

Einführung in die funktionale Programmierung mit Java

FALK SIPPACH // EMBARC

JCON 2022

Donnerstag, 22. September 2022, 17:00 Uhr



1

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.



2

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

🐦 @sipsack

➔ xing.to/fsi



Einführung in die funktionale Programmierung mit Java

embarc.de

3

3

Workshop Functional Programming mit Gregor



3 – 8 Stunden



Hands-on



Ensemble
Programming



Contact us!
falk@jug-da.de



Pure Functions, Referentielle Transparenz, Persistent/Functional Data Structures,
Higher Order Functions, Komposition, Currying, Partial Function Application, ...

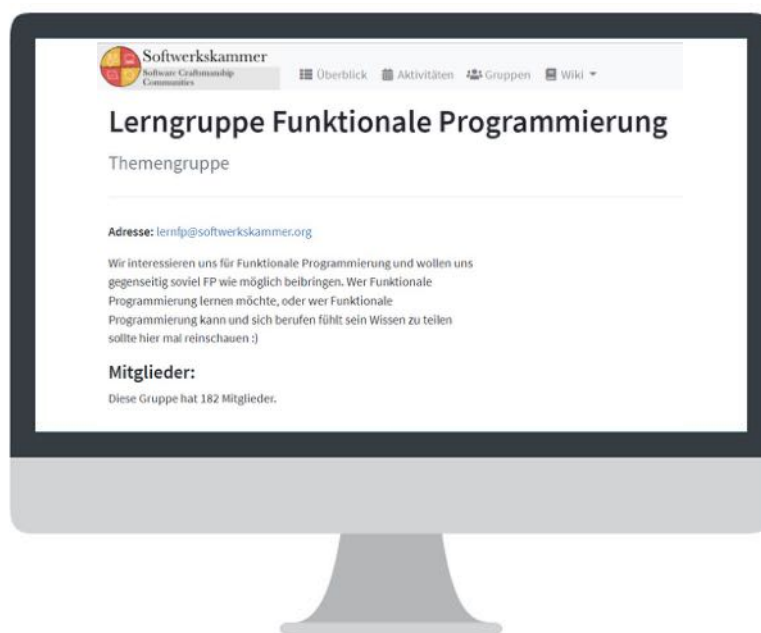


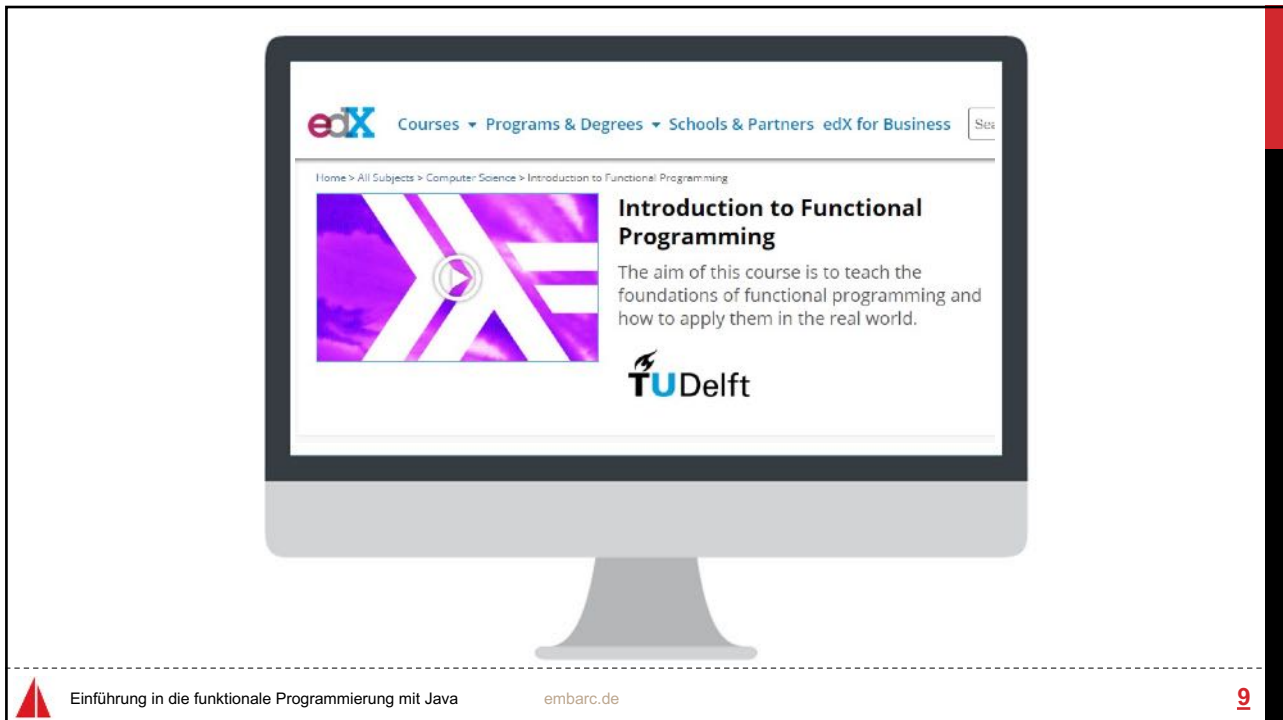
Einführung in die funktionale Programmierung mit Java

