

JRuby: Ruby for the JVM

Charles Nutter and Thomas Enebo

The JRuby Guys

Sun Microsystems



Except where otherwise noted, the content of this presentation is licensed under the Creative Commons Attribution-Share Alike 3.0 United States License (<http://creativecommons.org/licenses/by-sa/3.0/us/>).

Not Just



JRuby: Ruby for the JVM

Charles Nutter and Thomas Enebo

The JRuby Guys

Sun Microsystems

Agenda

- JRuby background
- JRuby internal design (How does it work?)
 - Basics
 - Parser, Lexer, AST
 - Core Classes
 - Interpreter and Compiler, Performance Optimizations
 - Threading
 - Extensions, POSIX, and Java Integration
- Use cases
- Closing and Q/A

The JRuby Guys

- Charles Oliver Nutter and Thomas Enebo
- Longtime Java developers (10+ yrs each)
- Engineers at Sun Microsystems for 1 yr
- Full-time JRuby developers
- Also working on JVM dynlang support
- Wide range of past experience
 - > C, C++, C#, Perl, Python, Delphi, Lisp, Scheme
 - > Java EE and ME, JINI, WS

JRuby

- Java implementation of Ruby language
- Based originally on Matz's Ruby (MRI) 1.6
- Started in 2002, open source, many contributors
 - > Ola Bini, Nick Sieger, Marcin Mielczynski, Bill Dortch
- Aiming for compatibility with current Ruby version
- Native threading, solid performance
- **New! 1.0.2 release, mostly compat fixes**
- **New! 1.1b1 release, focused on performance**

JRuby Design Overview

- Basics
- Parser
- Core Classes
- Interpreter and Compiler
- Performance Optimizations
- Threading
- Extensions and POSIX support
- Java Integration

JRuby Design: Basics

- jruby/
 - > bin/
 - jruby startup scripts for UNIX and Windows
 - jrubyc, jrubysrv, jrubbycli
 - > lib/
 - ruby
 - 1.8
 - Full copy of Ruby 1.8 stdlib
 - site_ruby
 - RubyGems preinstalled
 - gems
 - RSpec, Rake preinstalled
 - jruby and dependency JAR files

JRuby Design: Basics

- Installation: 1. unpack binary dist; 2. set PATH
- Dependencies: Java 5+ (1.4+ for 1.0)
- jruby.jar contains full runtime
- jruby-complete.jar contains runtime + stdlib
- .rb files can be loaded from within JAR file
 - > entire app + runtime + stdlib in one executable file

JRuby Design: Lexer and Parser

- Hand-written lexer
 - > originally ported from MRI
 - > many changes since then
- LALR parser
 - > Port of MRI's YACC/Bison-based parser
 - We use Jay, a Bison for Java
 - > `DefaultRubyParser.y => DefaultRubyParser.java`
- Abstract Syntax Tree similar to MRI's
 - > we've made a few changes/additions

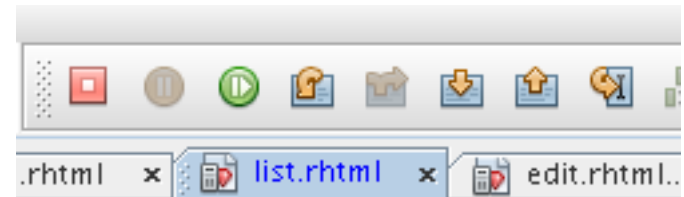
JRuby Parser Enables Tooling

- JRuby's lex/parse/AST used by many Ruby IDEs
 - > NetBeans 6 / NetBeans Ruby IDE
 - > Eclipse RDT/RadRails/Aptana, DLTK, 3rd Rail
 - > IntelliJ
 - > JEdit

```
class Post < ActiveRecord::Migration
  def self.up
    create_table(name, options) ActiveRecord
    create_
  end
end
```

SchemaStatements.create_table(name, opt:

Creates a new table There are two ways to work with it: use the block form or the regular form, like this:



```
6
  <p>
  <table>
  <% @mydocs.each_with_index
  <tr>
    <td>
```

