Java Memory Model Unlearning Experience

or, «Crazy Russian Guy Yells About JMM»

Aleksey Shipilëv

shade@redhat.com
@shipilev

Safe Harbor / Тихая Гавань

Anything on this or any subsequent slides may be a lie. Do not base your decisions on this talk. If you do, ask for professional help.

Всё что угодно на этом слайде, как и на всех следующих, может быть враньём. Не принимайте решений на основании этого доклада. Если всё-таки решите принять, то наймите профессионалов.

Theory

Spec: ...vs Implementation

Everybody intuitively understands the difference between the *specification* and the *implementation*

```
class Integer {
  /**
   * Returns a {@code String} object representing the
   * specified integer. The argument is converted to signed decimal
   * representation and returned as a string, exactly as if ...
   */
  public static String toString(int i) {
    // Who cares what is going on here?
  }
}
```

Spec: Good Spec Is A Balance

- **Under**specify, and things become unusable:

```
/**
 * This method can do whatever it pleases.
 */
public void summonNasalDemons(int count) { ... }
```

- **Over**specify, and implementation choices are limited:

```
/**
 * This method checks if Java program halts.
 */
public boolean checkHalt(String program) { ... }
```

Spec: Abstract Machines

Language semantics is *specified* by the behavior of the *abstract machine*

```
public int m() {
  int x = 42;
  int y = 34;
  int t = x + y;
  return t;
}
```

$\Rightarrow$

```
m:
  ...prolog...
  mov $76$, %rax
  ...epilog...
  ret
```

If the result is not distinguishable from the *abstract machine* behavior, nobody cares how it was achieved!

Spec: JMM Is Part Of Abstract Machine

If the result is not distinguishable from the *abstract machine* behavior, nobody cares how it was achieved!

```
volatile int x;
public int m() {
  x = 1;
  x = 2;
  return x;
}
```

$\Rightarrow$

```
m:
...prolog...
mov $2$, (mem)
mov $2$, %rax
...epilog...
ret
```

(In practice, not all optimizations are... practical)

JMM: Talk Idea

JMM is simple! (Not joking.)

JMM: Talk Idea

JMM is simple! (Not joking.)

The problem is educational:

- Most JMM talks discuss what JMM **is** about:
    - Works, but piles on naive misconceptions
    - Also talks about implementations, blurring the whole thing

JMM: Talk Idea

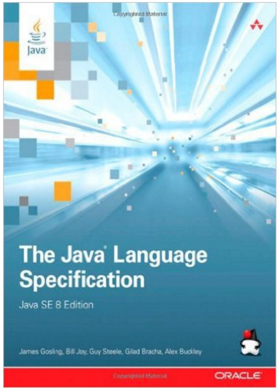
JMM is simple! (Not joking.)

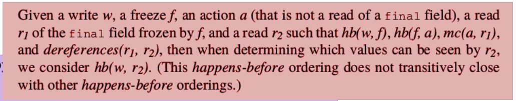
The problem is educational:

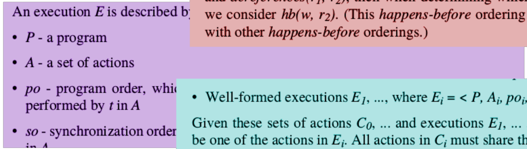
- Most JMM talks discuss what JMM **is** about:
  - Works, but piles on naive misconceptions
  - Also talks about implementations, blurring the whole thing
- This talk discusses what JMM **is not** about:
  - This is the **unlearning** experience!
  - (And we try to deconstruct misconceptions)

JMM: Problem

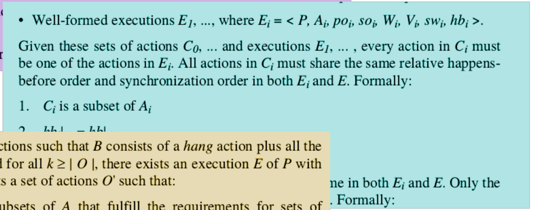
«Oh, give me 5 minutes to read up on JMM!»


Java
The Java Language Specification
Java SE 8 Edition
James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley
ORACLE


Given a write w, a freeze f, an action a (that is not a read of a `final` field), a read r_1 of the `final` field frozen by f, and a read r_2 such that $hb(w, f)$, $hb(f, a)$, $mc(a, r_1)$, and $dereferences(r_1, r_2)$, then when determining which values can be seen by r_2, we consider $hb(w, r_2)$. (This *happens-before* ordering does not transitively close with other *happens-before* orderings.)


An execution E is described b

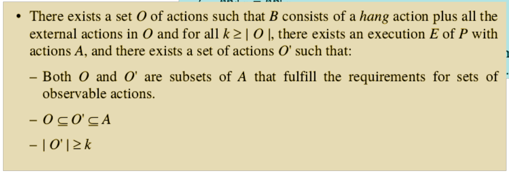
- P - a program
- A - a set of actions
- po - program order, whi
  performed by t in A
- so - synchronization order


- Well-formed executions E_1, ..., where $E_i = < P, A_i, po_i, so_i, W_i, V_i, sw_i, hb_i >$.

Given these sets of actions C_0, ... and executions E_1, ... , every action in C_i must be one of the actions in E_i. All actions in C_i must share the same relative happens-before order and synchronization order in both E_i and E. Formally:

1. C_i is a subset of A_i

ne in both E_i and E. Only the
. Formally:


- There exists a set O of actions such that B consists of a *hang* action plus all the external actions in O and for all $k \geq | O |$, there exists an execution E of P with actions A, and there exists a set of actions O' such that:
  - Both O and O' are subsets of A that fulfill the requirements for sets of observable actions.
  - $O \subseteq O' \subseteq A$
  - $| O' | \geq k$

JMM: Actions and Executions

$$\text{Executions} \approx \text{Actions} \cup \text{Orders} \cup \text{Consistency Rules}$$

JMM: Actions and Executions

$$\text{Executions} \approx \text{Actions} \cup \text{Orders} \cup \text{Consistency Rules}$$

Executions are the behaviors of the **abstract machine**, not the behavior of final implementation. They define all possible ways the Java program can possibly execute.

JMM: Actions and Executions

Executions $\approx$ Actions $\cup$ Orders $\cup$ Consistency Rules

Actions:

- $w(field, V)$ – write value V into $field$
- $r(field) : V$ – read value V from $field$
- $L(monitor)$ – lock the $monitor$
- $UL(monitor)$ – unlock the $monitor$
- ...

