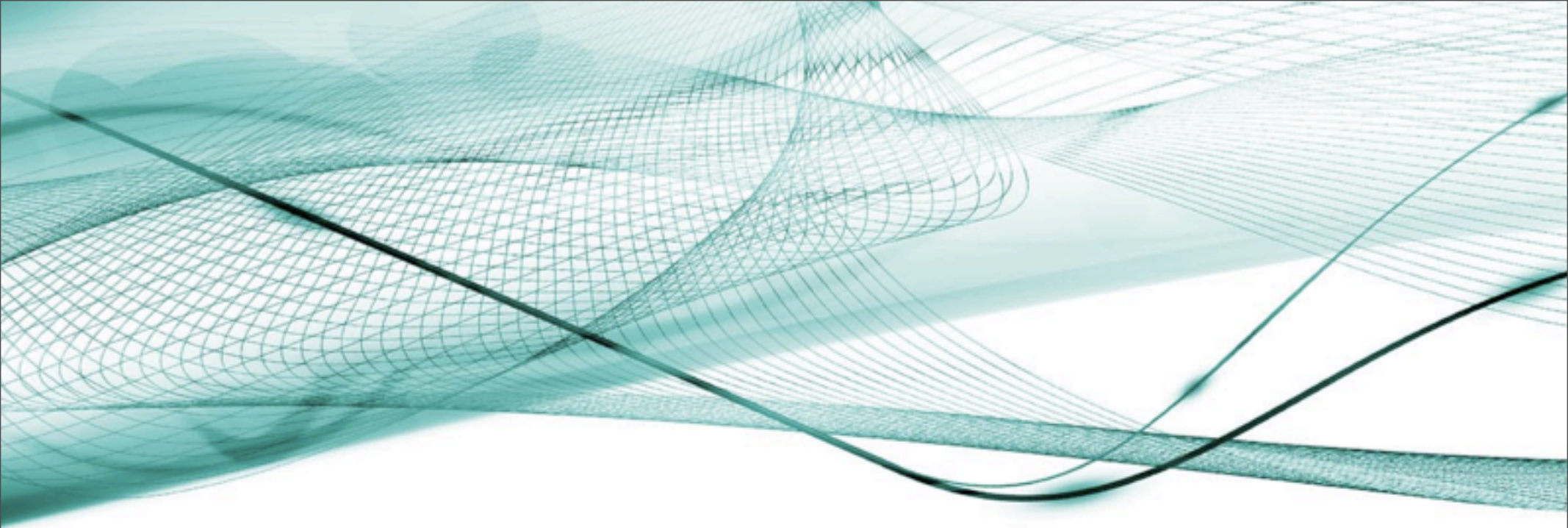


JRuby
Power on the JVM

Ola Bini
JRuby Core Developer
ThoughtWorks

Vanity slide

- Ola Bini
- From Stockholm, Sweden
- Programming language nerd (Lisp, Ruby, Java, Smalltalk, Io, Erlang, ML, C/C++, etc)
- JRuby Core Developer (2 years and running)
- Author of Practical JRuby on Rails (APress)

ThoughtWorks®

Agenda

- What is JRuby
- How to get started
- The implementation
- Cool things
- Possibly some Rails
- Q&A

What is JRuby

- Implementation of the Ruby language
- Java 1.5+ (1.4 for JRuby 1.0)
  - Retroweaver can be used for 1.4 compatibility
- Open source
- "It's just Ruby"
- Compatible with Ruby 1.8.6
- JRuby 1.0.3 and 1.1RC3 current releases
  - 1.0-branch rapidly falling behind

Community

- 8 Core developers
- 40-50 contributors
- Outstanding contributions
  - Like Marcin's Oniguruma port
    - Joni is probably the fastest Regexp engine for Java right now

Ruby Issues - Threading

- Ruby 1.8: Green threading
  - No scaling across processors/cores
  - C libraries won't/can't yield
  - One-size-fits-all scheduler
- Ruby 1.9: Native, non-parallel execution
- JRuby:
  - Ruby threads are Java threads
  - World class scheduler with tunable algorithms