DEMO

NetBeans Ruby IDE

JRuby Design: Core Classes

- Mostly 1:1 core classes to Java types
 - > String is RubyString, Array is RubyArray, etc
- Annotation-based method binding

```
public @interface JRubyMethod {
    String[] name() default {};
    int required() default 0;
    int optional() default 0;
    boolean rest() default false;
    String[] alias() default {};
    boolean meta() default false;
    boolean module() default false;
    boolean frame() default false;
    boolean scope() default false;
    boolean rite() default false;
    Visibility visibility() default
                                Visibility.PUBLIC;
}
...
@JRubyMethod(name = "open", required = 1, frame = true)
```

DEMO

RubyString.java

JRuby Design: Interpreter

- Simple switch-based AST walker
- Recurses for nested structures
- Most code starts out interpreted
 - > command-line scripts compiled immediately
 - > precompiled scripts (.class) instead of .rb
 - > eval'ed code always interpreted (for now)
- Reasonably straightforward code

JRuby Design: Compiler

- New in JRuby 1.1: Full bytecode compilation
 - > 1.0 had a partial (25% complete?) JIT compiler
- AST walker visits code structure
- Bytecode emitter generates Java class+methods
 - > Yes, it's real Java bytecode
 - > AOT mode: 1:1 mapping .rb file to .class file
 - not a “real” Java class...more like a bag of methods
 - ...but it has a “main” for CLI execution
 - > JIT mode: 1:1 mapping method to in-memory class
 - configurable threshold; default is 20 invocations
 - does not JIT evals (yet?)

JRuby Compiler

```
require 'benchmark'
```

```
def fib_ruby(n)  
  if n < 2  
    n  
  else  
    fib_ruby(n - 2) + fib_ruby(n - 1)  
  end  
end
```

```
5.times { puts Benchmark.measure { fib_ruby(30) } }
```

JRuby Compiler

```
~ $ jruby bench_fib_recursive.rb
```

```
Compiling file "bench_fib_recursive.rb" as class  
"bench_fib_recursive"
```

```
~ $ jruby benchmark.rb
```

```
Compiling file "benchmark.rb" as class "benchmark"
```

```
~ $ ls bench*
```

```
bench_fib_recursive.class benchmark.class
```

```
bench_fib_recursive.rb benchmark.rb
```

```
~ $ rm bench*.rb
```

```
~ $ java -server bench_fib_recursive
```

```
...
```

DEMO

JRuby Compiler

JRuby Compiler: Ready for You

- First Ruby compiler for a general-purpose VM
- Fastest 1.8-compatible execution
- AOT mode
 - > Avoids JIT warmup time
 - > Works well with “compile, run” development
 - > Maybe faster startup in future? (a bit slower right now)
- JIT mode
 - > Fits with typical Ruby “just run it” development
 - > Eventually as fast as AOT
 - > You don't have to do anything different

JRuby Design: Perf Optimizations

- Compiler
- ObjectSpace
- Custom core class implementations
- Regular Expressions

Optz #1: Compilation

```
# interpreted mode, compiler disabled
~ $ jruby -J -Djruby.jit.enabled=false -J -server
bench_fib_recursive.rb
2.459000 0.000000 2.459000 ( 2.459000)
2.553000 0.000000 2.553000 ( 2.553000)
2.539000 0.000000 2.539000 ( 2.539000)
2.531000 0.000000 2.531000 ( 2.532000)
2.733000 0.000000 2.733000 ( 2.733000)
```

```
# compiler enabled
~ $ jruby -J -server bench_fib_recursive.rb
0.882000 0.000000 0.882000 ( 0.882000)
0.648000 0.000000 0.648000 ( 0.649000)
0.648000 0.000000 0.648000 ( 0.647000)
0.646000 0.000000 0.646000 ( 0.645000)
0.635000 0.000000 0.635000 ( 0.635000)
```

