Funktionale Programmierung für Java-Entwickler

Dominik Schlosser

ISO Software Systeme GmbH

Über mich

- Software-Entwickler bei ISO Software Systeme GmbH
- Java EE, Spring, Angular
- Privat u.a. Haskell, Elm, Scala

NOTE

Ich arbeite als externer Consultant vor allem im Enterprise Java Umfeld. Privat beschäftige ich mich seit einigen Jahren mit funktionaler Programmierung - u.a. in Haskell, Elm und auch Java.
Dieses Wissen konnte ich auch mehrfach im Job einsetzen und damit Erfahrungen in echten Projekten sammeln.

Dieses Dokument wurde direkt aus den Vortragsfolien generiert. Allerdings sind im Vortrag einige Live-Coding Abschnitte enthalten, deren Inhalt hier nicht wiedergegeben wird.

Die ISO-Gruppe

NOTE

Die ISO-Gruppe (gegründet 1979) ist eine mittelständische Firmengruppe (ca. 500 Mitarbeiter) mit Hauptsitz in Nürnberg. Weitere Standorte befinen sich verteilt über Deutschland sowie in Österreich, Polen und Kanada.
Ich arbeite in der ISO Software Systeme GmbH im Bereich 27 - Öffentliche Verwaltung.

Lange Zeit eine Randerscheinung

NOTE

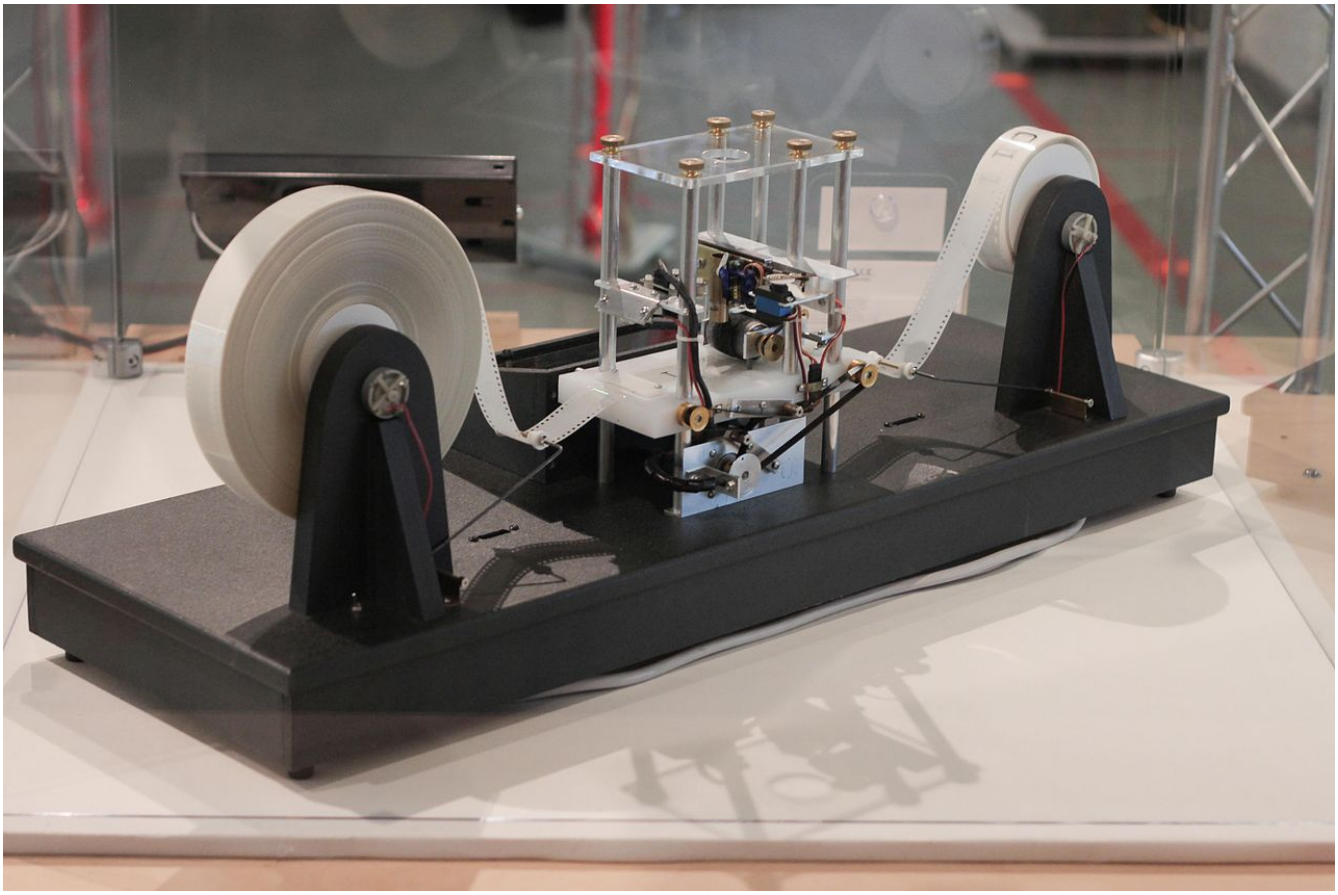
Funktionale Programmierung konnte sich lange Zeit außerhalb akademischer Anwendungsbereiche nicht durchsetzen.
Das ändert sich allerdings seit einiger Zeit und ein Ziel dieses Vortrags ist es, Euch näher zu bringen, warum das so ist.
Wir werden an ein paar einfachen Beispielen sehen, worum es bei funktionaler Programmierung geht, welche Probleme es löst und wie man funktionale Ansätze gewinnbringend in einem ansonsten objektorientierten Java-Programm einsetzen kann.

