

HO GENT

OOSDII

Polymorfisme & Interfaces

Table of Contents

1. Doelstellingen	1
2. Inleiding	1
3. Polymorfisme	1
3.1. Regels voor toekenning	3
4. Gekende methodes van een referentie	3
4.1. Static & dynamic binding	4
4.2. Voorbeeld van dynamic binding in Java	5
5. abstract klasse bij polymorfisme	6
6. Dependency Injection	6
7. Interfaces	9
8. Objecten samenbrengen in een context	13
8.1. KunstCollectie en KunstWinkel	16
8.2. Drawable en Saleable	17
8.3. Huisdier en Voertuig	17
8.4. Kat, Hond, Auto en Boot	18
9. Implementatie van meerdere interfaces	19
10. Objecten sorteren	20
10.1. Comparable interface	20
10.2. The Comparator interface	22

1. Doelstellingen

Na het studeren en maken van de oefeningen van dit hoofdstuk ben je in staat om volgende zaken te herkennen, toe te lichten, te definiëren, toe te passen en te implementeren:

- Overerving en polymorfisme
- Static en dynamic binding
- Dependency injection
- Interfaces

2. Inleiding

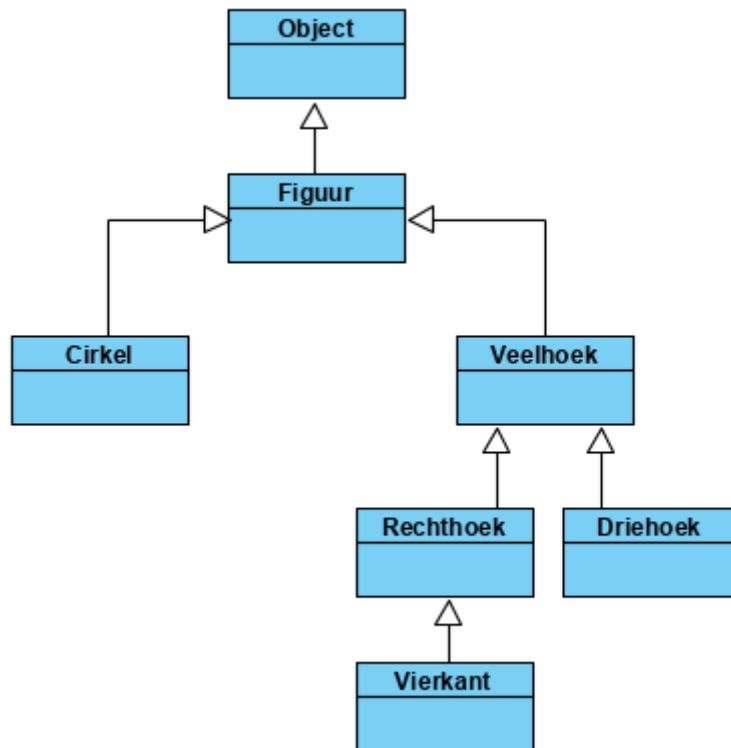
Het woord polymorfisme (letterlijk: veelvormigheid) betekent dat iets meerdere vormen of gedaanten kan hebben. Bv. water kan ijsvormig zijn, vloeibaar of gasvormig.

Binnen Java betekent de term dat een referentie variabele van een bepaald type kan verwijzen naar objecten van verschillende types: het type zelf of één van zijn subtypes.

Dankzij polymorfisme kan je achteraf ook nog subklassen toevoegen die de eigenschappen en het gedrag van de superklasse overnemen en mee in een polymorfische verzameling kunnen opgenomen worden. De enige code die daarvoor moet geschreven/aangepast worden is de code van deze nieuwe subklasse. Analooch zal een wijziging in de implementatie van een (bestaande) subklasse ook geen gevolgen hebben voor de andere subklassen of voor de superklasse. Enkel klassen die gebruik maken van de nieuwe/gewijzigde subklasse zullen mogelijk herbekeken moeten worden.

3. Polymorfisme

Polymorfisme of veelvormigheid komt voort uit de '**IS EEN**' relatie: overerving. Het laat toe om objecten in dezelfde klassenhiërarchie te aanzien als objecten van de superklasse.



Polymorfisme betekent dat een referentie variabele van een bepaald type kan verwijzen naar verschillende objecten van dit type, maar ook naar objecten van subtypes van dit type.

Een referentie variabele van het type Object kan refereren naar een Figuur, gezien de relatie 'Figuur IS EEN Object'. Dezelfde referentie kan verwijzen naar een Vierhoek, gezien de relatie 'Vierhoek IS EEN Object'.

```

1 public class VierkantApp {
2     public static void main( String args[]) {
3         // Dit statement instantieert een Vierkant en kent
4         // de referentie toe aan ref_1
5         Vierkant ref_1 = new Vierkant("Vierkant 1");
6
7         // Aangezien Rechthoek de superklasse is
8         // van Vierkant kan de superklasse referentie
9         // ref_2 verwijzen naar een instantie van Vierkant
10        Rechthoek ref_2 = new Vierkant("Vierkant 2"); ①
11
12        // Een referentie van een superklasse kan niet direct
13        // toegekend worden aan een referentie van een subklasse.
14        // Expliciet casten is nodig!
15        if (ref_2 instanceof Vierkant) {
16            Vierkant ref_3 = (Vierkant) ref_2; ②
17        }
18    }
19 }
  
```

① Een superklasse referentie kan refereren naar een instantie van een subklasse. Dit noemt men

upcasting: de referentie naar een object van een subtype (lager in de hiërarchie) wordt toegekend aan een referentie van een supertype (hoger in de hiërarchie).

- ② Een superklasse referentie kan enkel toegekend worden aan een subklasse referentie na een *downcast*. De waarde van een referentie van een supertype (hoger in de hiërarchie) wordt toegekend aan een referentie van een subtype (lager in de hiërarchie).



Let op met een downcast: ga vooraf na of de referentie wel refereert naar een object van het juiste type vooraleer de downcast uit te voeren. Gebruik het keyword `instanceof` om na te gaan of een object van een specifiek type is.

3.1. Regels voor toekenning

Polymorfisme bepaalt duidelijk wat de regels voor toekenning van referenties zijn:

- Als het type van de variabele een klasse is, kan de rechteroperand ofwel `null` zijn ofwel een referentie met als type:
 - dezelfde klasse
 - een subklasse van die klasse
- Als het type van de variabele een interface is, kan de rechter operand ofwel `null` zijn ofwel een referentie met als type::
 - dezelfde interface
 - een sub-interface van die interface
 - een klasse die de interface implementeert
 - een klasse die een subinterface van die interface implementeert

4. Gekende methodes van een referentie

Een referentie heeft altijd één specifiek type. Dit type bepaalt welke instantiemethodes kunnen aangeroepen worden via die referentie.



