Invoke Dynamic

By Mohit Kumar

JRuby Implementation

- Highly Dynamic
  - Dynamic Typing and dispatch
    - Lookup by name only (Single)
  - Mutable Classes
    - Blank States filled at loadtime
  - Closure and cross frame data($~,$2,etc)
  - eval

Doesnt the JVM do this internally?

- Internally yes.
  - Majority of calls are dynamically bound.
  - Hotspot makes them static fast
- Externally, Not before invokedynamic.

Caching Callsite

- Monomorphic inline cache
  - Cache method +class serial number
  - Compare serial number to guard
- Eliminates costly hash lookups
  - No JVM can inline through it.

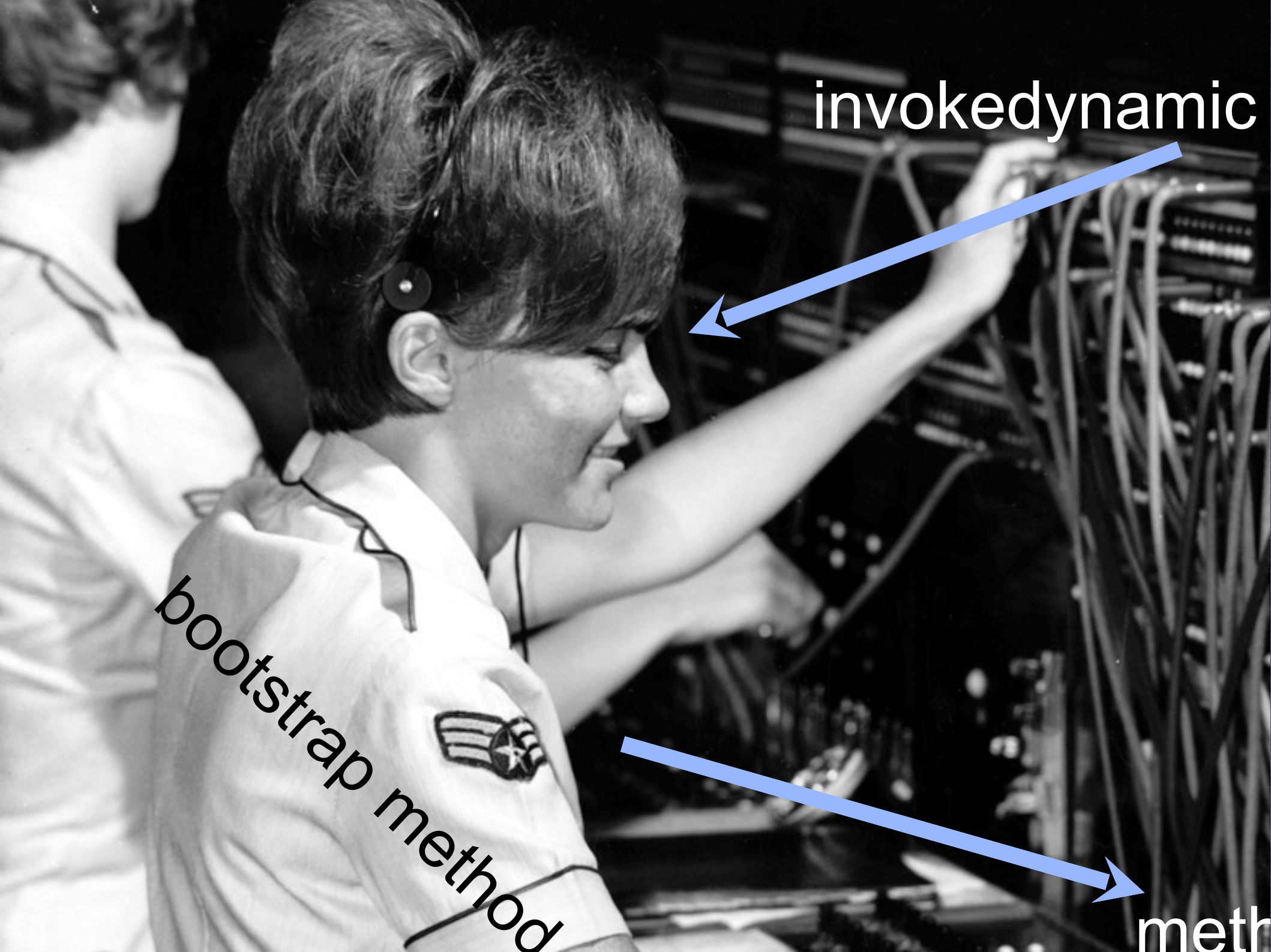
What JVM sees

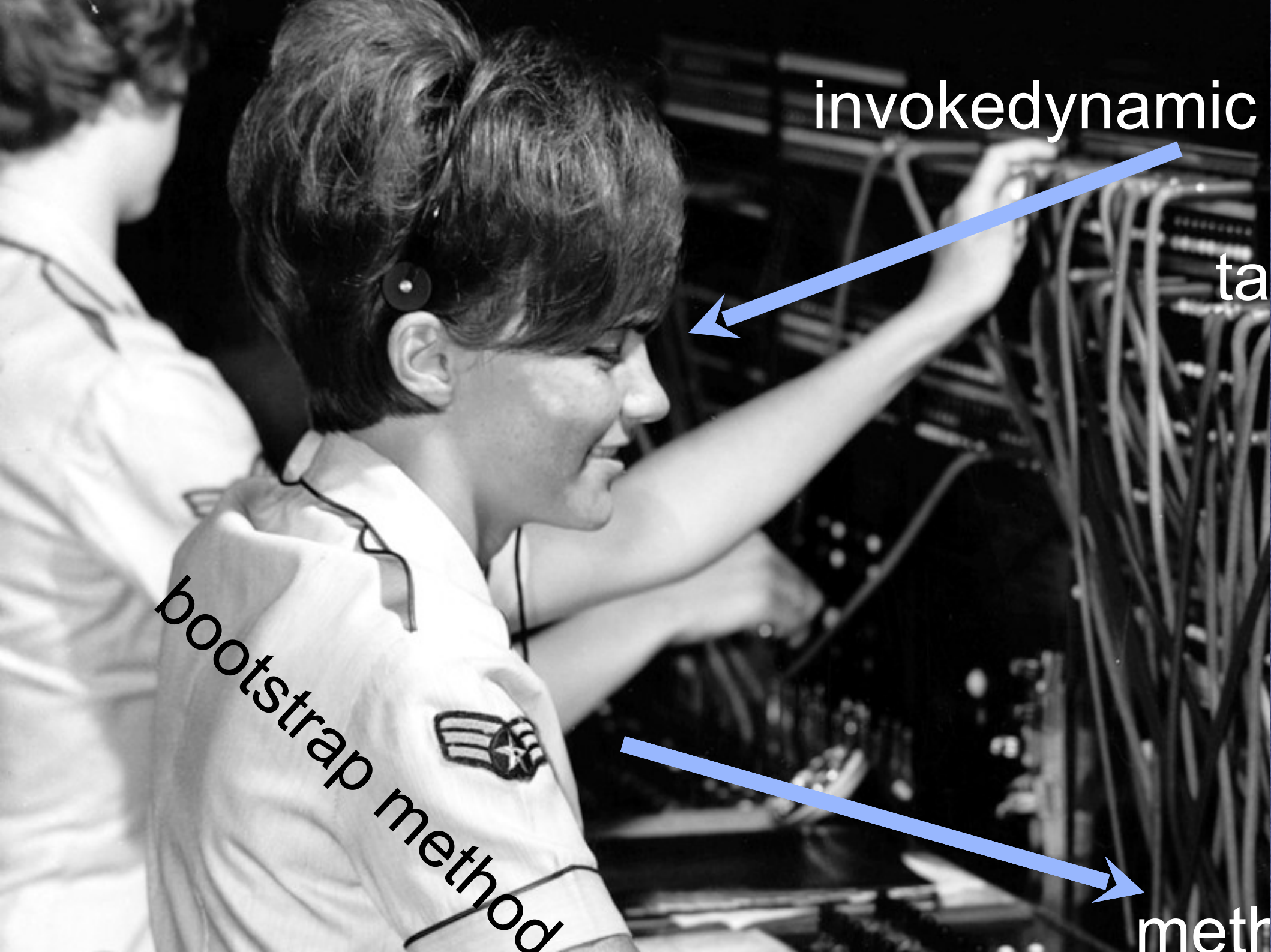
- Ruby Code calls Caching callsite
  - Monomorphic
- Caching Callsite looks up calls method
  - Megamorphic
- Lamda will have the same problem
  - And maybe the same solution

