Chapter 15

An Interface to Julia

15.1 R and Julia

This chapter describes an interface from R to computations in the Julia language, implemented in the XRJulia package and following the XR structure described in Chapter 13. The interface is described as it would be used in an application project, via an application package that incorporates computational techniques integrated into R but using functions and/or data types implemented in Julia. The XRJulia package, the XR package it imports and the juliaExamples package are available from `github.com/johnmchambers`.

Julia is described as a "high-level, high-performance dynamic programming language for technical computing".[1] Its intended applications focus on computational methods for numerical, scientific and similar applications. Its design combines high-level programming structures with efficient code, compiled on-the-fly from Julia language source code.

The language and user environment are quite similar to R in many respects. Interacting with Julia, one types in expressions; the system computes results and prints output back. Function definitions and calls are very much the heart of programming with Julia. Many of the base functions and operators closely resemble those in R.

A particularly strong similarity, not shared with many other languages, is that Julia, like R, implements functional OOP: generic functions with methods selected according to the classes of one or more of the arguments in the call. Classes are called "types" in Julia and the system for type definition works differently; in particular, the use of macro-like templates for type and method definition is a key

[1] `http://julialang.org`

feature. But type definitions are objects, including the specification of properties, as in R.

The interface provides direct analogues to Julia function calls and other computations through methods for evaluator objects from the `"JuliaInterface"` class or through equivalent R function calls (Sections 15.2 and 15.3).

An application using the interface can define proxy functions in R that call corresponding functions in Julia, including the functional methods defined for these functions (Section 15.4). Proxy classes in R can be defined corresponding to types in Julia, with access to fields consistent with reference classes in R (Section 15.5). Julia does not have encapsulated methods.

Julia emphasizes a form of functional *computing*, suggestive of the FUNCTION principle, but the design is not related to functional *programming*, in the sense of protecting against side effects. Arguments are passed as references and in this sense Julia types are more analogous to reference classes than functional OOP classes in R.

In some sizable collections of functions (for example, some graphics applications) function values are largely irrelevant, with the side-effects of the function call on external objects being the main point.

Nevertheless, for "programming in the small", programming with Julia is largely based on defining functions. These are easy to define in the language and are immediately available for use:

```
function myMean(x)
  sum(x)/length(x)
end
```

This defines the function and assigns it as `"myMean"`.

Functions are generic by default. Julia has optional typing; by not declaring the type of the argument to `myMean()`, we essentially define a default method. Definitions of functions with the same function name but with explicit type declarations are the equivalent of method definitions. Argument names are arbitrary in methods; there are no formal arguments.

For medium-scale programming, Julia has packages (collections of source code) and within a package *modules*, which are declarations surrounding a collection of source code. As in other languages, modules can be imported by various mechanisms and used to define the namespace for new applications.

Applications using the interface are likely to define some Julia functions and types in module(s) associated with the package. These may usefully define methods for existing functions, simply by declaring the arguments to correspond to types that will also often be defined in the application's Julia code. The interface includes facilities for importing packages and modules from Julia (Section 15.3).

Another strong similarity between R and Julia is in their treatment of R's basic vectors, matrices and arrays. As in R, Julia has taken over the essential organization of these data structures originating in Fortran. The XRJulia interface converts data with such structure into the corresponding class in the other language. In addition, there is a general data conversion mechanism following the XR structure that supports conversion of arbitrary classes in either language (Section 15.6).

15.2 Julia Computations

Computations in Julia using the XRJulia interface are carried out by a Julia interface evaluator, an object from the `"JuliaInterface"` class. The current evaluator from this class is returned by the function `RJulia()`:

```
ev <- RJulia()
```

If no evaluator exists, one is started.

The XRJulia interface uses a connection, via a socket, to a process running Julia. By default, this will be a process on the machine running R. The Julia process is started when the evaluator is initialized, and given a startup script that tells it to accept and execute commands written on the socket by the R process.

