TU München, Fakultät für Informatik
Lehrstuhl III: Datenbanksysteme
Prof. Alfons Kemper, Ph.D.

Übung zur Vorlesung *Einführung in die Informatik 2 für Ingenieure (MSE)*

Alexander van Renen (renen@in.tum.de)
http://db.in.tum.de/teaching/ss17/ei2/

Lösungen zu Blatt 4

Aufgabe 1: Stack mit verketteter Liste

Implementieren Sie einen Stack für `int`-Werte mithilfe einer verketteten Liste. Der Stack sollte dabei die Methoden `push()` und `pop()` anbieten, mit denen Elemente auf den Stack gelegt bzw. von ihm heruntergenommen werden. Eine mögliche Modellierung ist in Abbildung 1 gezeigt.

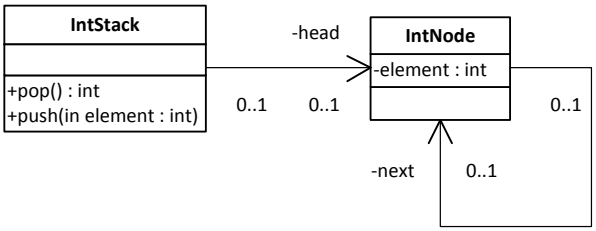

Abbildung 1: Stack mit verketteter Liste

Lösung 1

Der folgende Quelltext setzt den Stack entsprechend der UML-Modellierung in Java um.

```
1  import java.lang.RuntimeException;
2
3  // Ausnahme, falls auf einem leeren Stack pop()
4  // aufgerufen wird
5  class EmptyStackException extends RuntimeException {}
6
7  // Die Knoten der verketteten Liste
8  class Node {
9    // Der in diesem Knoten gespeicherte Wert
10   int value;
11   // Der naechste Knoten
12   Node next;
13 }
14
15 // Der Stack
16 public class Stack {
17   // Referenz auf das oberste Element im Stack,
18   // welches wiederum auf die darunter liegenden
19   // Elemente verweist
20   Node head;
```

```
21
22    // Das oberste Element herunternehmen
23    public int pop() {
24      // Stack leer -> Ausnahme werfen
25      if (head == null) {
26        throw new EmptyStackException();
27      }
28      // Oberstes Element aktualisieren und dessen
29      // Wert zurueckgeben
30      Node oldHead = head;
31      head = head.next;
32      return oldHead.value;
33    }
34
35    // Element oben auf den Stack legen
36    public void push(int element) {
37      Node newElement = new Node();
38      newElement.value = element;
39      newElement.next = head;
40      head = newElement;
41    }
42  }
```

Aufgabe 2: Türme von Hanoi

In dieser Aufgabe implementieren wir das bekannte Knobelspiel „Die Türme von Hanoi“.[1]. Dabei verwenden wir den Stack aus Aufgabe 3 um die drei Türme zu modellieren.

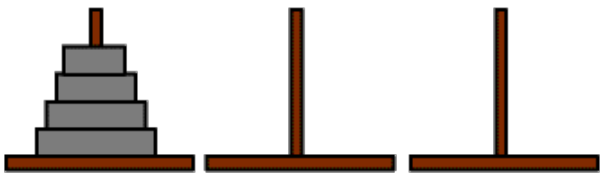

Abbildung 2: Die Türme von Hanoi mit vier Scheiben

Überlegen Sie sich einen rekursiven Algorithmus um alle Scheiben von Position 1 auf Position 3 zu verschieben. Dabei darf immer nur die oberste Scheibe eines Turms auf einen anderen Turm gelegt werden, wenn dessen oberste Scheibe größer ist (oder an der Zielposition gar keine Scheiben sind).

Tipp: Damit die unterste Scheibe zur Position 3 bewegt werden kann, muss der darüber liegende Turm erstmal auf Position 2 verschoben werden. Dann kann man die größte Scheibe verschieben und den kleineren Turm von Position 2 auf die große Scheibe verschieben. Überlegen Sie sich wie dieses Prinzip auch für das Verschieben der kleineren Türme zum Tragen kommt.

[1] Für eine ausführliche Beschreibung siehe Wikipedia: `http://de.wikipedia.org/wiki/Türme_von_Hanoi`. Das ganze kann auch auf unzähligen Seiten ausprobiert werden, z.B.: `http://vornlocher.de/tower.html`

Lösung 2

Die folgende Methode `move()` löst das Problem rekursiv. Sie verschiebt die als Parameter angegebene Scheibe mitsamt den darauf liegenden kleineren Scheiben vom spezifizierten Startfeld auf das gewünschte Zielfeld. Dabei geht sie wie folgt vor: (1) Verschiebe den Turm ohne die unterste Scheibe vom Startfeld auf das Zwischenfeld, (2) bewege die nun freigewordene unterste Scheibe aufs Zielfeld und (3) ziehe jetzt den kleineren Turm vom Zwischenfeld nach. Die beiden schwierigen Schritte (1) und (3) führt die Methode durch, indem sie sich rekursiv mit passenden Parametern aufruft. Machen Sie sich klar, dass dies nur geht, da immer ein vollständiger Turm verschoben wird und das Zwischen- und Zielfeld dadurch nur größere Scheiben enthält, auf die die (kleineren) Scheiben dieses Turms gelegt werden dürfen.

```
1  public class Hanoi {
2    // Anzahl der Scheiben
3    private static final int SCHEIBEN = 5;
4    // Die drei Felder
5    static Stack feld1, feld2, feld3;
6
7    // Ausfuehrbare main-Methode
8    public static void main(String[] args) {
9      // Lege die drei Felder an
10     feld1 = new Stack();
11     feld2 = new Stack();
12     feld3 = new Stack();
13
14     // Lege die Scheiben auf Feld 1
15     for (int scheibe = SCHEIBEN; scheibe > 0; scheibe--) {
16       feld1.push(scheibe);
17     }
18
19     // Verschiebe den Turm von Feld 1 ueber Feld 2 nach Feld 3
20     move(SCHEIBEN, feld1, feld2, feld3);
21   }
22
23   // Verschiebe den Turm mit allen Scheiben kleiner als "scheibe" von
24   // "start" ueber "ablage" nach "ziel"
25   public static void move(int scheibe, Stack start, Stack ablage,
       Stack ziel) {
26     if (scheibe > 0) {
27       // Verschiebe zuerst den auf der Scheibe liegenden Turm auf die
           Ablage
28       move(scheibe-1, start, ziel, ablage);
29
30       // Verschiebe die unterste Scheibe des Turms auf das Zielfeld
31       ziel.push(start.pop());
32
33       // Verschiebe den kleineren Turm von der Ablage auf das
           Zielfeld
34       move(scheibe-1, ablage, start, ziel);
```

```
    }
  }
}
```

