embarc
Software Consulting GmbH

Funktionale Programmierung geht auch mit/trotz Java!

FALK SIPPACH // EMBARC

Karlsruher Entwicklertag 2021

Mittwoch, 09. Juni 2021, 14:15 Uhr

Karlsruher Entwicklertag 2021

Funktionale Programmierung geht auch mit/trotz Java!

Zusammenfassung

Java ist keine funktionale Sprache, aber dank Streams und Lambdas kann man nun seit einiger Zeit auf funktionale Art und Weise programmieren. Reicht das etwa schon, um ausdrucksstärkeren und besser lesbaren Sourcecode zu entwickeln? Passt dieses Programmierparadigma überhaupt zur imperativen Denkweise von uns Java-Entwicklern?

Anhand eines Real-World-Szenarios machen wir uns mit den fehlenden Puzzlestücken der funktionalen Programmierung vertraut. Dabei geht es um Value Types, Pattern Matching, praktische Anwendung von Monaden (Optional, Try, Either, Validierung), Bedarfsauswertung, partielle Funktionsaufrufe, Currying, Funktionskomposition, persistente Datenstrukturen, Seiteneffektfreiheit, referentielle Transparenz und einiges mehr. Wir diskutieren Lösungsansätze in Java und werfen vor allem einen Blick auf nützliche Bibliotheken wie Immutables und Vavr. Denn erst dadurch macht funktionale Programmierung auch in Java wirklich Spass.

Falk Sippach

- Softwarearchitekt, Berater, Trainer bei embarc
- früher bei Orientation in Objects (OIO), Trivadis

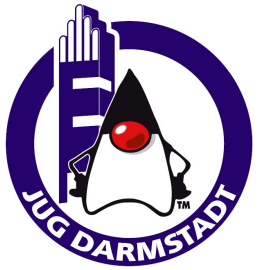
Schwerpunkte:

- Architekturberatung und -bewertung
- Cloud- und Java-Technologien

fs@embarc.de

@sippsack

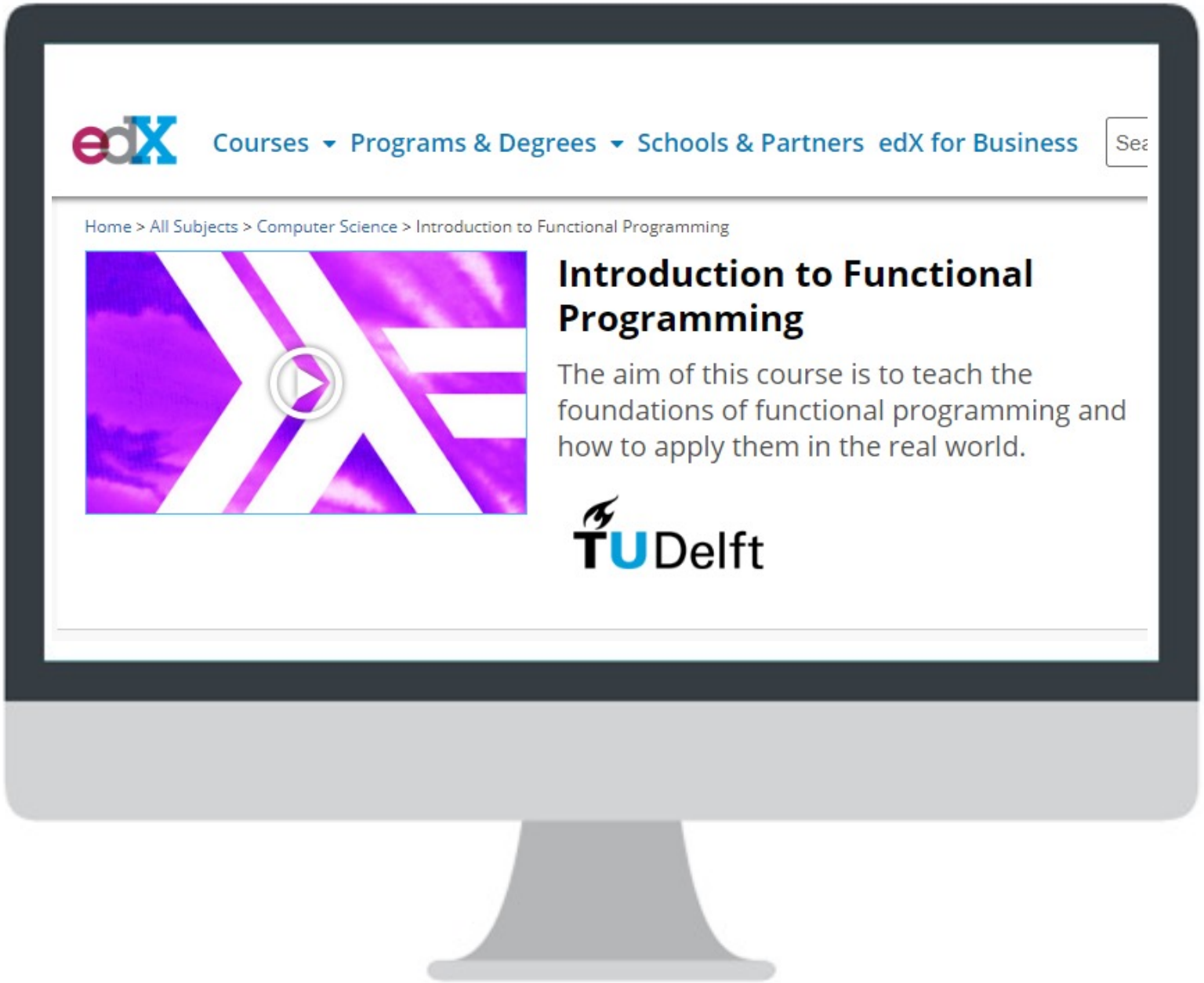
➔ xing.to/fsi

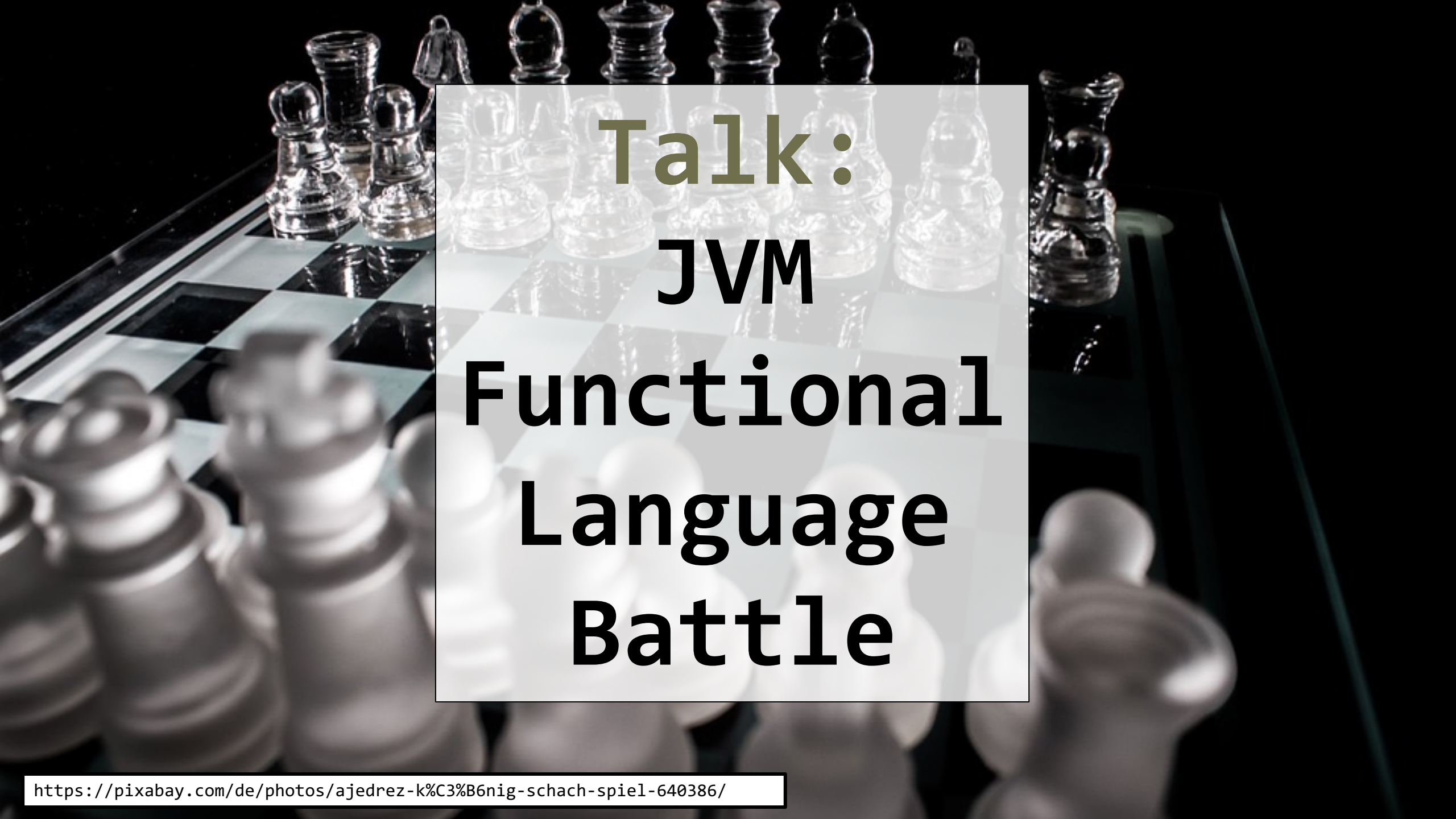


embarc
Software Consulting GmbH


Groovy


Softwerkskammer
Software Craftsmanship Communities
Überblick
Aktivitäten
Gruppen
Wiki
Lerngruppe Funktionale Programmierung
Themengruppe
Adresse: lernfp@softwerkskammer.org
Wir interessieren uns für Funktionale Programmierung und wollen uns gegenseitig soviel FP wie möglich beibringen. Wer Funktionale Programmierung lernen möchte, oder wer Funktionale Programmierung kann und sich berufen fühlt sein Wissen zu teilen sollte hier mal reinschauen :)
Mitglieder:
Diese Gruppe hat 182 Mitglieder.


edX
Courses
Programs & Degrees
Schools & Partners
edX for Business
Home > All Subjects > Computer Science > Introduction to Functional Programming
Introduction to Functional Programming
The aim of this course is to teach the foundations of functional programming and how to apply them in the real world.
TU Delft



Talk: JVM Functional Language Battle

https://pixabay.com/de/photos/ajedrez-k%C3%B6nig-schach-spiel-640386/

Agenda

1. Warum funktional programmieren?
2. Java ist doch schon funktional, oder?
3. Erweiterte funktionale Konzepte
4. Fazit und Ausblick

Agenda

1 **Warum funktional programmieren?**

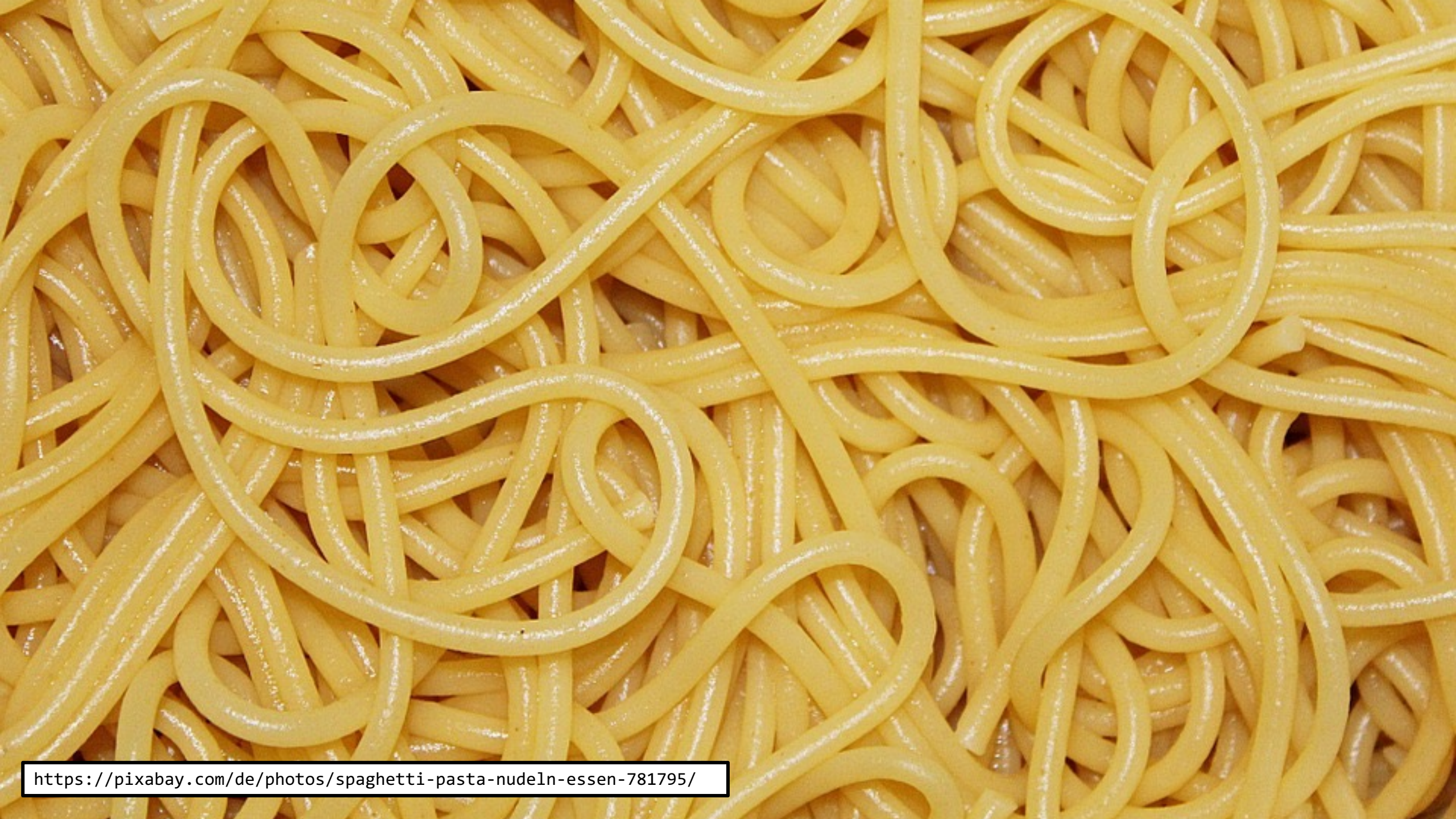
2 Java ist doch schon funktional, oder?

