TENTAMEN

PROGRAMMEREN 1

vakcode: 213500
datum: 29 november 2001
tijd: 13.30–17.00 uur

Algemeen

- Bij dit tentamen mag alleen gebruik worden gemaakt van het boek van Jaime Niño & Frederick A. Hosch, getiteld *An Introduction to Programming and Object Oriented Design using JAVA*.
- Dit tentamen bestaat uit 5 opgaven, waarvoor in het totaal 100 punten behaald kunnen worden. Het minimaal aantal punten per opgave bedraagt 0 punten.
- Het eindcijfer voor het vak wordt als volgt bepaald. Als `n` het totaal aantal punten is dat voor het tentamen behaald is, dan krijgt `cijfer` de volgende waarde:

```
if (n <= 10)
    cijfer = 1;
else
    cijfer = (n+5)/10;
```

  Hierbij zijn `n` en `cijfer` beiden **`int`**-variabelen.

Opgave 1 (15 punten)

Geef van elk van de volgende beweringen aan of ze *waar* of *onwaar* zijn. Voor elk fout antwoord worden er drie punten afgetrokken.

a. Een *is-a* relatie tussen twee klassen kan met behulp van overerving gerealiseerd worden.

b. Op elk object kan de methode `equals` worden aangeroepen.

c. Een constructor van een subklasse zal altijd eerst de constructor van de superklasse aanroepen.

d. Hergebruik van code is alleen mogelijk bij het gebruik van overerving.

e. Als `x` en `y` twee **`double`**-variabelen zijn, dan levert `((int)x)/((int)y)` altijd dezelfde waarde op als `(int)(x/y)`.

f. Een abstracte klasse kan géén instantievariabelen bevatten.

g. Ook bij ongesorteerde lijsten is het mogelijk om een binair zoekalgoritme te gebruiken; het duurt dan alleen langer om het gewenste element te vinden.

h. Een **`static`** methode van een klasse kan gebruik maken van alle instantievariabelen van die klasse.

i. De methoden van een subklasse kunnen gebruik maken van de **`public`** en **`protected`** methoden en instantievariabelen van de superklasse.

j. In de implementatie (body) van een methode dient de preconditie (d.w.z. `require`) van die methode gecontroleerd te worden.

Opgave 2 (20 punten)

Beschouw de methode `even_oneven` met de volgende heading:

```
public boolean even_oneven(int[] a)
```

De methode `even_oneven` levert de waarde `true` op als de array `a` de 'even-oneven' eigenschap heeft. Een array heeft de 'even-oneven' eigenschap als de array uit getallen bestaat die alternerend even en oneven zijn. Het maakt daarbij niet uit of de array met een even of oneven getal begint of eindigt.
Voorbeelden:

```
[]            // heeft de even-oneven eigenschap
[27]          // heeft de even-oneven eigenschap
[4,1]         // heeft de even-oneven eigenschap
[1,2,1,4,1]   // heeft de even-oneven eigenschap
[1,2,3,2,4]   // heeft niet de even-oneven eigenschap
```

Opdrachten:

a. *(12 punten)* Geef de specificatie (inclusief pre- en postcondities) en een implementatie van de methode `even_oneven`.

b. *(8 punten)* Geef duidelijk aan wat de (invariante) betekenissen zijn van de variabelen die u in het algoritme gebruikt. M.a.w. geef de lusinvariant van de herhaling van het algoritme.

Opgave 3 (20 punten)

Beschouw de volgende incomplete definitie van een klasse `Datum`.

```
public class Datum {
    private int jaar;
    private int maand;
    private int dag;

    public Datum(int jaar, int maand, int dag) { ... }

    public int     getJaar() { ... }
    public int     getMaand() { ... }
    public int     getDag() { ... }
    public boolean kleinerDan(Object ander) { ... }
    public void    setDatum(int jaar, int maand, int dag) { ... }

    public static boolean isDatum(int jaar, int maand, int dag) { ... }
    public static boolean isSchrikkelJaar(int jaar) { ... }
}
```

De instantievariabelen, de constructor en de `get`-methoden spreken voor zichzelf. De methode `kleinerDan` controleert of *dit* `Datum`-object met een eerdere datum correspondeert dan `ander`. De methode `setDatum` kan gebruikt worden om een `Datum`-object te veranderen.

