The Singleton Design Pattern

If you didn't provide implementations of a lazy and eager singleton pattern in Question 6 do so now. (You should provide a static getInstance method. Imagine that we now wish to use the code in a multi-threaded environment. Two threads concurrently access the class, thread t1 gives the first call to the getInstance() method, it will check if the static variable that holds the reference to the singleton instance is null and then gets interrupted due to some reason. Another thread t2 calls the getInstance() method successfully passes the instance check and instantiates the object. Then, thread t1 wakes and it also creates the object. At this time, there would be two objects of this class which was supposedly a singleton.

(a) How could we use the synchronized keyword to the getInstance() method to operate correctly.
I have made the method getInstance synchronized to prevent thread interference and duplicated instances of the object. See '*SingletonMultiThreaded.java*'

(b) The synchronised version comes with a price as it will decrease the performance of the code — why?
Once a thread has accessed the getInstance() method it cannot be interrupted by other threads, this will delay the other threads from continuing until the first thread has returned from the method.

(c) If the call to the getInstance() method isn't causing a substantial overhead for your application, then you can forget about it.
Didn't understand this question – will ask in lab.

(d) If you want to use synchronisation (or need to), then there is another technique known as double-checked locking which reduces the use of synchronisation. With double-checked locking, we first check to see if an instance is created, and if not, then we synchronise. Provide a sample implementation of this technique.
See *'SingletonDoubleChecked.java'*

There are some other ways to break the singleton pattern:

- If the class is Serializable.
- If it is Cloneable.
- It can be broken by reflection.
- If the class is loaded by multiple class loaders.

Try and write a class SingletonProtected that addresses some (all?) of these issues.
See *'SingletonProtected.java'*