HO
GENT

H7 Pijlers van OO

Table of Contents

1. Doelstellingen . . . 1
2. Inleiding . . . 1
3. De 4 pijlers van OO . . . 1
   3.1. Inkapseling . . . 1
   3.2. Abstractie . . . 2
   3.3. Overerving . . . 2
   3.4. Polymorfisme . . . 2
4. Meer over overerving . . . 3
   4.1. Terminologie . . . 3
   4.2. Een voorbeeld . . . 3
      4.2.1. Honden en katten . . . 3
      4.2.2. Generalisatie . . . 5
      4.2.3. Overerving toepassen in het voorbeeld . . . 8
      4.2.4. Toegangsrechten . . . 10
      4.2.5. Uitbreiding en specialisatie . . . 10
      4.2.6. Een kleine applicatie . . . 13
      4.2.7. Gebruik van het keyword super . . . 15
   4.3. Meer dan één niveau van overerving . . . 19
   4.4. De klasse Object . . . 20
5. Meer over polymorfisme . . . 21
   5.1. Wat is polymorfisme? . . . 21
   5.2. Gebruik van polymorfisme . . . 23
   5.3. Waarom polymorfisme? . . . 24
      5.3.1. De code wordt eenvoudiger door polymorfisme . . . 25
      5.3.2. Parameters en returntypes kunnen polymorf gedefinieerd worden . . . 26
      5.3.3. Een polymorfe hiërarchie is gemakkelijk uitbreidbaar . . . 27
      5.3.4. Toegepast op het voorbeeld . . . 27
   5.4. Andere voorbeelden van polymorfisme . . . 31
      5.4.1. Vierhoeken . . . 31
      5.4.2. Rekeningen . . . 31
      5.4.3. Werknemers . . . 32
6. Drielagenmodel . . . 32
   6.1. Waarom drie lagen? . . . 32
   6.2. Welke zijn de drie lagen? . . . 33
   6.3. Wat hoort waar en hoe communiceren de lagen met elkaar? . . . 35
      6.3.1. De presentatielaag . . . 35
      6.3.2. De domeinlaag . . . 36
      6.3.3. De persistentielaag . . . 36

1. Doelstellingen

Na het bestuderen van dit hoofdstuk ben je in staat

- De **vier pijlers van objectoriëntatie** te herkennen, te kunnen benoemen en verklaren
- **Overerving en polymorfisme** herkennen, kunnen definiëren, kunnen toepassen en implementeren
- **Het nut en de kracht** van overerving en polymorfisme kunnen toelichten
- De toegangsclausule **protected** kunnen gebruiken en implementeren
- De **constructor** in een subklasse kunnen implementeren
- De klasse **Object** en haar methodes kunnen gebruiken

2. Inleiding

In dit hoofdstuk gaan we echt volgens de principes van objectoriëntatie te werk. We breiden de domeinlaag uit met sub- en superklasses, die al dan niet abstract kunnen zijn.

Voor wie een mondje Grieks spreekt, is de term "polymorfisme" makkelijk te vertalen tot "veelvormigheid", maar wat we daar precies mee bedoelen mag voor een IT-student straks ook geen geheim meer zijn.

Wanneer we al deze nieuwe concepten op ontwerpniveau goed begrijpen, gaan we er ook mee aan de slag in onze code en leggen zo de basis voor een volledig objectgeoriënteerd programma.

3. De 4 pijlers van OO

Object georiënteerd ontwerpen/programmeren steunt op 4 grote pijlers:

	Nederlandstalige term	**Engelstalige term**
1	inkapseling	encapsulation
2	abstractie	abstraction
3	overerving	inheritance
4	polymorfisme	polymorfism

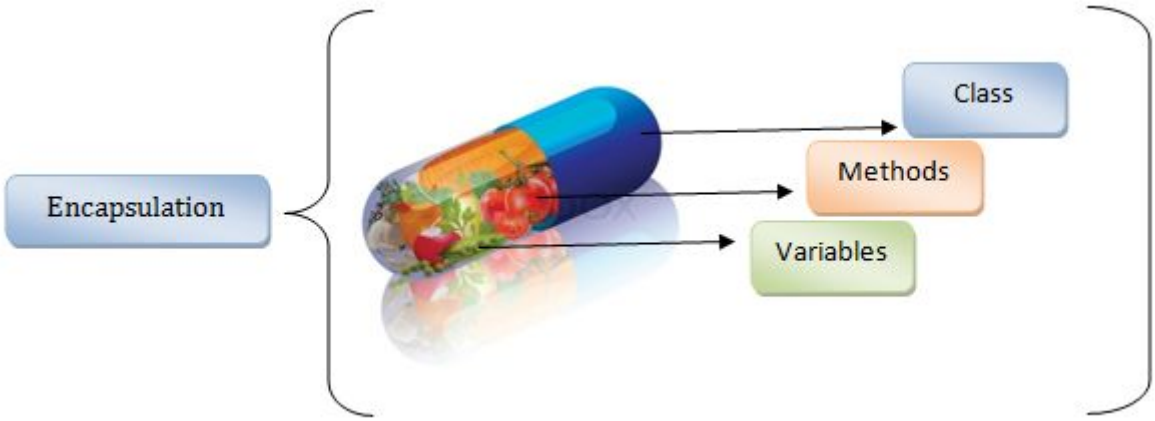
3.1. Inkapseling

Het wijzigen van de interne opbouw van een abstract datatype mag weinig of géén invloed hebben op de rest van de programmatuur.

Hoe wordt dit gerealiseerd?

De methodes die manipulaties uitvoeren op de attributen worden bij elkaar geplaatst en gekoppeld aan dat type (= black box of object). Met andere woorden, de implementatiedetails zitten verborgen

in het object zelf.



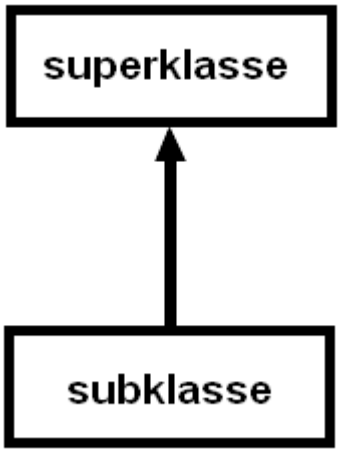
3.2. Abstractie

Elke klasse bevat attributen (instantievariabelen) én een aantal methodes, die de attributen kunnen manipuleren. Een klasse is een specificatie voor objecten. Een object is een instantie van een klasse die gemaakt is volgens deze specificatie.

BESLUIT: de focus in Java ligt op het **maken van objecten** ipv op het schrijven van methodes zoals in procedurele programmeertalen.

3.3. Overerving

Overerving is een mechanisme waarbij software opnieuw wordt gebruikt: nieuwe klassen worden gecreëerd vertrekkende van bestaande klassen, waarbij de attributen en methodes worden geërfd van de superklasse en uitgebreid met nieuwe mogelijkheden, noodzakelijk voor de nieuwe klasse die subklasse noemt.



3.4. Polymorfisme

Met polymorfisme is het mogelijk om systemen te ontwerpen en te implementeren die **eenvoudig uitbreidbaar** zijn.

Klassen die nog niet bestaan tijdens de ontwikkeling van het programma kunnen mits kleine of geen wijzigingen toegevoegd worden aan het generieke deel van het programma op voorwaarde dat deze klassen deel uitmaken van de hiërarchie.

De enige delen van het programma, die gewijzigd of uitgebreid moeten worden, zijn deze die rechtstreeks te maken hebben met de specifieke klasse die toegevoegd wordt.

4. Meer over overerving

4.1. Terminologie

Klassen hebben twee soorten afstammelingen:

- Instanties (of objecten) van een klasse
- Subklassen

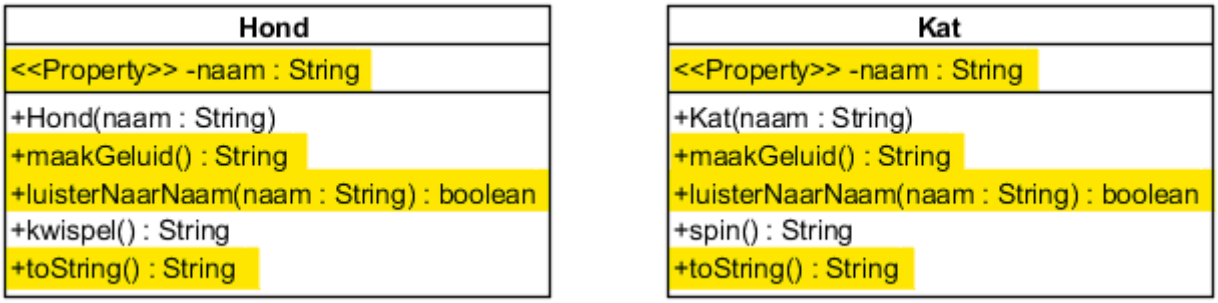
Klassen kunnen

- **Concreet** zijn, wat betekent dat er instanties (objecten) van die klasse kunnen bestaan.
- **Abstract** zijn, wat betekent dat ze geen instanties kunnen hebben.
- Een **interface** klasse zijn, deze bestaan enkel uit operaties. Ze beschrijven een contract (gedrag) zonder verdere specificaties.

4.2. Een voorbeeld

4.2.1. Honden en katten

Laten we beginnen met een voorbeeld. Beschouw volgend klassendiagram:



Merk op dat de getter en setter voor het attribuut naam hier niet meer expliciet vermeld staan. In plaats daarvan is het attribuut genoteerd als Property, wat betekent dat er een getter en/of een setter aanwezig is.

Je merkt dat de klassen Hond en Kat veel overeenkomsten vertonen (zie fluokleur):

- ze hebben hetzelfde attribuut `naam`
- ze hebben dezelfde operaties: `getNaam`, `setNaam`, `maakGeluid`, `luisterNaarNaam`, `toString`

In de Java-code merken we op dat deze methodes meestal ook dezelfde implementatie hebben. Alleen bij `maakGeluid` is dat niet zo.

