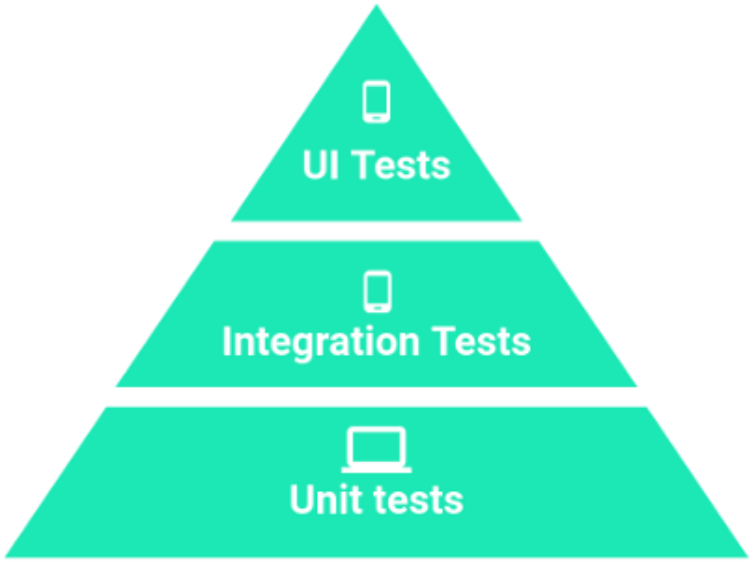
Mobile Programming

Android 7 – Testing & Android Build System

Inhalt

- Testing
  - Testing Support Library
  - Instrumentation Tests mit Junit 4
  - UI Testing mit Espresso

- Android Build System
  - Gradle Basics
  - Build Types: Debug und Release-Version
  - Product Flavors: App-Varianten



Testing

https://developer.android.com/training/testing/fundamentals

Test Before Release!

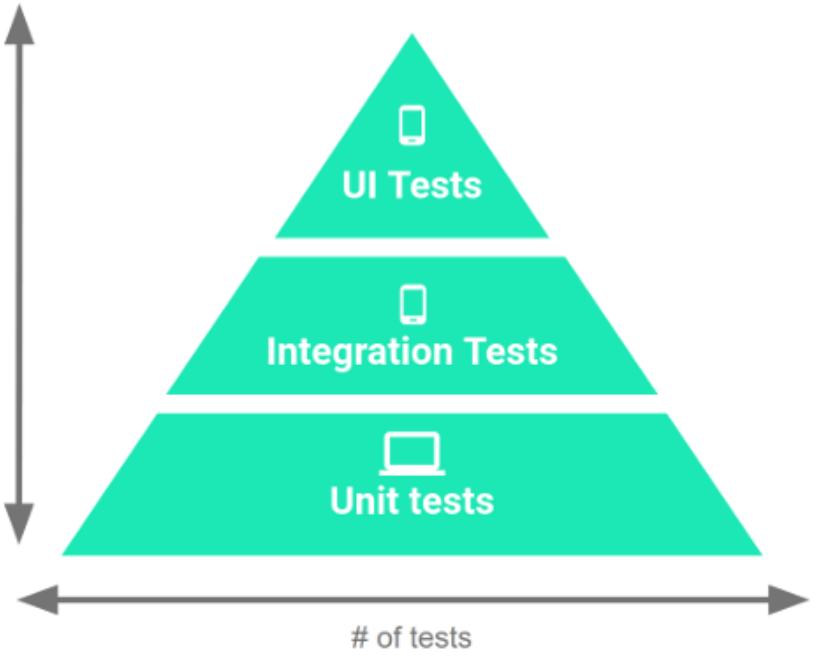
- Apps manuell & automatisch testen
  - sollte selbstverständlich sein ☺
- Manuell auf mindestens einem Gerät
  - Besser: mehrere Geräte...
    - ...mit unterschiedlicher Bildschirmauflösung, inkl. Tabletts
    - ...mit unterschiedlichen Android-Versionen
    - ...von unterschiedlichen Herstellern

➔Fazit: Sicherstellen, dass ein App auf vielen Geräten & Android-Versionen läuft, bedeutet viel Aufwand!

Hinweis/Tipp: Nicht unterschätzen für professionelle/kommerzielle Apps!!

Testing Pyramide

- Grosse Tests: E2E / UI Tests
  - Wenige, da langsam
- Mittlere Tests: Integration Tests
  - Pro für jedes Modul in einem
  - E2E gibt es ein mittleren Test
- Kleine Tests: Unit Tests
  - Viele, da schnell

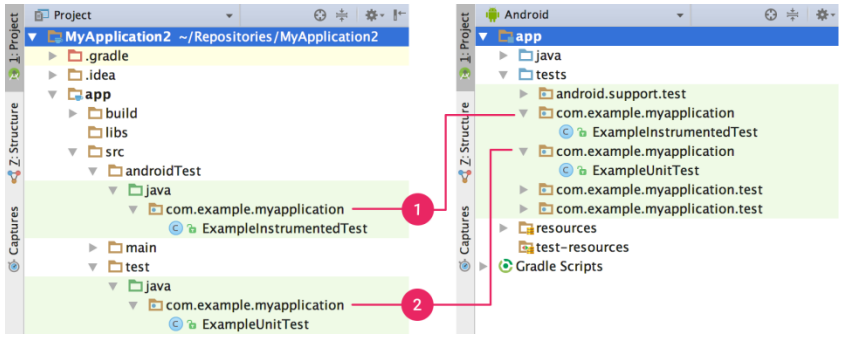


https://developer.android.com/training/testing/fundamentals

Automatisiertes Testen: Unit Tests

- Android unterstützt folgende »Geräte» typen
  - Real (zuverlässig, aber langsam)
  - Simuliert (unzuverlässig, aber schnell) --> robolectric
  - Virtuelle Geräte (Emulator) → Balance zwischen real und simuliert
- New-Project-Wizard generiert Test-Klasse
  - Unterschiedliche Source-Roots für Unit-Tests und Integration-Tests nutzen!
    - `src/test/java/…`
    - `src/androidTest/java/…`

Für normale, lokale Unit-Tests *ohne* Platform-Abhängigkeiten



Für Instrumentation-Tests *mit* Platform-Abhängigkeiten (laufen im Emulator oder auf HW Device)

