your next JVM:
Panama, Valhalla, Metropolis

John Rose, JVM Architect

Devoxx San Jose, March 2017

ORACLE

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.

Purpose of this talk:

- Outline major trends in the evolution of the Java Runtime (JVM)
- Show how we are pushing those trends forward in OpenJDK.

- *Warning: The future is complicated. Download in 3, 2, 1…*

ORACLE

What should the JVM look like in ~~20~~ 18 years?

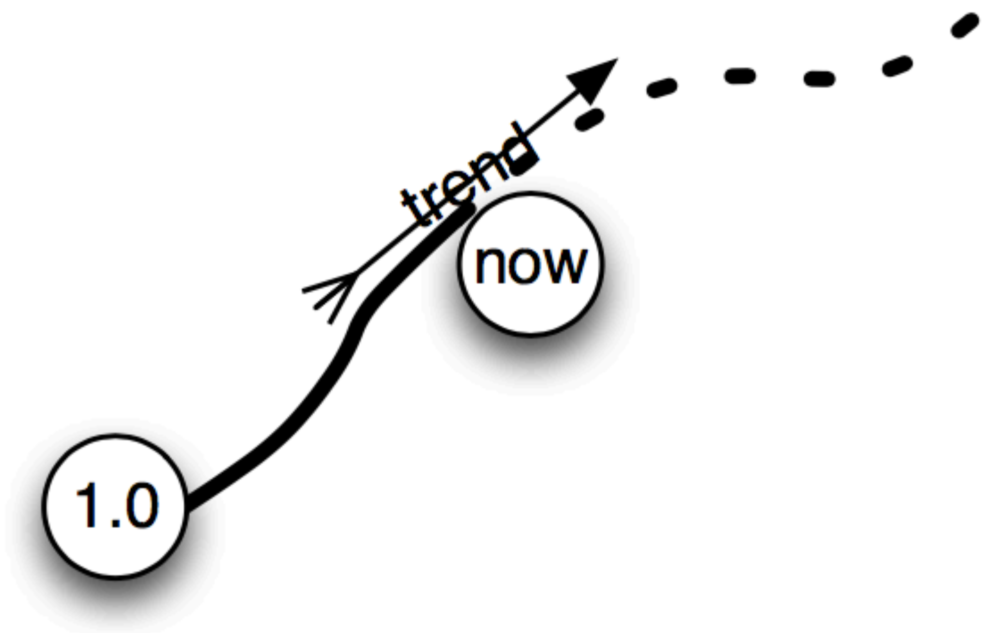
(eight not-so-modest goals, from JVMLS 2015)

- **Uniform** model: Objects, arrays, values, types, methods “feel similar”
- Memory efficient: tunable **data** layouts, naturally local, pointer-thrifty
- Optimizing: Shared **code** mechanically customized to each hot path
- Post-threaded: Confined/immutable data, granular **concurrency**
- **Interoperable**: Robust integration with non-managed languages
- Broadly useful: Safely and reliably runs most modern **languages**.
- **Compatible**: Runs 30-year-old dusty JARs.
- Performant: Gets the most out of major **CPUs** and systems.

ORACLE

Forecasting is hard

(but sometimes you can draw a vector)



ORACLE

Some trends

- Predictability / reliability / security
  - Manageable behavior: low-pause GCs, JVM provisioning
  - Ergonomics, metrics, monitoring: Flight Recorder, logging, telemetry
- Density / scaling
  - Data sharing, fast startup (AOT, Class Data Sharing), big heaps
  - Immutability (of native and AOT code now, data later)
- Polyglot / interoperability (more indy, Panama, machine code snippets)
- Layered implementations – strong abstractions + lowering
  - Macro-instructions (indy), simplified data model (value types)
  - Java-on-Java (JSR 292 v2, Panama, Graal, Metropolis)

Big Idea: Platform interoperability

- Idea: Make native code/data look more like Java code/data
  - A "heal the rift" move: abate invidious choices between on- and off-heap
- Depends partly on value types
  - To import native types (vectors, unsigned, complex) w/o boxing
  - To optimize "smart cursors" into foreign arrays and structs
- Depends on low-level bridges to non-Java ABIs and data layouts
  - The JIT makes a good code generator for this sort of thing
- Enables tight coupling with a wider range of data, APIs, ISAs
  - Both legacy (COBOL) and cutting edge (GPUs)
  - Low level punch-through to access ISA intrinsics directly ("Vector API")