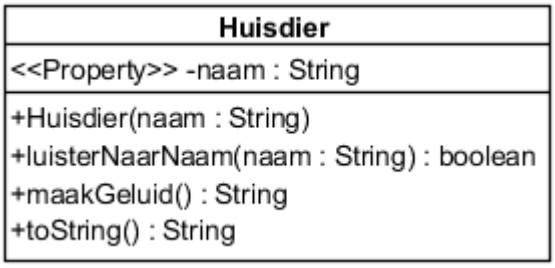
```
1  public class Hond
2  {
3      private String naam;
4
5      public Hond(String naam)
6      {
7          setNaam(naam);
8      }
9      public String getNaam()
10     {
11         return naam;
12     }
13     public final void setNaam(String naam)
14     {
15         this.naam = naam;
16     }
17     public String maakGeluid()
18     {
19         return "waf waf";
20     }
21     public boolean luisterNaarNaam(String naam)
22     {
23         return naam.equals(this.naam);
24     }
25     public String kwispel()
26     {
27         return "kwispel-kwispel-kwispel";
28     }
29     public String toString() ①
30     {
31         return String.format("%s met naam %s", this.getClass().getSimpleName(),
   naam);
32     }
33 }
```

```
1  public class Kat
2  {
3      private String naam;
4
5      public Kat(String naam)
6      {
7          setNaam(naam);
8      }
9      public String getNaam()
10     {
11         return naam;
12     }
13     public final void setNaam(String naam)
14     {
15         this.naam = naam;
16     }
17     public String maakGeluid()
18     {
19         return "miauw";
20     }
21     public boolean luisterNaarNaam(String naam)
22     {
23         return naam.equals(this.naam);
24     }
25     public String spin()
26     {
27         return naam + " spint";
28     }
29     public String toString() ①
30     {
31         return String.format("%s met naam %s", this.getClass().getSimpleName(),
   naam);
32     }
33 }
```

① Zowel bij Hond als Kat wordt gebruik gemaakt van de toString-methode om het dier in een applicatie gemakkelijk te kunnen tonen. Zie verder voor meer informatie hierover.

4.2.2. Generalisatie

Om dubbele code te vermijden passen we **generalisatie** toe: we maken een aparte klasse Huisdier die de gemeenschappelijke kenmerken (attributen, methoden, associaties) bevat. Deze klasse heet een **superklasse**.



De implementatie van de attributen en methodes uit deze klasse in Java ziet er als volgt uit:

```
1 public class Huisdier //extends Object ①
2 {
3     private String naam;
4
5     public Huisdier(String naam) ②
6     {
7         setNaam(naam); ③
8     }
9     public String getNaam()
10    {
11        return naam;
12    }
13    public final void setNaam(String naam) ④
14    {
15        this.naam = naam;
16    }
17    public boolean luisterNaarNaam(String naam) ⑤
18    {
19        return (naam.equals(this.naam));
20    }
21    public String toString() ⑥
22    {
23        return String.format("%s met naam %s", this.getClass().getSimpleName(),
  naam);
24    }
25 }
```

① Aangezien deze klasse niet expliciet van een andere klasse erft, is deze klasse impliciet een extends van de klasse Object. Je mag dit er bij schrijven in de code, maar het is niet noodzakelijk. Dit betekent ook dat deze klasse de methodes van de klasse Object erft. We bekijken verder in dit hoofdstuk welke methodes dit precies zijn en wat je ermee kan doen.

② In de constructor van Huisdier wordt **VOOR** de setter impliciet de constructor van de superklasse Object aangeroepen. Zie verder voor meer informatie over hoe je dit ook expliciet kan doen.

③ Het is belangrijk dat vanuit een constructor geen methodes aangeroepen worden die overridden zijn: wanneer een object van de subklasse wordt gemaakt, kan dit leiden tot een aanroep van een overridden methode door de constructor van de superklasse **VOOR** het subklasse-object volledig geïnitialiseerd is! Dit kan leiden tot fouten wanneer in de overridden-methode gebruik

gemaakt wordt van attributen van de subklasse die nog niet geïnitialiseerd zijn!

④ Zoals we al weten, zijn setters bij voorkeur private of als ze dat niet kunnen zijn, declareren we ze als public final. Als een methode **final** is, kan ze bij overerving niet meer gewijzigd worden. De subklasse moet met de implementatie van deze methode uit de superklasse werken. Private methoden en static-methoden zijn impliciet final. Door de setter private of public final te maken, zorgen we er dus voor dat deze ook zonder problemen in de constructor kan aangeroepen worden.

⑤ Als je vraagt aan het Huisdier om te luisteren naar een bepaalde naam, dan zal het object zijn eigen naam gaan vergelijken met de opgegeven naam. Indien deze namen overeenkomen, dan luistert het dier, anders luistert het niet. Met andere woorden: het dier luistert alleen naar zijn eigen naam!

⑥ De methode toString is één van de methoden die elke klasse direct of indirect overerft van de klasse Object.

- De methode geeft een String voorstelling (tekstuele weergave) weer van een object. Je dient in deze methode dus te schrijven wat je precies van het object wilt te zien krijgen als je in de applicatie iets van het object wilt afbeelden. Voordeel: je hoeft op die manier niet allerlei getters (apart) aan te roepen, maar kan ineens de informatie die in verschillende attributen is opgeslagen, combineren tot één grote String.
- Deze methode wordt impliciet aangeroepen wanneer een object moet geconverteerd worden naar een String. Dat betekent dat je om het object te printen niet noodzakelijk hoeft te schrijven `System.out.println(object.toString());`, maar dat het volstaat om `System.out.println(object);` te gebruiken. Vanzelfsprekend zijn beide vormen wel juist!
- In de implementatie van de toString() methode wordt gebruik gemaakt van `this.getClass().getSimpleName()`. De methode getClass(), hier toegepast op het huidig object (this), is opnieuw een methode die overgeërfd wordt van de klasse Object en die de volledige klassenaam (inclusief het woord "class" en de packagenaam) teruggeeft. Je kan hierop de methode getName() toepassen, waardoor de uitvoer beperkt wordt tot de klassenaam met de package erbij (zonder "class") of je kan getSimpleName() gebruiken, zoals in dit voorbeeld, waardoor enkel de klassenaam zelf getoond wordt (zonder package of "class").

Merk op dat de methode `maakGeluid` nog ontbreekt! Het probleem met deze methode is dat deze een verschillende implementatie heeft in de klasse `Hond` en `Kat`. Hoe lossen we dat nu op?

Als je de zinsnede "if ... then ..., if ... then ... else ..." krijgt in een methode, denk dan even aan overerving.

We gaan hier dus gebruik maken van overerving:

- in de **superklasse** voorzien we een default implementatie (later in OOSD II zullen we deze methode **abstract** maken)
- voor elk van de types maken we een **subklasse** die de implementatie van de methode voor dat type bevat

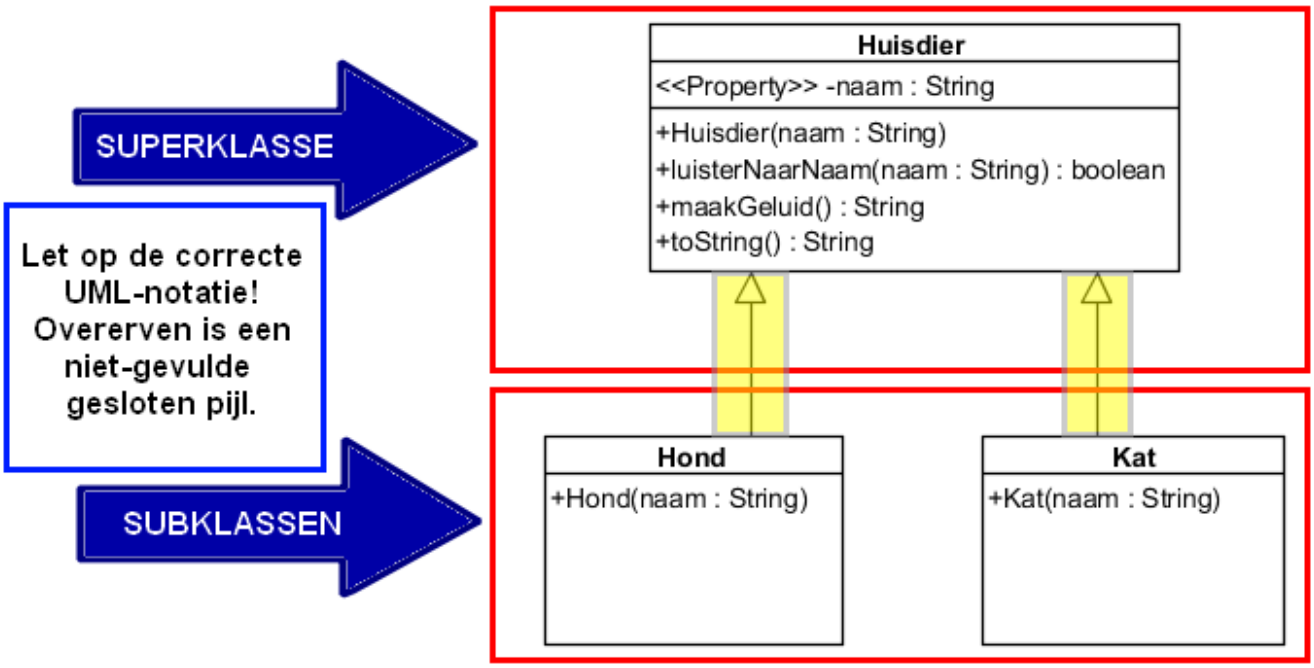
We voorzien dus geen attribuut `type`!

De code in de klasse Huisdier kan bijvoorbeeld de volgende zijn:

```
1     public String maakGeluid()
2     {
3         return "!";
4     }
```

4.2.3. Overerving toepassen in het voorbeeld

Uitgaande van de **superklasse Huisdier** kan je nu 1 of meer specifieke **subklasses** aanmaken (= **specialisatie**)



Overerven is een "is-een"-relatie

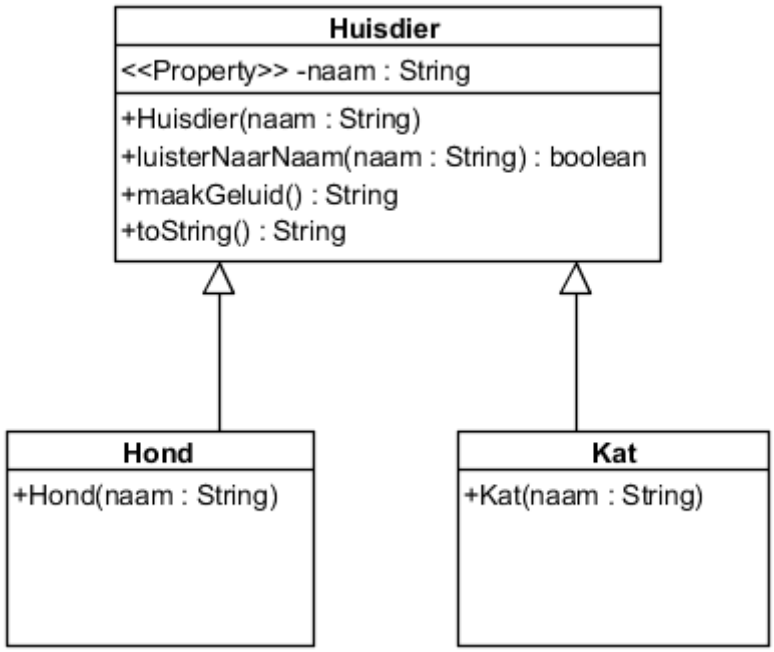
- een hond "is een" huisdier ⇒ de subklasse Hond erft van de superklasse Huisdier
- een kat "is een" huisdier ⇒ de subklasse Kat erft van de superklasse Huisdier
- een auto "is een" voertuig ⇒ de subklasse Auto erft van de superklasse Voertuig

Een klasse kan maar van één andere klasse erven.