Automatisiertest Testen: Unit Tests

- Lokale Unit Tests
  - Lokal auf der Entwickler Maschine
  - schnell
- Lokale Unit-Tests mit Robolectric (for JVM-powered development machine)
  - Simuliert, nicht ganz so schnell
  - Lokal auf der Entwickler Maschine
- Android Instrumentation Tests
  - Gerät oder Emulator
  - langsam
- (Wie oft) gut dokumentiert in der Android-Doku:

  https://developer.android.com/training/testing
- Ausserdem in Android Code Lab:
  https://developer.android.com/codelabs/advanced-android-kotlin-training-testing-basics#0

1) Klassische Unit-Tests

- Reine Java-Unit-Tests sind (natürlich) möglich,
  d.h. von Android-Platform unabhängige Tests mit JUnit

```
@Test
fun getActiveAndCompletedStats_noCompleted_returnsHundredZero() {
    val tasks = listOf(
        Task("title", "desc", isCompleted = false)
    )
    // When the list of tasks is computed with an active task
    val result = getActiveAndCompletedStats(tasks)

    // Then the percentages are 100 and 0
    assertThat(result.activeTasksPercent, `is`(100f))
    assertThat(result.completedTasksPercent, `is`(0f))
}
```

2) Lokale Unit-Tests mit Robolectric

- Macht aus 3) eine 1) dank Robolectric
  - `gradle.build` Dependencies mit Robolectric erweitern
    - `testImplementation "org.robolectric:robolectric:4.3.1"`
  - Robolectric simuliert das Android Environment

```
private lateinit var tasksViewModel: TasksViewModel

@Before
fun setupViewModel() {
    tasksViewModel = TasksViewModel(ApplicationProvider.getApplicationContext())
}


@Test
fun setFilterAllTasks_tasksAddViewVisible() {

    // When the filter type is ALL_TASKS
    tasksViewModel.setFiltering(TasksFilterType.ALL_TASKS)

    // Then the "Add task" action is visible
    assertThat(tasksViewModel.tasksAddViewVisible.getOrAwaitValue(), `is` (true))

}
```

androidx.test.espresso:espresso-core:3.4.0

3) Android Instrumentation-Tests

- Laufen im Emulator oder auf einem phys. Gerät
  - Benutzen das «echte» Android-Laufzeitsystem
  - Instr.-Tests liegen in Src-Root `src/androidTest/java`
  - `gradle.build` Settings der App

```
dependencies {
    androidTestImplementation 'androidx.test:runner:1.4.0'
    androidTestImplementation 'androidx.test:rules:1.4.0'
    // Optional -- Hamcrest library
    androidTestImplementation 'org.hamcrest:hamcrest-library:1.3'
    // Optional -- UI testing with Espresso
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'
    // Optional -- UI testing with UI Automator
    androidTestImplementation 'androidx.test.uiautomator:uiautomator:2.2.0'
}
```

Sehr wichtig, sonst werden Tests als normale Unit-Tests lokal ausgeführt

```
android {
    defaultConfig {
        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }
}
```

Instrumentation-Test: Beispiel mit Parcelable Interface

```
// @RunWith is required only if you use a mix of JUnit3 and JUnit4.
@RunWith(AndroidJUnit4::class)
@SmallTest
class LogHistoryAndroidUnitTest {
    private lateinit var logHistory: LogHistory

    @Before
    fun createLogHistory() {
        logHistory = LogHistory()
    }
    @Test
    fun logHistory_ParcelableWriteRead() {
        val parcel = Parcel.obtain()
        logHistory.apply {
            // Set up the Parcelable object to send and receive.
            addEntry(TEST_STRING, TEST_LONG)

            // Write the data.
            writeToParcel(parcel, describeContents())
        }
        // After you're done with writing, you need to reset the parcel for reading.
        parcel.setDataPosition(0)

        // Read the data.
        val createdFromParcel: LogHistory = LogHistory.CREATOR.createFromParcel(parcel)
        createdFromParcel.getData().also { createdFromParcelData: List<Pair<String, Long>> ->

            // Verify that the received data is correct.
            assertThat(createdFromParcelData.size).isEqualTo(1)
            assertThat(createdFromParcelData[0].first).isEqualTo(TEST_STRING)
            assertThat(createdFromParcelData[0].second).isEqualTo(TEST_LONG)
```

UI-Tests mit Espresso (Verhaltenstests)

- Espresso ist ein «white box» UI-Test Framework für Android
  - Es wird nur «beobachtbares» Verhalten getestet (Stimulus und Reaktion), keine Implementationsdetails
  - Alle Espresso-Tests sind Integration-Tests
  - UI-Steuerungsaufrufe erfolgen synchron (d.h. Testcode fährt erst weiter, wenn die Aktion ausgeführt wurde)
  - Arbeitet mit *Hamcrest*-Matchern
- Testen
  - 1. View finden, 2. Aktion ausführen, 3. Zustand testen

UI-Test: Beispiel mit Espresso

```
import static org.hamcrest.text.IsEqualIgnoringCase.equalToIgnoringCase;
import static org.hamcrest.text.StringContainsInOrder.stringContainsInOrder;

@RunWith(AndroidJUnit4.class)
public class RegistrationEspressoTest {

    // specify start activity
    @Rule
    public ActivityTestRule<RegisterActivity> activityRule = new ActivityTestRule<>(RegisterActivity.class);

    @Test
    public void inputUsernameFreddy_clickRegister_resultsInAlreadyTakenMessage() throws Exception {
        onView(withHint("Username")).perform(typeText("freddy"),closeSoftKeyboard());
        onView(withText(equalToIgnoringCase("Register"))).perform(click());

        onView(withId(R.id.lbl_error))
                .check(matches(withText(stringContainsInOrder(Arrays.asList("username","already taken")))));
    }

    @Test
    public void registerOk_showsWelcomeTextAndRandomBackground() throws Exception {
        onView(withHint("Username")).perform(typeText("alfredo"));
        onView(withHint("Password")).perform(typeText("alFr.3d0"));
        onView(withId(R.id.btn_register)).perform(click());

        // this will open a new activity, espresso will wait until it is ready
        onView(withText("Welcome alfredo")).check(matches(isDisplayed()));

        // wait a bit until background is shown
        onView(withId(R.id.img_avatar)).check(matches(withAnyDrawable()));
    }

}
```

Statische Imports für Matcher

Szenario durchspielen (Robot)

Resultat testen

Eigener Matcher (siehe nächste Folie)

Espresso: Eigene Matcher / Assertions schreiben

Eigene Matcher schreiben ist nicht schwer!

