funktionale programmierung.
gedankenfutter.

Martin Boßlet

35+ Jahre Programmierer (seit dem 6. Lebensjahr)
20+ Jahre IT-Berufserfahrung
10+ Jahre selbständig als Architekt, Berater, Trainer, Speaker
Java seit 1998
Dipl.-Mathematik/-Informatik
Mitglied im Kernteam der Entwickler der Sprache Ruby
Themen: Java, Spring, Web (Frontend & Backend), Kryptographie, Digitale Signatur, Programmiersprachen

Was ist Funktionale Programmierung?

Funktionale Programmierung

ist die

Zukunft

Doch, steht so im Internet:

https://spectrum.ieee.org/functional-programming
https://github.com/readme/featured/functional-programming
https://www.ing.jobs/deutschland/person/funktionale-programmierung-im-trend.htm

Heißt es seit Jahrzehnten:

`http://www.paulgraham.com/avg.html`

Was ist dran am Hype?

Wir programmieren weiter in

Java, C/++/#, JavaScript, ...

...oder COBOL

Aber FP ist doch viel:

kompakter
wartbarer
schlauer
besser
...

Erklärungsversuche:

Lernkurve

Narzissmus vs. Ignoranz

"Seit Haskell ist alles anders."

Wie ich glaube, dass andere meinen Code sehen:

Wie andere meinen Code sehen:

Funktionale Sprachen

fristen weiterhin ein

Nischendasein

Scala-Meetup während einer Java-Konferenz

Ist FP gescheitert?

Teilerfolg:

Funktionale Aspekte in traditionellen Sprachen

Lambdas

Funktionen als Variablen

Immutable vs Mutable

Funktionales Iterieren (z.B. Java Stream API)

Reactive Programming (z.B. React, Java 9 Flow API, Spring Webflux)

"Die Idee ist gut, doch die Welt noch nicht bereit"

Die gute Nachricht:

Man kann Aspekte von FP
in **jeder** Sprache nutzen

"Boring FP"

Sinnvolle Konstrukte integrieren

Was heißt nun genau "Funktionale Programmierung" ?

https://en.wikipedia.org/wiki/Functional_programming

I. DEKLARATIV

II. PURE FUNKTIONEN OHNE SEITENEFFEKTE

deklarativ

vs.

imperativ, prozedural

imperativ - "Do what I say"

```
public static void main(String[] args) {
    List<String> names = List.of("Anna", "Nadine", "Martin");
    System.out.println(toUpperCase(names));
}

public static List<String> toUpperCase(List<String> strings) {

    List<String> upperCased = new ArrayList<>();

    for (int i=0; i < strings.size(); i++) {
        upperCased.add(strings.get(i).toUpperCase());
    }

    return upperCased;
}
```

deklarativ - "Do what I mean"

```
public static void main(String[] args) {
    List<String> names = List.of("Anna", "Nadine", "Martin");
    System.out.println(toUpperCase(names));
}

public static List<String> toUpperCase(List<String> strings) {
    return strings.stream().map(
        String::toUpperCase
    ).collect(Collectors.toList());
}
```

Warum ist deklarativ besser?

deklarativ vs. imperativ

```java
public static int sizeOf(String s) {
    // Dauert ca. 10s, bis man es verstanden hat
    int len = 0;
    for (char c : s.toCharArray()) {
        len++;
    }
    return len;
}

public static int sizeOf2(String s) {
    // Dauert ca. 0s, bis man es verstanden hat
    return s.length();
}
```

FP bietet

deklarative Funktionen
für
Datenstrukturen

Schweizer Taschenmesser

Baukasten für Algorithmen

Stell dir vor:

Java Stream API,
aber überall

FP ist ein Paradigma,

dass **pure** Funktionen
ohne **Seiteneffekte**
bevorzugt

Unreine Funktion mit Seiteneffekt

```
private interface PaymentService {
    void charge(int amount);
}

private interface Item {
    int getPrice();
}

public static void buyItems(List<Item> items) {
    int sum = items.stream().map(Item::getPrice).reduce(0, Integer::sum);
    PaymentService payPal = new PayPal();
    payPal.charge(sum);
}
```

Nachteile:

