowing about a class. Obviously, the R engine must know how to convert a class before it can be converted. But equally importantly, the Omegahat engine must be told that a class is considered convertible and whether derived classes or a class implementing an interface that is considered convertible. This is achieved by Omegahat having a reference to *ConvertiblClassifierInt* object. An object that implements this interface (see *BasicConvertibleClassifer*) maintains a table of classes that it knows about and how matching should be done for that class. The default implementation of this interface is a hash-table that stores the classes that are considered convertible and an integer indicating which type of matching to perform when an object is being compared to that class entry. This classifier works recursively when given an array. It looks at the type of element in the array and determines whether that is convertible or not. (Note that nested arrays are not currently considered convertible.)

The key point is that when we add a C-level routine to convert a Java object to an R object, we must also inform the Omegahat *ConvertibleClassifierInt* that this class is considered convertible and how (i.e. the appropriate matching operation).
The 4 default converters registered when the JVM is initialized include a converter for the `Properties`. This means that when the result of a Java method returns a `Properties` object, the appropriate converter C routine will be called and create an R object. In this case, it creates a named character vector: the names are the keys in the properties table and the values are the corresponding entries. This means that we can retrieve any `Properties` table from Java including the `System` properties.

```
[]
 jprops <-  .Java("System","getProperties")
 jprops[["user.home"]]
[1] "/home/duncan"
```

To register a converter in C code, one specifies the routine that does the conversion; a routine that indicates whether it is prepared to process a given object; some user defined data object that is stored with the converter and passed to it and the matching routine each time they are called (e.g. a class type against which to compare the class of the object being converted) ; a description which is used to describe the current converters as in *getJavaConverterDescriptions()*. Finally, the call to register a converter can indicate whether the basic conversion mechanism should process arrays by calling this for each element or not. (Needs more explanation!)
While conversion routines can be registered in C, they can also be specified in R via the *setJavaConverter()*. This expects the same set of arguments as the C registration, but rather than receiving function pointers, it expects the names of C routines for the conversion and matching routines. An example of such a call is shown below and can be executed. (The function *.RSJava.symbol()* merely converts its argument to match the C routine name - `RS_JAVA_RealVariableConverter()` in this case.) The *matcher* argument identifies one of the built-in class comparators rather than the name of a C routine that performs the comparison of the object's class and target class of the converter. The *userObject* is used to parameterize that matching function. It specifies the name of a Java class which is expanded automatically to, in this case, *org.omegahat.DataStructures.Data.RealVariable*. This is then stored with the matching routine and is used by it to determine if the object being converted is assignable to an object of that class. This particular converter translates the basic Omegahat *RealVariable* data structure to a numeric vector in R.

```
[*]
 val <- setJavaConverter(.RSJava.symbol("RealVariableConverter"),
                         matcher="AssignableFrom",
                         autoArray=T,
                         description="Omegahat RealVariable to numeric vector",
                         userObject="RealVariable"
                        )
```

The final argument of the *setJavaConverter()* function is *register*. This is expected to be a logical value indicating whether this function call should also notify the Omegahat *ConvertibleClassifierInt* that the class for which the R converter is being registered should also be considered convertible by Omegahat. This is a necessary step if Omegahat is to ever allow objects of this type to be passed to the low-level conversion mechanism. If it does not know that the class is considered convertible, it will simply return a reference to the object.
The default for the *register* argument is currently **T**. If the particular call to *setJavaConverter()* does not specify the appropriate information for Omegahat to digest, one can register that the class is convertible in a separate expression via the function *setJavaConvertible()*. This is a direct way to add the class to Omegahat's list of convertible classes. The arguments this expects are the name of the class and how classes related to it are to be treated. This is the equivalent to the specification of the class matching for the converter itself: instance of, assignable from, or direct equality of class.
The *setJavaConvertible()* function also allows one to remove a class from the list of classes considered convertible. This allows one to instruct Omegahat not to attempt to convert a type of object. In this way, we need not remove

a converter from the C-level converter tables, but need only avoid attempting the conversion. The following causes Omegahat to treat the `javax.swing.JButton` class as non-convertible and will return a reference to any such object.

```
[]
 setJavaConvertible("JButton", F)
```

Just as we can discover what converters are registered in the C code to convert between R and Java objects, we can also query what Java classes Omegahat considers as convertible. We can ask the Omegahat evaluator to give us this information using its `getConvertibleClasses()` methods.

```
[]
 .Java("__Evaluator", "getConvertibleClasses")
```

This returns a character vector listing the Java classes that the Omegahat interpreter will attempt to convert. We must also query how it performs the matching so as to determine how derived classes and those that implement interfaces listed in this group are handled. This information is available from the evaluator's *ConvertibleClassifierInt* object which can be retrieved via the method `getConvertibleClassifier()`. The matching mechanism for each of the classes in this list is available via methods in that class.

