Aufgabe 1a

Die Methoden sind nicht als kritische Bereiche implementiert. Das heisst, das Inkrementieren und das Dekrementieren sind nicht atomar (auch nicht das Testen-und-Dekrementieren in `withdraw()`).

Das kann dazu führen, dass einerseits jemand seinen soeben eingezahlten Betrag nicht abheben kann, obwohl jeder Thread zuerst einen Betrag einzahlt, den er dann wieder abhebt.

Zudem stimmt der Kontostand am Ende des Programms nicht unbedingt (müsste bei sauberer Synchronisation 0 sein).

Aufgabe 1b

Das korrigierte Programm ist langsamer, weil es Synchronisation einsetzt. Die Synchronisation bewirkt eine Reduktion der Parallelität und ist relativ teuer: Kontext-Wechsel der Threads, Einreihen in Wartelisten und Ordnungs-Einschränkung der Speicheroperationen (Memory Fences, wird später behandelt).

Aufgabe 2a

Das Abfragen des Kontostandes und das nachfolgende Abheben müssen atomar sein (kritischer Abschnitt). Ansonsten kann ein anderer Thread den Kontostand in Zwischenzeit bereits reduzieren, d.h. nachdem der Kontostand als genügend getestet wurde und bevor der Betrag abgehoben werden soll. Eine Thread-sichere Klasse bedeutet lediglich, dass die einzelnen Methodenaufrufe von aussen auf einer Instanz im gegenseitigen Ausschluss sind. Eventuell ist aber eine Sequenz von äusseren Methodenaufrufen auf dieser Instanz wiederum ein kritischer Bereich und müssten eigentlich atomar auf der Instanz laufen. Zur Korrektur müsste man die Logik direkt als Methode innerhalb der Thread-sicheren Klasse anbieten oder das Objekt von aussen sperren (z.B. `synchronized(account) { … }`). Letzteres kann aber zu Deadlocks führen (wird auch später genauer behandelt).

Aufgabe 3b

Folgende Laufzeiten wurden mit einem Intel Core i7-4712MQ (4 physikalische Cores, 8 logische Kerne), 2.3 GHz, 64-bit Windows 10, mit JRE 1.8 (u121) gemessen. Median von 3 hintereinander folgenden Ausführungen in Sekunden (gerundet auf 2 signifikante Stellen). Bufferkapazität ist 1.

Konfiguration	Zeit [sec]
1 Producer, 1 Consumer	17
1 Producer, 10 Consumer	47

Aufgabe 3c

Messergebnisse mit vorangehenden beschriebener Umgebung.

Ausgeglichenes Szenario:

Konfiguration (1 Producer, 1 Consumer)	Zeit [sec]
BoundedBuffer mit Monitor	17
Java ArrayBlockingQueue (non-fair)	8.9

Unausgeglichenes Szenario:

Konfiguration (1 Producer, 10 Consumer)	Zeit [sec]
BoundedBuffer mit Monitor	47
Java ArrayBlockingQueue (non-fair)	9.7

Die vorgefertigte `ArrayBlockingQueue` ist vor allem beim unausgeglichenen Fall deutlich effizienter (Grund: intern mit Lock & Conditions realisiert, wird in der nächsten Vorlesung behandelt). Die faire Variante von ArrayBlockingQueue (Parameter isFair = true) ist hier beim unausgeglichenen ca. 50-100% langsamer, aber dennoch schneller als der Monitor-basierte Buffer. Beim ausgeglichenen Fall ist die faire Implementation von ArrayBlockingQueue ca. 10% teurer im Vergleich zur nicht-fairen Variante. LinkedBlockingQueue ist hier ungefähr gleich effizient wie ArrayBlockingQueue.

Aufgabe 4

Ein einzelner `notify()` reicht hier aus, weil:

- Alle Threads im Monitor auf die gleiche Bedingung warten (ein Platz wurde frei) (*uniform waiters*)
- Die Bedingung jeweils für nur einen wartenden Thread zutreffen kann (*one-in, one-out*)

Mit `notify()` ist allerdings zu beachten, dass die Warte-Datenstruktur in Java nicht unbedingt fair sein muss, d.h. `notify()` weckt gemäss Spezifikation irgendein im Monitor wartender Thread auf. Das bedeutet, dass eventuell ein länger wartender Thread nicht bevorzugt aufgeweckt wird, sondern dauernd von weniger lang wartenden Threads überholt wird (Starvation-Problem, wird später diskutiert).

Achtung: Der aufgeweckte Thread kann von einem anderen eintretenden Thread überholt werden und der Raum nach `wait()` wieder besetzt sein. Deshalb braucht es nach wie vor die Schlaufe, um die Wartebedingung zu testen. Zusätzlich wird das durch den sogenannten „Spurious Wakeup“-Effekt notwendig.