JCStress Workshop

or, «One Awful Java Concurrency Test After Another»

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.

Workshop Plan (Provisional)

- Part I: Basic Bits
  - JCStress: Why do it? How does it look?
  - JMM: Looking at basic examples

(Coffee Break)

- Part II: Advanced Bits
  - JCStress: How does it work? Why would you not do it manually?
  - JMM: Looking at advanced examples

(Breathing Exercises)

- Part III: Fun Bits *(optional)*
  - JCStress: Real JVM/JDK bugs discovered
  - Breakout: Discussions, Future Work, etc.

redhat.

Workshop Resources

Most code is available as runnable JCStress Samples:

```
# Setup ---------------------------------------------
git clone https://github.com/openjdk/jcstress/
cd jcstress
export JAVA_HOME=<path-to-jdk-11>
export PATH=$JAVA_HOME/bin:$PATH

# Build ---------------------------------------------
mvn clean install -DskipTests -T 1C

# Run -----------------------------------------------
java -jar jcstress-samples/target/jcstress.jar ...
```

Workshop Involvement

- JCStress Samples have «How to run this test» section with useful one-liners. `-h` shows some of the runner options that might be useful.
- You **could** run the samples during the workshop. However, you **are not required** to do so. Note interesting cases for yourself, and then run them later.
- If you have questions, ask them in Telegram chat for the talk. I can then modify the samples on the fly, or refer to other samples.

Part I. Basic Bits

JCStress Basics

JCStress Basics: Historical Context

- Circa 2013 ($\approx$ JDK 8), we suddenly realized there are no regular concurrency tests that ask hard questions about JMM conformance
- Attempts to contribute JMM tests to JCK were futile: probabilistic tests
- JVMs are notoriously awkward to test: many platforms, many compilers, many compilation and runtime modes, dependence on runtime profile

JCStress Basics: Scoping

JCStress is ${value} testing framework

Where ${value} is:

- *Java*: targets low-level JVM work, but touches HW too
- *empirical*: distrust lower layers (JVM, OS, HW) are sane
- *combinatorial*: tests many configurations of the same test
- *experimental*: fluid implementation to fit new techiques

redhat.

JCStress Basics: Empirical, Not Model Checking

If you want model checking, go for Lincheck workshop:

Day 1. June 15

Time UTC+03:00	Lecture
18:45 **Track 2**	Workshop. Lincheck: Testing concurrency on the JVM **Maria Sokolova** *JetBrains* *#concurrency #jvm #model-checking #testing* EN
20:30 **Track 2**	Workshop. Lincheck: Testing concurrency on the JVM (part 2) **Maria Sokolova** *JetBrains* *#concurrency #jvm #model-checking #testing* EN

redhat.

JCStress Basics: Prior Art

1. Java Compatibility Kit (JCK)
   - Developed by Oracle, JCP
   - Tests Java Language Specification, Chapter §17
   - Limited to normative clauses
2. JSR166 TCK
   - Developed by Doug Lea et al.
   - Tests `java.util.concurrent.*`
3. Litmus/DIY
   - Developed by Peter Sewell et al.[1]
   - Tests hardware semantics

[1] http://www.cl.cam.ac.uk/~pes20/

redhat

Early Attempts: Concurrency Testing

Concurrency bugs are (data) race (condition) bugs

Need to create a *controllable race condition*:

- large enough, so that threads meet
- small enough, so that we can trust the results
- fast enough, so that timings are not masked

Unfortunately, naive tests do not check all these boxes...

Early Attempts: Try #1

```
volatile int v;

void doTest() {
  Thread t1 = new Thread(() -> v++);
  Thread t2 = new Thread(() -> v++);
  t1.start(); t2.start();
  t1.join(); t2.join();

  Assert.assertTrue(2, v);
}
```

«Collision Window» is far too small to capture anything interesting.

redhat.

Early Attempts: Try #2

```
volatile int v;
final CountDownLatch l = new CountDownLatch(2);

void doTest() {
  Thread t1 = new Thread(() -> l.countDown(); l.await(); v++);
  Thread t2 = new Thread(() -> l.countDown(); l.await(); v++);
  t1.start(); t2.start();
  t1.join(); t2.join();

  Assert.assertTrue(2, v);
}
```

Threads still rarely meet; synchronization costs dominate.

Current Form: Idea At A Glance

```
@JCStressTest
@State
public class MyTest {
  volatile int v;
  @Actor void actor1(I_Result r) { r.r1 = v++; }
  @Actor void actor2(I_Result r) { r.r2 = v++; }
}
```

- Large array of single-use state-bearing objects
- Actors access state objects under race
- Actors save their observations in provided storage
- Test infrastructure counts the observations

Current Form: Idea At A Glance

```
@JCStressTest
@State
public class MyTest {
  volatile int v;
  @Actor void actor1(I_Result r) { r.r1 = v++; }
  @Actor void actor2(I_Result r) { r.r2 = v++; }
}
```

End result: counted (r1, r2) outcomes

RESULT	SAMPLES	FREQ	EXPECT	DESCRIPTION
1, 1	46,946,789	10.1%	Interesting	...
1, 2	110,240,149	23.8%	Acceptable	...
2, 1	306,529,420	66.1%	Acceptable	...

redhat

Current Form: JCStress Examples

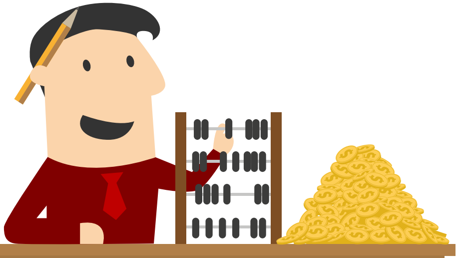
If you are reading the slides offline,
we are about to look through these examples:

https://github.com/openjdk/jcstress/tree/master/jcstress-samples/src/main/java/org/openjdk/jcstress/samples/api/

redhat

Current Form: Examples

Switching to JCStress in 3... 2... 1...



JMM Basics

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?
  }
}
```


redhat

Spec: Good Spec Is A Balance

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

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

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

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


redhat.

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!

redhat

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;
}
```

⇒

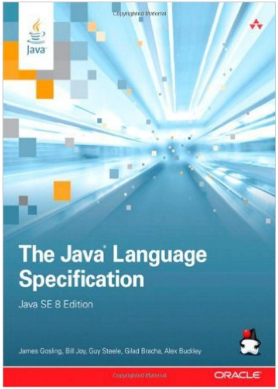
```
m:
...prolog...
mov $2$, (mem)
mov $2$, %rax
...epilog...
ret
```

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

redhat.

JMM: Problem

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



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

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.)

- 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:

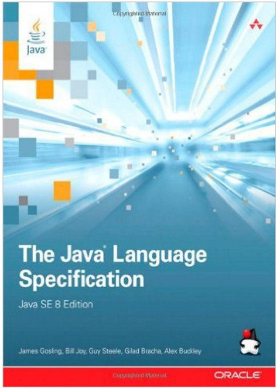
1. C_i is a subset of A_i

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: Problem

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



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 by

- P - a program
- A - a set of actions
- po - program order, whic… 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.
  - $O \subseteq O' \subseteq A$
  - $| O' | \geq k$

redhat

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.

redhat.

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

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

Orders:

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

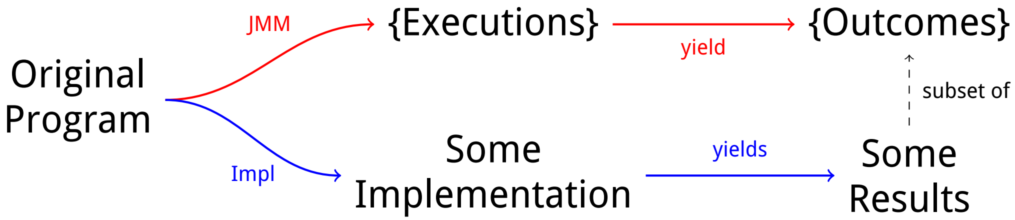
Consistency rules:

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

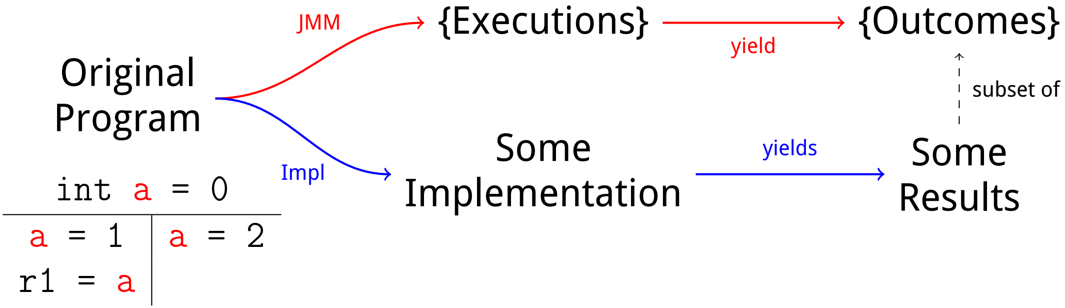
JMM: Umm...

