Mandatory week1

1) I did not get the expected output. I got 19 649 454 instead of 20 million.
2) This time I expected to get 200 and I indeed got 200. The reason is the code we are using in this exercise has a risk component to it. The risk is that one of the threads gets in the way of the other thread. One case where that could happen is if one thread reads the value of the shared field at the same time as the other thread. They then both increase the value and one of the increases is just lost. The risk of an operation such as this happening is proportional to how many operations happen. Going from 10 million to 200 strongly reduces the probability of this situation occurring. Having it run 200 times only does **not** guarantee that the value will be 200. One situation like the one I described before can still happen.
3) Changing the code to code++ or code = code+1 will not make a difference. These are all different **syntaxes** but not actually different operations. They all read and then add something to the state. It could make a difference to use atomic integers, that treat reading and adding as an atomic, indivisible operation. This way the operations would never be first read then write and interleaving could never happen between read and write.
4) I have chosen to use the lock method around the operation `count = count + 1;` inside the method where this operation is defined. This way, any call done in any number of threads will always lock the access to the shared state variable, and release the lock when it is done altering it. This makes the process of accessing and changing the variable “exclusive property” of one thread at a time, necessarily. Because of that, a situation where one of the threads reads and older value or any situations that invoke race conditions, as described earlier on, will never happen.
5) This is also the least lines of code as it is written straight to the method, any new thread wont add more lines of code. If I was to write this around every call of the method in each of the threads, the critical part of the code would be bigger, and with more threads, I would have to write it even more.

1.2)

1.2.2) What happens is that there are two actions being performed in the method: a print of “-” and then after 50 milliseconds a print of “|”. Lets call these actions 1 and 2. So a thread T would perform this actions as follows, in a non concurrency environment:
T1,T2,T1,T2,T1,T2.

Now, without proper synchronization, which was the case before we intervened, if we have two threads, T and G, it could happen that while the code is waiting to print the “|”, the other thread could print a “-”. A possible interleaving where that happens is:
T1,G1,T2,G2. That is, thread T and thread G both start action 1 almost at the same time, and before any of them get to action 2. This leads to the mistakes in the printing and can be prevented if the bundle action1, action2 is executed by only one thread at a time. This can be achieved in a number of ways.

1.2.3) The new execution of this program , using the proper synchronization mechanism, will never commit the same mistake as before. The reason is the whole operation of printing

both symbols(- and |) is wrapped around a lock. This means that there will never be interleaving between printing - and the other thread printing -.

1.3)
I have wrapped the counter adding operation, which in reality involves reading, adding and writing on the counter shared variable, in locks. This way, everytime a turnstile turns its as if it blocks the other one from turning at the same time. They can only alter the shared counter one at a time and racing conditions do not occur. The operations will happen one at a time and eventually hit the imposed limit of 15000.

Assignment 2

2.1) I created a class called MyClass in java that contains a reader and a writer method. After so, I created a main class where I created 3 threads, 2 are read only and one is a writer. A thread is a reader when it uses the method to read the shared value and a writer when it uses the method to overwrite it. I **initially** included a synchronized block on the reader and on the writer method. This way a thread reads in between a writing, and there is no race condition. The problem, however, **was** that two threads will have to wait on one another to read something even when there is no new values. This can **not only** cause the performance of the threads to be slower, but it does not fulfill the requirement where all the reads can be done at the **same time.**

So I changed the solution to use a `ReentrantReadWriteLock()`**.**

This allowed me to do two different kinds of locking. The read locking allows multiple readers at the same time, but when a writer lock wants the lock, it prevents the other readers to read the shared data. Its as if the readers didnt have a lock among themselves but they had one with the writer. Again, just using a normal lock would also lock the reader from each other, which is not what we want.

This is a solution I found to be fair with the writers, but its a built in solution. What I had previously tried would allow writers to write but the readers would have to wait on one another.

2.2)
2.2.1) Yes I observed that t is started on line 14, so the main thread is running and t is running concurrently. The thread uses the initial value of mi, zero as that is the one when it starts. The main thread continues the execution of the program and prints out that mi is set to 42. I even called the get method inside the printing to asure that indeed the main thread sets the mi value to 42 but the t thread cannot see that. This uncoordination between the two leads to the reading of a stale value by the thread.

2.2.2) In between the loops of the thread, at some point, the main thread will ask for the key so that it can alter the value to 42. Once 42 is put there the key is released and then the thread will run one more time the mi.get() but this time it will get a different value (42) and

now the loop will be exited, the thread will run “I completed, mi =42” and the last statement, “Thread t completed…” will be printed.

So, when taking off synchrony from the get, the code could fully run but there is no guarantee, and hence its not a thread safe approach. I originally expected the program to terminate even without synchronized, but I still didnt know about volatile values. Now I see that if writing a new value is done in the main thread, this needs to be a volatile value so that another thread can see that that value has changed. When its not a volatile value and they are not synchornized, the thread t may never realise that main changed that value and hence **never terminate.**

To conclude, with the last question, being a volatile value would cause it to necessarily terminate. At some point the main thread changes the value and with volatile, we are sure that the other thread sees that change, and by seeing that, it stops its loop.

2.2.4) This particular exercise is interesting because its not particularly problematic that the loop is run one more time. The usage of volatile seems like a different approach that is also more fitting to this scenario. Volatile seems a little less safe but more efficient. The code will see the changes but not in such a secure way like the lock. There is no need to get a lock and a key because there wont be conflicts, its not a writting situation, its more of a checking situation, so its less problematic that the value took longer to be updated then if we were passing this value to something and it was a stale value.

2.3) Yes, there are race conditions. When running it multiple times, I didnt get the correct value once, I got values like
Sum is 1671646,000000 and should be 2000000,000000 and
Sum is 1093946,000000 and should be 2000000,000000
but didnt get the expected value.

2.3.2) The problem is that when I have one static method and one instance method, the locks of the synchronized are different. Hence despite looking like a thread safe class, because its methods respect each other when accessing shared data, it actually is not a thread safe class: the locks are not guaranteeing serial access to the shared data. One thread must not give up on its lock so another thread can use it, because the locks are different. This way, race conditions are created.

2.3.3) Changing both methods to static would suffice as they would then share the same lock, but as the exercise requested us to do something other than that, I just create an actual common lock. In the main thread I created an object of class Object, and in the body of the threads, I made them share such common lock. Now as they are properly sharing the **same** lock, there is serial access to the shared data and no race conditions.

2.3.4) The last method is only called after the threads have been merged to main. The sum method is called after both methods are asked to join main. Therefore, at that point there is no longer a concurrency scenario. If this method was called, due to some reason, within a concurrent scenario, then it would need to have a proper synchronization mechanism with the other methods. But the very idea of the method seems to be a final call, so in the situation we have it now, it should be fine to not have it with a lock.