



Universität Trier

Zentrum für Informations-, Medien- und Kommunikationstechnologie

Bernhard Baltes-Götz

# Einführung in die Entwicklung von Apps für Android 8



2018 (Rev. 180907)

Herausgeber: Zentrum für Informations-, Medien- und Kommunikationstechnologie (ZIMK)  
an der Universität Trier  
Universitätsring 15  
D-54286 Trier  
WWW: [zimk.uni-trier.de](http://zimk.uni-trier.de)  
E-Mail: [zimk@uni-trier.de](mailto:zimk@uni-trier.de)

Autor: Bernhard Baltes-Götz  
WWW: <https://www.uni-trier.de/~baltes>  
E-Mail: [baltes@uni-trier.de](mailto:baltes@uni-trier.de)

Copyright © 2018; ZIMK

---

## Vorwort

Dieses Manuskript entstand als Begleitlektüre zum Kurs *Einführung in die Entwicklung von Android-Apps*, den das Zentrum für Informations-, Medien- und Kommunikationstechnologie (ZIMK) an der Universität Trier im Sommersemester 2018 angeboten hat, ist aber auch für das Selbststudium geeignet. In hoffentlich seltenen Fällen enthält der Text noch Formulierungen, die nur für Kursteilnehmer perfekt passen.<sup>1</sup>

### Voraussetzungen bei den Lesern(innen)

- **Programmierkenntnisse**

Die Leser sollten über Erfahrungen mit einer objektorientierten Programmiersprache (bevorzugt Java) verfügen. Eine mögliche Informationsquelle zu Java ist das Manuskript zum Java-Kurs im ZIMK (Baltes-Götz & Götz 2018).

- **EDV-Ausstattung**

Dass die Leser über ein eigenes Android-Smartphone oder -Tablet verfügen, ist nützlich, aber nicht unbedingt erforderlich, weil in verwendeten Entwicklungsumgebung Emulatoren für Android-Geräte zur Verfügung stehen. Zur Softwareentwicklung wird im Manuskript ein PC unter Windows verwendet. Weil dabei ausschließlich Java-Software zum Einsatz kommt, ist die Plattformunabhängigkeit jedoch garantiert.

### Verwendete Software

Für die unverzichtbaren Übungen sollte ein Rechner zur Verfügung stehen, auf dem das von der Firma Google angebotene Android Studio samt Android SDK installiert ist. Diese Software ist kostenlos für alle signifikanten Plattformen (z. B. Linux, MacOS, UNIX, Windows) im Internet verfügbar. Nähere Hinweise zum Bezug, zur Installation und zur Verwendung folgen in Kapitel 2.

### Aktuelle Ausgabe des Manuskripts und ergänzende Dateien

Die aktuelle Version dieses Manuskripts ist zusammen mit den behandelten Beispielen und Lösungsvorschläge zu vielen Übungsaufgaben auf dem Webserver der Universität Trier von der Startseite (<http://www.uni-trier.de/>) ausgehend folgendermaßen zu finden:

[IT-Services \(ZIMK\) > Downloads & Broschüren >](#)  
[Programmierung > Einführung in die Entwicklung von Apps für Android 8](#)

Leider blieb zu wenig Zeit für eine sorgfältige Kontrolle des Textes, so dass einige Fehler und Mängel verblieben sein dürften. Entsprechende Hinweise an die Mail-Adresse

[baltes@uni-trier.de](mailto:baltes@uni-trier.de)

werden dankbar entgegen genommen.<sup>2</sup>

Trier, im Juli 2018

Bernhard Baltes-Götz

---

<sup>1</sup> Zur Vermeidung von sprachlichen Umständlichkeiten wird in diesem Manuskript in der Regel die männliche Form verwendet.

<sup>2</sup> An dieser Stelle geht ein herzliches Dankeschön an Frau Birgit Feltrup für das Aufspüren von zahlreichen Tippfehlern.



