Kotlin

Functional Programming

https://kotlinlang.org/

Functional Programming - Konzepte

- Immutability
- Lambda Expressions
  - Anonyme Funktionen, die es erlauben präzise Funktionen zu definieren, wo diese gebraucht werden.
  - Da diese Funktionen sonst nirgends gebraucht werden, erhalten sie keinen Namen und werden in keinem Objekt gespeichert
- High Order Functions
  - Ein Funktion wird als Argument übergeben und/oder als Funktionswert zurückgegeben
  - Die ergibt Code der besser wiederverwendbar ist

Lazy Evaluation

- Strict evaluation
- Lazy evaluation
    - somit können „unendliche“ Listen verarbeitet werden

Lambdas

```
{ x: Int -> x * 2 }
```

nicht notwendig, kann vom Compiler abgeleitet werden

```
val timesTwo:(Int) -> Int = { x: Int -> x * 2 }
```

```
val timesTwo = { x: Int -> x * 2 }
```

mit zwei Eingangsparametern

```
val add: (Int, Int) -> Int = { x: Int, y: Int -> x + y }
```

Higher Order Functions

```
val list = (1..100).toList()
```

predicate: (Int) -> Boolean

```
list.filter()
```

Falls es nur einen Parameter gibt, kann man auch den impliziten Iterator „it“ verwenden

```
list.filter({ element: Int -> element % 2 == 0 })

list.filter({ element -> element % 2 == 0 })

print(list.filter({ element: Int -> element % 2 == 0 }))

print(list.filter({ it % 2 == 0 }))
```

Run basics.LambdasKt

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/bin/java ...
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66,
 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100]
Process finished with exit code 0
```

Fortsetzung auf nächster Folie

Extension function

```
package basics

fun main(args: Array<String>) {

    val list = (1..100).toList()

    print(list.filter({ it.even() }))
}

fun Int.even() = this % 2 == 0

print(list.filter { it.even() })
```

Verwendung der „Extension function“

Deklaration der „Extension function“

Ein letzte Verkürzung, da nur ein Parameter bzw. der letzte Parameter

```
print(list.filter({ it.even() }))

Int.even() = this
```

```
m equals(other: Any?)
λ even() for Int in basics
m and(other: Int)
```

wird auch in der IntelliSense angezeigt

Function References

```
package basics

fun main(args: Array<String>) {

    val list = (1..100).toList()

    print(list.filter(::isEven))
}

fun isEven(i: Int) = i % 2 == 0
```

eigener Operator ::

.map()

wendet eine Funktion bei jedem Element an

```
package basics

fun main(args: Array<String>) {

    val list = (1..100).toList()

    val doubled =list.map { element -> element * 2 }

    println(doubled)
}
```

```
package basics

fun main(args: Array<String>) {

    val list = (1..100).toList()

    val doubled =list.map { it * 2 }

    println(doubled)
}
```

kürzer mit implizitem Iterator

map()

```
package basics

fun main(args: Array<String>) {

    // map()
    val list = (1..100).toList()

    // val doubled =list.map { element -> element * 2 }
    val doubled =list.map { it * 2 }

    val average = list.average()
    val shifted = list.map { it - average}

    println(doubled)
    println(shifted)

}
```

flatMap()

wendet eine Funktion bei jedem Element an <u>und</u> „verflacht“ (flatten) verschachtelte Collections

```
// flatMap()
val nestedList = listOf(
        (1..10).toList(),
        (11..20).toList(),
        (21..30).toList()
)

val notFlattened = nestedList.map { it.sortedDescending() }
println(notFlattened)

val flattened = nestedList.flatMap { it.sortedDescending() }
println(flattened)
```

Alternative zu flatMap()

```
val notFlattened = nestedList.map { it.sortedDescending() }.flatten()
```

BEACHTE: die jeweiligen absteigend sortierten Maps wurden „nur“ konkateniert. Die gesamte Liste ist <u>nicht</u> geordnet

Run basics.MapKt

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/bin/java ...
[[10, 9, 8, 7, 6, 5, 4, 3, 2, 1], [20, 19, 18, 17, 16, 15, 14, 13, 12, 11], [30, 29, 28, 27, 26, 25, 24, 23, 22, 21]]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21]

Process finished with exit code 0
```

take() and drop()

take() nimmt n Elemente und fügt diese einer Liste hinzu
drop() nimmt die ersten n Elemente nicht und fügt den Rest der Elemente der Liste hinzu

```
package basics

fun main(args: Array<String>) {

    val list = (1..1000).toList()

    val first10 = list.take(10)
    val withoutFirst900 = list.drop(900)

    println(first10)
    println(withoutFirst900)
}
```