Writing a C-level converter relies on low-level JNI code. It is not very complicated, but requires some experience and familiarity. It is similar to programming routines to be called from R or S via the *.C()* and *.Call()* functions. However, there are at least 2 books to help in the endeavor and clarify issues. Also, there are several different examples in the code. Additionally, one can call the converters for the basic types that the R-Omegahat interface already performs. In this way, a converter need only access the appropriate fields and call the necessary methods in the object being converted to get the primitive types and pass these to existing code to create R objects. See the files **ConverterExamples.c** and **Converters.c** in the distribution. Also, see Section **??** for more information about the different C routines to access the Java environment from within C code loaded by R.

In spite of the fact that the converters are not hard to write, we understand that most people will not relish the idea of compiling, linking, loading and debugging C code that connects two systems which the may not understand that well. (Why ever not? It sounds ideal:-)) As a result, we have added the capability of specifying functions for both the converter and the matching mechanism. At present, you must use functions for both. You *cannot* use the built-in C routines to do the matching and an R function to perform the actual conversion. (This will be changed in the future so that one can mix types). There is an example (reproduced here) in the **functionConverters.R** file in the **examples/** directory that is installed with the package code.

One uses the *setJavaFunctionConverter()* to register the pair of converter and matching functions. Each of these functions expects two arguments. The first is the object to be converted. This comes as an anonymous reference and one can use the *.Java()* to query its contents and perform other operations necessary to fulfill the task of the function. The second argument is a string which gives the name of the class of the object to be converted. This is the full class name.

The first example below shows how we might provide a silly conversion of a Swing button object. The matching function compares the class name given to it with the string `"javax.swing.JButton"`, the full class name of the type we are prepared to convert with the other object. The converter function uses the *.Java()* function to retrieve the text displayed on the button object and also its action command (the value passed to an event). It returns these as the result of the conversion in the form of a character vector.

```
[]
setJavaFunctionConverter(function(x, className) {
           print("This is a silly converter for a JButton")
           val <- .Java(x,"getText")
           val <- c(val,.Java(x,"getActionCommand"))
           print(val)
           return(val)
    }, function(obj, className){
     ok <- className == "javax.swing.JButton"
```

```
     cat("In match:",ok,"\n")
     return(ok)
})

setJavaConvertible("JButton")
.JavaConstructor("JButton", "testing")
```

A second example provides mutable state for the converter pair using a closure. We will use this to provide a converter for the Omegahat class *RealVariable*. You can compare this with the native C-level converter mentioned earlier. We create a closure definition which has an instance-specific variable *n()* that counts the number of times its been called. This closure generator is named *realVariableConverterHandler()*. We create an instance of it by calling the function and assigning it to *rvCvt()*.

```
[]
realVariableConverterHandler <-
 function() {
   n <- 0
   cvt <- function(obj, className) {
      n
<<- n + 1
      .Java(obj, "getValues")
   }
   matcher <- function(obj, className) {
     return(className == "org.omegahat.DataStructures.Data.RealVariable")
   }
   return(list(converter=cvt, matcher=matcher, count = function(){ n } ))
 }


rvCvt <- realVariableConverterHandler()
```

At this point, we can call *setJavaFunctionConverter()*. It expects 2 two functions as separate arguments. Thus, we extract each of the functions from the *rvCvt()*. This still allows them to share the closure's environment and hence access *n()*.

```
[]
setJavaFunctionConverter(rvCvt$converter, rvCvt$matcher)
setJavaConvertible("RealVariable")
```

And we can check how it works by creating an instance of the *RealVariable* class.

```
[]
.JavaConstructor("RealVariable", rnorm(10))
```

One result of this setup is that one can cache converted values for use in future conversion calls and also ensure that one returns objects that are identical in reference and not just value. The converter closure can return references to the same R object ensuring that modifications to it are visible to all that have access to it. (Of course, those objects must be closures themselves.)

