Exercise 12

12.1.1

```
 private static void testBankConcurrent(PicoBank bank, String className) throws InterruptedException,
BrokenBarrierException {
        BarrierTimer bt = new BarrierTimer();
        final AtomicLong atomicSum = new AtomicLong();
        final int numberOfThreads = 8;
        final int numberOfOperations = 1_000_000;
        final CyclicBarrier barrier = new CyclicBarrier(numberOfThreads + 1,bt);

        for (int i=0; i< numberOfThreads; i++)
        {
            new Thread(() -> {
                try {
                    barrier.await();
                    for (int k=0; k< numberOfOperations / numberOfThreads; k++)
                    {
                        int bankSource = rnd.nextInt(numberOfBankAccounts);
                        int bankTarget = (bankSource + rnd.nextInt(numberOfBankAccounts-
2)+1) % numberOfBankAccounts;
                        long amount = rnd.nextInt(5000)+100; // Just a random possitive amount
                        bank.transfer(amount,bankSource,bankTarget);
                    }
                    barrier.await();
                }
                catch (InterruptedException | BrokenBarrierException ex){
                    System.out.println("Exception");
                }
            }).start();
        }
        barrier.await();
        barrier.await();

        long time = bt.getTime();

        long sum = 0L;
        for (int i=0; i<numberOfBankAccounts; i++) {
            sum += bank.balance(i);
        }

        System.out.println(className);
        System.out.println(className + " thread test: " + (sum == 0 ? "SUCCESS" : "FAILURE"));
        System.out.printf(className + " Multi thread time: %,dns\n", time );
    }
```

When transferring money from one to another we are first withdrawing from the source account and then depositing the same amount to the target account. Hence, these two values should equal 0.

If any race conditions happen, the sum from all accounts would not be 0. Therefore, we check whether the sum is 0 to see, whether there was any race conditions.

12.1.2

This implementations show that there was no race conditions.

```
class PicoBankSynchronized implements PicoBank{
    final int N; // Number of accounts
    final Account[] accounts ;

    PicoBankSynchronized(int noAccounts){
        N = noAccounts;
        accounts = new Account[N];
        for( int i = 0; i < N; i++){
            accounts[i] = new Account(i);
        }
    }

    public synchronized void transfer(long amount, int source, int target){
            accounts[source].withdraw(amount);
            accounts[target].deposit(amount);

    }
    public long balance(int accountNr){
        return accounts[accountNr].getBalance();
    }

    static class AccountSynchronized{
        // should have transaction history, owners, account-type, and 100 other real things
        public final int id;
        private long balance = 0;
        AccountSynchronized( int id ){ this.id = id;}
        public void deposit(long sum){ balance += sum; }
        public void withdraw(long sum){ balance -= sum; }
        public long getBalance(){ return balance; }
    }

    static class Account{
        // should have transaction history, owners, account-type, and 100 other real things
        public final int id;
        private long balance = 0;
        Account( int id ){ this.id = id;}
        public void deposit(long sum){ balance += sum; }
        public void withdraw(long sum){ balance -= sum; }
        public long getBalance(){ return balance; }
    }
}
```

12.1.3

We implemented a timer action as an runnable and pass it into the CyclicBarrier which then handles the time measurements:

```
class BarrierTimer implements Runnable { private boolean started;
    private long startTime, endTime;
    public synchronized void run() { long t = System.nanoTime(); if (!started) {
        started = true;
        startTime = t;
    } else
        endTime = t;
    }
    public synchronized void clear() {
        started = false;
    }
    public synchronized long getTime() { return endTime - startTime;
    }
}
```

12.1.4

```
class PicoBankLockOrder implements PicoBank{
    final int N; // Number of accounts
    final Account[] accounts ;

    PicoBankLockOrder(int noAccounts){
        N = noAccounts;
        accounts = new Account[N];
        for( int i = 0; i < N; i++){
            accounts[i] = new Account(i);
        }
    }

    public void transfer(long amount, int source, int target){

        if(source<target) {


            synchronized (accounts[source]) {
                synchronized (accounts[target]) {
                    accounts[source].withdraw(amount);
                    accounts[target].deposit(amount);
                }
            }
        }
        else {
            synchronized (accounts[target]) {
                synchronized (accounts[source]) {
                    accounts[source].withdraw(amount);
                    accounts[target].deposit(amount);
                }
            }
        }
    }
    public long balance(int accountNr){
        return accounts[accountNr].getBalance();
    }



    static class Account{
        // should have transaction history, owners, account-type, and 100 other real things
        public final int id;
        private long balance = 0;
        Account( int id ){ this.id = id;}
        public void deposit(long sum){ balance += sum; }
        public void withdraw(long sum){ balance -= sum; }
        public long getBalance(){ return balance; }
    }
}
```

12.1.5

When the balance is an atomic long, we ensure that the balance of the account object is locked during both operations - the deposit and withdraw.

Hence, two threads cannot access the deposit or withdraw while another thread is doing either one of these operations.

