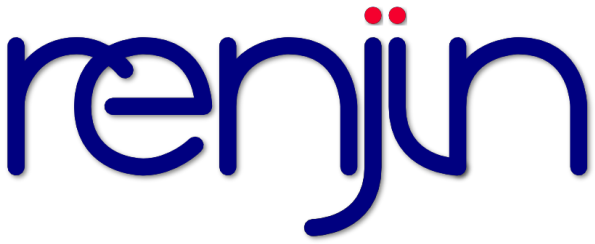

Renjin Documentation

Release 0.8.2356

BeDataDriven

Apr 10, 2017

Contents

1 Introduction **1**
- 1.1 About Renjin . . . 1
- 1.2 Understanding Renjin and package versions . . . 2

2 Using Renjin Interactively **5**
- 2.1 Prerequisites . . . 5
- 2.2 Installation . . . 5
- 2.3 Using Packages . . . 6

3 Using Renjin as a Library **7**
- 3.1 Project Setup . . . 7
- 3.2 Using Packages . . . 10
- 3.3 Evaluating R Language Code . . . 10
- 3.4 Moving Data between Java and R Code . . . 11
- 3.5 Thread-Safety . . . 20
- 3.6 Customizing the Execution Context . . . 21

4 Using Renjin as an R Package **25**
- 4.1 Prerequisites . . . 25
- 4.2 Installation . . . 25
- 4.3 Usage . . . 25

5 Importing Java classes into R code **27**
- 5.1 Bean classes . . . 28

6 Writing Renjin Extensions **29**
- 6.1 Package directory layout . . . 29
- 6.2 Renjin Maven plugin . . . 30
- 6.3 Package NAMESPACE file . . . 32
- 6.4 Using the *hamcrest* package to write unit tests . . . 33

7 Renjin Java API specification **37**
- 7.1 org.renjin.sexp . . . 37
- 7.2 org.renjin.primitives.matrix . . . 41
- 7.3 Exceptions . . . 41

CHAPTER 1

Introduction

This guide covers Renjin version 0.8 and is aimed at developers looking to:

1. integrate R code in their Java applications and to exchange data between Java and R code, and/or to
2. create extension packages that can be used by Renjin much like packages are used to extend GNU R's functionality.

The guide also covers the parts of Renjin's Java API that are most relevant to these goals.

About Renjin

Renjin is an interpreter for the R programming language for statistical computing written in Java much like JRuby[1] and Jython[2] are for the Ruby and Python programming languages. The official R project[3], hereafter referred to as *GNU R*, is the reference implementation for the R language.

The goal of Renjin is to eventually be compatible with GNU R such that most existing R language programs will run in Renjin without the need to make any changes to the code. Needless to say, Renjin is currently not 100% compatible with GNU R so your mileage may vary.

Executing R code from Java or visa versa is not new: there is the rJava[4] package for GNU R that allows R code to call Java methods and there is the RCaller[5] package to call R from Java which is similar to the JRI[6] package (now shipped as part of *rJava*). *JRI* loads the R dynamic library into Java and provides a Java API to the R functionality. *RCaller* works a little different by running R as a separate process that the package communicates with. Finally, *rJava* uses the Java Native Interface (JNI) to enable R to call Java applications running in the JVM.

The biggest advantage of Renjin is that the R interpreter itself is a Java module which can be seamlessly integrated into any Java application. This dispenses with the need to load dynamic libraries or to provide some form of communication between separate processes. These types of interfaces are often the source of much agony[7] because they place very specific demands on the environment in which they run.

[1] http://www.jruby.org
[2] http://www.jython.org
[3] http://www.r-project.org
[4] http://www.rforge.net/rJava/
[5] https://code.google.com/p/rcaller/
[6] http://www.rforge.net/JRI
[7] http://stackoverflow.com/tags/rjava/hot

Renjin also benefits from the Java ecosystem which, amongst many things, includes professional tools for component (or application) life-cycle management. Think of Apache Maven[8] as a software project management tool for building components (i.e. artifacts in Maven parlance) and managing dependencies as well as the likes of Artifactory[9] and Nexus[10] for repository management.

Another advantage of Renjin is that no extra sauce is required to enable the R/Java interface. Packages like *rJava* and *RCaller* require that you litter your code with special commands that are part of their API. As the chapter *Importing Java classes into R code* (page 27) shows, Renjin provides direct access to Java methods using an interface that is completely unobtrusive.

See http://www.renjin.org for more information on Renjin.

Understanding Renjin and package versions

We version two things: Renjin itself and the individual extension packages which we build for Renjin.

Versions and builds of Renjin

The Renjin version number consists of two pieces of information: the major version number and the build number:

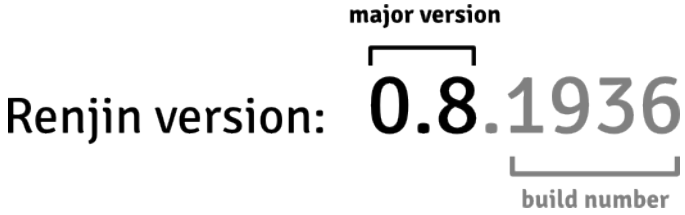

Fig. 1.1: Renjin version numbering

Every time we commit a change to Renjin's source on GitHub[11], a build job is automatically triggered on our build server which assigns the build number to the Renjin version number. If the build succeeds, the artifacts are deployed to our public repository.

The build number in Renjin's version number always increases and is independent of the major version (i.e. it isn't reset to 1 when we increase the major version).

Package versions and builds

R extension packages from CRAN and Bioconductor have their own version numbers which we also use in Renjin. Depending on what changes were committed to Renjin's source, we will manually trigger a build of packages, either all 10000+ of them or a random selection, to assess the effect of the changes on the test results.

Following the explanation in this blog post[12], to fully reference packages in Renjin one would use the following format:

The labels at the top correspond to the fields in a Maven project (POM) file whereas the bottom labels explain how package references are constructed. The package detail page in Renjin's package repository browser tells you how to load extension packages from the command line or using a POM file (see the section sec-using-r-packages-in-renjin).

[8] http://maven.apache.org
[9] http://www.jfrog.com
[10] http://www.sonatype.org/nexus/
[11] https://github.com/bedatadriven/renjin
[12] http://www.renjin.org/blog/2015-09-14-new-packages-renjin-org.html

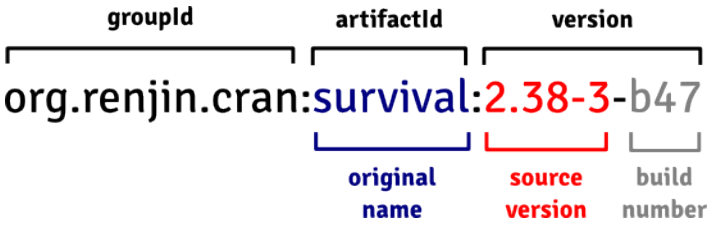

Fig. 1.2: Version numbering of Renjin-compatible extension packages

CHAPTER 2

Using Renjin Interactively

Though Renjin's principle goal is to make it easier to embed R code in existing systems, it can also be used as an interactive Read-Eval-Print-Loop (REPL) similar to that of GNU R.

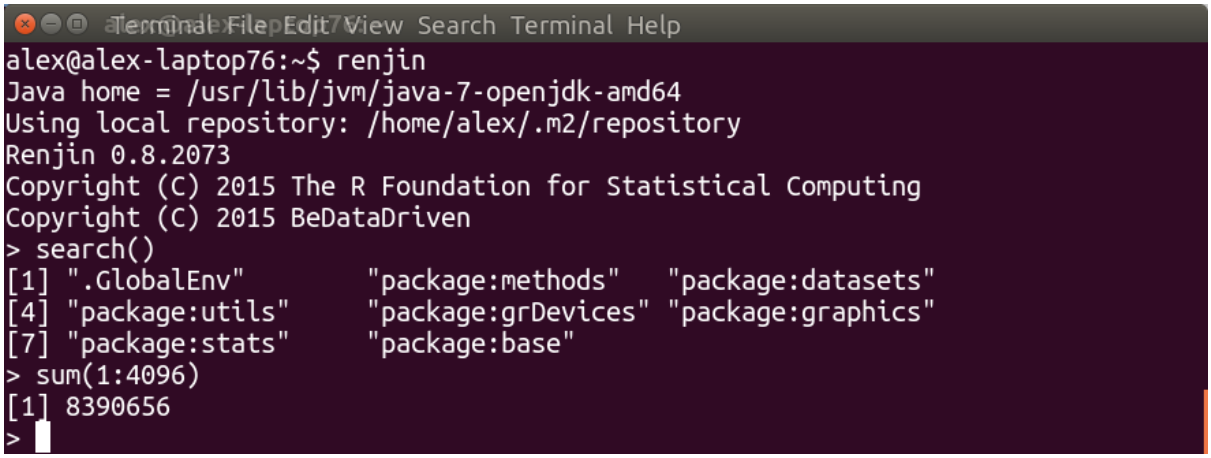

Fig. 2.1: Interactive interpreter run from the command line

Prerequisites

Renjin requires a Java Runtime Environment, version 7 or later. We recommend that you install the latest version of the Oracle's JDK.

