Let's talk about

`invokedynamic`

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, JS, many others
- No official backing by Sun until 2006
- JRuby team hired
- JSR-292 “invokedynamic” rebooted 2007
- Java 7 shipped invokedynamic in 2011

What is

`invokedynamic`

Only for Dynamic Languages?

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

New form of invocation?

That's one use, but there are many others

New Bytecode?

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

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

user-def'd bytecode	Method handles
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

```
// has access to everything visible to this code
MethodHandles.Lookup LOOKUP =
        MethodHandles.lookup();

// has access to only public fields and methods
MethodHandles.Lookup PUBLOOKUP =
        MethodHandles.publicLookup();
```

MethodHandles.Lookup

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

```
// example Java
String value1 = System.getProperty("java.home");
System.out.println("Hello, world");

// getProperty signature
MethodType type1 =
        MethodType.methodType(String.class, String.class);
// println signature
MethodType type2 =
        MethodType.methodType(void.class, Object.class);

MethodHandle getPropertyMH = LOOKUP
        .findStatic(System.class, "getProperty", type1);
MethodHandle printlnMH = LOOKUP
        .findVirtual(PrintStream.class, "println", type2);


String value2 = (String) getPropertyMH.invoke("java.home");
printlnMH.invoke(System.out, (Object) "Hello, world");
```

```java
// example Java
PrintStream out1 = System.out;

MethodHandle systemOutMH = LOOKUP
        .findStaticGetter(System.class, "out", PrintStream.class);

PrintStream out3 = (PrintStream) systemOutMH.invoke();
```

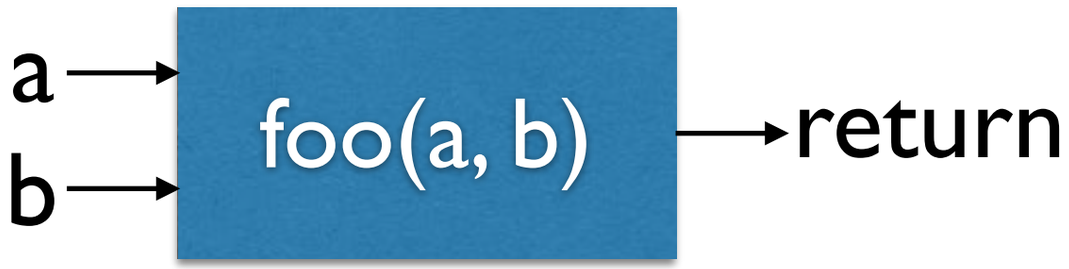
```java
class MyStruct {
    public String name;
}

// example Java
MyStruct ms = new MyStruct();
ms.name = "Tom";

MethodHandle nameSetter = LOOKUP
        .findSetter(MyStruct.class, "name", String.class);

nameSetter.invoke(ms, "Charles");
```

Adapters

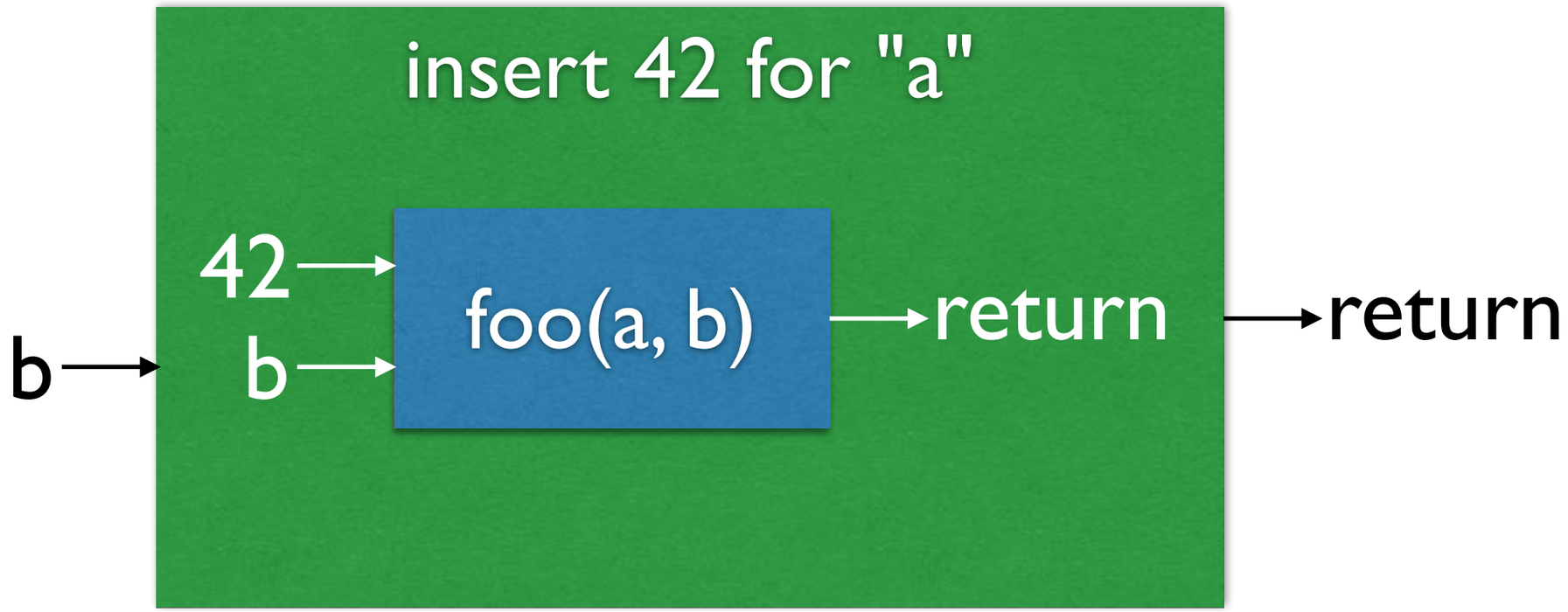
- Methods on j.l.i.MethodHandles
- Argument manipulation, modification
- Flow control and exception handling
- Similar to writing your own command-pattern utility objects

