TOWARDS A UNIVERSAL VIRTUAL MACHINE

Alex Buckley
Specification Lead,
Java Language and VM
Sun Microsystems

John Rose
Lead Engineer,
Multi-Language Virtual Machine
Sun Microsystems

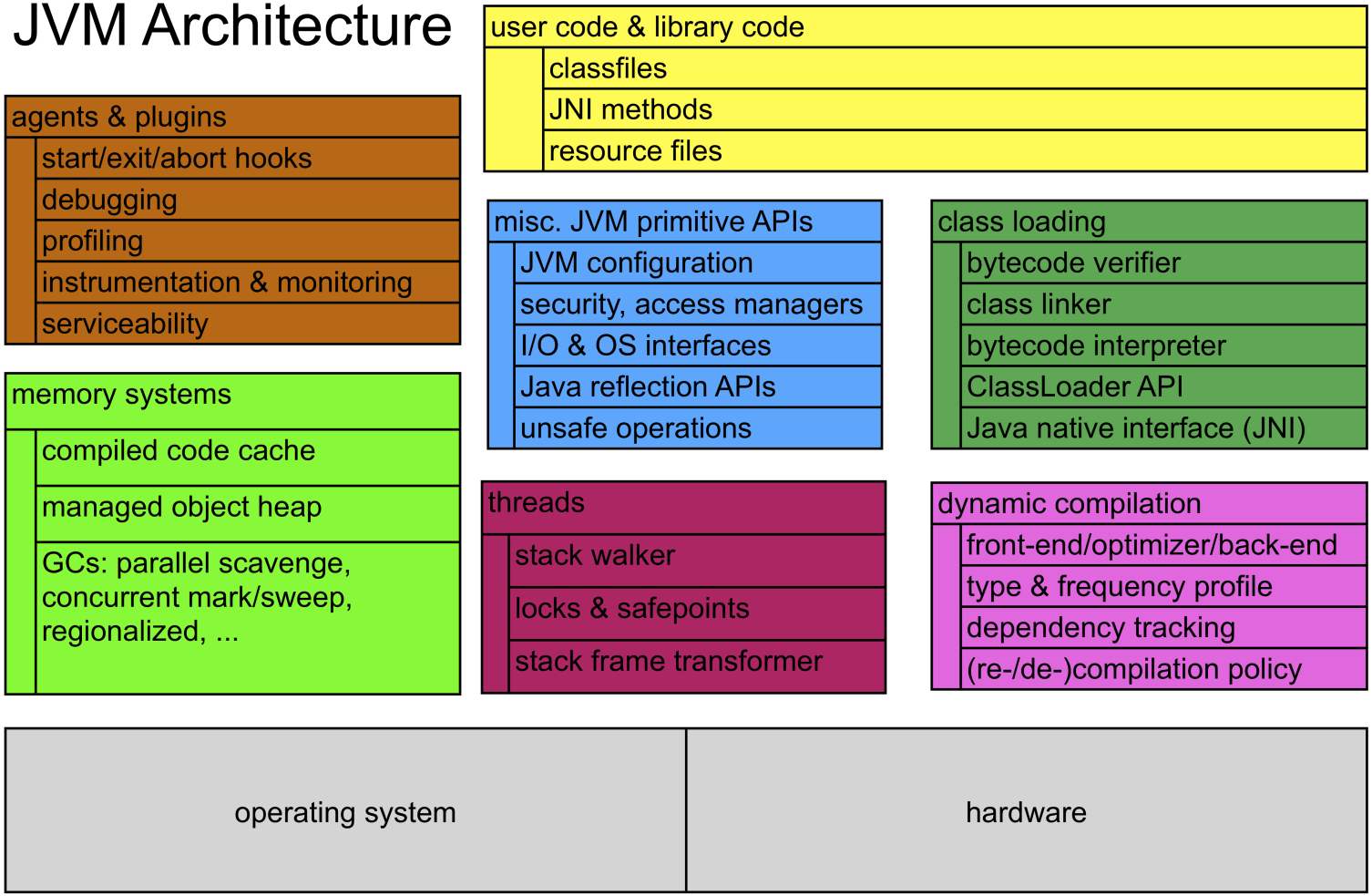
Overview

- The Java Virtual Machine (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

What is a virtual machine?

- A software implementation of a computer architecture
- Some virtual machines simulate a whole machine
  - VMWare, VirtualBox, VirtualPC, Parallels
- Some virtual machines host a single application
  - Java Virtual Machine, .NET Common Language Runtime
  - Usually implement a custom instruction set chosen to suit general applications
  - Application is isolated from the Operating System and from applications in other virtual machines

JVM Architecture



“Java is slow because it runs on a VM”

- Early implementations of the JVM executed bytecode with an interpreter [slow]

“Java is fast because it runs on a VM”

- Major breakthrough was the advent of “Just In Time” compilers [fast]
  - Compile from bytecode to machine code at runtime
  - Optimize using information *available at runtime only*
- Simplifies static compilers
  - javac and ecj generate “dumb” bytecode and trust the JVM to optimize
  - Optimization is real, even if it is invisible

