Testat 1: Risks of concurrency

Week 4

Submission: Donnerstag 23. March 2023

1. Submission via Moodle
2. Group work (max. 2 persons) please mark in the comment when submitting

Goals:

1. Deepen and analyse the problems of race conditions.
2. Detect and fix deadlock and starvation issues.
3. Develop fair advanced synchronization primitives yourself.
4. Get started with .NET Concurrency and use checker for concurrency errors.

Task1: Broken Cyclic Barrier (Theory)

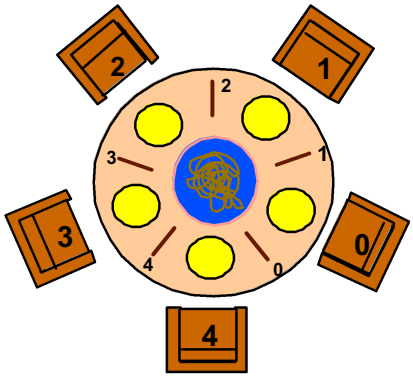
The following approach tries in vain to realize a cyclic synchronization with a latch (code in the template in Java). Determine the Concurrency Error.

```
CountDownLatch latch = new CountDownLatch(NOF_THREADS);

void multiRounds(int number) throws InterruptedException {
  for (int round = 0; round < NOF_ROUNDS; round++) {
    latch.countDown();
    latch.await();
    if (number == 0) {
      latch = new CountDownLatch(NOF_THREADS); // new latch for new round
    }
    System.out.println("Round " + round + " thread " + number);
  }
}
```

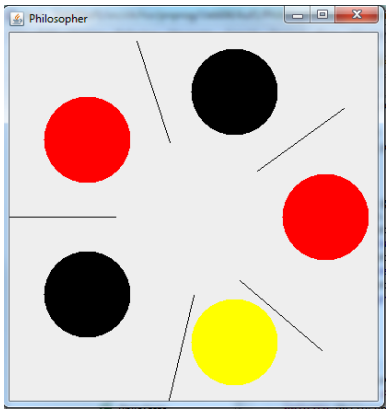
Task 2: Dining Philosophers

E. W. Dijkstra described the famous philosopher's problem in 1965 to illustrate deadlocks: Five philosophers sit at a dining table with a spaghetti bowl. In addition to thinking, philosophers also have to eat from time to time. A philosopher must use the two forks next to him to eat. Curiously, however, there are only five forks on the dining table. For these reasons, a philosopher must wait and starve as long as one of his neighbors is eating.



The template implements the scenario as a concurrent Java program. The issues show the state of the individual philosopher threads: the black circles represent thinking philosophers, the yellow eating and the red

starving. When you launch the application, after some time, the philosophers get into a deadlock situation and starve.



a) Show the deadlock with an equipment graph (Betriebsmittelgraph) (as a diagram).

b) Someone suggests the following correction. What is the problem with this approach?

```
table.acquireFork(leftForkNo);
while (!table.tryAcquireFork(rightForkNo)) {
  table.releaseFork(leftForkNo);
  // …
  table.acquireFork(leftForkNo);
}
```

Note: tryAcquireFork() tries to take the fork if it is free (atomic, without warten). The return is true if the fork could be taken or false if it is currently being used by another thread.

c) Remove the deadlock from the template from subtask a) by using a linear lock order for the forks.

Task 3: Upgradeable Read-Write Locks

Java Read-Write Locks in Java have the restriction that a thread within a read-lock section cannot elevate (upgrade) the lock to a write-lock. Therefore, a new extended read-write lock synchronization primitive with upgrade possibility is now to be implemented.

```
rwLock.upgradeableReadLock();
if (!contains(x) == 0) {
  rwLock.writeLock();
  add(x);
  rwLock.writeUnlock();
}
rwLock.upgradeableReadUnlock();
```

A special feature of the solution is that a read lock can only be increased to a write lock if this is already announced at the read lock (upgradeableReadLock() instead of just readLock()).

The synchronization primitive has three different lock types with the following compatibility matrix:

Parallel	`readLock()`	`upgradeableReadLock()`	`writeLock()`
`readLock()`	Yes	Yes	No
`upgradeableReadLock()`	Yes	No	No
`writeLock()`	No	No	No

a) Why should the upgrade request be announced at the Read Lock specifically with upgradeableReadLock()?

b) Implement such an Upgradeable Read-Write Lock class (fairness is not mandatory). You can use the scaffolding and the Unit Tests from the template. Internally, you can use a synchronization mechanism of your choice.

c) (demanding) Realize a certain fairness in the synchronization primitive: Writers should not be continuously overtaken by new readers (Starvation).

Task 4: Parallel Checker (Optional)

Use the Parallel Checker (https://parallel-checker.com to analyze the C# samples in template for concurrency errors.

a. Briefly describe the error.
b. Suggest a correction.

Example Scenario:

- For-Loop mit Thread Starts
- BankAccount with Transfer
- Paralleler Prime Scanner
- Double-Checked Locking
- Finalizers