

New bytecodes, new objects

Keeping the JVM useful for 20 more years.


John Rose, Java VM Architect

JVM Language Summit, Santa Clara, August 2015

<http://cr.openjdk.java.net/~jrose/pres/>

201508-JVMComplexity.pdf

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.



Purposes of today's talk

- Speak about JVM ISA architecture—challenges and futures
- Look again at the basics of what the JVM does
- Point out some “sharp corners” in the current JVM design
- Sketch some fixes and future directions
- See also this year's JFokus talk “JVM implementation challenges”
<http://cr.openjdk.java.net/~jrose/pres/201502-JVMChallenges.pdf>

Last time we spoke...

The JVM:
a language toolkit





JVM Vision, 2014

(flash back to last year)

- Let's find some more pain points, where JVM semantics...
 - ... align too rigidly with Java language semantics,
 - ... fail to align closely with modern hardware,
 - ... or impose excessive costs in some other way.
- It appears we can relieve major pain points.
 - Improve simplicity and performance for new users
 - Retain compatibility and performance for present users



VM DESIGN



What's in a JVM?

DATA
& CODE
SAFE
& PORTABLE



What's in a JVM?

Data & code, safe & portable

- Data structures connected by managed pointers, dynamically typed
 - Computation with primitives and objects (methods, classes, interfaces)
- (Byte-)code that runs fast (hardware speed) without preprocessing
 - Name binding and optimization are deferred; lazy load and lazy link
- Safe, type-enforcing, robust, secure
 - Limits damage caused by error or malice, bug-resistant
- Portable, architecture-neutral, multiprocessing, large memory
 - Keeps pace with hardware technologies, grows with data paths & memory



Clever JVM moves

- Data: GC, uniform reflectability, primitives with optional boxing
- Code: JIT compilation, mixed-mode execution; profiling, deoptimization
- Safe: Redundant checks (verifier); abnormalities throw exceptions
- Portable: Clear specifications, hidden details; long-term compatibility

- The best moves are invisible, or can safely be neglected by the user.

Result: A simple user experience despite the complex technology.



Reality check

(or, how do we know when we win?)

- Data: Easy to understand, flexible, regular, compact (in memory)
- Code: Efficiently encodes source code meaning, runs fast & flexible
- Safe: Doesn't crash, easy to analyze, hard to crack, no sharp edges
- Portable: Gets the best out of every major CPU and system



Reality check

(searching and fearless inventory, anyone?)

- Data? Diffuse (too many indirections); racy; weak genericity (hard to tune)
- Code? Compilation unit size is too small; 16-bit limits; Java-specific
- Safe? JVM architecture is complex; HotSpot is complex (~1M LOC)
- Portable? Undersized arrays and primitives; dinosaur threads

“By seeking and blundering we learn.” — Goethe

τὸν πάθει μάθος θέντα — “Suffering makes learning.” — Aeschylus

JVM pain points (from “Evolving the JVM”, [JVMLS 2014](#))

Pain Point	Tools & Workarounds	Upgrade Possibilities
Names (method, type)	mangling to Java identifiers	unicode IDs ✓1.5/JSR-202, structured names
Invocation (mode, linkage)	reflection, intf. adapters	indy/MH/CS ✓1.7/JSR-292, tail-calls, basic blocks
Type definition	static gen., class loaders	specialization, value types
Application loading	JARs & classes, JIT compiler	Jigsaw, AOT compilation
Concurrency	threads, synchronized	Streams ✓1.8/JSR-335, Sumatra (GPU), fibers
(Im-)Mutability	final fields, array encap.	VarHandles, JMM, frozen data
Data layout	objects, arrays	Arrays 2.0, value types, FFI
Native code libraries	JNI	Panama



BYTECODE COMPLEXITY



Regarding complexity...

Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary.

Scheme demonstrates that a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a practical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today.

– *Revised Report on the Algorithmic Language Scheme*



VMs too

Virtual machines Programming languages should be designed ... by removing the weaknesses and restrictions that make additional features appear necessary.

An ideal VM Scheme demonstrates that a very small number of rules for **loading programs** ~~forming expressions~~ with no restrictions on how they are composed, suffice to form a practical and efficient programming **platform language** that is flexible enough to support most of the major programming paradigms in use today.

