Question 1
If all memory locations are marked as volatile in a Java program, what is the result?

1. SC
2. TSO consistency
3. normal Java memory model semantics
4. deadlock

Question 2
For reads and writes of volatile locations in Java, running on a TSO processor, when are fences necessary?

1. after reads
2. after writes
3. after both
4. after neither

Question 3
What guarantee does the Java Memory Model provide for correctly synchronized programs?

1. termination
2. TSO consistency
3. SC
4. no guarantees

Question 4

In Sevcik and Aspinall's example of Redundant Write after Read Elimination, why is it unlikely that the two reads (r1=x and r2=x) would return different values in practice?

1. JVMs don't perform redundant write-after-read elimination
2. JVMs are likely to eliminate the now-redundant read of r2=x
3. The example program will always execute in an SC manner because it is data-race-free.
4. Returning different values is the result of out-of-thin-air values, which the JMM prohibits.

Question 5

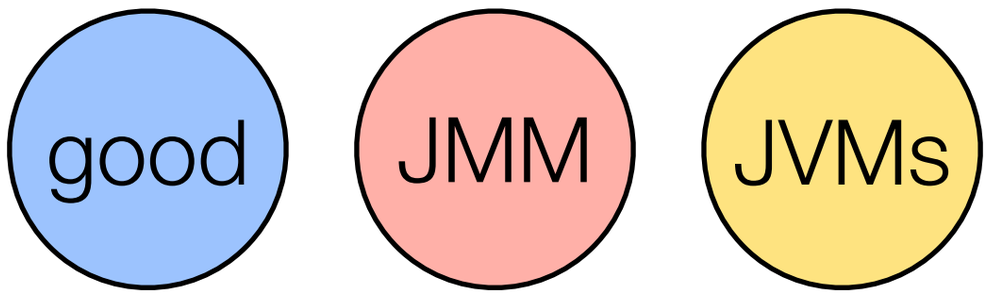
In Sevcik and Aspinall's discussion of Irrelevant Read Introduction (page 6), they mention that hardware can introduce irrelevant reads through speculation, and thus violate the JMM. Is Java code running on modern architectures broken?

1. No, because hardware is non-speculative.
2. No, because hardware always commits speculative operations in program order.
3. No, because reads do not modify any architected state.
4. Yes, because hardware does not track control dependences for speculative operations.

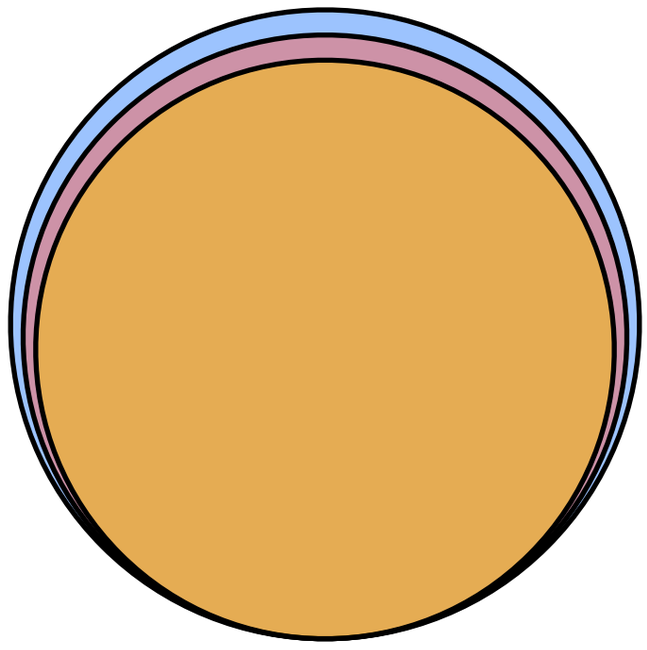
TABLE 4.4: TSO Ordering Rules. An "X" Denotes an Enforced Ordering. A "B" Denotes that Bypassing is Required if the Operations are to the Same Address. Entries that are Different from the SC Ordering Rules are Shaded and Shown in Bold.

		Operation 2			
		Load	Store	RMW	FENCE
Operation 1	Load	X	X	X	**X**
	Store	**B**	X	X	**X**
	RMW	X	X	X	**X**
	FENCE	**X**	**X**	**X**	**X**

design goal of the JMM



goal



Flag-based synchronization

```
boolean ready = false;
```

```
// wait for condition
while (!ready) {}

// go for it...
```

```
// initialize...

ready = true;
```

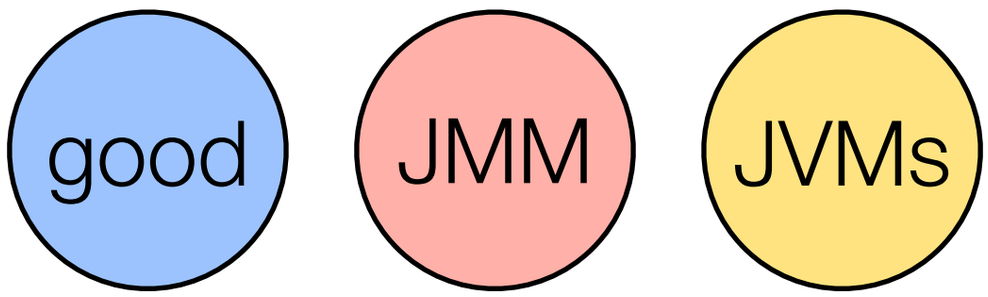
double-checked locking

```
class Foo {
    private Singleton s = null;
    public Singleton getS() {
        if (s == null) {
            s = new Singleton();
        }
        return s;
    }
}
```

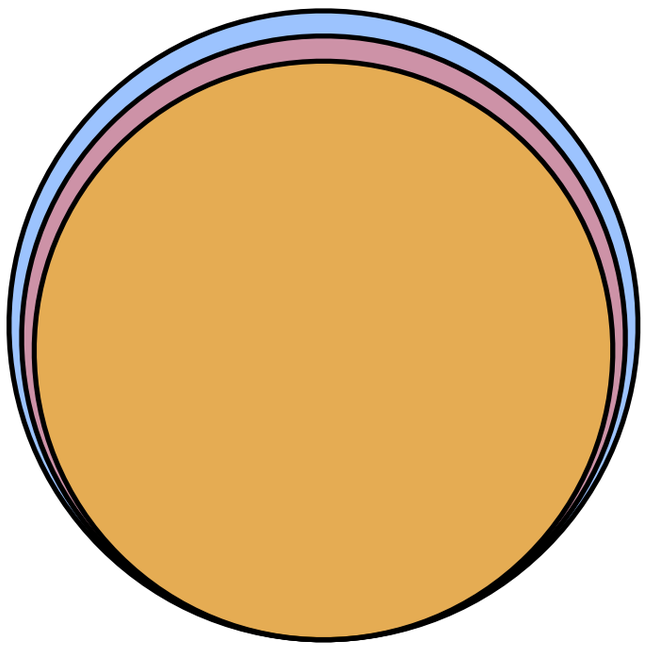
Dekker's algorithm

```
flag[0] = false
flag[1] = false
turn    = 0


flag[0] = true;
while (flag[1] == true) {
    if (turn ≠ 0) {
        flag[0] = false;

        while (turn ≠ 0) {}

        flag[0] = true;
    }
}

// critical section

turn    = 1;
flag[0] = false;
```

Is the JMM fatally flawed?



ideal



after S+A