At present, the support for functions is new and so the error handling is not in place. These functions should not generate an error. (You have been warned!!!!)

If we can register converters, we must also be able to remove them. This allows us to temporarily change how conversion is done by inserting a new converter, using it and then removing it and restoring the previous one. The

removeJavaConverter() allows one to remove an entry from the an internal list of converters, either the "from Java to R" or "from R to Java" lists. This *currently* takes the index of the particular converter in the list to be removed.

```
[]
> getJavaConverterDescriptions(F)
[[1]]
[1] "Converts any Java InterfaceReference"
[2] "class == java.lang.reflect.Method"
[3] "class == java.lang.reflect.Constructor"
[4] "instanceof java.util.Properties"
> removeJavaConverter(3)
class == java.lang.reflect.Constructor
                                    2
> getJavaConverterDescriptions(F)
[[1]]
[1] "Converts any Java InterfaceReference"
[2] "class == java.lang.reflect.Method"
[3] "instanceof java.util.Properties"
```

One can also pass the description of the converter to identify it. This has the advantage that subsequent identifiers don't change when we remove an entry. (This happens for example if we wanted to remove converters indexed 2 and 3. Having removed 2, the element 3 becomes 2 and we would remove 2 again.)

```
[]
> getJavaConverterDescriptions(F)
[[1]]
[1] "Converts any Java InterfaceReference"
[2] "class == java.lang.reflect.Method"
[3] "class == java.lang.reflect.Constructor"
[4] "instanceof java.util.Properties"
> removeJavaConverter(getJavaConverterDescriptions(F)[[1]][3])
[[1]]
[1] "Converts any Java InterfaceReference"
[2] "class == java.lang.reflect.Method"
[3] "instanceof java.util.Properties"
```

11 Foreign References

A significantly more complete example of this is given in the **Examples.pdf** and **Overview.pdf** files in the same directory as this one. Please read these.

We have discussed R objects that are references to Java objects. We now turn our attention to the flips-side of this communication mechanism: Java objects that are references to R objects. Just as there is no obvious way to represent an arbitrary Java object as an R object, there is a similar disparity in converting non-primitive R objects to Java objects. On some occasions, there is no available Java class to which we can convert the R object. On other occasions, we simply do not want to convert the object but have the same unique reference to be shared by both R and Java. In the same way as we store Java objects locally in Omegahat and export a reference to these to R, the R-Omegahat interface has a mechanism by which we export R objects to Java by storing them within R and exporting a reference to them.

We export an object to the Java/Omegahat system in two steps. The first step is to register it with a foreign reference manager. This is done via the function *foreignReference()*. Effectively, we are checking in our object and being given a receipt for it. The receipt can be copied, but always identifies the same object. The second step is to pass the

reference (or receipt) as an argument in one of the R-Java interface functions (i.e. *.Java()*, *.JavaConstructor()*, *.JavaArrayConstructor()*, etc.) The conversion mechanism recognizes the R object (by its class) and creates an appropriate representation on the Java/Omegahat side that still points to the R object.

Java is not quite as liberal as R is regarding what values can be passed to functions as arguments. Java is a strongly-typed language meaning that the types of a methods arguments are specified at compile time. To call a method, we must have an object of the appropriate class (or actually type to include primitives). Simply passing a generic R foreign reference to a Java method will not work. Instead, we must convert it to a Java object which is both a reference to an R object and also an object of the appropriate Java class to satisfy the strong typing.

An example may help. We return to the editing of an R data frame via a Java GUI. To pass the data frame to the *DataFrameViewer*, we must create a Java object that implements the interface *DataFrameInt*. This object must also maintain a reference to the R object by storing the name by which the foreign reference manager knows it. What we need to have happen is that this Java object should implement the methods require by the interface *DataFrameInt* by calling the corresponding function in R which has access to the data frame (either as part of a closure or passing the data frame as an explicit argument).

