10

JRUBY: RUBY ON THE JVM

In Chapters 1 through 9 we learned how the standard version of Ruby works internally. Because Ruby is written in C, its standard implementation is often known as *CRuby*. It's also often referred to as *Matz's Ruby Interpreter (MRI)*, after Yukihiro Matsumoto, who created the language in the early 1990s.

In this chapter we'll see an alternative implementation of Ruby called *JRuby*. JRuby is Ruby implemented in Java instead of C. The use of Java allows Ruby applications to run like any other Java program, using the Java Virtual Machine (JVM). It also allows your Ruby code to interoperate with thousands of libraries written in Java and other languages that run on the JVM. Thanks to the JVM's sophisticated garbage collection (GC) algorithms, just-in-time (JIT) compiler, and many other technical innovations, using the JVM means that your Ruby code often runs faster and more reliably.

In the first half of this chapter, we'll contrast standard Ruby—that is, MRI—with JRuby. You'll learn what happens when you run a Ruby program using JRuby and how JRuby parses and compiles your Ruby code. In the latter half of the chapter, we'll see how JRuby and MRI save your string data using the `String` class.

ROADMAP

Running Programs with MRI and JRuby

The normal way to run a Ruby program using standard Ruby is to enter `ruby` followed by the name of your Ruby script, as shown in Figure 10-1.

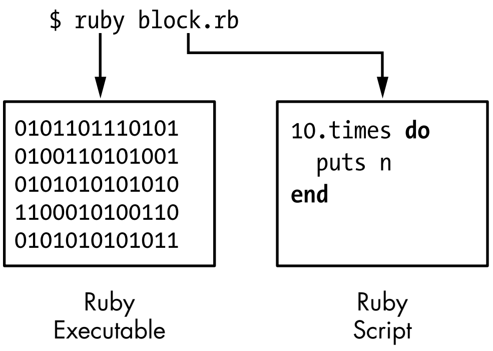
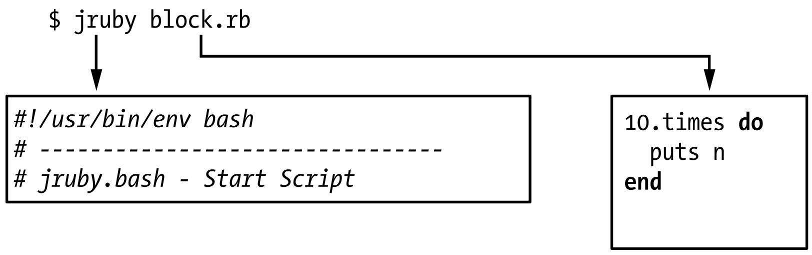


Figure 10-1: Running a script at the command line using standard Ruby

As you can see in the rectangle at the left, entering `ruby` at a terminal prompt launches a binary executable, the product of compiling Ruby's C source code during the Ruby build process. On the right, you see that the command line parameter to the `ruby` command is a text file containing your Ruby code.

To run your Ruby script using JRuby, you normally enter `jruby` at your terminal prompt. (Depending on how you installed JRuby, the standard `ruby` command might be remapped to launch JRuby.) Figure 10-2 shows how this command works at a high level.



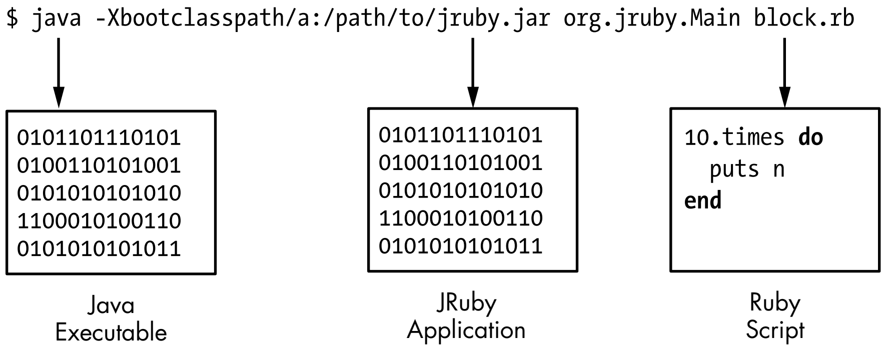
```
$ jruby block.rb
```

```
#!/usr/bin/env bash
# -------------------------------
# jruby.bash - Start Script
```

```
10.times do
  puts n
end
```

Figure 10-2: The `jruby` command actually maps to a shell script.

Unlike the `ruby` command, the `jruby` command doesn't map to a binary executable. It refers to a shell script that executes the `java` command. Figure 10-3 shows a simplified view of the command JRuby uses to launch Java.



```
$ java -Xbootclasspath/a:/path/to/jruby.jar org.jruby.Main block.rb
```

```
0101101110101
0100110101001
0101010101010
1100010100110
0101010101011
```

```
0101101110101
0100110101001
0101010101010
1100010100110
0101010101011
```

```
10.times do
  puts n
end
```

Figure 10-3: A simplified version of the command JRuby uses to launch the JVM

Notice in Figure 10-3 that JRuby executes your Ruby script using a binary executable known as the *Java Virtual Machine (JVM)*. Like the standard Ruby executable, the JVM is written in C and compiled into a binary executable. The JVM runs Java applications, while MRI runs Ruby applications.

Notice, too, that in the center of Figure 10-3 one of the parameters to the `java` program, *-Xbootclasspath*, specifies an additional library, or collection, of compiled Java code to make available to the new program: *jruby.jar*. The JRuby Java application is contained inside *jruby.jar*. Finally, on the right, you see the text file containing your Ruby code again.

In sum, here's what happens when standard Ruby and JRuby launch your Ruby programs:

- When you run a Ruby script using MRI, you launch a binary executable, originally written in C, that directly compiles and executes your Ruby script. This is the standard version of Ruby.
- When you run a Ruby script using JRuby, you launch a binary executable, the JVM, which executes the JRuby Java application. This Java application, in turn, parses, compiles, and executes your Ruby script while running inside the JVM.

How JRuby Parses and Compiles Your Code

Once you launch JRuby, it needs to parse and compile your code. To do this, it uses a parser generator, just as MRI does. Figure 10-4 shows a high-level overview of the JRuby parsing and compiling process.

