



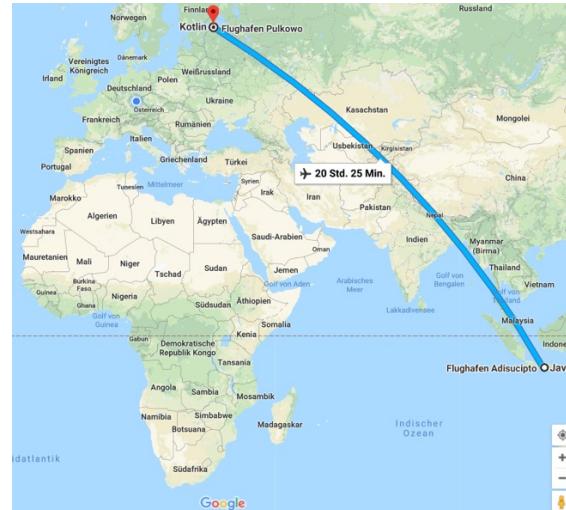
Herbstcampus

Von Java nach Kotlin

Was ist neu, was ist anders?

Werner Eberling

E-Mail: werner.eberling@mathema.de
Twitter: @Wer_Eb





Wer steht hier?



Werner Eberling

Principal Consultant / Autor

Email: werner.eberling@mathema.de

Twitter: [@Wer_Eb](https://twitter.com/Wer_Eb)



Was haben wir vor?

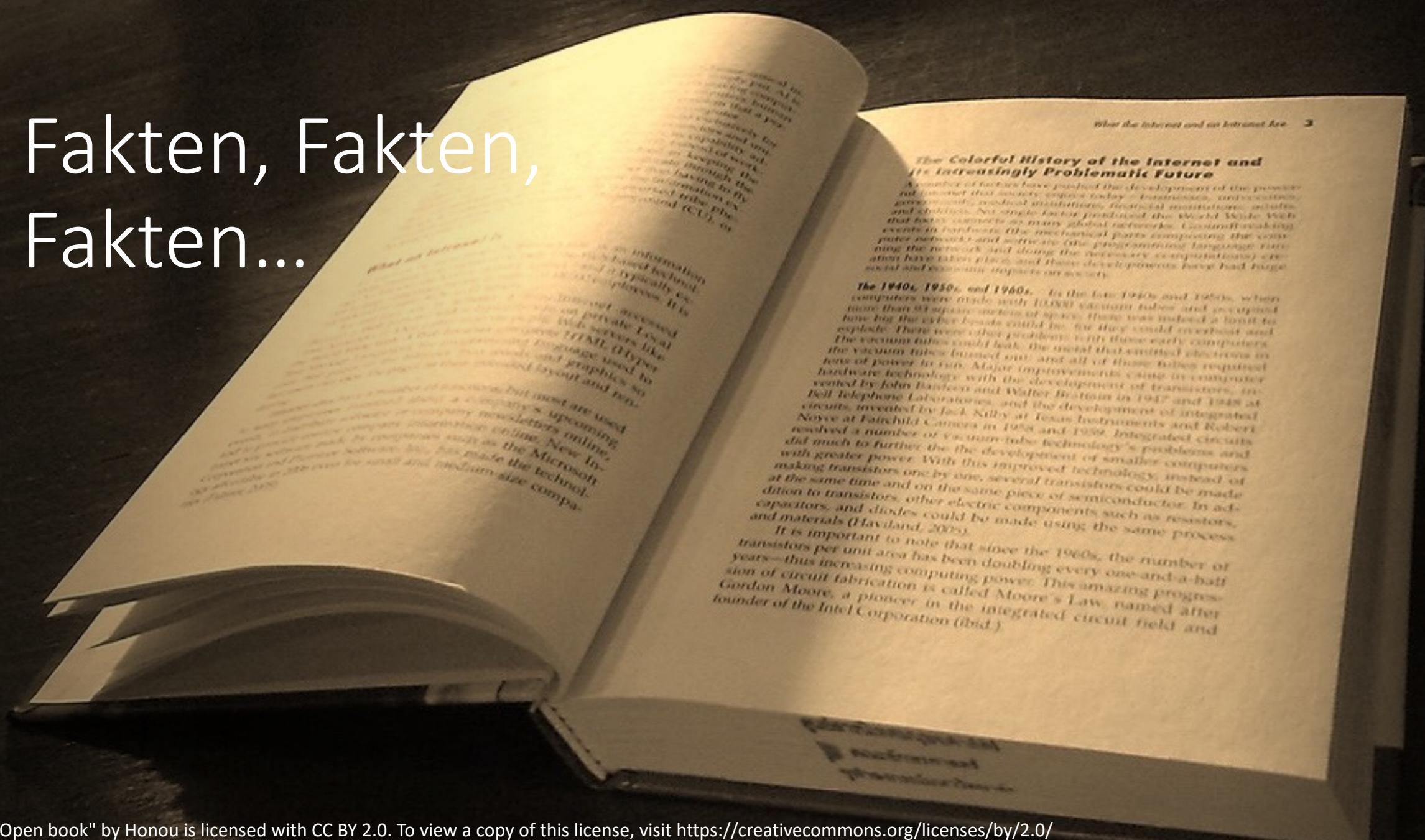
- Grundlagen
- Funktionen
- Flusskontrolle
- Klassen & Objekte
- Nebenläufigkeit



Und vor allem ...

```
<!-- Navbar -->
<div class="w3-top">
  <div class="w3-bar w3-black w3-hide-large w3-right" href="javascript:void(0)" title="Toggle Navigation Menu"><i class="fa fa-bars"></i>
    <a href="#" class="w3-bar-item w3-button w3-padding-large">HOME</a>
    <a href="#" class="w3-bar-item w3-button w3-padding-large">BAND</a>
    <a href="#tour" class="w3-bar-item w3-button w3-padding-large">TOUR</a>
    <a href="#" class="w3-bar-item w3-button w3-padding-large w3-hide-small">CONTACT</a>
  </div>
  <div class="w3-dropdown-hover w3-hide-small">
    <button class="w3-padding-large w3-button" title="More">MORE<i class="fa fa-caret-down"></i></button>
    <div class="w3-dropdown-content w3-bar-block w3-padding-large">
      <a href="#" class="w3-bar-item w3-button w3-padding-large">ABOUT</a>
      <a href="#" class="w3-bar-item w3-button w3-padding-large">MEMBER</a>
      <a href="#" class="w3-bar-item w3-button w3-padding-large">CONTACT</a>
    </div>
  </div>
</div>
```

Fakten, Fakten, Fakten...





