# Funktioniert's?

## Funktionale Programmierung für Anfänger

Nicole Rauch

Softwareentwicklung und Entwicklungscoaching

# Funktional, das ist doch nur für Esoteriker?!

# Funktional, das ist doch nur für Esoteriker?!

- ABN AMRO Amsterdam *Risikoanalysen Investmentbanking*
- AT&T *Netzwerksicherheit: Verarbeitung von Internet-Missbrauchsmeldungen*
- Bank of America Merril Lynch
  *Backend: Laden & Transformieren von Daten*
- Barclays Capital Quantitative Analytics Group
  *Mathematische Modellierung von Derivaten*
- Bluespec, Inc. *Modellierung & Verifikation integrierter Schaltkreise*
- Credit Suisse *Prüfen und Bearbeiten von Spreadsheets*
- Deutsche Bank Equity Proprietary Trading, Directional Credit Trading *Gesamte Software-Infrastruktur*
- Facebook *Interne Tools*
- Factis Research, Freiburg *Mobil-Lösungen (Backend)*
- fortytools gmbh *Webbasierte Produktivitätstools - REST-Backend*
- Functor AB, Stockholm *Statische Codeanalyse*

# Funktional, das ist doch nur für Esoteriker?!

- Galois, Inc *Security, Informationssicherheit, Kryptographie*
- Google *Interne Projekte*
- IMVU, Inc. *Social entertainment*
- JanRain *Netzwerk- und Web-Software*
- MITRE *Analyse kryptographischer Protokolle*
- New York Times *Bildverarbeitung für die New York Fashion Week*
- NVIDIA *In-House Tools*
- Parallel Scientific *Hochskalierbares Cluster-Verwaltungssystem*
- Sankel Software *CAD/CAM, Spiele, Computeranimation*
- Silk, Amsterdam *Filtern und Visualisieren großer Datenmengen*
- Skedge Me *Online-Terminvereinbarungen*
- Standard Chartered *Bankensoftware*
- Starling Software, Tokio *Automatisiertes Optionshandelssystem*
- Suite Solutions *Verwaltung technischer Dokumentationen*

(Quelle: `http://www.haskell.org/haskellwiki/Haskell_in_industry`)

# Bekannte funktionale Sprachen

Scheme

Erlang

Clojure

ML

F#

Miranda

OCaml

Haskell

Lisp

Scala

# Bekannte funktionale Sprachen

Scheme

Erlang

Clojure

ML

F#

Miranda

OCaml

Haskell

(Java 8)

Lisp

Scala

(JavaScript)

# Bekannte funktionale Sprachen

Scheme

Erlang

Clojure

ML

F#

Miranda

OCaml

Haskell

(Java 8)

Lisp

Scala

(JavaScript)

# Was ist denn an funktionaler Programmierung so besonders?

# Was ist denn an funktionaler Programmierung so besonders?

Immutability

# Was ist denn an funktionaler Programmierung so besonders?

## Immutability

Jeder Variablen darf nur einmal ein Wert zugewiesen werden

# Was ist denn an funktionaler Programmierung so besonders?

## Immutability

Jeder Variablen darf nur einmal ein Wert zugewiesen werden

## Funktionen sind „first order citizens“

# Was ist denn an funktionaler Programmierung so besonders?

## Immutability

Jeder Variablen darf nur einmal ein Wert zugewiesen werden

## Funktionen sind „first order citizens“

Mit Funktionen kann man dasselbe machen wie mit Strings oder Zahlen

# Funktionen sind Werte

# Funktionen sind Werte

## Java 8: Statische Methoden

```
class Examples { static int staticTimes (int x, int y) { return x * y; } }

IntBinaryOperator timesVar = Examples::staticTimes;

assertThat(timesVar.applyAsInt(3, 5), is(15));
```

# Funktionen sind Werte

## Java 8: Objektmethoden

```
class Examples { int times (int x, int y) { return x * y; } }

Examples examples = new Examples();
IntBinaryOperator timesVar = examples::times;

assertThat(timesVar.applyAsInt(3, 5), is(15));
```

