TENTAMEN

Programmeren 2

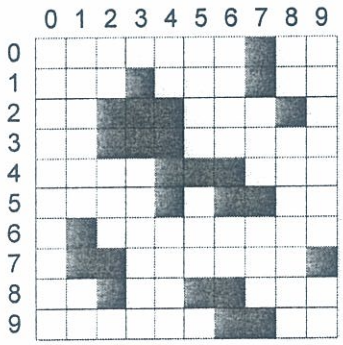
vakcode: 192135050
datum: 19 april 2012
tijd: 13:45 - 17:15

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 A), en van uitgeprinte kopieën van de hoorcollegesheets zonder aantekeningen.
- 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 minimale aantal punten per opgave bedraagt 0 punten. Het eindcijfer van het tentamen wordt bepaald door de optelsom van de punten per opgave.

Opgave 1 (20 punten)

Een *boolean-matrix* is een matrix met daarin uitsluitend de waarden `true` en `false`. In Figuur 1 staat een voorbeeld van een dergelijke boolean-matrix. De donkere hokjes representeren de waarde `true`, terwijl de witte hokjes de waarde `false` representeren.



Figuur 1: Grafische representatie van een boolean-matrix.

We zijn nu geïnteresseerd in een zogeheten *kluit* van donkere hokjes. Een *kluit* bestaat uit een donker hokje en alle donkere hokjes die vanuit dat hokje bereikt kunnen worden door omhoog, omlaag, naar links of naar rechts te gaan via alleen maar donkere hokjes. Zo zit het hokje op rij

2 en kolom 3 in een kluit van 13 hokjes. Hokje (6,1) zit in een kluit van 4 hokjes en hokje (7,3) zit niet in een kluit.

De onderstaande klasse `BMatrix` modelleert een boolean-matrix. Een `BMatrix`-object kan geconstrueerd worden middels een array van `String`-objecten, die de matrix specificeert. Voor het oplossen van deze opgave is de interne representatie van boolean-matrix in deze klasse het belangrijkste. De constructor is alleen gegeven voor de volledigheid.

```
public class BMatrix {
    private boolean[][] mat;

    /**
     * Construeert een BMatrix op grond van rs.
     * @require isGeldigeStringRepresentatie(rs)
     * @ensure  this.rows() == rs.length
     *          this.cols() == rs[0].length()
     *          voor alle 0 <= r < rows(), 0 <= c < cols():
     *              mat(r,c) == (rs[r].charAt(c) == 'x')
     */
    public BMatrix(String[] rs) {
        init(rs);
    }

    protected void init(String[] rs) {
        int rows = rs.length;
        int cols = rs[0].length();
        mat = new boolean[rows][cols];
        for (int i=0; i<rows; i++)
            for (int j=0; j<cols; j++)
                mat[i][j] = rs[i].charAt(j) == 'x' ;
    }

    /** @require 0 <= r < rows(), 0 <= c < cols() */
    public boolean  mat(int row, int col)    { return mat[row][col]; }
    public int      rows()                   { return mat.length;    }
    public int      cols()                   { return mat[0].length; }

    /**
     * Levert true op als rs een geldige String-representatie is voor een
     * BMatrix-object. rs is een geldige String-representatie als:
     *   rs != null &&
     *   voor alle s in rs geldt:
     *       s != null && s.length() == rs[0].length
     *       s bevat alleen de characters '.' and 'x'
     */
    public static boolean isGeldigeStringRepresentatie(String[] rs) {
        // implementatie is onbelangrijk
    }

    /** Levert een String-representatie van de matrix. */
    public String toString() {
        // implementatie is onbelangrijk
    }

    /**
     * Levert het aantal elementen in de kluit op positie (r,c).
     * @ensure  als r < 0 || r >= rows() ||
     *              c < 0 || c >= rows() ||
```

```
     *              !mat(r,c)
     *          dan    result == 0
     *          anders is result het aantal elementen van
     *                 de kluit waar mat(r,c) zich in bevindt
     */
    public int kluit(int r, int c) {
        // body nog toe te voegen
    }
}
```