Argument Juggling

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


a
b
foo(a, b)
return

insertArgument

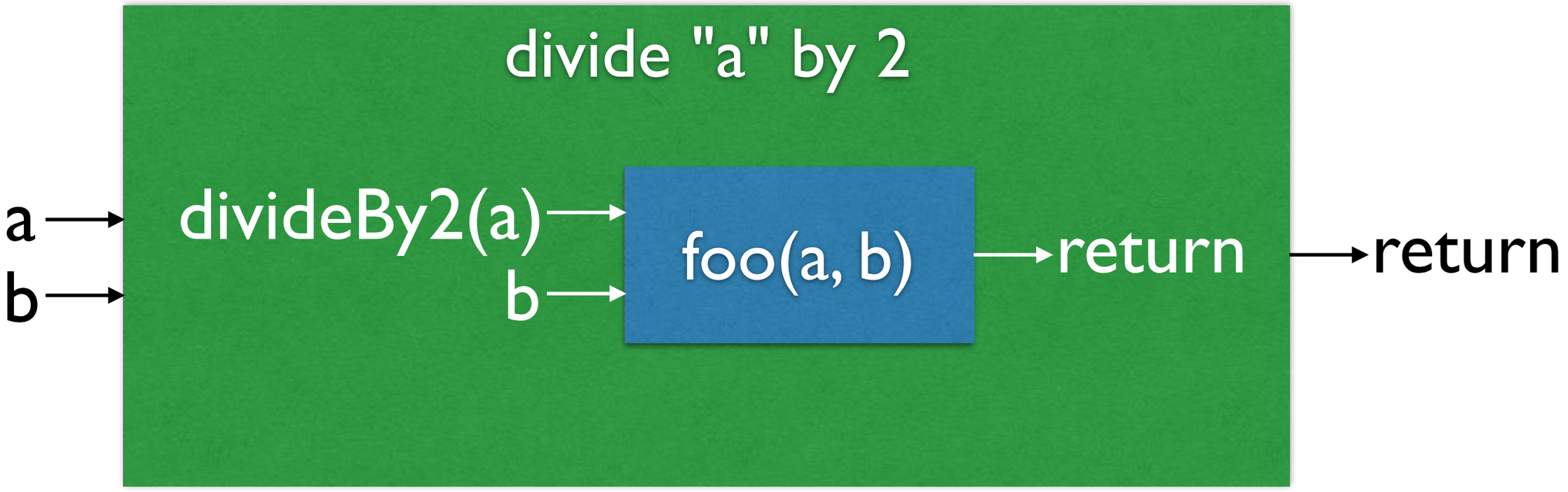


insertArgument(fooHandle, 0, 42)

Use a constant value 42 for argument 0

```
// insert is basically partial application
MethodHandle getJavaHomeMH =
        MethodHandles.insertArguments(getPropertyMH, 0, "java.home");
MethodHandle systemOutPrintlnMH =
        MethodHandles.insertArguments(printlnMH, 0, System.out);

// same as getProperty("java.home")
getJavaHomeMH.invokeWithArguments();
// same as System.out.println(...
systemOutPrintlnMH.invokeWithArguments("Hello, world");
```

filterArguments

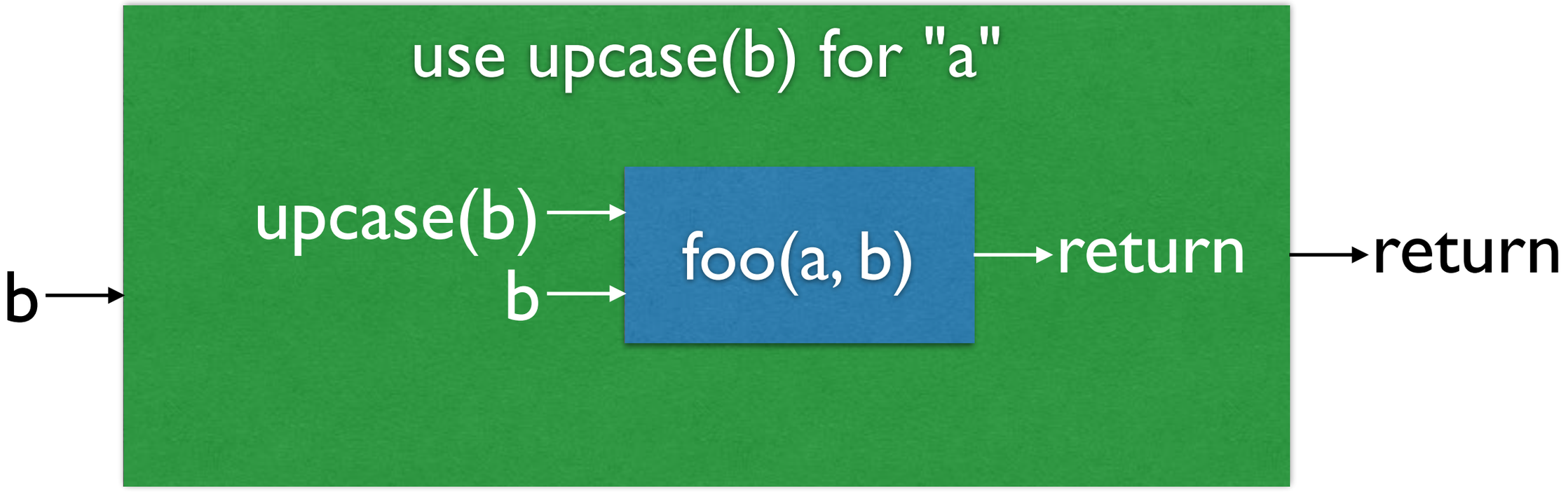


filterArguments(fooHandle, 0, divideHandle)

