ORACLE®

One VM, Many Languages

Brian Goetz
Java Language Architect, Oracle Corporation

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Overview

The Java Virtual Machine (JVM) has, in large part, been the engine behind the success of the Java programming language

- The JVM is undergoing a transformation: to become a Universal VM
- In years to come, it will power the success of other languages too

ORACLE

“Java is slow because it runs on a VM”

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

ORACLE

“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, but invisible

Monday, October 4, 2010

Optimizations are universal

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

ORACLE

HotSpot 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 uber-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 only *one* target method)
  - Monomorphic inline caching
  - Profile-driven inline caching
- Goal is *inlining*: copying called method's body into caller
  - Gives more code for the optimizer to chew on

Inlining: Example

```
public interface FooHolder<T> {
    public T getFoo();
}

public class MyHolder<T> implements FooHolder<T> {
    private final T foo;

    public MyHolder(T foo) { this.foo = foo; }

    public T getFoo() { return foo; }
}
```

Inlining: Example

```
public interface FooHolder<T> {
    public T getFoo();
}

public class MyHolder<T> implements FooHolder<T> {
    private final T foo;

    public MyHolder(T foo) { this.foo = foo; }

    public T getFoo() { return foo; }
}
...
public String getString(FooHolder<String> holder) {
    if (holder == null)
        throw new NullPointerException("You dummy.");
    else
        return holder.getFoo();
}
```

Inlining: Example

```
public interface FooHolder<T> {
    public T getFoo();
}

public class MyHolder<T> implements FooHolder<T> {
    private final T foo;

    public MyHolder(T foo) { this.foo = foo; }

    public T getFoo() { return foo; }
}
...
public String getString(FooHolder<String> holder) {
    if (holder == null)
        throw new NullPointerException("You dummy.");
    else
        return holder.getFoo();
}
...
public String foo(String x) {
    FooHolder<String> myFooHolder = new MyHolder<String>(x);
    return getString(myFooHolder);
}
```

Inlining: Example

Step 1

Inline getString()

```
public interface FooHolder<T> {
    public T getFoo();
}

public class MyHolder<T> implements FooHolder<T> {
    private final T foo;

    public MyHolder(T foo) { this.foo = foo; }

    public T getFoo() { return foo; }
}
...
public String getString(FooHolder<String> holder) {
    if (holder == null)
        throw new NullPointerException("You dummy.");
    else
        return holder.getFoo();
}
...
public String foo(String x) {
    FooHolder<String> myFooHolder = new MyHolder<String>(x);
    return getString(myFooHolder);
}
```

Inlining: Example

```
public interface FooHolder<T> {
    public T getFoo();
}

public class MyHolder<T> implements FooHolder<T> {
    private final T foo;

    public MyHolder(T foo) { this.foo = foo; }

    public T getFoo() { return foo; }
}
...
public String foo(String x) {
    FooHolder<String> myFooHolder = new MyHolder<String>(x);
    if (myFooHolder == null)
        throw new NullPointerException("You dummy.");
    else
        return myFooHolder.getFoo();
}
```

Inlining: Example

Step 2

Dead code

```
public interface FooHolder<T> {
    public T getFoo();
}

public class MyHolder<T> implements FooHolder<T> {
    private final T foo;

    public MyHolder(T foo) { this.foo = foo; }

    public T getFoo() { return foo; }
}
...
public String foo(String x) {
    FooHolder<String> myFooHolder = new MyHolder<String>(x);
    if (myFooHolder == null)
        throw new NullPointerException("You dummy.");
    else
        return myFooHolder.getFoo();
}
```

Inlining: Example

```
public interface FooHolder<T> {
    public T getFoo();
}

public class MyHolder<T> implements FooHolder<T> {
    private final T foo;

    public MyHolder(T foo) { this.foo = foo; }

    public T getFoo() { return foo; }
}
...
public String foo(String x) {
    FooHolder<String> myFooHolder = new MyHolder<String>(x);
    if (myFooHolder == null)
        throw new NullPointerException("You dummy.");
    else
        return myFooHolder.getFoo();
}
```

Inlining: Example

Step 3

Type sharpen and inlining

```
public interface FooHolder<T> {
    public T getFoo();
}

public class MyHolder<T> implements FooHolder<T> {
    private final T foo;

    public MyHolder(T foo) { this.foo = foo; }

    public T getFoo() { return foo; }
}
...
public String foo(String x) {
    FooHolder<String> myFooHolder = new MyHolder<String>(x);
    return myFooHolder.getFoo();
}
```

Inlining: Example

```
public interface FooHolder<T> {
    public T getFoo();
}

public class MyHolder<T> implements FooHolder<T> {
    private final T foo;

    public MyHolder(T foo) { this.foo = foo; }

    public T getFoo() { return foo; }
}
...
public String foo(String x) {
    FooHolder<String> myFooHolder = new MyHolder<String>(x);
    return myFooHolder.foo;
}
```

Inlining: Example

Step 4

Escape analysis

```
public interface FooHolder<T> {
    public T getFoo();
}

public class MyHolder<T> implements FooHolder<T> {
    private final T foo;

    public MyHolder(T foo) { this.foo = foo; }

    public T getFoo() { return foo; }
}
...
public String foo(String x) {
    FooHolder<String> myFooHolder = new MyHolder<String>(x);
    return myFooHolder.foo;
}
```

Inlining: Example

```
public interface FooHolder<T> {
    public T getFoo();
}

public class MyHolder<T> implements FooHolder<T> {
    private final T foo;

    public MyHolder(T foo) { this.foo = foo; }

    public T getFoo() { return foo; }
}
...
public String foo(String x) {
    return x;
}
```

Inlining is the uber-optimization

- Each time we inlined, we exposed information from the outer scope
- Which could be used to optimize the inner scope further, now that there is more information available
- Code often gets smaller and faster at the same time
- HotSpot works hard to inline everything it can
- Will apply “inline caching” when it can't predict inlining perfectly
- Will inline speculatively based on current loaded class hierarchy

