



## Java Essentials

# Hoofdstuk 10

## Overerving en klasse-hiërarchie

### DE HOGESCHOOL MET HET NETWERK

Hogeschool PXL – Elfde-Liniestraat 24 – B-3500 Hasselt  
[www.pxl.be](http://www.pxl.be) - [www.pxl.be/facebook](https://www.pxl.be/facebook)



# Inhoud

1. Inleiding
2. Subklassen definiëren in Java
3. Eigenschappen van subklassen
4. Methoden van subklassen
5. Constructors van subklassen
6. Klasse-eigenschappen en klasse-methoden
7. Final-klassen en methoden
8. Abstracte klassen
9. De superklasse Object
10. Polymorfisme
11. Code hergebruik: overerving versus associaties
12. Samenvatting



# 1. Inleiding

Wat is het voordeel van OO-programmeren?

→ Hergebruik van bestaande code

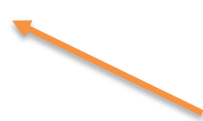
Manieren om code te hergebruiken:

- associaties: zie H9
- overerving



# 1.1 Subklassen en superklassen

Klassen kunnen gebaseerd zijn of afgeleid zijn van andere klassen



subklasse



superklasse

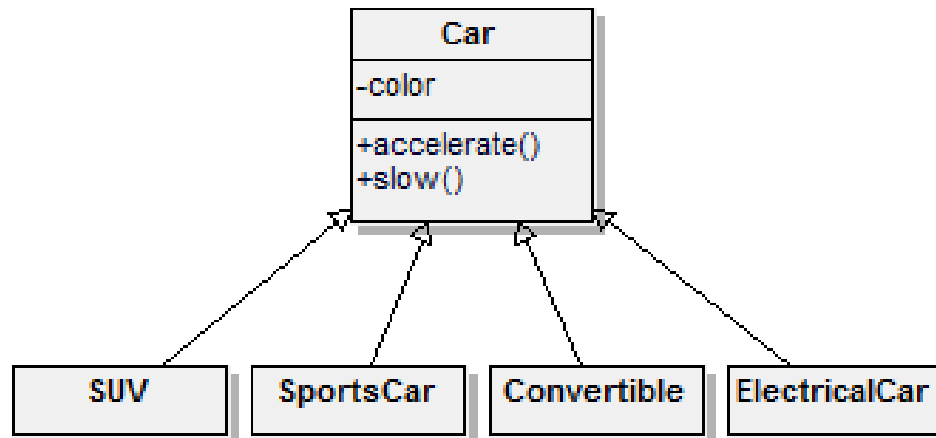
Er wordt een nieuwe klasse aangemaakt (= subklasse)  
op basis van een bestaande klasse (= superklasse)

De subklasse **erft** alle eigenschappen en methoden over van de superklasse,  
er kunnen eigenschappen en methoden **toegevoegd** worden  
en de implementatie van bepaalde methoden kan **vervangen** worden.

# 1.1 Subklassen en superklassen: voorbeeld

Voorbeeld: Car

- Veel verschillende soorten
- Van elke soort kan een klasse gemaakt worden → subklasse
- Alle gemeenschappelijke kenmerken → superklasse



Vb een Terreinwagen is een Wagen  
een Student is een Persoon  
een Hond is een Dier

overerving is een 'IS A'-relatie

## 1.2 Overerving

Subklassen zijn afgeleid van superklassen:

- Ze erven alle eigenschappen en methoden over
- Extra eigenschappen en methoden kunnen toegevoegd worden
- Er kunnen methoden vervangen worden

De code wordt in de superklasse geschreven en vervolgens herbruikt in 1 of meer subklassen.



## 1.2.1 Overerven

**Terreinwagen** is een speciaal soort auto.

Met een terreinwagen kan je alles doen wat je ook met een gewone auto kan doen.

Terreinwagen erft alle eigenschappen en methoden over van auto, vb kleur, mogelijkheid om te versnellen, remmen,...

## 1.2.2 Toevoegen

Specifieke kenmerken worden toegevoegd, vb 4x4 aandrijving



### 1.2.3 Vervangen (override)

**Een elektrische auto** is een speciaal soort auto.

Bij een gewone auto → methode `accelerate()`

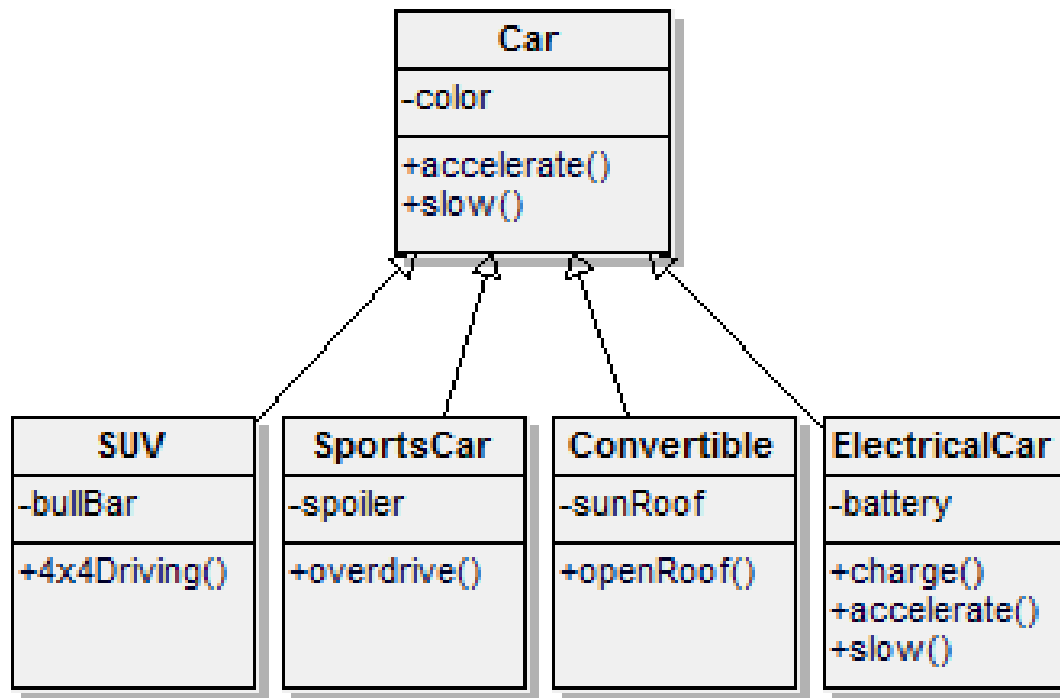
Deze methode wordt overgeërfd in de klasse `ElectricalCar`, maar kan daar vervangen worden.

Idem methode `slow()` → wordt overgeërfd, maar kan aangepast worden aan de specifieke werking van de `ElectricalCar`



## 1.2 Overerving

UML-diagram van Car met subklassen en de hierin toegevoegde en vervangen eigenschappen en methoden



## 1.3 Klasse-hiërarchie

Buiten auto's zijn er nog veel andere voertuigen (vb bus, motor,...)

Alle wegvoertuigen hebben gemeenschappelijke kenmerken

→ hiervan een superklasse maken

Wegvoertuig is een superklasse van Auto.

Auto is een superklasse van Terreinwagen.

→ Ganse hiërarchie, waarbij de subklassen telkens alles overerven van de superklassen en eigenschappen/methoden kunnen toevoegen/vervangen.