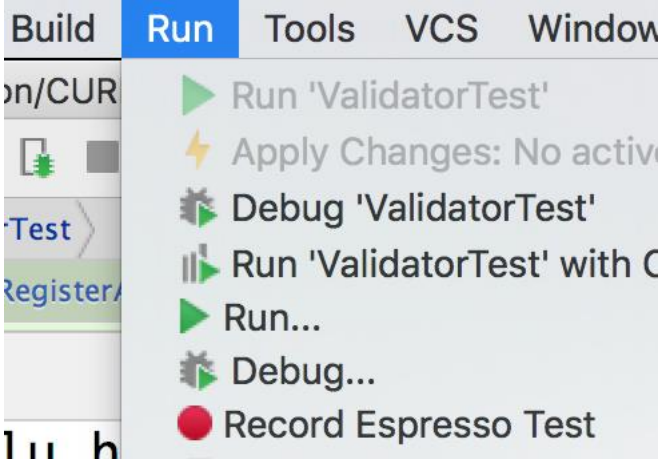
1. Statische Factory-Methode liefert Matcher-Instanz
2. Matcher testet entsprechende Bedingung auf View
3. Beschreibung für Fehlerfall

```
public class MyEspressoMatchers {

    public static Matcher<View> withAnyDrawable() {
        return new DrawableMatcher(0);
    }

    public static Matcher<View> noDrawable() {
        return new DrawableMatcher(-1);
    }

    private static class DrawableMatcher extends TypeSafeMatcher<View> {
        private final int expectedId;
        private String resourceName;

        public DrawableMatcher(int resourceId) {
            super(View.class);
            this.expectedId = resourceId;
        }

        @Override
        protected boolean matchesSafely(View item) {
            if (!(item instanceof ImageView)) {
                return false;
            }
            ImageView imageView = (ImageView) item;
            Drawable actualDrawable = imageView.getDrawable();
            if (expectedId == -1) {
                return actualDrawable == null;
            }
            if (expectedId == 0) {
                return actualDrawable != null;
            }
```

Sprechende Namen!

Bedingungen testen

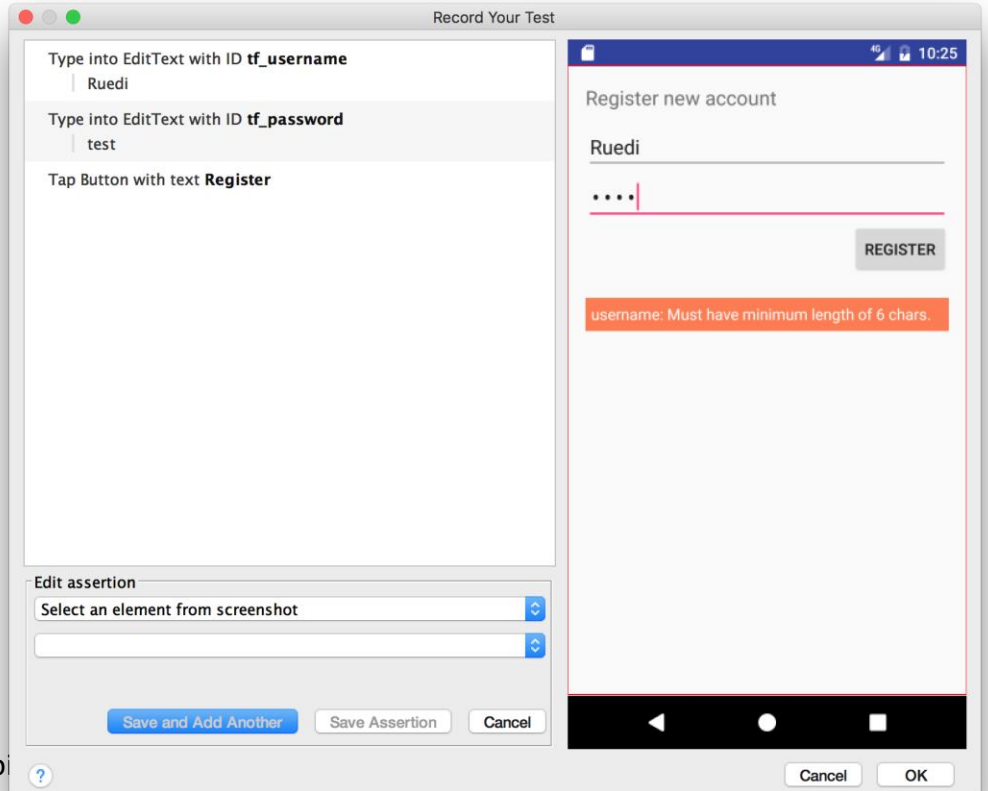
Espresso: Tests aufzeichnen



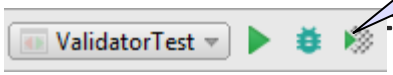
- Neue Funktion von der IDE
  - Seit Android Studio 2.3 ☺
- Erzeugt Test-Klassen

...Spielerei?

Selber ausprobieren!

