DER FLUSS DER DATEN

Funktionale Programmierung in Java meistern
(Band 3)

Dietrich Boles

Impressum

Autor: Dietrich Boles

Kontakt: dietrich@boles.de

Vorwort

Liebe Leserinnen und Leser,

willkommen zu einem der faszinierendsten Paradigmenwechsel Ihrer Programmierlaufbahn!

Wenn Sie dieses Buch aufschlagen, beherrschen Sie die Grundlagen der Java-Programmierung bereits souverän. Sie wissen, wie man Klassen entwirft, Objekte instanziiert und Datenstrukturen mit for-Schleifen und if-Abfragen Schritt für Schritt verarbeitet. Bisher folgte Ihr Code dabei einem vertrauten, imperativen Paradigma: Sie haben dem Computer haargenau diktiert, *wie* er ein Problem lösen soll, und dabei kontinuierlich den Zustand Ihrer Variablen und Objekte verändert. Die Welt bestand aus Befehlen und Mutationen.

Mit dem Aufschlagen dieses Buches lade ich Sie ein, diese gewohnte, aber zunehmend fehleranfällige Welt hinter sich zu lassen.

Die Realität moderner, hochkomplexer Softwaresysteme hat den klassischen, zustandsbehafteten Code längst an seine Grenzen gebracht. Je größer eine objektorientierte Anwendung wird, desto schwieriger wird es, sie zu bändigen. Objekte verändern heimlich den Zustand anderer Objekte, Methoden haben versteckte Seiteneffekte, und sobald Sie versuchen, Ihr Programm auf mehrere Threads aufzuteilen, bricht das Chaos aus. Wer sich bereits mit der nebenläufigen Programmierung beschäftigt hat, der kennt die schmerzhafte Wahrheit: Geteilter, veränderlicher Zustand ist der absolute Feind der Parallelisierung.

Die funktionale Programmierung bietet für dieses Dilemma die eleganteste und nachhaltigste Lösung.

Wir werden in diesem Buch lernen, nicht mehr in Zustandsänderungen zu denken, sondern in **Datenflüssen**. Wir verlassen die traditionelle Handwerkswerkstatt, in der ein Bauteil ständig angefasst und verbogen wird, und errichten stattdessen eine hochmoderne, automatisierte Datenfabrik. Auf unseren Fließbändern werden Daten nicht mehr verändert, sondern durch reine Funktionen in völlig neue, makellose Endprodukte transformiert.

Dabei müssen Sie keine neue, akademische Sprache wie Haskell oder Clojure erlernen. Java hat sich in den letzten Jahren rasant weiterentwickelt und mächtige funktionale Konzepte tief in seine DNA integriert.

Dieses Buch nimmt Sie mit auf eine systematische Reise durch diese neuen Fähigkeiten:

- Im **ersten Teil** lernen Sie, wie Sie Verhalten (Code) endlich als Datenpakete verpacken und durch Ihr System reichen können. Lambdas und funktionale Interfaces werden Ihre alten, starren Klassenstrukturen aufbrechen.
- Im **zweiten Teil** starten wir die Maschinen. Sie werden sehen, wie die Stream-API klassische Schleifen durch flüssige, deklarative und hochperformante Pipelines ersetzt.
- Im **dritten Teil** härten wir unsere Architektur. Wir besiegen die gefürchtete NullPointerException mit der Optional-Monade, schließen Daten-Konflikte durch unveränderliche Java Records aus und lernen, wie man Exceptions funktional zähmt.
- Im **vierten Teil** wird es schließlich ernst. Wir betreten einen historisch gewachsenen, imperativen E-Commerce-Shop und refaktorieren seinen fehleranfälligen Spaghetti-Code in sieben Iterationen zu einer meisterhaften, funktionalen Architektur.

Ein Ratschlag für Ihre Lektüre: Funktionale Programmierung lernt man nicht durch bloßes Lesen, sondern durch das Aufbrechen alter Denkmuster. Öffnen Sie Ihre Entwicklungsumgebung. Tippen Sie die Beispiele ab. Nehmen Sie sich eine alte, aufgedunsene Klasse aus einem Ihrer eigenen Projekte und versuchen Sie, sie auf einen einzigen, eleganten Stream zu reduzieren.

Sie werden feststellen: Funktionale Programmierung macht Ihren Code nicht nur sicherer und robuster. Sie führt zu einer Ausdruckskraft und Eleganz, die das Programmieren wieder zu dem macht, was es sein sollte – einer wahren Freude.

Lassen Sie uns die Maschinen starten!

Dietrich Boles, Oldenburg, März 2026

Inhaltsverzeichnis

1 Einleitung

Der Übergang von der imperativen und objektorientierten Programmierung zur funktionalen Programmierung gleicht dem Wechsel von einer traditionellen Handwerkswerkstatt zu einer hochmodernen, automatisierten Datenfabrik.

In der klassischen Werkstatt (der objektorientierten Welt) nimmt der Handwerker (Ihre Methode) ein Werkstück (Ihr Objekt), spannt es ein und bearbeitet es Schritt für Schritt. Er sägt, er hämmert, er verändert den Zustand des Werkstücks permanent. Wenn zwei Handwerker gleichzeitig an demselben Werkstück arbeiten wollen, entsteht Chaos.

In einer funktionalen Datenfabrik hingegen rollen die Rohmaterialien auf einem Fließband herein. Anstatt das Originalmaterial zu verändern, nimmt jede Station das Material, leitet daraus ein völlig neues, veredeltes Produkt ab und reicht dieses an die nächste Station weiter. Das Original bleibt absolut unberührt. Das Fließband läuft stetig, vorhersehbar und lässt sich problemlos in hundert parallele Bänder aufteilen, ohne dass jemals ein Konflikt entsteht.

Genau den Aufbau und die Steuerung dieses Fließbands werden Sie in diesem Buch erlernen.

1.1 Motivation für funktionale Programmierung

Warum sollten wir unsere vertraute, imperative Welt verlassen, in der wir mit for-Schleifen und if-Abfragen doch bisher jedes Problem lösen konnten?

Der Grund ist die unaufhaltsam wachsende Komplexität moderner Software. Wenn Anwendungen größer werden, wird der **veränderliche Zustand (Mutable State)** zum größten Feind des Entwicklers. Ein Objekt ändert heimlich den Wert eines anderen, Methoden haben versteckte Seiteneffekte und Bugs lassen sich kaum noch reproduzieren, weil sie vom exakten zeitlichen Ablauf des Programms abhängen.

Die funktionale Programmierung entzieht diesem Chaos den Nährboden. Sie zwingt uns, in reinen Transformationen und unveränderlichen Datenstrukturen (Immutability) zu denken.

Ein zweiter, ebenso wichtiger Grund ist die pure Lesbarkeit des Codes. Imperativer Code diktiert der Maschine das *Wie* (Wie iteriere ich über eine Liste? Wie verwalte ich den Index?). Funktionale Programmierung erlaubt es uns, das *Was* zu beschreiben. Code liest sich plötzlich nicht mehr wie eine mechanische Bauanleitung, sondern wie eine

natürliche Fehlerbeschreibung oder eine klare Geschäftsregel.

Letztlich bereitet die funktionale Denkweise den perfekten Boden für moderne, skalierbare Architekturen. Wo es keine geteilten, veränderbaren Zustände gibt, gibt es auch keine Race Conditions, wenn wir unser Programm parallelisieren.

1.2 Voraussetzungen zum Lesen dieses Buches

Wenn Sie dieses Buch aufschlagen, sollten Sie die Grundlagen der Java-Programmierung bereits gemeistert haben. Sie sollten wissen, wie man Klassen entwirft, Methoden schreibt und grundlegende Algorithmen formuliert.

Darüber hinaus sind folgende Konzepte hilfreich:

- **Objektorientierung:** Ein sicherer Umgang mit Konzepten wie Vererbung, Interfaces und Polymorphie.
- **Basiswissen der Java-API:** Sie sollten das Collections-Framework (Listen, Maps, Sets) kennen und mit dem Konzept der Generics (List<T>) umgehen können.

Sie benötigen hingegen absolut kein Vorwissen im Bereich der rein funktionalen Sprachen wie Haskell oder Lisp. Wir erarbeiten uns die theoretischen Konzepte pragmatisch, praxisnah und direkt in der Syntax von Java.

1.3 Aufbau des Buches

Um Sie sicher durch diesen Paradigmenwechsel zu führen, ist dieses Buch in vier große Teile gegliedert, die logisch aufeinander aufbauen:

- **Teil I: Die funktionalen Bausteine in Java (Die Werkzeuge der Fabrik):** Im ersten Teil lernen Sie, wie Java Funktionen endlich als Bürger erster Klasse behandelt und wie Sie Verhalten – nicht nur Daten – in Form von Lambdas flexibel durch Ihr System reichen.
- **Teil II: Daten im Fluss – Die Stream-API (Das Fließband starten):** Hier verlassen wir die Welt der manuellen Schleifen. Sie lernen, wie Sie Datenströme elegant filtern, transformieren und aggregieren.
- **Teil III: Praxis, Muster, Perspektiven – Funktionales Design (Die Fabrik absichern):** Wir beseitigen die größten Fehlerquellen der Programmierung (wie die berüchtigte NullPointerException) durch funktionale Entwurfsmuster und bauen unveränderliche, robuste Architekturen.
- **Teil IV: Die Datenfabrik in der Praxis – Das E-Commerce-Projekt:** Im vierten Teil des Buches führen wir alle zuvor behandelten Konzepte in einem großen, iterativen

Praxisbeispiel zusammen: der konsequenten funktionalen Modernisierung eines klassischen Webshops.

1.4 Überblick über die einzelnen Themen des Buches

Damit Sie genau wissen, was Sie auf dieser Reise erwartet, hier ein detaillierter Ausblick auf die Inhalte:

Im **ersten Teil** klären wir zunächst die fundamentalen Unterschiede zwischen deklarativer und imperativer Denkweise. Wir untersuchen das @FunctionalInterface und zeigen, wie Lambda-Ausdrücke und Methodenreferenzen die alten, schwerfälligen anonymen inneren Klassen ablösen. Zudem stellen wir Ihnen den Standard-Werkzeugkasten von Java vor (Predicate, Function, Consumer und Supplier), mit dem Sie fast jedes fachliche Problem modellieren können.

Im **zweiten Teil** widmen wir uns dem Herzstück: der Stream-API. Wir bauen leistungsstarke Pipelines aus einer Quelle, intermediären Operationen (map, filter, flatMap) und terminalen Operationen (reduce, collect). Sie lernen das geniale Konzept der "Lazy Evaluation" kennen und sehen, wie sich diese Pipelines fast ohne Mehraufwand massiv parallelisieren lassen.

Im **dritten Teil** rüsten wir Sie für die Produktion. Wir ersetzen die gefürchtete null-Referenz durch den bewussten Einsatz der Optional-Monade. Wir lernen, wie Java Records uns bei der schnellen Modellierung unveränderlicher Datenstrukturen helfen und wie wir Fehler (Exceptions) funktional behandeln, ohne den Datenfluss gewaltsam zu unterbrechen. Abschließend betrachten wir moderne Architekturmuster wie "Functional Core, Imperative Shell" und werfen einen Blick auf externe funktionale Bibliotheken wie Vavr.

Im **vierten Teil** führen wir diese Konzepte in einem großen Praxisbeispiel zusammen. In sieben iterativen Schritten transformieren wir einen fehleranfälligen, rein imperativen E-Commerce-Shop in eine elegante, funktionale Datenfabrik. Wir beginnen mit dem Ablösen verschachtelter for-Schleifen, sichern den mutierbaren Warenkorb durch Records ab und modellieren komplexe Bestellzustände absolut typsicher mit Pattern Matching und Sealed Classes.

Ein Ratschlag für Ihre Reise: Lesen Sie dieses Buch nicht einfach nur passiv durch. Funktionale Programmierung erfordert ein echtes Umdenken im Kopf. Setzen Sie sich an Ihre Entwicklungsumgebung, tippen Sie die Code-Beispiele ab und versuchen Sie mutig, alte Legacy-Klassen aus Ihren eigenen Projekten in funktionale Pipelines zu

refaktorieren. Nur wer den alten, aufgedunsenen Code einmal selbst auf einen sauberen Einzeiler reduziert hat, versteht den wahren Wert der funktionalen Eleganz.

Lassen Sie uns beginnen!

2 Grundlagen der funktionalen Denkweise

Willkommen in der Datenfabrik! Bevor wir uns in den nächsten Kapiteln die konkreten Werkzeuge und Maschinen ansehen, die Java uns für den Bau unserer Fließbänder zur Verfügung stellt, müssen wir uns mit der Philosophie dieser Fabrik vertraut machen.

Funktionale Programmierung ist keine reine Syntax-Frage. Sie können in Java 25 programmieren und trotzdem Code schreiben, der konzeptionell in den 1990er Jahren feststeckt. Funktionale Programmierung ist in erster Linie ein Paradigmenwechsel in Ihrem Kopf. Es geht darum, wie Sie Datenflüsse betrachten, wie Sie Fehlerquellen isolieren und wie Sie dem Computer Ihre Absichten mitteilen.

2.1 Motivation: Warum das Ganze?

Vielleicht stellen Sie sich die berechtigte Frage: *Warum sollte ich mein bewährtes objektorientiertes Denken aufgeben? Meine for-Schleifen haben bisher immer funktioniert!*

Die Antwort liegt in der wachsenden Komplexität moderner Software und der Hardware, auf der sie läuft. Heutzutage werden Prozessoren nicht mehr signifikant schneller, sondern breiter – sie erhalten mehr Rechenkerne. Um diese Kerne effizient zu nutzen, müssen wir unsere Programme parallelisieren. Und genau hier wird die klassische Objektorientierung, die stark auf das Verändern von Zuständen (Mutable State) setzt, zu einem Minenfeld aus Race Conditions und Deadlocks.

Die funktionale Programmierung bietet einen eleganten Ausweg aus diesem Dilemma. Indem sie veränderliche Zustände verbietet und Seiteneffekte minimiert, macht sie unseren Code von Natur aus "Thread-safe". Gleichzeitig führt sie zu Code, der wesentlich kompakter, ausdrucksstärker und leichter zu testen ist. Wir verlagern den Fokus vom *Management des Kontrollflusses* (Indizes hochzählen, Schleifen abbrechen) auf die *Transformation von Daten*.

2.2 Das Problem: Zustand (State) und Seiteneffekte

Das Fundament der klassischen, imperativen Programmierung ist der veränderliche Zustand. Variablen werden deklariert und ihr Wert ändert sich im Laufe des Programms ständig. Eine Methode fügt einer Liste ein Element hinzu, ein Setter ändert den Namen eines Kunden, eine globale Zählervariable wird inkrementiert.

Das Problem dabei ist: Veränderlicher Zustand ist schwer zu überblicken. Wenn ein

Objekt quer durch Ihr System gereicht wird und jede Methode potenziell dessen Zustand ändern darf, verlieren Sie schnell die Kontrolle.

Das führt uns zu einem der wichtigsten Begriffe der funktionalen Programmierung: dem **Seiteneffekt** (Side Effect). Ein Seiteneffekt tritt auf, wenn eine Funktion oder Methode nicht nur einen Wert zurückgibt, sondern "heimlich" den Zustand außerhalb ihres eigenen Gültigkeitsbereichs (Scopes) verändert.

Betrachten wir ein typisches Beispiel aus der alten Handwerkswerkstatt:

```
public class PreisKalkulator {
    private double gesamtRabatt = 0.0; // Veränderlicher Zustand

    public double berechneEndpreis(List<Artikel> warenkorb) {
        double summe = 0.0;
        for (Artikel artikel : warenkorb) {
            summe += artikel.getPreis();
            if (artikel.isAktionsware()) {
                // SEITENEFFEKT: Wir ändern den Zustand der Klasse!
                this.gesamtRabatt += 5.0;
                // SEITENEFFEKT: Wir manipulieren das übergebene Objekt!
                artikel.setPreis(artikel.getPreis() - 5.0);
            }
        }
        return summe;
    }
}
```

Wenn Sie diese Methode berechneEndpreis aufrufen, erwarten Sie eigentlich nur eine Zahl zurück. Aber unter der Haube passieren Dinge, die das restliche System beeinflussen: Die Artikel im übergebenen Warenkorb wurden manipuliert und eine Instanzvariable wurde verändert. Rufen Sie die Methode ein zweites Mal mit demselben Warenkorb auf, erhalten Sie ein völlig anderes Ergebnis! Wenn nun noch zwei Threads gleichzeitig auf diesen PreisKalkulator zugreifen, ist die Katastrophe programmiert.

In der Datenfabrik gilt daher die eiserne Regel: **Daten werden nicht verändert, sie werden transformiert.** Wenn eine Station am Fließband einen rabattierten Artikel erzeugen soll, nimmt sie den Originalartikel und gibt eine völlig neue, rabattierte Kopie zurück. Das Original bleibt unangetastet.

2.3 Imperativ vs. Deklarativ: "Wie" gegen "Was"

Dieser Paradigmenwechsel äußert sich auch in der Art, wie wir Code schreiben.

- **Imperativ (Der alte Weg):** Sie sagen dem Computer Schritt für Schritt, **WIE** er etwas tun soll. Sie schreiben eine Bauanleitung.
- **Deklarativ (Der funktionale Weg):** Sie beschreiben, **WAS** Sie als Ergebnis haben möchten. Sie geben eine Bestellung auf.

Lassen Sie uns den Unterschied an einem Code-Beispiel festmachen. Wir wollen aus einer Liste von Rechnungen alle extrahieren, die noch unbezahlt sind und einen Wert von über 100 Euro haben.

Der imperative Ansatz:

```
public List<Rechnung> findeOffeneGrossrechnungen(
                          List<Rechnung> alleRechnungen) {
    List<Rechnung> ergebnis = new ArrayList<>(); // Zustand initialisieren
    for (Rechnung r : alleRechnungen) {          // Kontrollfluss definieren
                                                 // (Wie iterieren wir?)
        if (!r.isBezahlt() && r.getBetrag() > 100.0) { // Bedingungen prüfen
            ergebnis.add(r);       // Zustand verändern (Seiteneffekt lokal)
        }
    }
    return ergebnis;
}
```

Dieser Code ist voller "Boilerplate" (Verwaltungs-Code). Wir müssen uns um das Erstellen der Liste kümmern, die Schleife verwalten und Elemente manuell hinzufügen.

Der deklarative Ansatz:

```
public List<Rechnung> findeOffeneGrossrechnungen(List<Rechnung>
alleRechnungen) {
    return alleRechnungen.stream()
            .filter(r -> !r.isBezahlt())
            .filter(r -> r.getBetrag() > 100.0)
            .toList();
}
```

Hier beschreiben wir nur noch das *Was*. Wir etablieren einen Datenstrom (stream), setzen zwei Filter (filter) und sammeln das Ergebnis ein (toList). Der Code liest sich fast wie ein englischer Satz. Wie die Iteration unter der Haube genau abläuft – ob sequenziell oder hochgradig parallel –, interessiert uns auf dieser Abstraktionsebene nicht mehr.

2.4 Die mathematische Basis: Pure Functions und Referenzielle Transparenz

Die funktionale Programmierung hat ihre Wurzeln im sogenannten Lambda-Kalkül, einem mathematischen Konzept aus den 1930er Jahren. In der Mathematik gibt es keinen veränderlichen Zustand. Die Funktion f(x) = x + 1 ändert nicht den Wert von x. Sie nimmt x und liefert ein neues Ergebnis zurück.

Dieses Prinzip übertragen wir auf unsere Softwareentwicklung und nennen es **Pure Function (Reine Funktion)**. Eine Funktion gilt als "pur", wenn sie zwei strikte Bedingungen erfüllt:

1. **Determinismus:** Bei gleicher Eingabe liefert sie immer exakt dieselbe Ausgabe. Sie ist nicht abhängig von globalen Variablen, der Systemuhrzeit oder einer Datenbank.
2. **Keine Seiteneffekte:** Sie verändert keinen Zustand außerhalb ihres eigenen Scopes. Sie schreibt nichts in eine Datenbank, verändert keine übergebenen Objekte und gibt nichts auf der Konsole aus.

Wenn Sie eine Architektur aus reinen Funktionen aufbauen, erreichen Sie den heiligen Gral der funktionalen Programmierung: die **Referenzielle Transparenz (Referential Transparency)**.

Das bedeutet, Sie könnten in Ihrem Code jeden Aufruf einer Funktion direkt durch ihr Resultat ersetzen, ohne dass sich das Verhalten Ihres Programms ändert.

Wenn Sie eine reine Funktion add(2, 3) haben, können Sie diesen Aufruf in Ihrem gesamten Code bedenkenlos durch die Konstante 5 ersetzen. Diese Eigenschaft macht funktionalen Code unfassbar leicht testbar. Sie müssen für Ihre Unit-Tests keine komplexen Datenbank-Mocks oder Spring-Kontexte hochfahren. Sie stecken Daten in die Funktion hinein und prüfen, was am anderen Ende des Fließbands herausfällt.

2.5 Die Brücke zu Java: Rückwärtskompatibilität und funktionale Interfaces

Vielleicht fragen Sie sich nun: *Wie hat Java, das 1995 als rein objektorientierte Sprache das Licht der Welt erblickte, diesen Paradigmenwechsel vollzogen, ohne Milliarden von bestehenden Codezeilen auf der ganzen Welt unbrauchbar zu machen?*

Die Architekten von Java standen bei der Einführung funktionaler Konzepte in Java 8 vor einer gigantischen Herausforderung. Sie mussten Funktionen als Bürger erster Klasse ("First-Class Citizens") einführen – also erlauben, dass Code (Verhalten) wie Daten in

Variablen gespeichert und an Methoden übergeben werden kann. Neue, rein funktionale Sprachen wie Scala erzeugten dafür völlig neue Konstrukte.

Java wählte einen anderen, genialen Weg, um die Abwärtskompatibilität zu wahren: die **Funktionalen Interfaces**.

Anstatt das Typsystem der Sprache komplett umzubauen, definierte Java: Jedes Interface, das *genau eine einzige abstrakte Methode* (Single Abstract Method - SAM) besitzt, kann ab sofort als Träger für eine Funktion dienen.

Das bedeutet, das Verhalten (die Funktion) verkleidet sich auf Ebene der Java-Virtual-Machine (JVM) weiterhin als Objekt, um die alten Regeln zu befolgen. Für uns Entwickler bietet der Compiler jedoch eine stark verkürzte Syntax, die diese Verkleidung unsichtbar macht: die Lambda-Ausdrücke.

Wie genau dieser geniale Trick der Java-Sprachendesigner funktioniert und wie wir ihn nutzen, um Verhalten elegant durch unsere Datenfabrik zu leiten, schauen wir uns im nun folgenden ersten Hauptteil des Buches im Detail an.

Teil I:
Die funktionalen Bausteine in Java

3 Verhalten übergeben – Ein Überblick

Willkommen im ersten Hauptteil unseres Buches. Wir haben in den vorherigen Kapiteln die theoretische Notwendigkeit für reine Funktionen und unveränderliche Daten kennengelernt. Nun betreten wir die Werkhalle unserer Datenfabrik und schauen uns die Maschinen an, die uns Java zur Verfügung stellt.

Das zentrale Problem, das wir lösen müssen, lautet: Wie übergeben wir ein reines *Verhalten* (eine Aktion, eine Regel, eine Berechnung) an eine andere Komponente unseres Programms?

In der objektorientierten Welt übergeben wir Daten in Form von Objekten. Wir übergeben einen Kunden, eine Rechnung oder einen String. Aber was ist, wenn wir eine *Sortierregel* übergeben wollen? Oder eine *Filterbedingung*? In diesem Kapitel vollziehen wir die Evolution von schwerfälligen Objekten hin zu eleganten, reinen Funktionen in drei Ebenen.

3.1 Ebene 1: Die klassischen Wege (Das Königreich der Nomen)

Erinnern Sie sich an unsere Metapher aus der klassischen Handwerkswerkstatt: Im "Königreich der Nomen" darf ein Verb (ein Verhalten) nicht ohne ein Nomen (ein Objekt) existieren. Jede Aktion muss zwingend in einer Klasse eingesperrt sein.

Lassen Sie uns das an einem konkreten Code-Beispiel durchspielen. Wir haben eine Liste von Namen, die wir in unserer Fabrik sortieren möchten – und zwar nicht alphabetisch, sondern nach der Länge der Namen (der kürzeste Name zuerst).

Die Methode Collections.sort() oder List.sort() benötigt dafür eine Sortierregel. Vor Java 8 sah die Übergabe dieser Regel so aus:

```
List<String> namen = Arrays.asList("Maximilian", "Jan", "Anna",
                                   "Christopher");

// Ebene 1: Die anonyme innere Klasse
namen.sort(new Comparator<String>() {
    @Override
    public int compare(String name1, String name2) {
        return Integer.compare(name1.length(), name2.length());
    }
});

System.out.println(namen); // Ausgabe: [Jan, Anna, Maximilian, Christopher]
```

Analysieren wir dieses Ungetüm. Um dem Compiler mitzuteilen, wie er zwei Strings vergleichen soll, müssen wir eine sogenannte *anonyme innere Klasse* (Anonymous Inner Class) erzeugen. Wir rufen new Comparator<String>() auf, deklarieren ad hoc eine Klasse ohne Namen, überschreiben die Methode compare und verstecken unsere winzige, einzeilige Geschäftslogik in diesem massiven Rüstzeug.