Pass argument 0 to divide and replace with result

```
// filter translates arguments before passing them on

// example Java
class UpperCasifier {
    public String call(String inputString) {
        return inputString.toUpperCase();
    }
}

// pointer to String.toUpperCase
MethodHandle toUpperCaseMH = LOOKUP.findVirtual(
        String.class,
        "toUpperCase",
        methodType(String.class));

// Change its type to Object ...(Object)
MethodHandle objectToUpperCaseMH =
        toUpperCaseMH.asType(methodType(Object.class, Object.class));

// Make a println that always upcases
MethodHandle upcasePrintlnMH =
        filterArguments(systemOutPrintlnMH, 0, objectToUpperCaseMH);

// prints out "THIS WILL BE UPCASED
upcasePrintlnMH.invokeWithArguments("this will be upcased");
```

foldArguments



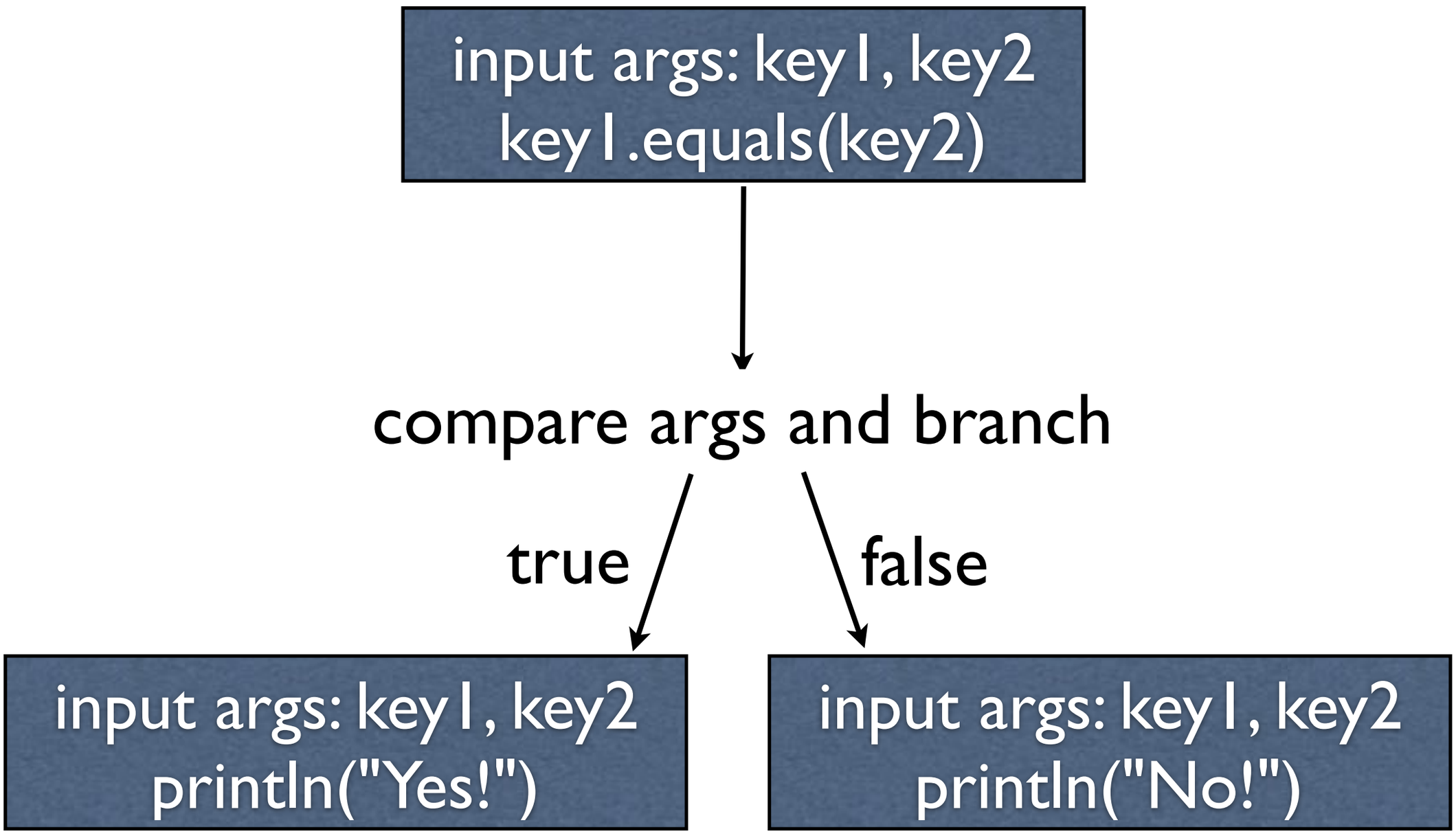
filterArguments(fooHandle, upcaseHandle)