Run basics.Take_dropKt

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/bin/java ...
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[901, 902, 903, 904, 905, 906, 907, 908, 909, 910, 911, 912, 913, 914, 915, 916, 917, 918, 919, 920, 921, 922, 923, 924, 925,
 926, 927, 928, 929, 930, 931, 932, 933, 934, 935, 936, 937, 938, 939, 940, 941, 942, 943, 944, 945, 946, 947, 948, 949, 950,
 951, 952, 953, 954, 955, 956, 957, 958, 959, 960, 961, 962, 963, 964, 965, 966, 967, 968, 969, 970, 971, 972, 973, 974, 975,
 976, 977, 978, 979, 980, 981, 982, 983, 984, 985, 986, 987, 988, 989, 990, 991, 992, 993, 994, 995, 996, 997, 998, 999, 1000]

Process finished with exit code 0
```

take()

```
val list = generateSequence(0) {
    println("Calculating ${it + 10}")
    it + 10
}

val first10 = list.take(10).toList()
val first20 = list.take(20).toList()

println(first10)
println(first20)
```

```
Calculating 10
Calculating 20
Calculating 30
Calculating 40
Calculating 50
Calculating 60
Calculating 70
Calculating 80
Calculating 90
Calculating 10
Calculating 20
Calculating 30
Calculating 40
Calculating 50
Calculating 60
Calculating 70
Calculating 80
Calculating 90
Calculating 100
Calculating 110
Calculating 120
Calculating 130
Calculating 140
Calculating 150
Calculating 160
Calculating 170
Calculating 180
Calculating 190
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150, 160, 170, 180, 190]
```

Leider werden die ersten 10 Werte 2x berechnet. Trotzdem ist die Funktion take() für die Performance sehr wichtig: Zuerst sind mit take() die Elemente auszuwählen und erst anschließend sind die weiteren Berechnungen auf diese Elemente auszuführen

Weitere wichtige Funktionen:

- first()
- last()

first() and last()

- n/a

zip()

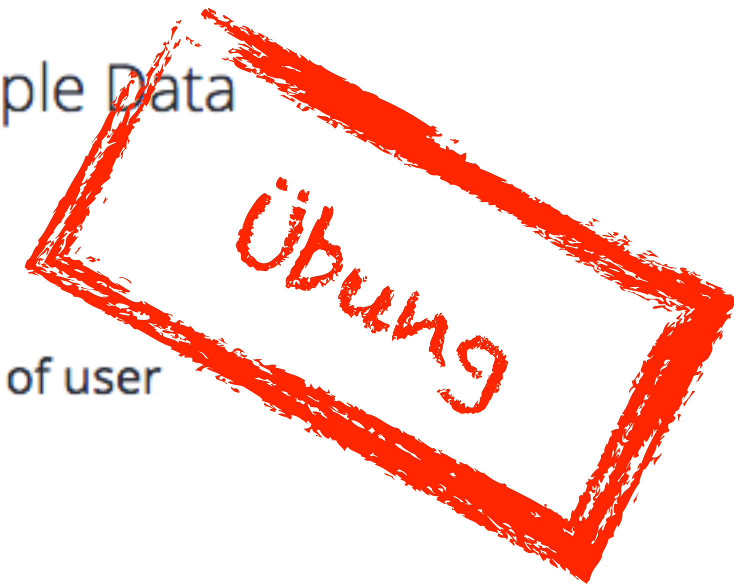
fügt zusammengehörende Werte zu je einem Paar zusammen

```
val list = listOf("Hi", "there", "Kotlin", "fans")
val containsT = listOf(false, true, true, false)

// Pair<String, Boolean>
val zipped: List<Pair<String, Boolean>> = list.zip(containsT)

// diese Vorgangsweise ist eher real-world
val mapping = list.zip(list.map { it.contains("t") })

println(zipped)
println(mapping)
```

Run basics.ZipKt

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/bin/java ...
[(Hi, false), (there, true), (Kotlin, true), (fans, false)]
[(Hi, false), (there, true), (Kotlin, true), (fans, false)]
```

Challenge: Apply Functional Programming for Simple Data Analysis



We decided to gather data on the age of our users.

