MATHEMA

# Überraschend funktional


## Michael Wiedeking

MATHEMA Software GmbH
Henkestraße 91
91052 Erlangen
info@mathema.de
www.mathema.de

# Was Sie erwartet

- Ganz wenig über mich
- Einen kleinen Überblick über das, was das Funktionale Paradigma ausmacht
- Eine Idee davon, welche Vorteile das Funktionale Paradigma haben kann

# Was Sie nicht erwartet

- Ein Einblick in die verfügbaren Funktionalen Sprachen für JVM und CLI/CLR
- In Folge dessen: Die Gegenüberstellung der einzelnen Sprachen

# Ganz wenig über mich

- Java-Entwickler der ersten Stunde
- Gründer und Geschäftsführer der MATHEMA Software GmbH

  http://www.mathema.de
- Herausgeber des KaffeeKlatsch

  http://www.bookware.de/kaffeeklatsch
- Veranstalter des Herbstcampus

  http://www.herbstcampus.de
- Sammler von Programmiersprachen

# Motivation

# Das ursächliche Problem…

Was bitte liefert denn

$p$.doIt($o$.f(), $o$.g())

?

# Das ursächliche Problem…

```
p.doIt(o.f(), o.g())

int f() {
    this.x = this.x + 1;
    return this.x;
}
int g() {
    this.x = this.x * 2;
    return this.x;
}
```

# Das ursächliche Problem…

```
this.x = 3;
a = f();        // this.x = this.x + 1;    // x == 4
b = g();        // this.x = this.x * 2;    // x == 8
doIt(a, b);     // doIt(4, 8)
```

```
this.x = 3;
b = g();        // this.x = this.x * 2;    // x == 6
a = f();        // this.x = this.x + 1;    // x == 7
doIt(a, b);     // doIt(6, 7)
```

# Das ursächliche Problem

$$i = i{+}{+} + {+}{+}i;$$

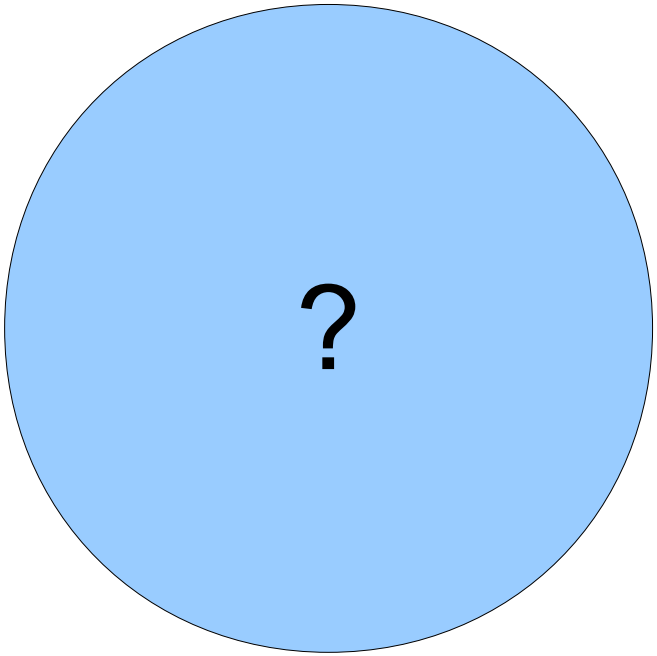
# Aber das macht doch niemand!