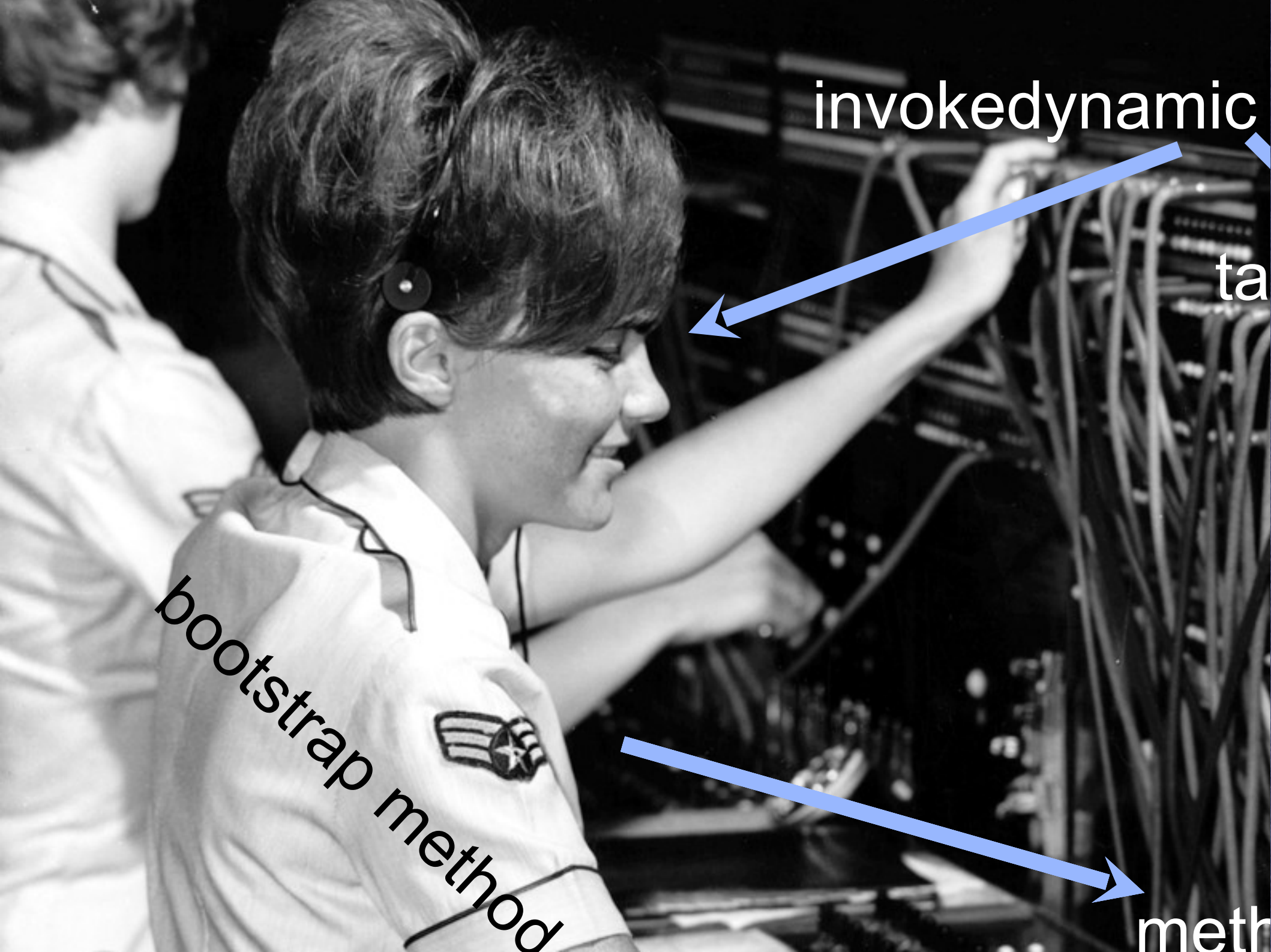
invokedynamic

- Allows custome code for call protocol
- JVM inlines through it
- The JVMs new “Golden Hammer”
- Jruby Masters Today, 1.7ish soon
  - 2-10x Performance Improvement
  - It really does works