Alan Turing

> Anything that is "effectively" computable can be computed by a **Universal Turing Machine**
>
> — Alan Turing

NOTE Die Turing Maschine wurde 1936 von Alan Turing als ein mathematisches Modell entwickelt. Mit dieser Maschine lässt sich alles berechnen, was "effektiv berechenbar" ist.

Turing-Maschine



By Rocky Acosta - Own work, CC BY 3.0, https://commons.wikimedia.org/w/index.php?curid=24369879

NOTE

Eine Turing Maschine ist sehr simpel aufgebaut. Es gibt ein Band, das in Zellen aufgeteilt ist und einen Schreibkopf, der Symbole auf das Band schreiben sowie das Band nach vorn und hinten weiterbewegen kann.
Zusätzlich gibt es eine Tabelle mit Instruktionen (bewege das Band, schreibe auf das Band, lösche eine Zelle), die abhängig vom aktuellen Zustand ausgeführt werden.

Dieses Modell (lesen/schreiben von Speicherzellen) findet sich später im Von Neumann-Computer sowie den meisten Mainstream-Programmiersprachen (Assembler, C, C++, Java, ...) wieder.

Alonzo Church

> Anything that is "effectively" computable can be computed by **λ-calculus**
>
> — Alonzo Church

NOTE

Der sog. "Lambda-Kalkül" wurde von Alonzo Church den 1930ern entwickelt. Es handelt sich hierbei um eine formale Sprache, die Berechnungen und Logik vollständig mittels mathematischer Funktionen ausdrückt.
Bereits früh wurde entdeckt, dass der Lambda-Kalkül und Turing-Maschinen gleichmächtig sind.

Lambda Calculus

> Lambda calculus (also written as λ-calculus) is a **formal system in mathematical logic** for expressing computation based on **function abstraction** and application using **variable binding and substitution**.
>
> It is a universal model of computation that can be used to simulate any Turing machine.
>
> — https://en.wikipedia.org/wiki/Lambda_calculus

NOTE

Der Lambda-Kalkül drückt alles (selbst Konstanten) als Funktion aus. Durch Bindung freier Variablen (Funktionsabstraktion) entstehen neue Funktionen.
Funktionen selbst können als Argument für andere Funktionen verwendet werden (Funktionen höherer Ordnung).
Diese abstrakt klingenden Konzepte werden wir später in den Beispielen wiederfinden.

