Funktionale Programmierung

- für vielbeschäftigte Javaentwickler

(Falk Sippach, Till Rauch)

1

Funktionale Programmierung für vielbeschäftigte Javaentwickler

Seit Java 8 halten funktionale Konzepte zunehmend Einzug in die Java-Welt. Trotz moderner Features wie Lambdas, Pattern Matching, Records oder Sealed Classes bleibt in vielen Projekten jedoch Skepsis: Die Lernkurve wirkt hoch und der resultierende Code angeblich unlesbar oder „nicht Java-typisch“. Dabei verspricht funktionale Programmierung geringere Komplexität, klarere Datenflüsse, erwartungskonformes Verhalten und eine deutlich bessere Testbarkeit – Vorteile, die sich gerade in größeren Systemen auszahlen.

In diesem Vortrag untersuchen wir, wie weit man mit modernem Java funktional denken und arbeiten kann. Wir zeigen anhand eines durchgängigen Beispiels zunächst die Eleganz funktionaler Lösungen (z. B. in Haskell) und entwickeln dann Schritt für Schritt eine idiomatische Java-Alternative. Dabei betrachten wir sowohl gut unterstützte Konzepte wie Immutability, Higher-Order Functions und algebraische Datentypen (über Records und Sealed Classes) als auch Bereiche, in denen Java an Grenzen stößt oder zusätzliche Bibliotheken nötig sind – etwa List Comprehensions, Currying, Partial Function Application oder Zipping.

Am Ende wisst ihr, welche funktionalen Prinzipien sich sinnvoll auf Java übertragen lassen, wo Workarounds nötig sind und wie ihr durch kluge Datenmodellierung ungültige Zustände vermeidet, natürlichere Kontrollflüsse erzeugt und die Wartbarkeit eurer Anwendungen verbessert – ohne das Java-Ökosystem zu verlassen.

2

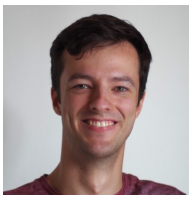
Wer sind wir?

Falk Sippach (embarc)

falk.sippach@embarc.de

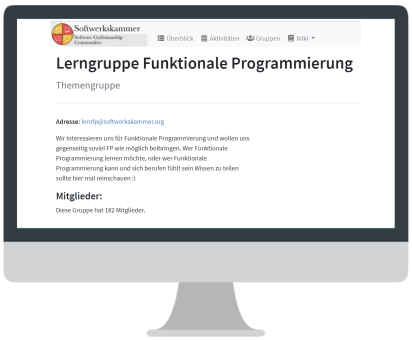
LinkedIn: /in/falk-sippach

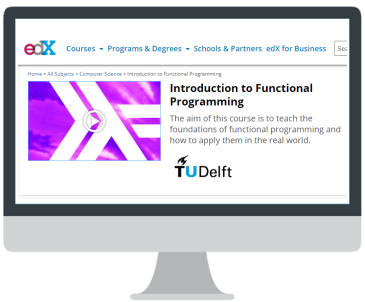
Till Rauch (Active Group)

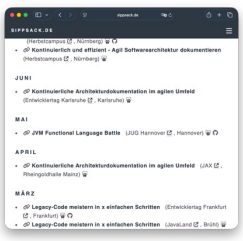


till.rauch@active-group.de

3







2016:

1. Vortrag: **Functional Language Battle** (Java vs. ...)

4

Agenda

Warum funktionale Programmierung?

Ist Java eigentlich auch funktional?

Kann Java mit Haskell mithalten?

Wohin geht die Reise bei Java?

6

Agenda

Warum funktionale Programmierung?

Ist Java eigentlich auch funktional?

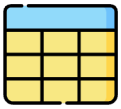
Kann Java mit Haskell mithalten?

Wohin geht die Reise bei Java?

7

Warum funktionale Programmierung?

Abstraktion



Trennung von Logik und Daten

Fallstricke bei imperativer Programmierung

Lesbarer/ Testbarer Code

Modularisierung

8

Was gehört dazu?

Kernfeatures

- Pure Functions
- Funktionskomposition
- Immutability
- First-Class Functions
- Referenzielle Transparenz
- …

Sprachfeatures

- Pattern Matching
- List Comprehension
- Algebraische Datentypen
- Currying
- …

9

Agenda

Warum funktionale Programmierung?

Ist Java eigentlich auch funktional?

