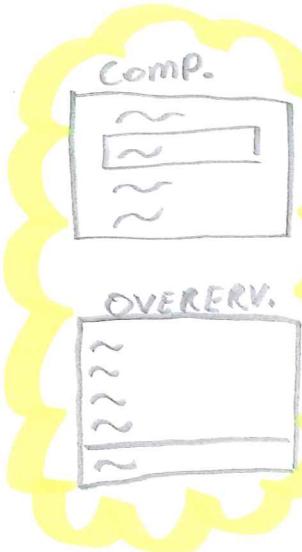


## Hergebruik van klassen

### Doelstellingen

Na dit hoofdstuk

- weet je hoe je nieuwe complexe types kunt samenstellen d.m.v. compositie
- weet je wat overerving is en hoe je deze techniek kunt aanwenden
- weet je hoe bij overerving de basisklasse kan geïnitialiseerd worden d.m.v. `super`, en wanneer dit impliciet gebeurt
- ken je het onderscheid tussen `public`, `protected` en `private`
- weet je dat methoden uit de basisklasse opnieuw geïmplementeerd kunnen worden (`overriding`)
- kan je tekenen op een panel d.m.v. de methoden uit de `Graphics` klasse
- weet je welke bijzondere rol de klasse `Object` speelt in Java
- kan je een `toString` en een `equals` methode implementeren
- weet je hoe en wanneer complexe types gecast kunnen worden



### 3.1 Inleiding

Uit vorig semester ken je reeds het basismechanisme om reeds geschreven code te herbruiken. We hebben het hierbij natuurlijk niet over **copy-paste**, maar wel over het feit dat **functionaliteit van één methode kan aangewend worden binnen de context van een andere methode**. Zo vermijd je herhaling van code en wordt je klasse in zijn geheel overzichtelijker. Naast het structureren in methoden heb je ook reeds de techniek van **method overloading** gezien. Hiermee kan je **varianten** van een bepaalde functionaliteit zinvol opdelen, en dat **op basis van zijn set van parameters**. Daarbij kan het zinvol zijn dat **een methode één van zijn overloaded versies aanspreekt**.

Het mooie aan Java als object-georiënteerde programmeertaal is mogelijkheid die je hebt om ook **bestaande klassen te herbruiken**. Het idee is dat, wanneer je een nieuwe klasse moet schrijven, je niet iedere keer van nul hoeft te beginnen, maar dat je **bestaande klassen als basis kunt nemen, klassen die mogelijk zelfs door iemand anders werden geschreven, getest en gedebuggd**. Het wiel moet je m.a.w. niet telkens opnieuw uitvinden. Java kent **twee specifieke technieken** om klassen te herbruiken: **compositie en overerving**.

**Bevat een klasse een veld van het objecttype, dan bestaat er een compositie-relatie tussen beide betrokken klassen.** Schrijf je een **nieuwe klasse voortbouwend** op de toestand en het gedrag **van een reeds bestaande klasse**, dan bestaat er een **overervingsrelatie** tussen beide betrokken klassen. Aldus overgeërfd methoden **herimplementeren kan via de techniek van method overriding**.

### 3.2 Compositie

Compositie is een fancy benaming voor een eenvoudig en intuïtief idee: je maakt gebruik van bestaande objecten om een nieuw complex type samen te stellen. Op het moment dat je **object referenties in een nieuwe klasse als veld opneemt** spreken we van compositie. **Compositie geeft een heeft-een relatie aan tussen klassen**.

Nemen we er de klasse **Persoon** uit het vorige hoofdstuk even terug bij, dan zien we dat we reeds compositie gebruikt hebben zonder het te weten. Er zijn hier twee compositierelaties, eentje voor de naam en eentje voor de voornaam. Aangezien **String** een objecttype is, bevatten **naam** en **voornaam** referenties:

```
// Persoon.java
public class Persoon {
```

```

private String naam;
private String voornaam;
private int leeftijd;
private boolean geslacht;

public Persoon(String naam, String voornaam) {
    this.naam = naam;
    this.voornaam = voornaam;
    this.leeftijd = 18; // default leeftijd
    this.geslacht = true; // default geslacht
}

public Persoon(String naam, String voornaam,
               int leeftijd, boolean geslacht) {
    this.naam = naam;
    this.voornaam = voornaam;
    this.leeftijd = leeftijd;
    this.geslacht = geslacht;
}

public boolean isVolwassen() {
    return (this.leeftijd >= 18);
}

public void verjaar() {
    this.leeftijd++;
}

public String geefVolledigeNaam() {
    return this.naam + " " + this.voornaam;
}
}

```

Maar zoals we ondertussen weten: onze eigen klassen vormen ook types die in niets onder moeten doen voor bestaande types zoals `String`. Stel, we willen een applicatie schrijven voor een bank. Wat we hiervoor nodig kunnen hebben is een klasse `Bankrekening` waarin we naast het rekeningnummer en het saldo ook informatie over de rekeninghouder willen bijhouden. Die informatie kan de naam van de houder zijn, maar bv. ook de leeftijd of het geslacht. En, waarom hiervoor geen `Persoon`-object gebruiken? We hebben die klasse namelijk reeds eerder voorzien van functionaliteit om met personen te werken!

```

// Bankrekening.java
public class Bankrekening {
    private String nummer;
    private Persoon houder;
    private double saldo;
}

```

er bij overerving **vertrokken van een bestaande klasse om er een eigen uitbreiding of variant van te maken.** Je zegt als het ware: dit nieuw type is zoals dat bestaande type, maar dan **uitgebreid of hier en daar gewijzigd.** Om overerving aan te geven wordt er een specifiek sleutelwoord gebruikt, nl. **extends.** Overerving geeft aanleiding tot **is-een relaties tussen klassen.**

### 3.3.1 Initialisatie van de basisklasse

Laat ons beginnen met een **eenvoudig voorbeeld.** We willen een **klasse Docent schrijven** waarin we informatie over een docent en het vak dat hij geeft kunnen bijhouden. **Eigenlijk hebben we al mooie code waarvan we kunnen vertrekken, een docent is immers een Persoon die een vak geeft.** We kunnen als volgt aangeven dat we van de **klasse Docent een uitbreiding willen maken op de bestaande klasse Persoon:**

```
// Docent.java
public class Docent extends Persoon {  
}
```

Probleem is: bovenstaande klasse zal niet compileren. Het punt is dat er nu niet één maar twee klassen in het spel zijn: de basisklasse (**Persoon**) en de afgeleide klasse (**Docent**). **Elk Docent-object bevat door overerving ook een Persoon-object, of meer algemeen: elk afgeleid object bevat door overerving een subobject van de basisklasse, en dat subobject moet geïnitialiseerd worden door middel van een constructor uit de basisklasse.** Nu zijn er twee mogelijkheden: ofwel verloopt de initialisatie **implicit via de default constructor uit de basisklasse, ofwel moeten we expliciet code voorzien.** Aangezien bovenstaande klasse nog geen code bevat wordt een default constructor voorzien, en aangezien deze klasse van een ander is afgeleid zal die default constructor de default constructor van de basisklasse zoeken. Maar die is er niet, met als gevolg: compilatiefout.