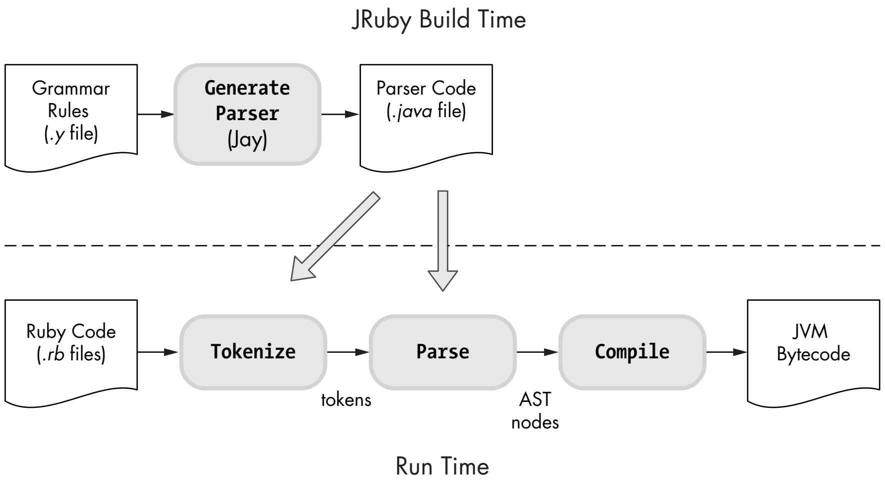


Figure 10-4: JRuby uses a parser generator called Jay.

Just as MRI uses Bison, JRuby uses a parser generator called *Jay* during the JRuby build process to create the code that will parse your Ruby code. Jay is very similar to Bison, except that it's written in Java instead of C. At run time, JRuby tokenizes and parses your Ruby code using the generated parser. As with MRI, this process produces an abstract syntax tree (AST).

Once JRuby parses your code and produces an AST, it compiles your code. However, instead of producing YARV instructions as MRI does, JRuby produces a series of instructions, known as *Java bytecode* instructions, that the JVM can execute. Figure 10-5 shows a high-level comparison of how MRI and JRuby process your Ruby code.

The left side of the figure shows how your Ruby code changes when you execute it with MRI. MRI converts your code into tokens, then into AST nodes, and finally into YARV instructions. The *Interpret* arrow indicates that the MRI executable reads the YARV instructions and interprets, or executes, them. (You don't write the C or machine language code; that work is done for you.)

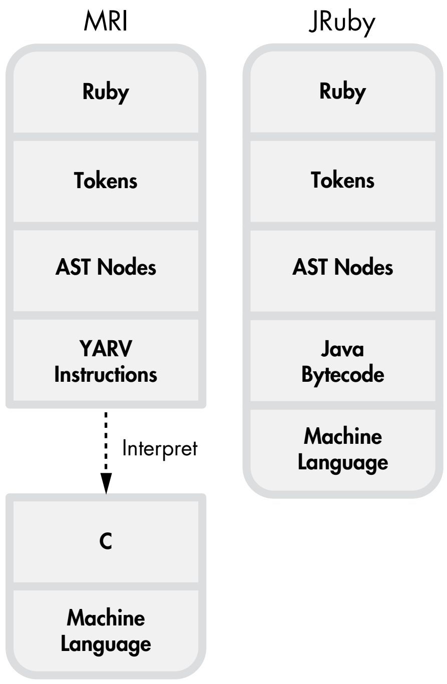


Figure 10-5: The different forms your Ruby code takes inside MRI (left) and JRuby (right)

The high-level overview at the right side of the figure shows how JRuby handles your Ruby code internally. The boxes in the one large rectangle show the different forms your code takes as JRuby executes it. You can see that, like MRI, JRuby first converts your code into tokens and later into AST nodes. But then MRI and JRuby diverge: JRuby compiles the AST nodes into Java bytecode instructions, which the JVM can execute. In addition, the JVM can convert the Java bytecode into machine language using a JIT compiler, which speeds up your program even more because executing machine language is faster than executing Java bytecode. (We'll look at the JIT compiler in more detail in Experiment 10-1.)

How JRuby Executes Your Code

We've seen that JRuby tokenizes and parses your code almost the same way that MRI does. And just as MRI Ruby 1.9 and 2.0 compile your code into YARV instructions, JRuby compiles it into Java bytecode instructions.

But that's where the similarity ends: MRI and JRuby use two very different virtual machines to execute your code. Standard Ruby uses YARV, but JRuby uses the JVM to execute your program.

The whole point of building a Ruby interpreter with Java is to be able to execute Ruby programs using the JVM. The ability to use the JVM is important for two reasons:

Environmental The JVM allows you to use Ruby on servers, in applications, and in IT organizations where previously you could not run Ruby at all.

Technical The JVM is the product of almost 20 years of intense research and development. It contains sophisticated solutions for many difficult computer science problems, like garbage collection and multithreading. Ruby can often run faster and more reliably on the JVM.

To get a better sense of how this works, let's see how JRuby executes the simple Ruby script *simple.rb* in Listing 10-1.

```
puts 2+2
```

Listing 10-1: A one-line Ruby program (simple.rb)

First, JRuby tokenizes and parses this Ruby code into an AST node structure. Next, it iterates through the AST nodes and converts your Ruby into Java bytecode. Use the `--bytecode` option, as shown in Listing 10-2, to see this bytecode for yourself.

```
$ jruby --bytecode simple.rb
```

Listing 10-2: JRuby's `--bytecode` *option displays the Java bytecode your Ruby code is compiled into.*

As the output of this command is complex, I won't dig into it here, but Figure 10-6 summarizes how JRuby compiles and executes this script.

At the left of this figure, you see the code `puts 2+2`. The large downward pointing arrow indicates that JRuby converts this code into a series of Java bytecode instructions that implement a Java class called `simple` (after the script's filename). The `class simple extends AbstractScript` notation is Java code; here, it declares a new Java class called `simple`, which uses `AbstractScript` as a superclass.

The `simple` class is a Java version of our Ruby code that adds 2 + 2 and prints the sum. The `simple` Java class does the same thing using Java. Inside `simple`, JRuby creates a Java method called `__file__` that executes the `2+2` code as indicated with the inner `__file__` rectangle at the bottom of the figure. The method rectangle `<init>` is the constructor for the `simple` class.

