# DIE ARCHITEKTUR DER OBJEKTE

Objektorientierte Programmierung in Java meistern (Band 2)

Dietrich Boles

## Impressum

**Autor:** Dietrich Boles

**Kontakt:** dietrich@boles.de

## Vorwort

Liebe Leserinnen und Leser,

willkommen zum zweiten und vielleicht prägendsten Schritt Ihrer Reise in die Welt der Softwareentwicklung!

Wenn Sie dieses Buch aufschlagen, stehen Sie nicht mehr ganz am Anfang. Sie haben das grundlegende Handwerk der Programmierung bereits erlernt. Sie wissen, wie man dem Computer Anweisungen erteilt, den Kontrollfluss mit Schleifen steuert, Entscheidungen mit if-Abfragen trifft und Algorithmen in statischen Methoden formuliert. In unserem ersten Buch haben Sie eine tröstliche, imperative Welt gemeistert, in der Sie der Maschine Schritt für Schritt diktiert haben, was sie tun soll.

Doch wenn Sie sich an das Praxisprojekt am Ende des ersten Buches zurückerinnern – unser kleines Dungeon-Spiel –, dann haben Sie vermutlich gemerkt: Je mehr Code wir schreiben, desto mehr stoßen wir an architektonische Grenzen.

In der rein imperativen Welt sind unsere Daten von unseren Funktionen getrennt. Wir haben Spielfiguren, Monster und Schätze als einfache Arrays oder primitive Variablen modelliert. Diese Daten lagen völlig ungeschützt im Speicher des Computers – als wehrlose Datencontainer. Gleichzeitig hatten wir statische Funktionen, die wild auf diese Daten zugegriffen und sie verändert haben. Jede x-beliebige Methode in unserem Programm konnte theoretisch die Lebenspunkte unserer Spielfigur auf null setzen oder die Position des Monsters auf ungültige Koordinaten verschieben.

Solange ein Programm nur aus hundert Zeilen Code besteht, lässt sich dieses Chaos noch im Kopf beherrschen. Doch was passiert, wenn Sie eine Bankensoftware, ein Steuerungssystem für ein Kraftwerk oder ein modernes Videospiel mit Millionen Zeilen Code entwickeln? In großen, professionellen Programmen ist dieser imperative Ansatz nicht nur unübersichtlich, er ist ein massives Sicherheitsrisiko.

Mit dem Aufschlagen dieses Buches lade ich Sie ein, dieses Problem zu lösen und die Architektur Ihrer Programme radikal zu überdenken.

Um komplexe Systeme bändigen zu können, müssen wir unseren Klassen beibringen, sich selbst zu verteidigen. Wir verlassen die Welt der ungeschützten Variablen und losgelösten Funktionen. Stattdessen werden wir Daten und Verhalten zu echten, intelligenten Einheiten verschmelzen: den *Objekten*.

Die objektorientierte Programmierung (OOP) erlaubt es uns, die reale Welt im Code nachzubilden. Sie werden lernen, wie Objekte Geheimnisse bewahren und ihren inneren Zustand vor fremden Zugriffen schützen (Kapselung). Sie werden sehen, wie Klassen ihr Wissen und ihre

Fähigkeiten an andere Klassen weitergeben können, ohne dass wir Code doppelt schreiben müssen (Vererbung). Und Sie werden Verträge entwerfen, die garantieren, dass unterschiedliche Bausteine Ihres Programms nahtlos und sicher miteinander kommunizieren (Schnittstellen und Polymorphie).

Auf dieser Reise werden Sie Ihre Rolle verändern: Sie sind nun nicht länger nur der Handwerker, der imperative Codezeilen aneinanderreiht. Sie werden zum *Architekten*. Sie entwerfen Blaupausen (Klassen) und erschaffen strukturierte, interagierende Systeme, die auch dann noch wartbar und fehlerfrei bleiben, wenn sie über Jahre hinweg wachsen.

Dieses Buch bildet das entscheidende Bindeglied Ihrer Ausbildung. Es nimmt Sie bei Ihren imperativen Kenntnissen an die Hand und führt Sie tief in das objektorientierte Fundament, auf dem nahezu jede moderne Unternehmenssoftware aufbaut. Erst wenn Sie verstanden haben, wie Objekte Zustand kapseln und interagieren, sind Sie bereit für die großen Herausforderungen der Zukunft: die funktionale Transformation von Daten und die parallele Choreografie von Threads, die in den Folgebänden dieser Reihe auf Sie warten.

Die Umstellung auf das objektorientierte Denken erfordert Übung. Lesen Sie nicht nur den Text, sondern tippen Sie die zahlreichen Code-Beispiele ab. Experimentieren Sie mit den Klassenhierarchien. Bauen Sie eigene, kleine Welten im Speicher Ihres Computers.

Sie haben das Handwerk gelernt. Lassen Sie uns nun lernen, wie man stabile, sichere und elegante Architekturen entwirft – Software, auf die Sie stolz sein können!

Viel Erfolg auf Ihrem Weg zum Software-Architekten!

Dietrich Boles, Oldenburg, März 2026

# Inhaltsverzeichnis

# Teil I: Von Daten zu Objekten (Das neue Fundament)

# 1 Klassen und Objekte I – Datencontainer

Willkommen in der Architektur. Bevor wir lernen, wie wir intelligente, interagierende Systeme entwerfen, müssen wir uns ansehen, wie wir unsere Daten überhaupt strukturieren. In diesem Kapitel werden wir lernen, wie wir aus primitiven Datentypen eigene, komplexe Datentypen erschaffen. Wir bauen das Fundament.

## 1.1 Der Weg aus der imperativen Welt

Erinnern Sie sich an unser Praxisprojekt aus dem ersten Buch? Wir haben ein kleines Dungeon-Spiel für die Konsole programmiert. Unsere Spielfigur, der Held, hatte verschiedene Eigenschaften: einen Namen, Lebenspunkte und eine Position auf dem Spielfeld (X- und Y-Koordinate).

In der imperativen Welt haben wir das so gelöst:

```
public class ImperativesDungeon {
    public static void main(String[] args) {
        // Daten unseres Helden
        String heldName = "Artus";
        int heldLeben = 100;
        int heldX = 5;
        int heldY = 2;

        System.out.println(heldName + " hat " + heldLeben + " HP.");
    }
}
```

Das funktioniert einwandfrei. Doch was passiert, wenn unser Held nicht allein ist? Was, wenn wir zehn Goblins in unserem Dungeon platzieren wollen? In der imperativen Welt müssten wir nun auf Arrays zurückgreifen, da wir nicht für jedes Monster eigene Variablen (goblin1Leben, goblin2Leben ...) anlegen können:

```
// Imperativer Albtraum bei vielen Entitäten:
String[] monsterNamen = {"Goblin", "Ork", "Troll"};
int[] monsterLeben = {30, 80, 200};
int[] monsterX = {1, 4, 8};
int[] monsterY = {1, 5, 9};

// Wenn wir den Ork bewegen wollen, müssen wir wissen,
// dass er an Index 1 in ALLEN Arrays steht.
monsterX[1] = 5;
```

Spüren Sie das Problem? Der Name, die Lebenspunkte und die Koordinaten des Orks gehören logisch untrennbar zusammen. Sie definieren *eine* Entität: den Ork. In unserem imperativen Code sind diese Daten jedoch über vier völlig voneinander unabhängige Arrays verstreut. Wenn

wir beim Sortieren oder Löschen eines Monsters versehentlich nur drei der vier Arrays aktualisieren, heißt unser Ork plötzlich "Troll" oder übernimmt die Lebenspunkte des Goblins.

Die imperative Programmierung zwingt uns, zusammengehörige Daten auseinanderzureißen. Die objektorientierte Programmierung (OOP) erlaubt es uns, sie wieder zusammenzufügen.

## 1.2 Die Klassendefinition: Blaupausen für den Speicher

Um das Problem der verstreuten Daten zu lösen, bietet uns Java die Möglichkeit, **eigene Datentypen** zu erfinden. Neben den eingebauten primitiven Typen wie int, double oder boolean, können wir dem Compiler sagen: *"Pass auf, ab heute gibt es in meinem Programm einen neuen Datentyp namens Monster!"*

Diesen neuen Datentyp definieren wir mithilfe einer **Klasse** (class). Eine Klasse ist eine reine abstrakte Beschreibung. Sie ist der **Bauplan** (oder die Ausstechform). Wenn Sie den Bauplan eines Hauses zeichnen, können Sie noch nicht darin wohnen. Der Bauplan beschreibt nur, *dass* es Wände und Türen geben wird.

Lassen Sie uns den Bauplan für ein Monster schreiben:

```
// Datei: Monster.java
// Mit dem Schlüsselwort 'class' definieren wir einen neuen Datentyp.
public class Monster {

    // Die Eigenschaften (Attribute), die JEDES Monster haben wird:
    String name;
    int lebenspunkte;
    int positionX;
    int positionY;

}
```

Das ist unsere erste echte Klasse! Alles, was wir getan haben, ist Variablen innerhalb der geschweiften Klammern der Klasse zu deklarieren. Variablen, die direkt in einer Klasse (und nicht innerhalb einer Methode) definiert werden, nennt man **Attribute** (oder Felder / *Fields*).

Unsere Klasse Monster bündelt nun vier primitive Variablen zu einer einzigen logischen Einheit. Wichtig: Zu diesem Zeitpunkt existiert noch kein einziges Monster im Speicher unseres Computers. Wir haben Java lediglich erklärt, *wie* ein Monster aussehen würde, *wenn* wir eines erschaffen.

## 1.3 Klassennutzung: Objekte erschaffen (Instanziierung)

Nun wollen wir aus unserer Blaupause ein reales Haus bauen. Wir wollen mit unserer Ausstechform einen echten Keks backen. In der Fachsprache der Objektorientierung sagen wir:

Wir erzeugen ein **Objekt** (auch **Instanz** genannt) aus der Klasse.

Um ein Objekt zu erschaffen, nutzen wir in Java das Schlüsselwort new.

Schauen wir uns an, wie wir das in unserem Hauptprogramm machen:

```
// Datei: Spiel.java
public class Spiel {
    public static void main(String[] args) {

        // 1. Deklaration einer Variablen vom Typ unseres neuen Datentyps
        Monster meinErsterFeind;

        // 2. Erschaffung des eigentlichen Objekts im Speicher mit 'new'
        meinErsterFeind = new Monster();

        // Beide Schritte kombiniert in einer Zeile (der Standardweg):
        Monster boss = new Monster();

    }
}
```

Was passiert in der Zeile Monster boss = new Monster(); ganz genau? Das ist einer der wichtigsten Momente Ihrer Java-Ausbildung, also betrachten wir es unter der Lupe:

1. **new Monster()**: Das Schlüsselwort new weist die Java Virtual Machine (JVM) an, im Arbeitsspeicher des Computers Platz zu reservieren. Und zwar exakt so viel Platz, wie benötigt wird, um einen String und drei int-Werte zu speichern (denn das diktiert unser Bauplan!). Die JVM baut das Objekt im Speicher zusammen.
2. **Monster boss**: Wir deklarieren eine Variable namens boss. Ihr Datentyp ist nicht int oder boolean, sondern unser eigener Typ Monster.
3. **=**: Das Zuweisungszeichen verbindet die Variable mit dem neu erschaffenen Objekt im Speicher.

## Vorsicht, wichtige Architektur-Regel:

Die Variable boss *enthält* nicht das riesige Monster-Objekt selbst! Sie speichert lediglich die Speicheradresse (eine Art Fernbedienung), wo das Objekt im Arbeitsspeicher zu finden ist. Man nennt solche Variablen **Referenzvariablen**.

## 1.4 Arbeiten mit Attributen (Zustand)

Wir haben nun zwei Monster-Objekte (meinErsterFeind und boss) erschaffen. Aktuell sind diese Objekte noch leere Hüllen. Die Zahlen-Attribute (int) stehen standardmäßig auf 0, der Text (String) auf null (dem Nichts).

Wir wollen unseren Monstern nun Leben einhauchen. Um auf das Innere eines Objekts zuzugreifen – also um seine Attribute zu lesen oder zu verändern –, nutzen wir den **Punkt-Operator** (.).

Der Punkt-Operator liest sich wie ein "Besitz-Anzeiger": boss.name bedeutet "der Name *von* boss".

Lassen Sie uns unser komplettes Programm mit Werten füllen:

```
public class Spiel {
    public static void main(String[] args) {

        // Wir erschaffen das erste Monster
        Monster goblin = new Monster();

        // Wir setzen die Attribute des Goblins über den Punkt-Operator
        goblin.name = "Fieser Goblin";
        goblin.lebenspunkte = 30;
        goblin.positionX = 2;
        goblin.positionY = 5;

        // Wir erschaffen ein ZWEITES, völlig unabhängiges Monster
        Monster drache = new Monster();
        drache.name = "Smaug";
        drache.lebenspunkte = 5000;
        drache.positionX = 10;
        drache.positionY = 10;

        // Wir lesen die Daten wieder aus und drucken sie auf die Konsole
        System.out.println("Achtung! Ein " + goblin.name + " erscheint!");
        System.out.println("Er hat " + goblin.lebenspunkte + " HP.");

        System.out.println("Der Boss " + drache.name + " wartet auf Feld "
                           + drache.positionX + "/" + drache.positionY);

        // Kampf-Simulation: Der Goblin verliert 10 Lebenspunkte
        goblin.lebenspunkte = goblin.lebenspunkte - 10;
        System.out.println(goblin.name + " hat jetzt " +
                           goblin.lebenspunkte + " HP.");
    }
}
```

## Die Magie der unabhängigen Objekte:

Obwohl goblin und drache denselben Bauplan (die Klasse Monster) nutzen, sind sie im Speicher völlig getrennt. Wenn wir dem Goblin 10 Lebenspunkte abziehen (goblin.lebenspunkte = goblin.lebenspunkte - 10;), kratzt das den Drachen überhaupt nicht. drache.lebenspunkte bleibt

bei stolzen 5000.

Jedes Objekt besitzt seinen eigenen, individuellen **Zustand**. Der Zustand eines Objekts ist zu jedem beliebigen Zeitpunkt im Programm die Gesamtheit der Werte all seiner Attribute.

Unsere Daten sind jetzt nicht mehr in unzähligen Arrays verstreut. Wenn wir den Drachen als Ganzes an eine andere Stelle in unserem Code weitergeben wollen, müssen wir nicht mehr vier verschiedene Variablen übergeben. Wir übergeben einfach die eine Referenz drache. Das Objekt trägt all seine Daten huckepack mit sich herum. Wir haben glorreiche, zusammenhängende Datencontainer erschaffen!

---

Doch ein echter Architekt weiß: Reine Datencontainer sind erst der halbe Weg. Unsere Monster liegen immer noch dumm im Speicher. Wenn sich ein Monster bewegen soll, muss das Hauptprogramm mühsam die X- und Y-Koordinaten von außen anpassen. Das Monster weiß nicht, wie man läuft. Im nächsten Kapitel werden wir unseren Datencontainern beibringen, eigenständig zu agieren. Wir werden ihnen **Verhalten** geben.

# 2 Klassen und Objekte II – Verhalten hinzufügen

Im ersten Kapitel haben wir gelernt, wie wir zusammengehörige Daten in einer Klasse bündeln. Unsere Monster existieren nun als kompakte Einheiten im Speicher. Doch wenn wir ehrlich sind: Bisher sind unsere Objekte ziemlich passiv. Sie sind reine Datencontainer, die darauf warten, dass ein externes Programm (wie unsere main-Methode) ihre Werte ausliest oder verändert.

Ein echter Objekt-Architekt gibt sich damit nicht zufrieden. In der realen Welt haben Dinge nicht nur Eigenschaften (Zustand), sie können auch Dinge tun (Verhalten). Ein Monster *hat* nicht nur Lebenspunkte, es *kann* auch brüllen, sich bewegen oder Schaden nehmen. In diesem Kapitel bringen wir unseren Objekten bei, selbst aktiv zu werden.

## 2.1 Klassendefinition: Erweiterung um Methoden

Um einem Objekt Verhalten hinzuzufügen, definieren wir **Methoden** innerhalb der Klasse. Eine Methode ist im Grunde nichts anderes als eine Funktion (wie Sie sie aus der imperativen Programmierung kennen), die jedoch fest in eine Klasse eingebaut ist.

Der entscheidende Unterschied zur imperativen Welt: Eine Methode, die zu einem Objekt gehört, hat direkten Zugriff auf die Attribute genau dieses Objekts!

Lassen Sie uns unserer Monster-Klasse aus dem vorherigen Kapitel zwei Fähigkeiten geben: Es soll sich vorstellen können und es soll in der Lage sein, Schaden zu nehmen.

```
// Datei: Monster.java
public class Monster {
    // 1. Attribute (Zustand)
    String name;
    int lebenspunkte;
    int positionX;
    int positionY;

    // 2. Methoden (Verhalten)

    // Methode ohne Parameter und ohne Rückgabewert (void)
    public void bruellen() {
        // Die Methode greift direkt auf das Attribut 'name' zu!
        System.out.println(name + " brüllt: WAAAGH! Ich habe " +
                           lebenspunkte + " HP!");
    }

    // Methode mit Parameter, die den eigenen Zustand verändert
    public void schadenNehmen(int schaden) {
        lebenspunkte = lebenspunkte - schaden;
        System.out.println(name + " erleidet " + schaden + " Schaden.");
```

```
        if (lebenspunkte <= 0) {
            System.out.println(name + " röchelt und bricht zusammen.");
        }
    }
}
```

Fällt Ihnen etwas Magisches an der Methode bruellen() auf? Wir übergeben ihr gar keine Parameter! Woher weiß die Methode dann, welchen name und welche lebenspunkte sie auf die Konsole drucken soll?

Die Antwort ist das Herzstück der Objektorientierung: **Methoden werden immer *auf* einem konkreten Objekt aufgerufen.** Wenn das Objekt die Methode ausführt, schaut es in seinen *eigenen* Bauch (seinen eigenen Speicherbereich) und verwendet instinktiv seine eigenen Attributwerte.

Schauen wir uns an, wie wir diese Methoden in unserem Hauptprogramm über den Punkt-Operator aufrufen:

```
// Datei: Spiel.java
public class Spiel {
    public static void main(String[] args) {
        // Erstes Objekt erschaffen und manuell befüllen
        Monster goblin = new Monster();
        goblin.name = "Fieser Goblin";
        goblin.lebenspunkte = 30;

        // Zweites Objekt erschaffen und befüllen
        Monster drache = new Monster();
        drache.name = "Smaug";
        drache.lebenspunkte = 5000;

        // Jetzt rufen wir das Verhalten auf!
        // Der Befehl vor dem Punkt bestimmt, WER die Aktion ausführt.

        goblin.bruellen(); // Druckt: "Fieser Goblin brüllt: ... Ich habe
                           // 30 HP!"
        drache.bruellen(); // Druckt: "Smaug brüllt: ... Ich habe 5000 HP!"

        System.out.println("--- Ein Kampf beginnt ---");

        // Der Goblin nimmt 10 Schaden. Seine eigene Methode reduziert
        // SEINE Lebenspunkte.
        goblin.schadenNehmen(10);

        // Der Drache nimmt 6000 Schaden.
        drache.schadenNehmen(6000);
    }
}
```

Unsere Architektur wird langsam elegant. Das Hauptprogramm (Spiel) muss nicht mehr selbst berechnen, wie die Lebenspunkte eines Monsters nach einem Treffer aussehen. Es sagt dem Monster einfach: *"Nimm 10 Schaden!"* Wie das Monster diesen Schaden verarbeitet, ist nun die alleinige Verantwortung des Monsters. Daten und Verhalten sind endlich vereint!

## 2.2 Konstruktoren zur sicheren Initialisierung

Es gibt jedoch noch einen großen Schwachpunkt in unserem bisherigen Code. Um ein Monster einsatzbereit zu machen, müssen wir nach dem new-Aufruf mühsam jedes einzelne Attribut manuell setzen:

```
Monster ork = new Monster(); // Hier ist der Ork noch ein leeres,
                             // "kaputtes" Objekt
ork.name = "Grumsh";         // Wir setzen mühsam den Namen
ork.lebenspunkte = 80;       // Wir setzen mühsam die HP
// ... was ist mit X und Y? Vergessen!
```

Das ist extrem fehleranfällig. Was passiert, wenn wir in der Eile vergessen, die Lebenspunkte zu setzen? Der int-Wert bleibt auf dem Standardwert 0. Unser Ork spawnt im Dungeon und ist auf der Stelle tot! Ein intelligentes System sollte es gar nicht erst zulassen, dass ungültige oder unfertige Objekte im Speicher herumliegen.

Wir müssen garantieren, dass ein Objekt direkt im Moment seiner Erschaffung mit allen lebenswichtigen Daten versorgt wird. Dafür gibt es **Konstruktoren**.

Ein Konstruktor ist ein spezieller Blockabschluss innerhalb einer Klasse, der **exakt einmal** aufgerufen wird: nämlich genau in der Millisekunde, in der das Objekt mit new geboren wird.

**Regeln für Konstruktoren:**

1. Ein Konstruktor heißt *immer exakt so wie die Klasse* (inklusive Groß-/Kleinschreibung).
2. Ein Konstruktor hat *keinen Rückgabewert*, nicht einmal void.

Lassen Sie uns unsere Monster-Klasse um einen Konstruktor erweitern:

```
public class Monster {
    String name;
    int lebenspunkte;
    int positionX;
    int positionY;

    // Dies ist der Konstruktor!
    // Er verlangt vier Parameter, die bei der Erschaffung zwingend
mitgegeben werden müssen.
    public Monster(String startName, int startLeben,
```

```
                    int startX, int startY) {
        // Wir nehmen die übergebenen Start-Werte und speichern sie
        // dauerhaft in den Attributen des neuen Objekts.
        name = startName;
        lebenspunkte = startLeben;
        positionX = startX;
        positionY = startY;

        System.out.println(">>> System: Ein neues Monster namens " + name +
                           " ist gespawnt!");
    }

    public void bruellen() {
        System.out.println(name + " brüllt: WAAAGH!");
    }
}
```

Sobald wir einen solchen Konstruktor in unsere Klasse schreiben, ändert sich die Art und Weise, wie wir Objekte instanziieren, radikal. Der leere Aufruf new Monster() funktioniert ab sofort **nicht mehr**! Der Compiler wirft einen Fehler und sagt uns sinngemäß: *"Halt! Der Bauplan verlangt vier Informationen, um ein Monster zu bauen. Ohne diese Daten weigere ich mich, das Objekt zu erschaffen!"*

Wir haben unser System manipulationssicher gemacht. Das Erschaffen und Befüllen passiert nun zwangsweise in einem einzigen, sicheren Schritt:

```
public class Spiel {
    public static void main(String[] args) {

        // Fehler! Der Compiler weigert sich, das zu kompilieren:
        // Monster fehlerhafterGoblin = new Monster();

        // Korrekt: Erschaffung und Initialisierung in einer Zeile.
        // Die Argumente in den runden Klammern werden direkt an den
        // Konstruktor übergeben.
        Monster goblin = new Monster("Fieser Goblin", 30, 2, 5);
        Monster drache = new Monster("Smaug", 5000, 10, 10);

        goblin.bruellen();
    }
}
```

Mit Konstruktoren garantieren wir, dass Objekte ab der ersten Zeile ihrer Existenz in einem gültigen, vollständig initialisierten Zustand sind. Wir als Programmierer müssen uns nicht mehr merken, welche Attribute wir nach dem new noch setzen müssen – der Compiler erinnert uns automatisch daran, falls wir etwas vergessen.

Sie haben nun das Fundament der Objektorientierung verstanden: Klassen sind Baupläne, die Zustand (Attribute) und Verhalten (Methoden) definieren. Mit Konstruktoren erschaffen wir gültige Objekte aus diesen Bauplänen.

---

Im nächsten Kapitel widmen wir uns dem "Feinschliff" dieser Konzepte: Wir lernen unter anderem, wie sich Objekte mit dem Schlüsselwort this auf sich selbst beziehen können, und entdecken, dass manche Eigenschaften gar nicht den einzelnen Objekten gehören, sondern der gesamten Klasse.

# 3 Klassen und Objekte III – Der Feinschliff

Sie haben nun die beiden tragenden Säulen der Objektorientierung kennengelernt: den Zustand (Attribute) und das Verhalten (Methoden). Sie können mithilfe von Konstruktoren sichere, einsatzbereite Objekte erschaffen. In diesem Kapitel widmen wir uns dem architektonischen Feinschliff. Wir klären, wie Objekte über sich selbst sprechen, wie sie Wissen mit der gesamten Klasse teilen und wie wir als Architekten überprüfen, ob zwei Objekte eigentlich "dasselbe" sind.

## 3.1 Das Schlüsselwort this

Erinnern Sie sich an den Konstruktor aus dem letzten Kapitel? Wir haben unsere Parameter dort startName oder startLeben genannt, um sie von den Attributen name und lebenspunkte zu unterscheiden.

```
// Bisheriger Ansatz: Unterschiedliche Namen
public Monster(String startName) {
    name = startName;
}
```

In großen Projekten führt das ständige Erfinden von neuen, leicht abgewandelten Variablennamen zu Verwirrung. Es wäre viel sauberer, wenn der Parameter für den Namen einfach name hieße. Doch wenn wir das tun, geraten wir in ein Problem:

```
public Monster(String name) {
    name = name; // Hä? Was weise ich hier wem zu?
}
```

Wenn ein Methoden- oder Konstruktorparameter exakt denselben Namen hat wie ein Attribut der Klasse, dann **verdeckt** der Parameter das Attribut. Der Compiler denkt bei name nun immer an den Parameter, nicht mehr an das Attribut des Objekts.

Um dieses Problem zu lösen, gibt uns Java das Schlüsselwort this. this bedeutet übersetzt "dieses Objekt". Es ist eine interne Fernbedienung, mit der ein Objekt auf sich selbst zeigen kann. Sobald Sie this. vor eine Variable schreiben, zwingen Sie den Compiler, das Attribut des Objekts zu nehmen und den Parameter zu ignorieren.

### Beispiel: Konstruktor mit this

```
public class Monster {
    String name;
    int lebenspunkte;

    // Die Parameter heißen nun exakt wie die Attribute (Best Practice)
    public Monster(String name, int lebenspunkte) {
        // this.name ist das Attribut des Objekts.
```

```
        // name ist der übergebene Parameter.
        this.name = name;
        this.lebenspunkte = lebenspunkte;
    }

    public void heilen(int heilungsWert) {
        // Auch in Methoden können wir this nutzen, um es deutlich zu machen.
        // Oft ist es optional, aber es erhöht die Lesbarkeit.
        this.lebenspunkte = this.lebenspunkte + heilungsWert;
        System.out.println(this.name + " heilt sich.");
    }
}
```

Gewöhnen Sie sich dieses Muster an. Es ist der absolute Standard in der Java-Welt.

## 3.2 Klassenattribute und Klassenmethoden (static)

Bisher gehörte jedes Attribut zu einem konkreten Objekt. Der Goblin hatte 30 HP, der Drache 5000 HP. Wenn der Goblin Schaden nahm, interessierte das den Drachen nicht. Man nennt diese Variablen daher auch **Instanzvariablen**.

Was aber, wenn wir eine Information benötigen, die nicht einem einzelnen Objekt, sondern der *gesamten Bauart* (der Klasse) gehört? Stellen Sie sich vor, wir wollen mitzählen, wie viele Monster im Verlauf unseres Spiels insgesamt erschaffen wurden.

Wenn wir ein normales Attribut int zaehler nutzen, startet es für jedes neue Monster bei 0. Wir brauchen eine Variable, die nur **ein einziges Mal im Speicher** existiert und die sich alle Objekte teilen. Hierfür nutzen wir das Schlüsselwort static.

```
public class Monster {
    String name;

    // Ein Klassenattribut (static). Es existiert nur EINMAL für die
    // gesamte Klasse!
    static int monsterZaehler = 0;

    public Monster(String name) {
        this.name = name;

        // Jedes Mal, wenn ein Monster erschaffen wird, erhöhen
        // wir den GEMEINSAMEN Zähler um 1.
        monsterZaehler = monsterZaehler + 1;
    }

    // Eine Klassenmethode (static). Auch sie gehört zur Klasse, nicht
    // zum Objekt.
    public static void zeigeStatistik() {
        System.out.println("Es existieren " + monsterZaehler +
```

```
                                " Monster auf der Welt.");

        // ACHTUNG: Eine statische Methode kann NICHT auf normale Attribute
        // (wie 'name') zugreifen! Es gibt hier kein bestimmtes Objekt, auf
        // das sich 'name' beziehen könnte.
        // System.out.println(this.name); // --> Führt zu einem
                                          // Compilerfehler!
    }
}
```

## Nutzung in der main-Methode:

Statische Attribute und Methoden rufen wir nicht über ein Objekt (goblin.zeigeStatistik()) auf, sondern direkt über den Namen der Klasse:

```
public class Spiel {
    public static void main(String[] args) {
        // Wir können die Methode aufrufen, BEVOR überhaupt ein Monster
        // existiert!
        Monster.zeigeStatistik(); // Druckt: Es existieren 0 Monster auf der
                                  // Welt.

        Monster m1 = new Monster("Goblin");
        Monster m2 = new Monster("Troll");
        Monster m3 = new Monster("Drache");

        Monster.zeigeStatistik(); // Druckt: Es existieren 3 Monster auf der
                                  // Welt.
    }
}
```

## 3.3 Konstanten in der Objektwelt

Oft haben wir Werte, die für alle Objekte gleich sind und sich *niemals* ändern dürfen. Zum Beispiel das absolute Lebenspunkte-Maximum in unserem Spiel.

Dafür kombinieren wir static (gilt für alle) mit dem Schlüsselwort final (darf nach der ersten Zuweisung nie wieder geändert werden). So erschaffen wir eine **Konstante**. Konstanten sind Namen für Werte. Wenn wir später mal den Wert ändern wollen, müssen wir ihn nur an einer einzigen Stelle im Code ändern. Konstanten werden in Java per Konvention komplett großgeschrieben und Wörter mit Unterstrichen getrennt.

```
public class Monster {
    // Eine globale Konstante
    public static final int MAX_LEBENSPUNKTE = 9999;
```

```
    String name;
    int lebenspunkte;

    public void heilen(int wert) {
        this.lebenspunkte = this.lebenspunkte + wert;
        // Schutzmechanismus unter Nutzung der Konstante
        if (this.lebenspunkte > MAX_LEBENSPUNKTE) {
            this.lebenspunkte = MAX_LEBENSPUNKTE;
        }
    }
}
```

## 3.4 Default-Konstruktor vs. Copy-Konstruktor

Sie wissen: Wenn wir keinen Konstruktor schreiben, zwingt uns der Compiler nicht dazu, beim new-Aufruf Parameter zu übergeben. Warum ist das so?