- Soll eine moderne Alternative zu Java sein
- Syntax zu Java nicht kompatibel
- Interoperabilität sehr wichtig



Was ist Kotlin?

- Statisch typisiert, objektorientiert
- Läuft in der JVM
- Kann nach JavaScript transpiliert werden
- Unter Android bevorzugte Sprache
- Kann LLVM nutzen (Kotlin/Native)



- Juli 2011 stellt JetBrains „Project Kotlin“ als JVM-Sprache vor
- Februar 2012: Open Source (Apache 2-Lizenz)
- Februar 2016: Kotlin 1.0



- März 2017: Kotlin 1.1
 - Koroutinen (experimentell)
 - Underscores in numerischen Literalen
- Google IO 2017: Offizielle Unterstützung von Kotlin in Android Studio
- November 2017: Kotlin 1.2
 - Multiplatform-Projekte (experimentell)
 - lateinit für Toplevel-Eigenschaften u. lokale Variablen



Kotlin - Timeline

- Oktober 2018: Kotlin 1.3
 - Kotlin/Native
 - Koroutinen released (stable)
- August 2020: Kotlin 1.4
 - Argument Mixing
 - Koroutinen Debugger
- Oktober 2021: Kotlin 1.5
 - Default JVM Target 1.8
 - JVM Record Classes
- November 2021: Kotlin 1.6
 - Verbesserte Type Inference
 - Suspending Functions als Supertype (stable)
- Juni 2022: Kotlin 1.7
 - Neuer Kompiler (alpha)
 - Underscore Operator für Type Argumente (Type Inference)



- IntelliJ
- Android Studio
- Eclipse
- Kommandozeile mit Standalone-Compiler
- Online (play.kotlinlang.org)



Der Klassiker (in Java)

```
public class HelloWorld {  
  
    public static void main(String[] args) {  
        System.out.println("Hallo KKON!");  
    }  
  
}
```

- Ganz schön viel Overhead für ein kleines „Hello World!“, oder?



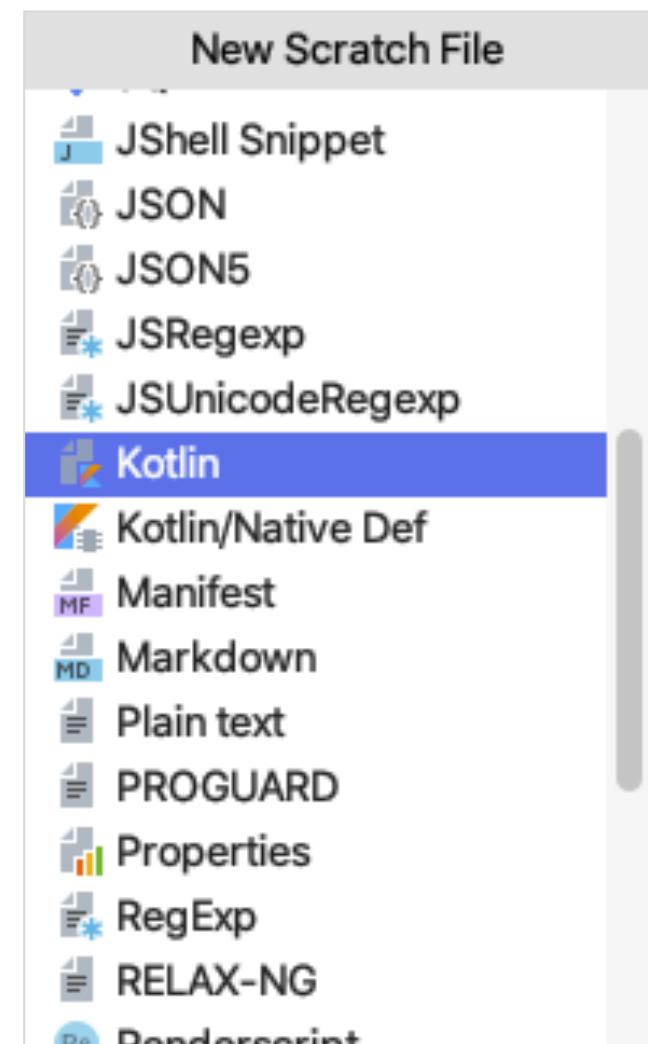
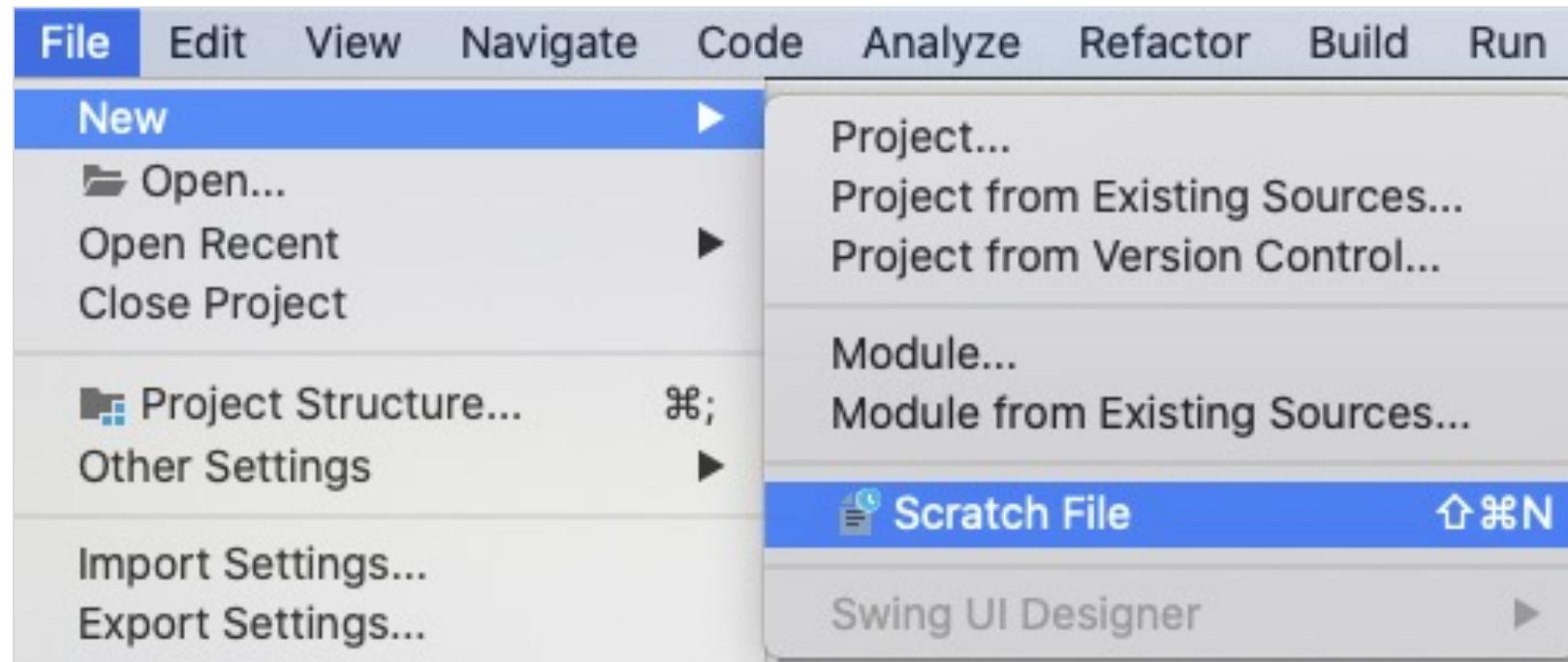
Und jetzt in Kotlin...

```
fun main() {  
    println("Hallo KKON!")  
}
```