---

# Inhaltsverzeichnis

<b>VORWORT .....</b>	<b>III</b>
<b>1 JAVA UND ANDROID.....</b>	<b>1</b>
<b>1.1 Quellcode, Bytecode und Maschinencode .....</b>	<b>1</b>
<b>1.2 Standardbibliothek .....</b>	<b>4</b>
<b>1.3 Android-Systemarchitektur .....</b>	<b>5</b>
<b>1.4 Android-Versionen und API-Levels .....</b>	<b>6</b>
<b>1.5 Dokumentation zu Android und seinem Java-API .....</b>	<b>7</b>
<b>1.6 Alternativen zu Java bei der Entwicklung von Android-Apps .....</b>	<b>7</b>
<b>1.7 Übungsaufgaben zu Kapitel 1 .....</b>	<b>8</b>
<b>2 INSTALLATION DER ENTWICKLUNGSUMGEBUNG UNTER WINDOWS .....</b>	<b>10</b>
<b>2.1 Android Studio installieren .....</b>	<b>10</b>
<b>2.2 Android-Studio zum ersten Mal starten und das Android SDK installieren.....</b>	<b>13</b>
<b>2.3 Die erste Android-App.....</b>	<b>17</b>
<b>2.3.1 App erstellen.....</b>	<b>17</b>
<b>2.3.2 App im Emulator ausführen.....</b>	<b>22</b>
<b>2.4 Wichtige Pfade für Einstellungen und Projekte .....</b>	<b>24</b>
<b>2.5 Übungsaufgaben zu Kapitel 2 .....</b>	<b>25</b>
<b>3 APPS AUF VIRTUELLELLER UND REALER HARDWARE TESTEN .....</b>	<b>26</b>
<b>3.1 SDK-Manager.....</b>	<b>26</b>
<b>3.1.1 SDK-Tools aktualisieren bzw. ergänzen.....</b>	<b>26</b>
<b>3.1.2 SDK-Plattformen aktualisieren bzw. ergänzen.....</b>	<b>27</b>
<b>3.2 AVD-Manager .....</b>	<b>29</b>
<b>3.2.1 Virtuelle Android-Geräte definieren und konfigurieren .....</b>	<b>29</b>
<b>3.2.2 Ressourcen-Verbrauch.....</b>	<b>33</b>
<b>3.3 Intel(R) Hardware Accelerated Execution Manager (HAXM) .....</b>	<b>34</b>
<b>3.4 Android Debug Bridge.....</b>	<b>36</b>
<b>3.5 Reales Android-Gerät per USB mit der Entwicklungsumgebung verbinden .....</b>	<b>39</b>
<b>3.5.1 USB-Debugging auf dem Android-Gerät aktivieren .....</b>	<b>39</b>
<b>3.5.2 Android USB-Treiber installieren .....</b>	<b>40</b>
<b>3.5.3 Entwicklungsrechner auf dem Android-Gerät zulassen.....</b>	<b>41</b>
<b>3.6 Übungsaufgaben zu Kapitel 3 .....</b>	<b>42</b>

<b>4 ANDROID STUDIO KENNENLERNEN.....</b>	<b>43</b>
4.1 Projekt anlegen.....	43
4.2 Bedienoberfläche entwerfen .....	47
4.3 Click-Behandlungsmethode für den Schalter erstellen.....	54
4.4 Programm ausführen.....	58
4.5 Sichern und Wiederherstellen.....	61
4.6 Instant Run .....	62
4.7 Arbeitshilfen im Android Studio .....	62
4.7.1 Syntaxvervollständigung .....	62
4.7.2 Code-Inspektion und QuickFixes .....	63
4.7.3 Orientierungshilfen .....	64
4.7.4 Refaktorieren .....	65
4.7.5 Javadoc-Information abrufen .....	65
4.7.6 Sonstige Hinweise .....	66
4.7.6.1 Änderungen zurücknehmen.....	66
4.7.6.2 Android-Bedienoberfläche zurücksetzen .....	67
4.7.6.3 Parameternamen im Quellcodeeditor .....	67
4.8 Eigene Apps auf einem beliebigen Android-Gerät installieren.....	68
4.9 Projektkonfiguration einsehen und ändern .....	71
4.10 Kursprojekte öffnen .....	73
4.11 Übungsaufgaben zu Kapitel 4.....	73
<b>5 KOMPONENTEN UND SYSTEMUMGEBUNG EINER APP .....</b>	<b>75</b>
5.1 Komponenten einer Android-App .....	75
5.2 Manifest.....	77
5.3 Task und Back Stack .....	79
5.4 Apps und Prozesse.....	80
5.5 Sicherheit .....	82
5.6 Übungsaufgaben zu Kapitel 5 .....	83
<b>6 LEBENSZYKLUS EINER AKTIVITÄT .....</b>	<b>84</b>
6.1 Zustände.....	84
6.2 Zustandswechselmethoden .....	87
6.2.1 onCreate() .....	87
6.2.2 onStart() .....	87
6.2.3 onRestoreInstanceState() .....	88
6.2.4 onResume().....	88
6.2.5 onSaveInstanceState().....	88
6.2.6 onPause() .....	89
6.2.7 onStop() .....	90
6.2.8 onDestroy().....	90