In ons voorbeeld moeten we met andere woorden **expliciet vanuit Docent een basisklasse constructor oproepen**, en dat doe je aan de hand van het **super sleutelwoord.** Voor de eenvoud laten we even de velden **leeftijd** en **geslacht** achterwege.

```
// Docent.java
public class Docent extends Persoon {
    private String vak;

    public Docent(String nm, String vnm, String vak) {
        super(nm, vnm);
        this.vak = vak;
    }
}
```

```

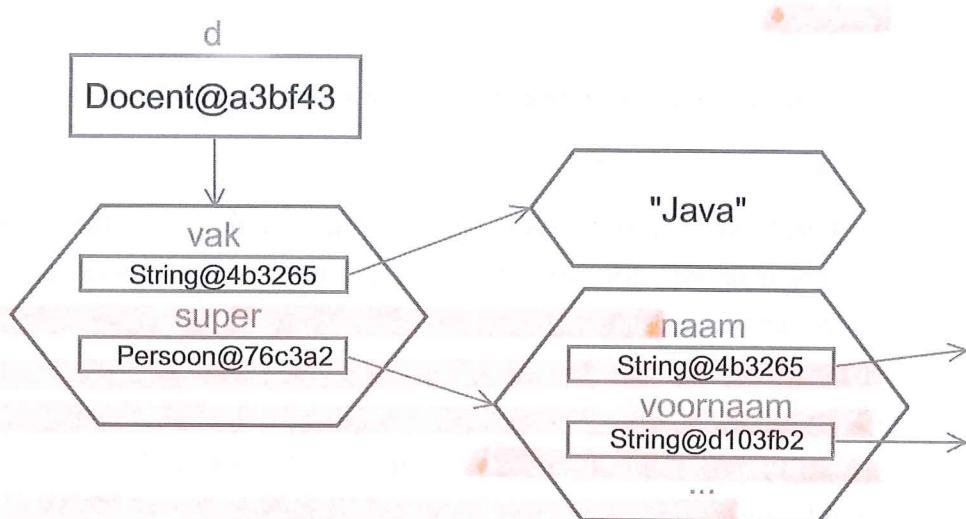
}

// ergens anders
Docent d = new Docent("Gruyaert", "Hans", "Java");

```

Merk op dat de initialisatie van het basisobject altijd moet gebeuren vóór de initialisatie van het afgeleid object. In bovenstaande constructor kunnen we met andere woorden niet eerst het veld `vak` initialiseren vóór `super`, dat zou ook een compilatiefout opleveren.

Schematisch kunnen we een `Docent`-object voorstellen zoals in afbeelding 3.2. Zoals je kan zien: ook al bevat `Docent` geen expliciete referentie naar `Persoon`, toch is er een `Persoon` (sub)-object aanwezig waar we kunnen naar refereren met het sleutelwoord `super`.



Figuur 3.2: Geheugenlayout van een afgeleid object

### 3.3.2 Uitbreiden en variëren

Overerving zorgt er nu voor dat alles wat publiek is in de basisklasse, ook beschikbaar is voor het afgeleid object. Alle methoden uit `Persoon` kunnen ook op `Docent`-objecten toegepast worden:

```

Docent d = new Docent("Gruyaert", "Hans", "Java");

if (d.isVolwassen()) {
    // doe iets
}

System.out.println(d.geefVolledigeNaam());

```

De methoden `isVolwassen` en `geefVolledigeNaam` heeft `Docent` als het ware geërfd van `Persoon`. Maar het hoeft hier niet bij te blijven: we kunnen `Docent` verder uitbreiden met extra methoden, bv. een getter en een setter voor het vak:

```
// Docent.java
public class Docent extends Persoon {
    private String vak;

    public Docent(String naam, String voornaam, String ↵
                  →vak) {
        super(naam, voornaam);
        this.vak = vak;
    }

    public void setVak(String vak) {
        this.vak = vak;
    }

    public String getVak() {
        return vak;
    }
}
```

Het spreekt voor zich dat deze methoden enkel toepasbaar zijn op `Docent`-objecten. Maar er is meer, we kunnen ook methoden uit de basisklasse herimplementeren. Stel dat we niet tevreden zijn met `geefVolledigeNaam` uit `Persoon`, een docent mag immers wel met meneer of mevrouw aangesproken worden. We kunnen in `Docent` ook een methode `geefVolledigeNaam` schrijven met exact dezelfde signatuur als de methode uit `Persoon`, dergelijke herimplementatie wordt ook wel overriding van een methode genoemd:

```
// Docent.java
public class Docent extends Persoon {
    private String vak;

    ...

    public String geefVolledigeNaam() {
        return "Mr. " + super.geefVolledigeNaam();
    }
}

// ergens anders
Docent d = new Docent("Gruyaert", "Hans", "Java");
// output: Mr. Gruyaert Hans
System.out.println(d.geefVolledigeNaam());
```

Let op het gebruik van `super`, hier om de versie van `geefVolledigeNaam` uit de basisklasse op te roepen. Met `super` kan dus niet alleen een constructor uit de basisklasse opgeroepen worden, maar eigenlijk alle publieke velden of methoden!

We zitten echter nog met één probleem: wat doen we met de vrouwelijke docenten, die kunnen we toch moeilijk ook met Mr. aanspreken. Laat ons eerst een constructor bijschrijven zodat we een vrouwelijke docent kunnen initialiseren:

```
// Docent.java
public class Docent extends Persoon {
    private String vak;

    public Docent(String naam, String voornaam, int ↗
                  leeftijd, boolean geslacht, String vak) {
        super(naam, voornaam, leeftijd, geslacht);
        this.vak = vak;
    }

    ...
}

// ergens anders
Docent d2 = new Docent("Van Assche", "Kristien", 35, ↗
                      true, "Java");
```

Ideaal zou zijn moesten we `geefVolledigeNaam` kunnen koppelen aan het geslacht, maar dit gegeven wordt in de basisklasse bewaard, en hoe kunnen we dit bereiken? Het gebruik van `super` is op zich een mogelijkheid, ware het niet dat het veld `geslacht` privaat is, niet toegankelijk dus, en dat willen we niet zomaar opgeven. Er zijn twee oplossingen: ofwel schrijven we een getter voor `geslacht` in `Persoon`, ofwel maken we `geslacht` `protected`.

Het toegangsrecht (ook wel **access specifier** genoemd) **protected** geeft een toegangsniveau aan tussen **public** en **private**: **protected** wil in het kort zeggen publiek voor de afgeleide klasse, maar **privaat** voor de buitenwereld. Noteer dat wij in deze cursus steeds expliciet het toegangsniveau bepalen. Strikt gezien kan je ook de modifier voor je velden of methoden weglaten, hiermee val je terug op het standaard toegangsniveau. Een volledig overzicht vind je terug in onderstaande tabel.




Modifier	Access Levels			
	class	package	subclass	world
public	J	J	J	J
protected	J	J	J	N
no modifier	J	J	N	N
private	J	N	N	N

Laat ons nu de twee hiervoor aangegeven oplossingen - schrijven van een getter voor geslacht resp. geslacht **protected** maken - bekijken:

```
// Persoon.java
public class Persoon {
    private String naam;
    private String voornaam;
    private int leeftijd;
    protected boolean geslacht;

    ...

    public boolean getGeslacht() {
        return geslacht;
    }
}
```

De versie van **geefVolledigeNaam** die gebruik maakt van de getter ziet er dan als volgt uit:

```
// Docent.java
public class Docent extends Persoon {
    private String vak;

    ...

    public String geefVolledigeNaam() {
        // true = vrouwelijk
        if (super.getGeslacht()) {
            return "Mevr. " + super.geefVolledigeNaam();
        }
    }
}
```

```
    return "Mr. " + super.geefVolledigeNaam();  
}  
}
```

Merk op dat we `getGeslacht` ook via `super` oproepen, hoewel dit strikt gezien niet nodig is. Er is immers geen verwarring mogelijk: er is slechts één methode `getGeslacht` in de beschouwde klassenhiërarchie, namelijk in de klasse `Persoon`. De prefix `super` mag hier dus evengoed weggelaten worden.

De versie met `protected` ziet er dan zo uit. Nogmaals: elk veld dat `protected` wordt gemaakt in de basisklasse, kan als publiek veld gebruikt worden in de afgeleide klasse. Deze keer gebruiken we geen `super`, hoewel we ook `super.geslacht` hadden kunnen schrijven. best wel voor mogelijke toekomst uitbreidingen.

```
// Docent.java
public class Docent extends Persoon {
    private String vak;
    ...
    public String geefVolledigeNaam() {
        // true = vrouwelijk
        if (geslacht) {
            return "Mevr. " + super.geefVolledigeNaam()
        }
        return "Mr. " + super.geefVolledigeNaam();
    }
}
```

### 3.4 Tekenen op een panel

In Java is het mogelijk om zelf te tekenen in een Swing applicatie door gebruik te maken van een **JPanel** component. Een default JPanel is een effen vlak dat je net zoals een ander component op een JFrame kunt plaatsen. Overerving biedt ons echter een techniek om een eigen variant te maken waarin het een en ander wordt getekend.

