Section 7: Objektorientierte Programmierung – Klassen & Vererbung

7.1 Was ist Objektorientierte Programmierung (OOP)?

OOP ist ein Programmierparadigma, das auf **Objekten** basiert. Objekte sind Instanzen von **Klassen** und kombinieren Daten (Felder/Attribute) mit Verhalten (Methoden).

Die 4 Grundprinzipien der OOP:

1. **Kapselung (Encapsulation)** – Daten verstecken, Zugriff nur über Methoden
2. **Vererbung (Inheritance)** – Klassen können Eigenschaften von anderen Klassen erben
3. **Polymorphismus (Polymorphism)** – Gleiche Methode, verschiedenes Verhalten (→ Section 8)
4. **Abstraktion (Abstraction)** – Komplexität verbergen, nur das Wesentliche zeigen (→ Section 11)

7.2 Klassen und Objekte

Was ist eine Klasse?

Eine **Klasse** ist ein Bauplan (Blueprint) für Objekte. Sie definiert:

- **Felder (Fields/Attribute)**: Die Daten, die ein Objekt speichert
- **Methoden (Methods)**: Das Verhalten, das ein Objekt hat
- **Konstruktoren**: Spezielle Methoden zum Erstellen von Objekten

Was ist ein Objekt?

Ein **Objekt** (auch **Instanz** genannt) ist eine konkrete Ausprägung einer Klasse, die im Speicher existiert.

```
// Klasse = Bauplan
publicclass Car{ ...}

// Objekt = konkrete Instanz
Car car=new Car();
```

```
Car targa=new Car();
// car und targa sind zwei verschiedene Objekte der gleichen Klass
e
```

Eine einfache Klasse erstellen

```
publicclass Car{
// Felder (Attribute) – beschreiben den Zustand
private String make="Tesla";
private String model="Model X";
private String color="Gray";
privateint doors=2;
privateboolean convertible=true;

// Getter – Wert lesen
public String getMake(){
return make;
}

// Setter – Wert setzen (mit Validierung!)
publicvoid setMake(String make){
if(make==null) make="Unknown";
        String lowercaseMake= make.toLowerCase();
switch(lowercaseMake){
case"holden","porsche","tesla" ->this.make= make;
default ->this.make="Unsupported";
}
}

// Methode – beschreibt Verhalten
publicvoid describeCar(){
        System.out.println(doors+"-Door "+ color+" "+
                make+" "+ model+" "+
(convertible?"Convertible":""));
}
}
```

Objekte erstellen und verwenden

```
Car car=new Car();// Objekt erstellen mit "new"
car.setMake("Porsche");// Setter aufrufen
car.setModel("Carrera");
```

```
car.setColor("black");
car.setDoors(2);
car.setConvertible(true);

System.out.println("make = "+ car.getMake());// Getter aufrufen
car.describeCar();// Methode aufrufen
```

7.3 Das Schlüsselwort `this`

`this` bezieht sich auf das **aktuelle Objekt** – die Instanz, auf der die Methode gerade aufgerufen wird.

Verwendung 1: Unterscheidung zwischen Feld und Parameter

Wenn ein Parameter den gleichen Namen hat wie ein Feld:

```
private String name;

publicvoid setName(String name){
this.name= name;// this.name = Feld, name = Parameter
}
```

Ohne `this` würde Java denken, dass du den Parameter dir selbst zuweist (was nichts bewirkt).

Verwendung 2: Konstruktor-Verkettung mit `this()`

```
public Customer(){
this("Nobody","nobody@email.com");// Ruft den 2-Parameter-Konstruk
tor auf
}

public Customer(String name, String email){
this(name,1000, email);// Ruft den 3-Parameter-Konstruktor auf
}

public Customer(String name,double creditLimit, String email){
this.name= name;// Hier wird die eigentliche Zuweisung gemacht
this.creditLimit= creditLimit;
this.email= email;
}
```

Wichtige Regel: `this()` muss die **erste Anweisung** im Konstruktor sein!

7.4 Getter und Setter

Was sind Getter und Setter?