Ruby Issues - Unicode

- Ruby 1.8: Partial Unicode
  - Internet connection applications MUST have solid Unicode
  - Ruby 1.8 provides very partial support
  - App devs roll their own: Rails Multi-byte
- Ruby 1.9: Full encoding support
  - Drastic changes to interface and implementation
  - Performance issues
  - Each string can have its own encoding
- JRuby: Java Unicode

Ruby Issues - Performance

- Ruby 1.8: Slower than most languages
  - 1.8 is usually called "fast enough"
  - ...but routinely finishes last
  - ...and no plans to improve in 1.8
- Ruby 1.9: Improvement, but not scalable
  - New engine about 1.5x for general appliciations
  - Only implicit AOT compilation
  - No JIT, no GC or threading changes
- JRuby: Compiler provides better performance

Ruby Issues - Memory

- Ruby 1.8: Memory management
  - Simple design
  - Good for many apps, but not scalable
  - Stop-the-world GC
- Ruby 1.9: No change
  - Improved performance => more garbage
  - GC problems could multiply
- JRuby: World class Java GC's

Ruby Issues - C

- Ruby 1.8 & 1.9: C language extensions
  - C is difficult to write well
  - Badly-behaved extensions can cause large problems
  - Threading and GC issues relating to extensions
  - Portable, but often with recompilation
  - No security restrictions in the system
- JRuby
  - Java extensions
  - GC and threading no problem

Ruby Issues - Politics

- Politics
  - "You want me to switch to what?"
  - "... and it needs servers/software/training?"
  - Potentially better with time (e.g. 1.9)
- Legacy
  - Lots of Java apps in the world
  - Extensive amount of Java frameworks
- JRuby solves both of these by running on top of Java
  - "Credibility by association"

C libraries

- JRuby can't support native extensions
  - Designed around single-threaded execution
    - (i.e. one app, one request per process at a time
  - Stability, security problems
  - Too permissive Ruby extension API
- But who cares?
  - If you want to do it, there's a Java library
  - If no, we support natives access through JNA
  - And even porting is not that hard

Getting started

- Java installation
- Download JRuby binary distro
  - Includes JRuby, Ruby stdlib, RubyGems and rake
- Unpack
  - Multiple copies on the system is fine
- Add <jruby-dir>/bin to PATH
- Install gems (gem install or jruby -S gem install)

Calling Ruby from Java

```
// 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();
}
```

DEMO

Java Integration

Implementation: Lexing, parsing

- Hand written lexer
  - Originally ported from MRI
  - Many changes since then
- LALR parser
  - Port of MRI's YACC/Bison-based parser
- Abstract Syntax Tree quite similar to MRI
  - We've made a few changes and additions

Implementation: Core classes

- Mostly 1:1 core classes map 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)
```

Implementation: Interpreter

- Simple switch based AST walker
- Recurses for nested structures
- Most code start out interpreted
  - Command line scripts compiled immediately
  - Precompiled script (.class) instead of .rb
  - Eval'ed code is always interpreted (for now)
- Reasonably straight forward code

Implementation: Compilation

- Full Bytecode compilation
  - 1.0 had partial JIT compiler (25%)
- AST walker emits code structure
- Bytecode emitter generates Java class + methods
  - Real Java bytecode
  - AOT mode: 1:1 mapping .rb file to .class file
    - Not a "real" Java class, more a bunch of methods
    - ... but has a "main" for CLI execution
  - JIT mode: 1:1 mapping method to in-memory class

DEMO

Precompilation

Compiler problems

- AOT pain
  - Code bodies as Java methods need method handles
    - Generated as adaptor methods
  - Ruby is very terse - the compiled output is much more verbose
  - Mapping symbols safely (class, package, method names)
- JIT pain
  - Method body must live on a class
    - Class must be live in separate classloader to GC
    - Class name must be unique within that classloader

Compiler optimizations

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

Core class implementations

- 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
  - Example: implementing analogs for libc IO functions
- Two custom Regexp implementations
  - New one works with byte[] directly

Threading

- JRuby supports only native OS threads
  - Much heavier than Ruby's green threads
  - But truly parallel, unlike Ruby 1.9 (GIL)
- Emulates unsafe green operations
  - Thread#kill, Thread#raise inherently unsafe
  - Thread#critical impossible to guarantee
  - All emulated with checkpoints (pain...)
- Pooling of OS threads minimizes spinup cost

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, ...)

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

Performance

- No, it's not all that important
  - Until it is!
- JRuby 1.0 was about 2x slower than Ruby 1.8.6
- JRuby 1.1 Beta 1 was about 2x faster
- JRuby trunk is 5x faster, often faster than 1.9
  - As a result, we've stopped working on perf for now
  - ...but targeting Java performance next

DEMO

Benchmarks

ThoughtWorks®

JRuby Internals

- `JRuby::ast_for("1+1") #-> Java AST`

  `JRuby::ast_for { 1+1 } #-> Java AST`

  `JRuby::compile("1+1") #-> CompiledScript`

  `CompiledScript.inspect_bytecode`

  `JRuby::runtime`

  `JRuby::reference("str")`

... evil stuff