<b>6.3 App zur Lifecycle-Demonstration.....</b>	<b>90</b>
6.3.1 Logging.....	92
6.3.2 Protokollausgaben in den Methoden für Activity-Zustandswechsel.....	94
6.3.2.1 Neustart nach Konfigurationsänderung.....	96
6.3.2.2 Neukreation nach Prozessterminierung.....	97
<b>6.4 Speicherverbrauch einer App .....</b>	<b>98</b>
6.4.1 Speicherinformationsmethoden .....	99
6.4.2 Beispielprogramm zur Speicherverschwendungen.....	100
6.4.3 Speicherbedarf per Profiler beobachten.....	102
6.4.4 OutOfMemoryError.....	103
<b>6.5 Übungsaufgaben zu Kapitel 6 .....</b>	<b>104</b>
<b>7 RESSOURCEN .....</b>	<b>105</b>
<b>7.1 Ressourcentypen.....</b>	<b>105</b>
<b>7.2 Konfigurationsabhängige Ressourcen.....</b>	<b>110</b>
<b>7.3 Ressourcen ansprechen.....</b>	<b>112</b>
7.3.1 Plattform-Ressourcen .....	112
7.3.2 Zugriff in einer Ressourcendefinitionsdatei.....	112
7.3.3 Zugriff im Java-Quellcode.....	114
7.3.4 Steuerelement-IDs .....	116
<b>7.4 Zeichenfolgen.....</b>	<b>117</b>
<b>7.5 Größenangaben .....</b>	<b>120</b>
<b>7.6 Farben .....</b>	<b>121</b>
<b>7.7 Zeichenbare Ressourcen.....</b>	<b>122</b>
7.7.1 Neun-Feld - Bitmaps .....	122
7.7.2 Zustandslisten .....	125
<b>7.8 Animationen .....</b>	<b>126</b>
<b>7.9 Mediendateien .....</b>	<b>128</b>
<b>7.10 Entspannung und Motivationsstärkung .....</b>	<b>129</b>
7.10.1 Projekt anlegen .....	129
7.10.2 Bedienoberfläche entwerfen .....	130
7.10.3 Click-Behandlungsmethode für den Schalter erstellen .....	133
<b>7.11 Übungsaufgaben zu Kapitel 7.....</b>	<b>137</b>
<b>8 BEDIENOBERFLÄCHE .....</b>	<b>138</b>
<b>8.1 Layoutdefinition per XML .....</b>	<b>139</b>
<b>8.2 Steuerelementklassen .....</b>	<b>141</b>
8.2.1 Die View - Klassenhierarchie .....	141
8.2.2 Elementare Steuerelementattribute bzw. -eigenschaften .....	142
8.2.2.1 Layout-Parameter .....	142
8.2.2.2 Attributes-Fenster des Layout-Editors im Android Studio .....	143
8.2.2.3 Steuerelement-ID .....	143
8.2.2.4 Breite und Höhe .....	144
8.2.2.5 Ränder (Margins) und Einrückung (Padding) .....	145