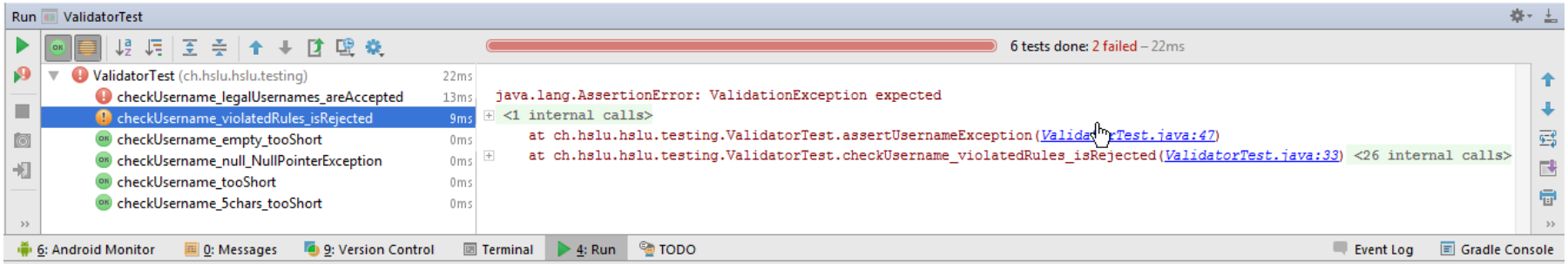


Demo: Unit- und Instrumentation-Tests

- Tests ausführen in Android Studio
  - Klasse öffnen und 

t-F10 / Shift-F9)
  - Auch für ganzes Projekt oder Package möglich (nach Selektion in der Project View)
  - Test-Resultate werden in Run-View angezeigt

Tipp: «Run with Coverage»



Mehr Informationen

- Android-Testing ist an mehreren Orten gut beschrieben in der Android Developer-Doku, u.a. http://developer.android.com/training/testing/index.html

- Weitere Möglichkeiten
  - UI-Automator (ab API 18; App-übergreifende Tests)
  - Monkey & Monkey Runner: Zufallseingaben erzeugenCloud-Test-Lab (siehe nächste Folie)

- Build-Integration: gleich wie in IDE, dank Gradle-Build
  - Emulator oder HW-Gerät *auf Build-Server*

Für Android-Cracks: Cloud Test Lab

- Mittels *Cloud Test Lab* kann eine App simultan auf vielen Devices mit unterschiedlichen Auflösungen, Sprachen und Android-Versionen getestet werden.
  - Die Tests laufen in *Google Datenzentren* auf physikalischen Geräten.
  - Tests können mit Hilfe der integrierten Tools in Android Studio vorbereitet, konfiguriert und deployed werden.
  - Benötigt *Google Cloud Account* und ist *kostenpflichtig*
  - Mehr Infos unter http://developer.android.com/training/testing/start/index.html#run-ctl

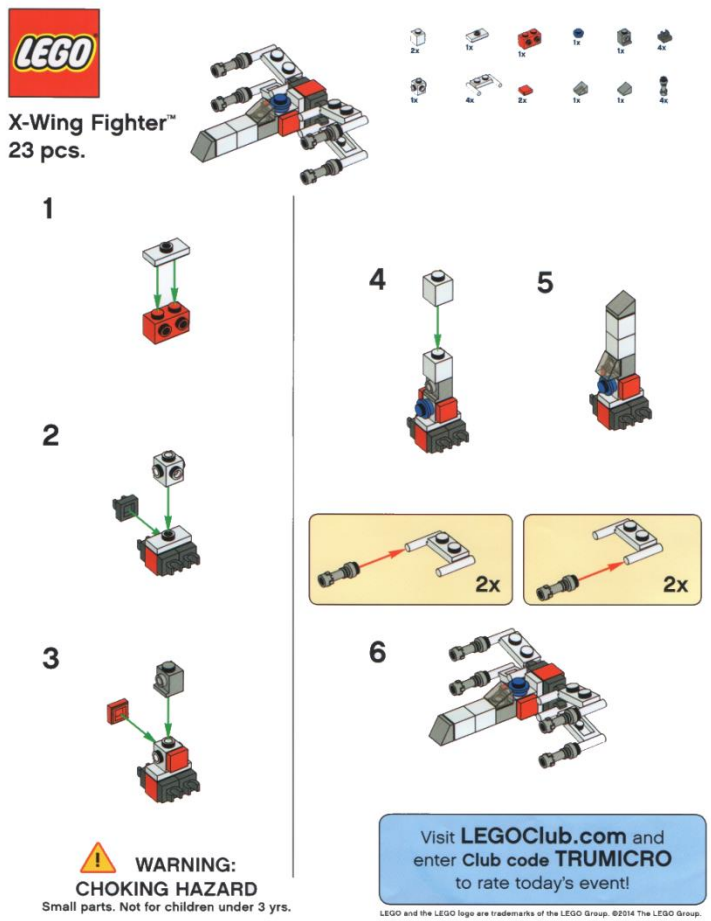
Browserstack

- Mittels *Browserstack* kann eine App simultan auf vielen Devices mit unterschiedlichen Auflösungen, Sprachen und Android-Versionen getestet werden.
  - Die Tests laufen auf physikalischen Geräten.
  - Tests können mit Hilfe der integrierten Tools in Android Studio vorbereitet, konfiguriert und deployed werden.
  - Bieten auch Testgeräte wo man apk hochladen kann.
  - Benötigt *Account* und ist *kostenpflichtig*
  - Mehr Infos unter https://www.browserstack.com/

Gradle Build

https://proandroiddev.com/make-your-build-gradle-great-again-c84cc172a654

Build?



1. Check out code
2. Resolve dependencies
3. Compile
4. Static code analysis
5. Run tests
6. Check coverage
7. Obfuscate
8. Generate documentation
9. Create Artifacts
10. Deploy

Build-Systeme für Java

Ant (2000)

Deklarativ (XML)

Ant Tasks

Kein Projektmodell

Keine Konvention

Kein Lifecycle

Kein Dep.-Mgmt

Kein Skripting

Maven (2004)

Deklarativ (XML)

Projektmodell (POM)

Phases & Goals

Konventionen
- Projektstruktur
- Lifecycle-Phasen

Dependency-Mgmt

Plugins

Kein Skripting

Gradle (2007)

Deklarativ (DSL)

Projektmodell

Tasks + Lifecycle

Konventionen
- Projektstruktur

Dependency-Mgmt

Plugins

Skripting

Ant

```xml
<project name="MyProject" default="dist" basedir=".">
    <description>
        simple example build file
    </description>
  <!-- set global properties for this build -->
  <property name="src" location="src"/>
  <property name="build" location="build"/>
  <property name="dist"  location="dist"/>

  <target name="init">
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Create the build directory structure used by compile -->
    <mkdir dir="${build}"/>
  </target>

  <target name="compile" depends="init"
        description="compile the source " >
    <!-- Compile the java code from ${src} into ${build} -->
    <javac srcdir="${src}" destdir="${build}"/>
  </target>

  <target name="dist" depends="compile"
        description="generate the distribution" >
    <!-- Create the distribution directory -->
    <mkdir dir="${dist}/lib"/>

    <!-- Put everything in ${build} into the MyProject-${DSTAMP}.jar file -->
    <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar" basedir="${build}"/>
  </target>

  <target name="clean"
        description="clean up" >
    <!-- Delete the ${build} and ${dist} directory trees -->
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
  </target>
</project>
```

```
> ant clean dist
```

