

# Towards a Universal VM

The JVM is not just for the  
Java Language anymore

**Brian Goetz, Sr. Staff Engineer**  
**Alex Buckley, Spec lead, Java Language & VM**  
**Sun Microsystems**

# Overview

The JVM has, in large part, been the engine behind the success of the Java Language.

In years to come,  
it will power the success of other languages too.

# Agenda

Virtual machines

The Java Virtual Machine

Dynamic method invocation

Interface injection

JSR 292

# Virtual Machines

A *virtual machine* is a software implementation of a specific computer architecture

Could be a real hardware architecture or a fictitious one

*System* virtual machines simulate a complete computer system

VMWare, VirtualBox, VirtualPC, Parallels

Usually implement a real hardware architecture (e.g., X86)

*Process* virtual machines host a single application

Like the JVM

Usually implement a fictitious instruction set designed for a specific purpose

Instruction set can be chosen to maximize implementation flexibility

Initial implementations of VMs are often interpreted (and slow)

More mature implementations allow sophisticated compilation techniques

# Isolation and Abstraction

VMs isolate the hosted application from the host system

- VM appears as an ordinary process in the host system

- Applications running in separate VMs are isolated from each other

VMs isolate the host system from the hosted application

- VM acts as intermediary between hosted application and host system

- Hosted application can only access resources provided by the VM

VMs provide *a higher level of abstraction*

- Sensible layer for portability across underlying platforms

- Abstracts away low-level architectural considerations

  - Size of register set, hardware word size

- 1990s buzzword: ANDF (Architecture-Neutral Distribution Format)

  - Write once, run anywhere

# VMs win as compilation targets

Today, it is silly for a compiler to target actual hardware

- Much more effective to target a VM

- Writing a native compiler is lots more work!

Languages need runtime support

- C runtime is tiny and portable (and wimpy)

- More sophisticated language runtimes need

  - Memory management

  - Security

  - Reflection

  - Concurrency control

  - Libraries

  - Tools (debuggers, profilers, etc)

Many of these features are baked into VMs

# VMs win as compilation targets

If the VM doesn't provide these features, you have two choices

- Reinvent them yourself (lots of work!)

- Do without them

If the VM does provide them, you'll use them

- Less work

- Makes your programming language better

Targeting an existing VM also reuses libraries, tools

- Platform features such as reflection, threads

- APIs for tools (JVMTI), management, monitoring

- Rich ecosystems of tools (debuggers, profilers, IDEs)

VM-based facilities become common across languages

- Java code can reflectively call JRuby code

- Java objects and Jython objects are garbage-collected together

# VMs win as compilation targets

Dynamic ("Just In Time") compilation often yields better performance than static compilation

