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 2017)

1 Ballspiele (ca. 60')

1.1 Lernziele

- Umgang mit Swing auffrischen
- Threads erzeugen (und „sterben“ lassen)

1.2 Grundlagen

Diese Aufgabe basiert auf dem AD Input N11 und dem OOP Input O14.

1.3 Aufgabe

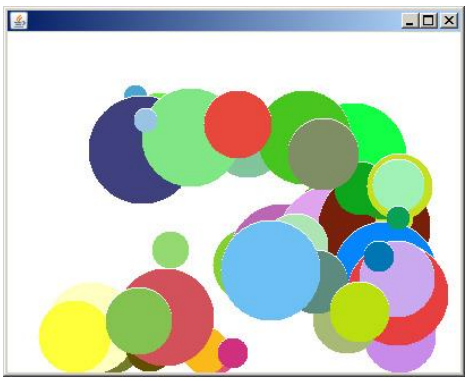
Ihre Aufgabe ist es mit Hilfe von Java (und Ihrer Entwicklungsumgebung) ein paar bunte Bälle zu produzieren und diese der Schwerkraft auszusetzen. Das Programm soll mit Hilfe von Swing eine 2D Grafik erstellen und muss folgende Eigenschaften aufweisen:

- Die Bildgrösse beträgt mindesten 600x400 Punkte. Kann aber auch eine flexible Grösse haben.
- Ein Ball wird mit Drücken der (linken und/oder rechten) Maustaste generiert.
- Der Ballradius ist zufällig zwischen 20 und 50 Punkten.
- Die Position des Zentrums des Balls ist beim Start an der x-y-Position der Maus.
- Die Farbe des Balls ist eine zufällige Rot-Grün-Blau Farbe.
- Nach dem Erzeugen fällt der Ball nach unten, am unteren Bildrand bleibt er stehen.
- Die Fallgeschwindigkeit des Balls ist zufällig.
- **Optional**: Wenn der Ball steht, wird die Farbe bei jedem Frame-Wechsel heller und durchsichtiger, bis er unsichtbar ist.

Tipps:

- **Wichtig**: Halten Sie das GUI einfach. Es ist kein MVC für die Lösung dieser Aufgabe notwendig.
- Erstellen Sie für den Ball eine Klasse.
- Das „Leben“ eines Balles wird durch einen Thread „gelebt“.
- Sorgen Sie dafür, dass der Thread am Ende (wenn der Ball verblasst ist) auch wirklich stirbt.
- Erstellen Sie eine Fenster-Applikation und nicht eine Dialog-Applikation.
- Als Zeichnungsfläche nehmen Sie ein `JPanel`, die Events lassen Sie diese Ihre Entwicklungsumgebung erstellen.
- Falls Sie sich nicht (mehr) mit Swing auskennen hilft Ihnen dieses Tutorial weiter: https://docs.oracle.com/javase/tutorial/uiswing/
- Einen Überblick über Swing bekommen Sie hier: https://de.wikipedia.org/wiki/Swing_(Java)

Wie die „Ballspiele“ aussehen könnten zeigt Ihnen der kleine Film `balls.mov`.



1.4 Reflektion

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

- Warum überhaupt Threads verwenden?
- Wie werden die Threads erzeugt?
- Wann „sterben“ die Threads?
- Wie wird die Darstellung der Bälle aktualisiert?

2 Hello World (ca. 45')

2.1 Lernziele

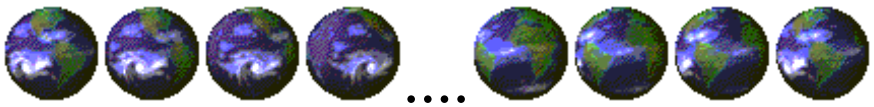
- Umgang mit Swing auffrischen
- Threads erzeugen
- Mit Threads interagieren

2.2 Grundlagen

Diese Aufgabe basiert auf dem AD Input N11 und dem OOP Input O14. Zudem können Sie Ihre Resultate aus der Übung D1 verwenden. Sie dürfen auch auf dem Resultat aus Aufgabe 1 weiter aufbauen.

