Inleiding Programmeren
College 10

Universiteit van Amsterdam

Robin de Vries

9 oktober 2017

Inhoud

Interfaces en overerving

Vorige week hebben we gekeken naar twee principes:

- Interfaces: een contract dat definieert welke methoden een klasse moet bevatten
- Overerving: een klasse erft alle methoden en variabelen van een andere klasse

Een interface is **niet** direct instantieerbaar en kan ook geen implementaties bevatten van methoden.

Er is ook een tussenweg: een **abstract**e klasse.

Abstracte klassen

Abstracte klassen hebben de volgende eigenschappen:

- Ze zijn niet instantieerbaar
- Bevatten zowel methoden **met** een implementatie,
- als methoden **zonder** een implementatie: abstracte methoden

In tegenstelling tot interfaces kan een abstracte klasse wel instantiatievariabelen bevatten.

Definieren van een abstracte klasse

```
abstract class Vorm {
    protected int x, y;

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    abstract int getOppervlakte();
}
```

De abstracte vorm klasse

Deze abstracte klasse heeft dus instantiatievariabelen, getters, en een abstracte methode.

Deze abstracte methode is hetzelfde als een methode-"definitie" in een interface.

```
Vorm vorm = new Vorm();
System.out.println(vorm);
```

Wat is de uitvoer van bovenstaande code?

```
error: Vorm is abstract; cannot be instantiated
        Vorm vorm = new Vorm();
```

Het compileert niet eens ;-) Een abstracte klasse kun je niet instantiëren!

Implementeren van een abstracte klasse

Vervolgens kun je net zoals een reguliere klasse, een abstracte klasse uitbreiden.
Als je alle abstracte methoden override, kun je hem dan als normale klasse instantiëren!

```
class Vierkant extends Vorm {

    int h, w;

    Vierkant(int x, int y, int w, int h) {
        this.x = x; this.y = y; this.w = w; this.h = h;
    }

    int getOppervlakte() {
        return w * h;
    }
}
```

Wanneer een interface en wanneer een abstracte klasse?

Abstracte klasse:

- Als je code wilt delen tussen verschillende (gerelateerde) klassen
- Als je non-static of non-final variabelen wilt gebruiken
- Als de sub en superklasse een “is”-relatie hebben (single inheritance).

Interface:

- Als de klasse en de interface niet per se volledig gerelateerd zijn
- Als je garanties wilt hebben over het bestaan van methoden maar verder niet bent geïnteresseerd in het gedrag van een klasse
- Als je gebruik moet maken van multiple inheritance

Wanneer ben je verplicht een klasse abstract te maken?

Als jouw klasse niet alle abstracte methoden van een abstracte superklasse of interface implementeert, **moet** jouw klasse ook `abstract` worden!

Inhoud

Overriding

Als een klasse een andere klasse uitbreidt, kan de **sub**klasse een methode van de **super**klasse opnieuw implementeren, dit noemen we: **overriding**.

@Override

Als je bewust een methode wilt overriden mag je ook de decorator `@Override` boven de definitie van de methode plaatsen.

De compiler waarschuwt je dan als jouw methode niet een andere methode override.

Override voorbeeld

```
class Speelgoed {

    public String toString() {
        return "Speelgoed!";
    }
}

class Bordspel extends Speelgoed {

    public String tostring() {
        return "Bordspel";
    }
}
```

Een tikfout is snel gemaakt! De **toString**-methode van de Bordspel klasse is nu niet anders dan die van de Speelgoed klasse.

Override voorbeeld (2)

```
class Speelgoed {

    public String toString() {
        return "Speelgoed!";
    }
}

class Bordspel extends Speelgoed {
    @Override
    public String tostring() {
        return "Bordspel";
    }
}
```

```
error: method does not override or implement a method from a supertype
    @Override
    ^
1 error
```

Inhoud

Constructoren met overerving

Als je een constructor definieert, **garandeert** Java dat deze wordt aangeroepen. Normaal gesproken gebeurde dit direct bij het aanmaken van een nieuw object. Maar hoe zit dat met overerving?

De constructor van een subklasse moet altijd de constructor van de superklasse aanroepen. Indien je dit niet zelf doet, zal Java dit zelf doen!

Constructoren met overerving

Aanschouw de volgende klassen:

```
class Persoon {
    String naam;
    public String toString() { return naam; }
}

class Student extends Persoon {
    int studentnummer;
}

class InprogStudent extends Student {
    int oefentoetsCijfer;
}
```

Let op: iedere `InprogStudent` heeft dus ook een studentnummer en een naam.

Constructoren met overerving (2)

Nu gaan we langzaam aan alle klassen een constructor toevoegen:

```
class Persoon {
    String naam;
    public String toString() { return naam; }

    Persoon(String naam) {
        System.out.println("Persoon constructor");
        this.naam = naam;
    }
}
```

Vervolgens compileert de code niet meer!

```
error: constructor Persoon in class Persoon cannot be applied to gi[..]
class Student extends Persoon {
^
  required: String
  found: no arguments
  reason: actual and formal argument lists differ in length
1 error
```

Waarom compileert de code niet?

