

Building a Multilanguage VM

Charles Oliver Nutter
Language Hacker
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/>).

Agenda

- Introduction
- Virtual Machines
- JVM Languages
 - > Overview and example
- Case Study: JRuby
- The Multilanguage VM
- The Future

Who Am I

- Charles Oliver Nutter <charles.nutter@sun.com>
- JRuby co-lead
 - > Compiler and runtime work
- Language enthusiast
 - > C, C++, Pascal, VB, Java, Ruby...
- Bringing implementers together
- Helping to drive JVM changes

Virtual Machines

Virtual Machines

- A *virtual machine* is a software implementation of a specific computer architecture
 - Could be a real hardware architecture or a fictitious one
- *System* virtual machines simulate a complete computer system
 - VMWare, VirtualBox, VirtualPC, Parallels
 - Usually implement a real hardware architecture (e.g., X86)
- *Process* virtual machines host a single application
 - Like the JVM
 - Usually implement a fictitious instruction set designed for a specific purpose
 - Instruction set can be chosen to maximize implementation flexibility
- Initial implementations of VMs are often interpreted (and slow)
 - More mature implementations allow sophisticated compilation techniques

VMs win as compilation targets

- Today, it is silly for a compiler to target actual hardware
 - > Much more effective to target a VM
 - > Writing a native compiler is lots more work!
- Languages need runtime support
 - > C runtime is tiny and portable (and wimpy)
 - > More sophisticated language runtimes need
 - Memory management
 - Security
 - Reflection
 - Concurrency control
 - Libraries
 - Tools (debuggers, profilers, etc)
- Many of these features are baked into VMs

Lots of VMs out there

- Java Virtual Machine (JVM)
- .NET Common Language Runtime (CLR)
- Smalltalk
- Squeak
- Perl
- Parrot (Perl 6)
- Python
- YARV
- Rubinius
- Tamarin (ActionScript)
- Valgrind (C++)
- Lua
- LLVM
- TrueType
- Dalvik
- Flash
- p-code (USCD Pascal)
- Zend

Is there a universal VM in sight?

JVM Languages

Language Survey

- The JVM is multilanguage today
 - > 200+ implementations: more than any other VM
 - > Dozens of languages: nearly all major ones
 - > Some quality, some not, some alive, some dead
- No language is impossible
 - > Functional languages most “different” from Java
 - > Dynamic languages hardest to optimize
- No language is too weird
 - > LOLCODE – Programming the LOL way.
 - > NestedVM – MIPS assembly to JVM bytecode

Java Example

```
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JButton;
import java.awt.GridLayout;
import java.awt.event.ActionListener;

public class MyApp {
    public static void main(String[] args) {
        JFrame frame = new JFrame("My Frame");
        final JButton button = new JButton("Press Me!");

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                button.setText("Hello Java!");
            }
        });

        frame.add(button);
        frame.setLayout(new GridLayout(2, 2, 3, 3));
        frame.setSize(300, 80);
        frame.setVisible(true);
    }
}
```

Clojure

- <http://clojure.sourceforge.net/>
- A dynamic, compiled, functional dialect of Lisp for the JVM, with strong concurrency support.
 - > Dynamic typing, with optional type hints
- “substantially easier to read and write”
 - > Vector, list, map literals
 - > “far fewer parentheses”
- Strong emphasis on immutability by default
 - > Key to safe concurrency

Clojure Example

```
(import '(javax.swing JFrame JLabel JTextField JButton)
        '(java.awt.event ActionListener)
        '(java.awt GridLayout))
(let [frame (new JFrame "My Frame")
      my-button (new JButton "Press me!")]
  (. my-button
    (addActionListener
      (proxy [ActionListener] []
        (actionPerformed [evt]
          (setText "Hello Clojure!"))))))
(doto frame
  (setLayout (new GridLayout 2 2 3 3))
  (add my-button)
  (setSize 300 80)
  (setVisible true)))
```

Groovy

- <http://groovy.codehaus.org/>
- “agile and dynamic language for the JVM”
- Many features from Ruby, Python
 - > Closures, mutable classes, literal syntax
- Syntax very similar to Java
 - > Some Java will “just run”
 - > Idiomatic Groovy can be as terse as Ruby
- Many libraries built in
 - > GUI builder, testing, mocking

Groovy Example

```
import java.awt.event.ActionEvent
import javax.swing.JFrame
import javax.swing.JButton
import java.awt.GridLayout
import java.awt.event.ActionListener

def frame = new JFrame("My Frame")
final button = new JButton("Press Me!")

button.addActionListener( {
    button.text = "Hello Java!"
} as java.awt.event.ActionListener )

frame.add button
frame.layout = new GridLayout(2, 2, 3, 3)
frame.setSize(300, 80)
frame.visible = true
```

