Lucerne University of
Applied Sciences and Arts

HOCHSCHULE
LUZERN

Informatik

Algorithmen & Datenstrukturen (AD)

Übung: Threads & Synchronisation (N1)

Themen: Erzeugen und Starten von Threads, Beenden von Threads, Elementare Synchronisationsmechanismen, Synchronisation durch Monitor-Konzept
Zeitbedarf: ca. 240min.

Roger Diehl, Version 1.0 (FS 2021)

1 Ballspiele (max. 60' ohne optionale Teile)

1.1 Lernziele

- Threads erzeugen (und „sterben“ lassen)
- Zugriff auf gemeinsame Ressourcen

1.2 Grundlagen

Diese Aufgabe basiert auf dem AD Input N11. Zudem erhalten Sie die Code Vorlage `ch.hslu.ad.exercise.n1.balls`.

1.3 Aufgabe

Ihre Aufgabe ist es mit Hilfe von Java ein paar bunte Bälle zu produzieren und diese der Schwerkraft auszusetzen. Benutzen Sie dazu die gegeben Klassen `Canvas` und `Circle`. Die beiden Klassen stammen aus dem OOP BlueJ Code. Das Programm soll eine 2D Grafik erstellen und muss folgende Eigenschaften aufweisen:

- Die Bildgrösse beträgt 600x400 Punkte (Standardeinstellung). Sie können aber auch eine andere Grösse in der Klasse `Canvas` einstellen.
- Der Ballradius beträgt zufällig zwischen 20 und 50 Punkten.
- Die Farbe des Balls ist zufällig aus der gegebenen Palette: `"red", "black", "blue", "yellow", "green", "magenta"` .
- Nach dem Erzeugen fällt der Ball nach unten, am unteren Bildrand „platzt“ er, d.h. der Ball verschwindet.
- Die Fallgeschwindigkeit des Balls ist zufällig.
- Erzeugen Sie viele Bälle beim Start des Programms. **Jeder Ball muss ein Thread!**
- **Optional**: Wenn der Ball den unteren Bildrand erreicht, schrumpft er, bis er nicht mehr sichtbar ist.
- **Optional**: Wenn der Ball den unteren Bildrand erreicht, bleibt er stehen. Danach wird die Farbe bei jedem Frame-Wechsel heller und durchsichtiger, bis er unsichtbar ist. Dies bedingt einen Eingriff in die Klassen `Canvas` und `Circle`.
- **Optional**: Ein Ball wird mit Drücken der (linken und/oder rechten) Maustaste generiert. Die Position des Zentrums des Balls ist beim Start an der x-y-Position der Maus. Dies ist eine etwas grössere Herausforderung.

Randbedingung: Die Ausgabe und Bewegung der Bälle können, bzw. dürfen flackern. Ziel der Aufgabe ist nicht die perfekte grafische Applikation, sondern das «Leben» eines Balles zu implementieren.

Tipps:

- Analysieren Sie zuerst die Klassen `Canvas` und `Circle`.
- Erstellen Sie für den Ball eine Klasse.
- Das „Leben“ eines Balles wird durch einen Thread „gelebt“.
- Falls Exceptions auftreten, dürfen Sie die Klassen `Canvas` und `Circle` gerne ändern.

Wie die „Ballspiele“ aussehen könnten zeigt Ihnen dieses Bild.

1.4 Reflektion

Reflektieren Sie die Aufgabe (hilft auch bei einer eventuellen Präsentation) und beantworten Sie sich die folgenden Fragen:

- Was ist in dieser Aufgabe die gemeinsame Ressource?
- Was beobachten Sie?
- Wie erklären Sie sich Programmverhalten?
- Warum überhaupt Threads verwenden?
- Wie werden die Threads erzeugt?
- Wann „sterben“ die Threads?
- Wie wird die Darstellung der Bälle aktualisiert?

2 Bankgeschäfte (ca. 90')

2.1 Lernziele

- Threads erzeugen
- Auf das Ende von Threads warten
- Zugriff auf gemeinsame Ressourcen

2.2 Grundlagen

Diese Aufgabe basiert auf dem AD Input N11 und N12. Sie erhalten die Code Vorlage `ch.hslu.ad.exercise.n1.bank`. Zudem können Sie Ihre Implementation aus der Übung D1 nutzen. Lesen Sie zuerst die ganze Aufgabe durch.