JMM: Actions and Executions

Executions ≈ Actions ∪ Orders ∪ Consistency Rules

Orders:

$$w(a, 1) \xrightarrow{\text{hb}} r(a) : 1 \ldots w(a, 2)$$

Consistency rules:

- PO consistency
- SO consistency, SO - PO consistency
- HB consistency

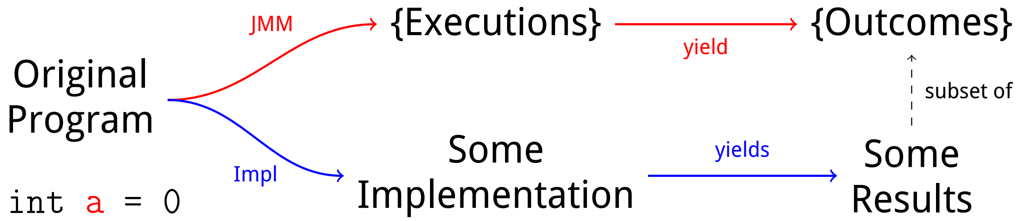
JMM: Umm...

When someone explains something to you multiple times but you still have no idea wtf is going on

JMM: Why?

$$\begin{tikzcd}
 & \{\text{Executions}\} \arrow[r, "\text{yield}"'] & \{\text{Outcomes}\} \\
\text{Original Program} \arrow[ur, bend left, "\text{JMM}"] \arrow[dr, bend right, "\text{Impl}"'] & & \\
 & \text{Some Implementation} \arrow[r, "\text{yields}"] & \text{Some Results} \arrow[uu, dashed, "\text{subset of}"']
\end{tikzcd}$$

JMM: Why?



```
   int a = 0
-----------------
 a = 1  | a = 2
r1 = a  |
```

JMM: Why?

$$w(a, 1) \xrightarrow{\text{hb}} r(a) : 1 \ldots w(a, 2)$$

$$w(a, 1) \xrightarrow{\text{hb}} r(a) : 2 \ldots w(a, 2)$$

$$\begin{tikzcd}
 & \text{\{Executions\}} \arrow[r, "\text{yield}"'] & \text{\{Outcomes\}} \\
\text{Original Program} \arrow[ur, bend left, "\text{JMM}"] \arrow[dr, bend right, "\text{Impl}"'] & & \\
 & \text{Some Implementation} \arrow[r, "\text{yields}"] & \text{Some Results} \arrow[uu, dashed, "\text{subset of}"']
\end{tikzcd}$$

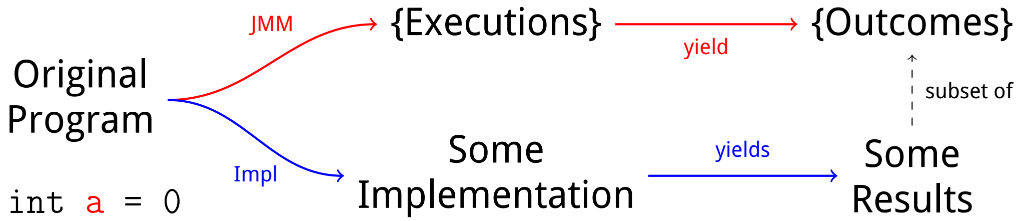
```
      int a = 0
-------------------
 a = 1   |  a = 2
r1 = a   |
```

JMM: Why?

$$w(a,1) \xrightarrow{\text{hb}} r(a) : 1 \ldots w(a,2)$$

$$w(a,1) \xrightarrow{\text{hb}} r(a) : 2 \ldots w(a,2)$$

$$r1 \in \{1, 2\}$$



```
   int a = 0
a = 1    | a = 2
r1 = a   |
```

JMM: Why?

$$w(a,1) \xrightarrow{\text{hb}} r(a):1 \ldots w(a,2)$$

$$w(a,1) \xrightarrow{\text{hb}} r(a):2 \ldots w(a,2)$$

$$r1 \in \{1,2\}$$

$$\begin{tikzcd}
 & \text{\{Executions\}} \arrow[r, "\text{yield}"'] & \text{\{Outcomes\}} \\
\text{Original Program} \arrow[ur, bend left, "\text{JMM}"] \arrow[r, bend right, "\text{Impl}"'] & \text{Some Implementation} \arrow[r, "\text{yields}"] & \text{Some Results} \arrow[u, dashed, "\text{subset of}"']
\end{tikzcd}$$

```
     int a = 0
--------------------
 a = 1    | a = 2
 r1 = a   |
```

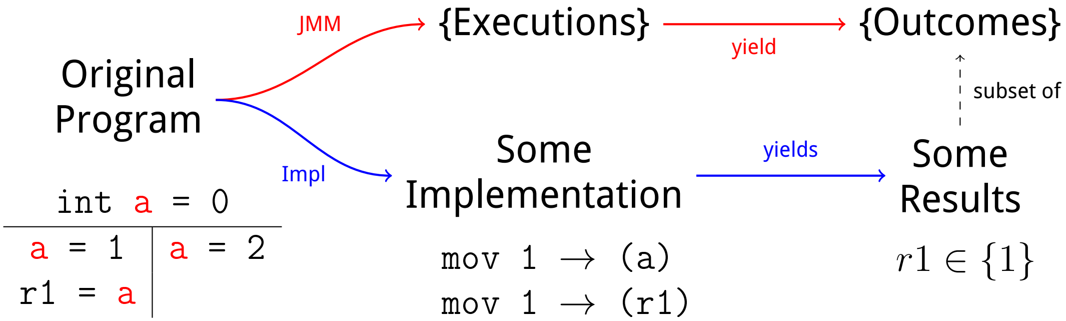
```
mov 1 → (a)
mov 1 → (r1)
```

JMM: Why?

$$w(a,1) \xrightarrow{\text{hb}} r(a):1 \ldots w(a,2)$$

$$w(a,1) \xrightarrow{\text{hb}} r(a):2 \ldots w(a,2)$$

$$r1 \in \{1,2\}$$

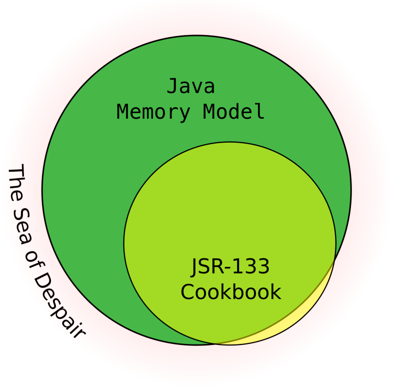


```
   int a = 0
-------------------
 a = 1  | a = 2
r1 = a  |
```

```
mov 1 → (a)
mov 1 → (r1)
```

$$r1 \in \{1\}$$

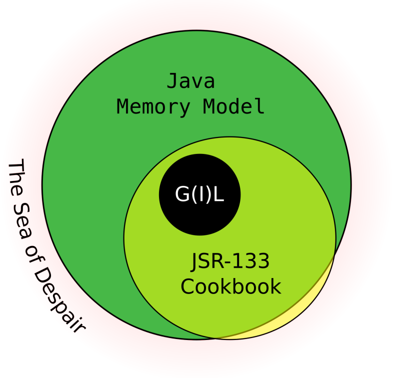
JMM: Takeaway #1: Studying Implementations



Implementations are allowed to generate the **subset** of allowed outcomes, not all of them

- You can study JSR 133 Cookbook, but take it with a grain of salt

JMM: Takeaway #1: Studying Implementations



Implementations are allowed to generate the **subset** of allowed outcomes, not all of them

- You can study JSR 133 Cookbook, but take it with a grain of salt
- Reductio ad absurdum: Global Interpreter Lock

JMM: JSR 133 Cookbook For Compiler Writers!

```
...
volatile load
[LoadLoad + LoadStore]
...
```

```
...
[StoreStore + LoadStore]
volatile store
[StoreLoad]
...
```

«Oh wow, so simple!»