- Beliebige Datei (CamelCase) mit Endung .kt
- main ist Toplevel-Funktion
- Keine Strichpunkte nötig



Scratch Files – Alles direkt ausprobieren



Beispiele und Materialien

git clone

<https://github.com/wern/java2kotlin-ws-HC-2022>

Alternativ per Download:

<https://tinyurl.com/ws-kotlin-hc22>

Passwort: J2K-HC2022

Grundlegendes



```
var i = 42
var j = "Hallo KKON!"
val k = 3.14
```

- Automatische Erkennung des Typs (Typinferenz)
- Mit `var` deklarierten Variablen können neue Werte zugewiesen (nur gleicher Typ!)
- Mit `val` deklarierten Variablen können nicht neu zugewiesen werden (Read-Only)



```
const val inputFileName = "Input.txt"
```

- Konstanten zum Übersetzungszeitpunkt können durch Voranstellen von `const` deklariert werden
 - Nur auf oberster Ebene (Toplevel)
 - Nur für konstante Werte
 - Nur für `val` Deklarationen



Einfache Variablen-deklarationen

```
val i = 42
val n = i as Number

val l = i.toLong()
```

- Explizite Casts mit `as`
- Umwandlung in einen anderen Typ mit `to<Type>()`



Datentypen – Ganze Zahlen

Typ	Bits	Kleinster Wert	Größter Wert
Byte	8	-128	127
UByte	8	0	255
Short	16	-32768	32767
UShort	16	0	65535
Int	32	$-2,147,483,648 (-2^{31})$	$2,147,483,647 (2^{31} - 1)$
UInt	32	0	$2^{32} - 1$
Long	64	$-9,223,372,036,854,775,808 (-2^{63})$	$9,223,372,036,854,775,807 (2^{63} - 1)$
ULong	64	0	$2^{64} - 1$



Datentypen - Fließkommazahlen

Typ	Bits	Signifikant	Bit Exponent	Dezimalstellen
Float	32	24	8	6-7
Double	64	53	11	15-16



```
val c = 'c'
```

- Typ Char
- Kann nicht direkt als Zahl verwendet werden
(Java: char a = 65;)
- Literal zwischen ''
- Bekannte Escape-Sequenzen: \t, \b, \n, \r, \', \", \\ und \\$
- Unicode: '\u...'



```
val b = true
```

- Typ Boolean (true, false)
- Operatoren u. a. ||, &&, !
- Alternativ: or und and (werden vollständig ausgewertet)



Felder (Klasse Array<...>)

```
val intArray = arrayOf(1, 2, 3)  
println(intArray[intArray.size - 1])
```

```
val emptyIntArray = arrayOfNulls<Int>()  
println(emptyIntArray[emptyIntArray.size - 1])
```

- Erzeugung über built-in Funktion

- Typ wird automatisch ermittelt

- Direkte Angabe der einzelnen Werte

- Alternativ: Erzeugung eines leeren Arrays



Felder (Klasse Array<...>)

```
val ints = Array(3) { i -> i + 1 }
ints.forEach { println(it) }
```

- Erzeugung der Werte durch Angabe einer entsprechenden Factory Funktion
Wird für jedes Element mit Indexangabe gerufen



```
val intArray = intArrayOf(1, 2, 3)  
println(intArray[intArray.size - 1])
```

```
val ints = IntArray(3) { i -> i + 1 }  
ints.forEach { println(it) }
```

- Für alle „Primitivtypen“ gibt es eigene Array Implementierungen um unnötigen Laufzeit-Overhead zu vermeiden



```
val s = "Hallo KKON!"
```

- Strings sind unveränderlich
- Bestehen aus Chars
- Zugriff auf Position mit []
- Länge: length
- Vergleich mit == bzw. !=



Zeichenketten – Raw Strings

```
val text = """
    for (c in "foo")
        print(c)
"""

println(text)
```

```
val text = """
|for (c in "foo")
|    print(c)
""".trimMargin()

println(text)
```

- *Raw Strings* können Whitespaces, Zeilenumbrüche und „beliebigen“ Text beinhalten
Führende Leerzeichen können per *trimMargin()* entfernt werden



```
val i = 10  
println("i = $i")
```

```
val s = "abc"  
println("$s.length is ${s.length}")
```

- Strings unterstützen *templates und template expressions*
 - Template: Referenzzugriff via `$<Referenzname>`
 - Template Expression via `{$ <auszuwertender Ausdruck>}`
- Auch in *Raw Strings* möglich