2.3 Aufgabe

Bei dieser Aufgabe ist es wichtig, dass Sie die einzelnen Punkte a), b), c) und d) nacheinander abarbeiten, auch wenn Sie die Lösung schon zu Beginn sehen. Nur so können Sie den Aha-Effekt erleben, den diese Aufgabe erzielen soll.

Gegeben ist die folgende Bankkonto Klasse.

```
/**
 * Einfaches Bankkonto, das nur den Kontostand beinhaltet.
 */
public final class BankAccount {

    private int balance;

    /**
     * Erzeugt ein Bankkonto mit einem Anfangssaldo.
     * @param balance Anfangssaldo
     */
    public BankAccount(final int balance) {
        this.balance = balance;
    }

    /**
     * Erzeugt ein Bankkonto mit Kontostand Null.
     */
    public BankAccount() {
        this(0);
    }
```

```
    /**
     * Gibt den aktuellen Kontostand zurück.
     * @return Kontostand.
     */
    public int getBalance() {
        return this.balance;
    }

    /**
     * Addiert zum bestehen Kontostand einen Betrag hinzu.
     * @param amount Einzuzahlender Betrag
     */
    public void deposite(final int amount) {
        this.balance += amount;
    }

    /**
     * Überweist einen Betrag vom aktuellen Bankkonto an ein Ziel-Bankkonto.
     * @param target Bankkonto auf welches der Betrag überwiesen wird.
     * @param amount zu überweisender Betrag.
     */
    public void transfer(final BankAccount target, final int amount) {
        this.balance -= amount;
        target.deposite(amount);
    }
}
```

a) Mit Instanzen dieser Klasse sollen Sie Überweisungen tätigen. Sie können sich vorstellen, dass im Bankalltag weltweit tausende von Überweisung pro Sekunde stattfinden. Auch sind Zugriffe denkbar, die gleichzeitig auf ein Konto gemacht werden. Diesen Sachverhalt wollen wir simulieren. Gegeben ist folgendes Szenario (`ch.hslu.ad.exercise.n1.bank`):

- Es gibt eine Liste von Quell Konten und eine Liste von Ziel Konten. Optional: Nutzen Sie Ihre eigene Listen Implementation aus der Übung D.
- Ein (grosser) Betrag, am besten immer in gleicher Höhe, soll von den Quell Konten an die Ziel Konten überwiesen werden und wieder zurück.
- Die Überweisung von der Quelle zum Ziel wird von einem Bankauftrag (Klasse `AccountTask`) mit Hilfe eines Threads gemacht.
- Die Rück-Überweisung vom Ziel zur Quelle wird von einem neuen Bankauftrag (Klasse `AccountTask`) mit Hilfe eines anderen Threads gemacht.
- Damit ein sehr grosser Betrag bei der Überweisung nicht auf dem Radar der Finanzaufsichtsbehörde erscheint, wird die Überweisung in Micro-Überweisungen mit jeweils sehr kleinen Beträgen aufgeteilt.
- Starten Sie die Bankaufträge mit der `DemoBankAccount` Klasse:

```
final Thread[] threads = new Thread[number * 2];
for (int i = 0; i < number; i++) {
    threads[i] = new Thread(new AccountTask(
        sourceBankAccounts.get(i), targetBankAccounts.get(i), amount));
    threads[i + number] = new Thread(new AccountTask(
        targetBankAccounts.get(i), sourceBankAccounts.get(i), amount));
}
for (final Thread thread : threads) {
    thread.start();
}
```

- Die Ausgabe der Liste der Quell und Ziel Konten sollte wie folgt aussehen, nachdem alle Bankaufträge durchgeführt, bzw. die Threads beendet wurden.

```
...
2021-03-05 10:17:48,496 INFO  - Bank accounts after transfers
2021-03-05 10:17:48,496 INFO  - source(0) = 100000; target(0) = 0;
2021-03-05 10:17:48,496 INFO  - source(1) = 100000; target(1) = 0;
2021-03-05 10:17:48,506 INFO  - source(2) = 100000; target(2) = 0;
2021-03-05 10:17:48,506 INFO  - source(3) = 100000; target(3) = 0;
2021-03-05 10:17:48,506 INFO  - source(4) = 100000; target(4) = 0;
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time: 1.910 s
Finished at: 2021-03-05T10:17:48+01:00
Final Memory: 7M/30M
------------------------------------------------------------------------
```