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

JMM: Why?


Original Program
JMM
{Executions}
yield
{Outcomes}
subset of
Impl
Some Implementation
yields
Some Results

JMM: Why?

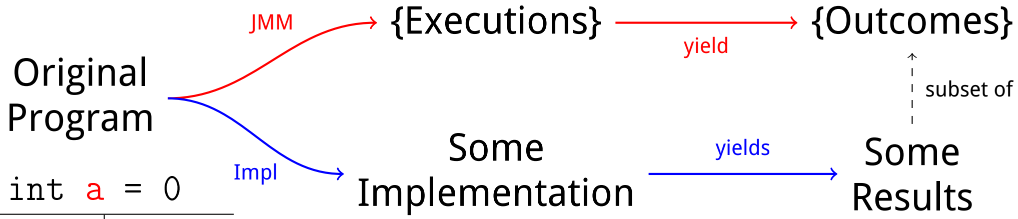


JMM
{Executions}
yield
{Outcomes}
Original Program
subset of
Impl
Some Implementation
yields
Some Results
int a = 0
a = 1
r1 = a
a = 2

JMM: Why?

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

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




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

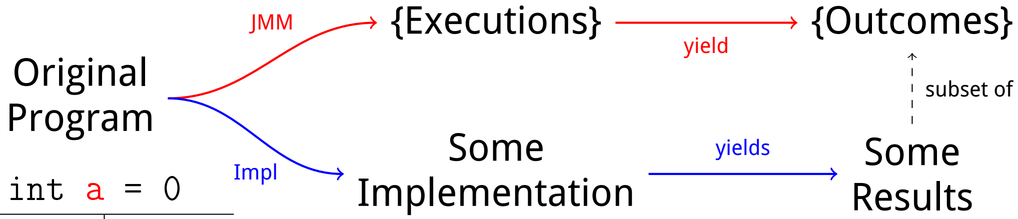

redhat

JMM: Why?

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

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

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



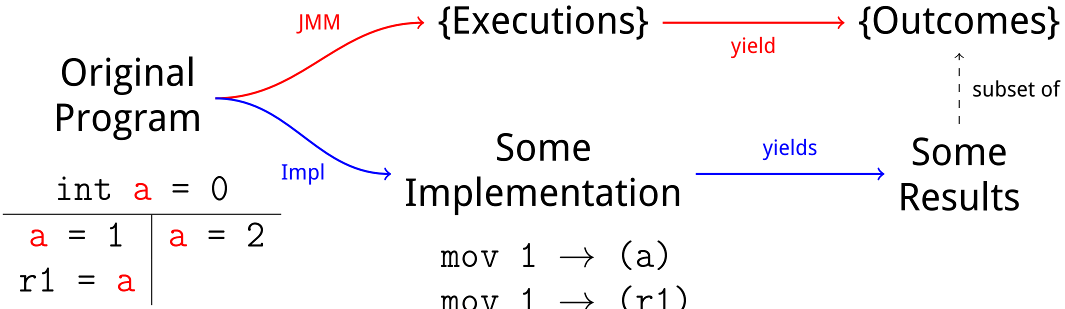

```
int a = 0
------------------
a = 1    | a = 2
r1 = a   |
```

JMM: Why?

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

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

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



JMM: Why?

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

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

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

Original Program —JMM→ {Executions} —yield→ {Outcomes}

Original Program —Impl→ Some Implementation —yields→ Some Results

Some Results —subset of→ {Outcomes}

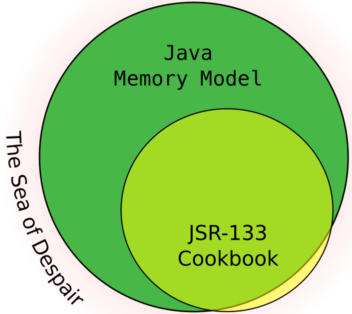
```
int a = 0
-----------------
a = 1    | a = 2
r1 = a   |
```

```
mov 1 → (a)
mov 1 → (r1)
```

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

redhat

JMM: Takeaway #1: Studying Implementations

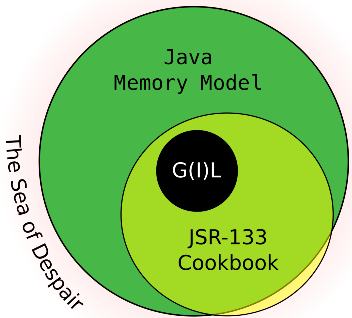


Implementations are allowed to generate the **subset** of allowed outcomes, not all of them

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

redhat.

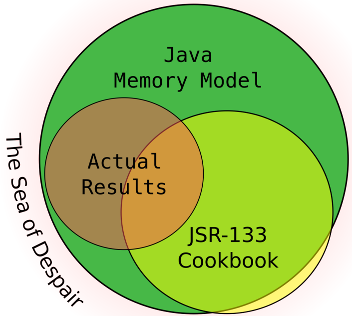
JMM: Takeaway #1: Studying Implementations



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: Takeaway #2, Interpreting Empirical Tests



The Universe is under no obligation to show you **all** of the outcomes allowed by spec or implementation

- «Not reproducible» does not mean «Not possible»
- Frequency is a *soft* evidence on possibility

JMM: JEP 188

There used to be the JMM Update JEP:

`https://openjdk.java.net/jeps/188`

- Improved formalization
- JVM coverage
- Extended scope
- C11/C++11 compatibility
- Implementation guidance
- ...

JMM: JEP 188

There used to be the JMM Update JEP:

`https://openjdk.java.net/jeps/188`

- Improved formalization
- JVM coverage
- Extended scope
- C11/C++11 compatibility
- Implementation guidance
- ...

STALLED

JMM: VarHandles

java.lang.Object
    java.lang.invoke.VarHandle

```
public abstract class VarHandle
extends Object
```

A VarHandle is a dynamically strongly typed reference to a variable, or to a parametrically-defined family of variables, including static fields, non-static fields, array elements, or components of an off-heap data structure. Access to such variables is supported under various *access modes*, including plain read/write access, volatile read/write access, and compare-and-swap.

• • • • • •

Access modes control atomicity and consistency properties. *Plain* read (`get`) and write (`set`) accesses are guaranteed to be bitwise atomic only for references and for primitive values of at most 32 bits, and impose no observable ordering constraints with respect to threads other than the executing thread. *Opaque* operations are bitwise atomic and coherently ordered with respect to accesses to the same variable. In addition to obeying Opaque properties, *Acquire* mode reads and their subsequent accesses are ordered after matching *Release* mode writes and their previous accesses. In addition to obeying Acquire and Release properties, all *Volatile* operations are totally ordered with respect to each other.

redhat

JMM: Overview

Java 8	Java 9	Defined	Atomic	Coherence	Causality	Consensus	Resilient
plain	VH Plain	Y	≈	N	N	N	N
-	VH Opaque	Y	Y	Y	N	N	N
-	VH Acq/Rel	Y	Y	Y	Y	N	N
volatile	VH SeqCst	Y	Y	Y	Y	Y	N
final	-	Y	≈	≈	N	N	Y

redhat.