# Doch!

# Aber, warum nur machen die sowas?

# Weil‘s geht!

# Das Problem (mit der Objektorientierung)

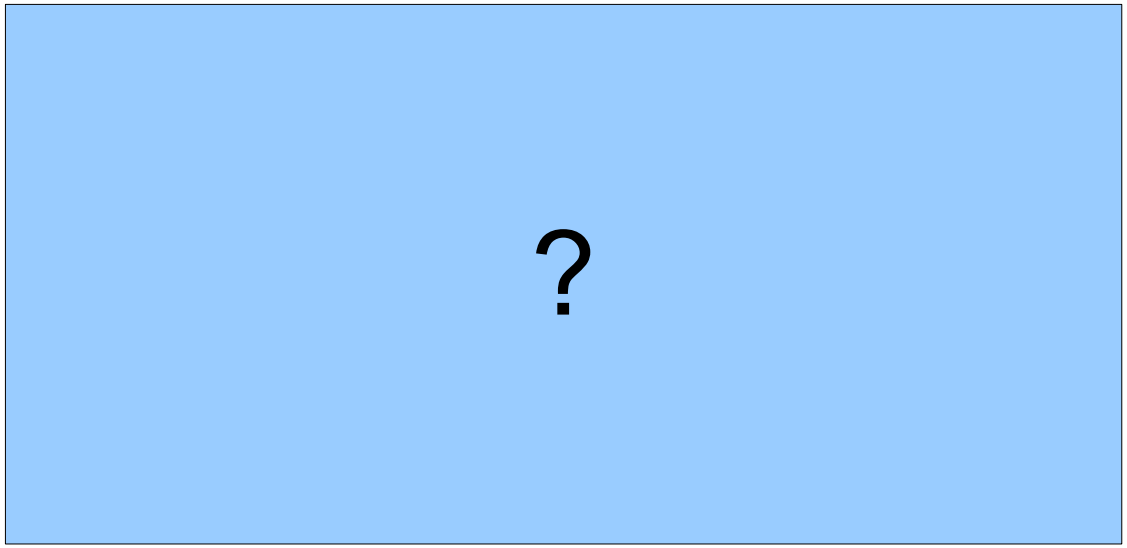


# Typische „Vorteile“ der Objektorientierung

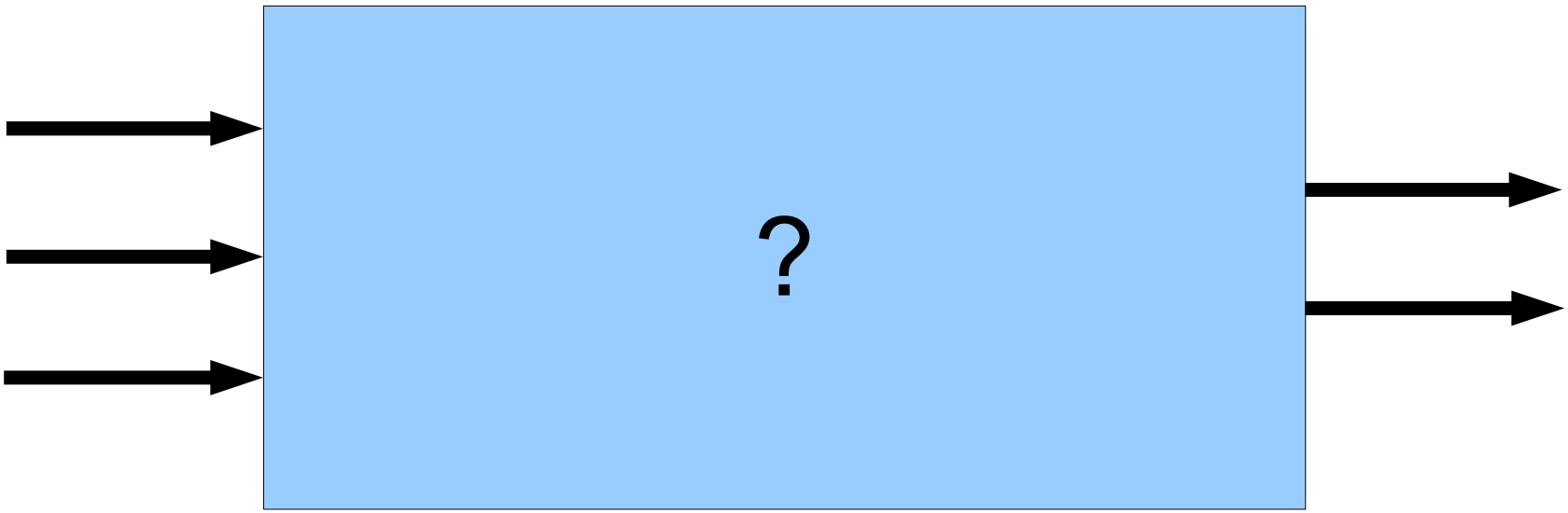
- Gemeinsame Organisation von Daten und Funktionen
- Information-Hiding
- Schnittstellen
- Vererbung
- Orts-Transparenz
- Kommunikation über Nachrichten (Methoden)
- Zustandskapselung