Project Panama — what comes after JNI

- Zero hand-written code
  - Header file “groveller” extracts metadata which drives binder
  - Example: Replace 10Kb of handwritten Java + C with 6Kb of metadata.
- Direct, optimizable native calls from JIT. (FFI intrinsic MethodHandles)
- Direct, optimizable access to off-heap, fewer copies to/from on-heap.
  - Smart, type-safe, storage-safe pointers, references, structs, arrays.
- Direct access to hardware, including jumbo primitives like uint256_t.
  - Vector API for direct coding with platform vector instructions.
  - JIT-integrated assembly-language snippet mechanisms.
- *Better use of native resources, data structures, and libraries.*

Big Idea: Java-on-Java

- Idea: Implement the Java runtime using Java itself (less C++/asm)
  - Another “heal the rift” move (see a pattern?)
  - Up-level and simplify critical technology components
- Depends partly on interoperability (Panama)
  - To integrate Java components (e.g., Graal) into HotSpot
- Depends partly on flat data (Valhalla)
  - Temporary IR spikes in JIT are limited (partly) by data density
- Reduces all costs: maintenance, innovation, security
- Current practice: the whole JDK, MH runtime, AOT (coming soon)
- Some current experiments: Graal, Substrate VM, Bifröst

ORACLE

Project Metropolis: City of Tomorrow

- Experimental clone of JDK 10 (*not* for immediate release)
- Hosting work on AOT and the Graal compiler
- Definition of “System Java” for implementing HotSpot modules.
  - Experimentation with SVM-style deployment.
- Translation of discrete HotSpot modules into System Java.
  - Candidates: MethodLiveness, verifier, reflection, class file parser
- The Big One: Compilation of Graal as System Java for JIT
  - Replacement for C2, then C1, then stub and interpreter generators.
  - This will take a long time, but it’s a necessary technology refresh.
- *Tomorrow’s reference implementation!*

Big Idea: Post-thread concurrency

- Idea: work around “dinosaur threads”
  – Break the live bits into smaller parts
  – Leave behind the fossils
- “Fibers”
- Depends on:
  – Tricky cuts in JVM interpreter and JIT
  – Lots of library work; all blocking APIs need attention

ORACLE

Fibers: Dinosaurs as draft animals, not pets

- Fibers = caller/callee snippets decoupled from threads
- A fiber ***mounts*** on a thread
  - can also ***dismount*** into the heap
- Current experiments:
  - better stack walking
  - flexible bytecode interpretation (more modes)
- Lots of library work needed; all **blocking** APIs need attention
- Tricky interactions with synchronization (VarHandles, transactions?)
- JVM is responsible for right-sizing a **frame object** (continuation)

Big Idea: Value types — Project Valhalla

- Idea: "heal the rift" ("caulk the seam") between classes and primitives
- Distinguish legacy, by-reference "L-types" from new, by-value "Q-types"
- Value proposition for Q-types: *"Codes like a class, works like an int."*
- Depends on parametric polymorphism (any-vars not limited to L-types)
  - Which depends on template-like classes and their "species"
- Requires deep cuts to JVM code and data model
  - Quintessential "big ticket item". These are the Last Types We Will Need.
- Enables pervasive changes throughout the stack
  - Comparable to the impact of generics or lambdas
- Extends the benefits of class-based encapsulation to all values.

ORACLE

Big Idea: Parametric polymorphism

- Idea: Make generalizations across ***all*** values (reference/primitive)
  - A “generalization”, in Java code, is a generic class or method.
- Depends on templates
- Depended on by:
  - Stream cleanup (no more IntStream)
  - Values
  - New array types
  - Foreign/off-heap reference and pointer types
- This is also part of Valhalla! Type variable = universal value container.

ORACLE

Parametric polymorphism is hard

- Challenge: Primitives look really, really different from references.
  - This is why today's generics only talk about object references.
  - But today's situation doesn't scale to value types. So let's solve primitives.
- Issues: Static typing, equality, data model, no methods on primitives.
- Possible solution
  - Build an efficient mapping to treat primitives as Q-types.
  - Then everything boils down to Q-types (values) and L-types (object refs)
  - Top it off with U-types (disjoint union of Q- and L-types) for generic code.
- (This is not the way we'd do it if we weren't already adding Q-types!)