JMM: JCStress Examples

If you are reading the slides offline,
we are about to look through these examples:

https://github.com/openjdk/jcstress/tree/master/jcstress-samples/src/main/java/org/openjdk/jcstress/samples/jmm/basic

Data Races: Definitions

- **Conflict:** at least 2 threads accessing the same variable, and at least 1 thread is writer
  - Concurrent only-readers are fine
  - Write-write conflicts are fun

Data Races: Definitions

- **Conflict**: at least 2 threads accessing the same variable, and at least 1 thread is writer
  - Concurrent only-readers are fine
  - Write-write conflicts are fun
- **Data Race**: a **conflict** that is not ordered by synchronization
  - Key problem in low-level concurrency
  - SC-DRF: Sequential Consistency for Data Race Freedom

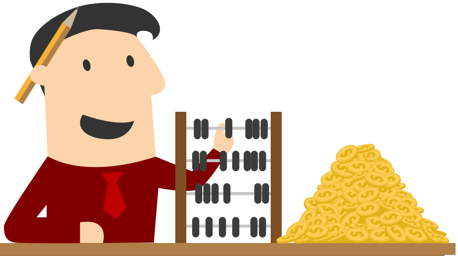
Data Races: Definitions

- **Conflict**: at least 2 threads accessing the same variable, and at least 1 thread is writer
  - Concurrent only-readers are fine
  - Write-write conflicts are fun
- **Data Race**: a **conflict** that is not ordered by synchronization
  - Key problem in low-level concurrency
  - SC-DRF: Sequential Consistency for Data Race Freedom
- **Race Condition**: system behavior is dependent on the timing of events
  - Not a problem, *unless* some behaviors are undesirable

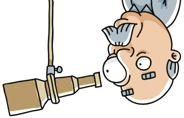
redhat

Data Races: Examples

Switching to JCStress in 3... 2... 1...



Data Races: Takeway, #1



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

Data Races: Takeaway #2

1. Data race behavior is still somewhat deterministic
   - Racy reads are stronger than in other languages
   - Weird stuff still happens, but not completely catastrophic
   - *(This is what allows JCStress to even exist)*

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;
}
```

redhat

Data Races: Overview

Java 8	Java 9	Defined	Atomic	Coherence	Causality	Consensus	Resilient
plain	VH Plain	Y	≈	N	N	N	N
-	VH Opaque	Y	Y	Y	N	N	N
-	VH Acq/Rel	Y	Y	Y	Y	N	N
volatile	VH SeqCst	Y	Y	Y	Y	Y	N
final	-	Y	≈	≈	N	N	Y

Access Atomicity: Definition

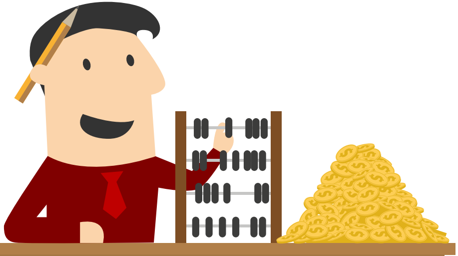
For any built-in type T:

```
T t = V1;

@Actor void actor1() {
  t = V2;
}

@Actor void actor2(T_Result r) {
  r.r1 = t;
}
```

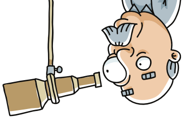
$$\mathrm{r1} \in \{V1, V2\}$$

Access Atomicity: Examples

Switching to JCStress in 3... 2... 1...



Access Atomicity: Takeaway



1. Most built-in types are access atomic
   - Almost all are naturally aligned
   - Unless 32-bit JVMs are present

2. Doing unnatural accesses break atomicity again
   - `ByteBuffer`-s «compound» operations
   - `Unsafe` «compound» operations

3. Larger types would break access atomicity again
   - Watch out for value/inline/primitive types

redhat.

Access Atomicity: Overview

Java 8	Java 9	Defined	Atomic	Coherence	Causality	Consensus	Resilient
plain	VH Plain	Y	≈	N	N	N	N
-	VH Opaque	Y	Y	Y	N	N	N
-	VH Acq/Rel	Y	Y	Y	Y	N	N
volatile	VH SeqCst	Y	Y	Y	Y	Y	N
final	-	Y	≈	≈	N	N	Y

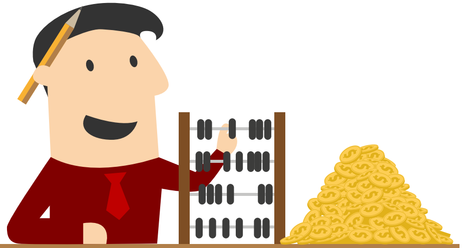
Coherence: Definition

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

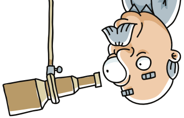
- Most hardware gives this for free
- Most optimizers give up on this by default (i.e. do not track the order of reads)

Coherence: Examples

Switching to JCStress in 3... 2... 1...



Coherence: Takeaway



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: Overview

Java 8	Java 9	Defined	Atomic	Coherence	Causality	Consensus	Resilient
plain	VH Plain	Y	≈	N	N	N	N
-	VH Opaque	Y	Y	Y	N	N	N
-	VH Acq/Rel	Y	Y	Y	Y	N	N
volatile	VH SeqCst	Y	Y	Y	Y	Y	N
final	-	Y	≈	≈	N	N	Y

redhat.

Causality: Definition

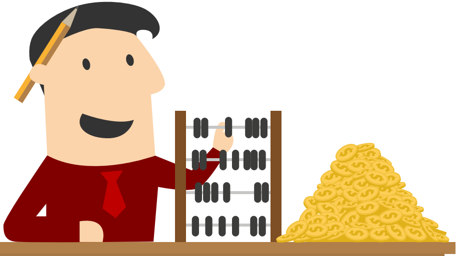
Causality:
If **A** happened, then **B** happened too.

- By far the most **basic** guarantee made by most memory models, extremely hard to accept as the guiding principle
- The cornerstone of most (all?) distributed consistency models

redhat

Causality: Examples

Switching to JCStress in 3... 2... 1...

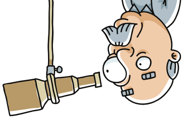


Causality: Safe Publication

```
volatile int v;

x = ...;                        acquire  if (v == 1) {
y = ...;                                   int lx = x;
z = ...;            «send»                 int ly = y;
v = 1;   release                           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



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

redhat.

Causality: Overview

Java 8	Java 9	Defined	Atomic	Coherence	Causality	Consensus	Resilient
plain	VH Plain	Y	≈	N	N	N	N
-	VH Opaque	Y	Y	Y	N	N	N
-	VH Acq/Rel	Y	Y	Y	Y	N	N
volatile	VH SeqCst	Y	Y	Y	Y	Y	N
final	-	Y	≈	≈	N	N	Y

redhat

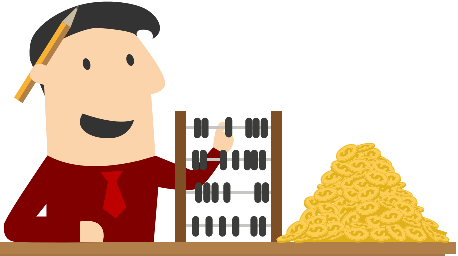
Consensus: Definition

Consensus:
Momentary agreement among threads about program state

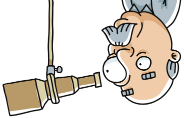
- There are different powers of consensus, but even the most basic Consensus-1 is useful.
- Consensus is mostly about *multiple* variables at once. Otherwise, coherence is enough...

Consensus: Examples

Switching to JCStress in 3... 2... 1...



Consensus: Takeaway #6



1. Consensus is good
   - Extremely useful to think about correctness
   - Avoid non-SC data races by going `volatile`
   - Sprinkle enough `volatiles` around your program, and it eventually becomes data-race-free! /s

2. Consensus is bad
   - Extreme costs to get SC in distributed systems
   - Most examples so far were fine with just Release/Acquire!
   - Relaxing SC is by far the most common optimization technique

Consensus: Overview

Java 8	Java 9	Defined	Atomic	Coherence	Causality	Consensus	Resilient
plain	VH Plain	Y	≈	N	N	N	N
-	VH Opaque	Y	Y	Y	N	N	N
-	VH Acq/Rel	Y	Y	Y	Y	N	N
volatile	VH SeqCst	Y	Y	Y	Y	Y	N
final	-	Y	≈	≈	N	N	Y

redhat.