- More information available at runtime gives better optimization decisions

  - Online profiling information, e.g. "hot" methods

  - Whole-program information, e.g. which classes are loaded right now

  - Knowledge of target hardware architecture, e.g. cache line size

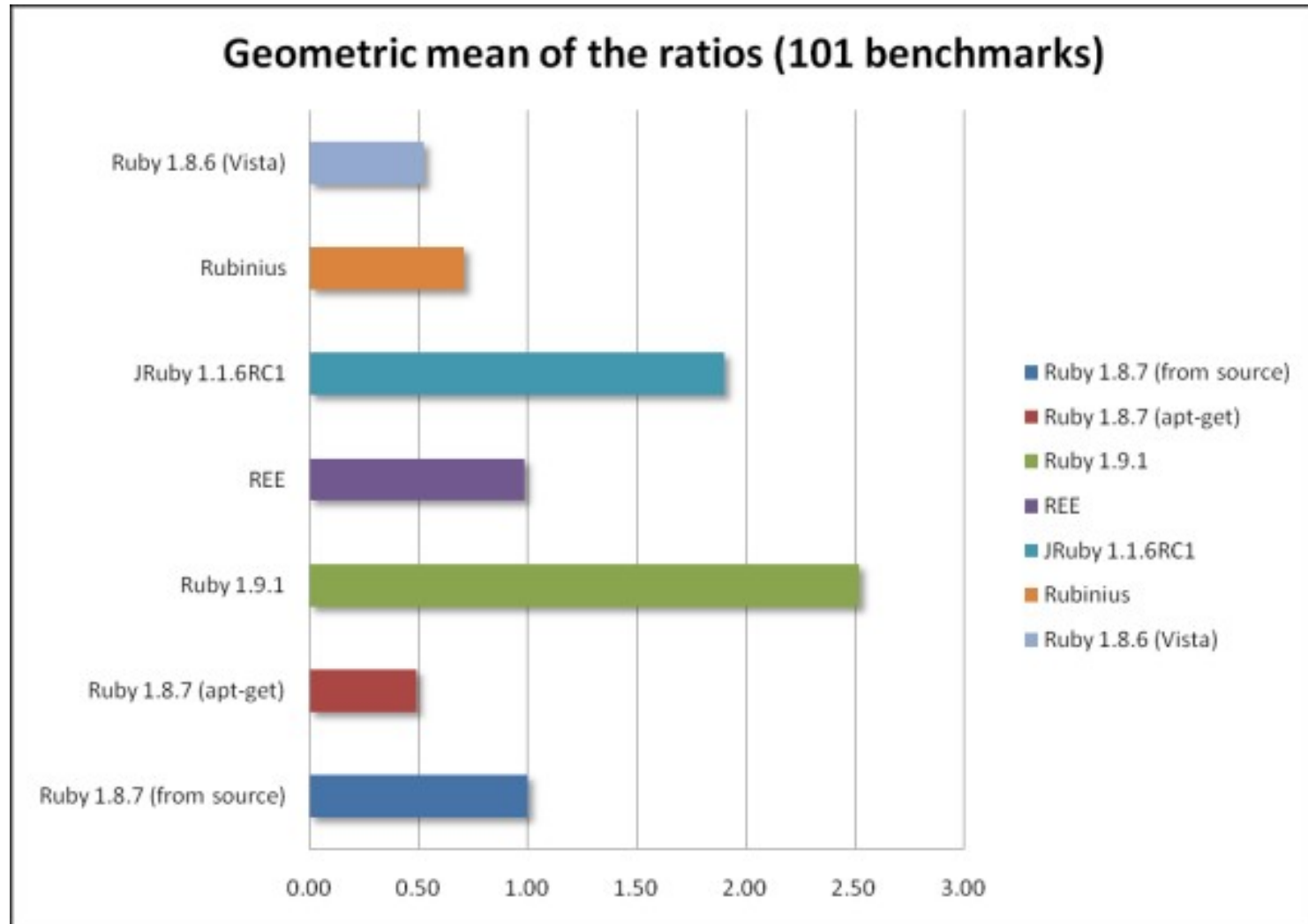
Can use adaptive and speculative techniques

- Compile optimistically, deoptimize when proven necessary

Targeting a VM allows compilers to generate "dumb" code

- The VM will optimize it better at runtime anyway

# The Great Ruby Shootout 2008



<http://antoniocangiano.com/2008/12/09/the-great-ruby-shootout-december-2008/>

## Lots of VMs out there

Java Virtual Machine (JVM)

.NET Common Language  
Runtime (CLR)

Smalltalk

Squeak

Perl

Parrot (Perl 6)

Python

YARV

Rubinius

Tamarin (ActionScript)

Valgrind (C++)

Lua

LLVM

TrueType

Dalvik

Flash

p-code (USCD Pascal)

Zend

Is there a universal VM in sight?

# VM Architecture

Lots of choices to make in designing a VM!

Where do programs store their data?

On a stack (like the JVM) or in registers (like real CPUs)?

Stack-based VMs abstract away details like register set sizes

Which data types should be supported natively?

Are there certain types that are “special”, like 32-bit signed integer?

How much indirection do you want in computing  $2+3$ ?

Is “everything an object”?

Instruction format and granularity

Assumptions about underlying machine word length and architecture

Imperative instruction set, or functional?

Exotic primitives, e.g. call/cc

Implementation flexibility: must tailcall v. may tailcall

# VM Architecture

Is there an object model?

Class-based or object-based?

Single inheritance or multiple inheritance or mixin composition?

Strongly typed or weakly typed?

Can type safety be assured statically?

How much do you trust compilers?

User-defined coercions?

How are errors handled?

Localized or centralized exceptions?

How much is rolled back or cleaned up after an error?

Provide additional primitives for language runtimes?

Weak references, foreign-function interface, reflection

Calling native code?