1. Do not push operations after the `volatile store`
2. Do not pull operations before the `volatile load`
3. Do (1), (2) for `synchronized` `enter/exit`
4. Do not push operations after writing `final` fields

JMM: Lock Coarsening with JSR 133 Cookbook

```
void m() {
  synchronized(this) {
    x = 1;
  }
  synchronized(this) {
    y = 1;
  }
}
```

$\xrightarrow{\text{Cookbook}}$

```
void m() {
  [LoadStore]   // monitorenter
  x = 1;
  [StoreStore]  // monitorexit
  [StoreLoad]

  [LoadStore]   // monitorenter
  y = 1;
  [StoreStore]  // monitorexit
  [StoreLoad]
}
```

Can we reorder `x = 1` and `y = 1`?
JSR 133 Cookbook: Nope, you cannot.

JMM: Lock Coarsening with JMM

```
void m() {
  synchronized(this) {
    x = 1;
  }
  synchronized(this) {
    y = 1;
  }
}
```

$\xrightarrow{\text{coarsening}}$

```
void m() {
  synchronized(this) {
    y = 1;
    x = 1;
  }
}
```

JSR 133 Cookbook: Nope, you cannot.
Java Memory Model: Of course you can.
HotSpot: OK, doing it!

JMM: Takeaway #2, Implementation Details

These are not mandated by specification,
these are implementation details:

```
void barrier() {
  synchronized(this) {}; // do barrier!
}
```

JMM: Takeaway #2, Implementation Details

These are not mandated by specification,
these are implementation details:

```
void barrier() {
  synchronized(this) {}; // do barrier!
}
```

```
volatile int v;
void barrier() {
  v = 1; // do barrier!
}
```

JMM: Takeaway #2, Implementation Details

These are not mandated by specification,
these are implementation details:

```
void barrier() {
  synchronized(this) {}; // do barrier!
}
```

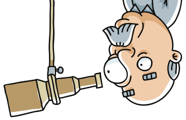
```
volatile int v;
void barrier() {
  v = 1; // do barrier!
}
```

```
class MyClass {
  volatile int v;
  MyClass() {
    this.v = 42;
    // do barrier!
  }
}
```

redhat

Behaviors

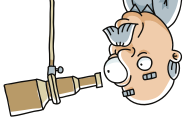
Races: Example 1.1



```
class M { ... }
M m;
```

Thread 1	Thread 2
`m = new M();`	`M lm = m;`
`m = null;`	`r1 = (lm != null);`
	`r2 = (lm != null);`

Races: Example 1.1



```
class M { ... }
M m;
```

`m = new M();`	`M lm = m;`
`m = null;`	`r1 = (lm != null);`
	`r2 = (lm != null);`

JMM allows only (F, F) and (T, T)

Races: Example 1.1, Counter-Argument

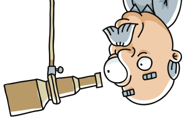
Can’t compiler «inline» the local variable?

```
class M { ... }
M m;
```

`m = new M();`	
`m = null;`	`r1 = (m != null);`
	`r2 = (m != null);`

redhat.

Races: Example 1.1, Counter-Argument



Can’t compiler «inline» the local variable?

```
               class M { ... }
               M m;
---------------------------------------------
m = new M();  |
m = null;     | r1 = (m != null);
              | r2 = (m != null);
```

See, there is an obvious execution that yields (T, F) now!

$$\ldots r(m) : !null \xrightarrow{\text{po}} r(m) : null$$

JMM: Program Order

Program order (PO) provides the link between the execution and the program in question

- PO – total order for any given thread in isolation
- **PO consistency**: PO is consistent with the source code order in the original program

JMM: PO And Transformations

Original program:

```
M lm = m;
r1 = (lm != null);
r2 = (lm != null);
```

$$w(m, *) \xrightarrow{\text{po}} w(m, null)$$

$$r(m) : *$$

Transformed program:

```
r1 = (m != null);
r2 = (m != null);
```

$$w(m, *) \xrightarrow{\text{po}} w(m, null)$$

$$r(m) : * \xrightarrow{\text{po}} r(m) : *$$

JMM: PO And Transforma

This execution does not relate to the original program, oops

Original program:

```
M lm = m;
r1 = (lm != null);
r2 = (lm != null);
```

$$w(m,*) \xrightarrow{\text{po}} w(m,null)$$

$$r(m):*$$

Transformed program:

```
r1 = (m != null);
r2 = (m != null);
```

$$w(m,*) \xrightarrow{\text{po}} w(m,null)$$

$$r(m):* \xrightarrow{\text{po}} r(m):*$$

JMM: PO And Transformat

This execution should be used to reason about outcomes for the transformed program

Original pr

```
M lm = m;
r1 = (lm != null);
r2 = (lm != null);
```

$$w(m, *) \xrightarrow{\text{po}} w(m, null)$$

$$r(m) : *$$

Transformed program:

```
r1 = (m != null);
r2 = (m != null);
```

$$w(m, *) \xrightarrow{\text{po}} w(m, null)$$

$$r(m) : * \xrightarrow{\text{po}} r(m) : *$$

JMM: PO And Transformations

Original program:

```
M lm = m;
r1 = (lm != null);
r2 = (lm != null);
```

$$w(m,*) \xrightarrow{\text{po}} w(m,null)$$
$$r(m) : *$$

Transformed program:

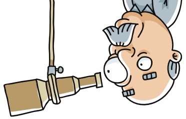
```
r1 = (m
r2 = (m
```

$, null)$

$) : *$

PO consistency:
Original program has single read?
Relatable executions also have single read!

Races: Example 1.2, Null-Checks



In Java, unlike C/C++:

```
int s() {
  M lm = m;
  if (lm != null) {
    return lm.x; // <--- This does not risk NPE
  else
    return 0;
}
```

This would later become a building block for so called «benign» data races

Races: Takeaway #3

1. Data race behavior is still somewhat deterministic
   - Racy reads are stronger than in other languages
   - Weird stuff still happens, but not completely catastrophic

2. Memory-model-wise, there is a difference:

```
int m1() {
  int x1 = field;
  int x2 = field;
  return x1 + x2;
}
```

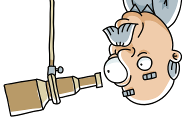
```
int m2() {
  int x1 = field;
  int x2 = x1;
  return x1 + x2;
}
```

Races: JMM and Ordering Modes

Java 8	Java 9	Definite
-	-	N
plain	VH Plain	Y
volatile	VH SeqCst	Y

Coherence

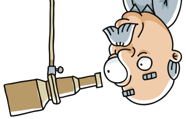
Coherence: Example 2.1



`int x;`

`x = 1;`	`r1 = x; //` r_1
	`r2 = x; //` r_2

Coherence: Example 2.1

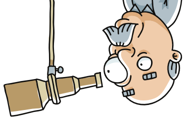


int x;	
x = 1;	r1 = x; // r_1
	r2 = x; // r_2

JMM allows observing $(1,0)$, see:

$$w(x,1)\ \ldots\ r_1(x) : 1 \xrightarrow{\text{po}} r_2(x) : 0$$

Coherence: Example 2.1



```
        int x;
x = 1; | r1 = x; // r1
       | r2 = x; // r2
```

JMM allows observing $(1, 0)$, see:

$$w(x, 1) \ \ldots \ r_1(x) : \boxed{1} \xrightarrow{\text{po}} r_2(x) : \boxed{0}$$

This execution is PO consistent, both reads are here!