3 Erweiterte funktionale Konzepte

4 Fazit und Ausblick

1

```java
public class FooAlgorithm{
    public static boolean isValid(long number) {
        int sum = 0;
        boolean alternate = false;
        while(number > 0) {
            long digit = number % 10;
            if (alternate) {
                sum += 2 * digit;
                if (digit >= 5) {
                    sum -= 9;
                }
            } else {
                sum += digit;
            }
            number = number / 10;
            alternate = !alternate;
        }
        return sum % 10 == 0;
    }
}
```



https://pixabay.com/de/photos/spaghetti-pasta-nudeln-essen-781795/

```java
public class FooAlgorithm{
    public static boolean isValid(long number) {
        int sum = 0;
        boolean alternate = false;
        while(number > 0) {
            long digit = number % 10;
            if (alternate) {
                sum += 2 * digit;
                if (digit >= 5) {
                    sum -= 9;
                }
            } else {
                sum += digit;
            }
            number = number / 10;
            alternate = !alternate;
        }
        return sum % 10 == 0;
    }
}
```

Prüfsummen-Berechnung

```java
public class LuhnAlgorithm {
    public static boolean isValid(long number) {
        int sum = 0;
        boolean alternate = false;
        while(number > 0) {
            long digit = number % 10;
            if (alternate) {
                sum += 2 * digit;
                if (digit >= 5) {
                    sum -= 9;
                }
            } else {
                sum += digit;
            }
            number = number / 10;
            alternate = !alternate;
        }
        return sum % 10 == 0;
    }
}
```

Aufspalten in Ziffern

Jede zweite verdoppeln

Aufsummieren

Validierungsprüfung

	4716347184862961															
①	4	7	1	6	3	4	7	1	8	4	8	6	2	9	6	1
②	1	6	9	2	6	8	4	8	1	7	4	3	6	1	7	4
③	1	12	9	4	6	16	4	16	1	14	4	6	6	2	7	8
④	1	1 2	9	4	6	1 6	4	1 6	1	1 4	4	6	6	2	7	8
⑤	1	3	9	4	6	7	4	7	1	5	4	6	6	2	7	8
⑥	1 + 3 + 9 + 4 + 6 + 7 + 4 + 7 + 1 + 5 + 4 + 6 + 6 + 2 + 7 + 8															
	80															
⑦	80 % 10 == 0															
	gültig															

```java
public class LuhnAlgorithm {
    public static boolean isValid(long number) {
        int sum = 0;
        boolean alternate = false;
        while(number > 0) {
            long digit = number % 10;
            if (alternate) {
                sum += 2 * digit;
                if (digit >= 5) {
                    sum -= 9;
                }
            } else {
                sum += digit;
            }
            number = number / 10;
            alternate = !alternate;
        }
        return sum % 10 == 0;
    }
}
```

- Variablen
- Schleifen
- Verzweigungen
- Tiefe der Verschachtelung
- Zustands-änderungen
- Reihenfolge nicht beliebig
- Fehler-behandlung?

https://pixabay.com/de/photos/eisberg-antarktis-polaren-blau-eis-404966/

Agenda

1. Warum funktional programmieren?
2. **Java ist doch schon funktional, oder?**
3. Erweiterte funktionale Konzepte
4. Fazit und Ausblick

2

Funktionsliterale und Higher-Order-Functions

1

2

```
1  package de.oio.luhn;
2
3  public class LuhnAlgorithmJava8 {
4      public static boolean isValid(String creditCardNumber) {
5          int[] i = { creditCardNumber.length() % 2 == 0 ? 1 : 2 };
6
7          return creditCardNumber
8                  .chars()
9                  .map(in -> in - '0')
10                 .map(n -> n * (i[0] = i[0] == 1 ? 2 : 1))
11                 .map(n -> n > 9 ? n - 9 : n)
12                 .sum() % 10 == 0;
13     }
14 }
```

Verkappte Zustandsänderung

Unendliche Streams

3

```java
package de.oio.luhn.thomas_much;

import java.util.PrimitiveIterator;
import java.util.stream.IntStream;

public class Luhn {
    public static boolean isValid(String number) {
        PrimitiveIterator.OfInt faktor =
                IntStream.iterate(1, i -> 3 - i).iterator();
        return (new StringBuilder(number)
                .reverse()
                .chars()
                .map(c -> faktor.nextInt() * (c - '0'))
                .reduce(0, (a, b) -> a + b / 10 + b % 10) % 10) == 0;
    }
}
```

OHNE Zustandsänderung

Generator: 1 2 1 2 …

nach Idee von Thomas Much

Funktionskomposition

Verketten/Komposition von Teil-Funktionen: (f . g)(x) == f(g(x))

4

```
Function<Integer, Integer> times2 = e -> e * 2;
Function<Integer, Integer> squared = e -> e * e;

System.out.println(times2.compose(squared).apply(4)); // 32
System.out.println(times2.andThen(squared).apply(4)); // 64
```

Currying und partielle Funktionsaufrufe

Currying: Konvertierung einer Funktion mit n Argumenten
in n Funktionen mit jeweils einem Argument.

5

```
IntBinaryOperator simpleAdd = (a, b) -> a + b;
IntFunction<IntUnaryOperator> curriedAdd = a -> b -> a + b;

System.out.println(simpleAdd.applyAsInt(4, 5));

System.out.println(curriedAdd.apply(4).applyAsInt(5));
```

Currying und partielle Funktionsaufrufe

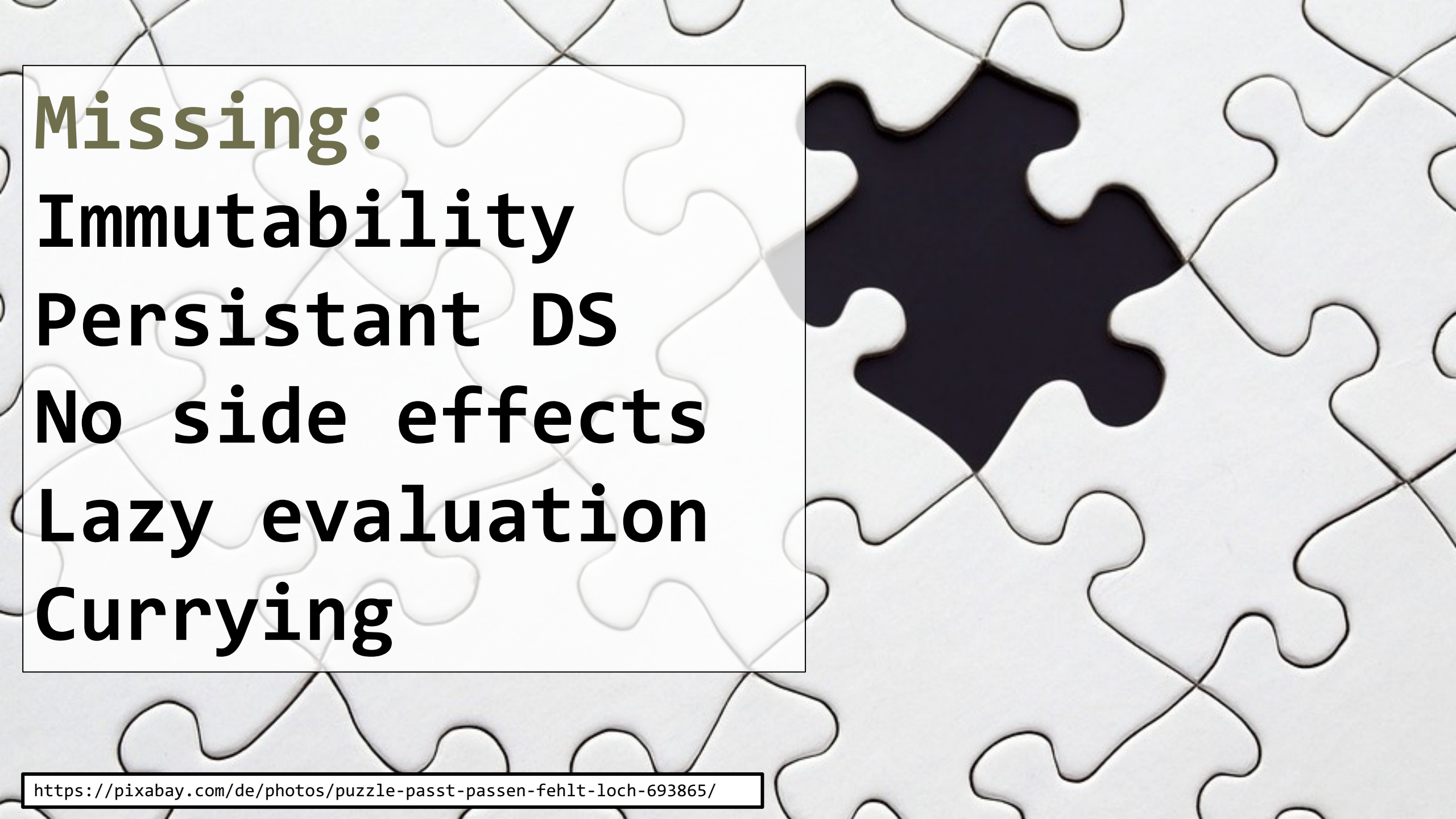
Partielle Aufrufe: Spezialisierung von allgemeinen Funktionen

5

```
IntUnaryOperator adder5 = curriedAdd.apply(5);
System.out.println(adder5.applyAsInt(4));
System.out.println(adder5.applyAsInt(6));
```

Missing:

Immutability
Persistant DS
No side effects
Lazy evaluation
Currying



https://pixabay.com/de/photos/puzzle-passt-passen-fehlt-loch-693865/

RxJava

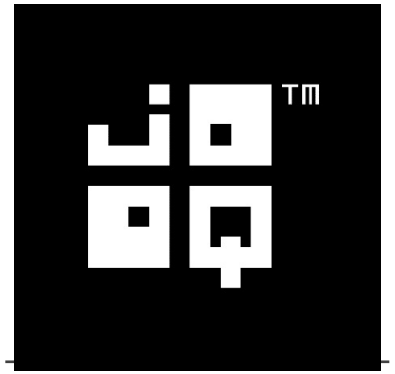
PROJECT REACTOR

VERT.X

Immutables stars 1,662

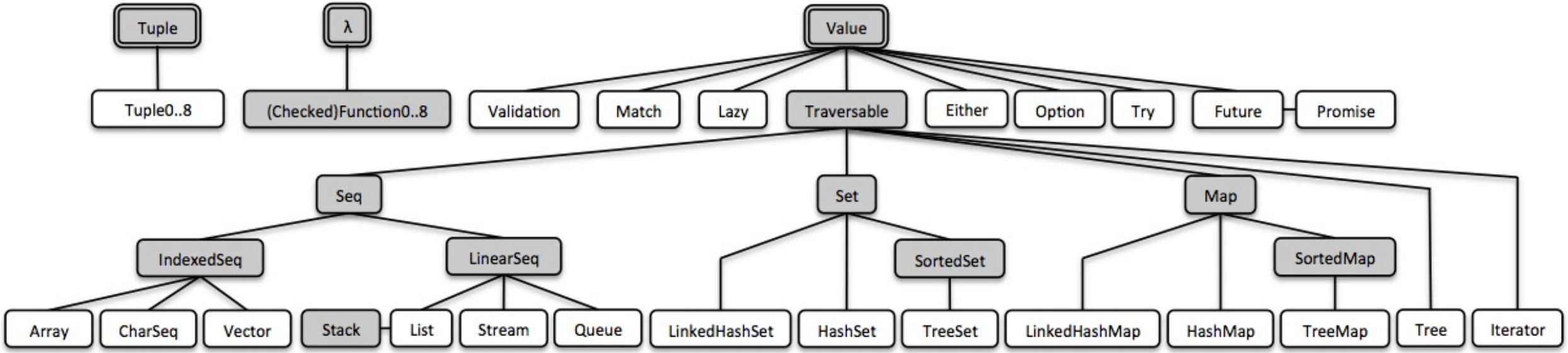
VAVR.IO

Project Lombok



JDeferred

Java Deferred / Promise library



http://www.vavr.io/vavr-docs/

Agenda

1. Warum funktional programmieren?
2. Java ist doch schon funktional, oder?
3. **Erweiterte funktionale Konzepte**
4. Fazit und Ausblick