Voeg een nieuw JPanel Form aan je NetBeans project toe (zie figuur 3.3) en noem het **TekenPanel**. De code die gegenereerd wordt (een beetje uitgekust voor het overzicht) is als volgt:

```
// TekenPanel.java  
import javax.swing.*;
```

```

public class TekenPanel extends JPanel {
    public TekenPanel() {
        initComponents();
    }

    // Generated Code
    private void initComponents() {
        ...
    }
}

```



Figuur 3.3: Een JPanel toevoegen

Een paar dingen vallen op aan de code. Het is duidelijk dat de **klasse TekenPanel afgeleid is van JPanel**, wat wil zeggen dat alle publieke methoden uit de basisklasse toepasbaar zullen zijn op ons panel. Wat ook opmerkelijk is, is dat er **geen basisklasse constructor-oproep met super voorkomt in de constructor**. Dit komt omdat, zo kunnen we zien in de Java API, de klasse JPanel een default constructor heeft. We zouden de constructor dus ook als volgt kunnen schrijven, alleen is dat niet nodig aangezien die oproep sowieso impliciet gebeurt:

```

public TekenPanel() {
    super();
    initComponents();
}

```

Maar hoe kunnen we nu tekenen? Een **JPanel beschikt over een paintComponent methode** die uitgevoerd wordt wanneer het panel wordt getekend. Wat we kunnen doen, is **deze methode opnieuw implementeren (overridden dus) met onze eigen teken-code**. De signatuur van deze methode is als volgt - merk op dat we effectief ook mogen overiden aangezien het over een publieke methode gaat:

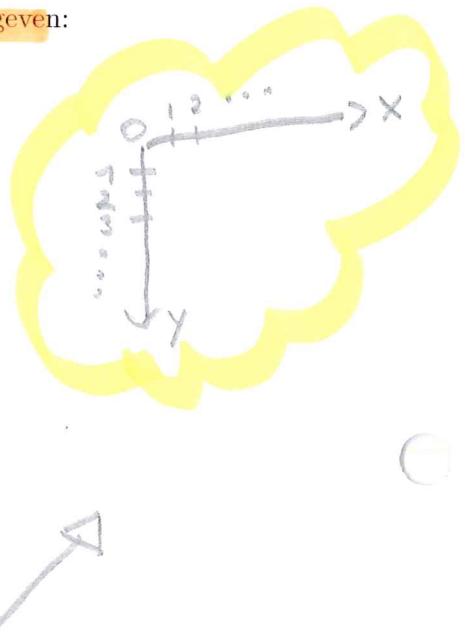
```
public void paintComponent(Graphics g)
```

Het **Graphics** argument wordt geïnitialiseerd met de grafische context van het panel. Het is dat object dat de visuele verschijning van Swing componenten bepaalt.

De meeste componenten hebben hiervoor ingebouwde code, maar bij een JPanel blijft dat beperkt tot een uniforme achtergrond, wat dit component natuurlijk uiterst geschikt maakt om op te tekenen. De **klasse Graphics** bevat een hele resem methoden om te tekenen, waarvan we er hier een selectie geven:

Most Used

- void setColor(Color color)
- void setFont(Font font)
- void drawString(String text, int x, int y)
- void drawLine(int x1, int y1, int x2, int y2)
- void drawRect(int x, int y, int width, int height)
- void drawOval(int x, int y, int width, int height)
- void fillRect(int x, int y, int width, int height)
- void fillOval(int x, int y, int width, int height)



De **meeste methoden maken gebruik van coördinaten**. Belangrijk om weten hierbij is dat de **X-as normaal georiënteerd is** (t.t.z. van links naar rechts), terwijl de **Y-as van boven naar onder loopt**. Het punt (0,0) ligt m.a.w. links bovenaan, het punt (breedte, hoogte) rechts onderaan. De breedte en hoogte van een panel kunnen opgevraagd worden met de `getWidth` en `getHeight` getters.

Stel nu dat we een vierkant ter grootte van het panel en een lijn van links boven naar rechts onder willen tekenen, elk in een ander kleurtje, dan kan de code in `paintComponent` er bv. als volgt uitzien. **Let er wel op dat je voor Graphics het package `java.awt` moet includeren.**

```
// TekenPanel.java
import javax.swing.*;
import java.awt.*;

public class TekenPanel extends JPanel {
    public TekenPanel() {
        initComponents();
    }

    // Generated Code
    private void initComponents() {
        ...
    }
}
```

```

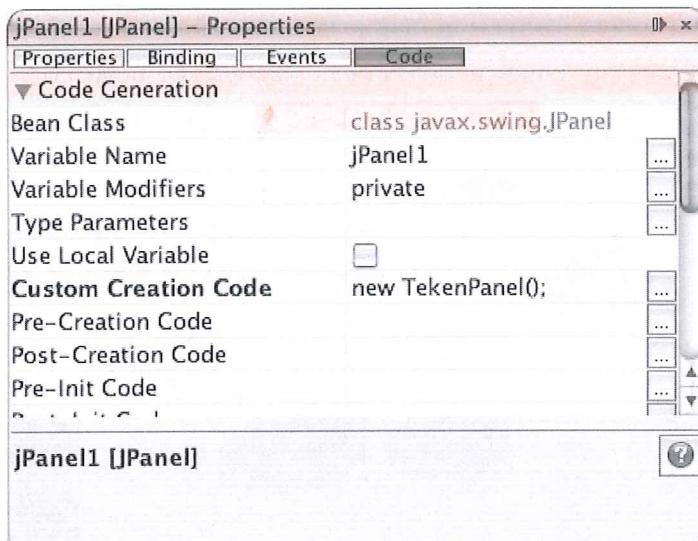
public void paintComponent(Graphics g) {
    int breedte = this.getWidth();
    int hoogte = this.getHeight();

    g.setColor(Color.BLUE);
    g.drawRect(10, 10, breedte-20, hoogte-20);

    g.setColor(Color.RED);
    g.drawLine(11, 11, breedte-11, hoogte-11);
}
}

```

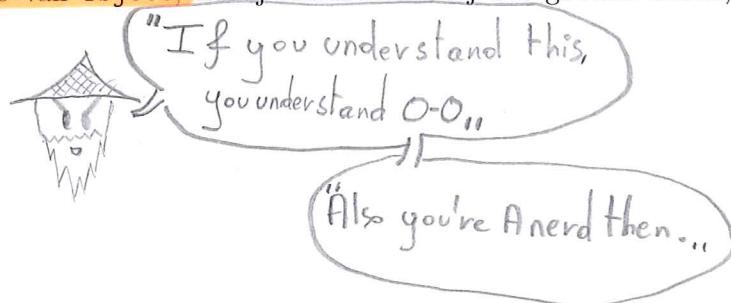
Om je eigen custom panel te kunnen gebruiken in NetBeans voeg je natuurlijk eerst een nieuw JFrame aan je project toe. Op het frame plaats je eerst een default panel (terug te vinden onder Swing Containers in het Palette). Om er voor te zorgen dat er geen gewoon JPanel wordt gebruikt maar ons TekenPanel, passen we de Custom Creation Code property van het panel aan (zie figuur 3.4). Wanneer we de code runnen krijgen we de Swing applicatie uit afbeelding 3.5 te zien.

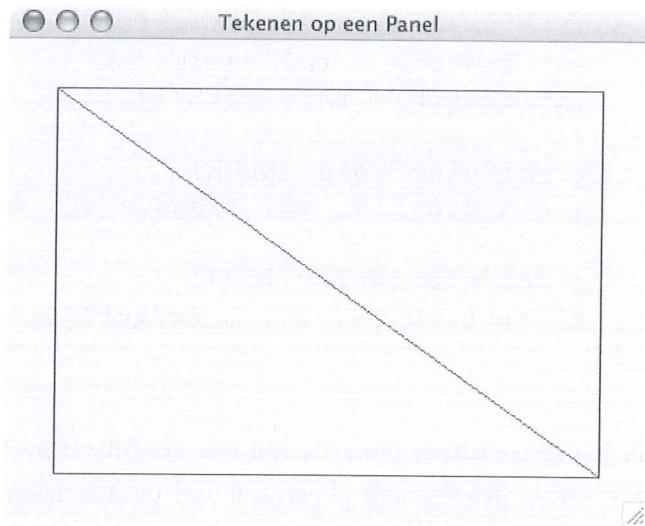


Figuur 3.4: Custom Creation Code

## 3.5 Het object Object

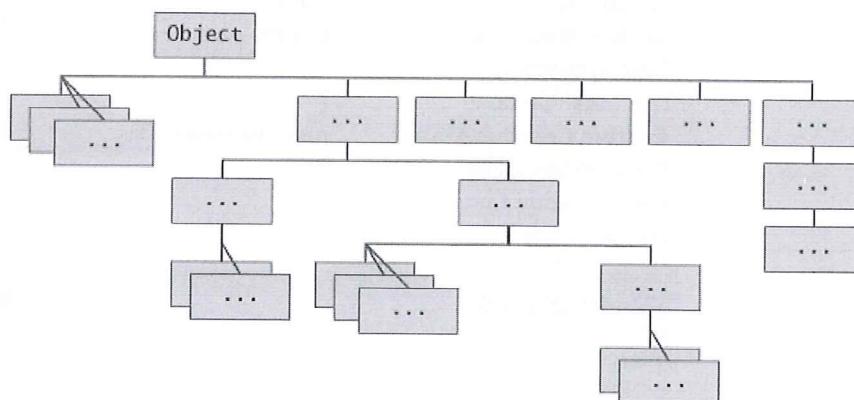
Een bijzondere rol wordt toegewezen aan de klasse **Object**. Want elke klasse in Java is ultiem afgeleid van **Object**. Erven van **Object** gebeurt nooit expliciet met extends, zoals we met Persoon en JPanel hebben gedaan. Nee, elke klasse erft sowieso van **Object**, hetzij rechtstreeks bij een gewone klasse, hetzij met één of