Common JVM optimizations

- compiler tactics
  - delayed compilation
  - tiered compilation
  - on-stack replacement
  - delayed reoptimization
  - program dependence graph representation
  - static single assignment representation
- proof-based techniques
  - exact type inference
  - memory value inference
  - memory value tracking
  - constant folding
  - reassociation
  - operator strength reduction
  - null check elimination
  - type test strength reduction
  - type test elimination
  - algebraic simplification
  - common subexpression elimination
  - integer range typing
- flow-sensitive rewrites
  - conditional constant propagation
  - dominating test detection
  - flow-carried type narrowing
  - dead code elimination
- language-specific techniques
  - class hierarchy analysis
  - devirtualization
  - symbolic constant propagation
  - autobox elimination
  - escape analysis
  - lock elision
  - lock fusion
  - de-reflection
- speculative (profile-based) techniques
  - optimistic nullness assertions
  - optimistic type assertions
  - optimistic type strengthening
  - optimistic array length strengthening
  - untaken branch pruning
  - optimistic N-morphic inlining
  - branch frequency prediction
  - call frequency prediction
- memory and placement transformation
  - expression hoisting
  - expression sinking
  - redundant store elimination
  - adjacent store fusion
  - card-mark elimination
  - merge-point splitting
- loop transformations
  - loop unrolling
  - loop peeling
  - safepoint elimination
  - iteration range splitting
  - range check elimination
  - loop vectorization
- global code shaping
  - inlining (graph integration)
  - global code motion
  - heat-based code layout
  - switch balancing
  - throw inlining
- control flow graph transformation
  - local code scheduling
  - local code bundling
  - delay slot filling
  - graph-coloring register allocation
  - linear scan register allocation
  - live range splitting
  - copy coalescing
  - constant splitting
  - copy removal
  - address mode matching
  - instruction peepholing
  - DFA-based code generator

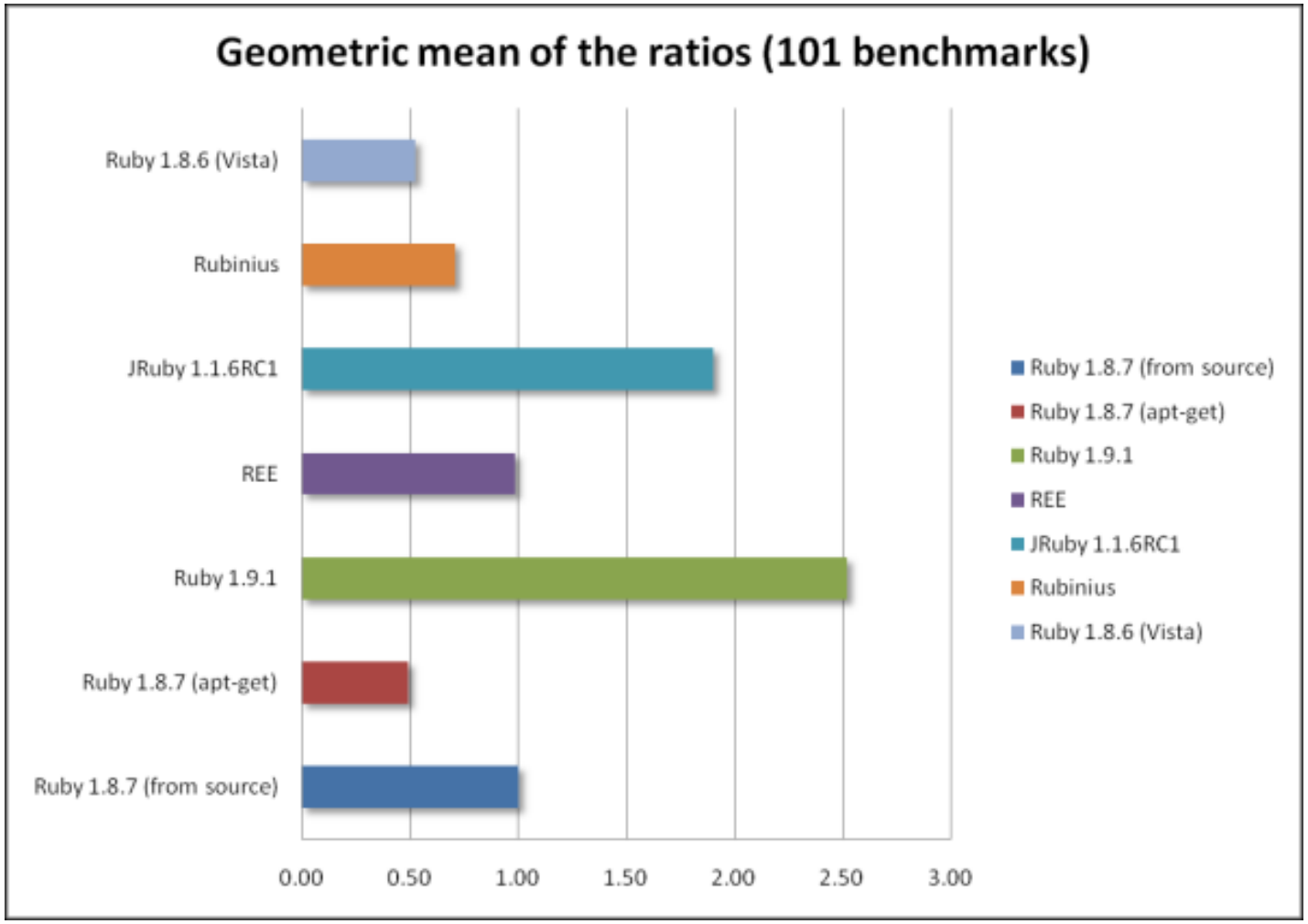
Inlining is the Über-Optimization

- Speeding up method calls is the big win
- For a given method call, try to predict which method should be called
- Numerous techniques available
  - Devirtualization (prove there's *one* target method)
  - Monomorphic inline caching
  - Profile-driven inline caching
- Goal is *inlining*: copying method body into the caller
  - Gives more code for the optimizer to chew on

Optimizations are universal

- Optimizations work on bytecode in .class files
- A compiler for any language – not just Java – can produce a .class file
- *All* languages can benefit from dynamic compilation and optimizations like inlining

The Great Ruby Shootout 2008



2.00 means "twice as fast"