2.3 Aufgabe

Um eine Animation zu erstellen müssen Bilder in einem Frame geladen und nacheinander angezeigt werden. Sind die Bilder so gestaltet, dass jedes Bild einen Teil einer Bewegung darstellt, so spielt das Frame für den Betrachter einen Film ab. Die folgenden Bilder stellen die Erddrehung dar.



`IMG0001.GIF` .... `IMG0018.GIF`

Die Bilder stehen Ihnen im ILIAS als `worlddata.zip` zur Verfügung. Selbstverständlich können Sie auch eigene Bilder, bzw. Bildserien verwenden. Es muss auch nicht eine Weltkugel sein. Allerdings muss die Richtung der Drehbewegung klar erkennbar sein.

a) Laden Sie die 18 Bilder der Erde in eine Liste. Es ist wichtig, dass Sie einen schnellen Zugriff auf die Elemente in der Liste haben. Stellen Sie die Bilder `IMG0001.GIF` bis `IMG0018.GIF` so dar, dass eine Animation der Erddrehung entsteht. Damit die Erddrehung nicht zu schnell wird, sollten Sie ca. 50 msec zwischen den einzelnen Bildern warten. Mit dem Start der Applikation soll die Erde dargestellt werden und sich drehen.

b) Die Erweiterung der Applikation besteht darin, die Erddrehung umzukehren. Auf den Mausklick hin, sind die Bilder in umgekehrter Richtung (IMG0018.GIF bis IMG0001.GIF) darzustellen. Ein erneuter Mausklick kehrt die Drehrichtung wieder um.

Tipps:

- **Wichtig**: Halten Sie das GUI einfach. Es ist kein MVC für die Lösung dieser Aufgabe notwendig.
- Erstellen Sie eine Fenster-Applikation und nicht eine Dialog-Applikation.
- Als Zeichnungsfläche nehmen Sie ein `JPanel`, die Events lassen Sie diese Ihre Entwicklungsumgebung erstellen.
- Starten Sie nie einen Thread im Konstruktor.
- Um die Bilddateien aus einer SDK-Ressource zu laden, verwenden Sie folgenden Code:

```
String path = "verzeichnis/bilddatei";
URL url = getClass().getClassLoader().getResource(path);
Image img = ImageIO.read(url);
```

2.4 Reflektion

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

- Benötigen Sie überhaupt einen oder mehrere Threads? Begründen Sie.
- Welche Thread Implementierung verwenden Sie? Begründen Sie.
- Wo starten Sie den/die Threads?
- Wie viele Threads starten Sie?
- Wie wird die Darstellung der Welt aktualisiert?
- Wann „sterben“ die Threads?
- Was passiert, wenn die Applikation gestoppt wird?

3 Bankgeschäfte (ca. 90')

3.1 Lernziele

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

3.2 Grundlagen

Diese Aufgabe basiert auf dem AD Input N11 und N12. Zudem können Sie Ihre Resultate aus der Übung D1 nutzen.