<b>8.3 Container .....</b>	<b>145</b>
8.3.1 LinearLayout .....	146
8.3.2 RelativeLayout.....	150
8.3.3 ConstraintLayout .....	153
8.3.3.1 Constraints in Bezug auf den Container .....	154
8.3.3.2 Constraints in Bezug auf andere Steuerelemente .....	156
8.3.3.3 match_constraint .....	159
8.3.3.4 Restriktionen aufheben.....	159
8.3.3.5 Ausrichtung an Orientierungslinien oder Barrieren .....	160
8.3.3.6 Steuerelementketten mit Platzaufteilung.....	162
8.3.3.7 Automatisch erstellte Constraints.....	164
8.3.3.8 Größenangabe per Verhältnis.....	165
8.3.4 FrameLayout.....	166
8.3.5 TableLayout.....	168
8.3.6 GridLayout .....	170
<b>8.4 Ereignisbehandlung .....</b>	<b>174</b>
8.4.1 Elementare und aufbereitete Ereignisse .....	174
8.4.2 Optionen zum Registrieren von Ereignisempfängern .....	175
8.4.2.1 Ereignisbehandlung durch eine anonyme Klasse .....	175
8.4.2.2 Ereignisbehandlung durch einen Lambda-Ausdruck .....	176
8.4.2.3 Klick-Ereignismethode per XML-Attribut registrieren.....	178
8.4.3 Ereignisverarbeitungskette (unterbrechen) .....	178
<b>8.5 Bedienelemente .....</b>	<b>179</b>
8.5.1 Beschriftungen (TextView) .....	179
8.5.1.1 Schriftfamilien und -attribute .....	179
8.5.1.2 Attribut freezeText.....	181
8.5.1.3 Attribut autoLink.....	181
8.5.2 Texteingabefelder (EditText).....	182
8.5.2.1 Optik .....	183
8.5.2.2 Irreguläre Eingaben verhindern.....	183
8.5.2.3 Mehrzeiliges Texteingabefeld .....	185
8.5.2.4 Fullscreen-Texteingabe .....	187
8.5.2.5 AutoCompleteTextView .....	188
8.5.2.6 Weitere Attribute für EditText-Elemente .....	188
8.5.3 Schaltflächen .....	189
8.5.4 Umschalter.....	192
8.5.4.1 Kontrollkästchen .....	195
8.5.4.2 Toggle-Button und Switch .....	196
8.5.4.3 Optionsschalter.....	197
8.5.5 Einfachauswahl per Spinner-Element.....	199
8.5.6 Fortschrittsanzeige .....	202
8.5.7 ListView und ListActivity .....	205
<b>8.6 Menüs .....</b>	<b>208</b>
8.6.1 Optionsmenü.....	209
8.6.1.1 Hardware-Menütaste versus App Bar .....	209
8.6.1.2 Menüdeklaration per XML-Datei.....	209
8.6.1.3 Menü-Ressource einfügen.....	214
8.6.1.4 Optionsmenü zur Laufzeit anpassen .....	215
8.6.1.5 Menüereignisbehandlung .....	216
8.6.1.6 Icons auf der Hauptebene des Optionsmenüs.....	217
8.6.2 Kontextmenü .....	219
8.6.2.1 Schwebendes Kontextmenü .....	219
8.6.2.2 Kontextabhängige Aktionen.....	221
8.6.3 PopUp-Menü .....	224
<b>8.7 Übungsaufgaben zu Kapitel 8 .....</b>	<b>226</b>

