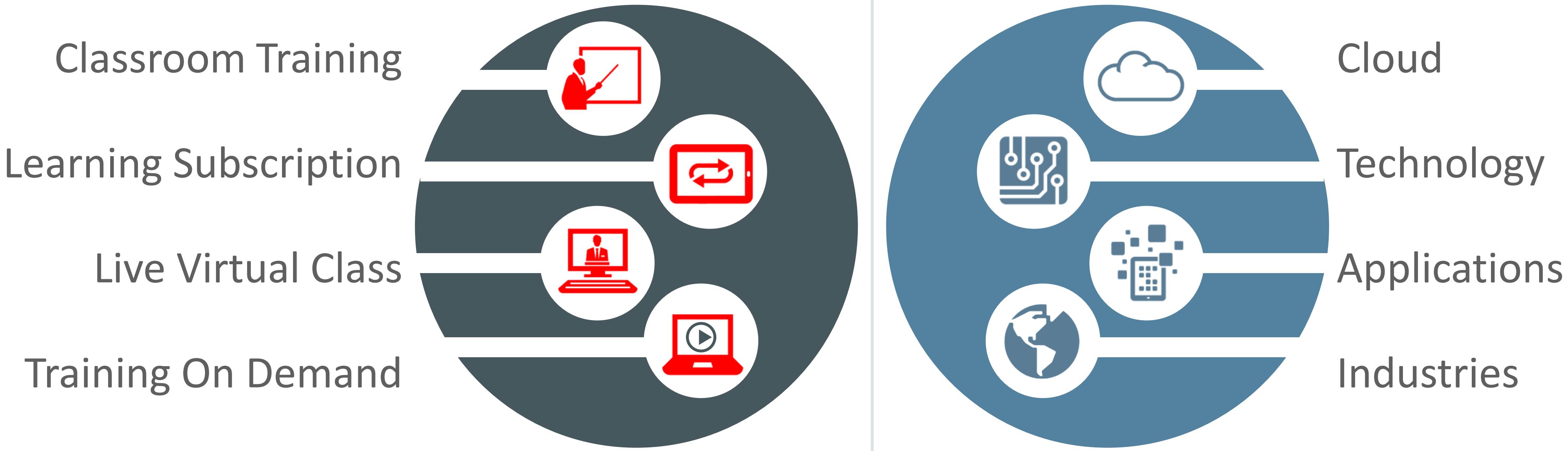
ORACLE®

JavaOne™

ORACLE®

Keep Learning with Oracle University

ORACLE UNIVERSITY



education.oracle.com

Session Surveys

Help us help you!!

- Oracle would like to invite you to take a moment to give us your session feedback. Your feedback will help us to improve your conference.
- Please be sure to add your feedback for your attended sessions by using the Mobile Survey or in Schedule Builder.

Invokedynamic for Mere Mortals

David Buck
Principal Member of Technical Staff
Java SE
October 26, 2015

Safe Harbor Statement

Program Agenda

Introduction

Target Audience

- Not compiler writers
- Curious

Motivation

- Understand javap output better
- Understand the value JVM has as a multi-language JVM

Da Vinci Machine Project

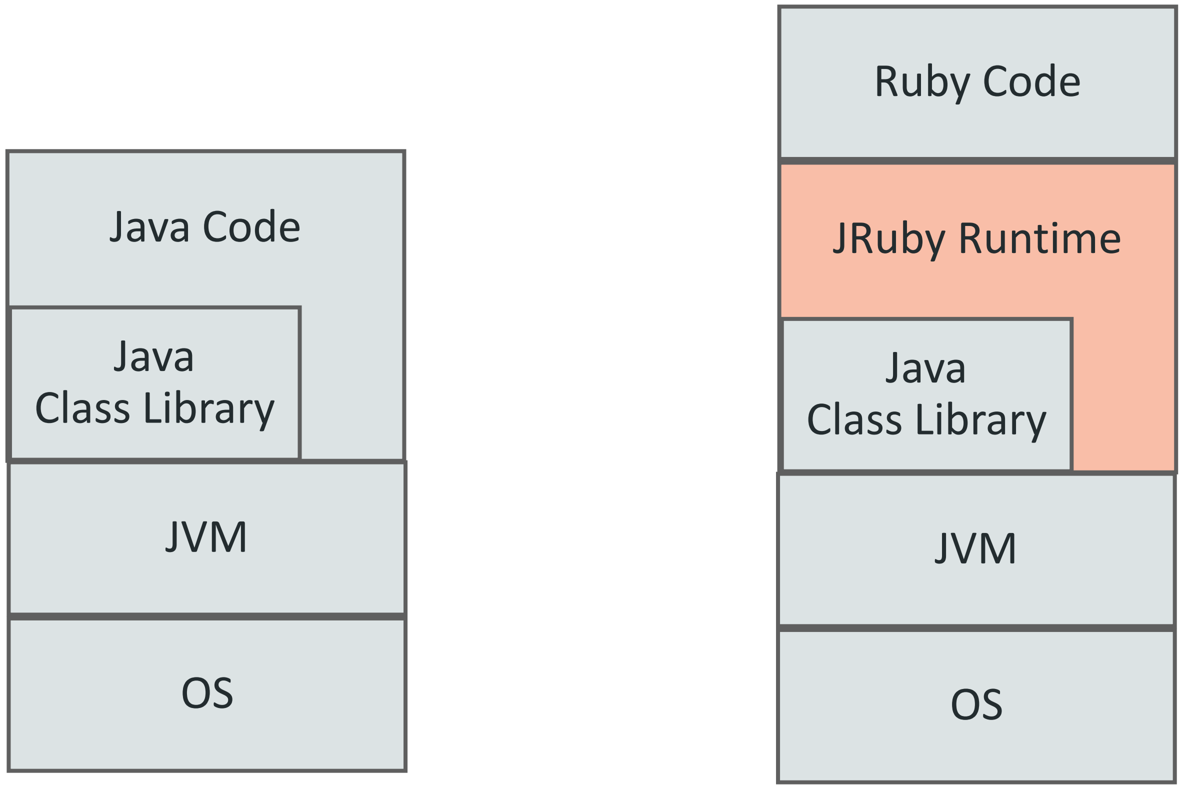
- The JVM is a great platform for running all sorts of languages
  - Great performance
  - Portability
  - Security (sandbox)
  - Pre-existing libraries and frameworks