# Funktionen sind Werte

## Java 8: Lambdas

```
IntBinaryOperator times = (x, y) -> x * y;
IntBinaryOperator timesVar = times;

assertThat(timesVar.applyAsInt(3, 5), is(15));
```

# Funktionen sind Werte

Java 8: Lambdas (mit eigenem Funktionsinterface)

```
interface TimesFunction { int eval(int x, int y); }

TimesFunction times = (x, y) -> x * y;
TimesFunction timesVar = times;

assertThat(timesVar.eval(3, 5), is(15));
```

# Funktionen sind Werte

Java 8: Lambdas (mit eigenem Funktionsinterface)

```
interface TimesFunction { int eval(int x, int y); }

TimesFunction times = (x, y) -> x * y;
TimesFunction timesVar = times;

assertThat(timesVar.eval(3, 5), is(15));
```

Haskell:

```
times x y = x * y

timesVar = times

timesVar 3 5 == 15
```

# Funktionen sind Funktionsparameter

# Funktionen sind Funktionsparameter

Java 8:

```
class Examples {
    static int apply(IntUnaryOperator func, int arg) {
        return func.applyAsInt(arg);
    }
}
IntUnaryOperator func = x -> 3 * x;

assertThat(Examples.apply(func, 5), is(15));
```

# Funktionen sind Funktionsparameter

Java 8:

```
class Examples {
    static int apply(IntUnaryOperator func, int arg) {
        return func.applyAsInt(arg);
    }
}
IntUnaryOperator func = x -> 3 * x;

assertThat(Examples.apply(func, 5), is(15));
```

Haskell:

```
apply func arg = func arg

apply (\ x -> 3 * x) 5 == 15
```

# Funktionen sind Rückgabewerte

# Funktionen sind Rückgabewerte

Java 8:

```
interface FunctionFunction { IntUnaryOperator eval(int x); }

FunctionFunction times = x -> { return y -> x * y; };

assertThat(times.eval(3).applyAsInt(5), is(15));
```

# Funktionen sind Rückgabewerte

Java 8:

```
interface FunctionFunction { IntUnaryOperator eval(int x); }

FunctionFunction times = x -> { return y -> x * y; };

assertThat(times.eval(3).applyAsInt(5), is(15));
```

Haskell:

```
times x = (\y -> x * y)

times 3 5 == 15
```

# Komisch, oder?

## Java 8: Zwei verschiedene Aufrufe

```
IntBinaryOperator times = (x, y) -> x * y;
assertThat(times.applyAsInt(3, 5), is(15));

FunctionFunction times = x -> { return y -> x * y; };
assertThat(times.eval(3).applyAsInt(5), is(15));
```

## Haskell: Zweimal derselbe Aufruf

```
times x y = x * y
times 3 5 == 15

times x = (\y -> x * y)
times 3 5 == 15
```

# Currying! (oder auch Schönfinkeln)

# Currying! (oder auch Schönfinkeln)

In echten funktionalen Sprachen schreiben wir:

```
times x y = x * y
```

und eigentlich passiert Folgendes:

```
times x = (\y -> x * y)
```

# Currying! (oder auch Schönfinkeln)

In echten funktionalen Sprachen schreiben wir:

```
times x y = x * y
```

und eigentlich passiert Folgendes:

```
times x = (\y -> x * y)
```

Denn: Funktionen haben immer genau ein Argument

# Currying! (oder auch Schönfinkeln)

In echten funktionalen Sprachen schreiben wir:

```
times x y = x * y
```

und eigentlich passiert Folgendes:

```
times x = (\y -> x * y)
```

Denn: Funktionen haben immer genau ein Argument

Nutzen: Partielle Evaluierung:

```
times x y = x * y

times3 = times 3

times3 5 == 15
```

# Und wenn ich kein Argument haben will?

- In echten funktionalen Sprachen bekommen Funktionen immer genau ein Argument!
- Was ist, wenn ich nichts habe?!

