invokedynamic

IN 45 MINUTES!!!

Me

- Charles Oliver Nutter
  - headius@headius.com, @headius
  - blog.headius.com
- JRuby Guy at Sun, Engine Yard, Red Hat
- JVM enthusiast, educator, contributor
- Earliest adopter of invokedynamic

27 April, 2011

Charles Nutter
@headius

Invokedynamic is the most important addition to Java in years. It will change the face of the platform.

History

- JVM authors mentioned non-Java languages
- Language authors have targeted JVM
  - Hundreds of JVM languages now
- But JVM was a mismatch for many of them
  - Usually required tricks that defeated JVM optimizations
  - Or required features JDK could not provide

JVM Languages Through the Years

- Early impls of Python (JPython), JS (Rhino) and many others
- No official backing by Sun until 2006
- JRuby team hired
- JSR-292 “invokedynamic” rebooted in 2007
- Java 7 shipped invokedynamic in 2011

What is

`invokedynamic`

New Bytecode?

Well, yes...but what does that mean?
And is that all?

New form of invocation?

That's one use, but there are many others

Only for Dynamic Languages?

Dynamic dispatch is a common use,
but there are many others

A User-definable Bytecode

You decide how the JVM implements it

A User-definable Bytecode

You decide how the JVM implements it

+

Method Pointers and Adapters

Faster than reflection, with user-defined argument, flow, and exception handling

invokedynamic

user-def'd bytecode

invokedynamic

user-def'd bytecode	method pointers
`invoke`	`dynamic`

bytecode + bootstrap	MethodHandles
`invoke`	`dynamic`

https://github.com/headius/indy_deep_dive

MethodHandles

Method Handles

- Function/field/array pointers
- Argument manipulation
- Flow control
- Optimizable by the JVM
  - This is very important

java.lang.invoke

- MethodHandle
  - An invokable target + adaptations
- MethodType
  - Representation of args + return type
- MethodHandles
  - Utilities for acquiring, adapting handles

MethodHandles.Lookup

- Method pointers
  - findStatic, findVirtual, findSpecial, findConstructor
- Field pointers
  - findGetter, findSetter, findStaticGetter, findStaticSetter

Why Casting?

```
value1 = (String)mh1.invoke("java.home");
mh2.invoke((Object)"Hello, world");
```

- invoke() is “signature polymorphic”
  - Call-side types define signature
  - At compile time
- Signature must match MethodHandle type
  - Or use invokeWithArguments

Adapters

- Methods on j.l.i.MethodHandles
- Argument manipulation, modification
- Flow control and exception handling
- Similar to writing your own command-pattern utility objects

Argument Juggling

- insert, drop, permute
- filter, fold, cast
- splat (varargs), spread (unbox varargs)

Flow Control

- guardWithTest: boolean branch
  - condition, true path, false path
  - Combination of three handles
- SwitchPoint: on/off branch
  - true and false paths
  - Once off, always off

Exception Handling

- catchException
  - body, exception type, handler
- throwException
  - Throws Throwable in argument 0

bytecode

Goals of JSR 292

- A user-definable bytecode
  - Full freedom to define VM behavior
- Fast method pointers + adapters
- Optimizable like normal Java code
- Avoid future modifications

JVM 101

JVM 101

200 opcodes

JVM 101

200 opcodes

Ten (or 16) “endpoints”

JVM 101

200 opcodes

Ten (or 16) “endpoints”

Invocation
`invokevirtual`
`invokeinterface`
`invokestatic`
`invokespecial`

JVM 101

200 opcodes

Ten (or 16) “endpoints”

Invocation
`invokevirtual`
`invokeinterface`
`invokestatic`
`invokespecial`

Field Access
`getfield`
`setfield`
`getstatic`
`setstatic`

JVM 101

200 opcodes

Ten (or 16) “endpoints”

Invocation	Field Access	Array Access
`invokevirtual`	`getfield`	`*aload`
`invokeinterface`	`setfield`	`*astore`
`invokestatic`	`getstatic`	`b,s,c,i,l,d,f,a`
`invokespecial`	`setstatic`	