Een subklasse erft van zijn superklasse

- de toestand (= de verzameling van attributen)
- de associaties
- het gedrag (= de verzameling van methodes)

In het voorbeeld: Kat en Hond zijn beide subklassen van Huisdier en erven daardoor de volledige toestand (hier: enkel het attribuut `naam`) en alle gedrag (de methodes `getNaam`, `setNaam`, `maakGeluid` en `luisterNaarNaam`) over van hun superklasse Huisdier. Dit staat niet in de subklasse vermeld, dus je ziet het niet in de UML.



Een constructor kan je nooit overerven!

Een subklasse erft van zijn superklasse (Java: keyword `extends`).

Een constructor kan je nooit overerven, maar je kan wel de constructor van de superklasse vanuit een constructor in een subklasse aanroepen met behulp van het keyword `super`. Dit kan wel uitsluitend als eerste opdracht.

Indien je in de subklasseconstructor de constructor van de directe superklasse niet expliciet aanroept, dan gebeurt dit impliciet. In dat geval wordt de default constructor (no-argument constructor `super()`) van de superklasse gebruikt. Indien deze niet aanwezig is in de superklasse, geeft de compiler een foutmelding.

Zorg er in elk geval voor dat de attributen die geërfd worden van de superklasse correct geïnitialiseerd worden!

De implementatie van de subklasses Hond en Kat wordt dan:

```
1 public class Hond extends Huisdier
2 {
3     public Hond(String naam)
4     {
5         super(naam);
6     }
7 }
```

```java
public class Kat extends Huisdier
{
    public Kat(String naam)
    {
        super(naam);
    }
}
```

Opdracht:
Maak een kat Garfield en vraag hem om geluid te maken en zijn naam te zeggen.

4.2.4. Toegangsrechten

Subklassen hebben vanuit code geen toegang tot toestand en gedrag die private zijn in de superklasse

symbool	keyword	betekenis
-	private	nergens zichtbaar in de buitenwereld
+	public	overal zichtbaar in de buitenwereld
#	protected	zichtbaar voor subklassen en package
~		zichtbaar voor alle klassen in de package

Subklassen hebben dus enkel toegang tot public en protected operaties van de superklasse. Eventueel komen daar ook methodes met packagetoegankelijkheid bij als de super- en subklasse zich in dezelfde package bevinden. Om de methodes aan te roepen maak je in de subklasse gebruik van het keyword `super`.

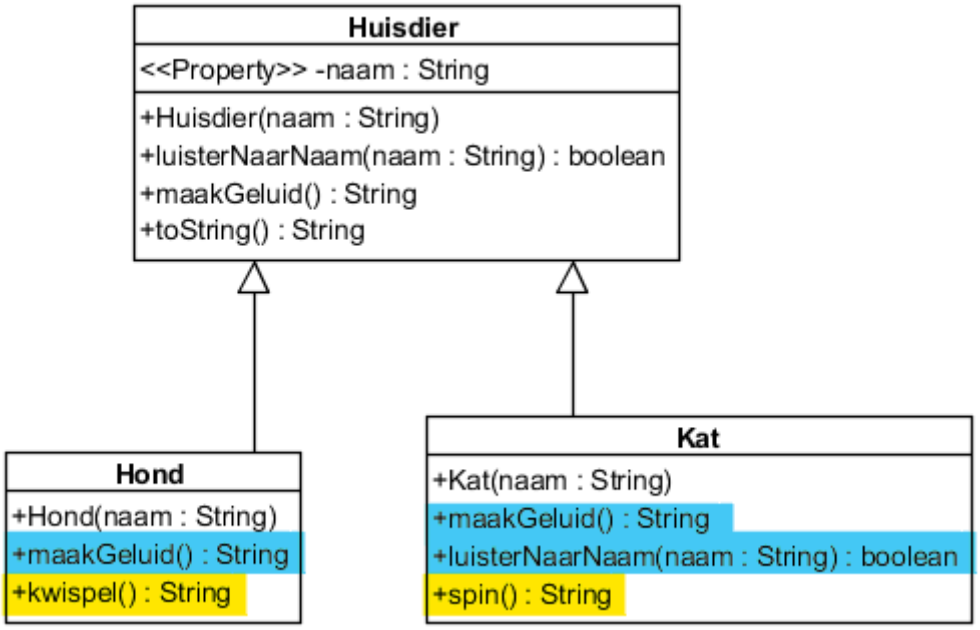
Subklassen erven wel de toestand (attributen) van de superklasse, maar ze kunnen enkel de toestand raadplegen/wijzigen via de getters en setters (op voorwaarde dat deze niet private zijn), daar attributen visibiliteit private hebben.

Voorbeeld: subklasse Hond
Een hond heeft ook een naam (attribuut van de superklasse Huisdier). In de superklasse staat een public final setter voor de naam, dus je kan in de subklasse de naam van de hond wijzigen via super.setNaam(nieuweNaam);

4.2.5. Uitbreiding en specialisatie

In een subklasse kan het gedrag van de superklasse verder **uitgebreid** en/of **gespecialiseerd** worden.

In het voorbeeld passen we de UML als volgt aan:



Uitbreiden gebeurt door nieuwe attributen en/of associates en/of methodes toe te voegen. In de UML hierboven zijn de uitbreidingen in het **geel** aangeduid.

Specialiseren gebeurt door de implementatie van een methode in een subklasse aan te passen. We noemen dit het **overriden** van een methode. We gaan dus de betekenis van deze methode in de subklasse herdefiniëren of overschrijven. De overridden methodes staan in de UML hierboven in het **blauw** aangeduid.

Wanneer je gebruik maakt van **specialisatie**, betekent dit dat je de methode van de superklasse overschrijft. Daarom moet de subklasse altijd een methode declareren met dezelfde signatuur (header) als de methode van de superklasse. Een overriden methode die dus in de superklasse geen informatie teruggeeft (void), kan in de subklasse niet plots een boolean gaan teruggeven. Evenmin is het mogelijk dat een overridden methode in de superklasse 2 parameters nodig heeft, maar dat deze in de subklasse verdwijnen.

De **@Override annotatie** geeft aan dat een methode de methode van de superklasse met dezelfde signatuur overschrijft. Zo niet wordt een compilatiefout gegenereerd. Als je dus @Override boven een methodeheader noteert, controleert de compiler of deze header dezelfde is als in de superklasse. Indien dit niet het geval is, gaat het niet om een overridden methode en moet je dus ofwel de signatuur aanpassen, ofwel de annotatie wegnemen.
Als je de annotatie weglaat, betekent dit echter niet noodzakelijk dat er geen override plaatsvindt. Het is dus niet verplicht om de annotatie te schrijven, maar het is een handig hulpmiddel om het systeem te laten controleren of je wel de juiste methodenaam en parameters en het juiste terugkeerwaardetype hebt gebruikt.

In de klasse **Hond** hebben we dus:

- methode maakGeluid() via specialisatie

```
    public String maakGeluid()
    {
        String geluid = "waf waf";
        geluid += super.maakGeluid();
        return geluid;
    }
```

Het geluid dat een Hond maakt, overschrijft dus het geluid dat een algemeen Huisdier maakt!

- methode kwispel() via uitbreiding

```
    public String kwispel()
    {
        return "kwispel-kwispel-kwispel";
    }
```

En in de klasse **Kat** voegen we volgende code toe:

- methodes maakGeluid() en luisterNaarNaam() via specialisatie

```
    @Override
    public String maakGeluid()
    {
        return "miauw";
    }
```

```
    @Override
    public boolean luisterNaarNaam(String naam)
    {
        SecureRandom r = new SecureRandom();
        if (r.nextInt(10)== 5)
            return super.luisterNaarNaam(naam);
        return false;
    }
```

Ook hier hebben we dus een nieuwe definitie voor de opdracht "geluid maken" die je naar een Kat kan sturen, ten opzichte van dezelfde opdracht, afgevuurd op een algemeen Huisdier.
Voor een Kat betekent het ook iets anders om naar zijn naam te luisteren dan voor een algemeen Huisdier. Waar een algemeen Huisdier altijd luistert als je de juiste naam roept, is het bij een Kat een toevalstreffer: je hebt slechts 1 kans op 10 dat de Kat wel degelijk luistert.

- methode spin() via uitbreiding

```
    public String spin()
    {
        return super.getNaam() + " spint";
    }
```

Merk op dat er geen uitbreiding of specialisatie nodig is voor de methode toString: we behouden gewoon de implementatie van de superklasse, waardoor we het soort Huisdier (Kat of Hond) en de naam ook voor de subklasses kunnen tonen als tekstuele weergave.

4.2.6. Een kleine applicatie

We bekijken even wat we nu allemaal kunnen doen met onze objecten van de super- en subklasses aan de hand van een kleine applicatie:

```java
package cui;

import domein.Hond;
import domein.Huisdier;
import domein.Kat;

public class HuisdierApplicatieMetOvererving
{
    public static void main(String[] args)
    {
        HuisdierApplicatieMetOvererving app = new HuisdierApplicatieMetOvererving();

        ①
        Huisdier hd = new Huisdier("Nijntje");
        Kat k = new Kat("Musti");
        Hond h = new Hond("Rintje");

        ②
        System.out.println("Dit zijn de huisdieren:");
        System.out.println(hd);
        System.out.println(k);
        System.out.println(h);
        System.out.println();

        ③
        System.out.println("Ze kunnen ook allemaal een soort van geluid maken...");
        System.out.printf("%s maakt geluid %s%n", hd.getNaam(), hd.maakGeluid());
        System.out.printf("%s maakt geluid %s%n", k.getNaam(), k.maakGeluid());
        System.out.printf("%s maakt geluid %s%n", h.getNaam(), h.maakGeluid());
        System.out.println();

        ④
        System.out.println("En de kat en de hond kunnen tenslotte nog iets bijzonders:");
        System.out.printf("Het bijzondere wat een %s kan, is %s%n", k.getClass().getSimpleName(), k.spin());
        System.out.printf("Het bijzondere wat een %s kan, is %s%n", h.getClass().getSimpleName(), h.kwispel());
    }
}
```

① Eerst maken we een object aan van elk van de klassen.

② We beelden hier telkens een van de huisdieren af. Daarvoor wordt de methode toString gebruikt, die hier impliciet aangeroepen wordt. In dit geval gaat het telkens om de methode toString uit de superklasse Huisdier aangezien de subklasses geen eigen implementatie hebben. De klassenaam die getoond wordt door het gebruik van `this.getClass().getSimpleName()` zal wel telkens die van de juiste (sub)klasse zijn!

③ Daarna gaan we het geluid van elk dier bepalen. We geven de naam van het dier in kwestie

weer door de methode getNaam() uit de superklasse aan te roepen en vragen het geluid op via de methode maakGeluid(). Voor getNaam() wordt telkens de implementatie uit superklasse Huisdier gebruikt aangezien deze niet wordt overridden in de subklasses. Bij maakGeluid() daarentegen gebruikt elk object van een klasse de implementatie uit de klasse zelf. Zo gebruikt bijvoorbeeld het object k, dat een Kat is, de implementatie uit de klasse Kat.

