Programmeren (1) *Examen 26–8–2009*

NAAM:

Schrijf al je antwoorden op deze vragenbladen (op de plaats die daarvoor is voorzien) en geef zowel klad als net af. Bij heel wat vragen moet je zelf Java-code schrijven. Hou dit kort en bondig. Je hoeft geen commentaarregels te schrijven en ook eventuele `import`-opdrachten zijn niet nodig. Programmeer 'helder' en 'met stijl'.

Wanneer er gevraagd wordt naar een programma*fragment* hoef je dit niet te verpakken in een methode of een klasse. Vergeet echter niet variabelen die je introduceert ook te declareren.

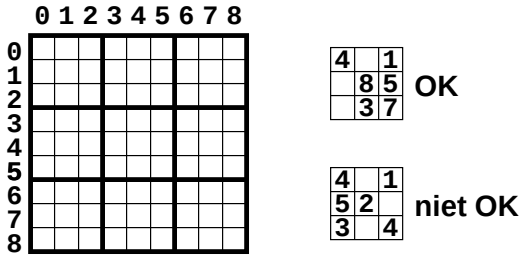
Gebruik zoveel mogelijk versie 5.0 van Java — zoals in de les: lijsten zijn 'generisch' en in plaats van iteratoren schrijf je een 'for each'-lus. Verwar niet tussen 'tabel' (Engels: array) en 'lijst' (`ArrayList` of `List`).

1. (10pt) **Lees eerst de volledige opgave grondig door vooraleer je begint.**

Een *sudoku* is een rooster van 9 op 9 vakjes die elk één cijfer kunnen bevatten (van 1 t.e.m. 9). Vakjes kunnen ook leeg zijn.

Opdat een dergelijk rooster echt een sudoku zou mogen worden genoemd, moet het aan enkele bijzondere eigenschappen voldoen. Zo mag elke rij elk cijfer van 1 tot 9 hoogstens één keer bevatten — en dezelfde regel geldt voor de kolommen.

Daarenboven wordt het rooster ook nog onderverdeeld in 9 afzonderlijke 3-bij-3 vierkanten (zie figuur) waarvan opnieuw geëist wordt dat ze elk cijfer hoogstens één keer bevatten. (We noemen dit de *sudoku-eigenschap*.)



We zullen de rijen, kolommen en vierkanten vanaf nu *gebieden* noemen. Merk op dat elk gebied precies 9 vakjes bevat.

In een toepassing waarmee dergelijke sudoku's op hun correctheid kunnen worden gecontroleerd, slaan we de inhoud van het rooster op als een twee-dimensionale tabel van gehele getallen. Ingevulde cijfers stellen we voor met de waarden 1 tot 9 en we gebruiken 0 om aan te geven dat een vakje (nog) leeg is.

(. . . vervolgt op afzonderlijk blad)

(Vervolg van vraag 1.)

De toepassing maakt gebruik van de volgende vier klassen: een *abstracte* klasse *GebiedChecker* en de klassen *RijChecker*, *KolomChecker* en *VierkantChecker*. Objecten van deze klasse hebben tot doel de sudoku-eigenschap van een gebied te controleren.

Het hoofdprogramma bevat een lijst van 27 dergelijke gebiedencheckers: 9 checkers voor rijen, 9 voor kolommen en 9 voor vierkanten.

Om een sudoku op zijn geldigheid te controleren, overloopt het programma deze lijst en roept telkens de methode *isGeldig* op voor het overeenkomstige gebied. (De methode *isGeldig* heeft *geen* parameters.) De volledige sudoku is enkel correct wanneer elk van deze oproepen `true` retourneert.

Belangrijk: De 27 gebiedencheckers worden één keer aangemaakt in het begin van het programma, op het moment dat het sudokurooster nog leeg is. Daarna wordt hetzelfde sudokorooster telkens opnieuw ingevuld en op zijn geldigheid gecontroleerd. Hierbij worden steeds dezelfde gebiedencheckerobjecten gebruikt en hetzelfde twee-dimenionale rooster. De inhoud van dit rooster kan echter telkens opnieuw verschillen.

Opgave

- Schrijf de Java-code voor deze vier klassen. Beperk je tot de methoden die je voor deze opgave nodig hebt. Vergeet echter niet om gepaste constructoren te implementeren.
- De methode *isGeldig* mag enkel in de klasse *GebiedChecker* worden gedefinieerd (en *niet* in één van de andere klassen).
- Om hergebruik van code te bevorderen laat je de gebiedcheckers de inhoud van hun gebied overbrengen naar een tabel van 9 gehele getallen die je daarna op geldigheid controleert door een functie op te roepen met de volgende hoofding:

```
public boolean sudokuEigenschap (int[] tabel)
```

- Implementeer deze functie *sudokuEigenschap* op een *efficiënte* manier: zorg ervoor dat je de tabel slechts één keer hoeft te doorlopen.
- Schrijf ook een programmafragment (onderdeel van het hoofdprogramma) waarin de lijst van gebiedcheckers wordt gedeclareerd, aangemaakt en opgevuld met de 27 gebieden.

Beantwoord deze vraag op de binnen- en achterkant van het dubbel blad!

