Exercise 6.1

Question 1

`TestAccountDeadlock` run into a deadlock as `clerk1` and `clerk2` are running simultaneously locking account 1 and 2 respectively while waiting for the lock of the other account to be released. Thus, none of them are able to acquire the other lock making them both wait indefinitely.

Question 2

No, the code did not enter a deadlock. It is possible for the code to enter a deadlock as two objects could have the same identity hashcode even though they are not the same object. If we call the method with two identity hashcode equal objects, they will enter the same branch in the if sentence, but due to ordering, they might not lock on the same object first, introducing a deadlock.

If waiting long enough, the code could enter a deadlock. We tried making a loop that ran until two identical hashcodes appeared, but we did not actually manage to hit a hash collision.

Question 3

We used the Goetz idea by adding a special case for identity hashcode collisions. In case we get a collision, we acquire a special shared tie lock, which must be acquired before any locking on the account objects happen. In the case that we try to transfer in both directions between two identity hashcode equal objects, we will only be able to acquire the account locks, if we have the shared tie lock, thereby preventing a deadlock.

If we should happen to call the same method with two other identity hashcode equal objects, they must wait for the same shared tie lock used by the others. This is not really a problem, as the identity hashcode collisions should happen rarely.

The code works and it still does not enter a deadlock.

Question 4

It would be safe and deadlock-free to ignore the hashcodes and using a tie lock instead, but it would not be scalable as all transfers must acquire the tie lock even if they do not transfer between the same accounts.

This would become a concurrency bottleneck.

Exercise 6.2

Question 1

The program might deadlock if each philosopher grabs the fork to their left simultaneously and then try to grab the fork to their right. All the forks will be taken leaving all philosophers waiting for a fork to free up, thereby introducing a deadlock.

Question 2

The program enters a deadlock after running various lengths of time.

The program enters a deadlock only if all the philosophers have one fork at the same time.

The fewer philosophers you have it is higher risk of each philosopher having one fork each. It seems like the more cores you have, the easier it is to enter a deadlock.

Question 3

The following output was generated by a thread dump:

```
Found one Java-level deadlock:
=============================
"Thread-3":
  waiting to lock monitor 0x00007fc8ac0b9758 (object 0x00000007957bc330, a Fork),
  which is held by "Thread-0"
"Thread-0":
  waiting to lock monitor 0x00007fc8ac06b958 (object 0x00000007957bc340, a Fork),
  which is held by "Thread-1"
"Thread-1":
  waiting to lock monitor 0x00007fc8ac06b8a8 (object 0x00000007957bc350, a Fork),
  which is held by "Thread-2"
"Thread-2":
  waiting to lock monitor 0x00007fc8ac0b9808 (object 0x00000007957bc360, a Fork),
  which is held by "Thread-3"
```

This shows that each thread holds a lock to a fork and are waiting for another the right hand fork lock to be released.

Question 4

This version of the program does not enter a deadlock as each philosopher will always take the fork with the lowest index first. Thus, the last philosopher will first try to acquire the lock for fork 0 and then it will try to acquire the lock for its own fork. In this way all the philiosophers will never wait for a fork lock to be released and it will never enter a deadlock.

Question 5

Yes, it does not enter a deadlock at any time. This will never happen as one philisopher will never wait for another lock.

A possible problem with the program is that livelock can occur. This occurs when all philosophers

repeatedly tries to take the locks without succeeding, thereby releasing their locks starting over the attempt of eating. This is not very likely and if it does occur, the risk of multiple failed attempts in a row is low as we use sleep with a random time.

Question 6

There is nothing to ensure that all of them each eats the same number of times. A philosopher could grab the same fork two times in a row.

Exercise 6.3

Question 1

Question 2

Question 3

Exercise 6.4

Question 1

Yes, it passes initially when everything is synchronized.

After removing synchronized on `increment`, ThreadSafe reports "unsynchronized read/write" on the method.

The `getSpan` method is not reported by ThreadSafe as it is accessing a final value.

Question 2

The `addAll` method needs to lock while adding the buckets to the histogram as it needs the increment (+=) operations to be atomic.

ThreadSafe agrees as it does not report anything. When removing the lock on this it reports that we have unsynchronized read and writes during the increment as expected.

Question 3

In the `madAll` we only synchronize on `this`. ThreadSafe reports that the access to `that.counts` is only sometimes synchronized, as we do not hold the lock on `that` while reading. Since we do not write back to `that`, it should not be a problem. The only drawback here, will be that the reading of the whole histogram will not be atomic.