Tight Coupling
Schwer zu testen

Flexibler, aber weiterhin unrein: Dependency Injection

```java
public static void buyItems(List<Item> items, PaymentService paymentService) {
    int sum = items.stream().map(Item::getPrice).reduce(0, Integer::sum);
    paymentService.charge(sum);
}
```

Merke:

void ist immer **unrein**

Pure Funktion ohne Seiteneffekt

```
// Teaser: Charge ist immutable
public static class Charge {
    public Charge(List<Item> items, PaymentService paymentService) {
        this.items = new ArrayList<>(items); // Kopie!
        this.paymentService = paymentService;
    }

    private final List<Item> items;
    private final PaymentService paymentService;

    public List<Item> getItems() { return items; }

    public PaymentService getPaymentService() { return paymentService; }

    public Charge combineWith(Charge other) { ... };
}


public static Charge buyItems(List<Item> items, PaymentService paymentService) {
    return new Charge(items, paymentService);
}
```

Vorteil:

Referenzielle Transparenz

Ein Ausdruck e ist referenziell transparent,
wenn für alle Programme p
alle Vorkommen von e in p ersetzt werden können
durch die Evaluation von e,
ohne das Verhalten von p zu verändern

Substitutionsmodell

Wenn f(x) = c

Dann: Überall, wo f(x) steht,
kann ich es durch c ersetzen

Pure Funktion

<->

Mathematische Funktion

f: A -> B

Immer ein Rückgabewert

Deterministisch (gleicher Input -> gleicher Output)

Ohne Seiteneffekte

Verändert **bestehende** Daten nicht

(-> **Immutability**)

Verstehen, **Debuggen** und **Warten**
von Code

EINFACHER

Aber ich brauche doch Seiteneffekte?

FP kann auch Seiteneffekte

aber sparsam

FP gruppiert Code

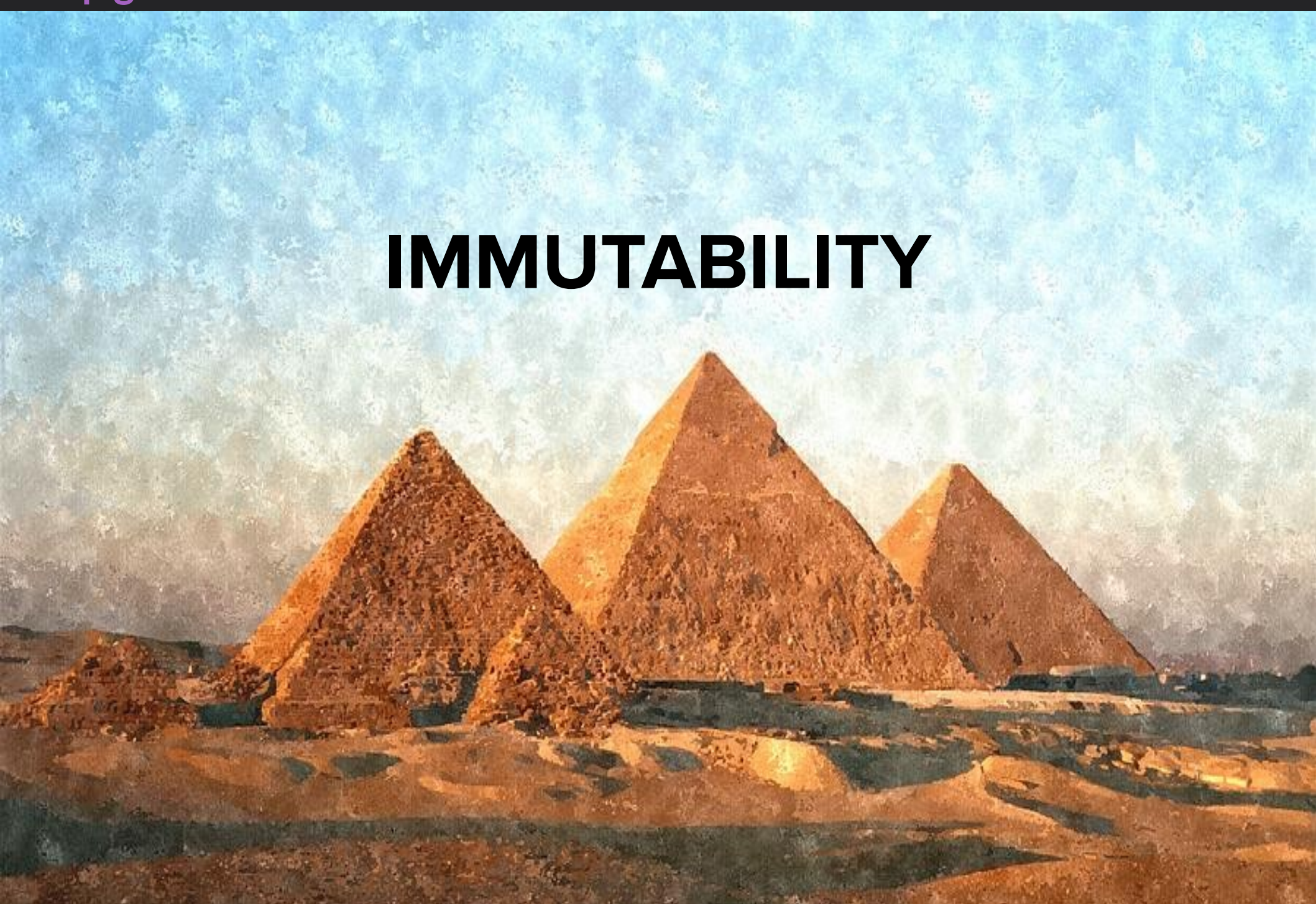
Daten	Berechnungen	Aktionen
viel	viel	wenig