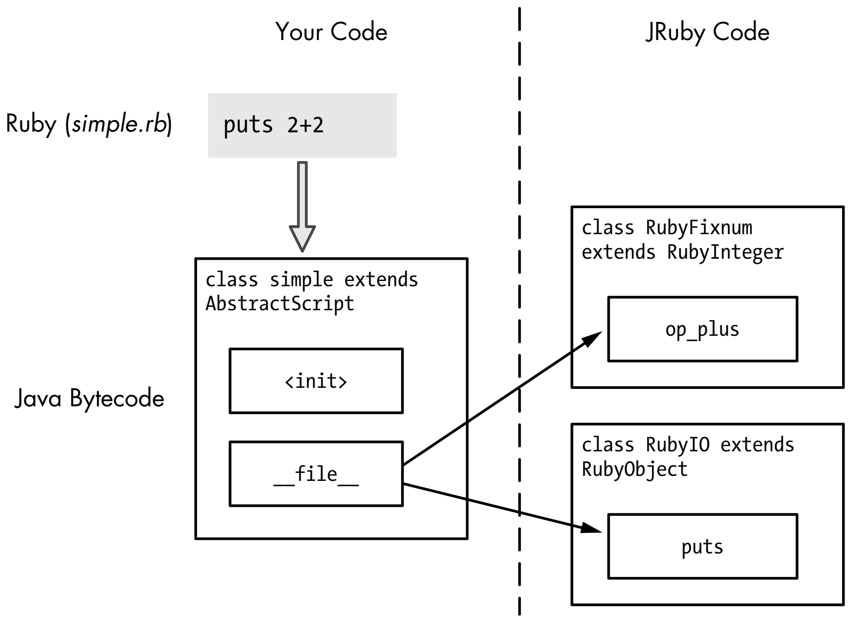

Your Code
JRuby Code
Ruby (*simple.rb*)
puts 2+2
Java Bytecode
class simple extends AbstractScript
<init>
__file__
class RubyFixnum extends RubyInteger
op_plus
class RubyIO extends RubyObject
puts

Figure 10-6: JRuby converts your Ruby code into Java classes.

At the right of Figure 10-6, you see a small part of JRuby's library of Ruby classes. These are Ruby's built-in classes, such as `Fixnum`, `String`, and `Array`. MRI implements these classes using C. When your code calls a method from one of these classes, the method dispatch process uses the CFUNC method type. However, JRuby implements all of the built-in Ruby classes using Java code. On the right side of Figure 10-6, you see two built-in Ruby methods that our code calls.

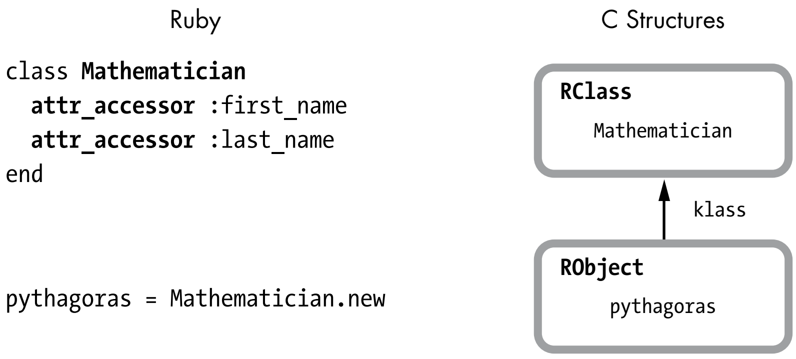
- First, your code adds 2 + 2, using the `+` method of the Ruby `Fixnum` class. JRuby implements the Ruby `Fixnum` class using a Java class called `RubyFixnum`. In this example, your code calls the `op_plus` Java method in this `RubyFixnum` class.
- To print the sum, the code calls the `puts` method of the built-in Ruby `IO` class (actually via the `Kernel` module). JRuby implements this in a similar way, using a Java class called `RubyIO`.

Implementing Ruby Classes with Java Classes

As you know, standard Ruby is implemented internally using C, which doesn't support the notion of object-oriented programming. C code can't use objects, classes, methods, or inheritance the way that Ruby code does.

However, JRuby is implemented in Java, an object-oriented programming language. While not as flexible and powerful as Ruby itself, Java does support writing classes, creating objects as instances of those classes, and relating one class to another through inheritance, which means that JRuby's implementation of Ruby is also object oriented.

JRuby implements Ruby objects with Java objects. To get a better idea of what this means, see Figure 10-7, which compares Ruby code with MRI C structures.



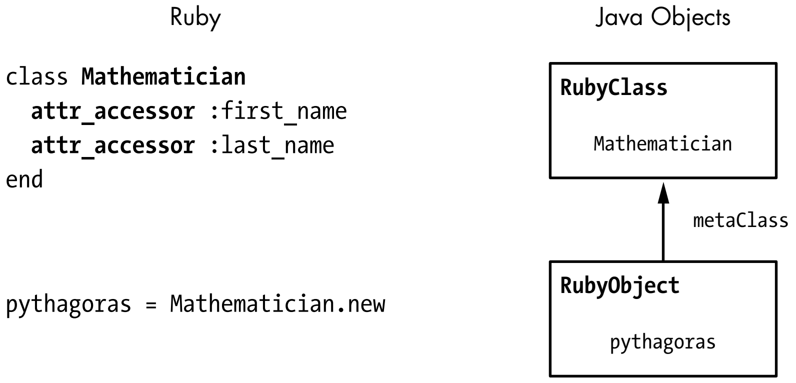
```
class Mathematician
  attr_accessor :first_name
  attr_accessor :last_name
end

pythagoras = Mathematician.new
```

Figure 10-7: MRI implements objects and classes using C structures.

Internally Ruby creates an `RClass` C structure for each class and an `RObject` structure for each object. Ruby tracks the class for each object using the `klass` pointer in the `RObject` structure. Figure 10-7 shows one `RClass` for the `Mathematician` class and one `RObject` for `pythagoras`, an instance of `Mathematician`.

Figure 10-8 shows that the situation is very similar in JRuby, at least at first glance.



```
class Mathematician
  attr_accessor :first_name
  attr_accessor :last_name
end

pythagoras = Mathematician.new
```

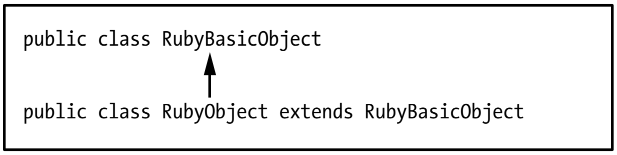
Figure 10-8: Internally, JRuby represents objects using the `RubyObject` Java class and classes using the `RubyClass` Java class.