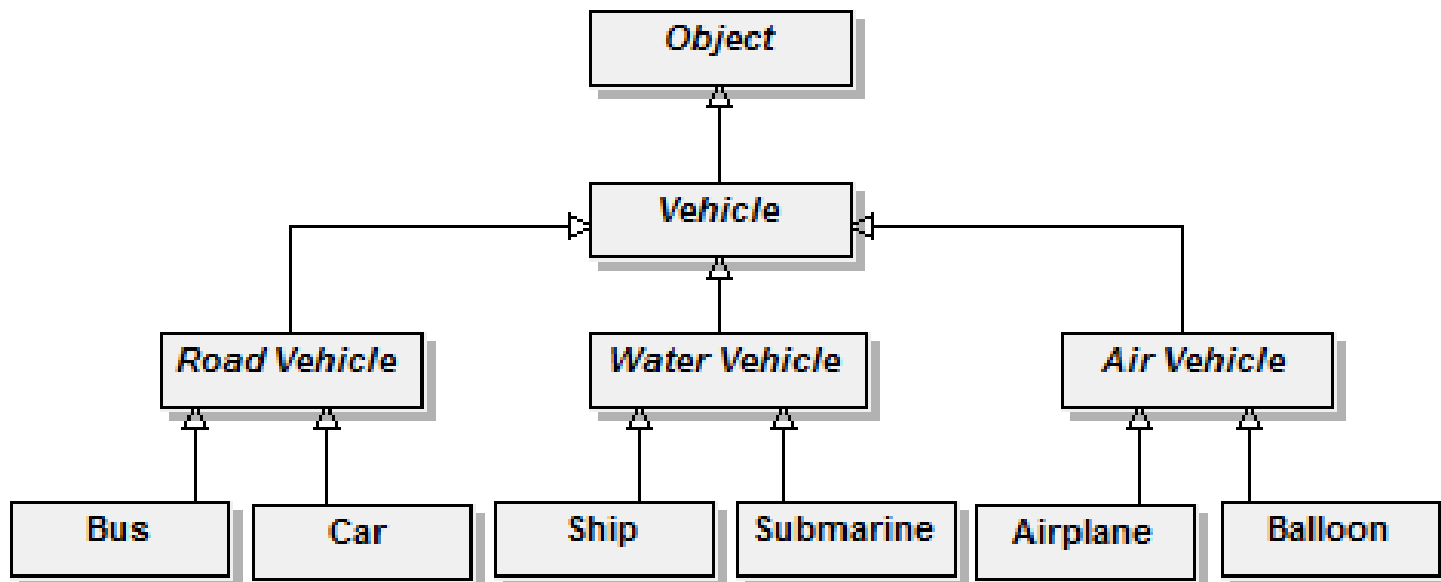
## 1.3 Klasse-hiërarchie

Vb `length` = eigenschap vd klasse `Vehicle`

wordt door elke subklasse overgeërfd

Vb `accelerate()` = methode vd klasse `Vehicle`

elke subklasse heeft een eigen implementatie van deze methode



## 1.4 Abstracte klassen

Klasse Vehicle

- Niet bedoeld om concrete objecten van te maken
- Definieert enkel gemeenschappelijke eigenschappen en methoden

=> Kan men 'abstract' definiëren (zie 10.8)



## 2. Subklassen definiëren in Java

Nieuwe klasse maken vertrekkende van een bestaande klasse: **extends**

```
class SubClass extends SuperClass {  
    ...  
}
```

Het woord 'extends' in de definitie van de subklasse, duidt aan welke de directe superklasse is.

SubClass

- Is een 'uitbreiding' van de klasse SuperClass.
- Erft eigenschappen en methoden over van SuperClass.
- Kan eigenschappen en methoden toevoegen/vervangen.



## 2. Subklassen definiëren in Java

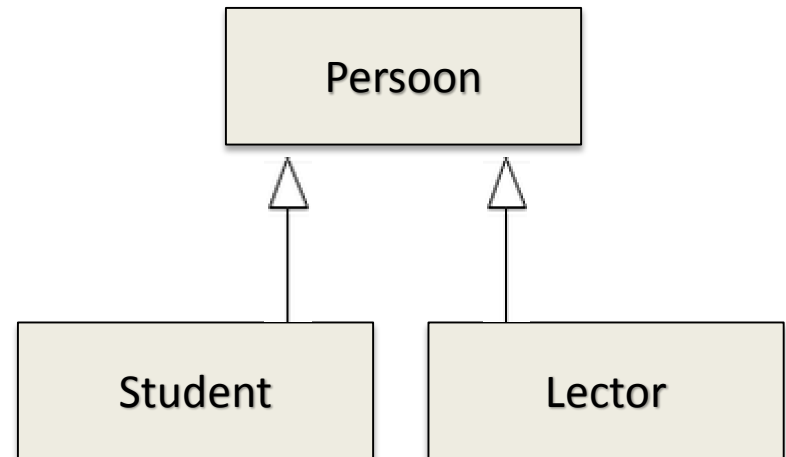
### Voorbeeld

Klasse **Persoon** met eigenschappen naam en voornaam.

Een **student** is een persoon met extra eigenschappen: studentnr, leerkrediet en opleiding.

Een **lector** is een persoon met extra eigenschappen: personeelsnr, diploma, aanstellingspercentage en salaris.

```
public class Persoon {  
    ...    //body  
}  
public class Student extends Persoon {  
    ...    //body  
}  
public class Lector extends Persoon {  
    ...    //body  
}
```



# Opdracht 1

1. Maak een project aan met naam “H10”.
2. Maak een package met naam “be.pxl.h10.voorbeeld”.
3. Maak een klasse Persoon met eigenschappen naam en voornaam. Voorzie een default constructor en een constructor met 2 parameters. Genereer de get- en set-methoden.
4. Maak een klasse Student die afgeleid is van Persoon.
5. Maak een klasse SchoolApp met een main-methode. Hierin creëer je een Persoon-object en een Student-object met de constructor zonder parameters.



# 3. Eigenschappen van subklassen

## 3.1 Overerven van eigenschappen

Subklassen erven alle **eigenschappen**/methoden over van de superklasse.  
De toegang ertoe hangt af van het toegangs niveau!

zichtbaarheid van variabele of methode				
Toegangs niveau	eigen klasse	klassen in zelfde package	subklassen (ook in ander package)	alle klassen (ook in ander package)
private	X			
- (package = default)	X	X		
protected	X	X	X	
public	X	X	X	X

Eigenschappen naam en voornaam van Persoon zijn private!

Geen rechtstreekse toegang vanuit klasse Student.





## 3.2 Toevoegen van eigenschappen

Een Student heeft extra eigenschappen tov een gewone Persoon:  
studentnr, leerkrediet, opleiding

```
public class Student extends Persoon {  
    private int leerkrediet;  
    private String studentnr;  
    private String opleiding;  
  
    ... //body  
}
```

Een Student-object heeft dus de eigenschappen:  
naam, voornaam, studentnr, leerkrediet en opleiding



## 3.3 Vervangen (verbergen) van eigenschappen



In een subklasse kan men eigenschappen definiëren met dezelfde naam als een eigenschap van de superklasse → **vermijden!!!!**

kan wel zinvol zijn bij klassevariabelen (zie later)

Vb in Persoon: variabele aantal om het aantal personen te tellen  
in Student: variabele aantal om het aantal studenten te tellen

# 4. Methoden van subklassen

## 4.1 Overerven van methoden

Subklassen erven alle **methoden** over van de superklasse:  
Zowel instance-methoden als klasse-methoden

De toegang ertoe hangt af van het toegangsniveau!

De get- en set-methoden van Persoon zijn public!  
Dus: klasse SchoolApp kan er gebruik van maken



## Opdracht 2

In de klasse SchoolApp geef je je Student-object een naam en voornaam en druk je deze af.



## 4.2 Toevoegen van methoden

Vb methode om leerkrediet te wijzigen (vermeerderen en verminderen)  
methode om leerkrediet op te vragen

```
public class Student extends Persoon {  
    private int leerkrediet;  
    private String studentnr;  
    private String opleiding;  
  
    public void setLeerkrediet (int lk) {  
        leerkrediet += lk;  
        // controle of leerkrediet tussen 0 - 140  
    }  
    public int getLeerkrediet () {  
        return leerkrediet;  
    }  
}
```



## Opdracht 3

1. Voeg aan de klasse Student methoden toe om het leerkrediet te wijzigen (een positief getal als argument zorgt voor een stijging van het leerkrediet, een negatief voor een daling). Zorg ervoor dat het leerkrediet nooit lager dan 0 kan gaan en nooit hoger dan 140.
2. Voeg een methode toe om het leerkrediet te kunnen opvragen.
3. Voeg eveneens methoden toe om het studentnr en de opleiding te kunnen wijzigen en opvragen.
4. In de klasse SchoolApp test je deze methoden uit.



## 4.3 Vervangen van methoden (override)

Vb methode om gegevens af te drukken :

In de klasse Persoon:

```
public void print() {  
    System.out.println("naam: " + voornaam + " " + naam);  
}
```

In de klasse Student:

```
public void print() {  
    super.print();  
    System.out.println("studentnr: " + studentnr);  
    System.out.println("opleiding: " + opleiding);  
    System.out.println("leerkrediet: " + leerkrediet);  
}
```



## 4.3 Vervangen van methoden

### Opmerking1: signatuur

Methoden die vervangen worden moeten **dezelfde signatuur** hebben!

**signatuur** = naam vd methode + aantal en type van de parameters

Bij methoden die vervangen worden moet ook het terugkeertype hetzelfde zijn (maar voor objecten mag het ook een subklasse zijn).

Als de signatuur niet hetzelfde is, wordt de methode niet vervangen, maar wordt een methode toegevoegd.

**@Override** → **compiler controleert!**





## 4.3 Vervangen van methoden

### Opmerking2: toegangsniveau

Subklassen kunnen het toegangsniveau van overschreven methoden enkel vergroten, niet verkleinen!

Vb methode `getNaam()` is public in `Persoon`

→ bij overschrijven in `Student` mag je ze niet private maken

Private methoden worden overgeërfd, maar zijn niet toegankelijk in de subklasse.