④ Tenslotte tonen we nog eens de klassenaam van het Kat- en Hond-object (via `this.getClass().getSimpleName()`, dit keer toegepast op het object k en h. Hetzelfde hadden we natuurlijk ook kunnen doen met het object hd. Daarnaast tonen we ook nog eens iets wat specifiek is voor een Kat, door de methode spin() toe te passen op het object k, en iets wat alleen een Hond kan, door de methode kwispel() toe te passen op het object h.

Het resultaat zie je hier:

```
Dit zijn de huisdieren:
Huisdier met naam Nijntje
Kat met naam Musti
Hond met naam Rintje

Ze kunnen ook allemaal een soort van geluid maken...
Nijntje maakt geluid !
Musti maakt geluid miauw
Rintje maakt geluid waf waf!

En de kat en de hond kunnen tenslotte nog iets bijzonders:
Het bijzondere wat een Kat kan, is Musti spint
Het bijzondere wat een Hond kan, is kwispel-kwispel-kwispel
```

4.2.7. Gebruik van het keyword super

We bekijken nog eens de implementatie van de methode maakGeluid in de klasse Hond:

```
1     public String maakGeluid()
2     {
3         String geluid = "waf waf";
4         geluid += super.maakGeluid();
5         return geluid;
6     }
```

Hier wordt aan de tekst "waf waf", die specifiek is voor het geluid dat een Hond maakt, ook nog het uitroepteken geplakt, dat bij het geluid van een Huisdier hoort.

Maar... hoe komen we aan dat geluid uit de superklasse? Als we de methode maakGeluid() aanroepen, kijkt de compiler namelijk naar het "dichtstbijzijnde" voorkomen van deze methode. In dat geval komen we bij de maakGeluid() van de klasse Hond zelf terecht en gaan we dus de methode recursief aanroepen! Dat is niet de bedoeling, want zo zouden we in een oneindige lus terechtkomen!

Om dus expliciet te verwijzen naar de methode uit de superklasse, moeten we gebruik maken van

het keyword **super**. Zoals this een object voorstelt van de huidige klasse, stelt super namelijk een object voor van de superklasse. En aangezien je met een object van een bepaalde klasse kan verwijzen naar methodes van die klasse, kan je met **super** verwijzen naar de methodes van de superklasse.

Analoog gebruiken we ook in de methode luisterNaarNaam van de klasse Kat het keyword **super**:

```
1     @Override
2     public boolean luisterNaarNaam(String naam)
3     {
4         SecureRandom r = new SecureRandom();
5         if (r.nextInt(10)== 5)
6             return super.luisterNaarNaam(naam);
7         return false;
8     }
```

Hier wordt eerst een random geheel getal uit het interval [0,9] bepaald. Als dit getal toevallig 5 is, dan roepen we de methode luisterNaarNaam op van de superklasse Huisdier, die bepaalt of de Kat al dan niet naar de naam luistert door de naam die als parameter meegegeven wordt te vergelijken met de naam die het Huisdier als attribuut heeft (m.a.w. de Kat luistert in dat geval naar de naam als het zijn eigen naam is). Is het random getal iets anders dan 5, dan zal de Kat gewoon niet luisteren!

Beschouw ook nog even een nieuwe subklasse Duif van Huisdier.

```java
package domein;

public class Duif extends Huisdier
{
    private final long ringnr;

    public Duif(String naam, long ringnr)
    {
        super(naam);
        controleerRingnr(ringnr);
        this.ringnr = ringnr;
    }

    public String maakGeluid()
    {
        return "roekoekoe";
    }

    public long getRingnr()
    {
        return ringnr;
    }

    private void controleerRingnr(long ringnr)
    {
        if (ringnr <= 19000000000L || ringnr > 99999999999L)
            throw new IllegalArgumentException("Geen geldig ringnummer");
    }

    public String toString() ①
    {
        return String.format("%s met ringnummer %d", super.toString(), ringnr);
    }
}
```

① In de methode toString gebruiken we opnieuw het keyword **super** om te verwijzen naar de methode toString uit de superklasse. Op die manier kunnen we het gedeelte van de tekstuele weergave dat reeds werd opgebouwd in Huisdier immers hergebruiken, waarbij we de extra informatie in verband met het ringnr nog toevoegen.

Sowieso kunnen we niet volgende code gebruiken voor de toString:

```java
public String toString()
{
    return String.format("%s %s met ringnummer %d", this.getClass().
  getSimpleName(), naam, ringnr);
}
```

Dit geeft namelijk een compilatiefout bij de variabele naam, aangezien deze niet bereikbaar is.

De naam is immers een attribuut van de superklasse, dus daar kunnen we niet rechtstreeks aan.

Een eerste oplossing daarvoor zou kunnen zijn om het attribuut naam in de superklasse **protected** te maken, waardoor het wel toegankelijk is in de subklasse. Echter, in dat geval respecteren we niet meer het principe van inkapseling en kan het attribuut naam dus ook vanuit andere subklasses (of zelfs klasses in dezelfde package) gemanipuleerd worden.

Een tweede mogelijkheid is om het attribuut toch als private eigenschap van Huisdier te behouden, maar gebruik te maken van de (public) getter om de naam op te vragen. De code van de toString-methode in Duif wordt dan:

```
1 public String toString()
2 {
3     return String.format("%s %s met ringnummer %d", this.getClass().
  getSimpleName(), getNaam(), ringnr);
4 }
```

In dit geval gaat het maar om 1 getter, maar het is natuurlijk mogelijk dat je op die manier verschillende gegevens uit de superklasse moet gaan opvragen in de subklasse. Dit zorgt natuurlijk voor extra overhead. Ook dit is dus niet zo interessant. Bovendien kan het gebeuren dat de implementatie van de toString-methode in de superklasse op een bepaald moment moet wijzigen en dan zouden we er moeten op letten om deze ook in de subklasse aan te passen. Vandaar is het dus beter om zoveel mogelijk de code uit de superklasse te hergebruiken!

> We merken nog op dat je de methodes uit de superklasse kunt aanroepen zonder het keyword **super** te gebruiken als er geen verwarring mogelijk is. In de klasse Duif mag je dus getNaam(), super.getNaam() en zelfs this.getNaam() door elkaar gebruiken. We raden wel aan om dit eenduidig te doen en dus bijvoorbeeld overal getNaam() te gebruiken.

Tenslotte kan het keyword **super** ook nog, zoals we al eerder vermeldden, gebruikt worden als naam van de superklasseconstructor. Ook daarvan zie je in de klasse Duif een voorbeeld. De superklasseconstructor wordt hier expliciet aangeroepen en komt dus als eerste statement voor in de constructor van Duif. Daarna volgt nog de controle op en het toekennen van het ringnummer. Bij het aanmaken van een object van de subklasse Duif wordt dus eerst een object van de superklasse Huisdier aangemaakt en daarna worden dingen toegevoegd die specifiek zijn voor een Duif (in casu: het ringnummer).

In bovenstaande klasse Duif wordt trouwens ook gebruik gemaakt van het "principe van het kleinste voorrecht" (principle of least privilege) dat ons voorschrijft om enkel de toegang te verlenen die nodig is om de vooropgestelde taak uit te voeren, niets meer. Zo kan de javacode niet per ongeluk gewijzigd worden.

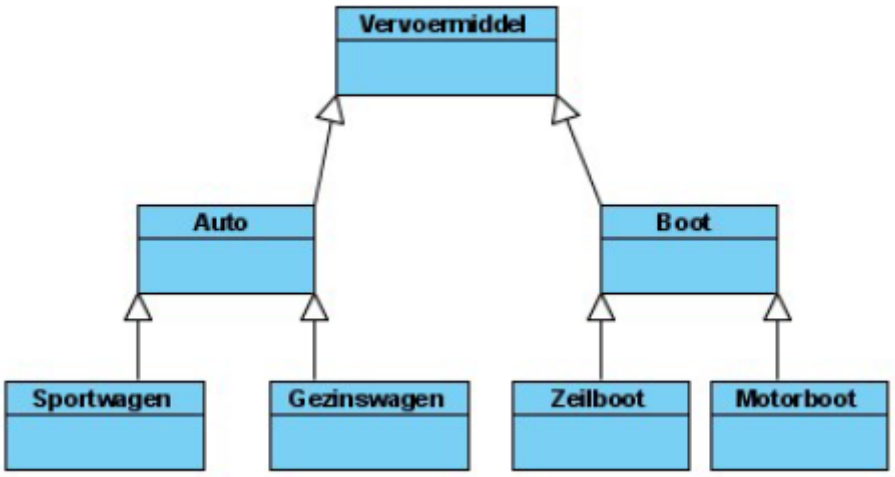
We declareren daarom het attribuut ringnr, dat niet mag gewijzigd worden, final. Uit hoofdstuk 5 nemen we mee dat een final attribuut bij declaratie of in elke constructor een waarde moet krijgen en dat we voor dit attribuut GEEN set-methode maken. Het feit dat het attribuut final moet zijn, vind je niet terug in de UML, maar je zal in de opgave lezen dat het om een niet wijzigbaar gegeven gaat en je vindt geen setter terug, maar wel een controleermethode. In de constructor zal je dus in

plaats van de setter aan te roepen, eerst de controleermethode moeten aanroepen en daarna nog de waarde van het attribuut moeten toekennen.

4.3. Meer dan één niveau van overerving

Een klasse die erft van een andere klasse kan op zijn beurt terug de superklasse zijn van een derde klasse.

Bijvoorbeeld:



- Auto erft/is subklasse van Vervoermiddel
- Sportwagen erft/is subklasse van Auto
- Dus Sportwagen erft/is subklasse van Vervoermiddel

We spreken van *directe* sub- en superklasses wanneer de ene klasse precies 1 niveau hoger staat dan de andere. Bijvoorbeeld: de *directe* subklasses van Vervoermiddel zijn Auto en Boot. De klasse Boot is de *directe* superklasse van Zeilboot.
Een superklasse kan meerdere *directe* subklassen hebben.
Bij een subklasse hoort steeds één *directe* superklasse.

Waarom zou je dergelijke subklasse hiërarchieën gebruiken?

Deze manier van werken laat toe om beschrijving van complexe klassen stapgewijs op te bouwen. Bijvoorbeeld: Huisdier – Hond – Rashond - …

Bovendien is dit een gemakkelijk onderhoudbaar systeem: deze manier van werken laat aanpassingen toe die gelokaliseerd zijn op het niveau waar ze gespecifieerd zijn. Bijvoorbeeld: als er iets moet veranderen aan de methode getNaam(), dan dien je maar op 1 plaats aan te passen (in de klasse Huisdier).

Tenslotte is het ook eenvoudig om nieuwe subklassen toe te voegen zonder de reeds bestaande klassen te beïnvloeden. Bijvoorbeeld: zo hebben we bijvoorbeeld gezien dat als er een nieuw soort huisdier (zoals Duif) moet worden aangemaakt, het volstaat om hiervoor een nieuwe klasse te schrijven die een extends is van Huisdier. Huisdier zelf en de andere subklasses hebben daar geen last van.