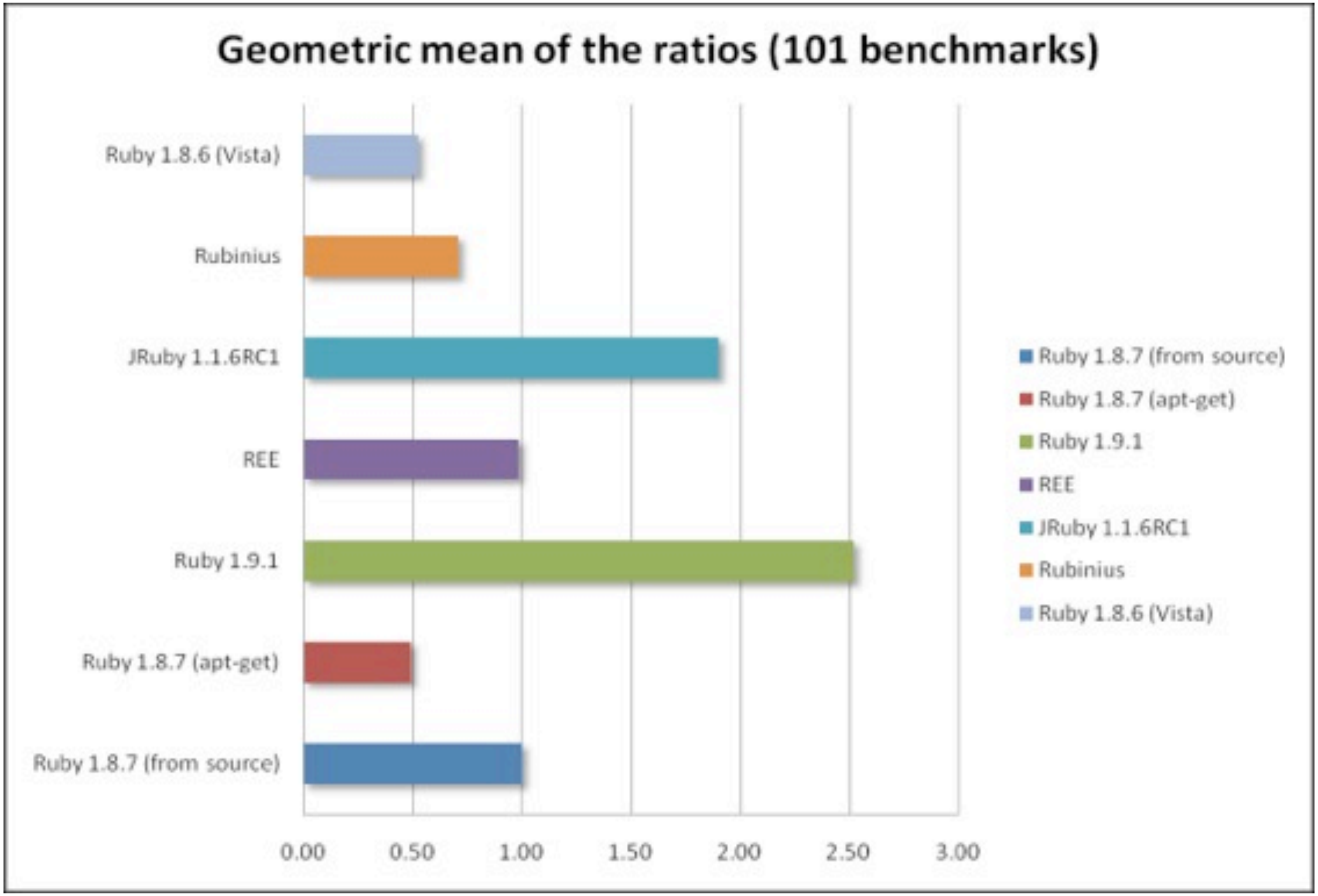
ORACLE

Languages ♥ Virtual Machines

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

ORACLE

The Great Ruby Shootout 2008



2.00 means "twice as fast"

0.50 means "half the speed"

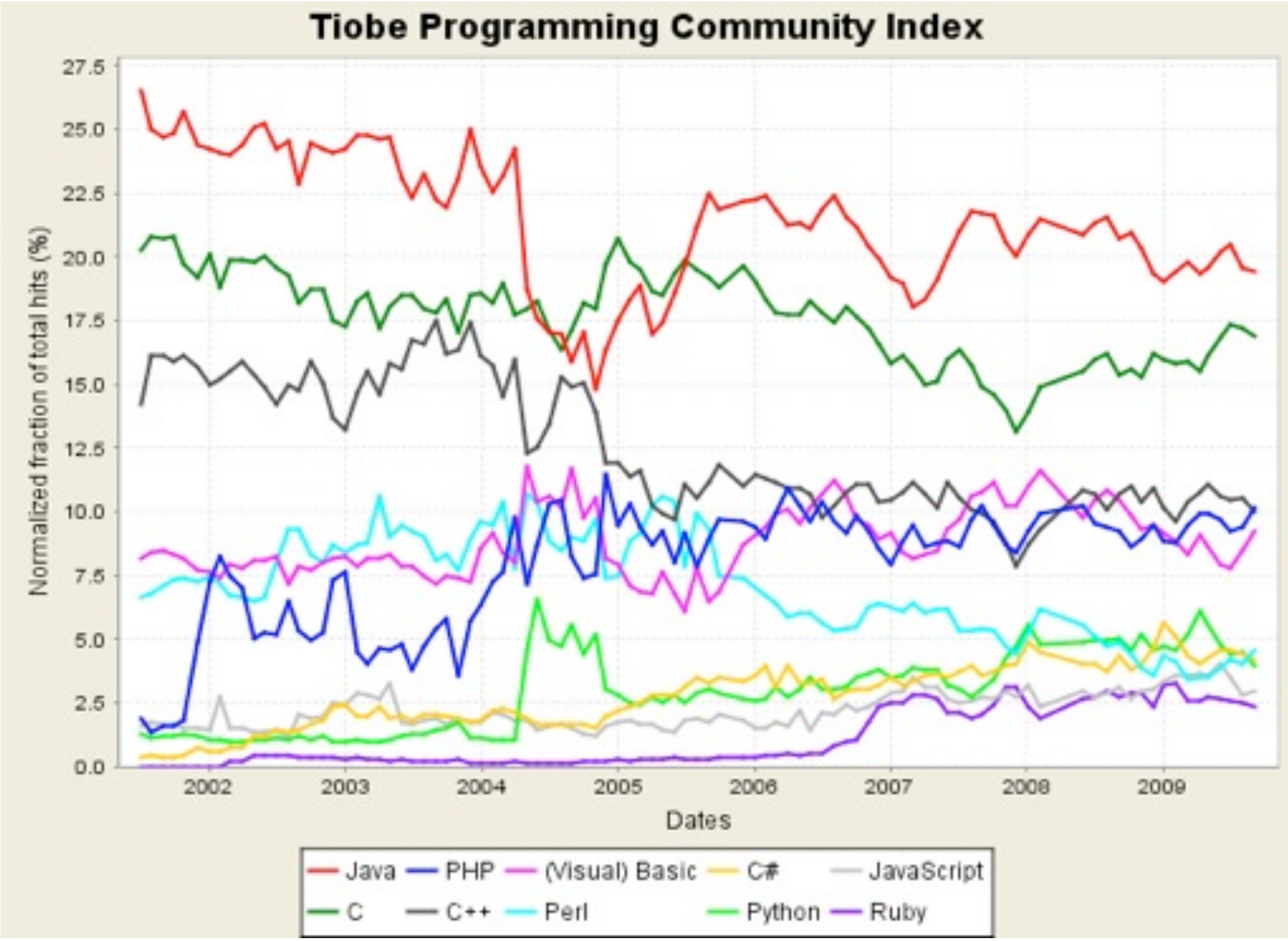
http://antoniocangiano.com/2008/12/09/the-great-ruby-shootout-

