Testatserie 1: Gefahren der Nebenläufigkeit

Semesterwoche 4

Abgabetermin: Freitag 20. März 2020

- Abgabe per Moodle
- Gruppenarbeiten (max. 2 Personen) bitte bei der Abgabe im Kommentar kennzeichnen

Zielsetzung:

- Problematik der Race Conditions vertiefen und analysieren.
- Deadlock- und Starvation-Probleme erkennen und beheben.
- Faire erweiterte Synchronisationsprimitive selber entwickeln.

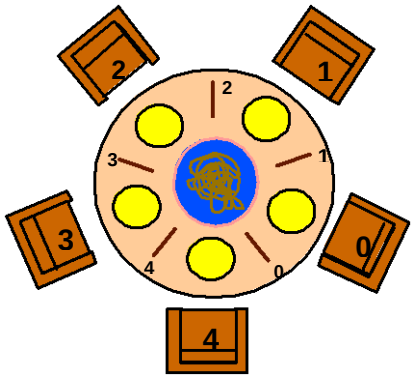
Aufgabe 1: Broken Cyclic Barrier (Theorie)

Folgender Ansatz versucht vergeblich eine zyklische Synchronisation mit einem Latch zu realisieren (Code in der Vorlage). Bestimmen Sie den Nebenläufigkeitsfehler.

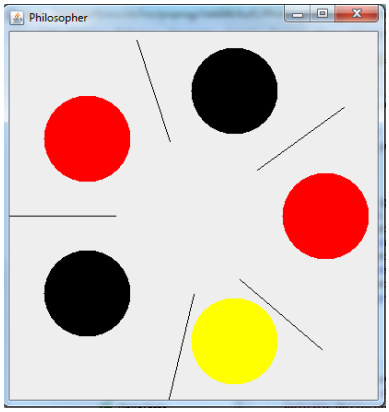
```
CountDownLatch latch = new CountDownLatch(NOF_THREADS);

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);
  }
}
```

Aufgabe 2: Dining Philosophers

E. W. Dijkstra hat 1965 das berühmte Philosophen-Problem zur Veranschaulichung von Deadlocks beschrieben: Fünf Philosophen sitzen an einem Esstisch mit einer Spaghetti-Schüssel. Neben dem Denken müssen die Philosophen auch ab und zu essen. Ein Philosoph muss zum Essen die zwei neben ihm liegenden Gabeln gebrauchen. Sonderbarerweise gibt es auf dem Esstisch jedoch nur fünf Gabeln. Aus diesen Gründen muss ein Philosoph warten und hungern, solange einer seiner Nachbarn am Essen ist.



Die Vorlage implementiert das Szenario als nebenläufiges Programm. Die Ausgaben zeigen den Zustand der einzelnen Philosophen-Threads: Die schwarzen Kreise stellen denkende Philosophen dar, die gelben essende und die roten hungernde. Wenn Sie die Anwendung starten, geraten die Philosophen nach einiger Zeit in eine Deadlock-Situation und verhungern.



a) Zeigen Sie den Deadlock mit einem Betriebsmittelgraphen (als Diagramm).

b) Jemand schlägt nachfolgende Korrektur vor. Was ist das Problem dieses Ansatzes?

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

Hinweis: `tryAcquireFork()` versucht die Gabel zu nehmen, falls sie frei ist (atomar, ohne zu warten). Die Rückgabe ist `true`, falls die Gabel genommen werden konnte, bzw. `false`, falls sie gerade von einem anderen Thread benutzt wird.

c) Beseitigen Sie den Deadlock ausgehend von der Vorlage aus Teilaufgabe a), indem Sie eine lineare Sperrordnung für die Gabeln benutzen.

Aufgabe 3: Upgradeable Read-Write Locks

Java Read-Write Locks haben die Einschränkung, dass ein Thread innerhalb eines Read-Lock Abschnitts die Sperre nicht auf einen Write-Lock erhöhen (upgraden) können. Deshalb soll nun eine neue erweiterten Read-Write Lock Synchronisationsprimitive mit Upgrade-Möglichkeit implementiert werden.

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

Speziell an der Lösung ist, dass ein Read Lock nur dann zu einem Write Lock erhöht werden kann, wenn dies bereits beim Read Lock bekannt gegeben wird (upgradeableReadLock() statt nur readLock()).

Die Synchronisationsprimitive hat drei verschiedene Lock-Arten mit folgender Kompatibilitätsmatrix:

Parallel	`readLock()`	`upgradeableReadLock()`	`writeLock()`
`readLock()`	Ja	Ja	Nein
`upgradeableReadLock()`	Ja	Nein	Nein
`writeLock()`	Nein	Nein	Nein

a) Wieso soll der Upgrade-Wunsch bereits beim Read Lock speziell mit `upgradeableReadLock()` bekannt gegeben werden?

b) Implementieren Sie eine solche Upgradeable Read-Write Lock Klasse (Fairness ist nicht zwingend). Sie können dazu das Gerüst und die Unit Tests aus der Vorlage verwenden. Intern können Sie einen Synchronisationsmechanismus Ihrer Wahl benutzen.

c) (fakultativ, anspruchsvoll) Realisieren Sie eine gewisse Fairness in der Synchronisationsprimitive: Writers sollen nicht kontinuierlich von neuen Readers überholt werden können (Starvation).