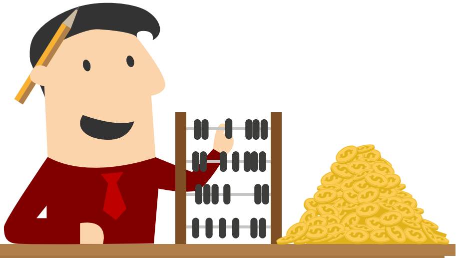
Finals: Idea

Finals:
«Declared immutable» fields, with additional semantics

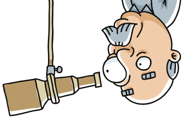
- `final`-s are very special: able to hide data races
- The defense-in-depth strategy for concurrent code: work even when external synchronization is broken

Finals: Examples

Switching to JCStress in 3… 2… 1…



Finals: Takeaway



1. Safe construction rule
   - No reason to omit `final` from effectively immutable fields[2]
   - This would be a building block for benign races

2. «Defense in Depth»: extra safety in the face of data races
   - Users are known to misread, misinterpret, misuse the docs
   - Most API-external objects need to be safely constructed

[2] Why JVM does not do it itself then, Aleksey?

Finals: Overview

Java 8	Java 9	Defined	Atomic	Coherence	Causality	Consensus	Resilient
plain	VH Plain	Y	≈	N	N	N	N
-	VH Opaque	Y	Y	Y	N	N	N
-	VH Acq/Rel	Y	Y	Y	Y	N	N
volatile	VH SeqCst	Y	Y	Y	Y	Y	N
final	-	Y	≈	≈	N	N	Y

redhat.

Benign Races: Canonical Form

There are cases when races are very useful:

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

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

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

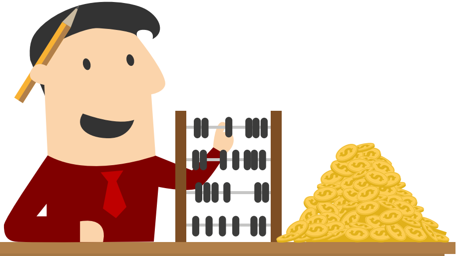
Benign Races: Real JDK Code

```
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;
  }
}
```

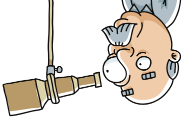

redhat.

Benign Races: Examples

Switching to JCStress in 3... 2... 1...



Benign Races: Takeaway



1. Benign races are useful, albeit dangerous tool
   - Allows avoiding synchronized ops on critical paths!
   - The only sane way to deliberately use races in the program?

2. Works only if three rules are followed:
   - single (racy) read
   - reliability check
   - recovery path that safely constructs

redhat

Benign Races: Overview

Java 8	Java 9	Defined	Atomic	Coherence	Causality	Consensus	Resilient
plain	VH Plain	Y	≈	N	N	N	N
-	VH Opaque	Y	Y	Y	N	N	N
-	VH Acq/Rel	Y	Y	Y	Y	N	N
volatile	VH SeqCst	Y	Y	Y	Y	Y	N
final	-	Y	≈	≈	N	N	Y

redhat

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

redhat.

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

redhat

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!

redhat.

Summing Up: Workshop Plan (Provisional)

- Part I: Basic Bits
  - JCStress: Why do it? How does it look?
  - JMM: Looking at basic examples

(Coffee Break)

- Part II: Advanced Bits
  - JCStress: How does it work? Why would you not do it manually?
  - JMM: Looking at advanced examples

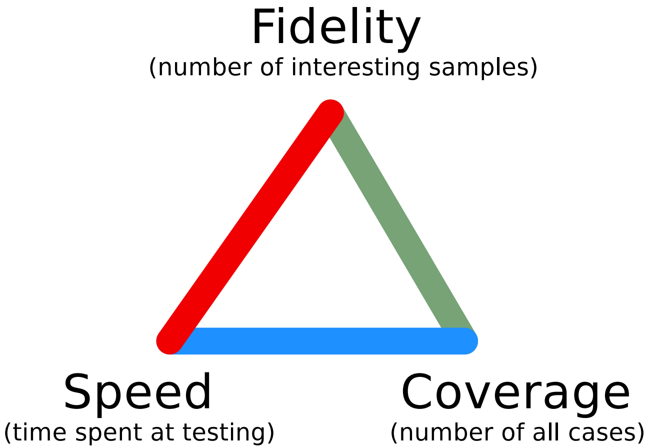
(Breathing Exercises)

- Part III: Fun Bits *(optional)*
  - JCStress: Real JVM/JDK bugs discovered
  - Breakout: Discussions, Future Work, etc.

Part II. Advanced Bits

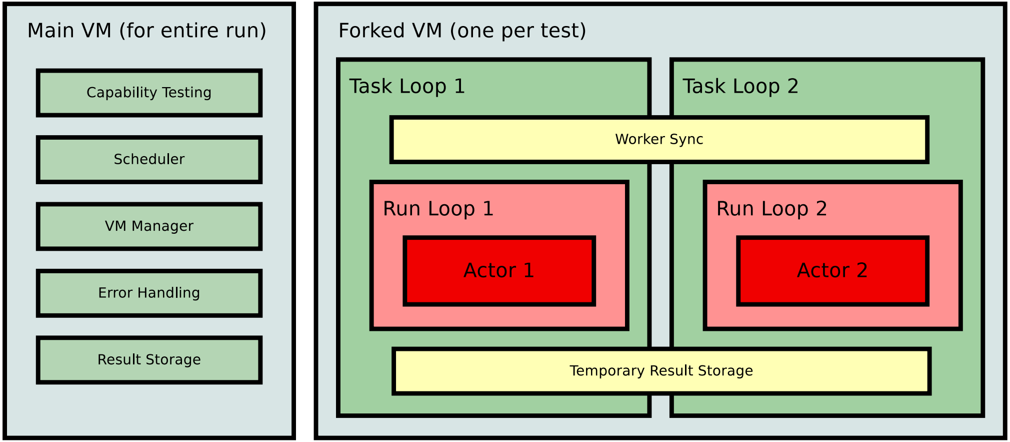
JCStress Advanced Topics

Architecture: Empirical Testing



redhat.

Architecture: JCStress Architecture

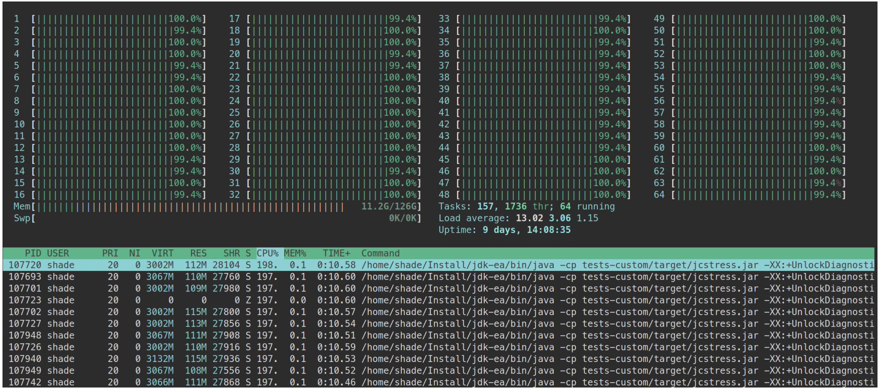


Architecture: Guiding Principles

- Test fidelity:
  - Generate most of the stuff at build time
  - Assume the role of optimizing compiler where possible
- Test speed:
  - Initialize most of the stuff in host VM
  - Force forked VMs to do absolute minimum
- Test coverage:
  - Run forked VMs in all the interesting modes
  - Run forked VMs in all the interesting affinities

Balancing Heap Sizes: Problem

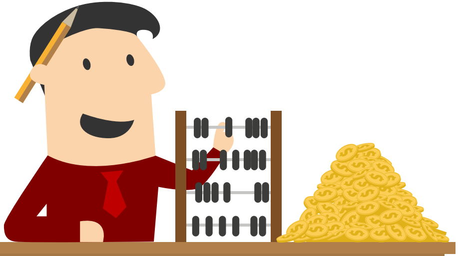
Speed/Fidelity:
JCStress normally runs tens/hundreds of JVMs concurrently



redhat

Balancing Heap Sizes: Examples

Switching to JCStress in 3... 2... 1...



