HO
GENT

OOSDII

Overerving Deel 2

Table of Contents

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
- Klasse Object, met zijn methodes toString, equals en getClass
- Het verschil tussen de methode 'equals' en de "==" operator
- De initialisatie van een object binnen zijn klassenhiërarchie
- Uitbreiding en specialisatie van een klasse
- Scope van een declaratie
- Klasse variabelen en klasse methoden
- Keywords 'abstract', 'static' en 'final'
- Visibiliteit

2. Inleiding

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

Overerving kwam reeds aan bod in het vak OOSDI. Binnen dit hoofdstuk gaan we deze kennis opfrissen en verder uitdiepen.

3. Een aantal zaken opfrissen…

Bij overerving is het zeer belangrijk om onderscheid te maken tussen verschillende types enerzijds en hun onderlinge relaties anderzijds.

3.1. JAVA Types

Java kent twee soorten types: primitieve types en referentie types

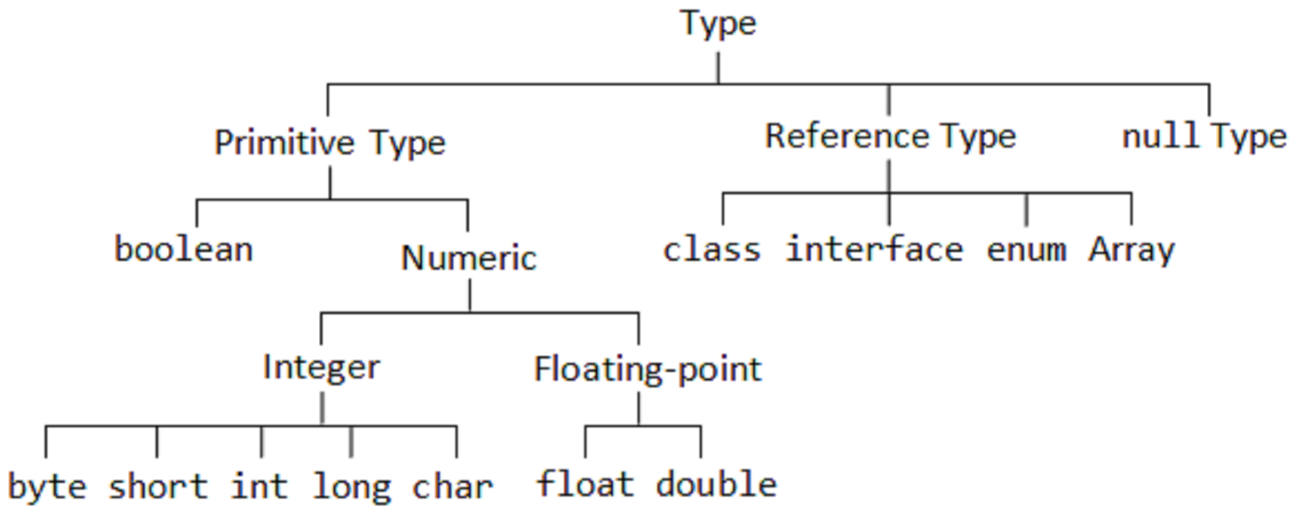
De **primitieve types** zijn:

- het booleantype
- de numerieke types
  - types voor gehele getallen (byte, short, int, long en char)
  - floating point types (float en double)

De **referentietypes** zijn:

- het klassetype
- het interfacetype
- het arraytype

Als referentiewaarde is er ook een speciale 'null' waarde.



Een type limiteert welke waarde aan een variabele kan toegekend worden. Deze waarde kan ook het resultaat van een expressie zijn. Het type van een waarde beperkt ook het aantal operaties dat mogelijk is.

3.2. Waarden en referentie variabelen

Een variabele declareer je altijd als een bepaald type: een primitief data type of een referentie type. Zo wordt bepaald welke 'data' waarden je aan die variabele kan toekennen.

Analoog aan de soorten types bestaan er slechts twee soorten data waarden die je kan toekennen aan een variabele, doorgeven als argument en terug laten geven door een methode.

Deze waarden kan je linken aan het type:

1. primitieve waarden: toegekend aan een variabele met als type een primitief data type
2. en referentiewaarden: toegekend aan een variabele met als type een referentie type

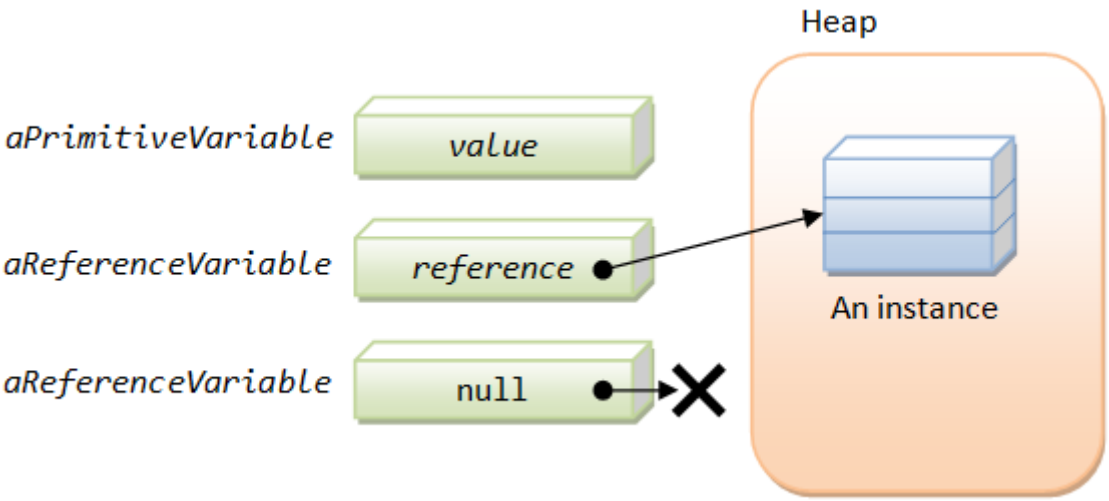
Een variabele met als type een primitief data type omvat de waarde die eraan toegekend werd.

```
1    int a = 5;
```

Een variabele met als type een referentie type omvat een referentie. Deze referentie is ook een bepaalde *waarde*, maar met een speciale betekenis: het is een verwijzing naar een object, een instantie van hetzelfde type (of een subtype hiervan) als dat waarmee de variabele gedeclareerd werd.

```
1    String myString = new String("Hello world!");
```

Bevat een referentie variabele de waarde 'null', dan verwijst de referentie niet naar een concreet object! Probeer je op een referentie met de waarde null een methode aan te roepen, dan krijg je bij het uitvoeren een 'NullPointerException'.