Daten sind **Datenstrukturen,**
auf denen
Berechnungen & Aktionen
durchgeführt werden

Berechnungen sind
seiteneffektfreie
Algorithmen
auf Daten

Aktionen
haben Seiteneffekte

IMMUTABILITY



Pure Funktion
verändert bestehende
Daten **nicht**

Wie setzt man dies um?

Eine Variable
darf nicht erneut
zugewiesen werden

Immutable Variable mit final

```
final String immutable = "Won't change";
immutable = "You will though"; // Compile-time error: Cannot assign a value to final variable 'immutable'
```

Vorsicht: final-Objekte sind nicht automatisch immutable

```
final List<String> list = Stream.of("a", "b", "c").collect(Collectors.toCollection(ArrayList::new));
list.add("d");
System.out.println(list); // [ a b c d ]
```

java.util.List ist mutable

```java
public static void listIsMutable() {
    List<String> mutable = new ArrayList<>();
    mutable.add("a");
}


public interface List<E> extends Collection<E> {

    boolean add(E e); // de-facto void

}
```

Immutable Datenstrukturen
dürfen ihren Zustand
nicht ändern

Immutable Datenstrukturen
müssen bei Änderungen
ein **neues** Objekt allokieren

Einfachster Fall:

Kopie

Immutable mittels Kopie

```
public static <E> List<E> add(List<E> list, E item) {
    List<E> copy = new ArrayList<>(list);
    copy.add(item); // OK, weil von außen nicht feststellbar
    return copy;
}
```

Kopien sind ineffizient

Reduziertes Interface

Immutable vs Mutable

ImmutableList vs MutableList

in Kotlin

(halbherzig: Collections.unmodifiableList in Java)

Umständlich

Immutable
für alles nutzen?

Funktionale/Persistente Datenstrukturen

https://en.wikipedia.org/wiki/Persistent_data_structure

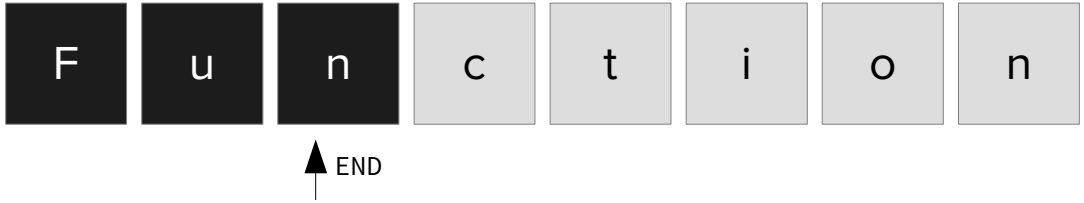
https://www.cambridge.org/core/books/purely-functional-data-structures/0409255DA1B48FA731859AC72E34D494

Structural Sharing

Statt krudem Kopieren
wird geschickt geteilt

String s = "Function"

String t = s.substring(0, 3)




Wirkt alles ein wenig...
akademisch
überdimensioniert?