Wenn eine Klasse *keinen einzigen* Konstruktor besitzt, generiert Java beim Kompilieren unsichtbar einen leeren sogenannten **Default-Konstruktor**: public Monster() {}. Sobald Sie aber auch nur einen einzigen eigenen Konstruktor (z.B. mit dem Namen) schreiben, verschwindet dieser geschenkte Default-Konstruktor!

Manchmal möchten wir ein Objekt erschaffen, das ein exakter Klon eines bereits existierenden Objekts ist. Wenn ein Goblin sich teilt, wollen wir einen zweiten Goblin mit denselben Werten. Dafür schreiben wir einen **Copy-Konstruktor**. Er akzeptiert als Parameter ein Objekt derselben Klasse.

```
public class Monster {
    String name;
    int lebenspunkte;

    // Normaler Konstruktor
    public Monster(String name, int lebenspunkte) {
        this.name = name;
        this.lebenspunkte = lebenspunkte;
    }

    // Copy-Konstruktor: Nimmt ein fertiges Monster als Vorlage
    public Monster(Monster original) {
        // Wir kopieren die Werte aus dem Original in unser neues Objekt
        this.name = original.name;
        this.lebenspunkte = original.lebenspunkte;
    }
}

// Nutzung des Copy-Konstruktors
Monster urgoblin = new Monster("Goblin", 30);
```

```
Monster klon = new Monster(urgoblin); // Klon hat jetzt auch "Goblin" und
                                      // 30 HP
```

## 3.5 Objektreferenzen und Subobjekte (Delegation)

Bisher waren unsere Attribute simple Datentypen wie String oder int. Aber die wahre Stärke der Objektorientierung entfaltet sich, wenn Objekte aus anderen Objekten zusammengebaut werden!

Stellen Sie sich vor, unser Monster trägt eine Waffe. Die Waffe selbst hat komplexe Eigenschaften (Name, Schaden, Haltbarkeit). Es wäre falsch, all diese Eigenschaften direkt in die Monster-Klasse zu stopfen. Stattdessen bauen wir eine eigene Klasse Waffe und geben dem Monster ein Attribut vom Typ Waffe.

```
// 1. Die Klasse für das Subobjekt
public class Waffe {
    String bezeichnung;
    int schaden;

    public Waffe(String bezeichnung, int schaden) {
        this.bezeichnung = bezeichnung;
        this.schaden = schaden;
    }
}

// 2. Die Hauptklasse delegiert Aufgaben an das Subobjekt
public class Monster {
    String name;
    Waffe meineWaffe; // Ein Attribut vom Typ eines anderen Objekts!

    public Monster(String name, Waffe startWaffe) {
        this.name = name;
        this.meineWaffe = startWaffe;
    }

    public void angreifen() {
        // Das Monster nutzt das Subobjekt, um an den Schaden zu kommen.
        // Man nennt das "Delegation".
        System.out.println(this.name + " greift mit " +
                           this.meineWaffe.bezeichnung +
                           " an und macht " + this.meineWaffe.schaden +
                           " Schaden!");
    }
}

// Im Hauptprogramm verbinden wir die Objekte wie Legosteine
public class Spiel {
```

```
    public static void main(String[] args) {
        Waffe schwert = new Waffe("Eisenschwert", 15);
        Monster ork = new Monster("Grumsh", schwert);

        ork.angreifen();
    }
}
```

Unsere Architektur wächst! Objekte rufen Methoden auf anderen Objekten auf. Ein Netzwerk aus interagierenden Bausteinen entsteht.

## 3.6 Gleichheit: Referenz- vs. Objektgleichheit (equals)

In der imperativen Programmierung haben Sie gelernt, dass man mit == prüft, ob zwei Dinge gleich sind (if (x == 5)). In der Objektwelt ist der doppelte Gleichheitsoperator jedoch extrem gefährlich!

Schauen wir uns folgendes Beispiel an:

```
Monster m1 = new Monster("Goblin", 30);
Monster m2 = new Monster("Goblin", 30);

if (m1 == m2) {
    System.out.println("Die Monster sind gleich!");
} else {
    System.out.println("Die Monster sind unterschiedlich!");
}
```

Was wird gedruckt? Die Antwort ist: **"Die Monster sind unterschiedlich!"**

Warum? Erinnern Sie sich an Kapitel 1: Die Variablen m1 und m2 speichern nicht das Monster selbst, sondern nur die *Speicheradresse* (die Fernbedienung). Da wir zweimal new aufgerufen haben, existieren zwei getrennte Monster im Speicher. Der Operator == prüft bei Objekten nur, ob die *Fernbedienungen auf exakt dasselbe Objekt im Speicher zeigen*. Man nennt das **Referenzgleichheit**.

Um zu prüfen, ob zwei inhaltlich identische, aber physisch getrennte Objekte die gleichen Daten haben (**Objektgleichheit**), rufen wir traditionell eine Methode auf, die wir selbst schreiben müssen: die equals-Methode.

```
public class Monster {
    String name;
    int lebenspunkte;

    public Monster(String name, int lebenspunkte) {
        this.name = name;
        this.lebenspunkte = lebenspunkte;
```

```
    }

    // Wir bauen uns eine eigene Vergleichsmethode
    public boolean equals(Monster anderesMonster) {
        // Wenn das andere Monster gar nicht existiert, sind sie nicht gleich
        if (anderesMonster == null) {
            return false;
        }

        // Sie sind inhaltlich gleich, wenn Name UND Lebenspunkte
        // übereinstimmen
        if (this.name.equals(anderesMonster.name) &&
            this.lebenspunkte == anderesMonster.lebenspunkte) {
            return true;
        }

        return false;
    }
}
```

*(Hinweis für spätere Kapitel: Wie Sie sehen, rufen wir beim String-Vergleich ebenfalls .equals() auf! String ist nämlich auch ein Objekt. Später, im Kapitel über Vererbung, werden wir lernen, wie man die equals-Methode absolut professionell überschreibt).*

Wenn wir nun if (m1.equals(m2)) aufrufen, liefert Java uns true.

## 3.7 Die main-Funktion im objektorientierten Kontext

Lassen Sie uns zum Abschluss dieses Teils ein Geheimnis lüften, das Sie seit dem ersten Tag Ihrer Java-Reise begleitet. Die Zeile:

```
public static void main(String[] args)
```

Warum musste die Hauptmethode in unseren imperativen Programmen immer static sein?

Die Antwort liegt nun auf der Hand: Wenn Sie ein Java-Programm starten, existiert noch kein einziges Objekt im Speicher. Niemand hat bisher new aufgerufen. Wenn die main-Methode ein normales Objekt-Verhalten (ohne static) wäre, bräuchte die Java Virtual Machine erst ein Objekt der Klasse, um die Methode aufrufen zu können. Da es aber am Anfang keine Objekte gibt, beißt sich die Katze in den Schwanz.

Indem wir die main-Methode als static markieren, machen wir sie zu einer *Klassenmethode*. Die JVM kann sie direkt über den Klassennamen (Spiel.main(...)) aufrufen, ohne dass ein Objekt im Speicher existieren muss. Von dort aus – dem absoluten Nullpunkt – können wir dann mit new unsere Architektur hochfahren und die ersten Objekte in die Welt setzen.

Was hat es mit String[] args auf sich?  String[] ist ein Array aus Texten (Strings). Der Name args ist eine Abkürzung für *Arguments* (Argumente).

Als Architekt bauen Sie ein in sich geschlossenes Programm. Aber oft möchten Sie diesem Programm direkt beim Start von außen (vom Betriebssystem) bestimmte Startinformationen mitgeben, noch bevor das Programm überhaupt richtig läuft. Stellen Sie sich das so vor, als würden Sie dem Bauleiter einen Umschlag mit Sonderwünschen in die Hand drücken, bevor er die Baustelle aufschließt.

Wenn Sie Ihr Programm später als Profi über die Kommandozeile (das Terminal) Ihres Betriebssystems starten, sieht der Befehl oft so aus:

```
java Spiel schwer Artus
```

Sie rufen das Programm Spiel auf und übergeben ihm durch Leerzeichen getrennt zwei Wörter: "schwer" und "Artus". Die Java Virtual Machine nimmt diese Wörter, packt sie in ein Array und übergibt dieses Array an Ihre main-Methode – exakt in die Variable args!

---

Sie haben es geschafft! Der erste große Teil des Buches ist abgeschlossen. Sie wissen nun, was Objekte sind, wie man sie sicher instanziiert und verwaltet. Doch bisher ist alles, was wir programmiert haben, völlig öffentlich. Jeder kann unsere Attribute von außen manipulieren.

Im kommenden Teil II („Ordnung und Geheimniskrämerei“) werden wir unseren Architekten-Werkzeugkasten um Schutzmechanismen erweitern. Wir werden lernen, wie wir Code in Paketen strukturieren und unsere Daten mit Zugriffsrechten vor böswilligen Eingriffen absichern.

# Teil II: Ordnung und Geheimniskrämerei

# 4 Pakete (Packages)

Bisher haben wir unsere Klassen (Spiel, Monster, Waffe) einfach als einzelne Textdateien in denselben Ordner auf unserer Festplatte gelegt. Für unser kleines Dungeon-Spiel aus dem ersten Buch oder die ersten Experimente mit Objekten war das völlig ausreichend. Doch als angehende Software-Architekten müssen wir größer denken.

## 4.1 Motivation: Strukturierung von Klassenbibliotheken

Stellen Sie sich vor, Sie entwickeln ein echtes, kommerzielles Videospiel. Sie werden nicht drei Klassen haben, sondern dreitausend. Sie haben Klassen für die Grafik-Engine, für das Netzwerk, für das Speichern von Spielständen, für Hunderte verschiedene Gegner und Items.

Wenn Sie alle 3000 Java-Dateien in einen einzigen Ordner werfen, entsteht ein absolutes Chaos. Schlimmer noch: Was passiert, wenn zwei Programmierer in Ihrem Team unabhängig voneinander eine Klasse namens Spieler schreiben? Der eine programmiert den Spieler für das Netzwerk (Verbindungsdaten), der andere den Spieler für die 3D-Welt (Koordinaten). Sie können keine zwei Dateien mit dem exakt selben Namen in einem Ordner speichern. Das Betriebssystem wird streiken, und der Java-Compiler ist verwirrt.

Die Lösung für dieses Problem kennen Sie aus Ihrem Alltag: **Ordner**.

In der Java-Welt nennen wir diese Ordnerstrukturen **Pakete (Packages)**. Pakete erlauben es uns, Klassen logisch zu gruppieren und Namenskonflikte zu vermeiden. Eine Klasse netzwerk.Spieler ist für den Compiler eine völlig andere Klasse als welt.Spieler.

## 4.2 Definition von Paketen (package-Anweisung)

Um eine Klasse in ein Paket zu legen, müssen wir Java das ganz explizit mitteilen. Das tun wir mit dem Schlüsselwort package.

**Die wichtigste Regel:** Die package-Anweisung muss immer die **allererste Code-Zeile** in Ihrer Java-Datei sein! Nichts (außer Kommentaren) darf davor stehen.

Lassen Sie uns unsere Monster-Klasse in ein passendes Paket verschieben:

```
// Datei: Monster.java
// Dies MUSS die erste Zeile sein. Wir legen die Klasse in das Paket
"charaktere".
package charaktere;

public class Monster {
    String name;
    int lebenspunkte;

    public Monster(String name, int lebenspunkte) {
        this.name = name;
```

```
        this.lebenspunkte = lebenspunkte;
    }
}
```

In der realen Welt der Softwareentwicklung nutzt man oft verschachtelte Pakete, um noch genauer zu strukturieren. Pakete werden dabei durch einen Punkt (.) getrennt. Die Konvention in Java ist es, die umgedrehte Internetadresse (Domain) der eigenen Firma oder des Projekts als Basis zu nutzen, um weltweite Eindeutigkeit zu garantieren.

```
// Ein professioneller Paketname
package de.meinspiel.charaktere.feinde;

public class Goblin {
    // ...
}
```

## Die physikalische Realität:

Wenn Sie schreiben package de.meinspiel.charaktere.feinde;, zwingt Sie der Java-Compiler dazu, dass die Datei Goblin.java auf Ihrer Festplatte auch exakt in dieser Ordnerstruktur liegt! Der Pfad muss lauten: de/meinspiel/charaktere/feinde/Goblin.java. Wenn der Textcode und die echten Windows/Mac-Ordner nicht übereinstimmen, weigert sich Java, den Code zu kompilieren.

## 4.3 Nutzung von Paketen (import)

Nun haben wir unsere Klassen wunderschön in Pakete sortiert. Unsere Monster-Klasse liegt im Paket charaktere, und unser Hauptprogramm Spiel liegt in einem anderen Paket namens haupt.

Wenn wir nun im Hauptprogramm ein Monster erschaffen wollen, laufen wir in ein Problem:

```
// Datei: Spiel.java
package haupt;

public class Spiel {
    public static void main(String[] args) {
        // FEHLER! Der Compiler kennt "Monster" hier nicht.
        Monster m = new Monster("Ork", 100);
    }
}
```

Der Compiler meldet einen Fehler: *"Cannot find symbol"*. Warum? Weil das Hauptprogramm nur die Klassen sieht, die im *selben* Paket (haupt) liegen. Es hat keine Ahnung, dass im Paket charaktere ein Bauplan für Monster liegt.

Um dieses Problem zu lösen, müssen wir die Klasse aus dem anderen Paket **importieren**. Das geschieht mit dem Schlüsselwort import. Die Import-Anweisungen stehen immer ganz oben in der Datei, direkt *unter* der package-Anweisung, aber *vor* der eigentlichen Klassendefinition (public

class ...).

```
package haupt;

// Wir verraten dem Compiler, in welchem Paket er die Klasse "Monster"
// findet.
import charaktere.Monster;

public class Spiel {
    public static void main(String[] args) {
        // Jetzt funktioniert es reibungslos!
        Monster m = new Monster("Ork", 100);
    }
}
```

## Mehrere Klassen importieren:

Wenn wir im Paket charaktere neben dem Monster auch noch einen Held, einen Händler und einen König haben und alle im Hauptprogramm benötigen, können wir jede Klasse einzeln importieren. Wenn uns das zu viel Tipparbeit wird, bietet Java den **Stern-Import** (Wildcard):

```
package haupt;

// Importiert ALLE Klassen, die direkt im Ordner "charaktere" liegen.
import charaktere.*;

public class Spiel {
    // ...
}
```

*Tipp für Architekten: Auch wenn der Stern-Import verlockend ist, bevorzugen Profis oft den expliziten Import einzelner Klassen. So sieht man auf den ersten Blick, welche fremden Bausteine eine Klasse genau verwendet.*

Hier ist ein Entwurf für den kompakten ergänzenden Abschnitt, der sich in Kapitel 4 (Pakete) perfekt als Unterkapitel 4.6 einfügt:

## 4.4 Der vollständige Klassenname (Nutzung ohne import)

Bisher haben wir gelernt, dass wir fremde Klassen mit dem import-Befehl am Anfang der Datei bekannt machen müssen, um sie bequem im Code nutzen zu können. Doch es gibt Situationen, in denen ein import nicht ausreicht oder sogar zu Fehlern führt.

Stellen Sie sich vor, Sie schreiben ein Programm, das sowohl mit der Datenbank von Java als auch mit normalen Kalenderdaten arbeitet. Zufälligerweise gibt es in der riesigen Java-Bibliothek **zwei völlig verschiedene Klassen, die beide exakt Date heißen**:

- java.util.Date (für normale Zeitangaben)
- java.sql.Date (für Datenbank-Zeitstempel)

Wenn Sie nun versuchen, beide mit import einzubinden, wird der Compiler streiken. Er weiß bei dem Wort Date im Code nicht mehr, welche der beiden Klassen Sie meinen. Die Namen kollidieren.

Um diesen Konflikt zu lösen, greifen wir als Architekten auf den **vollständigen Klassennamen (Fully Qualified Class Name)** zurück. Dieser besteht immer aus dem kompletten Paketnamen, einem Punkt und dem eigentlichen Klassennamen.

Wenn Sie den vollständigen Namen nutzen, **benötigen Sie überhaupt keinen import-Befehl!** Sie schreiben den kompletten Pfad einfach direkt in Ihren Code:

```
package haupt;

// KEIN import notwendig!

public class ZeitManager {
    public static void main(String[] args) {

        // Nutzung des vollständigen Namens direkt bei der Deklaration und
        // Erschaffung
        java.util.Date normalesDatum = new java.util.Date();

        java.sql.Date datenbankDatum =
          new java.sql.Date(System.currentTimeMillis());

        System.out.println("Normal: " + normalesDatum);
    }
}
```

## Wann nutzen Sie das?

Da das ständige Tippen von java.util.Date den Code schnell unübersichtlich und lang macht (Boilerplate-Code), nutzen Profis den vollständigen Klassennamen wirklich **nur im absoluten Notfall**: nämlich dann, wenn es zu Namenskonflikten zwischen zwei Paketen kommt. In allen anderen Fällen ist der saubere import am Dateianfang der bessere Architektur-Stil.

## 4.5 Das Paket java.lang und Anonyme Pakete

Vielleicht wundern Sie sich jetzt: *"Moment mal! In Buch 1 haben wir ständig String oder System.out.println() benutzt. Das sind doch auch Klassen. Warum mussten wir die nie importieren?"*

Das ist eine hervorragende architektonische Beobachtung! Die Antwort lautet: Java hat uns geholfen.

Die absolute Basis-Ausstattung von Java liegt in einem Paket namens java.lang (lang für Language). Weil man Klassen wie String, Math oder System in fast jedem Programm braucht, führt der Java-Compiler unsichtbar in jeder Ihrer Dateien einen automatischen Import durch: import java.lang.*;. Sie mussten es also nie selbst schreiben. Alle anderen Java-Pakete, wie z.B. java.util.Scanner, mussten Sie bereits in Buch 1 händisch importieren.

## Was ist ein Anonymes Paket?

Und was ist mit unseren allerersten Klassen, bei denen wir gar kein package oben hingeschrieben haben? Wenn Sie die package-Zeile weglassen, landet Ihre Klasse im sogenannten **anonymen Paket** (Default-Package).

Für winzige Test-Skripte ist das erlaubt. Aber **für professionelle Software ist das Default-Paket streng verboten!** Klassen aus dem namenlosen Paket können nämlich von Klassen, die in echten Paketen liegen, nicht importiert werden. Gewöhnen Sie sich daher an: Jede Klasse bekommt ab heute eine package-Anweisung.

## 4.6 Statischer Import

Erinnern Sie sich an unser Kapitel über Klassenattribute und Klassenmethoden (static)? Wenn wir eine statische Methode oder Konstante einer anderen Klasse nutzen wollen, müssen wir normalerweise immer den Namen der Klasse davor schreiben.

Wenn wir zum Beispiel viel Mathematik betreiben, sieht das schnell mühsam aus:

```
package geodaten;

public class KreisBerechnung {
    public double flaeche(double radius) {
        // Math ist eine Klasse aus java.lang.
        // PI und pow() sind statische Mitglieder dieser Klasse.
        return Math.PI * Math.pow(radius, 2);
    }
}
```

Um den Code lesbarer zu machen, insbesondere wenn wir ständig auf dieselben statischen Konstanten oder Methoden zugreifen, gibt es den **statischen Import**.

Anstatt eine ganze Klasse zu importieren, importieren wir gezielt nur eine bestimmte static-Eigenschaft aus dieser Klasse. Danach können wir den Klassennamen (z.B. Math.) im Code einfach weglassen!

```java
package geodaten;

// Wir importieren gezielt die Konstante PI und die Methode pow
import static java.lang.Math.PI;
import static java.lang.Math.pow;

public class KreisBerechnung {
    public double flaeche(double radius) {
        // Wir können Math weglassen. Der Code liest sich wie normale
        // Mathematik.
        return PI * pow(radius, 2);
    }
}
```

---

Mit Paketen haben Sie gelernt, wie Sie Ordnung in Ihren Code bringen. Sie wissen, wie Sie Klassen wie Akten in verschiedenen Schränken (Ordnern) ablegen und gezielt daraus hervorholen (import).

Doch bisher gilt: Wenn jemand eine Klasse importiert hat, kann er alles darin sehen und verändern. Das ist, als würden Sie zwar Ihr Werkzeug in Kisten sortieren, aber die Kisten stehen offen auf der Straße. Im nächsten Kapitel rüsten wir unsere Architektur mit Schlössern aus. Wir lernen **Zugriffsrechte und Kapselung** kennen – das vielleicht wichtigste Schutzkonzept der gesamten objektorientierten Programmierung.

# 5 Zugriffsrechte und Kapselung

Im letzten Kapitel haben wir gelernt, wie wir unsere Klassen in Paketen organisieren. Wir haben Ordnung in unser Projekt gebracht. Doch eine geordnete Werkstatt allein reicht nicht aus, wenn jeder, der durch die Tür kommt, ungefragt an unseren Maschinen herumschrauben darf. Genau dieses Problem haben wir aktuell in unserem Code: Jede Klasse kann beliebig auf die Attribute unserer Monster zugreifen und diese verändern. Es ist an der Zeit, unsere Architektur mit Schlössern auszurüsten.

## 5.1 Kapselung und das Geheimnisprinzip

Erinnern Sie sich an unsere Monster-Klasse? Wir haben dem Monster eine Methode schadenNehmen(int schaden) gegeben, damit es seine Lebenspunkte selbstständig und kontrolliert reduzieren kann. Das war ein toller Schritt!

Das Problem ist jedoch: Nichts hindert einen anderen Programmierer (oder uns selbst an einem unkonzentrierten Tag) daran, diese Methode einfach zu ignorieren und direkt auf das Attribut zuzugreifen:

```
package haupt;
import charaktere.Monster;

public class Spiel {
    public static void main(String[] args) {
        Monster boss = new Monster("Drache", 5000);

        // Der gedachte Weg:
        boss.schadenNehmen(100);

        // Der katastrophale Weg (aktuell leider erlaubt!):
        boss.lebenspunkte = -999999;
    }
}
```

Wenn wir den zweiten Weg wählen, umgehen wir alle Schutzmechanismen, die wir vielleicht in der Methode schadenNehmen eingebaut haben (z.B. die Prüfung, ob das Monster bereits tot ist oder ob es einen Schild trägt). Das direkte Manipulieren von Attributen von außen ist die Hauptursache für fehlerhafte, schwer zu wartende Programme.

Die Lösung für dieses Problem ist das **Geheimnisprinzip** (Information Hiding) und die daraus resultierende **Kapselung** (Encapsulation).

Als Architekt entwerfen Sie eine Klasse wie einen Tresor oder einen modernen Kaffeeautomaten. Der Anwender des Kaffeeautomaten (das Hauptprogramm) sieht nur die Knöpfe auf der Außenseite (die Methoden). Er drückt auf "Kaffee machen". Wie die Maschine

intern das Wasser erhitzt, die Bohnen mahlt und den Druck aufbaut, ist ihr **Geheimnis**. Der Anwender darf niemals direkt in die Schläuche der Maschine (die Attribute) greifen!

Um das in Java umzusetzen, müssen wir dem Compiler mitteilen, welche Teile unserer Klasse öffentlich zugänglich (die Knöpfe) und welche streng geheim (die Schläuche) sind. Dafür nutzen wir **Zugriffsmodifizierer**.

## 5.2 Zugriffsrechte für Attribute und Methoden

Java bietet vier Stufen von Zugriffsrechten, um festzulegen, wer auf ein Attribut oder eine Methode zugreifen darf. Wir schreiben diese Modifizierer direkt vor die Deklaration.

## 1. private (Streng geheim)

Wenn wir ein Attribut oder eine Methode als private markieren, darf sie **nur und ausschließlich innerhalb derselben Klasse** verwendet werden. Niemand von außen, nicht einmal Klassen aus demselben Paket, kann sie sehen oder verändern.

*Die goldene Regel der Objektorientierung lautet:* **Ab heute machen wir JEDES Attribut private!**

## 2. public (Komplett öffentlich)

Das genaue Gegenteil. Ein public Attribut oder eine public Methode darf von absolut jeder anderen Klasse im gesamten Projekt aufgerufen werden, egal in welchem Paket sie liegt. Unsere Methoden, die von außen nutzbar sein sollen (wie bruellen() oder schadenNehmen()), machen wir public.

Lassen Sie uns unsere Monster-Klasse absichern:

```
package charaktere;

public class Monster {
    // 1. Attribute sind jetzt PRIVATE! Niemand von außen kann sie mehr lesen
    // oder ändern.
    private String name;
    private int lebenspunkte;

    // 2. Der Konstruktor muss PUBLIC sein, sonst darf niemand ein Monster
    // erschaffen.
    public Monster(String name, int lebenspunkte) {
        this.name = name;
        this.lebenspunkte = lebenspunkte;
    }

    // 3. Diese Methode ist die öffentliche Schnittstelle (PUBLIC)
```

```
    public void schadenNehmen(int schaden) {
        if (schaden < 0) {
            System.out.println("Fehler: Negativer Schaden nicht erlaubt!");
            return; // Wir brechen ab, Schutzmechanismus greift!
        }
        this.lebenspunkte = this.lebenspunkte - schaden;
    }

    // 4. Eine PRIVATE Hilfsmethode. Nur das Monster selbst darf sie intern
    // nutzen!
    private void berechneVerzweiflung() {
        // Interne, komplexe Logik, die niemanden außerhalb der Klasse etwas
        // angeht.
    }
}
```

Wenn Sie nun in Ihrem Hauptprogramm versuchen, boss.lebenspunkte = 0; zu schreiben, wird Ihr Programm gar nicht erst starten. Der Compiler wirft einen Fehler: *"lebenspunkte has private access in Monster"*. Wir haben unsere Daten erfolgreich gekapselt!

## Getter und Setter (Zugriffsmethoden)

Aber Moment! Was ist, wenn unser Hauptprogramm auf dem Bildschirm anzeigen möchte, wie viele Lebenspunkte das Monster noch hat? Es darf das Attribut ja nicht mehr lesen.

Die Lösung: Wir bauen kontrollierte, öffentliche Lese- und Schreibmethoden. Man nennt diese **Getter** und **Setter**.

```
public class Monster {
    private int lebenspunkte;

    // Ein Getter: Erlaubt der Außenwelt, den Wert zu LESEN (aber nicht zu
    // ändern)
    public int getLebenspunkte() {
        return this.lebenspunkte;
    }

    // Ein Setter: Erlaubt der Außenwelt, den Wert zu ÄNDERN (aber
    // kontrolliert!)
    // Oft lassen wir Setter für lebenspunkte weg und nutzen stattdessen
    // 'schadenNehmen' oder 'heilen', da das mehr Sinn ergibt. Für z.B. einen
    // Namen könnte ein Setter so aussehen:
    public void setName(String neuerName) {
        if (neuerName != null && !neuerName.isEmpty()) {
            this.name = neuerName;
        }
    }
}
```

Nun sieht der korrekte Aufruf im Hauptprogramm so aus: System.out.println(boss.getLebenspunkte());.

## 3. Default (Paket-privat)

Erinnern Sie sich, dass wir im ersten Kapitel gar kein public oder private vor unsere Attribute geschrieben haben? Wenn Sie den Modifizierer einfach weglassen, verwendet Java das "Default"-Zugriffsrecht (auch "package-private" genannt).

Das bedeutet: Das Attribut/die Methode ist für alle Klassen sichtbar, die im **selben Paket** liegen. Für Klassen aus anderen Paketen ist es unsichtbar.

## 4. protected (Für die Familie)

Es gibt noch einen vierten Modifizierer: protected. Er funktioniert fast wie das Default-Recht (Sichtbarkeit im selben Paket), erlaubt aber zusätzlich den Zugriff durch sogenannte *Unterklassen*, selbst wenn diese in einem anderen Paket liegen. Was genau Unterklassen sind, werden wir im nächsten großen Teil des Buches (Vererbung) ausführlich besprechen.

## 5.3 Zugriffsrechte für Klassen

Nicht nur Attribute und Methoden brauchen Zugriffsrechte, sondern auch die Klassen selbst. Wenn Sie eine Klasse schreiben, müssen Sie entscheiden: Soll diese Klasse im gesamten Projekt nutzbar sein, oder ist sie nur eine kleine Hilfsklasse für mein aktuelles Paket?

Für Klassen gibt es nur zwei sinnvolle Modifizierer: public und Default (kein Modifizierer).

## Public-Klassen:

Wenn eine Klasse public ist (z.B. public class Monster), kann sie aus jedem anderen Paket per import importiert werden.

*Wichtige Regel für Java:* Wenn eine Klasse public ist, **muss** die Datei exakt so heißen wie die Klasse (z.B. Monster.java). Es darf pro Datei immer nur maximal eine einzige public Klasse geben!

## Paket-private Klassen (Default):

Wenn Sie das public weglassen (class GeheimeHilfsklasse), dann kann diese Klasse nur von anderen Klassen im exakt selben Paket benutzt werden. Ein import aus einem anderen Paket ist verboten! Das ist extrem nützlich, wenn Sie komplexe Pakete bauen und verhindern wollen, dass andere Programmierer Ihre internen Hilfsklassen benutzen.

```java
package charaktere;

// Darf von ÜBERALL importiert und genutzt werden
public class Held {
    private Waffe meineWaffe;
    // ...
}

// Darf NUR von anderen Klassen im Paket "charaktere" gesehen werden.
// Die Klasse "Spiel" im Paket "haupt" weiß nicht einmal, dass diese Klasse
// existiert!
class InterneBerechnungsEngine {
    // ...
}
```

## 5.4 Abstrakte Datentypen (ADT)

Mit der konsequenten Anwendung von private für Attribute und public für Methoden erschaffen wir sogenannte **Abstrakte Datentypen (ADT)**. Ein ADT ist die theoretische Krönung der Kapselung.

Ein Abstrakter Datentyp trennt strikt zwischen zwei Dingen:

1. **Der Schnittstelle (Was tut das Objekt?):** Das sind alle public Methoden. Sie bilden den "Vertrag" mit der Außenwelt.
2. **Der Implementierung (Wie macht es das?):** Das sind alle private Attribute und private Hilfsmethoden.

## Der gewaltige Architektur-Vorteil:

Wenn Sie einen perfekten ADT gebaut haben, können Sie die interne Implementierung (das "Wie") in der Zukunft komplett verändern, umschreiben oder optimieren, **ohne** dass auch nur eine einzige andere Klasse in Ihrem Projekt davon etwas merkt oder angepasst werden muss!

