kotlin.
grundlagen und anwendung.

Martin Boßlet

Kryptographie & Digitale Signatur
Web-Anwendungen
Programmiersprachen
Open Source

/me

Spring Boot
Spring & Spring 5
Groovy / Grails
Kryptographie
Ruby / Rails
JavaScript
Git

Agenda

I. Kotlin Basics

Prozedurale Programmierung
Objektorientierte Programmierung
Funktionale Programmierung
Fortgeschrittene Konzepte

II. Kotlin im Alltag

I. Kotlin Basics

Viele Wege führen

zum Hello World

Hello World - IntelliJ

1. Create New Project

2. Java + Kotlin/JVM

3. „helloworld-intellij“

4. src → New Kotlin File/Class „app“

5. fun main() { println(„Hello World!“) }

6. Run ‚AppKt‘

Hello World – Compiler JVM

1. Kotlin Compiler:
https://github.com/JetBrains/kotlin/releases/tag/v<Version>
https://github.com/JetBrains/kotlin/releases/tag/v1.3.20

2. app.kt mit
fun main() { println(„Hello World!“) }

3. kotlinc/bin/kotlinc app.kt -include-runtime -d app.jar

4. java -jar app.jar

oder

5. kotlinc/bin/kotlin -classpath app.jar AppKt

Hello World – Compiler Native

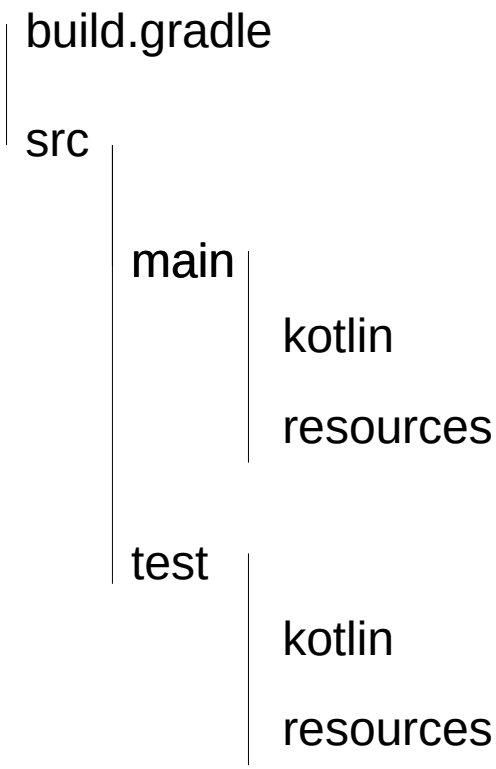
1. Kotlin Native Compiler:
https://github.com/JetBrains/kotlin/releases/tag/v<Version>
https://github.com/JetBrains/kotlin/releases/tag/v1.3.20

2. app.kt mit
fun main() { println(„Hello World!“) }

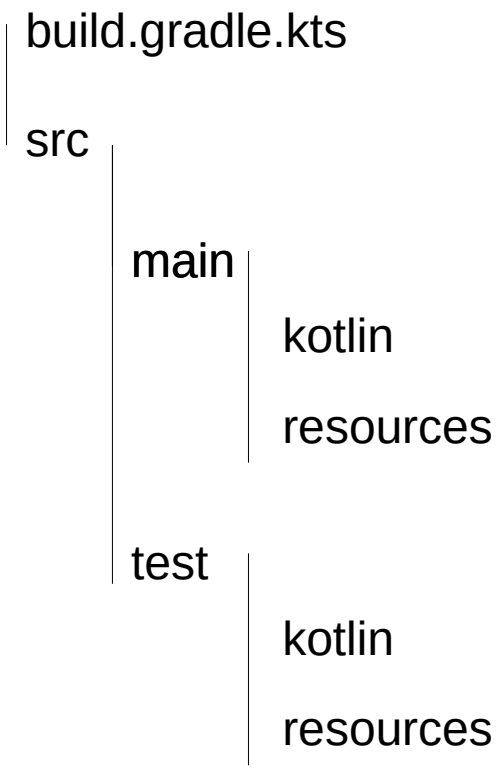
3. kotlin-native-linux-1.1.1/bin/kotlinc-native app.kt -o app

4. ./app.kexe

Hello World – Gradle (Groovy DSL)


build.gradle
src
main
kotlin
resources
test
kotlin
resources

Hello World – Gradle (Kotlin DSL)


build.gradle.kts
src
main
kotlin
resources
test
kotlin
resources

build.gradle.kts

```
plugins {
    kotlin("jvm") version "1.3.31"
    application
}

repositories {
    mavenCentral()
}

dependencies {
    implementation(kotlin("stdlib-jdk8"))
}

application {
    mainClassName = "AppKt"
}

tasks {
    withType<Jar> {
     manifest {
         attributes["Main-Class"] = application.mainClassName
     }
     from(configurations.compile.get().map { if (it.isDirectory) it else zipTree(it) })
    }
}
```

Wir nutzen im weiteren Verlauf Gradle mit Groovy-DSL

(da momentan noch weiter
verbreitet und mehr Hilfestellung
im Netz, wenn etwas schief läuft)

Kotlin ↔ Java

Interoperabilität

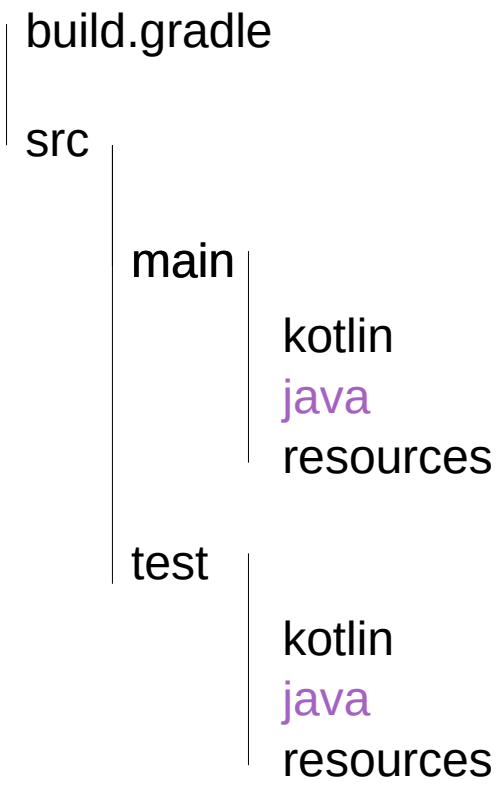
Gute Nachricht:

In build.gradle kommt nichts
Neues hinzu

build.gradle

```
plugins {
    id 'org.jetbrains.kotlin.jvm' version '1.3.31'
    id 'application'
}

repositories {
    mavenCentral()
}

dependencies {
    implementation('org.jetbrains.kotlin:kotlin-stdlib-jdk8')
}

application {
    mainClassName = "AppKt"
}

jar {
    manifest {
     attributes("Main-Class": application.mainClassName)
    }

    from {
        configurations.compile.collect { it.isDirectory() ? it : zipTree(it) }
    }
}
```

Auf Verzeichnisebene
kommt src/java hinzu


build.gradle
src
main
kotlin
java
resources
test
kotlin
java
resources

Grob vereinfacht:

Kotlin-Klassen in Java wie Java-Klassen nutzen

Java-Klassen in Kotlin wie Kotlin-Klassen nutzen

Kotlin-Klasse in Java nutzen

```
package workshop;

public class JavaWithKotlinGreeter implements Greeter {

    @Override
    public String greet() {
        return "Hello from Java. Kotlin says: " +
            new KotlinGreeter().greet();
    }
}
```

Java-Interface und -Klasse in Kotlin nutzen

```
package workshop



class KotlinWithJavaGreeter : Greeter {
    override fun greet(): String {
        return "Hello from Kotlin. Java says: ${JavaGreeter().greet()}"
    }
}
```

Übung: 06-gradle-java-kotlin

Prozedurale Programmierung

Der Anfang von allem:

Variablen deklarieren

Immutable → val

Unveränderliche Variablen mit val

```
val a = 42
a = 19 // => Compile-time error „a cannot be reassigned“

val b = „Ein String“  // Typ wird gefolgert („inferred“)

val c: Int = 1  // Explizite Typdeklaration

val d: String // Deklaration ohne Initialisierung
d = „Lazy assignment“  // Nachträgliche Zuweisung

val x
// Fehlende Typdeklaration => Compile-time error
// „This variable must either have a type annotation or be initialized“
```

Vorsicht:

val bedeutet nicht,
dass sich das Objekt selbst
nicht mehr verändert

val heißt nicht unveränderlich

```
val list = arrayListOf(1, 2, 3)

println(list) // => [1, 2, 3]

list.add(4)

println(list) // => [1, 2, 3, 4]
```

val bedeutet nur:

Die Referenz auf das Objekt
ist unveränderlich

Und kann nicht neu zugewiesen werden

Mutable → var

Veränderliche Variablen mit var

```
var a = 42
a = 19 // => OK

var b = 1
b += 1

var b = „Ein String“  // Typ wird gefolgert („inferred“)

var c: Int = 1  // Explizite Typdeklaration

var d: String // Deklaration ohne Initialisierung
d = „Lazy assignment“  // Nachträgliche Zuweisung

var x
// Fehlende Typdeklaration => Compile-time error
// „This variable must either have a type annotation or be initialized“
```

var erlaubt zwar Neuzuweisung,

aber der Typ ist fest!

```
var a = 1  // a: Int
a = „Hallo“  // geht nicht!
```

Kommentare: Wie in Java

Kommentare

```
// Zeilenweise
// Kommentare
// schreiben


/* Oder
   als
   Block
*/


/* Oft werden
 * in den einzelnen
 * weitere ‚*‘ eingefügt
 * zur Orientierung,
 * diese sind aber
 * optional
 */
```

Funktionsdeklarationen → fun

Funktionsdeklarationen

```
fun b(): Unit {
    println(„b“)
}

fun a() {
    println(„a“)
}  // Unit als Rückgabetyp optional

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

fun adder2(a: Int, b: Int) {
    return a + b
} // Kompiliert nicht, braucht Rückgabetyp

fun adder3(a: Int, b: Int) = a + b
// Funktion mit „Expression-Body“: Rückgabetyp wird gefolgert („inferred“)

fun adder4(a: Int, b: Int): Int = a + b
// Funktion mit „Expression-Body“ und explizitem Rückgabetyp
```

fun main

Einstiegspunkt für ausführbares Programm

Das kleinste Kotlin-Programm

```
fun main(){}


// main mit Argumenten

fun main(args: Array<String>) { … }
```

Argumenttypen müssen angegeben werden

Unit als Rückgabetyp optional

Funktion mit Expression-Body:
Rückgabetyp optional

Primitive Typen

Zahlen

Byte	8 Bit
Short	16 Bit
Int	32 Bit
Long	64 Bit
Float	32 Bit
Double	64 Bit

Zahlen - Beispiele

```
val a = 42
println(a.javaClass) // => int
// Ganzzahlen sind per Default vom Typ Int

val b = 3.14
println(b.javaClass) // => double
// Fließkommazahlen sind per Default vom Typ Double

val l = 1000L
println(l.javaClass) // => long

val f = 3.14F
(val f = 3.14f)
println(f.javaClass) // => float

val s: Short = 1

val b: Byte = 1

val hex = 0xFF

val binary = 0b011001

val oneMillion = 1_000_000
```

Vorsicht:

Keine impliziten Casts wie in Java

„Kleinere“ Typen sind keine Subtypen

Zahlen - Typ-Casts

```
val a: Byte = 1
val b: Int

b = a // Type mismatch

b = a.toInt() // OK


.toByte()

.toShort()

.toInt()

.toLong()

.toFloat()

.toDouble()
```

Vorsicht:

Explizite Konvertierung ist keine Garantie

Zahlen – Vorsicht bei expliziter Konvertierung

```
val a = 1000
val b: Byte = a.toByte()

println a // => 1000
println b // => -24
```

Characters

Char

```
val c = ‚A‘
val d: Char = ‚A‘

if (a == 42) // Compile-time error „Operator == cannot be applied to Char and Int“

println(a.toInt()) // => 65

// Escape-Sequenzen

var v = ‚\t‘

v = ‚\n‘

v = ‚\\‘

v = ‚\‘‘

v = ‚\$‘

v = ‚\u00AB‘
```

Booleans

Boolean

```
val b = true

val c: Boolean = false
```

Strings

Strings

```
val s = „Ein String“

val t: String = „Zwei String“

// Interpolation

val st = „$s $t“ // => „Ein String Zwei String“

val u = „Das Ergebnis: ${calculator.getResult()}“  // Ausdrücke (Expressions) mit ${…}


// Konkatenation

val st = s + “ “ + t // => gleiches Ergebnis, effizient!
```

Raw Strings

```
val s = """
  a
    b
      c
"""

// Zeilenenden & Einschübe bleiben erhalten

val xml = """
    |<root>
    |  <document>
    |    <text>Eins</text>
    |  </document>
    |</root>
""".trimMargin()

// Zeilenenden bleiben erhalten. Die ‚|‘ inklusive des Whitespace davor werden eliminiert.

val xml = """
    |<root>
    |  <document>
    |    <text>${document.getText()}</text>
    |  </document>
    |</root>
""".trimMargin()
```

Control Flow
(Kontrollfluss)

if else

if als Statement

```
if (variable == 42) {
    println(„42!“)
} else {
    println(„Something else“)
}


if (maybeTrue) {
    println(„Yes, it was true!“)
}


if (firstCondition) {
    println(1)
} else if (secondCondition) {
    println(2)
} else {
    println(„Die Else“)
}


if (a < b) println(„A kleiner“)

if (a < b) println(„a“) else println(„b“)
```

if als Expression

```
val max = if (a < b) b else a
// Statt in Java: max = a < b ? b : a


val result = if (variable == 42) {
    „42“
} else {
    „Something else“
}



// Geht nicht:

val result = if (fulfilled) 19

val result = if (fulfilled) {
    19
}

// Compile-time error
```

when

when

```
when (level) {
    1 → println(„basic“)
    2 → println(„advanced“)
}


val label = when (level) {
    1 → „basic“
    2 → „advanced“
    else → unknown
}  // else wird benötigt bei when als Expression außer Enums etc.



val result = when {
    set.isEmpty() → „Empty“
    else → „Not empty“
}

val result = when (obj) {
    is String → „String“
    is Int → „Integer“
    else → „Something“
}
```

for in

for in

```
for (item in collection) println(item)

for (item in collection) {
    doSomethingWith(item)
}

for (c: Char in „hello“) println(c)


for (i in array.indices) {
    println(array[i])
}

for (i in 1..10) println(i)  // inklusive 10, gleich mehr


// Geht auf allem, was iterator() implementiert und einen Iterator zurückliefert

// Iterieren mit Index:

for ((item, index) in collection.withIndex()) {
    …
}
```

while

do...while

while, do while

```
while (line != null) {
    process(line)
    line = readLine()
}
```

```
do {
    val line = readLine()
    if (line != null) process(line)
}  while (line != null)
```

Ranges

Ranges

```
for (i in 1..10)

for (i in 10 downTo 1)

for (i in 2..10 step 2)

for (i in 10 downTo 2 step 2)

for (i in (1..10).reversed())

...
```

return
break
continue

return

```
fun makeString(): String {
  return „ABC“
}

// Vorsicht:

fun makeString2() {
  „ABC“
}


fun helper(a: A) {
  val name = a.name ?: return „DEFAULT“
  return doSomethingWith(name)
}
```

break, continue

```
for (i in 1..5) {
    if (i == 3) break else println(i)
}

// => 1 2


for (i in 1..5) {
    if (i == 3) continue else println(i)
}

// => 1 2 4 5
```

Exceptions

Keine Checked Exceptions

Immer ein Streitthema.
Moderne Sprachen mehrheitlich ohne.

Exceptions werfen mit throw

```
if (errorCondition) {
  throw IllegalArgumentException(„Nicht mit mir!“)
}

// throw ist Expression (ungleich Java)

val age: Int? = if (person.age != null) person.age else throw IllegalArgumentException(„Kein Alter“)

// Kürzer mit Elvis-Operator

val age: Int? = person.age ?: throw IllegalArgumentException(„Kein Alter“)



println(age)  // OK, Compiler weiß, dass age != null
```

Exceptions behandeln mit try - catch - finally

```
try {
    danger()
} catch (e: DangerousException) {
    …
} catch (e: RuntimeException) {
    …
} finally {
    …
}

// Auch try-catch-finally ist eine Expression

val s = try {
   "Huhu"
} catch (e: IllegalArgumentException) {
    "won‘t happen"
} catch (e: UnsupportedOperationException) {
    "won‘t happen either"
}

println(s) // Huhu
```

Übung: 07-fizz-buzz

Übung: 07a-dice

Übung: 08-number-parser

Arrays

Generell:
Array<T>, Array<Any>

Primitive Arrays:
IntArray, DoubleArray, BooleanArray...

