

# Kapitel 1

## Einführung Mobile GUIs

### 1.1 Besonderheiten Mobiler GUIs

- anderer Platzbedarf als Desktop-GUIs
- knappere Ressourcen (Prozessor, RAM, Flashspeicher)
- andere haptische Eingabetechniken als Desktop-GUIs
- Geräte haben andere Eigenschaften und Funktionalitäten als der klassische Laptop:
  - klein, passen in jede Jackentasche
  - Telefoniedienste (GSM, UMTS),
  - Lagesensor (Gyroskop),
  - GPS (Standortbestimmung, Navigationsanwendungen),
  - Kamera, Mikrofon,
  - Ad-hoc-Netzwerke wie z.B. Bluetooth
- Displays sind klein
- Applikationen sind klein – Informationen müssen nach wenigen Klicks / Swypes auf dem Schirm sein.
- Unzuverlässige Datenverbindungen

- Mitunter hohe Latenzen bei der Datenübertragung

## 1.2 Geschichte mobiler Systeme

Die ersten mobilen Systeme – hier sind nicht Laptops, sondern wirkliche „Pocket“-Systeme gemeint – entstanden bereits seit Anfang der 90er Jahre. Allerdings war hier die Welt der Telefonie (Mobiltelefone) noch getrennt von der Welt der Pocket PCs.

- PDAs mit Palm OS seit 1996
- Windows CE (u. a. für Handheld PCs) seit 1996
- Erstes Blackberry 1999 (Smartphone)
- Erste Tablet PCs ab 2001 (Windows Surf Pad)
- Erstes iPhone 2007
- Erste Android-Smartphones seit 2008
- Apple iPad seit 2010
- Erste Android Tablets seit 2011
- Windows 8/10 Smartphones und Tablets seit Ende 2012/Anfang 2013 (Phones mittlerweile nicht mehr)
- SmartWatches / Intelligent Wearables seit 2014

## 1.3 Arten von Apps

Es gibt 3 Grundtypen mobiler Applikationen – kurz Apps:

- native Apps: Hiermit sind Applikationen gemeint, die auf dem jeweiligen nativen, mobilen Betriebssystem, z.B. Android oder iOS aufsetzen. Sie sind nur auf der Architektur lauffähig, für die sie programmiert wurden. Dafür können sie aber auch sämtliche Funktionalitäten des jeweiligen mobilen Betriebssystems ausschöpfen.
- WebApps: Diese sind auf Basis von HTML, Javascript und CSS aufgebaut. Sie sind plattformunabhängig. Damit sind sie auf den Browsern der meisten gängigen mobilen Betriebssysteme lauffähig.
- Hybride Apps: Hier bestehen die GUI-Elemente meist aus WebApp-Elementen. Die App enthält jedoch auch native Elemente (z. B. Sensoransteuerungen) und ggf. auch einen nativen Wrapper. Es gibt Frameworks, die die Entwicklung hybrider Apps erleichtern (z. B. PhoneGap).



# Kapitel 2

## Einführung Android

### 2.1 Versionen von Android

Bei der Angabe von Android-Versionen werden **Versionsnummern** und **API-Level** angegeben. Der API Level ist eine ganzzahlige ID für die API-Version, die von einer Version der Android-Plattform angeboten wird. Ab Version 1.5 (API-Level 3) haben die Versionen außerdem den Namen einer Süßigkeit.

Versionen und API-Level von Android			
Plattform- Version	Name	API- Level	Highlights
1.0	Base	1	
1.1	Base	2	
1.5	Cupcake	3	
1.6	Donut	4	
2.0	Eclair	5	Für Smartphones
2.0.1	Eclair	6	

Fortsetzung auf nächster Seite

<b>Plattform- Version</b>	<b>Name</b>	<b>API- Level</b>	<b>Anmerkungen</b>
2.1.x	Eclair	7	
2.2.x	Froyo	8	
2.3	Gingerbread	9	Unterstützung diverser Sensoren wie Gyroskop, Beschleunigungssensor, etc.
2.3.1			
2.3.2			
2.3.3	Gingerbread	10	
2.3.4			
3.0.x	Honeycomb	11	Für Tablets
3.1.x		12	
3.2		13	
4.0	Ice Cream Sandwich	14	Ab Version 4.0 für Smartphones und Tablets (2.x und 3.x zusammengeführt); Einführung von Android Beam (NFC-Übertragung zwischen Mobiltelefonen)
4.0.1			
4.0.2			
4.0.3	Ice Cream Sandwich	15	
4.0.4			
4.1	Jelly Bean	16	Erweiterte i18n-Unterstützung (Leserichtung Schriften), User-installierbare Keyboard Layouts, Android Beam via Bluetooth verbessert

Fortsetzung auf nächster Seite

## KAPITEL 2. EINFÜHRUNG ANDROID

---

<b>Plattform- Version</b>	<b>Name</b>	<b>API- Level</b>	<b>Anmerkungen</b>
4.1.1			
4.2	Jelly Bean	17	External Display Support, Native RTL (Right To Left) Support
4.2.2			
4.3	Jelly Bean	18	Unter anderem: BlueTooth Low Energy Support, Sup- port OpenGL ES 3.0, View Overlays
4.4	Kitkat	19	Unter anderem: Prin- ting Framework, Neue Permission-Policy (keine READ_EXTERNAL_STORAGE bzw. WRITE_EXTERNAL_STORAGE- Privilegien mehr notwendig, wenn die App auf die App- spezifischen Bereiche der SD-Karte zugreifen will), Zusätzliche Permissions eingeführt, neue Sensor- Typen bzw. Sensor-Zugriffe eingeführt

Fortsetzung auf nächster Seite

---

<b>Plattform- Version</b>	<b>Name</b>	<b>API- Level</b>	<b>Anmerkungen</b>
4.4W		20	Angepasste Version für SmartWatches und Wearables
5.0	Lollipop	21	Unter anderem: Ersatz der Dalvik-VM durch die neue Android Runtime, 64-Bit-Support, TLS/SSL Default Configuration geändert (TLSv1.2, TLSv1.1 enabled, AES-GCM (AEAD) enabled, MD5 u. 3DES disabled,...)
5.1	Lollipop	22	Unter anderem: Multiple SIM Card Support, alte HTTP-Klassen ersetzt
6.0	Marshmallow	23	Unter anderem: Now on Tap, Intelligenter Akkuverwaltung (Doze), Änderungen im Berechtigungs-Management
7.0	Nougat	24	Unter anderem: Multiwindow-Apps, Weiterentwicklung von Doze
7.1	Nougat	25	Unter anderem: App-Shortcuts
8.0	Nougat	26	Unter anderem: Picture in Picture, Fonts als XML-Ressourcen, Neural Networks API

---

Fortsetzung auf nächster Seite

---

## KAPITEL 2. EINFÜHRUNG ANDROID

---

<b>Plattform- Version</b>	<b>Name</b>	<b>API- Level</b>	<b>Anmerkungen</b>
8.1	Nougat	27	Diverse API-Verbesserungen,
9.0	Pie	28	Unter anderem: Display Cu- tout

Tabelle 2.1: Versionen von Android

## 2.2 Android Software Stack

Android Apps lassen sich in einen 5-schichtigen Software-Stack einordnen:

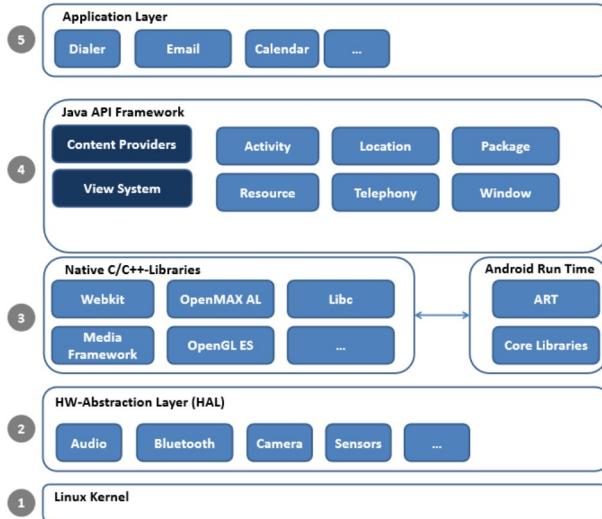


Abbildung 2.1: Android Software Stack(frei nach [12])

Von unten nach oben gelesen, sind die 4 Schichten folgendermaßen zu interpretieren (vgl. [7], S. 25 und [12]):

- **Schicht 1: Linux Kernel**
- **Schicht 2: Hardware Abstraction Layer (HAL)**

stellt Standard-Interfaces für die Einbindung von z.B. Kamera oder Bluetooth bereit.
- **Schicht 3: Native C/C++-Libraries:** Hiermit sind in C bzw. C++ geschriebene Core-Bibliotheken gemeint. Neben der *libc* u.a.:

- **Open GL** für 2D und 3D-Graphik.
  - **OpenMAX AL**: C-Schnittstelle für Multimedia-Anwendungen
  - **SQLite**: ein kleines SQL-basiertes DBMS für die Unterstützung von Datenbank-Funktionalitäten.
  - **WebKit**: HTML-Rendering und Web-Zugriffe
- **Schicht 3: Android Run Time**: Hier beginnt die Unterstützung der App-Funktionalitäten:
    - **Core Libraries**: Die meisten Android Apps sind in Java geschrieben. Da die Android Runtime jedoch keine gewöhnliche Java VM ist, müssen Java-Core-Funktionalitäten hier in einer eigenen Bibliothek angeboten werden. Weiterhin werden hier Android-spezifische Funktionalitäten angeboten.
    - **Android Runtime (ART)**: Ersatz für die frühere Dalvik VM. Sie führt letztendlich den Code der App aus. Unterschied zur früher: Für die Dalvik VM wurde Just-In-Time-Compilierung (JIT) der Dalvik Executables (.dex-Bytecode) durchgeführt. Bei der ART wird dagegen bei der Installation der App plattformabhängiger Binärkode erzeugt (Ahead Of Time- Compilierung / AOT). Grund für diesen Wechsel war eine verbesserte Performance bei der Ausführung der Apps.
  - **Schicht 4: Java API Framework**: Hier werden die Java-Klassen angeboten, aus denen eine native Android-App entwickelt werden kann, beispielsweise:
    - **Content Provider**: Klassen, mit denen Android-Apps ihre Daten teilen können.
    - **Views**: Klassen für die GUI-Elemente einer App.
    - **Activities**: Klassen für die Darstellung **einer** Display-Ansicht einer App.
  - **Schicht 5: Anwendungs-Schicht**: Auf dieser Schicht werden alle Arten von Android Apps eingeordnet.