Impact of parametric polymorphism

- There are fewer “kinds” of types.
  - Everything has methods, and (maybe) has interfaces.
  - Everything has the same levels of equality (Lisp EQ, EQV, EQUAL).
- Introduction of templated classes
  - Pointer polymorphism isn’t enough; we need representation polymorphism
  - This implies some new distinctions; ArrayList<int> ≠ ArrayList<byte>
  - C++ templates do this statically; the JVM can do this dynamically.
- Dangerous Opportunity (not to be confused with “crisis” 危劫)
  - Maybe the expression “x * y” is generic?
  - A type-ful solution would amount to restricted operator overloading.

JVM “template classes” and “species”

- In JVM, template-ness is “really” constant-polymorphism.
  - I.e., a template class has holes in its constant pool.
- Holes are type-variables for parametric polymorphism.
  - Maybe holes could be numbers, strings, functions, etc.? (Cf. C++)
- Requires deep thinking about “what’s a constant pool”.
- Hardest problem: Avoid premature “code splitting”
  - Execute cold code from one set of bytecodes shared by species
  - The JIT inlines and customizes hot code, in the usual way.
  - Result: No footprint cost for seldom-used template instances.

Let’s look at some smaller ideas

(perhaps more self-contained – or maybe just more distant)

- Stack introspection
- Length-polymorphism
- Bootstrap methods everywhere
- Immutability & montonicity

VM design, like language design, is complicated like a Chinese puzzle.

When it’s finished it should look simple, from the outside.

Stack reification (introspecting the thread)

(backtraces 2.0)

- Walk “live” stack frames.
- See classes & methods, not just their names.
- Observe full frame states (locals, stack, monitors, BCI)
- ***Edit*** the stack
  - Replace ongoing computations with “better” ones
- Application: Dismount a blocking call from a thread onto a fiber
- Application: Generator coroutines
- Application: Replace a serial algorithm by a parallel one.

ORACLE

Length polymorphism

(fused arrays)

- Poster child: String sub-object costs 16-24 bytes
  - String.value pointer = 4, 2nd header = 8, 2nd klass = 4, 2nd frag = 0..8
  - Matches or betters “bytewise string” tactic for String.length ≤ 24
- Generalization: Mark a private array as fused.
  - To the JVM, it looks like an array with an object jammed on the front.
- Requires array-like proxies to implement “baload”, etc.
- Requires GC cooperation to interpret embedded length field.
- Requires deep refactoring of new/invoke<init> ⇒ invoke<new>
  - ~~Constructor~~ Factory must be reponsible for object sizing & allocation.
- Bonus: Multiple fused sub-arrays?

Bootstrap methods everywhere

(works in progress)

- Embrace and double down on dynamicity of indy BSMs
- Use them wherever the JVM already has a “hook” for resolution
  – ldc of CONSTANT_foo
  – invoke of CONSTANT_Methodref or field ref
  – creation of a method or class

- Three examples:
  - Constants
  - Lazy Boilerplate Methods
  - BSMs in source code

ORACLE

Bootstrap methods everywhere

- **Constants**
- Lazy Boilerplate Methods
- BSMs in source code

What’s in a constant?

(in the current JVM)

- Primitives: CONSTANT_{Int,Long,Float,Double}
- Strings, type names: CONSTANT_{String,Class}
- API points: field and method references (w/ nameAndType)
- Indy and MHs: CONSTANT_{InvokeDynamic,Method{Handle,Type}}

What’s in a constant?

(possible future use cases)

- Containers: arrays, Lists, Sets, Maps
- Value types: enums, BigInteger, Complex<F>, Point, Line, Color
- DSL snippet, compiled AST: RE Pattern, *(insert Linq-like feature here)*
- Support for ***templates*** for most of the above
  - "foo" + bar+(++barCount) + "baz" — *DONE!*
  - #"foobar{++barCount}baz" — *maybe?*
  - List#("foo", bar+(++barCount), "baz”)
- Key constraint: They can be internally mutable, but their value must persist reliably. Just like java.lang.String.
  - They are “materialized” lazily, perhaps interned, at resolution time.