## 4.3 Vervangen van methoden

Opmerking3: verschil tussen overridding en overloading



## Opdracht 4

Voeg aan de klasse Persoon en Student methoden toe om de gegevens af te drukken en test deze uit in de klasse SchoolApp.

Voeg ook de annotatie `@Override` toe en test uit.



## 4.4 Polymorfisme

### Voorbeeld

```
public static void main (String[] args) {  
    Persoon pol = new Student();  
    pol.setNaam("Jans");  
    System.out.println("naam: " + pol.getNaam());  
}
```

pol is een polymorf object!

→ gedeclareerd als Persoon

→ gecreëerd als Student

Dit kan omdat elke Student ook een Persoon is.



## 4.4 Polymorfisme

### Voorbeeld

```
public static void main (String[] args) {  
    Persoon pol = new Student();  
    pol.setNaam("Jans");  
    System.out.println("naam: " + pol.getNaam());  
    pol.setOpleiding("TIN"); //compilation error!!  
}
```

### Opgepast!

object pol: je kan enkel methoden gebruiken die bestaan in de klasse Persoon

### Waarom?

in de loop van de code kan je de variabele wijzigen: `pol = new Persoon();`

→ alles blijft werken



## 4.4 Polymorfisme

Voorbeeld: wat als we de methode print() aanroepen?  
print() bestaat zowel in Persoon als in Student

```
public static void main (String[] args) {  
    Persoon pol = new Student();  
    pol.setNaam("Jans");  
    pol.print();  
}
```

de print-methode van Student wordt opgeroepen!

De koppeling van de uit te voeren code gebeurt pas tijdens de uitvoering van het programma → late binding

Tijdens compilatie is niet geweten welke methode opgeroepen gaat worden, want misschien is de inhoud van de variabele pol wel afhankelijk van de keuze van de gebruiker.



## Opdracht 5a

Maak in de klasse SchoolApp een polymorf object en test uit welke methoden je kan gebruiken. Bekijk de output van de methode print().

## Opdracht 5b

1. Maak een klasse Lector aan die afgeleid is van Persoon. Een Lector heeft (buiten een naam en een voornaam) een personeelsnr, een aanstellingspercentage en een salaris.
2. Voorzie get- en set-methoden voor de toegevoegde eigenschappen.  
Zorg ervoor dat het aanstellingspercentage altijd tussen 0 en 100% ligt. Zorg ervoor dat als het aanstellingspercentage wijzigt, het salaris overeenkomstig aangepast wordt (let op: dit mag niet gebeuren bij een eerste ingave! Enkel als het salaris en het aanstellingspercentage reeds een waarde hadden, zal bij een wijziging van het aanstellingspercentage het salaris mee gewijzigd worden).
3. Voorzie een methode print() die alle gegevens van een lector afdrukt.
4. In de klasse SchoolApp test je deze methoden uit.



# 5. Constructors van subklassen

Constructors worden NIET overgeërfd van superklassen.

Indien geen expliciete constructor gedefinieerd wordt, is de default constructor aanwezig, maar enkel als ook de superklasse een default constructor heeft.

In klassen Lector en Student → geen constructor gedefinieerd.

→ compiler maakt standaard constructor

→ deze roept de standaard constructor van Persoon op

In feite gebeurt het volgende:

```
public class Student extends Persoon {
```

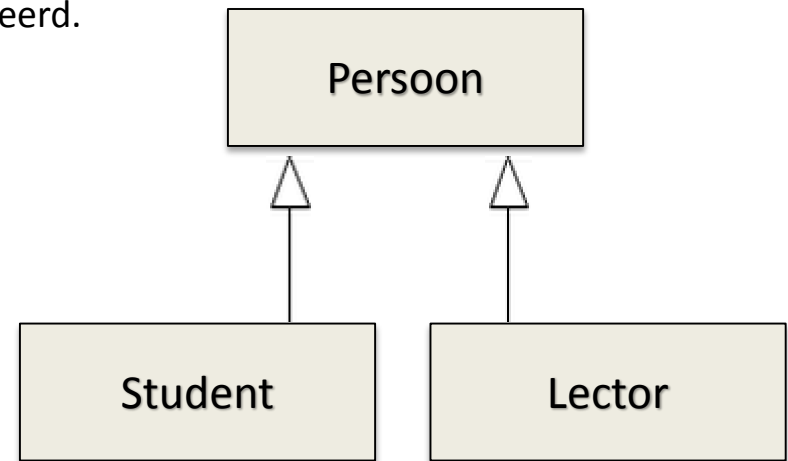
```
    ...    //eigenschappen
```

```
    public Student() {  
        super();  
    }
```

```
    ...    //body
```

```
}
```

super() roept de  
default constructor  
van de superklasse op.

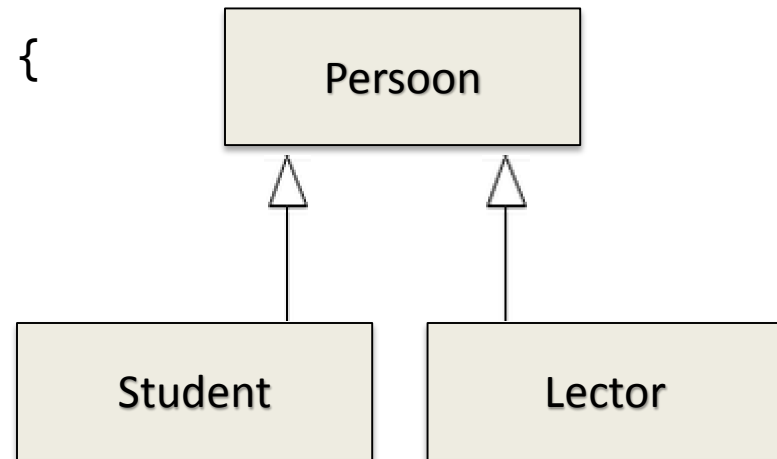




# 5. Constructors van subklassen

Voor de klasse Student maken we expliciet volgende constructors:

```
public class Student extends Persoon {  
  
    ...    //eigenschappen  
  
    public Student() {  
        this("onbekend", "onbekend");  
    }  
  
    public Student(String n, String vn) {  
        super(n, vn);  
    }  
    ...    //body  
}
```

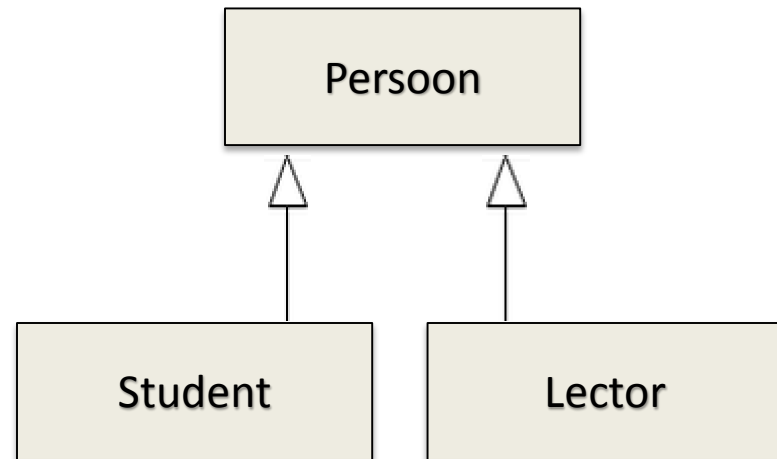


**super(n, vn) roept de constructor met 2 parameters op van Persoon.**

# 5. Constructors van subklassen

## opmerkingen bij super():

- roept de constructor van de superklasse op
- dit moet de eerste instructie in de constructor zijn
- als de constructor van de superklasse niet expliciet opgeroepen wordt, wordt automatisch de default constructor opgeroepen → werkt enkel als de superklasse zo'n constructor heeft!
- De aanroep van de constructor van de superklasse gebeurt vóór het uitvoeren van andere statements in de constructor van de subklasse. (uitz: wanneer de constructor een andere constructor vd subklasse oproept)



# Opdracht 6

## 1. Voeg aan de klasse Student volgende constructors toe:

- ❖ 1 zonder parameters: de naam en voornaam krijgen de waarde “onbekend”.
- ❖ 1 met naam en voornaam als parameters.
- ❖ 1 met naam, voornaam, studentnr en opleiding als parameters.

Laat de constructors met minder parameters de constructor met de meeste parameters oproepen.

Indien het studentnr niet gekend is wordt dit ingesteld op 15999999.

Indien de opleiding niet gekend is wordt dit ingesteld op “XXX”.

Het leerkrediet wordt standaard ingesteld op 140.

## 2. Voeg aan de klasse Student ook een constructor toe die een Student-object maakt op basis van een bestaande student.