Ter illustratie hebben we ook nog een nieuwe applicatie gemaakt met de 3 subklasses (Kat, Hond en Duif). De code daarvan vind je hier:

```
1  package cui;
2
3  import domein.Duif;
4  import domein.Hond;
5  import domein.Huisdier;
6  import domein.Kat;
7
8  public class HuisdierApplicatieMet3Subklasses
9  {
10     public static void main(String[] args)
11     {
12         HuisdierApplicatieMet3Subklasses app = new
   HuisdierApplicatieMet3Subklasses();
13
14         Huisdier hd = new Huisdier("Nijntje");
15         Kat k = new Kat("Musti");
16         Hond h = new Hond("Rintje");
17         Duif d = new Duif("Wittekop", 20201234567L); ①
18
19         System.out.println("Dit zijn de huisdieren:");
20         System.out.println(hd);
21         System.out.println(k);
22         System.out.println(h);
23         System.out.println(d);  ②
24         System.out.println();
25     }
26 }
```

① Het object van de nieuwe subklasse klasse Duif wordt aangemaakt.

② Het object van de klasse Duif wordt hier afgebeeld zoals in de methode toString werd beschreven.

De uitvoer van dit programma ziet er als volgt uit:

```
Dit zijn de huisdieren:
Huisdier met naam Nijntje
Kat met naam Musti
Hond met naam Rintje
Duif met naam Wittekop met ringnummer 20201234567
```

4.4. De klasse Object

Alle klassen erven direct of indirect van de klasse Object. Aangezien een klasse in Java maar van één klasse direct kan ervan, erf je dus alleen indirect van Object als er al een extends staat. Indien er geen extends aanwezig is in de klasseheader, dan betekent dit impliciet dat er “extends Object” staat, zodat je dus directe overerving hebt.

Maar wat houdt deze directe of indirecte overerving nu in? In de API vinden we terug dat de klasse Object 11 methoden bevat. Elke klasse erft dus deze 11 methoden van Object.

Modifier and Type	Method	Description
protected Object	clone()	Creates and returns a copy of this object.
boolean	equals(Object obj)	Indicates whether some other object is "equal to" this one.
protected void	finalize()	**Deprecated.** The finalization mechanism is inherently problematic.
Class<?>	getClass()	Returns the runtime class of this Object.
int	hashCode()	Returns a hash code value for the object.
void	notify()	Wakes up a single thread that is waiting on this object's monitor.
void	notifyAll()	Wakes up all threads that are waiting on this object's monitor.
String	toString()	Returns a string representation of the object.
void	wait()	Causes the current thread to wait until it is awakened, typically by being *notified* or *interrupted*.
void	wait(long timeoutMillis)	Causes the current thread to wait until it is awakened, typically by being *notified* or *interrupted*, or until a certain amount of real time has elapsed.
void	wait(long timeoutMillis, int nanos)	Causes the current thread to wait until it is awakened, typically by being *notified* or *interrupted*, or until a certain amount of real time has elapsed.

Een aantal van die methodes kennen we al:

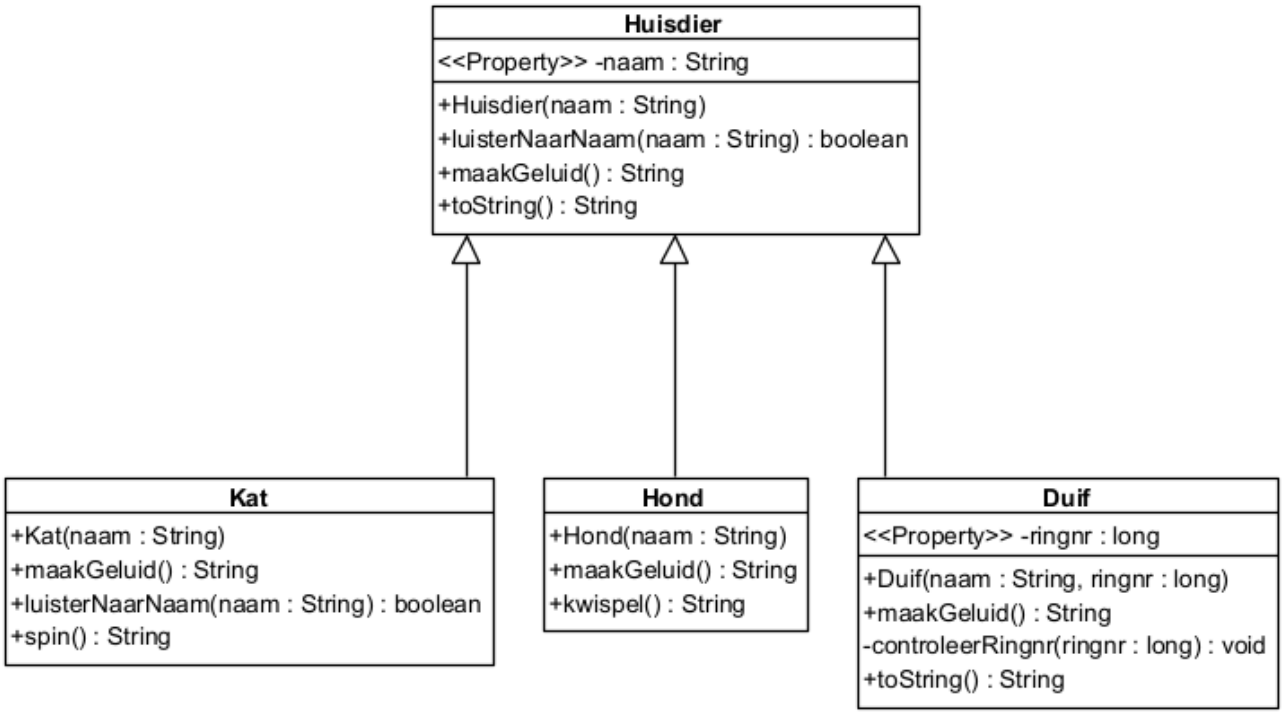
- We gebruiken de methode **equals** om 2 objecten inhoudelijk met elkaar te vergelijken. De klasse String overschrijft (overrides) deze methode.
- De methode **getClass** (in combinatie met getSimpleName) waarmee je de klasse kunt opvragen waartoe het object behoort.
- De methode **toString** die een tekstuele weergave van het object teruggeeft.

Deze drie methodes kan je dus sowieso op elk object toepassen.

5. Meer over polymorfisme

5.1. Wat is polymorfisme?

Beschouwen we opnieuw het voorbeeld met de superklasse Huisdier en de subklasses Kat, Hond en Duif.



We hebben eerder instanties gemaakt van elk van deze klasses als volgt:

```
1 Huisdier hd = new Huisdier("Nijntje");
2 Kat k = new Kat("Musti");
3 Hond h = new Hond("Rintje");
4 Duif d = new Duif("Wittekop", 20201234567L);
```

Zoals we hebben gezien, konden we hiermee perfect werken in de applicatie. Toch kan het nog beter. Dat gaat dan zo:

```
1 Huisdier dierVanJan, dierVanEls, dierVanTante, dierVanOpa;
2 dierVanJan = new Huisdier("Nijntje");
3 dierVanEls = new Kat("Musti");
4 dierVanTante = new Hond("Rintje");
5 dierVanOpa = new Duif("Wittekop", 20201234567L);
```

Doordat alle dieren nu gedeclareerd zijn als Huisdier-referenties, kan je er natuurlijk ook gewoon een object van een andere subklasse van Huisdier aan toekennen. Dus het volgende is perfect mogelijk...

```
1 Huisdier dierVanOpa = new Duif("Wittekop", 20201234567L);
2 /* Doe van alles met deze Duif */
3 dierVanOpa = new Kat("Garfield");
4 /* Alles wat je vanaf nu nog doet met het object dierVanOpa heeft betrekking op een
  Kat */
```

Volgens deze code heeft opa zijn oorspronkelijk huisdier, dat een duif was, dus niet langer in zijn bezit, maar heeft hij in de plaats een kat in huis genomen!

Hierdoor begrijpen we wat **polymorfisme** in essentie is: de term **veelvormigheid** komt vanuit het feit dat een object vele vormen kan aannemen. In dit geval kan een Huisdier een Kat, Hond, Duif of zelfs een ander Huisdier zijn. Het huisdier van opa was eerst een duif en werd later een kat.

5.2. Gebruik van polymorfisme

Beschouw volgende code:

```
1 Huisdier dierVanEls = new Kat("Musti");
2 System.out.println(dierVanEls.maakGeluid());
```

Wat gaat er nu geprint worden? `!` omdat dierVanEls een Huisdier is of `miauw` omdat dierVanEls naar een Kat-object verwijst?

In Java zijn er twee types van polymorfisme:

- **static of compile-time** polymorfisme
- **dynamic of runtime** polymorfisme

Dankzij **static binding** gaat de compiler eerst op zoek naar de methode in de klasse van je referentie. Dat betekent dat je een compilatiefout zou krijgen als in het voorbeeld hierboven in de klasse Huisdier geen methode maakGeluid zonder parameters zou voorkomen. Ook zorgt dit ervoor dat je in het geval van methode overloading (2 of meer methodes die dezelfde naam dragen) de juiste vorm van de methode gekozen wordt.

Maar de grote kracht van polymorfisme ligt in de **dynamische binding**. Bij het runnen van de code wordt namelijk gekeken naar het werkelijke object (hier: een Kat) en wordt onderzocht of de methode (hier: maakGeluid) in deze klasse zelf voorkomt. Indien dit (zoals hier) het geval is, dan wordt deze code uitgevoerd. Indien niet, dan gaan we in de hiërarchie omhoog via de directe superklasse en eventuele indirecte superklasses tot we ergens de methode tegenkomen.

Bijvoorbeeld als we als code hebben

```
1 System.out.println(dierVanEls);
```

Er komt geen compilatiefout op deze code aangezien de methode **toString** die hier moet worden toegepast op het object dierVanEls voorkomt in de klasse Huisdier. De **static binding** zou trouwens ook lukken als de methode niet voorkomt in deze klasse, aangezien het een van de methodes van de klasse Object is, die dus sowieso altijd overgeërfd wordt door elke klasse.

Wanneer we echter naar de **dynamische binding** gaan kijken, dan moeten we op zoek naar de toString-methode in de klasse Kat. Die komt daar niet voor, dus dan moeten we omhoog in de hiërarchie. Kat erft van Huisdier, dus daar gaan we als tweede op zoek. Aangezien er in Huisdier wel een versie van toString staat, is het deze code die zal uitgevoerd worden, met andere woorden: de volgende output zal op het scherm verschijnen: `Kat met naam Musti`

Maar wat als we het volgende proberen te doen?

```
System.out.println(dierVanEls.spin());
```

Hier krijgen we wel degelijk een compilatiefout aangezien de **static binding** niet lukt! Er is namelijk geen methode spin in de klasse Huisdier aanwezig (alleen katten kunnen spinnen, andere huisdieren niet). Maar... het dier van Els is wel degelijk een kat! Weet deze kat dan niet meer hoe ze moet spinnen?