## 2.3 Auflösung von Android-Geräten

Android-Geräte gibt es unter anderem in folgenden Auflösungskategorien:

Wichtigste Auflösungskategorien von Android-Geräten	
Kategorie	Erläuterung
ldpi	Niedrige Auflösung mit ca. 120 dpi
mdpi	Mittlere Auflösung mit ca. 160 dpi
tvdpi	Mittlere bis hohe Auflösung mit ca. 213 dpi
hdpi	Hohe Auflösung mit ca. 240 dpi
xhdpi	Sehr hohe Auflösung mit ca. 320 dpi
xxhdpi	Noch höhere Auflösung mit ca. 480 dpi
xxxhdpi	Noch höhere Auflösung mit ca. 640 dpi

Tabelle 2.2: Wichtigste Auflösungen von Android-Geräten. Quellen: [30], S. 427 und [21]

Diese Auflösungen spielen z. B. bei der Erzeugung von Icons eine Rolle. In der Regel werden Android-Apps so entwickelt, dass sie auf möglichst vielen Android-Geräten unterschiedlichster Auflösungen laufen. Das Android-SDK erlaubt es dem Entwickler, hier sofort alle benötigten Icons für **alle gewünschten Auflösungen** zu erzeugen.

## 2.4 Android Entwicklungswerkzeuge

Die wichtigsten Android-Entwicklungswerkzeuge sind:

- **Android SDK mit Android Studio:** Wird als IntelliJ-Plugin auf der Entwicklerseite <http://developer.android.com/sdk/index.html> angeboten. Android-Studio ist die offizielle Android-Entwicklungsumgebung. Der veraltete Eclipse-Plugin ADT wird

nicht mehr weiterentwickelt, ist aber noch beziehbar. Er muss jedoch mittlerweile eigenhändig in die neuesten Eclipse-Plattformen eingebunden werden.

- **Android SDK und AVD Manager:** Mit ihm können wir die neuesten API-Levels herunterladen und in unser Android SDK integrieren. Auch **Android Virtual Devices** (AVDs) können mit ihm definiert werden.
- **Android Emulator** ist eine Implementierung der Android VM. Mit ihm können VMs der verschiedensten Android Geräte emuliert werden, so dass Sie Ihre Programme auf ihnen testen können.
- **Android API:** Die API-Beschreibungen der verschiedenen API-Levels sind unter <https://developer.android.com/reference/packages> zu finden.

## 2.5 Aufbau einer nativen Android-App am Beispiel „Hello Android“

### 2.5.1 Bestandteile einer Android-App

Eine Android-App besteht aus

- **Activities:** Eine Activity ist immer der GUI-Bestandteil, der auf *einem* Display dargestellt wird. Er besteht im Wesentlichen aus:
  - **XML-Ressourcen** zur Beschreibung von Layout, Beschriftungen, Farben, Bildinhalten, etc.
  - Einer in Java geschriebenen **Activity-Klasse**. Sie erbt direkt oder indirekt von der Android-Klasse *Activity*.
  - Sonstigen Java-Klassen.
- **Manifest:** Eine Manifest-Datei enthält globale Informationen über die gesamte App, beispielsweise Zugriffsberechtigungen, etc.

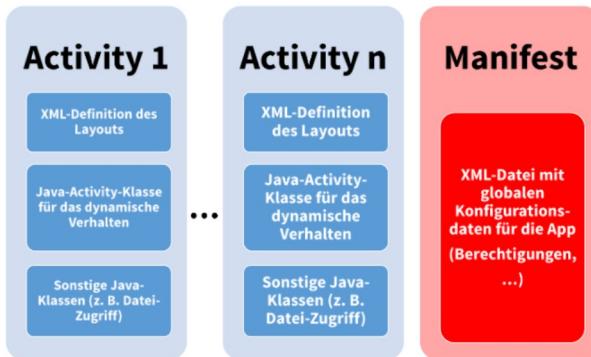


Abbildung 2.2: Bestandteile einer Android-App

## 2.5.2 Hauptbaustein einer Android-App: Die Activity

Eine Android-App für Smartphones oder Tablets besteht immer aus einer oder mehreren Teil-Oberflächen, die meist jeweils das gesamte Display ausfüllen. Diese Teil-Oberflächen werden auch als **Activities** bezeichnet. Sie bestehen unter anderem aus:

- XML-Definitionen für das Layout und
- Java-Klassen, die das Verhalten der Activity bestimmen.

Das Grund-Layout einer einfachen Activity sieht folgendermaßen aus:

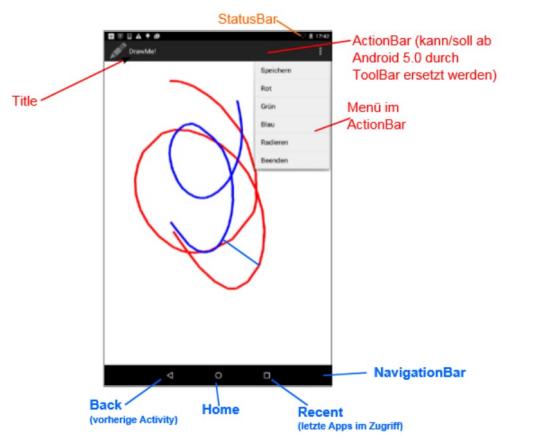


Abbildung 2.3: Grund-Layout einer Android-Activity

Die Activity, welche automatisch beim Start einer Android-App erscheint, wird auch als **Start-Activity** bezeichnet. Mehr dazu wird noch in Kapitel 4 erläutert.

### 2.5.3 Konfigurieren des ersten Android-Projektes im Android-Studio

Wir starten das Android-Studio durch Doppelklick:

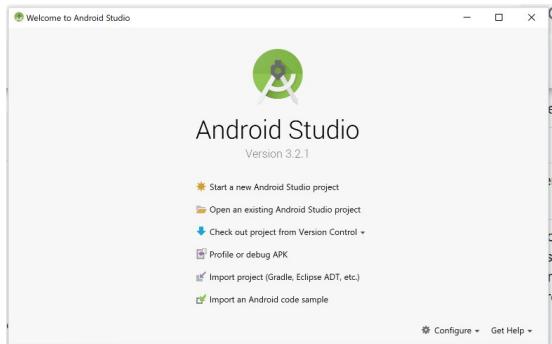


Abbildung 2.4: Welcome-Page von Android-Studio

- 17 Wir erzeugen zunächst ein neues Android-Projekt: Wir können hierbei den absoluten Pfad angeben, unter

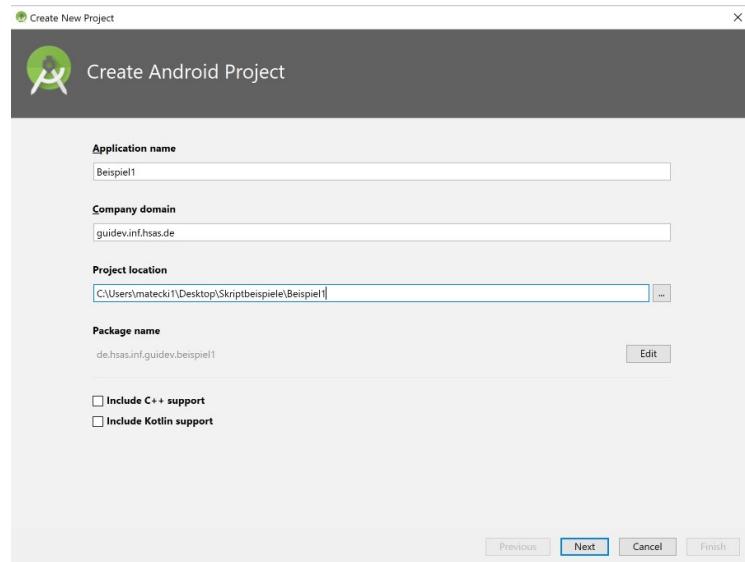


Abbildung 2.5: Neues Android-Projekt anlegen: Projektdaten

dem der Projektordner abgelegt werden soll. Danach **Next**.

In diesem Wizard konfigurieren wir, welche Art von App wir wollen:

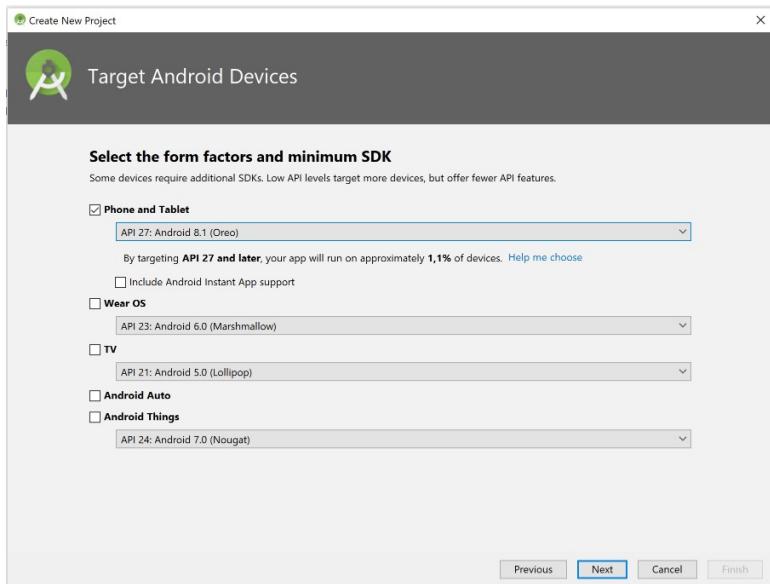


Abbildung 2.6: Art der App konfigurieren

Wir möchten eine für Smartphone und Tablet geeignete App erstellen.  
Danach **Next**.