What’s in a constant?

(a future-proof design)

- Should require only a handful of new CONSTANT_Foo types.
- Should be programmable via meta-factories (a la indy).
- Should enable but not mandate compression schemes.
- Should allow a generous amount of constant “payload bits”.
- Should clearly work for arrays, collections, and something value-like.

What's in a constant?

(ConDy, and more)

- CONSTANT_Dynamic (puts a BSM in a constant)
  - Equipped with BSM+args and type *(cf. indy = {{BSM+args},{name+type}})*
  - Resolved by executing BSM.invoke(lookup, type.class, static arg…)
  - Same rules as indy for dealing with exceptions, races, etc.
- CONSTANT_Group (overcomes the arity limit to BSMs)
  - Arbitrary group of constants, with 16-bit count.
  - Each constant can refer to main constant pool or can be inline.
  - Can nest, so can act like an S-Expression (for ASTs).
- CONSTANT_Bytes (delivers maximum sharing and density)
  - Up to Integer.MAX_VALUE bytes, reified as zero-copy ByteSequence.

ORACLE

What’s in a constant?

(application to arrays)

- CONSTANT_Dynamic[ArrayFactory::constArray, int[].class, theBits]
- Where theBits = CONSTANT_Bytes[…bit image of desired array...]
- Where constArray(int[].class, theBits) makes and fills the array.
- What’s in the bits? Private agreement between javac and constArray.
  - Probably fodder for DataInput. (Big-endian by tradition.)
  - Could be var-ints (UNSIGNED5) or some other compression method.
- KEY CONSTRAINT: The array must be immutable (like a string).
  - Dependency: Support for frozen arrays.
- Alternative to frozen arrays: Only support CharSequence and similar.

Bootstrap methods everywhere

- Constants
- **Lazy Boilerplate Methods**
- BSMs in source code

Boilerplate reduction (aka. “bridge-o-matic”)

(use cases)

- Signature adjustment bridges
  - Adjust arguments: compareTo(String) ≡ compareTo(T<:Comparable)
  - Adjust returns: clone():Object ≡ clone():EnumSet
  - Bridges must be generated statically by javac
  - Sometimes this is too early to know critical class hierarchy info.
- Boilerplate methods
  - For Amber data classes: constructor, toString, hashCode, equals
  - Other method builders, such as Comparable.compareTo (lexicographic
- Panama binder (metadata-driven accessor creation)
- Valhalla per-species code (unclear if this is needed, though)

JVMs have boilerplate problems too!

- Amber might reduce your data-class source code to a single line.
- But if your class file is 100x larger your app. is still fat.
- Solution: Generate only the bytecodes you actually use.
- Pretty toString is there when you want it, but has no static cost.

- We must eagerly generate boilerplate when circumstances require.
  - AOT of shared, provisioned application runtime.
  - Bootstrap cycle breaking in the JDK itself.

Lazy boilerplate methods

(details)

- Allow a method to be equipped with its own bootstrap method (+args).
- Method is resolved on first execution (as if it contains an indy)
- Resolved by executing BSM.invoke(lookup, name, type, static arg…)

- Trivia question: What kind of method, *today*, has no bytecodes but is executable?
  – I.e., it is not ACC_ABSTRACT, yet it lacks a Code attribute.
- Answer: A native method.
  – Perhaps ACC_NATIVE+ACC_BRIDGE is the marker for a bridge-o-matic.

ORACLE

Lazy boilerplate methods

(more variations)

- Existing bridges correspond to MH::asType transforms.
  - Actual as-loaded class hierarchy can be queried and validated.
- Type-polymorphism: A class declares a BSM for all types on a name.
  - Callers can avoid boxing primitives and varargs arrays.
- Name-polymorphism: A class declares a BSM for all names.
  - Aka. the “doesNotUnderstand” hook. (The original “invokedynamic”.)
- Inherited boilerplate
  - Interface “PrettyToStringForDataClasses” defines a default method
  - Default method is lazy boilerplate, instantiated *in each concrete subtype*

Bootstrap methods everywhere

- Constants
- Lazy Boilerplate Methods
- **BSMs in source code**

ORACLE

BSMs in source code

(Const-able types, method builder recipes)

- Project Amber is exploring source-code syntaxes for BSM constructs
- Programmable “ConDy” constants using special javac intrinsics
- Programmable declarative method builders using InDy
- Result: Wider adoption of BSM-based mechanisms
- The JVM will be ready.