Het probleem is dat er geen methode spin bestaat in de superklasse Huisdier. We moeten dus het dierVanEls, dat een Huisdier is, eerst **downcasten** naar een Kat. Dat gaat zo:

```
Kat katVanEls = (Kat) dierVanEls;
```

Het gaat hier om hetzelfde object, maar we verwijzen er nu naar met een Kat-referentie. En in de klasse Kat vinden we wel degelijk de methode spin terug, dus onze kat kan nu weer spinnen...

```
System.out.println(katVanEls.spin());
```

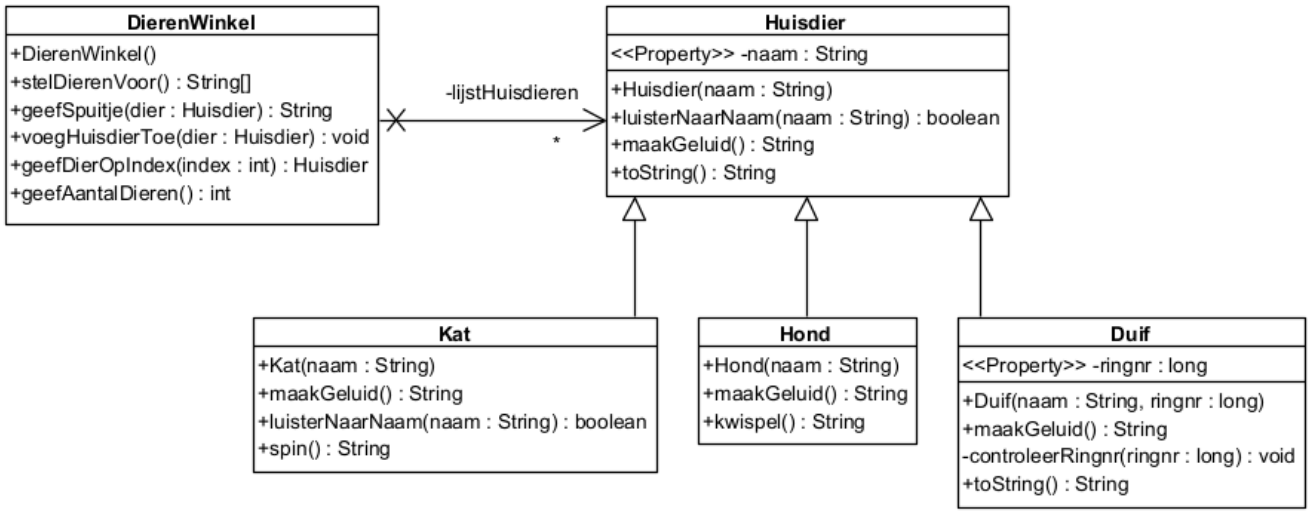
We krijgen dus als resultaat: `Musti spint`.

Let op: het dierVanEls kan alleen gedowncast worden naar een Kat-object, niet naar (bijvoorbeeld) een Hond-object omdat dierVanEls natuurlijk niet verwijst naar een Hond, maar wel naar een Kat.

Je kan dus casten van een bepaald supertype naar een subtype. Om te checken of een object wel van een bepaald subtype is, gebruik je `instanceof`. Bovenstaande code kan je dus nog wat "veiliger" maken, door eerst deze controle toe te voegen als volgt:

```
Huisdier dierVanEls = new Kat("Musti");
if (dierVanEls instanceof Kat)
{
    Kat katVanEls = (Kat) dierVanEls;
    System.out.println(katVanEls.spin());
}
else
{
    System.out.println("Dit dier kan niet spinnen!");
}
```

5.3. Waarom polymorfisme?



We vullen de UML van Huisdier / Kat / Hond / Duif aan met een nieuwe klasse DierenWinkel. Uit de associatie tussen DierenWinkel en Huisdier kan je afleiden dat er een groep huisdieren moet kunnen worden bijgehouden. We kiezen voor de implementatie via een List<Huisdier>.

```
1     private List<Huisdier> lijstHuisdieren;
```

5.3.1. De code wordt eenvoudiger door polymorfisme

Dankzij polymorfisme wordt de code eenvoudiger. Als we aparte referenties van Hond-, Kat- en Duifobjecten willen bijhouden, dan hebben we 3 verschillende lijsten nodig. Hier gebruiken we Huisdier-referenties en kunnen we dus in de lijst zowel Honden, Katten als Duiven bijhouden!

Om een DierenWinkel aan te maken (constructor), kunnen we dus de lijst met Huisdieren eerst creëren en dan de dieren (in willekeurige volgorde) toevoegen.

```
1  public DierenWinkel()
2  {
3      lijstHuisdieren = new ArrayList<>();
4      lijstHuisdieren.add(new Duif("Blauwe geschelpte", 20180000001L));
5      lijstHuisdieren.add(new Kat("Minoe"));
6      lijstHuisdieren.add(new Hond("Rex"));
7      lijstHuisdieren.add(new Hond("Lassie"));
8      lijstHuisdieren.add(new Duif("Witoog", 20199876543L));
9      lijstHuisdieren.add(new Kat("Garfield"));
10 }
```

Ook bepaalde methodes worden op die manier eenvoudiger. Zo kan je bijvoorbeeld een methode stelDierenVoor schrijven die alle dieren (ongeacht of het katten, honden of duiven zijn) overloopt en daarvan de tekstuele weergave (toString) toont, gevolgd door het geluid dat dit dier maakt. Dat gaat zo:

```
1     public String[] stelDierenVoor()
2     {
3         String[] voorstelling = new String[lijstHuisdieren.size()];
4         int index = 0;
5         for (Huisdier dier : lijstHuisdieren)
6         {
7             voorstelling[index] = String.format("%s zegt %s", dier.toString(),
  dier.maakGeluid());
8             index++;
9         }
10        return voorstelling;
11    }
```

5.3.2. Parameters en returntypes kunnen polymorf gedefinieerd worden

Een methode kan zo gemaakt worden dat je om het even welk object uit de hiërarchie kan gebruiken als parameter en/of returnwaarde.

Bijvoorbeeld:

```
1     public String geefSpuitje(Huisdier dier)
2     {
3         return String.format("Awwwwww... dat doet pijn! %s", dier.maakGeluid());
4     }
```

We kunnen aan om het even welk dier een spuitje geven. Dat doet telkens een beetje pijn, dus het dier maakt geluid om zijn onbehagen te uiten. We geven als parameter een Huisdier-object mee, maar de versie van maakGeluid die wordt uitgevoerd is die van de passende (sub)klasse waartoe het object eigenlijk behoort.

Analoog kunnen we op deze manier ook de constructor "verlichten". In plaats van de huisdieren hier één voor één toe te voegen, kunnen we hiervoor een methode voegHuisdierToe maken, die ervoor zorgt dat we in de constructor enkel nog de initialisatie van het attribuut kunnen zetten, waarna we achteraf de lijst kunnen opvullen.

```
1     public DierenWinkel()
2     {
3         lijstHuisdieren = new ArrayList<>();
4     }
5     public void voegHuisdierToe (Huisdier dier)
6     {
7         if (dier != null )
8             lijstHuisdieren.add(dier);
9     }
```

In het derde voorbeeld, bij de methode geefDierOpIndex, is het returntype een Huisdier. We halen namelijk één element uit de lijst en dit kan zowel een Kat, Hond of Duif zijn.

```
    public Huisdier geefDierOpIndex(int index)
    {
        return lijstHuisdieren.get(index);
    }
```

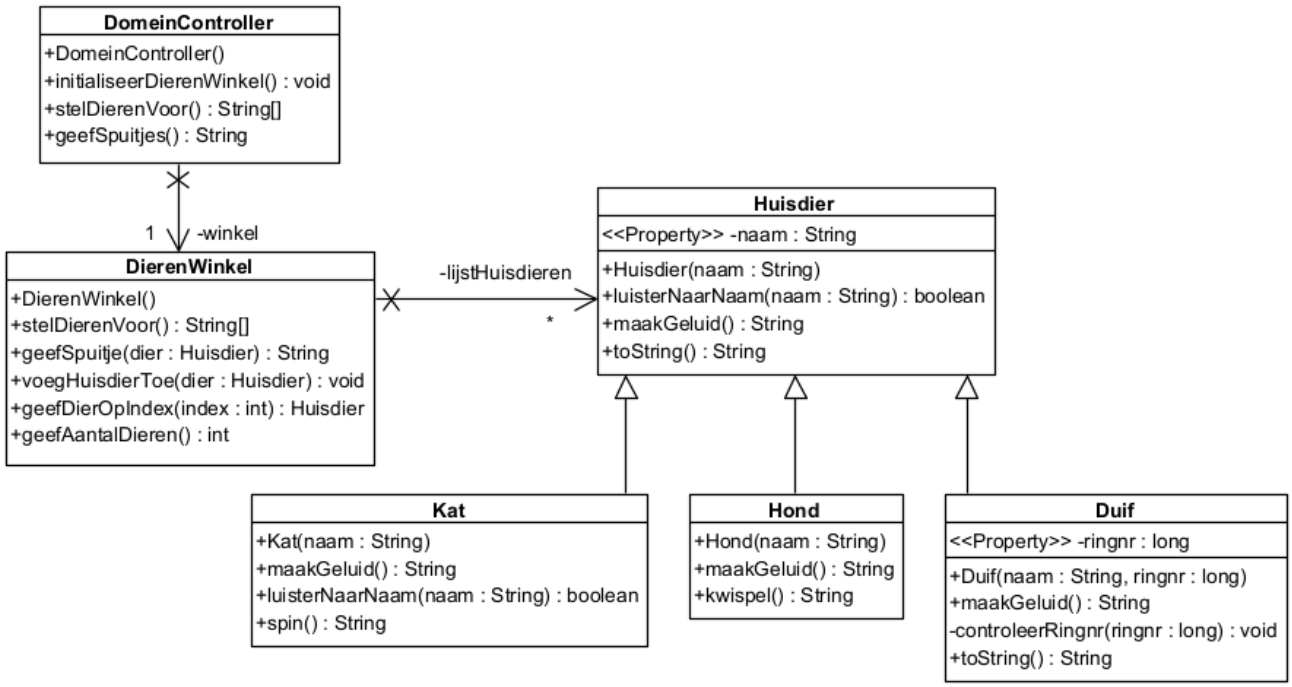
5.3.3. Een polymorfe hiërarchie is gemakkelijk uitbreidbaar

Zoals we gezien hebben bij het toevoegen van de klasse Duif aan de hiërarchie, is het gemakkelijk om een extra subklasse in te passen. Ook nu, als we kijken naar de implementatie van DierenWinkel, dan zien we dat er niets zou veranderen als we bijvoorbeeld nog een klasse Hamster zouden willen toevoegen. De methodes maakGeluid en toString die we in de DierenWinkel willen kunnen aanroepen, kunnen in de nieuwe klasse Hamster overridden worden of men kan de defaultimplementatie uit de klasse Huisdier gebruiken. In geen van beide gevallen dient er iets te wijzigen aan de implementatie van DierenWinkel door het toevoegen van deze extra klasse.

