ANGABENSAMMLUNG DER ZWEITEN EPROG-TEILPRÜFUNG

WINTERSEMESTER 2007

DIESE SAMMLUNG ERHEBT KEINEN ANSPRUCH AUF VOLLSTÄNDIGKEIT ODER RICHTIGKEIT.

EIN HERZLICHES DANKE FÜRS POSTEN DER ANGABEN AN (OHNE SPEZIELLE REIHUNG):
Thomarsch, Kuoni, Horrendus, kodiacc, snake!, nowi
SOLLTE IN DIESER LISTE JEMAND VERGESSEN WORDEN SEIN, SCHICKT MIT BITTE EINE PM.

EINES NOCH:
DANKE **a9bejo** FÜR DAS AUSPROGRAMMIEREN DER GANZEN STRING-BEISPIELE. DEINE LÖSUNGSVORSCHLÄGE HELFEN EINEM WIRKLICH WEITER.

ANGABE 1

BEISPIEL 1

Ein Algorithmus für eine Stringüberprüfung:
Eine Klasse `myString` enthält einen String `internal` (zB. : *"sdroxebra"*) und man soll überprüfen, ob man den eingegebenen String (zB.: *"boxer"*) mit dem `internal` bilden kann. Wobei man auch auf Doppelwerte aufpassen muss. Mit *"rogdraodmpe"* kann man beispielsweise nicht *"programme"* bilden, weil ein m fehlt.

BEISPIEL 2

Time: speichert eine Uhrzeit im Format HH:MM. Soll `toString methode` und eine um die *"Zeit in Minuten"* zurückzugeben enthalten.
Event: speichert ein Event mit den Parametern: `String event; Time from; Time to`. Also quasi den Titel des Events inkl. Start- und Endzeit. Es soll weiters mit einer Exception überprüft werden ob Start- und Endzeit korrekt sind (Startzeit darf nicht später sein als Endzeit etc..)
Timetable: enthält einen Wochentag im Format `String` (z.B.: Montag) über den Konstruktor, und soll eine beliebige Anzahl von Events handhaben können (Collections! ArrayList war in dem Fall vorgegeben).
Der Knackpunkt war die Methode, die Kollisionen in Events anzeigen soll. (Dh. man muss überprüfen ob sich ein Event im Zeitraum des anderen befindet und diese ausgeben)

ANGABE 2

BEISPIEL 1

Klasse *myString* (oder so Ähnlich): In dieser Klasse wird ein *String internal* gespeichert. Die Methode *public int comprimize()* war zu realisieren:
Dabei sollte *internal* (zB *"1Dies 2ist 3ein 5Programmier00test2"*) komprimiert werden.
Der komprimierte *String* war in *internal* zu speichern *("Dies ist ein Programmiertest")* und die Ziffern die sozusagen wegkomprimiert wurden als *Integer* zurückgeliefert werden (1235002)

Hier wurden die Funktionen *Character.isDigit(char c)* und *Character.digit(Char c, int radix)* um zur Lösung zu kommen vorgeschlagen. (*radix* --> auf 10 setzen dann wird *char c* in *int mit Basis 10 (Dezimalsystem)* umgerechnet)

BEISPIEL 2

War ein Kino. Die Klassen hießen *Cinema*, *Tier*, *Main* und *Exception*...
Also man hat in der Main-Klasse den Aufruf mit *Cinema cin1 = new Cinema("Saal1", int a[] = {5,5,5,5,5,4,4,4,4,3,3,3})* z.B. Das heißt er legt einen Raum an mit dem Namen und dann die Reihen. Reihe 1 hat 5 Plätze, Reihe 2 hat 5 Plätze etc. und so weiter. Hinterher muss man dann die Plätze in den Reihen mit *from, to* buchen können.

