Ian Angillis

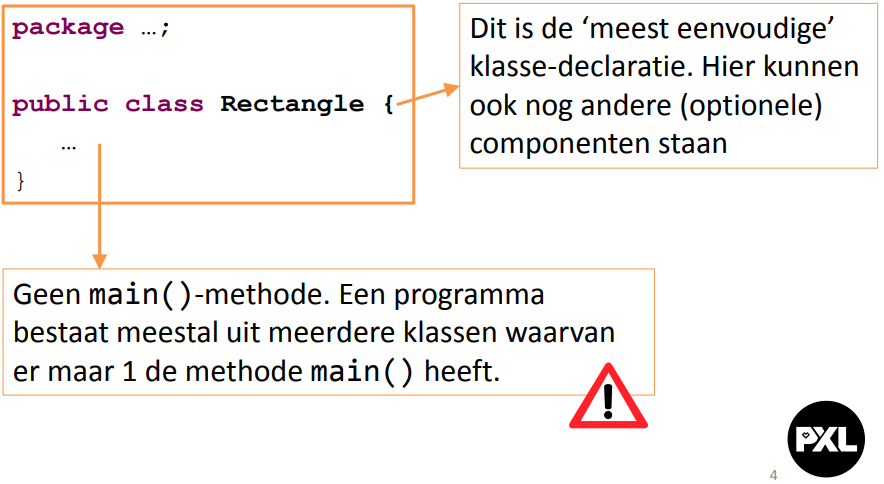
1ste bachelor Toegepaste Informatica @ PXL

Samenvatting Java Essentials

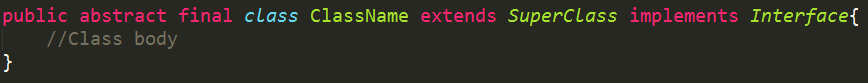
**Hoofdstuk 8: Klassen Definiëren**

**8.1 Declaratie van een klasse**

We kunnen zelf klassen maken.



Een algemene klassendeclaratie met de belangrijkste componenten ziet er zo uit:



Verklaring van verschillende componenten

|  |  |
| --- | --- |
| **Component** | **Beschrijving** |
| *public* | * Geeft aan dat deze klasse voor publiek gebruik is. * De klasse kan gebruikt worden door andere klassen. * Indien een klasse niet expliciet als public gedeclareerd wordt, kan ze enkel gebruikt worden door klassen binnen dezelfde package |
| *abstract* | * Geeft aan dat het gaat om een abstracte klasse. * Kunnen geen concrete objecten van gemaakt worden. * Worden enkel gebruikt om andere klassen van af te leiden |
| *final* | * Geeft aan dat van deze klasse geen andere subklassen van kunnen worden afgeleid. * het is als het ware een finale klasse * *abstract* en *final* kunnen **NIET** samen voorkomen |
| *class ClassName* | * Declaratie van een klasse met de naam ClassName * *class* component is verplicht voor iedere declaratie van een klasse |
| *extends SuperClass* | * De klasse die hier gedeclareerd wordt is afgeleid van de superklasse met de naam SuperClass |
| *Implements Interface* | * *implements* geeft aan dat de klasse de volgende interfaces implementeert. * Meerdere interfaces worden met een komma van elkaar gescheiden |

Indien de optionele componenten niet expliciet worden weergegeven, geldt dat de klasse:

* Niet publiek is
* Niet abstract is
* Niet final is
* Een subklasse is van de klasse *Object*
* Geen interfaces implementeert

**8.2 De klassenomschrijving (body)**

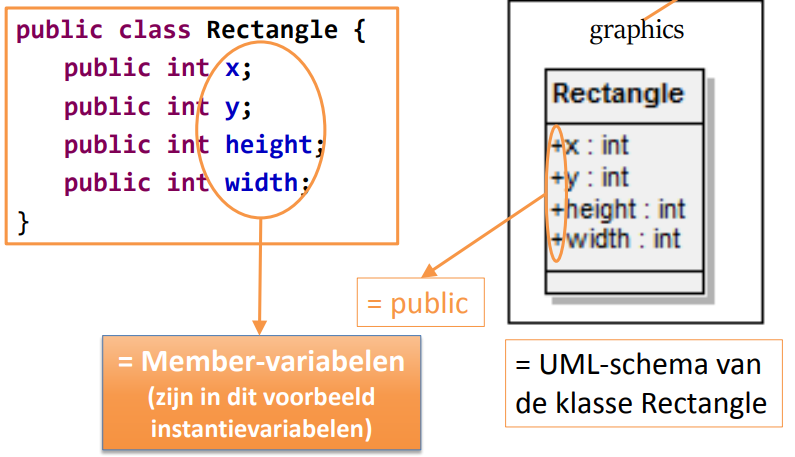
De omschrijving van de body van de klasse bevindt zich tussen de accolades volgend op de declaratie

De body bevat:

* Eigenschappen (member-variabelen)
  + Instantievariabelen
  + Klassevariabelen
* Methodes (member-methods)
  + Instantiemethodes
  + Klassemethodes
* Constructors

**8.2.1 Eigenschappen**

Bij de omschrijving van de klasse worden meestal eerst de variabelen of eigenschappen van de klasse gedefinieerd.



De declaratie van variabelen ziet er in het algemeen als volgt uit:



Public, private en protected zijn acces modifiers. Ze bepalen het toegangsniveau van de eigenschap (variabele) van de klasse.

Toegangsniveau:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Toegangsniveau** | **Eigen klasse** | **pakketklassen** | **subklassen** | **alle klassen** |
| *private* | X |  |  |  |
| * *(package/friendly)* | X | X |  |  |
| *protected* | X | X | X |  |
| *public* | X | X | X | X |

Het toegangsniveau *package* is geldig als geen van de drie andere toegangsniveaus gespecificeerd wordt.

|  |  |
| --- | --- |
| **Component** | **Omschrijving** |
| *static* | De variabele is een klassenvariabele en geen instance-variable |
| *final* | De variabele is een constante |
| *type* | Type van de variabele. Kan zowel een primitief datatype als een referentietype zijn. |
| *name* | De naam van de variabele |
| *initValue* | De initiële waarde van de variabele |

Indien de variabelen niet expliciet geïnitialiseerd worden, krijgen ze automatisch de waarde 0, null of false naargelang het datatype.

De eigenschappen kunnen worden gebruikt door de naam van het object, gevolgd door een punt en de naam van de eigenschap: *objectname.propertyname*

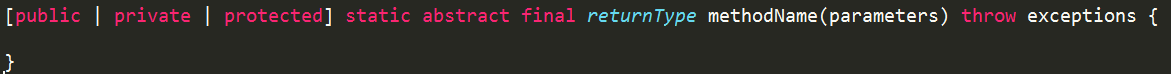
**8.2.2 Methoden**

Methoden zijn codeblokken verbonden aan het object. Door middel van methoden kunnen andere objecten boodschappen sturen naar dit object en eventueel een resultaat terugkrijgen.

Methoden hebben zoals klassen een methode-declaratie en een methode-body.

**8.2.2.1 Declaratie van methoden**

Algemene declaratie van een methode:



Enkel returnType en methodName(Parameters) zijn verplicht, de overige componenten zijn optioneel.

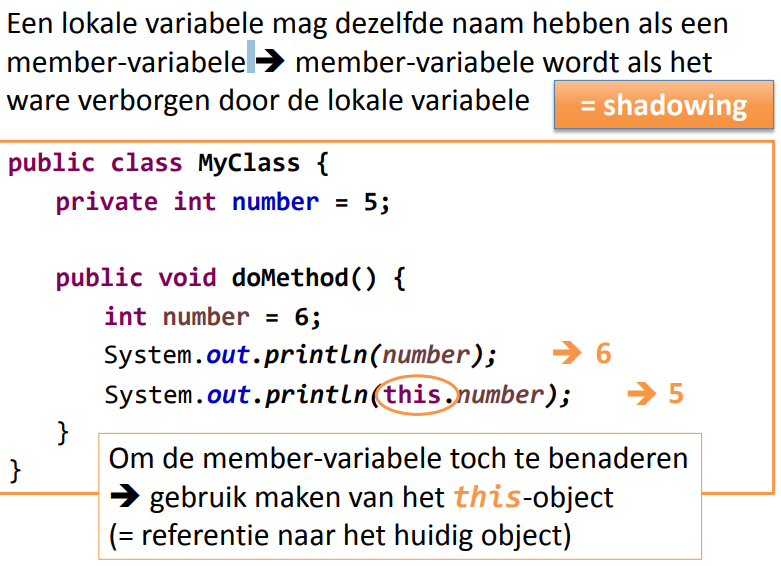
Indien het toegansniveau niet gespecificeerd wordt, geldt het package - toegangsniveau. Voor de rest geldt het toegangsniveau zoals uitgelegd bij de eigenschappen ook op methodes.

Overige componenten:

|  |  |
| --- | --- |
| **Component** | **Beschrijving** |
| *static* | Geeft aan dat de methode een klassenmethode is in plaats van een instance-methode |
| *abstract* | * Heeft nog geen implementatie * Moeten worden vervangen (overriden) door de methode van subklassen * Abstracte methode kan alleen gedefinieerd worden in een abstracte klasse |
| *final* | De methode kan niet vervangen (overriden) worden door de subklasse(s) |
| ***returnType*** | Het datatype van de data die door de methode wordt teruggegeven |
| ***methodeName*** | Naam van de methode |
| ***(parameters)*** | De parameters die aan de methode worden meegegeven |
| throws exceptions | Bepaalt de exceptions die kunnen optreden tijdens de uitvoering van deze methode. |

**8.2.2.2 De body van de methode**





**8.2.2.3 Gegevens doorgeven aan een methode**

Zie samenvatting programming basics

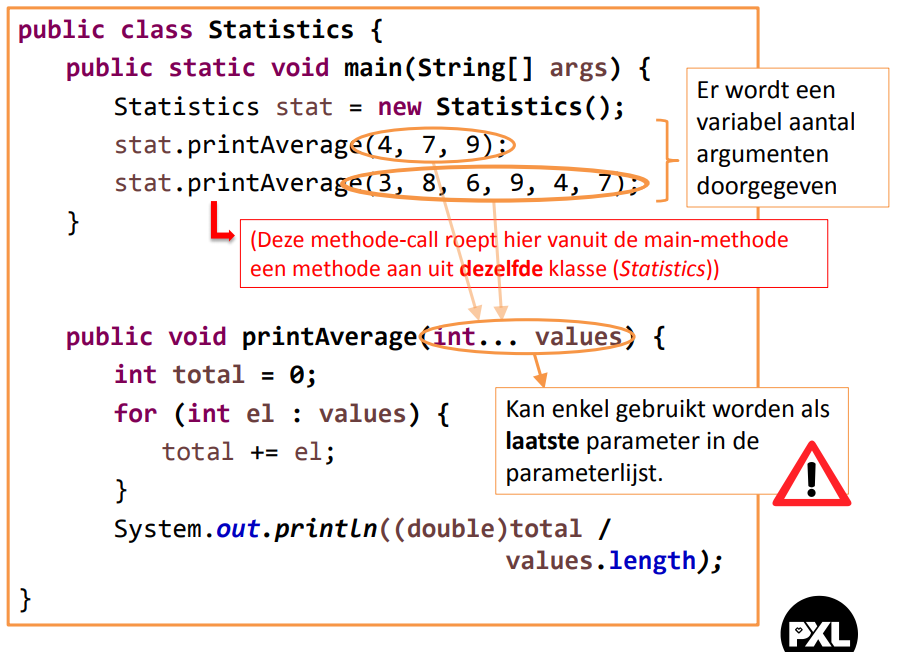
Men zou de parameters kunnen beschouwen als lokale variabelen die in de methode gedeclareerd worden en onmiddellijk geïnitialiseerd worden tijdens het aanroepen van de methode.

Rekening houden met call by value en call by reference. Klassen zijn objecten, en een object van eender welk type wordt dus altijd doorgegeven als een call-by-reference.

**8.2.2.4 Waarden teruggeven via een methode**

Zie samenvatting Programming Basics

Methodes met een variabel aantal parameters:



Private of publieke membervariabelen?

* Het is een slechte gewoonte om membervariabelen public te maken. Iedereen buiten de klasse kan dan de waarde van die variabelen aanpassen naar eender welke waarde wat schadelijk kan zijn voor de werking van de klasse
* Private => we schermen de variabelen af van de buitenwereld
  + Dit noemen we ook encapsulation (inkapseling; data hiding)
* We maken de variabelen toegankelijk via aangepaste methodes (getters en setters)
* Via de public methode kan je dan controle uitoefenen op de waarde van de private membervariabelen
  + Zo kunnen we bijvoorbeeld kizen om negatieve getallen te vervangen door hun absolute waarde of nul.