Aangezien Hamster de volledige toestand en het gedrag van Huisdier overerft, dien je enkel de uitbreidingen en/of specialisaties toe te voegen aan Hamster. In zijn meest eenvoudige vorm kan Hamster er dus zo uitzien:

```
public class Hamster extends Huisdier
{
    public Hamster(String naam)
    {
        super(naam);
    }
}
```

5.3.4. Toegepast op het voorbeeld



In het voorbeeld hebben we dus een extra klasse DierenWinkel nodig, waarvan we alle methodes hierboven al besproken hebben, behalve deze:

```
1     public int geefAantalDieren()
2     {
3         return lijstHuisdieren.size();
4     }
```

Deze methodes worden allemaal gebruikt in de DomeinController, die er als volgt uitziet:

```java
package domein;

import java.security.SecureRandom;

public class DomeinController
{
    private DierenWinkel winkel;

    public DomeinController() ①
    {
        winkel = new DierenWinkel();
    }

    public void initialiseerDierenWinkel() ②
    {
        Duif duif1 = new Duif("Blauwe geschelpte", 20180000001L);
        winkel.voegHuisdierToe(duif1);
        Kat kat1 = new Kat("Minoe");
        winkel.voegHuisdierToe(kat1);
        Hond hond1 = new Hond("Rex");
        winkel.voegHuisdierToe(hond1);
        Hond hond2 = new Hond("Lassie");
        winkel.voegHuisdierToe(hond2);
        Duif duif2 = new Duif("Witoog", 20199876543L);
        winkel.voegHuisdierToe(duif2);
        Kat kat2 = new Kat("Garfield");
        winkel.voegHuisdierToe(kat2);
    }

    public String[] stelDierenVoor() ③
    {
        return winkel.stelDierenVoor();
    }

    public String geefSpuitjes() ④
    {
        SecureRandom random = new SecureRandom();
        String uitvoer = "";
        for (int i=0; i<3; i++)
        {
            int aantalDieren = winkel.geefAantalDieren();
            int randomIndex = random.nextInt(aantalDieren);
            Huisdier patient = winkel.geefDierOpIndex(randomIndex);
            uitvoer += String.format("%s%n", winkel.geefSpuitje(patient));
        }
        return uitvoer;
    }
}
```

① In de constructor wordt een DierenWinkel gemaakt.

② De methode initialiseerDierenWinkel zorgt ervoor dat er enkele dieren aan de winkel worden toegevoegd. Hiervoor wordt de methode voegHuisdierToe van de DierenWinkel gebruikt.

③ De methode stelDierenVoor roept de gelijknamige methode van de DierenWinkel aan en geeft het resultaat verder door.

④ De methode geefSpuitjes laat ons toe om 3 keer een spuitje te geven. Daarvoor wordt telkens een random index bepaald die in het interval [0,aantalDieren[ligt. Om het aantal dieren te kennen, wordt de methode geefAantalDieren van de DierenWinkel gebruikt. Het dier dat het spuitje moet ondergaan (= de patiënt) wordt dan opgehaald uit de winkel via de methode geefDierOpIndex en tenslotte wordt de methode geefSpuitje uit de DierWinkel klasse uitgevoerd met als parameter de patiënt en wordt het resultaat daarvan toegevoegd aan de uitvoerstring.

Als we dan een kleine applicatie maken, zou die er als volgt kunnen uitzien:

```
1  package cui;
2
3  import domein.DomeinController;
4
5  public class HuisdierApplicatieMetPolymorfisme
6  {
7      public static void main(String[] args)
8      {
9          ①
10         DomeinController dc = new DomeinController();
11
12         ②
13         dc.initialiseerDierenWinkel();
14
15         ③
16         String[] overzicht = dc.stelDierenVoor();
17         for (String eenDier : overzicht)
18             System.out.println(eenDier);
19         System.out.println();
20
21         ④
22         System.out.println(dc.geefSpuitjes());
23     }
24 }
```

① Eerst wordt een DomeinController-object gemaakt om te communiceren met de domeinlaag.

② We roepen dan de methode initialiseerDierenWinkel aan om een aantal dieren (vast gecodeerd in de domeinlaag) toe te voegen aan de winkel.

③ De aanroep van de methode stelDierenVoor levert een array van Strings op. Deze kan doorlopen worden met een enhanced for om telkens de gegevens van één dier vast te krijgen. Deze gegevens worden dan telkens geprint en op het einde wordt nog een lege regel toegevoegd.

④ Tenslotte zorgt de methode geefSpuitjes ervoor dat we het resultaat zien van de toediening van 3 spuitjes op een random dier uit de winkel.

Het resultaat van een run van deze code zou dan kunnen zijn:

```
Duif met naam Blauwe geschelpte met ringnummer 20180000001 zegt roekoekoe
Kat met naam Minoe zegt miauw
Hond met naam Rex zegt waf waf!
Hond met naam Lassie zegt waf waf!
Duif met naam Witoog met ringnummer 20199876543 zegt roekoekoe
Kat met naam Garfield zegt miauw

Awwwwww... dat doet pijn! miauw
Awwwwww... dat doet pijn! miauw
Awwwwww... dat doet pijn! waf waf!
```

5.4. Andere voorbeelden van polymorfisme

5.4.1. Vierhoeken

De superklasse **Vierhoek** kan verschillende vormen aannemen:

- subklasse Rechthoek
- subklasse Vierkant
- subklasse Parallellogram
- subklasse Trapezium
- ...

Methodes die in de superklasse kunnen gedefinieerd worden, maar dan telkens overridden zullen worden in de subklasse zijn:

- methode `teken() : void` - bepaalt hoe we een Vierhoek zullen tekenen
- methode `berekenOppervlakte() : double` - bepaalt hoe we de oppervlakte van dit type Vierhoek moeten bepalen
- methode `berekenOmtrek() : double` - bepaalt hoe we de omtrek van dit type Vierhoek moeten bepalen

5.4.2. Rekeningen

De superklasse **Rekening** heeft bepaalde eigenschappen en methoden die kunnen worden overgeërfd door:

- subklasse SpaarRekening
- subklasse ZichtRekening
- ...

De methode `haalAf(bedrag : double) : void` kan anders geïmplementeerd worden naargelang de klasse waarin ze voorkomt. Zo kan het zijn dat je bij een algemene Rekening om het even welk positief bedrag mag afhalen, terwijl je bij een SpaarRekening alleen maar mag sparen en dus niet

onder 0 mag gaan door een bedrag af te halen. Bij een ZichtRekening is het dan weer de bedoeling dat je wel onder 0 mag gaan, maar slechts tot een bepaalde grens.

5.4.3. Werknemers

De superklasse **Werknemer** kan verschillende subklassen hebben:

- subklasse WerknemerOpCommissie
- subklasse WerknemerMetUurloon
- subklasse WerknemerMetVastSalaris
- ...

En de subklasse **WerknemerOpCommissie** heeft zelf ook nog een subklasse: WerknemerOpCommissieMetBasisloon

Er is een methode `berekenLoon() : double` die een andere implementatie heeft in elke klasse:

- Voor een algemene Werknemer weten we nog niet hoe we het loon gaan berekenen, dus zullen we hier bijvoorbeeld de waarde 0 teruggeven.
- Een WerknemerOpCommissie krijgt een bepaald commissiePercentage van het verkochtBedrag uitbetaald.
- Een WerknemerMetUurLoon krijg het uurloon vermenigvuldigd met het totaal aantal gewerkte uren.
- Een WerknemerMetVastSalaris krijgt dit vastSalaris uitbetaald.
- Een WerknemerOpCommissieMetBasisloon krijgt bovenop het loon die een WerknemerOpCommissie krijgt, nog een basisloon.

6. Drielagenmodel

6.1. Waarom drie lagen?

Met al wat we in deze cursus geleerd hebben, kunnen we reeds een "goede" applicatie schrijven. Maar wat is nu een goede applicatie?

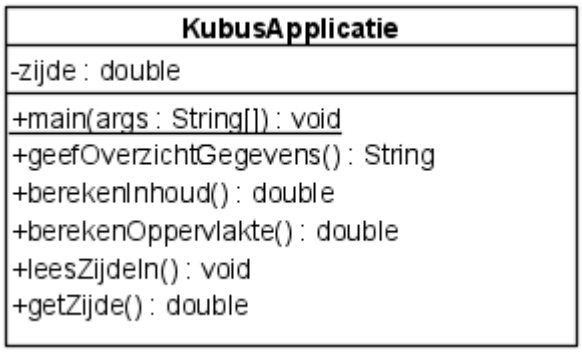
Deze moet:

- aan de verwachtingen voldoen
- herbruikbare onderdelen bevatten
- uitbreidbaar en onderhoudbaar zijn
- performant en stabiel / bugvrij zijn
- gebruiksvriendelijk en veilig zijn

Maar... dat loopt vaak mis! Stel dat we bijvoorbeeld een eenvoudige applicatie maken om kubusvormige doosjes te kunnen maken voor een klant die in de cadeautjestijd snel wil kunnen

berekenen hoeveel materiaal hij nodig heeft om de gewenste doos te maken (oppervlakteberekening) en ook wil kunnen zeggen hoe groot de doos dan zal worden (inhoudsberekening).

Ons eerste idee leggen we vast in een DCD:



We bedenken ons dat we in deze consoleapplicatie zowel in- en uitvoer als berekeningen aan het doen zijn, maar daar zal onze klant toch niks van merken, zeker?

Alleen… het idee van een consoleapplicatie bevalt onze klant niet! Hij wil een GUI (graphical user interface). We moeten dus herbeginnen! Onze applicatie voldeed niet aan de verwachtingen en was niet gebruiksvriendelijk.

Bovendien merken we bij het herwerken van onze applicatie dat we eigenlijk ook geen rekening gehouden hebben met dingen die herbruikbaar zouden moeten zijn in deze nieuwe versie en dat onze applicatie dus ook niet makkelijk uitbreidbaar en onderhoudbaar is!

Dit is het gevolg van een slecht ontwerp!