- **Getter**: Methoden die den Wert eines privaten Feldes **zurückgeben**
- **Setter**: Methoden die den Wert eines privaten Feldes **setzen** (oft mit Validierung)

Namenskonventionen

Typ	Getter	Setter
Normaler Typ	`getFieldName()`	`setFieldName(Type value)`
boolean	`isFieldName()`	`setFieldName(boolean value)`

```
// Für String, int, double etc.:
public String getName(){return name;}
publicvoid setName(String name){this.name= name;}

// Für boolean:
publicboolean isConvertible(){return convertible;}
publicvoid setConvertible(boolean convertible){this.convertible= c
onvertible;}
```

Setter mit Validierung

```
publicvoid setAge(int age){
if(age<0|| age>100){
        age=0;// Ungültige Werte auf 0 setzen
}
this.age= age;
}

publicvoid setMake(String make){
if(make==null) make="Unknown";
    String lowercaseMake= make.toLowerCase();
switch(lowercaseMake){
case"holden","porsche","tesla" ->this.make= make;
default ->this.make="Unsupported";
}
}
```

Warum Getter und Setter?

1. **Datenkontrolle**: Du kannst ungültige Werte abfangen
2. **Kapselung**: Die interne Darstellung kann sich ändern, ohne die Schnittstelle zu brechen
3. **Lesezugriff ohne Schreibzugriff**: Man kann nur einen Getter ohne Setter anbieten (Read-Only)

7.5 Konstruktoren (Constructors)

Was ist ein Konstruktor?

Ein **Konstruktor** ist eine spezielle Methode, die beim Erstellen eines Objekts mit `new` aufgerufen wird. Er hat:

- Den **gleichen Namen** wie die Klasse
- **Keinen Rückgabetyp** (nicht mal `void`!)

Der Standard-Konstruktor (Default Constructor)

Wenn du keinen Konstruktor definierst, erstellt Java automatisch einen **leeren Konstruktor** ohne Parameter. Sobald du **irgendeinen** Konstruktor definierst, wird der Default-Konstruktor **nicht mehr** automatisch erstellt!

Konstruktor ohne Parameter (No-Args Constructor)

```
public Account(){
this("56789",2.50,"Default name","Default email","Default phone");
    System.out.println("Empty constructor called");
}
```

Konstruktor mit Parametern

```
public Account(String number,double balance, String customerName,
               String email, String phone){
    System.out.println("Account constructor with parameters calle
d");
this.number= number;
this.balance= balance;
this.customerName= customerName;
    customerEmail= email;
    customerPhone= phone;
}
```

Konstruktor-Überladung (Constructor Overloading)

Wie bei Methoden kann man mehrere Konstruktoren mit verschiedenen Parametern definieren:

```
publicclass Customer{
private String name;
privatedouble creditLimit;
private String email;

// Konstruktor 1: Keine Parameter → ruft Konstruktor 2 auf
public Customer(){
this("Nobody","nobody@email.com");
}

// Konstruktor 2: Name + Email → ruft Konstruktor 3 auf
public Customer(String name, String email){
this(name,1000, email);
}

// Konstruktor 3: Alle Parameter (der "Master-Konstruktor")
public Customer(String name,double creditLimit, String email){
this.name= name;
this.creditLimit= creditLimit;
this.email= email;
}
}
```

Aufruf:

```
Customer c1=new Customer();// → "Nobody", 1000, "nobody@email.com"
Customer c2=new Customer("Joe","joe@email.com");// → "Joe", 1000,
"joe@email.com"
Customer c3=new Customer("Max",5000,"max@mail.at");// → "Max", 500
0, "max@mail.at"
```

Konstruktor-Verkettung (Constructor Chaining)

Best Practice: Alle Konstruktoren leiten an **einen** Hauptkonstruktor weiter, der die eigentliche Initialisierung macht. So wird der Initialisierungscode nicht dupliziert.

```
Customer()  →  this("Nobody", "nobody@email.com")
                          ↓
Customer(name, email)  →  this(name, 1000, email)
```

```
                                        ↓
Customer(name, creditLimit, email)  ←  HIER wird tatsächlich zugew
iesen
```

7.6 Statische Felder und Methoden (static)

`static` Felder

Ein `static` Feld gehört zur **Klasse**, nicht zu einzelnen Objekten. Es gibt nur **eine Kopie** davon, die von allen Instanzen geteilt wird.

```
publicclass Employeeextends Worker{
privatelong employeeId;
privatestaticint employeeNo=1;// Wird von allen Objekten geteilt!

public Employee(String name, String birthDate, String hireDate){
super(name, birthDate);
this.employeeId= Employee.employeeNo++;// Jeder neue Employee beko
mmt eine eindeutige ID
}
}
```