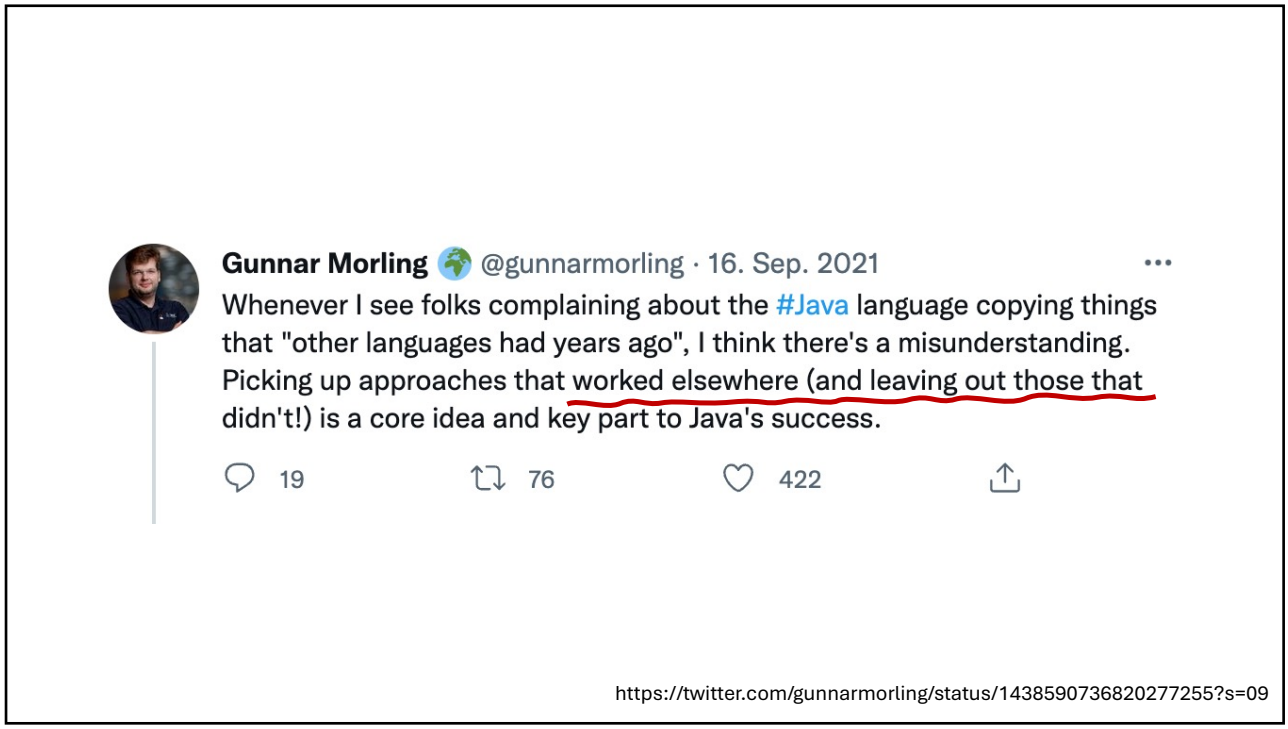
Kann Java mit Haskell mithalten?

Wohin geht die Reise bei Java?

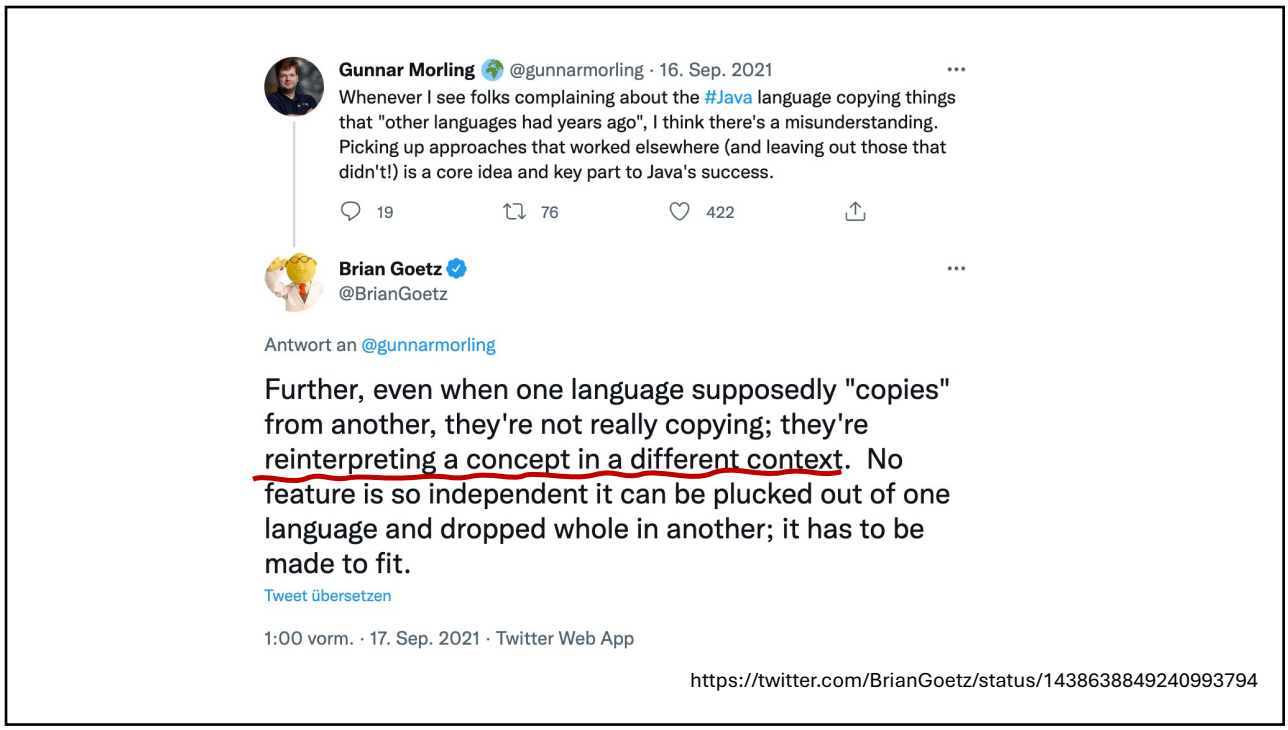
10

Jein!

