ex|Xcellent solutions

Benjamin Schmid
Technology Advisor
@bentolor

Workshop : Effiziente & sichere JVM-Entwicklung mit Kotlin

2018-09-20

? for help

Java++: Effiziente & sichere JVM-Entwicklung mit Kotlin by Benjamin Schmid is licensed under a Creative Commons License (BY-NC-SA/4.0).

WAS IST KOTLIN?

Russische Insel

im finnischen Golf, 32km vor Petersburg

statisch typisierte

Allzweck-Sprache für

- **JVM**
- **Browser**
- **Native** (LLVM)

PRÄGNANZ

LESBARKEIT

SICHERHEIT

INTEROPERABILITÄT

TOOLING

DATENBLATT

Freie Software

APL 2.0 – IDE, compiler, libs, build tools

backed by JetBrains

20 Vollzeit & Community
„dogfooding“: 0,5 Mio. LOC

Stabil

Start: ~2010, v1.0 Feb. '16
Long-term backward compatibility

Integrationen

IntelliJ, Eclipse, Netbeans
Maven, Ant, Gradle, CLI

HISTORIE

Juli 2011
Offizielle Vorstellung

1.0 – Februar 2016
Production-ready
Long-term stability version

1.1 – April 2017
Coroutines, Type Aliases, JS, Android

Mai 2017
Google integriert Kotlin in Android

1.2 – November 2017
Multiplattform-Support

1.3 – ~Q4 2018
Contracts, Unsigneds,
Inline Classes

PRÄGNANZ VS. LESBARKEIT

← Optional semicolons

```
import java.util.*

fun main(args: Array<String>) {
    val name = if (args.size > 0) args[0] else "Publikum"
    val zuschauer = Gast(name, title = Title.wertes)

    println(zuschauer)
    println("Hallo ${zuschauer.title} ${zuschauer.name}")
}

class Gast(val name: String,
           var zeit: Date = Date(),
           val title: Title)
enum class Title { Herr, Frau, wertes }
```

PRÄGNANZ VS. LESBARKEIT

```
import java.util.*

fun main(args: Array<String>) {
    val name = if (args.size > 0) args[0] else "Publikum"
    val zuschauer = Gast(name, title = Title.wertes)

    println(zuschauer)
    println("Hallo ${zuschauer.title} ${zuschauer.name}")
}

class Gast(val name: String,
           var zeit: Date = Date(),
           val title: Title)
enum class Title { Herr, Frau, wertes }
```

← Optional semicolons

← Top-level functions

PRÄGNANZ VS. LESBARKEIT

```
import java.util.*

fun main(args: Array<String>) {
    val name = if (args.size > 0) args[0] else "Publikum"
    val zuschauer = Gast(name, title = Title.wertes)

    println(zuschauer)
    println("Hallo ${zuschauer.title} ${zuschauer.name}")
}

class Gast(val name: String,
           var zeit: Date = Date(),
           val title: Title)
enum class Title { Herr, Frau, wertes }
```

← Optional semicolons

← Top-level functions

← Kein `new`, named Params

PRÄGNANZ VS. LESBARKEIT

```
import java.util.*

fun main(args: Array<String>) {
    val name = if (args.size > 0) args[0] else "Publikum"
    val zuschauer = Gast(name, title = Title.wertes)

    println(zuschauer)
    println("Hallo ${zuschauer.title} ${zuschauer.name}")
}

class Gast(val name: String,
           var zeit: Date = Date(),
           val title: Title)
enum class Title { Herr, Frau, wertes }
```

← Optional semicolons

← Top-level functions

← Kein `new`, named Params

← String interpolation

PRÄGNANZ VS. LESBARKEIT

```
import java.util.*

fun main(args: Array<String>) {
    val name = if (args.size > 0) args[0] else "Publikum"
    val zuschauer = Gast(name, title = Title.wertes)

    println(zuschauer)
    println("Hallo ${zuschauer.title} ${zuschauer.name}")
}

class Gast(val name: String,
           var zeit: Date = Date(),
           val title: Title)
enum class Title { Herr, Frau, wertes }
```

← Optional semicolons

← Top-level functions

← Kein `new`, named Params

← String interpolation

← Default values

PRÄGNANZ VS. LESBARKEIT

```
import java.util.*

fun main(args: Array<String>) {
    val name = if (args.size > 0) args[0] else "Publikum"
    val zuschauer = Gast(name, title = Title.wertes)

    println(zuschauer)
    println("Hallo ${zuschauer.title} ${zuschauer.name}")
}

class Gast(val name: String,
           var zeit: Date = Date(),
           val title: Title)
enum class Title { Herr, Frau, wertes }
```

← Optional semicolons

← Top-level functions

← Kein `new`, named Params

← String interpolation

← Default values

← `val` ≙ `public final`

KOTLIN ENTWURFSPRINZIPIEN

in a nutshell

KOTLIN TARGETS

„Statically typed, run anywhere!“

- **JVM**
- **Android**
- **Browser** (Javascript)
- **Native**, z.B. iOS (via LLVM)

+ – EIN DREAM-TEAM?

- effiziente & fehlerarme Entwicklung
- konzipiert für den **industriellen Einsatz**
- erzeugt **Java 6 bzw. 8 Bytecode** – perfekt für Legacy & Android
- **100% bidirektionale Java Interoperabilität**
- **Tooling-Heimspiel**
- Geringe Runtimegröße & teils sogar weniger Overhead

Die Java Virtual Machine (JVM)

Java = **JVM** (Bytecode, Memory Model, GC, Hotspot)

+ **JRE** (Standard-Bibliothek wie `java.lang.*`)

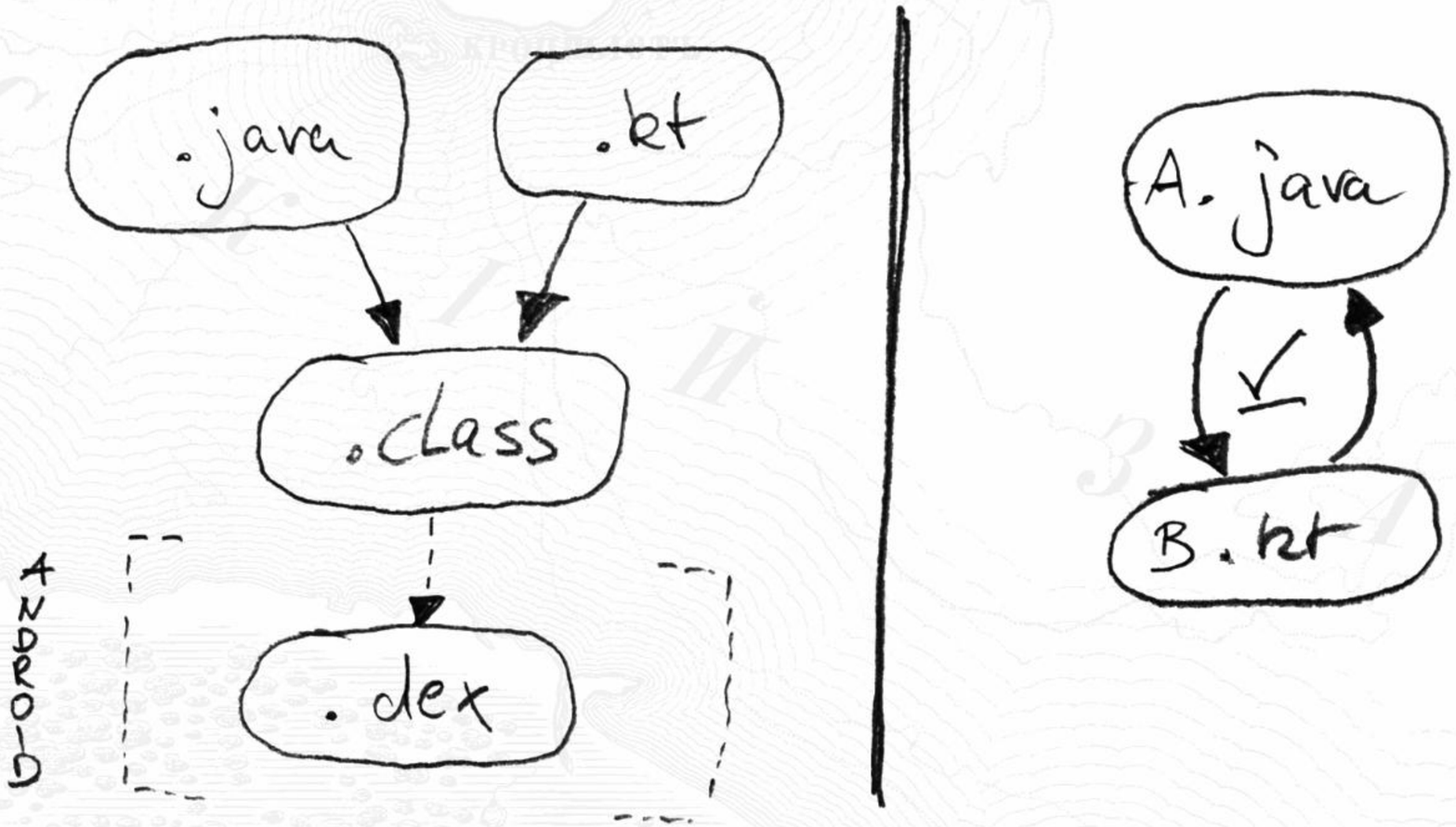
+ **JDK** (Compiler & Tools)

KOTLIN & JAVA

Nahtlose Java ⇆ Kotlin Interoperabilität ist Top Priorität

- Compiliert in Java 6 bzw. 8 Bytecode
  - damit legacy- & Android-friendly
  - Compiler versteht Kotlin & Java-Code
- Kein separates Kotlin-SDK
  - Kotlin **nutzt direkt die JDK-Klassen**
  - ergänzt viele, kleine **Extensions** – ca. 736KB in 1.0.6