Balancing Strides: Problem

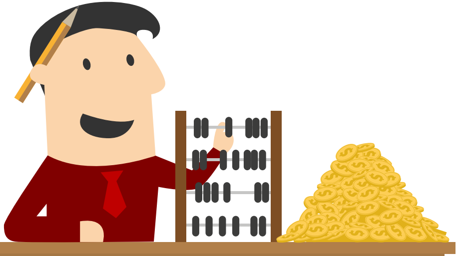
Fidelity:
When heap size is set, tests may OOME!

```
@JCStressTest
@State
public class MyTest {
  byte[] arr = new byte[1024*1024];

  @Actor void actor1(I_Result r) {
    int s = 0;
    for (byte b : arr) s += b;
    r.r1 = s;
  }
}
```


redhat

Balancing Strides: Examples

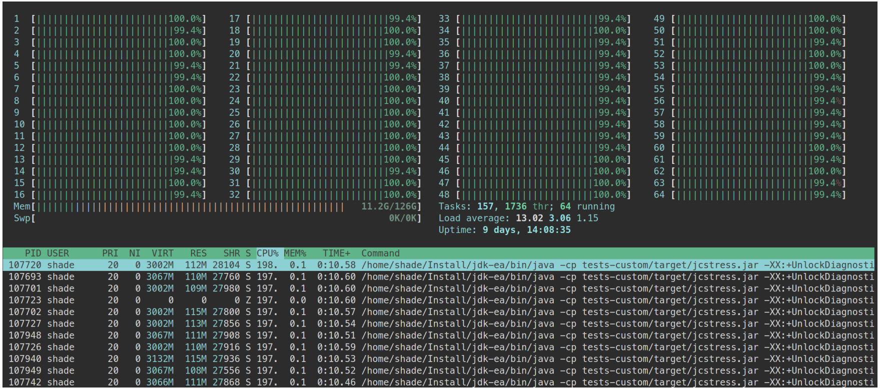
Switching to JCStress in 3... 2... 1...



Balancing Thread Counts: Problem

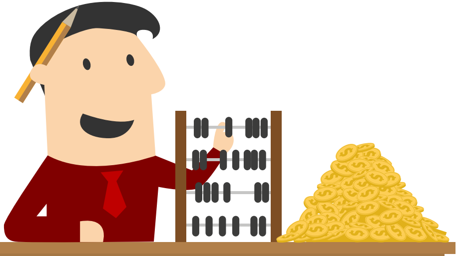
Speed/Fidelity:

JCStress normally runs tens/hundreds of JVMs concurrently



Balancing Thread Counts: Examples

Switching to JCStress in 3... 2... 1...



VM Modes: Problem

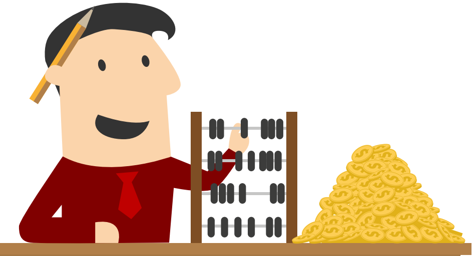
Coverage:
Some tests might fail only with some compilers

Hotspot VM has *at least* three ways to execute Java code:

- Interpreter (`-Xint`)
- C1 (baseline, client, `-XX:-TieredStopAtLevel=1`)
- C2 (optimized, server, `-XX:-TieredCompilation`)

VM Modes: Examples

Switching to JCStress in 3... 2... 1...



Split Compilation: Problem

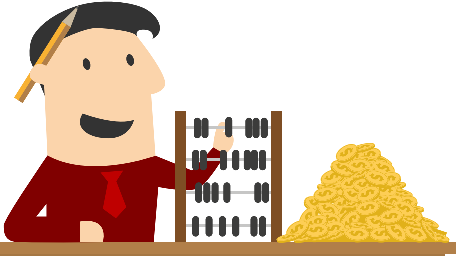
Coverage:

Some outcomes only manifest in odd compilation conditions

```
@Actor void actor1(II_Result r) {
  // Compile with C1, one barrier scheme
}

@Actor void actor2(II_Result r) {
  // Compile with C2, another barrier scheme
  ...
}
```

Split Compilation: Examples

Switching to JCStress in 3... 2... 1...



Compiler Fuzzing: Problem

Coverage:
Many outcomes are compiler-induced.

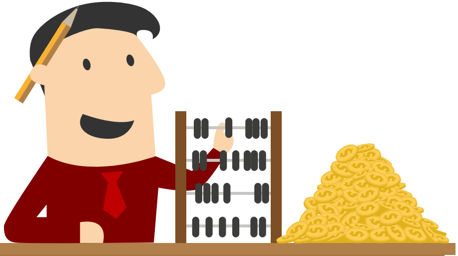
Hotspot VM provides a few randomizing flags:

- `-XX:+StressLCM`, `-XX:+StressGCM` – added for JCStress
- `-XX:+StressIGVN`, `-XX:+StressCCP` – added later
- Host VM probes which ones are available
- Host VM mixes these as the separate configuration

redhat.

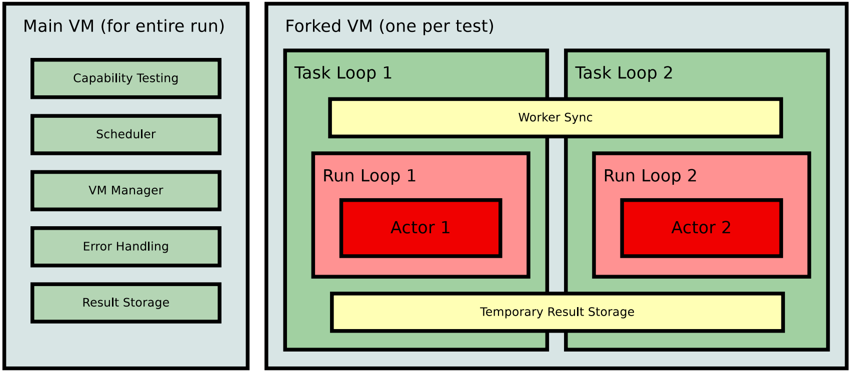
Compiler Fuzzing: Examples

Switching to JCStress in 3... 2... 1...



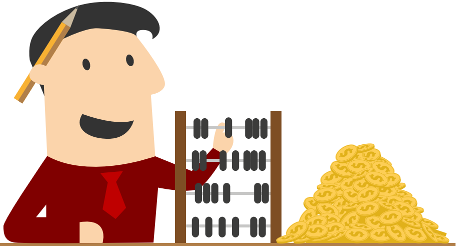
Loops: Problem

Fidelity:
JVMs like short and active loops.



Loops: Examples

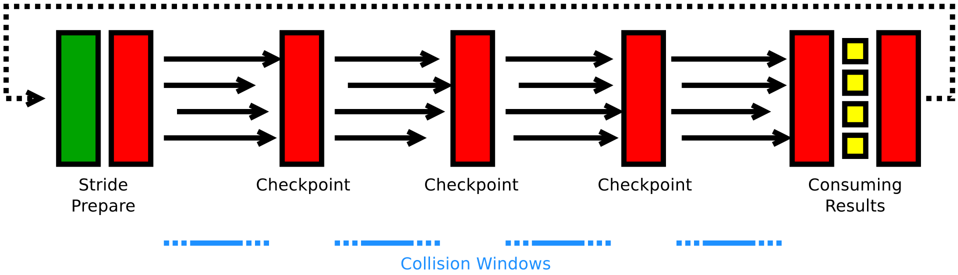
Switching to JCStress in 3... 2... 1...



Rendezvous: Problem

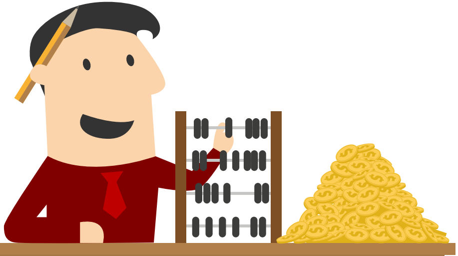
Fidelity:

Tests are rarely symmetric, so actors outpace each other



Rendezvous: Examples

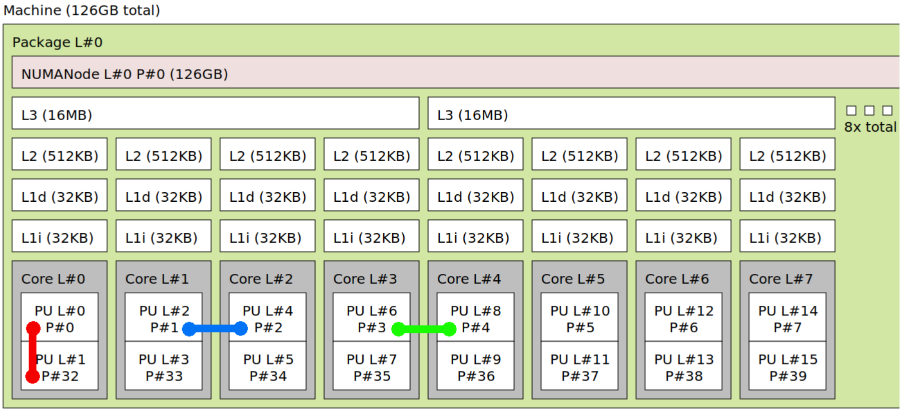
Switching to JCStress in 3... 2... 1...



