

---

# Modeloplossing Bundel 7

---

**Oefening 1:** De uitvoer van het gegeven programma ziet er als volgt uit:

```
a1
a3
a1
a3
a2
a1
a4
```

De methode die “a3” afdrukt, herdefinieert de methode “a1”. Voor referenties van het type B, is er sprake van overloading van de methodes “a2”, “a3”, “a4” en “a5”; voor referenties van het type A, van de methode “a1” en “a5”.

**Oefening 2:** “*Smurfenland*”

Figuur 1 toont een mogelijke klassenhierarchie. We gaan er hiervoor vanuit dat alle opsommingen in de opgave exhaustief zijn, e.g., “mannelijke Smurfen zijn zwart of blauw” interpreteren we als: alle mannelijke Smurfen zijn ofwel zwart ofwel blauw. De klasse `ManSmurf` is dus een abstracte klasse. We maken hier echter één uitzondering op, namelijk bij `WitSmurf`. We weten immers dat `Brilsmurf` noch een `Noordsmurf`, noch een `Zuidsmurf` is en dus heeft de klasse `WitSmurf` op zijn minst één instantiatie.

De kenners weten natuurlijk dat blauwe Smurfen in zwarte Smurfen kunnen veranderen en omgekeerd en zullen dus niet tevreden zijn met de klassenhierarchie van figuur 1. Als we aannemen dat zwarte Smurfen hun pakje aanhouden en hun noord/zuid geaardheid behouden, kunnen we de kleur beschouwen als attribuut van een `ManSmurf` of zelfs `Smurf` en methodes voorzien die die kleur kunnen veranderen. De klassen `ZwartSmurf` en `BlauwSmurf` verdwijnen dan uit de klassenhierarchie.

**Oefening 3:** “*Inheritance gedemonstreerd*”

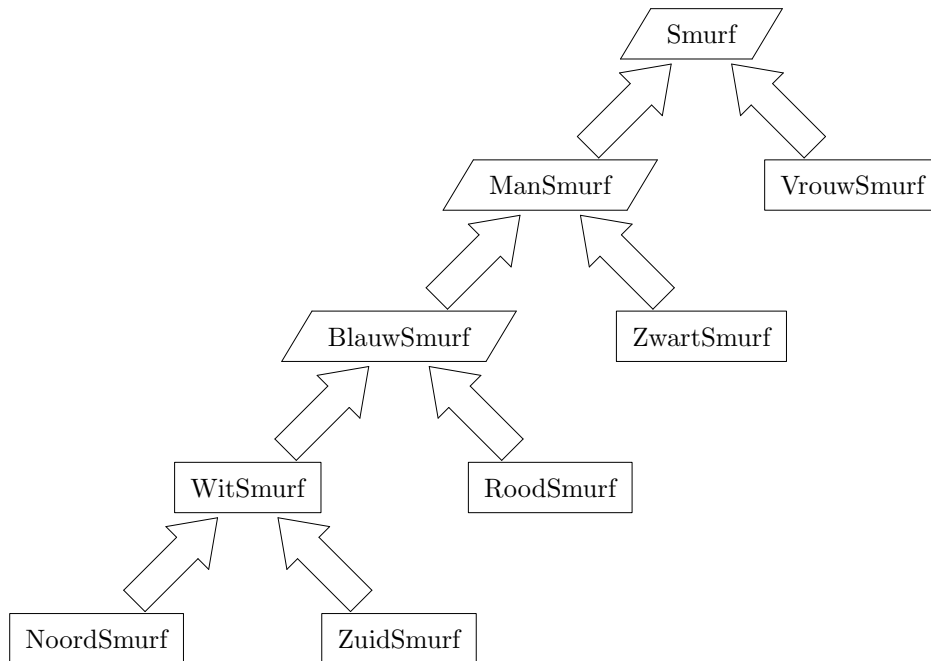
1. Schrijf een klasse `A` met een constructor `A()`. Binnen die constructor wordt de `String` “A” afgedrukt op het scherm.

```
// A.java
public class A {
    public A() {
        System.out.println("A");
    }
}
```

2. Schrijf een klasse `B` die een subklasse is van `A`. Definieer voor `B` geen constructor.

```
// B.java
public class B extends A {
}
```

