Praktikum Concurrency 1

1. Dreiecksfläche: Warmup mit Exceptions

Die Fläche eines Dreiecks mit den Seitenlängen `a, b, c` kann mit folgender Formel berechnet werden

```
flaeche = Math.sqrt(s * (s - a) * (s - b) * (s - c));
```

wobei `s = (a + b + c)/2` ist.

Aufgaben:

a) Schreiben Sie eine Methode, welche die Fläche eines Dreiecks aus den Seiten a, b, c (verwenden Sie den Typ `double`) berechnet. Die Methode soll eine **`IllegalArgumentException`** werfen, falls ungültige Werte übergeben werden oder die drei Seiten kein Dreieck darstellen. Überlegen Sie sich, welche Bedingungen erfüllt sein müssen, damit es sich um ein Dreieck handelt.

b) Testen Sie Ihre Methode mit Hilfe von Unit-Tests. Welche Fälle müssen getestet werden? (Vergessen Sie nicht die Grenzfälle).
Tipp: Das Werfen von Exceptions testen Sie in JUnit 4.x mit folgender Annotation:

```
@Test(expected=IndexOutOfBoundsException.class)
public void testIndexOutOfBoundsException() throws Exception { ... }
```

c) Erstellen Sie eine eigene **checked** Exception: `NotATriangleException`, die geworfen wird wenn die Angaben kein gültiges Dreieck ergeben. Was muss alles angepasst werden? Ist das Verhalten Ihrer *checked* Exception anders als bei der originalen *unchecked* Variante?

2. Printer-Threads: Verwendung von Java Threads

Nachfolgend einige Basisübungen zum Starten und Stoppen von Threads in Java.

```
class PrinterThread extends Thread {
    char ch;
    int sleepTime;

    public PrinterThread(String name, char c, int t) {
        super(name);
        ch = c; sleepTime = t;
    }

    public void run() {
        System.out.println( getName() + " run laueft an");
        for (int i = 1; i < 100; i++) {
            System.out.print (ch);
            try {
                Thread.sleep(sleepTime);
            } catch (InterruptedException e) { ; }
        }
```

```
        System.out.println( '\n' + getName() + " run  fertig");
    }
}

public class Printer1 {
    public static void main(String[] arg) {
        PrinterThread a = new PrinterThread("PrinterA", '.', 0);
        PrinterThread b = new PrinterThread("PrinterB", '*', 0);
        a.start();
        b.start();
        b.run(); //Wie kann das abgefangen werden?
    }
}
```

a) Studieren Sie das Programm Printer1.java: Die Methode **`Thread.run()`** ist **`public`** und kann daher direkt aufgerufen werden. Erweitern Sie die Methode **`run()`** so, dass diese sofort terminiert, wenn sie direkt und nicht vom Thread aufgerufen wird.
Tipp: Was liefert die Methode **`Thread.currentThread()`** zurück?

b) Schreiben Sie das Programm so um, dass die run-Methode über das Interface **`Runnable`** implementiert wird.
Führen Sie dazu eine Klasse **`Printer`** ein, die das Interface **`Runnable`** implementiert.
Starten Sie zwei Threads, so dass dieselbe Ausgabe entsteht wie bei a).
Verwenden Sie den untenstehenden Code für den Test ihrer Implementation:

```
public class Printer2 {
    public static void main(String[] arg) {
        Printer a = new Printer('.', 0);
        Printer b = new Printer('*', 0);
        Thread t1 = new Thread(a, "PrinterA");
        Thread t2 = new Thread(b, "PrinterB");
        t1.start();
        t2.start();
    }
}
```

c) Wie kann man es erreichen, die Fairness zu erhöhen, d.h. dass der Wechsel zwischen den Threads häufiger erfolgt? Wirkt es sich aufs Resultat aus?

d) Wie muss man das Hauptprogramm anpassen, damit der Main-Thread immer als letztes endet?

