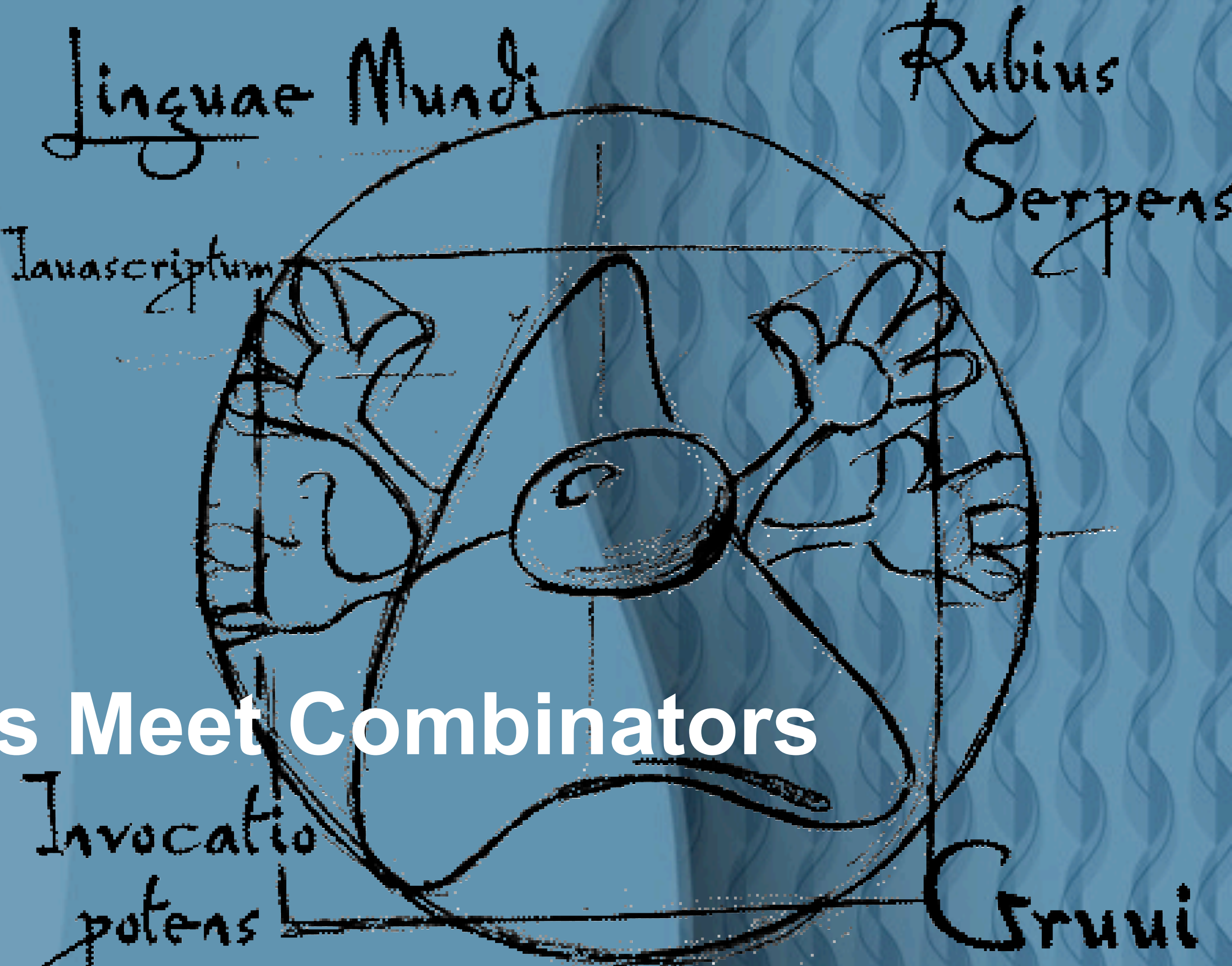


Bytecodes Meet Combinators

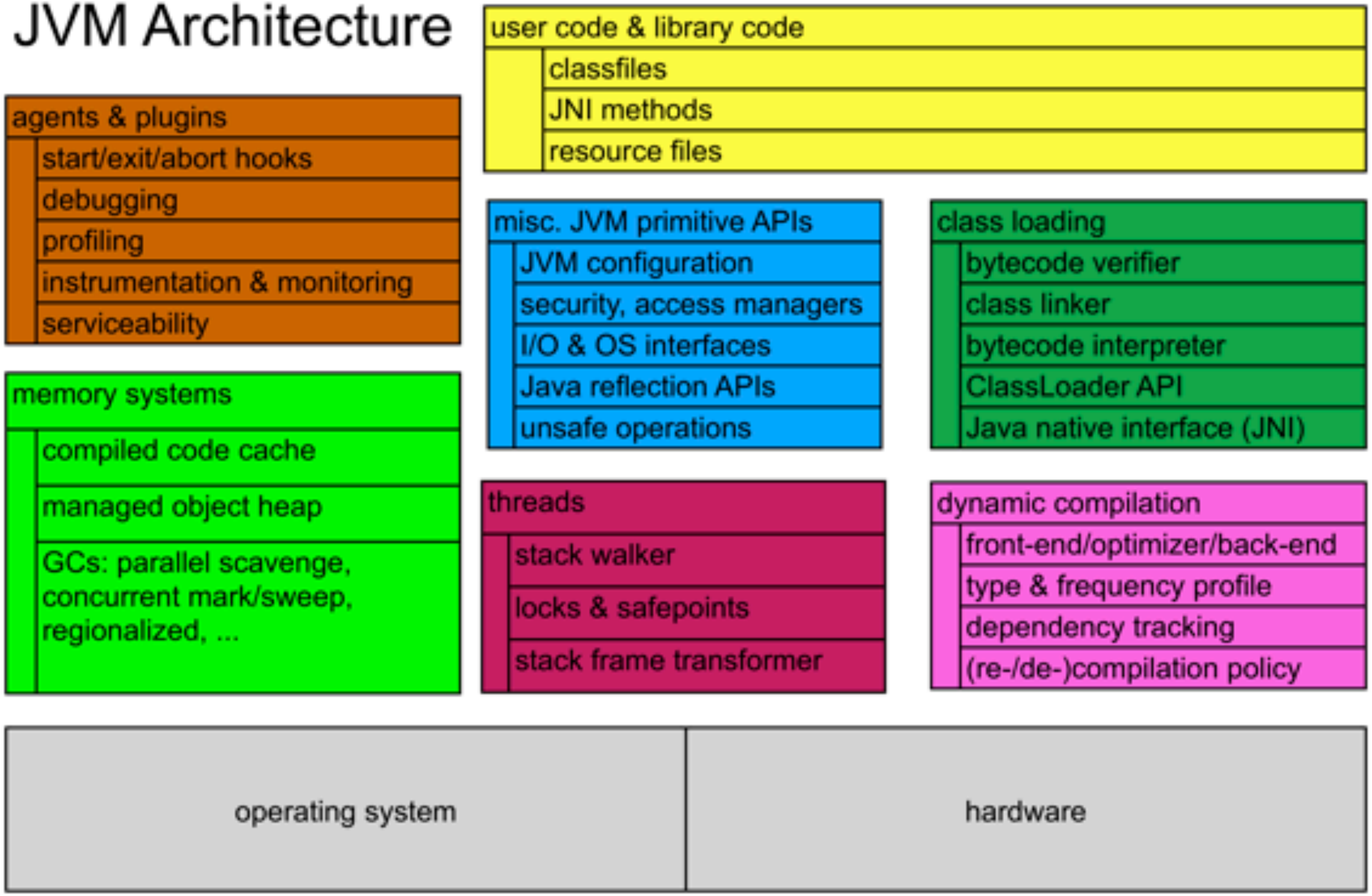
John Rose
Architect, Da Vinci Machine Project
Sun Microsystems

VMIL 2009: 3rd Workshop on Virtual Machines and Intermediate Languages
(OOPSLA, Orlando, Florida October 25, 2009)
http://cr.openjdk.java.net/~jrose/pres/200910-BcsMeetCmbs.pdf

Overview

- The JVM is useful for more languages than Java
- Bytecodes have been successful so far, but are showing some “pain points”
- Composable “function pointers” fill in a key gap