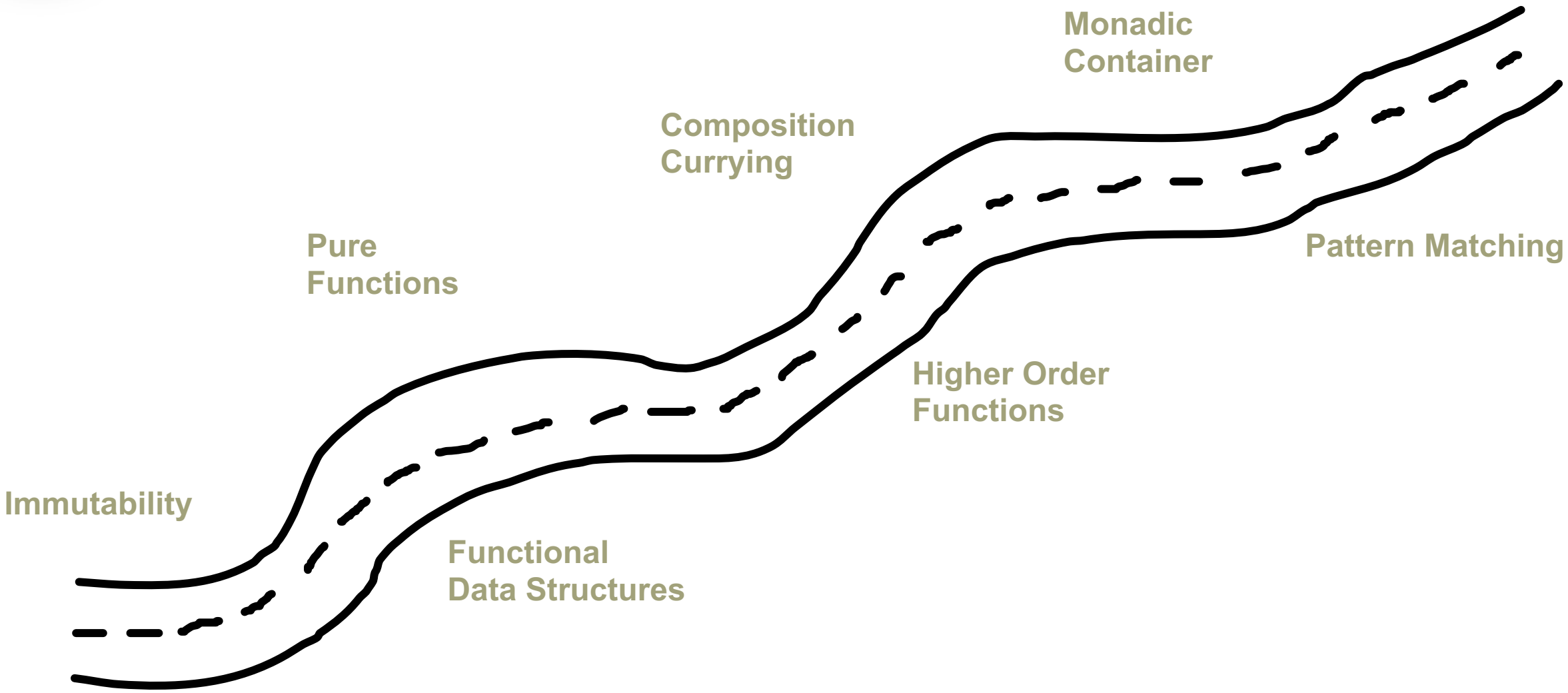
3

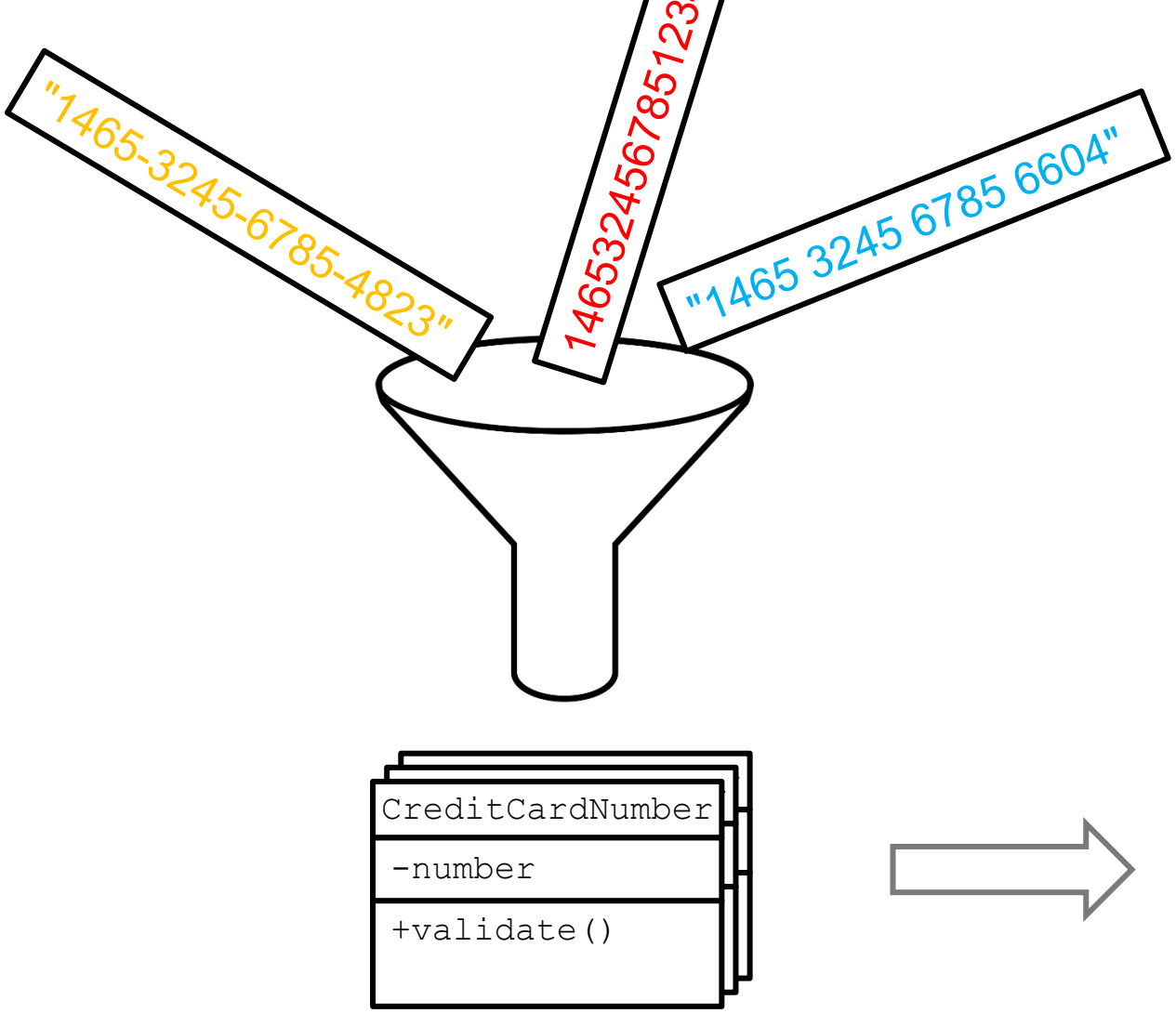
Dr. Gernot Starke
@gernotstarke

To understand recursion, you need to understand recursion first.

Tweet übersetzen
12:30 nachm. · 18. Dez. 2019 · Twitterrific for Mac

Functional Concepts



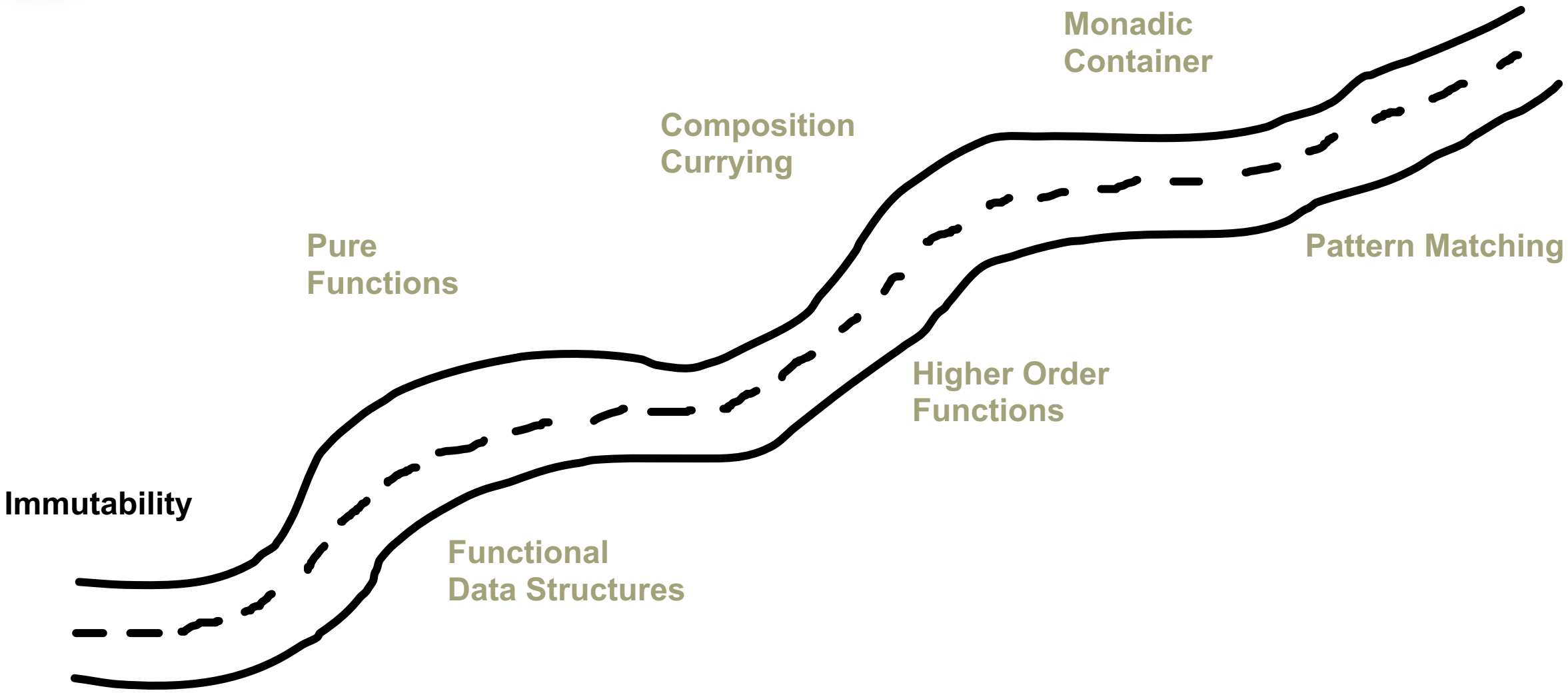


Nummer	gültig
1465324567854823	☑
1465324567851234	☐
1465324567856604	☑

Funktionale Konzepte

1. Immutability/Value Types
2. Seiteneffektfreiheit/Referentielle Transparenz/Pure Functions
3. Immutable/Persistent/Functional Data Structures
4. Funktionen/Komposition/Currying/Partielle Funktionsaufrufe
5. Funktionsliterale/Higher Order Functions
6. Values/Monadic Container
7. Pattern Matching

Functional Concepts



Bauplan immutable Klasse

Keine Mutatoren zur Verfügung stellen

- zustandsverändernde Methoden, z.B. „setter“

Überschreiben von Methoden verhindern

- Klasse final setzen

Alle Felder final setzen

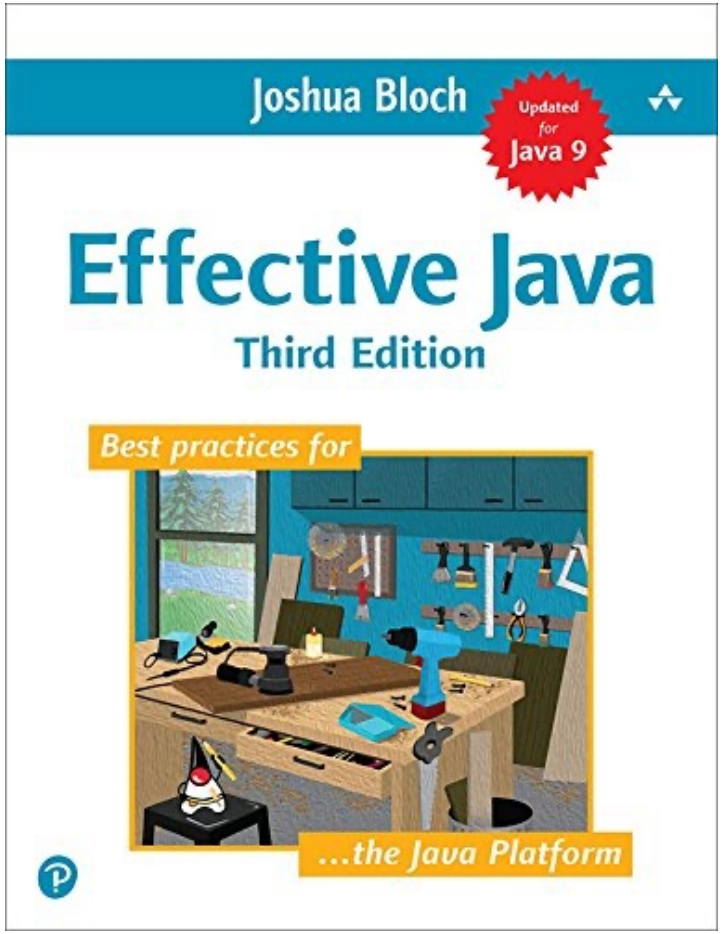
- Ausnutzen der System Restriktionen

Alle Felder private setzen

- Verhindert direktes Ändern durch Clients

Exklusiven Zugang zu mutable Feldern gewährleisten

- Defensive Copies in Konstruktoren, Accessoren und readObject()