**8.2.2.5 Method name overloading**

Zie samenvatting Programming basics

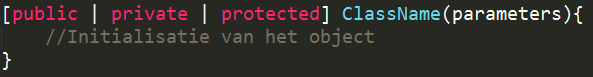
Methodes met dezelfde naam maar met een verschillende parameterlijst (signatuur). Op basis van de parameters die worden meegegeven kiest de compiler de juiste methode.

**8.2.3 Constructors**

Iedere klasse heeft een constructor:

* Stuk code dat wordt uitgevoerd bij het creëren van het object met de new-operator
* Wordt gebruikt om het object te initialiseren
* Lijkt op een gewone methode maar heeft geen return-type
* Naam komt overeen met de klassenaam
* Methode kan maar 1x aangeroepen worden, tijdens de constructie van het object

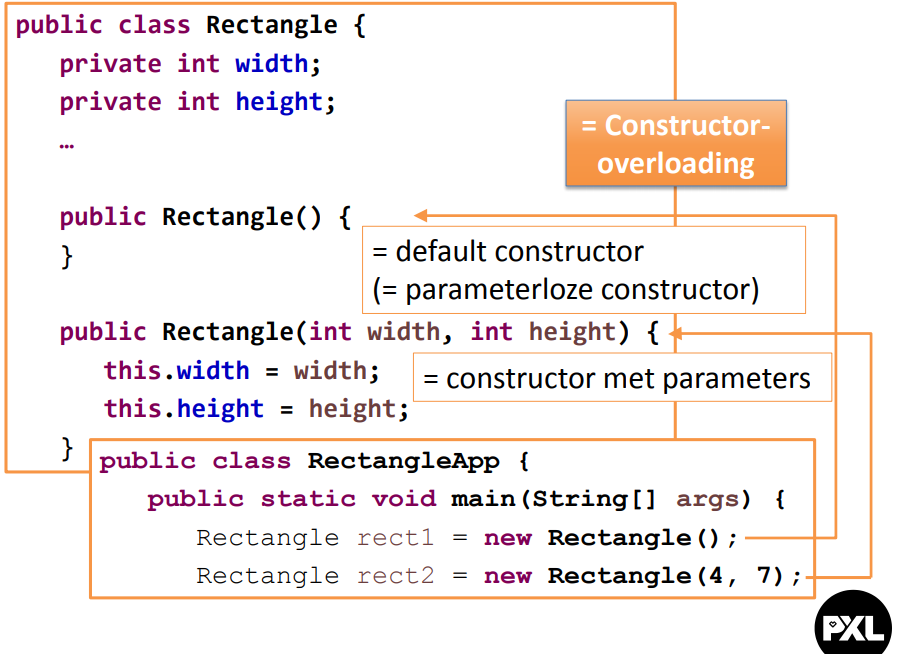
Definitie van een constructor:



Het toegangsniveau van de constructor gaat analoog zoals beschreven. Indien geen expliciet toegangsniveau wordt gespecificeerd, geldt het package niveau.

|  |  |
| --- | --- |
| **Component** | **Beschrijving** |
| *Public* | Iedere andere klasse kan een object maken van deze klasse door middel van deze constructor |
| *Private* | * Geen enkele andere klasse kan een object maken van deze klasse door middel van deze constructor. * De klasse zelf kan deze constructor wel gebruiken |
| *Protected* | Enkel subklassen en paketklassen kunnen deze constructor gebruiken |
| * *(package)* | Enkel klassen binnen hetzelfde pakket (package) kunnen deze constructor gebruiken |

Indien de constructor niet expliciet gedefinieerd wordt, krijgt de klasse een standaardconstructor zonder parameters. Zodra een constructor gespecificeerd wordt, vervalt deze standaardconstructor en moet die dan eventueel expliciet gedeclareerd worden.

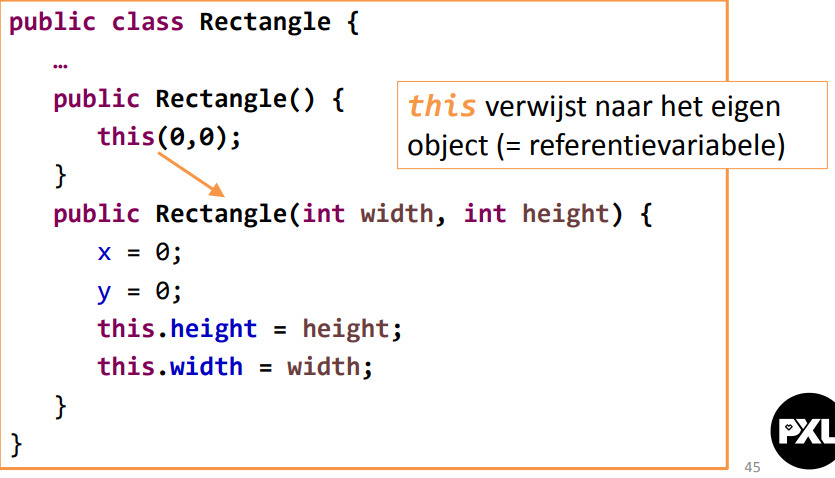


Een constructor kan een andere constructor met de nodige parameters aanroepen door *this*

*This* verwijst naar het eigen object (=referentievariabele)

Voordelen:

* De eigenlijke code voor het initialiseren van het object moet maar op 1 plaats geschreven worden
* Code wordt compacter en overzichtelijker
* Risico op fouten daalt: aanpassingen moeten immers gebeuren in 1 constructor i.p.v. in elke constructor
* Indien een constructor een andere constructor aanroept met dit steeds gebeuren op de eerste regel



**8.2.4 Instance members en class members**

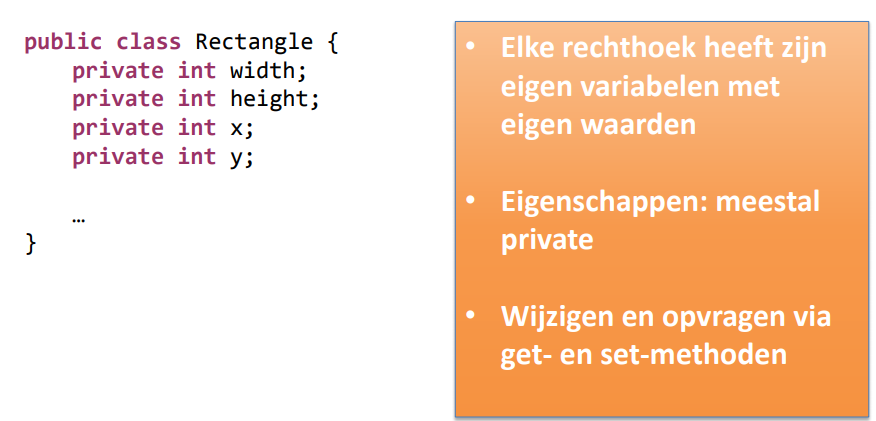
Er is een onderscheid tussen instance-variabelen en klassenvariabelen. Evenzo is er een onderscheiden tussen isntance-methods en klassenmethods.

Instance members = instance-variabelen en instance-methoden

Class members = klassenvariabelen en klassenmethoden

**8.2.4.1 instance variabelen**

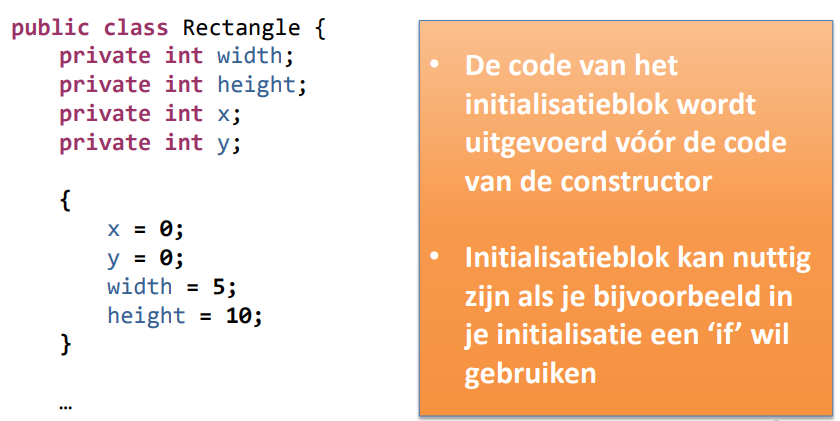
Instance – variabelen = ieder object van een bepaalde klasse heeft een eigen kopie van de variabelen met elk hun eigen waarde





Initialisatie van instance-variabelen kan gebeuren op 3 manieren:

* Tijdens de declaratie (vooral voor primitieve datatypen)
* In een initialisatieblok
  + Laat ons toe code voor de constructor uit te voeren, maar pas nadat de impliciete of expliciete aanroep van super() is gebeurd
* In de constructor (waarden komen binnen als parameters)



**8.2.4.2 Klassenvariabelen**

Klassenvariabelen:

* Zijn gemeeschappelijk voor alle objecten van dezelfde klasse
* Van elke klasse-variabele is er slechts 1, die door alle objecten gedeeld wordt
* Worden gedefinieerd met het woord static

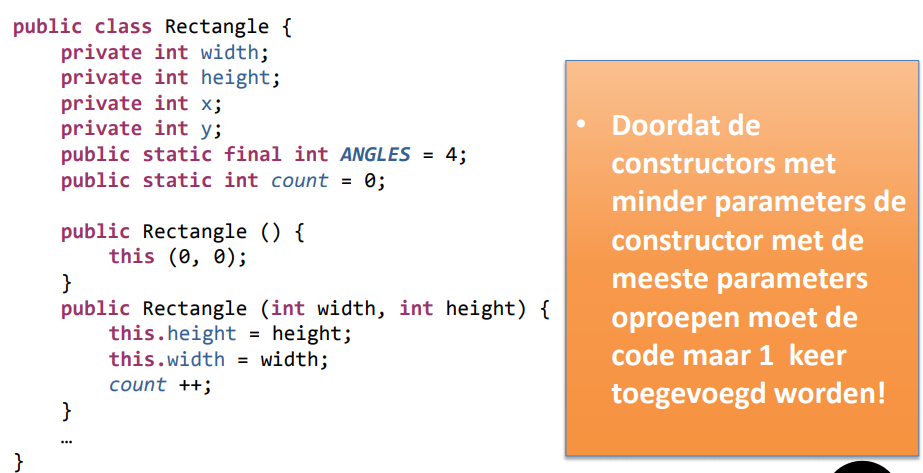
Klassenvariabelen kunnen gebruikt worden door: ClassName.Variabele

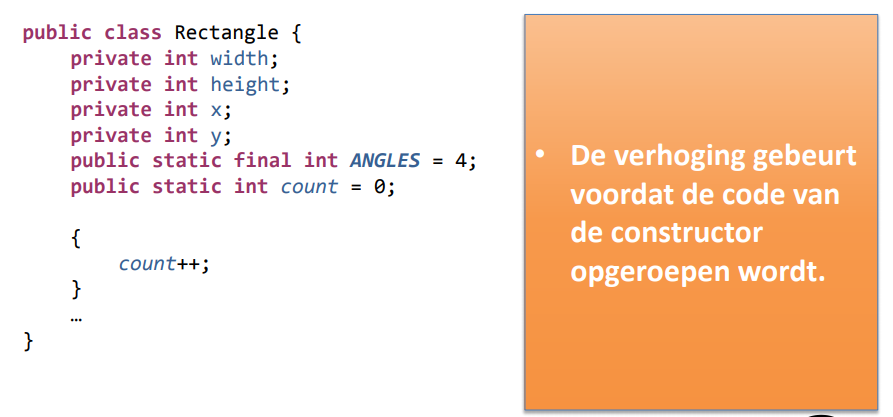
Voorbeeld:





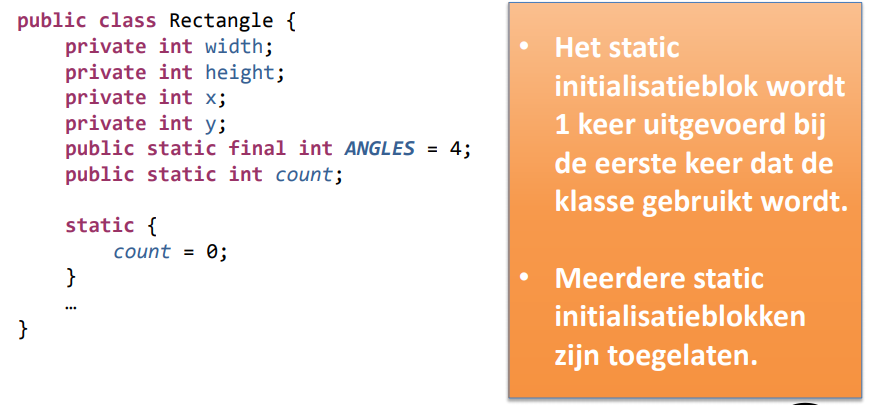
Een ander voorbeeld is met een teller die bijhoudt hoeveel keer een object is aangemaakt van die klasse. Zorg wel dat dit in uw uiteindelijke code private is.