Installation

Visit the downloads page[13] on renjin.org.

[13] http://www.renjin.org/downloads.html

Using Packages

There are some differences between the way Renjin manages packages compared to the way that GNU R manages packages.

In GNU R, you must first run `install.packages()`, which will download and build a package from source. After the package is installed, then it can be loaded with a call to `library()`.

From within Renjin's REPL, there is no `install.packages()` function: the first time you try to load a package with `library()`, Renjin will check the repository for a package with the matching name and download it to a local repository located in `~/.m2/repository`.

As a service, BeDataDriven provides a repository with all CRAN (the Comprehensive R Archive Network[14]) and BioConductor packages at http://packages.renjin.org. The packages in this repository are built and packaged for use with Renjin. Not all packages can be built for Renjin so please consult the repository to see if your favorite package is available for Renjin.

[14] https://cran.r-project.org

CHAPTER 3

Using Renjin as a Library

Project Setup

Renjin is a essentially a Java library that allows you to evaluate scripts written in the R language. This library must be added as dependency to your project.

- *Maven* (page 7)
- *Gradle* (page 8)
- *Scala Build Tool (SBT)* (page 8)
- *Eclipse* (page 8)
- *JBoss* (page 9)
- *Spark* (page 9)

Maven

For projects organized with Apache Maven, you can simply add Renjin's Script Engine as dependency to your project:

```
<dependencies>
  <dependency>
    <groupId>org.renjin</groupId>
    <artifactId>renjin-script-engine</artifactId>
    <version>0.8.2356</version>
  </dependency>
</dependencies>
```

For this to work you will also need to add BeDataDriven's public repository to your `pom.xml`:

```
<repositories>
  <repository>
    <id>bedatadriven</id>
    <name>bedatadriven public repo</name>
```

```
    <url>https://nexus.bedatadriven.com/content/groups/public/</url>
  </repository>
</repositories>
```

You can use `RELEASE` instead of `0.8.2356` in the project file to use the very latest versions of the Renjin components.

Gradle

For projects organized with Gradle, add the following to your `build.gradle` file:

```
repositories {
  maven { url "https://nexus.bedatadriven.com/content/groups/public" }
}

dependencies {
  compile "org.renjin:renjin-script-engine:0.8.2356";
}
```

See the renjin-gradle-example[15] on GitHub for a complete example.

Scala Build Tool (SBT)

The following is an example of `build.sbt` that includes Renjin's Script Engine:

```
// IMPORTANT: sbt may fail if http*s* is not used.
resolvers +=
    "BeDataDriven" at "https://nexus.bedatadriven.com/content/groups/public"

lazy val root = (project in file(".")).
  settings(
    name := "renjin-test",
    version := "1.0",
    scalaVersion := "2.10.6",
    libraryDependencies += "org.renjin" % "renjin-script-engine" % "0.8.2356"
  )
```

See the renjin-sbt-example[16] on GitHub for a complete example.

Note: There has been a report[17] that the coursier[18] plugin fails to resolve Renjin's dependencies. If you encounter class path problems with the plugin, try building your project without.

Eclipse

We recommend using a build tool to organize your project. As soon as you begin using non-trivial R packages, it will become increasingly difficult to manage dependencies (and the dependencies of those dependencies) through a point-and-click interface.

If this isn't possible for whatever reason, you can download a single JAR file called:

> renjin-script-engine-0.8.2356-jar-with-dependencies.jar

[15] https://github.com/bedatadriven/renjin-gradle-example
[16] https://github.com/bedatadriven/renjin-gradle-example
[17] http://stackoverflow.com/questions/40888063/load-rdata-from-an-r-script-in-scala-using-renjin#answer-40999169
[18] https://github.com/alexarchambault/coursier

from the Renjin website and manually add this as a dependency in Eclipse.

See the eclipse-dynamic-web-project[19] example project for more details.

JBoss

There have been reports of difficulty loading Renjin within JBoss without a specific `module.xml` file:

```
<module xmlns="urn:jboss:module:1.1" name="org.renjin">
  <resources>
    <resource-root path="renjin-script-engine-0.8.2356-jar-with-dependencies.jar"/>
  </resources>
  <dependencies>
    <module name="javax.api"/>
  </dependencies>
</module>
```

Spark

The *spark-submit* command line tool requires you to explicitly specify the dependencies of your Spark Job. In order to avoid specifying all of Renjin's dependencies, as well as those of CRAN, and BioConductor packages, or your own internal packages, you can still use Maven (or Gradle or SBT) to automatically resolve your dependencies and build a single JAR that you can pass as an argument to *spark-submit* or *dse spark-submit*.

```
<dependencies>
  <dependency>
    <groupId>com.datastax.dse</groupId>
    <artifactId>dse-spark-dependencies</artifactId>
    <version>5.0.1</version>
    <scope>provided</scope>
  </dependency>

  <dependency>
    <groupId>org.renjin</groupId>
    <artifactId>renjin-script-engine</artifactId>
    <version>0.8.2356</version>
  </dependency>

  <dependency>
    <groupId>org.renjin.cran</groupId>
    <artifactId>randomForest</artifactId>
    <version>4.6-12-b34</version>
  </dependency>
</dependencies>

<build>
  <!--- Assembly plugin to build single jar -->
</build>

<repositories>
  <!-- Renjin and Spark/DataStax repositories -->
</repositories>
```

See the renjin-spark-executor[20] project or the datastax/SparkBuildExamples[21] repository from DataStax for complete examples.

[19] https://github.com/bedatadriven/renjin-examples/tree/master/eclipse-dynamic-web-project
[20] https://github.com/onetapbeyond/renjin-spark-executor/tree/master/examples/java/hello-world
[21] https://github.com/datastax/SparkBuildExamples

You can then submit your job as follows:

```
mvn clean package
spark-submit --class org.renjin.ExampleJob target/renjin-example-0.1-dep.jar
```

Using Packages

When using the Renjin Script Engine, R packages are treated almost exactly like any other Java or Scala dependency, and must be placed on your application's classpath by Maven or a similar build tool.

As a service, BeDataDriven provides a repository with all CRAN (the Comprehensive R Archive Network[22]) and Bioconductor[23] packages at http://packages.renjin.org. The packages in this repository are built and packaged for use with Renjin. Not all packages can (yet) be built for Renjin so please consult the repository to see if your favorite package is available for Renjin.

If you use Maven you can include a package to your project by adding it as a dependency. For example, to include the *exptest* package you add the following to your project's `pom.xml` file (don't forget to add BeDataDriven's public repository as described in the section *Project Setup* (page 7)):

```
<dependency>
    <groupId>org.renjin.cran</groupId>
    <artifactId>exptest</artifactId>
    <version>1.2-b214</version>
</dependency>
```

You will find this information on the package detail page as well. For this example this page is at http://packages.renjin.org/packages/exptest.html. Inside your R code you can now simply attach this package to the search path using the `library(exptest)` statement.

Evaluating R Language Code

The best way to call R from Java is to use the javax.scripting[24] interfaces. These interfaces are mature and guaranteed to be stable regardless of how Renjin's internals evolve.

You can create a new instance of a Renjin ScriptEngine using the RenjinScriptEngineFactory class and then instantiate Renjin's ScriptEngine using the factory's `getEngine()` method.

The following code provides a template for a simple Java application that can be used for all the examples in this guide.

```
import javax.script.*;
import org.renjin.script.*;

// ... add additional imports here ...

public class TryRenjin {
  public static void main(String[] args) throws Exception {
    // create a script engine manager:
    RenjinScriptEngineFactory factory = new RenjinScriptEngineFactory();
    // create a Renjin engine:
    ScriptEngine engine = factory.getScriptEngine();

    // ... put your Java code here ...
  }
}
```

[22] http://cran.r-project.org
[23] http://bioconductor.org
[24] http://docs.oracle.com/javase/6/docs/technotes/guides/scripting/programmer_guide/

Note: We recommend using `RenjinScriptEngineFactory` directly, as the standard javax.script silently returns null and hides any exceptions encountered when loading Renjin, making it very difficult to debug any project setup problems.

If you're using Renjin in a more generic context, you can load the engine by name by calling `ScriptEngineManager.getEngineByName("Renjin")`.

With the ScriptEngine instance in hand, you can now evaluate R language source code, either from a String, or from a Reader interface. The following snippet, for example, constructs a data frame, prints it out, and then does a linear regression on the two values.

```
engine.eval("df <- data.frame(x=1:10, y=(1:10)+rnorm(n=10))");
engine.eval("print(df)");
engine.eval("print(lm(y ~ x, df))");
```

You should get output similar to the following:

```
   x      y
 1  1   -0.188
 2  2    3.144
 3  3    1.625
 4  4    3.426
 5  5     6.45
 6  6     5.85
 7  7    7.774
 8  8    8.495
 9  9    9.276
10 10   10.603

Call:
lm(formula = y ~ x, data = df)

Coefficients:
(Intercept) x
-0.582       1.132
```

Note: The ScriptEngine won't print everything to standard out like the interactive REPL does, so if you want to output something, you'll need to call the R `print()` command explicitly.

