OOSD II - Hoofdstuk 1 - Overerving

Herhaling

Java Types

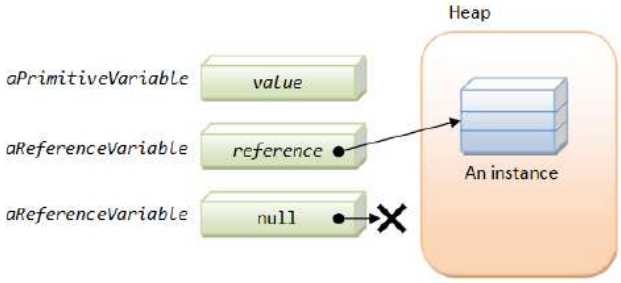
Primitief type	Referentie type
boolean	klassetype
byte, short, int, long, char	arraytype
float, double	

In het geval van een primitief datatype zit het datatype effectief op de plaats in het RAM. Terwijl bij referentie types zit daar enkel de verwijzing naar het adres in de Heap. In de heap bevindt zicht dan het datatype.

> **Tip**
>
> 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.

Een variabele met als type een referentietype omvat een referentie. Deze is ook een bepaalde waarde, met *een speciale betekenis*: Het is een verwijzing naar een object.

> **Tip**
>
> Bevat een referentie variabele de waarde `null`, dan verwijst de referentie niet naar een concreet object. Indien je op een referentie met de waarde `null` een methode aanroept dan krijg je bij het uitvoeren een `NullPointerException`



Object

> ⚠ Warning
>
> Een **object** is een instantie van een klasse of instantie van een array.

Referentiewaarden (of referenties) zijn:

- pointers (=verwijzingen) naar deze objecten
- of een `null` referentie, dewelke aangeeft dat er geen objecten bestaan om naar te refereren.

Klasse

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

Declaratie van een klasse:

```
public class MijnKlasse{
    //attributen, constructor(s)
    //methode declaraties
}
```

Overerving

> ⚠ Warning
>
> **Overerving** is is een mechanisme waarbij software opnieuw wordt gebruikt: nieuwe klassen worden gecreëerd 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.

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

> 💡 Tip

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

🎯 Overerving

Duidelijke hiërarchie opzetten met **een betere maintenance factor**

- Meer/eenvoudiger overzicht in structuur en gebruik
- Generalisatie (*algemene superklasse*) en specialisatie (*specifiekere subklassen*)
- Hergebruik van code

> **Important**
>
> Eigenschappen overerving
>
> - een subklasse erft van 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 subklasse hebben.
> - de hoogste klasse in elke klassenhiërarchie in Java is de klasse `Object`

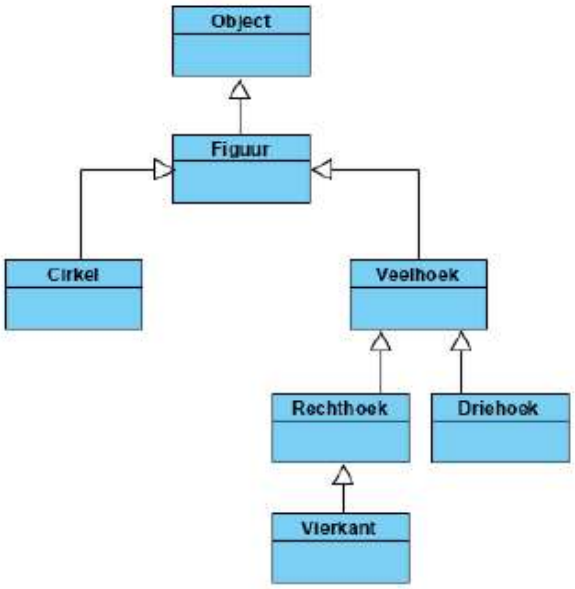
> **Caution**
>
> **Een constructor is geen methode en wordt niet overgeërfd door de subklasse!**

> **Tip**
>
> Een UML diagram met overerving kan je lezen als een "IS EEN"-relatie. Lees je dezelfde pijl in de tegenovergestelde richting dan kan dit gelezen worden als een "KAN EEN"-relatie beschouwd worden. Wat rechtstreeks aanleiding geeft to **polymorfisme**

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

> **Tip**
>
> Polymorfisme is een handig concept om een lijst bij te houden van verschillende type variabelen of objecten.

In sommige context kan het nodig zijn om te verifiëren of een object van een bepaald referentie type is alvorens het te gebruiken. Dit doen we door het keyword `instanceof`.




```
Rechthoek rechthoek = new Rechthoek();
boolean a = rechthoek instanceof Rechthoek; //returned true
boolean b = rechthoek instanceof Veelhoek; //returned true
boolean c = rechthoek instanceof Vierkant; //returned false
```

De klasse `Object`

Elke klasse is impliciet een subklasse van de klasse `Object`. (keyword `extends` hoeft niet vermeld te worden)

Object methoden

- **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

`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, echter zijn er belangrijke verschillen:

- `equals` is een methode, `==` is een operator
- De operator `==` wordt gebruikt om waarden te vergelijken. In geval voor referentiewaarden gaat het om adressen in het geheugen. De methode `equals` gaat vergelijken op basis van de inhoud.