(a small subset of) JVM languages

- JVM-specific
  - Scala
  - Clojure
  - Groovy
  - Ceylon
  - Fortress
  - Gosu
  - Kotlin

- Ported to JVM
  - JRuby
  - Jython
  - Smalltalk
  - Ada
  - Scheme
  - REXX
  - Prolog
  - Pascal
  - Common LISP

Language Runtime



non-Java language wish list

- Continuations
- Dynamic invocation
- Tail recursion
- Interface injection
- Other stuff

non-Java language wish list

- Continuations
- Dynamic invocation
- Tail recursion
- Interface injection
- Other stuff

What is dynamic typing?

What is dynamic typing?

```
def addtwo(a, b)
    a + b;
end
```

What is dynamic typing?

We do not know what the types are until runtime

statically-typed vs. dynamically-typed

When do we type check / link?

- Compilation time (javac)
- Runtime

Compile-time checking / linking

- Catch errors early
- Limits the type of code we can write (false positives)

Run time checking / linking

- Allow more freedom of programming (less false positives)
- Less guarantees about runtime behavior

dynamic typing != type inference

```
object InferenceTest1 extends App {
  val x = 1 + 2 * 3           // the type of x is Int
  val y = x.toString()        // the type of y is String
  def succ(x: Int) = x + 1    // succ returns Int values
}
```

(**Shamelessly** copied from http://docs.scala-lang.org/tutorials/tour/local-type-inference.html)

dynamic typing != week typing

```
a = "40"
b = a + 2
```

Dynamically-typed languages

- Allow more programs, but have to do more runtime checking.
- No perfect type information at compile time

Polymorphism != Dynamic typing (?!)

```
public String bar(Object o) {
        return "You passed me " + o.toString();
}
```

The original invocation lineup

- invokestatic
  - Class method
- invokevirtual
  - Instance method
- invokeinterface
  - Interface method
- Invokespecial
  - Everything else (private, super class, constructors)

The original invocation lineup

- **invokestatic**
  - **Class method**
- invokevirtual
  - Instance method
- invokeinterface
  - Interface method
- Invokespecial
  - Everything else (private, super class, constructors)

invokestatic

```java
public class InvokeStaticExample {
    public static void main(String[] args) {
        InvokeStaticExample.foo();
    }

    public static void foo() {
        System.out.println("I am foo!");
    }
}
```

The original invocation lineup

- invokestatic
  - Class method
- **invokevirtual**
  - **Instance method**
- invokeinterface
  - Interface method
- Invokespecial
  - Everything else (private, super class, constructors)

invokevirtual

```
public class InvokeVirtualExample {
    public static void main(String[] args) {
        InvokeVirtualExample ive = new InvokeVirtualExample();
        ive.foo();
    }

    public void foo() {
        System.out.println("I am foo!");
    }
}
```

The original invocation lineup

- invokestatic
  - Class method
- invokevirtual
  - Instance method
- **invokeinterface**
  - **Interface method**
- Invokespecial
  - Everything else (private, super class, constructors)

invokeinterface

```
public class InvokeInterfaceExample
  implements MyInterface {

    public static void main(String[] args)
  {

        MyInterface iie = new
  InvokeInterfaceExample();

        iie.foo();

    }


    public void foo() {

        System.out.println("I am foo!");

    }

}
```

```
interface MyInterface {

    public void foo();

}
```

The original invocation lineup

- invokestatic
  - Class method
- invokevirtual
  - Instance method
- invokeinterface
  - Interface method
- **Invokespecial**
  - **Everything else (private, super class, constructors)**

invokespecial

```
public class InvokeSpecialExample {
    public static void main(String[] args) {
        InvokeSpecialExample ise = new InvokeSpecialExample();
        ise.foo();
    }

    private void foo() {
        System.out.println("I am foo!");
    }
}
```

