The Da Vinci Report

What’s happening with JVM futures?

Executive Summary

- The Da Vinci Machine Project is incubating significant changes to the JVM™ bytecode architecture including JSR 292.
- The people in this room are helping to do this.

A Bit of Back Story

From the 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.”

What’s happened in the 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

What’s happened in the last year?

- 1H2009 – JSR 292 E.G. hammers on indy spec.
- 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 (JavaOne Preview)

What’s happened recently?

- May 2009 – Java support promoted to JDK7
- Jun 2009 – Java One: http://cr.openjdk.java.net/~jrose/pres/
  - “Call for collaboration” 200906-DVMCollab.pdf
  - “Renaissance VM” 200906-RenaisVM.pdf
  - “JSR 292 Cookbook” 200906-Cookbook.pdf
- Jun/Jul 2009 – inlining of invokedynamic & MH calls
- Aug 2009 – JRuby “fib” benchmark wins w/ indy

8/23, Nutter: “This is the first time we've had JRuby performing better with indy than with our built-in logic. And even more exciting: I don't think this is actually inlining the dynamic calls, eventually still doing a slow virtual call to the target body of code.”

What’s happening now?

- Active developer community
  - mlvm-dev@openjdk.java.net
  - irc.freenode.org #mlvm
- JSR 292 RI has 2 coders (Rose, Thalinger)
- JSR 292 backport has 1 coder (Forax)
- Working patches currently exist for:
  - JSR 292 (method handles, invokedynamic, etc.)
  - JVM interface injection, continuations, tailcall, hotswap
- JSR 292 EG discussing the design
  - issues: generic vs. exact invoke, inheriting from MethodHandle, etc., etc.

Integrations to JDK 7

- 6/2009 – Java One Preview
  - runs basic (demo) codes, buggy
- 7-8/2009 – no integrations, just mlvm patch updates
  - filling out the JSR 292 APIs
  - implementing initial compiler optimizations (MH inlining!)
  - initial support for x86/64
  - fixing GC problems (managed pointers in code)
- 9/2009 – GC adjustments integrated
  - ability of compiled code to point to managed user data
- 10/2009 (M5 planned) – current mlvm patches
- before JDK7 FCS: bug fixes, more ports, performance

And for the future?

More Da Vinci Machine subprojects!

- 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?

Future fixnums

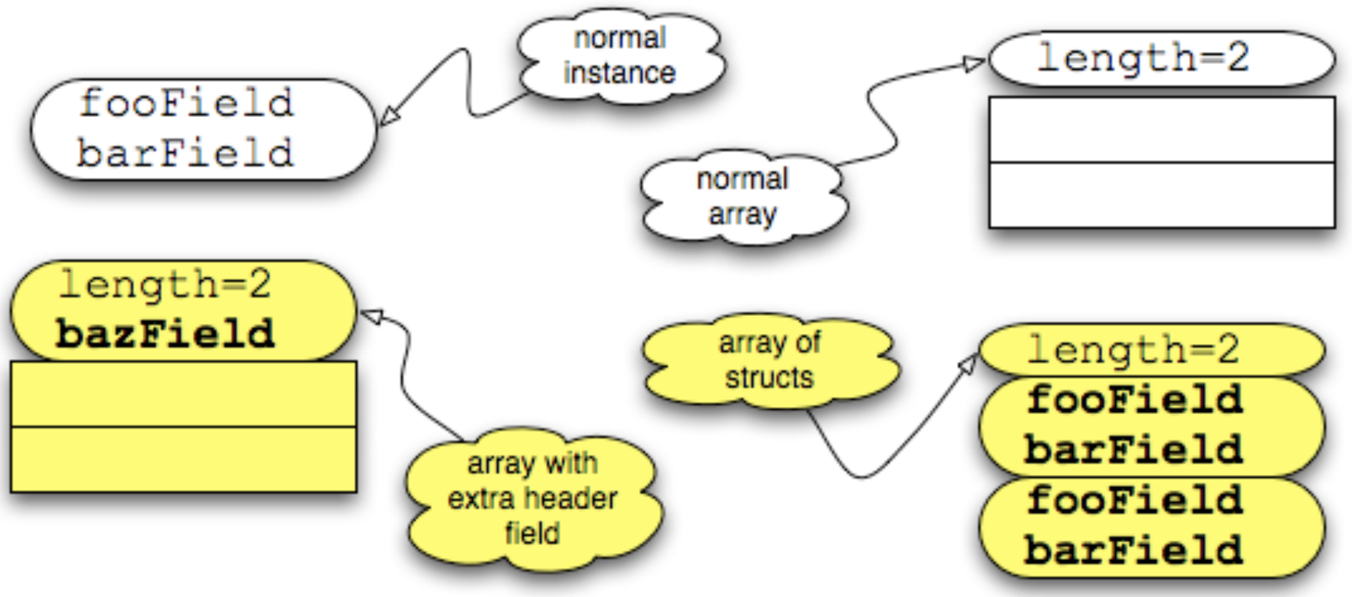
- What: Optimization of autoboxing (`Integer.valueOf`).
  - > Tagged pointer, carrying 24 to 63 bits of immediate data
  - > No indirections, no memory usage
  - > Good for all primitive wrapper types (except maybe float/double)
- Why: Dynamic languages need primitives too.
  - > But they need to interconvert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

