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

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

Christoph Anneser (anneser@in.tum.de)
http://db.in.tum.de/teaching/ss21/ei2/

Lösungen zu Blatt 4

Aufgabe 1: Einfach verkettete Liste ⌨

Implementieren Sie eine generische, einfach verkettete Liste. Nutzen Sie hierfür die folgende Vorlage, die Sie auf gitlab.db.in.tum.de finden. Klicken Sie hierfür auf das Download-Symbol und laden Sie sich das Projekt als Zip-Datei herunter. Entpacken Sie die Datei und öffnen Sie sie in Ihrer IDE. Machen Sie sich mit den Dateien und der Struktur des Projekts vertraut:

```
LinkedList
└── src
    ├── main
    │   └── java
    │       ├── LinkedList.java // todo: Ihre Implementierung
    │       └── List.java // Interface
    └── test
        └── java
            └── LinkedListTest.java // die Tests
└── pom.xml // Maven Konfigurationsdatei, mehr Infos maven.apache.org
```

Weitere Informationen erhalten Sie in der Zentralübung (siehe Aufzeichnung in Moodle).

Implementieren Sie nun die mit todo gekennzeichneten Methoden in der Klasse `LinkedList.java`.

Tipp 1: Erarbeiten Sie sich vor der Implementierung der Methoden ein Grundverständnis, wie die Liste funktionieren soll. Erstellen Sie z.B. Skizzen von Listen, wie Sie nach bestimmten Operationen im Speicher aussehen könnte (vgl. Abbildung 1).

Tipp 2: Benutzen Sie für Vergleiche stets die `equals`-Methode. Eine generische Liste kann z.B. auch Punkt-Objekte speichern.

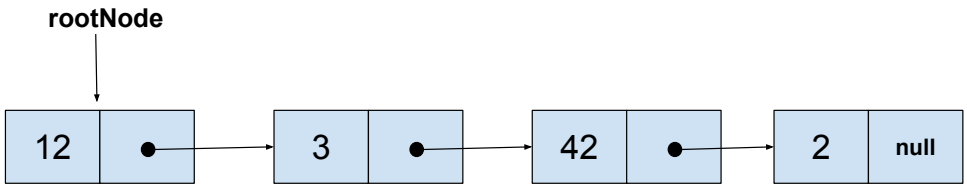

Abbildung 1: Interner Zustand einer Integer speichernden Liste, nachdem `list.add(12)`, `list.add(3)`, `list.add(42)`, `list.add(2)` aufgerufen wurden.

Lösung 1

Unter gitlab.db.in.tum.de finden Sie die Lösung.

Aufgabe 2: Don't repeat yourself (DRY)

Auf der Website[1] finden Sie die Klasse `Warenhaus.java`.

Lösung 2

Redundant vorhandene Informationen (z. B. Code-Duplizierungen in Quellcode) sind schwierig zu pflegen, da die Konsistenz zwischen den einzelnen Duplikaten von Hand gewährleistet werden muss. Vergisst man bei Änderungen eine Stelle, kommt es zu Inkonsistenz und als Folge oft zu nur schwer nachvollziehbarem Fehlverhalten des Programms. In dem Programm aus diese Aufgabe ist dies die Preisberechnung mit der Mehrwertsteuer, die an vielen Stellen redundant vorkommt. Bei Systemen, die dem DRY-Prinzip treu bleiben, brauchen Änderungen hingegen nur an einer Stelle vorgenommen zu werden und es können dabei keine Inkonsistenzen entstehen.

a) Nennen Sie mindestens zwei Gründe, warum die Modellierung nicht optimal ist – beispielsweise, wenn sich der Mehrwertsteuersatz mal wieder ändern sollte.

   - Die Mehrwertsteuer ist in den Klassen mehrfach fest eingetragen. Diese Redundanz kann zu Inkonsistenzen führen, da bei einer Erhöhung des Steuersatzes eine Stelle übersehen werden könnte. Der Steuersatz sollte stattdessen an einer einzigen Stelle definiert sein und von dort jeweils referenziert werden.
   - Alle fünf Produkte haben eine unterschiedlich benannte Methode um ihren Preis festzustellen. Dies führt zu Mehraufwand bei der Verwendung der Klassen, da ein Klient erst herausfinden muss, wie die jeweilige Methode heißt.
   - Die fünf Produktklassen haben keine gemeinsame Oberklasse, obwohl sie ähnliches Verhalten aufweisen. Eine gemeinsame Oberklasse würde es erlauben, Objekte der verschiedenen Klassen gleich zu behandeln – beispielsweise könnten diese dann in Kollektionen zusammengefasst werden.

b) Implementieren Sie den Aufzählungstyp `Mehrwertsteuersatz`, der zwei Werte für den normalen und den vergünstigten Mehrwertsteuersatz hat.

   Auf der Website finden Sie die Lösung `WarenhausLoesung.java`.

c) Ziehen Sie die Gemeinsamkeiten der Produkte in eine gemeinsame Oberklasse `Produkt`. Führen Sie eine Methode ein, die für ein Produkt angibt, ob die vergünstigte Mehrwertsteuer anwendbar ist. Verwenden Sie diese Methode bei der Preisberechnung.

   Auf der Website finden Sie die Lösung `WarenhausLoesung.java`.

d) Führen Sie die Klasse `Einkaufskorb` ein, die eine Menge von Produkten verwaltet und deren Gesamtpreis bestimmen kann. Verwenden Sie diese Klasse in der main-Methode des Warenhauses.

   Auf der Website finden Sie die Lösung `WarenhausLoesung.java`.

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.

[1] Hier: `http://www3.in.tum.de/teaching/ss13/ei2/`

```
1  class PersonA {
2    int alter;
3    public PersonA(int alter) {
4      this.alter = alter;
5    }
6  }
7
8  class PersonB {
9    Alter alter;
10   public PersonB(Alter alter) {
11     this.alter = alter;
12   }
13 }
14
15 class Alter {
16   public int jahre;
17   public Alter(int jahre) {
18     this.jahre = jahre;
19   }
20 }
21
22 class WerteVsObjekte {
23   public static void main(String[] args) {
24     PersonA mickeyA = new PersonA(50);
25     PersonA donaldA = new PersonA(55);
26
27     donaldA.alter = mickeyA.alter;
28     mickeyA.alter = 51;
29     System.out.println("MickeyA ist " + mickeyA.alter);
30     System.out.println("DonaldA ist " + donaldA.alter);
31
32     System.out.println("===");
33
34     PersonB mickeyB = new PersonB(new Alter(50));
35     PersonB donaldB = new PersonB(new Alter(55));
36
37     donaldB.alter = mickeyB.alter;
38     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.

Zusatzaufgabe 5: 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 2 gezeigt. Wie könnte man diesen Stack generischer machen?

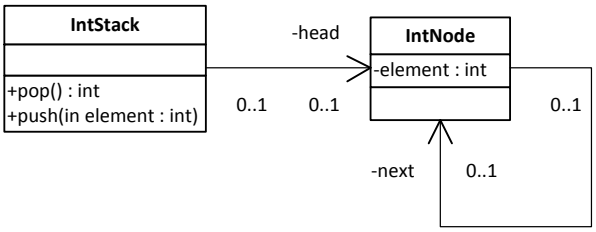

Abbildung 2: Stack mit verketteter Liste

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  }
```