Opmerking. Het originele examen had een enigszins andere bladschikking. Antwoorden op de vragen kwamen op afzonderlijke antwoordbladen die we hier niet in de PDF hebben opgenomen.

Bij heel wat vragen moet je zelf Java-code schrijven. Hou dit kort en bondig. Je schrijft *geen* commentaarregels en *geen* import-opdrachten, tenzij gevraagd. Programmeer 'helder' en 'met stijl' en volg zoveel mogelijk de gangbare conventies, in het bijzonder bij de keuze van de juiste lus: while, for of for each. Schrijf broncode met een duidelijk leesbare structuur, o.a. door correct gebruik van indentatie (witruimte vóór elke regel).

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.

Verwar niet tussen 'tabel' (array) en 'lijst' (*ArrayList* of *List*).

Opgave 1. (1 pt – 9 ln[1]) (Opwarmertje) Schrijf een methode *maxindex*(*tabel*) die van een tabel (array) van kommagetallen teruggeeft (retourneert) wat de *index* (positie) is van het grootste getal in die tabel.

Voor de eenvoud mag je ervan uitgaan dat de tabel minstens 1 element bevat en dat alle elementen in de tabel verschillend zijn. Het is belangrijk dat je in je oplossing het correcte lustype (while, for, for each) gebruikt.

Oplossing:

```
public int maxindex(double[] tabel) {
    int index = 0;
    for (int i = 1; i < tabel.length; i++) {
        if (tabel[i] > tabel[index]) {
            index = i;
        }
    }
    return index;
}
```

Ook al moet je elk element in de tabel bekijken, is een for each hier niet aangewezen, want je hebt de index nodig van de elementen.

Als alternatief kan je ook de waarde van *tabel*[*index*] in een lokale variabele bijhouden. (Dat vraagt 2 bijkomende lijnen.)

Opgave 2. (2 pt – 7 ln)

Het getal π kan je berekenen met behulp van de volgende formule (François Viète, 1593) :

$$\frac{2}{\pi} = \frac{\sqrt{2}}{2} \cdot \frac{\sqrt{2+\sqrt{2}}}{2} \cdot \frac{\sqrt{2+\sqrt{2+\sqrt{2}}}}{2} \cdot \frac{\sqrt{2+\sqrt{2+\sqrt{2+\sqrt{2}}}}}{2} \cdots$$

Schrijf een programmafragment dat de waarde van π (benaderd) berekent uit de eerste 10 factoren van dit oneindig product. Stop het resultaat in een variabele *pi*. Let op dat je de waarden onder de worteltekens niet telkens helemaal opnieuw berekent. Gebruik de klassenmethode *sqrt* uit *Math* om een vierkantswortel te berekenen.

Oplossing:

```
double product = 1.0;
double wortel = 0;
for (int i=0; i < 10; i++) {
   wortel = Math.sqrt(2.0 + wortel);
   product = product * wortel / 2.0;
}
double pi = 2.0 / product;
```

Opgave 3. (3 pt – 5 ln) Schrijf een programmafragment dat alle drietallen x, y, z van positieve gehele getallen afdrukt met de eigenschap dat $x+y+z=2020$ en $0<x<y<z$. De uitvoer ziet er dus als volgt uit

```
1, 2, 2017
1, 3, 2016
...
1, 1009, 1010
2, 3, 2015
...
672, 673, 675
```

(eventueel in een andere volgorde)

Het is de bedoeling dat je deze opgave oplost *zonder* een if-opdracht te gebruiken. (Hiervoor moet je bijzondere aandacht besteden aan de grenzen van de lussen die je gebruikt.)

Vind je niet hoe dit moet, dan mag je een oplossing geven die wel een if-opdracht gebruikt, maar dan kan je maximaal 2 punten behalen op deze vraag. Probeer hoe dan ook het aantal condities dat je controleert tot een minimum te beperken.