Was passiert:

```
Employee e1=new Employee(...);// employeeId = 1, employeeNo wird 2
Employee e2=new Employee(...);// employeeId = 2, employeeNo wird 3
Employee e3=new Employee(...);// employeeId = 3, employeeNo wird 4
```

`static` Methoden

- Gehören zur Klasse, nicht zu einer Instanz
- Können **ohne Objekt** aufgerufen werden
- Können **nicht** auf nicht-statische Felder zugreifen (weil kein Objekt existiert)
- `main` ist immer `static`!

```
publicstaticvoid main(String[] args){ ...}// Braucht kein Objekt
publicstaticboolean isEvenNumber(int n){ ...}// Hilfsmethode ohne
Zustand
```

7.7 Records (Java 16+)

Was ist ein Record?

Ein **Record** ist eine kompakte Art, eine Datenklasse zu definieren. Java generiert automatisch:

- Private finale Felder
- Einen Konstruktor mit allen Feldern
- Getter-Methoden (ohne "get"-Prefix!)
- `toString()`, `equals()` und `hashCode()`

POJO (Plain Old Java Object) vs Record

POJO – Traditionelle Klasse (viel Boilerplate-Code):

```
publicclass Student{
private String id;
private String name;
private String dateOfBirth;
private String classList;

public Student(String id, String name, String dateOfBirth, String
classList){
this.id= id;
this.name= name;
this.dateOfBirth= dateOfBirth;
this.classList= classList;
}

    @Override
public String toString(){
return"Student{id='"+ id+"', name='"+ name+"', ...}";
}

public String getId(){return id;}
publicvoid setId(String id){this.id= id;}
public String getName(){return name;}
publicvoid setName(String name){this.name= name;}
// ... weitere Getter und Setter
}
```

Record – Einzeilig (Java generiert alles automatisch):

```
public record LPAStudent(String id, String name, String dateOfBirt
h, String classList){
```

```
}
```

Diese eine Zeile generiert automatisch:

- Konstruktor `LPAStudent(String id, String name, String dateOfBirth, String classList)`
- Getter: `id()`, `name()`, `dateOfBirth()`, `classList()` (ohne "get"!)
- `toString()`, `equals()`, `hashCode()`

Unterschiede POJO vs Record

Eigenschaft	POJO (class)	Record
Felder änderbar	Ja (mit Setter)	**Nein (immutable!)**
Getter-Syntax	`getName()`	`name()`
Setter vorhanden	Ja	**Nein**
Boilerplate-Code	Viel	Minimal
Vererbung	Ja (`extends`)	**Nein** (kann keine Klasse erweitern)
toString/equals	Manuell	Automatisch

Verwendung:

```
Student pojoStudent=new Student("S923006","Ann","05/11/1985","Jav
a");
LPAStudent recordStudent=new LPAStudent("S923007","Bill","05/11/19
85","Java");

// POJO: Getter mit "get"
pojoStudent.getName();// → "Ann"
pojoStudent.setClassList("...");// Änderung möglich

// Record: Getter OHNE "get"
recordStudent.name();// → "Bill"
// recordStudent.setClassList(...) → FEHLER! Records sind immutabl
e!
```

Wann Record, wann POJO?

- **Record**: Wenn die Daten nach der Erstellung **nicht mehr geändert** werden sollen (Datenträger, DTOs)
- **POJO**: Wenn die Felder **veränderbar** sein müssen

7.8 Vererbung (Inheritance)

Was ist Vererbung?