Actor Affinity: Problem

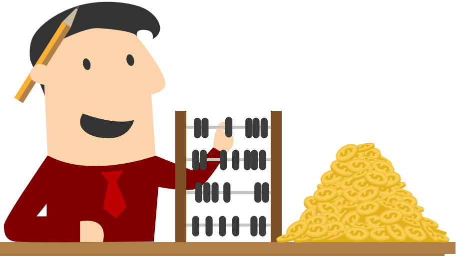
Coverage:

Some outcomes manifest on some test topologies



Actor Affinity: Examples

Switching to JCStress in 3... 2... 1...



Busy Waiting: Problem

Speed/Fidelity: Workers often have to wait actively.

```
# HARD
  0, 0     1,439,041,804    6.1%
  0, 1    10,878,860,414   46.6%
  1, 0    10,967,111,106   47.0%
  1, 1        23,684,276    0.1%
```

```
# THREAD_SPIN_WAIT
  0, 0     1,368,587,258    5.5%
  0, 1    11,458,577,433   46.4%
  1, 0    11,834,865,693   47.9%
  1, 1        23,107,536    0.1%
```

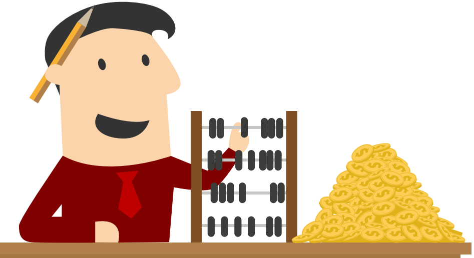
```
# THREAD_YIELD
  0, 0       728,512,816    2.8%
  0, 1    12,766,383,247   49.1%
  1, 0    12,448,519,780   47.9%
  1, 1        10,079,197    0.1%
```

```
# LOCKSUPPORT_PARK_NANOS
  0, 0        12,051,745    0.1%
  0, 1     3,876,507,240   50.1%
  1, 0     3,843,372,966   49.7%
  1, 1           179,409   <0.1%
```


redhat.

Busy Waiting: Examples

Switching to JCStress in 3… 2… 1…



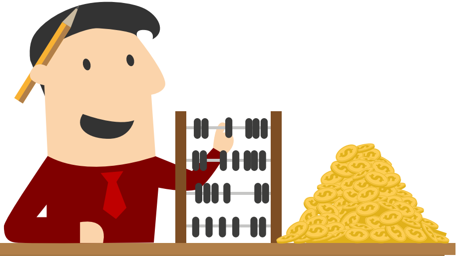
False Sharing: Problem

Fidelity:
Adjacent fields are false-shared,
decreasing the interesting outcomes frequency

```
volatile int a, b; // false shared?

@Actor void actor1(II_Result r) {
   ...
}

@Actor void actor2(II_Result r) {
   ...
}
```

False Sharing: Examples

Switching to JCStress in 3... 2... 1...



Reusing Objects: Problem

Speed:
API demands `@State` and `@Result` objects are one-shot

Without reuse:

RESULT	SAMPLES	FREQ
0, 0	64,113,081	5.9%
0, 1	522,012,200	47.8%
1, 0	504,635,282	46.2%
1, 1	1,608,557	0.1%

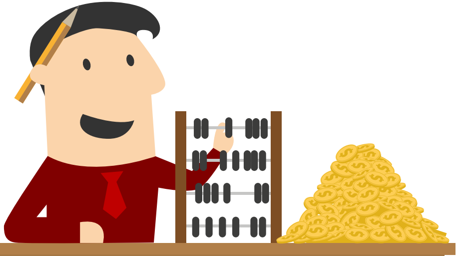
With reuse:

RESULT	SAMPLES	FREQ
0, 0	1,368,587,258	5.5%
0, 1	11,458,577,433	46.4%
1, 0	11,834,865,693	47.9%
1, 1	23,107,536	0.1%

redhat

Reusing Objects: Examples

Switching to JCStress in 3... 2... 1...



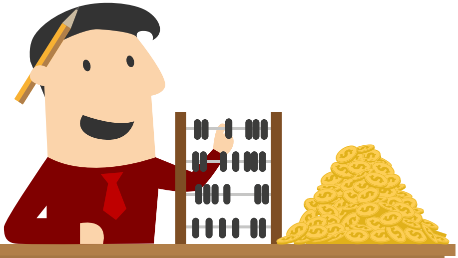
Null-Pointer Checks: Problem

Fidelity:
NPE semantics induces orderings

```
void actor1(II_Result r) {
  r.r1 = x; // null-check "r"
  r.r2 = x;
}
```

Null-Pointer Checks: Examples

Switching to JCStress in 3… 2… 1…



JMM Advanced Topics

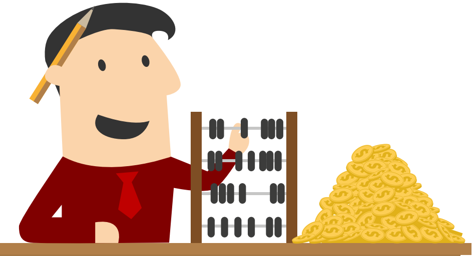
JMM Advanced Topics: JCStress Examples

If you are reading the slides offline,
we are about to look through these examples:

https://github.com/openjdk/jcstress/tree/master/jcstress-samples/src/main/java/org/openjdk/jcstress/samples/jmm/advanced

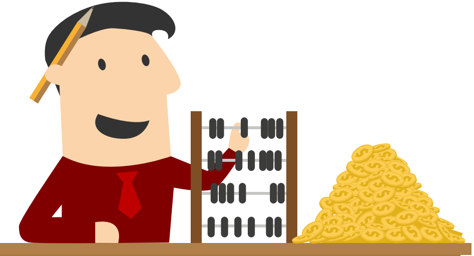
Synchronized Barriers: Examples

Switching to JCStress in 3… 2… 1…



Multi-Copy Atomicity: Examples

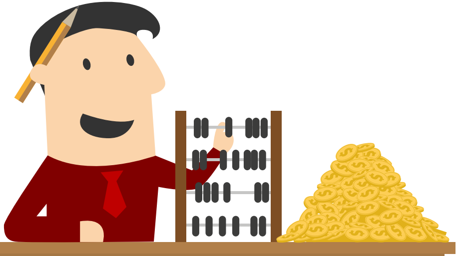
Switching to JCStress in 3... 2... 1...



redhat.

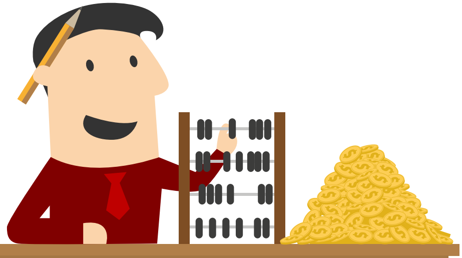
Losing Updates: Examples

Switching to JCStress in 3... 2... 1...



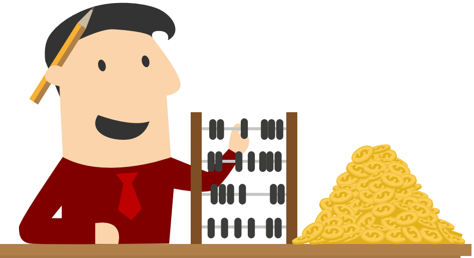
Misplaced Volatiles: Examples

Switching to JCStress in 3... 2... 1...



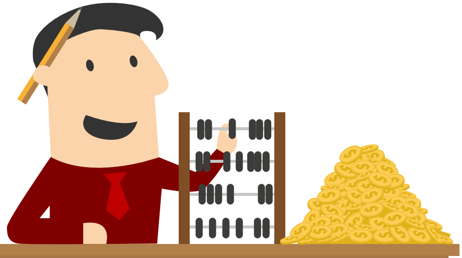
Semi-Synchronized: Examples

Switching to JCStress in 3... 2... 1...