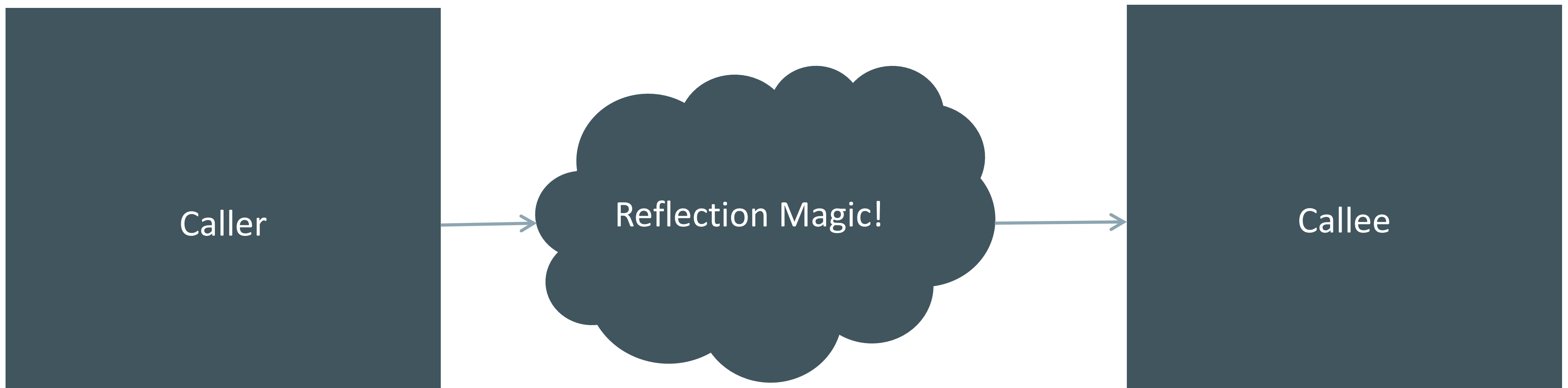
Poor dynamic languages on JVM?

- invocation logic is not baked into the JVM like it is for Java
- we need to fall back on reflection

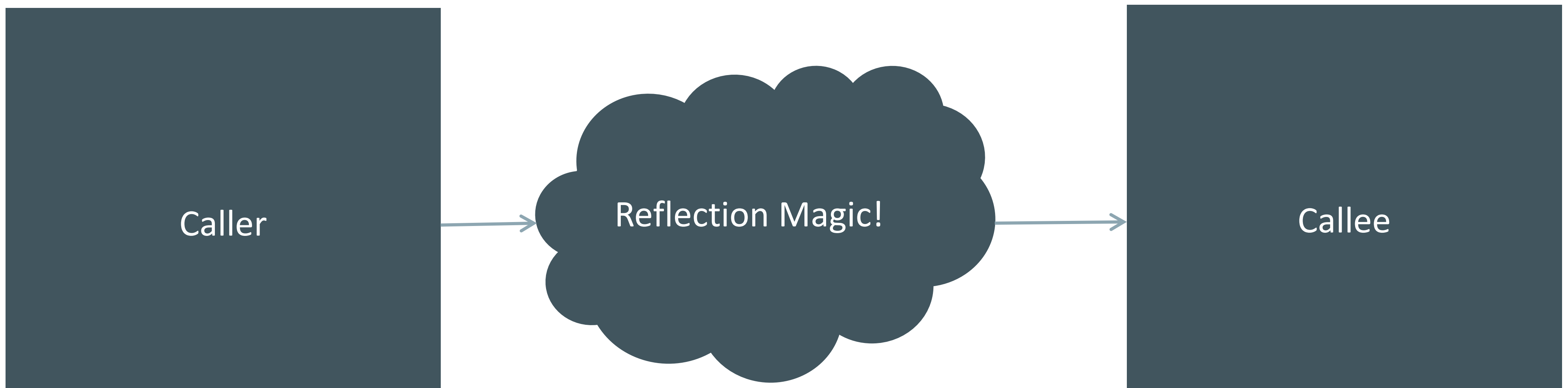
Reflection is slow

- security check on each invocation
- all arguments are Objects (boxing)

What the JVM doesn’t know **can** hurt it



Reflection prevents inlining!



No one writes code like this

```
if (false) {
        // do some important stuff...
        System.out.println("I'm important!");
}
```

Or this…

```
boolean cond = true;
if (cond) {
    // do some important stuff...
    System.out.println("I'm important!");
}
```

But we do write stuff like

```
public void methodB() {
        // ...
        methodA(false);
        // ...
}
```

```
public void methodA(boolean optionalStuff) {
        // ...
        if (optionalStuff) {
                // do some optional, but important stuff...

                System.out.println("I'm important sometimes!");
        }
        // ...
}
```

JSR-292

- java.lang.invoke API

  A “better reflection”

- invokedynamic bytecode

  Allows us to dispatch to linkage logic defined by invoke API

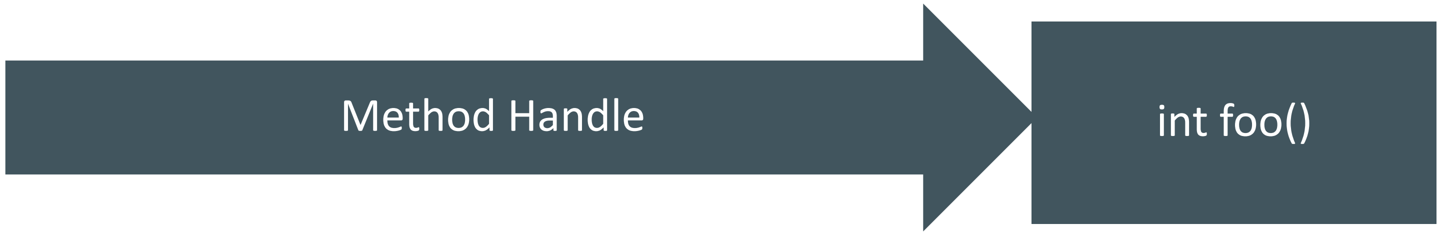
invokedynamic

- We call it “indy”
- No clear way to express in Java language
- Important milestone for JVM
  - First new instruction in decades
  - First new JVM feature to only (mainly) target non-java languages

java.lang.invoke API

- MethodHandle
- CallSite
- Bootstrap Method (BSM)

