



JRuby

- Charles Nutter
- JRuby Guy
- Sun Microsystems



Agenda

- Ruby/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

What Is Ruby

- Dynamic-typed, pure OO language
 - > Interpreted
 - > Open source, written in C
 - > Good: easy to write, easy to read, powerful, “fun”
 - > Bad: green threads, unicode support, libraries, “slow”
- Created 1993 by Yukihiro “Matz” Matsumoto
 - > “More powerful than Perl and more OO than Python”
- Very active community, wide range of apps
- Ruby 1.8.x is current, 1.9 is in development to become 2.0

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

Why is Ruby Hard?

- Parsing
- Dynamic invocation
- Closures with mutable containing scopes
- Open classes
- Heavy use of metaprogramming
- Heavy use of 'eval'
- Cross-call-accessible frame data
 - > Perl-like cross-call “globals” (\$~, \$_)
 - > Visibility as part of frame state ('public', etc are methods)
 - > Synthetic stack trace (mixed mode)

Parsing

- 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 clone
 - > DefaultRubyParser.y => DefaultRubyParser.java
- Abstract Syntax Tree similar to MRI's
 - > we've made a few changes/additions

Dynamic Invocation

- DynamicMethod subtypes as “method handles”
- Multiple 'call' overloads to avoid Object[] boxing
- CallSite object per call site
- Monomorphic cache holds/guards DynamicMethods
- Polymorphic sites fail over to slow path
- Ruby core => call site => method handle => target

A Ruby Sample

```
require 'some_library'

def top_level_method
  puts 'Hello, World'
end

module MyMixin
  def print_name
    puts(yield name)
  end
end
```

```
class Foo < Object
  include MyMixin
  def initialize
    @name = 'Charles'
  end
  attr_accessor :name
end

f = Foo.new
f.print_name {|n|
  n + ' Nutter'
}
```

Many Method Handles (Call Targets)

- Interpreted Ruby code
 - > Possibly containing JIT-compiled bytecode later
- Precompiled Ruby core
- Core Java-based Ruby methods
 - > Generated handles to avoid reflection...
- External Java library methods
- Various Ruby wrappers (aliased methods, etc)

JIT Compilation

- Simple invocation counter-based
- Method body compiled to JVM bytecode
 - > Java method requires a class to contain it
 - > Class requires its own ClassLoader to GC
 - > “Polly PermGen” becomes our enemy
- Hard to balance our JIT and HotSpot's desires
- Other code bodies do not JIT (closures, class body)

Why JIT (Why Mixed-mode)?

- Very dense language, expensive to compile
 - > Command-line culture
- Heavy use of eval + code generation
 - > Rails generates more code than it loads
- Frequent use of throw-away code
 - > AST is cheap to throw away; bytecode is not
- Ruby too dynamic? JVM not dynamic enough?

AOT Compiled Ruby

- Exactly 1:1 .class per .rb
 - > A “method bag” dynamically assembled at runtime
 - > Java method per distinct code body
 - Methods, closures, class bodies, and script body
- main() leads to script body for root exec
- load() leads to script body for load exec
- run() and `__file__()` lead.... for JIT exec

foo.rb

```
puts 'hello'  
class Foo  
  def bar  
    baz { }  
  end  
end
```

fib.class

Compiled from "foo.rb"

```
public class foo extends org.jruby.ast.executable.AbstractScript{
    public foo();
    public static {};
    public IRubyObject __file__(TreadContext, IRubyObject, Block);
    public IRubyObject class_0$RUBY$Foo(ThreadContext, IRubyObject, Block);
    public IRubyObject method__1$RUBY$bar(ThreadContext, IRubyObject, Block);
    public IRubyObject block_0$RUBY$__block__(ThreadContext, IRubyObject, IRubyObject);
    public IRubyObject method__1$RUBY$bar(ThreadContext, IRubyObject, IRubyObject[],
        Block);
    public IRubyObject class_0$RUBY$Foo(ThreadContext, IRubyObject, IRubyObject[], Block);
    public IRubyObject __file__(ThreadContext, IRubyObject, IRubyObject[], Block);
    public IRubyObject load(ThreadContext, IRubyObject, IRubyObject[], Block);
    public static void main(String[]);
```

