Spickzettel Java OOP & UML · Objektorientierte Programmierung

Klassen · Konstruktor · Vererbung · Interface · Listen

Holger Sebastião · www.sebastiao.org
Version 1.0 · 2026-10-25

Klasse = Bauplan

```
// Datei: Tier.java
public class Tier {
  // Attribute: private
  private String name;
  private int alter;

  // Konstruktor
  public Tier(String name, int alter) {
    this.name = name;
    this.alter = alter;
  }

  // Methoden: public
  public String getName() {
    return name;
  }

  public void lautGeben() {
    System.out.println("...");
  }
}
```

Klasse	Bauplan, z. B. `Tier`
Objekt	ein nach dem Bauplan gebautes Exemplar (Instanz), z. B. `bello`
Attribut	Eigenschaft: was ein Objekt hat
Methode	Fähigkeit: was ein Objekt kann

Bei uns: Attribute `private`, Methoden `public`. Eine Klasse pro Datei, Dateiname = Klassenname.

Objekte erzeugen — new

```
Tier bello = new Tier("Bello", 3);
Tier mia = new Tier("Mia", 5);

bello.lautGeben();          // Methode aufrufen
String n = bello.getName(); // "Bello"
```

`new` baut ein neues Objekt, ruft den Konstruktor auf und gibt einen Verweis darauf zurück. Die Variable speichert nur diesen Verweis.

```
Tier t = bello;   // kein neues Objekt!
                  // t und bello zeigen
                  // auf dasselbe Tier
```

Variable ohne Objekt ist `null`. `t.lautGeben()` → `NullPointerException`.

`==` prüft, ob es **dasselbe** Objekt ist. `equals` vergleicht Inhalte nur, wenn die Klasse es überschreibt (String tut das).

Konstruktor

Heißt wie die Klasse, hat **keinen Rückgabetyp** (auch kein `void`), läuft bei `new` und setzt die Startwerte.

STANDARDKONSTRUKTOR

Ohne eigenen Konstruktor erzeugt Java einen leeren: `public Tier() { }`. Attribute sind dann `0`, `false` oder `null`.

Gibt es einen eigenen Konstruktor, fehlt der Standardkonstruktor. `new Tier()` geht dann nur, wenn du ihn selbst schreibst (unten).

EIGENE KONSTRUKTOREN

```
public Tier(String name, int alter) {
  this.name = name;   // Attribut = Parameter
  this.alter = alter;
}

public Tier() {           // ohne Parameter
  this.name = "Unbekannt";
  this.alter = 0;
}
```

`this.name` ist das Attribut des aktuellen Objekts, `name` der Parameter. Mehrere Konstruktoren = überladen.

Getter & Setter — Kapselung

```
public int getAlter() { return alter; }
public void setAlter(int alter) {
  if (alter >= 0) {          // Wert prüfen
    this.alter = alter;
  }
}
```

Kapselung: Attribute `private`, Zugriff nur über Methoden. Der Setter weist falsche Werte ab. Bei boolean: `isFertig()`.

Abstrakte Klasse — abstract

```
public abstract class Form {
  // abstrakte Methode: nur Kopf
  public abstract double flaeche();
}
// Kreis extends Form muss flaeche() schreiben
```

Keine Objekte: `new Form()` geht nicht, sie ist nur Oberklasse. Unterklassen **müssen** die abstrakten Methoden überschreiben. UML: Name *kursiv*.

Anders als ein Interface darf sie Attribute, Konstruktoren und fertige Methoden haben. Man erbt aber nur von **einer** Klasse.

Vererbung — extends

```
public class Hund extends Tier {
  private String rasse;

  public Hund(String name, int alter,
              String rasse) {
    super(name, alter);  // zuerst: Tier bauen
    this.rasse = rasse;
  }

  @Override
  public void lautGeben() {
    System.out.println("Wuff");
  }
}
```

Hund **ist ein** Tier: Er erbt alle Attribute und Methoden und kann eigene dazu haben. `Tier` ist die **Oberklasse**, `Hund` die **Unterklasse**.

