Extracted from:

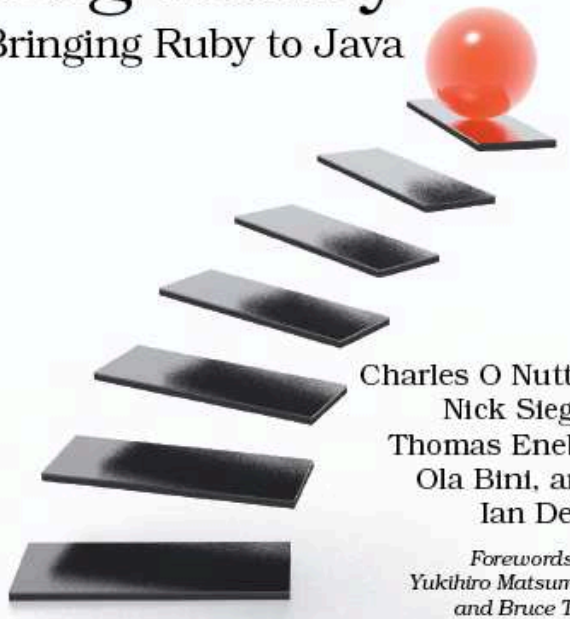
Using JRuby

Bringing Ruby to Java

The Pragmatic Programmers

Using JRuby

Bringing Ruby to Java



Charles O Nutter,
Nick Sieger,
Thomas Enebo,
Ola Bini, and
Ian Dees

Forewords by
Yukihiro Matsumoto
and Bruce Tate

Edited by Jacquelyn Carter

The Facets of Ruby Series

Chapter 2

Driving Java from Ruby

It might be tempting to think of Java/Ruby integration as nothing more than calling from one language to another. But that's selling JRuby short. In a typical project, you're really *interacting* with both platforms. You might construct a Ruby object, pass it to a Java function, and watch the Java code call other Ruby methods you've defined.

In this chapter, we'll look at cases where the interaction starts in Ruby: calling Java methods from Ruby code, implementing Java interfaces in Ruby, and so on. In the next chapter, we'll start with a Java program and work our way back to Ruby.

2.1 Seeing Java Through Ruby Glasses

The first use case for JRuby, and still the most common one today, is calling a Java method from Ruby. Why would someone want to do this? There are thousands of reasons. Here are just a few of the things you can do with this interoperability:

- Visualize geographic data with NASA's World Wind project.[1] In Figure 2.1, on the following page, you can see a map of our hometowns that we put together with just a few lines of Ruby.
- Render beautiful SVG graphics with the Apache Batik project, like the folks at Atomic Object did for their cross-platform simulation app.[2]. The elegant visuals they achieved are shown in Figure 2.2,

1. http://worldwind.arc.nasa.gov
2. http://spin.atomicobject.com/2009/01/30/ruby-for-desktop-applications-yes-we-can