Java/Kotlin Interop



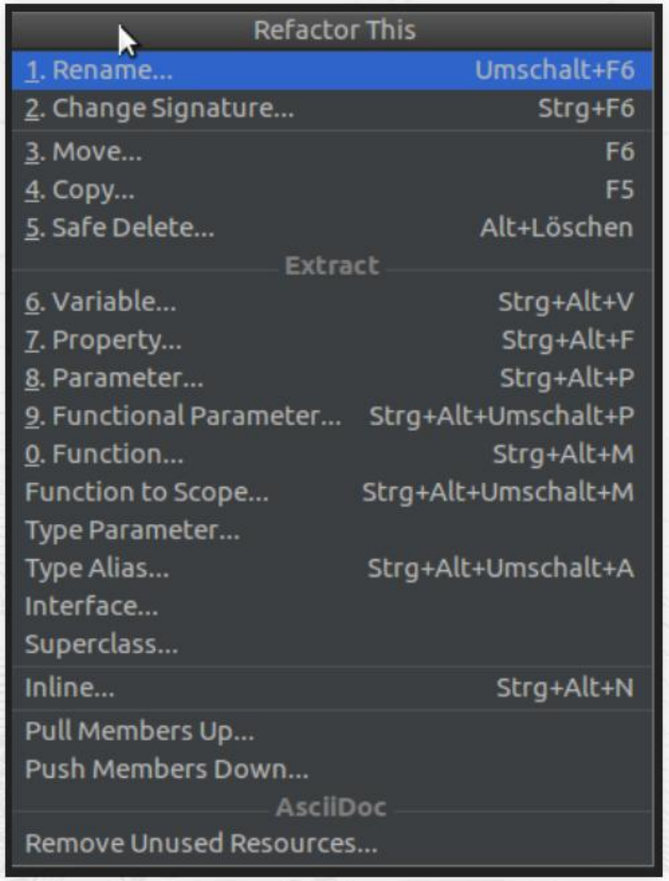
Mischen von Java & Kotlin-Code

bei problemlosem, gleichzeitigen Zugriff in beide Richtungen

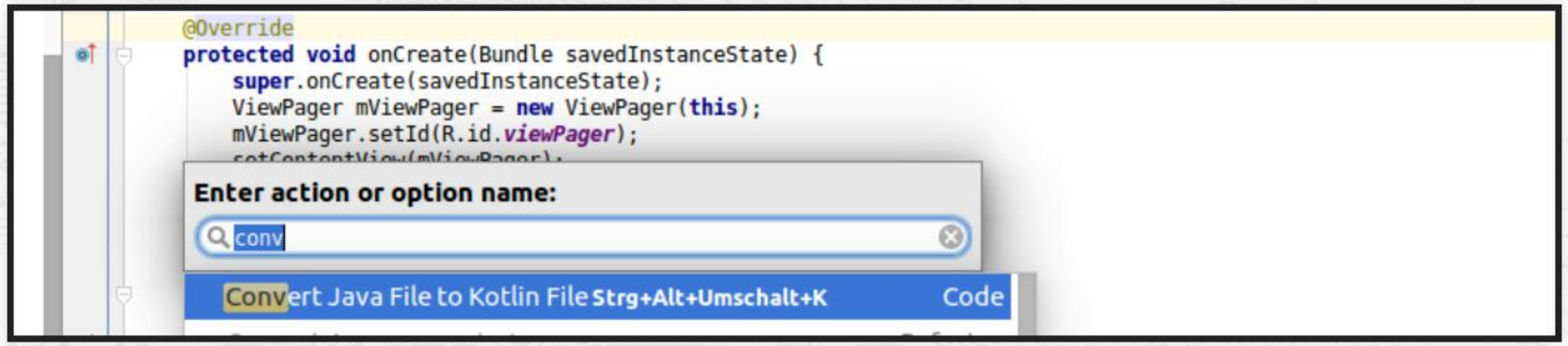
Schrittweise Einführung von Kotlin im Projekt damit problemlos möglich!

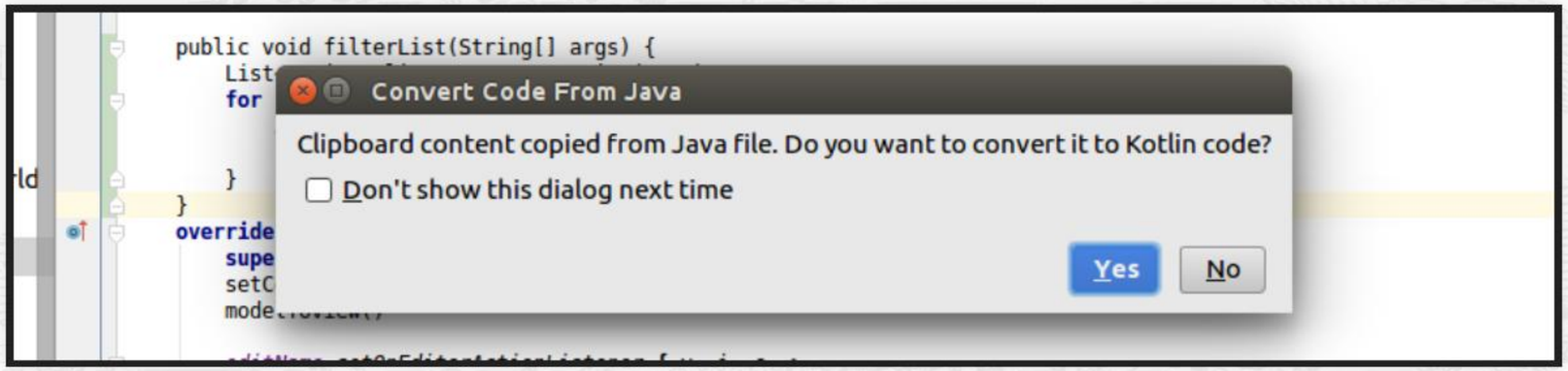
TOOLING: DESIGN GOALS

- Compiler **vergleichbar schnell wie Java**
  (vs. Scala)
- **Exzellenter IDE-Support:**
  (mit DSLs Grund #1 für Gradle für Kotlin)
  - Navigation
  - Refactorings
  - Inspections
  - Completions
- **Nahtloses Zusammenspiel**
  mit bestehenden Tools & Plattformen



Automatische Konvertierung Java → Kotlin





QUICK-START

KOTLIN VS. JAVA

In Kotlin gelöst:

- `NullPointerException`
- Keine Raw-Types
- echte Funktions-Typen
- Arrays sind invariant
- Type Projections

In Kotlin fehlt:

- Checked Exceptions
- Primitive Typen
- (nicht-private) Felder
- `static`
- Wildcards
- `a ? b : c`

Kotlin hat zusätzlich:

Lamba-Ausdrücke, `inline`, Extensions, Typ-Inferenz String-Templates, Smart Casts, Properties, Überladbare Operatoren Property/Class Delegation, `data class`, r/o Collections, Coroutines, Singletons, `<in T1, out T2>`, ...

Siehe auch: https://kotlinlang.org/docs/reference/comparison-to-java.html

KOTLIN KONVENTIONEN

- Folgt den meisten Java-Konventionen
  - Typen in Uppercase
  - Methoden & Properties in lower camelCase
- Pakete: Übliche Reverse Notation `de.exxcellent.foo`
  - Mehrere Klassen pro Datei erlaubt
  - Dateiname ≠ Klassenname
  - Ordnerstruktur kann ≠ Paketstruktur
  - auch Top-Level Funktionen (klassenlos) möglich

CRASH-KURS

- **Funktionen:** `fun methode( p1: T<X>, p2: T2 ) : String {…}`
- **Variablen:** `var` bzw. `val` (read-only)
- **Typen:**
  - `void` → `Unit`, `Integer` → `Int`, …
  - `String[]` → `Array<String>`
  - keine Sprach-Primitiven & Array-Typen (wie `int`)
- **Klassen:**
  - `public final class X {…}` → `class X {…}`
  - Konstruktor: `class X(a: Int, b: String) {…}`
- kein `new`, kein `static`, kein *package protected*
- Semicolon optional

BASICS

SESSION #1: BASICS

- `fun main(args: Array<String>)`
  - String interpolation
- `var` vs. `val`
- Funktionen
  - `if … else`-Expression
  - Kurzform
  - default params
  - Named params
- Type inference
- Control Structures: Loops
  - `while` & `for`
  - `in`
  - Ranges (`..`, `until`, `downTo`, `step`)

TYPE INFERENCE

```
fun noTypeInfer(vals: List<Int>): Unit {
  val distinct: Set<Int> = vals.toSet()
  val first: Int = vals[0]
  val negs: List<Int> = vals.filter(
    fun(n: Int): Boolean {
        return n < 0
    }
  )
}
```

TYPE INFERENCE

```
fun noTypeInfer(vals: List<Int>): Unit {
  val distinct: Set<Int> = vals.toSet()
  val first: Int = vals[0]
  val negs: List<Int> = vals.filter(
    fun(n: Int): Boolean {
        return n < 0
    }
  )
}
```

→

```
fun typeInfer(vals: List<Int>){
  val distinct = vals.toSet()
  val first = vals[0]
  val negs = vals.filter(
     fun(n) = n < 0


  )
}
```