`super(…)` ruft den Konstruktor der Oberklasse und muss die erste Zeile sein. `super.lautGeben()` ruft die alte Methode.

Nur **eine** Oberklasse erlaubt. Private Attribute der Oberklasse sind in Hund nur über Getter erreichbar.

Jede Klasse erbt von `Object`, daher gibt es `toString()` und `equals()` überall.

Überladen & Überschreiben

ÜBERLADEN (OVERLOADING)

Gleicher Name, **andere Parameterliste**, meist in derselben Klasse. Nur ein anderer Rückgabetyp reicht nicht.

```
public void fuettern() { … }
public void fuettern(int menge) { … }
```

ÜBERSCHREIBEN (OVERRIDING)

Die Unterklasse schreibt eine geerbte Methode mit **gleichem Kopf** neu. `@Override` lässt den Compiler prüfen, ob es sie oben wirklich gibt.

```
@Override
public String toString() {
  return name + " (" + alter + ")";
}
// println(bello) → Bello (3)
```

POLYMORPHIE

```
Tier t = new Hund("Rex", 2, "Dackel");
t.lautGeben();  // Wuff: Methode von Hund
```

Interface — implements

```
public interface Haustier {
  void streicheln();  // nur Kopf, kein Rumpf
}

public class Hund extends Tier
                  implements Haustier {
  @Override
  public void streicheln() {
    System.out.println("wedelt");
  }
}
```

Ein Interface ist ein **Vertrag**: Es legt fest, *welche* Methoden eine Klasse haben muss, aber nicht *wie*. Die Methoden sind automatisch `public abstract`. Eine Klasse darf **mehrere** Interfaces implementieren: `implements A, B`.

`new Haustier()` geht nicht. Aber: `Haustier h = new Hund(…);`

ArrayList — wachsende Liste

```
import java.util.ArrayList;
ArrayList<String> namen = new ArrayList<>();
namen.add("Ada");          // hinten anfügen
namen.add(0, "Max");       // an Stelle 0
String n = namen.get(0);   // "Max"
namen.set(1, "Eva");       // Stelle 1 ersetzen
namen.remove(0);           // Stelle 0 löschen
int anzahl = namen.size();
namen.contains("Eva")      // true
namen.isEmpty()            // false: nicht leer
namen.clear();             // alles löschen
```

Anzahl	Array: `a.length`	Liste: `liste.size()`
Zugriff	`a[i]`	`liste.get(i)`

Nur Objekttypen: `ArrayList<Integer>` statt `<int>`, ebenso `Double`, `Boolean`.

for-each-Schleife

```
for (Tier t : tiere) {
  t.lautGeben();
}
```

Lies: „für jedes Tier `t` in `tiere`“. Geht mit ArrayList und Arrays, aber ohne Index. Wer `i` braucht, nimmt die normale for-Schleife mit `get(i)`.

Während for-each nichts hinzufügen oder löschen: `ConcurrentModificationException`.

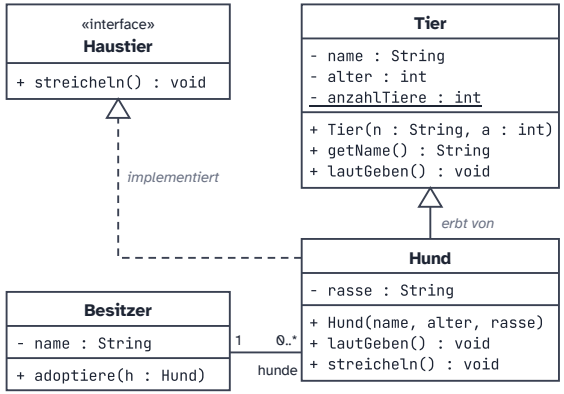
Mit Lambda: `tiere.forEach(t -> t.lautGeben());`

