**Exercise 9.1** This exercise is based on the program `AccountExperiments.java` (in the exercises directory for week 9). It generates a number of transactions to move money between accounts. Each transaction simulate transaction time by sleeping 50 milliseconds. The transactions are randomly generated, but ensures that the source and target accounts are not the same.

*Mandatory*

1. Use Mark7 (from Benchmark.java in the `benchmarking` package) to measure the execution time and verify that the time it takes to run the program is proportional to the number of transactions: `NO_TRANSACTION`.

The time it takes to run the program to do N transactions is proportional to the the number of the transactions specified.

```
> Task :app:AccountExperiments.main()
Running doNTransactions where N0_TRANSACTION is 5     309046095.0 ns 1263711.63          2
```

```
> Task :app:AccountExperiments.main()
Running doNTransactions where N0_TRANSACTION is 10     619825145.0 ns 1590267.57          2
```

2. Now consider the version in `ThreadsAccountExperimentsMany.java` (in the directory `exercises09`).

   Consider these four lines of the `transfer`:

```
Account min = accounts[Math.min(source.id, target.id)];
Account max = accounts[Math.max(source.id, target.id)];
synchronized(min){
  synchronized(max){
```

   Explain why the calculation of `min` and `max` are necessary? Eg. what could happen if the code was written like this:

```
Account s= accounts[source.id];
Account t = accounts[target.id];
synchronized(s){
  synchronized(t){
```

   Run the program with both versions of the code shown above and explain the results of doing this.

The `min` and `max` calculation is a deadlock-avoidance technique. It ensures the accounts are always locked in a fixed order by calculating maximum/minimum number out of them "Account min = accounts[Math.min(source.id, target.id)];" , thus preventing deadlock scenarios where threads wait on each other in conflicting lock orders.

Results can be seen in an exercise folder

```
Account s = accounts[source.id];
Account t = accounts[target.id];
synchronized(s) {
    synchronized(t) {
        source.withdraw(amount);
        target.deposit(amount);
    }
}
```

```
public void transfer(){  1 usage  Jørgen Staunstrup *
//      Account s = accounts[source.id];
//      Account t = accounts[target.id];
        Account min = accounts[Math.min(source.id, target.id)];
        Account max = accounts[Math.max(source.id, target.id)];
        synchronized(min){
          synchronized(max){
```

3. Change the program in `ThreadsAccountExperimentsMany.java` to use a the executor framework instead of raw threads. Make it use a ForkJoin thread pool. For now do not worry about terminating the main thread, but insert a print statement in the `doTransaction` method, so you can see that all executors are active.

4. Ensure that the executor shuts down after all tasks has been executed.

   Hint: See slides for suggestions on how to wait until all tasks are finished.

It is ensured by finally block where pool.shutdown is called in case that any of the threads will run into the deadlock and therefore causing that a pool will be running infinitely. If we were not using finally block, it might happen that pool would wait for a thread to finish, which won't happen due to the deadlock.

```
} finally {
  // Ensure pool is always shut down
  pool.shutdown();
  try {
    if (!pool.awaitTermination(60, TimeUnit.SECONDS)) {
      System.out.println("Forcing shutdown as tasks didn't finish in time");
      pool.shutdownNow(); // Force shutdown if tasks didn't complete in time
```

```
      }
    } catch (InterruptedException e) {
      pool.shutdownNow();
    }
  }
}
```

*Challenging*

5. Use Mark8Setup to measure the execution time of the solution that ensures termination.

   Hint: Be inspired by `PoolSortingBenchmarkable.java` (in the `code-lecture` directory for week 9)

Can be seen in my report, I added String function to avoid having printed these:

```
exercises09.ThreadsAccountExperimentsMany_Mark8$Transaction@3ef29a98
exercises09.ThreadsAccountExperimentsMany_Mark8$Transaction@2a211912
exercises09.ThreadsAccountExperimentsMany_Mark8$Transaction@6329c88d
exercises09.ThreadsAccountExperimentsMany_Mark8$Transaction@57e35e54
exercises09.ThreadsAccountExperimentsMany_Mark8$Transaction@6f5610e5
exercises09.ThreadsAccountExperimentsMany_Mark8$Transaction@122cc3e7
exercises09.ThreadsAccountExperimentsMany_Mark8$Transaction@53dcd070
exercises09.ThreadsAccountExperimentsMany_Mark8$Transaction@1d12300a
exercises09.ThreadsAccountExperimentsMany_Mark8$Transaction@1613557a
exercises09.ThreadsAccountExperimentsMany_Mark8$Transaction@30356336
exercises09.ThreadsAccountExperimentsMany_Mark8$Transaction@49977a4
exercises09.ThreadsAccountExperimentsMany_Mark8$Transaction@16
```

**Exercise 9.2** In the lecture it was shown how to estimate the maximal speed-up for Quicksort using up-to 8 threads (processors/cores) and that this estimate was consistent with Amdahls law.

1. Use the same method to estimate the maximum speed-up for Quicksort on hardware with 16 and 32 cores.

2. Estimate the maximum speed-up for sorting 1.000.000 and 10.000.000 numbers with Quicksort on hardware with 16 and 32 cores.

**Exercise 9.3** Use the code in file `TestCountPrimesThreads.java` (in the exercises directory for week 9) to count prime numbers using threads.

*Mandatory*

1. Report and comment on the results you get from running `TestCountPrimesThreads.java`.

- `countSequential` is the slowest, as expected, because it processes the entire range in a single thread without parallelization. This serves as the baseline for comparing the performance of the parallel solutions.
- **countParallelN**: This version uses multiple threads to count primes, but it shares a single `AtomicLong` counter across threads. Because of this shared counter, there is an overhead due to atomic operations and potential contention between threads. When a thread needs to update the counter, it may be put into a queue if the counter is currently being modified by another thread, which slows down performance.As the number of threads increases, the performance differences between `countParallelN` and `countParallelNLocal` become less consistent.It might be caused by thread scheduling and the overhead that is caused by accessing the shared variable in case of `countParallelN`.
- **countParallelNLocal**: In this version, each thread maintains a local count, reducing contention. Threads work independently, and only at the end are the local results aggregated. This strategy significantly reduces overhead and contention compared to `countParallelN`.: There seems to be a sweet spot around 16-24 threads where `countParallelNLocal` performs best, taking full advantage of parallelization without being significantly affected by scheduling or resource constraints, which can be seen in standard deviations as they have the lowest amount of them all in this range.

2. Rewrite `TestCountPrimesThreads.java` using Futures for the tasks of each of the threads in part 1. Run your solutions and report results. How do they compare with the results from the version using threads?

**Exercise 9.4** This exercise is a continuation of the histogram exercise in week05 (exercise 5.1).

*Mandatory*

1. Use the benchmarking tools introduced in week08 to compare the running times for `CasHistogram` and a monitor implementation (see file `HistogramLocks` in the code-exercises folder for Week 9).

   You may use the skeleton in the file `...exercise94/TestCASLockHistogram` in the code-exercises folder for Week 9).

   What implementation performs better? The monitor implementation or the CAS-based one?

   Is the result you got expected? Explain why.