Hieronder staat een voorbeeld van het gebruik van de methode `kluit` van de klasse `BMatrix`.

```
    public static void main(String[] args) {
        String[] test  = {".......x..","...x...x..","..xxx...x.","..xxx.....",
                          "....xxx...","....x.xx..",".x........", ".xx......x",
                                                   "..x..xx...","......xx.."};
        String[] bmstr = (args.length == 0 ? test : args);
        if (isGeldigeStringRepresentatie(bmstr)) {
            BMatrix m = new BMatrix(bmstr);
            System.out.print(m);
            System.out.println("kluit(2,3) = " + m.kluit(2,3));
            System.out.println("kluit(6,1) = " + m.kluit(6,1));
            System.out.println("kluit(7,3) = " + m.kluit(7,3));
            System.out.println("kluit(0,7) = " + m.kluit(0,7));
            System.out.println("kluit(7,9) = " + m.kluit(7,9));
        } else {
            System.out.println("Geen geldige string representatie");
        }
    }
```

Als we dit programma zonder parameters aanroepen, zal het programma de volgende uitvoer genereren:

```
.......x..
...x...x..
..xxx...x.
..xxx.....
....xxx...
....x.xx..
.x........
.xx......x
..x..xx...
......xx..
kluit(2,3) = 13
kluit(6,1) = 4
kluit(7,3) = 0
kluit(0,7) = 2
kluit(7,9) = 1
```

Gevraagd wordt nu om de body van de methode `kluit` van de klasse `BMatrix` te schrijven. Je wordt gevraagd om hierbij een recursieve oplossingsmethode te gebruiken.

Hint: De *kluit* van een donker hokje kan (makkelijk) bepaald worden door recursief de kluit van zijn buren te bepalen en daarbij bij te houden welke posities in de `BMatrix` al verwerkt zijn.

Opgave 2 (20 punten)

Beschouw de volgende subklassen van de klasse `Exception`.

```
class A extends Exception { A() { super("A"); } A(String s) { super(s); } }
class B extends A         { B() { super("B"); } B(String s) { super(s); } }
class C extends A         { C() { super("C"); } C(String s) { super(s); } }
class D extends B         { D() { super("D"); } D(String s) { super(s); } }
class E extends D         { E() { super("E"); } E(String s) { super(s); } }
```

Zij ook gegeven de volgende `interface EEE`:

```
interface EEE {
     public Exception doit(Exception e) throws Exception;
}
```

Deze `interface` definieert dus een methode `doit` die een `Exception`-object als parameter heeft, een `Exception`-object oplevert, maar ook nog een `Exception` kan gooien. Vandaar de naam `EEE`.

We definiëren nu een klasse `Fn1` als een implementatie van `EEE`, die alleen de methode `doit` implementeert.

```
class Fn1 implements EEE {
    public Exception doit(Exception e) throws Exception {
        try                 { throw e;                }
        catch (C c)         { throw doit(new B()); }
        catch (B b)         { try             { throw e;                }
                              catch (E ee)    { return new A();         }
                              catch (D dd)    { throw  new B();         }
                              catch (C cc)    { throw  b;               }
                              catch (B bb)    { return new C();         }
                              catch (A aa)    { return doit(new D());   }
                            }
        catch (Exception x) { return new E();       }
    }
}
```

Een andere implementie van `EEE` is de klasse `Fn2`, waarvan de methode `doit` de `doit` van een (ander) `EEE`-object twee keer aanroept.

```
class Fn2 implements EEE {
    private EEE other;
    public Fn2(EEE other) { this.other = other; }

    public Exception doit(Exception e) throws Exception {
        return other.doit(other.doit(e));
    }
}
```