> **Important**
>
> De operator `==` vergelijkt dus waarden met elkaar in tegenstelling tot de `.equals()`-methode die twee objecten vergelijkt op basis van hun toestand.

> **Important**
>
> Indien een klasse de methode `equals` niet overschrijft, dan zal de klasse deze overerven van de dichtsbijzijnde superklasse of de Object-klasse.

De methode `equals` bestaat ook in de klasse `Object` indien we zelf de klasse willen herschrijven zullen we steeds de `@Override` prefix moeten gebruiken.

De methodes `equals` en `hashCode` horen beide samen. Indien beide objecten aan elkaar gelijk zijn moeten steeds de gegenereerde hashCode identiek zijn.

> **Tip**
>
> Beide methodes kunnen automatisch gegenereerd worden door Eclipse. `right click > Source > "Generate hashCode() and equals()..."`

> **Tip**
>
> De `contains`-methode gebruikt de `equals`-methode om na te gaan of een object reeds bestaat in een lijst. Als we in een klasse zelf een `equals`-methode gespecifieer hebben zal hij ook deze gebruiken.

`toString` - methode

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. (bv. door het object in een print-statement te plaatsen)

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. (vb. een rechthoek-klasse kan ook een parent figuur-klasse hebben.

Wanneer `getClass()` wordt aangeroepen zal dit altijd de rechthoek-klasse teruggeven) Methode bevat verschillende info over de klasse, die via hulpmethodes kan aangeroepen worden

- `getClass().getSimpleName()`: geeft String ipv java.lang.String
- `getClass().getSuperClass()`: geeft de superklasse van de klasse

Initialisatie van een hiërarchie

❗ Constructoren worden nooit overgeërfd: ze kunnen dus ook niet overschreven worden.

> **Important**
>
> Als een klass geen declaratie voorziet van een constructor, dan wordt er impliciet een default constructor gedeclareerd.

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

> **Note**
>
> Als het eerste statement binnen een constructor body geen expliciete aanroep is van een andere constructor (en het gaat niet om de klasse Object) dan zal de constructor bodu **impliciet** beginnen met het aanroepen van de superklasse constructor `super()`, een aanroep van diens default constructor. Als de default constructor niet bestaat zal een constructor met de juiste parameters expliciet moeten aangeroepen worden.

Specialisatie van een klasse

> **Warning**
>
> Wanneer een methode gedeclareerd en geïmplementeerd is in een superklasse en later overschreven wordt in een subklasse, spreken we van **een specialisatie van een klasse**

> **Tip**

> Vergeet hierbij niet om steeds de prefix `@Override` te vermelden.

Scope van een declaratie

> ⚠ Warning
>
> **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.

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 schaduw geplaatste declaratie niet meer gebruikt worden om naar die gedeclareerde entiteit te verwijzen.

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

> ⓘ Note
>
> Vaak kunnen we de in schaduw geplaatste variabele/attribuut raadplegen door het aan te spreken via `this.naam_attribuut`

`abstract` keyword

Van *een abstracte klasse* kan geen instantie gecreëerd worden. We kunnen een methode of klasse abstract maken door het aanroepen van het keyword `abstract`

> ⚠ Warning

> **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 geïntstantieerd kan worden

Indien een klasse abstract is kan deze ook abstracte methodes declareren. Deze bevatten enkel een declaratie, zonder implementatie van de methode.

> **Note**
>
> Een niet abstracte klasse of methode benoemen we als **een concrete klasse of methode**

> **Warning**
>
> Een declaratie van **een abstracte methode** introduceert een methode als gedrag, waaronder zijn handtekening, return value en throws clausule indien gewenst, maar zonder de implementatie van de methode te voorzien.

> **Important**
>
> In UML wordt een abstracte methode of klasse in *italic* geschreven.

`static` keyword

`static` attribuut

Als een attribuut `static` wordt gedeclareerd bestaat er slecht één instantie van dat attribuut, ongeacht het aantal instanties dat van die klasse gemaakt worden.

> **Note**
>
> Een `static` attribuut, ook benoemt als **klassevariabele**, wordt aangemaakt als de klasse zelf voor het eerst geïnitialiseerd wordt. (= op het moment dat de klasse ingeladen wordt in de JVM, wat overeen komt bij de declaratie van het attribuut)

> **Note**
>
> Een attribuut gedeclareerd zonder het keyword `static`, ook wel non-static attribuut, noemt men **een instantievariabele**. Non-static attributen worden opnieuw aangemaakt en geassocieerd bij elke klasse-instantie.

`static` methode

> **Note**
>
> 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.

> **Note**
>
> Een methode niet gedeclareerd als `static` noemt men **een instantiemethode** of een non-static methode.

`final` keyword

`final` klasse

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

`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 de constructor.

> **Note**
>
> De afspraak in Java is om static final variabelen (de echte constanten) in hoofdletters te schrijven.

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

`final` variabele

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

> **Important**
>
> Als een variabele of final attribuut een referentie bevat naar een object of array, dan kan de toestand van dat object nog steeds wijzigen. Enkel de referentie naar het object of de array is final.

`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.

> **Caution**
>
> 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!

> **Tip**
>
> Zoveel mogelijk `final` gebruiken

Toegangscontrole

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