Future tuples & 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.

Future mixed arrays (hybrids)

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



Let’s get technical about JSR 292...

Example: Class-based single dispatch

> For this source code

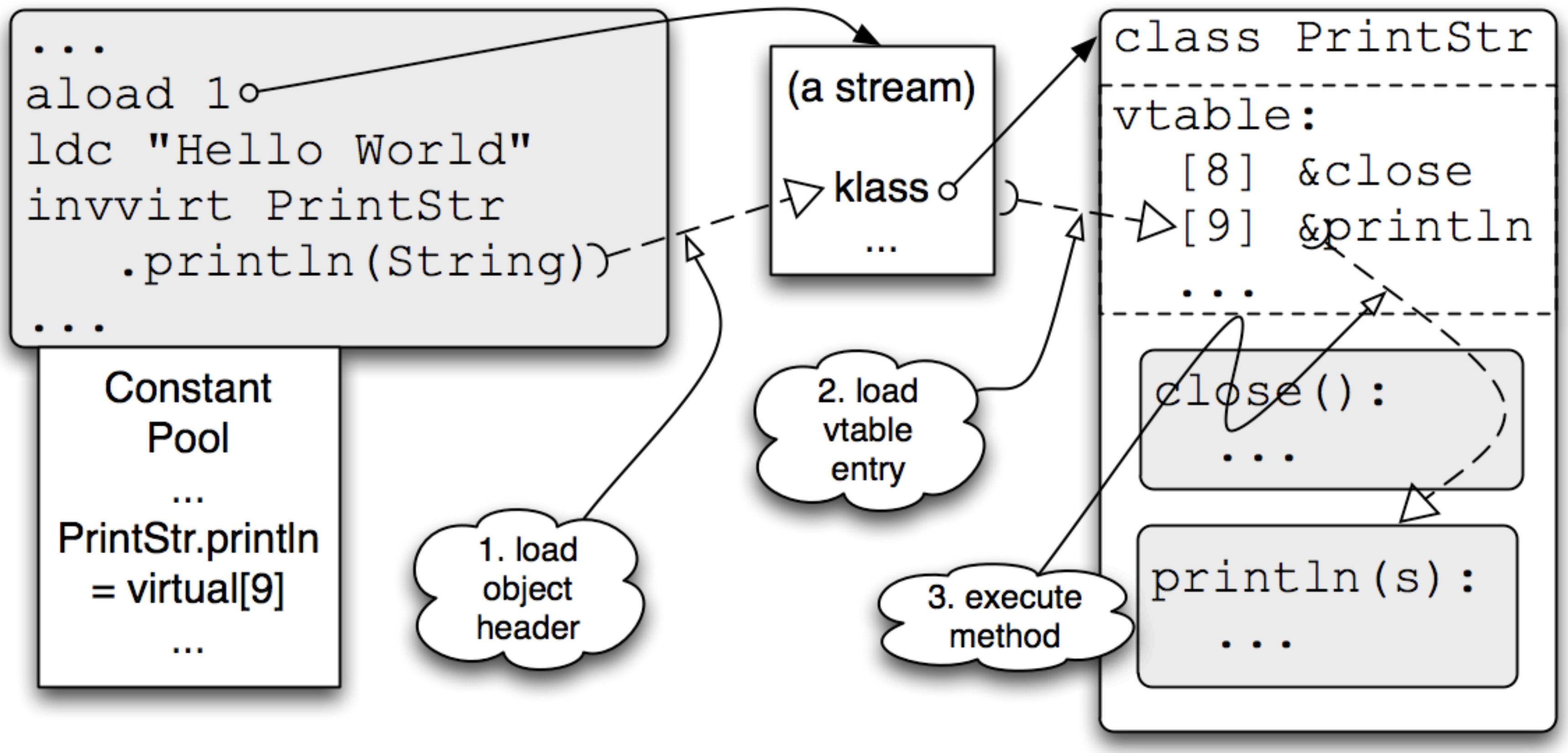
```
//PrintStream out = System.out;
out.println("Hello World");
```