invokedynamic
bootstrap method


invokedynamic
bootstrap method
meth


invokedynamic
ta
bootstrap method
meth


invokedynamic
ta
bootstrap method
meth


invokedynamic
ta

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

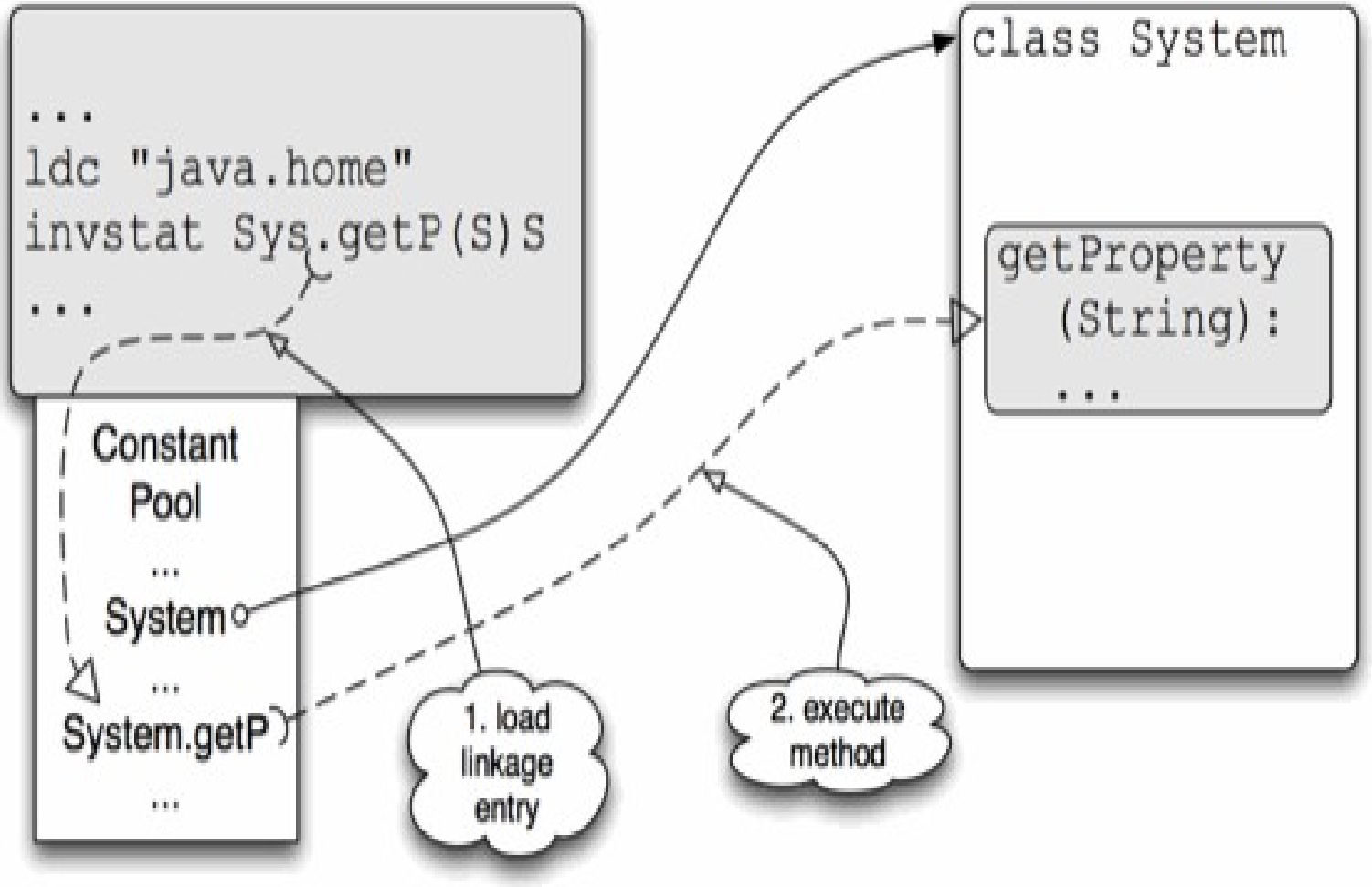
```
String s = System.getProperty("java.home");
```

