Confinement in the VM
(or, How Boundaries Can Liberate)

John Rose, JVM Architect

JVM Language Summit, Santa Clara, July/August 2017
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.
Observation: Java abstractions are so worth it

ref: oracle.com/technetwork/java/index-136113.html (Gosling 1996)

- Good abstractions are costly,
  - in design, development, specification, optimization, IDE support,
  - and sometimes in programmer learning curve or performance.
- But bad abstractions are prohibitive,
  - in muddled logic, bugs, exploits, untrustworthy APIs, confused tools;
  - they usually require ongoing programmer therapy (stackoverflow).
- Java is an balanced set of deep tools and facilities which cost a little performance, but not too much, and pay the user back with interest.
  - Features include: classes, bytecode IR, static and dynamic types, safe managed pointers, threads. (The white paper list!)
The Java ecosystem slowly introduces new APIs and even paradigms, upleveling everyone’s abstraction game. It’s expensive but worth it.

Since the early days we have added more deep tools: profiling JIT, generics, annotations, invokedynamic, lambdas, modules, scripting

Java is not perfect, of course, but **Java is vibrantly growing**.

Confinement is one way to think about causal isolation, and it’s a good time to think *more* about causal isolation. It’s everywhere if we squint:

- The old: access control, heap integrity, synch., good libraries.
- The new: value types, specialization, native bridges, JLS upgrades.

Today’s suggestions: **freezing, racer arrest, lambda cracking**.
On abstraction and logic

- Facing some complex X, we **pull out** L, a true but simplified account.
  - X ⊨ L, L is true of X. (“animal” doesn’t abstract a rock. – Aquinas)
  - L is orderly, logical (a *logos*), often in math-language, even if X isn’t.
  - L gives our mind something to work on, without losing track of X.
- Abstraction of abstraction (see previous) is a 2500-year-old pastime.
  - What makes an account abstract? How does logic work, and why?
  - From Aristotle to Boethius, Leibniz to Gödel, logic allures us.
- **Formal** methods of reason took centuries of patient labor to elucidate (Aristotle quantifiers, al-Khwarizmi/Descartes algebra, Boole truth values, Cantor sets, Church/Turing machines, Gödel meta-math, Backus/Iverson programming languages – to name just a few.)
(digression on Raymond Smullyan)
ref: amazon.com/gp/product/0486492370

- This February past a great 20th-century logician died at 97. His witty, well crafted tours through the inner landscapes of logic bring us insight and joy, a half-century and counting.

- Smullyan’s presentations are consistently lucid, economical, and beautiful, like elegant software code.

- I recommend his books on Logic and Set Theory (incl. temporal logic), and his puzzles.

R.I.P.
Abstraction in computing

- An abstraction is a part of a program which is worth looking at by itself.
  - Could be a macro, function, type, module, or protocol.
  - Often accompanied by a specification (the *logos* of the abstraction).
- Abstractions, being separable from the surrounding system:
  - Can and should be understood apart from any particular use
  - Are more powerful as they have a wider variety of applications
  - Are more reliable as they have clearer specifications
- In a dynamical formalism (software), causation is a pervasive relation.
  - Abstractions must **account for** their in-causes and their out-causes.
  - This is a good general notion of **confinement** as a design heuristic.
Causes vs. confinement

- A confined program fragment is embedded in a sea of causes.
  - And it has a boundary, across which causes come in and go out.
- Complex, ill-specified, or unpredictable causes blur the boundary.
  - So, remove irrelevant or disruptive causes, and account for the rest.
- **Incoming** causes are the rest of the system signaling into the boundary.
  - Arguments to this method, return value (or exception) from that callee, a write into this variable, a read out of that variable.
- **Outgoing** causes (the dual) are when the fragment acts on the system.
  - Return (or exception or side effect) from this method or expression.
- Some (composite) causes go both ways: CAS, synchronization.
- **Local** causes stay completely inside an abstraction: local state, etc.
Confinement

... a magic circle

... my code ...
Confinement

... with an inside, an outside, and paths across the boundary

... other code or state ...

... my code ...

... other code or state ...

causality

causality

boundary

confinement
A small puzzler
what does this do?