# Und wenn ich kein Argument haben will?

- In echten funktionalen Sprachen bekommen Funktionen immer genau ein Argument!
- Was ist, wenn ich nichts habe?!
- `Unit` to the rescue!
- `Unit` ist ein Typ mit nur einem Element
- Das Element heißt in Haskell `()`

# Wichtige Bibliotheksfunktionen: filter

- filter oder auch select

# Wichtige Bibliotheksfunktionen: filter

- filter oder auch select
- Nimmt eine Collection und eine Funktion
- Liefert diejenigen Elemente der Collection, für die die Funktion `true` ergibt

# Wichtige Bibliotheksfunktionen: filter

- filter oder auch select
- Nimmt eine Collection und eine Funktion
- Liefert diejenigen Elemente der Collection, für die die Funktion `true` ergibt

Java 8:

```
assertThat(Arrays.asList(1, 2, 3, 4).stream().filter(x -> x % 2 == 0).
    toArray(), is(new Integer[]{2, 4}));
```

# Wichtige Bibliotheksfunktionen: filter

- filter oder auch select
- Nimmt eine Collection und eine Funktion
- Liefert diejenigen Elemente der Collection, für die die Funktion `true` ergibt

Java 8:

```
assertThat(Arrays.asList(1, 2, 3, 4).stream().filter(x -> x % 2 == 0).
    toArray(), is(new Integer[]{2, 4}));
```

Haskell:

```
filter (\x -> x `mod` 2 == 0) [1,2,3,4] == [2,4]
```

# Wichtige Bibliotheksfunktionen: map

- map oder auch collect

# Wichtige Bibliotheksfunktionen: map

- map oder auch collect
- Nimmt eine Collection und eine Funktion
- Liefert eine Collection, in der die Funktion auf jedes Element der ursprünglichen Collection angewandt wurde

# Wichtige Bibliotheksfunktionen: map

- map oder auch collect
- Nimmt eine Collection und eine Funktion
- Liefert eine Collection, in der die Funktion auf jedes Element der ursprünglichen Collection angewandt wurde

Java 8:

```
assertThat(Arrays.asList(1, 2, 3, 4).stream().map(x -> x + 5).toArray(),
    is(new Integer[]{6, 7, 8, 9}));
```

# Wichtige Bibliotheksfunktionen: map

- map oder auch collect
- Nimmt eine Collection und eine Funktion
- Liefert eine Collection, in der die Funktion auf jedes Element der ursprünglichen Collection angewandt wurde

Java 8:

```
assertThat(Arrays.asList(1, 2, 3, 4).stream().map(x -> x + 5).toArray(),
    is(new Integer[]{6, 7, 8, 9}));
```

Haskell:

```
map (\x -> x + 5) [1,2,3,4] == [6,7,8,9]
```

# Wichtige Bibliotheksfunktionen: fold

- fold oder auch reduce oder inject

# Wichtige Bibliotheksfunktionen: fold

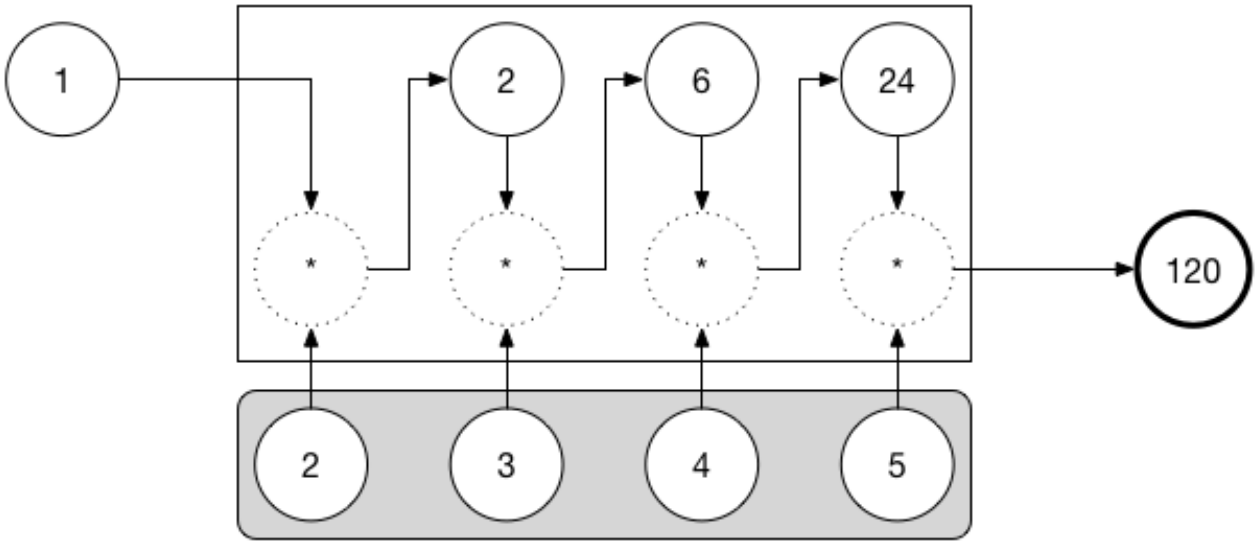
- fold oder auch reduce oder inject
- Nimmt eine Collection, eine Funktion und einen Startwert
- Verbindet Startwert und erstes Element der Collection mit Hilfe der Funktion
- Verbindet das Ergebnis mit dem nächsten Element der Collection
- Setzt dies für alle Elemente der Collection fort, bis nur noch ein Element übrigbleibt