11


Gunnar Morling @gunnarmorling · 16. Sep. 2021
Whenever I see folks complaining about the #Java language copying things that "other languages had years ago", I think there's a misunderstanding. Picking up approaches that worked elsewhere (and leaving out those that didn't!) is a core idea and key part to Java's success.
19
76
422
https://twitter.com/gunnarmorling/status/1438590736820277255?s=09

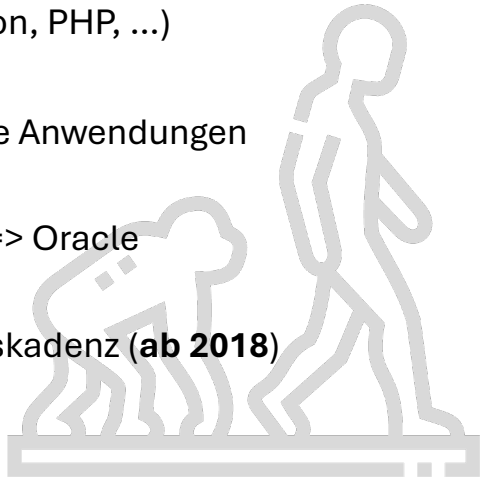
12


Gunnar Morling @gunnarmorling · 16. Sep. 2021
Whenever I see folks complaining about the #Java language copying things that "other languages had years ago", I think there's a misunderstanding. Picking up approaches that worked elsewhere (and leaving out those that didn't!) is a core idea and key part to Java's success.
19
76
422
Brian Goetz
@BrianGoetz
Antwort an @gunnarmorling
Further, even when one language supposedly "copies" from another, they're not really copying; they're reinterpreting a concept in a different context. No feature is so independent it can be plucked out of one language and dropped whole in another; it has to be made to fit.
Tweet übersetzen
1:00 vorm. · 17. Sep. 2021 · Twitter Web App
https://twitter.com/BrianGoetz/status/1438638849240993794

13

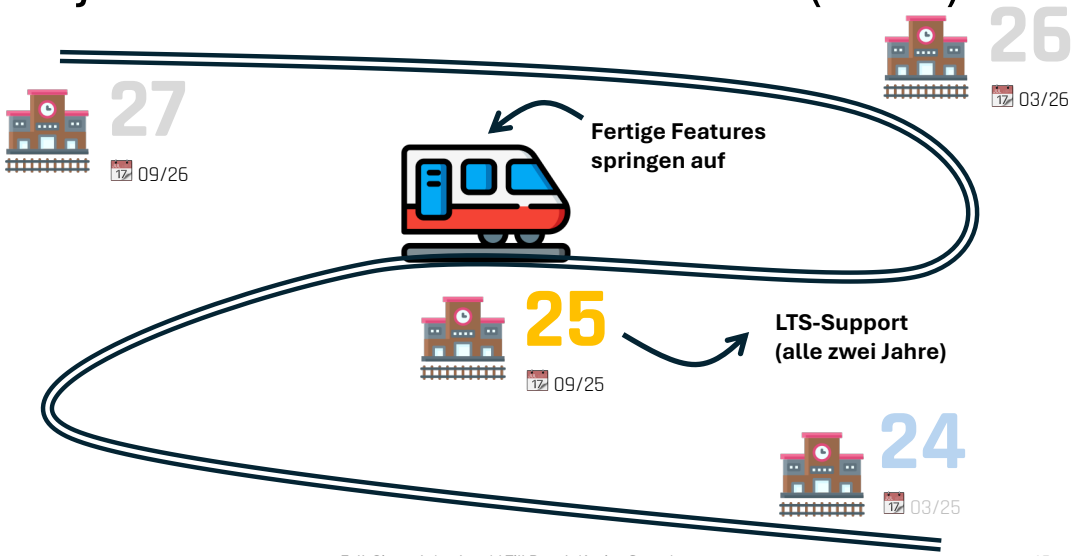
Geschichte von Java

- **30 Jahre** alt (ähnlich wie JavaScript, Python, PHP, ...)
- **90er**: WORA, Applets, **ab 2000**: Enterprise Anwendungen
- Open Sourcing (JCP, JSR, JEP), **2009** Sun => Oracle
- Releases von alle 3 – 5 Jahre auf 6 Monatskadenz (**ab 2018**)



14

Halbjährliche Releases seit Java 10 (2018)



15

Wo war Java theoretisch schon funktional?

- Pure Functions sind "nicht verboten"
- Immutability über einen manuellen, umständlichen Bauplan
- Bibliotheken: Vavr, Functional Java
  - z. B. Persistent Data Structures
  - Pattern Matching
  - Tuple
  - Monaden (Either, Try, Optional, ...)