- A fragment from some thread’s execution.
- Reads some previously written ‘x’.
- Clips negative values to zero, stores back the zero.

```java
if (x < 0)
    x = 0;
```

- What’s complicated about that?
A small puzzler
here’s one thing it might do

```
void g() {
    int x = f();
    if (x < 0)
        x = 0;
    return x;
}
```
Complications

- The variable might be a local, which limits its causes and effects.
- The variable might be in the heap,
  - as a field ‘this.x’, a local static ‘Me.x’, or an import ‘C.x’,
  - and it may independently be marked ‘volatile’ or left plain.
- (There is also stuff like @Contended, @Stable, etc.)
private int x;
void g() {
    if (x < 0)
        x = 0;
}

A small puzzler
or, it might go off-thread
More complications

- If it’s static, the first “touch” might trigger a lengthy subroutine call.
- If separately recompiled to ‘static final’, there will be a LinkageError.
- Unless your program is carefully written, there can be races.
  - If there is a race, the JMM lets *any read* see *any write*.
  - (More on this sinister statement in a moment.)
A small puzzler

or it might go into a class initializer, which *might* be off-thread

```java
import static C.x;
void g() {
    if (x < 0)
        x = 0;
}
```
A small puzzler
and there are races out there too

```java
private int x;
void g() {
    if (x < 0)
        x = 0;
}
```
Confinement keeps complicated outsides, out
...so we can enjoy undisturbed contemplation of our abstraction

...my code ...

causality
confinement boundary

...off-thread
state and actions ...

concurrent
causality

...off-thread
state and actions ...

causality
concurrent
causality

Confinement is only as good as the boundary abstraction fails when rogue causes get in or out

... UNEXPECTED off-thread state and actions ...

my code ...

... off-thread state and actions ...

boundary with a gap

concurrent causality

that we were not accounting for

known causality

confinement

known causality
Java Memory Model  *(suit up; we’re going in)*

- Just a review, heh.
- Everybody in this room knows this stuff already, right?
- We use mutability all the time, and JMM = semantics of mutability.

- Full disclosure: I learn something new every time I study the JMM. Disquieting adumbrations trouble my callow credence in Java. Maybe even last week.
Java Memory Model – the basics

ref: cs.umd.edu/~pugh/java/memoryModel/

- Threads run concurrently, mostly independent of each other.
  - Life is great if your data structure is only touched by one thread.
- Threads communicate (99%) through “actions” on heap variables.
  - You exclude races with “final”, “synchronized”, or “volatile”.
  - With enough keywords you win the “Sequential Consistency” prize.
- Otherwise, your code may race, but must not leave the track or crash.
  - Key requirement: Every read returns something “previously” written.
  - “Previous:” No causal cycles (for some very technical “causality”).
- (Everybody hopes that) JITs can still do all their crazy optimizations.
Java Memory Model – still the easy parts

- There is a partial order on All Events called “happens-before”, $x \rightarrow y$.
  - For any two events exactly one is true: $x \rightarrow y$, or $y \rightarrow x$, or $x \leftrightarrow y$.
  - It’s also transitive, like any good order. And irreflexive: $x \not\leftrightarrow x$.
- Each thread is, from its own POV, fully $\rightarrow$ boringly $\rightarrow$ sequential.
  - So intra-thread events march like $p_1 \rightarrow p_2 \rightarrow p_3 \rightarrow \ldots$
- Monitor enter/exit (just like volatile reads/writes) are linked by $\rightarrow$
  - These “sync-with” $\rightarrow$ edges zig-zag between threads and memory.
  - This is the part where you and I wish we could stop here, roll credits, and declare victory.
- Non-volatile reads and writes do not create $\rightarrow$ links; they rely on them.
A not-completely-misleading graphic
So why aren’t we already insane?