Übung: Datentypen

- `javakotlinworkshop.java.DataTypes` in Kotlin implementieren
- Paket `javakotlinworkshop.teilnehmer` für eigene Lösungen nutzen

```
fun printClassName(num: Any) {  
    println(num::class.qualifiedName)  
}
```



```
<!-- Navbar -->  
<div class="w3-top">  
<div class="w3-bar w3-black w3-hide-large w3-right" href="javascript:void(0)" title="Toggle Navigation Menu"><i class="fa fa-bars"></i>  
<a href="#" class="w3-bar-item w3-button w3-padding-large">HOME</a>  
<a href="#" class="w3-bar-item w3-button w3-padding-large">BAND</a>  
<a href="#tour" class="w3-bar-item w3-button w3-padding-large">TOUR</a>  
<a href="#contact" class="w3-bar-item w3-button w3-padding-large w3-hide-small">CONTACT</a>  
<div class="w3-dropdown-hover w3-hide-small">  
<button class="w3-padding-large w3-button" title="More"><i class="fa fa-caret-down"></i></button>  
<div class="w3-dropdown-content w3-bar-block w3-padding-large">
```

Funktionen





```
fun name(p1: Typ [= wert][, ...]): Typ {  
    ...  
    return ...  
}
```

- Deklaration durch das Keyword `fun`
- Parameter werden in der Form `name: Typ` angegeben
- Parameter- und Rückgabetypen müssen (bei Blocksyntax) **explizit** definiert werden
- Angabe von Defaultwerten für Parameter möglich



Funktionen - Beispiel

```
public static int pow(int n) {  
    return n * n;  
}
```

Java

```
fun pow(n: Int): Int {  
    return n * n  
}
```

Kotlin



Funktionen – Parameter immer read-only

```
scratch.kts
```

```
1  fun pow2(n: Int): Int {  
2      n = n * n  
3      return n  
4  }  
5  
6  println(pow2( 3))
```

- In Kotlin sind Parameter unveränderlich
- In Java müssen sie hingegen explizit auf `final` gesetzt werden



Funktionen – Beispiel (mit unveränderbaren Parametern)

```
public static int pow(final int n) {  
    return n * n;  
}
```

Java

```
fun pow(n: Int): Int {  
    return n * n  
}
```

Kotlin



```
fun name(p1: Typ [= wert], ...) = ...
```

- Besteht eine Funktion aus nur einem Ausdruck, kann dieser per Zuweisung als Funktion definiert werden
- Rückgabetyp kann aus Ausdruck ermittelt werden
- Rückgabe des Ausdrucks definiert Rückgabewert der Funktion



Funktionen – Beispiel (Single expression functions)

```
public static int pow(final int n) {  
    return n * n;  
}
```

Java

```
fun pow(n: Int) = n * n
```

Kotlin



Funktionen ohne Rückgabewert

```
fun main(): Unit {  
    println("Hallo KKON!")  
}
```

- *Unit* entspricht *void* in Java

```
fun main() {  
    println("Hallo KKON!")  
}
```

- Angabe des Rückgabetyps *Unit* kann entfallen



Funktionen – Defaultwerte

```
fun pow(n: Int = 2) = n * n
```

```
println(pow(4)) // => 16
```

```
println(pow()) // => 4
```

- Parameter mit Defaultwerten sind beim Aufruf optional



```
fun send(msg: String,  
        to: String,  
        encrypt: Boolean = false) {...}
```

```
send(to = "werner@mathema.de",  
      msg = "Hello Kotlin")
```

- Argumente können beim Aufruf explizit benannt werden
- Reihenfolge dabei nicht(!) eingehalten werden



- Variable Argumentenliste

```
fun arrayOf(varargs elements: T): Array<T> {...}
```

- Generic Functions

- Function Scope

- Top-Level Functions

- Local Functions

- Member Functions

- Extension Functions

```
fun outer(i: Int) {  
    fun inner(j: Int) = i * j  
  
    println(inner(i+1))  
}
```

- Infix Notation

```
infix fun Int.doSomethingWith(i: Int) =  
    this * i * 42
```

```
println(2 doSomethingWith 5) // => 638
```



Add-On: Extension Functions

```
fun String.replaceAWithB(): String {  
    return this.replace('A', 'B')  
}  
  
val s = "AB"  
println(s.replaceAWithB())
```

- Klassen um neue Funktionen erweitern, ohne...
 - von ihr abzuleiten
 - auf Muster zurückzugreifen (Decorator)
- Ist hilfreich, wenn der Quelltext nicht zur Verfügung steht (Klasse aus fremder Bibliothek)



- Der *Receiver Type* kann generisch sein
- `this` im Rumpf der Funktion ist optional
- Erweiterungen werden statisch aufgelöst
Modifizieren Klassen nicht
„Normaler“ Methodenaufruf
Evtl. unerwartete Effekte bei Vererbung!

```
open class A
class B: A()

fun A.getName() = "A"
fun B.getName() = "B"

fun printClassName(s: A) {
    println(s.getName())
}

printClassName(B())
```

Flusskontrolle



for-Schleifen

```
for (int i = 1; i <= 3; i++) {  
    System.out.println(i);  
}  
  
for (int i = 6; i >= 0; i -= 2) {  
    System.out.println(i);  
}
```

Java

```
for (i in 1..3) {  
    println(i)  
}  
  
for (i in 6 downTo 0 step 2) {  
    println(i)  
}
```

Kotlin



foreach (Arrays)

```
var array = new String [] {"Eins",
                           "Zwei", "Drei"};
for (var s : array) {
    System.out.println(s);
}
```

Java

```
val array = arrayOf("Eins",
                     "Zwei", "Drei")
for (i in array.indices) {
    println(array[i])
}
```

Kotlin



foreach (Collections)

```
var list = Arrays.asList("Eins",
                         "Zwei", "Drei");
for (var s : list) {
    System.out.println(s);
}

list.forEach(s ->
    System.out.println(s));
```

Java

```
val list = listOf("Eins",
                  "Zwei", "Drei")
for (i in list) {
    println(i)
}

list.forEach {println(it)}
```

Kotlin



while-Schleifen

```
int i = 0
while (i++ < 3) {
    System.out.println(i);
}

int j = 0
do {
    System.out.println(i);
} while (i++ < 3)
```

Java

```
var i = 0
while (i++ < 3) {
    println(i)
}

var i = 0
do {
    println(i)
} while (i++ < 3)
```

Kotlin



Bedingungen

```
int num = (int) (Math.random() * 4f);
System.out.println(String.format("%d ist %s",
    num,
    isEven(num) ? "gerade" : "ungerade"));
```

Java

```
val num = (Math.random() * 4).toInt()
println("$num ist ${if (isEven(num)) "gerade" else "ungerade"})
```