JRuby

- <http://www.jruby.org>
- Ruby for the JVM
 - > Dynamic typing, terse syntax, operator overloading, metaprogramming, very active and growing community
 - > Rails and apps like it run great
 - > Better performance, scaling, than all other impls
 - > Interesting case for JVM improvements
- Ruby for Java
 - > Simplifies complex libraries
 - > Integrates well with Java classes
 - > Rails for Java shops

JRuby Example

```
require 'java'  
import javax.swing.JFrame  
import javax.swing.JButton  
import java.awt.GridLayout  
  
frame = JFrame.new("My Frame")  
button = JButton.new("Press me!")  
  
button.add_action_listener { button.text = "Hello Ruby!" }  
frame.layout = GridLayout.new(2, 2, 3, 3)  
frame.add button  
frame.set_size 300, 80  
frame.visible = true
```

Scala

- <http://www.scala-lang.org/>
- Object-oriented: classes, traits, and mixins
- Functional: function objects, currying, pattern matching, case classes
- Statically-typed: type inference, rich type system
- Extensible: nice-looking DSLs, closure creation
- Fully interoperable with Java

Scala Example

```
import javax.swing.{JFrame, JLabel, JTextField, JButton}
import java.awt.event.{ActionListener, ActionEvent}
import java.awt.GridLayout
```

```
val frame = new JFrame("MyFrame")
val myButton = new JButton("Press me!")
myButton.addActionListener(
  new ActionListener {
    def actionPerformed(evt: ActionEvent) {
      myButton.setText("Hello Scala!")
    }
  }
)
```

```
def chain[T](what: T)(actions: T => Unit*) = actions.foreach(_(what))
```

```
chain(frame)(_.setLayout(new GridLayout(2,2,3,3)),
  _.add(myButton),
  _.setSize(300,80),
  _.setVisible(true))
```

Case Study: JRuby

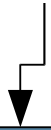
Ruby for the JVM

- Why it's hard:
 - > Dynamic typing makes optimization tough
 - > Brings along its own type system
 - > Real-world dependencies on C extensions
 - > Difficult features: “eval”, continuations, byte-based String
- Why it's rewarding:
 - > Beautiful, elegant language
 - > New blood, new ideas for the Java world
 - > A better runtime
 - > Pushing the bounds of the JVM

Java VM vs Java Language

Java language fictions Ruby language fictions

JVM features



Checked exceptions
Generics
Enums
Overloading
Constructor chaining
Program analysis

Open classes
Dynamic typing
'eval'
Closures
Mixins
Rich set of literals

Primitive types+ops
Object model
Memory model
Dynamic linking
Access control
GC
Unicode

Primitive types+ops
Object model
Memory model
Dynamic linking
Access control
GC
Unicode

~~Primitive types+ops~~
Object model
Memory model
Dynamic linking
~~Access control~~
GC
Unicode

JRuby

- Complex
 - > Mixed-mode: interprets, then compiles later
 - > Custom String, Array (List), Hash (Map), Regexp
 - > Generated code to avoid reflection overhead
 - Lots and lots
 - > Multiple VM: run many Ruby apps on one JVM
- What would make it easier
 - > Lightweight code generation and loading
 - > Faster-than-reflection method objects
 - > Toolchain for parsing, interpreting, compiling
 - > More flexible JVM type system

The Multi-Language VM

Making MLVM a Reality

- Many common goals and challenges
 - > Why keep reinventing the wheel?
- We're all growing the platform
 - > We want the Java platform to succeed
 - > We're working hard to make sure it does
- We're all extending the platform
 - > Challenges are opportunities
 - > The platform isn't perfect, but it's Open
 - > It's up to you and us

JVM Languages Group

- <http://groups.google.com/group/jvm-languages>
- Implementers from most major languages
- 678 members, 1963 posts since mid 2007
- Discussions on parsing, compiling, threads, more
- Sharing information, ideas
- Discussing future plans for languages and JVM
- Great fun to read, participate
 - > Even if you're not a language person!