Generics <...>
bitte vorerst noch ignorieren :)

Leider (noch) keine
Array-Literale

=> Factory-Methoden

Arrays erzeugen

```
// Allgemeine Arrays

val a: Array<Any> = arrayOf(„Kraut“, „Rüben“, 7, true)

val b: Array<Int> = arrayOf(1, 2, 3, 4)

val c = arrayOf(„a“, „b“, „c“) // => c hat Typ Array<String>

// Spezialisierte Arrays (Primitiver Inhalt, kein „Boxing“)

val ints: IntArray = intArrayOf(1, 2, 3, 4)

val notWorking: IntArray = b // => IntArray kein Subtyp von Array<Int>

val shorts: ShortArray = shortArrayOf(…)
val bytes: ByteArray = byteArrayOf(…)
val longs: LongArray = longArrayOf(…)
val floats: FloatArray = floatArrayOf(…)
val doubles: DoubleArray = doubleArrayOf(…)
val chars: CharArray = charArrayOf(…)
val bools: BooleanArray = booleanArrayOf(…)

// kein StringArray! String kein Primitivtyp

val array: Array<Object?> = arrayOfNulls(1000)
val byteArray: ByteArray = ByteArray(8192)
```

Primitive Typen ↔ Boxing

https://docs.oracle.com/javase/specs/jls/se11/html/jls-5.html#jls-5.1.7
https://docs.oracle.com/javase/1.5.0/docs/guide/language/autoboxing.html
https://effective-java.com/2015/01/autoboxing-performance/

Kotlin Primitivtypen sind JVM-Primitivtypen

außer:

wenn Nullable (kommt noch), z.B. Int?
in Array<T>, z.B. arrayOf(1,2,3)

Daher spezialisierte Arrays

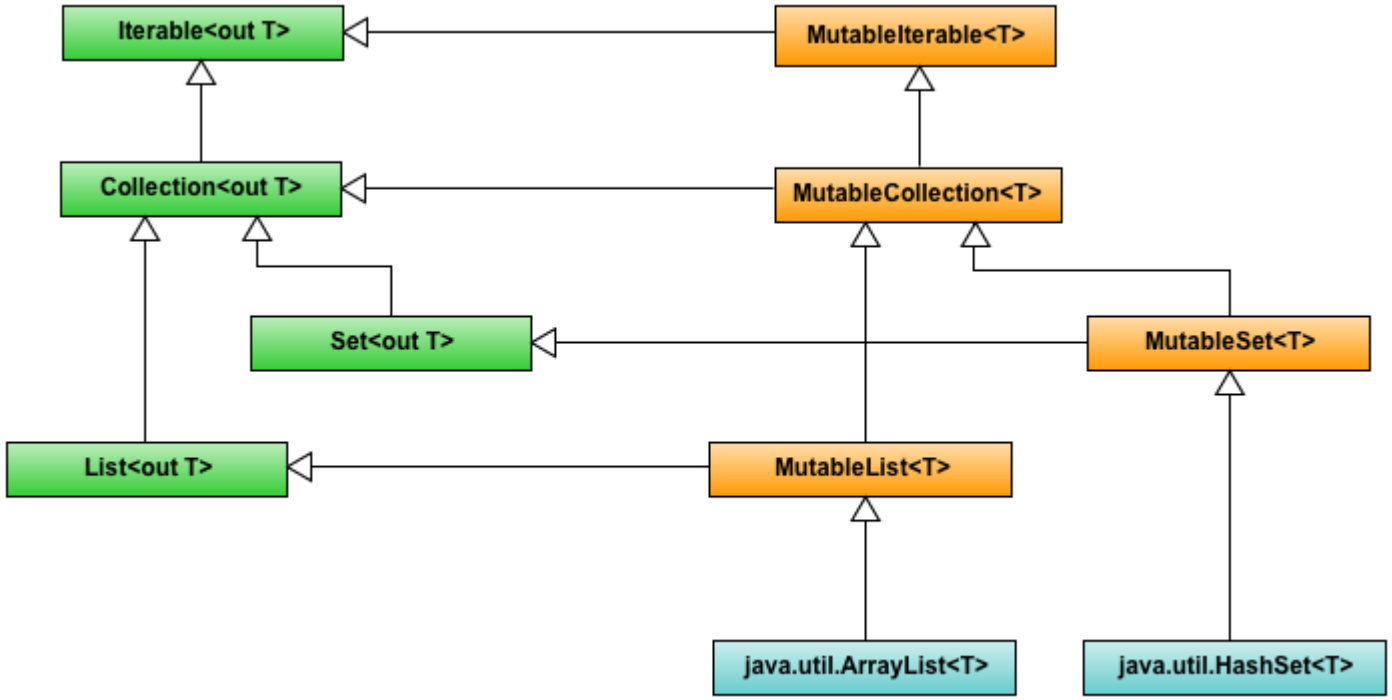
wie z.B. IntArray

Collections

Es wird strikt zwischen

Mutable / Immutable

unterschieden


Iterable<out T>
MutableIterable<T>
Collection<out T>
MutableCollection<T>
Set<out T>
MutableSet<T>
List<out T>
MutableList<T>
java.util.ArrayList<T>
java.util.HashSet<T>

Leider auch hier keine Literale

=> Factory-Methoden

(ähnlich wie bei Arrays)

List

```
// Immutable List

val list: List<Int> = listOf(1, 2, 3)

val empty = listOf<String>()
val empty2: List<String> = listOf()

val empty3 = emptyList<String>()
val empty4: List<String> = emptyList()

println(list.javaClass) // => class java.util.Arrays$ArrayList

// Mutable List

val mutableList: MutableList<Int> = mutableListOf(1, 2, 3)

val emptyMutable = mutableListOf<Int>()
val emptyMutable2: MutableList<Int> = mutableListOf()
// kein emptyMutableList()

println(mutableList.javaClass) // => class java.util.ArrayList

val arrayList = arrayListOf(1, 2, 3)
println(arrayList.javaClass) // => class java.util.ArrayList
```

Set

```
// Immutable Set

val set: Set<Int> = setOf(1, 2, 3)

val empty = setOf<String>()
val empty2: Set<String> = setOf()

val empty3 = emptySet<String>()
val empty4: Set<String> = emptySet()

println(set.javaClass) // => class java.util.LinkedHashSet
println(empty.javaClass) // => class kotlin.collections.EmptySet

// Mutable Set

val mutableSet: MutableSet<Int> = mutableSetOf(1, 2, 3)

val emptyMutable = mutableSetOf<Int>()
val emptyMutable2: MutableSet<Int> = mutableSetOf()
// kein emptyMutableSet()

println(mutableSet.javaClass) // => class java.util.LinkedHashSet

val linkedSet = linkedSetOf(1, 2, 3)
val hashSet = hashSetOf(1, 2, 3) // => class java.util.HashSet
val sortedSet = sortedSetOf(1, 2, 3) // => class java.util.TreeSet
```

Map

```
// Immutable Map

val map: Map<String, Int> = mapOf(„a“ to 1, „b“ to 2, „c“ to 3)

val empty = mapOf<String, Int>()
val empty2: Map<String, Int> = mapOf()

val empty3 = emptyMap<String, Int>()
val empty4: Map<String, Int> = emptyMap()

println(map.javaClass) // => class java.util.LinkedHashMap
println(empty.javaClass) // => class kotlin.collections.EmptyMap

// Mutable Map

val mutableMap: MutableMap<String, Int> = mutableMapOf(„a“ to 1, „b“ to 2, „c“ to 3)

val emptyMutable = mutableMapOf<String, Int>()
val emptyMutable2: MutableMap<String, Int> = mutableMapOf()
// kein emptyMutableMap()

println(mutableSet.javaClass) // => class java.util.LinkedHashMap
val linkedMap = linkedMapOf(„a“ to 1, „b“ to 2, „c“ to 3)
val hashMap = hashMapOf(„a“ to 1, „b“ to 2, „c“ to 3) // => class java.util.HashMap
val sortedMap = sortedMapOf(„a“ to 1, „b“ to 2, „c“ to 3) // => class java.util.TreeMap
```

Map - Zugriff

```
val map: Map<String, Int> = mapOf(„a“ to 1, „b“ to 2, „c“ to 3)

println map[„a“]  // 1

println map[„b“]  // 2

println map[„c“]  // 3

Alternativ:

println map.get(„a“)  // 1
```

Mutable Collections

können Immutable Collections sein,

aber nicht umgekehrt

Immutable ↔ Mutable

```
val mutable = mutableListOf(1, 2, 3)
val immutable = listOf(1, 2, 3)

val list: List<Int> = mutable // OK
mutable[0] = 7 // OK
immutable[0] = 7 // compile-time error
list[0] = 7 // compile-time error

println(list) // [7, 2, 3] Vorsicht!!!


// Unveränderliches Abbild geht nur mit Kopie

val list: List<Int> = mutable.toList()
mutable[0] = 7

println(list) // [1, 2, 3]
```