– *We're not building ideal VMs here, but we can improve*



Scheme fundamental operations

- Names and constants: *(the usual)*
- Function Call: ($\text{<expr}_{\text{func}} \text{<expr}_{\text{arg}} \dots$)
- Function: ($\lambda (\text{<name}_{\text{arg}} \dots) \text{<expr}_{\text{body}} \dots$)
- Definition: ($\text{define } \text{<name}_{\text{var}} \text{<expr>}$)
- Assignment: ($\text{set! } \text{<name}_{\text{var}} \text{<expr>}$)
- Control flow: ($\text{if } \text{<expr}_{\text{pred}} \text{<expr}_{\text{t}} \text{<expr}_{\text{f}} \text{>}$)
or: $\left(\left(\text{select } \text{<expr}_{\text{pred}} \right) \left(\lambda () \text{<expr}_{\text{t}} \right) \left(\lambda () \text{<expr}_{\text{f}} \right) \right) \right)$
... everything else is just a library function or syntax sugar



Some Scheme VM bytecodes (Rose/esh)

L:make-lambda	C/J:plain/tail call	S:get stack	H:set stack
A:take-arguments	R: return	X:get lexical	Y:set lexical
\$.reserve-stack	B/T/F: branch*	V:get global	W:set global
E/U:enter/exit scope	P: pop	Q/G:get literal	I:init global



Smalltalk fundamental operations

- Names and constants: *(the usual)*
- Message Send: $\langle \text{expr}_{\text{obj}} \rangle \text{ message: } \langle \text{expr}_{\text{arg}} \rangle \dots$
- Function: $[: \langle \text{name}_{\text{arg}} \rangle \dots \mid \langle \text{expr}_{\text{body}} \rangle \dots]$
- Function Return: $\wedge \langle \text{name}_{\text{res}} \rangle$
- Definition: $\mid \langle \text{name}_{\text{var}} \rangle \mid$
- Assignment: $\langle \text{name}_{\text{var}} \rangle := \langle \text{expr} \rangle$
- ... everything else is just a library class or method*
- Control flow: $\langle \text{expr}_{\text{pred}} \rangle \text{ ifTrue: } [\langle \text{expr}_{\text{t}} \rangle] \text{ ifFalse: } [\langle \text{expr}_{\text{f}} \rangle]$



Some Smalltalk VM bytecodes (Bonzini/GNU)

PUSH_INSTANCE_VAR	STORE_INSTANCE_VAR	SEND*
PUSH_LOCAL	STORE_LOCAL	SEND_SUPER*
PUSH_OUTER_LOCAL	STORE_OUTER_LOCAL	JUMP*
PUSH_GLOBAL	STORE_GLOBAL	POP_JUMP (T/F)
PUSH_CONST	POP_STACK_TOP	LINE_NUMBER...
PUSH_SELF	DUP_STACK_TOP	MAKE_BLOCK...
PUSH_SPECIAL	POP_INTO...STACK_TOP	EXIT_THREAD
PUSH_INTEGER	*RETURN_STACK_TOP	EXT_BYTE (wide)



Java fundamental operations

- Names and constants: *(the usual, plus qualified names)*
- Method Call: `<nameobj> . <name>(<exprarg>...)`
- Field Reference: `<nameobj> . <name>`
- Function: `(<namearg>...) -> <exprbody>`
- Definition: `<Type> <namevar> = <expr>;`
- Assignment: `<namevar> = <expr>;`
- Object creation: `new <Type> ... (etc.)`
- Arithmetic: `<expr> + <expr> (etc.)`
- Array Access: `<expr> [<expr>] = <expr>;`
- *(Lots more...)*



The Java VM bytecodes

(local data movement)

aload	astore	areturn ¹	pop	aconst_null
iload	istore	ireturn ¹	pop2	iconst/ipush
lload	lstore	lreturn ¹	swap/dup...	lconst
fload	fstore	freturn ¹	dup2...	fconst
dload	dstore	dreturn ¹	ldc*	dconst

¹ *first part of return instruction sets outgoing value*



The Java VM bytecodes

(array management)

aaload	aastore	newarray
iaload	iastore	anewarray
laload	lastore	multianew.
faload	fastore	
daload	dastore	
baload	bastore	
caload	castore	
saload	sastore	arraylength



The Java VM bytecodes

(control flow and name references)

goto	ifCC	icmp/acmp ³
*return ²	if_TcmpCC ³	lcmp
athrow	if_[non]null	fcmp
jsr/ret (obs.)	*switch	dcmp*

² last part of any return instruction resumes control after an earlier invoke
³ if_icmpCC/if_acmpCC combine hypothetical acmp/icmp and ifCC

getfield	putfield	invokevirtual
getstatic	putstatic	invokestatic
ldc (Class)	new	invokespecial
checkcast/i'of	invokedynamic	invokeinterface



