Specifiaction: Every set of commands ,besides State and Transfer toghether, will run concurently. By set of commands I mean any group of any number of same or different commands, for example: State,State,Open,Convert all running at the same time). Assumption 3: Commands State and and Transfer can't run concurently because there is the risk that State will be executed in between a transfer operations. The 3 classes are documented additionally in the code.

Methods to prevent Race condition:

1. As a data structure I used a Concurrent Hash Map which is thread-safe and whose methods help avoiding race conditions. In the class AccountManager lines 58,59 I avoided the posibility that if 2 threads try to open an account with the exact same number at the exact same time, the second one will not override the first one, instead will get a message error that the account already exists. I have also added a test at line 61 to check if everything worked as intended. Methods used at line 58,59: containsKey(),putIfAbsent().

2. Assumption 1: To protect against a race condition situation I used 2 semaphores, 2 locks and 2 conditions on locks and between State and Transfer commands. SemaC at lines 75 to 86 is a hold lock for State while SemaB at lines 134 to 145 is a hold lock for Transfer. They are used to prevent the case that the State command will get executed when one or more transfers are in progress .This state shown would be innacurate as currency data would be missing.

3. At lines 120 and 154,155 in the AccountManager class, the changes withing the Convert and Transfer commands are called. They will get executed in the Account class at lines 23,24 and 27,28. In the Account class all methods are **synchronized (lines 22,26)** in order to lock each individual object that is getting its currecy modified, instead of the whole collection of object (whole system). This not only allows multiple concurrent account modifications but also protects against a race condition where if 2 threads could modify the same account at the same time this would produce innacurate data (due to the nature of the operations).

Methods to prevent Deadlock:

1. To protect against a possible deadlock between the lock on the semaphoreB and semaphoreC , I only instantiated the waiting state of the commands State and Transfer at lines 80,82 and 139,141 with a condition and a method awaitUninterruptibly(). They will automatically exit get the lock and semaphore underneath when is their turn to execute, when the get signalled by the other command method signalAll().

2. Important: At lines 79,85,138,142 locks are acquired and released to solve an assumption monitor/deadlock error, not actuallly to acquire a lock.

3. To protect against a possible deadlock when 2 transfers occur between Account1 - Account2 and Account2 - Account3 which may deadlock on Account2, I used indiviual account locks which get released immediatly after the currency change was produced lines 154,155.

4. Assumption 2: I also used the 2 locks and a Thread.sleep(500) to protect against a deadlock situation in the case of infinitely calls of a Transfer or State command at the exact same time, which would not release the lock between State and Transfer if there was only one (lock). However, in our case the maximum number of threads is known so this shouldn't be the case (Unless there is a way to simulate that).