# JVM Architecture

## Stack-based program representation and execution

Representation is ~ post-order traversal of AST (easy to compile into)

JVM is responsible for efficient register allocation

## Core instructions

Stack and local variable management

Arithmetic, conversions, comparisons, logical operations

Object creation, array creation, exceptions, monitors

Method invocation, field access/assignment

## Data types: objects, arrays, eight primitive data types

## Object model: single inheritance with interfaces

## Dynamic linking

Untrusted code from the Web motivates static typechecking (at load-time)

Symbolic resolution (Base classes are not fragile in the JVM)

# JVM Architecture

Objects, signed integers, single inheritance, static typechecking?

Sounds a lot like the Java language!

But some of this is only skin-deep

200 compiler writers can't be wrong

# Languages on the JVM



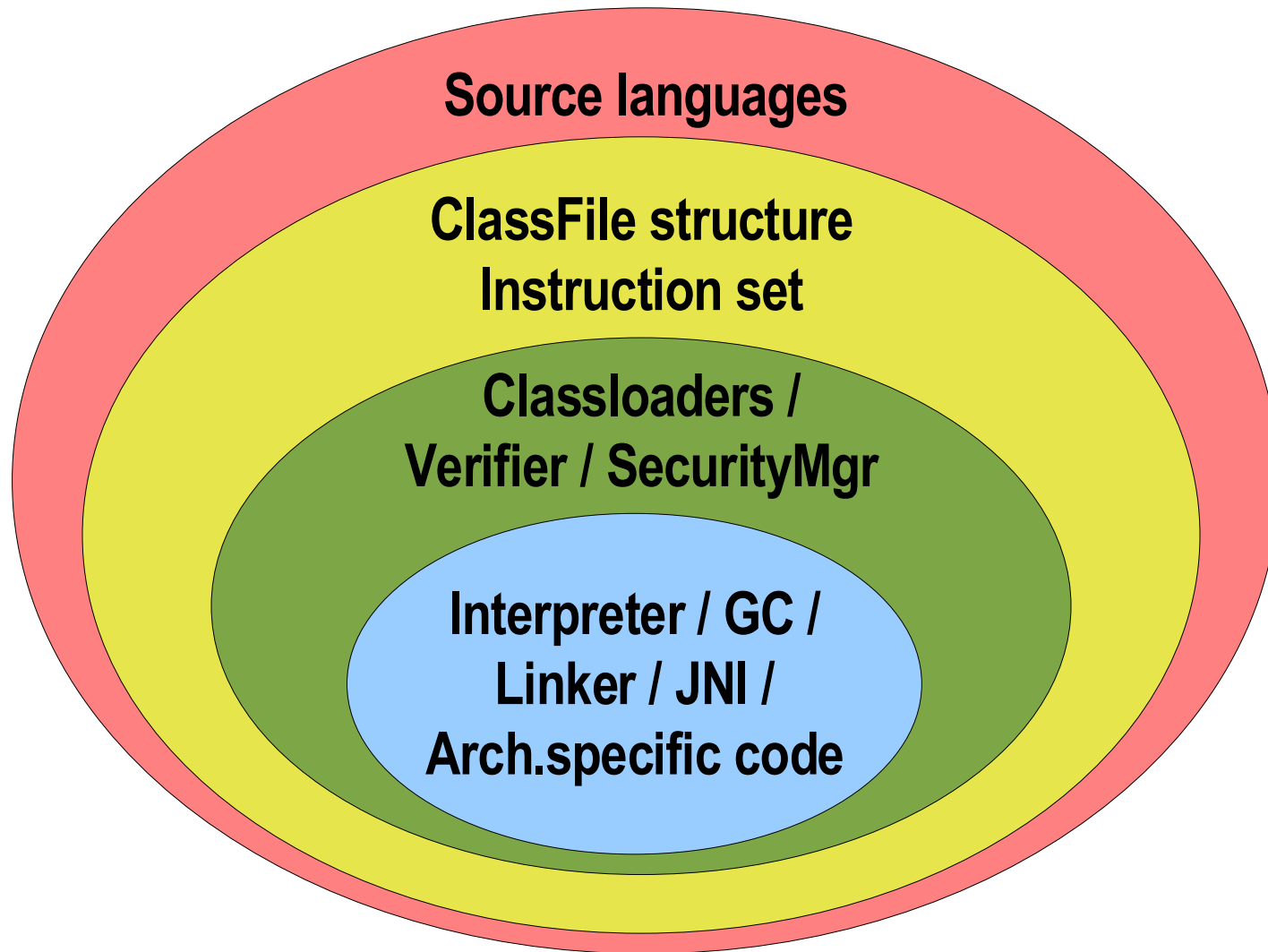
Languages on the JVM include:

- Bex Script
- WebL
- Funnel
- Sleep
- Zigzag
- Modula-2
- JESS
- Tea
- Jickle
- iScript
- Correlate
- Simkin
- Nice
- Lisp
- Icon
- Groovy
- Simkin
- Eiffel
- Prolog
- Mini
- Basic
- JudoScript
- Scala
- Rexx
- Tcl
- JavaFX Script
- v-language
- PLAN
- Pascal
- Luck
- Tiger
- Anvil
- Smalltalk
- Yassl
- Hojo
- Scala
- E
- Logo
- Tiger
- G
- Scheme
- JHCR
- JRuby
- Ada
- Processing
- Dawn
- Clojure
- Phobos
- Sather
- BeanShell
- LLP
- TermWare
- Pnuts
- C#
- Forth
- PHP
- Yoix
- SALSA
- Piccola
- ObjectScript
- Jython

# JVM Specification First Edition, 1997

- "The Java Virtual Machine knows nothing about the Java programming language, only of a particular binary format, the class file format."
- "A class file contains Java Virtual Machine instructions (or bytecodes) and a symbol table, as well as other ancillary information."
- "Any language with functionality that can be expressed in terms of a valid class file can be hosted by the Java virtual machine."
- "Attracted by a generally available, machine-independent platform, implementors of other languages are turning to the Java Virtual Machine as a delivery vehicle for their languages."
- "In the future, we will consider bounded extensions to the Java Virtual Machine to provide better support for other languages."

# Some things are more core than others



# Java VM vs Java Language

Java language  
fictions

Ruby language  
fictions

JVM  
features

<p>Primitive types+ops Object model Memory model Dynamic linking Access control GC Unicode</p>	<p><i>Checked exceptions</i> <i>Generics</i> <i>Enums</i> <i>Overloading</i> <i>Constructor chaining</i> <i>Program analysis</i> Primitive types+ops Object model Memory model Dynamic linking Access control GC Unicode</p>	<p><i>Open classes</i> <i>Dynamic typing</i> <i>'eval'</i> <i>Closures</i> <i>Mixins</i> <i>Regular expressions</i> <del>Primitive types+ops</del> Object model Memory model Dynamic linking <del>Access control</del> GC Unicode</p>
--	--	---

# Towards a universal VM

Easy to implement language fictions in a source compiler

Many are costless

- Checked exceptions in Java

- Traits and views in Scala

Some are very difficult to implement **efficiently**

- Open classes in Ruby

- Alternate numeric towers a la Scheme

There are lots of JVM modifications we could make

- But we should target the areas that are most costly in practice

- For dynamic languages, that is method selection and invocation

  - Calling a method is cheap; selecting the right method is expensive

  - Static languages do most of their method selection at compile-time

  - Dynamic languages do almost none at compile-time (obviously)

# JSR 292

Often called the "invokedynamic" JSR

Because it originally proposed a specific bytecode for method invocation  
Scope has widened since then

Work currently going on in JSR 292 includes

**invokedynamic** bytecode

Allow the language runtime to work hand-in-hand with the JVM on  
method selection

Method handles

Many languages have constructs like closures

Classes are too heavy a container for a single block of code

Interface injection

Add new methods and types to existing classes

# Virtual method invocation in Java

Take some simple source code:

```
String s = "Hello World";
System.out.println(s);
```

Let's look at the bytecode:

```
0:   ldc #2           //String "Hello World"
2:   astore 1
3:   getstatic #3     //Field
                        java/lang/System.out:
                        Ljava/io/PrintStream;
6:   aload 1
7:   invokevirtual #4 //Method
                        java/io/PrintStream.println:
                        (Ljava/lang/String;)V
```

A Java compiler chooses a version of `println` and embeds its **static type signature** into the bytecode  
 (Pop quiz: Local variables are untyped; why?)

# Virtual method invocation in Java

The only dynamism in method invocation is for the receiver

Different implementation of `size()` for `ArrayList` vs `LinkedList`

This is called single dispatch: Java's method selection algorithm does not (and cannot) consider the runtime types of arguments

Other languages may want multiple dispatch, which takes into account the dynamic types of the arguments

Given

```
invokevirtual Foo.bar:(int)int
```

The JVM looks for `bar:(int)int` in the class of the receiver

(The receiver is referenced from the stack)

If the receiver's class doesn't have that method, the JVM recurses up to its superclass...

