Section 8: OOP Fortgeschritten – Komposition, Kapselung, Polymorphismus & Packages

8.1 Komposition (Composition)

Was ist Komposition?

Komposition beschreibt eine "HAT-EIN"-Beziehung (has-a relationship). Eine Klasse enthält **Objekte anderer Klassen** als Felder. Im Gegensatz dazu beschreibt Vererbung eine "IST-EIN"-Beziehung (is-a).

Beziehung	Schlüsselwort	Beispiel
Vererbung (IS-A)	`extends`	Ein Hund **ist ein** Tier
Komposition (HAS-A)	Feld vom Typ einer anderen Klasse	Ein PC **hat einen** Monitor

Praxisbeispiel: PersonalComputer

Ein PC **ist ein** Produkt, aber er **hat** ein Gehäuse, einen Monitor und ein Motherboard:

```
publicclass Product{
private String model;
private String manufacturer;

public Product(String model, String manufacturer){
this.model= model;
this.manufacturer= manufacturer;
}
}

class Monitorextends Product{
privateint size;
private String resolution;

publicvoid drawPixelAt(int x,int y, String color){
        System.out.println(String.format(
"Drawing pixel at %d,%d in color %s", x, y, color));
}
}
```

```
class Motherboardextends Product{
privateint ramSlots;
privateint cardSlots;
private String bios;

publicvoid loadProgram(String programName){
        System.out.println("Program "+ programName+" is now loadin
g...");
}
}

class ComputerCaseextends Product{
private String powerSupply;

publicvoid pressPowerButton(){
        System.out.println("Power button pressed");
}
}
```

PersonalComputer – Komposition + Vererbung:

```
publicclass PersonalComputerextends Product{// IS-A Product

private ComputerCase computerCase;// HAS-A ComputerCase
private Monitor monitor;// HAS-A Monitor
private Motherboard motherboard;// HAS-A Motherboard

public PersonalComputer(String model, String manufacturer,
                            ComputerCase computerCase, Monitor mon
itor,
                            Motherboard motherboard){
super(model, manufacturer);
this.computerCase= computerCase;
this.monitor= monitor;
this.motherboard= motherboard;
}

privatevoid drawLogo(){
        monitor.drawPixelAt(1200,50,"yellow");// Delegiert an Moni
tor
}

publicvoid powerUp(){
```

```
        computerCase.pressPowerButton();// Delegiert an Case
        drawLogo();
}
}
```

Das Prinzip der Delegation

Anstatt dem Benutzer Zugriff auf interne Teile zu geben (mit Gettern), **delegiert** die Klasse Aufgaben an ihre Teile:

```
// SCHLECHT: Interne Teile nach außen geben
thePC.getMonitor().drawPixelAt(10,10,"red");
thePC.getMotherboard().loadProgram("Windows OS");
thePC.getComputerCase().pressPowerButton();

// GUT: Eine einzige Methode, die alles koordiniert
thePC.powerUp();// PersonalComputer delegiert intern an seine Teil
e
```

Warum? Weil der Benutzer nicht wissen muss (und nicht wissen soll), wie der PC intern aufgebaut ist. Das ist **Kapselung**.

Praxisbeispiel: SmartKitchen

```
publicclass SmartKitchen{
private CoffeeMaker brewMaster;// HAS-A CoffeeMaker
private Refrigerator iceBox;// HAS-A Refrigerator
private DishWasher dishWasher;// HAS-A DishWasher

public SmartKitchen(){
        brewMaster=new CoffeeMaker();// Erstellt Teile im Konstruk
tor
        iceBox=new Refrigerator();
        dishWasher=new DishWasher();
}

// Zustand über eine Methode setzen (statt einzelne Getter)
publicvoid setKitchenState(boolean coffeeFlag,boolean fridgeFlag,b
oolean dishWasherFlag){
        brewMaster.setHasWorkToDo(coffeeFlag);
        iceBox.setHasWorkToDo(fridgeFlag);
        dishWasher.setHasWorkToDo(dishWasherFlag);
}
```

```
// Alle Geräte arbeiten lassen
publicvoid doKitchenWork(){
        brewMaster.brewCoffee();
        iceBox.orderFood();
        dishWasher.doDishes();
}
}
```

