JAVA 09 – Fortgeschrittenes OOP

0. Wiederholung der Grundlagen

Wir erstellen Sie ein Programm für einen Obsthändler. Es soll in der Lage sein Beispielhaft die Anzahl von Äpfel, Pfirsiche und Orangen zu verändern, anzuzeigen und anhand eines Preises pro Stück den aktuellen Wert des Lagers in Euro wiederzugeben.

0.1. Primitive Datentypen, Kontrollstrukturen und Exceptions

Nutzen Sie zunächst drei **Integer** Variablen für Obst-Anzahl, den **Scanner** für die Eingaben und **System.out.print** für die Ausgaben. In einer endlosen **Do**-**While** Schleife wird ein Menü angezeigt, das per Eingabe von Zahlen gesteuert wird:

1. *Aktuelles Lager (Anzahl und Euro-Wert)*
2. *Anzahl Äpfel verändern*
3. *Anzahl Birnen verändern*
4. *Anzahl Orangen verändern*
0. *Programm beenden*

Die Auswertung der Eingabe erfolgt per **switch**, Fehlerbehandlung des Scanners per **Exceptions** *(Denken Sie an das nextLine-Problem bei der Eingabe von Zahlen!).* Beides führt zur Ausgabe einer Fehlermeldung gefolgt von erneuten Eingabe. Das Programm soll nicht einfach beendet werden (also kein Absturz durch Fehlerhafte Eingaben).

Die restlichen Optionen bitte sinnvoll in **2 statischen Methoden** implementieren. Die erste Methode muss als Parameter die aktuellen Mengen der drei Obstsorten enthalten. Die Einzelpreise können Sie zunächst einfach innerhalb der Methode deklarieren. Eine Beispielausgabe:

```
In unserem Lager befinden sich:
2 Äpfel zum Stückpreis 0.56€ - Summe: 1.12€
24 Pfirsiche zum Stückpreis 0.74€ - Summe: 17.76€
83 Orangen zum Stückpreis 0.46€ - Summe: 38.18€
Gesamtsumme: 57.06€
```

➔ Es kann hierbei zu „seltsamen“ Rundungsproblemen kommen, die man auf unterschiedliche Weise korrigieren könnte. Hierfür einfach mal das Thema „**JAVA float string format currency**“ nachschlagen (z.B. googeln)

Die zweite Methode für die Punkte 2, 3 und 4 soll wiederverwendbar sein. Als Parameter sind der Scanner und die jeweilige Anzahl vorgesehen. Die Methode fragt nach der relativen Veränderung der Anzahl (also nicht nach einer neuen Anzahl als Gesamtwert). *Die Anzahl soll nicht unter 0 sinken*. Denken Sie auch daran, dass primitive Datentypen NICHT innerhalb einer Methode direkt geändert werden kann. (**CallByValue**)

0.2. OOP, MVC, Enum und Collections

➔ Verändern Sie das Programm mit der Nutzung von Klassen:

Statt dreier Integer Variablen schreiben Sie nun 3 **Klassen** für Apfel, Birne und Orange – erstellen Sie also **Model**-Klassen. **Attribute/Member** sind:

- *fruitType* Erstellen Sie ein **Enum** „*FruitType*“ der den deutschen Namen beinhaltet!
- *amount* eine Integer Variable
- *UNIT_PRICE* der Preis pro Stück als **public final static float** Klassen-Attribut.

Ersetzen Sie außerdem die Integer-Variablen durch ein jeweiliges **Objekt** und passen Sie die notwendigen Stellen im Code an. *(inkl. Ausgabe der Obstnamen mit dem enum)*

Als nächstes packen Sie diese Obst-Objekte und Funktionalitäten von der main-Methode in einen Obst-**Controller**. Funktionalitäten (also **Methoden**) in diesem Controller sollen sein:

- *Verändern der Anzahl* Parameter: *FruitType, AnzahlVeränderung*
- *Zurückgeben der Anzahl nach ObstTyp* als **HashMap<FruitType, Integer>**
- *Zurückgeben des Lagerwertes nach ObstTyp* als **HashMap<FruitType, Float>**
- *Zurückgeben des aufsummierten Gesamt-Lagerwerts*

Die Methoden sollen im Falle eines Fehlers eine **Exception schmeißen**, bei der die Fehlermeldung sinnvollerweise als *message* zurückgegeben wird.

