TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

**Lehrstuhl für Angewandte Informatik / Kooperative Systeme**
**Praktikum Grundlagen der Programmierung**
Prof. Dr. A. Brüggemann-Klein, R. Palenta, J. Kranz, Dr. A. Reuss

WS 15/16
**Übungsblatt 7**
27.11.2015

Abgabe: 07.12.2015 (vor 5:00 Uhr)

**Hinweis:** Implementieren Sie alle Hausaufgaben in einem Package namens **blatt07**.

**Hinweis zu den Hausaufgaben**: Ab sofort wird vorab eine Aufteilung der erreichbaren Punkte auf Teilaufgaben bereitgestellt. Beachten Sie bitte, dass die erreichbaren Punkte sich nicht rein auf die zu implementierende Funktionalität beziehen. Die Funktionalität muss leicht nachvollziehbar umgesetzt werden. Insbesondere soll das Programm übersichtlich strukturiert und sinnvoll kommentiert sein. Hierbei geht es nicht nur um die Bewertung der Abgabe. Die genannten Aspekte im Auge zu behalten, kann das Programmieren selbst auch wesentlich erleichtern.

## Aufgabe 7.1 (P) Bibliothek

Themengebiet: OO-Modellierung, Collections (LinkedList)

Schreiben Sie eine Klasse `Buch`, die die Attribute `String titel, autor, isbn` und `double preis` hat. Implementieren Sie

- zwei Konstruktoren `Buch(String titel, String autor)` und `Buch(String titel, String autor, String isbn, double preis)`,
- entsprechende get- und set-Methoden für die vier Attribute (`void setAutor(String autor)`, `String getTitel()`, usw.),
- eine Methode `String toString()`, die den Titel, den Autor, die ISBN-Nummer und den Preis jeweils in einer eigenen Zeile als String zurückgibt,
- eine Methode `boolean hasSameAutor(Buch a)`, die den Autor des übergebenen Buch-Objekts mit dem der aktuellen Instanz vergleicht und `true` zurückgibt, falls diese übereinstimmen, und `false` sonst.

Schreiben Sie eine Klasse `Bibliothek`, die als privates Attribut eine Bibliothek als `LinkedList<Buch>` verwaltet. Implementieren Sie

- einen Konstruktor `Bibliothek()`, der eine leere Bibliothek anlegt,
- eine Methode `public void add(Buch buch)`, das ein Buch zur Liste hinzufügt,
- eine Methode `public void delete(Buch buch)`, die ein Buch, falls vorhanden, aus der Liste entfernt,
- eine Methode `public LinkedList<Buch> getBuecherMitAutor(String autor)`, die eine Liste aller Bücher des im Parameter übergebenen Autors zurückgibt, die in der Bibliothek vorhanden sind,
- eine Methode `public Buch getBuchMitTitel(String titel)`, die das erste in der Bibliothek gefundene Buch mit entsprechendem Titel zurückgibt, und `null`, falls keines mit diesem Titel vorhanden ist.

Testen Sie die Klassen `Buch` und `Bibliothek`.

**Aufgabe 7.2 (P) Bubblesort**

Themengebiet: Aufbau/Zerlegung, Collections (LinkedList)

Ein elementares Sortierverfahren läuft wie folgt ab:

> Zu Beginn befinden sich alle Elemente unsortiert in einer Liste. Diese wird immer wieder durchlaufen und dabei jedesmal benachbarte Elemente vertauscht, wenn sie nicht gemäß der Ordnung, nach welcher sortiert werden soll, geordnet sind. Sobald bei einem Durchlauf kein Tausch mehr stattfindet, ist die Liste sortiert.

Implementieren Sie den beschriebenen Algorithmus in einer Methode `public static void bubbleSort(LinkedList<Integer> numbers)`. Testen Sie Ihr Programm mit zufälligen Zahlen. Zusatzfrage: Wieso kann die Liste überhaupt sortiert werden, obwohl sie mittels *Call by value* übergeben wird und somit nur gelesen, aber nicht geschrieben werden kann?