The Java VM bytecodes

(misc. and arithmetic)

monitorenter
monitorexit
wide*
nop
<i>(method entry)</i>

iadd	ladd
isub	lsub
imul	lmul
idiv	ldiv
irem	lrem
ineg	lneg
ishl	lshl
ishr	lshr
iushr	lushr
iand	land
ior	lor
ixor	lxor
iinc	

fadd
dadd
fsub
dsub
fmul
dmul
fdiv
ddiv
frem
drem
fneg
dneg

i2l
i2f
i2d
l2i
l2f
l2d
f2i
f2l
f2d
d2i
d2l
d2f
i2b
i2c



SIMPLER BYTECODE



Idea: Make [ailfd] prefixes optional

(*generic* local data movement)

- istore, iload, lload, lstore, fload, dload, aload → vload, vstore
- ireturn, lreturn, etc. → vreturn
- pop/pop2 → vpop (as necessary)
- Effect: Every local value has the syntax of an “any” value.
- **Preserve strong typing, but in a more regular syntax.**
- Parameterized types for “everyday” use.
- Naturally extends to new value types.



Details, please?

- Well, the details are messy and at present uncertain.
 - Take the letter “v” with several grains of salt. Maybe it’s really “a” or “u”.
- Multiple proposals and insights are on the table!
- Valhalla any-generics currently the most fruitful driving investigation.
- Caveat: *Not* doing Lisp/Smalltalk typeless vars. Static typing is good.



Simplified Java VM bytecodes

(*generic* local data movement)

vload	vstore	vreturn	pop	vldc_default
			pop2	
			swap/dup...	
			dup2...	
			ldc*	



But, let's keep the [ailfd] code points

- Existing [ailfd]load bytecodes are compressed notation for vload.
- iload *abbreviates* vload :I (*warning: don't take smiley syntax seriously*)
- Slot pairs? Only *legacy* uses of lload, dstore, ldc2_w, etc.
- **For arrays:** daload *abbreviates* vaload :D
- Further, vaload :D *abbreviates* invokeinterface Array.getElement
- Compare Scheme (IF A B C) as an *abbreviation* for a library function.



Idea: Consolidate array APIs to use interfaces

```
public interface Array<any X extends Ordinal, any E> {  
    X arrayLength();  
    E getElement(X n);  
    void setElement(X n, E e);  
  
    Array<X,E> clone();  
    Array<X,E> freeze();  
    Array<X,E> slice(X from, X to); // creates shared view  
  
    Array<X,E> copyOf(X length); // Arrays.copyOf  
    Array<X,E> copyOfRange(X from, X to); // Arrays.copyOfRange  
    boolean arrayEquals(Array<E> that); // Arrays.equals  
    int arrayHashCode(); // Arrays.hashCode  
    String arrayToString(); // Arrays.toString
```



Types for Arrays 2.0

- “Classic” arrays are given new interface supertypes
 - `Array<int,E>` and/or `ClassicArray<E>`
- More difficult: The hardwired VM type `[I` becomes an interface.
 - All array operations become polymorphic (`aaload`, `iastore`, `arraylength`)
 - Will allow existing array code to become generic (only one copy!!)
 - Will allow libraries to plug in new array implementations
 - Will allow legacy APIs which mention arrays to be retrofitted
- Risk: The JVM needs to continue to optimize arrays vigorously
 - (Cue current experiments by Paul Sandoz and Roland Westrelin.)



Example code (from POC work)

```
String[] frz = (String[]) av.freeze();  
if (!frz.arrayEquals(av)) throw new AssertionError();  
  
String x = av.getElement(0);  
System.out.println("x = "+x);  
  
//List<String> ls = av.asList();  
//System.out.println("ls = "+ls);  
  
int[] ints = { 1, 2, 3 };  
System.out.println("ints.string = "+ints.arrayToString());
```




Simplified Java VM bytecodes

(generifed array management – best case)

vaload	vastore	
		invokestatic
		invokeinterface



Idea: Give API points to all primitive functions

(generified primitives allow generic types like `Array<Indexlike,double>`)

- `ladd` → `Long.add(JJ)J`
- `l2c` → `Long.toChar(JJ)C`
- `fcmpg` → `Float.compareTo(FFZ)I`

- `interface Integral<X> { X add(X x); X sub(X x); X neg(); ... }`
- `interface Bitwise<X> { X and(X x); X xor(X x); X shl(int n); ... }`
- *Serious* design problems here of splitting vs. lumping, axiomatization.
- Benefit: We can add new numeric types and write generic numerics.