Coherence: Definition

Coherence *(def.)*:
The writes to the single memory location appear to be in a total order consistent with program order

- Most hardware gives this for free
- Most optimizers give up on this by default (i.e. do not track the order of reads)

redhat

JMM: Consistency Rules

PO consistency affects the **structure** of the execution. What we need: a consistency rule that affects **values** observed by the actions.

In JMM, there are two of them:

1. Happens-before (HB) consistency
2. Synchronization order (SO) consistency

JMM: Consistency Rules

PO consistency affects the **structure** of the execution. What we need: a consistency rule that affects **values** observed by the actions.

In JMM, there are two of them:

1. Happens-before (HB) consistency
2. Synchronization order (SO) consistency ← **now!**

JMM: SO – Synchronization Order

SO covers all *synchronization actions*:
volatile read/write, lock/unlock, etc.

- **SO** is a total order («All SA actions relate to each other»)
- **SO - PO consistency**: $\xrightarrow{so}$ and $\xrightarrow{po}$ agree
- **SO consistency**: reads see only the latest write in $\xrightarrow{so}$

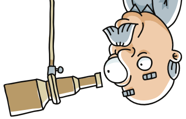
JMM: SO – Synchronization Order

SO covers all *synchronization actions*:
volatile read/write, lock/unlock, etc.

- **SO** is a total order («All SA actions relate to each other»)
- **SO - PO consistency**: $\xrightarrow{so}$ and $\xrightarrow{po}$ agree
- **SO consistency**: reads see only the latest write in $\xrightarrow{so}$

Just what coherence wants!

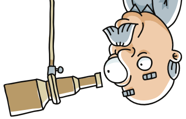
Coherence: Example 2.2



`volatile int x;`

`x = 1;`	`r1 = x; //` r_1
	`r2 = x; //` r_2

Coherence: Example 2.2



volatile int x;	
x = 1;	r1 = x; // r_1
	r2 = x; // r_2

Valid executions give $(0,0), (1,1), (0,1)$:[a]

$$w(x,1) \xrightarrow{\text{so}} r_1(x) : \boxed{1} \xrightarrow{\text{so}} r_2(x) : \boxed{1}$$

$$r_1(x) : \boxed{0} \xrightarrow{\text{so}} w(x,1) \xrightarrow{\text{so}} r_2(x) : \boxed{1}$$

$$r_1(x) : \boxed{0} \xrightarrow{\text{so}} r_2(x) : \boxed{0} \xrightarrow{\text{so}} w(x,1)$$

[a] Proving no other outcomes exist is left as an exercise for the reader

Coherence: Takeaway #4

1. Races laugh at our presuppositions about order
   - Most of the time, there is a complete free-for-all
   - Madness usually manifests after code transformations
   - Although hardware can also get us down
2. Coherency, while basic, is not guaranteed, unless...
   - We use `volatile` that is naturally coherent
   - We use weaker forms of `VarHandles` that are coherent
   - We use properly synchronized (non-racy) reads

Coherence: JMM and Ordering Modes

Java 8	Java 9	Definite	Coherence
-	-	N	N
plain	VH Plain	Y	N
-	VH Opaque	Y	Y
volatile	VH SeqCst	Y	Y

Causality

Causality: SW – Synchronizes-With Order

When one SA «sees» the value of another SA, they are said to be in «synchronizes-with» (SW) relation

- SW is a partial order
- SW connects the operations that «see» each other
- Acts like the «bridge» between the threads

Causality: HB – Happens-Before Order

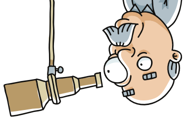
HB is a transitive closure over the union of PO and SW

- **HB** is a partial order (Translation: not everything is connected)
- **HB consistency**: reads observe either:

  the last write in $\xrightarrow{hb}$, or

  any other write, not ordered by $\xrightarrow{hb}$

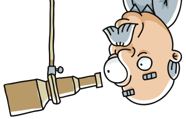
Causality: Example 3.1



```
int x;
volatile int y;
```

x = 1;	r1 = y;
y = 1;	r2 = x;

Causality: Example 3.1



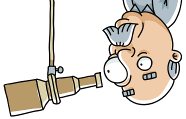
```
    int x;
volatile int y;
```

x = 1;	r1 = y;
y = 1;	r2 = x;

We are dealing with this class of executions:

$$w(x,1) \xrightarrow{\text{po}} w(y,1) \;\ldots\; r(y):* \xrightarrow{\text{po}} r(x):*$$

Causality: Example 3.1



```
int x;
volatile int y;
```

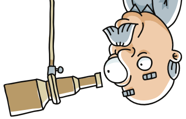
`x = 1;`	`r1 = y;`
`y = 1;`	`r2 = x;`

Racy subclass:

$$w(x,1) \xrightarrow{hb} w(y,1) \ \ldots \ r(y) : \boxed{0} \xrightarrow{hb} r(x) : \boxed{0}$$

$$w(x,1) \xrightarrow{hb} w(y,1) \ \ldots \ r(y) : \boxed{0} \xrightarrow{hb} r(x) : \boxed{1}$$

Causality: Example 3.1



```
      int x;
volatile int y;
```

`x = 1;`	`r1 = y;`
`y = 1;`	`r2 = x;`

Non-racy subclass:

$$w(x,1) \xrightarrow{hb} w(y,1) \xrightarrow{hb} r(y) : \boxed{1} \xrightarrow{hb} r(x) : \boxed{1}$$

$$w(x,1) \xrightarrow{hb} w(y,1) \xrightarrow{hb} r(y) : \boxed{1} \xrightarrow{hb} r(x) : \boxed{0}$$

Causality: Look Closer, #1

Happens-before is defined over *actions*,
not over statements: notice no HB between `volatile` ops!

$$\begin{tikzcd}
\boxed{w(x,1)} \arrow[r, "\text{hb}"] & w(y,1) \quad \ldots \quad r(y) : 0 \arrow[r, "\text{hb}"] & \boxed{r(x) : 0} \arrow[ll, dashed, bend left]
\end{tikzcd}$$

not required to see this

Causality: Look Closer, #2

This violates HB consistency:

$$\begin{tikzcd} \boxed{w(x,1)} \arrow[r, "hb"] & w(y,1) \arrow[r, "hb"] & r(y) : 1 \arrow[r, "hb"] & \boxed{r(x) : 0} \arrow[lll, dashed, bend left, "\text{should have seen this!}"] \end{tikzcd}$$

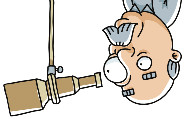
should have seen this!