Vererbung ermöglicht es einer Klasse (Kindklasse/Subclass), alle Felder und Methoden einer anderen Klasse (Elternklasse/Superclass) zu **erben** und zu erweitern.

```
publicclass Animal{// Elternklasse (Superclass)
protected String type;
private String size;
privatedouble weight;

publicvoid move(String speed){
        System.out.println(type+" moves "+ speed);
}

publicvoid makeNoise(){
        System.out.println(type+" makes some kind of noise");
}
}

publicclass Dogextends Animal{// Kindklasse (Subclass) erbt von An
imal
private String earShape;
private String tailShape;
// Dog hat automatisch: type, size, weight, move(), makeNoise()
// PLUS eigene Felder: earShape, tailShape
}

publicclass Fishextends Animal{// Eine andere Kindklasse
privateint gills;
privateint fins;
}
```

Das Schlüsselwort `extends`

```
publicclass Dogextends Animal{ ...}
```

- `Dog` **erbt** alle `public` und `protected` Felder und Methoden von `Animal`
- `Dog` kann **eigene** Felder und Methoden hinzufügen
- `Dog` kann geerbte Methoden **überschreiben** (Override)
- Java unterstützt nur **Einfachvererbung** (eine Klasse kann nur von EINER Klasse erben)

`super` – Zugriff auf die Elternklasse

`super()` im Konstruktor

```
publicclass Dogextends Animal{

public Dog(){
super("Mutt","Big",50);// Ruft den Konstruktor von Animal auf
}

public Dog(String type,double weight, String earShape, String tail
Shape){
super(type,// type
              weight<15?"small":(weight<35?"medium":"large"),// si
ze berechnen
              weight);// weight
this.earShape= earShape;
this.tailShape= tailShape;
}
}
```

Wichtig: `super()` muss die **erste Anweisung** im Konstruktor sein!

`super.methode()` – Elternmethode aufrufen

```
@Override
publicvoid move(String speed){
super.move(speed);// Zuerst die Elternversion aufrufen
// Dann eigenes Verhalten hinzufügen:
if(speed=="slow"){
        walk();
        wagTail();
}else{
        run();
        bark();
}
}
```

`super.toString()` – toString der Elternklasse einbinden

```
@Override
public String toString(){
return"Dog{earShape='"+ earShape+"', tailShape='"+ tailShape+
```

```
"'} "+super.toString();// Hängt Animal.toString() an
}
```

Zugriffsmodifikatoren und Vererbung

Modifier	Gleiche Klasse	Gleiches Paket	Kindklasse	Überall
`private`	Ja	Nein	Nein	Nein
(default)	Ja	Ja	Nein	Nein
`protected`	Ja	Ja	**Ja**	Nein
`public`	Ja	Ja	Ja	Ja

```
publicclass Animal{
protected String type;// Kindklassen können darauf zugreifen
private String size;// NUR Animal selbst kann darauf zugreifen
}
```

7.9 Methoden überschreiben (Method Overriding)

Was ist Overriding?

Wenn eine Kindklasse eine Methode der Elternklasse **neu definiert**, nennt man das **Überschreiben (Overriding)**.

```
// In Animal:
publicvoid makeNoise(){
    System.out.println(type+" makes some kind of noise");
}

// In Dog (überschreibt makeNoise):
publicvoid makeNoise(){
if(type=="Wolf"){
        System.out.print("Ow Wooooo! ");
}
    bark();
    System.out.println();
}
```

Die `@Override` Annotation

```
@Override
publicvoid move(String speed){
```

```
super.move(speed);
// ... eigenes Verhalten
}
```

`@Override` ist **optional**, aber stark empfohlen:

- Der Compiler prüft, ob die Methode wirklich eine Elternmethode überschreibt
- Verhindert Tippfehler (z.B. `moove` statt `move` → Compile Error!)
- Macht den Code lesbarer

Overriding vs Overloading

Eigenschaft	Overriding (Überschreiben)	Overloading (Überladen)
Wo?	In Kindklasse	In gleicher Klasse
Methodenname	Gleich	Gleich
Parameter	**Gleich**	**Verschieden**
Rückgabetyp	Gleich (oder kovarianter Typ)	Kann verschieden sein
`@Override`	Ja	Nein
Zweck	Verhalten **ändern**	Verschiedene **Varianten** anbieten

---

7.10 Die Object-Klasse

Jede Klasse erbt von Object

In Java erbt **jede** Klasse automatisch von `java.lang.Object`. Auch wenn du kein `extends` schreibst:

```
publicclass Mainextends Object{// "extends Object" ist implizit
    ...
}
```

Wichtige Methoden von Object

Methode	Beschreibung
`toString()`	String-Darstellung des Objekts
`equals(Object o)`	Vergleich zweier Objekte auf Gleichheit
`hashCode()`	Hashwert des Objekts
`getClass()`	Gibt die Klasse des Objekts zurück

`toString()` überschreiben

Standardmäßig gibt `toString()` etwas wie `Student@1a2b3c` aus (Klassenname + Hashcode). Das ist nicht nützlich! Deshalb überschreibt man es:

```
@Override
public String toString(){
return name+" is "+ age;
}
```

Jetzt gibt `System.out.println(student)` etwas Sinnvolles aus, weil `println` automatisch `toString()` aufruft.