On the left side of the figure, we see the same Ruby code. On the right are two Java objects, one an instance of the `RubyObject` Java class and the other an instance of the `RubyClass` Java class. JRuby's implementation of Ruby objects and classes closely resembles MRI's, but JRuby uses Java objects instead of using C structures. JRuby uses the names `RubyObject` and `RubyClass` because these Java objects represent your Ruby object and class.

But when we look a bit closer, things aren't so straightforward. Because `RubyObject` is a Java class, JRuby can use inheritance to simplify its internal implementation. In fact, the superclass of `RubyObject` is `RubyBasicObject`. This reflects how the Ruby classes are related, as we can see by calling the `ancestors` method on `Object`.

```
p Object.ancestors
 => [Object, Kernel, BasicObject]
```

Calling `ancestors` returns an array containing all the classes and modules in the superclass chain for the receiver. Here, we see that `Object`'s superclass is the `Kernel` module and its superclass is `BasicObject`. JRuby uses the same pattern for its internal Java class hierarchy, as shown in Figure 10-9.



```
public class RubyBasicObject

public class RubyObject extends RubyBasicObject
```

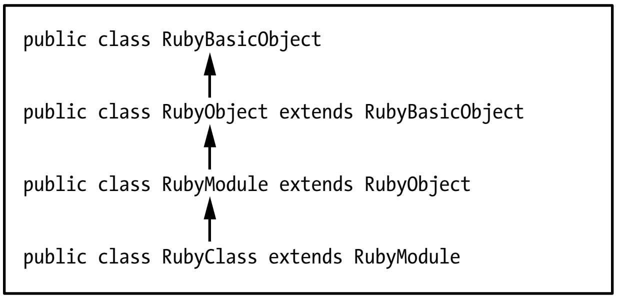
Figure 10-9: `RubyBasicObject` is the superclass of the `RubyObject` Java class.

The `Kernel` module aside, we can see that JRuby's internal Java class hierarchy reflects the Ruby class hierarchy that it implements. This similarity is made possible by Java's object-oriented design.

Now for a second example. Let's use `ancestors` again to show the superclasses for the `Class` Ruby class.

```
p Class.ancestors
 => [Class, Module, Object, Kernel, BasicObject]
```

Here, we see that the superclass of `Class` is `Module`, its superclass is `Object`, and so on. And as we would expect, JRuby's Java code uses the same design internally (see Figure 10-10).



```
public class RubyBasicObject

public class RubyObject extends RubyBasicObject

public class RubyModule extends RubyObject

public class RubyClass extends RubyModule
```

Figure 10-10: JRuby's internal Java class hierarchy for `RubyClass`

Experiment 10-1: Monitoring JRuby's Just-in-Time Compiler

I mentioned earlier that JRuby can speed up your Ruby code by using a JIT compiler. JRuby always translates your Ruby program into Java bytecode instructions, which the JVM can compile into machine language that your computer's microprocessor can execute directly. In this experiment we'll see when this happens and measure how much it speeds up your code.

Experiment Code

Listing 10-3 shows a Ruby program that prints out 10 random numbers between 1 and 100.

```
❶ array = (1..100).to_a
❷ 10.times do
❸   sample = array.sample
    puts sample
  end
```

Listing 10-3: A sample program for testing JRuby's JIT behavior (jit.rb)

At ❶ we create an array with 100 elements: 1 through 100. Then, at ❷ we iterate over the following block 10 times. Inside this block, we use the `sample` method at ❸ to pick a random value from the array and print it. When we run this code, we get the output shown in Listing 10-4.

```
$ jruby jit.rb
87
88
69
5
38
--snip--
```

Listing 10-4: The output from Listing 10-3

Now let's remove the `puts` statement and increase the number of iterations. (Removing the output will make the experiment more manageable.) Listing 10-5 shows the updated program.

```
  array = (1..100).to_a
  1000.times do
❶   sample = array.sample
  end
```

Listing 10-5: We remove `puts` and increase the number of iterations to 1,000.

Using the -J-XX:+PrintCompilation Option

Of course, if we run the program now, we won't see any output because we've removed `puts`. Let's run the program again—this time using a debug flag (shown in Listing 10-6) to display information about what the JVM's JIT compiler is doing.

```
$ jruby -J-XX:+PrintCompilation jit.rb
    101   1       java.lang.String::hashCode (64 bytes)
    144   2       java.util.Properties$LineReader::readLine (452 bytes)
    173   3       sun.nio.cs.UTF_8$Decoder::decodeArrayLoop (553 bytes)
    200   4       java.lang.String::charAt (33 bytes)
--snip--
```

Listing 10-6: The output generated by the `-J-XX:+PrintCompilation` option

Here, we use the `-J` option for JRuby and pass the `XX:+PrintCompilation` option to the underlying JVM application. `PrintCompilation` causes the JVM to display the information you see in Listing 10-6. The line `java.lang.String::hashCode` means that the JVM compiled the `hashCode` method of the `String` Java class into machine language. The other values show technical information about the JIT process (`101` is a time stamp, `1` is a compilation ID, and `64 bytes` is the size of the bytecode snippet that was compiled).

The goal of this experiment is to validate the hypothesis that Listing 10-5 should run faster once the JVM's JIT compiler converts it into machine language. Notice that Listing 10-5 has just one line of Ruby code inside the loop at ❶ that calls `array.sample`. Therefore, we should expect our Ruby program to finish noticeably faster once the JIT compiles JRuby's implementation of `Array#sample` into machine language because `Array#sample` is called so many times.

Because the output in Listing 10-6 is quite long and complex, we'll use `grep` to search the output for occurrences of `org.jruby.RubyArray`.

```
$ jruby -J-XX:+PrintCompilation jit.rb | grep org.jruby.RubyArray
```

The result is no output. None of the lines in the `PrintCompilation` output match the name `org.jruby.RubyArray`, which means the JIT compiler is not converting the `Array#sample` method into machine language. It doesn't do this conversion because the JVM only runs the JIT compiler to compile Java bytecode instructions that your program executes numerous times—areas of bytecode instructions known as *hot spots*. The JVM spends extra time compiling hot spots because they are called so many times. To prove this, we can increase the number of iterations to 100,000 and repeat our test, as shown in Listing 10-7.

```
array = (1..100).to_a
100000.times do
```

```
  sample = array.sample
end
```

Listing 10-7: Increasing the number of iterations should trigger the JIT compiler to convert `Array#sample` to machine language.