Nun bestimmen wir, ob und welche Start-Activity wir mit dem Projekt gleich mit erzeugen möchten:

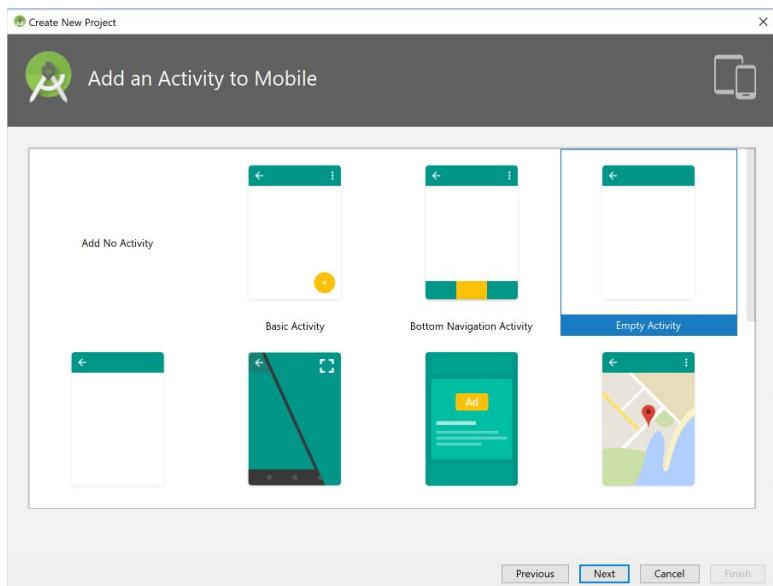


Abbildung 2.7: Art der Start-Activity definieren

Wir benötigen eine „Empty Activity“. Sie enthält bereits rudimentäre Layout-Definitionen, sowie eine zugehörige Java-Klasse. Diese werden wir später beide noch modifizieren. Weiter mit **Next**.

Nun bestimmen wir noch die Dateinamen für die Bestandteile unserer Activity:

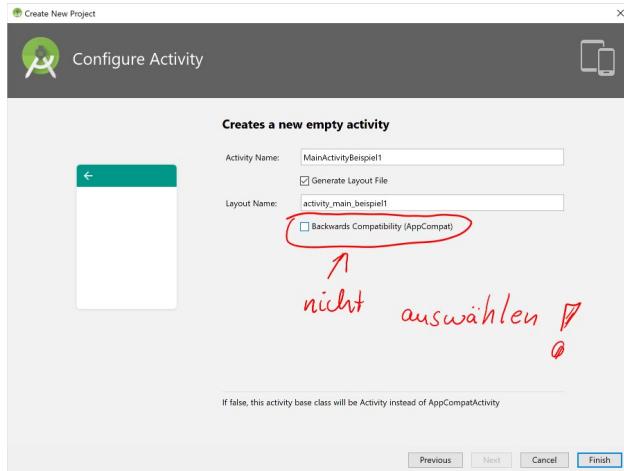


Abbildung 2.8: Dateinamen für Bausteine der Start-Activity angeben

Hierbei bedeuten:

- **Activity Name:** Name der Java-Klasse, die das Verhalten dieser Activity implementiert.
- **Layout Name:** Name der XML-Datei (ohne Suffix .xml), die die Layout-Definitionen für diese Activity enthält.
- **Backwards Compatibility:** Hier würde die Activity-Klasse eine andere Superklasse bekommen, die wir erst in einer späteren Phase der Vorlesung kennenlernen.

Danach bestätigen wir mit **Finish**.

## 2.5.4 Bearbeitung und Start des ersten Android-Projektes

Um eine Android-App auf einem Android-Smartphone oder Tablet zum Laufen zu bringen, sind folgende Schritte notwendig:

- Schritt 1: SDK-Version für die App festlegen (geschieht bereits bei der Projekterstellung)
- Schritt 2: Launcher-Icon für die App erstellen
- Schritt 3: Layout-Definition(en) überarbeiten.
- Schritt 4: Java-Klasse(n) der Activity/Activities überarbeiten
- Schritt 4a: Eventuell weitere Java-Klassen für Dateizugriffe, Netzwerkzugriffe u.ä. implementieren
- Schritt 5: Android Gerät über USB-Kabel mit dem Rechner verbinden (USB-Treiber von Google müssen installiert sein, Gerät muss USB-Debugging erlauben (Entwickleroptionen auf dem Gerät vorher freischalten!).)
- Schritt 6: App auf dem Gerät starten

Das erstellte Projekt aus dem vorherigen Kapitel sieht nun folgendermaßen aus:

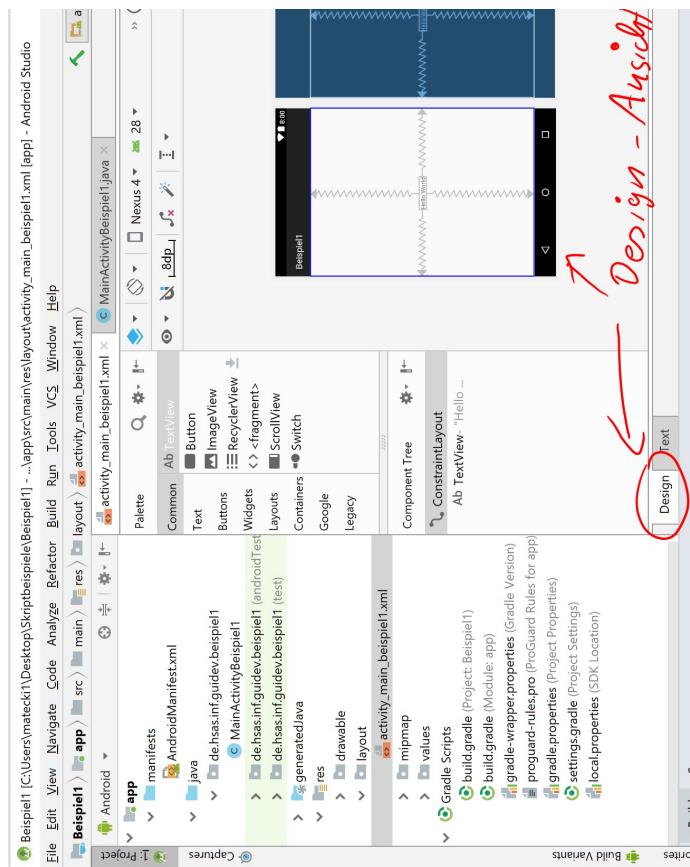


Abbildung 2.9: Design-Ansicht des XML-Layouts

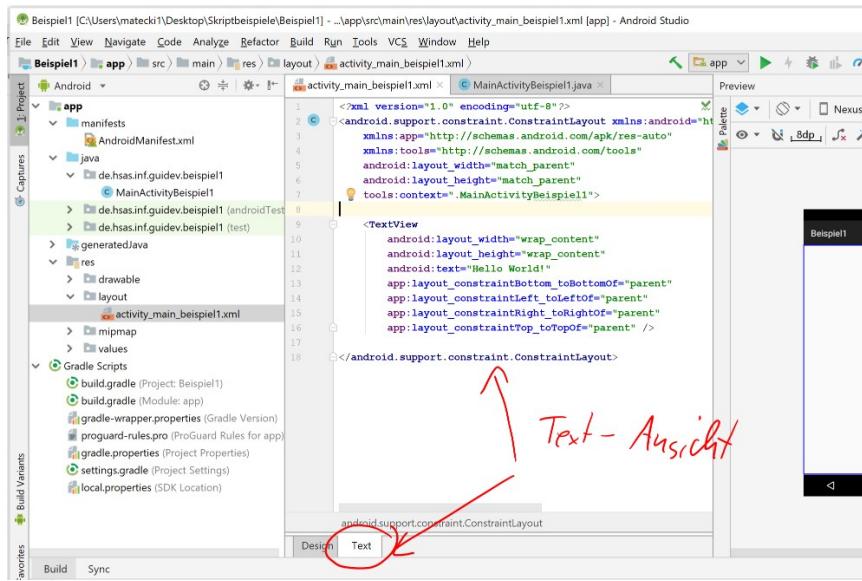


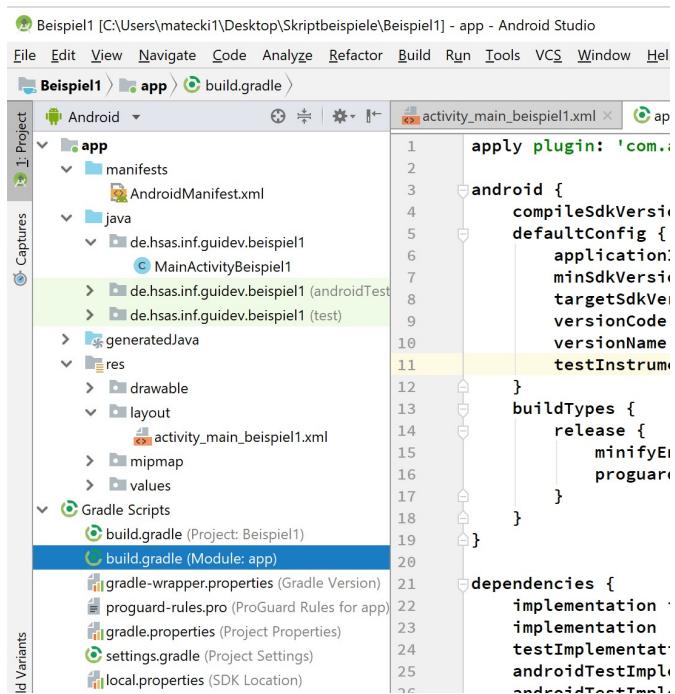
Abbildung 2.10: Text-Ansicht des XML-Layouts

### 2.5.4.1 Schritt 1: Android-Version festlegen

Hier gibt es 2 mögliche Angaben:

- ***minSdkVersion***: Sie gibt an, welcher API-Level mindestens erforderlich ist für die App. Wird hier beispielsweise 11 angegeben, so bedeutet dies, dass das Gerät mindestens API-Level 11 haben muss. Hat es einen höheren API-Level, so läuft die App ebenfalls, aber sie nutzt die neuen Inhalte des höheren Levels nicht.
- ***targetSdkVersion***: Sie gibt an, für welchen API-Level die App gedacht ist.