# Wichtige Bibliotheksfunktionen: fold

- fold oder auch reduce oder inject

Java 8:

```
assertThat(Arrays.asList(2, 3, 4, 5).stream().reduce(1, (x, y) -> x*y),
    is(120));
```



# Wichtige Bibliotheksfunktionen: fold

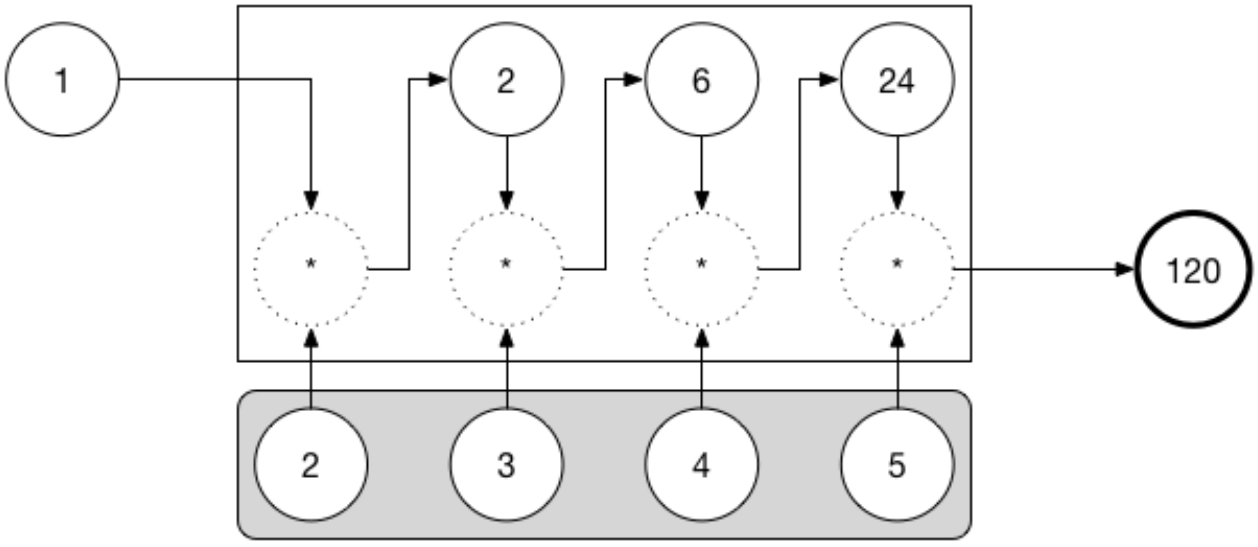
- fold oder auch reduce oder inject

Java 8:

```
assertThat(Arrays.asList(2, 3, 4, 5).stream().reduce(1, (x, y) -> x*y),
    is(120));
```

Haskell:

```
foldl (*) 1 [2,3,4,5] == 120
```



# Typinferenz

- Haskell: starkes statisches Typsystem
- Leichtgewichtige Verwendung dank Typinferenz
- Herleitung des allgemeinst möglichen Typs

