The Da Vinci Machine Project: Collaborating on JVM™ Futures

John Rose
Sun Microsystems

Overview

The Da Vinci Machine Project is incubating significant changes to the JVM™ bytecode architecture including JSR 292.

Is VM architecture and language support your cup of joe?
Join us!

Boredom alert: JVM geek-fest ahead

- This talk is not about:
  - any particular great new Java API or JVM language
  - the Java language (present or future)
  - the present design of the JVM
  - any *decided* future of the JVM (except JSR 292)

- This talk *is* about:
  - some *possible* futures of the JVM
  - why they are important
  - how language and runtime implementers can help

Computer languages are important

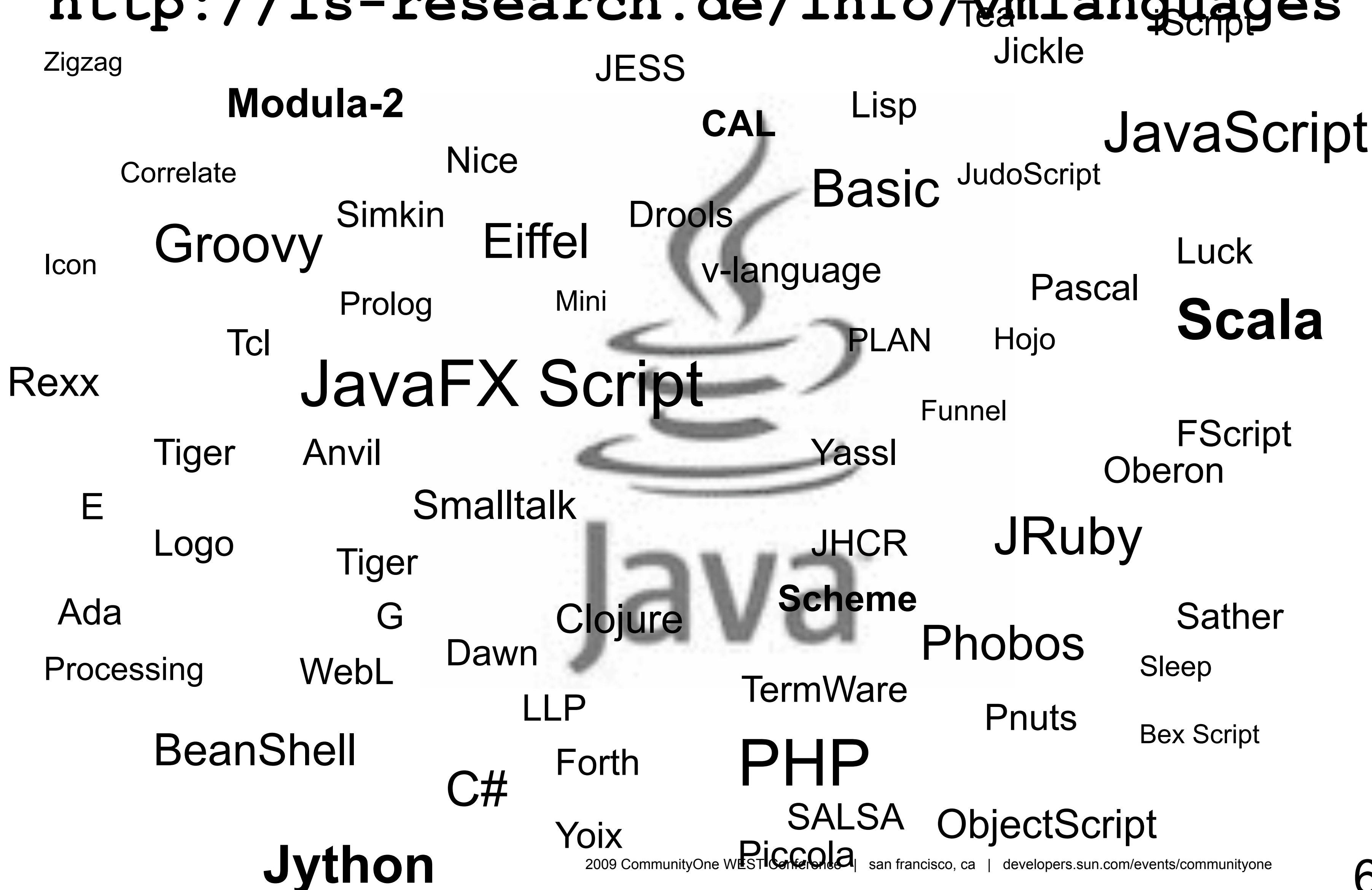
- They make us more productive
  - useful languages ➔ effective (happy) programmers
- There seems to be no limit to language invention
  - (Much of it is endless re-invention, which is OK.)
  - Language adaptation probably culturally necessary
- As computers grow powerful, languages change
  - more/faster gates ➔ less detail work for humans
  - yesterday's HLL = today's SAL (structured assembly language)

Virtual machines are important

- Today, it is silly for a compiler to target hardware
  - a VM provides a simple, powerful, portable target
  - writing a native compiler is hard work
  - …and the work is endless: hardware change
- Similar points for: GC, IDE, threads, algo's, etc.
- VMs are a major investment site since the '90s
  - JVMs represent programmer *centuries* of effort
  - BYO runtime = BYO operating system (fun+futile)

Languages on the JVM (240 and counting)

`http://is-research.de/info/vmlanguages`



Build on a JVM for *speed*:

- uniprocessor performance (optimizing JIT, not interpreter)
- efficient memory usage (copy/compact GC, not malloc)
- multiprocessor scaling (pervasive threading)