Pass all arguments to upcase, insert result as argument 0

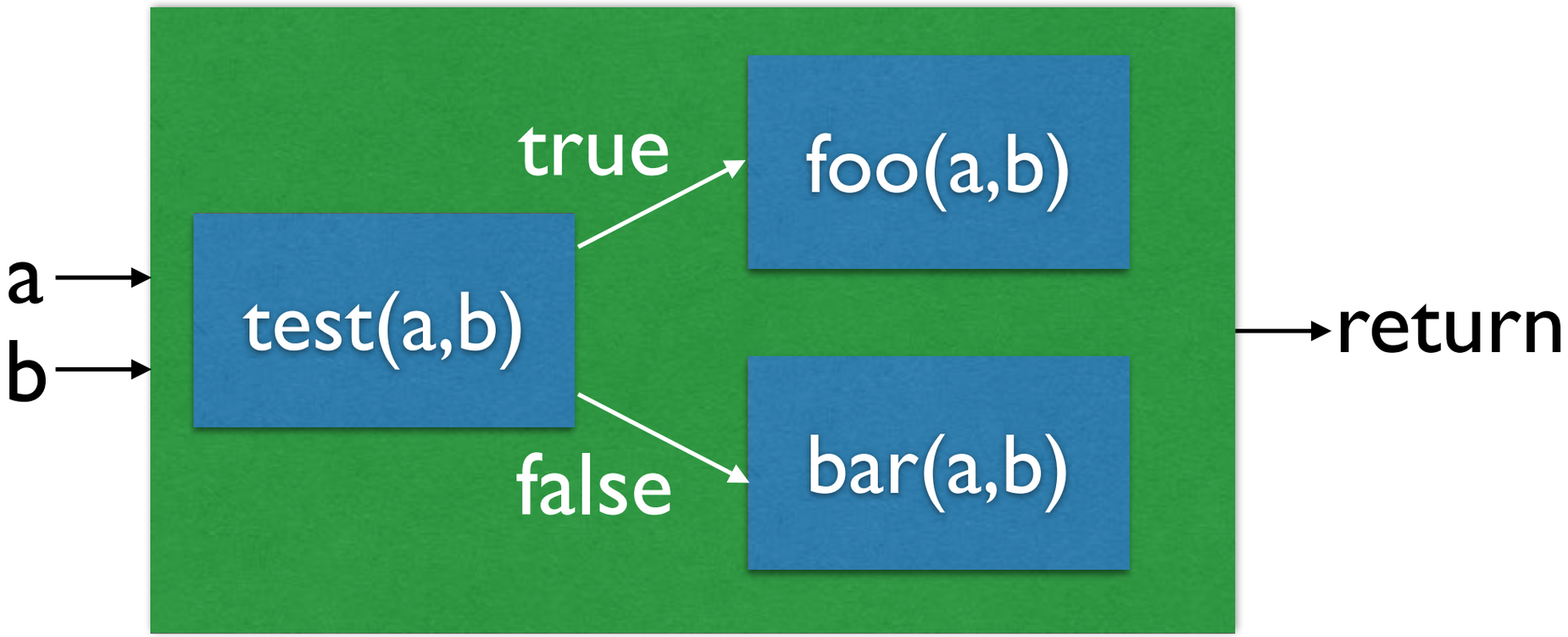
Flow Control

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

Branch based on test



guardWithTest



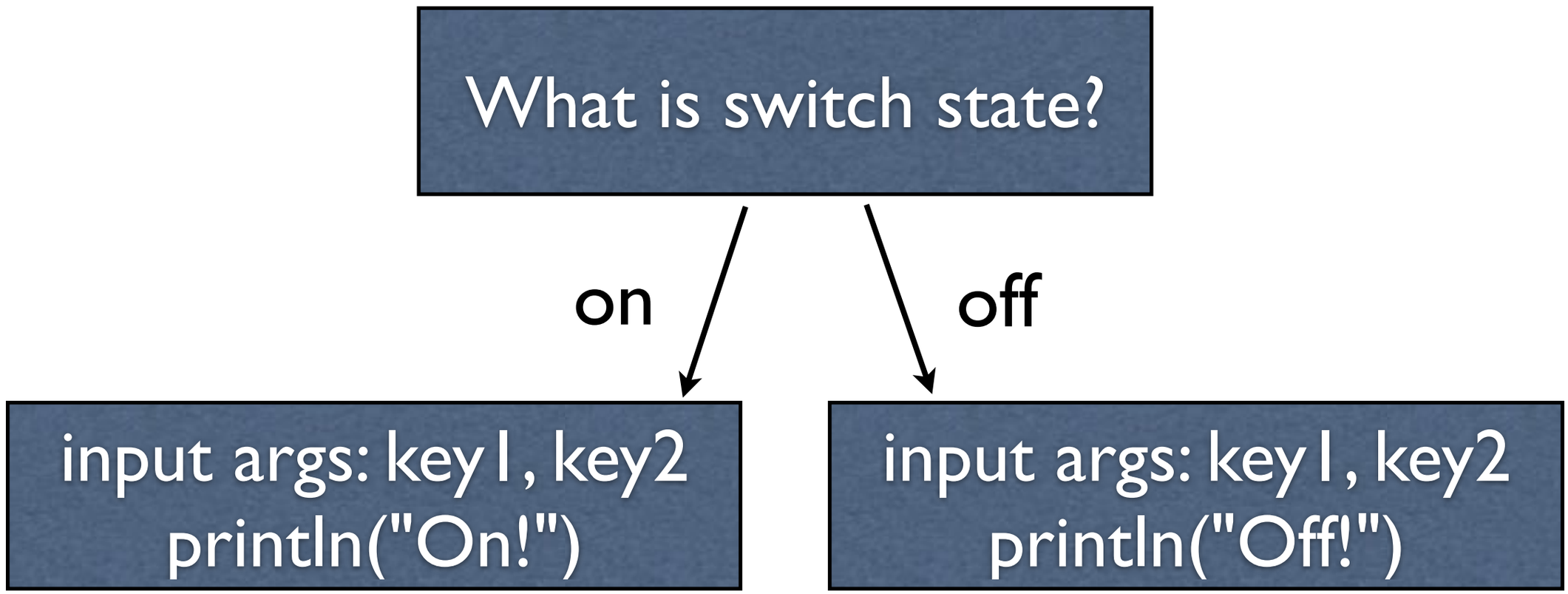
guardWithTest(testHandle, fooHandle, barHandle)

```
// boolean branch

// example Java
class UpperDowner {
    public String call(String inputString) {
        if (randomBoolean()) {
            return inputString.toUpperCase();
        } else {
            return inputString.toLowerCase();
        }
    }
}
```