At the time this book appears, the XRJulia interface is new, and all the examples shown here used the default configuration. The design, however, anticipates a more general use of sockets, as illustrated by the parallel package, for example. The evaluator object would be initialized to communicate with an existing socket connection to a Julia process using a similar startup script.

```
ev2 <- RJulia(connection = jCon)
```

In this case, `jCon` will be an open socket connection object; for example, to a Julia process initialized on a remote host for an R interface.

The choice of a connected rather than embedded interface was partly to illustrate this approach, given that the Python interface in Chapter 14 was embedded, but there are other advantages.

While embedded interfaces tend to be more efficient, at least in communicating between the languages, connected interfaces free the server computations from constraints on the design due to running the server language within the R process.

Since a connected interface is communicating with an independent process, there should be no constraints on the Julia computations because of the interface. Connected interfaces also raise the possibility of distributing computations across machines; for example, using a more powerful machine that you have to pay for when capacity is needed but a local process for less demanding computations.

General expressions and commands can be evaluated by the methods:

```
ev$Eval(expr, ...)
ev$Command(expr, ...)
```

In Julia, in contrast to R, not all statements can be evaluated as expressions; these statements usually have side-effects but no value, and will throw an error if called through `$Eval()`. For these the appropriate method is `$Command()`, which has the same arguments as `$Eval()` and evaluates the Julia string but makes no attempt to treat the result as an expression:

```
ev$Command("rtpi = sqrt(pi)")
```

Any piece of code that is complete and valid in Julia should be executable via `$Command()`.

The `$Eval()` and `$Command()` methods and all other methods in this section have functional equivalents `juliaEval()`, `juliaCommand()`, etc. These have the same arguments as the methods, plus an argument `evaluator=`, by default and usually the current Julia evaluator, which will be started if none exists.

For computations where no special evaluator is needed, the functional forms may be more natural looking in R and avoid explicit reference to the evaluator object. They do nothing but call the corresponding method.

In these methods, `expr` is a character string to be parsed and evaluated by the Julia evaluator. Additional arguments are objects that will be inserted into the expressions corresponding to C-style `"%s"` fields in the string. These may be results previously computed through the interface and returned as proxies for the Julia object or R objects, which will be substituted as a string that evaluates to the Julia equivalent of the R object:

```
> y <- juliaEval("reverse(%s)", 1:5)
> y
Julia proxy object
Server Class: Array{Int64,1}; size: 5
> juliaEval("pop!(%s)",y)
[1] 1
```

Scalar results are usually converted back to R values; more extensive or structured results are assigned in Julia and returned as proxy objects.

Objects can be explicitly sent to Julia and got back by the `$Send()` and `$Get()` methods and their functional equivalents. In both directions the computations rely on some conversions between objects in the two languages, as we'll consider in more detail in Section 15.6.

```
> juliaGet(y)
[1] 5 4 3 2
> x <- matrix(rnorm(1000),20,5)
> xm <- juliaSend(x)
> xm
Julia proxy object
Server Class: Array{Float64,2}; size: 100
> xjm <- juliaGet(xm)
> all.equal(x, xjm)
[1] TRUE
```

Julia has a full set of typed arrays, differing in details from R but very naturally mapped to R arrays. Essentially no information is lost in transferring numerical matrices, as in this example.

Julia operates with its own version of the $\boxed{\text{FUNCTION}}$ principle; most interesting computations are done by functions. Defining new functions and/or new functional methods is the central step in programming-in-the-small, just as in R. Care is needed because this is functional *computing* rather than functional programming: functions frequently have side effects. In the examples above, for instance, the call to `pop!()` altered the object `y`.

For convenience, a function call has a short-cut for `$Eval()` that avoids messing with format strings. The first argument is the character string name of the Julia function, the remainder the arguments to the call. The expression to call the function `pop!()` above could have been written:

```
juliaCall("pop!",y)
```

`juliaCall()` or the `$Call()` method may be useful if the function name is computed rather than a constant or if the function is only called in one instance. Otherwise it's usually more convenient to define proxy functions in R, as discussed in Section 15.4.