JVM Architecture



“Java is slow because it is interpreted”

- Early implementations of the JVM executed bytecode with an interpreter [slow]

“Java is fast because it runs on a VM”

- Major breakthrough was the advent of “Just In Time” compilers [fast]
  - Compile from bytecode to machine code at runtime
  - Optimize using information *available at runtime only*
- Simplifies static compilers
  - javac and ecj generate “dumb” bytecode and trust the JVM to optimize
  - Optimization is *real*...
    ...even if it is invisible

HotSpot optimizations

compiler tactics
- delayed compilation
- tiered compilation
- on-stack replacement
- delayed reoptimization
- program dependence graph representation
- static single assignment representation

proof-based techniques
- exact type inference
- memory value inference
- memory value tracking
- constant folding
- reassociation
- operator strength reduction
- null check elimination
- type test strength reduction
- type test elimination
- algebraic simplification
- common subexpression elimination
- integer range typing

flow-sensitive rewrites
- conditional constant propagation
- dominating test detection
- flow-carried type narrowing
- dead code elimination

language-specific techniques
- class hierarchy analysis
- devirtualization
- symbolic constant propagation
- autobox elimination
- escape analysis
- lock elision
- lock fusion
- de-reflection

speculative (profile-based) techniques
- optimistic nullness assertions
- optimistic type assertions
- optimistic type strengthening
- optimistic array length strengthening
- untaken branch pruning
- optimistic N-morphic inlining
- branch frequency prediction
- call frequency prediction

memory and placement transformation
- expression hoisting
- expression sinking
- redundant store elimination
- adjacent store fusion
- card-mark elimination
- merge-point splitting

loop transformations
- loop unrolling
- loop peeling
- safepoint elimination
- iteration range splitting
- range check elimination
- loop vectorization

global code shaping
- inlining (graph integration)
- global code motion
- heat-based code layout
- switch balancing
- throw inlining

control flow graph transformation
- local code scheduling
- local code bundling
- delay slot filling
- graph-coloring register allocation
- linear scan register allocation
- live range splitting
- copy coalescing
- constant splitting
- copy removal
- address mode matching
- instruction peepholing
- DFA-based code generator

Feedback multiplies optimizations

- On-line profiling and CHA produces information
- ...which lets the JIT ignore unused paths
- ...and helps the JIT sharpen types on hot paths
- ...which allows calls to be devirtualized
- ...allowing them to be inlined
- ...expanding an ever-widening optimization horizon

Result: Large nmethods containing tightly optimized machine code for hundreds of inlined calls.

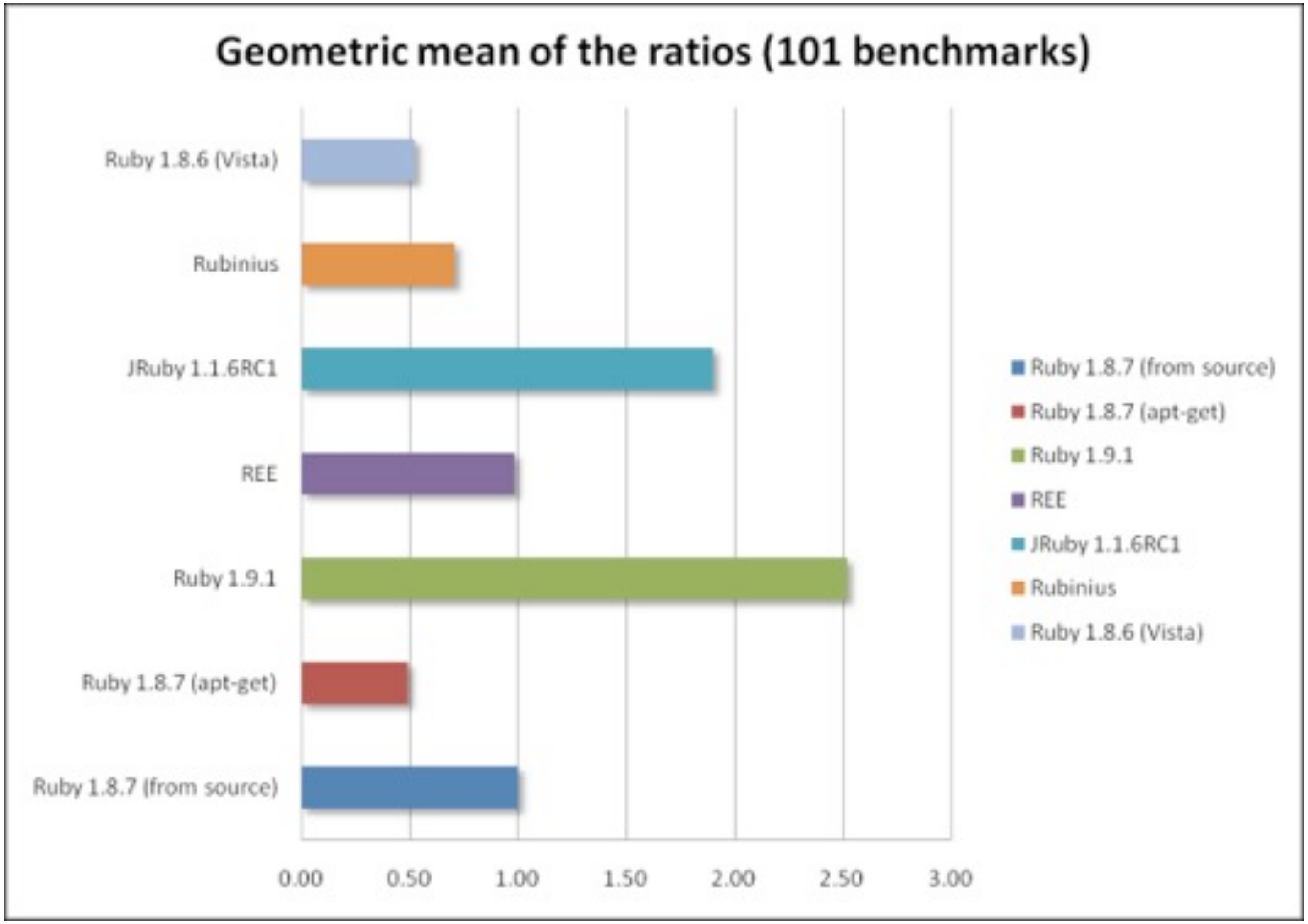
Why should Java have all the fun?

These optimizations apply very well elsewhere

- Static languages: JavaFX, Scala, ...
- Dynamic languages: JRuby, JavaScript, Python, ...
- Functional languages: Clojure, Scala (again)

“Ruby is slow because it is interpreted”

The Great Ruby Shootout 2008



2.00 means "twice as fast"

0.50 means "half the speed"