Anregungen:

- Spielen Sie das Verfahren zunächst per Hand an einigen konkreten Beispielen von Listen mit ganzen Zahlen durch.
- Wenn Sie Schwierigkeiten haben, einen Einstieg zu finden, teilen Sie die Aufgabe zunächst in kleinere Teilaufgaben auf. Wie können Sie z.B. zwei einzelne Einträge vertauschen (vgl. Aufgabe 1.6)? Implementieren **und testen** Sie das Vertauschen separat (eigene Methode!).
- Diskutieren Sie, wieso es notwendig ist, die LinkedList mit dem Typ `Integer` zu instanzieren statt mit `int`. Schlägt sich dieser Umstand im restlichen Programmcode nieder? Wenn ja, inwiefern? Wenn nein, wieso nicht? Was wurde dazu in der Vorlesung gesagt? Zusatzfrage: Lassen sich Integer-Objekte z.B. mit dem Vergleichsoperator „<“ für Ganzzahlen vergleichen? Warum (nicht)?
- Machen Sie sich Gedanken, weshalb das Bubblesort-Verfahren überhaupt *terminiert*, d.h. bei jedem Aufruf nach einer (von der jeweiligen Eingabe abhängigen) festen Zahl von Rechenschritten ein Ergebnis liefert statt immer weiter zu rechnen.
- Diskutieren Sie, wie eine objektorientierte Lösung der Aufgabe aussehen könnte (mit privatem Attribut `LinkedList<Integer>` numbers).

**Aufgabe 7.3 (P) Parkhaus mit Listen**

Themengebiet: OO-Modellierung, Collections (LinkedList)

Ziel dieser Aufgabe ist es, ein Parkhaus zu simulieren. Dabei soll das Parkhaus mit Hilfe einer LinkedList von Autos umgesetzt werden. Parklücken können hierbei einfach durch `null`-Einträge dargestellt werden.

a) Implementieren Sie die Klasse `Auto` mit den privaten Attributen `marke` und `kennzeichen`, sowie den beiden Methoden `String getMarke()` und `String getKennzeichen()` und einem sinnvollen Konstruktor.

b) Erweitern Sie die Klasse `Auto` um ein privates Attribut `int fahrgestellnummer` und eine öffentliche Methode `int getFahrgestellnummer()`. Die Fahrgestellnummer soll bei jedem neu erzeugten `Auto`-Objekt um eins erhöht werden.

c) Implementieren Sie die Klasse `Parkhaus`, die Parkhäuser mit $n$ Parkplätzen repräsentiert. Dabei soll die Größe $n$ des Parkhauses im Konstruktor übergeben werden und anschließend nicht mehr verändert werden können.

d) Fügen Sie der Klasse `Parkhaus` eine Methode `int parken(Auto auto)` hinzu, bei der das Auto `auto` den ersten freien Parkplatz des Parkhauses belegt. Die Methode soll dabei die Nummer des belegten Parkplatzes zurückliefern (0 bis $n - 1$). Wenn das Parkhaus voll ist, soll `-1` zurückgegeben werden.

e) Erweitern Sie die Klasse `Parkhaus` um die Methode `int suchen(String kennzeichen)`, die mit Hilfe des Kennzeichens ein Auto im Parkhaus sucht. Ist das Auto

mit dem Kennzeichen `kennzeichen` im Parkhaus, soll die Methode die Nummer des Parkplatzes ausgeben, ansonsten `-1`.

f) Erweitern Sie die Klasse `Parkhaus` um die Methode `Auto wegfahren(String kennzeichen)`, das den vom Auto mit dem Kennzeichen `kennzeichen` belegten Parkplatz wieder freigibt und das Auto zurückgibt. Falls das Auto mit Kennzeichen `kennzeichen` nicht im Parkhaus ist, soll `null` zurückgegeben werden.

**Hinweis:** Um Ihre Klassen zu testen, sollten Sie sich eine Klasse `TestParkhaus` mit einer `main`-Methoden anlegen. Simulieren Sie dabei auch die Interaktion zwischen beiden Klassen. Sie sollten alle Methoden auch auf korrekte Fehlerbehandlung testen. Was passiert z.B., wenn das Parkhaus voll ist? Was passiert, wenn in einem leeren Parkhaus nach einem Auto gesucht wird?

