Praktikum Grundlagen der Programmierung

Prof. Dr. Harald Räcke, R. Palenta, A. Reuss, S. Schulze Frielinghaus WS 2016/17

Übungsblatt 12 Abgabefrist: 30.01.2017 (vor 05:00 Uhr)

Σ Punkte: 21 | Σ Bonuspunkte: 0

Aufgabe 12.1 (P) Threads: Extends vs. Implements

In Java gibt es mehrere Möglichkeiten einen Thread zu erstellen. Schauen wir uns zwei geläufige Varianten an:

```
1  class A extends Thread {
2
3      @Override
4      public void run() {
5          System.out.println("Hello Student!");
6      }
7  }
8
9  class B implements Runnable {
10
11     @Override
12     public void run() {
13         System.out.println("Hello Student!");
14     }
15 }
16
17 public class ThreadTest {
18
19     public static void main(String[] args) {
20         Thread t1 = new A();
21         Thread t2 = new Thread(new B());
22         t1.start(); t2.start();
23     }
24 }
```

Der Unterschied zwischen den Klassen `A` und `B` ist, dass Klasse `A` von der Klasse `Thread` erbt, wobei die Klasse `B` das Interface `Runnable` implementiert. In den Zeilen 20 und 21 werden dann jeweils Thread-Objekte erstellt. Darauffolgend, in Zeile 22, werden die Threads dann schlussendlich gestartet. Thread `t1` als auch Thread `t2` berechnen jeweils das gleiche werden aber auf unterschiedliche Arten erstellt. Nun stellt sich natürlich die Frage, auf welche Art und Weise sollten denn nun Threads erstellt werden? Die kurze Antwortet lautet: In den meisten Fällen sollte ein Thread via einem *Runnable*-Objekt erstellt werden – so wie in der Klasse `B`.

Schauen wir uns das mal etwas genauer an:

- Implementiert eine Klasse `Foo` das Interface `Runnable`, ist es immer noch möglich, dass die Klasse `Foo` von einer anderen Klasse, z.B. `Bar`, erbt.

- Eine andere Frage, die man sich hier stellen kann ist: Will man an dieser Stelle Vererbung verwenden oder vielleicht doch lieber Komposition von Objekten? Sprich ist das ein Problem welches durch Vererbung gelöst werden sollte? Eigentlich nicht!

- Eine weitere Frage, die hier aufkommt: Möchte man wirklich die eigentlich zu berechnende Aufgabe an eine Threadimplementierung koppeln? Könnte uns das später hindern, die Aufgabe via einem anderen Konzept zu lösen? Vielleicht über Futures oder Ähnlichem. Sprich auch hier ist die Antwort, dass die Implementierung via einem `Runnable` *leichtgewichtiger* erscheint. An dieser Stelle sei noch angemerkt, dass die Klasse `Thread` das Interface `Runnable` implementiert.

Aufgabe 12.2 (P) Sesselfabrik

Eine Sesselfabrik produziert ununterbrochen Sessel, d.h. sie belegt freie Plätze im Lager und erhöht die Anzahl bestellbarer Sessel. Es gibt Platz für maximal 20 Sessel. Lieferanten bestellen Sessel für Sessel, bis sie ihren Bedarf gedeckt haben und holen sie anschliessend aus dem Lager. Während dieser Abholaktion belegen Sie exklusiv die Rampe zum Lager der Fabrik. Verwenden Sie die Programmgerüste.

Bearbeiten Sie die folgenden Aufgaben unter Vermeidung von Busy-Waiting:

1. Erweitern Sie `SesselFabrik` um die Methoden:
   - `sesselEinlagern`, die einen Sessel einlagert und bestellbar macht, sobald dafür genügend Platz vorhanden ist. Falls ein Lieferant darauf wartet, daß ein Sessel bestellbar wird, soll dieser entsprechend benachrichtigt werden.
   - `sesselBestellen`, die von Lieferanten aufgerufen wird, um jeweils *einen* Sessel zu bestellen. Falls im Moment kein Sessel mehr bestellbar ist, soll der aufrufende Lieferant entsprechend warten.
   - `sesselAbholen`, die von Lieferanten aufgerufen wird, um jeweils *einen* Sessel abzuholen. Die Sesselfabrik soll dabei benachrichtigt werden, dass wieder ein Platz im Lager freigeworden ist.
   - `rampeBelegen`, die von Lieferanten aufgerufen wird, wenn sie zum Lager fahren und dementsprechend die Rampe belegen wollen. Wenn die Rampe bereits belegt ist, soll der aufrufende Lieferant warten, bis sie wieder frei ist.
   - `rampeFreigeben`, die von Lieferanten aufgerufen wird, wenn sie ihre Ware abgeholt haben und die Sesselfabrik wieder verlassen. Falls ein Lieferant darauf wartet, dass die Rampe frei wird, soll dieser entsprechend benachrichtigt werden.
2. Überschreiben Sie nun die `run`-Methode der Klasse `SesselLieferant`. Hier sollen zunächst die feste Anzahl `anzahl` von Sesseln bei der Sesselfabrik `sf` bestellt und schliesslich abgeholt werden. Ausserdem soll darauf geachtet werden, dass nur jeweils ein Lieferant gleichzeitig die Rampe der Fabrik benutzen kann.