# Das Ideal

# „Was ist eine Dampfmaschine?“



# „Was ist eine Dampfmaschine?“



# Typische „Vorteile“ der ~~Objektorientierung~~ Funktionalen Programmierung

- Gemeinsame Organisation von Daten und Funktionen
- Information-Hiding
- Schnittstellen
- Vererbung
- Orts-Transparenz
- Kommunikation über Nachrichten (Funktionen)
- Zustandskapselung

# *Strikt* und *Faul*

# Die Rolle der Auswertereihenfolge

$$f(x + 1, 2 \cdot (x + 1 \cdot (x - 2)) + 3)$$

# Überflüssiges Ausführen von Anweisungen

```
public static void log(String s) {
    if (debug_enabled) {
        System.logger.log(s);
    }
}

log("Beware: " + x + " ≥ " + valid_max);
```

# „Faules“ Auswerten

```
if (p ≠ null) {
    if (p.isValid()) {
        …
    }
}
```

```
if (p ≠ null && p.isValid()) {
    …
}
```

```
function . && . : (p : 𝔹, ⟦lazy⟧ q : 𝔹) → 𝔹 is
    if p then
        return q
    end
end
```

```
public procedure log(⟦lazy⟧ s : String) {
    if debug_enabled then
        System.logger.log(s);
    end
end
```

# „Faule“ Konsequnzen

$x \leftarrow n^{th}PrimeNumber(10\ 000)$

$y \leftarrow x + 1$

$x \leftarrow n^{th}PrimeNumber(9999)$

$y \leftarrow n^{th}PrimeNumber(10\ 000)$

$y \leftarrow x + y$

# „Faule“ Konsequnzen

```
x ← nthPrimeNumber(10 000)   // schnell
y ← x + 1                    // langsam
x ← nthPrimeNumber(9999)     // schnell
y ← nthPrimeNumber(10 000)   // schnell(!)
y ← x + y                    // langsam
```

Ein Mops kam in die Küche
und stahl dem Koch ein Ei.

Da nahm der Koch das Messer
und schlug den Mops entzwei.

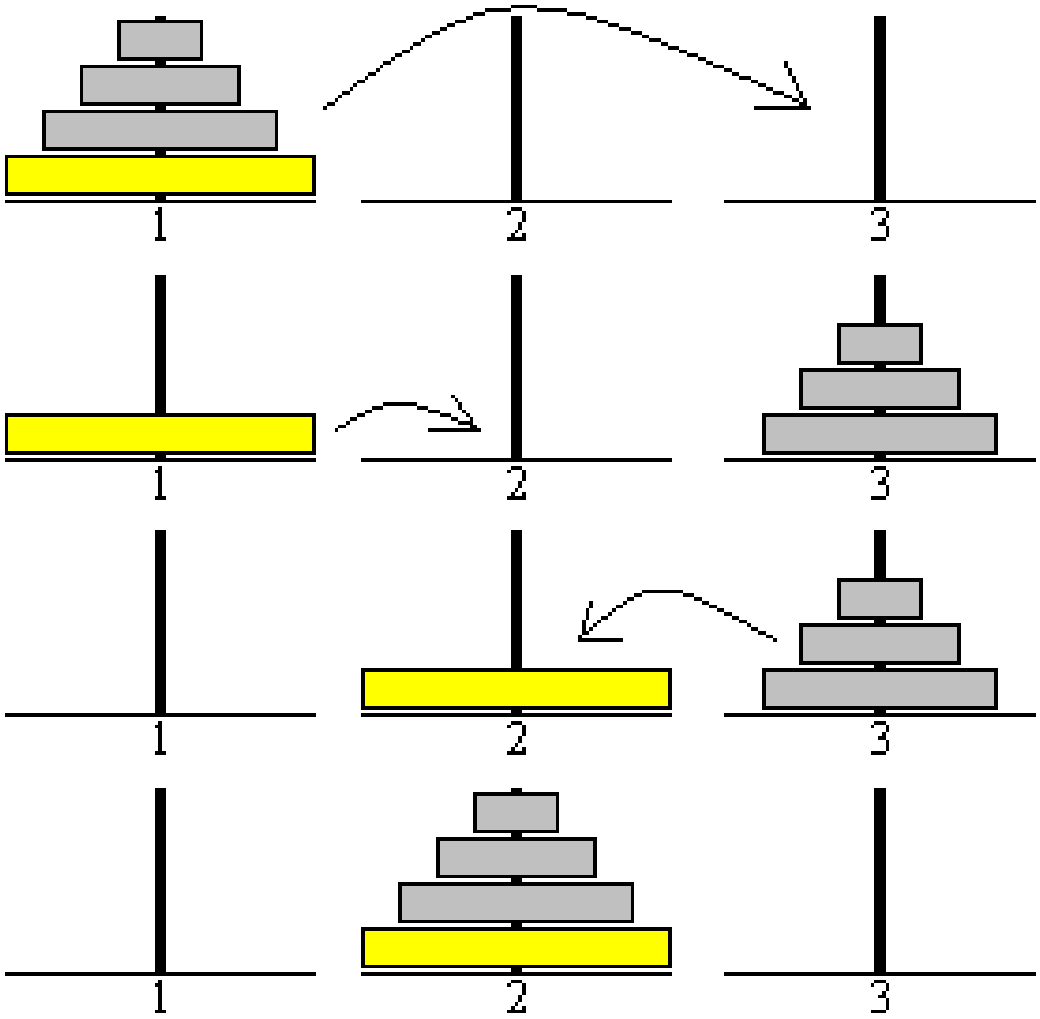
Da kamen viele Möpse
und gruben ihm ein Grab