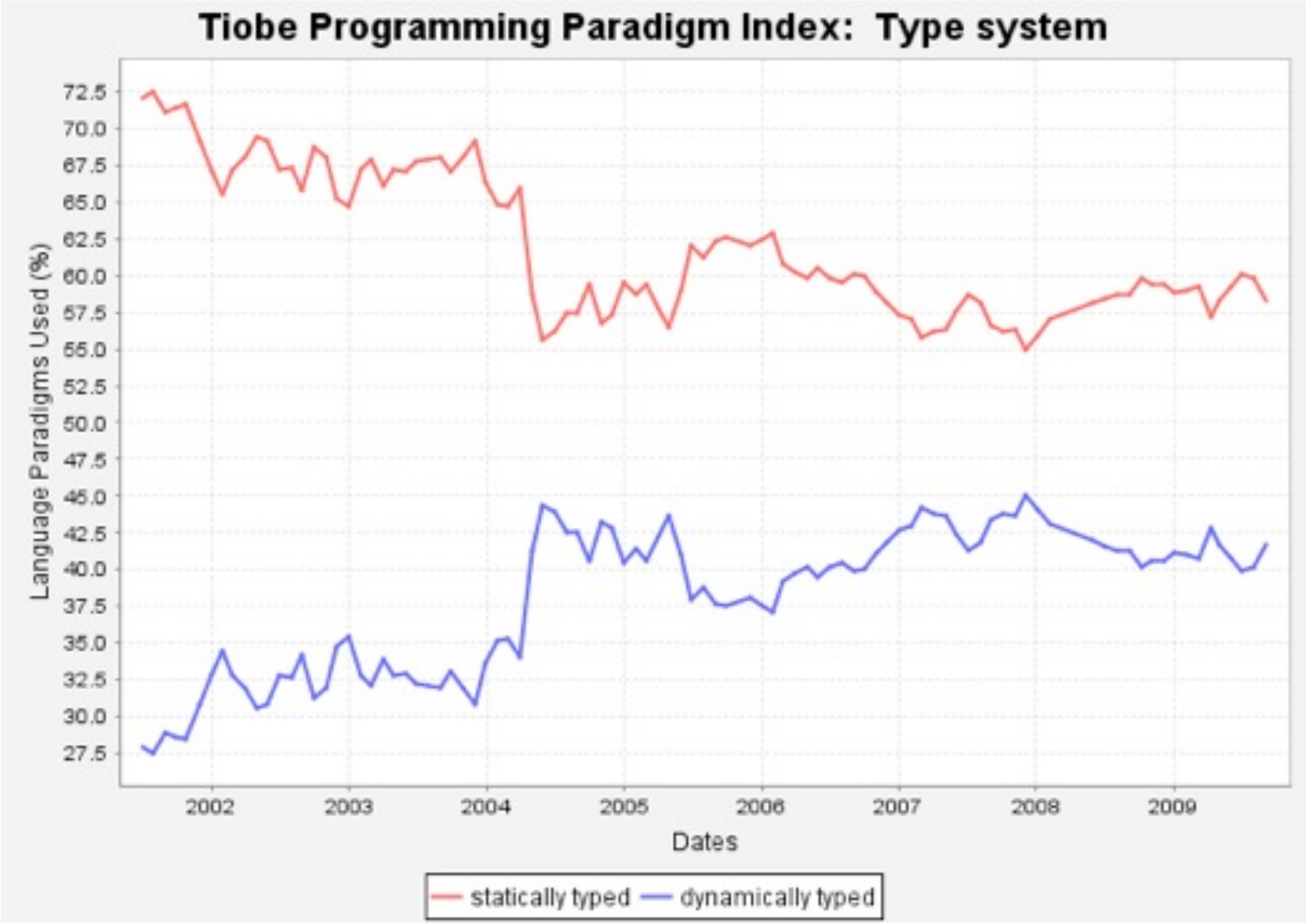
http://antoniocangiano.com/2008/12/09/the-great-ruby-shootout-december-2008/

Languages ♥ The JVM

Tea
iScript
Jickle
Zigzag
JESS
Modula-2
CAL
Lisp
JavaScript
Correlate
Nice
Basic
JudoScript
Simkin
Drools
Groovy
Eiffel
Icon
v-language
Luck
Pascal
Prolog
Mini
Scala
Tcl
PLAN
Hojo
Rexx
JavaFX Script
Funnel
FScript
Tiger
Anvil
Yassl
Oberon
E
Smalltalk
JRuby
Logo
Tiger
JHCR
Ada
G
Clojure
Scheme
Sather
Phobos
Processing
WebL
Dawn
TermWare
Sleep
LLP
Pnuts
Bex Script
BeanShell
Forth
C#
PHP
SALSA
ObjectScript
Jython
Yoix
Piccola

Java™

Different *kinds* of languages



Data courtesy of TIOBE: www.tiobe.com

JVM Specification, 1997

The Java Virtual Machine knows nothing about the Java programming language, only of a particular binary format, the class file format.

A class file contains Java Virtual Machine instructions (or bytecodes) and a symbol table, as well as other ancillary information.

Any language with functionality that can be expressed in terms of a valid class file can be hosted by the Java virtual machine.

Attracted by a generally available, machine-independent platform, implementors of other languages are turning to the Java Virtual Machine as a delivery vehicle for their languages.

In the future, we will consider bounded extensions to the Java Virtual Machine to provide better support for other languages.

Predicting the future is not easy...

- Hardware: Asymmetric MC? RISC? VPU ? Fibers?
- Memory: NUMA? Flash tier? (S)TM? Functional?
- Systems: Central? Mobile? Embedded? Realtime?
- Languages: Parallel? Reactive? Dynamic?
- Programming: Script-based? Tool-based?
- Java Franchise: Java++? Other Java companies?

Key bet: Run dynamic languages

- Scripting is here to stay.

  PHP, JavaScript, Ruby, Python

  Key value: Flexibility (informal notations)

- Late binding is getting cheaper
  (late composition getting more common)

- The JVM must interoperate with such languages.

- Tactic: JSR 292, the “invokedynamic” bytecode.

Bytecode Strengths (historical)

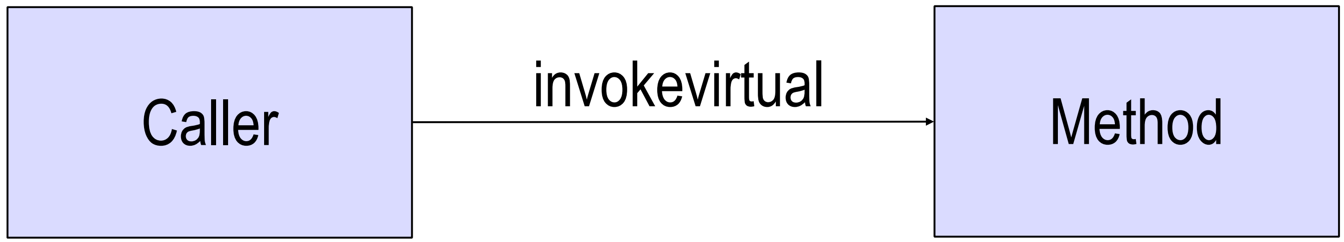
- Compact
- Portable
- Verifiable
- Composable (linkage through names, everywhere)
- Flexible: *Both* interpretable and compilable