Een object, en dus ook de referentie naar dit object, wordt in Java meestal aangemaakt gebruik makende van het keyword 'new'. Het object wordt geïnstantieerd met behulp van het keyword 'new', de waarde die deze expressie oplevert is de referentie naar het zopas geïnstantieerde object. Een uitzondering hierop is bv. een lambda expressie.

3.3. Wat is een *object*?

Een *object* is een *instantie van een klasse* of een *instantie van een array*.

Referentiewaarden (of *referenties*) zijn:

- pointers (=verwijzingen) naar deze objecten,
- of een speciale 'null' referentie, dewelke aangeeft dat er geen object bestaat om naar te refereren.

> There may be many references to the same object. Most objects have state, stored in the fields of objects that are instances of classes or in the variables that are the components of an array object. If two variables contain references to the same object, the state of the object can be modified using one variable's reference to the object, and then the altered state can be observed through the reference in the other variable.
>
> — The Java Language Specification

3.4. Wat is een klasse?

Klasse declaraties definiëren nieuwe referentie types en beschrijven de implementatie ervan. Ze zijn het *grondplan* op basis waarvan één of meerdere objecten kunnen geïnstantieerd worden.

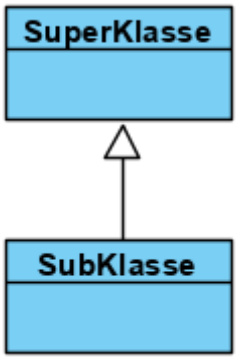
De declaratie van een nieuwe klasse met als identifier 'MijnKlasse' ziet er als volgt uit

```
1 public class MijnKlasse {
2     // Attributen, constructor(s), and
3     // methode declaraties
4 }
```

De identifier 'MijnKlasse' is een nieuw klassereferentietype.

4. Overerving

Overerving betekent dat we bij de declaratie van een klasse eigenschappen en gedrag kunnen laten overnemen van een andere klasse (keyword 'extends'). Dit overgeërfde deel kunnen we uitbreiden of specialiseren. De klasse waarvan eigenschappen en gedrag wordt overgeërfd is de ***superklasse***, de klasse die de eigenschappen en het gedrag overneemt is de ***subklasse***.



Klassen in Java ondersteunen **enkelvoudige overerving**, waarbij elke klasse slechts één enkele superklasse kan hebben.

Elke klasse erft de eigenschappen en het gedrag van zijn superklasse en van andere klassen hogerop in de klassenhiërarchie.

4.1. Doel van overerving

Overerving is het opzetten van een gestructureerde hiërarchie met als doel het halen van een **betere maintenance factor**:

- Meer/eenvoudiger overzicht in structuur en gebruik
- Generalisatie en specialisatie

- Hergebruik van code

(Abstracte) klassen bovenaan de hiërarchie leggen eigenschappen en gedrag vast dat wordt overgeërfd door klassen lager in de hiërarchie.

- Een subklasse erft van zijn superklasse alle eigenschappen en gedrag.
- Een subklasse kan maar één superklasse hebben in Java.
- Een subklasse kan op zijn beurt een superklasse zijn.
- Een superklasse kan meerdere directe subklassen hebben.
- De hoogste klasse in elke klassenhiërarchie in Java is de klasse Object.

4.2. Wat erft de subklasse?

Een klasse erft ***eigenschappen*** en ***gedrag*** van zijn superklasse, en de klassen erboven in de hiërarchie tot uiteindelijk de klasse Object.

Een constructor is geen methode en wordt niet overgeërfd door de subklasse!!!

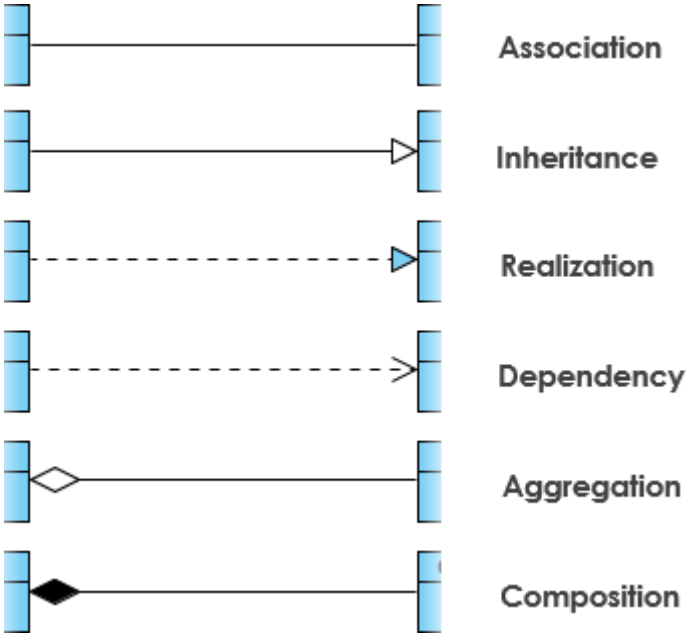
Een constructor heeft een rechtstreeks link met het gedeclareerde klasse referentie type. De naam van de constructor is de naam van de klasse en wordt gebruikt om te differentiëren tussen verschillende referentie types.