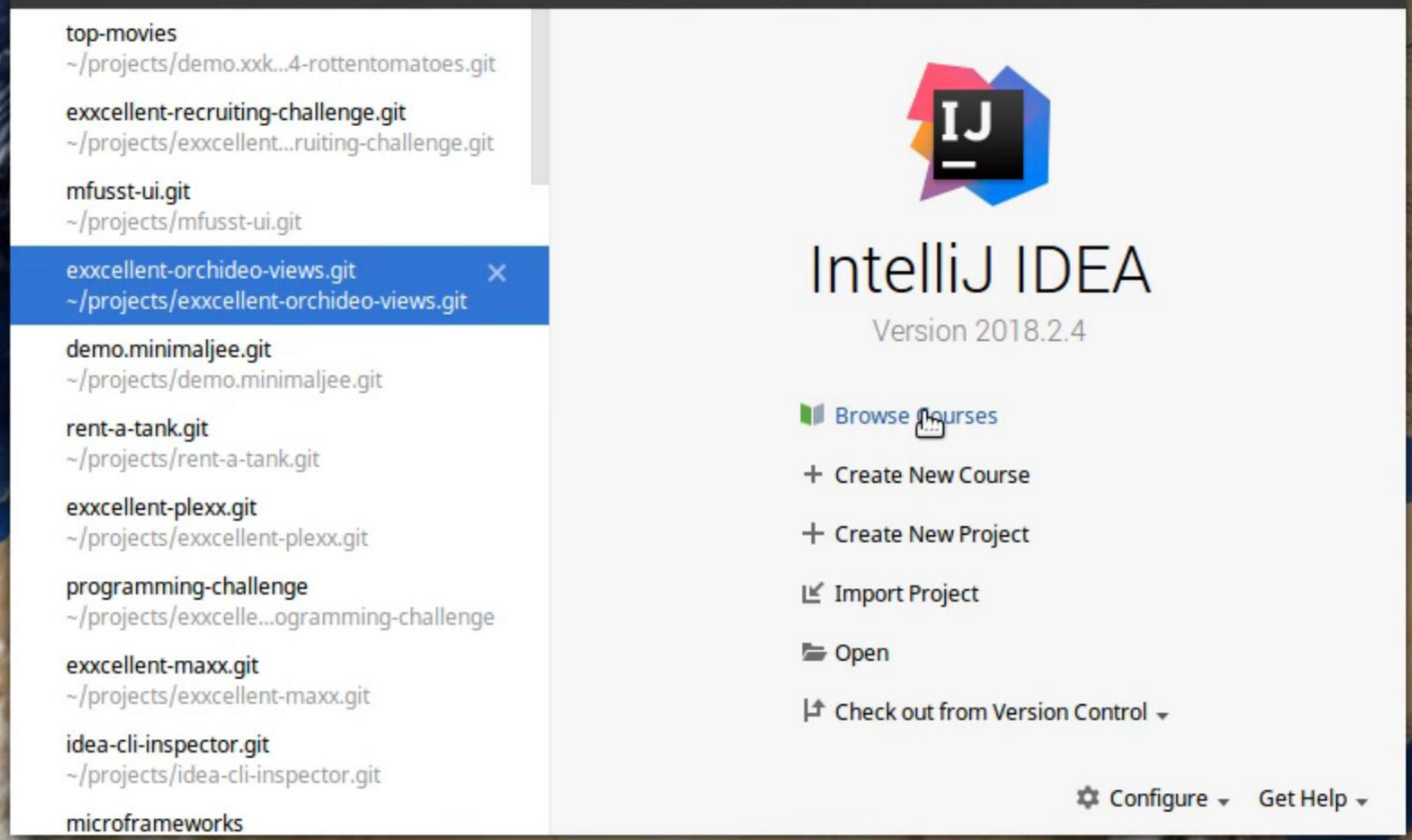
TYPE INFERENCE

```
fun noTypeInfer(vals: List<Int>): Unit {
  val distinct: Set<Int> = vals.toSet()
  val first: Int = vals[0]
  val negs: List<Int> = vals.filter(
    fun(n: Int): Boolean {
        return n < 0
    }
  )
}
```

→

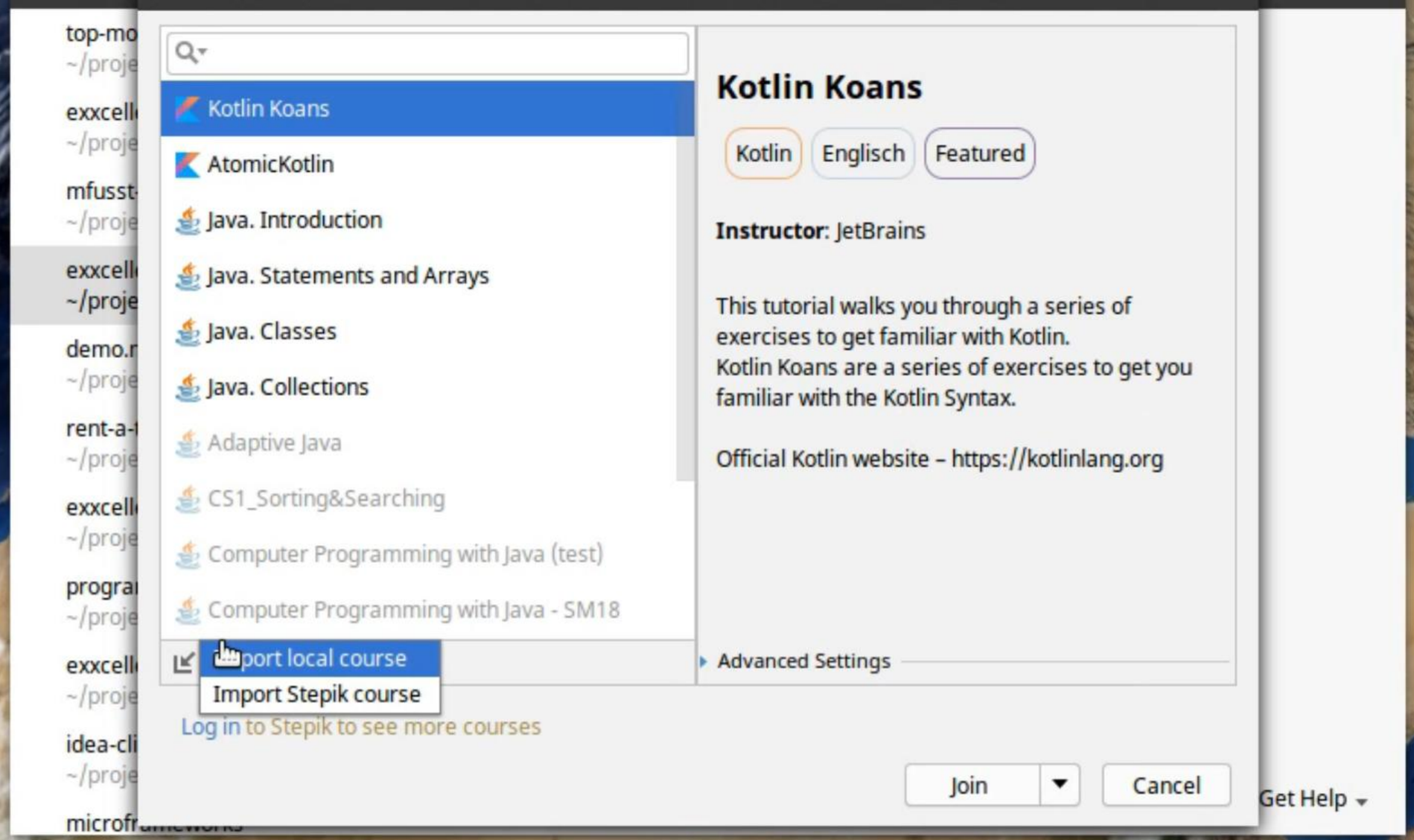
```
fun typeInfer(vals: List<Int>){
  val distinct = vals.toSet()
  val first = vals[0]
  val negs = vals.filter(
     fun(n) = n < 0


  )
}
```

Typangaben üblicherweise **nur an Schnittstellen** erforderlich



HANDS-ON – VORBEREITUNG

Download unter: https://raw.githubusercontent.com/bentolor/kotlin-workshop/master/doc/edutools-workshop.zip



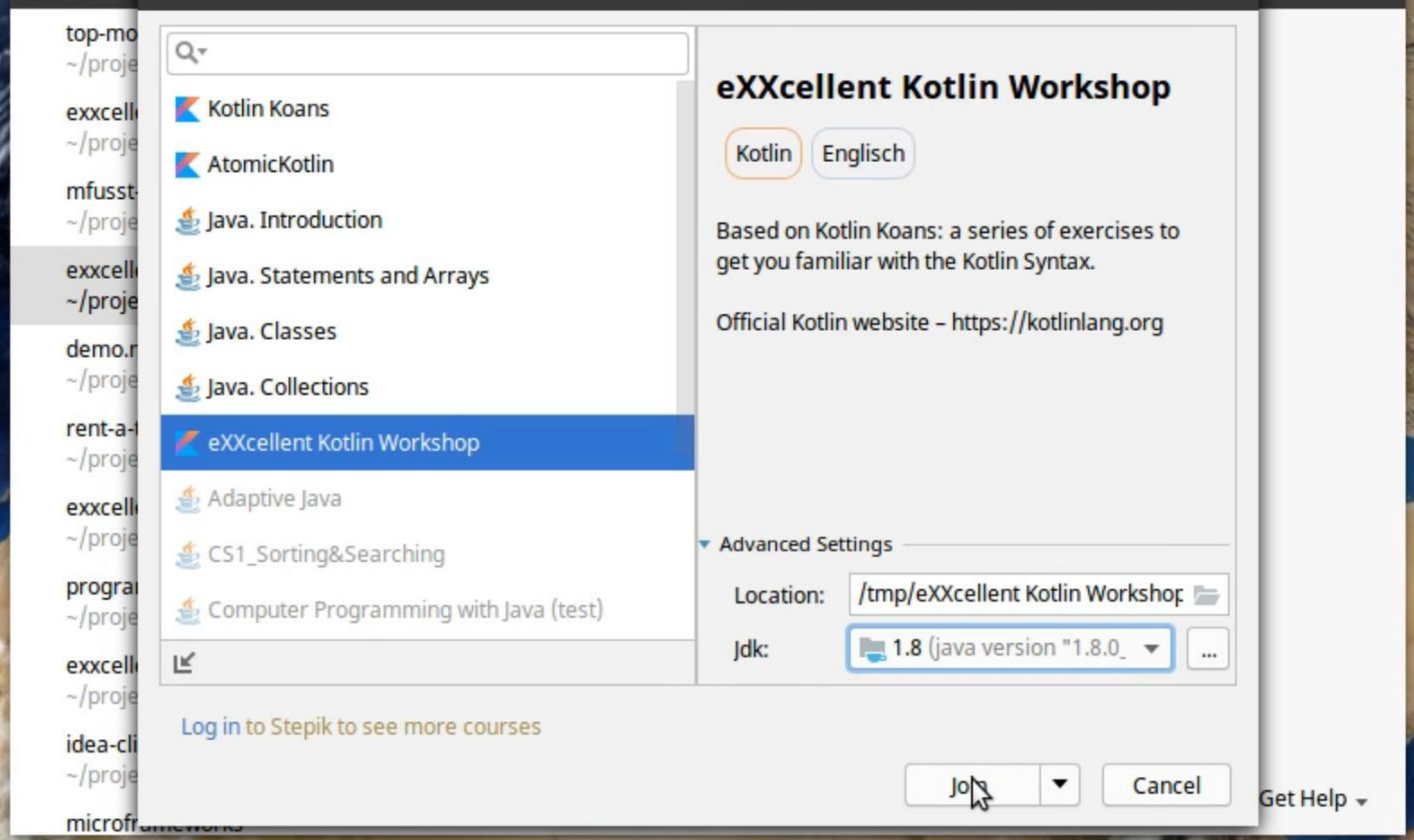
HANDS-ON – VORBEREITUNG

Download unter: https://raw.githubusercontent.com/bentolor/kotlin-workshop/master/doc/edutools-workshop.zip

... mit Standbard-Bibliotheks Extension

```
class WeatherIndicatorKt {
    fun rateWeather(celsius: Int) {
       val (status, color) = when {
         celsius < 5 -> "Saukalt!" to Color.BLUE
         celsius in 5..20 ->  "Geht so!" to Color.ORANGE
         else -> "Urlaub!" to Color.RED
       }
    }
}
```

```
fun <A, B> A.to(that: B): Pair<A, B> = Pair(this, that)
```



HANDS-ON – VORBEREITUNG

Download unter: https://raw.githubusercontent.com/bentolor/kotlin-workshop/master/doc/edutools-workshop.zip

HANDS-ON #1 - „BASICS“

Tipps

- `Strg` + `Alt` + `Eingabe` startet/testet direkt das Kata
- Aufgabenbeschreibung lesen
- Hinter Mausover über die Platzhalter liegen gelegentlich Hinweise & Tipps

KLEINES QUIZ?