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
```

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

How the VM selects the target method:



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)*

Example: Dynamic 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 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 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

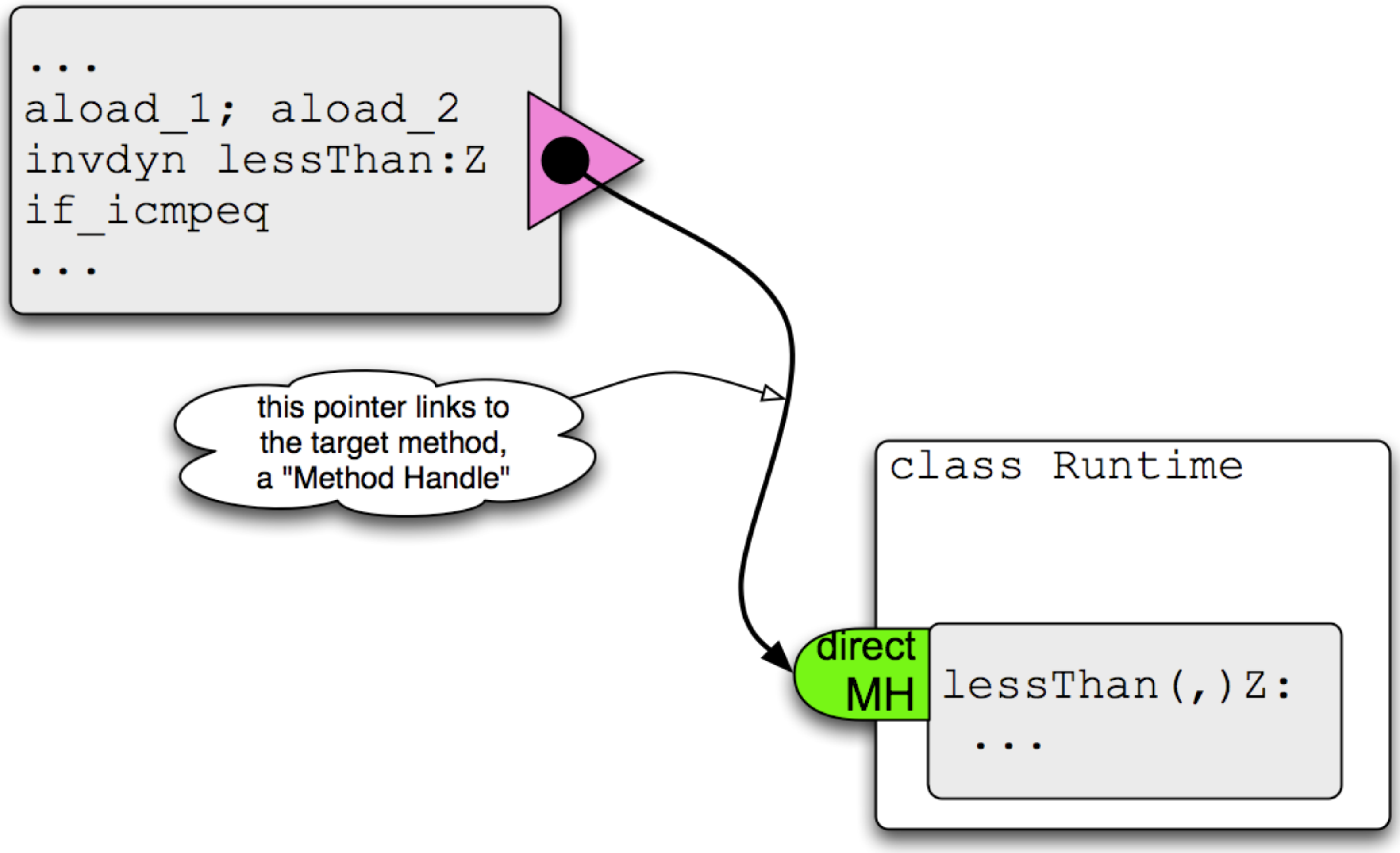
Dynamic invocation requirements

- Application-defined call site linkage state & behavior
  - > Linkage is to arbitrary behavior (code/data closures)
  - > Complete generality (polymorphism) over signatures
- Aggressive optimization
  - > Inlining of *present* linkage state
