TU Berlin
Software and Business Engineering
Prof. Dr. Ingo Weber
Dr. Tobias Heindel
Richard Hobeck
Martin Grambow

WiSe 2020/21
Aufgabe 12
Block 4
8. Februar 2021

Abgabetermin: 14. Februar 2021

Aufgabe 12 *Einführung in die Programmierung mit Java*

Hinweise

- Die Abgabe dieser Übungsaufgaben muss bis spätestens Sonntag, den 14. Februar 2021 um 23:59 Uhr im ISIS-Kurs erfolgt sein. Es gelten die Ihnen bekannten Übungsbedingungen.
- Lösungen zu diesen Aufgaben sind als gezippter Projektordner abzugeben. Eine Anleitung zum Zippen von Projekten finden Sie auf der Seite des ISIS-Kurses. *Bitte benutzen Sie einen Dateinamen der Form* `VornameNachname.zip`.
- Bitte beachten Sie, dass Abgaben im Rahmen der Übungsleistung für die Zulassung zur Klausur relevant sind. Durch Plagiieren verwirken Sie sich die Möglichkeit zur Zulassung zur Klausur in diesem Semester.

In der Vorlesung haben Sie das „Consumer-Producer“-Problem kennengelernt, für den Spezialfall eines Puffers, eines „Consumer“s und eines „Producer“s. In der ersten Aufgabe gibt es zwei Variationen. Zum einen geht es um das „Einfügen“ und “Einsortieren“ in eine *Liste* (anstatt des „Schreibens“ und „Lesens“ in einen *Puffer*) und zum anderen gibt es für das Einfügen und Einsortieren jedes einzelnen Elements in eine fixe Liste einen eigenen Thread, sodass es eine Vielzahl von Threads gibt, die geeignet miteinander interagieren sollten.

```
public class List<T> {
    private volatile ListEl<T> head;

    public ListEl<T> getHead() {return head;}

    public void setHead(ListEl<T> head) {this.head = head;}

    public List(){
        head = null;
    }
}

class ListEl<T>{
    T val;
    ListEl<T> next;

    ListEl(T v, ListEl<T> n){
        val = v;
        next = n;
    }
}
```

Abbildung 1: Die Klasse für Listen

Aufgabe 12.1 *Nebenläufige Operationen auf Listen* (3 Punkte)

- Schreiben Sie eine Klasse `ListInserter`, die das `Runnable`-Interface implementiert und deren Konstruktor eine Liste vom Typ `List<Integer>` und einen Thread als Formalparameter hat. Die `run`-Methode soll auf jeden Fall ein kurze Zeit warten (per Aufruf der `sleep`-Methode) bevor Sie einen zufälligen Wert an den Kopf der Liste einfügt. Der übergebene Thread des Konstruktors dient der Möglichkeit, mit einem weiteren Thread zu „kommunizieren“ (über Methoden wie `wait()`, `notify()`, `join()`, etc.).
- Schreiben Sie eine Klasse `ListBubbler`, die das `Runnable`-Interface implementiert und deren Konstruktor eine Liste vom Typ `List<Integer>` und einen Thread als Formalparameter hat.

Die `run`-Methode durchläuft die Liste und tauscht aufeinander folgende Elemente aus, falls der `val`-Wert des ersten der beiden Elemente größer ist als der des zweiten. Der übergebene Thread des Konstruktors dient der Möglichkeit, mit einem weiteren Thread zu „kommunizieren“ (über Methoden wie `wait(), notify(), join()`, etc.).

- Schreiben Sie eine `main`-Methode in der Klasse `List`, die eine neue, leere Liste erzeugt, dann geeignet je 40 Threads von neuen `ListInserter`- und `ListBubbler`-Objekten erzeugt, startet und schließlich die Liste ausgibt, nachdem alle Threads vollständig ausgeführt sind.