embarc.de

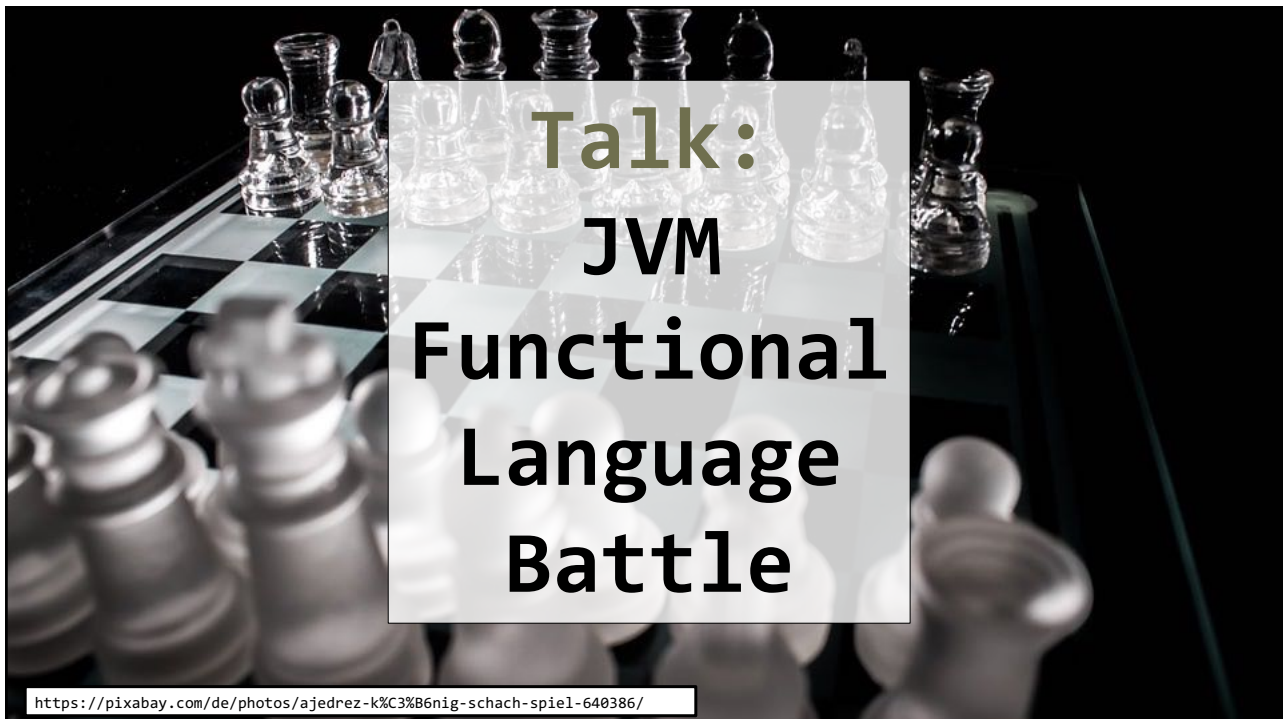
4

4





9



10

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



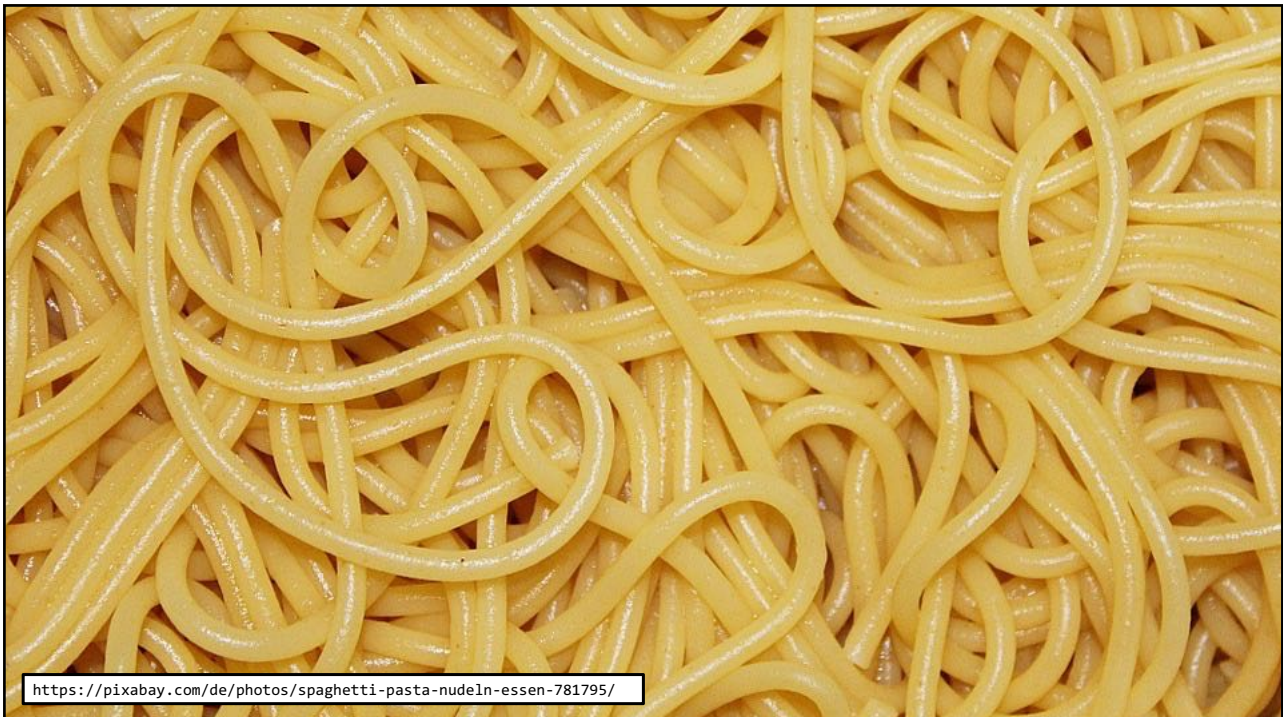
```

3 public class FooAlgorithm{
4     public static boolean isValid(long number) {
5         int sum = 0;
6         boolean alternate = false;
7         while(number > 0) {
8             long digit = number % 10;
9             if (alternate) {
10                 sum += 2 * digit;
11                 if (digit >= 5) {
12                     sum -= 9;
13                 }
14             } else {
15                 sum += digit;
16             }
17             number = number / 10;
18             alternate = !alternate;
19         }
20         return sum % 10 == 0;
21     }
22 }

```



13



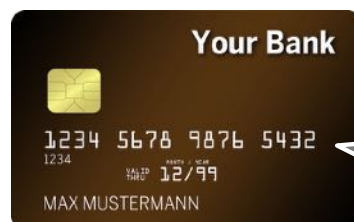
<https://pixabay.com/de/photos/spaghetti-pasta-nudeln-essen-781795/>

14

```

3 public class FooAlgorithm{
4     public static boolean isValid(long number) {
5         int sum = 0;
6         boolean alternate = false;
7         while(number > 0) {
8             long digit = number % 10;
9             if (alternate) {
10                 sum += 2 * digit;
11                 if (digit >= 5) {
12                     sum -= 9;
13                 }
14             } else {
15                 sum += digit;
16             }
17             number = number / 10;
18             alternate = !alternate;
19         }
20         return sum % 10 == 0;
21     }
22 }

```



rein clientseitige
Prüfung möglich

Prüfsummen- Berechnung



```

3 public class LuhnAlgorithm {
4     public static boolean isValid(long number) {
5         int sum = 0;
6         boolean alternate = false;
7         while(number > 0) {
8             long digit = number % 10;
9             if (alternate) {
10                sum += 2 * digit;
11            }
12            if (digit >= 5) {
13                sum -= 9;
14            }
15            else {
16                sum += digit;
17            }
18            number = number / 10;
19            alternate = !alternate;
20        }
21        return sum % 10 == 0;
22    }
}

```

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
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															



```

3 public class LuhnAlgorithm {
4     public static boolean isValid(long number) {
5         int sum = 0;
6         boolean alternate = false;
7         while(number > 0) {
8             long digit = number % 10;
9             if (alternate) {
10                 sum += 2 * digit;
11                 if (digit >= 5) {
12                     sum -= 9;
13                 }
14             } else {
15                 sum += digit;
16             }
17             number = number / 10;
18             alternate = !alternate;
19         }
20         return sum % 10 == 0;
21     }
22 }

```

Variablen

Schleifen

Verzweigungen

Tiefe der
Verschachtelung

Zustands-
änderungen

Reihenfolge
nicht beliebig

Fehler-
behandlung?



**But wait,
isn't
Java 8
already
functional?**

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

Agenda



2

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



Funktionslitterale 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

```

1 package de.oio.luhn.thomas_much;
2
3 import java.util.PrimitiveIterator;
4 import java.util.stream.IntStream;
5
6 public class Luhn {
7     public static boolean isValid(String number) {
8         PrimitiveIterator.OfInt faktor =
9             IntStream.iterate(1, i -> 3 - i).iterator();
10        return (new StringBuilder(number)
11            .reverse()
12            .chars()
13            .map(c -> faktor.nextInt() * (c - '0'))
14            .reduce(0, (a, b) -> a + b / 10 + b % 10) % 10) == 0;
15    }
16 }

```

OHNE Zustandsänderung

Generator: 1 2 1 2 ...