Compiler Optimizations

- Preallocated, cached literals
- Java opcodes for local flow-control
 - > Explicit local “return” as cheap as implicit
 - > Explicit local “next”, “break”, etc simple Java ops
- Java local variables when possible
 - > Methods and leaf closures
 - leaf == no contained closures
 - > No eval(), binding() calls present
- Monomorphic inline method cache
 - > Polymorphic for 1.1 (probably)

Optz #2: ObjectSpace

- Difficult to support `each_object` on modern VMs
 - > Limited control over GC, memory model
- Only way is to save a weak reference to everything
 - > Double the objects, much more allocation overhead
 - > Perf drops 2-5x
- Very few real-world consumers of `each_object`
 - > test/unit's `each_object(Class)`; we have workaround
 - > `each_object` isn't deterministic; inappropriate for libraries
- JRuby 1.1 disables `each_object` by default
 - > Error if you use it; pass `+O` flag to enable

Optz #3: Custom Core Classes

- String as copy-on-write byte[] impl
- Array as copy-on-write Object[] impl
- Fast-read Hash implementation
- Java “New IO” (NIO) based IO implementation
- Two custom Regexp implementations

Optz #4: Regexp

- Normal Java regex library in Java SE (pre 1.0)
 - > Simplest way to start
 - > However recursive design blew up for large alternations
- JRegex - fast third-party regex library (1.0.x)
 - > Iterative, “compiled” engine
 - > However, char[] based...much encode/decode overhead
- Rej - direct port of MRI regex engine (1.1, maybe)
 - > Not as fast as JRegex, but no encode/decode
 - > Most compatible with MRI (obviously)
- Joni - Java port of Oniguruma (1.1, maybe)
 - > In progress, but could be the holy grail

JRuby Design: Threading

- JRuby supports only native OS threads
 - > Much more heavy than MRI's green threads
 - > But truly parallel, unlike MRI or Ruby 1.9
- Emulates unsafe green operations
 - > Thread#kill, Thread#raise inherently unsafe
 - > Thread#critical impossible to guarantee
 - > All emulated with periodic checkpoints
- Pooling of OS threads minimizes spinup cost
 - > Spinning up threads from pool as cheap as green
 - > Hopefully done for 1.1

JRuby Design: Extensions, POSIX

- Normal Ruby native extensions not supported
 - > Maybe in future, but Ruby API exposes too much
- Native libraries accessible with JNA
 - > Not JNI...JNA = Java Native Access
 - > Programmatically load libs, call functions
 - > Similar to DL in Ruby
 - > Could easily be used for porting extensions
- JNA used for POSIX functions not in Java
 - > Filesystem support (symlinks, stat, chmod, chown, ...)
 - > Process control

JRuby Design: Java Integration

- Java types are presented as Ruby types
 - > Construct instances, call methods, pass objects around
 - > camelCase or under_score_case both work
 - > Most Ruby-calling-Java code looks just like Ruby
- Integration with Java type hierarchy
 - > Implement Java interfaces
 - longhand “include SomeInterface”
 - shorthand “SomeInterface.impl { ... }”
 - closure conversion “add_action_listener { ... }”
 - > Extend Java concrete and abstract Java types
 - Looks, feels like normal Ruby extension

Calling Ruby from Java (Java 6)

```
// One-time load Ruby runtime
ScriptEngineManager factory =
    new ScriptEngineManager();
ScriptEngine engine =
    factory.getEngineByName("jruby");

// Evaluate JRuby code from string.
try {
    engine.eval("puts('Hello')");
} catch (ScriptException exception) {
    exception.printStackTrace();
}
```

Calling Java from Ruby

```
# pull in Java support (require 'java' works too)
include Java
```

```
# import classes you need
import_java java.util. ArrayList
include_class "javax.swing. JFrame"
```

```
# use them like normal Ruby classes
list = ArrayList.new
frame = JFrame.new( "Ruby SWINGS! ")
```

```
# ...but with Ruby features added
list << frame
list.each { |f| f.set_size( 200, 200 ) }
```