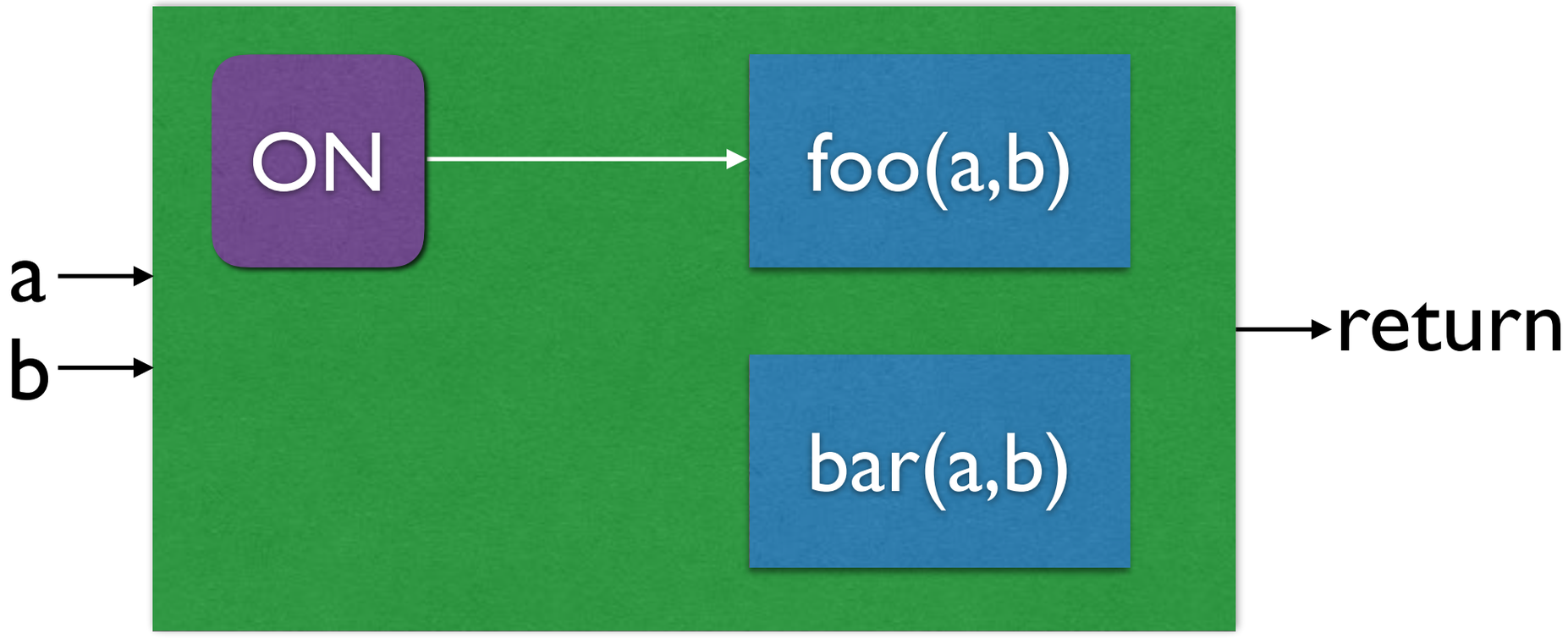
```java
// randomly return true or false
MethodHandle upOrDown = LOOKUP.findStatic(
        BasicHandles.class,
        "randomBoolean",
        methodType(boolean.class));

// guardWithTest calls boolean handle and branches
MethodHandle upperDowner = guardWithTest(
        upOrDown,
        toUpperCaseMH,
        toLowerCaseMH);
```

```java
// print out the result
MethodHandle upperDownerPrinter = filterArguments(
        systemOutPrintlnMH,
        0,
        upperDowner.asType(methodType(Object.class, Object.class)));

upperDownerPrinter.invoke("Hello, world"); // HELLO, WORLD
upperDownerPrinter.invoke("Hello, world"); // hello, world
upperDownerPrinter.invoke("Hello, world"); // HELLO, WORLD
upperDownerPrinter.invoke("Hello, world"); // HELLO, WORLD
upperDownerPrinter.invoke("Hello, world"); // hello, world
```