De `Student`-klasse heeft geen constructor, dus voegt Java **impliciet** een "lege" constructor toe:

```
class Student extends Persoon {
    int studentnummer;

    Student() {
        super();
    }
}
```

Waarbij **super()** een aanroep naar de constructor van de **super** klasse is.
Omdat `Persoon` nu geen constructor zonder argumenten meer heeft, werkt dit niet.

Waarom compileert de code niet? (2)

Een mogelijke oplossing lijkt het zelf schrijven van een Student-constructor:

```
class Student extends Persoon {
    int studentnummer;

    Student(int studentnummer) {
        this.studentnummer = studentnummer;
    }
}
```

```
error: constructor Persoon in class Persoon cannot be applied to g[..]
    Student(int studentnummer) {
                               ^
  required: String
  found: no arguments
  reason: actual and formal argument lists differ in length
1 error
```

Waarom compileert de code niet? (2)

Maar ook dit compileert niet, omdat Java de code van de vorige slide, intern **impliciet** verbouwt naar het volgende:

```
class Student extends Persoon {
    int studentnummer;

    Student(int studentnummer) {
        super();
        this.studentnummer = studentnummer;
    }
}
```

Wat nog steeds niet werkt omdat er nog steeds geen lege **Persoon** constructor is!

Wat observeren we?

- Java voegt een aanroep naar **super()** toe in je constructoren!
- Tenzij je zelf een aanroep naar een constructor van de super-klasse toevoegt!

De oplossing!

```
class Student extends Persoon {
    int studentnummer;

    Student(String naam, int studentnummer) {
        super(naam);
        this.studentnummer = studentnummer;
    }
}
```

Inhoud

Visibility modifiers

Zoals eerder besproken, er zijn 4 mogelijk **Visibility modifiers**: `public`, `private`, `protected` en `package-private` (geen modifier).

Modifier	Class	Package	Subclass	World
`public`	X	X	X	X
`protected`	X	X	X	-
geen modifier	X	X	-	-
`private`	X	-	-	-

Private variabelen

Private variabelen kunnen dus niet door een subklasse benaderd worden:

```
class Persoon {
    private String naam;
}

class Student extends Persoon {
   int studentnummer;

   public String toString() {
       return naam;
   }
}
```

```
error: naam has private access in Persoon
       return naam;
              ^
1 error
```

Protected

Als je een methode of variabele dus voor de buitenwereld wilt afschermen, maar wel zichtbaar wil maken voor subklassen moet je de variabele dus **protected** maken.

Access modifiers met overriden

Je mag als je een methode override enkel de zichtbaarheid **vergroten**.

Een **public** methode in de superklasse mag je dus niet overriden met een **private** methode in de subklasse.

Public methode private overriden

```
class Persoon {
    protected String naam;
    public String getNaam() { return naam; }
}

class Student extends Persoon {
    int studentnummer;
    private String getNaam() { return naam + " " + studentnummer; }
}
```

```
error: getNaam() in Student cannot override getNaam() in Persoon
    private String getNaam() { return naam + " " + studentnummer; }
                   ^
  attempting to assign weaker access privileges; was public
1 error
```

Inhoud

Overriding en hiding

Bij methoden zagen we dat een nieuwe definitie een bestaande override, onafhankelijk van het type van de referentie.

Product en Armband

```
class Product {
    static int prijs = 300;
    static String getNaam() { return "Product"; }
}

class Armband extends Product {
    static int prijs = 500;
    static String getNaam() { return "Armband"; }
}
```

Static dynamic binding

```
Armband armbandje = new Armband();
System.out.println(armbandje.prijs);
System.out.println(Armband.prijs);
System.out.println(((Product) armbandje).prijs);
System.out.println(Product.prijs);
```

```
500
500
300
300
```

Static methoden

```
Armband armbandje = new Armband();
System.out.println(armbandje.getNaam());
System.out.println(Armband.getNaam());
System.out.println(((Product) armbandje).getNaam());
System.out.println(Product.getNaam());
```

```
Armband
Armband
Product
Product
```

Om te onthouden

Welke variabele of methode zal Java gebruiken?

	Non-static	Static
Variabele	Type van de referentie	Type van de referentie
Methode	Type van het object	Type van de referentie

Inhoud

Overloading

We hebben al gezien dat Java automatisch primitieve datatypen upcast als je een methode hebt welke een `double` als argument nodig heeft en je een `integer` meegeeft als argument.

Wat doet Java bij objecten? Aanschouw onderstaande 2 objecten:

```
class Fruit { }
class Appel extends Fruit { }
```

Wat print onderstaande code?

```
public class College {

    public static void main(String[] args) {
        Appel  a = new Appel();
        Fruit  f = a;
        Object o = f;
        test(a);
        test(f);
        test(o);
    }

    static void test(Object o) {
        System.out.println("Object!");
    }
    static void test(Fruit f) {
        System.out.println("Fruit!");
    }
    static void test(Appel a) {
        System.out.println("Appel!");
    }
}
```

Overloading (II)

```
$ java College
Appel!
Fruit!
Object!
```

In dit geval zal Java de methode aanroepen welke het beste past bij de **referentie**, niet het daadwerkelijke type van het geïnstantieerde object.

