Modul Verteilte Software WS2022/23
Aufgaben 4

1. Threads

1.1
Erstellen Sie eine Anwendung, die die Veränderung einer Datei innerhalb eines Threads beobachtet.
Erstellen Sie dazu eine von der Klasse Thread abgeleitete Klasse (z.B. Beobachter) mit dem Objekt der Klasse **File** als Attribut. Übergeben Sie an den Konstruktor der Klasse den Namen der zu überwachenden Datei und öffnen Sie dort die Datei (new File(dateiname)).
In der *run()*-Methode der Klasse soll in der regelmäßigen Zeitabständen in einer endlosen Schleife (solange bis Thread angehalten wird) mit *lastModified()* die Zeit der letzten Veränderung abgefragt werden und die entsprechende Meldung ausgegeben werden.
Die main-Methode einer anderen Klasse soll nach dem Start des Threads auf die Eingabe des Kommandos „Stop" warten und beim Eintreffen des Kommandos den Thread mit *interrupt()* anhalten.

1.2.
Es ist eine Anwendung zu erstellen, die der Umsetzung des folgenden Verhaltensmusters dient:
Der Opa öffnet ein Taschengeld-Konto für seine Enkel und zahlt regelmäßig auf dieses Konto beliebige Geldbeträge ein. Seine Enkel entnehmen diesem Konto verschiedene Beträge und übertragen sie auf ihre eigenen Konten.

a). Implementieren Sie erst die Teilaufgabe: die Enkel buchen vom Konto, das zur Beginn auf einen Wert initialisiert wird, solange verschiedene Beträge ab, bis das Konto leer ist.

Erstellen Sie dazu die Klasse **Bank** mit dem Attribut Kontostand. Mit der get-Methode soll es möglich sein, den aktuellen Kontostand abzufragen. Zwei weitere Methoden
void ab(double betrag) und
void zu(double betrag)
realisieren das Abbuchen bzw. die Einzahlung. Vorm Abbuchen ist zu überprüfen, ob genug Geld auf dem Konto zur Verfügung steht.
Die Instanz der Klasse Bank wird in der main-Methode der Klasse **Opa** erzeugt und an die Enkel-Objekte übergeben.

Die von der Klasse Thread abgeleitete Klasse **Enkel** besitzt neben dem Attribut Bank die Attribute Name, Kontostand sowie eine Wartezeit in Millisekunden. Der Transfervorgang, der in dieser Klasse zu implementieren ist, soll in einer Schleife innerhalb der *run()*-Methode durchgeführt werden. Der jeweilige Auszahlungsbetrag soll zufällig generiert werden. Zur Realisierung des Zusammenspiels zwischen den Threads verwenden Sie die Methoden *Thread.sleep()* und *Thread.yield()*.

Zum Testen der Klasse **Enkel** sind im main-Methode der Klasse **Opa** *drei* Instanzen der Klasse **Enkel** anzulegen (oder mehr, aber mit zu vielen Instanzen verliert man die Übersicht über das Testergebnis) und zu starten.

b). Erweitern Sie das Programm mit den Einzahlungen durch den Großvater.

Die Klasse **Opa** realisiert innerhalb der main-Methode in einer Schleife die Einzahlungen beliebiger Beträge auf das Konto der Klasse Bank solange bis der Betrag <=0 eingegeben wird. Verwenden Sie für die Eingabe Klasse Scanner.
Halten Sie die Enkel-Threads mit ***interrupt()*** an. Vergessen Sie nicht auch „wartende" Threads zu beenden.

Verändern Sie die Klasse **Enkel** so, dass die Schleife in der run-Methode erst dann beendet wird, wenn der Thread angehalten wird. Die Abfrage darüber ist mit der Methode ***isInterrupted()*** möglich.
Hinweis: statt die Threads einzelnen anzuhalten kann mit System.exit(0) das komplette Programm beendet werden.
Verändern Sie die Implementierung der Methoden *ab()* und *zu()* so, dass beim Abheben vom nicht ausreichendem Guthaben der jeweilige Vorgang über ***wait()*** blockiert, und nachdem Gutschreiben von Opa über ***notifyAll()*** fortgesetzt werden kann.

Synchronisieren Sie die Methoden *ab()* und *zu()* unter Verwendung des Schüsselworts ***synchronized*** und mit Hilfe von ***Lock***.

Verwenden Sie beim Synchronisieren mit Lock die Klasse *java.util.concurrent.ReentrantLock* zum Markieren der kritischen Abschnitte. Mit dem Objekt der Klasse *java.util.concurrrent.Condition* können die Threads in Wartezustand versetzt und wiedergeweckt werden.

Definition der Objekte:

static ReentrantLock lock=new ReentrantLock();
static Condition condition=lock.newCondition();

Benötigte Methoden:

lock(), unlock(), signalAll(), await()

2. ExecutorService

Zur Auswertung mit einer Anwendung liegen mehrere Dateien vor, die die stündliche Verfügbarkeit verschiedener Server an einem Tag in der Form protokollieren:

available 2020-11-11T01:00:01
available 2020-11-11T02:00:01
not available 2020-11-11T03:00:01
not available 2020-11-11T04:00:01
available 2020-11-11T05:00:01
available 2020-11-11T06:00:01

In der Anwendung soll mit Hilfe eines **ExecutorServices** (z.B. Executors.*newFixedThreadPool*) ermittelt werden wie oft die Server insgesamt ausgefallen sind. Einzelne Dateien sind jeweils in einer Interface ***Callable*** implementierenden Klasse auszuwerten. Die Ergebnisse sind über **Future**-Objekte an die main-Funktion einer anderen Klasse zu übermitteln, wo diese aufsummiert und ausgegeben werden.

Erfassen Sie die Dateinamen in einem Array, z.B.

```
String[] files={"Server1.txt","Server2.txt","Server3.txt"};
```

und die Future-Objekte in einer ArrayList:

```
ArrayList<Future<Long>> list=new ArrayList<Future<Long>>();
```

3. Blocking Queue

Viele Thread-Synchronisationsprobleme lassen sich mittels der threadsicheren Warteschlangen (Queue) elegant lösen, z. B. mit der Klasse **ArrayBlockingQueue**.
Der Transaktionsmanager einer Bank soll in dem zu erstellenden Programm mittels Blocking-Queue die Transaktionen zwischen den Bankkonten verwalten.

- Jedes Konto (Klasse **Account**) verfügt dabei über eine Kontonummer und einen Kontostand s. Klassendiagramm).
- Die Transaktionen sind Objekte der Klasse **Transaktion**, die neben dem Betrag die Ein- und Ausgangskontonummer besitzen.
- In der **main()** – Methode der Klasse **TransactionHandler** werden mehrere Objekte der Klasse Transaction generiert und der Warteschlange (dem Objekt der Klasse **ArrayBlockingQueue**) hinzufügt.
- In der ***run()*** - Methode der Klasse **Bank** werden die einzelnen Transaktionen verarbeitet indem sie aus der Warteschlange geholt werden (Methode ***take()***) und die entsprechenden Kontostände aktualisiert werden.
- Die Objekte der Klassen ArrayBlockingQueue, Bank sind in der main()-Methode der Klasse **TransactionHandler** anzulegen und der Thread ist zu starten.
- Verwenden Sie zum Speichern von Accounts eine Collection-Klasse, z.B. HashMap.