

---

## Arrays van objecten

---

### Doelstellingen

Na dit hoofdstuk

- weet je dat arrays in Java ook objecten zijn
- kan je de geheugenlayout van objectarrays uittekenen
- ken je de `ArrayList` en kan je ze gebruiken
- weet je wat een interface is
- kan je de interface `Comparable` implementeren
- ken je de verschillende methoden om arrays en arraylists te sorteren

## 4.1 Inleiding

Rijen of arrays zijn frequent gebruikte constructies in het programmeren. In de cursus Java Basisconcepten hebben we ons vooral beziggehouden met arrays van primitieve types, zoals `int[]` en `double[]`, maar **je kan in Java van elk type een array maken**. Ook van je eigen objecttypes dus, die zoals we reeds weten datatypes zijn zoals een ander. Maar er is meer. Java voorziet naast de gewone array nog een aantal manieren om met verzamelingen van objecten te werken, de zogenaamde **Collections**. Eén van die collections is de **ArrayList** die wat later in dit hoofdstuk ook aan bod komt.

## 4.2 Arrays van objecten

**Arrays in Java zijn ook objecten.** Dat betekent dat een variabele van een array type een referentie bevat, een referentie die wijst naar het array object waar effectief de verschillende elementen uit de array in bewaard worden. Nemen we er onze klasse `Punt` terug even bij:

```
// Punt.java
public class Punt {
    private int x;
    private int y;

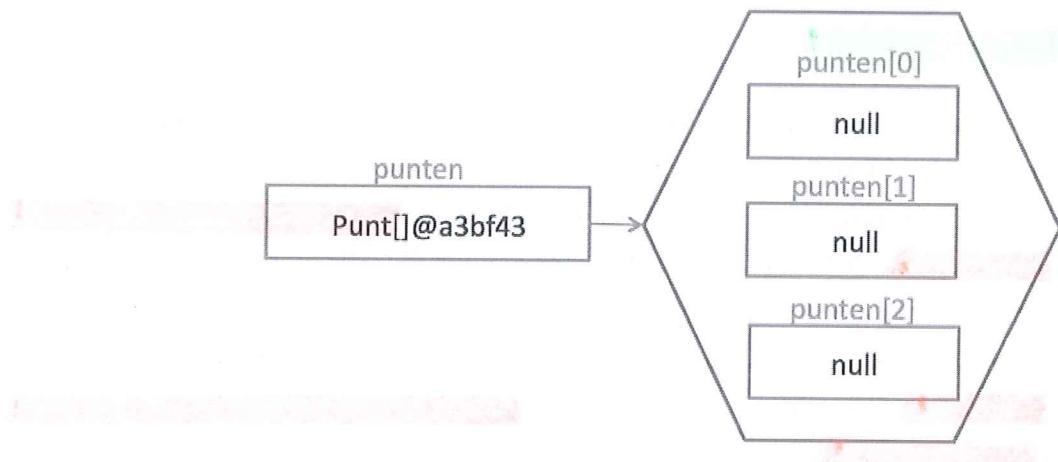
    public Punt(int x, int y) {
        this.x = x;
        this.y = y;
    }

    ...
}
```

We kunnen nu als volgt **een array maken waar we punten in kunnen bewaren**. Let op het gebruik van `new` om de array aan te maken.

```
Punt[] punten = new Punt[3];
```

De array `punten` bestaat nu en voorziet plaats voor 3 referenties naar `Punt` objecten, maar die referenties zijn nog niet geïnitialiseerd. Eigenlijk is dit een analoge situatie als bij arrays van een primitief type, waar we de arrays ook nog moeten initialiseren, **bijvoorbeeld met een for-lus**. De layout van het geheugen kan je zien in figuur 4.1.