Vererbungskette und toString()

```
class Student{
    @Override
public String toString(){
return name+" is "+ age;
}
}

class PrimarySchoolStudentextends Student{
    @Override
public String toString(){
return parentName+"'s kid, "+super.toString();
// → "Carole's kid, Jimmy is 8"
}
}
```

7.11 Vererbungshierarchie – Praxisbeispiel

Die Worker → Employee → SalariedEmployee/HourlyEmployee Hierarchie

```
Worker
  ├── name, birthDate, endDate
  ├── getAge(), collectPay(), terminate()
  │
  └── Employee extends Worker
        ├── employeeId, hireDate
        ├── static employeeNo (auto-increment)
        │
```

```
        ├── SalariedEmployee extends Employee
        │     ├── annualSalary, isRetired
        │     ├── collectPay() → annualSalary / 26
        │     └── retire()
        │
        └── HourlyEmployee extends Employee
              ├── hourlyRate
              ├── collectPay() → 40 * hourlyRate
              └── getDoublePay() → 2 * collectPay()
```

Worker (Basisklasse):

```
publicclass Worker{
private String name;
private String birthDate;
protected String endDate;

publicdouble collectPay(){
return0.0;// Standardimplementierung
}
}
```

Employee (erbt von Worker):

```
publicclass Employeeextends Worker{
privatelong employeeId;
private String hireDate;
privatestaticint employeeNo=1;

public Employee(String name, String birthDate, String hireDate){
super(name, birthDate);// Worker-Konstruktor aufrufen
this.employeeId= Employee.employeeNo++;// Auto-ID
this.hireDate= hireDate;
}
}
```

SalariedEmployee (erbt von Employee):

```
publicclass SalariedEmployeeextends Employee{
double annualSalary;
boolean isRetired;

    @Override
```

```
publicdouble collectPay(){
double paycheck= annualSalary/26;// 26 Gehaltsperioden pro Jahr
return(int)(isRetired? paycheck*0.9: paycheck);// 90% bei Rente
}

publicvoid retire(){
        terminate("12/12/2025");// Methode von Worker aufrufen
        isRetired=true;
}
}
```

HourlyEmployee (erbt von Employee):

```
publicclass HourlyEmployeeextends Employee{
privatedouble hourlyRate;

    @Override
publicdouble collectPay(){
return40* hourlyRate;// 40 Stunden pro Woche
}

publicdouble getDoublePay(){
return2* collectPay();// Feiertags-Bezahlung
}
}
```

7.12 Strings in Java

String ist eine Klasse, kein primitiver Typ

Strings sind **Objekte** der Klasse `java.lang.String`. Sie sind **immutable** (unveränderbar) – jede "Änderung" erstellt ein neues String-Objekt.

String-Formatierung

Escape-Sequenzen

Sequenz	Bedeutung
`\n`	Neue Zeile
`\t`	Tabulator
`\\`	Backslash
`\"`	Anführungszeichen

Sequenz	Bedeutung
`\u2022`	Unicode-Zeichen (z.B. Aufzählungspunkt •)

```
String bulletIt="Print a Bulleted List:\n "+
"\t\u2022 First Point\n"+
"\t\t\u2022 Sub Point";
```

Text Blocks (Java 15+)

```
String textBlock="""
        Print a Bulleted List:
\u2022 First Point
\u2022 Sub Point""";
```

printf – Formatierte Ausgabe

```
System.out.printf("Your age is %d%n", age);
System.out.printf("Age = %d, Birth year = %d%n", age, yearOfBirt
h);
System.out.printf("Your age is %.2f%n",(float) age);// 2 Dezimalst
ellen
System.out.printf("Printing %6d %n", number);// Rechtsbündig, 6 Ze
ichen breit
```