De methoden `Fn1` en `Fn2` worden in de volgende klasse getest.

```
public class ExceptionOpgave {
    /**
     * Probeert eee.doit(e) uit te voeren.
     *  - Als er GEEN exceptie gegooid wordt door eee.doit(e) zal
     *      "r-" plus de String-representatie van de return Exceptie
     *      van eee.doit(e) opgeleverd worden.
     *  - Als er WEL een exceptie gegooid wordt door eee.doit(e)
     *      dan zal er "e-" plus de String-representatie van de
     *      gegooide Exceptie opgeleverd worden.
     */
    public static String tryit(EEE eee, Exception e) {
```

```
        String    prefix;
        Exception res;
        try                    { res = eee.doit(e) ; prefix = "r-"; }
        catch (Exception ee)   { res = ee          ; prefix = "e-"; }
        return (prefix + res.getMessage());
    }

    public static void main(String[] args) {
        Fn1 f1 = new Fn1();
        Fn2 f2 = new Fn2(f1);
        Exception[] a = { new A(), new B(), new C(), new D(), new E() };

        for (int i=0; i < a.length; i++)
            System.out.println(
                "Fn1 " + a[i].getMessage() + " : " + tryit(f1, a[i]) );

        for (int i=0; i < a.length; i++)
            System.out.println(
                "Fn2 " + a[i].getMessage() + " : " + tryit(f2, a[i]) );
    }
}
```

De methode `tryit` probeert dus gegeven een `EEE`-object `eee` en een `Exception`-object `e`, de methode `eee.doit(e)` aan te roepen. Zij opgemerkt dat de methode `getMessage` van de klasse `Exception` de `String` oplevert die bij de constructie van het `Exception`-object meegegeven is.

De opgave bestaat nu uit het bepalen van de uitvoer van de methode `main`. Geef voor elke aanroep van `tryit` in `main` precies aan wat de uitvoer zal zijn. Dit zijn in het totaal `2` maal `5` regels met eerst de naam van de `EEE`-klasse (d.w.z. `Fn1` of `Fn2`), dan de naam van de `Exception`-klasse, dan een dubbele punt, gevolgd door of `r-` of `e-`, en tenslotte een hoofdletter.

Als je meent dat er bij een bepaalde `tryit` géén uitvoer zal worden afgedrukt (als bijvoorbeeld een `StackOverflowException` gegooid wordt), dan moet je dit bij de desbetreffende regel als antwoord noteren. Van de overige `tryit`-aanroepen dien je dan nog wel de uitvoer te vermelden, alsof het programma niet voortijdig beëindigd was.

Hint: Om deze opgave op te lossen moet je het executiepad van het programma bijhouden. Bijvoorbeeld, voor `tryit (f1, a[0])` werkt het programma als volgt:

- aanroep `doit(a[0]`;
- gooi `a[0]` (van type `A`);
- gevangen als `Exception` want zowel `B` als `C` zijn specifieker dan `A`;
- enzovoorts.

Opgave 3 (20 punten)

Een kleine ondernemer wil zijn voorraad bijhouden en bewerken met behulp van een Java-programma. Per artikel wil hij alleen de *prijs* en het *aantal* bijhouden. Hieronder staat een eerste aanzet van een klasse `Voorraad` waarmee hij zijn artikelen wil opslaan.

```
public class Voorraad {
    private Map<String, Pair<Double,Integer>> artikelen;

    public Voorraad() {
        // vraag a.
    }

    /** Voegt een artikel toe aan deze Voorraad. */
```

```
    public void voegToe(String naam, double prijs, int aantal) {
        artikelen.put(naam, new Pair<Double,Integer>(prijs, aantal));
    }
}
```

De `artikelen` worden in een `Map` van `Strings` naar `Pair`-objecten opgeslagen. Een `Pair`-object kan referenties naar twee objecten bevatten, in dit geval een `Double` (voor de prijs) en een `Integer` (voor het aantal). Hieronder staat de klasse `Pair`.

```
public class Pair<X,Y> {
    public X first;
    public Y second;

    public Pair(X first, Y second) {
        this.first  = first;
        this.second = second;
    }
}
```

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. *(3 pnt.)* Geef de body van de constructor van de klasse `Voorraad`.