> Constructors are similar to methods, but cannot be invoked directly by a method call; they are used to initialize new class instances. Like methods, they may be overloaded.
>
> — The Java Language Specification

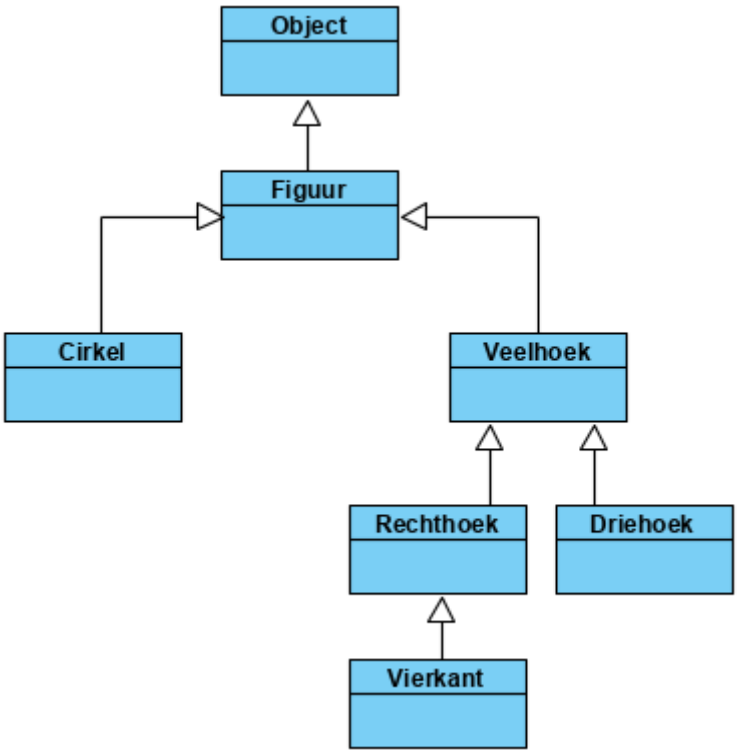
4.3. "IS EEN" relatie

Overerving (= Inheritance in het Engels) resulteert in een **'IS EEN'** relatie

Bij ontwerp, in een UML, wordt deze 'IS EEN' relatie weergegeven met een specifieke pijl, wat helpt om de relatie te visualiseren. Hieronder een overzicht van de mogelijke relaties tussen verschillende types. De "IS EEN" relatie duidt op een overerving ('Inheritance' in het Engels).



4.4. Voorbeeld van een hiërarchie



4.4.1. Bottom to top

- Vierkant *IS EEN* Rechthoek: Let op de richting van de pijl. Een Vierkant is 100% zeker een Rechthoek.
- Rechthoek *IS EEN* Veelhoek → Vierkant *IS EEN* Veelhoek
- Veelhoek *IS EEN* Figuur → Rechthoek *IS EEN* Figuur → …
- Cirkel *IS EEN* Figuur
- Figuur *IS EEN* Object → Veelhoek *IS EEN* Object → …

4.4.2. Top to bottom

- Object *KAN EEN* Figuur zijn
- Figuur *KAN EEN* Cirkel zijn
- Figuur *KAN EEN* Veelhoek zijn: Let op de richting van de pijl. Een Figuur kan een Cirkel zijn, maar ook een Veelhoek…
- Veelhoek *KAN EEN* Rechthoek zijn
- Rechthoek *KAN EEN* Vierkant zijn

Volg je de pijl van de *IS EEN* relatie in de tegengestelde richting, dan krijg je dit verwoorden met *KAN EEN* verband. Dit geeft rechtstreeks aanleiding tot ***polymorfisme***.

Een variabele met als referentie type Figuur kan verschillende vormen aannemen. M.a.w. een referentie variabele van het type Figuur kan refereren naar een object van het type Cirkel maar ook naar een object van het type Veelhoek. Van zowel Cirkel als Veelhoek ben je zeker dat ze alle eigenschappen en gedrag van Figuur overnemen: een Cirkel is een Figuur en ook een Veelhoek is een Figuur.

Polymorfisme laat toe dat variabelen van een klasse type een referentie kunnen bevatten naar een instantie van die klasse of naar een instantie van eender welke subklasse van die klasse.

Polymorfisme laat zo het declareren en gebruik van nieuwe types toe die beschikken over reeds bestaande methodes.

4.5. instanceof

Referentie variabelen kunnen dus verwijzen naar objecten van verschillende types (polymorfisme). In een bepaalde context kan het nodig zijn om te verifiëren of een object van een bepaald referentie type is alvorens het te gebruiken. Dit kan je doen door gebruik te maken van het keyword 'instanceof'.

```
1    Rechthoek rechthoek = new Rechthoek();
2    boolean a = rechthoek instanceof Rechthoek; ①
3    boolean b = rechthoek instanceof Veelhoek; ②
4    boolean c = rechthoek instanceof Vierkant; ③
```

1. a zal 'true' zijn (rechthoek IS EEN Rechthoek)
2. b zal 'true' zijn (rechthoek IS EEN Veelhoek)
3. c zal 'false' zijn (rechthoek KAN EEN Vierkant zijn, maar is daarom nog geen Vierkant. In deze context is rechthoek geen Vierkant)

5. De klasse *Object*

De klasse Object is de *ultieme superklasse* van elke andere klasse.

Object is de klasse aan de top van elke klassenhiërarchie. Als een klasse niet expliciet een subklasse is van een andere klasse, dan wordt ze impliciet een subklasse van de klasse Object.

Klassen ondersteunen **enkelvoudige overerving** in Java, waarbij elke klasse slechts één superklasse kan hebben, behalve de klasse Object, die geen superklasse heeft.

> Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.
>
> — Java API

5.1. Object methoden

Elk klasse- en arraytype erft eigenschappen en gedrag van de klasse Object. Dat gedrag omvat o.a.:

- **equals**: definieert de ***notie van gelijkheid gebaseerd op de toestand van een object***, los van de referentie.
- **toString**: geeft een *String* representatie terug van het object.
- **getClass**: geeft een referentie terug naar het *Class* object, een representatie van de klasse van dat object.

5.1.1. `equals` methode

De methode equals vergelijkt twee objecten en retourneert 'true' als ze 'gelijk' zijn, anders 'false'.

Zowel de `equals` methode als de operator "==" vergelijken 'iets' met elkaar. Er zijn echter belangrijke verschillen tussen beide:

- Het belangrijkste verschil is dat 'equals' een methode is, daar waar "==" een operator is.
- De operator "==" wordt gebruikt om waarden te vergelijken. In geval van referentiewaarden gaat het om adressen in het geheugen. De methode 'equals' gaat vergelijken op basis van inhoud.

De operator "==" vergelijkt of twee waarden, deze links en rechts van de "==" operator, wiskundig gelijk zijn aan elkaar. In geval variabelen gebruikt worden, worden de waarden van de variabelen vergeleken. Gaat het om referentie variabelen, dan verwijzen deze waarden naar een locatie in het geheugen. Verwijzen twee referenties naar dezelfde locatie in het geheugen, dan verwijzen ze naar hetzelfde object.