# Virtual method invocation in Java

Repeated recursive method lookup makes invocation slow

Fortunately, this can often be heavily optimized

## Devirtualize monomorphic methods

If VM can prove there is only one target method body,  
then invocation turns into a single jump

Can then inline the method call, avoiding invocation overhead

Bigger basic block enables further optimizations

## Inline caching

Figure out the most likely receiver type for a call site, and cache it

JIT compiler generates code of the form

"If the receiver type is X, jump to Y, otherwise do a (slower) virtual  
dispatch"

Can generalize to cache multiple predicted receiver types

Optimizes for the most likely case(s)

# What about other languages?

Many features of the JVM are influenced by the Java language

- Single dispatch, statically typed method invocation

- Numeric types

- Single inheritance of implementation

It is fairly easy to work around these limitations...

- ...if you don't care about performance

- Dynamic calls can be implemented via reflection

- Multiple dispatch can be built with Visitor or by unrolling dispatch

- Other numeric types can be built with objects (like BigInteger)

- Multiple inheritance can be simulated with interfaces

All of this is great for getting to an implementation quickly

- But not necessarily a quick implementation

- Users are usually pretty happy at first, but then the hate mail starts

# Dynamically typed method invocation

Compiling dynamic languages directly to the JVM is tricky

```
function max(x, y) {  
    if x.lessThan(y) then y else x  
}
```

What do we compile the lessThan() call to?

```
invokevirtual lessThan: (unknownArgType) boolean
```

That's not going to work

- No receiver type

- No static argument type

- Maybe the return type isn't even boolean, maybe it's the type of y or x

# Dynamically typed method invocation

We would like to compile it as if it were the Java code:

```
boolean max(Object x, Object y) {  
    if (x.lessThan(y)) y; else x;  
}
```

But Object does not have a lessThan() method

Which would cause invokevirtual to fail

Need to pretend the Java code is

```
boolean max(Dynamic x, Dynamic y) {  
    if (x.lessThan(y)) y; else x;  
}
```

Where Dynamic is a special type

Proposed by Bill Joy in 1997

Defines every method imaginable

# Dynamically typed method invocation

Dynamic is a magic type

- A superclass of Object, so everything can be assigned to it

- A subclass of every class, so it can be assigned to everything

No such type in the JVM today

- Verification is complicated enough without a magic type

But if the JVM had Dynamic, invokeinterface is almost flexible enough to support it

- The verifier ignores invokeinterface!

- Why? Because subtyping of interfaces is complicated

- This code will verify:

```
interface Super {}  
interface Sub extends Super {}  
Super x = ...  
Sub y = x; // Assign supertype to subtype?!
```

# How can a language runtime manage dynamic invocation?

Creative solutions have been proposed

- Could define an interface for each possible method signature

  - Complex, fragile, expensive

- Could use reflection for everything

  - Use "inline caching" trick to cache Method objects for specific combinations of argument types

  - But...heavyweight and slow if you use it for every method call

  - Despite impressive improvements in reflective performance

It is easy to conclude

"the JVM isn't a match for dynamic languages"

## A little help goes a long way...

It turns out that the static type checking is closer to the surface than it first appears

Sun's JVM implementation is internally similar to Smalltalk

Smalltalk / Strongtalk is dynamic *and* fast

Need to fool the verifier to get past static type checking

The big need: first-class language-specific method resolution

So the language can identify the call target

But then **get out of the VM's way**

This is the rationale behind **invokedynamic**

VM up-calls into language runtime to resolve method

Language runtime decides whether or not it needs to stay in the loop

# invokedynamic

Consider this (hypothetical) instruction:

```
invokedynamic Object.lessThan: (Object) boolean
```

and suppose the receiver is an Integer and the argument is a Long

**We** know that invokedynamic should act as if it were:

```
invokevirtual Integer.lessThan: (Long) boolean
```

If the VM knew that, it could find the method **and inline it**

The language runtime can bridge the gap, if we ask it nicely

# Method selection in dynamic languages

A language runtime wants to take

```
invokedynamic Object.lessThan: (Object) boolean
```

and do what a Java compiler does with static types at overload resolution, only with dynamic types:

Inspect the dynamic type of the receiver

Inspect the dynamic type of the argument

Check which methods are available **now** in the receiver's dynamic type

Decide if a single argument is acceptable to those methods

Considering language rules on arity, optional parameters, default parameters...