**Hinweis:** Beachten Sie, dass es sich in solchen Modellen, wie sie hier umgesetzt werden, kaum vermeiden lässt, dass Simulation und Wirklichkeit sich an manchen Stellen unterscheiden. Beispielsweise kann in der Wirklichkeit ein einzelnes Auto nur einmal in ein Parkhaus fahren und nicht z.B. in zwei verschiedenen Parkhäusern jeweils zweimal geparkt werden. Auch die Problematik, dass verschiedene Autos das selbe Kennzeichen bekommen können, soll nicht in den Klassen Auto und Parkhaus behandelt werden, sondern muss von den diese Klassen verwendenden Klassen ggf. geeignet behandelt werden.

## Aufgabe 7.4 [9 Punkte] (H) Sortieren mit InsertionSort

**Themen**:

- Sortieralgorithmus *InsertionSort*
- Objektorientierte Modellierung

In dieser Aufgabe geht es darum, einen weiteren Sortieralgorithmus kennenzulernen und diesen auf einer Liste von Büchern anzuwenden.

Erstellen Sie dazu zunächst eine Klasse `Book` (insgesamt 4 Punkte). Ein Buch soll aus einem Titel und Autor bestehen, außerdem ist über Bücher noch deren Auflage bekannt. Die Klasse `Book` soll geschützten Zugriff auf die Daten eines Buches über `get`-Methoden zur Verfügung stellen. Die Klasse soll über einen geeigneten Konstruktor verfügen. Um ein Buch ausgeben zu können, soll die Methode `public String toString()` angeboten werden.

Um eine Liste von Büchern sortieren zu können, muss man Bücher miteinander vergleichen können. Ein Buch ist dabie *kleiner* als ein anderes Buch, wenn der Autor lexikographisch sortiert vor dem Autor des anderen Buches steht; ist der Autor zweier Bücher identisch, werden sie nach Titel (lexikographisch, wieder aufsteigend) geordnet. Sind sowohl Autor als auch Titel identisch, entscheidet sich die Anordnung entsprechend der Auflage (normale Zahlenordnung, aufsteigend). Um Objekte der Klasse `Book` vergleichen zu können, implementieren Sie eine Methode `int compareTo(Book other)`[1], die folgende Werte zurückgibt:

[1] vergleiche auch `https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html`; von diesem Interface soll allerdings aktuell nicht abgeleitet werden, stattdessen wird es auf den folgenden Aufgabenblättern behandelt.

| Bedingung | Rückgabewert | Beschreibung |
|---|---|---|
| Object < anderes Objekt | negative Zahl | Ist das aktuelle Objekt vor dem Vergleichsobjekt angeordnet, wird eine negative Zahl zurückgegeben. |
| Object = anderes Objekt | Null (0) | Gleicht das aktuelle Objekt dem Vergleichsobjekt, wird eine Null (0) zurückgegeben. |
| Object > anderes Objekt | positive Zahl | Ist das aktuelle Objekt nach dem Vergleichsobjekt angeordnet, wird eine positive Zahl zurückgegeben. |

Die eigentliche Sortierung soll durch das sogenannte *InsertionSort*[2]-Verfahren geschehen. Dieses Sortierverfahren gehört – wie das Ihnen schon aus den Tutorübungen bekannte BubbleSort – zu den sehr einfach verständlichen und langsamen Sortierverfahren. Die Idee besteht darin, durch die Liste zu laufen und Elemente jeweils an der richtigen Stelle in der sortierten Teilsequenz einzufügen. Gehen wir zum Beispiel davon aus, wir möchten die Sequenz `[27, 4, 5]` sortieren. Im ersten Schritt ist unsere sortierte Sequenz leer und wir fügen die 27 ein, müssen also nichts tun und erhalten die Sequenz `[27, 4, 5]` (die sortierte Teilsequenz ist rot markiert). Im zweiten Schritt müssen wir die 4 an die richtige Stelle in die Sequenz einfügen, die aktuell nur aus der 27 besteht, dies führt zum Ergebnis `[4, 27, 5]`. Wir fahren mit der 5 fort und fügen diese an richtige Stelle ein; so erhalten wir schließlich das sortierte Ergebnis `[4, 5, 27]`.

