VU Grundlagen der Programmkonstruktion — Übungsteil

Organisatorisches

Wie sollen Ihre Lösungen aussehen

Beachten Sie, dass nicht die konkrete Lösung im Vordergrund steht, sondern der Weg, wie Sie zu dieser gelangen. Geben Sie daher bei Aufgaben immer auch den Rechenweg an. Bei jeder Aufgabe ist ein Beispiel angeführt, wie dieser aussehen soll. Ohne Angabe des Rechenwegs erhalten Sie für die Aufgabe keine Punkte.

Geben Sie ihre Lösungen im TUWEL als (bevorzugt) Text-Dokument (Erweiterung `.txt`) oder als PDF-Dokument (Erweiterung `.pdf`) ab.

Gruppenarbeiten

Sie können die Aufgaben in Gruppen zu 2-6 Personen lösen. Die Zusammenstellung und Organisation der Gruppen ist dabei Ihnen überlassen.

Wenn Sie die Aufgabe in einer Gruppe gelöst haben, so soll EIN Gruppenmitglied die Aufgabe in TUWEL abgeben. Die Namen MIT Matrikelnummern aller Gruppenmitglieder müssen im abgegebenden Dokument am Anfang ihres Dokuments vermerkt sein.

Abgabe Ihrer Lösungen

Die Abgabe Ihrer Lösungen erfolgt elektronisch via TUWEL (TU Wien E-Learning Center). Sie finden einen Link dorthin am Beginn der LVA-Seite im TISS ("Zum TUWEL Online-Kurs").

Die Abgabe muss via TUWEL bis spätestens 12. Jänner 2012, 12:00, erfolgen. Nachträgliche Abgaben werden nicht berücksichtigt.

Fragen

Bei Fragen zu TUWEL wenden Sie sich bitte zunächst an das TUWEL-Hilfe-System und Ihre Kollegen. Sollte in der Angabe etwas unklar sein, sehen Sie bitte im TUWEL nach, ob dort bereits ergänzende Anmerkungen zum Übungsteil stehen. Wichtige Ankündigungen zum Übungsteil werden auch in der Vorlesung und online bekannt gegeben.

5. Übungsaufgabe

1 Exceptions

Gegeben ist folgende Klasse `IntNumber`, mit einer zusätzlichen Methode `div`, die eine neue Instanz von `IntNumber` erzeugt, welche das Ergebnis von `this` dividiert durch `that` (Parameter) enthält:

```
public class IntNumber {
    private int value;

    public IntNumber(int initValue) {
        this.value = initValue;
    }

    public boolean isZero() {
        return value == 0;
    }

    public IntNumber div(IntNumber that) {
        return new IntNumber(this.value / that.value);
    }

    public String toString() {
        return Integer.toString(value);
    }
}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    IntNumber num1 = new IntNumber(sc.nextInt());
    IntNumber num2 = new IntNumber(sc.nextInt());

    System.out.println(num1.div(num2));
}
```

Aufgabe

In den folgenden Teilaufgaben sollen Sie den Fehlerfall *Division durch 0* behandeln.

Teil 1 - Lösung ohne Exceptions

Änderin Sie die Methode `main` so ab, dass, wenn eine Division durch 0 auftritt, die Meldung `Division durch 0` ausgegeben wird. Benutzen Sie dazu *keine* Exceptions.

Teil 2 - `Exception`

- Geben Sie eine Klasse `DivByZeroException` an, die bei einer Division durch 0 geworfen werden soll. Erweitern Sie dabei die Klasse `Exception`.
- Ändern Sie die Methode `div` so ab, dass, falls eine Division durch 0 auftritt, eine Instanz von `DivisionByZeroException` geworfen wird.
- Änderin Sie die Methode `main` so ab, dass, wenn eine Division durch 0 auftritt, die Meldung `Division durch 0` ausgegeben wird.

Teil 3 - `RuntimeException`

- Geben Sie eine Klasse `DivByZeroRuntimeException` an, die bei einer Division durch 0 geworfen werden soll. Erweitern Sie dabei die Klasse `RuntimeException`.
- Ändern Sie die Methode `div` so ab, dass, falls eine Division durch 0 auftritt, eine Instanz von `DivByZeroRuntimeException` geworfen wird.
- Änderin Sie die Methode `main` so ab, dass, wenn eine Division durch 0 auftritt, die Meldung `Division durch 0` ausgegeben wird.

Welche Unterschiede können Sie zwischen den Lösungen von Teil 1, Teil 2 und Teil 3 erkennen?

2 Datei lesen - Teil 1

Schreiben Sie ein Programm, das die Zeilen einer Datei (der Dateiname ist im String `filename`) zählt und am Ende ausgibt. Benutzen Sie dazu die Methode `readLine` in `BufferedReader` und gehen Sie von folgendem Code-Fragment aus.

Achten Sie darauf, dass Sie im Fehlerfall eine sinnvolle Fehlermeldung ausgeben und alle offenen Streams schließen.

```
import java.io.*;

public class LineCount {
    private static final String errmsg = "Usage: java LineCount <in>";

    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println(errmsg);
            return;
        }

        String filename = args[0];

        int lineCount = 0;

        // ... Ihr Programmcode ...

        System.out.println(lineCount);
    }
}
```

3 Dateien lesen und schreiben - Teil 2

Schreiben Sie ein Programm, welches zwei Dateien (die Dateinamen sind in den Strings `inFilename1` und `inFilename2`) parallel zeilenweise einliest. Sollten zwei zugleich eingelesene Zeilen gleich sein, so soll diese Zeile einmal in die Datei `outFilename` geschrieben werden (siehe Beispiel unten).