Spickzettel Java OOP & UML · Objektorientierte Programmierung

Klassen · Objekte · Use Case · Sequenz

Holger Sebastião · www.sebastiao.org
Version 1.0 · 2026-10-25

Klassendiagramm — Aufbau und Beziehungen der Klassen



Eine Klasse: drei Fächer

oben	Name, fett. Abstrakt: *kursiv*. Interface: «interface»
Mitte	Attribute: `- name : String`
unten	Methoden: `+ setAlter(alter : int) : void`

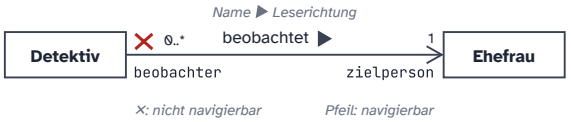
Erst Sichtbarkeit, dann Name, dann **Doppelpunkt und Typ**, also andersherum als in Java. Konstruktor ohne Rückgabetyp. `static` wird unterstrichen.

`+` public `-` private `#` protected `~` Paket

Aus dem Diagramm wird Code

`- name : String`	`private String name;`
`+ getName() : String`	`public String getName()`
Hund —▷ Tier	`class Hund extends Tier`
Hund ┄▷ Haustier	`class Hund implements Haustier`
Besitzer 1 —— 0..* Hund	`private ArrayList<Hund> hunde;`

Eine Assoziation genau lesen



Lies: „Ein Detektiv **beobachtet** genau eine Ehefrau.“ **Pfeil**: Der Detektiv kennt seine Zielperson. ✕: Die Ehefrau darf den Detektiv nicht kennen. **Rolle** = Attributname:

```
private Ehefrau zielperson;  // in Detektiv
// in Ehefrau: kein Attribut für den Detektiv
```

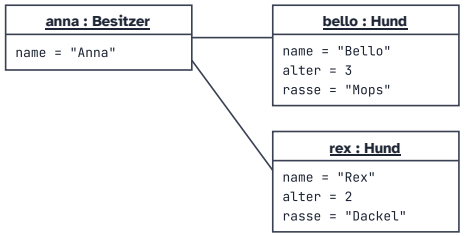
Beziehungen

——▷	**Vererbung** `extends`	„ist ein“. Dreieck zeigt zur Oberklasse.
┄┄▷	**Realisierung** `implements`	Klasse erfüllt ein Interface.
——	**Assoziation**	„kennt“: Attribut vom Typ der anderen Klasse. Pfeil —> = nur in eine Richtung.
◇——	**Aggregation**	„hat“: Teil lebt auch allein (Kurs ◇— Schüler).
◆——	**Komposition**	„besteht aus“: Teil stirbt mit dem Ganzen (Haus ◆— Raum).
┄┄>	**Abhängigkeit**	„benutzt“ kurz, z. B. als Parameter.

Multiplizität: wie viele?

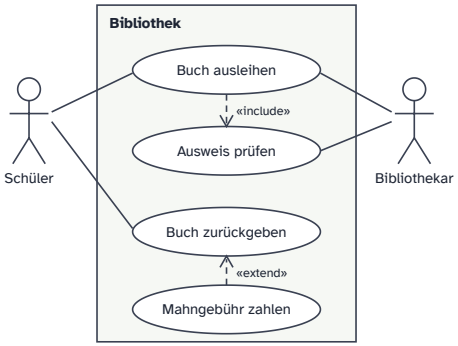
`1`	genau eins	`0..1`	höchstens eins
`*` = `0..*`	beliebig viele	`1..*`	mindestens eins

Objektdiagramm — Momentaufnahme



Zeigt **Objekte** zu einem Zeitpunkt: Kopf `name : Klasse` unterstrichen, darunter Attribute **mit Werten**. Keine Methoden, Verbindungen ohne Multiplizität.

Anwendungsfalldiagramm — Use Case



Was leistet das System für wen? **Akteur**: Person oder System außerhalb. **Anwendungsfall**: Ellipse, Nomen + Verb. Rechteck = Systemgrenze.