16

1. FP Offensive mit Java 8 (2014)

- Lambdas und Stream-API
- Higher Order Functions
- Function Composition (compose() & andThen())
- Monaden (Optional/Maybe, Either, ...)

17

2. Offensive: Post Java 8 (ab 2018)

- Pattern Matching (Type Patterns & Pattern Matching for switch)
- switch-Expressions
- Sealed Classes
- Records
- Record Patterns (Deconstruction) & Unnamed Patterns
- Value Classes (Primitive Type Patterns)
- Virtual Threads, Structured Concurrency

Legende:
nützlich
Zukunft
nicht relevant

18

Agenda

Warum funktionale Programmierung?

Ist Java eigentlich auch funktional?

Kann Java mit Haskell mithalten?

Wohin geht die Reise bei Java?

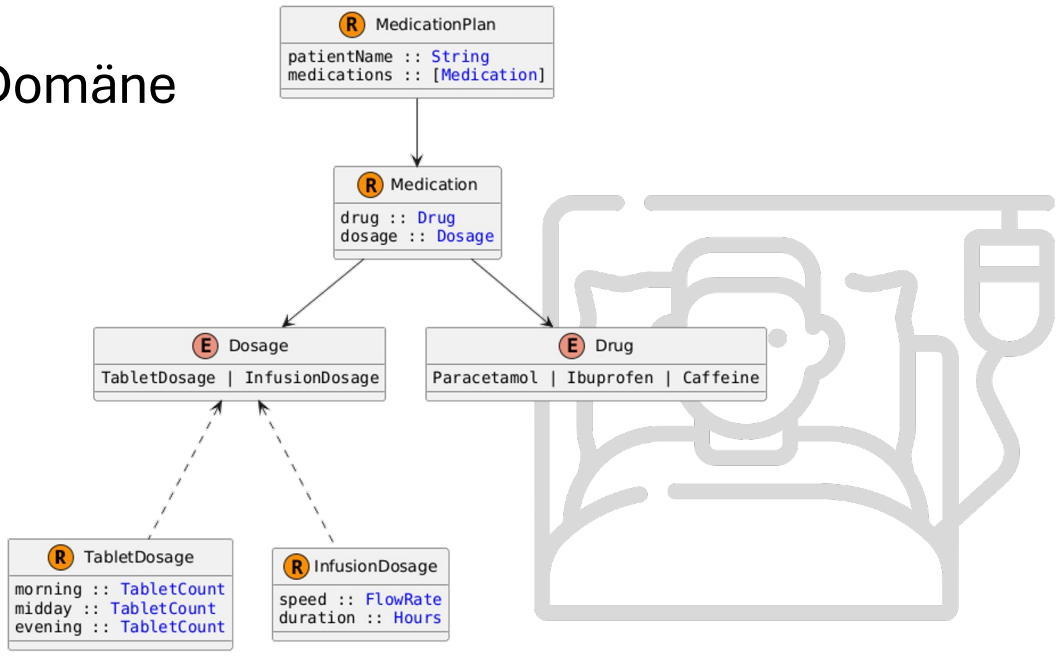
19

Jein!

... mit Tendenz zu "nicht so richtig"!