- ```
  a = "foobar"
  a.freeze
  JRuby::reference(a).setFrozen(false)
  ```

- ```
  class Foobar; end
  something = Object.new
  JRuby::reference(something).setMetaClass(Foobar)
  ```

- ```
  class Foobar; end
  JRuby::reference(Foobar).getMethods()
  ```
  ```

JRuby on Rails - end to end

- Create application
- Package into a WAR-file, using
  - Warbler
  - JRubyWorks
  - Goldspike
- Deploy WAR file to any Java Web Container
  - Jetty, Tomcat, GlassFish
  - Oracle Application Server, JBoss, WebSphere
  - WebLogic

JtestR

- Test Java code with Ruby
- Glues JRuby together with state of the art Ruby libraries
- Includes RSpec, Test::Unit, dust, Mocha, etc
- Ant and Maven 2 integration
- 0.2 to be released “any time now” (tm)

Rubiq

- Lisp layer on top of JRuby
- Transforms to JRuby AST
- ... and lets JRuby execute it
  - Macros
  - Read macros (used to implement regexp syntax, for example)
  - Pure lexical scoping
  - Lambdas transparently transforms to blocks or Proc.new

ActiveHibernate

```
# define a model (or you can use existing)
class Project
  include Hibernate
  with_table_name "PROJECTS" #optional
  #column name is optional
  primary_key_accessor :id, :long, :PROJECT_ID
  hattr_accessor :name, :string
  hattr_accessor :complexity, :double
end

# connect
ActiveHibernate.establish_connection(DB_CONFIG)

# create
project = Project.new(:name => "JRuby", :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
```

Ruvlets

- 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:
  - Evaluates file from load path based on URL
  - File returns an object with a 'service' method defined
  - Object cached for all future requests