Außerdem soll man mit einem *toString* wo dann gebuchte als „*#*" ausgegeben werden und ungebuchte Plätze mit „*.*"
z.B.
```
.......##..#..
...###..##....
.#..##.##.....
usw.
```

ANGABE 3

BEISPIEL 1

Klasse *String2*, mit einem *String internal* und eine *Methode squeezeString()* ist zu schreiben

```
public void squeezeString (char c)
//alle Vorkommnisse von c sollen aus internal rausgelöscht werden
(z.B. internal = eprogtest ... squeezeString(e) -> progtst
```

BEISPIEL 2

Typisches Objektorientiertes Beispiel um Collections & Exceptions anzuwenden
Item: Speichert ein Item mit *String name* und *double price*
CartEntry: Speichert ein Objekt der *Klasse Item* und einen *int Anzahl*
Cart: Einkaufswagerl, soll eine unbestimmte Anzahl von *CartEntrys* speichern
Beim hinzufügen soll nur eine *Anzahl > 0* hinzugefügt werden können und falls es sich schon im Einkaufswagerl befindet soll bei diesem Eintrag um die hinzugefügte Menge CartEntries erhöht werden.
Beim Entfernen soll auch wieder nur eine Menge >= 0 entfernt werden können, wenn sich dadurch die Menge des Items auf 0 reduziert soll der gesamte Eintrag gelöscht werden, sonst soll er nur reduziert werden, wenn es das zu entfernende Item gar nicht gibt --> Fehler.

ANGABE 4

BEISPIEL 1

Zwei Strings miteinander verschmelzen. Also z.B:
internal = "Hello"
s = "World"
Ergebnis "HWeolrllod".
Der *String s* wird der Methode in der die Operation durchgeführt wird übergeben der andere *String internal* ist in der Klasse definiert und wird mittels Konstruktor gesetzt.

BEISPIEL 2

War ein Bücherregal wobei jedes *Buch* einen *Titel* und einen *Author* und eine *Breite (double)* hat. Es gibt eine *Klasse Bücherregal* welche Bücherobjekte verwaltet (mittels ArrayList). Dabei muss bevor ein Buch hinzugefügt wird geprüft werden, ob das Buch in das Regal passt, da dieses eine bestimmte *Breite (double)* hat. Weiters hat jedes Regal eine *Bezeichnung* (zb. Regal 1). Dann gibt es noch eine *Klasse Bibliothek* welche *Bücherregalobjekte* verwaltet (mittels ArrayList). Bibliothek hat eine *Methode der der Titel und der Autor eines Buches* übergeben werden, und es soll geprüft werden ob das Buch in einem Regal ist. Wenn ja, dann soll Folgendes ausgegeben werden: *„Regalbezeichnung: Anzahl der Exemplare"* wenn nicht vorhanden dann *"Nicht Lagernd"*. Alle Methoden die man benötigt waren vorgegeben, nur die Funktion musste hinzugefügt werden. *ArrayList* musste verwendet werden da oben stand *import java.util.ArrayList;*

ANGABE 5

BEISPIEL 1

Die gegebene, unvollständige *Klasse String2* soll eine spezielle Funktionalität zum Bearbeiten von Strings zur Verfügung stellen. Die Besitzt ein *Datenelement internal*. Vervollständigen Sie die Methode
public int deleteStringsIgnoreCase(String s)
welche alle Vorkommen des *Strings s* aus dem String *internal* entfernt (Gross-/Kleinschreibung NICHT signifikant) und als Rückgabeparameter die Anzahl der gelöschten Vorkommen zurückgibt. Nach dem Aufruf der Funktion darf s in internal nicht mehr vorkommen. Beispiel *„Ab caabbAABABaB BC“* mit *„ab“* oder *„Ab“* oder *„aB“* oder *„AB“* ergibt *„ cA BC“* und als Rückgabeparameter 6.

BEISPIEL 2

Die *Klasse Transaction* repräsentiert einen Vermerk über eine Geldtransaktion. Sie besitzt einen Konstruktor dem der *Geldbetrag*, der *Transaktionstext* und das *Transaktionsdatum* übergeben werden.