JVM 101

200 opcodes

Ten (or 16) "endpoints"

Invocation	Field Access	Array Access
`invokevirtual`	`getfield`	`*aload`
`invokeinterface`	`setfield`	`*astore`
`invokestatic`	`getstatic`	`b,s,c,i,l,d,f,a`
`invokespecial`	`setstatic`	

All Java code revolves around these endpoints

Remaining ops are stack, local vars, flow control, allocation, and math/boolean/bit operations

The Problem

- JVM spec (pre-7) defined 200 opcodes
- All bytecode lives within these 200
- What if your use case does not fit?
  - Dynamic language/dispatch
  - Lazy initialization
  - Non-Java features

```
// Static
System.currentTimeMillis()
Math.log(1.0)

// Virtual
"hello".toUpperCase()
System.out.println()

// Interface
myList.add("happy happy")
myRunnable.run()

// Special
new ArrayList()
super.equals(other)
```

```
// Static
invokestatic java/lang/System.currentTimeMillis:()J
invokestatic java/lang/Math.log:(D)D

// Virtual
invokevirtual java/lang/String.toUpperCase:()Ljava/lang/String;
invokevirtual java/io/PrintStream.println:()V

// Interface
invokeinterface java/util/List.add:(Ljava/lang/Object;)Z
invokeinterface java/lang/Runnable.add:()V

// Special
invokespecial java/util/ArrayList.<init>:()V
invokespecial java/lang/Object.equals:(java/lang/Object)Z
```

`invokestatic`

`invokevirtual`

`invokeinterface`

`invokespecial`

`invokevirtual`

1. Confirm object is of correct type
2. Confirm arguments are of correct type
3. Look up method on Java class
4. Cache method
5. Invoke method

`invokestatic`

1. Confirm arguments are of correct type
2. Look up method on Java class
3. Cache method
4. Invoke method

`invokeinterface`

1. Confirm object's type implements interface
2. Confirm arguments are of correct type
3. Look up method on Java class
4. Cache method
5. Invoke method

`invokespecial`

1. Confirm object is of correct type
2. Confirm arguments are of correct type
3. Confirm target method is visible
4. Look up method on Java class
5. Cache method
6. Invoke method

`invokevirtual`

1. Confirm object is of correct type
2. Confirm arguments are of correct type
3. Look up method on Java class
4. Cache method
5. Invoke method

`invokestatic`

1. Confirm arguments are of correct type
2. Look up method on Java class
3. Cache method
4. Invoke method

`invokeinterface`

1. Confirm object's type implements interface
2. Confirm arguments are of correct type
3. Look up method on Java class
4. Cache method
5. Invoke method