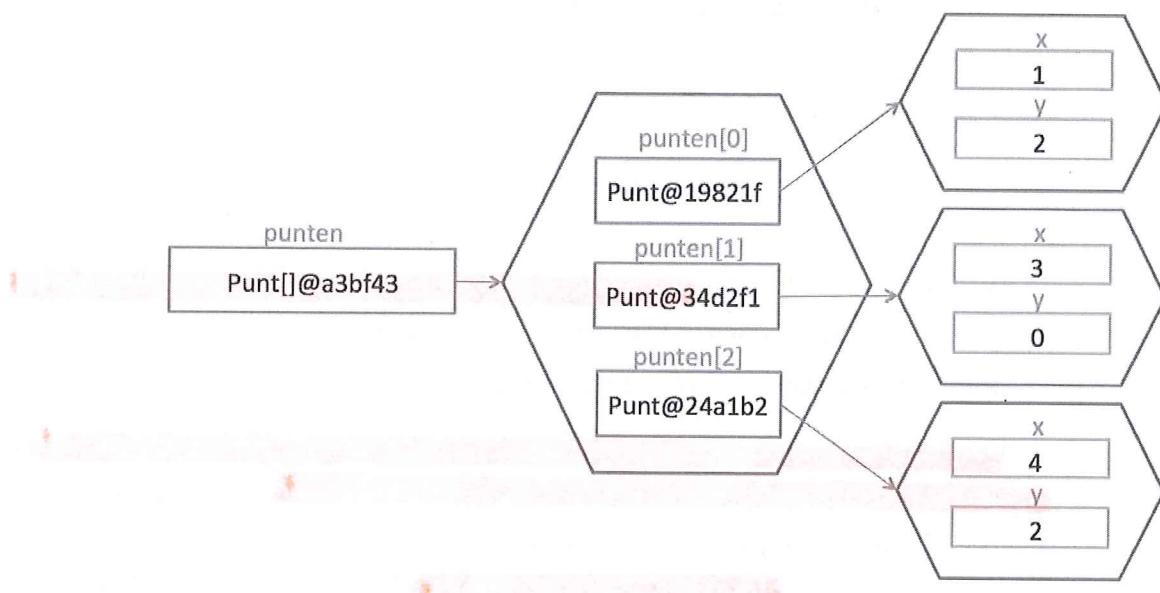


Figuur 4.1: Aangemaakt, maar nog niet geïnitialiseerd.

Om de array te initialiseren moet elk van de 3 Punt objecten met de constructor aangemaakt worden:

```
punten[0] = new Punt(1, 2);
punten[1] = new Punt(3, 0);
punten[2] = new Punt(4, 2);
```

De geheugenlayout van de array kan je zien in figuur 4.2. Zoals je kunt zien bevat de variabele punten een referentie naar een Punt [] object, en is elk element uit de array (punten[0], punten[1], ...) op zijn beurt een Punt object. Het is belangrijk dat je het type punt-array ook als een objecttype ziet.



Figuur 4.2: Een array van Punt objecten.

Stel dat we nu bij het eerste punt uit de array alle volgende punten willen optellen, dan kunnen we het volgende schrijven. Met de lus lopen we door de punten, en elk Punt object `punten[i]` tellen we bij het eerste punt `punten[0]` op d.m.v. de `telBij` methode. Merk op dat we dus bij elk object uit de array alle methoden uit de Punt klasse kunnen toepassen.

```
for (int i = 1; i < punten.length; i++) {  
    punten[0].telBij(punten[i]);  
}
```

## 4.3 ArrayList

Een collection is simpelweg een object waarin een aantal elementen als één geheel gegroepeerd worden. Collecties worden gebruikt om data te bewaren, op te halen, te manipuleren en te communiceren. Typisch bevatten ze objecten die een natuurlijke groep vormen zoals een collectie van kaarten, een mail folder (een collectie van mails) of een telefoonboek (namen en telefoonnummers). Er bestaan heel wat soorten collecties, elk met hun eigen wijze waarop de data wordt georganiseerd en beheerd. Eén van de meest eenvoudige collecties is de ArrayList.

