Übungsblatt 11

3 Aufgaben, 20 Punkte

Objektorientierte Modellierung und Programmierung (inf031) Sommersemester 2020
Carl von Ossietzky Universität Oldenburg, Fakultät II, Department für Informatik

Dr. C. Schönberg

Ausgabe: 2020-06-30 14:00 **Abgabe:** 2020-07-07 12:00

Aufgabe 1: *Threads: Synchronisation* **(5 Punkte)**

Schauen Sie sich das folgende Programm an:

```
1  class Output extends Thread {
2
3    public void run() {
4      synchronized (InOut.getLock()) {
5        if (!InOut.isEntered()) {
6          try {
7            InOut.getLock().wait();
8          } catch (InterruptedException e) {
9          }
10       }
11       System.out.println(InOut.getValue() * InOut.getValue());
12     }
13   }
14 }
15
16 class Input extends Thread {
17
18   public void run() {
19     synchronized (InOut.getLock()) {
20       InOut.setValue(IO.readInt("Value: "));
21       InOut.setEntered(true);
22       InOut.getLock().notify();
23     }
24   }
25 }
26
27 public class InOut {
28
29   private static Object lock = new Object();
30   private static boolean entered = false;
31   private static int value = 0;
32
33   public static Object getLock() { return lock; }
34
35   public static boolean isEntered() { return entered; }
36
37   public static void setEntered(boolean entered) { InOut.entered = entered; }
```

```
38
39    public static int getValue() { return value; }
40
41    public static void setValue(int value) { InOut.value = value; }
42
43    public static void main(String[] args) {
44      new Output().start();
45      new Input().start();
46    }
47
48  }
```

Überlegen Sie für sich: Was tut das Programm? Zur Synchronisation der beiden Threads wird hier das **`synchronized`**-Konstrukt zusammen mit `wait` und `notify` eingesetzt.

Wandeln Sie dann die drei gegebenen Klassen in drei jeweils äquivalente Klassen um, in denen aber kein **`synchronized`**, kein `wait` und kein `notify` mehr vorkommen, sondern zur Synchronisation der beiden Threads das Konzept der *Semaphore* auf der Grundlage der Klasse `java.util.concurrent.Semaphore` verwendet wird.

Aufgabe 2: *Threads: Barrieren* **(5 Punkte)**

In folgendem Programm werden n Threads gestartet, die unabhängig voneinander die Zahlen 1 bis 1000 ausgeben.

```
1  class NumberRunner extends Thread {
2
3    private int number;
4
5    public NumberRunner(int n) {
6      number = n;
7    }
8
9    @Override
10   public void run() {
11     for (int i = 0; i < 1000; i++) {
12       System.out.println("Thread " + number + ": " + i);
13     }
14   }
15 }
16
17 public class Barriers {
18
19   private final static int NUMBER = 3;
20
21   public static void main(String[] args) {
22     NumberRunner[] runner = new NumberRunner[NUMBER];
23     for (int i = 0; i < NUMBER; i++) {
24       runner[i] = new NumberRunner(i);
25     }
26     for (int i = 0; i < NUMBER; i++) {
27       runner[i].start();
28     }
29   }
30
31 }
```

Bauen Sie in das Programm Thread-Mechanismen derart ein, dass die Threads nach jeweils 10 Ausgaben auf die anderen warten (*Barrier*), bevor sie dann weitermachen.