Kotlin



Bedingungen

```
val text = if(isEven(num)) {  
    println("$num ist gerade.")  
    "gerade"  
} else {  
    println("$num ist ungerade.")  
    "ungerade"  
}
```

- If-Statement in Kotlin ist ein Ausdruck
 - Letzter Ausdruck wird als Ergebnis zurückgegeben
 - Bei Zuweisung *else* Zweig notwendig
- Ähnlich dem Ternären Operator (?:) in Java
 - In Kotlin aber auch in Blocksyntax möglich



When Expression

```
switch (num) {  
    case 0:  
        System.out.println("0");  
        break;  
    case 2:  
        System.out.println("2");  
        break;  
    default:  
        System.out.println("1 oder 3");  
}
```

Java

```
when (num) {  
    0 -> println("0")  
    2 -> println("2")  
    else -> println("1 oder 3")  
}
```

Kotlin



When Expression – Mehr als ein switch Statement

```
val value : Any = ...

when (value) {
    0, 1 -> println("$value wäre gerne binär.")
    is String -> println ("$value ist ein String.")
    42 -> println ("$value ist die Antwort!")
    else -> println("$value ist seltsam...")
}
```

- `when` prüft die angegebene Bedingung und führt im Wahrheitsfall den jeweiligen Branch aus
Kein Fall-Through!
- Bedingungen sowohl Konstanten als auch beliebige Ausdrücke enthalten

Übung: Hangman

- Es können einzelne Buchstaben oder ganze Wörter geraten werden
- Aktueller Stand wird in Form R-cht- angezeigt
- Paket `javakotlinworkshop.teilnehmer` für eigene Lösungen nutzen

```
val eingabe = readLine().orEmpty()  
.toLowerCase()
```

```
<!-- Navbar -->  
<div class="w3-top">  
<div class="w3-bar w3-black w3-hide-large w3-right" href="javascript:void(0)" title="Toggle Navigation Menu"><i class="fa fa-bars" style="font-size:16px;"></i></div>  
<a href="#" class="w3-bar-item w3-button w3-padding-large" style="font-size:16px;">HOME</a>  
<a href="#" class="w3-bar-item w3-button w3-padding-large" style="font-size:16px;">BAND</a>  
<a href="#tour" class="w3-bar-item w3-button w3-padding-large" style="font-size:16px;">TOUR</a>  
<a href="#contact" class="w3-bar-item w3-button w3-padding-large" style="font-size:16px;">CONTACT</a>  
<div class="w3-dropdown-hover w3-hide-small">  
<button class="w3-padding-large w3-button" style="font-size:16px;">fa fa-caret-down"></i></button>  
<div class="w3-dropdown-content w3-bar-block w3-padding-large" style="background-color: #f1f1f1; min-width: 160px; position: absolute; z-index: 1; left: -10px; top: 100%>  
<a href="#" class="w3-bar-item w3-button w3-padding-large" style="font-size:16px;">MORE</a>
```



Das Nichts



- Null-Referenzen sind oft Grund für Abstürze/Exceptions
- Fehlerquelle ist oft schwer zu finden
- Kotlin erlaubt standardmäßig **keine** Null-Werte für Variablen
 - Variablen müssen bei Definition initialisiert werden
 - Zuweisung von `null` grundsätzlich erst einmal nicht erlaubt

The screenshot shows a code editor with the following code:

```
var a = "Hello"
a = null
```

A tooltip is displayed over the line `a = null`, showing the error message:

Null can not be a value of a non-null type String

Below the message are two buttons:

- Add 'toString()' call
- More actions...



```
var a: Int? = 42
a = null
if (a != null) {
    println(a.toString(2))
}
```

- Wenn null-Referenzen unterstützt werden sollen, müssen explizit nullable Types (`String?`, `Int?`, ...) verwendet werden
- ABER: Zugriff nur über bestimmte Aufrufkonstrukte (null-Prüfung, *Safe Call, Elvis Operator, Assertion Operator*) möglich!



```
val s: String? = null  
println(s?.length)
```

- Direkter Zugriff auf *Nullable Types* per *Safe Call* möglich
 - ? . Methode wird nur aufgerufen, falls die Objektreferenz nicht-null ist, ansonsten wird null geliefert
 - Auch als safe call chain möglich (kunde?.adresse?.strasse?.hausnummer)
- Auch als *Safe Call Chain* möglich
 - kunde?.adresse?.strasse?.hausnummer



```
val s: String? = null  
println(s?.length ?: -1)
```

- Soll etwas anders als `null` geliefert werden, können sperrige if-Abfragen durch den sog. *Elvis Operator* verschlankt werden
 - ?.: Methode wird nur aufgerufen, falls die Objektreferenz nicht-null ist
 - : Code wird ausgeführt, falls die Objektreferenz Objekt `null` ist



Nullable Types – Augen zu und durch...

```
try {
    val s: String? = null
    println(s!!.length)
} catch (e: NullPointerException) {
    println("Crash")
}
```

- Der Methodenaufruf ohne vorherigen Check kann mit `!!.` erzwungen

werden

NullPointerException wird geworfen, falls Referenz `null` ist
Nutzung grundsätzlich nicht empfohlen



```
val l: List<String?> = listOf("Eins", null, "Drei")
for (i in l) {
    i?.let { println(it) }
}
```

- Aufrufe waren bisher immer direkte Methoden des *Nullable Types*
- Dank des funktionalen Ansatzes auch für beliebige Ausdrücke nutzbar
 - ? . let(...) führt die angegebene Funktion nur aus, falls die Referenz null ist
 - Zugriff auf die Referenz ist per it möglich

Übung: Null-Safety

- `javakotlinworkshop.java.NullSafety` in Kotlin implementieren
- Paket `javakotlinworkshop.teilnehmer` für eigene Lösungen nutzen

```
<!-- Navbar -->
<div class="w3-top">
  <div class="w3-bar w3-black w3-hide-large w3-right" href="javascript:void(0)" title="Toggle Navigation Menu"><i class="fa fa-bars"></i>
    <a href="#" class="w3-bar-item w3-button w3-padding-large">HOME</a>
    <a href="#" class="w3-bar-item w3-button w3-padding-large">BAND</a>
    <a href="#" class="w3-bar-item w3-button w3-padding-large">TOUR</a>
    <a href="#" class="w3-bar-item w3-button w3-padding-large w3-hide-small">CONTACT</a>
  </div>
  <div class="w3-dropdown-hover w3-hide-small">
    <button class="w3-padding-large w3-button" title="More"><i class="fa fa-caret-down"></i></button>
    <div class="w3-dropdown-content w3-bar-block w3-padding-large">
      <a href="#" class="w3-bar-item w3-button w3-padding-large">MORE</a>
    </div>
  </div>
</div>
```

