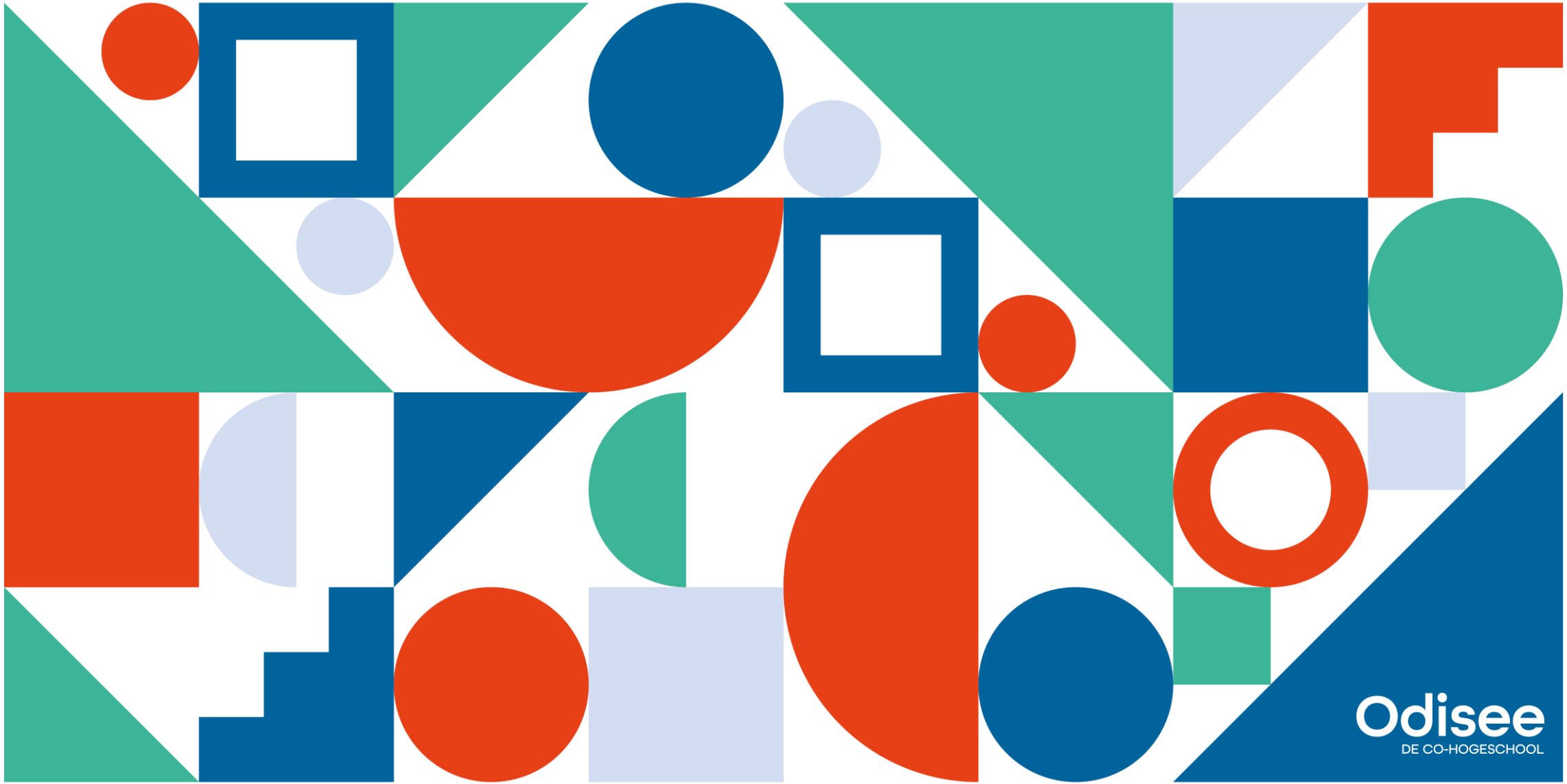

Odisee
DE CO-HOGESCHOOL

Polymorphism - Composition

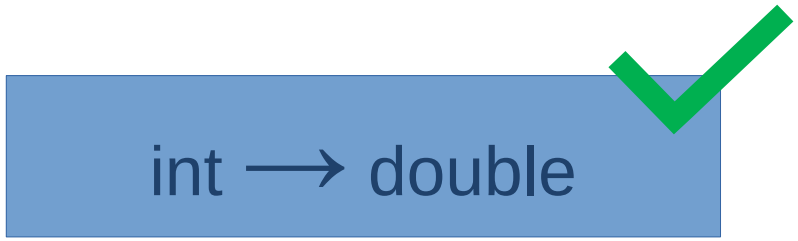
Maarten Troost – Jens Baetens

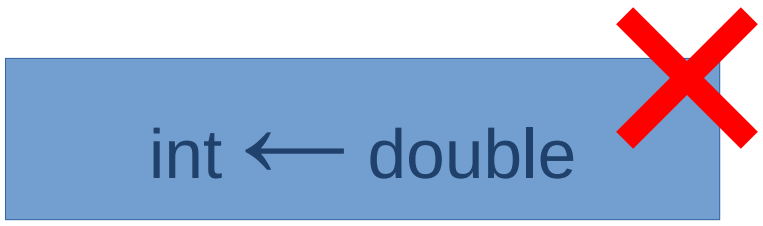
1. Polymorphism

Java == strongly typed

- Elke variabele **moet** een type hebben.