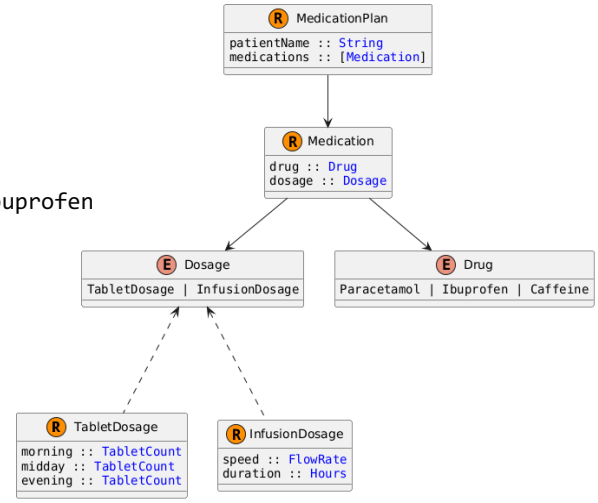
20

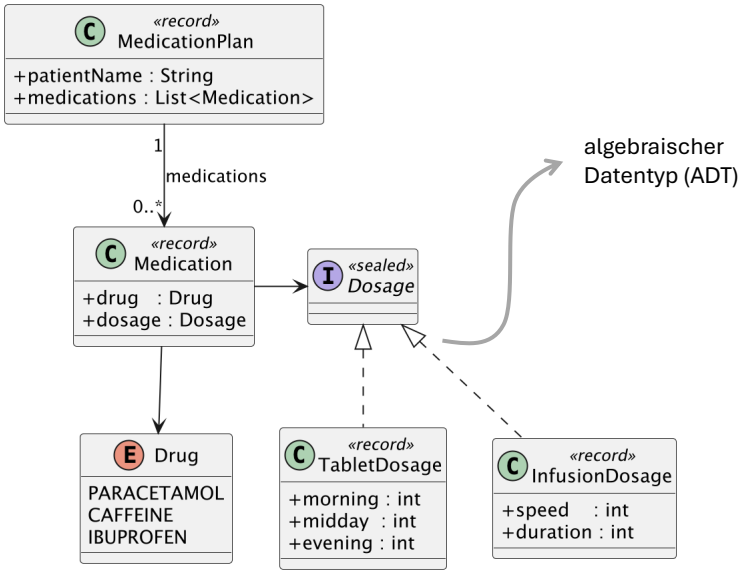
Die Domäne



22

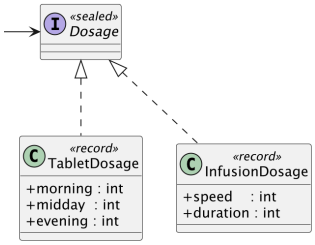
```haskell
data MedicationPlan = MedicationPlan
  { patientName :: String,
    medications :: [Medication]
  }

data Medication = Medication
  { drug :: Drug,
    dosage :: Dosage
  }

data Drug = Paracetamol | Caffeine | Ibuprofen
  deriving (Eq, Ord, Show)

data Dosage
  = TabletDosage {morning :: TabletCount,
                  midday  :: TabletCount,
                  evening :: TabletCount}
  | InfusionDosage {speed :: FlowRate,
                    duration :: Hours}
```





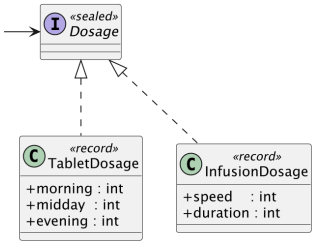
ADT: Dosage

```
data Dosage
 = TabletDosage {morning :: TabletCount,
                 midday  :: TabletCount,
                 evening :: TabletCount}
 | InfusionDosage {speed :: FlowRate, duration :: Hours}
```



ADTs in Java

```
sealed interface Dosage permits TabletDosage, InfusionDosage { }

record TabletDosage(int morning, int midday, int evening) implements Dosage { }

record InfusionDosage(int speed, int duration) implements Dosage { }
```




```
data Dosage
 = TabletDosage {morning :: TabletCount,
                 midday  :: TabletCount,
                 evening :: TabletCount}
 | InfusionDosage {speed :: FlowRate, duration :: Hours}
```

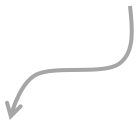
Type Synonyms

```
type TabletCount = Int
type Hours = Int
type FlowRate = Int
```

28

Type Synonyms

```
import Numeric.Natural

type TabletCount = Int
type Hours = Natural
type FlowRate = Natural
```



ADT + “Domänen-Primitive“
verhindern illegale Zustände

29

Type Synonyms: Value Classes in Java

```
sealed interface Dosage permits TabletDosage, InfusionDosage { }

value record TabletDosage(int morning, int midday, int evening) implements Dosage { }

value record InfusionDosage(int speed, int duration) implements Dosage { }
```

"Codes like a class, works like an int!"

Optimierungen: Inlining, Flattening

Null-Restricted Types werden kommen

List<int> => bessere Integration von primitiven und Referenztypen

Valhalla

30

Pure Functions

```haskell
formatMedication :: Medication -> String
formatMedication (Medication drug dosage) =
  show drug ++ ": " ++ formatDosage dosage
```

Destructuring (da kommen wir beim Pattern Matching darauf zurück)

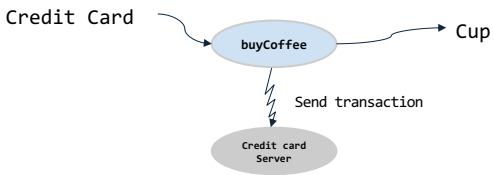
31

Functional Core, Procedural Shell

Pure Functions in Java

```
static String formatMedication(Medication medication) {
  return "%s: %s".formatted(
      medication.drug(),
      formatDosage(medication.dosage()));
}
```

Seiteneffekte:
Variable modifizieren
Exception werfen
Konsolenausgabe
Schreiben in Datei
…



Pattern Matching

```haskell
formatDosage :: Dosage -> String
formatDosage (TabletDosage morning midday evening) =
  show morning ++ "-" ++ show midday ++ "-" ++ show evening
formatDosage (InfusionDosage speed duration) =
  show speed ++ "ml/min for " ++ show duration ++ "h"
```

```haskell
eveningTabletCount :: Dosage -> TabletCount
eveningTabletCount (TabletDosage _ _ evening) = evening
eveningTabletCount (InfusionDosage _ _) = 0
```

Wildcards

Pattern Matching in Java (ab Java 12)

- switch Expression
- Sealed Classes und Records (ADTs)
- Type Patterns (Pattern Matching for instanceof)
- Record Patterns (Deconstruction)
- Unnamed Patterns
- Primitive Type Patterns
- Array Patterns, Deconstruction Patterns