KLASSEN

PROPERTIES

- Klassen in Kotlin kennen keine Felder, nur Properties
- Compiler **generiert** Getter- & Setter sowie Backing Field
- `getX()`/`setX()`-Paare aus **Java** erscheinen als Property `x`

- Properties unterstützen **Delegation**

KLASSEN – PROPERTIES

```
class Person  {
    val name: String = "Fix"
    var age: Int = -1
}

val p = Person()
println(p.name);
// p.name = "Hans"
p.age = 32
println(p.age)
```

Sicht aus Java

```
Person p = new Person();
p.getName();
//p.setName()
p.setAge(32);
p.getAge();
```

KLASSEN – PRIMÄRER KONSTRUKTOR

```
// public final Klasse mit Property "name" und Parameter "age"
class Person(val name: String, age: Int)  {



}

val vater = Person("Hans", 63)
```

KLASSEN – VERERBUNG

```
 // Inheritable class w/ property
open class AgingElement(var age: Int)

// public final Klasse mit Property "name" und Parameter "age"
class Person(val name: String, age: Int) : AgingElement(age) {




}

val vater = Person("Hans", 63)
```

KLASSEN – ²ND CONSTRUCTOR

```
 // Inheritable class w/ property
open class AgingElement(var age: Int)

// public final Klasse mit Property "name" und Parameter "age"
class Person(val name: String, age: Int) : AgingElement(age) {
    var parentName: String = "unknown"

    // Secondary Constructor
    constructor(name: String, parent: Person) : this(name, -1) {
        this.parentName = parent.name
    }

}

val vater = Person("Hans", 63)
val sohn = Person("Stefan", vater)
```

SESSION #2: KLASSEN

- Constructor & Properties
- Kurz-Form vs. Langform
- Steuerung Getter/Setter
  - `get()` & `set(value)`
  - Sichtbarkeiten
- `equals()` vs. reference
- Tuples
- Destructuring `componentN()`
- `data class`
- `interface`
- Hierarchies: `open` & `override`
- `object:`-Expression

DATA CLASSES

```
data class Gast(
    val name: String,
    var zeit: Date = Date(),
    val title: Title?
)
```

Werte-Container

- Um Methoden erweiterbar
- unterstützt Basisklassen seit Kotlin 1.1

Kotlin übernimmt

- Getter & Setter
- `equals()`
- `hashCode()`
- `toString()`
- `copy()` für Varianten
- `componentN()` für Destructuring

… exakt dasselbe mit Java

```java
public final class Gast {
  @NotNull
  private final String name;
  @NotNull
  private Date zeit;
  @Nullable
  private final Title title;

  @NotNull
  public final String getName() {
    return this.name;
  }

  @NotNull
  public final Date getZeit() {
    return this.zeit;
  }

  public final void setZeit(@NotNull Date date) {
    checkParameterIsNotNull(date, "<set-?>");
    this.zeit = date;
  }

  @Nullable
  public final Title getTitle() {
    return this.title;
  }

  public Gast(@NotNull String name,
              @NotNull Date zeit,
              @Nullable Title title) {
    checkParameterIsNotNull(name, "name");
    checkParameterIsNotNull(zeit, "zeit");
    this.name = name;
    this.zeit = zeit;
    this.title = title;
  }
```

```java
  public Gast(String string, Date date,
              Title title, int n) {
    if ((n & 2) != 0) {
      date = new Date();
    }
    this(string, date, title);
  }

  @NotNull
  public final String component1() {
    return this.name;
  }

  @NotNull
  public final Date component2() {
    return this.zeit;
  }

  @Nullable
  public final Title component3() {
    return this.title;
  }

  @NotNull
  public final Gast copy(@NotNull String name,
                         @NotNull Date zeit,
                         @Nullable Title title) {
    checkParameterIsNotNull(name, "name");
    checkParameterIsNotNull(zeit, "zeit");
    return new Gast(name, zeit, title);
  }

  public String toString() {
    return "Gast(name=" + this.name +
           ", zeit=" + this.zeit +
           ", title=" + this.title + ")";
  }
```

```java
  public int hashCode() {
    String s = this.name;
    Date d = this.zeit;
    Title t = this.title;
    return ((s != null ? s.hashCode() : 0) * 31
          + (d != null ? d.hashCode() : 0)) * 31
          + (t != null ? t.hashCode() : 0);
  }

  public boolean equals(Object object) {
    if (this == object) return true;

    if (!(object instanceof Gast))
        return false;

    Gast gast = (Gast)object;
    if (!areEqual(this.name, gast.name)
        || !areEqual(this.zeit, gast.zeit)
        || !areEqual(this.title, gast.title))
        return false;

    return true;
  }
}
```

… 3 Attribute!

HANDS-ON #2

CLASSES

KLEINES QUIZ?

PRÄGNANZ & LESBARKEIT

Schneller an's Ziel!

`when`-EXPRESSION

Quasi `switch`-Ersatz, nur mit deutlich mehr Power

```
when (x) {
    1 -> print("x == 1")
    2 -> print("x == 2")
    else -> { // Note the block
        print("x is neither 1 nor 2")
    }
}
```

`when` ON SPEED (1)

```
val validNumbers = setOf(4, 7, 13)

val result = when (x) {
    1, -1, 0 -> "x is simple"

    in 2..10 -> "x is in the range"
    in validNumbers -> "x is still valid"

    !in -1..20 -> "x is outside the range"

    validNumbers.first() * 2  -> "double of first"

    else -> "none of the above"
}
```

Enumerations, Ranges, Expressions, `when` als Expression

HANDS-ON #3

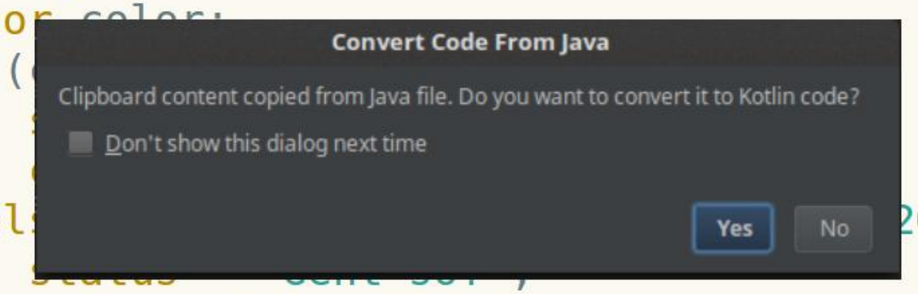
WHEN

„Wie ist denn das Wetter?“ in Java

```java
public class WeatherIndicator {
    void rateWeather(int celsius) {
        String status;
        Color color;
        if (celsius < 5) {
            status = "Saukalt!";
            color = Color.BLUE;
        } else if (celsius >= 5 && celsius <= 20) {
            status = "Geht so!";
            color = Color.ORANGE;
        } else {
            status = "Urlaub!";
            color = Color.RED;
        }
    }
}
```

„Wie ist denn das Wetter?“ in Java

```
public class WeatherIndicator {
    void rateWeather(int celsius) {
        String status;
        Color color;
        if (
                                                  20) {
            color = Color.ORANGE;
        } else {
            status = "Urlaub!";
            color = Color.RED;
        }
    }
}
```



… in Kotlin

```kotlin
class WeatherIndicator {
    fun rateWeather(celsius: Int) {
        val status: String
        val color: Color
        if (celsius < 5) {
            status = "Saukalt!"
            color = Color.BLUE
        } else if (celsius >= 5 && celsius <= 20) {
            status = "Geht so!"
            color = Color.ORANGE
        } else {
            status = "Urlaub!"
            color = Color.RED
        }
    }
}
```

… in Kotlin

```
class WeatherIndicator {
    fun rateWeather(celsius: Int) {
        val status: String
        val color: Color
        if (celsius < 5) {
            status = "Saukalt!"
            color = Color.BLUE
        } else if (celsius >= 5 && celsius <= 20) {
            status
            color =
        } else {
            status
            color = Color.RED
        }
    }
}
```

Convert to range check
Add braces to 'else' statement
Replace '&&' with '||'
Replace overloaded operator with function call
Flip '<='

...mit Range Check

```
class WeatherIndicator {
    fun rateWeather(celsius: Int) {
        val status: String
        val color: Color
        if (celsius < 5) {
            status = "Saukalt!"
            color = Color.BLUE
        } else if (celsius in 5..20) {
            status = "Geht so!"
            color = Color.ORANGE
        } else {
            status = "Urlaub!"
            color = Color.RED
        }
    }
}
```

… mit Destructuring & `if`-Expression

```
class WeatherIndicator {
    fun rateWeather(celsius: Int) {
        val (status: String, color: Color) =
          if (celsius < 5) {
              Pair("Saukalt!", Color.BLUE)
          } else if (celsius in 5..20) {
              Pair("Geht so!", Color.ORANGE)
          } else {
              Pair("Urlaub!", Color.RED)
          }
    }
}
```

… mit Type Inference durch `Pair<A,B>`

```
class WeatherIndicator {
    fun rateWeather(celsius: Int) {
        val (status, color) =
          if (celsius < 5) {
              Pair("Saukalt!", Color.BLUE)
          } else if (celsius in 5..20) {
              Pair("Geht so!", Color.ORANGE)
          } else {
              Pair("Urlaub!", Color.RED)
          }
    }
}
```

… mit Type Inference durch `Pair<A,B>`

```
class WeatherIndicator {
    fun rateWeather(celsius: Int) {
        val (status, color) =
          if (celsius < 5) {
              Replace 'if' with 'when'
              Invert 'if' condition
              alt!", Color.BLUE)
          } else if (celsius in 5..20) {
              Pair("Geht so!", Color.ORANGE)
          } else {
              Pair("Urlaub!", Color.RED)
          }
    }
}
```

... mit Pattern Matching über `when`

```
class WeatherIndicator {
    fun rateWeather(celsius: Int) {
       val (status, color) = when {
         celsius < 5 -> Pair("Saukalt!", Color.BLUE)
         celsius in 5..20 -> Pair("Geht so!", Color.ORANGE)
         else -> Pair("Urlaub!", Color.RED)
       }
    }
}
```

EXTENSIONS

EXTENSION METHODS

```
fun String.lastChar() = this.get(this.length - 1)
```

- ... erlauben das Erweitern beliebiger Klassen
- `this` zeigt auf das erweitertete Objekt
- insbesondere im Android-Umfeld sehr hilfreich (z.B. Anko)

```
// Verwendung
val c = "Hallo Zuschauer!".lastChar()
```

EXTENSION PROPERTIES

```
// Extension property für das Android EditText Widget
var EditText.stringValue: String
    get() = text.toString()
    set(str) { setText(str)  }
```

```
// Verwendung
nameField.stringValue = "Max Muster"
```

LAMBDAS

FUNKTIONEN ALS FIRST-CLASS ELEMENTS

Funktionen sind Sprachelemente erster Klasse und können daher auch als **Variablen, Properties, Parameter** und **Rückgabewerte** genutzt werden.