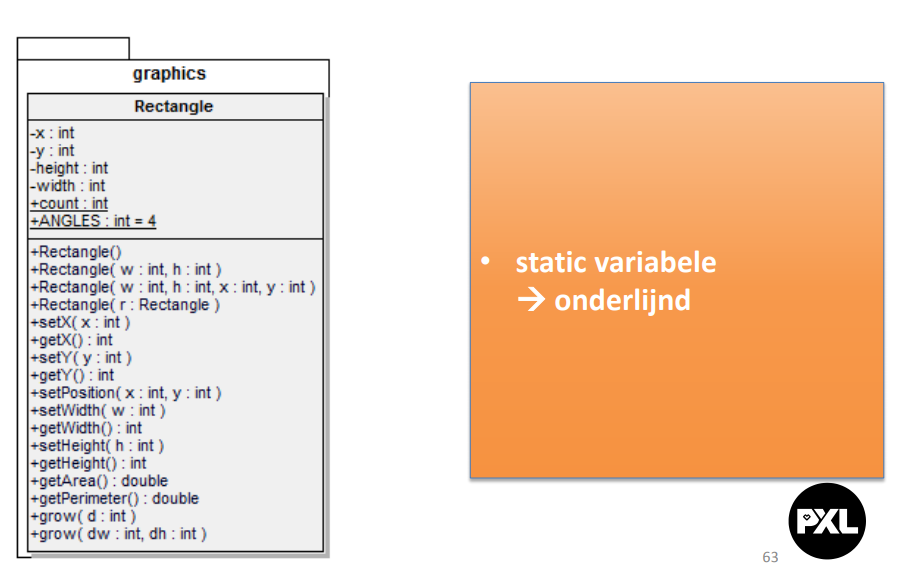




De initialisatie van klasse-variabelen kan op 2 manieren:

1. Tijdens de declaratie (vooral voor primitieve datatypen)
   1. Vb. public static final int ANGLES = 4
2. In een static initialisatieblok voor meer complexe initialisaties van meer dan 1 statement
   1. Doet dienst als een soort klassenconstructor
   2. Het wordt eenmalig uitgevoerd bij het eerste gebruik van van de klasse
   3. 1 klasse kan meerdere static initialisatieblokken hebben





**8.2.4.3 Instance – methoden**

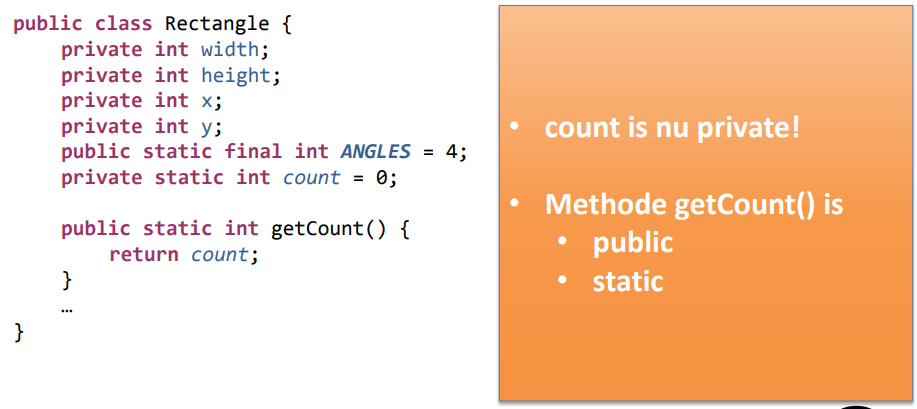
* Moeten opgeroepen worden op een concreet object van de klasse
* De methode kan gebruik maken van de instance – variabelen van dat specifieke object
* De methode kan gebruik maken van de klassenvariabelen
* De methode kan gebruik maken van lokale variabelen

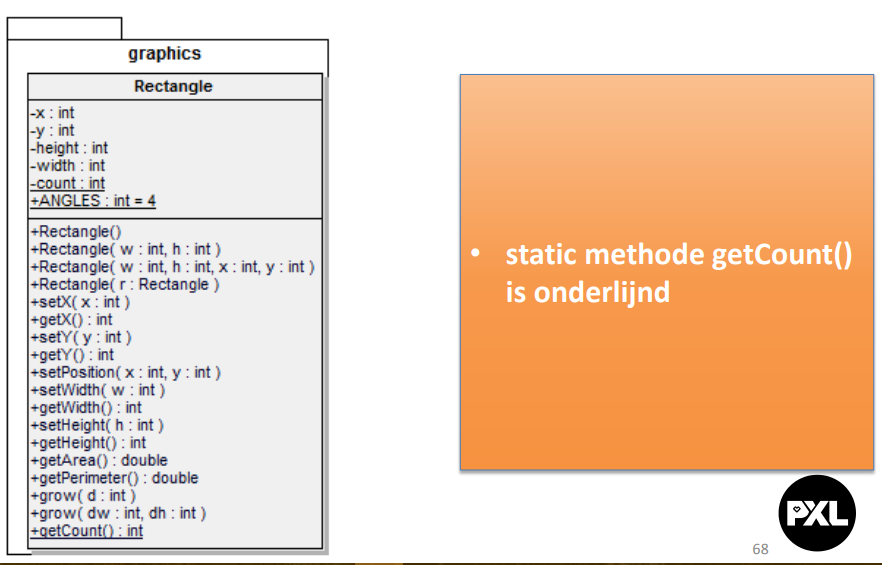
**8.2.4.4 Klassenmethodes**

Klasse – methoden = een methode die wordt opgeroepen op basis van een klasse-naam

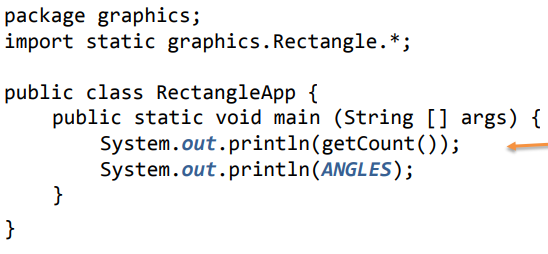
ClassName.methodName();

* Methode kan enkel gebruik maken van klassevariabelen (en lokale variabelen binnen de methode)
* Binnen een klassenmethode is de this-referentie onbestaand





Enkele opmerkingen:

* De main-methode is een static method
  + We kunnen enkel klassenvariabelen gebruiken
  + We kunnen wel lokale variabelen creëren en gebruiken
* Je kan static members importeren
  + Geen vermelding van de klassenaam meer nodig
  + 
* Binnen een klassenmethode is de this-referentie onbestaand

**8.2.5 De klasse Math**

Sommige klassen hebben hebben enkel static members. Dit noemen we utility-klassen.

Utility-klasse:

* Een klasse die hulpmethoden aanreikt
* Bevatten enkel static members
* We kunnen er geen objecten van maken

De klasse Math bevat:

* 2 klassenvariabelen
  + Math.PI Het getal π
  + Math.E Het getal e
* Een groot aantal methoden voor wiskundige berekeningen
  + Zie javadoc

**Hoofdstuk 9: Associaties**

**9.1 Inleiding**

Objecten worden geschreven in klassen

* Variabelen en methoden

Wat is het voordeel van OO-programmeren?

* Hergebruik van bestaande code

Als een klasse goed gedefinieerd is kan ze in een andere context hergebruikt worden. Dit doen we door relaties te leggen tussen klassen (en uiteindelijk tussen objecten).

3 soorten relaties:

1. Associaties : ene object maakt gebruik van het andere
2. Aggregaties: ene object is eigenaar van het andere, maar de objecten kunnen afzonderlijk bestaan
3. Composities: objecten zijn onlosmakelijk met elkaar verbonden.

**9.2 Associaties**

Associatie = algemene term voor een relatie tussen klassen of objecten; de ene klasse gebruikt de functionaliteit van de andere.

Het komt erop neer dat het ene object gebruik maakt van de mogelijkheden van een ander object.

Een associatie is een ‘HAS A’ relatie.

Voorbeelden:

A Person **HAS AN** Adres

A Persoon **HAS A** Birthdate

A Book **HAS An** Author

Navigeerbaarheid = de mogelijkheid van vanuit de ene klasse de andere aan te spreken

Het lijkt wel alsof we van de ene klasse kunnen navigeren naar de andere. Deze navigeerbaarheid is meestal in 1 richting (uni-directioneel). In de ULM wordt did aangeduid door een pijltje.

Voorbeeld: Klasse Random





**9.3 Aggregaties**

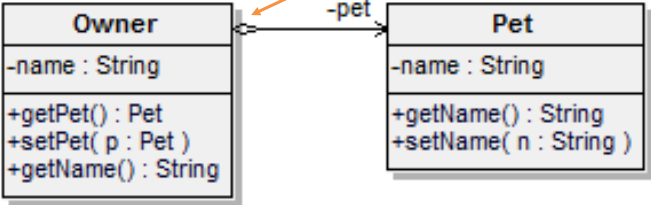
Een aggregatie:

* Is een bijzondere vorm van associatie
* Het ene object is eigenaar van het andere
* Relatie tussen 2 klassen in nauwer

Beiden klassen zijn niet onlosmakelijk aan elkaar gekoppeld en kunnen beiden op hun eigen bestaan. Vanuit het voorbeeld dat wordt gegeven betekent dit:

* Eigenaar kan ander huisdier nemen
* Huisdier kan van eigenaar veranderen. Daarom is in de klasse Owner een methode setPet()

In het UML-diagram wordt een aggregatie voorgesteld met een open ruit.



Voorbeeld:





**9.4 Composities**

Compositie-relatie

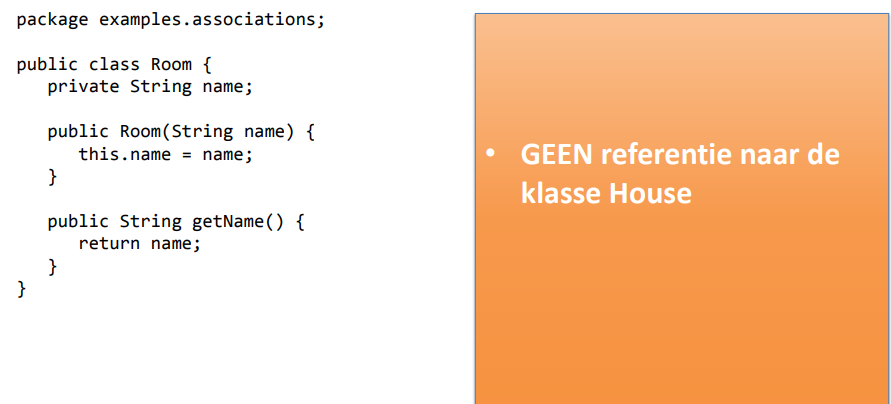
* 2 objecten zijn onlosmakelijk met elkaar verbonden
* Het ene object kan niet bestaan zonder het andere
* Als het ene ophoudt te bestaan zal het andere ook ophouden te bestaan

Voorbeeld: Relatie tussen een huis en een kamer

* Als het huis wordt afgebroken zal ook de kamer verdwijnen. (De kamer kan niet bestaan zonder het huis).

In het UML-Diagram wordt compositie voorgesteld met een volle ruit.

Voorbeeld :





Het verschil tussen de soorten relaties, namelijk associaties, aggregaties en composities is vooral theoretisch en concuepteel. In Java wordt een relatie tussen 2 objecten altijd gerealiseerd door het ene object een referentie te laten hebben naar het andere object. De drie verschillende soorten relaties worden dus technisch op dezelfde manier gerealiseerd. Referenties naar andere objecten kunnen soms wel gedeeld worden (aggregatie) en soms niet (compositie).