Figuur 3.5: Tekenen op een JPanel

meer tussenstations wanneer die klasse al erft van een andere klasse (die op zijn beurt misschien een directe afstammeling is van `Object`). Anders gezegd: bij elke **klasse zonder overerving mag je er extends Object bijdenken**. Dit geeft aanleiding tot **één grote klassenhiërarchie met Object helemaal bovenaan**, daarom wordt deze klasse soms ook **de root class genoemd** (figuur 3.6).



Figuur 3.6: Alle klassen zijn afgeleid van `Object`

Uit een vorige paragraaf weten we ondertussen dat alle methoden die `public` of `protected` zijn in de basisklasse ook toepasbaar zijn in de afgeleide klasse. D.w.z. dat **alle methoden uit Object bruikbaar zijn op elk object in Java**. Alle methoden uit `Object` vormen m.a.w. een gemeenschappelijkheid voor alle objecten. Laten we even kijken naar de methoden uit de klasse `Object`:

De klasse Object
protected Object clone()
boolean equals(Object obj)
protected void finalize()
Class getClass()
int hashCode()
void notify()
void notifyAll()
String toString()
void wait()
void wait(long timeout)
void wait(long timeout, int nanos)

Het zou ons nu te ver leiden om al deze methoden in detail te bekijken, maar er zijn er toch twee die de moeite zijn om even bij stil te staan: `toString` en `equals`. Over `toString` lezen we het volgende in de API:

*Returns a string representation of the object. In general, the `toString` method returns a string that "textually represents" this object. The result should be a concise but informative representation that is easy for a person to read. It is recommended that all subclasses override this method.*

In sommige gevallen willen we een manier hebben om onze objecten als tekst voor te stellen. Hernemen we even onze klasse `Persoon`:

```
// Persoon.java
public class Persoon {
    private String naam;
    private String voornaam;
    private int leeftijd;
    protected boolean geslacht;

    ...
}

// ergens anders
Persoon p =
    new Persoon("Gruyaert", "Hans", 32, false);
System.out.println(p);
```

Wanneer we het object `p` afprinten krijgen we gewoon de referentie te zien (genre Punt@2a3bf4). Dit is het default gedrag van `toString` zoals we de methode hebben geërfd van `Object`. Immers, een object meegeven aan `System.out.println` zal automatisch resulteren in een oproep van `toString`. Het voorgaande is m.a.w. identiek aan:

```
Persoon p =
    new Persoon("Gruyaert", "Hans", 32, false);
System.out.println(p.toString());
```

Wanneer we echter een **tekstuele representatie** aan ons object willen geven (zoals we in de API lezen), dan is de geijkte manier om dat te doen de **toString methode uit Object overiden**. Dat wil zeggen dat we exact dezelfde signatuur in onze eigen klasse moeten overnemen en deze methode een **invulling geven die onze noden past**. Er is **geen sluitende regel over hoe je een toString methode moet implementeren**, de inhoud van de String die je teruggeeft zal afhangen van object tot object. Voor Persoon zou dit zoets kunnen zijn:

```
// Persoon.java
public class Persoon {
    private String naam;
    private String voornaam;
    private int leeftijd;
    protected boolean geslacht;

    ...

    public String toString() {
        // true = vrouwelijk
        if (geslacht) {
            return naam + " " + voornaam + " (V)"
        }
        return naam + " " + voornaam + " (M)";
    }
}
```

Zoals de API ook al aangeeft: **het is een goede gewoonte om in elk van je klassen een toString methode te voorzien!** *Help bij debuggen!*

En dan is er **equals**. We weten al dat strings met **equals** worden vergeleken (waar we primitieven gewoon met **==** vergelijken):

```
String s1 = "hallo";
String s2 = "wereld";
if (s1.equals(s2)) {
    // doe iets
}
```

```
}
```

Zoals je weet worden objecten met een constructor geïnitialiseerd. Wanneer een klasse via compositie wordt samengesteld, moet je er goed op letten dat de object-referenties ook effectief geïnitialiseerd worden voor je ze gebruikt. Voor onze klasse Bankrekening hebben we twee opties: we kunnen houder initialiseren aan de hand van de afzonderlijke gegevens, of we kunnen er een Persoon-object voor gebruiken. Laat ons voor beide situaties een mogelijke constructor bekijken:

```
// Bankrekening.java
public class Bankrekening {
    private String nummer;
    private Persoon houder;
    private double saldo;

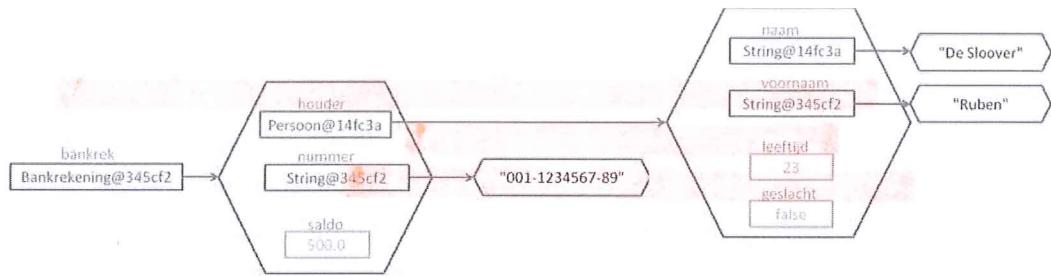
    public Bankrekening(String nummer,
                        String naam, String voornaam,
                        int leeftijd, boolean geslacht) {
        this.nummer = nummer;
        this.houder = new Persoon(naam, voornaam,
                                leeftijd, geslacht);
        this.saldo = 0.0;
    }

    public Bankrekening(String nr, Persoon houder) {
        this.nummer = nr;
        this.houder = houder;
        this.saldo = 0.0;
    }
}
```

Welke van beide opties je kiest hangt van situatie tot situatie af, je kan ze natuurlijk ook allebei voorzien (constructor overloading). Voor deze klasse lijkt de initialisatie via een Persoon-object het meest elegant en bruikbaar:

```
Persoon p = new Persoon("De Sloover", "Ruben", 23,
                       false);
Bankrekening bankrek = new Bankrekening("001-1234567-89", p);
```

En verder werken de klassen die d.m.v. compositie zijn samengesteld natuurlijk niet anders dan andere klassen. Het fijne is gewoon dat je een deel van de functionaliteit niet hoeft te herschrijven en je kan concentreren op waar het in je code echt om draait. Stel, bij wijze van voorbeeld, dat je bij die bank geen geld van je rekening kunt halen indien je jonger bent dan 18. Het beheer van de rekening hoort natuurlijk thuis in Bankrekening, maar om de leeftijd te testen hebben we al een



**Figuur 3.1:** Geheugenlayout van een Bankrekening object

methode in `Persoon`, dus kunnen we die gebruiken:

```
// Bankrekening.java
public class Bankrekening {
    private String nummer;
    private Persoon houder;
    private double saldo;

    public Bankrekening(String nr, Persoon houder) {
        this.nummer = nr;
        this.houder = houder;
        this.saldo = 0.0;
    }

    public boolean haalBedragAf(double bedrag) {
        if (houder.isVolwassen() && (bedrag <= saldo)) {
            saldo -= bedrag;
            return true;
        }
        return false;
    }
}
```

We laten de methode `haalBedragAf` een `boolean` teruggeven die aangeeft of het bedrag succesvol van de rekening werd gehaald of niet. Dit kan immers mislukken indien de houder te jong is (hiervoor roepen we de methode `isVolwassen()` aan uit de klasse `Persoon`), of als er geen geld genoeg meer op de rekening staat (hiervoor voorzien we de nodige code).

