Übung 3: Spezifische Synchronisationsprimitiven

Zielsetzung:

- Semaphore, Lock & Condition, Latches und Barrieren anwenden.
- Eignungsszenarien der Synchronisationsprimitiven beurteilen.
- Performanceunterschiede von Monitor, Semaphore und Lock & Condition verstehen.

Aufgabe 1: Lock-Primitiven

Für ein Warensystem soll der Lagerbestand von einem verfügbaren Produkt verwaltet werden. Die Schnittstelle des Warenlagers ist unten angegeben (eine Art Bounded Buffer).

Mehrere Threads können dabei nebenläufig eine Anzahl von Exemplaren **auf einmal** einfügen oder herausnehmen und so den Bestand erhöhen bzw. verringern. Das Einfügen und Herausnehmen blockieren, solange der freie Platz nicht ausreicht bzw. der Bestand zu klein ist.

```
public interface Warehouse {
  // Wait until the amount can be added without exceeding the capacity.
  // If this is true, increase the stock size by the specified amount.
  void put(int amount) throws InterruptedException;
  // Wait until the stock contains at least the specified amount.
  // If this is true, decrease the stock size by the specified once.
  void get(int amount) throws InterruptedException;
}
```

Implementieren Sie das Warehouse Interface in drei Varianten:

a) **Java Monitor**
b) **Semaphore**
c) **Lock & Condition**

Sie können das Gerüst aus der Vorlage vervollständigen, welches auch eine Performance-Messung enthält.

Untersuchen Sie anschliessend folgende Fragen:

- Was sind Ihre Performance-Resultate für Ihre Umgebung?
- Was ist der Performance-Overhead der fairen Synchronisation (Semaphore und Lock & Condition)?
- Können Sie die gefundenen Resultate mittels Theorie erklären?

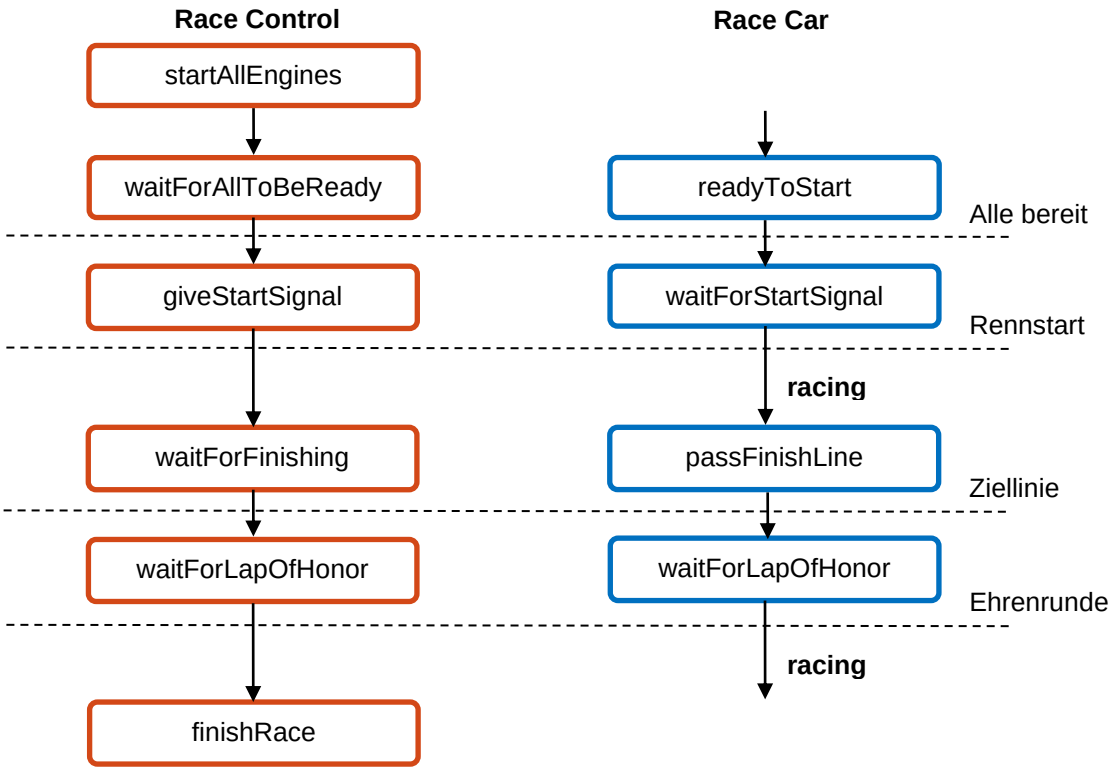
Hinweis: Beachten Sie speziell bei Lock & Condition, ob es `signal()` oder `signalAll()` braucht.

Es braucht signalAll(), da es mehrer Consumer Threads haben kann als Produces

Aufgabe 2: Latches und Barrieren

In der Vorlage finden Sie eine Simulation eines Autorennens, wobei die Autos Threads sind. Die Vorlage ist mit Monitor-Synchronisation gelöst.

Die einzelnen Schritte der Rennfahrzeuge (`RaceCar`) sowie der Rennsteuerung (`AbstractRaceControl`) während der Simulation lassen sich wie folgt illustrieren. Die gestrichelten Linien bedeuten einen Synchronisationspunkt zwischen den Threads.



Implementieren Sie die Simulation in zwei Varianten

a) **Nur mit CountDownLatches**
b) **CyclicBarriers (soweit möglich)**

Erstellen Sie hierfür Unterklassen von AbstractRaceControl, um die eigene Varianten umzusetzen. Versuchen Sie ein ähnliches Synchronisationsverhalten wie MonitorBasedRaceControl zu realisieren.

Überlegen Sie anschliessend, welcher Synchronisationspunkt mit welcher Synchronisationsprimitive aus Ihrer Sicht ideal implementiert wäre. Es gibt z.T. mehrere gleichwertige Varianten.

Alle Autos sind bereit	
Rennstart	
Passieren der Ziellinie	
Alle Autos fertig zur Ehrenrunde	
Alle Autos fertig mit der Ehrenrunde	`Thread.join()`