De operator "==" vergelijkt dus waarden met elkaar in tegenstelling tot de '.equals()' methode die twee objecten vergelijkt op basis van hun toestand.

- Indien een klasse de methode 'equals' niet overschrijft, dan zal de overgeërfde default implementatie van de Object klasse gebruikt worden of de implementatie uit de dichtstbijzijnde super klasse. De default implementatie van de Object klasse valt terug op de operator "==".

```
1 public class EqualsTest {
2     public static void main(String[] args)
3     {
4         String s1 = new String("HELLO");
5         String s2 = new String("HELLO");
6         System.out.println(s1 == s2); ①
7         System.out.println(s1.equals(s2)); ②
8     }
9 }
```

① false

② true

In het voorbeeld worden twee objecten geïnstantieerd: s1 en s2. Beide referenties verwijzen naar verschillende objecten waardoor ze volgens de "==" operator verschillend zijn: twee verschillende objecten kunnen nooit op dezelfde locatie in het geheugen staan.

Via de `equals` methode worden de objecten echter vergeleken op inhoud: de klasse String specialiseert de methode `equals` en vergelijkt twee String objecten op basis van de inhoud van de bijgehouden tekst.

> Compares this string to the specified object. The result is 'true' if and only if the argument is not 'null' and is a ' String' object that represents the same sequence of characters as this object.
>
> — Java API - String class - equals method comment

Nog een voorbeeld:

```java
public class Robot {
    private final String type;
    private final int serieNummer; // Uniek serienummer

    public Robot(String type, int serieNummer) {
        super();
        this.type = type;
        this.serieNummer = serieNummer;
    }

    @Override
    public int hashCode() { ②
        final int prime = 31;
        int result = 1;
        result = prime * result + serieNummer;
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Robot other = (Robot) obj;
        if (serieNummer != other.serieNummer) ①
            return false;
        return true;
    }
}
```

In bovenstaan voorbeeld heeft een Robot een bepaald serieNummer dat uniek is:

① Twee Robot objecten kunnen met elkaar vergeleken worden (@Override van 'equals') en zijn gelijk aan elkaar als hun serieNummer hetzelfde is.

② De methode `equals` gaat altijd samen met de methode `hashCode`. Indien je de ene specialiseert moet je vaak ook de ander specialiseren.

De methodes `equals` en `hashCode` horen samen:

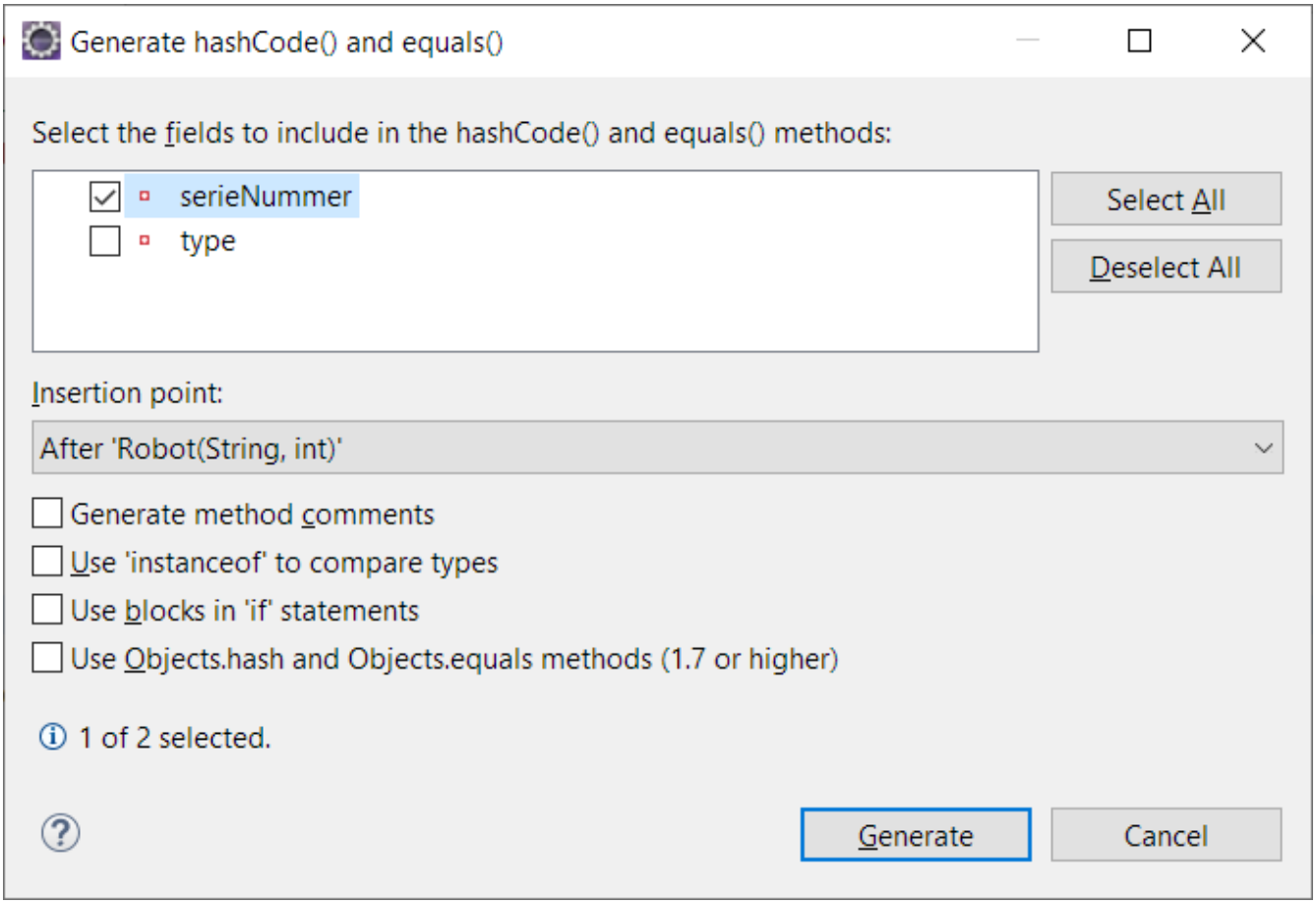
- als twee objecten gelijk zijn aan elkaar (`o1.equals(o2)` is `true`), dan moet de gegenereerde hashcode dezelfde zijn (`o1.hashCode() == o2.hashCode()` moet altijd `true` zijn in dit geval).
- als twee objecten een gelijke hashcode hebben (`o1.hashCode() == o2.hashCode()` is `true`), dan betekent dit **niet** dat de twee objecten gelijk zijn aan elkaar (`o1.equals(o2)` hoeft niet `true` te zijn).