Das ist, als ob Sie einen Kollegen bitten, einen Brief zu sortieren, und ihm dafür nicht nur die Sortierregel mitteilen, sondern ihm ein komplett neues Bürogebäude inklusive Schreibtisch und Namensschild bauen, nur damit er dort diese eine Regel ausführt. Es ist enorm viel Boilerplate-Code (syntaktischer Ballast), der die Lesbarkeit drastisch verschlechtert.

3.2 Ebene 2: Der Paradigmenwechsel (Lambdas)

Mit Java 8 brach die Ära der **Lambda-Ausdrücke** an. Ein Lambda-Ausdruck ist im Grunde nichts anderes als eine anonyme Funktion: eine Funktion, die keinen Namen braucht und nicht an eine umschließende Klasse gebunden sein muss.

Mit Lambdas können wir Verhaltensweisen endlich als kompakte Datenpakete verpacken und direkt übergeben. Der Compiler übernimmt die schwere Arbeit im Hintergrund. Schauen wir uns an, wie unser Sortier-Beispiel auf Ebene 2 aussieht:

```
List<String> namen = Arrays.asList("Maximilian", "Jan", "Anna",
                                   "Christopher");

// Ebene 2: Der Lambda-Ausdruck
namen.sort((String name1, String name2) -> {
    return Integer.compare(name1.length(), name2.length());
});
```

Das sieht schon viel aufgeräumter aus! Wir haben das Schlüsselwort new, den Interface-Namen und den Methodennamen komplett eliminiert. Alles, was übrig bleibt, ist die Essenz der Funktion.

Die Syntax eines Lambda-Ausdrucks besteht aus drei Teilen:

1. **Die Parameter-Liste:** (String name1, String name2) – Die Eingabewerte, die unsere Fabrik-Station erwartet.
2. **Der Pfeil-Operator:** -> – Er trennt die Parameter von der Logik und signalisiert dem Compiler: "Hier kommt eine Funktion."
3. **Der Rumpf (Body):** { return ...; } – Die eigentliche Logik.

Aber wir können noch einen Schritt weitergehen. Java besitzt eine sogenannte **Type Inference** (Typableitung). Da die Methode sort auf einer List<String> aufgerufen wird, *weiß* der Compiler bereits, dass die Parameter der Sortierregel Strings sein müssen. Wir können die Typen weglassen! Und da unser Body nur aus einer einzigen Anweisung besteht, können wir auch die geschweiften Klammern und das return-Schlüsselwort weglassen.

Die maximal reduzierte Form unseres Lambdas sieht so aus:

```
namen.sort((name1, name2) ->
                Integer.compare(name1.length(), name2.length()));
```

Das ist der Moment, in dem Code anfängt, elegant zu werden. Wir übergeben nur noch das reine Verhalten.

3.3 Ebene 3: Absolute Deklarativität (Methodenreferenzen)

Es gibt jedoch Situationen, in denen selbst ein Lambda-Ausdruck noch zu gesprächig ist. Sehr oft schreiben wir Lambdas, die nichts anderes tun, als die übergebenen Parameter direkt an eine bereits existierende Methode weiterzureichen.

Nehmen wir an, Sie haben eine Schleife, die jedes Element auf der Konsole ausgibt:

```
namen.forEach(name -> System.out.println(name));
```

Das Lambda nimmt name entgegen und reicht es sofort an println weiter. Für diese Fälle bietet Java eine noch kompaktere Syntax: die **Methodenreferenz (Method Reference)**. Sie wird durch den doppelten Doppelpunkt :: eingeleitet.

Anstatt eine neue anonyme Funktion zu schreiben (das Lambda), *verweisen* wir einfach auf eine Funktion, die es schon gibt. Der Code von oben wird zu:

```
namen.forEach(System.out::println);
```

Lassen Sie uns nun unsere Sortier-Logik auf diese dritte, höchste Ebene der Deklarativität heben. Java bietet in der Klasse Comparator praktische Hilfsmethoden an, die Methodenreferenzen perfekt unterstützen. Wir weisen Java an: "Erzeuge einen Comparator, indem du einen Integer-Wert vergleichst, und hole dir diesen Wert über die length-Methode der String-Klasse."

```
List<String> namen = Arrays.asList("Maximilian", "Jan", "Anna",
                                   "Christopher");
```

```
// Ebene 3: Die Methodenreferenz in Kombination mit funktionalen Helfern
namen.sort(Comparator.comparingInt(String::length));
```

Lesen Sie diesen Code laut vor: *Sortiere die Namen, indem du sie als Integer anhand ihrer String-Länge vergleichst*. Wir haben den Übergang vollständig vollzogen. Von einem unübersichtlichen, sechszeiligen Block aus abstrakter Objektorientierung (Ebene 1) sind wir bei einem deklarativen Einzeiler angekommen (Ebene 3), der sich fast wie natürliche Sprache liest.

Sie wissen nun, *wie* die Syntax aussieht, um Verhalten zu übergeben. Doch eine entscheidende Frage ist noch offen: Wie weiß der Java-Compiler eigentlich, dass unser kompakter Lambda-Ausdruck (name1, name2) -> ... exakt an die Stelle des alten Comparator-Interfaces passt? Woher kennt er die Verträge unserer Fabrik?

Die Antwort darauf liegt in einer der cleversten Design-Entscheidungen der Java-Geschichte. Wir lüften dieses Geheimnis im nächsten Kapitel: **Der funktionale Vertrag – Das @FunctionalInterface**.

4 Der funktionale Vertrag – Das @FunctionalInterface

Im letzten Kapitel haben wir gesehen, wie extrem kompakt und elegant Code wird, wenn wir Verhalten in Form von Lambda-Ausdrücken übergeben. Aber vielleicht hat sich in Ihrem Kopf dabei eine drängende Frage geformt: *Woher weiß der Java-Compiler eigentlich, dass mein kurzer Lambda-Ausdruck (name1, name2) -> ... exakt an die Stelle passt, an der die sort-Methode eigentlich ein komplexes Comparator-Objekt erwartet?*

Java ist eine streng typisierte Sprache. Der Compiler lässt normalerweise nicht zu, dass wir Äpfel mit Birnen vergleichen oder eine Funktion übergeben, wo ein Objekt gefordert ist. Wie wurde dieser Konflikt gelöst, ohne die Regeln von Java komplett auf den Kopf zu stellen?

Die Antwort ist der "funktionale Vertrag". Java nutzt einen brillanten Trick, um reine Funktionen in das Gewand von klassischen Objekten zu hüllen. Dieser Trick basiert auf sogenannten **Funktionalen Interfaces**.

4.1 Das Konzept hinter Single Abstract Method (SAM)

Ein Interface in Java ist wie ein Vertrag. Es definiert, welche Methoden eine Klasse implementieren muss, wenn sie diesen Vertrag unterschreibt. Ein normales Interface kann dutzende Methoden vorschreiben (denken Sie an das List-Interface mit Methoden wie add, remove, size usw.).

Ein funktionales Interface ist hingegen ein Spezialfall. Die Definition ist so simpel wie mächtig: **Ein funktionales Interface ist ein Interface, das exakt eine einzige abstrakte Methode besitzt.** Man nennt dies auch einen **SAM-Typ** (Single Abstract Method).

Warum ist diese Beschränkung auf exakt eine Methode so wichtig? Weil sie Eindeutigkeit schafft! Wenn ein Interface nur eine einzige Methode fordert, dann gibt es absolut keinen Zweifel daran, *welche* Methode gemeint ist, wenn wir einen namenlosen Lambda-Ausdruck übergeben. Der Compiler nimmt unseren Lambda-Ausdruck und stülpt ihm quasi unsichtbar das Interface über.

Um diesen Vertrag im Code explizit zu machen, hat Java die Annotation @FunctionalInterface eingeführt.

```
@FunctionalInterface
public interface QualitaetsKontrolleur {
    boolean pruefe(Artikel artikel);
}
```

Diese Annotation funktioniert ähnlich wie @Override. Sie ist nicht zwingend notwendig – der Compiler erkennt ein SAM-Interface auch ohne sie. Aber sie ist ein Schutzmechanismus: Wenn Sie oder ein Kollege später versuchen, eine zweite abstrakte Methode zu diesem Interface hinzuzufügen, wird der Compiler sofort einen Fehler werfen. Die Annotation garantiert, dass dieses Interface für alle Zeiten kompatibel mit Lambda-Ausdrücken bleibt.

4.2 Der Vertrag in der Praxis

Lassen Sie uns diesen Vertrag in unserer Datenfabrik anwenden. Wir bauen eine Station an unserem Fließband, die Artikel aussortiert, die unsere Qualitätsstandards nicht erfüllen.

Zunächst definieren wir unsere Fabrik-Methode, die unseren funktionalen Vertrag (QualitaetsKontrolleur) als Parameter erwartet:

```
public class Fliessband {

    // Die Methode erwartet ein Objekt, das den Vertrag erfüllt
    public List<Artikel> aussortieren(List<Artikel> alleArtikel,
                                      QualitaetsKontrolleur kontrolleur) {
        List<Artikel> bestandeneArtikel = new ArrayList<>();
        for (Artikel artikel : alleArtikel) {
            // Wir rufen die einzige Methode des Interfaces auf
            if (kontrolleur.pruefe(artikel)) {
                bestandeneArtikel.add(artikel);
            }
        }
        return bestandeneArtikel;
    }
}
```

Aus Sicht der aussortieren-Methode ist alles beim Alten geblieben. Sie ruft ganz klassisch die Methode pruefe() auf einem Objekt auf.

Der Zauber passiert beim **Aufrufer** dieser Methode. Da QualitaetsKontrolleur ein funktionales Interface ist, müssen wir keine Klasse mehr schreiben, die dieses Interface implementiert. Wir übergeben einfach die Logik für die pruefe-Methode als Lambda:

```
Fliessband band = new Fliessband();
List<Artikel> lieferung = // ... Initialisierung ...

// Der Compiler mappt das Lambda automatisch auf die Methode
// 'boolean pruefe(Artikel artikel)'
List<Artikel> premiumArtikel = band.aussortieren(lieferung, artikel ->
artikel.getPreis() > 50.0);
```

```
List<Artikel> guenstigeArtikel = band.aussortieren(lieferung, artikel ->
artikel.getPreis() <= 10.0);
```

Der Compiler leistet hier Schwerstarbeit für uns (Type Inference). Er sieht den Aufruf und denkt sich:

1. *"Die Methode aussortieren erwartet einen QualitaetsKontrolleur."*
2. *"Das ist ein funktionales Interface mit der Methode boolean pruefe(Artikel artikel)."*
3. *"Der Programmierer hat ein Lambda artikel -> ... übergeben."*
4. *"Passt der Rückgabewert des Lambdas (boolean) zur geforderten Methode? Ja!"*
5. *"Ich erzeuge im Hintergrund die notwendige Verknüpfung."*

Dieser Mechanismus ist der Grund, warum wir in Java funktional programmieren können, ohne dass die JVM (Java Virtual Machine) in ihrem Kern verändert werden musste. Alles bleibt typsicher.

4.3 Die eiserne Regel: Lexikalisches Scoping und "effectively final"

Wenn wir Funktionen als Datenpakete durch unser Programm reichen, stoßen wir sehr bald auf ein interessantes Problem: Was passiert, wenn unser Lambda-Ausdruck auf Variablen zugreifen möchte, die *außerhalb* des Lambdas definiert wurden?

Man nennt dieses Konzept eine **Closure** (Funktionsabschluss). Das Lambda "schließt" die Umgebung, in der es definiert wurde, "ein".

Schauen wir uns ein scheinbar harmloses Beispiel an. Wir wollen den Mindestpreis für unsere Qualitätskontrolle nicht fest in das Lambda schreiben (Hardcoding), sondern aus einer lokalen Variablen lesen:

```
public void starteSchicht() {
    double mindestPreis = 50.0; // Eine lokale Variable

    Fliessband band = new Fliessband();

    // Das Lambda greift auf die Variable von "außen" zu. Das ist erlaubt!
    List<Artikel> premium = band.aussortieren(lieferung,
                                artikel -> artikel.getPreis() > mindestPreis);
}
```

Dieser Code kompiliert und funktioniert fehlerfrei. Das Lambda darf die lokale Variable mindestPreis lesen.

Aber was passiert, wenn wir versuchen, diese Variable **zu verändern**?

```
public void starteSchicht() {
```

```
    int aussortiertZaehler = 0; // Lokale Variable

    Fliessband band = new Fliessband();

    band.aussortieren(lieferung, artikel -> {
        if (artikel.getPreis() < 10.0) {
            // KOMPILIERFEHLER!
            aussortiertZaehler++;
            return false;
        }
        return true;
    });
}
```

Hier wirft der Compiler sofort einen Fehler: *„Variable used in lambda expression should be final or effectively final.“* (Variablen, die in Lambda-Ausdrücken verwendet werden, müssen final oder effektiv final sein).

Warum ist Java hier so streng?

Um das zu verstehen, müssen wir wieder an unsere Fabrik denken. Die Methode starteSchicht() ist der Vorarbeiter. Er deklariert die Variable aussortiertZaehler. Dann schreibt er eine Arbeitsanweisung (das Lambda) und schickt diese Anweisung an das Fließband (die Methode aussortieren).

Das Problem ist der Lebenszyklus (Scope). Lokale Variablen (wie aussortiertZaehler) leben auf dem *Stack* des aktuellen Threads und werden gelöscht, sobald die Methode starteSchicht() beendet ist. Ein Lambda-Ausdruck wird jedoch oft asynchron oder zu einem viel späteren Zeitpunkt ausgeführt (z.B. in einem anderen Thread bei parallelen Streams). Wenn das Lambda dann versuchen würde, den Zähler zu erhöhen, existiert diese Variable auf dem Stack des ursprünglichen Threads vielleicht gar nicht mehr!

Java löst dieses Problem, indem es dem Lambda eine **Kopie** der äußeren Variablen mitgibt. Damit es nun nicht zu völliger Verwirrung kommt (das Lambda ändert seine Kopie, aber der Vorarbeiter sieht die Änderung in seiner Original-Variable nicht), zieht Java eine harte Grenze: **Variablen, die in ein Lambda hineingereicht werden, dürfen nicht verändert werden.** Sie müssen Konstanten sein.

Sie müssen nicht explizit das Schlüsselwort final vor die Variable schreiben. Es reicht, wenn Sie die Variable nach ihrer ersten Zuweisung nie wieder ändern. Der Compiler nennt das **"effectively final"** (effektiv final). Sobald Sie versuchen, aussortiertZaehler++ zu schreiben, verliert die Variable ihren "effectively final"-Status und der Compiler bricht ab.

Diese eiserne Regel ist kein lästiges Hindernis, sondern Ihr größter Beschützer. Sie

zwingt uns dazu, Nebenläufigkeitsprobleme und versteckte Seiteneffekte (wie das heimliche Hochzählen einer lokalen Variable) gar nicht erst entstehen zu lassen. Sie erzieht uns zur reinen, funktionalen Programmierung.

5 Der Standard-Werkzeugkasten – Das java.util.function-Paket

Im letzten Kapitel haben wir einen eigenen Vertrag geschrieben: das funktionale Interface QualitaetsKontrolleur. Es hat seinen Zweck hervorragend erfüllt und uns erlaubt, Lambda-Ausdrücke an unsere aussortieren-Methode zu übergeben.

Doch stellen Sie sich vor, was passieren würde, wenn jeder Entwickler und jeder Hersteller von Java-Bibliotheken für jedes noch so kleine funktionale Problem eigene Interfaces schreiben würde.

5.1 Das Problem mit eigenen Interfaces

In einer großen Anwendung hätten Sie bald ein Interface ArtikelPruefer im Backend, ein Interface KundenValidator in der Datenbank-Schicht und ein Interface RechnungsFilter in der Rechnungsstellung. Alle drei Interfaces hätten exakt dieselbe Signatur: Sie nehmen ein Objekt entgegen und geben einen boolean (wahr oder falsch) zurück. Trotzdem wären sie untereinander völlig inkompatibel, weil sie unterschiedliche Namen tragen. Sie könnten eine Filter-Funktion aus der Datenbank-Schicht nicht an die Rechnungsstellung übergeben. Das Fließband würde ins Stocken geraten.

Um dieses Chaos zu verhindern, haben die Java-Architekten mit Java 8 das Paket java.util.function eingeführt. Es ist der universelle Standard-Werkzeugkasten unserer Datenfabrik. Anstatt eigene Interfaces zu definieren, greifen wir ab sofort auf diese standardisierten Schablonen zurück. Sie kategorisieren Funktionen nicht nach ihrem fachlichen Zweck (wie "Qualitätskontrolle"), sondern ausschließlich nach ihrer **strukturellen Form** (Eingabe und Ausgabe).

Lassen Sie uns die vier wichtigsten Maschinen dieses Werkzeugkastens genauer betrachten.

5.2 Werte transformieren: Function und BiFunction

Die mit Abstand häufigste Aufgabe in einer Fabrik ist die Transformation. Rohmaterial kommt herein, ein fertiges Produkt kommt heraus. In der Programmierung bedeutet das: Wir stecken ein Objekt vom Typ T hinein und erhalten ein Objekt vom Typ R (Resultat) zurück.

Dafür gibt es das Interface Function<T, R>. Seine einzige abstrakte Methode heißt apply(T t).

Nehmen wir an, wir haben eine Liste von Artikeln, aber für den Druck von Etiketten benötigen wir nur die Artikelnummern (Strings). Wir benötigen eine Maschine, die einen Artikel in einen String transformiert.

```
// Die Standard-Schablone Function<Eingabe, Ausgabe>
Function<Artikel, String> extrahiereNummer = artikel -> artikel.getNummer();

// Anwendung der Funktion
Artikel meinArtikel = new Artikel("A-123", "Hammer", 15.99);
String nummer = extrahiereNummer.apply(meinArtikel);

System.out.println(nummer); // Ausgabe: A-123
```

Noch eleganter wird es, wenn wir, wie in Kapitel 3 gelernt, Methodenreferenzen nutzen:

```
Function<Artikel, String> extrahiereNummer = Artikel::getNummer;
```

Manchmal reicht ein einziger Eingabewert nicht aus. Wenn Sie zum Beispiel einen Rabatt auf einen Artikel anwenden wollen, benötigen Sie den Artikel und den Prozentsatz. Dafür existiert die BiFunction<T, U, R>, die zwei Eingabewerte (T und U) in ein Ergebnis (R) transformiert:

```
// BiFunction<Eingabe1, Eingabe2, Ausgabe>
BiFunction<Artikel, Double, Double> rabattRechner =
    (artikel, rabatt) -> artikel.getPreis() * (1.0 - rabatt);

double neuerPreis = rabattRechner.apply(meinArtikel, 0.20); // 20% Rabatt
```

5.3 Bedingungen prüfen: Predicate

Erinnern Sie sich an unseren QualitaetsKontrolleur? Er nahm einen Artikel und gab true oder false zurück. Für genau dieses Muster – das Testen einer Bedingung – gibt es das Interface Predicate<T>. Die abstrakte Methode heißt test(T t).

Ein Prädikat (Predicate) ist der ultimative Weichensteller an unserem Fließband. Es entscheidet, ob Daten weiterverarbeitet werden oder in den Ausschuss wandern.

Wir können unsere alte aussortieren-Methode nun standardisieren und unser eigenes Interface löschen:

```
// Die Methode nutzt nun den Standard-Vertrag Predicate<T>
public List<Artikel> aussortieren(List<Artikel> alle, Predicate<Artikel>
bedingung) {
    List<Artikel> ergebnis = new ArrayList<>();
```

```
    for (Artikel a : alle) {
        if (bedingung.test(a)) { // Aufruf der test-Methode
            ergebnis.add(a);
        }
    }
    return ergebnis;
}
```

Der Aufruf bleibt dabei dank der Lambda-Syntax exakt gleich und wunderbar lesbar:

```
List<Artikel> teuer = aussortieren(lieferung, a -> a.getPreis() > 100.0);
```

5.4 Daten konsumieren und erzeugen: Consumer und Supplier

Nicht jede Station am Fließband nimmt etwas an und gibt etwas anderes zurück. Es gibt auch den Anfang und das Ende des Bandes.

Der Consumer (Verbraucher):

Ein Consumer<T> nimmt ein Objekt vom Typ T entgegen, aber er gibt nichts zurück (void). Seine Methode heißt accept(T t).

Er ist die Endstation. Hier passieren typischerweise die unausweichlichen Seiteneffekte, die wir an den Rand unseres Systems drängen wollen: Daten in eine Datenbank schreiben, eine E-Mail versenden oder etwas auf der Konsole ausgeben.

```
Consumer<Artikel> druckeEtikett = artikel -> {
    System.out.println("Drucke Etikett für: " + artikel.getName());
    // ... Logik zur Ansteuerung des Druckers ...
};

druckeEtikett.accept(meinArtikel);
```

Der Supplier (Lieferant):

Der Supplier<T> ist das exakte Gegenteil. Er nimmt keine Eingabeparameter entgegen, generiert aber bei jedem Aufruf ein Objekt vom Typ T. Seine Methode heißt get().

Er ist der Startpunkt des Fließbands. Man nutzt ihn häufig für das verzögerte Erzeugen von teuren Objekten (Lazy Initialization) oder zum Generieren von Zufallswerten und IDs.

```
Supplier<String> idGenerierer = () -> UUID.randomUUID().toString();

String neueId = idGenerierer.get();
```

5.5 Spezialisten für Performance: Primitive funktionale Interfaces

Wenn Sie aufmerksam mitgelesen haben, ist Ihnen vielleicht ein Detail aufgefallen: Wir haben bei den generischen Typen der Interfaces (wie Predicate<T>) immer Objekt-Typen wie Double oder Integer verwendet. In Java können Generics leider (noch) nicht mit primitiven Datentypen wie int, double oder boolean umgehen.

Wenn wir nun eine Liste von Millionen von primitiven int-Werten haben und ein normales Predicate<Integer> verwenden, muss Java für jede einzelne Zahl ein neues Integer-Objekt erzeugen (Autoboxing) und es danach wieder entpacken (Unboxing). Bei großen Datenmengen führt dieser ständige Objekt-Erzeugungs-Overhead zu massiven Performance-Einbrüchen und zwingt den Garbage Collector in die Knie.

Um dieses Problem zu lösen, liefert der Werkzeugkasten für jeden primitiven Typ maßgeschneiderte Spezial-Interfaces mit.

- Statt Predicate<Integer> verwenden Sie **IntPredicate**.
- Statt Function<Double, Integer> verwenden Sie **DoubleToIntFunction**.
- Statt Consumer<Long> verwenden Sie **LongConsumer**.

Diese spezialisierten Interfaces arbeiten direkt auf den primitiven Werten, ganz ohne Autoboxing.

```
// Langsam: Autoboxing bei jedem Aufruf (int -> Integer -> int)
Predicate<Integer> istGeradeObjekt = zahl -> zahl % 2 == 0;

// Schnell: Arbeitet direkt auf dem primitiven int
IntPredicate istGeradePrimitiv = zahl -> zahl % 2 == 0;
```

In unserer täglichen Geschäftslogik (dem Verarbeiten von Rechnungen und Kunden) spielen diese primitiven Spezialisten kaum eine Rolle. Aber wenn Sie Algorithmen schreiben, die physikalische Messdaten oder Pixel von Bildern verarbeiten, sind sie der Schlüssel zur Hochperformance.

Fazit des ersten Teils

Sie haben nun die Werkzeuge unserer Datenfabrik kennengelernt. Sie wissen, wie Sie Verhalten als Lambda-Ausdrücke verpacken, wie Methodenreferenzen den Code auf das absolute Minimum reduzieren und welche Standard-Schablonen (Function, Predicate, Consumer, Supplier) Sie nutzen müssen, um diese Funktionen typsicher

durch Ihre Anwendung zu reichen.

Bisher haben wir diese Werkzeuge jedoch nur in unsere eigenen, handgeschriebenen for-Schleifen (wie die Methode aussortieren()) gesteckt. Wir haben das volle Potenzial noch lange nicht entfesselt. Es ist an der Zeit, die handbetriebenen Laufbänder abzureißen und die echten, hochautomatisierten Fließbänder von Java in Betrieb zu nehmen.

Willkommen in Teil II: **Daten im Fluss – Die Stream-API**.

Teil II:
Daten im Fluss – Die Stream-API

6 Deklarative Datenverarbeitung – Parallele und Sequenzielle Streams

Bisher haben wir in unserer Datenfabrik nur die einzelnen Maschinen (unsere funktionalen Interfaces wie Predicate oder Function) betrachtet. Wir haben gelernt, wie wir das Verhalten dieser Maschinen kompakt als Lambda-Ausdrücke definieren. Doch eine Maschine allein macht noch keine Fabrik. Die Maschinen müssen hintereinandergeschaltet werden. Das Rohmaterial muss von der Qualitätskontrolle zur Transformation und schließlich in die Verpackung fließen.

In der klassischen Java-Welt haben wir den Transport der Daten zwischen diesen Maschinen manuell übernommen – meist durch verschachtelte for-Schleifen und das ständige Umkopieren von Elementen in neue Listen. Ab Java 8 nehmen uns die Architekten der Sprache diese schwere Arbeit ab. Sie präsentieren uns das ultimative Fließband: die **Stream-API**.