The Da Vinci Machine Project

- <http://openjdk.java.net/projects/mlvm/>
- OpenJDK Multi-language VM
- Feature testbed for future JDKs
 - Anonymous classloading (prototype working)
 - Lightweight method handles (prototype almost done?)
 - Optimized dynamic invocation (waiting on handles)
 - Continuations (proof-of-concept working, prototype coming)
 - Tail call optimizations (under research)
 - Tuples
 - <YOUR FEATURE HERE>

JSR 292

- Often called the "invokedynamic" JSR
 - > Because it originally proposed a specific bytecode for method invocation
 - > Scope has widened since then
- Work currently going on in JSR 292 includes
 - > **invokedynamic** bytecode
 - Allow the language runtime to work hand-in-hand with the JVM on method selection
 - > Method handles
 - Many languages have constructs like closures
 - Classes are too heavy a container for a single block of code
 - > Interface injection
 - Add new methods and types to existing classes

Virtual method invocation in Java

- Take some simple source code:

```
String s = "Hello World";  
System.out.println(s);
```

- Let's look at the bytecode:

```
0:   ldc #2           //String "Hello World"  
2:   astore 1  
3:   getstatic #3      //Field  
                        java/lang/System.out:  
                        Ljava/io/PrintStream;  
6:   aload 1  
7:   invokevirtual #4 //Method  
                        java/io/PrintStream.println:  
                        (Ljava/lang/String;)V
```

- A Java compiler chooses a version of `println` and embeds its **static type signature** into the bytecode

Dynamically typed method invocation

- Compiling dynamic languages directly to the JVM is tricky

```
def max(x, y)
  x.less_than(y) ? y : x
end
```

- What do we compile the `less_than()` call to?

```
invokevirtual less_than: (unknownArgType)boolean
```

- That's not going to work
 - > No receiver type
 - > No static argument type
 - > Maybe the return type isn't even boolean, maybe it's the type of `y` or `x`

invokedynamic

- Consider this (hypothetical) instruction:
 - `invokedynamic Object.lessThan: (Object)boolean`
 - > and suppose the receiver is an Integer and the argument is a Long
- **We** know that invokedynamic should act as if it were:
 - `invokevirtual Integer.lessThan: (Long)boolean`
- If the VM knew that, it could find the method **and inline it**
- The language runtime can bridge the gap, if we ask it nicely

Method selection in dynamic languages

- A language runtime wants to take
 - > `invokedynamic Object.lessThan: (Object)boolean`
- and do what a Java compiler does with static types at overload resolution, only with dynamic types:
 - > Inspect the dynamic type of the receiver
 - > Inspect the dynamic type of the argument
 - > Check which methods are available **now** in the receiver's dynamic type
 - > Decide if a single argument is acceptable to those methods
 - Considering language rules on arity, optional parameters, default parameters...
 - > Box values of primitive type to values of reference types
 - > Box the argument list into an array and optimize it
- eventually deciding to invoke
 - > `invokedynamic Integer.lessThan: (Long)boolean`

A language runtime wants to do all this...

ONCE

(Until the receiver variable is assigned to a different object,

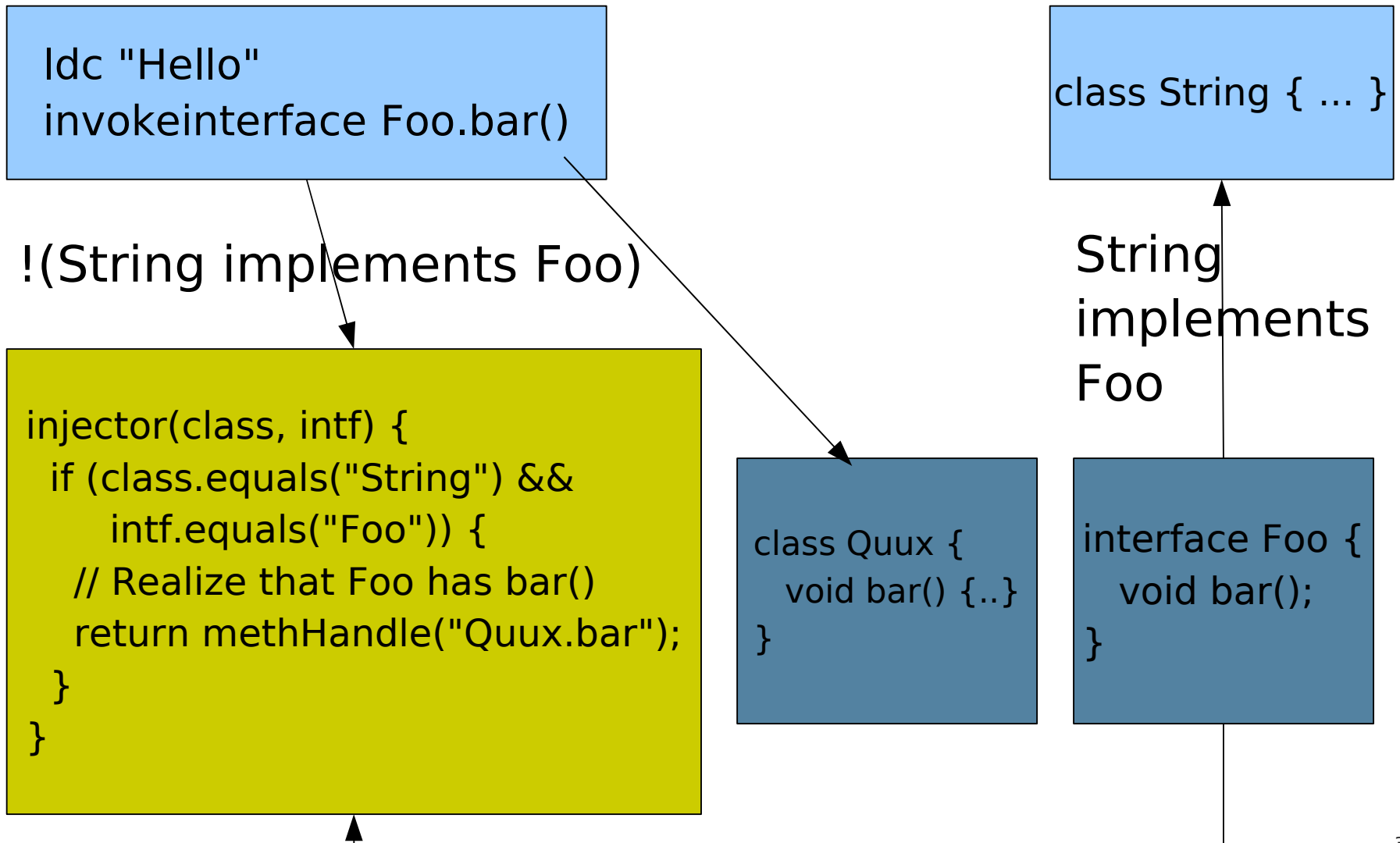
or the receiver object's dynamic type is changed,

