



Java Memory Model

This lesson lays out the ground work for understanding the Java Memory Model.

A memory model is defined as the set of rules according to which the compiler, the processor or the runtime is permitted to reorder memory operations. Reordering operations allow compilers and the like to apply optimizations that can result in better performance. However, this freedom can wreak havoc in a multithreaded program when the memory model is not well-understood with unexpected program outcomes.

Consider the below code snippet executed in our **main** thread. Assume the application also spawns a couple of other threads, that'll execute the method `runMethodForOtherThreads()`

```
1.  public class BadExample {
2.
3.      int myVariable = 0;
4.      boolean neverQuit = true;
5.
6.      public void runMethodForMainThread() {
7.
8.          // Change the variable value to lucky 7
9.          myVariable = 7;
10.     }
11.
12.     public void runMethodForOtherThreads() {
13.
14.         while (neverQuit) {
15.             System.out.println("myVariable : " + myVariable);
16.         }
17.     }
18. }
```

Now you would expect that the other threads would see the **myVariable** value change to 7 as soon as the **main** thread executes the assignment on **line 9**. This assumption is false in modern architectures and other threads may see the change in the value of the variable **myVariable** with a delay or not at all. Below are some of the reasons that can cause this to happen



- Use of sophisticated multi-level memory caches or processor caches
- Reordering of statements by the compiler which may differ from the source code ordering
- Other optimizations that the hardware, runtime or the compiler may apply.

One likely scenario can be that the variable is updated with the new value in the processor's cache but not in the main memory. When another thread running on another core requests the variable **myVariable**'s value from the memory, it still sees the stale value of **0**. This is a specific example of the **cache coherence** problem. Different processor architectures have different policies as to when an individual processor's cache is reconciled with the main memory

The above discussion then begets the question: *"Under what circumstances does a thread reading the **myVariable** value would see the updated value of 7"*. This can be answered by understanding the Java memory model (JMM).

Within-Thread as-if-serial



The Java language specification (JLS) mandates the JVM to maintain ***within-thread as-if-serial*** semantics. What this means is that, as long as the result of the program is exactly the same if it were to be executed in a strictly sequential environment (think single thread, single processor) then the JVM is free to undertake any optimizations it may deem necessary. Over the years, much of the performance improvements have come from these clever optimizations as clock rates for processors become harder to increase. However, when data is shared between threads, these very optimizations can result in concurrency errors and the developer needs to inform the JVM through synchronization constructs of when data is being shared.

Let's see in the next lesson how these optimizations can give surprising results in multithreaded scenarios.

[← Back](#)[Miscellaneous Topics](#)[Next →](#)[Reordering Effects](#)[Mark as Completed](#)

53% completed, meet the [criteria](#) and claim your course certificate!

[Claim Certificate](#)[Report
an Issue](#)[Ask a Question](#)

(https://discuss.educative.io/tag/java-memory-model__java-memory-model__java-multithreading-for-senior-engineering-interviews)





Reordering Effects

This lesson discusses the compiler, runtime or hardware optimizations that can cause reordering of program instructions

Take a look at the following program and try to come up with all the possible outcomes for the variables **ping** and **pong**.

```
1 class Demonstration {
2     public static void main( String args[] ) throws Exception {
3         (new ReorderingExample()).reorderTest();
4     }
5 }
6
7 class ReorderingExample {
8
9     private int ping = 0;
10    private int pong = 0;
11    private int foo = 0;
12    private int bar = 0;
13
14    public void reorderTest() throws InterruptedException {
15
16        Thread t1 = new Thread(new Runnable() {
17
18            public void run() {
19                foo = 1;
20                ping = bar;
21            }
22        });
23
24        Thread t2 = new Thread(new Runnable() {
25
26            public void run() {
27                bar = 1;
28                pong = foo;
```



Effect of reordering on program output



Most folks would come up with the following possible outcomes:

- 1 and 1
- 1 and 0
- 0 and 1

However, it might surprise many but the program can very well print **0 and 0**! How is that even possible? Think from the point of view of a compiler, it sees the following instructions for **thread t1's run()** method:

```
bar = 1;  
pong = foo;
```

The compiler doesn't know that the variable `bar` is being used by another thread so it may take the liberty to **reorder** the statements like so:

```
pong = foo;  
bar = 1;
```