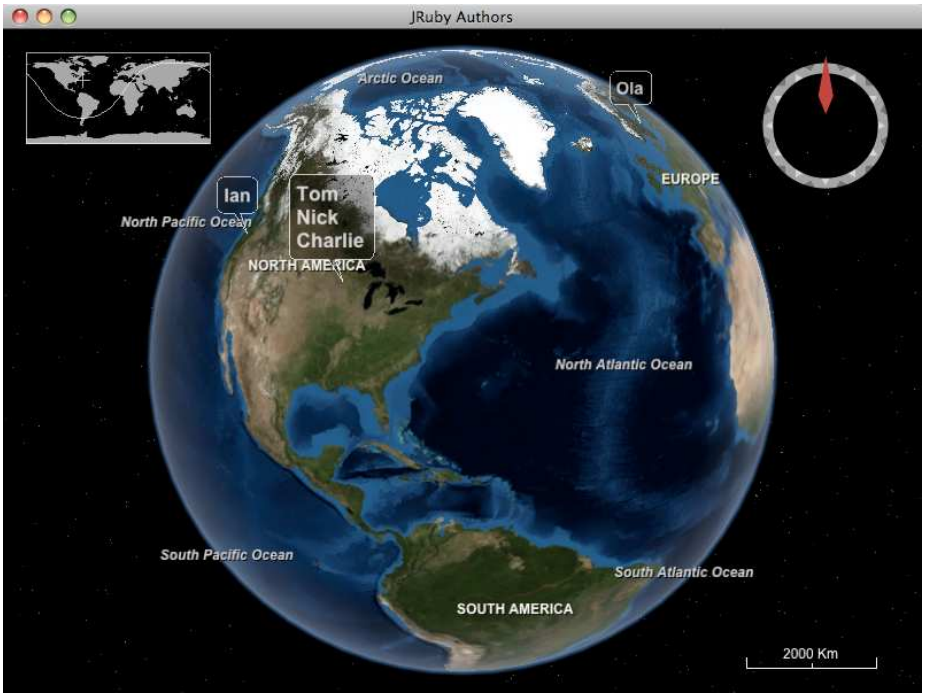


Figure 2.1: Locating JRuby authors with World Wind

on the next page. (Image used with permission of the Avraham Y. Goldratt Institute, LP.)

- Handle a protocol or data format for which a Java library is the best fit. For example, you might choose the Java-based iText library to add PDF support to your Ruby program—especially if you need digital signatures or some other feature specific to iText.[3]
- Slay the "cross-platform Ruby GUI" dragon by writing a Swing or SWT program in Ruby.
- Boost the performance of a Ruby program. For example, the team behind the Redcar text editor know they will always have the option of dropping down into Java for any performance-critical parts.[4]

3. http://www.itextpdf.com
4. http://redcareditor.com

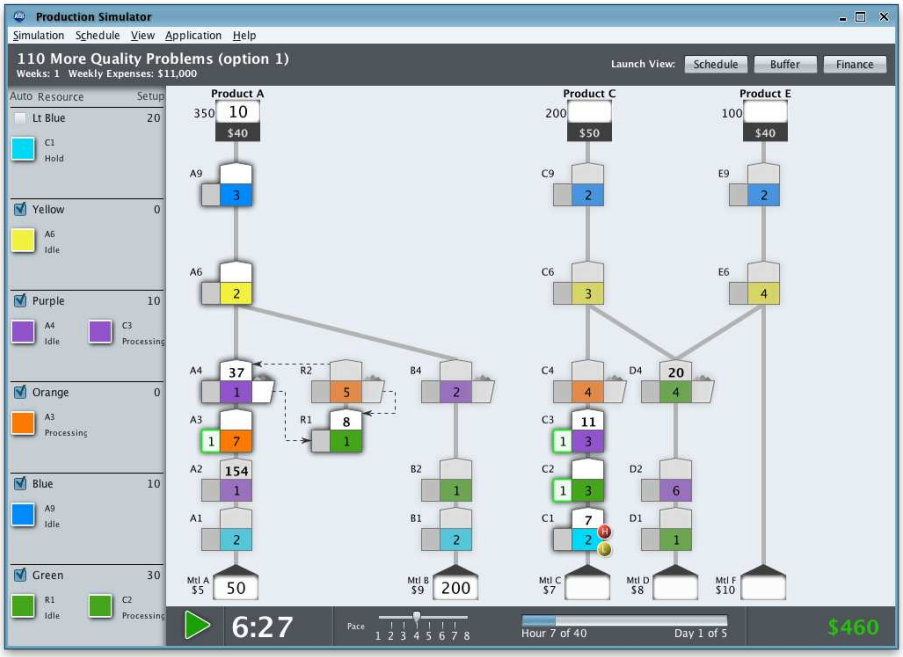


Figure 2.2: Simulating industrial processes with Batik

- Tame a legacy Java project by walling off the old spaghetti code behind a clean Ruby interface.
- Sneak Ruby into a Java shop; after all, JRuby is "just another .jar file."
- Write great tests for your Java code, using one of Ruby's outstanding test frameworks.
- Index and search huge amounts of text with the Lucene search engine.[5]
- Write a database-backed web application in the Rails framework. Behind the scenes, Rails's database adapters call into Java's database libraries to do the heavy SQL lifting.

All of these scenarios are the bread and butter of JRuby and are well supported. But as in any domain where two languages meet, there are

5. http://lucene.apache.org

some subtleties, gotchas, and impedance mismatches.[6] This chapter will address many of these edge cases.