Drauf setzten sie ’nen Grabstein,
auf dem geschrieben stand:

Ein Mops kam in die Küche
und stahl dem Koch ein Ei.

…

# Turm von Hanoi



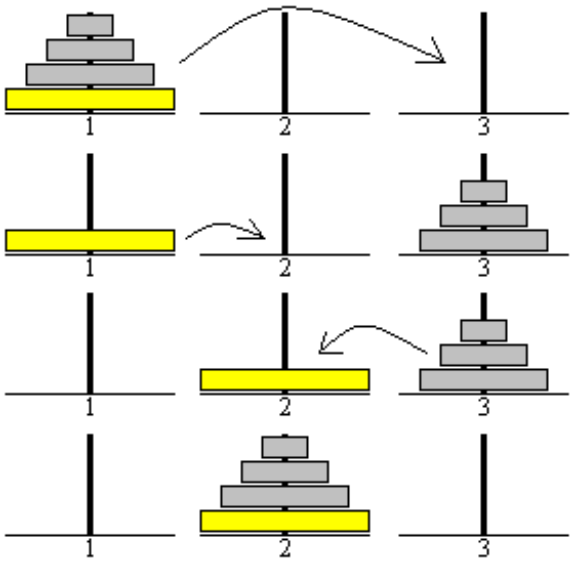

Quelle: http://www.mathematische-basteleien.de/hanoi.htm

**procedure** turmVonHanoi($n$, $v$, $n$, $ü$) **is** // 4 S. von 1 nach 2 via 3

- $n = 1$:
  bewegeScheibe($v$, $n$)

- $n > 1$:
  turmVonHanoi($n - 1$, $v$, $ü$, $n$) // 3 Scheiben von 1 nach 3 via 2
  turmVonHanoi(1, $v$, $n$, –) // 1 Scheiben von 1 nach 2 via –
  turmVonHanoi($n - 1$, $ü$, $n$, $v$) // 3 Scheiben von 3 nach 2 via 1

**end procedure**




Quelle: http://www.mathematische-basteleien.de/hanoi.htm

# „Der“ Klassiker schlechthin

$$n! = 1 \cdot 2 \cdot 3 \cdots (n-2) \cdot (n-1) \cdot n$$

**function** $.! : (n : \mathbb{N}) \to \mathbb{N}$ **is**

▢ $n = 0$:

    **return** 1

▢ $n > 0$:

    **return** $n \cdot (n-1)!$

**end function**

$$n! = 1 \cdot 2 \cdot 3 \cdots (n-2) \cdot (n-1) \cdot n$$

**function** .! : $(n : \mathbb{N}) \to \mathbb{N}$ **is**

**return** fac($n$, 1) **where**

**function** fac($n : \mathbb{N}, k : \mathbb{N}$) $\to \mathbb{N}$ **is**

▢ $n = 0$:

**return** 1

▢ $n > 0$:

**return** fac($n - 1, n \cdot k$)

**end function**

**end function**

$n! = 1 \cdot 2 \cdot 3 \cdots (n-2) \cdot (n-1) \cdot n$

```
function .! : (n : ℕ) → ℕ is

    return fac(n, 1) where

        function fac(n : ℕ, k : ℕ) → ℕ is
        ▢ n = 0:
            return 1
        ▢ n > 0:
            return fac(n – 1, n · k)
        end function

end function
```

```
//   k :← 1

// LOOP:
//   if n = 0 then
//       return k
//   elseif n > 0 then
//       n ← n – 1
//       k ← n · k
//   end if
//   goto LOOP
```

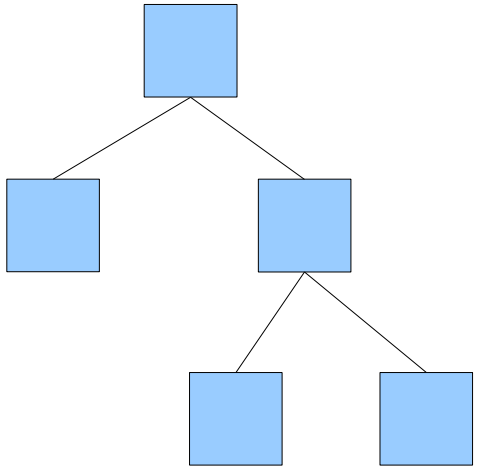
# *Innen drin* und *Außen rum*

**interface** Iterator⟦T⟧ **is**

**method** hasNext() → 𝔹

**method** next() → T

**end interface**

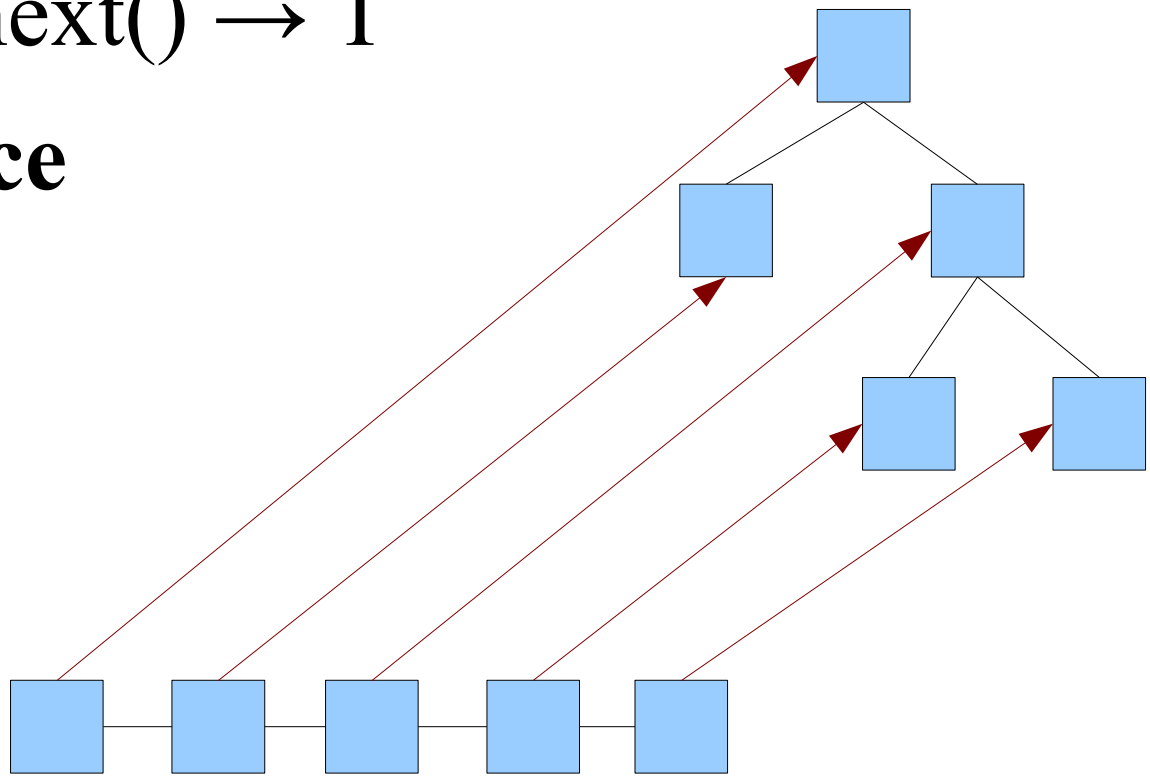


**interface** Iterator⟦T⟧ **is**

**method** hasNext() → 𝔹

**method** next() → T

**end interface**



**forall** *x* **in** *collection* **do**

print(*x*)