Klassen und Objekte



Einfache Klassen (Definition)

Java

```
public class Person {  
    private String name;  
    private int age;  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setAge(int age) {  
        this.age = age;  
    }  
    public int getAge() {  
        return age;  
    }  
}
```

Kotlin

```
class Person(var name: String, var age: Int) {  
}
```



Primärer Konstruktor

```
class Person(var name: String, var age: Int) {  
  
    init {  
        println("Name: $name, Alter: $age")  
    }  
  
}
```

- Folgt auf Klassennamen und ggf. vorhandenem Typparameter
- Enthält keinen Code
- Bei Bedarf `init`-Blöcke
 - Konstruktor-Parameter können verwendet werden
 - Werden entspr. Reihenfolge im Code abgearbeitet



Sekundäre Konstruktoren

```
class Person(var name: String, var age: Int) {  
  
    init {  
        println("Name: $name, Alter: $age")  
    }  
  
    constructor() : this("Max", 42) {  
        println("Alles muss man selber machen... ;)")  
    }  
}
```

- Werden mit `constructor` deklariert
- Muss direkt oder indirekt primären Konstruktor aufrufen
`(this(...))`
- `init`-Blöcke werden vor Konstruktor-Code ausgeführt



Eine Frage der Reihenfolge

```
class ConstructorDemo(var name: String) {  
  
    val first = "First property: $name".also(::println)  
  
    init {  
        println("First block: $name")  
    }  
  
    val second = "Second property: ${name.toUpperCase()}".also(::println)  
  
    init {  
        println("Second block can access $second")  
    }  
  
    constructor() : this("Hallo") {  
        println("secondary constructor called")  
    }  
}
```



Einfache Klassen (Verwendung)

Java

```
public class ClassDemo {  
  
    public static void main(String[] args) {  
        Person person = new Person("John", 50);  
        person.setName("Max");  
        person.setAge(42);  
        print(person);  
    }  
  
    private static void print(Person person) {  
        System.out.println(String.format("%s, %d",  
            person.getName(),  
            person.getAge()));  
    }  
}
```

Kotlin

```
fun main(args: Array<String>) {  
    val person = Person("John", 50)  
    person.name = "Max"  
    person.age = 42  
    print(person)  
}  
  
fun print(person: Person) {  
    println("${person.name}, ${person.age}")  
}
```



Einfache Klassen (Verwendung)

```
fun main(args: Array<String>) {
    val person = Person("John", 50)
    person.name = "Max"
    person.age = 42
    print(person)
}

fun print(person: Person) {
    println("${person.name}, ${person.age}")
}
```

- Kein *new* zur Erzeugung nötig
- Property Zugriff über Pfad Syntax
 - Zugriff erfolgt dennoch per Getter bzw. Setter



Java

```
public class Person {  
    private String name;  
    private int age;  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
    // ...  
}  
  
public class Student extends Person {  
    private String majorSubject;  
  
    public Student (String name, int age,  
                   String majorSubject) {  
        super(name, age);  
        this.majorSubject = majorSubject;  
    }  
    // ...  
}
```

```
open class Person(var name: String, var age: Int) {  
}  
  
class Student(name: String,  
             age: Int,  
             val majorSubject: String)  
    : Person(name, age) {  
}
```

Kotlin



```
class Student(name: String,  
             age: Int,  
             val majorSubject: String) : Person(name, age) {  
  
    init {  
        println("A new Student...")  
    }  
  
}
```

- Basisklasse wird im Klassenkopf (nach Parameterliste des primären Konstruktors) angegeben
 - Any ist die Elternklasse, wenn Klassen keine explizit angeben
- Hat Basisklasse einen primären Konstruktor, muss dieser aufgerufen werden (... : Person(...))



```
open class Person(var name: String, var age: Int) {  
  
    init {  
        println("Name: $name, Alter: $age")  
    }  
  
    constructor() : this("Max", 42) {  
        println("Alles muss man selber machen... ;)")  
    }  
}
```

- Standardmäßig sind alle Klassen und Methoden `final`
- `open` deklariert Klasse und Methoden als erweiterbar
- `override` für überschriebene Methoden
- Kennzeichnung von abstrakten Klassen und Methoden mit `abstract`

Übung: Klassen und Vererbung

- Implementierung einer Person Basisklasse
 - Attribute: Name, Vorname
 - Ableitung einer Klasse User
 - Attribute: Login, Passwort
 - In welcher Reihenfolge werden Init Blöcke und Konstruktoren gerufen?

```
<div class="w3-top">
<div class="w3-bar w3-black w3-hide-large w3-right" href="javascript:void(0)" title="Toggle Navigation Menu"><i class="fa fa-bars"></i>
<a href="#" class="w3-bar-item w3-button w3-padding-large">HOME
<a href="#band" class="w3-bar-item w3-button w3-padding-large">BAND
<a href="#tour" class="w3-bar-item w3-button w3-padding-large">TOUR
<a href="#contact" class="w3-bar-item w3-button w3-padding-large w3-hide-small">CONTACT
<div class="w3-dropdown-hover w3-hide-small">
<button class="w3-padding-large w3-button" title="More">MORE
<div class="w3-dropdown-content w3-bar-item w3-padding-large w3-hide-small">
```



Singlets in Java

```
public class Singleton {  
  
    public static final Singleton INSTANCE = new Singleton();  
  
    public void doSomething() {  
        System.out.println("Hello");  
    }  
  
    public static void main(String[] args) {  
        Singleton.INSTANCE.doSomething();  
    }  
}
```



- Kotlin kennt kein static Schlüsselwort
- Funktionen und Variablen außerhalb einer Klasse sind Top- oder Packagelevel

Möglicher Ersatz für Utility-Klassen aus Java
Implementierungsvariante für Singletons (Objekt Deklaration)

```
fun main(args: Array<String>) {
    Singleton.doSomething()
}

object Singleton {
    fun doSomething() = println("Hello Kotlin")
}
```