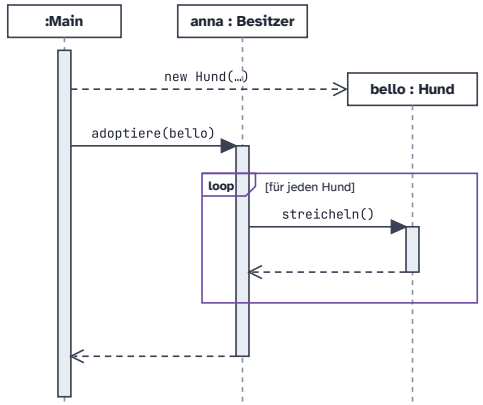
«include»: läuft **immer** mit. **«extend»**: nur **manchmal**, Pfeil zeigt zum Basisfall.

Klassenvariablen — static

```
private static int anzahlTiere = 0;
public Tier(String name, int alter) {
  // … Attribute setzen wie oben
  anzahlTiere = anzahlTiere + 1;
}
public static int getAnzahlTiere() {
  return anzahlTiere;
}
```

`static` gehört zur **Klasse**: `anzahlTiere` gibt es nur einmal für alle Tiere. Aufruf über die Klasse: `Tier.getAnzahlTiere()`. UML: unterstrichen.

Sequenzdiagramm — Ablauf in der Zeit



Zeigt, welche Objekte sich **nacheinander** welche Nachrichten schicken. Die Zeit läuft von oben nach unten.

——▶	**synchrone Nachricht**: normaler Methodenaufruf, der Sender wartet
<┄┄	**Antwort**: Rückkehr, evtl. mit Rückgabewert
——▶	**asynchrone Nachricht**: Sender wartet nicht
┄┄>	**Objekt erzeugen** mit `new`: Pfeil endet am Kopf

Lebenslinie (gestrichelt): solange das Objekt existiert. **Aktivierungsbalken**: das Objekt arbeitet gerade.

`alt`	`[Bedingung] / [else]`: wie if-else
`opt`	`[Bedingung]`: wie if ohne else
`loop`	`[Bedingung]`: Schleife

```
public void adoptiere(Hund h) {
  hunde.add(h);
  for (Hund hund : hunde) {
    hund.streicheln();
  }
}
```

Typische Fehlermeldungen

`cannot be applied to given types`: falsche Parameter bei `new`

`has private access`: Getter oder Setter benutzen

`does not override abstract method`: Interface-Methode fehlt

`NullPointerException`: Objekt nie mit `new` erzeugt

Sichtbarkeit — Zugriffsmodifikatoren

JAVA	UML	KLASSE	PAKET	UNTER-KLASSE	ÜBERALL
`public`	`+`	✓	✓	✓	✓
`protected`	`#`	✓	✓	✓	–
ohne	`~`	✓	✓	–	–
`private`	`-`	✓	–	–	–

protected für Attribute, die Unterklassen direkt brauchen. Achtung: Auch jede Klasse im selben Paket darf zugreifen.

Pakete — package

```
// Datei: tiere/Hund.java
package tiere;          // ganz oben

public class Hund extends Tier { … }
```

```
// in einem anderen Paket
import tiere.Hund;      // eine Klasse
import tiere.*;         // alle aus tiere
```

Ein Paket ist ein Ordner für zusammengehörige Klassen. Paketname = Ordnername, klein geschrieben.

