Lehrstuhl Stuckenschmidt
Javakurs FSS 2012
Tag 3 - Objektorientierung

Tim Endlich, Thorsten Knöller, Niklas Dürr, Christian Meilicke

25.07.2012

Aufgabe 1: Haus

`haus`

Im package `haus` finden Sie die Klassen `Main` und `Haus`. Erweitern Sie `Haus` so, dass Sie in `Main` die Anzahl der erstellten Häuser ausgeben können ohne in dieser Klasse zu zählen.

Aufgabe 2: Time

`time`

Realisieren Sie eine Klasse `Time`, die in der Lage ist, eine Uhrzeit, bestehend aus Stunde, Minute und Sekunde, darzustellen. Intern soll die Uhrzeit durch einen einzigen Wert vom Typ int dargestellt werden, der die Uhrzeit in Sekunden speichert. Die Klasse soll folgende Methoden zur Verfügung stellen:

- Einen parameterlosen Konstruktor, der ein Time-Objekt mit der aktuellen Uhrzeit initialisiert.
- Einen Konstruktor mit drei Argumenten, der die Konstruktion beliebiger Uhrzeiten ermöglicht. Falls die eingegebene Uhrzeit nicht gültig ist, soll die aktuelle Uhrzeit verwendet werden.
- `getHour`, `getMinute` und `getSecond` zur Abfrage der eingestellten Uhrzeit.
- `setHour`, `setMinute` und `setSecond` zum Einstellen der Uhrzeit.
- `incrementSecond`, um die Uhrzeit eine Sekunde weiter zu stellen. Steht die Uhrzeit auf 23:59:59, soll sie auf 00:00:00 geändert werden.
- `toString` zur Umwandlung des Uhrzeitobjekts in einen String im Format ḧh:mm:ss.

Hinweis: Um die aktuelle Uhrzeit zu bestimmen können sie `java.util.Calendar` importieren und in der API sich anschauen wie man die Klasse benutzt.

Aufgabe 3: Kuchen

`kuchen`

Im package `kuchen` finden Sie zwei Klassen, die jeweils nur partiell implementiert sind. In der Klasse `Kuchen` ist der Konstruktor zu implementieren und die beiden Methoden `equals` und `toString` sind zu überschreiben. Die Rückgabe der `toString` Methode sollte nach diesem Schema sein:
`Sorte Kokos, Groesse klein, Form rund, Gebacken nein`

Die Klasse `VerzierterKuchen` soll von `Kuchen` erben. Zu der vorhandenen Funktionalität kommt die Variable `verziert` hinzu, ein Kuchen soll erst verziert werden können, wenn er gebacken wurde. In den Methoden `equals` und `toString` soll die neue Variable ebenfalls berücksichtigt werden die Rückgabe der `toString` Methode sollte wie folgt aussehen:
`Sorte Kokos, Groesse klein, Form rund, Gebacken nein, Verziert nein`

Hinweis: Die Testklasse `VerzierterKuchen` kann erst ausgeführt werden, nachdem die Vererbung eingebaut wurde.

Aufgabe 4: Fortbewegungsmittel

`fortbewegung`

Im package `fortbewegungsmittel` finden Sie zwei Interfaces, die Sie implementieren sollen. `Auto` soll `Fortbewegungsmittel` implementieren und `Oldtimer` von `Auto` erben und `Sammlerstueck` implementieren. In den Klassen `Fahrrad` und `SammlerFahrrad` wurden die Interfaces bereits implementiert. Sie sollten erstmal versuchen die Aufgabe zu lösen ohne sich die Fahrrad Klassen anzusehen, falls Sie Probleme haben können Sie sich dann daran orientieren.

Hinweis: Die Testklassen für die Autos können sobald die Interfaces implementiert und Oldtimer von Auto geerbt hat genutzt werden.

Aufgabe 5: Schützenfest

`shooting`

Im package `shooting` finden Sie zwei Klassen, die jeweils nur partiell implementiert sind. In der Klasse `Shooting` werden 10 Schüsse „abgefeuert“ und in einem Array angespeichert. Die Klasse `Shot` modeliert einen Schuss. Im Konstruktor werden x und y Koordinaten angegeben; würde man hier x = 0.0 und y = 0.0 übergeben, so wäre dies ein perfekter Schuss, der genau in die Mitte der Zielscheibe trifft. Die Methode `getDistance` soll den Abstand des Schusses zum Mittelpunkt der Zielscheibe, d.h. zu (0.0,0.0) angeben. Die Methode `getNumber` soll zurückgeben um den wievielten „abgefeuerten“ Schuss es sich handelt.

- Implementieren Sie zunächst den in der Klasse `Shot` fehlenden Code ohne die vorgegebenen Signaturen zu ändern. Danach können Sie die `main` Methode der Klasse `Shooting` ausführen, um zu sehen, ob ihre Implementierung funktioniert.
- Implementieren Sie nun den fehlenden Code in der Klasse `Shooting`, durch den der beste Schuss bestimmt wird!

Aufgabe 6: Türme von Hanoi *

`hanoi`

Die Türme von Hanoi sind ein mathematischens Knobelspiel.