Functional goes mainstream

- C#: Lambda-Syntax, LINQ, Pattern Matching
- C++: Lambda-Syntax
- JavaScript: Lambda-Syntax, Bibliotheken wie ramda
- Java: Lambda-Syntax, Stream-API usw seit Version 8

NOTE

Funktionale Anteile finden sich heutzutage in vielen Mainstream-Sprachen wieder. C# war hier mit dem sog. "Language Integrated Query" (LINQ) Vorreiter.
Seit Version 8 gibt es auch in Java eine (eher minimalistische) Sprachunterstützung funktionaler Konzepte.

Beispiel (Imperativ)

```
List<Integer> result = new ArrayList<>();

for(int i = 1; i <= 100; i++){
    if(number % 2 == 0){
        result.add(number);
    }
}

System.out.println(result);
```

NOTE

Ein einfaches, imperatives Beispiel.
Ähnlichen Code hat jeder schon hunderte Male geschrieben.

Aber wie oft hatten wir schon +/-1 Fehler in for-Schleifen? Und wird hier wirklich die Intention klar?

Im imperativen Ansatz beschreiben wir WIE wir etwas tun, nicht WAS wir tun.

Beispiel (Funktional)

```
List<Integer> numbers = IntStream.rangeClosed(1, 100);

input.filter(number -> number % 2 == 0)
     .forEach(System.out::println);
```

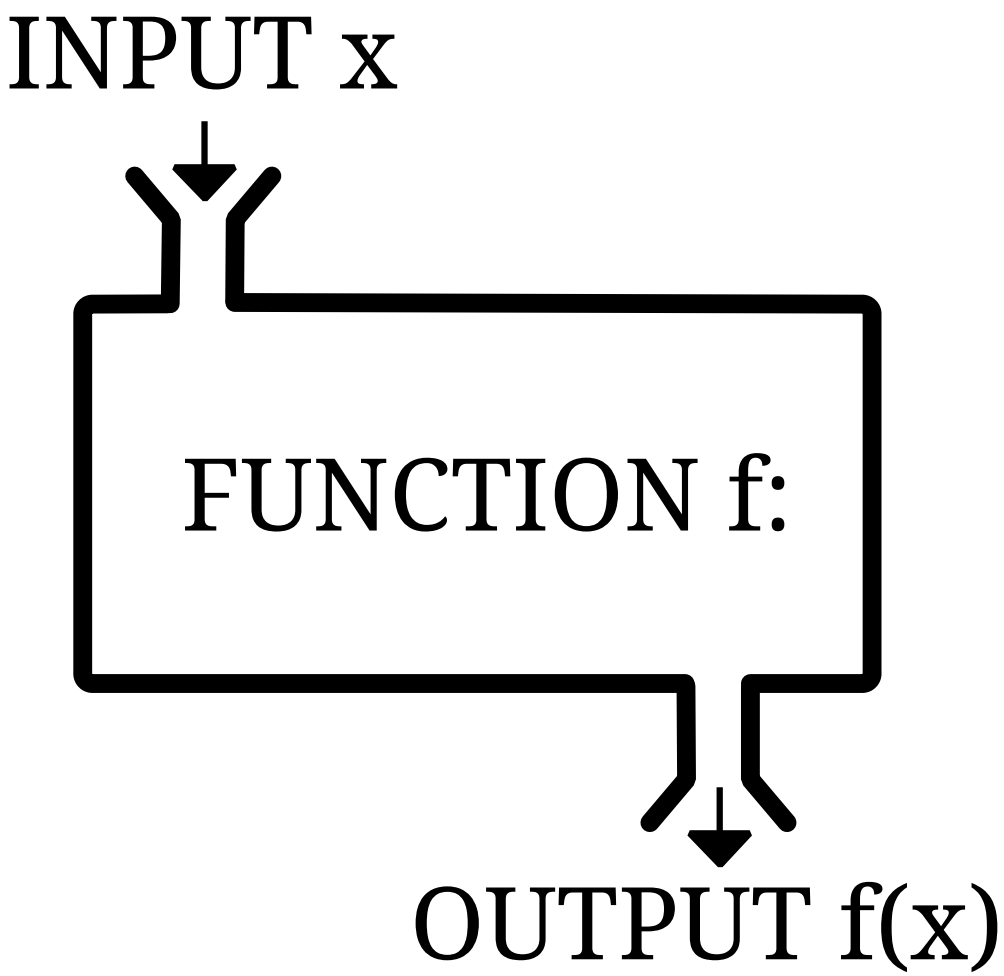
Mit obligatorischem Java-Overhead

```
List<Integer> numbers = IntStream.rangeClosed(1, 100)
                                 <strong>.mapToObj(number -> number)
                                 .collect(Collectors.toList())</strong>;

input<strong>.stream()</strong>
     .filter(number -> number % 2 == 0)
     .forEach(System.out::println);
```