Stellen Sie sich vor, wir speichern das Inventar unseres Helden aktuell in drei verschiedenen Variablen (waffe1, waffe2, waffe3). Das Hauptprogramm nutzt nur die Methode public void waffeAufnehmen(Waffe w).

In einem halben Jahr lernen wir (in Buch 1!), was Arrays sind, und wollen die drei Variablen durch ein schickes Array private Waffe[] rucksack; ersetzen.

Da die Variablen private waren und das Hauptprogramm nur die öffentliche Methode genutzt hat, können wir den Code innerhalb des Monsters umschreiben. Das Hauptprogramm wird weiterhin einfach waffeAufnehmen(w) aufrufen und muss nicht mit einer einzigen Codezeile geändert werden!

Das ist die wahre Macht der Objektorientierung. Wir bauen Systeme aus schwarzen Boxen, die über klar definierte Verträge (public Methoden) miteinander sprechen, während das chaotische Innenleben (private Attribute) sicher verborgen bleibt.

---

Herzlichen Glückwunsch! Sie haben den zweiten Teil des Buches abgeschlossen. Sie beherrschen nun die Werkzeuge, um Klassen zu entwerfen, zu instanziieren, zu organisieren und abzusichern. Sie haben solide, in sich geschlossene Bausteine erschaffen.

Doch bisher stehen unsere Klassen – der Held, das Monster, die Waffe – mehr oder weniger unverbunden nebeneinander. Im kommenden Teil III betreten wir das faszinierendste Gebiet der OOP: **Die Vererbung**. Wir werden lernen, wie Klassen Wissen voneinander erben können, wie aus einem allgemeinen "Monster" spezielle "Orks" und "Drachen" entstehen, ohne dass wir Code doppelt schreiben müssen.

# Teil III: Familienbande und Verträge (Vererbung & Abstraktion)

# 6 Vererbung – Wissen weitergeben

Als professioneller Software-Architekt haben Sie ein oberstes Gebot, das über allem steht: **DRY – Don't Repeat Yourself** (Wiederhole dich nicht). Jeder Code, den Sie doppelt schreiben, ist ein potenzieller Fehlerherd. Wenn Sie in einem halben Jahr einen Fehler in diesem Code finden, müssen Sie ihn an dutzenden Stellen gleichzeitig korrigieren. Vergessen Sie nur eine einzige Stelle, haben Sie einen Bug im System.

Bisher haben wir unsere Klassen als isolierte Inseln betrachtet. In diesem Kapitel lernen wir, wie wir Klassen miteinander verwandt machen können, sodass sie Wissen und Fähigkeiten (Attribute und Methoden) an ihre "Kinder" vererben.

## 6.1 Motivation & Definitionen (Ober- und Unterklassen)

Stellen Sie sich vor, unser Dungeon-Spiel wächst. Wir wollen nicht mehr nur ein generisches Monster haben, sondern spezifische Feinde: einen Ork, einen Goblin und einen Drache.

Ohne Vererbung (der "Copy-Paste-Handwerker-Weg") würde das so aussehen:

```
// Der schlechte Weg: Code-Duplizierung
public class Ork {
    private String name;
    private int lebenspunkte;
    private int wutLevel; // Spezifisch für den Ork

    public void schadenNehmen(int schaden) { ... }
    public void wutAufbauen() { ... }
}

public class Drache {
    private String name;
    private int lebenspunkte;
    private int feuerAtemKraft; // Spezifisch für den Drachen

    public void schadenNehmen(int schaden) { ... }
    public void fliegen() { ... }
}
```

Fällt Ihnen etwas auf? Wir haben die Attribute name und lebenspunkte sowie die Methode schadenNehmen exakt kopiert. Wenn wir nun beschließen, dass *jedes* Lebewesen in unserem Spiel auch eine X- und Y-Koordinate braucht, müssen wir alle Klassen einzeln anfassen und anpassen. Ein absoluter Albtraum für die Wartbarkeit!

Die objektorientierte Lösung ist es, die gemeinsamen Eigenschaften herauszufiltern und in einer **Oberklasse** (Superklasse / Basisklasse) zu sammeln. Die spezifischen Klassen werden dann zu

**Unterklassen** (Subklassen), die von der Oberklasse erben.

Es entsteht eine sogenannte **"Ist-ein"-Beziehung** (Is-a-Relationship):

- Ein Ork *ist ein* Monster.
- Ein Drache *ist ein* Monster.

## 6.2 Implementierung mit extends

Um in Java eine Vererbungshierarchie aufzubauen, nutzen wir bei der Definition der Unterklasse das Schlüsselwort extends (übersetzt: "erweitert").

Lassen Sie uns das Architektur-Problem von oben elegant lösen:

```
// 1. Die Oberklasse: Hier liegt das GEMEINSAME Wissen
public class Monster {
    // protected statt private, damit die Kinder darauf zugreifen dürfen!
    protected String name;
    protected int lebenspunkte;

    public Monster(String name, int lebenspunkte) {
        this.name = name;
        this.lebenspunkte = lebenspunkte;
    }

    public void schadenNehmen(int schaden) {
        this.lebenspunkte -= schaden;
        System.out.println(this.name + " verliert " + schaden + " HP.");
    }
}
```

*(Architektur-Hinweis: Erinnern Sie sich an Kapitel 5? Wenn wir private nutzen würden, wäre das Attribut für die Unterklasse unsichtbar. Mit protected öffnen wir das Schloss exklusiv für unsere Familienmitglieder!)*

```
// 2. Die Unterklasse: Sie erbt alles vom Monster und ERWEITERT es
public class Ork extends Monster {
    // Ork muss 'name' und 'lebenspunkte' NICHT neu definieren. Er hat sie
    // bereits!
    private int wutLevel;

    public Ork(String name, int lebenspunkte, int wutLevel) {
        // Achtung: Wie wir den Konstruktor hier genau bauen,
        // klären wir im übernächsten Abschnitt!
        super(name, lebenspunkte);
        this.wutLevel = wutLevel;
    }
```

```
    // Eine Methode, die NUR der Ork kann
    public void wutAufbauen() {
        this.wutLevel += 10;
        System.out.println(this.name + " wird wütend! Wut: " +
                           this.wutLevel);
    }
}
```

Wenn wir nun im Hauptprogramm einen Ork erschaffen, hat dieser magischerweise Zugriff auf die Methode schadenNehmen(), obwohl das Wort "schadenNehmen" in der Klasse Ork gar nicht vorkommt! Der Compiler leitet die Anfrage einfach an die Oberklasse weiter.

```
Ork meinOrk = new Ork("Grumsh", 100, 0);
// Die Methode aus der Monster-Klasse wird aufgerufen:
meinOrk.schadenNehmen(20);
// Die spezifische Methode des Orks wird aufgerufen:
meinOrk.wutAufbauen();
```

## 6.3 Besonderheiten in Java (Einfachvererbung)

Ein wichtiges Detail für Java-Architekten: In Java gilt das Prinzip der **Einfachvererbung**.

Das bedeutet: Eine Klasse darf immer nur **exakt eine** direkte Oberklasse haben. Das Schlüsselwort extends darf nur von einem einzigen Klassennamen gefolgt werden. class Ork extends Monster, Krieger ist in Java streng verboten!

Warum? Stellen Sie sich vor, sowohl das Monster als auch der Krieger hätten eine Methode namens kaempfen(). Wenn der Ork nun von beiden erben würde und wir meinOrk.kaempfen() aufrufen – welche Methode soll Java ausführen? Um dieses sogenannte "Diamond-Problem" zu vermeiden, hat sich Java für die sichere, saubere Einfachvererbung entschieden. (Wie wir dieses scheinbare Limit später mit "Interfaces" umgehen, lernen Sie in Kapitel 10).

Was jedoch erlaubt ist, sind beliebig tiefe Vererbungsketten: Ein BossOrk darf von Ork erben, und der Ork erbt von Monster. Der BossOrk hat dann alles Wissen von beiden!

## 6.4 Konstruktoren und das Schlüsselwort super

Vielleicht ist Ihnen im Ork-Beispiel oben das Schlüsselwort super aufgefallen. Dies ist einer der häufigsten Stolpersteine für Anfänger, also schauen wir ganz genau hin.

Wenn wir ein new Ork(...) erschaffen, baut die Java Virtual Machine (JVM) das Objekt im Speicher. Da ein Ork aber im Kern auch ein Monster ist, muss das Objekt auch die Eigenschaften eines Monsters bekommen. Die Regel lautet: **Bevor ein Kind gebaut werden kann, muss erst das Fundament der Eltern stehen.**

Das bedeutet: Der Konstruktor der Unterklasse **muss zwingend** als allererste Amtshandlung den Konstruktor der Oberklasse aufrufen! Dafür nutzen wir das Schlüsselwort super(). Es verhält sich wie ein Methodenaufruf, der den Konstruktor der Superklasse auslöst.

```
public class Ork extends Monster {
    private int wutLevel;

    public Ork(String name, int lebenspunkte, int wutLevel) {
        // MUSS die absolut erste Zeile sein!
        // Wir reichen "name" und "lebenspunkte" nach oben an das Monster
        // weiter, damit die Monster-Klasse sich um deren Initialisierung
        // kümmert.
        super(name, lebenspunkte);

        // Erst wenn das Monster fertig gebaut ist, machen wir den
        // Ork-spezifischen Teil:
        this.wutLevel = wutLevel;
    }
}
```

Wenn Sie super() weglassen, versucht Java heimlich und unsichtbar super() (ohne Parameter) aufzurufen. Wenn Ihre Monster-Klasse aber gar keinen leeren Default-Konstruktor hat, wirft der Compiler sofort einen Fehler!

## 6.5 Überschreiben von Methoden (@Override)

Manchmal vererbt die Oberklasse eine Fähigkeit, die für die Unterklasse eigentlich fast richtig ist, aber eben nicht ganz. Ein generisches Monster brüllt vielleicht einfach "WAAAGH!". Ein Drache hingegen sollte beim Brüllen auch Feuer spucken.

Wir wollen der Drache-Klasse nicht eine neue Methode drachenBruellen() geben, denn das Hauptprogramm soll einfach zu jedem Monster sagen können: "Brüll!". Wir wollen die geerbte Methode austauschen. Das nennt man **Überschreiben (Overriding)**.

Um eine Methode zu überschreiben, definieren wir in der Unterklasse eine Methode mit dem **exakt gleichen Namen, den gleichen Parametern und dem gleichen Rückgabetyp** wie in der Oberklasse.

```
public class Monster {
    public void bruellen() {
        System.out.println("Das Monster brüllt schauerlich!");
    }
}

public class Drache extends Monster {

    // Die Annotation @Override ist ein Sicherheits-Feature.
```

```
    // Sie sagt dem Compiler: "Ich beabsichtige hier, eine geerbte Methode zu
    // überschreiben!"
    @Override
    public void bruellen() {
        System.out.println("Der Drache brüllt und spuckt einen Feuerball!");
    }
}
```

Wenn wir nun meinDrache.bruellen() aufrufen, wird die Methode der Monster-Klasse komplett ignoriert und die spezifische Drachen-Version ausgeführt.

*Warum nutzen Profis @Override?* Angenommen, Sie vertippen sich und schreiben public void brüllen(). Ohne @Override denkt der Compiler einfach: "Aha, der Drache bekommt eine völlig neue, zusätzliche Methode". Das Monster-Brüllen bleibt bestehen. Der Fehler fiele erst zur Laufzeit auf. Mit @Override überprüft der Compiler streng, ob in der Oberklasse wirklich eine Methode existiert, die exakt so aussieht. Falls nicht, gibt es einen schützenden Compilerfehler!

## 6.6 Die Ur-Klasse java.lang.Object (Das universelle Protokoll)

In der Java-Architektur gibt es ein ehernes Gesetz: **Es gibt keine Waisen.** Jedes einzelne Objekt, das jemals in einer Java-Maschine das Licht der Welt erblickt, hat eine Herkunft.

Wenn Sie eine Klasse programmieren und das Schlüsselwort extends weglassen (wie wir es bei unserem allerersten Monster getan haben), greift der Compiler heimlich ein. Er fügt im Hintergrund automatisch extends java.lang.Object hinzu.

Die Klasse Object ist die kosmische Ur-Klasse, die Mutter aller Klassen. Da Vererbung transitiv ist (der Ork erbt vom Monster, das Monster erbt von Object), erbt ausnahmslos *jede* Klasse in Java von Object.

Warum haben die Erfinder von Java das getan? Weil sie sicherstellen wollten, dass **jedes** Objekt auf der Welt ein minimales Set an grundlegenden Fähigkeiten (ein universelles Protokoll) besitzt. Wir schauen uns nun das komplette Repertoire an Methoden an, das uns die Ur-Klasse Object vererbt.

### Die Textrepräsentation: toString()

Wenn Sie versuchen, ein Objekt direkt auf der Konsole auszugeben (System.out.println(meinOrk);), ruft Java im Hintergrund automatisch die Methode toString() des Objekts auf.

Die Standard-Implementierung aus der Klasse Object liefert jedoch oft kryptischen Kauderwelsch: Ork@7a81197d (den Klassennamen und eine Speicheradresse).

*Als Architekt sollten Sie toString() in fast jeder Ihrer Klassen überschreiben (@Override), um*

*einen sauberen, lesbaren Text (z.B. "Böser Ork (50 HP)") zurückzugeben!*

## Der echte Vergleich: equals(Object obj)

Erinnern Sie sich an den Vergleichsoperator ==? Bei primitiven Zahlen prüft er den Wert (5 == 5). Bei Objekten prüft er jedoch nur die **Identität** (die Speicheradresse)! Er fragt: *"Sind diese beiden Variablen exakt dasselbe physische Objekt im Arbeitsspeicher?"*

Oft wollen wir aber die **Gleichheit** prüfen: *"Haben diese beiden unterschiedlichen Ork-Objekte zufällig dieselben Lebenspunkte und denselben Namen?"*

Dafür existiert equals(). Die Ur-Methode in Object nutzt intern leider nur ein simples ==. Wenn Sie also echte Objekt-Gleichheit wollen, müssen Sie equals in Ihrer Klasse zwingend überschreiben und die Attribute vergleichen (wie wir es in Kapitel 15 bei den Records gesehen haben, wo der Compiler das glücklicherweise automatisch für uns tut).

## Der Hash-Wert: hashCode()

Diese Methode liefert eine ganze Zahl (int) zurück. Sie ist wie ein numerischer Fingerabdruck des Objekts.

Hier gibt es einen eisernen, architektonischen Vertrag in Java: **Wenn Sie equals() überschreiben, müssen Sie zwingend auch hashCode() überschreiben!** Wenn zwei Objekte laut equals() gleich sind, *müssen* sie denselben Hash-Wert liefern.

Warum? Weil Hochleistungs-Datenstrukturen wie das HashSet oder die HashMap (siehe Kapitel 17) diesen Fingerabdruck nutzen, um Objekte blitzschnell in ihren internen Regalen wiederzufinden.

## Die Selbstreflexion: getClass()

Manchmal hält ein Programm eine Referenzvariable vom Typ Object in der Hand und fragt sich: *"Was für ein Objekt bist du eigentlich wirklich?"* Die Methode getClass() liefert ein spezielles Metadaten-Objekt (vom Typ Class) zurück, das den genauen, zur Laufzeit ermittelten Bauplan des Objekts enthält. Das ist das Fundament der sogenannten *Reflection-API*, mit der Programme sich selbst zur Laufzeit analysieren können.

## Das Klonen: clone()

Diese Methode existiert, um eine exakte Kopie (einen Klon) des aktuellen Objekts im Arbeitsspeicher zu erschaffen. Aus Sicherheitsgründen ist sie standardmäßig blockiert (sie wirft eine Exception). Wenn Ihre Klasse das Klonen erlauben soll, muss sie erst das leere Interface Cloneable implementieren und die Methode freischalten. *(In der modernen Java-Architektur gilt*

*clone() jedoch als veraltet und fehleranfällig; man nutzt heute lieber Copy-Konstruktoren).*

## Die Kommunikation zwischen Threads: wait(), notify(), notifyAll()

Diese drei Methoden wirken auf den ersten Blick völlig deplatziert. Warum kann jedes Objekt "warten" oder jemanden "benachrichtigen"?

Die Antwort führt uns direkt zu unserem vierten Buch (Parallele Programmierung). Wenn hunderte Prozesse (Threads) gleichzeitig laufen, fungiert in Java *jedes* Objekt als potenzielles Ampelsystem (ein sogenannter Monitor), um den Verkehr im Arbeitsspeicher zu regeln. Ein Thread kann an einem Objekt wait() aufrufen und schlafen gehen, bis ein anderer Thread an demselben Objekt notify() ruft, um ihn aufzuwecken. Diese Methoden sind ebenfalls final.

## Der Friedhof der Objekte: finalize()

Wenn ein Objekt im Arbeitsspeicher nicht mehr benötigt wird (weil keine Variable mehr darauf zeigt), kommt der automatische Müllsammler von Java (der Garbage Collector) und vernichtet es. Kurz bevor das Objekt endgültig gelöscht wird, rief der Garbage Collector früher als letzten Wunsch die Methode finalize() auf.

*(Architektur-Warnung: Diese Methode wurde in Java 9 offiziell als veraltet (deprecated) markiert und wird in modernen Versionen nicht mehr genutzt, da sie den Arbeitsspeicher massiv ausbremst. Man nutzt heute Konstrukte wie Try-with-Resources aus Kapitel 11.9).*

## 6.7 Das Schlüsselwort final

Wir schließen das Kapitel mit einem Gegenpol zur Vererbung. Manchmal wollen Sie als Architekt bewusst *verhindern*, dass Wissen weitergegeben oder überschrieben wird. Hier hilft das Schlüsselwort final (endgültig). Wir haben es in Kapitel 3 schon bei Variablen kennengelernt (Konstanten). Es hat aber auch Auswirkungen auf Methoden und Klassen:

## Finale Klassen:

Wenn Sie eine Klasse als final markieren, drehen Sie ihr gewissermaßen den Fortpflanzungshahn zu. Niemand darf von dieser Klasse erben.

```
public final class Endboss extends Monster { ... }
// public class SuperEndboss extends Endboss { ... } --> COMPILERFEHLER!
```

*(Wussten Sie, dass die berühmte Klasse String in Java final ist? Sie können keinen MeinBessererString extends String bauen! Die Entwickler von Java wollten sicherstellen, dass Strings absolut manipulationssicher bleiben).*

## Finale Methoden:

Wenn Sie eine Methode als final markieren, darf sie vererbt, aber von den Unterklassen **nicht überschrieben** werden.

```
public class Monster {
    // Jedes Monster darf sterben, aber der Vorgang darf niemals abgeändert
    // werden!
    public final void sterben() {
        System.out.println("Das Lebewesen zerfällt zu Staub.");
    }
}
```

---

Sie haben nun gelernt, wie Sie durch Vererbung extrem effizienten, wartbaren und strukturierten Code schreiben können. Sie definieren allgemeines Wissen oben (Monster) und spezielles Wissen unten (Ork, Drache).

Doch der wirkliche Clou der Vererbung – und der Moment, in dem die Objektorientierung wie echte Magie wirkt – steht uns erst noch bevor. Was passiert, wenn wir eine Variable vom Typ Monster haben, aber einen Drache hineinlegen?

Genau darum kümmern wir uns im nächsten Kapitel: **Polymorphie**.

# 7 Polymorphie

Im vorherigen Kapitel haben wir gelernt, wie wir durch Vererbung Hierarchien aufbauen. Ein Ork und ein Drache erben beide von der Klasse Monster. Wir haben damit doppelten Code vermieden. Doch der wahre architektonische Geniestreich der Vererbung zeigt sich erst jetzt.

Das Wort **Polymorphie** stammt aus dem Griechischen und bedeutet „Vielgestaltigkeit“. In der Programmierung beschreibt es die faszinierende Fähigkeit einer Variablen, zur Laufzeit des Programms unterschiedliche Formen anzunehmen.

## 7.1 Definition & Prinzip (Zuweisungskompatibilität)

Bisher galt in unserer Java-Welt ein eisernes Gesetz: Der Datentyp der Variablen auf der linken Seite des Gleichheitszeichens musste exakt mit dem Datentyp des Objekts auf der rechten Seite übereinstimmen.

```
Monster m = new Monster("Goblin", 30); // Links Monster, rechts Monster.
Drache d = new Drache("Smaug", 5000, 100); // Links Drache, rechts Drache.
```

Durch die Vererbung weicht Java dieses Gesetz nun auf. Die neue Regel der **Zuweisungskompatibilität** lautet: Eine Referenzvariable einer Oberklasse darf jederzeit auf ein Objekt ihrer Unterklasse zeigen!

```
// Magie! Die Variable ist vom Typ Monster, das Objekt im Speicher ist ein
// Drache!
Monster unbekanntesWesen = new Drache("Smaug", 5000, 100);
```

Warum erlaubt der Java-Compiler das? Weil die "Ist-ein"-Beziehung greift. Jeder Drache *ist zwingend auch ein* Monster (denn er erbt davon). Der Compiler weiß: Egal, was dieses Objekt im Speicher genau ist, es hat auf jeden Fall einen Namen, Lebenspunkte und die Methode schadenNehmen(). Es erfüllt alle Anforderungen, die wir an ein generisches Monster stellen.

*Architektur-Metapher:* Stellen Sie sich die Variable unbekanntesWesen als eine Universal-Fernbedienung vor, die nur die grundlegenden Knöpfe für einen Fernseher hat (An/Aus, Lautstärke). Das Objekt im Speicher ist ein hochmoderner Smart-TV (der Drache). Sie können den Smart-TV problemlos mit der Universal-Fernbedienung bedienen – solange Sie nur die Grundfunktionen nutzen!

## 7.2 Protokolleinschränkung bei Objektvariablen

Wenn wir einen Drache in einer Monster-Variablen speichern, stoßen wir jedoch auf eine wichtige Einschränkung. Schauen wir uns folgenden Code an:

```
public class Spiel {
    public static void main(String[] args) {
        // Wir packen einen Drachen in eine Monster-Variable
        Monster boss = new Drache("Smaug", 5000, 100);
```

```
        // Das funktioniert einwandfrei (wurde von Monster geerbt):
        boss.schadenNehmen(50);

        // COMPILERFEHLER!
        boss.fliegen();
    }
}
```

Warum wirft der Compiler in der letzten Zeile einen Fehler? Das Objekt im Speicher *ist* doch ein Drache, und Drachen können fliegen()!

Hier müssen Sie als Architekt scharf zwischen der **Referenzvariablen** und dem eigentlichen **Objekt** trennen.

Der Compiler bewertet Ihren Code, *bevor* das Programm überhaupt läuft. Er schaut sich nur die Variable boss an. Der deklarierte Datentyp von boss ist Monster. Die Klasse Monster definiert das Protokoll (den Vertrag), was diese Variable tun darf. Da in der Klasse Monster keine Methode namens fliegen() steht, verbietet der Compiler den Aufruf rigoros.

Man nennt dies die **Protokolleinschränkung**. Die Fernbedienung (Monster boss) hat einfach keinen Knopf für "Fliegen", obwohl der Fernseher (das Drache-Objekt) die Funktion theoretisch beherrschen würde.

## 7.3 Implizite Typumwandlung (Upcast)

Die automatische Umwandlung von unten nach oben im Vererbungsbaum (vom speziellen Drachen zum allgemeinen Monster) nennt man **Upcast**. Er geschieht **implizit**, also automatisch und unsichtbar im Hintergrund, weil er absolut sicher ist.

Wofür brauchen wir das in der Praxis? Der Upcast erlaubt es uns, extrem flexiblen und allgemeingültigen Code zu schreiben.

Stellen Sie sich vor, wir schreiben eine Methode für unseren Helden, mit der er ein beliebiges Wesen angreifen kann. Ohne Polymorphie müssten wir für jede Monster-Art eine eigene Methode schreiben (Überladung):

```
// Der Handwerker-Weg (ohne Polymorphie):
public void angreifen(Ork o) { ... }
public void angreifen(Goblin g) { ... }
public void angreifen(Drache d) { ... }
```

Wenn wir in Zukunft ein Skelett zum Spiel hinzufügen, müssten wir die Klasse des Helden umschreiben. Das verletzt jedes Architektur-Prinzip!

Mit Polymorphie und dem impliziten Upcast lösen wir das elegant mit einer einzigen Methode:

```
public class Held {
    private int angriffsKraft = 20;

    // Die Methode akzeptiert JEDES Objekt, das vom Typ Monster ist
    // oder von Monster erbt!
    public void angreifen(Monster ziel) {
        System.out.println("Der Held greift an!");
        ziel.schadenNehmen(this.angriffsKraft);
    }
}

// Nutzung im Hauptprogramm:
Held meinHeld = new Held();
Ork grumsh = new Ork("Grumsh", 100, 0);
Drache smaug = new Drache("Smaug", 5000, 100);

// Impliziter Upcast beim Methodenaufruf!
// Der Ork und der Drache werden für die Dauer der Methode als "Monster"
// betrachtet.
meinHeld.angreifen(grumsh);
meinHeld.angreifen(smaug);
```

## Das ultimative Beispiel: Polymorphe Arrays

Dank der Polymorphie können wir nun auch Objekte unterschiedlicher (aber verwandter) Klassen in eine gemeinsame Datenstruktur packen:

```
// Wir erschaffen ein Array, das Platz für 3 abstrakte "Monster" bietet.
Monster[] dungeonRaum = new Monster[3];

// Dank Upcast können wir das Array mit völlig verschiedenen Unterklassen
füllen!
dungeonRaum[0] = new Ork("Ork-Krieger", 80, 0);
dungeonRaum[1] = new Monster("Gewöhnlicher Schleim", 10);
dungeonRaum[2] = new Drache("Feuerdrache", 2000, 50);

// Wir können mit einer einzigen Schleife alle Feinde im Raum bearbeiten!
for (int i = 0; i < dungeonRaum.length; i++) {
    // Egal was es genau ist, jedes Element ist garantiert ein Monster.
    dungeonRaum[i].schadenNehmen(5);
}
```

Das ist Architektur! Unser Spiel kann hunderte Monster-Arten haben, unsere Schleife bleibt immer gleich.

## 7.4 Explizite Typumwandlung (Typecast / Downcast)

Manchmal geraten wir in eine Situation, in der die Protokolleinschränkung uns im Weg steht. Wir

haben eine Variable vom Typ Monster, wissen aber durch die Spiellogik ganz genau, dass sich darin in Wahrheit ein Drache verbirgt. Wir *müssen* dringend die fliegen()-Methode aufrufen.

Um das zu tun, müssen wir einen **Downcast** (Umwandlung im Vererbungsbaum nach unten) durchführen. Im Gegensatz zum Upcast passiert dieser nicht automatisch, denn er ist potenziell gefährlich. Wir müssen den Compiler **explizit** dazu zwingen.

```
Monster unbekannt = new Drache("Smaug", 5000, 100);

// 1. Die harte (und veraltete) Methode: Der klassische Typecast
// Wir sagen dem Compiler: "Vertrau mir, das ist ein Drache!"
Drache echterDrache = (Drache) unbekannt;

// Jetzt haben wir eine Drachen-Referenz und das Protokoll erlaubt das
// Fliegen!
echterDrache.fliegen();
```

## Die Gefahr des Downcasts:

Was passiert, wenn wir lügen? Wenn in der Variablen unbekannt in Wahrheit ein Ork steckt, wir dem Compiler aber mit (Drache) sagen, er soll es umwandeln?

Der Compiler glaubt uns zunächst und das Programm startet. Doch zur Laufzeit (wenn das Programm tatsächlich ausgeführt wird) merkt die Java Virtual Machine den Betrug. Ein Ork kann kein Drache sein! Das Programm stürzt sofort mit einer gefürchteten Fehlermeldung ab: der ClassCastException.

## Der sichere Weg: instanceof mit Pattern Matching (Java SE 16+)

Als verantwortungsvolle Architekten casten wir niemals blind. Wir prüfen vorher, ob das Objekt wirklich das ist, was wir erwarten. Dafür gibt es den Operator instanceof (ist ein Exemplar von).

In modernen Java-Versionen (wie unserem Java SE 25) ist dieser Operator unfassbar elegant geworden. Wenn die Prüfung erfolgreich ist, erschafft Java vollautomatisch eine neue Variable im richtigen Typ für uns! Man nennt das *Pattern Matching for instanceof*.

```
public class Spiel {
    public static void interagiereMit(Monster wesen) {

        // Wir prüfen: Ist das 'wesen' im Speicher in Wahrheit ein Ork?
        // Wenn JA: Erschaffe sofort eine neue Ork-Variable namens 'o'
        // und fülle sie mit dem Objekt!
        if (wesen instanceof Ork o) {
            System.out.println("Ein Ork steht vor dir!");
            o.wutAufbauen(); // Protokoll von 'o' erlaubt Ork-Methoden!
        }
        else if (wesen instanceof Drache d) {
```

```
            System.out.println("Achtung, ein Drache!");
            d.fliegen(); // Protokoll von 'd' erlaubt Drachen-Methoden!
        }
        else {
            System.out.println("Es ist nur ein gewöhnliches Monster.");
            wesen.schadenNehmen(10);
        }
    }

    public static void main(String[] args) {
        Monster m1 = new Ork("Grumsh", 100, 0);
        Monster m2 = new Drache("Smaug", 5000, 100);

        interagiereMit(m1); // Löst den Ork-Block aus
        interagiereMit(m2); // Löst den Drachen-Block aus
    }
}
```

Mit instanceof und dem Pattern Matching haben Sie das ultimative Werkzeug, um sicher im Vererbungsbaum nach unten zu navigieren, wenn es sich absolut nicht vermeiden lässt.

*(Goldene Architektur-Regel: Ein exzessiver Gebrauch von instanceof ist oft ein Zeichen für schlechtes Klassendesign. Meistens ist es besser, das gewünschte Verhalten über überschriebene Methoden zu lösen – wie wir im nächsten Kapitel sehen werden!)*

---

Sie haben nun verstanden, wie Polymorphie funktioniert. Sie können Objekte in allgemeineren Variablen speichern und so flexiblen, zukunftssicheren Code schreiben.

Doch es fehlt noch ein Puzzleteil. Wir haben gesehen, dass wir Methoden überschreiben können (z.B. bruellen()), und wir haben gesehen, dass wir Referenzen hoch- und runtercasten können. Was passiert eigentlich, wenn wir beides kombinieren? Welches bruellen() wird aufgerufen, wenn die Variable ein Monster ist, das Objekt aber ein Drache?