Eine mögliche Ausgabe, mit zusätzlichen „Nachrichten“ der Threads, ist wie folgt:

```
ListInserter@209928d8 insterted 29.
ListInserter@499626a3 insterted 89.
New element 29 was "sorted" by ListBubbler@62ac0033.
ListInserter@927b287 insterted 7.
New element 7 was "sorted" by ListBubbler@21d9783.
ListInserter@9a8cc17 insterted 72.
New element 72 was "sorted" by ListBubbler@7eba351e.
ListInserter@6ca972b3 insterted 95.
ListInserter@6bad06fa insterted 13.
New element 13 was "sorted" by ListBubbler@4a99f9a6.
ListInserter@3eb69766 insterted 76.
New element 76 was "sorted" by ListBubbler@53ef2d05.
ListInserter@5cfdb626 insterted 98.
ListInserter@19c9fdc insterted 99.
ListInserter@4a14494 insterted 83.
ListInserter@5f41bac insterted 59.
New element 83 was "sorted" by ListBubbler@ff2ceca.
New element 59 was "sorted" by ListBubbler@2764fbfd.
ListInserter@22f1c89c insterted 6.
New element 6 was "sorted" by ListBubbler@154aed9b.
ListInserter@21fb6f50 insterted 70.
New element 70 was "sorted" by ListBubbler@152c3b82.
ListInserter@76a7b3dc insterted 29.
New element 29 was "sorted" by ListBubbler@6bb47690.
ListInserter@7bddccb1 insterted 19.
New element 19 was "sorted" by ListBubbler@86c59d6.
ListInserter@13a6f77f insterted 73.
New element 73 was "sorted" by ListBubbler@eeed4d4.
ListInserter@150a7ec8 insterted 92.
New element 92 was "sorted" by ListBubbler@283642b1.
ListInserter@75f9d2f8 insterted 22.
ListInserter@5c90f164 insterted 92.
New element 22 was "sorted" by ListBubbler@6a2007ad.
ListInserter@4e202e1f insterted 7.
New element 92 was "sorted" by ListBubbler@5f57cf75.
ListInserter@66305d30 insterted 10.
New element 7 was "sorted" by ListBubbler@6dd11338.
ListInserter@1033a3e4 insterted 83.
New element 10 was "sorted" by ListBubbler@215df48d.
ListInserter@7ffe95c3 insterted 90.
New element 83 was "sorted" by ListBubbler@20d8c0b2.
ListInserter@2e73747d insterted 67.
New element 89 was "sorted" by ListBubbler@173b8a36.
ListInserter@30f2e78b insterted 28.
New element 67 was "sorted" by ListBubbler@1e488fcf.
New element 28 was "sorted" by ListBubbler@688a8307.
ListInserter@70dd1b75 insterted 55.
New element 55 was "sorted" by ListBubbler@57372cf0.
ListInserter@174aef46 insterted 11.
New element 11 was "sorted" by ListBubbler@75f29075.
ListInserter@167c8476 insterted 56.
New element 56 was "sorted" by ListBubbler@73caab39.
ListInserter@2e99d278 insterted 91.
ListInserter@58f25711 insterted 5.
New element 91 was "sorted" by ListBubbler@38c97494.
New element 5 was "sorted" by ListBubbler@71e43bd4.
ListInserter@43948f41 insterted 9.
New element 9 was "sorted" by ListBubbler@211cd654.
ListInserter@28b09aef insterted 26.
New element 26 was "sorted" by ListBubbler@63bac131.
ListInserter@6ab3edc insterted 56.
New element 56 was "sorted" by ListBubbler@6f0a2bee.
ListInserter@d8ea65 insterted 29.
New element 29 was "sorted" by ListBubbler@14394e8b.
ListInserter@5dca1a9c insterted 96.
New element 96 was "sorted" by ListBubbler@72125d1c.
ListInserter@5bdb174b insterted 65.
New element 65 was "sorted" by ListBubbler@6655a692.
ListInserter@283266d0 insterted 52.
New element 52 was "sorted" by ListBubbler@2295b4fc.
ListInserter@7a17549a insterted 19.
New element 19 was "sorted" by ListBubbler@665512ae.
ListInserter@6b632e0d insterted 29.
ListInserter@3d55665b insterted 0.
New element 29 was "sorted" by ListBubbler@2c212e9b.
New element 0 was "sorted" by ListBubbler@42b258ac.
[0, 5, 6, 7, 7, 9, 10, 11, 13, 19, 19, 22, 26, 28, 29, 29, 29, 29, 52, 55, 56, 56, 59, 65, 67, 70, 72, 73, 76, 83, 83, 89, 90, 91, 92, 92, 95, 96, 98, 99]
```

Hinweis Die Hauptschwierigkeit der Aufgabe liegt darin, die Threads *geeignet* aufeinander warten zu lassen, sodass die Liste am Ende sortiert ist und sortiert ausgegeben wird.

Ein Problem bei der Verwendung von (verschachtelten) `synchronized`-Blöcken sind „deadlocks“. Ein klassisches, einfaches, erklärendes Beispiel sind die speisenden Philosoph*innen.