## 3. Voeg aan de klasse Lector volgende constructors toe:

- ❖ 1 met 5 parameters.
- ❖ 1 met 4 parameters (alles behalve personeelsnr).
- ❖ 1 zonder parameters.

Default waarden voor de eigenschappen zijn: “onbekend” voor de naam en de voornaam, 20009999 voor het personeelsnr, 2000 voor salaris en 100% aanstelling.

## 4. In de klasse SchoolApp maak je verschillende studenten en lectoren aan gebruik makend van de verschillende constructors.

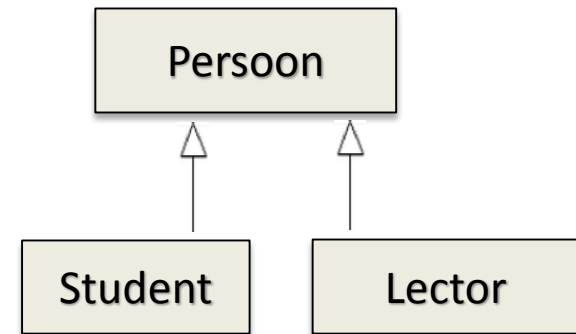


# 6. Klasse-eigenschappen en klasse-methoden

Ook klasse-eigenschappen en klasse-methoden worden overgeërfd van superklassen.

voorbeeld:

```
public class Persoon {  
  
    ... //eigenschappen  
    private static int aantal;  
  
    {  
        aantal++;  
    }  
  
    public static int getAantal() {  
        return aantal;  
    }  
  
    ... //body  
}
```




bij de creatie van een Persoon wordt aantal opgehoogd.

# 6. Klasse-eigenschappen en klasse-methoden

In de main-methode kunnen we het aantal personen opvragen als volgt:

```
public static void main (String[] args) {  
    System.out.println("aantal : " + Persoon.getAantal());  
    System.out.println("aantal : " + Student.getAantal());  
}
```



**Opgelet: het gaat hier wel degelijk over het aantal personen, niet om het aantal studenten!**

Indien je het aantal studenten wil kennen, moeten in de klasse Student de eigenschap "aantal" en de methode "getAantal()" geherdefinieerd worden!

# 6. Klasse-eigenschappen en klasse-methoden



Statische methoden kan men ook oproepen op een object ipv op een klasse → **vermijden!!!!**

**zeker vermijden bij objecten van subklassen!!**

## Opdracht 7

1. Doe het nodige in de klasse Persoon om te kunnen bijhouden hoeveel persoon-objecten gecreëerd werden.
2. In de klasse SchoolApp test je de methode getAantal() uit op de klasse Persoon en op de klasse Student.
3. Doe het nodige in de klassen Student en Lector om te kunnen bijhouden hoeveel student- en lector-objecten gecreëerd werden.
4. Test alles uit in de klasse SchoolApp.



# *Oefening 1*





# 7. Final-klassen en methoden

## Final-klassen

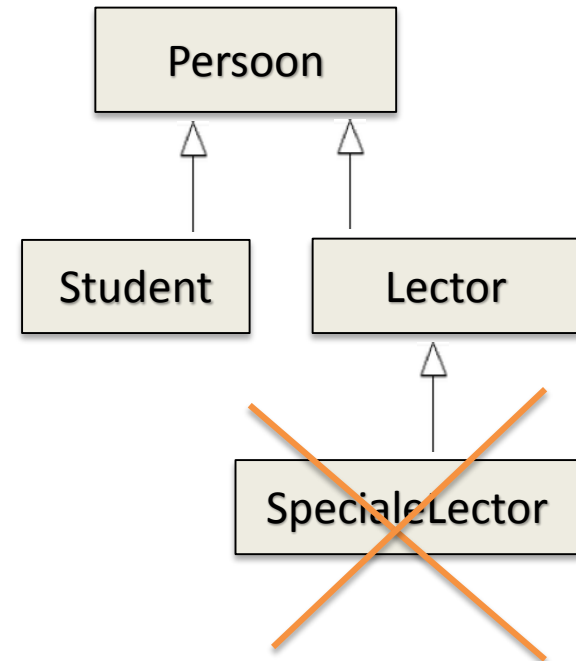
→ KUNNEN GEEN subklassen hebben.

voorbeeld:

```
public final class Lector extends Persoon {  
    ...    //body  
}
```

```
public class SpecialeLector extends Lector
```

compilation error



# 7. Final-klassen en methoden

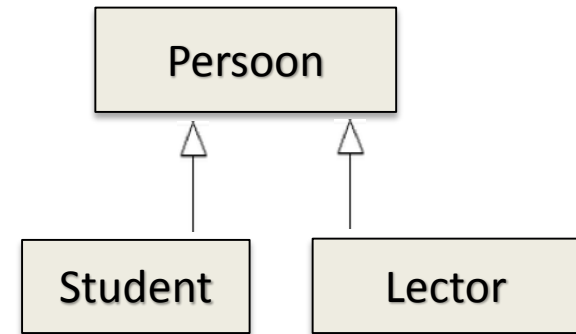
## Final-methoden

→ KUNNEN NIET overschreven worden in subklassen.

voorbeeld:

```
public class Persoon {  
    public final String getNaam() {  
        return naam;  
    }  
    ... //body  
}
```

```
public class Lector extends Persoon {  
    public String getNaam() ...  
}
```



compilation error



# 7. Final-klassen en methoden

3 belangrijke redenen voor final-klassen en methoden:

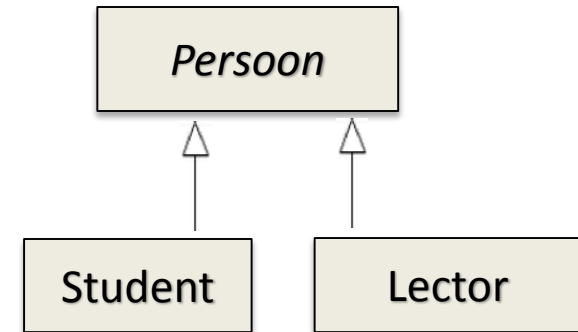
- Beveiliging: men kan geen subklassen maken met een heel andere implementatie
- Ontwerp: bij het ontwerp van de klasse-hiërarchie kan men beslissen dat bepaalde klassen “af” zijn
- Snelheid: bij final-klassen of methoden weet de compiler vaak reeds bij compilatie welke implementatie van een methode gebruikt wordt, dus deze beslissing moet niet meer genomen worden tijdens uitvoering van het programma (late binding)



# 8. Abstracte klassen

## abstracte klassen

- er KUNNEN GEEN objecten van gemaakt worden
- ze dienen om zoveel mogelijk gemeenschappelijke code onder te brengen die vervolgens overgeërfd wordt door de subklassen



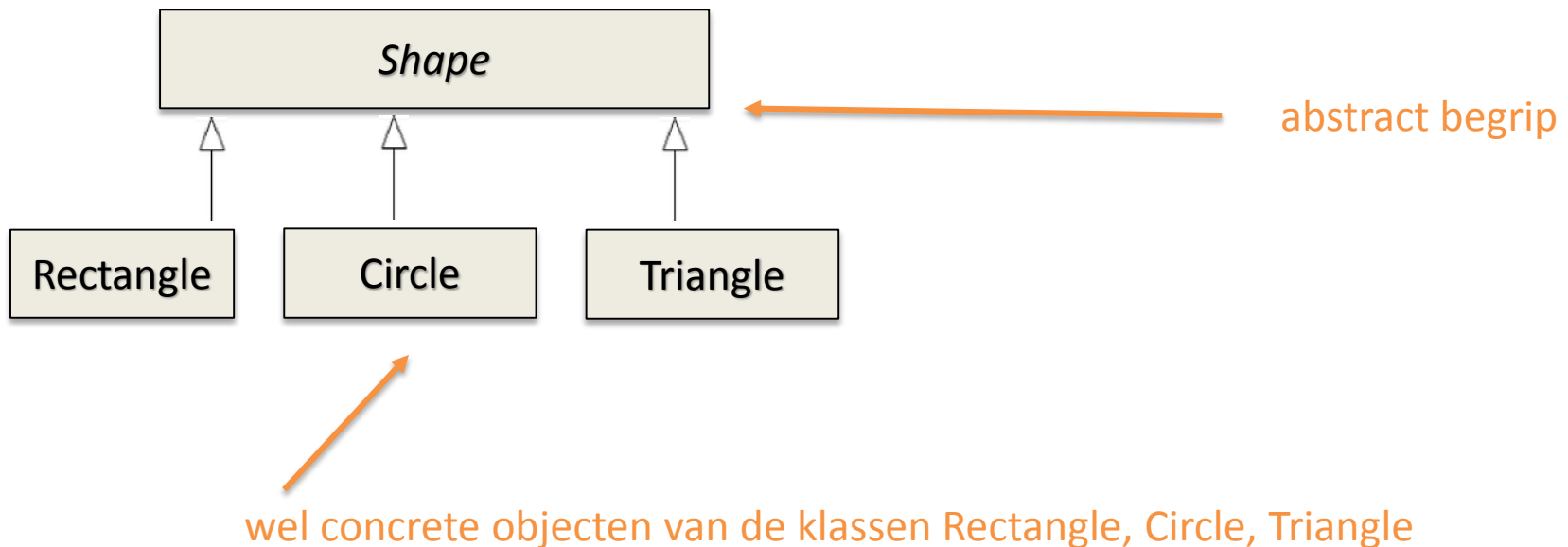
voorbeeld:

```
public abstract class Persoon {  
    ... //body  
}
```

kan abstract, want een gewoon Persoon-object gaat niet gecreëerd worden; wel lectoren, studenten, administratief personeel, leveranciers, ...