- **Answer #1**: If my program is “DRF”, I win “SC”.
  - “DRF” = data-race free, “SC” = sequential consistency
  - I wish my code to be DRF. Maybe it is!
- **Answer #2**: The implementation of the JMM I use is restricted.
  - I rely on hardware memory that is a Model Citizen (today).
  - Or, I rely on timid JITs which don’t optimize fully.
- **Answer #3**: Lady Luck likes me! (Not looking back.)
  - That coder who fell into the snake pit was a bad coder.
- **Answer #4**: I know some good incantations for confinement.
  - I hide pointers and copy defensively. I read stackoverflow a lot.
Sequential Consistency — consolation prize
ref: Lamport 1979, “How to Make a Multiprocessor Computer…”

- The easiest concurrent memory model, defined by Leslie Lamport.
  - He also coined “happens-before” →, and is the brains behind TLA+.
  - As if all your thread traces were shuffled together like a deck of cards.
  - As if memory performs only one operation at a time, w/o caches.
  - Each read connects with the unique most recent™ write.
- You still have to guess which interleaving gets chosen by the system.
  - Lots of room for complexity but no OOTA or causal wormholes.
- SC hazards: Reads might look in cache and see data from old writes.
  - Most HW is polite about this; JITs have to avoid the SW equivalent.
  - (Similar point about tricky store buffers and queues.)
We have some very special prizes for racers

ref: shipilev.net/blog/2014/jmm-pragmatics/

- For any given variable V, a read r of V can see any write to V…
- …Except those specifically forbidden by the “happens-before” relation.
  - What’s forbidden is w0 and w2 in w0 → w1 → r → w2. That’s all.
- There is no particular preference for w1 if some w007 is lurking.
  - You have to exclude all racing w007’s if you want to be sure of w1.
- For racy programs, this leaves a LOT of room to get off track.
  - In practice, implementations avoid Crazy Stuff. (just keep on whistling)
- Sprinkle liberally with “synchronized”, “volatile”, and “private”.
  - Without class-based encapsulation we couldn’t sleep nights.
A graphic with an extradimensional tentacle

Thread A
- `aload`
- `iconst 3`
- `putfield v`
- `invoke`
- `mon-enter`
- `getfield w`
- `iconst 2`
- `iadd`
- `putfield w`
- `mon-exit`
- `invoke`...

Memory
- `v = 0`
- `w = 0`
- `v = 3`
- `m/45`
- `v = 4`
- `w = 2`
- `m/46`

Thread B
- `getfield w`
- `getfield v`
- `mon-enter`
- `getfield w`
- ...

Thread C
- `invoke`
- `getfield q`
- `mon-enter`...

Thread D
- `iconst 3`
- `checkcast`
- `aload 4`...

General drift of causality

not racy

racy!

not racy

not racy

...still racy!!
A worst-case JMM scenario
Back-to-back reads

- Two back-to-back reads *might* be optimized by the JIT to return the same value.
- But the JMM treats these reads independently. If there is a race on `a[0]`, any read can pick up (almost) any write.
- Therefore, either `r1` or `r2` *might* be zero. (E.g., after de-opt.)
- Fix this today with `volatile`.
- Fix it tomorrow with fences?

```java
static volatile int[] a;

// Thread T1:
a = new int[] { 42 };

// Thread T2:
boolean test() {
    int r1 = a[0];
    int r2 = a[0];
    return r1 == r2;
} // must return true?
```
A worst-case JMM scenario
Then the flying monkeys attack

- An attacker can unilaterally set up a race on the fields of any exposed reference.
- The JMM excludes races only if both threads agree to handshake. (Nope, it’s not all about fences; JMM has no fences.)
- Especially dangerous when coupled with varargs. (We need frozen varargs!)

// Thread T007:
for (int i=0;;) {
    a[0] = ++i;   //race!
}

// Thread T2:
void test() {
    sanityCheck(a[0]);
    trustAndExecute(a[0]);
}
Eldritch horror at the mutations of madness

Sinuous threads tangle our work,
Undermining causality itself;
It cannot be fenced or killed, yet
Ceaseless vigilance can confine it;
Perhaps it can be frozen.
A JMM sweet spot
You only get one kind of fence pattern: acquire/release

- The only cross-thread ordering that matters to the JMM is volatile (which is about the same as synchronized).
- So, writing a volatile releases all the good stuff you just did.
- And reading a volatile acquires that stuff from another thread.

```java
static volatile int[] a;

// Thread T1:
a = new int[]{ 42 });

// Thread T2:
void test() {
    int r1 = a[0];
    int r2 = a[0];
    assert r1 == r2;
}
```
The red lines, only, give us “release → acquire”

Exercise: Why can’t the last “getfield w” see the initial “w = 0”? 
A moral of this story: Uncertainty is expensive
ref: https://youtu.be/-4Yp3j_jk8Q (Lamport, “Thinking Above the Code”)