NOTE

Das gleiche Beispiel, diesmal funktional. Erst wie es sein sollte und dann wie es in Java wirklich ist.
Hier werden keine temporären Variablen benötigt. Es wird nicht beschrieben WIE das Filtern funktioniert, sondern lediglich nach WAS ich filtern möchte.

Was ist eine Funktion?



Quelle: Wikipedia

NOTE

Eine Funktion hat folgende Eigenschaften:
* Gleicher Input → gleicher Output (referentielle Transparenz)
* Sowohl Input als auch Output können selbst wieder Funktionen sein (damit kann man bspw. jede Funktion als Funktion mit nur einem Argument auffassen, die eine Funktion mit den übrigen Argumenten zurückgibt)
* Eine Funktion hat keine Seiteneffekte!

Seiteneffekte

Nicht-lokale Zuweisungen

```
public class Product {
    private double price;

    public void applyTax(double taxPercent){
        this.price = this.price + (this.price * taxPercent / 100);
    }
}
```

Immutability

```
public class Product {
    private double price;

    public Product applyTax(double taxPercent){
        return new Product(this.price + (this.price * taxPercent / 100));
    }
}
```

NOTE

Seiteneffekte reichen von nicht-lokalen Variablenzuweisungen (Änderung eines Zustands) über Konsolenausgaben bis zu Änderungen in einer Datenbank oder Aufrufen von Webservices.
All das wollen wir in puren Funktionen nicht haben.
Zustandsänderungen können über das Prinzip unveränderlicher Daten verhindert werden. Anstatt einen Wert zu ändern, wird hier ein neuer Wert erzeugt.
Das ist hinsichtlich Speicherbedarf und Performance zunächst mal nachteilig, bringt uns aber auch deutlich leichtere Verständlichkeit auch komplexer Programme.
Den Nachteilen kann bspw. über persistente Datenstrukturen (bspw. Linked Lists) entgegengewirkt werden.

Bonusfrage: Was ist an dem Beispiel (neben der Zustandsänderung) noch problematisch?

Seiteneffekte in rein funktionalen Sprachen

Quelle: https://guide.elm-lang.org/architecture/effects/

NOTE

Die Elm-Architektur zeigt, wie man in der rein-funktionalen Welt mit Seiteneffekten umgehen kann:
Anstatt etwas seiteneffektbehaftetes zu tun, wird ein Command an die Runtime übergeben. Diese führt den Command (bspw. Webservice-Call) aus und erzeugt aus dem Ergebnis eine Message, auf die das Programm reagieren kann.

"Command Pattern"

```
public interface Command<T> {
    /**
     * ACHTUNG: Hat potenziell Seiteneffekte!
     */
    T execute();
}

public class LoadProductsCommand implements Command<Product> {
    private int productId;

    public LoadProductsCommand(int id) { productId = id; }

    public Product execute() {
        return loadFromDb(id);
    }
}
```

NOTE Als Java-Klasse kann man sich das Ganze wie ein Command-Pattern vorstellen: Durch die Erzeugung des Commands geschieht noch nichts aber das Command-Objekt beschreibt, was geschehen soll.