6.1 Der Paradigmenwechsel: Fließband statt for-Schleife

Lassen Sie uns den Unterschied an einem konkreten Problem festmachen. Wir haben eine Liste von Sensordaten (Temperaturmessungen). Unsere Aufgabe: *Finde alle Messwerte über 30 Grad, wandle sie in den Datentyp String um und sortiere sie absteigend.*

Der klassische, imperative Weg sieht so aus:

```
List<Double> messungen = Arrays.asList(22.5, 31.0, 18.9, 35.5, 29.0);

// 1. Filtern in eine neue Liste (Zustand!)
List<Double> heisseMessungen = new ArrayList<>();
for (Double m : messungen) {
    if (m > 30.0) {
        heisseMessungen.add(m);
    }
}

// 2. Sortieren der Zwischenliste
Collections.sort(heisseMessungen, Collections.reverseOrder());

// 3. Transformieren in Strings (Noch eine Liste!)
List<String> warnungen = new ArrayList<>();
for (Double m : heisseMessungen) {
    warnungen.add(m.toString() + " Grad");
}
```

Dieser Code ist nicht nur langatmig, er verschwendet auch Speicherplatz durch das ständige Erzeugen von Zwischenlisten (heisseMessungen). Außerdem diktieren wir der Maschine jeden einzelnen Schritt (das *Wie*).

Wenn wir dasselbe Problem deklarativ mit der Stream-API lösen, bauen wir stattdessen ein Fließband auf:

```
List<String> warnungen = messungen.stream()
    .filter(m -> m > 30.0)
    .sorted(Collections.reverseOrder())
    .map(m -> m.toString() + " Grad")
    .toList();
```

Fällt Ihnen auf, wie elegant das ist? Wir deklarieren nur noch unsere Absicht (das *Was*). Es gibt keine Zwischenlisten, die wir manuell verwalten müssen, und keine Index-Variablen.

Noch beeindruckender wird es, wenn unsere Datenfabrik plötzlich Millionen von Messwerten gleichzeitig verarbeiten muss. Beim imperativen Ansatz müssten Sie nun komplexe Thread-Pools und synchronisierte Listen einrichten. Bei der Stream-API ändern Sie genau ein einziges Wort:

```
// Aus .stream() wird .parallelStream()
List<String> warnungen = messungen.parallelStream()
    // ... der Rest bleibt exakt gleich!
```

Da wir in unserem Fließband auf veränderliche Zustände verzichten und nur reine Funktionen (Lambdas) übergeben, kann Java den Datenstrom völlig gefahrlos in mehrere Stränge aufteilen, diese auf allen CPU-Kernen parallel verarbeiten und am Ende wieder zusammensetzen. Das ist die wahre Macht der funktionalen Programmierung.

6.2 Die Anatomie einer Pipeline: Quelle, Operation, Senke

Jedes funktionale Fließband (eine sogenannte *Stream-Pipeline*) besteht immer aus exakt drei Bausteinen, die in einer strikten Reihenfolge stehen.

1. **Die Quelle (Source):** Wo kommen die Rohdaten her? Ein Stream speichert selbst keine Daten. Er ist nur ein Kanal, durch den Daten fließen. Quellen können Listen sein (liste.stream()), Arrays (Arrays.stream(array)), Dateien (Files.lines(path)) oder sogar unendliche Generatoren (Stream.generate(...)).
2. **Intermediäre Operationen (Zwischenschritte):** Das sind die Maschinen an unserem Fließband. Methoden wie filter(), map() oder sorted(). Das Besondere an

ihnen: Sie geben *immer* wieder einen neuen Stream zurück. Dadurch können wir sie wie Legosteine fast endlos aneinanderketten (Method Chaining).

3. **Die Senke (Terminale Operation):** Die Endstation. Ein Stream muss immer mit einer terminalen Operation enden. Diese Operation gibt keinen Stream mehr zurück, sondern ein echtes Resultat (z. B. eine List, einen boolean oder eine einzelne Zahl) oder sie führt einen finalen Konsumvorgang aus (z. B. forEach). Sobald eine terminale Operation aufgerufen wurde, ist der Stream "verbraucht" und kann nicht noch einmal verwendet werden.

6.3 Die Magie unter der Haube: Lazy Evaluation

Vielleicht haben Sie beim Betrachten der Pipeline oben eine Befürchtung: *Wenn ich eine Liste mit 10.000 Elementen habe, wandert die filter-Methode dann erst 10.000 Mal über alle Daten, reicht das Ergebnis an die map-Methode weiter, die dann wieder 10.000 Iterationen macht? Ist das nicht extrem ineffizient?*

Die Antwort ist ein klares Nein. Streams in Java nutzen ein Konzept namens **Lazy Evaluation (Faule Auswertung)**.

Das bedeutet: Wenn Sie intermediäre Operationen wie filter oder map aneinanderreihen, passiert im Hintergrund zunächst **absolut gar nichts**. Java notiert sich lediglich den Bauplan Ihres Fließbands. Die Daten fangen erst an zu fließen, wenn die *terminale Operation* (die Senke) aufgerufen wird! Man sagt auch: Terminale Operationen "ziehen" die Daten durch die Pipeline.

Zudem fließen die Daten nicht phasenweise, sondern **vertikal (Element für Element)**.

Beweisen wir das durch ein kleines Experiment. Wir nutzen die Methode peek(), die wie ein Fenster am Fließband funktioniert: Sie nimmt ein Element entgegen, erlaubt uns einen Blick darauf (z. B. für eine Konsolenausgabe) und reicht es unverändert weiter.

```
List<String> namen = Arrays.asList("Anna", "Bob", "Charlie", "David");

System.out.println("Baue das Fließband auf...");
Stream<String> fliessband = namen.stream()
    .peek(name -> System.out.println("Filter prüft: " + name))
    .filter(name -> name.startsWith("C"))
    .peek(name -> System.out.println("Map transformiert: " + name))
    .map(String::toUpperCase);

System.out.println("Fließband ist gebaut. Noch ist nichts passiert!");
```

```
// Jetzt schalten wir die Maschine ein (Terminale Operation)
List<String> ergebnis = fliessband.toList();

Baue das Fließband auf...
Fließband ist gebaut. Noch ist nichts passiert!
Filter prüft: Anna
Filter prüft: Bob
Filter prüft: Charlie
Map transformiert: Charlie
Filter prüft: David
```

Beobachten Sie das Verhalten genau: "Anna" und "Bob" betreten das Fließband, scheitern am filter und fallen direkt herunter. Dann kommt "Charlie". Er besteht den filter und wird *sofort* an die nächste Station (map) weitergereicht, bevor "David" überhaupt den ersten Schritt macht!

Diese faule Auswertung ist unglaublich effizient. Wenn unsere terminale Operation beispielsweise findFirst() (Finde das erste passende Element) wäre, würde das Fließband sofort nach "Charlie" komplett stoppen. "David" würde nicht einmal mehr geprüft werden. Wir sparen uns enorme Mengen an Rechenzeit.

6.4 Gefahren und Anti-Patterns: Seiteneffekte in Streams

Mit großer Macht kommt große Verantwortung. Die Stream-API verleitet durch ihre elegante Syntax schnell dazu, alte, imperative Gewohnheiten in ein neues Gewand zu pressen. Das führt zum gefährlichsten Anti-Pattern der funktionalen Programmierung in Java: **Seiteneffekte (Side Effects) innerhalb von Streams**.

Erinnern Sie sich an unsere eiserne Regel: Die Maschinen am Fließband dürfen das restliche System nicht beeinflussen. Sie transformieren nur.

Ein katastrophales Beispiel, das in der Praxis leider häufig vorkommt:

```
// ANTI-PATTERN: Tun Sie das niemals!
List<String> fehlerhafteArtikel = new ArrayList<>();
  // Zustand außerhalb des Streams

alleArtikel.stream()
    .filter(a -> a.getPreis() < 0)
    .forEach(a -> fehlerhafteArtikel.add(a.getName())); // Seiteneffekt!
```

Hier missbrauchen wir das Fließband. Anstatt die Daten elegant bis zum Ende fließen zu lassen und dort in einer neuen Liste zu sammeln, greifen wir *aus dem Fließband heraus* und modifizieren eine externe Liste.

Warum ist das so gefährlich?

1. **Verlust der Parallelisierbarkeit:** Sobald Sie aus .stream() ein .parallelStream() machen, greifen plötzlich mehrere Threads unkoordiniert auf die ArrayList zu. Da ArrayList nicht Thread-safe ist, kommt es zu Race Conditions, Datenverlust oder Abstürzen (ConcurrentModificationException).
2. **Schlechte Lesbarkeit:** Der Code täuscht eine funktionale Pipeline vor, ist aber im Herzen immer noch rein imperativ und zustandsbehaftet.

Der korrekte, rein funktionale Weg sieht so aus: Wir nutzen die Senke, um den Zustand typsicher zu erzeugen.

```
// BEST PRACTICE: Rein funktional ohne Seiteneffekte
List<String> fehlerhafteArtikel = alleArtikel.stream()
    .filter(a -> a.getPreis() < 0)
    .map(Artikel::getName) // Erst das Objekt in den Namen transformieren
    .toList();             // Die Endstation sammelt alles sicher ein
```

Wir haben in diesem Kapitel die Architektur unserer Fabrik verstanden. Wir kennen die Quelle, die Zwischenschritte und die Senke, und wir wissen, warum Lazy Evaluation unsere Programme so performant macht.

Im nächsten Kapitel steigen wir in die Details der Datenmanipulation ein. Wir werfen einen detaillierten Blick auf die mächtigsten Maschinen unseres Fließbands: Die intermediären Operationen, und lüften das Geheimnis, wie man mit flatMap sogar mehrdimensionale Datenstrukturen flachklopft.

7 Das Fließband steuern – Intermediäre Operationen

In Kapitel 6 haben wir das Grundgerüst unserer Fließbänder aufgebaut. Wir wissen nun, dass Daten faul (lazy) und vertikal durch unsere Pipeline fließen. Doch was genau passiert zwischen der Datenquelle und der finalen Senke?

Hier kommen die **intermediären Operationen** ins Spiel. Sie sind die eigentlichen Werkzeugmaschinen unserer Datenfabrik. Jede dieser Maschinen nimmt einen Stream entgegen, führt eine spezifische Aufgabe aus und gibt sofort wieder einen (oftmals veränderten) Stream zurück, der an die nächste Station weitergeleitet wird.

In diesem Kapitel betrachten wir die wichtigsten dieser Maschinen im Detail. Wir klären, wie wir Daten filtern, transformieren und wie wir mit dem gefürchteten flatMap komplexe, verschachtelte Datenstrukturen elegant flachklopfen.

7.1 Der Qualitätskontrolleur: filter()

Die einfachste und gleichzeitig häufigste Maschine an unserem Fließband ist der Filter. Wie der Name schon sagt, reduziert diese Operation die Anzahl der Elemente im Datenstrom.

Die filter()-Methode erwartet als Parameter unseren alten Bekannten aus Kapitel 5: ein Predicate<T>. Zur Erinnerung: Ein Prädikat ist eine Funktion, die ein Objekt testet und einen boolean (true oder false) zurückgibt.

Nur wenn das Prädikat true liefert, darf das Element auf dem Fließband bleiben und zur nächsten Station weiterrollen. Liefert es false, wird das Element gnadenlos vom Band geworfen und vom Garbage Collector entsorgt.

```
List<Rechnung> alleRechnungen = // ... Liste von hunderten Rechnungen

// Wir filtern alle offenen Rechnungen mit einem Wert über 1000 Euro heraus
List<Rechnung> kritischeRechnungen = alleRechnungen.stream()
    .filter(r -> !r.isBezahlt())             // 1. Qualitätskontrolle
    .filter(r -> r.getBetrag() > 1000.0)     // 2. Qualitätskontrolle
    .toList();
```

Beachten Sie, wie wir zwei filter-Aufrufe hintereinandergeschaltet haben. Wir hätten die Bedingungen auch mit einem logischen UND (&&) in ein einziges Lambda packen können: filter(r -> !r.isBezahlt() && r.getBetrag() > 1000.0). Die Aufteilung in zwei separate Filter-Stationen ist jedoch oft besser lesbar und hat dank der "Lazy Evaluation" (faule Auswertung) keinerlei negative Auswirkungen auf die Performance.

7.2 Die Transformation: map()

Während filter die *Menge* der Daten reduziert, ändert map die *Form* der Daten.

Die map()-Methode erwartet eine Function<T, R>. Sie nimmt ein Element vom Typ T vom Fließband, steckt es in die Funktion und legt das Ergebnis vom Typ R wieder auf das Band. Es handelt sich hierbei um eine strikte **1-zu-1-Transformation**. Für jedes Element, das in die map-Maschine hineingeht, kommt exakt ein Element heraus.

Häufig nutzen wir map, um einzelne Eigenschaften aus großen Objekten zu extrahieren oder Typen umzuwandeln.

```
List<Kunde> kunden = // ...

// Aus einem Stream von <Kunde> wird ein Stream von <String>
List<String> kundenNamen = kunden.stream()
    .filter(Kunde::istAktiv)
    .map(Kunde::getName)        // Transformation: Kunde -> String (Name)
    .map(String::toUpperCase)  // Transformation: String -> String (Großb.)
    .toList();
```

Hier sehen Sie die Eleganz von Methodenreferenzen in Perfektion. Das Fließband wandelt aktive Kunden in ihre Namen um und formatiert diese anschließend in Großbuchstaben.

7.3 Dimensionen sprengen: flatMap()

Wir kommen nun zu der Maschine, die bei Einsteigern in die funktionale Programmierung die meiste Verwirrung stiftet, aber gleichzeitig eines der mächtigsten Werkzeuge überhaupt ist: flatMap.

Um flatMap zu verstehen, müssen wir uns ein Problem ansehen, bei dem map kläglich versagt. Stellen Sie sich vor, jeder Kunde in unserem System besitzt nicht nur einen Namen, sondern eine ganze Liste von Bestellungen (List<Bestellung>).

Unsere Aufgabe: *Erzeuge eine einzige flache Liste aller jemals getätigten Bestellungen von allen Kunden.*

Wenn wir naiv an die Sache herangehen und unser bewährtes map verwenden, passiert Folgendes:

```
// VORSICHT: Das Ergebnis ist nicht das, was wir wollen!
List<List<Bestellung>> bestellListen = kunden.stream()
```

```
    .map(Kunde::getBestellungen)
    .toList();
```

Sehen Sie das Problem im Datentyp? Wir erhalten eine List<List<Bestellung>>.

In unserer Fabrik-Metapher bedeutet das: Wir haben einen Strom von Kunden auf das Fließband gelegt. Die map-Maschine hat jeden Kunden genommen und stattdessen einen *Karton* voller Bestellungen auf das Band gestellt. Am Ende haben wir eine Liste von Kartons. Das wollen wir aber nicht! Wir wollen die einzelnen Bestellungen selbst haben.

Hier kommt flatMap ("flaches Mappen") ins Spiel.

Die Regel lautet: Die Funktion, die Sie an flatMap übergeben, **muss selbst wieder einen Stream zurückgeben**. flatMap nimmt dann diesen kleinen, temporären Stream, packt dessen Elemente aus (es "öffnet den Karton") und legt die einzelnen Elemente flach auf das Haupt-Fließband.

So sieht die korrekte, elegante Lösung aus:

```
// KORREKT: Aus verschachtelten Listen wird ein einziger flacher Stream
List<Bestellung> alleBestellungen = kunden.stream()
    .flatMap(kunde -> kunde.getBestellungen().stream()) // Auspacken!
    .toList();
```

- Schritt 1: Ein Kunde betritt die flatMap-Maschine.
- Schritt 2: Wir rufen kunde.getBestellungen().stream() auf. Das ist unser "Karton", den wir zu einem temporären Stream machen.
- Schritt 3: flatMap gießt den Inhalt dieses temporären Streams nahtlos in den großen Haupt-Stream.
- Schritt 4: Am Ende haben wir eine perfekte, eindimensionale List<Bestellung>.

Mit flatMap lassen sich beliebig tiefe Objekt-Hierarchien (Firma -> Abteilungen -> Mitarbeiter -> Adressen) in einem einzigen, eleganten Ausdruck flachklopfen.

7.4 Zustandsbehaftete Operationen: distinct(), sorted(), limit()

Die Operationen filter, map und flatMap haben eine wunderbare Eigenschaft: Sie sind **zustandslos (stateless)**. Wenn die map-Maschine ein Element umwandelt, interessiert sie sich überhaupt nicht dafür, welches Element vorher auf dem Band lag oder welches als Nächstes kommt. Sie arbeitet komplett isoliert. Das macht diese Operationen extrem schnell und perfekt parallelisierbar.

Es gibt jedoch Aufgaben, die sich isoliert nicht lösen lassen. Dafür bietet die Stream-API **zustandsbehaftete (stateful)** intermediäre Operationen. Diese Maschinen müssen sich Dinge "merken" oder sogar das gesamte Fließband kurzzeitig anhalten.

- **distinct() (Duplikate entfernen):** Diese Operation lässt jedes Element nur ein einziges Mal passieren. Dafür muss sie sich intern in einem Set merken, welche Elemente sie in der Vergangenheit bereits gesehen hat. (Sie nutzt dafür die equals()- und hashCode()-Methoden der Objekte).
- **sorted() (Sortieren):** Das ist die teuerste Maschine an unserem Fließband. Denken Sie an die faule Auswertung (Lazy Evaluation): Eigentlich fließt jedes Element direkt von oben nach unten durch. Aber wie kann man ein Element sortieren, wenn man die anderen noch gar nicht gesehen hat? Die Antwort: Gar nicht! Die sorted()-Maschine bricht die vertikale Abarbeitung auf. Sie zwingt das Fließband zum Stillstand, sammelt *alle* Elemente, die von oben kommen, in einem internen Zwischenspeicher, sortiert diesen Speicher komplett durch und lässt die Elemente erst danach wieder einzeln weiterrollen. Bei unendlich großen Streams oder sehr großen Datenmengen führt sorted() daher unweigerlich zu Speicherproblemen (OutOfMemoryError).
- **limit(n) (Kurzschließend / Short-Circuiting):** Diese Maschine merkt sich, wie viele Elemente bereits an ihr vorbeigerollt sind. Sobald die Zahl n erreicht ist, schaltet limit das gesamte Fließband brutal ab. Die Datenquelle hört sofort auf, weitere Daten zu produzieren.

Lassen Sie uns diese zustandsbehafteten Operationen in einer finalen Pipeline kombinieren:

```
List<String> topDreiKategorien = bestellungen.stream()
    .map(Bestellung::getKategorie) // Nur die Kategorien betrachten
    .distinct()          // Duplikate entfernen (Zustand: Gemerkte Kategorien)
    .sorted()            // Alphabetisch sortieren (Zustand: Alles puffern!)
    .limit(3)            // Nach exakt 3 Elementen das Fließband abschalten!
    .toList();
```

Zusammenfassung

Wir beherrschen nun die Steuerung unseres Fließbands. Wir können den Datenstrom mit filter ausdünnen, mit map transformieren, verschachtelte Strukturen mit flatMap auflösen und die Daten mit Operationen wie distinct oder sorted aggregieren.

Doch was geschieht am Ende der Leitung? Bisher haben wir fast ausschließlich toList() verwendet, um die Resultate wieder in eine simple Liste zu gießen. Aber die Endstation

(die Senke) kann weitaus mehr. Wir können Daten aufsummieren, Durchschnittswerte berechnen oder sie in hochkomplexe Maps gruppieren.

Genau das ist das Thema unseres nächsten Kapitels: **8 Ergebnisse sammeln – Terminale Operationen**.

8 Ergebnisse sammeln – Terminale Operationen

Wir stehen nun am Ende unseres funktionalen Fließbands. In den letzten Kapiteln haben wir Rohmaterialien auf das Band gelegt (die Quelle) und sie durch verschiedene Maschinen geschickt, um sie zu filtern, zu transformieren und flachzuklopfen (intermediäre Operationen).

Doch wie wir in Kapitel 6 gelernt haben, ist unser Fließband "faul" (lazy). Solange wir am Ende keine Abnahmestation aufbauen, wird sich nicht ein einziges Bauteil bewegen. Wir benötigen eine **terminale Operation** (die Senke). Sie schaltet den Strom an, zieht die Daten durch alle Zwischenstationen und gießt sie in eine finale, nutzbare Form. Sobald eine terminale Operation ausgeführt wurde, ist der Stream verbraucht.

In diesem Kapitel betrachten wir die wichtigsten Abnahmestationen unserer Fabrik. Wir lernen, wie wir Daten auf einen einzigen Wert zusammendampfen, sie in Standard-Datenstrukturen verpacken und wie wir hochkomplexe Gruppierungen vornehmen, für die wir früher hunderte Zeilen Code benötigt hätten.

8.1 Die ultimative Reduktion: reduce()

Die grundlegendste Art, einen Datenstrom abzuschließen, ist die Reduktion. Stellen Sie sich vor, Sie haben tausende Rechnungen auf dem Fließband und möchten am Ende genau eine einzige Zahl haben: den Gesamtumsatz. Sie reduzieren eine Menge von N Elementen auf genau 1 Element.

Dafür bietet die Stream-API die Methode reduce(). Sie benötigt zwei Parameter:

1. **Den Startwert (Identity):** Der anfängliche Zustand unserer Berechnung (z. B. 0.0 bei einer Summe).
2. **Die Akkumulator-Funktion (Accumulator):** Eine BiFunction (genauer gesagt ein BinaryOperator), die definiert, wie das bisherige Zwischenergebnis mit dem *nächsten* Element auf dem Fließband kombiniert werden soll.

Lassen Sie uns den Gesamtpreis aller Artikel in einem Warenkorb berechnen:

```
List<Artikel> warenkorb = // ...

// Wir extrahieren erst die Preise und reduzieren sie dann auf eine Summe
double gesamtSumme = warenkorb.stream()
    .map(Artikel::getPreis)
    .reduce(0.0, (bisherigeSumme, aktPreis) -> bisherigeSumme + aktPreis);
```

```
System.out.println("Zu zahlen: " + gesamtSumme + " Euro");
```

Wie arbeitet reduce im Detail?

1. Das Fließband startet. Der erste Parameter (0.0) wird als bisherigeSumme gemerkt.
2. Der erste Preis (z.B. 10.50) rollt vom Band. Der Akkumulator rechnet: 0.0 + 10.50. Das neue Zwischenergebnis ist 10.50.
3. Der zweite Preis (5.00) kommt an. Der Akkumulator rechnet: 10.50 + 5.00. Das neue Ergebnis ist 15.50.
4. Das geht so weiter, bis das Band leer ist. Das letzte Zwischenergebnis wird als finaler Rückgabewert der Pipeline ausgegeben.

Dank Methodenreferenzen lässt sich das Lambda für die Addition sogar noch weiter verkürzen. Die Klasse Double bringt dafür die Methode sum mit:

```
.reduce(0.0, Double::sum);
```

8.2 Die Standard-Sammler: toList(), toSet(), toMap()

Nicht immer wollen wir Daten auf eine einzige Zahl reduzieren. Meistens wollen wir die transformierten Daten einfach in einer neuen Datenstruktur (Collection) sammeln und an eine andere Schicht unserer Anwendung weitergeben.

Dafür nutzen wir die collect()-Methode in Kombination mit der Werkzeugklasse Collectors, die uns fertige Schablonen für alle gängigen Datenstrukturen liefert.

(Hinweis: Seit Java 16 gibt es für Listen die bequeme Kurzschreibweise .toList() direkt auf dem Stream, die wir in den vorherigen Kapiteln schon genutzt haben. Unter der Haube und für alle anderen Strukturen arbeiten wir jedoch mit collect().)

In ein Set sammeln (Duplikate ignorieren):

Wenn wir sicherstellen wollen, dass unsere finale Sammlung jedes Element nur einmal enthält (und uns die Reihenfolge egal ist), sammeln wir in ein Set.

```
// Wir wollen wissen, in welche Städte wir heute liefern müssen
Set<String> lieferStaedte = bestellungen.stream()
    .map(Bestellung::getKundenStadt)
    .collect(Collectors.toSet()); // Sammelt in ein HashSet
```

In eine Map sammeln (Schlüssel-Wert-Paare aufbauen):

Das Sammeln in eine Map ist ein extrem häufiger Anwendungsfall, etwa wenn Sie eine Liste von Objekten später schnell über eine ID suchen wollen. Collectors.toMap() benötigt zwei Funktionen: Eine, die festlegt, was der Schlüssel (Key) wird, und eine für den Wert (Value).

```
List<Kunde> kundenListe = // ...

// Wir bauen ein schnelles In-Memory-Verzeichnis (ID -> Kunde)
Map<String, Kunde> kundenVerzeichnis = kundenListe.stream()
    .collect(Collectors.toMap(
        Kunde::getId,     // Der Schlüssel ist die ID des Kunden
        kunde -> kunde    // Der Wert ist das Kunden-Objekt selbst
    ));
```

(Tipp: Für kunde -> kunde bietet Java die Hilfsmethode Function.identity(), was oft noch professioneller aussieht).

8.3 Komplexe Gruppierungen: Collectors.groupingBy() und partitioningBy()

Jetzt betreten wir das Gebiet, in dem die funktionale Programmierung ihre absolute Überlegenheit gegenüber dem imperativen Ansatz beweist.