Implementieren Sie eine Methode `static void sort(LinkedList<Book> books)` (5 Punkte), die eine Liste von Büchern mittels InsertionSort aufsteigend entsprechend der oben definierten Ordnung auf Büchern sortiert. Die Methode soll dabei zum Vergleich von Büchern die oben gezeigt `compareTo`-Methode verwenden. Implementieren Sie außerdem ein Hauptprogramm und testen Sie so Ihre Implementierung der Klasse `Book` und der Sortierung.

**Hinweis:** Sie müssen in dieser Aufgabe nicht auf Fehlerbehandlung achten, insbesondere können Sie davon ausgehen, dass kein Attribut den Wert `null` hat.

## Aufgabe 7.5 [9 Punkte] (H) Waschsalon

In dieser Aufgabe geht es um die Modellierung und Simulation eines *Waschsalons*. Überlegen Sie sich dazu zunächst, durch welche Klassen und mit welchen Attributen Sie die Elemente des Salons repräsentieren möchten. Implementieren Sie die Methoden erst anschließend entsprechend der folgenden Beschreibung. In unserem Szenario finden Sie

- den Waschsalon (1, 5 Punkte), der drei Waschmaschinen der Marken Gründlich, Mühle und Busch beherbergt.
- Cent-Münzen (2 Punkte) in den Werteinheiten 1, 5, 10, 20 und 50. Jede Münze hat eine Methode, die sie in eine entsprechende `LinkedList<Muenze>` von Münzen der nächst kleineren Einheit umwandelt (sofern möglich, 1-Cent-Münzen bleiben erhalten), sowie eine Methode, die Auskunft über den Wert der Münze als Zahl zurückgibt.
- Socken (1 Punkt) mit der Sockengröße als Ganzzahl, und einer Gleitkommazahl, die den Grauwert als Farbe der Socke darstellt. 0 entspricht dabei *Weiß* und 1 entspricht *Schwarz*.
- Waschmaschinen mit einer maximalen Kapazität und einem Preis pro Waschgang. Waschmaschinen geben bei Überbezahlung passend zurück in Form einer Münzliste. Implementieren Sie folgende Marken von Waschmaschinen:

[2] siehe auch `https://de.wikipedia.org/wiki/Insertionsort`

  - *Mühle* mit Platz für bis zu 8 Socken zum Preis von 81 Cent pro Waschgang (1 Punkt)
  - *Busch* mit Platz für bis zu 9 Socken zum Preis von 12 Cent pro Socke (1 Punkt)
  - *Gründlich* mit Platz für bis zu 10 Socken zum Preis 91 Cent pro Waschgang; *Gründlich*-Waschmaschinen tendieren mit einer Warscheinlichkeit von 15% dazu, eine Socke aus dem Waschgang zu „fressen“[3]. Diese Socken werden aus dem Waschgang entfernt und bleiben stattdessen in der Maschine, um zukünftige Waschgänge durch ihre Farbe zu beeinflussen, solange bis ihre Größe unter den Wert 2 fällt, d.h. sie klein genug sind, um durch den Abfluß zu entkommen; so verschwinden sie schließlich aus der Maschine (2 Punkte).
- Waschen von Socken lässt Socken um eine Nummer einlaufen, und ändert den Farbwert der Socke auf den Durchschnitt aller gewaschenen Socken (0, 5 Punkte).

Beachten Sie bei Ihrer Implementierung, die Interna der beteiligten Objekte vor unvorgesehenem Zugriff durch andere Objekte zu schützen.

Testen Sie Ihre Simulation mit einem geeigneten `main`-Programm, durch welches jede Methode mindestens einmal aufgerufen wird.

**Hinweis:** Gehen Sie schrittweise an diese Aufgabe heran. Wenn Ihnen die Aufgabenstellung als komplex erscheint, kann es nützlich sein, zunächst auf einige Elemente zu verzichten und diese dann später einzubauen. Eine Idee wäre hier, zunächst nur 1-Cent-Münzen zu erlauben und sich so einige Funktionalität zu sparen.

**Hinweis:** Sie müssen für diese Aufgabe nicht auf Fehlerbehandlung achten.

[3] Die Waschmaschinen der anderen Marken verschlucken Socken niemals.