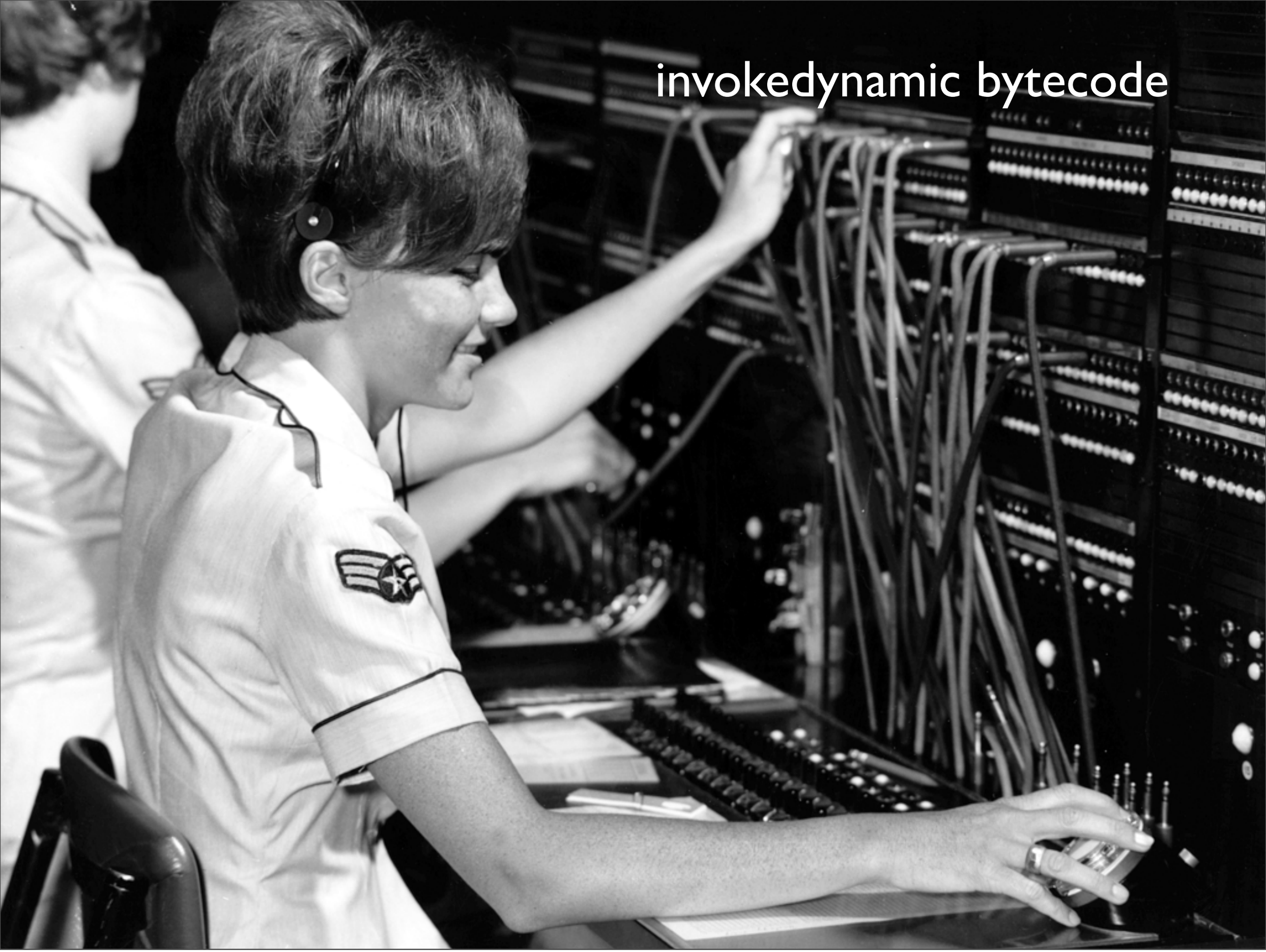
`invokespecial`

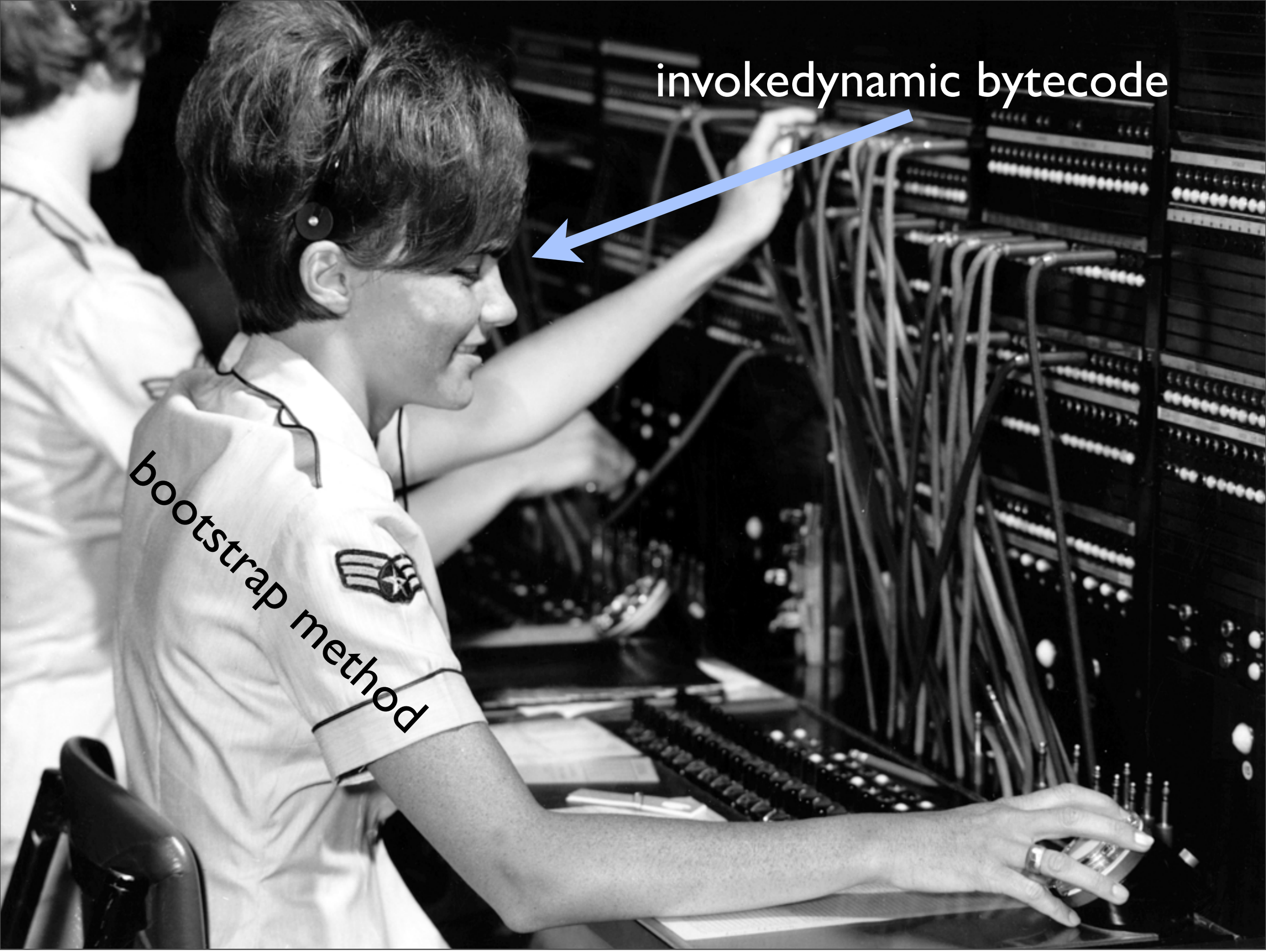
1. Confirm object is of correct type
2. Confirm arguments are of correct type
3. Confirm target method is visible
4. Look up method on Java class
5. Cache method
6. Invoke method