When we repeat the same `jruby` command again with `grep`, we see the output shown in Listing 10-8.

```
❶ $ jruby -J-XX:+PrintCompilation jit.rb | grep org.jruby.RubyArray
     1809 165       org.jruby.RubyArray::safeArrayRef (11 bytes)
     1810 166  !    org.jruby.RubyArray::safeArrayRef (12 bytes)
     1811 167       org.jruby.RubyArray::eltOk (16 bytes)
     1927 203       org.jruby.RubyArray$INVOKER$i$0$2$sample::call (36 bytes)
❷    1928 204  !    org.jruby.RubyArray::sample (834 bytes)
     1930 205       org.jruby.RubyArray::randomReal (10 bytes)
```

Listing 10-8: The output after running Listing 10-7 with `-J-XX:+PrintCompilation` piped to `grep`

Because we used `grep org.jruby.RubyArray` at ❶, we see only Java class names that match the text `org.jruby.RubyArray`. At ❷ we can see that the JIT compiler compiled the `Array#sample` method because we see the text `org.jruby.RubyArray::sample`.

Does JIT Speed Up Your JRuby Program?

Now to see if the JIT sped things up. Based on a command-line parameter—`ARGV[0]`—which I save in `iterations` at ❶, Listing 10-9 measures the amount of time it takes to call `Array#sample` a given number of times.

```
require 'benchmark'

❶ iterations = ARGV[0].to_i

Benchmark.bm do |bench|
  array = (1..100).to_a
  bench.report("#{iterations} iterations") do
    iterations.times do
      sample = array.sample
    end
  end
end
```

Listing 10-9: Sample code for benchmarking JIT performance

By running this listing as shown below, we can measure how long it takes to execute the loop 100 times, for example.

```
$ jruby jit.rb 100
```

Figure 10-11 shows the results for 100 to 100 million iterations using both JRuby and MRI.

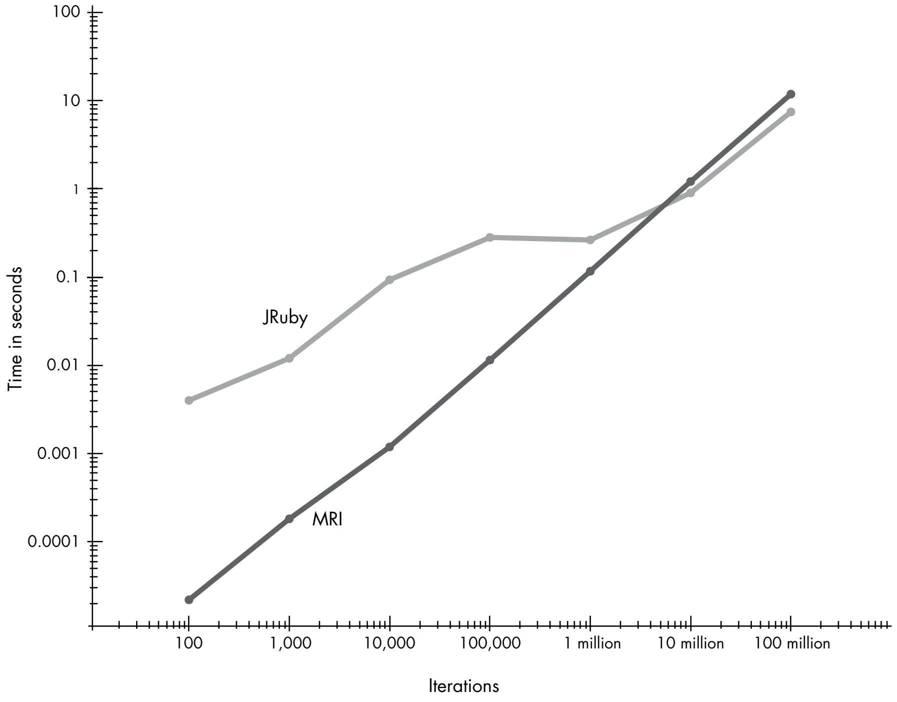


Figure 10-11: JRuby vs. MRI performance. Time is shown in seconds vs. number of iterations (using JRuby 1.7.5 and Java 1.6; MRI Ruby 2.0).

The graph for MRI is more or less a straight line moving up to the right. This means it always takes Ruby 2.0 about the same amount of time to execute the `Array#sample` method. The results for JRuby, however, are not so simple. At left you can see that for fewer than 100,000 iterations, JRuby takes longer to execute Listing 10-9. (The chart uses a logarithmic scale, so the absolute time differences on the left side are small.) However, once we reach about 1 million iterations, JRuby speeds up dramatically and starts to take less time to execute `Array#sample`.

Ultimately for many, many iterations, JRuby is faster than MRI. But what's important here is not simply that JRuby might be faster but that its performance characteristics vary. The longer your code runs, the longer the JVM has to optimize it, and the faster things will be.

Strings in JRuby and MRI

We've learned how JRuby executes bytecode instructions, passing control between your code and a library of Ruby objects implemented with Java. Now we'll take a closer look at this library, specifically at how JRuby implements the `String` class. How do JRuby and MRI implement strings? Where do they save the string data you use in your Ruby code, and how do their implementations compare? Let's begin to answer these questions by looking at how MRI implements strings.

How JRuby and MRI Save String Data

This code saves a famous quote from Pythagoras in a local variable. But where does this string go?

```
str = "Geometry is knowledge of the eternally existent."
```

Recall from Chapter 5 that MRI uses different C structures to implement built-in classes, such as `RRegexp`, `RArray`, and `RHash`, as well as `RString`, which saves your strings. Figure 10-12 shows how MRI represents the `Geometry...` string internally.