- Being close to the HW feels great, as long as you and the HW agree.
- But in the end your software has to serve your customers, not the HW.
  - If you can’t meet a spec., you are racing nowhere fast.
- “A spec. is more important, when you don't know what the program is supposed to do… It has to do something, and you have to decide what it should do.” (Lamport, “Thinking Above the Code”, 2014)
- Recent changes in llvm make indeterminate C++ “act out” more often.
  - This makes C++ friendly to optimizers, but legalistic to coders.
- Java has always gone a different path: Coders first, then optimizers.
  - As CPU counts and memories scale up, mutability has more risk.
The bad news, in a nutshell

- Unless the writer releases and the reader acquires, there are races.
- If there are races, any reader can be matched with any writer.
  - You probably aren’t OK with that, even if your code seems to work.
What about benign races?

- There are benign races, but they are tricky to set up correctly.
  - (And if you make a mistake, you won’t find out until after you ship.)
- The best benign race, by far, is when a variable is written exactly once.
  - Then the readers are forced to read the value you wrote.
- This ONLY works for final variables (via a special JMM rule).
  - For other variables, the initial zero/null/false counts as a 2nd write.
- The second best benign race is when you are OK with all values.
  - Perhaps the system is improving itself: v = good; v = better; v = best
  - Be ready to see “v==good” even after somebody sees “v==best”.
Can we live on Sequential Consistency alone?

- Most Java code wouldn’t change if SC were imposed over the JMM.
- But it would get N% slower. (Some claim the cost is worth it. I don’t.)
  - It would be as if every variable were marked volatile or final.
  - Bugs would jump out of code that accidentally relies on races.
- Super-coders who need races would use VarHandles or some such.
- Yes, races are often the best answer, if you are a library coder.
  - If every possible read-value is good enough, you trust in ¬OOTA.
- But this is not a cure-all: Mutability is hard to manage even under SC.

ref: microsoft.com/en-us/research/publication/the-silently-shifting-semicolon/
And now a word from our API developers…

https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html#SideEffects

- Most stream operations accept parameters that describe user-specified behavior, which are often lambda expressions. To preserve correct behavior, these behavioral parameters must be **non-interfering**, and in most cases must be **stateless**…
- Preventing interference means ensuring that the data source is not **modified** at all during the execution of the stream pipeline…
- Modifying a stream’s data source during execution of a stream pipeline can cause **exceptions**, incorrect answers, or **nonconformant behavior**…
- Stream pipeline results may be **nondeterministic** or incorrect if the behavioral parameters to the stream operations are stateful.
- A stateful lambda (or other object implementing the appropriate functional interface) is one whose result **depends on any state** which might change during the execution of the stream pipeline…
- **Side-effects** in behavioral parameters to stream operations are, in general, discouraged,
  as they can often lead to unwitting **violations** of the statelessness requirement,
  as well as other **thread-safety hazards**.
- If the behavioral parameters do have side-effects, unless explicitly stated, there are no guarantees as to the **visibility** of those side-effects to other threads,
  nor are there any guarantees that different operations on the "same" element within the same stream pipeline are executed in the same thread. Further, the ordering of those effects may be **surprising**.
- Even when a pipeline is constrained to produce a result that is consistent with the encounter order of the stream source, **no guarantees** are made as to the order in which the mapper function is applied to individual elements, or in what thread any behavioral parameter is executed for a given element.
Sequential Consistency isn’t even the main issue

- Most of the complexity of streams comes from unconfined effects.
  - (With the right kind of squint.)
- “Stateless” lambdas have the right restrictions on causality links.
- A stateless lambda cannot be the cause of “side” effects.
  - System.out.println(boo), sourceList.add(mostStuff)
- A stateless lambda cannot be effected by side causes.
  - Should not read state which changes (or, worse, races).
- There is no way to verify this. It is a snake pit in our playground.
Digression: Toughen up and step over the snake-pit.
A smart guy like you can figure it out

When controversies arise, let us calculate.

We must know — we shall know!

how soon can you get it to me?

Sufficiet enim sedere ad abacos, et ... dicere: calculemus.

kein Ignorabimus... Wir müssen wissen — wir werden wissen.