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

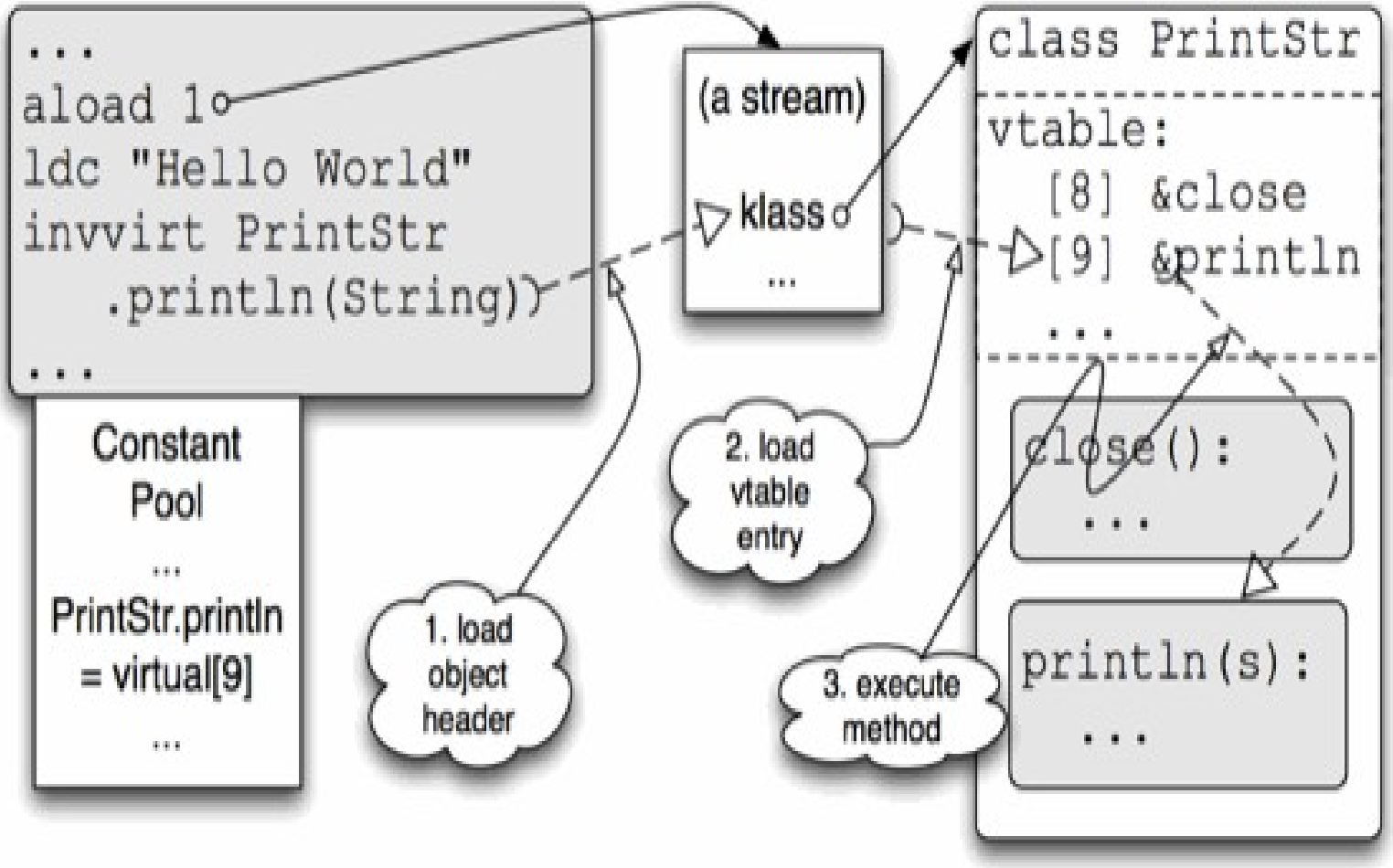
```
//PrintStream out = System.out;
out.println("Hello World");
```