Instantiemethodes kunnen enkel aangeroepen worden via een referentievariabele als en slechts als de methodenaam gekend is binnen het type van die referentievariabele.

- Als het type van de variabele een klasse is dan zijn de gekende instantiemethodes:
 - de zichtbare methodes gedeclareerd binnen die klasse
 - Let op: binnen de klasse zelf zijn alle methodes binnen die klasse gedeclareerd zichtbaar (dus ook de `private` methodes)
 - de zichtbare methodes gedeclareerd binnen een superklasse van die klasse
 - de zichtbare methodes gedeclareerd in een interface die die klasse implementeert
- Als het type van de variabele een interface is dan zijn de gekende instantiemethodes:

- de methodes gedeclareerd binnen die interface
- de zichtbare methodes gedeclareerd in een superinterface van die interface



Is het type van een variabele een klasse, en verwijst deze naar een object van een subklasse van die klasse, dan zijn de instantiemethodes waarmee de subklasse werd uitgebreid niet gekend voor die variabele!

4.1. Static & dynamic binding

Het linken van een methode declaratie aan zijn implementatie noemt men **binding**. Er zijn twee types binding:

- **Static Binding** → gebeurt tijdens het compileren (@ compile time)
- **Dynamic Binding** → gebeurt tijdens de uitvoering (@ runtime)

Bij static binding kan de link tussen een methode declaratie en zijn implementatie gelegd worden tijdens het compileren. Dit is enkel mogelijk indien het type van het object, waarop de methode wordt uitgevoerd, gekend is tijdens het compileren.

Bij het gebruik van private, final of static methodes en attributen kan static binding plaatsvinden: deze zijn verbonden met het type zelf.



static, private en final methodes kunnen niet overschreven worden. Het bepalen tot welke klasse de methode behoort kan plaatsvinden tijdens het compileren. Het codeblok dat hoort bij de methode kan bij het uitvoeren rechtstreeks aangeroepen worden.

Wegens polymorfisme, waarbij een superklasse referentie kan verwijzen naar een object van een subklasse, is de link tussen de implementatie en de declaratie van een methode tijdens het compileren niet altijd te leggen (@Override). Er moet gewacht worden tot het type van het object waarnaar de referentie verwijst, gekend is. Dit kan enkel tijdens de uitvoering van het programma en noemen we 'dynamic binding'.

Het overschrijven van een methode in een subklasse is een voorbeeld van 'dynamic binding'. In dit geval hebben zowel de super- als subklasse dezelfde declaratie van een methode en moet gewacht worden tot het precieze type van het object gekend is vooraleer de implementatie van de methode uit te voeren.



Bij 'dynamic binding' bepaalt het type van het object (niet het type van de referentie) welke implementatie zal uitgevoerd worden. Het type van het object wordt @runtime dynamisch bepaald, vandaar de term 'dynamic binding'.

```

1 class App {
2     public static void tekenFiguur(Figuur obj) {
3         obj.teken();
4     }
5
6     public static void main( String args[]) {
7         tekenFiguur(new Cirkel()); ①
8         tekenFiguur(new Rechthoek()); ②
9     }
10 }

```

- ① De referentie van type Figuur (= parameter 'obj' in de methode tekenFiguur) zal verwijzen naar een object van type **Cirkel**. Bij het aanroepen van de methode **teken** zal de implementatie van het type Cirkel worden uitgevoerd.
- ② De referentie van type Figuur in de tekenFiguur methode zal verwijzen naar een object van type **Rechthoek**. Bij het aanroepen van de methode **teken** zal de implementatie van het type Rechthoek worden uitgevoerd.

Bij het aanroepen van de methode **teken** kan de compiler niet bepalen welk gedrag precies zal uitgevoerd worden: zal de methode een Cirkel tekenen of een Rechthoek of een ...? Het uitgevoerde gedrag hangt af van het type van het argument dat wordt meegegeven aan de methode tekenFiguur, en kan pas @runtime bepaald worden.

4.2. Voorbeeld van dynamic binding in Java

```

1 public class DynamicBindingTest {
2     public static void main(String args[]) {
3         Voertuig voertuig;
4
5         voertuig = new Voertuig();
6         voertuig.start();
7
8         voertuig = new Auto();
9         voertuig.start();
10    }
11 }
12
13 class Voertuig {
14     public void start() {
15         System.out.println("Inside start method of Vehicle");
16     }
17 }
18
19 class Auto extends Voertuig {
20     @Override
21     public void start() {
22         System.out.println("Inside start method of Car");
23     }
24 }

```

5. abstract klasse bij polymorfisme

Een **abstracte klasse** is een klasse die gedeclareerd is met het keyword **abstract**.



- Een abstracte klasse kan niet geïntantieerd worden, maar kan wel subklassen hebben.
- Een subklasse van een abstracte klasse kan eveneens een abstracte klasse zijn.
- Een abstracte klasse mag abstracte methodes bevatten, maar dit is niet verplicht.



Een abstracte klasse wordt gebruikt om een hoger niveau van **generalisatie** te bereiken.

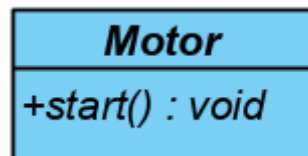
Een **abstracte methode** is een methode declaratie zonder implementatie (zonder {}), gevolgd door een puntkomma.

6. Dependency Injection



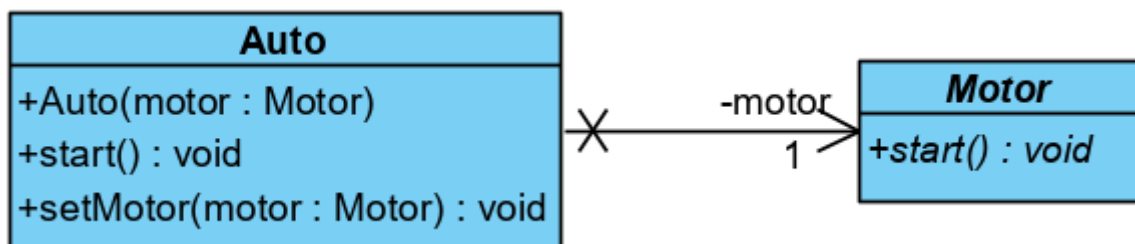
Een abstracte methode dwingt gedrag af dat kan verwacht worden voor het type, maar dat nog niet kan geïmplementeerd worden op dat niveau in de hiërarchie. Een hogere laag in software kan gebruik maken van abstracte methodes, waarvan de eigenlijke implementatie op een later moment geïnjecteerd wordt: dit noemen we 'Dependency Injection'.