Format	Typ	Beispiel
`%d`	int/long	`42`
`%f`	float/double	`3.140000`
`%.2f`	2 Dezimalstellen	`3.14`
`%s`	String	`"Hello"`
`%c`	char	`'A'`
`%n`	Neue Zeile (plattformunabhängig)	
`%6d`	Mindestens 6 Zeichen breit	`42`

String.format() und .formatted()

```
String result= String.format("Your age is %d", age);
String result2="Your age is %d".formatted(age);// Java 15+
```

Wichtige String-Methoden

Länge und Prüfungen

```
string.length()// Anzahl Zeichen
string.isEmpty()// true wenn length == 0
string.isBlank()// true wenn nur Whitespace (Java 11+)
string.charAt(0)// Erstes Zeichen
string.charAt(length-1)// Letztes Zeichen
```

Suchen

```
helloWorld.indexOf('r')// Position des ersten 'r' → 8
helloWorld.indexOf("World")// Position von "World" → 6
helloWorld.indexOf('l',3)// Erstes 'l' ab Position 3 → 3
helloWorld.lastIndexOf('l')// Position des LETZTEN 'l' → 9
helloWorld.lastIndexOf('l',8)// Letztes 'l' bis Position 8 → 3
```

Vergleichen

```
// NIEMALS == für Strings verwenden! (vergleicht Referenzen, nicht
Inhalte)
string1.equals(string2)// Exakter Vergleich
string1.equalsIgnoreCase(string2)// Groß-/Kleinschreibung ignorier
en
string1.contentEquals(string2)// Wie equals, funktioniert auch mit
CharSequence
string1.startsWith("Hello")// Beginnt mit...
string1.endsWith("World")// Endet mit...
string1.contains("World")// Enthält...
```

Extrahieren

```
birthDate.substring(6)// Ab Position 6 bis Ende → "1982"
birthDate.substring(3,5)// Von Position 3 bis 5 (exklusiv) → "11"
```

Transformieren

```
string.toLowerCase()// → "hello world"
string.toUpperCase()// → "HELLO WORLD"
string.replace('/','-')// Zeichen ersetzen: "25/11" → "25-11"
string.replace("2","00")// String ersetzen: "25" → "005"
string.replaceFirst("/","-")// Nur erstes Vorkommen
```

```
string.replaceAll("/","---")// Alle Vorkommen (mit Regex!)
string.repeat(3)// String wiederholen: "ABC\n" × 3
string.indent(8)// 8 Leerzeichen Einrückung hinzufügen
string.indent(-2)// 2 Leerzeichen Einrückung entfernen
string.trim()// Whitespace am Anfang/Ende entfernen
```

Zusammenfügen

```
String.join("/","25","11","1982")// → "25/11/1982"
"25".concat("/").concat("11")// → "25/11"
"25"+"/"+"11"// → "25/11" (+ Operator)
```

StringBuilder

`String` ist immutable – jede Änderung erstellt ein **neues Objekt**. Bei vielen Änderungen ist `StringBuilder` effizienter:

```
StringBuilder builder=new StringBuilder("Hello World");
builder.append(" and Goodbye");// Anhängen (verändert das GLEICHE
Objekt!)
builder.deleteCharAt(16);// Zeichen löschen
builder.insert(16,'g');// Zeichen einfügen
builder.replace(16,17,"G");// Bereich ersetzen
builder.reverse();// Umkehren
builder.setLength(7);// Auf 7 Zeichen kürzen
```

String vs StringBuilder

Eigenschaft	String	StringBuilder
Veränderbar?	Nein (immutable)	Ja (mutable)
Performance bei vielen Änderungen	Langsam (neue Objekte)	Schnell (gleiches Objekt)
Thread-sicher?	Ja	Nein
Capacity	Keine	Hat eine interne Kapazität
Verwendung	Allgemein	Viele String-Manipulationen

```
// StringBuilder hat eine interne Kapazität:
StringBuilder sb=new StringBuilder();// Standard-Kapazität: 16
StringBuilder sb32=new StringBuilder(32);// Eigene Kapazität: 32
```

```
System.out.println(sb.capacity());// → 16 (oder mehr)
System.out.println(sb.length());// → 0 (noch kein Inhalt)
```