There is nothing magical about a Java method calling an R function. Again, it is part of the low-level communication mechanism that bridges the R-Java systems. The Omegahat class *RForeignReference* is an abstract class that anyone can inherit from and which provides a method which uses native code to call an R function. One gives this method the collection of arguments and the name of the function. This calls the R function and returns the result of the call to Java. Additional methods in this class help in converting the Java arguments to R objects and equivalently, converting the return value to Java. One can write new Java classes by extending this one and implementing different Java interfaces. Then, one can create instances of these classes using an R foreign reference as an argument to the main constructor, arming that object with knowledge of which R object on which to call the function.

Again, we return to the example of the data frame. Suppose we create a new Java class which extends *RForeignReference* and also implements the methods in the interface *DataFrameInt*. Each of these methods collects their arguments and calls the inherited `eval()` method with its own method name and these arguments. The native methods calls the appropriate R function. It does this by passing the request to a central R broker. This resolves the R reference passed to it by the *RForeignReference*. Then it looks to see if this is a list containing functions. If so, it looks for the function there by comparing the name of the method passed to it from the Java method call. If it finds one, it calls that function, passing it the arguments given it to it from the Java method.

This object that is a list a functions is usually the value exported as an R foreign reference. It is typically created as a closure with its own data instances. This is how we setup the data frame that we exported in this running example. We create a closure that has access to the data frame and has a function for each of the different methods in the Java interface *DataFrameInt*. When the Java methods is called, the corresponding R function is invoked and this has access to the closure's instance of the data frame. The R code for creating the closure looks something like the following:

```
[]
dataFrameClosure <- function(data) {

    # following are methods specified in
    # the DataFrameInt interface (inherited
    # from BasicDataFrameInt
  getVariableNames <- function() {
    colnames(data)
  }
  numObservations <- function() {
    dim(data)[1]
  }

  return(list(numObservations=numObservations, getVariableNames=getVariableNames))
}
```

11.1 Default Handler

The R object that manages the exported references and brokers the incoming function call requests from the Java method calls can be found by calling the R function *getJavaHandler()*. This is also a closure and maintains a list containing the different objects that are exported. It also has several functions which allow one to manipulate that list: creating, adding and removing objects, resolving references and dispatching function calls to the appropriate function and object.

11.2 Mutable State

By passing an R object that is a closure instance, subsequent calls from Java (and also R) to the functions within that closure can cause its contents to be changed. These changes are visible to all other functions and methods that call the closure's functions in the future. This means that we can use the *mutable state* of an R object in much the same way as a Java object.

An example of where this mutability is useful is when we export a data frame from R to Java. Within the Java framework, we display the contents of the data frame in a data-grid or spreadsheet-like editor. The user can edit individual cells. The Java editor assigns such modifications by invoking a method in the Java data frame representation. Since this calls the corresponding R function which has access to the original data frame on the R side, the changes are made to that one instance of the data values. In this way, we can edit the data in one system and have the changes available to us as they occur in the other system

An example of how all of this R foreign reference material can be used is below. Here, we create a Java GUI consisting of a button in a window. We arrange that when the user clicks on the button, an R function is called. This function counts the number of times the button was clicked. It does this by having a count value associated with, created as part of a closure. This is the *handler()* function below and *cb()* is the instance of this closure.

We connect this closure instance with the button and the user action callback by creating an instance of the Omegahat class *RManualFunctionActionListener*. This takes a reference to the R object as the only argument to its constructor. This allows it to identify that R object when calling the `actionPerformed()` function in R. The Java object's own `actionPerformed()` method is called when the button is clicked on by the user. This connection is established when we register the *RManualFunctionActionListener* as a listener for this button's events via the the call to `addActionListener()`. And that is all there is.

```
[Button Callback]
 handler <- function() {
   n <- 0
   actionPerformed <- function(event) {
     n
<<- n + 1
     print(event)
     print(n)
   }
   return(actionPerformed)
 }


cb <- handler()
ref <- foreignReference(cb,"btnCB")

l <- .JavaConstructor("org.omegahat.R.Java.RManualFunctionActionListener", ref)

b <- .JavaConstructor("javax.swing.JButton","Click me")
f <- .JavaConstructor("GenericFrame",b,T)

.Java(b,"addActionListener", l)
```

11.3 Dynamically Creating Interfaces

Even though it is relatively simple given the template above, it is tedious to have to manually generate Java classes to act as proxies for R objects or functions. Given the template above, one would think that we could automate the task, and indeed we can. Omegahat provides a class – *ForeignReferenceClassGenerator* – that provides this dynamic, automated compilation. One supplies it with

1. one or more interfaces to be implemented (either as strings or `Class` objects),
2. the name of the class to create, and
3. optionally, the name of the class from which the new class will inherit methods. By default, this is *RForeignReference*.

