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`.

```
class Demonstration {
    public static void main( S
        (new ReorderingExample()
    }
}

class ReorderingExample {

    private int ping = 0;
    private int pong = 0;
    private int foo = 0;
    private int bar = 0;

    public void reorderTest()

        Thread t1 = new Thread

            public void run()
                foo = 1;
                ping = bar;
            }
        });

        Thread t2 = new Thread

            public void run()
                bar = 1;
                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.