As an alternative to a separate call to `$Get()`, the `$Eval()` and `$Call()` methods have an optional argument, `.get=`, that can be used to force conversion of an arbitrary result by supplying it as `.get = TRUE`.

```
> xt <- juliaCall("transpose",xm, .get=TRUE)
> dim(xt)
[1]  5 20
```

The proxy functions in Section 15.4 also have a `.get` argument with the same interpretation.

15.3 Julia Programming

As with R packages and Python modules, the names to refer to functions and other objects in Julia are organized by *modules*. In all three languages, the result is the ability to refer to objects by a fully qualified form, `package::name` in R and `module.name` in the other languages.

All three languages take slightly different routes to the organization of source when the evaluator is searching for a module or equivalent. In R and Python, the package/module is defined by the directory and file structure of the package. In R a package is a directory structured as we discussed in Section 7.1. In Python a module is a single file of source. Julia has both forms, in its own style.

Julia, like R, has the notion of a package within which directories and files are organized according to a particular structure. However, the Julia evaluator will also recognize separate files of source code through the file suffix `".jl"`. Either way, the directory or file must contain a matching `module` declaration in a particular form in the source code.

The XRJulia package, for example, has a file and module containing the various functions and data types used through the interface. This could have any name; for simplicity we use the package name and put the code in the file

```
inst/julia/XRJulia.jl
```

in the package source. This file declares the module of the same name:

```
module XRJulia
... # all the Julia source code
end
```

The evaluator will look for packages or files in one of a list of declared directory locations, analogous to `.libPaths()` in R. An application package that contains any Julia modules of its own will need to make these available by calling the function:

```
juliaAddToPath(directory, package)
```

In general, this will add any named directory to the search path, from the specified R package, or a directory unrelated to any R package if `package=""`. The `directory` is interpreted relative to the installation directory of the package. A package can refer to its own installation directory by omitting the `package` argument. If the package follows the XR convention of putting the files of Julia code into a directory `"inst/julia"` in the package source, that directory can be added to the search list by the empty call:

```
juliaAddToPath()
```

Importing modules not associated with an R package may raise difficulties for portability; see Section 13.4, page 270 for some comments.

Once a module is accessible by being on the search path, it must be imported to make its objects available by reference. As in R, there are some base objects always available, including the standard library. Objects from other modules need to be made available by importing; unlike R, the fully qualified reference will not load the module automatically.

The interface function

```
juliaImport(module, ...)
```

generates suitable `"import"` commands in Julia. Fully qualified imports are provided by calling `juliaImport()` with only the module name. To use the name in an unqualified form, supply it explicitly as a separate argument. If we wanted to use the `undigit()` function in Julia module Digits:

```
juliaImport("Digits") # Julia calls to Digits.undigit()
juliaImport("Digits", "undigit") # Julia calls to undigit()
```

Julia actually has two directives, `using` and `import`, that behave somewhat differently. The main difference is that `using` makes all the exported names available in unqualified form. The XRJulia package supports the equivalent range of options using the argument list to `juliaImport()` See the online documentation for details.

The functions and methods in the previous section were essentially equivalent, but for the path and import operations in this section the functional version is preferred. A call to `juliaAddToPath()` from the source code of a package adds the directory to a table of the path lists for all interfaces. Similarly, the `juliaImport()` function adds to a table of import directives. The functional form will add the path and module information to the current `"JuliaInterface"` evaluator and to all future evaluator objects.

For application packages, the functional form is preferred, except in the unusual case where one evaluator object needs its own path or imports. Then the method would be used for that evaluator explicitly.

Modules and source files are distinct concepts in Julia, even though a module can correspond to a single source file. The function `juliaSource()`, or the interface method `$Source()`, parses and evaluates the code in a specified file, using the Julia function `include()`.

