

# Funktionale Programmierung

Wie *map*, *filter* und *flatMap* den Tag retten

# Funktionale Programmierung

- Funktionale Programmierung ist ein Paradigma  
Aus Wikipedia:

Ein **Paradigma**<sup>[1]</sup> ([Pl.](#) *Paradigmen* oder *Paradigmata*) ist eine grundsätzliche Denkweise. Seit dem späten 18. Jahrhundert bezeichnet Paradigma eine bestimmte Art der [Weltanschauung](#) oder eine [Lehrmeinung](#).

- Funktionale Programmierung != Imperatives/Objektorientiertes Programmieren



Imperativ

Funktional

Computer als Maschine

Computer als Auswerter von  
Ausdrücken

Programm als eine Folge von  
Befehlen

Programm als einzelner Ausdruck

```
var x = 41  
x++
```

```
val a = x(y(z(param1)), f(param2))
```



# Imperativ      60      Funktional

Loops und Jumps als  
Grundelemente

```
for(x in xs) {  
    break;  
}
```

Ändern von Speicherzellen als  
Memory

```
var m = 0  
m = 1  
m = 2
```

Ausdruck als Grundelement

```
x * y + z
```

Zuweisen von Namen zu  
Ausdrücken als Memory

```
val a = x * y + z  
val b = a * 5
```



DRAW!

# Vorteile von FP

- Parallelisierbar  
Operationen

# Was ist FP?

- Funktionen als Operationen benutzen
- Funktionen als Abstraktion benutzen
- Funktionen als Werte behandeln
- Funktionen als Resultate bekommen
- Funktionen...

**It's all about functions!**

# Was ist eine Funktion?

Aus Wikipedia:

Eine Funktion  $f$  ordnet jedem Element  $x$  einer Definitionsmenge  $\mathbb{D}$  genau ein Element  $y$  einer Zielmenge  $Z$  zu.

# Was ist eine Funktion – in Kotlin

```
fun sum(a: Int, b: Int): Int {  
    return a + b  
}
```

```
fun sum(a: Int, b: Int) = a + b
```

```
fun printSum(a: Int, b: Int): Unit {  
    println("sum of $a and $b is ${a + b}")  
}
```



# Theorie: Referentielle Transparenz

- Beschreibt eine Eigenschaft von Expressions:

```
val x = f(42)  
val y = g(x)  
val z = h(x)
```