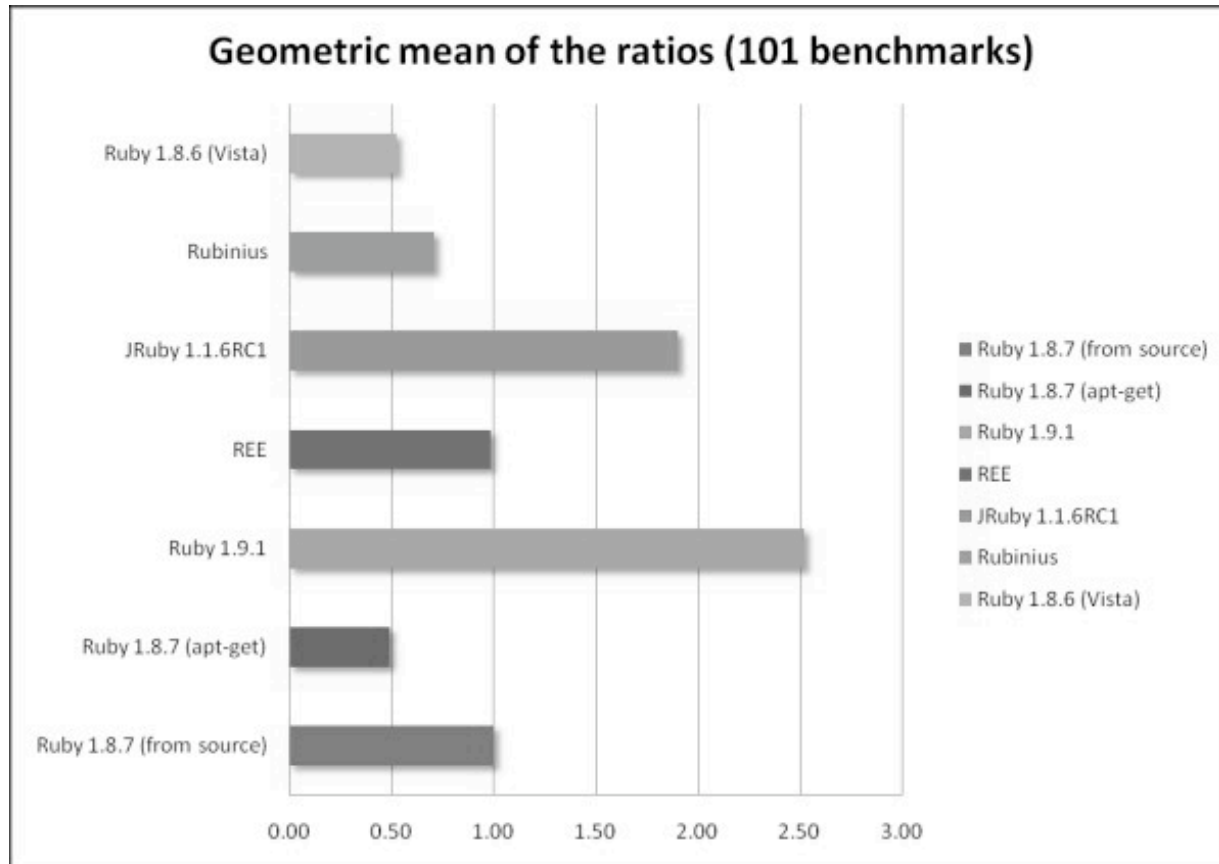
Other JVM advantages:

- reliability (solid quality engineering)
- security (sandboxing, partitioning)
- numerous options for deployment
- corporate legitimacy of JVM bytecodes
  - all's fair in a WAR (-file)

More benefits hovering over the JVM:

- big ecosystem of Java tools and libraries
- multi-language (multi-paradigm) development
  - clean inter-language commerce
  - fast evolution of design boundaries
- faster bootstrap for new developers
  - (it’s easier to be competent in Java than in C)

The Great Ruby Shootout, 2008



JVM Specification, circa 1997

- “The Java virtual machine knows nothing about the Java programming language, only of a particular binary format, the class file format.”
- “Any language with functionality that can be expressed in terms of a valid class file can be hosted by the Java virtual machine.”
- “In the future, we will consider bounded extensions to the Java Virtual Machine to provide better support for other languages.”

Enter the Da Vinci Machine Project

`http://openjdk.java.net/projects/mlvm`

- alias mlvm = “Multi-Language Virtual Machine”
- first-class support for languages beyond Java
  - especially dynamic languages
  - performance comparable to Java itself
- removing “pain points” experienced by developers
  - integrated general extensions, not hacks or plugins
  - well-balanced platform for interoperability & reuse

the mlvm code is visible to the public

- OpenJDK project page:

`http://openjdk.java.net/projects/mlvm`

- Patch repositories (Mercurial forest):

```
http://hg.openjdk.java.net/mlvm/mlvm
http://hg.openjdk.java.net/mlvm/mlvm/hotspot
http://hg.openjdk.java.net/mlvm/mlvm/jdk
http://hg.openjdk.java.net/mlvm/mlvm/langtools
```

mlvm communications are public

- Email: `mlvm-dev@openjdk.java.net`
- design notes:

`http://wikis.sun.com/display/mlvm`

`http://wikis.sun.com/display/HotSpotInternals`

`http://blogs.sun.com/jrose/category/JVM`

- IRC: `irc.freenode.org #mlvm`

A cluster of subprojects (present)

> invokedynamic (indy.patch)
> method handles (meth.patch)
> *small* Java language changes (langtools/meth.patch)
> interface injection (inti.patch)

More subprojects:

- tail calls (tailc)
- continuations (callcc)
  - thread checkpoint objects
- anonymous classes (anonk)

Da Vinci Machine Project events (early)

- June 2007 – JSR 292 Expert Group resumes
  - JVM “futures” series begins on `blogs.sun.com/jrose`
- Oct 2007 – mlvm project launched in OpenJDK
  - Arnold Schwaighofer begins tail-calls
- 1Q 2008 – public interest warms up
  - Lukas Stadler works on continuations
  - Rémi Forax works on anonymous classes

Da Vinci Machine Project events (last year)

- Apr 2008 – first code posted to mlvm repository
  - anonymous classes (anonk), continuations (callcc)
- May 2008 – JSR 292 E.G. releases draft for review
  - Rémi Forax commits code to JSR 292 Backport project
- Aug 2008 – working method handle code
  - 8/26/2008 = International Invokedynamic Day
- Sep 2008 – initial Java support code (quid, meth)
  - 9/24/2008 = first JVM Language Summit
  - Charlie Nutter begins to refactor JRuby for indy

Da Vinci Machine Project events (3)

- Nov 2008 – anonk promoted to JDK7
- Feb 2009 – working tail-call code (Arnold Schwaighofer)
  - v2 of JSR 292 backport (Rémi Forax)
- Mar 2009 – preliminary interface injection code
  - inti.patch contributed by Tobias Ivarsson
- Apr 2009 – indy promoted to JDK7
- May 2009 – Java support promoted to JDK7
- 1H2009 – JSR 292 E.G. hammers on indy spec.