Oplossing:

```
for (int x = 1; x <= 672, x++) {
   for (int y = x+1; y < 2020 - x - y; y++) {
      System.out.println (x + ", " + y + ", " + 2020 - x - y);
   }
}
```

De conditie `y < 2020 - x - y` kan je ook schrijven als `2*y + x < 2020` of als `y + x/2 < 1010`.

Opgave 4. (1 pt) In de constructor van de klasse *Rechthoek* willen we de oppervlakte van een rechthoek bewaren zodat we ze later nog kunnen opvragen met een getter.

```
public class Rechthoek {

    public Rechthoek (double hoogte, double breedte) {
        ...
        opp = hoogte * breedte;
    }

    public double getOpp () {
        return opp;
    }

}
```

Welk soort variabele gebruiken we best voor *opp*?

1. Een lokale variabele — **double** *opp*
2. Een instantievariabele — **private double** *opp*
3. Een klassenvariabele — **private static double** *opp*

Dit is een meerkeuzevraag. Precies één van de drie mogelijke antwoorden is correct. Je krijgt 1 punt bij een correct antwoord, 0 punten bij een fout antwoord of als het antwoord ontbreekt.

Oplossing: 2, een instantievariabele.

Opgave 5. (2 pt – 5 ln) Geef de definitie van een nieuwe uitzonderingsklasse *MyException* die we op de volgende manier kunnen gebruiken in een programma:

```
if (...) {
    throw new MyException ("Er deed zich een onverwachte fout voor");
}
```

Het resultaat moet een *ongecontroleerde* uitzondering zijn (*unchecked exception*).

Oplossing:

```
public class MyException extends RuntimeException {

    public MyException(String message) {
        super(message);
    }

}
```

Opgave 6. (3 pt – 18 ln) Schrijf een methode *woorden(lijn)* die een lijst teruggeeft van alle *woorden* op de gegeven lijn, in dezelfde volgorde als waarin ze op die lijn voorkomen. (En komt hetzelfde woord meer dan één keer voor in die lijn, dan moet het ook meer dan één keer voorkomen in de resulterende lijst.)

De parameter *lijn* is van het type *String*. Met *woord* bedoelen we een zo lang mogelijke opeenvolging van *letters*. (Dus "Dit is een zin." en "1a23bc4d5ef6" bevatten allebei 4 woorden.) Gebruik *Character.isLetter*(..) om na te gaan of een teken een letter is.

Los dit op door de lettertekens in de string *lijn* maar één keer van voor naar achter te doorlopen.

Oplossing:

```
public List<String> woorden (String lijn) {
    List<String> result = new ArrayList<> ();
    boolean inWoord = false;
    int pos = 0;
    for (int i=0; i < lijn.length(); i++) {
       if (inWoord && !Character.isLetter(lijn.charAt(i)) {
           result.add (lijn.substring(pos, i);
           inWoord = false;
       } else if (!inWoord && Character.isLetter(lijn.charAt(i)) {
           pos = i;
           inWoord = true;
       }
    }
    if (inWoord) {
       result.add (lijn.substring(pos));
    }
    return result;
}
```

---

7. Volgend fragment is een onderdeel van een groter programma. Hierbij stelt *list* een lijst voor van gehele getallen.

```
if (list != null) {
   int eerste = list .get(0);
   ...
}
```

Opgave 7A. (1 pt) Het programma blijkt echter een fout te bevatten: soms gebeurt het dat tijdens het uitvoeren van de tweede lijn hierboven een *IndexOutOfBoundsException* wordt opgegooid. Wat zou hiervan de oorzaak kunnen zijn?

Oplossing: Als de lijst leeg is (= geen enkel element bevat). (Een lege lijst is niet hetzelfde als een (verwijzing naar een) lijst die `null` is!)

Opgave 7B. (1 pt) (Detailvraagje) En ook kan het gebeuren dat er tijdens het uitvoeren van diezelfde lijn in de plaats een *NullPointerException* wordt opgegooid. Wat zou daarvan de oorzaak kunnen zijn?