The Julia commands `require` and `reload` also evaluate the contents of a specified file, but they put their results into the main module, meaning that assignments in the file will not be visible from interface expressions: `juliaSource()`

works through the evaluator so that all results are stored in the same module as other interface computations.

Julia, like R, returns the value of the last expression computed in the file. The following little source file defines a Julia type and returns an example object from it by calling the generator function:

```
type testT
    x::Array{Int64,1}
    y::ASCIIString
end

testT([1,-99,666],  "test1")
```

Assuming that this is file "testT.jl" in the current working directory of the R process, we can use it to compute a proxy for the object returned by testT():

```
> xt <- juliaSource("testT.jl")
> xt
Julia proxy object
Server Class: testT; size: NA
> juliaGet(xt)
R conversion of Julia object of composite type "testT"

Julia fields:
$x
[1]   1 -99 666

$y
[1] "test1"
```

The field names for an object of type "testT" are used in evaluating the juliaGet() call to create an Robject of class "from_Julia". Data conversion techniques will be discussed in Section 15.6.

In developing software for the interface, it may be convenient to have Julia print some result, rather than having to convert that to an R object and bring it back. For this purpose, you can use the juliaPrint() function. It can take one argument, typically a proxy object. Or, you can give it several arguments that will be interpreted as if given to the $Eval() method and will print the result of that computation. See page 337 for an example.

15.4 Julia Functions

A proxy function in R to call a Julia function of a specific name is returned by

```
JuliaFunction(name, module)
```

with the argument `module` only required to ensure that the corresponding module is imported, if it is not by default. As an example, the Julia function `svdfact()` computes a singular value factorization of a matrix. An R proxy function for it could be created by:

```
svdJ <- JuliaFunction("svdfact")
```

Calls to `svdJ()` will generate calls through the XRJulia interface to `svdfact()`. The arguments to `svdJ()` will be converted as needed from the R objects; more likely, except for simple scalars, they are R proxies for Julia objects previously computed or converted. By default, the evaluator used is the current Julia evaluator, which will be started if necessary; an optional argument allows a different evaluator to be specified. The `svdfact()` function is part of the standard library, so no explicit module import is needed.

We can construct the decomposition of the Julia array `xm` shown on page 325:

```
> sxm <- svdJ(xm)
> sxm
Julia proxy object
Server Class: SVD{Float64,Float64}; size: NA
```

The composite Julia type for the result has essentially the same information as the result of the R function `svd()`. In Section 15.5, proxy R classes for the type will be shown. With or without a proxy class, the interface evaluator can get the information in the decomposition back to R, as we will show in Section 15.6.

Julia functions are generic by default; that is, a function definition actually creates a method associated with that function name. Optional type declarations for the arguments specify the signature for the method. Additional function definitions for the same name, with different type declarations for the arguments, will define additional methods for the generic.

As with R, the classes (types) of the actual arguments in a call will be used to select the best method for that call. An important difference, however, is that Julia uses the selection to compile the appropriate method for this case.

This distinction has some implications for an interface from R. There is no formal argument list for the function, and indeed no pre-determined number of arguments. Different methods may have different argument names or number of arguments.

Julia methods are best seen as prescriptions for creating (by compilation) an actual executable method. Method "dispatch" examines the type declarations for existing methods to find a match to the types of the actual arguments. The `svdfact()` function, for example, is implemented for (currently) 9 signatures corresponding to the function declarations:

```
svdfact(D::Diagonal, thin=true)
svdfact(M::Bidiagonal, thin::Bool=true)
svdfact{T<:BlasFloat}(A::StridedMatrix{T};thin=true)
svdfact{T}(A::StridedVecOrMat{T};thin=true)
svdfact(x::Number; thin::Bool=true)
svdfact(x::Integer; thin::Bool=true)
svdfact{T<:BlasFloat}(A::StridedMatrix{T}, B::StridedMatrix{T})
svdfact{TA,TB}(A::StridedMatrix{TA}, B::StridedMatrix{TB})
svdfact(A::Triangular)
```

Many of these are templates; that is, specific argument types will match the signature for some macro-style substitution of the template argument, such as `T`, `TA`, `TB` in the methods above. It is part of the central Julia design that this provides a flexible, dynamic method selection system. It would not be straightforward, however, to check on the R side that the arguments to the proxy are consistent with the available methods.