- Elke assignment **moet** van het correcte type zijn.
- Bij verkeerde types kan implicit conversion optreden, als en slechts als dit gegarandeerd zonder verlies van data is.

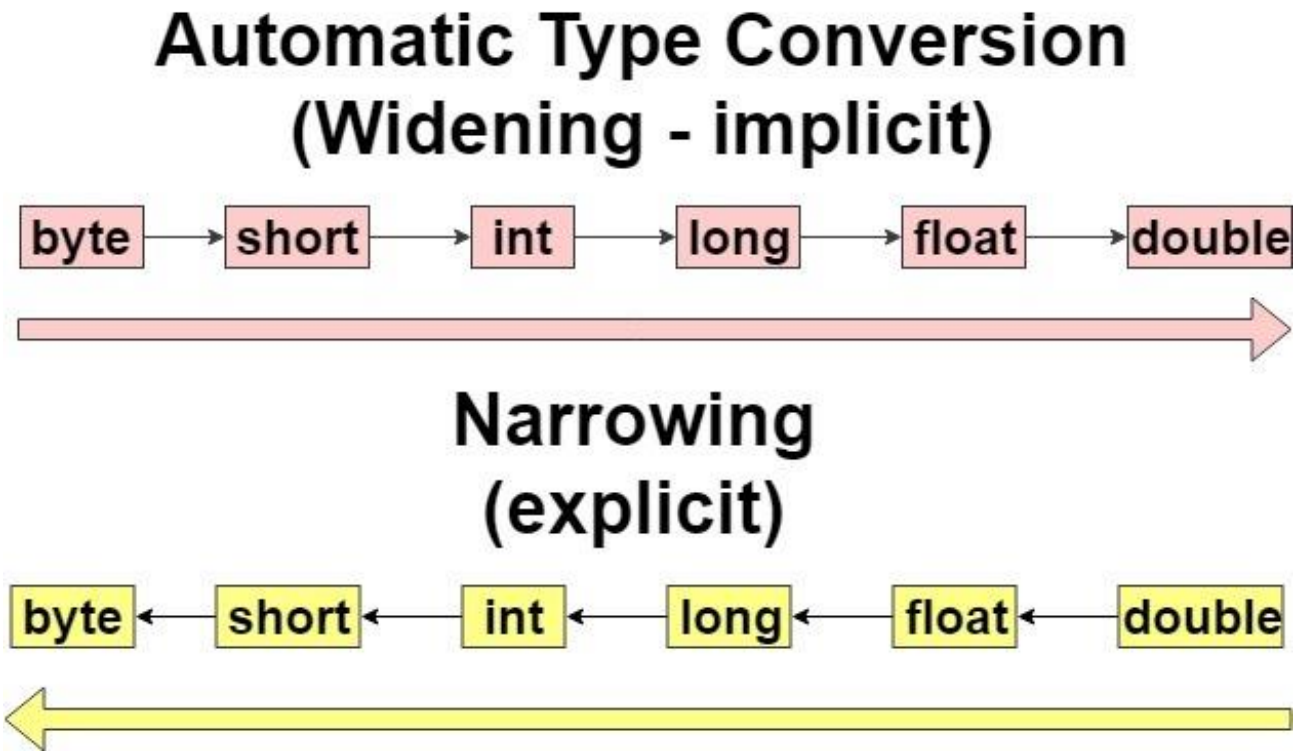
```
int vari = 6;
vari = 666.0; //verkeerde type
double vard = vari; //implicit conversion
```