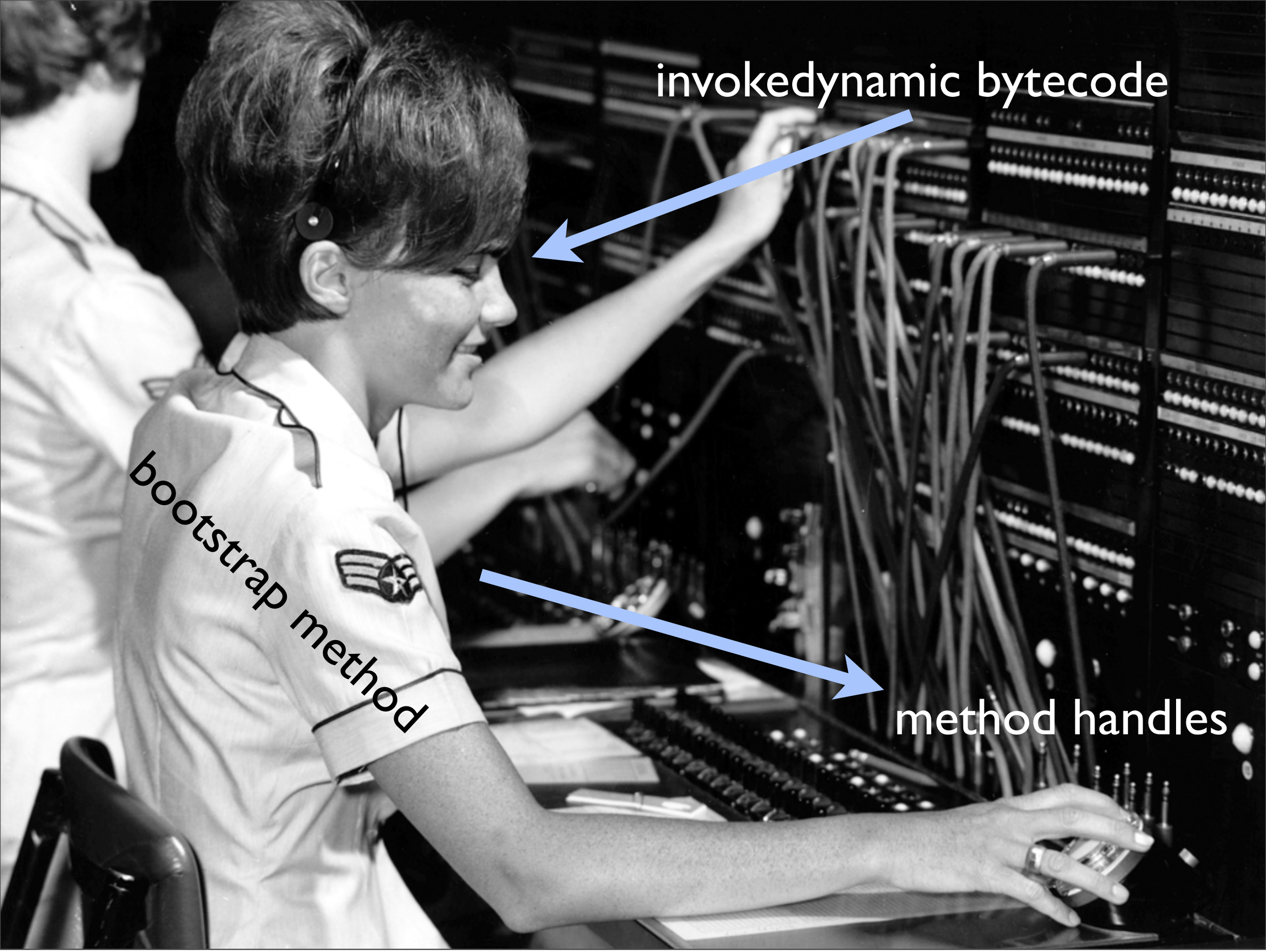
`invokedynamic`

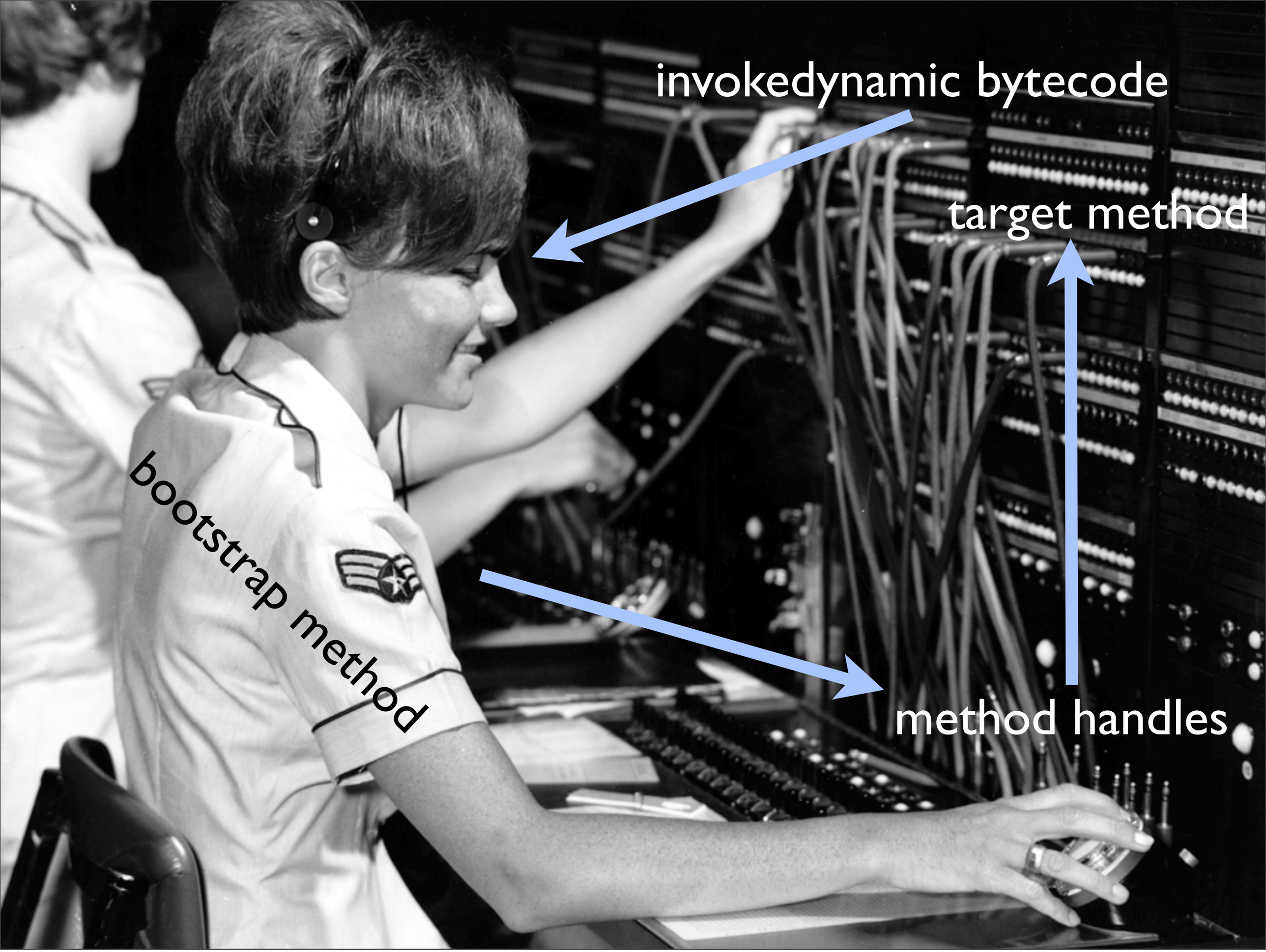
1. Call bootstrap handle (your code)
2. Bootstrap prepares CallSite + MethodHandle
3. MethodHandle invoked now and future (until CallSite changes)