Strings sind immutable

String Interning

https://en.wikipedia.org/wiki/String_interning

State in React ist immutable

https://react.dev/learn/state-a-components-memory

Immutable.js

State in Vue.js ist es **nicht**

=> Komplexe "Reactive Objects"

https://vuejs.org/guide/extras/reactivity-in-depth.html

Effective Java, Item 17
Minimize Mutability

Immutable objects are simple

(siehe React- vs. Vue-State)

Immutable objects are
inherently thread-safe;
they require no synchronization.

(siehe Kotlin & Clojure vs. Java)

Immutable objects can be shared freely.

(siehe Structural Sharing und Null-Pointer-Safety in Kotlin)

Persistent Data Structures

https://www.vavr.io/
https://immutable-js.com/
https://github.com/Kotlin/kotlinx.collections.immutable

PROGRAMMIERSPRACHEN FÜR FP

Typing	Static	Dynamic
Strong	Haskell **Kotlin** **Scala** OCaml TypeScript ...	**Clojure** Scheme Erlang ...
Weak	C C++	JavaScript

fp.gedankenfutter.

Kotlin

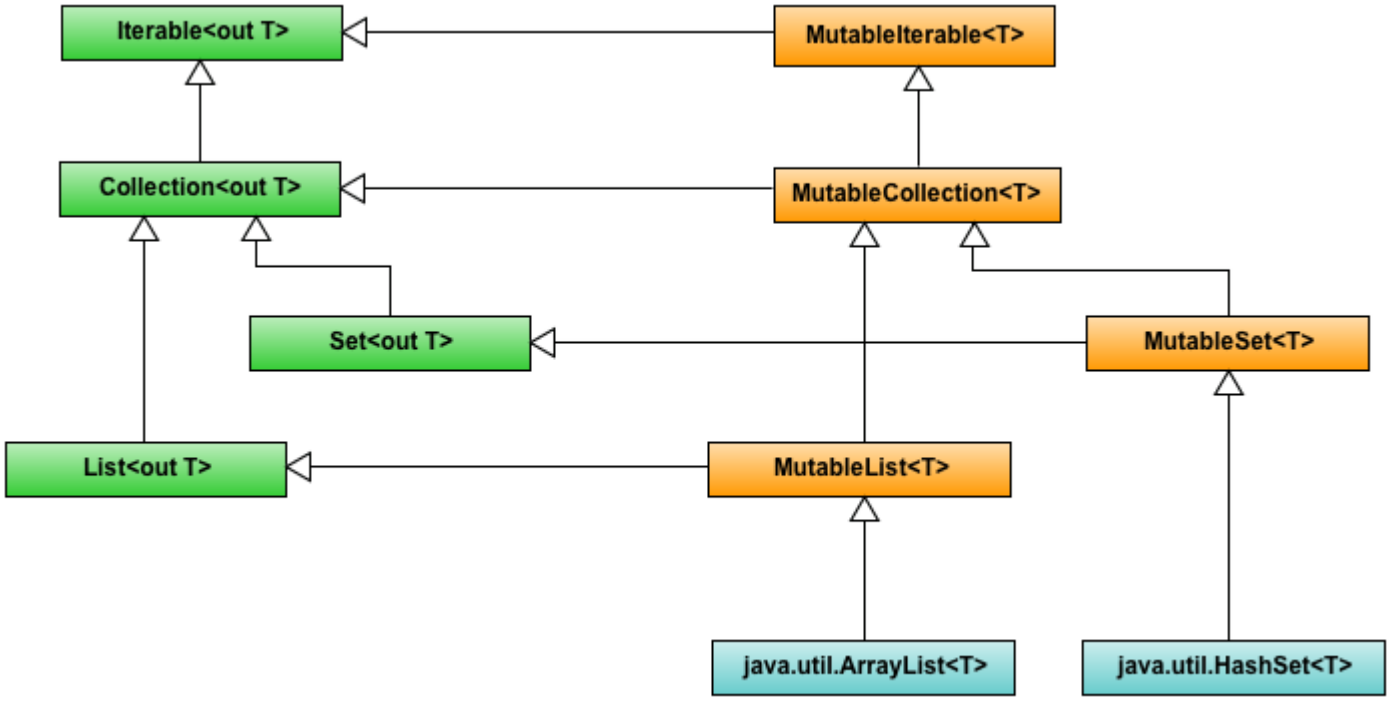
https://kotlinlang.org/
https://play.kotlinlang.org