Implicit conversion bij primitive types

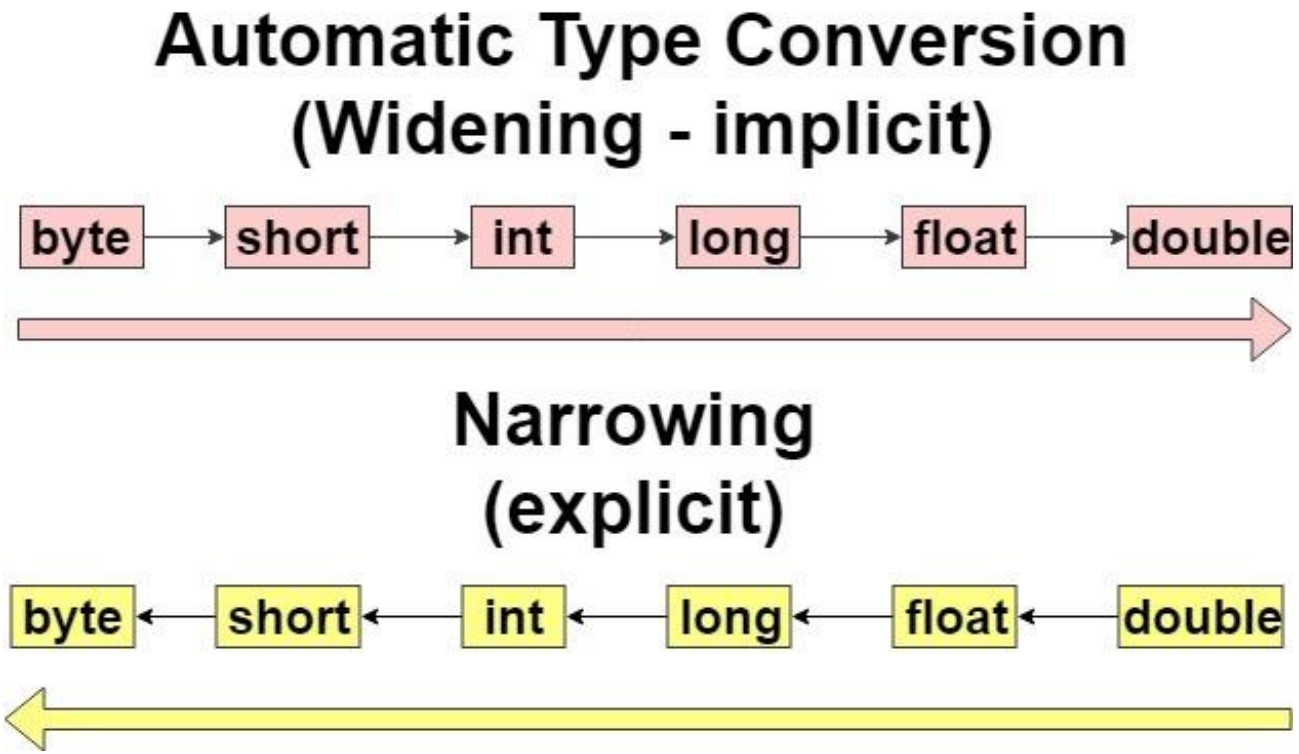
- Om dataverlies tegen te gaan zal implicit conversion enkel van een ‘small’ naar een ‘breder’ type gaan.