You can also collect the R commands in a separate file

```
# script.R
df <- data.frame(x=1:10, y=(1:10)+rnorm(n=10))
print(df)
print(lm(y ~ x, df))
```

and evaluate the script using the following snippet:

```
engine.eval(new java.io.FileReader("script.R"));
```

Moving Data between Java and R Code

If you read the *Evaluating R Language Code* (page 10) to this guide you already know how to execute R code from a Java application. In this chapter we will take things a little further and explain how you can move data between Java and R code.

Renjin provides a mapping from R language types to Java objects. To use this mapping effectively you should have at least a basic understanding of R's object types. The next section provides a short introduction which is essentially a condensed version of the relevant material in the R Language Definition manual[25]. If you are already familiar with R's object types you can skip this section and head straight to the section *Pulling data from R into Java* (page 15) or *Pushing data from Java to R* (page 19).

A Java Developer's Guide to R Objects

R has a number of objects types that are referred to as *basic types*. Of these, we only discuss those that are most frequently encountered by users of R: vectors, lists, functions, and the `NULL` object. We also discuss the two common compound objects in R, namely data frames and factors.

Attributes

Before we discuss these objects, it is important to know that all objects except the `NULL` object can have one or more attributes. Common attributes are the `names` attribute which contains the element names, the `class` attribute which stores the name of the class of the object, and the `dim` attribute and (optionally) its `dimnames` companion to store the size of each dimension (and the name of each dimension) of the object. For each object, the `attributes()` command will return a list with the attributes and their values. The value of a specific attribute can be obtained using the `attr()` function. For example, `attr(x, "class")` will return the name of the class of the object (or `NULL` if the attribute is not defined).

Vectors

There are six basic vector types which are referred to as the *atomic vector types*. These are:

logical: a boolean value (for example: `TRUE`)

integer: an integer value (for example: `1`)

double: a real number (for example: `1.5`)

character: a character string (for example: `"foobar"`)

complex: a complex number (for example: `1+2i`)

raw: uninterpreted bytes (forget about this one)

These vectors have a length and can be indexed using `[` as the following sample R session demonstrates:

```
> x <- 2
> length(x)
[1] 1
> y <- c(2, 3)
> y[2]
[1] 3
```

As you can see, even single numbers are vectors with length equal to one. Vectors in R can have missing values that are represented as `NA`. Because all elements in a vector must be of the same type (i.e. logical, double, int, etc.) there are multiple types of `NA`. However, the casual R user will generally not be concerned with the different types for `NA`.

```
> x <- c(1, NA, 3)
> x
[1]  1 NA  3
> y <- as.character(NA)
> y
[1] NA
> typeof(NA) # default type of NA is logical
```

[25] http://cran.r-project.org/doc/manuals/r-release/R-lang.html

```
[1] "logical"
> typeof(y) # but we have coerced 'y' to a character vector
[1] "character"
```

R's `typeof()` function returns the internal type of each object. In the example above, `y` is a character vector.

Factors

Factors are one of R's compound data types. Internally, they are represented by integer vectors with a `levels` attribute. The following sample R session creates such a factor from a character vector:

```
> x <- sample(c("A", "B", "C"), size = 10, replace = TRUE)
> x
 [1] "C" "B" "B" "C" "A" "A" "B" "B" "C" "B"
> as.factor(x)
 [1] C B B C A A B B C B
Levels: A B C
```

Internally, the factor in this example is stored as an integer vector `c(3, 2, 2, 3, 1, 1, 2, 2, 3, 2)` which are the indices of the letters in the character vector `c(A, B, C)` stored in the `levels` attribute.

Lists

Lists are R's go-to structures for representing data structures. They can contain multiple elements, each of which can be of a different type. Record-like structures can be created by naming each element in the list. The `lm()` function, for example, returns a list that contains many details about the fitted linear model. The following R session shows the difference between a list and a list with named elements:

```
> l <- list("Jane", 23, c(6, 7, 9, 8))
> l
[[1]]
[1] "Jane"

[[2]]
[1] 23

[[3]]
[1] 6 7 9 8

> l <- list(name = "Jane", age = 23, scores = c(6, 7, 9, 8))
> l
$name
[1] "Jane"

$age
[1] 23

$scores
[1] 6 7 9 8
```

In R, lists are also known as *generic vectors*. They have a length that is equal to the number of elements in the list.

Data frames

Data frames are one of R's compound data types. They are lists of vectors, factors and/or matrices, all having the same length. It is one of the most important concepts in statistics and has equivalent

implementations in SAS[26] and SPSS[27].

The following sample R session shows how a data frame is constructed, what its attributes are and that it is indeed a list:

```
> df <- data.frame(x = seq(5), y = runif(5))
> df
  x         y
1 1 0.8773874
2 2 0.4977048
3 3 0.6719721
4 4 0.2135386
5 5 0.3834681
> class(df)
[1] "data.frame"
> attributes(df)
$names
[1] "x" "y"

$row.names
[1] 1 2 3 4 5

$class
[1] "data.frame"

> is.list(df)
[1] TRUE
```

Matrices and arrays

Besides one-dimensional vectors, R also knows two other classes to represent array-like data types: `matrix` and `array`. A matrix is simply an atomic vector with a `dim` attribute that contains a numeric vector of length two:

```
> x <- seq(9)
> class(x)
[1] "integer"
> dim(x) <- c(3, 3)
> class(x)
[1] "matrix"
> x
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

Likewise, an array is also a vector with a `dim` attribute that contains a numeric vector of length greater than two:

```
> y <- seq(8)
> dim(y) <- c(2,2,2)
> class(y)
[1] "array"
```

The example with the matrix shows that the elements in an array are stored in column-major order[28] which is important to know when we want to access R arrays from a Java application.

Note: In both examples for the `matrix` and `array` objects, the `class()` function derives the class from

[26] http://www.sas.com
[27] http://www.ibm.com/software/analytics/spss/
[28] http://en.wikipedia.org/wiki/Row-major_order#Column-major_order

the fact that the object is an atomic vector with the `dim` attribute set. Unlike data frames, these objects do not have a `class` attribute.

Overview of Renjin's type system

Renjin has corresponding classes for all of the R object types discussed in the section *A Java Developer's Guide to R Objects* (page 12). Table *Renjin's Java classes for common R object types* (page 15) summarizes these object types and their Java classes. In R, the object type is returned by the `typeof()` function.

Table 3.1: Renjin's Java classes for common R object types

R object type	Renjin class
logical	LogicalVector
integer	IntVector
double	DoubleVector
character	StringVector
complex	ComplexVector
raw	RawVector
list	ListVector
function	Function
environment	Environment
NULL	Null

There is a certain hierarchy in Renjin's Java classes for the different object types in R. Figure *Hierarchy in Renjin's type system* (page 16) gives a full picture of all classes that make up Renjin's type system. These classes are contained in the *org.renjin.sexp* Java package. The vector classes listed in table *Renjin's Java classes for common R object types* (page 15) are in fact abstract classes that can have different implementations. For example, the `DoubleArrayVector` (not shown in the figure) is an implementation of the `DoubleVector` abstract class. The `SEXP` (page 39), `Vector` (page 40), and `AtomicVector` classes are all Java interfaces.

Note: Renjin does not have classes for all classes of objects that are know to (base) R. This includes objects of class `matrix` and `array` which are represented by one of the `AtomicVector` classes and R's compound objects `factor` and `data.frame` which are represented by an `IntVector` and `ListVector` (page 38) respectively.

Pulling data from R into Java

Now that you have a good understanding of both R's object types and how these types are mapped to Renjin's Java classes, we can start by pulling data from R code into our Java application. A typical scenario is one where an R script performs a calculation and the result is pulled into the Java application for further processing.

Using the Renjin Script Engine as introduced in the *Evaluating R Language Code* (page 10), we can store the result of a calculation from R into a Java object. By default, the `eval()` method of `javax.script.ScriptEngine`[29] returns an `Object`[30], i.e. Java's object superclass. We can always cast this result to a `SEXP` (page 39) object. The following Java snippet shows how this is done and how the `Object.getClass()`[31] and `Class.getName()`[32] methods can be used to determine the actual class of the R result:

[29] http://docs.oracle.com/javase/8/docs/api/javax/script/ScriptEngine.html
[30] https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html
[31] https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#getClass()
[32] https://docs.oracle.com/javase/7/docs/api/java/lang/Class.html#getName()

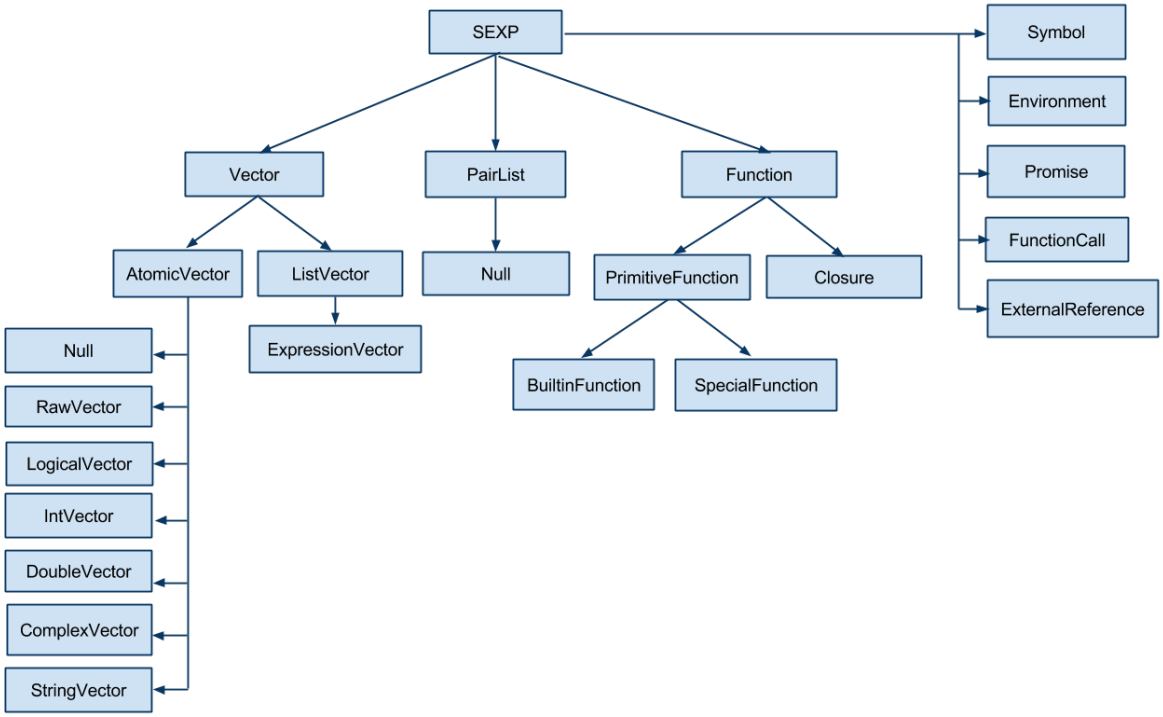

Fig. 3.1: Hierarchy in Renjin's type system