=

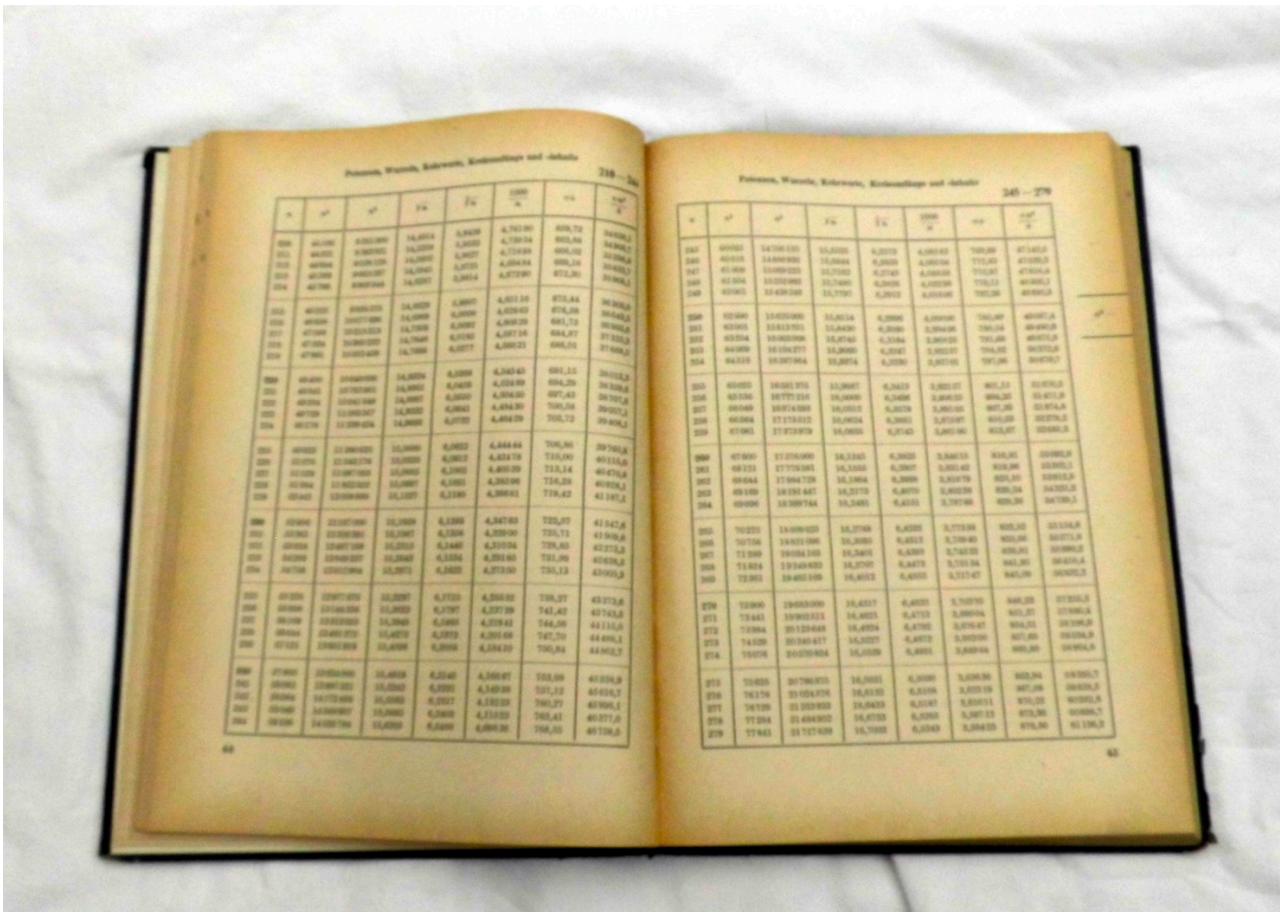
```
val y = g(f(42))  
val z = h(f(42))
```

- Erlaubt (möglicherweise) den Beweis der Korrektheit
- Erlaubt automatische Optimierungen wie Caching, Lazy Evaluation und Parallelisierung

# Funktionen als Objekte

Wie soll man sich das vorstellen?

# Logarithmen-Buch – Eine alte ‘log’-Funktion



# Eine Tabelle...

x	y
1	1
2	4
3	9
4	16
5	25

# Ein Schredder!



# Arbeiten mit Funktionen

```
fun f(x: Int): Int = ...
fun g(y: Int): String = ...
fun h(z: String): Int = ...
```

```
val x:Int = 10
val y:Int = f(x)
val z:String = g(y)
val a:Int = h(z)
```

```
val a:Int = h(g(f(10)))
```



# Arbeiten mit Funktionen – Tiramisu machen



# Schoggimousse machen

```
fun schoggiMousse_machen(eier: Eier, zucker: Zucker, rahm: Rahm,  
schokolade: Schokolade): SchoggiMousse {  
    val (eiweiss, eigelb) = trennen(eier)  
    val eischnee = schlagenBisFestUndGlänzend(  
        mischen(zucker, schlagen(eiweiss)))  
    val grundmasse = schlagenBisLuftigUndHellgelb(mischen(zucker, eigelb))  
    val schlagrahm = schlagenBisFest(Rahm)  
    val schockoGrundmasse = mischen(grundmasse, schmelzen(schokolade))  
    return kühlen(unterheben(  
        eischnee, unterheben(schockoGrundmasse, schlagrahm)))  
}
```

# Anatomie einer Kotlin Funktion

Eine Funktion braucht:

- Parameter Liste

```
(a:Int, b:String)
```

- Rückgabetyp

```
: String
```

- Body

```
    return "$a $b"  
}
```

- oder

```
= "$a $b"
```

# Was ist ein Lambda?

- Ein Block in geschweiften Klammern

```
val lambda: (Unit) -> Unit = { }
```

- Optional mit Parameter Liste

```
val param: (Int) -> Int = { p -> p }
val params: (Int, String) -> Int = { i, s -> i + s.length }
```