Stel dat je een Auto wenst te bouwen, maar de motor op zich is nog niet geïmplementeerd. Wat je kan doen is een 'standaard' afspreken waaraan een motor moet voldoen zodat je met de ontwikkeling van de Auto kan starten: je legt een abstracte klasse Motor vast met daarin een aantal afspraken omtrent zijn gedrag, een 'standaard' of 'contract' waaraan een motor moet voldoen.



```
1 public abstract class Motor {
2     public abstract void start();
3 }
```

Op basis van deze abstracte klasse kan de implementatie van een Auto starten, zonder dat een concrete motor reeds geïmplementeerd werd.

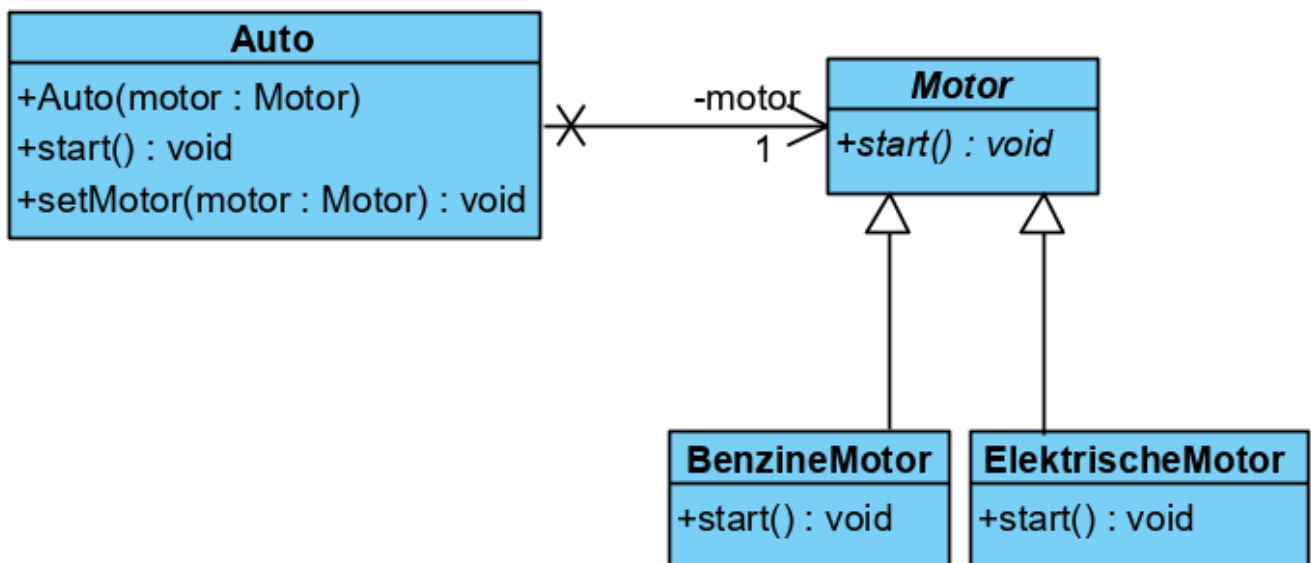


```
1 public class Auto {
2     private Motor motor;
3
4     public Auto(Motor motor) {
5         setMotor(motor);
6     }
7
8     public void start() {
9         motor.start();
10    }
11
12    public final void setMotor(Motor motor) {
13        this.motor = motor;
14    }
15 }
```



Merk op dat de eigenlijke motor die een Auto zal gebruiken bij creatie van een Auto als argument wordt meegegeven aan de constructor: de werking van een Auto hangt af van een Motor (relatie: Auto HEEFT EEN Motor). Het meegeven van een concrete motor bij de creatie van een Auto noemen we '**Dependency Injection**'. Over welk type Motor het zal gaan is nog niet zeker bij de implementatie van de klasse Auto, wel is zeker dat deze concrete Motor een subtype zal zijn van de klasse Motor. Via polymorfisme wordt het gedrag van dit subtype uitgevoerd.

Op een later tijdstip kunnen verschillende types van Motor geïmplementeerd worden: een ElektrischeMotor en een BenzineMotor. Eén van beide kan geïnjecteerd worden in de Auto bij creatie.



```
1 public class ElektrischeMotor extends Motor {
2
3     @Override
4     public void start() {
5         System.out.println("Gedrag van een elektrische motor");
6     }
7
8 }
```

```
1 public class BenzineMotor extends Motor {
2
3     @Override
4     public void start() {
5         System.out.println("Gedrag van een benzine motor");
6     }
7 }
```



Merk op: een Auto hoeft niet te weten met welke specifieke motor hij werkt, enkel dat het een Motor is.

Op een nog hoger niveau kan diegene die de Auto aanmaakt beslissen over welke Motor zijn Auto zal beschikken. Een ElektrischeMotor of een BenzineMotor. Het staat hem vrij om eventueel nog een derde Motor te ontwikkelen: een DieselMotor.

```
1 public class AutoApp {
2     public static void main(String[] args) {
3         Auto mijnElektrischeAuto = new Auto(new ElektrischeMotor());
4         mijnElektrischeAuto.start();
5
6         Auto mijnBenzineAuto = new Auto(new BenzineMotor());
7         mijnBenzineAuto.start();
8
9         // Verbouwing aan mijn benzine auto
10        mijnBenzineAuto.setMotor(new ElektrischeMotor());
11        mijnBenzineAuto.start();
12    }
13 }
```



Via de methode `setMotor` kan de gebruiker van een Auto zelfs het type Motor van de Auto wijzigen, @runtime! Ook dit is een vorm van 'Dependency Injection'.

7. Interfaces

There are a number of situations in software engineering when it is important for disparate groups of programmers to agree to a "**contract**" that spells out how their software interacts. Each group should be able to write their code without any knowledge of how the other group's code is written. Generally speaking, **interfaces are such contracts**.