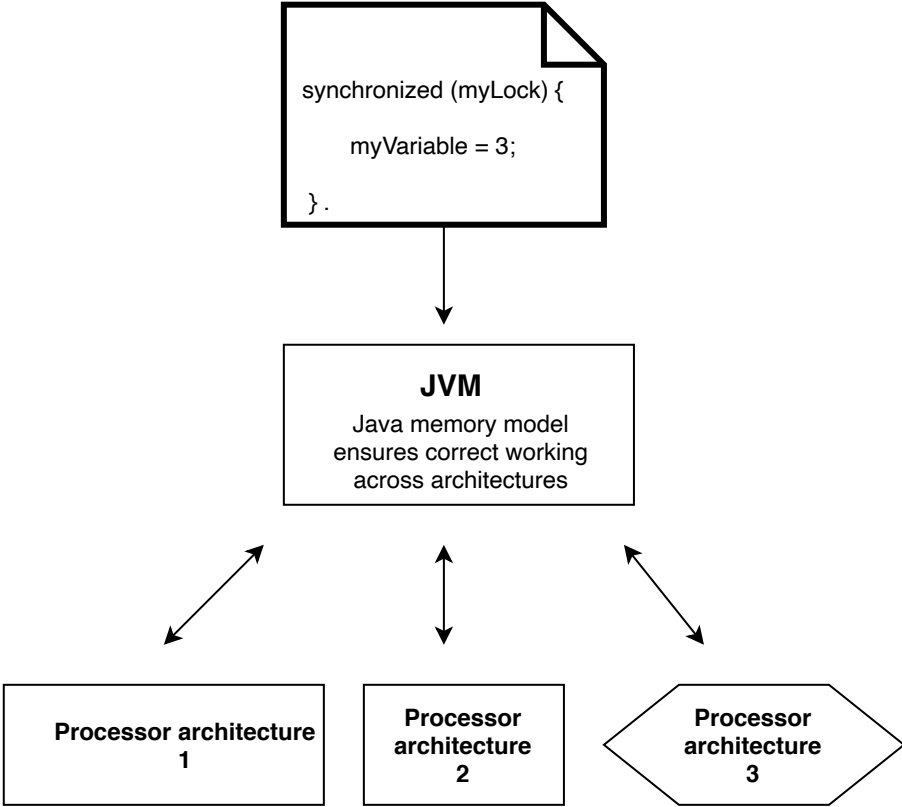
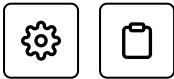
The two statements don't have a dependence on each other in the sense that they are working off of completely different variables. For performance reasons, the compiler may decide to switch their ordering. Other forces are also at play, for instance, the value of one of the variables may get flushed out to the main memory from the processor cache but not for the other variable.

Note that with the reordering of the statements the JVM still is able to honor the *within-thread as-if-serial* semantics and is completely justified to move the statements around. Such performance and optimization tricks by the compiler, runtime or hardware catch unsuspecting developers off-guard and lead to bugs which are very hard to reproduce in production.



Platform

Java is touts the famous ***code once, run anywhere*** mantra as one of its strengths. However, this isn't possible without Java shielding us from the vagrancies of the multitude of memory architectures that exist in the wild. For instance, the frequency of reconciling a processor's cache with the main memory depends on the processor architecture. A processor may relax its memory coherence guarantees in favor of better performance. The architecture's memory model specifies the guarantees a program can expect from the memory model. It will also specify instructions required to get additional memory coordination guarantees when data is being shared among threads. These instructions are usually called *memory fences or barriers* but the Java developer can rely on the JVM to interface with the underlying platform's memory model through its own memory model called JMM (Java Memory Model) and insert these platform memory specific instructions appropriately. Conversely, the JVM relies on the developer to identify when data is shared through the use of proper synchronization.



[← Back](#)

Java Memory Model

[Next →](#)

The happens-before Relationship

☒ **Mark as Completed**

53% completed, meet the criteria and claim your course certificate!

[Claim Certificate](#)

Report an Issue

Ask a Question
(https://discuss.educative.io/tag/reordering-effects__java-memory-model__java-multithreading-for-senior-engineering-interviews)



The happens-before Relationship

This lesson continues the in-depth discussion of Java memory model

The JMM defines a ***partial ordering on all actions within a program***. This might sound like a loaded statement so bear with me as I explain below.

Total Order

You are already familiar with total ordering, the sequence of natural numbers i.e. 1, 2, 3 4, is a total ordering. Each element is either greater or smaller than any other element in the set of natural numbers (**Totality**). If $2 < 4$ and $4 < 7$ then we know that $2 < 7$ necessarily (**Transitivity**). And finally if $3 < 5$ then 5 can't be less than 3 (**Asymmetry**).