Stellen Sie sich vor, Sie haben eine Liste tausender Rechnungen. Ihre Aufgabe: *Gruppieren Sie alle Rechnungen nach dem Rechnungsjahr. Das Ergebnis soll eine Map sein, bei der das Jahr der Schlüssel ist und der Wert eine Liste aller Rechnungen aus diesem Jahr.*

Versuchen Sie einmal im Kopf, den imperativen Code dafür zu entwerfen. Sie müssten eine Map<Integer, List<Rechnung>> initialisieren. In einer for-Schleife müssten Sie prüfen, ob das Jahr schon in der Map existiert. Wenn nicht, eine neue leere Liste anlegen, diese in die Map legen und dann die Rechnung zur Liste hinzufügen. Das sind schnell 10 Zeilen extrem fehleranfälliger Code.

Mit der Stream-API ist das ein eleganter Einzeiler. Wir nutzen Collectors.groupingBy(). Wir müssen dem Sammler nur sagen, nach welcher Eigenschaft er gruppieren soll (die Klassifizierungs-Funktion).

```
// Gruppierung nach Jahr
Map<Integer, List<Rechnung>> rechnungenProJahr = alleRechnungen.stream()
    .collect(Collectors.groupingBy(Rechnung::getJahr));
```

Das ist alles. Java übernimmt die komplette Initialisierung und das sichere Befüllen der Listen im Hintergrund.

Es gibt noch einen Spezialfall der Gruppierung: das **Partitionieren (partitioningBy)**.

Wenn wir Daten nicht in viele verschiedene Kategorien (wie Jahre oder Abteilungen) aufteilen wollen, sondern strikt in exakt zwei Lager ("Gut" und "Schlecht", "Bezahlt" und "Unbezahlt"), nutzen wir partitioningBy. Diese Methode nimmt ein Predicate entgegen und liefert immer eine Map<Boolean, List<T>> zurück.

```
// Wir trennen die Spreu vom Weizen
Map<Boolean, List<Rechnung>> mahnungsPruefung = alleRechnungen.stream()
    .collect(Collectors.partitioningBy(Rechnung::isBezahlt));

List<Rechnung> allesGut = mahnungsPruefung.get(true);
List<Rechnung> offenePosten = mahnungsPruefung.get(false);
```

8.4 Eigene Sammler bauen: Der Collector-Vertrag

Die Klasse Collectors deckt 99 Prozent aller Anwendungsfälle im Alltag eines Java-Entwicklers ab. Sie können groupingBy sogar verschachteln, um beispielsweise nach Jahr und *dann* nach Monat zu gruppieren.

Doch was, wenn Sie eine hochspezifische Abnahmestation benötigen, die es so noch nicht gibt? Die Stream-API ist offen für Erweiterungen. Sie können das Interface Collector<T, A, R> selbst implementieren.

Ein eigener Collector ist ein Vertrag, der aus vier essenziellen Bausteinen (Funktionen) besteht:

1. **supplier():** Wie wird der interne Zwischenspeicher erzeugt? (z.B. ArrayList::new)
2. **accumulator():** Wie wird ein einzelnes Element in diesen Zwischenspeicher integriert? (z.B. List::add)
3. **combiner():** Wie werden zwei Zwischenspeicher zusammengeführt? (Dies ist zwingend nötig, wenn das Fließband mit parallelStream() in mehrere parallele Bänder aufgeteilt wird und die Teilergebnisse am Ende wieder vereint werden müssen).
4. **finisher():** Gibt es noch einen finalen Umwandlungsschritt, bevor das Ergebnis an den Aufrufer übergeben wird?

Für die meisten Projekte reicht es zu wissen, dass dieser Mechanismus existiert und Sie die Stream-API jederzeit an Ihre domänenspezifischen Datenstrukturen anpassen können. Die Methode Collector.of(...) erlaubt es Ihnen, diese vier Funktionen on-the-fly zusammenzustecken, ohne eine komplett neue Klasse schreiben zu müssen.

Fazit des zweiten Teils

Herzlichen Glückwunsch! Sie haben den Maschinenraum der Java-Stream-API vollständig durchquert. Sie haben gelernt, wie Sie Datenströme erschaffen (Quellen), wie Sie das Material bearbeiten und sortieren (Intermediäre Operationen) und wie Sie die Resultate sicher und elegant einsammeln (Terminale Operationen). Sie können nun komplexe Datenschleifen, die früher fehleranfällig und schwer lesbar waren, in flüssige, deklarative Pipelines verwandeln.

Doch ein elegantes Fließband bringt uns wenig, wenn die Daten, die wir hineinstecken, fehlerhaft oder unerwartet leer sind. In der realen Welt stürzen Programme ab, weil Werte fehlen oder Berechnungen fehlschlagen.

Im kommenden dritten Teil unseres Buches widmen wir uns daher dem funktionalen Design und der Sicherheit. Wir werden den größten Feind des Java-Entwicklers – die NullPointerException – ein für alle Mal besiegen.

Willkommen in Teil III: **Praxis, Muster, Perspektiven – Funktionales Design**.

Teil III:

Praxis, Muster, Perspektiven – Funktionales Design

9 Die Jagd nach der NullPointerException – Optional richtig nutzen

Wir haben unsere Datenfabrik in den letzten Kapiteln mit Hochleistungs-Fließbändern ausgestattet. Die Maschinen laufen, die Transformationen sind elegant und zustandslos. Doch es gibt einen Feind, der fähig ist, selbst die eleganteste funktionale Pipeline augenblicklich in die Luft zu jagen. Es ist ein Feind, den Sie alle aus schmerzhafter Erfahrung kennen: die NullPointerException.

In diesem dritten Hauptteil des Buches verlassen wir die reine Mechanik der Streams und widmen uns dem funktionalen Software-Design. Wir lernen, wie wir unsere Fabrik kugelsicher machen. Den Anfang macht der funktionale Umgang mit der Abwesenheit von Werten.

9.1 Die Katastrophe: Der Milliarden-Dollar-Fehler (null)

Im Jahr 1965 erfand der britische Informatiker Tony Hoare die Null-Referenz für die Sprache ALGOL W. Jahrzehnte später entschuldigte er sich öffentlich dafür und nannte es seinen "Milliarden-Dollar-Fehler".

Warum ist null so katastrophal? Weil es das Typsystem von Java untergräbt. Wenn eine Methode laut Signatur einen Kunde zurückgibt, vertraut der Compiler darauf, dass Sie Methoden wie getName() auf diesem Kunden aufrufen können. Wenn die Methode aber heimlich null zurückgibt, merkt der Compiler das nicht. Das Programm stürzt erst zur Laufzeit ab.

In der imperativen Welt haben wir uns daran gewöhnt, dieses Problem mit Defensiv-Code zu bekämpfen: endlose Kaskaden von if (objekt != null)-Abfragen.

In der funktionalen Welt, in der wir Methodenaufrufe zu langen Ketten verknüpfen (Method Chaining), ist null jedoch ein absoluter Blocker. Betrachten Sie diese klassische Kette:

```
// Was passiert, wenn getRechnungsAdresse() null zurückgibt? -> Bumm!
String stadt = kunde.getRechnungsAdresse().getStadt().toUpperCase();
```

Wenn wir diesen Code mit klassischen Null-Checks absichern wollen, zerstören wir jede Lesbarkeit und fallen tief in die imperative Welt zurück. Wir brauchen einen besseren Weg, um unserem Fließband mitzuteilen: *"Hier könnte vielleicht ein Wert fehlen."*

9.2 Der Optional-Container: Die funktionale Lösung

Java 8 brachte die Lösung in Form der Klasse java.util.Optional<T>.

Stellen Sie sich Optional wie eine standardisierte, gepanzerte Transportkiste in unserer Fabrik vor. Auf der Kiste steht ein Aufkleber, z. B. Optional<Kunde>. Wenn diese Kiste auf dem Fließband ankommt, wissen Sie eine Sache absolut sicher: Die Kiste selbst ist *niemals* null.

Wenn Sie die Kiste jedoch öffnen, gibt es genau zwei Möglichkeiten:

1. Darin liegt sicher verpackt ein Kunde-Objekt.
2. Die Kiste ist komplett leer (Optional.empty()).

Indem eine Methode ein Optional zurückgibt, zwingt sie den Aufrufer (also uns), sich mit dem Fall der Leere auseinanderzusetzen. Der Compiler lässt uns nicht mehr einfach getName() aufrufen. Wir müssen die Kiste zuerst behandeln.

Wie erzeugen wir diese Kisten? Java bietet uns dafür drei Fabrik-Methoden:

```
// 1. Eine Kiste, die garantiert voll ist (wirft Exception bei null!)
Optional<String> vollerName = Optional.of("Max Mustermann");

// 2. Eine garantiert leere Kiste
Optional<String> keinName = Optional.empty();

// 3. Der Lebensretter: Eine Kiste, die vielleicht voll, vielleicht leer ist
String eingabe = // ... kommt vielleicht aus einer unzuverlässigen Datenbank
Optional<String> sichereEingabe = Optional.ofNullable(eingabe);
```

9.3 Optional als Monade: map, filter und flatMap

Jetzt kommt der eigentliche Geniestreich. Viele Entwickler nutzen Optional anfangs nur als glorifizierten Null-Check: *Ist die Kiste voll? Dann packe sie aus. Wenn nicht, mache etwas anderes.* Das ist jedoch nicht funktional gedacht. Ein Optional ist eine sogenannte **Monade**. Ohne zu tief in die Kategorientheorie der Mathematik abzudriften, bedeutet das für uns: **Wir können auf einem Optional exakt dieselben funktionalen Operationen ausführen wie auf einem Stream!**

Stellen Sie sich ein Optional einfach als einen Stream vor, der maximal ein einziges Element enthalten kann (entweder 1 oder 0).

Lassen Sie uns das imperative Desaster von vorhin (Kunde -> Adresse -> Stadt) rein

funktional absichern. Nehmen wir an, unsere Getter geben nun Optional zurück, da ein Kunde vielleicht keine Rechnungsadresse hat.

```
Optional<Kunde> sichererKunde = holeKundeAusDatenbank(123);

Optional<String> stadt = sichererKunde
    .flatMap(Kunde::getRechnungsAdresse)
       // flatMap, weil der Getter ein Optional zurückgibt
    .map(Adresse::getStadt)
       // map, weil getStadt einen normalen String liefert
    .filter(s -> s.startsWith("B"))
       // Nur Städte mit 'B' (sonst wird das Optional leer!)
    .map(String::toUpperCase);
```

Analysieren wir die Eleganz dieses Codes:

Wir rufen flatMap und map direkt auf der Transportkiste auf! Wenn sichererKunde leer ist, schlägt der Aufruf nicht fehl. flatMap merkt: *"Aha, die Kiste ist leer. Ich mache gar nichts und reiche einfach eine leere Kiste weiter."* Wenn der Kunde existiert, aber keine Adresse hat, macht das zweite map nichts und reicht die leere Kiste weiter.

Das gesamte Null-Handling verschwindet komplett unter der Haube der Monade. Wir schreiben die "Happy Path"-Logik (den Idealfall), und das Optional fängt jegliche Abwesenheit von Werten sicher und unsichtbar für uns ab. Keine einzige NullPointerException kann hier auftreten.

9.4 Das berüchtigte Anti-Pattern: get() ohne vorherige Prüfung

Mit einem Werkzeug wie Optional kann man jedoch auch viel falsch machen. Das größte Anti-Pattern ist der Versuch, die Kiste mit Gewalt aufzubrechen.

Die Klasse Optional besitzt eine Methode namens get(). Sie liefert den Wert aus der Kiste. Wenn die Kiste aber leer ist, wirft get() eine NoSuchElementException. Wenn Sie .get() aufrufen, haben Sie die NullPointerException im Grunde nur umbenannt. Sie haben nichts gewonnen!

```
// ANTI-PATTERN: Nutzen Sie get() nicht auf diese Weise!
Optional<Kunde> kundeOpt = holeKunde();
if (kundeOpt.isPresent()) {
    System.out.println(kundeOpt.get().getName());
} else {
    System.out.println("Unbekannt");
}
```

Das ist imperativer Code im funktionalen Schafspelz. Wir haben die Kiste manuell geprüft (isPresent) und dann gewaltsam geöffnet (get).

Wie machen wir es richtig? Optional bietet uns wunderbare terminale Methoden an, um die Kiste elegant auszupacken und direkt Standardwerte (Fallbacks) zu definieren, falls sie leer sein sollte:

- **orElse(T other):** Liefere den Wert, oder einen Standardwert.
- **orElseGet(Supplier<? extends T> supplier):** Liefere den Wert, oder berechne den Standardwert funktional (Lazy Evaluation – wichtig, wenn der Standardwert "teuer" zu berechnen ist!).
- **orElseThrow(Supplier<? extends X> exceptionSupplier):** Liefere den Wert, oder wirf eine von mir definierte, fachliche Exception.
- **ifPresent(Consumer<? super T> action):** Wenn der Wert da ist, führe einen Seiteneffekt aus (z.B. Loggen), sonst tue gar nichts.

Hier ist die korrekte, voll funktionsfähige Lösung für unser Beispiel:

```
// BEST PRACTICE: Deklaratives Auspacken mit Fallback
Optional<Kunde> kundeOpt = holeKunde();

// Variante A: Einen Standardnamen liefern
String name = kundeOpt.map(Kunde::getName).orElse("Unbekannter Kunde");

// Variante B: Nur etwas tun, wenn der Kunde existiert
kundeOpt.map(Kunde::getName).ifPresent(System.out::println);

// Variante C: Fachliche Exception werfen, wenn der Kunde zwingend nötig ist
Kunde pflichtKunde = kundeOpt.orElseThrow(
          () -> new KundeNichtGefundenException("ID existiert nicht!"));
```

Durch den Einsatz von Optional zwingen wir unsere Datenfabrik dazu, Ausnahmefälle in den normalen Datenfluss zu integrieren. Es gibt keine versteckten null-Rückgaben mehr, die nachts um drei Uhr in der Produktion einen Absturz verursachen. Die Signaturen unserer Methoden dokumentieren sich nun von selbst: *Achtung, hier könnte etwas fehlen!*

Im nächsten Kapitel gehen wir noch einen Schritt weiter. Wir entziehen unseren Datenstrukturen die Fähigkeit, sich nach ihrer Erschaffung jemals wieder zu verändern. Wir tauchen ein in die wichtigste Best-Practice der modernen Architektur: **Unveränderliche Datenstrukturen (Immutability)**.

10 Unveränderliche Datenstrukturen (Immutability)

Wir haben in den bisherigen Kapiteln unsere Datenfabrik aufgebaut, elegante Fließbänder (Streams) konstruiert und im letzten Kapitel unsere Pipelines gegen das Fehlen von Werten (Optional) abgesichert. Doch es gibt noch eine gigantische Schwachstelle in unserem System.

Stellen Sie sich vor, ein Bauteil rollt auf unserem Fließband von einer Station zur nächsten. Während Station B das Bauteil gerade vermisst und prüft, greift plötzlich ein Arbeiter von Station A heimlich über das Band und verbiegt das Bauteil nachträglich. Das Ergebnis? Station B berechnet völlig falsche Werte, die Produktion steht still, und niemand kann den Fehler reproduzieren.

In der Softwareentwicklung nennen wir dieses Szenario "Shared Mutable State" (geteilter, veränderbarer Zustand). Es ist die Hauptursache für unerklärliche Bugs, insbesondere in nebenläufigen (parallelen) Systemen. In diesem Kapitel lernen wir die mächtigste Verteidigungslinie der funktionalen Programmierung kennen: **Immutability (Unveränderlichkeit)**.

10.1 Die absolute Best Practice: Immutability

Die Regel der Unveränderlichkeit ist bestechend einfach: **Sobald ein Objekt vollständig erschaffen wurde, darf sich sein Zustand niemals wieder ändern.**

Wenn Sie einen Kunden mit dem Namen "Müller" und der Adresse "Berlin" instanziieren, dann bleiben diese Daten bis zur Zerstörung des Objekts durch den Garbage Collector exakt so bestehen. Es gibt keine Setter-Methoden (setName(...)). Es gibt keine Methoden, die interne Listen modifizieren.

Warum ist das so wichtig?

Wie Sie vielleicht wissen, entstehen Race Conditions (Wettlaufsituationen) immer dann, wenn zwei Threads gleichzeitig auf dieselben Daten zugreifen und *mindestens einer davon* diese Daten verändert. Wenn Daten jedoch unveränderlich (immutable) sind, können tausende Threads gleichzeitig lesend auf dasselbe Objekt zugreifen, ohne dass wir komplexe und langsame synchronized-Blöcke oder Locks benötigen. Unveränderliche Objekte sind von Natur aus absolut Thread-safe.

Zudem machen sie unseren Code extrem vorhersehbar (Referenzielle Transparenz). Wenn Sie ein unveränderliches Objekt an eine fremde Methode übergeben, haben Sie

die absolute Garantie, dass diese Methode Ihr Objekt nicht heimlich manipuliert.

10.2 Warum das final-Schlüsselwort nicht ausreicht

Wenn Java-Entwickler das erste Mal von Immutability hören, greifen sie instinktiv zu einem bekannten Werkzeug: dem Schlüsselwort final. Doch final bietet nur eine trügerische Sicherheit.

Betrachten Sie diesen Code:

```
public class Bestellung {
    private final List<Artikel> artikelListe;

    public Bestellung() {
        this.artikelListe = new ArrayList<>();
    }

    public List<Artikel> getArtikelListe() {
        return this.artikelListe;
    }
}
```

Das Feld artikelListe ist als final deklariert. Das bedeutet lediglich, dass wir die *Referenz* nicht mehr ändern dürfen. Wir können also innerhalb der Klasse nicht plötzlich this.artikelListe = new LinkedList<>() aufrufen. Das schützt die Referenz, aber **nicht den Inhalt des Objekts**!

Jeder, der die Methode getArtikelListe() aufruft, erhält direkten Zugriff auf die interne ArrayList. Ein Aufrufer kann nun fröhlich Elemente hinzufügen oder löschen:

```
Bestellung meineBestellung = new Bestellung();
// FATAL: Wir manipulieren den internen Zustand der Bestellung von außen!
meineBestellung.getArtikelListe().add(new Artikel("Betrug", 0.0));
```

Das Objekt Bestellung ist nicht immutable. Es ist hochgradig mutabel und anfällig für Seiteneffekte. final reicht nicht aus, um tiefe Unveränderlichkeit (Deep Immutability) zu garantieren.

10.3 Unveränderliche Collections (List.of, Map.copyOf)

Um das Problem aus dem vorherigen Abschnitt zu lösen, müssen wir die Datenstruktur selbst gegen Veränderungen absperren. Vor Java 9 war dies mühsam und erforderte Hilfskonstrukte wie Collections.unmodifiableList().

Seit Java 9 bietet uns die Sprache brillante, hochperformante Fabrik-Methoden (Factory Methods), um echte, unveränderliche Collections direkt zu erzeugen.

```
// Erzeugt eine unveränderliche Liste
List<String> farben = List.of("Rot", "Grün", "Blau");

// Erzeugt ein unveränderliches Set
Set<Integer> primzahlen = Set.of(2, 3, 5, 7, 11);

// Erzeugt eine unveränderliche Map
Map<Integer, String> fehlerCodes = Map.of(
    404, "Not Found",
    500, "Internal Server Error"
);
```

Wenn Sie nun versuchen, auf diesen Collections mutierende Methoden aufzurufen (wie farben.add("Gelb") oder farben.clear()), wirft Java sofort zur Laufzeit eine UnsupportedOperationException.

Noch wichtiger für unsere Fabrik ist das sichere Kopieren. Wenn Sie eine unsichere, mutierbare Liste als Eingabe erhalten, können Sie diese mit List.copyOf() sofort in eine unveränderliche Festung verwandeln:

```
public class Bestellung {
    private final List<Artikel> artikelListe;

    // Wir akzeptieren eine mutierbare Liste von außen...
    public Bestellung(List<Artikel> unsichereListe) {
        // ... aber wir speichern eine tiefe, unveränderliche Kopie!
        this.artikelListe = List.copyOf(unsichereListe);
    }

    public List<Artikel> getArtikelListe() {
        return this.artikelListe;
           // Gefahrlos, da List.copyOf() immutable ist!
    }
}
```

10.4 Die Revolution der Datenmodellierung: Java Records

Wir haben nun unveränderliche Listen. Aber was ist mit unseren eigenen fachlichen Klassen? Den Rechnungen, Kunden und Artikeln?

In der klassischen Java-Welt war das Erstellen einer wirklich unveränderlichen Klasse (eines sogenannten POJOs - Plain Old Java Object) eine Qual. Sie mussten alle Felder

private final deklarieren, einen riesigen Konstruktor schreiben, dutzende Getter generieren und die Methoden equals(), hashCode() und toString() überschreiben, um Wertevergleiche zu ermöglichen. Für eine einfache Klasse mit drei Eigenschaften produzierten Sie schnell 60 Zeilen Boilerplate-Code.

Mit Java 14 (als Preview) und endgültig in Java 16 hat Oracle den größten Schritt in Richtung funktionaler Datenmodellierung vollzogen: **Die Einführung von Records.**

Ein Record ist die Deklaration der reinen Absicht, Daten unveränderlich zu transportieren. Schauen wir uns an, wie wir einen kompletten Kunden als Record definieren:

```
public record Kunde(String id, String name, String email) {
    // Keine Felder, keine Getter, kein Konstruktor nötig!
}
```

Das ist alles. Diese einzige Zeile Code generiert im Hintergrund (durch den Compiler) eine vollwertige, perfekt auf Immutability abgestimmte Klasse.

- Alle Eigenschaften (id, name, email) sind implizit private final.
- Ein kanonischer Konstruktor, der alle Felder entgegennimmt, ist vorhanden.
- Lesende Zugriffsmethoden (die absichtlich nicht get... heißen, sondern wie das Feld selbst, z. B. name()) werden generiert.
- equals() und hashCode() werden so generiert, dass zwei Records genau dann gleich sind, wenn *all ihre Werte* gleich sind (Werte-Semantik), nicht wenn sie dieselbe Speicheradresse haben.

So verwenden wir unser neues, funktionales Datenpaket:

```
Kunde k1 = new Kunde("123", "Anna", "anna@mail.com");
Kunde k2 = new Kunde("123", "Anna", "anna@mail.com");

System.out.println(k1.name()); // Ausgabe: Anna

// Records haben eine korrekte Werte-Semantik
System.out.println(k1.equals(k2)); // Ausgabe: true!
```

Achtung, Falle (Shallow Immutability):

Ein Record ist nur flach unveränderlich (shallow immutable). Er garantiert, dass die *Referenzen* der Felder sich nicht ändern. Wenn Sie jedoch eine mutierbare ArrayList in einen Record packen, kann der Inhalt dieser Liste weiterhin manipuliert werden. **Die**

goldene Regel lautet daher: Wenn Sie Records verwenden, dürfen alle darin enthaltenen Listen und Maps ausschließlich mit List.copyOf() oder List.of() erzeugt werden!

10.5 Zustandsänderung durch Kopien (Wither-Methoden)

Eine Frage drängt sich nun unweigerlich auf: *Wenn absolut alles unveränderlich ist, wie kann ich dann in meinem Programm überhaupt noch fachliche Prozesse abbilden? Was passiert, wenn Frau Anna heiratet und sich ihr Nachname ändert?*

In der imperativen Welt rufen wir k1.setName("Müller") auf.

In der funktionalen Datenfabrik ist dieses Verbiegen des Bauteils strengstens verboten. Wenn wir einen veränderten Zustand benötigen, **bauen wir ein neues Objekt auf Basis des alten Objekts.** Dieses Pattern nennt man "Wither-Methoden" (in Anlehnung an Getter und Setter, nur eben mit dem Präfix with...). Ein Record kann problemlos um eigene Methoden erweitert werden.

```
public record Kunde(String id, String name, String email) {

    // Die Wither-Methode: Nimmt den neuen Wert und liefert eine Kopie
    public Kunde withName(String neuerName) {
        return new Kunde(this.id, neuerName, this.email);
    }

    public Kunde withEmail(String neueEmail) {
        return new Kunde(this.id, this.name, neueEmail);
    }
}
```

Wenn Annas Name sich nun ändert, rufen wir die Wither-Methode auf:

```
Kunde annaLedig = new Kunde("123", "Anna Schmidt", "anna@mail.com");

// Wir erzeugen einen komplett neuen Zustand (Kopie)
Kunde annaVerheiratet = annaLedig.withName("Anna Müller");

System.out.println(annaLedig.name());       // Immer noch "Anna Schmidt"
System.out.println(annaVerheiratet.name()); // "Anna Müller"
```

Das ursprüngliche Objekt annaLedig bleibt absolut intakt. Jeder Thread, der gerade auf dieses alte Objekt zugreift (z. B. ein Thread, der gerade die alte Rechnung als PDF generiert), kann gefahrlos weiterarbeiten. Der neue Zustand annaVerheiratet existiert völlig unabhängig davon.

