

# Tentamen

## Object-orientatie (I00063)

Woensdag 19 januari 2005, 8.30 – 10.30, HG 00.303

Dit tentamen is “gesloten boek” en bestaat uit vijf (5) opgaven. Bij elke opgave staat in de kantlijn het maximale aantal punten dat u kunt verdienen met het oplossen van de opgave. In totaal zijn er 100 punten te verdienen. De eerste tien punten heeft u al binnen als u op elk in te leveren blad uw naam en studentnummer schrijft. Succes!

### Opgave 1

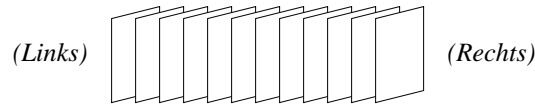
10

Om er even in te komen: tien goed/fout beweringen over OO en Java. Geef bij elke bewering aan of deze waar of onwaar is. Licht uw antwoorden kort toe (maak uw toelichting niet veel langer dan de bewering zelf).

1. Een Java klasse kan maar één superklasse hebben.
2. Een anonieme klasse kan alleen binnen een methode-body gedeclareerd worden.
3. Velden die `static` zijn, worden nooit “ge-garbage-collect.”
4. Voor ieder object `a` waarvoor `a.clone()` een legale expressie is, geldt dat `a.clone().equals(a)` `true` oplevert.
5. Excepties zijn eigenlijk overbodig. Als programmeurs correcte code zouden schrijven dan zou er nooit iets onverwachts gebeuren en dan zou er nooit een exceptie opgevangen hoeven worden.
6. Een klasse die “immutable” is, kan geen publieke velden hebben.
7. Als de variabele `a` een primitief type heeft, dan is `a.length` een illegale expressie (de compiler klaagt).
8. Als `B` een klasse is en de toewijzing `B b = (A)a;` is legaal (de compiler klaagt niet), dan is `A` een superklasse van `B`.
9. Encapsulatie wordt bereikt door alle velden `public` te maken.
10. De modelleertaal UML kan alleen gebruikt worden voor het modelleren van applicaties die in een OO taal (zoals Java) geïmplementeerd gaan worden.

## Opgave 2

Een *deque* (uitspraak: “dek”) is een datastructuur die zowel op een stack als op een queue lijkt.



Aan een deque kunnen zowel links en rechts objecten worden toegevoegd en ook weer worden verwijderd. Meestal wordt een deque vergeleken met een pak speelkaarten waarvan alleen de eerste en laatste kaart toegankelijk zijn.

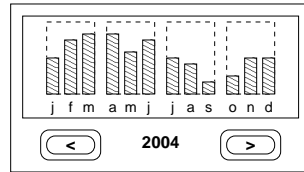
1. Implementeer in Java een klasse **Deque**. U hoeft geen Javadoc commentaar voor deze klasse te schrijven.
  - (a) Voorzie uw klasse van een constructor om een lege **Deque** te maken en methodes voor het toevoegen van objecten uiterst links en rechts.
  - (b) Voorzie uw klasse van methodes voor het verwijderen van objecten links en rechts. Deze methodes moeten ook het verwijderde object als resultaat teruggeven.
  - (c) Voorzie uw klasse van een `reverse` methode die de **Deque** omkeert.
    - Ofwel het return type is `void` en het huidige object verandert als gevolg van een aanroep van de methode,
    - of de methode levert een nieuw **Deque** object op en het huidige object blijft onveranderd.

Welke optie is in uw ogen beter? Licht uw antwoord toe.
  - (d) Voorzie uw klasse van een `equals` methode met dezelfde signatuur als de `equals` methode uit de **Object** klasse.  
 Zorg ervoor dat uw `equals` methode aan de gebruikelijke eisen voldoet die aan een equivalentierelatie gesteld worden.  
 Zorg ervoor dat gelijkheid tussen deque's invariant is onder omkeren. Dat wil zeggen: Als men een deque omkeert, dan is het resultaat gelijk (volgens `equals`) aan de oorspronkelijke deque.

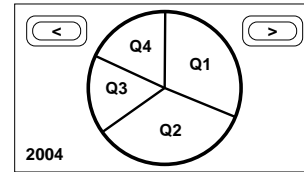
2. Door middel van een **Deque** zou u een **Stack** kunnen implementeren door alleen toevoegen en verwijderen aan één kant toe te staan. Zou u ervoor kiezen om **Stack** van **Deque** te laten erven, of andersom, of geen van beide? Licht uw antwoord toe.

## Opgave 3

Een nieuw te ontwikkelen grafische gebruikersinterface voor een financiële applicatie moet twee verschillende *views* op financieel cijfermateriaal kunnen laten zien: een taartdiagram en een staafdiagram.



*Staafdiagram*



*Taartdiagram*

Beide views zijn tegelijkertijd zichtbaar op het scherm, en in beide views kan de gebruiker de financiële gegevens aanpassen door middel van een aantal knoppen. Als de gegevens in de ene view aangepast worden, dan moeten wijzigingen instantaan in beide views zichtbaar zijn.