Partial Order

Elements of a set will exhibit *partial ordering* when they possess transitivity and asymmetry but not totality. As an example think about your family tree. Your father is your ancestor, your grandfather is your father's ancestor. By transitivity, your grandfather is also your ancestor. However, your father or grandfather aren't ancestors of your mother and in a sense they are incomparable.

Java Memory Model Details



The compiler in the spirit of optimization is free to reorder statements however it must make sure that the outcome of the program is the same as without reordering. The sources of reordering can be numerous. Some examples include:

- If two fields **X** and **Y** are being assigned but don't depend on each other, then the compiler is free to reorder them
- Processors may execute instructions out of order under some circumstances
- Data may be juggled around in the registers, processor cache or the main memory in an order not specified by the program e.g. **Y** can be flushed to main memory before **X**.

Note that all these reorderings may happen behind the scenes in a single-threaded program but the program sees no ill-effects of these reorderings as the JMM guarantees that the outcome of the program would be the same as if these reorderings never happened.

However, when multiple threads are involved then these reorderings take on an altogether different meaning. Without proper synchronization, these same optimizations can wreak havoc and program output would be unpredictable.

The JMM is defined in terms of *actions* which can be any of the following

- read and writes of variables
- locks and unlocks of monitors
- starting and joining of threads

The JMM enforces a ***happens-before*** ordering on these actions. When an action A *happens-before* an action B, it implies that A is guaranteed to be ordered before B and visible to B. Consider the below program



```
public class ReorderingExample {
```



```
    int x = 3;
    int y = 7;
    int a = 4;
    int b = 9;
    Object lock1 = new Object();
    Object lock2 = new Object();
```

```
    public void writerThread() {
```

```
        // BLOCK#1
        // The statements in block#1 and block#2 aren't dependent
        // on each other and the two blocks can be reordered by the
        // compiler
        x = a;

        // BLOCK#2
        // These two writes within block#2 can't be reordered, as
        // they are dependent on each other. Though this block can
        // be ordered before block#1
        y += y;
        y *= y;

        // BLOCK#3
        // Because this block uses x and y, it can't be placed before
        // the assignments to the two variables, i.e. block#1 and block#2
        synchronized (lock1) {
            x *= x;
            y *= y;
        }

        // BLOCK#4
        // Since this block is also not dependent on block#3, it can be
        // placed before block#3 or block#2. But it can't be placed before
```

```

// block#1, as that would assign a different value to
o x
    synchronized (lock2) {
        a *= a;
        b *= b;
    }
}

```



Now note that even though all this reordering magic can happen in the background but the notion of *program order* is still maintained i.e. the final outcome is exactly the same as without the ordering. Furthermore, **block#1** will appear to *happen-before* **block#2** even if **block#2** gets executed before. Also note that **block#2** and **block#4** have no ordering dependency on each other.

One can see that there's no partial ordering between **block#1** and **block#2** but there's a partial ordering between **block#1** and **block#3** where **block#3** must come after **block#1**.

The reordering tricks are harmless in case of a single threaded program but all hell will break loose when we introduce another thread that shares the data that is being read or written to in the `writerThread` method. Consider the addition of the following method to the previous class.

```
public void readerThread() {
```

```
    a *= 10;
```

```
    // BLOCK#4
```

```
    // Moved out to here from writerThread method
```

```
    synchronized (lock2) {
```

```
        a *= a;
```

```
        b *= b;
```

```
    }
```

```
}
```



Note we moved out **block#4** into the new method `readerThread`. Say if the `readerThread` runs to completion, it is possible for the `writerThread` to never see the updated value of the variable `a` as it may never have been flushed out to the main memory, where the `writerThread` would attempt to read from. There's no *happens before* relationship between the two code snippets executed in two different threads!

To make sure that the changes done by one thread to shared data are visible immediately to the next thread accessing those same variables, we must establish a *happens-before* relationship between the execution of the two threads. A happens before relationship can be established in the following ways.