Experimentieren Sie mit verschiedenen Einstellungen:

- Anzahl der Bankkonten (Grösse der Liste)
- Anzahl Bankaufträge (Threads)
- Höhe des zu überweisenden Betrages
- Anzahl Micro-Überweisungen

Reflektion:

- Was sollte beim Szenario passieren, wenn das Programm korrekt ablaufen würde?
- Was beobachten Sie?
- Wie erklären Sie sich Programmverhalten?

b) Analysieren Sie die Bankkonto Klasse und identifizieren Sie die Schwachstelle. Wie können Sie ein noch stärkeres Fehlverhalten provozieren?

c) Eliminieren Sie die Schwachstelle mit Hilfe des elementaren Synchronisations-mechanismus wie in Folie 7 aus N12_IP_Synchronisation gezeigt. Welche Art von Synchronisation setzen Sie ein (Instanz oder Klasse)?

Reflektion:

- Welche Art von Synchronisation ist für die Bankkonto Klasse besser? Warum?
- Was beobachten Sie nun?
- Wie erklären Sie sich Programmverhalten?

d) Eliminieren Sie die Schwachstelle nun vollständig, falls immer noch Fehler passieren. Wie machen Sie das am besten?

2.4 Reflektion

Reflektieren Sie die Aufgabe (hilft auch bei einer eventuellen Präsentation) und beantworten Sie sich die folgenden grundsätzlichen Fragen (die Antworten gelten nicht nur für Java):

- Wie müssen Zugriffe auf gemeinsame Ressourcen am besten geschützt werden?
- Was sollte bei der Synchronisation in jedem Fall vermieden werden?
- Was verursacht der Einsatz von Synchronisation im Allgemeinen?

3 Das Ende eines Threads (ca. 45')

3.1 Lernziele

- Threads erzeugen
- Threads abbrechen

3.2 Grundlagen

Diese Aufgabe basiert auf dem AD Input N11, Abschnitt „Beenden von Threads“.

3.3 Aufgabe

Allgemein ist ein Thread beendet, wenn eine der folgenden Bedingungen zutrifft:

- Die `run`-Methode wurde ohne Fehler beendet.
- In der `run`-Methode tritt eine Ausnahme (`Exception`) auf, welche die Methode beendet.
- Wenn irgendwo `System.exit` aufgerufen wird.
- Der Thread wurde von aussen abgebrochen.

Den letzten Punkt, den Thread von aussen abzubrechen, sollen Sie in dieser Aufgabe nachvollziehen.

a) Implementieren Sie, nach N11_IP_Threads, eine Klasse `AdditionTask`, mit der man die Quersumme einer einfachen Zahlenreihe berechnen kann. Die `run`-Methode könnte in etwa wie folgt aussehen.

```
@Override
public void run() {
    this.runThread = Thread.currentThread();
    // Initialisierungsphase
    long sum = 0;
    // Arbeitsphase
    for (int i = this.rangeBegin; i <= this.rangeEnd; i++) {
        sum += i;
    }
    // Abschlussphase
    if (!isStopped()) {
        LOG.info(runThread.getName() + ": SUM" + n + " -> " + sum);
    }
    else {
        LOG.info(runThread.getName() + ": interrupted.");
    }
}
```

Selbstverständlich müssen Sie die Anweisungen, welche den Abbruch einleiten noch ergänzen. Es steht Ihnen frei welche Art des Abbruchs Sie wählen. Sie dürfen für diese Aufgabe auch den erzwungenen Abbruch mit der Methode `stop` einmal ausprobieren. Der Thread soll möglichst zeitnah abgebrochen werden.

b) Erstellen Sie eine Demo Applikation, die ein paar Instanzen von `AdditionTask` mit unterschiedlich langen Zahlenreihen erzeugt, dann mit jeweils einem Thread (Name nicht vergessen) startet und diesen nacheiner kurzen Zeit (z.B. 500msec) wieder abbricht.

c) Die Klasse `AdditionTask` soll bei der Berechnung etwas gebremst werden. Nach jeder Addition soll der Thread 15msec warten, bis die nächste Addition durchgeführt wird. Wiederholen Sie nun Punkt b).