— Oracle JAVA tutorial

Een interface is een referentietype, vergelijkbaar met een klasse, dat enkel het volgende kan bevatten:

- constanten
- abstracte methodes (enkel de signatuur, zonder implementatie)
- `default` methodes met implementatie
- `static` methodes met implementatie



Implementaties van methodes binnen een interface bestaan enkel voor `default` methodes en `static` methodes.

Een default methode binnen een interface is impliciet **public** en moet een concrete implementatie bevatten; als een klasse die de interface implementeert, geen override voorziet voor deze methode, dan gebruikt ze de default-definitie.

Een default methode laat toe om nieuwe methodes aan een interface toe te voegen, zodat deze automatisch ook beschikbaar zijn binnen een implementatie van de interface. Binnen het hoofdstuk over Lambda expressies wordt het gebruik van enkele default methodes toegelicht.

Daarnaast kan een methode ook **static** gedeclareerd worden. Een static methode declareren binnen een interface is identiek als bij een klasse. Net zoals bij klassemethodes horen deze static methodes niet bij een object. Je kan ze aanroepen in combinatie met de naam van de interface.



Een interface heeft GEEN constructor! Deze kan dus nooit geïnstantieerd worden.

Het interface type heeft geen instantie variabelen, en declareert typisch één of meerdere abstracte methodes.



Een '**Functionele interface**' is een interface die slechts één abstracte methode declareert.



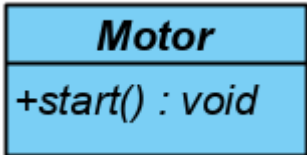
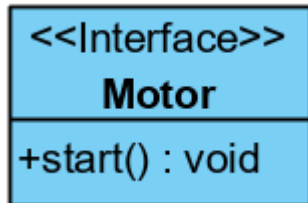
Een interface methode zonder **private**, **default**, of **static** modifier is impliciet **public** en **abstract**.



Elk attribuut gedeclareerd binnen de body van een interface is impliciet **public**, **static** en **final**.

```
1 public interface Constants
2 {
3     public static final int INSERT = 1;
4     int MODIFY = 2; // implicitly public, static and final
5     int DELETE = 3;
6 }
```

In het voorbeeld van onze Auto kan de abstracte klasse Motor vervangen worden door een interface Motor.

Abstracte klasse	Interface
	

```

1 public abstract class Motor {
2     public abstract void start();
3 }

```

```

1 public interface Motor {
2     void start();
3 }

```

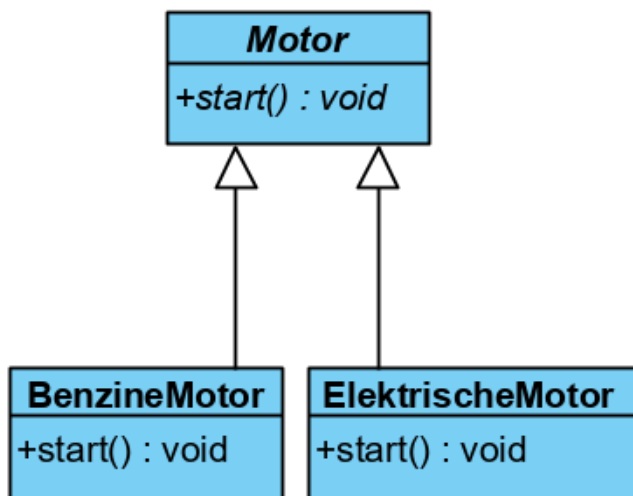


Een klasse kan een interface **implementeren** door de abstracte methoden uit de interface concreet te maken.



Let op de nieuwe relatie in het UML om aan te duiden dat een klasse een interface implementeert.

superklasse

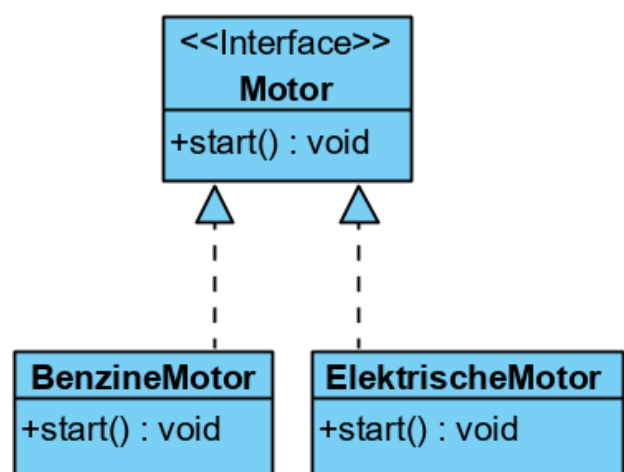


```

1 public class ElektrischeMotor extends
  Motor {
2
3     @Override
4     public void start() {
5         System.out.println("Gedrag van
  een elektrische motor");
6     }
7
8 }

```

interface



```

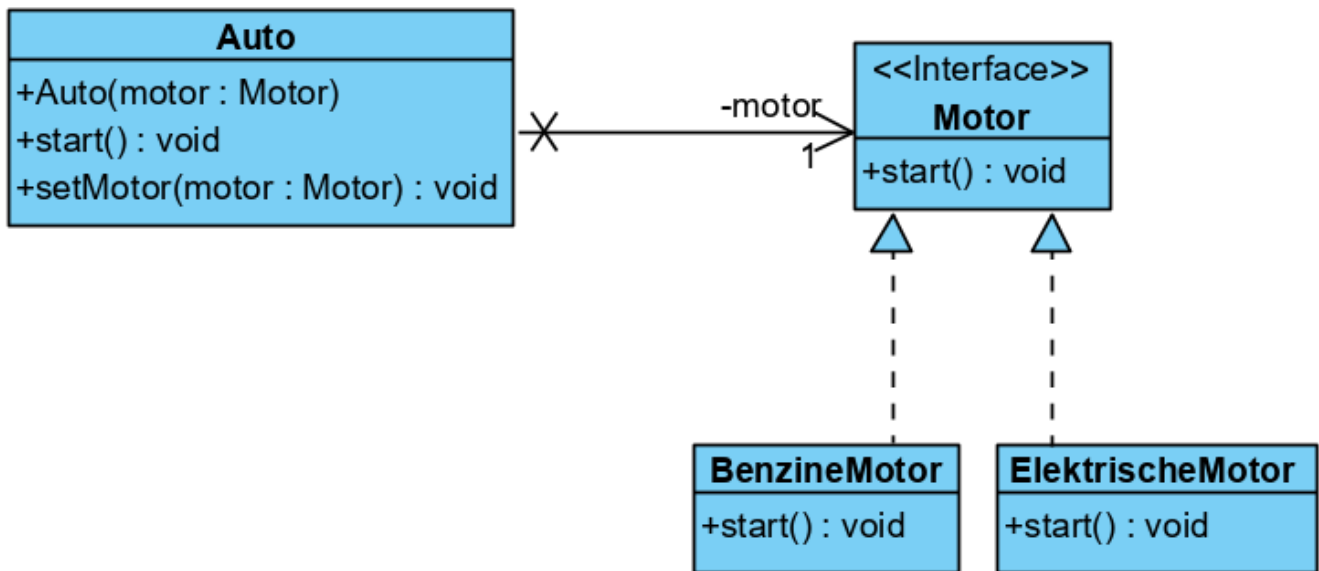
1 public class ElektrischeMotor
  implements Motor {
2
3     @Override
4     public void start() {
5         System.out.println("Gedrag van
  een elektrische motor");
6     }
7
8 }