Causality: Observing the `volatile` store causes observing everything stored before it

Causality: Example 3.2

Notice the order is different

```
        int x;
   volatile int y;
-------------------
y = 1;  |  r1 = x;
x = 1;  |  r2 = y;
```



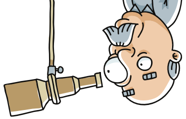
Causality: Example 3.2

Notice the order is different

```
      int x;
volatile int y;
-----------------
y = 1; | r1 = x;
x = 1; | r2 = y;
```

Hey, look how $(1, 0)$ is allowed:

$$w(y, 1) \xrightarrow{\text{hb}} w(x, 1) \;\ldots\; r(x) : 1 \xrightarrow{\text{hb}} r(y) : \boxed{0}$$



Causality: Example 3.2

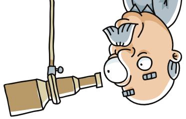
Notice the order is different

```
        int x;
  volatile int y;
-------------------
y = 1;  | r1 = x;
x = 1;  | r2 = y;
```

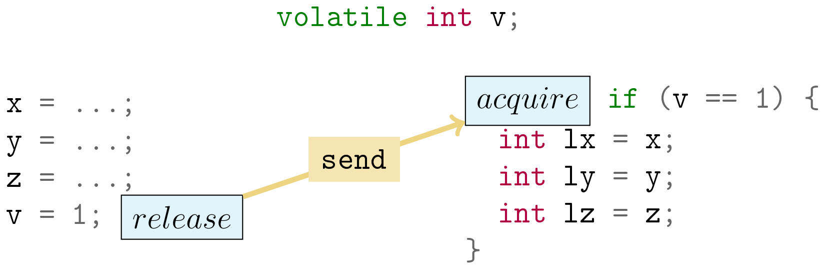
Hey, look how $(1,0)$ is allowed:

$$w(y,1) \xrightarrow{\text{hb}} w(x,1) \;\ldots\; r(x):1 \xrightarrow{\text{hb}} r(y): \boxed{0}$$

Look: irrelevant that `y` is `volatile`!



Causality: Safe Publication



```
volatile int v;

x = ...;                  if (v == 1) {
y = ...;                    int lx = x;
z = ...;                    int ly = y;
v = 1;                      int lz = z;
                          }
```

- As if «commits to memory», but only for acq/rel pair
- `release` «commits», `acquire` gets the committed
- `acquire` has to see `release` witness!

Causality: Takeaway #5

1. Safe publication is the major (and simple) rule
   - Identify your **acquires** and **releases**
   - Check that **acquires/releases** are on all paths
   - Learn this rule! Then learn it again!

2. The whole thing does not require JMM reasoning
   - Hardly anyone applies «happens-before» correctly
   - Hardly anyone can do it reliably
   - It is very easy to miss the racy access

Causality: JMM and Ordering Modes

Java 8	Java 9	Definite	Coherence	Causality
-	-	N	N	N
plain	VH Plain	Y	N	N
-	VH Opaque	Y	Y	N
-	VH Acq/Rel	Y	Y	Y
volatile	VH SeqCst	Y	Y	Y

Consensus

Consensus: Example 4.1

```
volatile int x, y;
```

`x = 1;`	`y = 1;`	`int r1 = y;`
		`int r2 = x;`
			`int r3 = x;`
			`int r4 = y;`

HB alone allows seeing $(1, 0, 1, 0)$:

$$w(y,1) \xrightarrow{\text{hb}} r_1(y) : \boxed{1} \xrightarrow{\text{hb}} r_3(x) : \boxed{0}$$

$$w(x,1) \xrightarrow{\text{hb}} r_2(x) : \boxed{1} \xrightarrow{\text{hb}} r_4(y) : \boxed{0}$$

Consensus: SC

Sequential Consistency (SC): *(def.)*
«…the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program»

Consensus: SO – Synchronization Order

SO covers all *synchronization actions*:
volatile read/write, lock/unlock, etc.

- **SO** is a total order («All SA actions relate to each other»)
- **SO - PO consistency**: $\xrightarrow{so}$ and $\xrightarrow{po}$ agree
- **SO consistency**: reads see only the latest write in $\xrightarrow{so}$

Just what Sequential Consistency wants!

Consensus: Takeaway #6

1. SO $\approx$ Sequential Consistency
   - Want SC? You have to go full-blown `volatile`
   - Seed enough `volatiles` around your program, and it eventually becomes data-race-free! /s

2. Sequential Consistency is not always needed
   - Extreme costs to get it in distributed systems
   - Most examples so far were fine with just Release/Acquire!

Consensus: JMM and Ordering Modes

Java 8	Java 9	Definite	Coherence	Causality	Consensus
-	-	N	N	N	N
plain	VH Plain	Y	N	N	N
-	VH Opaque	Y	Y	N	N
-	VH Acq/Rel	Y	Y	Y	N
volatile	VH SeqCst	Y	Y	Y	Y

Finals: Example 5.1

```
class M { final int x = 42; }
M m;
```

`m = new M()`	`M lm = m`
	`if (lm != null)`
	`  r1 = lm.x`
	`else`
	`  r1 = 1`

Finals: Example 5.1

```
class M { final int x = 42; }
M m;
```

m = new M()	M lm = m
	if (lm != null)
	r1 = lm.x
	else
	r1 = 1

JMM guarantees seeing the value of `final` field here:

$$r1 \in \{1, 42\}$$

Finals: Example 5.1

```
class M { final int x = 42; }
M m;
```

`m = new M()`	`M lm = m`
	`if (lm != null)`
	`  r1 = lm.x`
	`else`
	`  r1 = 1`

Special rule, if `x` is a `final` field:

$$w(x, 42) \xrightarrow{hb} r(x) : 42$$

Finals: Example 5.2

```
class M { volatile int x = 42; }
M m;
--------------------------------------
m = new M() | M lm = m
            | if (lm != null)
            |   r1 = lm.x
            | else
            |   r1 = 1
```

Finals: Example 5.2

```
class M { volatile int x = 42; }
M m;
```

m = new M()	M lm = m
	if (lm != null)
	r1 = lm.x
	else
	r1 = 1

JMM allows (0) here:

$$w(cm.x, 42) \xrightarrow{\text{hb}} w(cm, m) \ \ldots \ r(m) : lm \xrightarrow{\text{hb}} r(lm.x) : 0$$

Finals: Example 5.2

```
class M { volatile int x = 42; }
M m;
```

m = new M()	M lm = m
	if (lm != null)
	r1 = lm.x
	else
	r1 = 1

$$\texttt{volatile} \notin \texttt{final}$$
$$\texttt{final} \notin \texttt{volatile}$$

Finals: Safe Construction

Special rule for `final` fields:

$$writes_{final} \xrightarrow{\text{hb}} reads_{final}$$

The derivation for that rule is complicated.