Konsequenzen Immutability

Vorteile	Nachteile
genau ein Zustand	Jeder einzelne Zustand benötigt ein eigenes Objekt
Thread-safe	Alternative für vorhersagbare Operationen
Instanzen können gemeinsam genutzt werden	Ansonsten öffentliche mutable „Companion Class“ ▪ vgl. String und StringBuffer
Auch innere Zustände können gemeinsam genutzt werden	

Lombok

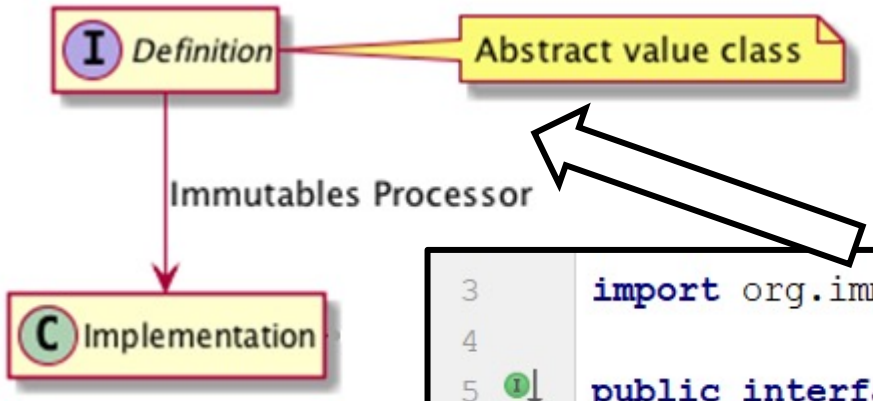
```
1  package de.oio.luhn.values;
2
3  import lombok.Value;
4
5  @Value
6  public class CreditCardNumber {
7      private Long number;
8
9      public static void main(String[] args) {
10         new CreditCardNumber(123456789L).
11     }
12 }
13
```

```
getNumber()          Long
number               Long
equals(Object o)     boolean
hashCode()           int
toString()           String
clone()              Object
```

private, final,
keine Setter,
Argument-Konstruktor
equals/hashcode/toString

optional:
Builder
Lazy-Evaluierung
….

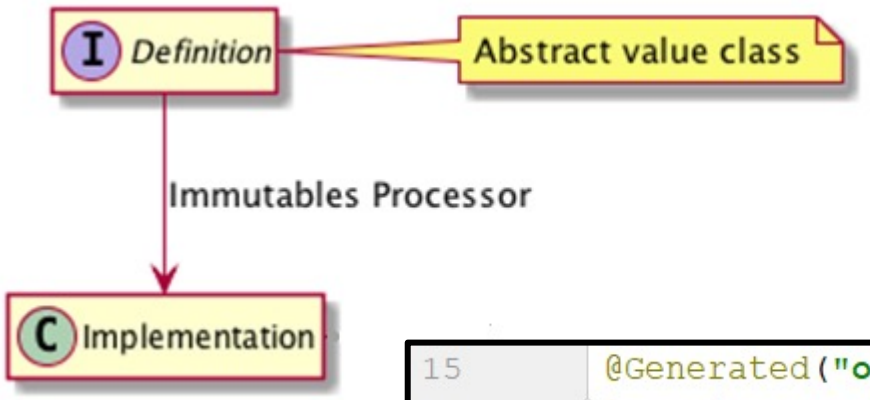
Immutables - Definition



optional:
Customizing (Styles)
Preconditions/Normalisierung
Abgeleitete Properties
Lazy Attribute
Auxiliary/Redacted Attribute

```
3   import org.immutables.value.Value;
4
5   public interface CreditCardNumber {
6
7       @Value.Check
8       default void check(){
9           if (getNumber() < 1) {
10              throw new IllegalStateException("Credit card number must be 1 or greater");
11          }
12      }
13
14      Long getNumber();
15  }
```

Immutables - Implementierung



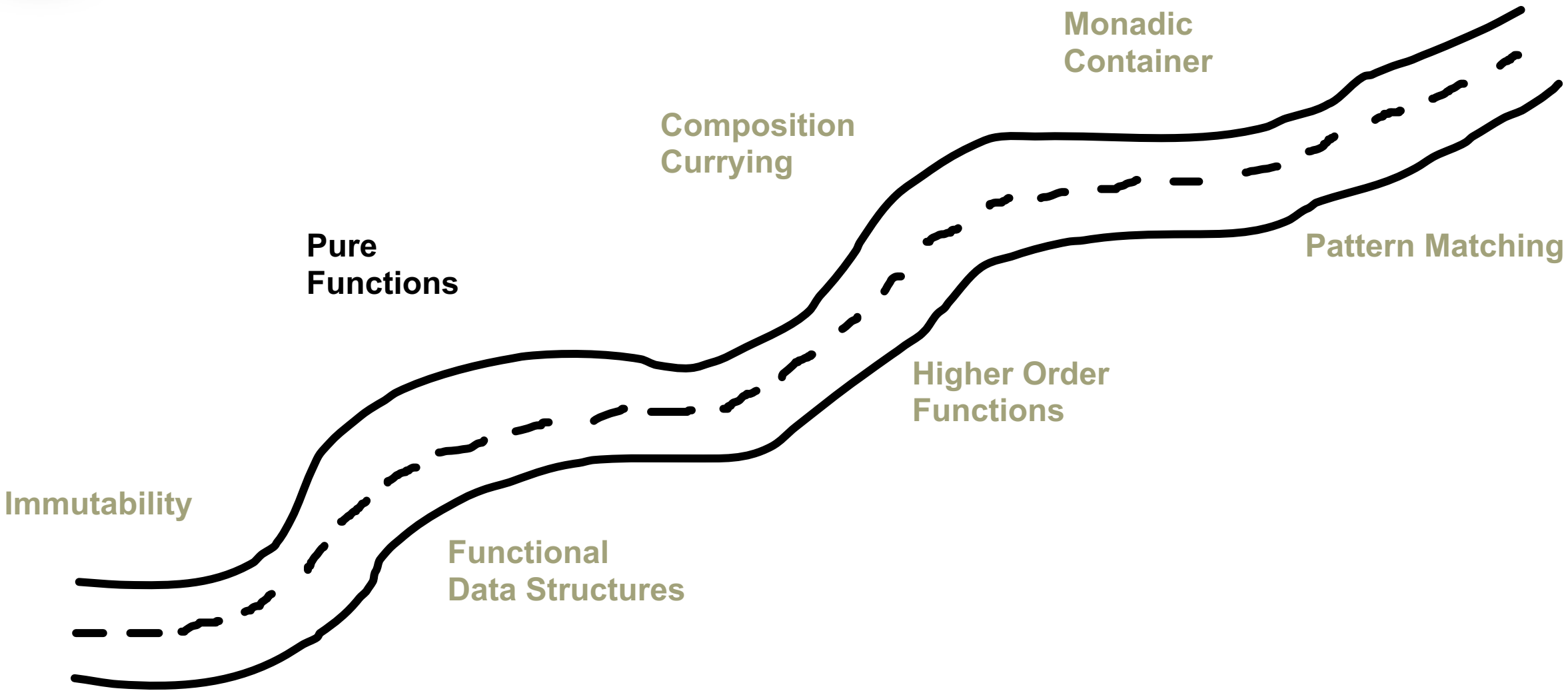
private final Attribute
equals/hashCode
toString
Builder/Factory Method
private Constructor

```java
@Generated("org.immutables.processor.ProxyProcessor")
@org.immutables.value.Generated(from = "CreditCardNumber", generator = "Immutables")
public final class ImmutableCreditCardNumber implements CreditCardNumber {
  private final Long number;

  private ImmutableCreditCardNumber(Long number) { this.number = number; }

  /**
   * @return The value of the {@code number} attribute
   */
  @Override
  public Long getNumber() { return number; }
```

Immutables - Verwendung

```
6  public class Main {
7      public static void main(String[] args) {
8          System.out.println(ImmutableCreditCardNumber.builder().number(1000L).build());
9          ImmutableCreditCardNumber.builder().number(-1L).build();
10     }
11 }
```

Functional Concepts



“Construct [our] programs using pure functions only”

“Pure functions have no side effects”

“A function has a side effect if it does something other than simply return a result”

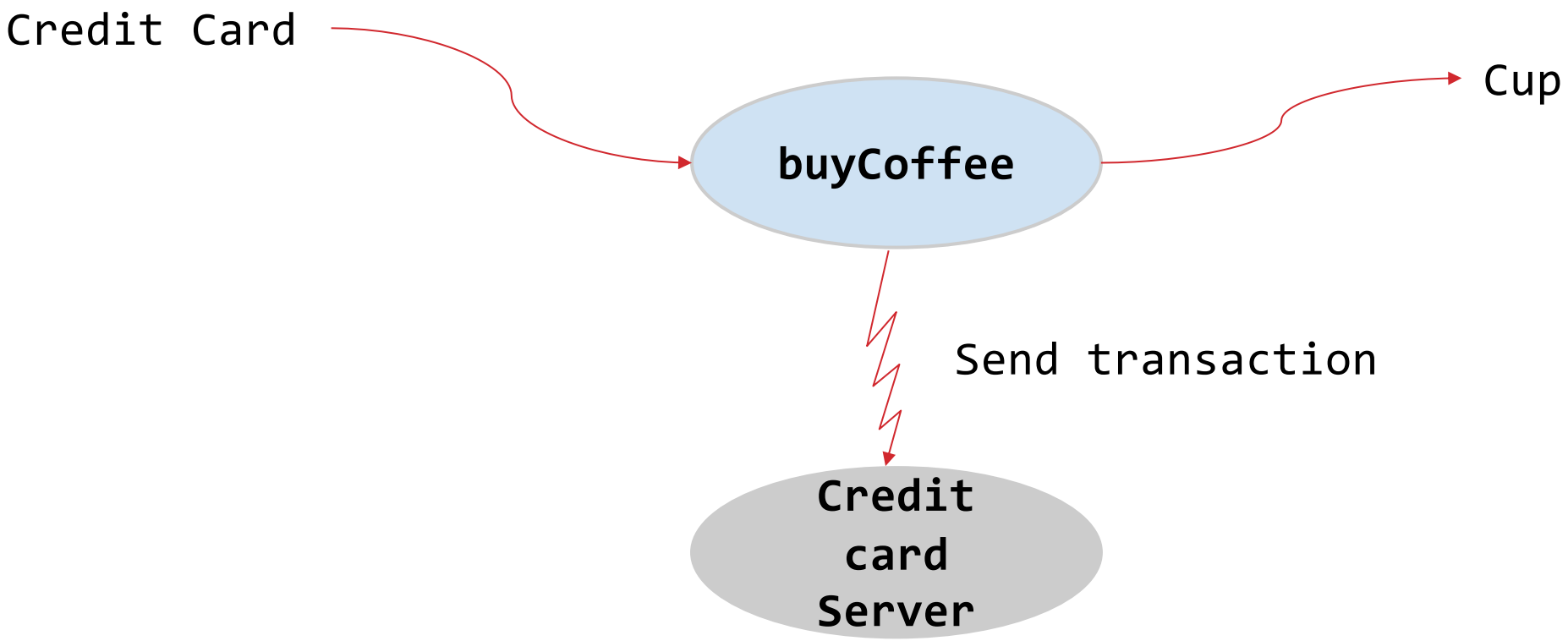
Side effects

Modifying a variable
Throwing an exception
Printing to the console
Writing to a file

Side effects

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

Side effects



But …

… a purely functional programmed application is useless!