Branch on on/off switch

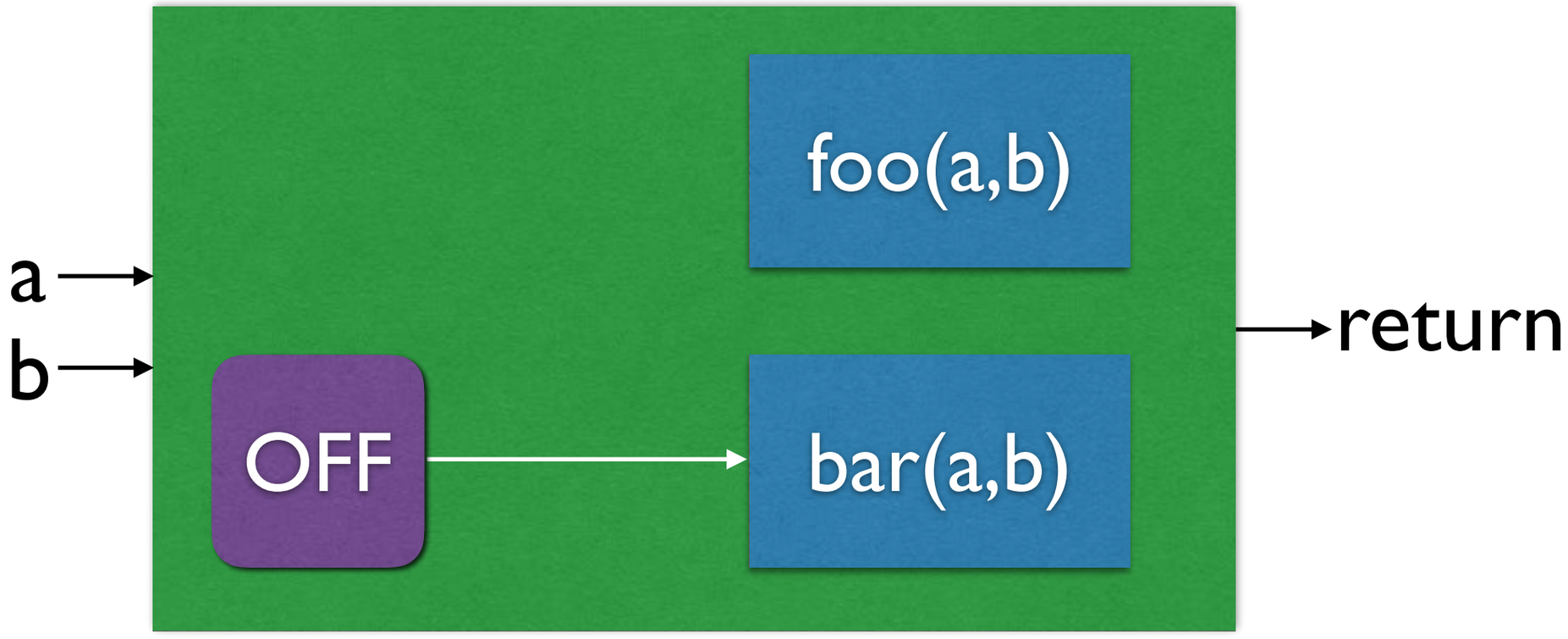


SwitchPoint.guardWithTest



guardWithTest(testHandle, fooHandle, barHandle)

SwitchPoint.guardWithTest



guardWithTest(testHandle, fooHandle, barHandle)

```
// switch branch

// example Java
 class UpperDownerSwitch {
     private volatile boolean on = true;
     public String call(String inputString) {
         if (on) {
             return inputString.toUpperCase();
         } else {
             return inputString.toLowerCase();
         }
     }

     public void turnOff() {
         on = false;
     }
}
```

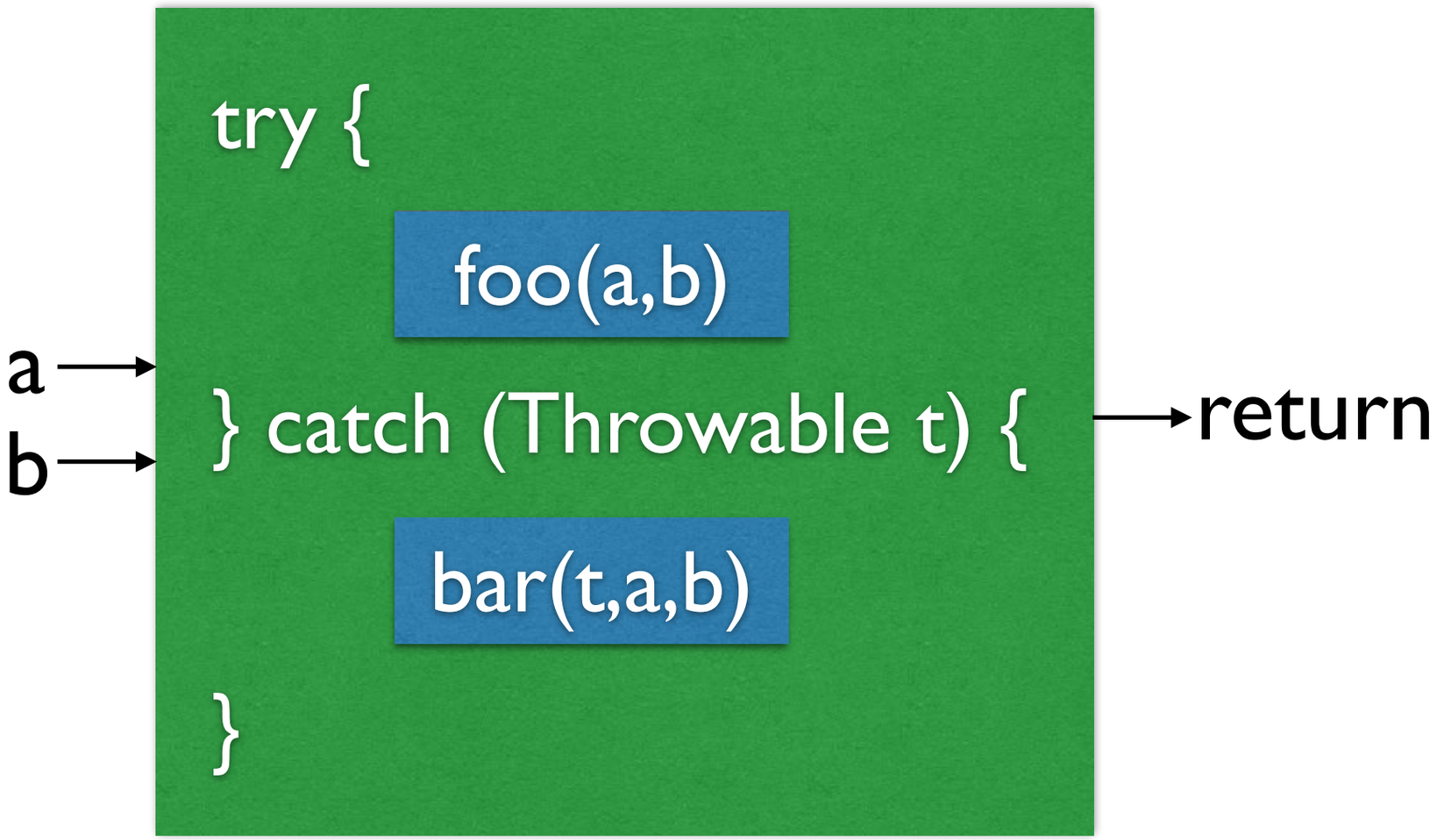
```
SwitchPoint upperLowerSwitch = new SwitchPoint();

MethodHandle upperLower =
        upperLowerSwitch.guardWithTest(toUpperCaseMH, toLowerCaseMH);
upperLower.invoke("MyString"); // => "MYSTRING"
upperLower.invoke("MyOtherString"); // => "MYOTHERSTRING"
SwitchPoint.invalidateAll(new SwitchPoint[]{upperLowerSwitch});
upperLower.invoke("MyString"); // => "mystring"
```

Think cache invalidation, volatile-free atomic boolean, ...