Two absolutely necessary things:

- Field is `final`
- Constructor does not publish `this`

Finals: Benign Races

```java
V v; // deliberately non-volatile

public V racyRacy() {
  V lv = v;              // RULE 1: Read it once (racily)
  if (lv == null) {      // RULE 2: Check it is fine
    v = compute();       // RULE 3: Recover by safely constructing
  }
  return lv;
}
```

Forgo one of the rules, and you get the **non-benign** race.

Finals: Benign Races, Real Example

```
public class AbstractMap<K, V> {
  transient Set<K> keySet; // non-volatile

  public Set<K> keySet() {
    Set<K> ks = keySet;  // RULE 1: Read it once (racily)
    if (ks == null) {    // RULE 2: Check it's fine
      ks = new KeySet(); // RULE 3: Recover by safely constructing
      keySet = ks;
    }
    return ks;
  }
}
```

Finals: Takeaway #7

1. Safe construction is another major (and simple) rule
   - Use it to protect against inadvertent races!
   - When it doubt, make all fields `final`

2. Benign races are seldom useful
   - Allow avoiding synchronized ops on critical paths
   - Work only if three rules are followed: single (racy) read, reliability check, recovery path that safely constructs

redhat.

Locks

Locks: JMM and Ordering Modes

Java 8	Java 9	Definite	Coherence	Causality	Consensus	Mutual Excl
-	-	N	N	N	N	N
plain	VH Plain	Y	N	N	N	N
-	VH Opaque	Y	Y	N	N	N
-	VH Acq/Rel	Y	Y	Y	N	N
volatile	VH SeqCst	Y	Y	Y	Y	N
locks	-	Y	Y	Y	Y	Y

Summing Up

Summing Up: Rule #1: Safe Publication

Golden Rule:
Thread 1: store everything, then **release**
Thread 2: **acquire**, then read anything

- Automatically happens when publishing via well-designed concurrency primitives
- Has to happen on **all possible** execution paths
- Has to happen in correct order

Summing Up: Rule #2: Safe Construction

Golden Rule:
When in doubt, make all fields `final`.

- Makes the whole thing more resilient to races
- Think «defense in depth»: survive in case some path fails to publish the instance safely

Summing Up: Rule #3: Benign Races

Golden Rule:
Object is safely constructed, and there is single read.

- Exotic optimization technique, rarely needed
- The (only) easy way to avoid synchronization

Summing Up: Rule #4: Exotic Modes

Golden Rule:
Don't.

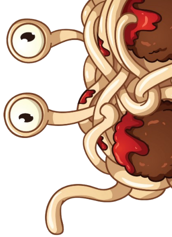
- Just don't!
- There are cases where performance is so important, you want to have weaker than `volatile`, but stronger than `plain` – `VarHandles` to rescue!

Practice

Practice: Double-Checked Locking

```
volatile T val;
public T get() {
  if ([1] val == null) {
    synchronized (this) {
      if ([2] val == null) {
        [3] val = new T();
      }
    }
  }
  return [4] val;
}
```

Practice: Double-Checked Locking

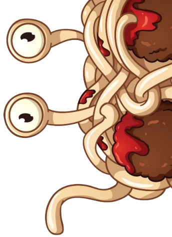


```
volatile T val;
public T get() {
  if ( 1  val == null) {
    synchronized (this) {
      if ( 2  val == null) {
         3  val = new T();
      }
    }
  }
  return  4  val;
}
```

Holy Macaroni, it does not work without `volatile`!

But why do you need it?

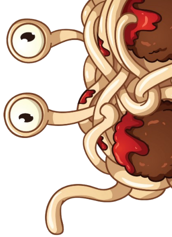
Practice: Double-Checked Locking



```
volatile T val;
public T get() {
  if ([1] val == null) {
    synchronized (this) {
      if ([2] val == null) {
        [3] val = new T();
      }
    }
  }
  return [4] val;
}
```

What ordering modes are necessary at 1, 2, 3, 4?

Practice: Double-Checked Locking

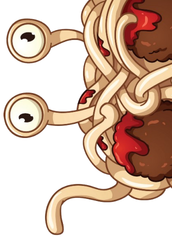


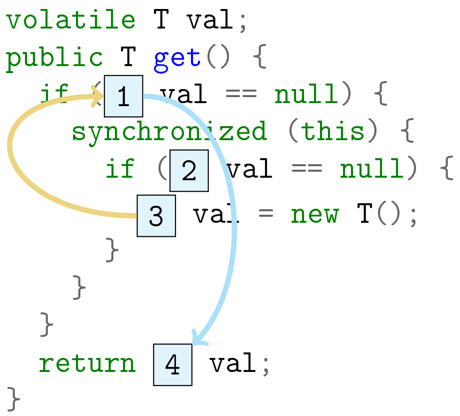
```
volatile T val;
public T get() {
  if ([1] val == null) {
    synchronized (this) {
      if ([2] val == null) {
        [3] val = new T();
      }
    }
  }
  return [4] val;
}
```

What ordering modes are necessary at 1, 2, 3, 4?

- Release/acquire: 3 $\rightarrow$ 1

Practice: Double-Checked Locking





```
volatile T val;
public T get() {
  if ([1] val == null) {
    synchronized (this) {
      if ([2] val == null) {
        [3] val = new T();
      }
    }
  }
  return [4] val;
}
```

What ordering modes are necessary at 1, 2, 3, 4?

- Release/acquire: 3 $\rightarrow$ 1
- Coherence: 1 $\rightarrow$ 4

Practice: DCL with VarHandles

```
static final VH = ...;
V val; // not volatile, specify at use-site

public V get() {
  if (VH.getAcquire(this) == null) {
    synchronized (this) {
      if (VH.get(this) == null) {
        VH.setRelease(this, new T());
      }
    }
  }
  return VH.get(this);
}
```

Lazy<V>: The Purest Form

```
public class Lazy<V> {
  final Supplier<V> s;
  V v;
  public Lazy(Supplier<V> s) {
    this.s = s;
  }

  public synchronized V get() {
    if (v == null) {
      v = s.get();
    }
    return v;
  }
}
```

Lazy instantiator:
obviously correct, right?

Lazy<V>: The Purest Form

```
public class Lazy<V> {
  final Supplier<V> s;
  V v;
  public Lazy(Supplier<V> s) {
    this.s = s;
  }

  public synchronized V get() {
    if (v == null) {
      v = s.get();
    }
    return v;
  }
}
```

Lazy instantiator:
obviously correct, right?

Let us optimize it a little.

Lazy<V>: The Purest Form

```java
public class Lazy<V> {
  final Supplier<V> s;
  V v;
  public Lazy(Supplier<V> s) {
    this.s = s;
  }

  public synchronized V get() {
    if (v == null) {
      v = s.get();
    }
    return v;
  }
}
```

Lazy instantiator:
obviously correct, right?

