TENTAMEN

Programmeren 2

vakcode: 213505
datum: 24 juni 2009
tijd: 9.00–12.30 uur

Algemeen

- Bij dit tentamen mag alleen gebruik worden gemaakt van de boeken van Niño en Hosch en van der Linden en één ander Java-leerboek naar keuze, van de practicumhandleiding van Programmeren 2 (in het bijzonder bijlage B), en van uitgeprinte kopieën van de hoorcollegesheets.
- Het aantal punten voor het tentamen wordt meegenomen in de berekening van het eindcijfer, op de manier zoals aangegeven in de handleiding.
- 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 van het tentamen wordt bepaald door de optelsom van de punten per opgave.

JAVADOC. Bij enkele opgaven van dit tentamen wordt gevraagd om, gegeven een `javadoc`-specificatie van een methode, de implementatie van die betreffende methode te schrijven. Bij het beantwoorden van een dergelijke opgave is het uiteraard niet nodig om het `javadoc`-commentaar te herhalen. Voeg alleen `javadoc`-commentaar toe als daar expliciet om gevraagd wordt.

Opgave 1 (20 punten)

In deze opgave beschouwen we een klasse `Tree` die gebruikt kan worden om `int`-waarden in een boomstructuur op te slaan. De klasse `Tree` ziet er als volgt uit.

```
public class Tree {
    public static final int MAX_VALUE = 100;
    public static final int MIN_VALUE = 0;

    private int   value;          /** @invariant: MIN_VALUE <= value && *
                                    *              value <= MAX_VALUE    */
    private Tree  left, right;

    /**
     * Construeert een Tree object.
     * @require Tree.MIN_VALUE <= value && value <= Tree.MAX_VALUE
     * @ensure  this.getValue() == value,
     *          this.getLeft()  == left, this.getRight() == right
     */
    public Tree(int value, Tree left, Tree right) {
        this.value = value;
        this.left  = left;
        this.right = right;
    }
```

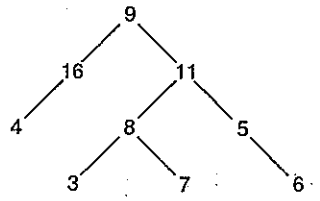
```
    /** @ensure MIN_VALUE <= result && result <= MAX_VALUE */
    public int  getValue()  { return value; }

    public Tree getLeft()   { return left; }
    public Tree getRight()  { return right; }
}
```

Het volgende programmafragment construeert bijvoorbeeld een `Tree tt`.

```
Tree tt = new Tree(9, new Tree(16, new Tree(4, null, null),
                                   null),
                      new Tree(11, new Tree(8, new Tree(3, null, null),
                                               new Tree(7, null, null)),
                                   new Tree(5, null,
                                               new Tree(6, null, null))
                               )
                   );
```

Figuur 1 toont een grafische representatie van `tt`.



Figuur 1: Grafische weergave van `Tree tt`.

a. (*8 pnt.*) Schrijf een methode `min` met de volgende heading:

```
/**
 * Levert de kleinste waarde binnen de Tree t.
 * Als t gelijk is aan null, levert de methode (Tree.MAX_VALUE+1) op.
 * @param  t de Tree waarvan het minimum bepaald wordt.
 * @return het minimum van t
 */
public static int min(Tree t)
```

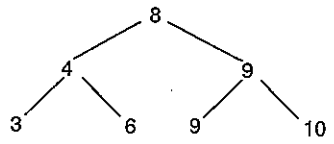
De methode `min` levert de kleinste waarde op die zich in de boomstructuur `t` bevindt. U mag er van uitgaan dat `Tree t` géén lussen bevat. Het is niet toegestaan om binnen de methode `min` de `private` instantie variabelen van `Tree` te gebruiken: u dient de `public` methoden van `Tree` te gebruiken.

Voorbeeld: `min(tt)` levert de waarde `3` op.

b. (*12 pnt.*) Schrijf een method `gesorteerd` met de volgende heading:

```
/**
 * Controleert of t gesorteerd is,
 * dwz voor elke boom zijn de elementen in de linker-subboom kleiner
 * of gelijk aan de waarde, de elementen in de rechter-subboom groter
 * of gelijk aan de waarde, en de subbomen zelf zijn ook gesorteerd.
 * @param  t de Tree waarvan wordt bepaald of deze gesorteerd is.
 * @return is t gesorteerd
 */
public static boolean gesorteerd(Tree t)
```

De methode `gesorteerd` controleert of de elementen in een boom gesorteerd zijn. Figuur 2 laat een voorbeeld zien van een gesorteerde boom.



