

# Zusammenfassung: Aufgaben zerlegen, Klassen planen & Objektinteraktionen

---

## ◇ 1. Abstraktion

- Abstraktion heisst: Unwichtige Details weglassen, um das Wesentliche zu sehen.
- Fokus auf das Was, nicht das Wie.
- Grundlage jeder objektorientierten Modellierung.

Beispiel:

Beim Modellieren eines Autos ignorierst du Motor-Details und nutzt einfach:

```
auto.fahren();  
auto.bremsen();
```

## ◇ 2. Modularisierung (Teile-und-herrsche)

- Eine grosse Aufgabe wird in kleine, gut verständliche Module zerlegt.
- Jedes Modul wird als eigene **Klasse** implementiert.
- Reduziert Komplexität → **Klassen** können unabhängig entwickelt, getestet und geändert werden.

Beispiel:

Ein Spiel

Spiel nutzt intern **Spieler**, **Karte**, **Deck**, **Punkteverwaltung** usw.

## ◇ 3. Klassen als Datentypen

Ein **Klassenname** ist ein **Typname**.

Variablen können Objektverweise speichern.

```
Auto a;           // a ist eine Variable vom Typ 'Auto'  
a = new Auto(); // Objekterzeugung
```

#### ◇ 4. Klassendiagramme & Objektdiagramme

- **Klassendiagramm** → **statisch**

Zeigt: Klassen, Attribute, Methoden, Beziehungen.

- **Objektdiagramm** → **dynamisch**

Zeigt: Welche Objekte existieren zur Laufzeit und wie sie verbunden sind.

#### ◇ 5. Objektreferenzen

Variablen von Objekttypen speichern Referenzen, nicht das Objekt selbst.

```
Person p1 = new Person("Anna");  
Person p2 = p1;      // p1 und p2 zeigen auf dasselbe Objekt
```

#### ◇ 6. Primitive Typen

- Keine Objekte.
- Haben keine Methoden.
- Typen: `int`, `double`, `boolean`, `char`, `long`, ...

```
int zahl = 5;
```

#### ◇ 7. Objekterzeugung

Objekte werden mit `new` erzeugt.

```
Auto a = new Auto();
```

#### ◇ 8. Überladen (Overloading)

Eine Klasse darf mehrere **Methoden** mit gleichem Namen haben, solange die Parameterlisten unterscheidbar sind.

Beispiel:

```
class Punkt {  
    Punkt(int x, int y) { }  
    Punkt() { }           // überladener Konstruktor  
}
```

## ◇ 9. Methodenaufrufe

### Interner Methodenaufruf

- Eine Methode ruft eine Methode derselben Klasse auf.

```
void starte() {  
    initialisiere(); // interner Aufruf  
}  
  
void initialisiere() { }
```

### Externer Methodenaufruf

- Ein Objekt ruft eine Methode eines anderen Objekts auf (Punkt-Notation).

```
auto.beschleunigen(10); // externer Aufruf
```

## ◇ 10. Debugging

- Ein Debugger zeigt schrittweise, was das Programm macht.
- Hilft, Fehler zu finden:
  - Variablen anschauen
  - Programme pausieren
  - Haltepunkte setzen
  - Objektzustände prüfen

## ■ Komplettes Beispiel: Zerlegen einer Aufgabe in Klassen

Aufgabe: Ein Buchverwaltungssystem

```
class Buch {
    String titel;

    Buch(String t) {
        titel = t;
    }
}

class Regal {
    ArrayList<Buch> buecher = new ArrayList<>();

    void hinzufuegen(Buch b) { // externer Methodenaufruf
        buecher.add(b);
    }

    int anzahl() { // sondierende Methode
        return buecher.size();
    }
}

class Bibliothek {
    Regal regal = new Regal(); // Objektreferenz

    void start() {
        erzeugeBeispielBuecher(); // interner Aufruf
        System.out.println(regal.anzahl() + " Bücher im Regal.");
    }

    void erzeugeBeispielBuecher() {
        regal.hinzufuegen(new Buch("Java lernen"));
    }
}

class Main {
    public static void main(String[] args) {
        new Bibliothek().start();
    }
}
```

Dieses Beispiel zeigt:

- Abstraktion **Buch**, **Regal**, **Bibliothek**
- Modularisierung
- Objekterzeugung
- Objektreferenzen
- interne & externe Methodenaufrufe

### **Merksätze**

**Grosse Probleme** → in kleine **Klassen** zerlegen.

**Klassen** = Baupläne, **Objekte** = konkrete Behälter für Zustand & Verhalten.

**Debugger** hilft, Programme zu verstehen.

**Methoden** können intern oder extern aufgerufen werden.

**Überladen** erlaubt mehrere **Methoden** mit demselben Namen.