<b>9 INTENTS UND INTENT-FILTER .....</b>	<b>228</b>
<b>  9.1 Bestandteile eines Intent-Objekts .....</b>	<b>229</b>
9.1.1 Name der Komponente .....	229
9.1.2 Aktion .....	230
9.1.3 Daten.....	231
9.1.4 Kategorien .....	233
9.1.5 Extras.....	234
9.1.6 Flags .....	234
<b>  9.2 Intent-Filter .....</b>	<b>234</b>
9.2.1 Bestandteile eines Intent-Filters.....	235
9.2.2 data-Element.....	237
<b>  9.3 Eigene Activity über einen expliziten Intent starten .....</b>	<b>238</b>
9.3.1 Einfacher Aktivitätsstart .....	240
9.3.2 Navigation innerhalb der eigenen App .....	241
9.3.3 Extradaten übergeben und in der sekundären Aktivität verwenden.....	243
9.3.4 Rückmeldung der sekundären Aktivität auswerten .....	244
<b>  9.4 Fremde Aktivitäten über implizite Intents starten.....</b>	<b>245</b>
9.4.1 Intent-Auflösung.....	245
9.4.2 Aktivitätsstart ohne Rückmeldung.....	247
9.4.3 Wahl zwischen alternativen Intent-Auftragnehmern .....	250
9.4.4 Rückmeldung der sekundären Aktivität auswerten .....	251
<b>  9.5 Intents in Wartestellung .....</b>	<b>252</b>
<b>  9.6 Übungsaufgaben zu Kapitel 9 .....</b>	<b>252</b>
<b>10 MULTITHREADING .....</b>	<b>254</b>
<b>  10.1 Main-Thread und Multithreading .....</b>	<b>255</b>
<b>  10.2 Hintergrund-Thread mit elementaren Mitteln realisieren .....</b>	<b>257</b>
<b>  10.3 Altlasten durch schlecht programmierte Helfer-Threads.....</b>	<b>261</b>
<b>  10.4 AsyncTask .....</b>	<b>262</b>
<b>  10.5 Nachrichtenübermittlung zwischen verschiedenen Threads .....</b>	<b>267</b>
<b>  10.6 Übungsaufgaben zu Kapitel 10.....</b>	<b>269</b>
<b>11 DIENSTE .....</b>	<b>271</b>
<b>  11.1 Lebenszyklusmethoden bei Diensten .....</b>	<b>273</b>
<b>  11.2 Gestarteter Dienst aus der Klasse IntentService .....</b>	<b>273</b>
11.2.1 Mit Android 8 eingeführte Restriktionen für gestartete Dienste .....	273
11.2.2 Besonderheiten der Klasse IntentService.....	274
11.2.3 IntentService in ein Projekt aufnehmen .....	275
11.2.4 Dienst starten und stoppen .....	277
11.2.5 Ergebniskommunikation per LocalBroadcastManager .....	278

<b>11.3 Gebundener Dienst.....</b>	<b>281</b>
11.3.1 Rohling für einen bindungsfähigen Dienst per Entwicklungsumgebung erstellen .....	281
11.3.2 Bindung herstellen und aufheben.....	283
11.3.3 Klienten-Server - Kommunikation via Binder-Objekt und Handler .....	284
11.3.3.1 ServiceConnection .....	284
11.3.3.2 Binder.....	285
11.3.3.3 Ergebniskommunikation über Message-Objekte .....	286
11.3.4 Service-Ableitung definieren .....	288
<b>11.4 Vordergrunddienste .....</b>	<b>290</b>
<b>11.5 Übungsaufgaben zu Kapitel 11.....</b>	<b>291</b>
<b>12 FRAGMENTE .....</b>	<b>292</b>
<b>12.1 Aufgabenverteilung zwischen Aktivitäten und Fragmenten .....</b>	<b>293</b>
<b>12.2 Vorschlag für ein AVD-Tablet .....</b>	<b>294</b>
<b>12.3 Beispiel für die Verwendung von Fragmenten.....</b>	<b>297</b>
<b>12.4 Fragmente in eine Aktivität integrieren .....</b>	<b>298</b>
<b>12.5 Lebenszyklusmethoden bei Fragmenten .....</b>	<b>300</b>
<b>12.6 ListFragment-Initialisierung .....</b>	<b>302</b>
<b>12.7 Fragment-Transaktionen.....</b>	<b>305</b>
12.7.1 Der Fragment-Manager.....	306
12.7.2 Die Klasse FragmentTransaction .....	307
12.7.3 Back Stack für Fragment-Transaktionen .....	308
12.7.4 Animierte Übergänge.....	309
<b>12.8 Argument-Bundles und Fabrikmethoden .....</b>	<b>309</b>
<b>12.9 Fragmente im Single-Pane – Modus .....</b>	<b>311</b>
<b>12.10 Übungsaufgaben zu Kapitel 12.....</b>	<b>313</b>
<b>13 DATENBANKVERWALTUNG MIT ROOM UND SQLITE .....</b>	<b>314</b>
<b>13.1 Benötigte Typen für den Datenbankzugriff via Room.....</b>	<b>315</b>
13.1.1 Entities .....	315
13.1.1.1 Annotation @Entity .....	316
13.1.1.2 Öffentlich zugängliche Felder .....	316
13.1.1.3 Primärschlüssel .....	317
13.1.2 DAO (Data Access Object).....	317
13.1.2.1 INSERT.....	318
13.1.2.2 UPDATE.....	318
13.1.2.3 DELETE .....	319
13.1.2.4 SELECT .....	319
13.1.3 RoomDatabase .....	320
13.1.4 Typkonverter.....	320
<b>13.2 Ereignistagebuch anlegen und befüllen .....</b>	<b>321</b>
13.2.1 Room in die Abhängigkeitsliste des Moduls aufnehmen .....	322
13.2.2 Entity Event .....	323
13.2.3 Schnittstelle EventDAO als Grundlage für das Data Access Object .....	324
13.2.4 Typkonverter.....	325
13.2.5 RoomDatabase-Ableitung .....	325