**9.5 High Cohesion**

* Klasse moet gericht zijn op 1 kerntaak
* Hierdoor kan de klasse hergebruikt worden

Vb. Klasse Random: enkel verantwoordelijk om een willekeurig getal te genereren.

**Hoofdstuk 10: Overerving en klassenhiërarchie**

Overerving is een andere manier om al geschreven code te hergebruiken binnen de OO -filosofie.

**10.1.1 Subklassen en superklassen**

Klassen kunnen gebaseerd zijn op of afgeleid zijn van andere klassen

Er wordt een nieuwe klasse aangemaakt (=subklasse)

Op basis van een bestaande klasse (= superklasse)

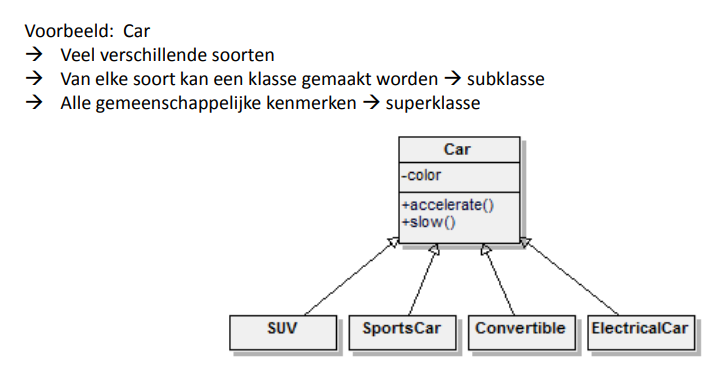
**10.1.2 Overerving**

De subklasse:

* Erft alle eigenschappen en methoden over van de superklasse
* Er kunnen eigenschappen en methoden toegevoegd worden en
* De implementatie van bepaalde methoden kan vervangen worden (override)
* Heeft een IS EEN relatie
  + Een Hond IS EEN Dier
  + Een TerreinWagen IS EEN Wagen
  + …

Het concept van overerving in objectgeoriënteerd programmeren zorgt ervoor dat we de code kunnen hergebruiken (code reuse). De code wordt in een superklasse gemaakt en vervolgens hergebruikt in 1 of meerdere subklassen.

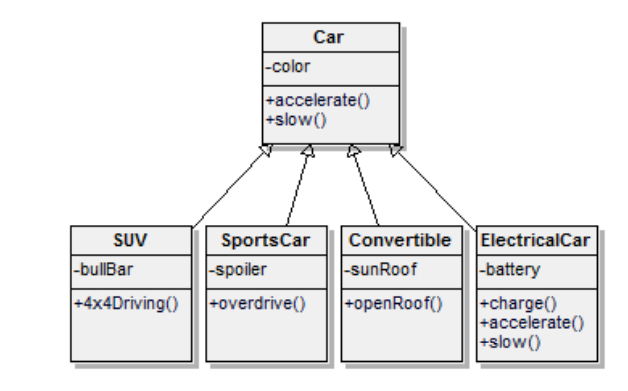
Voorbeeld:



Tegelijkertijd voegt de subklasse ook een aantal eigenschappen en methoden toe die specifiek zijn voor die subklasse.

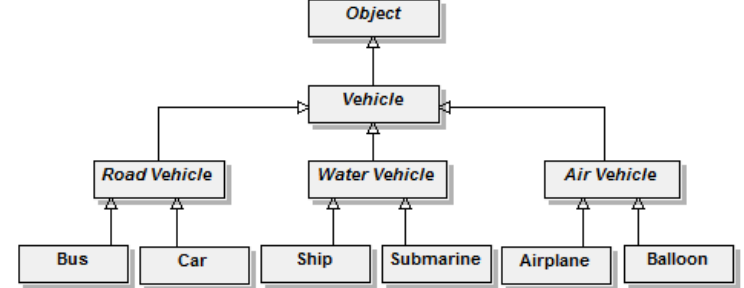
Een subklasse kan ook methoden vervangen (override) zodat die specifiek worden voor die specifieke subklasse. Neem het voorbeeld met de Car, een elektrische wagen versnelt en vertraagt anders dan een wagen die op benzine rijdt. Dus elke subklasse heeft een andere implementatie van accelerate() en slow().

UML Diagram:



**10.1.3 Klasse-hiërarchie**

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



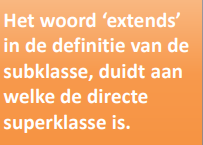
**10.1.4 Abstracte klassen**

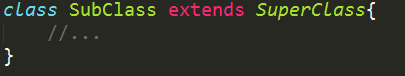
Van abstracte klassen worden andere klassen afgeleid.

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

**10.2 Subklassen definiëren**

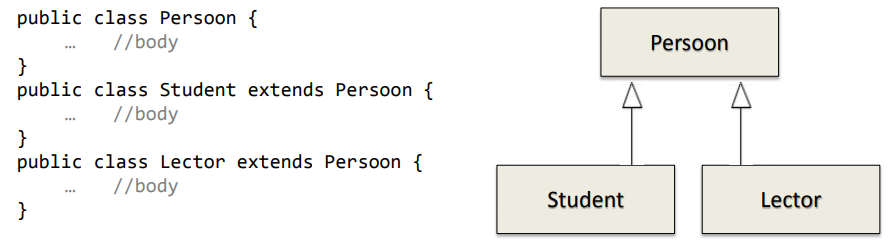
Nieuwe klasse maken vertrekkende van een bestaande klasse : extends





De klasse SubClass:

* Is een ‘uitbreiding’ van de klasse SuperClass
* Erft eigenschappen en methoden over van SuperClass
* Kan eigenschappen en methoden toevoegen/vervangen



**10.3 Eigenschappen van subklassen**

**10.3.1 Overerven van eigenschappen**

Subklassen erven alle eigenschappen van hun superklasse over, zowel instance-eigenschappen als klasseneigenschappen. De toegang ertoe hang af van het toegangsniveau

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Toegangsniveau** | **Eigen klasse** | **Pakketklassen** | **Subklassen** | **Alle klassen** |
| *Private* | X |  |  |  |
| * *(package)* | X | X |  |  |
| *Protected* | X | X | X |  |
| *Public* | X | X | X | X |

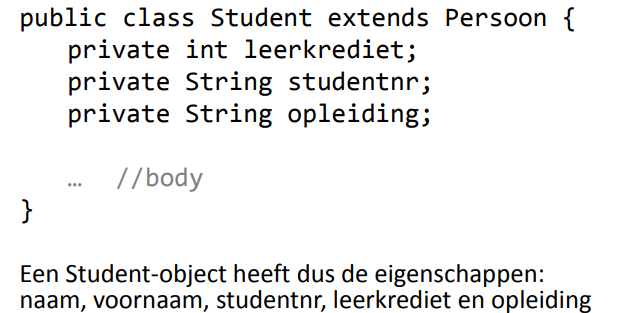
Enkel *protected* en *public* eigenschappen zijn (direct) toegankelijk vanuit een subklasse.

**10.3.2 Toevoegen van eigenschappen**

Een subklasse kan eigenschappen toevoegen die niet bestaan in de superklasse.

Voorbeeld:

Een klasse Student heeft extra eigenschappen ten opzichte van een gewone Persoon (naam, voornaam): studentennr, leerkrediet, opleiding.



**10.3.3 Vervangen (verbergen van eigenschappen)**

Als men in de subklasse een variabele declareert met dezelfde naam als een variabele uit een superklasse, dan wordt de variabele van de superklasse verborgen door de variabele uit de subklasse (ghosting)

**10.4 Methoden van subklassen**

**10.4.1 Overerven van methoden**

Subklassen erven alle methoden over van de superklasse: Zowel instance – methoden als klasse – methoden.

De toegang ertoe hangt af van het toegangsniveau (zie boven)

**10.4.2 Toevoegen van methoden**

Een subklasse kan methodes toevoegen, die niet bestaan in de superklasse, die eigen zijn aan de subklasse.

**10.4.3 Vervangen van methoden (override)**

Net zoals bij overgeërfde eigenschappen zijn de functies van de superklasse verborgen. Methoden die vervangen worden moeten dezelfde signatuur (naam van de methode + aantal en type van de parameters) hebben.

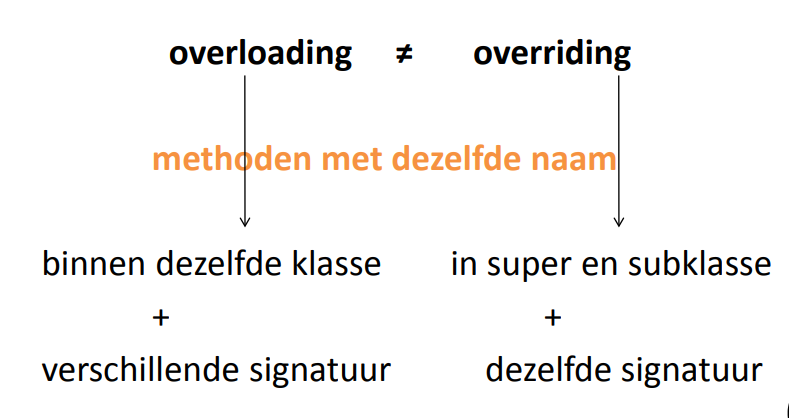
Bij methoden die vervangen worden moet ook het terugkeer type hetzelfde zijn. (Voor objecten mag dit een subklasse zijn = covariant return type).

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

De compiler controleert @override

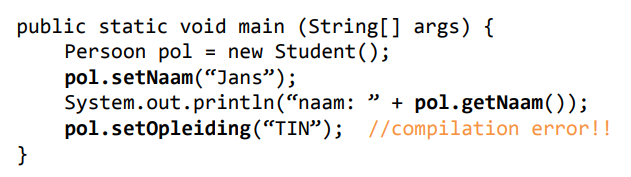
Subklassen kunnen het toegangsniveau van overschreven methoden enkel vergroten, niet verkleinen. Private methoden worden overgeërfd, maar zijn niet toegankelijk in de subklasse.

Verschil tussen overloading en overriding:



**10.4.4 Polymorfisme**

Voorbeeld:

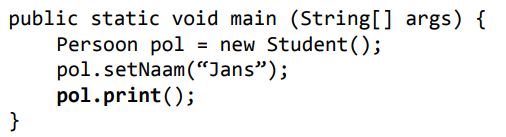


Opgepast!

Object pol: kan enkel methodes gebruiken die bestaan in de klasse pol. Je zegt tegen de computer dat het een persoon is maar stiekem steek je er student in.

In de loop van de code kan je de variabele wijzigen: pol = new Persoon(); en alles blijft werken.

Wat als we een methode print() aanroepen die zowel in Persoon als Student bestaat:



De printmethode van Student zal worden opgeroepen.

De koppeling van de uit te voeren code gebeurt pas tijdens de uitvoeren van het programma

* Late binding
  + Tijdens de compilatie is niet geweten welke methode opgeroepen gaat worden, want misschien is de inhoud van de variabele pol wel afhankelijk van de keuze van de gebruiker. Dus pas tijdens de uitvoering van het programma wordt er gekeken wat het concrete object nu juist is en welke functie moet worden opgeroepen.

Polymorfisme is alleen van toepassing op methodes, niet op variabelen!

**10.5 Constructors van subklassen**

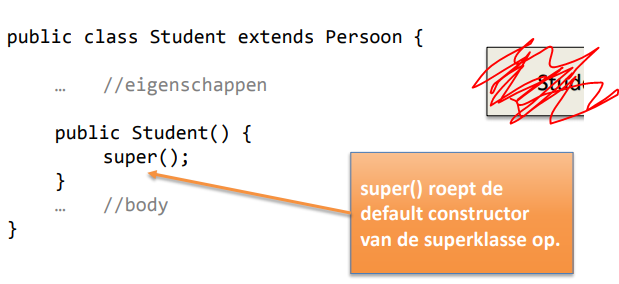
Constructors worden niet overgeërfd van de superklasse.

Iedere subklasse moet zijn eigen constructors hebben en daarbij eventueel een constructor van de superklasse aanroepen.