Verwendung – sauber und einfach:

```
SmartKitchen kitchen=new SmartKitchen();
kitchen.setKitchenState(true,false,true);// Kaffee und Spülmaschin
e an
kitchen.doKitchenWork();// Alles arbeitet
```

Praxisbeispiel: Bedroom (Kompositionsübung)

```
publicclass Bedroom{
private String name;
private Wall wall1, wall2, wall3, wall4;// HAS 4 Walls
private Ceiling ceiling;// HAS-A Ceiling
private Bed bed;// HAS-A Bed
private Lamp lamp;// HAS-A Lamp

publicvoid makeBed(){
        System.out.print("Bedroom -> Making bed |");
        bed.make();// Delegiert an Bed
}
}
```

Komposition vs Vererbung – Wann was?

Frage	Antwort	Verwende
"Ist X ein Y?"	Ja	**Vererbung** (`extends`)
"Hat X ein Y?"	Ja	**Komposition** (Feld)
Kann sich die Beziehung ändern?	Ja	**Komposition** (flexibler)
Willst du Code wiederverwenden?	Ja	Beides möglich

Faustregel: Bevorzuge Komposition gegenüber Vererbung. Komposition ist flexibler und vermeidet die Probleme tiefer Vererbungshierarchien.

8.2 Kapselung (Encapsulation)

Was ist Kapselung?

Kapselung bedeutet, die internen Daten einer Klasse zu **verstecken** und den Zugriff nur über **kontrollierte Methoden** (Getter/Setter) zu erlauben.

Das Problem OHNE Kapselung

```
publicclass Player{
public String fullName;// Jeder kann direkt zugreifen!
publicint health;// Jeder kann direkt ändern!
public String weapon;

publicvoid loseHealth(int damage){
this.health-= damage;
if(this.health<=0){
            System.out.println("Player knocked out");
}
}

publicvoid restoreHealth(int extraHealth){
        health+= extraHealth;
if(health>100){
            health=100;
}
}
}
```

Problem:

```
Player player=new Player();
player.health=200;// Direkt auf 200 setzen – Umgeht restoreHealth
()!
player.health=-500;// Ungültiger Wert – kein Schutz!
```

Die Validierungslogik in `restoreHealth()` wird einfach umgangen, weil `health` `public` ist.

Die Lösung MIT Kapselung

```
publicclass EnhancedPlayer{
private String fullName;// private! Kein direkter Zugriff
privateint healthPercentage;// private!
private String weapon;// private!
```

```
public EnhancedPlayer(String fullName,int healthPercentage, String
weapon){
this.fullName= fullName;
// Validierung im Konstruktor!
if(healthPercentage<=0){
this.healthPercentage=1;
}elseif(healthPercentage>100){
this.healthPercentage=100;
}else{
this.healthPercentage= healthPercentage;
}
this.weapon= weapon;
}

publicint healthRemaining(){
return healthPercentage;
}

// Kein Setter für healthPercentage!
// Health kann nur über loseHealth/restoreHealth geändert werden
}
```

Jetzt:

```
EnhancedPlayer max=new EnhancedPlayer("Max",200,"Sword");
System.out.println(max.healthRemaining());// → 100 (auf 100 begren
zt!)
// max.healthPercentage = 200;  // COMPILE ERROR! private!
```

Die Regeln der Kapselung

1. **Alle Felder** `private` machen
2. **Konstruktoren** mit Validierung verwenden
3. **Getter** für Lesezugriff (wo nötig)
4. **Setter** für Schreibzugriff (nur wo nötig, mit Validierung)
5. **Methoden** für Geschäftslogik (z.B. `deposit()`, `withdraw()` statt `setBalance()`)