nach Idee von Thomas Much



Funktionskomposition

Verketten/Komposition von Teil-Funktionen: $(f \cdot g)(x) == f(g(x))$

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));
```



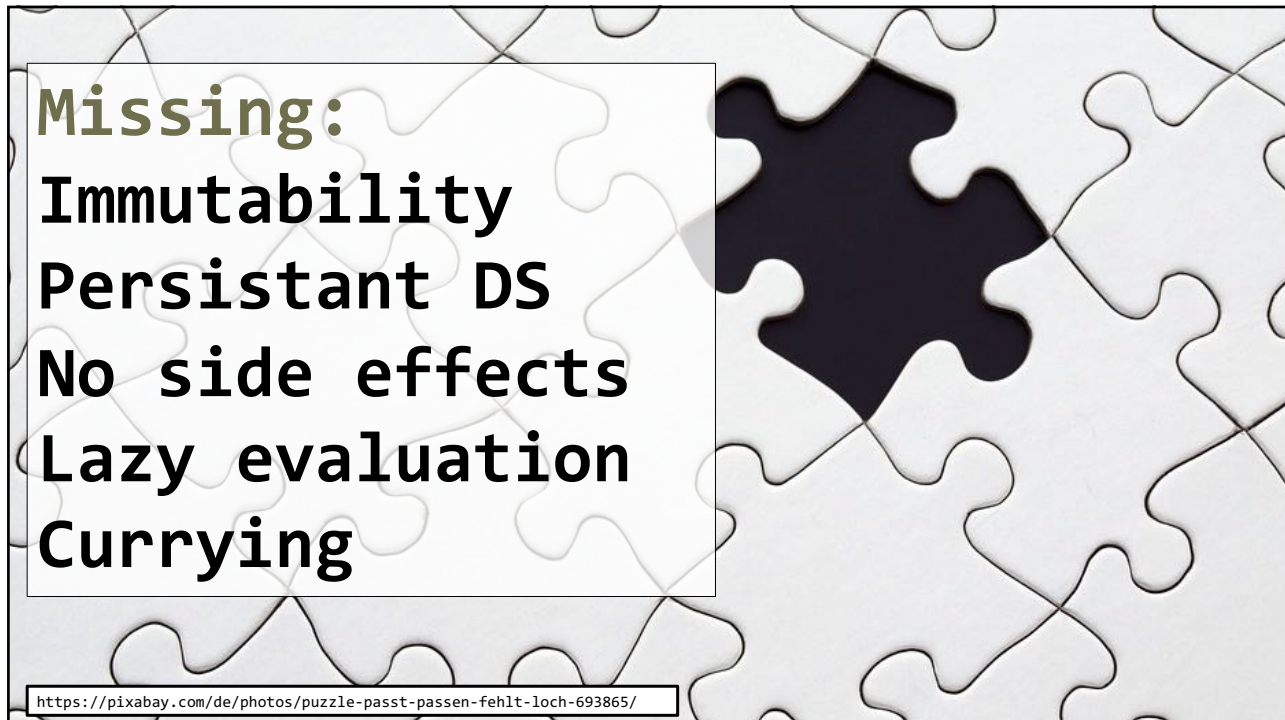
Currying und partielle Funktionsaufrufe

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
Persistent DS
No side effects
Lazy evaluation
Currying

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

27



PROJECT REACTOR

RxJava

VERT.X

Immutables  1,662

VAVR.io

Project Lombok

JDeferred
Java Deferred / Promise library

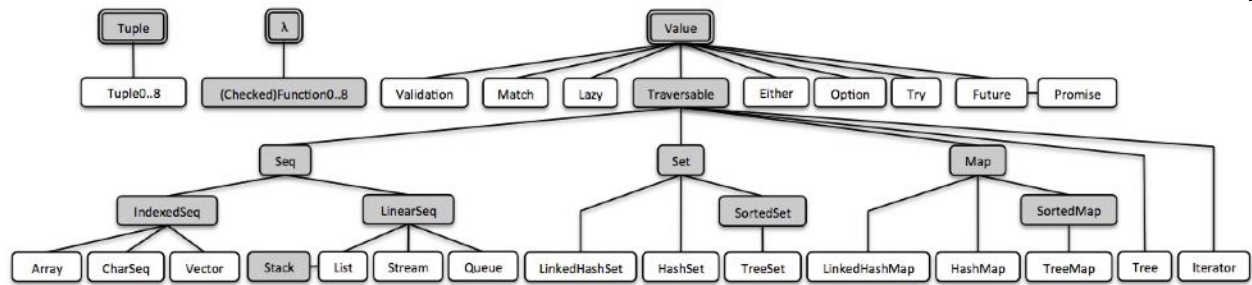
functional.java

 Einführung in die funktionale Programmierung mit Java

embarc.de

28

28



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



29

Agenda



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

3



30



Dr. Gernot Starke
@gernotstarke

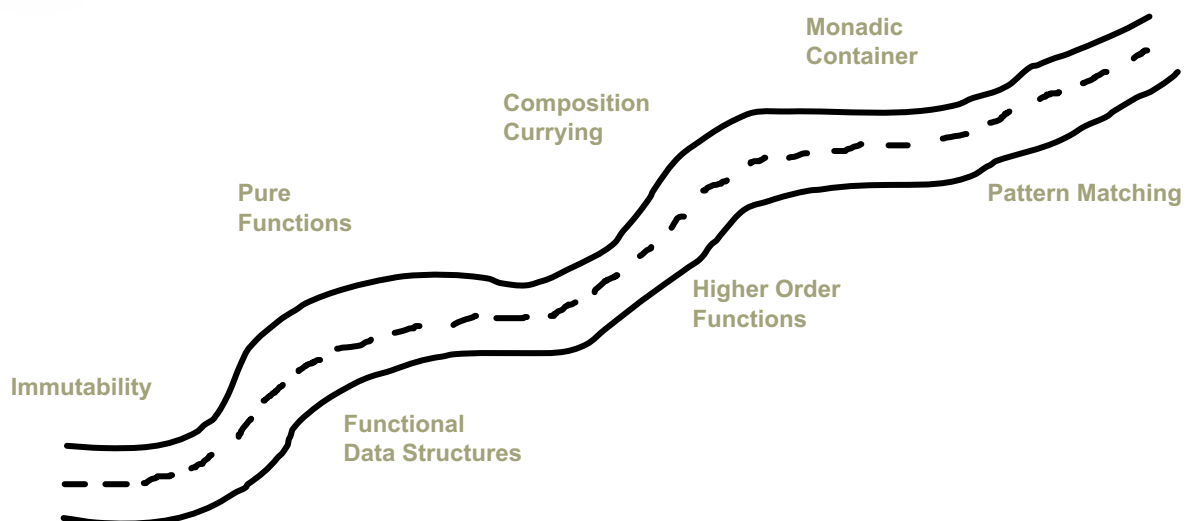
To understand recursion, you need to understand recursion first.

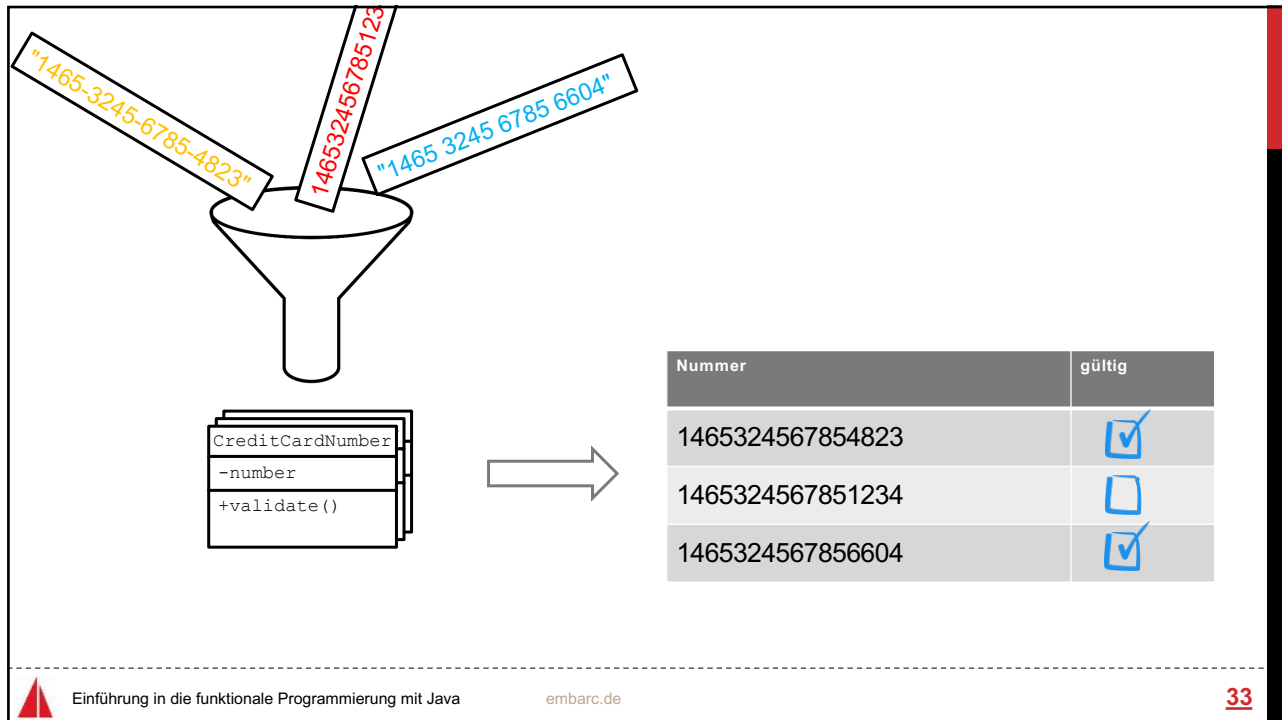
[Tweet übersetzen](#)

12:30 nachm. · 18. Dez. 2019 · [Twitterrific for Mac](#)



Functional Concepts





33

Funktionale Konzepte

- Immutability/Value Types
- Seiteneffektfreiheit/Referentielle Transparenz/Pure Functions
- Immutable/Persistent/Functional Data Structures
- Funktionen/Komposition/Currying/Partielle Funktionsaufrufe
- Funktionslitterale/Higher Order Functions
- Values/Monadic Container
- Pattern Matching

1

2

3

4

5

6

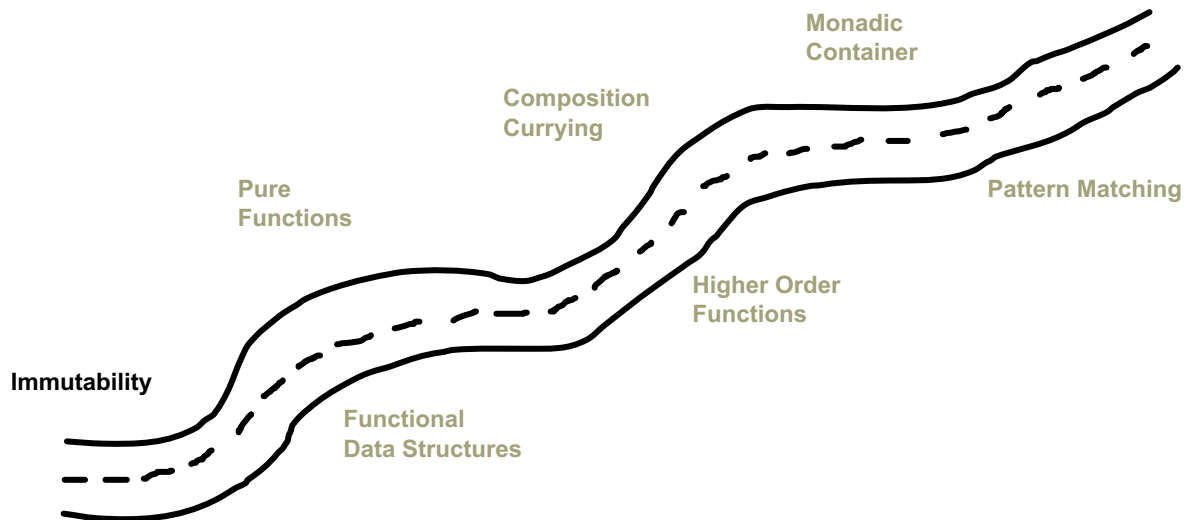
7

Einführung in die funktionale Programmierung mit Java embarc.de **34**

34



Functional Concepts



Einführung in die funktionale Programmierung mit Java

embarc.de

35

35

Bauplan immutable Klasse

Keine Mutatoren zur Verfügung stellen

- zustandsverändernde Methoden, z.B. „setter“

Überschreiben von Methoden verhindern

- Klasse final setzen

Alle Felder final setzen

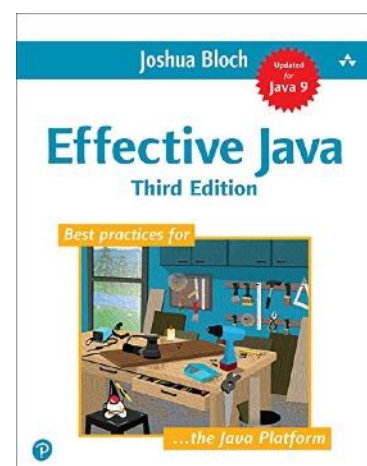
- Ausnutzen der System Restriktionen

Alle Felder private setzen

- Verhindert direktes Ändern durch Clients

Exklusiven Zugang zu mutable Feldern gewährleisten

- Defensive Copies in Konstruktoren, Accessoren und readObject()



Einführung in die funktionale Programmierung mit Java

embarc.de

36

36

Konsequenzen Immutability

Vorteile

genau ein Zustand

Thread-safe

Instanzen können gemeinsam genutzt werden

Auch innere Zustände können gemeinsam genutzt werden

Nachteile

Jeder einzelne Zustand benötigt ein eigenes Objekt

Alternative für vorhersagbare Operationen

Ansonsten öffentliche mutable „Companion Class“

▪ vgl. String und StringBuffer



Einführung in die funktionale Programmierung mit Java

embarc.de

37

37

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