=> many machine cycles run via JVM bytecodes

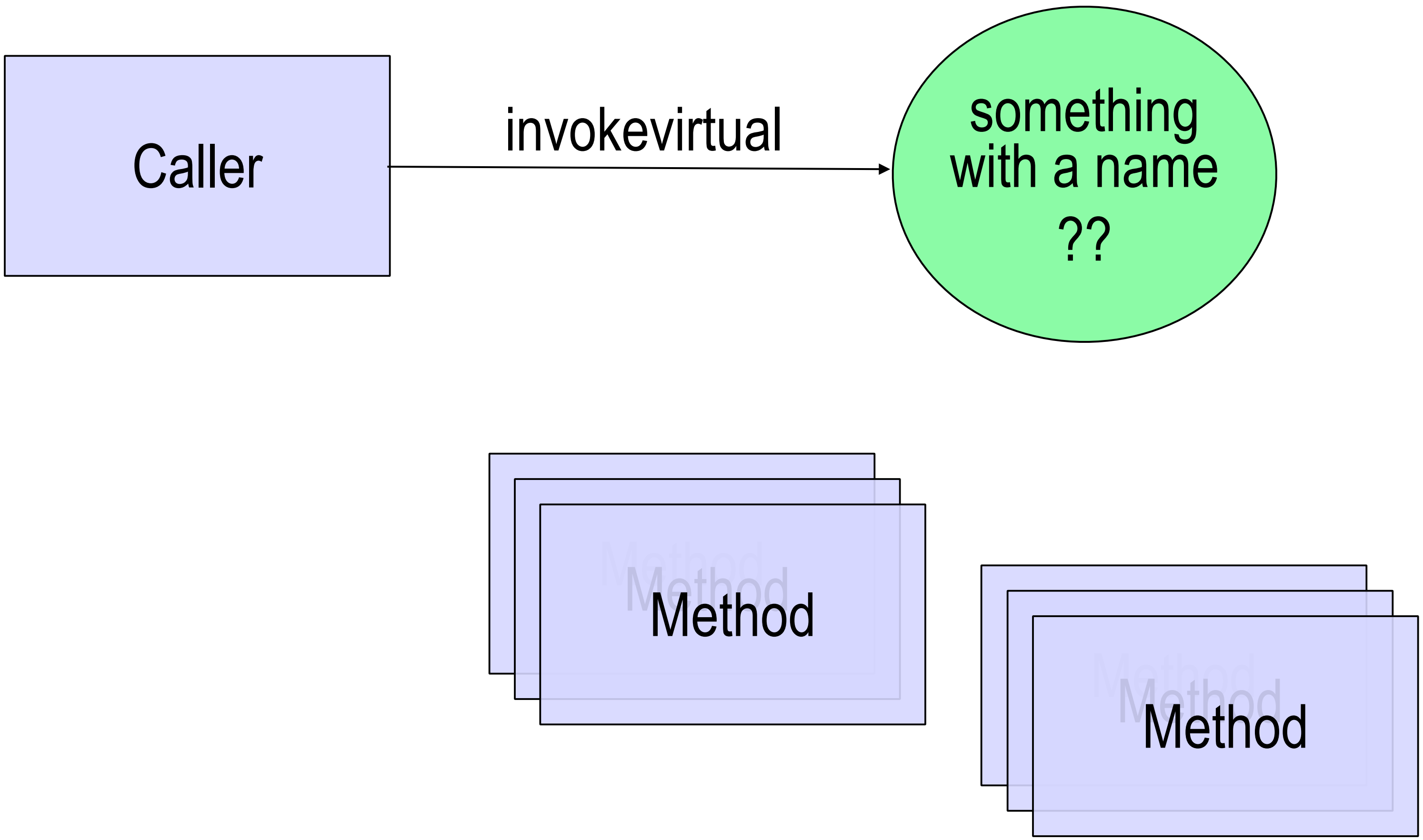
Bytecode Pain Points

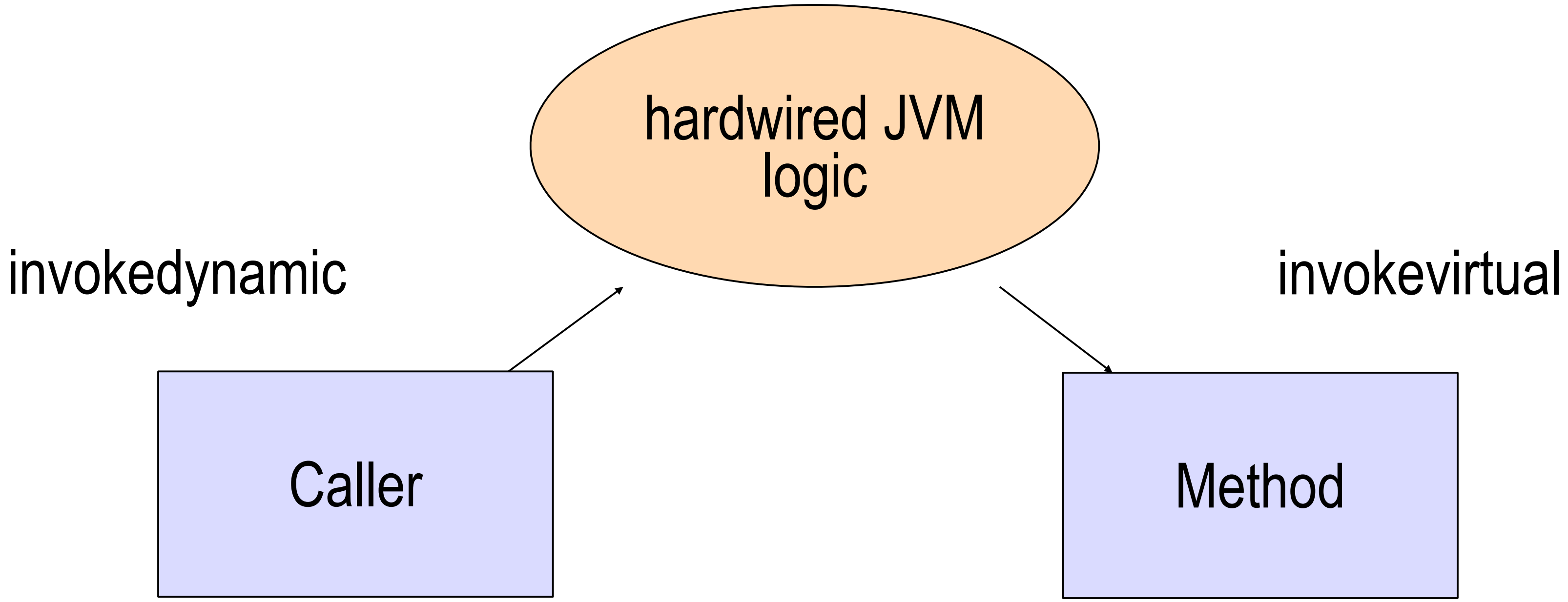
- Designed for one language (starts with a “J”)
- No “join points” other than name resolution
- Name resolution is not (very) programmable
- Changes require reassembly
- => reverification, recompilation, reoptimization
- Small devices object to bytecode weaving

Key composition: Invoke a method



Initial state is unlinked




hardwired JVM
logic
invokedynamic
invokevirtual
Caller
Method

Digression: Details of Method Calls

The next group of slides, headlined “JavaOne”, are taken from Brian Goetz’s and John’s JavaOne 2009 talk, Towards a Renaissance VM. These slides were omitted from the actual VMIL presentation.

The full talk is available:

http://cr.openjdk.java.net/~jrose/pres/200906-RenaisVM.pdf

The related talk “JSR 292 Cookbook” is here:

http://cr.openjdk.java.net/~jrose/pres/200906-Cookbook.htm

The deal with method calls (in one slide)

- Calling a method is cheap (VMs can even inline!)
- Selecting the right target method can be costly
  - Static languages do most of their method selection at compile time (e.g., `System.out.println(x)`)
    - Single-dispatch on receiver type is left for runtime
  - Dynamic langs 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 (& re-decide) linkage

What’s in a method call?

> Naming — using a symbolic name
> Linking — reaching out somewhere else
> Selecting — deciding which one to call
> Adapting — agreeing on calling conventions

> *(…and finally, a parameterized control transfer)*

A connection from caller A to target B

- Including naming, linking, selecting, adapting:
- …where B might be known to A only by a name
- …and A and B might be far apart
- …and B might depend on arguments passed by A
- …and a correct call to B might require adaptations

- *(After everything is decided, A jumps to B’s code.)*

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
```

Example: Fully static invocation

- For this source code

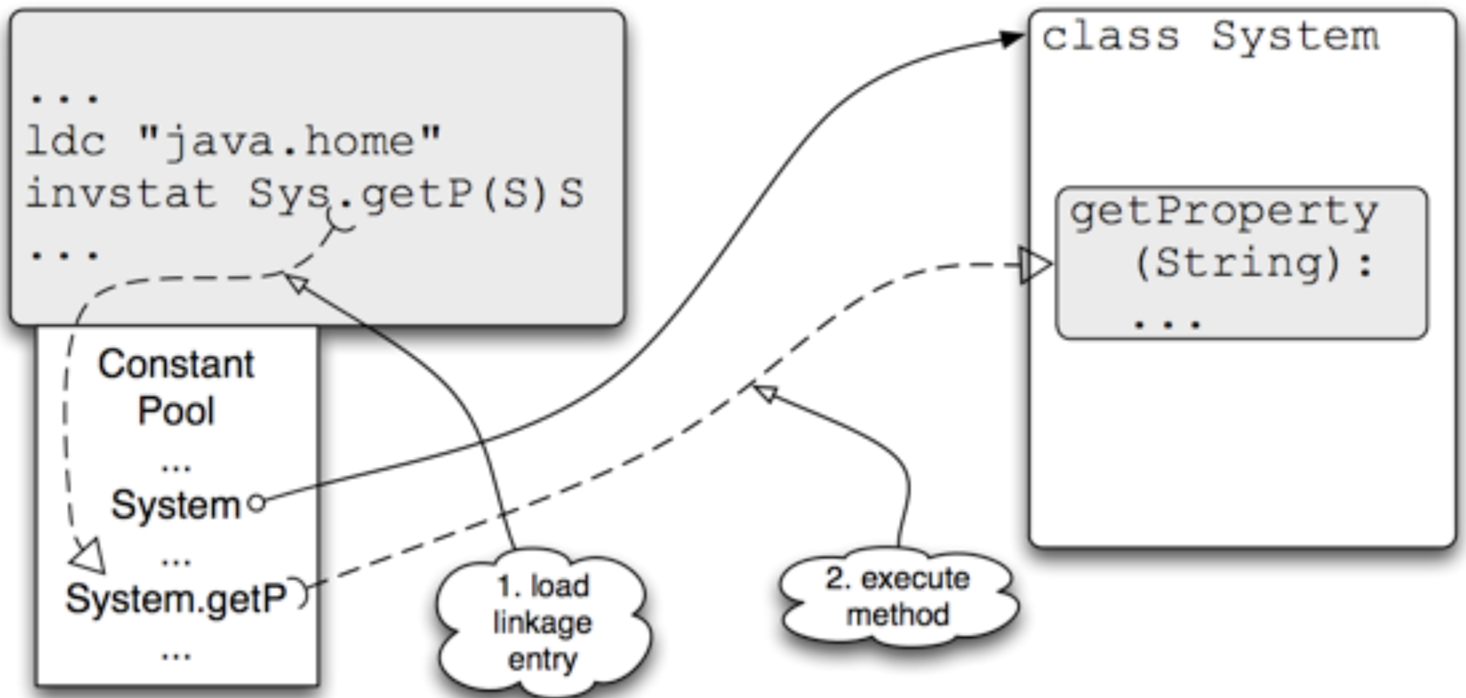
```
String s = System.getProperty("java.home");
```

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
```