3.3 Aufgabe

Gegeben ist die folgende Bankkonto Klasse.

```java
/**
 * 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. Erstellen Sie deshalb folgendes Szenario:

- Es gibt eine Liste von Quell Konten und eine Liste von Ziel Konten. Als Liste können Sie eine eigene Implementation aus der Übung D1 nutzen.
- 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 (Thread) gemacht.
- Die Rück-Überweisung vom Ziel zur Quelle wird von einem andern Bankauftrag (anderer Thread) gemacht.
- Damit ein sehr grosser Betrag bei der Überweisung nicht auf dem Radar der Finanzaufsichtsbehörde erscheint, teilen Sie die Überweisung in Micro-Überweisungen mit jeweils sehr kleinen Beträgen auf.
- Starten Sie die Bankaufträge (Threads) ähnlich (oder gleich) wie am Beispiel Counter in der OOP Folie 8 (O16_IP_Synchronisation) gezeigt.
- Geben Sie die Liste der Quell Konten aus, nachdem alle Bankaufträge durchgeführt, bzw. die Threads beendet wurden.

Experimentieren Sie mit verschiedenen Einstellungen:

- Grösse der Bankkonto Listen
- 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 korrekt. Wie machen Sie das am besten?

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):

- 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?

4 Das Ende eines Threads (ca. 45')

4.1 Lernziele

- Threads erzeugen
- Threads abbrechen

4.2 Grundlagen

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

4.3 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.

Die letzte Möglichkeit, den Thread von aussen abzubrechen, sollen Sie in dieser Aufgabe nachvollziehen.

a) Implementieren Sie, nach der Idee in den Folien 27+28+32 aus 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;
    }
    // Aufräumphase
    if (!isStopped()) {
        System.out.println(runThread.getName() + ": SUM" + n + " -> " + sum);
    }
    else {
        System.out.println(runThread.getName() + ": interrupted.");
    }
}
```

Selbstverständlich müssen Sie die Anweisungen, welche den Abbruch einleiten noch ergänzen. Es steht Ihnen frei welche Art des Abbruchs Sie wählen.

Der Thread soll zeitnah abgebrochen werden. Aber Sie können **optional** auch einen Algorithmus in `AdditionTask` einfügen, der entscheidet, ob bei einem bestimmten Stand der Berechnung, den Task wirklich beendet oder die Berechnung noch zu Ende führt.

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).

d) Zum Schluss, einfach damit Sie das einmal probiert haben. Stoppen Sie die Thread mit der `stop`-Methode. Was beobachten Sie?

4.4 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 einen Grund, weshalb die `stop`-Methode `deprecated` ist? Können Sie sich eine Situation vorstellen, in der man die `stop`-Methode trotzdem verwendet?

5 Optional: JoinAndSleep

5.1 Lernziele

- Threads erzeugen
- Auf Threads warten
- Threads unterbrechen

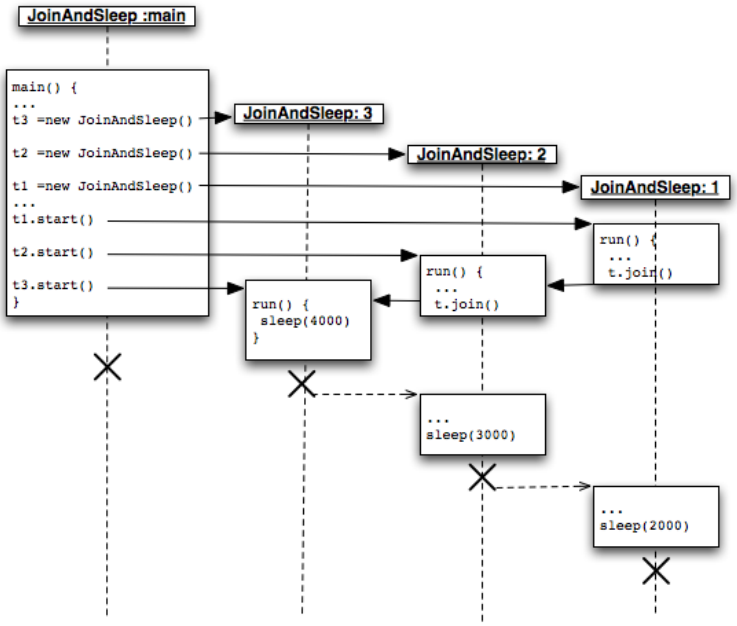
5.2 Grundlagen

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

5.3 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 der Konsole ausgegeben
2) Als erstes nach dem Start wartet der Thread bis ein anderer Thread, auf den er zeigt, beendet hat. Ist kein anderer Thread referenziert, so geht er sofort über zum nächstens Schritt.
3) Die Threads schlafen für eine vorgegebene Zeit in Millisekunden.
4) Die Threads beenden sich.



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

5.4 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 muss die Threads abbrechen?