Je mag hashCode en equals zo implementeren dat de hashCode voor twee objecten een gelijke waarde oplevert terwijl de equals methode voor diezelfde twee objecten false retourneert. Retourneert de equals methode true, dan moet de hashcode dezelfde waarde opleveren.

De compiler kan dit gedrag niet afdwingen, het is aan de software ontwikkelaar om dit te garanderen.

> If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.
>
> — Java API

Eclipse biedt de mogelijkheid om de methodes `equals` en `hashCode` samen te genereren:



5.1.2. `toString` methode

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

De `toString` methode geeft een string voorstelling weer , meestal met een in 'mensentaal' begrijpbare beschrijving van het object.

Deze methode wordt ook impliciet aangeroepen wanneer een object moet geconverteerd worden

naar een string.

```
1     String myString = "Hello world.";
2     System.out.println(myString.toString()); ①
3     System.out.println(myString); ②
```

① De uitvoer is gelijk aan <2>

② De uitvoer is gelijk aan <1>

Zowel <1> als <2> roepen de `toString` methode aan van het object myString.

5.1.3. Het Class object

De methode `getClass` waarover elk object beschikt, retourneert een referentie naar een instantie van de klasse Class: de runtime klasse van het object. Deze Class instantie beschikt over alle informatie van het type van het object waarop `getClass` werd aangeroepen.

Een *Class* object bestaat voor elk type. Het kan gebruikt worden om, als voorbeeld, de volledige naam van de klasse, zijn eigenschappen en gedrag, zijn onmiddellijke superklasse en enige interfaces die worden geïmplementeerd op te vragen.

Een referentie naar een *Class* object kan opgevraagd worden met de methode *getClass()*, gedeclareerd in de klasse Object. Deze methode is dus voor elk object beschikbaar. De runtime klasse is altijd gerelateerd aan het object waarop de methode wordt aangeroepen.

```
1 Object obj1 = new Object();
2 System.out.println(obj1.getClass());  // runtime klasse van java.lang.Object
3
4 String obj2 = "";
5 System.out.println(obj2.getClass());  // runtime klasse van java.lang.String
6
7 obj1 = obj2;
8 System.out.println(obj1.getClass());  // runtime klasse van java.lang.String, not
  Object.
```

Via de runtime klasse kan extra informatie opgevraagd worden omtrent het type van het object, bv. de eenvoudige naam van de klasse zelf via 'java.lang.Class.getSimpleName()'.

> Returns the simple name of the underlying class as given in the source code.
>
> — JAVA API - getSimpleName()

Een voorbeeld van het gebruik van de methode 'getSimpleName':

```
1    // Retrieve class simple name
2    Object o = new Person();
3    String simpleNameOfClassObject = o.getClass().getSimpleName(); ①
```

① simplenameOfClassObject zal de string "Person" bevatten

Een referentie variabele gedeclareerd met het type van een superklasse kan ook verwijzen naar objecten van diens subklassen (= polymorfisme). Dit is gekend als *upcasting*. Indien de methode *getSimpleName()* aangeroepen wordt via die superklasse referentie variabele, dan zal de naam van de subklasse teruggegeven worden, aangezien de methode 'getClass' de runtime klasse van het object retourneert!

6. Initialisatie van een hiërarchie

6.1. Constructor

Constructors worden nooit overgeërfd: ze kunnen dus niet overschreven worden.

Constructors worden aangeroepen vanuit expressies die de klasse instantiëren, vanuit conversies en concatenaties t.g.v. de string concatenatie operator en door het expliciet aanroepen van een constructor vanuit een andere constructor.

Constructoren kunnen nooit aangeroepen worden door een methode expressie. Een constructor is geen methode!

6.2. Default constructor

Als een klasse geen declaratie voorziet van een constructor, dan wordt er impliciet een default constructor gedeclareerd.

6.3. Constructor body

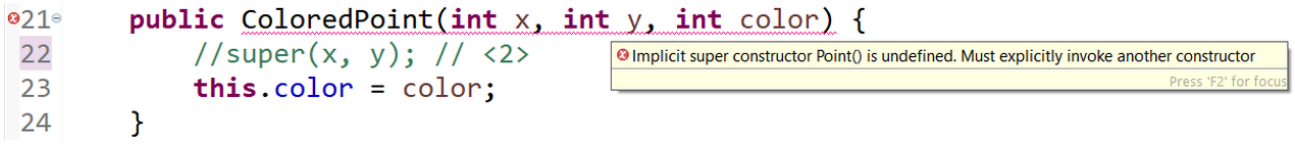
Het eerste statement binnen een constructor body mag een expliciete aanroep zijn naar een constructor van dezelfde klasse, of de directe superklasse.

Als het eerste statement binnen een constructor body geen expliciete aanroep is van een andere constructor (en het niet gaat om de klasse Object) dan zal de constructor body **impliciet** beginnen met het aanroepen van de superklasse constructor **super()**, een aanroep van diens default constructor.

```java
class Point {
    private int x, y;

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

public class ColoredPoint extends Point {
    private static final int WHITE = 0, BLACK = 1;
    private int color;

    public ColoredPoint(int x, int y) {
        this(x, y, WHITE); ①
    }

    public ColoredPoint(int x, int y, int color) {
        super(x, y); ②
        this.color = color;
    }
}
```

① de eerste constructor van ColoredPoint roept de tweede constructor aan met een extra argument

② De tweede constructor van ColoredPoint roept als eerste statement de constructor aan van zijn superklasse Point, en geeft de nodige parameters door. Doe je dit niet, dan zal impliciet de default constructor aangeroepen worden. Als deze niet bestaat, krijg je een compiler error.



6.4. Sequentie van initialisatie: Top to bottom!

In een klassenhiërarchie zal eerste de toestand van de hoogste klasse in de hiërarchie geïnitialiseerd worden (= Object), daarna de 2de hoogste, dan de 3de etc.