But enough history — let's talk technology!

The deal with method calls (in one slide)

- Calling a method is cheap (VMs can even inline!)
- Selecting the right target method can be expensive
  - Static languages do most of their method selection at compile time
    - Single-dispatch on receiver type is left for runtime
  - Dynamic languages do almost none at compile-time
    - But it would be nice to not have to re-do method selection for *every single invocation*
- Each language has its own ideas about linkage
  - The VM enforces static rules of naming and linkage
    - Language runtimes want to decide (and re-decide) linkage

Example: Fully static invocation

> For this source code

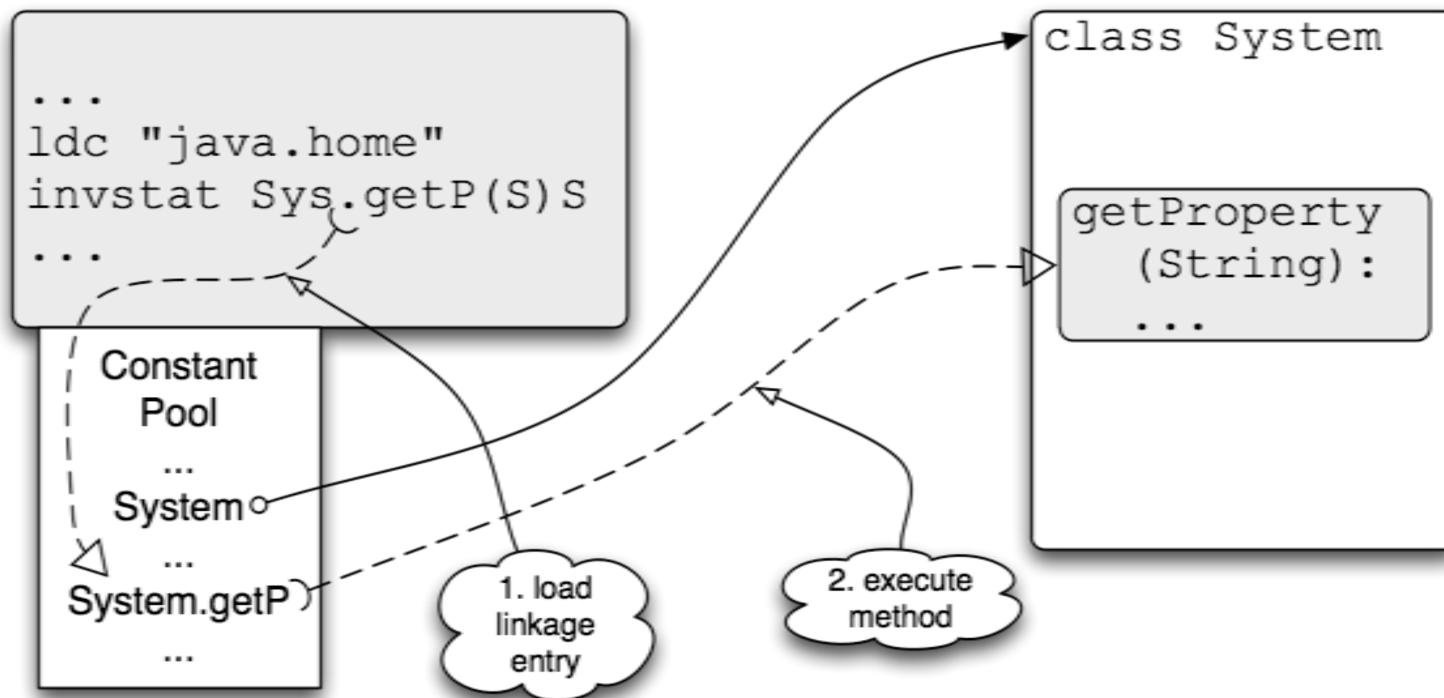
```
String s = System.getProperty("java.home");
```

The compiled byte code looks like

```
0:   ldc #2            //String ”java.home”
2:   invokestatic #3  //Method java/lang/System.getProperty:
                       (Ljava/lang/String;)Ljava/lang/String;
5:   astore_1
```

a) Names are embedded in the bytecode
b) Linking handled by the JVM with fixed Java rules
c) Target method selection is not dynamic at all
d) No adaptation: Signatures must match exactly

How the VM sees it:



(Note: This implementation is typical; VMs vary.)