Maven

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.8.2</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

```
> mvn clean package
```

Gradle

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:1.2.2'
    }
}

repositories {
    jcenter()
}

apply plugin: 'com.android.application'

android {
    compileSdkVersion 21
    buildToolsVersion "22.0.1"

    defaultConfig {
        applicationId "ch.example.hsludemo"
        minSdkVersion 16
        targetSdkVersion 21
        versionCode 1
        versionName "1.0"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.android.support:appcompat-v7:22.1.1'
}
```

```
> gradlew build
```

Gradle Build-Scripts

Grundsätzlich Groovy mit einer DSL (domain specific language)

http://www.groovy-lang.org/
und natürlich
https://gradle.org/

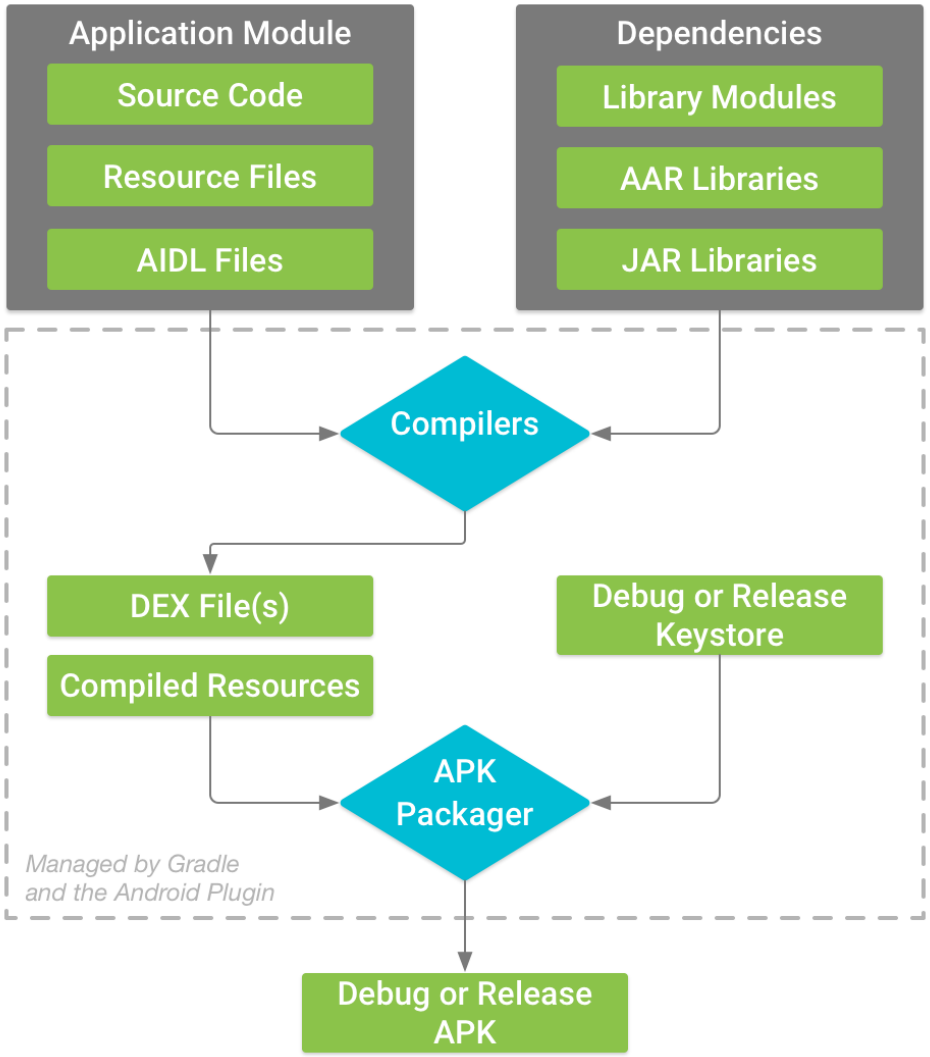
- Gradle Buildskripts enthalten
  - Task-Definitionen
  - Konfigurationen von Modellobjekten und Tasks
  - Ausführbaren Code

Objektnotation ähnlich JSON

- Ein Gradle-Build läuft in 2 Phasen ab
  1. Konfiguration des Builds
  2. Ausführung des Builds (d.h. von spezifischen Tasks)

DAG =
Direct Acyclic Graph
für Ausführungsreihenfolge und
mögl. Parallelisierung

Build Prozess



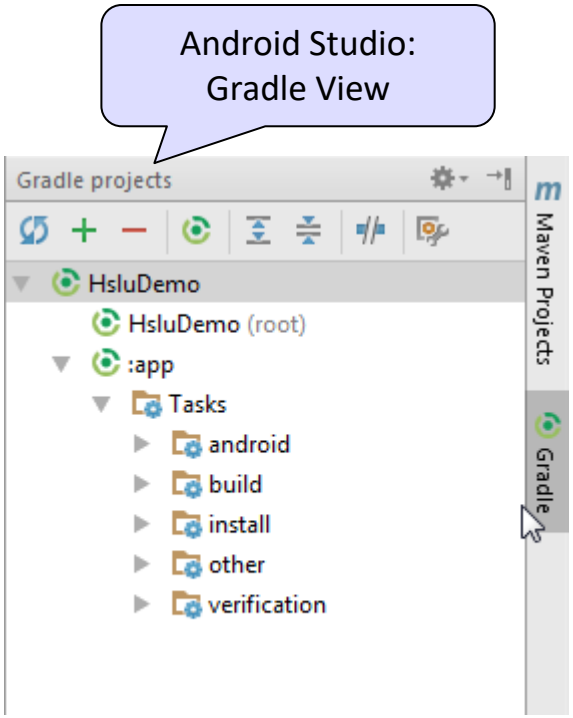
Gradle Plugins

- Ein Gradle-Skript kann Plugins anwenden
  - Plugin als Abhängigkeit hinzufügen (im *buildscript*-Block)
  - Plugin mit ‘apply’ aktivieren
  - Wird automatisch installiert, falls noch nicht vorhanden
- Plugins…
  - definieren neue Modellobjekte
  - fügen einem Projekt weitere Tasks hinzu
  - konfigurieren Buildparameter gemäss Konventionen

Durch «anhängen» an Standard-Tasks werden die neuen Tasks zusammen mit diesen ausgeführt

Android Plugin

- Definiert u.a. die folgenden Tasks
  - assemble
  - check → lint
  - test
  - build → assemble, check
  - install
  - clean
- Definiert das Modellobjekt
  - android



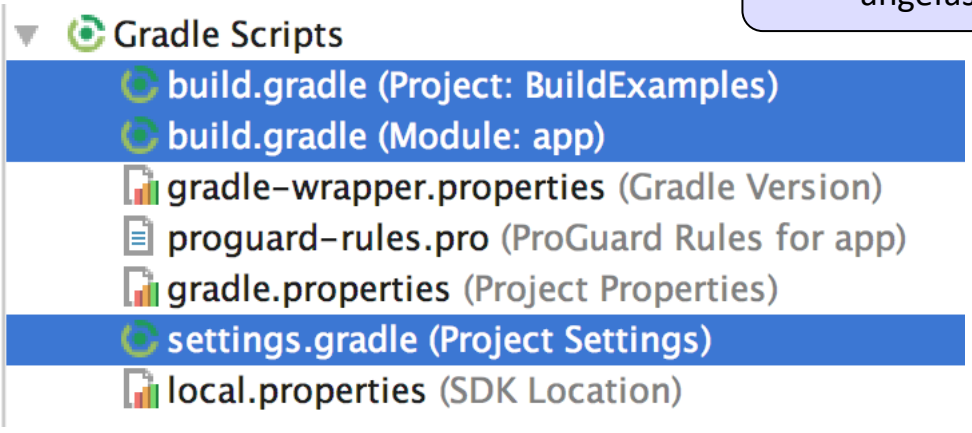