Praxisbeispiel: Printer

```
publicclass Printer{
privateint tonerLevel;// private – wird validiert
privateint pagesPrinted;// private – wird nur intern geändert
privateboolean duplex;// private – wird nur im Konstruktor gesetzt

public Printer(int tonerLevel,boolean duplex){
this.pagesPrinted=0;
this.tonerLevel=(tonerLevel>=0&& tonerLevel<=100)? tonerLevel:-1;
this.duplex= duplex;
}

publicint addToner(int tonerAmount){
int tempAmount= tonerAmount+ tonerLevel;
if(tempAmount>100|| tonerAmount<0){
return-1;// Fehler: Zu viel oder negativ
}
        tonerLevel+= tonerAmount;
return tonerLevel;
}

publicint printPages(int pages){
// Duplex: 2 Seiten pro Blatt → weniger physische Seiten
int jobPages=(duplex)?(pages/2)+(pages%2): pages;
        pagesPrinted+= jobPages;
return jobPages;
}

publicint getPagesPrinted(){
return pagesPrinted;// Nur Getter, kein Setter!
}
}
```

Kapselungsvorteile bei Printer:

- `tonerLevel` kann nie über 100 oder unter 0 sein
- `pagesPrinted` kann nie direkt gesetzt werden – nur über `printPages()`
- Duplex-Logik ist intern – der Benutzer ruft einfach `printPages()` auf

---

8.3 Polymorphismus (Polymorphism)

Was ist Polymorphismus?

Polymorphismus (griechisch: "vielgestaltig") bedeutet, dass eine **Variable vom Typ der Elternklasse** ein **Objekt einer Kindklasse** halten kann, und die Methoden des Kindklasse-Objekts aufgerufen werden.

Das Grundprinzip

```
Movie movie=new Adventure("Jaws");// Variable: Movie, Objekt: Adve
nture
movie.watchMovie();// Ruft Adventure.watchMovie() auf!
```

Obwohl die Variable als `Movie` deklariert ist, wird die **überschriebene Methode** von `Adventure` aufgerufen. Java entscheidet zur **Laufzeit** (nicht zur Kompilierzeit), welche Version aufgerufen wird.

Beispiel: Movie-Hierarchie

```
publicclass Movie{
private String title;

public Movie(String title){this.title= title;}

publicvoid watchMovie(){
        String instanceType=this.getClass().getSimpleName();
        System.out.println(title+" is a "+ instanceType+" film");
}

// Factory Method – erstellt den richtigen Subtyp
publicstatic Movie getMovie(String type, String title){
returnswitch(type.toUpperCase().charAt(0)){
case'A' ->new Adventure(title);
case'C' ->new Comedy(title);
case'S' ->new ScienceFiction(title);
default ->new Movie(title);
};
}
}

class Adventureextends Movie{
public Adventure(String title){super(title);}

    @Override
publicvoid watchMovie(){
super.watchMovie();// Erst die Eltern-Version
```

```
        System.out.printf("..%s%n".repeat(3),
"Pleasant Scene","Scary Music","Something Bad Happens");
}
}
```

Polymorphismus in Aktion:

```
Movie movie= Movie.getMovie("A","Jaws");// Gibt ein Adventure-Obje
kt zurück
movie.watchMovie();// Ruft Adventure.watchMovie() auf, obwohl der
Typ Movie ist!
```

Factory Method Pattern

Die statische Methode `getMovie()` ist ein **Factory Method** – sie erstellt Objekte des richtigen Typs basierend auf einem Parameter:

```
publicstatic Movie getMovie(String type, String title){
returnswitch(type.toUpperCase().charAt(0)){
case'A' ->new Adventure(title);
case'C' ->new Comedy(title);
case'S' ->new ScienceFiction(title);
default ->new Movie(title);
};
}
```

Der Aufrufer bekommt ein `Movie`-Objekt zurück und muss nicht wissen, welcher konkrete Subtyp es ist – das ist die **Stärke von Polymorphismus**.