Since argument names are not restricted by the generic function, as the example shows, function calls in Julia cannot refer to arguments by name. Julia does provide a mechanism for "keyword" arguments. These are defined by a special syntax in the formal argument list for a particular method; in effect, they match elements in a dictionary to keyword arguments in the call. But ordinary arguments are accepted only positionally.

Considering these characteristics, the present version of the XRJulia interface leaves argument checking up to the server language side of the interface. Proxy functions in R for Julia functions pass the actual arguments on unmodified. The number and order of actual arguments should be what is intended for the Julia call and named arguments will be passed on with the same names. Note that named (aka keyword) arguments must follow positional arguments in the call.

If the `module` argument is specified in the call to `JuliaFunction()`, the named function is assumed to be exported from that Julia module. The body of the proxy function will include an import call for the module; because the XRJulia evaluator keeps a table of imported modules, only one actual import command will be issued to Julia. The actual Julia function call uses a fully qualified name; therefore, proxy functions can interface to two functions of the same name in distinct modules.

15.5 Julia Types

Julia provides for definitions of what are called "composite types" and are in effect classes with specified fields. Since Julia supports a form of functional OOP, these are used more as functional classes in R. They appear in Julia as type declarations in method definitions, analogous to the signatures for R methods. They do not have encapsulated methods, in contrast to classes in Python or Java.

Unlike functional class objects in R, Julia objects use reference semantics; when you change a field in a Julia object the change is not local to the function call where it takes place.

A call to `setJuliaClass()` creates a proxy class in R for a type in Julia:

```
setJuliaClass(juliaType, module)
```

The arguments are the type name in Julia and the module name, which can be omitted for classes in the base software. Metadata in Julia defines the fields, which will be accessible as reference class fields in R, using the `` `$` `` operator.

Many relevant Julia types are *parametrized*, in that their definition contains one or more template- or macro-style arguments. In the example on page 329, the result returned was a proxy for an object of type `"SVD{Float64,Float64}"`. The `"SVD"` type is parametrized by (at least) two numeric types, for the input data and the output values.

When the type is parametrized, either the specific version or the whole family may have a proxy class defined in R. The field names of the class are generally defined by the family, with only the field types affected by the specific type; however, it may be undesirable in R to use the same proxy class for all the specific types. When a proxy object is returned from Julia, the XRJulia interface looks first for a proxy class to the parametrized type and then for the unparametrized version. The application package can choose which version to set up, or both. In the example:

```
setJuliaClass("SVD")
```

would create a proxy class for any member of the family; all have the same fields, `"U"`, `"S"` and `"Vt"`. If this proxy class had been defined before setting up the proxy function `svdJ()`:

```
> sxm <- svdJ(xm)
> sxm
R Object of class "SVD_Julia", for Julia proxy object
Server Class: SVD{Float64,Float64}; size: NA
> sxm$S
```

```
Julia proxy object
Server Class: Array{Float64,1}; size: 5
```

Julia types have the additional option of being `"immutable"`; effectively, this means that all the fields are read-only in the sense discussed in Chapter 11. Their fields may be accessed but not assigned. Having the relevant fields read-only may be a useful way to avoid accidental invalidation of the object when fields must have a fixed relationship. Such is definitely the case with `"SVD"` and other matrix factorizations; manipulating values in any of the fields will usually invalidate the object as a correct factorization. And in fact the `"SVD"` type is declared `immutable`.

If XRJulia detects an immutable type, it makes the proxy fields read-only.

```
> sxm$S <- 0
Error: Server field "S" of server class "SVD{Float64,Float64}"
             is read-only
```

15.6 Data Conversion

Data conversion in XRJulia is based on that described in Section 13.8 for XR, including facilities for representing general objects from R and from Julia, but provides additional features that may be particularly relevant for numeric and other algorithmic interface applications.