```
> gradlew tasks
```

Module- und Project-Buildfile

- Android-Applikationen können grundsätzlich aus mehreren Modulen bestehen
- Im einfachsten Fall gibt es ein einzelnes «app» Modul mit einem eigenen Build-File
- Auf Projekt-Ebene gibt es noch ein «root» Build-File, welches gemeinsame Build-Konfigurationen definiert für ein Multi-Modul-Projekt

Hier finden 99% der Anpassungen für den App-Build statt

Braucht normalerweise nicht angefasst zu werden



Beispiel: build.gradle (auf Stufe Modul, meist «App»)

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:1.2.2'
    }
}
```

Konfiguration Code-Abhängigkeiten, die für Ausführung *des Buildskripts* notwendig sind (Android-Plugin)

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 30

    defaultConfig {
        applicationId "ch.example.hsludemo"
        minSdkVersion 21
        targetSdkVersion 30
        versionCode 1
        versionName "1.0"
    }
}
```

Anwendung des Plugins und minimale Konfiguration des „android“ Modellobjekts

defaultConfig: falls nicht definiert, werden Werte aus Manifest.xml übernommen

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'androidx.appcompat:appcompat:X.X.X'
}
```

Definition der Abhängigkeiten für den »implementation"-Task

Andere Tasks können andere Abhängigkeiten haben (test z.B.)

Build Konfiguration

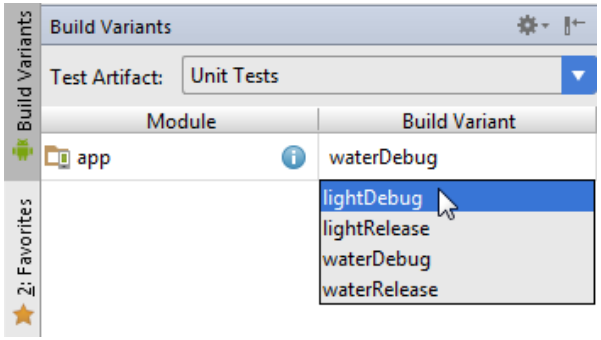
- Default Android Build Configuration
  - applicationId, minSdkVersion, targetSdkVersion, versionCode, versionName
- Dependencies für Tasks
  - implementation
  - testImplementation
- *applicationId* vs. *package* (im Manifest)
  - ID für Store und App
  - Base-Package für Java-Klassen (und R-Klasse)

Muss nicht statisch sein, kann auch durch Skripting berechnet werden!

Sind unabhängig voneinander!

Vor Gradle@Android gab‘s keine applicationId...

Build Variants



- Der Android Build kann von derselben Applikation mehrere Varianten erstellen
  - Build Types: Debug, Release
  - Product Flavors: Free, Paid, Company1, Company2
- Ergibt Task Matrix für Build + Priorität

	Debug	**Release**
Free	buildFreeDebug	buildFreeRelease
Paid	buildPaidDebug	buildPaidRelease