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:

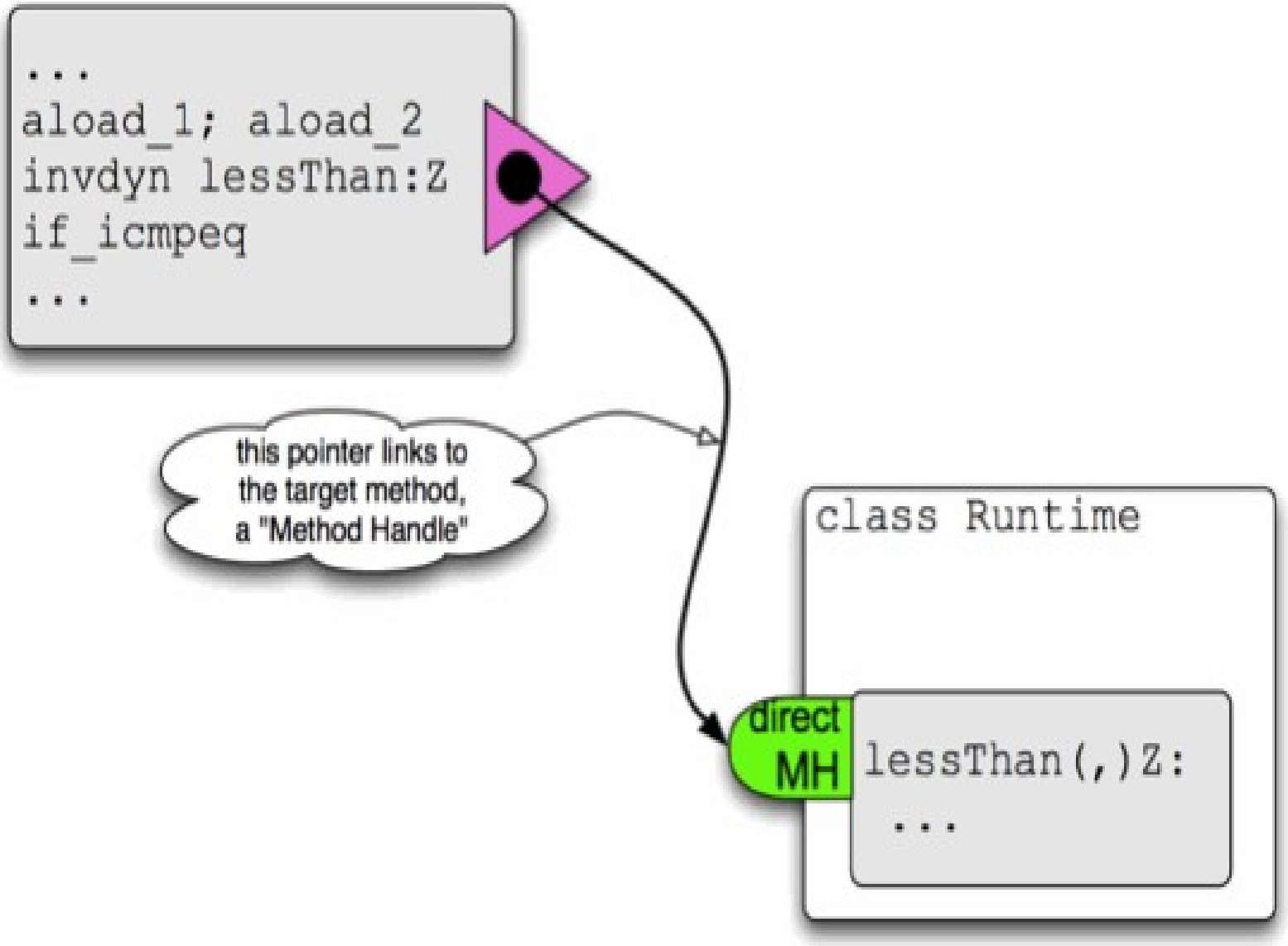
```
0:   aload_1; aload_2
2:   invokedynamic #3  //NameAndType lessThan:
                    (Ljava/lang/Object;Ljava/lang/Object;)Z
5:   if_icmpeq
```