### 3.3 Overerving

Overerving is een tweede techniek om bestaande klassen te herbruiken. In plaats van een nieuw type samen te stellen uit bestaande types zoals bij compositie, wordt

Het verhaal van `equals` gaat echter verder dan de klasse `String` alleen: dé standaard manier om objecten in Java te vergelijken is met `equals`. Aangezien elke klasse erft van `Object` kunnen we sowieso al voor ieder object deze methode gebruiken, bv.

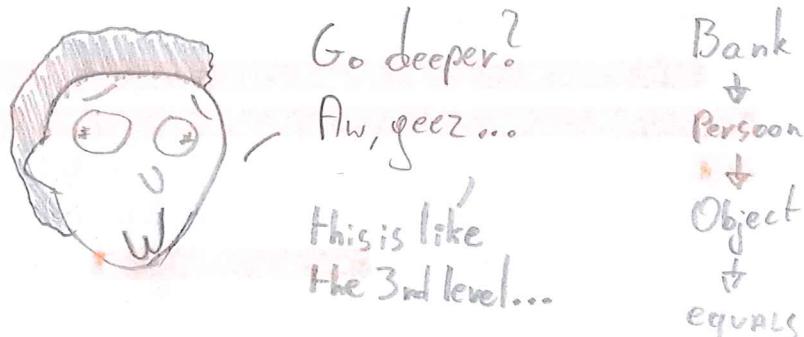
```
Person p1 =
    new Person("Gruyaert", "Hans", 32, false);
Person p2 =
    new Person("Van Assche", "Kristien", 34, true);
if (p1.equals(p2)) {
    // doe iets
}
```

De default versie van `equals` zoals geïmplementeerd in `Object` doet echter niets meer dan twee referenties vergelijken. Ook al zouden `p1` en `p2` inhoudelijk gelijk zijn, dan nog zou `equals` geen `true` geven. Dit kunnen we veranderen door `equals` opnieuw te implementeren in onze eigen klasse. Een belangrijk punt hierbij is dat we gebonden zijn aan de signatuur van de methode zoals die is vastgelegd in `Object`:

```
// Person.java
public class Person {
    private String naam;
    private String voornaam;
    private int leeftijd;
    protected boolean geslacht;

    ...
    public boolean equals(Object obj) {
        ...
    }
}
```

Het type van het argument is `Object`. Ook al wordt de methode gebruikt om in ons geval `Person`-objecten te vergelijken, dit is een vast gegeven. We zullen dus een manier moeten hebben om `Object` naar `Person` te casten. Laat ons daar eerst even op inzoomen voor we verder de implementatie van `equals` uitwerken.



## 3.6 Objecttypes casten

Zoals we al weten uit de basiscursus Java kunnen we primitieve types naar elkaar casten. In sommige gevallen verloopt dat impliciet, zoals bv. om van int naar double te gaan - vermits alle ints in een double kunnen is er geen risico op verlies van informatie. Omgekeerd, van double naar int, moet de cast wel explicet gebeuren aangezien we van meer naar minder gaan.

```
double d = 3.2;
int i = (int) d;
```

Om primitieve types naar complexe types te converteren bestaat er niet iets zoals een cast. We kunnen immers niet zomaar een waarde in een referentie omtoveren, en omgekeerd. Voor dergelijke conversies voorziet Java een aantal methoden. Voor een vollediger overzicht verwijzen we naar de appendix.

```
int i = Integer.parseInt("24");
String s = String.valueOf(i);
```

En dan is er nog een derde categorie van conversies: tussen complexe types onderling. Hier kunnen we wel casten, maar enkel in heel beperkte omstandigheden. Als twee klassen in rechte lijn via overerving met elkaar verbonden zijn, dan is casting mogelijk. Als we van een dieper gelegen type in de hiërarchie naar een hoger gelegen type gaan, dan kan de cast ook impliciet gebeuren. Immers, elk afgeleid object bevat een subobject van het basistype, dus die informatie is sowieso aanwezig. Concreet voorbeeld: aangezien Docent van Persoon is afgeleid, geen probleem!

```
Docent d = new Docent("Gruyaert", "Hans", "Java");
Persoon p = d;
```

Dergelijke casts worden upcasts genoemd, verwijzend naar de richting die we uitgaan met de cast in de klassehiërarchie. Het is belangrijk om te onthouden dat we geen informatie verliezen, hetgeen in Docent aan Persoon werd toegevoegd blijft bestaan, maar wordt enkel tijdelijk verborgen. We kunnen m.a.w. op elk moment terug naar ons oorspronkelijk type:

```
Docent d2 = (Docent) p;
```

Dit soort casts worden downcasts genoemd en moeten wel explicet gebeuren. De reden hiervoor is dat er in principe van Persoon meerdere klassen afgeleid kunnen zijn (bv. nog een klasse Student), en de compiler kan bij compilatie onmogelijk weten wat het oorspronkelijk type van p was. Een downcast kan m.a.w. mislukken, en dat wordt at runtime een **ClassCastException** opgegooid. Volgende cast zou m.a.w. niet lukken:

```
Student s = (Student) p;
```

Een speciale rol wordt opnieuw vervuld door `Object`. Elk complex type kan je upcasten naar `Object`, en kan van `Object` terug gedowncast worden naar het oorspronkelijk type.

---

### 3.7 De methode equals

Laat ons het verhaal van de `equals` methode hernemen. Wanneer we twee **Persoon-objecten vergelijken zouden we de upcast van het argument ook expliciet kunnen doen, alleen wordt dit nooit zo geschreven:** ~gebeurt dus impliciet

```
Persoon p1 =
    new Persoon("Gruyaert", "Hans", 32, false);
Persoon p2 =
    new Persoon("Van Assche", "Kristien", 34, true);
if (p1.equals((Object)p2)) {
    // doe iets
}
```

Gezien vanuit onze implementatie van `equals` is het argument van het type `Object`, maar we weten dat het hier eigenlijk om een `Persoon`-object gaat. Om met de data uit dat argument te kunnen werken zullen we het dus eerst **terug moeten casten naar Persoon**:

```
// Persoon.java
public class Persoon {
    private String naam;
    private String voornaam;
    private int leeftijd;
    protected boolean geslacht;

    ...
}

public boolean equals(Object obj) {
    Persoon arg = (Persoon) obj;
    ...
}
}
```

Nu rest ons nog te bepalen wanneer twee personen gelijk zijn. Dit is een keuze die je maakt, je kan enkel de naam vergelijken, of er ook nog de leeftijd bijnemen, etc.

```
public boolean equals(Object obj) {
    Persoon arg = (Persoon) obj;
    if (this.naam.equals(arg.naam) && this.voornaam.✓
        →equals(arg.voornaam)) {
        return true;
    }
    return false;
}
```

Merk op dat deze implementatie nog niet 100% waterdicht is. Eigenlijk kunnen we eender welk objecttype aan deze `equals` voeden, de compiler zal daar niet moeilijk over doen. Java voorziet een operator `instanceof` waarmee je kan testen of een bepaald object een instantie is van een bepaalde klasse. Hiermee kunnen we de implementatie van `equals` vervolledigen. *Let wel: the result of calling equals on the wrong type is false, not ClassCastException.* Anders gezegd: het is niet de bedoeling dat je een `ClassCastException` gaat opgooien vanuit `equals`.

```
public boolean equals(Object obj) {
    if (!(obj instanceof Persoon)) {
        return false;
    }
    Persoon arg = (Persoon) obj;
    if (this.naam.equals(arg.naam) && this.voornaam.✓
        →equals(arg.voornaam)) {
        return true;
    }
    return false;
}
```

Je zult merken dat NetBeans zal aandringen om samen met `equals` ook de methode `hashCode` te implementeren. Er zijn goede redenen om dat te doen, maar de implementatie van `hashCode` valt buiten het bestek van deze nota's. Het volstaat om NetBeans een default implementatie te laten genereren. Voor meer informatie verwijzen we bv. naar