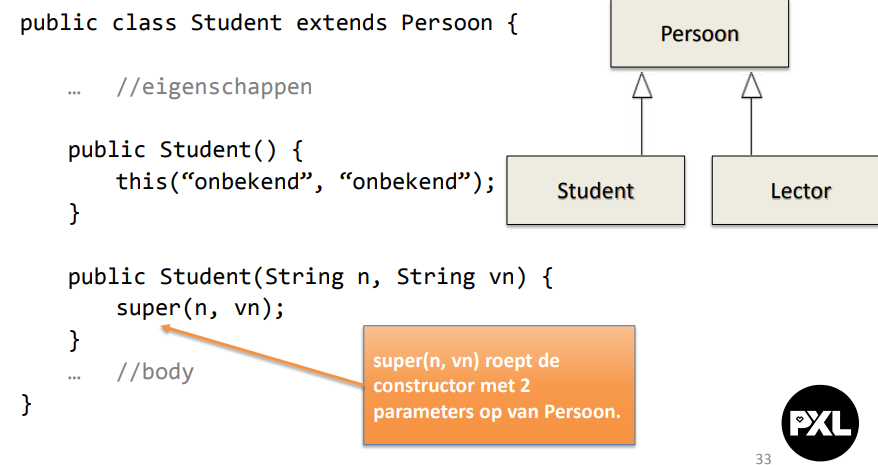
Super(*eventuele parameters*):

* Roept de constructor van de superklasse op
* Moet de eerste instructie in de constructor zijn
* Als de constructor van de superklasse niet expliciet opgeroepen wordt, wordt automatisch de default constructor opgeroepen
  + Werkt enkel als de supperklasse ook zo’n constructor heeft (zonder parameters)
* De aanroep van de constructor van de superklasse gebeurt voor het uitvoeren van andere statements in de constructor van de subklasse (Uitzondering: wanneer de constructor een andere constructor van de subklasse oproept).

Zonder constructors

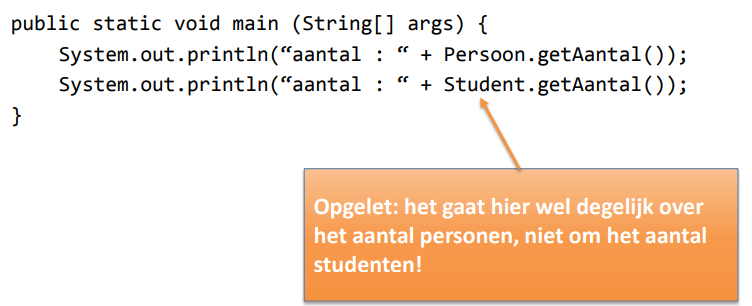


Met constructors



**10.6 Klasseneigenschappen en klassenmethoden**

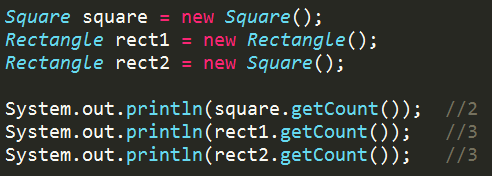
Klasseneigenschappen en klassenmethodes worden overgeërfd van superklassen.



Indien we afzonderlijk het aantal studenten willen kennen, moeten in de klasse Student de eigenschap “aantal” en de methode “getAantal()” geherdefinieerd worden!

Statische methoden kan men ook oproepen op een object in plaats van een klasse

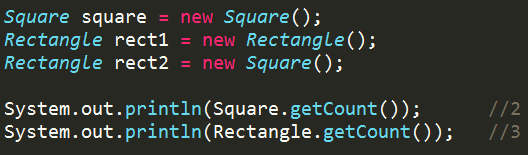
* VERMIJDEN, zeker bij objecten van subklassen



Ook al is rect2 eigenlijk een vierkant, toch zal de methode getCount() van Rectangle opgeroepen worden omdat rect2 van het type Rectangle is. Dus de methode die uitgevoerd wordt hoort bij het datatype van de referentievariabele.

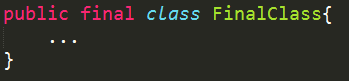
* Later binding is niet geldig bij statische methoden

Oplossing: gebruik klassenaam



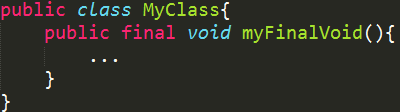
**10.7 Final klassen en methoden**

Final klassen zijn klassen die geen subklassen kunnen hebben. Ze worden gedefinieerd met het woord *final*.



Als we nu een klasse maken waarvan FinalClass de superklasse is, krijgen we een compilatie error.

Final methods zijn methods die niet overschreven kunnen worden in subklassen.



Als we nu een klasse maken waarvan FinalClass de superklasse is, en we proberen de final functie in de superklasse te overschrijven in de subklasse. Dan krijgen we een compilatie error.

3 belangrijke redenen voor final-klassen en methoden:

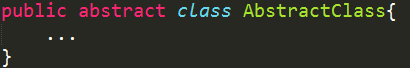
* Beveiliging: men kan geen subklassen maken met een heel andere implementatie
* Ontwerp: bij het ontwerp van de klasse-hiërarchie kan men beslissen dat bepaalde klassen “af” zijn
* Snelheid: bij final-klassen of methoden weet de compiler vaak reeds bij compilatie welke implementatie van een methode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

**10.8 Abstracte klassen**

Abstracte klassen:

* Kunnen geen concrete instanties hebben
* Dienen enkel om een abstract kader te scheppen voor de subklassen
  + Shape
  + Dier
  + Voertuig
  + …
* Dienen om zoveel mogelijk gemeenschappelijke code onder te brengen die vervolgens overgeërfd wordt door de subklassen

Abstracte klasses worden als volgt gedefinieerd:



Abstracte klassen hebben dan wel geen concrete instanties maar ze hebben wel constructors. Die we ook kunnen oproepen in de constructor van de subklasse. Voor de rest werkt alles met constructors hetzelfde als bij normale klassen.

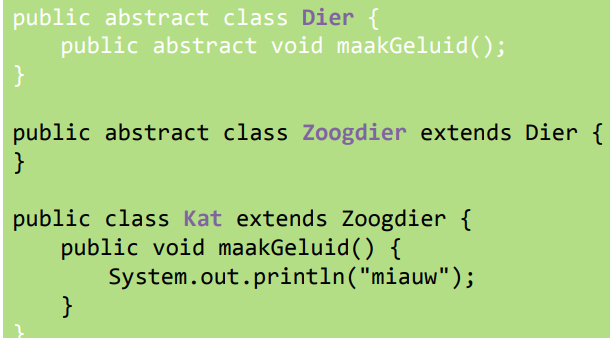
Abstracte methodes worden als volgt gedefinieerd:



Enkel abstracte klassen kunnen abstracte methoden hebben.

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

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



**10.9 De superklasse Object**

**10.9.1 Klassenhiërarchie**

De superklasse van alle klassen in Java is de klasse Object

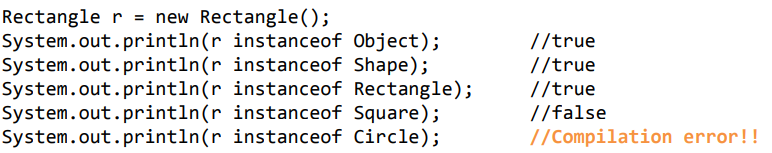
Iedere klasse is rechtstreeks of onrechtstreeks een subklasse van Object. De object klasse bepaalt de algemene toestand en het gedrag van een object.

**10.9.2 De operator instanceof**

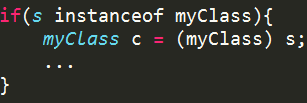
Iedere klasse behoort dus tot dezelfde boomstructuur waarbij het hoogste element de klasse Object is.

De operator *instanceof*:

* Werkt enkel bij objecten die in dezelfde hiërarchische tak zitten als de klasse.
* Wordt gebruikt om na te gaan of een object een instant is van 1 of andere (super)klasse
* Geeft als resultaat een boolean terug
* Wordt gebruikt voor een veilige “downcast”
  + Het forceren van een referentievariabele naar een andere datatype: doorgaans is dat een datatype van een lager niveau in de klassenhiërarchie(downcast)

Voorbeeld: 

Voorbeeld van downcast:



**10.9.3 Methoden van de Object-klasse**

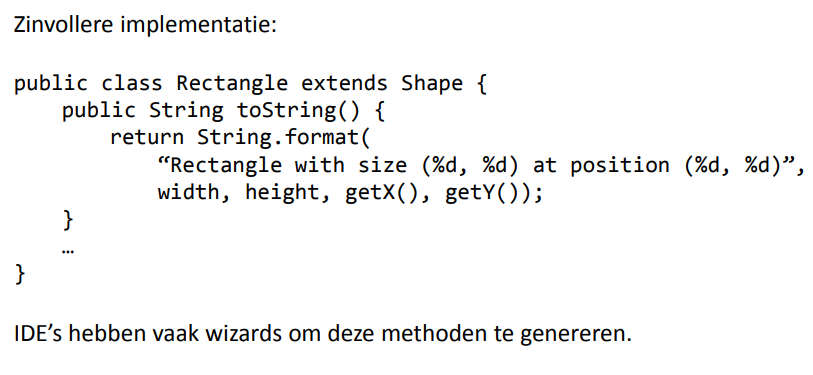
De methoden van de klasse Object worden door ALLE klassen overgeërfd.

Sommige van deze methode zijn final en kunnen niet vervangen worden.

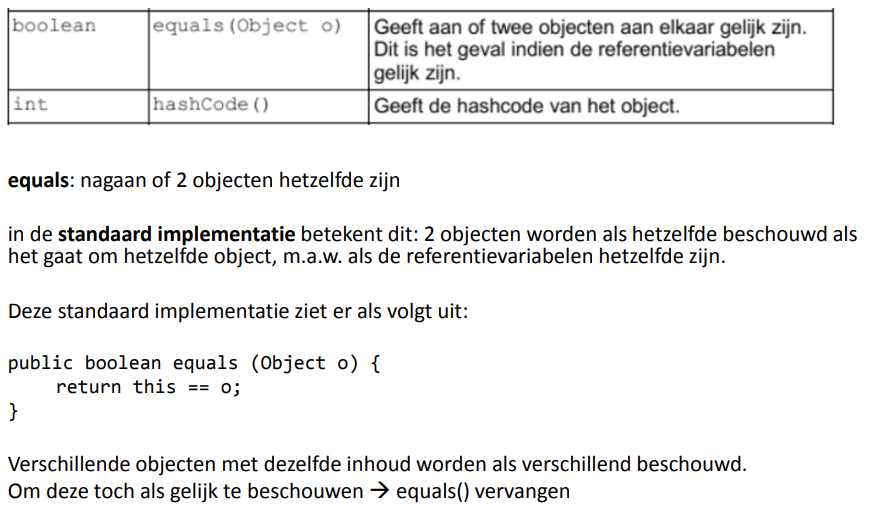
|  |  |  |
| --- | --- | --- |
| **Return-type** | **Methode** | **Beschrijving** |
| String | toString() | Geeft een string terug waarin het object beschreven wordt. |
| Boolean | Equals(Object o) | Geeft aan of twee objecten aan elkaar gelijk zijn. Dit is het geval indien de referentievariabelen gelijk zijn. |
| Int | Hashcode() | Geeft de hashcode van het object |
| Class | getClass() | Geeft een object terug dat de concrete klasse voorstelt. |

**10.9.3.1 tostring()**





**10.9.3.2 equals() en hashCode()**





**10.9.3.3 getclass()**



**10.10 Polymorfisme (bis)**

Korte uitleg:

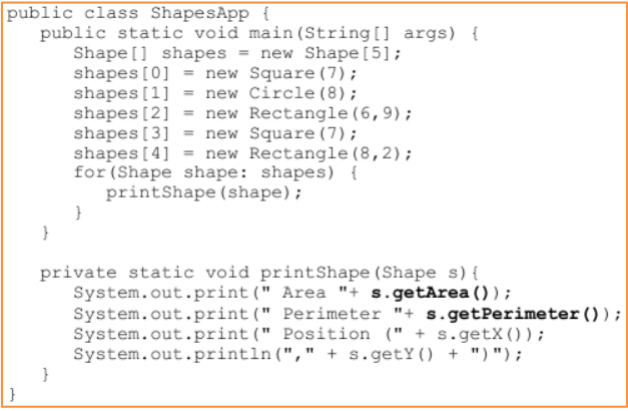
Polymorfisme = een object dat veel vormen kan aannemen

* Gedeclareerd als type van een superklasse
* Gecreëerd als type van een subklasse

Lange uitleg:

Objecten van een klasse die afgeleid is van een superklasse kunnen beschouwd en behandeld worden als objecten van die superklasse. Dit noemt men polymorfisme. Een object van de superklasse kan als het ware veel (poly) vormen (morf) aannemen, zich voordoen als een object van verschillende klassen.