Aufgabe 3: *Rucksack-Problem* **(3 + 3 + 4 Punkte)**

Das Rucksack-Problem (engl. *knapsack problem*) ist ein Optimierungsproblem, vom dem bekannt ist, dass es NP-vollständig ist. Es gibt verschiedene Varianten des Problems, von denen wir hier folgende betrachten wollen:

- Gegeben sei ein *Rucksack* mit einer festen KAPAZITÄT (hier: maximales Füllgewicht, d.h. die Gegenstände im Inneren dürfen insgesamt nicht mehr wiegen als die Kapazität erlaubt).
- Zusätzlich gibt es eine Reihe von *Gegenständen*, die jeweils ein festes GEWICHT und einen festen WERT haben.
- Ziel ist es nun, eine optimale Füllung des Rucksacks zu erreichen: Die Füllung soll den höchstmöglichen Gesamtwert haben, ohne die Gewichtskapazität zu überschreiten. Jeder Gegenstand darf mehrfach in den Rucksack gepackt werden (es gibt allerdings auch die Variante, dass jeder Gegenstand nur einmal eingepackt werden kann – diesen Spezialfall betrachten wir hier nicht!).

Hintergrund: Den Panzerknackern ist es gelungen, in Dagobert Ducks Geldspeicher einzudringen. Leider haben sie nur einen einzigen Rucksack dabei, in dem sie Beute fortschaffen können. Damit dieser Rucksack nicht reißt, darf er aber auch nicht zu schwer gepackt werden. Deshalb müssen die Panzerknacker entscheiden, wie viele Goldbarren, Diamanten und Geldscheine sie hineinstopfen können, um den maximalen Wert zu erhalten. Helfen Sie den Panzerknackern, dieses Problem zu lösen.

Aufgabe: Gegeben sind eine abstrakte Klasse `Knapsack`, die den Rucksack repräsentiert; eine Klasse `Item`, die einen Gegenstand repräsentiert; sowie eine Klasse `Selection`, welche die Auswahl von Gegenständen für den Rucksack repräsentiert.

Die Klasse `Knapsack` verwaltet die Kapazität des Rucksacks (`capacity`), sowie die zur Verfügung stehenden Gegenstände (`candidates`). Die abstrakte Methode **`public abstract`** `Selection pack()` soll die beste Auswahl von Gegenständen bezüglich der gegebenen Kapazität und der Kandidaten ermitteln. Die `main`-Methode testet verschiedene konkrete Implementierungen des Rucksacks.

Die Klasse `Item` verwaltet den Namen (`name`), den Wert (`value`) und das Gewicht (`weight`) eines Gegenstands. Sie implementiert außerdem die Methode `toString`, um eine einfache Ausgabemöglichkeit zur Verfügung zu stellen, sowie die Methoden `equals` und `hashCode`, damit Gegenstände verglichen und beispielsweise in Hashtabellen (`HashMap`) eingefügt werden können.

Die Klasse `Selection` verwaltet für eine Menge von Gegenständen ihre jeweilige Anzahl (`items`), sowie das Gesamtgewicht und den Gesamtwert aller Gegenstände. Sie stellt eine `add()`-Methode zur Verfügung, die einen Gegenstand hinzufügt. Sie implementiert ebenfalls die Methoden `toString`, `equals` und `hashCode`.

a) Schreiben Sie eine Klasse `KnapsackRecursive`, die von `Knapsack` erbt, und implementieren Sie dort die `pack`-Methode. Implementieren Sie die Methode *rekursiv*, also indem Sie ganz naïv alle Möglichkeiten durchprobieren. Achten Sie auf passende Abbruchbedingungen!

b) Schreiben Sie eine Klasse `KnapsackGreedy`, die von `Knapsack` erbt, und implementieren Sie dort die `pack`-Methode. Implementieren Sie die Methode nach dem *greedy*-Verfahren, also indem Sie immer die aktuell am besten erscheinende Auswahl treffen und diese Auswahl später nicht wieder revidieren.

c) Schreiben Sie eine Klasse `KnapsackDynamic`, die von `Knapsack` erbt, und implementieren Sie dort die `pack`-Methode. Implementieren Sie die Methode mittels *dynamischer Programmierung* und *Memoisation*, also indem Sie Zwischenergebnisse zwischenspeichern und gegebenenfalls wiederverwenden.
Hinweis: Überlegen Sie sich, mit welcher bekannten Datenstruktur Sie am besten Zwischenergebnisse speichern können.