3. Schrijf een klasse `Main` met een methode `main(String[] args)` waarin een object van het type `B` gecreëerd wordt.



Figuur 1: Smurfenhierarchie

```

// Main.java
public class Main {
    public static void main(String[] args) {
        B b = new B();
    }
}
  
```

Uitvoer:

A

- Geef klasse B een constructor B() waarbij binnen die constructor de String "B" op het scherm wordt afgedrukt.

```

// B.java
public class B extends A {
    public B() {
        System.out.println("B");
    }
}
  
```

Uitvoer:

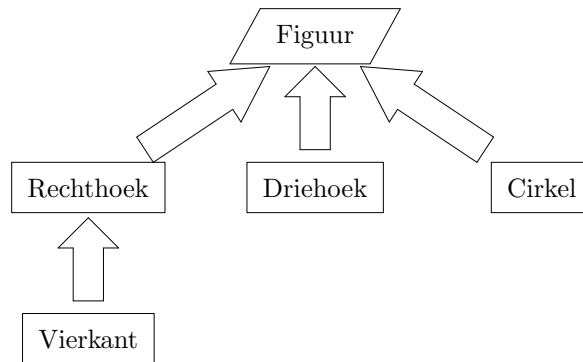
A

B

- Verander de constructor A() van klasse A in A(int i), i.e., de constructor moet nu opgeroepen worden met een int als argument.

```

// A.java
public class A {
  
```



Figuur 2: Figuren

```

public A(int i) {
    System.out.println("A");
}
}

```

⇒ het programma compileert nu natuurlijk niet meer aangezien de constructor van B impliciet de default constructor van A oproept die nu niet meer bestaat.

- Verander de constructor van B() zodat eerst A(int i) expliciet opgeroepen wordt (m.b.v. **super**) en daarna de String "B" afgedrukt wordt.

```

// B.java
public class B extends A {
    public B() {
        super(57);
        System.out.println("B");
    }
}

```

Uitvoer is identiek aan die uit puntje 4.

- Keer de volgorde van de twee regels code in B() om.

⇒ dit leidt tot compilatiefouten. De **super(...)** aanroep **moet** de eerste opdracht van de constructor zijn.

#### Oefening 4: "Figuur"

Het is duidelijk dat **Figuur** een abstracte klasse is. We kunnen immers niet in het algemeen de omtrek en oppervlakte van een figuur berekenen. Rechthoeken, driehoeken en cirkels zijn natuurlijk figuren en een vierkant is een speciaal geval van een rechthoek. Dit leidt tot de klassenhiërarchie van figuur 2.

We gebruiken een klasse **Positie** waarmee we de posities van de figuren gaan voorstellen.

```

// Positie.java
public class Positie {
    private int x, y;

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

```

```

    public int getX() { return x; }
    public int getY() { return y; }
}

```

De abstracte klasse `Figuur` houdt de positie bij die aan de constructor wordt doorgegeven en voorziet voor de rest enkel abstracte methodes.

```

// Figuur.java
import java.awt.*;

public abstract class Figuur {
    protected Positie pos;

    public Figuur(Positie _pos) {
        pos = _pos;
    }
    public abstract void draw(Graphics g);
    public abstract double omtrek();
    public abstract double oppervlakte();
}

// Cirkel.java
import java.awt.*;

public class Cirkel extends Figuur {
    protected double straal;

    public Cirkel(Positie pos, double r) {
        super(pos);
        straal = r;
    }
    public void draw(Graphics g) {
        g.drawOval(pos.getX(), pos.getY(), (int) straal, (int) straal);
    }
    public double omtrek() {
        return 2*Math.PI*straal;
    }
    public double oppervlakte() {
        return Math.PI*straal*straal;
    }
}

```

Wat betreft driehoeken was de opgave niet volledig gespecificeerd. Er was enkel sprake van een breedte en een hoogte en één zijde die evenwijdig is aan de X-as. Dit geeft ons niet voldoende informatie om de driehoek te tekenen. We kiezen ervoor om een gelijkbenige driehoek te tekenen met als derde zijde de zijde evenwijdig aan de X-as.

```

// Driehoek.java
import java.awt.*;

public class Driehoek extends Figuur {
    protected int breedte;
    protected int hoogte;

    public Driehoek(Positie pos, int _breedte, int _hoogte) {
        super(pos);
    }
}