- Parameter List erlaubt Destrukturierung

```
val tuples: (Pair<Int, Long>, String) -> Long = { (i, l), s -> i + l +
s.length }
```

- Impliziter einzelner Parameter 'it' wenn Parameter Liste fehlt

```
val withIt: (Int) -> String = { it.toString() }
```

# Was ist ein Lambda?

- Ein Block in geschweiften Klammern

```
val lambda: (Unit) -> Unit = { }
```

- Optional mit Parameter Liste

```
val param: (Int) -> Int = { p -> p }
val params: (Int, String) -> Int = { i, s -> i + s.length }
```

- Parameter List erlaubt Destrukturierung

```
val tuples: (Pair<Int, Long>, String) -> Long = { (i, l), s -> i + l +
s.length }
```

- Impliziter einzelner Parameter 'it' wenn Parameter Liste fehlt

```
val withIt: (Int) -> String = { it.toString() }
```

# Was ist ein Lambda?

- Resultat

```
val lambda = { 10 } // Letzter Ausdruck ist Resultat  
  
lambda() == 10
```

- Return?

```
fun funktion(): Int {  
    listOf(100, 100).forEach {  
        return 42  
    }  
    return 10  
}
```

# Anonyme Funktion

- Wird mit dem ‘fun’-Keyword erstellt

```
fun name() : Int {  
    return 42  
}
```



```
val anonym = fun() : Int {  
    return 42  
}
```

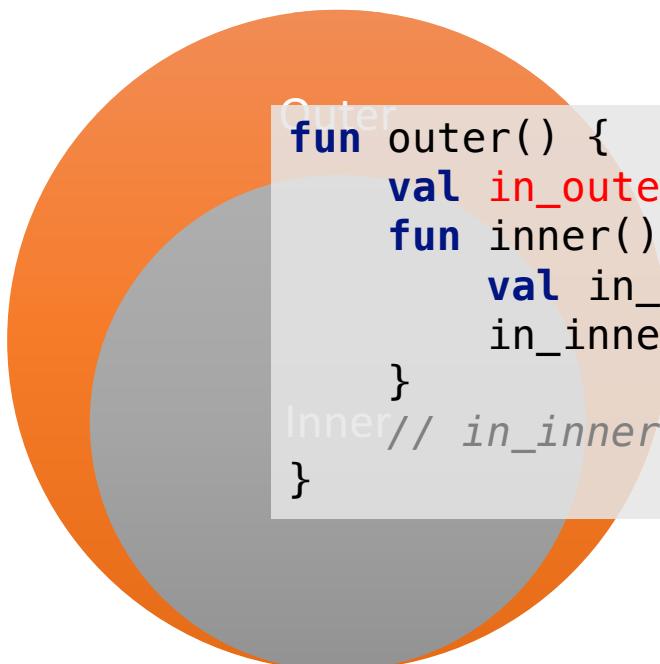
- Entweder ein einzelner Ausdruck oder ein Block

```
val anonym = fun() : Int {  
    return 42  
}
```