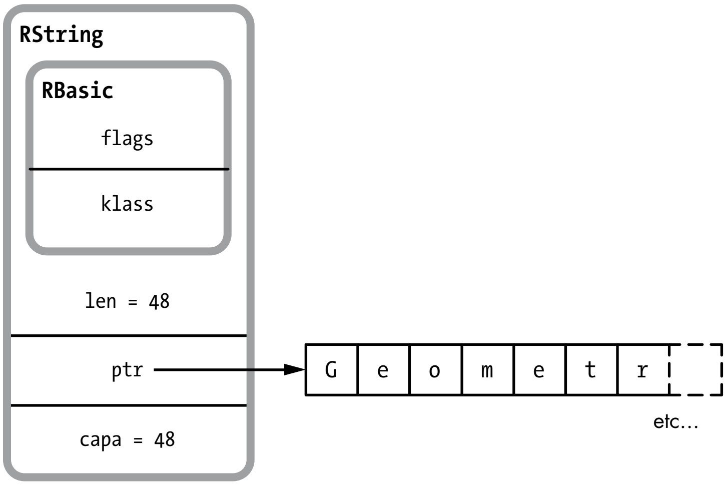


Figure 10-12: Part of the `RString` C structure

Notice that MRI saves the actual string data in a separate buffer, or section of memory, shown on the right. The `RString` structure itself contains a pointer to this buffer, `ptr`. Also notice that `RString` contains two other integer values: `len`, or the length of the string (48 in this example), and `capa`, or the capacity of the data buffer (also 48). The size of the data buffer can be longer than the string, in which case `capa` would be larger than `len`. (This would be the case if you executed code that reduced the length of the string.)

Now let's consider JRuby. Figure 10-13 shows how JRuby represents this string internally. JRuby uses the Java class `RubyString` to represent strings in your Ruby code, following the naming pattern we saw above with `RubyObject` and `RubyClass`. `RubyString` uses another class to track the actual string data: `ByteList`. This lower-level code tracks a separate data buffer (called `bytes`) similar to the way that the `RString` structure does so in MRI. `ByteList` also stores the length of the string in the `realSize` instance variable.

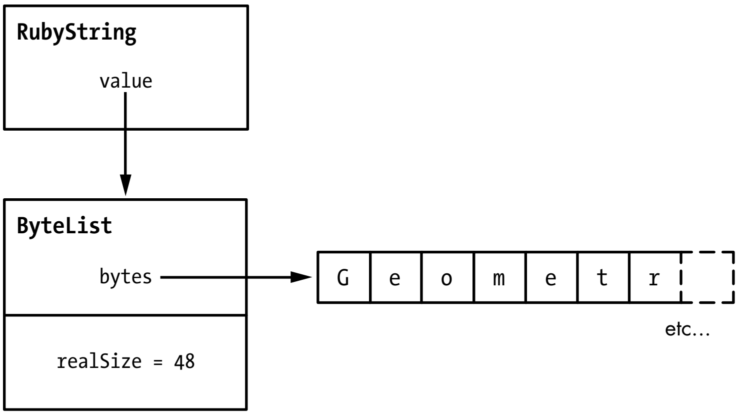


Figure 10-13: JRuby uses two Java objects and a data buffer for each string.

Copy-on-Write

Internally, both JRuby and MRI use an optimization called *copy-on-write* for strings and other data. This trick allows two identical string values to share the same data buffer, which saves both memory and time because Ruby avoids making separate copies of the same string data unnecessarily.

For example, suppose we use the `dup` method to copy a string.

```
str = "Geometry is knowledge of the eternally existent."
str2 = str.dup
```

Does JRuby have to copy the `Geometry is...` text from one string object to another? No. Figure 10-14 shows how JRuby shares the string data across two different string objects.

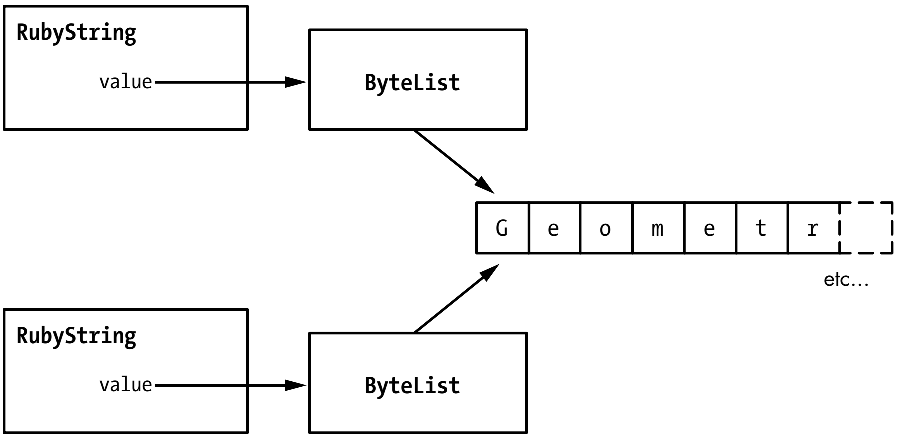


Figure 10-14: Two JRuby string objects can share the same data buffer.

When we call dup, JRuby creates new RubyString and ByteList Java objects, but it doesn't copy the actual string data. Instead, it sets the second ByteList object to point to the same data buffer used by the original string. Now we have two sets of Java objects but only one underlying string value, as shown on the right of the figure. Because strings can contain thousands of bytes or more, this optimization can often save a tremendous amount of memory.

MRI uses the same trick, although in a slightly more complex way. Figure 10-15 shows how standard Ruby shares strings.

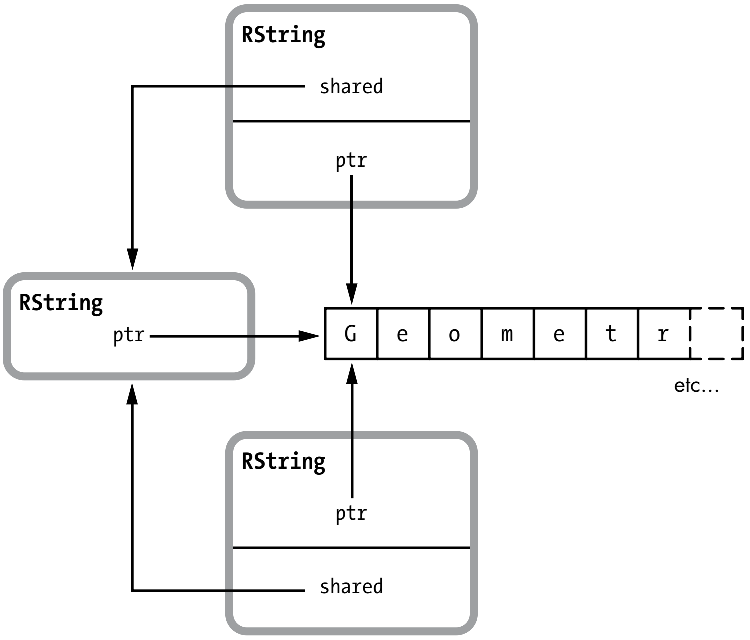


Figure 10-15: MRI shares strings by creating a third RString *structure.*

Like JRuby, MRI shares the underlying string data. However, when you copy a string in standard MRI Ruby, it creates a third RString structure and then sets both the original RString and new RString to refer to it using the shared pointer.

