Aufgabe 1

Folgende Laufzeiten wurden mit einem Intel Core i7-4712MQ (4 physikalische Cores, 8 logische Kerne), 2.3 GHz, 64-bit Windows 10, mit JRE 13 gemessen. Minimum von 3 hintereinander folgenden Ausführungen in Sekunden (gerundet auf 2 signifikante Stellen).

Das Warehouse mit Lock & Condition muss in dieser Aufgabe signalAll() statt signal() benutzen (siehe Erklärung unten).

1 Producer, 1 Consumer Kapazität 5	**Zeit [s]**
Monitor (unfair)	2.8
Semaphore unfair	2.8
Semaphore fair	2.8
Lock & Condition unfair	2.9
Lock & Condition fair	9.1

5 Producer, 5 Consumer Kapazität 5	**Zeit [s]**
Monitor (unfair)	6.1
Semaphore unfair	2.5
Semaphore fair	6.2
Lock & Condition unfair	9.2
Lock & Condition fair	15

1 Producer, 10 Consumer Kapazität 5	**Zeit [s]**
Monitor (unfair)	5.1
Semaphore unfair	3.9
Semaphore fair	5.6
Lock & Condition unfair	17
Lock & Condition fair	52

100 Producer, 100 Consumer Kapazität 5	**Zeit [s]**
Monitor (unfair)	14
Semaphore unfair	3.1
Semaphore fair	6.7
Lock & Condition unfair	9.8
Lock & Condition fair	150

Der Semaphor ist in der Regel am effiziensten, weil hier gezielt die erfüllte Bedingung notifziert wird (Anzahl Releases auf entsprechendem Semaphor). Bei Monitor und Lock & Condition gibt es jeweils gegenseitiger Ausschluss, was die Performance bei einer hohen Anzahl bremst. Zudem müssen beim Monitor jeweils alle Threads bei Einfügen/Entfernen signalisiert werden. Dies gilt hier ähnlich auch bei Lock & Condition: Es müssen alle Threads der jeweiligen Bedingung (nicht leer bzw. nicht voll) (signalAll()) benachrichtigt werden. Der Grund dafür ist, dass die Threads hier eine unterschiedliche Anzahl von Elementen hineinlegen bzw. hinausnehmen können. Es gibt also quasi pro Condition-Variable mehrere semantische Unter-Bedingungen (unterschiedliche Anzahl der Elemente). Zudem kann ein einziges Einfügen gerade mehreren Consumer dienen bzw. ein Herausnehmen mehreren Producer nützen (One In – Multiple Out).

Der zusätzliche Performance-Overhead durch die faire Synchronisation ist beachtlich (z.T. mehr als Faktor 3). Das Java API implementiert die Fairness mit eigenen Warteschlangen, was vor allem bei Lock & Condition sehr ineffizient ist.

Die Messungen unterliegen gerade bei den unfairen Varianten hohen Schwankungen, weil es zu Starvation-Effekten kommen kann (wird später behandelt).

Aufgabe 2

Alle Autos sind bereit	CountDownLatch (Barriere würde Autos unnötig blockieren.)
Rennstart	CountDownLatch (RaceControl würde mit Barriere unnötig blockiert.)
Passieren der Ziellinie	CountDownLatch oder Monitor (oder volatile oder AtomicVariable - wird später behandelt)
Alle Autos fertig zur Ehrenrunde	CylicBarrier
Alle Autos fertig mit der Ehrenrunde	`Thread.join()`