The resulting byte-code that defines the new class can be written to a file for use in future sessions and/or loaded into the running Omegahat session. Writing to a file (or stream) is done via the `write()` method of the *ForeignReferenceClassGenerator* object. The dynamic loading of the class is performed by passing the *ForeignReferenceClassGenerator* instance to the Omegahat evaluator's dynamic class loader. The Omegahat command

```
[]
  evaluator().dynamicClassLoader().defineClass(gen)
```

illustrates how this is done.
The Omegahat *ForeignReferenceClassGenerator* class is a specialization of a general dynamic compilation mechanism. See `http://www.omegahat.org/DynamicCompilation`.

11.4 Example

An example of how we can use this should help in understanding the power of this automation technique. Suppose we wish to receive events from a collection of Swing `JCheckBox` objects when the user selects any of them. (See `http://java.sun.com/docs/books/tutorial/uiswing/components/example-swing/index.html`CheckBoxDemo.
We define an R function that responds to such event notifications. This function sets the value of a variable within a closure to the action command (`getActionCommand()`) of the button which generated the event.

```
[Checkbox Event]
 checkbox <- function() {
    # the variable that holds the current setting.
    # Initial value is unset.
   value <- NULL

   itemStateChanged <- function(ev) {
      btn <- .Java(ev,"getItem")
      value
<<-  .Java(btn,"getActionCommand")
   }

  return(list(itemStateChanged=itemStateChanged, value = function() value))
 }
```

Given this function, we now have to have a Java class which is both an *RForeignReference* (for the `eval()` method that calls an R function) and a Java `java.awt.event.ItemListener`. We can now create an instance of *ForeignReferenceClassGenerator* to define the new class. We use the *.JavaConstructor()* to create the compiler-instance, and give it the name of the interface to implement (`java.awt.event.ItemListener`) and the name of the class to create. Since we are inheriting from the default base class *RForeignReference* we need not specify any additional arguments. We store a reference to the compiler object via the *R* variable *compiler()*

```
[ItemListener]
compiler <- .JavaConstructor("ForeignReferenceClassGenerator",
                             "java.awt.event.ItemListener",
                             "RItemListener")
```

When this returns (without error!), the class has been compiled. We might write it to the file **/tmp/RItemListener.class** via the *R* command

```
[*]
 .Java(compiler, "write", .JavaConstructor("File","/tmp/RItemListener.class"))
```

We can make it available to the Omegahat session for immediate use. This is what we need in this situation.

```
[*]
dyn <- .Java("__Evaluator","dynamicClassLoader")
.Java(dyn,"defineClass", compiler)
```

At this point, we have the necessary class and can create an R/Java listener using an closure instance created from calling the *checkbox()* function and the newly created *RItemListener* class.

```
[*]
  ref <- foreignReference(checkbox())
  listener <- .JavaConstructor("RItemListener",ref)
  .Java(jcheck, "addItemListener",listener)
```

In summary, all we had to do to create a suitable Java class to implement a Java interface with an R closure instance was

1. create a *ForeignReferenceClassGenerator*,
2. pass this to the evaluator's dynamic class loader,

In future versions, we can automate this. When one creates the Foreign Reference in *R* (via the *foreignReference()*) function, one can specify the interfaces that one wishes to implement. These are given via the *targetClasses* argument and stored in the reference. When the *RForeignReference* is created, it receives this information. When attempting to locate an appropriate method, the Omegahat mechanism has licence to match a parameter type that is in the list of potential target classes. It is at this stage that it defines an appropriate class that implements that Java interface or class and creates a new instance of it.
This functionality is not in this release as it does interfere with the generality of the method dispatching. As more feedback is received about how people are using the interface, we will hopefully understand more about the appropriate default, implicit conversion.