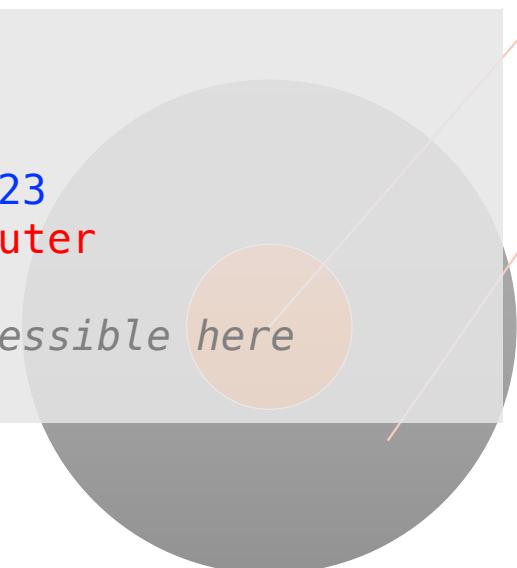
```
val anonym2 = fun() : Int = 42
```

# Scope

## Function Nesting

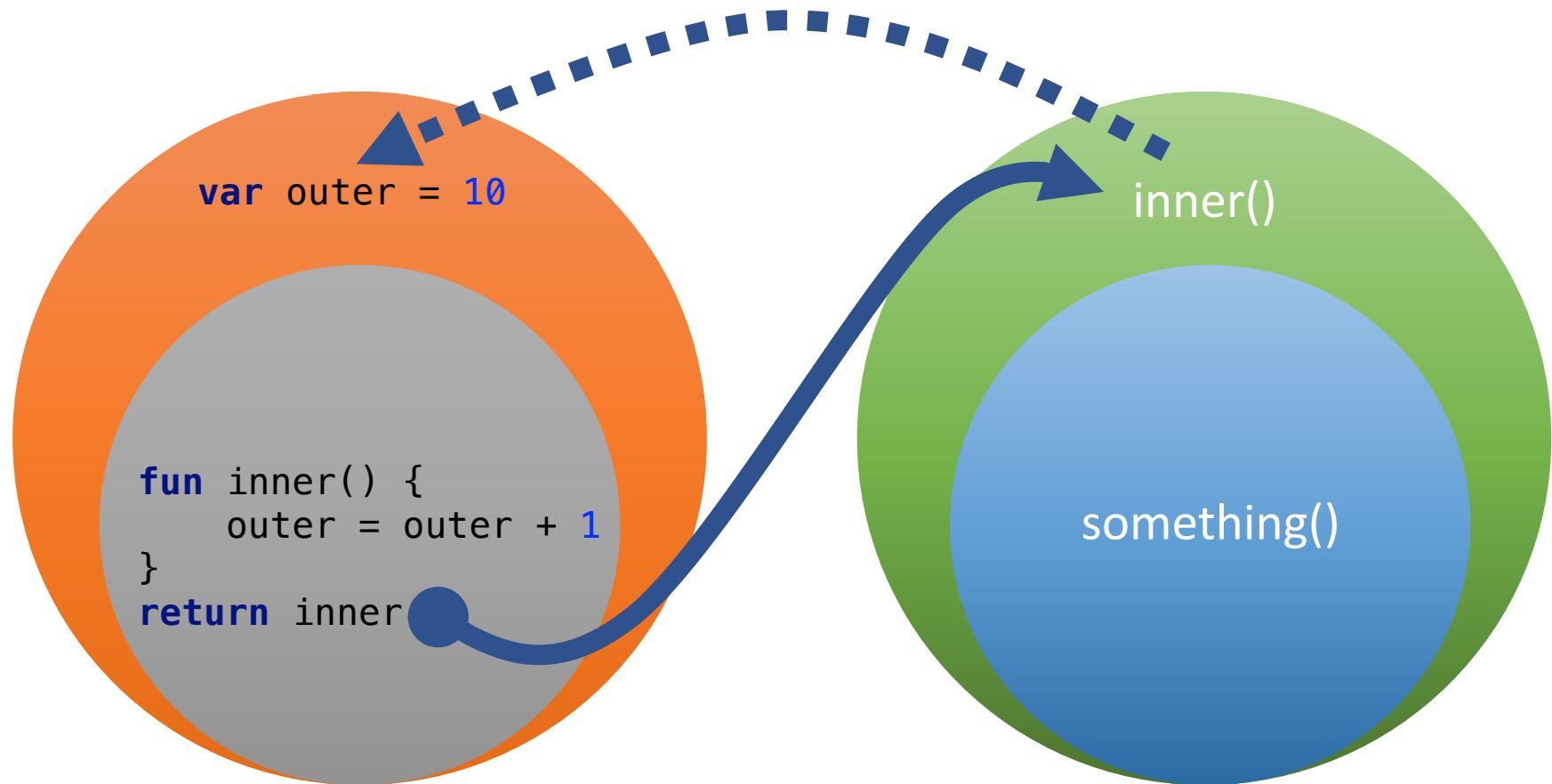


## Scope Nesting



Outer  
Inner

# Closure



# Currying – Spezielle Closure

```
fun multi(a: String, b: Int) : String {  
    return "$a $b"  
}
```

```
fun curry(a: String) : (Int) -> String {  
    return { b ->  
        "$a $b"  
    }  
}
```