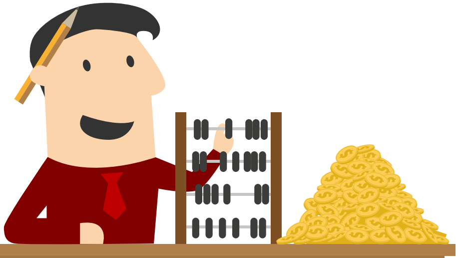
Volatile Arrays: Examples

Switching to JCStress in 3... 2... 1...



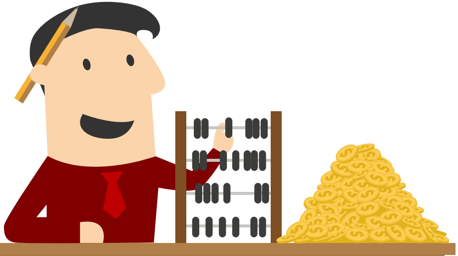
Acquire/Release Orders Wrong: Examples

Switching to JCStress in 3... 2... 1...



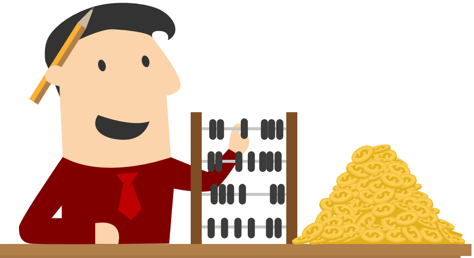
Synchronized «Barriers»: Examples

Switching to JCStress in 3... 2... 1...



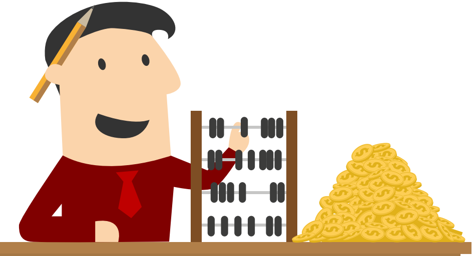
Volatile «Barriers»: Examples

Switching to JCStress in 3... 2... 1...



Volatile != Final: Examples

Switching to JCStress in 3… 2… 1…



Summing Up: Takeaway

- JMM guarantees are weaker than a single implementation might show

- You have to code to JMM rules, not to implementation behavior!

Part III. Extreme Bits

A Taste Of Real Bugs

Wrong Labels: Problem

```
int x; volatile int y;

@Actor void actor1() {
  x = 1;
  y = 1;
}

@Actor void actor2(II_Result r) {
  // int t = x;
  r.r1 = y;
  r.r2 = x;
}
```

`(1, 0)` is illegal under JMM rules, but breaks when prior copy «exists»!

redhat

Wrong Labels: C1 Bug[3]

C1 CSE bug, ignores `volatile` read:

```
t = x;
r1 = y;
r2 = x;
```

...so it coalesced the read:

```
t = x;
r1 = y;
r2 = t;
```

[3] https://bugs.openjdk.java.net/browse/JDK-7170145

Immortal Referents: Problem

```
final WeakReference<Object> ref = new WeakReference<>(obj);

@Actor void actor1() {
  while (ref.get() != null); // wait
}

@Actor void actor2(II_Result r) {
  ref.clear();
}
```

Expected behavior: test eventually terminates.
Actual behavior: test is stuck.

Immortal Referents: Compiler/GC Bug[4]

```
public abstract class Reference<T> {
  private T referent;
  ...
  public T get() { return referent; }
}
```

```
// just wait...
// ...a little
while (ref.get() != null);
```

⇒

```
if (ref.referent != null) {
  while (true); // burn !
}
```

[4] https://bugs.openjdk.java.net/browse/JDK-7190310

redhat.

Stuck Threads: Problem?

```java
@Actor void actor1() {
  while (!Thread.interrupted()); // wait
}

@Signal void signal(Thread actor1) {
  actor1.interrupt();
}
```

Expected behavior: test eventually terminates
Actual behavior: test eventually terminates!

redhat

Stuck Threads: Problem!

```
private boolean check() { return Thread.interrupted(); }

@Actor void actor1() {
  while (!check()); // wait
}

@Signal void signal(Thread actor1) {
  actor1.interrupt();
}
```

Actual behavior: test is stuck!

redhat

Stuck Threads: C2 Bug[6]

- `Thread.interrupted()` *used to* check a flag in the native

```
@HotSpotIntrinsicCandidate
private native boolean isInterrupted(boolean ClearInterrupted);
```

- Access was written in C2 IR; effectively a plain read, unless it is specifically written like a «volatile»
- Since then, it was rewritten to plain `volatile` field[5]

[5] https://bugs.openjdk.java.net/browse/JDK-8229516

[6] https://bugs.openjdk.java.net/browse/JDK-8003135

redhat

Eat My Shorts: Problem?

```
short s;

@Actor void actor1() {
  s = 0xFFFF;
}

@Actor void actor2(S_Result r) {
  r.r1 = s;
}
```

`short` is supposed to be atomic:
$r1 \in \{0x0000, 0xFFFF\}$, and it is indeed the case.

redhat

Eat My Shorts: Problem!

```
short s;

@Actor void actor1() {
  s = 0xFFFF;
}

@Actor void actor2(BB_Result r) {
  short t = s;
  r.r1 = (byte)((t >> 0) & 0xFF);
  r.r2 = (byte)((t >> 8) & 0xFF);
}
```

Expected: $(0x00, 0x00), (0xFF, 0xFF)$

Actual: $(0x00, 0x00), (0xFF, 0xFF), (0x00, 0xFF), (0xFF, 0x00)$

Eat My Shorts: Gradual Graph Rewrite

```
// Original code
short t = short_load(s.x);
r.r1 = byte_store(and(shift(t, 0), 0xFF)));
r.r2 = byte_store(and(shift(t, 8), 0xFF)));

// First round of simplifications
short t = short_load(s.x);
r.r1 = byte_store(t);
r.r2 = byte_store(shift(t, 8));

// Final round of simplifications
r.r1 = byte_store(unsigned_short_load(s.x));
r.r2 = byte_store(shift(signed_short_load(s.x), 8));
```


redhat

Eat My Shorts: C2 Bug[7]

```
short t = s.x;
r.r1 = (byte) ((t >> 0) & 0xFF);
r.r2 = (byte) ((t >> 8) & 0xFF);
```

Compiles to:

```
movzwl 0xc(%rdx), %r11d    ; read s.x
mov    %r11b,0xc(%rcx)     ; store r.r1
movswl 0xc(%rdx), %r10d    ; read s.x again!
shr    $0x8,%r10d          ; shift
mov    %r10b,0xd(%rcx)     ; store r.r2
```

[7] https://bugs.openjdk.java.net/browse/JDK-8000805

redhat

Volatile Clash: Problem

```
volatile double d = 0.0D;

@Actor void actor1() {
  d = Double.toRawLongBits(-1L);
}

@Actor void actor2(D_Result r) {
  r.r1 = d;
}
```

Expected: $r1 \in \{0x0...0, 0xF...F\}$
Actual: that, plus garbage!

redhat

Volatile Clash: Native Unsafe Code

```
#define GET_FIELD_VOLATILE(obj, offset, type_name, v) \
oop p = JNIHandles::resolve(obj); \
type_name v = OrderAccess::load_acquire( \
  (volatile type_name*) index_oop_from_field_offset_long(p, offset));
```

`Unsafe_GetDoubleVolatile()` compiles to:

```
mov     0x18(%esp),%ebp
add     %ebp,%eax
; field offset in %eax
fldl    (%eax)
fstpl   0x18(%esp)
```


redhat

Volatile Clash: Native Code Bug[8]

```
#define GET_FIELD_VOLATILE(obj, offset, type_name, v) \
oop p = JNIHandles::resolve(obj); \
type_name v = OrderAccess::load_acquire( \
  (volatile type_name*) index_oop_from_field_offset_long(p, offset));
```

`Unsafe_GetDoubleVolatile()` actually compiled to:

```
mov    0x4(%eax),%edx     ; WHAT
mov    (%eax),%eax        ; WHAT
mov    %eax,0x20(%esp)    ; THE
mov    %edx,0x24(%esp)    ; THE
fldl   0x20(%esp)         ; HELL
fstpl  0x18(%ebx)         ; HELL
```

[8] https://bugs.openjdk.java.net/browse/JDK-8016538

Power Dekker: Problem

```
volatile int x, y;

@Actor void actor1(II_Result r) {
  x = 1;
  r.r1 = y;
}

@Actor void actor2(II_Result r) {
  y = 1;
  r.r2 = x;
}
```

SC executions: $(r1, r2) \notin \{(0,0)\}$

Power Dekker: Piece 1: Optimizing For Hardware

- POWER ISA has a lot of registers
  - *(take that, lousy x86)*

- Hotspot PPC port is capitalizing on that
  - Profitable to schedule loads as soon as possible
  - ...unless something prevents it

redhat.