# 8. Abstracte klassen

voorbeeld2: in een tekenprogramma wordt gebruik gemaakt van volgende structuur



# Klasse *Shape*

is een abstract begrip: er bestaan geen rechtstreekse objecten van deze klasse

elke figuur heeft:

- (x, y)-coördinaat
- getters & setters voor x en y
- van elke figuur **moet** de oppervlakte kunnen berekend worden
- van elke figuur **moet** de omtrek berekend kunnen worden



```
public abstract class Shape {  
  
    private int x, y;  
  
    public Shape (int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public void setPosition (int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public int getX() {  
        return x;  
    }  
    public int getY() {  
        return y;  
    }  
    public abstract double getArea();  
    public abstract double getPerimeter();  
}
```

abstracte klasse:

er kunnen geen objecten van  
deze klasse gemaakt worden



```
public abstract class Shape {  
  
    private int x, y;  
  
    public Shape (int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public void setPosition (int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public int getX() {  
        return x;  
    }  
    public int getY() {  
        return y;  
    }  
    public abstract double getArea();  
    public abstract double getPerimeter();  
}
```

eigenschappen:  
elke figuur heeft een  
(x,y)-coördinaat





```
public abstract class Shape {  
  
    private int x, y;  
  
    public Shape (int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public void setPosition (int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public int getX() {  
        return x;  
    }  
    public int getY() {  
        return y;  
    }  
    public abstract double getArea();  
    public abstract double getPerimeter();  
}
```

### constructor:

er kunnen geen objecten van deze klasse gemaakt worden, maar er is toch een constructor!

Deze constructor wordt gebruikt via keyword 'super' in de afgeleide klassen (Rectangle, Circle, Triangle)



```
public abstract class Shape {  
  
    private int x, y;  
  
    public Shape (int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public void setPosition (int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public int getX() {  
        return x;  
    }  
    public int getY() {  
        return y;  
    }  
    public abstract double getArea();  
    public abstract double getPerimeter();  
  
}
```

setters en getters voor x, y  
deze methodes kunnen in deze  
klasse geïmplementeerd  
worden



```
public abstract class Shape {  
  
    private int x, y;  
  
    public Shape (int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public void setPosition (int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public int getX() {  
        return x;  
    }  
    public int getY() {  
        return y;  
    }  
    public abstract double getArea();  
    public abstract double getPerimeter();  
}
```

Niet geweten: hoe berekent  
men de omtrek van een figuur?  
hoe berekent men de  
oppervlakte van een figuur?

→ methodes `getArea()` en  
`getPerimeter()` abstract



```

public abstract class Shape {

    private int x, y;

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

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

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public abstract double getArea();
    public abstract double getPerimeter();

}

```

methods `getArea()` en `getPerimeter()` abstract

***wel definitie:***

- toegangsniveau
- terugkeertype
- naam van de method
- parameters

***geen body!***

- geen { }
- wel ;



```
public abstract class Shape {  
  
    private int x, y;  
  
    public Shape (int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public void setPosition (int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public int getX() {  
        return x;  
    }  
    public int getY() {  
        return y;  
    }  
    public abstract double getArea();  
    public abstract double getPerimeter();  
}
```

methods `getArea()` en  
`getPerimeter()` abstract

*geen implementatie:*

de implementatie MOET  
voorzien worden in elke (niet-  
abstracte) subklasse van deze  
klasse



# Klasse *Rectangle*

van deze klasse kunnen concrete objecten gemaakt worden

elke Rectangle heeft:

- (x, y)-coördinaat, height, width
- van elke Rectangle kan je de oppervlakte berekenen
- van elke Rectangle kan je de omtrek berekenen



```
public class Rectangle extends Shape {  
    private int height, width;  
  
    public Rectangle (int x, int y,  
                      int h, int w) {  
        super(x, y);  
        height = h;  
        width = w;  
    }  
  
    //andere methoden  
  
    public double getArea() {  
        return width * height;  
    }  
  
    public double getPerimeter() {  
        return 2 * (width + height);  
    }  
}
```

extends:

Rectangle is een subklasse van Shape



```
public class Rectangle extends Shape {  
  
    private int height, width;  
  
    public Rectangle (int x, int y,  
                      int h, int w) {  
        super(x, y);  
        height = h;  
        width = w;  
    }  
  
    //andere methoden  
  
    public double getArea() {  
        return width * height;  
    }  
  
    public double getPerimeter() {  
        return 2 * (width + height);  
    }  
}
```

extra eigenschappen:  
height en width





```
public class Rectangle extends Shape {  
  
    private int height, width;  
  
    public Rectangle (int x, int y,  
                      int h, int w) {  
        super(x, y);  
        height = h;  
        width = w;  
    }  
  
    //andere methoden  
  
    public double getArea() {  
        return width * height;  
    }  
  
    public double getPerimeter() {  
        return 2 * (width + height);  
    }  
}
```

constructor:

oproep van de constructor van  
Shape  
→ moet steeds op de eerste  
regel



```
public class Rectangle extends Shape {  
  
    private int height, width;  
  
    public Rectangle (int x, int y,  
                      int h, int w) {  
        super(x, y);  
        height = h;  
        width = w;  
    }  
  
    //andere methoden  
  
    public double getArea() {  
        return width * height;  
    }  
  
    public double getPerimeter() {  
        return 2 * (width + height);  
    }  
}
```

toevoegen van extra methoden:  
vb getHeight, setHeight,  
getWidth, setWidth, ...

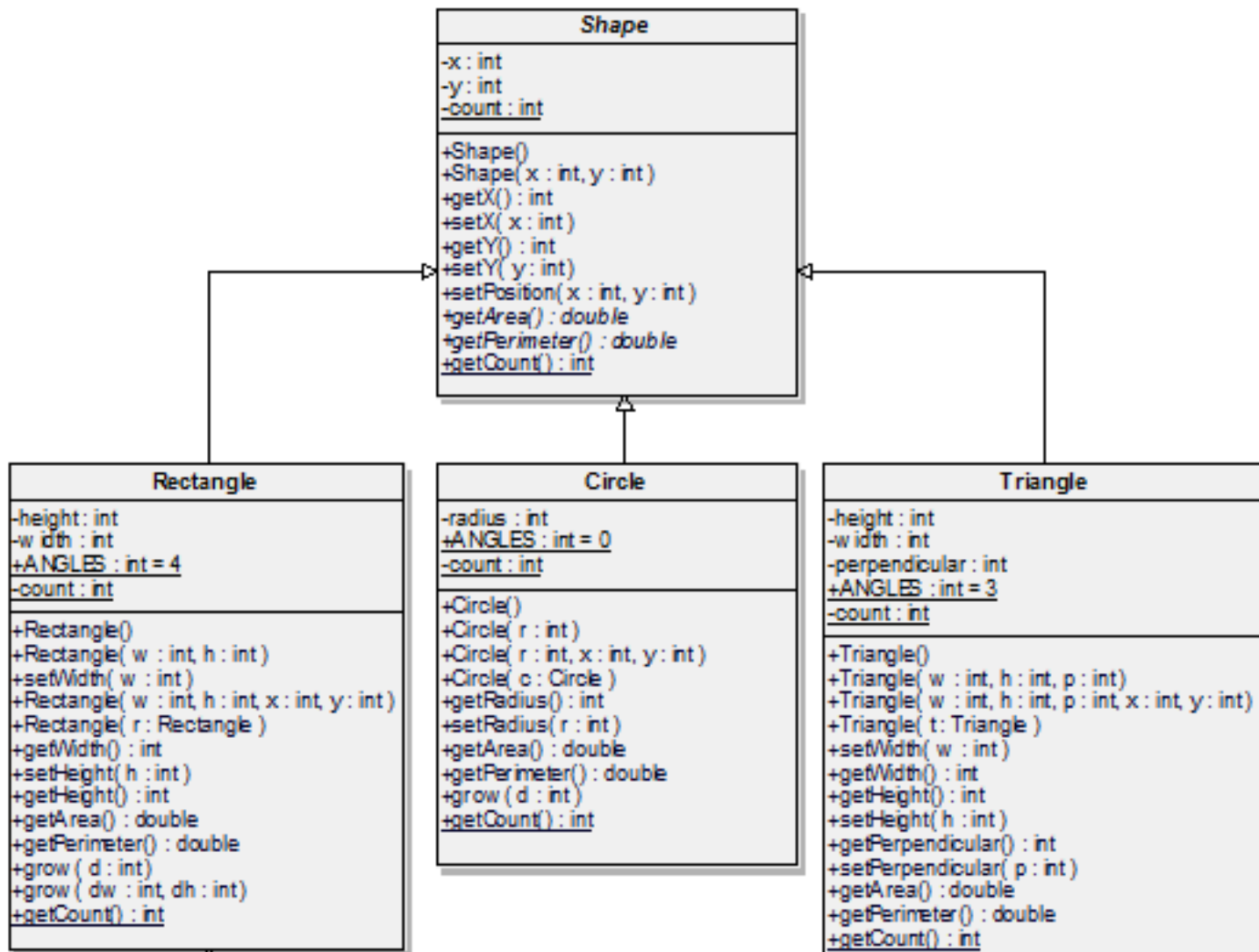


```
public class Rectangle extends Shape {  
  
    private int height, width;  
  
    public Rectangle (int x, int y,  
                      int h, int w) {  
        super(x, y);  
        height = h;  
        width = w;  
    }  
  
    //andere methoden  
  
    public double getArea() {  
        return width * height;  
    }  
  
    public double getPerimeter() {  
        return 2 * (width + height);  
    }  
}
```