In either case, we have a problem. What if we change one of the string variables? For example, suppose we convert one of the strings to uppercase as follows:

```
  str = "Geometry is knowledge of the eternally existent."
❶ str2 = str.dup
❷ str2.upcase!
```

At ❶ in both JRuby and MRI, we have two shared strings, but at ❷ I change the second string using the upcase! method. Now the two strings differ, which means that Ruby clearly can't continue to share the underlying string buffer or upcase! would change both strings. We can see the strings are now different by displaying the string values.

```
p str
 => "Geometry is knowledge of the eternally existent."
p str2
 => "GEOMETRY IS KNOWLEDGE OF THE ETERNALLY EXISTENT."
```

At some point, Ruby must have separated these two strings, creating a new data buffer. This is what the phrase *copy-on-write* means: Both MRI and JRuby create a new copy of the string data buffer when you write to one of the strings.

Experiment 10-2: Measuring Copy-on-Write Performance

In this experiment we'll collect evidence that this extra copy operation really occurs when we write to a shared string. First, we'll create a simple, nonshared string and write to it. Then we'll create two shared strings and write to one of them. If copy-on-write really occurs, then writing to a shared string should take a bit longer because Ruby has to create a new copy of the string before writing.

Creating a Unique, Nonshared String

Let's begin by creating our example string again, str. Initially Ruby can't possibly share str with anything else because there is only one string. We'll use str for our baseline performance measurement.

```
str = "Geometry is knowledge of the eternally existent."
```

But as it turns out, Ruby shares str immediately! To see why, we'll examine the YARV instructions that MRI uses to execute this code, as shown in Listing 10-10.

```
  code = <<END
  str = "Geometry is knowledge of the eternally existent."
  END

  puts RubyVM::InstructionSequence.compile(code).disasm
  == disasm: <RubyVM::InstructionSequence:<compiled>@<compiled>>==========
  local table (size: 2, argc: 0 [opts: 0, rest: -1, post: 0, block: -1] s1)
❶ [ 2] str
  0000 trace            1                                             (   1)
❷ 0002 putstring        "Geometry is knowledge of the eternally existent."
❸ 0004 dup
❹ 0005 setlocal_OP__WC__0 2
  0007 leave
```

Listing 10-10: MRI Ruby uses a dup YARV instruction internally when you use a literal string constant.

Reading the YARV instructions above carefully, we see at ❷ that Ruby puts the string onto the stack using `putstring`. This YARV instruction internally copies the string argument to the stack, creating a shared copy already. At ❸ Ruby uses `dup` to create yet another shared copy of the string to use as an argument for `setlocal`. Finally, at ❹ `setlocal_OP__WC__0 2` saves this string into the `str` variable, shown as `[2]` in the local table ❶.

Figure 10-16 summarizes this process.

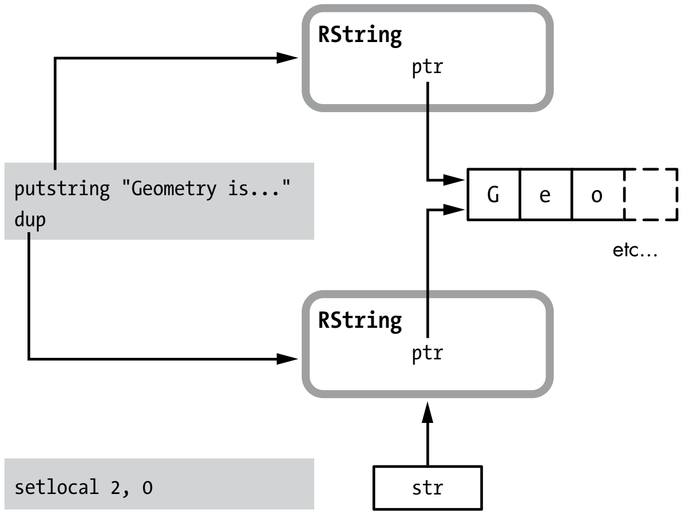


Figure 10-16: Executing `putstring` and `dup` creates shared strings in MRI.

On the left are the YARV instructions `putstring`, `dup`, and `setlocal`. On the right are the `RString` structures that these instructions create, as well as the underlying shared string data. As I just mentioned, `putstring` in fact copies the string constant from a third `RString` left off the diagram, meaning the string is actually shared a third time.

Because Ruby initially shares strings created from constant values, we need to create our string differently by concatenating two strings together as follows:

```
str = "This string is not shared" + " and so can be modified faster."
```

The result of this concatenation will be a new, unique string. Ruby will not share its string data with any other string objects.

Experiment Code

Let's take some measurements. Listing 10-11 shows the code for this experiment.

```
require 'benchmark'

ITERATIONS = 1000000

Benchmark.bm do |bench|
  bench.report("test") do
    ITERATIONS.times do
❶     str = "This string is not shared" + " and so can be modified faster."
❷     str2 = "But this string is shared" + " so Ruby will need to copy it
              before writing to it."
❸     str3 = str2.dup
❹     str3[3] = 'x'
    end
  end
end
```

Listing 10-11: Measuring a delay for copy-on-write

Before we run this test, let's walk through this code. At ❶ we create a unique, unshared string by concatenating two strings. This is `str`. Then at ❷ we create a second unshared string, `str2`. But at ❸ we use `dup` to create a copy of this string, `str3`, and now `str2` and `str3` share the same value.

Visualizing Copy-on-Write

At ❹ in Listing 10-11 we change the fourth character in `str3` using the code `str3[3] = 'x'`. But here Ruby can't change the character in `str3` without changing `str2` as well, as shown in Figure 10-17.