Aus Immutable Mutable machen

```
val immutable = listOf(1, 2, 3)

val mutable = immutable.toMutableList()

mutable[0] = 7 // OK

println(immutable) // [1, 2, 3]
println(mutable)   // [7, 2, 3]


// => toMutableList() erzeugt Kopie

// => sowohl toList() als auch toMutableList() erzeugt Kopie

// => alle toXXX() erzeugen Kopien
```

Objektorientierte Programmierung

Klassen und Objekte

Klasse deklarieren

Klasse deklarieren

```
class Empty

val empty = Empty()  // kein ‚new‘!



class EmptyWithFun {
    fun sayHi() {
        println(„Hello“)
    }
}

EmptyWithFun().sayHi()
```

Klasse mit Properties

```
class Count(val n: Int)

val c = Count(42)
println(c.n)

class MutableCount(var n: Int)

val mc = MutableCount(42)
println(mc.n)
mc.n += 1
println(mc.n)


class User(val email: String, val userName: String) {
    val token: String = generateToken()
}

// Properties, die nicht im Konstruktor sind, müssen initialisiert werden
```

Properties innerhalb der Klasse
müssen initialisiert werden

Beißt sich z.B. mit Dependency Injection

→ lateinit

lateinit zur nachträglichen Initialisierung

```
class User {
    lateinit var email: String
    lateinit var userName: String

    fun printName() {
        if (::email.isInitialized && ::userName.isInitialized) {
            println(„$userName ($email)“)
        }
    }
}


val user = User()

// println(user.email)
// kotlin.UninitializedPropertyAccessException: lateinit property email has not been initialized

user.printName()  // nichts

user.email = „martin.bosslet@gmail.com“
user.userName = „martin“

println(user.email) // OK
println(user.userName) // OK

user.printName()  // martin (martin.bosslet@gmail.com)
```

lateinit funktioniert nur mit var

Nicht mit val

Klasse mit anonymem Argument im Konstruktor

```
class Count(n: Int)

val c = Count(42)
println(c.n)  // compile-time error



// Wozu ist es dann gut?


class Count(n: Int) {

    init {
      println(„n ist $n“)
    }

}
```

Klasse mit Initializer

```
class Sum(a: Int, b: Int) {

    val sum: Int

    init {
      sum = a + b
    }
}


val s = Sum(2, 3)
println(s.sum)  // => 5


// Noch kürzer:

class Sum(a: Int, b: Int) {
    val sum = a + b
}
```

Klasse mit Secondary Constructor

```
class Empty() {

    constructor(s: String) : this() {
        println(s)
    }
}

Empty() // => keine Ausgabe

Empty(„Hallo“) // => Hallo


// Vorsicht

class Empty {
    constructor(s: String) {
        println(s)
    }
}

Empty()  // compile-time error
Empty(„Hello“)  // => Hello
```

Vererbung

Klassen sind per Default
„final“

→ „open“

Vererbung

```
class Empty
val e = Empty()
println(e is Any) // => true
// => Alle Klassen erben implizit von Any

open class A(val a:Int) {
    fun sayHi() {
        println(„Hello from A“)
    }
}

class B : A(42)

class C(a: Int) : A(a)

// nicht: class D(val a: Int) : A(a)  Das val a würde die Property in A verdecken

val b = B()
println(b.a)
b.sayHi()

val c = C(1)
println(c.a)
c.sayHi()
```

Polymorphismus

```
open class A {
    open fun sayHi() {
        println("A")
    }
}

open class B : A() {
    override fun sayHi() {
        println("B")
    }
}

class C : B() {
    override fun sayHi() {
        println("C")
    }
}

open class B2 : A() { // Wenn nicht open, wäre die Klasse final und demnach sayHi eh nicht erweiterbar
    final override fun sayHi() {
        println("B and no more")
    }
}
```

Polymorphismus

```
open class A {
    open fun sayHi() {
        println("A")
    }
}

open class B : A() {
    override fun sayHi() {
        super.sayHi()
        println("B")
    }
}
```

Polymorphismus bei Properties

```
open class A {
    open val a = 1
}


open class B : A() {
    override val a = 2
}


println(A().a) // => 1
println(B().a) // => 2
```

Abstrakte Klassen

```
abstract class A {

    abstract val greeting: String

    fun sayHi() {
        println(greeting)
    }
}


class B : A() {
    override val greeting = „Hi from B“
}


B().sayHi() // => Hi from B
```

Properties

Getter / Setter

Explizite Getter / Setter

braucht es nur bei dynamischen Werten

Implizite Getter / Setter

```
class A {
    val a = 17
    var b: Int? = null  // Properties müssen initialisiert werden
}


val a = A()
println(a.a)  // => 17
println(a.b)  // => null

a.b = 99

println(a.b)  // compile-time error wegen var

// schnell, aber gefährlich
println(a.b as Int)  // => 99
println(a.b!!)  // => 99

// Besser, aber umständlich
val test = a.b
if (test != null) {
  println(test)
}

// Besser ( naja :( )
a.b?.let { println(it) }
```

Explizite Getter / Setter

```
class Rectangle(val a: Int, val b: Int) {
    val area: Int
        get() {
            return a * b
        }

        // kürzer: get() = a * b
}

val r = Rectangle(5, 4)
println(r.area) // => 20
```

Explizite Getter / Setter

```
class WeirdSquare(var a: Double) {

    var area: Double = a * a  // Initialisierung notwendig
        set(value) {
            println("Value $value")
            a = Math.sqrt(value)
            field = value
        }

}

val s = WeirdSquare(4.0)
println(s.a)  // => 4.0
println(s.area)  // => 16.0

s.area = 25.0
println(s.a)  // => 5.0
println(s.area)  // => 25.0
```

Interfaces

Interface deklarieren

```
interface Closeable {

    fun close(): Unit

}

interface Person {

    val firstName: String // automatisch abstract
    val lastName: String

    val fullName: String
        get() = „$firstName $lastName“

    fun getMessage()

    fun sayHi() {
        println(„User $fullName says hi: ${getMessage()}“)
    }
}
```

Interfaces können bereits
Implementierungen enthalten

Diese können sich auf abstrakte
Funktionen/Properties beziehen

Interface implementieren

```
interface Person {
    val firstName: String
    val lastName: String

    val fullName: String
        get() = „$firstName $lastName“

    fun getMessage(): String

    fun sayHi() {
        println(„User $fullName says hi: ${getMessage()}“)
    }
}

class User(
    override val firstName:String,
    override val lastName: String,
    val email: String
) : Person {
    override fun getMessage() = „Hi from a User“
}

val u = User(„Martin“, „Boßlet“, „martin.bosslet@gmail.com“)

u.sayHi() // User Martin Boßlet says hi: Hi from a User
```

Klassen können
mehrere Interfaces
implementieren

Aufgrund von Default-Implementierung in den Interfaces können dabei Konflikte entstehen

(„Diamond Problem“)

Kotlin forciert manuelle
Auflösung des Konflikts

Diamond Problem

```
interface A {
    fun foo() { print("A") }
    fun bar()
}

interface B {
    fun foo() { print("B") }
    fun bar() { print("bar") }
}

class C : A {
    override fun bar() { print("bar") }
}

class D : A, B {
    override fun foo() {
        super<A>.foo()
        super<B>.foo()
    }

    override fun bar() {
        super.bar() // super<B>.bar()
    }
}
```

Packages deklarieren → package

Packages deklarieren

```
package workshop

package de.martinbosslet.kotlin

…
```

package:

Wenn, dann am Anfang einer Datei

Ohne: „Default“ package

Wichtigster Unterschied zu Java:

package-Deklaration
muss nicht zur Verzeichnisstruktur passen!

Import aus Packages → import

Imports

```
import workshop.Helper

import workshop.*

import workshop.Helper as WorkshopHelper
import other.Helper as OtherHelper
import third.Helper as YetAnotherHelper


…
```

Keine „static“ imports

„static imports“ in Java werden
in Kotlin auch über import realisiert

Default Imports

kotlin.*
kotlin.annotation.*
kotlin.collections.*
kotlin.comparisons.*
kotlin.io.*
kotlin.ranges.*
kotlin.sequences.*
kotlin.text.*

(http://kotlinlang.org/api/latest/jvm/stdlib/index.html)

Daher kommen

listOf
setOf
mapOf
arrayOf
& Co.