R and Julia have a number of similarities in the representation of important classes of data, particularly those corresponding to vectors, matrices and arrays in R. There is also a natural relation between classes in R and composite types in Julia. Data conversion in XRJulia uses these characteristics for a cleaner and more direct matching between the languages than provided by the default strategy. You can usually assume that objects map automatically in both directions if they come from classes that are vectors or arrays of any of the types known in R or that consist of slots/fields that can themselves be mapped.

Applications can customize the conversion when that is helpful. Conversion to Julia implements methods directly for `asServerObject()`, omitting the JSON intermediate form. Conversion to R (page 335) uses methods for the generic functions `toR()` in Julia and `asRObject()` in R. We'll note limitations for converting some Julia types. Applications can often work directly from the general representation forms for the corresponding classes in the other language. An example using the R `"data.frame"` class is on page 337; a small example with a Julia class is in Section 15.3, page 328.

Vectors and Arrays in R and Julia

The two languages share an approach to arrays. In both languages, arrays are defined by a block of elements of a particular type; in other words, a "vector" in R terminology. This vector is interpreted as a k-way array by associating with it k integers for the range of indices in each dimension. In R the concept is implemented as the "array" class with slots for data and dimensions. Julia has a parametrized set of types, without explicit fields for data and dimensions but with a paradigm for programming that supports essentially the same range of objects.

In addition to the matching of array structure between the languages, R and Julia support a variety of basic data types for arrays, as opposed to the JSON notation, which only supports lists of arbitrary elements. Conversion between corresponding classes is automatic in both directions provided the basic Julia type corresponds to one of the R vector types.

Julia has a set of parametrized `Array` types

```
Array{T,N}
```

where `T` is a Julia type corresponding to the type of the elements and `N` is the number of dimensions.

Vectors in R map into one of the `Array{T,1}` types, with `T` determined by the R type of the vector. The type parameter `T` has a variety of options, considerably more than the range of basic types in R. Integer, floating point and bit-string types have options for length; R maps "integer", "numeric" and "raw" into particular choices that reflect the R implementation. Julia type "Any" corresponds to type "list". Sending vectors of various types from R will create suitable Julia array objects:[2]

```
> ev$Send(1:3)
Julia proxy object
Server Class: Array{Int64,1}; size: 3
> ev$Send(c(1,2,3))
Julia proxy object
Server Class: Array{Float64,1}; size: 3
> ev$Send(c("red","white","blue"))
Julia proxy object
Server Class: Array{ASCIIString,1}; size: 3
> ev$Send(list("Today", 1:2, FALSE))
Julia proxy object
Server Class: Array{Any,1}; size: 3
```

[2]In this section, we are looking at implementation details and will revert to showing the method version of `$Send()`, etc., rather than the equivalent functions.

The Julia server language expression is the list of elements, written out explicitly:

```
> ev$AsServerObject(1:3)
[1] "[1,2,3]"
> ev$AsServerObject(c(1,2,3))
[1] "[1.0,2.0,3.0]"
> ev$AsServerObject(c("red", "white", "blue"))
[1] "[\"red\",\"white\",\"blue\"]"
> ev$AsServerObject(list("Today", 1:2, FALSE))
[1] "{ \"Today\", [1,2], false }"
```

Julia interprets the list as an array of the type needed, similar to the `c()` function in R, except that elements of length `> 1` effectively force a list-style object.

Complex is not a basic type in Julia but essentially a parametrized type for representing pairs of values. The `"complex"` vector in R corresponds to one of those, for pairs of floating point numbers.

```
> cx
[1] 7.8+3.8i 5.5+3.4i 5.2+3.1i 6.8+0.1i
> cxj <- ev$Send(cx)
> cxj
Julia proxy object
Server Class: Array{Complex{Float64},1}; size: 4
> ev$Get(cxj)
[1] 7.8+3.8i 5.5+3.4i 5.2+3.1i 6.8+0.1i
```

Complex vectors in R are sent by a call to the generator function for the Julia type:

```
> ev$AsServerObject(cx)
[1] "complex([7.8,5.5,5.2,6.8], [3.8,3.4,3.1,0.1])"
```

The `Complex` types in Julia have a generator with two vectors for the real and imaginary parts as arguments.