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

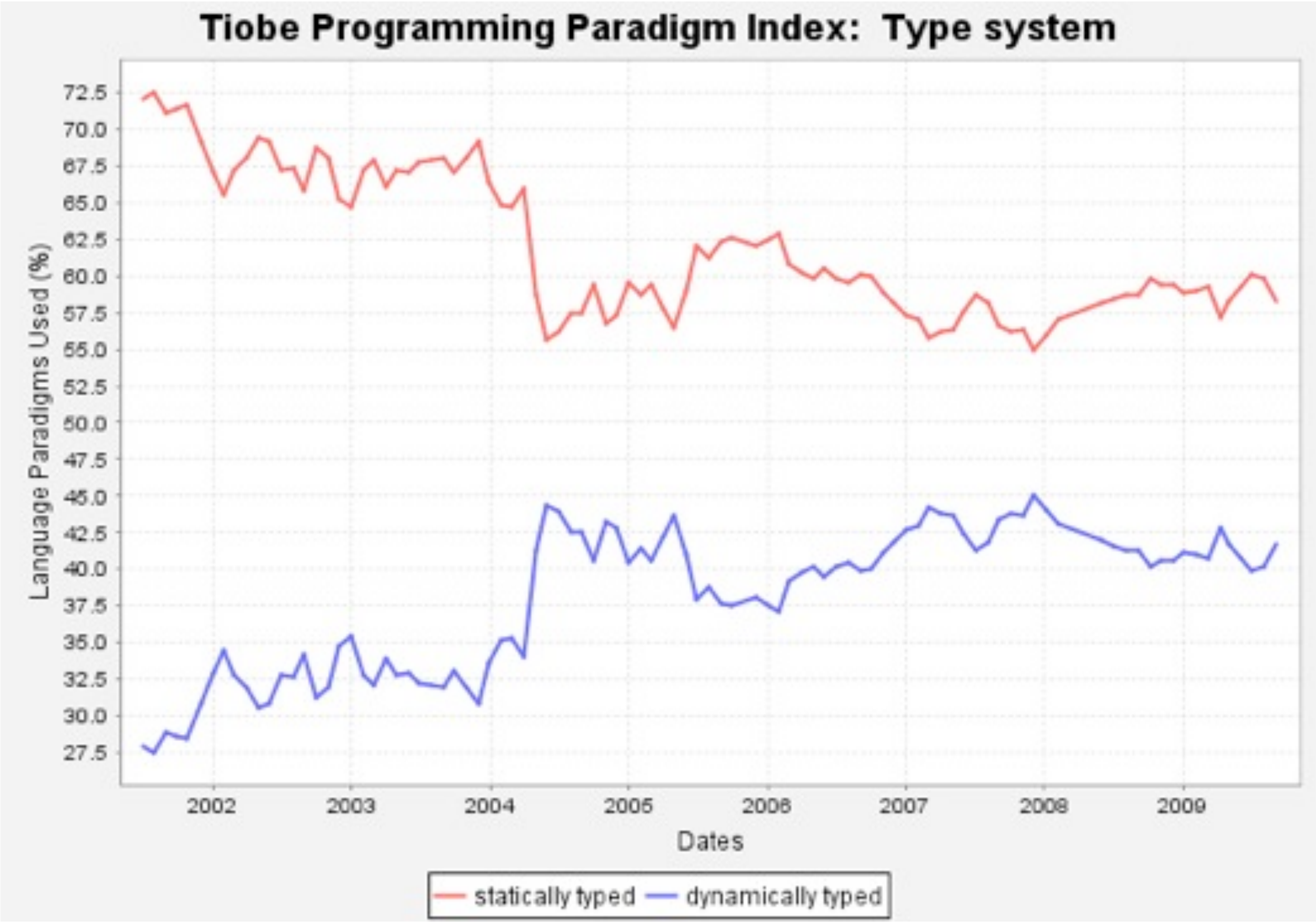
ORACLE

Trends in programming languages



Monday, October 4, 2010

Different kinds of languages



ORACLE

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
}
```

ORACLE

Not as similar as they look

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

ORACLE

Reality is a simulation

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
- With expertise and taste, the JVM can grow to benefit *all* languages
  - Adding a little more gains us a lot!
  - Each additional “stretch” helps many more languages

Java VM 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.*

JVM extensions for other languages

JVM extensions for other languages

- There’s no shortage of JVM feature suggestions

JVM extensions for other languages

- There's no shortage of JVM feature suggestions
  – Dynamic method linkage (non-Java method lookup)

JVM extensions for other languages

- There’s no shortage of JVM feature suggestions
  - Dynamic method linkage (non-Java method lookup)
  - Tail calls (more dynamic control flow)

JVM extensions for other languages

- There's no shortage of JVM feature suggestions
  - Dynamic method linkage (non-Java method lookup)
  - Tail calls (more dynamic control flow)
  - Continuations (fibers vs. threads, mobile vs. bound, …)

ORACLE

JVM extensions for other languages

- There's no shortage of JVM feature suggestions
  - Dynamic method linkage (non-Java method lookup)
  - Tail calls (more dynamic control flow)
  - Continuations (fibers vs. threads, mobile vs. bound, …)
  - Tuples (a.k.a. value types, structs)

ORACLE

JVM extensions for other languages

- There's no shortage of JVM feature suggestions
  - Dynamic method linkage (non-Java method lookup)
  - Tail calls (more dynamic control flow)
  - Continuations (fibers vs. threads, mobile vs. bound, …)
  - Tuples (a.k.a. value types, structs)
  - Open classes (e.g., for "monkey patching")

ORACLE

JVM extensions for other languages

- There's no shortage of JVM feature suggestions
  - Dynamic method linkage (non-Java method lookup)
  - Tail calls (more dynamic control flow)
  - Continuations (fibers vs. threads, mobile vs. bound, …)
  - Tuples (a.k.a. value types, structs)
  - Open classes (e.g., for "monkey patching")
  - Interface injection (making new views of old types)

JVM extensions for other languages

- There’s no shortage of JVM feature suggestions
  - Dynamic method linkage (non-Java method lookup)
  - Tail calls (more dynamic control flow)
  - Continuations (fibers vs. threads, mobile vs. bound, …)
  - Tuples (a.k.a. value types, structs)
  - Open classes (e.g., for “monkey patching”)
  - Interface injection (making new views of old types)
  - Tagged fixnums (autoboxing without tears)

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

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

More flexible method calls

- The invokevirtual bytecode performs a method call

More flexible method calls

- The invokevirtual bytecode performs a method call
- Its behavior is Java-like and fixed

More flexible method calls

- The invokevirtual bytecode performs a method call
- Its behavior is Java-like and fixed
- Other languages need custom behavior

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

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 gives an answer, and decides if it needs to stay in the loop

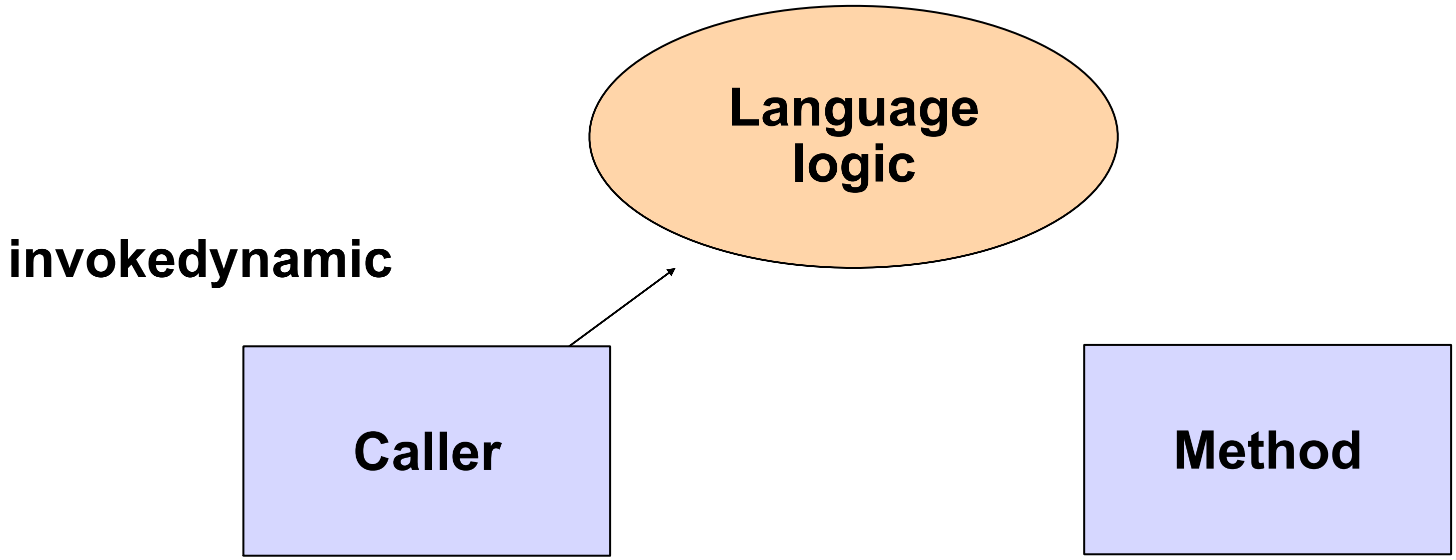
Virtual method call in Java