  - > Correct execution whenever linkage state changes
- Complete access to semantics of existing “invoke” ops
  - > Ability to link to any existing (accessible) method
- Reasonable ease of use for programmers

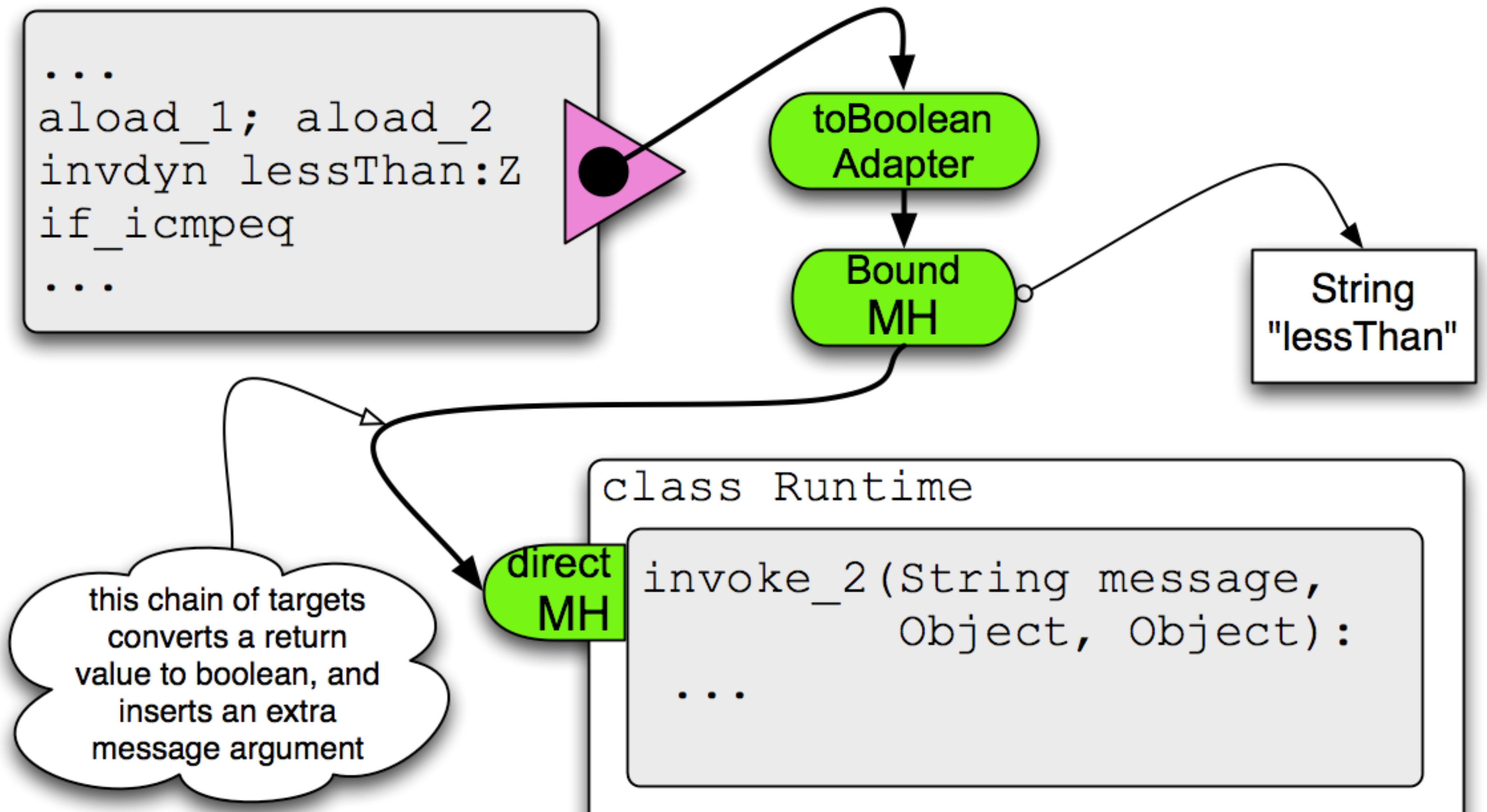
Dynamic invocation (missing details)

- 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:

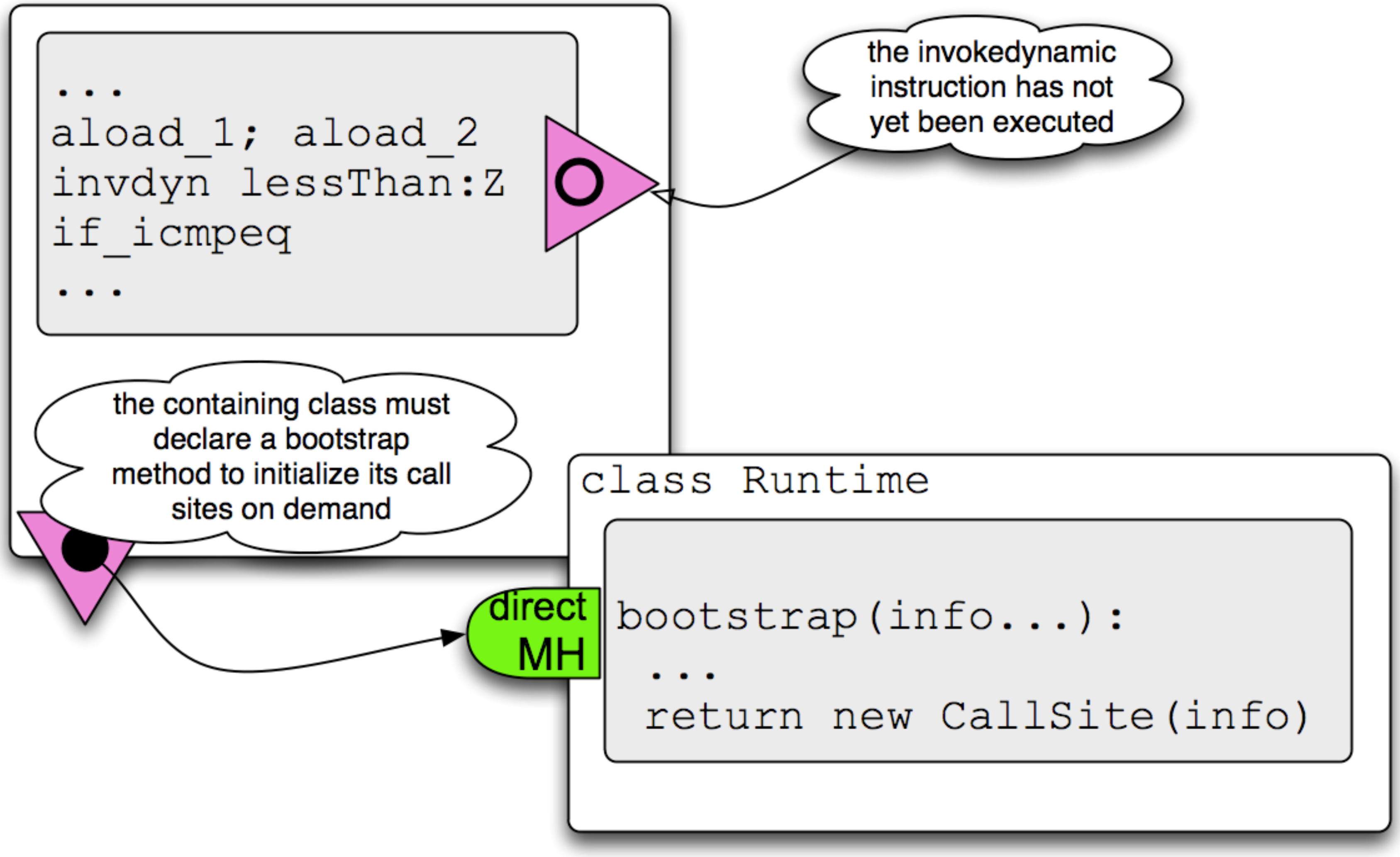


more invokedynamic plumbing: “adapters”



...
aload_1; aload_2
invdyn lessThan:Z
if_icmpeq
...
toBoolean Adapter
Bound MH
String "lessThan"
class Runtime
direct MH
invoke_2(String message, Object, Object):
...
this chain of targets converts a return value to boolean, and inserts an extra message argument

meta-plumbing: the bootstrap method