Im Java-Kontext zusätzlich:

java.lang.*
kotlin.jvm.*

(http://kotlinlang.org/api/latest/jvm/stdlib/kotlin.jvm/index.html)

Visibility Modifiers

public
private
protected
internal

Ohne Modifier:

public

Zwei Ebenen:

1. „Top-Level“

2. Innerhalb von Klassen/Interfaces

Top-Level

Top-level visibility modifiers

```
package a

fun f01() {…}  // public

public fun f02 {…}  // public überflüssig

private fun f03 {…}

internal fun f04 {…}

// protected gibt es hier nicht
```

public

Überall sichtbar, ggf. via Import

private

Nur innerhalb der gleichen Datei sichtbar

internal

Innerhalb dieses „Moduls“ sichtbar

Modul

- Gradle Source Set
- Maven-Projekt
- Files innerhalb eines <kotlinc>-Ant-Tasks
- IntelliJ IDEA Modul

Intention von internal:

Nur innerhalb des „Projekts“ sichtbar
Nicht von außerhalb
→ sehr interessant für Libraries

Javas „package private“ ist unzureichend

→ Libraries müssen API als public deklarieren,
die eigentlich nicht sichtbar sein sollte

→ internal

Top-level modifier anwendbar für:

- Funktionen
- Properties
- Klassen
- Interfaces
- Objects

Klassen-/Interface-Level

Gilt für:

„Member“ der Klasse / des Interface

→ Funktionen, Properties, innere Klassen,
Interfaces und Objects

public

Alle, die die Klasse sehen können,
sehen auch dies

private

Nur die Klasse sieht dies

internal

Alle innerhalb des Moduls,
die die Klasse sehen,
sehen auch dies

protected

Die Klasse
inklusive all ihrer Unterklassen
sehen dies

Übung: 09-tennis

Funktionale Programmierung

Lambdas

Beispiel für einen Lambda-Ausdruck

```
val lambda = { i: Int → i + 1 }

println(lambda.javaClass)  // => class workshop.MainKt$main$lambda$1

// Werden ausgeführt mittels #invoke

println(lambda.invoke(1))  // => 2

// Kann alternativ direkt wie eine Funktion aufgerufen werden:

println(lambda(1)) // => 2

// Unmittelbare Invokation
println({ a: Int, b: Int → a + b }(2, 3)) // => 5
```

Allgemeiner Aufbau:

{ arg1, ..., argn → <body> }

Rückgabetyp des Lambda-Ausdrucks
ist der Wert der
zuletzt ausgeführten Expression

(kann auch Unit sein)

Typ-Bezeichner für einen Lambda-Ausdruck

```
val lambda: (Int) → Int = { i → i + 1 }

val sum: (Int, Int) → Int = { a, b → a + b }

// val sum = { a: Int, b: Int → a + b }
```

Allgemeiner Aufbau:

(T1, ..., Tn) → TRückgabewert

Lambdas erlauben nicht die
explizite Angabe des Rückgabetyps:

```
{ a: Int, b: Int → a + b} : Int
```

Anonyme Funktionen
als explizitere Alternative

Anonyme Funktionen

```
val increment = fun (i: Int): Int = i + 1

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

println(increment(1)) // => 2

println(sum(2, 3)) // => 5
```

Ganz nett, aber wozu?

Closures

Closures

```
// Lambda, dass neues Lambda zurückgibt
// val createMultiplier: (Int) → [ (Int) → Int ]
val createMultiplier = { multiplyBy: Int ->
    { arg: Int -> arg * multiplyBy }
}

// doubler „merkt“ sich multiplyBy als 2
val doubler = createMultiplier(2)
println(doubler(5)) // => 10

// tripler „merkt“ sich multiplyBy als 3
val tripler = createMultiplier(3)
println(tripler(9)) // => 27
```

Closures haben Zugriff auf Variablen in ihrem Aufrufkontext

```
var sum = 0

listOf(1, 2, 3).forEach { i → sum += i }

println(sum) // => 6
```

Higher-Order Functions

Funktionen mit Lambdas parametrisieren

Funktionen parametrisieren mit Lambdas

```
fun concatenateLowerCase(strings: List<String>): String {
    return strings.joinToString(„ “).toLowerCase()
}

fun concatenateUpperCase(strings: List<String>): String {
    return strings.joinToString(„ “).toUpperCase()
}


// In einer Funktion:

fun concatenateAndFormat(strings: List<String>, toUpperCase: Boolean): String {
    val concatenated = strings.joinToString(„ “)
    if (toUpperCase) concatenated.toUpperCase() else concatenated.toLowerCase()
}
```

Hässlich!

Außerdem: Was, wenn 3 Optionen?

- wie gehabt
- lowercase
- uppercase

Funktionen parametrisieren mit Lambdas

```
fun concatenateAndFormat(strings: List<String>, formatter: (String) → String): String {
    return formatter(strings.joinToString(„ “))
}

val strings = listOf(„Martin“, „Nadine“, „Anna“)

println(concatenateAndFormat(strings) { s → s.toLowerCase() })
// => martin nadine anna

println(concatenateAndFormat(strings) { s → s.toUpperCase() })
// => MARTIN NADINE ANNA

println(concatenateAndFormat(strings) { s → s })
// => Martin Nadine Anna
```

Konvention:

Wenn letztes Funktionsargument
ein Lambda,

darf dieses außerhalb der Klammern stehen

„it“ bei Lambdas

it

```
fun concatenateAndFormat(strings: List<String>, formatter: (String) → String): String {
    return formatter(strings.joinToString(„ “))
}

val strings = listOf(„Martin“, „Nadine“, „Anna“)

//println(concatenateAndFormat(strings) { s → s.toLowerCase() })
println(concatenateAndFormat(strings) { it.toLowerCase() })

//println(concatenateAndFormat(strings) { s → s.toUpperCase() })
println(concatenateAndFormat(strings) { it.toUpperCase() })

//println(concatenateAndFormat(strings) { s → s })
println(concatenateAndFormat(strings) { it })
```

Lambdas erleichtern
Umgang mit Collections

UNGEMEIN

„Do what I mean

instead of

how I say“

Beispiel:

Alle ungeraden Zahlen
von 1 bis 100
quadrieren,
in umgekehrter Reihenfolge
per Komma getrennt als String ausgeben

Collections & Lambdas

```
println(
    (1..100)                    // Zahlen 1-100
        .filter { it % 2 == 1 }  // ungerade
        .map { it * it }         // quadrieren
        .reversed()              // in umgekehrter Reihenfolge
        .joinToString(„,“)       // zu einem String kombinieren getrennt durch ,,‘
)
```

Die wichtigsten Operationen bei Collections

map

Jedes einzelne Element
nach dem gleichen Schema
transformieren

map

```
val squaresOfOneToTen = (1..10).map { i -> i * i }

println(squaresOfOneToTen)

//=> [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

filter

Elemente basierend
auf einer Ja/Nein-Aussage
aus der Collection elimieren

filter

```
val evenNumbers = (1..10).filter { i → i % 2 == 0 }

println(evenNumbers)

//=> [2, 4, 6, 8, 10]
```

fold / reduce

Alle Elemente einer Collection
nach dem gleichen Schema
zu einem Einzelergebnis
zusammenfassen

fold

```
val sum = (1..10).fold(0) { sum, i → sum + i }

oder

val sum = (1..10).reduce { sum, i → sum + i }

println(sum)


//=> 55


// 0 ist der Startwert von sum
// Bei reduce wird als Startwert der erste Wert der Collection genommen
// sum ist der sogenannte Akkumulator
// sum wird in jedem Durchlauf aktualisiert mit dem nächsten Rückgabewert
// Daher der Name: Die Collection wird in sum „reingefaltet“
```

Viele weitere eingebaute Operationen:

https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-collection/index.html#kotlin.collections.Collection

Lambdas mittels Funktionsreferenzen

Lambdas mittels Funktionsreferenzen

```
val elegantSum = (1..10).reduce(Int::plus)

println(elegantSum)
//=> 55


listOf(1, 2, 3).forEach(::println)  // => 1 2 3


val factorial = (1..5).reduce(Int::times)
println (factorial) // => 120


fun joiner(a: String, b: String): String = a + b


val abc = listOf(„a“, „b“, „c“).reduce(::joiner)
println(abc) // => abc
```

Funktionsreferenzen

dank Kotlin Reflection

https://kotlinlang.org/docs/reference/reflection.html

Funktionale Programmierung mit Kotlin

Currying, Monads & Co.
→ Arrow

https://arrow-kt.io/

Übung: 10-lambdas

Fortgeschrittene Konzepte von Kotlin

Null Safety

Ziel von Kotlin:

Bye, bye NullPointerException!

Dank Compile-Time Checks

„Normalen“ Variablen

kann null nicht zugewiesen werden

null nicht erlaubt

```
val a: Int

a = null  // Compile-time error


var s: String = null // Compile-time error
```

Was tun, wenn aber
null explizit gebraucht wird?

→ Nullable Types

Nullable Types

```
val s: String?

s = null

println(s)  // => null, keine NPE
```

Was bringt dies?

Zugriff auf Nullable Types