Spickzettel Java OOP & UML · Objektorientierte Programmierung

Holger Sebastião · www.sebastiao.org
Version 1.0 · 2026-10-25

Assoziation im Code — kennt

0..1 ODER 1: EIN ATTRIBUT




```
public class Besitzer {
  private Hund meinHund;     // Rolle = Name

  public void setMeinHund(Hund hund) {
    this.meinHund = hund;
  }
}
```

0..* ODER 1..*: EINE LISTE




```
private ArrayList<Hund> hunde = new ArrayList<>();

public void adoptiere(Hund hund) {
  hunde.add(hund);
}
```

IN BEIDE RICHTUNGEN

Ohne Pfeil kennen sich beide: Jede Klasse bekommt ein Attribut. Beim Verbinden beide Seiten setzen:

```
anna.setMeinHund(bello);
bello.setHalter(anna);
```

Polymorphie mit Listen

```
ArrayList<Tier> tiere = new ArrayList<>();
tiere.add(new Hund("Rex", 2, "Dackel"));
tiere.add(new Katze("Mia", 5));

for (Tier t : tiere) {
  t.lautGeben();   // Wuff, dann Miau
}
```

In eine Liste vom Typ der Oberklasse passen alle Unterklassen. Aufgerufen wird immer die Methode des **echten** Objekts.

```
if (t instanceof Hund) {
  Hund h = (Hund) t;   // erst prüfen, dann casten
}
```

Aggregation — hat, Teil lebt allein




```
public class Kurs {
  private ArrayList<Schueler> teilnehmer
      = new ArrayList<>();

  public void aufnehmen(Schueler s) {
    teilnehmer.add(s);
  }
}

Schueler ada = new Schueler("Ada");
kurs.aufnehmen(ada);
```

Das Teil wird **außen** erzeugt und übergeben. Löscht man den Kurs, gibt es `ada` weiter. Ein Schüler kann in mehreren Kursen sein.

Komposition — besteht aus, Teil stirbt mit




```
public class Haus {
  private ArrayList<Raum> raeume
      = new ArrayList<>();

  public Haus() {
    raeume.add(new Raum("Küche"));
    raeume.add(new Raum("Bad"));
  }
}
```

Das Ganze erzeugt seine Teile **selbst**, meist im Konstruktor. Kein anderes Objekt bekommt sie. Ist das Haus weg, sind die Räume weg. Am Rautenende steht immer `1` (oder `0..1`).

> **Merkhilfe:** Kann das Teil ohne das Ganze sinnvoll weiterleben? Ja = Aggregation ◇, nein = Komposition ◆. Im Zweifel einfache Assoziation.

Beziehungen auf einen Blick

BEZIEHUNG	UML	IN JAVA
Vererbung	—▷	`extends`
Realisierung	┄▷	`implements`
Assoziation	—>	Attribut vom anderen Typ
Aggregation	◇—	Attribut, Teil kommt von außen
Komposition	◆—	Teil mit `new` im Konstruktor
Abhängigkeit	┄>	nur Parameter oder lokale Variable

Exceptions — try-catch

Eine Exception ist ein Fehler **während** das Programm läuft. Ohne `catch` bricht es ab.

```
try {
  // hier kann es schiefgehen
  int zahl = Integer.parseInt(text);
  System.out.println(zahl);
} catch (NumberFormatException e) {
  // läuft nur bei diesem Fehler
  System.out.println("Keine Zahl!");
} finally {
  // läuft immer (optional)
}
```

EINGABE WIEDERHOLEN, BIS SIE PASST

```
import java.util.InputMismatchException;

int alter = 0;
boolean ok = false;
do {
  try {
    alter = eingabe.nextInt();
    ok = true;
  } catch (InputMismatchException e) {
    System.out.println("Bitte eine Zahl!");
    eingabe.nextLine();  // Falsches wegwerfen
  }
} while (!ok);
```

> ! Eine Variable, die im `try`-Block entsteht, gibt es danach nicht mehr. Darum `alter` vorher anlegen.

SELBST EINEN FEHLER MELDEN

```
public void setAlter(int alter) {
  if (alter < 0) {
    throw new IllegalArgumentException(
        "Alter darf nicht negativ sein");
  }
  this.alter = alter;
}
```

`e.getMessage()` liefert den Fehlertext. Mehrere `catch`-Blöcke sind erlaubt, der speziellste zuerst.