Praktikum Grundlagen der Programmierung

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

Übungsblatt 13 Abgabefrist: keine Abgabe erforderlich

Σ Punkte: 0 | Σ Bonuspunkte: 0

Aufgabe 13.1 (P) Synchronisiertes Lesen und Schreiben

Betrachten Sie folgendes Programm:

```
public class TimesTwo implements Runnable {

    private final SyncNumber n;

    public TimesTwo(SyncNumber c) {
        n = c;
    }

    @Override
    public void run() {
        int i = n.read();
        n.write(i * 2);
        System.out.println("write " + i * 2);
    }

    public static void main(String[] args) {
        SyncNumber n = new SyncNumber();
        TimesTwo d1 = new TimesTwo(n);
        TimesTwo d2 = new TimesTwo(n);
        new Thread(d1).start(); // Thread 1
        new Thread(d2).start(); // Thread 2
    }

}
```

Die Klasse `SyncNumber` ist dabei gegeben durch:

```
public class SyncNumber {

    private int c = 1;

    public synchronized int read() {
        return c;
    }

    public synchronized void write(int c) {
        this.c = c;
    }
}
```

Geben Sie zwei (verschiedene) mögliche Konsolenausgaben des Programms an und füllen Sie für jeden dieser Programmdurchläufe die Tabelle unten aus. Jede Zeile steht für einen Zeitschritt und darf daher auch nur genau einen Eintrag enthalten. Als Einträge sind *ausschließlich* folgende erlaubt:

- `d1.run:Enter` / `d1.run:Return` analog für `d2`
- `n.read:Enter` / `n.read:Return`
- `n.write:Enter` / `n.write:Return`
- `n.lock:Enter` / `n.lock:Return`
- `n.unlock()`
- `println(txt)`

Dabei bedeuten die Einträge folgendes:

- `x.foo:Enter` gibt an, dass die Methode `foo` auf dem Objekt `x` aufgerufen wurde. Das Pendant `x.foo:Return` gibt an, dass der Aufruf zum Aufrufer zurückkehrt.
- `x.lock:Enter` versucht das Lock auf dem Objekt `x` zu akquirieren. Das Pendant `x.lock:Return` gibt an, dass das Lock akquiriert wurde.
- `x.unlock()` gibt das Lock auf dem Objekt `x` frei (dies ist eine Kurzschreibweise für die zwei konsekutiven Kommandos `x.unlock:Enter; x.unlock:Return`).
- `println(txt)` gibt an, dass der String `txt` via einem `System.out.println()`-Aufruf auf der Konsole ausgegeben wird. D.h. `txt` enthält keine Variablen oder noch auszuwertende Ausdrücke.

Wird eine `synchronized`-Methode mit dem Namen `foo` von dem Objekt `x` aufgerufen, so wird erst `x.foo:Enter` ausgeführt und dann das jeweilige Lock `x.lock:Enter` akquiriert. Den Aufruf `System.out.println(String txt)` können Sie durch `println()` darstellen.

Programmdurchlauf Variante 1:

Thread 1	Thread 2
`d1.run:Enter`	
	`d2.run:Enter`

Ausgabe auf der Konsole:

_____

_____

Programmdurchlauf Variante 2:

Thread 1	Thread 2
`d1.run:Enter`	
	`d2.run:Enter`

Ausgabe auf der Konsole:

_____

_____

Aufgabe 13.2 (P) Iterator symmetrischer Stack

Ergänzen Sie die Klasse `SymmetricStack` von Aufgabenblatt 8 um einen Iterator. Dazu soll die Klasse das Interface `Iterable<Integer>` implementieren, d.h. die Klasse muss um eine Methode `public Iterator<Integer> iterator()` ergänzt werden. Verwenden Sie für diese Methode das folgende Programmgerüst:

```
@Override
   public Iterator<Integer> iterator() {
       return new Iterator<Integer>() {
           //TODO
       };
   }
```

Der Iterator soll also in einer anonymen Klasse implementiert werden, die das Interface `Iterator<Integer>` implementiert. Sie brauchen im Iterator nur die beiden Methoden `hasNext()` und `next()`, nicht jedoch die optionale Methode `remove()` implementieren. Achten Sie darauf, dass die `next`-Methode eine `NoSuchElementException` wirft, wenn es keine weiteren Elemente gibt, vgl. `https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html#next--`.

Aufgabe 13.3 (P) ConcurrentModificationException

In der Vorlesung wurden Iteratoren vorgestellt. Die Java Standardbibliothek stellt für die Implementierung von Iteratoren zwei Interfaces zur Verfügung: Das Interface `Iterator<E>`, siehe `https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html` für den Iterator und das Interface `Iterable<T>`, siehe `https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html` für Datenstrukturen, die einen Iterator zur Verfügung stellen. Für diese Datenstrukturen kann die Kurzschreibweise für **for**-Schleifen verwendet werden. Beispiel: Alle Collections der Java Standardbibliothek implementieren das Interface `Iterable<T>`, stellen also einen Iterator zur Verfügung. Dieser wird bei der folgenden Kurzschreibweise verwendet:

```
LinkedList<Integer> list = new LinkedList<>();
// add some elements to list
for(int tmp : list)
     System.out.println(tmp + " ");
```

Alternativ kann man auch direkt mit dem Iterator über die Liste iterieren. Dafür stehen die Methoden `hasNext()` und `next()` zur Verfügung.

```
LinkedList<Integer> list = new LinkedList<>();
// add some elements to list
for(Iterator<Integer> it = list.iterator(); it.hasNext(); )
     System.out.println(it.next() + " ");
```

Möchte man nicht nur über eine Liste iterieren sondern diese auch gleichzeitig modifizieren, darf man das nur über Methoden, die vom Iterator selbst zur Verfügung gestellt werden. Der `listIterator` der Klasse `List` implementiert das Interface `ListIterator<E>` und stellt deshalb einige Funktionen zur Modifizierung einer Liste (`add(E e)`, `remove()`) zur Verfügung, siehe `https://docs.oracle.com/javase/8/docs/api/java/util/ListIterator.html`. Verändert man jedoch eine Liste anderweitig, während man mit einem Iterator über diese iteriert, wird eine `ConcurrentModificationException` geworfen, siehe `https://docs.oracle.com/javase/8/docs/api/java/util/ConcurrentModificationException.html`. Probieren Sie das mit dem folgenden Code aus:

```
LinkedList<Integer> list = new LinkedList<>();
for(int i = 0; i < 20; i++)
     list.add(i);
for (int tmp : list) {
     System.out.print(tmp + " ");
     list.remove(tmp); //ConcurrentModificationException
}
```

Prinzipiell sollte man sich deshalb merken, dass man eine Liste nicht anderweitig modifiziert während man mit einem Iterator darüber iteriert! Für Modifikationen kann man auf die vom Iterator zur Verfügung gestellten Methoden zurückgreifen. Insbesonder sollte man sich auch nicht auf eine `ConcurrentModificationException` verlassen, da es bestimmte Konstellationen gibt, in denen keine `ConcurrentModificationException` geworfen wird und der Iterator dann nicht vorhersehbares verhalten zeigt. Probieren Sie dazu den folgenden Code aus:

```
LinkedList<Integer> list = new LinkedList<>();
for(int i = 0; i < 20; i++)
   list.add(i);
System.out.println(list);
int i = 0;
for(int tmp : list){
   System.out.print(tmp + " ");
   i++;
   if(i > 18)
       list.remove(0);
}
System.out.println("\n" + list);
```

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.