Simplified Java VM bytecodes

(and maybe also generic bytecodes, as abbreviations)

vadd	
vsub	
...	
	v2v
	...
vxor	



Simplified user model!

- Try to consolidate legacy primitives into Valhalla value types.
- Net decrease in effective complexity, even if there are dark corners.
- The legacy bytecodes can be abbreviations, not specially privileged.
- Caveat: “int” and “long” will always be special, as data indexes.
 - Similar points about other primitives, but it’s a slippery slope.



Idea: Use indy for nominal instructions

- Any “nominal” instruction referring to a class member can be “indified”
 - {get,put}{field,static}, invoke{static,special,virtual,interface}
- `invokeinterface C.m` → `invokedynamic BSM0, Constant(C.m)`
- Existing invoke instructions continue to be useful abbreviations.
- *So, this is a heavy step, but we can keep it in reserve.*
- Can help us add new invocation modes without new bytecodes.
 - Example: Bridge from Point A to Point B with flexible signature matching



Idea: Encapsulate extra-“special” invocations

The “dance” of `new` + `invokespecial` `<init>`

- The idiom `new/invokespecial<init>` must become a factory call
- Deprecate/encapsulate naked “new” opcodes.
- Allows instances, generics, arrays to be created with uniform syntax
- `File f = invokestatic File.<new>(path);`
- `ArrayList<int> ints = invokestatic ArrayList.<new>(int.class);`
- `String[] sa = invokestatic Object[].<new>(String[].class, 25);`
- Leads towards a saner story for serialization: *factory injection*.



BESIDES BYTECODES



Problem: classes are not modules

- Class files are too small (JARs, packages are more natural units)
- Class files are also too large (sometimes we load just one method)
- Class files are also too rigid (want to load cold parts of a class later)
- Scary thought: We might rethink class file format

(Could be more bikeshed colors here than a Portland Tie-Dye Festival.)



Problem: unstructured static dependencies

- `getstatic` / `putstatic` / `invokestatic` – OK until the first side effect
- Actually, not OK, if you also need to configure dependencies.
- HotSpot implements `getstatic` / `putstatic` as fields on `Class` objects
- So does SmallTalk (in the official standard)
- Java could move this way too:
 - `getstatic C.x` → `ldc C.class; getfield C.x`



Encapsulate “static” initializations?

- JVM hardwires behavior of <clinit> (Java static code blocks)
- Really useful language & VM feature...
- Until the initialization order the VM chooses doesn't work...
- Or until some “static” side effect hits your code the wrong way.
- A better framework: Layered, multi-phase initialization?
 - Should be able to execute early initialization at well-chosen early points.
- Early points = { javac-time, jlink-time, <clinit>-time, BSM-time }



Problem: Java objects and data are too fluxy

- Everything is non-final by default.
- There is no “final” word on array contents.
- Even “final” variables can get new values jammed in.
 - Evil Reflection, serialization, Unsafe
- What’s an optimizer to do? (Cue discussion of MH inlining.)
- Why is my memory backplane so warm?



Problem: cache is scarce, memory is far away

what Moore's Law didn't tell you

- Rule #1: Cache lines should contain 50% of each bit (1/0)
 - E.g., if cache lines are 75% zeroes, your D\$ size is effectively halved
- Rule #2: Use a cache-line worth of data per random access
 - Chaining through a 8-byte pointer could waste 75% of a cache line fetch
- The ideal: Make sequential accesses to high-entropy payloads
- Today's JVM violates both rules by its heavy reliance on pointers
 - Arrays can help, but they often devolve to *array-of-pointer-to-payload*
- The JVM can't express one value containing another (except via pointer)



Problem: side effects are round-the-world trips

Even after data is compacted, fluxy data is expensive

- N writes to a variable can circle the (memory) world N times.
 - Memory should be used (mainly) for one-way communication.
- A (Java) object is (basically) a message in a cache line.
- In most cases, it is best to write once and share a frozen record.
 - Unexpected non-local write-to-read effects are usually bugs.