```

optional:
Builder
Lazy-Evaluierung
....

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

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



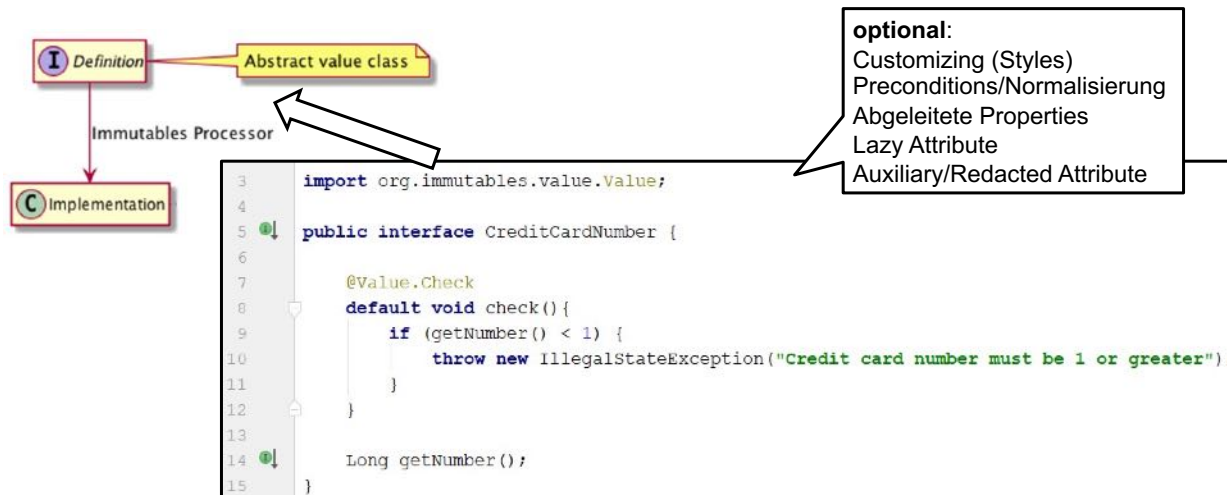
Einführung in die funktionale Programmierung mit Java