```



Een klasse duidt met het keyword **implements** aan dat het een interface implementeert.

De klasse Auto duidt op dezelfde manier aan als voorheen dat deze gebruik maakt van een Motor (Auto HEEFT EEN Motor): via een associatie.



```

1 public class Auto {
2     private Motor motor;
3
4     public Auto(Motor motor) {
5         setMotor(motor);
6     }
7
8     public final void start() {
9         motor.start();
10    }
11
12    public final void setMotor(Motor motor) {
13        this.motor = motor;
14    }
15 }
  
```



Merk op: een Auto hoeft nog steeds niet te weten met welke specifieke motor hij werkt, enkel dat het een Motor is. De interface Motor is hier op dezelfde manier bruikbaar als de abstracte klasse Motor in het vorige voorbeeld. Er is nog altijd sprake van '**Dependency Injection**' en polymorfisme.

Merk ook op dat de hoger liggende applicatieklasse niet wijzigt bij de overgang naar een interface:

```

1 public class AutoAppMetInterface {
2     public static void main(String[] args) {
3         Auto mijnElektrischeAuto = new Auto(new ElektrischeMotor());
4         mijnElektrischeAuto.start();
5
6         Auto mijnBenzineAuto = new Auto(new BenzineMotor());
7         mijnBenzineAuto.start();
8
9         // Verbouwing aan mijn benzine auto
10        mijnBenzineAuto.setMotor(new ElektrischeMotor());
11        mijnBenzineAuto.start();
12    }
13 }

```

Program to an interface, not an implementation!

— Design Principle

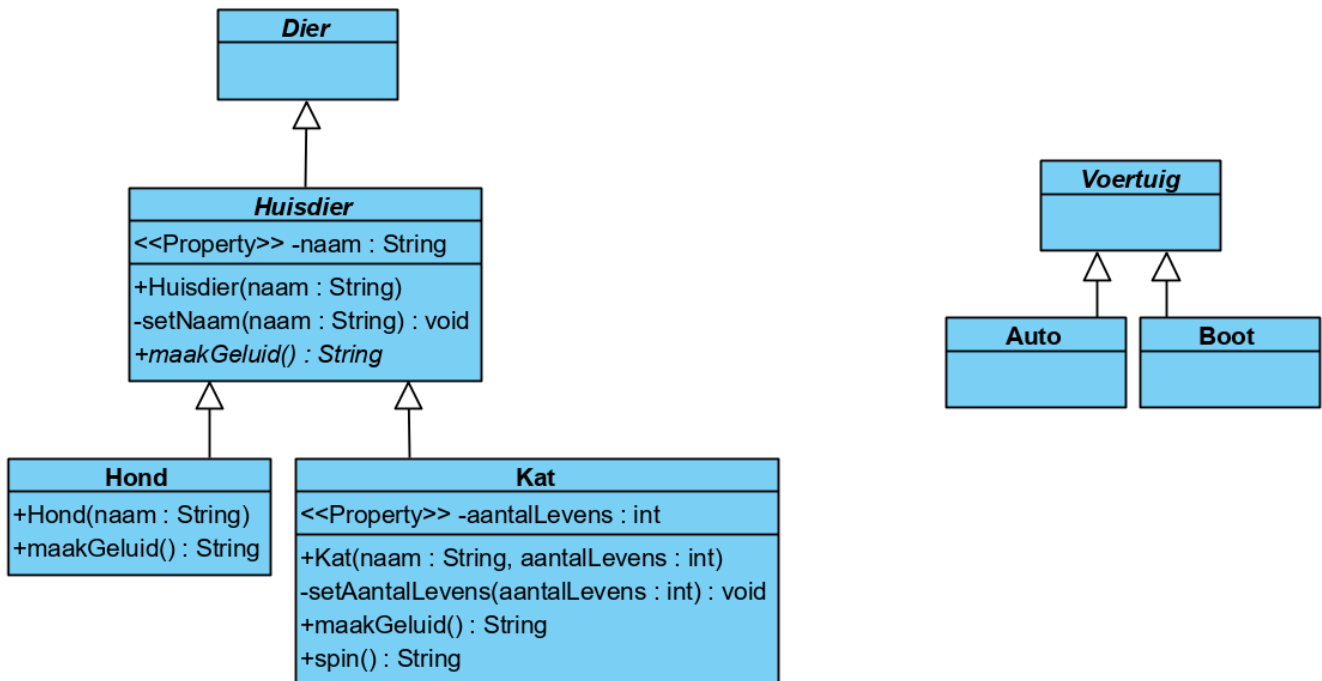
Voorgaand principe duidt aan dat je vaak meer flexibiliteit krijgt door te programmeren naar een interface (Motor) dan naar een implementatie (ElektrischeMotor of BenzineMotor). In combinatie met een associatie (de 'HEEFT EEN' relatie) en 'Dependency Injection' geeft dit de mogelijkheid om later eenvoudig van implementatie te wisselen. (Dit zal later leiden tot het 'Strategy Design Pattern' - keuzepakket 'Development'.)

8. Objecten samenbrengen in een context

Een andere toepassing van interfaces is het onderbrengen van objecten van niet gerelateerde klassen in een bepaalde context.

Instanties van een klasse kunnen door middel van een interface in een bepaalde context ondergebracht worden. Zo kunnen ook instanties van klassen die niets met elkaar te maken hebben in een bepaalde context samen komen door de implementatie van een gemeenschappelijke interface (= de implementatie van de abstracte methodes van die interface).

De klasse Huisdier en Voertuig hebben ogenschijnlijk niets met elkaar te maken.

