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.