embarc.de

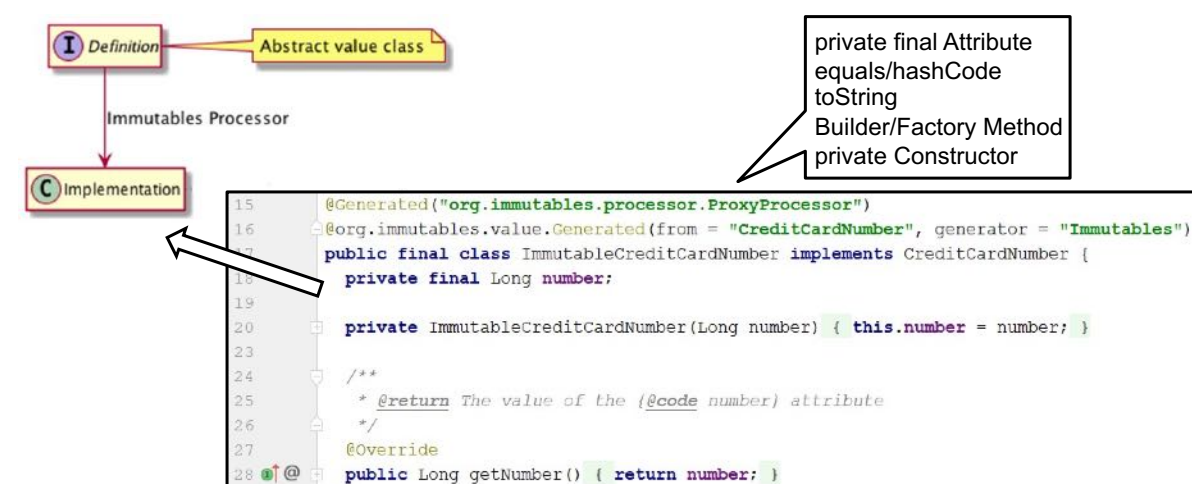
38

38

Immutables - Definition



Immutables - Implementierung



Immutable - Verwendung

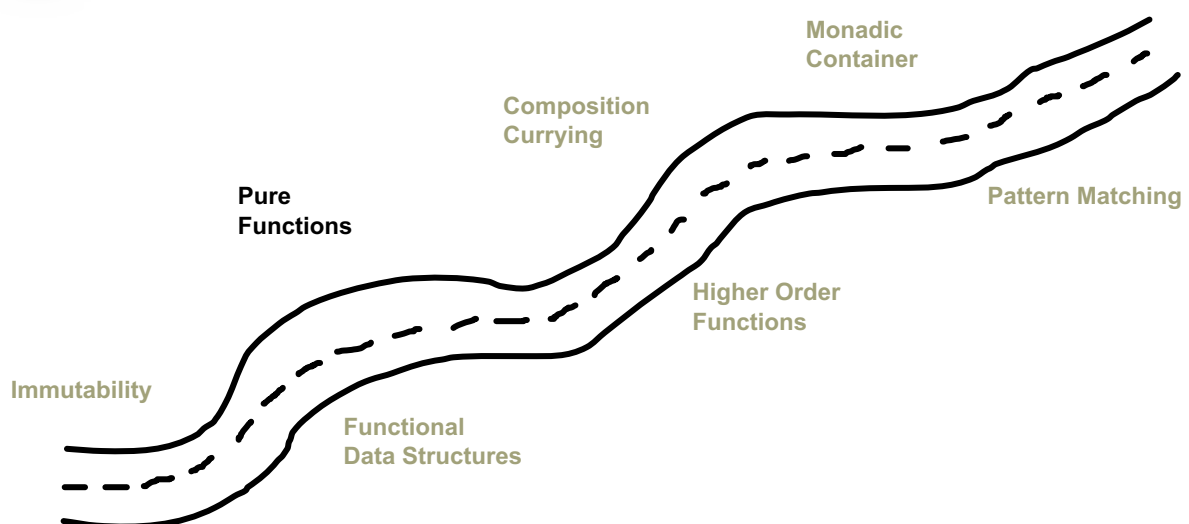
```

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”

aus G. Trefs, F. Sippach: Workshop zu Functional Programming, JavaLand 2022



Einführung in die funktionale Programmierung mit Java

embarc.de

43

43

“Pure functions have no side effects”

aus G. Trefs, F. Sippach: Workshop zu Functional Programming, JavaLand 2022



Einführung in die funktionale Programmierung mit Java

embarc.de

44

44

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

aus G. Trefs, F. Sippach: Workshop zu Functional Programming, JavaLand 2022



Einführung in die funktionale Programmierung mit Java

embarc.de

45

45

Side effects

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

aus G. Trefs, F. Sippach: Workshop zu Functional Programming, JavaLand 2022



Einführung in die funktionale Programmierung mit Java

embarc.de

46

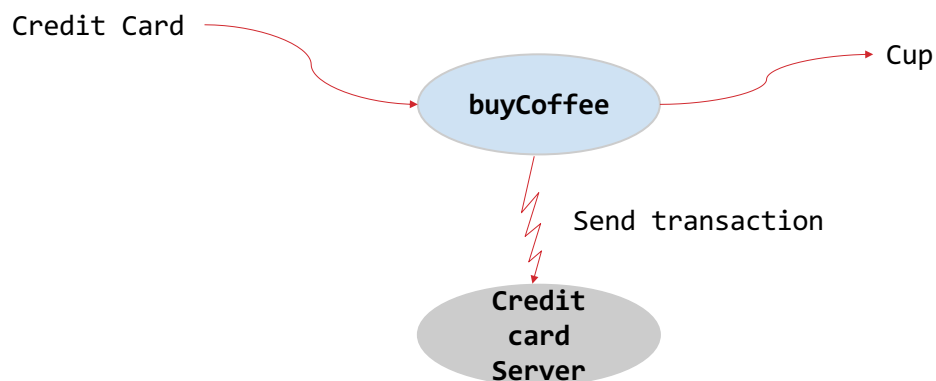
46

Side effects

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



Side effects



aus G. Trefs, F. Sippach: Workshop zu Functional Programming, JavaLand 2022



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”**

aus G. Trefs, F. Sippach: Workshop zu Functional Programming, JavaLand 2022



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

aus G. Trefs, F. Sippach: Workshop zu Functional Programming, JavaLand 2022



`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

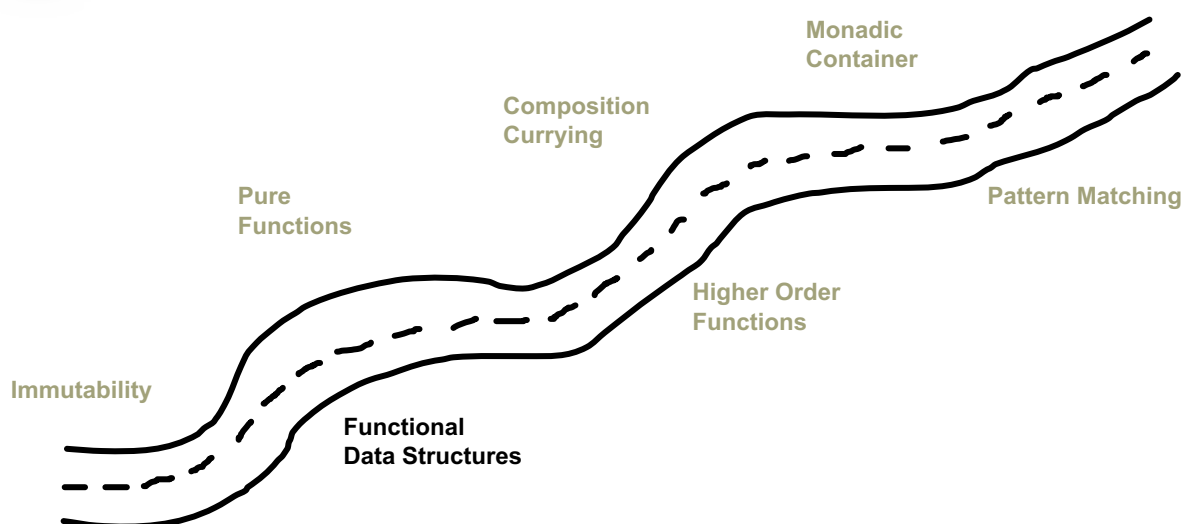
Immutable Datentypen + referentiell transparente Funktionen!



55



Functional Concepts



56

Java Collections sind änderbar!

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

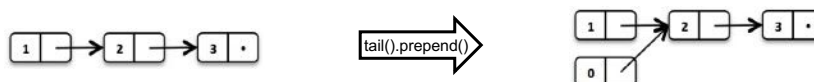
```
List<String> list = Collections
    .unmodifiableList(otherList);