Voorbeeld:



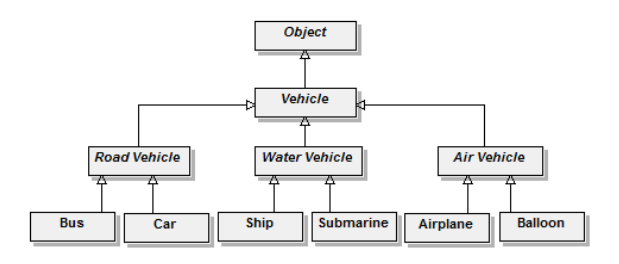
getArea() en getPerimeter() zijn in Shape gedefinieerd

* Beschikbaar voor alle objecten van een superklasse
* Welke methode precies gebruikt wordt, beslist tijdens de uitvoering van het programma (late binding)

**10.11 Code hergebruik: overerving versus associaties**

Het doel van object-georiënteerd programmeren is het hergebruiken van bestaande code. Dit kunnen we op 2 manieren realiseren:

1. **Associaties**: een klasse heeft een relatie met een andere klasse
   1. “HAS A” – relatie
   2. Een persoon **heeft een** adres, een persoon **heeft een** huisdier
2. **Overerving**: een klasse erft over van een superklasse en hergebruikt daarmee de overgeërfde code.
   1. “IS A” -relatie
   2. Een student **is een** persoon, een rechthoek **is een** figuur
   3. Relatie kan getest worden door *instanceof*



Een auto **is een** wegvoertuig en een bus **is een** wegvoertuig. Dit is overerving.

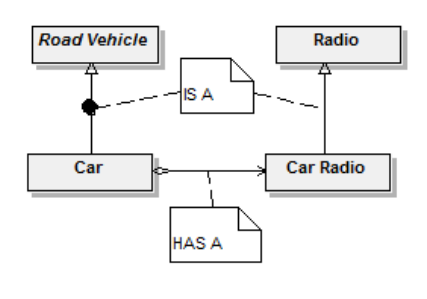
Stel dat we zowel een auto als een bus willen voorzien van een radio. We kunnen dan een bestaand toestel (autoradio) inbouwen

* Associatie

Deze autoradio zou dan kunnen afgeleid zijn van de klasse radio (de autoradio is een radio)

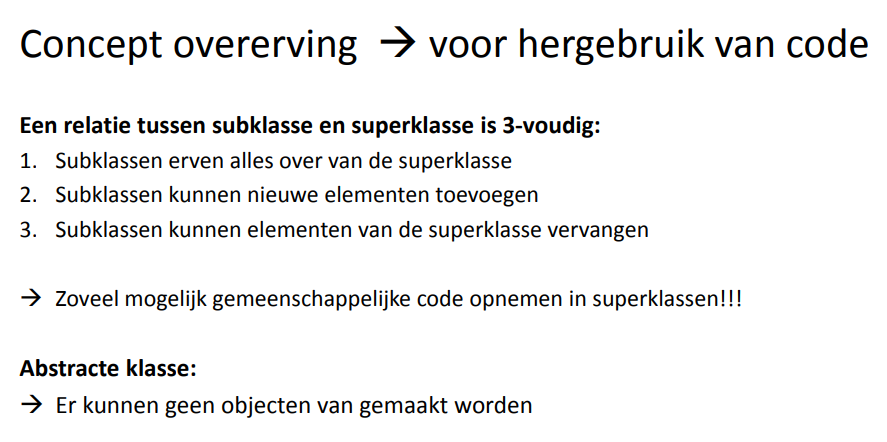
* Overerving

Een auto **is een** wegvoertuig en **heeft een** autoradio. Een autoradio **is een** radio.



Vaak hebben we dus een combinatie van overerving en associatie om het code hergebruik te maximaliseren.

Recap:



**Hoofdstuk 11: De opsomming**

Het opsommingstype:

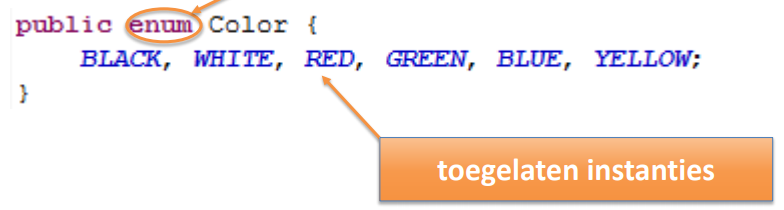
* is een klasse waarvan beperkt aantal instanties beschikbaar zijn.
  + Deze instanties worden tijdens de declaratie van de enumeratie opgesomd
  + => We kunnen zelf geen instanties maken
  + Deze instanties worden als constanten gebruikt (met hoofdletters)
  + Kan men beschouwen als statische eigenschappen van de klasse.

Waarom enum?

Het is veiliger en duidelijker dan een gewone klasse met statische variabelen

**11.1 Definitie en gebruik**

Definitie van het opsommingstype



De enum is een subklasse van de klasse enum en erft dan ook de methoden van de klasse over.

.name() - geeft de naam weer

.ordinal() - geeft de volgorde weer

.values() - array van enums in de gedefinieerde volgorde

Als we de waarden van het opsommingstype vaak nodig hebben kunnen we de enum static importeren



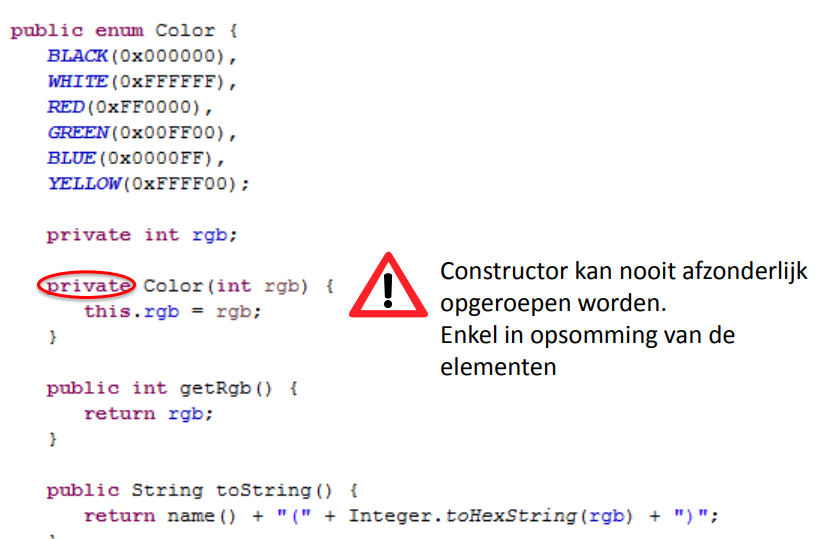
**11.2 Eigenschappen, methoden en constructors**

Net als bij andere klassen beschikken opsommingen over eigenschappen, methoden en constructors.

Zo erft het opsommingstype eigenschappen en methoden over van de superklasse Enum, en kunnen er eigenschappen en methoden toegevoegd en vervangen worden.

Als er geen constructor is gedefinieerd neemt de compiler de default constructor. De constructor van een enum is impliciet private.

We kunnen ook argumenten meegeven die dan eigen zijn aan de gegeven instanties iqn de enum.



**Hoofdstuk 12: eenvoudige klassen**

**12.1 Wrappers voor primitieve datatypes**

Voor elk primitief datatype **(int)** bestaat er een object variant = wrapperklasse(Integer)

Wrapper objecten zijn immutable, als we ze veranderen wordt er een nieuw object aangemaakt.

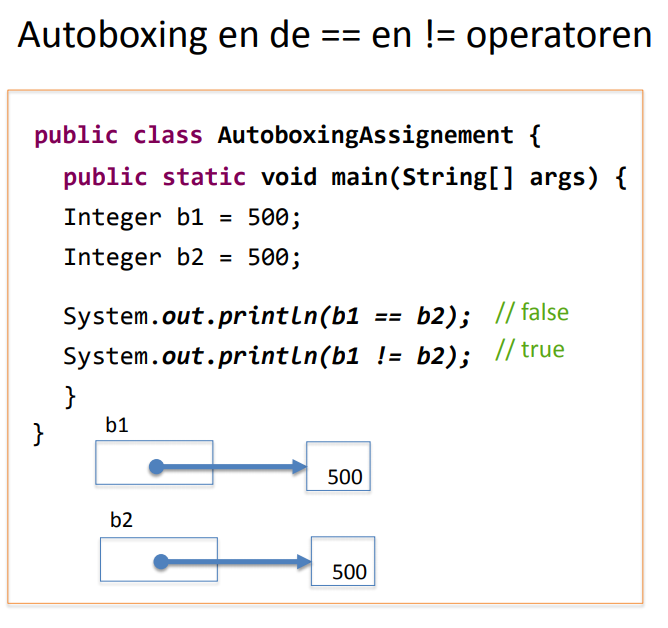
|  |  |
| --- | --- |
| **Klasse** | **Primitief datatype** |
| Byte | byte |
| Short | short |
| Integer | int |
| Long | long |
| Float | float |
| Double | double |
| BigInteger | Geheel getal met onbeperkte precisie |
| BigDecimal | Decimaal getal met onbeperkte precisie |
| Boolean | boolean |
| Character | char |

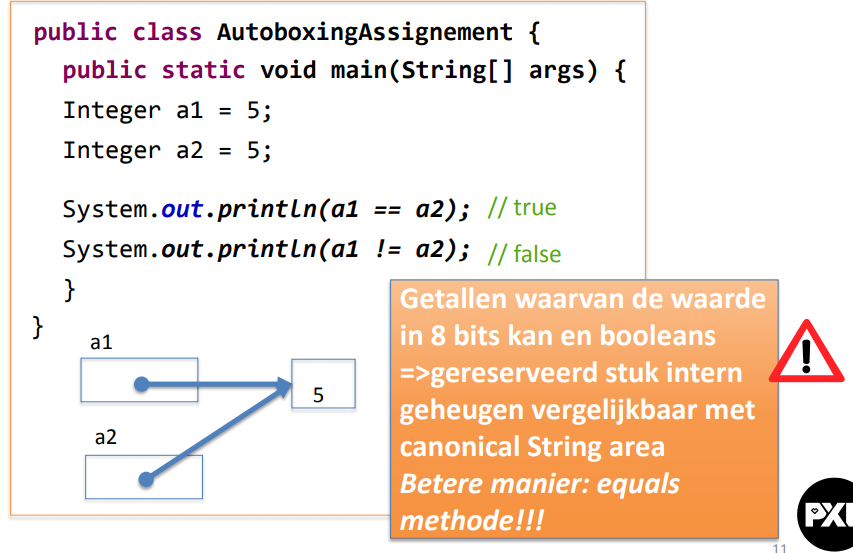
Autoboxing: primitief datatype wordt omgezet naar wrapper object

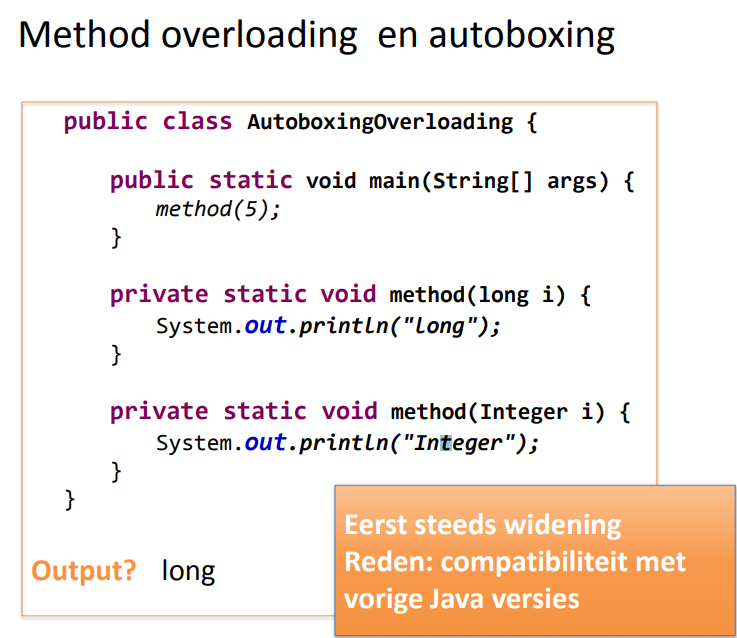
auto-unboxing: wrapper object wordt omgezet naar primitief datatype

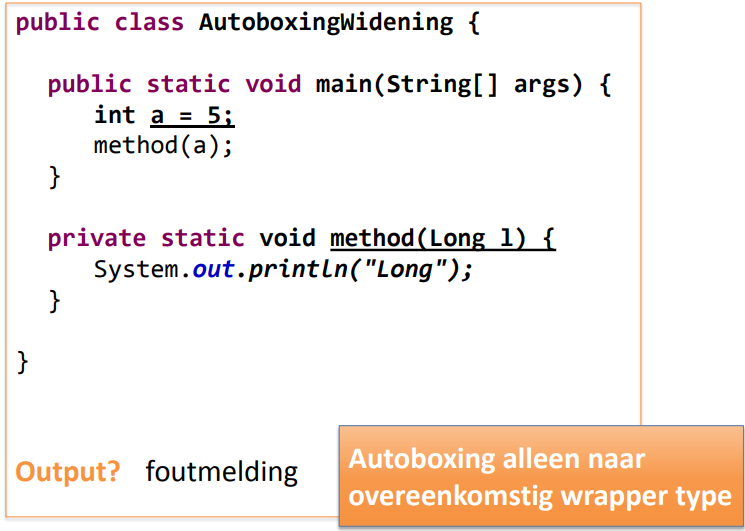
omzettingen gebeuren automatisch door de compiler. Wrapper objecten en primitieve datatypes mogen door elkaar gebruikt worden.

