Lernskript: Objektorientierte Programmierung (OOP) - Die 4 Säulen

Die Objektorientierung ist das Paradigma, auf dem Java aufgebaut ist. Es geht darum, die reale Welt in Form von Objekten mit Zuständen (Attributen) und Verhalten (Methoden) abzubilden. Das Ganze ruht auf vier fundamentalen Säulen, die wir uns jetzt nach unserem "'Classic vs. Modern'-Prinzip" ansehen.

---

1. Kapselung (Encapsulation)

Konzept: Das Bündeln von Daten (Attributen) und den Methoden, die auf diesen Daten arbeiten, in einer einzigen Einheit (einer Klasse). Dabei werden die Daten vor direktem Zugriff von außen geschützt, um die Kontrolle und Integrität zu wahren.

Klassisch (Der "JavaBean"-Weg)

Man deklariert die Attribute als `private` und stellt kontrollierten Zugriff über öffentliche `public` Methoden zur Verfügung: **Getter** (um den Wert zu lesen) und **Setter** (um den Wert zu ändern).

```
// Klassisch
public class Person {
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        // Hier könnte man Logik einbauen, z.B. prüfen, ob der Name nicht leer
ist.
        if (name != null && !name.isEmpty()) {
            this.name = name;
        }
    }
}

Modern (mit Records, seit Java 16)

Für einfache, unveränderliche (immutable) Daten-Trägerklassen gibt es eine extrem
kurze und sichere Schreibweise: Records. Der Compiler generiert für uns
automatisch private final Attribute, einen öffentlichen Konstruktor, Getter (ohne
get-Präfix), equals(), hashCode() und toString(). Dies ist der moderne Weg für
reine Datenklassen.
Generated java

// Modern für unveränderliche Daten
public record PersonRecord(String name, int alter) {}

// Benutzung:
```

```
var person = new PersonRecord("Anna", 30);
System.out.println(person.name()); // Moderner Getter: person.name() statt
person.getName()
// person.name = "Peter"; // Funktioniert nicht, da Records unveränderlich sind!
```

2. Vererbung (Inheritance)

Konzept: Eine Klasse (Subklasse/Kindklasse) kann die Attribute und Methoden einer anderen Klasse (Superklasse/Elternklasse) erben und wiederverwenden. Es entsteht eine "ist-ein"-Beziehung (z.B. ein Auto ist ein Fahrzeug).
Klassisch/Modern

Das Kernkonzept mit dem Schlüsselwort extends ist unverändert. Die Subklasse kann mit @Override Methoden der Superklasse überschreiben und mit dem Schlüsselwort super auf den Konstruktor oder die Methoden der Superklasse zugreifen.
Generated java

```
class Fahrzeug {
    public void hupen() { System.out.println("Hup!"); }
}

class Auto extends Fahrzeug { // Auto erbt von Fahrzeug
    @Override
    public void hupen() { System.out.println("Möp Möp!"); } // Überschreibt die
Methode
}
```

Moderne Einschränkung (mit sealed, seit Java 17)

Klassisch konnte jede öffentliche Klasse von einer (nicht als final markierten) Klasse erben. Moderne sealed Klassen erlauben es einer Superklasse, exakt festzulegen, welche Klassen von ihr erben dürfen. Das macht Vererbungshierarchien sicherer und vorhersagbarer.
Generated java

```
// Nur Auto und LKW dürfen von Fahrzeug erben.
public sealed class Fahrzeug permits Auto, LKW { /* ... */ }
```

3. Polymorphie (Polymorphism)

Konzept: "Vielgestaltigkeit". Es bedeutet, dass ein Objekt viele Formen annehmen

kann. Eine Variable vom Typ der Superklasse kann auf ein Objekt einer ihrer Subklassen verweisen. Beim Methodenaufruf wird dann dynamisch zur Laufzeit entschieden, welche überschriebene Methode (die der Subklasse) ausgeführt wird.
Generated java

```
Fahrzeug meinFahrzeug = new Auto(); // Polymorphie! Variable ist vom Typ Fahrzeug, Objekt ist vom Typ Auto.
meinFahrzeug.hupen(); // Gibt "Möp Möp!" aus, weil die Methode von Auto aufgerufen wird.
```

Moderne Vereinfachung (Pattern Matching für instanceof, seit Java 16)

Klassisch musste man oft prüfen, ob ein Objekt ein bestimmter Typ ist (instanceof), und es dann manuell in diesen Typ umwandeln (casten). Das war umständlich und fehleranfällig.
Generated java

```
// Klassisch
if (meinFahrzeug instanceof Auto) {
    Auto meinAuto = (Auto) meinFahrzeug; // Manueller Cast nötig
    meinAuto.tuerenOeffnen();
}

// Modern
if (meinFahrzeug instanceof Auto meinAuto) { // Check und Zuweisung in einem Schritt!
    meinAuto.tuerenOeffnen(); // meinAuto ist hier direkt verfügbar
}
```

4. Abstraktion (Abstraction)

Konzept: Verstecken der komplexen Implementierungsdetails und das Zeigen einer vereinfachten, wesentlichen Schnittstelle nach außen. Abstraktion wird in Java hauptsächlich durch abstrakte Klassen und Interfaces erreicht.
Abstrakte Klassen

Eine Mischung aus Vorlage und Implementierung. Können nicht selbst instanziiert werden und können sowohl Methoden mit Körper als auch abstrakte Methoden ohne Körper enthalten, die von den Subklassen implementiert werden müssen. Hier hat sich wenig geändert.
Interfaces

Definieren einen reinen "Vertrag", was eine Klasse tun kann.

- Klassisch (vor Java 8): Interfaces durften nur public abstract Methoden und Konstanten enthalten. Sie waren ein reiner, abstrakter Vertrag.

- Modern (seit Java 8): Interfaces können jetzt auch default-Methoden (Methoden

mit einer Standard-Implementierung) und sogar private-Methoden enthalten. Das ist eine riesige Verbesserung, da man Interfaces nun erweitern kann (z.B. eine neue Methode hinzufügen), ohne alle existierenden implementierenden Klassen anpassen zu müssen.