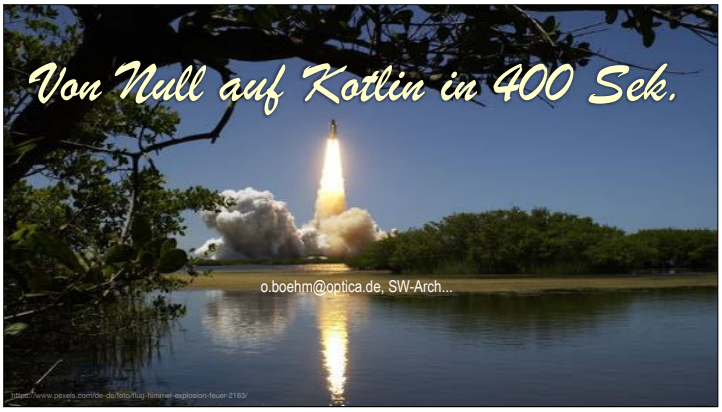


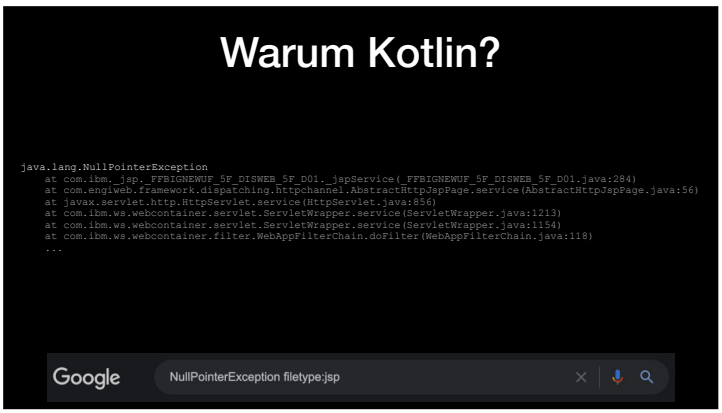
Willkommen zu „von Null auf Kotlin in 400 Sek.“ oder auch „von nall auf kotlin in 400 sekunden“
Null ist hier bewußt doppeldeutig:

1. Null nimmt in Kotlin eine Sonderstellung ein - den Wert gibt es in Kotlin (so gut wie) nicht
2. **Wie** komme ich mit null Vorkenntnissen in Kotlin schnell zu einem funktionierenden Kotlin-Projekt? (-> Ziel des Vortrags)

Disclaimer: viele Wege führen nach Kotlin. Ich beschreibe hier nur einen möglichen, der für mich funktioniert hat.
Hinweis: ursprünglich war der Vortrag vor 2 Jahren als Pecha-Kucha-Vortrag geplant und konzipiert (400 Sek.)
Nach der Einreichung wurde aber aus Lizenz-Gründen die Pecha-Kucha-Track gestrichen.
jetzt: 400-Sek.-Demo + Vortrag

zu mir:

- seit 1999 JUGS-Mitglied
- Autor mehrerer Bücher
- seit 2015 bei Optica Abrechnungszentrum als Bindeglied zwischen alter Welt (SW-Archäologe) und neuer Welt (SW-Architekt)



Nach Groovy und Scala schon wieder ein Java-Killer? Und warum Kotlin?
Wichtigster Grund für mich: Kotlin ist Null-Safe und kennt keine NPEs

zum Ausprobieren: googlen nach „NullPointerException filetype:jsp“
Ergebnis: http://booking.tirrenia.it/BookingOnLineWeb/servlet/AdapterHTTP?PAGE=EMISSIONE_BIGLIETTO_NUOVO_SITO_WEB&NEW_SESSION=TRUE