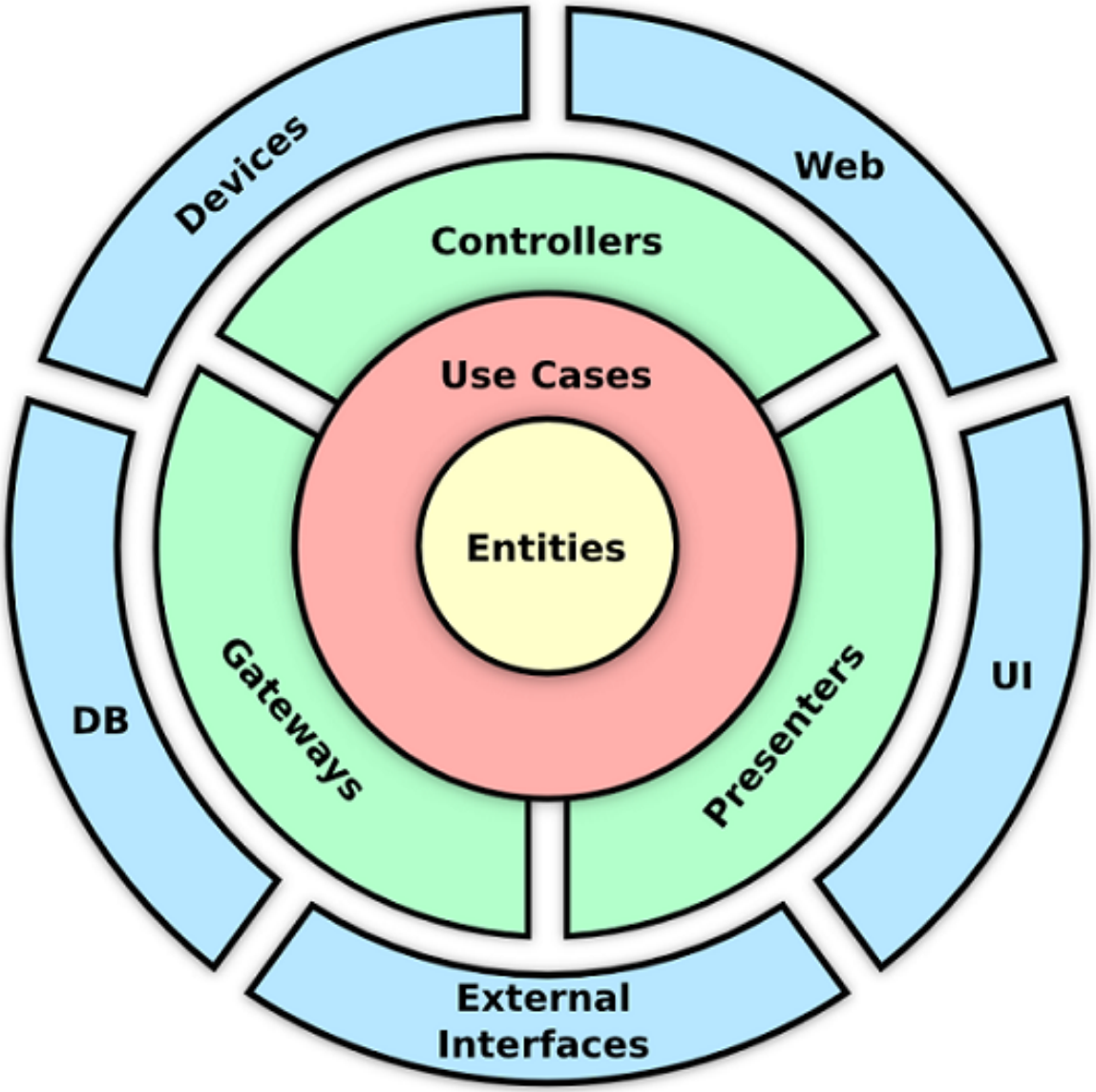
Die Runtime (konzeptionell)

```
public interface Runtime {
    Message dispatch(Command<T> command);
}

// ...

Message message = runtime.dispatch(loadProductsCommand);

update(message);
```

NOTE

So könnte man sich sowas wie die Elm-Runtime in Java konzeptionell vorstellen:
Sie nimmt Commands entgegen, führt sie aus und erzeugt aus dem Ergebnis (kann auch ein Fehler sein) ein Command.
Die Message wird dann an die Update-Funktion übergeben, die das Programm bereitstellt.