Die Versions-Konfiguration für unsere App wird in Gradle-Build-Dateien erledigt:



The screenshot shows the Android Studio interface with the project 'Beispiel1' open. The left sidebar displays the project structure, including the app module with its Java files (MainActivityBeispiel1, MainActivityBeispiel1Test), resources (res), and build scripts (build.gradle). The right pane shows the content of the build.gradle file. The code is color-coded, and specific sections like 'apply plugin', 'android', and 'dependencies' are highlighted.

```

apply plugin: 'com.android.application'

android {
    compileSdkVersion 28
    defaultConfig {
        applicationId "de.hdas.inf.guidev.beispiel1"
        minSdkVersion 16
        targetSdkVersion 28
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
    }
}

dependencies {
    implementation 'com.google.android.material:material:1.2.1'
    implementation 'androidx.core:core-ktx:1.3.1'
    implementation 'androidx.appcompat:appcompat:1.2.0'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    implementation 'com.google.firebase:firebase-core:17.4.3'
    androidTestImplementation 'androidx.test:runner:1.3.0'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.3.0'
}

```

Abbildung 2.11: Gradle-Konfiguration (Datei build.gradle)

Diese Build-Dateien werden in der Syntax von **Groovy** geschrieben. Für unsere erste App sieht diese Konfigurations-Datei folgendermaßen aus:

```
1 apply plugin: 'com.android.application',
2
3 android {
4     compileSdkVersion 28
5     defaultConfig {
6         applicationId "de.lsas.inf.guidev.beispiel1"
7         minSdkVersion 27
8         targetSdkVersion 28
9         versionCode 1
10        versionName "1.0"
11        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
12    }
13    buildTypes {
14        release {
15            minifyEnabled false
16            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
17        }
18    }
19}
20
```

```
21 dependencies {  
22     implementation fileTree(dir: 'libs', include: ['*.jar'])  
23     implementation 'com.android.support.constraint:constraint-  
24         layout:1.1.3'  
}
```

Listing 2.1: Gradle-Konfiguration einer Android-App

Früher (unter Eclipse) wurde der API-Level in der Manifest-Datei angegeben:

The screenshot shows the Android Studio interface with the project 'Beispiel1' open. The left sidebar displays the project structure under 'app'. The 'manifests' folder contains the 'AndroidManifest.xml' file, which is highlighted with a red circle and has a red arrow pointing from it to the code editor. The code editor shows the XML content of the manifest file. A red handwritten note 'Manifest-Datei' is written vertically next to the code editor.

```
<?xml version="1.0" encoding="utf-8" standalone="no" ?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="de.hssas.inf.guidev.beispiel1">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="Beispiel 1"
        android:roundIcon="true"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivityBeispiel1">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Abbildung 2.12: Manifest-Datei im Projektbaum

### 2.5.4.2 Schritt 2: Launcher-Icons erzeugen

Das Launcher-Icon einer App ist das Icon, welches auf dem Desktop Ihres Gerätes sichtbar ist, um die App zu starten. Standardmäßig wird der kleine „Androide“ von Android als Launcher-Icon verwendet:

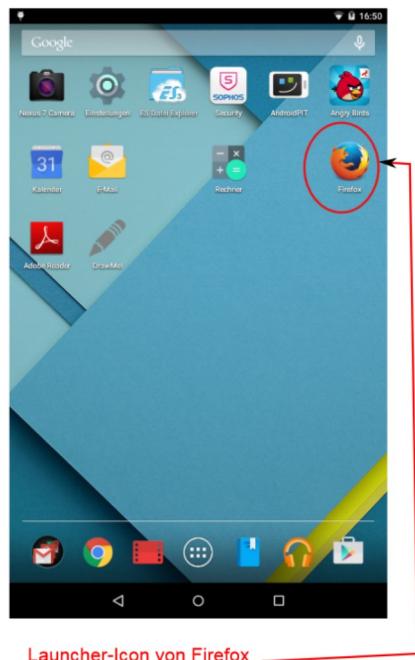


Abbildung 2.13: Launcher Icon unter Android

Wenn wir ein anderes Icon möchten, haben wir mehrere Möglichkeiten:

- Selbst zeichnen und als Icon in das App-Projekt integrieren oder  
...
- ein von Android bereits vorgefertigtes Icon verwenden.

In beiden Fällen werden durch AndroidStudio automatisch Icons für alle gängigen Geräteauflösungen erzeugt. Wir erläutern die Arbeitsschritte beispielhaft für ein vorgefertigtes Icon.

Wir wählen im **File**-Menü **New→Image Asset**:

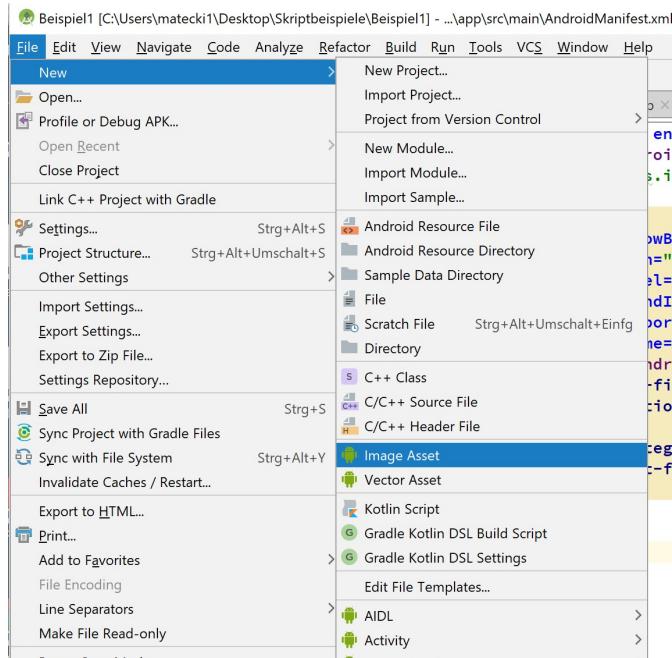


Abbildung 2.14: Launcher Icon erzeugen – Menü

## KAPITEL 2. EINFÜHRUNG ANDROID

Es erscheint der Wizard für die Icon-Erstellung: Wir könnten hier auch

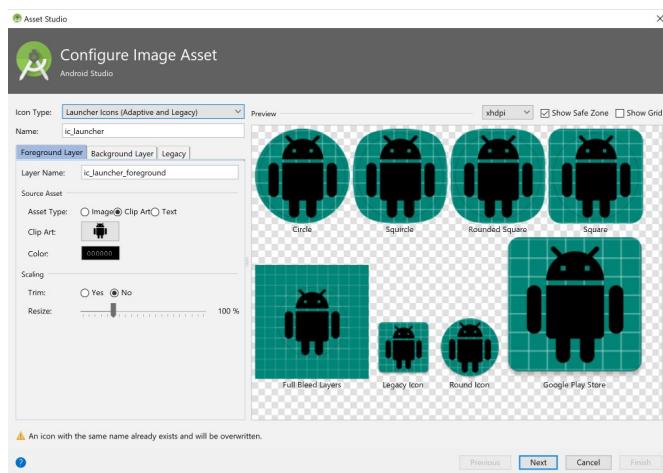


Abbildung 2.15: Launcher Icon erzeugen – Einstiegs-Wizard

aus den von Android zur Verfügung gestellten Cliparts ein anderes als das vorgeschlagene Icon wählen. Auch ein eigenes Bild könnte hier ausgewählt werden. Wir wählen die Art der Icons aus – hier Launcher Icons und nehmen den Clipart-Vorschlag von Android an. Weiter mit **Next**.

Teile der Icons sind Vektorgrafiken – sie werden im Ordner *drawable* gespeichert. Die Pixelgrafik-Anteile werden – je nach Auflösung – in den Ordnern *mipmap-...* zu als Bildpyramide gespeichert:

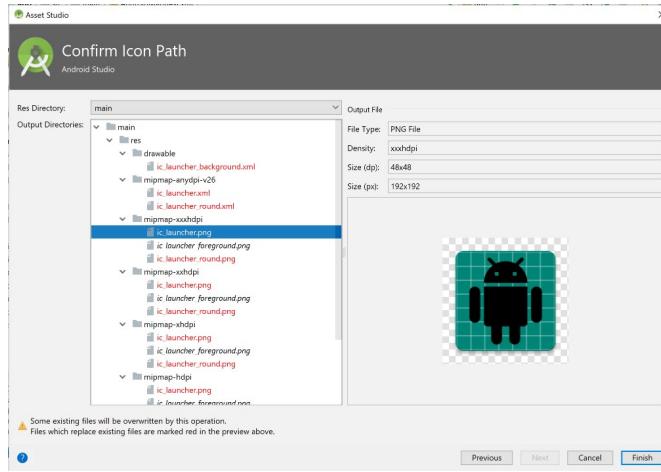


Abbildung 2.16: Launcher Icon erzeugen – Abspeicherungsverzeichnisse

Wir beenden mit **Finish**.

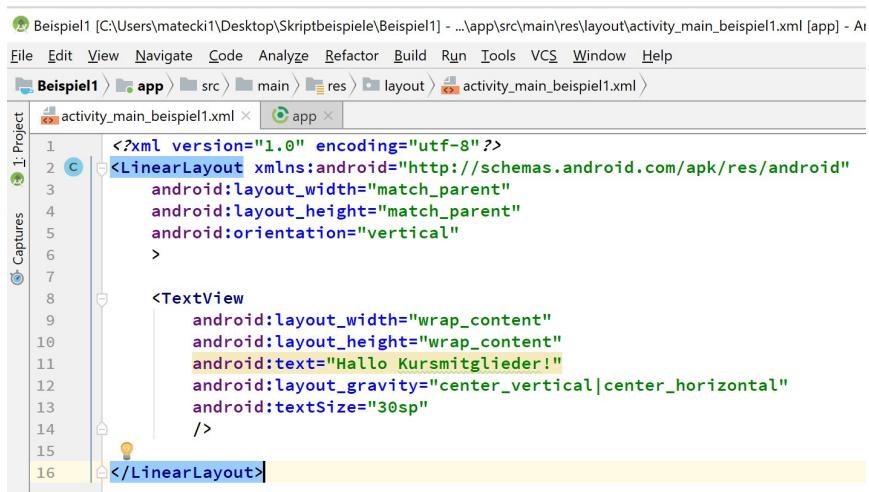
Anmerkung: In früheren Versionen wurden Icons genauso wie andere grafische Ressourcen behandelt und landeten automatisch im Ordner *res→drawable*. Zum Begriff „**mipmap**“: Hierbei handelt es sich um ein Verfahren aus der Bildverarbeitung, um über **Bild-Pyramiden** bestmöglich die verschiedenen Auflösungen abbilden zu können. Derartige (Pixel-)Ressourcen werden mittlerweile getrennt von Vektorgrafik-Informationen aus dem **drawable**-Ordner gehalten.

### 2.5.4.3 Schritt 3: Layout-Definition überarbeiten

Wir bearbeiten nun zuerst die XML-Layout-Definition unserer App. Diese finden wir im Projektordner `res→layout`. Es gibt 2 Modi zur Layout-Bearbeitung:

- **XML-Modus:** In ihm sehen wir wirklich den Code des Layouts und können diesen auch direkt beeinflussen.
- **WYSIWYG-Modus:** (What You See Is What You Get): Dieser Modus wird häufig eher zur Kontrolle des vorher in XML kodierten Layouts verwendet.

Wir öffnen zunächst das XML-Layout in der Datei `activity_main_beispiel1.xml` im Textmodus und bearbeiten sie folgendermaßen:



The screenshot shows the Android Studio interface with the code editor open. The file path is `Beispiel1 [C:\Users\matecki1\Desktop\Skriptbeispiele\Beispiel1] - ...\\app\\src\\main\\res\\layout\\activity_main_beispiel1.xml [app] - Ar`. The code editor displays the following XML code:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    >
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hallo Kursmitglieder!"
        android:layout_gravity="center_vertical|center_horizontal"
        android:textSize="30sp"
    />
</LinearLayout>
```