P.H.B.
…except some worthy problems are intractable

* in any given formal system

Not all proofs can be automated.*

Not all automatons can be proven.*

* to do anything particular, like terminate
Not just a matter of training/talent/discipline.

- Gödel: Not all proofs can be automated. (Within a fixed system.)
- Turing: Not all automatons can be proven to work. (Statically, offline.)
- Rice: Anything worth proving about software isn’t always provable. 
  \[ \forall \text{ pl } \in \text{ TCPLs}, \text{ S } \in \text{ SPs(pl)}, \exists \text{ P } \in \text{ Progs(pl)}, \nexists \text{ D} \ . \text{ D decides S(P)} \ ]

  [https://youtu.be/dWdy_AngDp0?t=1m6s](https://youtu.be/dWdy_AngDp0?t=1m6s)

- Pairs very well with Ron’s 2017 followup, about Lamport’s TLA+. 
Inscrutability vs. Lines of Code

$x = \text{specification complexity in LOC}; y=10 \Rightarrow \text{"provably unknowable"}$

**simple code**

- 2+2=4
- bugfix
- homework
- 4Color/orig
- BB-5
- BB-4
- Goldbach

**unaccountable behavior**

- UTM*
- BB-7918
- HotSpot*
- *multiple inputs
- RH

4Color/proof

[43] Copyright © 2017, Oracle and/or its affiliates. All rights reserved.
Some more details...

<table>
<thead>
<tr>
<th>Name</th>
<th>Complexity</th>
<th>Inscrutability</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2+2=4</td>
<td>1</td>
<td>1</td>
<td>self-evident fact</td>
</tr>
<tr>
<td>bugfix</td>
<td>10</td>
<td>2</td>
<td>find and fix a small bug</td>
</tr>
<tr>
<td>homework</td>
<td>100</td>
<td>3</td>
<td>do your math homework</td>
</tr>
<tr>
<td>4Color/proof</td>
<td>60000</td>
<td>4</td>
<td>review modern proof of 4C using Coq, Gonthier 2005</td>
</tr>
<tr>
<td>BB-4</td>
<td>1</td>
<td>5</td>
<td>all 4-state TMs analyzed; max output = 13 ones, Brady 1975</td>
</tr>
<tr>
<td>4Color/orig</td>
<td>50</td>
<td>6</td>
<td>four color conjecture, Guthrie, 1852 (before Appel/Haken)</td>
</tr>
<tr>
<td>Hotspot*</td>
<td>533615</td>
<td>6</td>
<td>HotSpot virtual machine, proven “good enough”, many inputs</td>
</tr>
<tr>
<td>BB-5</td>
<td>2</td>
<td>6</td>
<td>max output ≥ 47,176,870, Marxen/Buntrock 1990</td>
</tr>
<tr>
<td>BB-6</td>
<td>2</td>
<td>8</td>
<td>very difficult to know, lower bound ≈ 10^10^4.6, Kropitz 2010</td>
</tr>
<tr>
<td>Goldbach</td>
<td>50</td>
<td>8</td>
<td>Goldbach conjecture: famous open problem</td>
</tr>
<tr>
<td>RH</td>
<td>3866</td>
<td>8</td>
<td>Riemann hypothesis: famous open problem (using Yedidia TM)</td>
</tr>
<tr>
<td>BB-23</td>
<td>8</td>
<td>9</td>
<td>probably impossible to know, lower bound &gt; Graham’s number</td>
</tr>
<tr>
<td>BB-7918</td>
<td>5920</td>
<td>10</td>
<td>value not expressible in ZF, Yedidia &amp; Aaronson 2016</td>
</tr>
<tr>
<td>UTM*</td>
<td>1000</td>
<td>10</td>
<td>universal Turing Machine, halting problem on any input</td>
</tr>
</tbody>
</table>

Notes: Bits per LOC ≈ 40 (using gzip java.base/share/classes/java/**). Inscrutability: 10=impossible (9=probably), n=takes (r^n) seconds to “master”, r ≈ 30. Data on BB’s and other TMs from [http://www.scottaaronson.com/blog/?p=2725](http://www.scottaaronson.com/blog/?p=2725)
Java safety checking is never finished

- Pick one: Online check OR offline check OR error.
- Offline (static) checks are inherently inconclusive (though helpful).