```
// evaluate Renjin code from String:
SEXP res = (SEXP)engine.eval("a <- 2; b <- 3; a*b");

// print the result to stdout:
System.out.println("The result of a*b is: " + res);
// determine the Java class of the result:
Class objectType = res.getClass();
System.out.println("Java class of 'res' is: " + objectType.getName());
// use the getTypeName() method of the SEXP object to get R's type name:
System.out.println("In R, typeof(res) would give '" + res.getTypeName() + "'");
```

This should write the following to the standard output:

```
The result of a*b is: 6.0
Java class of 'res' is: org.renjin.sexp.DoubleArrayVector
In R, typeof(res) would give 'double'
```

As you can see the `getTypeName` (page 39) method of the `SEXP` (page 39) class will return a String object with R's name for the object type.

Note: Don't forget to import `org.renjin.sexp.*` to make Renjin's type classes available to your application.

In the example above we could have also cast R's result to a *DoubleVector* object:

```
DoubleVector res = (DoubleVector)engine.eval("a <- 2; b <- 3; a*b");
```

or you could cast it to a *Vector*:

```
Vector res = (Vector)engine.eval("a <- 2; b <- 3; a*b");
```

You can't cast R integer results to a `DoubleVector`: the following snippet will throw a `ClassCastException`[33]:

```
// use R's 'L' suffix to define an integer:
DoubleVector res = (DoubleVector)engine.eval("1L");
```

Accessing individual elements of vectors

Now that we know how to pull R objects into our Java application we want to work with these data types in Java. In this section we show how individual elements of the Vector objects can be accessed in Java.

As you know, each vector type in R, and thus also in Renjin, has a length which can be obtained with the `length()` method. Individual elements of a vector can be obtained with the `getElementAsXXX()` methods where `XXX` is one of `Double`, `Int`, `String`, `Logical`, and `Complex`. The following snippet demonstrates this:

```
Vector x = (Vector)engine.eval("x <- c(6, 7, 8, 9)");
System.out.println("The vector 'x' has length " + x.length());
for (int i = 0; i < x.length(); i++) {
    System.out.println("Element x[" + (i + 1) + "] is " + x.getElementAsDouble(i));
}
```

This will write the following to the standard output:

```
The vector 'x' has length 4
Element x[1] is 6.0
Element x[2] is 7.0
Element x[3] is 8.0
Element x[4] is 9.0
```

As we have seen in the *Lists* (page 13) section above, lists in R are also known as *generic vectors*, but accessing the individual elements and their elements requires a bit more care. If an element (i.e. a vector) of a list has length equal to one, we can access this element directly using one of the `getElementAsXXX()` methods. For example:

```
ListVector x =
    (ListVector)engine.eval("x <- list(name = \"Jane\", age = 23, scores = c(6, 7, 8, 9))");
System.out.println("List 'x' has length " + x.length());
// directly access the first (and only) element of the vector 'x$name':
System.out.println("x$name is '" + x.getElementAsString(0) + "'");
```

which will result in:

```
List 'x' has length 3
x$name is 'Jane'
```

being printed to standard output. However, this approach will not work for the third element of the list as this is a vector with length greater than one. The preferred approach for lists is to get each element as a `SEXP` (page 39) object first and then to handle each of these accordingly. For example:

```
DoubleVector scores = (DoubleVector)x.getElementAsSEXP(2);
```

Dealing with matrices

As described in the section *Matrices and arrays* (page 14) above, matrices are simply vectors with the `dim` attribute set to an integer vector of length two. In order to identify a matrix in Renjin, we need to therefore check for the presence of this attribute and its value. Since any object in R can have one or more attributes, the `SEXP` (page 39) interface defines a number of methods for dealing with attributes.

[33] https://docs.oracle.com/javase/7/docs/api/java/lang/ClassCastException.html

In particular, hasAttributes (page 39) will return true if there are any attributes defined in an object and getAttributes (page 39) will return these attributes as a AttributeMap (page 37).

```
Vector res = (Vector)engine.eval("matrix(seq(9), nrow = 3)");
if (res.hasAttributes()) {
    AttributeMap attributes = res.getAttributes();
    Vector dim = attributes.getDim();
    if (dim == null) {
        System.out.println("Result is a vector of length " +
            res.length());

    } else {
        if (dim.length() == 2) {
            System.out.println("Result is a " +
                dim.getElementAsInt(0) + "x" +
                dim.getElementAsInt(1) + " matrix.");
        } else {
            System.out.println("Result is an array with " +
                dim.length() + " dimensions.");
        }
    }
}
```

Output:

```
Result is a 3x3 matrix.
```

For convenience, Renjin includes a wrapper class Matrix that provides easier access to the number of rows and columns.

Example:

```
// required import(s):
import org.renjin.primitives.matrix.*;

Vector res = (Vector)engine.eval("matrix(seq(9), nrow = 3)");
try {
    Matrix m = new Matrix(res);
    System.out.println("Result is a " + m.getNumRows() + "x"
        + m.getNumCols() + " matrix.");
} catch(IllegalArgumentException e) {
    System.out.println("Result is not a matrix: " + e);
}
```

Output:

```
Result is a 3x3 matrix.
```

Dealing with lists and data frames

The ListVector (page 38) class contains several convenience methods to access a list's components from Java. For example, we can the extract the components from a fitted linear model using the name of the element that contains those components. For example:

```
ListVector model = (ListVector)engine.eval("x <- 1:10; y <- x*3; lm(y ~ x)");
Vector coefficients = model.getElementAsVector("coefficients");
// same result, but less convenient:
// int i = model.indexOfName("coefficients");
// Vector coefficients = (Vector)model.getElementAsSEXP(i);

System.out.println("intercept = " + coefficients.getElementAsDouble(0));
System.out.println("slope = " + coefficients.getElementAsDouble(1));
```

Output:

```
intercept = -4.4938668397781774E-15
slope = 3.0
```

Handling errors generated by the R code

Up to now we have been able to execute R code without any concern for possible errors that may occur when the R code is evaluated. There are two common exceptions that may be thrown by the R code:

1. `ParseException` (page 41): an exception thrown by Renjin's R parser due to a syntax error and
2. `EvalException` (page 41): an exception thrown by Renjin when the R code generates an error condition, for example by the `stop()` function.

Here is an example which catches an exception from Renjin's parser:

```
// required import(s):
import org.renjin.parser.ParseException;

try {
    engine.eval("x <- 1 +/ 1");
} catch (ParseException e) {
    System.out.println("R script parse error: " + e.getMessage());
}
```

Output:

```
R script parse error: Syntax error at line 1 char 0: syntax error, unexpected '/'
```

And here's an example which catches an error condition thrown by the R interpreter:

```
// required import(s):
import org.renjin.eval.EvalException;

try {
    engine.eval("stop(\"Hello world!\")");
} catch (EvalException e) {
    // getCondition() returns the condition as an R list:
    Vector condition = (Vector)e.getCondition();
    // the first element of the string contains the actual error message:
    String msg = condition.getElementAsString(0);
    System.out.println("The R script threw an error: " + msg);
}
```

Output:

```
The R script threw an error: Hello world!
```

`EvalException.getCondition()` (page 41) is required to pull the condition message from the R interpreter into Java.

Pushing data from Java to R

Like many dynamic languages, R scripts are evaluated in the context of an environment that looks a lot like a dictionary. You can define new variables in this environment using the `javax.script`[34] API. This is achieved using the `ScriptEngine.put()`[35] method.

[34] http://docs.oracle.com/javase/8/docs/api/javax/script/package-summary.html
[35] http://docs.oracle.com/javase/8/docs/api/javax/script/ScriptEngine.html#put-java.lang.String,java.lang.Object-

Example:

```
engine.put("x", 4);
engine.put("y", new double[] { 1d, 2d, 3d, 4d });
engine.put("z", new DoubleArrayVector(1,2,3,4,5));
engine.put("hashMap", new java.util.HashMap());
// some R magic to print all objects and their class with a for-loop:
engine.eval("for (obj in ls()) { " +
    "cmd <- parse(text = paste('typeof(', obj, ')', sep = ''));" +
    "cat('type of ', obj, ' is ', eval(cmd), '\\n', sep = '') }");
```

Output:

```
type of hashMap is externalptr
type of x is integer
type of y is double
type of z is double
```

Renjin will implicitly convert primitives, arrays of primitives and `String`[36] instances to R objects. Java objects will be wrapped as R `externalptr` objects. The example also shows the use of the `DoubleArrayVector` constructor to create a double vector in R. You see that we managed to put a Java `java.util.HashMap`[37] object into the global environment of the R session: this is the topic of the chapter *Importing Java classes into R code* (page 27).

Thread-Safety

R code must always be evaluated in the context of a Session, which carries certain state, such as the Global Environment the list of loaded packages, global options, and the state of the random number generator.

Sessions are not thread-safe in the sense that two different R expressions cannot be evaluated concurrently within the same Session.

When using GNU R, a new R Session begins when the interpreter is started in a process, either from the command line, or via REngine. Because of the way that GNU R is implemented, every R Session must have its own process, and so you can not evaluate two R expressions concurrently in the same process.

Renjin is implemented differently, with the goal of being able to run multiple Sessions within the same process or the same JVM.

Every new instance of RenjinScriptEngine has its own independent Session. A single Session cannot execute multiple R scripts concurrently, but you can execute multiple R scripts concurrently within the same JVM, as long as each concurrent script has its own ScriptEngine.

If you want to execute several R scripts in parallel, you have a few options.

Thread-Local ScriptEngine

You can use the standard `java.lang.ThreadLocal`[38] class to maintain exactly one ScriptEngine per thread. This approach is useful when a small number of long-running worker threads each need to execute independent R scripts. This is the case for most Java Servlet containers, for example.

The following is a simple example based on the appengine-servlet[39] example:

[36] https://docs.oracle.com/javase/7/docs/api/java/lang/String.html
[37] https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html
[38] https://docs.oracle.com/javase/7/docs/api/java/lang/ThreadLocal.html
[39] https://github.com/bedatadriven/renjin-examples/blob/master/appengine-servlet/src/main/java/org/renjin/example/appengine/RenjinServlet.java#L44

```
public class MyServlet extends HttpServlet {

    /**
     * Maintain one instance of Renjin for each request thread.
     */
    private static final ThreadLocal<ScriptEngine> ENGINE = new ThreadLocal<ScriptEngine>();


    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException,␣
↪IOException {

        ScriptEngine engine = ENGINE.get();
        if(engine == null) {
            // create a new engine for this thread
            RenjinScriptEngineFactory factory = new RenjinScriptEngineFactory();
            engine = factory.getScriptEngine();
            ENGINE.set(engine);
        }
        // evaluate a script using the engine reference
    }
```

ScriptEngine Pooling

If your application, in constrast, uses a larger number of threads, or short-lived threads, it may make more sense to use a pool of ScriptEngines that can be leased to worker threads as needed.

The Apache Commons Pool[40] project provides a solid implementation that can be easily used to pool Renjin ScriptEngine sessions.

Sharing data between ScriptEngines

One of the principal advantages of running multiple, concurrent R Sessions in the same process is that you can share data between them.

By design, Renjin requires Vector implementations, which correspond to R objects of type "list", "character", "double", "integer", "complex", and "raw", to be _immutable_, which means that they cannot be changed after they are created. Environment and pairlist objects are _mutable_, both within the R language and in their Java implementations, and so _cannot_ be shared between Sessions.

This means that a data.frame object, for example, can be created or loaded once, and then shared between multiple concurrent Sessions which do their own computation on the dataset.

Customizing the Execution Context

R code must always be evaluated in the context of a Session, with tracks the global environment, which packages have been loaded, etc.

Each new `ScriptEngine` instance has it's own independent Session, and for each Session, Renjin allows you to customize the environment in which the R scripts are evaluated with regard to:

- *File System* (page 22)
- *Package Loading* (page 23)

[40] https://commons.apache.org/proper/commons-pool/

- *Class Loading* (page 24)
- *Command-Line Arguments* (page 24)

The `SessionBuilder` object provides an API for creating a customized Renjin ScriptEngine instance:

```
import javax.script.*;
import org.renjin.eval.*;
import org.renjin.script.*;

Session session = new SessionBuilder()
  .withDefaultPackages()
  .build();

RenjinScriptEngineFactory factory = new RenjinScriptEngineFactory();
RenjinScriptEngine engine = factory.getScriptEngine(session);
```

The sections below outline how the methods of the SessionBuilder object can be used to customize the execution context.

File System

The R language provides many builtin functions that allow scripts to interact with the file system, such as `getwd()`, `setwd()`, `file()`, `list.files()` etc.

In some contexts, however, direct access to the file system may not be appropriate. For example:

- You may want to limit the ability of a script to write to the file system.
- You may want to run an R script that expects data on the local file system, but redirect calls to `file()` to an alternate data source, such as a network resource or a database.

For this reason, Renjin mediates all calls to R file system functions through the Apache Commons Virtual File System (VFS)[41] library.

You can provide your own `FileSystemManager` instance to `SessionBuilder` configured for your particular use case.

The following example demonstrates how a ScriptEngine instance is configured by default:

```
DefaultFileSystemManager fsm = new DefaultFileSystemManager();
fsm.addProvider("jar", new JarFileProvider());
fsm.addProvider("file", new LocalFileProvider());
fsm.addProvider("res", new ResourceFileProvider());
fsm.addExtensionMap("jar", "jar");
fsm.setDefaultProvider(new UrlFileProvider());
fsm.setBaseFile(new File("/"));
fsm.init();

Session session = new SessionBuilder()
  .withDefaultPackages()
  .setFileSystemManager(fsm)
  .build();

RenjinScriptEngineFactory factory = new RenjinScriptEngineFactory();
RenjinScriptEngine engine = factory.getScriptEngine(session);
```

The `renjin-appengine` module provides a more complex example. There, the AppEngineContextFactory[42] class prepares a FileSystemManager that is configured with a AppEngineLocalFileSystem-

[41] https://commons.apache.org/proper/commons-vfs/

[42] https://github.com/bedatadriven/renjin/blob/87518a254c0d67788aa36e0ecb4038d25a6d9384/appengine/src/main/java/org/renjin/appengine/AppEngineContextFactory.java#L88-L88

Provider[43] subclass that provides read-only access to the servlet's directory. This allows R scripts access to "/WEB-INF/data/model.R", which is translated into the absolute path at runtime.

Package Loading

In contrast to GNU R, which always loads packages from the local file system, Renjin's package loading mechanism is customizable in order to support different use cases:

- When used interactively, analysts expect to be able download and run packages interactively from CRAN or BioConductor.
- When embedding specific R code in a web application, R packages can be declared as dependencies in Maven, Gradle, or SBT, and shipped along with the application just as any other JVM dependency.
- When allowing users to execute arbitrary R code in your application, you may want to limit R packages to some approved subset and load from an internal repository.

For this reason, Renjin mediates all package loading, such as calls to `library()` or to `require()` through a `PackageLoader` interface. This allows the application executing R code to choose an appropriate implementation.

Renjin itself provides two PackageLoader implementations:

- The `ClasspathPackageLoader`, which is the default for ScriptEngines and only loads packages that are already on the classpath.
- The `AetherPackageLoader`, which will download packages on demand from a remote Maven repository. This is used by the Renjin interactive REPL.

If you are embedding Renjin in your application and want packages to be loaded on demand, then you can configure SessionBuilder with an instance of an `AetherPackageLoader`.

The following example shows to add this dynamic behavior to a Renjin ScriptEngine, and adds an additional, internal Maven repository that is used to resolve packages:

```
RemoteRepository internalRepo = new RemoteRepository.Builder(
    "internal", /* id */
    "default",  /* type */
    "https://repo.acme.com/content/groups/public/").build();

List<RemoteRepository> repositories = new ArrayList<>();
repositories.add(internalRepo);
repositories.add(AetherFactory.renjinRepo());
repositories.add(AetherFactory.mavenCentral());

ClassLoader parentClassLoader = getClass().getClassLoader();