- Names are embedded in the bytecode
- Linking handled by the JVM with fixed Java rules
- Target method selection is not dynamic at all
- 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
```

Example: Class-based single dispatch

- For this source code

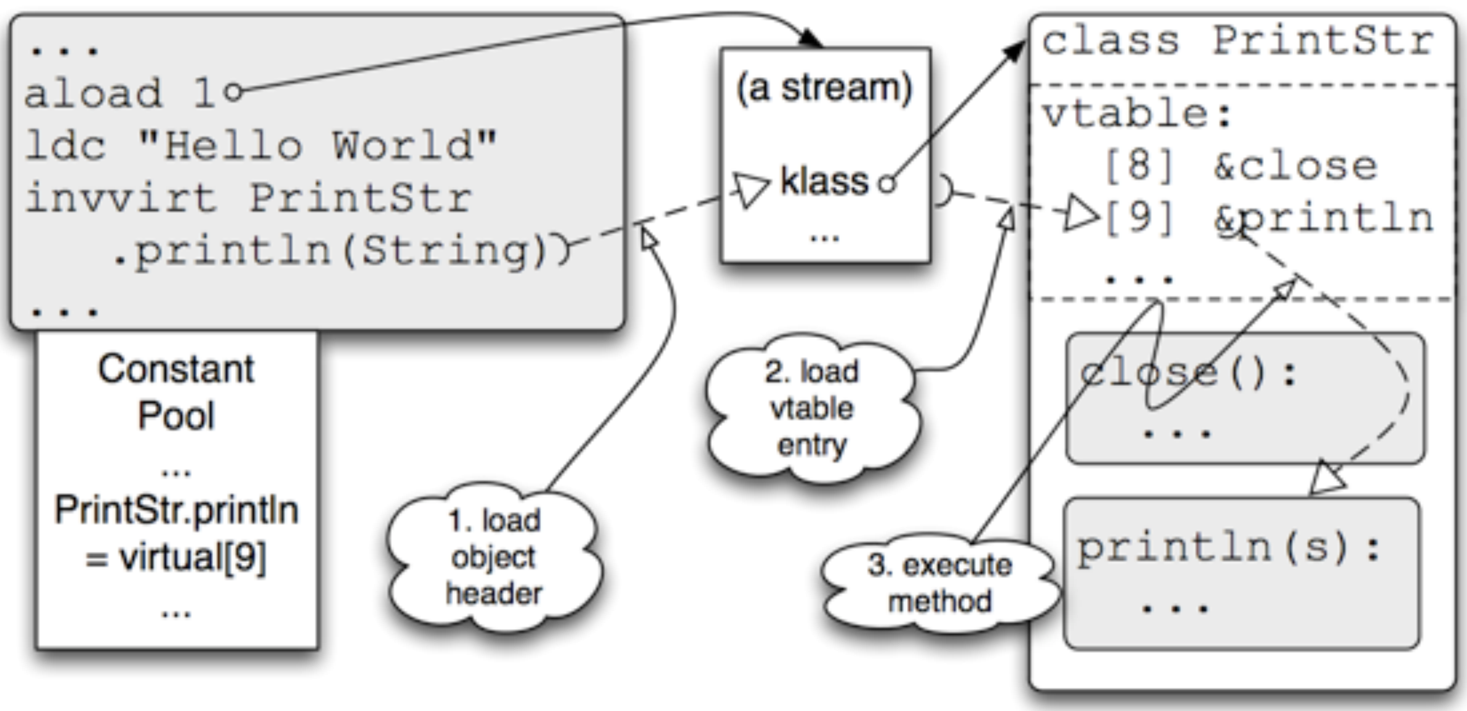
```
//PrintStream out = System.out;
out.println("Hello World");
```

- 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/from varargs
  - 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
}
```

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
  ```

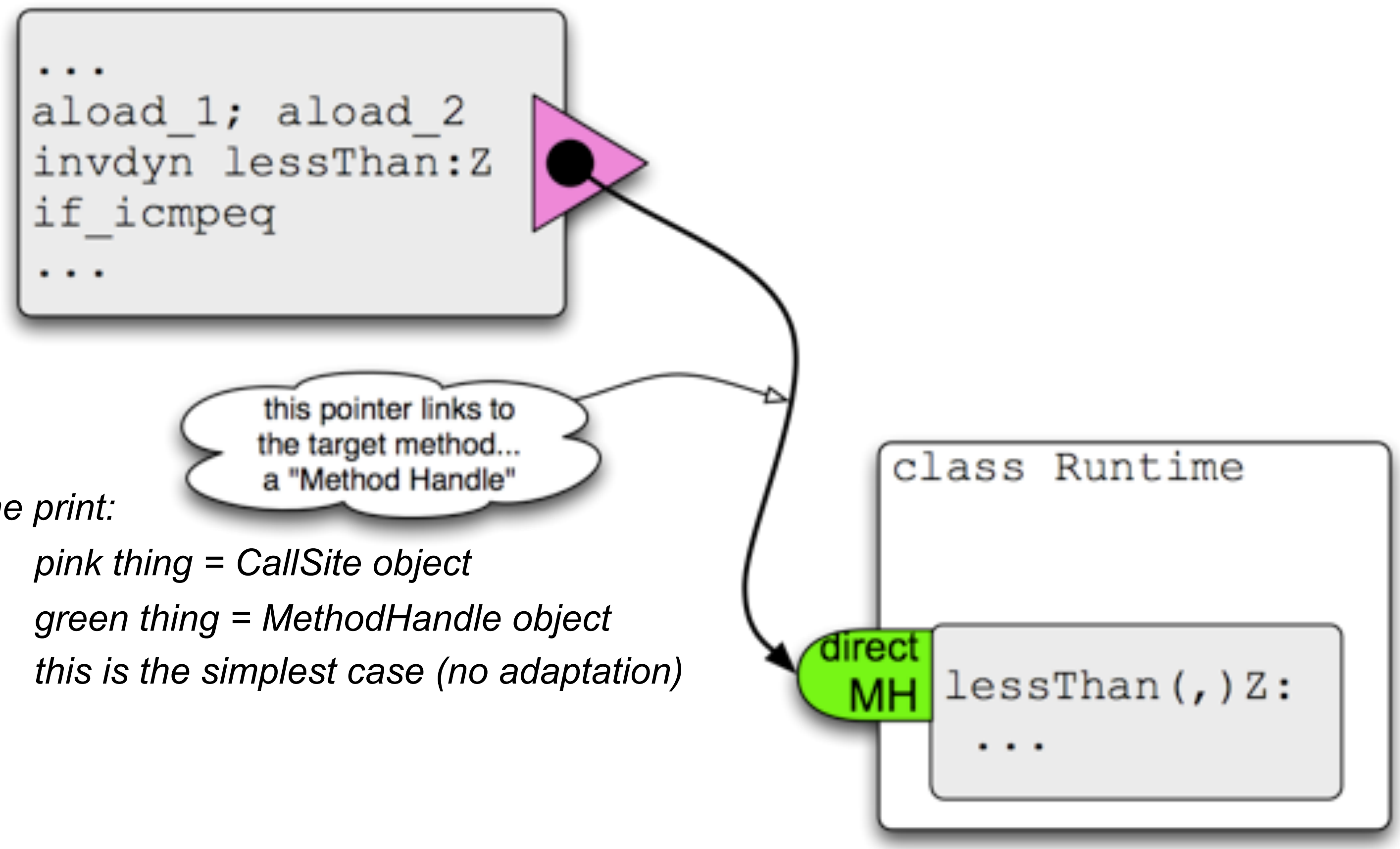
- That doesn't work
  - No receiver type
  - No argument type
  - Return type might not even be boolean (‘Z’)
  ```