FUNKTIONEN ALS FIRST-CLASS ELEMENTS

Funktionen sind Sprachelemente erster Klasse und können daher auch als **Variablen, Properties, Parameter** und **Rückgabewerte** genutzt werden.

Durch Verschachtelung lassen sich **Funktionen höherer Ordnung** *(Higher-order functions)* erschaffen

LAMBDAS: GRUNDLEGENDE SYNTAX

```
val implicitType = { x: Int, y: Int -> x + y }

var explicitType: (Int, Int) -> Int   // Funktionstyp als Variable !
explicitType = { x, y -> x + y }      // Lambda-Ausdruck
```

Lambdas – Syntactic sugar

```
view.setOnClickListener({ e -> doSomething(e) })
```

Lambdas – Syntactic sugar

```
view.setOnClickListener({ e -> doSomething(e) })
```

↓

```
view.setOnClickListener() { e -> doSomething(e) }
```

Lambdas – Syntactic sugar

```
view.setOnClickListener({ e -> doSomething(e) })
```

↓

```
view.setOnClickListener() { e -> doSomething(e) }
```

↓

```
view.setOnClickListener { e -> doSomething(e) }
```

Lambdas – Syntactic sugar

```
view.setOnClickListener({ e -> doSomething(e) })
```

↓

```
view.setOnClickListener() { e -> doSomething(e) }
```

↓

```
view.setOnClickListener { e -> doSomething(e) }
```

↓

```
view.setOnClickListener { doSomething(it) }
```

COLLECTIONS & LAMBDAS

```
val employees: List<Employee>

data class Employee(
   val city: City, val age: Int
)
```

```
// What's the average age of employees working in Prague?
employees.filter { it.city == City.PRAGUE }.map { it.age }.average()
```

Utilities „frei Haus“

Zahlreiche Extensions in der Standardbibliothek bereits enthalten.
Sie **erweitern bestehende Klassen und Interfaces** und sind
komfortabel über Code-Completion auffindbar.

```
List<String> list = Arrays.asList("a","b","c");
for (int index = 0; index < list.size(); index++) {
    if (index % 2 == 0)
        System.out.println(index + " -> " + list.get(index));
}
```

Sie helfen wesentlich beim **prägnanten**, dennoch **intuitiv lesbar** Code.

```
val list = listOf("a", "b", "c")
for (index in list.indices.filter { it % 2 == 0 }) {
    println("$index -> ${list.get(index)}")
}
```

Im Beispiel: `listOf`, `indices`, `filter`. Auch Bibliotheken für Android (Anko)

Kotlin Extensions für Collections

```
list.max
λ max() for Iterable<T> in kotlin.collections                    Int
λ maxBy {...} (selector: (Int) -> R) for Iterable<T> in kot…     Int?
λ maxWith(comparator: Comparator<in Int>) for Iterable<T> i…     Int?
^↓ and ^↑ will move caret down and up in the editor >>
```

- `filter()`
- `map()`
- `reduce()`
- `count()`
- `find()`
- `any()`
- `flapMap()`
- `groupBy()`
- …

Typischweise `inline`-Extensions, d.h. **kein Performance Overhead**!

HANDS-ON #4

LAMBDAS

SICHERHEIT

FIRST-CLASS IMMUTABLES

Nur-lesbare Variablen aufwandsfrei durch `val` statt `var`

Standard-Collection Interfaces (`List`, `Set`, ...) sind read-only

Zum ändern muss man z.B. `MutableList` statt `List nutzen`

Klassen & Methoden sind `final` by Default

```
open class ExtendableClass {
    open fun overridableMethod() { … }
}

class Derived: ExtendableClass() {
    override fun overridableMethod() { … }
}
```

FIRST-CLASS IMMUTABLES

Nur-lesbare Variablen aufwandsfrei durch `val` statt `var`

Standard-Collection Interfaces (`List`, `Set`, ...) sind read-only

Zum ändern muss man z.B. `MutableList` statt `List nutzen`

Klassen & Methoden sind `final` by Default

```
open class ExtendableClass {
    open fun overridableMethod() { … }
}

class Derived: ExtendableClass() {
    override fun overridableMethod() { … }
}
```

Mutige Entscheidung!

... Framework-Entwickler müssen Acht geben!

SESSION #3: IMMUTABLES

- `List` vs. `MutableList`
- `open class` & `override`

THE
BILLION-DOLLAR
MISTAKE
Tony Hoare

THE BILLION-DOLLAR MISTAKE

Tony Hoare

`null` und seine Folgen

MODERNERER ANSATZ

Mache `NullPointerException` zu einem compile-time Fehler, nicht run-time Fehler

NULLWERT-SICHERHEIT

Nur *explizit Nullwert-fähige Typen* akzeptieren `null`.
Ungeprüfte Zugriffe führen zu **Compile-Fehlern**!

```
var nullable: String? = null
var nonNullable: String = "Hi!"

// Ab hier compiliert nichts mehr …
nonNullable = nullable
nonNullable = null
nullable.isEmpty()
```

Nullwerte sind Bestandteil des Typsystems!

An den Schnittstellen baut Kotlin Prüfungen & `@NotNull` Annotationen ein.

Effizienter Umgang mit Nullwert-Typen

```
TextView child = null;
if (toolbar != null && toolbar.getChildAt(0) instanceof TextView) {
    child = (TextView) toolbar.getChildAt(0);
}
textView = child != null ? child : new TextView(getBaseContext());

if (textView instanceof EditText) {
    ((EditText) textView).selectAll();
}
```

```
val child = toolbar?.getChildAt(0) as? TextView
textView = child ?: TextView(baseContext)

if (child is EditText) {
    child.selectAll()
}
```

SESSION #4: NULLABILITY

- Safe-navigation, Elvis
- Smart-cast
- `!!`

Effizienter Umgang mit Nullwert-Typen

Safe nagivation operator

```
nullable = nullable?.toUpperCase()
```

Elvis operator

```
nonNullable = nullable ?: ""
```

Kotlin extension für Nullable-Typen

```
// equals/orEmpty für String?
nonNullable = nullable.orEmpty()
val b = nullable.equals("foo")
```

Smartcast

```
if (nullable != null) {
   nonNullable = nullable.trim()
}
```

Ich-weiss-es-besser™

```
//NPE-Gefahr voraus!
nullable!!.trim(3)
nonNullable = nullable!!
```

EXTENSION METHODS & `null`

Extensions können auf `null`-fähigen Typen definiert werden

```
val nullValue: String? = null

// Kotlin Extensions – hier keine NPE!
nullValue.equals("foo")
nullValue.toString()
```

KLEINES QUIZ?

INTEROPERABILITÄT

Kotlin vs. Java Types

Kotlin →	Java
`Any`	`java.lang.Object`
`Int`	`int`
`Int?`	`java.lang.Integer`
`Double`	`double`
`Double?`	`java.lang.Double`
`Array<Int>`	`Integer[]`
`IntArray`	`int[]`

`null`-Werte aus Java-Code

Alle Rückgabewerte aus Java-Code als `T?` zu verstehen wäre zwar richtig, aber extrem unkomfortabel.

Daher besitzt Kotlin das Konzept sog. *Platform Types* `T!`.

```
fun myStr4(): String {
    return System.getenv("_PWD")
}
```

```
getenv(name: String!)                    String!
getProperty(key: String!)                String!
getProperty(key: String!, def: .. String!
getProperties()                          Properties!
```

Kann nicht aus Kotlin heraus erzeugt werden!

↓

```
System.getenv("_PWD").length   // Potentiell NPE!
```

Java Interoperabilität – Allgemeines

Kotlin → Java

Klassenweises **Mischen von Java & Kotlin-Code** problemlos

- Jeglicher Java-Code kann elegant & direkt genutzt werden
- Inkrementelle Migration gut möglich
- Java 8 seit Kotlin 1.1, Java 9 Module seit Kotlin 1.2

Java → Kotlin

Elegante APIs erfordern gelegentlich sperrige Kotlin-Idiome.

Beispiel: `static`-Feld

kein direktes `static` in Kotlin, sondern *companion object*

```
public static final int VISIBLE = 0x00000000;
```

↓

```
companion object {
   @JvmField val VISIBLE: Int = 0x00000000
}
```

Kotlin Interop-Features

Element	Beschreibung
`@JvmStatic`	Statische Methode / Property
`companion object`	Singleton / static Object
`@JvmField`	Property als Java Feld verfügbar machen
`@JvmName`	Bytecode/JVM Namen steuern
`@JvmOverloads`	Optionale Parameter ausmultiplizieren
`@Throws`	Checked exceptions Signatur deklarieren
`@Nullable / @NotNull`	`T?` vs. `T`

OPTIONAL SESSION: INTEROP

- Bidirectional Java ↔ Kotlin
- Java-Interop & `Nullable`

```java
public class JavaClass {
    public static void main(String[] args) {
        new JavaClass().callKotlinClass();
    }

    void callKotlinClass() {
        KotlinClass kotlinClass = new KotlinClass(this);
        kotlinClass.callJavaClassMethod();
    }

    public String getValueFromJavaClass()     { return "Hi from Java!";   }

    public void printInJavaClass(String text) { System.out.println(text); }
}
```

```kotlin
class KotlinClass(val jclass: JavaClass) {
    fun callJavaClassMethod() {
        jclass.printInJavaClass("Kotlin got: ${jclass.valueFromJavaClass}" )
    }
}
```

EFFIZIENZ & IDIOME

AUSDRÜCKE & KONVENTIONEN

Ausdrucksstärke

Mehr Ausdrücke (`if`, `?:`, `?.`, ...)
Pattern matching (`when`)
String Templates
Syntactic Sugar

Standard-Bibliothek

`listOf(…)`, `repeat(3){…}`,
`with(x){…}`, `x.apply{…}`, ...