Monday, October 4, 2010

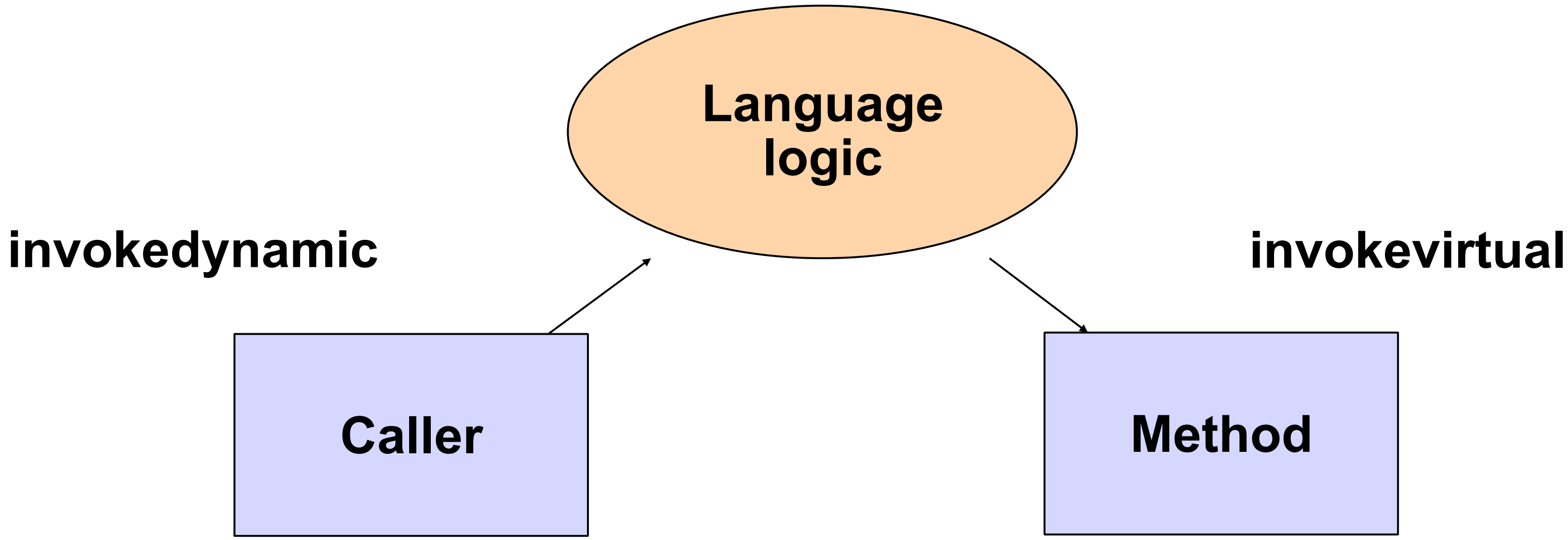
Dynamic method call

invokedynamic

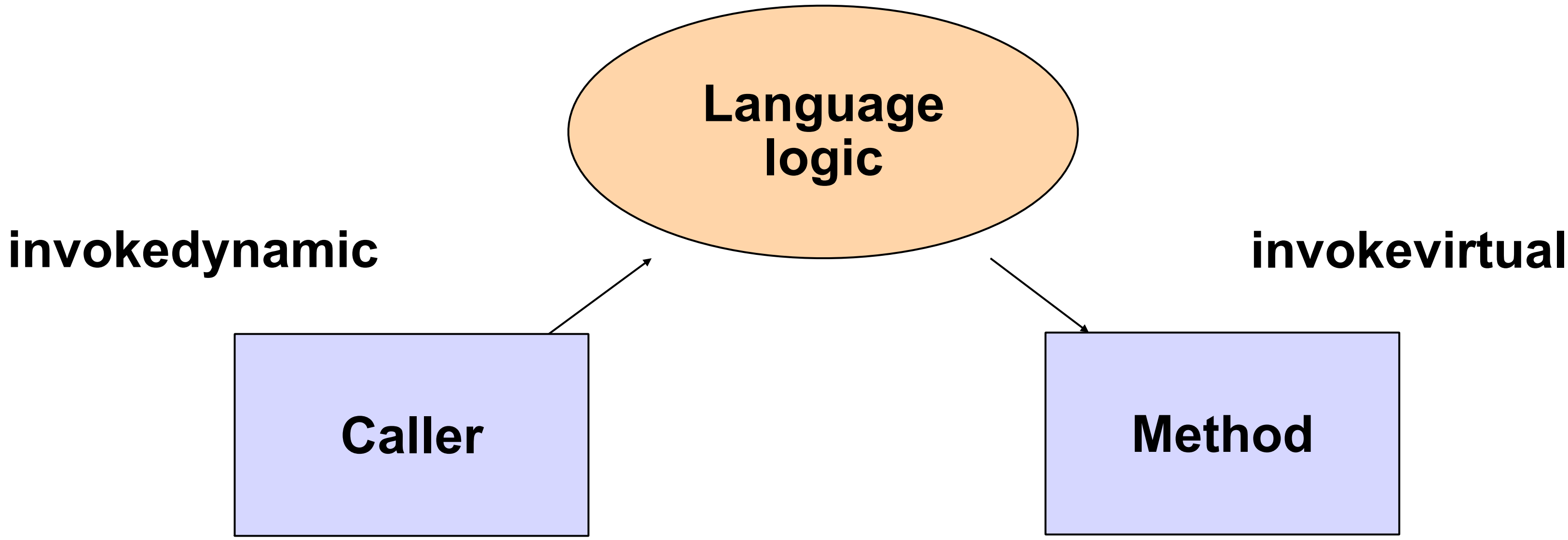
Dynamic method call



Dynamic method call



Dynamic method call

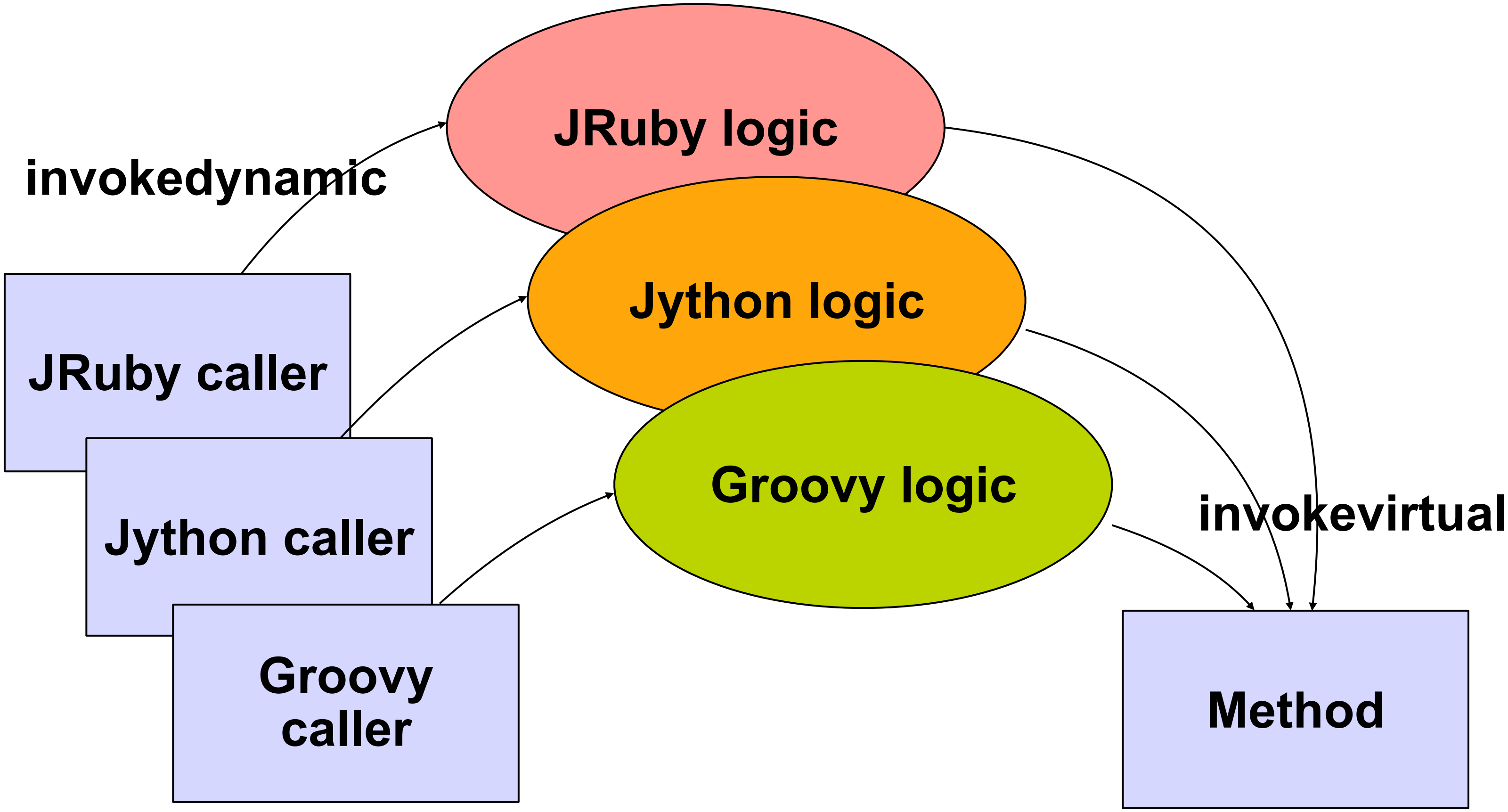


Check which methods are available *now* in each class [open classes]

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]



Monday, October 4, 2010

Language logic is only needed...

*†‡

Language logic is only needed...

*†‡

Language logic is only needed...

ONCE *†‡

Language logic is only needed...

ONCE *†‡

ORACLE

Language logic is only needed...

ONCE *†‡

* Until a different object is assigned to the receiver variable

† Until the receiver's dynamic type is changed

‡ Until the arguments' dynamic types are changed

ORACLE

The deal with method calls (in one slide)

The deal with method calls (in one slide)

- Calling a method is cheap (VMs can even inline!)

The deal with method calls (in one slide)

- Calling a method is cheap (VMs can even inline!)
- Selecting the right target method can be costly
  - Static languages do most of their method selection at compile time (e.g., `System.out.println(x)`)

    Single-dispatch on receiver type is left for runtime
  - Dynamic languages do almost none at compile-time

    Don't re-do method selection for every single invocation!

ORACLE

The deal with method calls (in one slide)

- Calling a method is cheap (VMs can even inline!)
- Selecting the right target method can be costly
  - Static languages do most of their method selection at compile time (e.g., `System.out.println(x)`)

    Single-dispatch on receiver type is left for runtime
  - Dynamic languages do almost none at compile-time

    Don't re-do method selection for every single invocation!
- Each language has its own ideas about linkage
  - The VM enforces static rules of naming and linkage

    Language runtimes want to decide (& re-decide) linkage

What’s in a method call? A sequence of tasks

What's in a method call? A sequence of tasks

- Naming — using a symbolic name

What’s in a method call? A sequence of tasks

- Naming — using a symbolic name
- Selecting — deciding which one to call

What’s in a method call? A sequence of tasks

- Naming — using a symbolic name
- Selecting — deciding which one to call
- Adapting — agreeing on calling conventions

What's in a method call? A sequence of tasks

- Naming — using a symbolic name
- Selecting — deciding which one to call
- Adapting — agreeing on calling conventions

- *Calling – finally, a parameterized control transfer*

What's in a method call? Connection from A to B

What’s in a method call? Connection from A to B

- Including naming, linking, selecting, adapting:

What’s in a method call? Connection from A to B

- Including naming, linking, selecting, adapting:
- …callee B might be known to caller A only by a name

What's in a method call? Connection from A to B

- Including naming, linking, selecting, adapting:
- …callee B might be known to caller A only by a name
- …and A and B might be far apart

ORACLE

What’s in a method call? Connection from A to B

- Including naming, linking, selecting, adapting:
- …callee B might be known to caller A only by a name
- …and A and B might be far apart
- …and B might depend on arguments passed by A

What’s in a method call? Connection from A to B

- Including naming, linking, selecting, adapting:
- …callee B might be known to caller A only by a name
- …and A and B might be far apart
