**Exercise 2.1**

2. Our solution is fair towards writer threads. We implemented it using two counters:
   - *readsAcquires*: it counts how many reading threads have entered their critical section (so have started to read )
   - *readsReleases*: it counts how many reading threads have finished to read:

   When readsAcquires is equal to readsReleases it means that no one is currently reading the shared memory, so when this happens, a writer can proceed. This solution is more fair towards writers since there is more interleaving between writers and readers.

3. It isn't possible to ensure the absence of starvation using ReentrantLock or intrinsic Java locks (synchronized) only. Both prevent multiple threads from concurrently executing a specific critical section of code. However, they do not guarantee fairness in thread scheduling, and therefore, they cannot ensure the absence of starvation, for example if in the reader-writer problem there is an infinite number of readers it might happen that the writer never enter his critical section, so the writer "starves".

**Exercise 2.2**

1. There can be a visibility problem: the main thread only updates the cache, therefore can happen the t thread can only read the original value because the main memory is not up to date with the new value written in the cache, so it's possible that it runs forever.

2. The use of synchronized ensures mutual exclusion so there is no longer a visibility problem in fact when the thread writes, it also updates the shared memory, so the t thread is able to read the new value and does not loop forever.

3. Get method must be synchronized for the program to always terminate. If you synchronize the get method, you ensure that whenever a thread accesses the value of the shared variable, it gets the most recent value from main memory. Instead, if you synchronize the set method, you are ensuring that when one thread writes a new value to the shared variable, that value is actually written to main memory but the other can continue reading from the cache if the get method is not synchronized, so the visibility problem persists.

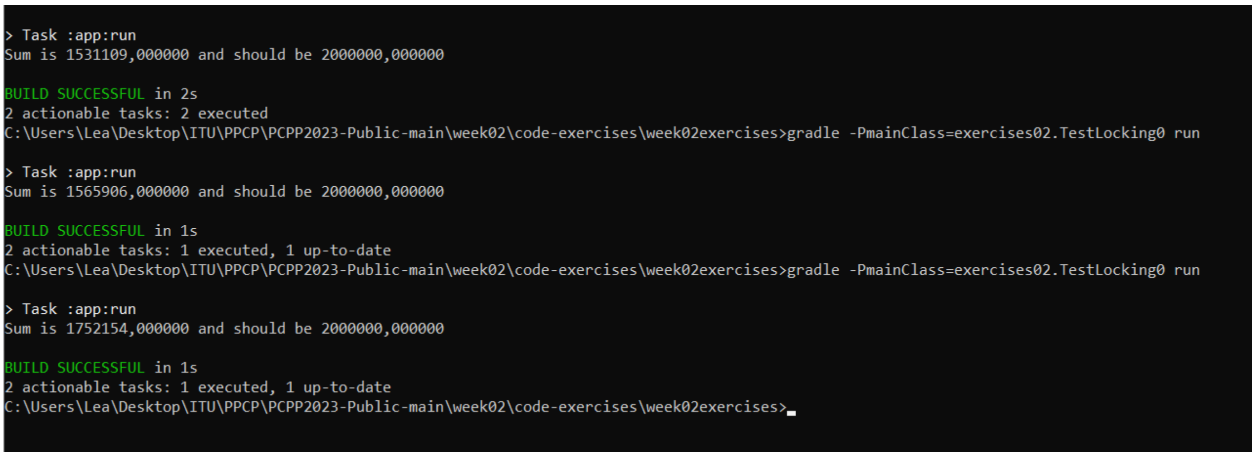
4. If we use the volatile variable the program will terminate. In fact, volatile variables ensure visibility between threads however they do not ensure mutual exclusion so race conditions can happen

5. In point 3 if we do not synchronize the get() there is no guarantee of happen-before relationship between main thread setting mi to the value 42 and the thread t that repeatedly reads the variable mi. this can lead to infinite looping since t thread may never see the update done by the main thread.
   In point 4 defining value as volatile we establish an happens before relationship between reads and writes of this variable; so writes to value by the "main" thread are

always visible to the t thread when it reads the value, preventing it from looping forever due to the visibility issue. However there is no mutual exclusion; it can happen that the t thread exit the loop prematurely, leading to incorrect value being printed.

Exercise 2.3

1. A race condition occurs when threads modify the variable m without being sure that the other has already made the sum. In fact every time we execute the code we get different results, as shown below there are race conditions:



```
> Task :app:run
Sum is 1531109,000000 and should be 2000000,000000

BUILD SUCCESSFUL in 2s
2 actionable tasks: 2 executed
C:\Users\Lea\Desktop\ITU\PPCP\PCPP2023-Public-main\week02\code-exercises\week02exercises>gradle -PmainClass=exercises02.TestLocking0 run

> Task :app:run
Sum is 1565906,000000 and should be 2000000,000000

BUILD SUCCESSFUL in 1s
2 actionable tasks: 1 executed, 1 up-to-date
C:\Users\Lea\Desktop\ITU\PPCP\PCPP2023-Public-main\week02\code-exercises\week02exercises>gradle -PmainClass=exercises02.TestLocking0 run

> Task :app:run
Sum is 1752154,000000 and should be 2000000,000000

BUILD SUCCESSFUL in 1s
2 actionable tasks: 1 executed, 1 up-to-date
C:\Users\Lea\Desktop\ITU\PPCP\PCPP2023-Public-main\week02\code-exercises\week02exercises>
```

2. Race conditions appear because t1 and t2 access the variable sum in the object Mystery concurrently without the right synchronization.
   In fact t1 uses the non static method so it works on the lock of the instance m, while t2 modifies the variable using the static method so it references to the class and not the instance m. Since t1 and t2 are using different instances of the Mystery class (m and the class itself), they do not block each other when accessing addInstance, that is why we have a race condition.

3. We used the lock and unlock functions from the ReentrantLock. The race condition happens when the threads interleave the execution of the addInstance() and the addStatic() functions on the m variable. Locking before and unlocking after ensures the correct behavior of the code.

4. There is no race condition in the code, therefore the use of synchronized is not necessary.