Bootstrap methods everywhere

(THE DOWNSIDE!)

The smarter your bytecode instructions become, the smarter your compiler has to be.

- JIT compilers usually work after resolution (including BSM calls)
- But AOT compilers cannot do this, in general.
  - Must be able to leave “holes” for non-AOT resolution logic (slow)
  - Or, must be able to simulate or predict the action of the BSM (tricky)
- Solution: Allow programmers to code BSMs directly when needed.
- Solution: Teach jlink and AOT to “expand” BSMs when needed.
  - BSMs must somehow support partial evaluation at AOT time (WIP!)

Immutability

(motivations)

- Simpler Semantics: Fewer program states to reason about.
  - Surprising or non-local side effects lead to bugs and races.
- Better sharing (better system density)
  - Between mutually untrusting actors. (Race condition = security vuln.)
  - Between JVM instances (if bits are in mapped file or shared memory)
- No more defensive copying.
  - If identity is suppressed, “expansive copying” can improve memory locality.
- Semantic simplicity enables ahead-of-time execution.
  - AOT of bootstrap methods or constant expressions, for example.

Immutability

(a quick observation for us Java programmers)

Imagine a version of Java where “final” is the default, and you need a keyword “nonfinal” to get variable mutability, method overridability, or class extensibility.

You would have to opt-in to those power features. Simpler-looking codes would have simpler semantics.

From that POV, today’s Java gets the defaults backwards.

ORACLE

Immutability
(concrete cases)

- Frozen arrays or “array values”
  - array constants, secure array sharing
- Boxes that really work: Immutable final fields.
  - May require stronger guarantees on final field semantics.
  - We need to tame setAccessible(true) and deserialization.
- Frozen objects: Immutable ***non-final*** fields. (Cf. C++ const structs.)
  - Requires final-polymorphic classes, and/or a lock-bit in object headers.
  - Builder packs a private “larval” mutable, delivers a frozen result.

ORACLE

Immutability

(the connection with identity)

- Headers can carry state if we don't watch out.
  – Monitor entry must be forbidden on frozen objects. (Check the lock bit.)
- More subtly, object identity can cause bugs similar to mutation-bugs.
  – Imagine if List.contains "forgot" to use equals and just used op==
- Current thinking: Leave identity as-is, but *only for legacy L-types*
- Add new Q-types for identity-hostile values
  - Some legacy L-type classes can "divest" identity by conversion to Q-types
  - Add U-types (union of L- and Q-) for code paths which must mix kinds.

Fit and finish: Small polishes to the JVM

- Nestmates
- Sealed interfaces
- Sealed fields

Nestmates: Smaller circles of trust

- In Java, all nested classes under one top-level class trust each other.
- This is not true in the JVM.
  - Javac generates package-private bridge methods for side-channels.
  - This makes bridged privates accessible throughout the package!
- Better solution: Clearly define “nestmates” at JVM level.
  - Use a new attribute, rather than parsing InnerClasses attribute.
  - Hardwire this attribute into the JVM’s access checking logic.
  - Result: No more package-private bridges into nest privates.
- Will also help clarify the status of runtime injected code (“host class”).
- Clean concentric circles: nest < package < module < everywhere