Core Ruby Methods

- Reflection is too slow
 - > Slower than direct
 - > **DEAD** slow when used for every call
 - > Argument boxing is hard on the heap
 - > Spending our precious inlining budget
- All core JRuby directly invoked
 - > Tiny class-per-method generated at build time
 - > From call site to method is 2 hops max
 - > Opens up primitive unboxing potential (eventually)
 - > Vastly improved performance

Core Methods: What Else?

- Reflection wrappers
 - > Slow, slow, slow
 - > But less code, shorter startup time
 - > Still available in JRuby
- Hand-written “invoker” impls
 - > Infeasible; must generate code
- Generated “callback” wrapped in generic invoker
 - > Better, but slow; generic wrapper is megamorphic

External Java Library Methods

- Plain old reflection
 - > Handle-per-method is too expensive (pre-292)
 - > “Fast enough” :)
 - > Back to argument boxing
- Edging toward JLS-compliant dispatch
 - > Complicated by many incoming types + coercion
 - > Atilla's MOP project could help
- Common ground across JVM dynlangs

'call' overloads

- Arity-specific for zero to three args
 - > Diminishing returns beyond that
 - > Measurable improvement eliminating arg[]
- With or without closure argument
- Still Object (IRubyObject) throughout
 - > Unboxing primitives is hard and I don't want to do it.
 - > Fixnums please.

CallSite

- Encapsulates logic of making a call
 - > Simplifies generated bytecode quite a bit
- Monomorphic cache
 - > type == guard
 - > Actively flushed when types change
 - Potential threading issue...
 - > Failure (50 misses?) leads to slow path forever
 - > Majority are successful
- Custom site types for operators (experimental)
- Ideally only 1-2 hops added to stack

Dynamic Calls: What Else?

- No caching at all
 - > Obviously still there as a fallback
- Polymorphic inline cache
 - > Not useful for simple and common cases
 - > Experimentally no improvement in current case
 - > Type-splitting core methods could make use
 - Multi-dispatch for core, Java methods
- Selector Table Indexing
 - > “Big switch” dispatch
 - > Numbered classes and method selectors
 - > Big switches are too slow in hotspot

Heap scopes

- Closures require a heap-based scope
 - > ...but we can statically inspect code and see closures
- eval and friends require heap-based scope
 - > Many ways to reuse a “binding”
 - > ...but we can cheat, treat 'eval', etc as pseudo-keyword
- For many cases we can omit heap scope
 - > Normal JVM local variables
 - > Both faster and easier on heap/GC
 - > JITed methods too
- Multiple size-specific scope types to avoid arrays

Scopes: What Else?

- Classic impl was hash-based for some scopes
 - > ...but still static lexical scoping
 - > ...so obviously wrong, unnecessary
- “Big array”
 - > Bite off chunks during method activation
 - > Release after deactivation
 - > Incompatible with closures, long-lived scopes

Open Classes

- All classes created at runtime
 - > First opening instantly creates
 - > Method defs are mutating class structure
 - > Always can re-open
- Hierarchy mutations
 - > Mix-in inheritance at any time
 - > Singleton object types pulled off at any time
- Used heavily, even against core types
 - > Adding methods to String, Fixnum, Object, Class...
 - > Nothing is sacred

Open Classes

- No mapping to Java classes
 - > Must implement or extend to call from Java to Ruby
- Types are not (can't be) normal Java types
 - > “Old” invokedynamic would have been useless
- “Pure” open classes and “eval” keyword cheat
 - > Have to limit openness a bit...
- eval and code generation limits static inspection
 - > Classes may not even exist in inspectable Ruby code!
 - > Dynamic upon dynamic upon dynamic

Heap-based Frame

- Traditionally static data are mutable
 - > Visibility
 - > 'self' or 'this'
 - > Class we're invoking against
 - > File name, line number, method name (backtrace)
- Can be captured for “binding” methods like eval
- BUT...high percentage of uses are static!