Konventionen

`public` sind Default
kein `new` – `;` optional
Operatoren & Dekomposition über Namen

KOTLIN & ANDROID

IM MAI 2017 – DIE ANTWORT

19.05.2017 07:52 **Benjamin Schmid** 173

Kommentar: Kotlin für Android – Googles fremde Lorbeeren

Nach dem Wechsel von Eclipse zu IntelIJ IDEA mag man die Kotlin-Unterstützung in Android als weiteren Ritterschlag verstehen. Doch der Geadelte ist hier nicht das Projekt Kotlin, sondern Google selbst – sagt Technologieexperte Benjamin Schmid.

Bei der Hauskonferenz I/O 2017 überraschte Google mit der Ankündigung, die Programmiersprache Kotlin zum offiziellen Bestand von Android zu er

https://heise.de/-3717940

ON SPEED

Typischer Android-Boilerplate ...

```
EditText editTitle = (EditText) v.findViewById(R.id.edit_title);
editTitle.setText(mItem.getTitle());

CheckBox enabledBox = (CheckBox) v.findViewById(R.id.enable_box);
enabledBox.setChecked(true);

Button createButton = (Button) v.findViewById(R.id.create_entry);
createButton.setOnClickListener(new OnClickListener() {
    @Override public void onClick(View button) {
        createElement();
    }
});
```

... ständige Lookups & Typecasts von Views über `findViewById()`.

Kotlin Extensions aktivieren

android-kotlin-helloworld.git > app > build.gradle

app ×

1:Project

7:Structure

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'

android {
    compileSdkVersion 25
    buildToolsVersion "25.0.0"
    defaultConfig {
        applicationId "de.bentolor.helloworld"
        minSdkVersion 19
        targetSdkVersion 25
```

add this line

VIRTUELLE PROPERTIES FÜR UI-ELEMENTE

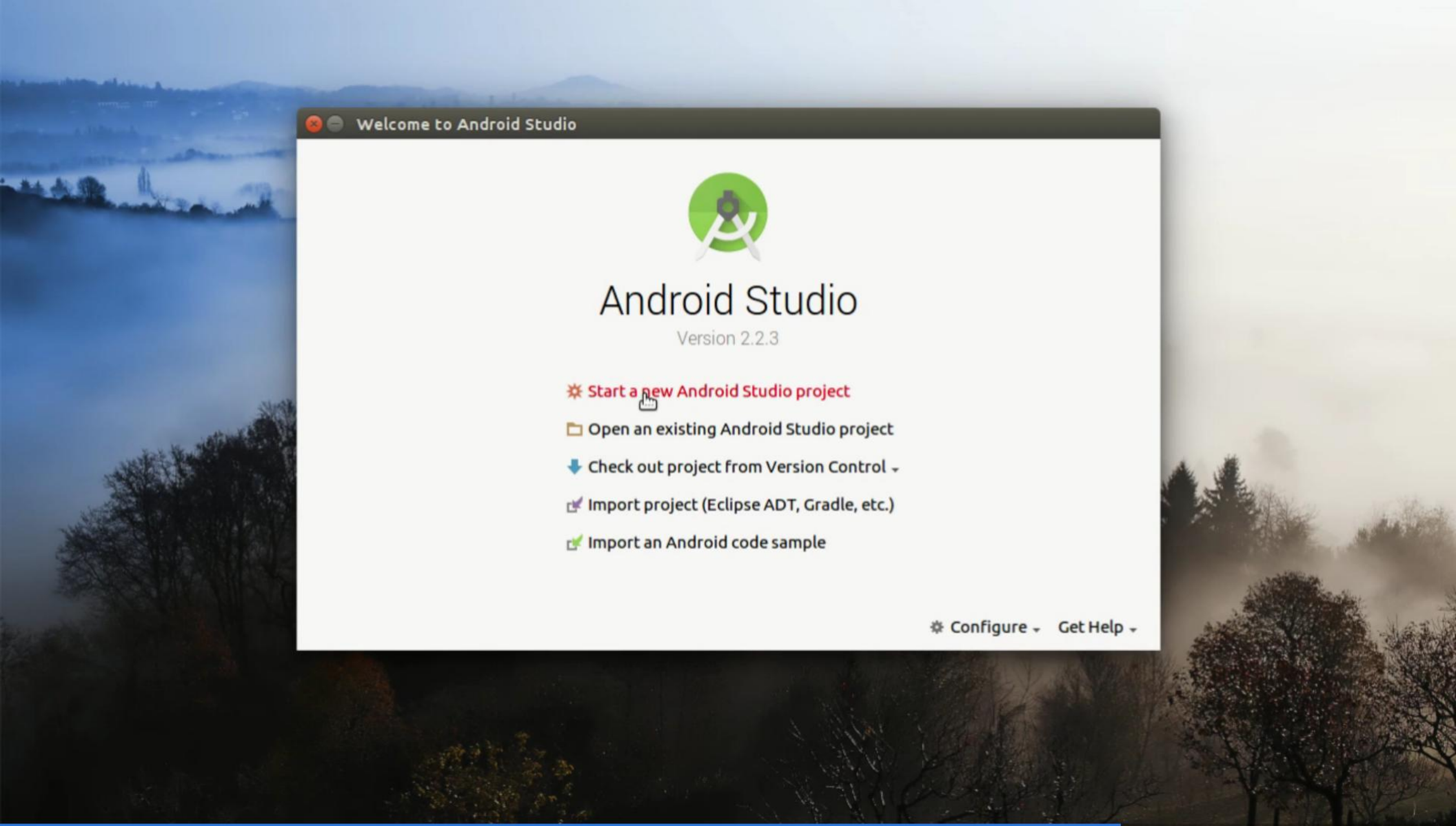
Das **Kotlin Android Extensions** erzeugt virtuelle `mylayout.xml`-Pakete

```
import kotlinx.android.synthetic.main.mylayout.*
```

```
edit_title.setText(mItem.title)
enable_box.isChecked = true
create_entry.setOnClickListener { createElement() }
```

Ein einfacher Import erlaubt **typsicheren** Zugriff auf alle darin enthaltene View-Elemente direkt als simples Property.

Ohne zusätzlichen Code, Annotations oder *Runtime!*


Welcome to Android Studio
Android Studio
Version 2.2.3
Start a new Android Studio project
Open an existing Android Studio project
Check out project from Version Control
Import project (Eclipse ADT, Gradle, etc.)
Import an Android code sample
Configure
Get Help

Nexus 4 | 25 | AppTheme | Language | 50%

6:00

KotlinGreetsAndroid

Grußtext

Anrede

wertes | Herr | Frau

Name

RelativeLayout
activity_hello_android

labelGruss

labelAnrede

RadioButton radioWertes | RadioButton radioHerr | RadioGroup anredeGroup | RadioButton radioFrau

labelName

EditText
editName

Properties

ID	editName
layout_width	match_parent
layout_height	wrap_content
EditText	
inputType	textPersonName
hint	
style	editTextStyle
singleLine	
selectAllOnF...	
TextView	
text	
text	
contentDesc...	
textAppea...	al.Medium.Inverse
fontFamily	sans-serif
typeface	none
textSize	18sp
lineSpacingE...	none
textColor	ttr/editTextColor
textStyle	B I Tt
textAlignment	

Hello Android: Code part #1

```kotlin
package de.bentolor.kotlingreetsandroid

import android.support.v7.app.AppCompatActivity
import android.os.Bundle
import kotlinx.android.synthetic.main.activity_hello_android.*
import java.util.*

class HelloAndroid : AppCompatActivity() {

    enum class Titel { wertes, Herr, Frau, Doktor }

    data class Gast(var titel: Titel = Titel.wertes,
                    var name: String,
                    val geburt: Date? = null) {
        val anrede: String
            get() = "Hallo $titel $name!"
    }

    private val gast = Gast(name = "Publikum")
```

Hello Android: Code part #2

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_hello_android)
    modelToView()

    editName.setOnEditorActionListener {...}
    anredeGroup.setOnCheckedChangeListener { group, id ->
        gast.titel = when(id) {
            radioHerr.id -> Titel.Herr
            radioFrau.id -> Titel.Frau
            else -> Titel.wertes
        }
        modelToView()
    }
}

fun modelToView() {
    labelGruss.text = gast.anrede
}
}
```

K - ADVANCED

Higher-order functions – Beispiel

```
class FirstClassFunction(
        val f1: (String) -> Int,
        val f2: (Int) -> Boolean) {

    fun strToBool(str: String): Boolean {
        val f: (String) -> Boolean = higherOrderFun()
        return f(str)
    }

    private fun higherOrderFun(): (String) -> Boolean {
        return { x -> f2(f1(x)) }
    }
}

fun main(args: Array<String>) {
    val c = FirstClassFunction(
                Integer::parseInt,
                { it % 2 == 0 }
            )
    val strings = listOf("2", "7", "8")
    println(strings.map(c::strToBool))
}
```

Higher-order functions – Beispiel

```
class FirstClassFunction(
        val f1: (String) -> Int,
        val f2: (Int) -> Boolean) {

    fun strToBool(str: String): Boolean {
        val f: (String) -> Boolean = higherOrderFun()
        return f(str)
    }

    private fun higherOrderFun(): (String) -> Boolean {
        return { x -> f2(f1(x)) }
    }
}

fun main(args: Array<String>) {
    val c = FirstClassFunction(
                Integer::parseInt,
                { it % 2 == 0 }
            )
    val strings = listOf("2", "7", "8")
    println(strings.map(c::strToBool))
}
```

← Funktion als **Parameter**

Higher-order functions – Beispiel

```
class FirstClassFunction(
        val f1: (String) -> Int,
        val f2: (Int) -> Boolean) {

    fun strToBool(str: String): Boolean {
        val f: (String) -> Boolean = higherOrderFun()
        return f(str)
    }

    private fun higherOrderFun(): (String) -> Boolean {
        return { x -> f2(f1(x)) }
    }
}

fun main(args: Array<String>) {
    val c = FirstClassFunction(
                Integer::parseInt,
                { it % 2 == 0 }
            )
    val strings = listOf("2", "7", "8")
    println(strings.map(c::strToBool))
}
```