Typ-Casting und `instanceof`

Manchmal muss man wissen, welchen **konkreten Typ** ein Objekt hat:

Explizites Casting (Downcasting)

```
Object comedy= Movie.getMovie("C","Airplane");// Typ: Object
Comedy comedyMovie=(Comedy) comedy;// Downcast zu Comedy
comedyMovie.watchComedy();// Jetzt kann man Comedy-Methoden aufruf
en
```

Vorsicht: Wenn das Objekt **kein** Comedy ist, gibt es eine `ClassCastException`!

`instanceof` – Typprüfung

```
Object unknownObject= Movie.getMovie("S","Star Wars");

if(unknownObjectinstanceof Adventure){
((Adventure) unknownObject).watchAdventure();// Cast + Aufruf
}
```

Pattern Matching mit `instanceof` (Java 16+)

```
if(unknownObjectinstanceof ScienceFiction syfy){
    syfy.watchScienceFiction();// syfy ist automatisch gecastet!
}
```

Das ist kürzer als:

```
if(unknownObjectinstanceof ScienceFiction){
    ScienceFiction syfy=(ScienceFiction) unknownObject;
    syfy.watchScienceFiction();
}
```

`getClass().getSimpleName()` – Klassenname als String

```
String instanceType=this.getClass().getSimpleName();
// → "Adventure", "Comedy", "ScienceFiction", etc.
```

`var` – Lokale Typinferenz (Java 10+)

```
var airplane= Movie.getMovie("C","Airplane");
// Compiler erkennt: airplane ist vom Typ Movie
airplane.watchMovie();// OK: watchMovie ist in Movie definiert
```

`var` erkennt den **deklarierten** Rückgabetyp (`Movie`), nicht den tatsächlichen Typ (`Comedy`).

Polymorphismus-Challenge: Auto-Hierarchie

```
publicclass Car{
publicvoid startEngine(){
        System.out.println("Car -> startEngine()");
}

protectedvoid runEngine(){
```

```
        System.out.println("Car -> runEngine()");
}

publicvoid drive(){
        System.out.println("Car -> driving, type is "+ getClass().
getSimpleName());
        runEngine();// Welche Version wird aufgerufen? → Die des t
atsächlichen Typs!
}
}

class GasPoweredCarextends Car{
privatedouble avgKmPerLitre;
privateint cylinders;

    @Override
publicvoid startEngine(){
        System.out.printf("Gas -> All %d cylinders are firing up,
Ready!%n", cylinders);
}

    @Override
protectedvoid runEngine(){
        System.out.printf("Gas -> usage exceeds the average: %.2f
%n", avgKmPerLitre);
}
}

class ElectricCarextends Car{
    @Override
publicvoid startEngine(){
        System.out.printf("BEV -> switch %d kWh battery on, Ready!
%n", batterySize);
}

    @Override
protectedvoid runEngine(){
        System.out.printf("BEV -> usage under the average: %.2f %
n", avgKmPerCharge);
}
}

class HybridCarextends Car{
```

```
    @Override
publicvoid startEngine(){
        System.out.printf("Hybrid -> %d cylinders are fired up.%
n", cylinders);
        System.out.printf("Hybrid -> switch %d kWh battery on, Rea
dy!%n", batterySize);
}

    @Override
protectedvoid runEngine(){
        System.out.printf("Hybrid -> usage below average: %.2f %
n", avgKmPerLitre);
}
}
```