Abbildung 2.17: Layout-Datei bearbeiten – Text-Modus

Die Einträge haben folgende Bedeutung: Es wird ein sehr einfacher Layout-Manager definiert, der vertikal orientiert ist. Er selbst füllt sein „Parent-Widget“ – also das Display – vollkommen aus. Für den Layout-Manager werden drei Attribute mit Werten belegt:

- **android:orientation**: Hier wird angegeben, ob die „Kinder“ des Layouts nebeneinander oder untereinander angeordnet werden.
- **android:layout\_width**: Hier wird angegeben, wie weit sich die Breite des Layout-Managers in sein Parent erstreckt. Hier: Volle Breite des Parents.
- **android:layout\_height**: Hier wird angegeben, wie weit sich die Höhe des Layout-Managers in sein Parent erstreckt. Hier: Volle Höhe des Parents.

Im Layout-Manager wird ein *TextView* angelegt. Dieses entspricht dem *Label* im SWT. Für die *TextView* werden derzeit folgende Attribute belegt:

- **android:layout\_width**: Hier wird angegeben, wie weit sich die Breite der *TextView* in ihr Parent erstreckt. Hier: so viel wie der Textinhalt benötigt.
- **android:layout\_height**: Hier wird angegeben, wie weit sich die Höhe der *TextView* in ihr Parent erstreckt. Hier: so viel wie der Textinhalt benötigt.
- **android:gravity**: Hier wird angegeben, wie der Text der *TextView* ausgerichtet wird. Hier: Horizontal zentriert kombiniert mit Vertikal zentriert (vgl. Style-Flagbits im SWT).
- **android:textSize**: Hier wird die Textgröße der *TextView* angegeben.
- **android:text**: Hier wird der Textinhalt der *TextView* angegeben. An dieser Stelle haben wir gemogelt – wir haben den Textinhalt „hard-coded“ angegeben, was eigentlich schlechter Stil ist. Im nächsten Beispiel werden wir lernen, wie ein solcher Textinhalt in Form von weiteren Ressourcen ausgelagert wird.

**WYSIWYG-Modus:** Wir kontrollieren das Layout im graphischen Modus (Design-Modus):

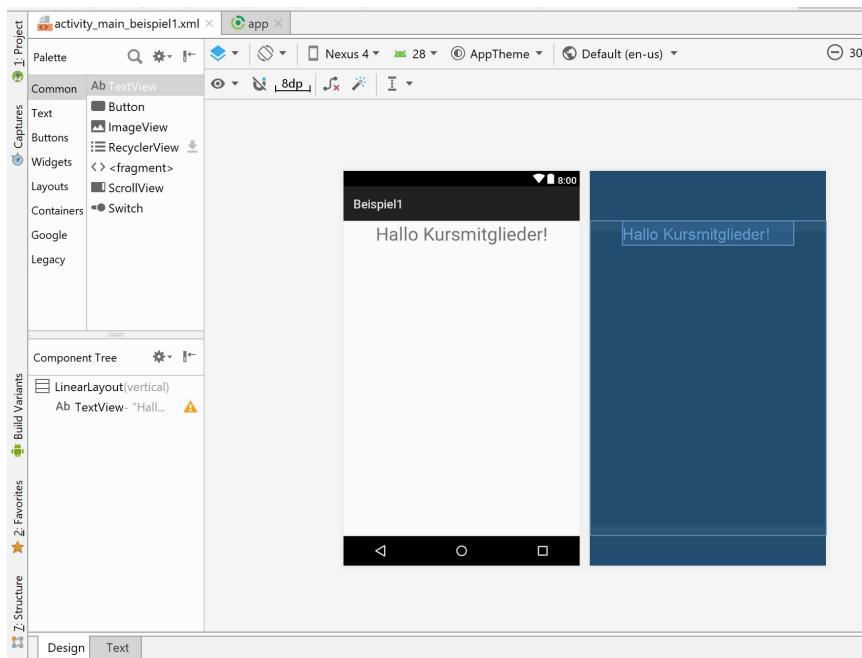
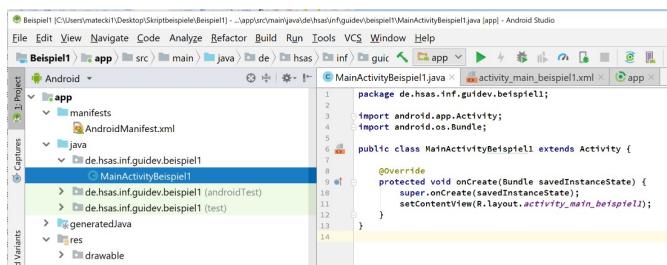


Abbildung 2.18: Layout-Datei im Design-Modus

#### 2.5.4.4 Schritt 4: Java-Klasse der Activity bearbeiten

In diesem Schritt bearbeiten wir die vom AndroidStudio generierte Java-Klasse für unsere Main-Activity. Sie ist im src-Verzeichnis des Projektes im Package `guidev.inf.hsas.de.beispiel1` verankert:



```

Beispiel1 [C:\Users\matecki1\Desktop\Skriptbeispiele\Beispiel1] - .../app/src/main/java/de/hsas/inf/guidev/beispiel1/MainActivityBeispiel1.java [app] - Android Studio
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
Beispiel1 app src main java de hsas inf guic app
Captures Variants
I Project
Android
app
manifests
AndroidManifest.xml
java
de.hdas.inf.guidev.beispiel1
MainActivityBeispiel1
> de.hdas.inf.guidev.beispiel1 (androidTest)
> de.hdas.inf.guidev.beispiel1 (test)
generatedJava
res
drawable

MainActivityBeispiel1.java
activity_main_beispiel1.xml
app

1 package de.hdas.inf.guidev.beispiel1;
2
3 import android.app.Activity;
4 import android.os.Bundle;
5
6 public class MainActivityBeispiel1 extends Activity {
7
8     @Override
9     protected void onCreate(Bundle savedInstanceState) {
10         super.onCreate(savedInstanceState);
11         setContentView(R.layout.activity_main_beispiel1);
12     }
13 }
14

```

Abbildung 2.19: Activity-Klasse im Android-Projekt

Die Klasse *MainActivityBeispiel1* ist von der Klasse *Activity* abgeleitet. Sie enthält derzeit nur eine Methode: Die Methode *onCreate()* wird beim Start der Activity – in unserem Falle beim Start der App – ausgeführt. Sie führt den Aufbau des Fensters durch. Wir zeigen den Code der Klasse:

```
1 package de.hdas.inf.guidev.beispiel1;
2
3 import android.app.Activity;
4 import android.os.Bundle;
5
6 public class MainActivityBeispiel1
7     extends Activity {
8
9     @Override
10    protected void onCreate(
11        Bundle savedInstanceState) {
12        super.onCreate(savedInstanceState);
13        setContentView(
14            R.layout.activity_main_beispiel1);
15    } // end method
16 } // end class
```

Die Methode *onCreate()* wird für diese Activity neu implementiert – sie überschreibt die gleichnamige Methode der Superklasse *Activity*. Sie ruft allerdings als allererstes Statement genau diese überschriebene Elternklassen-Methode auf. Danach gibt sie an, mit welchem Layout diese Activity verknüpft werden soll. Hierbei kommt eine ebenfalls generierte Klasse *R* zur Anwendung. Sie enthält im Wesentlichen **Inner Classes** mit Konfigurationskonstanten für die Activities unserer App. Eine dieser Konfigurationskonstanten – der Bezeichner für das Layout der Main Activity – wird hier mit der Activity verknüpft.

#### 2.5.4.5 Schritt 5: App auf einem Android-Gerät starten:

Hierzu müssen einige Dinge im Vorfeld geschehen: Um eine neu entwickelte App direkt auf dem Gerät zu testen, müssen einige Dinge vorbereitet werden:

- Gerät als „Entwicklergerät“ konfigurieren (Entwickleroptionen freischalten),
- USB-Einstellungen am Gerät konfigurieren,
- USB-Treiber für das ADK nachinstallieren, falls nicht schon geschehen
- USB-Treiber für Ihr spezielles Gerät auf dem PC installieren.