```
val safe: String = „Nicht null“

println(s.length)  // OK, da s garantiert ungleich null


val danger: String? = null

println(s.length) // Compile-time error
```

Wie greife ich aber auf
solche Variablen zu?

Expliziter Null-Check

```
val danger: String? = …

if (danger != null) println(danger.length)


// Der Compiler weiß, dass der Zugriff erlaubt ist
```

Sicherer Zugriff mittels ?

Expliziter Null-Check

```
val danger: String? = …

val len = danger?.length

// len ist vom Typ Int?
// len kann selbst wiederum null sein

if (danger != null) {
    val len2 = danger.length // len2 ist vom Typ Int
}

deeply?.nested?.property?.access
```

Vorsicht bei var

```
class A(var n: Int?)

val a = A(null)
println(a.n)  // OK
a.n = 42
println(a.n)  // Compile-time error

if (a.n != null) {
  println(a.n) // Immer noch compile-time error?!
}

// Ein anderer Thread könnte a.n geändert haben, daher Smart Cast nicht möglich

val n = a.n
if (n != null) println(n) // OK
```

let zur konditionellen Ausführung

let

```
val list = listOf(„1“, null, „2“, null)  // List<String?>

for (s in list) {
  s?.let { println(it) }  // Lambda nur wenn s ungleich null
}

// => 1 2
```

Elvis-Operator

?:

Elvis-Operator ?:

```
val danger: String? = …

val len = if (danger != null) danger.length else -1

// Kürzer:

val len = danger?.length ?: -1
```

Casts

Explizit oder „Smart“

Explizite Casts mit „as“

```
val unknownType: Any = …

val asString = unknownType as String

// Kann Exception werfen, falls Cast nicht möglich

//Vorsicht bei null

val nullable: Any? = null

val nullAsString: String = nullable as String

// => Exception weil val nicht nullable!

// => korrekt:
val nullAsString: String? = nullable as String
```

Explizite Casts ohne Exception: „Safe“ Casts

```
val unknownType: Any = …

val possibleString: String? = unknownType as? String

// Falls unknownType kein String → possibleString ist null
```

Automatische Casts: „Smart“ Casts

```
val unknownType: Any = …

if (unknownType is String) {
  println(unknownType.length)
}

// Compiler castet unknownType automatisch zu String
```

Automatische Casts: „Smart“ Casts

```
if (x !is String) return

print(x.length) // x zu String gecastet


// rechte Seite von && und ||

if (x !is String || x.length == 0) return


if (x is String && x.length > 0) return
```

Gleichheit

Kotlin unterscheidet:

- Exakt das gleiche Object
- Semantische Gleichheit („Equals“)

Kotlin unterscheidet:

- Referenzielle Gleichheit
(Exakt das gleiche Object)

- Strukturelle Gleichheit
(„Equals“)

Referenzielle Gleichheit

===

und

!==

===

```
val s = „ABC“
val t = „ABC“
val u = s
val v = java.lang.String(„ABC“)


println(s === s) // => true

println(s === t) // => true (Wegen String Interning)

println(s === u) // => true

println(s === v) // => false
```

Strukturelle Gleichheit

==

und

!=

==

```
val s = „ABC“
val t = „ABC“
val u = java.lang.String(„ABC“)

println(s == t) // => true

println(s == v) // => true


Vorsicht bei Klassen:

class A(n: Int)

val a = A(1)
val b = A(1)

println (a == b) // => false

=> Dafür gibt es Data Classes
```

Strukturelle Gleichheit

implementieren

== implementieren mit equals(other: Any?): Boolean

```
class A(n: Int) {
    override fun equals(other: Any?): Boolean {
      return other is A &&
          this.n == other.n // nicht „other?.n“ weil „is A“ impliziert non-null
    }
}

val a = A(1)
val b = A(1)

println (a == b) // => true
```

Default- und Named-Argumente

bei Funktionen

Viele Argumente werden schnell unübersichtlich

```
fun complex(a: Int?, b: String?, c:Int?, d: Boolean?): String {
    val a1 = if (a == null) 42 else a
    val b1 = if (b == null) "Default Message" else b
    val c1 = c ?: 10  // Elvis Operator
    val d1 = d ?: true

    val result = a1 * c1
    val message = "$b1: $result"

    return if (d1) message.toUpperCase() else message
}

println(complex(null, null, null, null))

println(complex(8, „Ergebnis“, 100, true))
```

Sauberer: Default Arguments

```
fun complex(a: Int = 42, b: String = „Default Message“, c: Int = 10, d: Boolean = true): String {
    val result = a * c
    val message = "$b: $result"

    return if (d) message.toUpperCase() else message
}

println(complex())

println(complex(8, „Ergebnis“, 100, true))
```

Noch sauberer: Named Arguments

```
fun complex(a: Int = 42, b: String = „Default Message“, c: Int = 10, d: Boolean = true): String {
    val result = a * c
    val message = "$b: $result"

    return if (d) message.toUpperCase() else message
}


println(complex(a = 8, d = false))
```

Defaults müssen nicht mehr
explizit als „null“
übergeben werden

Dank Named Arguments
kann man gezielt Defaults
wählen / überschreiben

Pattern: Mandatory vorne, Optional hinten

```
fun complexSum(a: Int, b: Int, msg: String = „Default Message“, toUpper: Boolean = true): String {
    val result = a * b
    val message = "$msg: $result"

    return if (toUpper) message.toUpperCase() else message
}


println(complexSum(7, 8)) // statt complexSum(a = 7, b = 8)

println(complexSum(7, 8, "Summe"))
println(complexSum(7, 8, "Summe", false))
println(complexSum(7, 8, toUpper = false))
```

Default-Argumente dürfen sich auf vorherige(!) Argumente beziehen

(sogar beliebige Expressions)

Default Arguments: Expressions

```
fun copyBytes(ary: ByteArray, length: Int = ary.size) { …  }


fun complex(a: Int, b: Int, c = evaluate(a, b)) { … }


fun nope(a: evaluate(b), b: evaluate(c), c: Boolean = true) { … }
// immer nur vorhergehende Argumente
```

varargs-Argumente

bei Funktionen

varargs

```
fun sumOfInts(vararg ns: Int) {
    …
    // ns hat hier Typ Array<Int>
    // genauer: Array<out Int>
    // gleich: Generics
}


sumOfInts(1, 2, 3) // => 6
sumOfInts(5, 5, 5, 5) // => 20
```

Object Expressions & Declarations

Häufiger Fall: Singleton

„Helper“

Container für Methoden

Mehrere Instanzen ohne Sinn

Singletons mit Object Declarations

```
object Helper {
    fun snakeToCamelCase(s: String): String { … }
    fun camelToSnakeCase(s: String): String { … }
    …
}


Helper.snakeToCamelCase(„a_b_c“)

Helper.camelToSnakeCase(„camelCase“)



// Hier wären auch Extension Methods sinnvoll → später!
```

Ad-Hoc-Objekte mit Object Expressions

(ähnlich zu Anonymous Class in Java)

Object Expressions

```
val point = object {
    val x = 2
    val y = 3
}


println(point.x) // => 2

println(point.y) // => 3




val r = object : Runnable {
    override fun run() {
        println(„run is fun“)
    }
}

r.run()
```

Single Abstract Method (SAM) Conversion / @FunctionalInterface

https://docs.oracle.com/javase/specs/jls/se11/html/jls-9.html#jls-9.8

Interfaces mit nur einer Methode
und @FunctionalInterface-Annotation

können per Lambda implementiert werden

SAM Conversion

```
val runner: Runnable = Runnable { println(„Running!“) }

runner.run()


val list = mutableListOf(1, 2, 3)
val descender: Comparator<Int> = Comparator { a, b → b - a }
Collections.sort(list, descender)

println(list)  // => [3, 2, 1]
```

Companion Object

Der Ersatz für „static“

Companion Object

```
class Person(val name: String) {

    companion object {
        fun create() = Person(„anonymous“)
    }

}


val p = Person.create()
println(p.name) // => anonymous
```

Companion Objects
sind vollwertige Objekte,
die z.B. auch Interfaces implementieren
können

Sie können aber auch
auf JVM-Ebene als echte
„static“-Bestandteile generiert werden

siehe @JvmStatic

Kotlin-Äquivalent zu static import

```
A.kt:

package a

object A {
  fun test() {
    println(„Hello from A“)
  }
}
-------------------------------------------------
B.kt:

package b

import a.A.test as aTest

fun main() {
  aTest() // => Hello from A
}
```

Compile-time Constants

const val

Compile-Time Constants