Polymorphismus in Aktion:

```
publicstaticvoid runRace(Car car){// Akzeptiert JEDEN Car-Subtyp!
    car.startEngine();// Ruft die richtige Version auf
    car.drive();// drive() ruft runEngine() auf → richtige Versio
n!
}

Car ferrari=new GasPoweredCar("Ferrari 296 GTS",15.4,6);
Car tesla=new ElectricCar("Tesla Model 3",568,75);
Car hybrid=new HybridCar("Ferrari SF90",16,8,8);

runRace(ferrari);// Gas-Verhalten
runRace(tesla);// Elektro-Verhalten
runRace(hybrid);// Hybrid-Verhalten
```

Wichtiges Detail: In `Car.drive()` wird `runEngine()` aufgerufen. Obwohl `drive()` in der Klasse `Car` definiert ist, wird die **überschriebene Version** von `runEngine()` im jeweiligen Subtyp aufgerufen. Das ist Polymorphismus!

```
Car.drive() → ruft this.runEngine() auf
                    ↓
       Wenn this ein GasPoweredCar ist → GasPoweredCar.runEngine()
       Wenn this ein ElectricCar ist   → ElectricCar.runEngine()
       Wenn this ein HybridCar ist     → HybridCar.runEngine()
```

8.4 Praxisbeispiel: Bill's Burger Challenge

Dieses Beispiel zeigt Vererbung, Polymorphismus und Komposition in einem größeren Projekt.

Klassen-Hierarchie

```
Item (Basis für alles Bestellbare)
  ├── type, name, price, size
  ├── getAdjustedPrice() – Preis je nach Größe
  │
  └── Burger extends Item
        ├── extra1, extra2, extra3 (Toppings = Item-Objekte)
        ├── addToppings(), getExtrasPrice()
        │
        └── DeluxBurger extends Burger
              ├── deluxe1, deluxe2 (2 zusätzliche Toppings)
              └── getExtrasPrice() → return 0 (Toppings kostenlo
s!)

MealOrder (Komposition)
  ├── HAS-A Burger
  ├── HAS-A Item (drink)
  └── HAS-A Item (side)
```

Kernkonzepte demonstriert:

1. Preisberechnung mit Override (Polymorphismus):

```
// Item: Preis hängt von der Größe ab
publicdouble getAdjustedPrice(){
returnswitch(size){
case"SMALL" -> getBasePrice()-0.5;
case"LARGE" -> getBasePrice()+1.0;
default -> getBasePrice();
};
}

// Burger: Preis = Basis + Toppings
@Override
publicdouble getAdjustedPrice(){
return getBasePrice()+
((extra1==null)?0: extra1.getAdjustedPrice())+
```

```
((extra2==null)?0: extra2.getAdjustedPrice())+
((extra3==null)?0: extra3.getAdjustedPrice());
}

// DeluxBurger: Toppings sind kostenlos
@Override
publicdouble getExtrasPrice(String toppingName){
return0;// Alle Toppings gratis!
}
```

2. `instanceof` mit Pattern Matching in MealOrder:

```
publicdouble getTotalPrice(){
if(burgerinstanceof DeluxBurger){
return burger.getAdjustedPrice();// Deluxe: Alles inklusive
}
return burger.getAdjustedPrice()+ drink.getAdjustedPrice()+ side.g
etAdjustedPrice();
}

publicvoid addBurgerToppings(String e1, String e2, String e3, Stri
ng e4, String e5){
if(burgerinstanceof DeluxBurger db){// Pattern Matching!
        db.addToppings(e1, e2, e3, e4, e5);// 5 Toppings für Delux
e
}else{
        burger.addToppings(e1, e2, e3);// Nur 3 für normal
}
}
```

3. Statische Methode vs Instanz-Methode:

```
// Statische Methode: Braucht kein Objekt
publicstaticvoid printItem(String name,double price){
    System.out.printf("%20s:%6.2f%n", name, price);
}

// Instanz-Methode: Ruft die statische Methode auf
publicvoid printItem(){
    printItem(getName(), getAdjustedPrice());
}
```

8.5 Packages (Pakete)

Was sind Packages?