Das Gerät wird über USB an den Rechner angeschlossen. Es fragt Sie ggf., ob Sie USB-Debugging zulassen wollen. Hierzu müssen wir uns eine **Run Configuration** definieren. Mit dieser sagen wir, auf welchem emulierten oder realen Gerät wir die App starten. Wir starten mit **Run → Edit Configurations** .... Im dadurch geöffneten Dialogfenster benötigen wir zuerst den Tab **General**:

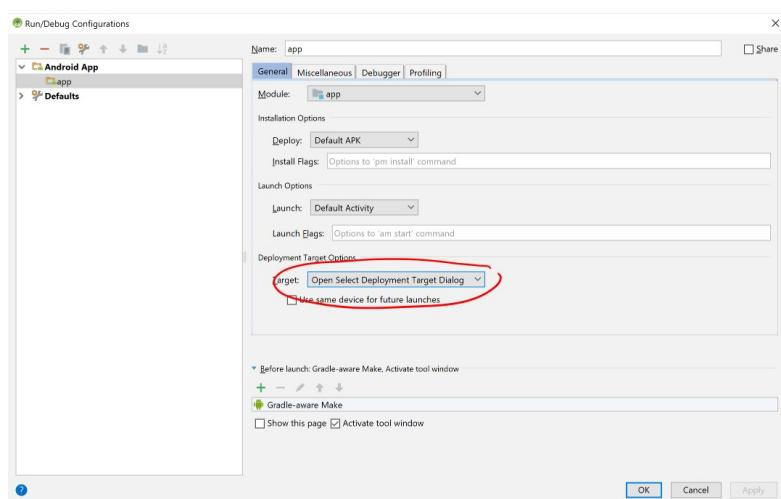


Abbildung 2.20: Run Configuration erstellen

Wir wählen hier auf der linken Seite des Dialogs **Android Application** → **app**. Wir selektieren hier

- Deploy Default APK
- Launch Default Activity (Wir haben nur eine einzige Activity)
- Show Chooser Dialog (wir werden gefragt, ob wir auf echtem oder virtuellem Gerät testen wollen)

Danach **Ok**.

Nun starten wir die Konfiguration mit **Run → Run app ...**.

Nach einiger Zeit erscheint der „Choose Dialog“, mit dem Sie das Gerät auswählen können, auf dem Sie testen wollen. Bei mir ist das eines der Hochschulgeräte (derzeit ein Nexus 5X):

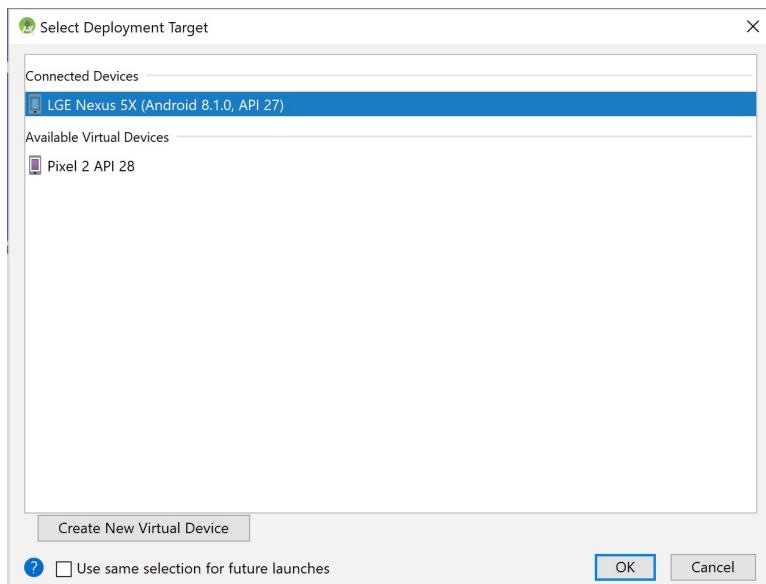


Abbildung 2.21: App auf dem Gerät starten

Wir sehen, dass unser reales Gerät angezeigt wird. Da wir auf dem Gerät selbst noch nicht bestätigt haben, dass wir „USB-Debugging“ zulassen wollen, wird dies auch im Launcher-Dialog der Entwicklungsumgebung so gemeldet. Wir

- bestätigen zunächst auf **unserem Gerät** die Frage nach USB-Debugging und
- können danach den Start der App im Launcher-Dialog der Entwicklungsumgebung bestätigen. Der Start kann selbst bei einer so

kleinen Anwendung eine Weile dauern, da die vollständige APK-Datei übertragen und auf dem Gerät installiert wird.

## 2.5.5 Struktur einer Android-App

### 2.5.5.1 Verzeichnisaufteilung einer Android-App

Wir fassen zusammen: Die Struktur einer nativen Android-App sieht wie folgt aus:

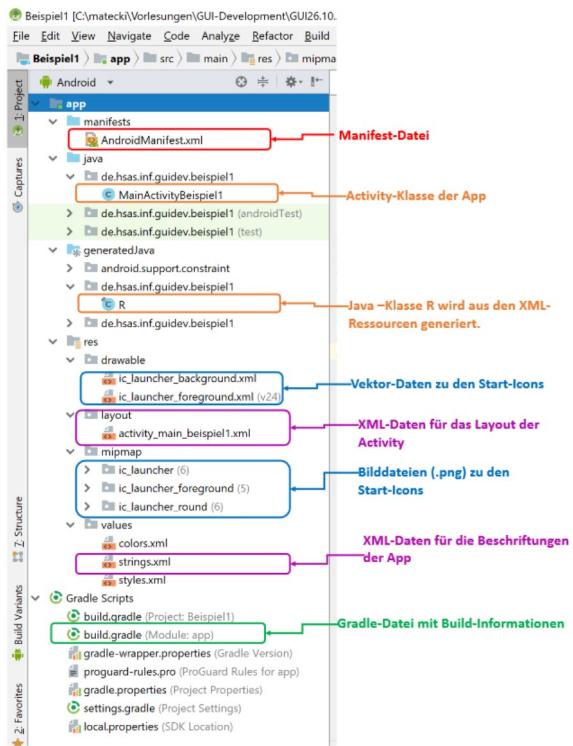


Abbildung 2.22: Vollständige Darstellung Beispiel1Android im Project-Explorer

Die Projektstruktur ist in folgende Teilbereiche untergliedert:

<b>Projektstruktur von Android-Projekten</b>	
<b>Verzeichnis</b>	<b>Erläuterung</b>
manifests	enthält die Manifest-Datei.
java	enthält alle selbst geschriebenen/bearbeiteten Java-Klassen, unter anderem die, welche die Activities implementieren
generatedJava	Enthält alle generierten Java-Klassen, unter anderem die Klasse R.
res	Ressourcen der App. Dies können beispielsweise sein: <ul style="list-style-type: none"> <li>• ausgelagerte Strings,</li> <li>• Icons für die verschiedenen Auflösungen der unterschiedlichen Gerätetypen,</li> <li>• sämtliche Layout-Definitionen incl. Menüdefinitionen – sie werden unter Android als XML-Ressourcen betrachtet.</li> </ul>
Gradle Scripts	enthält die Gradle-Build-Dateien zur Erzeugung der ausführbaren App.

Tabelle 2.3: Projektstruktur von Android-Projekten

### 2.5.5.2 Ressourcen in XML und die daraus generierte Klasse *R* – *Beispiel2Android*

Die Klasse *R* wird bei der Erzeugung eines Android-Projektes immer automatisch, auf Basis der Layout-, String- und Bild-Ressourcen generiert. Wir zeigen die Layout-Datei von *Beispiel2Android*. Der Name der Layout-Dateien ist frei wählbar.

## KAPITEL 2. EINFÜHRUNG ANDROID

Datei activity\_button.xml:

```
1 <LinearLayout xmlns:android="http://schemas.android.com/apk/res
  /android"
2   android:orientation="vertical"
3   android:layout_width="fill_parent"
4   android:layout_height="wrap_content">
5     <!-- weitere Definitionen ... -->
6     <CheckBox android:id="@+id/check1"
7       android:layout_width="wrap_content"
8       android:layout_height="wrap_content"
9       android:textSize="30.0sp"
10      android:text="@string/check_title1"/>
11    <Button android:id="@+id/button1"
12      android:background="#FF0000"
13      android:textSize="30.0sp"
14      android:layout_width="fill_parent" Inner Class id abgebildet.
15      android:layout_height="wrap_content"
16      android:text="@string/button1_title"
17      android:onClick="button1Action"/>
18    <!-- weitere Definitionen ... -->
19 </LinearLayout>
```

Datei R.java:

```
1 public final class R {
2   // weitere Inner Classes ...
3   public static final class id {
4     public static final int button1=0x7f070004;
5     public static final int check1=0x7f070001;
6     // weitere ID-Konstanten ...
7   } // end inner class id
8   // weitere Inner Classes ...
9   public static final class string {
10     public static final int app_name=0x7f040000;
11     public static final int button1_title=0x7f040006;
12     public static final int button2_title=0x7f040007;
13     public static final int check_title1=0x7f040003;
14     // ... weitere Definitionen
15   } // end inner class string
16
17   // ... weitere Inner Classes
18 } // end class R
```

Abbildung 2.23: Layout-Ressourcen und deren Abbildung in Klasse *R*

Das Listing zeigt wieder ein einfaches, lineares Layout – diesmal mit einigen Check-Boxen und einem PushButton. Sämtliche Elemente des Layouts besitzen hier eine ID – gekennzeichnet mit:

```
<TextView android:id="@+id/textview1" ...>
```

Das „+“ nach dem „@“ bedeutet dabei, dass hier eine neue ID vereinbart wird. Diese IDs werden in Form von Konstanten in **Inner Classes** der Klasse *R* gehalten. Unter diesen Java-Konstanten sind die Widgets innerhalb ihrer Activity-Klasse ansprechbar.

