Six New Trends in the JVM

John Rose, JVM Architect

Brussels FOSDEM, February 2018

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.

What should the JVM look like in ~~20~~ 17 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 [**A-Z]PUs** and systems.

Keep Java vibrant *(the JVM parts)*

Applicable to today's workloads: Cloud dev/ops

Applicable to today's (& tomorrow's) platforms

Compatible with past code, trusted by today's coders (>10M)

Reliable, even secure (fewer sharp edges)

Debuggable; good tooling: IDEs, jcmd, Java flight recorder, …

Open source collaboration, no paywall, rapid progress

Best AOT/JIT/GC/RT: throughput, latency, scaling, footprint, startup…

Best access to the "metal" of your platform (fast, safe, simple)

⇒ **Run the World on Java!**

Many Technical Initiatives on the JVM

Value types, “minimal” and “full” (Project Valhalla)

Better generics (Project Valhalla “template classes”)

Arrays 2.0 (on hold pending better generics)

Vector API, lambda cracking (Project Panama)

Native data and code integration (Project Panama)

scaling: startup/footprint/latency/throughput (CDS, AOT, …)

JVM refresh: tech. debt cleanse, indy/condy, nest mates, …

Java-on-Java (Graal adoption, Project Metropolis, enabled by Jigsaw)

Post-threading: fibers, coroutines (Project Loom)

De-racing: confinement and immutability (mostly after values)

Six Trends to Watch (in the JVM technology)

1. Java hosts more of itself – Metropolis
2. Primitives get classier – Valhalla *value types & generics*
3. Language support gets richer – Amber / Valhalla
4. Hardware access gets faster – Panama
5. Concurrency gets more granular – Loom
6. Scale gets bigger *(…more scalier?)* – Shenandoah, ZGC

All this happens in the OpenJDK, more quickly, more collaboratively.

1. Java hosts more of itself

Metropolis

Graal JIT
&
Java-on-Java

Graal: code generator to the Java ecosystem

- Mature, robust technology
- Deep investment in research and development
- Decade-scale calendar, century-scale staffing (thanks JKU, Labs)
- Used in OpenJDK 9 for AOT technology
- Candidate for replacing C2 JIT — Metropolis
- Used with interpreters and scripting languages (with Truffle)

Project Metropolis: City of Tomorrow

- Experimental clone of JDK 11 (*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.
- 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!*

Rough roadmap

- Status: OpenJDK project launched 10/2017; still booting repos etc.
- Experiments in replacing HotSpot C++ code with Java code
- Key technology: Graal (used successfully in Java 9 AOT engine)
- Key technology: “Native” or “substrate” compilation mode for Java
- Grand goal is finding C2’s successor technology
- Additional goals: Replace other C++ by Java

2. Primitives get classier

Valhalla

Value types
&
Template classes

Big Idea: Value types — Project Valhalla

- Goal: “heal the rift” (“caulk the seam”) between classes and primitives
- Distinguish new ***value classes*** from legacy ***object classes***
- Value proposition: *“Codes like a class, works like an int.”*
- Depends on parametric polymorphism (any-vars not limited to objects)
  - 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 (does not require) pervasive changes throughout the stack
  - Comparable to the impact of generics or lambdas
- Extends the benefits of class and interface encapsulation to all data.

codes like a class, works like an int (primitive)

- if you heal the rift, both sides get stronger
- primitives are “really” restricted classes
  - object classes are “really” another kind of restricted class
  - interfaces can bind both together (abstract over both)
- this includes generics: `List<byte>` (*which is not* `List<Byte>`)

some theory about value types

- JVM is *encouraged* to flatten them (just like an `int` in an `int[]` array)
- …and therefore must be non-nullable
- identity-free (no data from acmp, monitor, Object.wait, etc.)
- …and therefore freely copyable (in register, on stack, in TLS, on heap)
- flat when stored in fields or arrays, buffered on JVM stack and in locals
- rebuffering (splitting) and de-duplication (merging) are unobservable
- can implement any interface (rebuffering on heap as needed)
- always “implements” the “interface” of java.lang.Object
- …and therefore is interoperable with today’s generics

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.

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 value instances.
  - Then everything boils down to value instances and object instances
  - Top it off with Object and interfaces for generic code.

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
  - …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.

3. Language support gets richer

Amber

better encapsulation,
enhanced constants,
bootstrap methods everywhere

JVM = language implementor's toolkit

- Nestmates, sealing (smaller trust boundaries)
- Intrinsics for direct programming of class-file contents
- The "Last Two" constants (bits, groups)
- (and then we refactor the CP for templates, oh well)
- Bootstrap methods everywhere: indy, condy, bridge-o-matic
- Frozen arrays, frozen objects

… All this enables a new generation of medium-sized Java language features, such as records and pattern matching.