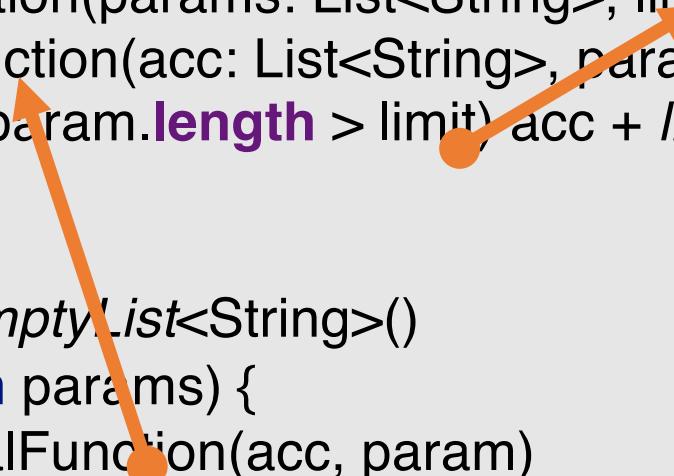


```
fun curry(a: String) : (Int) -> String = { b -> "$a $b" }
```

# Lokale Funktionen

```
fun someFunction(params: List<String>, limit: Int): List<String> {
    fun localFunction(acc: List<String>, param: String): List<String> {
        return if(param.length > limit) acc + listOf(param) else acc
    }

    var acc = emptyList<String>()
    for(param in params) {
        acc = localFunction(acc, param)
    }
    return acc
}
```



# Funktionen als Abstraktionen verwenden

Higher-Order Functions erlauben die Umsetzung von Verhalten auf der Basis von anderen Funktionen

→ HOF setzen das Strategy-Pattern um – jede Strategy ist eine Funktion

# Schoggimousse machen

```
fun schoggiMousse_machen(eier: Eier, zucker: Zucker, rahm: Rahm,  
schokolade: Schokolade): SchoggiMousse {  
    val (eiweiss, eigelb) = trennen(eier)  
    val eischnee = schlagenBisFestUndGlänzend(  
        mischen(zucker, schlagenBisFest(eiweiss))  
    )  
    val grundmasse = schlagenBisLuftigUndHellgelb(mischen(zucker, eigelb))  
    val schlagrahm = schlagenBisFest(rahm)  
    val schockoGrundmasse = mischen(grundmasse, schmelzen(schokolade))  
    return kühlen(unterheben(  
        eischnee, unterheben(schockoGrundmasse, schlagrahm))  
    )  
}
```

## Schoggimousse machen

```
<T> fun schlagen(  
    was: T,  
    bis: T -> Boolean = istFest) {  
    ...  
}
```

# Schoggimousse machen

```
fun schoggiMousse_machen(eier: Eier, zucker: Zucker, rahm: Rahm,  
schokolade: Schokolade): SchoggiMousse {  
    val (eiweiss, eigelb) = trennen(eier)  
    val eischnee = schlagen(mischen(zucker, schlagen(eiweiss))  
        )  
        → bis = masse → isFest(masse) && isGlänzend(masse)  
    val grundmasse = schlagen(mischen(zucker, eigelb),  
        )  
        → bis = masse → isLuftig(masse) && isHellgelb(masse)  
    val schlagrahm = schlagen(Rahm)  
    val schockoGrundmasse = mischen(grundmasse, schmelzen(schokolade))  
    return kühlen(unterheben(eischnee,  
        unterheben(schockoGrundmasse, schlagrahm))  
    )  
}
```

# Funktionen Höherer Ordnung

Operationen auf Kotlin Collections

- Map
- Filter
- FlatMap

# Map – Elemente verarbeiten

```
fun <in A, out B> map(elements: List<A>, f: (A) -> B) : List<B> {  
    var bs = listOf<B>()  
    for(element in elements) {  
        bs = listOf(f(element)) + bs  
    }  
    return bs  
}
```

```
map(listOf(1, 2, 3, 4, 5, 6), { it * 10 }) == listOf(10, 20, 30, 40, 50, 60)
```

1	10
2	20
3	30
4	40
5	50
6	60

# Filter – Elemente auswählen

```
fun <A> filter(elements: List<A>, p: (A) -> Boolean): List<A> {  
    var filtered = listOf<A>()  
    for (element in elements) {  
        if (p(element))  
            filtered = listOf(element) + filtered  
    }  
    return filtered  
}
```

```
filter(listOf(1, 2, 3, 4, 5, 6), { it % 2 == 0 }) == listOf(2, 4, 6)
```