First things first, though. We'll lead off with the basics of accessing Java classes from JRuby, starting with how your Ruby code can load and interact with Java libraries. Then we'll explore the details of parameter passing and automatic type conversions. Finally, we'll show a few tips and tricks to make Java classes and objects a natural part of your Ruby programs.

A Simple Example: Wrapping a Library

Let's start with a working program to drive a Java library. We'll expand on one of the examples we described earlier: using the iText library to generate a PDF file. This will be just enough to give a hint of the flavor of driving Java, without having to bang our heads against the more obscure edge cases (yet). Download the latest .jar (for example, iText-5.0.1.jar) from the official site, and copy it into the directory where you're following along in code.[7] Next, add this snippet to a file called pdf_demo.rb:

Download java_from_ruby/pdf_demo.rb

```
require 'java'

pdf  = com.itextpdf.text.Document.new
para = com.itextpdf.text.Paragraph.new 'Brought to you by JRuby'
file = java.io.FileOutputStream.new 'pdf_demo.pdf'

com.itextpdf.text.pdf.PdfWriter.get_instance pdf, file

pdf.open
pdf.add para
pdf.close
```

In the spirit of walking before we run, let's walk through the source before we run the program. In the opening lines, we create a few Java objects the same way we'd create Ruby ones—by calling the class's new method. We use a typical full-package name for each class (for example, com.itextpdf.text.Document).

6. The term *impedance mismatch* comes from electrical engineering. It refers to the power lost to reflection when two circuits are connected. It's also a poetic way to describe the conceptual losses between two different software domains.
7. http://sf.net/projects/itext/files

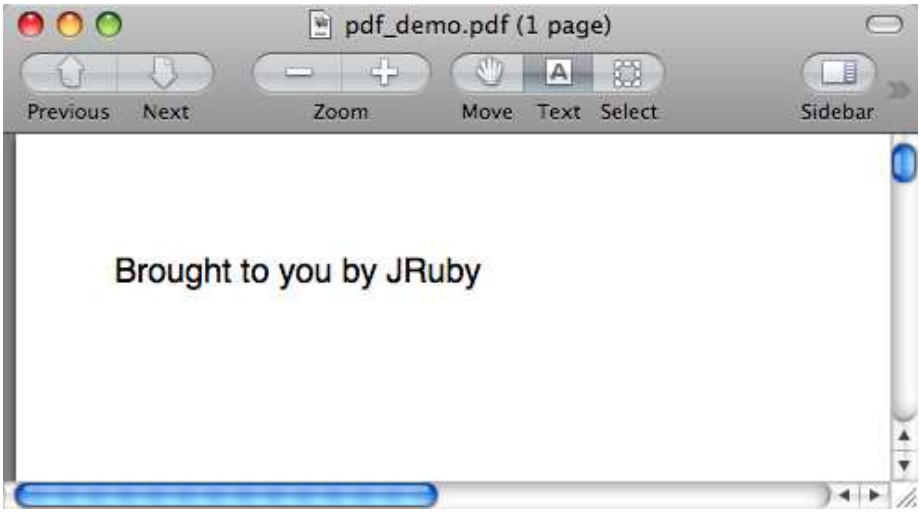


Figure 2.3: The generated PDF in all its glory

In JRuby, Java methods look and act like Ruby ones. All the method names you see in this snippet—open, add, and close—belong to Java classes. That includes get_instance, an alias JRuby has created for getInstance() to make it fit better in the Ruby universe.

Some Ruby types get converted into their Java counterparts automatically for you, such as the "Brought to you..." string. Others need a little hand holding; you'll see a few of those cases later.

Now that you've had a chance to look through the code, let's run it. You'll need to tell JRuby where the external iText library lives, by setting the classpath. Java provides the -cp option for this purpose. JRuby will forward any option to the underlying Java runtime if you preface it with -J. Go ahead and try the following command, adjusting the version number of iText to match what you downloaded:

```
$ jruby -J-cp iText-5.0.1.jar pdf_demo.rb
```

That'll create a PDF file called pdf_demo.pdf in the same directory. If you open it, you should see something like Figure 2.3. It's not the most visually breathtaking use of the format, but you get the idea.