Functional Core, Procedural Shell

“A function is pure if calling it with referential transparent arguments is also referential transparent”

“An expression is **referential transparent**, if it can be replaced by its result without changing the meaning of the program”

```
Math.random();
```

```
Math.max(1, 2);
```

Benefits

“Es funktioniert, wenn es kompiliert …”

Inhalt leicht zu schlussfolgern

sehr gut testbar

kein Debugging notwendig

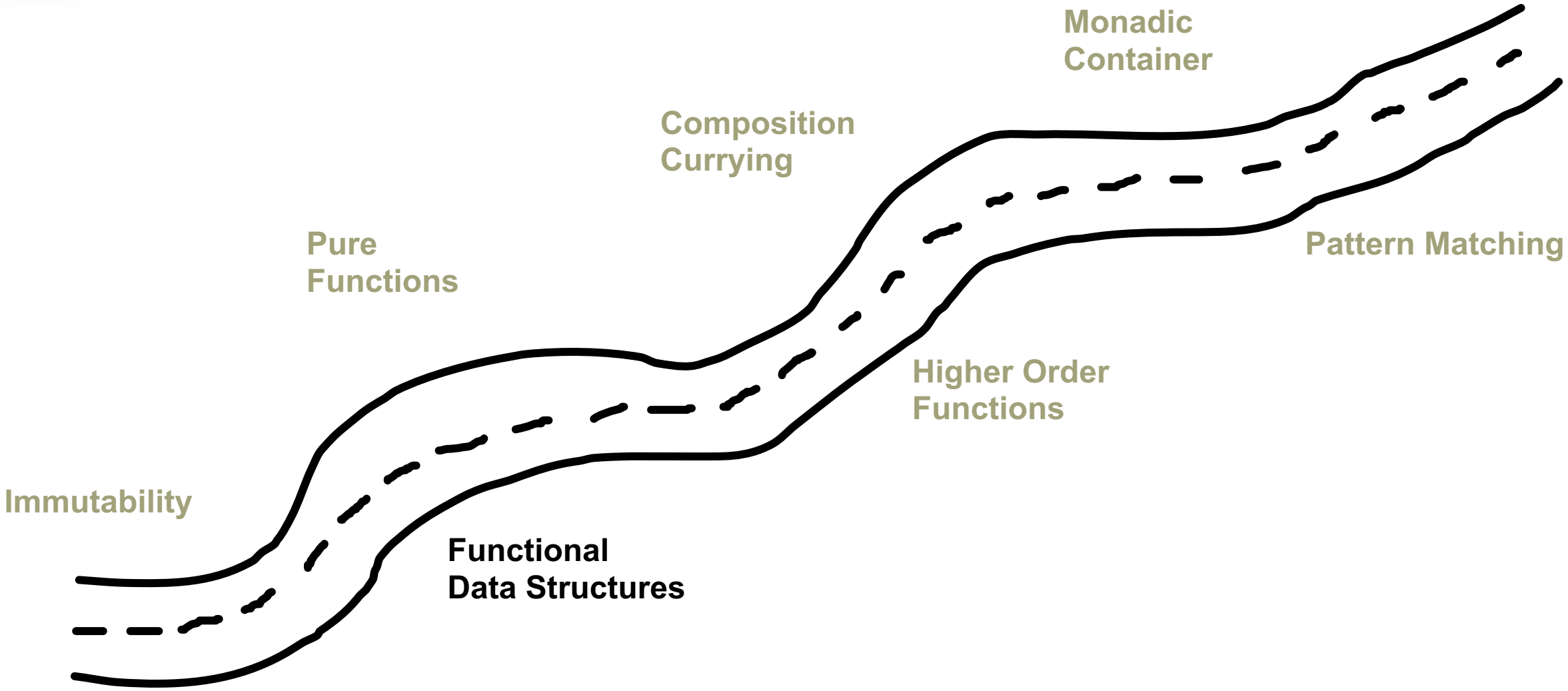
Schlüssel für besseres Java

Immutable Datentypen

+

referentiell transparente Funktionen!

Functional Concepts



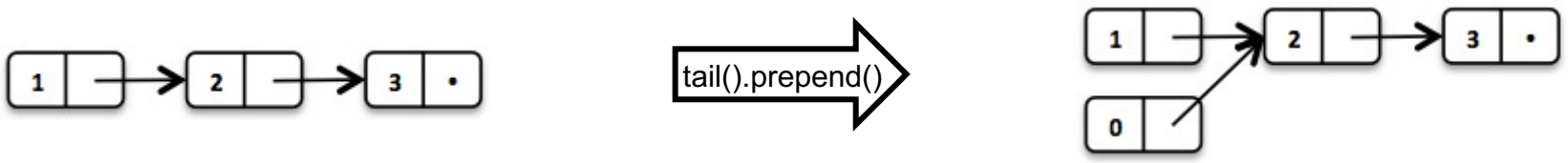
Java Collections sind änderbar!

```
interface Collection<E> {
    void clear();
}
```

```
List<String> list = Collections
                    .unmodifiableList(otherList);
list.add("Boom");
```

Persistente/Funktionale Datenstrukturen in Vavr

```
List<Integer> list1 = List.of(1, 2, 3);
List<Integer> list2 = list1.tail().prepend(0);
```



http://www.vavr.io/vavr-docs/#_functional_data_structures

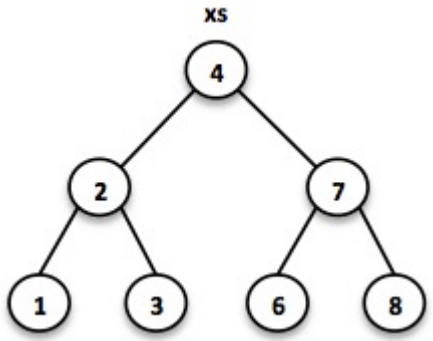
Persistente/Funktionale Datenstrukturen in Vavr

```
Queue<Integer> queue = Queue.of(1, 2, 3)
        .enqueue(4).enqueue(5);
Queue<Integer> queue2 = queue
        .dequeue().dequeue().dequeue()
```

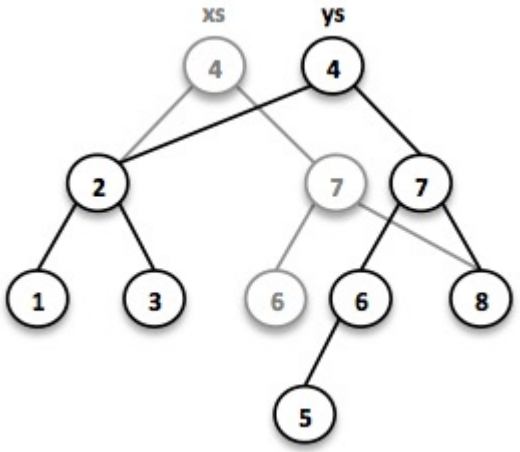
http://www.vavr.io/vavr-docs/#_functional_data_structures

Persistente/Funktionale Datenstrukturen in Vavr

```
// = TreeSet(1, 2, 3, 4, 6, 7, 8)
SortedSet<Integer> xs =
    TreeSet.of(6, 1, 3, 2, 4, 7, 8);
```




```
// = TreeSet(1, 2, 3, 4, 5, 6, 7, 8)
SortedSet<Integer> ys = xs.add(5);
```



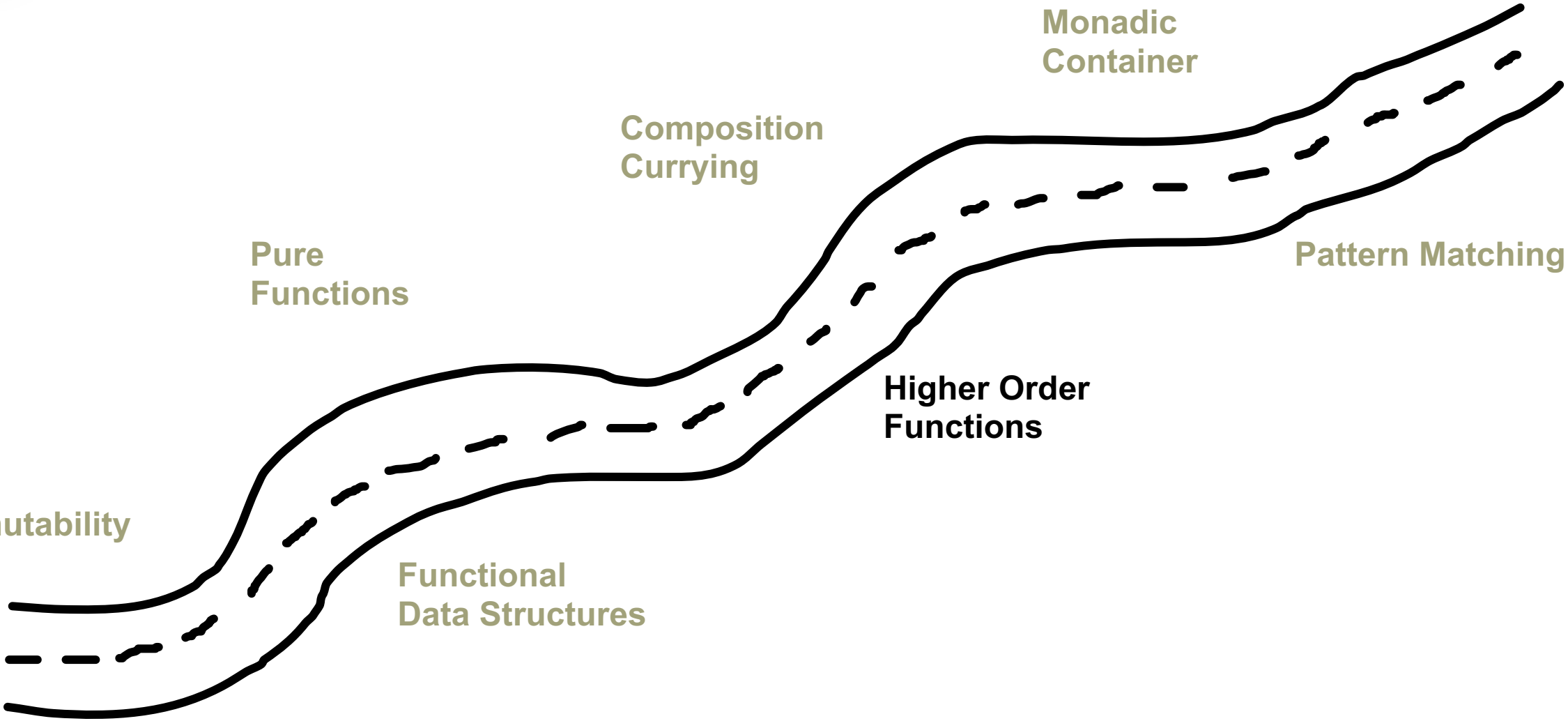
http://www.vavr.io/vavr-docs/#_functional_data_structures

```
List<User> result = users.stream()
  .filter(user -> {
    try {
       return user.validate();
    } catch (Exception ex) {
       return false;
    }
  })
  .map(user -> user.name)
  .collect(Collectors.toList());
```


```
List<User> result = List.ofAll(users)
    .filter(user ->
       Try.of(user::validateAddress)
          .getOrElse(false)
    )
    .map(user -> user.name);

java.util.List<User> result2 =
   result.toJavaList();
```

java.util.List

Functional Concepts



Functions are values.

A function can be computed, passed around and returned.

“A higher order function is a function that takes a function as an argument and/or returns a function.”

Higher Order Functions

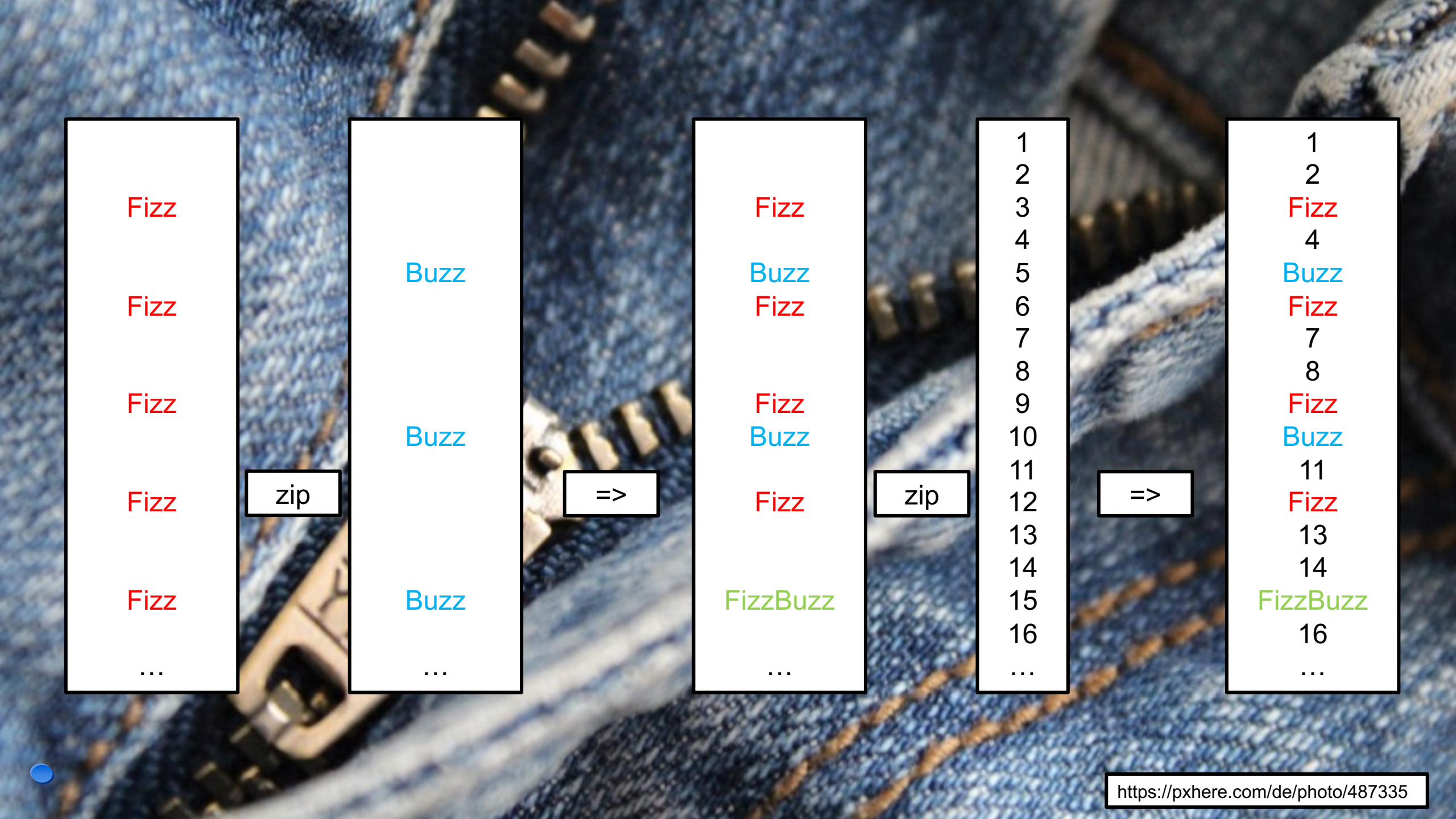
```
creditCardNumber
    .chars() // convert to int
    .map(in -> in - '0') // multiply by 1, 2 alternating
    .map(n -> n * (i[0] = i[0] == 1 ? 2 : 1)) // sum of digits
    .map(n -> n > 9 ? n - 9 : n)
    .sum() % 10 == 0;
```