Power Dekker: Piece 2: Optimizing Barriers Bug[9]

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

This produces, roughly:

MB → store(x, 1) → MB → load(r1, y) → MB

Barrier optimization code mistakenly removes the barrier after volatile store, because it thinks there is a leading membar before volatile load:

MB → store(x, 1) → load(r1, y) → MB

[9] https://bugs.openjdk.java.net/browse/JDK-8007898

AArch64 Zero barriers: Problem

```
int x; volatile long y;

@Actor void actor1() {
  x = 1;
  y = 1;
}

@Actor void actor2(IJ_Result r) {
  r.r1 = y;
  r.r2 = x;
}
```

Observing $(1, 0)$ on AArch64!

redhat

AArch64 Zero barriers: Barrier Selection Bug[10]

```
#ifdef ARM
  #define LIGHT_MEM_BARRIER __kernel_dmb()
#else // ARM
  #ifdef PPC
    #define LIGHT_MEM_BARRIER __asm __volatile ("lwsync":::"memory")
  #else // PPC
    #ifdef ALPHA
      #define LIGHT_MEM_BARRIER  __sync_synchronize()
    #else // ALPHA
      #define LIGHT_MEM_BARRIER  __asm __volatile ("":::"memory")
    #endif // ALPHA
  #endif // PPC
#endif // ARM
```

[10] https://bugs.openjdk.java.net/browse/JDK-8253284

redhat.

ARM32 Zero barriers: Problem

Actually, JCStress would not even pass initialization:

```
$ java -jar jcstress-tests-all-20200917.jar
Java Concurrency Stress Tests
---------------------------------------------------------------------------------
Rev: ad66703e2ed0, built by buildbot with 11.0.5-testing at 2020-09-1

...
Caused by: java.lang.OutOfMemoryError:
  Cannot reserve 8192 bytes of direct buffer memory
  (allocated: 0, limit: -5290888278393214624)
...
```

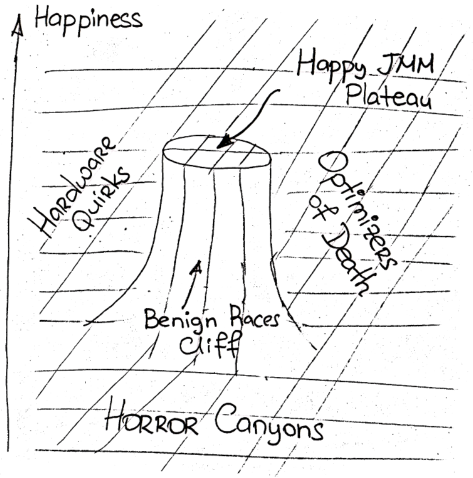
ARM32 Zero barriers: Atomicity Bug[11]

```
static void atomic_copy64(const volatile void *src,
                                   volatile void *dst) {
  jlong tmp;
  asm volatile ("ldrexd  %0, [%1]\n"
                : "=r"(tmp)
                : "r"(src), "m"(src));
  *(jlong *) dst = tmp;
}
```

[11] https://bugs.openjdk.java.net/browse/JDK-8253464

Conclusions

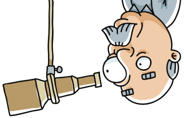
Conclusions: In One Picture



redhat

Backup: JMM Arguments

Data Races: Formal Argument

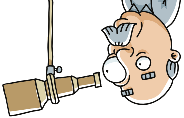


```
class M { ... }
M m;
```

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

redhat.

Data Races: Formal Argument



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

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

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

Data Races: Formal Counter-Argument

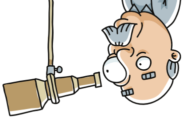
Can't compiler «inline» the local variable?

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

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

redhat

Data Races: Formal 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!

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

redhat

Data Races: 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

Data Races: 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);
```

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

redhat.

Data Races: PO And Trans

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);
```

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

redhat

Data Races: PO And Trans

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

Original pr[ogram:]

```
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);
```

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

Data Races: PO And Transformations

Original program:

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

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

Transformed program:

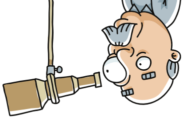
```
r1 = (m
r2 = (m
```

$, null)$

$) : *$

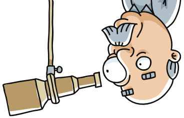
PO consistency:
Original program has single read?
Relatable executions also have single read!

Coherence: Example 2.1



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

Coherence: Example 2.1



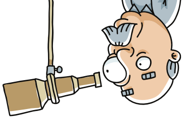
`int x;`

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

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

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

Coherence: Example 2.1



```
int x;
```

Thread 1	Thread 2
`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!

redhat

Coherence: 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

Coherence: 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!**

redhat

Coherence: 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 $\xrightarrow{so}$

redhat

Coherence: SO – Synchronization Order

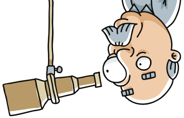
SO covers all *synchronization actions*:
volatile read/write, lock/unlock, etc.

- **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!

redhat.

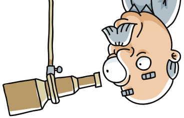
Coherence: Example 2.2



```
volatile int x;
```

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

Coherence: Example 2.2



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

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

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

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

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

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

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

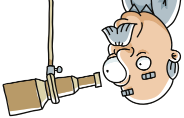
- **HB** is a partial order
  (Translation: not everything is connected)
- **HB consistency**: reads observe either:

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

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

redhat

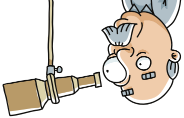
Causality: Example 3.1



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

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

Causality: Example 3.1

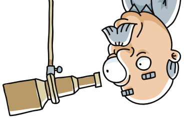


```
int x;
volatile int y;
-----------------------
x = 1;  | r1 = y;
y = 1;  | r2 = x;
```

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;
```

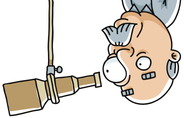
`x = 1;`	`r1 = y;`
`y = 1;`	`r2 = x;`

Racy subclass:

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

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

Causality: Example 3.1



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

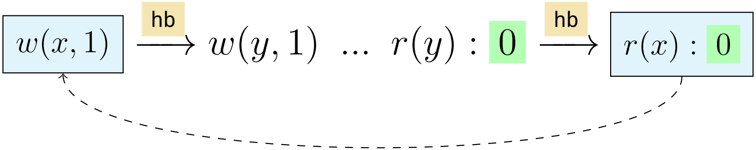
Non-racy subclass:

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

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

Causality: Look Closer, #1

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



not required to see this

redhat

Causality: Look Closer, #2

This violates HB consistency:

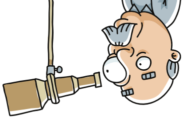
$$w(x,1) \xrightarrow{hb} w(y,1) \xrightarrow{hb} r(y) : 1 \xrightarrow{hb} r(x) : 0$$

should have seen this!

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

Causality: Example 3.2

Notice the order is different

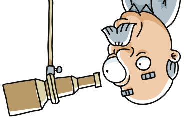


```
int x;
volatile int y;
```

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

redhat

Causality: Example 3.2



Notice the order is different

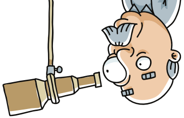
```
int x;
volatile int y;
-----------------------
y = 1;   | r1 = x;
x = 1;   | r2 = y;
```

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

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

redhat.

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`!

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 $(1, 0, 1, 0)$:

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

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

redhat

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!

redhat

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

redhat

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:

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

redhat.

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:

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

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

redhat

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:

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

redhat

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`

volatile $\notin$ final

final $\notin$ volatile

redhat

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`