Aufgabe 12.3 (H) Map [9 Punkte]

Verwenden Sie für diese Aufgabe nur die erlaubten Java-Methoden (siehe Anhang).

Ein *Map* wendet eine Funktion auf jedes Element in einem Array an und speichert die Resultate in einem neuen Array. Die Ordnung der Elemente aus dem ursprünglichen Array spiegelt sich in den Elementen des resultierenden Arrays wieder. Beispiel, sei die Funktion $f\colon \mathbb{Z} \to \mathbb{N}$ und ein Array $[i_1, \ldots, i_k]$ mit Elementen aus $\mathbb{Z}$ gegeben. Das resultierende Map ist dann gegeben als $[f(i_1), \ldots, f(i_k)]$, wobei die Elemente alle aus $\mathbb{N}$ kommen.

Im Folgenden wollen wir solch ein Map implementieren. Da wir davon ausgehen, dass unsere Funktion f sehr teuer ist, wollen wir das Map *parallel* implementieren. Dafür zerlegen wir ein Array der Länge k in $n \in \mathbb{N}$ viele Bereiche. Dabei gilt, dass $k\%n$ viele Bereiche die Größe $\lfloor k/n \rfloor + 1$ haben und alle übrigen Bereiche die Größe $\lfloor k/n \rfloor$. Jeder Bereich soll von einem Thread bearbeitet werden.

Für die Implementierung gehen wir wie folgt vor. Wir definieren ein Interface `Fun<T, R>`, welches die Schnittstelle zu einer (unären) Funktion $f\colon$ `T` $\to$ `R` beschreibt:

```
public interface Fun<T, R> {
    public R apply(T x);
}
```

Schreiben Sie eine öffentliche Klasse `Map`, welche die Methode

```
public static <T, R> void map(Fun<T, R> f, T[] a, R[] b, int n)
        throws InterruptedException
```

bereitstellt. Die Methode `map` erstellt `n` viele Threads. Jedem Thread wird ein unterschiedlicher Bereich aus dem Array `a` zugeteilt, auf dem ein jeweiliger Thread die Funktion `f` auf die Elemente anwendet. Wenn die Methode zurückkehrt muss folgendes gelten: `f(a[i])` = `b[i]` für alle validen Indizes `i`. Des Weiteren müssen dann alle in der Methode `map` gestarteten Threads beendet sein. Überlegen Sie sich, mit welchen nicht sinnvoll behandelbaren bzw. ungültigen Parametern die Methode `map` aufgerufen werden kann und fangen Sie diese Fälle über eine `IllegalArgumentException` ab. D.h. die Methode `map` muss für alle möglichen Parameter das korrekte Ergebnis liefern oder eine `IllegalArgumentException` werfen. Überlegen Sie sich, wie Sie die Resultate von den einzelnen Threads in der Methode `map` akkumulieren können.

Schreiben Sie eine Klasse `IntToString` welche das Interface `Fun<Integer, String>` implementiert. Die Methode `apply` soll einen Integer entgegennehmen und die jeweilige Stringrepräsentation des Integers zurückgeben. Testen Sie Ihre parallele Map-Implementierung mit der Funktion.

Lösungsvorschlag 12.3

Korrekturbemerkungen:

- Diese Fälle müssen mittels einer Exception behandelt:
  `f == null || a == null || b == null || a.length > b.length || n < 0`
  pro Fall 1/2 Punkte (insgesamt also 5/2 Punkte
- Der Fall `a.length < b.length` kann auch ignoriert werden
- Der Fall `n > a.length` kann auch mittels einer Exception gelöst werden. Falls irgendwie sinnvoll gelöst, dann 1/2 Punkte
- Bereiche *sinnvoll* aufgeteilt: 2 Punkte
- `n` viele Threads erstellt/gestartet: 1 Punkt
- `b` enthält die richtigen Ergebnisse: 1 Punkt

- die Reihenfolge der Ergebnisse in `b` ist richtig: 1 Punkt
- die in `map` gestarteten Threads sind beendet bevor `map` beendet wird: 1 Punkt
- Implementierung nicht getestet oder Fehler im Test (z.B. in der Klasse IntToString): 1 Punkt Abzug

Aufgabe 12.4 (H) Rock-Paper-Scissors [12 Punkte]

Verwenden Sie für diese Aufgabe nur die erlaubten Java-Methoden (siehe Anhang).