3.4 Reflektion

Reflektieren Sie die Aufgabe (hilft auch bei einer eventuellen Präsentation) und beantworten Sie sich die folgenden grundsätzlichen Fragen (die Antworten gelten nicht nur für Java):

- Macht ein aktives Beenden von Threads überhaupt Sinn? Begründen Sie in jedem Fall Ihre Antwort.
- Welche Art des Abbruchs (Attribut oder Interrupt) haben Sie in Teil a) implementiert? Warum? Wie würde die andere Art des Abbruchs aussehen?
- Welche Art des Abbruchs (Attribut oder Interrupt) finden Sie besser? Begründen Sie Ihre Antwort.
- Was ist beim Abbruch mit Interrupt in Teil c) zu beachten?
- Erkennen Sie den Grund, weshalb die `stop`-Methode `deprecated` ist? Können Sie sich eine Situation vorstellen, in der man die `stop`-Methode trotzdem verwendet?

4 JoinAndSleep (ca. 45')

4.1 Lernziele

- Threads erzeugen
- Auf Threads warten
- Threads unterbrechen

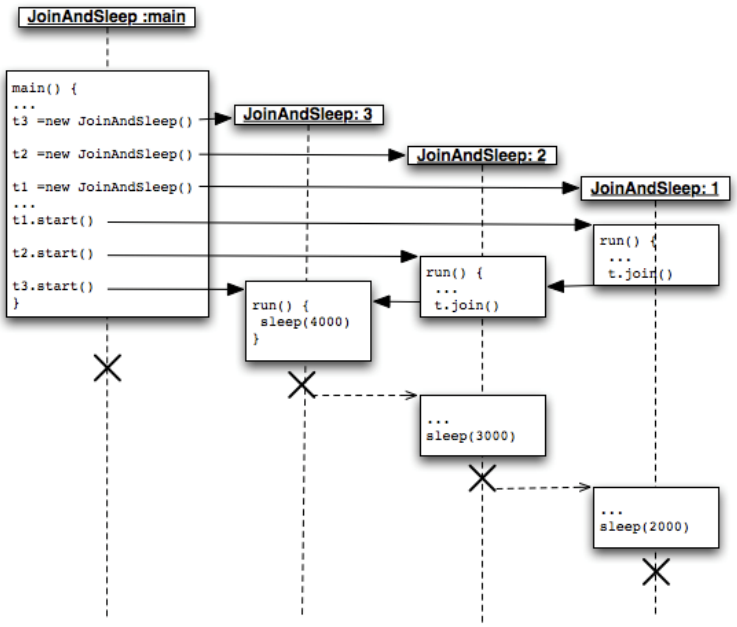
4.2 Grundlagen

Diese Aufgabe basiert auf dem AD Input N11 und der Aufgabe 4.

4.3 Aufgabe

Ziel der Aufgabe ist es drei Threads zu programmieren die auf das Beenden des jeweils anderen Threads warten und dann eine Zeit schlafen:

1) Jeder neue Zustand wird auf die Konsole ausgegeben (`LOG.info`).
2) Als erstes nach dem Start wartet der Thread bis der Ziel-Thread, auf den er referenziert, beendet ist. Ist kein Ziel-Thread referenziert, so geht der Thread sofort über zum nächstens Schritt.
3) Die Threads schlafen für eine vorgegebene Zeit in Millisekunden.
4) Die Threads beenden ordentlich.



a) Die Demo Applikation soll folgende Aktionen ausführen, wie im obigen Bild gezeigt:

- Erzeugen von Thread 3: Er soll auf keinen Thread warten und dann 4000ms schlafen
- Erzeugen von Thread 2: Er soll auf Thread 3 warten und dann 3000ms schlafen
- Erzeugen von Thread 1: Er soll auf Thread 2 warten und dann 2000ms schlafen
- Start von Thread 1
- Start von Thread 2
- Start von Thread 3

b) **Optional**: Ändern Sie die Demo Applikation, so dass ein beliebiger Thread zu einem Zeitpunkt, der innerhalb seiner „Schlafenszeit“ liegt, unterbrochen wird.

4.4 Reflektion

Reflektieren Sie die Aufgabe und beantworten Sie sich die folgenden grundsätzlichen Fragen:

- In welchem Zustand ist ein Thread, der auf einen andern Thread wartet?
- Welcher Programmteil, bzw. Thread, muss die laufenden Threads abbrechen?