MethodHandle



MethodHandle

- Points to a method
- Is a “function pointer” (am I allowed to say this?)
- Polymorphic signature

MethodHandle Performance

MethodHandle Performance

- Early performance was not ideal

MethodHandle Performance

- Early performance was not ideal
- Performance improved tremendously with lambda forms

MethodHandle Performance

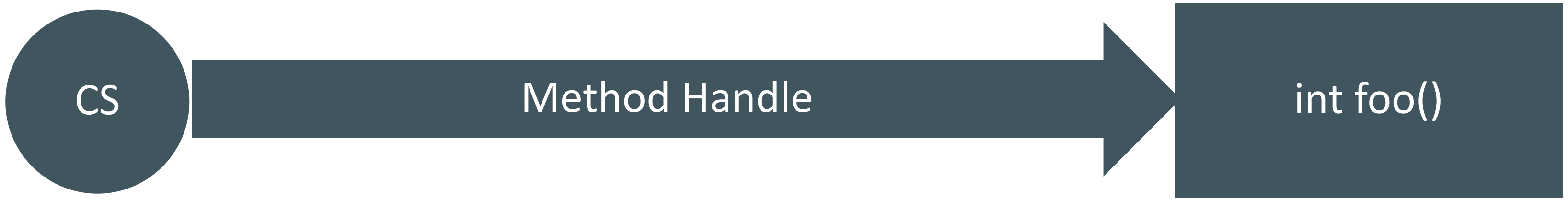
- Early performance was not ideal
- Performance improved tremendously with lambda forms
- Is now often significantly faster than reflection

MethodHandle Performance

- Early performance was not ideal
- Performance improved tremendously with lambda forms
- Is now often significantly faster than reflection
- Can be used independently of invokedynamic

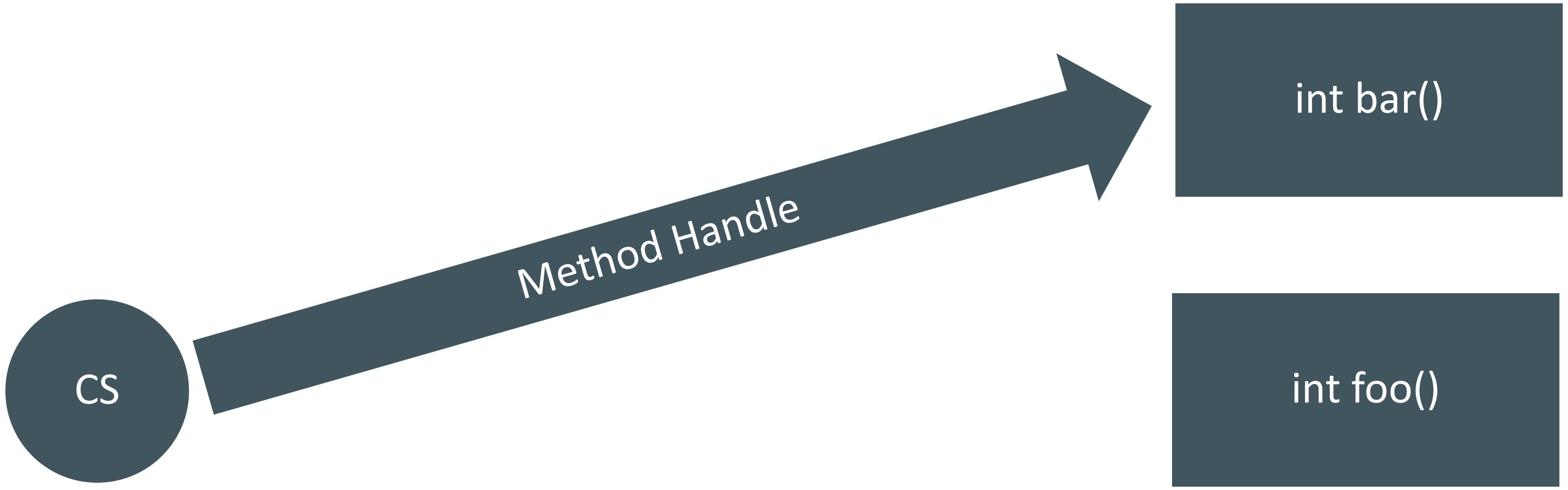
CallSite

```
private void doStuff();
  descriptor: ()V
  flags: ACC_PRIVATE
  Code:
    stack=2, locals=2, args_size=1
       0: new           #7
       3: dup
       4: invokespecial #8
       7: astore_1
       8: aload_1
       9: aload_0

      10: invokedynamic #9,  0

      15: invokevirtual #10
      18: return
```



CallSite

```
private void doStuff();
   descriptor: ()V
   flags: ACC_PRIVATE
   Code:
     stack=2, locals=2, args_size=1
        0: new           #7
        3: dup
        4: invokespecial #8
        7: astore_1
        8: aload_1
        9: aload_0

       10: invokedynamic #9,  0

       15: invokevirtual #10
       18: return
```