- GC OR hand-made storage allocation OR dangling pointers.
- Verifier-enforced RTTI OR hand-checked types OR invalid data.
- Array range checks OR hand-solved diophantines OR buffer overrun.

- Racer detection OR static concurrency checks OR races and excursions.
Java safety checking is never finished

- **Dynamic** safety checks are a Java super-power.
  - It’s not that we are too lazy to build a static checker; we do those.
  - Rice’s Theorem seems to indicate that static checkers can’t check…
    - (Can’t check useful, data-dependent safety invariants.)
  - Java’s excellent trick: Useful dynamic checks that are **cheap enough**.
    - (Cheap enough that they can be always on.)
    - (Useful to help programmers reason about their code.)
Can races be rare and obvious?

ref:  https://blogs.oracle.com/jrose/larval-objects-in-the-vm

- Let’s be incremental: Guide users away from the snake pit.
  - Make immutability easier to get, easier to use.
  - Can we make mutable the rare opt-in? For new code (Amber).
- Give new ways to rehabilitate racy old types
  - Frozen arrays: No more defensive copying. (This is pretty simple.)
  - Typestate in objects: Larval-mutable, then adult-frozen. (Harder.)
- New safety checks:
  - Mutability (simpler), thread confinement (harder)
  - Maybe publication handshakes, redundant read detection (?)
Reasons to watch for racers
Brought to you by “Code against the spec., not the behavior.”

- Discover questionable data flows immediately.
- Better to fail during testing than to get bad data under load.
  - Often, it’s better to fail under load than get bad data!
- Relying on lucky hardware will lock all of us into that hardware.
  - See Shipilev, “Close Encounters”, section 3 (“Pitfalls”)
- Same point for this year’s JIT. Let’s get to next year’s aggressive JIT.
  - We don’t have -XX:-EnableOptimizationsInThisRelease
- There is less causality flying around if racers are arrested.
  - The remaining causes are simpler to understand and manage.
Design pattern: Immutable objects

- Supported in today’s JMM for object fields.
  - Amber and Valhalla will push more of these into the mainstream.
- The constructor does a special “freeze” operation.
- In the future, we might give users control over “freeze” also.
  - Allows a builder pattern which freezes a larval object to R/O adult.
- Also useful for deserialization (the Mother of all Builders).
  - Can rationalize the current mess with writes to finals.
JVM support: Frozen arrays

- A frozen array throws an exception if you try to write it.
- Its components are like “final” fields, in the JMM.
  - The factory does the same special “freeze” operation.
- New method: \( b = a.frozen() \). (Also Arrays.isFrozen.)
  - \( \text{int}[] a = \text{foo}(); \text{int}[] b = a.frozen(); \quad // \text{ensure that } b \text{ is frozen} \)
  - equals/hashCode/toString taken from Arrays.equals, etc.
- Deploying this to varargs would cut out some expensive bugs.
- Existing range check optimizations easily extend to frozen check.
  - Frozen check can use the same header as range check
Design pattern: Stable variables

- Supported in today’s VM only with private @Stable annotation.
- Racy access will see either the initial default or the final stable.
  - The JIT is specially tuned to prefer the non-default and inline it.
  - Allows cheap lazy data structures with full optimization.
- In the future we might give users access to this.
  - But it requires online enforcement of the “one write” rule.
  - (The current system just trusts itself. Only works for family.)
- (If arrays can be frozen, they can also be stable. More array fun.)
Design pattern sketch: Monotonic variables

- A variable that “only goes one way” in some (partial) order.
  - Racy reads would see a set of mutually ordered values, at worst.
  - Non-racy reads would see the largest/longest/completest value so far.
- Could use hardware atomic add/and/or in some cases.
- With Valhalla, special behaviors can be field-local with a wrapper type.
  - (Currently, such functions are always in a separate wrapper object.)
  - (Except volatile and final; they can apply to any field.)