Therefore, we consider this a valid solution for an implementation of a thread safe class.

```
class PicoBankAtomicBalance implements PicoBank{
    final int N; // Number of accounts
    final AccountAtomicBalance[] accounts ;

    PicoBankAtomicBalance(int noAccounts){
        N = noAccounts;
        accounts = new AccountAtomicBalance[N];
        for( int i = 0; i < N; i++){
            accounts[i] = new AccountAtomicBalance(i);
        }
    }

    public void transfer(long amount, int source, int target){
        accounts[source].withdraw(amount);
        accounts[target].deposit(amount);
    }

    public long balance(int accountNr){
        return accounts[accountNr].getBalance();
    }

    static class AccountAtomicBalance{
        // should have transaction history, owners, account-type, and 100 other real things
        public final int id;
        private AtomicLong balance = new AtomicLong();
        AccountAtomicBalance( int id ){ this.id = id;}
        public void deposit(long sum){ balance.addAndGet(sum); }
        public void withdraw(long sum){ balance.addAndGet(-sum); }
        public long getBalance(){ return balance.get(); }
    }

    static class Account{
        // should have transaction history, owners, account-type, and 100 other real things
        public final int id;
        private long balance = 0;
        Account( int id ){ this.id = id;}
        public void deposit(long sum){ balance += sum; }
        public void withdraw(long sum){ balance -= sum; }
        public long getBalance(){ return balance; }
    }
}
```

12.1.6

```
class PicoBankAtomicTransfer implements PicoBank{
    final int N; // Number of accounts
    final AccountAtomicBalanceTransfer[] accounts ;

    PicoBankAtomicTransfer(int noAccounts){
        N = noAccounts;
        accounts = new AccountAtomicBalanceTransfer[N];
        for( int i = 0; i < N; i++){
            accounts[i] = new AccountAtomicBalanceTransfer(i);
        }
    }

    public void transfer(long amount, int source, int target){
        accounts[source].withdraw(amount);
        accounts[target].deposit(amount);
    }

    public long balance(int accountNr){
        return accounts[accountNr].getBalance();
    }


    static class AccountAtomicBalanceTransfer{
        // should have transaction history, owners, account-type, and 100 other real things
        public final int id;
        private AtomicLong balance = new AtomicLong();
        AccountAtomicBalanceTransfer( int id ){ this.id = id;}
        public void deposit(long sum){
            synchronized (balance){
                var amount = balance.get();
                balance.compareAndSet(amount, amount + sum);
            }
        }
        public void withdraw(long sum){
            synchronized (balance){
                var amount = balance.get();
                balance.compareAndSet(amount, amount - sum);
            }

        }
        public long getBalance(){ return balance.get(); }
    }

    static class Account{
        // should have transaction history, owners, account-type, and 100 other real things
        public final int id;
        private long balance = 0;
        Account( int id ){ this.id = id;}
        public void deposit(long sum){ balance += sum; }
        public void withdraw(long sum){ balance -= sum; }
        public long getBalance(){ return balance; }
    }
}
```

We needed to synchronize on the withdraw and deposit operations, to ensure that no race conditions can happen, as they are not atomic.

12.2.1

a) One major deficiency of the sequential functional test of the hashmap is that since you have to write out the individual tests to account for all of the possible behaviours of each of the functions of the hashmap, it is very easy to miss the test for one behaviour of the hashmap. For instance, in the sequential test for the hashmap in the StripedMap hashmap, the behaviour for removing an item that is not present in the hashmap was omitted.

b) Another deficiency is the lack of organisation or structure in the sequential function tests. Instead of just randomly testing the functionality of the hashmap, the tests could have been sequentially constructed where we would test one case where a function should work as it should immediately followed by a test case where the function is not expected to work as it should. The use of better structure in sequential tests would help us to keep easier track of the 'correctness' of our hashmap implementation

12.2.2

```
interface sumValue{
    void increment();
    void decrement();
    int get();
}

static class threadValue implements sumValue{
    int counter = 0;

    public synchronized void increment(){
        counter++;
    }

    public synchronized void decrement(){
        counter--;
    }

    public int get(){
        return counter;
    }
}
```

```java
public static void testingStrippedMap(final StripedMap<Integer, String> map) throws Exception{
    final int noCores = 2;
    final threadValue sv = new threadValue();
    final int noThreads = noCores * 2;
    final int noReps = 10;
    final CyclicBarrier barrier = new CyclicBarrier(noThreads+1);
    final int resultingSum[] = new int[1];

    Runnable manipulateHashMap = () -> {
        try{
            barrier.await(); //wait for all the threads to be ready
            Random rand = new Random(); //instance of random class


            for(int i = 0; i < noReps; i++){
                //generate a random number to formulate a key
                int upperbound = 99;
                int randomKey1 = rand.nextInt(upperbound);

                //check to see if the hashMap contains the generatedKey
                boolean keyResult = map.containsKey(randomKey1);
                if(keyResult == true){
                    map.put(randomKey1, "Value");
                }else{
                    map.put(randomKey1, "Value");
                    resultingSum[0] += randomKey1;
                    sv.increment();
                }

                //generate another random key
                int randomKey2 = rand.nextInt(upperbound);


                boolean keyResult2 = map.containsKey(randomKey2);
                if(keyResult2 == true){
                    map.putIfAbsent(randomKey2, "Value");
                }else{
                    map.putIfAbsent(randomKey2, "Value");
                    resultingSum[0] += randomKey1;
                    sv.increment();
                }


                //generate another random key
                int randomKey3 = rand.nextInt(upperbound);
                map.remove(randomKey3);
                resultingSum[0] -= randomKey1;
                sv.decrement();

            }
            barrier.await();
        }catch(Exception e){
            e.printStackTrace();
        }
    };
```

```
    for(int i = 0; i < noThreads; i++){
        new Thread(manipulateHashMap).start();
    }

    barrier.await(); //release all the threads
    barrier.await(); //wait for all of them to finish

    String result = sv.get() == resultingSum[0] ? "SUCCESS" : "FAIL";
    System.out.println(sv.get());
    System.out.println(resultingSum[0]);
    System.out.println(result);

}
```