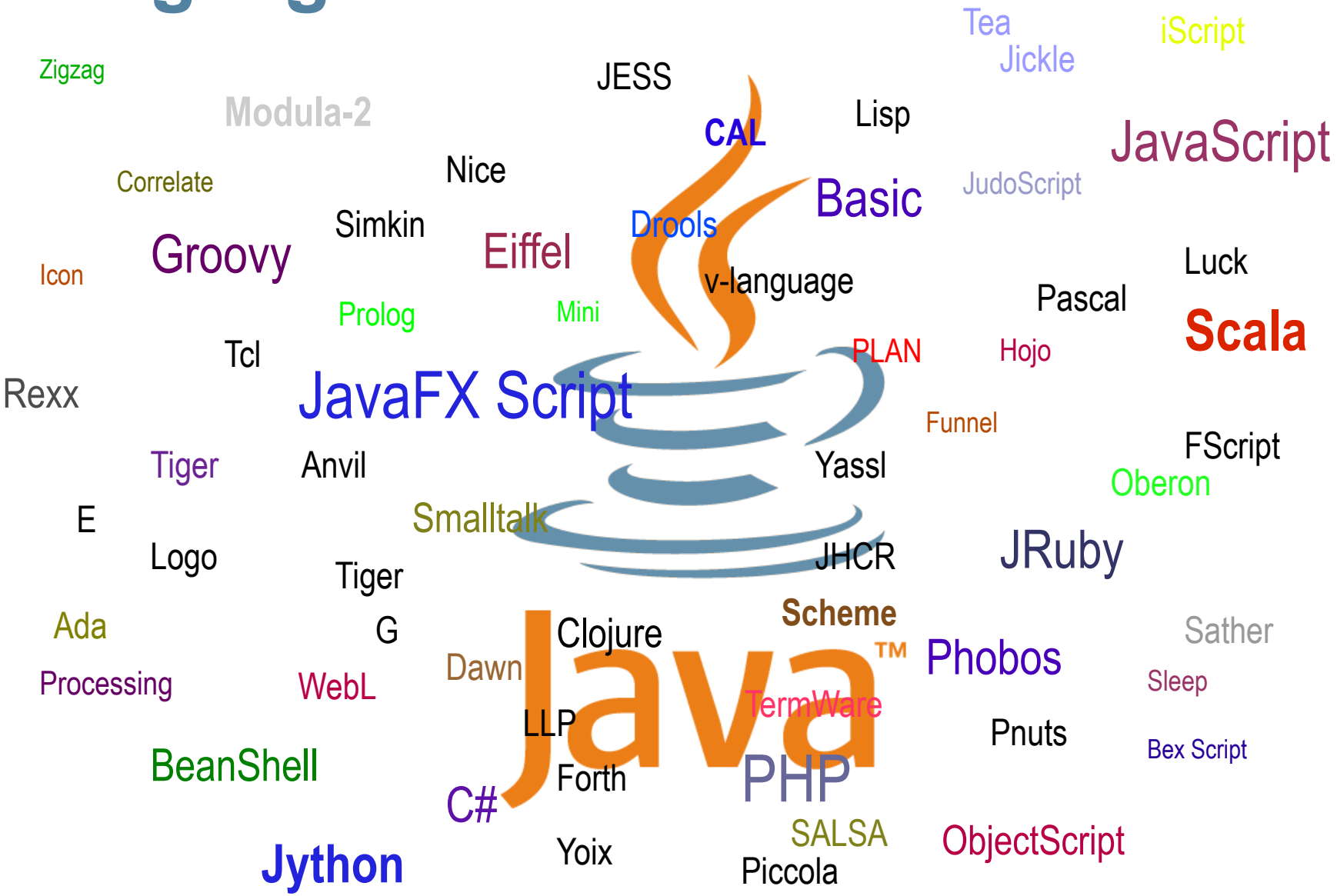
0.50 means "half the speed"

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

Languages ♥ Virtual Machines

- Programming languages need runtime support
  - Memory management / Garbage collection
  - Concurrency control
  - Security
  - Reflection / Debugging / Profiling
  - Standard libraries (collections, database, XML)
- Traditionally, language implementers coded these features themselves
- Today, many implementers choose to target a virtual machine to reuse its infrastructure