7.13 Klassen als Bausteine – Komposition in Übungen

Mehrere Klassen zusammenarbeiten lassen

In den Übungen wird gezeigt, wie Klassen zusammenarbeiten:

Beispiel: Teppich-Kostenrechner

```
publicclass Floor{
privatedouble width, length;
publicdouble getArea(){return width* length;}
}

publicclass Carpet{
privatedouble cost;
publicdouble getCost(){return cost<0?0: cost;}
}

publicclass Calculator{
private Floor floor;// Hat einen Floor
private Carpet carpet;// Hat einen Carpet

public Calculator(Floor floor, Carpet carpet){
this.floor= floor;
this.carpet= carpet;
}

publicdouble getTotalCost(){
return floor.getArea()* carpet.getCost();
}
}
```

Verwendung:

```
Carpet carpet=new Carpet(3.5);
Floor floor=new Floor(2.75,4.0);
Calculator calc=new Calculator(floor, carpet);
System.out.println("total= "+ calc.getTotalCost());// → 38.5
```

Vererbung in Übungen: Circle → Cylinder

```
publicclass Circle{
privatedouble radius;

public Circle(double radius){
this.radius= radius<0?0: radius;
}

publicdouble getArea(){
return Math.PI* radius* radius;
}
}

publicclass Cylinderextends Circle{
privatedouble height;

public Cylinder(double radius,double height){
super(radius);// Circle-Konstruktor mit Radius
this.height= height<0?0: height;
}

publicdouble getVolume(){
return getArea()* height;// getArea() von Circle verwenden!
}
}
```

Methodenüberladung in Übungen: Point

```
publicclass Point{
privateint x, y;

// Distanz zum Ursprung (0,0)
publicdouble distance(){
return Math.sqrt(Math.pow(x,2)+ Math.pow(y,2));
}

// Distanz zu einem anderen Point-Objekt
publicdouble distance(Point p){
return Math.sqrt(Math.pow(x- p.x,2)+ Math.pow(y- p.y,2));
}
```

```
// Distanz zu gegebenen Koordinaten
publicdouble distance(int x,int y){
return Math.sqrt(Math.pow(this.x- x,2)+ Math.pow(this.y- y,2));
}
}
```

Die `distance`-Methode ist 3-fach überladen – gleicher Name, verschiedene Parameter.

7.14 Nützliche Methoden

`Math.sqrt()` und `Math.pow()`

```
Math.sqrt(25)// → 5.0 (Quadratwurzel)
Math.pow(3,2)// → 9.0 (3 hoch 2)
Math.pow(x,2)// → x² (x zum Quadrat)

// Euklidische Distanz:
Math.sqrt(Math.pow(x1- x2,2)+ Math.pow(y1- y2,2))
```

`Integer.parseInt()` mit `substring()`

```
String birthDate="16/08/1994";
int birthYear= Integer.parseInt(birthDate.substring(6));// "1994"
→ 1994
```

7.15 Zusammenfassung der wichtigsten Konzepte

Konzept	Beschreibung
Klasse	Bauplan für Objekte mit Feldern, Methoden und Konstruktoren
Objekt	Konkrete Instanz einer Klasse, erstellt mit `new`
Felder (Fields)	Variablen innerhalb einer Klasse
this	Referenz auf das aktuelle Objekt
Getter/Setter	Kontrollierter Zugriff auf private Felder
Konstruktor	Spezielle Methode zum Initialisieren von Objekten
Konstruktor-Verkettung	`this()` / `super()` – Konstruktoren rufen sich gegenseitig auf
static	Gehört zur Klasse, nicht zur Instanz
Record	Kompakte, immutable Datenklasse (Java 16+)
Vererbung	`extends` – Kindklasse erbt von Elternklasse

Konzept	Beschreibung
super	Zugriff auf Elternklasse (Konstruktor und Methoden)
@Override	Methode der Elternklasse überschreiben
Object	Basisklasse aller Java-Klassen
toString()	String-Darstellung eines Objekts
String-Methoden	length, indexOf, substring, replace, equals, etc.
StringBuilder	Veränderbare Alternative zu String
protected	Sichtbar in der eigenen Klasse und allen Kindklassen

---

Dieser Konspekt basiert auf dem Code und den Übungen der Section 7 des Java-Kurses.