- …and B might depend on arguments passed by A
- …and a correct call to B might require adaptations

What's in a method call? Connection from A to B

- Including naming, linking, selecting, adapting:
- …callee B might be known to caller A only by a name
- …and A and B might be far apart
- …and B might depend on arguments passed by A
- …and a correct call to B might require adaptations

- *After everything is decided, A jumps to B's code.*

What’s in a method call? Several phases

- Source code: What the language says
- Bytecode: What’s (statically) in the classfile

What's in a method call? Several phases

- Source code: What the language says
- Bytecode: What's (statically) in the classfile
- Linking: One-time setup done by the JVM

ORACLE

What’s in a method call? Several phases

- Source code: What the language says
- Bytecode: What’s (statically) in the classfile
- Linking: One-time setup done by the JVM
- Executing: What happens on every call

Phases versus tasks (before invokedynamic)

	Source code	**Bytecode**	**Linking**	**Executing**
Naming	Identifiers	Utf8 constants	JVM “dictionary”	
Selecting	Scopes	Class names	Loaded classes	V-table lookup
Adapting	Argument conversion		C2I / I2C adapters	Receiver narrowing
Calling				*Jump with arguments*

ORACLE

Phases versus tasks (before invokedynamic)

	Source code	Bytecode	Linking	Executing
Naming	Identifiers	Utf8 constants	JVM “dictionary”	
Selecting	Scopes	Class names	Loaded classes	V-table lookup
Adapting	Argument conversion		C2I / I2C adapters	Receiver narrowing
Calling				*Jump with arguments*

Invokedynamic removes some limits

Invokedynamic removes some limits

- Method naming is not limited to Java APIs

Invokedynamic removes some limits

- Method naming is not limited to Java APIs
- Method lookup is not limited to class scopes
  - Completely generalized via Bootstrap Methods

ORACLE

Invokedynamic removes some limits

- Method naming is not limited to Java APIs
- Method lookup is not limited to class scopes
  - Completely generalized via Bootstrap Methods
- Invocation targets can be mixed and matched
  - Adapter method handles can transform arguments
  - Bound method handles can close over “live” data

ORACLE

Phases versus tasks (with invokedynamic)

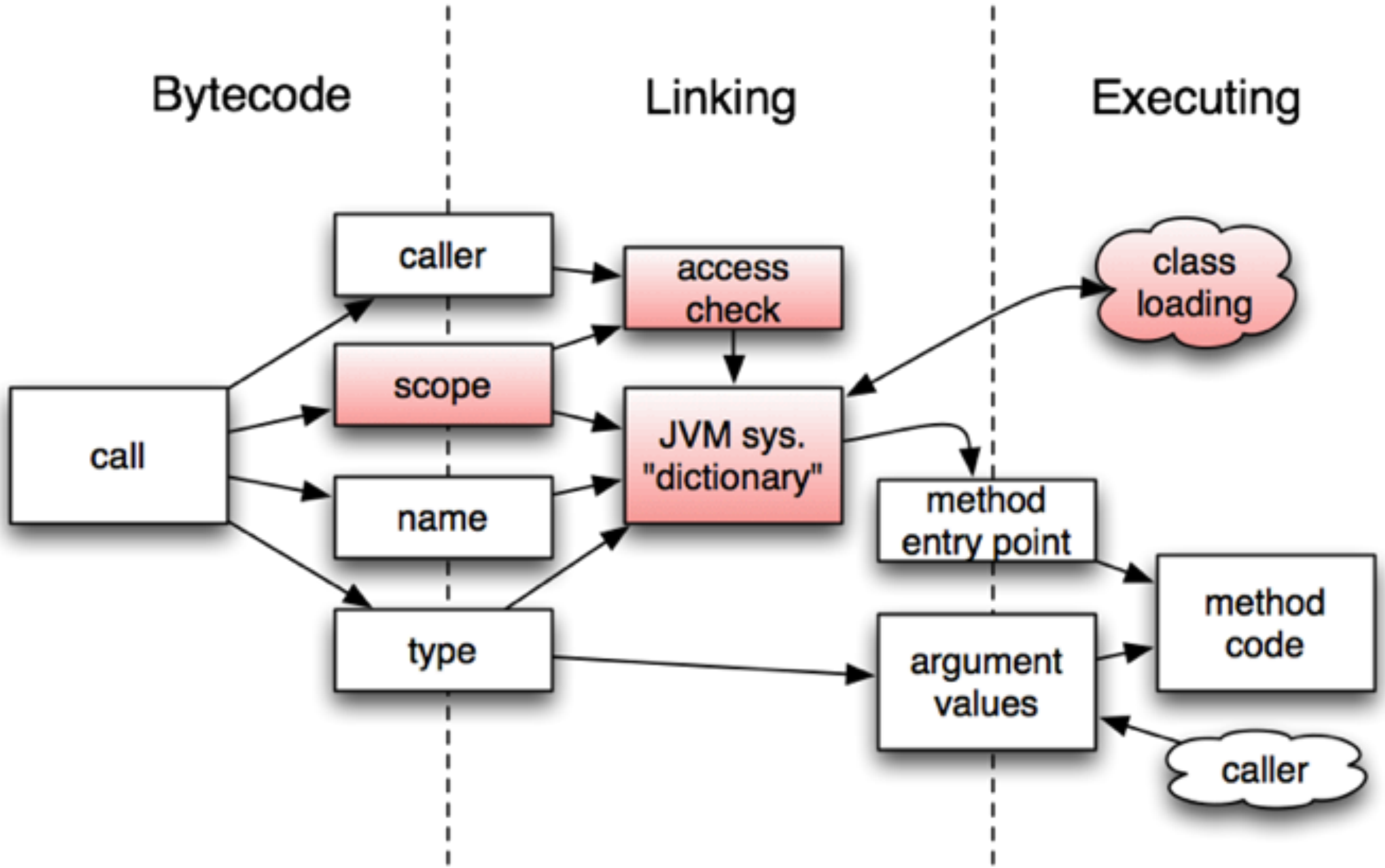
	Source code	Bytecode	Linking	Executing
Naming	∞	∞	∞	∞
Selecting	∞	Bootstrap methods	Bootstrap method call	∞
Adapting	∞		Method handles	∞
Calling				*Jump with arguments*

ORACLE

Phases versus tasks (with invokedynamic)

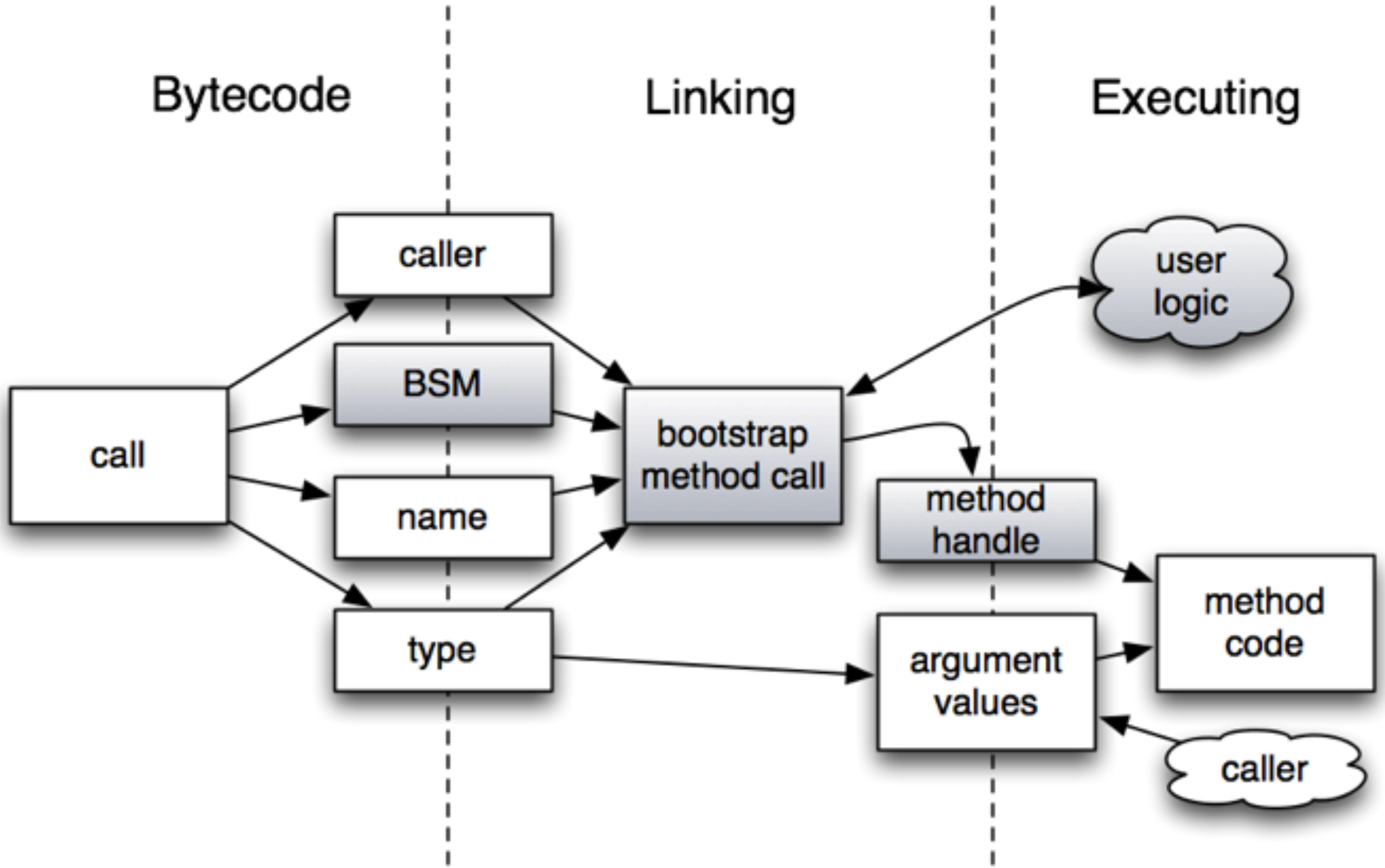
	Source code	Bytecode	Linking	Executing
Naming	∞	∞	∞	∞
Selecting	∞	Bootstrap methods	Bootstrap method call	∞
Adapting	∞		Method handles	∞
Calling				*Jump with arguments*

Phases versus tasks (before invokedynamic)



Monday, October 4, 2010

Phases versus tasks (after invokedynamic)



Method handles and closures

Method handles and closures

- We are working on closures in Java
  - More flexible, less bulky than anonymous inner classes

Method handles and closures

- We are working on closures in Java
  - More flexible, less bulky than anonymous inner classes
- What's in a closure?
  - A small bit of code specified in an expression
  - Optionally, some data associated with it at creation
  - A *target (SAM) type* specifying how the closure will be used

ORACLE

Method handles and closures

- We are working on closures in Java
  - More flexible, less bulky than anonymous inner classes
- What's in a closure?
  - A small bit of code specified in an expression