Figuur 2: Grafische weergave van een gesorteerde boom.

U dient weer de `public` methoden van `Tree` te gebruiken. Verder dient u alle elementen in de boom maar één keer te inspecteren.

Hints: verget niet dat bij een recursieve oplossing gaat het om het kiezen van het simpel geval ('base case') en een ingewikkeld geval ('general case'). In deze opdracht is het gebruik van een hulpmethode aanbevolen.

Opgave 2 (20 punten)

Beschouw de `BankAccount` klasse gedefinieerd als volgt:

```
public class BankAccount {

        private String name;
        private double balance;

        public BankAccount(String name, double balance) {
                this.name = name;
                if (balance > 0)
                        this.balance = balance;
                else
                        this.balance = 0;
        }

        public boolean deposit(double amount) {
                if (amount < 0) return false;
                balance = balance + amount;
                return true;
        }

        public boolean withdraw(double amount) {
                if (amount > balance || amount < 0) return false;
                balance = balance - amount;
                return true;
        }
}
```

- Geef een nieuwe definitie van `BankAccount` waarin verschillende excepties worden gegooid in het geval dat een poging wordt gedaan om (i) een rekening aan te maken met een negatief initieel saldo, (ii) een negatief bedrag over te maken en (iii) een negatief bedrag op te nemen. Dit zijn geen standaard Java excepties, d.w.z. ze dienen alle drie gedefinieerd te worden als speciale excepties. Geef de definities van deze excepties ook in uw antwoord.

- Schrijf een test-programma dat alle drie mogelijke excepties gooit en vervolgens opvangt en hun omschrijving afdrukt. Het programma moet alle drie de gevallen testen. Het moet compileren en mag geen excepties propageren.

Opgave 3 (15 punten)

Een interface `java.util.Map` is een model van een wiskundige *functie*. Gegeven een *key*, levert een `Map` immers de bijbehorende *value*. In deze opgave ontwikkelen we enkele hulpmethoden voor het gebruik van (wiskundige) functies. Beschouw de klasse `Functie`.

```
public class Functie<X,Y> {

    /** Controleert of f een injectie is.
     * @return true als voor alle y in f.values() er precies
     *         een x in f.keySet() is met y == f.get(x) */
    public static <X,Y> boolean isInjectie(Map<X, Y> f) {
        // vraag a.
    }

    /** Levert de inverse van f op.
     * @requires isInjectie(f)
     * @ensures  result.values() == f.keySet() &&
     *           result.keySet() == f.values() &&
     *           voor alle (x,y) in result geldt: x == f.get(y) */
    public static <X,Y> Map<Y,X> inverse(Map<X,Y> f) {
        // vraag b.
    }
}
```

Bij deze opgave zult u de verschillende `static` methoden van de klasse `Functie` moeten implementeren. Hieronder staat een voorbeeld van het gebruik van deze methoden.

```
    public static void main(String[] args) {
        Map<Integer,Integer> f = new HashMap<Integer,Integer>();
        Map<Integer,Integer> g = new HashMap<Integer,Integer>();

        f.put( 1, 1);   f.put( 2, 4);   f.put( 3, 9);
        f.put( 4, 16);  f.put( 5, 25);  f.put( 6, 36);

        g.put( 1, 6);   g.put( 2, 5);   g.put( 3, 4);
        g.put( 4, 3);   g.put( 5, 2);   g.put( 6, 1);

        System.out.println("f:␣" + f);
        System.out.println("g:␣" + g);

        if (isInjectie(f))
            System.out.println("inverse(f):␣" + inverse(f));
        else
            System.out.println("f␣is␣geen␣injectie");
        if (isInjectie(g))
            System.out.println("inverse(g):␣" + inverse(g));
        else
            System.out.println("g␣is␣geen␣injectie");

    }
```

Het programma zal de volgende uitvoer (kunnen) genereren:

```
f: {2=4, 4=16, 6=36, 1=1, 3=9, 5=25}
```

```
g: {2=5, 4=3, 6=1, 1=6, 3=4, 5=2}
inverse(f): {4=2, 36=6, 9=3, 1=1, 16=4, 25=5}
inverse(g): {2=5, 4=3, 6=1, 1=6, 3=4, 5=2}
```

Let bij de implementaties van onderstaande methoden ook op de efficiëntie van de algoritmen. Met name teveel (impliciete) iteraties en/of (onnodige) cast-operaties worden aangerekend.

a. *(7 pnt.)* Geef een implementatie van de methode `isInjectie`.

b. *(8 pnt.)* Geef een implementatie van de methode `inverse`.