1. Build Type
2. Flavor
3. Default Config

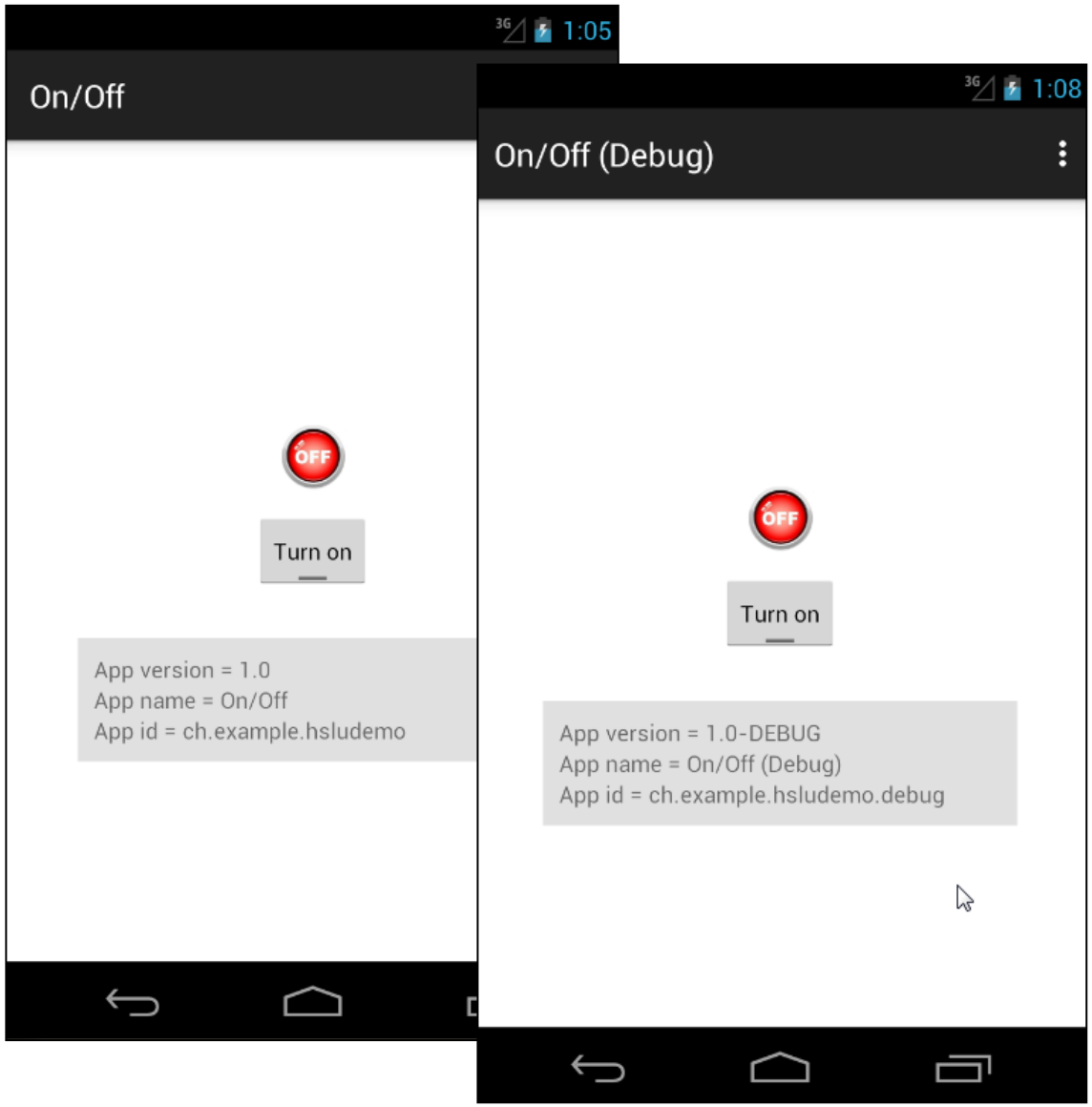
Build Types

- Default Build Typen: `debug`, `release`
  - Unterschiedliche Signing-Konfigurationen (Keys)
  - ProGuard-Konfig (Obfuscator/Shrinker) für Release APK
  - Setzen des «debuggable»-Flags
  - Unterschiedliche `applicationId` und `versionName`

Ermöglicht parallele Installation von Debug- und Release-Version

- Sub-Konfiguration im Build-File: **android.buildTypes**
  - Pro Build-Typ gibt ein neues *`src/<buildType>`* Directory im Modul
  - Definition von alternativen Ressourcen und Klassen für einen spezifischen Build-Type möglich

Demo: Build-Types (debug/release)

