## Subtask 1: A naive implementation

| Philosophers | Problem occurs at Philosoph | EatingTime Bound | Random Eating Bound | ThinkingTime Bound | Random ThinkingTime | Reason/Finding |
|---|---|---|---|---|---|---|
| 5 | 1 | 200 | - | 200 | 151 | Exception: current thread is not owner.<br><br>Thread 0 took the fork and is eating at the moment, when Thread 1 wanted to pick it up. So, when entering the put method, the Thread can't put the fork down, because it's not the owner |
| 100 | 48 | 500 | - | 50 | 47 | Exception: current thread is not owner.<br><br>Same as above, but it took more time. I guess, it is due to the probability. Maybe it will in the range if I would normalize the results. |
| 100 | 27 | 500 | 489 | 50 | 4 | Exception: current thread is not owner.<br><br>Same as above, but it took more time. I guess, it is due to the probability. Maybe it will in the range if I would normalize the results. |

With adding the synchronized keyword to the methods take and put, there is just one owner. The next Thread must wait until the resource (the fork) is available again. Nevertheless, in this solution I couldn't provoke the deadlock. I was just able to provoke an error as shown in the table above without synchronizing the methods.

## Subtask 2: Deadlock prevention

**What are the necessary conditions for deadlocks (discussed in the lecture) [0.5 points]?**

For this example, I come up with the following point:

- Circular wait
    - All philosophers could pick up the left fork. Therefore, no right fork is available, and all philosophers are waiting.

**Why does the initial solution lead to a deadlock (by looking at the deadlock conditions) [0.5 points]? Hint: if you cannot provoke a deadlock add sleeps to make it more frequent (in the lecture we also had aribrary sleeps)**

As described above, there could be the use case, that all philosophers are taking the left fork. Therefore, no right fork is available, and all philosophers are waiting for the right fork to be free. I couldn't provoke a deadlock. Maybe, I have a bug in the code, but I tried to log and debug as much as possible to find a way. With the logs, I couldn't really figure out why it worked out well with no deadlock.

**Switch the order in which philosophers take the fork by using the following scheme: Odd philosophers start with the left fork, while even philosophers start with the right hand [6 points]. Make sure to use concurrency primitives correctly!**

See code.

**Does this strategy resolve the deadlock and why [1 point]?**

Yes, because it breaks the circular wait.

**Measure the total time spent in waiting for forks and compare it to the total runtime. Interpret the measurement - Was the result expected? [3 points].**

Number of the philosophers at the table:
5

Maximal thinking time of philosophers:
500

Maximal eating time of philosophers:
5000

For this specific run I got the following result:
Runtime: 32034 ms
Waiting time: 57478 ms

At first it was quite irritating that the runtime is slower than the waiting time of the threads, but when thinking about it in more detail, it makes sense, because in this run there were 5 philosopher threads, which all had to wait at some point. With the sum of the waiting time, there is the opportunity that the waiting time is greater than the run time.

**Can you think of other techniques for deadlock prevention?**

Philosophers can only pick up both or none of the forks. So a philosopher don’t lock a certain fork, but locks both, if they are free.

**Make sure to always shutdown the program cooperatively and to always cleanup all allocated resources [2 points]**
See code.