In this challenge, you'll be presented with this partly faulty data of user ages which is based on four input files:

```kotlin
// Some faulty data with ages of our users
val data = mapOf(
        "users1.csv" to listOf(32, 45, 17, -1, 34),
        "users2.csv" to listOf(19, -1, 67, 22),
        "users3.csv" to listOf(),
        "users4.csv" to listOf(56, 32, 18, 44)
)
```

Apply the functions you learned about as well as other functions from Kotlin's standard library to solve the following data analysis tasks:

1. Find the average age of users (use only valid ages for the calculation!)
2. Extract the names of input files that contain faulty data
3. Count the total number of faulty entries across all input files

Hints

- map() and flatMap() are often very useful functions for such tasks
- Use IntelliJ's autocompletion to explore which other functions, that were not presented in the lectures, are available (they will drastically simplify the tasks)

Lösung

```kotlin
val data = mapOf(
        "users1.csv" to listOf(32, 45, 17, -1, 34),
        "users2.csv" to listOf(19, -1, 67, 22),
        "users3.csv" to listOf(),
        "users4.csv" to listOf(56, 32, 18, 44)
)

// 1. Average age of users
val validList = data.flatMap { it.value }.filter { it -> it > 0 }
val average = validList.average()
println(data)
println(validList)
println("Arithmet. Durchschnitt = %.2f".format(average))
println()

// 2. Files with faulty data
val filenames = data.filter { it.value.any{ it < 0 } }
                    .map {it.key}
println("Dateien mit fehlerhaften Werten: $filenames")

// 3. Number of faulty entries
val numberOfFaults = data.flatMap { it.value }
                         .filter { it < 0 }
                         .count()
println("Anzahl der Fehler: $numberOfFaults")
```

```
{users1.csv=[32, 45, 17, -1, 34], users2.csv=[19, -1, 67, 22], users3.csv=[], users4.csv=[56, 32, 18, 44]}
[32, 45, 17, 34, 19, 67, 22, 56, 32, 18, 44]
Arithmet. Durchschnitt = 35.09

Dateien mit fehlerhaften Werten: [users1.csv, users2.csv]
Anzahl der Fehler: 2
```

chaining

```
val inputRows = listOf(
        mapOf("input01.csv" to listOf(3, 5, -9977, 7, 11, 66)),
        mapOf("input02.csv" to listOf(1, 3, 4)),
        mapOf("input03.csv" to listOf()),
        mapOf("input04.csv" to listOf(9989, 33, 14, 12, 5))
)

val cleaned = inputRows.flatMap { it.values }
        .flatten()
        .filter { it in 0..100 }
        .toIntArray()   // nicht notwendig

println(cleaned.joinToString())
```

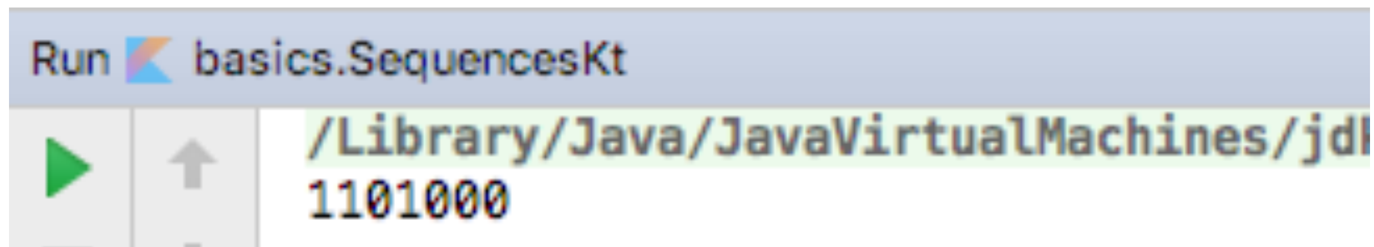
Run basics.ChainingKt

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/bin/java ...
3, 5, 7, 11, 66, 1, 3, 4, 33, 14, 12, 5

Process finished with exit code 0
```

Lazy Sequences 1

```
val veryLongList = (1..999999L).toList()

val sum = veryLongList
        .filter { it > 50 }
        .map { it * 2 }
        .take(1000)
        .sum()

println(sum)
```

Da hier Long verwendet wird, ist auch „sum“ vom Typ Long und es gibt keinen Überlauf beim Ergebnis



Dies ist noch keine Sequence sondern „nur“ eine List. Auch ist hier take(…) nach map(…), was nicht vorteilhaft ist, da Berechnungen auf Elementen ausgeführt werden, die später verworfen werden.

Lazy Sequence 2