3. Konto-Übertrag

Nachfolgend eine einfache Klasse, um ein Konto zu verwalten, den Saldo abzufragen oder zu aktualisieren.

```
class Account {
    private int id;
    private int saldo = 0;

    public Account(int id, int initialSaldo) {
        this.id = id;
        this.saldo = initialSaldo;
    }
    public int getId() {
        return id;
    }
    public int getSaldo () {
        return saldo;
    }
    public void changeSaldo (int delta) {
        this.saldo += delta;
    }
}
```

Ein Entwickler implementiert aufbauend auf der Klasse `Account` eine Operation für den Transfer eines Geldbetrages zwischen zwei Konti. Die Klasse `AccountTransferThread` implementiert dazu die Methode `accountTransfer`, welche in einer Schleife mehrfach aufgerufen wird, um viele kleine Transaktionen zu simulieren. Das Testprogramm `AccountTransferTest` (siehe abgegebenen Code) erzeugt schlussendlich mehrere Threads, die teilweise auf denselben Konto-Objekten operieren.

```
class AccountTransferThread extends Thread {
    private Account fromAccount, toAccount;
    private int amount;
    public AccountTransferThread (Account fromAccount, Account toAccount,
                                  int amount) {
        this.fromAccount = fromAccount;
        this.toAccount = toAccount;
        this.amount = amount;
    }

    public void accountTransfer() {
        if (fromAccount.getSaldo() > amount) { //do not overdraw account
            fromAccount.changeSaldo(-betrag);
            toAccount.changeSaldo(betrag);
        }
    }

    public void run() {
        for (int i=0; i<10000; i++) {
            accountTransfer();
        }
    }
}
```

a) Was stellen Sie fest, wenn Sie das Testprogramm laufen lassen?
Erklären Sie wie die Abweichungen zustande kommen.

b) Im Unterricht haben Sie gelernt, dass sie kritische Bereiche Ihres Codes durch Mutual-Exclusion geschützt werden sollen. Wie macht man das in Java?
Versuchen Sie mit Hilfe von Mutual-Exclusion sicher zu stellen, dass keine Abweichungen entstehen. Reicht es, wenn Sie die kritischen Methoden in Account schützen?
Untersuchen Sie mehrere Varianten von Locks (Lock auf Methode oder Block, Lock auf Instanz oder Klasse).
Ihre Implementierung muss noch nebenläufige Transaktionen erlauben, d.h. wenn Sie zu stark synchronisieren, werden alle Transaktionen in Serie ausgeführt und Threads machen keinen Sinn mehr.
Stellen Sie für sich folgende Fragen:
   - Welches ist das Monitor-Objekt?
   - Braucht es eventuell das Lock von mehr als einen Monitor während der Transaktion?

c) (optional) Wenn Sie es geschafft haben die Transaktion thread-safe zu implementieren, ersetzen Sie in `AccountTransferTest` die die folgende Zeile

```
AccountTransferThread t1 =
    new AccountTransferThread("Worker 1", account3, account1, 1);
```

durch

```
AccountTransferThread t1 =
    new AccountTransferThread("Worker 1", account1, account3, 1);
```

und starten Sie das Programm noch einmal. Was stellen Sie fest? (evtl. müssen Sie es mehrfach versuchen, damit der Effekt auftritt). Was könnte die Ursache sein und wie können Sie es beheben? (Tipp: Deadlocks?)

4. Thread Priority

Diese Aufgabe vermittelt einen Einblick in das Prioritätensystem der Java Threads. Zur Aufgabe gehört die vorgegebene Klasse `ThreadPrioDemo.java`. Diese muss für Teile der Aufgabe modifiziert werden. Mit Hilfe der Klasse soll das Verhalten von Java Threads mit verschiedenen Prioritäten analysiert werden.
Hinweis: Es kann sein, dass verschiedene Betriebssysteme und Java-Versionen sich unterschiedlich verhalten (siehe auch http://www.javamex.com/tutorials/threads/priority.shtml).

Je nach Priorität im Bereich von `Thread.MIN_PRIORITY=1` über `Thread.NORM_PRIORITY=5` bis `Thread.MAX_PRIORITY=10`, sollte der Thread vom Scheduler bevorzugt behandelt werden, d.h. der Zähler `count` sollte häufiger inkrementiert werden.
Folgende Fragen müssen abgeklärt und beantwortet werden:

a) Wie verhält es sich, wenn alle Threads die gleiche Priorität haben?

b) Was stellen Sie fest, wenn die Threads unterschiedliche Priorität haben? Erhöhen Sie auch die Anzahl Threads (z.B. 100), um eine Ressourcen-Knappheit zu provozieren.