Wir haben in der vorherigen Abbildung gesehen, dass jedem Widget in seiner Layout-Definition eine ID gegeben wird. Die Beschriftungen unserer Widgets hatten wir bei der ersten Hello-World-App noch „fest verdrahtet“ im Layout vereinbart. Dies haben wir bei unserem jetzigen Beispiel schon geändert. Die Beschriftungen werden ebenfalls in eine eigene Ressourcen-Datei ausgelagert:

```

Datei activity_button.xml:
1 <LinearLayout xmlns:android="http://schemas.android.com/apk/res
  /android"
2   android:orientation="vertical"
3   android:layout_width="fill_parent"
4   android:layout_height="wrap_content">
5   <!-- weitere Definitionen ... -->
6   <CheckBox android:id="@+id/check1"
7     android:layout_width="wrap_content"
8     android:layout_height="wrap_content"
9     android:textSize="30.0sp"
10    android:text="@string/check_title1"/>
11  <Button android:id="@+id/button1"          Die Beschriftungs-IDs
12    android:background="#FF0000"           werden in der Klasse R
13    android:textSize="30.0sp"             in der Inner Class string
14    android:layout_width="fill_parent"    abgebildet.
15    android:layout_height="wrap_content"
16    android:text="@string/button1_title"
17    android:onClick="button1Action"/>
18  <!-- weitere Definitionen ... -->
19 </LinearLayout>

Datei strings.xml enthält die Belegungen der Widget-Beschriftungen:
1 <?xml version="1.0" encoding="utf-8"?>
2 <resources>
3   <string name="check_title1">checkbox 1</string>
4   <string name="button1_title">Press To Check</string>
5   <!-- weitere String-Definitionen .... -->
6 </resources>

Datei R.java:
1 public final class R {
2   // weitere Inner Classes ....
3   public static final class string {
4     public static final int app_name=0x7f040000;
5     public static final int button1_title=0x7f040006;
6     public static final int button2_title=0x7f040007;
7     public static final int check_title1=0x7f040003;
8     // ... weitere Definitionen
9   } // end inner class string
10
11  // ... weitere Inner Classes
12 } // end class R

```

Abbildung 2.24: String-Ressourcen und deren Abbildung in Klasse R

### 2.5.5.3 Activities

Jede Android-App enthält mindestens eine Activity – jede Activity beinhaltet i. d. R. den Inhalt eines Display-Fensters. Die Namen der Activities sind mit ihren Package-Zuordnungen im **Manifest** der App vermerkt:

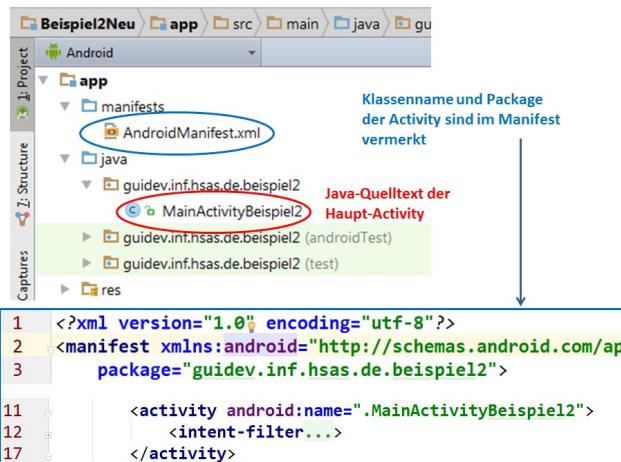


Abbildung 2.25: Vermerk einer Activity im Manifest

Das Manifest ist die zentrale Konfigurationsdatei des App-Projektes (siehe Kap. 2.5.5.4).

Im Gegensatz zu unseren gewohnten Java-Programmen enthält eine Android-App **keine Klasse mit einer main()-Methode**. Statt dessen sorgt die Haupt-Acitivity im src-Verzeichnis dafür, dass die Applikation gestartet wird:

```
1 import android.app.Activity;
2 import android.os.Bundle;
3
4 public class ButtonActivity
5     extends Activity {
6     @Override
7     protected void onCreate(
8         Bundle savedInstanceState) {
9         // Aufruf der onCreate () -Methode
10        // der Elternklasse
11        super.onCreate(savedInstanceState);
12
13        // Aufruf der ererbten Methode
14        // setContentView () .
15        // Sie laedt hier das Layout der
16        // Haupt -Activity
17        setContentView(
18            R.layout.activity_button);
19    } // end method onCreate()
20 } // end class
```

Von ihr aus werden auch ggf. weitere Activities gestartet. Diese Abhängigkeiten zwischen den Activities sind ebenfalls im Manifest vermerkt.

Eine Activity ist immer direkt oder indirekt von der Superklasse *Activity* abgeleitet. Unsere Klasse *MainActivityBeispiel1* **überschreibt** die von *Activity* ererbte **Callback-Methode** *onCreate()*. Bei ihr handelt es sich um eine Reaktionsmethode, welche ausgeführt wird, sobald die Activity erzeugt wird. In unserem Falle tut die Methode zwei Dinge:

- Sie ruft die *onCreate()*-Methode der Superklasse auf.
- Sie baut den Inhalt der Activity gemäß dem Layout der Hauptactivity auf. Die ID dieses Layouts beziehen wir aus der generierten Klasse *R*. Diese wiederum hat ihren ID-Vermerk bei der Generierung aus den von uns definierten Layout-Daten bezogen.

Eine Activity durchläuft folgenden Lebenszyklus:

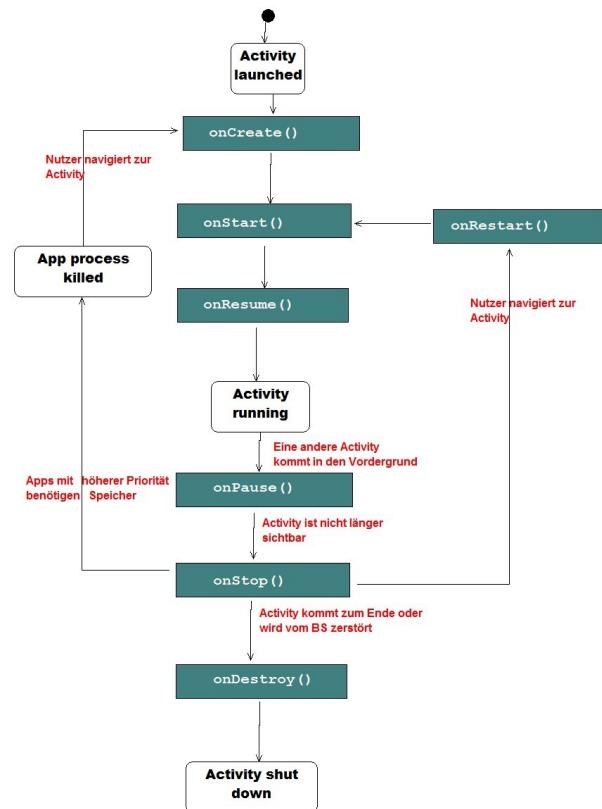


Abbildung 2.26: Lebenszyklus einer Activity (Quelle: [13])

Die grauen Kästen zwischen den Zuständen kennzeichnen **Callback-Methoden** einer Activity. Hierbei handelt es sich um Reaktionsmethoden einer Activity, welche bei Zustandswechseln automatisch aufgerufen werden. Sie beginnen immer mit der Silbe **on**. Sie werden häufig in abgeleiteten Klassen **überschrieben** – so wie im oberen Beispiel die

*onCreate()*-Callback-Methode. Die folgende Tabelle zeigt eine Beschreibung der Callback-Methoden:

Wichtigste Callback-Methoden einer Activity	
Methode	Erläuterung
<code>onCreate()</code>	Wird aufgerufen <ul style="list-style-type: none"><li>• beim ersten Start der Activity,</li><li>• nach einem Kill und Restart der Activity, sowie</li><li>• wenn ein neues Rendering (z. B. Portrait/Landscape) mit neuen Ressourcen notwendig ist.</li></ul>
<code>onDestroy()</code>	Wird aufgerufen, <ul style="list-style-type: none"><li>• wenn die Activity <code>finish()</code> aufruft, sowie</li><li>• wenn die Activity von Android vorzeitig wegen Speicherplatzbedarf geschlossen wird.</li></ul> <p>Die Methode gibt die in <code>onCreate()</code> reservierten Ressourcen wieder frei.</p>
<code>onStart()</code>	Wird aufgerufen, <ul style="list-style-type: none"><li>• wenn die Activity zum ersten Mal innerhalb der App gestartet wird,</li><li>• wenn die Activity wieder im Vordergrund landet, nachdem sie durch eine andere Activity oder App in den Hintergrund geschickt wurde (z. B. Alarm, eingehender Anruf, ...).</li></ul>

---

Fortsetzung auf nächster Seite

---

Methode	Erläuterung
onRestart()	wird aufgerufen, wenn die Activity gestoppt wurde und nun wieder startet.
onStop()	wird aufgerufen, wenn die Activity gestoppt wird.
onPause()	wird aufgerufen, wenn die Activity z. B. durch den Wechsel in eine andere Activity (andere Seite) unterbrochen wird.
onResume()	wird aufgerufen, kurz bevor die Activity wieder in den Vordergrund geschickt wird. Hier werden dann z. B. neue hinzugekommene Verzeichnisinhalte, Tabelleninhalte, etc. neu gerendert.

Tabelle 2.4: Wichtigste Callback-Methoden einer Activity

#### 2.5.5.4 Das Manifest

Das Manifest ist **die** zentrale Konfigurationsdatei einer App. Sie enthält unter anderem:

- unter welchen Android-API-Levels die App lauffähig sein soll,
- welche Berechtigungen die App besitzt – hier wird dann bei der Installation auf dem Smartphone nachgefragt, ob der Nutzer damit einverstanden ist, der App diese Berechtigungen zu erteilen.
- eine Beschreibung jeder Activity der App,
- einen Vermerk, welche Activity innerhalb des Projektes die Hauptactivity ist. Sie beinhaltet das Startfenster unserer App.
- welche Activities welche Sub-Activities starten.

Wir zeigen das vollständige Manifest unseres zweiten Beispiels:

```

Beispiel2/AndroidManifest.xml
1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3 package="de.hsas.ti.bo.beispiel2" >
4
5 <application
6     android:allowBackup="true"
7     android:icon="@drawable/ic_launcher"
8     android:label="@string/app_name"
9     android:theme="@style/AppTheme" >
10    <activity
11        android:name=".ButtonActivity"
12        android:label="@string/app_name" >
13        <intent-filter>
14            <action android:name="android.intent.action.MAIN" />
15            <category android:name="android.intent.category.LAUNCHER" />
16        </intent-filter>
17    </activity>
18 </application> Definition einer Activity (hier die Haupt-Activity). Eine <application>...
19 </application> <application>-Definition kann mehrere solcher Activity-Definitionen
20 </manifest> enthalten.