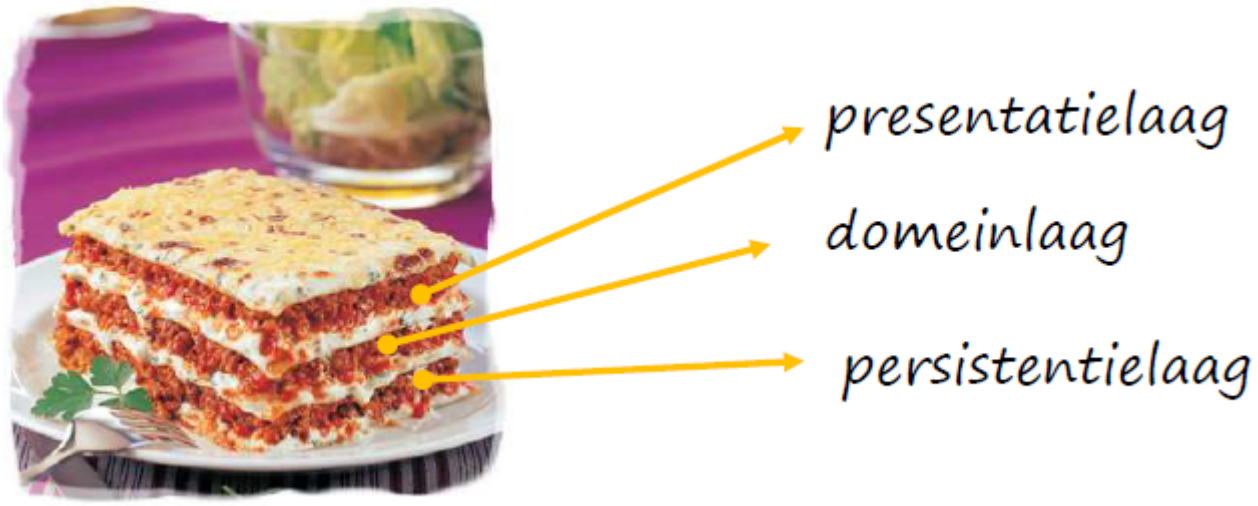
Voor een goed objectgeoriënteerd ontwerp moet elke klasse eigen duidelijk afgebakende verantwoordelijkheden krijgen. Daarvoor hebben we GRASP gebruikt… maar dat is blijkbaar een beetje misgegaan bij bovenstaand ontwerp, want de verantwoordelijkheden voor in- en uitvoer en verwerking horen niet allemaal in dezelfde klasse! Op deze manier gaan we spaghetticode schrijven!

6.2. Welke zijn de drie lagen?

In een objectgeoriënteerde softwarearchitectuur groeperen we klassen volgens hun verantwoordelijkheden:

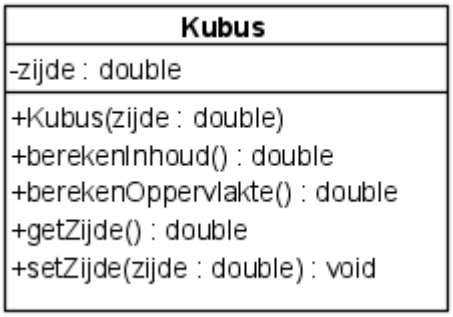
- klassen die **de communicatie met de gebruiker** verzorgen
- klassen die **het hart van de applicatie** doen kloppen
- klassen die **de communicatie met een databank** kunnen verzorgen

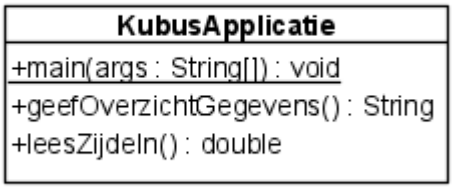
In plaats van spaghetti krijgen we op die manier lasagne, een gerecht dat in laagjes wordt bereid:



- De **presentatielaag** voor klassen die instaan voor de voorstelling van gegevens (in- en uitvoer)
- De **domeinlaag** voor klassen die instaan voor de verwerking van gegevens (logica)
- De **persistentielaag** voor klassen die instaan voor de opslag van gegevens (databank-functionaliteit)

Elke laag wordt voorgesteld door een package in het Java-project. Tot nog toe hebben we altijd met 2 lagen/packages gewerkt aangezien we nog geen gebruik hebben gemaakt van een databank (of andere permanente opslagmogelijkheid). We zouden dus onze code uit de KubusApplicatie kunnen herschikken over een domeinklasse en een applicatieklasse die in de presentatielaag thuishoort.



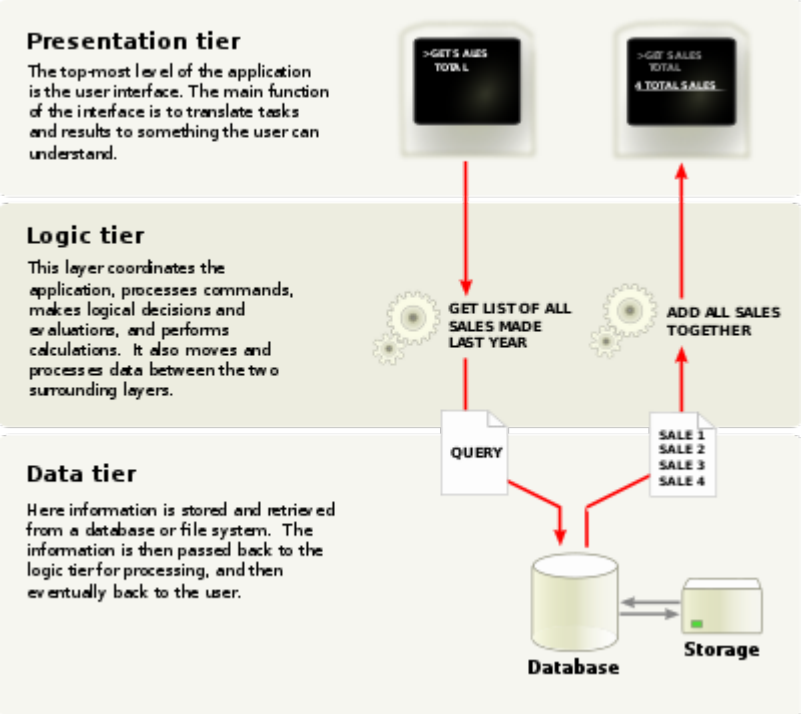


Door deze opdeling in 2 klassen, hebben we het volgende bereikt:

- **GEEN in- en uitvoer** in de domeinklasse
- **GEEN logica** in de applicatie
- Consoleapplicatie kan makkelijk vervangen worden door een GUI of webapplicatie, **zonder dat het domein moet gewijzigd worden**

Als we nu nog willen bijhouden welke dozen er al gemaakt zijn, kunnen we aan deze applicatie ook een database koppelen. In de persistentielaag komt dan een (of meer) mapperklasse(n) die de vertaalslag maken van klassen en objecten naar tabellen en records.

6.3. Wat hoort waar en hoe communiceren de lagen met elkaar?



Aangezien er in grote projecten in elke laag al gauw een tiental klassen kunnen zitten en het de bedoeling is dat het programma de use cases in een bepaalde vastgelegde volgorde afwerkt, willen we ervoor zorgen dat het volledige project maar op één manier kan gestart worden. Er mag dus in heel het project maar één main-methode zijn.

We gebruiken een aparte klasse, vaak **StartUp** genoemd, die deze main-methode bevat. Meestal wordt deze klasse ook in een aparte package (**main**) gezet. De implementatie van deze klasse bestaat uit het maken van een object van de eerste ui-klasse met als parameter een DomeinController-object. Aan de hand van dit object wordt dan een methode uit deze klasse aangeroepen die de verdere controle van het systeem overneemt.

6.3.1. De presentatielaag

De ui kan een consoleapplicatie (cui) of een grafische applicatie (gui) zijn. De constructor van de eerste klasse krijgt een DomeinController-object mee vanuit de StartUp en zal aan de hand van dit object gegevens kunnen opvragen uit de domeinlaag die dan op het scherm kunnen getoond worden. Gegevens die door de gebruiker worden ingevoerd kunnen, eveneens via het DomeinController-object, worden verwerkt in de domeinlaag. Wanneer een volgende ui-klasse wordt aangeroepen, krijgt de constructor hiervan opnieuw de DomeinController mee zodat daarin op dezelfde manier kan verder gewerkt worden.

BELANGRIJK:

- Wanneer gegevens uit de database moeten geraadpleegd worden of er data moet worden weggeschreven, kan dit **NIET** rechtstreeks vanuit de presentatielaag gebeuren. Alle communicatie van de presentatielaag naar de persistentielaag gebeurt steeds via de domeinlaag!
- Er worden in de presentatielaag geen objecten uit de domeinlaag (behalve het DomeinController-object) gebruikt. Enkel primitieve datatypes en objecten van klassen uit de API (Scanner, ArrayList, …) zijn toegestaan.

6.3.2. De domeinlaag

De domeinlaag vormt het hart van de applicatie. Via de DomeinController wordt informatie uit de user interface doorgesluisd naar de betreffende domeinklasse die dan validaties en berekeningen hierop kan doen. De DomeinController heeft daartoe public methodes die overeenkomen met de systeemoperaties uit het SSD.

Indien de applicatie te groot wordt, kan besloten worden om nog verdere UseCaseControllers te gebruiken die ervoor zorgen dat de operaties logisch opgedeeld kunnen worden per use case. De DomeinController "delegeert" dan naar de UseCaseControllers, die dan zelf weer de andere domeinklasses aanspreken.

In de domeinlaag kan er ook communicatie plaatsvinden met de persistentielaag. Dit gebeurt typisch in de repositories.

6.3.3. De persistentielaag

In de persistentielaag wordt de link gelegd met de databank. Maar… in de databank zit data, geen objecten! Het is dus de taak van de persistentielaag om van de data in de database, op aanvraag van het domein, objecten te maken en omgekeerd, om op aanvraag van het domein, objecten op te slaan als data in de database (persistent maken).

Enkele problemen bij het werken met een database zijn dat:

- SQL datatypes verschillen van Java datatypes
- associaties in een DCD navigeerbaar zijn, terwijl je in het relationeel model van een database steeds verbanden in de twee richtingen hebt via de primaire en vreemde sleutel
- in het relationeel model geen n op n verbanden bestaan, terwijl het wel kan dat een associatie aan beide kanten multipliciteit n heeft in een DCD.
- er in het relationeel model nooit gesproken wordt over encapsulatie, overerving en polymorfisme

Dit noemen we de "object relational gap". Om deze kloof te dichten zal er een Object Relational Mapping moeten gebeuren! Daar bestaan ORM-tools voor (Hibernate, JPA, …) waar we later mee zullen leren werken, maar we kunnen er ook voor opteren om zelf de vertaalslag te implementeren.

We doen dit aan de hand van:

- een klasse **Connectie** die alle info bevat over hoe de databank kan geconnecteerd worden
- **Mapperklassen** die verantwoordelijk zijn voor het aanmaken van connecties naar de database met behulp van de gegevens uit de Connectie-klasse en dan:
  - queries of stored procedures te formuleren voor de CRUD-operaties en deze dan door de database te laten uitvoeren:
    - **C(reate)**: opslaan van nieuwe gegevens in de database
    - **R(ead)**: ophalen van gegevens uit de database
    - **U(pdate)**: opslaan van gewijzigde gegevens in de database
    - **D(elete)**: verwijderen van 1 of meerdere records uit database
  - de types uit de applicatie te transformeren naar de types uit de database en vice versa

Maak 1 mapperklasse per tabelhiërarchie (vb klant met verschillende leveringsadressen)