```
const val ENVIRONMENT = „development“

class Circle {
    companion object {
        const val PI = 3.14
    }
}

object Constants {
    const val TARGET_PATH = „/opt/target“
}


println(ENVIRONMENT)
println(Circle.PI)
println(Constants.TARGET_PATH)
```

Vorteil:

Können auch in Annotations referenziert werden

Intermezzo:

Beispiel: 11-DSL

Data Classes

Der Traum für „DTOs“

Data Class

```
data class Person(val firstName: String, val lastName: String)


// entspricht in Java:

public class Person {
    private final String firstName;
    private final String lastName;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }
}
```

Wo ist der Unterschied zu

class Person(…)

(also ohne „data“) ?

Vorteile von Data Classes

```
data class Person(val firstName: String, val lastName: String)

class Person2(val firstName: String, val lastName: String)

val p = Person(„Hans“, „Schmidt“)
val q = Person(„Hans“, „Schmidt“)

val p2 = Person2(„Hans“, „Schmidt“)
val q2 = Person2(„Hans“, „Schmidt“)

println(p == q)  // => true

println(p2 == q2) // => false
```

data class

- hashCode / equals
- toString
- copy

z.B. val daughter = martin.copy(firstName = „Anna“)

Enum Classes

Enum Classes

```
enum class Direction {
    NORTH, EAST, SOUTH, WEST
}


enum class Color(val rgb: Int) {
    RED(0xFF0000),
    GREEN(0x00FF00),
    BLUE(0x0000FF)
}

enum class Color {
    Red {
        override fun hex() = 0xFF0000
    },
    Green {
        override fun hex() = 0x00FF00
    },
    Blue {
        override fun hex() = 0x0000FF
    }

    abstract fun hex(): Int
}
```

Decorator

mittels Delegation

Delegation

```
class NoisyList<out T>(private val list: List<T>) : List<T> by list {

    override fun get(index: Int): T {
        return list[index].also { println("You are getting element $it from index $index") }
    }
}


val noisy = NoisyList(listOf("a", "b", "c"))

noisy[0]
noisy[2]

println(noisy.size)

//=>

You are getting element a from index 0
You are getting element c from index 2
3
```

Extension Functions

Wie oft wünscht man sich,
eine Klasse hätte eine
zusätzliche Funktion…

Ausweg:

„Util-Klassen“

Util-Klassen verschändeln den Code

```
val nr = …

if (MathUtil.divisibleBy(nr, 5) && MathUtil.divisibleBy(nr, 3)) {
    println(„Divisible by 15!“)
}
```

Extension Functions

```
fun Int.divisibleBy(divisor: Int): Boolean {
    return this % divisor == 0
}


// kürzer:


fun Int.divisibleBy(divisor: Int) = this % divisor == 0
```

Extension Methods

```
// geht auch mit Generics!


fun <T> MutableList<T>.swap(index1: Int, index2: Int) {
    val tmp = this[index1]
    this[index1] = this[index2]
    this[index2] = tmp
}
```

this:

Der „Receiver Type“, also die Instanz der Klasse, für die die Extension deklariert wird

Scope von Extensions

kann beschränkt werden!

Extension Methods - Scope

```
private fun Int.sayHiFromTopLevel() = println(„Top-level hi!“)

fun a() {
    fun Int.sayHi() = println(„Int says hi“)
    7.sayHi()
    42.sayHiFromTopLevel()
}


fun b() {
    42.sayHiFromTopLevel()  // OK
    7.sayHi()  // Compile-time error
}
```

Generics

Container<T>

```
class Container<T>(val value: T)



val a = Container(5)

val b = Container(„Hello“)

val c = Container(true)
```

Wo ist der Vorteil
gegenüber z.B. Any?

Container mit Any

```
class Container<T>(val value: T)

class AnyContainer(val value: Any)




val a = Container(„Hello“)
val b = AnyContainer(„Hello“)

val v1 = a.value
println(v1.length) // 5

val v2 = b.value
println(v2.length) // compile-time error da nur Any
```

Generic-Typen eingrenzen

Generic Upper Bounds

```
class NumberContainer<T : Number>(val value: T)

val a = NumberContainer(7)

val b = NumberContainer(3.0)

val c = Container(„Hello“) // compile-time error
```

Generische Funktionen

Index Finder

```
fun <T, C : Iterable<T>> indexOf(hayStack: C, needle: T): Int? {
    for ((index, value) in hayStack.withIndex()) {
        if (value == needle) return index
    }
    return null
}


val list = listOf(1, 2, 3)
val list2: List<String> = listOf("a", "b", "c")

println(indexOf(list, 1)) // => 0
println(indexOf(list2, "c")) // => 2

println(indexOf(list, 4)) // => null


// Warum geht das hier?!  true ist Boolean, aber list2 ist List<String>
println(indexOf(list2, true)) // => null
```

Invarianz

Kovarianz

Kontravarianz

Invarianz

Keine Relation der Generic-Typen untereinander

Kotlin-Arrays sind invariant

Invarianz

```
val a: Array<String> = arrayOf(„a“, „b“, „c“)

val b: Array<Any> = a  // compile-time error


auch nicht

val a: Array<Any> = arrayOf(„a“, 2, true)

val b: Array<String> = a  // compile-time error, macht intuitiv Sinn
```

Kovarianz

Gegeben Thing<T>.

Thing ist kovariant in T, falls:

Wenn A : B gilt, dann gilt
Thing<A> : Thing<B>

Intuitiv bei Arrays

String : Any,

also Array<String> : Array<Any>

ODER?

Java – Kovariante Arrays

```
String[] strings = new String[] { „a“, „b“, „c“ };

Object[] container = strings;

container[0] = 42; // RuntimeException!
```

Es würde gutgehen,

solange nur Dinge aus
dem Array heraus geholt werden!

Beispiel:

Immutable List

Es können nur Dinge entnommen werden,

aber keine in die Liste
hinzugefügt werden

Kovariante List<T>

```
val listOfStrings: List<String> = listOf(„a“, „b“, „c“)

val listOfAnys: List<Any> = listOfStrings

val first: Any = listOfAnys[0]  // OK, String : Any


// Probleme gäbe es nur beim Hinzufügen:

listOfAnys.add(true)  // geht aber nicht, immutable!
```

In der Tat erlaubt
Kotlin diese Zuweisung.

Wie geht das?

→ out

out

```
interface List<out E> {

…

}

Alle Methoden von List haben E nur auf der „out“-Seite

Also: Nur als Rückgabewert.

Niemals auf der „in“-Seite, also als Funktionsparameter
```

out

```
interface Producer<out T> {

    fun produceT(): T  // OK

    fun consumeT(arg: T): Unit // compile-time error

}
```

Kontravarianz

Gegeben Thing<T>.

Thing ist kontravariant in T, falls:

Wenn A : B gilt, dann gilt
Thing<B> : Thing<A>

in

```
interface Comparable<in T> {
    fun compareTo(other: T)
}

class C : Comparable<Any> {
    override fun compareTo(other: Any): Int {
        return this.compareTo(other)
    }
}


val c: Comparable<String> = C()
```

Intention: Wenn es alles vergleichen kann, kann es erst recht Strings vergleichen

Alle Methoden von Comparable haben T nur auf der „in“-Seite

Also: Nur als Funktionsparameter.

Niemals auf der „out“-Seite, also als Rückgabewert

Daher müssen Arrays
invariant sein!

Sie haben T sowohl

auf „out“-Seite (get)

als auch auf „in“-Seite (set)

„Producer“

Kovarianz ist oft nützlich
bei Klassen, die den generischen
Typ nur erzeugen

Als Rückgabewert

„Consumer“

Kontravarianz ist oft nützlich
bei Klassen, die den generischen
Typ nur entgegennehmen

Als Funktionsparameter

Mantra:

Consumer in,

Producer out

Zurück zu unserem Problem

Index Finder

```
fun <T, C : Iterable<T>> indexOf(hayStack: C, needle: T): Int? {
    for ((index, value) in hayStack.withIndex()) {
        if (value == needle) return index
    }
    return null
}


// Warum geht das hier?!  true ist Boolean, aber list2 ist List<String>
println(indexOf(list2, true)) // => null

Iterable ist Iterable<out T>


Daher wird indexOf(list2, true) so aufgelöst: T: Any, C: List<String>
Weil Iterable kovariant ist, ist List<String> : Iterable<Any>
```

Index Finder

```
fun <T : Comparable<T>, C : Iterable<T>> indexOf(hayStack: C, needle: T): Int? {
    for ((index, value) in hayStack.withIndex()) {
        if (value == needle) return index
    }
    return null
}

val list = listOf(„a“, „b“, „c“)

println(indexOf(list, true))

