Themen:
Algorithmik: Sequenz, Verzweigung

Aufgabe 5.1

a) Modifizieren Sie die Klasse Konto aus Übungsblatt 3, die ein Konto bei der Pleite & Geier-Bank modellieren soll.

Ein Konto ist gekennzeichnet durch

- den Kontoinhaber (ein Objekt vom Typ Person)
- die Kontonummer (ein **int**-Wert)
- den aktuellen Kontostand (saldo vom Typ **double**)
- Der Konstruktor erhält den Kontoinhaber als Parameter. (Es ist durchaus möglich, dass eine Person mehrere Konten hat.)
- Die Kontonummer wird automatisiert fortlaufend vergeben.
  Das erste erzeugte Konto erhält die Nummer 1000, das nächste die Nummer 1001 usw.
- Es soll keine Methode geben, um die Kontonummer oder den Inhaber zu ändern.
  Änderungen des Saldos sind möglich, aber nur über die Methoden einzahlen () bzw abheben() bzw ueberweisen() (siehe unten).
- Mit den beiden Methoden einzahlen () bzw abheben() ändert sich der Kontostand um den übergebenen Betrag.
  Einzahlungen und Abhebungen sind nur für positive Geldbeträge möglich. Bei Eingabe eines negativen Werts wird die Kontobewegung nicht ausgeführt, statt dessen die Fehlermeldung „negativer Betrag nicht möglich“ auf dem Monitor ausgegeben.
- Abhebungen sind außerdem nur in einer solchen Höhe möglich, dass das Konto nicht „überzogen“ wird, dh nur, wenn dadurch der Saldo nicht unter 0 sinkt. Falls der Kontostand zu gering ist, wird die Auszahlung nicht ausgeführt, statt dessen wird dann die Fehlermeldung „Auszahlung nicht möglich“ auf dem Monitor ausgegeben.
- Sehen Sie eine Methode ueberweisen() vor, um Überweisungen auf ein anderes Konto zu tätigen. Dadurch kann ein (positiver) Betrag von einem Konto auf ein anderes Konto überwiesen werden - aber auch dies nur, sofern das Konto dadurch nicht „überzogen“ wird.
  (Ansonsten: $\rightsquigarrow$ Fehlermeldung)
- Die toString ()-Methode soll den Namen des Inhabers, die Kontonummer und den aktuellen Kontostand als String liefern.
  Eine Methode kontoauszug() zeigt diesen String auf dem Monitor an.

b) Ergänzen Sie die Klasse um folgende features:

- Die Bank erhebt für jede Kontobewegung (Abheben, Einzahlen, Überweisen) eine Gebühr (von zum Beispiel 0.12 Euro). Die Höhe dieser Gebühr kann sich ändern; wenn sie sich ändert, dann für alle Konten gleichermaßen.
- In einer Methode abschlussRechnung() werden die Gesamt-Gebühren für alle Kontobewegungen (seit der letzten Abschlussrechnung) **eines Kontos** berechnet. Für Kontobewegungen **vor** der Erhöhung wird die alte Gebühr berechnet, für alle Kontobewegungen **nach** der Änderung wird die neue Gebühr berechnet. (Die Gebühr könnte sich auch mehrmals innerhalb eines Abrechungszeitraums ändern.)

c) Erstellen Sie in einer Test-Methode in einer Main-Klasse drei Konten und lassen Sie auf diese Konten einige Beträge einzahlen und abheben. Lassen Sie nach jeder Kontobewegung den aktuellen Kontoauszug anzeigen.
Lassen Sie nun von einem Konto einen Betrag auf ein anderes Konto überweisen und sich von beiden

Konten anschließend den Kontoauszug anzeigen.
Die Bank erhöht nun die Gebühren um einige Cent pro Aktion. Lassen Sie auch danach noch einige Kontobewegungen ausführen.
Lassen Sie sich für alle drei Konten die Abschlussrechnung erzeugen.

Hinweise:

- **Bevor Sie zu coden beginnen**, konstruieren Sie **zuerst** auf dem Papier einen Beispiel-Verlauf für die Test-Main-Klasse. Wählen Sie Beispielwerte, für die Sie leicht von Hand die erwarteten Ergebnisse berechnen können.
- **Bevor Sie zu coden beginnen**, überlegen Sie genau, welche Variablen als Instanz-Attribute eines Kontos definiert werden sollten, welche als Klassen-Attribute zu definieren sind und welche als Parameter an Methoden übergeben werden müssen.
- Sie dürfen natürlich zusätzliche Hilfsvariablen (als Attribute oder als lokale Variablen innerhalb von Methoden) definieren.
- Sie dürfen natürlich auch zusätzliche Hilfsmethoden definieren - diese sollten dann aber alle **private** sein.

