**Part B**

Please suggest (but do not implement) modifications or additions to the Order and/or OrderBook classes to make them better suited to support real-life, latency-sensitive trading operations.

1. Since the **Order** class was already provided I haven’t touch it, but we shouldn’t use a ***long*** type as an ***ID***, instead we should use a ***String*** type and store a UUID in it. Once the trading solution process/interacts with a huge numer of orders (more than Long.MAX\_VALUE ) everything will fail.
2. In addition the **Order** class, although is immutable it should still have an equals and hashcode.  
   **Reasoning**: Imagine we consume a message of creating a new Order but we don’t send the acknowledge until everything is done, so next time we are gonna grab it again and we will end up in the system will 2 orders that have the same values & ids. In the same time is possible that the same message is placed twice in the Queue by another system so we will have to protect against it.
3. Since in the part A of the problem there wasn’t any suggestion to implement any ***fulfilment*** ***action*** (consume a matching bid & offer by creating an execution/trade) , I decided to go for using a REST Api when getting the price/totalSize or all the orders using the side, but probably the best approach will be to change this and move everything via a queing system of kafka if needed.
4. Now since the trading system happens inside the same microservice, this will definetly not support too many users, so we will have to scale it and spin more instances.  
   The catch is that with the current implementation it will never work since the state is kept inside each microservice. So we have 2 options to manage the state in **OrderBook** class:

* Use an Distributed In Memory DB like **Hazelcast** to keep/acces the collections by all microservices (we can protect them on write using distributed locks from Hazelcast). In addition the configuration should allow the peristence in the file system, in case the Hazelcast goes down we will be sure that we won’t loose data.
* Another option will be to use like a **MongoDB** and store all orders in the same collection but with a different partitioning key, like based on the side. Since mongo is itself scalable and supports transactions at the level of collections this should be a good fit

1. In the current implementation I’m using JMS, but a better approach will be to use like RabbitMQ /ActiveMQ and configure it to have persistence on the file systems in case of failures, or even better use something like KAFKA, although that is more costly and complicated to set up.
2. Different mecahisms will have to be implemented in order to ensure like the same message (order creation/update) is not processed twice. E.g. we could sent the ackownledge back to the queing system only if we first store the data in the DB with the state ***INIT*** and a unique id/key and later on it will be changed to ***PROCESSED*** by another thread that is doing the processing. In case we get the same message again(system could have crashed before sending ack) we first check if the message is in DB and if yes, we only send ack.  
   In case of an error during the processing we can sent back another message via the queing system. Similar if the save in DB fails we can put the message in the DeadLetterQueue.