```
package basics

fun main(args: Array<String>) {

    val veryLongList = (1..999999L).toList()

    val start = System.currentTimeMillis()

    val sum = veryLongList
            .asSequence()
            .filter { it > 50 }
            .map { it * 2 }
            .take(1000)
            .sum()

    println(sum)
    println(System.currentTimeMillis() - start)
}
```

„Lazy Evaluation“

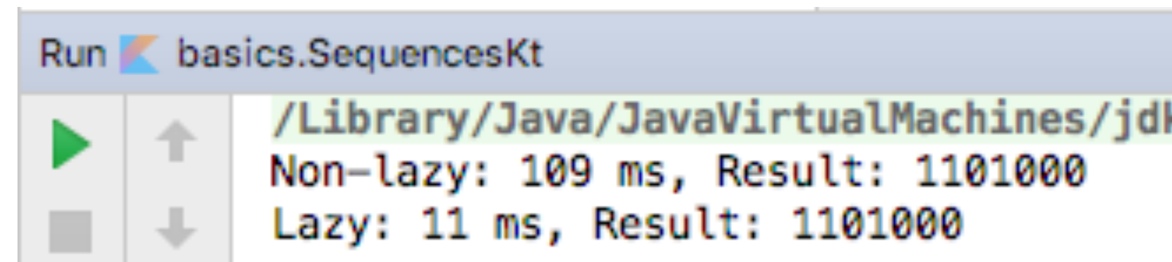
Ohne asSequence() benötigt ein Lauf ca 85-95 ms mit asSequence() ca. 12-16 ms

Hier wird nicht die gesamte Liste in den Hauptspeicher geladen und gefiltert und anschließend die Berechnungen für die gesamte Liste durchgeführt, sondern nur die ersten 1000 Elemente

measureTimeMillis{ }

```
val veryLongList = (1..999999L).toList()

var sum = 0L
var lazysum = 0L

val msNonLazy = measureTimeMillis {
    sum = veryLongList
        .filter { it > 50 }
        .map { it * 2 }
        .take(1000)
        .sum()
}

val msLazy = measureTimeMillis {
    sum = veryLongList
            .asSequence()
            .filter { it > 50 }
            .map { it * 2 }
            .take(1000)
            .sum()
}
val seq = generateSequence(1, { it + 1})

println("Non-lazy: $msNonLazy ms, Result: $sum")
println("Lazy: $msLazy ms, Result: $sum")
```

eine weitere Möglichkeit die Zeit zu messen



mit Optimierung

```
val veryLongList = (1..999999L).toList()

var sum = 0L
var lazysum = 0L

val msNonLazy = measureTimeMillis {
    sum = veryLongList
        .filter { it > 50 }
        .take(1000)
        .map { it * 2 }
        .sum()
}

val msLazy = measureTimeMillis {
    sum = veryLongList
            .asSequence()
            .filter { it > 50 }
            .take(1000)
            .map { it * 2 }
            .sum()
}
val seq = generateSequence(1, { it + 1})

println("Non-lazy: $msNonLazy ms, Result: $sum")
println("Lazy: $msLazy ms, Result: $sum")
```

Bei der Sequence sind keine Performancesteigerungen erkennbar, bei der Liste sehr wohl. —> Regel: Die take()-Methode so bald als möglich verwenden, da sich so die Anzahl der Elemente reduziert

```
Run basics.SequencesKt
/Library/Java/JavaVirtualMachines/
Non-lazy: 32 ms, Result: 1101000
Lazy: 15 ms, Result: 1101000
```

Größere Last

```
val veryLongList = (1..999999L).toList()

var sum = 0L
var lazysum = 0L

val msNonLazy = measureTimeMillis {
    sum = veryLongList
            .filter { it > 50 }
            .map { it * 2 }
            .map { it / 3 }
            .map { it + 17 }
            .take(1000)
            .sum()
}

val msLazy = measureTimeMillis {
    sum = veryLongList
            .asSequence()
            .filter { it > 50 }
            .map { it * 2 }
            .map { it / 3 }
            .map { it + 17 }
            .take(1000)
            .sum()
}
val seq = generateSequence(1, { it + 1 })

println("Non-lazy: $msNonLazy ms, Result: $sum")
println("Lazy: $msLazy ms, Result: $sum")
```

Erhöht man die „Last" und gibt das take() wieder nach dem map() so sieht man bei der Liste schon größere Laufzeiteunbußen