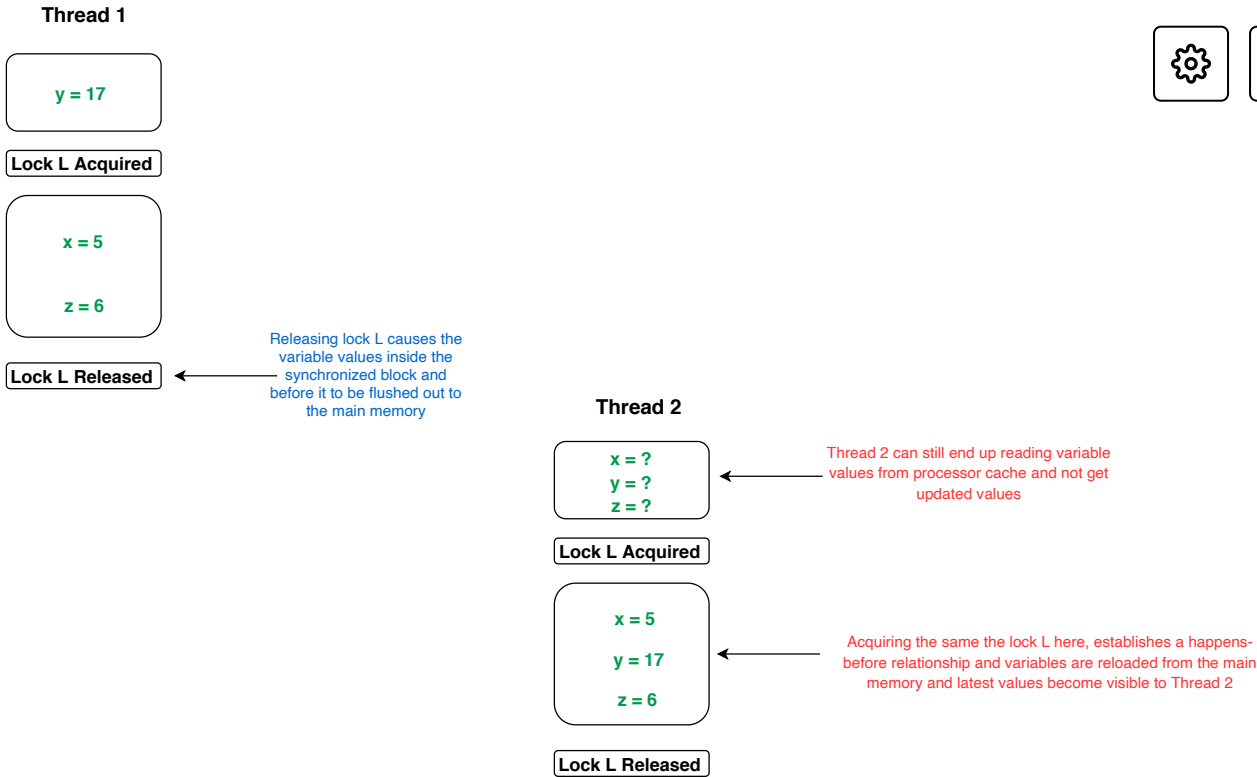
- Each action in a thread *happens-before* every action in that thread that comes later in the program's order. However, for a single-threaded program, instructions can be reordered but the semantics of the program order is still preserved.
- An unlock on a monitor *happens-before* every subsequent lock on that same monitor. The `synchronization` block is equivalent of a monitor.
- A write to a volatile field *happens-before* every subsequent read of that same volatile.

- A call to `start()` on a thread *happens-before* any actions in the started thread.
- All actions in a thread *happen-before* any other thread successfully returns from a `join()` on that thread.
- The constructor for an object *happens-before* the start of the finalizer for that object
- A thread interrupting another thread *happens-before* the interrupted thread detects it has been interrupted.



This implies that any memory operations which were visible to a thread before exiting a synchronized block are visible to any thread after it enters a synchronized block protected by the same monitor, since all the memory operations happen before the release, and the release happens before the acquire. Exiting a synchronized block causes the cache to be flushed to the main memory so that the writes made by the exiting thread are visible to other threads. Similarly, entering a synchronized block has the effect of invalidating the local processor cache and reloading of variables from the main memory so that the entering thread is able to see the latest values.

In our `readerThread` if we synchronize on the same lock object as the one we synchronize on in the `writerThread` then we would establish a *happens-before* relationship between the two threads. Don't confuse it to mean that one thread executes before the other. All it means is that when `readerThread` releases the monitor, up till that point, whatever shared variables it has manipulated will have their latest values visible to the `writerThread` as soon as it acquires the **same** monitor. If it acquires a different monitor then there's no happens-before relationship and it may or may not see the latest values for the shared variables



Happens-Before Relationship

←


Back

Next

→

Reordering Effects


Blocking Queue | Bounded Buffer | Co...




Mark as Completed

53% completed, meet the criteria and claim your course certificate!

Claim Certificate

- 

Report an Issue
- 

Ask a Question

(https://discuss.educative.io/tag/the-happens-before-relationship__java-memory-model__java-multithreading-for-senior-engineering-interviews)