1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
...

```
final Stream<String> fizzes = Stream.of("", "", "Fizz").cycle();
final Stream<String> buzzes = Stream.of("", "", "", "", "Buzz").cycle();
final Stream<String> fizzBuzzes = fizzes.zipWith(buzzes, (t1, t2) -> t1 + t2);
final Stream<String> result = fizzBuzzes
        .zipWith(Stream.from(1), (_1, _2) -> _1.isEmpty() ? _2.toString() : _1);

result.take(20).forEach(System.out::println);
```

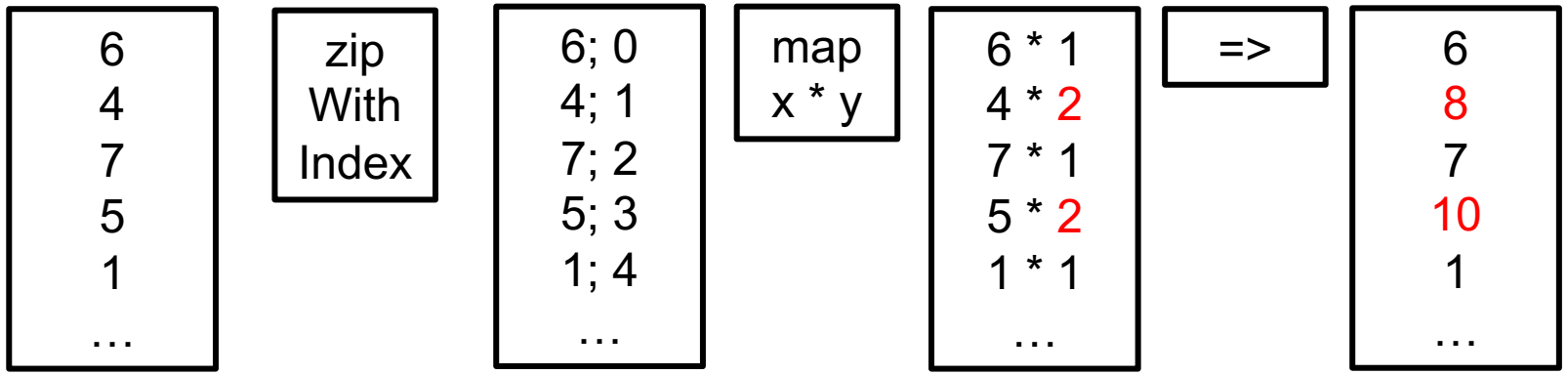
Beispiel von: https://www.sitepoint.com/functional-fizzbuzz-with-vavr

https://pxhere.com/de/photo/487335

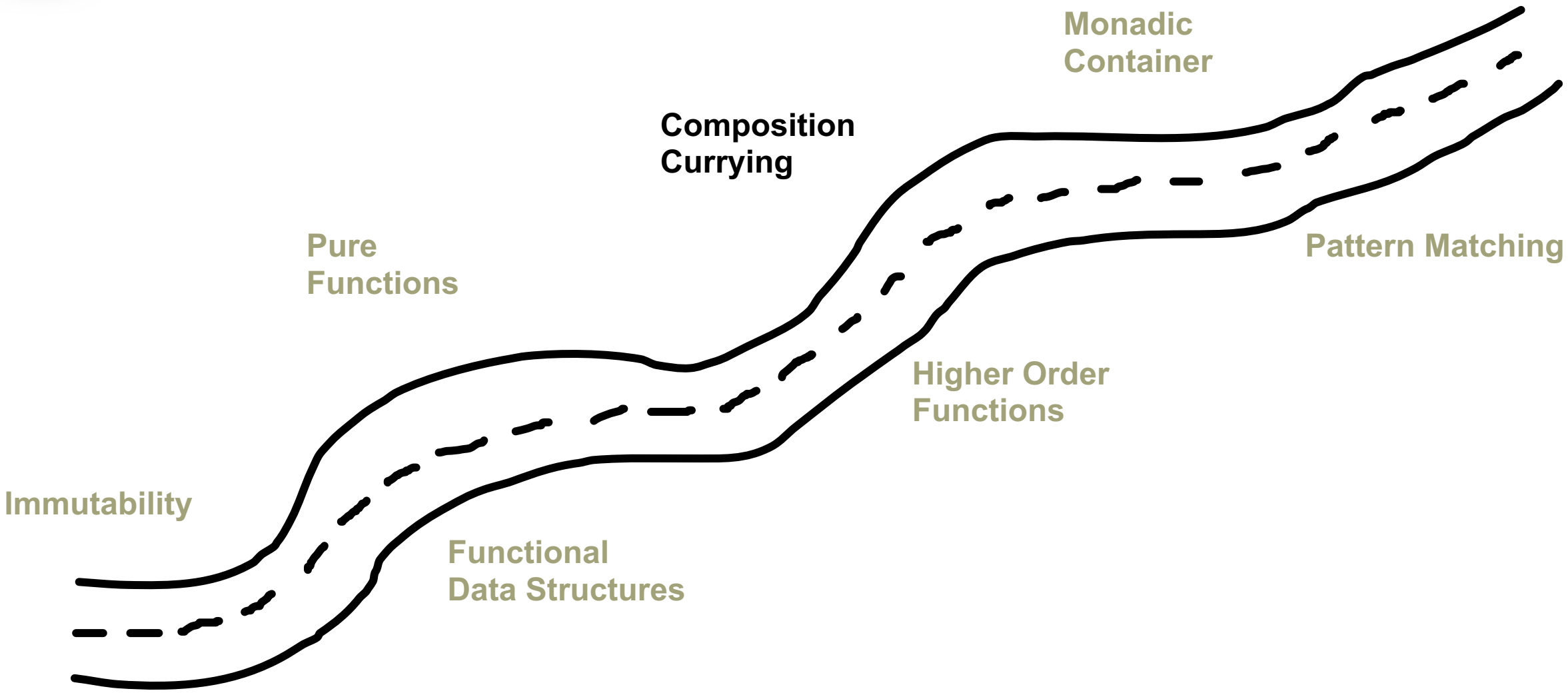

Fizz
Fizz
Fizz
Fizz
Fizz
...
zip
Buzz
Buzz
Buzz
...
=>
Fizz
Buzz
Fizz
Fizz
Buzz
Fizz
FizzBuzz
...
zip
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
...
=>
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
...
https://pxhere.com/de/photo/487335

Luhn: Zippen zum Verdoppeln jeder zweiten Ziffer

```
static Function1<Seq<Integer>, Seq<Integer>> double2nd =
    digits -> digits.zipWithIndex()
                    .map(t -> t._1 * (t._2 % 2 + 1));
```



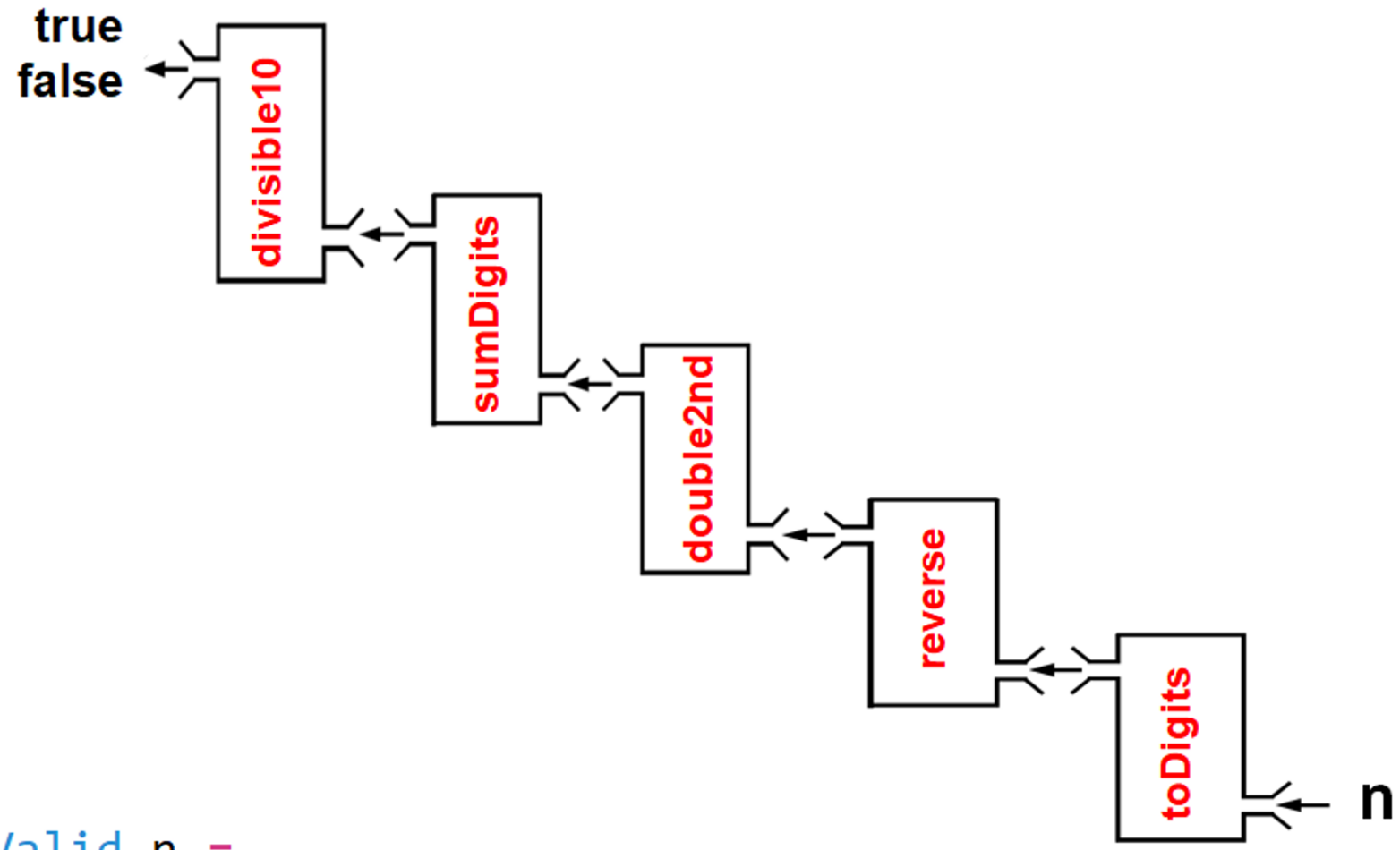
Functional Concepts



"... function composition is an act or mechanism to combine simple functions to build more complicated ones."

```
static Function1<Long, Boolean> isValid =
        toDigits.andThen(reverse)
                .andThen(double2nd)
                .andThen(sumDigits)
                .andThen(divisibleBy10);
```