Explicit conversion bij primitive types

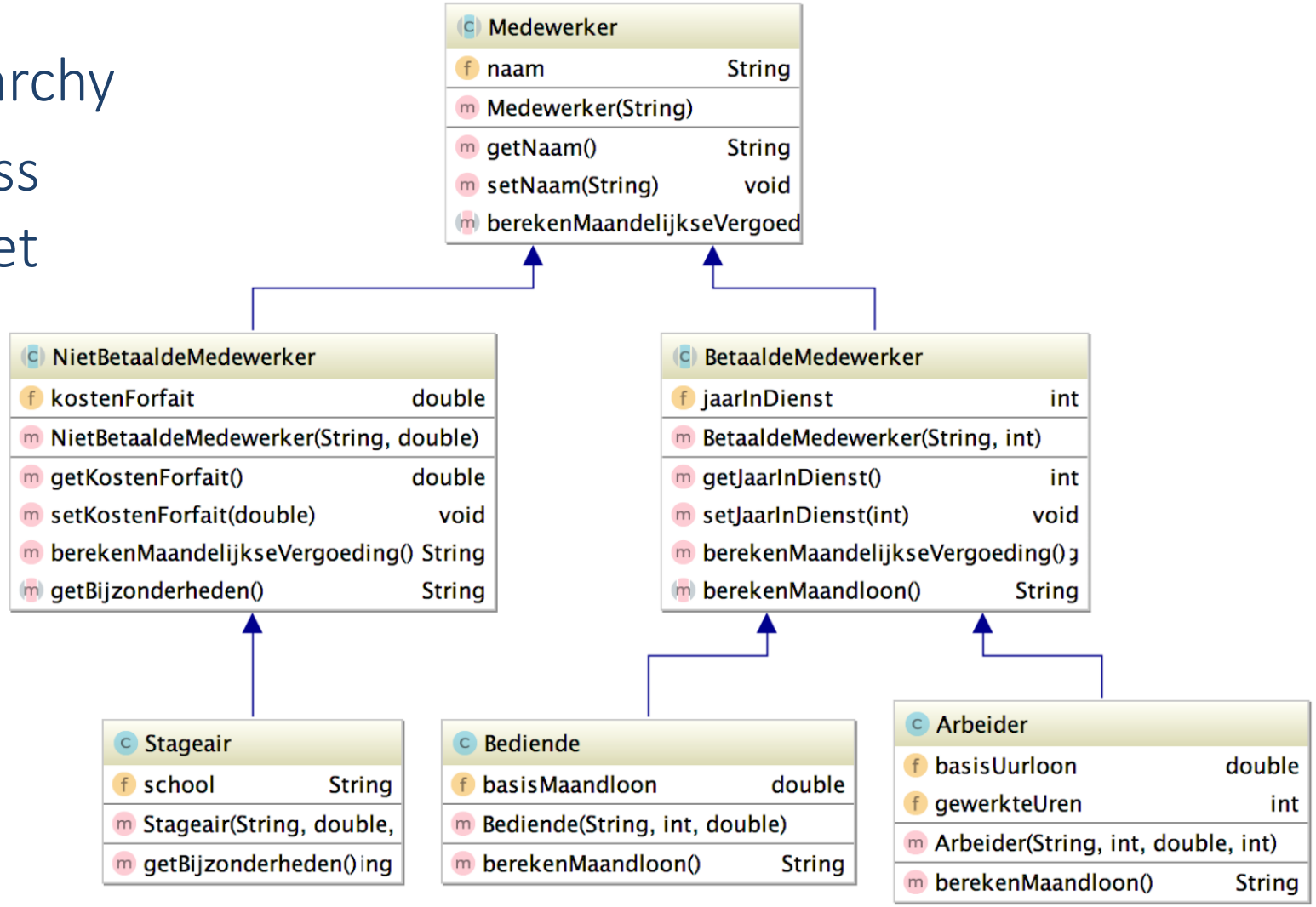
- Narrow kan enkel door expliciet een conversie af te dwingen in je code. Dat heet ‘casting’.

```
vari = (int)666.0;   //casting
```



Conversion bij object types

- Volgt de inheritance hierarchy
- Conversie van derivedClass naar baseClass kan impliciet
- Conversie van baseClass naar derivedClass moet expliciet



Odisee
DE CO-HOGESCHOOL

Implicit conversion bij object types

- Impliciet van derivedClass (Stageair) naar baseClass (Medewerker)

```
Stageair Aaron = new Stageair("Aaron", 0.0,"Odisee");
Medewerker Aaron2 = Aaron;
NietBetaaldeMedewerker Aaron3 = Aaron;
Aaron.getSchool(); //getSchool is method van Stageair
Aaron2.getSchool(); // maar niet van Medewerker
```

- Niet impliciet van baseClass (Medewerker) naar derivedClass (Stageair)

```
Medewerker Bea = new Medewerker("Bea");
NietBetaaldeMedewerker Bea2 = Bea; //kan niet impliciet
```

Explicit conversion bij object types

- Toegelaten door de taal maar kan runtime problem veroorzaken:

```
Medewerker Bea = new Medewerker("Bea");
NietBetaaldeMedewerker Bea2 = (NietBetaaldeMedewerker) Bea;
Bea2.getKostenForfeit();   //dat weten we niet!
```

- Toegelaten en zonder problemen want Carlos is een Stageair

```
Medewerker carlos = new Stageair("Carlos",0.0,"Odisee");
Stageair carlos2 = (Stageair) carlos;
carlos2.getSchool();    //weten we: Odisee
```

Good practices voor conversion

- Implicit conversion mag altijd
- Expliciet conversion (casting) enkel indien je 100% zeker bent dat
  - de data in het primitieve type past in het smaller type
  - het object van deze klasse (of een afgeleide) is

Good practices voor conversion

- Controle van het type at runtime

```
Medewerker carlos = new Stageair("Carlos",0.0,"Odisee");
Medewerker carlos2 = (Medewerker) carlos;
Medewerker dalia = new Medewerker("Dalia");
System.out.println("Geheime identiteiten van Carlos: ");
if (carlos instanceof Stageair) System.out.print("Stageair! ");
if (carlos instanceof NietBetaaldeMedewerker) System.out.print("NietBetaaldeMedewerker! ");
if (carlos instanceof Medewerker) System.out.println("Medewerker!");
…
```

```
C:\Users\maarten.troost\.jdks\openjdk-18.0.1\bin\java.exe
Geheime identiteiten van Carlos:
Stageair! NietBetaaldeMedewerker! Medewerker!
Geheime identiteiten van Carlos2:
Stageair! NietBetaaldeMedewerker! Medewerker!
Geheime identiteiten van Dalia:
Medewerker!

Process finished with exit code 0
```

Oefening conversion

- Toledo > inhoud > week 5 > Enquete Conversion

Wat is polymorfisme?