**end forall**

**class** Collection⟦T⟧ **is**

    **method** forall( (T) → () )

**end**

*collection*.forall(
    (*x*) ↦ print(*x*)
)

# Transformation über eine äußere Schleife

```
collection :← new Collection⟦Int⟧

…

other1 :← new Collection⟦Int⟧

other2 :← new Collection⟦String⟧

forall x in collection do

    y :← 2 · x

    other1.append(y)
    other2.append(if even(y) then ”even“ else ”odd“ end)

end forall
```

# Transformation über eine innere Schleife

**class** Collection **is**

**method** map⟦S⟧( $f$ : ((T) → (S)) ) → Collection⟦T⟧

**end**

$other1$ :← $collection$.map($(x) \mapsto 2 \cdot x$)

$other2$ :← $collection$.map(
$(x)$ ↦ **if** even($x$) **then** ”even“ **else** ”odd“ **end**
)

# Curry
# (ein Gewürz, das eine besondere Schärfe mit sich bringt)

# Eine ganz normale Funktion

**function** add($x : \mathbb{N}, y : \mathbb{N}$) $\rightarrow \mathbb{N}$ is

    **return** $x + y$

**end function**

# Eine normale Funktion!

$$plus2 :\leftarrow \text{add}(2)$$

# Eine normale Funktion!

$$plus2 :\leftarrow \text{add}(2)$$

$$plus2(3) \quad // \mapsto 5$$

**function** add($x : \mathbb{N}$) $\longrightarrow$ ($y : \mathbb{N}$) $\longrightarrow \mathbb{N}$ is

    **return** $x + y$

**end function**

**function** add($x : \mathbb{N}$) $\longrightarrow$ (($y : \mathbb{N}$) $\longrightarrow \mathbb{N}$) **is** …

```
class Adder {

    private final int x;

    Adder(int x) {
        this.x = x;
    }

    int apply(int y) {
        return x + y;
    }

}
```

```
Adder plus2 = new Adder(2);          // plus2 :← add(2)
print(plus2.apply(3));               // print(plus2(3))
```

# Ehrenbürger

$$f \circ g$$

# Objektorientierte vs Funktionale Programmierung

```
Customer doIt(String name, int salary) {
    c = new Customer(name);
    c.paySalary(salary);
    c.doThis();
    c.doThat();

    return c;
}
```

```
Customer doIt(String name, int salary){
    c = new Customer(name);
    c = paySalary(c, salary);
    c = doThis(c);
    c = doThat(c);

    return c;
}
```

# Erzeugen neuer Funktionen

```
Customer doIt(String name, int salary) {
    c = new Customer(name);
    c = paySalary(c, salary);
    c = doThis(c);
    c = doThat(c);

    return c;
}
```

```
function doIt(name : String, salary : Int) → Customer is
    thingsToDo :← paySalary(., salary) ∘ doThis(.) ∘ doThat(.)
    c ← new Customer(name)
    c ← thingsToDo(c)

    return c
end function
```

```
function doIt(name : String, salary : Int) → Customer is

    thingsToDo :← paySalary(., salary) ∘ doThis(.) ∘ doThat(.)
    c ← new Customer(name)
    c ← thingsToDo(c)

    return c

end function


alias Customer C

function doIt(name : String,  thingsToDo : (C) → C) → C is

    c ← new C(name)
    return thingsToDo(c)

end function
```