```

Abbildung 2.27: Manifestdatei von *Beispiel2Android*

## 55 2.5.6 Zugriffsberechtigungen einer Android-App

### 2.5.6.1 Verwaltung bis einschließlich API-Level 22

Bis API-Level 22 (Android Version 5.1) war es nur notwendig, die benötigten Berechtigungen einer Android-App im Manifest einzutragen:

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3     package="gui.inf.hhas.de.permissionexample">
4     <!-- Permissions werden VOR den App-Definitionsdaten eingetragen -->
5     <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
6     <uses-permission android:name="android.permission.CAMERA"/>
7     <!-- Erst jetzt folgen die App-Definitionen -->
8     <application ... >
9         <!-- ... Activity-Definitionen ... -->
10    </application>
11 </manifest>
```

Listing 2.2: Zugriffsprivilegien im Manifest

In unserem Beispiel haben wir Leseberechtigung für den Zugriff auf unsere SD-Karte, sowie Kamerazugriff eingetragen.

Diese Berechtigungen werden bei den älteren Versionen von Android (bis API-Level 22) einmalig bei der Installation der App genehmigt.

### 2.5.6.2 Verwaltung ab API-Level 23

Ab API-Level 23 (Android-Version 6.0.1) unterscheidet Android zunächst zwischen **gefährlichen** und **ungefährlichen** Zugriffsprivilegien. Als **gefährlich** werden alle Zugriffsprivilegien angesehen, die die Privatsphäre des Nutzers verletzen könnten, wie beispielsweise Kamerazugriff, Mikrofon, aber auch Datenzugriffe. Derartige Privilegien sind **Privilegien-Gruppen (Permission-Groups)** zugeordnet und müssen vor der Nutzung ausdrücklich beim ersten Start der App vom Nutzer genehmigt werden.

Gefährliche Zugriffsprivilegien für Android-Applikationen	
Privilegien-Gruppe	Einzel-Privilegien
CALENDAR	READ_CALENDAR, WRITE_CALENDAR
CAMERA	CAMERA
CONTACTS	READ_CONTACTS, WRITE_CONTACTS, GET_ACCOUNTS
LOCATION	ACCESS_FINE_LOCATION, ACCESS_COARSE_LOCATION
MICROPHONE	RECORD_AUDIO
PHONE	READ_PHONE_STATE, CALL_PHONE, READ_CALL_LOG, WRITE_CALL_LOG, ADD_VOICE_MAIL, USE_SIP, PROCESS_OUTGOING_CALLS

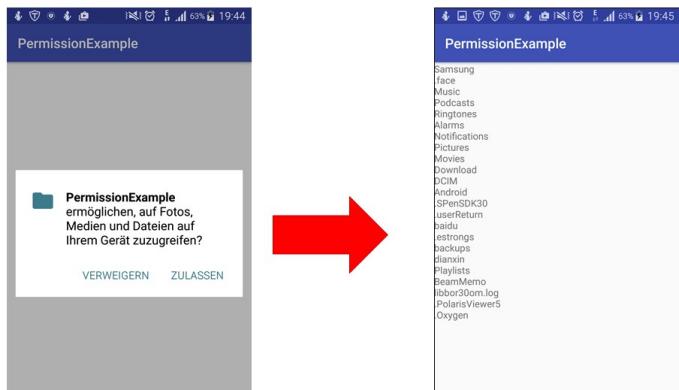
Fortsetzung auf nächster Seite

---

Privilegien-Gruppe	Einzel-Privilegien
SENSORS	BODY_SENSORS
SMS	SEND_SMS, READ_SMS, RECEIVE_SMS, RECEIVE_WAP_PUSH, RECEIVE_MMS
STORAGE	READ_EXTERNAL_STORAGE, WRITE_EXTERNAL_STORAGE

Tabelle 2.5: Gefährliche Zugriffsprivilegien lt. [20]

Unsere nächste Beispiel-App versucht, Zugriff auf die Verzeichnis-Daten unserer SD-Karte zu bekommen:



Privilegien beim Benutzer erfragen

Falls die Privilegien erteilt wurden:  
Anzeige Verzeichnisinhalt

Abbildung 2.28: Demo-App für die Erteilung von Privilegien

(c) Prof. Dr. U. Matecki Das Manifest sieht ähnlich aus wie im vorherigen Abschnitt. Unsere einzige benötigte Zugriffsberechtigung ist die Leseberechtigung auf unsere SD-Karte:

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3     package="gui.inf.hdas.de.permissionexample">
4         <!-- Permissions werden VOR den App-Definitionsdaten eingetragen -->
5         <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
6         <!-- Erst jetzt folgen die App-Definitionen -->
7         <application
8             android:allowBackup="true"
9             android:icon="@mipmap/ic_launcher"
10            android:label="@string/app_name"
11            android:supportsRtl="true"
12            android:theme="@style/AppTheme">
13             <activity android:name=".PermissionActivity">
14                 <intent-filter>
15                     <action android:name="android.intent.action.MAIN" />
16
17                     <category android:name="android.intent.category.LAUNCHER" />
18                 </intent-filter>
19             </activity>
20         </application>
21     </manifest>
```

Listing 2.3: Manifest der Demo-App *PermissionExample*

Das Layout ist sehr einfach aufgebaut. Wir haben lediglich ein Textfeld, in welchem der Verzeichnisinhalt dargestellt werden soll:

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <LinearLayout
3   xmlns:android="http://schemas.android.com/apk/res/
     android"
4   android:layout_width="match_parent"
5   android:layout_height="match_parent"
6   android:orientation="vertical">
7   <TextView
8     android:id="@+id/textfield"
9     android:layout_width="match_parent"
10    android:layout_height="wrap_content"/>
11 </LinearLayout>
```

Im Quelltext der Activity geschieht nun etwas mehr:

```
1 package gui.inf.hsas.de.permissionexample;
2
3 import android.Manifest;
4 import android.content.pm.PackageManager;
5 import android.os.Environment;
6 import android.support.v4.app.ActivityCompat;
7 import android.support.v4.content.ContextCompat;
8 import android.support.v7.app.AppCompatActivity;
9
10 import android.os.Bundle;
11 import android.widget.TextView;
12
13 import java.io.File;
14
15 public class PermissionActivity
16   extends AppCompatActivity {
17
18   private int PERM_REQUEST_CODE = 42;
19   private TextView textfield;
```

Unsere Activity ist nun von *AppCompatActivity* abgeleitet. Diese Klasse bietet etwas mehr Komfort bei einigen Implementierungsaspekten wie Zugriffsprivilegien und Menügestaltung.

```
20  @Override
21  protected void onCreate(Bundle savedInstanceState) {
22      super.onCreate(savedInstanceState);
23      setContentView(R.layout.activity_permission);
24
25      // Referenz des Textfeldes abfragen
26      textfield = (TextView)findViewById(R.id.textfield);
27
28      // Permissions abfragen
29      checkPermission();
30  }
```

In der `onCreate()`-Methode wird nun als letztes Statement die Überprüfung der Zugriffsprivilegien mit einer privaten Hilfsmethode durchgeführt.

```
31 private void checkPermission() {  
32     // Sofern noch keine Zugriffsberechtigung erteilt,  
33     // wird diese beim ersten Gebrauch der App  
34     abgefragt  
35     if ( ContextCompat.checkSelfPermission(this,  
36         Manifest.permission.READ_EXTERNAL_STORAGE)  
37         != PackageManager.PERMISSION_GRANTED) {  
38  
38     // Permissions werden in ein String-Array verankert  
39     // Hier nur eine Permission -- daher nur ein Array-  
40     // Element  
41     ActivityCompat.requestPermissions(this,  
42         new String[]{Manifest.permission.  
43             READ_EXTERNAL_STORAGE},  
44             PERM_REQUEST_CODE);  
45     }  
45 } // end method checkPermissions
```

Diese Methode überprüft, ob schon bei einem früheren Start der App das Lese-Privileg auf die SD-Karte erteilt wurde. Falls nein, so erfragt sie dieses Privileg. Bei Erteilung merkt die App sich einen **REQUEST\_CODE**, der innerhalb der App mit einem ganzzahligen Wert (hier 42) vorbelegt wird.

```
45  @Override
46  public void onRequestPermissionsResult(int
47      requestCode,
48      String permissions[],
49      int[] grantResults) {
50      // nur reagieren, wenn auch tatsaechlich die
51      // uns interessierenden Rechte benoetigt werden
52      if (requestCode == PERM_REQUEST_CODE) {
53
54          // Wenn Rechte erteilt, benutzen wir sie hier
55          if ((grantResults.length > 0 &&
56              grantResults[0] ==
57              PackageManager.PERMISSION_GRANTED)) {
58              testAccess();
59          }
60      } // end method onRequestPermissionsResult
```

Dieser Wert wird von der Callback-Methode `public void onRequestPermissionsResult()` überprüft, sobald die Privilegienvergabe abgeschlossen ist. Falls das Privileg erfolgreich erteilt wurde, so wird unsere private `testAccess()`-Methode aufgerufen, mit der das Wurzelverzeichnis der SD-Karte ausgelesen wird.

Zuletzt folgt die eigentliche Zugriffsmethode auf unsere SD-Karte:

```
61 public void testAccess(){
62     // Pfad zur SD-Karte holen
63     String ext_path =
64         Environment.getExternalStorageDirectory().
65             getAbsolutePath();
66
67     // File-Objekt auf diesen Pfad
68     File access= new File(ext_path);
69
70     // Inhalt des Pfades abfragen
71     String [] content = access.list();
72
73     String sTextField = "";
74
75     if (content != null) {
76         for (int i=0; i<content.length; i++) {
77             sTextField += content[i] + "\n";
78         }
79     } else {
80         sTextField = "nichts gefunden!";
81     }
82     // Abgefragten Inhalt einblenden
83     textfield.setText(sTextField);
84 } // end method testAccess()
85 } // end class
```

Sie gibt einfach den Inhalt des Wurzelverzeichnisses auf unser Textfeld aus.

**Merke**

Bei gefährlichen Privilegien muss ab Android Version 6 jedes Privileg zur Laufzeit überprüft werden. Die Callback-Methode `onRequestPermissionsResult()` überprüft hierbei das Ergebnis der Privilegien-Abfrage.

Es folgt noch ...