Zutaten

PROJEKT

Was wird gebraucht?
- IDE -> IntelliJ
- einfaches Projekt
- Unit-Tests

---

jFachwert

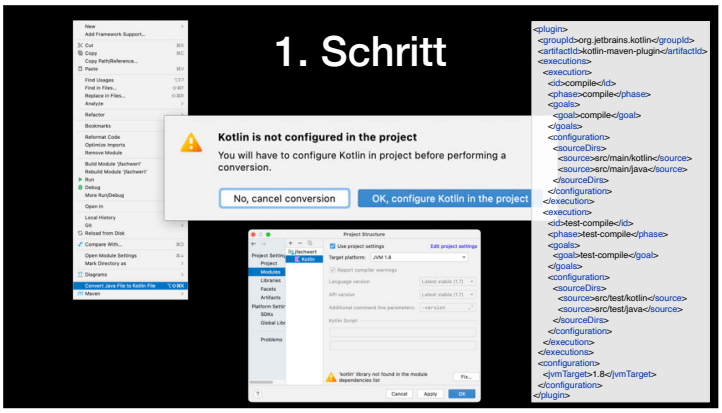
> *Fachwerte sind Erweiterungen von primitiven Datentypen objektorientierter Programmiersprachen… Im DDD ist dieses Konzept als Value Object bekannt.*
>
> —*Wikipedia*
> *Werkzeug- und Materialansatz*

Warum jFachwert?
- Einfaches Projekt
- Gute Testabdeckung

Zu Fachwerten: der Begriff stammt aus WAM (Werkzeug, Automat, Material) und ist vergleichbar mit „Value Objects“ aus DDD (d.h. einfache Objekte)

Vorteil: Typsicherheit. Wenn ich z.B. eine Bankverbindung mit IBAN und BIC anlege, kann ich folgende Vorteile:
- keine Verwechslungsgefahr der Argumente (im Gegensatz zur Übergabe zweier String)
- Validierung kann von Fachwerte übernommen werden

Die Idee zu einer eigenen Bibliothek stammt aus meiner Zeit im Bankbereich bei RWG, später Fiducia.



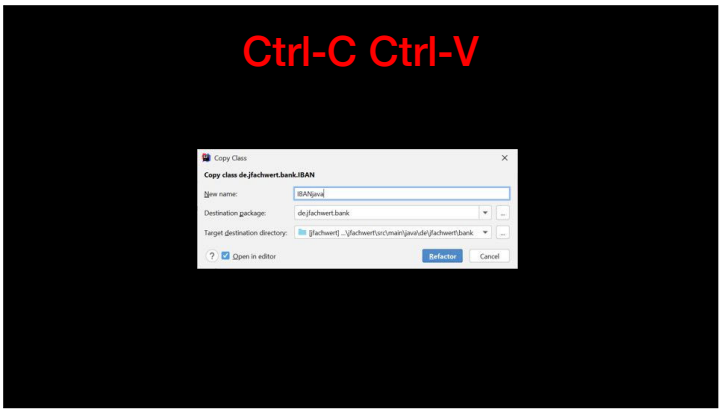
Java-Projekt in Kotlin-Projekt umwandeln:

- Projekt (oder IBAN.java) selektieren
- Convert Java File to Kotlin File => POM wird automatisch angepasst
- kotlin.version von 1.0.0 auf 1.7.0 anheben (s. Preferences / Kotlin-Plugin
- src/main/kotlin als Source-Dir anlegen
- Open Module Settings
- jfachwert selektieren + Kotlin + fix... (Kotlin library not found)

Erste einfache Klasse auswählen: IBAN
Danach lässt man den Unit-Test für die Klasse (IBANTest) und überprüft die Testabdeckung:

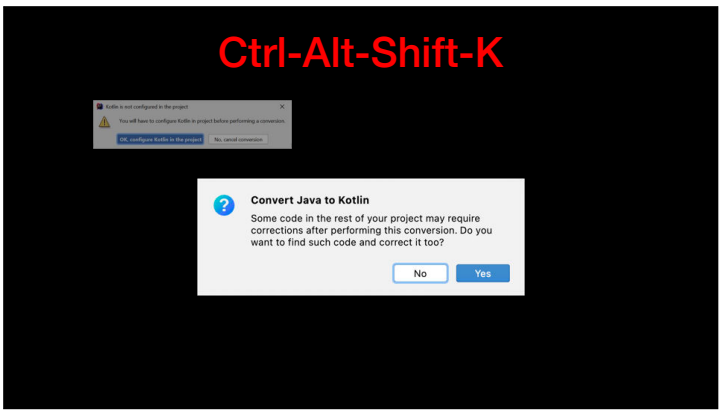
- ggfs. Unit-Tests ergänzen oder
- toten Code entfernen (oder als @deprecated markieren)

Dies sollte vor der Umwandlung nach Kotlin erfolgen (einfacherer Vorher-/Nachher-Vergleich)



IBAN-Klasse nach IBANjava kopieren kopieren (Ctrl-C / Ctrl-V), aber nicht einchecken
Warum? Die kopierte Klasse dient als Referenz für Unklarheiten, während Unit-Test zeigt nach wie vor IBAN testet

---



IBAN-Klasse selektieren und „Convert Java file to Kotlin file“ (bzw. Ctrl-Alt-Shift-K (PC) bzw. Option-Shift-Command-K (Mac))
Falls man Projekt noch nicht nach Kotlin konvertiert hat, kommt beim 1. Mal der Hinweis mit„OK, configure Kotlin in the project“
Wichtig: Kotlin-Libs müssen im Projekt hinzugefügt werden (notfalls manuell) – sonst kommt immer wieder „OK, configure Kotlin in the project“
Danach haben wir IBAN.kt, das noch nachbearbeitet werden muss.

