

2017 samenvatting

Tip: import java.util.* boven al uw code zetten.

Overerving

Geen overerving: Nadelen

- Dubbele code voor bijna identieke lijsten en classes
- Oplossing? Private ArrayList<Object> rekeningen;
 - Klasse Object = moeder van alle klassen;
 - Alle klassen erven over van object(subklasse), object is de superklasse.

Voor en nadelen overerving

Voordelen

- Geen dubbele code
- Makkelijk verschillende klassen in 1 lijst

Nadelen

- Typecasting is nodig om specifieke methodes op te roepen
- Compiler weet niet welke subklasse het object toebehoort

Oplossing: instanceof. → Bv. If (o instanceof BankRekening){} **Oplossing:** casting. → Bv. String s = (String) o;

Statisch en Dynamisch type van een object

Object o = rekeningen.get(0);

Statische type van o

Het type dat je gebruikt om een variabele te declareren

@Compile time gebruikt om te zien of methode bestaat

Dynamische type van o

Het type dat je gebruikt om een variabele te initialiseren.

Bv. SpaarRekening of BankRekening

@runtime gebruikt om te checken welke versie van de methode uitgevoerd moet worden

Dynamische binding

Op het ogenblik van uitvoering wordt de toString() van het dynamisch type opgeroepen

Generalisatie en Specialisatie bij Overerving:

Generalisatie (*van beneden naar boven*)

Gemeenschappelijke instantievariabelen en methoden afzonderlijke klasse.

Dit is de superklasse

Specialisatie (*van boven naar beneden*)

Specifieke instantievariabelen en methode: subklassen

Override

Om toe te voegen aan de toString() van de superklasse: super.toString() + "nieuw"

Bij overridding gaat het om een methode met dezelfde naam, hetzelfde returntype en precies dezelfde argumenten in een superklasse en zijn subklasse, maar andere implementatie

Typecasting voor Equals

```
public boolean equals(Object o){  
    boolean result = false;  
    if (o instanceof Persoon) {  
        Persoon p = (Persoon) o;
```

```
if(this.getNaam().equals(p.getNaam()) && (this.getVoornaam().equals(p.getVoornaam()))){  
    result = true;    }  
}  
return result; }
```

Exceptions

Exception blijven opgooien tot aan de plaats waar de afhandeling kan gebeuren (UI/main).

Een try block kan gevolgd worden door meerdere catch blocks.

Exceptions afhandelen : finally

In een finally block staan statements die uitgevoerd worden nadat de try - catch processing voorbij is (dus ongeacht of er een exception is opgetreden !). In een finally block staat typisch opruim code, zoals bijv. het sluiten van bestanden.

```
try {  
    // code waarin een exception kan optreden  
} catch (ExceptionType e) {  
    // code die exceptionType fouten behandelt  
} finally {  
    // code die altijd uitgevoerd wordt  
}
```

Exception Classes

Unchecked exceptions

RuntimeException en subklassen: fouten die binnen de applicatie liggen, gevolg zijn van programmeerfouten (bugs), niet kunnen geanticipeerd worden door de applicatie, waarvan de applicatie niet kan herstellen, die gefixt moeten worden i.p.v . opvangen, ...

Voorbeelden: NullPointerException, NumberFormatException.

Checked exceptions

Exception en subklassen : fouten die buiten de controle van de developer vallen, geanticipeerd moeten worden door de applicatie, gecheckt worden door de compiler, waarvan de applicatie moet kunnen herstellen Voorbeeld : FileNotFoundException.

Compiler: checkt of deze exception gegooid kan worden en waarschuwt als dit het geval is → developer moet er dus rekening mee houden

Methode moet declareren welke checked exceptions het kan opgooien

Belangrijk: In catch-clausule handel je **altijd** de exception af : of je gooit een **nieuwe fout** op, of je toont een **foutboodschap** aan de gebruiker, maar je stopt de uitvoering van het programma!

Eigen Exception class:

```
class EigenException extends RuntimeException{
    public EigenException(String message){
        super(message);
    }
}
```

Soorten exceptions

Java Unchecked RuntimeException.

- **ArithmeticException** Arithmetic error, such as divide-by-zero.
- **ArrayIndexOutOfBoundsException** Array index is out-of-bounds.
- **ClassCastException** Invalid cast.
- **IllegalArgumentException** Illegal argument used to invoke a method.
- **IndexOutOfBoundsException** Some type of index is out-of-bounds.
- **NullPointerException** Invalid use of a null reference.
- **NumberFormatException** Invalid conversion of a string to a numeric

Checked Exceptions Defined in java.lang.