Das Spiel besteht aus drei Stäben A, B und C, auf die mehrere gelochte Scheiben gelegt werden, alle verschieden groß. Zu Beginn liegen alle Scheiben auf Stab A, der Größe nach geordnet, mit der größten Scheibe unten und der kleinsten oben. Ziel des Spiels ist es, den kompletten Scheiben-Stapel von A nach C zu versetzen.

Bei jedem Zug darf die oberste Scheibe eines beliebigen Stabes auf einen der beiden anderen Stäbe gelegt werden, vorausgesetzt, dort liegt nicht schon eine kleinere Scheibe. Folglich sind zu jedem Zeitpunkt des Spieles die Scheiben auf jedem Feld der Größe nach geordnet.

Erweitern Sie im Paket `hanoi` die Klassen `Tower` und `Hanoi`. In der Klasse `Tower` sind folgende Methoden zu implementieren:

- `init` soll den Tower mit Scheiben (`Brick`) füllen, das Array tower sollte so gefüllt werden, dass die größte Scheibe an Position 0 ist und die kleinste Scheibe an der letzten Position.
- `push` soll eine Scheibe auf den `Tower` legen, also an die erste freie Stelle in das Array einfügen.
- `pop` soll die oberste Scheibe vom `Tower` nehmen (das letzte Element zurückgeben und aus dem Array löschen)

In der Klasse `Hanoi` sollen sie eine Methode hinzufügen, die das Spiel löst, also die Scheiben sortiert von Turm 1 zu Turm 3 verschiebt. Diese Methode können Sie in der `Main` Klasse aufrufen um zu überprüfen ob ihr Algorithmus richtig funktioniert. Sie können auch die `print` Funktion verwenden um zu überwachen, wie ihr Algorithmus arbeitet und Fehler zu finden. Weitere Informationen finden Sie auf

Wikipedia: http://de.wikipedia.org/wiki/Türme_von_Hanoi

Es steht ihnen natürlich frei weitere Methoden zu implementieren um damit das Problem lösen.

Aufgabe 7: Kamele und Karawane *

`kamel`

Stellen Sie sich vor, Sie kommen in die engere Wahl für die Ausschreibung um das lukrative Projekt E-Camel 2000, mit dem führenden nordafrikanischen Karawansereien elektronisch kontrollieren möchten. Entwickeln Sie daher die nachfolgend beschriebenen Klasse.

Eine Karawane besteht aus einer Anzahl Kamele. Ein Kamel hat folgende Eigenschaften:

`int maxpace`
Die Anzahl der Meilen, die das Kamel unbeladen pro Stunde zurücklegt. Dieser Wert ist unveränderlich.

`int load`
Die Anzahl Ballen, die das Kamel im Moment trägt. Mit jedem Ballen Ladung nimmt die tatsächliche Reisegeschwindigkeit gegenüber der Maximalen Marschgeschwindigkeit um eine Meile/Stunde ab. Für `load >= maxpace` lässt sich das Kamel nieder und steht nicht mehr auf.

`Camel next`
Die Kamele einer Karawane sind in einer Reihe aneinander gebunden. An jedem einzelnen Kamel hängt genau ein nachfolgendes Kamel. Nur am letzten Kamel der Karawane hängt kein weiteres mehr.

Definieren Sie eine Klasse `Camel` mit den folgenden Methoden:

`Camel int mp`
Erzeugt ein neues Kamel mit der unveränderlichen Marschgeschwindigkeit `mp > 0`

`int maxPace`
Liefert die maximale Marschgeschwindigkeit dieses Kamels.

`int pace`
Liefert die tatsächliche Marschgeschwindigkeit dieses Kamels. Dabei wird die aktuelle Ladung berücksch-tigt.

`void setLoad(int l)`
Belädt diese Kamel mit `l` Ballen (`l>=0`). Diese Ladung ersetzt eine eventuell vorher vorhandene Ladung.

`int getLoad()`
Liefert die aktuelle Ladung dieses Kamels als Anzahl Ballen.

`void setNext(Camel c)`
Hängt das Kamel `c` hinten an dieses Kamel an. Mit `c = null` wird dieses Kamel zum letzten der Karamwane.

`Camel getNext()`
Liefert das nachfolgende Kamel oder `null`, wenn dieses Kamel das letzte in der Karawane ist.

Eine Karawane besteht aus einer Anzahl Kamele, möglicherweise auch gar keinen. Definieren Sie die Klasse `Caravan` mit den folgenden Methoden:

`Caravan()`
Erzeugt eine neue Karawane. Die Karawane hat zunächst noch keine Kamele.

`int pace()`
Liefert die Reisegeschwindigkeit dieser Karawane, die vom langsamsten Kamel bestimmt wird. Dabei wird die Ladung der Kamele berücksichtigt.

`void addCamel(Camel c)`
Fügt das Kamel `c` in diese Karawane ein.

`void removeCamel(Camel c)`
Nimmt das Kamel `c` aus dieser Karawane heraus.

`void unload()`
Entlädt alle Kamele dieser Karawane.

`void addLoad(int l)`
Verteilt zusätzlich `l` Ballen Ladung so auf die Kamele dieser Karawane, dass die Reisegeschwindigkeit möglichst hoch bleibt.[1]

Hinweis: Die Testklassen funktionieren erst nach Implementation der obigen Methoden.

[1] Entnommen aus: 'Das Java-Praktikum', von Reinhard Schiedermeier und Klaus Köhler