Stel: Rechthoek IS EEN Veelhoek IS EEN Figuur IS EEN Object

```java
public class Figuur {
    private String naam;

    public Figuur(String naam) {
        System.out.println("Constructor Figuur");

        this.naam = naam;
    }
}
```

```java
public class Veelhoek extends Figuur {
    public Veelhoek(String naam) {
        super(naam);
        System.out.println("Constructor Veelhoek");
    }
}
```

```java
public class Rechthoek extends Veelhoek {
    public Rechthoek(String naam) {
        super(naam);
        System.out.println("Constructor Rechthoek");
    }
}
```

```java
public class Vierkant extends Rechthoek {
    public Vierkant(String naam) {
        super(naam);

        System.out.println("Constructor Vierkant");
    }
}
```

```java
public class InitialisatieVanObjecten {
    public static void main(String[] args) {
        new Vierkant("Mijn vierkant");
    }
}
```

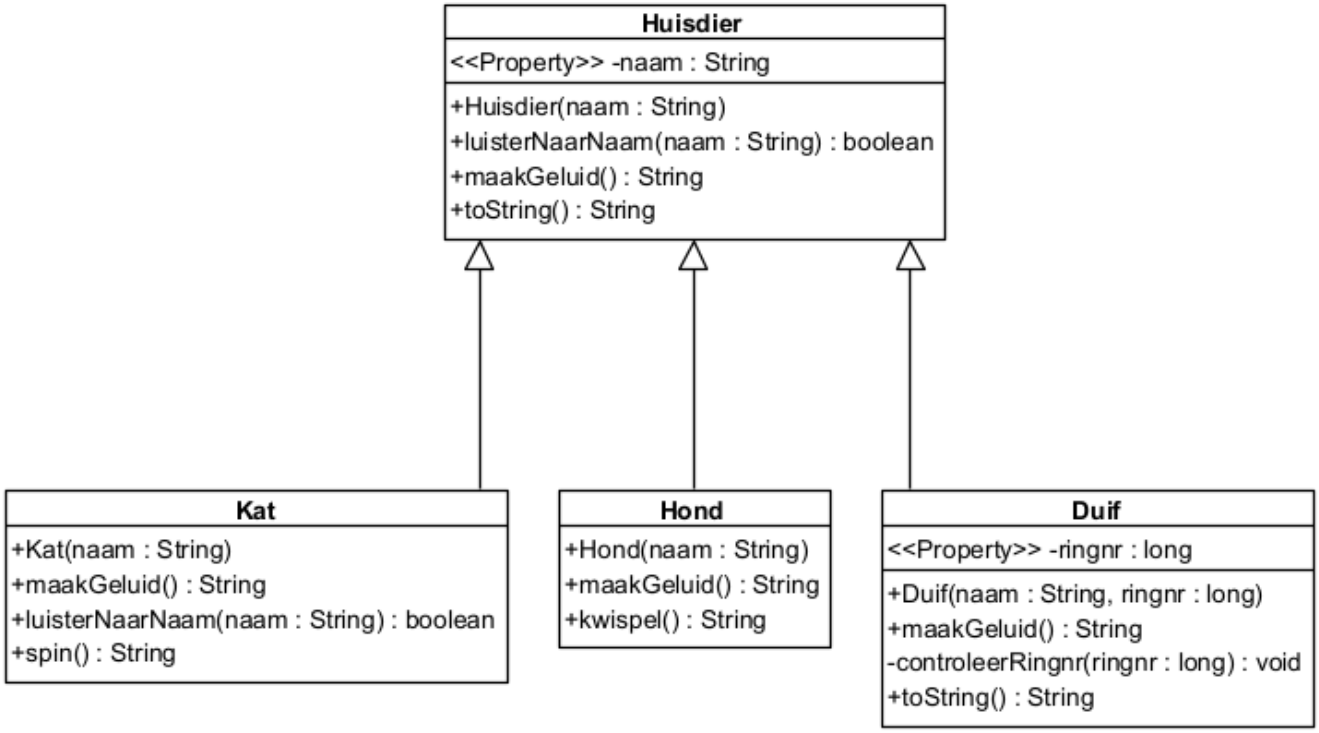
Bekijk de uitvoer van dit voorbeeld!

Vergeet geen noodzakelijke parameters door te geven vanuit de subklasse naar de superklasse, zodat de overgeërfde eigenschappen correct kunnen geïnitialiseerd worden.

Als de superklasse een constructor declareert en niet beschikt over een default constructor, dan moet deze constructor expliciet aangeroepen worden. Anders krijg je een compile error.

7. Uitbreiding van een klasse

Een klasse erft eigenschappen en gedrag van zijn superklasse, kan nieuwe eigenschappen en gedrag toevoegen of bestaand gedrag specialiseren.



In voorgaande hiërarchie breiden de subklassen Kat, Hond en Duif de superklasse Huisdier uit:

- Kat voegt het gedrag `spin` toe
- Hond voegt het gedrag `kwispel` toe
- Duif voegt de eigenschap `ringnr` samen met het bijhorende gedrag

8. Specialisatie van een klasse

In voorgaande hiërarchie merk je ook op dat de subklassen Kat, Hond en Duif elk hun eigen versie implementeren van de methode `maakGeluid`. Dit gedrag werd reeds in de klasse Huisdier gedeclareerd en wordt door de subklassen overschreven. Dit noemen we specialisatie. Via het keyword 'super' kan de subklasse nog steeds het gedrag uit de superklasse aanroepen.

```java
class Radio { // Superklasse
    public void speelMuziek() {
        System.out.println("De radio speelt muziek");
    }
}

class KlassiekeRadio extends Radio { // Subklasse
    @Override
    public void speelMuziek() {
        super.speelMuziek(); ①
        System.out.println("De klassieke radio speelt: Mozart");
    }
}

public class RadioApp {
    public static void main(String args[]) {
        KlassiekeRadio radio = new KlassiekeRadio();
        radio.speelMuziek();
    }
}
```

① De gespecialiseerde methode `speelMuziek` roept het gedrag `speelMuziek` uit de superklasse aan.

9. Scope van een declaratie

De ***scope*** van een declaratie is de 'omgeving' in het programma waarbinnen men kan refereren naar de gedeclareerde entiteit (de variabele, methode of klasse/interface) gebruik makende van zijn 'simple name', voor zover deze naam niet in de schaduw werd geplaatst.

Op een bepaald punt in een programma is een declaratie ***in scope*** als en slechts als dat punt binnen de *scope* van de declaratie ligt. Indien dit niet het geval is is de declaratie ***out of scope***.