Funktionaler Kern



NOTE

In nicht rein-funktionalen Sprachen hat sich dazu ein Architekturpattern namens "Clean Architecture" oder auch "Ports and Adapters" bzw. "Hexagonale Architektur" etabliert:
Seiteneffekte werden dabei möglichst an den Systemrand gedrängt, in dem der rein-funktionale Kern nur mit Abstraktionen (Ports) arbeitet, die dort über Adapter implementiert werden.

Umgang mit Fehlern

```
public Product loadProduct(int id) {
    try {
        return loadProductFromDb(id);
    } catch (Exception e) {
        LOGGER.error(e);
        throw new RuntimeException(e);
    }
}
```

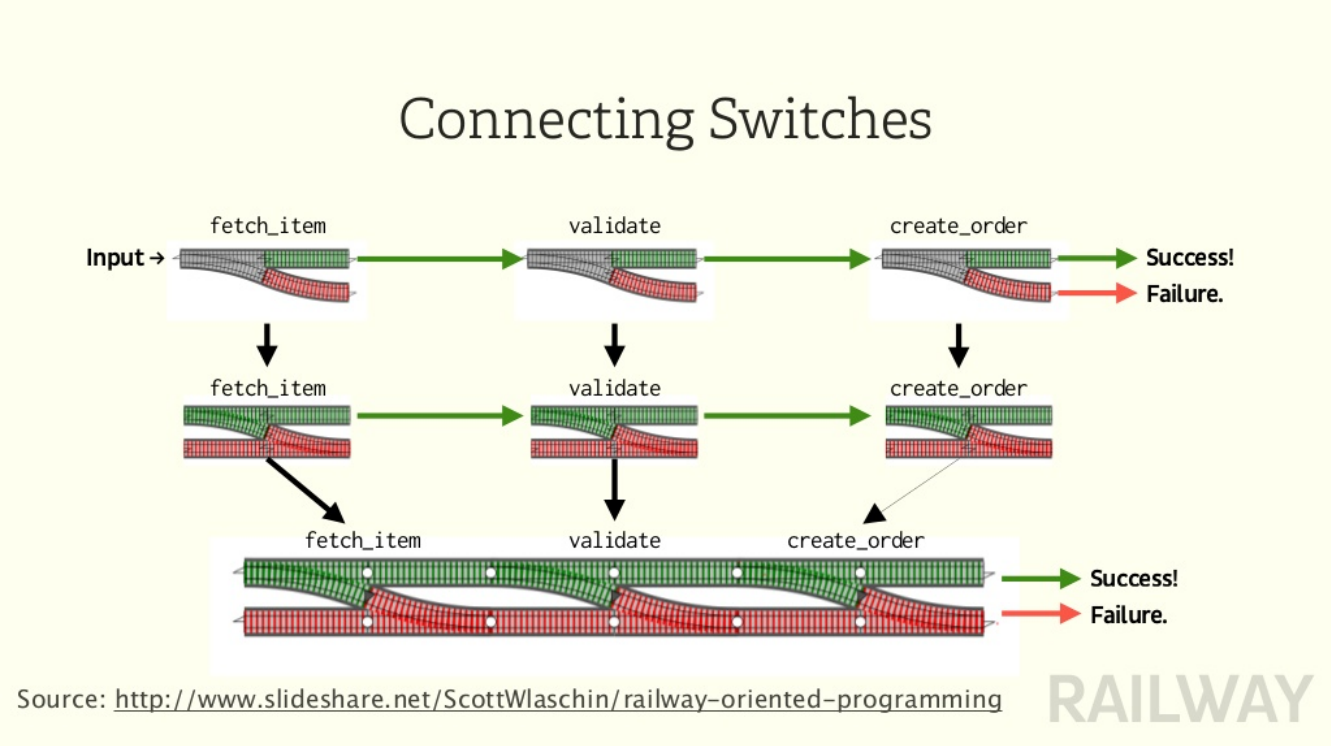
Wenn das Fehlerhandling alles übertönt

```
Order placeOrder(String itemId){
    Item item = fetchItem(itemId);
    if(item != null) {
        ValidationResult validationResult = validate(item);
        if(ValidationResult.OK.equals(validationResult)){
            return createOrder(item);
        } else {
            throw new ValidationException("Validation for Item with id " + itemId + "
failed");
        }
    } else {
        throw new ItemNotFoundException("Item with id " + itemId + " not found");
    }
}
```

NOTE

Fehlerhandling ist in objektorientierten Sprachen oft sehr dominant. Man sieht vor lauter Fehlerbehandlungscode nicht mehr, was der Code eigentlich im Normalfall tun soll.
Zudem werden Fehler häufig in jeder Schicht / in jeder Klasse der Aufrufkette "behandelt", da der Entwickler sich nicht darauf verlassen kann/will, dass es an der Systemgrenze eine Fehlerbehandlung gibt.