Dieses Geheimnis, das Herzstück der objektorientierten Laufzeitumgebung, lüften wir im nächsten Kapitel: **Dynamisches Binden**.

# 8 Dynamisches Binden

Im letzten Kapitel haben wir unsere Architektur mit Polymorphie flexibel gemacht. Wir haben gelernt, dass eine Referenzvariable vom Typ Monster problemlos einen Drache speichern kann. Wir wissen auch, dass der Compiler penibel darauf achtet, dass wir nur Methoden aufrufen, die im Protokoll der Monster-Klasse stehen (Protokolleinschränkung).

Doch eine entscheidende Frage ist bisher unbeantwortet geblieben. Erinnern Sie sich an das Überschreiben von Methoden (@Override) aus Kapitel 6? Das generische Monster brüllt "WAAAGH!", der Drache hingegen brüllt und spuckt Feuer.

Was passiert nun bei folgendem Code?

```
Monster unbekannt = new Drache("Smaug", 5000);
unbekannt.bruellen();
```

Welches Brüllen hören wir? Das Brüllen der Variablen-Klasse (Monster) oder das Brüllen der Objekt-Klasse (Drache)? Die Antwort auf diese Frage führt uns zum elegantesten Mechanismus der Objektorientierung: dem **Dynamischen Binden** (Dynamic Binding).

## 8.1 Definition: Statisches vs. Dynamisches Binden

Um zu verstehen, wie Java arbeitet, müssen wir uns als Architekten klar machen, dass ein Java-Programm zwei völlig unterschiedliche Lebensphasen hat: die **Kompilierzeit** (Compile Time) und die **Laufzeit** (Runtime).

In der imperativen Programmierung (Buch 1) gab es keine Vererbung. Wenn Sie eine Methode aufgerufen haben, wusste der Compiler bereits beim Übersetzen des Textcodes in Maschinencode zu 100 Prozent, welche Methode ausgeführt werden muss. Er hat den Methodenaufruf fest verdrahtet. Diesen Vorgang nennt man **Statisches Binden** (Early Binding).

In der Objektorientierung ist das nicht mehr so einfach. Wenn der Compiler den Text unbekannt.bruellen(); liest, sieht er nur eine Variable vom Typ Monster. Er hat keine Ahnung, was genau in dieser Variablen steckt, wenn das Programm läuft. Es könnte ein normaler Ork sein, ein Drache oder ein völlig neues Monster, das der Spieler erst viel später per Download hinzufügt.

Daher verschiebt Java die Entscheidung, welche Methode *wirklich* ausgeführt wird, in die zweite Lebensphase: die Laufzeit. Erst wenn das Programm tatsächlich ausgeführt wird und das Objekt im Arbeitsspeicher liegt, wird die Methode an das Objekt gekoppelt. Das ist das **Dynamische Binden** (Late Binding).

## 8.2 Prinzip der Methodenwahl zur Laufzeit

Lassen Sie uns die eiserne Regel der Methodenwahl in Java aufstellen. Wenn Sie eine Methode

aufrufen, arbeiten der Compiler (vor dem Start) und die Java Virtual Machine (während des Spiels) als Team zusammen:

1. **Der Compiler prüft das Protokoll (Statisch):** Er schaut *nur* auf den Typ der Referenzvariablen (links vom Gleichheitszeichen). Existiert die Methode dort? Wenn nein: Fehler! Wenn ja: Er gibt grünes Licht.
2. **Die JVM wählt die Implementierung (Dynamisch):** Während das Programm läuft, ignoriert die JVM die Referenzvariable völlig. Sie schaut tief in den Arbeitsspeicher auf das *echte Objekt* (rechts vom Gleichheitszeichen). Sie sucht in der Klasse dieses Objekts nach der Methode.

```
public class Spiel {
    public static void main(String[] args) {
        // 1. Eine normale Zuweisung
        Monster m1 = new Monster("Goblin", 30);
        m1.bruellen(); // Druckt: Das Monster brüllt schauerlich!

        // 2. Die polymorphe Zuweisung
        Monster m2 = new Drache("Smaug", 5000);

        // Compiler fragt: Hat 'Monster' eine Methode bruellen()? Ja. -> OK.
        // JVM fragt zur Laufzeit: Welches Objekt liegt im Speicher? Ein
        // 'Drache'.
        // Führe die bruellen()-Methode der Drache-Klasse aus!
        m2.bruellen(); // Druckt: Der Drache brüllt und spuckt einen
                       // Feuerball!
    }
}
```

Das Objekt im Speicher entscheidet! Es verhält sich immer gemäß seiner wahren Natur, völlig unabhängig davon, in was für eine "Universal-Fernbedienung" (Variable) wir es gestopft haben.

## 8.3 Zusammenspiel mit überschriebenen Methoden

Das dynamische Binden funktioniert auch über beliebig tiefe Vererbungsketten. Die Java Virtual Machine sucht beim dynamischen Binden immer **von unten nach oben**.

Stellen Sie sich folgende Hierarchie vor: Monster vererbt an Ork, und Ork vererbt an BossOrk.

Alle drei Klassen überschreiben die Methode angreifen().

```
Monster m = new BossOrk("Ober-Grumsh", 500, 100);
m.angreifen();
```

Die JVM schaut auf das Objekt im Speicher: Es ist ein BossOrk. Sie fragt:

1. *"Hat die Klasse BossOrk eine eigene angreifen()-Methode?"* * Wenn **Ja**, wird diese ausgeführt. Die Suche endet sofort.

    - Wenn **Nein**, klettert die JVM einen Ast nach oben zur Klasse Ork.
2. *"Hat die Klasse Ork die Methode überschrieben?"*
    - Wenn **Ja**, wird diese ausgeführt.
    - Wenn **Nein**, klettert die JVM weiter nach oben zum Monster.

Die JVM führt immer die **spezifischste** Implementierung aus, die sie finden kann.

## 8.4 Bindung von Attributen und Klassenmethoden (Verdeckung)

Achtung, jetzt betreten wir eine gefährliche Falle! Eine klassische Prüfungsfrage für angehende Architekten und eine häufige Fehlerquelle bei Anfängern.

Das dynamische Binden gilt in Java **ausschließlich für normale Objekt-Methoden** (nicht-statische Methoden).

Für **Attribute** und für **statische Methoden** gibt es *kein* dynamisches Binden! Diese werden immer **statisch gebunden**, also durch den Compiler anhand der Referenzvariablen entschieden.

Schauen wir uns dieses tückische Beispiel an: Wir definieren ein Attribut name in der Oberklasse und aus Versehen noch einmal mit demselben Bezeichner in der Unterklasse. Das nennt man **Verdeckung (Shadowing)**.

```
public class Monster {
    public String name = "Unbekanntes Monster"; // Attribut

    public static void info() {                  // Statische Methode
        System.out.println("Hier spricht die Monster-Klasse!");
    }
}

public class Ork extends Monster {
    // Verdeckung: Wir definieren 'name' unsauber noch einmal!
    public String name = "Böser Ork";

    // Verdeckung der statischen Methode
    public static void info() {
        System.out.println("Hier spricht die Ork-Klasse!");
    }
}
```

Was passiert nun bei einer polymorphen Zuweisung?

```
public class Spiel {
    public static void main(String[] args) {
        Monster wesen = new Ork();
```

```
        // 1. Zugriff auf das Attribut (STATISCH gebunden!)
        // Der Compiler sieht nur die Variable 'wesen' vom Typ 'Monster'.
        System.out.println(wesen.name);
        // Ausgabe: Unbekanntes Monster  <-- SCHOCK! Es ist nicht der
        // Ork-Name!

        // 2. Aufruf der statischen Methode (STATISCH gebunden!)
        wesen.info();
        // Ausgabe: Hier spricht die Monster-Klasse!
    }
}
```

Warum verhält sich Java hier so starr? Weil Attribute und Klassenmethoden keinen Laufzeit-Overhead haben sollen. Der Compiler verknüpft sie unwiderruflich beim Kompilieren.

*Die Architektur-Lehre daraus:* 1. Sie sollten Attribute ohnehin niemals public machen (wir erinnern uns an Kapselung aus Kapitel 5). Wenn Sie saubere get- und set-Methoden nutzen, greift wieder das dynamische Binden und alles funktioniert wie gewünscht!

2. Überschreiben Sie niemals einfach so Attribute der Oberklasse in der Unterklasse.

## 8.5 Architektur-Vorteile des dynamischen Bindens

Warum ist dynamisches Binden so unfassbar wichtig? Warum bezeichnen es viele als das eigentliche Herzstück der Objektorientierung?

Weil es das **Open/Closed Principle** ermöglicht. Dieses Prinzip besagt, dass gute Softwarearchitektur offen für Erweiterungen (Open), aber geschlossen für Modifikationen (Closed) sein muss.

Stellen Sie sich vor, wir haben unser Videospiel fertig programmiert und ausgeliefert. In unserer Game-Engine gibt es einen Raum, der Monster sammelt und sie einmal pro Sekunde brüllen lässt:

```
// Kern-Code der Game-Engine (bereits kompiliert und ausgeliefert)
public void raumUpdate(Monster[] raum) {
    for (int i = 0; i < raum.length; i++) {
        raum[i].bruellen(); // DYNAMISCHES BINDEN!
    }
}
```

Ein Jahr später beschließen wir, eine Erweiterung (DLC) auf den Markt zu bringen. Wir erfinden ein völlig neues Monster: den Nekromant. Wir programmieren eine neue Klasse Nekromant extends Monster und überschreiben bruellen().

Das Geniale: **Wir müssen unseren alten Game-Engine-Code nicht eine einzige Zeile ändern**

**oder neu kompilieren!** Wenn wir ein neues Nekromant-Objekt in das Array packen, wird die JVM zur Laufzeit völlig automatisch die neue, dem alten Code völlig unbekannte Methode ausführen. Der alte Code ruft blind bruellen() auf, und die JVM leitet den Aufruf wie ein intelligenter Telefonist an die richtige, neue Klasse weiter.

In der imperativen Welt (Buch 1) hätten wir eine riesige switch-case-Anweisung gebraucht (case "Nekromant": nekromantBruellen();). Bei jedem neuen Monster hätten wir den alten Code anfassen, erweitern und neu testen müssen. Dank Polymorphie und dynamischem Binden steuern Objekte ihr Verhalten nun völlig autark.

---

Sie sind nun in der Lage, echte, erweiterbare Systeme zu bauen. Sie kennen die Mechanismen der Vererbung, der Polymorphie und der Laufzeitbindung in- und auswendig.

Doch bisher basierte unsere Vererbung immer darauf, dass eine Klasse (Monster) bereits fertigen Code hatte, den sie an andere weitergab. Was ist, wenn wir eine Oberklasse erschaffen wollen, die *gar keinen eigenen Code* hat, sondern nur existiert, um ein gemeinsames Protokoll für ihre Kinder zu erzwingen? Ein Wesen, das so allgemein ist, dass es in der Realität gar nicht existieren darf?

Im nächsten Kapitel erheben wir die Objektorientierung auf ein noch höheres philosophisches Level: Wir lernen **Abstrakte Klassen** kennen.

# 9 Abstrakte Klassen

In den letzten drei Kapiteln haben wir eine mächtige Vererbungshierarchie aufgebaut. Wir haben die Klasse Monster erschaffen, die als Oberklasse für Ork und Drache dient. Dank der Polymorphie und dem dynamischen Binden können wir unser gesamtes Dungeon-Spiel programmieren, indem wir einfach nur gegen die allgemeine Monster-Klasse programmieren.

Doch als kritischer Software-Architekt haben Sie vielleicht schon ein gewisses Unbehagen verspürt, wenn Sie sich unsere bisherige Monster-Klasse genauer angesehen haben.

## 9.1 Motivation: Abstraktion gemeinsamer Oberklassen

Schauen wir uns zwei gravierende konzeptionelle Probleme unserer aktuellen Architektur an:

### Problem 1: Sinnlose Objekte im Speicher

Bisher hindert uns nichts daran, in unserem Hauptprogramm folgenden Code zu schreiben:

```
Monster einWesen = new Monster("Irgendwas", 50);
```

Was genau ist das jetzt für ein Objekt in unserem Spiel? Es ist kein Ork, es ist kein Drache, es ist einfach nur ein "Monster". In der realen Welt (und in einer guten Spielwelt) gibt es so etwas nicht. Wenn Sie in einen Wald gehen, sehen Sie niemals "ein Tier". Sie sehen einen Hund, einen Vogel oder ein Reh. "Tier" ist nur ein abstraktes Konzept, ein Sammelbegriff des menschlichen Verstandes, um Lebewesen mit gemeinsamen Eigenschaften zu gruppieren. Genau wie "Fahrzeug" oder eben "Monster".

Es ergibt absolut keinen Sinn, dass unser Programm es erlaubt, ein physisches Objekt von diesem abstrakten Konzept zu erschaffen.

### Problem 2: Dumme Platzhalter-Methoden

Erinnern Sie sich an die Methode bruellen() in unserer Monster-Klasse?

```
public class Monster {
    public void bruellen() {
        System.out.println("Das Monster brüllt schauerlich!");
    }
}
```

Wir haben diese Methode eigentlich nur geschrieben, damit sie im Protokoll der Klasse steht und wir sie über unsere Universal-Fernbedienung (die Monster-Variable) aufrufen können. Wir wussten ohnehin, dass der Ork und der Drache diese Methode überschreiben werden. Der Code innerhalb der Monster-Methode war also nur ein dummer Platzhalter.

Java bietet uns für genau dieses Problem ein grandioses Werkzeug: das Schlüsselwort abstract.

## 9.2 Definition und Syntax (abstract class)

Wenn wir eine Klasse als abstract markieren, teilen wir dem Compiler Folgendes mit: *"Diese Klasse ist nicht ganz vollständig. Sie ist nur ein theoretisches Konzept. Sie existiert ausschließlich, um gemeinsames Wissen an Unterklassen zu vererben. Verbiete es strengstens, dass jemals ein Objekt direkt aus dieser Klasse erschaffen wird!"*

Lassen Sie uns unsere Monster-Klasse in eine abstrakte Klasse umwandeln:

```
// Das Schlüsselwort 'abstract' steht direkt vor 'class'
public abstract class Monster {
    protected String name;
    protected int lebenspunkte;

    // Auch eine abstrakte Klasse DARF (und sollte!) einen Konstruktor haben.
    // Dieser wird von den Unterklassen über super() aufgerufen.
    public Monster(String name, int lebenspunkte) {
        this.name = name;
        this.lebenspunkte = lebenspunkte;
    }

    // Eine normale (konkrete) Methode.
    // Jedes Unter-Monster erbt sie exakt so.
    public void schadenNehmen(int schaden) {
        this.lebenspunkte -= schaden;
    }
}
```

Sobald wir das tun, ändert sich das Verhalten in unserem Hauptprogramm dramatisch:

```
public class Spiel {
    public static void main(String[] args) {

        // COMPILERFEHLER!
        // "Monster is abstract; cannot be instantiated"
        Monster fehler = new Monster("Blob", 10);

        // KORREKT: Polymorphie funktioniert weiterhin perfekt!
        // Die Variable darf vom Typ Monster sein, das Objekt MUSS eine
        // Unterklasse sein.
        Monster boss = new Drache("Smaug", 5000);
    }
}
```

Wir haben Problem 1 gelöst! Niemand kann mehr versehentlich formlose, generische Monster in unsere Spielwelt setzen.

## Abstrakte Methoden

Nun lösen wir Problem 2 (die dummen Platzhalter-Methoden). Wenn wir in einer abstrakten Klasse wissen, dass *jedes* Monster brüllen können muss, wir aber auf der Ebene des allgemeinen Monsters absolut nicht wissen, *wie* dieses Brüllen aussieht, dann machen wir die Methode selbst abstract.

Eine abstrakte Methode hat **keinen Rumpf** (keine geschweiften Klammern). Sie besteht nur aus der sogenannten Signatur und endet sofort mit einem Semikolon:

```
public abstract class Monster {
    protected String name;

    // ... Konstruktor etc. ...

    // Eine abstrakte Methode! Kein { ... }, nur ein Semikolon.
    public abstract void bruellen();
}
```

Damit diktiert die abstrakte Klasse Monster ein eisernes Gesetz für alle ihre Nachfahren: *"Wer sich rühmen will, ein Monster zu sein, der MUSS eine Methode namens bruellen() besitzen. Ich gebe euch keinen Standard-Code dafür – ihr müsst euch selbst darum kümmern!"*

## 9.3 Nutzung und Grenzen

Was passiert nun, wenn wir eine neue Unterklasse schreiben, zum Beispiel einen Troll, der von der abstrakten Monster-Klasse erbt?

```
public class Troll extends Monster {

    public Troll(String name, int lebenspunkte) {
        super(name, lebenspunkte);
    }

    // Wenn wir den Code jetzt kompilieren, wirft der Compiler einen Fehler!
}
```

Der Compiler streikt mit der Meldung: *"Troll is not abstract and does not override abstract method bruellen() in Monster"*.

Hier greift der geniale Zwang der Architektur:

Wenn eine Klasse (wie Troll) von einer abstrakten Klasse erbt, **muss** sie alle abstrakten Methoden der Oberklasse überschreiben (implementieren). Tut sie das nicht, ist sie selbst unvollständig und der Compiler zwingt uns, auch die Klasse Troll als abstract zu deklarieren!

Um den Troll als echtes, instanziierbares (konkretes) Wesen in unser Spiel zu bringen, müssen wir den Vertrag erfüllen:

```
public class Troll extends Monster {

    public Troll(String name, int lebenspunkte) {
        super(name, lebenspunkte);
    }

    // Wir implementieren die erzwungene Methode!
    @Override
    public void bruellen() {
        System.out.println(this.name + " grunzt tief und bedrohlich.");
    }
}
```

## Das Zusammenspiel mit dem dynamischen Binden

Abstrakte Klassen und Methoden sind der ultimative Beweis für die Kraft des dynamischen Bindens aus Kapitel 8. Schauen wir uns diese Zeilen an:

```
Monster m = new Troll("Höhlentroll", 200);
m.bruellen();
```

Der Compiler prüft beim Kompilieren den Typ der Variablen m. Das ist Monster. Er schaut in die Klasse Monster. Dort findet er die abstrakte Methode public abstract void bruellen();. Der Compiler ist zufrieden – der Knopf auf der Fernbedienung existiert.

Dass die Methode gar keinen Code (keine geschweiften Klammern) hat, stört den Compiler überhaupt nicht! Er weiß ja: Zur Laufzeit wird dynamisch gebunden. Zur Laufzeit wird die JVM auf das Objekt schauen (den Troll) und die Methode *dort* ausführen. Und da der Compiler beim Troll vorher streng erzwungen hat, dass die Methode existiert, kann das Programm niemals abstürzen. Ein perfektes Sicherheitssystem.

## Zusammenfassung der Regeln für abstrakte Klassen:

1. Eine Klasse mit dem Wort abstract kann nicht mit new instanziiert werden.
2. Wenn eine Klasse mindestens eine abstract Methode besitzt, **muss** die gesamte Klasse abstract sein.
3. Eine abstrakte Klasse darf (im Gegensatz zu unserer Annahme ganz zu Beginn) sehr wohl auch normalen, voll ausprogrammierten Code (konkrete Methoden und Attribute) enthalten, den sie ganz normal vererbt.
4. Die erste nicht-abstrakte (konkrete) Unterklasse in der Vererbungshierarchie muss alle geerbten abstrakten Methoden mit Leben füllen.

Mit abstrakten Klassen haben wir gelernt, wie wir das Grundgerüst für Familien von Objekten bauen. Wir erzwingen Verhalten (abstract void bruellen()) und teilen gleichzeitig Code (schadenNehmen()).

Doch in Java gibt es, wie in Kapitel 6 besprochen, die Einfachvererbung. Ein Troll kann immer nur von *einer* abstrakten Klasse erben. Was aber, wenn unser Troll nicht nur ein Monster ist, sondern auch die Fähigkeit besitzen soll, in unserem Inventar gesammelt zu werden (wie ein Item), weil er nach seinem Tod Beute fallen lässt? Er müsste von Monster und von SammelbaresObjekt erben – was Java verbietet.

Die Lösung für dieses Dilemma und die reinste Form der Abstraktion lernen wir im nächsten Kapitel kennen. Wir lassen die klassische Vererbung hinter uns und schließen **Verträge**: Willkommen in der Welt der **Interfaces**.

# 10 Interfaces (Schnittstellen)

Im letzten Kapitel haben wir abstrakte Klassen kennengelernt. Wir haben festgestellt, dass sie hervorragend geeignet sind, um ein Grundgerüst für eng verwandte Familien von Objekten (wie Monster, Ork, Troll) zu bauen. Doch wir stießen auf eine architektonische Mauer: die Einfachvererbung. Ein Java-Objekt darf immer nur von exakt *einer* Oberklasse erben.

## 10.1 Motivation: Schnittstellen als "Verträge"

Stellen Sie sich vor, unser Dungeon-Spiel wird komplexer. Wir haben eine abstrakte Klasse Monster, von der ein Troll erbt. Gleichzeitig haben wir ein Klassensystem für Gegenstände im Spiel: Eine abstrakte Klasse Item, von der Heiltrank und Schwert erben.

Nun wollen wir, dass der Spieler bestimmte besiegte Monster (wie einen kleinen Goblin oder einen magischen Schleim) in sein Inventar aufnehmen kann, um sie später als Begleiter zu beschwören. Wir brauchen also eine Methode aufheben().

Wo definieren wir diese Methode?

- Wenn wir sie in Monster packen, kann der Spieler plötzlich riesige Drachen aufheben. Das ergibt keinen Sinn.
- Wir können das Monster auch nicht von Item erben lassen, denn es erbt ja bereits von Monster (Einfachvererbung).

Wir brauchen eine Möglichkeit, Eigenschaften ("kann aufgehoben werden") völlig unabhängig von der familiären Abstammung (Vererbungshierarchie) zu verteilen. Hier betritt das **Interface** (die Schnittstelle) die Bühne.

Ein Interface ist kein Bauplan für ein Objekt. Ein Interface ist ein rein juristischer **Vertrag**. Es definiert *was* getan werden muss, aber nicht *wie*.

## 10.2 Syntax: Definition und Implementierung (implements)

Ein Interface wird nicht mit dem Wort class, sondern mit dem Schlüsselwort interface definiert.

In seiner klassischsten Form enthält ein Interface ausschließlich **abstrakte Methoden** und Konstanten. Da in einem Interface traditionell ohnehin alles öffentlich und abstrakt ist, erlaubt uns Java, die Wörter public und abstract bei den Methoden einfach wegzulassen.

```
// Datei: Sammelbar.java
public interface Sammelbar {
    // Das ist der Vertrag: Wer "Sammelbar" sein will,
    // MUSS diese Methode anbieten!
    void aufheben();
```

```
    // Konstanten sind in Interfaces immer automatisch public, static und
    // final
    int MAX_GEWICHT = 50;
}
```

Nun wollen wir, dass unser Schleim (ein Monster) und unser Heiltrank (ein Item) diesen Vertrag unterschreiben. Um ein Interface in eine Klasse einzubinden, nutzen wir nicht extends, sondern das Schlüsselwort **implements** (implementiert / erfüllt).

```
public class Schleim extends Monster implements Sammelbar {

    public Schleim(String name, int lebenspunkte) {
        super(name, lebenspunkte);
    }

    @Override
    public void bruellen() {
        System.out.println("Der Schleim blubbert.");
    }

    // Hier erfüllen wir den Vertrag des Interfaces!
    // ACHTUNG: Die Methode MUSS hier zwingend 'public' sein.
    @Override
    public void aufheben() {
        System.out.println("Der Schleim wird in ein Glas gestopft und " +
                           "eingesteckt.");
    }
}
```

Wenn eine Klasse ein Interface implementiert, zwingt der Compiler sie gnadenlos dazu, alle Methoden des Interfaces mit Leben zu füllen (genau wie bei abstrakten Klassen). Tut sie das nicht, gibt es einen Compilerfehler.

## 10.3 Implementierung mehrerer Interfaces

Hier zeigt sich die wahre Befreiung in der Java-Architektur: Während eine Klasse nur eine einzige Oberklasse (extends) haben darf, darf sie **beliebig viele Interfaces** (implements) unterschreiben! Die Verträge werden einfach durch Kommata getrennt.

Lassen Sie uns einen zweiten Vertrag erfinden:

```
public interface Kaempfbar {
    void attackieren();
}
```

Ein KampfSchleim kann nun von Monster erben UND beide Verträge unterschreiben:

```
// Erbt von einer Klasse und implementiert ZWEI Interfaces
public class KampfSchleim extends Monster implements Sammelbar, Kaempfbar {

    public KampfSchleim(String name, int hp) { super(name, hp); }

    @Override
    public void bruellen() { System.out.println("Blubb!"); }

    @Override
    public void aufheben() { System.out.println("Eingesteckt!"); }

    @Override
    public void attackieren() { System.out.println("Schleim-Spucke!"); }
}
```

## 10.4 Polymorphie und dynamisches Binden bei Interfaces

Jetzt wird es magisch. Erinnern Sie sich an Kapitel 7 und 8? Polymorphie und dynamisches Binden funktionieren mit Interfaces exakt genauso wie mit Klassen!

**Ein Interfacename darf als Datentyp für eine Referenzvariable genutzt werden!**

```
public class Spiel {
    public static void main(String[] args) {
        // Die Variable ist vom Typ des Interfaces!
        Sammelbar beute = new KampfSchleim("Gift-Schleim", 20);

        // Protokolleinschränkung: Die Variable kennt NUR die
        // aufheben()-Methode.
        beute.aufheben();

        // beute.bruellen(); // COMPILERFEHLER! Das Interface Sammelbar kennt
        // kein bruellen().
    }
}
```

Der gewaltige architektonische Vorteil: Wir können nun ein Inventar für unseren Helden programmieren, das völlig blind dafür ist, *was* es speichert. Es interessiert sich nicht dafür, ob das Objekt von Monster, von Item oder von Möbelstück erbt. Es verlangt nur die Unterschrift unter dem Vertrag Sammelbar:

```
public class Inventar {
    private Sammelbar[] tasche = new Sammelbar[10];

    public void einpacken(Sammelbar ding) {
        ding.aufheben(); // Dynamisches Binden sorgt für die richtige
                         // Ausführung!
```

```
        // Logik zum Speichern im Array...
    }
}
```

## 10.5 Erweitern von Interfaces

Manchmal wollen wir einen bestehenden Vertrag nehmen und ihn um neue Klauseln erweitern. Interfaces können von anderen Interfaces erben! Hierfür wird das Schlüsselwort extends verwendet (da ein Vertrag einen anderen Vertrag *erweitert*, er *implementiert* ihn nicht).

```
public interface MagischSammelbar extends Sammelbar {
    // Erbt automatisch void aufheben() aus Sammelbar!
    void magieEntfesseln();
}
```

Eine Klasse, die nun implements MagischSammelbar schreibt, muss *beide* Methoden ausprogrammieren.

## 10.6 Vergleich: Abstrakte Klassen vs. Interfaces

Wann nutzen Sie als Architekt was? Das ist eine der wichtigsten Design-Entscheidungen in Ihrem Projekt.

| **Eigenschaft** | **Abstrakte Klasse (abstract class)** | **Interface (interface)** |
|---|---|---|
| **Zweck** | Bauplan für verwandte Familien (Ist-ein). | Vertrag für Fähigkeiten (Kann-etwas / Hat-ein). |
| **Zustand (Attribute)** | Kann normale Attribute haben (Zustand speichern). | Kann **keine** Attribute haben (nur Konstanten). |
| **Konstruktoren** | Hat Konstruktoren (aufgerufen über super()). | Hat **keine** Konstruktoren. |

| | | |
|---|---|---|
| **Vererbungslimit** | Eine Klasse darf nur von **einer** (abstrakten) Klasse erben. | Eine Klasse darf **beliebig viele** Interfaces implementieren. |

*Faustregel:* Nutzen Sie abstrakte Klassen, um doppelten Code in Hierarchien zu vermeiden. Nutzen Sie Interfaces, um Klassen aus völlig unterschiedlichen Hierarchien eine gemeinsame Schnittstelle für die Polymorphie zu geben.

## 10.7 Default-Methoden und static-Methoden

Lange Zeit durften Interfaces in Java absolut keinen ausführbaren Code enthalten. Doch ab Java 8 gab es ein Problem: Was passiert, wenn Tausende Programmierer weltweit Ihr Interface Sammelbar in ihren Klassen implementiert haben, und Sie als Architekt beschließen, eine neue Methode void fallenlassen(); zum Interface hinzuzufügen?

Sofort würden zehntausende Programme weltweit nicht mehr kompilieren, weil all diese Klassen ihren Vertrag plötzlich nicht mehr erfüllen!

Um Verträge in Zukunft straffrei erweitern zu können, führte Java die **Default-Methoden** ein. Mit dem Schlüsselwort default können Sie einer Methode im Interface einen Standard-Codeblock geben.

```
public interface Sammelbar {
    void aufheben();

    // Eine Default-Methode hat echten Code!
    // Klassen MÜSSEN diese Methode NICHT überschreiben, dürfen es aber.
    default void fallenlassen() {
        System.out.println("Das Objekt fällt scheppernd zu Boden.");
    }
}
```

Zusätzlich können Interfaces auch **static-Methoden** enthalten. Diese funktionieren exakt wie statische Methoden in Klassen und dienen oft als kleine, nützliche Hilfsfunktionen, die logisch zum Interface gehören.

```
public interface Sammelbar {
    void aufheben();

    // Aufruf überall im Projekt per: Sammelbar.info()
    static void info() {
        System.out.println("Ein Interface für alles, was in die " +
```

```
                                "Tasche passt.");
    }
}
```

## 10.8 Funktionale Interfaces

Wenn Sie einen Blick in die moderne Java-Welt werfen, werden Sie oft auf einen speziellen Interface-Typ stoßen. Ein Interface, das **exakt eine einzige abstrakte Methode** besitzt, nennt man ein **Funktionales Interface** (SAM-Type: Single Abstract Method).

Man markiert sie oft (freiwillig) mit der Annotation @FunctionalInterface, damit der Compiler meckert, falls jemand versehentlich eine zweite abstrakte Methode hinzufügt.

```
@FunctionalInterface
public interface Konsumierbar {
    void trinken();
    // default- oder static-Methoden zählen nicht mit! Es darf nur eine
    // abstrakte geben.
}
```

Warum sind diese so wichtig? Sie bilden das absolute Fundament für *Lambda-Ausdrücke* und die funktionale Programmierung. Wenn wir in Buch 3 lernen, wie wir Code (und nicht nur Daten) durch unser Programm schicken, werden funktionale Interfaces unser tägliches Werkzeug sein. In Kapitel 18 dieses Buches werfen wir bereits einen ersten, detaillierten Blick darauf.