**function** doSomethingImportant(*f* : (T) → T, …) →T **is**

    **if** *logging_enabled* **then**
        *f* ← logBefore ∘ *f* ∘ logAfter
    **end if**

    **return** *f*(…)

**end function**

**function** enableLogging(*f* : (T) → T) → ((T) → T) **is**

    **if** *logging_enabled* **then**
        *f* ← logBefore ∘ *f* ∘ logAfter
    **end if**

    **return** *f*

**end function**

# Fazit

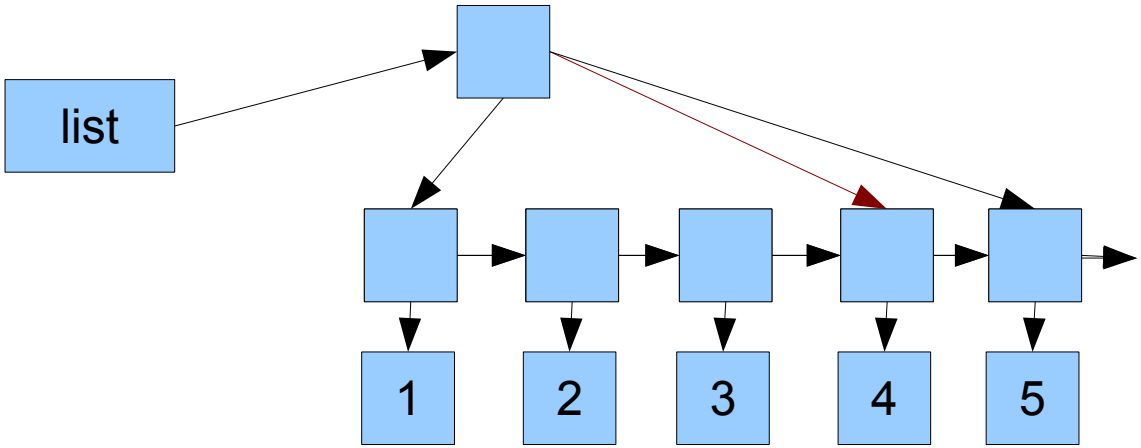
# Collection⟦T⟧

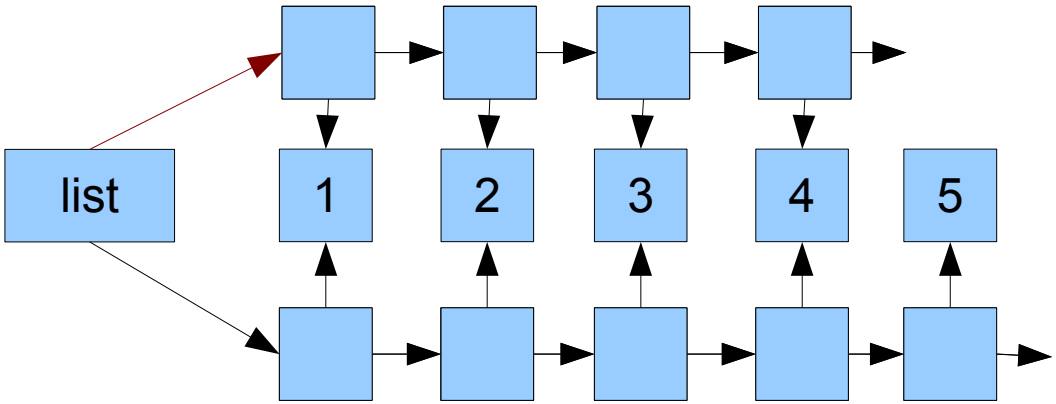
# Framework[[$\mathcal{F}$]]

# Mutable vs Immutable

List *list*;

*list*.add(*x*);




*list* : List

*list* ← *list*.add(x)

# Was funktionale Programmierung **nicht** ist

- **Völlig** frei von Nebeneffekten
- Einfacher
- Schwieriger
- Komplizierter
- Prägnanter
- Effizienter
- Langsamer
- …

# Was funktionale Programmierung ist

- **Möglichst** frei von Nebeneffekten
- Einfacher
- Schwieriger
- Komplizierter
- Prägnanter
- Effizienter
- Langsamer
- …

# Einfach anders!

# Einfach ein anderes Paradigma!

# Ein anderes Vorgehen erfordert ein anderes Denken

# Jede(!) Umgebung
# hat ihre eigenen Probleme

# Beispiel: Groß- und Kleinschreibung

ich habe in moskau liebe genossen

# Haben Sie noch irgendwelche Fragen?

# Vielen Dank!

Michael Wiedeking
michael.wiedeking@mathema.de
http://www.mathema.de