Popular Use Case: Swing GUIs

- Swing API is very large, complex
 - > Ruby magic simplifies most of the tricky bits
- Java is a very verbose language
 - > Ruby makes Swing actually fun
- No consistent cross-platform GUI library for Ruby
 - > Swing works everywhere Java does (i.e. everywhere)
- No fire-and-forget execution
 - > No dependencies: any script works on any JRuby install

Swing Option 1: Direct approach

```
import javax.swing. JFrame
import javax.swing. Button

frame = JFrame.new( "Swing is easy now! ")
frame.setSize 300, 300
frame.alwaysOnTop = true

button = Button.new( "Press me! ")
button.addActionListener do |evt|
    evt.source.text = "Don't press me again!"
    evt.source.enabled = false
end

frame.add( button )
frame.show
```

DEMO

Simple Swing

Option 2: Cheri (builder approach)

```
include Cheri :: Swing
```

```
frame = swing.frame("Swing builders!") { |form|  
  size 300, 300  
  box_layout form : Y_AXIS  
  content_pane { background : WHITE }  
  
  button("Event binding is nice") { |btn|  
    on_click { btn.text = "You clicked me!" }  
  }  
}
```

```
frame.visible = true
```

Option 3: Profligacy (targeted fixes)

```
class ProfligacyDemo
  import javax.swing.*
  include Profligacy

  def initialize
    layout = "[<translate][*input][>result]"
    @ui = Swing::LEL.new(JFrame, layout) {|cmps, ints|
      cmps.translate = JButton.new("Translate")
      cmps.input = JTextField.new
      cmps.result = JLabel.new

      translator = proc {|id, evt|
        original = @ui.input.text
        translation = MyTranslator.translate(original)
        @ui.result.text = translation
      }

      ints.translate = {:action => translator}
    }
  end
end
```

Option 4: MonkeyBars (tool-friendly)

- GUI editor friendly (e.g. NetBeans “Matisse”)
- Simple Ruby MVC-based API
- Combines best of both worlds

MonkeyBars + NetBeans Matisse

The screenshot displays the NetBeans IDE interface. On the left, the **Inspector** window shows a tree hierarchy of components for the **Form RssViewer**. The components listed are:

- Other Components
- JFrame
 - jLabel1 [JLabel]
 - feedURL [JTextField]
 - goButton [JButton]
 - JScrollPane1 [JScrollPane]
 - jLabel2 [JLabel]
 - JScrollPane2 [JScrollPane]

The main workspace shows the **Design** view of the **RssViewer.java** form. The form contains a text field labeled **Feed URL** with a **Go** button to its right. Below the text field is a large area labeled **Articles**, which is currently empty. A lightbulb icon and a text box above the design view state: "Inspector window displays a tree hierarchy of components in the opened form."

MonkeyBars Controller

```
class RssController < Monkeybars::Controller
  set_view "RssView"
  set_model "RssModel"

  close_action :exit
  add_listener :type => :mouse,
    :components => ["goButton", "articleList"]

  def go_button_mouse_released(view_state, event)
    model.feed_url = view_state.feed_url
    content = Kernel.open(model.feed_url).read
    @ss = RSS::Parser.parse(content, false)

    model.articles = @ss.items.map {|art| art.title}
    model.article_text =
      CGI.unescapeHTML(@ss.items[0].description)
    update_view
  end
  ...
end
```

Web applications

- Classic Java web dev is too complicated
 - > Modern frameworks follow Rails' lead
- Over-flexible, over-configured
 - > Conventions trump repetition and configuration
- Java is often too verbose for agile work
 - > Ruby makes even raw servlets look easy

Option 1: JRuby on Rails

- Rails works in JRuby for almost a year now
 - > Both 1.2.x and edge/2.0
- ActiveRecord-JDBC for DB
- GoldSpike/Warbler for app server deployment
- GlassFish gem for Mongrel-like deployment