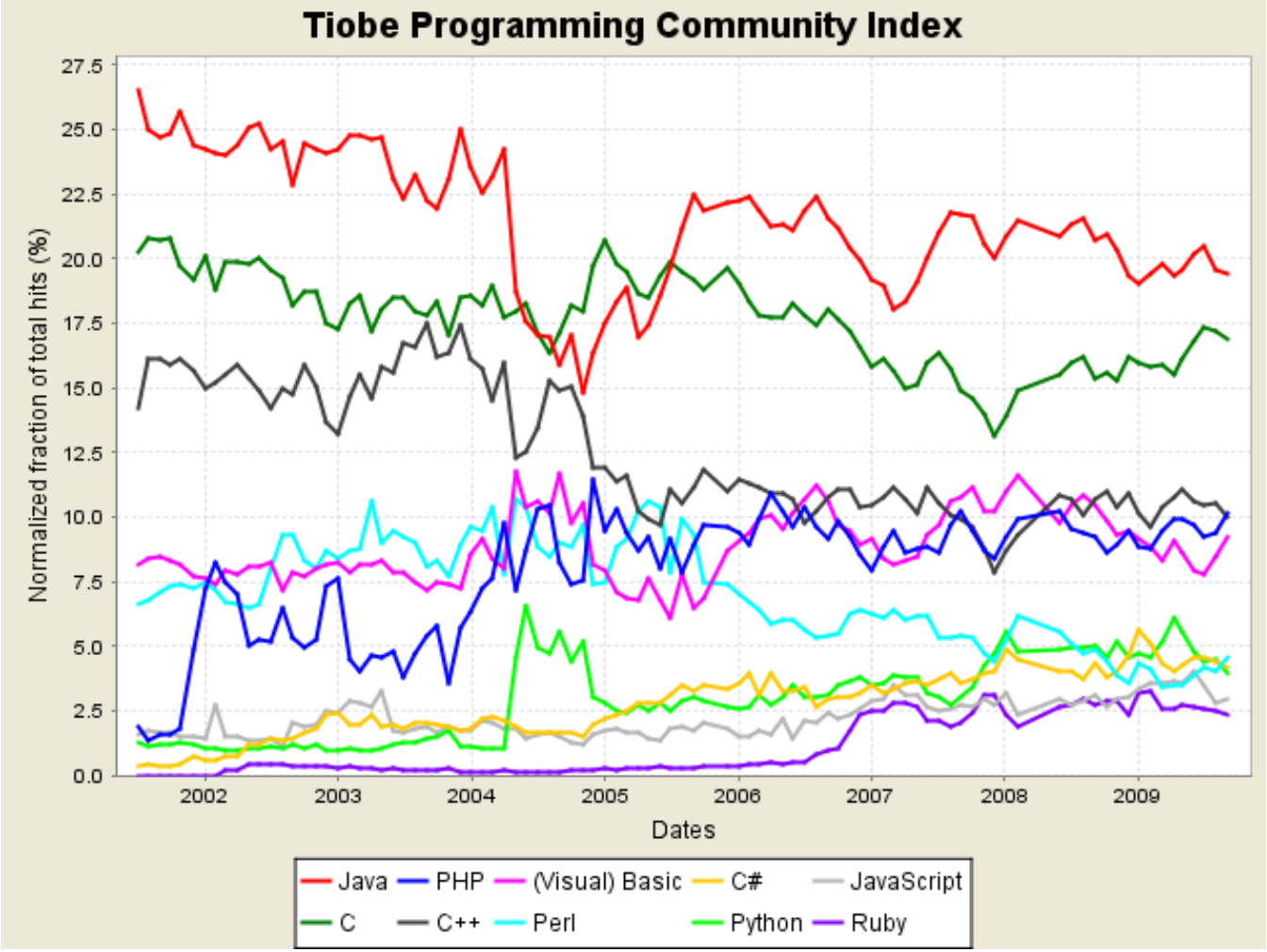
Languages ♥ The JVM



Benefits for the developer

- Choice
  - Use the right tool for the right job, while sharing infrastructure
  - Unit tests in Scala, business logic in Java, web app in JRuby, config scripts in Jython...
  - …with the same IDE, same debugger, same JVM
- Extensibility
  - Extend a Java application with a Groovy plugin
- Manageability
  - Run RubyOnRails with JRuby on a managed JVM

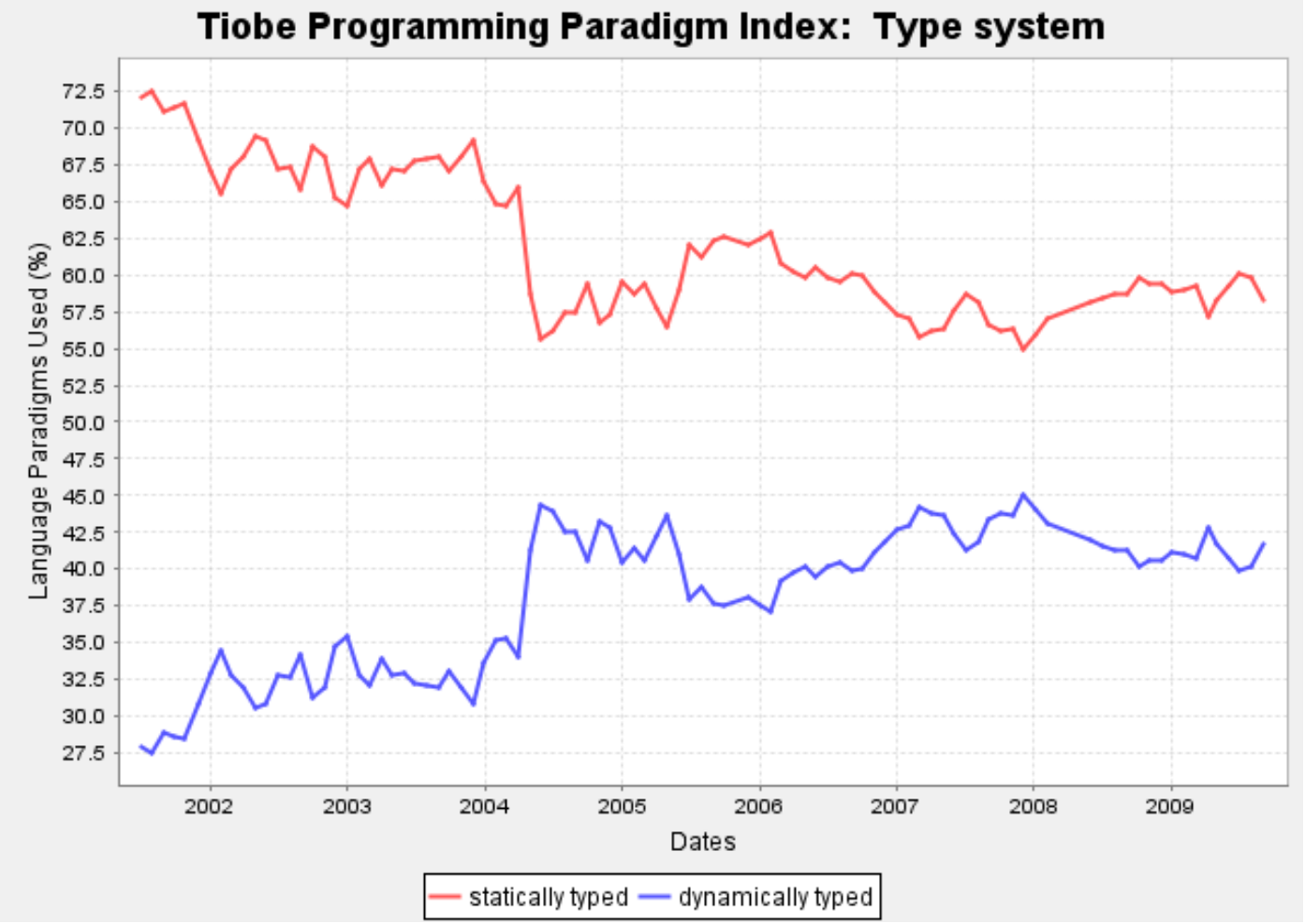
Trends in programming languages



Data courtesy of TIOBE: www.tiobe.com

Is the JVM a Universal VM?