Aufgabe 3: Werte vs. Objekte

In dieser Aufgabe schauen wir uns noch einmal den Unterschied zwischen Werten und Objekten an. Überlegen Sie sich, was die Ausgabe sein sollte und warum. Überprüfen Sie Ihre Antwort indem Sie das Programm ausführen.

```
class PersonA {
  int alter;
  public PersonA(int alter) {
    this.alter = alter;
  }
}

class PersonB {
  Alter alter;
  public PersonB(Alter alter) {
    this.alter = alter;
  }
}

class Alter {
  public int jahre;
  public Alter(int jahre) {
    this.jahre = jahre;
  }
}

class WerteVsObjekte {
  public static void main(String[] args) {
    PersonA mickeyA = new PersonA(50);
    PersonA donaldA = new PersonA(55);

    donaldA.alter = mickeyA.alter;
    mickeyA.alter = 51;
    System.out.println("MickeyA ist " + mickeyA.alter);
    System.out.println("DonaldA ist " + donaldA.alter);

    System.out.println("====");

    PersonB mickeyB = new PersonB(new Alter(50));
    PersonB donaldB = new PersonB(new Alter(55));

    donaldB.alter = mickeyB.alter;
    mickeyB.alter.jahre = 51;
```

```
39        System.out.println("MickeyB ist " + mickeyB.alter.jahre);
40        System.out.println("DonaldB ist " + donaldB.alter.jahre);
41    }
42 }
```

Lösung 3

Die Variante mit Werten verhält sich so, wie man es erwartet: In Zeile 27 wird das Alter von MickeyA kopiert und DonaldA ist somit ebenfalls 50. Anschließend wird das Alter von MickeyA auf 51 gesetzt, ohne dass sich das Alter von DonaldA ändert.

Anders ist es, wenn das Alter als Objekt umgesetzt wird. Nun führt die syntaktisch äquivalente Zuweisung in Zeile 27 dazu, dass DonaldB auch das Alter-Objekt von MickeyB referenziert, das zu diesem Zeitpunkt den Wert 50 hat. Anschließend wird aber genau diese Objekt modifiziert und damit ändert sich das Alter von beiden auf 51.

Diese Aufgabe verdeutlicht einen wichtigen Unterschied zwischen Werten und Objekten: Während bei der Zuweisung von Objekten nur die Referenz kopiert wird, werden Werte vollständig kopiert. Bei der Umsetzung mit einem Objekt ist dieses anschließend mehrfach referenziert und kann über zwei Zugriffspfade verändert werden. Diese Änderung ist entsprechend auch über beide Zugriffspfade sichtbar.

Aufgabe 4: Operationen

siehe unten

Lösung 4

(a) Welche drei Gruppen von Operationen haben wir eingeführt und welche Eigenschaften machen diese Gruppen aus?

1. Konstruktoren (erzeugen Objekte)
2. Beobachter (geben den Zustand eines Objektes zurück ohne das Objekt in irgendeiner Art und Weise zu verändern)
3. Mutatoren (verändern Objekte)

(b) Warum sollte eine Beobachterfunktion das Objekt nicht verändern?

Dies wäre für den Klienten der Klasse unerwartet, wenn eine Beobachterfunktion das Objekt verändert. Eine Beobachter-Methode sollte frei von Seiteneffekten sein, so ist sie definiert.

(c) Welcher Rückgabetyp macht bei einer Beobachterfunktion keinen Sinn?

Der Rückgabetyp `void` macht keinen Sinn, da ein Beobachter immer auch „etwas“ beobachten sollte.

(d) Welche Access Modifier gibt es und was bedeuten sie?

- `private`: Sichtbarkeit nur innerhalb der Klasse selbst
- `protected`: auch in Unterklasen sichtbar
- `package`: alle Klassen im gleichen Paket (siehe unten)
- `public`: im gesamten Programm sichtbar

Bei einem *Paket* (engl. package) handelt es sich um einen weiteren Namensraum für die darin enthaltenen Klassen. Zusammengehörige Klassen, die zusammen eine bestimmte Funktionalität erfüllen, werden üblicherweise in einem Paket zusammengefasst. Diese können dann untereinander auf Variablen und Methoden zugreifen, die mit dem Access Modifier `package` versehen wurden. Klassen außerhalb des Pakets haben darauf keine Zugriffsmöglichkeit. Dies verringert die Kopplung von nicht zusammengehörigen Klassen und erhöht dadurch die Änderbarkeit.