Das Erzeugen von Kopien klingt zunächst nach einer massiven Speicher- und CPU-Verschwendung. Dank der hochoptimierten Architektur der Java Virtual Machine (insbesondere der "Escape Analysis" des Just-in-Time-Compilers) und der Tatsache, dass sich kurzlebige, kleine Objekte in der "Eden Space" des Heaps extrem effizient abräumen lassen, ist der Overhead in der Praxis jedoch vernachlässigbar. Die gewonnene Sicherheit, Parallelisierbarkeit und Bug-Freiheit überwiegen diesen minimalen Preis um ein Vielfaches.

Mit reinen Funktionen, der mächtigen Stream-API, sicheren Optional-Rückgaben und unveränderlichen Records haben wir den Kern unserer funktionalen Architektur fertiggestellt.

Doch was passiert, wenn auf unserem Fließband etwas katastrophal schiefgeht? Wenn die Datenbank nicht erreichbar ist oder eine Datei nicht gelesen werden kann? Im nächsten Kapitel werden wir sehen, wie die klassische Fehlerbehandlung mittels Exceptions unsere eleganten Pipelines zerstören kann – und wie wir **Fehlerbehandlung auf die feine Art** betreiben.

11 Fehlerbehandlung auf die feine Art

Unsere Datenfabrik ist mittlerweile ein architektonisches Meisterwerk. Die Fließbänder laufen dank der Stream-API reibungslos, Optional bewacht die Pipelines vor leeren Kisten, und unsere Records garantieren, dass niemand die Bauteile während der Fahrt heimlich verbiegt.

Doch wir haben bisher in einer idealen Welt programmiert: dem sogenannten "Happy Path". In der Realität ist die Welt chaotisch. Datenbanken sind plötzlich nicht erreichbar, Netzwerkverbindungen brechen ab, und Dateien, die wir einlesen wollen, existieren nicht. In der Java-Welt signalisieren wir solche Ausnahmezustände traditionell mit **Exceptions**.

In diesem Kapitel werden wir sehen, warum Javas klassische Fehlerbehandlung und funktionale Fließbänder natürliche Feinde sind – und wie wir sie dennoch elegant miteinander versöhnen können.

11.1 Das Problem: Checked Exceptions brechen den Stream-Fluss

Javas Typsystem unterscheidet zwischen zwei Arten von Ausnahmen: den unkontrollierten (Unchecked Exceptions wie die NullPointerException) und den kontrollierten (Checked Exceptions wie die IOException). Bei Letzteren zwingt uns der Compiler, sie entweder mit einem try-catch-Block abzufangen oder sie in der Methodensignatur mit throws weiterzureichen.

Stellen Sie sich vor, wir haben eine Liste von Dateipfaden auf unserem Fließband. Unsere Aufgabe: *Lies den Inhalt aller Dateien ein und sammle ihn in einer Liste.*

Wir haben dafür eine nützliche Methode leseDatei:

```
public String leseDatei(String pfad) throws IOException {
    return Files.readString(Path.of(pfad));
}
```

Nun versuchen wir, diese Methode in unsere elegante Stream-Pipeline einzubauen:

```
List<String> dateiPfade = Arrays.asList("config.txt", "daten.csv",
"geheim.txt");

// KOMPILIERFEHLER: Unhandled exception: java.io.IOException
List<String> inhalte = dateiPfade.stream()
    .map(pfad -> leseDatei(pfad))
    .toList();
```

Warum streikt der Compiler hier?

Erinnern Sie sich an Kapitel 5. Die Methode map erwartet eine Function<T, R>. Der Vertrag (das Interface) für Function definiert die Methode exakt so: R apply(T t);.

Sehen Sie dort ein throws Exception? Nein!

Das bedeutet, das Fließband von Java akzeptiert strikt *keine* Maschinen, die Checked Exceptions werfen können. Eine Funktion in der funktionalen Programmierung muss rein sein – das Werfen einer Exception ist jedoch ein massiver, unvorhersehbarer Seiteneffekt, der den Kontrollfluss komplett sprengt.

11.2 Workarounds: Unchecked Exceptions und Wrapper-Methoden

Der erste Reflex vieler Entwickler ist es, das Problem mit roher Gewalt zu lösen. Wenn das Interface keine Checked Exceptions erlaubt, fangen wir die Exception direkt im Lambda ab und verpacken sie in eine Unchecked Exception (z. B. RuntimeException), die der Compiler ignoriert.

```
List<String> inhalte = dateiPfade.stream()
    .map(pfad -> {
        try {
            return leseDatei(pfad);
        } catch (IOException e) {
            // Wir verstecken den Fehler vor dem Compiler!
            throw new RuntimeException("Fehler beim Lesen von: " + pfad, e);
        }
    })
    .toList();
```

Dieser Code kompiliert. Aber haben wir das Problem wirklich gelöst?

Stellen Sie sich vor, auf unserem Fließband liegen 10.000 Dateien. Die ersten 5.000 werden erfolgreich eingelesen. Bei Datei 5.001 tritt ein Netzwerkfehler auf, und unsere RuntimeException wird geworfen.

Was passiert mit unserem Fließband? **Es hält augenblicklich mit einem lauten Knall an.** Die gesamte Anwendung stürzt (im schlimmsten Fall) ab. Die 5.000 bereits gelesenen Dateien sind unwiederbringlich verloren, und die restlichen 4.999 Dateien werden nicht einmal mehr angeschaut.

Das Werfen von Exceptions bricht die funktionale Pipeline. Es ist das Äquivalent dazu, den Not-Aus-Schalter der gesamten Fabrik zu drücken, nur weil ein einziges Bauteil fehlerhaft war.

11.3 Die professionelle Lösung: Das Either-Pattern implementieren

In einer echten funktionalen Architektur drücken wir nicht den Not-Aus-Schalter. Wir wollen, dass das Fließband bis zum Ende weiterläuft! Wenn eine Maschine ein Bauteil nicht verarbeiten kann, soll sie das kaputte Bauteil markieren und einen Fehlerbericht daranhängen. Am Ende des Fließbands sortieren wir dann sauber: Die erfolgreichen Produkte kommen in Kiste A, die Fehlerberichte in Kiste B.

Um das zu erreichen, nutzen wir ein Konzept, das in reinen funktionalen Sprachen fest verankert ist: **Die Either-Monade** (teilweise auch Try genannt).

Either bedeutet übersetzt "Entweder". Es ist ein Container (ähnlich wie Optional), der genau zwei Zustände annehmen kann: *Entweder* er enthält einen erfolgreichen Wert (oft Right genannt), *oder* er enthält einen Fehler (oft Left genannt).

Da Java ein solches Konstrukt standardmäßig (noch) nicht mitbringt, bauen wir uns eine simple, funktionale Version dieses Containers für unsere Fehlerbehandlung selbst:

```
// Eine simple Implementierung des Either-Patterns
public class Try<T> {
    private final T wert;
    private final Exception fehler;

    private Try(T wert, Exception fehler) {
        this.wert = wert;
        this.fehler = fehler;
    }

    // Fabrik-Methode für den Erfolgsfall
    public static <T> Try<T> success(T wert) {
        return new Try<>(wert, null);
    }

    // Fabrik-Methode für den Fehlerfall
    public static <T> Try<T> failure(Exception fehler) {
        return new Try<>(null, fehler);
    }

    public boolean isSuccess() { return fehler == null; }
    public boolean isFailure() { return fehler != null; }
```

```
    public T getWert() { return wert; }
    public Exception getFehler() { return fehler; }
}
```

Jetzt schreiben wir eine kleine Hilfsmethode (eine Wrapper-Maschine), die unsere fehleranfällige leseDatei-Methode aufruft, aber *niemals* eine Exception wirft. Stattdessen gibt sie immer einen Try-Container zurück.

```
public Try<String> leseDateiSicher(String pfad) {
    try {
        String inhalt = leseDatei(pfad);
        return Try.success(inhalt); // Alles gutgegangen!
    } catch (IOException e) {
        return Try.failure(e);       // Fehler sicher verpackt!
    }
}
```

Das Fließband neu starten:

Lassen Sie uns nun unser Stream-Fließband mit diesem neuen, funktionalen Werkzeug aufbauen.

```
List<String> dateiPfade = Arrays.asList("config.txt", "daten.csv",
                                        "geheim.txt");

// 1. Wir mappen die Pfade in unsere sicheren Try-Container
List<Try<String>> alleErgebnisse = dateiPfade.stream()
    .map(pfad -> leseDateiSicher(pfad))
    .toList();
```

Der Stream läuft nun zu 100 % sauber durch, egal wie viele Dateien fehlen oder kaputt sind. Keine Abstürze, keine Unterbrechungen. In der Liste alleErgebnisse liegen nun unsere Container.

Im letzten Schritt können wir die Ergebnisse wunderbar separieren. Erinnern Sie sich an Collectors.partitioningBy() aus Kapitel 8? Das ist das perfekte Werkzeug dafür:

```
// 2. Wir trennen Erfolge und Fehler
Map<Boolean, List<Try<String>>> getrennt = alleErgebnisse.stream()
    .collect(Collectors.partitioningBy(Try::isSuccess));

// 3. Die erfolgreichen Werte extrahieren
List<String> erfolgreicheInhalte = getrennt.get(true).stream()
    .map(Try::getWert)
    .toList();
```

```
// 4. Die Fehler professionell loggen
getrennt.get(false).stream()
    .map(Try::getFehler)
    .forEach(fehler -> System.err.println("Fehler aufgetreten: " +
fehler.getMessage()));
```

Fazit zur Fehlerbehandlung

Mit dem Either- bzw. Try-Pattern haben wir das letzte große Hindernis auf unserem Weg zur funktionalen Meisterschaft aus dem Weg geräumt. Ausnahmen (Exceptions) sind nun keine unvorhersehbaren Bomben mehr, die unsere Architektur bedrohen. Sie sind zu einfachen, berechenbaren Datenobjekten geworden, die ganz normal über unser Fließband rollen und am Ende sortiert werden.

Dieses Prinzip – Seiteneffekte und Fehler als reine Daten zu modellieren – ist die Königsdisziplin der funktionalen Architektur. Im nächsten Kapitel heben wir unseren Blick vom einzelnen Stream auf das gesamte System. Wir betrachten, wie man eine komplette Softwarearchitektur nach diesen Idealen aufbaut: **Architektur: Functional Core, Imperative Shell**.

12 Architektur: Functional Core, Imperative Shell

In den bisherigen Kapiteln haben wir unsere Werkzeuge geschärft. Wir beherrschen die Syntax von Lambdas, wir können Datenströme durch hochkomplexe Pipelines lenken, wir sichern uns gegen das Fehlen von Werten mit Optional ab und wir wissen, wie wir Fehler als reine Datenobjekte behandeln.

Doch wenn wir aus diesen mikroskopischen Bausteinen herauszoomen und auf das große Ganze blicken – die Architektur unserer gesamten Anwendung –, stehen wir vor einem scheinbaren Widerspruch.

12.1 Die physikalischen Grenzen: Warum rein funktional nicht immer geht

Die oberste Regel der funktionalen Programmierung, die wir in Kapitel 2 aufgestellt haben, lautet: *Keine Seiteneffekte*. Eine reine Funktion (Pure Function) nimmt Daten entgegen und gibt neue Daten zurück. Sie verändert nichts an der Außenwelt.

Wenn wir diese Regel nun auf unsere gesamte Anwendung anwenden, bauen wir ein Programm, das beim Start eine Eingabe entgegennimmt, hochkomplexe und perfekte Berechnungen im Arbeitsspeicher durchführt, ein Ergebnis produziert ... und sich dann beendet. Es schreibt nichts in eine Datenbank. Es sendet keine HTTP-Antwort an den Browser. Es druckt keine Rechnung aus. Ein solches Programm ist völlig nutzlos. Es ist im Grunde nur eine sehr teure Heizung für Ihren Prozessor.

Software existiert *nur* wegen ihrer Seiteneffekte! Wir wollen, dass am Ende ein Datensatz in der PostgreSQL-Datenbank landet oder eine E-Mail verschickt wird. Wie lösen wir diesen Widerspruch?

12.2 IO, Datenbanken und UI an die Systemgrenzen schieben

Die Lösung für dieses Dilemma ist eines der elegantesten Architekturmuster der modernen Softwareentwicklung: **Functional Core, Imperative Shell** (Funktionaler Kern, imperative Schale). Das Konzept wurde maßgeblich von Gary Bernhardt geprägt und lässt sich wunderbar mit Prinzipien der Hexagonalen Architektur (Ports and Adapters) kombinieren.

Die Idee ist verblüffend einfach: Wir unterteilen unsere Anwendung in zwei strikt getrennte Welten.

1. **Der funktionale Kern (Functional Core):** Das ist das Herz unserer Datenfabrik.

Hier lebt unsere gesamte Geschäftslogik. Dieser Bereich besteht *ausschließlich* aus unveränderlichen Objekten (Java Records) und reinen Funktionen. Er kennt keine Datenbank, kein Netzwerk, keine Spring-Boot-Controller und keine Dateien. Er liest nichts und er schreibt nichts. Er berechnet nur. Er ist zu 100 % deterministisch und ohne jegliche Mocks testbar.

2. **Die imperative Schale (Imperative Shell):** Das ist die Laderampe unserer Fabrik. Hier ist es schmutzig. Hier passieren alle I/O-Operationen (Input/Output). Die Schale kommuniziert mit der REST-API, holt Daten aus der Datenbank, fängt Netzwerk-Exceptions ab und reicht die Rohdaten in den Kern. Wenn der Kern fertig ist, nimmt die Schale die berechneten Ergebnisse (z. B. eine neue Rechnungs-Kopie) und speichert sie imperativ in der Datenbank.

Betrachten wir ein Beispiel. Ein Kunde möchte einen Artikel kaufen.

Schlecht (Vermischung): Ein Service lädt den Kunden aus der DB, prüft den Kontostand, berechnet den Rabatt, ändert das Kunden-Objekt und ruft direkt repository.save(kunde) auf. Logik und Seiteneffekte sind untrennbar verschmolzen.

Gut (Functional Core, Imperative Shell): Die **Schale** lädt den Kunde (Record) aus der DB. Sie reicht den Kunden und den Artikel an den **Kern**. Der Kern prüft die Logik und liefert ein *neues* Objekt KundeMitNeuemKontostand zurück. Die **Schale** nimmt dieses neue Objekt und speichert es in der DB.

12.3 Pattern Matching und Sealed Classes (Make illegal states unrepresentable)

Wenn unser Kern nur aus reinen Funktionen und Daten (Records) besteht, müssen wir sicherstellen, dass diese Daten immer konsistent sind. Ein berühmtes Prinzip des funktionalen Designs lautet: **Make illegal states unrepresentable** (Mache illegale Zustände im Code unwiderruflich unmöglich).

In der imperativen Welt nutzen wir Validierungs-Logik (z. B. if (status == 1) ...), um Fehler abzufangen. In der funktionalen Welt nutzen wir das Typsystem von Java, um Fehler schon zur Compile-Zeit unmöglich zu machen.

Seit Java 17 haben wir dafür **Sealed Classes** (versiegelte Klassen) und in den darauffolgenden Versionen **Pattern Matching für switch**. Zusammen bilden sie sogenannte *Algebraische Datentypen (ADTs)*.

Nehmen wir den Status einer Bestellung. Eine Bestellung ist entweder Offen, Bezahlt (mit

einer Transaktions-ID) oder Storniert (mit einem Grund). In klassischem Java hätten wir eine Klasse Bestellung mit diversen Null-Feldern gebaut.

Mit Sealed Classes definieren wir exakt die erlaubten Varianten:

```
// Das Interface ist versiegelt. Nur die drei genannten Records dürfen es
// implementieren!
public sealed interface BestellStatus permits Offen, Bezahlt, Storniert {}

// Die konkreten, unveränderlichen Ausprägungen (Records)
public record Offen() implements BestellStatus {}
public record Bezahlt(String transaktionsId) implements BestellStatus {}
public record Storniert(String grund) implements BestellStatus {}
```

Das Geniale passiert nun in unserem funktionalen Kern, wenn wir Logik auf diesen Status anwenden. Wir nutzen das neue switch-Statement von Java als *Ausdruck (Expression)*, das einen Wert zurückgibt.

```
public String generiereKundenNachricht(BestellStatus status) {
    // Pattern Matching im switch-Ausdruck
    return switch (status) {
        case Offen o      -> "Ihre Bestellung wird bald bearbeitet.";
        case Bezahlt b    -> "Ihre Zahlung (ID: " + b.transaktionsId() +
                             ") ist eingegangen.";
        case Storniert s  -> "Bestellung storniert. Grund: " + s.grund();
    };
}
```

Warum ist das so revolutionär? **Exhaustiveness (Vollständigkeitsprüfung).** Der Java-Compiler weiß dank des sealed-Schlüsselworts, dass es *exakt* diese drei Zustände gibt und keinen weiteren. Es gibt keinen default-Zweig! Wenn morgen ein Kollege eine neue Variante public record Versendet(String trackingNummer) implements BestellStatus {} hinzufügt, wird der Compiler den obigen switch-Ausdruck augenblicklich als Fehler markieren, weil der Fall Versendet fehlt.

Wir haben einen illegalen Zustand (eine vergessene Status-Behandlung) unrepräsentierbar gemacht. Das ist funktionales Design in Perfektion.

12.4 Currying und Partial Application in Java

Zum Abschluss dieses Architektur-Kapitels werfen wir noch einen Blick auf ein fortgeschrittenes Konzept, das uns hilft, Abhängigkeiten (Dependencies) in unserem funktionalen Kern zu verwalten.

In Spring Boot nutzen wir oft "Dependency Injection", um z. B. einen SteuerRechner in unseren BestellService zu injizieren. Wenn wir aber reine Funktionen (z. B. als statische Methoden oder einfache Lambdas) schreiben, wie bekommen diese Funktionen ihre Abhängigkeiten?

Hier helfen Konzepte, die in Sprachen wie Haskell allgegenwärtig sind: **Currying** und **Partielle Applikation**.

Stellen Sie sich vor, wir haben eine Funktion zur Preisberechnung, die den Steuersatz und den Nettopreis benötigt.

```
BiFunction<Double, Double, Double> berechneBrutto =
    (steuer, netto) -> netto * (1.0 + steuer);
```

In einer funktionalen Pipeline (z. B. .map()) haben wir aber nur den netto-Wert auf dem Fließband. Der Steuersatz (z. B. 0.19) ist quasi die "Abhängigkeit" oder Konfiguration unserer Fabrikmaschine.

Beim *Currying* zerlegen wir eine Funktion, die zwei Parameter erwartet, in eine Funktion, die *einen* Parameter erwartet und eine *neue Funktion* zurückgibt!

```
// Currying: Eine Funktion, die den Steuersatz nimmt und eine neue Funktion
für das Fließband liefert
Function<Double, Function<Double, Double>> konfigurierteSteuer =
    steuer -> (netto -> netto * (1.0 + steuer));
```

Jetzt können wir unsere Maschine vorab konfigurieren (partielle Applikation):

```
// Wir "injizieren" die Abhängigkeit (den Steuersatz von 19%)
Function<Double, Double> deutscheSteuerMaschine =
    konfigurierteSteuer.apply(0.19);

List<Double> nettoPreise = Arrays.asList(10.0, 50.0, 100.0);

// Die vorkonfigurierte Maschine wird im Fließband genutzt
List<Double> bruttoPreise = nettoPreise.stream()
    .map(deutscheSteuerMaschine) // Rechnet jetzt automatisch mit 0.19!
    .toList();
```

Mit dieser Technik können Sie Konfigurationen (wie Datenbank-Verbindungen, API-Keys oder statische Steuersätze) in der imperativen Schale laden, Ihre Funktionen damit "vorladen" (partiell applizieren) und diese vorkonfigurierten, aber weiterhin reinen Funktionen in den funktionalen Kern schicken.

Fazit des dritten Teils

Wir haben in diesem dritten Hauptteil den Sprung vom reinen Code-Schreiben zur echten Softwarearchitektur geschafft. Wir haben Null-Werte durch Optional eliminiert, Race Conditions durch Immutability besiegt, Exceptions durch Try/Either gezähmt und schließlich verstanden, wie wir Seiteneffekte mit der "Functional Core, Imperative Shell"-Architektur an die Systemgrenzen drängen.

Unsere theoretische Ausbildung ist hiermit fast abgeschlossen. Bevor wir jedoch unsere eigene Legacy-Applikation (den E-Commerce-Shop) in Teil IV systematisch umbauen, blicken wir im nächsten Kapitel noch einmal kurz über den Tellerrand der Standard-Java-Bibliothek hinaus.

Willkommen bei **Kapitel 13: Über den Tellerrand – Alternative Paradigmen und Bibliotheken**.

13 Über den Tellerrand – Alternative Paradigmen und Bibliotheken

Unsere Datenfabrik ist nun mit den modernsten Maschinen ausgestattet, die das Standard-Java-Ökosystem (die JDK) zu bieten hat. Wir haben die Stream-API gemeistert, Optional und Java Records in unsere Architektur integriert und sogar eigene funktionale Konstrukte wie die Try-Monade entworfen.

Doch so mächtig Java in den letzten Jahren auch geworden ist – es bleibt eine Sprache, die den Kompromiss zwischen Objektorientierung, Abwärtskompatibilität und funktionaler Programmierung balancieren muss. In diesem Kapitel werfen wir einen Blick über die Mauern unserer eigenen Fabrik. Wir betrachten Werkzeuge und Sprachen, die keine Kompromisse machen, und werfen einen exklusiven Blick in die Zukunft der Java Virtual Machine (JVM).

13.1 Vavr (früher Javaslang): Persistente Datenstrukturen für Java

Erinnern Sie sich an Kapitel 10, als wir über Unveränderlichkeit (Immutability) sprachen? Wir haben gelernt, dass wir Zustandsänderungen durch das Erstellen von Kopien abbilden müssen (Wither-Methoden). Wenn wir eine List.copyOf() verwenden und ein Element "hinzufügen" wollen, müssen wir intern ein komplett neues Array erzeugen und alle alten Elemente herüberkopieren.

Bei kleinen Listen von 100 Elementen lacht die JVM darüber. Doch was passiert, wenn Ihre Liste zehn Millionen Einträge hat und Sie in einem funktionalen Algorithmus tausende Male ein Element am Ende anfügen? Der Speicherbedarf und die CPU-Last für das ständige Kopieren würden explodieren.

Hier betritt **Vavr**[1] die Bühne. Vavr (früher bekannt als Javaslang) ist die populärste funktionale Bibliothek für Java. Ihr größtes Geschenk an uns Entwickler sind **persistente Datenstrukturen**.

Eine persistente Liste in Vavr kopiert beim Hinzufügen eines Elements *nicht* die gesamte Liste. Stattdessen nutzt sie ein Konzept namens "Structural Sharing" (strukturelles Teilen), meist implementiert durch spezielle Baumstrukturen (Tries).

Die neue Liste enthält das neue Element und einen simplen Zeiger auf den Rest der alten Liste. Da die alte Liste absolut unveränderlich ist, ist das Teilen von Speicheradressen

[1] https://github.com/vavr-io/vavr

völlig gefahrlos!

Lassen Sie uns sehen, wie sich Vavr in unserem Code anfühlt. Die Bibliothek bringt völlig eigene Interfaces für Collections mit (io.vavr.collection.List statt java.util.List):

```
import io.vavr.collection.List;
import io.vavr.Tuple;
import io.vavr.Tuple2;

public class VavrBeispiel {
    public void demonstriereVavr() {
        // 1. Persistente Listen
        List<Integer> alteListe = List.of(1, 2, 3);

        // Es wird fast nichts kopiert, nur Zeiger verbogen!
        List<Integer> neueListe = alteListe.prepend(0).append(4);

        System.out.println(alteListe); // Ausgabe: List(1, 2, 3) - Unberührt!
        System.out.println(neueListe); // Ausgabe: List(0, 1, 2, 3, 4)

        // 2. Tupel (Daten ohne eigene Klassen verpacken)
        // Ideal, wenn eine Funktion zwei Werte zurückgeben muss
        Tuple2<String, Integer> kundeMitAlter = Tuple.of("Anna", 28);
        String name = kundeMitAlter._1;
        Integer alter = kundeMitAlter._2;
    }
}
```

Darüber hinaus bringt Vavr all die Dinge out-of-the-box mit, die wir in den letzten Kapiteln mühsam selbst gebaut haben: Ein voll ausgereiftes Try für die Ausnahmebehandlung, ein Either für Architektur-Grenzen und extrem mächtiges Pattern Matching, das selbst über die Fähigkeiten von Java 21 hinausgeht. Wer professionell und kompromisslos funktional in Java arbeiten will, kommt an Vavr kaum vorbei.

13.2 Ein Blick auf funktionale JVM-Sprachen: Scala und Clojure

Java ist nicht die einzige Sprache, die auf der Java Virtual Machine läuft. Die JVM ist im Grunde nur ein Motor, und Sie können verschiedene Karosserien (Sprachen) auf diesen Motor setzen. Zwei dieser Sprachen haben die funktionale Programmierung im Java-Ökosystem massiv geprägt.

Scala:

Scala (Scalable Language) wurde 2004 veröffentlicht und hatte ein klares Ziel:

Objektorientierung und funktionale Programmierung perfekt zu verschmelzen. Scala war der Pionier, der bewies, dass die JVM bereit für funktionale Konzepte ist.