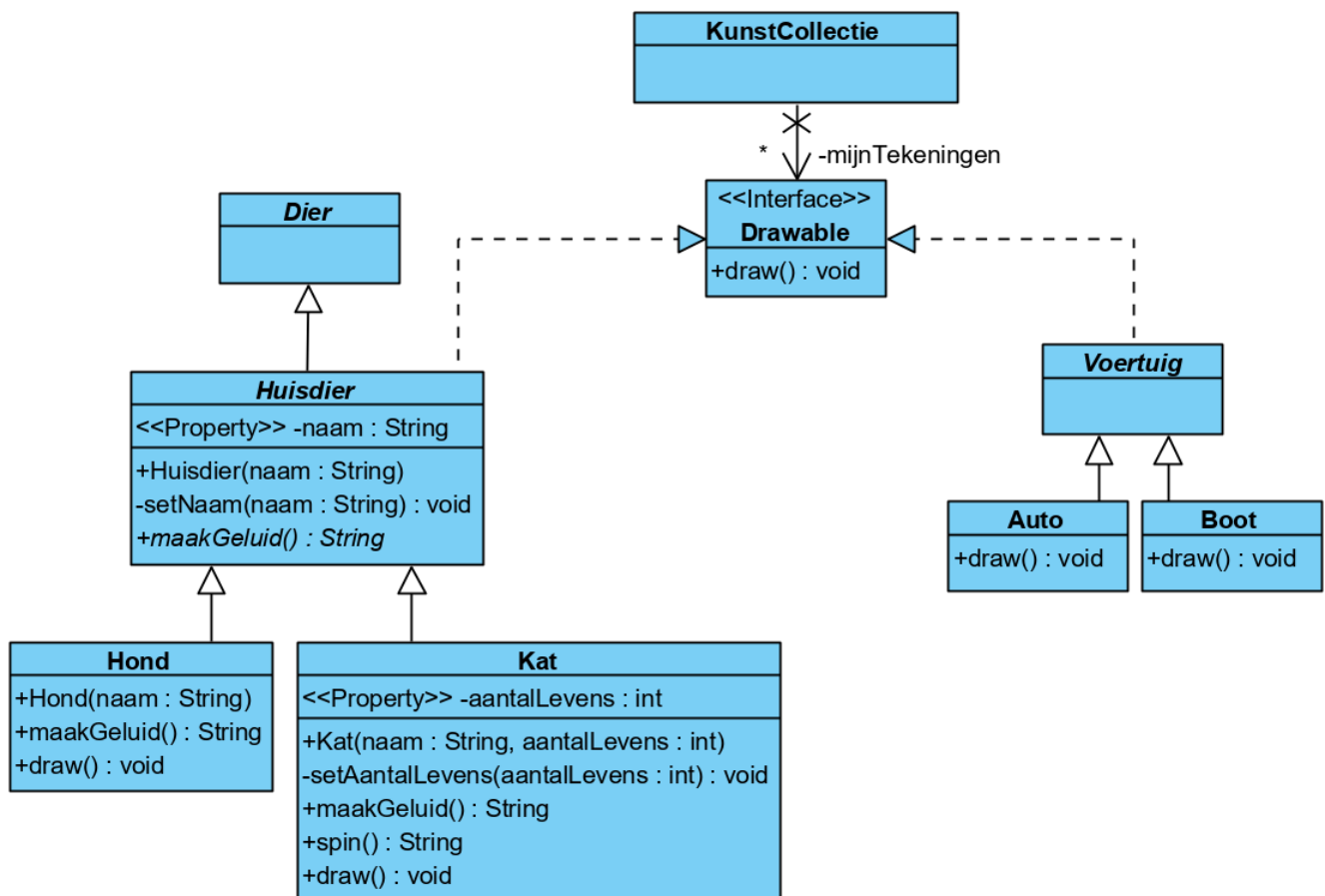


Door de klasse Huisdier en Voertuig een gemeenschappelijke interface te laten implementeren kunnen ze wel in een bepaalde context samenkomen.

Stel dat een kunstverzamelaar een verzameling wenst vast te leggen van "tekenbare" (in het Engels "Drawable") objecten. Graag had hij tekeningen van zijn Huisdieren en zijn Voertuigen samengebracht in zijn collectie.

In bovenstaand voorbeeld is dit niet mogelijk via overerving, aangezien de klasse Huisdier al een superklasse Dier heeft.

Door zowel Huisdier als Voertuig de interface "Drawable" te laten implementeren kunnen objecten van beide klassen toch ondergebracht worden in een gemeenschappelijke context: de kunstcollectie.