Het is belangrijk dat een `Datum`-object altijd een geldige datum representeert. Om te controleren of een tuple `(jaar,maand,dag)` met een geldige datum correspondeert, bevat de klasse `Datum` de klasse-methode `isDatum`. De methode `isDatum` dient rekening te houden met schrikkeljaren. Daartoe heeft de klasse `Datum` de `static`-methode `isSchrikkelJaar` die controleert of een `jaar` een schrikkeljaar is.

Een jaar is een schrikkeljaar als het deelbaar is door `4`. Uitzonderingen zijn jaren die deelbaar zijn door `100`; deze jaren zijn geen schrikkeljaren. Tenzij het jaar ook nog deelbaar is door `400`, dan is het jaar weer wel een schrikkeljaar. Voorbeelden:

```
27    // geen schrikkeljaar, niet deelbaar door 4
28    // schrikkeljaar, deelbaar door 4
1900  // geen schrikkeljaar, deelbaar door 100 en niet door 400
2000  // schrikkeljaar, deelbaar door 400
```

Opdrachten:

a. *(8 punten)* Specificeer (inclusief pre- en postcondities) en implementeer de constructor, de `get`-methoden en de methoden `kleinerDan` en `setDatum` van de klasse `Datum`.

b. *(8 punten)* Implementeer de klassenmethoden `isDatum` en `isSchrikkelJaar` van de klasse `Datum`.

c. *(4 punten)* Geef de klasseninvarianten van de klasse `Datum`.

Het is uiteraard toegestaan om de drie deelopgaven in een volledige klassendefinitie van de klasse `Datum` te definiëren.

Opgave 4 (25 punten)

In deze en de volgende opgave specificeren en implementeren we (een gedeelte van) een eenvoudige makelaarapplicatie. Een makelaar heeft panden in de verkoop en hij kan zelf bij andere makelaars panden aankopen. Er zijn verschillende soorten panden, bijvoorbeeld: appartementen, villa's, tussenwoningen, hoekwoningen, etc. In de makelaarapplicatie hebben al deze 'panden' één abstracte superklasse, de klasse `Pand`. Een `Pand` kan maar bij één makelaar tegelijk in de verkoop zijn. Als een makelaar een `Pand` in de verkoop heeft, dan kunnen andere makelaars zich bij de makelaar aanmelden als geïnteresseerden voor dat `Pand`. Een verkopende makelaar kan maar met één geïnteresseerde makelaar tegelijk in onderhandeling zijn. De volgorde van aanmelden bij de verkopende makelaar is hierbij belangrijk: wie-het-eerst-komt, die-het-eerst-maalt. Als de verkopende makelaar niet met de eerste geïnteresseerde makelaar tot overeenstemming kan komen, dan is de tweede makelaar aan de beurt. Een makelaar wordt gerepresenteerd door de klasse `Makelaar`.

Wanneer een `Pand` in de verkoop komt, dan stellen de eigenaar en de verkopende makelaar samen een *vraagprijs* op; dit is het bedrag waarvoor ze de woning willen verkopen. Tijdens de onderhandeling doet de geïnteresseerde makelaar een tegenbod, waarvoor hij het `Pand` wel zou willen kopen. Als de verkopende partij het tegenbod te laag vindt, maar wel verder wil onderhandelen kan hij de vraagprijs verlagen. Dit 'afdingen' gaat net zolang door totdat de verkopende en kopende partij het over de prijs eens zijn. Maar het komt natuurlijk ook voor dat de beide partijen niet tot overeenstemming kunnen komen. [1]

`Pand`-objecten en `Makelaar`-objecten worden in deze applicatie opgeslagen in `PandList`- en `MakelaarList`-objecten. De twee `List`-klassen bevatten de gebruikelijke `List`-methoden (zie ook Niño & Hosch). Hieronder volgt allereerst de partiële definitie van de klasse `Makelaar`.

```
/**
  * Makelaar. Een klasse voor het bijhouden van de gegevens
  * van een makelaar.
  */
public class Makelaar {

    private String    naam;             // naam van de Makelaar
    private PandList  tekoop;           // panden in de verkoop

    /**
     * Construeert een Makelaar-object.
     * @require naam != null && naam.size() > 0
     * @ensure  this.getNaam() == naam && this.getPanden().size() == 0
```