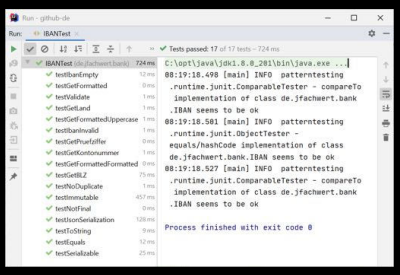
? !!

ⓘ Error:(137, 5) Kotlin: Class 'Validator' is not abstract and does not implement abstract member public abstract fun validate(value: String?): String? defined in de.jfachwert.SimpleValidator

Erste Herausforderung: was bedeutet die Fehlermeldung mit „String?“?

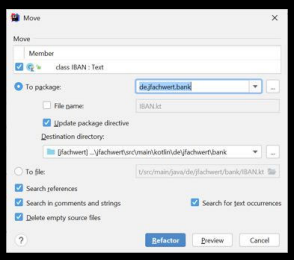
- Datentyp „String“: Zeichenkette, die nicht null ist
- Datentyp „String?“: Zeichenkette, die auch null sein darf (nullable)

Tipp: „?“ am Ende weglassen, weil das genau das Verhalten ist, das man will (keine null-Werte)
Falls Variable nullable ist, gibt es beim Zugriff einen Compile-Fehler, wenn nicht auf null überprüft wird. Diese Prüfung kann explizit mit !! übergangen werden.



Nach der Konvertierung und Korrektur der Compile-Fehler werden die Unit-Test gestartet.
Sollten Fehler auftreten, hat man zur Unterstützung der Fehlersuche noch die kopierte Klasse (IBANjava).
Fehler können sich z.B. durch andere Defaults (s. später) einschleichen.

Trennung java → kotlin



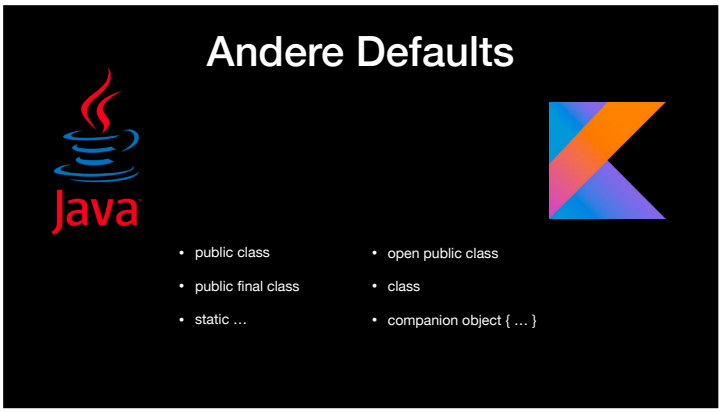
Die Trennung Kotlin-Klassen von den Java-Klassen hilft,

- die Übersicht zu wahren
- und notfalls auch einen Misch-Betrieb zu erleichtern (wenn man denn will)

Dazu legt man src/main/kotlin in IntelliJ als Source-Verzeichnis an und verschiebt IBAN.kt dorthin.