Wanneer er veel gerekend moet worden niet met objecten werken.









Wrapperklassen hebben ook hun eigen static variabelen en methods, zie javaDoc.

Conversiemethodes:

String -> int : Integer.parseInt();

String -> float : Float.parseInt();

**12.2 Datums en tijden**

Sinds Java 8 zijn er nieuwe klassen in java.time

Werken met tijd en datums brengen enkele moeilijkheden met zich mee:

* 2 manieren om naar tijd te kijken
  + mens: jaren, maanden, uren, …
  + Computer: aantal tijdseenheden vanaf een bepaald punt (Epoch, 1970)
* We werken met tijdzones
  + Tijd verschilt op ieder punt op de aarde
    - Europe/London Z Greenwich
    - Europe/Brussels +01:00
    - America/New\_York -05:00
* We werken met schrikkeljaren
* Zomertijd

**12.2.1 de klasse instant**

Machines hebben geen notie van jaren, maanden, dagen enzovoort. Maar enkel van de fundamentele tijdseenheden zoals seconden, milliseconden of nanoseconden. Een tijdstip wordt aangeduid met het aantal tijdseenheden voor of na een soort nulpunt.

Computertijd = aantal tijdseenheden voor of na een nulpunt

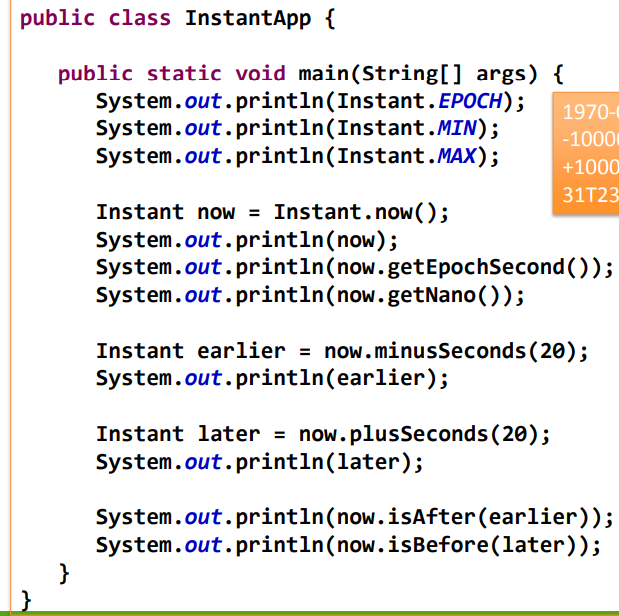
Nulpunt = 1 januari 1970 om 00:00:00 (EPOCH)



Klasse Instant heeft geen publieke constructor => onmogelijk om zelf instanties van klasse Instant aan te maken .

Dus statische methoden geven een instantie van de klasse terug.

De klasse bevat ook methodes om bewerkingen te doen op een tijdstip.



**12.2.2 Menselijke datums en tijden**

**12.2.2.1 enumeraties voor weekdagen en maanden**

Enumeratie Month

* Bevat 12 instanties voor de maanden
* methoden voor bewerkingen en informatie (zie javadoc, cursus)

Enumeratie DayOfWeek

* Bevat 7 instanties voor de dagen
* methoden voor bewerkingen en informatie (zie javadoc, cursus)

**12.3 Lokale Datums en tijden**

Mensen beschouwen de datum en tijd in eerste instantie zoals die datum en tijd op hun plek beleefd wordt. Dus lokale tijd houdt geen rekening met de tijdzones of het zomeruur.

De klassen die voor lokale datums en tijden gebruikt worden:

* LocalTime
  + Lokale tijd in uren, minuten, seconden, …
* LocalDate
  + Lokale datum in jaren, maanden, dagen, …
* LocalDateTime
  + Lokale datum en tijd gecombineerd

Ook deze klassen hebben geen constructors en maken gebruik van statische methodes die instanties teruggegeven.

.now() geeft nu terug

.of(args) geeft een instantie terug met de gegeven parameters.

**12.4 Tijdzones**

Datums en tijd die rekening houden met de plek op de aarde en eventueel de regionale verschillen inzake zomertijd.

Tijdzones bevatten twee soorten informatie

1. info over regio/stad (bvb Europe/Brussels).
   1. Klasse **ZoneId** tijdzone gebaseerd op regio/stad
2. Info over verschil met standaard tijdzone UTC/Greenwich
   1. Klasse **ZoneOffset** gebaseerd op verschil.met standaard tijdzone.
   2. Klasse **ZoneOffset** subklasse **ZoneId**

Ook deze klassen hebben geen constructors en maken gebruik van statische methodes die instanties teruggeven.

**12.4.1 Datums en tijden met tijdzones**

De klasse ZonedDateTime stelt ons in staat datums en tijden te gebruiken die rekening houden met de tijdzones.

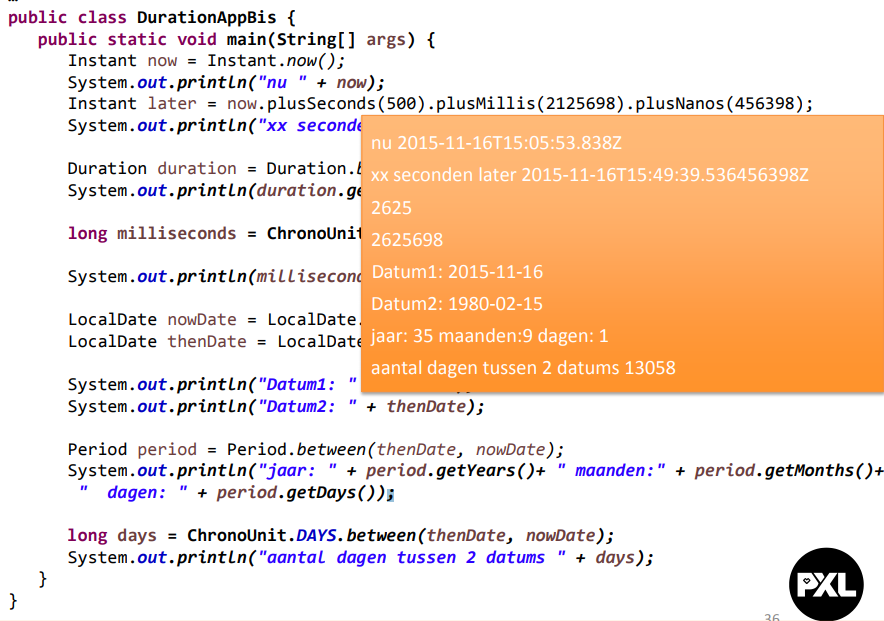
Ook deze klasse heeft geen publieke constructor en maakt gebruik van statische methodes die instanties teruggegeven.

Andere methoden: zie javadoc

**12.5 Tijdsduur**

Om een tijdsduur aan te geven gebruiken we de volgende klassen:

* Duration
  + Tijdsverschillen tussen machinetijden (= instanties van de klasse Instant)
* Period
  + Tijdsverschillen tussen “datebased-time (= menselijke tijden, LocalDate etc ..)
* ChronoUnit: enumeratie
  + Verschillende tijdseenheden: ChronoUnit.HOURS, ChronoUnit.DAYS, …
  + Tijdsverschil kan met een methode between() omgezet worden in de gekozen tijdseenheid.



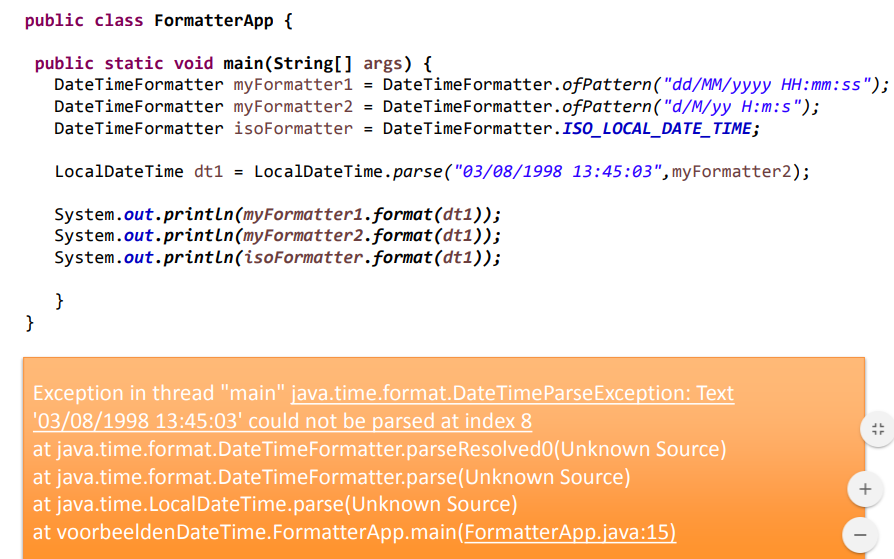
**12.6 Formattering van datums en tijden**

Datums en tijden kunnen in de vorm van tekst op verschillende manieren worden voorgesteld. Hiervoor gebruiken we de klasse DateTimeFormatter.

DateTimeFormatter wordt gebruikt voor:

* De conversie tussen tekst-voorstelling en Java datum/tijd - objecten
* De conversie van tekenreeks naar datum/tijd-object (parse-functie)
* De conversie van datum/tijd-object naar een tekenreeks (format - functie)
* volgens eigen opgegeven patroon of volgens standaard patronen datum/tijden in te geven of te printen
* Verdere uitleg zie Javadoc



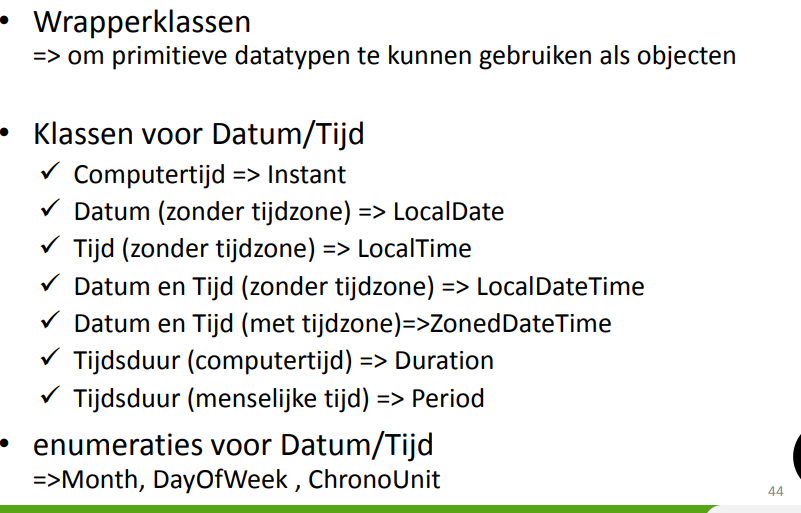


**12.7 Omzetten van en naar Date en Calendar**

Voor Java 8 werden voor datums en tijden gebruik gemaakt van de klassen Date en Calendar.