DEMO

JRuby on Rails

Coming Soon: ActiveRecord

```
# define a model (or you can use existing)
class Project
  include ActiveRecord
  with_table_name "PROJECTS" #optional

  #column name is optional
  primary_key_accessor :id, :long, :PROJECT_ID
  attr_accessor :name, :string
  attr_accessor :complexity, :double
end

# connect
ActiveRecord.establish_connection( DB_CONFIG )

# create
project = Project.new( :name => "J Ruby", :complexity => 10 )
project.save
project_id = project.id

# query
all_projects = Project.find( :all )
jruby_project = Project.find( project_id )

# update
jruby_project.complexity = 37
jruby_project.save
```

Option 2: Ruvlets (Ruby servlets)

- Expose Servlets as Ruby API
 - > Because we can!
 - > People keep asking for this....really!
 - > Expose highly tuned web-infrastructure to Ruby
 - > Similar in L&F to Camping
- How it works:
 - i. Evaluates file from load path based on URL
 - ii. File returns an object with a 'service' method defined
 - iii. Object cached for all future requests

Bare Bones Ruvlets

```
class HelloWorld  
  def service(context, request, response)  
    response.content_type = "text/html"  
    response.writer << <<-EOF  
      <html>  
        <head><title>Hello World!</title></head>  
        <body>Hello World!</body>  
      </html>  
    EOF  
  end  
end
```

HelloWorld.new

Servlet-like Ruvlets

```
class HelloWorld2 < HTTPServlet
  def doGet(context, request, response)
    response.contentType = "text/html"
    response.writer << << EOF
      <html>
        <head><title>Hello World! </title></head>
        <body>Hello World! </body>
      </html>
    EOF
  end

  def doPost(context, request, response)
    ...
  end
end
```

HelloWorld2.new

Ruvlets with Meta-Magic

```
class HTTPRuvlet
  def service(context, request, response)
    # HTTP method 'POST' => Ruby method doPost
    method = "do" + request.method.downcase.capitalize

    begin
      # call the method
      __send__ method, context, request, response
    rescue NoMethodError
      context.log "Unimplemented method: #{method}"
      handle_error context, request, response, $!
    end
  end
end

def handle_error(context, request, response, error)
end
end
```

Test and Behavior-driven

- Test-driven development is hard in Java
 - > Ruby strips down tests to simple, readable code
- Write, compile, run cycle plays havoc with tests
 - > Dynamic typing makes it a snap...who needs compilers?
- Testing frameworks don't sync well with specs
 - > Behavior-driven development turns tests into specs

Option 1: test/unit

```
require 'test/unit'
```

```
import java.net. ServerSocket
```

```
class SockTestCase < Test::Unit::TestCase  
  def test_verify_local_port  
    socket = ServerSocket.new(6789)  
    assert_equal(6789, socket.getLocalPort)  
  end  
end
```

Option 2: RSpec

```
import java.net. ServerSocket

describe "ServerSocket" do
  it "should know its own port" do
    server_socket = ServerSocket.new( 5678)
    server_socket.getLocalPort.should == 5678
  end
end
```

Takeaways

- Ruby on the JVM opens many possibilities
 - > Finally making many APIs approachable
 - > Teaching an old dog new tricks
 - > Fits in great with existing libraries and apps
- JRuby is more than just a Ruby implementation
 - > Opening up Ruby to the vast Java world
 - > Enabling a new solutions to existing problems
 - > Pushing Ruby forward
- JRuby needs your help!
 - > JRuby community is the most important contributor

Links

- JRuby: www.jruby.org
- NetBeans: www.netbeans.org
- Ruby: www.ruby-lang.org
- Rails: www.rubyonrails.org
- Cheri: cheri.rubyforge.org
- Profligacy:
ihate.rubyforge.org/profligacy
- MonkeyBars:
monkeybars.rubyforge.org
- ActiveHibernate:
code.google.com/p/activehibernate
- RSpec: rspec.rubyforge.org

JRuby: Ruby for the JVM

**Charles Nutter and Thomas
Enebo**

The JRuby Guys

Sun Microsystems