CallSite

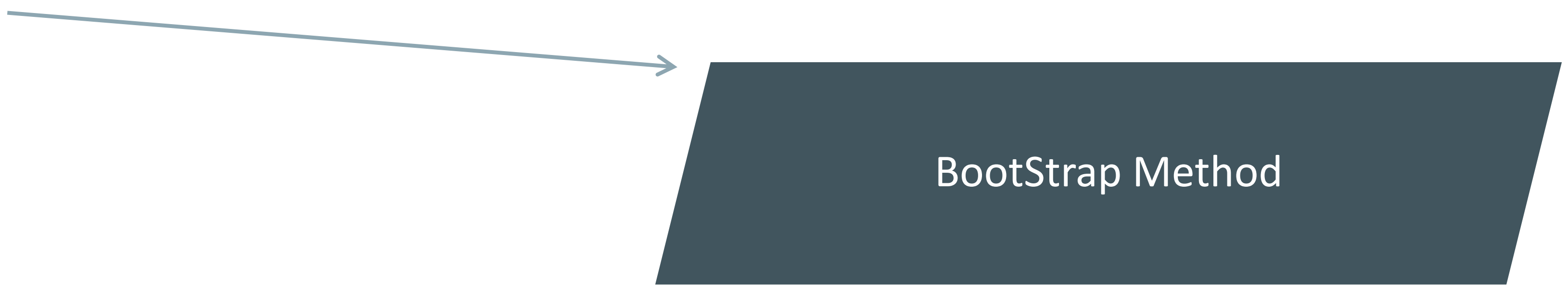
- Reifies Indy invocation side
- Has a MethodHandle

Bootstrapping Step 1

```
private void doStuff();
  descriptor: ()V
  flags: ACC_PRIVATE
  Code:
    stack=2, locals=2, args_size=1
       0: new           #7
       3: dup
       4: invokespecial #8
       7: astore_1
       8: aload_1
       9: aload_0

      10: invokedynamic #9,  0

      15: invokevirtual #10
      18: return
```

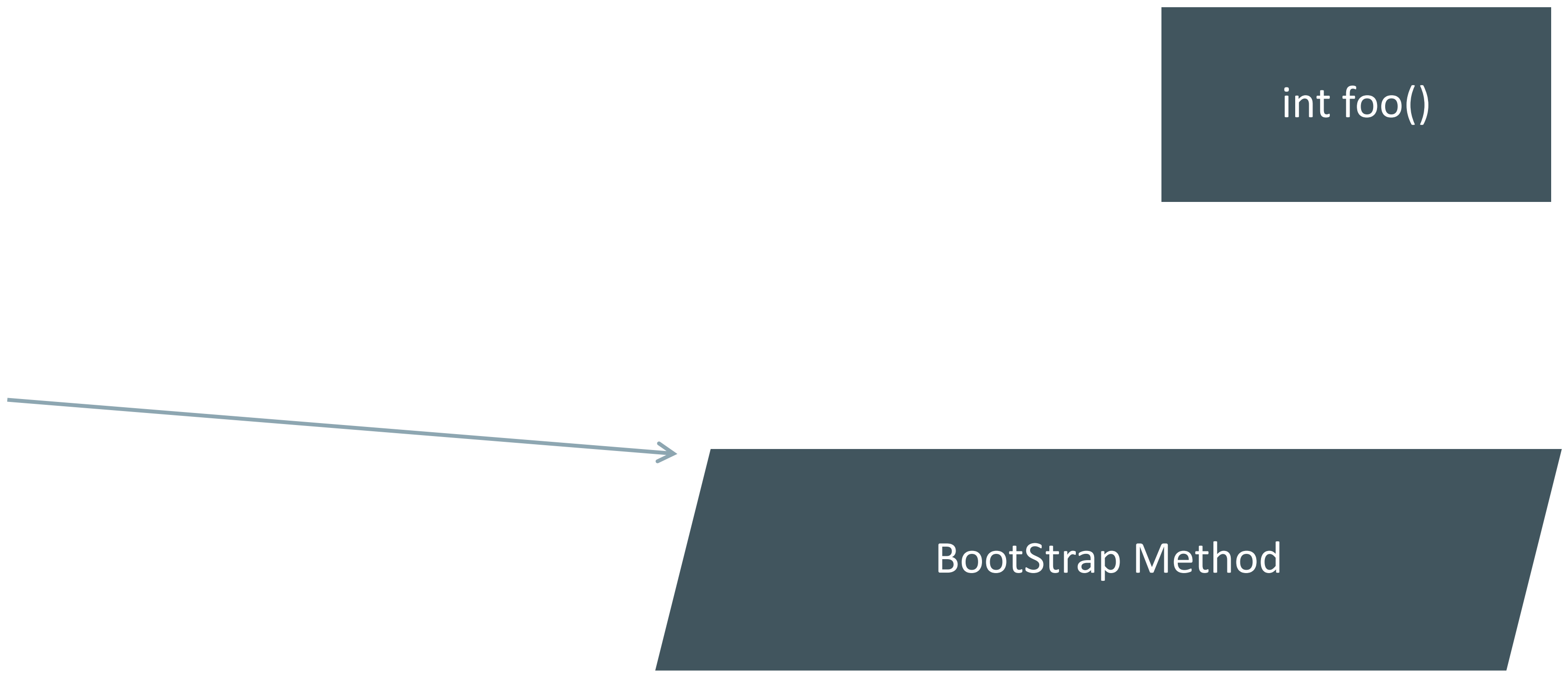
Bootstrapping Step 2

```
private void doStuff();
  descriptor: ()V
  flags: ACC_PRIVATE
  Code:
    stack=2, locals=2, args_size=1
       0: new           #7
       3: dup
       4: invokespecial #8
       7: astore_1
       8: aload_1
       9: aload_0

      10: invokedynamic #9,  0

      15: invokevirtual #10
      18: return
```