# Typinferenz

Was ist der Typ von

```
foldl (*) 1 [2,3,4,5]
```

# Typinferenz

Was ist der Typ von

```
foldl (*) 1 [2,3,4,5]
```

$$\text{foldl} :: (a \to b \to a) \to a \to [b] \to a$$

# Typinferenz

Was ist der Typ von

```
foldl (*) 1 [2,3,4,5]
```

$$\text{foldl} :: \underbrace{(a \to b \to a)}_{\text{Funktion}} \to a \to [b] \to a$$

# Typinferenz

Was ist der Typ von

```
foldl (*) 1 [2,3,4,5]
```

$$\text{foldl} :: \underbrace{(a \to b \to a)}_{\text{Funktion}} \to \underbrace{a}_{\text{Startwert}} \to [b] \to a$$

# Typinferenz

Was ist der Typ von

```
foldl (*) 1 [2,3,4,5]
```

$$\text{foldl} :: \underbrace{(a \to b \to a)}_{\text{Funktion}} \to \underbrace{a}_{\text{Startwert}} \to \underbrace{[b]}_{\text{Liste}} \to a$$

# Typinferenz

Was ist der Typ von

```
foldl (*) 1 [2,3,4,5]
```

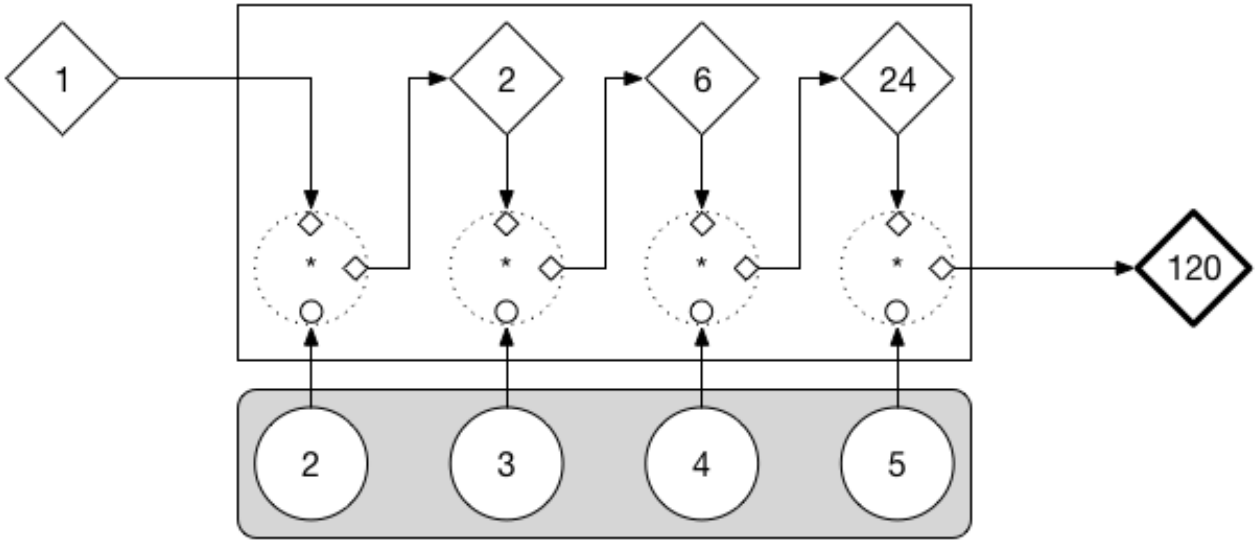
$$\text{foldl} :: \underbrace{(a \to b \to a)}_{\text{Funktion}} \to \underbrace{a}_{\text{Startwert}} \to \underbrace{[b]}_{\text{Liste}} \to \underbrace{a}_{\text{Rückgabewert}}$$

# Typinferenz

Was ist der Typ von

```
foldl (*) 1 [2,3,4,5]
```

$$\text{foldl} :: \underbrace{(a \to b \to a)}_{\text{Funktion}} \to \underbrace{a}_{\text{Startwert}} \to \underbrace{[b]}_{\text{Liste}} \to \underbrace{a}_{\text{Rückgabewert}}$$