- Poly = meerdere
- Morf = vorm
- Er zijn verschillende technieken voor polymorfisme en wij bekijken:
  - Overloading
  - Subtyping
- Dezelfde “operatie/requirement/gedrag” kan anders geïmplementeerd worden door andere types.

Wat is overloading?

- Dezelfde method kan anders geïmplementeerd worden voor andere parameter types.

```
public class MyNumber {
    private static String format(int value) {
        return String.format("%d", value);
    }
    private static String format(double value) {
        return String.format("%.3f", value);
    }
    public static void main(String[] args) {
        System.out.println(format(20));
        System.out.println(format(1.2345));
        System.out.println(format((float)15.455554));
    }
}
```

Wat is subtyping?

- Dezelfde “operatie” kan anders geïmplementeerd worden door methods overriden in childclasses.

```
public static void main(String[] args) {
    SoundProducer[] noisyNeighbours = new SoundProducer[]{
      new Human(),
      new Kat(),
      new Hond(),
      new SoundProducer()
    };

    for(SoundProducer soundProducer : noisyNeighbours){
        soundProducer.makeSound();
    }
}
```

```
public class SoundProducer {

    public void makeSound(){
        System.out.println("Some generic sound");
    }

}
```

```
public class Human extends SoundProducer {

    @Override
    public void makeSound(){
        System.out.println("Ik ben een mens");
    }
}
```

Wat is subtyping?

- De juiste method (Human.makeSound of SoundProducer.makeSound) wordt bepaald door welke class het object is.

```
public static void main(String[] args) {
    SoundProducer[] noisyNeighbours = new SoundProducer[]{
      new Human(),
      new Kat(),
      new Hond(),
      new SoundProducer()
    };

    for(SoundProducer soundProducer : noisyNeighbours){
        soundProducer.makeSound();
    }
}
```

```
public class Kat extends SoundProducer {

    @Override
    public void makeSound(){
        System.out.println("Miauw");
    }
}
```

```
Ik ben een mens
Miauw
Woof woof
Some generic sound

Process finished with exit code 0
```

Wanneer gebruiken we polymorfisme?

- Indien de implementatie van de methode afhankelijk is van het type van de parameters/object
- Elke methode met dezelfde naam moet dezelfde operatie implementeren m.a.w. moet hetzelfde idee uitvoeren.
  - ~~Spaarrekening.deposit(int amount)~~ ← stort geld
  - ~~Spaarrekening.deposit(double interestrate)~~ ← voegt interest toe
- Dit is geen taalrestrictie maar een methodiek.

Good practices

- Maak gebruik van casting/impliciete conversion indien mogelijk
- Implementeer bij voorkeur in de parentclass
- Let op voor commutatieve operaties: implementeer ook de commutatieve versie

Compile time vs run time binding

- Compile time polymorfisme of static polymorfisme
- Beslist welke functie gaat uitgevoerd worden bij compilatie
  - Door overloading (verschillend type in parameter van een functie)

```java
public class Addition {

    void sum (int a, int b){
        int result = a+b;
        System.out.println("Addition of two numbers:" + result);
    }

    void sum (int a, int b, int c){
        int result = a+b+c;
        System.out.println("Addition of three numbers:" + result);
    }

    public static void main(String[] args) {
        Addition obj = new Addition();
        obj.sum( a: 10,  b: 7);
        obj.sum( a: 3, b: 9, c: 16);
    }
}
```

```
"C:\Program Files\Java\jdk-18\bin\
Addition of two numbers:17
Addition of three numbers:28
```

Compile time vs run time binding

- Run time polymorfisme of dynamic polymorfisme
- Beslist welke functie gaat uitgevoerd worden at runtime
  - Door overloading (verschillend type in parameter van een functie)

```
class Animal{
    void eat(){
        System.out.println("Animals Eat");
    }
}
class herbivores extends Animal{
    void eat(){
        System.out.println("Herbivores Eat Plants");
    }
}
class omnivores extends Animal{
    void eat(){
        System.out.println("Omnivores Eat Plants and meat");
    }
}
class carnivores extends Animal{
    void eat(){
        System.out.println("Carnivores Eat meat");
    }
}
```

```
class Main{
    public static void main(String args[]){
        Animal A = new Animal();
        Animal h = new herbivores(); //upcasting
        Animal o = new omnivores(); //upcasting
        Animal c = new carnivores(); //upcasting
        A.eat();
        h.eat();
        o.eat();
        c.eat();
    }
}
```

```
"C:\Program Files\Java\jdk-18\bin
Animals Eat
Herbivores Eat Plants
Omnivores Eat Plants and meat
Carnivores Eat meat
```

Oefening overloading

- Open het project tijdspanne_opgave in de klasrepo
- Lees de bestaande code
- Implementeer de Tijdspanne.bijTellen method tot de tests lukken
- Schrijf de lege test in PreciezeTijdspanneTest
- Implementeer de juiste productie code tot de testen slagen
  - Hint: overloading

Oefening overloading oplossing

- Neen! Niet valsspelen! Wat je zelf vindt onthou je 10x beter!