Een ArrayList kan je bekijken als een dynamische array van objecten. Met dynamisch bedoelen we dat, in tegenstelling tot de klassieke arrays waarbij je de grootte vooraf moet vastleggen, de ArrayList automatisch groeit. De ArrayList is volledig geschreven in termen van het type Object wat wil zeggen dat alle mogelijke objecttypes in aanmerking komen, je hoeft dus ook niet vooraf te bepalen wat het type van de elementen zal zijn. De klasse ArrayList maakt deel uit van het java.util package.

Om objecten aan een ArrayList toe te voegen wordt de add methode gebruikt. Merk op dat er aan de constructor van ArrayList geen grootte noch type moet meegegeven worden.

```
ArrayList a = new ArrayList();
a.add(new Punt(1, 2));
a.add(new Punt(3, 0));
a.add(new Punt(4, 2));
```

Het is belangrijk om te beseffen dat er onder de motorkap in de klasse ArrayList met een gewone array wordt gewerkt. Dat betekent dat er ook sprake is van een index. Het totaal aantal objecten in de lijst kan je opvragen met de size methode, het opvragen van een object doe je met get waaraan de index van het gewenste object wordt meegegeven. Wanneer we objecten uit een ArrayList halen moeten die terug gedowncast worden naar het oorspronkelijk type. Herinner je dat het (up)casten naar Object, wat gebeurt bij het toevoegen van objecten, impliciet kan gebeuren, maar dat een (down)cast steeds expliciet moet gebeuren. De code om alle punten bij het eerste op te tellen kan er dan als volgt uitzien:

```
Punt eerstePunt = (Punt)a.get(0);
for (int i = 1; i < a.size(); i++) {
    eerstePunt.telBij((Punt)a.get(i));
}
```

Andere nuttige methoden die je op een ArrayList kunt toepassen zijn clear

(de lijst leegmaken), **contains** (testen of een gegeven object in de lijst voorkomt), **isEmpty** (testen of de lijst leeg is) en **remove** (verwijderen van het object op een gegeven positie). **Wanneer een object uit de lijst wordt verwijderd, schuiven alle objecten die er na komen één positie op - er gaan dus nooit gaten vallen in de indexen.**

```
System.out.println(a.size()); // 3 A, B, C
a.remove(1);
System.out.println(a.size()); // 2 A, C
```

## 4.4 Sorteren

Er bestaan heel wat algoritmen om rijen te sorteren: bubble sort, selection sort, quick sort, etc. Hoewel het interessant is om een idee te hebben van hoe die sorteringsalgoritmen werken, is het geen goed idee om die zelf te gaan implementeren. De Java bibliotheek voorziet verschillende sorteringsmethoden, en je zult al heel erg je best moeten doen om die nog te verbeteren.

Voor **primitieve arrays** kan je gebruik maken van de statische **Arrays.sort** methode uit het `java.util` package. Zoals je weet worden argumenten in Java by value meegegeven, wat wil zeggen dat een kopie van de variabele aan de methode wordt meegegeven. In het geval van arrays wil dat zeggen dat de referentie wordt gekopieerd, maar die blijft natuurlijk wel wijzen naar hetzelfde array object als waar de oorspronkelijke referentie naar wijst. Dat is de reden waarom volgende code effectief de oorspronkelijke array **getallen** kan sorteren.

```
int[] getallen = { 3, 5, 1, 2, 6, 4 };

Arrays.sort(getallen);

for (int i = 0; i < getallen.length; i++) {
    System.out.print(getallen[i] + "\t");
}
```

In het geval van objecten is er wel een catch. Primitieve waarden hebben een natuurlijke volgorde, d.m.v. de relationele operatoren kan bv. van twee getallen bepaald worden welke van de twee het kleinste is. Bij complexe types is die volgorde natuurlijk niet bepaald, hier zullen we **code moeten schrijven die kan gebruikt worden om uit te maken welk object 'kleiner' is**. We moeten natuurlijk wel vermijden dat iedereen een eigen manier van vergelijken gaat implementeren in z'n klassen. Van **Arrays.sort** bestaat er een versie om arrays van objecten mee te sorteren (`Object[]`), maar hoe kan die methode op een uniforme en eenduidige manier weten

hoe de objecten in kwestie gesorteerd moeten worden? Het antwoord is **de interface Comparable**.