Opgave 4 (25 punten)

Een enthousiaste Java programmeur en sportliefhebber wil de tussenstand van zijn favoriet basketbalteam in Java modelleren. Hij heeft daartoe een klasse `StandJFrame` geschreven die de tussenstand van een wedstrijd op het scherm laat zien. De klasse `StandJFrame` ziet er als volgt uit:

```
1  public class StandJFrame extends JFrame
   implements ActionListener {
          private class Stand {

                  private int thuis;
6                 private int uit;

                  public void thuisScoort(int punten)    {
                          thuis = thuis + punten;
                  }
11
                  public void uitScoort(int punten)      {
                          uit = uit + punten;
                  }

16                public String getStand()     {
                          return "" + thuis + "-" + uit;
                  }
          }

21        private JButton     btThuis, btUit;
          private JTextField  tfStand;
          private JRadioButton vwRadio, _2ptRadio, _3ptRadio;
          private ButtonGroup punten;

26        private Stand       stand = new Stand();

          public StandJFrame() {
                  super("Stand");
                  init();
31        }

          public void init() {
                  Container c = getContentPane();
                  c.setLayout(new FlowLayout());
36
                  vwRadio = new JRadioButton("Vrije worp");
                  _2ptRadio = new JRadioButton("2 punten",true);
                  _3ptRadio = new JRadioButton("3 punten");
                  punten = new ButtonGroup();
41                punten.add(vwRadio);
```

```
                punten.add(_2ptRadio);
                punten.add(_3ptRadio);

                btThuis = new JButton("Thuis");
46              btThuis.addActionListener(this);

                btUit = new JButton("Uit");
                btUit.addActionListener(this);

51              tfStand = new JTextField(5);
                tfStand.setText(stand.getStand());
                tfStand.setEditable(false);

                c.add(vwRadio);
56              c.add(_2ptRadio);
                c.add(_3ptRadio);
                c.add(btThuis);
                c.add(btUit);
                c.add(tfStand);
61
                setSize(140,200);
                setVisible(true);
        }

66      public void actionPerformed(ActionEvent ev) {
                if (ev.getSource() == btThuis) {
                        stand.thuisScoort(getPunten());
                        tfStand.setText(stand.getStand());
                } else if (ev.getSource() == btUit) {
71                      stand.uitScoort(getPunten());
                        tfStand.setText(stand.getStand());
                }
        }

76      private int getPunten() {
                if (_2ptRadio.isSelected()) return 2;
                if (_3ptRadio.isSelected()) return 3;
                return 1;
        }
81
        public static void main(String[] args) {
                new StandJFrame();
        }
}
```

De klasse `StandJFrame` is een `JFrame` met drie componenten. Er zijn twee `JButtons` waarmee het scroren van het thuisspelende respectievelijk het uitspelende team kan worden doorgegeven. Het `JTextField` laat een `String`-representatie van de tussenstand zien. Figuur 3 toont een screenshot van het programma `StandJFrame`.

Hoewel een aardige poging, is de klasse `StandJFrame` niet volgens het *Model-View-Controller* principe ontworpen. In deze opgave moet de klasse `StandJFrame` worden opgesplitst in drie `public` klassen `Stand`, `StandView` en `StandController`, zodat de ontstane applicatie wel volgens het *Model-View-Controller* patroon ontworpen is. De *`public`* klasse `StandController` dient de interface `ActionListener` te implementeren. De klasse `StandView` bevat een methode `main` die er als volgt uit ziet:

```
public static void main(String[] args) {
    Stand model = new Stand();
    StandController controller = new StandController(model);
```

Figuur 3: Screenshot van de klasse `StandJFrame` in actie.

```
        StandView view = new StandView(controller);
        model.addObserver(view);
        controller.addView(view);
        view.update(model, null);
    }
```

Hint: Het nieuwe ontwerp komt er grotendeels op neer dat stukken van `StandJFrame` naar `Stand`, `StandView` en `StandController` verhuizen. Het is niet nodig om alle code van `StandJFrame` over te schrijven. U kunt volstaan met aan te geven welke gedeelten naar welke klasse verhuizen en wat er veranderd dient te worden.

Opgave 5 (20 punten)

Bij deze opgave dient u de mogelijke uitkomsten van enkele programma's te bepalen. Naast 'normale uitvoer' zouden ook de volgende 'uitkomsten' kunnen optreden:

- *vertaalfout*: het programma is geen correct Java programma;
- *runtime-fout*: er wordt een `Exception` gegooid;
- *geen uitvoer*: bijvoorbeeld vanwege een deadlock.