```
Run basics.SequencesKt
/Library/Java/JavaVirtualMachines/
Non-lazy: 233 ms, Result: 383667
Lazy: 11 ms, Result: 383667
```

Sieb des Erathostenes

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120

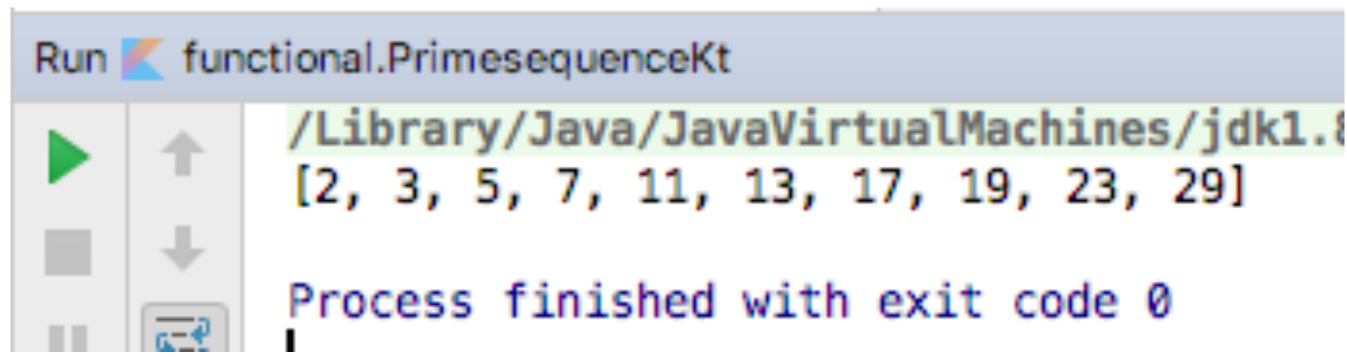
Primzahlen:

Die Vielfachen von Zahlen werden gestrichen.
Der Rest sind Primzahlen.

https://de.wikipedia.org/wiki/Sieb_des_Eratosthenes

Sieb des Erathostenes

```kotlin
package functional

fun main(args: Array<String>) {

    val possiblePrimesAfter2 = generateSequence(3) { it + 2 }
    val primes = generateSequence( 2 to possiblePrimesAfter2 ) {

        // next prime number
        val p = it.second.first()

        // filter out all elements divisible by p
        val possiblePrimesAfterP = it.second.filter { it % p != 0 }

        // return the next element in the sequence
        //Pair(p, possiblePrimesAfterP)
        p to possiblePrimesAfterP

    }.map { it.first }

    println(primes.take(10).toList())
}
```



let()

useful for **scoping** and working with **nullables**

```
// Scoping
File("example.txt").bufferedReader().let {
    if (it.ready()) {
        println(it.readLine())
    }
}

// reader should not be visible
```

Scoping bedeutet, dass der bufferedReader nur innerhalb des let-Blockes gültig ist. Es gibt daher keine Seiteneffekte

```
// Working with nullables
val str: String? = "Kotlin for Android"
str?.let {
    if (str.isNotEmpty()) {
        str.toLowerCase()
    }
```

Smart cast to kotlin.String

Dieser Block wird nur ausgeführt, wenn der String nicht null ist. Innerhalb des Blocks erfolgt ein Smart cast zu einem „normalen“ String

with()

Um mehrere Aufrufe auf ein Objekt ausführen zu können, ohne den Variablennamen wiederholen zu müssen. vgl WITH-Klausel in SQL

```
val props = System.getProperties()

with(props) {
    list(System.out)
            // anstelle von props.list(System.out)
    println(propertyNames().toList())
            // anstelle von props.propertyNames(...)
    println(getProperty("user.home"))
            // anstelle von props.getProperty(...)
}
```

erhöht die Strukturierung des Codes

use()

äquivalent zu Javas try-with-resources

```
FileReader("mayexist.txt").use {
    val str = it.readText()
    println(str)
}
```

für alle Objekte verfügbar, die das Closeable-Interface implementieren

inline-Functions

Variante 1

```
fun modifyString(str: String): String {
    return str.toUpperCase()
}
```

Variante 2

```
inline fun modifyString(str: String, operation: (String) -> String): String {
    return operation(str)
}

fun main(args: Array<String>) {
    modifyString("Kotlin is amazing", { it.toUpperCase() })
}
```

mit dem Schlüsselwort „inline“ optimiert der Compiler den Code und ersetzt „operation(str)“ mit „it.toUpperCase()“

Noch Fragen?