list.add("Boom");
```



Persistente/Funktionale Datenstrukturen in Vavr

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



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



Einführung in die funktionale Programmierung mit Java

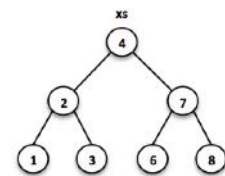
embarc.de

59

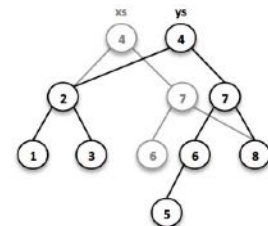
59

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



Einführung in die funktionale Programmierung mit Java

embarc.de

60

60

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

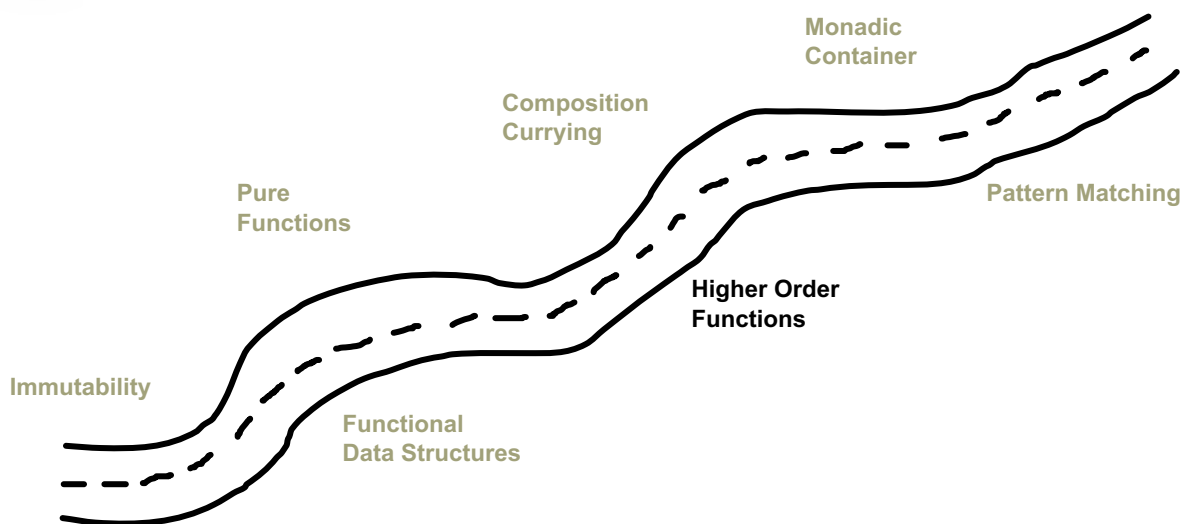
```
List<User> result = List.ofAll(users)
    .filter(user ->
        Try.of(user::validateAddress)
            .getOrElse(false)
    )
    .map(user -> user.name);
```

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

java.util.List



Functional Concepts



Functions are values.

**A function can be
computed, passed around
and returned.**

aus G. Trefs, F. Sippach: Workshop zu Functional Programming, JavaLand 2022



Einführung in die funktionale Programmierung mit Java

embarc.de

63

63

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

aus G. Trefs, F. Sippach: Workshop zu Functional Programming, JavaLand 2022



Einführung in die funktionale Programmierung mit Java

embarc.de

64

64

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;
```

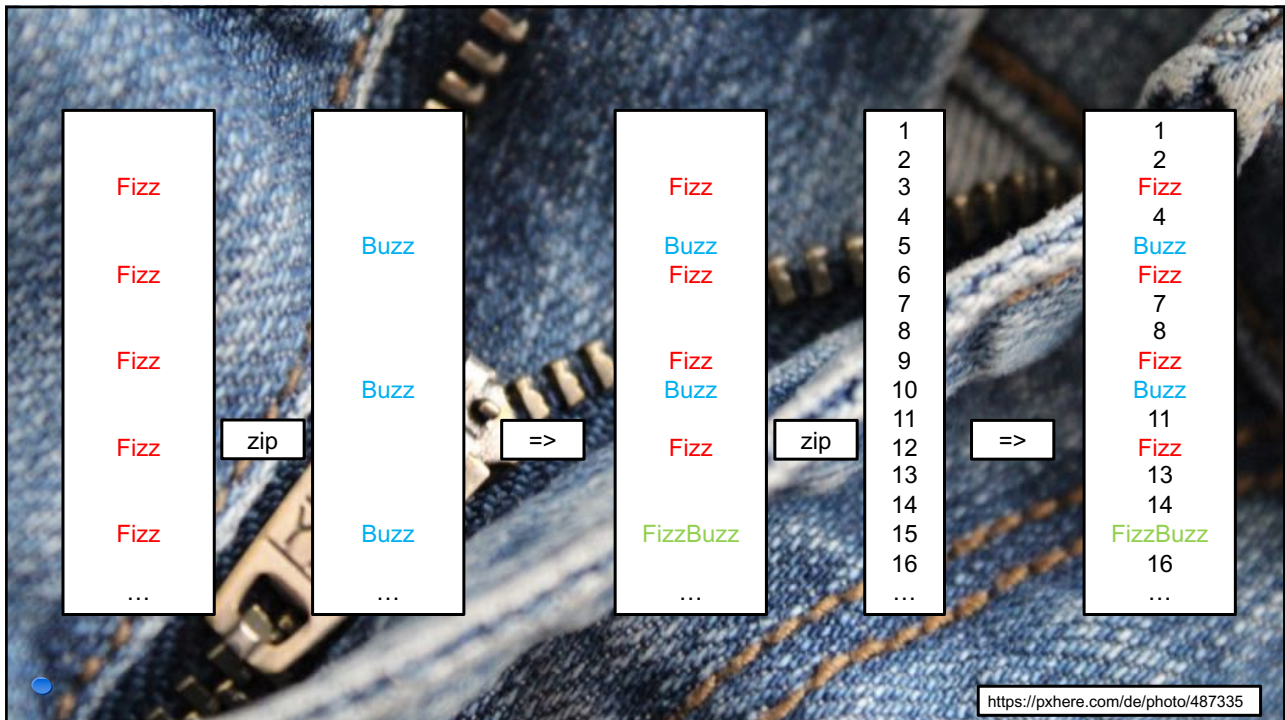



```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
...
```

```
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);
```

<https://gtrefs.github.io/code/functional-fizzbuzz/>

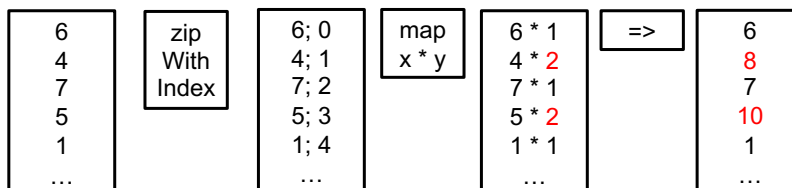
<https://pxhere.com/de/photo/487335>



67

Luhn: Zippen zum Verdoppeln jeder zweiten Ziffer

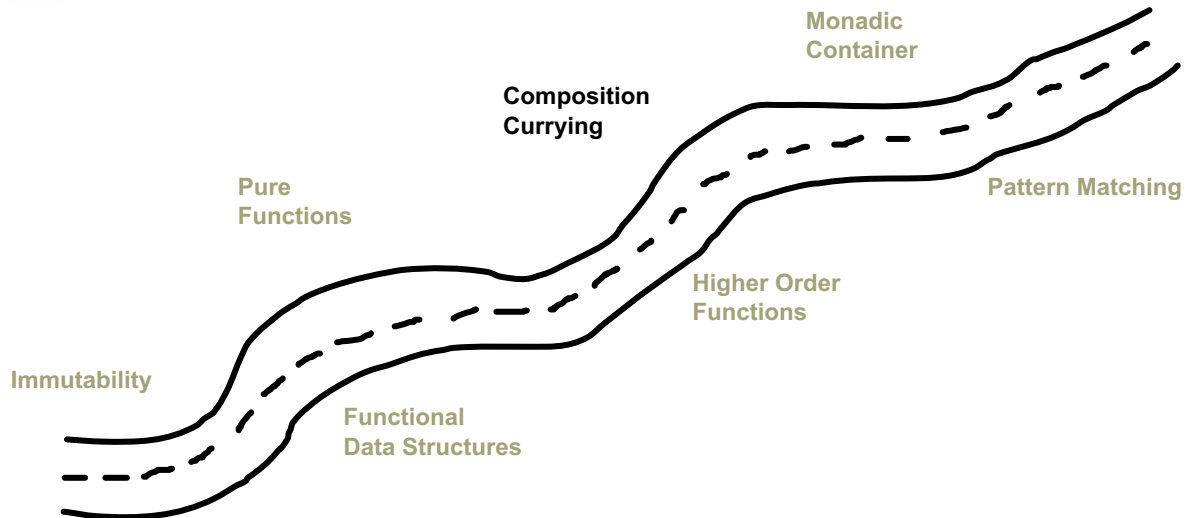
```
static Function1<Seq<Integer>, Seq<Integer>> double2nd =
  digits -> digits.zipWithIndex()
    .map(t -> t._1 * (t._2 % 2 + 1));
```