Let us optimize it a little.
First, let's apply DCL...

Lazy<V>: Optimized #1

```
final Supplier<V> s;
volatile V v;

public V get() {
  if (v == null) {
    synchronized (this) {
      if (v == null) {
        v = s.get();
      }
    }
  }
  return v;
}
```

Still works?

Lazy<V>: **Optimized #1**

```
final Supplier<V> s;
volatile V v;

public V get() {
  if (v == null) {
    synchronized (this) {
      if (v == null) {
        v = s.get();
      }
    }
  }
  return v;
}
```

Still works? It does!

Lazy<V>: Optimized #1

```
final Supplier<V> s;
volatile V v;

public V get() {
  if (v == null) {
    synchronized (this) {
      if (v == null) {
        v = s.get();
      }
    }
  }
  return v;
}
```

Still works? It does!

Let us polish it a bit:

1. `Supplier` is not really needed after first and only use
2. What if `Supplier` returns `null`?

Lazy<V>: Optimized #2

```
[what?] V v;
[what?] Supplier<V> s;

public V get() {
  if (s != null) {
    synchronized (this) {
      if (s != null) {
        v = s.get();
        s = null;
      }
    }
  }
  return v;
}
```

Ummm... Where to put `volatile` now?

- A. To field `v`
- B. To field `s`
- C. To both `v` and `s`
- D. 50:50
- E. Phone A Friend

Lazy<V>: Optimized #2, Try 1

```
volatile V v;
Supplier<V> s;

public T get() {
  if (□ s != null) {
    synchronized (this) {
      if (s != null) {
        v = s.get();
        □ s = null;
      }
    }
  }
  return v;
}
```

Let us put volatile to v.
Any problems?

`Lazy<V>`: Optimized #2, Try 1

```
volatile V v;
Supplier<V> s;

public T get() {
  if (□ s != null) {
    synchronized (this) {
      if (s != null) {
        v = s.get();
        □ s = null;
      }
    }
  }
  return v;
}
```

Let us put `volatile` to `v`. Any problems?

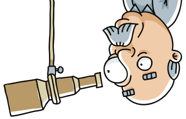
Oops, release/acquire is misplaced! Racy read of `s` potentially exposes

$$r(v) : null$$

`Lazy<V>`: Example 3.2

Notice the order is different

```
    int x;
volatile int y;
-----------------
y = 1; | r1 = x;
x = 1; | r2 = y;
```



Hey, look how $(1, 0)$ is allowed:

$$w(y,1) \xrightarrow{\text{hb}} w(x,1) \;\ldots\; r(x) : 1 \xrightarrow{\text{hb}} r(y) : \boxed{0}$$

Look: irrelevant that `y` is `volatile`!

`Lazy<V>`: Optimized #2, Try 2

```
V v;
volatile Supplier<V> s;

public T get() {
  if (□ s != null) {
    synchronized (this) {
      if (s != null) {
        v = s.get();
        □ s = null;
      }
    }
  }
  return v;
}
```

Let us put `volatile` to `s`.
Any problems?

Lazy<V>: Optimized #2, Try 2

```
V v;
volatile Supplier<V> s;

public T get() {
  if (□ s != null) {
    synchronized (this) {
      if (s != null) {
        v = s.get();
        □ s = null;
      }
    }
  }
  return v;
}
```

Let us put `volatile` to `s`.
Any problems?

No problem, our release/acquire witness gets us the proper `v`.

Lazy<V>: Optimized #2, Try 2

```
V v;
volatile Supplier<V> s;

public T get() {
  if (□ s != null) {
    synchronized (this) {
      if (s != null) {
        v = s.get();
        □ s = null;
      }
    }
  }
  return v;
}
```

Let us put volatile to s. Any problems?

No problem, our release/acquire witness gets us the proper v.

Optimizing further? We don't really like the volatile read!

Lazy<V>: **Optimized #3, Try 1**

```
V v;
volatile Supplier<V> s;

public V get() {
  if (□ v == null && s != null) {
    synchronized (this) {
      if (s != null) {
        v = s.get();
        s = null;
      }
    }
  }
  return □ v;
}
```

Non-`volatile` fast-path, nice! Any problems?

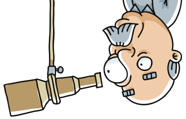
Lazy<V>: Optimized #3, Try 1

```
V v;
volatile Supplier<V> s;

public V get() {
  if (□ v == null && s != null) {
    synchronized (this) {
      if (s != null) {
        v = s.get();
        s = null;
      }
    }
  }
  return □ v;
}
```

Non-volatile fast-path, nice! Any problems?

Oops: no coherence between reads

`Lazy<V>`: Example 2.1



```
          int x;
x = 1; | r1 = x; // r1
       | r2 = x; // r2
```

JMM allows observing $(1, 0)$, see:

$$w(x, 1) \;\; \ldots \;\; r_1(x) : \boxed{1} \xrightarrow{\text{po}} r_2(x) : \boxed{0}$$

This execution is PO consistent, both reads are here!

Lazy<V>: Optimized #3, Try 2

```
V v;
volatile Supplier<V> s;

public V get() {
  V lv = □ v;
  if (lv == null && s != null) {
    synchronized (this) {
      if (s != null) {
        □ v = lv = s.get();
        s = null;
      }
    }
  }
  return lv;
}
```

Fixing up coherency with single read. Any problems left?

Lazy<V>: Optimized #3, Try 2

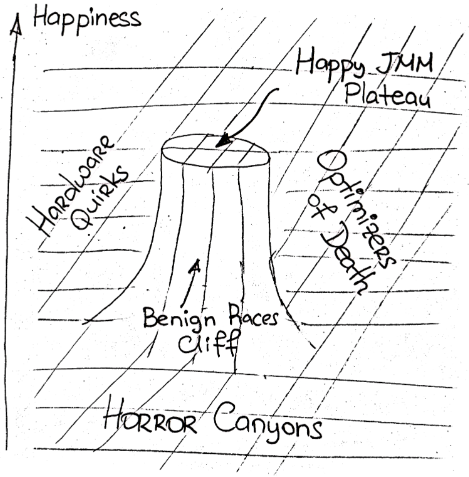
```
V v;
volatile Supplier<V> s;

public V get() {
  V lv = v;
  if (lv == null && s != null) {
    synchronized (this) {
      if (s != null) {
        v = lv = s.get();
        s = null;
      }
    }
  }
  return lv;
}
```

Fixing up coherency with single read. Any problems left?

No release/acquire on this path, oops.