AetherPackageLoader loader = new AetherPackageLoader(parentClassLoader, repositories);

Session session = new SessionBuilder()
    .withDefaultPackages()
    .setPackageLoader(loader)
    .build();
```

You can also provide your own implementation of `PackageLoader` which resolves calls to `import()` and `require()` in any other way that meets your needs.

[43] https://github.com/bedatadriven/renjin/blob/87518a254c0d67788aa36e0ecb4038d25a6d9384/appengine/src/main/java/org/renjin/appengine/AppEngineLocalFilesSystemProvider.java

Class Loading

When R packages depend on JVM classes by using Renjin's `importClass()` directive, the JVM class is loaded indirectly via the Session's `PackageLoader` interface.

However, R scripts can also load JVM classes on an ad-hoc basis using the `import(com.acme.MyClass)` function.

Such classes are loaded not through the `PackageLoader` mechanism but through the Session object's own `ClassLoader` instance. This can also be set through the SessionBuilder object:

```
URLClassLoader classLoader = new URLClassLoader(
    new URL[] {
        new File("/home/alex/my_dir_with_jars").toURI().toURL(),
        new File("/home/alex/my_other_dir_with_jars").toURI().toURL()
    });

Session session = new SessionBuilder()
    .setClassLoader(classLoader)
    .build();
```

Command-Line Arguments

If you have an existing script that relies on the R *commandArgs()* function to obtain parameters from the environment, you can set these via the setCommandLineArguments method:

```
Session session = new SessionBuilder()
    .withDefaultPackages()
    .build();

session.setCommandLineArguments("/usr/bin/renjin", "X", "Y", "--args", "Z");

RenjinScriptEngineFactory factory = new RenjinScriptEngineFactory();
RenjinScriptEngine engine = factory.getScriptEngine(session);

engine.eval("print(commandArgs(trailingOnly = FALSE))");  // c("/usr/bin/renjin", "X", "Y", "--args
↪", "Z")
engine.eval("print(commandArgs(trailingOnly = TRUE))");   // c("Z")
```

Note that the Java Scripting API provides a richer API for moving values between Java and R. See *Moving Data between Java and R Code* (page 11).

CHAPTER 4

Using Renjin as an R Package

Though Renjin's ultimate goal is to be a complete, drop-in replacement for GNU R, in some cases you may want to run only part of your existing R code with Renjin, from within GNU R.

For this use case, you can load Renjin as a package and evaluate performance-sensitive sections of your code, without having to rewrite the R code.

Prerequisites

You must have Java 7 installed or higher. For best performance, we recommend using the latest version of Oracle or OpenJDK 8. The rJava package is also required, but should be installed automatically.

Installation

```
install.packages("https://nexus.bedatadriven.com/content/groups/public/org/renjin/renjin-gnur-
↪package/0.8.2353/renjin-gnur-package-0.8.2353.tar.gz")
```

Usage

```
library(renjin)

bigsum <- function(n) {
  sum <- 0
  for(i in seq(from = 1, to = n)) {
    sum <- sum + i
  }
  sum
}
bigsumc <- compiler::cmpfun(bigsum) # GNU R's byte code compiler

system.time(bigsum(1e8))
system.time(bigsumc(1e8))
system.time(renjin(bigsum(1e8)))
```

CHAPTER 5

Importing Java classes into R code

A true testament of the level of integration of Java and R in Renjin is the ability to directly access (public!) Java classes and methods from R code. Renjin introduces the `import()` function which adds a Java class to the environment from which it is called. In the section *Pushing data from Java to R* (page 19) in the previous chapter we had already seen how a Java class could be put into the global environment of the R session.

Consider the following sample R script:

```
import(java.util.HashMap)

# create a new instance of the HashMap class:
ageMap <- HashMap$new()

# call methods on the new instance:
ageMap$put("Bob", 33)
ageMap$put("Carol", 41)

print(ageMap$size())

age <- ageMap$get("Carol")
cat("Carol is ", age, " years old.\n", sep = "")

# Java primitives and their boxed types
# are automatically converted to R vectors:
typeof(age)
```

As we showed in the *Introduction* (page 1), we can execute this script using the `java.io.FileReader`[44] interface:

```
engine.eval(new java.io.FileReader("import_example.R"));
```

Output:

```
[1] 2
Carol is 41 years old.
```

The first line in the output is the output from the `print(ageMap$size())` statement.

[44] https://docs.oracle.com/javase/7/docs/api/java/io/FileReader.html

Bean classes

For Java classes with accessor methods that conform to the getXX() and setXX() Java bean convention, Renjin provides some special sauce to make access from R more natural.

Take the following Java bean as an example:

```
package beans;

public class Customer {
    private String name;
    private int age;

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public int getAge() { return age; }
    public void setAge(int age) { this.age = age; }
}
```

You can construct a new instance of the `Customer` class and provide initial values with named arguments to the constructor. For example:

```
import(beans.Customer)

bob <- Customer$new(name = "Bob", age = 36)
carol <- Customer$new(name = "Carol", age = 41)
cat("'bob' is an ", typeof(bob), ", bob$name = ", bob$name, "\n", sep = "")
```

If the previous R code is stored in a file `bean_example.R` then the following Java code snippet runs this example:

```
// required import(s):
import beans.Customer;

engine.eval(new java.io.FileReader("bean_example.R"));
```

Output:

```
'bob' is an externalptr, bob$name = Bob
```

CHAPTER 6

Writing Renjin Extensions

This chapter can be considered as Renjin's equivalent of the Writing R Extensions[45] manual for GNU R. Here we discuss how you create extensions, or *packages* as they are referred to in R, for Renjin. Packages allow you to group logically related functions and data together in a single archive which can be reused and shared with others.

Renjin packages do not differ much from packages for GNU R. One notable difference is that Renjin treats unit tests as first class citizens, i.e. Renjin includes a package called *hamcrest* that provides functionality for writing unit tests right out of the box. We encourage you to include as many unit tests with your package as possible.

One feature currently missing for Renjin packages is the ability to document your R code. You can use Javadoc to document your Java classes and methods.

Package directory layout

The files in a Renjin package must be organized in a directory structure that adheres to the Maven standard directory layout[46]. A directory layout that will cover most Renjin packages is as follows:

```
projectdir/
    src/
        main/
            java/
                ...
            R/
            resources/
        test/
            java/
                ...
            R/
            resources/
    NAMESPACE
    pom.xml
```

The table *Directories in a Renjin package* (page 30) gives a short description of the directories and files in this layout.

[45] http://cran.r-project.org/doc/manuals/r-release/R-exts.html
[46] http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html

Table 6.1: Directories in a Renjin package

Directory	Description
src/main/java	Java source code (*.java files)
src/main/R	R source code (*.R files)
src/main/resources	Files and directories to be copied to the root of the generated JAR file
src/test/java	Unit tests written in Java using JUnit
src/test/R	Unit tests written in R using Renjin's Hamcrest package
src/test/resource	Files available to the unit tests (not copied into the generated JAR file)
NAMESPACE	Almost equivalent to R's NAMESPACE file
pom.xml	Maven's Project Object Model file

The functionality of the *DESCRIPTION* file used by GNU R packages is replaced by a Maven *pom.xml* (POM) file. In this file you define the name of your package and any dependencies, if applicable. The POM file is used by Renjin's Maven plugin to create the package. This is the subject of the next section.

Renjin Maven plugin

Whereas you would use the commands `R CMD check`, `R CMD build`, and `R CMD INSTALL` to check, build (i.e. package), and install packages for GNU R, packages for Renjin are tested, packaged, and installed using a Maven plugin. The following XML file can be used as a *pom.xml* template for all Renjin packages:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.acme</groupId>
    <artifactId>foobar</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <!-- general information about your package -->
    <name>Package name or title</name>
    <description>A short description of your package.</description>
    <url>http://www.url.to/your/package/website</url>
    <licenses>
        <!-- add one or more licenses under which the package is released -->
        <license>
            <name>Apache License version 2.0</name>
            <url>http://www.apache.org/licenses/LICENSE-2.0.html</url>
        </license>
    </licenses>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <renjin.version>0.8.2356</renjin.version>
    </properties>

    <dependencies>
        <!-- the script engine is convenient even if you do not use it explicitly -->
        <dependency>
            <groupId>org.renjin</groupId>
            <artifactId>renjin-script-engine</artifactId>
            <version>${renjin.version}</version>
        </dependency>
        <!-- the hamcrest package is only required if you use it for unit tests -->
        <dependency>
```

```
            <groupId>org.renjin</groupId>
            <artifactId>hamcrest</artifactId>
            <version>${renjin.version}</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <repositories>
        <repository>
            <id>bedatadriven</id>
            <name>bedatadriven public repo</name>
            <url>https://nexus.bedatadriven.com/content/groups/public/</url>
        </repository>
    </repositories>

    <pluginRepositories>
        <pluginRepository>
            <id>bedatadriven</id>
            <name>bedatadriven public repo</name>
            <url>https://nexus.bedatadriven.com/content/groups/public/</url>
        </pluginRepository>
    </pluginRepositories>

    <build>
        <plugins>
            <plugin>
                <groupId>org.renjin</groupId>
                <artifactId>renjin-maven-plugin</artifactId>
                <version>${renjin.version}</version>
                <executions>
                    <execution>
                        <id>build</id>
                        <goals>
                            <goal>namespace-compile</goal>
                        </goals>
                        <phase>process-classes</phase>
                    </execution>
                    <execution>
                        <id>test</id>
                        <goals>
                            <goal>test</goal>
                        </goals>
                        <phase>test</phase>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

This POM file provides a lot of information:

- fully qualified name of the package, namely *com.acme.foobar*;
- package version, namely *1.0-SNAPSHOT*;
- package dependencies and their versions, namely the Renjin Script Engine and the *hamcrest* package (see the next section);
- BeDataDriven's public repository to look for the dependencies if it can't find them locally or in Maven Central;

Important: Package names is one area where Renjin takes a different approach to GNU R and adheres

to the Java standard of using fully qualified names. The package in the example above must be loaded using its fully qualified name, that is with `library(com.acme.foobar)` or `require(com.acme.foobar)`. The group ID (*com.acme* in this example) is traditionally a domain over which only you have control. The artifact ID should have only lower case letters and no strange symbols[47]. The term *artifact* is used by Maven to refer to the result of a build which, in the context of this chapter, is always a package.

Now you can use Maven to test, package, and install your package using the following commands:

mvn test run the package tests (both the Java and R code tests)

mvn package create a JAR file of the package (named *foobar-1.0-SNAPSHOT.jar* in the example above) in the `target` folder of the package's root directory

mvn install install the artifact (i.e. package) into the local repository

mvn deploy upload the artifact to a remote repository (requires additional configuration)

mvn clean clean the project's working directory after a build (can also be combined with one of the previous commands, for example: `mvn clean install`)

Package NAMESPACE file

Since R version 2.14, packages are required to have a `NAMESPACE` file and the same holds for Renjin. Because of dynamic searching for objects in R, the use of a `NAMESPACE` file is good practice anyway. The `NAMESPACE` file is used to explicitly define which functions should be *imported* into the package's namespace and which functions the package exposes (i.e. *exports*) to other packages. Using this file, the package developer controls how his or her package finds functions.

Usage of the `NAMESPACE` in Renjin is almost exactly the same as in GNU R save for two differences:

1. the directives related to S4 classes are not yet supported by Renjin and
2. Renjin accepts the directive `importClass()` for importing Java classes into the package namespace.

Here is an overview of the namespace directives that Renjin supports:

`export(f)` or `export(f, g)` Export an object `f` (singular form) or multiple objects `f` and `g` (plural form). You can add as many objects to this directive as you like.

`exportPattern("^[^\\.]")` Export all objects whose name does not start with a period ('.'). Although any regular expression can be used in this directive, this is by far the most common one. It is considered to be good practice not to use this directive and to explicitly export objects using the `export()` directive.

`import(foo)` or `import(foo, bar)` Import all exported objects from the package named `foo` (and `bar` in the plural form). Like the `export()` directive, you can add as many objects as you like to this directive.

`importFrom(foo, f)` or `importFrom(foo, f, g)` Import only object `f` (and `g` in the plural form) from the package named `foo`.

`S3method(print, foo)` Register a print (S3) method for the `foo` class. This ensures that other packages understand that you provide a function `print.foo()` that is a print method for class `foo`. The `print.foo()` does not need to be exported.

`importClass(com.acme.myclass)` A namespace directive which is unique to Renjin and which allows Java classes to be imported into the package namespace. This directive is actually a function which does the same as Renjin's `import()` function that was introduced in the chapter *Importing Java classes into R code* (page 27).

To summarize: the R functions in your package have access to all R functions defined within your package (also those that are not explicitely exported) as well as the Java classes imported into the package names

[47] http://maven.apache.org/guides/mini/guide-naming-conventions.html

using the `importClass` directive. Other packages only have access to the R objects that your package exports as well as to the public Java classes. Since Java has its own mechanism to control the visibility of classes, there is no `exportClass` directive in the `NAMESPACE` file.s

Using the *hamcrest* package to write unit tests

Renjin includes a built-in package called *hamcrest* for writing unit tests using the R language. The package and its test functions are inspired by the Hamcrest framework. From hamcrest.org[48]: *Hamcrest is a framework for writing matcher objects allowing 'match' rules to be defined declaratively.* The Wikipedia article on Hamcrest[49] gives a good and short explanation of the rationale behind the framework.

If you are familiar with the 'expectation' functions used in the testthat[50] package for GNU R, then you will find many similarities with the assertion and matcher functions in Renjin's *hamcrest* package.

A test is a single R function with no arguments and a name that starts with `test.`. Each test function can contain one or more assertions and the test fails if at least one of the assertions throws an error. For example, using the package defined in the previous section:

```
library(hamcrest)
library(com.acme.foobar)

test.df <- function() {
    df <- data.frame(x = seq(10), y = runif(10))

    assertThat(df, instanceOf("data.frame"))
    assertThat(dim(df), equalTo(c(10,2)))
}
```

Test functions are stored in R script files (i.e. files with extension `.R` or `.S`) in the `src/test/R` folder of your package. Each file should start with the statement `library(hamcrest)` in order to attach the *hamcrest* package to the search path as well as a `library()` statement to load your own package. You can put test functions in different files to group them according to your liking.

The central function is the `assertThat(actual, expected)` function which takes two arguments: `actual` is the object about which you want to make an assertion and `expected` is the matcher function that defines the rule of the assertion. In the example above, we make two assertions about the data frame `df`, namely that it should have class *data.frame* and that its dimension is equal to the vector `c(10, 2)` (i.e. ten rows and two columns). The following sections describe the available matcher functions in more detail.

Testing for (near) equality

Use `equalTo()` to test if `actual` is equal to `expected`:

```
assertThat(actual, equalTo(expected))
```

Two objects are considered to be equal if they have the same length and if `actual == expected` is `TRUE`.

Use `identicalTo()` to test if `actual` is identical to `expected`:

```
assertThat(actual, identicalTo(expected))
```

Two objects are considered to be identical if `identical(actual, expected)` is `TRUE`. This test is much stricter than `equalTo()` as it also checks that the type of the objects and their attributes are the same.

Use `closeTo()` to test for near equality (i.e. with some margin of error as defined by the `delta` argument):

[48] https://code.google.com/p/hamcrest/wiki/Tutorial
[49] http://en.wikipedia.org/wiki/Hamcrest
[50] http://cran.r-project.org/web/packages/testthat/index.html

```
assertThat(actual, closeTo(expected, delta))
```

This assertion only accepts numeric vectors as arguments and `delta` must have length 1. The assertion also throws an error if `actual` and `expected` do not have the same length. If their lengths are greater than 1, the largest (absolute) difference between their elements may not exceed `delta`.

Testing for TRUE or FALSE

Use `isTrue()` and `isFalse()` to check that an object is identical to `TRUE` or `FALSE` respectively:

```
assertThat(actual, isTrue())
assertTrue(actual) # same, but shorter
assertThat(actual, identicalTo(TRUE)) # same, but longer
```

Testing for class inheritance

Use `instanceOf()` to check if an object *inherits* from a class:

```
assertThat(actual, instanceOf(expected))
```

An object is assumed to inherit from a class if `inherits(actual, expected)` is `TRUE`.

Tip: Renjin's *hamcrest* package also exists as a GNU R package with the same name available at https://github.com/bedatadriven/hamcrest. If you are writing a package for both Renjin and GNU R, you can use the *hamcrest* package to check the compatibility of your code by running the test files in both Renjin and GNU R.

Understanding test results

When you run `mvn test` within the directory that holds the POM file (i.e. the root directory of your package), Maven will execute both the Java and R unit tests and output various bits of information including the test results. The results for the Java tests are summarized in a section marked with:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
```

and which will summarize the test results like:

```
Results :

Tests run: 5, Failures: 1, Errors: 0, Skipped: 0
```

The results of the R tests are summarized in a section marked with:

```
-------------------------------------------------------
 R E N J I N   T E S T S
-------------------------------------------------------
```

The R tests are summarized per R source file which will look similar to the following example:

```
Running tests in /home/foobar/mypkg/src/test/R
Running function_test.R
No default packages specified
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.898
```

Note that the number of tests run is equal to the number of `test.*` functions in the R source file + 1 as running the test file is also counted as a test.

CHAPTER 7

Renjin Java API specification

This chapter includes a selection of public classes and methods in some of Renjin's Java packages which are most useful to users of Renjin looking to exchange data between Java and R code.

org.renjin.sexp

The `org.renjin.sexp` package contains Renjin's classes that represent the different object types in R.

AttributeMap

class **`AttributeMap`**

Renjin's implementation to store object attributes. See the *Attributes* (page 12) section for a short introduction to attributes on objects in R.

SEXP (page 39) **`getClassVector`**`()`

See `hasClass` (page 37) for an example.