- Okee dan, een hint:
  - Het bijtellen van een PreciezeTijdspanne en een Tijdspanne of het bijtellen van 2 Tijdspannes is identiek
  - Bovenstaande scenarios roepen Tijdspanne.bijTellen(Tijdspanne) op
  - Hierbij negeren we de seconden

Oefening overloading oplossing

- Blijven valsspelen? Tsssssss….

- Nog een hint?
  - Enkel het bijtellen van 2 PreciezeTijdspannes moet seconden behandelen
  - In welke class moet dit staan? Wat is het type van de parameter?

```
class PreciezeTijdspanneTest {

    @Test
    void bijTellen_preciezeTijdspanne2en3_Geeft5() {
        PreciezeTijdspanne t1=new PreciezeTijdspanne( minuten: 2, seconden: 3);
        PreciezeTijdspanne t2=new PreciezeTijdspanne( minuten: 3, seconden: 2);

        t1.bijTellen(t2);
        assertEquals( expected: 5,t1.getMinuten());
        assertEquals( expected: 5,t1.getSeconden());
    }
```

Oefening overloading oplossing

```
public class Tijdspanne {
    protected int minuten;
    …
    public void bijTellen(Tijdspanne andere) {
        this.minuten+=andere.getMinuten();
    }
}
class PreciezeTijdspanne extends Tijdspanne {
    protected int seconden;
    …
    public void bijTellen(PreciezeTijdspanne andere) {
        super.bijTellen(andere);
        this.seconden+=andere.getSeconden();
    }
}
```

2. Composition

Composition and aggregation

- Als objecten van verschillende klasses intens samenwerken (elkaar vaak oproepen) hebben ze vaak referenties naar elkaar. Vb params

```
Person teacher = new Person();
Person Elias = new Person();
teacher.scores(Elias, course: "Software Engineering Fundamentals", score: 16);
```

- Als deze samenwerking onthouden wordt dan worden dit vaak variabelen in de relevante classes welke 2 vormen aannemen: compositie en aggregatie.

```
class Course {
    private Person teacher;
    private List<Person> pupils;
}
```

Composition

- Voorbeeld: een busTraject class bevat een lijst van BusStops waar de bus op een bepaald uur stopt. Die BusStops zijn onlosmakelijk verbonden met het busTraject en kunnen zonder niet bestaan. Als de bus niet bestaat zal er ook niet gestopt worden.

- In een compositie bestaat compositieobject alleen als deel van een bevattend object. Als het bevattende object vernietigd wordt, dan houden ook de compositieobjecten op met bestaan.
- Een compositieobject kan niet zelfstandig bestaan.
- Een compositie wordt ook een “sterke” relatie/associatie genoemd.

Aggregation

- Voorbeeld: Om een machine in een fabriekshal te bedienen zijn er arbeiders nodig. De Machine class onthoudt welke Arbeiders welke machine de bediening hebben. Een machine kan ongebruikt zijn, zonder bediening en een arbeider kan ook bestaan zonder een machine te bedienen.

- In een aggregatie bestaan alle objecten ook onafhankelijk van elkaar. Een object mag maar moet niet in een aggregatie opgenomen zijn
- Een aggregatie wordt ook een “zwakke” relatie/associatie genoemd.

Composition: code

- Een stop (in een halte op een bepaald tijdstip) bestaat niet buiten de context van een traject.

```
public class BusTraject {
    private List<Stop> stops = new ArrayList<>();
    /*Een stop wordt gemaakt en vernietigd door het traject*/
    public void voegStopToe(String halte,LocalTime tijdstip) {
        Stop nieuweStop=new Stop(halte,tijdstip);
        stops.add(nieuweStop);
    }
    public void verwijderStop(String halte) {
        //TODO implement
    }
```

~~public void voegStopToe(Stop nieuweStop) {~~
~~stops.add(nieuweStop);~~
~~}~~

Composition: code

- Het bevattende object (traject) kan het compositieobject (stop) beheren en is niet afhankelijk van andere klassen. De verantwoordelijkheid voor de levensloop van het compositieobject ligt bij het bevattende object.

```
BusTraject myBusTraject=new BusTraject();
myBusTraject.voegStopToe("Bommerskonte",LocalTime.of(11,11,11));
```