# Typinferenz

Was ist der Typ von

```
foldl (*) 1 [2,3,4,5]
```

$$\text{foldl} :: \underbrace{(a \to b \to a)}_{\text{Funktion}} \to \underbrace{a}_{\text{Startwert}} \to \underbrace{[b]}_{\text{Liste}} \to \underbrace{a}_{\text{Rückgabewert}}$$

$$(*) :: \text{Num } a \Rightarrow a \to a \to a$$

# Typinferenz

Was ist der Typ von

```
foldl (*) 1 [2,3,4,5]
```

$$\text{foldl} :: \underbrace{(a \to b \to a)}_{\text{Funktion}} \to \underbrace{a}_{\text{Startwert}} \to \underbrace{[b]}_{\text{Liste}} \to \underbrace{a}_{\text{Rückgabewert}}$$

$$(*) :: \underbrace{\text{Num } a}_{\text{Bedingung}} \Rightarrow \underbrace{a}_{\text{1. Arg}} \to \underbrace{a}_{\text{2. Arg}} \to \underbrace{a}_{\text{Rückgabe}}$$

# Typinferenz

Was ist der Typ von

```
foldl (*) 1 [2,3,4,5]
```

$$\text{foldl} :: \underbrace{(a \to b \to a)}_{\text{Funktion}} \to \underbrace{a}_{\text{Startwert}} \to \underbrace{[b]}_{\text{Liste}} \to \underbrace{a}_{\text{Rückgabewert}}$$

$$(*) :: \underbrace{\text{Num } a}_{\text{Bedingung}} \Rightarrow \underbrace{a}_{\text{1. Arg}} \to \underbrace{a}_{\text{2. Arg}} \to \underbrace{a}_{\text{Rückgabe}}$$

$$\text{foldl} :: \text{Num } a \Rightarrow (a \to a \to a) \to a \to [a] \to a$$

# Typinferenz

Compilerfehler für:

```
foldl (*) "x" [2,3,4,5]

No instance for (Num [Char]) arising from a use of ‘*’
Possible fix: add an instance declaration for (Num [Char])
```

# Eine einfache Berechnung

$$sum = \sum_{i=1}^{10} i^2$$

# Eine einfache Berechnung

$$sum = \sum_{i=1}^{10} i^2$$

```
int sum = 0;
for(int i = 1; i <= 10; i++) {
  sum = sum + i * i;
}
```

# Exkurs: Clean Code

Single Responsibility Principle

# Exkurs: Clean Code

Single Responsibility Principle

Wie viele Verantwortlichkeiten hat dieser Code?

```
int sum = 0;
for(int i = 1; i <= 10; i++) {
  sum = sum + i * i;
}
```

# Exkurs: Clean Code

Single Responsibility Principle

Wie viele Verantwortlichkeiten hat dieser Code?

```
int sum = 0;
for(int i = 1; i <= 10; i++) {
  sum = sum + i * i;
}
```

- Erzeugen der Zahlenfolge von 1 bis 10

# Exkurs: Clean Code

Single Responsibility Principle

Wie viele Verantwortlichkeiten hat dieser Code?

```
int sum = 0;
for(int i = 1; i <= 10; i++) {
  sum = sum + i * i;
}
```

- Erzeugen der Zahlenfolge von 1 bis 10
- Quadrieren einer Zahl

# Exkurs: Clean Code

Single Responsibility Principle

Wie viele Verantwortlichkeiten hat dieser Code?

```
int sum = 0;
for(int i = 1; i <= 10; i++) {
  sum = sum + i * i;
}
```

- Erzeugen der Zahlenfolge von 1 bis 10
- Quadrieren einer Zahl
- Berechnen der Quadratzahl jeder Zahl in der Folge

# Exkurs: Clean Code

Single Responsibility Principle

Wie viele Verantwortlichkeiten hat dieser Code?

```
int sum = 0;
for(int i = 1; i <= 10; i++) {
  sum = sum + i * i;
}
```

- Erzeugen der Zahlenfolge von 1 bis 10
- Quadrieren einer Zahl
- Berechnen der Quadratzahl jeder Zahl in der Folge
- Addieren zweier Zahlen