U hoeft uw antwoorden *niet* te motiveren. U zult zien zijn dat de opgaven na opgave a. variaties zijn op het oorspronkelijke programma. Als u meent dat een variatie de uitkomst van het oorspronkelijke programma niet verandert, kunt u volstaan met "als a." als antwoord.

Beschouw het Java-programma `Worker`.

```
class Cell {
    int v = 1;
    synchronized public int  get()       { return v;       }
    synchronized public void set(int v)  { this.v = v;     }
}

class Alex extends Worker {
    public Alex(Cell c)     { super(c);          }
    public void run()       { int v=c.get();
                              c.set(v+4);        }
}

class Bram extends Worker {
    public Bram(Cell c)     { super(c);          }
    public void run()       { int v=c.get();
                              c.set(v*4);        }
}
```

```
public class Worker extends Thread {
    protected Cell c;
    public Worker(Cell c)    { this.c = c;        }

    public static void main(String[] args) {
        Cell cell = new Cell();
        Worker w1 = new Alex(cell); w1.start();
        Worker w2 = new Bram(cell); w2.start();

        try { w1.join(); w2.join(); }
        catch (InterruptedException e) {}

        System.out.println(cell.get());
    }
}
```

a. *(3 pnt.)* Wat zijn de mogelijke uitkomsten van het programma `Worker`?

b. *(3 pnt.)* In het oorspronkelijke programma wordt het gehele `try-catch`-blok (twee regels dus) uit de methode `main` verwijderd. Wat zijn nu de mogelijke uitkomsten van het programma?

c. *(2 pnt.)* In het oorspronkelijke programma wordt het gehele `try-catch`-blok met hun inhoud (twee regels dus) uit de methode `main` verwijderd. Daarnaast worden ook de aanroepen van `w1.start()` en `w2.start()` in de methode `main` veranderd in respectievelijk `w1.run()` en `w2.run()`. Wat zijn nu de mogelijke uitkomsten van het programma?

d. *(2 pnt.)* In het oorspronkelijke programma wordt de methode `set` van de klasse `Cell` als volgt veranderd:

```
public synchronized void set(int v) {
    this.v = v;
    notifyAll();
}
```

Wat zijn nu de mogelijke uitkomsten van het programma?

e. *(2 pnt.)* In het oorspronkelijke programma worden de methoden `run` van de klassen `Alex` en `Bram` als `synchronized` methoden gedefinieerd. Wat zijn nu de mogelijke uitkomsten van het programma?

f. *(3 pnt.)* In het oorspronkelijke programma wordt de methode `run` van de klasse `Alex` vervangen door:

```
public void run() {
    synchronized(c) { c.set(c.get()+4); }
}
```

Tevens wordt de methode `run` van `Bram` vervangen door:

```
public void run() {
    synchronized(c) { c.set(c.get()*4); }
}
```

Wat zijn nu de mogelijke uitkomsten van het programma?

g. *(2 pnt.)* In het oorspronkelijke programma wordt in de klasse `Worker` twee `public static` objecten `l1` en `l2` gedefinieerd.

```
public static Object o1 = new Object ();
public static Object o2 = new Object ();
```

De run methode van Alex wordt als volgt gewijzigd:

```
public void run()       {
    synchronized(o1) {
        synchronized(o2) {
            int v=c.get();
            c.set(v+4);       }
    }
```

De run methode van Bram wordt als volgt gewijzigd:

```
public void run()       {
    synchronized(o2) {
        synchronized(o1) {
            int v=c.get();
            c.set(v*4);
        }
```

Wat zijn nu de mogelijke uitkomsten van het programma?

h. *(3 pnt.)* In het oorspronkelijke programma wordt de methode run van de klasse Alex vervangen door:

```
public void run()  {
    synchronized(c) {
        if (c.get() == 1)
            try { c.wait(); }
            catch (InterruptedException e) {}
        c.set(c.get()+4);
    }
}
```

Tevens wordt de methode run van Bram vervangen door:

```
public void run()  {
    synchronized(c) {
        c.notifyAll();
        c.set(c.get()*4);
    }
}
```

Wat zijn nu de mogelijke uitkomsten van het programma?

Hint: bij het beantwoorden van deze vragen denk aan het probleem besproken op sheet 25 van hoorcollege 4, en het toestandsdiagram op sheet 39 van hetzelfde hoorcollege.