68



Functional Concepts



Einführung in die funktionale Programmierung mit Java

embarc.de

69

69

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

aus G. Trefs, F. Sippach: Workshop zu Functional Programming, JavaLand 2022



Einführung in die funktionale Programmierung mit Java

embarc.de

70

70

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

function composition

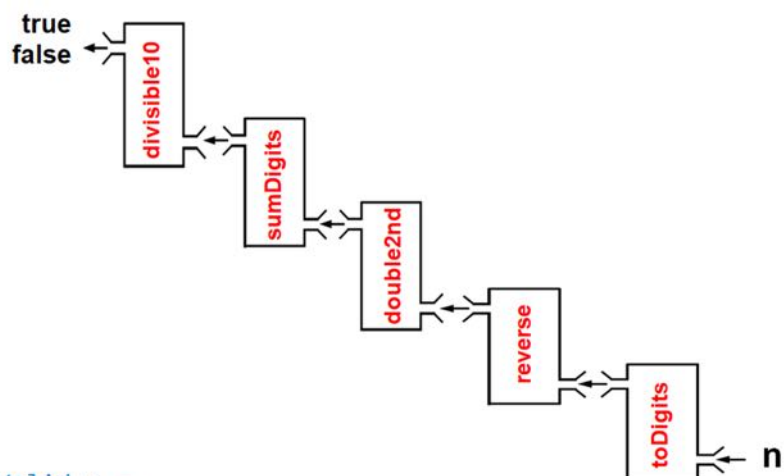


Einführung in die funktionale Programmierung mit Java

embarc.de

71

71



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



72 Einführung in die funktionale Programmierung mit Java

embarc.de

72

72

Luhn Algorithmus in funktional

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



Validierungsfunktion

Teilbar durch 10?

Aufsummieren

jede 2. verdoppeln

Ziffern umdrehen

Aufsplitten in Ziffern



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));

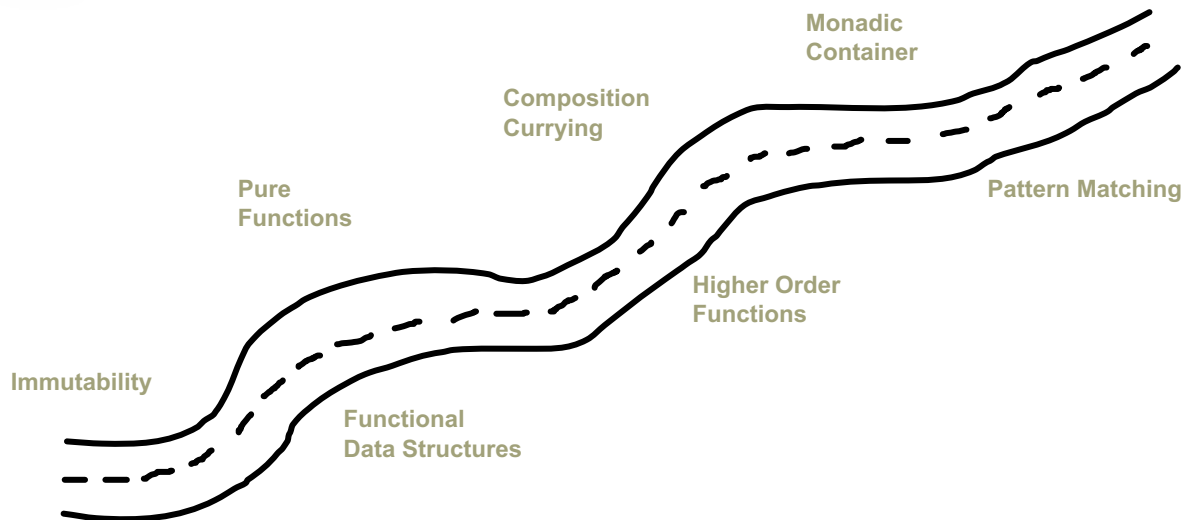
static Function1<Seq<Integer>, Integer> sumDigits = digits ->
  digits.map(i -> i.longValue()).flatMap(toDigits).sum().intValue();

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





Functional Concepts



Einführung in die funktionale Programmierung mit Java

embarc.de

75

75

"... **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."

aus G. Trefs, F. Sippach: Workshop zu Functional Programming, JavaLand 2022



Einführung in die funktionale Programmierung mit Java

embarc.de

76

76

```
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



Einführung in die funktionale Programmierung mit Java

embarc.de

77

77

```
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



Einführung in die funktionale Programmierung mit Java

embarc.de

78

78

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

aus G. Trefs, F. Sippach: Workshop zu Functional Programming, JavaLand 2022



```
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



Einführung in die funktionale Programmierung mit Java

embarc.de

81

81

```
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(result1).isEqualTo(Option.none());

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

lifting



Einführung in die funktionale Programmierung mit Java

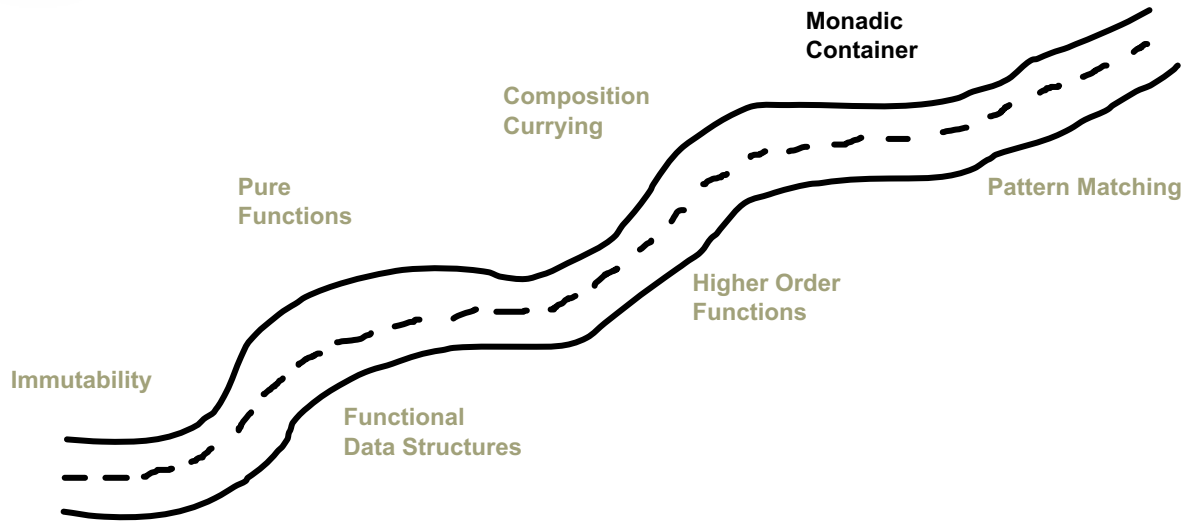
embarc.de

82

82



Functional Concepts



Einführung in die funktionale Programmierung mit Java

embarc.de

83

83

```
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



Einführung in die funktionale Programmierung mit Java

embarc.de

84

84

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

Algebraische Datentypen: Summen- oder Variantentypen



85

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



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