Packages organisieren Java-Klassen in **Namensräume** (wie Ordner). Sie verhindern Namenskonflikte und ermöglichen Zugriffskontrolle.

Package-Deklaration

```
package com.abc.first;// MUSS die erste Anweisung in der Datei sei
n

publicclass Item{
private String type;

public Item(String type){
this.type= type;
}

    @Override
public String toString(){
return"Item{type='"+ type+"'}";
}
}
```

Import

```
package dev.lpa;

import com.abc.first.*;// Alle Klassen aus dem Package importieren

publicclass Main{
publicstaticvoid main(String[] args){
        Item firstItem=new Item("Burger");// Verwendet com.abc.fir
st.Item
        System.out.println(firstItem);
}
}
```

Package-Namenskonventionen

Konvention	Beispiel	Beschreibung
Reverse Domain	`com.abc.first`	Basierend auf der Firmen-Domain (abc.com)
Projekt-basiert	`dev.lpa`	Für Kurse/eigene Projekte
Firmen-basiert	`at.mci`	Basierend auf der Firmen-Domain (mci.at)

Verzeichnisstruktur = Package-Struktur

```
src/
├── com/abc/first/
│   └── Item.java          (package com.abc.first;)
└── dev/lpa/
    └── Main.java           (package dev.lpa;)
```

Warum Packages?

1. **Namenskonflikte vermeiden**: `com.abc.first.Item` und `dev.lpa.Item` können gleichzeitig existieren
2. **Zugriffskontrolle**: Default-Sichtbarkeit (kein Modifier) gilt nur innerhalb des gleichen Packages
3. **Organisation**: Große Projekte in logische Gruppen aufteilen
4. **Wiederverwendung**: Klassen können in andere Projekte importiert werden

---

8.6 Zusammenfassung der wichtigsten Konzepte

Konzept	Beschreibung
Komposition (HAS-A)	Klasse enthält Objekte anderer Klassen als Felder
Vererbung (IS-A)	Klasse erbt von einer anderen Klasse
Delegation	Klasse leitet Aufrufe an ihre internen Teile weiter
Kapselung	Daten `private`, Zugriff nur über kontrollierte Methoden
Polymorphismus	Variable vom Elterntyp kann Kindobjekt halten; Methode des Kindes wird aufgerufen
Method Overriding	Kindklasse definiert geerbte Methode neu
Factory Method	Statische Methode erstellt Objekte des richtigen Subtyps
Casting (Downcasting)	`(Subtyp) elternVariable` – Zugriff auf Subtyp-Methoden
instanceof	Prüft den tatsächlichen Typ eines Objekts zur Laufzeit
Pattern Matching	`instanceof Subtyp variable` – Cast + Zuweisung in einem Schritt (Java 16+)
var	Lokale Typinferenz – Compiler erkennt den Typ automatisch (Java

Konzept	Beschreibung
	10+)
getClass().getSimpleName()	Gibt den Klassennamen als String zurück
Packages	Namensräume für Klassen mit `package` und `import`
Komposition > Vererbung	Bevorzuge Komposition – flexibler und weniger gekoppelt

Die drei OOP-Konzepte im Vergleich

```
┌──────────────────────┬──────────────────────┬──────────────────────┐
│ KAPSELUNG            │ VERERBUNG            │ POLYMORPHISMUS       │
│                      │                      │                      │
│ private Felder       │ extends              │ Override-Methoden    │
│ Validierung im       │ super()              │ Elterntyp-Variable   │
│ Konstruktor/Setter   │ Code wiederverwenden │ hält Kindobjekt      │
│                      │                      │                      │
│ "Schütze die Daten"  │ "Erweitere die       │ "Gleiches Interface  │
│                      │  Funktionalität"     │  verschiedenes       │
│                      │                      │  Verhalten"          │
└──────────────────────┴──────────────────────┴──────────────────────┘
```

Dieser Konspekt basiert auf dem Code und den Übungen der Section 8 des Java-Kurses.