Een interface kan je zien als **een soort lege klasse**: ze bevat enkel methodesignaturen maar geen code. **Een interface legt m.a.w. de vorm vast, maar niet de inhoud.** De interface die gebruikt wordt in ons sorteringsvraagstuk is Comparable (uit `java.lang`). Deze interface bevat één methode signatuur en ziet er uit als volgt. Merk op dat het **interface keyword** op hetzelfde niveau staat als **class**.

```
interface Comparable {
    int compareTo(Object o);
}
```

**Klassen kunnen interfaces implementeren.** Waar je met `extends` kunt aangeven dan een klasse van een andere is afgeleid, kan je met **implements** aangeven dat een klasse een interface implementeert.

```
// Punt.java
public class Punt implements Comparable {
    ...
}
```

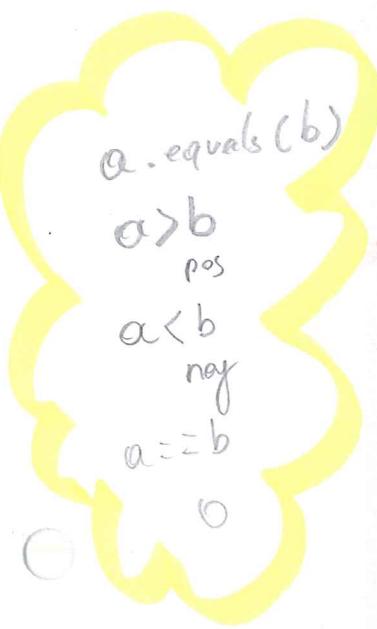
Een interface implementeren kan je bekijken als een contract tekenen: **je verbindt jezelf er toe om de methoden die in de interface gedefinieerd staan te implementeren.** Wanneer je dit niet doet krijg je een foutmelding van de compiler. De interface Comparable bevat maar één methode, nl. `compareTo`, dus die methode zal in onze klasse Punt geïmplementeerd moeten worden. Maar **hoe doen we dat?**

In de Java API lezen we het volgende over `compareTo`:

*Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.*

Anders gezegd: het oproepende object moet vergeleken worden met het object dat als parameter wordt doorgegeven, **het is aan ons als programmeur om te bepalen welk van die twee objecten kleiner is, groter of gelijk.** In het geval van onze punten zouden we bv. voor een sortering in wijzerzin kunnen kiezen: **een punt met een grotere x is sowieso groter, en bij gelijke x wordt naar de y gekeken.** In code vertaalt zich dat naar het volgende. Merk op dat het begin vrij gelijklopend is als bij de implementatie van `equals`. Wanneer het parameter object geen instantie is van Punt wordt een `ClassCastException` object opgegooid.

```
// Punt.java
public class Punt implements Comparable {
```



```

public int compareTo(Object obj) {
    if (!(obj instanceof Punt)) {
        throw new ClassCastException();
    }
    Punt arg = (Punt)obj;

    if (this.x < arg.x) {
        return -1;
    }
    else if (this.x > arg.x) {
        return 1;
    }
    else {
        if (this.y < arg.y) {
            return -1;
        }
        else if (this.y > arg.y) {
            return 1;
        }
    }
    return 0;
}

```

Elke klasse die de interface Comparable implementeert kan met `Arrays.sort` gesorteerd worden. Hetzelfde gaat op voor `ArrayList`, alleen wordt dan met een andere statische methode gewerkt: `Collections.sort`.

```

ArrayList a = new ArrayList();
a.add(new Punt(4, 1));
a.add(new Punt(4, 4));
a.add(new Punt(1, 1));
a.add(new Punt(1, 4));

Collections.sort(a);

for (int i = 0; i < a.size(); i++) {
    System.out.println((Punt)a.get(i));
}

```

## 4.5 Probeer het uit!

Voor deze *Probeer het uit!* gaan we een paar klassen ontwerpen waarmee we een highscore kunnen beheren. Een highscore bestaat uit een geordende lijst van

namen en scores. Enkel de top 5 wordt weergegeven. We willen natuurlijk ook dat de highscore persistent blijft over uitvoeringen van het spel heen, we gaan m.a.w. de scores in een tekstbestand moeten wegschrijven.