```
...
aload_1; aload_2
invdyn lessThan:Z
if_icmpeq
...
```

the invokedynamic instruction has not yet been executed
the containing class must declare a bootstrap method to initialize its call sites on demand
direct MH
class Runtime

```
bootstrap(info...):
 ...
 return new CallSite(info)
```

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

So we’re done?

- Not there yet.
- More engineering to do (watch for JDK7 M5!)
  - ports: x86/64, SPARC, compressed oops, C++ interpreter
  - bugs & performance
- One more pass of specification work
  - ...now that we have an RI to play with (and not before)
  - JSR 292 EG is discussing certain known issues
  - we need more smart people trying to use the RI

How to mature a specification

- Make early drafts and prototypes public
- Get a community of experts, including a few key users
- Find out what really works and doesn't, *in practice*
  - Make sure you can implement it in at least one VM
  - Make sure at least a few users can actually benefit from it
  - Avoid excessive esthetics & philosophy (& bike sheds)
  - But expect experience to birth new insights & refactorings
- Iterate on the specification in light of all of the above
  - Involve VM vendors and key users, all the way to the end
- And, expect a little "you Fool, you've destroyed Java!"

Burning issue: varieties of invoke

- There are three “natural” forms of JVM invocation:
  - `exactInvoke` (identical signatures, existing linkage match)
  - `genericInvoke` (value-preserving, box/unbox/cast)
  - `varargsInvoke` (the most general; uses argument array)
- These are in order of increasing complexity & cost
- Any of them can simulate any of the others.
  - Can any be dropped? Experience shows use cases for all.
- Which should be favored as `MethodHandle.invoke`?
  - IBM/Oracle: generic invoke is useful; let’s default to it
  - Sun RI: possible before useful; start w/ exact version
  - users: (hello out there?)

Burning issue: subclassing JMH

- RI includes “JavaMethodHandle”, a subclassable MH
  - > used in RI for factoring method handle state/behavior
  - > may be used with inner classes, for closure-like expressions
- Can be used to make supertypes of MH
  - > for example, settable method pointers:

```
abstract class SettableMH
  extends JavaMethodHandle { ...
    abstract MethodHandle setter();  }
```

  - > (or self-identifying method pointers, etc.)
- Implementation is simple: A self-bound object.
- Alternative: Non-MH objects which wrap MHs.

Burning issue: MH constants

- RI includes reflective-style factories for MH’s & types
  - findVirtual(class, name, type), findStatic, findSpecial
  - methodType(rtype, ptype...)
  - All existing Java example codes use these
- For bytecode compilers, support these too?
  - CONSTANT_VirtualMethodHandle (class, name&type)
  - CONSTANT_{Static,Special}MethodHandle
  - CONSTANT_MethodType (method signature)
- Advantages: Static analysis, caching, prebinding
- Disadvantages: Two ways to do one thing

Burning issue: Thrown exceptions

- The JVM has no exception checking rules
  - > ...but Java does
  - > so what exceptions should a dynamic invoke throw?
- The painful truth: “throws Throwable”
  - > Avoids putting another hole in Java’s checked exceptions
- MH-using and invokedynamic-using code looks strange
  - > must have “throws Throwable” on every subroutine
  - > when returning to regular Java code, must catch & dispose
- Therefore, we need a code pattern for safe disposal

```
try { dynamic stuff... }
catch (Throwable t) { throw
    checkException(t, IOException.class); }
```

Burning issue: Call site invalidation

- Call sites are linked (and reified) via up-calls
  - ...to the app. supplied “bootstrap method”
  - this happens lazily, and once only
- Sometimes apps need to do mass invalidation
  - does this mean call sites are reified again?
  - or does it mean they get reset to some neutral value?
- This needs to be a privileged instruction
  - so that cannot be in a race with call site execution
  - must be done at a “safepoint”
- Also, what is the right API for batching the victims?

Burning issue: Call site splitting

- Oddly, invokedynamic has a data structure per BCI
  - this gives the crucial “hook” for building inline caches
  - otherwise, per-BCI state is minimal (linkage status)
- Question: May a call site ever be split in two?
  - perhaps a JVM will want to clone (inline) some code
  - or maybe we’ll invent a “method customization” mechanism
- This issue interacts with the previous:
  - invalidation can be viewed as splitting and discarding

Non-Conclusion

- Let’s talk more... JSR 292 needs wise users!
- There’s a workshop at 4:00 to talk more about this.