In Scala ist *alles* ein Ausdruck (Expression). Es gibt kein void, und selbst ein einfaches if-else gibt immer einen Wert zurück. Scala führte "Case Classes" ein – das direkte Vorbild für unsere heutigen Java Records. Auch die Stream-API von Java ist tief von Scalas mächtigen Collection-Bibliotheken inspiriert. Wenn Sie Scala lernen, werden Sie verstehen, wohin sich Java in den nächsten fünf bis zehn Jahren entwickeln wird.

Clojure:

Clojure geht einen radikaleren Weg. Es ist ein Dialekt der altehrwürdigen Sprache Lisp. Wenn Sie Clojure-Code sehen, werden Sie im ersten Moment erschrecken: Er besteht gefühlt nur aus Klammern. Es gibt keine Klassen, keine klassischen Objekte und keine statische Typisierung.

Clojure ist zu 100 % funktional. Der veränderliche Zustand wird nicht nur vermieden, er ist in der Sprache fast unmöglich zu erzeugen, es sei denn, man nutzt spezielle, streng kontrollierte Konstrukte (Software Transactional Memory), die Race Conditions auf Compiler-Ebene ausschließen. Clojure zwingt Sie dazu, ausschließlich in Datenflüssen und reinen Transformationen zu denken. Eine Woche Clojure zu programmieren, wird Ihren Java-Code nachhaltig sauberer machen.

13.3 Javas Antwort auf die Zukunft (Project Valhalla)

Wir beenden unseren theoretischen Ausflug mit einem Blick in die Zukunft von Java selbst.

Wir haben in diesem Buch gelernt, dass wir Daten als unveränderliche Werte (Records) modellieren sollten. In der aktuellen Java-Welt hat das jedoch einen physikalischen Haken: Jeder Record ist unter der Haube immer noch ein Objekt.

Jedes Objekt in Java liegt auf dem Heap-Speicher. Es besitzt einen sogenannten "Object Header" (der Metadaten und Locks für synchronized speichert), der aktuell mindestens 12 bis 16 Bytes verschwendet – selbst wenn Ihr Objekt nur einen einzigen int-Wert enthält. Wenn Sie eine Liste mit 1.000 unveränderlichen Punkt-Records record Punkt(int x, int y) haben, haben Sie 1.000 Speicheradressen (Pointer), die kreuz und quer über den Heap verteilt sind. Der Prozessor hasst das, weil er Daten nicht effizient in seine rasend schnellen L1/L2-Caches laden kann (schlechte "Data Locality").

Das ist der Preis, den wir aktuell für Javas alte "Alles ist ein Objekt"-Philosophie zahlen.

Das **Project Valhalla** (ein laufendes Langzeit-Entwicklungsprojekt von Oracle) wird dieses Problem endgültig lösen, indem es sogenannte **Value Objects** (Werte-Objekte) einführt.

Die Philosophie von Valhalla lautet: *"Codes like a class, works like an int"* (Programmiert sich wie eine Klasse, funktioniert wie ein primitiver Datentyp).

Wenn Valhalla in das Standard-Java integriert wird, können wir unsere funktionalen Records mit einem speziellen Schlüsselwort (z.B. value record) deklarieren. Java wird dann den kompletten Object Header entfernen. Diese Daten haben keine eigene Speicheridentität mehr (es gibt kein == mehr für Speicheradressen, nur noch echtes Werte-Muster).

Das Fließband unserer Stream-API kann diese Value Objects dann exakt so effizient verarbeiten, als würden wir primitive Arrays (int[]) durch die CPU pumpen. Keine Cache-Misses mehr, kein Garbage-Collection-Overhead für Millionen von kleinen, funktionalen Zwischenobjekten. Project Valhalla wird die funktionale Programmierung in Java auf ein Performance-Level heben, das bisher C++ oder Rust vorbehalten war.

Fazit des dritten Teils

Wir haben es geschafft. Wir haben die Theorie, die Werkzeuge und die Architekturmuster der funktionalen Programmierung vollständig durchdrungen. Sie haben gesehen, wo Java heute steht, welche Bibliotheken wie Vavr uns helfen können und wie rosig die Zukunft durch Project Valhalla aussieht.

Doch Theorie allein baut keine Software. Es ist an der Zeit, die Werkzeuge in die Hand zu nehmen, die Ärmel hochzukrempeln und unsere eigene Fabrik zu errichten.

Wir lassen die abstrakten Beispiele hinter uns. Im vierten und letzten Teil dieses Buches nehmen wir uns ein realistisches, historisch gewachsenes Legacy-Projekt vor – einen klassischen E-Commerce-Shop voller Schleifen, Seiteneffekte und NullPointerExceptions – und refaktorieren ihn in sieben strikten Iterationen in eine makellose, funktionale Datenfabrik.

Willkommen im Finale. Willkommen in **Teil IV: Die Datenfabrik in der Praxis – Das E-Commerce-Projekt**.

Teil IV:

Die Datenfabrik in der Praxis – Das E-Commerce-Projekt

14 Überblick über das E-Commerce-Projekt

Herzlichen Glückwunsch! Sie haben die anspruchsvolle Theorie der funktionalen Programmierung gemeistert. Sie kennen die Syntax von Lambdas, die Macht der Stream-API, den Schutz durch unveränderliche Records und das funktionale Abfangen von Fehlern.

Doch wie in der Architektur oder im Handwerk gilt auch in der Softwareentwicklung: Die isolierte Betrachtung einzelner Werkzeuge macht noch keinen Meister. Der wahre Test Ihres Könnens besteht darin, diese Werkzeuge in einem großen, zusammenhängenden System harmonisch orchestrieren zu können.

In diesem vierten und letzten Teil unseres Buches verlassen wir die kleinen, sterilen Code-Snippets. Wir betreten die reale Welt.

14.1 Die Rückkehr in den Online-Shop

Unser Patient ist ein klassischer, historisch gewachsener E-Commerce-Shop – nennen wir ihn "J-Shop".

Das Backend dieses Shops wurde vor vielen Jahren geschrieben. Es ist ein Paradebeispiel für imperativen, zustandsbehafteten Code.

Wenn ein Kunde im aktuellen J-Shop eine Bestellung aufgibt, passiert Folgendes:

- Verschachtelte for-Schleifen iterieren mühsam über Artikelkataloge.
- Die Klasse Warenkorb ist voller Setter-Methoden, und von überall im System greifen fremde Klassen auf diesen Warenkorb zu, um heimlich Preise zu überschreiben oder Rabatte abzuziehen (Seiteneffekte).
- Die Methode zur Rechnungserstellung gibt regelmäßig null zurück, was an anderen Stellen zu den gefürchteten NullPointerExceptions führt.
- Fehler bei der Netzwerkanbindung an den Zahlungsanbieter werfen harte Checked Exceptions, die den gesamten Bestellprozess augenblicklich abstürzen lassen.

Kurzum: Der Code funktioniert, aber er ist unlesbar, schwer zu testen und absolut nicht bereit für eine parallele Verarbeitung auf modernen Mehrkern-Prozessoren.

Unsere Aufgabe ist klar: Wir werden diesen Spaghetti-Code nehmen und ihn schrittweise in eine makellose, funktionale Datenfabrik umbauen.

14.2 Die 7 Iterationen auf unserem Weg zur Perfektion

Wir werden den J-Shop nicht an einem einzigen Tag neu schreiben. Solche "Big Bang"-Refactorings enden in der Praxis meistens im Desaster. Stattdessen werden wir das System in sieben kontrollierten, aufeinander aufbauenden Iterationen (Kapiteln) modernisieren.

Damit Sie den roten Faden nicht verlieren, hier der Fahrplan für unsere Umbauarbeiten:

- **Iteration 1 – Der alte Webshop eröffnet (Kapitel 15):** Wir sichten den Bestand. Wir schauen uns den schlimmsten und fehleranfälligsten Code des aktuellen Systems an. Wir identifizieren die versteckten Zustandsänderungen und verstehen genau, *warum* dieser Code uns auf Dauer in den Ruin treiben wird.
- **Iteration 2 – Der Qualitätskontrolleur (Kapitel 16):** Wir beginnen mit dem Fundament. Wir reißen die alten, starren Strategie-Klassen (Interfaces mit anonymen inneren Klassen) ab und ersetzen sie durch leichtgewichtige Lambdas und Methodenreferenzen. Die Rabattregeln des Shops werden endlich als Funktionen übergeben.
- **Iteration 3 – Die Fließbänder laufen an (Kapitel 17):** Der große Umbau der Datenverarbeitung. Wir werfen alle for-Schleifen und manuellen Listen-Initialisierungen aus dem Code. Wir etablieren deklarative Pipelines mit der Stream-API, klopfen Kundenbestellungen mit flatMap flach und aggregieren Monatsumsätze mit reduce.
- **Iteration 4 – In Stein gemeißelt (Kapitel 18):** Wir frieren den Zustand ein. Wir wandeln unsere alten, mutierbaren Klassen (wie Artikel und Kunde) in moderne Java Records um. Wir entfernen alle Setter und implementieren das Wither-Pattern, um Warenkorb-Updates durch saubere Kopien abzubilden. Ab hier ist unser Shop zu 100 % Thread-safe.
- **Iteration 5 – Schutz vor der Leere (Kapitel 19):** Wir jagen die Milliarden-Dollar-Fehler. Wir refaktorieren alle Methoden, die potenziell keine Daten finden (z. B. die Suche nach einer Kundenadresse), so, dass sie Optional zurückgeben. Wir verketten diese Optionals elegant, sodass null-bedingte Abstürze der Vergangenheit angehören.
- **Iteration 6 – Fehler ohne Absturz (Kapitel 20):** Wir bauen die Sicherheitsnetze ein. Wenn der Rechnungs-Export fehlschlägt, darf der Stream nicht mehr abbrechen. Wir führen die funktionale Try/Either-Monade in den J-Shop ein und trennen am Ende des Fließbands saubere Erfolge von detaillierten Fehlerberichten.
- **Iteration 7 – Der moderne Kern (Kapitel 21):** Der architektonische Feinschliff. Wir nutzen Pattern Matching und Sealed Classes, um die Bestell-Zustände (Offen,

Bezahlt, Storniert) so abzusichern, dass der Compiler illegale Zustände gar nicht erst kompiliert ("Make illegal states unrepresentable"). Wir trennen reine Logik von Datenbank-Zugriffen.

14.3 Bereit für die Praxis?

Wenn Sie dieses Projekt am eigenen Rechner nachvollziehen möchten, lade ich Sie ein, Ihre bevorzugte Entwicklungsumgebung (IDE) zu öffnen. Sie benötigen mindestens Java 21, um alle modernen Features wie Records und das vollständige Pattern Matching für switch nutzen zu können.

Lesen Sie den Code der kommenden Kapitel nicht nur durch. Versuchen Sie, die imperativen Ausgangs-Szenarien gedanklich (oder physisch) selbst nachzubauen und dann den Refactoring-Schritt durchzuführen, bevor Sie meine funktionale Lösung betrachten. Der Moment, in dem ein 20-Zeilen-Block voller if-Abfragen vor Ihren Augen zu einem brillanten 3-Zeilen-Stream verschmilzt, ist der Moment, in dem Sie die funktionale Programmierung wirklich verinnerlichen.

Schalten Sie die Maschinen an. Öffnen wir die Türen des alten J-Shops für **Iteration 1**.

15 Iteration 1 – Der alte Webshop eröffnet (Imperativ & Mutable)

Willkommen im Maschinenraum des J-Shops. In dieser ersten Iteration werden wir noch keinen funktionalen Code schreiben. Bevor wir unser neues Fließband aufbauen können, müssen wir die alte Handwerkswerkstatt betreten, den Staub aufwirbeln und genau verstehen, warum die aktuellen Prozesse uns langfristig in den Ruin treiben werden.

Wir betrachten in diesem Kapitel den Kern unseres E-Commerce-Systems im aktuellen, historisch gewachsenen Zustand: rein imperativ, objektorientiert und hochgradig mutabel (veränderlich). Wir werden den Code sezieren und die versteckten architektonischen Zeitbomben identifizieren.

15.1 Der Flaschenhals: Verschachtelte for-Schleifen und if-Kaskaden

Jeder Online-Shop benötigt eine Suchfunktion. Im J-Shop existiert eine Klasse namens KatalogService, die dafür verantwortlich ist, Artikel nach bestimmten Kriterien zu filtern.

Die aktuelle Anforderung des Marketings lautet: *"Finde alle Elektronik-Artikel, die mehr als 100 Euro kosten, aktuell auf Lager sind, und gib uns eine Liste ihrer Namen, sortiert nach dem Preis (absteigend)."*

Der Entwickler, der diese Funktion vor fünf Jahren implementiert hat, kannte nur die imperative Welt. Sein Code beschreibt dem Computer exakt das *Wie*:

```
public class KatalogService {
    private List<Artikel> gesamterKatalog;

    public KatalogService(List<Artikel> gesamterKatalog) {
        this.gesamterKatalog = gesamterKatalog;
    }

    public List<String> findePremiumElektronik() {
        // 1. Manueller Zustand für das Zwischenergebnis
        List<Artikel> gefilterteArtikel = new ArrayList<>();

        // 2. Wie iterieren wir? Die klassische Schleife
        for (Artikel artikel : gesamterKatalog) {
            // 3. Verschachtelte Kontrollstrukturen (Cognitive Load)
            if ("Elektronik".equals(artikel.getKategorie())) {
                if (artikel.getPreis() > 100.0) {
                    if (artikel.getBestand() > 0) {
                        gefilterteArtikel.add(artikel);
                          // Seiteneffekt auf lokale Liste
```

```
                    }
                }
            }
        }

        // 4. Imperatives Sortieren durch anonyme innere Klasse
        Collections.sort(gefilterteArtikel, new Comparator<Artikel>() {
            @Override
            public int compare(Artikel a1, Artikel a2) {
                return Double.compare(a2.getPreis(), a1.getPreis());
                    // Absteigend
            }
        });

        // 5. Manueller Zustand für das Endergebnis
        List<String> ergebnisNamen = new ArrayList<>();
        for (Artikel artikel : gefilterteArtikel) {
            ergebnisNamen.add(artikel.getName());
        }

        return ergebnisNamen;
    }
}
```

Die Analyse des Grauens:

Dieser Code funktioniert. Die Unit-Tests sind grün. Aber er ist ein architektonischer Albtraum.

- **Schlechte Lesbarkeit (Cognitive Load):** Die eigentliche Geschäftsregel ist in fünf Ebenen von Einrückungen (for -> if -> if -> if) begraben. Um zu verstehen, was hier passiert, müssen Sie den Code wie eine Maschine im Kopf ausführen.
- **Hoher Boilerplate-Anteil:** Fast 70 % des Codes bestehen aus reiner Verwaltung (Listen initialisieren, Elemente hinzufügen, Comparatoren definieren).
- **Flaschenhals Parallelisierung:** Was passiert, wenn der gesamterKatalog auf 10 Millionen Artikel anwächst? Wir können diesen Code nicht ohne massiven Umbau auf mehrere Threads verteilen, da die gefilterteArtikel-Liste nicht Thread-safe ist. Mehrere Kerne würden sich beim add()-Aufruf gegenseitig überschreiben.

15.2 Der mutierbare Warenkorb: Versteckte Zustandsänderungen

Das Filtern von Daten ist nur die halbe Miete. Das eigentliche Herzstück des J-Shops ist der Bestellprozess. Hier finden wir die Klasse Warenkorb. Sie ist ein klassisches POJO (Plain Old Java Object) mit Gettern und Settern.

```
public class Warenkorb {
    private List<Artikel> positionen = new ArrayList<>();
    private double gesamtPreis = 0.0;

    public void addArtikel(Artikel artikel) {
        this.positionen.add(artikel);
        this.gesamtPreis += artikel.getPreis();
    }

    public List<Artikel> getPositionen() {
        return positionen; // GEFAHR: Wir geben die interne Referenz heraus!
    }

    public double getGesamtPreis() {
        return gesamtPreis;
    }

    public void setGesamtPreis(double gesamtPreis) {
        this.gesamtPreis = gesamtPreis;
    }
}
```

Auf den ersten Blick sieht das nach sauberer Objektorientierung aus. Zustand und Verhalten sind gekapselt. Doch in der Realität eines großen Systems führt dieses Design unweigerlich zu fatalen Seiteneffekten (Side Effects).

Schauen wir uns den RabattService an. Er soll Kunden einen Treuerabatt von 10 % gewähren.

```
public class RabattService {

    public void wendeTreueRabattAn(Warenkorb warenkorb) {
        // 1. Wir manipulieren den Gesamtpreis des Warenkorbs (Seiteneffekt!)
        double neuerPreis = warenkorb.getGesamtPreis() * 0.9;
        warenkorb.setGesamtPreis(neuerPreis);

        // 2. FATAL: Wir manipulieren die Artikel selbst!
        for (Artikel artikel : warenkorb.getPositionen()) {
            artikel.setPreis(artikel.getPreis() * 0.9);
        }
    }
}
```

Die architektonische Zeitbombe:

Wenn Sie die Methode wendeTreueRabattAn aufrufen, passiert etwas Katastrophales.

Der Service ändert nicht nur den Zustand des Warenkorbs, er greift in die Liste der Positionen und ruft setPreis auf jedem einzelnen Artikel auf!

Erinnern Sie sich an Referenzen in Java? Wenn dieser Artikel direkt aus dem globalen KatalogService (dem Cache) stammt, haben wir gerade den Preis dieses Artikels *für alle anderen Kunden im gesamten System* um 10 % reduziert. Ein einziger Methodenaufruf hat den globalen Zustand der gesamten Anwendung korrumpiert.

In der objektorientierten Welt versuchen wir, solche Bugs durch defensives Kopieren (clone()) und strenge Team-Regeln zu verhindern. In der funktionalen Datenfabrik werden wir dieses Problem an der Wurzel packen, indem wir Objekte völlig unveränderlich machen. Wenn es keine setPreis-Methode gibt, gibt es auch keine versteckten Preis-Manipulationen.

15.3 Was wir erreicht haben (und wo die Bugs lauern)

Wir haben nun eine Bestandsaufnahme unseres J-Shops gemacht. Das System befindet sich in der **Iteration 1**.

- **Status:** Der Shop ist funktionsfähig, solange nur ein einzelner Thread läuft und niemand unerwartete Daten übergibt.
- **Die identifizierten Bugs und Risiken:**
  1. Daten-Transformationen sind in unleserlichen, manuellen Schleifen verborgen.
  2. Logik (Wie filtere ich?) und Verhalten (Was filtere ich?) sind untrennbar miteinander verschmolzen.
  3. Geteilter, veränderlicher Zustand (Shared Mutable State) führt zu globalen Seiteneffekten und macht eine Parallelisierung unmöglich.
  4. Die Klasse Artikel und Warenkorb vertrauen blind darauf, dass niemand ihre Setter missbraucht.

Wir können diesen Code nicht länger in der Produktion belassen. Es ist Zeit, die Werkzeuge, die wir im theoretischen Teil dieses Buches gelernt haben, in die Praxis umzusetzen.

Wir beginnen den Umbau in kleinen, sicheren Schritten. Bevor wir die großen Datenflüsse (Streams) etablieren, müssen wir unsere Logik flexibel machen. Wir müssen die harten if-Abfragen herausreißen und sie durch parametrisierbares Verhalten ersetzen.

Schlagen Sie das nächste Kapitel auf. Willkommen in **Iteration 2 – Der Qualitätskontrolleur (Lambdas & Predicates)**.

16 Iteration 2 – Der Qualitätskontrolleur (Lambdas & Predicates)

In der ersten Iteration haben wir die Schwachstellen unseres historischen J-Shops schonungslos offengelegt. Wir sahen Methoden, die vor lauter if-Abfragen ihre eigentliche Geschäftslogik versteckten, und wir sahen, wie starr die alte, rein objektorientierte Architektur sein kann, wenn sich Geschäftsregeln ändern.

In dieser zweiten Iteration machen wir den ersten echten Schritt in Richtung Datenfabrik. Wir reißen noch nicht das gesamte Gebäude ab (die for-Schleifen bleiben vorerst bestehen), aber wir tauschen die fest verschraubten, unflexiblen Kontrollmaschinen gegen dynamische, programmierbare Roboterarme aus. Wir lernen, Verhalten zur Laufzeit zu übergeben.

16.1 Strategie-Muster ablösen: Rabattregeln als Funktionen übergeben

Beginnen wir mit dem Rabatt-System unseres Shops. Die Marketing-Abteilung ist sehr kreativ: Mal gibt es 10 % auf alles, mal 20 % nur auf Elektronik, mal 5 Euro festen Rabatt für Neukunden.

Im alten J-Shop (vor Java 8) hätte ein guter Softwarearchitekt dieses Problem mit dem klassischen **Strategie-Entwurfsmuster (Strategy Pattern)** gelöst. Er hätte ein Interface RabattStrategie gebaut und für jede neue Idee des Marketings eine eigene Klasse geschrieben:

```
// Der alte OOP-Weg: Das Strategie-Muster
public interface RabattStrategie {
    double berechneRabatt(Artikel artikel);
}

public class ElektronikRabatt implements RabattStrategie {
    @Override
    public double berechneRabatt(Artikel artikel) {
        if ("Elektronik".equals(artikel.getKategorie())) {
            return artikel.getPreis() * 0.20; // 20 % Rabatt
        }
        return 0.0;
    }
}

// Aufruf:
RabattStrategie heutigeAktion = new ElektronikRabatt();
```

```
double ersparnis = heutigeAktion.berechneRabatt(meinArtikel);
```

Dieses Design ist absolut korrekt nach den Regeln der Objektorientierung, aber es produziert eine Lawine an kleinen Dateien (Klassen). Für jede noch so triviale Regel (z. B. "5 % auf Bücher") müssen wir eine neue Klasse anlegen, kompilieren und ausliefern.

In unserer neuen funktionalen Datenfabrik erkennen wir: Das Interface RabattStrategie hat nur exakt eine abstrakte Methode. Es ist ein Single Abstract Method (SAM) Interface! Das bedeutet, wir können das gesamte architektonische Muster durch einfache Lambdas ersetzen und uns das Schreiben dutzender Klassen komplett sparen.

Noch besser: Wir verwenden einfach die Standard-Werkzeuge aus dem java.util.function-Paket. Eine Funktion, die einen Artikel entgegennimmt und einen Double-Wert (den Rabatt) zurückgibt, ist schlichtweg eine Function<Artikel, Double>.

Wir bauen den Rabatt-Service unseres Shops um:

```
public class RabattService {

    // Die Methode nimmt das VERHALTEN als Parameter entgegen
    public double berechneGesamtRabatt(Warenkorb korb,
                            Function<Artikel, Double> rabattRegel) {
        double gesamtErsparnis = 0.0;
        for (Artikel artikel : korb.getPositionen()) {
            // Wir wenden die von außen übergebene Regel an
            gesamtErsparnis += rabattRegel.apply(artikel);
        }
        return gesamtErsparnis;
    }
}
```

Schauen Sie sich an, wie elegant die Marketing-Abteilung nun neue Rabatte definieren kann – direkt dort, wo sie gebraucht werden (am Aufrufort), ohne eine einzige neue Klasse zu erzeugen:

```
RabattService service = new RabattService();

// Aktion 1: 10 % auf alles
double rabatt1 = service.berechneGesamtRabatt(korb,
                    artikel -> artikel.getPreis() * 0.10);

// Aktion 2: 5 Euro fest auf Bücher, sonst nichts
double rabatt2 = service.berechneGesamtRabatt(korb, artikel -> {
    if ("Buch".equals(artikel.getKategorie())) return 5.0;
    return 0.0;
```

```
});
```

Wir haben die Logik (*Was* ist die Rabattregel?) vollständig von der Ausführung (*Wie* iteriere ich über den Warenkorb?) entkoppelt.

16.2 Filtern von Artikelkatalogen mit dem java.util.function-Paket

Wenden wir dieses Prinzip nun auf das Monster aus Kapitel 15 an: die Methode findePremiumElektronik() mit ihren tief verschachtelten if-Kaskaden.

Das Hauptproblem dieser alten Methode war ihre Starrheit. Sie war fest verdrahtet (hardcoded) auf "Elektronik", "Preis > 100" und "Bestand > 0". Wenn das Marketing morgen "Premium-Bücher" suchen wollte, musste der Entwickler die Methode kopieren und findePremiumBuecher() schreiben.

Wir lösen dieses Problem, indem wir einen generischen Qualitätskontrolleur an unserem Fließband installieren: das Predicate.

Wir refaktorieren den KatalogService und ersetzen die spezifische Methode durch eine völlig universelle Filter-Methode:

```
public class KatalogService {
    private final List<Artikel> gesamterKatalog;

    public KatalogService(List<Artikel> gesamterKatalog) {
        this.gesamterKatalog = gesamterKatalog;
    }

    // Die neue, generische Suchmaschine
    public List<Artikel> findeArtikel(Predicate<Artikel> bedingung) {
        List<Artikel> ergebnis = new ArrayList<>();
        for (Artikel artikel : gesamterKatalog) {
            // Der Qualitätskontrolleur entscheidet!
            if (bedingung.test(artikel)) {
                ergebnis.add(artikel);
            }
        }
        return ergebnis;
    }
}
```

Die komplexe Logik (die drei verschachtelten if-Abfragen) wandert nun aus dem Service heraus und wird zur Verantwortung des Aufrufers. Der Aufrufer kombiniert einfache Prädikate zu einer komplexen Regel:

```
KatalogService katalog = new KatalogService(alleArtikel);

// Wir definieren kleine, wiederverwendbare Qualitätskontrolleure
Predicate<Artikel> istElektronik = a ->
"Elektronik".equals(a.getKategorie());
Predicate<Artikel> istTeuer = a -> a.getPreis() > 100.0;
Predicate<Artikel> istAufLager = a -> a.getBestand() > 0;

// Wir kombinieren sie mit logischem UND (and)
Predicate<Artikel> premiumElektronikRegel =
istElektronik.and(istTeuer).and(istAufLager);

// Wir übergeben das kombinierte Verhalten an die Suchmaschine
List<Artikel> premiumArtikel = katalog.findeArtikel(premiumElektronikRegel);
```

Vergleichen Sie diesen Aufruf mit dem Code aus Kapitel 15. Die Lesbarkeit hat sich dramatisch verbessert. Der Code dokumentiert sich selbst. Wir lesen: istElektronik.and(istTeuer).and(istAufLager). Das ist fast reine natürliche Sprache.