Different *kinds* of languages



Data courtesy of TIOBE: www.tiobe.com

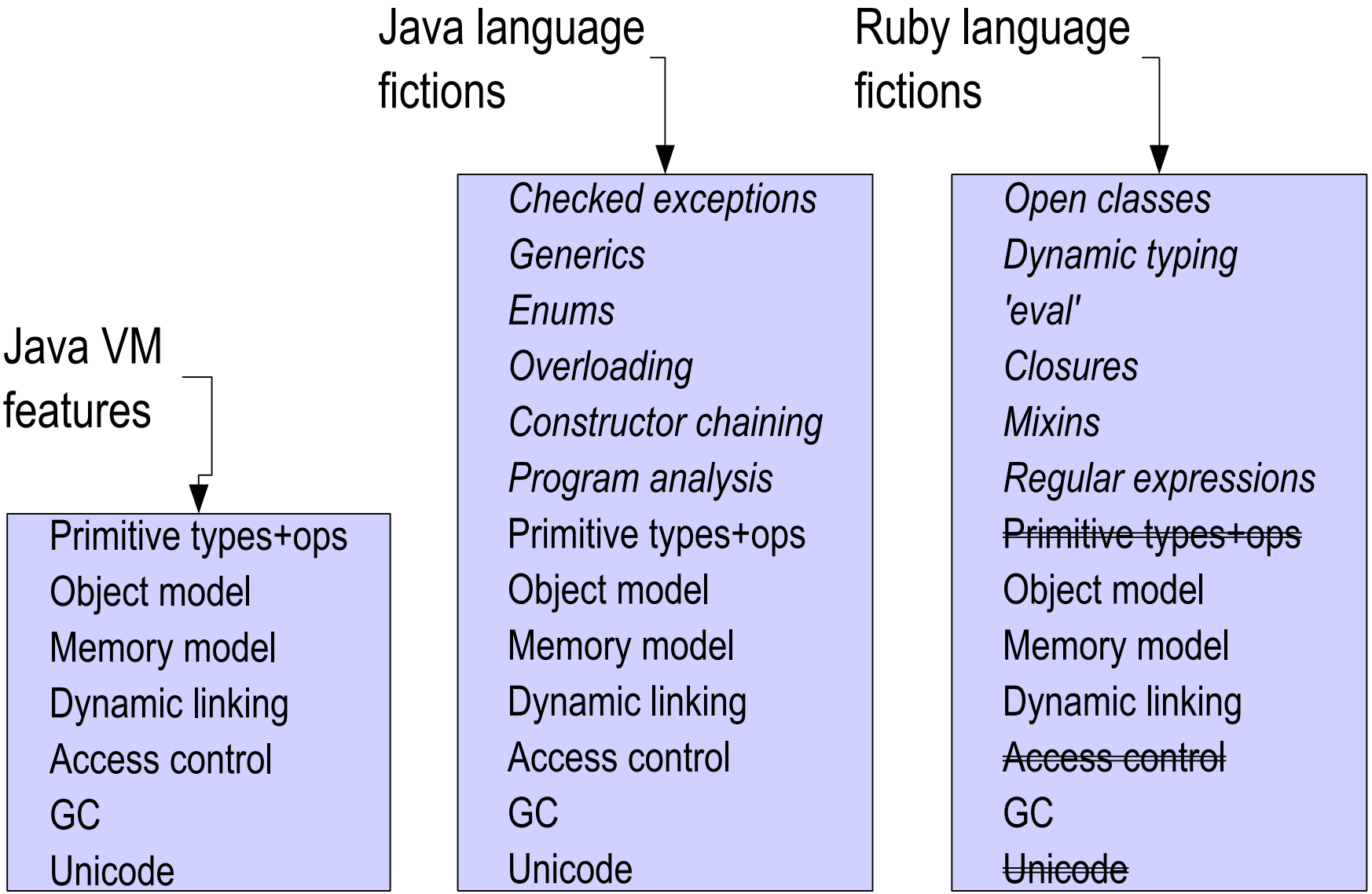
Fibonacci in Java and Ruby

```
int fib(int n) {
  if (n<2)
    return n;
  else
    return fib(n-1)+fib(n-2);
}
```

```
def fib(n) {
  if n<2
    n
  else
    fib(n-1)+fib(n-2)
  end
}
```

Not as similar as they look

- Data types
  - Not just char/int/long/double and java.lang.Object
- Collections
  - Not just java.util.*
- Method call
  - Not just Java-style overloading and overriding
- Control structures
  - Not just 'for', 'while', and 'break'


Java VM features
Primitive types+ops
Object model
Memory model
Dynamic linking
Access control
GC
Unicode
Java language fictions
Checked exceptions
Generics
Enums
Overloading
Constructor chaining
Program analysis
Primitive types+ops
Object model
Memory model
Dynamic linking
Access control
GC
Unicode
Ruby language fictions
Open classes
Dynamic typing
'eval'
Closures
Mixins
Regular expressions
Primitive types+ops
Object model
Memory model
Dynamic linking
Access control
GC
Unicode