11.4.1 What is the Class Compiler Doing?

What is that we really do manually? We start by defining a new class that extends *RForeignReference*. Then, we "copy" the constructors from *RForeignReference*, modifying them to use the name of the new class and to pass the arguments to the base class' constructors. Then, we iterate over the methods in the interface being implemented and define a corresponding method for each. The body of each of these methods simply packages the arguments into an Omegahat *List* class and calls the inherited `eval()` method with this argument collection, the name of the method, the signature of the method and the type of the expected return value.
Because of reflectance, we can iterate over the methods of the interface being implemented programmatically. Each `Method` definition (obtained via the `getMethods()` method) provides us with information about its name, the number and types of its argument, return type (and what exceptions it throws). This is all we need to know to create

the instructions that collect the arguments into a *List*, compute the signature and provide the method name and return type. The Jas classes then allow us to specify the instructions and generate the Java byte-code.
The constructors are handled in a very similar fashion. We use reflectance to query the constructors in the base class (e.g. *RForeignReference* by default). Again, we iterate over these and generate the appropriate call to `super()` which passes the arguments to the inherited constructor.

11.5 The `.convert` Argument and `identity` Method

A slightly different use of the *.convert* is when we want to use a constructor for its computational side-effect rather than creating a Java object for use in future Java method calls. For example, suppose we have a converter from Java to an R representation for the class *StatDataURL*. When we create a *StatDataURL*, on some occasions we just want the contents of the URL and on other occasions we want the Java object. We use the *.convert* argument of the *.JavaConstructor()* to differentiate between these two circumstances. In the first of these case, we give the value `TRUE` (the default) so that the default conversion is applied. When we wish to prohibit this conversion, we specify the value `FALSE` for this argument and have the Omegahat evaluator return an anonymous reference to the resulting object.

```
[]
setJavaConverter(function(x) {.Java(x,"getValues")},
                  match="Equals",userObject="StatDataURL")
contents <- .JavaConstructor("StatDataURL",
                             "http://www.omegahat.org", .convert=TRUE)
reference <- .JavaConstructor("StatDataURL",
                             "http://www.omegahat.org", .convert=FALSE)
```

The following creates an array and leaves it in the Omegahat database. Then we insert values into the array's elements by operating on the array with other Java methods. And finally we retrieve the elements of the array as a character vector by using the `identity()` method of the Omegahat evaluator.

```
[]
r <- .JavaArrayConstructor("String", dim=c(3))
   # populate the first r.length elements with the
   # names of the states.
.Java("States","fillIn", r)
.Java("__Evaluator", "identity", r)
```

12 Omegahat expressions

It can be tedious to have to constantly type `.Java` and `.JavaConstructor`. It would be more convenient to have a syntax that was more Java-like of the form

```
[]
   obj.method(arg1, arg2, arg3)
```

Well, obviously we cannot use the dot (.) notation since this is a legitimate character in R. We can borrow the idea we use in S version 4 which is to think of the $ operator as being the field or method accessor of a Java object.

```
[]
  obj$methodName(arg1, arg2, arg3)
```

Java fields can be accessed in a similar manner.

```
[]
col <- .Java("__Evaluator", "findClass", "Color")
col$red()
col$blue()
```

This syntax is in the spirit of both Java objects and also of accessing data and methods within a closure instance.
Another syntax is that of Omegahat. The interactive Omegahat language is similar to both Java and R and allows one to invoke Java commands dynamically, i.e. without compiling them first. The R-Java interface provides access to the Omegahat evaluator and hence we can send it a string which contain valid Omegahat expressions to evaluate. All of the functionality available via the *.Java()* is available within such expressions, but some would argue more directly.
We can invoke an Omegahat expression using the *.OmegahatExpression()*. The first argument is a the Omegahat expression as a string.

```
[]
.OmegahatExpression("System.getProperty(\"java.class.path\")")
```

While the input to evaluating an Omegahat expression is a string, the result is a real object in exactly the same manner as the *.Java()* function returns its value. The usual conversions are performed. The result of the example above is a character vector containing the class path. Other types of objects are converted appropriately.
One of the difficulties with using string expressions to execute commands is that one has to construct the string. This is easy in interactive use, but for programmatically generated commands can be trickier. There is of course just the difficulty of pasting the appropriate elements together. This is rarely complex, just tedious. The more challenging issue is how we reference local variables in such an expression so that they will make sense in the foreign system – Omegahat. Fortunately, we are familiar with such a mechanism - *substitute()*. One can reference local R variables in an Omegahat expression by listing them in the $\cdots$ argument. One gives each of these arguments a name by which it can be referenced in the expression string. For example,

```
[]
  .OmegahatExpression("new DataFrame(data)", data=mtcars)
```