## 10.9 Versiegelte Hierarchien (sealed und permits)

Wir schließen dieses Kapitel mit einem der mächtigsten Architektur-Werkzeuge aus dem modernen Java (eingeführt in Java 17): den **Versiegelten Klassen und Interfaces**.

Bisher konnte jeder, der unsere Datei Sammelbar.java oder Monster.java sah, einfach implements oder extends schreiben und Teil unserer Hierarchie werden. Manchmal wollen wir als Architekten aber ein geschlossenes System bauen. Stellen Sie sich vor, wir schreiben eine Finanz-Software und haben ein Interface Zahlungsart. Wir wollen absolut garantieren, dass es im gesamten Universum dieses Programms nur Kreditkarte, PayPal und Rechnung gibt. Niemand soll heimlich eine Klasse Kryptowaehrung implements Zahlungsart bauen dürfen.

Wir können unser Interface (oder unsere abstrakte Klasse) mit dem Schlüsselwort **sealed** (versiegelt) markieren. Gleichzeitig müssen wir mit **permits** (erlaubt) eine exklusive Gästeliste der Klassen angeben, die diesen Vertrag unterschreiben dürfen!

```
// Nur die drei genannten Klassen dürfen dieses Interface jemals
implementieren!
public sealed interface Zahlungsart permits Kreditkarte, PayPal, Rechnung {
    void bezahlen(double betrag);
```

```
}

// Das ist erlaubt, weil Kreditkarte auf der permits-Liste steht.
// (Hinweis: Die implementierende Klasse muss zwingend 'final', 'sealed' oder
// 'non-sealed' sein)
public final class Kreditkarte implements Zahlungsart {
    @Override
    public void bezahlen(double betrag) {
        System.out.println("Zahlung per Karte.");
    }
}

// COMPILERFEHLER! Kryptowaehrung steht nicht auf der permits-Liste von Zahlungsart!
public final class Kryptowaehrung implements Zahlungsart { ... }
```

Mit sealed und permits haben Sie die absolute Kontrolle über Ihre Vererbungshierarchien zurückgewonnen. Der Compiler garantiert Ihnen, dass es niemals unvorhergesehene Vertragsnehmer geben wird.

---

Sie haben in diesem dritten großen Teil des Buches die Königsdisziplin der Objektorientierung gemeistert: Vererbung, Polymorphie, dynamisches Binden, abstrakte Klassen und Interfaces. Sie können nun hochkomplexe, flexible und strikt kontrollierte Architekturen entwerfen.

Doch egal wie perfekt Ihre Architektur ist: Zur Laufzeit passieren Dinge, die wir nicht kontrollieren können. Eine Netzwerkverbindung bricht ab, eine Datei fehlt auf der Festplatte, oder der Spieler gibt Text statt einer Zahl ein. Unser Programm würde unweigerlich abstürzen.

Im nächsten Teil (Teil IV: Robustheit und Typsicherheit) lernen wir, wie echte Architekten mit dem Scheitern umgehen. Wir widmen uns in Kapitel 11 den **Exceptions** (der Ausnahmebehandlung).

# Teil IV: Robustheit und Typsicherheit

# 11 Exceptions (Ausnahmebehandlung)

Sie haben als Architekt nun fantastische, hochkomplexe Klassenhierarchien entworfen. Unsere Monster und Helden interagieren über saubere Schnittstellen miteinander. In einer idealen Welt würde unser Programm nun für immer fehlerfrei laufen.

Doch die Realität der Softwareentwicklung ist unerbittlich. Die Festplatte des Spielers könnte voll sein, wenn wir den Spielstand speichern wollen. Die Internetverbindung bricht ab, während wir eine Highscore-Liste laden. Oder der Spieler gibt in einem Textfeld für das Alter versehentlich "Zwanzig" statt "20" ein.

Ein professionelles Programm darf in solchen Situationen nicht einfach lautlos abstürzen oder einfrieren. Es muss den Fehler erkennen, abfangen und elegant darauf reagieren.

## 11.1 Klassifizierung von Programmierfehlern

In der Programmierung unterscheiden wir grob zwischen drei Arten von Fehlern:

1. **Syntaxfehler:** Sie haben ein Semikolon vergessen oder eine Klammer falsch gesetzt. Diese Fehler fängt der Compiler ab. Das Programm startet gar nicht erst.
2. **Logische Fehler:** Das Programm läuft fehlerfrei durch, aber der Ork macht 50 statt 5 Schaden, weil Sie ein Plus statt einem Minus getippt haben. Hier helfen nur systematisches Testen und Debugging.
3. **Laufzeitfehler (Runtime Errors):** Das Programm ist syntaktisch korrekt und die Logik stimmt. Doch während das Programm läuft, tritt ein unvorhergesehenes Ereignis (eine "Ausnahme" / Exception) ein. Genau um diese Laufzeitfehler kümmern wir uns in diesem Kapitel.

## 11.2 Motivation: Trennung von Normalfall und Fehlerbehandlung

Erinnern Sie sich, wie wir in der imperativen Welt (Buch 1) mit Fehlern umgegangen sind? Wenn eine Funktion fehlschlagen konnte, haben wir oft spezielle Fehlercodes als Rückgabewert genutzt (z.B. -1 oder false).

Stellen Sie sich vor, unser Held möchte einen Trank aus dem Rucksack trinken:

```
// Der veraltete, imperative Weg (Fehlercodes)
int fehlerCode = meinHeld.trankTrinken(5);

if (fehlerCode == 0) {
    System.out.println("Trank erfolgreich getrunken.");
} else if (fehlerCode == -1) {
    System.out.println("Fehler: Rucksack ist leer!");
} else if (fehlerCode == -2) {
    System.out.println("Fehler: Trank ist abgelaufen!");
```

```
}
```

Dieser Ansatz hat massive architektonische Nachteile:

1. **Vermischung von Logik:** Der eigentliche "Happy Path" (das Trinken des Tranks) geht völlig im Chaos der unzähligen if-else-Fehlerabfragen unter.
2. **Ignoranz:** Nichts zwingt den Programmierer im Hauptprogramm, den Rückgabewert fehlerCode überhaupt zu überprüfen! Er kann den Fehler einfach ignorieren, und das Programm läuft munter in seinen eigenen Untergang.

Die objektorientierte Lösung für dieses Problem sind **Exceptions** (Ausnahmen). In Java sind Fehler keine simplen Zahlen mehr, sondern – Sie ahnen es – **echte Objekte**! Wenn etwas schiefgeht, erschafft die Methode ein Fehler-Objekt und wirft es dem Aufrufer buchstäblich vor die Füße.

## 11.3 Die Exception-Hierarchie (Throwable, Error, Exception)

Da Fehler in Java Objekte sind, gibt es für sie natürlich auch eine Vererbungshierarchie. Das Wissen aus Teil III dieses Buches zahlt sich nun direkt aus!

Ganz oben an der Spitze des Fehlerbaums steht die Klasse Throwable (das Werfbare). Von ihr erben zwei große Hauptäste:

1. **Error**: Das sind katastrophale Systemausfälle der Java Virtual Machine (JVM). Zum Beispiel ein OutOfMemoryError (der Arbeitsspeicher ist physikalisch voll) oder ein StackOverflowError. Als Architekt gilt die Regel: **Errors fangen wir niemals ab!** Wenn ein Error auftritt, ist das Programm ohnehin unrettbar verloren. Es darf und soll abstürzen.
2. **Exception**: Das sind Ausnahmesituationen innerhalb unseres eigenen Programms. Ein fehlendes Dokument, eine unterbrochene Netzwerkverbindung oder eine ungültige Benutzereingabe. Diese Exceptions *können* und *sollen* wir behandeln!

## 11.4 Checked- und Unchecked-Exceptions

Der Zweig der Exception-Klasse unterteilt sich in Java in zwei fundamental unterschiedliche philosophische Lager:

## 1. Unchecked Exceptions (Laufzeit-Bugs)

Alle Klassen, die von der speziellen Unterklasse RuntimeException erben, gelten als "unchecked" (ungeprüft). Das sind Fehler, die eigentlich durch sauberes Programmieren vermieden werden sollten.

Beispiele:

- NullPointerException (Sie greifen auf ein Objekt zu, das null ist).
- ArithmeticException (Sie teilen durch Null).

- IndexOutOfBoundsException (Sie greifen auf Index 10 eines Arrays zu, das nur 5 Elemente hat).
  Der Compiler mischt sich hier nicht ein. Er zwingt Sie nicht, diese Fehler zu behandeln, denn theoretisch könnte eine NullPointerException in jeder einzelnen Codezeile auftreten.

## 2. Checked Exceptions (Äußere Umstände)

Alle anderen Exceptions, die direkt von Exception erben (aber *nicht* von RuntimeException), sind "checked" (geprüft). Hier sagt der Compiler: *"Halt! Du versuchst hier eine Datei von der Festplatte zu lesen. Ich weiß aus Erfahrung, dass Dateien oft fehlen oder gesperrt sind. Ich weigere mich, diesen Code zu kompilieren, bis du mir bewiesen hast, dass du einen Notfallplan für diesen Fehler geschrieben hast!"*

## 11.5 Fangen und Behandeln (try, catch, finally)

Wie sieht so ein Notfallplan aus? Wir nutzen den try-catch-Block. Er fungiert wie ein Sicherheitsnetz für Trapezkünstler.

Im **try**-Block (versuche) schreiben wir unseren normalen "Happy Path"-Code, als gäbe es keine Fehler auf der Welt.

Im **catch**-Block (fange) schreiben wir den Code, der *nur* dann ausgeführt wird, wenn das Trapez reißt.

```
import java.io.File;
import java.util.Scanner;
import java.io.FileNotFoundException; // Eine Checked Exception!

public class SpielstandLader {

    public void ladeSpielstand() {
        // Wir spannen das Sicherheitsnetz
        try {
            // Der "Happy Path": Wir tun so, als würde alles klappen
            System.out.println("Versuche Datei zu öffnen...");
            File datei = new File("savegame.txt");
            Scanner leser = new Scanner(datei); // Hier KANN eine Exception
                                                // fliegen!

            System.out.println("Spielstand erfolgreich geladen!");
            leser.close();
        }
        catch (FileNotFoundException e) {
            // Dieser Block wird NUR betreten, wenn im try-Block die
            // FileNotFoundException geworfen wurde! Das Programm stürzt
            // NICHT ab.
```

```
            System.out.println("Fehler: 'savegame.txt' existiert nicht!");
            System.out.println("Starte ein neues Spiel mit Level 1...");

            // Wir können das Exception-Objekt 'e' auch befragen:
            // System.out.println(e.getMessage());
        }

        System.out.println("Das Programm läuft ganz normal weiter.");
    }
}
```

## Was passiert genau?

Wenn new Scanner(datei) scheitert, bricht die JVM den try-Block in exakt dieser Millisekunde ab. Die Zeile "Spielstand erfolgreich geladen!" wird niemals ausgeführt! Die JVM erschafft ein FileNotFoundException-Objekt und wirft es in den catch-Block.

## Der finally-Block für Aufräumarbeiten

Oft öffnen wir Ressourcen (wie eine Datei oder eine Netzwerkverbindung), die wir unbedingt wieder schließen müssen – egal ob ein Fehler aufgetreten ist oder nicht. Dafür gibt es den optionalen finally-Block. Er wird **immer** ausgeführt, absolut unvermeidbar.

```
try {
    // Gefährlicher Code
} catch (Exception e) {
    // Fehlerbehandlung
} finally {
    System.out.println("Ich werde IMMER ausgeführt, egal ob Try oder Catch "
                       + "erfolgreich waren!");
    // Hier schließt man typischerweise Dateien oder Datenbankverbindungen
    // ab.
}
```

## 11.6 Werfen (throw) und Weiterleiten (throws)

Bisher haben wir Exceptions nur gefangen, die von Javas Standardklassen (wie Scanner) geworfen wurden. Doch als Architekt eigener Klassen müssen wir Fehler auch selbst produzieren und werfen können!

Stellen Sie sich vor, wir haben eine Methode, um unserem Helden Schaden zuzufügen. Wenn jemand versucht, negativen Schaden zu übergeben, ist das ein logischer Fehler, den wir sofort ahnden müssen.

Um ein Exception-Objekt aktiv auszulösen, nutzen wir das Schlüsselwort **throw** (wirf).

```
public class Held {
    private int lebenspunkte = 100;

    public void schadenNehmen(int schaden) {
        if (schaden < 0) {
            // Wir erschaffen ein Fehler-Objekt und werfen es der JVM ins
            // Gesicht!
            // IllegalArgumentException ist eine eingebaute Unchecked
            // Exception.
            throw new IllegalArgumentException("Schaden darf nicht negativ "
                                                + "sein!");
        }

        this.lebenspunkte -= schaden;
    }
}
```

Sobald das Schlüsselwort throw ausgeführt wird, stirbt die Methode auf der Stelle. Es wird kein weiterer Code in dieser Methode ausgeführt.

## Das Schlüsselwort throws (Der Vertrag)

Was passiert, wenn wir in einer unserer Methoden eine *Checked Exception* erzeugen (oder eine aufrufen, die wir nicht selbst fangen wollen)? Wir müssen dem Compiler vertraglich zusichern, dass diese Methode gefährlich ist und den Fehler an den Aufrufer weiterleitet (die heiße Kartoffel weiterreicht).

Dafür schreiben wir **throws** (wirft) in die Signatur der Methode, direkt vor die öffnende geschweifte Klammer.

```
import java.io.IOException;

public class NetzwerkManager {

    // Mit "throws" warnen wir jeden, der diese Methode nutzen will:
    // "Achtung! Wenn du mich aufrufst, musst DU dich um die IOException
    // kümmern!"
    public void verbindeZumServer() throws IOException {
        boolean serverOnline = false; // Simulation

        if (!serverOnline) {
            throw new IOException("Server antwortet nicht!");
        }
        System.out.println("Verbunden.");
    }
}
```

Wenn nun unser Hauptprogramm verbindeZumServer() aufruft, zwingt der Compiler das Hauptprogramm, einen try-catch-Block um den Aufruf zu bauen! Die Verantwortung wurde erfolgreich delegiert.

## 11.7 Definition eigener Exception-Klassen

Die eingebauten Exceptions von Java (wie IllegalArgumentException) sind nützlich, aber sie sprechen nicht die Sprache unseres Spiels. Ein echter Objekt-Architekt schreibt seine eigenen Fehlerklassen!

Dank der Vererbung ist das unfassbar einfach. Wir erschaffen eine Klasse, die einfach von Exception (für Checked) oder RuntimeException (für Unchecked) erbt.

Lassen Sie uns eine eigene Exception für den Fall schreiben, dass der Held versucht, mit einer zerbrochenen Waffe anzugreifen:

```
// 1. Wir definieren unsere eigene Checked Exception
public class WaffeZerbrochenException extends Exception {

    // Wir bieten einen Konstruktor an, um eine Nachricht zu übergeben
    public WaffeZerbrochenException(String nachricht) {
        // Wir reichen die Nachricht an die Ur-Klasse Exception weiter
        super(nachricht);
    }
}
```

Jetzt integrieren wir diesen brandneuen Fehler in unsere Spielwelt:

```
// 2. Nutzung in der Waffe-Klasse
public class Waffe {
    private int haltbarkeit = 0; // Waffe ist kaputt!

    public void zuschlagen() throws WaffeZerbrochenException {
        if (haltbarkeit <= 0) {
            // Wir werfen unser selbstgebautes Objekt!
            throw new WaffeZerbrochenException("Das Schwert ist in zwei " +
                                               "Teile zersprungen!");
        }
        haltbarkeit--;
        System.out.println("Zack! Ein harter Treffer.");
    }
}

// 3. Fehlerbehandlung im Hauptprogramm
public class Spiel {
    public static void main(String[] args) {
```

```
        Waffe eisenschwert = new Waffe();

        try {
            System.out.println("Der Held holt aus...");
            eisenschwert.zuschlagen(); // Compiler ERZWINGT den
                                       // try-catch-Block!
            System.out.println("Der Angriff war erfolgreich.");
        }
        // Wir fangen EXAKT unsere eigene Exception!
        catch (WaffeZerbrochenException e) {
            System.out.println("KAMPF-LOG: " + e.getMessage());
            System.out.println("Der Held wechselt schnell zu den Fäusten!");
        }
    }
}
```

## Der Geniestreich der Polymorphie bei Exceptions:

Sie können auch Exceptions polymorph fangen! Wenn Sie catch (Exception e) schreiben, fangen Sie automatisch *alle* Unterklassen von Exception (also auch Ihre WaffeZerbrochenException oder Javas IOException). Ein sauberes Architektur-Design fängt jedoch immer so spezifisch wie möglich, um nicht versehentlich völlig andere Fehler unbeabsichtigt zu verschlucken.

## 11.8 Der Multi-Catch-Block (Mehrere Fehler gleichzeitig fangen)

Als Architekt haben wir in Kapitel 6 das oberste Gebot der Softwareentwicklung kennengelernt: **DRY (Don't Repeat Yourself)**. Code-Duplizierung ist böse.

Doch beim Fangen von Exceptions stießen Programmierer früher oft auf folgendes Problem: Was passiert, wenn in unserem try-Block zwei völlig unterschiedliche Checked Exceptions fliegen können (z.B. eine FileNotFoundException und eine SQLException der Datenbank), wir aber bei *beiden* Fehlern exakt denselben Rettungsplan ausführen wollen?

Vor Java 7 sah das so aus:

```
try {
    // Code, der Dateien liest und in eine Datenbank schreibt
    ladeDatenbank();
}
catch (FileNotFoundException e) {
    System.out.println("LOG: Ein kritischer Fehler ist aufgetreten: " +
                       e.getMessage());
    vorgangAbbrechen();
}
catch (SQLException e) {
    // Furchtbar! Exakt der gleiche Code wird hier dupliziert.
```

```
    System.out.println("LOG: Ein kritischer Fehler ist aufgetreten: " +
                       e.getMessage());
    vorgangAbbrechen();
}
```

Ein fauler Programmierer würde nun einfach catch (Exception e) schreiben, um alles auf einmal zu fangen. Aber das ist eine Todsünde in der Architektur! Damit würden wir auch NullPointerExceptions oder andere Bugs blind schlucken, die wir eigentlich reparieren müssten.

Die elegante, moderne Lösung ist der **Multi-Catch-Block**. Mit dem einfachen senkrechten Strich | (dem logischen ODER / Pipe-Symbol) können wir völlig verschiedene Exceptions in einem einzigen catch-Block auflisten und in derselben Variablen e auffangen:

```
try {
    ladeDatenbank();
}
// Wir fangen hochspezifisch, aber in einem einzigen Block!
catch (FileNotFoundException | SQLException e) {
    System.out.println("LOG: Ein Ein-/Ausgabe- oder Datenbankfehler: " +
                       e.getMessage());
    vorgangAbbrechen();
}
```

## Zwei wichtige Regeln für den Multi-Catch:

1. Die Variable e ist in diesem Block automatisch **final**. Sie dürfen ihr kein neues Exception-Objekt zuweisen.
2. Sie dürfen im Multi-Catch **niemals** Exceptions kombinieren, die in der Vererbungshierarchie voneinander abstammen (z.B. catch (FileNotFoundException | IOException e) ist verboten, da die FileNotFoundException ohnehin eine Unterklasse der IOException ist und der Compiler das als sinnlosen Code erkennt).

Mit dem Multi-Catch halten Sie Ihre Fehlerbehandlung spezifisch, sicher und absolut frei von redundantem Code.

## 11.9 Automatisches Aufräumen (Try-with-Resources)

In Abschnitt 11.5 haben wir den finally-Block kennengelernt. Als Architekt wissen Sie: Wenn Sie eine externe Ressource öffnen (wie eine Datei für den Spielstand oder eine Netzwerkverbindung zum Multiplayer-Server), müssen Sie diese zwingend wieder schließen. Vergessen Sie das, blockieren Sie den Arbeitsspeicher oder die Datei für andere Programme (Resource Leak).

Doch der klassische Weg mit finally führt oft zu furchtbar hässlichem Code. Warum? Weil die .close()-Methode selbst oft wieder eine Exception wirft! Sie müssten also im finally-Block schon

wieder einen neuen try-catch-Block bauen. Das ist Boilerplate-Code aus der Hölle.

Die moderne und elegante Lösung (eingeführt in Java 7) nennt sich **Try-with-Resources**.

Wir können die Ressource direkt bei der Erschaffung in **runde Klammern** hinter das try schreiben. Wenn wir das tun, gibt uns der Compiler eine magische Garantie: Sobald der try-Block verlassen wird – egal ob erfolgreich oder durch einen Fehler! – schließt Java diese Ressource völlig automatisch im Hintergrund für uns ab. Der finally-Block entfällt komplett.

```
import java.io.File;
import java.util.Scanner;
import java.io.FileNotFoundException;

public class SpielstandManager {

    public void ladeSpielstand() {
        // Die Ressource wird in den runden Klammern deklariert und
        // instanziiert!
        try (Scanner leser = new Scanner(new File("savegame.txt"))) {

            String daten = leser.nextLine();
            System.out.println("Spielstand geladen: " + daten);

            // Sobald diese geschweifte Klammer erreicht wird, ruft Java
            // unsichtbar leser.close() auf.
        }
        catch (FileNotFoundException e) {
            System.out.println("LOG: Datei nicht gefunden.");
        }
        // Auch wenn die Exception geflogen ist, ist der Scanner hier
        // garantiert geschlossen!
    }
}
```

## Das Architektur-Geheimnis: Der Vertrag AutoCloseable

Wie um alles in der Welt weiß der Java-Compiler, dass das Objekt in den runden Klammern überhaupt eine .close()-Methode besitzt?

Hier schließt sich der Kreis zu Kapitel 10. Das Try-with-Resources-Konstrukt ist streng typisiert. Der Compiler erlaubt in den runden Klammern des try *ausschließlich* Objekte, deren Klasse ein spezielles Interface aus der Java-Bibliothek implementiert hat: den Vertrag **AutoCloseable** (automatisch schließbar).

Dieser Vertrag diktiert genau eine einzige Methode: void close() throws Exception;. Da Klassen wie Scanner oder Dateileser diesen Vertrag unterschrieben haben, kann die Java Virtual

Machine sich blind darauf verlassen, dass die Methode existiert, und ruft sie bei Bedarf für Sie auf.

Ein brillantes Beispiel dafür, wie Interfaces (Verträge) die Basis für sichere und komfortable Sprach-Features bilden.

---

Mit Exceptions haben Sie gelernt, wie man Normalfall und Fehlerbehandlung elegant voneinander trennt. Ihr Programm ist nun robust und wehrt sich aktiv gegen ungültige Eingaben oder unvorhergesehene Laufzeitereignisse, ohne abzustürzen.

Doch es gibt noch einen Bereich, in dem unsere Architektur aktuell erschreckend unsicher ist. Erinnern Sie sich an unsere polymorphen Arrays aus Kapitel 7? Wenn wir ein Array vom Typ Monster[] haben, können wir zwar Orks und Drachen hineinlegen, aber was ist, wenn wir ein Inventar bauen wollen, das flexibel *alles* speichern kann? Bisher mussten wir auf die Ur-Klasse Object zurückgreifen und wussten beim Herausholen nie, was für ein Typ das Objekt eigentlich ist.

Dieses Typ-Chaos beenden wir im nächsten Kapitel: Wir lernen **Generics** (Typparameter) kennen, die Schablonen der Objektorientierung.

# 12 Generics (Generische Programmierung)

Im vorherigen Kapitel haben wir gelernt, wie wir unser Programm mit Exceptions vor Laufzeitfehlern schützen. Doch der beste Architekt ist nicht derjenige, der Abstürze elegant abfängt, sondern derjenige, der dafür sorgt, dass Fehler gar nicht erst entstehen können!

Ein Großteil der Laufzeitfehler in älteren Java-Programmen stammte aus einem architektonischen Dilemma: Wie bauen wir Datenstrukturen, die flexibel alles speichern können, ohne dabei die Kontrolle darüber zu verlieren, *was* genau darin liegt? Die Antwort darauf sind **Generics** (Typparameter).

## 12.1 Motivation: Typisierte Sammlungen und Typsicherheit

Stellen Sie sich vor, wir wollen für unser Spiel eine allgemeine Schatzkiste programmieren. In diese Kiste soll man genau ein Objekt hineinlegen und wieder herausnehmen können. Da wir noch nicht wissen, ob der Spieler später ein Schwert, einen Heiltrank oder gar einen Ork in die Kiste sperrt, nutzen wir unser Wissen aus Kapitel 6 und verwenden die Ur-Klasse Object als Datentyp:

```
// Der veraltete Vor-Generics-Weg (Bitte nicht nachmachen!)
public class Schatzkiste {
    private Object inhalt;

    public void hineinlegen(Object ding) {
        this.inhalt = ding;
    }

    public Object herausnehmen() {
        return this.inhalt;
    }
}
```

Das funktioniert scheinbar wunderbar. Da alles in Java von Object erbt (Upcast), können wir alles in die Kiste packen. Doch beim Herausholen schnappt die Falle zu:

```
public class Spiel {
    public static void main(String[] args) {
        Schatzkiste kiste = new Schatzkiste();
        kiste.hineinlegen(new Heiltrank(50)); // Upcast zu Object
                                              // (unsichtbar)

        // ... später im Spiel ...

        // Wir müssen casten, denn die Kiste liefert nur "Object" zurück!
        // Der Compiler weiß nicht mehr, dass es ein Heiltrank war.
        Heiltrank t = (Heiltrank) kiste.herausnehmen();
```

```
        t.trinken();
    }
}
```

Dieses ständige, manuelle Casten (der Downcast) ist extrem gefährlich. Was passiert, wenn ein anderer Programmierer aus Versehen ein Schwert in *diese* Kiste gelegt hat? Der Compiler erlaubt es (ein Schwert ist ein Object). Aber wenn wir versuchen, es als Heiltrank herauszuholen, stürzt unser Programm mit einer ClassCastException ab. Die Typsicherheit ist komplett zerstört.

Wir brauchen eine Kiste, bei der wir im Moment der Erschaffung sagen können: *"Du bist ab heute eine Schatzkiste EXKLUSIV für Heiltränke!"*

## 12.2 Definition und Nutzung von generischen Klassen (<T>)

Um dieses Problem zu lösen, können wir aus unserer Klasse eine **Schablone** (ein Template) machen. Wir verpassen der Klasse einen Platzhalter für einen Datentyp.

Diesen Platzhalter definieren wir in spitzen Klammern < > direkt hinter dem Klassennamen. Konventionsgemäß verwendet man dafür in Java einzelne Großbuchstaben, meistens T (für Type) oder E (für Element).

```
// Wir definieren die Klasse als generisch. 'T' ist unser Platzhalter.
public class Schatzkiste<T> {

    // Anstatt 'Object' nutzen wir nun überall unseren Platzhalter 'T'
    private T inhalt;

    public void hineinlegen(T ding) {
        this.inhalt = ding;
    }

    public T herausnehmen() {
        return this.inhalt;
    }
}
```

Wie nutzen wir diese Schablone nun? Wenn wir im Hauptprogramm eine Kiste erschaffen, müssen wir das T durch einen konkreten Datentyp ersetzen. Das tun wir, indem wir den gewünschten Typ in spitze Klammern schreiben.

```
public class Spiel {
    public static void main(String[] args) {

        // Wir gießen die Schablone mit dem Typ 'Heiltrank' aus!
        // Der Compiler ersetzt nun im Hintergrund jedes 'T' der Klasse durch
        // 'Heiltrank'.
        Schatzkiste<Heiltrank> trankKiste = new Schatzkiste<Heiltrank>();
```

```
        // Seit Java 7 reicht auf der rechten Seite der leere
        // "Diamond-Operator" <>.
        // Der Compiler leitet den Typ von der linken Seite ab:
        Schatzkiste<Schwert> waffenKiste = new Schatzkiste<>();

        // --- Die Genialität der Generics in Aktion ---

        trankKiste.hineinlegen(new Heiltrank(50));

        // COMPILERFEHLER! Absolute Typsicherheit!
        // Der Compiler weiß, dass in diese Kiste NUR Heiltränke dürfen.
        // trankKiste.hineinlegen(new Schwert(10));

        // Beim Herausholen ist KEIN Cast mehr nötig! Die Methode liefert
        // direkt 'Heiltrank'.
        Heiltrank t = trankKiste.herausnehmen();
        t.trinken();
    }
}
```

Wir haben das Problem der ClassCastException gelöst, bevor das Programm überhaupt startet. Wenn wir einen falschen Datentyp verwenden, weigert sich der Compiler, den Code zu übersetzen. Generics verlagern Fehler von der unsicheren Laufzeit in die sichere Kompilierzeit!

## 12.3 Typ-Parameter mit Einschränkungen (extends)

Mit <T> erlauben wir, dass unsere Kiste absolut *jeden* Datentyp der Welt annehmen kann, sogar abstrakte Dinge wie String oder Scanner.

Manchmal wollen wir als Architekt aber eine generische Klasse bauen, die zwar flexibel ist, aber nur für eine bestimmte **Familie von Klassen** gilt. Stellen Sie sich vor, wir bauen einen magischen Kaefig. In einen Käfig sollen nur Wesen der Familie Monster gesperrt werden dürfen.

Wir können den Platzhalter T einschränken (Type Bounding). Dafür nutzen wir das Schlüsselwort extends.

*Architektur-Hinweis:* In den spitzen Klammern bedeutet extends sowohl "erbt von der Klasse" als auch "implementiert das Interface".

```
// T darf nicht mehr JEDER Typ sein. T MUSS ein Monster sein (oder davon
erben).
public class Kaefig<T extends Monster> {
    private T gefangener;

    public void einsperren(T wesen) {
        this.gefangener = wesen;
```

```
        // Da wir wissen, dass T auf jeden Fall ein Monster ist,
        // DÜRFEN wir hier Methoden der Monster-Klasse aufrufen!
        this.gefangener.bruellen();
    }
}

// Korrekte Nutzung: Drache erbt von Monster
Kaefig<Drache> drachenKaefig = new Kaefig<>();

// COMPILERFEHLER! Schwert erbt nicht von Monster. Die Schablone passt nicht!
// Kaefig<Schwert> waffenKaefig = new Kaefig<>();
```

## 12.4 Wildcards

Wir kommen nun zu einer der größten architektonischen Fallen in Java, an der selbst erfahrene Programmierer oft scheitern: **Das Problem der Varianz.**

Schauen Sie sich folgenden logischen Aufbau an:

Ein Drache ist ein Monster (Vererbung).