Exception Handling

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

catchException



catchException(fooHandle, Throwable.class, barHandle)

bytecode

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

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

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

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

```java
// 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
```

`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`

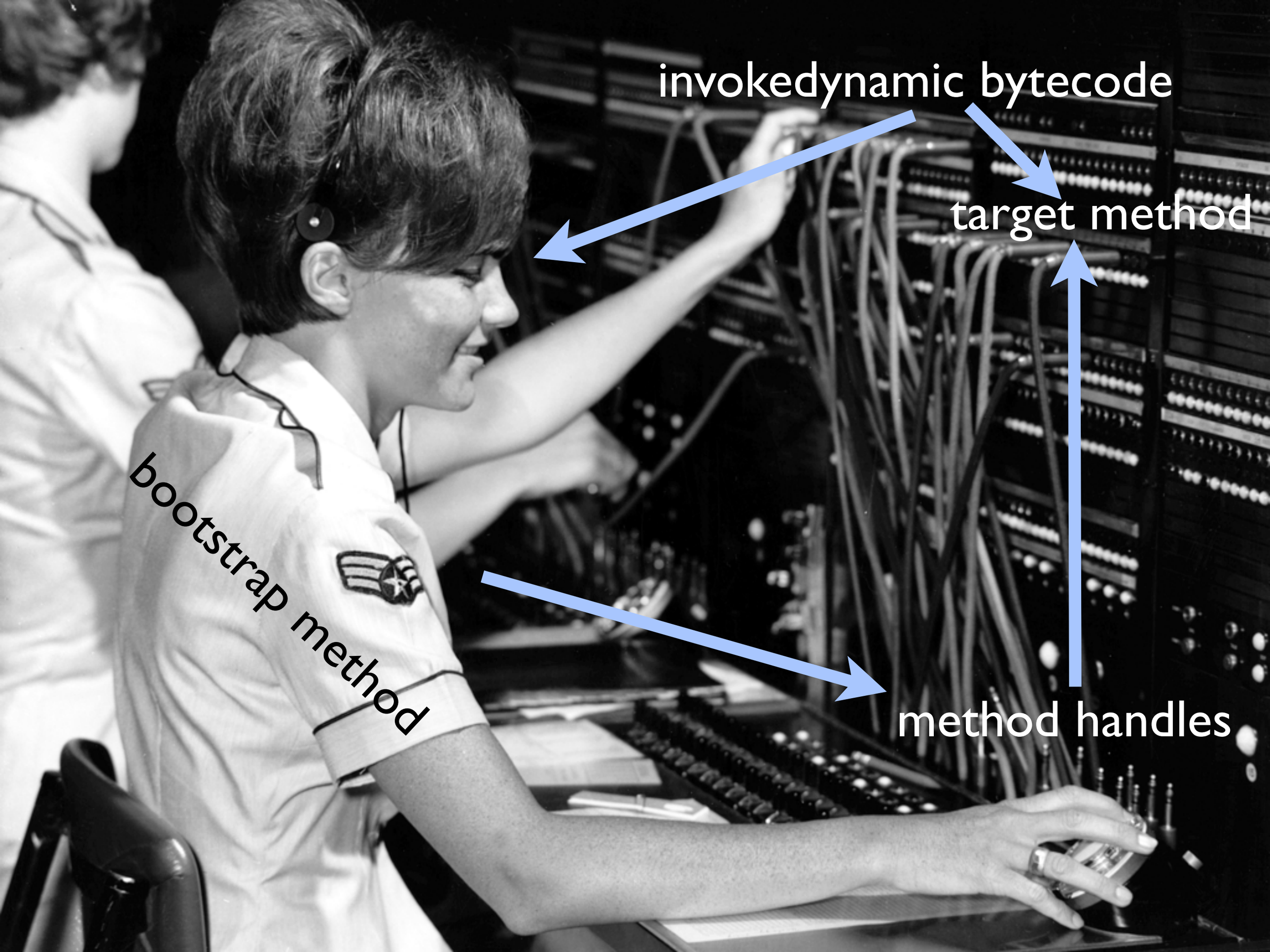
1. Confirm object's type implements interface
2. Confirm arguments are of correct type
3. Look up method on Java class
4. Cache method
5. Invoke method

`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
target method
bootstrap method
method handles

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

CallSite

- A holder for a MethodHandle
- Given to JVM by bootstrap method
  - Replaces invokedynamic bytecode
  - JVM watches it for changes
- Constant, Mutable, or Volatile

```
static int counter = 1;
public static String foo() {
    String result = "Hello, world #" + counter++;

    return result;
}
```

```
runnable = new Runnable() { public void run() {
    invokedynamic("foo", sig(String.class), bootHandle);

    aprintln(); // print out string

    voidreturn(); // return;
}
```

Do a dynamic call to "foo" returning String,
and call bootHandle to set it up.

```
Handle bootHandle = new Handle(
        Opcodes.H_INVOKESTATIC,
        p(SimpleBinding.class),
        "simpleBootstrap",
        sig(
                CallSite.class,
                MethodHandles.Lookup.class,
                String.class,
                MethodType.class));
```

Use simpleBootstrap(...) for the bootstrap "bootHandle"

```
public static CallSite simpleBootstrap(
        MethodHandles.Lookup lookup,
        String name,
        MethodType type) throws Exception {

    // Create and bind a constant site, pointing at the named method
    return new ConstantCallSite(
            lookup.findStatic(SimpleBinding.class, name, type));

}
```