de abstracte methoden  
getArea() en getPerimeter()  
kunnen (en moeten) hier  
geïmplementeerd worden



De klasse-hiërarchie van figuren zou er als volgt kunnen uitzien:



# Opmerking abstracte methodes

Enkel abstracte klassen kunnen abstracte methoden hebben.

Abstracte methoden hebben geen implementatie in de abstracte klasse, maar moeten wel geïmplementeerd worden in de eerstvolgende concrete klasse.

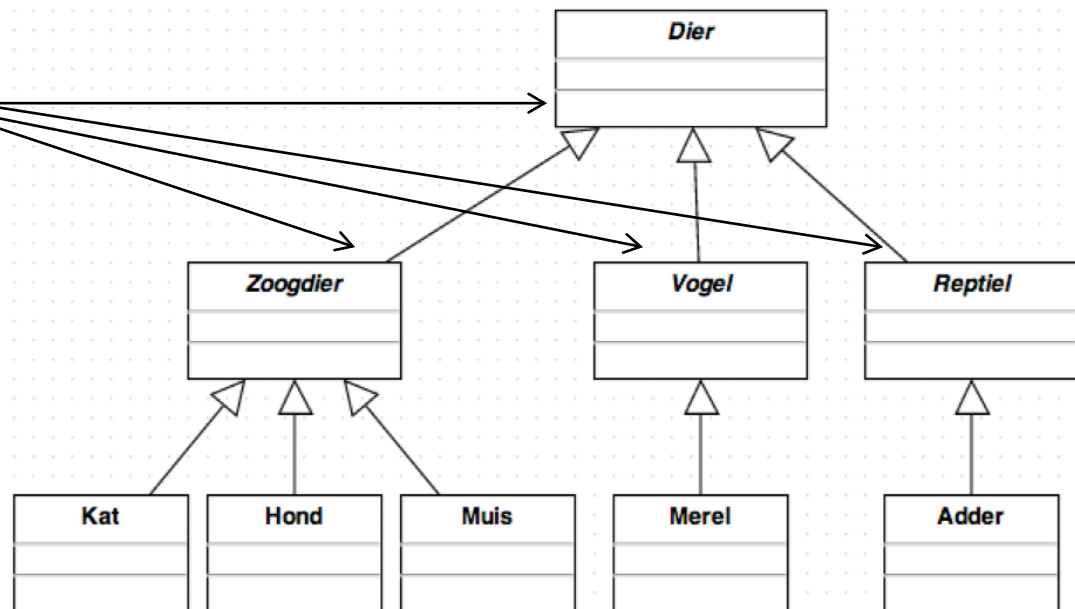
Dus: een subklasse van een abstracte klasse, die zelf ook abstract is, moet niet alle abstracte methoden implementeren.



# Voorbeeld implementatie abstracte methoden

abstracte methodes moeten in een (niet-abstracte) subklasse geïmplementeerd worden

abstract



```
public abstract class Dier {  
    public abstract void maakGeluid();  
}  
  
public abstract class Zoogdier extends Dier {  
}  
  
public class Kat extends Zoogdier {  
    public void maakGeluid() {  
        System.out.println("miauw");  
    }  
}
```

Dier is abstract

- abstracte methode  
maakGeluid()



```
public abstract class Dier {  
    public abstract void maakGeluid();  
}  
  
public abstract class Zoogdier extends Dier {  
}  
  
public class Kat extends Zoogdier {  
    public void maakGeluid() {  
        System.out.println("miauw");  
    }  
}
```

Zoogdier is abstract

- afgeleid van Dier
- geen implementatie van maakGeluid()





```
public abstract class Dier {  
    public abstract void maakGeluid();  
}  
  
public abstract class Zoogdier extends Dier {  
}  
  
public class Kat extends Zoogdier {  
    public void maakGeluid() {  
        System.out.println("miauw");  
    }  
}
```

Kat is niet abstract

- afgeleid van Zoogdier
- hier moet maakGeluid() geïmplementeerd worden



## Opdracht 8

1. Maak een package “be.pxl.h10.opdracht8\_shapes”.
2. Maak een abstracte klasse Shape zoals beschreven in dia 47 (eigenschappen: x, y; methoden: setPosition(), getX(), getY(); abstracte methoden getArea(), getPerimeter()).
3. Doe het nodige om het aantal aangemaakte figuren te kunnen opvragen.
4. Maak een klasse Circle (afgeleid van Shape) met als eigenschap de straal, en methoden om de straal te kunnen instellen en opvragen.  
Maak een constructor met 3 parameters (voor x, y en straal).  
Voor de berekening van de omtrek en de oppervlakte maak je gebruik van de klasse Math.
5. Maak een klasse ShapeApp met een main-methode waarin je alles uittest.  
Druk de omtrek en de oppervlakte van de cirkel af met 2 decimalen.



# 9. De superklasse Object

## 9.1 Klasse-hiërarchie

De superklasse van alle klassen is de klasse **Object**.

Iedere klasse is rechtstreeks of onrechtstreeks een subklasse van Object.

Dus als je een klasse definieert als:

```
public class A {  
    ...  
}
```

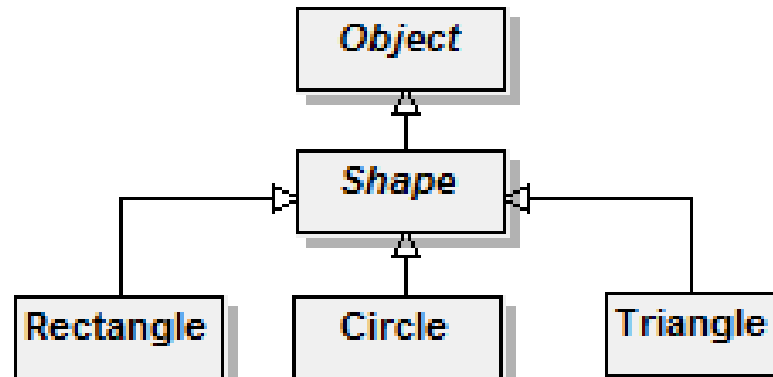
is dit eigenlijk hetzelfde als:

```
public class A extends Object {  
    ...  
}
```



## 9.1 Klasse-hiërarchie

Hierdoor ziet de klasse-hiërarchie van het vorige voorbeeld er eigenlijk uit als volgt:



## 9.2 De operator instanceof

Iedere klasse behoort dus tot dezelfde boomstructuur waarbij het hoogste element de klasse Object is.

Met de operator instanceof kan je nagaan of een object een instantie is van 1 of andere (super)klasse.

Voorbeeld:

```
Rectangle r = new Rectangle();
System.out.println(r instanceof Object);           //true
System.out.println(r instanceof Shape);            //true
System.out.println(r instanceof Rectangle);        //true
System.out.println(r instanceof Square);           //false
System.out.println(r instanceof Circle);           //Compilation error!!
```

instanceof werkt enkel bij objecten die in dezelfde hiërarchische tak zitten als de klasse.

instanceof kan gebruikt worden voor veilige “downcast”



## 9.3 Methoden van de Object-klasse

De methoden van de klasse Object worden door ALLE klassen overgeërfd. Sommige van deze methoden zijn final en kunnen dus niet vervangen worden.