Frame Elimination

- Visibility only needed in presence of method def
- 'self' or 'this' for visibility (so pass caller too)
- Class we're invoking against needed for 'super'
- Backtrace can be mined from Java trace
- “eval” keyword cheat

Library Challenges

- Mutable string that doubles as byte-bucket
- Regexp support for same
- libc IO behavioral dependencies
- Green thread behavioral dependencies
- POSIX features
- C library support
- Continuations, generators, fibers
- ObjectSpace

String

- Custom-implemented byte[]-based String
- Encoding-agnostic
- Copy-on-write
- No mismatch with IO
- Mismatched with Java (!!!)
 - > Decode/encode perf problems

String: What Else?

- `char[]` or `java.lang.String`-based
 - > Mismatch for Ruby's usage
 - > Perf issues constantly encoding and decoding
- Hybrid implementation
 - > Could work
 - > Lots of effort involved

Regex

- Custom bytecode-based Regex engine
- Encoding-agnostic; operates on `byte[]`
 - > If String is `byte[]`, Regex must be `byte[]`
 - > Regex against NIO ByteBuffer?
- Port of Oniguruma engine (65kloc library)

Regex: What Else

- `java.util.regex`
 - > Performs great
 - > Blows stack for deep alternations
- `JRegex`
 - > Stackless (bytecodish)
 - > Fast
 - > `char[]`-based

libc IO

- Custom buffered IO built atop NIO
- File-descriptor-like abstraction
- FILE*-like abstraction
- Attempts to mimic libc behaviors
 - > ...yes, I have had to read glibc code in some cases
 - > ...yes, it's frequently unpleasant
- NIO's separate channel types get in the way
 - > fdopen is tricky, for example
 - > select doesn't always work

IO: What Else?

- java.io
 - > Hah.
- NIO directly
 - > Too low-level
 - > But obviously formed the basis of our libc IO

Green thread behaviors

- Heavy thread churn in some apps
 - > Built-in thread pool (java concurrency API-based)
- kill/raise in a target thread (not current)
 - > Checkpointing (please kill yourself)
- critical sections (really, critical flag)
 - > Check critical lock at checkpoint
 - > Wait for signal to resume exec
- We try to teach Rubyists not to use these
 - > ...but they do anyway
 - > ...but they're starting to feel dirty about it

POSIX Features

- Java Native Access for arbitrary calls
- jna-posix project aggregating functions as needed
 - > Abstracts platform specifics like structure layout
 - > Filesystem stuff (symlink, chmod, ...)
 - > IO stuff (unix sockets, ...)
 - > etc/passwd stuff
 - > Process control that doesn't suck
 - > fork!!! (Not recommended)
 - > ...and so on...looking for contributors
- Filling in the blanks in Java class libs

POSIX: What Else?

- There is no what else.
 - > Missing features are a **glaring** flaw in current JDK
 - > We have no alternative to calling them directly
 - > We do not have the option of omitting them

C Library Support

- JNA again, with a Ruby-friendly wrapper

```
require 'ffi'
module LibC
  extend FFI::Library
  callback :qsort_cmp, [ :pointer, :pointer ],
    :int
  attach_function :qsort, [ :pointer, :int, :int,
    :qsort_cmp ], :int
end

p = MemoryPointer.new(:int, 2)
p.put_array_of_int32(0, [ 2, 1 ])
LibC.qsort(p, 2, 4) do |p1, p2|
  i1 = p1.get_int32(0)
  i2 = p2.get_int32(0)
  i1 < i2 ? -1 : i1 > i2 ? 1 : 0
end
```

C Libs: What Else?