Example: Class-based single dispatch

> For this source code

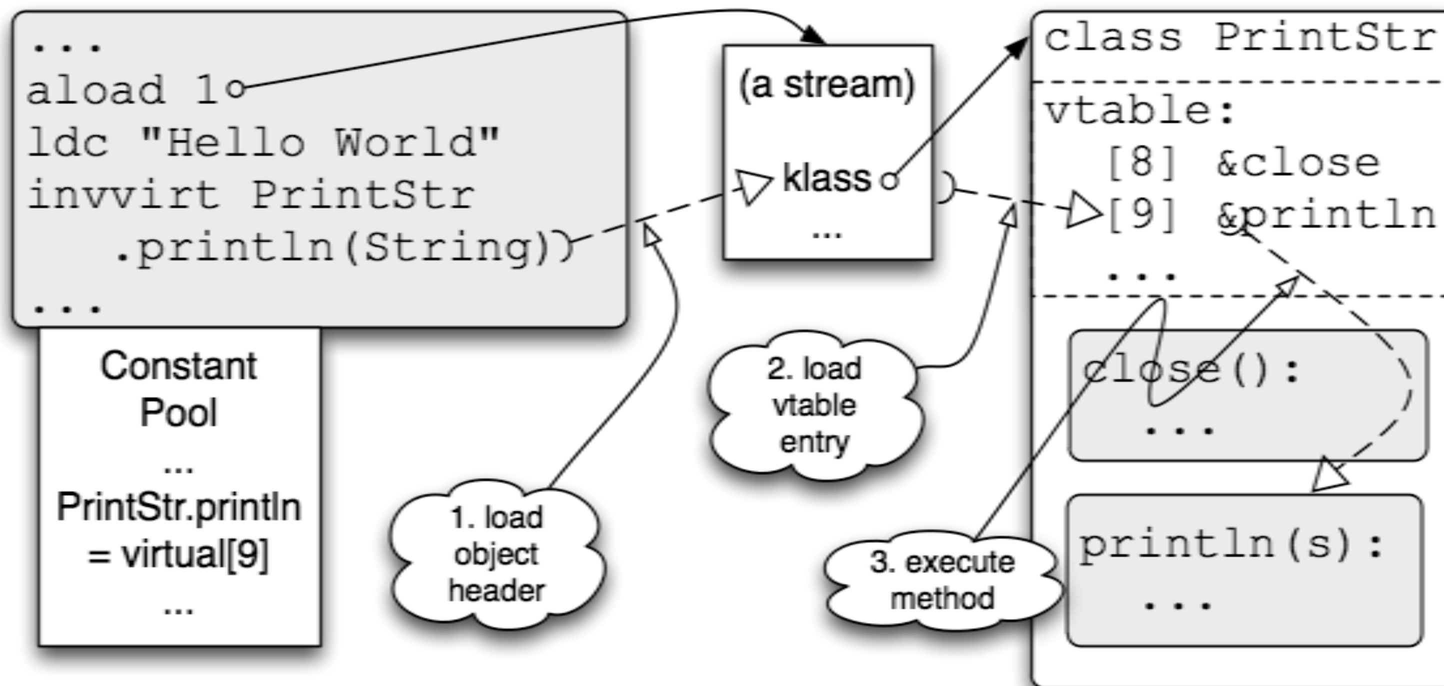
```
//PrintStream out = System.out;
out.println("Hello World");
```

The compiled byte code looks like

```
4:   aload_1
5:   ldc #2              //String "Hello World"
7:   invokevirtual #4  //Method java/io/PrintStream.println:
                                   (Ljava/lang/String;)V
```

a) Again, names in bytecode
b) Again, linking fixed by JVM
c) *Only* the receiver type determines method selection
d) *Only* the receiver type can be adapted (narrowed)

How the VM selects the target method:



(Note: This implementation is typical; VMs vary.)

What more could anybody want? (1)

- Naming — not just Java names
  - arbitrary strings, even structured tokens (XML??)
  - help from the VM resolving names is ***optional***
  - caller and callee do ***not*** need to agree on names
- Linking — not just Java & VM rules
  - can link a call site to any callee the runtime wants
  - can *re-link* a call site if something changes
- Selecting — not just static or receiver-based
  - selection logic can look at any/all arguments
  - (or any other conditions relevant to the language)

What more could anybody want? (2)

- Adapting — no exact signature matching
  - widen to Object, box from primitives
  - checkcast to specific types, unbox to primitives
  - collecting/spreading to/fromvarargs
  - inserting or deleting extra control arguments
  - language-specific coercions & transformations

- *(…and finally, the same fast control transfer)*

- *(…with inlining in the optimizing compiler, please)*

Dynamic method invocation

- How would we compile a function like

```
function max(x, y) {
  if (x.lessThan(y)) then y else x
}
```

- Specifically, how do we call `.lessThan()`?

Dynamic method invocation (how not to)

- How about:

```
0:   aload_1; aload_2
2:   invokevirtual #3  //Method Unknown.lessThan:
                                            (LUnknown;)Z
5:   if_icmpeq
```

- That doesn't work
  - No receiver type
  - No argument type
  - Return type might not even be boolean (‘Z’)

Dynamic method invocation (how to)

> A new option:

```
0:   aload_1; aload_2
2:   invokedynamic #3  //NameAndType lessThan:
                    (Ljava/lang/Object;Ljava/lang/Object;)Z
5:   if_icmpeq
```

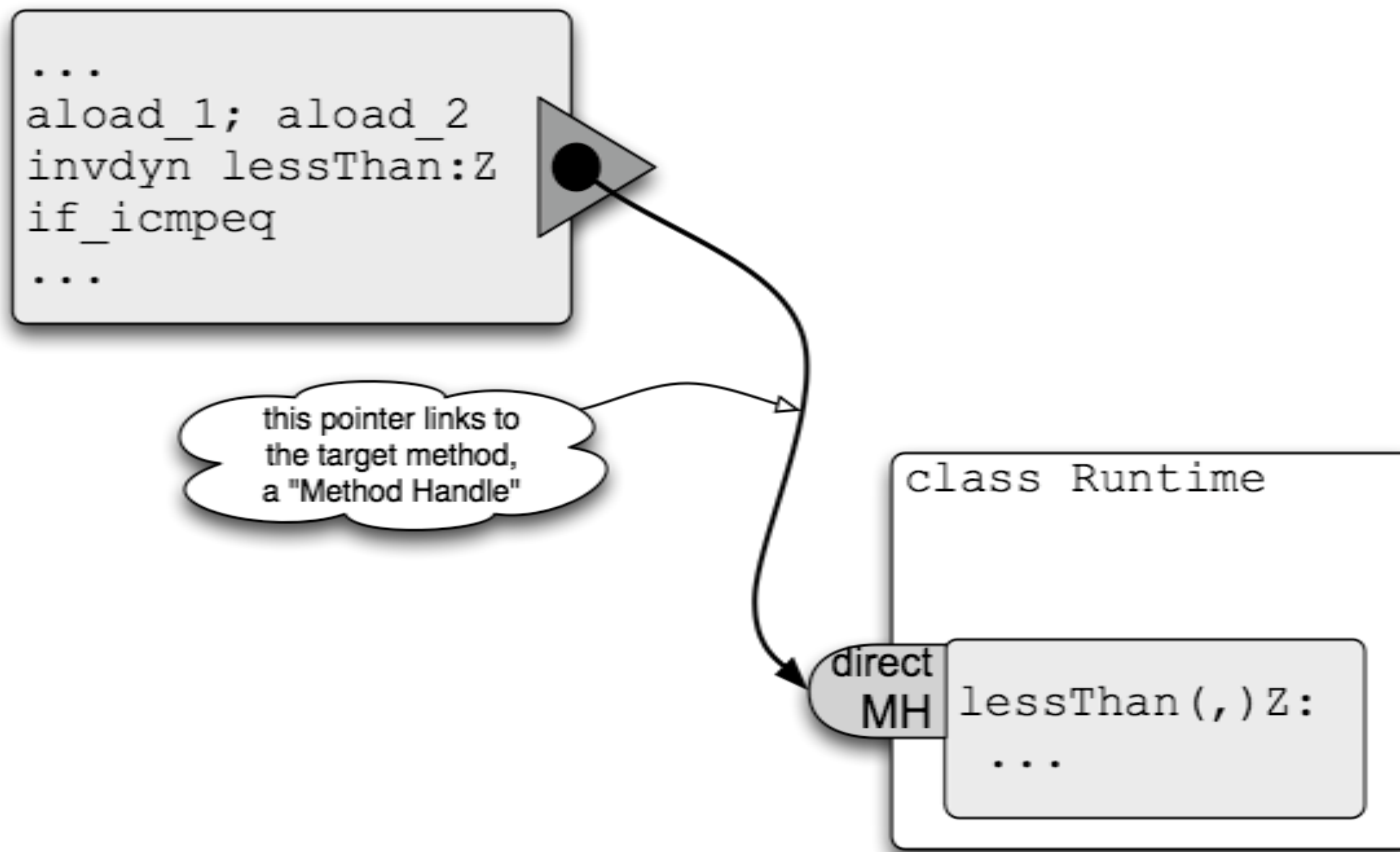
> Advantages:

- Compact representation
- Argument types are untyped Objects
- Required boolean return type is respected

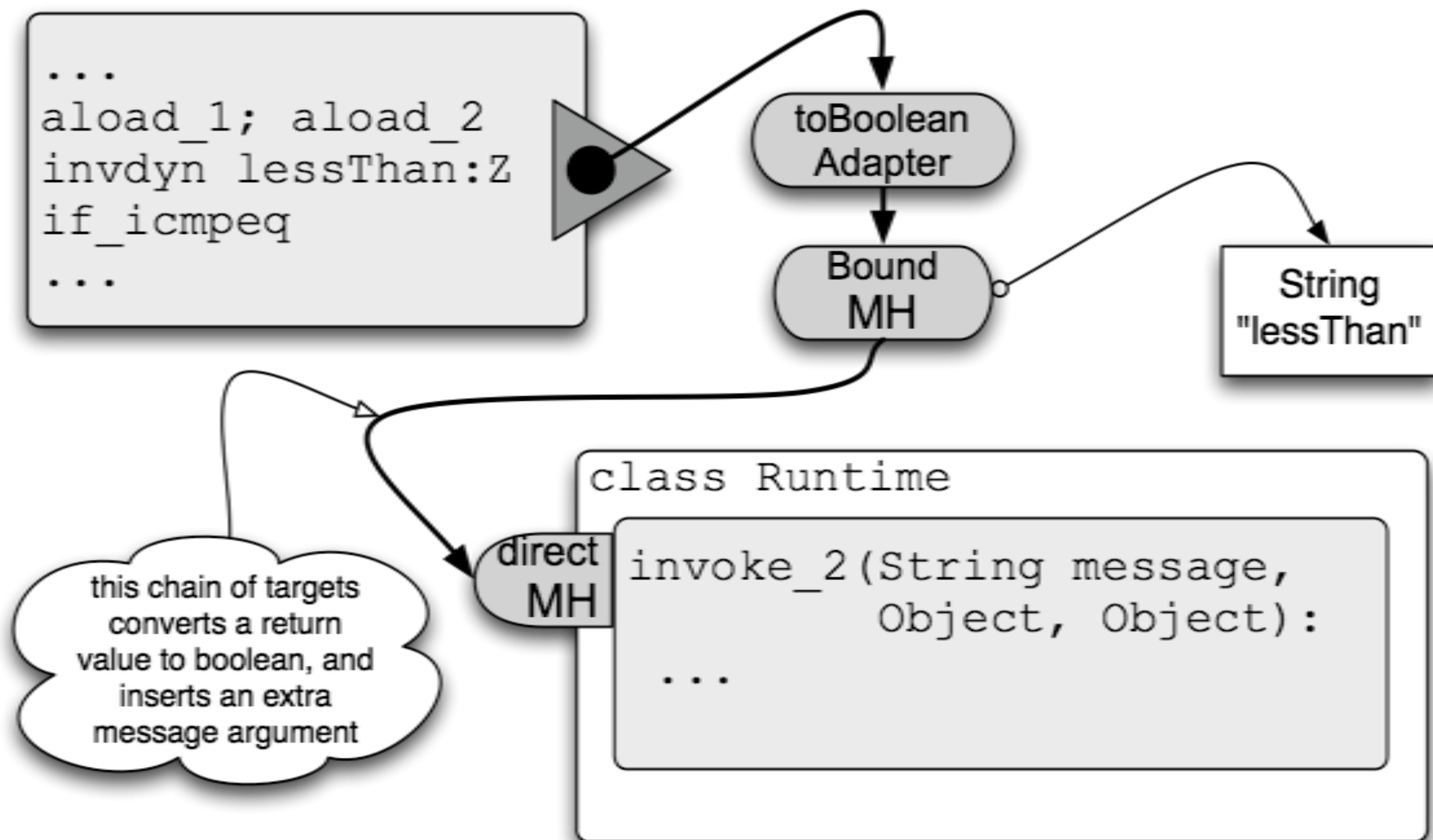
Dynamic method invocation (details)

- But where is the dynamic language plumbing??
  - We need something like **`invoke_2`** and **`toBoolean`**!
  - How does the runtime know the name **`lessThan`**?
- Answer: it's all *method handles* (MH).
  - A MH can point to any accessible method
  - (A MH can do normal receiver-based dispatch)
  - The target of an invokedynamic is a MH