ORACLE

Sealed interfaces: Better information hiding

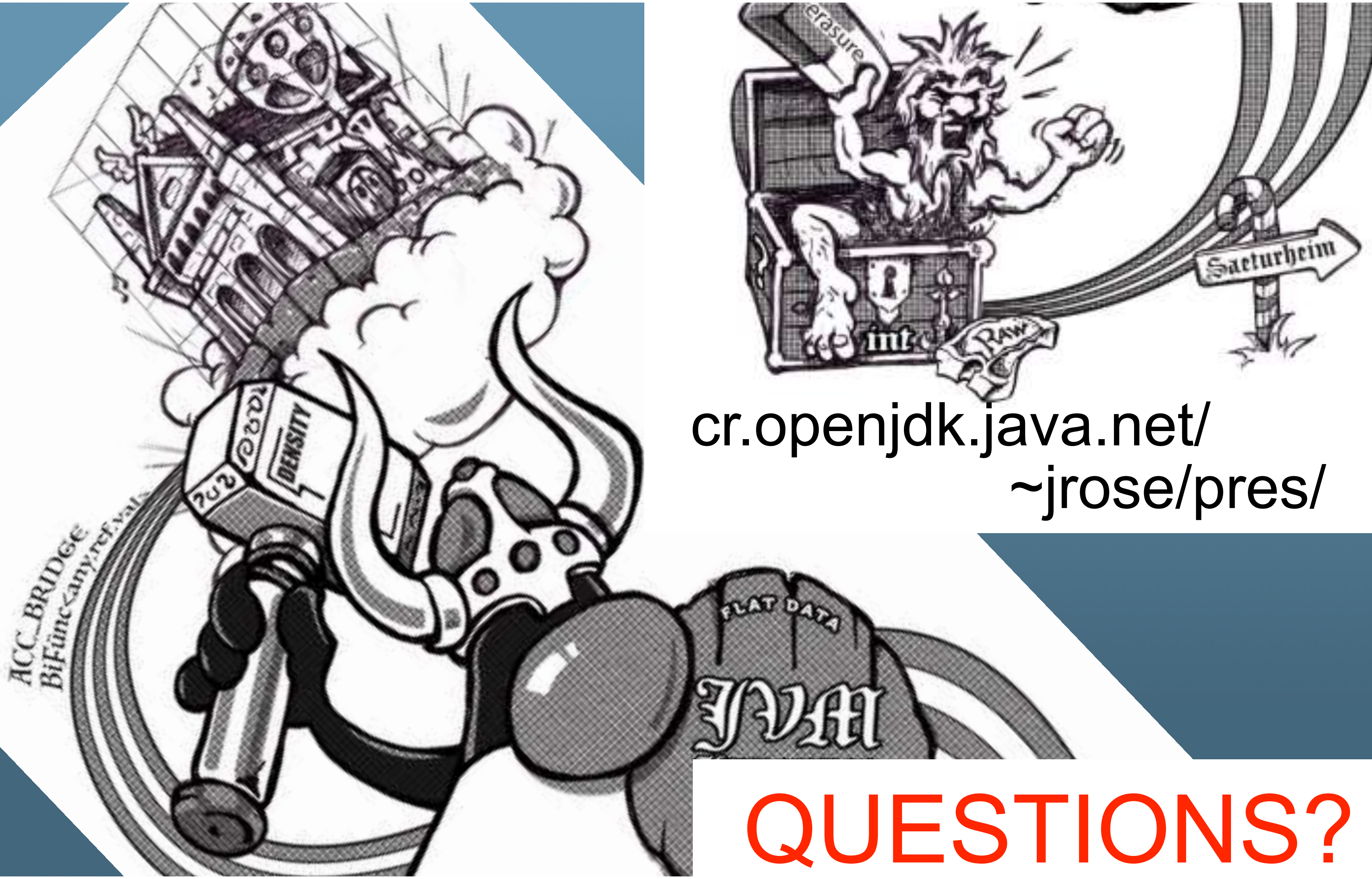
- Interfaces, combined with private implementations, hide information
- Problem: They can be spoofed by untrusted code.
  - Root cause: Interfaces don’t have constructors.
  - Workaround: Recode as abstract class with private constructor.
- Better solution: Allow interfaces to be declared “sealed”.
  - In effect, it is as if the interface declared a private constructor.
  - Interface-specific contracts can be rigorously imposed—no spoofing.
- Possible “seal” granularity is nest, package, and/or module.
  - Nest is the most useful: Only nest mates may implement.

Sealed fields: Better mutability

- Many objects have public getters and private setters (or no setters)
- Problem: The most natural notation would be public fields
  - But that would allow untrusted parties to scribble non-final fields
- Cover this case by adding *asymmetric* field accessibility
- A “sealed field” looks mutable inside the class, and final outside.
- Important usability tweak for mutable data classes (if we do those)

ORACLE

The previous 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.



cr.openjdk.java.net/
~jrose/pres/

QUESTIONS?