Conclusions

Conclusions: In One Picture



Conclusions: In Four Paragraphs

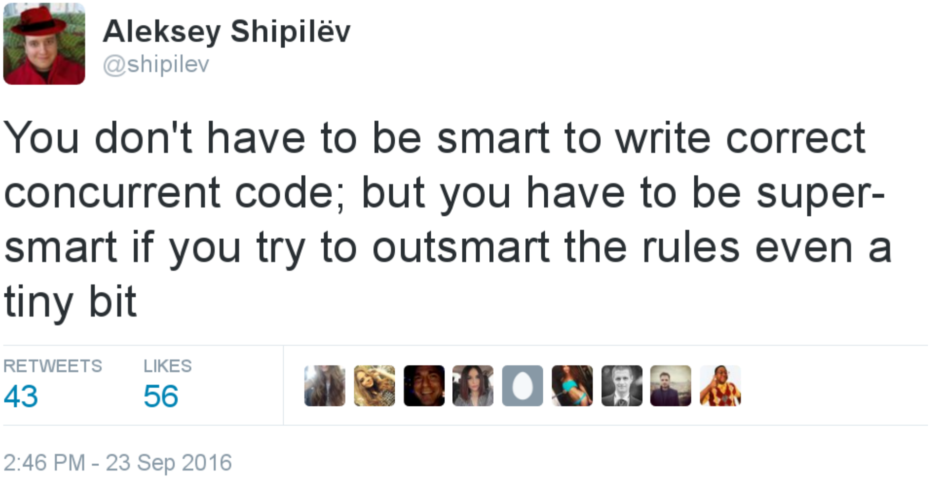
1. **Safe publication** and **safe construction** cover 99.99% of real concurrency cases!

2. **Benign races** cover another 0.00999% of performance optimization cases

3. All other fantasies on «what optimizers do», «what hardware does» – please keep them out

4. Want more? **Study JMM rules!**

Conclusions: In 280 Symbols



Aleksey Shipilëv
@shipilev

You don't have to be smart to write correct concurrent code; but you have to be super-smart if you try to outsmart the rules even a tiny bit

RETWEETS 43 LIKES 56

2:46 PM - 23 Sep 2016

Conclusions: Further Reading

In ascending order of difficulty:

1. «Safe Publication and Safe Initialization in Java»:
   `https://shipilev.net/blog/2014/safe-public-construction/`
2. «Java Memory Model Pragmatics»:
   `https://shipilev.net/blog/2014/jmm-pragmatics/`
3. «Close Encounters of JMM Kind»:
   `https://shipilev.net/blog/2016/close-encounters-of-jmm-kind/`
4. «Using JDK 9 Memory Order Modes»:
   `http://gee.cs.oswego.edu/dl/html/j9mm.html`

Backup

Backup: Global Memory Illusion

«Reordering» makes sense when there is an illusion of global synchronized memory.

```
               volatile int x, y;
-------------------------------------------------------
x = 1; | y = 1; | int r1 = y; | int r3 = x;
       |        | int r2 = x; | int r4 = y;
```

If we stick the barriers around the operations, everything is fine, right?

Backup: IRIW

```
volatile int x, y;
```

`x = 1;`	`y = 1;`	`int r1 = y;`	`int r3 = x;`
		`int r2 = x;`	`int r4 = y;`

$(r1, r2, r3, r4) = (1, 0, 1, 0)$ is forbidden by JMM: `volatile` ops are sequentially consistent.

Backup: IRIW With Barriers

`volatile int x, y;`

`<fullFence>`	`<fullFence>`	`<loadFence>`	`<loadFence>`
`x = 1;`	`y = 1;`	`int r1 = y;`	`int r3 = x;`
`<fullFence>`	`<fullFence>`	`<loadFence>`	`<loadFence>`
		`int r2 = x;`	`int r4 = y;`
		`<loadFence>`	`<loadFence>`

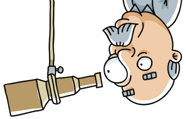
Backup: IRIW With Barriers

`volatile int x, y;`

`<fullFence>`	`<fullFence>`	`<loadFence>`	`<loadFence>`
`x = 1;`	`y = 1;`	`int r1 = y;`	`int r3 = x;`
`<fullFence>`	`<fullFence>`	`<loadFence>`	`<loadFence>`
		`int r2 = x;`	`int r4 = y;`
		`<loadFence>`	`<loadFence>`

PowerPC: LOL, nice try, but
$(r1, r2, r3, r4) = (1, 0, 1, 0)$

Backup: Consensus, Example 4.2

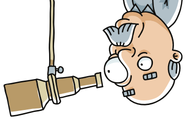


```
synchronized(m) {
  x = 1;
  y = 1;
}
```

```
synchronized(m) {
  x = 2;
  y = 2;
}
```

```
synchronized(m) {
  r1 = x;
  r2 = y;
}
```

Backup: Consensus, Example 4.2



```
synchronized(m) {      synchronized(m) {      synchronized(m) {
  x = 1;                 x = 2;                 r1 = x;
  y = 1;                 y = 2;                 r2 = y;
}                      }                      }
```

If we only have HB, $(1, 2)$ is possible:

$$\begin{tikzcd}
w(x,1) \arrow[r, "\text{hb}"] & w(y,1) \arrow[r, "\text{hb}"] & UL(m)\ \square \arrow[d, bend left] & & \\
 & & \square\ L(m) \arrow[r, "\text{hb}"] & r(x) :? \arrow[r, "\text{hb}"] & r(y) :? \\
w(x,2) \arrow[r, "\text{hb}"] & w(y,2) \arrow[r, "\text{hb}"] & UL(m)\ \square \arrow[u, bend right] & &
\end{tikzcd}$$

Backup: Optimized #4, Try 1

```
V v;
Supplier<V> s;

public V get() {
  V lv = v;
  if (lv == null && s != null) {
    synchronized (this) {
      if (s != null) {
        v = lv = s.get();
        s = null;
      }
    }
  }
  return lv;
}
```

What if we rely on `V` being safely constructed? That would allow us to drop `volatile` here, right?

Backup: Optimized #4, Try 1

```
V v;
Supplier<V> s;

public V get() {
  V lv = v;
  if (lv == null && s != null) {
    synchronized (this) {
      if (s != null) {
        v = lv = s.get();
        s = null;
      }
    }
  }
  return lv;
}
```

What if we rely on `V` being safely constructed? That would allow us to drop `volatile` here, right?

No! Here is where it goes downhill returning `null`:

$$w(v, V) \xrightarrow{\text{hb}} w(s, null)$$

$$r(v) : null \xrightarrow{\text{hb}} r(s) : null$$

Backup: Optimized #4, Try 2

```
V v;
Supplier<V> s;

public V get() {
  V lv = v;
  if (lv == null) {
    synchronized (this) {
      if (s != null) {
        v = lv = s.get();
        s = null;
      }
    }
  }
  return lv;
}
```

What if we rely on `V` being safely constructed? That would allow us to drop `volatile` here, right?

Backup: Optimized #4, Try 2

```
V v;
Supplier<V> s;

public V get() {
  V lv = v;
  if (lv == null) {
    synchronized (this) {
      if (s != null) {
        v = lv = s.get();
        s = null;
      }
    }
  }
  return lv;
}
```

What if we rely on `V` being safely constructed? That would allow us to drop `volatile` here, right?

Now it is fine, follows the benign race pattern.