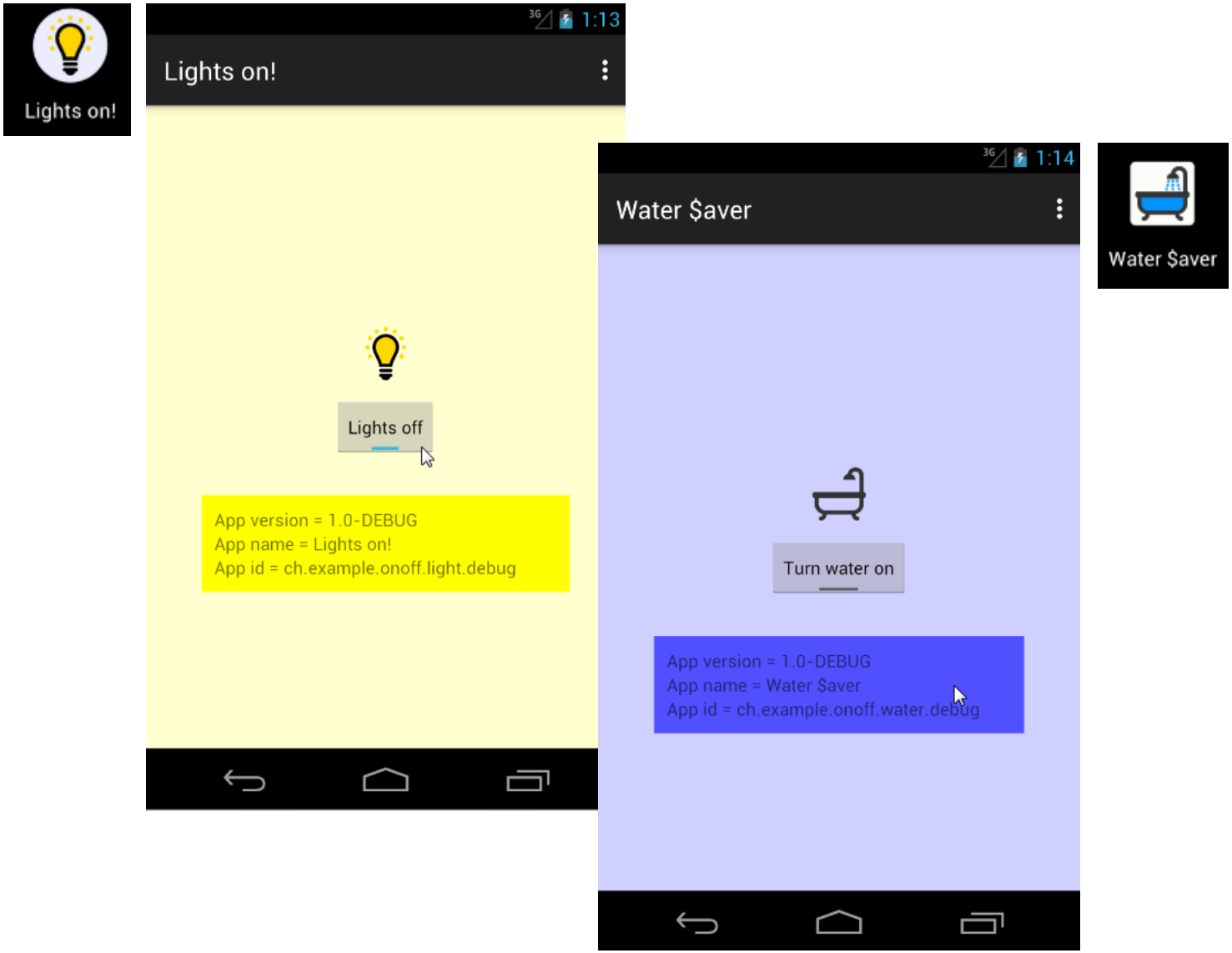


Product Flavors

- Mehrere Varianten derselben App
  - Branding: <Firma 1>, <Firma 2>
  - Funktionalitätsunterschied: <free>, <paid>

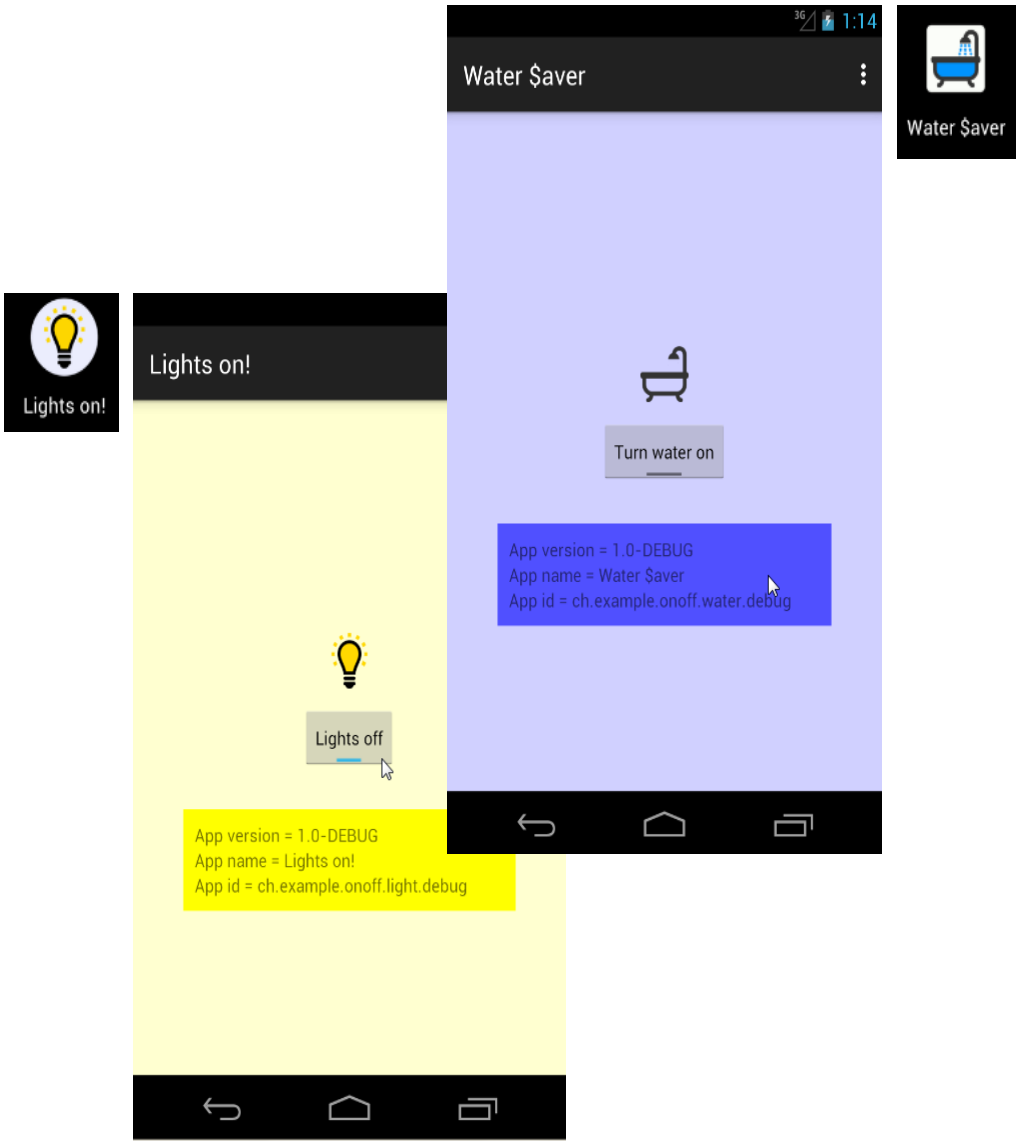
- Subconfig im Build-File: **android.productFlavors**
  - Pro Build-Typ gibt ein neues `src/<productFlavor>` Directory im Modul
  - Definition von alternativen Ressourcen und Klassen für einen spezifischen Product-Flavor möglich

Demo: Product-Flavors



Bsp.: ProductFlavors Gradle-Auszug

```
buildTypes {
    release {
        signingConfig debug.signingConfig
        minifyEnabled false
        proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
    }
    debug {
        debuggable true
        applicationIdSuffix ".debug"
        versionNameSuffix "-DEBUG"
    }
}
// Specifies one flavor dimension.
flavorDimensions "dim"
productFlavors {
    light {
        applicationIdSuffix '.light'
        versionNameSuffix "-light"
    }
    water {
        applicationIdSuffix '.water'
        versionNameSuffix "-water"
    }
}
```



Übung: Team-Projekt

Übung zu Android-7...

- Codelab für Testing + 4 Tests und 1 Espresso Test
  - https://developer.android.com/codelabs/advanced-android-kotlin-training-testing-basics#0
- Gradle Types und Flavors