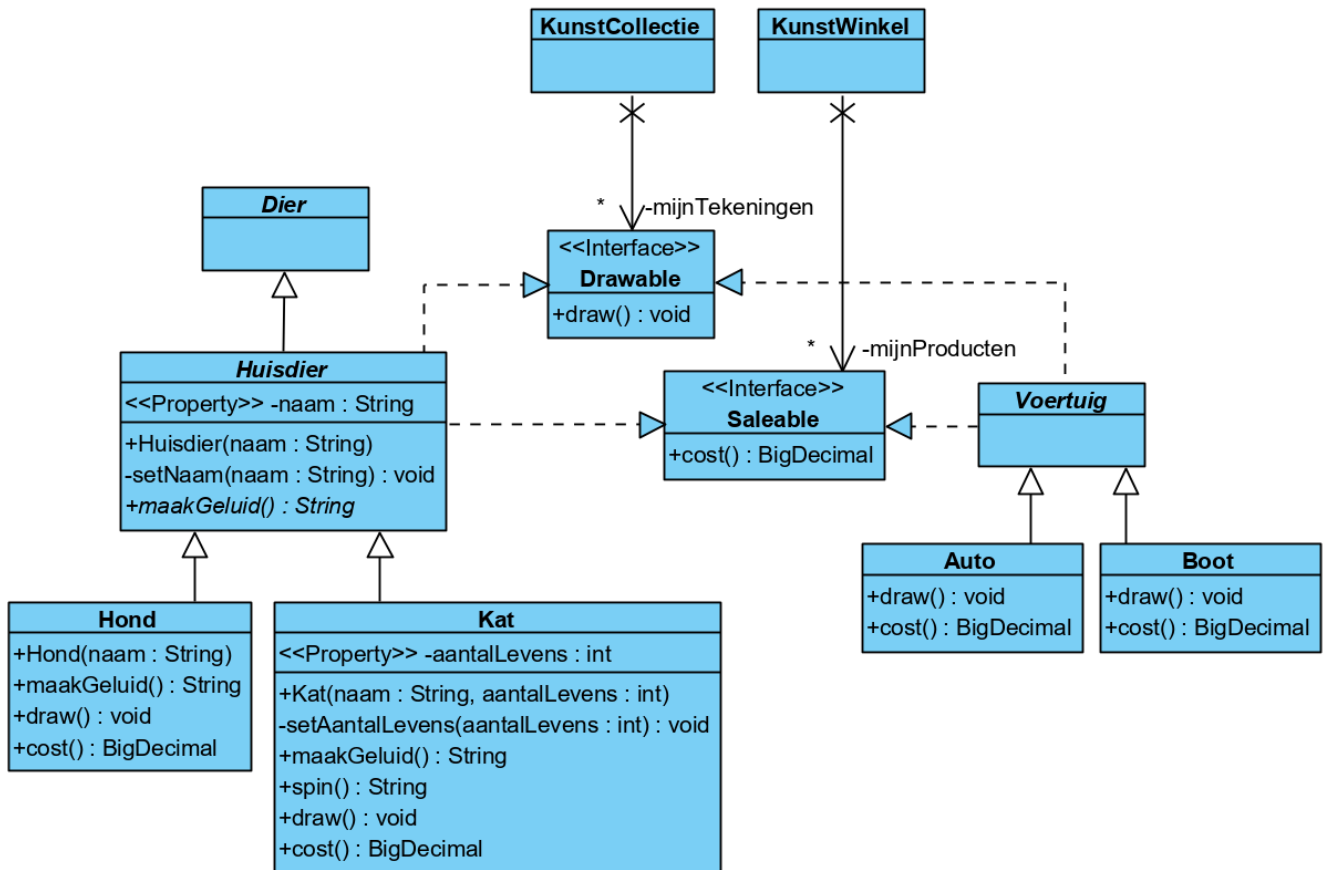


Ook via een interface kan je polymorfisme toepassen: de kunstcollectie 'HEEFT EEN' lijst van *drawable* objecten. Indien op zo een object de methode *draw* wordt aangeroepen zal het type object bepalen welke implementatie wordt uitgevoerd.

Wil de kunstverzamelaar later zijn kunstcollectie verkopen in een winkel, dan kan hij de objecten van zijn kunstcollectie samenbrengen in de context van deze winkel. Als een object verkocht zal worden in een winkel, moet het verkoopbaar (in het Engels "saleable") zijn: de verkoopprijs (*cost*) moet opgevraagd kunnen worden.



Vele namen van (bestaande) interfaces in het Engels eindigen op *-able*. Voorbeelden zijn *Serializable*, *Comparable*, *AutoCloseable*, *Runnable*... Je zelfgekozen Nederlandse naam van een dergelijke interface zou naar analogie vaak op *-baar* kunnen eindigen (*Verkoopbaar*, *Tekenbaar*, ...)



8.1. KunstCollectie en KunstWinkel

De klasse KunstCollectie heeft een associatie met de interfaces Drawable, de KunstWinkel met de interface Saleable.

Via polymorfisme kunnen objecten getekend worden binnen de kunstcollectie en kan de verkoopprijs opgevraagd worden bij verkoop. Polymorfisme zorgt ervoor dat objecten van nieuwe subklassen van Huisdier (bv. Konijn) en/of Voertuigen (bv. Vrachtwagen) zonder wijzigingen in de KunstCollectie en KunstWinkel kunnen opgenomen worden.

```
1 public class KunstCollectie {
2     private List<Drawable> mijnTekeningen;
3 }
```

```
1 public class KunstWinkel {
2     private List<Saleable> mijnProducten;
3 }
```

Code should be open for extension, but closed for modification!

— Open-closed design principle

De bedoeling is dus eigenlijk dat we onze code gemakkelijk kunnen uitbreiden met nieuwe klassen en methodes, maar geen wijzigingen in bestaande methodes moeten aanbrengen. Om dit perfect te

kunnen doen, zouden we echter de toekomst moeten kunnen voorspellen zodat we weten welke uitbreidingen er later eventueel nog mogelijk zijn. Het is dus moeilijk haalbaar om perfect aan dit principe te voldoen voor het hele systeem, maar we kunnen wel proberen om bepaalde klassen of methodes uitbreidbaar maar niet wijzigbaar te maken.

8.2. Drawable en Saleable

Drawable en Saleable zijn interfaces die de klassen die ze implementeren, samenbrengen in een kunstcollectie en/of kunstwinkel. Objecten van klassen die deze interfaces implementeren zijn tekenbaar en hebben een bepaalde kost.

```
1 public interface Drawable {  
2     public void draw();  
3 }
```

```
1 public interface Saleable {  
2     public BigDecimal cost();  
3 }
```

8.3. Huisdier en Voertuig

De klassen Huisdier en Voertuig hebben op het eerste zicht niets met elkaar te maken. Via hun gemeenschappelijke interface(s) komen ze onder een bepaalde context samen: in dit geval de KunstCollectie.



Huisdier en Voertuig implementeren elk twee interfaces. Huisdier heeft één superklasse: Dier. Voertuig heeft één superklasse: Object.



De methoden uit de interfaces worden in deze abstracte klassen niet geïmplementeerd.

```

1 public abstract class Huisdier extends Dier implements Drawable, Saleable {
2     private String naam;
3
4     public Huisdier(String naam) {
5         setNaam(naam);
6     }
7
8     public String getNaam() {
9         return naam;
10    }
11
12    private void setNaam(String naam) {
13        this.naam = naam;
14    }
15
16    public abstract String maakGeluid();
17 }

```

```

1 public abstract class Voertuig implements Drawable, Saleable {
2
3 }

```

8.4. Kat, Hond, Auto en Boot

De klassen Kat, Hond, Auto en Boot implementeren op hun beurt de abstracte methodes vastgelegd in de interfaces die ze volgens hun superklasse dienen te implementeren. Objecten van deze klassen kunnen opgenomen worden in de KunstCollectie en verkocht worden.

```

1 public class Hond extends Huisdier{
2
3     public Hond(String naam) {
4         super(naam);
5     }
6
7     @Override
8     public String maakGeluid(){
9         return "Woef, woef...";
10    }
11
12    @Override
13    public void draw() {
14        System.out.printf("Tekening van Hond.%n");
15    }
16
17    @Override
18    public BigDecimal cost() {
19        System.out.printf("Een Hond is onbetaalbaar.%n");
20        return BigDecimal.valueOf(1000000);
21    }
22 }