← Funktion als **Parameter**

← Funktion als **Variable**

Higher-order functions – Beispiel

```
class FirstClassFunction(
        val f1: (String) -> Int,
        val f2: (Int) -> Boolean) {

    fun strToBool(str: String): Boolean {
        val f: (String) -> Boolean = higherOrderFun()
        return f(str)
    }

    private fun higherOrderFun(): (String) -> Boolean {
        return { x -> f2(f1(x)) }
    }
}

fun main(args: Array<String>) {
    val c = FirstClassFunction(
                Integer::parseInt,
                { it % 2 == 0 }
            )
    val strings = listOf("2", "7", "8")
    println(strings.map(c::strToBool))
}
```

← Funktion als **Parameter**

← Funktion als **Variable**

← Funk. als **Rückgabewert**

Higher-order functions – Beispiel

```
class FirstClassFunction(
        val f1: (String) -> Int,
        val f2: (Int) -> Boolean) {

    fun strToBool(str: String): Boolean {
        val f: (String) -> Boolean = higherOrderFun()
        return f(str)
    }

    private fun higherOrderFun(): (String) -> Boolean {
        return { x -> f2(f1(x)) }
    }
}

fun main(args: Array<String>) {
    val c = FirstClassFunction(
                Integer::parseInt,
                { it % 2 == 0 }
            )
    val strings = listOf("2", "7", "8")
    println(strings.map(c::strToBool))
}
```

← Funktion als **Parameter**

← Funktion als **Variable**

← Funk. als **Rückgabewert**

← **Kombination** via **Lambda**

Higher-order functions – Beispiel

```
class FirstClassFunction(
        val f1: (String) -> Int,
        val f2: (Int) -> Boolean) {

    fun strToBool(str: String): Boolean {
        val f: (String) -> Boolean = higherOrderFun()
        return f(str)
    }

    private fun higherOrderFun(): (String) -> Boolean {
        return { x -> f2(f1(x)) }
    }
}

fun main(args: Array<String>) {
    val c = FirstClassFunction(
                Integer::parseInt,
                { it % 2 == 0 }
            )
    val strings = listOf("2", "7", "8")
    println(strings.map(c::strToBool))
}
```

← Funktion als **Parameter**

← Funktion als **Variable**

← Funk. als **Rückgabewert**

← **Kombination** via **Lambda**

← Funk.-**Referenz**

Higher-order functions – Beispiel

```
class FirstClassFunction(
        val f1: (String) -> Int,
        val f2: (Int) -> Boolean) {

    fun strToBool(str: String): Boolean {
        val f: (String) -> Boolean = higherOrderFun()
        return f(str)
    }

    private fun higherOrderFun(): (String) -> Boolean {
        return { x -> f2(f1(x)) }
    }
}

fun main(args: Array<String>) {
    val c = FirstClassFunction(
                Integer::parseInt,
                { it % 2 == 0 }
            )
    val strings = listOf("2", "7", "8")
    println(strings.map(c::strToBool))
}
```

← Funktion als **Parameter**

← Funktion als **Variable**

← Funk. als **Rückgabewert**

← **Kombination** via **Lambda**

← Funk.-**Referenz**

← **Lambda**-Ausdruck

Higher-order functions – Beispiel

```
class FirstClassFunction(
        val f1: (String) -> Int,
        val f2: (Int) -> Boolean) {

    fun strToBool(str: String): Boolean {
        val f: (String) -> Boolean = higherOrderFun()
        return f(str)
    }

    private fun higherOrderFun(): (String) -> Boolean {
        return { x -> f2(f1(x)) }
    }
}

fun main(args: Array<String>) {
    val c = FirstClassFunction(
                Integer::parseInt,
                { it % 2 == 0 }
            )
    val strings = listOf("2", "7", "8")
    println(strings.map(c::strToBool))
}
```

← Funktion als **Parameter**

← Funktion als **Variable**

← Funk. als **Rückgabewert**

← **Kombination** via **Lambda**

← Funk.-**Referenz**

← **Lambda**-Ausdruck

→ *Was kommt raus?*

Einsatz in vielen Bibliotheksfunktionen

```
println("I will not ...")
repeat(3) { println("  ...repeat myself") }
```

How it works

```
public inline fun repeat(times: Int, action: (Int) -> Unit) {
    for (index in 0..times - 1) {
        action(index)
    }
}
```

`inline`: Wirkung wie Makro-Element. **Kein Funktionsoverhead!**

`with(…) { }`: Parameter wird im Block zum `this`

```
private fun toJSON(aha: Aha): JSONObject {
    with(JSONObject()) {
        put(JSON_ID, aha.id.toString())
        put(JSON_TITLE, aha.title)
        put(JSON_USEFUL, aha.isUseful)
        return this
    }
}
```

Varianten

`apply()`, `let()` und
`use()` als try-with-Equivalent

Typsichere DSLs über „Empfänger“-Objekt

Es ist möglich, Funktionssignaturen zu definieren, die Lambda-Ausdrücke erwarten bei denen `this` auf Zielobjekte eines bestimmten Typs zeigen.

Darüber sind typsichere DSLs möglich:

```
class HTML {
    fun body() { ... }
}

fun html(init: HTML.() -> Unit): HTML {
  val html = HTML()
  html.init()
  return html
}

html {
    body()
}
```

Typsichere DSLs über „Empfänger“-Objekt

Es ist möglich, Funktionssignaturen zu definieren, die Lambda-Ausdrücke erwarten bei denen `this` auf Zielobjekte eines bestimmten Typs zeigen.

Darüber sind typsichere DSLs möglich:

```
class HTML {
    fun body() { ... }
}

fun html(init: HTML.() -> Unit): HTML {
  val html = HTML()
  html.init()
  return html
}

html {
    body()
}
```

← Erwartet Funktion mit `this` vom Typ `HTML`

Typsichere DSLs über „Empfänger“-Objekt

Es ist möglich, Funktionssignaturen zu definieren, die Lambda-Ausdrücke erwarten bei denen `this` auf Zielobjekte eines bestimmten Typs zeigen.

Darüber sind typsichere DSLs möglich:

```
class HTML {
    fun body() { ... }
}

fun html(init: HTML.() -> Unit): HTML {
  val html = HTML()
  html.init()
  return html
}

html {
    body()
}
```

← Erwartet Funktion mit `this` vom Typ `HTML`

← Erstellung des *receiver object*

Typsichere DSLs über „Empfänger“-Objekt

Es ist möglich, Funktionssignaturen zu definieren, die Lambda-Ausdrücke erwarten bei denen `this` auf Zielobjekte eines bestimmten Typs zeigen.

Darüber sind typsichere DSLs möglich:

```
class HTML {
    fun body() { ... }
}

fun html(init: HTML.() -> Unit): HTML {
  val html = HTML()
  html.init()
  return html
}

html {
    body()
}
```

← Erwartet Funktion mit `this` vom Typ `HTML`
← Erstellung des *receiver object*
← Führe Lambda auf *receiver object* aus

Typsichere DSLs über „Empfänger“-Objekt

Es ist möglich, Funktionssignaturen zu definieren, die Lambda-Ausdrücke erwarten bei denen `this` auf Zielobjekte eines bestimmten Typs zeigen.

Darüber sind typsichere DSLs möglich:

```
class HTML {
    fun body() { ... }
}

fun html(init: HTML.() -> Unit): HTML {
  val html = HTML()
  html.init()
  return html
}

html {
    body()
}
```

← Erwartet Funktion mit `this` vom Typ `HTML`

← Erstellung des *receiver object*

← Führe Lambda auf *receiver object* aus

← Kurzform von `html({…})`

Typsichere DSLs über „Empfänger“-Objekt

Es ist möglich, Funktionssignaturen zu definieren, die Lambda-Ausdrücke erwarten bei denen `this` auf Zielobjekte eines bestimmten Typs zeigen.

Darüber sind typsichere DSLs möglich:

```
class HTML {
    fun body() { ... }
}

fun html(init: HTML.() -> Unit): HTML {
  val html = HTML()
  html.init()
  return html
}

html {
    body()
}
```

← Erwartet Funktion mit `this` vom Typ `HTML`
← Erstellung des *receiver object*
← Führe Lambda auf *receiver object* aus

← Kurzform von `html({…})`
← **Lambda**-Ausdruck für einen `HTML`-„Empfänger“

KOTLIN SCRIPTING

Scripts: `*.kts`. Ausführung: `$ kotlinc -script script.kts`

```
import java.io.File

val dir = if (args.isNotEmpty()) args[0] else "/"
val folders = File(dir).listFiles { file -> file.isDirectory() }
folders?.forEach { folder -> println(folder) }
```

Gradle 3.x setzt auf Kotlin!

Ab Gradle 3.x alternativ zu Groovy-`build.gradle` die Verwendung Kotlin-Codes (`build.gradle.kts`) inkl. Code-Completion & Typsicherheit möglich!

```
plugins {  java }

repositories { jcenter() }

java {
    sourceCompatibility = JavaVersion.VERSION_1_8
    targetCompatibility = JavaVersion.VERSION_1_8
}

dependencies {
    testCompile("org.junit.jupiter:junit-jupiter-api:5.2.0")
    testCompile("org.mockito:mockito-core:2.21.0")
    testCompile("org.glassfish.jersey.core:jersey-common:2.22.2")
    compileOnly("javax:javaee-api:8.0")
}
```

Gradle Kotlin DSL – `build.gradle.kts`

```kotlin
import org.jetbrains.kotlin.gradle.tasks.KotlinCompile

plugins {
    base
    kotlin("jvm") version "1.2.0" apply false
}

allprojects {
    group = "org.gradle.kotlin.dsl.samples.multiprojectci"
    version = "1.0"

    repositories {
        jcenter()
    }
}

// Configure all KotlinCompile tasks on each sub-project
subprojects {

    tasks.withType<KotlinCompile> {
        println("Configuring $name in project ${project.name}...")
        kotlinOptions {
            suppressWarnings = true
        }
    }
}

dependencies {
    // Make the root project archives configuration depend on every subproject
    subprojects.forEach {
        archives(it)
    }
}
```

Operatoren

Über Namenskonventionen. Feste Menge & Präzedenz

Operator		Methode
`a + b`	→	`a.plus(b)`
`a += b`	→	`a.plusAssign(b)`
`+a`	→	`a.unaryPlus()`
`++a`, `a++`	→	`a.inc()`
`a..b`	→	`a.rangeTo(b)`
`a[x]`	→	`a.get(x)`
`!a`	→	`a.not()`
...		...