Aufgabe 5.2
Schreiben Sie in einer Klasse `MathUtil` je eine (statische) Methode `min2, min3, min4`, die von zwei bzw von drei bzw von vier ganzzahligen Eingabewerten den kleinsten bestimmt und zurückgibt.

Aufgabe 5.3
Ein Mobilfunkanbieter bietet verschiedene Tarife an, die aber alle nach dem gleichen Schema aufgebaut sind:

Der Vertrag kostet eine monatliche Grundgebühr. Dieser Betrag kann sich allerdings ändern; wenn er sich ändert, bewirkt das eine Kostenänderung für alle Verträge.

In der Grundgebühr ist ein gewisses Datenvolumen pro Monat frei. Sobald dieses Datenvolumen ausgeschöpft ist, kostet jedes weitere Daten-Paket einen zusätzlichen Betrag pro GB.

Das Datenvolumen und der Preis für zusätzliche GB sind in den verschiedenen Tarifen unterschiedlich. Erstellen Sie eine Klasse, die die unterschiedlichen Tarife modelliert.

In dieser Klasse sollte es eine Instanz-Methode `rechnung()` geben, die den für einen Monat zu zahlenden Betrag berechnet. Der Rechnungsbetrag hängt davon ab, wieviel GB Daten verbraucht wurden. Falls das Datenvolumen nicht ausgeschöpft wurde, gibt es keine „Gutschrift“.

Beispiele:
Die Grundgebühr sei 12.50 Euro pro Monat.

Im Basistarif gibt es 5 GB Datenvolumen pro Monat, zusätzliches Volumen kostet 0.80 Euro pro GB.
Für einen Verbrauch von 15 GB Daten innerhalb eines Monats wäre der Rechnungsbetrag

$= 12.50 + 10 \cdot 0.80 = 20.50$ Euro.

Im Premiumtarif hat man 20 GB Datenvolumen pro Monat, dafür kostet jedes darüber hinausgehende GB 1.50 Euro.
Bei diesem Tarif würde man für 15 GB verbrauchte Daten im Monat nur die Grundgebühr zahlen.

Falls Sie Lust haben, (jetzt und in Zukunft) ein wenig zu spielen:

Aufgabe 5.4

a) Erstellen Sie eine Klasse `Wuerfel`: Die Klasse verfügt über
   - einen statischen Zufallszahlengenerator (ein Objekt der API-Klasse `Random`),
   - ein einziges Instanz-Attribut `wert`, das ist die Augenzahl, die der Wuerfel zeigt, also ein Wert zwischen 1 und 6. Bereits bei Erzeugung eines Wuerfel-Objektes wird dieser Variablen ein zufälliger Wert zwischen 1 und 6 zugewiesen.
   - eine get-Methode für diesen Wert
   - eine Methode `wuerfeln()`, die sowohl das Instanzattribut `wert` neu setzt als auch den gewürfelten Wert zurückliefert
   - eine Klassenmethode `pasch()`, die zwei Würfel als Eingabe erhält und (ohne zu würfeln) darauf prüft, ob sie den gleichen Wert zeigen.

b) Implementieren Sie ein einfaches Spiel, das zwei (Computer-)Spieler gegeneinander spielen können: Jeder der Spieler wirft zwei Würfel. Die höhere Augensumme gewinnt (dh es kann auch ein „unentschieden“ geben). Allerdings gewinnt ein Pasch grundsätzlich gegen einen Wurf mit zwei verschiedenen Augenzahlen.

   Beispiel:
   $(2, 5)$ gewinnt gegen $(3, 1)$
   $(2, 2)$ gewinnt gegen $(5, 6)$
   $(3, 3)$ gewinnt gegen $(1, 1)$
   $(5, 2)$ gegen $(3, 4)$: unentschieden)

Im Foliensatz **Folien99** im LEA-Kurs (Kap 13: Exkurs) finden Sie Hinweise dazu, wie man in Java mit einem Zufallszahlengenrator umgeht.