Oplossing: Als het eerste element van de lijst (het element met index 0) de waarde `null` heeft. (De elementen van een lijst van gehele getallen zijn van het type `Integer`, niet `int`. De waarde `null` is toegelaten voor een `Integer`.)

Opgave 8. (4 pt – 19 ln) Van twee torens op een schaakbord wordt gezegd dat ze elkaar *slaan* als ze zich op het bord in dezelfde rij of in dezelfde kolom bevinden. We willen een aantal torens op een schaakbord plaatsen, zonder dat ze elkaar slaan.

Ontwerp hiervoor een klasse *Torenwachter* met de volgende methodes:

- Een methode *plaatsToren*(*r*,*k*) die een toren plaatst op rij *r* en kolom *k* van het schaakbord. (Rij- en kolomnummers lopen van 0 t.e.m. 7.) Je mag aannemen dat deze methode nooit zal gebruikt worden om een toren te plaatsen op een plaats waar er reeds eerder een toren is neergezet — je hoeft dit dus ook niet te controleren.

- Een methode *isGeldig*() die teruggeeft of het bord nog steeds *geldig* is. We noemen een bord geldig wanneer er geen twee verschillende torens op het bord staan die elkaar slaan.

De klasse *Torenwachter* zal op de volgende manier gebruikt worden: eerst wordt er een object van die klasse aangemaakt, daarna wordt de methode *plaatsToren* van dit object een aantal keer opgeroepen en tot slot wordt de methode *isGeldig* van ditzelfde object opgeroepen om te kijken of de torens elkaar niet slaan.

Belangrijke opmerkingen

- Laat een *Torenwachter* enkel van elke rij en kolom op het schaakbord bijhouden *hoeveel* torens ze bevatten — waar ze zich precies bevinden is niet belangrijk.

  Je mag dus *niet* bijhouden wat de coördinaten zijn van de torens die telkens worden aangeboden door *plaatsToren*. Je mag geen lijst of tabel van coördinatenparen opslaan en ook geen tweedimensionale tabel waarmee het schaakbord wordt voorgesteld. Je mag ook geen andere klassen definiëren om in *Torenwachter* te gebruiken.

- Zorg dat *isGeldig* efficiënt is: zodra je doorhebt dat één toren een andere toren slaat, hoef je niet meer verder te controleren of er misschien nog een ander paar torens is dat elkaar slaat.

Oplossing:

```
public class TorenWachter {

    private int[] horizontaal;
    private int[] verticaal;

    public TorenWachter () {
       horizontaal = new int[8]; // eventueel opvullen met nullen
       verticaal = new int[8];   // idem
    }

    public void plaatsToren(int r, int k) {
        verticaal[k]++;
        horizontaal[r]++;
    }

    private boolean isGeldig () {
        int i = 0;
        while (i < 8 && horizontaal[i] <= 1 && verticaal[i] <= 1) {
            i++;
        }
        return i == 8;
    }
}
```

Alternatieve oplossing: Je kan reeds bij het plaatsen van de toren controleren of het bord nog geldig is en dit onthouden.

```
public class TorenWachter {

    private int[] horizontaal;
    private int[] verticaal;
    private boolean geldig;

    public TorenWachter () {
       horizontaal = new int[8];
       verticaal = new int[8];
       geldig = true;
    }

    public void plaatsToren(int r, int k) {
        verticaal[k]++;
        if (verticaal[k] > 1) {
            geldig = false;
        }
        horizontaal[r]++;
        if (horizontaal[r] > 1) {
            geldig = false;
        }
    }

    private boolean isGeldig () {
        return geldig;
    }
}
```

Opgave 9. (4 pt – 28 ln) De Java-bibliotheek bevat een interface *Icon* met de volgende definitie:

> **public interface** *Icon* {
>
> **int** *getIconHeight*(); // (vaste) hoogte
> **int** *getIconWidth* (); // (vaste) breedte
>
> **void** *paintIcon*(*Component c*, *Graphics g*, **int** *x*, **int** *y*);
> // beeldt de afbeelding af op de gegeven (x,y)-positie
>
> }