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

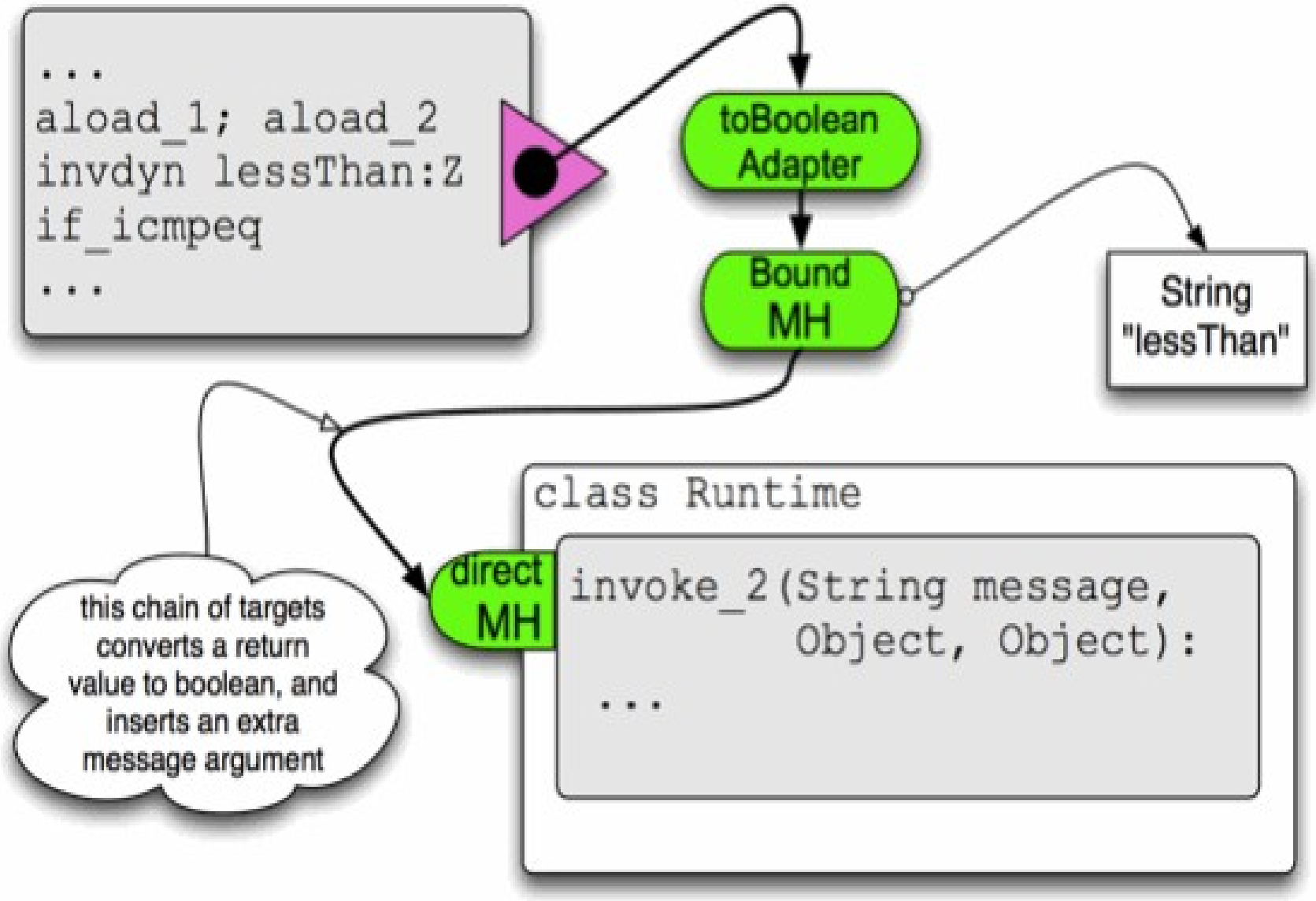
Dynamic method invocation (details)