Darf ich also eine Methode, die einen Kaefig<Monster> verlangt, auch mit einem Kaefig<Drache> füttern?

```
public class Tierpfleger {
    // Diese Methode akzeptiert einen Käfig für Monster
    public void kaefigPutzen(Kaefig<Monster> k) {
        System.out.println("Der Käfig wird gereinigt.");
    }
}

public class Spiel {
    public static void main(String[] args) {
        Tierpfleger pfleger = new Tierpfleger();
        Kaefig<Drache> meinDrachenKaefig = new Kaefig<>();

        // COMPILERFEHLER! Incompatible Types!
        pfleger.kaefigPutzen(meinDrachenKaefig);
    }
}
```

Warum zum Teufel gibt es hier einen Fehler? Ein Drache ist doch ein Monster!

Hier greift ein eiserner Schutzmechanismus von Java: **Generics sind nicht polymorph!** Ein Kaefig<Drache> ist **keine** Unterklasse von Kaefig<Monster>.

Warum muss das so sein? Stellen Sie sich vor, der Compiler würde den Aufruf erlauben. Der Tierpfleger hat nun die Variable k vom Typ Kaefig<Monster>. Er denkt, er hat einen allgemeinen Monsterkäfig vor sich. Nichts würde den Tierpfleger daran hindern, in der Methode k.einsperren(new Ork()) aufzurufen! Und plötzlich hätten wir einen Ork in unseren exklusiven Kaefig<Drache> gestopft. Die Typsicherheit wäre zerstört.

## Die Lösung: Wildcards (?)

Um eine Methode zu schreiben, die Käfige von *beliebigen* Unterklassen akzeptiert, nutzen wir die **Wildcard** (den Joker): ein Fragezeichen ?.

Wir sagen dem Compiler: *"Diese Methode akzeptiert einen Käfig von IRGENDEINEM unbekannten Typen, solange dieser Typ ein Monster oder eine Unterklasse davon ist."*

```
public class Tierpfleger {
    // Der Parameter nutzt das Wildcard ? extends Monster (Upper Bound)
    public void kaefigPutzen(Kaefig<? extends Monster> k) {
        System.out.println("Der Käfig wird gereinigt.");

        // WICHTIG: Da wir nicht wissen, OB es ein Drachen- oder Orkkäfig
        // ist, verbietet uns der Compiler hier strengstens, neue Objekte
        // in den Käfig hineinzulegen! Wir dürfen nur Dinge
        // herausholen/lesen.
    }
}
```

Jetzt funktioniert der Aufruf pfleger.kaefigPutzen(meinDrachenKaefig); fehlerfrei! Mit Wildcards können wir hochflexible Methoden schreiben, die generische Container verarbeiten, ohne die Typsicherheit zu gefährden.

Hier ist ein Entwurf für den ergänzenden Abschnitt. Wenn Sie „sub“ schreiben, gehe ich davon aus, dass Sie entweder das **Subtyping** (die Untertyp-Beziehung bei Generics) oder das direkte Gegenstück zu extends, nämlich die Lower Bounds mit **super**, meinen.

Da beides untrennbar zusammengehört, habe ich einen kompakten Abschnitt verfasst, der sich perfekt an die Wildcards (? extends) aus Kapitel 12.4 anschließt. Er erklärt, wie wir in generische Kisten nicht nur sicher hineinsehen, sondern auch sicher hineinschreiben können.

## 12.5 Subtyping, Lower Bounds (? super T) und PECS

In vorherigen Abschnitt haben wir die eiserne Regel des generischen Subtypings (Untertyp-Beziehung) kennengelernt: Ein Kaefig<Drache> ist **kein** Untertyp von Kaefig<Monster>. Die Vererbungshierarchie der Inhalte überträgt sich nicht automatisch auf die Kisten.

Um eine Methode zu schreiben, die Käfige von Drachen *und* Orks lesen kann, haben wir die

Wildcard mit einer Obergrenze (Upper Bound) genutzt: Kaefig<? extends Monster>.

Doch erinnern Sie sich an den Preis, den wir dafür zahlen mussten? Der Compiler hat uns strengstens verboten, neue Monster in diesen Käfig **hineinzulegen**! Er wusste ja nicht, ob wir gerade einen Ork in einen reinen Drachenkäfig stopfen.

Was machen wir aber als Architekten, wenn wir eine Methode schreiben wollen, deren einziger Zweck es ist, einen frisch geschlüpften Drachen in einen Käfig zu stecken?

Wir brauchen einen Käfig, der garantiert groß und allgemein genug ist, um einen Drachen aufzunehmen. Das kann ein Kaefig<Drache> sein, aber auch ein Kaefig<Monster> oder sogar ein völlig allgemeiner Kaefig<Object>.

Hierfür bietet Java das exakte Gegenteil zu extends: die Untergrenze (Lower Bound) mit dem Schlüsselwort **super** (die Oberklasse).

```
public class DrachenZuechter {

    // Wir fordern: "Einen Käfig vom Typ Drache ODER irgendeiner
    // seiner Oberklassen!"
    public void dracheUnterbringen(Kaefig<? super Drache> k) {

        // ERLAUBT! Da k mindestens Drachen (oder Allgemeineres) fasst,
        // ist es zu 100% sicher, hier einen Drachen hineinzulegen.
        k.einsperren(new Drache("Glurak", 500));

        // COMPILERFEHLER! Wir dürfen hier keinen Ork hineinlegen.
        // k könnte ja ein Kaefig<Drache> sein!
        // k.einsperren(new Ork());

        // ACHTUNG BEIM LESEN: Wenn wir etwas herausholen, garantiert uns der
        // Compiler nur, dass es ein 'Object' ist (denn k könnte
        // Kaefig<Object> sein).
        Object unbekannt = k.herausnehmen();
    }
}

// Nutzung im Hauptprogramm:
DrachenZuechter zuechter = new DrachenZuechter();

Kaefig<Drache> drachenKaefig = new Kaefig<>();
Kaefig<Monster> allgemeinerKaefig = new Kaefig<>();

zuechter.dracheUnterbringen(drachenKaefig);        // OK!
zuechter.dracheUnterbringen(allgemeinerKaefig);    // OK! (Monster ist super
                                                   // von Drache)
```

## Das Architektur-Prinzip: PECS

Diese Unterscheidung zwischen Lesen (extends) und Schreiben (super) verwirrt selbst erfahrene Java-Entwickler oft jahrelang. Um nie wieder durcheinanderzukommen, hat der berühmte Software-Architekt Joshua Bloch (einer der Entwickler von Java) ein geniales Akronym erfunden, das Sie sich einprägen sollten: **PECS**.

Das steht für: **P**roducer **E**xtends, **C**onsumer **S**uper.

- **Producer (Produzent/Lieferant):** Wenn Ihre Methode Daten aus einer generischen Kiste *herauslesen* will (die Kiste "produziert" also Daten für Ihre Methode), nutzen Sie **? extends T**. Sie dürfen dann nichts hineinlegen.
- **Consumer (Konsument/Verbraucher):** Wenn Ihre Methode Daten in eine generische Kiste *hineinschreiben* will (die Kiste "konsumiert" Ihre Daten), nutzen Sie **? super T**. Sie können dann beim Herausholen keine spezifischen Typen mehr garantieren.

Wenn Sie dieses Prinzip verinnerlicht haben, werden Sie nie wieder an unlesbaren Compilerfehlern bei Generics verzweifeln!

## 12.6 Generische Methoden

Generics sind nicht nur auf ganze Klassen beschränkt. Manchmal ist die Klasse selbst ganz normal, aber eine *einzelne Methode* soll flexibel für verschiedene Typen sein.

Um eine generische Methode zu definieren, schreiben wir die spitzen Klammern <T> direkt *vor* den Rückgabewert der Methode.

Ein klassisches Beispiel ist eine Hilfsmethode, die den Inhalt zweier generischer Kisten miteinander vertauscht:

```
public class MagieHelfer {

    // Eine generische Methode! Das <T> wird hier vor dem 'void' deklariert.
    public <T> void inhalteTauschen(Schatzkiste<T> kiste1,
                                    Schatzkiste<T> kiste2) {
        T zwischenspeicher = kiste1.herausnehmen();
        kiste1.hineinlegen(kiste2.herausnehmen());
        kiste2.hineinlegen(zwischenspeicher);
    }
}
```

Wenn wir diese Methode aufrufen, müssen wir das T oft gar nicht explizit angeben. Der Compiler ist clever genug, den Datentyp aus den Parametern abzuleiten, die wir übergeben (Type Inference).

```
MagieHelfer helfer = new MagieHelfer();
```

```
Schatzkiste<Heiltrank> k1 = new Schatzkiste<>();
Schatzkiste<Heiltrank> k2 = new Schatzkiste<>();

// Der Compiler sieht, dass beide Kisten <Heiltrank> sind.
// Er setzt das <T> der Methode unsichtbar auf <Heiltrank>.
helfer.inhalteTauschen(k1, k2);

// Würden wir eine <Heiltrank>-Kiste und eine <Schwert>-Kiste übergeben,
// würde der Compiler die Ausführung verbieten, da das T für beide gelten
// muss!
```

---

Mit Generics haben Sie das Werkzeug kennengelernt, um wiederverwendbare, extrem typsichere Datenstrukturen und Algorithmen zu entwerfen. Sie sind das Rückgrat der modernen Java-Standardbibliothek. Wenn wir in Kapitel 17 die Collection-API (Listen, Sets und Maps) kennenlernen, werden wir exakt dieses Wissen benötigen, um zu verstehen, warum man List<String> oder ArrayList<Monster> schreibt.

Doch bevor wir uns den gigantischen Bibliotheken Javas widmen, müssen wir unseren architektonischen Werkzeugkasten für das Klassendesign abschließen. Bisher haben wir Klassen immer als eigenständige Dateien betrachtet. Aber wussten Sie, dass man Klassen auch *in* andere Klassen hineinschreiben kann, wie russische Matroschka-Puppen?

Diese faszinierende Struktur besprechen wir im nächsten Kapitel: **Innere Klassen**.

# Teil V: Besondere Bauformen von Objekten

# 13 Innere Klassen

Bisher galt in unserer Architektur eine scheinbar unumstößliche Regel: Jede Klasse bekommt ihre eigene Java-Datei. Der Ork wohnt in Ork.java, das Monster in Monster.java. Für wiederverwendbare, eigenständige Bausteine ist das genau der richtige Weg.

Doch was passiert, wenn wir Bausteine entwerfen, die völlig abhängig von einem anderen Baustein sind? Was, wenn eine Klasse außerhalb einer bestimmten anderen Klasse überhaupt keinen Sinn ergibt? Hier greifen wir in die architektonische Trickkiste und lernen, wie wir Klassen *ineinander* verschachteln können – wie russische Matroschka-Puppen.

## 13.1 Motivation und Definition

Erinnern Sie sich an das Geheimnisprinzip (die Kapselung) aus Kapitel 5? Wir haben Attribute private gemacht, um sie vor der Außenwelt zu verstecken. Manchmal reicht das aber nicht. Manchmal wollen wir eine *komplette Klasse* vor der Außenwelt verstecken, weil sie nur ein interner Hilfsmechanismus ist.

Stellen Sie sich vor, wir programmieren ein Inventar für unseren Helden. Das Inventar speichert nicht nur Items, sondern organisiert sie in kleinen InventarSlot-Objekten (die festhalten, welches Item wie oft vorhanden ist).

Ein InventarSlot ergibt ohne ein Inventar absolut keinen Sinn. Niemand in unserem Spiel sollte jemals einfach so einen InventarSlot im Hauptprogramm erschaffen.

Die Lösung: Wir definieren die Klasse InventarSlot einfach *innerhalb* der geschweiften Klammern der Klasse Inventar. Solche Klassen nennen wir **Innere Klassen** (Nested Classes).

Es gibt in Java vier verschiedene Arten von inneren Klassen, die wir uns nun Schritt für Schritt ansehen werden.

## 13.2 Elementklassen (Nicht-statische innere Klassen)

Die häufigste Form der inneren Klasse ist die **Elementklasse** (oft einfach "Innere Klasse" genannt). Sie wird auf der gleichen Ebene wie normale Attribute und Methoden deklariert, aber ohne das Schlüsselwort static.

Der architektonische Clou einer Elementklasse: **Jedes Objekt der inneren Klasse ist untrennbar mit einem konkreten Objekt der äußeren Klasse verbunden!** Weil diese Verbindung existiert, darf die innere Klasse völlig frei auf alle private Attribute der äußeren Klasse zugreifen.

Lassen Sie uns das Inventar und seinen InventarSlot bauen:

```
public class Inventar {
    // Ein privates Attribut der äußeren Klasse
    private int maxGewicht = 100;

    // ------------------------------------------------------------
    // Beginn der inneren Klasse (Elementklasse)
    // Wir machen sie 'private', damit niemand von außen sie sieht!
    private class InventarSlot {
        private String itemZuweisung;
        private int anzahl;

        public InventarSlot(String item, int anzahl) {
            this.itemZuweisung = item;
            this.anzahl = anzahl;
        }

        public void slotPruefen() {
            // MAGIE: Die innere Klasse greift direkt auf 'maxGewicht' zu!
            System.out.println("Prüfe Slot für " + itemZuweisung);
            System.out.println("Das maximale Gewicht des Rucksacks ist " +
                               maxGewicht);
        }
    }
    // Ende der inneren Klasse
    // ------------------------------------------------------------

    // Eine Methode der äußeren Klasse, die die innere Klasse nutzt
    public void itemHinzufuegen(String item) {
        // Das Inventar erschafft einen eigenen, internen Slot
        InventarSlot neuerSlot = new InventarSlot(item, 1);
        neuerSlot.slotPruefen();
    }
}
```

Wenn eine Elementklasse ausnahmsweise nicht private, sondern public ist, können Sie sie sogar vom Hauptprogramm aus instanziieren. Da sie aber zwingend ein äußeres Objekt zum Überleben braucht, sieht die Syntax beim new-Aufruf sehr exotisch aus:

```
// WENN InventarSlot public wäre:
Inventar meinRucksack = new Inventar();

// Wir rufen 'new' AUF dem äußeren Objekt auf!
Inventar.InventarSlot slot = meinRucksack.new InventarSlot("Heiltrank", 5);
```

*Als Architekt sollten Sie sich diese Syntax zwar merken (für Prüfungen oder Fremdcode), in der Praxis hält man Elementklassen jedoch fast immer private.*

## 13.3 Statische innere Klassen

Was passiert, wenn wir eine Klasse in eine andere verschachteln wollen, weil sie logisch zusammengehören (Namensraum-Gruppierung), aber die innere Klasse *keine* direkte Verbindung zu einem konkreten äußeren Objekt braucht?

Dafür nutzen wir die **statische innere Klasse**. Sie verhält sich im Grunde wie eine völlig normale, eigenständige Java-Klasse, die rein optisch in einer anderen Klasse "geparkt" wurde.

```
public class DungeonSpiel {

    // Eine normale Instanzvariable des Spiels
    private String spielerName;

    // Eine statische innere Klasse
    public static class HighscoreEintrag {
        public String name;
        public int punkte;

        public HighscoreEintrag(String name, int punkte) {
            this.name = name;
            this.punkte = punkte;
        }

        public void drucken() {
            // FEHLER! Eine statische innere Klasse hat KEINEN Zugriff auf
            // die Instanzvariablen der äußeren Klasse (spielerName ist
            // unsichtbar).
            // System.out.println(spielerName);

            System.out.println(this.name + " - " + this.punkte);
        }
    }
}
```

Die Instanziierung im Hauptprogramm ist hier viel natürlicher als bei der Elementklasse, da wir kein äußeres Objekt als "Wirt" benötigen:

```
public class Spiel {
    public static void main(String[] args) {
        // Wir brauchen kein Objekt von DungeonSpiel!
        // Wir nutzen den Klassennamen nur als Pfad (wie bei einem Paket).
        DungeonSpiel.HighscoreEintrag eintrag =
            new DungeonSpiel.HighscoreEintrag("Artus", 999);
        eintrag.drucken();
    }
}
```

## 13.4 Lokale Klassen (Innerhalb von Methoden)

Wir tauchen nun eine Ebene tiefer in den Kaninchenbau. Wussten Sie, dass Sie eine Klasse nicht nur in einer anderen Klasse, sondern sogar **innerhalb einer einzelnen Methode** definieren können?

Diese sogenannten **lokalen Klassen** sind nur innerhalb der geschweiften Klammern der Methode gültig, in der sie geschrieben wurden. Sobald die Methode beendet ist, verschwindet der Bauplan!

Wir nutzen das, wenn wir einen sehr speziellen, komplexen Datentyp benötigen, um einen Algorithmus zu berechnen, diesen Typ aber absolut nirgendwo sonst im Projekt brauchen.

```
public class Navigation {

    public void berechneKuerzestenWeg(int startX, int startY) {

        // Eine lokale Klasse! Sie existiert NUR innerhalb dieser Methode.
        // Sie darf weder public noch private sein.
        class Wegpunkt {
            int x, y;
            double distanz;

            Wegpunkt(int x, int y, double distanz) {
                this.x = x;
                this.y = y;
                this.distanz = distanz;
            }
        }

        // Wir können die Klasse direkt hier instanziieren und nutzen
        Wegpunkt p1 = new Wegpunkt(startX + 1, startY, 1.5);
        Wegpunkt p2 = new Wegpunkt(startX, startY - 1, 2.0);

        System.out.println("Prüfe Wegpunkt bei X: " + p1.x);
    }
}
```

*(Hinweis für Architekten: Lokale Klassen sind in der Praxis sehr selten. Meist lagert man solche Datenstrukturen lieber als private statische innere Klassen aus, um die Methode schlank zu halten.)*

## 13.5 Anonyme Klassen (Definition und Anwendung)

Jetzt kommen wir zu einer Form der inneren Klasse, die Sie in der modernen Java-Entwicklung überall sehen werden. Sie ist die wichtigste Brücke zur funktionalen Programmierung, die wir in Buch 3 intensiv behandeln werden.

Stellen Sie sich vor, wir haben unser Interface Sammelbar aus Kapitel 10:

```
public interface Sammelbar {
    void aufheben();
}
```

Normalerweise müssten wir nun eine komplett neue Klasse schreiben (z.B. Schluessel.java), die implements Sammelbar sagt. Doch was, wenn wir im Hauptprogramm einfach nur ein einziges, völlig unscheinbares Objekt brauchen, das diesen Vertrag erfüllt, und wir uns die Mühe sparen wollen, eine extra Datei dafür anzulegen?

Wir erschaffen eine **Anonyme Klasse**. Es ist eine Klasse ohne Namen, die im exakt selben Atemzug definiert und instanziiert wird!

```
public class Spiel {
    public static void main(String[] args) {

        // Die Magie der anonymen Klasse!
        // Wir rufen 'new' auf dem Interface auf (was eigentlich verboten
        // ist), ABER wir öffnen sofort geschweifte Klammern und liefern die
        // fehlende Methode!
        Sammelbar geheimerBrief = new Sammelbar() {

            // Dies ist der Körper der namenlosen Klasse
            @Override
            public void aufheben() {
                System.out.println("Du liest den Brief. Er zerfällt zu " +
                                   "Staub.");
            }

        }; // ACHTUNG: Semikolon nicht vergessen, da es eine
           // Zuweisungs-Anweisung ist!

        // Wir nutzen das Objekt ganz normal
        geheimerBrief.aufheben();
    }
}
```

Anonyme Klassen sind fantastisch für kleine, einmalige Aktionen. Sie definieren *"ein Objekt, das diesen Vertrag erfüllt, und hier ist der Code dafür"*. (In Kapitel 18 werden wir sehen, wie man diese anonymen Klassen mit sogenannten "Lambdas" noch weiter zusammenschrumpfen kann!)

## 13.6 Zugriff auf Variablen des umschließenden Bereichs (effectively final)

Wenn Sie lokale Klassen oder anonyme Klassen innerhalb einer Methode nutzen, stoßen Sie auf ein architektonisches Sicherheitsproblem. Solche Klassen dürfen nämlich auf die lokalen

Variablen der Methode zugreifen, in der sie definiert wurden.

Schauen wir uns diesen Code an:

```
public void eventStarten() {
    // Eine lokale Variable der Methode
    int goldBelohnung = 500;

    Sammelbar schatz = new Sammelbar() {
        @Override
        public void aufheben() {
            // Die anonyme Klasse greift auf die Variable der Methode zu!
            System.out.println("Du findest " + goldBelohnung +
                               " Goldmünzen!");
        }
    };

    schatz.aufheben();
}
```

Das funktioniert einwandfrei. Doch Java legt hier eine sehr strenge Regel an: **Die innere Klasse darf die lokale Variable goldBelohnung nur lesen, aber niemals verändern!** Und auch Sie dürfen die Variable nach der Erschaffung der anonymen Klasse nicht mehr nachträglich ändern.

Die Variable muss entweder ausdrücklich als final markiert sein, oder sie muss **effectively final** sein (das bedeutet: Sie haben zwar kein final hingeschrieben, Sie ändern den Wert aber faktisch nach der ersten Zuweisung nie wieder).

Wenn Sie versuchen, goldBelohnung = 1000; unter das Objekt zu schreiben, wirft der Compiler einen Fehler: *"local variables referenced from an inner class must be final or effectively final"*.

## Warum ist Java hier so streng?

Die Antwort liegt tief im Arbeitsspeicher des Computers. Erinnern Sie sich (vielleicht aus anderen Kursen) an Heap und Stack? Lokale Variablen (goldBelohnung) leben auf dem Stack. Sie werden sofort vernichtet, wenn die Methode eventStarten() beendet ist. Objekte (schatz) leben aber auf dem Heap und können viel länger existieren als die Methode, in der sie erschaffen wurden (z.B. wenn wir das Objekt als Rückgabewert an das Hauptprogramm zurückgeben).

Wenn das Objekt später irgendwann aufheben() aufruft, ist die Methode längst beendet und die Variable goldBelohnung existiert physikalisch gar nicht mehr!

Damit das Programm nicht abstürzt, macht Java beim Erschaffen des Objekts heimlich eine Kopie der Variable in das Objekt hinein. Und damit diese Kopie und das Original nicht asynchron laufen (was zu furchtbaren Bugs führen würde), verbietet Java einfach jede Änderung an der

Variable.

---

Mit den inneren Klassen haben Sie gelernt, wie man Architekturen auf Mikro-Ebene kapselt. Sie können nun Hilfsklassen vor der Außenwelt verbergen und Verträge (Interfaces) mit eleganten anonymen Einweg-Objekten auf die Schnelle erfüllen.

Im nächsten Kapitel widmen wir uns einem weiteren, sehr speziellen Datentyp. Wir haben gesehen, wie man Konstanten (public static final int MAX_HP = 100) definiert. Was aber, wenn wir eine Konstante brauchen, die keine Zahl ist, sondern ein fester, unveränderlicher Zustand – zum Beispiel die Schwierigkeitsgrade "LEICHT", "MITTEL" und "SCHWER"?

Hierfür bietet uns die Objektorientierung die **Enums (Aufzählungstypen)**.

# 14 Enums (Aufzählungstypen)

Im letzten Kapitel haben wir gelernt, wie wir mit inneren Klassen logisch zusammenhängende Bausteine kapseln. In diesem Kapitel widmen wir uns einem architektonischen Problem, das auftritt, wenn wir Dinge modellieren wollen, die nur eine ganz bestimmte, fest definierte Anzahl von Zuständen annehmen dürfen.

Denken Sie an die Himmelsrichtungen (Nord, Ost, Süd, West), an die Wochentage oder an die Schwierigkeitsgrade in unserem Spiel (Leicht, Mittel, Schwer).

## 14.1 Motivation für typsichere Aufzählungen

Bisher haben wir für solche festen Werte Konstanten (public static final) verwendet, meistens in Kombination mit dem Datentyp int.

Stellen Sie sich vor, wir programmieren die Klasse Held. Ein Held kann eine von drei Klassen haben: Krieger, Magier oder Bogenschütze.

```
// Der veraltete "Handwerker-Weg" (Integer-Konstanten)
public class Held {
    public static final int KRIEGER = 0;
    public static final int MAGIER = 1;
    public static final int BOGENSCHUETZE = 2;

    private String name;
    private int heldenKlasse; // Speichert 0, 1 oder 2

    public Held(String name, int klasse) {
        this.name = name;
        this.heldenKlasse = klasse;
    }
}
```

Das funktioniert im Hauptprogramm scheinbar gut: Held h = new Held("Artus", Held.KRIEGER);.

Doch als Architekt sehen Sie sofort die klaffende Sicherheitslücke: Der Datentyp des Parameters klasse im Konstruktor ist ein völlig normaler int. Nichts und niemand hindert einen anderen Programmierer daran, Folgendes zu schreiben:

```
// Katastrophe! Der Compiler erlaubt es, da 99 ein gültiger Integer ist.
Held kaputterHeld = new Held("Buggy", 99);
```

Wenn wir später im Code prüfen if (heldenKlasse == Held.KRIEGER), wird unser Programm bei dem Wert 99 völlig unvorhersehbares Verhalten zeigen. Wir haben **keine Typsicherheit**. Wir zwingen den Compiler nicht dazu, unsere logischen Grenzen zu überprüfen.

Die Lösung für dieses Problem sind **Enums** (Kurzform für *Enumerations*, also Aufzählungen).

## 14.2 Definition von Enums

Ein Enum ist in Java eine besondere Form einer Klasse. Anstatt class oder interface verwenden wir das Schlüsselwort enum. Innerhalb des Enums definieren wir einfach eine kommagetrennte Liste aller erlaubten Werte. Diese Werte werden konventionsgemäß komplett großgeschrieben.

Lassen Sie uns das Problem unseres Helden elegant lösen:

```
// Datei: HeldenKlasse.java
// Wir erschaffen einen völlig neuen, typsicheren Datentyp!
public enum HeldenKlasse {
    KRIEGER,
    MAGIER,
    BOGENSCHUETZE
}
```

Was passiert hier architektonisch im Hintergrund? Der Java-Compiler ist unglaublich fleißig. Er übersetzt dieses kurze Konstrukt heimlich in eine vollwertige final class, in der KRIEGER, MAGIER und BOGENSCHUETZE fest verdrahtete, unveränderliche Objekte (Instanzen) dieser Klasse sind. Es kann und wird im gesamten Java-Universum niemals ein viertes Objekt dieses Typs geben!

Bauen wir das Enum in unsere Held-Klasse ein:

```
public class Held {
    private String name;
    // Der Datentyp ist nicht mehr 'int', sondern unser neues Enum!
    private HeldenKlasse klasse;

    public Held(String name, HeldenKlasse klasse) {
        this.name = name;
        this.klasse = klasse;
    }
}
```

Wenn jemand nun versucht, Unsinn in den Konstruktor zu werfen, schlägt der Compiler gnadenlos zu:

```
public class Spiel {
    public static void main(String[] args) {
        // PERFEKT: Typsicher und extrem lesbar.
        Held h1 = new Held("Artus", HeldenKlasse.KRIEGER);

        // COMPILERFEHLER! 99 ist nicht vom Typ 'HeldenKlasse'.
        // Held h2 = new Held("Buggy", 99);
```

```
        // COMPILERFEHLER! Auch Strings sind verboten.
        // Held h3 = new Held("Merlin", "MAGIER");
    }
}
```

Wir haben die Fehlerquelle durch starke Typisierung (Strong Typing) vollständig eliminiert.

## 14.3 Nutzung und Methoden von Enums

Da Enums im Hintergrund echte Java-Klassen sind, erben sie von einer eingebauten Java-Urklasse namens java.lang.Enum. Dadurch bringen sie von Haus aus fantastische Methoden mit, die uns das Leben leichter machen.

## 1. Die values()-Methode

Diese Methode liefert uns ein Array mit allen definierten Werten des Enums zurück, und zwar exakt in der Reihenfolge, in der wir sie aufgeschrieben haben. Das ist perfekt, um z.B. ein Auswahlmenü für den Spieler auf den Bildschirm zu drucken:

```
System.out.println("Wähle deine Klasse:");
for (HeldenKlasse hk : HeldenKlasse.values()) {
    System.out.println("- " + hk.name()); // .name() liefert den Bezeichner
                                          // als Text (String)
}
```

## 2. Enums in der modernen switch-Anweisung (Java SE 14+)

Enums und switch-Anweisungen sind architektonisch beste Freunde. Mit den modernen *Switch Expressions* in Java (die auf Pfeile -> statt auf fehleranfällige case: und break; setzen) können wir extrem eleganten Code schreiben.

Der geniale Vorteil: Wenn Sie ein Enum im Switch verwenden, weiß der Compiler genau, wie viele mögliche Werte es gibt. Wenn Sie einen Wert vergessen, kann der Compiler Sie warnen!

```
public void klassenFaehigkeitNutzen(HeldenKlasse hk) {

    // Das moderne Switch (ohne 'break').
    // Der Compiler prüft, ob wir alle Werte des Enums behandelt haben!
    switch (hk) {
        case KRIEGER      -> System.out.println("Wirbelt mit der Axt!");
        case MAGIER       -> System.out.println("Wirft einen Feuerball!");
        case BOGENSCHUETZE -> System.out.println("Schießt einen Pfeil!");
    }
}
```

## 3. Das Architektur-Geheimnis: Enums mit Attributen und Konstruktoren

In vielen älteren Programmiersprachen (wie C oder C++) sind Enums wirklich nur versteckte Integer-Zahlen. In Java sind sie **vollwertige Objekte**!

Das bedeutet, wir können unseren Enum-Werten eigene Attribute, Methoden und sogar einen privaten Konstruktor geben.