Aggregation: code

- Een machine en een arbeider worden niet door elkaar aangemaakt. Ze kunnen een verwijzing naar elkaar bijhouden, meestal door slechts 1 van beide.

```
class Arbeider {
    private String naam;
    public Arbeider(String naam) {
        this.naam = naam;
    }
    public String getNaam() {
        return naam;
    }
    public void setNaam(String naam) {
        this.naam = naam;
    }
}
```

```
public class Machine {
    private Arbeider controleur;
    private Arbeider materiaalLeverancier;
    public Arbeider getControleur() {
        return controleur;
    }
    public void setControleur(Arbeider controleur) {
        this.controleur = controleur;
    }
    public Arbeider getMateriaalLeverancier() {
        return materiaalLeverancier;
    }
    public void setMateriaalLeverancier(Arbeider
materiaalLeverancier) {
```

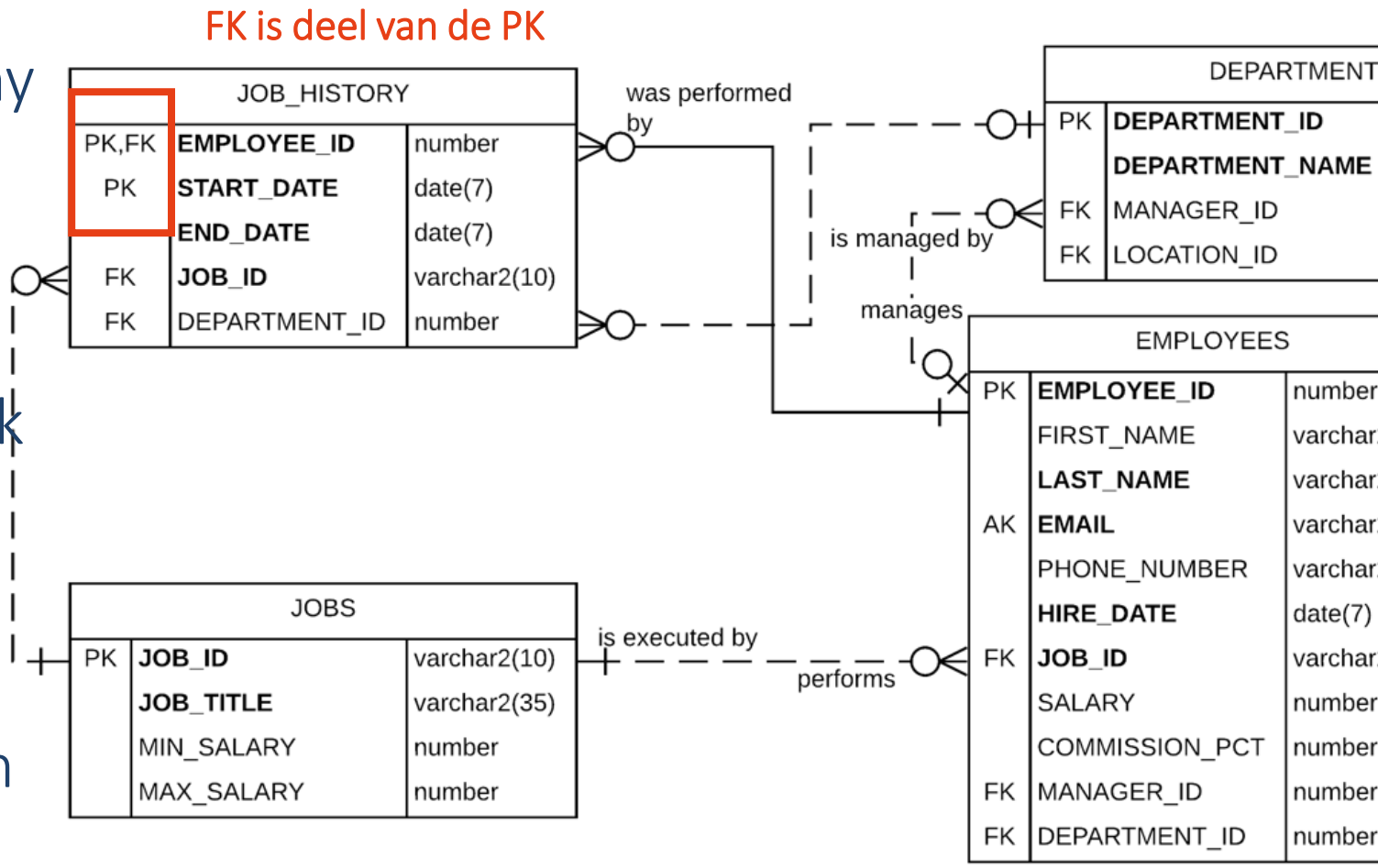
Aggregation: code

- Geen vermenging van verantwoordelijkheden door ongerelateerde klassen.
- Beide kunnen herbruikt worden in andere contexten.

```
Machine snijBot3000=new Machine();
Arbeider Tabita=new Arbeider("Tabita");
Arbeider Zeno=new Arbeider("Zeno");
snijBot3000.setControleur(Tabita);
snijBot3000.setMateriaalLeverancier(Zeno);
Machine afgedankteSnijBot2000=new Machine();
```

Composition: wanneer te gebruiken?