- 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

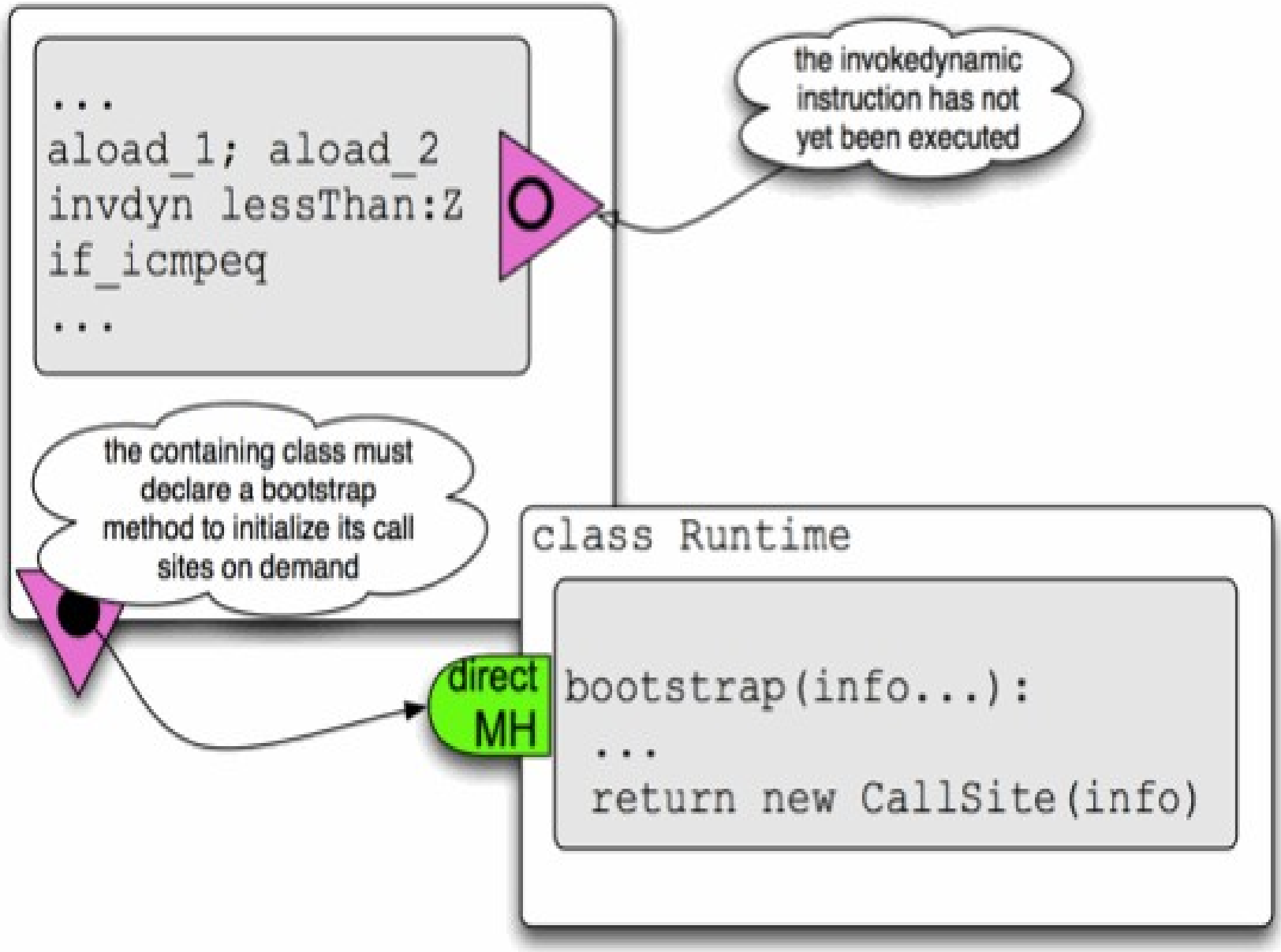
invokedynamic, as seen by the VM:



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**