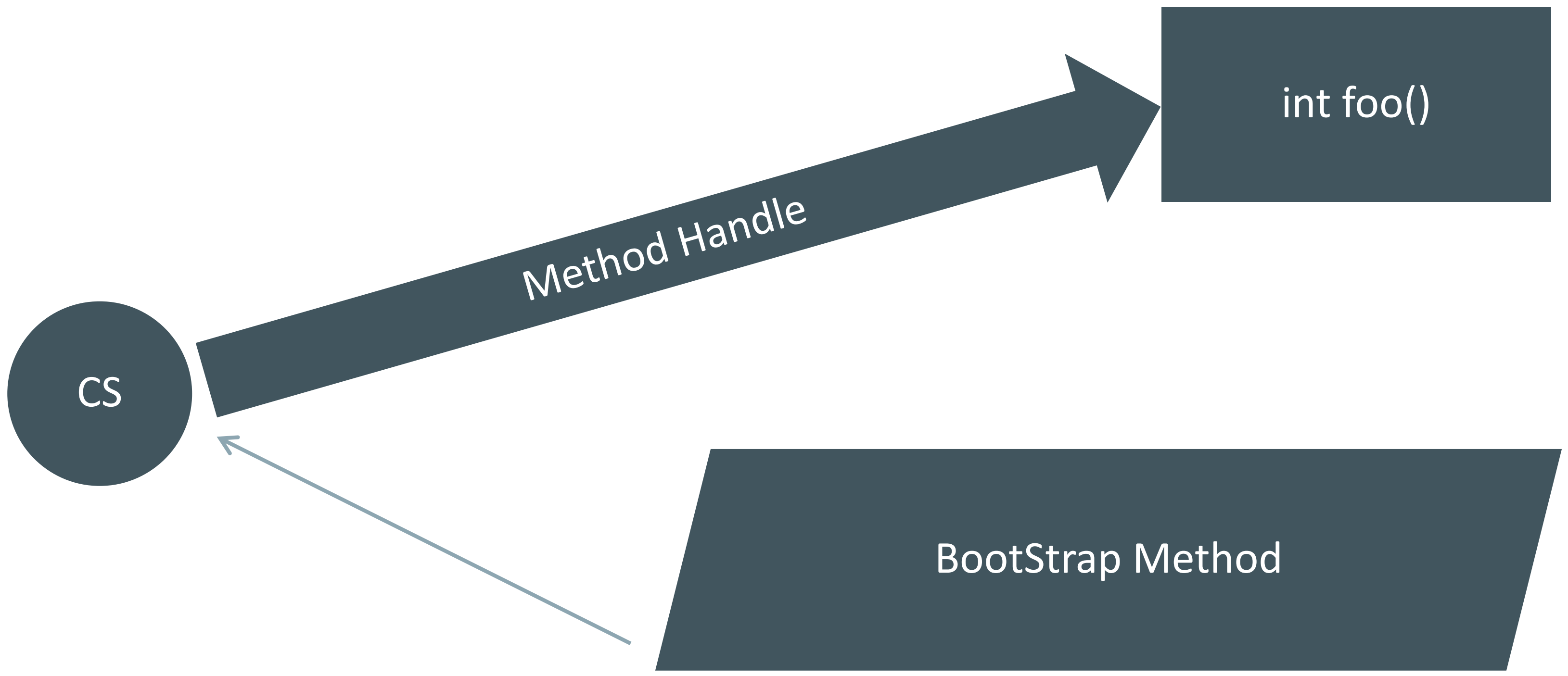
Bootstrapping Step 3

```
private void doStuff();
   descriptor: ()V
   flags: ACC_PRIVATE
   Code:
     stack=2, locals=2, args_size=1
        0: new           #7
        3: dup
        4: invokespecial #8
        7: astore_1
        8: aload_1
        9: aload_0

       10: invokedynamic #9,  0

       15: invokevirtual #10
       18: return
```