- Join table/intersection table/map table/many-to-many table
- Als er regels zijn welke het aanmaken van de compositieobjecten afhankelijk maakt van de toestand van andere objecten.
- Een compositieobject wordt enkel gebruikt als deel van een groter geheel.



Aggregation: wanneer te gebruiken?

- Als het geen composition is maar wel een relatie tussen de objecten.
- Als de relatie tussen de objecten langere tijd relevant is.

- Vb: aggregatie is hier een slecht idee, want geen langdurige relatie

```
class SomeIntegerClass {  4 usages
    protected Integer value=0;  2 usages
    private Integer otherTerm;  2 usages
    public void setOtherTerm(Integer otherTerm) { this.otherTerm = otherTerm; }
    public void addition() { value+=otherTerm; }
```

```
SomeIntegerClass myBadInt=new SomeIntegerClass();
myBadInt.setOtherTerm(5);
myBadInt.addition();
myBadInt.setOtherTerm(6);
myBadInt.addition();
```

```
class SomeIntegerClass {  4 usages
    protected Integer value=0;  2 usages
    public void addition(Integer theOtherTerm) {
        value+=theOtherTerm;
    }
```

```
SomeIntegerClass myGoodInt=new SomeIntegerClass();
myGoodInt.addition( theOtherTerm: 5);
myGoodInt.addition( theOtherTerm: 6);
```

Inheritance vs composition

- Inheritance: maak een childclass welke het gewenste gedrag overneemt
- Composition: maak een variabele voor een object met het gewenste gedrag

```
/** A queue implemented by composition of a list */
public class GoodQueue {
   private List elements;
   public String getNextItem() {
      return elements.getItem(0);
   }
   public void add(String item) {
      elements.add(item, elements.getItemCount());
   }
}
```

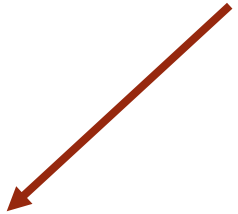
```
/** A queue implemented by inheritance of a list
 * this breaks several features of a List */
public class BadQueue extends List {
   @Override public String getItem(int index) {
      //ignore the index because we may only return the first element
      //this breaks the intent of the getItem(index) method
      return super.getItem(0);
   }
   @Override public synchronized String[] getItems() {
      //only return the list element iso all elements
      String[] ret= {super.getItem(0)};
      return ret;
   }
   @Override public void add(String item, int index) {
      //ignore the index
      super.add(item, getItemCount());
   }
```

Odisee
DE CO-HOGESCHOOL

SOLID – principes van software architectuur

- S – Single Responsibility Principle
- O – Open-closed principle
- L – Liskov Substitution principle
  - A derived class should be substitutable for their base class
  - Resulteert in meer hergebruik van code en een lossere koppeling
- I – Interface Segregation Principle
- D – Dependency inversion principle

Is BadQueue een vervanging voor List? Neen! Het kan niet wat List kan!



Oefening composition vs aggregation vs inheritance

- Bekijk de oplossing van het huistaak over de rekeningen (zie klasrepo)
- De Bank class werkt samen met de Account class. Op welke wijze werken ze samen?
  - Composition?
  - Aggregation?
  - Inheritance?
- Welke andere samenwerking/relatie zou ook mogelijk zijn? Waarom?

Oefening composition vs aggregation vs inheritance

- Maak van de Bank – Account relatie een compositie.
- Gebruik daarvoor de code in de directory “OefBank”.
- Requirements / functionaliteiten:
  - Gebruik composition
  - Account class wijzigt niet
  - Een nieuw account kan geopend worden (bij een bank)
  - Een account kan gesloten worden
  - Geld kan afgehaald worden (withdraw)
- Je mag (private) help methods maken

Huiswerk / Opdracht

In te dienen oefening

- De opdracht staat op github classroom en kan bereikt worden via Toledo > opdrachten > week 5

- Dien de code in door het te pushen naar github
  - eventueel via IntelliJ git integratie
  - of externe git applicatie zoals github desktop

- Vergeet ook het leerverslag niet toe te voegen als **pdf** in de git repository

Project

- Vergeet niet tegen zondag een eerste draft in te dienen van je project!

- Deadline tweede versie is een week verlaat tot 9/6
- Presentatie over het project op 13 of 14 juni
  - Planning volgt nog