4. Hardware access gets faster

Panama

FFI / FDI
&
Vector API

Project Panama — what comes after JNI

http://cr.openjdk.java.net/~jrose/pres/201708-PanamaExperts.pdf

- 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 intrinsic mechanisms.
- *Better use of native resources, data structures, and libraries.*

codes like Java, works like native

- JNI that stays in the background
- roughly homomorphic code shapes (native concepts ***surfaced*** to Java)
- the ***Binder pattern*** used for ANSI C, persistence, protocols, …
- safe, fast, flexible Java references to ***native memory*** (confined usage)
- direct contact with ***hardware*** (like C but better, stronger, safer)

Vector API

- What we have:
  - Excellent pathfinding, prototype & demos
  - Various JIT approaches (converging on C2 intrinsics)
  - Multiple vector formats supported (x86 only, but hey…)
  - C2 intrinsification of vector ops (ongoing)
- What we don’t have:
  - Ease of use, including natural lane-wise expressions
  - Polymorphism across vector shapes ***in a single loop***
  - Other similar platforms (SPARC, SVE)
  - A story for GPUs

other possible incoming

- Lambda cracking experiments (token codes, probably)
  - (token code = Forth-like concatenative single-stack IR in BSM args)
  - http://cr.openjdk.java.net/~jrose/panama/token-codes.html
- Layout language upgrades?
  - http://cr.openjdk.java.net/~jrose/panama/minimal-ldl.html
- Persistent memory?
- GPU attach?

More Panama references

http://mail.openjdk.java.net/pipermail/panama-dev/

http://cr.openjdk.java.net/~jrose/panama/

using-interfaces.html, minimal-ldl.html, metadata.html, cppapi.cpp.txt, token-codes.html

http://cr.openjdk.java.net/~vlivanov/panama/

linkToNative, code snippets, long4_on_avx

http://cr.openjdk.java.net/~mikael/webrevs/panama/

http://cr.openjdk.java.net/~psandoz/panama/

http://j9java.github.io/panama-docs/ IBM LDL design notes

5. Concurrency gets more granular

Loom

Fibers
&
Continuations

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

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
- Recent 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)

Fibers & Continuations for the JVM

- Ron Pressler started Project Loom in 2017
- Is adapting his excellent above-the-JVM work, Quasar
- Continuations are ***delimited*** not open-ended (good structure)
- Can be used for faster threads and coroutines
  - May also be used future language features (generators, …)
- We will naturalize this deep into the Java stack (APIs, JVM, JIT)
  - Key idea: small h-frames which execute off the system stack

6. Scale gets scalier

Shenandoah, ZGC
Loom
AppCDS, AOT

…And may all your Latencies be Low

- Project Shenandoah at Red Hat creates new pointer-based barriers
- Project ZGC (previously closed at Oracle) uses bit-based barriers
- Too much of a good thing? Not at all – we're a community here.
- Short term tasks: Lots of refactoring into common source base
  - Enables co-existence – and borrowing ideas, cross-pollination
- Long term result: Unified LL-GC tech. for OpenJDK
  - A curated toolkit of algorithms, easy experimentation & enhancement

Fibers for throughput relief

- Continuations are small heap-allocated objects
- Fibers are Thread-like objects build on top of them
- …therefore Fibers can number in millions
- …which outperforms Thread-level scaling

Sharing and provisioning for container scale-out

- AOT, AppCDS, SVM, and other startup/footprint work is ongoing

Less racy data for safe cross-thread sharing

- Frozen arrays and objects, with usual benefits of *safe publication*
- Crazy idea on the whiteboard: Stateful but *always-confined* objects
- An always-confined object can be safely frozen and then published
- It can also be queued (in a no-access state) and handed off
- Requires some deep cuts to Java objects
- See http://cr.openjdk.java.net/~jrose/pres/201707-Confinement.pdf

JMM race conditions are serious architectural debt – let's pay it down.

Impl Note: JVM can test confinement efficiently

- Each object has a header, which is currently under-utilized
- The header of an array includes the array object's length
  - checks are made efficient by constant folding, hoisting out of loops
- Idea: The header of *any* confined object can carry access bits
  - new states: read-only globally, read/write by one thread, no access
  - also efficiently checkable by constant folding, etc.
- (There is probably some connection with GC barriers here.)

That's a *lot* of future!

(Let's go there as a community.)