Towards a Universal VM

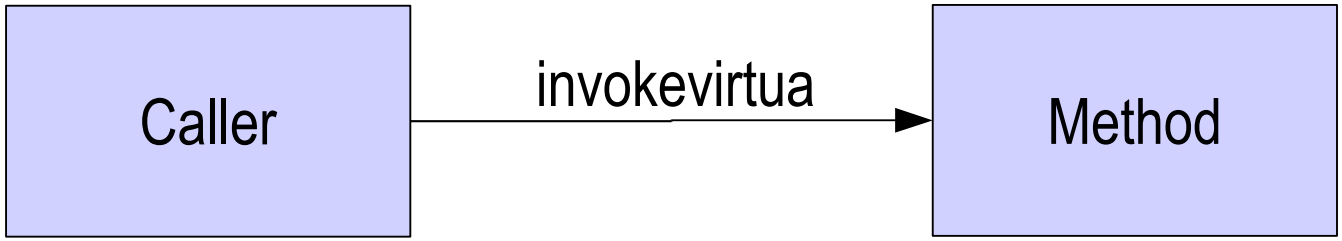
- Simulating language features at runtime is slow
- When multiple languages target a VM, common issues quickly become apparent
  - JVM Language Summits in 2008 and 2009
- With expertise and taste, the JVM's features can grow to benefit *all* languages
  - Adding a little more gains us a lot!
  - Each additional “stretch” helps many more languages

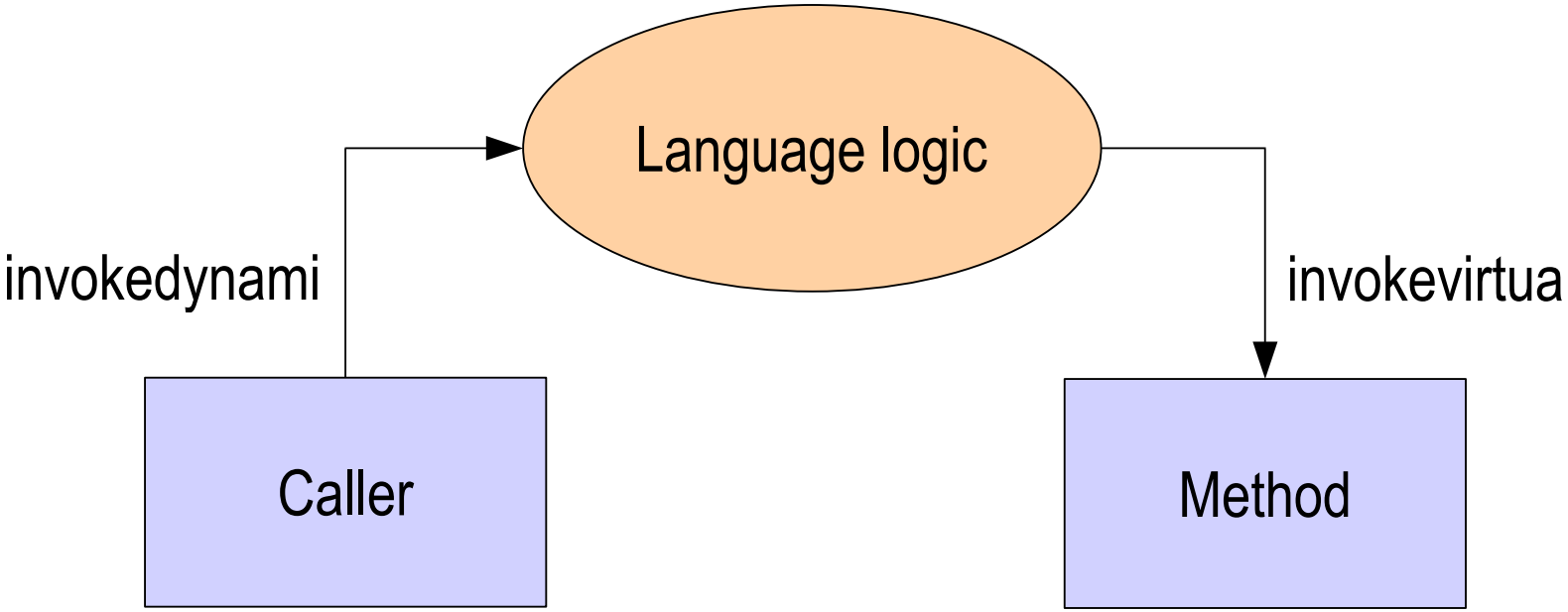
If we could make one change to the JVM to improve life for dynamic languages, what would it be?