GUI & `null` – Typische Konstellation

Werte für Initialisierung erst im Android-Callbacks bekannt ...

```
class EditPersonActivity : FragmentActivity() {

    var person: Person
    var anrede: String
    var imm: InputMethodManager
    …

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        val personId = savedInstanceState?.getInt("value",0)

        person = PersonManager.load(id)
        anrede = resources.getString(R.string.anrede)
        imm = getSystemService(INPUT_METHOD_SERVICE) as InputMethodManager
    }
}
```

GUI & `null` – Typische Konstellation

Werte für Initialisierung erst im Android-Callbacks bekannt ...

```
class EditPersonActivity : FragmentActivity() {

    var person: Person
    var anrede: String
    var imm: InputMethodManager
    …

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        val personId = savedInstanceState?.getInt("value",0)

        person = PersonManager.load(id)
        anrede = resources.getString(R.string.anrede)
        imm = getSystemService(INPUT_METHOD_SERVICE) as InputMethodManager
    }
}
```

So nicht möglich!

Lösungen: Verzögerte Initalisierung

`lateinit` Properties (empfohlen)

```
lateinit var person: Person
```

Lösungen: Verzögerte Initalisierung

`lateinit` Properties (empfohlen)

```
lateinit var person: Person
```

Delegation des Properties

```
var anrede: String by Delegates.notNull()
```

Lösungen: Verzögerte Initalisierung

`lateinit` Properties (empfohlen)

```
lateinit var person: Person
```

Wirft sprechende Exception, falls ohne Initialisierung gelesen wird!

Delegation des Properties

```
var anrede: String by Delegates.notNull()
```

Lazy-Initalisierung

```
val imm: InputMethodManager by lazy {
    getSystemService(INPUT_METHOD_SERVICE) as InputMethodManager
}
```

CALL-SIDE VARIANCE

Java

```
public interface Stream<T> {
    <R> Stream<R> map(Function<? super T, ? extends R> mapper);
}
```

Kotlin-Ansatz mit `in T` & `out T`

```
interface Function1<in P, out R> {
    operator fun invoke(p: P): R
}
```

INLINING

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

```
fun foo(l: Lock) {
    println("Before sync")

    synchronized(l) {
        println("Action")
    }

    println("After sync")
}
```

Generated code:

```
fun __foo__(l: Lock) {
    println("Before sync")
    l.lock()
    try {
        println("Action")
    } finally {
        l.unlock()
    }
    println("After sync")
}
```

code of the calling `foo` function

inlined code of the `synchronized` function

inlined code of the lambda body

INFIX NOTATION

```
infix fun Int.shl(x: Int): Int {
    …
}

1 shl 2        // als Equivalent zu 1.shl(2)
```

1. Sind Member-Funktionen oder Extension Methods
2. Haben einen einzigen Parameter
3. Und sind mit dem `infix` Schlüsselwort markiert

HACKING

REAL-LIFE APPS

ANDROID: AHA-MANAGER

https://github.com/bentolor/kotlin-android-aha-manager

```
git clone https://github.com/bentolor/demo-android-aha-manager.git
```

```
git checkout java-only
```

Aufgaben

- Konvertiere nach Kotlin

GIT CITY

https://github.com/bentolor/gitcity

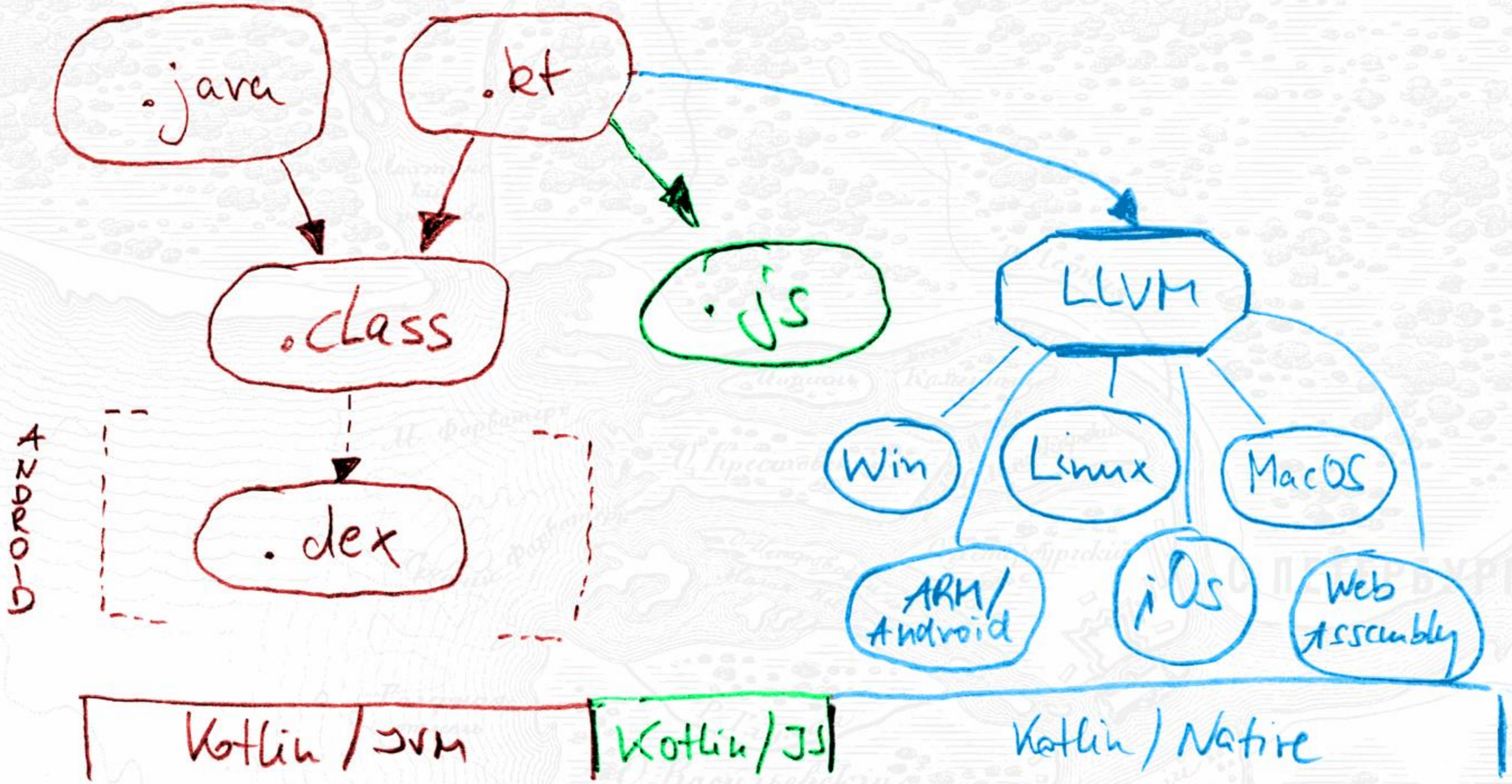
```
git clone https://github.com/bentolor/gitcity.git
```

Aufgaben

- Beautyfing & Inspection warnings wegbekommen
- Support für Remote Repositories

FAZIT
&
Ausblick

KOTLIN/NATIVE (EXPERIMENTELL)



LLVM-basierte Kompilierung von Kotlin-Code auf Plattformen ohne VM.

EXPERIMENTELLE FEATURES

Coroutines (1.1)

Sequentielle Programmierung von asynchroner Logik. Kann `async`/`await` von C#, `channel` und `select` von Go und `generator`/`yield` von C# bzw. Python nachbilden.

Multiplatform Projects (1.2)

Erlaubt die Compilation & **Wiederverwendung des selben Codes für verschiedene Zielplattformen**.

`expect` definiert Platzhalter für Deklarationen, die mittels `actual` Plattform-spezifisch implementiert werden.

IM BROWSER AUSPROBIEREN

Kotlin

LEARN CONTRIBUTE TRY ONLINE

JB

Shortcuts | Convert from Java | Fullscreen

Examples > Hello, world! > Simplest version > Simplest version.kt

- Examples
  - Hello, world!
    - Simplest version
      - Simplest version.kt
    - Reading a name from ...
    - Reading many names ...
    - A multi-language Hello
    - An object-oriented Hello
  - Basic syntax walk-through

Save | Save as | Arguments | JVM | Run

Program arguments

```
1 /**
2  * We declare a package-level function main which returns Unit and takes
3  * an Array of strings as a parameter. Note that semicolons are optional.
4  */
5
6 fun main(args: Array<String>) {
7     println("Hello, world!")
8 }
```

→ http://try.kotl.in/

Lernen: Kotlin Koans

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help

Project

- Kotlin Koans
  - Introduction
    - Hello, world!
      - Task.kt
    - Java to Kotlin conversion
    - Named arguments
    - Default arguments
    - Lambdas
    - Strings
    - Data classes
    - Nullable types
    - Smart casts
    - Extension functions
    - Object expressions
    - SAM conversions
    - Extensions on collections
  - Conventions
  - Collections
  - Properties
  - Builders
  - Generics

Task.kt ×

Congratulations!

```
fun start(): String = "OK"
```

Simple Functions

Take a look at function syntax and make the function `start` return the string `"OK"`.

In the tasks the function `TODO()` is used that throws an exception. Your job during the koans will be to replace this function invocation with a meaningful code according to the problem.

1 of 42 tasks completed

9: Version Control | Terminal | 0: Messages | 4: Run | 5: Debug | 6: TODO | Event Log

All files are up-to-date (moments ago) 2:1 n/a UTF-8 218 of 4062M

QUELLEN & MATERALIEN

Materialien für den Einstieg

- kotlinlang.org
- Kotlin Koans
  auch online: try.kotl.in in IntelliJ: Kotlin Edu Plugin
- Ein Überblick über Java-Alternativen für den industriellen Einsatz, Benjamin Schmid

Referenzen

- Sprachreferenz

Verzeichnisse

- Awesome Kotlin
  `http://kotlin.link`
- OSS Projects & Libraries

Diverses

- @kotlin, @bentolor, github.com/bentolor

Nennenswerte Frameworks

- KAndroid
- Fuel (HTTP f. Android)
- RxKotlin (Reactive)
- spek (Testing)
- kovenant
  (Android Promises)

Kotlin-optimierte Addons

- SparkJava
- jooby

Bildnachweis: Kotlin Islands Maps, Atomic bomb explosion, Android Robot, Kotlin Logo, Hand-written arrows