Return an immutable CallSite pointing at "static String foo()"

```
// first call hits bootstrap, prepares call site
runnable.run(); // Hello, world #1

// subsequent calls go straight through
runnable.run(); // Hello, world #2
runnable.run(); // Hello, world #3
```

#2: Modifying the CallSite

- Toggle between two targets each call
  - j.l.i.MutableCallSite
- Call site sent into target MethodHandle
  - ...so we can modify it
- Trivial example of late binding
- InvokeBinder instead of java.lang.invoke

```java
public static void first(...) {
    System.out.println("first!");
    ... // use "second" next time
}
```

```java
public static void second(...) {
    System.out.println("second!");
    ... // use "first" next time
}
```

```
runnable = new Runnable() { public void run() {
    invokedynamic("first", sig(void.class), bootHandle);
    voidreturn();
}
```

Invoke "first" using bootHandle to bootstrap

```
Handle bootHandle = new Handle(
        Opcodes.H_INVOKESTATIC,
        p(SimpleBinding.class),
        "mutableCallSiteBootstrap",
        sig(
                CallSite.class,
                MethodHandles.Lookup.class,
                String.class,
                MethodType.class));
```

```
public static CallSite mutableCallSiteBootstrap(
    MethodHandles.Lookup lookup,
    String name,
    MethodType type) throws Exception {

    MutableCallSite mcs = new MutableCallSite(type);

    // The same thing with InvokeBinder
    MethodHandle target = Binder.from(void.class)
            .insert(0, lookup, mcs)
            .invokeStatic(lookup, SimpleBinding.class, name);

    mcs.setTarget(target);

    return mcs;
}
```

```
public static void first(
        MethodHandles.Lookup lookup,
        MutableCallSite mcs) throws Exception {

    // Look up "second" method and add Lookup and MutableCallSite
    MethodHandle second = Binder.from(void.class)
            .insert(0, lookup, mcs)
            .invokeStatic(lookup, SimpleBinding.class, "second");

    mcs.setTarget(second);

    System.out.println("first!");
}
```

```
public static void second(
        MethodHandles.Lookup lookup,
        MutableCallSite mcs) throws Exception {

    // Look up "second" method and add Lookup and MutableCallSite
    MethodHandle second = Binder.from(void.class)
            .insert(0, lookup, mcs)
            .invokeStatic(lookup, SimpleBinding.class, "first");

    mcs.setTarget(second);

    System.out.println("second!");
}
```

```
runnable.run(); // => "first!"
runnable.run(); // => "second!"
runnable.run(); // => "first!"
runnable.run(); // => "second!"
runnable.run(); // => "first!"
runnable.run(); // => "second!"
```

#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"]
- defN <name> '\n' <op1> ['\n' <op2> ...] '\n' end
    - define function with given ops
- One builtin: print (System.out.println)

Implementation

- Simple parser generates AST Nodes
- Compiler walks nodes, emits bytecode
- Main script becomes run()
- def creates additional methods

Call with zero args

```
push hello
send 0
```

Call with one arg

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

```
// builtin function print (no args)
public static void print() {
    System.out.println();
}
```

```
// builtin function print (one arg)
public static void print(String arg0) {
    System.out.println(arg0);
}
```

Define a function

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

StupidScript main()

```
// our source code in the file "stupid.script"
String script = readFile("stupid.script");

// parse the script
List<Node> ast = new Parser().parse(script);

// invoke the compiler
Runnable runnable = new Compiler().compile(ast);

runnable.run();
```

Compile nodes to bytecode

```
void emit(..., Node node) {
    switch (node.op) {

        case "push":
            ldc(node.arg);
            break;

        case "def0":
            newFunction(..., 0, node.arg, node.children);
            break;

        case "def1":
            newFunction(..., 1, node.arg, node.children);
            break;
...
```

```
case "send":
    int arity = Integer.parseInt(node.arg);

    // load self, since we may need it for the function
    body.aload(0);

    // method name and args are on stack

    // invokedynamic given a method name and arity args on stack
    switch (arity) {
        case 0:
            body.invokedynamic("send",
                sig(
                    void.class,
                    String.class,  // name of function to call
                    Object.class), // main script object
                DYNLANG_BOOTSTRAP);
            break;
        case 1:
            body.invokedynamic("send",
                sig(
                    void.class,
                    String.class,  // name of function to call
                    String.class,  // argument
                    Object.class), // main script object
                DYNLANG_BOOTSTRAP);
            break;
    }
    break;
```

```
public static CallSite dynlangBootstrap(
        MethodHandles.Lookup lookup,
        String name,
        MethodType type) throws Exception {

    MutableCallSite mcs = new MutableCallSite(type);
    MethodHandle send = Binder.from(type)
            .insert(0, lookup, mcs)
            .invokeStatic(lookup, StupidScript.class, "doSend");

    mcs.setTarget(send);

    return mcs;
}
```

```
public static void doSend(
        MethodHandles.Lookup lookup,
        MutableCallSite mcs, String name,
        Object self) throws Throwable {

    MethodHandle target = null;

    switch (name) {

        case "print":
            // no-arg builtins
            target = Binder.from(void.class, String.class, Object.class)
                    .drop(0, 2)
                    .invokeStatic(lookup, StupidScript.class, "print");
            break;

        default:
            // user-defined function
            target = Binder.from(void.class, String.class, Object.class)
                    .drop(0)
                    .cast(void.class, self.getClass())
                    .invokeVirtual(lookup, name);
            break;
    }

    mcs.setTarget(target);

    target.invoke(name, self);
}
```

```
$ cat stupid.script
def hello
push print
push Hello, world!
send 1
end

push hello
send 0

$ java StupidScript
Hello, world!
```

Your Turn

- Play with MethodHandles...they're fun!
- Try out invokedynamic
  - "JVM Bytecode for Dummies" talk
- https://github.com/headius/indy_deep_dive

More Information

- My blog: http://blog.headius.com
- Follow me: @headius
- Google: "headius invokedynamic"
- Many other talks about indy and bytecode