Another Simple Example: Extending a Ruby Program

Let's consider another big use case: taking an existing Ruby program and rewriting part of it in Java for speed. Just for fun, we'll make this one a GUI app, albeit a trivial one. We're going to build a calculator for the famous stack-busting Ackermann function.[8] The Ruby code for this reads like the official mathematical definition:

Download java_from_ruby/ackerizer.rb

```
class Ackermann
  def self.ack(m, n)
    return n + 1                    if m == 0
    return ack(m - 1, 1)            if n == 0
    return ack(m - 1, ack(m, n - 1))
  end
end
```

This implementation is far too slow for a production app, as will become painfully clear after we wrap a Swing user interface around it. To build our GUI, we're going to use a Ruby helper called Rubeus.[9] Go ahead and install that now:

```
$ jruby -S gem install rubeus
```

We'll talk more about Rubeus in Chapter 10, *Building GUIs with Swing*, on page 237. For this short example, the code is simple enough to show without much explanation. It's just a couple of text inputs and a button:

Download java_from_ruby/ackerizer.rb

```
require 'rubygems'
require 'java'
require 'rubeus'

include Rubeus::Swing

JFrame.new('Ackerizer') do |frame|
  frame.layout = java.awt.FlowLayout.new

  @m = JTextField.new '3'
  @n = JTextField.new '9'

  JButton.new('->') do
    @result.text = Ackermann.ack(@m.text.to_i,
                                 @n.text.to_i).to_s
  end
```

8. http://en.wikipedia.org/wiki/Ackermann_function
9. http://code.google.com/p/rubeus/

Figure 2.4: The Ackermann calculator

```
  @result = JTextField.new 10

  frame.pack
  frame.show
end
```

Go and throw those two code snippets into a file called ackerizer.rb, and then launch the app. You'll most likely need to increase the JVM's stack size, using Java's standard -Xss setting together with JRuby's -J "passthrough" option:

```
$ jruby -J-Xss64m ackerizer.rb
```

You should see something like Figure 2.4. Try clicking the button to calculate ack(3, 9). The results will probably take several seconds to appear in the window. Because our app is a one-trick pony, there's only one suspect worth investigating: the ack method.[10]

There's a lot we could try in Ruby before jumping into Java. At the very least, we should be storing our intermediate values so that we don't have to calculate them over and over. But let's say you've done all that, and you still need faster results. Here's how you'd move the calculation into a Java class:

Download java_from_ruby/Ackermann.java

```
public class Ackermann {
    public static int ack(int m, int n) {
        if (m == 0)
            return n + 1;

        if (n == 0)
            return ack(m - 1, 1);
```

10. On any nontrivial project, you'll want to profile your code, rather than relying on inspection and guesswork. See Appendix C, on page 291 for how to do that with JRuby.

Charlie Says...

The Default Package

Notice here we're using the Java:: prefix. In this case, it's because our Java-based Ackermann class is in the *default* package. Such classes can be accessed immediately under the Java namespace.

```
        return ack(m - 1, ack(m, n - 1));
    }
}
```

...which you can then compile like so:

```
$ javac Ackermann.java
```

We need to make only one change to the Ruby code to use the new Java class. In the middle of the button's on_click handler, add the text Java:: to the beginning of the Ackermann.ack call, like this:

```
@result.text = Java::Ackermann.ack(@m.text.to_i,
                                   @n.text.to_i).to_s
```

When you rerun the program and click the button, the result should appear immediately. Now that we've seen examples of the most common ways people use JRuby, let's look at each step of the process in more detail.

2.2 Dealing with the Classpath

Before you can use that piece of external library wizardry, you have to *find* it. When you bring Java code into your app, you're playing by Java's rules. Rubyists are used to saying require 'some_file_name' and counting on the file to show up inside one of Ruby's search paths. By contrast, Java looks for each class by its fully specified package name; the physical location of the file isn't as important.

For readers coming from the Ruby world, the *Classpath* is the list of directories and .jar files where Java (and therefore JRuby) will look for external libraries. If you're doing a java_import (see Section 2.3, *By*

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Home Page for Using JRuby
http://pragprog.com/titles/jruby
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/jruby.

Contact Us

Online Orders:	www.pragprog.com/catalog
Customer Service:	support@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com
Contact us:	1-800-699-PROG (+1 919 847 3884)