1	
2	2
3	
4	4
5	
6	6

# FlatMap – Elemente vervielfachen

```
fun <in A,out B> flatMap(elements: List<A>, f: (A) -> List<B>): List<B> {  
    var bs = listOf<B>()  
    for (element in elements) {  
        bs = f(element) + bs  
    }  
    return bs  
}
```

```
flatMap(listOf(1, 2, 3, 4, 5, 6), { List(it) { i -> i } })  
== listOf(1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6)
```

1	1
2	2,2
3	3,3,3
4	4,4,4,4
5	5,5,5,5,5
6	6,6,6,6,6,6

# Aufgabe

Map and Filter mit Hilfe von FlatMap implementieren

→ In der Online Kotlin Konsole

# Hinweis: Map & Filter mit FlatMap

```
fun main(args: Array<String>) {  
    println( listOf(1,2,3).map { it + 10} )  
    println( listOf(1,2,3).flatMap { ??? } )  
    println( listOf(1,2,3,4,5).filter { it % 2 == 0 } )  
    println( listOf(1,2,3,4,5).flatMap { ??? } )  
}
```

# HOF Beispiel – Elemente verarbeiten

```
val elemente = listOf(1,2,3)
var resultate = listOf()
for(element in elemente) {
    resultate = listOf(element * 100) + resultate
}
```

```
val elemente = listOf(1,2,3)
val resultate = elemente.map { it * 100 }
```

```
val resultate = listOf(1,2,3).map { it * 100 }
```

# Aufgabe

Reddit Kommentare funktional verarbeiten

# Aufgabe

Imperative Verarbeitung der Reddit Kommentare funktional machen

- Branch ‘step\_06\_functional\_exercise’ auschecken
- Verwendet ‘map’, ‘filter’ und ‘flatMap’ an den markierten Stellen in den Funktionen
  - RedditNewsDataRemoteDataSource#flattenRetrofitResponse
  - RedditNewsDataRemoteDataSource#recursivlyParseResponse (Bonus)
- Ziel: die Loops und ‘Jumps’ (if) ersetzen

# Inlining

- Problem: Aufruf von Funktionen ist nicht gratis
  - Parameter bereitstellen: Allokation, Boxing, etc.
  - Eigentlicher Aufruf
  - Für Android: Dex-Limit
- Inline vermeidet diesen Aufwand
- Auch nicht gratis, aber andere Währung:  
Codegrösse → Compiler fügt den Code an der Aufrufstelle ein

# Inlining: Beispiel

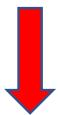
```
fun <T> lock(lock: Lock, body: () -> T): T {  
    lock.lock()  
    try {  
        return body()  
    }  
    finally {  
        lock.unlock()  
    }  
}
```

```
lock(l) { foo() }
```

# Inlining: Manuelle Variante

```
l.lock()
try {
    return foo()
}
finally {
    l.unlock()
}
```

# Inlining: Beispiel



```
inline fun <T> lock(lock: Lock, body: () -> T): T {  
    // ...  
}
```

# Inlining: Limitierungen

- Funktioniert nicht für lokale Funktionen
- Funktioniert nicht für Funktionen die lokale Funktionen enthalten
- Inline erstreckt sich automatisch auf Funktionsparameter – mit dem Keyword ‘noinline’ lässt sich das umgehen

```
inline fun foo(inlined: () -> Unit, noinline notInlined: () -> Unit) {  
    // ...  
}
```

# Quellen & Informationen über FP

## Quellen

- [www.lihaoyi.com/post/WhatsFunctionalProgrammingAllAbout.html](http://www.lihaoyi.com/post/WhatsFunctionalProgrammingAllAbout.html)

## Blogs & Videos

- <https://medium.com/@JorgeCastilloPr/kotlin-functional-programming-does-it-make-sense-36ad07e6bacf>
- <https://medium.com/@BladeCoder/exploring-kotlins-hidden-costs-part-1-fbb9935d9b62>
- <https://www.youtube.com/user/DrBartosz/playlists> → Category Theory (Gute Einführung)