Immutable Variablen als Default `(val)`

Mutable via `var`

`https://kotlinlang.org/docs/basic-syntax.html#variables`

Immutable vs. Mutable Datenstrukturen



https://kotlinlang.org/docs/collections-overview.html#collection-types

Arrow

https://arrow-kt.io/

"Idiomatic Functional Programming with Kotlin"

Lambdas bei Collections nativ unterstützt

Beispiel:

Alle Wörter aus einer Liste von Dateien als Versalien

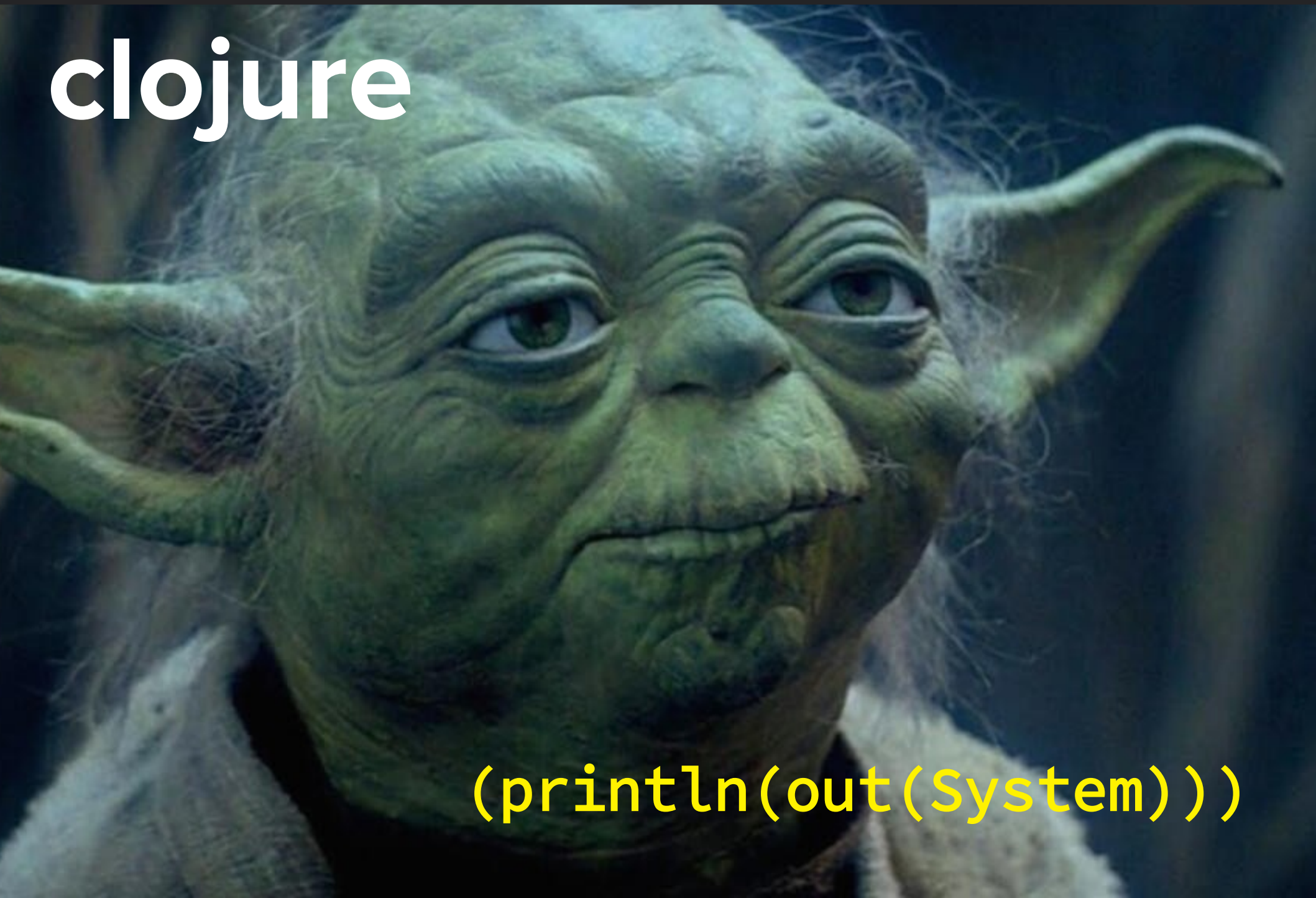
Java: Nervige Checked Exceptions (und mehr...)

```java
public static List<String> upperCaseWords(List<String> files) {
    return files.stream().map((file) -> {
        try (InputStream in = new FileInputStream(file)) {
            return new BufferedReader(new InputStreamReader(in)).lines();
        } catch (IOException ex) {
            throw new RuntimeException(ex);
        }
    }).flatMap((lines) ->
        lines.flatMap(
            (line) -> Stream.of(line.split("\\W"))
        )
    ).map(String::toUpperCase)
    .collect(Collectors.toList());
}
```