[1] De klassen `Makelaar` en `Pand` zijn verre van compleet bij dit tentamen. Zo is het bijvoorbeeld bij de klasse `Pand` wel mogelijk om als kopende partij te bieden, maar is het niet mogelijk om de koop daadwerkelijk te sluiten. Ook is het niet mogelijk om het bod van de verkopende partij te verlagen.

```
     * @param   naam de naam van de Makelaar
     */
    public Makelaar(String naam) {
        this.naam   = naam;
        this.tekoop = new PandList();
    }

    /** Levert de naam van deze Makelaar. */
    public String getNaam() {
        return naam;
    }

    /** Levert de panden die deze Makelaar te koop heeft. */
    public PandList getPanden() {
        return tekoop;
    }

    /**
     * Bepaalt de gemiddelde vraagprijs van de panden die
     * deze Makelaar in de verkoop heeft.
     * @require ! this.getPanden().isEmpty()
     * @ensure  result == gemiddelde van de vraagprijs van de
     *                    panden in this.getPanden()
     */
    public double gemiddeldeVraagPrijs() { ... }

    /**
     * Brengt een bod van bedrag guldens uit op pand.
     * @require pand != null && bedrag >= 0.0
     * @ensure  result == true als het gelukt is om een bod van
     *                    bedrag uit te brengen op pand
     *          als result
     *          dan pand.getVerkoper() == this
     *              pand'.getLaatsteBod() == bedrag
     * @param   pand   het Pand-object waarop een bod wordt gedaan.
     * @param   bedrag het bod (in guldens) dat wordt uitgebracht
     */
    public boolean doeBod(Pand pand, double bedrag) { ... }
}
```

Zoals u kunt zien worden in de klasse `Makelaar` alleen de naam van de makelaar en de lijst van `Pand`-objecten die de makelaar in de verkoop heeft bijgehouden. Hieronder volgt de eveneens partiële definitie van de abstracte klasse `Pand`.[2]

```
/**
 * Pand. Abstracte (super)klasse voor panden die door een
 * Makelaar verkocht en aangekocht kunnen worden.
 */
public abstract class Pand {
    private String        adres;
    private String        postcode;
    private Datum         tekoopPer;
    private double        vraagprijs;
    private double        laatsteBod;
    private Makelaar      verkoper;
    private MakelaarList  geinteresseerden;
```

[2] In dit tentamen worden de concrete subklassen van `Pand` zoals `Villa`, `Appartement`, `Tussenwoning`, etc. niet verder uitgewerkt.

```
    protected Pand(String adres, String postcode, Datum tekoopPer,
                   double vraagprijs, Makelaar verkoper) {
        this.adres      = adres;
        this.postcode   = postcode;
        this.tekoopPer  = tekoopPer;
        this.vraagprijs = vraagprijs;
        this.verkoper   = verkoper;
        this.laatsteBod = 0.0;
        this.geinteresseerden = new MakelaarList();
    }

    // De gebruikele get-queries.
    public String     getAdres()       { return adres; }
    public String     getPostcode()    { return postcode; }
    public Datum      getTeKoopDatum() { return tekoopPer; }
    public double     getVraagPrijs()  { return vraagprijs; }
    public double     getLaatsteBod()  { return laatsteBod; }
    public Makelaar   getVerkoper()    { return verkoper; }

    public MakelaarList getGeinteresseerden() {
        return geinteresseerden;
    }

    /**
     * Levert true op als de gespecificeerde Makelaar de huidige
     * (potentiele) koper is, d.w.z. als eerste aan de beurt is
     * in de lijst van potentiele kopers, n.l. this.getGeinteresseerden().
     * ...
     */
    public boolean isHuidigeKoper(Makelaar makelaar) { ... }

    /**
     * Voegt een potentiele koper (achteraan) de lijst van
     * potentiele kopers toe.
     * ...
     */
    public void voegKoperToe(Makelaar makelaar) { ... }

    /**
     * Verwijdert een makelaar uit de lijst van potentiele kopers.
     * Als makelaar de huidige potentiele koper is dan wordt het
     * laatste bod tevens op nul gezet.
     * ...
     */
    public void verwijderKoper(Makelaar makelaar) { ... }

    /**
     * Laat de makelaar een bod doen op dit Pand.
     * Een bod wordt alleen geaccepteerd als:
     *  -  de makelaar de huidige (potentiele) koper is
     *  -  het bedrag hoger is dan het laatste bod (van deze makelaar)
     * ...
     */
    public boolean doeBod(Makelaar makelaar, double bod) { ... }
}
```