Dynamic method invocation (slowly)

- How about:

```
0:    aload_1; aload_2
2:    ldc #2   // "lessThan"
4:    invokestatic #3  //Method my/Runtime.invoke_2:
  (Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;
5:    invokestatic #4  //Method my/Runtime.toBoolean:
                                      (Ljava/lang/
Object;)Z
8:    if_icmpeq
```

- That works, but … really … slowly
  - Argument types are (re-)computed remotely
  - Less possibility of local caching or optimization
  - Lots of indirections!
- ("Invoker" objects? Similar problems + complexity.)

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
  - (Flexibility from ***signature polymorphism***.)
  ```

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`?

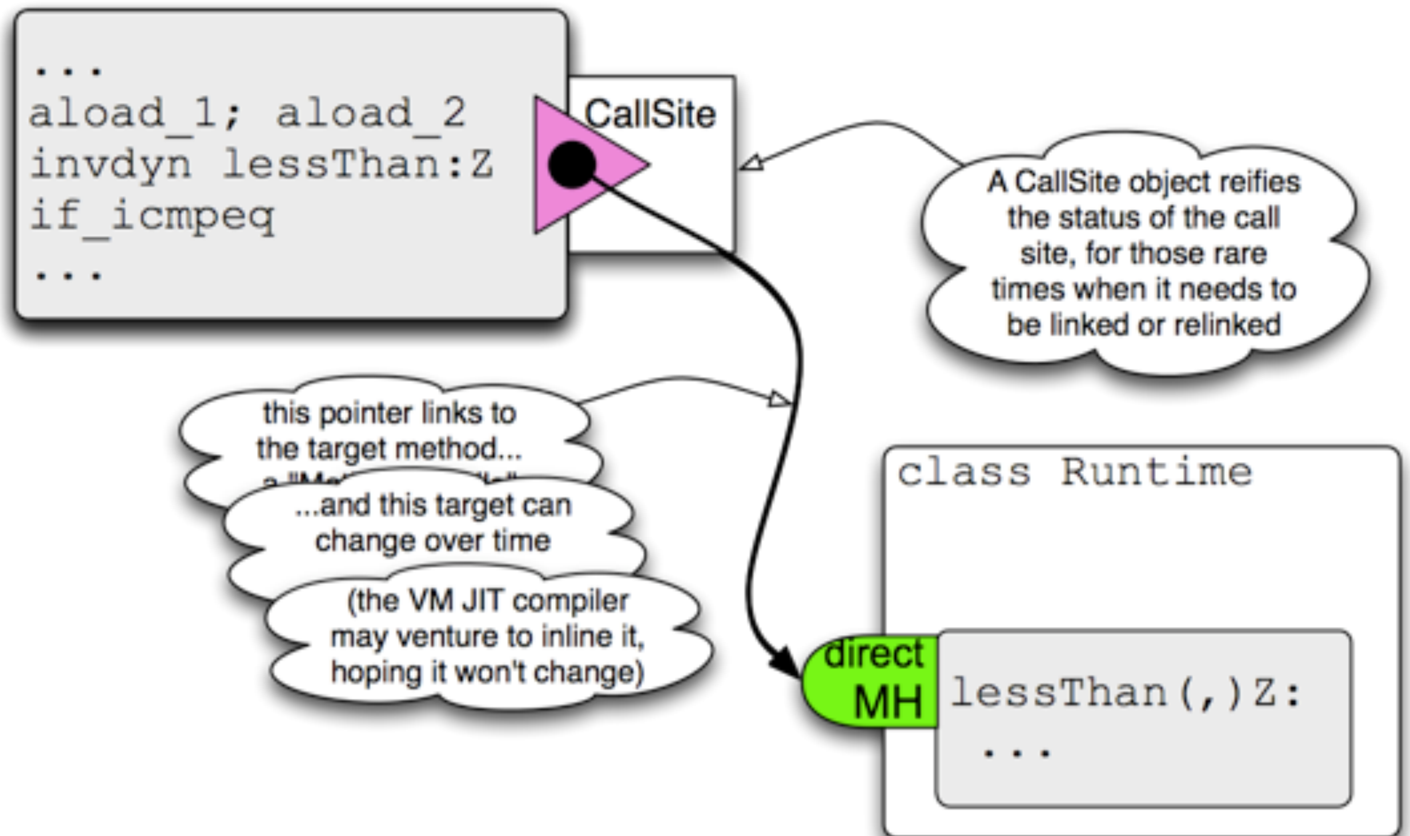
- Answer, part 1: it's all *method handles* (MH).
  - A MH can point to any accessible method
  - The target of an invokedynamic is a MH

invokedynamic, as seen by the VM:



- Fine print:
  - pink thing = CallSite object
  - green thing = MethodHandle object
  - this is the simplest case (no adaptation)

invokedynamic, as seen by the VM:



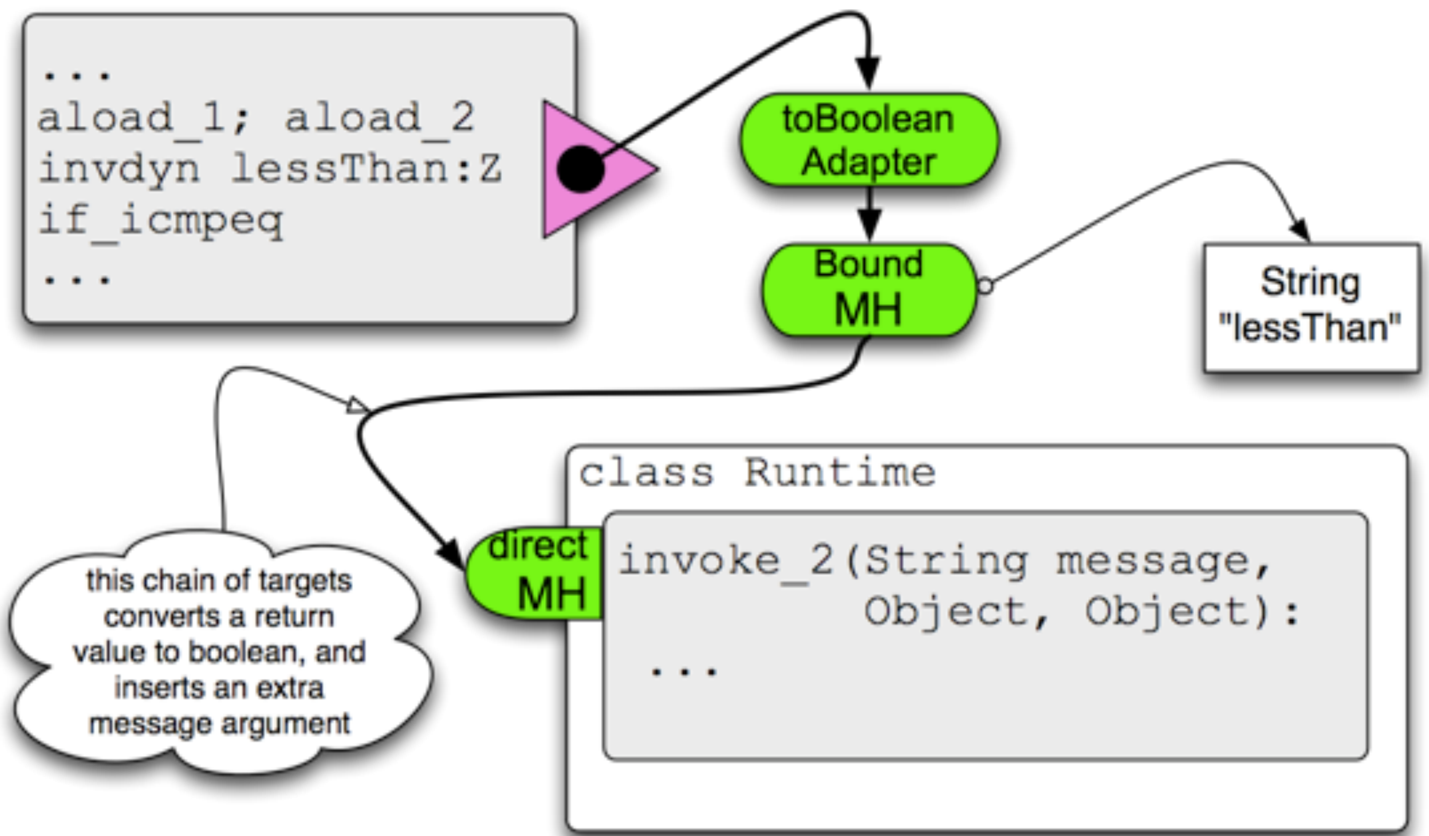
What's this method handle thing?

- Fundamental unit of "pluggable" behavior
  - Directly supports linkage to any method.
  - (Can also do normal receiver-based dispatch)
  - Works with *every* VM signature type.
    - (Not just `(Object...) => Object`)
  - Can be ***composed*** in chains (like pipelines).

Dynamic method invocation (details 2)

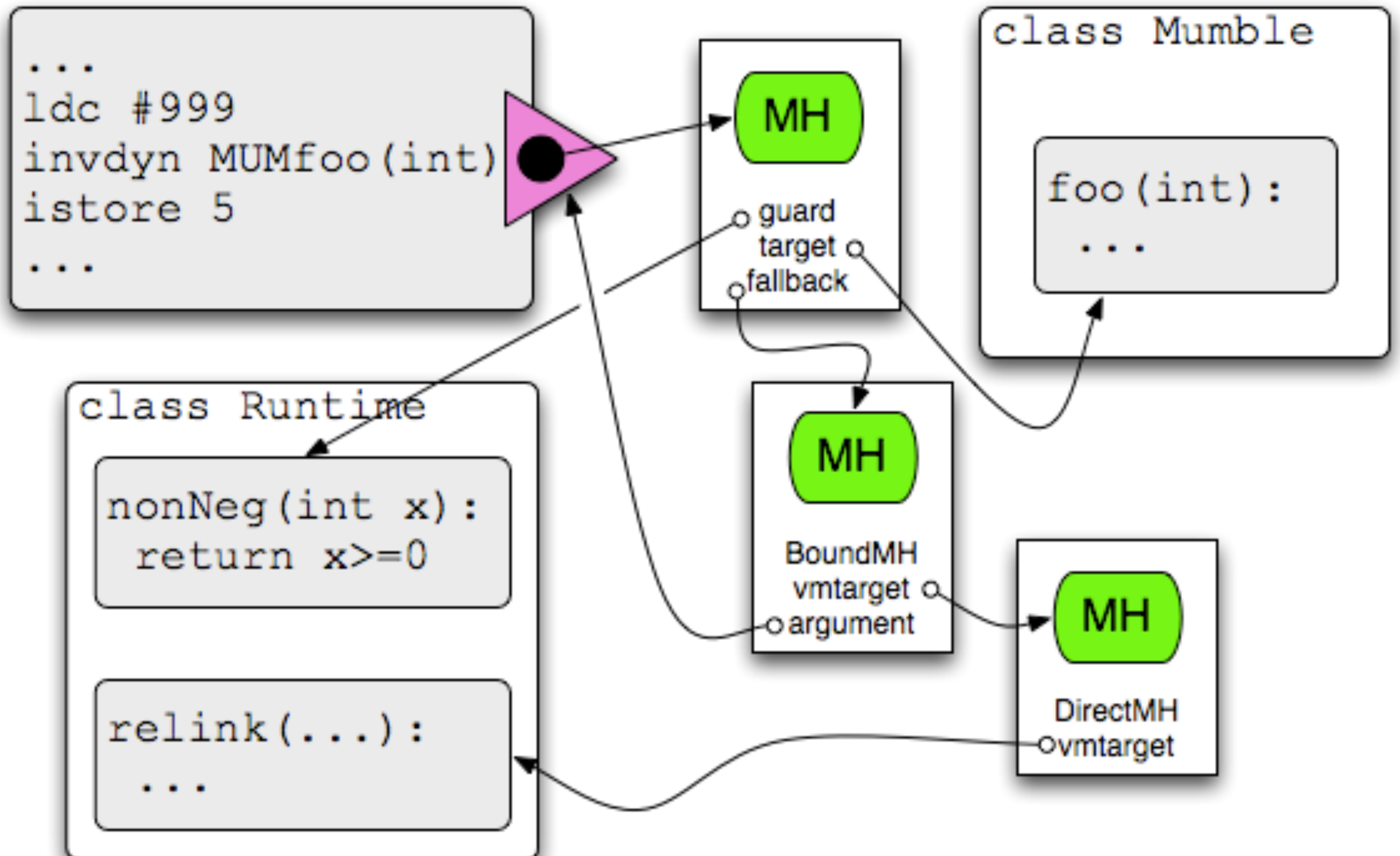
- But can I adjust names, types, arguments?