Folgende Methoden sollen den Zugriff auf die Datenelemente ermöglichen:

- *public double getAmount()*
- *public String getText()*
- *public Date getDate()*

Hinweis: Hier wird die Klasse *java.util.Date* benutzt um ein Datum zu repräsentieren. Diese besitzt unter anderem die Methoden *boolean before(Date)* und *boolean after(Date)* zum Vergleichen zweier Data und eine Methode *public String toString()* um eine lesbare Repräsentation des Datums, die unten gebraucht wird, zu erhalten.

Die *Klasse Account* repräsentiert ein Bankkonto, das Transaktionen durchführt und diese mit protokolliert, d.h. bei jeder durchgeführten Transaktion einen Vermerk speichert. Dem Konstruktor wird der *Überziehungsrahmen (positiver Betrag als double-Wert)* des Bankkontos übergeben. Weiters hat die Klasse ein *Datenelement vom Typ double*, das den *Kontostand* speichert. Der Kontostand wird mit 0 initialisiert.

Implementieren Sie die Methoden

- `public void deposit(double amount, String text, Date date) throws AccountException`

- `public void withdraw(double amount, String text, Date date) throws AccountException`

die einen Geldbetrag auf das Konto überweisen (*deposit*) bzw. vom Konto abheben (*withdraw*). Das Konto macht dabei im *Protokoll den Vermerk mit Geldbetrag, dem Transaktionstext und dem Transaktionsdatum*. Das Datum darf nicht vor dem Datum der letzten Transaktion liegen. Der Betrag muss in beiden Fällen größer als 0 sein. Wenn amount ≤ 0 ist, das Datum vor dem Datum der letzten Transaktion liegt oder bei einer Abhebung der Überziehungsrahmen überschritten würde (d.h. Geldbetrag < -Überziehungsrahmen), soll der Vorgang abgebrochen und eine

Exception vom *Typ AccountException* geworfen werden.
Die Methode

- `public double getBalance()`

liefert den aktuellen Kontostand zurück,

- `public boolean checkBalance()`

überprüft anhand des Transaktionsprotokolls ob der aktuelle Kontostand den richtigen Wert hat, d.h. ob die Summe aller Transaktionen den Kontostand ergibt. Sollte das nicht der Fall sein wird *false* zurückgegeben, sonst *true*. Wenn diese Methode *false* liefert, bedeutet das, dass die *Account Klasse* FEHLERHAFT ist (Programmierfehler).
Zusätzlich soll die Methode

- `public void printLog()`

den *Transaktionstext*, das *Transaktionsdatum* und den *Betrag* für JEDE TRANSAKTION im Protokoll der Reihe nach durch Zeilenvorschub getrennt in der Konsole ausgeben.

Hinweis: Vorbedingungen der gefragten Methoden müssen nicht überprüft werden, es sei denn es wird in der Angabe explizit verlangt.

ANGABE 6

BEISPIEL 1

Es soll in einer *`Klasse String2`* überprüft werden, ob der über den Konstruktor übergebene *`String internal`* ein Palindrom ist. Dazu ist die Methode

- `public boolean isPalindrome()`

zu implementieren. Zur Überprüfung haben nur die Zeichen „a“ bis „z“ Relevanz, Sonderzeichen und Gross-/Kleinschreibung werden ignoriert.

Hinweis: Ein Palindrom ist eine Zeichenkette, die von rechts nach links und links nach rechts gelesen das Gleiche ergibt, wie zum Beispiel „sum summus mus“ (sumsummusmus – heißt so viel wie „Ich bin die mächtigste Maus“ auf Deutsch) oder zum Beispiel „In girum imus nocte et consumimur igni“ (Wir irren des Nachts im Kreis umher und werden vom Feuer verzehrt).

Beispiel 2

???