En als een object meerdere interfaces implementeert?

Inhoud

Super

De `super`-methode kunnen we dus gebruiken om in een constructor de constructor van de superklasse aan te roepen.

Daarnaast kunnen we `super` ook gebruiken om bij verborgen instantiatievariabelen te komen en overriden methoden.

Super (I)

Even ophalen, wat was de uitvoer van onderstaande code?

```
class A {
    String naam() { return "A"; }
}

class B extends A {
    String naam() { return "B"; }
}
```

```
B b = new B();
A a = b;
System.out.println(a.naam());
System.out.println(b.naam());
```

```
B
B
```

Super (II)

```
class A {
    String naam() { return "A"; }
}

class B extends A {
    String naam() { return super.naam() + "B"; }
}
```

```
B b = new B();
A a = b;
System.out.println(a.naam());
System.out.println(b.naam());
```

```
AB
AB
```

Verborgen variabelen benaderen met super

```
class A {
    int a = 3;
}

class B extends A {
    int a = 4;
    int som() { return super.a + a; }
}
```

```
System.out.println(b.som());
```

```
7
```

Inhoud

Generieke klassen

Zoals we al zagen bij de `Comparable`-interface en `ArrayList`s kun je Java "hinten" wat het type van een object is waarvoor je de klasse gaat gebruiken.

```
ArrayList<Paar> termen = new ArrayList<Paar>();
termen.add(new Paar(1, 2));
```

Bovenstaande code zou anders dus alleen werken als de **ArrayList** klasse een methode heeft:

```
public int add(Paar o) {
    // sla nu het Paar op..
}
```

Maar uiteraard is de ArrayList niet ontworpen voor onze Paar-klasse.

Boekenkast

```
class Boekenkast<T> {
    Object[] planken;

    Boekenkast(int grootte) {
        planken = new Object[grootte];
    }

    void zetOpPlank(int plank, T voorwerp) {
        planken[plank] = voorwerp;
    }

    void printInhoud() {
        for(Object plank : planken) {
            System.out.println(plank != null ? plank : "leeg");
        }
    }
}
```

Boekenkast bouwen

Nu kunnen we op iedere plank van onze boekenkast een spelletje zetten!

```
Boekenkast<Bordspel> spellenkast = new Boekenkast<Bordspel>(5);
spellenkast.zetOpPlank(2, new Bordspel());
spellenkast.zetOpPlank(0, new Monopoly());
spellenkast.zetOpPlank(1, new Stratego());
spellenkast.printInhoud();
```

Dit werkt dus omdat onze boekenkast alleen bordspellen opslaat, en alle onze spellen borspellen zijn!

Beperkingen leggen aan het type van het generieke object

We kunnen nu ook een boekenkast maken waar we allerlei soorten objecten op een plank kunnen zetten:

```
Boekenkast<Object> generiekekast = new Boekenkast<Object>(5);
```

We kunnen de programmeur tegenhouden door af te dwingen dat alle boekenkasten enkel spelletjes mogen bevatten:

```
class Boekenkast<T extends Speelgoed> {
    // ..
}
```

Generics en extends

Met deze definitie kunnen we dus wel een Boekenkast maken van Speelgoed, of Bordspellen:

```
Boekenkast<Bordspel> bordspellenkast = new Boekenkast<Bordspel>(5);
Boekenkast<Speelgoed> speelgoedkast = new Boekenkast<Speelgoed>(5);
```

Maar niet meer van dingen Objecten die geen Speelgoed zijn!

Inhoud

Van de vorige keer

Exceptions:

- Worden gegenereerd door Java (of de programmeur) als er iets mis gaat
- Worden omhoog gegooid totdat ze worden opgevangen
- Worden ze niet opgevangen? Stopt de uitvoer van je programma
- Kun je ook zelf opgooien met `throw`.
- Kun je ook zelf opvangen met een try-catch-statement.
- Moeten verplicht worden opgevangen (checked exceptions) of niet (unchecked exceptions).

Meegeven van een foutmelding aan je exception

```
try
{
    // ergens in de code:
    throw new Exception("foo bar");
}
catch(Exception e)
{
    System.out.println("Daar ging iets mis!");
    System.out.println(e.getMessage());
}
```

Met de `getMessage`-methode van de `Throwable`-klasse kun je de foutmelding uitlezen.

Documentatie:
`https://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html`

Eigen exception definieren

Je kunt ook kinderlijk eenvoudig je eigen exception definieren door de `Exception` klasse uit te breiden:

```
class OngeldigeAfmetingenException extends Exception {

}
```

Vervolgens kun je de Exception opgooien met:

```
throw new OngeldigeAfmetingenException();
```

Foutmelding standaard meegeven

Exceptions hebben standaard een `message` instantiatievariabelen die informatie bevatten over wat er is mis gegaan. Je kunt deze ook automatisch voor iedere instantie van je eigen Exception instellen:

```
class OngeldigeAfmetingenException extends Exception {
    OngeldigeAfmetingenException() {
        super("Ongeldige afmetingen!");
    }
}
```