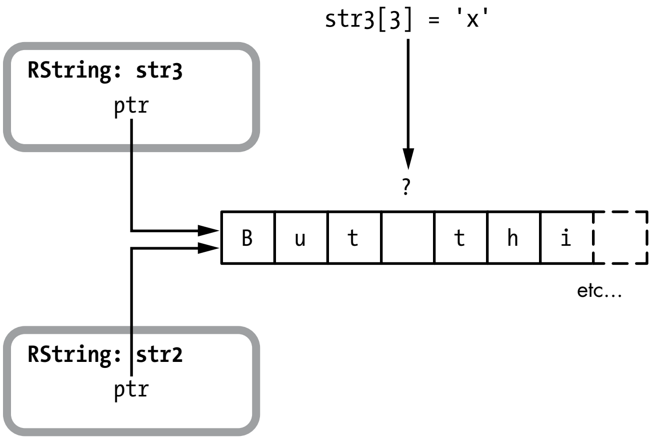


Figure 10-17: Ruby can't change `str3` without also changing `str2`.

Ruby has to make a separate copy of str3 first, as shown in Figure 10-18.

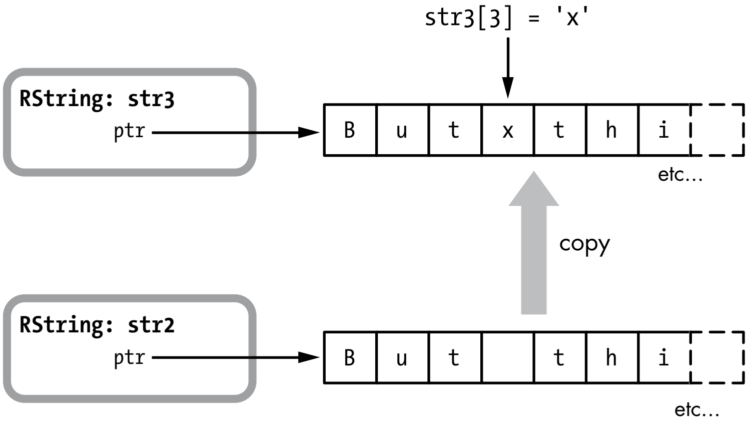


Figure 10-18: Ruby copies the string into a new buffer for str3 *before writing to it.*

Now Ruby can write into the new buffer for str3 without affecting str2.

Modifying a Shared String Is Slower

When we execute Listing 10-11, the benchmark library measures how long it takes to run the inner block 1 million times. This block creates str, str2, and str3 and then modifies str3. On my laptop, benchmark yields a measurement of about 1.87 seconds.

Next, let's change str3[3] = 'x' at ❹ to modify str instead.

```
#str3[3] = 'x'
str[3] = 'x'
```

Now we're modifying the unshared, unique string instead of the shared string. Running the test again yields a result of about 1.69 seconds, or about 9.5 percent less than the time benchmark reported for the shared string. As expected, it takes slightly less time to modify a unique string than it does to modify a shared one.

The graph in Figure 10-19 shows my cumulative results averaged over 10 different observations for both MRI and JRuby. On the left side of the graph are my average measurements for MRI. The bar on the far left represents the time required to modify the shared string, str3, and the right MRI bar shows how long it took to modify the unique string, str. The two bars on the right side exhibit the same pattern for JRuby, but the difference in the height of the bars is much less. Apparently, the JVM can make a new copy of the string faster than MRI.

But there's more: Notice that overall JRuby ran the experiment code in 60 percent less time. That is, it was 2.5 times faster than MRI! Just as in Experiment 10-1, we must be seeing the JVM optimizations, such as JIT compilation, speed up JRuby when compared to MRI.

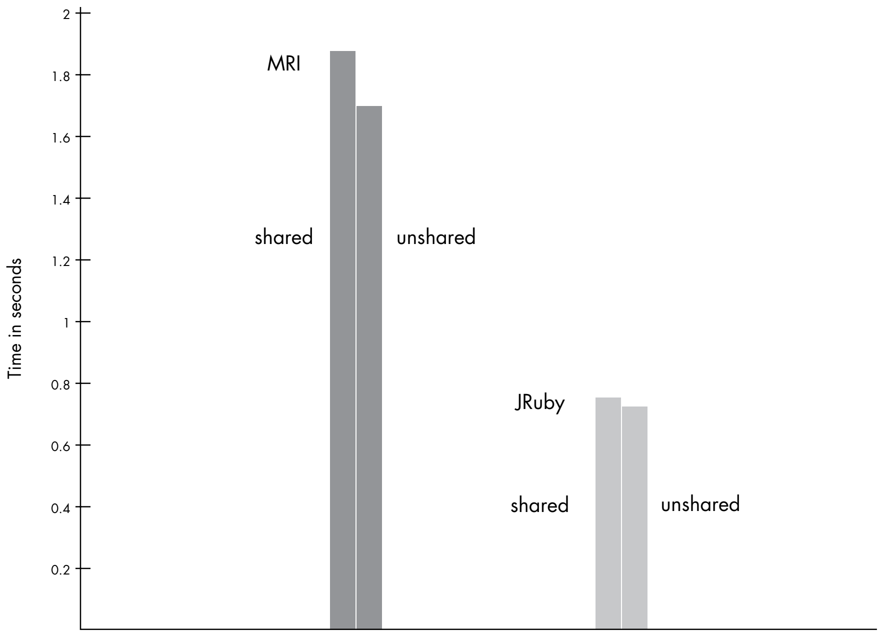


Figure 10-19: Both MRI and JRuby show a delay for copy-on-write (seconds).

Summary

In this chapter we took a look at JRuby, a version of Ruby written in Java. We saw how the `jruby` command launches the JVM, passing *jruby.jar* as a parameter. We explored how JRuby parses and compiles our code, and learned in Experiment 10-1 how the JVM can compile hot spots, or frequently executed snippets of Java bytecode, into machine language. Our results from Experiment 10-1 showed that compiling hot spots dramatically improves performance, allowing JRuby to run even faster than MRI in some cases.

In the second half of this chapter, we learned how MRI and JRuby represent our string data internally. We discovered that both versions of Ruby use copy-on-write optimization, sharing string data between different string objects when possible. Finally, in Experiment 10-2 we proved that copy-on-write actually occurred in both JRuby and MRI.

JRuby is a very powerful and clever implementation of Ruby: By running your Ruby code using the Java platform, you can benefit from the many years of research, development, tuning, and testing that have been invested in the JVM. The JVM is one of the most popular, mature, and powerful software platforms in use today. It's being used not only by Java and JRuby but also by many other software languages, such as Clojure, Scala, and Jython, to name a few. By using this shared platform, JRuby can take advantage of the speed, robustness, and diversity of the Java platform—and it can do this for free!

JRuby is a groundbreaking piece of technology with which every Ruby developer should be familiar.