2. (2 pt) Geef twee korte(!) Java-opdrachten die dienen om de kleine letters `'a'`, `'b'`,..., `'z'` om te zetten naar de overeenkomstige volgnummers 0,1,2,...,25, en omgekeerd.

Zet *ch* om naar zijn volgnummer en stop dit in *nr*	
Zet *nr* om naar de corresponderende kleine letter en stop die in *ch*.	

Hierbij is *ch* een variabele van het type `char` en *nr* van het type `int`. De opdrachten hoeven niet te werken wanneer *ch* iets anders bevat dan een kleine letter.

---

3. (1 pt) Een klasse *MyException* is een uitbreiding van *Exception* maar niet van *RuntimeException*. De volgende methode, die van deze klasse gebruik maakt, compileert niet.

```
public static void checkPositive (int count) {

   if (count < 0) {

      throw new MyException ("Niet positief");

   }

}
```

Verbeter de fout in bovenstaande code op een *zinvolle* manier.

---

4. (3 pt) Schrijf naast elk fragment een nieuw fragment dat precies hetzelfde doet, maar dan met een gewone for-lus in plaats van een 'for each'. (Let op, er zitten enkele addertjes onder het gras!)

```
for (String el : lijst) {

    System.out.println (el);

}
```

lijst is een lijst (*ArrayList*) van strings.

```
for (Double el : tabel) {

    el = 0.0;

}
```

tabel is een tabel (array) van reële getallen.

```
int som = 0;

for (int i : tab) {

   som += tab[i];

}
```

tab is een tabel (array) van gehele getallen.

5. (4 pt) Een rij getallen $a_0, a_1, \ldots, a_n$ noemen we *unimodulair* wanneer er een index $k, 0 \leq k \leq n$ bestaat zodat

$$a_0 < a_1 < \cdots < a_{k-1} < a_k > a_{k+1} > \cdots > a_n,$$

m.a.w., wanneer de rij eerst strikt stijgt en daarna strikt daalt. Het element a_k noemen we de *modus* van de rij. Een rij die volledig stijgend (of dalend) is, noemen we nog steeds unimodulair, met $k = n$ (of $k = 0$).

Opgave: Schrijf een methode *unimodulair* die een dergelijke rij als parameter neemt (in de vorm van een tabel van $n+1$ reële getallen) en de index k van de modus retourneert. Is de rij *niet* unimodulair, dan moet de methode een negatief getal teruggeven als waarde. (Je mag aannemen dat de rij minstens 1 getal zal bevatten. Je hoeft de methode niet in een klasse te plaatsen.)

Belangrijk:

Overloop de rij eerst in de ene richting en daarna in de andere richting (verplicht).

Je zal in je methode wellicht één of meerdere lussen nodig hebben. Je gebruikt hier best een zogenaamde 'while-lus met dubbele conditie'.

---

6. (4pt) In een twee-dimensionale tabel houden we de spelduur bij van de cricketwedstrijden die gespeeld werden tussen verschillende ploegen. (Elke ploeg heeft één keer tegen elke andere ploeg gespeeld.)

Deze tabel *tijden* is op de volgende manier gestructureerd:

- De tabel bevat reële getallen.
- Elke ploeg wordt voorgesteld door een uniek volgnummer, startend vanaf nul. Dit volgnummer wordt gebruikt als rij- en kolomnummer in de tabel *tijden*.
- De spelduur van de wedstrijd tussen de ploeg met volgnummer r en de ploeg met volgnummer k bevindt zich in de tabel op rij r en kolom k, en ook nog eens op rij k en kolom r.
- De 'diagonaal'elementen van de tabel (waar rij- en kolomnummer gelijk zijn) zijn allemaal nul.

Opgave : Schrijf een programmafragment dat de tijdsduur afdrukt van de wedstrijd die het minst lang heeft geduurd. *Belangrijk:* Vermijd hierbij al te veel onnodig werk.

7a. (2 pt) Een klasse *Kromme* bevat onder andere de volgende methoden :

```
public boolean isGesloten ();

public double oppervlakte ();
```

De eerste methode vertelt of de kromme al dan niet gesloten is. De tweede methode geeft de oppervlakte terug die door de kromme wordt ingesloten (een positief reëel getal) maar is enkel betrouwbaar wanneer het om een gesloten kromme gaat. (Bij andere krommen kan in principe eender welk reëel getal worden teruggegeven als resultaat).

Schrijf een methode *groteKrommen* die als argument een lijst van krommen neemt en als resultaat een nieuwe lijst teruggeeft met daarin alle gesloten krommen uit de originele lijst waarvan de oppervlakte strikt groter is dan 1.

7b. (2 pt) Voor een tweede versie van dit programma heeft men beslist om voor gesloten krommen een nieuwe klasse *GeslotenKromme* te definiëren die de klasse *Kromme* uitbreidt. De methode *oppervlakte* bestaat dan enkel nog in deze nieuwe klasse en de methode *isGesloten* is verdwenen (uit beide klassen).

Schrijf onder je eerste antwoord een kopie van *groteKrommen* maar nu aangepast aan deze tweede variant. Geef duidelijk aan wat je aan je oplossing hebt moeten veranderen. De oorspronkelijke lijst moet nog steeds zowel gesloten als ongesloten krommen kunnen bevatten en je methode moet hetzelfde retourtype hebben als in het eerste geval.