Object Declarations

- Initialisierung ist Thread-sicher
- Einfache Verwendung durch Zugriff über Namen
- Auch innerhalb von Klassen möglich
 - Benannter Zugriff innerhalb der Klasse aber „unschön“

```
fun main() {  
    MyClass.Singleton.doSomething()  
}  
  
class MyClass {  
    object Singleton {  
        fun doSomething() = println("Hello Kotlin")  
    }  
}
```



Companion Objects

```
class MyClass {  
    companion object Worker {  
        fun doSomething() = println("Hallo")  
    }  
}  
  
MyClass.doSomething()
```

- Companion-Objekte können einen Namen haben
 Zugriff immer „direkt“ möglich
- Member sind normale Instanz-Member
- Für JVM kann mit `@JvmStatic` ein statischer Member erzeugt werden



Gleichheit und Identität (in Java)

```
var a = "Hallo KKON";
var b = new String(a.getBytes(Charset.defaultCharset()),
    Charset.defaultCharset());

// referential equality
System.out.println(String.format("a == b: %b", a == b));

// structural equality
System.out.println(String.format("a.equals(b): %b", a.equals(b)));
```

- `==` prüft bei Primitiven auf Gleichheit (`2 == 2`, `2 != 3`)
- Problematisch: prüft bei Objekten Gleichheit der Referenz (Identität)
- `equals()` und `hashCode()` als Kompromiss



Vergleiche in Kotlin

```
val a = "Hallo KKON"
val b = String(a.toByteArray(Charset.defaultCharset()),
               Charset.defaultCharset())

// referential equality
println(String.format("a === b: %b", a === b))

// structural equality
println(String.format("a == b: %b", a == b))
```

- `==` prüft auf Gleichheit (auch bei Objekten!)
- Identitätsprüfung mit `===`
- Semantisches `equals()` und `hashCode()` mit *Data Classes*

Sichtbarkeit



- Klassen, Objekte, Interfaces, Konstruktoren, Funktionen, Eigenschaften und deren Setter können in ihrer Sichtbarkeit eingeschränkt werden
- `private`, `protected`, `internal`
- `public` ist Standard
 - war zu Beginn `internal`^{*)}
 - Achtung: Unterschied zu Java (package-private)



private

```
private val i: Int = 42

private fun doSomething(){
    println(i)
}

class Foo() {
    private val j: Int = 1
    private fun doSomethingElse(){
        println(i * j)
    }
}
```

- Private Toplevel-Deklarationen nur innerhalb der Datei sichtbar
- Klassen, Interfaces und Properties nur innerhalb der Klasse bzw. des Interfaces



protected

```
class Foo() {  
    protected val j: Int = 1  
    protected fun doSomething(){  
        println(i * j)  
    }  
}
```

- Für Toplevel-Deklarationen nicht verfügbar
- Funktionen, Klassen, Interfaces und Properties innerhalb der Klasse bzw. des Interfaces sowie in abgeleiteten Klassen



- Interne Toplevel-Deklarationen innerhalb eines **Moduls** sichtbar
- Jeder innerhalb desselben **Moduls**, der die deklarierende Klasse sieht, sieht auch einen internen Member



- Module bestehen aus zusammen übersetzten Kotlin-Dateien
- Quellen sind
 - IntelliJ IDEA-Module
 - Maven-Projekte
 - Gradle source sets
 - Mehrere Quelltextdateien, die in einem <kotlinc> Ant Task übersetzt wurden



Funktionen auf Objekte anwenden



- Auch Funktionen, die nicht Teil eines Objektes sind, können im Scope eines Objektes ausgeführt werden
 - Scope Function liefern den Scope des Objektes in die Funktion
- Scope Function bieten keine zusätzliche Funktionalität, führen aber zu besser strukturiertem und lesbarerem Code
- Die einzelnen Funktionen unterscheiden sich durch
 - den Zugriff auf das Objekt in dessen Scope sie laufen
 - den Rückgabewert



Scope Functions – let()

```
val l: List<String?> = listOf("Eins", null, "Drei")
for (i in l) {
    i?.let { println(it) }
}
```

- Zugriff auf das Objekt als Argument des Lambda Ausdrucks (it)
- Rückgabewert ist der Rückgabewert des Lambda Ausdrucks
- z.B. zur Ausführung von Code nach NotNull-Checks



Scope Functions – with()

```
val person = Person("John", 50)
with(person) {
    println("Die Person heißt ${name}.")
    println("Die Person ist ${age} Jahre alt.")
}
```

- Zugriff auf das Objekt via `this` (Verwendung ist optional)
Objekt wird als Argument „hereingegeben“
- Rückgabewert ist der Rückgabewert des Lambda Ausdrucks
- z.B. Ausführung einer Folge von Funktionsaufrufen auf einem Objekt ohne Interesse am Rückgabewert



Scope Functions – run()

```
val hexNumberRegex = run {  
    val digits = "0-9"  
    val hexDigits = "A-Fa-f"  
    val sign = "+-"  
  
    Regex("[\$sign]?[\$digits\$hexDigits]+")  
}
```

- Zugriff auf das Objekt via `this` (Verwendung ist optional)
`run` kann auch ohne Kontext-Objekt arbeiten
- Rückgabewert ist der Rückgabewert des Lambda Ausdrucks
- z.B. zur Kombination von Initialisierung und Berechnung oder
Zusammenfassung mehrerer Ausdrücke in einen



Scope Functions – apply()

```
val person = Person().apply {  
    name = "John"  
    age = 42  
    println("Die Person initialisiert mit ${name}, ${age}.")  
}
```

- Zugriff auf das Objekt via `this` (Verwendung ist optional)
- Rückgabewert ist wieder das Objekt
- z.B. Ausführung von Funktionen die das Objekt nicht zurückliefern, aber die Referenz auf das Objekt benötigt wird



Scope Functions – also()

```
val person = Person("John", 24)
    .also { println("Die Person wurde initialisiert mit ${it.name}, ${it.age}.") }
    .also { it.age = 42 }
    .also { println("Die Person ist jetzt ${it.age} Jahre alt.") }
}
```

- Zugriff auf das Objekt als Argument des Lambda Ausdrucks (it)
- Rückgabewert ist wieder Objekt
- z.B. zur Ausführung von Funktionen innerhalb einer Chain