Returns a character vector of classes or `null` if no `class` attribute is defined

Vector (page 40) **`getDim`**`()`

Returns the `dim` attribute as a `Vector` (page 40) or `null` if no dimension is defined

AtomicVector **`getNamesOrNull`**`()`

Returns the `names` attribute as a `AtomicVector` or `null` if no names are defined

boolean **`hasClass`**`()`

Returns `true` if the `class` attribute exists, `false` otherwise

Example:

```
Vector df = (Vector)engine.eval("df <- data.frame(x = seq(5), y = runif(5))");
AttributeMap attributes = df.getAttributes();
if (attributes.hasClass()) {
    System.out.println("Classes defined on 'df': " +
        (Vector)attributes.getClassVector());
}
```

Output:

```
Classes defined on 'df': "data.frame"
```

ListVector (page 38) **toVector**()
Convert the object's attributes to a list.

Returns attributes as a `ListVector` (page 38)

ListVector

class **ListVector** extends AbstractVector
Renjin's class that represent R lists and data frames. Data frames are lists with the restriction that all elements, which are atomic vectors, have the same length.

SEXP (page 39) **getElementAsSEXP**(int *index*)

Parameters

- **index** – zero-based index

Returns a `SEXP` that can be cast to a vector type

Vector (page 40) **getElementAsVector**(String[51] *name*)
Convenience method to get the named element of a list. See the section *Dealing with lists and data frames* (page 18) for an example.

Parameters

- **name** – element name as string

Returns a `Vector` (page 40)

int **indexOfName**(String[52] *name*)

Parameters

- **name** – element name as string

Returns zero-based index of `name` in the names attribute, -1 if `name` is not present in the names attribute or if the names attribute is not set

boolean **isElementNA**(int *index*)
Check if an element of a list is `NA`.

Parameters

- **index** – zero-based index

Returns `true` if the element at position `index` is an `NA`, false otherwise

Logical

enum **Logical**
A logical value in R can be `TRUE`, `FALSE`, or the logical `NA`.

static *Logical* (page 38) **valueOf**(boolean *b*)
Turn a Java boolean into an R logical value.

Parameters

- **b** – `true` or `false`

Returns R's `TRUE` or `FALSE` as Renjin's representation of a logical value

static *Logical* (page 38) **valueOf**(int *i*)
Turn an integer into an R logical value.

Parameters

- **i** – an integer value

Returns TRUE if i is 1, FALSE if i is 0, or (logical) NA otherwise

SEXP

interface **SEXP**

Renjin's superclass for all objects that are mapped from R's object types.

AttributeMap (page 37) **getAttributes**()

Get the attributes of an object as a AttributeMap which is Renjin's way of working with object attributes. R stores attributes as pairlists which are essentially the same as *generic lists*, therefore these attributes can safely be stored in a list. Renjin provides a toVector() (page 38) method to do just that.

Returns the attributes for the object as a, possibly empty, AttributeMap (page 37)

Example:

```
ListVector res = (ListVector)engine.eval(
        "list(name = \"Jane\", age = 23, scores = c(6, 7, 8, 9))");
    // use ListVector.toString() method to display the list:
    System.out.println(res);
    if (res.hasAttributes()) {
        AttributeMap attributes = res.getAttributes();
        // convert the attribute map to something more convenient:
        ListVector attributeList = attributes.toVector();
        System.out.println("The list has "
            + attributeList.length() + " attribute(s)");
    }
```

Output:

```
list(name = "Jane", age = 23.0, scores = c(6, 7, 8, 9))
The list has 1 attribute(s)
```

String[53] **getTypeName**()

Get the type of the object as it is known in R, i.e. the result of R's typeof() function.

Returns the object type as a string

Example:

```
Vector x = (Vector)engine.eval("NA");
System.out.println("typeof(NA) = " + x.getTypeName());
```

Output:

```
typeof(NA) = logical
```

boolean **hasAttributes**()

Check for the presence of attributes. See getAttributes (page 39) for an example.

Returns true if the object has at least one attribute, false otherwise

int **length**()

Get the length of the object. All objects in R have a length and this method gives the same result as R's length() function. Functions always have length 1 and the NULL object always has length 0. The length of an environment is equal to the number of objects inside the environment.

Returns length of the vector as an integer

Vector

interface **Vector** extends *SEXP* (page 39)

An interface which represents all vector object types in R: atomic vectors and *generic vectors* (i.e. *Lists* (page 13)).

double **getElementAsDouble**(int *index*)

Parameters

- **index** – zero-based index

Returns the element at `index` as a double, converting if necessary; `NaN` if no conversion is possible

Example:

```
// create a string vector in R:
Vector x = (Vector)engine.eval("c(\"foo\", \"bar\")");
double x1 = x.getElementAsDouble(0);
if (Double.isNaN(x1)) {
    System.out.println("Result is NaN");
}
String s = x.getElementAsString(0);
System.out.println("First element of result is " + s);
// call the toString() method of the underlying StringArrayVector:
System.out.println("Vector as defined in R: " + x);
```

Output:

```
Result is NaN
First element of result is foo
Vector as defined in R: c(foo, bar)
```

Note: All of the classes that implement the `Vector` (page 40) interface have a `toString()` method that will display (a short form of) the content of the vector. This method is provided for debugging purposes only.

int **getElementAsInt**(int *index*)

Parameters

- **index** – zero-based index

Returns the element at `index` as an integer, converting if necessary; `NaN` if no conversion is possible

String[54] **getElementAsString**(int *index*)

Parameters

- **index** – zero-based index

Returns the element at `index` as a string

Logical (page 38) **getElementAsLogical**(int *index*)

Parameters

- **index** – zero-based index

Returns the element at `index` as Renjin's representation of a boolean value

org.renjin.primitives.matrix

Matrix

class **Matrix**
Wrapper class for a `Vector` (page 40) with two dimensions. Simplifies interaction with R matrices from Java code.

Matrix(*Vector* (page 40) *vector*)
Constructor for creating a matrix from a `Vector` (page 40). Checks if the dimension attribute is present and has length 2, throws an `IllegalArgumentException`[55] if not. See the section *Dealing with matrices* (page 17) for an example.

Parameters

- **vector** – a vector with two dimensions

Throws

- **IllegalArgumentException** – if the `dim` attribute of `vector` does not have length 2

int **getNumRows**()

Returns number of rows in the matrix

int **getNumCols**()

Returns number of columns in the matrix

Exceptions

class **ParseException** extends RuntimeException[56]
An exception thrown by Renjin's parser when there is an error in parsing R code, usually due to a syntax error. See *Handling errors generated by the R code* (page 19) for an example that catches this exception.

class **EvalException** extends RuntimeException[57]
An exception thrown by Renjin's interpreter when the R code generates an error condition, e.g. by the `stop()` function. See *Handling errors generated by the R code* (page 19) for an example that catches this exception.

SEXP (page 39) **getCondition**()

Returns a `SEXP` (page 39) that is a list with a single named element `message`. Use `getElementAsString()` (page 40) to obtain the actual error message.

[55] https://docs.oracle.com/javase/7/docs/api/java/lang/IllegalArgumentException.html

Index

A

as.factor()
    R function, 13
atomic vectors, 12
attr()
    R function, 12
AttributeMap (Java class), 37
attributes()
    R function, 12

D

data.frame()
    R function, 14
dim()
    R function, 14

E

EvalException (Java class), 41
exceptions, 19, 41

F

factors, 13

G

generic vectors, 13
getAttributes() (Java method), 39
getClassVector() (Java method), 37
getCondition() (Java method), 41
getDim() (Java method), 37
getElementAsDouble(int) (Java method), 40
getElementAsInt(int) (Java method), 40
getElementAsLogical(int) (Java method), 40
getElementAsSEXP(int) (Java method), 38
getElementAsString(int) (Java method), 40
getElementAsVector(String) (Java method), 38
getNamesOrNull() (Java method), 37
getNumCols() (Java method), 41
getNumRows() (Java method), 41
getTypeName() (Java method), 39

H

hasAttributes() (Java method), 39
hasClass() (Java method), 37

I

import()
    Renjin function, 27
indexOfName(String) (Java method), 38
install.packages()
    R function, 6
is.list()
    R function, 14
isElementNA(int) (Java method), 38

J

Java
    length(), 39

L

length()
    Java, 39
    R function, 39
length() (Java method), 39
library()
    R function, 6
ListVector (Java class), 38
Logical (Java enum), 38

M

Matrix (Java class), 41
matrix()
    R function, 18
Matrix(Vector) (Java constructor), 41

N

NA, 12
NULL, 12

O

org.renjin.eval (package), 41
org.renjin.parser (package), 41
org.renjin.primitives.matrix (package), 41
org.renjin.sexp (package), 37

P

ParseException (Java class), 41
print()

R function, 11

R

R function
as.factor(), 13
attr(), 12
attributes(), 12
data.frame(), 14
dim(), 14
install.packages(), 6
is.list(), 14
length(), 39
library(), 6
matrix(), 18
print(), 11
stop(), 19
typeof(), 15
Renjin function
import(), 27
REPL, 5, 25

S

SEXP (Java interface), 39
stop()
R function, 19

T

toVector() (Java method), 38
typeof()
R function, 15

V

valueOf(boolean) (Java method), 38
valueOf(int) (Java method), 38
Vector (Java interface), 40
version, 2