```

```

        breedte = _breedte;
        hoogte = _hoogte;
    }
    public void draw(Graphics g) {
        int x = pos.getX();
        int y = pos.getY()+hoogte;
        g.drawLine(x,y,x+breedte,y);
        g.drawLine(x,y,x+breedte/2,y-hoogte);
        g.drawLine(x+breedte,y,x+breedte/2,y-hoogte);
    }
    public double omtrek() {
        return breedte+Math.sqrt(breedte*breedte+hoogte*hoogte)/2;
    }
    public double oppervlakte() {
        return breedte*hoogte/2;
    }
}

// Rechthoek.java
import java.awt.*;

public class Rechthoek extends Figuur {
    protected int breedte;
    protected int hoogte;

    public Rechthoek(Positie pos, int _breedte, int _hoogte) {
        super(pos);
        breedte = _breedte;
        hoogte = _hoogte;
    }
    public void draw(Graphics g) {
        g.drawRect(pos.getX(),pos.getY()+hoogte,breedte,hoogte);
    }
    public double omtrek() {
        return 2*(breedte+hoogte);
    }
    public double oppervlakte() {
        return breedte*hoogte;
    }
}

// Vierkant.java
public class Vierkant extends Rechthoek {
    public Vierkant(Positie pos, int zijde) {
        super(pos, zijde, zijde);
    }
}

```

Het volgende is een voorbeeld hoofdprogramma. Het stuk code van lijn 17 tot en met lijn 33 dient enkel om iets bruikbaar op het scherm te tonen en mag genegeerd worden.

```

// FiguurMain.java
import java.awt.*;
import java.awt.event.*;

```

```

5 public class FiguurMain extends Frame {
    public static int w = 200;
    public static int h = 200;

    private Figuur[] figuren;
10 private Canvas canvas;

    public static void main(String[] args) {
        new FiguurMain();
    }

15 public FiguurMain() {
    setVisible(true); //
    Insets insets = getInsets();
    setSize(w+insets.left+insets.right,
20         20+h+insets.top+insets.bottom);
    canvas = new Canvas();
    canvas.setSize(w,h);
    canvas.setVisible(true);
    add(canvas, BorderLayout.CENTER);
25 Button button = new Button("Draw");
    add(button, BorderLayout.SOUTH);
    button.addActionListener(
        new ActionListener() {
            public void actionPerformed(ActionEvent e) {
30                 draw();
            }
        }
    ); //

35 figuren = new Figuur[4];
    figuren[0] = new Cirkel(new Positie(30,50),20);
    figuren[1] = new Rechthoek(new Positie(40,60),20,30);
    figuren[2] = new Vierkant(new Positie(80,60),25);
    figuren[3] = new Driehoek(new Positie(70,40),20,20);

40 for (int i = 0; i < figuren.length; ++i)
    System.out.println("Omtrek figuur "+i+": "+
        figuren[i].omtrek());

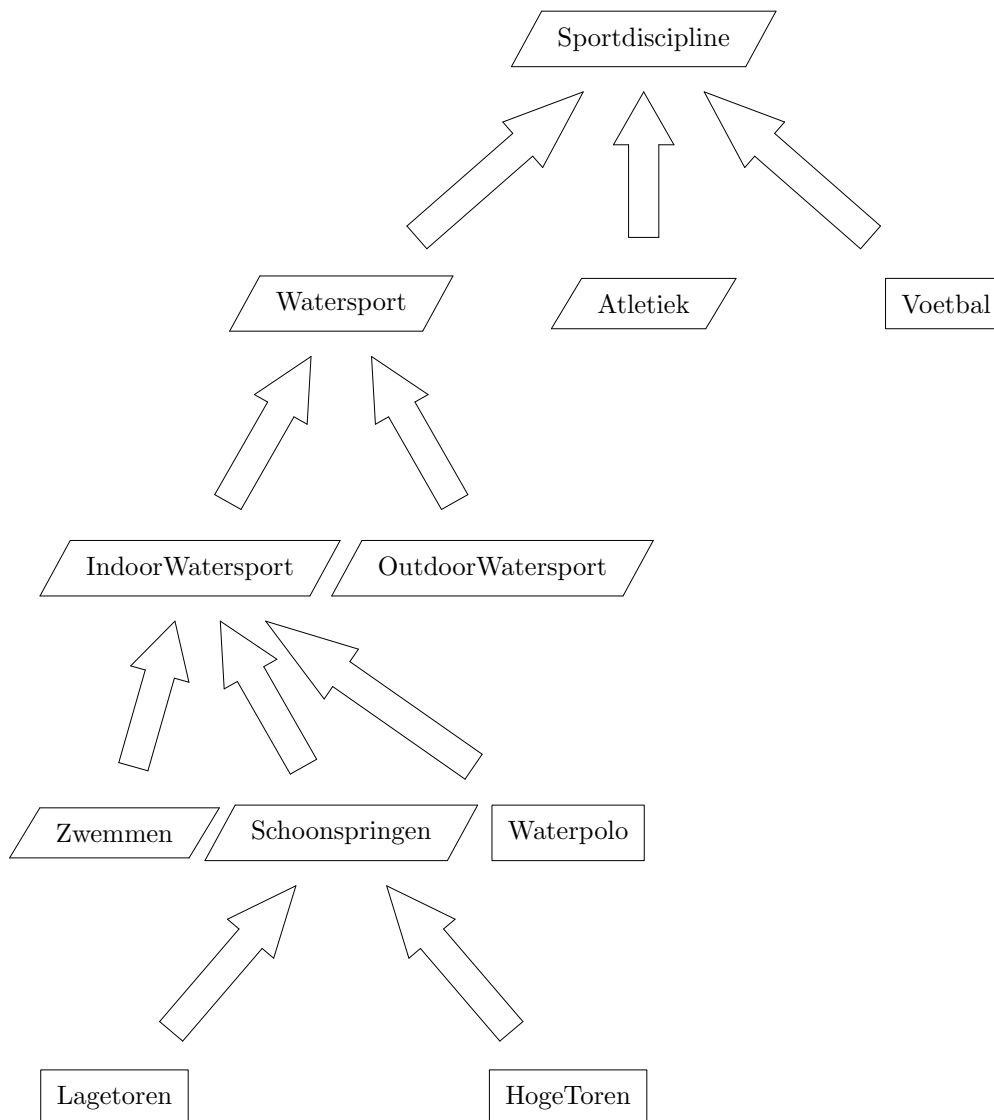
45 for (int i = 0; i < figuren.length; ++i)
    System.out.println("Oppervlakte figuur "+i+": "+
        figuren[i].oppervlakte());
}

50 private void draw() {
    Graphics g = canvas.getGraphics();
    for (int i = 0; i < figuren.length; ++i)
        figuren[i].draw(g);
    g.dispose();
55 }
}