```

```

1 public class Auto extends Voertuig {
2     @Override
3     public void draw() {
4         System.out.printf("Tekening van Auto.%n");
5     }
6
7     @Override
8     public BigDecimal cost() {
9         System.out.printf("Een auto is maar een voorwerp.%n");
10        return BigDecimal.valueOf(100);
11    }
12 }

```

9. Implementatie van meerdere interfaces



Een Java klasse kan slechts één super klasse hebben. Meervoudige overerving is niet toegelaten. Een Interface is echter geen klasse. Een klasse kan meerdere interfaces implementeren en een interface kan meerdere interfaces uitbreiden.

```

1 public interface Hockey extends Sports, Event

```

```
1 public class Sleutel extends Voorwerp implements DraagBaar, Kosten
```

10. Objecten sorteren

10.1. Comparable interface

Deze veelgebruikte interface legt een 'orde' vast tussen objecten van klassen die de interface implementeren.



De Comparable interface laat je toe om objecten te sorteren! Deze manier van sorteren op basis van deze Comparable interface noemt men ook '**natuurlijke sortering**'. Natuurlijke sortering van getallen is van klein naar groot. Natuurlijke sortering van letters en woorden is alfabetisch.

De Comparable interface legt volgende methode vast, dit is een functionele interface (met één abstracte methode):

```
1 public interface Comparable<T> {  
2     public int compareTo(T o);  
3 }
```



De methode `compareTo` vergelijkt het argument van de methode met het object waarop de methode wordt aangeroepen. Het resultaat is een negatief, 0, of een positief geheel getal (int) afhankelijk of het argument kleiner dan, gelijk aan (~equals), of groter dan het eigenlijke object zelf is. Als het argument niet kan vergeleken worden met het eigenlijke object, dan gooit de methode den `ClassCastException`.

It is strongly recommended (though not required) that natural orderings be consistent with equals. This is so because sorted sets (and sorted maps) without explicit comparators behave "strangely" when they are used with elements (or keys) whose natural ordering is inconsistent with equals. In particular, such a sorted set (or sorted map) violates the general contract for set (or map), which is defined in terms of the equals method.

— Java API

Volgende klasse Naam implementeert de interface Comparable:

```

1 public class Name implements Comparable<Name> {
2     private final String firstName, lastName;
3
4     public Name(String firstName, String lastName) {
5         if (firstName == null || lastName == null)
6             throw new NullPointerException();
7         this.firstName = firstName;
8         this.lastName = lastName;
9     }
10
11     public String getFirstName() { return firstName; }
12     public String getLastName() { return lastName; }
13
14     @Override
15     public int compareTo(Name n) { ①
16         int lastCmp = lastName.compareTo(n.lastName);
17         return (lastCmp != 0 ? lastCmp : firstName.compareTo(n.firstName));
18     }
19
20     @Override
21     public int hashCode() { ②
22         return 31*firstName.hashCode() + lastName.hashCode();
23     }
24
25     @Override
26     public boolean equals(Object o) { ③
27         if (!(o instanceof Name))
28             return false;
29         Name n = (Name) o;
30         return n.firstName.equals(firstName) && n.lastName.equals(lastName);
31     }
32
33     @Override
34     public String toString() {
35         return firstName + " " + lastName;
36     }
37 }

```

- ① De compareTo methode vergelijkt eerst op achternaam. Als deze gelijk moesten zijn wordt op voornaam vergeleken. De 'natuurlijke sortering' van de klasse String wordt gebruikt om achter- en voornamen alfabetisch te sorteren.
- ② Ook de methode hashCode wordt gespecialiseerd. Dit is nodig aangezien ook de methode equals wordt gedefinieerd. (Gelijke objecten dienen een gelijke hash code te hebben!)
- ③ De methode equals geeft aan dat Name objecten gelijk zijn als ze dezelfde achternaam en voornaam hebben. Let op: in dit geval is het resultaat van compareTo gelijk aan '0'.

10.2. The Comparator interface



De Comparator interface laat toe om objecten te sorteren op een andere manier dan op natuurlijke wijze.

De natuurlijke sortering voor de klasse Name is: eerst sorteren op achternaam, dan op voornaam. Stel dat je toch wil sorteren op voornaam, en dan op achternaam (dus niet op natuurlijke wijze volgens de Comparable interface) kan je dit doen dmv een Comparator, waarbinnen de nieuwe sortering wordt vastgelegd.

```
1 public interface Comparator<T> {  
2     int compare(T o1, T o2);  
3 }
```



De methode `compare` vergelijkt twee argumenten met elkaar. Het resultaat is een negatief, 0, of een positief geheel getal (int) afhankelijk of het eerste argument kleiner dan, gelijk aan (~equal), of groter dan het tweede argument.

```
1 public class FirstNameComparator implements Comparator<Name> {  
2  
3     @Override  
4     public int compare(Name o1, Name o2) {  
5         int firstCmp = o1.getFirstName().compareTo(o2.getFirstName());  
6         return (firstCmp != 0 ? firstCmp : o1.getLastName().compareTo(o2.  
7             getLastName()));  
8     }
```

Beide interfaces (Comparable en Comparator) zie je aan het werk in volgende applicatie:


```

1 public class ComparableApp
2 {
3     public static void main(String[] args)
4     {
5         Name nameArray[] = {
6             new Name("John", "Smith"),
7             new Name("Karl", "Ng"),
8             new Name("Jeff", "Smith"),
9             new Name("Tom", "Rich")
10        };
11
12        List<Name> names = Arrays.asList(nameArray);
13
14        Collections.sort(names); ①
15        System.out.println("Natural order (last name, then first name):");
16        System.out.println(names);
17
18        Collections.sort(names, new FirstNameComparator()); ②
19        System.out.println("Ordered by first name, then last name:");
20        System.out.println(names);
21    }
22 }

```

- ① Een sorteeralgoritme gebruikt standaard de sortering op natuurlijke wijze om een lijst van elementen te sorteren. Deze wordt beschreven in de `compareTo`-methode van de klasse waartoe de elementen behoren, die dus de interface `Comparable` dient te implementeren.
- ② Een alternatieve sortering kan bekomen worden door in de `sort`-methode als tweede parameter een klasse mee te geven die een implementatie bevat van de `Comparator` interface en waarbij dan in de methode `compare` beschreven staat op welke manier de elementen van de lijst gesorteerd moeten worden.