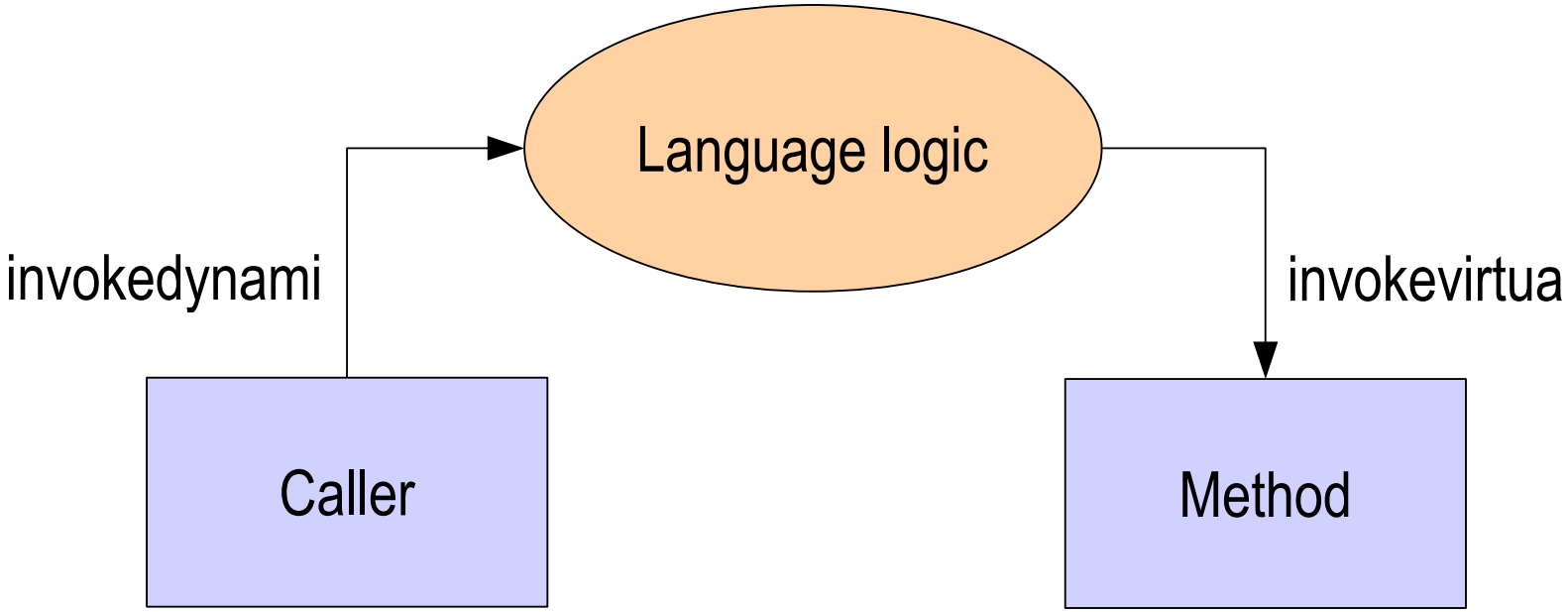
More flexible method calls

More flexible method calls

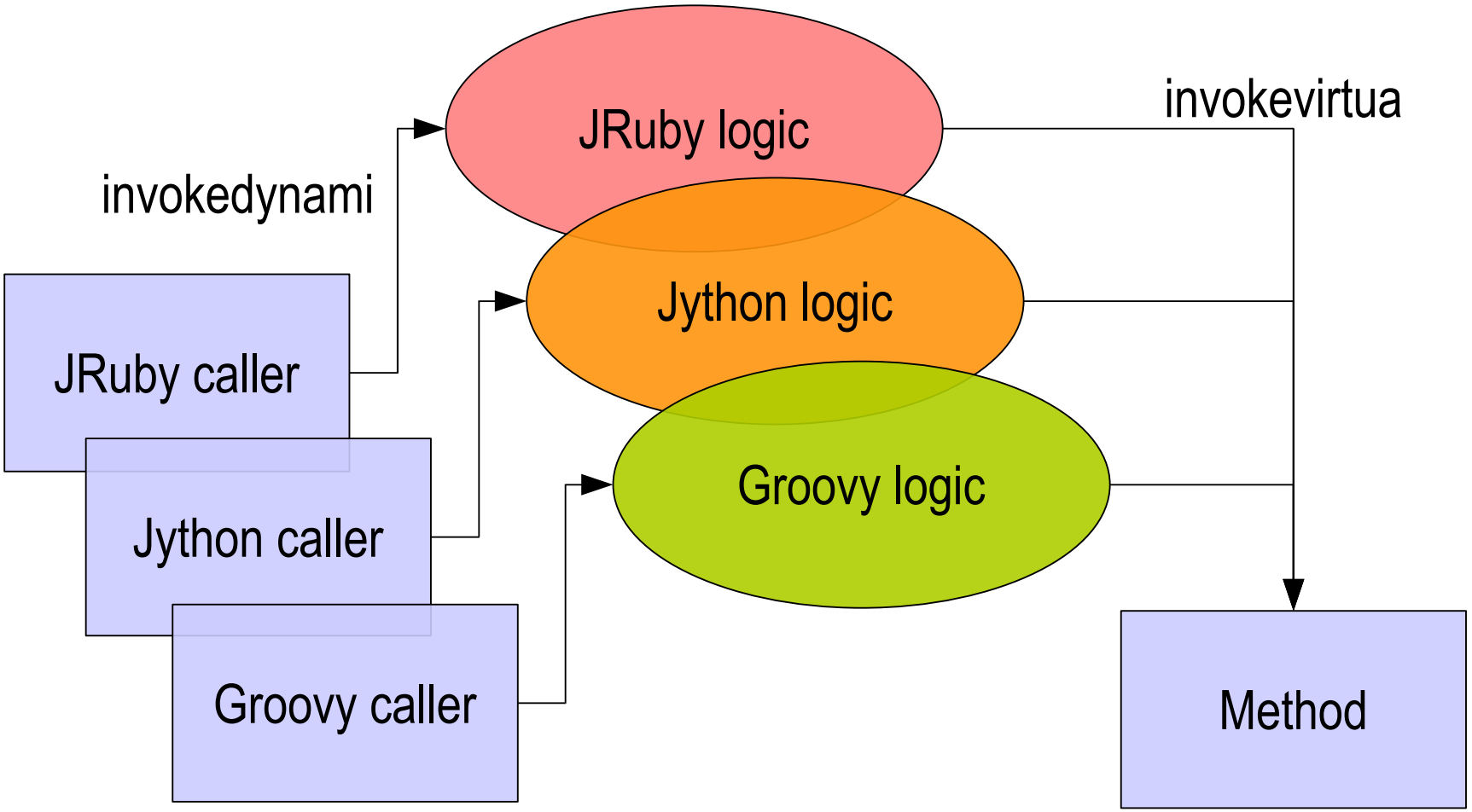
- The *invokevirtual* bytecode performs a method call
- Its behavior is Java-like and fixed
- Other languages need custom behavior
- Idea: Let some “language logic” determine the behavior of a JVM method call
- Invention: the *invokedynamic* bytecode
  - VM asks some “language logic” how to call a method
  - Language logic decides if it needs to stay in the loop


Caller
invokevirtua
Method


Language logic
invokedynami
invokevirtua
Caller
Method



- Check which methods are available *now* in each class [open classes]
- Check the dynamic types of arguments to the method [multimethods]
- Rearrange and inject arguments [optional and default parameters]
- Convert numbers to a different representation [fixnums]


JRuby logic
invokevirtua
invokedynami
Jython logic
JRuby caller
Groovy logic
Jython caller
Groovy caller
Method

Language logic is only needed...