- Answer, part 2: In short, **yes**.
- Method handles can be constructed, combined, and invoked in all kinds of useful patterns.
  - Direct MH points to one method (maybe virtual)
  - Adapter MH can adjust argument types on the fly
  - Bound MH can insert extra arguments on the fly
  - “Java MH” can run arbitrary code during a call
- MHs can be chained (and the chains can be inlined)

more invokedynamic plumbing:



>

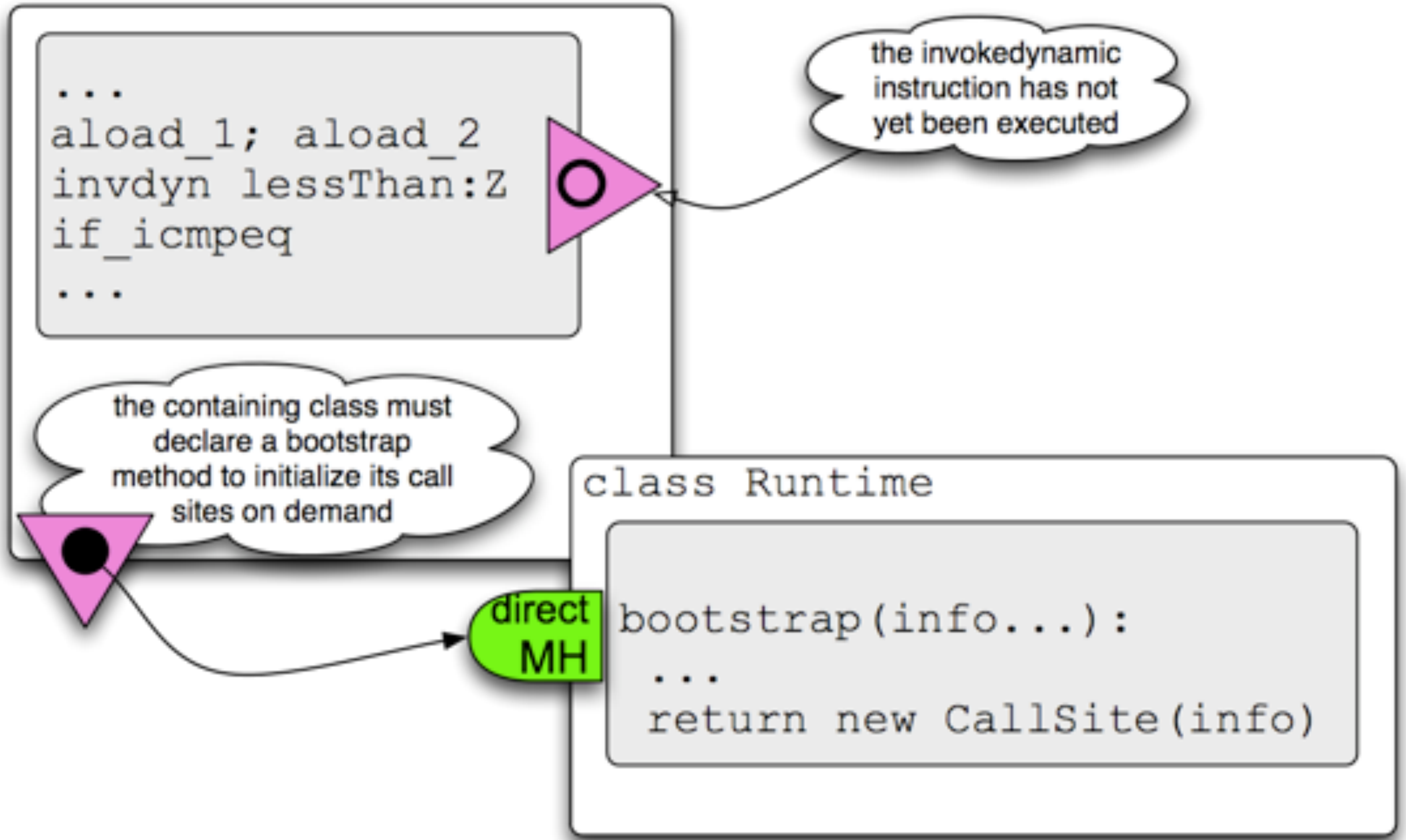
Guarded specialization (inline cache)



Dynamic method invocation (details 3)

- But who sets up these nests of method handles?
  - We need a hook for the runtime to set them up
  - And it needs to be a localizable, modular hook
- Answer, part 3: a *bootstrap method* (BSM).
  - Classes containing `invokedynamic` declare BSMs
  - BSM is called the first time a given instruction runs
  - The BSM gets to see all the details (name & type)
  - The BSM is required to construct call-site plumbing
  - (Yes, it too is a method handle.)

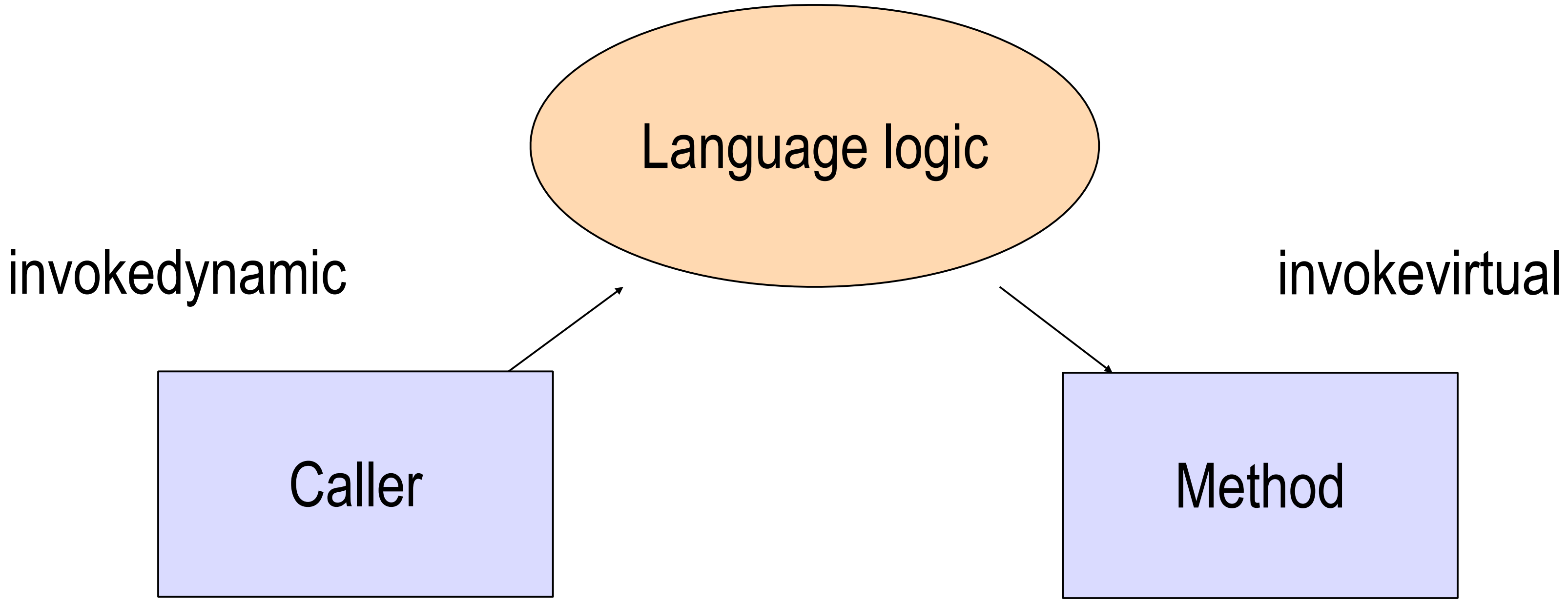
invokedynamic bootstrap logic:



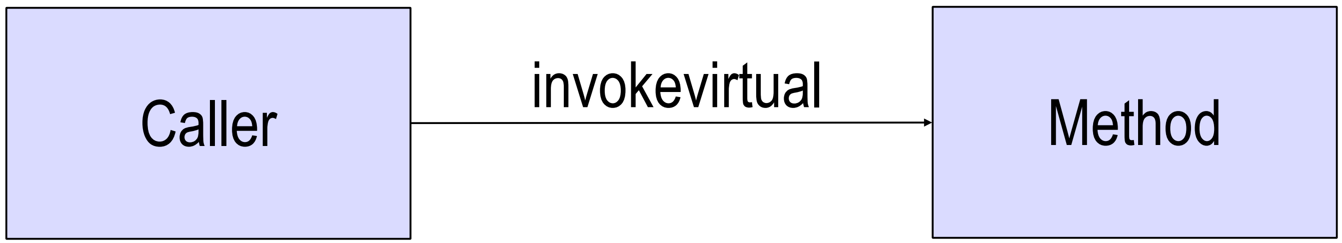
>

A budget of invokes

invokestatic	invokespecial	invokevirtual	invokeinterface	invokedynamic
no receiver	receiver class	receiver class	receiver interface	no receiver
no dispatch	no dispatch	single dispatch	single dispatch	custom dispatch
`B8 nn nn`	`B7 nn nn`	`B6 nn nn`	`B9 nn nn aa 00`	`BA nn nn 00 00`



- Check which methods are available *now* in each class [open classes]
- Check the dynamic types of arguments to the method [multimethods]