- **ClassNotFoundException** Class not found.

- **CloneNotSupportedException** No clone object not implementing Cloneable
- **IllegalAccessException** Access to a class is denied.
- **InstantiationException** No create object of abstract class /interface
- **NoSuchFieldException** A requested field does not exist.
- **NoSuchMethodException** A requested method does not exist.

Abstracte klassen en interfaces

Abstracte klassen of Polymorfie

Abstracte methodes enkel in abstracte klassen. Geen objecten maken van abstracte klassen. Elke subklasse **moet** alle abstracte methodes implementeren (tenzij zelf abstract)

toString() en equals():

Methodes geërfd van Object

Polymorfie (veelvormigheid)

Het geheel van overerving en dynamische binding, waarbij je een referentie hebt die als statisch type een superklasse heeft en als dynamisch type een subklasse .

Interfaces

- New → Interface

Alle methodes abstract (keyword abstract is niet nodig) en alle methodes public (keyword public is niet nodig)

Contractueel: klassen die interface implementeren moeten implementatie hebben voor elke methode, tenzij klasse abstract is.

Cloneable

Wanneer een klasse Cloneable implementeert mag de clone() methode gebruikt worden. Anders CloneNotSupportedException.

```
@Override
public Fiets clone() throws CloneNotSupportedException{
    return (Fiets)super.clone();
}
```

Serializable

Wanneer een klasse Serializable implementeert, kunnen de objecten geserialiseerd worden

Transient: Zorgt ervoor dat een instantievariabele niet geserialiseerd wordt, krijgt bij deserialiseren default-waarde. Static nooit geserialiseerd.

```
public class Database {
    static final long serialVersionUID = 3147609931653727026L;
    public void addWiel(Wiel wiel) throws Exception{
        FileOutputStream fs = new FileOutputStream("Wiel.txt");
        ObjectOutputStream os = new ObjectOutputStream(fs);
        os.writeObject(wiel);
        os.close();
    }
    public Wiel getWiel() throws Exception {
        FileInputStream fs = new FileInputStream("Wiel.txt");
        ObjectInputStream os = new ObjectInputStream(fs);
        Wiel wiel = (Wiel) os.readObject();
        os.close();
        return wiel;
    }
}
```

```
}
```

Abstracte klassen	Interfaces
Interfaces	Alleen public static final velden
Constructors	Geen constructor

Sommige methodes abstract	Alle methodes abstract
Klasse kan maar van 1 klasse extenden	Klasse kan ++ interfaces implementeren
Public, private of protected	Alles public

Writing and reading

Write

File namen = new File("Namen.txt");

```
File namen = new File("Namen.txt");
try{
    PrintWriter writer = new PrintWriter(namen);
    writer.println("Mieke Kemme");
    writer.println("Elke Steegmans");
    writer.close();
}catch(FileNotFoundException ex) {
    throw new DomainException("Fout bij het wegschrijven", ex);
}
```

Read

```
ArrayList<Persoon> personen = new ArrayList<>(); // alle info van bestand komt hier
File personenFile = new File("Personen.txt");

try{
    Scanner scannerFile = new Scanner(personenFile); // scanner voor File
    while (scannerFile.hasNextLine()) { // voor elke lijn van het bestand
        Scanner scannerLijn = new Scanner(scannerFile.nextLine());
        // scanner voor lijn
        scannerLijn.useDelimiter(" / "); // scheidingstekens van verschillende delen in de huidige lijn
        String voornaam = scannerLijn.next(); // eerste deel huidige lijn tot aan /
        String naam = scannerLijn.next(); // tweede deel huidige lijn tot aan /
        Persoon persoon = new Persoon(naam, voornaam);
        personen.add(persoon);
    }
}
```

```
}  
}catch(FileNotFoundException ex) {  
    throw new DomainException("Fout bij het inlezen", ex);  
}
```

Collections

Gebruik zoveel mogelijk interface of superklasse als statisch type.