Kotlin: Alles sauber

```
fun upperCaseWords(files: List<String>): List<String> {
    return files.map {
        File(it).bufferedReader().readLines()
    }.flatMap { lines: List<String> ->
        lines.flatMap {
            it.split("""\W""".toRegex())
        }
    }.map(String::uppercase)
}
```

clojure



Clojure

https://clojure.org/
https://tryclojure.org/

Lisp auf der JVM

Statt Postfix-Notation

```
s.toUpperCase()
```

oder Infix-Notation

```
5 + 3
```

nun Prefix-Notation:

```
(upper-case s)
(+ 5 3)
```

```
(Stark (sein (du (musst))))
```

Immutable Variablen als Default `(let)`

Mutable als Sonderfälle `(Var, Refs, Agents, Atoms)`

`https://clojure.org/reference/vars`

Funktionale/Persistente Datenstrukturen als Default

https://clojure.org/reference/data_structures

Dynamische Typen sinnvoll eingesetzt:

Heterogene Collections
Seq-Abstraktion

https://clojure.org/reference/sequences


SEQS
SEQS, EVERYWHERE

map, reduce, filter
etc.

operieren auf Seqs

(konkreter Typ zweitrangig)

"It is better to have
100 functions operate on **one** data structure
than to have
10 functions operate on 10 data structures."

- Alan J. Perlis

Lock-free Concurrency

mit
Software Transactional Memory

(STM)

`https://clojure.org/reference/refs`

Clojure: Versalien aus mehreren Dateien

```clojure
(defn upper-case-words [& files]
  (->> files
       (map #(slurp %))
       (mapcat #(str/split % #"\W"))
       (map str/upper-case)))
```

```kotlin
// Pseudo-Kotlin
files.map { slurp(it) }
     .flatMap { it.split("""\W""".toRegex()) }
     .map(String::uppercase)
```

Clojure: Zähle Vokale eines Worts

```clojure
(defn count-vowels [s]
  ;Count the number of vowels in a string
  (let [vowels #{ \A \E \I \O \U }]
    (reduce (fn [acc c]
              (if (contains? vowels c)
                 (inc acc)
                 acc))
            0
            (str/upper-case s))))


// Kotlin
fun vowels(s: String): Int {
    val vowels = setOf('A', 'E', 'I', 'O', 'U')
    return s.uppercase().fold(0) { acc, c ->
        when (c) {
            in vowels -> acc + 1
            else -> acc
        }
    }
}
```

"Na, das sind aber mal viele Klammern..."

the end

Ich danke Ihnen für Ihre Aufmerksamkeit
und wünsche viel Spaß mit FP!

Martin Boßlet

martin.bosslet@gmail.com
https://www.linkedin.com/in/martin-bosslet/