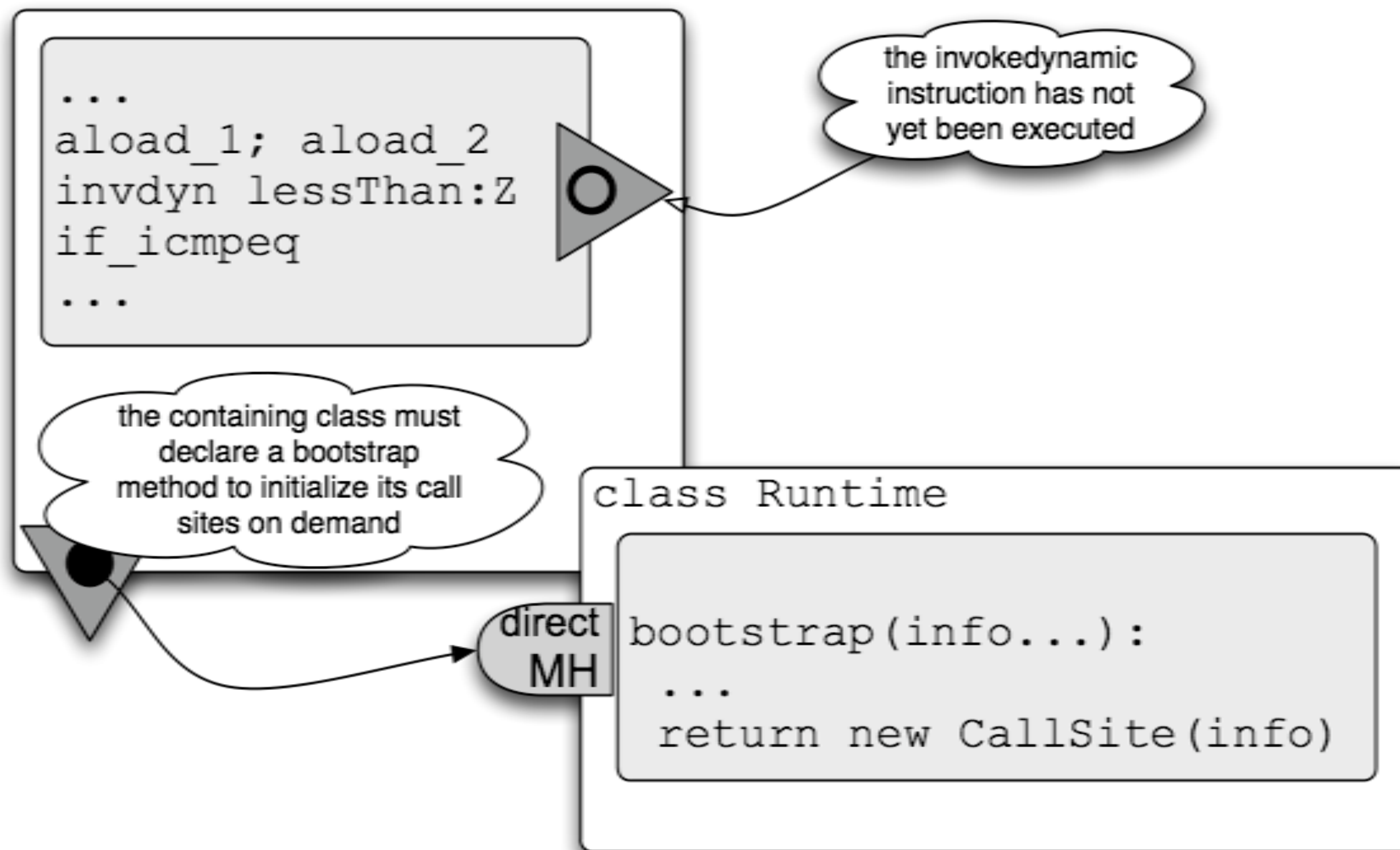
invokedynamic, as seen by the VM:



more invokedynamic plumbing: “adapters”



meta-plumbing: the bootstrap method



A Budget of Invokes

invoke-static	**invoke-special**	**invoke-virtual**	**invoke-interface**	**invoke-dynamic**
no receiver	receiver	receiver class	receiver interface	no receiver
no dispatch	no dispatch	single dispatch	single dispatch	adapter-based dispatch
B8 ***nn nn***	**B7** ***nn nn***	**B6** ***nn nn***	**B9** ***nn nn aa*** 00	**BA** ***nn nn*** 00 00

Speed, speed, and speed

- We are doing this to make languages run faster
  - more compact code and runtimes
  - simpler, more direct “plumbing” of calls
  - *direct application of existing VM optimizations!*
- The Java VMs have *centuries* of programmer effort invested in making the bytecodes *zoom*.
  - (And zoom scalably and reliably, too.)
- Why should Java have all the fun?

Method handles

- Like methods, can have any function type
- An object of static type **`java.dyn.MethodHandle`**
- Unlike (other) objects, signature-polymorphic
- Like methods, can be virtual, static, or “special”
- Unlike methods, not named
- Invoked like methods:
  - **`mh.invoke(`*`args`*`)`** — for exact signature match (JVM native)
  - **`MethodHandles.invoke(mh, `*`args`*`)`** — reflective style

Goals for invokedynamic

- Programmable linkage that behaves like reflection
- Call sites that look like regular method calls
  - Fast use of any signature
  - Type-safe
  - Inlinable
- … to a flexibly specified target implementation
  - Specified by language-specific "plug in"

Early user experience…

http://pylonshq.com/irclogs for #mlvm,2009-05-26

jrose One thing that would be nice to know for my Thursday talk is how much JRuby code you think you can throw away (make inactive) by using MH/indy.