9.1. Schaduwen

Sommige declaraties kunnen in de schaduw geplaatst worden binnen een deel van hun scope door een andere declaratie met dezelfde naam. Binnen dit deel kan de *simple name* van de in de schaduw geplaatste declaratie niet meer gebruikt worden om naar die gedeclareerde entiteit te verwijzen.

- De declaratie van een type kan de declaratie van ander type met dezelfde naam in de schaduw plaatsen.
- De declaratie van een attribuut of formele parameter kan de declaratie van een andere variabele met dezelfde naam in de schaduw plaatsen.
- De declaratie van een lokale variabele of exception parameter kan de declaratie van een attribuut met dezelfde naam in de schaduw plaatsen.

- De declaratie van een methode kan de declaratie een andere methode met dezelfde naam in de schaduw plaatsen.

9.1.1. Schaduw variabele

```
1 public class Shadow {
2     private int i = 5; ①
3
4     public static void main(String[] args) {
5         Shadow shadow = new Shadow();
6
7         shadow.foo();
8     }
9
10    public void foo() {
11        int i = 7; ②
12
13        System.out.println("Local i = " + i); ③
14        System.out.println("Class attribute i = " + this.i); ④
15    }
16 }
```

① Declaratie van een attribuut in een klasse

② Declaratie van een lokale variabele die een attribuut met dezelfde naam uit zijn eigen klasse in de schaduw plaatst

③ De lokale variabele is *in scope*. Ze kan gerefereerd worden door de haar *simple name*.

④ Het attribuut is *out of scope*, haal het uit de schaduw gebruik makende van het keyword 'this'.

10. `abstract` keyword

Een klasse of methode kan abstract gemaakt worden door het keyword 'abstract' toe te voegen aan de declaratie.

10.1. `abstract` klasse

Van een abstracte klasse kan geen instantie gecreëerd worden. Probeer je dit toch, dan resulteert dit in een compile-time error.

Een abstracte klasse is een klasse die kan aanzien worden als nog niet volledig afgewerkt of het is expliciet de bedoeling dat deze klasse niet kan geïnstantieerd worden.

Indien een klasse `abstract` is kan ze ook abstracte methodes declareren. Abstracte methodes omvatten enkel een declaratie, zonder implementatie van de methode.

Een klasse beschikt over abstracte methodes indien:

- er expliciet abstracte methodes gedeclareerd worden binnen de klasse
- enige overgeërfde methode als abstract gedeclareerd werd en nog niet geïmplementeerd werd

```
1 abstract class Dot { ①
2     public int x = 1, y = 1;
3
4     public void move(int dx, int dy) {
5         x += dx;
6         y += dy;
7         alert();
8     }
9
10    public abstract void alert(); ②
11 }
12
13 abstract class ColoredDot extends Dot { ③
14    public int color;
15 }
16
17 public class SimpleDot extends Dot { ④
18    @Override
19    public void alert() { }
20 }
```

① De abstracte klasse Dot: deze moet abstract zijn, aangezien ze een methode bevat die abstract is.

② De abstracte methode `alert`. Deze methode heeft een declaratie, maar geen implementatie.

③ De abstracte klasse ColoredDot: deze moet abstract zijn, aangezien ze een geërfde methode bevat die abstract is.

④ De concrete klasse SimpleDot: de geërfde methode `alert` krijgt hier pas een implementatie.

Een niet abstracte klasse kan je benoemen als een ***concrete*** klasse.

10.2. `abstract` methode

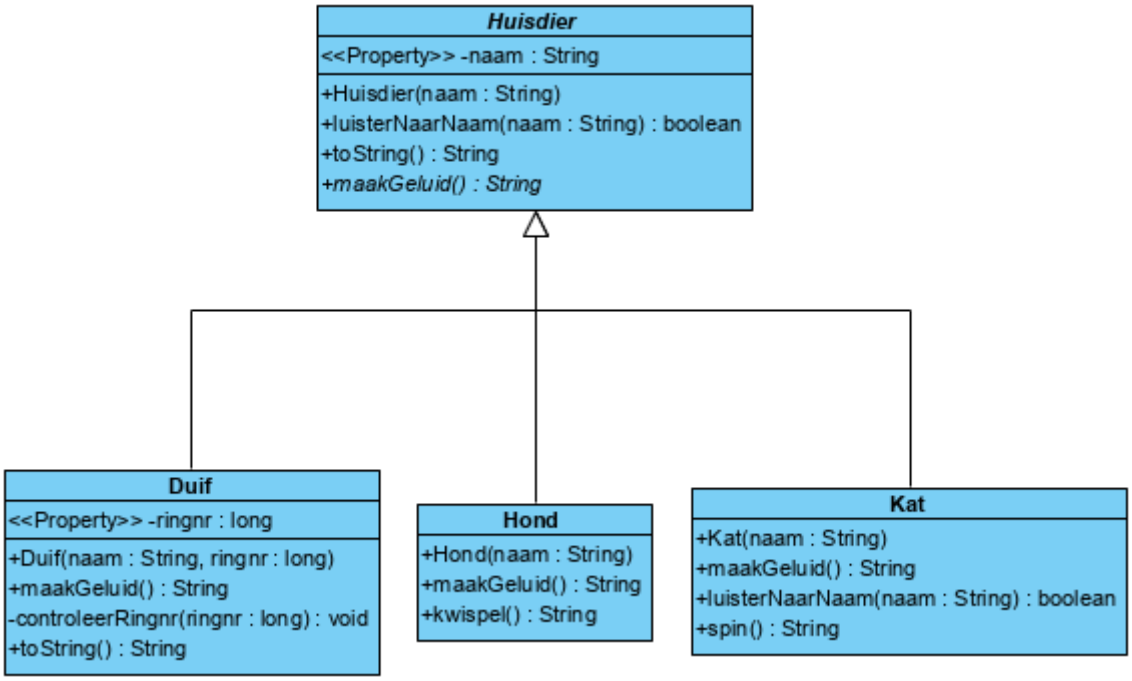
Een declaratie van een abstracte methode (keyword `abstract`) introduceert een methode als gedrag, waaronder zijn handtekening, return value en throws clausule indien gewenst, maar zonder de implementatie van de methode te voorzien.

Een niet abstracte methode kan je benoemen als een concrete methode.