Stellen Sie sich vor, jede Heldenklasse soll von Beginn an feste Start-Lebenspunkte haben. Anstatt diese Logik mühsam mit unzähligen if-Abfragen im Hauptprogramm zu verteilen, geben wir dem Enum dieses Wissen einfach selbst mit!

```
public enum HeldenKlasse {
    // 1. Wir rufen beim Erschaffen der Enum-Werte den eigenen Konstruktor
    // auf!
    KRIEGER(150),
    MAGIER(80),
    BOGENSCHUETZE(110);

    // 2. Wir definieren ein normales Attribut für die Instanzen
    private final int startLebenspunkte;

    // 3. Der Konstruktor MUSS in Enums immer private sein!
    // (Java erschafft die Objekte KRIEGER etc. intern, niemand darf von
    // außen 'new' rufen).
    private HeldenKlasse(int startLebenspunkte) {
        this.startLebenspunkte = startLebenspunkte;
    }

    // 4. Ein ganz normaler Getter
    public int getStartLebenspunkte() {
        return this.startLebenspunkte;
    }
}
```

Schauen Sie sich an, wie unfassbar elegant unser Hauptprogramm dadurch wird:

```
public class Spiel {
    public static void main(String[] args) {
        HeldenKlasse gewaehlteKlasse = HeldenKlasse.MAGIER;

        System.out.println("Du hast " + gewaehlteKlasse.name() +
                           " gewählt.");
        System.out.println("Deine Start-HP: " +
                           gewaehlteKlasse.getStartLebenspunkte());
        // Ausgabe: Deine Start-HP: 80
    }
```

```
}
```

Das ist echte Datenkapselung auf höchstem Niveau. Das Wissen darüber, wie viele Lebenspunkte ein Magier hat, liegt zentral im Enum HeldenKlasse und schwirrt nicht im restlichen Projekt umher.

---

Sie haben in diesem Kapitel gelernt, wie Sie Zustände limitieren, die Typsicherheit erhöhen und Code extrem wartbar machen, indem Sie das enum-Konstrukt verwenden.

Im nächsten Kapitel lernen wir den jüngsten und vielleicht revolutionärsten Bruder der Klasse kennen (eingeführt in Java 14). Manchmal wollen wir keine komplexe Logik, keine verschachtelten Methoden und keine Veränderbarkeit. Manchmal wollen wir einfach nur einen reinen, fehlerfreien Datencontainer für Informationen, den wir in einer einzigen Zeile definieren können.

Hier betreten die **Records** die Bühne – der modernste Weg, Daten in Java zu bündeln.

# 15 Records (Unveränderliche Datencontainer)

Im vorherigen Kapitel haben wir mit Enums gelernt, wie wir eine feste, unveränderliche Menge an Zuständen definieren. Doch sehr oft stehen wir als Architekten vor einem anderen Problem: Wir wollen einfach nur einen reinen, simplen Datencontainer erschaffen, um ein paar zusammengehörige Werte von A nach B zu transportieren – zum Beispiel die X- und Y-Koordinaten für einen Punkt auf unserer Dungeon-Karte.

## 15.1 Motivation: Vermeidung von Boilerplate-Code

Lassen Sie uns versuchen, diesen scheinbar trivialen Datencontainer mit dem bisherigen "Handwerker-Wissen" als normale Klasse zu programmieren. Wir wollen, dass der Punkt sicher ist (Kapselung) und dass wir zwei Punkte korrekt miteinander vergleichen können (Objektgleichheit).

```
// Der mühsame Weg: Eine klassische Daten-Klasse (POJO)
public class Punkt {
    private final int x;
    private final int y;

    // 1. Konstruktor
    public Punkt(int x, int y) {
        this.x = x;
        this.y = y;
    }

    // 2. Getter (Setter lassen wir weg, da der Punkt unveränderlich sein
    // soll)
    public int getX() { return x; }
    public int getY() { return y; }

    // 3. equals-Methode für echten Objektvergleich
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        Punkt punkt = (Punkt) obj;
        return x == punkt.x && y == punkt.y;
    }

    // 4. hashCode (muss immer mit equals überschrieben werden!)
    @Override
    public int hashCode() {
        return java.util.Objects.hash(x, y);
    }

    // 5. toString für eine saubere Konsolenausgabe
```

```
    @Override
    public String toString() {
        return "Punkt[x=" + x + ", y=" + y + "]";
    }
}
```

Schauen Sie sich diesen Code-Berg an! Wir haben fast 30 Zeilen Text geschrieben, nur um zwei simple Zahlen (int x und int y) vernünftig zu speichern. In der professionellen Softwareentwicklung nennt man diesen lästigen, immer wiederkehrenden und aufblähenden Code **Boilerplate-Code** (Schablonencode). Er verdeckt die eigentliche Architektur und macht das Lesen des Programms anstrengend.

Seit Java 14 (und als finaler Standard fest verankert in modernen Versionen wie unserem Java SE 25) bietet die Sprache eine revolutionäre Lösung für exakt dieses Problem.

## 15.2 Definition und Syntax

Wir können den gesamten 30-Zeilen-Koloss von oben durch ein einziges, neues Schlüsselwort ersetzen: **record**.

Ein Record ist eine spezielle, stark komprimierte Form einer Klasse. Sie definieren ihn, indem Sie hinter dem Namen direkt in runden Klammern die sogenannten *Komponenten* (die Daten, die er halten soll) auflisten.

```
// Der moderne Weg: Ein Record
// Diese EINE Zeile ersetzt den kompletten 30-Zeilen-Code von oben!
public record Punkt(int x, int y) {
}
```

Was passiert hier architektonisch? Wenn der Java-Compiler das Wort record sieht, läuft er zur Höchstform auf. Er erschafft unsichtbar im Hintergrund eine vollwertige Klasse und generiert all den lästigen Boilerplate-Code völlig automatisch!

Folgendes baut der Compiler für Sie in die Klasse Punkt ein:

1. private final Attribute für x und y.
2. Einen Konstruktor, der genau diese beiden Werte verlangt.
3. Perfekt ausprogrammierte equals()- und hashCode()-Methoden.
4. Eine wunderschöne toString()-Methode.
5. **Achtung, eine wichtige Änderung:** Die Getter-Methoden heißen bei Records nicht mehr getX() und getY(). Da sie reine Datenabrufe sind, heißen die Methoden exakt so wie die Variablen: x() und y().

## 15.3 Nutzung und Eigenschaften von Records

Im Hauptprogramm verwenden wir den Record exakt so, als hätten wir eine normale Klasse geschrieben. Wir instanziieren ihn mit new.

```
public class Spiel {
    public static void main(String[] args) {
        // Erschaffung über den automatisch generierten Konstruktor
        Punkt p1 = new Punkt(5, 10);
        Punkt p2 = new Punkt(5, 10);

        // Zugriff über die neuen, schlanken Getter
        System.out.println("Die X-Koordinate ist: " + p1.x());

        // Die toString()-Methode wird automatisch wunderschön formatiert!
        System.out.println(p1);
        // Ausgabe: Punkt[x=5, y=10]

        // equals() funktioniert "out of the box" perfekt auf Daten-Ebene
        if (p1.equals(p2)) {
            System.out.println("Punkte liegen auf derselben Koordinate!");
        }
    }
}
```

## Die Unveränderlichkeit (Immutability)

Eine der wichtigsten architektonischen Eigenschaften von Records ist ihre absolute **Unveränderlichkeit**. Alle Attribute in einem Record sind von Natur aus final. Es gibt keine Setter-Methoden!

Wenn Sie einen Punkt(5, 10) haben und ihn auf X=6 verschieben wollen, können Sie nicht p1.x = 6; schreiben. Sie **müssen** ein komplett neues Record-Objekt erschaffen: Punkt pNeu = new Punkt(6, 10);.

*Warum ist das fantastisch?*

Unveränderliche Objekte sind der Heilige Gral der modernen Softwareentwicklung. Wenn ein Objekt seinen Zustand niemals ändern kann, kann es niemals durch fremde Methoden (Seiteneffekte) beschädigt werden.

Wenn Sie Band 2 (Funktionale Programmierung) und Band 3 (Parallele Programmierung) lesen werden, werden Sie Records lieben lernen! Da sie unveränderlich sind, können tausende parallele Threads gleichzeitig auf denselben Record zugreifen, ohne dass jemals das Chaos von *Race Conditions* oder blockierten Speichern entsteht.

## Erweiterung von Records (Kompakter Konstruktor)

Obwohl der Compiler alles automatisch generiert, lässt er uns als Architekten nicht im Stich, wenn wir eingreifen wollen. Was passiert, wenn wir verhindern wollen, dass jemand einen Punkt mit negativen Koordinaten erschafft?

Wir können in einem Record einen sogenannten **kompakten Konstruktor** definieren. Wir lassen die Parameterliste einfach weg und fügen nur unsere Prüf-Logik ein!

```
public record Punkt(int x, int y) {

    // Ein kompakter Konstruktor! Keine Parameter in (), keine Zuweisungen
    // this.x = x.
    // Dieser Block wird automatisch aufgerufen, BEVOR die Zuweisung
    // passiert.
    public Punkt {
        if (x < 0 || y < 0) {
            throw new IllegalArgumentException(
                        "Koordinaten dürfen nicht negativ sein!");
        }
    }

    // Wir können auch völlig normale, zusätzliche Methoden in den Record
    // einbauen:
    public int berechneAbstandZumUrsprung() {
        return x + y; // Stark vereinfachte Mathematik für das Beispiel
    }
}
```

## Die Grenzen von Records

Als Architekt müssen Sie wissen, wo die Grenzen Ihrer Werkzeuge liegen:

1. Records können **nicht** von anderen Klassen erben (sie erben unsichtbar von java.lang.Record).
2. Niemand kann von Ihrem Record erben (Records sind implizit final).
3. Records dürfen **keine** eigenen, zusätzlichen Instanzvariablen (Zustand) haben, die nicht im "Kopf" (in den runden Klammern oben) definiert wurden.

Sie dürfen aber sehr wohl Interfaces implementieren!

```
public record Punkt(int x, int y) implements Verschiebbar {
    @Override
    public Punkt verschieben(int deltaX, int deltaY) {
        // Da wir unveränderlich sind, geben wir einen NEUEN Punkt zurück!
        return new Punkt(this.x + deltaX, this.y + deltaY);
    }
```

```
}
```

Mit Records haben Sie das ultimative Werkzeug kennengelernt, um Daten effizient, sicher und extrem lesbar durch Ihr Programm zu transportieren. Sie ersparen sich hunderte Zeilen Boilerplate-Code und zwingen sich selbst zu einer sauberen, unveränderlichen Architektur.

---

Wir haben nun alle Bauformen von Objekten (Klassen, Abstrakte Klassen, Interfaces, Enums, Records) besprochen. Sie beherrschen die Syntax und die Architektur-Prinzipien.

Im vorletzten Teil dieses Buches (Teil VI: Die Werkzeuge der Profis) öffnen wir nun den gewaltigen Werkzeugkoffer, den die Macher von Java uns kostenlos mitliefern. Warum sollten wir alles selbst programmieren, wenn die Standardbibliothek tausende vorgefertigte Bausteine enthält?

Wir starten in Kapitel 16 mit einem Überblick über die **JDK Klassenbibliothek** und die wichtigen Basis- und Wrapper-Klassen.

# Teil VI: Die Werkzeuge der Profis

# 16 Die JDK Klassenbibliothek

Bisher haben wir in diesem Buch fast alles von Grund auf selbst gebaut. Wir haben eigene Klassen für Monster, Waffe und Inventar geschrieben. Wir haben eigene Exceptions und eigene generische Käfige entworfen. Als angehender Architekt mussten Sie lernen, wie man Ziegelsteine brennt und Mörtel anmischt.

Doch wenn Sie in der realen Welt einen Wolkenkratzer bauen, stellen Sie sich nicht hin und gießen jede Schraube selbst. Sie greifen auf standardisierte, millionenfach erprobte Bauteile zurück. Genau das bietet Ihnen das **JDK (Java Development Kit)**. Es bringt eine gigantische Bibliothek mit tausenden vorgefertigten Klassen und Interfaces mit. Wer diese Bibliothek meistert, spart sich Jahre an Entwicklungszeit.

## 16.1 Übersicht über wichtige Pakete

Die Klassenbibliothek von Java ist, genau wie unser eigener Code in Kapitel 4, streng in Pakete (packages) unterteilt. Hier ist eine kleine Übersicht der wichtigsten Werkzeugkisten, denen Sie in Ihrer Karriere ständig begegnen werden:

- **java.lang**: Das absolute Herzstück von Java. Es enthält die fundamentalsten Klassen wie Object, String, Math, System und Thread. Wie wir in Kapitel 4 gelernt haben, wird dieses Paket völlig automatisch in jede Ihrer Dateien importiert.
- **java.util**: Der "Utility"-Werkzeugkoffer. Hier finden Sie Zufallsgeneratoren (Random), Datums- und Zeitklassen und vor allem die berühmte Collection-API (Listen, Mengen, Wörterbücher), der wir uns im nächsten Kapitel widmen.
- **java.io und java.nio**: Die Werkzeuge für Input und Output. Wenn Sie Dateien von der Festplatte lesen, Ordner erstellen oder Daten über ein Netzwerk schicken wollen, finden Sie hier die passenden Klassen.
- **java.net**: Alles für die Netzwerkprogrammierung (Sockets, HTTP-Verbindungen).

In diesem Kapitel picken wir uns die absoluten Basis-Bausteine heraus, die in fast jedem Java-Programm vorkommen.

## 16.2 Wichtige Basisklassen: String und Math

## Die Besonderheit der Klasse String

Wir haben String von Tag eins an benutzt, um Texte zu speichern. Doch als Architekt müssen Sie nun verstehen, was ein String unter der Haube wirklich ist: Es ist ein Objekt. String ist eine Klasse im Paket java.lang.

Sie hat jedoch zwei massive architektonische Besonderheiten:

1. **Syntaktischer Zucker:** Obwohl String ein Objekt ist, dürfen wir es erschaffen, ohne das

Schlüsselwort new zu benutzen. Java erlaubt die Kurzschreibweise mit doppelten Anführungszeichen.

2. **Absolute Unveränderlichkeit (Immutability):** Die Klasse String ist final und ihre internen Daten sind gekapselt und unveränderlich. Einmal erschaffen, kann ein String-Objekt niemals wieder verändert werden! (Genau wie unsere Records aus Kapitel 15).

Schauen Sie sich diese tückische Code-Falle an:

```
public class TextSpielerei {
    public static void main(String[] args) {
        String gruss = "Hallo";

        // Wir rufen eine Methode AUF dem Objekt auf, die es großschreiben
        //  soll.
        gruss.toUpperCase();

        System.out.println(gruss);
        // Ausgabe: Hallo  <-- Huch? Warum ist es nicht groß?
    }
}
```

Warum ist die Ausgabe immer noch "Hallo"? Weil das Objekt unveränderlich ist! Die Methode .toUpperCase() verändert nicht das Original, sondern sie erschafft heimlich ein **komplett neues String-Objekt** im Speicher und gibt dieses als Rückgabewert (Return) zurück. Da wir das neue Objekt im obigen Code in keiner Variablen auffangen, verpufft es einfach im Nichts.

Der korrekte Weg lautet:

```
String gruss = "Hallo";
// Wir überschreiben die Referenz unserer Variablen mit dem NEUEN Objekt!
gruss = gruss.toUpperCase();
System.out.println(gruss); // Ausgabe: HALLO
```

## Die Utility-Klasse Math

Die Klasse Math ist ein hervorragendes Beispiel für ein weiteres Architektur-Muster: die reine Werkzeugklasse (Utility Class).

Erinnern Sie sich an statische Methoden aus Kapitel 3? Die Macher von Java haben entschieden, dass es keinen Sinn ergibt, ein Objekt von der Mathematik zu erschaffen (new Math()). Stattdessen haben sie den Konstruktor der Klasse private gemacht und alle Methoden public static.

```
public class MathematikTest {
    public static void main(String[] args) {
```

```
        // Wir rufen die Methoden direkt über den Klassennamen auf
        double wurzel = Math.sqrt(16.0);
        int maximum = Math.max(10, 50);
        double zufall = Math.random(); // Liefert eine Zahl zwischen 0.0 und
                                       // 1.0

        System.out.println("Max: " + maximum + ", Wurzel: " + wurzel);
    }
}
```

## 16.3 Wrapper-Klassen für primitive Datentypen

Nun stoßen wir auf einen historischen Riss in der Architektur von Java. Java ist nicht zu 100 % objektorientiert. Aus reinen Leistungsgründen gibt es die sogenannten primitiven Datentypen: int, double, boolean, char. Sie sind extrem schnell, verbrauchen kaum Speicher, aber – **sie sind keine Objekte!** Sie haben keine Methoden und sie erben nicht von der Ur-Klasse Object.

Normalerweise ist das kein Problem. Doch im letzten Kapitel (Kapitel 12: Generics) haben wir gelernt, dass wir typsichere Schablonen <T> bauen können.

**Die eiserne Regel der Generics lautet:** Das <T> darf nur durch echte Objekte (Referenztypen) ersetzt werden!

```
Schatzkiste<String> textKiste = new Schatzkiste<>();
  // OK, String ist ein Objekt
Schatzkiste<Monster> monsterKiste = new Schatzkiste<>(); // OK

// COMPILERFEHLER! int ist kein Objekt.
// Schatzkiste<int> zahlenKiste = new Schatzkiste<>();
```

Was tun wir, wenn wir eine generische Kiste voll mit ganzen Zahlen brauchen?

Hier kommen die **Wrapper-Klassen** (Hüllklassen) ins Spiel. Java bietet für jeden der acht primitiven Datentypen eine passende Objekt-Klasse an, die als "Verpackung" dient:

- int -> Integer
- double -> Double
- boolean -> Boolean
- char -> Character
- (und analog für byte, short, long, float)

Ein Integer-Objekt ist also im Grunde ein ganz normales Objekt, das tief in seinem Inneren als einzigen Lebenszweck ein privates int-Attribut speichert.

```
// Der klassische Weg (bis Java 4): Manuelles Verpacken
Integer verpackteZahl = Integer.valueOf(42);
```

```
// Jetzt können wir die Wrapper-Klasse in Generics nutzen!
Schatzkiste<Integer> zahlenKiste = new Schatzkiste<>();
zahlenKiste.hineinlegen(verpackteZahl);

// Manuelles Entpacken, um wieder normal damit zu rechnen:
Integer geholt = zahlenKiste.herausnehmen();
int primitiveZahl = geholt.intValue();
```

## 16.4 Autoboxing und Unboxing

Wenn Sie den Code oben betrachten, werden Sie feststellen, dass das ständige manuelle Verpacken (valueOf) und Entpacken (intValue) extrem viel Schreibarbeit bedeutet. Code sollte flüssig lesbar sein!

Deshalb führte Java ab Version 5 das sogenannte **Autoboxing** und **Unboxing** ein. Der Compiler übernimmt nun die lästige Arbeit des Ein- und Auspackens völlig automatisch im Hintergrund.

Wir dürfen primitive Datentypen und ihre Wrapper-Objekte ab sofort im Code fast nahtlos mischen!

```
public class ZahlenMagie {
    public static void main(String[] args) {

        // AUTOBOXING: Wir weisen einen primitiven int (100) einer
        // Integer-Variablen zu.
        // Der Compiler macht heimlich daraus: Integer objektZahl =
        // Integer.valueOf(100);
        Integer objektZahl = 100;

        // UNBOXING: Wir weisen das Integer-Objekt einem primitiven int zu.
        // Der Compiler macht heimlich daraus:
        // int primitiv = objektZahl.intValue();
        int primitiv = objektZahl;

        // Wir können sogar direkt mit Objekten rechnen!
        // Hier entpackt Java das Objekt, addiert 50, und verpackt das
        // Ergebnis in ein neues Objekt.
        Integer ergebnis = objektZahl + 50;

        // Die perfekte Symbiose mit Generics:
        Schatzkiste<Integer> kiste = new Schatzkiste<>();

        // Wir übergeben einen primitiven int.
        // Java führt ein Autoboxing zu Integer durch, damit es in die Kiste
        // <T> passt!
        kiste.hineinlegen(99);
```

```
    }
}
```

## Die Performance-Falle für Architekten:

Autoboxing ist unfassbar bequem. Doch ein guter Architekt weiß, was unter der Haube passiert. Schauen Sie sich diese unscheinbare Schleife an:

```
Integer summe = 0;
for (int i = 0; i < 100000; i++) {
    summe = summe + 1; // ACHTUNG!
}
```

Was passiert hier wirklich? Weil summe ein Objekt ist (Integer) und Objekte (genau wie String) unveränderlich sind, muss Java in jedem Schleifendurchlauf das Objekt entpacken, 1 addieren und ein **komplett neues Objekt** erschaffen.

Sie haben soeben mit drei harmlosen Zeilen Code völlig sinnlos 100.000 Objekte im Arbeitsspeicher erzeugt, die der Garbage Collector von Java mühsam wieder aufräumen muss!

*Die goldene Regel lautet:* Wenn Sie massiv rechnen oder riesige Schleifen bauen, nutzen Sie immer die primitiven Datentypen (int, double). Nutzen Sie die Wrapper-Klassen (Integer) nur dann, wenn Sie zwingend Objekte brauchen – zum Beispiel, um sie in generische Strukturen einzufügen.

---

Sie kennen nun die fundamentale Basis der Java-Bibliothek. Sie verstehen die Unveränderlichkeit von Strings und wissen, wie Sie dank Autoboxing und Wrapper-Klassen die Brücke zwischen performanten primitiven Typen und der typsicheren Objektwelt schlagen.

Und genau dieses Wissen über Generics und Wrapper-Klassen ist die absolute Voraussetzung für das nächste Kapitel. Wir betreten nun das Herzstück der Java-Standardbibliothek: **Die Collection-API**. Wir werfen unsere primitiven Arrays, mit denen wir uns seit dem ersten Buch herumschlagen, endgültig über Bord und lernen, wie echte Software-Architekten dynamisch wachsende Listen, Sets und Maps nutzen.

# 17 Die Collection-API

Im ersten Buch (imperative Programmierung) haben wir gelernt, wie wir mehrere Werte des gleichen Typs in einem **Array** speichern können. Arrays waren ein großartiges Werkzeug, um unseren Code von hunderten Einzelvariablen (monster1, monster2...) zu befreien. In Kapitel 7 dieses Buches haben wir polymorphe Arrays (Monster[] dungeon = new Monster[3];) genutzt, um verschiedene Gegner-Typen in einem Raum zu verwalten.

Doch Arrays haben eine fatale, architektonische Schwachstelle.

## 17.1 Motivation für Collections: Sammlungen von Werten

Die eiserne Regel des Arrays lautet: **Seine Größe ist fest und unveränderlich.** Wenn Sie new Monster[3] schreiben, gießt Java exakt drei Speicherplätze in Beton. Was passiert nun, wenn unser Held in diesem Raum einen Alarm auslöst und ein viertes Monster durch die Tür stürmt?

Unser Array ist voll. Wir können das vierte Monster nicht hinzufügen. Wenn wir es zwingend speichern müssen, bleibt uns nur ein furchtbar ineffizienter "Handwerker-Weg": Wir müssen ein neues, größeres Array erschaffen (z.B. Größe 6), in einer for-Schleife alle alten Monster mühsam in das neue Array kopieren und dann das vierte Monster auf Platz 3 setzen. Das alte Array wird weggeworfen.

Ein Architekt baut keine Systeme, die bei jeder kleinen Änderung abgerissen und neu gebaut werden müssen. Wir brauchen **Collections** (Sammlungen) – Datenstrukturen, die intelligent genug sind, von ganz alleine zu wachsen und zu schrumpfen.

## 17.2 Dynamische Datenstrukturen: Eigene Implementierung einer ArrayList

Bevor wir Javas fertige Magie nutzen, schauen wir hinter den Vorhang. Wie baut man ein Array, das scheinbar wachsen kann? Die Antwort ist: Wir kapseln das dumme Array einfach in einem intelligenten Objekt!

Dank unseres Wissens über Generics (<T>) und Kapselung (private) können wir uns eine eigene, dynamisch wachsende Liste bauen. Wir nennen sie EigeneArrayList.

```
public class EigeneArrayList<T> {

    // Das dumme, starre Array wird vor der Außenwelt versteckt!
    private Object[] internesArray;
    private int anzahlElemente; // Wie viele Plätze sind WIRKLICH belegt?

    public EigeneArrayList() {
        // Wir starten heimlich mit Platz für 2 Elemente
```

```
        this.internesArray = new Object[2];
        this.anzahlElemente = 0;
    }

    public void hinzufuegen(T neuesElement) {
        // 1. Prüfen: Ist unser internes Array voll?
        if (this.anzahlElemente == this.internesArray.length) {
            arrayVergroessern();
        }

        // 2. Element an die nächste freie Stelle setzen
        this.internesArray[anzahlElemente] = neuesElement;
        this.anzahlElemente++; // Zähler erhöhen
    }

    // Die architektonische Geheimwaffe (private!)
    private void arrayVergroessern() {
        System.out.println("LOG: Internes Array ist voll. Verdopple " +
                           "Kapazität!");

        // Wir bauen ein neues Array, das doppelt so groß ist
        Object[] neuesArray = new Object[this.internesArray.length * 2];

        // Wir kopieren die alten Daten herüber
        for (int i = 0; i < this.internesArray.length; i++) {
            neuesArray[i] = this.internesArray[i];
        }

        // Wir tauschen das alte Array gegen das neue aus!
        this.internesArray = neuesArray;
    }

    // Methode zum Herausholen (mit sicherem Downcast, da wir nur T
    // hineinlassen)
    @SuppressWarnings("unchecked")
    public T get(int index) {
        if (index < 0 || index >= anzahlElemente) {
            throw new IndexOutOfBoundsException("Ungültiger Index!");
        }
        return (T) this.internesArray[index];
    }
}
```

Schauen Sie sich an, wie elegant unser Hauptprogramm nun wird. Das Hauptprogramm merkt überhaupt nicht mehr, dass im Hintergrund Arrays kopiert und weggeworfen werden. Es sieht nur ein Objekt, das unendlich viel Platz bietet:

```
public class Spiel {
    public static void main(String[] args) {
```

```
        EigeneArrayList<String> inventar = new EigeneArrayList<>();

        inventar.hinzufuegen("Schwert");
        inventar.hinzufuegen("Schild");

        // Hier ist das interne Array voll (Größe 2).
        // Beim nächsten Aufruf feuert unsere geheime Methode
        // arrayVergroessern()!
        inventar.hinzufuegen("Heiltrank");

        System.out.println("An Index 2 liegt: " + inventar.get(2));
        // Ausgabe: Heiltrank
    }
}
```

## 17.3 Verkettete Listen (Prinzip & eigene Implementierung)

Die ArrayList ist schnell, wenn wir Elemente am Ende hinzufügen oder über einen Index abrufen (get(5)). Doch was ist, wenn wir ständig Objekte ganz *vorne* einfügen wollen? Dann müssten wir im internen Array bei jedem Einfügen alle bisherigen 10.000 Elemente mühsam um einen Platz nach hinten schieben!

Es gibt einen völlig anderen architektonischen Ansatz, um Datenketten zu bilden, der überhaupt keine Arrays nutzt: die **Verkettete Liste** (Linked List).

Das Prinzip beruht auf unserem Wissen über *Subobjekte* (Kapitel 3) und *Innere Klassen* (Kapitel 13). Stellen Sie sich jedes Element in der Liste nicht als Fach in einem Regal vor, sondern als Waggon eines Zuges. Jeder Waggon enthält seine Fracht (die Daten) und ein Seil, das ihn mit dem *nächsten* Waggon verbindet (eine Referenz).

Bauen wir eine EigeneLinkedList:

```
public class EigeneLinkedList<T> {

    // ---------------------------------------------------------
    // Die innere Klasse für die Waggons (Nodes/Knoten)
    private class Knoten {
        T daten;              // Die Fracht
        Knoten naechster;     // Das Zugseil zum nächsten Waggon

        public Knoten(T daten) {
            this.daten = daten;
            this.naechster = null; // Zuerst hängt hinten nichts dran
        }
    }
    // ---------------------------------------------------------
```

```
    private Knoten kopf; // Die Lokomotive (der Start der Kette)

    // Ein Element ganz VORNE einfügen (Das geht hier blitzschnell!)
    public void vorneEinfuegen(T neuesElement) {
        Knoten neuerWaggon = new Knoten(neuesElement);

        // 1. Der neue Waggon hängt sein Seil an die bisherige Lok
        neuerWaggon.naechster = this.kopf;

        // 2. Der neue Waggon wird zur neuen Lok!
        this.kopf = neuerWaggon;
    }

    public void allesDrucken() {
        Knoten aktuell = this.kopf; // Wir starten ganz vorne

        // Solange es noch Waggons gibt...
        while (aktuell != null) {
            System.out.print(aktuell.daten + " -> ");
            aktuell = aktuell.naechster; // Wir hangeln uns am Seil zum
                                         // nächsten Waggon!
        }
        System.out.println("ENDE");
    }
}

// Nutzung der verketteten Liste:
EigeneLinkedList<String> zauber = new EigeneLinkedList<>();
zauber.vorneEinfuegen("Feuerball");
zauber.vorneEinfuegen("Eisblitz");
zauber.vorneEinfuegen("Heilung");

zauber.allesDrucken();
// Ausgabe: Heilung -> Eisblitz -> Feuerball -> ENDE
```

Wir haben hier nichts kopiert und nichts verschoben. Wir haben einfach neue Objekte im Speicher erschaffen und sie mit Referenzen (dem Seil) verknüpft! Das ist echte Objektorientierung.

## 17.4 Typen von Collections (Liste, Menge, Map)

Wir haben nun verstanden, wie Listen unter der Haube funktionieren. Das Großartige ist: Sie müssen diese Klassen nie wieder selbst schreiben! Das java.util-Paket bietet Ihnen perfekt optimierte Versionen davon.

Doch bevor wir den Code betrachten, müssen wir verstehen, dass "Sammlung" nicht gleich "Sammlung" ist. Das Java Collection-Framework unterteilt sich in drei große architektonische Konzepte (Interfaces):

1. **Die Liste (List):** Eine geordnete Sammlung. Die Elemente haben eine feste Reihenfolge (Index 0, 1, 2...). Ein Element darf mehrfach vorkommen (Duplikate sind erlaubt). *Beispiel: Das Inventar des Helden (zwei identische Heiltränke sind erlaubt).*
2. **Die Menge (Set):** Eine ungeordnete Sammlung. Die absolute Regel eines Sets: **Kein Element darf doppelt existieren!** Wenn Sie versuchen, ein Objekt hinzuzufügen, das bereits drin ist, ignoriert das Set den Befehl lautlos. *Beispiel: Die Liste der freigeschalteten Erfolge/Achievements.*
3. **Das Wörterbuch (Map):** *(Streng genommen kein Erbe des Collection-Interfaces, aber Teil des Frameworks).* Eine Map speichert keine einzelnen Elemente, sondern **Schlüssel-Wert-Paare** (Key-Value Pairs). Sie werfen einen Schlüssel (z.B. einen Namen) hinein und erhalten den dazugehörigen Wert (z.B. ein Monster-Objekt) zurück. *Beispiel: Ein Telefonbuch oder eine Spieler-Datenbank (Key: SpielerID, Value: SpielerObjekt).*

## 17.5 Interfaces und Klassen der Collection-API (java.util)

Wie ist die Standardbibliothek aufgebaut? Wie Sie in Kapitel 10 gelernt haben, trennt ein gutes Design immer den Vertrag (interface) von der konkreten Implementierung (class).