1. Welk standaard *design pattern* zou u hier toepassen?
2. Maak een ontwerp voor dit onderdeel van deze financiële applicatie en geef de verschillende klassen en interfaces en hun onderlinge relaties weer in een UML klassendiagram.
3. Geef voor elke klasse en interface in uw ontwerp een lijstje met belangrijke methodes met signatuur. U hoeft de methodes niet te implementeren.

## Opgave 4

20

Op het Keizer Karel Plein in Nijmegen komen zes verschillende wegen uit. We simuleren het plein met een Java implementatie. De simulatie wordt opgestart via een `main` methode die honderd auto's loslaat in de buurt van het plein.

```
public static void main(String[] arg) {
    for (int i = 0; i < 100; i++) {
        int vertrepunt = (int)(Math.random() * 6);
        int bestemming = (int)(Math.random() * 6);
        (new Thread(new Auto(vertrepunt, bestemming))).start();
    }
}
```

Elke auto start op één van de wegen die uitkomen op het plein, en kiest ook een bestemming aan één van deze wegen.

```
public class Auto implements Runnable
{
    private final int vertrepunt;
    private final int bestemming;

    public Auto(int vertrepunt, int bestemming) {
        this.vertrepunt = vertrepunt;
        this.bestemming = bestemming;
    }

    public void run() {
        try {
            Plein plein = Plein.getInstantie();
```

```

        rij(1); plein.op();
        rij((bestemming - vertrekpunt) % 6 + 6);
        plein.af(); rij(1);
    } catch (InterruptedException ie) {
    }
}

public void rij(int afstand) throws InterruptedException {
    Thread.sleep(afstand * 1000);
}
}

```

Op het plein kunnen maar tien auto's tegelijk rijden. Soms moet een automobilist dus even wachten voor hij of zij toegelaten kan worden. Daarom is voor het volgende protocol gekozen: Als een automobilist de claxon van een andere auto hoort, controleert de automobilist of het veilig is. Als het veilig is, rijdt de automobilist het plein op op zoek naar de juiste afslag. Bij het verlaten van het plein claxoneert de automobilist nog even om eventueel wachtende automobilisten erop te attenderen dat de toestand van het plein veranderd is.

```

public class Plein
{
    private static final Plein INSTANTIE = new Plein();
    private static final int CAPACITEIT = 10;
    private int autosOpPlein;

    public static Plein getInstantie() {
        return INSTANTIE;
    }

    public synchronized void op() throws InterruptedException {
        while (autosOpPlein + 1 > CAPACITEIT) {
            wait();
        }
        autosOpPlein++;
    }

    public synchronized void af() throws InterruptedException {
        autosOpPlein--;
        notifyAll();
    }
}

```

1. Alle velden van `Auto` zijn `final`. Is dit echt nodig?  
Zo ja, wat gebeurt er als het keyword `final` hier weggelaten wordt? Klaagt de compiler? Treedt er een runtime exceptie op?
2. Om ervoor te zorgen dat er maar één plein is, beschikt de klasse `Plein` over een statische instantie van zichzelf, die via de methode `getInstantie()` beschikbaar gesteld wordt.  
Welk standaard *design pattern* heeft men hier proberen toe te passen?

Is het, in de huidige implementatie, echt onmogelijk om meerdere instanties van `Plein` te maken? Zo nee, wat moet er aan de code veranderen om dit wel af te dwingen? Licht uw antwoord toe (ook als u “ja” antwoordt).

3. Kan het voorkomen, in de huidige implementatie, dat op het plein ooit meer dan tien auto’s rijden?

Zo ja, wat moet er veranderen om er voor te zorgen dat dit niet kan gebeuren? Licht uw antwoord toe (ook als u “nee” antwoordt).

4. Kan het voorkomen dat auto’s, die het plein op willen, niet worden toegelaten en voor altijd moeten blijven wachten?

Zo ja, wat moet er veranderen om er voor te zorgen dat dit niet kan gebeuren? Licht uw antwoord toe (ook als u “nee” antwoordt).

## Opgave 5

20

Gegeven is een klasse voor het representeren en uitrekenen van (één-dimensionale) wiskundige functies.

```
public abstract class Functie
{
    public abstract double eval(double x);

    public double[] plot(double laag, double hoog, double stapje) {
        double[] ys = new double[(int)((hoog - laag) / stapje)];
        for (double x = laag; x < hoog; x += stapje) {
            ys[(int)((x - laag) / stapje)] = eval(x);
        }
        return ys;
    }
}
```

Zoals u ziet, zijn niet alle methodes in deze klasse geïmplementeerd, vandaar dat besloten is de klasse `abstract` te declareren. Een bijzonder soort functies zijn de differentieerbare functies.

```
public abstract class DifferentieerbareFunctie extends Functie
{
    public abstract Functie afgeleide();
}
```

Een veelterm-functie wordt gekarakteriseerd door zijn coëfficiënten  $\alpha_0, \alpha_1, \dots, \alpha_n$  en heeft de volgende vorm.

$$f(x) = \alpha_n x^n + \dots + \alpha_1 x + \alpha_0$$

1. Maak een *concrete* klasse `VeeltermFunctie` die erft van `DifferentieerbareFunctie`. Voor verheffen van machten kunt u de methode `Math.pow(double, double)` gebruiken.