Laat ons eerst de grote lijnen uitzetten. Een highscore is een geordende lijst van naam-score paren. Voor een naam-score paar gaan we een complex type maken, we noemen dit bv. `HighscoreItem`. En aangezien we die items moeten kunnen sorteren, laten we die klasse de `Comparable` implementeren:

```
// HighscoreItem.java
public class HighscoreItem implements Comparable {
    private String naam;
    private int score;

    ...
}
```

De highscore zelf is een rij van `HighscoreItem` objecten. We kunnen hiervoor een `HighscoreItem[]` of een `ArrayList` gebruiken, maar die laatste lijkt de meest flexibele optie. Via `Collections.sort` gaan we de lijst kunnen sorteren vermits de `HighscoreItem` objecten `Comparable` implementeren.

```
// Highscore.java
public class Highscore {
    private ArrayList scores;
}
```

#### 4.5.1 HighscoreItem

We beginnen met de functionaliteit voor de `HighscoreItem` klasse. We gaan zeker een constructor nodig hebben, accessoren voor de velden lijken ook nuttig en als we de highscore willen kunnen afprinten zal een `toString` methode ook van pas komen. De code zelf is duidelijk.

```
// HighscoreItem.java
public class HighscoreItem implements Comparable {
    private String naam;
    private int score;

    public HighscoreItem(String naam, int score) {
        this.naam = naam;
        this.score = score;
    }

    public String getNaam() {
        return naam;
    }
}
```

```

public int getScore() {
    return score;
}

public String toString() {
    return naam + " " + score;
}
...
}

```

Rest ons de implementatie van `compareTo`, waartoe we ons verplicht hebben door het contract dat de `Comparable` interface ons oplegt. De **ordening lijkt duidelijk**: eerst kijken we naar de score, en bij gelijke score naar de naam. Merk wel op dat een highscore typisch omgekeerd wordt gesorteerd, t.t.z. de grootste score komt eerst. Daarmee dat een `HighscoreItem` kleiner zal zijn dan een ander als de score groter is, `Collections.sort` sorteert immers altijd van klein naar groot. De implementatie is vrij gelijklopend met wat we eerder in dit hoofdstuk hebben gedaan.

```

public int compareTo(Object obj) {
    if (!(obj instanceof HighscoreItem)) {
        throw new ClassCastException();
    }

    HighscoreItem item = (HighscoreItem) obj;

    if (this.score > item.score) {
        return -1;
    }
    else if (this.score < item.score) {
        return 1;
    }

    return this.naam.compareTo(item.naam);
}

```

### 4.5.2 Highscore

Het is de bedoeling dat we scores kunnen wegschrijven naar en ophalen uit een tekstbestand. Voor het lezen en schrijven van tekstbestandjes gebruiken we de `TextFile` klasse uit de Code Library (**zie appendix A**). Het tekstbestandje zou er als volgt uit kunnen zien - op elke lijn een naam-score paar dus.

```

Hans 23
Kristien 19
Davy 15