ONCE

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

Bootstrap methods

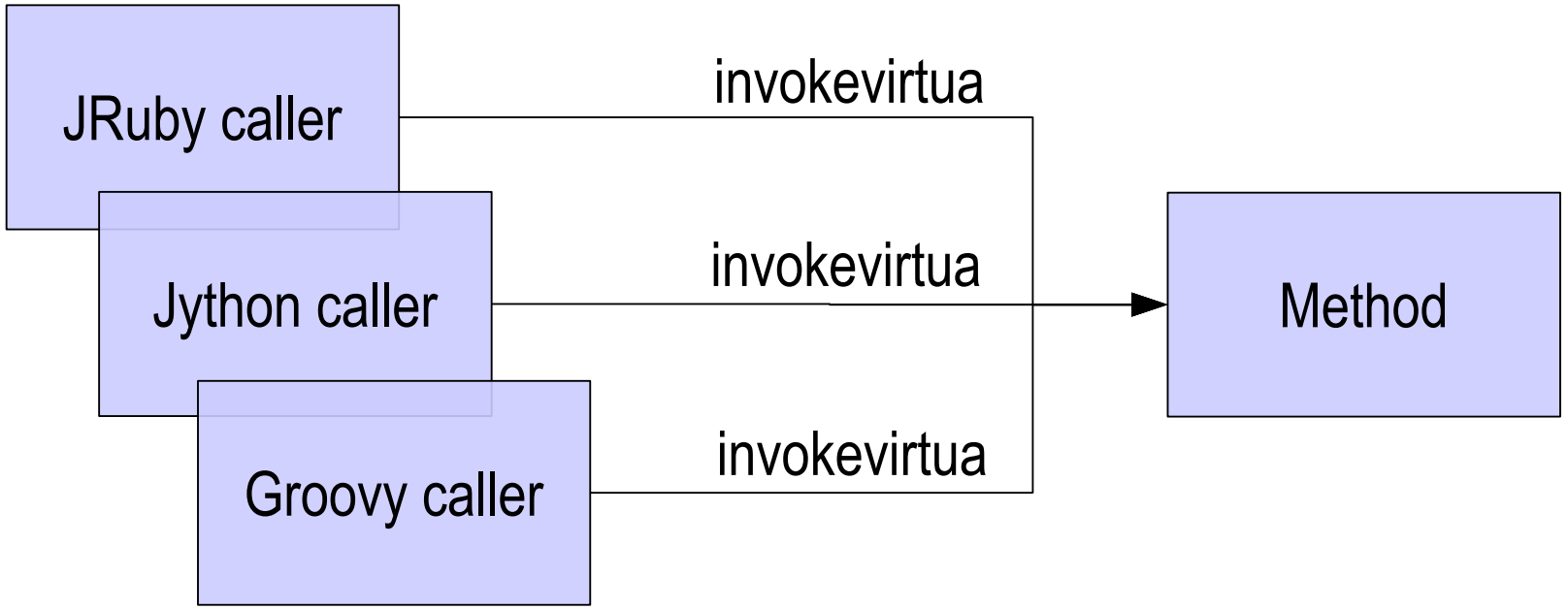
- **The first time the JVM executes**
  invokedynamic Object.lessThan(Object)boolean

  It consults a *bootstrap method* in the language logic to discover which method should be called
- Suppose the answer is “Integer.compare(Long)BOOL”

  The JVM associates that method with the invokedynamic instruction
- **The next time the JVM executes**
  invokedynamic Object.lessThan(Object)boolean

  It jumps to the previously chosen method immediately
- No language logic is involved
- Now the JVM knows the target method, it can start inlining!

It's as if invokedynamic never existed...



We're basically a direct participant in the JVM's method selection and linking process. So cool.

Charles O. Nutter, JRuby lead

JVM Specification, 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.

A budget of invokes

invokestatic	invokespecial	invokevirtual	invokeinterface	invokedynamic
no receiver	receiver class	receiver class	receiver interface	no receiver
no dispatch	no dispatch	single dispatch	single dispatch	custom dispatch
B8 ***nn nn***	**B7** ***nn nn***	**B6** ***nn nn***	**B9** ***nn nn aa*** **00**	**BA** ***nn nn*** **00 00**

Optimizing invokedynamic

- Don't want it to be slow for the first 10 years
- Most of the current optimization framework applies!
- Every invokedynamic links to *one* target method so inlining is possible
- Complex language logic can also be inlined
- A JVM implementation can even assist with *speculative* optimizations that are language-specific

JSR 292

- invokedynamic
- Interface injection
- Tail calls
- Continuations
- Hotswap

A Universal VM is in sight

- Where all languages run faster
  - Less overhead for their implementers
  - Direct application of existing JVM optimizations
  - Exploit the *centuries* of programmer effort invested in JVM implementations to make their bytecodes *zoom*!
- Where all languages are first-class citizens
  - Larger community working to improve the JVM
  - Virtuous circle of ideas and prototypes

What about Java (the language) ?

- Java depends on static types for method calls
- Code in dynamic languages has no static types
- How can Java code access libraries written in dynamic languages without losing type safety?
  - "Punch a hole" in the Java type system?
  - All access through privileged libraries?
  - Many details to consider, such as checked exceptions
- We are moving cautiously
  - Java Language Rule #0: "First, do no harm"

Innovation never stops

- CMT machines are here, but the software isn't
- Cheap CPU cycles make scripting practical
- New languages and APIs will continue to appear
- The JVM will continue to play a central role

Resources

- John Rose (JSR 292 spec lead)
  - http://blogs.sun.com/jrose
- Multi-Language Virtual Machine OpenJDK project
  - http://openjdk.java.net/projects/mlvm
- JVM Language Summit, September 2009
  - http://www.jvmlangsummit.com
- “JVM Languages” Google Group
  - http://groups.google.com/group/jvm-languages

THANK YOU

Alex Buckley
alex.buckley@sun.com

John Rose
john.rose@sun.com