Legende:
bereits fertig
noch in Arbeit
Zukunft

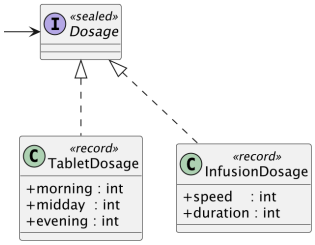
Pattern Matching in Java

```
static String formatDosage(Dosage d) {
  return switch (d) {
    case TabletDosage(int m, int md, int e) ->
        "%d-%d-%d".formatted(m, md, e);
    case InfusionDosage(int s, int dur) ->
        "%d ml/min for %d h".formatted(s, dur);
  };
}
```

switch Expression

Record Patterns,
Parameter werden befüllt

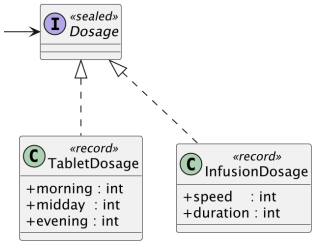
Exhaustiveness Prüfung,
kein default-Branch



Pattern Matching in Java (2)

```
static int eveningTabletCount(Dosage dosage) {
  return switch (dosage) {
    case TabletDosage(_, _, int evening) -> evening;
    case InfusionDosage(_, _) -> 0;
  };
}
```

Unnamed Patterns



Funktionskomposition

```
drugsInPlan :: MedicationPlan -> [Drug]
drugsInPlan (MedicationPlan _ medications) =
  Set.toList
    . Set.fromList
    . map drug
    $ medications
```

Komponiert zu einer Funktion

Funktionskomposition: Semantik

Semantik:

1. Nimm alle medications
2. Extrahiere drug
3. Entferne Duplikate (Set)
4. Konvertiere wieder zu einer Liste

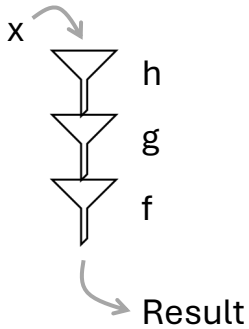
```
drugsInPlan :: MedicationPlan -> [Drug]
drugsInPlan (MedicationPlan _ medications) =
  Set.toList
    . Set.fromList
    . map drug
    $ medications
```

Funktionskomposition: Idiomatisches Java

```
static List<Drug> drugsInPlan(MedicationPlan plan) {
  return plan.medications().stream()
        .map(Medication::drug)
        .distinct()
        .toList();
}
```

```
drugsInPlan :: MedicationPlan -> [Drug]
drugsInPlan (MedicationPlan _ medications) =
  Set.toList
    . Set.fromList
    . map drug
    $ medications
```

Funktionskomposition: Schematisch



f(g(h(x) => h(x) -> g(h(x)) -> f(g(h(x)))

```
drugsInPlan :: MedicationPlan -> [Drug]
drugsInPlan (MedicationPlan _ medications) =
  Set.toList
    . Set.fromList
    . map drug
    $ medications
```

Funktionskomposition in Java

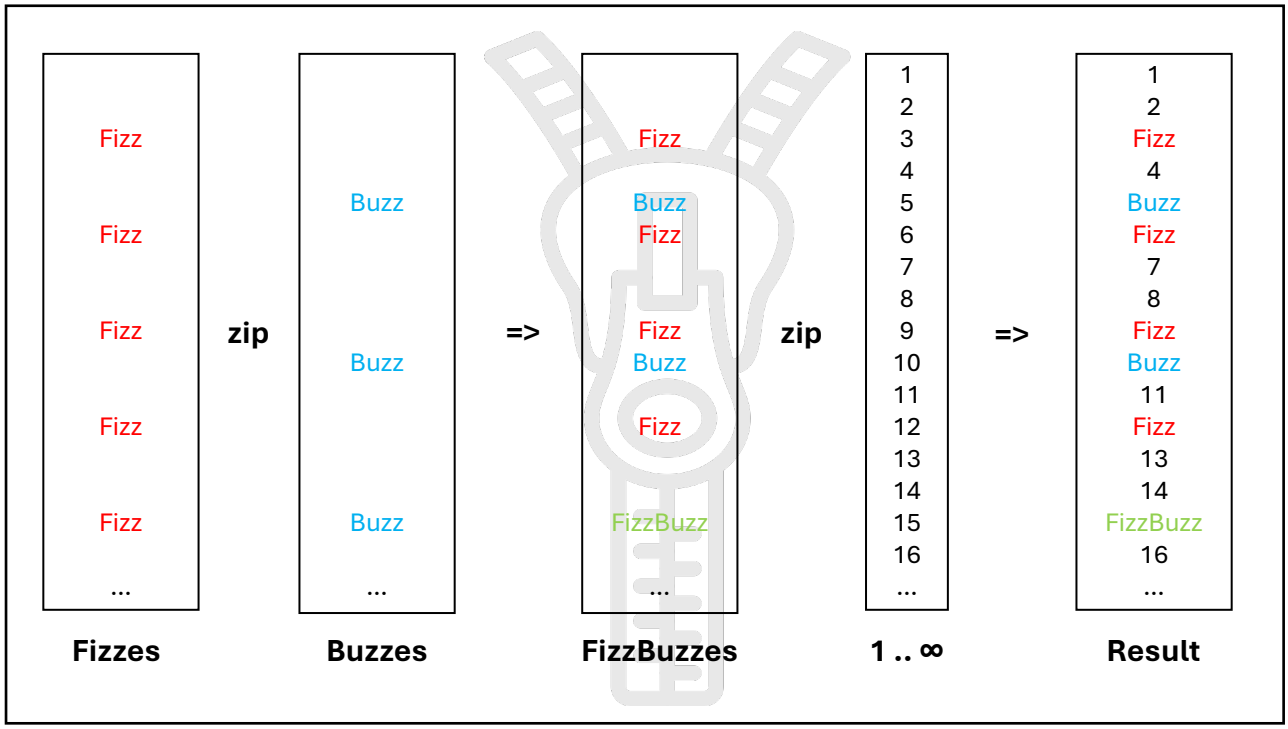
```
static Function1<MedicationPlan, List<Drug>> drugsInPlan =
    extractDrug.andThen(distinct)
        .andThen(toList);

// oder

static Function1<MedicationPlan, List<Drug>> drugsInPlan =
    toList.compose(distinct)
        .compose(extractDrug);
```