- Again, there is no what else
 - > C (dylib) invocation should be core to JDK
 - > Great power to be used carefully
 - > Too many libs we want access to
 - > Too many libs Rubyists won't live without

Continuations

- No.

Continuations

- No general continuations
- Scoped continuations (generators/fibers/coroutines)
 - > Via a native thread
 - > Enlists in JRuby thread pool (soon)
 - > “Good enough”
- Tried a stackless interpreter
 - > Unusably slow
- Ultimately, nobody uses continuations

ObjectSpace

- No.

ObjectSpace

- Not on by default
 - > Limited support for walking all classes
- Command-line flag, property, or API call to enable
- Essentially a parallel weak set for **all objects**
 - > Tremendous performance impact
- Class-walking is 90% of ObjectSpace use

Future: JRuby

- Ruby 1.8.x compatibility is 99%
- Ruby 1.9 compatibility is next
- Performance arms race forever
- Refactoring, cleanup forever
- Rubifying Java libraries for Rubyists
- Exposing Ruby libraries for Javaists
- **Learn from JRuby to improve JVM**
- **Learn from JRuby to help other languages**

Future: Object, not IRubyObject

- We desire to move away from IRubyObject
 - > Non-Ruby types must be wrapped
 - > Wrappers must be reused, idempotent (minimally)
 - > Heavy Java library invocation cost
- Moving to Object requires substantial rework
 - > New pervasive MOP structure
 - > Decouple MOP from object instances
- Attila's DynaLang project may serve as a base
- More code generation will help ease the pain

Future: Invokedyynamic

- Handles eliminate our generated versions
- Anonymous classloading simplifies code transience
- Dynamic invocation allows deleting **lots** of code
 - > Call site cache
 - > Hand-built call path overloads
 - > Hand-wired curried features
- Performance, maybe (probably?)
- Codebase simplification, definitely
- Faster Java library invocation, definitely

Future: Maxine

- Another (J)VM to expose impl holes
- A testbed for JVM modifications
 - > Dynamic invocation
 - > First-class native calls (rather than JNI to FFI to lib)
 - > Stack decoration for Ruby purposes
 - > Explicit runtime hinting for Ruby
 - Inlining
 - Explicitly transient objects
 - Explicitly thread-local, thread-bound objects
- A modified Maxine as a “JRuby VM”?
 - > (A modified X as a “JRuby VM”, really)

Links

- <http://www.jruby.org>
- <http://www.ruby-lang.org>
- <http://groups.google.com/group/jvm-languages>
- <http://openjdk.java.net/projects/mlvm/>
- <https://maxine.dev.java.net/>



JRuby

- Charles Nutter
- JRuby Guy
- Sun Microsystems



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`, `jrubycli`
 - > `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 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)
```

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

```
~ $ jrubyc bench_fib_recursive.rb
Compiling file "bench_fib_recursive.rb" as class
  "bench_fib_recursive"

~ $ jrubyc 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
...
```

JRuby Compiler

- 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

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

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

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.JButton

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

button = JButton.new("Press me!")
button.add_action_listener 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
```