Zusätzlich muss noch kotlin-maven-plugin und compiler-maven-plugin im POM angepasst werden:

```
<plugin>
  <groupId>org.jetbrains.kotlin</groupId>
  <artifactId>kotlin-maven-plugin</artifactId>
  <version>${kotlin.version}</version>
  <executions>
    <execution>
      <id>compile</id>
      <phase>compile</phase>
      <goals>
        <goal>compile</goal>
      </goals>
      <configuration>
        <sourceDirs>
          <source>src/main/kotlin</source>
          <source>src/main/java</source>
        </sourceDirs>
      </configuration>
    </execution>
    <execution>
      <id>test-compile</id>
      <phase>test-compile</phase>
      <goals>
        <goal>test-compile</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <jvmTarget>1.8</jvmTarget>
  </configuration>
</plugin>
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.6.1</version>
  <executions>
    <execution>
      <id>default-compile</id>
      <phase>none</phase>
    </execution>
    <execution>
      <id>default-testCompile</id>
      <phase>none</phase>
    </execution>
    <execution>
      <id>compile</id>
      <phase>compile</phase>
      <goals>
        <goal>compile</goal>
      </goals>
    </execution>
    <execution>
      <id>testCompile</id>
      <phase>test-compile</phase>
      <goals>
        <goal>testCompile</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <source>${java.version}</source>
    <target>${java.version}</target>
    <encoding>${project.build.sourceEncoding}</encoding>
  </configuration>
</plugin>
```



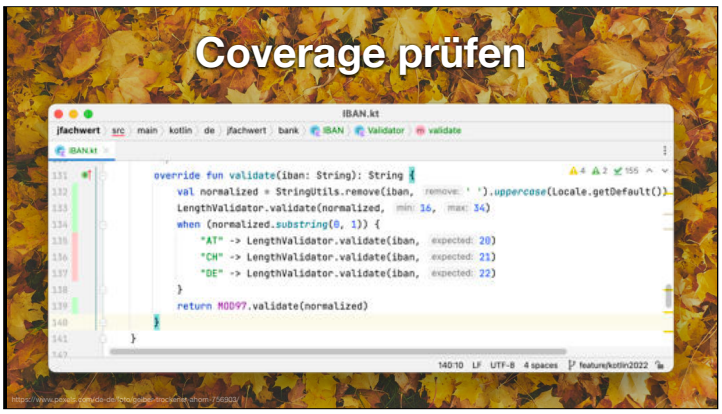
Die Defaults von Kotlin sind meist sinnvoller als die Java-Defaults, können aber für Überraschungen sorgen:

- per Default final
- per Default public
- kein static, dafür companion-Objekt (companion = Compagnion, Kumpel) + Annotations für Java (@JvmStatic, @JvmField, …)

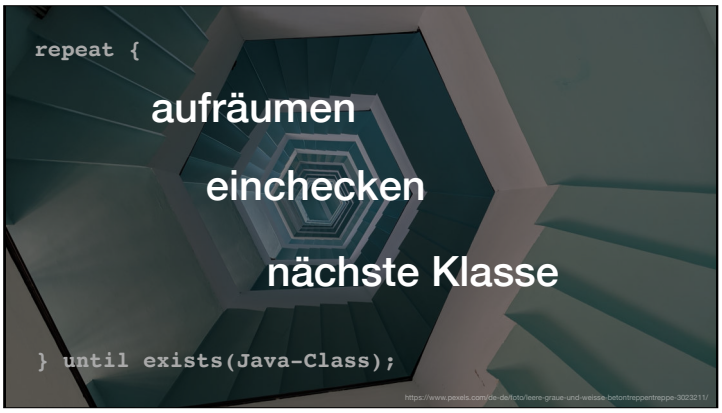
Tipp: Soll volle Kompatibilität zu Java gewährleistet werden, empfiehlt es sich, die Unit-Tests in Java zu belassen.

Im Gegensatz zu Java sind auch die Inhalte der Collections-Klassen nicht veränderbar (immutable), sondern es gibt die Mutable-Collections als Gegenstück bzw. Ergänzung.

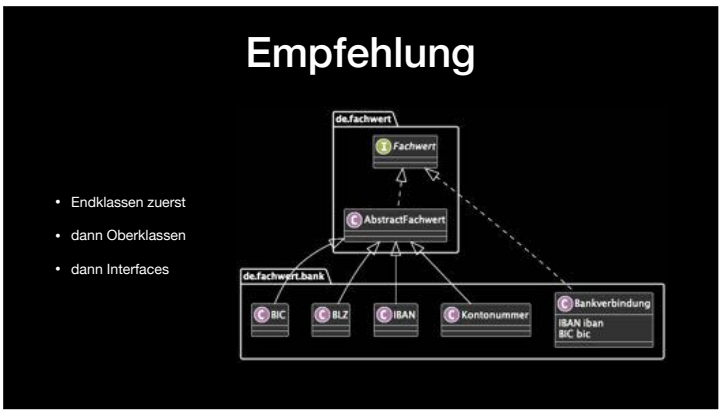
Da null als Argument in 99% aller Fälle keinen Sinn macht, sollten bei den Methoden die Argumente überprüft werden, ob „nullable“ („?“ am Ende) Sinn macht.
Meist kann „String?“ durch „String“ ersetzt werden.