Uitgebreid met methoden om ze om te zetten naar objecten van de klasse Instant of ZonedDateTime na Java 8

**12.8 Samenvatting**

****

**Hoofdstuk 13: Interfaces**

Het is mogelijk een soort abstracte klasse te maken die alleen maar abstracte methoden heeft, en geen concrete implementaties.

**Een interface:** is een verzameling van abstracte methoden die het mogelijk maakt op een bepaalde manier met een object om te gaan. Een interface kan ook constanten en *default* en *static* methoden bevatten.

Sinds Java 8 kan een interface ook default methoden en static methoden bevatten.  
Het doel van een interface is aan bepaalde klasse methoden opleggen die moeten geïmplementeerd worden.

Een klasse die een bepaalde interface implementeert, moet dus ook voor alle methoden van die interface een implementatie hebben. Dit lijkt op de implementatie van abstracte methoden van een abstracte superklasse.

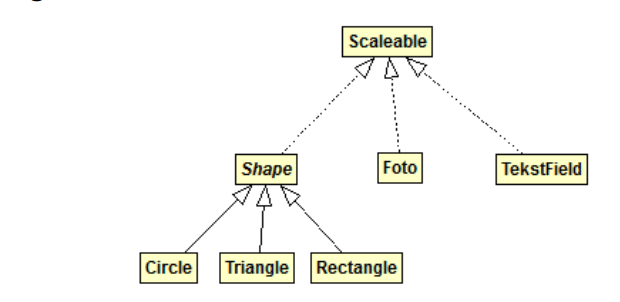
De interface kan men zien als een soort contract met buitenwereld waarin staat dat alle klassen die deze interface implementeren, daadwerkelijk een implementatie hebben van de opgelegde methoden. **Op die manier is het trouwens mogelijk objecten van verschillende klassen toch op eenzelfde manier te behandelen.** Dit is weer veelvormigheid of polymorfisme: objecten van dezelfde interface kunnen meerdere vormen aannemen en de concrete uitvoering van de methode wordt at run-time beslist. (Late binding)

Een klasse kan bovendien meerdere interfaces implementeren. In java is het niet mogelijk dat een klasse meerdere superklassen heeft, maar voor interfaces kan het wel.

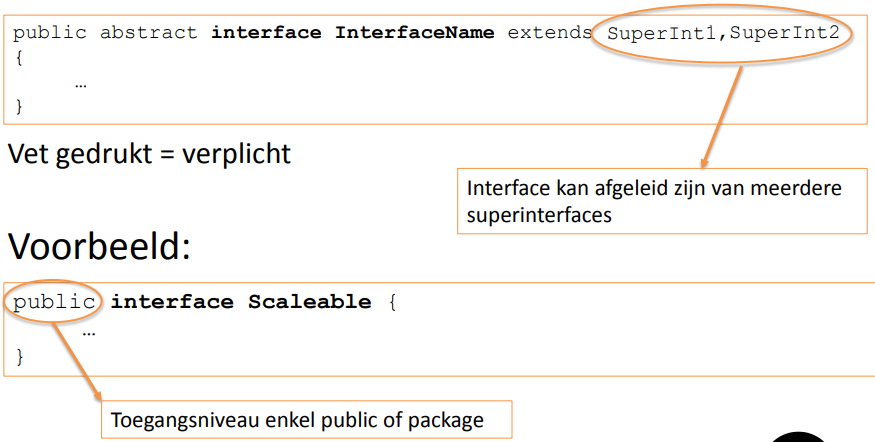
Waarom worden interfaces gebruikt:

* Interfaces maken het mogelijk om totaal verschillende objecten toch een identiek gedrag te geven, zonder dat ze daarom afgeleid zijn van dezelfde superklasse.
* Met interfaces kan men aan bepaalde klassen een aantal methodes opleggen die moeten geïmplementeerd worden.
* Men kan objecten met een bepaalde interface ter beschikking stellen zonder dat men de klasse van het object moet prijsgeven.
* Soms worden interfaces ook gewoon gebruik als een verzameling van constante waarden, dit is eerder een neveneffect.

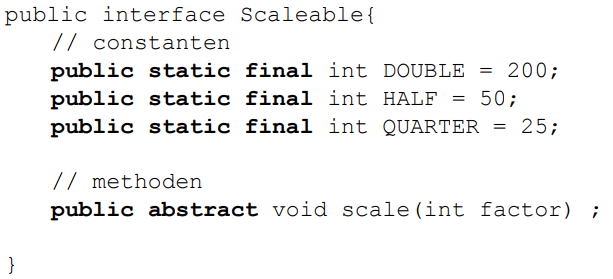
Voorbeeld:



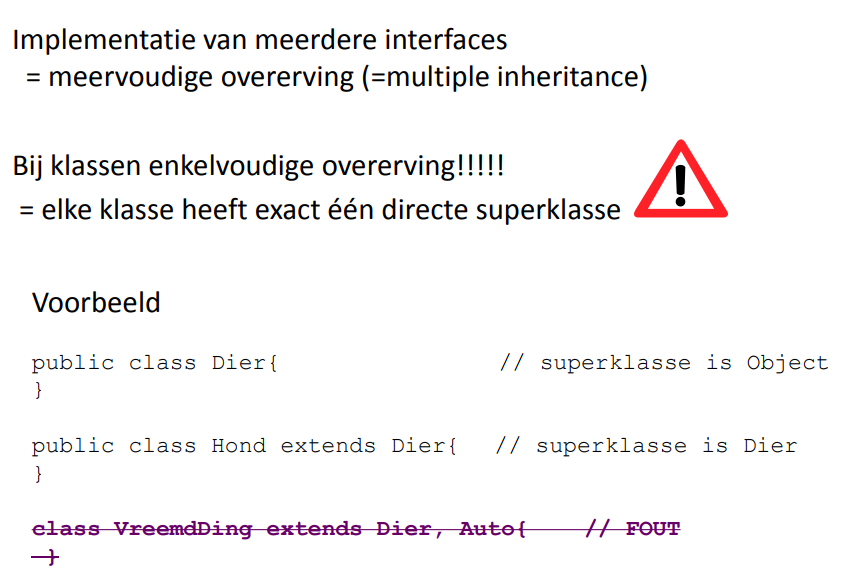
Declaratie van een interface:

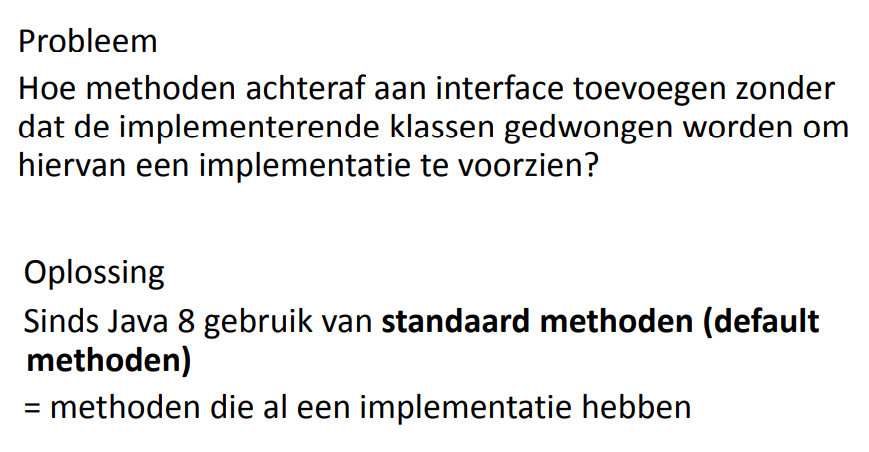


Declaratie van alle methoden en constanten:



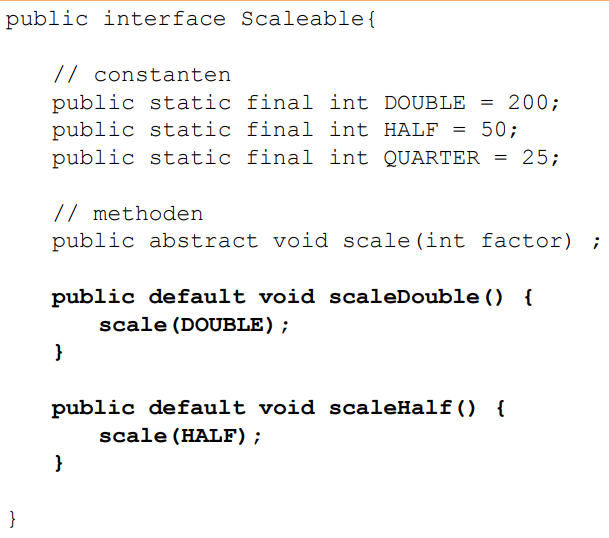
Interface extends interface, interface … mag wel, niet bij gewone klassen





Implementerende klassen of afgeleide interfaces kunnen met dergelijke methoden drie dingen doen:

* **Niets:** de implementerende klasse of subinterface neemt daarbij deze standaardimplementatie gewoon over.
* **Opnieuw abstract declareren:** Door de methode te herdefiniëren als abstracte methode verdwijnt voor deze klasse als het ware de standaardimplementatie. Concrete subklassen zullen terug verplicht worden een implementatie van de methode te voorzien.
* **Opnieuw implementeren:** Hierbij wordt de standaardimplementatie vervangen.

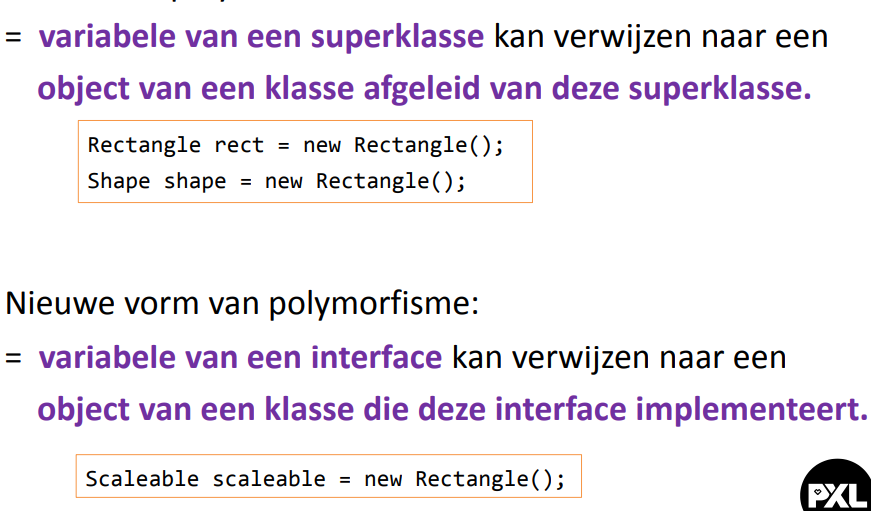


Dit is om te voorkomen dat oude code zou breken.

**13.1 Interface implementeren in een klasse**

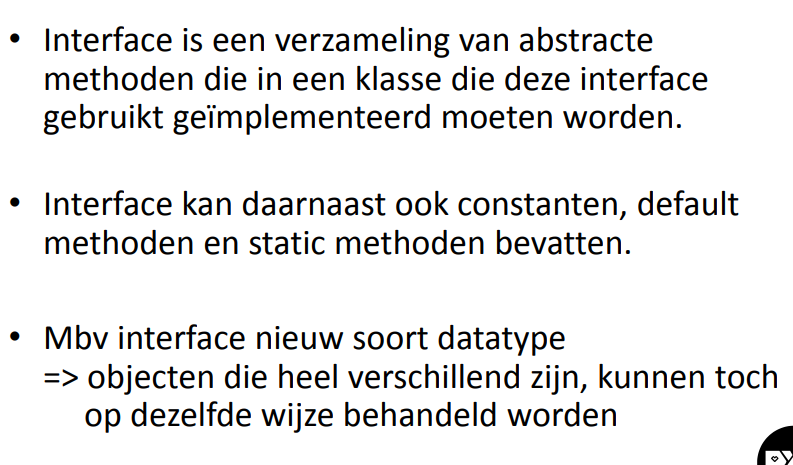
Een interface definieert een bepaald gedrag van een object. Indien een klasse een bepaalde interface implementeert, moet de klasse alle methoden van de interface en haar superinterfaces implementeren. Enkel een abstracte klasse is niet verplicht een implementatie te hebben. Van abstracte klassen kunnen geen objecten gemaakt worden en dus dient de eerstvolgende concrete subklasse van een abstracte klasse een implementatie te hebben voor alle abstracte methoden, inclusief de abstracte methoden van de interfaces.

**13.2 Interface als datatype**





**Samenvatting**



**Hoofdstuk 14:**