- Useful? Maybe.
One way to arrest racers…

- Add a function which asserts that an object is already confined.
  - System.assertLock(x)
  - No-op if x is locked by this thread; throws IMSE otherwise.
  - Quick check of header bits. Easy to optimize (via CSE).
- Use the assertion in accessor methods:
  - class C { int x; int x() { assertLock(this); return x; } }
  - JMM effect: prove pre-existing ACQUIRE op before a normal return.
    - Result: Non-racing code is OK, racy code self-diagnoses.
  - Extensions: Integrate with object freezing. Build into “getfield”.
Design pattern sketch: Race-resistant class

- Make most or all external API points synchronized.
- Use read/write accessors for possibly-unsync. field access.
  - Ensure that these accessors assert a previous acquire.
- Can make external API users responsible for ensuring sync.
  - It is a detected error if the user fails to sync. when required.
  - User can batch up operations under one sync. if it makes sense.
- (On the other hand: Sync. should be confinable as an API point…)

- Unknown: Whether any APIs would benefit from such patterns.
Design pattern sketch: Thread-confined class

- Like StringBuilder, advertise single-thread use only.
- Store the current thread in the object at construction.
  - All access methods recheck the thread. (Easy to optimize.)
- Forbid users to share the confined object.
  - It is a detected error if another thread calls a method on it.
- This pattern can apply to arrays too (thread-local buffers).
- Some Panama “Scope” objects will be thread-confined.
- Extension: Explicit “hand-off” operation thread → thread.
  - Or even thread → parkingLotObj → thread. (Escrow containers.)
Making the best of the Object header

- The 64 bits at the top of every object is a sunk cost.
  - Can we get more benefit from it? Probably.
- First, make the standard “synchronized” feature more useful.
  - Find a way to let an object fold in custom synchronization.
  - Allow the monitor to be confined when it shouldn’t be used.
  - Allow one object to declare that another’s monitor is it’s “guard”.
- Second, overlay the micro-lock for field translations on the same bits.
  - (This is the “SeqLock” required to defend against value tearing.)
  - Also, replay all our “biased locking” tricks, except for revocation.
  - (Revocation is the Achilles heel for BL, but we can just assert.)
Applying “confinement” to current projects…
Project Valhalla interactions with confinement

- Value types
  - values provide a strong “nudge” towards immutability
  - value types can be stored in registers (need not lay down in heap)
  - atomicity of multi-word values – need a SeqLock in obj. header

- Specialization
  - polymorphic value-holders *might* provide special JMM linkages
  - packed containers will provide alternatives to racy old int[]
  - perhaps we can specialize on behavior as well as type?
Project Amber, possible uses of confinement

- Non-mutable data classes are distorted by array mutability
  - Would be Nice to declare frozen arrays as normative for D.C.s.
  - D.C.s by default would freeze their array inputs.
  - Frozen arrays would act like lists on equals/hashCode/toString.
  - Normalize the defensive copying pattern. (Similarly for varargs.)
- Mutable data classes dare not expose plain fields to the clients.
  - Because field access is symmetric across reads and writes.
  - But the writes need more access control than the reads.
  - Possible solution: “sealed” fields (public-read, private-write).
Project Panama uses of confinement

- Vector API: uses (Minimal) Value Types, to get vectors up into registers
- Data binding
  - Off-heap pointers must be confined with extreme care!
  - Note: This is a pre-existing problem with DirectByteBuffers.
  - Tackling this vexed problem has forced us to wrestle the JMM.
- API binding
  - Lots of work figuring out how to “hide” native API details.
  - Binders and Java interfaces let us build clean O-O models for them.
  - This is just old-fashioned object-based abstraction.
Project Metropolis uses of confinement

- AOT compiler – ahead-of-time BSM execution needs confinement
- C2/Graal succession – confine Java code of JIT from app. code
- Java-on-Java implementation – same is true for non-JIT parts of VM
Confinement! Confinement!

▪ (He keeps saying that. Should we call the men in white coats?)
▪ Main point: Java’s policies on mutability don’t scale well.
  ▪ Require too much Programmer Discipline, elicits magic thinking.
  ▪ Coloring inside the JMM lines should not be my hardest task!
▪ Let’s add freezing to simplify the model, by blocking many bad writes.
  ▪ Later, add racer arrests, to support larval-adult transitions.
▪ Also, let’s aim for libraries which verify and exploit confined lambdas.
  ▪ Lambda cracking is needed. (It’s also needed for the Vector API.)
▪ And more: We will need to work out rules for “pure” code and data.
Confinement! Confinement!
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.
QUESTIONS?