```

Luk 13  
Frank 9

Voor het inlezen voorziet TextFile een **methode readLines**. Elke ingelezen string bevat een naam en een score, m.b.v. split kunnen we die string op de spatie splitsen. Met die data worden HighscoreItem objecten aangemaakt en toegevoegd aan de ArrayList. Merk op dat we de methode private hebben gemaakt aangezien dit tot de interne keuken van onze Highscore klasse behoort.

```
// Highscore.java
public class Highscore {
    private static final String HIGHSCORE_FILE
        = "highscores.txt";
    private ArrayList scores;

    private void load() {
        String[] in = null;
        try {
            in = TextFile.readLines(HIGHSCORE_FILE);
        }
        catch (IOException io) {
            return;
        }

        for (int i = 0; i < in.length; i++) {
            String[] values = in[i].split(" ");
            scores.add(new HighscoreItem(values[0],
                Integer.parseInt(values[1])));
        }

        Collections.sort(scores);
    }
}
```

Om de highscore te bewaren moet de lijst naar een string omgezet worden. Dit is eigenlijk de functionaliteit die we in een **toString** methode kunnen vastleggen. In plaats van alle scores weg te schrijven, beperken we ons tot bv. de 5 hoogste scores. Let er op hoe de **toString** van Highscore gebruik maakt van de **toString** uit HighscoreItem die we daarnet hebben geschreven.

```
// Highscore.java
public class Highscore {
    private static final int NUM_SCORES = 5;
    private static final String HIGHSCORE_FILE
        = "highscores.txt";
    private ArrayList scores;
```

```

public String toString() {
    int numScores = (scores.size() > NUM_SCORES ?
        NUM_SCORES : scores.size());
    String output = "";

    for (int i = 0; i < numScores; i++) {
        HighscoreItem item =
            (HighscoreItem)scores.get(i);
        output += item.toString() + "\n";
    }

    return output;
}

private void load() { ... }
}

```

Gewapend met deze `toString` methode is de implementatie van `save` een fluitje van een cent:

```

private void save() {
    Collections.sort(scores);
    String output = toString();

    try {
        TextFile.write(HIGHSCORE_FILE, output);
    }
    catch (IOException io) {
        return;
    }
}

```

De vraag is nu nog: wanneer gaan we `load` en `save` oproepen? Op het moment dat een `Highscore` object wordt aangemaakt kunnen we de `ArrayList` populeren met de scores uit het tekstbestand. De constructor kan er m.a.w. als volgt uitzien:

```
// Highscore.java
public class Highscore {
    private static final int NUM_SCORES = 5;
    private static final String HIGHSCORE_FILE
        = "highscores.txt";
    private ArrayList scores;

    public Highscore() {
        this.scores = new ArrayList();
        load();
    }

    public String toString() { ... }
    private void load() { ... }
    private void save() { ... }
}
```

Wat we nog niet hebben is een methode om een nieuwe score aan de highscore toe te voegen. Iedere keer wanneer dit gebeurt moeten we de lijst bewaren om geen scores te verliezen. De methode `addScore` vervolledigt de klasse `Highscore`. Het is een goed gebruik om bij parameters van een objecttype te testen op een lege referentie.

```
// Highscore.java
public class Highscore {
    private static final int NUM_SCORES = 5;
    private static final String HIGHSCORE_FILE
        = "highscores.txt";
    private ArrayList scores;

    public Highscore() { ... }

    public void addScore(HighscoreItem item) {
        if (item != null) {
            scores.add(item);
            save();
        }
    }

    public String toString() { ... }
    private void load() { ... }
    private void save() { ... }
}
```

We hebben in de afgelopen bladzijden een stuk code opgebouwd die ons toelaat om op een heel eenvoudige manier met een highscore bestand te werken. Het moet altijd je streefdoel als programmeur zijn om code te schrijven die op zichzelf robuust is en resulteert in zo eenvoudig en compact mogelijke code. Als ik de klasse `Highscore` gebruik, kan ik met een paar regels code een score aan m'n highscore toevoegen en de output opvragen. Dat is niet slecht, toch?

```
// Gebruik van de klasse Highscore
Highscore score = new Highscore();
System.out.println("De huidige highscore:\n" + score);
score.addScore(new HighscoreItem("Rogier", 14));
System.out.println("De nieuwe highscore:\n" + score);

// De output van deze code is:
De huidige highscore:
Hans 23
Kristien 19
Davy 15
Luk 13
Frank 9

De nieuwe highscore:
Hans 23
Kristien 19
Davy 15
Rogier 14
Luk 13
```