```

### Oefening 5: "Olympische Spelen"

Het is uiteraard onbegonnen werk om *alle* onderdelen van de spelen in een klassediagram te gieten, daarom geven we hier een mogelijke aanzet in figuur 3.



Figuur 3: Olympische Spelen

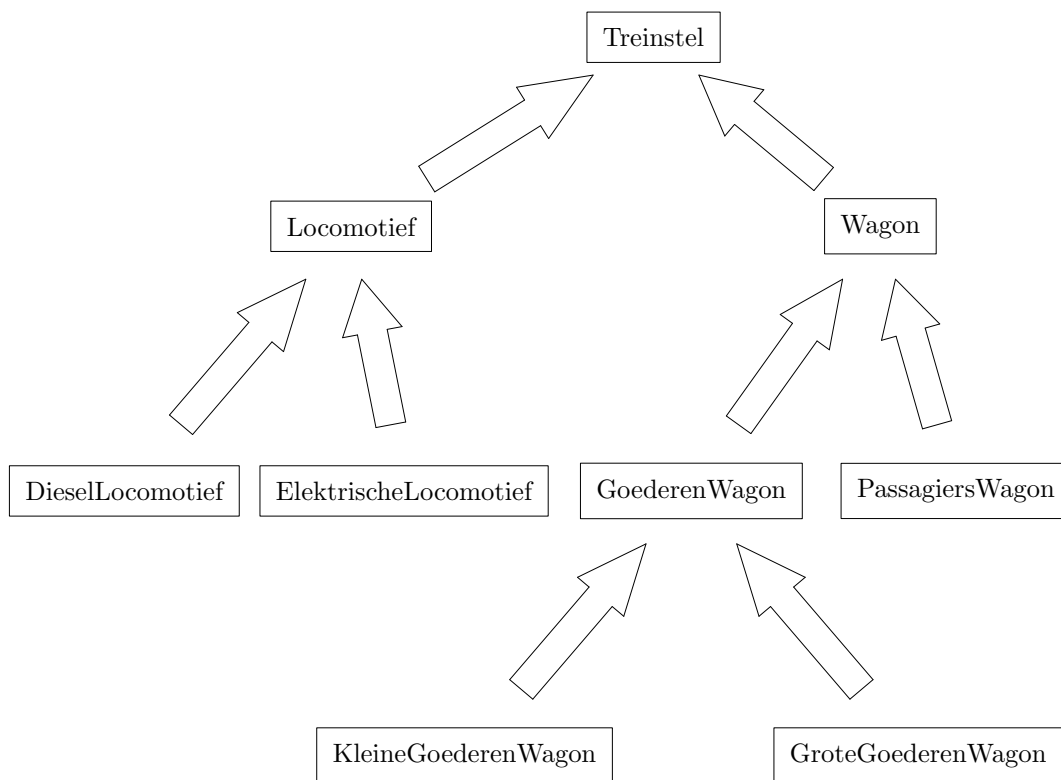
### Oefening 6:

De mogelijkheden om de gegeven opgave te modelleren zijn legio, in figuur 4 wordt een mogelijk klassendiagram getoond.

In de klasse `Treinstel` voorzien we meteen alle functionaliteit die een willekeurig treinstel (binnen de beperkingen van de opgave) moet voorzien. Zo heeft elk treinstel een maximale snelheid, een maximaal en een effectief aantal passagiers en eveneens een maximaal en een effectief aantal goederen. Passagiers kunnen op- of afstappen, goederen kunnen geladen en gelost worden.

```

public class Treinstel {
    protected final int maxAantalPassagiers;
    protected final int maxAantalKgGoederen;
    protected final int maxSnelheid;
}
  
```



Figuur 4: Treinstel hiërarchie

```

protected int aantalPassagiers;
protected int aantalKgGoederen;

public Treinstel(int _maxAantalPassagiers, int
    _maxAantalKgGoederen,
    int _maxSnelheid) {
    maxAantalPassagiers = _maxAantalPassagiers;
    maxAantalKgGoederen = _maxAantalKgGoederen;
    maxSnelheid = _maxSnelheid;
}

public int getMaxAantalPassagiers() {
    return maxAantalPassagiers;
}

public int getMaxKgGoederen() {
    return maxAantalKgGoederen;
}

public int getAantalKgGoederen() {
    return aantalKgGoederen;
}

public int getAantalPassagiers() {

```

```

    return aantalPassagiers;
}

public int getMaxSnelheid() {
    return maxSnelheid;
}

public int opstappen(int _aantalPassagiers) {
    int aantalPlaatsenOpen = maxAantalPassagiers -
        aantalPassagiers;
    int aantalGeweigerdePassagiers = 0;

    if (_aantalPassagiers <= aantalPlaatsenOpen) {
        aantalPassagiers += _aantalPassagiers;
        aantalGeweigerdePassagiers = 0;
    }
    else {
        aantalPassagiers = maxAantalPassagiers;
        aantalGeweigerdePassagiers = _aantalPassagiers -
            aantalPlaatsenOpen;
    }

    return aantalGeweigerdePassagiers;
}

public void afstappen(int _aantalPassagiers) {
    if (_aantalPassagiers <= aantalPassagiers) {
        aantalPassagiers -= _aantalPassagiers;
    }
    else {
        aantalPassagiers = 0;
    }
}

public int laden(int _aantalKgGoederen) {
    int aantalKgVrij = maxAantalKgGoederen - aantalKgGoederen;
    int aantalKgOver;

    if (_aantalKgGoederen <= aantalKgVrij) {
        aantalKgGoederen += _aantalKgGoederen;
        aantalKgOver = 0;
    }
    else {
        aantalKgGoederen = maxAantalKgGoederen;
        aantalKgOver = _aantalKgGoederen - aantalKgVrij;
    }

    return aantalKgOver;
}

public void lossen(int _aantalKgGoederen) {
    if (_aantalKgGoederen <= aantalKgGoederen) {
        aantalKgGoederen -= _aantalKgGoederen;
    }
}