Im Folgenden wollen wir das Spiel „Schere-Stein-Papier“ implementieren. Dafür sollen zwei Threads gestartet werden, die gegeneinander das Spiel spielen. Die Threads sollen via einer Zufallszahl sich für Schere, Stein oder Papier entscheiden und das Ergebnis schlussendlich an den Main-Thread weiterleiten. Welcher Thread gewonnen hat, wird via dem Main-Thread ausgegeben. D.h. der Computer spielt gegen sich selbst. Wir verwenden ganze Zahlen um Schere, Stein und Papier darzustellen:

- Schere $=:= 0$
- Stein $=:= 1$
- Papier $=:= 2$

Die öffentliche Klasse `Player` implementiert das Interface `Runnable`. Die `run`-Methode entscheidet sich zufällig für einen Wert (Schere, Stein, Papier) und wartet so lange, bis der Wert von dem Main-Thread gelesen wurde. Sobald der Wert gelesen wurde, muss ein neuer Wert zufällig berechnet werden. Dies soll beliebig oft wiederholt werden. Der berechnete Wert soll via der Methode `public int getChoice() throws InterruptedException` ausgelesen werden können. Die Methode blockiert den Aufrufer so lange, bis ein neuer Wert berechnet wurde, i.e., der `getChoice`-Aufrufer wartet bis die `run`-Methode einen neuen Wert berechnet hat und die `run`-Methode wartet auf einen `getChoice`-Aufrufer um schlussendlich einen neuen Wert zu berechnen. Bei einem Interrupt soll sich ein `Player`-Thread beenden.

Die öffentliche Klasse `RockPaperScissors` implementiert ebenfalls das Interface `Runnable`. Die `run`-Methode erstellt zwei Threads die jeweils ein `Player`-Object beinhalten. Die beiden Threads sollen 1000 mal Schere-Stein-Papier gegeneinander spielen. Danach sollen die Threads via einem Interrupt beendet werden. Sobald diese beendet wurden (siehe `join`) soll ausgegeben werden wie oft Thread 1 bzw. 2 gewonnen/verloren hat.

Um Zufallszahlen zu generieren, dürfen Sie die Klasse `java.util.Random` verwenden und alle Methoden daraus.

Hinweis: Das Schlüsselwort `synchronized` gehört nicht zur Methodensignatur.

Lösungsvorschlag 12.4

Korrekturbemerkung:

- `Player.run()` generiert ganze Zahlen aus $[0, 2]$ und speichert diese: 1 Punkt

- `Player.run()` wartet bis die generierte Zahl jeweils abgerufen wurde, bevor eine neue Zahl generiert wird: 2 Punkte
- `Player.run()` notifiziert andere(n) Threads, dass eine Zahl generiert wurde: 1 Punkt
- `Player.run()` terminiert, wenn ein Interrupt auftritt: 1 Punkt
- `Player.getChoice()` wartet bis eine neue Zahl generiert wurde: 2 Punkte
- `Player.getChoice()` notifiziert den Player-Thread, dass eine Zahl gelesen wurde: 1 Punkt
- `RockPaperScissors.run()`:
  - Erstellt 2 Player- und Thread-Objekte und startet letztere: 1 Punkt
  - Lässt die Threads gegeneinander 1000 mal spielen: 1 Punkt
  - Beendet die Threads via einem Interrupt: 1 Punkt
  - Wartet mittels `join`, dass beide Threads beendet wurden: 1 Punkt

Erlaubte Java-Methoden

MiniJava:

```
public static int readInt()
public static int readInt(String s)
public static int read()
public static int read(String s)
public static String readString()
public static String readString(String s)
public static void write(String output)
public static void write(int output)
public static int drawCard()
public static int dice()
```

returns an integer from the interval $[2, 11]$

returns an integer from the interval $[1, 6]$

Object bzw. allen anderen Klassen/Interfaces usw.:

```
public boolean equals(Object obj)
public final Class getClass()
public final void notify()
public final void notifyAll()
public final void wait() throws InterruptedException
```

String:

```
public char charAt(int index)
public boolean isEmpty()
public int length()
```

System:

`System.out.print(x)` prints the object `x` to the console

`System.out.println(x)` prints the object `x` to the console and terminates the line

Thread:

```
public void interrupt()
public final void join() throws InterruptedException
public void start()
public void run()
```

Selbstgeschriebene Methoden, die selber nur erlaubte Methoden verwenden, sind erlaubt, sofern dies nicht explizit in der Aufgabenstellung verboten wurde. Methoden, die im gegebenen Programmgerüst definiert wurden, dürfen verwendet werden.