Box values of primitive type to values of reference types

Box the argument list into an array and optimize it

eventually deciding to invoke

```
invokedynamic Integer.lessThan: (Long) boolean
```

# A language runtime wants to do all this...

# ONCE

(Until the receiver variable is assigned to a different object,  
or the receiver object's dynamic type is changed,  
or the argument object's dynamic types are changed)

# Bootstrap methods

The first time the JVM sees

```
invokedynamic Object.lessThan: (Object)boolean
```

it calls a bootstrap method which does all those things

Bootstrap chooses the ultimate method to be called

```
In this case, Integer.lessThan: (Long)boolean
```

VM associates that method with this invokedynamic instruction

"We're basically a direct participant in the JVM's method selection and linking process. So cool." - Charles O. Nutter

The next time the JVM sees

```
invokedynamic Object.lessThan: (Object)boolean
```

It jumps to the previously chosen method immediately

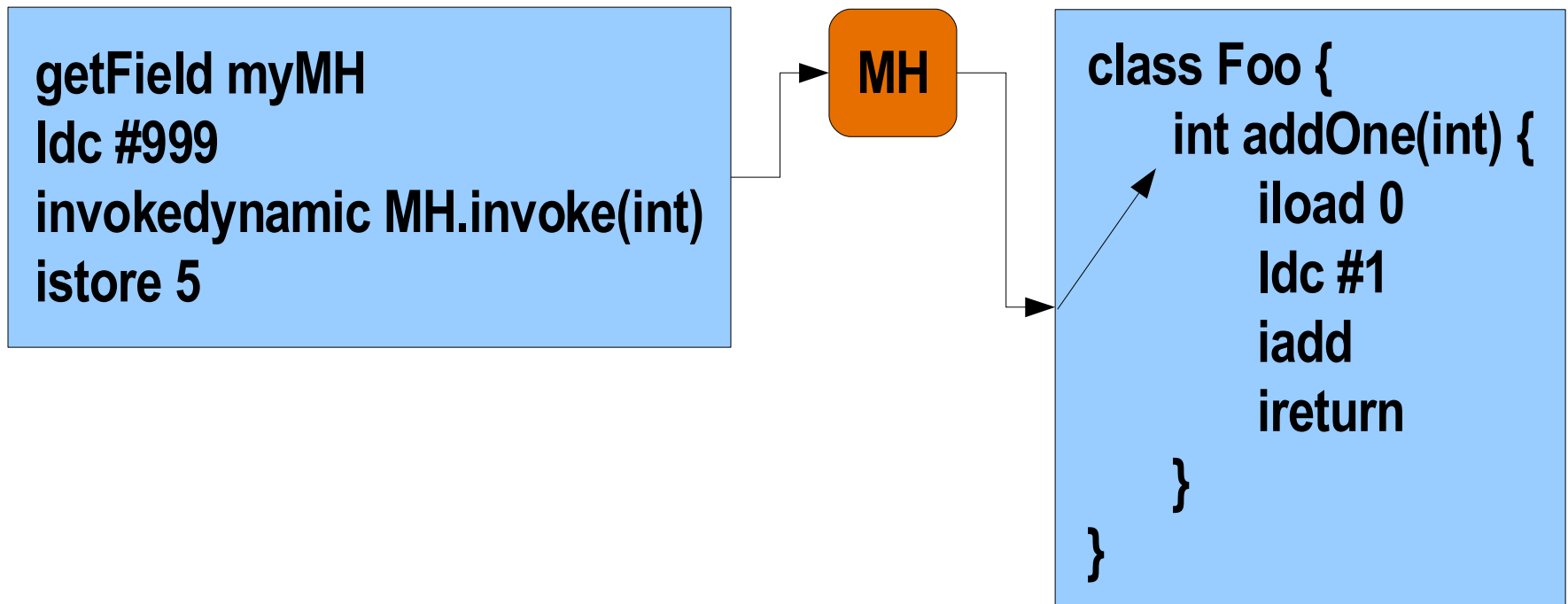
No language helper methods need be called

Hotspot's inliner can go to town

# Method handles

A method handle is an object-oriented function pointer

Represents the bootstrap method's desired target method



# Putting it all together

JVM method invocation is still statically typed

MethodHandle associated with

```
invokedynamic Object.lessThan: (Object)boolean
```

claims responsibility for the methods of exactly that (wide) type

The ultimate method invoked is arbitrary

Depends on the language's rules

Could even have a different name than in the instruction

May associate new method handles with a given call site frequently or infrequently

Depends on the language's rules and constructs

# Method handles are powerful

Consider this method again:

```
boolean max(Integer x, Long y) {
    if (x.lessThan(y)) y; else x;
}
```

Wouldn't it be great if we could partially evaluate it?