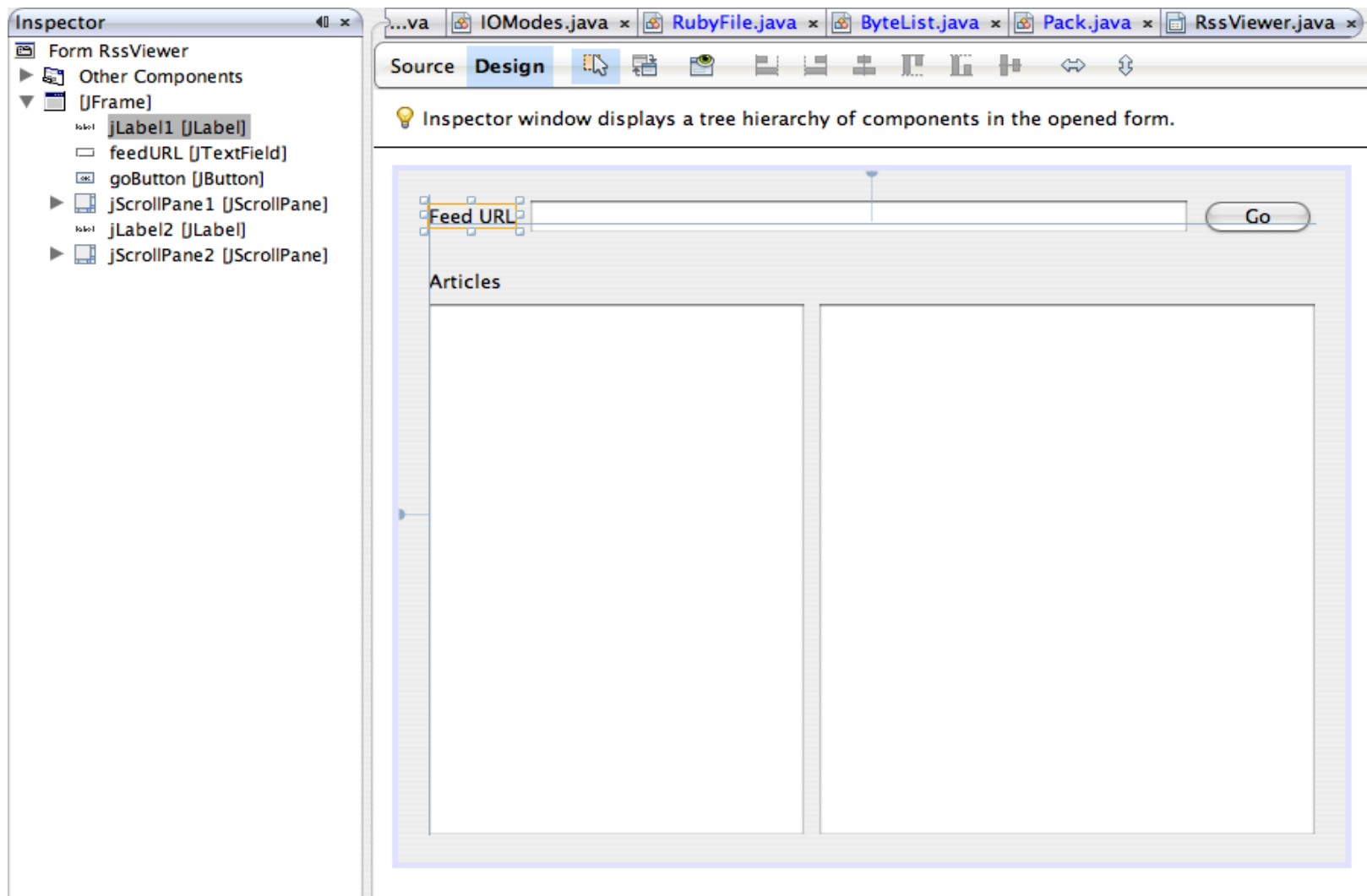
Profligacy
the world needs less swing

Option 4: MonkeyBars (tool-friendly)

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

monkeybars

MonkeyBars + NetBeans Matisse



The screenshot shows the NetBeans IDE interface. On the left, the **Inspector** window displays 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 is in **Design** mode, showing a visual representation of the form. At the top, there is a text field labeled **Feed URL** and a **Go** button. Below this, there is a section titled **Articles** which contains two empty scrollable panes.

A lightbulb icon with the text "Inspector window displays a tree hierarchy of components in the opened form." is positioned above the design view.

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
    @rss = RSS::Parser.parse(content, false)

    model.articles = @rss.items.map {|art| art.title}
    model.article_text =
      CGI.unescapeHTML(@rss.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: 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
```



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 < HTTPRuvlet
  def doGet(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

  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

  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