Collections.png

Soorten

Lists

get(i) is een methode die enkel van toepassing is op Lists. Dubbels mogen, volgorde van ingave telt, heeft indices

ArrayList

Je moet veel minder code zelf schrijven wanneer je ArrayList gebruikt dan wanneer je een Array gebruikt! default, meestal OK

LinkedList

Kies dit indien performantie belangrijk. LinkedList houdt pointer naar eerste element bij, elk element houdt pointer naar volgend element bij; elementen staan dus niet meer fysisch na elkaar zoals bij ArrayList. Implementeert potentieel Queue of Deque = double ended queue

Sets

Geen dubbels bij sets, ook geen duidelijke volgorde

HashSet

Implementeert Hashmap. default, meestal OK

LinkedHashSet

Volgorde van ingave, werkt met interne LinkedHashMap

Maps

Verzameling van key-value paren waarbij er maar 1 paar is met specifieke waarde voor key (volgens equals in klasse K) Maakt het mogelijk zeer snel informatie (value) op te zoeken op basis van een key Key is object van klasse K, value is object van klasse V Key-value paar wordt ook entry genoemd. Geen duidelijke volgorde

HashMap

Bij HashMap is de indexkey de hashCode

In array komen alle entries waarvan de key door de hashCode wordt afgebeeld op dezelfde waarde terecht in een LinkedList

LinkedHashMap

Elke entry houdt wijzer bij naar volgende

TreeMap

Keys worden ascending gesorteerd (bv. a→z) bijgehouden in een boomstructuur.

Hashen

```
@Override
public int hashCode() {
    Objects.hash(naam, voornaam, rnummer);
}
```

Overzicht

List	Map	Set
Dubbels Volgorde (van ingave) Index gebruiken	Key-value Key uniek Geen duidelijke volgorde	Geen dubbels Geen duidelijke volgorde
ArrayList , LinkedList,...	HashMap , LinkerHashMap,...	HashSet , LinkedHashSet,...
►List<String> list = new ArrayList<>(); ►list.add("Bert") ►list.get(0) ►list.remove(0)	►Map<String,String> map = new HashMap<>(); ►map.put("r0047001", "Bert") ►map.get("r0047001") ►map.remove("r0047001")	►Set<String>set = new HashSet<>(); ►set.add("Bert") ►set.get("Bert") ►set.remove("Bert")

Ik wil...	Collectie
-----------	-----------

...studenten snel kunnen terugvinden op basis van hun studentnummer.	Map
...foutboodschappen in een applicatie loggen	List
...bijhouden welke boeken je al gelezen hebt	Set
...deelnemers inschrijven voor kamp, aantal plaatsen beperkt is	List? Set?

Comparable, Comparator, Sorting

Comparable

Comparable is een interface. Je hebt deze bijvoorbeeld nodig als je een List wilt sorteren (`Collections.sort(studentenlijst)`). Dit is voor de default sorteervolgorde. Als je deze implementeert (implements `Comparable<Student>`), moet je een `compareTo(Student O)` methode implementeren.

```
public int compareTo(Student O)
```

Indien beide objecten gelijkwaardig zijn, moet een **0** gereturned worden. Als het this-object kleiner is dan Student O, dan return je een **negatief getal**. Als het this-object groter is dan Student O, dan return je een **positief getal**.

Als je wilt sorteren op basis van een string, bv r-nummer of naam, dan kun je gewoon de reeds geïmplementeerde `compareTo` van string gebruiken:

```
return this.naam.compareTo(O.getNaam())
```

Comparator

Wat als je eigen sorteermethode voor String wilt? Je kunt geen String-klasse gaan maken. Je kunt ook geen nieuwe subklasse van String maken, want String is Final. Comparator is de oplossing! Dit is voor alternatieve sorteervolgorde.

Uitgaande van een `ArrayList<String>` genaamd namen. Java vindt dat Uppercase letters eerst gesorteerd worden, en dan pas Lowercase. Dat willen we niet, dus we gaan dat eens aanpakken.

```
List<String> namen = new ArrayList<String>();
Collections.sort(namen, new Comparator<String>() {
    @Override
    public int compare(String o1, String o2) {
```

```
        return o1.compareToIgnoreCase(o2);  
    }  
});
```

Eclipse geeft dat van die IgnoreCase zelf aan, geen zorgen.

Verwijderen

```
//Creëer lijst van Strings  
List<String> strings = new ArrayList<>();  
strings.add("...");  
  
//Verwijder het woord "De"  
Iterator<String> stringsIterator = strings.iterator();  
while(stringsIterator.hasNext()){  
    String woord = stringsIterator.next();  
    if(woord.equals("De")){  
        stringsIterator.remove();  
    }  
}
```

Static and Final

Static

Statische methode

Keyword static → methode hoort niet bij een object maar bij een klasse. Je moet geen object instantiëren, maar roept de methode rechtstreeks op de klasse.