16.3 Analyse: Was wir aus Iteration 2 lernen

Halten wir einen Moment inne und betrachten die Architektur unseres neuen J-Shops am Ende von Iteration 2.

Die Erfolge:

1. **Reduzierung von Boilerplate:** Wir konnten ganze Klassen-Hierarchien (wie im Strategie-Muster) durch simple Lambda-Ausdrücke ersetzen.
2. **Separation of Concerns (Trennung der Zuständigkeiten):** Der KatalogService weiß nicht mehr, *welche* Artikel gesucht werden. Er weiß nur noch, *wie* er die Liste durchsucht. Die Geschäftsregel liegt nun sauber beim Aufrufer.
3. **Wiederverwendbarkeit:** Die kleinen Prädikate (wie istAufLager) können wir nun im gesamten System (z. B. auch bei der Anzeige im Frontend) wiederverwenden, ohne Code zu duplizieren.

Das verbleibende Problem:

Wir haben das Verhalten flexibilisiert, aber unsere Fabrik läuft immer noch auf alten, handbetriebenen Maschinen. Schauen Sie sich die Methode findeArtikel noch einmal an. Dort steht immer noch eine imperative for-Schleife. Wir initialisieren immer noch manuell eine ArrayList und wir verwalten den Zustand (ergebnis.add(artikel)) weiterhin selbst.

Wenn der Katalog 10 Millionen Artikel hat, wird diese for-Schleife auf einem einzigen

CPU-Kern ausgeführt, während die anderen 15 Kerne Ihres modernen Servers sich langweilen.

Wir haben das "Was" vom "Wie" getrennt. Nun ist es an der Zeit, das "Wie" komplett an die Java Virtual Machine abzugeben. Im nächsten Schritt reißen wir die for-Schleifen endgültig aus unserem System.

Schlagen Sie das nächste Kapitel auf. Willkommen in **Iteration 3 – Die Fließbänder laufen an (Die Stream API)**.

17 Iteration 3 – Die Fließbänder laufen an (Die Stream API)

In der letzten Iteration haben wir die starren, fest verdrahteten Regeln unseres J-Shops aufgebrochen. Das Marketing kann nun beliebige Suchbedingungen als kleine, handliche Prädikate (Lambdas) an den KatalogService übergeben.

Doch wenn wir einen Blick in den Maschinenraum werfen, sehen wir, dass die eigentliche Arbeit immer noch von Hand erledigt wird. Der Transport der Daten von einer Station zur nächsten erfolgt über klassische for-Schleifen und das ständige Umfüllen von Artikeln in temporäre ArrayList-Behälter.

In dieser dritten Iteration reißen wir diese handbetriebenen Förderbänder ab. Wir installieren das Herzstück der funktionalen Datenfabrik: die Java Stream API. Wir werden sehen, wie sich komplexer, fehleranfälliger Code in flüssige, deklarative Pipelines verwandelt.

17.1 Die alte Such-Schleife wird zur deklarativen Pipeline

Erinnern wir uns an den Zustand unseres KatalogService am Ende von Iteration 2. Wir hatten die Methode findeArtikel bereits mit einem Predicate ausgestattet, aber die Implementierung war noch rein imperativ:

```
// Der Zustand nach Iteration 2 (Imperativer Transport, funktionales Filtern)
public List<Artikel> findeArtikel(Predicate<Artikel> bedingung) {
    List<Artikel> ergebnis = new ArrayList<>();
    for (Artikel artikel : gesamterKatalog) {
        if (bedingung.test(artikel)) {
            ergebnis.add(artikel);
        }
    }
    return ergebnis;
}
```

Dieser Code zwingt uns, den Zustand (ergebnis) selbst zu verwalten. Wenn wir diesen Suchprozess für einen riesigen Katalog parallelisieren wollten, hätten wir ein massives Problem mit der Thread-Sicherheit der ArrayList.

Bauen wir nun das Fließband auf. Wir nutzen den Katalog als Quelle, schalten das übergebene Prädikat als Qualitätskontrolleur (Filter) dazwischen und sammeln die Ergebnisse am Ende in einer neuen, sicheren Liste ein.

```
// Der Zustand nach Iteration 3 (Vollständig funktionale Pipeline)
public List<Artikel> findeArtikel(Predicate<Artikel> bedingung) {
    return gesamterKatalog.stream()
            .filter(bedingung)
            .toList();
}
```

Das ist alles. Aus sieben Zeilen fehleranfälligem Boilerplate-Code sind drei Zeilen reine, deklarative Absicht geworden.

- **Die Quelle:** gesamterKatalog.stream() startet das Fließband.
- **Die Operation:** .filter(bedingung) wendet unsere dynamische Regel an.
- **Die Senke:** .toList() zieht die Daten durch die Pipeline und sammelt sie sicher auf.

Wenn das Marketing nun anruft und berichtet, dass die Suche bei 10 Millionen Artikeln zu langsam ist, müssen wir keine Locks, Thread-Pools oder synchronized-Blöcke schreiben. Wir ändern genau ein Wort: Wir tauschen .stream() gegen .parallelStream() aus. Die JVM teilt das Fließband automatisch auf alle verfügbaren Prozessorkerne auf, da unser Code dank des fehlenden geteilten Zustands zu 100 % Thread-safe ist.

17.2 Komplexe Transformationen (Kunde -> Bestellungen -> Artikel) mit flatMap

Die einfache Suche haben wir optimiert. Doch die Datenstrukturen in einem E-Commerce-Shop sind selten flach. Meistens haben wir es mit tief verschachtelten Hierarchien zu tun.

Die Marketing-Abteilung hat eine neue Anforderung: *"Wir möchten einem bestimmten Kunden personalisierte Werbung schicken. Dafür benötigen wir eine Liste aller eindeutigen Artikelkategorien, die dieser Kunde jemals bei uns gekauft hat."*

- Ein Kunde besitzt eine Liste von Bestellungen (List<Bestellung>).
- Eine Bestellung besitzt eine Liste von Positionen (List<Artikel>).
- Jeder Artikel hat eine Eigenschaft Kategorie (z. B. "Elektronik", "Buch").

Im alten J-Shop sah der Code für diese Aufgabe so aus:

```
public Set<String> ermittleGekaufteKategorien(Kunde kunde) {
    Set<String> kategorien = new HashSet<>();
      // Zustand für eindeutige Kategorien

    // Schleife Ebene 1: Bestellungen
    for (Bestellung bestellung : kunde.getBestellungen()) {
```

```
        // Schleife Ebene 2: Artikel in der Bestellung
        for (Artikel artikel : bestellung.getPositionen()) {

            // Seiteneffekt: Den Zustand verändern
            kategorien.add(artikel.getKategorie());
        }
    }
    return kategorien;
}
```

Doppelt verschachtelte for-Schleifen sind berüchtigt dafür, schwer lesbar zu sein ("Arrow Code" – der Code rückt immer weiter nach rechts ein). Wenn wir hier noch if-Abfragen (z. B. nur bezahlte Bestellungen) einfügen, verliert man völlig den Überblick.

Wir greifen in unseren funktionalen Werkzeugkasten und holen die Maschine für die Dimensionstransformation heraus: flatMap.

Zur Erinnerung: flatMap nimmt einen Container (wie eine Liste), öffnet ihn, nimmt die darin enthaltenen Elemente heraus und legt sie flach auf unser Haupt-Fließband.

```
public Set<String> ermittleGekaufteKategorien(Kunde kunde) {
    return kunde.getBestellungen().stream()
            // 1. Bestellungen flachklopfen in einen Stream von Artikeln
            .flatMap(bestellung -> bestellung.getPositionen().stream())
            // 2. Artikel transformieren in Kategorien (String)
            .map(Artikel::getKategorie)
            // 3. Eindeutige Elemente sammeln (toSet eliminiert Duplikate)
            .collect(Collectors.toSet());
}
```

Betrachten Sie die Eleganz dieses Datenflusses! Wir beginnen mit Bestellungen, entpacken diese in Artikel, wandeln die Artikel in ihre Kategorien um und sammeln das Ergebnis. Wir haben keine einzige Hilfsvariable und keinen manuellen Zustand mehr deklariert.

17.3 Das Aggregieren der Monatsumsätze mit reduce

Unsere Fließbänder laufen nun flüssig. Wir können filtern und komplexe Hierarchien entpacken. Im letzten Schritt dieser Iteration widmen wir uns der Königsdisziplin der Buchhaltung: dem Aggregieren (Zusammenfassen) von Daten.

Die Buchhaltung fordert: *"Berechne uns den gesamten Netto-Umsatz aller Bestellungen aus dem aktuellen Monat, die bereits bezahlt sind."*

Im imperativen Legacy-Code unseres J-Shops sah das so aus:

```
public double berechneMonatsUmsatz(List<Bestellung> alleBestellungen, Month
monat) {
    double gesamtUmsatz = 0.0; // Globaler, veränderlicher Zustand

    for (Bestellung b : alleBestellungen) {
        if ("BEZAHLT".equals(b.getStatus())) {
            if (b.getDatum().getMonth() == monat) {
                // Versteckte Preisberechnung für jede Position
                double bestellWert = 0.0;
                for (Artikel a : b.getPositionen()) {
                    bestellWert += a.getPreis();
                }
                gesamtUmsatz += bestellWert; // Seiteneffekt!
            }
        }
    }
    return gesamtUmsatz;
}
```

Auch hier sehen wir wieder die typischen Symptome: gesamtUmsatz ist ein veränderlicher Zustand, der bei jedem Durchlauf manipuliert wird. Die Logik für die Summenbildung ist mit der Filterlogik vermischt.

Wir setzen nun unsere Stream-Maschinen ein und beenden das Fließband mit der ultimativen Reduktions-Operation: reduce. Wir wollen Tausende von Bestellungen auf eine einzige Zahl (ein double) zusammendampfen.

```
public double berechneMonatsUmsatz(List<Bestellung> alleBestellungen,
                                   Month monat) {
    return alleBestellungen.stream()
            // 1. Filtern nach Status
            .filter(b -> "BEZAHLT".equals(b.getStatus()))
            // 2. Filtern nach dem gewünschten Monat
            .filter(b -> b.getDatum().getMonth() == monat)
            // 3. Transformation: Bestellung -> Gesamtwert als Double
            .mapToDouble(this::berechneBestellWert)
            // 4. Reduktion: Alle Werte aufsummieren
            .sum(); // .sum() ist eine bequeme Kurzform von
                    // .reduce(0.0, Double::sum) auf primitiven Streams
}

// Hilfsmethode, ebenfalls funktional!
private double berechneBestellWert(Bestellung bestellung) {
    return bestellung.getPositionen().stream()
            .mapToDouble(Artikel::getPreis)
```

```
                .sum();
}
```

Tipp aus der Praxis: Fällt Ihnen auf, dass wir hier mapToDouble statt map verwendet haben? Wie in Kapitel 5 besprochen, nutzen wir für primitive Datentypen (wie double oder int) die spezialisierten primitiven Streams (DoubleStream, IntStream). Das verhindert das langsame Autoboxing und bietet uns sofort fertige, mathematische Endstationen wie .sum(), .average() oder .max().

Fazit der Iteration 3

Wir haben in dieser Iteration enorme Fortschritte gemacht. Wenn Sie den Code unseres J-Shops jetzt betrachten, werden Sie feststellen, dass fast alle for-Schleifen verschwunden sind. Das "Wie" der Iteration liegt nun komplett in den Händen der Java Virtual Machine. Die Lesbarkeit unseres Codes hat sich drastisch erhöht, da wir unsere fachliche Intention (Filtern, Mappen, Reduzieren) direkt in der Syntax der Sprache ausdrücken.

Doch während unsere Fließbänder nun effizient und elegant laufen, haben wir ein grundlegendes Problem noch nicht gelöst: Die Bauteile (Objekte), die über diese Bänder rollen, sind immer noch aus weichem, formbarem Material. Die Klassen Kunde, Bestellung und Artikel haben weiterhin Setter-Methoden. Jeder böswillige (oder unachtsame) Entwickler könnte mitten in unserer Stream-Pipeline artikel.setPreis(0) aufrufen und den gesamten Datenfluss sabotieren.

Es ist an der Zeit, unsere Daten in Stein zu meißeln. Im nächsten Kapitel frieren wir den Zustand unseres Shops ein und verabschieden uns endgültig von Seiteneffekten.

Schlagen Sie das nächste Kapitel auf. Willkommen in **Iteration 4 – In Stein gemeißelt (Records & Immutability)**.

18 Iteration 4 – In Stein gemeißelt (Records & Immutability)

Unsere Datenfabrik ist in den ersten drei Iterationen extrem effizient geworden. Die Fließbänder (Streams) laufen reibungslos, filtern Millionen von Artikeln in Millisekunden und aggregieren Umsätze über komplexe Hierarchien hinweg. Wir könnten an dieser Stelle eigentlich sehr zufrieden sein.

Doch es gibt einen unsichtbaren Feind in unserem System. Ein Feind, der im Schatten der Objektorientierung lauert und darauf wartet, dass wir den Schalter von .stream() auf .parallelStream() umlegen, um unser System vollständig in die Luft zu jagen. Dieser Feind ist der **veränderliche Zustand (Mutable State)** unserer Domänenobjekte.

In dieser vierten Iteration werden wir das Material, aus dem unsere Bauteile bestehen, austauschen. Wir ersetzen weichen, formbaren Code durch massiven Stein. Wir machen unsere Domäne absolut unveränderlich.

18.1 Die Katastrophe abwenden: Setter aus den Entitäten entfernen

Lassen Sie uns einen Blick auf eine unserer zentralen Klassen im J-Shop werfen: den Artikel. In der historischen, imperativen Version des Shops sah diese Klasse aus wie ein klassisches POJO (Plain Old Java Object), das Sie aus zahllosen Tutorials kennen:

```
public class Artikel {
    private String id;
    private String name;
    private double preis;
    private String kategorie;

    public Artikel(String id, String name, double preis, String kategorie) {
        this.id = id;
        this.name = name;
        this.preis = preis;
        this.kategorie = kategorie;
    }

    // Unzählige Getter und Setter...
    public double getPreis() { return preis; }
    public void setPreis(double preis) { this.preis = preis; }

    public String getKategorie() { return kategorie; }
    public void setKategorie(String kategorie) { this.kategorie = kategorie;
}
}
```

Warum ist das brandgefährlich für unsere neuen Fließbänder?

Stellen Sie sich vor, wir haben eine globale Liste aller Artikel im Speicher (als Cache). Ein Entwickler erhält die Aufgabe: *"Berechne den hypothetischen Umsatz, wenn wir heute auf alle Elektronikartikel 20 % Rabatt geben."*

Der Entwickler liebt die neue Stream-API und schreibt folgenden Code:

```
// FATALER FEHLER: Seiteneffekt innerhalb der Pipeline
double hypothetischerUmsatz = gesamterKatalog.stream()
    .filter(a -> "Elektronik".equals(a.getKategorie()))
    .peek(a -> a.setPreis(a.getPreis() * 0.8))
                // Wir modifizieren das Original-Objekt!
    .mapToDouble(Artikel::getPreis)
    .sum();
```

Der Entwickler freut sich, denn die Berechnung stimmt. Aber er hat gerade den Cache des gesamten Shops zerstört. Die setPreis-Methode hat die Original-Artikel manipuliert. Alle Kunden, die in diesem Moment im Shop surfen, sehen plötzlich die reduzierten Preise, obwohl die Aktion nur "hypothetisch" berechnet werden sollte. Wenn mehrere Threads gleichzeitig diesen Code ausführen, entstehen "Race Conditions" – die Preise werden mehrfach reduziert oder völlig unvorhersehbar überschrieben.

Um unsere funktionale Pipeline zu schützen, müssen wir verhindern, dass Bauteile modifiziert werden, während (oder nachdem) sie produziert wurden. Wir müssen die Setter abreißen.

18.2 Die Transformation zu Java Records

Wir könnten nun in der Klasse Artikel alle Felder als final markieren und die Setter löschen. Das wäre der klassische Weg. Doch Java bietet uns seit Version 16 ein weitaus mächtigeres und eleganteres Werkzeug für reine Daten-Container: **Records**.

Wir löschen die alte, 50-zeilige Klasse Artikel komplett und ersetzen sie durch eine einzige Zeile Code:

```
public record Artikel(String id, String name, double preis, String kategorie)
{
    // Der Compiler generiert automatisch:
    // - private final Felder
    // - Konstruktor für alle Felder
    // - Lesende Methoden: id(), name(), preis(), kategorie()
    // - Korrektes equals(), hashCode() und toString()
```

```
}
```

Mit dieser einen Zeile haben wir eine Festung gebaut. Ein Artikel ist nun tiefgehend unveränderlich (immutable). Es gibt keine Möglichkeit mehr, den Preis eines einmal instanziierten Artikels zu verändern. Unser Fließband ist nun absolut sicher vor mutierenden Seiteneffekten.

Wir ziehen diese Änderung konsequent im gesamten Shop durch. Auch Kunde, Adresse und Bestellung werden zu Records umgewandelt.

```
public record Kunde(String id, String name, Adresse rechnungsAdresse,
                    List<Bestellung> bestellungen) {
    // Kompakt, deklarativ und felsenfest
}
```

Ein kritischer Hinweis zur Immutability: Ein Record macht nur seine eigenen Felder final. Wenn Sie, wie im obigen Kunde-Record, eine List<Bestellung> übergeben, könnte jemand diese Liste von außen manipulieren (kunde.bestellungen().add(...)), wenn es sich um eine normale ArrayList handelt. Um echte (tiefe) Unveränderlichkeit zu garantieren, müssen wir sicherstellen, dass die übergebenen Listen selbst unveränderlich sind, zum Beispiel durch die Nutzung von List.copyOf() im kompakten Konstruktor des Records.

18.3 Das Warenkorb-Update: Neue Kopien statt mutiertem Zustand

Das Entfernen von Settern führt uns sofort zum nächsten logischen Problem: *Wenn absolut alles unveränderlich ist, wie legt der Kunde dann einen neuen Artikel in seinen Warenkorb?*

Schauen wir uns die alte Warenkorb-Klasse an:

```
// Der alte, mutierbare Warenkorb
public class Warenkorb {
    private List<Artikel> positionen = new ArrayList<>();

    public void addArtikel(Artikel artikel) {
        this.positionen.add(artikel); // Zustand wird verändert!
    }
}
```

In der funktionalen Programmierung gilt das Mantra: **Wir verändern keinen Zustand, wir erzeugen neuen Zustand auf Basis des alten.**

Wir wandeln den Warenkorb in einen Record um und nutzen das sogenannte **Wither-**

Pattern (Methoden, die mit with... beginnen). Eine Wither-Methode nimmt das Objekt, wendet eine gewünschte Änderung an und gibt eine *komplett neue Kopie* des Objekts zurück. Das alte Objekt bleibt unangetastet.

```
public record Warenkorb(List<Artikel> positionen) {

    // Kompakter Konstruktor: Wir garantieren, dass die interne Liste
    // immutable ist!
    public Warenkorb {
        positionen = List.copyOf(positionen);
    }

    // Die funktionale Alternative zu "addArtikel"
    public Warenkorb withZusaetzlichemArtikel(Artikel neuerArtikel) {
        // 1. Wir kopieren die alten Positionen in eine neue,
        // veränderbare Liste
        List<Artikel> neuePositionen = new ArrayList<>(this.positionen);

        // 2. Wir fügen den neuen Artikel hinzu
        neuePositionen.add(neuerArtikel);

        // 3. Wir geben einen brandneuen Warenkorb zurück!
        return new Warenkorb(neuePositionen);
    }
}
```

Wenn ein Kunde nun einkauft, sieht der Ablauf im Controller unseres Systems so aus:

```
Warenkorb alterKorb = kundenSession.getWarenkorb(); // z.B. 2 Artikel

// Wir "verändern" den Korb nicht, wir lassen eine neue Version produzieren
Warenkorb neuerKorb = alterKorb.withZusaetzlichemArtikel(neuesBuch);

kundenSession.setWarenkorb(neuerKorb); // Wir speichern den neuen Zustand
```

Warum ist das so genial?

Stellen Sie sich vor, der Nutzer klickt wie wild mehrfach auf "In den Warenkorb", und zwei HTTP-Requests (Threads) kommen zeitgleich im Backend an.

Beim alten, mutierbaren Warenkorb hätten beide Threads gleichzeitig positionen.add() aufgerufen, was zu einer ConcurrentModificationException oder zu verlorenen Artikeln geführt hätte.

Mit unserem neuen Warenkorb-Record passiert Folgendes: Beide Threads lesen den

alterKorb. Beide Threads erzeugen *unabhängig voneinander* in ihrem eigenen Speicherbereich einen neuerKorb. Es gibt absolut keine Konflikte beim Schreiben, weil niemand den alterKorb verändert! Das Einzige, was wir am Ende (z.B. durch optimistische Sperren in der Datenbank) lösen müssen, ist, welche der beiden neuen Versionen wir als endgültige Wahrheit speichern. Unser Kern-Code ist von Haus aus Thread-safe.

Fazit der Iteration 4

Wir haben das Herzstück unseres Shops – die Domänenmodelle – massiv gehärtet. Durch den Einsatz von Java Records haben wir hunderte Zeilen Boilerplate-Code (Getter, Setter, Konstruktoren) gelöscht. Viel wichtiger jedoch: Wir haben eine Architektur geschaffen, in der Objekte nach ihrer Erschaffung niemals wieder verändert werden können.

Unsere funktionalen Streams aus Iteration 3 können nun Millionen von Artikeln parallel verarbeiten, ohne dass wir uns jemals wieder Sorgen um heimliche Seiteneffekte machen müssen. Die Daten fließen nicht nur elegant, sie fließen auch in absoluter Sicherheit.

Doch eine gewaltige Fehlerquelle existiert noch in unserem J-Shop. Was passiert, wenn wir nach einem Kunden suchen, dieser aber nicht in der Datenbank existiert? Der alte Code gibt in diesem Fall einfach null zurück – ein tickende Zeitbombe, die unsere neuen Pipelines sofort sprengen wird.

Es ist Zeit, den Milliard-Dollar-Fehler auszumerzen. Schlagen Sie das nächste Kapitel auf. Willkommen in **Iteration 5 – Schutz vor der Leere (Optional)**.

19 Iteration 5 – Schutz vor der Leere (Optional)

Unsere Datenfabrik, der J-Shop, hat in den letzten Iterationen eine erstaunliche Wandlung vollzogen. Die ratternden, fehleranfälligen for-Schleifen sind eleganten Fließbändern gewichen (Streams). Unsere Produkte (die Domänen-Objekte wie Artikel und Warenkorb) bestehen nun aus massivem, unveränderlichem Stein (Records), sodass niemand sie während der Fahrt auf dem Band heimlich verbiegen kann.

Doch was nützt uns das schönste Fließband, wenn wir ins Lager greifen und ins Leere fassen?

In dieser fünften Iteration widmen wir uns der berüchtigtsten Fehlerquelle in der Java-Welt: der NullPointerException. In einem klassischen E-Commerce-System gibt es unzählige Momente, in denen Daten fehlen können. Ein Kunde sucht nach einer Artikelnummer, die es nicht gibt. Ein Nutzer hat in seinem Profil keine Rechnungsadresse hinterlegt. Wenn wir diese "Leere" als null über unsere funktionalen Fließbänder schicken, stürzt das gesamte System ab.

19.1 Den null-Wahn besiegen: Kunden und Adressen sicher auflösen

Schauen wir uns an, wie der alte J-Shop vor unserem Umbau mit fehlenden Daten umgegangen ist. Wir betrachten die Klasse KundenService, die für den Versand von Newslettern zuständig ist.

```
// Der alte, imperative Weg: Null-Checks überall
public class KundenService {
    private KundenRepository repository; // Simuliert den Datenbank-Zugriff

    public void sendeNewsletter(String kundenId) {
        Kunde kunde = repository.findeKunde(kundenId);

        // 1. Die ständige Angst vor der Leere
        if (kunde != null) {
            String email = kunde.email();
            if (email != null && !email.isEmpty()) {
                System.out.println("Sende Newsletter an: " + email);
            } else {
                System.out.println("Kunde hat keine E-Mail-Adresse.");
            }
        } else {
            System.out.println("Kunde nicht gefunden.");
        }
    }
}
```

Das Problem hier ist nicht nur die tiefe Verschachtelung ("Arrow-Code"), sondern das fehlende Vertrauen in den Vertrag der Methode findeKunde. Die Signatur sagt: Kunde findeKunde(String id). Der Compiler vertraut darauf, dass immer ein Kunde zurückkommt. Aber zur Laufzeit bricht die Methode dieses Versprechen heimlich, indem sie null zurückgibt.