Bare bones Ruvlet

```
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
```

YARV & Rubinius machine

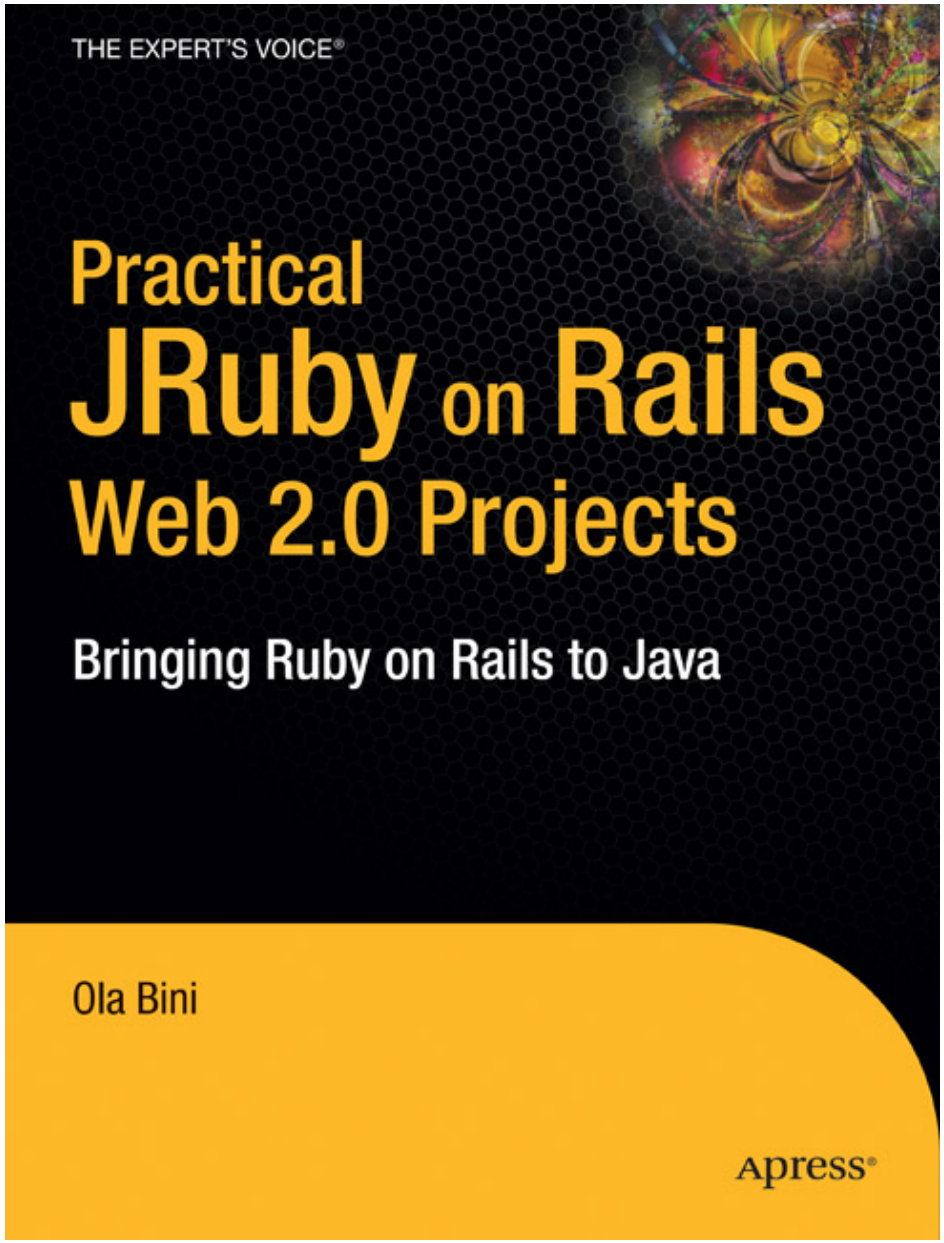
- YARV
  - 2.0 Compatibility
  - Simple machine
  - Simple compiler
  - Might give interpreted performance improvement
- Rubinius
  - Simple machine
  - Quite outdated at the moment
  - Why do it? Why not?

JSR292, JLR & DaVinci

- Dynamic invocation: non-java call sites
- Method handles
- Anonymous classes
- Faster reflection, escape analysis
  - Interface injection
  - Continuations
  - Value objects (Lisp fixnums)
  - Tuple types
  - Tail calls

JRuby's future

- Get 1.1 out there
- Rework the Java Integration features
- Create a stable public API for extensions
- Better performance (as always)
- Support for Ruby 1.9 features
- Light weight objects
- JSR292 support
- Rubinius?
- More primitives in Ruby?


THE EXPERT'S VOICE®
Practical
JRuby on Rails
Web 2.0 Projects
Bringing Ruby on Rails to Java
Ola Bini
Apress®

Resources

- jruby.org
- #jruby on freenode
- glassfish.dev.java.net
- openjdk.java.net/projects/mlvm
- jtestr.codehaus.org
- code.google.com/p/activehibernate
- headius.blogspot.com
- ola-bini.blogspot.com

ThoughtWorks®

Q&A