10.3. `abstract` voorbeeld

Binnen de Huisdier hiërarchie is het zinloos om een Huisdier te instantiëren: wat is precies een Huisdier en welk geluid maakt het? In deze context is het aangewezen om te verhinderen dat van

Huisdier een instantie kan gemaakt worden door deze klasse abstract te maken. Ook de methode `maakGeluid` kan op dit niveau in de hiërarchie weinig zinvol geïmplementeerd worden. Ook deze kan aangeduid worden als abstract.



In een UML wordt een abstracte klasse aangeduid door zijn naam *italic* te schrijven. Ook de naam van een abstracte methode wordt *italic* geschreven.

```java
public abstract class Huisdier //extends Object
{
    private String naam;

    public Huisdier(String naam)
    {
        setNaam(naam);
    }
    public String getNaam()
    {
        return naam;
    }
    public final void setNaam(String naam)
    {
        this.naam = naam;
    }
    public boolean luisterNaarNaam(String naam)
    {
        return (naam.equals(this.naam));
    }
    public String toString()
    {
        return String.format("%s met naam %s", this.getClass().getSimpleName(),
naam);
    }

    public abstract String maakGeluid();
}
```

11. `static` keyword

11.1. static attribuut

Als een attribuut `static` wordt gedeclareerd bestaat er slechts één instantie van dat attribuut, ongeacht het aantal instanties (mogelijks geen enkele) dat van die klasse gemaakt worden.

Een `static` attribuut, ook benoemt als klassevariabele, wordt aangemaakt als de klasse zelf voor de eerste keer wordt geïnitialiseerd (= op het moment dat de klasse wordt ingeladen in de JVM)

Een klassevariabele wordt dus gedeeld door alle instanties van die klasse.

Een attribuut gedeclareerd zonder het keyword `static`, ook wel non-static attribuut genoemd, noemt men een **instantievariabele**. Telkens een nieuwe instantie van de klasse wordt gecreëerd, zal een nieuwe variabele ontstaan geassocieerd met die klasse-instantie en dit voor elke instantievariabele gedeclareerd in die klasse of in elk van zijn superklassen.

11.2. `static` methode

Een `static` methode, ook benoemd als klassemethode, kan aangeroepen worden zonder een referentie naar een instantie van die klasse. Binnen een `static` methode resulteert het gebruik van de keywords `this` of `super` in een compile time error.

Een methode niet gedeclareerd als `static` noemt men een instantiemethode of een non-static methode. Een instantiemethode wordt altijd aangeroepen in relatie met een object, dat het huidige object wordt naar waar het keyword `this` verwijst tijdens de uitvoer van die methode.

12. `final` keyword

12.1. `final` klasse

Een klasse kan gedeclareerd worden als `final` om te voorkomen dat ze gebruikt wordt als superklasse. Een `final` klasse kan dus geen subklassen hebben.

12.2. `final` attribuut

Een attribuut kan als `final` gedeclareerd worden. Zowel klasse- als instantie variabelen (static en non-static) kunnen als `final` gedeclareerd worden.

Een `final` klasse variabele moet een waarde toegekend krijgen binnen een static initializer in die klasse, bij de declaratie van het attribuut of binnen een constructor.

```
1 public class Container {
2     private static final int MAXIMUM_GEWICHT = 100; ①
3 }
```

① Initialisatie van een `final` klasse variabele tijdens declaratie.

De afspraak in Java is dat static final variabelen (de echte constanten) in hoofdletters geschreven worden.

```
1 public class Container {
2     private static final int MAXIMUM_GEWICHT;
3
4     static {
5         MAXIMUM_GEWICHT = 5; ①
6     }
7 }
```

① Initializatie van een `final` klasse variabele binnen een static initializer van de klasse.

Een `final` instantie variabele die niet wordt geïnitialiseerd bij declaratie of binnen een static initializer moet binnen een constructor een waarde toegekend krijgen.

```
1 public class Product {
2     public final int barcode = 5; ①
3     public final int price;
4
5     public Product(int price) {
6         this.price = price; ②
7     }
8 }
```

① Initialisatie van een `final` instantie variabele bij declaratie.

② Initialisatie van een `final` instantie variabele binnen een constructor.

Ofwel wordt een `final` instantie variabele bij declaratie ofwel binnen een static initializer ofwel binnen een constructor een waarde toegekend. Slechts op één plaats.

Voor een final attribuut kan je geen set-methode maken!

12.3. `final` variabele

Een variabele kan als `final` gedeclareerd worden. Zo een variabele kan slechts éénmalig een waarde toegekend worden.

Als een final variabele of final attribuut een referentie bevat naar een object, dan kan de toestand van dat object nog steeds wijzigen. De variabele zelf zal echter altijd naar hetzelfde object verwijzen. Hetzelfde geld voor een referentie naar een array.

12.4. `final` methode

Een methode kan als `final` gedeclareerd worden om te voorkomen dat een subklasse deze overschrijft.

Een `private` methode en alle methodes binnen een `final` klasse gedragen zich alsof ze als `final` gedeclareerd zouden zijn, sinds het onmogelijk is hen te overschrijven.

12.4.1. Waarschuwing: niet geïnitialiseerde instantie variabelen

Roep vanuit een constructor enkel methodes aan die niet overschreven kunnen worden: `private` methodes of `final` methodes. Indien dit niet het geval is kunnen niet geïnitialiseerde instantie variabelen gebruikt worden, wat gevaarlijk is!

13. Toegangscontrole (Visibiliteit)

Onderstaande tabel geeft de mogelijkheden weer qua toegangscontrole en de daaruit volgende visibiliteit.

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

De `private` attributen en methoden van de superklasse zijn niet toegankelijk vanuit een subklasse, hoewel ze wel worden geërfd.

Attributen en methoden in de superklasse met de toegangsclausule `protected` zijn toegankelijk vanuit een subklasse alsook vanuit iedere klasse die in hetzelfde package is gedefinieerd.

13.1. Principle of least privilege

Dit principe geeft o.a. aan dat een onderdeel slechts die visibiliteit mag krijgen nodig om zijn taak te volbrengen. Als een onderdeel een bepaalde visibiliteit niet nodig heeft, zou ze deze niet mogen hebben.

Een ander voorbeeld is het `final` keyword: denk je dat een variabele slechts éénmalig dient ingesteld te worden, maak het dan `final`.