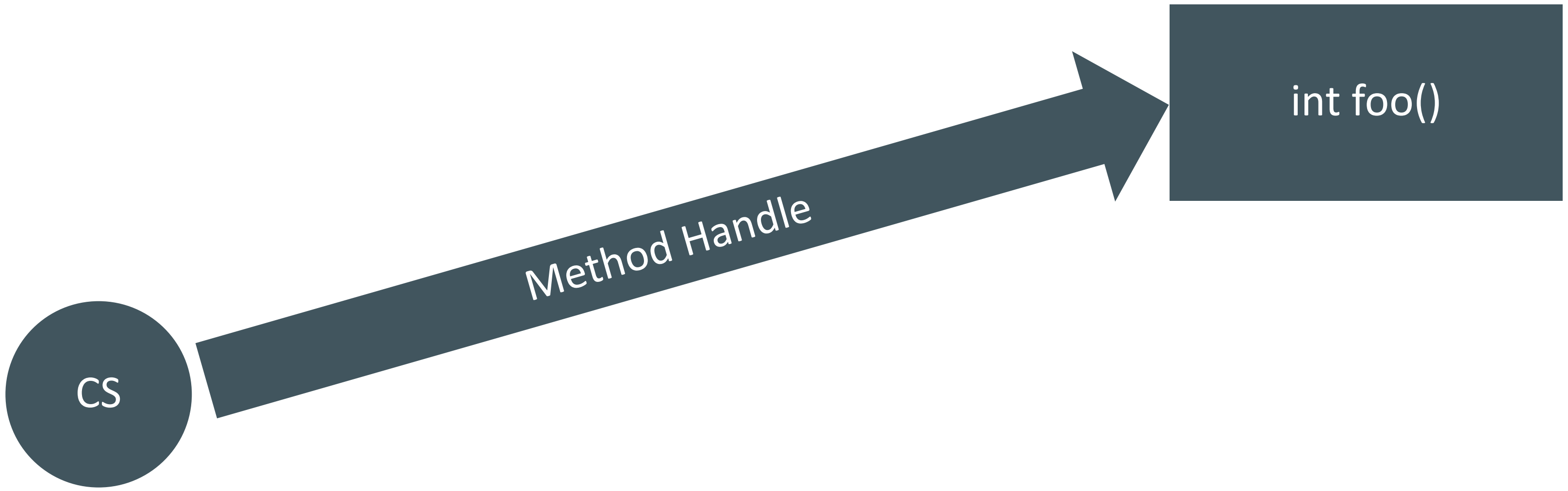
Bootstrapping Step 4

```
private void doStuff();
  descriptor: ()V
  flags: ACC_PRIVATE
  Code:
    stack=2, locals=2, args_size=1
       0: new           #7
       3: dup
       4: invokespecial #8
       7: astore_1
       8: aload_1
       9: aload_0

      10: invokedynamic #9,  0

      15: invokevirtual #10
      18: return
```



Bootstrapping Step 5

```
private void doStuff();
   descriptor: ()V
   flags: ACC_PRIVATE
   Code:
     stack=2, locals=2, args_size=1
        0: new           #7
        3: dup
        4: invokespecial #8
        7: astore_1
        8: aload_1
        9: aload_0

       10: invokedynamic #9,  0

       15: invokevirtual #10
       18: return
```



Bootstrap Method

- Only called on the first invocation of each indy bytecode
- Returns a CallSite

"Dr Martens, black, old" by Tarquin
is licensed under CC BY-SA 3.0

Indy lifecycle

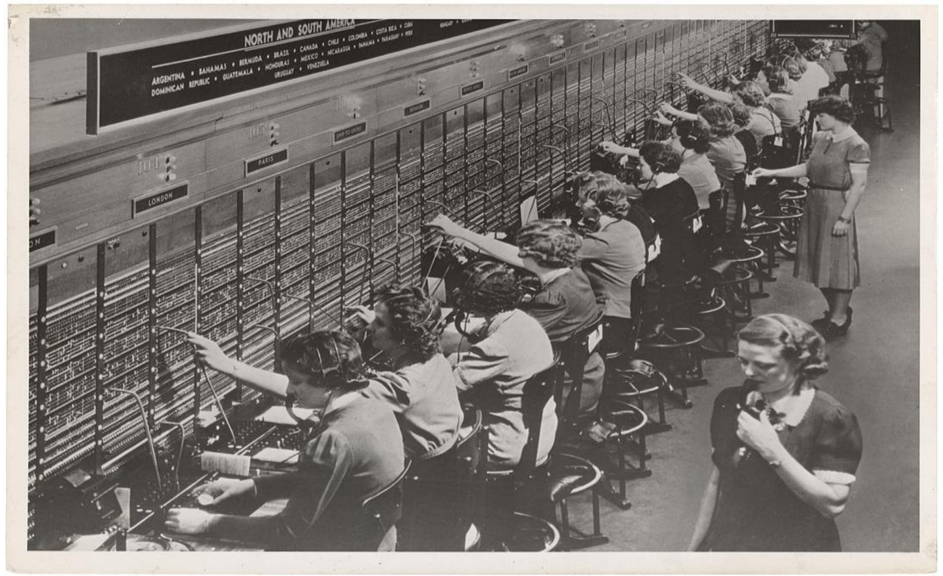
Initial Invocation

1. A specific indy invocation is executed for the first time
2. Bootstrap method is called and if finds (generates?!) a method to run
3. Botstrap method returns a permanent CallSite object for this indy invocation
4. We jump to the method pointed to by the CallSite