headius probably 80-90% of call pipeline

headius remember we still generate our own method handle classes for all core classes right now

headius that's over 1000 tiny classes in jruby binary dist

headius all can disappear

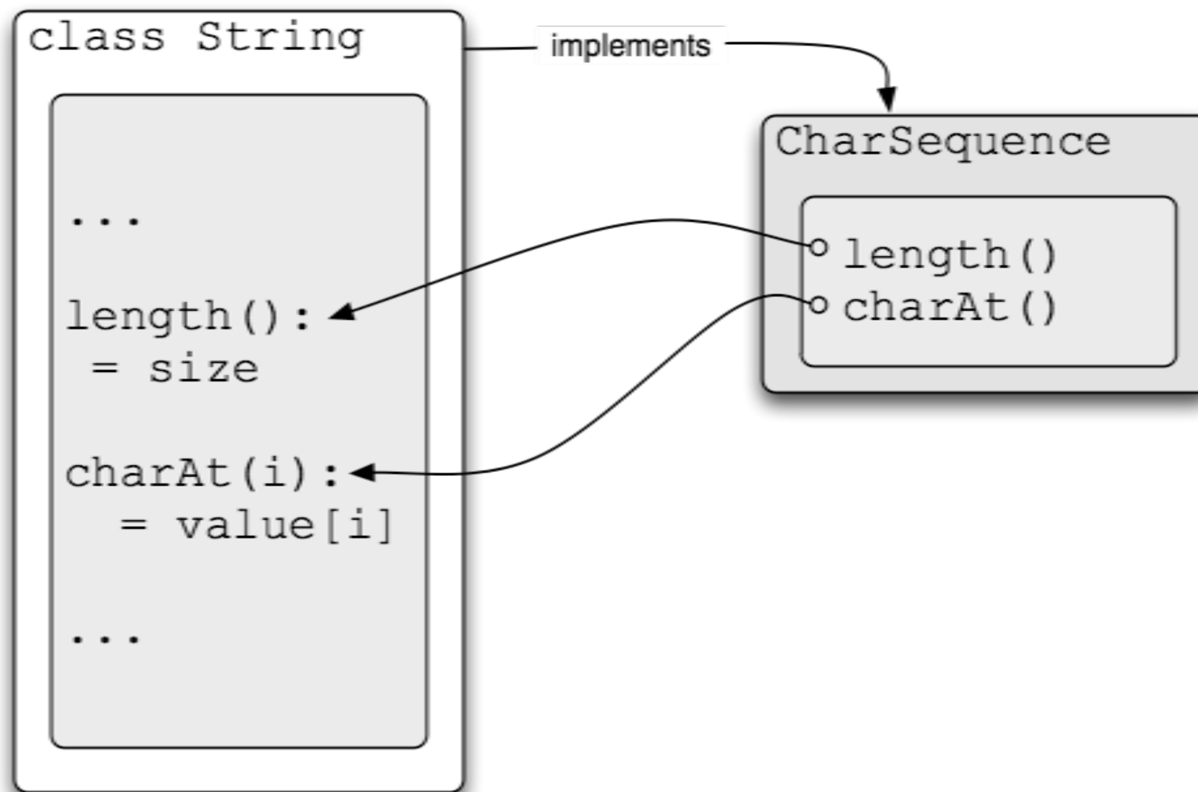
JSR 292 – the “invokedynamic” JSR

- Original charter of JSR-292 was to provide a specific bytecode for dynamic method invocation, plus (optionally) mechanisms for code evolution
- Current work includes
  - **`invokedynamic`** bytecode
    - Application-defined linkage (and re-linkage)
  - Method handles
    - All-purpose lightweight construct for "code pointers"
  - Interface injection (likely)
    - Add new methods and types to existing classes
- More later, we hope!

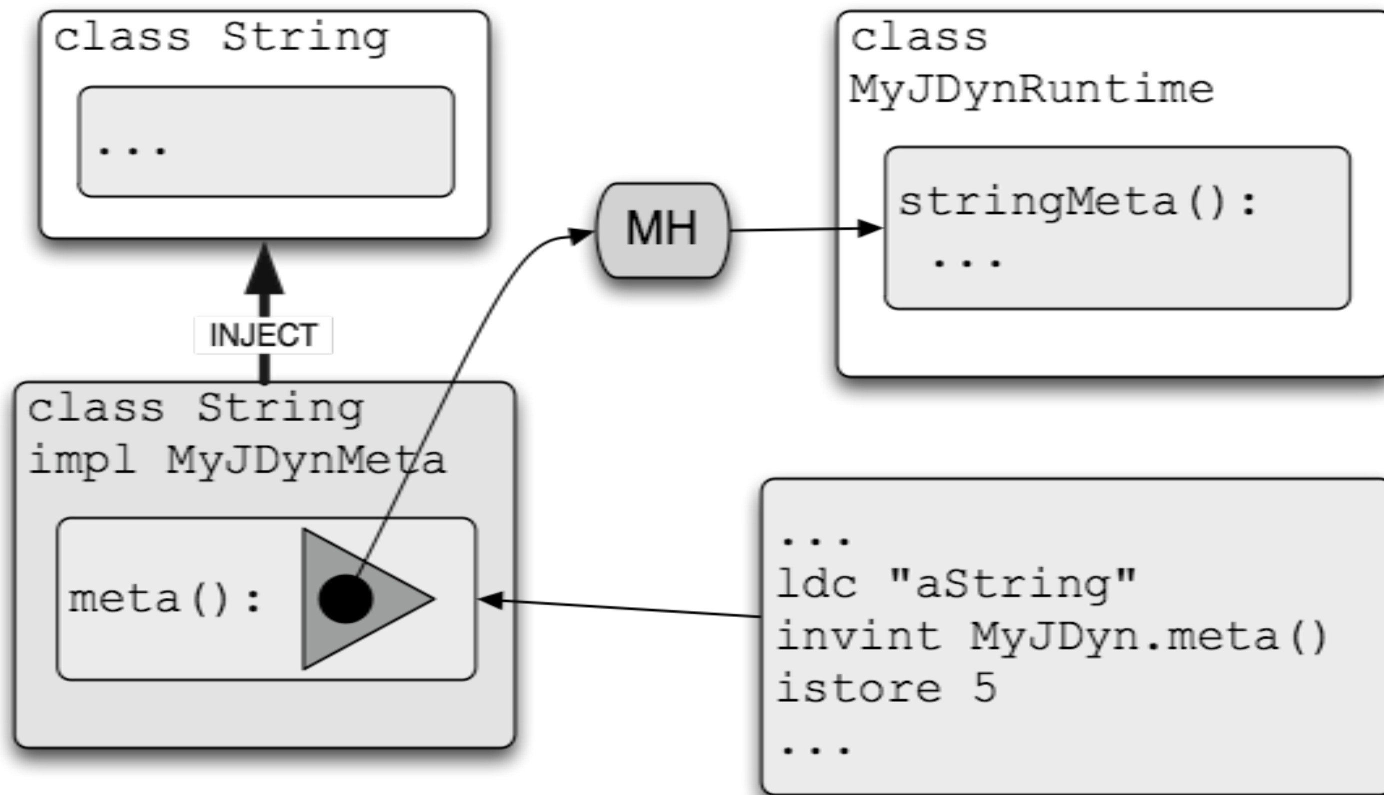
Interface injection

- Dynamically typed programs can be self-modifying
  - Class or method definition can change at runtime
    - Self-modifying code is hard to optimize, and dangerous
- So, don't restructure classes, just relabel them
  - Enter Interface injection
  - Limited ability to modify old classes
    - Just enough for them to implement new interfaces
  - interface supertypes are cheap for JVM objects
  - **`invokeinterface`** is fast these days

a statically defined interface:



an injected interface:



Tail-calls

- **What:** Classic call/return pattern, optimized.
  - A hard guarantee, never to blow the control stack.
- **Why:** Some languages require it.
  - Scheme, Haskell (functional programming)
  - Also, some code shapes need it
    (threaded interpreters, open-ended state machines, partially compiled methods)

more Da Vinci Machine subprojects

…in a future we hope for!

- fixnums – tagged immediate pseudo-pointers
  - `http://blogs.sun.com/jrose/entry/fixnums_in_the_vm`
- tuple types – primitive structs, structure-based identity
  - `http://blogs.sun.com/jrose/entry/tuples_in_the_vm`
- mixed arrays – fused hybrid of instance, struct, arrays
- new load units – modules, partial classes, shared images
- *what else?*

Fixnums

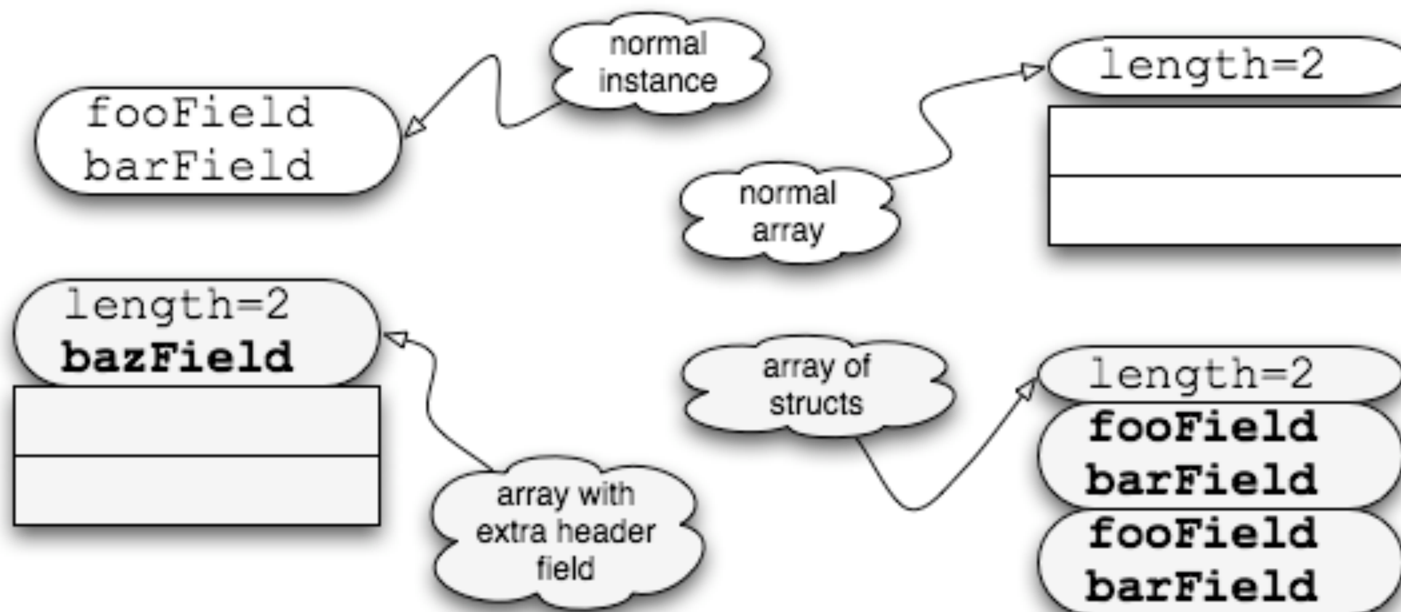
- **What:** Optimization of autoboxing (`Integer.valueOf`).
  - Tagged pointer, carrying 24 to 63 bits of immediate data
  - No indirections, no memory usage
  - Good for all primitive wrapper types (except maybe float/double)
- **Why:** Dynamic languages need primitives too.
  - But they need to interconvert efficiently with `Object`
  - JIT escape analysis and box analysis not systemic enough

Tuples and value types

- **What:** Data without state or identity.
  - Pass directly in multiple registers.
  - No side effects, ever.
  - Tuples, numeric types, immutable collections.
- **Wait:** Are they objects too? (Can go in Lists?)
  - Yes, allow references to “boxes” in heap.
  - Adjust “==” to perform structure comparison.
- **Why:** Languages need compact structs/tuples.
  - Numeric people want Complex, Rational, etc.
  - Even if it’s not in Java, the JVM has to help.

Mixed arrays

- **What:** An array fused onto the tail of an instance
- **Why:** Building block for data structures
  - fewer pointers, indirections, dependent loads



Still other projects waiting for attention…

- retarget your favorite runtime to invokedynamic
- retarget your favorite compiler to use method handles
- make your favorite language use the new features

Still other projects waiting for attention…

> bug finding/fixing
> unit tests

`http://hg.openjdk…/mlvm/…/netbeans/meth/test/…`

> ports (assembly language required)

`http://hg.openjdk…/mlvm/hotspot/…/indy-amd64.patch`

Yet more projects…

> JSR 292 Backport

`http://code.google.com/p/jvm-language-runtime`

> Dynalang MOP (metaobject protocol)

`http://dynalang.sourceforge.net`

None of these are easy…

But is one of them yours?

The Da Vinci Machine Project:
Collaborating on JVM™ Futures

John Rose

http://openjdk.java.net/projects/mlvm