```
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



86

```
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));
    }
}
```

```
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



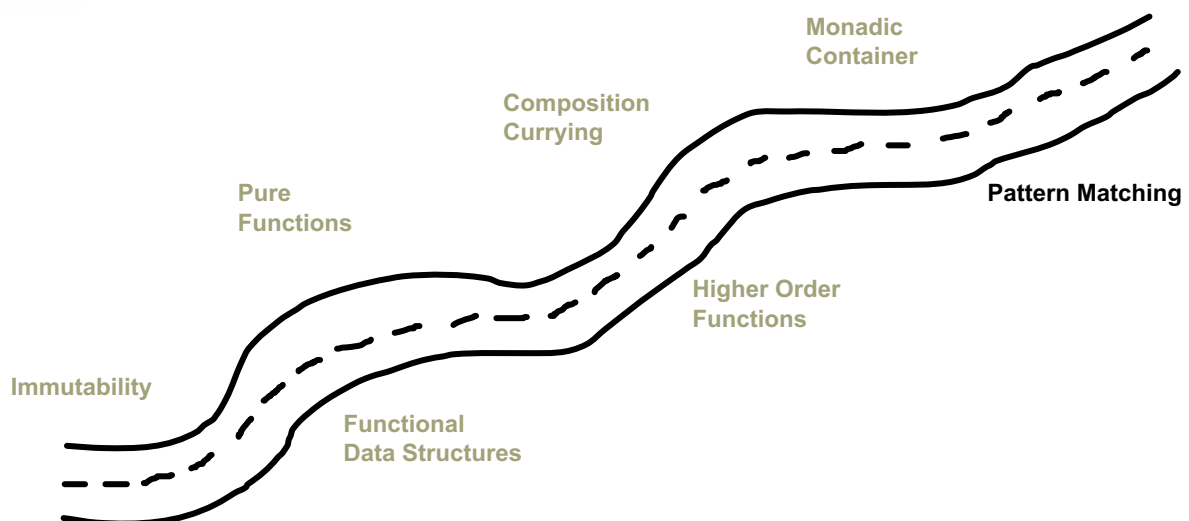
```
public class CreditCardValidator {
    public Validation<Seq<String>, CreditCard> validateNumber(String owner, String number) {
        final CreditCardNumber ccn = CreditCardNumber.from(number);
        return Validation.combine(
            owner == null || owner.isEmpty() ? Validation.invalid("Owner 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);
```



91

Agenda

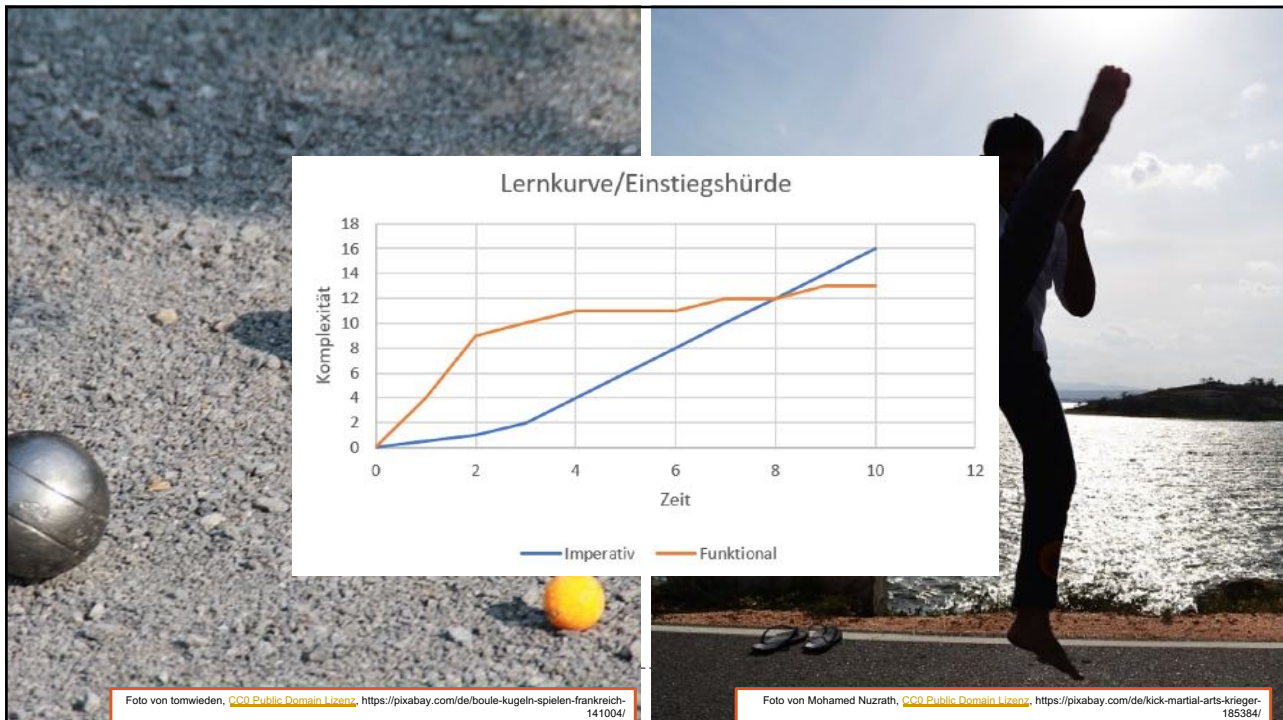


- 1 Warum funktional programmieren?
- 2 Java ist doch schon funktional, oder?
- 3 Erweiterte funktionale Konzepte
- 4 **Fazit und Ausblick**

4



92



93

Imperativ vs. Funktional

Imperativ:

**Wie erreiche
ich mein
Ziel?**

Funktional:

**Was will ich
erreichen?**



94

Vorteile funktionaler Programmierung

leicht verständlich, einfach zu schlussfolgern
 seiteneffektfrei
 einfach test-/debugbar
 leicht parallelisierbar
 modularisierbar und einfach wieder zusammenführbar
 hohe Code-Qualität





+

Immutable

stars 1,662



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/>

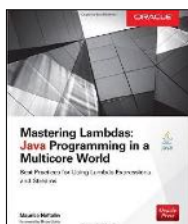


Literaturhinweise



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**

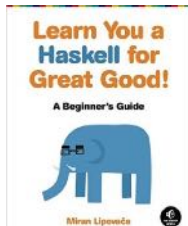


Mastering Lambdas

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

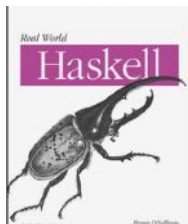


Literaturhinweise



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

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

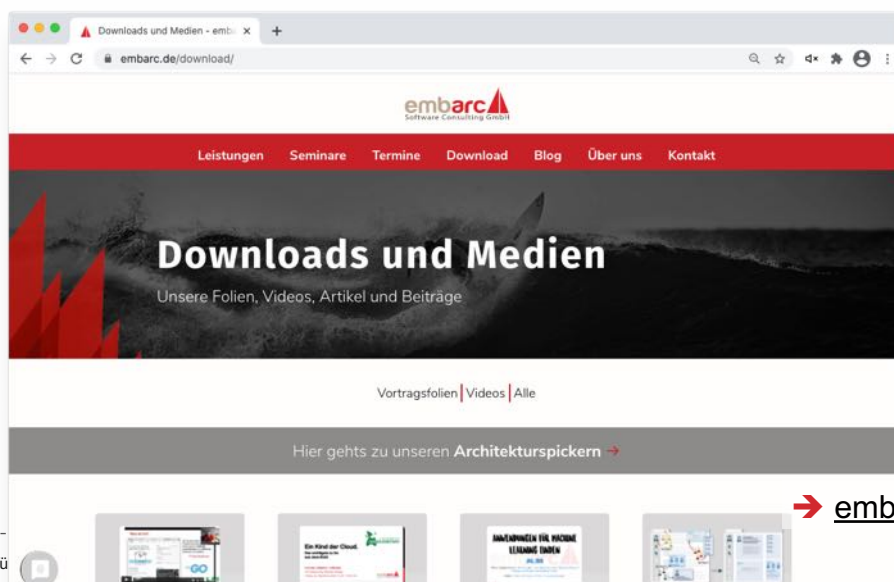


Real World Haskell

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



Folien von heute als PDF zum Download



→ embarc.de/download/



Spicken erlaubt!



Unsere Architektur-Spicker beleuchten die konzeptionelle Seite der Softwareentwicklung.



Spicker #1: „Der Architekturüberblick“

- Welche Zutaten gehören in einen Architekturüberblick?
- Welche Formen bewähren sich in welchen Situationen?
- Wie fertigen Sie einen Architekturüberblick an?

PDF, 4 Seiten
Kostenloser Download.

→ <http://architektur-spicker.de>



Einführung in die funktionale Programmierung mit Java

embarc.de

103

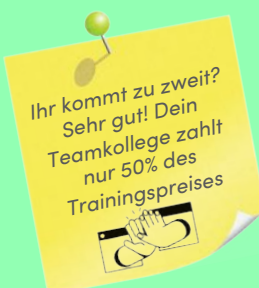
103



EINE KOLLABORATION VON



Wir haben unsere **Kompetenzen gebündelt** und eine einzigartige **Trainingsplattform** geschaffen, die alle relevanten Module und jedes verfügbare Zertifizierungslevel nach den **ISAQB®-Standards** umfasst. Zusammen sind wir die **Software Creators' Academy**!



Aktuelle
Termine:
socreatory.com

Unser Preismodell - Flexibel und transparent

104

Vielen Dank.

Ich freue mich auf Eure Fragen!



Falk Sippach



fs@embarc.de



@sippsack



→ [xing.to/fsi](https://www.xing.to/fsi)