Indy Lifecycle

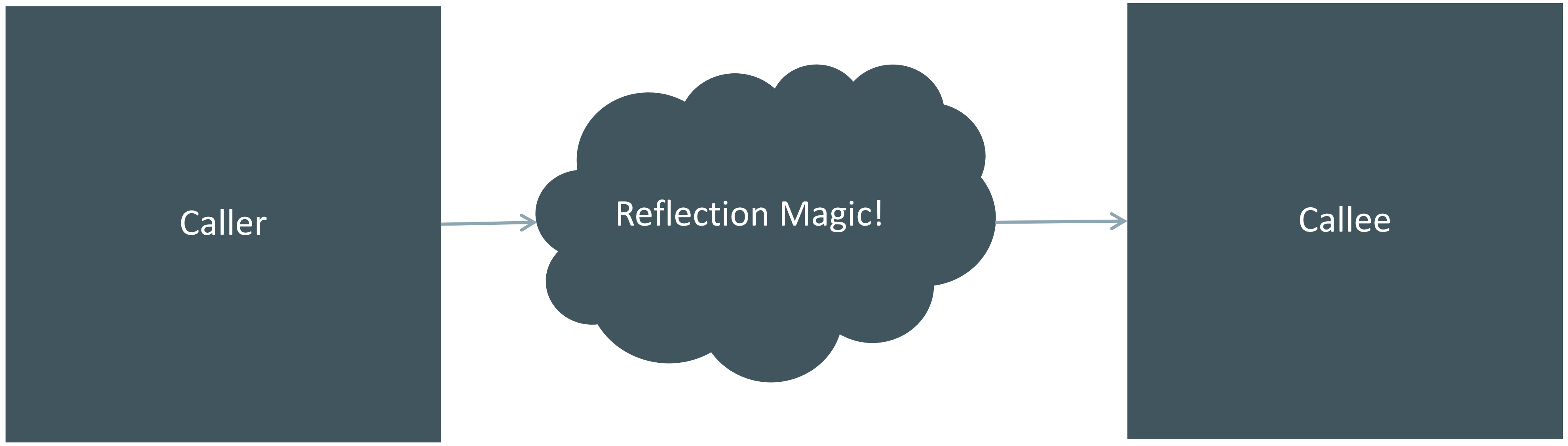
All subsequent calls

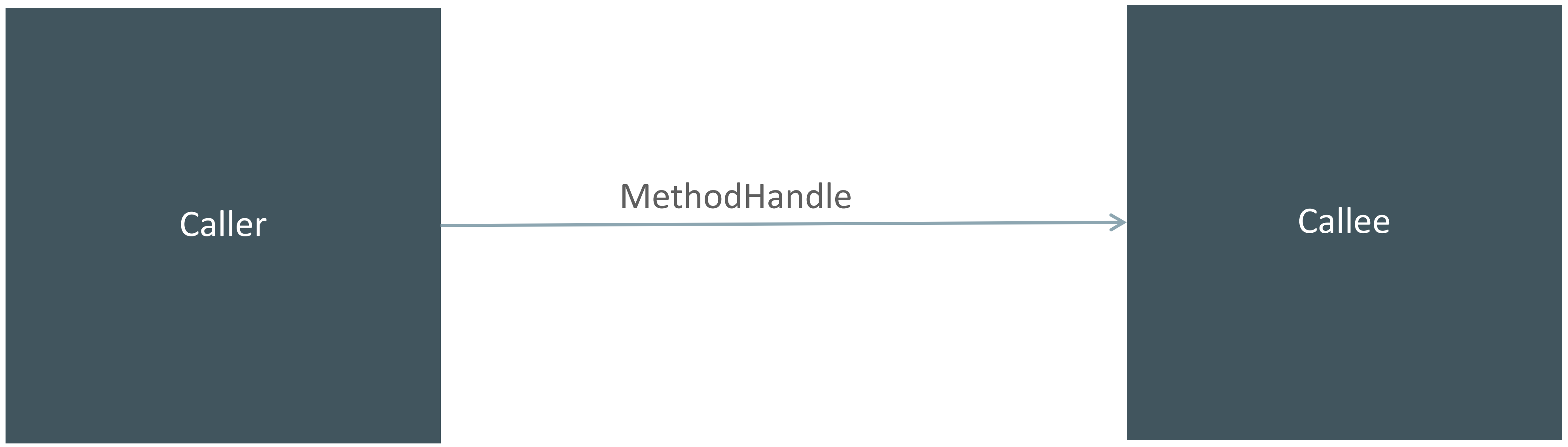
We jump to the method pointed to by the CallSite


NORTH AND SOUTH AMERICA
ARGENTINA · BAHAMAS · BERMUDA · BRAZIL · CANADA · CHILE · COLOMBIA · COSTA RICA · CUBA
DOMINICAN REPUBLIC · GUATEMALA · HONDURAS · MEXICO · NICARAGUA · PANAMA · PARAGUAY · PERU
URUGUAY · VENEZUELA
LONDON
PARIS

Picture from
National Archives and Records Administration

This performance tragedy becomes




Caller
MethodHandle
Callee

Linkage != Invocation

Linkage != Invocation

- Linkage (i.e. bootstrap)
  - Usually only needs to be done once
  - Is expensive

Linkage != Invocation

- Linkage (i.e. bootstrap)
  - Usually only needs to be done once
  - Is expensive
- Invocation
  - Done a **lot**
  - Only needs a jmp/call (and possibly a guard)

Linkage != Dispatch

- Avoid the cost of linkage on almost every call

Takeaways

Takeaways

- Invokedynamic lets us programmatically alter linkage

Takeaways

- Invokedynamic lets us programmatically alter linkage
- Then it gets out of the way! (linkage != invocation)

Takeaways

- Invokedynamic lets us programmatically alter linkage
- Then it gets out of the way! (linkage != invocation)
- The Invoke API can often be used without indy

Takeaways

- Invokedynamic lets us programmatically alter linkage
- Then it gets out of the way! (linkage != invocation)
- The Invoke API can often be used without indy
- JVM is a great platform for just about any language!

Resources

- JVM Language Summit

  http://openjdk.java.net/projects/mlvm/jvmlangsummit/
- Linkers & Loaders book

  http://linker.iecc.com/
- John Rose’s Blog

  https://blogs.oracle.com/jrose/

Thank You!

JavaOne™

ORACLE®

ORACLE®