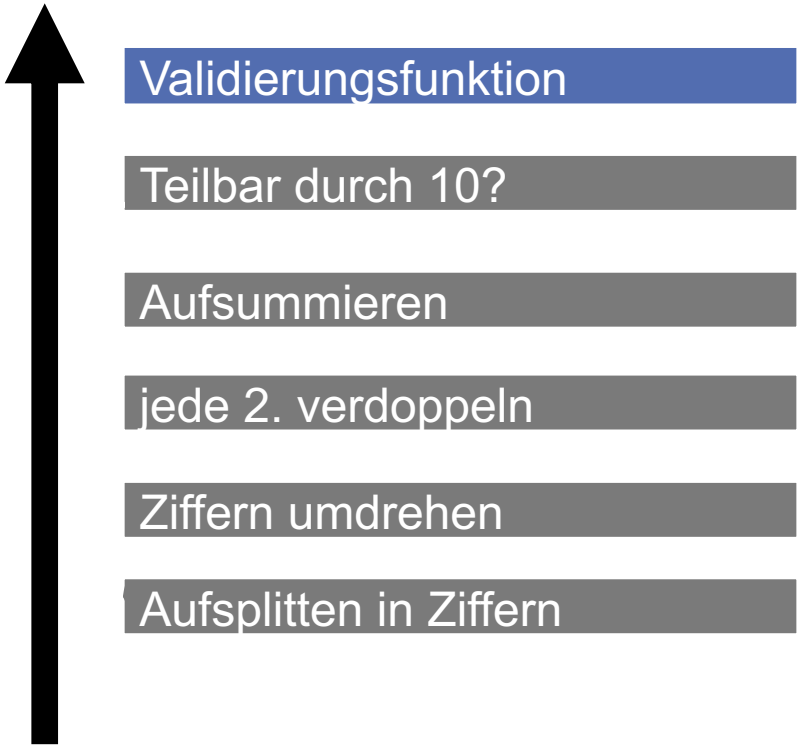
function composition




```
isValid n =
divisibleBy10(sumDigits(double2nd(reverse(toDigits(n)))))
```

Luhn Algorithmus in funktional

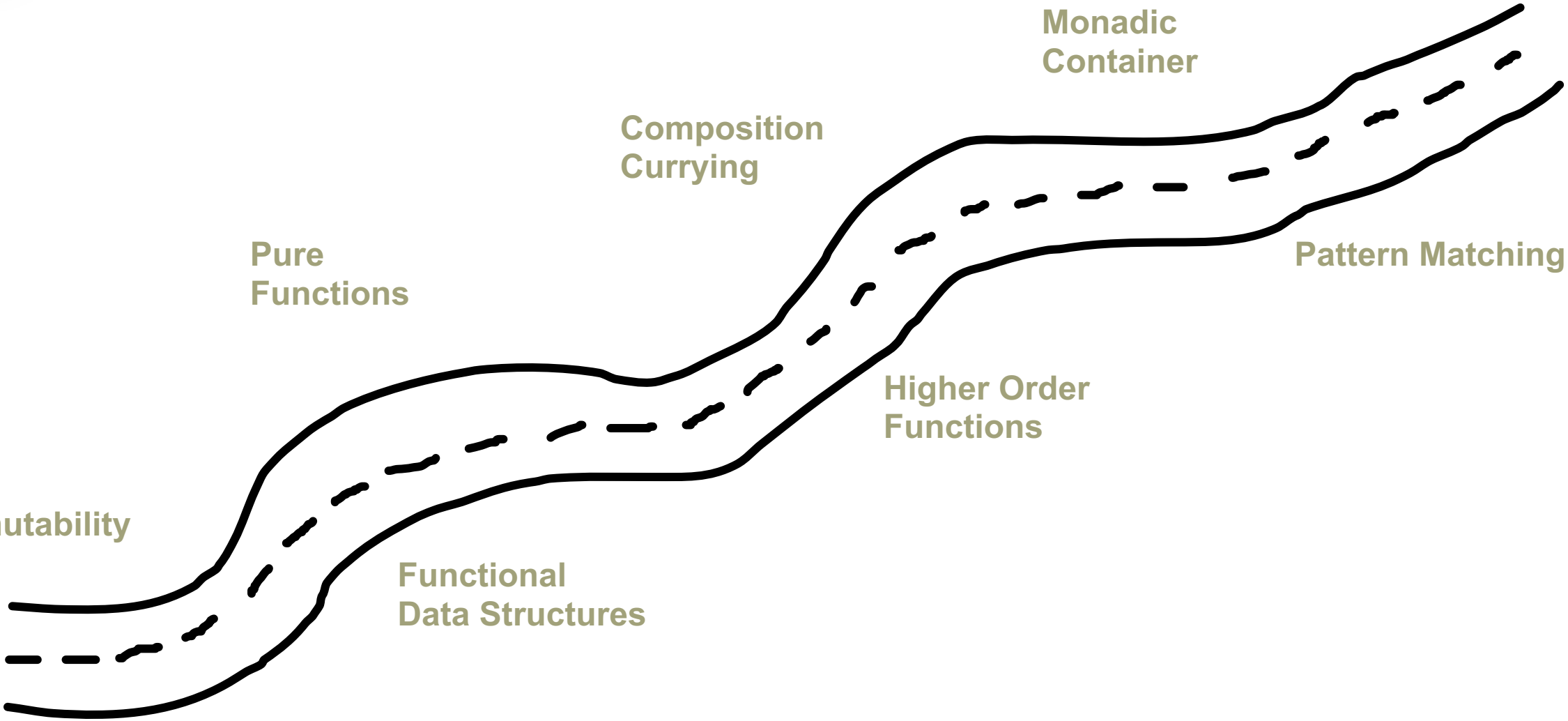
```
isValid n =
    divisibleBy10(
      sumDigits(
        double2nd(
          reverse(
            toDigits(n)
          )
        )
      )
    )
```



Luhn: Teilschritte als einzelne Funktionen

```
static Function1<Long, Seq<Integer>> toDigits = number ->
    CharSeq.of(Long.toString(number)).map(c -> c - '0');

static Function1<Seq<Integer>, Seq<Integer>> reverse = Seq::reverse;

static Function1<Seq<Integer>, Seq<Integer>> double2nd =
    digits -> digits.zipWithIndex().map(t -> t._1 * (t._2 % 2 + 1));

static Function1<Seq<Integer>, Integer> sumDigits = digits ->
    digits.map(i -> i.longValue()).flatMap(toDigits).sum().intValue();

static Function1<Integer, Boolean> divisibleBy10 = number ->
    number % 10 == 0;
```

Functional Concepts



"... currying is the technique of translating the evaluation of a function that takes multiple arguments into evaluating a sequence of functions, each with a single argument."

```
List<Integer> myZip(function, list1, list2) {
   ...
}

List<Integer> result = myZip((a, b) -> a * b), List(1, 2, 3), List(4, 5, 6));

var curriedMyZip = myZip.curried();

result = curriedMyZip.apply((a, b) -> a * b)
                                    .apply(List(1, 2, 3))
                                    .apply(List(4, 5, 6))
```

currying

```
Function3<BiFunction<Integer, Integer, Integer>,
        List<Integer>,
        List<Integer>,
        List<Integer>> myZip = ... // = myZip(function, list1, list2)

Function1<BiFunction<Integer, Integer, Integer>,
  Function1<List<Integer>,
    Function1<List<Integer>, List<Integer>>>> curriedMyZip
              = myZip.curried();

List<Integer> result = curriedMyZip.apply((a, b) -> a * b)
                                   .apply(List(1, 2, 3))
                                   .apply(List(4, 5, 6))
```

currying

“… partial function application refers to the process of fixing a number of arguments to a function, producing another function of smaller arity.”

```
Function3<BiFunction<Integer, Integer, Integer>,
        List<Integer>,
        List<Integer>,
        List<Integer>> myZip = ... // = myZip(function, list1, list2)

Function1<List<Integer>, List<Integer>> zipped =
    myZip.curried()
            .apply((a, b) -> a * b)
            .apply(List(1, 2, 3));

zipped.apply(List(4, 5, 6))
```

partial function application

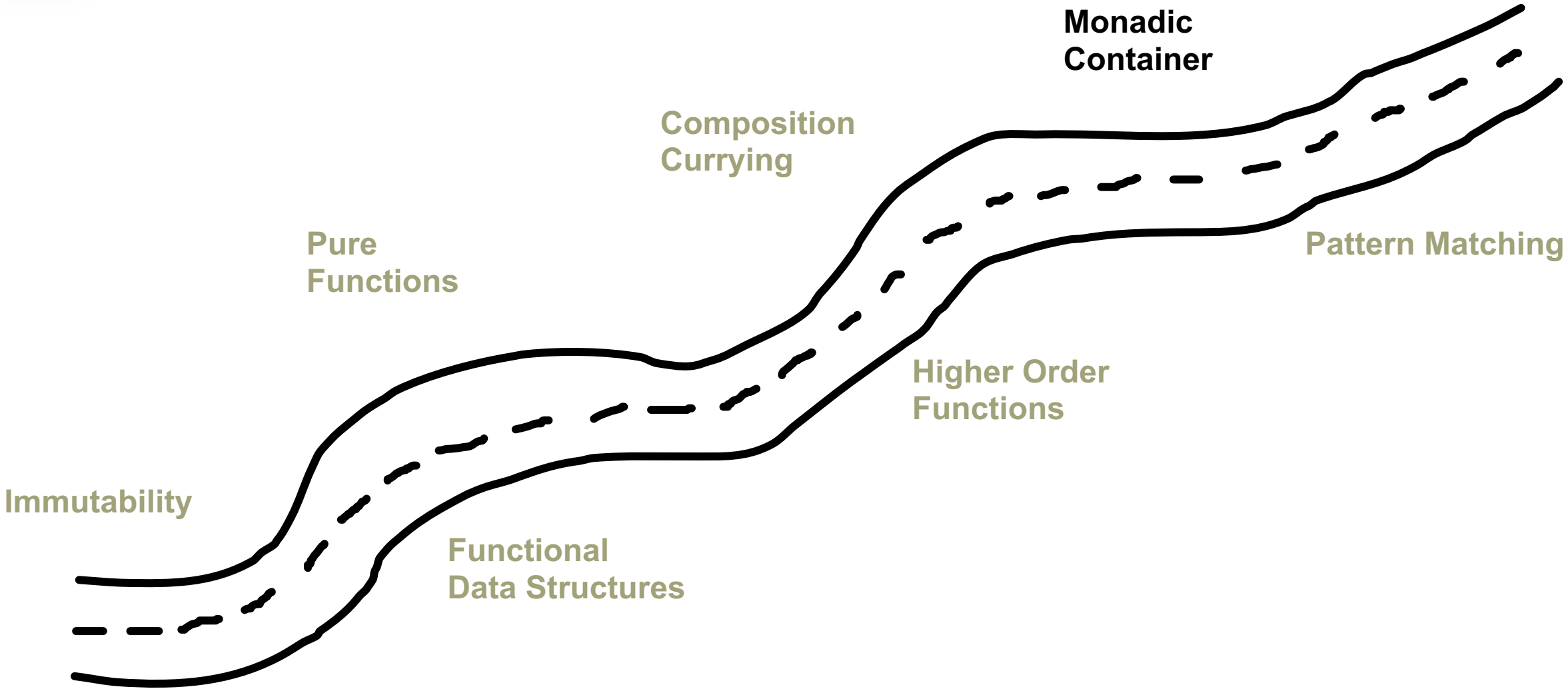
```
Function0<Double> cachedRandom =
    Function0.of(Math::random).memoized();

double randomValue1 = cachedRandom.apply();
double randomValue2 = cachedRandom.apply();

then(randomValue1).isEqualTo(randomValue2);
```

memoization

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

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

Option<Integer> result1 = safeDivide.apply(4, 0);
then(result11).isEqualTo(Option.none());

Option<Integer> result2 = safeDivide.apply(4, 2);
then(result2).isEqualTo(Option.some(2));
```

lifting

Functional Concepts



```
Tuple2<String, Integer> java8 = Tuple.of("Java", 8);

Tuple2<String, Integer> vavr1 = java8.map(
        s -> s.substring(2) + "vr",
        i -> i / 8);

String vavr = vavr1._1;
int one = vavr1._2;
```

Algebraische Datentypen: Produkttypen

```
Try (Sucess, Failure)
Either (Left, Right)
Option (Some, None)
Validation (Valid, NotValid)
```

Algebraische Datentypen: Summen- oder Variantentypen

```java
static CreditCardNumber from(String s) {
    return new CreditCardNumber(Long.parseLong(s));
}
```

```java
static Try<CreditCardNumber> fromWithTry(String s) {
    return Try.of(() -> Long.parseLong(s))
            .map(n -> new CreditCardNumber(n));
}
```

```java
System.out.println(CreditCardNumber.fromWithTry( s: "abc").getOrNull());
System.out.println(CreditCardNumber.fromWithTry( s: "abc").getCause().getClass().getName());
System.out.println(CreditCardNumber.fromWithTry( s: "123").get());
```

Try

```java
static Either<String, CreditCardNumber> fromWithEither(String s) {
    try {
        return Either.right(new CreditCardNumber(Long.parseLong(s)));
    } catch (NumberFormatException e) {
        return Either.left(String.format("wrong credit card number format: %s", s));
    }
}
```

