
Java™
ORACLE

# Java Memory Model
# JSR 133

Vaibhav Choudhary (@vaibhav_c)
Java Platforms Team
https://blogs.oracle.com/vaibhav

Java
Your
Next
(Cloud)

# Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

# Agenda of the day …

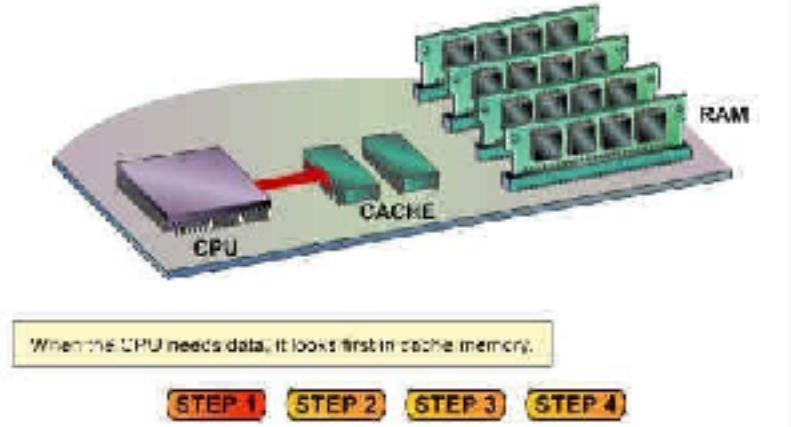
1. References
2. Threads
3. Locks
4. Volatile Variables
5. Data Races

# References

- Effective Java - By Joshua Bloch
- Concurrency in Practice - Brian Goetz
- JSR 133 - https://www.cs.umd.edu/users/pugh/java/memoryModel/
- JSR 133 FAQ - https://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html
- Chapter 17 of JLS - http://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html#jls-17.4

# Java Memory Model - Why to know

- All about thread talking to memory.
- Cache brings great performance but created great challenges as well.
- Write to processor is visible to other thread ?
- Does your processor exhibits strong memory model or weak memory model ?
- When to invalidate cache data ?
  - Correctness vs performance (old fight)
- Compiler brings extra complication.



# Reordering issue

- Conspiracy by Compiler, JIT, cache.
- Illusion as-if-serial semantics, because they see the code for optimization from single thread eyes.
- Inconsistent behavior if seen from more than one thread.

# Java Memory Model

- Write once, Run Anywhere - Tough to achieve with processor level dependency.
- Java memory model is designed and it is part of JLS.
- Best practices **must be followed.**

# Memory Reordering on various architecture

| Type | Alpha | ARMv7 | PA-RISC | POWER | SPARC RMO | SPARC PSO | SPARC TSO | x86 | x86 oostore | AMD64 | IA-64 | z/Architecture |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Loads reordered after loads | Y | Y | Y | Y | Y | | | | Y | | Y | |
| Loads reordered after stores | Y | Y | Y | Y | Y | | | | Y | | Y | |
| Stores reordered after stores | Y | Y | Y | Y | Y | Y | | | Y | | Y | |
| Stores reordered after loads | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y |
| Atomic reordered with loads | Y | Y | | Y | Y | | | | | | Y | |
| Atomic reordered with stores | Y | Y | | Y | Y | Y | | | | | Y | |
| Dependent loads reordered | Y | | | | | | | | | | | |
| Incoherent instruction cache pipeline | Y | Y | | Y | Y | Y | Y | Y | Y | | Y | |

Reference: https://en.wikipedia.org/wiki/Memory_ordering

# Re-ordering issue

```
Class Reordering {
  int x = 0, y = 0;
  public void writer() {
    x = 1;
    y = 2;
  }

  public void reader() {
    int r1 = y;
    int r2 = x;
  }
}
```

| Possible Value | r1 | r2 |
|---|---|---|
| | 2 | 1 |
| | 0 | 1 |
| | 2 | 0 |

# Understanding Memory Barrier

```
int  data=0;
boolean ready = false;
```

```
void method1() {
    while (!ready) {};
    // need a barrier
    assert data == 42;
}
```

```
void method1() {
    data = 42;
    // need a barrier
    ready = true;
}
```

# Kinds of memory barrier

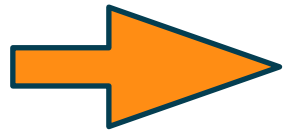
- Almost all processor supports coarse-grained barrier instruction called “Fence”.
- “Fence” guarantees that all the operations before fence will be strictly ordered before the operation after fence

- LoadLoad Barrier - Load1; Load-Load; Load2 means
  - Load1’s data is available before you access Load2’s data.
- StoreStore Barrier - Store1; Store-Store; Store2 means
  - Store1’s data are visible to other processor (flushed out) before Store2’s Data.