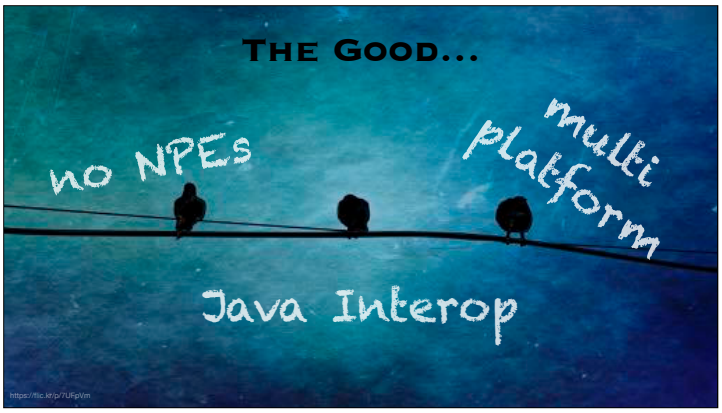
Bei der Überprüfung der Kotlin-Variante sind vor allem die nicht abgedeckten Zweige interessant und sollten nochmal überprüft werden, ob sie sich wie die Java-Variante verhalten.
Evtl. müssen Unit-Tests ergänzt werden, besser ist es aber, dies vor der Konvertierung zu machen.



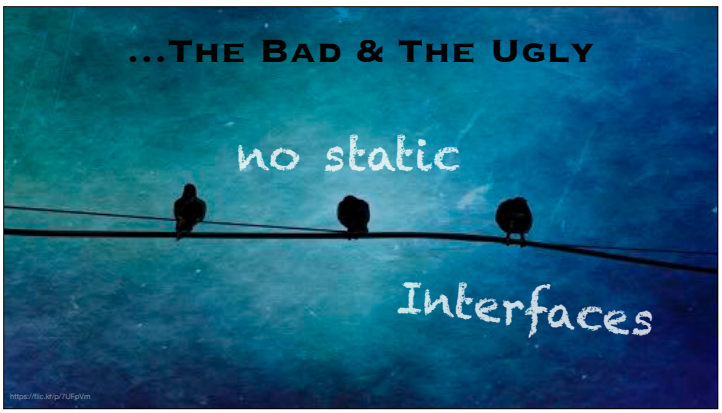
Als nächstes räumt man auf (IBANjava löschen), checkt die Klasse ein und nimmt sich die nächste Klasse vor.
Wichtig ist, dass man mit den einfachen Klassen anfängt, um sich nach und nach das nötige Kotlin-Knowhow für kniffligere Fälle aufzubauen.
Unit-Tests können in Java verbleiben, vor allem, wenn Java-Kompatibilät gewährleistet werden soll (z.B. für Libraries).



Allgemeine Empfehlung ist, mit den einfachsten Klassen anzufangen. Dies sind immer Endklassen.
Danach kommen die schwierigere Fölle.



Erfahrungen: @NotNull ist der Default, aber: man muss sich Gedanken um null (?) oder nicht null machen
Mehr Plattformen bedautet, dass Kotlin nicht nur die Java-VM unterstützt, sondern auch nach JavaScript übersetzt werden kann (Web). Auch Übersetzung nach „native“ (LLVM based backend) ist möglich (ähnlich GraalVM), was z.B. für Mobile-Bereich (Android, iOS) interessant ist.



Die Konstruktor-Syntax in Kotlin ist etwas gewöhnungsbedürftig.
Kritischer sind aber einige Fallstricke wie:

- kein static, stattdessen @JvmStatic
- Interface nicht vollständig kompatible -> https://youtrack.jetbrains.com/issue/KT-4779 (Default-Methoden)

Fazit: So wie C++ C (und B, A) ablöste, wird in Zukunft Kotlin Java ablösen oder anders ausgedrückt:
Java ist das neue COBOL und Kotlin die Zukunft.
Warum? 2 Gründe sprechen dafür:

- Android-Entwicklung findet bereits mit Kotlin statt
- 2 wichtige Firmen stehen hinter Kotlin: JetBrains + Google

Für Fragen bin ich erreichbar auf Twitter unter @oboehm oder per Mail an o.boehm@optica.de