De meeste instantievariabelen van `Pand`, de constructor en de `get`-methoden spreken voor zich. De vari-

abele `tekoopPer` bevat de `Datum` waarop het `Pand` in de verkoop is gegaan. De variabele `laatsteBod` bevat het laatste bod van de huidige (potentiële) koper. De variabele `geinteresseerden` bevat de lijst van geïnteresseerde `Makelaar`-objecten; de `Makelaar` die als eerst aan de beurt is, staat vooraan in de lijst.

Merk op dat de klasse `Pand` niet over de gebruikele `set`-methoden beschikt om de instantievariabelen te veranderen; de meeste instantievariabelen worden alleen bij de constructie van een `Pand`-object van een waarde voorzien.

Opdrachten:

a. *(6 punten)* Implementeer de methode `gemiddeldeVraagPrijs` van de klasse `Makelaar`.

b. *(7 punten)* Maak de specificaties van de methoden `isHuidigeKoper`, `voegKoperToe` en `verwijderKoper` van de klasse `Pand` af en implementeer deze methoden.

c. *(5 punten)* Implementeer de methode `doeBod` van de klasse `Makelaar`. Een makelaar kan alleen een bod doen op een pand als hij het niet zelf in de verkoop heeft (het is niet voldoende om dit in de preconditie van `doeBod` te eisen). De methode levert `true` op als het gelukt is om een bod uit te brengen.

d. *(7 punten)* Maak de specificatie van `doeBod` van de klasse `Pand` af en implementeer de methode. Het doen van een bod op een huis is alleen toegestaan als het `makelaar`-argument de huidige potentiële koper is (het is niet voldoende om dit in de preconditie van `doeBod` te eisen). Verder moet een nieuw bod altijd hoger zijn dat het laatst gedane bod. De methode levert `true` op als het gelukt is om een bod uit te brengen.

Opgave 5 (20 punten)

Een `Makelaar` heeft vaak een aanzienlijk aantal `Pand`-en in de verkoop. Om een goed overzicht te houden over de lijst met `Pand`-en in de verkoop, moet een `Makelaar` zijn `tekoop`-lijst op verschillende manieren kunnen sorteren. Daartoe wordt de volgende `interface` gedefinieerd:

```
interface PandSorteerCriterium {
    public boolean kleinerDan(Pand p1, Pand p2);
}
```

Opdrachten:

a. *(4 punten)* Schrijf een klasse `PSC_TeKoopDatum` die `PandSorteerCriterium` implementeert en `kleinerDan` de waarde `true` laat opleveren als `p1` eerder in de verkoop is gegaan dan `p2`.

b. *(4 punten)* Schrijf een klasse `PSC_AantalKopers` die `PandSorteerCriterium` implementeert en `kleinerDan` de waarde `true` laat opleveren als `p1` *meer* potentiële kopers heeft dan `p2`.

c. *(5 punten)* Voeg een methode `sorteerPanden` aan de klasse `Makelaar` toe die de lijst met panden in de verkoop sorteert volgens een sorteercriterium. De methode `sorteerPanden` heeft de volgende heading:

```
public void sorteerPanden(PandSorteerCriterium sc)
```

Hierbij mag u een sorteeralgoritme naar keuze gebruiken.

d. *(*7 punten) Schets een klassendiagram met daarin de klassen `Pand`, `Makelaar`, `PandList`, `MakelaarList`, `PandSorteerCriterium`, `PSC_TeKoopDatum` en `PSC_AantalKopers`. Hierin hoeft u niet de methoden van de klassen op te nemen.