FUNKTIONALE PROGRAMMIERUNG IN SWIFT

AGATA JANKOWSKI

PARADIGMA
STATT RELIGION

⛪

```swift
var x: Int = 1
let y: Int = 1

x = 2  // 👍
y = 2  // 👎
```

UNVERÄNDERBARE WERTE

```
let firstname = “Max“
let lastname = “Mustermann“
let name = firstname + “ “ + lastname

// statt var name = firstname + “ Mustermann“
```

```swift
let numbers = Array(1...10)
var total = 0

func addNumbers() {
    for number in numbers {
        total += number
    }
}
```

```
addNumbers()
total // 55

addNumbers()
total // 110

addNumbers()
total // 165
```

🕙

PURE FUNKTIONEN

KEINE NEBENWIRKUNGEN

Gleiche Eingabe = Gleiche Ausgabe

Die Definition von ***Wahnsinn*** *ist,*
immer wieder ***das Gleiche*** *zu tun*
und ***andere Ergebnisse*** *zu erwarten.*

Albert Einstein 💡

```swift
let numbers = Array(1...10)
var total = addNumbers(numbers)

func addNumbers(numbers: [Int]) -> Int {
    numbers.reduce(0,+)
}
```

```
total = addNumbers(myNumbers) // 55

total = addNumbers(myNumbers) // 55

total = addNumbers(myNumbers) // 55
```

🕙

Objektorientient im Großen, funktional im Kleinen.

Value Type

```
struct S { var number: Int = 1 }
var a = S()
var b = a
a.number = 42

// a.number = 42
// b.number = 1
```

Reference Type

```
class C { var number: Int = 1 }
var a = C()
var b = a
a.number = 42

// a.number = 42
// b.number = 42
```

My [...] remark is that our intellectual powers
are rather geared to **master static relations**
and that our powers to visualize processes **evolving**
in time are relatively poorly developed.

Go To Statement Considered Harmful. **Edsger W. Dijkstra**. Communications of the ACM Volume 11 Issue 3, March 1968 Pages 147–148

FIRST CLASS ✈️ Funktionen

FUNKTIONEN ALS WERTE

```swift
func addOne(number: Int) -> Int {
      return number++ }

let six = addOne(5) // 6
```

👉

```swift
let addOne = { $0 + 1 }
addOne(5) // 6
```

CAPTURE THE MOMENT! 📷

MIT CLOSURES

```swift
let addOne = { $0 + 1 }
addOne(5) // 6
```

```
{(params)-> ReturnType in
    statements }
```

```swift
let even = { (number: Int) -> Bool in
      return number % 2 == 0}
```

```swift
even(3) // false
```

CLOSURE SHORTHAND SYNTAX

```
    let numbers = Array(1...3)


    numbers.map({ (i:Int) -> Int in return i * 2})
👉  numbers.map({ i in return i * 2})
👉  numbers.map({ i in i * 2 })
👉  numbers.map({ $0 * 2 })
👉  numbers.map { $0 * 2 }
```

MAP FILTER REDUCE

```swift
let numbers = Array(1...3)
```

🗺️

```swift
// Funktion auf jedes Array Element
let mapResult = map(numbers) { x in x * x }
mapResult // [1,4,9]
```

🔍

```swift
// Filtert Array Elemente
let filterResult = filter(numbers) { x in x <= 2 }
filterResult // [1,2]
```

Funktionen HÖHERER ORDNUNG

FUNKTIONEN ALS RÜCKGABEWERTE
FUNKTIONEN ALS ARGUMENTE

Funktionen als INPUT ⬅️

```
func addOne(x: Int) -> Int {
      return x + 1 }

(1...3).map(addOne) // [2,3,4]
```

Funktionen als OUTPUT ➡️

```
func addTwoAfter(f: Int -> Int) -> (Int -> Int) {
      return { f($0) + 2 }
}

let addThree = addTwoAfter(addOne)
addThree(1) // 4
```

VERKETTEN ⚓

```
let numbers = Array(1...10)

let evenSum = numbers.filter {$0 % 2 == 0}
                     .reduce(0,+)
```

LAZY EVALUATION zzZ

```
let numbers = Array(1...999)
let firstnumber = numbers.lazy.map({$0 + 1}).first

firstnumber // 2
```

REKURSION 🔃

```swift
func printNumbers(n: Int) {
      if n > 1 {
            printNumbers( n - 1 )
      }
      print(n)
}

printNumbers(3)
// 1
// 2
// 3
```

GENERISCHE TYPEN 💭

🗺

```
func printStrings(array: [String]) {
      array.map { print ($0)} }

func printInts(array: [Int]) {
      array.map { print ($0)} }
```

🗺 💭

```
func printValues<Generic>(array: [Generic]) {
      array.map { print ($0)} }
```

IN SWIFT SIND VARIABLEN
NIE nil, AUßER
SIE SIND OPTIONALS.

MONADE

🗺️ ?

```
func map<U>(@noescape f: (Wrapped) throws -> U) rethrows -> U?
```

?

```
func addOne(someNumber: Int?) -> Int? {
        if let number = someNumber {
                return number + 1
        } else {
                return nil
        }
}
```

🗺️?

```
func addOne(someNumber: Int?) -> Int? {
        return someNumber.map { number in number + 1 }
}
```

```
addOne(5)   // Optional(6)
addOne(nil) // nil
```

.map DARF nil WIEDERGEBEN, .flatMap NICHT.

🗺️?

```
[[“a“], [nil, “b“]].flatMap { $0 } // [“a“, “b“]
```

CURRYING

🍛

🍚

```
func add🍚(x:Int, y:Int) -> Int {
        return x + y
}
```

🍛

```
func add🍛(x:Int) -> (Int -> Int) {
        return { y in x + y }
}
```

```
add🍚(1,2)      // 3
add🍛(1)(2)     // 3
```

```swift
let numbers = Array(1...3)
```

🍚

```swift
let addThree🍚 = { add🍚($0, 3) }
numbers.map(addThree🍚) // [4, 5, 6]
```

🍛

```swift
let addThree🍛 = add🍛(3)
numbers.map(addThree🍛) // [4, 5, 6]
```

DANKE!

Fragen? ✋