An R array object will also map to one of the Julia parametrized array types. For example, the `iris3` object in the datasets package is a three-way array:

```
> dim(iris3)
[1] 50  4  3
> typeof(iris3)
[1] "double"
```

Sending this object to Julia produces a corresponding Julia array object:

```
> irisJ <- ev$Send(iris3)
> irisJ
Julia proxy object
Server Class: Array{Float64,3}; size: 600
```

A general array object in R is sent to Julia by first creating the one-way array with the data part and then using the Julia function `reshape()` to specify the dimensions:

```
> xm <- matrix(1:6,3,2)
> ev$AsServerObject(xm)
[1] "reshape([1,2,3,4,5,6], 3,2)"
> ev$Send(xm)
Julia proxy object
Server Class: Array{Int64,2}; size: 6
```

Converting Julia objects

The conversion of Julia objects to R retains JSON notation in the string returned by the Julia evaluator to R. Where the Julia type has a matching R class, the JSON form uses the representation of a general R object by a specialized dictionary containing an element named `".RClass"`. The conversion produces an object of the corresponding R class. R methods for `asRObject()` may further specialize conversion of this object.

Two special R classes are particularly important: `"vector_R"` and `"from_Julia"`. The first of these explicitly represents various types of vectors in R, which would otherwise be ambiguous if written as just a JSON list. The second explicitly identifies a Julia object from a composite type, converted with a named list of the (converted) data in each of its slots. This representation is not dependent on the existence of a proxy class in R for the Julia type.

The Julia side of the interface consists of a collection of methods for the function `toR()`. Its argument is an arbitrary Julia object and it returns another object such that the JSON representation produces an R object matching the original Julia object.

Objects from the parametrized `"Array{T,N}"` types are returned as R vectors or arrays. The returned object will be a vector if `N` is `1` and an array otherwise. The type of the R vector will be numeric, integer, logical or character for `T` a corresponding Julia scalar type. Type `Any` will be returned as a list. Returned arrays are constructed by reshaping the array into a one-way array and converting this for the `".Data"` slot; therefore, the same type matching applies as for vectors.

Dictionaries will be returned as named lists of their elements. While Julia dictionaries are parametrized by the type for the keys and the type for the elements, named lists imply character string keys. JSON dictionaries also require strings as keys, so the necessary coercion has already taken place to produce the JSON string.

Scalars of the types recognized by JSON will turn into vectors of length 1 of the corresponding R vector class.

Putting all this together, the convertible Julia objects include all:

1. Scalars, arrays and dictionaries; and
2. Composite types

provided that the elements of the arrays and dictionaries and the fields of the composite types are themselves convertible objects. Any such object will be converted by the `$Get()` method or by a proxy function with `.get = TRUE`, to an R object of the simple forms described above.

Application packages may want to specialize the object returned, to generate a particular R class or perform some transformation of the fields in the Julia object. The most natural approach is to write one or more functional methods in Julia for the function `toR()`. In the Julia code for XRJulia, `toR()` takes as its argument the Julia object that results from evaluating an expression or command. The value returned by `toR()` should be the Julia object that will be converted to JSON representation and sent to R.

Two-way array objects, for example, will be turned into a dictionary with an explicit R representation for class `"matrix"`. The Julia function `RObject()` produces the explicit representation; the method for array objects will finish by calling `toR()` again with the value returned by `RObject()`.

Application methods are likely to do something similar, transforming the Julia type into a chosen class of R objects. Take a look at methods for `toR()` in the XRJulia package, in file `"julia/XRJulia.jl"`.