This call creates the association between the Omegahat variable `data` used in the string expression and the R variable *mtcars()*. When evaluating the Omegahat expression, the evaluator will attempt to locate the value of the variable `data`. This will be available in a special frame or database that is created for the duration of the expression evaluation. The contents of this Omegahat database consist of the objects passed via the $\cdots$ argument of *.OmegahatExpression()*. The names are those used in that function call for the corresponding argument. In other words, in a call such as

```
[]
 dyn <- .Java("__Evaluator", "dynamicClassLoader")
 .OmegahatExpression("x = new Object[]{a, b, c}", a= 1:10, b=mtcars, c=dyn)
```

Omegahat creates a new frame with entries `a`, `b` and `c`. These contain the Java objects created by converting the R objects.
Unfortunately, the example above will not work as is. We somehow have to tell the R-Omegahat bridge how to convert the R object `mtcars` to a Java object. We can use the dynamic compilation mechanism above, but we must instruct it as to what type of interface we wish to implement in a new car. We use the *.sigs* for this purpose. This not only instructs how to convert the object, but also which constructor the Omegahat expression is actually invoking. The following illustrates the basic idea.

```
[]
  .OmegahatExpression("new DataFrame(data)", data=mtcars, .sigs="DataFrameInt")
```

12.1 Debugging

There is not much explicit support for debugging Java calls from R and the R function callbacks. The architecture does provide many facilities that one can use to monitor certain computations. This will be outlined and enhanced in future releases.

13 C-level Programming Access

This R-Java interface is implemented using the Java Native Interface (JNI) that is part of the Java programming environment. This is more frequently used to use code from Java via native methods. We use it in this way to implement the callbacks to R functions in *RForeignReference* (one of its `eval()` method). However, we use the JNI primarily in the other direction – to access Java from C. Since the Java VM is embedded and initialized as part of this interface, R programmers can integrate other *C*-level code in either way – as part of Java or part of R – that uses the JVM.

13.1 Native Java Methods

Using Java classes with native methods is no different than using them within a regular stand-alone Java application. The user must call `System.loadLibrary()` to load the shared library containing the C routine(s) (even if R has already loaded that same library). These C routines are passed a reference to the Java environment (similar to thread-specific information) and with that can access all of the facilities in Java.

13.2 C routines in R

?? R users can use C routines that also manipulate the Java environment using the *.C()* and *.Call()* functions. Developers may wish to write such routines for a couple of reasons. Firstly, while the *.Java()* and *.JavaConstructor()* functions are very general and provide complete access to the Java facilities, it is interpreted on both sides of the connection: R and Omegahat.

speed Implementing calls to Java methods and constructors in C makes this faster for important data types. This is an issue when converting data types between R and Java as discussed in Section 10.

Existing *C* Code One can use code in other *C* libraries and connect it to the Java facilities directly within *C*. This avoids the indirect approach of writing an interface from R to this other library and then connecting the R-level access to Java.

The *C* code that implements the R-Java interface c ontains both access to the basic internal Java variables needed by any code communicating with Java and also many utility routines that make it easier to implement JNI. These routines are in no way fixed and do not define an API that will necessarily be supported in the future. However, others can use them to implement C code that access the JVM. They are unlikely to change considerably and any changes will typically involve adding extra arguments to routines.
The best source of documentation for the C routines is the code itself. We have written the C-level code using noweb, a literate programming tool that promotes writing code to be read easily by humans (and not specifically in a format that the compute expects.) From this, one can generate output in Postscript, PDF, HTML, etc. You can retrieve the code from the Omegahat CVS repository (`http://www.omegahat.org/Howto/CVSInstall.html`). (You will need (some of) the noweb tools. If you are having difficulties, let us know and we can make them available.)
A table of C routines and macros provided by the library is given in **CRoutines.nw**. Note that the R library is actually linked against an intermediate Omegahat library which provides the basic embedded Omegahat & Java facilities. This library can be used by any C level application, independently of R.

14 Installation Details & Customization

There are several variables and arguments that one can supply to customize the installation of the library. The **README** file in the tar file (**Java˙1.0.tar.gz**) provides the details in short form. In this section, we discuss them in more detail.