```java
System.out.println(CreditCardNumber.fromWithEither( s: "abc").isLeft());
System.out.println(CreditCardNumber.fromWithEither( s: "123").isRight());
System.out.println(CreditCardNumber.fromWithEither( s: "123").left().getOrElse( other: "no error"));
System.out.println(CreditCardNumber.fromWithEither( s: "abc").left().get());
```

Either

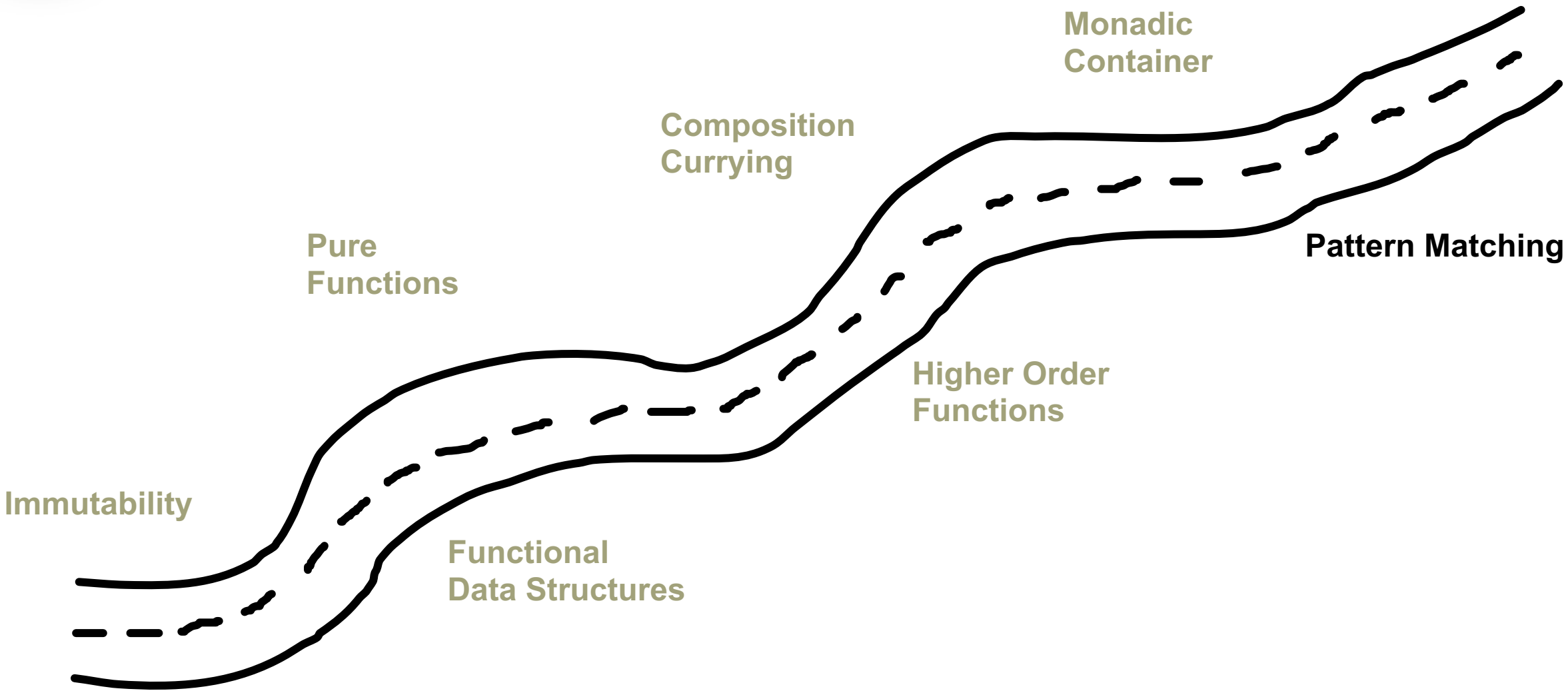
```
String helloWorld = Option.of("Hello")
        .map(value -> value + " Falk")
        .peek(value -> LOG.debug("Value: {}", value))
        .getOrElse(() -> "Hello World");
```

Option

```java
public class CreditCardValidator {
    public Validation<Seq<String>, CreditCard> validateNumber(String owner, String number) {
        final CreditCardNumber ccn = CreditCardNumber.from(number);
        return Validation.combine(
                owner == null || owner.isEmpty() ? Validation.invalid("Owner must not be empty!") : Validation.valid(owner),
                ccn.isValid() ? Validation.valid(ccn) : Validation.invalid("Credit card number is invalid: " + number))
                .ap(CreditCard::new);
    }

    public static void main(String[] args) {
        final CreditCardValidator ccv = new CreditCardValidator();
        final Validation<Seq<String>, CreditCard> validation = ccv.validateNumber( owner: "John",  number: "1234567");
        System.out.println(validation.isValid());
        System.out.println(validation.getError().intersperse(", "));
    }
}
```

Validation

Functional Concepts



Pattern Matching: Destrukturieren von Objekten

```
final CreditCard cc = new CreditCard( owner: "John", ImmutableCreditCardNumber.builder().number(123456789L).build());

if (cc != null && "John".equals(cc.getOwner())) {
    final CreditCardNumber ccNumber = cc.getNumber();
    if (ccNumber != null) {
        System.out.println(String.format("Creditcard of %s with number %s", cc.getOwner(), ccNumber.getNumber()));
    }
}


Long number = Match(cc).of(
        Case($CreditCard($( prototype: "John"), $CreditCardNumber($())), (name, no) -> no.getNumber()),
        Case($(), () -> 0L)
);
System.out.println(number);
```

Agenda

1. Warum funktional programmieren?
2. Java ist doch schon funktional, oder?
3. Erweiterte funktionale Konzepte
4. **Fazit und Ausblick**

4

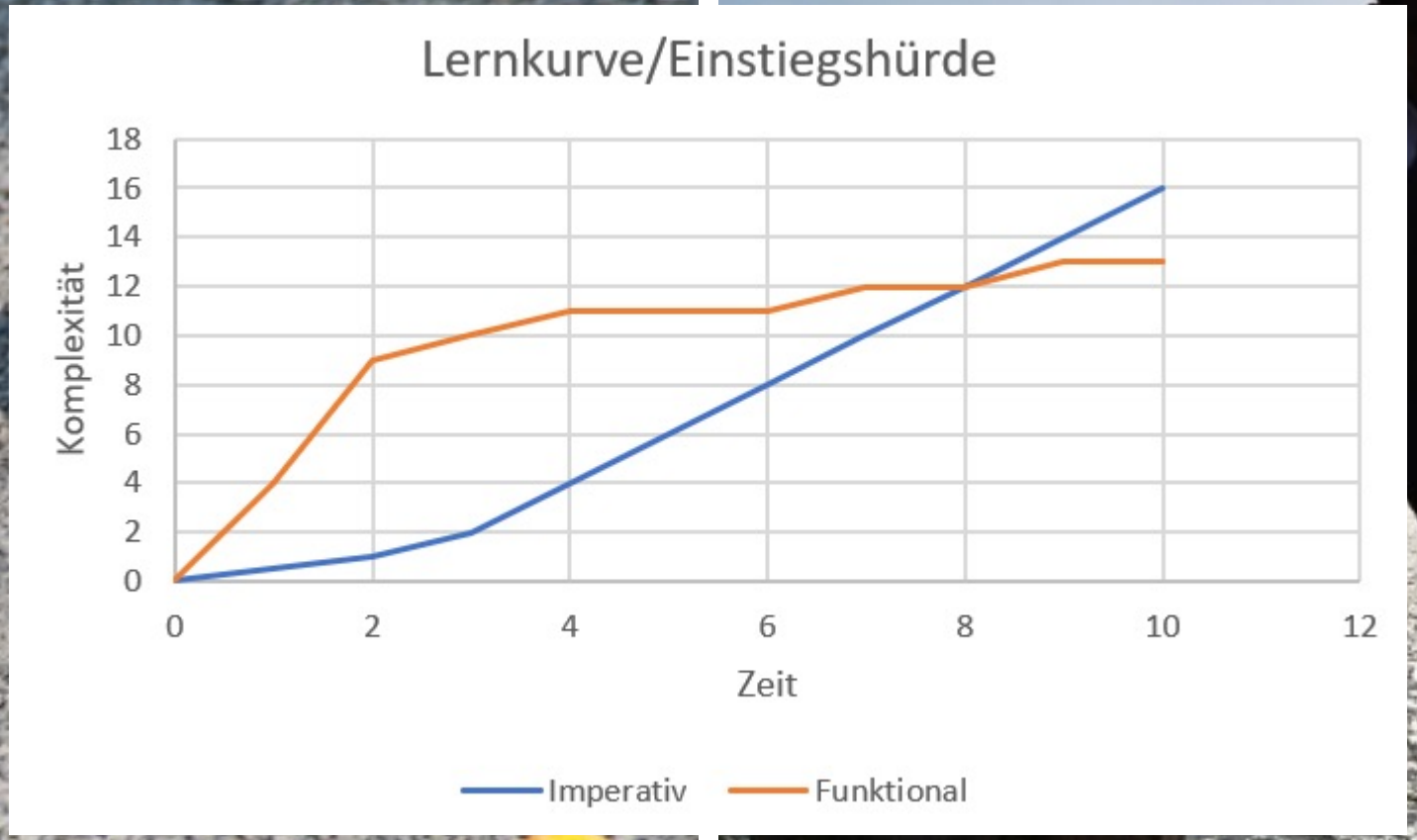


Foto von tomwieden, CC0 Public Domain Lizenz, https://pixabay.com/de/boule-kugeln-spielen-frankreich-141004/

Foto von Mohamed Nuzrath, CC0 Public Domain Lizenz, https://pixabay.com/de/kick-martial-arts-krieger-185384/

Imperativ vs. Funktional

Imperativ:

Wie erreiche ich mein Ziel?

Funktional:

Was will ich erreichen?

Vorteile funktionaler Programmierung

leicht verständlich, einfach zu schlussfolgern

seiteneffektfrei

einfach test-/debugbar

leicht parallelisierbar

modularisierbar und einfach wieder zusammenführbar

hohe Code-Qualität

oder höher

+

Immutables stars 1,662

VAVR.IO

Project Lombok

Links

Code-Beispiele

- https://github.com/sippsack/jvm-functional-language-battle

Learn You a Haskell for Great Good!

- http://learnyouahaskell.com/chapters

LYAH (Learn You a Haskell) adaptions for Frege

- https://github.com/Frege/frege/wiki/LYAH-adaptions-for-Frege

Onlinekurs TU Delft (FP 101):

- https://courses.edx.org/courses/DelftX/FP101x/3T2014/info

Links

Vavr

- http://www.vavr.io/

Immutables

- http://immutables.github.io/

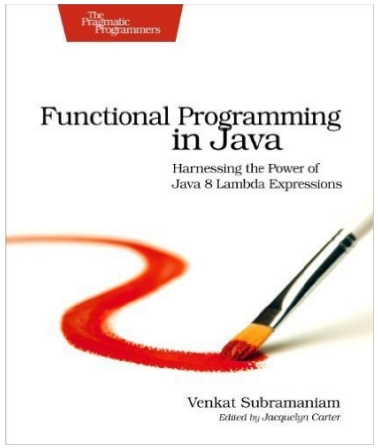
Project Lombok

- https://projectlombok.org/

Functional Java

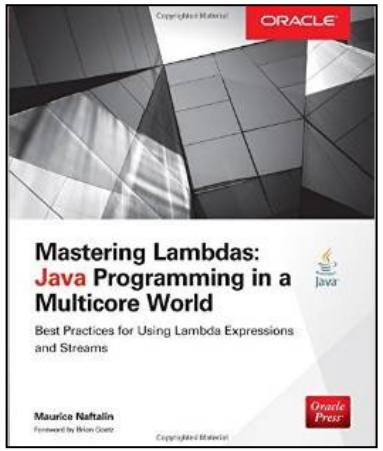
- http://www.functionaljava.org/

Literaturhinweise



Functional Programming in Java: Harnessing the Power Of Java 8 Lambda Expressions

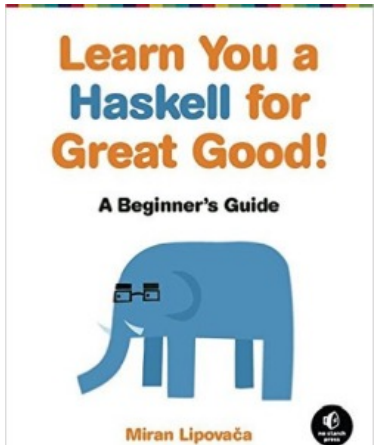
- **Venkat Subramaniam**
- **The Pragmatic Programmers, Erscheinungsdatum: Februar 2014**
- **ISBN: 978-1-93778-546-8**
- **Sprache: Englisch**



Mastering Lambdas

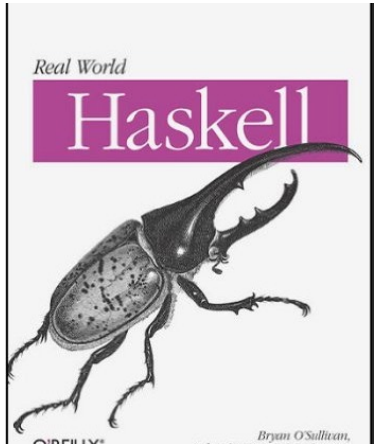
- **Maurice Naftalin**
- **Oracle Press**
- **Erscheinungsdatum: Oktober 2014**
- **ISBN: 0071829628**
- **Sprache: Englisch**

Literaturhinweise



Learn You a Haskell for Great Good!: A Beginner's Guide

- **Miran Lipovaca**
- **No Starch Press, Erscheinungsdatum: April 2011**
- **ISBN: 978-1593272838**
- **Sprache: Englisch**



Real World Haskell

- **Bryan O'Sullivan und John Goerzen**
- **O'Reilly, Erscheinungsdatum: 2010**
- **ISBN: 978-0596514983**
- **Sprache: Englisch**

Vielen Dank.

Ich freue mich auf Eure Fragen!

Falk Sippach

fs@embarc.de

@sippsack

➔ xing.to/fsi