Calling `max(3)` would give a method equivalent to

```
boolean max_is_y_bigger_than_3(Long y) {
    if (3.lessThan(y)) y; else x;
}
```

Calling `max(1000)` would give a method equivalent to:

```
boolean max_is_y_bigger_than_1000(Long y) {
    if (1000.lessThan(y)) y; else x;
}
```

`max_is_y_bigger_than_1000` could contain optimizations specific to the integer value 1000 and the Long type of `y`

# Bound method handle

Represents the "method equivalent to ..."

~ A pointer to  $\text{max}(x,y)$  with  $x$  already bound to 3

Great for functional languages which expose "currying" in the type system

Crucial in an OO setting too

A language runtime treats the receiver object in source as just another parameter to the invocation

Most dispatch activity in the language runtime is concerned with inspecting that parameter

The receiver object at a call site doesn't change very often

Even if the methods of its class do, or the argument values

Reifying an invocation where the receiver object is fixed is worthwhile

Applicable methods are already known

A bound method handle is precisely that reification

We bind a call site to a method handle **and** a method handle to a receiver object

# Method handles are composable

An *adapter* method handle takes another method handle, and executes code before and after invoking it

Endless applications!

- Coercing types of individual arguments

  - `java.lang.String` -> `org.jruby.RubyString` (different encoding)

- Boxing all arguments into a single array

- Pre-allocating stack frames

- Prepare thread-specific context

"Bit by bit, piece by piece, the complex vagaries of our call protocols can be decomposed into functions, referenced by method handles, and composed into fast, efficient, direct calls"

- Charles O. Nutter

# Interface injection

Dynamically typed programs look like self-modifying code

A class definition or method body can be different each time you look

Generally, self-modifying code is dangerous and hard to optimize

Class redefinition raises crazy new issues for VM engineering

Idea: Don't restructure classes, just relabel them

Interface injection: The ability to modify old classes just enough for them to implement new interfaces

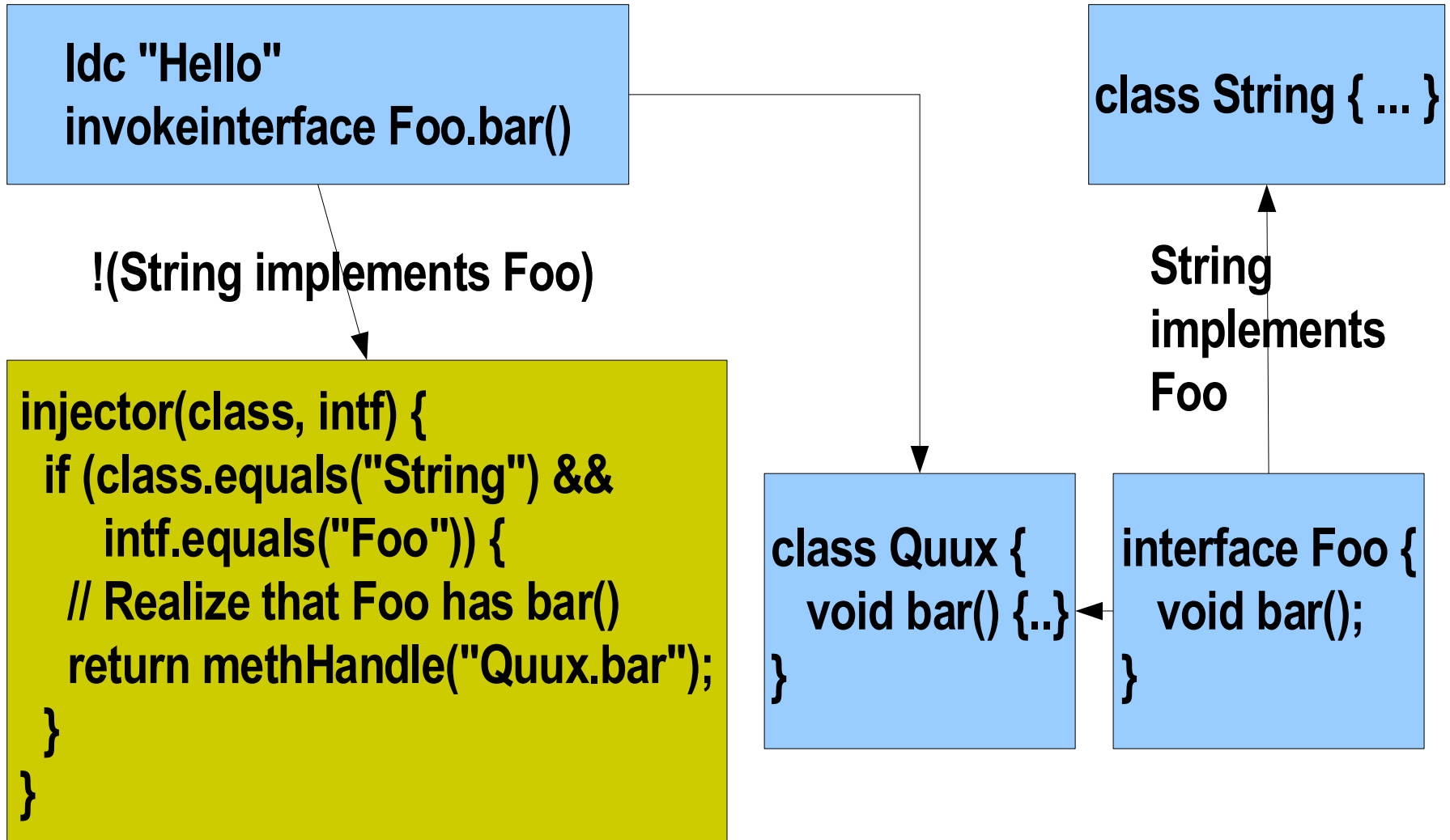
Superinterfaces are cheap for JVM objects