<http://www.ibm.com/developerworks/java/library/j-jtp05273.html>

*dit is niet ok!*

```
public int hashCode() {
    int hash = 3;
    return hash;
}
```

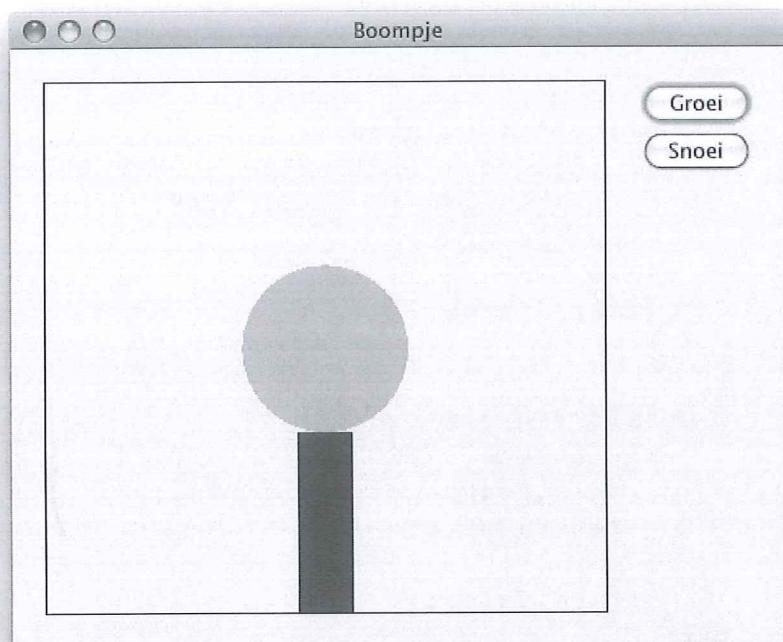
idee hash codes: je obj naar getal converteren, gelijke obj == gelijke hash code.

Dit is nodig voor hash tables (data storage)

Google een vb als je wilt. Zeer cool.

### 3.8 Probeer het uit!

Met deze *Probeer het uit!* gaan we een aantal technieken uit dit hoofdstuk illustreren. De bedoeling is om een applicatie te schrijven waarmee je een boompje kan laten groeien en snoeien. Er moet een eenvoudig boompje op een panel getekend worden, en je krijgt twee knoppen waarmee de boom vergroot of verkleind kan worden (zie figuur 3.7).



**Figuur 3.7:** Boom groeien en snoeien

Voor de boom zelf gaan we een afzonderlijke klasse schrijven. Je zou er het tekenen kunnen bijnemen, maar dit strookt met het idee om logica en presentatie te scheiden. Een boom met een grootte die kan groeien of gesnoeid worden is één ding, wat je er mee doet (in ons geval tekenen) is iets anders. De code voor de klasse Boom is vrij triviaal dus geven we ze meteen.

```
// Boom.java
public class Boom {
    public static final int DEFAULT_HOOGTE = 250;
    public static final int MAX_HOOGTE = 500;
    public static final int GROEI_STAP = 25;

    private int hoogte;
```

```

public Boom() {
    this.hoogte = DEFAULT_HOOGTE;
}

public Boom(int hoogte) {
    if (hoogte > 0 && hoogte <= MAX_HOOGTE) {
        this.hoogte = hoogte;
    }
    else {
        this.hoogte = DEFAULT_HOOGTE;
    }
}

public boolean groei() {
    if (hoogte + GROEI_STAP <= MAX_HOOGTE) {
        this.hoogte += GROEI_STAP;
        return true;
    }
    return false;
}

public boolean snoei() {
    if (hoogte - GROEI_STAP > 0) {
        this.hoogte -= GROEI_STAP;
        return true;
    }
    return false;
}

public int getHoogte() {
    return this.hoogte;
}

```

*best practice,  
ook toString()  
equals...“*

We stellen een boom voor d.m.v. z'n lengte in centimeter. De hoogte is begrensd op 500cm. Groeien of snoeien verloopt in stappen van 25cm. Let op het gebruik van publieke constanten. Dit houdt onze code overzichtelijker en nadien ook makkelijker aanpasbaar.

**De volgende stap is een eigen tekenpanel maken waarop we onze boom kunnen tekenen.** De basis voor deze klasse is identiek aan wat we al hebben gedaan:

```

// TekenPanel.java
import javax.swing.*;
import java.awt.*;

public class TekenPanel extends JPanel {

```

```

public TekenPanel() {
    initComponents();
}

// Generated Code
private void initComponents() {
    ...
}

// uitgevoerd wanneer het panel getekend wordt
public void paintComponent(Graphics g) {
    ...
}
}

```

We willen ons `TekenPanel` bouwen rond een `Boom`-object. Wat we ook willen is dat de grootte van de boom mee aangepast wordt aan de grootte van het panel, van daar de twee extra velden voor de dikte van de stam en de grootte van de kruin - deze velden gaan we straks invullen op basis van de breedte en hoogte van het panel.

```

// TekenPanel.java
public class TekenPanel extends JPanel {
    private Boom boom;
    private int stamDikte;
    private int kruinDiameter;

    public TekenPanel() {
        initComponents();
        this.boom = new Boom();
    }

    ...
}

```

Het tekenen zelf moet gebeuren vanuit `paintComponent`. Om onze code overzichtelijker te houden plaatsen we alle teken-code in een afzonderlijke methode `tekenBoom`. Merk op dat we die methode `private` hebben gemaakt aangezien die toch puur voor intern gebruik bestemd is, het is niet de bedoeling om die methode van buiten af te kunnen oproepen. De dikte van de stam stellen we tevens in op 10% van de breedte van het panel, de diameter van de kruin op 30%.

```

// TekenPanel.java
public class TekenPanel extends JPanel {
    private Boom boom;
    private int stamDikte;
    private int kruinDiameter;

```

```

public TekenPanel() {
    initComponents();

    this.boom = new Boom();
}

...

public void paintComponent(Graphics g) {
    super.paintComponent(g);

    // update na eventuele resize
    stamDikte = (int)(0.1 * this.getWidth());
    kruinDiameter = (int)(0.3 * this.getWidth());

    tekenBoom(g);
}

private void tekenBoom(Graphics g) {
    ...
}

```

De oproep via `super` van `paintComponent` uit de basisklasse `JPanel` is een handige manier om je panel te wissen. Het default gedrag van `paintComponent` is immers gewoon een effen vlak tekenen.

**De invulling van `tekenBoom` is iets minder eenvoudig.** Laten we beginnen met **de stam**. Het aantal beschikbare pixels voor de stam is de hoogte van het panel verminderd met de grootte van de kruin. We willen dat onze boom een beetje evenredig groeit, dus moeten we berekenen hoeveel pixels we per centimeter beschikking hebben. Hiervoor delen we het totaal aantal pixels door de maximale hoogte die een boom kan hebben (`MAX_HOOGTE`). Dit aantal pixels per centimeter moeten we vermenigvuldigen met de huidige grootte van de boom om de hoogte van de stam in pixels te kennen. Let op de cast naar `double` om te forceren dat de deling reëel wordt uitgevoerd.

```

private void tekenBoom(Graphics g) {
    double pixPerCm
        = ((this.getHeight() - kruinDiameter) / (double)boom.MAX_HOOGTE);
    int stamHoogte
        = (int)(pixPerCm * boom.getHoogte());
    ...
}

```

Om een rechthoek te tekenen moet je de coördinaten van de linker bovenhoek opgeven. Voor de X-coördinaat trekken we van het midden van het panel de helft van de dikte van de stam af, de Y-coördinaat is de hoogte van het panel verminderd met de hoogte van de stam (denk er aan dat de Y-as omgekeerd loopt). Met al deze gegevens kunnen we eindelijk onze stam tekenen:

```

private void tekenBoom(Graphics g) {
    double pixPerCm
        = ((this.getHeight() - kruinDiameter) / (double)boom.MAX_HOOGTE);
    int stamHoogte
        = (int)(pixPerCm * boom.getHoogte());
    int xLinksBoven
        = (this.getWidth() - stamDikte) / 2;
    int yLinksboven
        = this.getHeight() - stamHoogte;

    // teken stam
    g.setColor(Color.DARK_GRAY);
    g.fillRect(xLinksBoven, yLinksboven,
               stamDikte, stamHoogte);
    ...
}