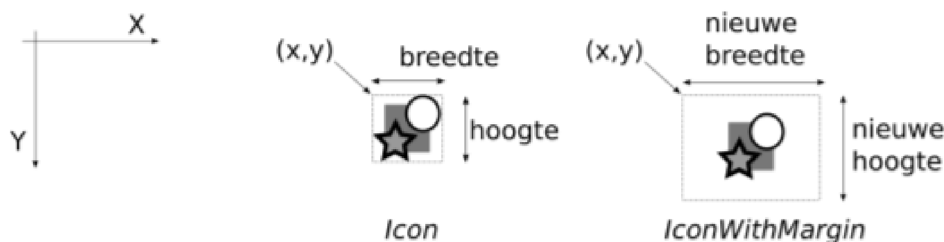
(Geen paniek: deze interface behoort inderdaad niet tot de geziene leerstof.)

Een object van het type *Icon* stelt een 'afbeelding' voor (afhankelijk van de implementatie kan dit een lijntekening zijn, een foto, een logo, ...). Je kan de dimensies (hoogte en breedte) van een afbeelding opvragen met *getIconHeight* en *getIconWidth* en je kan de afbeelding tekenen met behulp van *paintIcon*. (Een afbeelding wordt geplaatst op een bepaalde component *c* en maakt gebruik van een bepaalde grafische context *g*. Wat de exacte betekenis van deze begrippen is, is voor deze opgave echter niet van belang.)

We willen nu de functionaliteit van *Icon* wijzigen, zodat je ook een bijkomende marge kunt opgeven. Dit doen we in een nieuwe klasse *IconWithMargin*.

Opgave. Schrijf de volledige broncode van de klasse *IconWithMargin*.

- Je moet objecten van deze klasse overal kunnen gebruiken waar je een 'gewoon' icon kan gebruiken.
- De constructor van *IconWithMargin* neemt een *Icon* als argument die als basisafbeelding zal dienen.
- Oorspronkelijk heeft een *IconWithMargin* dezelfde afmetingen als de basisafbeelding — er wordt geen marge toegepast.
- Je moet de marge kunnen aanpassen door een nieuwe breedte en hoogte voor de afbeelding in te stellen met behulp van methoden *setIconHeight* en *setIconWidth*. (Merk op dat horizontale en verticale marge dus kunnen verschillen.)
- Het afbeelden van een *IconWithMargin* heeft hetzelfde effect als bij de basisafbeelding, behalve de positie waarop de afbeelding wordt getoond — ze wordt verschoven over de ingestelde marge, zoals geschetst in onderstaande figuur. De basisafbeelding bevindt zich precies in het midden van de rechthoek met de nieuwe afmetingen.



Je hoeft niet speciaal rekening te houden met het bijzondere geval waarin een nieuwe afmeting kleiner is dan de originele afmeting, m.a.w., de marge zal nooit negatief zijn.

Oplossing:

```
public class IconWithMargin implements Icon {

    private Icon icon;

    private int width;
    private int height;

    public IconWithMargin (Icon icon) {
        this.icon = icon;
        this.width = icon.getIconWidth();
        this.height = icon.getIconHeight();
    }

    public int getIconHeight() {
        return height;
    }

    public int getIconWidth() {
        return width;
    }
```

```java
    public void setIconHeight(int height) {
        this.height = height;
    }

    public void setIconWidth(int width) {
        this.width = width;
    }

    public void paintIcon(Component c, Graphics g, int x, int y) {
        icon.paintIcon (c, g,
              x + (width - icon.getIconWidth()) / 2,
              y + (height - icon.getIconHeight()) / 2
         );
    }

}
```

In de plaats van de nieuwe breedte en hoogte bij te houden, kan je ook de marges bijhouden, t.t.z., de halve verschillen tussen de nieuwe en oude breedte of hoogte.