invokeinterface is fast these days

If an interface-dependent operation is about to fail, call a static injector method to bind an interface to the object and provide MethodHandles for the interface's methods

One chance only for the injector to say yes!

# Injection in action



## Example: Dynamic dispatch in Groovy

A Java object has a Class where behavior is defined

A Java compiler demands a class provide behavior

```
invokevirtual java.lang.String.length: ()int
```

A Groovy object has a Class, but it just gives its MetaClass

A Groovy compiler does not demand a class provide behavior

All member access is sent to a MetaClass object

A method call compiles to

```
MetaClass.invokeMethod(recvObj, "methName", args)
```

invokeMethod does groovy things to choose a method

A GroovyObject has a MetaClass injected at construction

# Example: Dynamic dispatch in Groovy

Groovy maintains the fiction that Java strings have extra methods

```
Object asType(Class) // implements the 'as' operator
Object eachLine(Closure)
List tokenize(String)
```

Call to tokenize cannot compile to

```
invokevirtual java.lang.String.tokenize:(String)List
```

Groovy runtime must call tokenize by:

- Getting the receiver's class (java.lang.String)

- Performing a table lookup to map it to a MetaClass object

- Calling invokeMethod(s, "tokenize", ..) on the MetaClass object

The middle step is the JIT's nemesis - can't inline a table lookup!

# Interface injection to the rescue

Interface injection removes the need for a table lookup

Every object in a Groovy program becomes a GroovyObject

Even system types like java.lang.String

Then, the Groovy compiler's standard invocation works:

```
(GroovyObject) s.getMetaClass()  
                .invokeMethod(s, "tokenize", ..)
```

The first time s is cast to GroovyObject, interface injection occurs

Injection of the GroovyObject interface would including calling Groovy's own "injector" to set the metaclass of s

Thereupon, getMetaClass and invokeMethod calls can be inlined  
s.tokenize(..) is as fast as built-in s.length()

# Other fun stuff

Tail calls

Continuations

Tuples

Value objects

# Resources

John Rose (JSR 292 spec lead)

<http://blogs.sun.com/jrose/>

Charles O. Nutter (JRuby lead)

<http://blog.headius.com/>

Multi-Language Virtual Machine OpenJDK project

<http://openjdk.java.net/projects/mlvm/>

JVM Language Summit, September 2008

<http://openjdk.java.net/projects/mlvm/jvmlangsummit/>

"JVM Languages" Google Group

<http://groups.google.com/group/jvm-languages/>