  - Optionally, some data associated with it at creation
  - A *target (SAM) type* specifying how the closure will be used
- What does the JVM see?
  - A method handle constant specifying the raw behavior
    (Typically a synthetic private, but may be any method.)
  - Optionally, a "bind" operation on the method handle
  - A "SAM conversion" operation to convert to the target type

Invokedynamic and closures?

Invokedynamic and closures?

- An instructive possibility...

Invokedynamic and closures?

- An instructive possibility...

1. Compile the data type and target types as Bootstrap Method parameters.

Invokedynamic and closures?

- An instructive possibility...

1. Compile the data type and target types as Bootstrap Method parameters.
2. When the call is linked, a runtime library selects an efficient representation.

ORACLE

Invokedynamic and closures?

- An instructive possibility...

1. Compile the data type and target types as Bootstrap Method parameters.
2. When the call is linked, a runtime library selects an efficient representation.
3. The call is bound to a method handle which creates the needed closure.

ORACLE

Invokedynamic and closures?

- An instructive possibility...

1. Compile the data type and target types as Bootstrap Method parameters.
2. When the call is linked, a runtime library selects an efficient representation.
3. The call is bound to a method handle which creates the needed closure.
4. When the call is executed, data parameters (if any) are passed on the stack.

ORACLE

Invokedynamic and closures?

- An instructive possibility...

1. Compile the data type and target types as Bootstrap Method parameters.
2. When the call is linked, a runtime library selects an efficient representation.
3. The call is bound to a method handle which creates the needed closure.
4. When the call is executed, data parameters (if any) are passed on the stack.
5. The method handle folds it all together, optimally.

JSR 292 design news

JSR 292 design news

- Method handle constants
  - Allows bytecode to work with method handles, as directly as it works with methods themselves
  - Initially motivated by thinking about closure compilation

ORACLE

JSR 292 design news

- Method handle constants
  - Allows bytecode to work with method handles, as directly as it works with methods themselves
  - Initially motivated by thinking about closure compilation
- Bootstrap Methods localized to invokedynamic calls
  - Allows dynamic call sites to be woven independently

ORACLE

JSR 292 design news

- Method handle constants
  - Allows bytecode to work with method handles, as directly as it works with methods themselves
  - Initially motivated by thinking about closure compilation
- Bootstrap Methods localized to invokedynamic calls
  - Allows dynamic call sites to be woven independently
- Class-specific values (for metaclass caching)
  - ThreadLocal : Threads :: ClassValue : Class

JSR 292 design news

- Method handle constants
  - Allows bytecode to work with method handles, as directly as it works with methods themselves
  - Initially motivated by thinking about closure compilation
- Bootstrap Methods localized to invokedynamic calls
  - Allows dynamic call sites to be woven independently
- Class-specific values (for metaclass caching)
  - ThreadLocal : Threads :: ClassValue : Class
- “Live” constants
  - Generalization of Class and Method Handle constants
  - Linked into the constant pool by a user-specified BSM

What's next? A standard

What's next? A standard