Achten Sie darauf, dass Sie im Fehlerfall eine sinnvolle Fehlermeldung ausgeben und alle offenen Streams schließen.

```
import java.io.*;

public class Intersect {
    private static final String errmsg = "Usage: java Intersect <in1> <int2> <out>";

    public static void main(String[] args) {
        if (args.length != 3) {
            System.err.println(errmsg);
            return;
        }

        String inFilename1 = args[0];
        String inFilename2 = args[1];
        String outFilename = args[2];

        // ... Ihr Programmcode ...
    }
}
```

Beispiel

Gegeben sind die folgenden Dateien:

File1:

```
foo1
foo2
bar1
bar2
```

File2:

```
foo0
foo2
bar1
bar0
```

Der Aufruf `java Intersect File1 File2 OutFile` schreibt folgende Zeilen in die Datei `OutFile`:

```
foo2
bar1
```

4 Deadlocks* (Zusatzaufgabe, wird nicht beurteilt)

In der Programmierung mit mehreren Threads kann es zu vielen Problemen kommen, wenn sich Threads Resourcen miteinander teilen müssen (und ihre Zugriffe auf diese koordinieren müssen). Ein Deadlock tritt z.B. dann auf, wenn ein Thread A auf eine Resource warten, die einem Thread B zugeteilt ist, der ebenfalls auf eine Resource wartet, die allerdings Thread A zugeteilt ist. In diesem Fall blockieren sich beide Threads (deadlock).

Ein Beispiel für einen Deadlock ist das Philosophen-Problem (siehe z.B. `http://de.wikipedia.org/wiki/Philosophenproblem`). Hier sitzen (üblicherweise) fünf Philosophen im Kreis. Jeder Philosoph hat einen Teller mit Spaghetti vor sich. Zwischen je zwei Tellern liegt eine Gabel (d.h. es sind fünf Gabeln am Tisch).

Die Philosophen denken nun. Sobald sie hungrig sind, greifen sie zuerst zur Gabel links von ihrem Teller (die darauf ihrem linken Nachbarn nicht mehr zur Verfügung steht), danach zu der rechten Gabel. Mit beiden Gabeln beginnen sie nun zu essen, und danach legen Sie beide Gabeln zurück. Sollte eine Gabel gerade von einem der Nachbarn benutzt werden, wartet der Philosoph, bis der Nachbar die Gabel zurücklegt (ohne jedoch eine bereits genommene Gabel zurückzulegen), nimmt sie danach und beginnt zu essen.

Das Philosophenproblem ist in den folgenden Klassen und Methoden implementiert (für drei Philosophen):

```
public class Gabel {
    private boolean genommen = false;

    public synchronized void nehmen() { // Atomare Operation
        while(genommen) {
            try {
                System.out.println("Gabel besetzt");
                wait();
            } catch(InterruptedException e) {
                // Ignore
            }
        }

        genommen = true;
    }

    public synchronized void zuruecklegen() {
        genommen = false;
        notifyAll();
    }
}
```

Jeder Philosoph ist ein eigener Thread:

```
public class Philosoph implements Runnable {

    private int index;

    private Gabel links;
    private Gabel rechts;

    public Philosoph(int index, Gabel links, Gabel rechts) {
        this.index = index;

        this.links = links;
        this.rechts = rechts;
    }

    public void essen() {
        System.out.println("Philosoph " + index + " will essen");
        links.nehmen();
        System.out.println("Philosoph " + index + " hat linke Gabel genommen");

        rechts.nehmen();
        System.out.println("Philosoph " + index + " hat rechte Gabel genommen");

        // Philosoph isst nun.
        System.out.println("Philosoph " + index + " hat gegessen.");

        links.zuruecklegen();
        rechts.zuruecklegen();
    }

    public void denken() {
        try {
            // 1 Millisekunde warten.
            System.out.println("Philosoph " + index + " denkt");
            Thread.sleep(1);
        } catch(InterruptedException e) {
            // Ignorieren
        }
    }

    public void run() {
        while(true) {
            denken();
            essen();
        }
    }
}
```

Die `main`-Methode sieht folgendermaßen aus:

```
public static void main(String[] args) {
    final int anzahl = 3;

    // Tisch decken
    Gabel[] gabeln = new Gabel[anzahl];

    for(int i = 0; i < anzahl; i++) {
        gabeln[i] = new Gabel();
    }

    // Philosophen anlegen
    Philosoph[] philosophen = new Philosoph[anzahl];

    for(int i = 0; i < anzahl; i++) {
        philosophen[i] = new Philosoph(i, gabeln[i], gabeln[(i + 1) % anzahl]);
    }

    // Philosophen "starten"
    for(Philosoph phil : philosophen) {
        new Thread(phil).start();
    }
}
```

Aufgabe

- Wie sieht ein Deadlock in dieser Aufgabe aus? Geben Sie konkret ein Beispiel an und erläutern Sie es.
- Geben Sie Möglichkeiten an, in dieser Aufgabe Deadlocks zu vermeiden. Versuchen Sie dabei zu vermeiden, dass die Philosophen miteinander kommunizieren müssen und einzelne Philosophen verhungern.