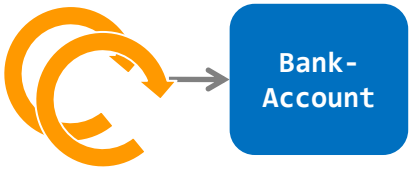
Übung 2: Monitor Synchronisation

Lernziele:

- Kritische Abschnitte erkennen und mit gegenseitigem Ausschluss schützen.
- Java Monitor Synchronisation anwenden und eigene Monitore implementieren.
- Monitor Wait und Signal Mechanismus nachvollziehen und vertiefen.

Aufgabe 1: Gegenseitiger Ausschluss

Das Programm `BankTest1` aus der Vorlage simuliert mehrere Bankkunden-Threads, die auf ein gemeinsames `BankAccount` zugreifen. Jeder der Threads zahlt jeweils einen Betrag ein und hebt diesen später wieder ab. Leider ist das Programm falsch.



a) Erweitern Sie das Programm, so dass im Falle eines ungültigen Zustandes wegen Race Conditions Exceptions geworfen werden.

b) Korrigieren Sie das Programm, so dass keine Race Conditions mehr auftreten können. Können Sie einen Geschwindigkeitsunterschied feststellen?

Aufgabe 2: Warten auf Bedingungen

Die Vorlage `BankTest2` implementiert ein Szenario mit untenstehender Kontoabhebelogik. Das Ziel ist es, von einem Konto erst dann einen Betrag abzuheben, wenn das Konto eine genügend hohe Deckung aufweist. Ansonsten wartet man.

a) Erklären Sie, wieso der untenstehende Ansatz nicht funktioniert. Korrigieren Sie das Programm.

```
BankAccount account;
…
while (account.getBalance() < amount) {
  Thread.yield();
}
account.withdraw(amount);
```

b) Unterstützen Sie für das Abheben ein Timeout. Die Methode `withdraw()` soll maximal die spezifizierte Zeit (Millisekunden) warten, bis dass das Konto mindestens den Kontostand hat. Wenn das Abheben gelingt, wird `true` zurückgegeben; sonst bei Timeout `false`.

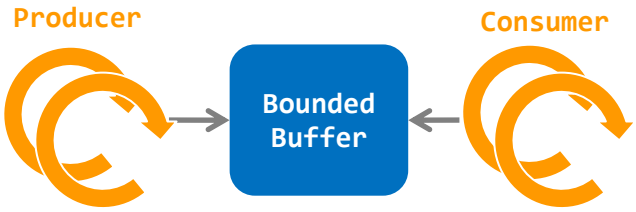
```
public synchronized boolean withdraw(int amount, long timeOutMillis)
```

Hinweise:

- `wait(long timeOut)` erlaubt es mit Timeout (Millisekunden) im Monitor zu warten.
- Das Argument `timeOut` darf nicht negativ sein. `wait(0)` enspricht `wait()` ohne Timeout.
- `System.currentTimeMillis()` gibt die aktuelle Systemzeit in Millisekunden wieder (Ticks).

Aufgabe 3: Producer-Consumer Szenario

Mehrere Producer-Threads und Consumer-Threads interagieren mit einem gemeinsamen Buffer. Die Producer fügen wiederholt Werte in den Buffer ein, während die Consumer wiederholt Werte herausnehmen.



Der Buffer hat eine beschränkte Kapazität, d.h. das Einfügen `put()` blockiert so lange, wie der Buffer voll ist. Umgekehrt blockiert das Entnehmen `get()` so lange, wie der Buffer leer ist.

a) Implementieren Sie eine `BoundedBuffer` Klasse als Monitor basierend auf der Vorlage.

b) Messen Sie die Laufzeit für folgende zwei Konfigurationen mit total je 1‘000‘000 ausgetauschten Elementen:

- 1 Producer, 1 Consumer, Bufferkapazität 1,
  je 1‘000‘000 Elemente pro Producer und Consumer
- 1 Producer, 10 Consumer, Bufferkapazität 1,
  1‘000‘000 Elemente von Producer und 100‘000 pro Consumer

Welche der zwei Konfigurationen läuft schneller und wieso?

c) In Java gibt es bereits einen Thread-sichere Bounded Buffer: `java.util.concurrent.ArrayBlockingQueue`. Setzen Sie diese Datenstruktur ein und vergleichen Sie die Performance.

Hinweis: Die Methoden `put()` und `take()` sind die blockierenden Aufrufe auf der `BlockingQueue` zum Einfügen bzw. Entfernen.

Aufgabe 4: Monitor Wait & Signal (fakultativ)

Modellieren Sie folgenden Sachverhalt mit Threads und einem Monitor: 10 Personen wollen einen Raum betreten, der allerdings nur maximal 5 Personen zulässt. Wenn der Raum voll ist und man eintreten will, muss man so lange warten, bis ein anderer den Raum wieder verlässt. Die Logik ist wie folgt:

```
room.Enter();
System.out.println("Room entered " + Thread.currentThread().getName());
Thread.sleep(1000);
room.Exit();
System.out.println("Room exited " + Thread.currentThread().getName());
```

Überlegen Sie, ob und wieso es hier wirklich die Methode `notifyAll()` braucht oder `notify()` ausreicht.