Een statische methode kan geen instantievariabelen aanspreken.

Statische variabele

Variabele hoort niet bij een object maar bij een klasse (klassevariabele). Dezelfde waarde voor alle objecten van die klasse

Oude bekende

We kennen Static al van de **main-methode**:

```
public static void main(String[] args){  
    System.out.println("Hello world");  
}
```

Deze methode moet statisch zijn, zodat de JVM geen App-object hoeft aan te maken.

System is een gewone klasse uit de package java.lang, **out** is een publieke klassevariable daarvan (PrintStream out). **println** is een objectmethode van die klasse PrintStream.

Gebruik static spaarzaam

Waar wel

- Constanten (zie final...)
- Methodes die objectvelden/-methodes niet nodig hebben
- Ter vervanging van constructoren

Waar niet

- Om eigen klassen uit te breiden (geen Arrays-achtige toestanden zelf aanmaken)
- Niet-final velden (tellen van objecten via een klassevariabele wordt niet gedaan!)

Final

Niet te verwarren met static. Is toepasbaar op velden/lokale variabelen/parameters, methodes en klassen.

Final variable

Mag niet meer veranderd worden. Veldnaam wordt éénmalig geïnitieerd in de constructor. Later wijzigen (e.g. setter) → compilerfout. Dus validatie in de constructor.

```
private final type veldnaam;
```

Final methode en klasse

Final methode mag niet overschreven (override) worden! Wordt zelden gebruikt, private methodes zijn trouwens de facto final.

Final klasse mag geen subklassen krijgen. Zelden gebruikt, voor security-doeleinden.

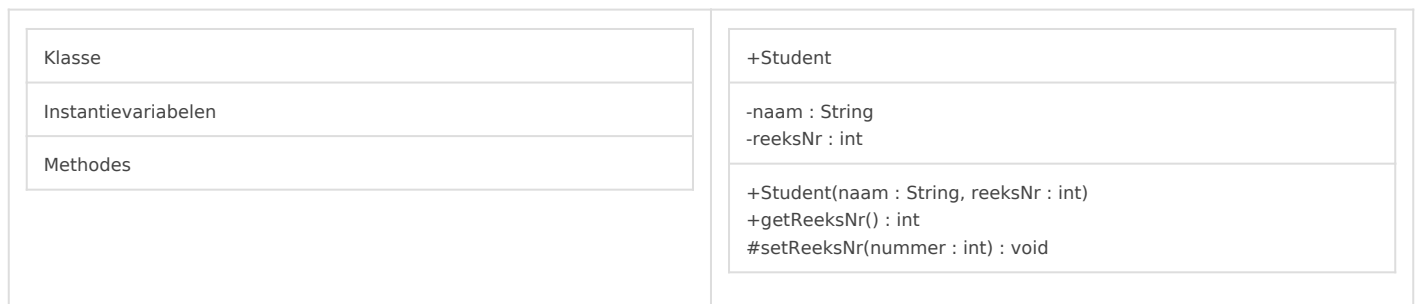
Constante

```
public class Week {  
    public static final int AANTAL_DAGEN = 7;  
}
```

Mag public (mag direct aangesproken worden), **static** (zelfde waarde voor alle objecten), **final** (niet meer aanpasbaar)

Afspraak: alles in hoofdletters, woorden gescheiden met underscore.

UML — Unified Modeling Language



+Public, -Private, #Protected, <<abstract>> & <<interface>> (boven titel en vóór methodes)

Relaties

Heeft een - relatie

Heeft-een.png

Je zet de instantievariabele van iets dat in het klassediagram staat niet in de klasse zelf.

Cardinaliteit kan ook bijvoorbeeld 0..* zijn (nul of meerdere studenten)

Gebruikt een - relatie

Gebruikt een.png

Een klasse zoals UI gebruikt de klasse Student maar heeft geen instantievariabele Student.

Is een - relatie (Overerving)

Is-een.png

Implementeert - relatie (interfaces)

Implementeert.png

Revision #1

Created 17 June 2021 12:30:29 by Jasper G.

Updated 3 December 2021 22:13:09 by Jasper G.