42

Higher Order Function

```
fizzBuzz :: [String]
fizzBuzz = zipWith3 display fizzes buzzes [1..]
  where
    fizzes = cycle ["", "", "Fizz"]
    buzzes = cycle ["", "", "", "", "Buzz"]
    display fizz buzz n
      | null (fizz ++ buzz) = show n
      | otherwise           = fizz ++ buzz
```

43



44

Fizz-Buzz in Java

```
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);
```

```
fizzBuzz :: [String]
fizzBuzz = zipWith3 display fizzes buzzes [1..]
  where
    fizzes = cycle ["", "", "Fizz"]
    buzzes = cycle ["", "", "", "", "Buzz"]
    display fizz buzz n
      | null (fizz ++ buzz) = show n
      | otherwise           = fizz ++ buzz
```

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

45

List Comprehension

```
interactionsInPlan :: InteractionTable -> MedicationPlan -> [(Drug, Drug)]
interactionsInPlan table plan =
  [ (d1, d2)
  | (d1, d2) <- drugPairs (drugsInPlan plan),
    interacts table d1 d2
  ]
```

$$\{(d_1, d_2) \mid (d_1, d_2) \in drugPairs(\ldots),\ interacts(d_1, d_2)\}$$

Java: Lösung mit Stream API

```
static List<DrugPair> drugPairs(List<Drug> drugs) {
  return drugs.stream()
      .flatMap(d1 -> drugs.stream()
          .filter(d2 -> d1.compareTo(d2) < 0)
          .map(d2 -> new DrugPair(d1, d2)))
      .toList();
}

static boolean interacts(Map<Drug, Set<Drug>> table, Drug d1, Drug d2) {
  return table.getOrDefault(d1, Set.of()).contains(d2);
}

static List<DrugPair> interactionsInPlan(Map<Drug, Set<Drug>> table, MedicationPlan plan) {
  return drugPairs(drugsInPlan1(plan)).stream()
      .filter(pair -> interacts(table, pair.first(), pair.second()))
      .toList();
}
```

```
interactionsInPlan :: InteractionTable -> MedicationPlan -> [(Drug, Drug)]
interactionsInPlan table plan =
  [ (d1, d2)
  | (d1, d2) <- drugPairs (drugsInPlan plan),
    interacts table d1 d2
  ]
```

Agenda

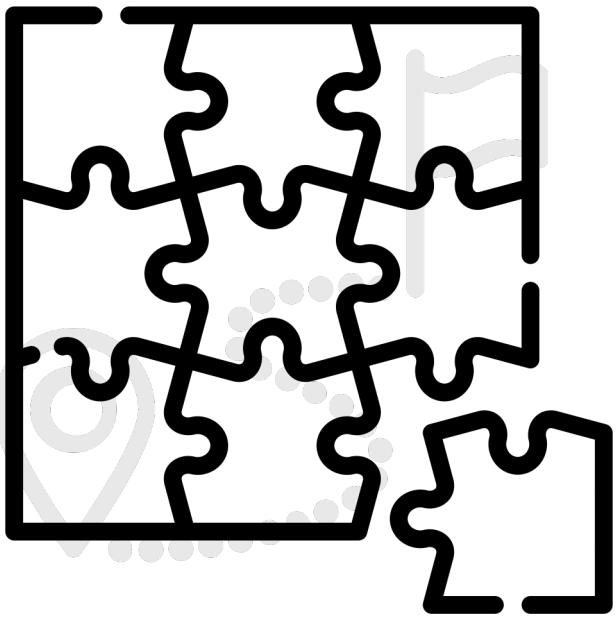
Warum funktionale Programmierung?

Ist Java eigentlich auch funktional?

Kann Java mit Haskell mithalten?

Wohin geht die Reise bei Java?

49

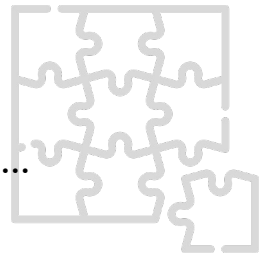


50

Was fehlt Java zu einer funktionalen Sprache?