b. *(5 pnt.)* Definieer de volgende methode binnen de klasse `Voorraad`:

```
/** Levert het totaalbedrag van alle artikelen op voorraad. */
public double totaalInVoorraad()
```

c. *(9 pnt.)* Definieer de volgende methode binnen de klasse `Voorraad`:

```
/**
 * Levert een Map op die gegeven een Integer n (de key) een Set
 * (de value) oplevert met daarin alle artikelen (d.w.z. Strings)
 * waarvan er precies n exemplaren op voorraad zijn.
 */
public Map<Integer, Set<String>> aantallenMap()
```

d. *(3 pnt.)* Definieer de volgende methode binnen de klasse `Voorraad`:

```
/**
 * Levert een Set op met alle artikelen (Strings) waar er maar een
 * van op voorraad is. Levert null op als er geen enkel artikel is
 * waarvan er maar een op voorraad is.
 */
public Set<String> eenlingen()
```

Hieronder staat een voorbeeld van het gebruik van de klasse `Voorraad` en de bovenstaande methoden.

```
public static void main(String[] args) {
    Voorraad vv = new Voorraad();

    vv.voegToe("pen",       2.95, 3);
    vv.voegToe("potlood",   1.45, 5);
    vv.voegToe("plakband", 2.95, 1);
    vv.voegToe("schaar",    4.25, 1);
    vv.voegToe("stift",     3.10, 3);

    System.out.println("totaal:␣␣␣␣" + vv.totaalInVoorraad());
```

```
        System.out.println("aantallen:␣" + vv.aantallenMap());
        System.out.println("eenlingen:␣" + vv.eenlingen());
    }
```

Het voorbeeld zou de volgende uitvoer (kunnen) genereren:

```
totaal:    32.6
aantallen: {1=[plakband, schaar], 3=[pen, stift], 5=[potlood]}
eenlingen: [plakband, schaar]
```

Opgave 4 (20 punten)

Een enthousiaste Java programmeur wil de afstandbediening van zijn HD-televisie in Java modelleren. Hij is begonnen met het schrijven van een klasse `Kanalen` die de kanalen van de televisie bijhoudt en mogelijk maakt om naar andere kanalen te zappen. De klasse `Kanalen` ziet er als volgt uit:

```
public class Kanalen {

    private final String[]  kanalen;
    private int             huidig;

    public Kanalen(String[] kanalen) {
        this.kanalen = kanalen;
        this.huidig  = 0;
    }

    public void volgende() {
        huidig = (huidig+1) % kanalen.length;
    }

    public void willekeurig() {
        huidig = (int) (Math.random()*kanalen.length);
    }
    public String getHuidig() {
        return kanalen[huidig];
    }

}
```

a. (*4 pnt.*) Herschrijf de klasse `Kanalen` zodat deze als model gebruikt kan worden, volgens het Model-View-Controller patroon.

b. (*6 pnt.*) Geef een implementatie van een `KController`, die de interface `ActionListener` implementeert.

Tenslotte wordt er een klasse `KView` gedefinieerd, die de grafische elementen implementeert. De klasse `KView` is een `JFrame` met drie componenten. Er zijn twee `JButton`-objecten: het knopje met de tekst `">"` kan gebruikt worden om naar een volgend kanaal te *zappen* en het knopje met de tekst `"?"` kan gebruikt worden om naar een willekeurig kanaal te springen. Het `JTextField` laat een `String`-representatie van het huidige kanaal zien.

Figuur 2 toont een screenshot van de `KView` GUI.

Van een aantal methoden in de klasse `KView` wordt het hieronder alleen een declaratie gegeven. Het is de bedoeling dat hier een implementatie voor gedefinieerd wordt.