```

Op basis van de positie van de stam kan de kruin vrij eenvoudig getekend worden, kwestie van kruinDiameter pixels boven de stam te beginnen.

```

private void tekenBoom(Graphics g) {
    double pixPerCm
        = ((this.getHeight() - kruinDiameter) / (double)boom.MAX_HOOGTE);
    int stamHoogte
        = (int)(pixPerCm * boom.getHoogte());
    int xLinksBoven
        = (this.getWidth() - stamDikte) / 2;
    int yLinksboven
        = this.getHeight() - stamHoogte;

    // teken stam

```

```

g.setColor(Color.DARK_GRAY);
g.fillRect(xLinksBoven, yLinksboven,
           stamDikte, stamHoogte);

// teken kruin
g.setColor(Color.GREEN);
g.fillOval(
    (int)((this.getWidth() - kruinDiameter) / 2),
    yLinksboven - kruinDiameter,
    kruinDiameter, kruinDiameter);
}

```

Ziezo. Het enige wat we nog mankeren in onze klasse `TekenPanel` zijn methoden om de boom te laten groeien/snoeien. Je kan een component (zoals ons panel) vanuit **code opnieuw laten tekenen via een oproep van de `repaint()` methode**. Een oproep naar `repaint` zorgt ervoor dat er een call naar `paintComponent` aan de wachtrij met events wordt toegevoegd. In deze wachtrij komen allerlei events terecht, zoals bv. klikken op een knop of een automatische repaint naar aanleiding van een window resize, die één voor één moeten uitgevoerd worden. Het is m.a.w. perfect mogelijk dat het is zoals bij de beenhouwer: nog zoveel wachtenden voor u. We spreken hier natuurlijk over fracties van seconden, maar toch, **het resultaat is dat `repaint` geen instant effect heeft**. Om animaties te maken is het dus geen oplossing om bv. 1000 keer na elkaar `repaint` op te roepen, hiervoor bestaan andere technieken. Nog één ding: **`paintComponent` rechtstreeks oproepen is not done!** ← geeft errors, hoofdprijs en soms zelfmoordneigingen

```

// TekenPanel.java
public class TekenPanel extends JPanel {
    private Boom boom;
    private int stamDikte;
    private int kruinDiameter;

    public TekenPanel() { ... }
    private void initComponents() { ... }
    public void paintComponent(Graphics g) { ... }
    private void tekenBoom(Graphics g) { ... }

    public boolean groeiBoom() {
        boolean res = boom.groei();
        repaint();
        return res;
    }

    public boolean snoeiBoom() {
        boolean res = boom.snoei();
    }
}

```

```
    repaint();  
    return res;  
}  
}
```

We kunnen analoog te werk gaan als voorheen om ons panel te koppelen aan een JFrame. We voorzien twee knoppen, één om te groeien en één om te snoeien. De code voor de knoppen is vrij triviaal, we moeten er enkel voor zorgen dat de knoppen gedisabled worden indien de boom zijn maximale of minimale grootte heeft bereikt (van daar het return-type boolean). Merk op dat het type van het panel JPanel blijft, ook al hebben we er een TekenPanel aan toegewezen. De referentie moet dus eerst gedowncast worden.

```
// SwingApp.java
public class SwingApp extends JFrame {
    public SwingApp() {
        initComponents();
    }

    private void initComponents() { ... }

    private void btnGroeiActionPerformed(ActionEvent evt) {
        boolean res = ((TekenPanel)tekenP).groeiBoom();
        if (!res) {
            btnGroei.setEnabled(false);
        }
        btnSnoei.setEnabled(true);
    }

    private void btnSnoeiActionPerformed(ActionEvent evt) {
        boolean res = ((TekenPanel)tekenP).snoeiBoom();
        if (!res) {
            btnSnoei.setEnabled(false);
        }
        btnGroei.setEnabled(true);
    }

    ...
}

// Variables declaration - do not modify
private javax.swing.JButton btnGroei;
private javax.swing.JButton btnSnoei;
private javax.swing.JPanel tekenP;
// End of variables declaration
}
```

Door de Horizontal Resizable en Vertical Resizable properties van het panel op true te zetten zorg je er voor dat de grootte van het panel automatisch mee evolueert met die van het frame.

Om te besluiten nog even de nadruk leggen op hoe de presentatie en de logica netjes gescheiden zijn gebleven in deze Swing applicatie: de klassen SwingApp en TekenPanel zorgen voor de presentatie, de klasse Boom vormt de logica. Het is trouwens enkel TekenPanel die van Boom gebruik maakt, wat SwingApp betreft bestaat die klasse zelf niet! Dit is een mooie gelaagde manier van werken die je ook in je eigen applicaties moet proberen nastreven.

## 3.9 Oefeningen

1. Implementeer een `toString` en `equals` methode in de klasse `Punt`. Test je code uit aan de hand van een Console testprogramma.
2. Gebruik compositie om een klasse `Cirkel` te schrijven met als velden het middelpunt (een `Punt`) en de straal (een `double`). Schrijf een gepaste constructor, alsook methoden om
  - de omtrek van de cirkel te berekenen
  - de oppervlakte van de cirkel te berekenen
  - de afstand te berekenen tussen een gegeven punt en het middelpunt van de cirkel

Voor het berekenen van de afstand tussen 2 punten maak je gebruik van de in hoofdstuk 2 aangereikte methode uit de klasse `Punt`:

```
public double berekenAfstand (Punt anderPunt)
```

Test je code uit voor onderstaand object en teken de bijhorende geheugenlayout.

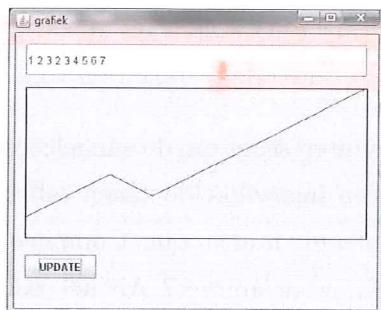
```
Cirkel c = new Cirkel(new Punt(1, 2), 2.0);
```

3. Implementeer een klasse `HighscoreItem`. Een object uit deze klasse stelt één highscore item voor, en bestaat bijgevolg uit een naam en een score. Voorzie een constructor in deze klasse, getters voor de naam en de score, en tenslotte een `toString` methode. In een later labo zullen we de highscores van meerdere spelers verzamelen in een lijst en in een functionele context plaatsen.

4. Zoek in de Java API op van welke klassen de klasse `String` en de klasse `ArrayList` afgeleid zijn. Teken de volledige hiërarchie.
5. Het is altijd interessant om de aangeleerde theorie eens toegepast te zien zonder dat je meteen ingewikkelde zaken zelf moet gaan doen. In die optiek kan de volgende oefening nuttig zijn. Compileer na elke stap en ga na of dit lukt. Als het niet lukt: waarom niet? Als het wel lukt: probeer het uit te voeren. Lukt dat probleemloos? Als het niet lukt: waarom niet? Als het wel lukt: is de uitvoer wat je verwacht had?
  - (a) Schrijf een klasse `A` met een constructor `A()`. Binnen die constructor wordt de `String` "A" afgedrukt op het scherm.
  - (b) Schrijf een klasse `B` die een subklasse is van `A`. Definieer voor `B` geen constructor.
  - (c) Schrijf een klasse `Test` met een methode `main` waarin een object van het type `B` gecreëerd wordt.
  - (d) Geef klasse `B` een constructor `B()` waarbij binnen die constructor de `String` "B" op het scherm wordt afgedrukt.
  - (e) Verander de constructor `A()` van klasse `A` in `A(int i)`, i.e., de constructor moet nu opgeroepen worden met een `int` als argument.
  - (f) Verander de constructor van `B()` zodat eerst `A(int i)` expliciet opgeroepen wordt (m.b.v. `super`) en daarna de `String` "B" afgedrukt wordt.
  - (g) Keer de volgorde van de twee regels code in `B()` om.
6. Implementeer de klasse `Uitgave` waarmee je de uitgaven van een drukkerij kan beschrijven. Een uitgave wordt gekenmerkt door een titel en een prijs. Maak deze velden `protected`. Voorzie een gepaste constructor, een `toString()` en een `equals()` methode.

