Reordering Effects

Introduction

As discussed earlier the JMM allows for reordering of statements in the program order as long as the outcome of the program isn't altered. Consider the program below and try to come up with all the possible outcomes the program can have:

Java

main.java

```
class ReorderingExample {

    // shared variables
    int sharedA = 0;
    int sharedB = 0;

    // executed by thread1
    void method1() {
        int localA;
        localA = sharedA;
        sharedB = 1;
        System.out.println("localA = " + localA);
```

Run

Example 1

The above program is adapted from an example in JSR-133. The program has two variables that are shared between two threads `sharedA` and `sharedB`. Each thread updates one of the shared variables that is used in the other thread. To keep the example simple, we can boil down the execution of the two threads as follows:

Instruction#	Thread1	Thread2
1.	localA = sharedA;	localB = sharedB;
2.	sharedB = 1;	sharedA = 2;

Most folks would come up with the following possible outcomes:

- `localA=0` and `localB=1`
- `localA=2` and `localB=0`
- `localA=0` and `localB=0`

However, it might surprise many but the program can very well print **`localA=2` and `localB=1`**! How is that even possible? Think from the point of view of a compiler, it sees the following instructions for `method1()`:

```
    localA = sharedA;
    sharedB = 1;
```

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

```
    sharedB = 1;
    localA = sharedA;
```

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. Compilers are allowed to reorder instructions in a given thread, when this does not affect the execution of that thread in isolation. 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 the same may not happen for the other variable.

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

Informally, the JSR-133 calls the write of a variable in one thread, read of the same variable in another and the two events (the read and the write) not being ordered by synchronization as a *data race*. An important concept we'll revisit in the next lesson.

Example 2

Here's another example of compiler optimization that can change the order of program statements along with the program outcome but is valid according to Java's memory model.

```
1  class ReorderingExample {
2
3      static class MyObject {
4          int x;
5      }
6
7      // shared variables
8      MyObject obj1;
9      MyObject obj2;
10
11
12     // executed by thread1
13     void method1() {
14
15         MyObject read1 = obj1;
16         int read2 = read1.x;
17         MyObject read3 = obj2;
18         int read4 = read3.x;
19         int read5 = read1.x;
20     }
21
22     // executed by thread2
23     void method2() {
24         MyObject read6 = obj1;
25         read6.x = 3;
```

The above program lacks synchronization between the two threads that execute `method1()` and `method2()`. The compiler can see that within `method1()` the value `read1.x` is used twice on lines 16 and 19. Since `obj1` is not written to between these lines, the compiler can choose to re-use the `read2` value in line 19 instead of reading `read1.x` again. The method would like as follows after the optimization change:

```
    void method1() {

        MyObject read1 = obj1;
        int read2 = read1.x;
        MyObject read3 = obj2;
        int read4 = read3.x;
        // compiler reuses read2
        int read5 = read2;
    }
```

However, with thread2 executing `method2()` we can have a sequence of executions that can seem to result in inconsistent reads of the same field. Consider:

1. `int read2 = read1.x;` is executed by thread1 and `read2` holds the value `0`.
2. `MyObject read3 = obj2;` is executed by thread1.
3. `MyObject read6 = obj1;` is executed by thread2.
4. `read6.x = 3;` is executed by thread2. At this point `obj1.x`'s value is changed to `3`
5. `int read4 = read3.x;` is executed by thread1 and `read4` is set to `3`.
6. `int read5 = read2;` is executed by thread1 and `read5` is set to `0`.

Thus the above program execution shows that the read of the same field `obj1.x`, from the developer's perspective changes from 0 to 3 and then back to 0.

Example 3

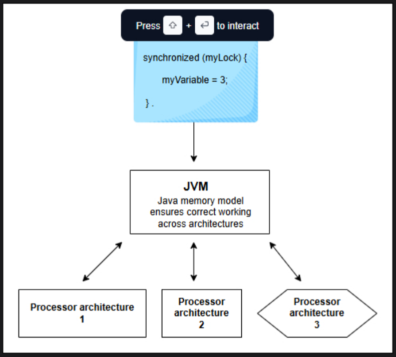
So far we have seen examples where compiler optimizations caused unexpected results. However, we can also see reorderings by processor architectures that can yield surprising results. Consider the below program:

```
public class ReorderingExample {

    int x = 0;

    // executed by thread1
    void method1() {
        int read1 = x;
        x = 1;
    }

    // executed by thread2
    void method2(){
        int read2 = x;
        x = 2;
    }
}
```

Note that the statements within the two methods can't be reordered by the compiler since that would change the semantics of the program. For instance in `method1()` we can't move the `x = 1` statement before the `int read1 = x` statement as that would give a value of `1` to `read1` when that may not be necessarily true from the execution of the original program order. Given this we can conclude that at least one of the two variables i.e. `read1` or `read2` should equal zero as the statements are never reordered by the compiler. However, it is possible to see `read1=2` and `read2=1`! The reason is that certain processor architectures can commit writes early in such a way that they aren't visible to the local reads that come before the writes. So in our example in `method1()` the write `x = 1` may get committed and be visible to thread2 but not to thread1. This would be valid according to JMM because thread1 still executes as if nothing changed for it.

Affect of memory architectures

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 lesson
Memory Model

Completed
Next →
The happens-before Relationship and Model