Dining Philosophers Problem

Die zwei zentralen Punkte dieses Beispiels bestehen darin, dass wir (1) begrenzte Ressourcen haben (in der Form von Essstäbchen) und dass (2) Prozesse oder Threads u.U. mehrere Ressourcen auf einmal benutzen – allerdings nur sequenziell blockieren können. In der nächsten Aufgabe sollen Sie den potenziellen „deadlock“ des zyklischen Wartens (auf das nächste Essstäbchen) unter Benutzung von `synchronized`-Blöcken reproduzieren.

```
public class Philosopher extends Thread{
    public final Object chopstick;
    final Boolean meFirst;
    private MyState state = MyState.THINKING;
    Philosopher next;

    public Philosopher(Object chopstick, Boolean whofirst) {
        this.chopstick = chopstick;
        meFirst = whofirst;
    }


    public static void main(String[] args) {
        final int count = 10;
        Philosopher[] philosophers = new Philosopher[count];

        philosophers[count-1].next = philosophers[0];
        for(int i=0; i<count-1; i++) {
            philosophers[i].next = philosophers[i+1];
        }
        for(int i=0; i< count; i++){
            philosophers[i] = new Philosopher(i,true);
        }
        for(int i=0; i< count; i++){
            philosophers[i].start();
        }
    }
}
```

```
public enum MyState {
    THINKING,
    HUNGRY
}
```

Abbildung 2: Die Speisenden Philosoph*innen

Zur großen Tafel der speisenden Philosoph*innen bringen alle Teilnehmer*innen ihre eigenen Essstäbchen (Engl. `chopstick`) mit, allerdings nur eins pro Person; das genügt jedoch erfahrungsgemäß, da Philosoph*innen den Hauptteil Ihrer Zeit mit Denken verbringen.
Die Eigenart der Philosoph*innen ist, dass, wenn sie hungrig geworden sind, sie zum Essen

- zwei Stäbchen *nacheinander* aufnehmen,
- und sie ein einmal aufgenommenes Stäbchen nicht wieder zurückgeben, sondern geduldig darauf warten, dass das noch fehlende Stäbchen wieder verfügbar wird.

Die Philosoph*innen nehmen entweder ihr eigenes Stäbchen zuerst, oder das der nächsten Person am Tisch (im Uhrzeigersinn); diese beiden Möglichkeiten sollten den beiden Möglichkeiten, die Variable `meFirst` zu belegen, entsprechen.
Die `main`-Methode der Klasse `Philosopher` erzeugt eine mögliche Konstellation (mit zehn Teilnehmer*innen).

(a) Alle Philosoph*innen sind in einer zyklischen Liste angeordnet.

(b) Alle Philosoph*innen nehmen immer zuerst ihr eigenes Stäbchen.

(c) Die entsprechenden Threads werden alle gestartet.

Aufgabe 12.2 *Große Tafel der Speisenden Philosoph*innen* **(2 Punkte)**

Ihre Aufgabe besteht darin, eine `run()` zu implementieren, sodass alle Philosoph*innen am Tisch zunächst eine Zeit lang im Zustand `THINKING` verbringen. Danach werden sie allerdings nacheinander `HUNGRY`, woraufhin jede Person versucht beide Essstäbchen *nacheinander* „aufzunehmen“; das „Aufnehmen“ eines Essstäbchens entspricht dem Eintreten in einen `synchronized`-Block, der das

entsprechende Objekt der `chopstick`-Referenz „blockiert“. Im Beispiel wird immer zuerst das eigene Stäbchen aufgenommen und danach das Stäbchen der nächsten Person in der Liste (welche durch die `next`-Referenz gegeben ist).

Hinweis

> *Fügen Sie unbedingt eine kurze Wartezeit zwischen Aufnahme des ersten Essstäbchens und Aufnahme des zweiten Essstäbchens ein.*

Eine mögliche Ausgabe sieht dann wie folgt aus.

```
Thread[Thread-5,5,main] has a new thought.
Thread[Thread-1,5,main] has a new thought.
Thread[Thread-3,5,main] has a new thought.
Thread[Thread-4,5,main] has a new thought.
Thread[Thread-6,5,main] has a new thought.
Thread[Thread-8,5,main] has a new thought.
Thread[Thread-7,5,main] has a new thought.
Thread[Thread-9,5,main] has a new thought.
Thread[Thread-0,5,main] has a new thought.
Thread[Thread-2,5,main] has a new thought.
```

Anmerkung Wenn Sie alles richtig gemacht haben, dann ergibt sich der Zustand des zyklischen Wartens (was die obige Ausgabe nur schlecht zu erkennen gibt). Welche Änderungen in der `main`-Methode sind nötig, um diesen „deadlock“ zu vermeiden?