In unserer funktionalen Architektur machen wir die Möglichkeit des Fehlens explizit. Wir ändern das Repository, sodass es unser gepanzertes Transportkistchen (Optional) zurückgibt:

```
// Der neue Vertrag: Die Leere wird kommuniziert
public interface KundenRepository {
    Optional<Kunde> findeKunde(String id);
}
```

Nun bauen wir die Methode sendeNewsletter funktional um. Wir nutzen die Monaden-Eigenschaften von Optional, um die Abwesenheit von Werten elegant abzufangen:

```
public void sendeNewsletter(String kundenId) {
    repository.findeKunde(kundenId)
        .map(Kunde::email) // Extrahiert die E-Mail (falls Kunde existiert)
        .filter(email -> !email.isEmpty()) // Prüft, ob sie nicht leer ist
        .ifPresentOrElse(
            email -> System.out.println("Sende Newsletter an: " + email),
            () -> System.out.println("Kein Versand möglich " +
                                     "(Kunde fehlt oder keine E-Mail).")
        );
}
```

Sehen Sie den Unterschied? Wir haben keine einzige if-Abfrage mehr geschrieben. Wir definieren den idealen Datenfluss (den "Happy Path"): *Nimm den Kunden, hole die E-Mail, filtere leere Strings heraus und sende die Nachricht.* Wenn an irgendeiner Stelle die Kiste leer ist (weil der Kunde nicht existiert oder keine E-Mail hat), wird die Operation automatisch abgebrochen und der Code springt in den orElse-Zweig.

19.2 Verkettete flatMap-Aufrufe für tiefe Objekt-Hierarchien

Richtig kompliziert wird null in der imperativen Welt erst dann, wenn wir tief in Objekt-Hierarchien absteigen müssen.

Das Marketing möchte wissen, in welcher Stadt ein bestimmter Kunde wohnt, um ihm regionale Angebote zu machen. Erinnern wir uns an unseren Kunde-Record aus Iteration

4. Er besitzt eine Adresse, und die Adresse besitzt eine stadt. Aber: Ein Kunde hat vielleicht keine Adresse hinterlegt.

Im alten J-Shop führte das zu solchen Code-Konstrukten:

```
public String ermittleKundenStadt(String kundenId) {
    Kunde kunde = repository.findeKunde(kundenId);
    if (kunde != null) {
        Adresse adresse = kunde.adresse();
        if (adresse != null) {
            String stadt = adresse.stadt();
            if (stadt != null) {
                return stadt.toUpperCase();
            }
        }
    }
    return "UNBEKANNT";
}
```

Dieser Code ist furchtbar zu lesen und extrem fehleranfällig. Ein einziger vergessener Null-Check in dieser Kette, und die Anwendung stürzt ab.

In unserer neuen Datenfabrik wissen wir, dass Werte fehlen können. Daher passen wir unsere Records konsequent an. Getter-Methoden (die bei Records einfach wie das Feld heißen), die etwas Optionales zurückgeben, sollten auch Optional liefern!

```
public record Adresse(String strasse, String stadt, String plz) {
    // Die Stadt ist zwingend, also geben wir den String direkt zurück
}

public record Kunde(String id, String name, Adresse rechnungsAdresse) {
    // Die Adresse ist optional! Wir geben sofort das gepanzerte
    // Kistchen zurück.
    public Optional<Adresse> optRechnungsAdresse() {
        return Optional.ofNullable(this.rechnungsAdresse);
    }
}
```

Nun refaktorieren wir die Methode ermittleKundenStadt. Wir nutzen flatMap, um die verschachtelten Kistchen ("Optional im Optional") flachzuklopfen:

```
public String ermittleKundenStadt(String kundenId) {
    return repository.findeKunde(kundenId)      // Liefert Optional<Kunde>
            .flatMap(Kunde::optRechnungsAdresse)   // flatMap, weil
```

```
                                // optRechnungsAdresse() ein Optional liefert!
            .map(Adresse::stadt)    // map, weil stadt() einen
                                    //  normalen String liefert
            .map(String::toUpperCase)  // Transformation
            .orElse("UNBEKANNT");      // Der Fallback am Ende des Fließbands
}
```

Wenn findeKunde eine leere Kiste liefert, macht flatMap nichts und reicht die leere Kiste weiter.

Wenn der Kunde existiert, aber rechnungsAdresse() eine leere Kiste liefert, macht das nachfolgende map nichts und reicht die leere Kiste weiter.

Erst am ganz am Ende, beim Auspacken mit orElse, entscheiden wir, was passiert, wenn die Kiste auf ihrer Reise leer geblieben ist.

19.3 Analyse: Robuster Code ohne NullPointerException

Lassen Sie uns den Zustand unseres J-Shops nach dieser fünften Iteration evaluieren.

Die Erfolge:

1. **Das Typsystem denkt für uns mit:** Indem wir Optional als Rückgabewert nutzen, zwingen wir jeden Entwickler im Team, sich mit dem potenziellen Fehlen von Werten auseinanderzusetzen. Der Compiler weigert sich schlichtweg, Code zu kompilieren, der blind auf potenziell leere Objekte zugreift.
2. **Kognitive Entlastung:** Die tiefen if-else-Kaskaden sind aus unserem Code verschwunden. Wir lesen die fachliche Anforderung wieder von oben nach unten, wie einen klaren Arbeitsauftrag.
3. **Nahtlose Integration in Streams:** Da Optional eine Monade ist, integriert sie sich perfekt in unsere Fließbänder aus Iteration 3. Wir können Listen von Optionals elegant filtern oder entpacken.

Das verbleibende Problem:

Wir haben den null-Wahn besiegt. Wenn Daten einfach nur *fehlen*, läuft unsere Fabrik ungestört weiter.

Aber was passiert, wenn etwas *aktiv kaputtgeht*? Was ist, wenn wir eine Rechnung als PDF auf die Festplatte schreiben wollen, aber die Festplatte voll ist? Was, wenn wir die Kreditkarte des Kunden belasten wollen, aber der Server der Bank nicht antwortet?

In diesen Momenten wirft die Java-Laufzeitumgebung eine Exception. Und wie wir in der

Theorie (Kapitel 11) gelernt haben, sind Exceptions der Not-Aus-Schalter für unsere funktionalen Fließbänder. Eine unbehandelte IOException zerreißt unsere Stream-Pipeline in der Luft.

Es ist Zeit für das Sicherheitsnetz. Im nächsten Kapitel rüsten wir den J-Shop mit funktionaler Fehlerbehandlung aus, damit selbst harte Systemabstürze elegant als Datenpakete über unser Fließband rollen.

Schlagen Sie das nächste Kapitel auf. Willkommen in **Iteration 6 – Fehler ohne Absturz (Funktionale Exception-Behandlung)**.

20 Iteration 6 – Fehler ohne Absturz (Funktionale Exception-Behandlung)

Unsere Datenfabrik, der J-Shop, ist mittlerweile ein Meisterwerk der Effizienz. Wir haben die alten, rostigen for-Schleifen durch rasante Streams ersetzt. Wir haben das weiche Material unserer Bauteile in massiven Stein (Records) verwandelt, und wir haben das schwarze Loch der NullPointerException durch die Optional-Monade versiegelt.

Doch es gibt ein Szenario, das unsere schönen neuen Fließbänder immer noch in Bruchteilen einer Sekunde zum Stillstand bringen kann. In der realen Welt der Softwareentwicklung interagieren wir mit unzuverlässigen Nachbarn: Datenbanken antworten nicht, Festplatten sind voll, und Drittanbieter-APIs werfen HTTP-500-Fehler.

In diesem Kapitel kümmern wir uns um diese externen Störfaktoren. Wir lernen, wie wir verhindern, dass ein einzelnes fehlerhaftes Bauteil den Not-Aus-Schalter unserer gesamten Fabrik auslöst.

20.1 Der Netzwerk-Fehler beim Rechnungs-Export zerreißt die Pipeline

Betrachten wir einen neuen, kritischen Prozess in unserem J-Shop. Jeden Abend um 23:00 Uhr läuft ein Batch-Job, der alle bezahlten Rechnungen des Tages nimmt und sie als PDF an das externe Buchhaltungssystem (DATEV) exportiert.

Der Entwickler, der die Stream-API frisch gelernt hat, schreibt folgenden eleganten Code:

```
public class RechnungsExportService {
    private final DatevClient datevClient;

    public void exportiereTagesRechnungen(List<Rechnung> rechnungen) {
        System.out.println("Starte Export von " + rechnungen.size() +
                           " Rechnungen...");

        // Die funktionale Pipeline
        List<String> exportierteIds = rechnungen.stream()
            .map(rechnung -> datevClient.sendeAnDatev(rechnung))
                 // Gefährliche Operation!
            .toList();

        System.out.println("Erfolgreich exportiert: " +
                           exportierteIds.size());
    }
```

```
}
```

Das sieht sauber aus. Doch schauen wir uns die Signatur des DatevClient an:

```
public interface DatevClient {
    // Wirft eine Checked Exception, wenn das Netzwerk streikt
    String sendeAnDatev(Rechnung rechnung) throws NetzwerkException;
}
```

Wie wir in der Theorie (Kapitel 11) gelernt haben, weigert sich der Java-Compiler an dieser Stelle. Die Methode map erwartet eine reine Funktion, und eine NetzwerkException ist ein massiver Seiteneffekt.

Der typische, imperative "Workaround" sieht leider so aus:

```
// Der "Workaround" (Anti-Pattern)
List<String> exportierteIds = rechnungen.stream()
    .map(rechnung -> {
        try {
            return datevClient.sendeAnDatev(rechnung);
        } catch (NetzwerkException e) {
            // Wir verstecken die Exception in einer RuntimeException!
            throw new RuntimeException("Export fehlgeschlagen für: " +
                                       rechnung.id(), e);
        }
    })
    .toList();
```

Was passiert, wenn dieser Job in der Produktion läuft und 10.000 Rechnungen exportieren soll?

Bei Rechnung Nummer 1 bis 5.000 funktioniert alles einwandfrei. Bei Rechnung 5.001 hat der DATEV-Server für den Bruchteil einer Sekunde einen Netzwerkhänger. Die NetzwerkException tritt auf. Unser Lambda fängt sie, verpackt sie in eine RuntimeException und wirft sie.

Das Fließband hält mit einem gewaltigen Knall an. Die gesamte Methode exportiereTagesRechnungen stürzt ab.

Die Konsequenz:

1. Wir wissen nicht, welche Rechnungen erfolgreich waren.
2. Rechnung 5.002 bis 10.000 werden überhaupt nicht mehr angefasst, obwohl der Server vielleicht schon wieder erreichbar ist.
   Ein einzelner Fehler hat den gesamten Tagesabschluss ruiniert.

20.2 Einführung der Try/Either-Monade für sicheres Stream-Processing

In einer funktionalen Fabrik lösen wir niemals den Not-Aus-Schalter aus, nur weil ein Bauteil kaputt ist. Stattdessen markieren wir das kaputte Bauteil, legen einen roten Fehlerbericht dazu und lassen es ganz normal auf dem Fließband bis zur Endstation weiterrollen.

Dafür holen wir unsere Try-Monade aus Kapitel 11 zurück. Wir nutzen sie als gepanzerte Transportkiste, die entweder das fertige Produkt (den Erfolg) oder den Fehlerbericht (die Exception) enthält.

```
// Unsere bewährte Try-Monade (zusammengefasst)
public record Try<T>(T wert, Exception fehler) {
    public static <T> Try<T> success(T wert) {
        return new Try<>(wert, null);
    }

    public static <T> Try<T> failure(Exception fehler) {
        return new Try<>(null, fehler);
    }

    public boolean isSuccess() {
        return fehler == null;
    }
}
```

Nun bauen wir für unseren gefährlichen DatevClient eine sichere Umhüllung (einen Wrapper). Diese Methode fängt die harte Exception ab und verwandelt sie in unser reines Daten-Objekt (Try):

```
public class RechnungsExportService {
    private final DatevClient datevClient;

    // ... Konstruktor ...

    // Die funktionale Schutzschicht: Wirft NIEMALS eine Exception!
    private Try<String> exportiereSicher(Rechnung rechnung) {
        try {
            String exportId = datevClient.sendeAnDatev(rechnung);
            return Try.success(exportId); // Kiste mit Erfolgs-Wert
        } catch (NetzwerkException e) {
            return Try.failure(e);        // Kiste mit Fehlerbericht
        }
    }
}
```

20.3 Analyse: Erfolge und Fehler am Ende des Fließbands trennen

Jetzt lassen wir unser Fließband erneut anlaufen. Dieses Mal rufen wir nicht direkt den unsicheren Client auf, sondern unsere neue, kugelsichere exportiereSicher-Methode.

```
public void exportiereTagesRechnungen(List<Rechnung> rechnungen) {

    // 1. Die Pipeline läuft zu 100 % sicher durch
    List<Try<String>> alleErgebnisse = rechnungen.stream()
        .map(this::exportiereSicher) // Jedes Element wird zu einem Try
        .toList();

    // 2. Wir stehen an der Endstation und sortieren die Kisten
    Map<Boolean, List<Try<String>>> getrennt = alleErgebnisse.stream()
        .collect(Collectors.partitioningBy(Try::isSuccess));

    List<Try<String>> erfolge = getrennt.get(true);
    List<Try<String>> fehler = getrennt.get(false);

    // 3. Auswertung und Alarmierung
    System.out.println("Erfolgreich exportiert: " + erfolge.size());

    if (!fehler.isEmpty()) {
        System.err.println("ACHTUNG: " + fehler.size() +
                           " Rechnungen sind fehlgeschlagen!");
        // Hier können wir nun gezielt das Monitoring-System alarmieren
        // oder Retries anstoßen
        fehler.forEach(f -> System.err.println("Grund: " +
                                               f.fehler().getMessage()));
    }
}
```

Was passiert nun in der Produktion?

Die 10.000 Rechnungen rollen über das Fließband. Rechnung 1 bis 5.000 generieren eine grüne Try.success-Kiste. Rechnung 5.001 löst den Netzwerkfehler aus, aber die exportiereSicher-Methode fängt ihn ab, packt ihn in eine rote Try.failure-Kiste und legt diese auf das Band. **Das Band läuft weiter!** Rechnung 5.002 bis 10.000 werden ganz normal verarbeitet und generieren wieder grüne Kisten.

Am Ende der Pipeline (der Senke) haben wir eine saubere Liste von 10.000 Try-Objekten. Mit Collectors.partitioningBy trennen wir die Spreu vom Weizen. Wir haben 9.999 erfolgreiche Exporte und genau 1 Fehlerprotokoll. Die Buchhaltung wird nicht

lahmgelegt, und die IT-Abteilung weiß exakt, welche einzelne Rechnung am nächsten Morgen neu angestoßen werden muss.

Fazit der Iteration 6

Mit der Integration der Try-Monade (oder alternativ dem Either-Pattern) haben wir den letzten großen Störfaktor für funktionale Architekturen in Java eliminiert. Wir haben gelernt, dass Exceptions in Streams kein Hindernis sein müssen, wenn wir aufhören, sie als Kontrollfluss-Abbruch zu betrachten, und stattdessen anfangen, sie als ganz normale Daten zu modellieren.

Unser J-Shop ist nun unfassbar robust. Die Bänder laufen performant, die Zustände sind unveränderlich, Null-Werte sind typsicher gekapselt und selbst Netzwerk-Ausfälle können das System nicht mehr zum Absturz bringen.

Wir stehen kurz vor dem Abschluss unseres Projekts. Im nächsten und letzten Kapitel werden wir die Grenzen unseres funktionalen Kerns klar definieren und die allerletzten Lücken in unserer Fachlogik durch moderne Java-Features (Pattern Matching) unwiderruflich versiegeln.

Schlagen Sie das finale Kapitel auf. Willkommen in **Iteration 7 – Der moderne Kern (Pattern Matching & Architektur)**.

21 Iteration 7 – Der moderne Kern (Pattern Matching & Architektur)

Willkommen zur finalen Iteration unseres Projekts! Wenn wir auf den alten J-Shop aus Iteration 1 zurückblicken, erkennen wir ihn kaum noch wieder. Wir haben eine rohe, fehleranfällige und schwerfällige Anwendung in eine hochmoderne Datenfabrik verwandelt. Unsere Fließbänder (Streams) arbeiten in perfekter Harmonie mit unveränderlichen Bauteilen (Records), fangen fehlende Daten elegant ab (Optional) und lassen sich selbst durch Netzwerkfehler nicht aus der Ruhe bringen (Try).

Doch ein letztes architektonisches Puzzleteil fehlt uns noch. Im Herzen unserer Fabrik – dem **funktionalen Kern** – müssen wir Entscheidungen auf Basis von Zuständen treffen. In diesem letzten Kapitel versiegeln wir unsere Fachlogik so stark, dass der Java-Compiler selbst zu unserem strengsten Qualitätskontrolleur wird und illegale Zustände im System physisch unmöglich macht.

21.1 Bestell-Zustände (Offen, Bezahlt, Storniert) als Sealed Classes modellieren

In fast jedem E-Commerce-System durchläuft eine Bestellung verschiedene Lebenszyklen. Im alten J-Shop wurde der Status einer Bestellung als einfacher String gespeichert:

```
// Der alte J-Shop: Fehleranfällig und ohne Struktur
public class Bestellung {
    private String status; // "OFFEN", "BEZAHLT" oder "STORNIERT"
    private String transaktionsId;
                        // Darf nur gefüllt sein, wenn status == "BEZAHLT"
    private String stornoGrund;
                        // Darf nur gefüllt sein, wenn status == "STORNIERT"

    // ... Getter und Setter ...
}
```

Dieses Design ist eine offene Einladung für Fehler. Was passiert, wenn ein Entwickler den Status "Bezahlt" (mit kleinem 'e') in die Datenbank schreibt? Was passiert, wenn eine Bestellung den Status "OFFEN" hat, aber versehentlich ein stornoGrund gesetzt wurde? Wir haben hier Zustände, die fachlich völlig illegal sind, vom Code aber problemlos repräsentiert werden können.

In der funktionalen Programmierung gilt das Prinzip: **Make illegal states unrepresentable** (Mache illegale Zustände unrepräsentierbar).

Wir nutzen dafür **Algebraische Datentypen (ADTs)**, die Java durch **Sealed Classes** (versiegelte Klassen) und Records zur Verfügung stellt. Wir definieren den Zustand nicht mehr als simplen Text, sondern als eine geschlossene Hierarchie von Datentypen, bei der jeder Zustand nur exakt die Felder besitzt, die er auch wirklich braucht.

```
// Wir versiegeln das Interface. Nur die genannten drei Records dürfen es implementieren!
public sealed interface BestellStatus permits Offen, Bezahlt, Storniert {}

// Die konkreten Ausprägungen (Records aus Iteration 4)
public record Offen() implements BestellStatus {}

public record Bezahlt(String transaktionsId, LocalDateTime zahlungsDatum) implements BestellStatus {}

public record Storniert(String grund) implements BestellStatus {}
```

Unsere Bestellung (die natürlich ebenfalls ein Record ist) hält nun ein Feld vom Typ BestellStatus.

Eine Bestellung, die "Offen" ist, *kann* technisch gar keine Transaktions-ID mehr haben, da der Record Offen dieses Feld nicht besitzt. Der illegale Zustand ist auf Compiler-Ebene unmöglich geworden!

21.2 Vollständiges Pattern Matching im Rechnungs-Service

Jetzt, da wir unsere Zustände versiegelt haben, müssen wir in unserer Geschäftslogik darauf reagieren. Wenn eine Bestellung verarbeitet wird, müssen wir je nach Status eine andere Aktion ausführen (z. B. eine E-Mail generieren).

Im alten Java hätten wir hierfür zahllose instanceof-Prüfungen und hässliche Type-Casts (Typumwandlungen) schreiben müssen. Seit Java 21 bietet uns die Sprache jedoch **Pattern Matching für switch** – das ultimative Werkzeug für funktionale Fallunterscheidungen.

Wir bauen einen Rechnungs-Service, der auf Basis des Status eine Nachricht für den Kunden generiert:

```
public class RechnungsService {

    // Eine reine Funktion im funktionalen Kern
    public String generiereKundenNachricht(BestellStatus status) {

        // Das funktionale switch-Statement als Ausdruck (Expression)
```

```
        return switch (status) {
            case Offen o      -> "Ihre Bestellung ist eingegangen und " +
                                 "wird bald bearbeitet.";
            case Bezahlt b    -> "Danke! Ihre Zahlung (ID: " +
                                  b.transaktionsId() + ") vom " +
                                  b.zahlungsDatum() + " ist eingegangen.";
            case Storniert s  -> "Die Bestellung wurde leider storniert. " +
                                 " Grund: " + s.grund();
        };
    }
}
```

Die Magie der Exhaustiveness (Vollständigkeitsprüfung):

Betrachten Sie den switch-Block genau. Fällt Ihnen auf, dass etwas fehlt? Es gibt keinen default-Zweig!

Warum lässt der Compiler das zu? Weil das Interface BestellStatus *versiegelt* (sealed) ist. Der Compiler weiß mit hundertprozentiger Sicherheit, dass es nur die Typen Offen, Bezahlt und Storniert gibt. Wenn wir alle drei Fälle abgedeckt haben, ist der switch-Ausdruck "exhaustiv" (vollständig).

Stellen Sie sich vor, das Marketing führt in sechs Monaten einen neuen Status ein: public record Versendet(String trackingLink) implements BestellStatus {}.

In dem Moment, in dem der Kollege diese Zeile Code hinzufügt, wird unser RechnungsService sofort rot aufleuchten und die Kompilierung verweigern. Der Compiler ruft uns zu: *"Achtung! Du hast den Fall 'Versendet' in deinem switch vergessen!"*

Das ist die Krönung der funktionalen Architektur. Wir müssen keine hundert Unit-Tests mehr schreiben, um zu prüfen, ob ein neuer Status das System zum Absturz bringt. Das Typsystem fängt den Fehler ab, bevor das Programm überhaupt gestartet wird.

Wir haben unseren funktionalen Kern isoliert. Das Generieren der Nachricht (die reine Funktion) ist völlig getrennt vom Versenden der Nachricht (dem Seiteneffekt). Die "Imperative Schale" unserer Architektur nimmt diesen generierten String entgegen und kümmert sich um den I/O-Aufwand (das tatsächliche Senden der E-Mail über das Netzwerk).

21.3 Fazit des E-Commerce-Projekts: Vom Spaghetti-Code zur eleganten Datenfabrik

Treten Sie einen Schritt zurück und betrachten Sie, was Sie in diesen sieben Iterationen geleistet haben.

Wir haben einen J-Shop übernommen, der durch imperativen Zustand, manuelle Schleifen, versteckte Mutationen und ein konstantes Risiko von Null- und Netzwerk-Abstürzen geprägt war.

- Wir haben mit **Lambdas** das Verhalten flexibel gemacht.
- Wir haben mit der **Stream-API** die "for-Schleifen-Ketten" durch hochperformante, parallele Datenfließbänder ersetzt.
- Wir haben den Zustand in **Java Records** eingefroren und damit "Shared Mutable State" aus der Domäne verbannt.
- Wir haben fehlende Werte mit **Optional** und Fehler mit **Try** explizit gemacht.
- Und wir haben mit **Sealed Classes und Pattern Matching** einen funktionalen Kern geschaffen, der illegale Zustände unmöglich macht.

Unser neuer J-Shop ist nicht nur kürzer und weitaus lesbarer geworden. Er ist extrem robust und von Natur aus bereit für die moderne Hardware-Welt. Da wir Zustand nicht mehr mutieren, können wir jederzeit parallelStream() aufrufen und unsere Berechnungen gefahrlos auf 64 Prozessorkerne verteilen.

Ein letztes Wort zum Abschluss

Wenn Sie schon einmal mit Java-Thread nebenläufig oder parallel programmiert haben, wissen Sie, wie schmerzhaft und komplex die Verwaltung von Locks, Monitoren und synchronisiertem Zustand sein kann. Die funktionale Programmierung, die Sie in diesem Buch erlernt haben, ist die ultimative Antwort auf diese Komplexität. Wenn es keinen veränderlichen Zustand gibt, gibt es keine Race Conditions. Wenn Funktionen referenziell transparent sind, können sie in beliebiger Reihenfolge und auf beliebigen Threads ausgeführt werden.

Die Umstellung vom imperativen "Wie" zum deklarativen "Was" erfordert anfangs Geduld und Disziplin. Ihre ersten Stream-Pipelines werden sich ungewohnt anfühlen. Doch sobald Sie den Schalter im Kopf umgelegt haben, werden Sie nie wieder zu den endlosen if-else-Kaskaden und for-Schleifen zurückkehren wollen.

Sie beherrschen nun die Werkzeuge, um Java-Anwendungen zu schreiben, die nicht nur funktionieren, sondern die sich wie flüssiger, sicherer Text lesen lassen. Nutzen Sie dieses Wissen, um Datenfabriken zu bauen, auf die Sie stolz sein können!

Viel Erfolg auf Ihrem weiteren Weg als moderner, funktionaler Java-Architekt.