or the argument object's dynamic types are changed)

Interface injection

- Dynamically typed programs look like self-modifying code
 - > A class definition or method body can change
- Self-modifying code is hard to optimize
- Idea: Don't restructure classes, just relabel them
- Interface injection: The ability to modify old classes just enough for them to implement new interfaces
 - > Superinterfaces are cheap for JVM objects
 - > `invokeinterface` is fast these days
- If an interface-dependent operation is about to fail, call a static injector method to bind an interface to the object and provide `MethodHandles` for the interface's methods
 - > One chance only for the injector to say yes!

Injection in action



Example: Dynamic dispatch in Groovy

- Groovy maintains the fiction that Java strings have extra methods
 - > `Object asType(Class) // implements the 'as' operator`
 - > `Object eachLine(Closure)`
 - > `List tokenize(String)`
- Call to tokenize cannot compile to
 - > `invokevirtual java.lang.String.tokenize:(String)List`
- Groovy runtime must call tokenize by:
 - > Getting the receiver's class (`java.lang.String`)
 - > Performing a table lookup to map it to a `MetaClass` object
 - > Calling `invokeMethod(s, "tokenize", ..)` on the `MetaClass` object
- The lookup is the JIT's nemesis - can't inline a table lookup!

Interface injection to the rescue

- Interface injection removes the need for a table lookup
- Every object in a Groovy program becomes a GroovyObject
 - Even system types like java.lang.String
- Then, the Groovy compiler's standard invocation works:
 - ```
((GroovyObject) s).getMetaClass().invokeMethod(s, "tokenize", ..)
```
- The first time s is cast to GroovyObject, interface injection occurs
- Thereupon, getMetaClass and invokeMethod calls can be inlined

# Other fun stuff

- Tail calls
- Continuations
- Tuples
- Value objects

# Your Future

# Next Steps for You!

- Make OpenJDK succeed
  - > It's really, truly, OSS
  - > The alternatives are closed and proprietary
  - > It's up to us to make the MLVM dream come true
- It's a polyglot world out there
  - > Attend talks for these languages!
  - > Try them out, report bugs and feedback!
  - > Rewrite a piece of code or library of yours!
  - > Get on the mailing lists, and see how you can help!
  - > Help out? Build your own?

# Get Involved!

- Languages are both easy and difficult
  - > Core types are easy
    - Run tests, fix bugs, write better methods (easy)
      - Vast bulk of work for JRuby is in core types
    - Mostly “just Java”, sometimes the language itself (easy, fun!)
  - > Interpreters are easy
    - Given a tree...walk it (easy)
    - Do stuff for each node (easy, maybe obscure)
  - > Compilers are difficult, but fun
    - Given a tree...walk it (easy)
    - Generate bytecode (tricky, obscure)
    - Generate **good** bytecode (hard, even more obscure)

**Questions?**

**Thank you!**