# Leads to “happened before” relationship

```
int  data=0;
boolean ready = false;
```

```
void synchronized method1() {
    while (!ready) {};
    assert data == 42;
}
```

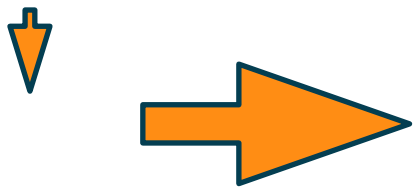


```
void synchronized method1() {
    data = 42;
    ready = true;
}
```

# volatile after JSR-133

```
int  data=0;
boolean volatile ready = false;
```

```
void synchronized method1() {
    while (!ready) {};
    assert data == 42;
}
```



```
void synchronized method1() {
    data = 42;
    ready = true;
}
```

# Volatile - will do it.

```
class VolatileExample {
  int x = 0;
  volatile boolean v = false;
  public void writer() {
    x = 42;
    v = true;
  }

  public void reader() {
    if (v == true) {
      //uses x - guaranteed to see 42.
    }
  }
}
```

# Work with final field

```
class FinalFieldExample {
  final int x;
  int y;
  static FinalFieldExample f;
  public FinalFieldExample() {
    x = 3;
    y = 4;
  }

  static void writer() {
    f = new FinalFieldExample();
  }

  static void reader() {
    if (f != null) {
      int i = f.x;  // 3 for sure
      int j = f.y;  // 4, not so sure
    }
  }
}
```

# Broken Double-check Singleton Pattern

```
// double-checked-locking - don't do this!

private static Something instance = null;

public Something getInstance() {
  if (instance == null) {
    synchronized (this) {
      if (instance == null)
        instance = new Something();
    }
  }
  return instance;
}
```

Details - http://www.javaworld.com/article/2074979/java-concurrency/double-checked-locking--clever--but-broken.html?page=2

# Unrevised Java Memory Model

- synchronized keyword
  - mutual exclusion
  - happened before
- volatile
  - visibility

**volatile doesn't establish a happened-before relationship**

**final values can change**

**many runtime optimizations where not allowed**

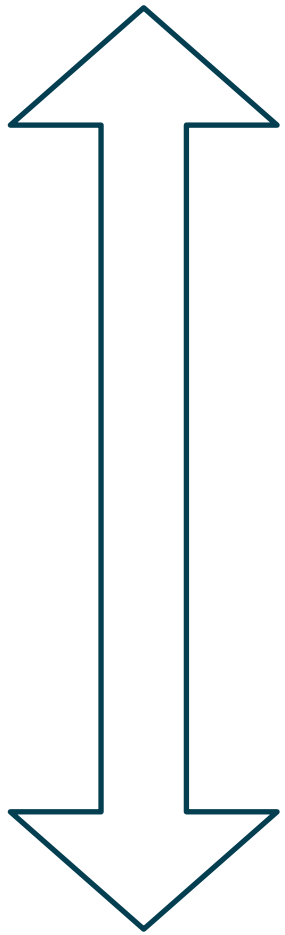
# Fix in Java 5 under JSR-133

- volatile **does** establish happen-before relationship.
- final values can’t change.
- Introduction of high level multi-threading library by Doug Lea and co.

- Read more - https://jcp.org/en/jsr/detail?id=133

# Avoid Low Level API

Abstract



java.util.concurrency, parallel streams

synchronized
- wait(), notify(), notifyAll()

Low Level

volatile, final …

# Breaking the spec

- Unexpected behavior on different architecture, OS, compiler.
- Intermittent failures.
- Always try to use high level concurrency package rather than low level data.