- Rearrange and inject arguments [optional and default parameters]
- Convert numbers to a different representation [fixnums]


Caller
invokevirtual
Method

Optimizing invokedynamic

- Don't want it to be slow for the first 10 years
- Most of the current optimization framework applies!
- Every invokedynamic links to *one* target method so inlining is possible
- Complex language logic can also be inlined
- A JVM implementation can even assist with *speculative* optimizations that are language-specific

Advantages of functions

- Natural unit of “live” behavior
- Simple to interpret (object method call)
- Composable (no reweaving required)
- Local expressiveness ≈ well-structured bytecode
- Simple to compile (because a simple, pure value!)

Why not always use functions?

- Not as compact as (byte-)code
- Not similar to machine code (semantic gap)
- Not clearly serializable (specific to VM instance)
- Hard to express contextual patterns (frame-relative)

A third way: What about AST?

- Serializable but not compact
- Easy to interpret, hard to compile
- Expresses *only* contextualized patterns

- Use bytecodes for macro-ops & contextuals
- Use functions for module composition

Resources

- John Rose (JSR 292 spec lead)
  - > http://blogs.sun.com/jrose
- Multi-Language Virtual Machine OpenJDK project
  - > http://openjdk.java.net/projects/mlvm
- JVM Language Summit, September 2009
  - > http://www.jvmlangsummit.com
- "JVM Languages" Google Group
  - > http://groups.google.com/group/jvm-languages

- JVM Language Summit, September 2008, 2009

http://www.jvmlangsummit.com