Railway-oriented Programming



NOTE Das sog. "Railway-oriented Programming" (Scott Wlaschin) ist eine Möglichkeit, Fehlerbehandlung typsicher und unaufdringlich in den Code zu integrieren.

Monaden: Ein Design Pattern

- Monaden sind **Container** für Werte
- Sie abstrahieren über den Zugriff auf diese Werte
- Zwei Bestandteile: unit (Erzeugung) und bind (Mapping des enthaltenen Werts)

Monaden: Beispiel

```
double divide(int dividend, int divisor){
    return (double) dividend / divisor;
}
```

```
Optional<Double> divide(int dividend, int divisor){
    if(divisor == 0){
        return Optional.empty();
    }

    return Optional.of((double) dividend / divisor);
}
```

```
String result = divide(5, 2)
                    .map(result -> "Das Ergebnis ist: " + result)
                    .orElse("Es konnte kein Ergebnis ermittelt werden");

System.out.println(result);
```

NOTE

Monaden sind lediglich ein Design Pattern mit ein paar besonderen Eigenschaften, die unproblematische Komposition ermöglichen.
Sie abstrahieren über den Zugriff, auf die enthaltenen Werte: * Optional kann bspw. keinen Wert enthalten
* Die Werte in Stream werden erst bei Zugriff erzeugt (lazy)
* Ein Try kann statt einem Wert einen Fehlerzustand enthalten
* Ein Observable kann die enthaltenen Werte noch nicht erhalten haben

usw.

Weiterführende Konzepte

Going forward

- Frameworks wie jooλ oder vavr
- Pattern Matching in zukünftiger Java-Version
- Alternative JVM-Sprachen wie Scala, Kotlin, Clojure, …

NOTE	Im Folgenden werden ein paar weiterführende Konzepte und zusätzliche Frameworks vorgestellt.

Pattern Matching

```
abstract class Expr {}

class Num extends Expr {
    private int value;
}

class Sum extends Expr {
    private Expr left;
    private Expr right;
}

class Prod extends Expr {
    private Expr left;
    private Expr right;
}
```

NOTE

Pattern Matching ist ein wichtiger Bestandteil funktionaler Sprachen, der allerdings in Java bisher nicht umgesetzt wurde.
Im Rahmen von "Project Amber" soll Java um Pattern Matching erweitert werden:
http://openjdk.java.net/jeps/305

instanceof-Kaskaden

```
if(x instanceof Num){
    Num num = (Num)x;
    // ...
} else if (x instanceof Sum){
    Sum sum = (Sum)x;
    // ...
} else if (x instanceof Prod){
    Prod prod = (Prod)x;
    // ...
}
```

NOTE

Pattern Matching ist dann wichtig, wenn man abhängig von dem Typ, den ein Objekt hat, etwas tun möchte.
Ohne Pattern Matching bleibt einem hier nur alle Möglichkeiten per instanceof zu prüfen. Der primäre Nachteil hierbei ist, dass der Compiler nicht prüft, ob es weitere Fälle geben könnte.
Dies ist insbesondere dann problematisch, wenn eine neue Möglichkeit hinzukommt.

Visitor Pattern

```
interface AstVisitor {
    void visit(Num num);
    void visit(Sum sum);
    void visit(Prod prod);
}

abstract class Expr {
    abstract void accept(AstVisitor visitor);
}

class Num extends Expr {
    void accept(AstVisitor visitor) {
        visitor.visit(this);
    }
}
```

NOTE Dies lässt sich über das Visitor Pattern lösen. Diese Lösung bietet sich bei sowas wie AST-Parsing an, führt sonst aber zu ziemlich viel Overhead.