# Exkurs: Clean Code

Single Responsibility Principle

Wie viele Verantwortlichkeiten hat dieser Code?

```
int sum = 0;
for(int i = 1; i <= 10; i++) {
  sum = sum + i * i;
}
```

- Erzeugen der Zahlenfolge von 1 bis 10
- Quadrieren einer Zahl
- Berechnen der Quadratzahl jeder Zahl in der Folge
- Addieren zweier Zahlen
- Aufsummieren der Quadratzahlen

# Trennen der Verantwortlichkeiten

- Erzeugen der Zahlenfolge von 1 bis 10
- Quadrieren einer Zahl
- Berechnen der Quadratzahl jeder Zahl in der Folge
- Addieren zweier Zahlen
- Aufsummieren der Quadratzahlen

# Trennen der Verantwortlichkeiten

- Erzeugen der Zahlenfolge von 1 bis 10

```
IntStream sequence = IntStream.range(1, 11);
```

- Quadrieren einer Zahl

- Berechnen der Quadratzahl jeder Zahl in der Folge

- Addieren zweier Zahlen

- Aufsummieren der Quadratzahlen

# Trennen der Verantwortlichkeiten

- Erzeugen der Zahlenfolge von 1 bis 10

```
IntStream sequence = IntStream.range(1, 11);
```

- Quadrieren einer Zahl

```
IntUnaryOperator square = x -> x*x;
```

- Berechnen der Quadratzahl jeder Zahl in der Folge

- Addieren zweier Zahlen

- Aufsummieren der Quadratzahlen

# Trennen der Verantwortlichkeiten

- Erzeugen der Zahlenfolge von 1 bis 10

```
IntStream sequence = IntStream.range(1, 11);
```

- Quadrieren einer Zahl

```
IntUnaryOperator square = x -> x*x;
```

- Berechnen der Quadratzahl jeder Zahl in der Folge

```
IntStream squaredSequence = sequence.map(square);
```

- Addieren zweier Zahlen

- Aufsummieren der Quadratzahlen

# Trennen der Verantwortlichkeiten

- Erzeugen der Zahlenfolge von 1 bis 10

```
IntStream sequence = IntStream.range(1, 11);
```

- Quadrieren einer Zahl

```
IntUnaryOperator square = x -> x*x;
```

- Berechnen der Quadratzahl jeder Zahl in der Folge

```
IntStream squaredSequence = sequence.map(square);
```

- Addieren zweier Zahlen

```
IntBinaryOperator add = (x,y) -> x+y;
```

- Aufsummieren der Quadratzahlen

# Trennen der Verantwortlichkeiten

- Erzeugen der Zahlenfolge von 1 bis 10

```
IntStream sequence = IntStream.range(1, 11);
```

- Quadrieren einer Zahl

```
IntUnaryOperator square = x -> x*x;
```

- Berechnen der Quadratzahl jeder Zahl in der Folge

```
IntStream squaredSequence = sequence.map(square);
```

- Addieren zweier Zahlen

```
IntBinaryOperator add = (x,y) -> x+y;
```

- Aufsummieren der Quadratzahlen

```
Integer sum = squaredSequence.reduce(0, add);
```

# Zusammensetzen der Komponenten

Java 8:

```
IntUnaryOperator square = x -> x*x;
IntBinaryOperator add = (x,y) -> x+y;
```

```
assertThat(IntStream.range(1, 11).map(square).reduce(0, add), is(385));
```

# Zusammensetzen der Komponenten

Java 8:

```
IntUnaryOperator square = x -> x*x;
IntBinaryOperator add = (x,y) -> x+y;
```

```
assertThat(IntStream.range(1, 11).map(square).reduce(0, add), is(385));
```

Haskell:

```
foldl (+) 0 (map (\x -> x*x) [1..10]) == 385
```

# Zusammensetzen der Komponenten

Java 8:

```
IntUnaryOperator square = x -> x*x;
IntBinaryOperator add = (x,y) -> x+y;
```

```
assertThat(IntStream.range(1, 11).map(square).reduce(0, add), is(385));
```

Haskell:

```
foldl (+) 0 (map (\x -> x*x) [1..10]) == 385
```

oder

```
(>.>) x f = f x
[1..10] >.> map (\x -> x*x) >.> foldl (+) 0 == 385
```

# Uff!

OK, alle einmal tief durchatmen :-)