- Reference Implementation driven as part of JDK 7

What's next? A standard

- Reference Implementation driven as part of JDK 7
- Experiments have been done with it:
  - JRuby retargeting (Charlie Nutter)
  - Rhino (JavaScript) investigation
  - "PHP Reboot" project (Rémi Forax)

What’s next? A standard

- Reference Implementation driven as part of JDK 7
- Experiments have been done with it:
  - JRuby retargeting (Charlie Nutter)
  - Rhino (JavaScript) investigation
  - “PHP Reboot” project (Rémi Forax)
- Expert Group has been actively discussing the spec.

What’s next? A standard

- Reference Implementation driven as part of JDK 7
- Experiments have been done with it:
  - JRuby retargeting (Charlie Nutter)
  - Rhino (JavaScript) investigation
  - “PHP Reboot” project (Rémi Forax)
- Expert Group has been actively discussing the spec.
- Nearing a second draft specification (this year)

What's next? Da Vinci projects

What’s next? Da Vinci projects

- The Da Vinci Machine Project continues

What's next? Da Vinci projects

- The Da Vinci Machine Project continues
- Community contributions:
  - Continuations
  - Coroutines
  - Hotswap
  - Tailcalls
  - Interface injection

ORACLE

What's next? Da Vinci projects

- The Da Vinci Machine Project continues
- Community contributions:
  - Continuations
  - Coroutines
  - Hotswap
  - Tailcalls
  - Interface injection
- Gleams in our eyes:
  - Object "species" (for splitting classes more finely)
  - Tuples and value types (for using registers more efficiently)
  - Advanced array types (for using memory more efficiently)

ORACLE

What's next? All of the above, fast and light

What's next? All of the above, fast and light

- Architecture $\neq$ optimization

What's next? All of the above, fast and light

- Architecture ≠ optimization
- Architecture → *enables* optimization

What's next? All of the above, fast and light

- Architecture ≠ optimization
- Architecture → *enables* optimization

- Efficient method handle creation

What's next? All of the above, fast and light

- Architecture ≠ optimization
- Architecture → *enables* optimization

- Efficient method handle creation
- Compact representations (fused MH/SAM nodes)

ORACLE

What's next? All of the above, fast and light

- Architecture ≠ optimization
- Architecture → *enables* optimization

- Efficient method handle creation
- Compact representations (fused MH/SAM nodes)
- Memory-less representations

ORACLE

“Fixnums” – tagged immediate pseudo-pointers

“Fixnums” – tagged immediate pseudo-pointers

- In Java, primitives can be “autoboxed”
  - This convenience was added in JDK 5

“Fixnums” – tagged immediate pseudo-pointers

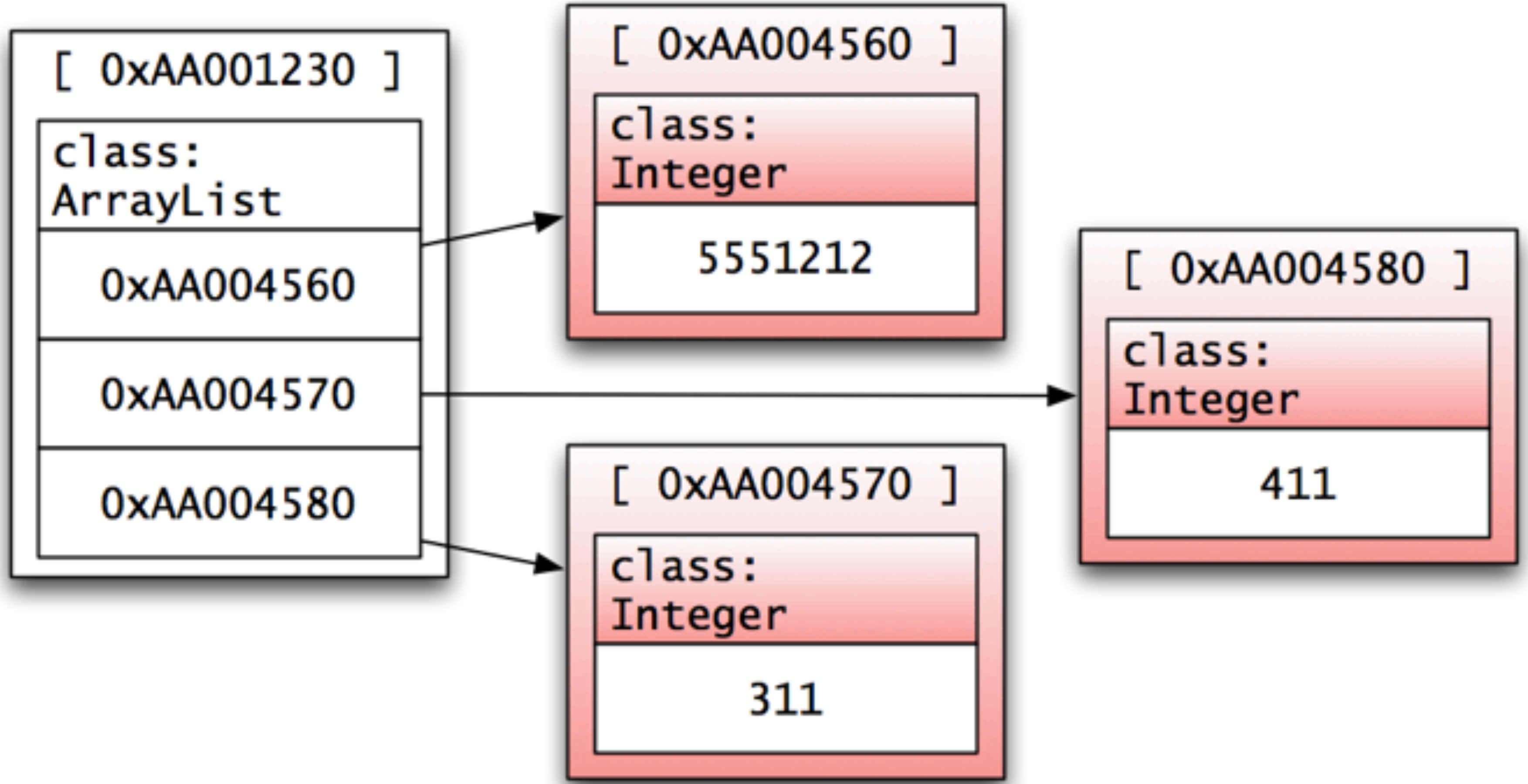
- In Java, primitives can be “autoboxed”
  - This convenience was added in JDK 5
- Boxing is expensive and tricky to optimize
  - In general it requires building a whole “wrapper” object

“Fixnums” – tagged immediate pseudo-pointers

- In Java, primitives can be “autoboxed”
  - This convenience was added in JDK 5
- Boxing is expensive and tricky to optimize
  - In general it requires building a whole “wrapper” object
- Some older systems (Lisp, Smalltalk) are smarter
  - They use the object pointer itself to store the primitive value
  - The pointer is “tagged” to distinguish it from a real address

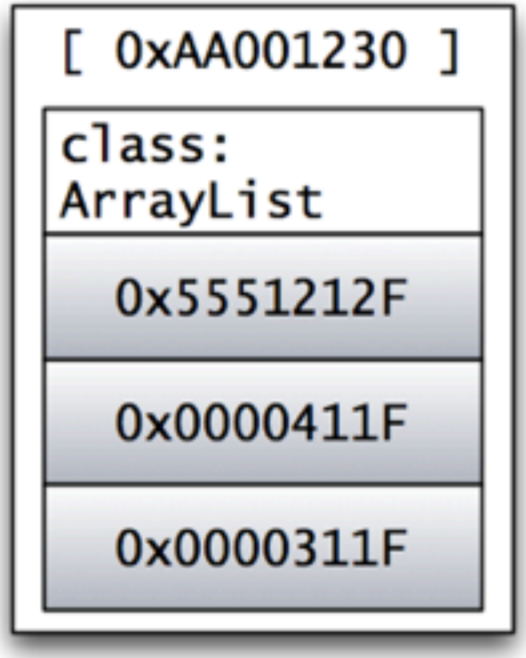
A list of integer values (before fixnums)

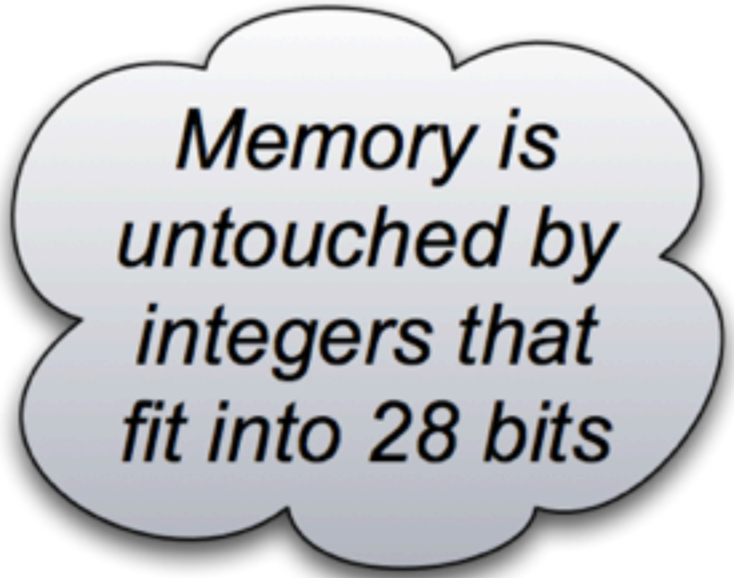
```
Arrays.asList(0x5551212, 0x411, 0x311)
```



A list of integer values (after fixnums)