Constraint: Coloring inside the cache lines

- The CPU is good at processing cache lines
 - A dirty cache line is close to a single CPU
 - A clean cache line can transmit data to many CPUs
 - Neighboring cache lines are often prefetched together
 - Data structures which fit in cache lines win!
- ➔ Java working storage should fit neatly in consecutive cache lines.
- This means Java objects and arrays, plus values, boxes, etc.
- It also means activation records (esp. when we get past threads)



Solution: Pointer-free data structures

(examples)

- Valhalla will proudly provide `ArrayList<int>`
 - box-free value aggregates
- Larger primitives without boxes (CPU vectors as fields)
- Inlined arrays (Var-Handle of array of field)
 - B-tree nodes (finally!)
 - Fused strings (header plus variable tail of chars or bytes)



Solution: value-based objects, frozen arrays

- Give a natural and easy way to shut down future changes
 - On both objects and arrays.
- Object and array freezing allows the system to copy freely.
 - Larval construction phase, followed by freeze, then free publication.
- Caveat: Object identity is also a side effect (esp. synchronization)
 - Identity-free values (or value-base arrays) can be freely copy-optimized
- ➔ We need more value-based objects and other value-like boxes.



Problem: threads are passé

- Java distinguished itself 15 years ago with a big investment in threads
 - At that time threads were hard to program; memory models were unreliable
- Threads use enormous amounts of stack memory
 - Their built-in synchronization mechanisms are obsolete
 - They cannot do SIMD lockstep synchronization, as today's GPUs require
 - A thread cannot release its resources until its initial task is done
- Event-driven or reactive code decomposes into many concurrent tasks
 - Would be overkill to give each task a thread



Beyond threads: fibers, warps, events, reaction

- A “fiber” is the lively bit of a thread, minus the large control stack
 - Fibers run on host threads, treating them as interchangeable commodities
 - It is reasonable to fire up a million fibers and set them running
 - A fiber might be an array, an index, and a bit of code to run for that index
- A “warp” is a group of similarly-shaped fibers which advance together
 - A warp processing an array could have fibers differing only by array index
- Fibers can wait on events, without tying up thread resources
- Reactive programming combines fibers to process events



Beyond threads: VM support for fibers etc.

- Fibers work best when three features coincide:
 - It is easy to start a fiber from a **closure** (anonymous function plus data)
 - An unfinished fiber execution can be paused, yielding a **continuation**
 - A finished fiber invoke a chosen successor using a **tail-call**
- (**Bug:** Event handler frame dragging – un-dead predecessors.)
- All of these features require additional JVM work to run smoothly
- When a fiber is created, it should package its continuation into **heap-frames**
 - It should be possible to teach the JIT to run a heap-frame directly
- JNI needs “suspend/resume hooks” to work properly with fibers



Beyond threads: What's in an activation record?

- Activation record for a single method call
- ...plus its immediate callees (if they don't block)
- When active, rides on a thread.
 - But can dismount and remount a different thread.
- More subtly: An ongoing sequence of bytecode executions
 - A principal actor (as much as Threads) in the Java Memory Model
 - Dismount/remount has ordering requirements.
- If done right, we can have millions at a time. (Paging Ron Pressler!)



Beyond threads: varying the variables

- Java threads communicate by reading and writing heap variables
 - Complicated rules determine whether there are “races”
- Variables used for thread communication are very hard to tame
 - As threads go light, the taming becomes more complicated
- We need better design patterns so variables are safe to use
 - Frozen (im-mutable) variables are safe to use by any thread
 - Variables protected by a lock are confined to the locking thread
 - Changeable data structures should be proven race-safe
- We need fewer variables which are vulnerable to races



Keeping up with the vectors

- CPUs now operate on 256-bit (soon 512-bit) data
- JVM needs flat value types of those sizes (Valhalla value types!)
- We are experimenting *now* using existing frameworks.
 - For-loops on primitive arrays
- New experiments will use vector-in-a-box structures (VarHandle-like)
 - Memory-to-memory prototyping
 - ASM-like tricks under Method Handles and Var Handles



Consolidate vector APIs to use interfaces

```
interface CPUVector<anyE, S extends Vector.Shape<Vector<?, S>>> {
```

```
    Class<anyE> elementType();
```

```
    S shape();    int length();    int bitSize();
```

```
    anyE getElement(int index);
```

```
    Vector<anyE, S> putElement(int index, anyE x);
```

```
    Array<int,anyE> asArray(); // array access
```

```
    Vector<anyE, S>      mapElements(UnaryOperator<anyE> op);
```

```
    <anyT> Vector<anyT, S> mapElements(Function<anyE,anyT> op);
```

```
    Mask<S>             testElements(Predicate<anyE> op);
```

```
    ...
```



BIG PICTURE



What should the JVM look like in 20 years?

(eight not-so-modest goals)

- 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: Routine confinement/immutability, 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.



QUESTIONS?