// => List<String> is not a subtype of Iterable<Boolean>


Funktioniert, weil Comparable<in T>, also kontravariant ist!
```

Lazy Processing mit Sequences

Operationen auf
Collections sind „final“ (terminal)

D.h. jede Operation
erzeugt unmittelbar
eine neue Collection

Ineffizient

bei vielen Zwischenschritten

Sequences

Lazy Evaluation

Operationen

werden spätestmöglich
realisiert

Unmittelbare Realisierung bei Collections

```
println("Vor Erzeugung")
val listPlus3 = listOf(1, 2, 3)
        .map { println("Map 1: $it"); it + 1 }
        .map { println("Map 2: $it"); it + 1 }
        .map { println("Map 3: $it"); it + 1 }

println("Vor Ausgabe Liste")
println(listPlus3)

println("Vor Erzeugung joined")
val joined = listPlus3.joinToString(",")

println("Vor Ausgabe joined")
println(joined)

// =>
Vor Erzeugung
Map 1: 1
Map 1: 2
Map 1: 3
Map 2: 2
Map 2: 3
Map 2: 4
Map 3: 3
Map 3: 4
Map 3: 5
Vor Ausgabe Liste
[4, 5, 6]
Vor Erzeugung joined
Vor Ausgabe joined
4,5,6
```

Realisierung „on demand“ bei Sequences

```
println("Vor Erzeugung")
val listPlus3 = listOf(1, 2, 3)
        .asSequence()
        .map { println("Map 1: $it"); it + 1 }
        .map { println("Map 2: $it"); it + 1 }
        .map { println("Map 3: $it"); it + 1 }

println("Vor Ausgabe Liste")
println(listPlus3)

println("Vor Erzeugung joined")
val joined = listPlus3.joinToString(",")

println("Vor Ausgabe joined")
println(joined)

// =>
Vor Erzeugung
Vor Ausgabe Liste
kotlin.sequences.TransformingSequence@68837a77
Vor Erzeugung joined
Map 1: 1
Map 2: 2
Map 3: 3
Map 1: 2
Map 2: 3
Map 3: 4
Map 1: 3
Map 2: 4
Map 3: 5
Vor Ausgabe joined
4,5,6
```

Nicht-finale Operationen werden Stück für Stück realisiert

Schrittweise Realisierung

```
listOf(1, 2, 3)
    .filter { println("filter: $it"); true }
    .forEach { println("forEach: $it") }

//=>
filter: 1
filter: 2
filter: 3
forEach: 1
forEach: 2
forEach: 3


sequenceOf(1, 2, 3)
    .filter { println("filter: $it"); true }
    .forEach { println("forEach: $it") }

//=>
filter: 1
forEach: 1
filter: 2
forEach: 2
filter: 3
forEach: 3
```

Sequences sind bei
mehreren Operationen
in den meisten Fällen

performanter & speichereffizienter

Sequences können potenziell unendlich sein

Infinite Sequences

```
val naturalNumbers = generateSequence { it +1 }

println(naturalNumbers.take(10).joinToString(„,“))

// => 1,2,3,4,5,6,7,8,9,10



val primeNumbers = naturalNumbers
    .filter { it > 1 && (2 until it).none { i → it % i == 0 } }

println(primeNumbers.take(10).joinToString(„,“))

// => 2,3,5,7,11,13,17,19,23,29
```

Coroutines

„Light-weight concurrency“

build.gradle

```
plugins {
    id 'org.jetbrains.kotlin.jvm' version '1.3.31'
    id 'application'
    …
}

…

dependencies {
    implementation('org.jetbrains.kotlin:kotlin-stdlib-jdk8')
    implementation('org.jetbrains.kotlinx:kotlinx-coroutines-core:1.2.1')
    …
}

…
```

Beispiel: 12-coroutines

II. Kotlin im Alltag

Kotlin & Spring Boot

https://spring.io/projects/spring-boot

Spring Boot mit Kotlin: build.gradle

```
plugins {
      id 'org.jetbrains.kotlin.plugin.jpa' version '1.3.20'
      id 'org.springframework.boot' version '2.1.3.RELEASE'
      id 'org.jetbrains.kotlin.jvm' version '1.3.20'
      id 'org.jetbrains.kotlin.plugin.spring' version '1.3.20'
      id 'org.jetbrains.kotlin.plugin.allopen' version '1.3.20'
}

apply plugin: 'kotlin-jpa'
apply plugin: 'kotlin-allopen'
apply plugin: 'io.spring.dependency-management'

sourceCompatibility = '1.8'

test {
      useJUnitPlatform()
}

allOpen {
      annotation("javax.persistence.Entity")
}
```

Spring Boot mit Kotlin: build.gradle Dependencies

```
dependencies {
      implementation 'org.springframework.boot:spring-boot-devtools'
      implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
      implementation 'org.springframework.boot:spring-boot-starter-mustache'
      implementation 'org.springframework.boot:spring-boot-starter-web'

      implementation 'com.fasterxml.jackson.module:jackson-module-kotlin'

      implementation 'org.jetbrains.kotlin:kotlin-reflect'
      implementation 'org.jetbrains.kotlin:kotlin-stdlib-jdk8'

      runtimeOnly 'com.h2database:h2'

      testImplementation('org.springframework.boot:spring-boot-starter-test') {
            exclude module: 'junit'
      }
      testImplementation('org.junit.jupiter:junit-jupiter-api')
      testRuntimeOnly('org.junit.jupiter:junit-jupiter-engine')
}
```

Beispiel: 13-spring-boot

Testen

JUnit 5

https://junit.org/junit5/

JUnit 5 Dependencies in build.gradle

```
dependencies {
    implementation('org.jetbrains.kotlin:kotlin-stdlib-jdk8')
    implementation('org.jetbrains.kotlin:kotlin-reflect')

    testCompile('org.junit.jupiter:junit-jupiter-api:5.4.0')
    testRuntime('org.junit.jupiter:junit-jupiter-engine:5.4.0')
}
```

@TestInstance(Lifecycle.PER_CLASS)

nutzen, um static zu vermeiden

Vergisst man schnell mal

Nervt auf Dauer

LifeCycle.PER_CLASS auf Dauer

In src/test/resources/junit-platform.properties:

```
junit.jupiter.testinstance.lifecycle.default = per_class
```

Beispiel: 14-junit5

Kotlin Test

https://github.com/kotlintest/kotlintest

Kotlin Test Dependencies in build.gradle

```
test {
  useJUnitPlatform()
}

dependencies {
  testImplementation 'io.kotlintest:kotlintest-runner-junit5:3.2.1'
}
```

Vorteile

Mehr Freiheit beim Beschreiben
Syntax näher an menschlicher Sprache
Vielzahl an Matchern
Property-Based Testing
Data-Driven Testing

Beispiel: 15-kotlin-test

Dokumentation

JavaDoc / KDoc

mit Dokka

https://github.com/Kotlin/dokka/blob/master/README.md
https://kotlinlang.org/docs/reference/kotlin-doc.html

Ähnlich zu JavaDocs

Etwas vereinfacht

Formate: HTML, Markdown

Dokka in build.gradle

```
plugins {
    id 'org.jetbrains.kotlin.jvm' version '1.3.20'
    id 'org.jetbrains.dokka' version '0.9.17'
}

dokka {
    outputFormat = 'html'
    outputDirectory = "$buildDir/javadoc"
}
```

Beispiel: 16-dokka

Coding Style

Offizieller Guide

https://kotlinlang.org/docs/reference/coding-conventions.html

Kann über IntelliJ
forciert werden

File → Settings → Editor
→ Code Style → Kotlin

Scheme Default
Set From...
Predefined Style...
Kotlin Style Guide

Dann Reformat Code
wo gewünscht

Via Gradle:

kotlin.code.style=official
in
gradle.properties

REST APIs mit Ktor

https://ktor.io/

Ktor Dependencies in build.gradle

```
dependencies {
    compile("io.ktor:ktor-server-core:$ktorVersion")
    compile("io.ktor:ktor-server-netty:$ktorVersion")
    compile("io.ktor:ktor-jackson:$ktorVersion")
    compile('ch.qos.logback:logback-classic:1.2.3')
}

application {
    mainClassName = "io.ktor.server.netty.EngineMain"
}
```

Asynchrone Clients & Server mittels DSL

ohne viel Zeremonie

Asynchrone Clients & Server mittels DSL

ohne viel Zeremonie

Ähnlich wie
Sinatra,
Spark,
Scalatra,
Compojure

Beispiel: 17-ktor

Beispiel: 18-android

The End

Ich danke Ihnen für Ihre Aufmerksamkeit
und wünsche viel Spaß mit Kotlin!

Martin Boßlet

martin.bosslet@gmail.com
martinbosslet.de
https://twitter.com/_emboss_
https://github.com/emboss
https://www.linkedin.com/in/martin-bosslet

www.gfu.net