```
Arrays.asList(0x5551212, 0x411, 0x311)
```





Fixnums in the Java VM

Fixnums in the Java VM

- The JVM can also do the “fixnum” trick

Fixnums in the Java VM

- The JVM can also do the “fixnum” trick
- This will make Integer / int conversions very cheap

Fixnums in the Java VM

- The JVM can also do the “fixnum” trick
- This will make Integer / int conversions very cheap
- No need for special “int” container types
  – Filter, Predicate vs. intFilter, intPredicate, etc.

ORACLE

Fixnums in the Java VM

- The JVM can also do the “fixnum” trick
- This will make Integer / int conversions very cheap
- No need for special “int” container types
  - Filter, Predicate vs. intFilter, intPredicate, etc.
- One catch: Doesn’t work well for “double” values

Implications for languages

Implications for languages

- Bootstrap Methods — new link-time hook
  - helps with call site management (JRuby, JavaScript)
  - can help with one-time representation setup (closures)

Implications for languages

- Bootstrap Methods — new link-time hook
  - helps with call site management (JRuby, JavaScript)
  - can help with one-time representation setup (closures)
- Method Handles — lower-level access to methods
  - faster and more direct than reflection
  - more compact than inner classes

Implications for languages

- Bootstrap Methods — new link-time hook
  - helps with call site management (JRuby, JavaScript)
  - can help with one-time representation setup (closures)
- Method Handles — lower-level access to methods
  - faster and more direct than reflection
  - more compact than inner classes
- SAM conversion — bridge to higher-level APIs
  - no more spinning of 1000’s of tiny inner classes (Scala)

Implications for languages

- Bootstrap Methods — new link-time hook
  - helps with call site management (JRuby, JavaScript)
  - can help with one-time representation setup (closures)
- Method Handles — lower-level access to methods
  - faster and more direct than reflection
  - more compact than inner classes
- SAM conversion — bridge to higher-level APIs
  - no more spinning of 1000’s of tiny inner classes (Scala)
- Class values — direct annotation of arb. classes
  - no more fumbling with class-keyed hash tables (Groovy)

ORACLE

Implications for languages

- Bootstrap Methods — new link-time hook
  - helps with call site management (JRuby, JavaScript)
  - can help with one-time representation setup (closures)
- Method Handles — lower-level access to methods
  - faster and more direct than reflection
  - more compact than inner classes
- SAM conversion — bridge to higher-level APIs
  - no more spinning of 1000’s of tiny inner classes (Scala)
- Class values — direct annotation of arb. classes
  - no more fumbling with class-keyed hash tables (Groovy)
- Fixnums — Less pain dealing with primitives

To find out more...

To find out more...

- “Bytecodes meet Combinators: invokedynamic on the JVM”, Rose VMIL 2009
  http://blogs.sun.com/jrose/entry/vmil_paper_on_invokedynamic

To find out more...

- “Bytecodes meet Combinators: invokedynamic on the JVM”, Rose VMIL 2009
  http://blogs.sun.com/jrose/entry/vmil_paper_on_invokedynamic
- “Optimizing Invokedynamic”, Thalinger PPPJ 2010
  http://blogs.sun.com/jrose/entry/an_experiment_with_generic_arithmetic

To find out more...

- “Bytecodes meet Combinators: invokedynamic on the JVM”, Rose VMIL 2009
  http://blogs.sun.com/jrose/entry/vmil_paper_on_invokedynamic
- “Optimizing Invokedynamic”, Thalinger PPPJ 2010
  http://blogs.sun.com/jrose/entry/an_experiment_with_generic_arithmetic
- JVM Language Summit 2010
  http://wiki.jvmlangsummit.com

ORACLE

To find out more...

- “Bytecodes meet Combinators: invokedynamic on the JVM”, Rose VMIL 2009
  http://blogs.sun.com/jrose/entry/vmil_paper_on_invokedynamic
- “Optimizing Invokedynamic”, Thalinger PPPJ 2010
  http://blogs.sun.com/jrose/entry/an_experiment_with_generic_arithmetic
- JVM Language Summit 2010
  http://wiki.jvmlangsummit.com
- Da Vinci Machine Project
  http://openjdk.java.net/projects/mlvm/

To find out more...

- "Bytecodes meet Combinators: invokedynamic on the JVM", Rose VMIL 2009
  http://blogs.sun.com/jrose/entry/vmil_paper_on_invokedynamic
- "Optimizing Invokedynamic", Thalinger PPPJ 2010
  http://blogs.sun.com/jrose/entry/an_experiment_with_generic_arithmetic
- JVM Language Summit 2010
  http://wiki.jvmlangsummit.com
- Da Vinci Machine Project
  http://openjdk.java.net/projects/mlvm/

- Friday 9/25 bonus: http://scalaliftoff.com/
  (discount code = *scalaone*)

ORACLE

SOFTWARE. HARDWARE. COMPLETE.