## 4.6 Oefeningen

1. Maak een array aan voor 3 `Persoon` objecten, initialiseer de objecten met zinvolle data en teken de geheugenlayout van die array.
2. Laat de `klasse Persoon de interface Comparable implementeren.` Maak een array aan met 3 personen, en sorteer die. Druk de array af, eenmaal vóór en eenmaal na het sorteren.
3. Herhaal voorgaande oefening voor een `ArrayList` gevuld met 3 personen.
4. Definieer `2 subklassen voor de klasse Persoon: Docent en Student.` Vul een array met 2 docenten en 40 studenten, en sorteer die. Maak vervolgens een `ArrayList` aan, en probeer ook die te sorteren. Druk ook hier de array, resp. `ArrayList` af, eenmaal vóór en eenmaal na het sorteren.

5. Gebruik de klasse **MeerkeuzeVraag** uit hoofdstuk 2 om een console applicatie te maken waarmee een meerkeuzetest wordt afgenoemd. Werk naast de klasse **MeerkeuzeVraag** met een **klasse VraagBank** die je opbouwt rond een **ArrayList** van **MeerkeuzeVraag** objecten. Bij constructie van een **VraagBank** object wordt de arraylist gepopuleerd met meerkeuzevragen - bedenk een manier om de data voor die vragen uit een tekstbestand op te halen. Voorzie o.a. een methode **geefVraag** in de klasse om at random een vraag uit de vraagbank op te halen. Je zult moeten bijhouden welke vragen reeds werden gesteld tijdens de sessie, zorg ook voor een methode **reset** om de vraagbank te herstarten. Vanuit de console applicatie krijgt de gebruiker 5 vragen voorgeschoteld. Op het einde van een test krijg je als gebruiker je score en de optie om opnieuw een test af te leggen. Zorg er voor dat de testen steeds bestaan uit een willekeurige selectie van vragen uit de vraagbank, en ook dat de antwoorden van de vragen steeds in een andere volgorde verschijnen (dit laatste is normaal gezien al geïmplementeerd in de klasse **MeerkeuzeVraag**).
6. Gebruik **de klassen HighscoreItem en Highscore** om een highscore te implementeren in het **Hoger Lager spel** uit hoofdstuk 1. Aangezien er maximaal 10 beurten zijn is de kans op dubbele scores vrij groot als we de score enkel berekenen op basis van het aantal gespeelde beurten, we gaan dus ook de speeltijd in rekening moeten brengen. De klasse **System** voorziet een statische methode **currentTimeMillis** waarmee de huidige tijd in milliseconden kan opgevraagd worden. Door zowel bij de start als bij het einde van een spel de tijd op te vragen kunnen we de gespeelde tijd berekenen.

```
// Bij het begin van een spel
startTime = System.currentTimeMillis();

// Aan het einde van een spel
endTime = System.currentTimeMillis();

// Speeltijd in seconden
long speeltijd =
    Math.round((endTime - startTime) / 1000);
```

Een score kunnen we nu berekenen met volgende formule - één gespeelde beurt telt m.a.w. voor 100 seconden.

```
int score = (int)
    (1500 - (gespeeldeBeurten * 100 + speeltijd));
```

Voor de verwerking van de highscores ga je het spel hier en daar wat moeten aanpassen. Voeg een tweede **JFrame** aan je project toe (hier hebben we het

HighscoreForm genoemd) waarop je de highscores weergeeft. Met de volgende code kan je vanuit je eerste frame het andere oproepen. Let wel op dat je de default close operation moet aanpassen, anders wordt bij het sluiten van het highscore form ook het spel afgesloten.

```
HighscoreForm form = new HighscoreForm();
form.setDefaultCloseOperation(
    JFrame.DISPOSE_ON_CLOSE);
form.setVisible(true);
```