Figuur 2: Screenshot van de klasse `Remote` in actie.

```
public class KView extends JFrame implements Observer {
    public static final String NEXT   = "NEXT";
    public static final String RANDOM = "RANDOM";

    private JButton         btNext, btRandom;
    private JTextField      tfNaamKanaal;

    public KView(ActionListener controller) {
        super("KView");
        init(controller);
    }
    public void init(ActionListener controller) {
    // vraag c
    }
    public void update(Observable model, Object arg) {
    // vraag d
    }
    public static void main(String[] args) {
     // vraag e
    }
}
```

c. (*4 pnt.*) Implementeer de method `init`. Maak gebruik van een flow layout. Een geschikte grootte is bijvoorbeeld 260 × 70.

d. (*2 pnt.*) Implementeer de method `update`.

e. (*4 pnt.*) Implementeer de method `main`.

Opgave 5 (20 punten)

Bij deze opgave dien je 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.

Motiveer alle antwoorden!

Beschouw het Java-programma `MyThreads`.

```
public class MyThreads {
    public static void main(String[] args) {
        Value v1  = new Value(2);
        Value v2  = new Value(3);
        Thread t1 = new Task(v1, v2); t1.start();
        Thread t2 = new Task(v2, v1); t2.start();
```

```
        try { t1.join(); t2.join(); }
        catch (InterruptedException e) {}

        System.out.println(v1.get() + v2.get());
    }
}

class Value {
    private int x;

    public Value(int x) {
        this.x = x;
    }
    public synchronized void multiply(Value v) {
        x = x * v.get();
    }
    public int get() {
        return x;
    }
}

class Task extends Thread {
    private Value v1, v2;

    public Task(Value v1, Value v2) {
        this.v1 = v1;
        this.v2 = v2;
    }

    public void run() {
        v1.multiply(v2);
    }
}
```

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

b. *(3 pnt.)* In het oorspronkelijke programma wordt het gehele `try-catch`-blok 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 uit de methode `main` verwijderd. Daarnaast worden ook de aanroepen van `t1.start()` en `t2.start()` in de methode `main` veranderd in respectievelijk `t1.run()` en `t2.run()`. Wat zijn nu de mogelijke uitkomsten van het programma?

d. *(2 pnt.)* In het oorspronkelijke programma wordt de methode `multiply` van de klasse `Value` *niet* langer als `synchronized` methode gedefinieerd. Wat zijn nu de mogelijke uitkomsten van het programma?

e. *(2 pnt.)* In het oorspronkelijke programma wordt de methode `get` van de klasse `Value` als `synchronized` methode gedefinieerd. Wat zijn nu de mogelijke uitkomsten van het programma?

f. *(2 pnt.)* In het oorspronkelijke programma wordt de methoden `run` van de klasse `Task` als `synchronized` methode gedefinieerd. Wat zijn nu de mogelijke uitkomsten van het programma?

g. *(3 pnt.)* In het oorspronkelijke programma wordt de methode `multiply` van de klasse `Value` als volgt veranderd:

```
public synchronized void multiply(Value v) {
    x = x * v.get();
    v.notifyAll();
}
```

Wat zijn nu de mogelijke uitkomsten van het programma?

h. *(3 pnt.)* In het oorspronkelijke programma wordt de methode `multiply` als volgt veranderd:

```
public synchronized void multiply(Value v) {
    if (x == 2)
        try { this.wait(); }
        catch (InterruptedException e) {}
    x = x + v.get();
    synchronized (v) { v.notify(); }
}
```

Wat zijn nu de mogelijke uitkomsten van het programma?

Hint: Controleer de werking van `synchronized` op verschillende ojecten (bijvoorbeeld, op `v1` en `v2`) en de werking van `obj.wait()`, `obj.notify()` en `obj.notifyAll()` in combinatie met `synchronized (obj)`.

ERRATA

Opgave 5, vraag h

h. *(3 pnt.)* In het oorspronkelijke programma wordt de methode `multiply` als volgt veranderd:

```
public synchronized void multiply(Value v) {
    if (x == 2)
        try { this.wait(); }
        catch (InterruptedException e) {}
    x = x * v.get();
    synchronized (v) { v.notify(); }
}
```

Wat zijn nu de mogelijke uitkomsten van het programma?