```

```

        else {
            aantalKgGoederen = 0;
        }
    }
}

```

Een locomotief is een treinstel met maximaal 40 passagiers, een laadvermogen van 0 en een bepaalde maximale snelheid die meegegeven wordt bij de constructie ervan. Een locomotief heeft dus ook de methodes `laden()` en `lossen()`, ook al zijn deze hier eigenlijk niet van toepassing daar een locomotief toch geen goederen kan vervoeren. We komen hier zo dadelijk op terug.

```

public class Locomotief extends Treinstel {
    public Locomotief(int _maxSnelheid) {
        super(40, 0, _maxSnelheid);
    }
}

```

Het onderscheid tussen een diesel locomotief en een elektrische locomotief was in de opgave niet duidelijk gedefinieerd. Deze oplossing neemt aan dat dit onderscheid zich louter uit op het vlak van de maximale snelheid die een locomotief kan halen: een locomotief op diesel kan maximaal 95 kilometer per uur halen, een elektrische locomotief 125 kilometer per uur.

```

public class DieselLocomotief extends Locomotief {
    public DieselLocomotief() {
        super(95);
    }
}

public class ElektrischeLocomotief extends Locomotief {
    public ElektrischeLocomotief() {
        super(125);
    }
}

```

Een wagon heeft snelheid 0 en kan algemeen gezien zowel passagiers als goederen vervoeren.

```

public class Wagon extends Treinstel {
    public Wagon(int _maxAantalPassagiers, int _maxAantalKgGoederen) {
        super(_maxAantalPassagiers, _maxAantalKgGoederen, 0);
    }
}

```

Een goederenwagon kan geen passagiers vervoeren. Een kleine goederenwagon kan 5000 kilogram goederen vervoeren, een grote dubbel zoveel.

```

public class GoederenWagon extends Wagon {
    public GoederenWagon(int _aantalKgGoederen) {
        super(0, _aantalKgGoederen);
    }
}

public class KleineGoederenWagon extends GoederenWagon {
    public KleineGoederenWagon() {
        super(5000);
    }
}

```

```

public class GroteGoederenWagon extends GoederenWagon {
    public GroteGoederenWagon() {
        super(10000);
    }
}

```

Een passagierswagon kan 120 passagiers en 250 kilogram goederen vervoeren.

```

public class PassagiersWagon extends Wagon {
    public PassagiersWagon() {
        super(120, 250);
    }
}

```

Zoals al werd opgemerkt heeft dit ontwerp zekere problemen. De voornaamste tekortkoming is dat er een teveel is, met name een teveel aan functionaliteit van bepaalde objecten. Zo houdt elk *Wagon*-object een maximale snelheid bij, die je ook kunt opvragen, ook al is die onveranderlijk 0. We haalden ook al het voorbeeld van de locomotief aan, die eigenlijk geen goederen kan vervoeren.<sup>1</sup>

### Oefening 7:

Vermits Mastermind uiteindelijk maar een klein programma was is het hier moeilijk om op natuurlijke wijze overerving in te brengen. De voorgestelde oplossing kan daarom overkomen als lichtjes geforceerd.

Men kan een *Antwoord* en een *Combinatie* zien als twee verschillende uitbreidingen van een algemene klasse *KleurenRij*.

De gemeenschappelijke kenmerken zijn dan:

- Zowel *Antwoord*-objecten als *Combinatie*-objecten bevatten een rij van kleuren.
- *Antwoord*-objecten en *Combinatie*-objecten kunnen getekend worden op een *Rooster*.

De twee klassen hebben ook verschillen. Een *Antwoord*-object bevat enkel zwarte of witte pionnen, terwijl een *Combinatie*-object pionnen van verschillende kleuren kan bevatten. De kleurenrij moet bij een *Antwoord*-object niet noodzakelijk gevuld zijn (als er maar 1 goed geraden kleur is, dan is er ook slechts 1 zwart of wit pionnetje te voorzien), terwijl *Combinatie*-objecten doorgaans wel volledig gevulde kleurenrijen hebben. Het grootste verschil zal echter zijn dat de *Combinatie*-objecten een methode moeten hebben waarmee ze met elkaar vergeleken kunnen worden. *Antwoord*-objecten moeten niet vergeleken worden.

---

<sup>1</sup>Misschien is het ook niet nodig dit overschot aan informatie überhaupt als nadelig te ervaren: het feit dat alle treinstellen dezelfde methodes hebben, maakt dat je alle treinstellen op een uniforme manier kunt behandelen. Je moet nu niet telkens gaan zien tot welke klasse een specifiek treinstel behoort om te weten welke methodes er op van toepassing zijn en welke niet.