- Immutability ist nicht erzwungen
- Seiteneffekte nicht vom Typsystem kontrolliert
- Keine garantierte Pure Functions
- keine echte Lazy Evaluation
- ...

- Java entwickelt sich weiter: Null-Restriction, Value Objects, Integrity by Default ("final means final"), ...



51

Funktionale Programmierung in der Praxis

@ctive group

52

FP ist ein Programmier-Paradigma

> *The goal of any programming paradigm is to **manage complexity**.*

https://www.infoq.com/articles/data-oriented-programming-java/

Brian Goetz

53

Paradigma: Datenorientierte Programmierung

> ***Data-oriented programming** encourages us to model data as data. **Records, sealed classes, and pattern matching**, work together to make that easier.*

https://www.infoq.com/articles/data-oriented-programming-java/

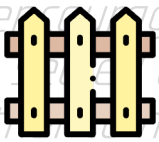
Brian Goetz

54

Grundprinzipien

Model data as data

Data is immutable



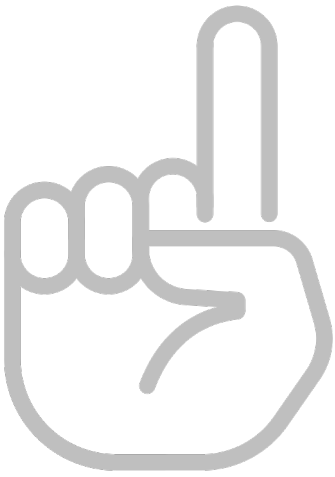
Validate at the boundary

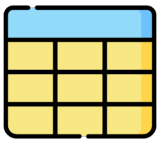
Make illegal states unrepresentable

Records

Algebraic Datatypes

55





Model ~~data~~ **information** as data

information kommt aus der Domäne

data = Repräsentation im Computer

56

Final words

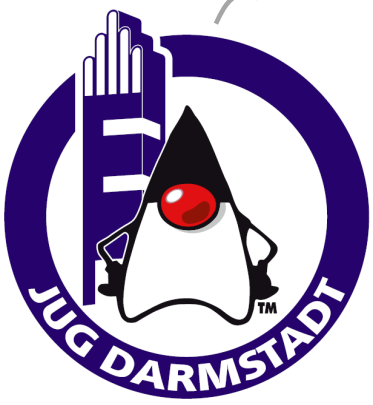
Haskell erzwingt *funktionale Programmierung durch das Typsystem.*

Java ermöglicht *funktionale Programmierung als Programmierstil.*

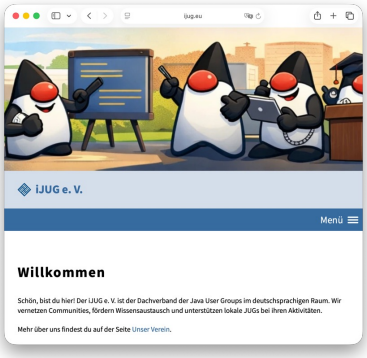
57

Lust auf Austausch in der Java Community

Nächster Termin am 16.07.



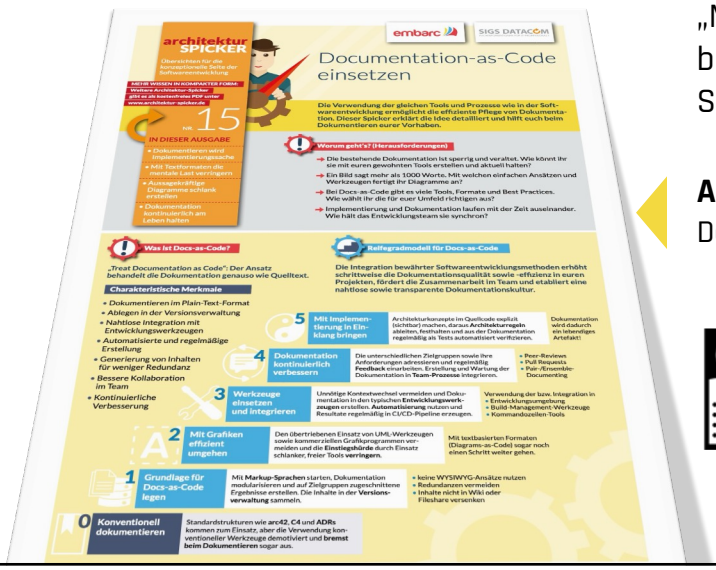
https://www.jug-da.de/



https://www.ijug.eu/

58

Architektur-Spicker



„Mit unseren Architektur-Spickern beleuchten wir die konzeptionelle Seite der Softwareentwicklung."

Architektur-Spicker #15
Documentation-as-Code einsetzen

PDF, 4 Seiten
Kostenloser Download.

➔ architektur-spicker.de/

59

Fragen + Feedback

Code-Beispiele:
https://github.com/sippsack/functional-haskell-vs-java

Falk Sippach (embarc)
falk.sippach@embarc.de
LinkedIn: /in/falk-sippach

Till Rauch (Active Group)
till.rauch@active-group.de

60