Voorzie vervolgens de klasse `Tijdschrift`, afgeleid van de klasse `Uitgave`. Een tijdschrift is een uitgave waarvoor tevens de maand en het jaar van publicatie worden bijgehouden. Implementeer een constructor waarmee alle gegevens van een tijdschrift (titel, prijs, maand en jaar) geïnitialiseerd kunnen worden. Voorzie ook op dit subklasse niveau een zinvolle `toString()` en `equals()` methode.

Ga op analoge manier te werk voor de klasse `Boek`. Een boek is een uitgave waarvoor ook de auteur en het aantal bladzijden worden bijgehouden.

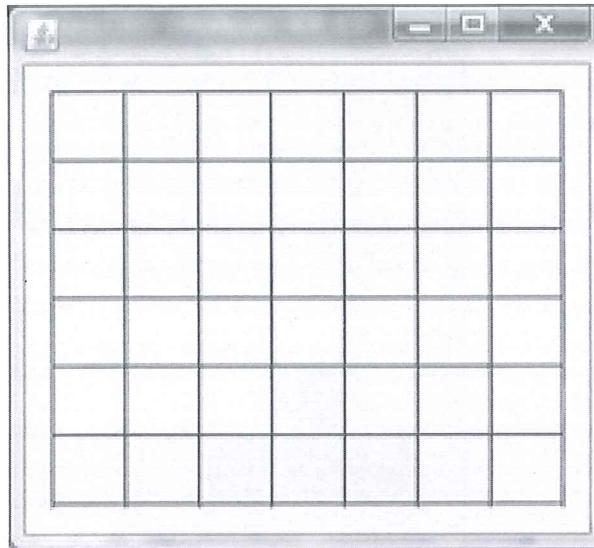


Rando  
afbeelding

**Figuur 3.8:** Een grafiek tekenen

Compileert je code nog wanneer je de velden in de basisklasse **private** maakt?  
Optimaliseer zo nodig je code opdat dit mogelijk wordt.

7. Implementeer een Java Swing applicatie die een grafiek tekent op basis van een set van ingegeven positieve meetwaarden. Zorg ervoor dat je grafiek evenwichtig verdeeld is over je tekenpanel, zowel horizontaal als verticaal (zie figuur 3.8).  
*Afrikaans...*
8. Implementeer een Java Swing applicatie die een quote weergeeft van een auteur. Op Toledo vind je het tekstbestandje **quotes.txt** terug met daarin een aantal quotes van auteurs. Het is de bedoeling dat je, door opeenvolgend te drukken op een knop, alle beschikbare quotes kan tonen. Na de laatste quote toon je opnieuw de eerste quote.
  - (a) Ontwerp eerst een klasse **Quote**. Voorzie velden voor de tekst en de auteur van de quote, een gepaste constructor en getters voor de velden. De klasse **Quote** gaan we gebruiken om de informatie van één quote in te bewaren.
  - (b) Schrijf vervolgens een klasse **Quotations** met daarin een standaard constructor en een methode **getNextQuote**.
    - i. Via de constructor wordt het opgegeven tekstbestand ingelezen, de ingelezen data wordt bewaard als een rij van **Quote**-objecten. Maak voor het inlezen van de quotes gebruik van de klasse **TextFile**. Dit is een utility klasse die we jullie ter beschikking stellen via een Java Bibliotheek (**CodeLibrary.jar**). Zie Appendix A voor meer informatie hierrond.
    - ii. Via de methode **getNextQuote** wordt de volgende quote uit de rij gehaald. Na de laatste quote wordt opnieuw de eerste quote uit de rij gekozen.



Figuur 3.9: Een 6x7 spelbord

9. Definieer een **JPanel** klasse in je project en noem dit **Spelbord**. Het is de bedoeling om er een spelbord bestaande uit 6 rijen en 7 kolommen in te tekenen.

```
public class Spelbord extends JPanel {
    public static final int RIJEN = 6;
    public static final int KOLOMMEN = 7;
    ...
}
```

Implementeer volgende methoden in de klasse **Spelbord**:

- **public void paintComponent(Graphics g)**

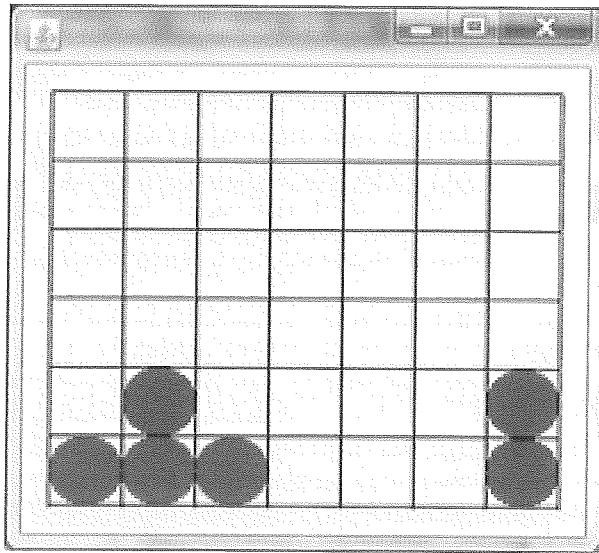
Deze methode wordt opgeroepen wanneer het panel getekend wordt. Van hieruit wordt de private methode **tekenBord** opgeroepen.

- **private void tekenBord(Graphics g)**

Tekent het spelbord zoals in figuur 3.9. Het spelbord moet mee aangepast worden met de grootte van het panel.

Test uw spelbord uit om te zien of alles naar behoren werkt. Voeg daarom een **JFrame** toe en plaats er een **JPanel** op dat verwijst naar de klasse **Spelbord** (zie sectie 3.4 voor meer uitleg hierover).

10. Breid je **Spelbord** klasse verder uit zó dat ze een spelbord voorstelt voor het spel **4-op-een-rij**. Het spelbord bestaat uit 6 rijen en 7 kolommen, de geworpen schijven en hun positie houden we intern bij als een array van getallen.



Figuur 3.10: Een 4-op-een-rij spelbord

```
public class Spelbord extends JPanel {
    public static final int RIJEN = 6;
    public static final int KOLOMMEN = 7;
    public static final int SPELER = 1;
    public static final int COMPUTER = 2;

    private int[][] schijven;
    ...
}
```

Via het `schijven` veld houden we voor elke cel bij wat er zich op die positie van het bord bevindt:

- geen schijf (=0)
- een schijf van de speler (=1)
- een schijf van de computer (=2)

Implementeer volgende methoden in de klasse `Spelbord`:

- `public void paintComponent(Graphics g)`

Deze methode wordt opgeroepen wanneer het panel getekend wordt. Van hieruit worden de private methoden `tekenBord` en `tekenSchijven` opgeroepen.

- `private void tekenBord(Graphics g)`

Tekent het spelbord zoals in figuur 3.9. Het spelbord moet mee aangepast

worden met de grootte van het panel.

- **private void tekenSchijven(Graphics g)**

Tekent de schijven op het spelbord zoals in figuur 3.10: geel voor de speler, rood voor de computer. Hou opnieuw rekening met de grootte van het panel.

- **public boolean werpSchijf(int speler, int kolom)**

Deze methode voegt een schijf toe aan de **schijven** array in de aangegeven kolom en voor de aangegeven speler (SPELER of COMPUTER). De methode geeft **true** terug indien dit lukt, en **false** indien niet (bv. als de kolom in kwestie reeds vol is).

De positie waar de gebruiker heeft geklikt kan je makkelijk achterhalen door het **mouseClicked** event van het panel op te vangen. Om dit event te implementeren klik je rechts op het panel en kies je voor Events > Mouse > mouseClicked. Via de **getX** en **getY** methoden kan je de coördinaten van de klik opvragen.

```
private void formMouseClicked(MouseEvent evt) {
    int x = evt.getX();
    int y = evt.getY();
    ...
}
```

Het is niet de bedoeling van deze oefening om een volledig werkende versie van het spel 4-op-een rij te implementeren. Het volstaat om enkel het inwerpen van de schijven uit te werken, m.a.w. bij elke klik wordt er afwisselend een rode en een gele schijf ingeworpen. Een detail waar je wel moet op letten is dat de schijven vallen volgens de regels van de zwaartekracht. M.a.w. klik je op een kolom, dan komt de schijf terecht op de laagst mogelijke vrije positie van het spelbord.

*met werking + CPU opponent = perfect!*