# Pattern Matching

Fibonacci-Funktion „naiv“:

```
fib x = if x < 2 then x else fib (x-1) + fib (x-2)
```

# Pattern Matching

Fibonacci-Funktion „naiv“:

```
fib x = if x < 2 then x else fib (x-1) + fib (x-2)
```

Fibonacci-Funktion mit Pattern Matching:

```
fib 0 = 0
fib 1 = 1
fib x = fib (x-1) + fib (x-2)
```

# Algebraische Datentypen

Binärbaum:

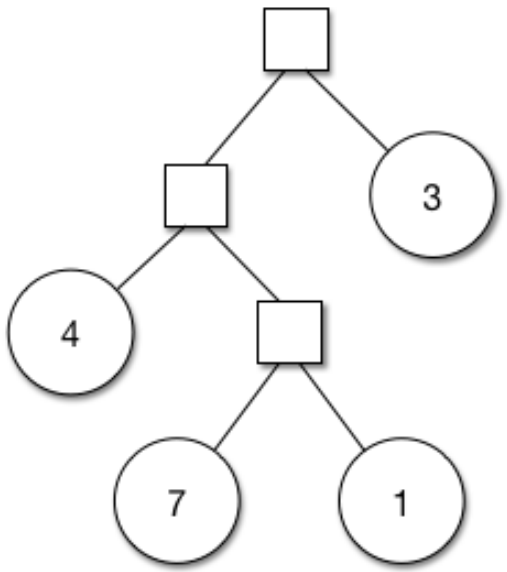
```
data Tree =
       Node Tree Tree
     | Leaf Int
```

# Algebraische Datentypen

Binärbaum:

```
data Tree =
       Node Tree Tree
     | Leaf Int
```

```
myTree = Node (Node (Leaf 4) (Node (Leaf 7) (Leaf 1))) (Leaf 3)
```



# Algebraische Datentypen

Binärbaum:

```
data Tree =
       Node Tree Tree
     | Leaf Int
```

```
myTree = Node (Node (Leaf 4) (Node (Leaf 7) (Leaf 1))) (Leaf 3)
```

Summenfunktion:

# Algebraische Datentypen

Binärbaum:

```
data Tree =
       Node Tree Tree
     | Leaf Int
```

```
myTree = Node (Node (Leaf 4) (Node (Leaf 7) (Leaf 1))) (Leaf 3)
```

Summenfunktion:

```
treeSum (Leaf x) = x
```

# Algebraische Datentypen

Binärbaum:

```
data Tree =
       Node Tree Tree
     | Leaf Int
```

```
myTree = Node (Node (Leaf 4) (Node (Leaf 7) (Leaf 1))) (Leaf 3)
```

Summenfunktion:

```
treeSum (Leaf x) = x

treeSum (Node m n) = treeSum m + treeSum n
```

# Algebraische Datentypen

Binärbaum:

```
data Tree =
       Node Tree Tree
     | Leaf Int
```

```
myTree = Node (Node (Leaf 4) (Node (Leaf 7) (Leaf 1))) (Leaf 3)
```

Summenfunktion:

```
treeSum (Leaf x) = x

treeSum (Node m n) = treeSum m + treeSum n
```

```
treeSum myTree == 15
```

# Fazit

- Funktionale Programmierung ist verbreiteter als man denkt
- Manches lässt sich in den nicht-funktionalen Alltag integrieren
- Viele Sprachen bringen funktionale Aspekte oder Zusatzmodule mit

# Fazit

- Funktionale Programmierung ist verbreiteter als man denkt
- Manches lässt sich in den nicht-funktionalen Alltag integrieren
- Viele Sprachen bringen funktionale Aspekte oder Zusatzmodule mit

Referenz:

- Haskell: `http://www.haskell.org`

# Vielen Dank!

Code & Folien auf GitHub:

`https://github.com`
`/NicoleRauch/FunctionalProgrammingForBeginners`


## Nicole Rauch

E-Mail `info@nicole-rauch.de`

Twitter `@NicoleRauch`

Web `http://www.nicole-rauch.de`