➔ Die Lagerwerte und auch die Summe sollen außerdem bereits „saubere“ Preise zurückgeben, also das Rundungsproblem soll danach kein Thema mehr sein!

Die Ausgabe und das Menü lassen wir zunächst einmal in der main-Methode (also erst einmal keine extra View-Klasse).

Das Ganze hat nun noch nicht so viel Benefit gegenüber der ersten Version. Damit das auch Sinn macht, werden wir die Funktionalität erweitern und neue OOP-Aspekte lernen!

1. Abstrakte Klassen und Interfaces

Es ist im Moment etwas unschön, dass wir bei allen drei Obst-Klassen den selben Code implementieren. Es wäre besser, wenn gleiche Attribute und Methoden zusammen genutzt werden.

1.1. Abstraktes Obst und Preis per Interface

Erstellen Sie eine **abstrakte Klasse** *Fruit.* Übernehmen Sie dieselben Member und Methoden der drei Obstsorten enthalten und lassen Sie die drei Obst-Klassen von Fruit ableiten. Alles (außer dem statischen UnitPrice) kann aus diesen **Child-Klassen** entfernt werden, da alle Member und Methoden vererbt werden.

Damit der FruitType korrekt gegeben ist, sollen die Child-Klassen im **Konstruktor** ihren jeweiligen FruitType an den Member von Fruit übergeben werden. Hierfür soll die Methode „setFruitType" genutzt werden. Um den FruitTypen nicht *von außen* verändern zu können, machen Sie sie **protected**.

Ein weiteres Problem ist der statische Preis. Nutzen wie ein „Fruit"-Objekt wissen wir ja nicht, dass die Child-Klasse den UnitPrice weiß. Daher soll die Fruit-Klasse ein **Interface** namens *UnitPriceable* implementieren, das eine Methoden „*int getAmount()*" und „*float getUnitPrice()*" fordert. Nutzen Sie das Interface bereits bei der Fruit-Klasse, implementieren Sie die getAmount() dort, aber die getUnitPrice-Methode erst in der konkreten Child-Klasse. Gleichzeitig soll der statische Member UNIT_PRICE nicht mehr public verfügbar sein.

Als letzten Schritt ersetzen Sie noch die drei Klassen im Controller durch eine **HashMap<FruiType, Fruit>** und passen Sie notwendige Stellen im Code an. Wir könnten nun diese Map dynamisch durch weitere Obsttypen erweitern (was wir aber aus Zeitgründen nicht tun werden, auch wenn ich zum herumexperimentieren anregen möchte…)

Passen Sie das restliche Programm bis hier hin an.

Besonderes Augenmerk legen Sie bitte auf die Berechnung der Summe. Nutzen Sie für die Berechnung eine statische Methode, der egal ist das es sich um Obst handelt. Sie soll stattdessen nur eine Collection des Interfaces *UnitPriceable* als Parameter erwarten und die Summe zurückgeben…

➔ ***Man kann keine ganze Collection casten, es müssen die einzelnen Objekte gecastet werden!***

Es sollte langsam auffallen, dass man noch einige Stellen vereinfachen und durch „generische" Konstruktionen zu verbessern kann.

Beispiel: die controller-methode „**getStorageAmounts**"

Es ist sinnvoll die Methode getStorageAmounts zu ersetzen und nur die neue HashMap, also die Collection aller Obstsorten zurückzugeben um direkt mit ihr zu arbeiten. Dies vereinfacht zum Beispiel die Methode zur Ausgabe des Lagers, die nun in der Lage ist, alle Obstsorten in einer kleinen Schleife auszugeben…

Etwas komplizierter ist die Anpassung des Menüs, dass sich bei der Erweiterung der ObstCollection ebenfalls dynamisch anpassen soll. Ziel ist es, dass in der main-Methode nicht mehr direkt mit Apfel, Pfirsich und Orange gearbeitet wird, sondern nur noch mit Obst (also Fruit und FruitType).

Hierfür könnte man zunächst eine weitere **HashMap<Integer, FruitType>** anhand der Größe der Obst-Collection zur Hilfe nehmen könnte, mit der dann das Menü erst zusammen gebaut werden kann und anschließend auch die gewählte Option verarbeitet werden…