Enkele methoden:

<b>Returntype</b>	<b>Methode</b>	<b>Beschrijving</b>
String	<code>toString()</code>	Geeft een string terug waarin het object beschreven wordt.
boolean	<code>equals(Object o)</code>	Geeft aan of twee objecten aan elkaar gelijk zijn. Dit is het geval indien de referentievariabelen gelijk zijn.
int	<code>hashCode()</code>	Geeft de hashcode van het object.
Class	<code>getClass()</code>	Geeft een object terug dat de concrete klasse voorstelt.

## 9.3 Methoden van de Object-klasse `getClass()`

Class	<code>getClass()</code>	Geeft een object terug dat de concrete klasse voorstelt.
-------	-------------------------	----------------------------------------------------------

Voorbeeld in de main-methode van shapeApp:

```
Circle cir1 = new Circle(10, 20, 50);  
System.out.println(cir1.getClass());
```

Output:

```
class be.pxl.opdracht8_shapes.Circle
```



## 9.3.1 toString()

String	toString()	Geeft een string terug waarin het object beschreven wordt.
--------	------------	------------------------------------------------------------

- geeft een tekstuele voorstelling van het object
- vrij cryptisch
- het is gebruikelijk om in klassen deze methode te vervangen

Kan gebruikt worden bij het afdrukken van een object:

```
Rectangle rect = new Rectangle(4, 5, 7, 10);  
System.out.println(rect);
```

Output:

be.pxl.opdracht8\_shapes.Rectangle@12204a1

impliciet wordt de methode `toString()` opgeroepen. Doordat `toString()` op het hoogste niveau gedefinieerd is, kan elk object zo gedrukt worden





## 9.3.1 toString()

String	toString()	Geeft een string terug waarin het object beschreven wordt.
--------	------------	------------------------------------------------------------

Zinvollere implementatie:

```
public class Rectangle extends Shape {  
    public String toString() {  
        return String.format(  
            "Rectangle with size (%d, %d) at position (%d, %d)",  
            width, height, getX(), getY());  
    }  
    ...  
}
```

IDE's hebben vaak wizards om deze methoden te genereren.

## Opdracht 9

1. Gebruik de code van de package “be.pxl.opdracht8\_shapes”.
2. In de klasse ShapeApp druk je je Circle-object af (1 keer zonder toString() en 1 keer met toString()).
3. Voeg in de klasse Circle een methode toString() toe waarbij een zinvolle beschrijving van het object wordt teruggegeven.  
Maak eventueel gebruik van de wizard van je IDE.
4. Voer het programma opnieuw uit.



## 9.3.2 equals() en hashCode()

boolean	equals (Object o)	Geeft aan of twee objecten aan elkaar gelijk zijn. Dit is het geval indien de referentievariabelen gelijk zijn.
int	hashCode ()	Geeft de hashcode van het object.

**equals:** nagaan of 2 objecten hetzelfde zijn

in de **standaard implementatie** betekent dit: 2 objecten worden als hetzelfde beschouwd als het gaat om hetzelfde object, m.a.w. als de referentievariabelen hetzelfde zijn.

Deze standaard implementatie ziet er als volgt uit:

```
public boolean equals (Object o) {  
    return this == o;  
}
```

Verschillende objecten met dezelfde inhoud worden als verschillend beschouwd.  
Om deze toch als gelijk te beschouwen → equals() vervangen



## 9.3.2 equals() en hashCode()

boolean	equals (Object o)	Geeft aan of twee objecten aan elkaar gelijk zijn. Dit is het geval indien de referentievariabelen gelijk zijn.
int	hashCode ()	Geeft de hashcode van het object.

**Eigen implementatie van equals:** meestal wordt nagegaan of de 2 objecten van hetzelfde type zijn en of de relevante gegevens hetzelfde zijn.

**Voorbeeld voor klasse Rectangle:**

```
public class Rectangle extends Shape {
    ...
    @Override
    public boolean equals(Object o) {
        if((o != null) &&
            (o.getClass() == this.getClass()) &&
            ((Rectangle)o).getX() == getX() &&
            ((Rectangle)o).getY() == getY() &&
            ((Rectangle)o).height == height &&
            ((Rectangle)o).width == width))
            return true;
        else return false;
    }
    ...
}
```

## Opdracht 10

1. Gebruik de code van de package “be.pxl.opdracht8\_shapes”.
2. In de klasse ShapeApp maak je een nieuw Circle-object met net dezelfde waarden voor de eigenschappen als een vorig gemaakte Circle. Test de methode equals uit.
3. We beschouwen 2 Circle-objecten als dezelfde als hun straal, x en y dezelfde waarden hebben. Doe het nodige hiervoor.
4. Voer het programma opnieuw uit.



# 10. Polymorfisme (bis)

Polymorfisme = een object dat veel vormen kan aannemen

- gedeclareerd als type van een superklasse
- gecreëerd als type van een subklasse

Stel dat we alle figuren in 1 array willen bijhouden

```
Shape [] shapes = new Shape[5];
```

- array van referenties naar Shape-objecten
- er bestaan geen rechtstreekse objecten van de klasse Shape (abstract!)
- er bestaan wel objecten van de subklassen van Shape



# 10. Polymorfisme (bis)

```
public class ShapesApp {
    public static void main(String[] args) {
        Shape[] shapes = new Shape[5];
        shapes[0] = new Square(7);
        shapes[1] = new Circle(8);
        shapes[2] = new Rectangle(6,9);
        shapes[3] = new Square(7);
        shapes[4] = new Rectangle(8,2);
        for(Shape shape: shapes) {
            printShape(shape);
        }
    }

    private static void printShape(Shape s){
        System.out.print(" Area "+ s.getArea());
        System.out.print(" Perimeter "+ s.getPerimeter());
        System.out.print(" Position (" + s.getX());
        System.out.println(", " + s.getY() + ")");
    }
}
```

getArea() en getPerimeter() zijn in Shape gedefinieerd

→ beschikbaar voor alle objecten van een superklasse

→ welke methode precies gebruikt wordt, wordt beslist tijdens de uitvoering van het programma (late binding)



# Opdracht 11

1. Importeer de klassen Rectangle en Triangle in de package “be.pxl.opdracht8\_shapes”.
2. Maak in de main-methode van ShapeApp een array met 5 figuren. Doorloop de array en maak een afdruk als volgt:

omtrek:	314	oppervlakte:	7854	Cirkel
omtrek:	314	oppervlakte:	7854	Cirkel
omtrek:	60	oppervlakte:	200	Rechthoek
omtrek:	240	oppervlakte:	2000	Rechthoek
omtrek:	73	oppervlakte:	50	Driehoek





# 11. Code hergebruik: overerving versus associaties

Doel objectgeoriënteerd programmeren: het hergebruiken van bestaande code

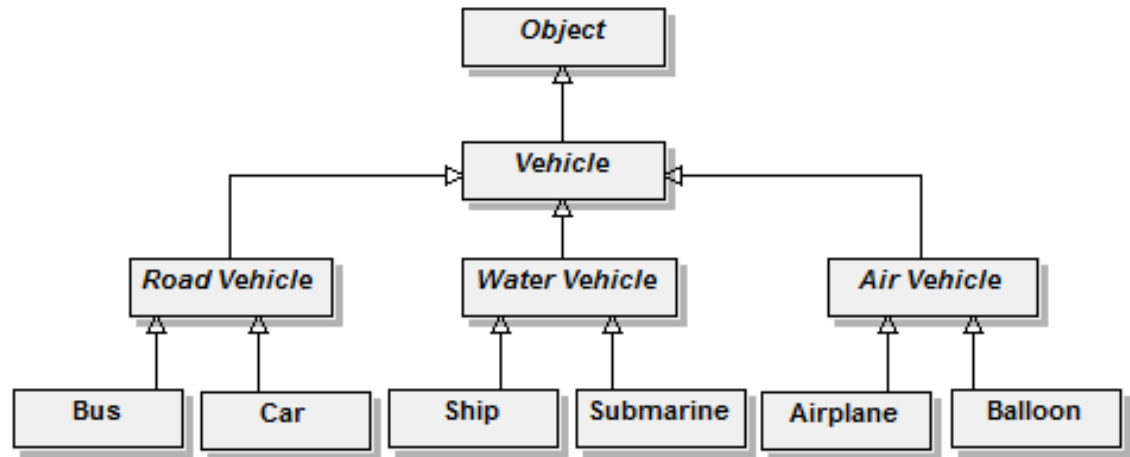
Dit kan op 2 manieren:

- 1) **Associaties**: een klasse heeft een relatie met een andere klasse  
→ “HAS a”-relatie  
Vb een logo **heeft een** rechthoek, een persoon **heeft een** adres,....
- 2) **Overerving**: een klasse erft over van een superklasse  
→ “IS A”-relatie  
Vb een student **is een** persoon, een rechthoek **is een** figuur,...  
Zo’n relatie kan getest worden via “instanceof”



# 11. Code hergebruik: overerving versus associaties

Voorbeeld voertuigen:



Een auto **is een** wegvoertuig, een bus **is een** wegvoertuig → overerving

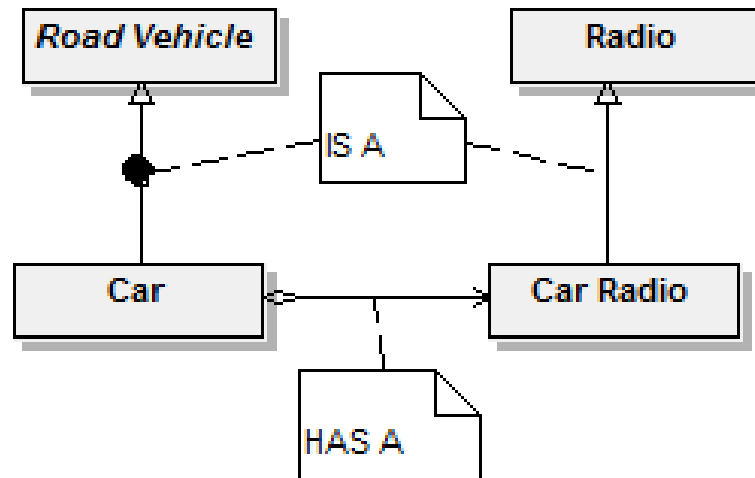
Stel dat we een auto en een bus willen voorzien van een radio.

We kunnen dan een bestaand toestel (autoradio) inbouwen → associatie

Deze autoradio zou kunnen afgeleid zijn van de klasse radio → overerving

# 11. Code hergebruik: overerving versus associaties

Voorbeeld voertuigen:



Een auto **is een** wegvoertuig.

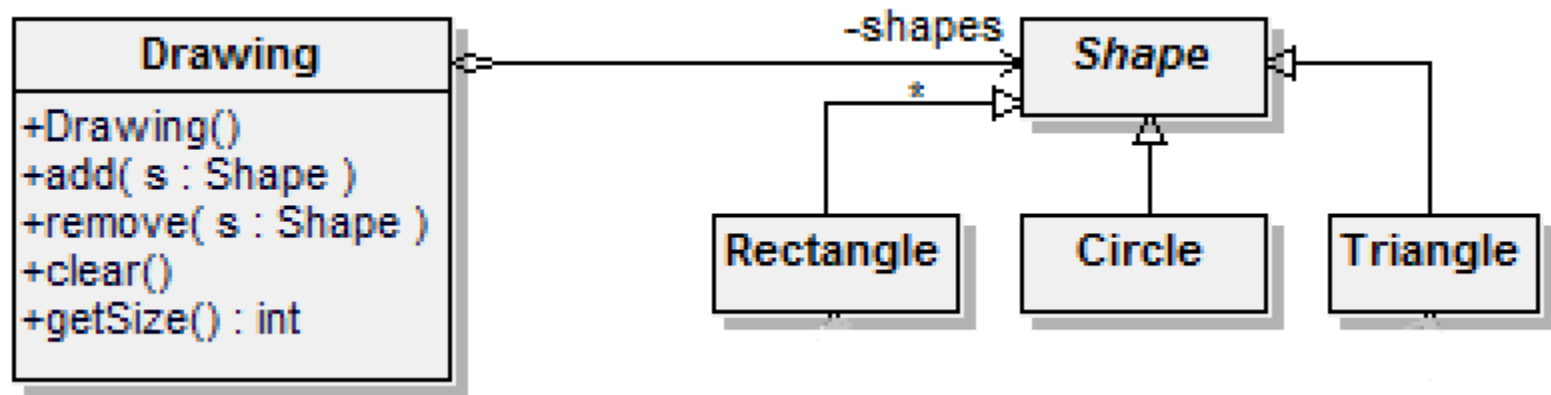
Een autoradio **is een** radio.

Een auto **heeft een** autoradio.

Vaak is er een combinatie van overerving en associatie om code-hergebruik te maximaliseren.

# 11. Code hergebruik: overerving versus associaties

Voorbeeld figuren:



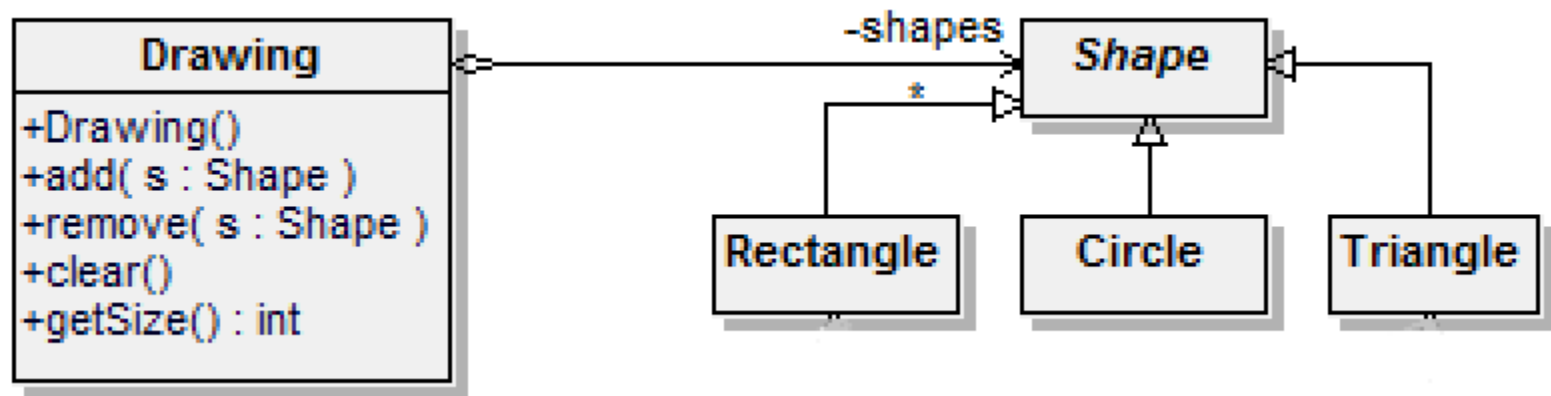
Klasse Drawing: een tekening met een aantal figuren.

Zo'n tekening is zelf geen figuur, maar bevat wel figuren:

- een object van de klasse Drawing heeft 1 of meerdere objecten van de klasse Shape
- de relatie tussen Drawing en Shape is "HEEFT"
- klasse Drawing heeft een private variabele die een referentie is naar een array van figuren.

# 11. Code hergebruik: overerving versus associaties

Voorbeeld figuren:



In de klasse Drawing worden de figuurobjecten bijgehouden in een array:

```
public class Drawing {
    private Shape[] Shapes;
    ...
}
```

## Opdracht 12

1. Maak in de package “be.pxl.opdracht8\_shapes” een klasse Drawing met 1 eigenschap: een array van Shape-objecten.
2. Voorzie een constructor met 1 parameter, nl het aantal elementen dat de array moet kunnen bevatten.
3. Voorzie een methode add(Shape s). Test bij het toevoegen of de figuur al in de tekening zit (gebruik hiervoor equals()). Plaats een nieuwe figuur op de eerstvolgende lege plaats in de array. Indien de array vol is, doe je niets.  
Hint: voeg een private methode isPresent() toe om na te gaan of een figuur reeds aanwezig is.  
Hint: voeg een private methode getFreeLocation() toe om de eerstvolgende vrije plaats in de array op te zoeken.
4. Voorzie een methode remove(Shape s) om een figuur te verwijderen.
5. Maak een methode clear() die de volledige tekening wist.
6. Maak een methode print() die van alle figuren de gegevens afdukt.
7. Maak een nieuw hoofdprogramma DrawingApp waarin je alles test.



# 12. Samenvatting

Concept overerving → voor hergebruik van code

**Een relatie tussen subklasse en superklasse is 3-voudig:**

1. Subklassen erven alles over van de superklasse
2. Subklassen kunnen nieuwe elementen toevoegen
3. Subklassen kunnen elementen van de superklasse vervangen

→ Zoveel mogelijk gemeenschappelijke code opnemen in superklassen!!!

**Abstracte klasse:**

→ Er kunnen geen objecten van gemaakt worden