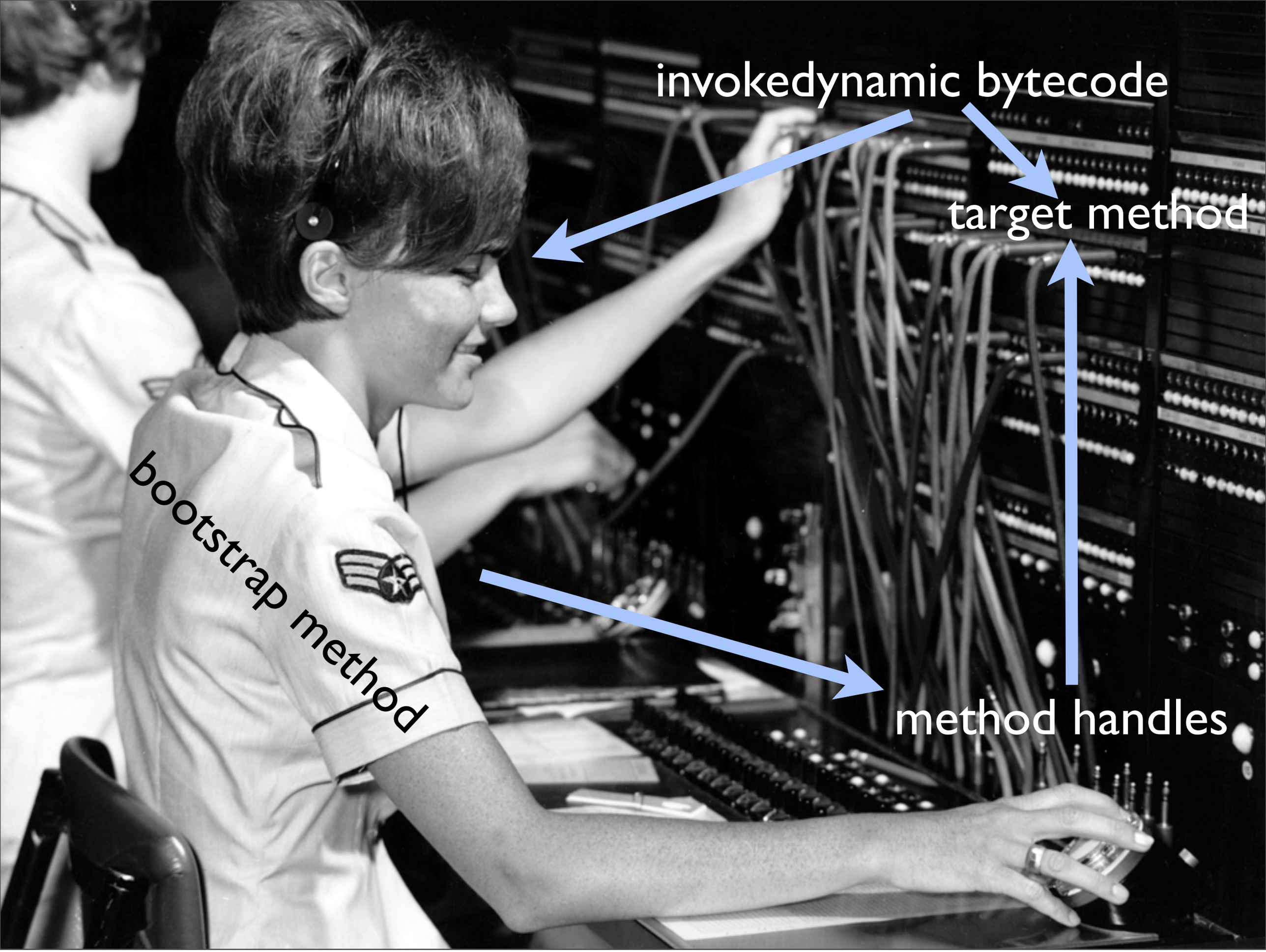
invokedynamic bytecode

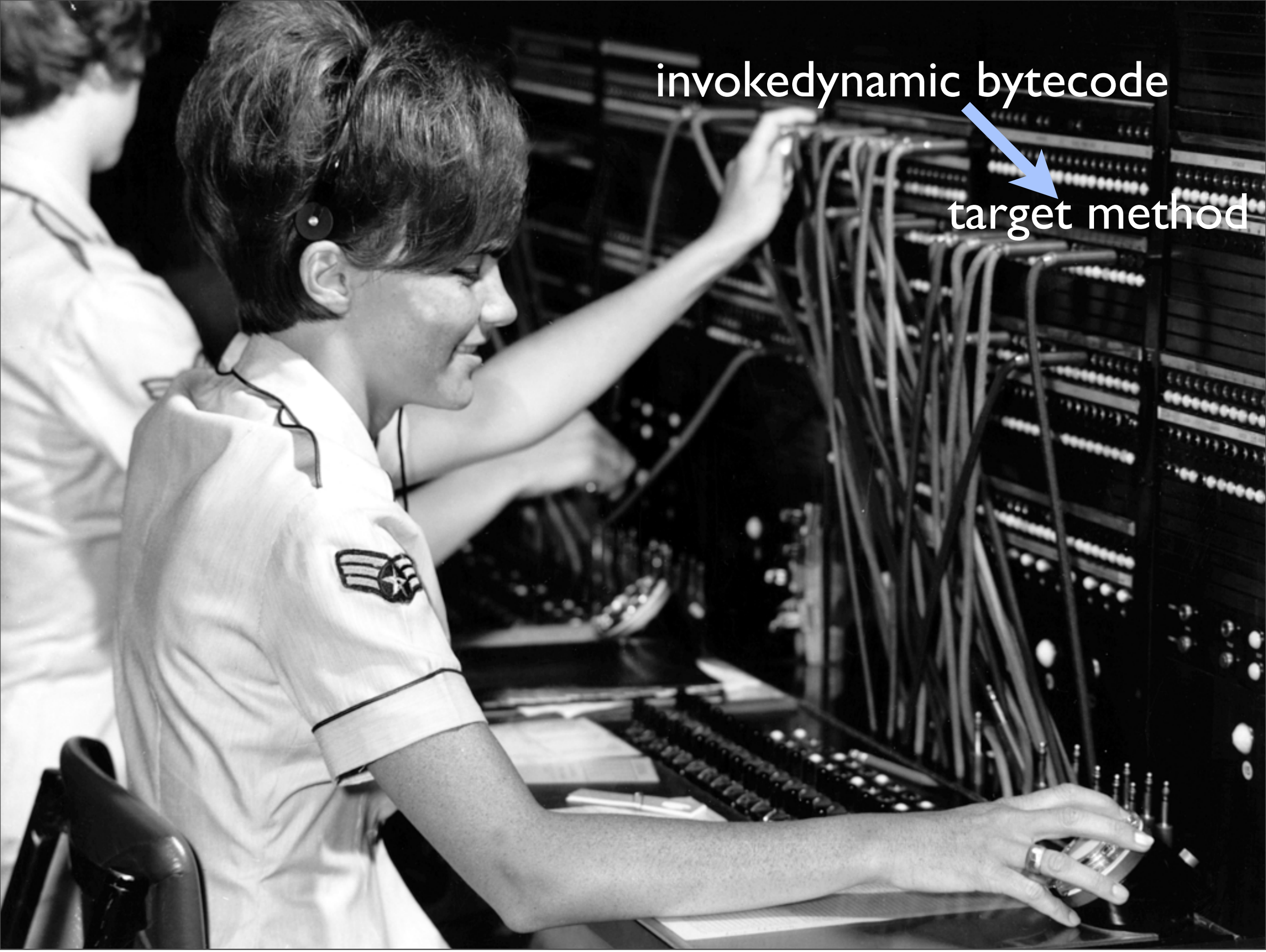



invokedynamic bytecode
bootstrap method


invokedynamic bytecode
bootstrap method
method handles


invokedynamic bytecode
target method
bootstrap method
method handles


invokedynamic bytecode
target method
bootstrap method
method handles


invokedynamic bytecode
target method

All Together Now...

Tools

- ObjectWeb's ASM library
  - De facto standard bytecode library
- Jitescript
  - DSL/fluent wrapper around ASM
- InvokeBinder
  - DSL/fluent API for building MH chains

#1: Trivial Binding

- Simple binding of a method
  - j.l.i.ConstantCallSite
- Method returns String
- Print out String

#2: Modifying the Site

- Toggle between two targets each call
  - j.l.i.MutableCallSite
- Call site sent into methods
  - ...so they can modify it
- Trivial example of late binding
- InvokeBinder usage

#3: Dynamic Dispatch

- Target method not known at compile time
- Dynamically look up after bootstrap
- Rebind call site to proper method

StupidScript

- push <string>: push string on stack
- send <number>: call function with n args
  - One arg call: ["print", "Hello, world"]
- def <name> '\n' <op1> ['\n' <op2> ...] '\n' end
  - define function with given ops
- One builtin: print (System.out.println)

StupidScript

- push <string>: push string on stack
- send <number>: call function with n args
- def <name> '\n' <op1> ['\n' <op2> ...] '\n' end
- One builtin: print (System.out.println)

Implementation

- Simple parser generates AST
- Compiler walks AST, emits .class
- Top level code goes into run() method
- defs create additional methods

Hello, world!

```
push Hello, world!
push print
send 1
```

Define Function

```
def hello
  push print
  push Hello, world!
  send 1
end
```

Call Function

```
push hello
send 0
```

More Information

- My blog: http://blog.headius.com
- Follow me: @headius
- Code on Github
  - https://github.com/headius/indy_deep_dive
- Slides posted online