14.1 Requirements

Firstly, one requires a Java run-time environment that is at least version 1.2 or 1.3 of Java. We have successfully tested this on Solaris (SunOS 5.6 and 5.7) and Linux (2.2.14-5). For Solaris, we have used the beta JDK1.3 from Sun (`http://www.javasoft.com`). For Linux, we have used IBM's JDK1.3 (`http://www.ibm.com/java/jdk`) and also the JDK1.2 from Blackdown (`http://www.blackdown.org`). The IBM version is preferrable for Linux as we have experienced some some difficulties with running GUIs in the Blackdown version.
Other JVMs such as japhar have not been tested. Kaffe is supports Java 1.1 and so cannot be used without (minor) modifications to the code. (Feel free to make them and let us know.)

14.2 Finding Java

The first thing the installation does is to look for an implementation of the Java runtime environment. This is usually an executable application named **`java`**. The configuration finds the first such entry in the callers path and from this, determines the location of the JDK installation. This is termed `JAVA_HOME` and is used to find the different shared libraries and header files needed to compile and link the R code that implements the interface.
Some people tend to have an old or inappropriate version of **`java`** in their path and use a more modern version. For the configuration to succeed, one can alter one's path and prepend the location of the directory containing the desired **`java`** version. Alternatively, one can define the value of the environment variable `JAVA_HOME` before invoking the **`R INSTALL`** script. The value should be the top-level directory of the JDK installation. This directory will typically contain the directories **include/**, **bin/**, **jre/** among others and is usually the directory in which the JDK was un-tar'ed.
The configuration script tests the version of Java and complains if it is not at least Java 1.2 capable. Note that version 0.52 of the package introduced an option for the configure script that allows one to treat an unknown version as a warning and not a fatal error. This is activated via the `--enable-force` argument and will merely warn about an unexpected JVM version and/or an unexpected vendor. When installing via the R package, the argument can be specified as

```
R INSTALL --configure-args="--enable-force"
```

14.3 Compiling & Linking

The configuration script then attempts to find the location of the necessary JNI header files and shared libraries. It does this by examining the creator or vendor of the JVM being used and the operating system on which we are located. The directories containing the include files and libraries are computed via the Java class *jniBashParamters*. The vendor of the JVM and from this the names of the actual JVM libraries against which we link are computed via the Java class *vendor*.
One can enable the debugging support in the C code that implements the R-Java interface by specifying the `--enable-debug` argument to the configure script. When used with **`R INSTALL`**, this is done via the command

```
[]
 R INSTALL --configure-args="--enable-debug" -c Java_1.0.tar.gz
```

One really does not want to do this. It generates an enormous amount of output. We will at some point implement a debug level, so that one can control what types of messages are output based on an integer value of *degree*.
The remainder of the configuration script computes derived flags and variables from the information previously computed earlier. Along with creating the necessary makefiles for the sub-library (**libRSNativeJava.so** in the **src/RSJava/**), it also configures the R code to get the
The conclusion of the configuration file compiles the *libRSNativeJava.so* and installs it in the **inst/libs//** directory. Additionally, it creates the JNI header files from the two Java classes *RForeignReference* and *RManualFunctionActionListener*.
The configuration script also produces a cleanup script that will be run if **`R INSTALL`** is invoked with the `-c` flag. This creates a symbolic link from the shared library *Java.so* that R will attempt to load in the call `library(Java)` and the library *libJava.so* that Java will attempt to load when the class *RForeignReference* is needed.

If one omits the `-c` flag when calling **`R INSTALL`**, one can invoke the **`cleanup`** script from the shell. It is installed in the Java package directory as **scripts/cleanup**. This is shell script.
In addition to the cleanup script, the package installs two other shell scripts in the **scripts/** directory. These are named **RJava.csh** and **RJava.bsh** and are for users of csh/tcsh and Bourne (bash, sh) shells, respectively. These modify the value of the `LD_LIBRARY_PATH` shell variable by appending the directories containing the JDK shared libraries. This is necessary so that the (implicit) call to the R function *dyn.load()* succeeds. The user must source one of these scripts *before* starting the R session. This is done via one of the following commands (Bourne and C shell respectively):

```
[]
  . RJava.bsh
  source RJava.csh
```

In the future, we will add facilities to the configuration script that add these directories to the run-time load path contained with the shared libraries we build. It currently does try, but cannot handle the secondary dependencies.