Pattern Matching - Scala

```
def evalExpr(e: Expr): Int = e match {
  case Num(n) => n
  case Sum(l, r) => evalExpr(l) + evalExpr(r)
  case Prod(l, r) => evalExpr(l) * evalExpr(r)
}
```

NOTE

Funktionale Sprachen bieten hierfür "Pattern Matching".
Man kann sich das wie ein stark erweitertes switch-Statement vorstellen:
* Der Compiler prüft "exhaustiveness", d.h. ob alle möglichen Fälle deklariert sind
* Im Case-Teil befindet sich ein "Pattern", d.h. hier wird beschrieben wie der zu matchende Fall aussieht. Hierüber werden also Bestandteile des Typs "extrahiert".

Pattern Matching - vavr

```
public int evalExpr(Expr expr) {
    return Match(expr).of(
        Case($(instanceOf(Num.class)), num -> num.getValue()),
        Case($(instanceOf(Sum.class)), sum -> evalExpr(sum.getLeft()) + evalExpr(sum
.getRight()),
        Case($(instanceOf(Prod.class)), prod -> evalExpr(prod.getLeft()) * evalExpr
(prod.getRight()),
        Case($(), o -> { throw new IllegalStateException(expr); })
    );
}
```

NOTE

Java als Sprache beherrscht kein Pattern Matching. Über Bibliotheken wie "vavr" lässt es sich emulieren, wobei es hier natürlich Einschränkungen gibt. Statt eines Exhaustiveness-Checks des Compilers gibt es hier bspw. einen "Else"-Zweig, in dem eine Exception geworfen wird.

Pattern Matching - Java vFuture

```
public int evalExpr(Expr expr) {
    switch(expr){
        case Num(int v): return v;
        case Sum(Expr l, Expr r): return evalExpr(l) + evalExpr(r);
        case Prod(Expr l, Expr r): return evalExpr(l) * evalExpr(r);
        default: throw new IllegalStateException(expr);
    }
}
```

NOTE

Es ist sehr wahrscheinlich, dass Java Pattern Matching in zukünftigen Versionen umsetzen wird. Wie es genau aussehen wird, ist allerdings schwer vorhersagbar. Laut dem offiziellen JEP und Aussagen von Brian Goetz wird der Support in mehreren Stufen erfolgen, wobei das Switch-Statement erweitert werden soll.

Fehlende Features mit jool nachrüsten

```
// (1, 2, 3, 4, 5, 6)
Seq.of(1, 2, 3).concat(Seq.of(4, 5, 6));

// (tuple(1, "a"), tuple(2, "b"), tuple(3, "c"))
Seq.of(1, 2, 3).zip(Seq.of("a", "b", "c"));

// (1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, ...)
Seq.of(1, 2, 3).cycle();

// tuple((1, 3), (2, 4))
Seq.of(1, 2, 3, 4).partition(i -> i % 2 != 0);
```

NOTE jOOL (https://github.com/jOOQ/jOOL) ist eine Bibliothek, die über einen "Seq"-Type viele in Standard-Java (List/Stream) fehlende Funktionen nachrüstet.

Currying mit vavr

```
Function2<Integer, Integer, Integer> sum = (a, b) -> a + b;
Function1<Integer, Integer> add2 = sum.curried().apply(2);

then(add2.apply(4)).isEqualTo(6);
```

Lifting mit vavr

```
Function2<Integer, Integer, Integer> divide = (a, b) -> a / b;

Function2<Integer, Integer, Option<Integer>> safeDivide = Function2.lift(divide);

// = None
Option<Integer> i1 = safeDivide.apply(1, 0);

// = Some(2)
Option<Integer> i2 = safeDivide.apply(4, 2);
```

NOTE

vavr (http://www.vavr.io/) ist eine weitere, populäre Bibliothek für funktionales Programmieren in Java.
Sie liefert viele persistente Datenstrukturen (List, Map, Set, Tree, ...) und jeweils sehr viele Funktionen für die Arbeit mit diesen Datenstrukturen.
Die API orientiert sich dabei sehr stark an Haskell.