Wenn wir in Java programmieren, nutzen wir als Datentyp für die Variable **immer das Interface** (List, Set, Map) und erschaffen mit new die konkrete Implementierung (ArrayList, HashSet, HashMap). Das ist Polymorphie in Reinkultur!

## Arbeiten mit Listen (List)

```
import java.util.List;
import java.util.ArrayList;
import java.util.LinkedList;

public class ListenTest {
    public static void main(String[] args) {

        // Polymorphie: Die Variable ist vom Typ List. Das Objekt ist eine
        // ArrayList.
        // Wir nutzen den Wrapper-Typ Integer (Autoboxing), da primitive int
        // nicht erlaubt sind!
        List<Integer> highscores = new ArrayList<>();

        highscores.add(1500); // Index 0
        highscores.add(800);  // Index 1
        highscores.add(1500); // Index 2 (Duplikate sind erlaubt!)

        // Zugriff
        System.out.println("Bester Score: " + highscores.get(0));
        System.out.println("Anzahl der Einträge: " + highscores.size());
```

```
        // Die moderne For-Each-Schleife (funktioniert mit allen
        // Collections!)
        for (Integer score : highscores) {
            System.out.println("- " + score);
        }

        // Architektur-Bonus:
        // Da 'highscores' vom Typ List ist, könnten wir die Implementierung
        // jederzeit tauschen:
        // highscores = new LinkedList<>();
        // Der Rest des Codes würde exakt gleich bleiben!
    }
}
```

*Moderne Kurzschreibweise (Java 9+):* Wenn Sie eine Liste brauchen, die sich nie wieder verändern soll (Immutability), nutzen Sie die Factory-Methode List.of():

```
List<String> wochentage = List.of("Mo", "Di", "Mi");
```

## Arbeiten mit Mengen (Set)

Ein Set ist gnadenlos. Es nutzt die .equals()- und .hashCode()-Methoden (die wir bei unseren Records in Kapitel 15 kennengelernt haben), um Duplikate zu vernichten.

```
import java.util.Set;
import java.util.HashSet;

public class SetTest {
    public static void main(String[] args) {
        Set<String> entdeckteOrte = new HashSet<>();

        entdeckteOrte.add("Wald");
        entdeckteOrte.add("Höhle");

        // Wir versuchen, den Wald noch einmal hinzuzufügen...
        boolean hatGeklappt = entdeckteOrte.add("Wald");
        System.out.println("Zweites Einfügen erfolgreich? " + hatGeklappt);
        // false!

        System.out.println("Anzahl der Orte: " + entdeckteOrte.size());
        // Bleibt bei 2!

        // Sets haben keinen Index! get(0) existiert hier nicht.
        // Wir müssen mit einer Schleife darüber iterieren:
        for (String ort : entdeckteOrte) {
            System.out.println("Entdeckt: " + ort);
        }
    }
```

```
}
```

## Arbeiten mit Wörterbüchern (Map)

Eine Map ist wie ein Schließfach. Der Schlüssel (Key) identifiziert das Schließfach eindeutig (darf also nur einmal existieren). Der Inhalt (Value) ist das, was drinliegt. Eine Map hat daher **zwei** Typparameter <K, V>.

```
import java.util.Map;
import java.util.HashMap;

public class MapTest {
    public static void main(String[] args) {

        // Key: String (Name des Spielers) | Value: Integer (Level des
        // Spielers)
        Map<String, Integer> spielerDaten = new HashMap<>();

        // Hineinlegen mit put() anstatt add()
        spielerDaten.put("Artus", 50);
        spielerDaten.put("Merlin", 99);

        // Herausholen über den Schlüssel (blitzschnell, ohne Schleife!)
        Integer level = spielerDaten.get("Merlin");
        System.out.println("Merlins Level ist: " + level);

        // Wenn wir denselben Schlüssel noch einmal verwenden, ÜBERSCHREIBEN
        // wir den Wert!
        spielerDaten.put("Artus", 51); // Artus ist ein Level aufgestiegen

        // Über eine Map iterieren:
        for (Map.Entry<String, Integer> eintrag : spielerDaten.entrySet()) {
            System.out.println("Spieler " + eintrag.getKey() + " (Lvl " +
                               eintrag.getValue() + ")");
        }
    }
}
```

---

Sie sind nun am Ziel der objektorientierten Datenverwaltung angekommen. Mit List, Set und Map haben Sie die Werkzeuge, um gigantische, flexible und typsichere Systeme zu bauen. Diese drei Interfaces werden Sie in jedem einzelnen Java-Projekt Ihrer Karriere begleiten.

Doch wie iterieren wir eigentlich über diese riesigen Listen? Bisher nutzen wir die einfache for-Schleife. Was aber, wenn wir eine Liste mit einer Million Monstern haben und nur die heraussuchen wollen, die rote Augen haben, über 50 Lebenspunkte besitzen und diese dann

nach ihrem Namen sortieren wollen?

Wenn wir das mit klassischen for-Schleifen und if-Abfragen programmieren (imperativ), schreiben wir Seiten voller unleserlichem Code.

Die Lösung für dieses Problem eröffnet ein völlig neues Programmierparadigma, das den Übergang zu unserem dritten Buch ("Funktionale Programmierung") markiert.

Im nächsten Kapitel werfen wir einen exklusiven Blick in die Zukunft: Wir lernen **Lambdas und die Stream-API** kennen.

# 18 Ausblick: Lambdas und Stream API

Im vorherigen Kapitel haben wir mit der Collection-API (List, Set, Map) mächtige Datenstrukturen kennengelernt, um tausende von Objekten in unserem Arbeitsspeicher zu verwalten. Wir haben eine perfekte objektorientierte Architektur für unser Spiel gebaut.

Doch wenn wir ehrlich sind, verarbeiten wir die Daten in diesen modernen Strukturen immer noch mit den altbackenen Werkzeugen eines imperativen Handwerkers. In diesem Kapitel werfen wir einen exklusiven Blick in die Zukunft der Java-Entwicklung. Wir verlassen die Welt, in der wir dem Computer *wie* ein Kleinkind erklären, *wie* er jeden einzelnen Schritt tun soll, und lernen zu sagen, *was* wir eigentlich wollen.

## 18.1 Motivation: Der Weg zur funktionalen Verarbeitung

Stellen Sie sich vor, wir haben eine riesige List<Monster> mit tausenden Feinden. Unsere Aufgabe als Programmierer: *"Finde alle Monster, die mehr als 50 Lebenspunkte haben, extrahiere ihre Namen und gib uns eine Liste dieser Namen, alphabetisch sortiert."*

In der imperativen Welt (Buch 1) und mit unserem bisherigen Wissen würden wir das so programmieren:

```
// Der alte, imperative "Wie"-Weg
List<String> starkeMonsterNamen = new ArrayList<>();

// Wir diktieren jeden einzelnen Schritt:
for (Monster m : alleMonster) {
    if (m.getLebenspunkte() > 50) {
        starkeMonsterNamen.add(m.getName());
    }
}

// Wir rufen eine Hilfsklasse auf, um die neue Liste zu sortieren
Collections.sort(starkeMonsterNamen);
```

Dieser Code funktioniert. Aber er ist geschwätzig, fehleranfällig und vor allem: Er ist starr. Er zwingt den Computer, die Liste exakt nacheinander abzuarbeiten. Wenn wir eine Million Monster hätten, könnten wir diesen Code nicht einfach auf unsere 8 Prozessorkerne aufteilen.

Die moderne Lösung für dieses Problem ist die **deklarative (funktionale) Programmierung**. Anstatt Schleifen zu schreiben, bauen wir eine **Datenfabrik** (Pipeline), durch die unsere Objekte wie auf einem Fließband hindurchströmen.

## 18.2 Lambdas (Kompakter Code)

Um Fließbänder zu bauen, müssen wir den Maschinen an den einzelnen Stationen Anweisungen

geben. In Kapitel 13 haben wir gelernt, wie wir mit **Anonymen Klassen** kleine Einweg-Objekte erschaffen können, die ein Interface (einen Vertrag) erfüllen.

Erinnern Sie sich an **Funktionale Interfaces** aus Kapitel 10? Das sind Interfaces, die exakt *eine* abstrakte Methode besitzen.

Die Entwickler von Java haben erkannt, dass das Schreiben einer kompletten anonymen Klasse für eine einzige Methode viel zu viel Boilerplate-Code ist. Daher haben sie die **Lambda-Ausdrücke** erfunden. Ein Lambda ist im Grunde eine anonyme Funktion – eine Methode ohne Namen, die wir direkt in eine Variable packen oder an eine andere Methode übergeben können.

Der Pfeiloperator -> trennt dabei die Parameter (links) von der eigentlichen Logik (rechts).

```
// 1. Das Funktionale Interface (Vertrag)
@FunctionalInterface
public interface MonsterFilter {
    boolean pruefen(Monster m);
}

public class LambdaTest {
    public static void main(String[] args) {

        // Der alte Weg (Anonyme Klasse):
        MonsterFilter alterFilter = new MonsterFilter() {
            @Override
            public boolean pruefen(Monster m) {
                return m.getLebenspunkte() > 50;
            }
        };

        // Der moderne Weg (Lambda-Ausdruck):
        // Der Compiler weiß, dass 'm' ein Monster sein muss, weil das
        // Interface es diktiert!
        MonsterFilter neuerFilter = m -> m.getLebenspunkte() > 50;

        // Nutzung:
        Monster ork = new Monster("Ork", 80);
        System.out.println("Besteht den Test? " + neuerFilter.pruefen(ork));
        // true
    }
}
```

Mit Lambdas können wir Verhalten (Logik) so einfach herumreichen wie normale Daten!

## 18.3 Streams und ihre Konzepte

Mit den Lambdas als Werkzeug können wir nun unser Fließband bauen. In Java nennt man dieses

Fließband einen **Stream**.

Ein Stream ist keine Datenstruktur. Er speichert keine Daten. Er ist eine reine *Verarbeitungskette*, an die wir eine Datenquelle (wie unsere List) anschließen.

Schauen wir uns an, wie wir das Monster-Problem aus Abschnitt 18.1 mit der Stream-API lösen:

```
// Der moderne, funktionale "Was"-Weg
List<String> starkeMonsterNamen = alleMonster.stream()
                                  // 1. Fließband starten
    .filter(m -> m.getLebenspunkte() > 50)  // 2. Filtern
    .map(m -> m.getName())                  // 3. Namen extrahieren
    .sorted()                               // 4. Sortieren
    .toList();                              // 5. In neue Liste packen
```

Das ist absolute Poesie in Code gegossen! Dieser Code liest sich wie ein englischer Satz. Wir sagen nur noch, *was* passieren soll, nicht mehr *wie* die for-Schleife hochzählen muss. Das ist die Architektur der Zukunft.

## 18.4 Unterschiede zu Collections

Warum haben die Java-Entwickler die Streams erfunden, anstatt diese Methoden direkt in die List- oder Set-Klassen einzubauen?

Als Architekt müssen Sie den fundamentalen Unterschied zwischen einer Collection und einem Stream verstehen:

1. **Speicher vs. Fluss:** Eine Collection (z.B. ArrayList) hält alle Daten physikalisch im Arbeitsspeicher. Ein Stream hält nichts. Er zieht die Daten nur Element für Element aus der Quelle, verarbeitet sie und leitet sie sofort weiter. (Metapher: Collection = DVD im Regal; Stream = Netflix-Video im Fluss).
2. **Unveränderlichkeit (Keine Seiteneffekte):** Ein Stream verändert **niemals** die Datenquelle! Wenn wir .filter(...) aufrufen, werden die schwachen Monster nicht aus der alleMonster-Liste gelöscht. Der Stream generiert lediglich einen neuen Fluss, in dem diese Monster nicht mehr vorkommen.
3. **Lazy Evaluation (Faule Auswertung):** Das ist das genialste Feature. Wenn Sie .filter() und .map() aufrufen, passiert im Computer *absolut gar nichts*. Der Stream merkt sich nur den Bauplan des Fließbandes. Erst in der allerletzten Millisekunde, wenn ganz unten die Terminal-Operation .toList() aufgerufen wird, wirft Java den Motor an und zieht die Daten durch die Pipeline! Das spart enorm viel Rechenzeit.

## 18.5 Wichtige Interfaces und Klassen der Stream-API

Um Streams meisterhaft bedienen zu können, liefert das JDK (im Paket java.util.function) verschiedene funktionale Standard-Interfaces mit, die exakt auf bestimmte Stationen unseres

Fließbandes passen. Hier sind die wichtigsten Operationen:

## Intermediäre Operationen (Zwischenstationen)

Diese Methoden bauen das Fließband weiter um und geben immer wieder einen neuen Stream zurück. Da sie "lazy" sind, starten sie den Motor noch nicht.

- **filter(Predicate<T>)**: Lässt nur Objekte durch, bei denen das Lambda true zurückgibt. (Das Interface Predicate ist ein Tester, der ein Objekt nimmt und einen boolean liefert).
- **map(Function<T, R>)**: Transformiert ein Objekt in etwas völlig anderes. Aus einem Monster (Typ T) wird z.B. ein String (Typ R). (Das Interface Function nimmt einen Typ und liefert einen anderen Typ).
- **limit(long maxSize)**: Schneidet den Stream ab. Lässt maximal die ersten X Elemente passieren.
- **distinct()**: Filtert alle Duplikate heraus (nutzt intern .equals()).

## Terminale Operationen (Endstationen)

Diese Methoden starten den Motor des Fließbandes. Danach ist der Stream "verbraucht" und darf nicht noch einmal verwendet werden.

- **toList()** *(Modern ab Java 16)*: Sammelt alles auf dem Fließband auf und wirft es in eine brandneue, unveränderliche List.
- **forEach(Consumer<T>)**: Nimmt jedes Element am Ende des Fließbandes und führt eine Aktion damit aus (z.B. auf die Konsole drucken). Liefert kein Ergebnis zurück. (Interface Consumer).
- **count()**: Zählt, wie viele Elemente nach allen Filtern am Ende noch übrig geblieben sind.
- **findFirst()**: Liefert das allererste Element des Streams, stoppt danach sofort die gesamte Fließband-Maschine (extrem effizient!).

Lassen Sie uns zum Abschluss unsere neu erlernten OOP-Architekturbausteine (Records) mit Streams kombinieren:

```
// Ein moderner Record
public record Held(String name, int level, boolean istPremium) {}

public class StreamMagie {
    public static void main(String[] args) {
        List<Held> spieler = List.of(
            new Held("Artus", 10, false),
            new Held("Merlin", 99, true),
            new Held("Lancelot", 50, true),
            new Held("Gawain", 5, false)
        );

        // Die Pipeline: Wie viele Premium-Spieler haben ein Level über 20?
```

```
        long anzahl = spieler.stream()
            .filter(h -> h.istPremium())    // Nur Premium durchlassen
            .filter(h -> h.level() > 20)    // Nur Level > 20 durchlassen
            .count();                       // Zählen! Motor startet!

        System.out.println("Anzahl der Elite-Premium-Spieler: " + anzahl);
    }
}
```

---

Sie haben es gesehen: Wenn die objektorientierte Architektur sauber aufgesetzt ist (saubere Kapselung, unveränderliche Records, klare Interfaces), lässt sich der Programmcode der Zukunft in einer Eleganz schreiben, die vor wenigen Jahren in Java noch undenkbar war.

Mit diesem Wissen über die Stream-API sind Sie nicht nur ein exzellenter Software-Architekt geworden, sondern Sie sind nun perfekt auf den Folgeband "Der Fluss der Daten" vorbereitet. Dort werden wir lernen, wie wir diese Pipelines so umbauen, dass sie völlig automatisch auf Dutzenden von Prozessorkernen gleichzeitig laufen (die perfekte Überleitung zu Band 3).

Doch bevor wir dieses Buch endgültig zuklappen, müssen Sie beweisen, dass Sie vom Handwerker wirklich zum Architekten herangereift sind. Im nächsten und letzten Teil dieses Buches (Teil VII) fügen wir alles zusammen.

Wir bauen ein voll funktionsfähiges, objektorientiertes **Praxisprojekt**.

# Teil VII: Die Meisterprüfung

# 19 Praxisprojekt – Der objektorientierte Dungeon

Willkommen am Ziel Ihrer Reise. In den vergangenen 18 Kapiteln haben Sie Ihren Werkzeugkasten gefüllt. Sie haben gelernt, wie man Baupläne zeichnet (Klassen), Geheimnisse bewahrt (Kapselung), Verträge schließt (Interfaces), Familien gründet (Vererbung) und Laufzeitkatastrophen verhindert (Exceptions).

Jetzt ist es an der Zeit, zu beweisen, dass Sie kein einfacher Handwerker mehr sind, der imperative Codezeilen aneinanderreiht. Sie sind ein Software-Architekt. Wir werden nun das Dungeon-Spiel aus dem ersten Buch, das dort noch aus chaotischen Arrays und statischen Funktionen bestand, von Grund auf neu entwerfen – als sauberes, objektorientiertes Meisterwerk.

## 19.1 Architektur und objektorientiertes Design

Bevor ein Architekt den ersten Stein legt, zeichnet er einen Plan. Was brauchen wir für unser Spiel?

1. **Fundament (Daten):** Wir brauchen Koordinaten auf einem Spielfeld und Himmelsrichtungen für die Bewegung.
2. **Verträge:** Objekte in unserer Welt müssen mit dem Helden interagieren können. Egal ob es ein Feind oder ein Trank ist – die Game-Engine will sie alle gleich behandeln.
3. **Klassenhierarchie:** Wir haben leblose Objekte (Tränke) und Lebewesen (Monster, Held). Alle haben eine Position und einen Namen.
4. **Verwaltung:** Wir brauchen ein dynamisches Spielfeld, das beliebig viele Objekte speichern kann (Collections).

## 19.2 Phase 1: Das Fundament (Records und Enums)

Wir beginnen mit den kleinsten, unveränderlichen Bausteinen unseres Systems. Für Koordinaten nutzen wir unser Wissen aus Kapitel 15 und erschaffen einen record. Für die Bewegungsrichtungen nutzen wir ein enum (Kapitel 14).

```
// Datei: Position.java
// Ein unveränderlicher Datencontainer für X- und Y-Koordinaten
public record Position(int x, int y) {

    // Eine Hilfsmethode, um eine neue Position basierend auf einer Richtung
    // zu berechnen
    public Position bewege(Richtung r) {
        return switch (r) {
            case NORD -> new Position(this.x, this.y - 1);
            case SUED -> new Position(this.x, this.y + 1);
            case OST  -> new Position(this.x + 1, this.y);
            case WEST -> new Position(this.x - 1, this.y);
```

```
        }; // Modernes Switch aus Java 14+!
    }
}

// Datei: Richtung.java
// Typsichere Aufzählung für die erlaubten Bewegungen
public enum Richtung {
    NORD, OST, SUED, WEST;
}
```

## 19.3 Phase 2: Verträge und Hierarchien (Interfaces & Abstrakte Klassen)

Nun definieren wir, wie die Welt kommuniziert. Wir erschaffen ein Interface Interagierbar. Alles, was der Held auf dem Spielfeld berühren kann, muss diesen Vertrag unterschreiben.

```
// Datei: Interagierbar.java
public interface Interagierbar {
    // Was passiert, wenn der Held auf das Feld dieses Objekts tritt?
    void interagiere(Held held);
}
```

Jetzt bauen wir unsere Vererbungshierarchie. Ganz oben steht das abstrakte SpielObjekt. Es existiert nur, um doppelten Code für Name und Position zu vermeiden.

```
// Datei: SpielObjekt.java
public abstract class SpielObjekt {
    protected String name;
    protected Position position;

    public SpielObjekt(String name, Position startPosition) {
        this.name = name;
        this.position = startPosition;
    }

    public Position getPosition() { return this.position; }
    public String getName() { return this.name; }
}
```

Von diesem Basis-Objekt leiten wir nun die Lebewesen ab. Auch Lebewesen ist abstract, denn es gibt keine formlosen Lebewesen, nur konkrete Helden oder Monster!

```
// Datei: Lebewesen.java
public abstract class Lebewesen extends SpielObjekt {
    protected int lebenspunkte;
    protected int angriffskraft;
```

```
    public Lebewesen(String name, Position pos, int hp, int angriff) {
        super(name, pos); // Aufruf des Konstruktors von SpielObjekt
        this.lebenspunkte = hp;
        this.angriffskraft = angriff;
    }

    public void schadenNehmen(int schaden) {
        this.lebenspunkte -= schaden;
        if (this.lebenspunkte < 0) this.lebenspunkte = 0;
    }

    public boolean istTot() {
        return this.lebenspunkte <= 0;
    }
}
```

## 19.4 Phase 3: Die konkreten Akteure (Polymorphie in Aktion)

Jetzt füllen wir die Welt mit Leben. Wir erschaffen den Held und den Ork. Beachten Sie, wie der Ork durch das Interface Interagierbar sein Verhalten für die Spielwelt definiert.

```
// Datei: Ork.java
// Erbt von Lebewesen und erfüllt den Vertrag zum Interagieren!
public class Ork extends Lebewesen implements Interagierbar {

    public Ork(Position pos) {
        // Ein Ork heißt immer "Böser Ork" und hat 50 HP / 10 Angriff
        super("Böser Ork", pos, 50, 10);
    }

    @Override
    public void interagiere(Held held) {
        System.out.println("Der Ork brüllt auf und greift an!");
        held.schadenNehmen(this.angriffskraft);
        System.out.println("Der Held verliert " + this.angriffskraft +
                           " HP.");
    }
}
```

Zusätzlich erschaffen wir ein Item: Den Heiltrank. Er ist kein Lebewesen, sondern erbt direkt von SpielObjekt, ist aber ebenfalls Interagierbar!

```
// Datei: Heiltrank.java
public class Heiltrank extends SpielObjekt implements Interagierbar {
    private int heilkraft;

    public Heiltrank(Position pos) {
        super("Großer Heiltrank", pos);
```

```
        this.heilkraft = 20;
    }

    @Override
    public void interagiere(Held held) {
        System.out.println("Du findest einen Heiltrank und trinkst ihn " +
                           "sofort.");
        held.schadenNehmen(-this.heilkraft); // Negativer Schaden = Heilung
        System.out.println("Du wurdest um " + this.heilkraft +
                           " HP geheilt.");
    }
}
```

Hier sehen Sie die absolute Macht der Objektorientierung: Ein Ork und ein Heiltrank haben völlig unterschiedliche Stammbäume. Aber beide besitzen die Methode interagiere(). Das ist alles, was unsere Game-Engine wissen muss!

```
// Datei: Held.java
public class Held extends Lebewesen {

    public Held(Position startPos) {
        super("Artus", startPos, 100, 25);
    }

    // Der Held kann seine Position verändern
    public void gehe(Richtung r) {
        this.position = this.position.bewege(r);
        System.out.println(this.name + " geht nach " + r.name());
    }

    public void statusAnzeigen() {
        System.out.println("--- HELD STATUS: " + this.lebenspunkte +
                           " HP ---");
    }
}
```

## 19.5 Phase 4: Fehlerbehandlung (Eigene Exceptions)

Was passiert, wenn der Spieler versucht, den Helden aus der Karte (z.B. ins Negative) zu bewegen? Wir werfen eine eigene, kontrollierte Checked Exception.

```
// Datei: WandBeruehrtException.java
public class WandBeruehrtException extends Exception {
    public WandBeruehrtException(String message) {
        super(message);
    }
}
```

## 19.6 Phase 5: Die Game-Engine (Collections und die Main-Loop)

Jetzt verbinden wir alle Bausteine in unserem Hauptprogramm. Wir nutzen eine List für das Spielfeld und demonstrieren das dynamische Binden.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class DungeonEngine {

    private Held spieler;
    private List<SpielObjekt> spielwelt;

    // Wir definieren die Grenzen unseres Dungeons
    private final int MAX_X = 5;
    private final int MAX_Y = 5;

    public DungeonEngine() {
        spieler = new Held(new Position(0, 0));
        spielwelt = new ArrayList<>();

        // Wir platzieren die Objekte auf unserer 5x5 Karte
        spielwelt.add(new Ork(new Position(2, 1)));
        spielwelt.add(new Heiltrank(new Position(4, 0)));
        spielwelt.add(new Ork(new Position(3, 3)));
    }

    public void spielen() {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Willkommen im OOP-Dungeon!");

        while (!spieler.istTot() && !spielwelt.isEmpty()) {
            // 1. Visuelles Feedback für den Spieler
            karteZeichnen();
            spieler.statusAnzeigen();

            System.out.print("Wohin gehen? (n/o/s/w): ");
            String eingabe = scanner.nextLine();

            try {
                // 2. Zug ausführen
                Richtung r = parserEingabe(eingabe);
                spieler.gehe(r);

                // 3. Sicherheitscheck (Ist er aus der 5x5 Karte gelaufen?)
                Position p = spieler.getPosition();
                if (p.x() < 0 || p.x() >= MAX_X || p.y() < 0 ||
                    p.y() >= MAX_Y) {
```

```
                    throw new WandBeruehrtException(
                       "Autsch! Du bist gegen die Dungeon-Mauer gelaufen.");
                }

                // 4. Spielfeld prüfen (Kollisionsabfrage)
                kollisionPruefen();

            } catch (WandBeruehrtException e) {
                System.out.println("FEHLER: " + e.getMessage());
            } catch (IllegalArgumentException e) {
                System.out.println("Bitte nur n, o, s oder w eingeben!");
            }
        }

        if (spieler.istTot()) {
            System.out.println("GAME OVER! Du bist gestorben.");
        } else {
            karteZeichnen(); // Ein letzter Blick auf das leere Dungeon
            System.out.println("SIEG! Du hast das Dungeon gesäubert.");
        }
    }

    // --- NEU: Die Visualisierung der Welt ---
    private void karteZeichnen() {
        System.out.println("\n=== DUNGEON KARTE ===");

        for (int y = 0; y < MAX_Y; y++) {
            for (int x = 0; x < MAX_X; x++) {

                Position aktuellesFeld = new Position(x, y);

                // Steht der Held auf diesem Feld? (Nutzt die automatische
                // equals-Methode des Records!)
                if (spieler.getPosition().equals(aktuellesFeld)) {
                    System.out.print("[H] ");
                    continue; // Gehe direkt zum nächsten Feld
                }

                // Liegt ein Objekt auf diesem Feld?
                SpielObjekt gefunden = null;
                for (SpielObjekt obj : spielwelt) {
                    if (obj.getPosition().equals(aktuellesFeld)) {
                        gefunden = obj;
                        break;
                    }
                }

                // Zeichnen basierend auf dem Typ des gefundenen Objekts
                if (gefunden instanceof Ork) {
                    System.out.print("[O] ");
```

```
                } else if (gefunden instanceof Heiltrank) {
                    System.out.print("[T] ");
                } else {
                    System.out.print("[.] "); // Leeres Feld
                }
            }
            System.out.println(); // Zeilenumbruch nach jeder X-Reihe
        }
        System.out.println("======================\n");
    }

    // Hilfsmethode zur Umwandlung von Text in Enums
    private Richtung parserEingabe(String input) {
        return switch (input.toLowerCase()) {
            case "n" -> Richtung.NORD;
            case "o" -> Richtung.OST;
            case "s" -> Richtung.SUED;
            case "w" -> Richtung.WEST;
            default -> throw new IllegalArgumentException(
                                    "Ungültige Richtung");
        };
    }

    // Die architektonische Meisterleistung: Dynamisches Binden
    private void kollisionPruefen() {
        SpielObjekt getroffenesObjekt = null;

        for (SpielObjekt obj : spielwelt) {
            if (obj.getPosition().equals(spieler.getPosition())) {
                getroffenesObjekt = obj;
                break;
            }
        }

        if (getroffenesObjekt != null) {
            // Pattern Matching for instanceof
            if (getroffenesObjekt instanceof Interagierbar ziel) {
                ziel.interagiere(spieler);
                spielwelt.remove(getroffenesObjekt); // Objekt entfernen
            }
        }
    }

    public static void main(String[] args) {
        DungeonEngine spiel = new DungeonEngine();
        spiel.spielen();
    }
}
```

## 19.7 Fazit des Praxisprojekts

Starten Sie dieses Programm (kopieren Sie die Klassen idealerweise in Ihre IDE). Spielen Sie es.

Sehen Sie sich die Methode kollisionPruefen() in der Engine ganz genau an. Dort gibt es kein einziges if (objekt == Ork) ... else if (objekt == Heiltrank) .... Die Engine vertraut blind auf den Vertrag Interagierbar.

Das ist das ultimative Ziel der Objektorientierung (das Open/Closed Principle aus Kapitel 8). Wenn Sie morgen beschließen, eine Schatzkiste, eine Stachelfalle oder einen Händler in das Spiel einzubauen, müssen Sie die Klasse DungeonEngine nicht mehr anfassen! Sie erschaffen einfach eine neue Klasse Falle implements Interagierbar, werfen sie in die spielwelt-Liste, und die Engine wird sie völlig automatisch und fehlerfrei verarbeiten.

Sie haben komplexe Systeme in kleine, autarke, intelligente schwarze Boxen unterteilt. Sie haben die imperative Welt verlassen.

## 19.8 Epilog

Liebe Leserinnen und Leser,

Sie haben das Buch erfolgreich beendet. Sie haben den Paradigmenwechsel vom sequenziellen Abarbeiten von Codezeilen hin zum Entwerfen lebendiger, kommunizierender Architekturen gemeistert. Sie verstehen nun, was Kapselung, Vererbung und Polymorphie bedeuten und wie Sie typsichere Generics und Streams nutzen.

Doch die Reise eines Software-Entwicklers endet nie. Jetzt, da Sie wissen, wie man stabile Objekte baut, sind Sie bereit für die nächsten Herausforderungen. Was passiert, wenn wir Daten nicht mehr verändern wollen, sondern sie wie Wasser durch funktionale Fließbänder schicken? Und wie verhindern wir, dass unsere schöne Architektur zusammenbricht, wenn zehn Prozesse gleichzeitig auf denselben Ork einschlagen?

All dies erfahren Sie in den nächsten Büchern dieser Reihe: *"Der Fluss der Daten – Funktionale Programmierung"* und *"Die Choreografie der Threads – Parallele Programmierung"*.

Sie haben das Handwerk gelernt. Sie sind zum Architekten herangereift. Bauen Sie nun Großes!