Occasionally, the application may need to do some further computations on the particular class of R objects returned in this way, by defining a method in R for the function `asRObject()`. In both R and Julia you need to import the generic function `asRObject()` or `toR()` into the application package in order to define methods for it.

Some current limitations on conversions are due to basic types that do not correspond between the languages. Julia has a range of parametrized scalar types that have no direct R equivalent; it's unclear how important these may be for data-based applications, but some extensions to the XRJulia facilities may address these types in the future.

Other types in Julia are useful as programming steps but have a transient form that doesn't survive current conversion computations. In the example on the next page, we used `juliaPrint()` to print an object returned by the `keys()` function, rather than converting the result to R. The object returned by `keys()` is an iterator over a dictionary, a useful type in Julia. But as a composite type for conversion, it contains the entire object over which the iteration takes place, not the keys as strings.

General R class representation: an example

The representation of a general R object as a dictionary with special keys allows computations for a class that does *not* have an obvious Julia counterpart. For an example, let's look once more at `"data.frame"`. As we discussed in Section 10.5, whether formally defined or not, `"data.frame"` effectively extends `"list"`, with slots `"names"` and `"row.names"`, equivalent to:

```
setClass("data.frame",
     slots = c(names = "character",
           row.names = "data.frameRowLabels"),
     contains = "list")
```

Julia has no type directly corresponding to this: It's essentially a dictionary, constrained by requiring the elements to represent variables with the same number of observations, plus a field for the row names. We could define such a composite type, but currently there is not much that can be done with it. More likely, a data frame sent from R will be the source for derived matrix objects, as it often is in R.

The conversion to Julia therefore uses the dictionary representation for a general R class. Section 13.8, page 289, showed an example in JSON notation, for class `"ts"`. The Julia dictionary form is similar. Let's look at a small sample from the data frame version of the `"iris"` data:

```
> iSample <- iris[sample(150,6),]
> jSample <- juliaSend(iSample)
> jSample
Julia proxy object
Server Class: Dict{Any,Any}; size: 7
> juliaPrint("keys(%s)", jSample)
{".type","names",".Data",".RClass",".extends","row.names",
  ".package"}
```

Elements `".type"`, `".extends"` and `".package"` further describe the object's class. All other elements are the slots of the R object, converted to Julia. The `".Data"` element is a list (type `"Array{Any,1}"`) of the 5 variables in the data frame.

Assuming some Julia computations modified this object or created a similar one, getting it back will create the correct R object. As we can test:

```
> iSampleBack <- juliaGet(jSample)
> all.equal(iSampleBack, iSample)
[1] TRUE
```

It's important that this works because of methods in both R and Julia, but *not* methods for the specific `"data.frame"` class in either case.

In the to-Julia direction, the relevant method is for `asServerObject()`:

```
> selectMethod("asServerObject",
+              c("data.frame", "JuliaObject"))
Method Definition:

function (object, prototype)
{
    attrs <- attributes(object)
    if (is.null(attrs) || identical(names(attrs), "names"))
        .asServerList(object, prototype)
    else .asServerList(XR::objectDictionary(object), prototype)
}
<environment: namespace:XRJulia>

Signatures:
        object       prototype
target  "data.frame" "JuliaObject"
defined "list"       "JuliaObject"
```

The method is inherited from the `"list"` method. If the object was simply a list, with or without names, it would be sent directly as a dictionary or array in Julia. But the method checks for additional attributes which will always be there for a class that extends `"list"`, such as `"data.frame"`. If so, the object converted, from `objectDictionary()`, will turn into a Julia dictionary with an element having the reserved name `".RClass"`, in this case containing `"data.frame"`.

The converted slots will be in elements of the dictionary with the slot names. Julia computations designed for the imported R object could modify these elements or construct new Julia objects with the same structure.

Coming back to R, the object will start as a dictionary in JSON. This turns into a list, with names. The `asRObject()` method for `"list"` checks for `".RClass"` among the names; if found, an object from that class will be constructed.