Nebenläufigkeit



Nebenläufigkeit in Java und in Kotlin

```
public class BlockingDemo {  
  
    public static void main(String[] args) {  
        Thread t = new Thread(() -> {  
            try {  
                Thread.sleep(1000);  
                System.out.println("KKON!");  
            } catch (InterruptedException e) {}  
        });  
        t.start();  
        System.out.print("Hallo ");  
        try {  
            t.join();  
        } catch (InterruptedException e) {}  
    }  
}
```

Java

```
fun main() = runBlocking {  
    launch {  
        delay(1000L)  
        println("KKON!")  
    }  
    println("Hallo")  
}
```

Kotlin

Benötigt
org.jetbrains.kotlinx:kotlinx-coroutines-core:1.5.2



Koroutinen in wenigen Worten

```
fun main() = runBlocking {
    launch {
        delay(1000L)
        println("World!")
    }
    println("Hello")
}
```

- Koroutinen können unterbrechen und wieder fortgesetzt werden
- Sind wie leichtgewichtige Threads
- Kotlin nutzt für die Ausführung „irgendwie“ Threads



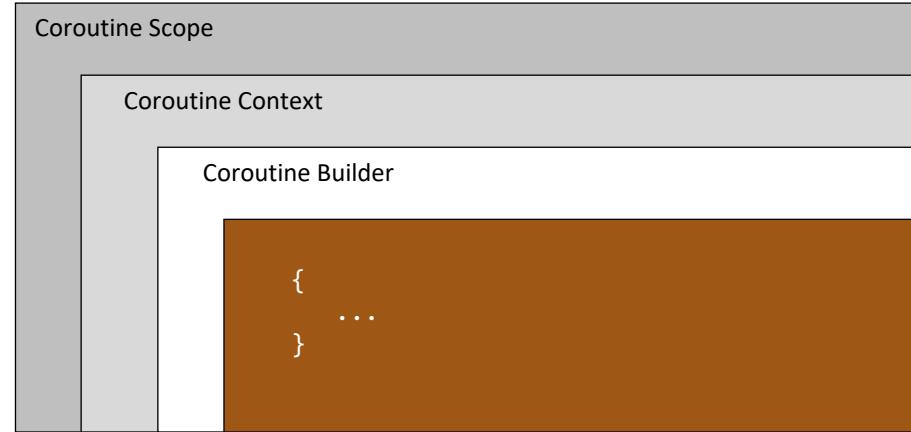
Strukturierte Nebenläufigkeit - Scopes

```
fun main() = runBlocking {
    launch {
        delay(1000L)
        println("KKON!")
    }
    println("Hallo")
}
```

- Koroutinen laufen immer innerhalb eines Scopes
 - Beschränkung der Laufzeit der Koroutinen (nur innerhalb des Scopes)
 - Auffangen von Fehlern
 - Keine „wild laufenden“ Threads
- Scope endet erst, wenn die letzte „Kind-Koroutine“ beendet wurde



Strukturierte Nebenläufigkeit



- Koroutinen laufen immer innerhalb eines Scopes
 - Beschränkung der Laufzeit der Koroutinen (nur innerhalb des Scopes)
 - Auffangen von Fehlern
 - Keine „wild laufenden“ Threads
- Scope endet erst, wenn die letzte „Kind-Koroutine“ beendet wurde



Koroutinen – Suspending Functions

```
fun main() = runBlocking {
    launch {
        delayedWorld()
    }
    println("Hallo")
}

suspend fun delayedWorld() {
    delay(1000L)
    println("KKON!")
}
```

- *Suspending Functions* können nur in Koroutinen verwendet werden
 - Verhalten sich wie „normale“ Funktionen
 - Können andere Suspending Funktionen rufen (z.B. `delay()`)



Koroutinen – Parallele Ausführung

```
fun main() = runBlocking {
    delayedWorlds()
    println("Fertig :)")
}

suspend fun delayedWorlds() = coroutineScope {
    launch {
        delay(2000L)
        println("Digital")
    }
    launch {
        delay(1000L)
        println("KKON")
    }
    println("Hallo")
}
```

- Start zweier paralleler Koroutinen
Innerer Scope endet erst, wenn beide Koroutinen beendet wurden!



Jobs – Wenn es „expliziter“ sein soll

```
fun main() = runBlocking {
    val job = launch {
        delayedWorld()
    }
    println("Hallo")
    job.join()
    println("Fertig :)")
}
suspend fun delayedWorld() {
    delay(1000L)
    println("KKON!")
}
```

- `launch` liefert bei der Erzeugung der Koroutine ein Job Objekt zurück
 - Ermöglicht Abfrage des Status der Koroutine
 - Kann zum expliziten Warten auf die Beendigung der Koroutine genutzt werden



Rückgabewerte mit `async` und `await`

```
fun main() = runBlocking {
    val audience = async { countAudience() }
    val presenter = async { countPresenter() }
    println("${audience.await()} Zuhörer und ${presenter.await()} Sprecher anwesend.")
}
suspend fun countAudience(): Int {
    delay(1000L)
    return 14
}
suspend fun countPresenter(): Int {
    delay(1000L)
    return 1
}
```

- `async` erzeugt eine Koroutine, ähnlich wie `launch`
Statt eines Jobs wird der Rückgabewert als Deferred zurückgegeben
- Mittels `await` kann auf das Beenden der Koroutine und den
Rückgabewerte gewartet werden



Weiterführende Themen

- Coroutine Dispatcher
- Start Modi
- Channels
- Asynchronous Flows
- Exception Handling
- ...



Kotlin
<https://kotlinlang.org/docs/coroutines-guide.html>

Referenzen

- Beispiele & Folien
 - <https://github.com/wern/java2kotlin-ws-HC-2022>
- Kotlin Dokumentation
 - <https://kotlinlang.org/docs/home.html>
- Kotlin Playground
 - <https://play.kotlinlang.org>



"Cambridge Public Library #4" by brokentrinkets is licensed with CC BY 2.0.
To view a copy of this license, visit <https://creativecommons.org/licenses/by/2.0/>



Herbstcampus

Vielen Dank! Fragen?

Werner Eberling

E-Mail: werner.eberling@mathema.de
Twitter: [@Wer_Eb](https://twitter.com/@Wer_Eb)

Beispiele:
<https://github.com/wern/java2kotlin-ws-HC-2022>

www.mathema.de



Beispiele?
Scan me ;)