<b>13.3 Ereignisliste aus der Datenbank abrufen und per ListView anzeigen.....</b>	<b>327</b>
13.3.1    Aktivität zum Anzeigen der Ereignisliste .....	327
13.3.2    Adapter-Klasse für das ListView-Steuerelement .....	328
<b>13.4 Neue Ereignisse erfassen.....</b>	<b>331</b>
<b>13.5 Tabellenzeilen aktualisieren .....</b>	<b>333</b>
<b>13.6 Tabellenzeilen per Kontextmenü löschen .....</b>	<b>337</b>
<b>13.7 Weitere wichtige Room-Optionen.....</b>	<b>339</b>
<b>13.8 Direktkontakt mit SQLite per ADB-Konsole.....</b>	<b>340</b>
13.8.1    SDK-Kommando adb shell .....	340
13.8.2    Interaktion mit SQLite .....	340
<b>14 CONTENT PROVIDER NUTZEN UND ANBIETEN.....</b>	<b>343</b>
<b>14.1 Content Provider fremder Anwendungen nutzen .....</b>	<b>343</b>
14.1.1    Content-URI.....	344
14.1.2    Cursor .....	344
14.1.3    Abfragen .....	345
14.1.4    Struktur des Kontakte-Providers in Android.....	349
14.1.5    Modifikationen.....	350
14.1.6    SQL-Injektion .....	353
14.1.7    Zugriffsrechte.....	353
14.1.8    Alternative Techniken für den Zugriff auf einen Content Provider .....	355
14.1.9    Dokumentation und Kontrakt-Klasse.....	356
<b>14.2 Content Provider erstellen.....</b>	<b>357</b>
14.2.1    Content-URLs und andere Bestandteile der Schnittstelle definieren .....	357
14.2.2    ContentProvider-Ableitung implementieren .....	358
14.2.2.1    Room Persistence Library .....	360
14.2.2.2    UriMatcher .....	362
14.2.2.3    onCreate().....	363
14.2.2.4    query().....	363
14.2.2.5    insert() .....	363
14.2.2.6    delete() .....	364
14.2.2.7    update() .....	364
14.2.2.8    getType().....	365
14.2.3    Provider-Eintrag im Anwendungsmanifest.....	365
14.2.4    Zugriffsrechte definieren .....	366
14.2.5    Modifikationen der bisherigen Event Diary - Klassen.....	367
14.2.5.1    Datenbankzugriffe nur noch im Content Provider .....	367
14.2.5.2    CursorAdapter .....	368
<b>14.3 Loader und LoaderManager .....</b>	<b>370</b>
14.3.1    Wichtige Typen im Loader-API .....	370
14.3.2    LoaderManager.LoaderCallbacks<Cursor> - Methoden implementieren .....	371
14.3.3    Loader-Initialisierung und -Aktualisierung.....	372
<b>15 EINSTELLUNGEN.....</b>	<b>375</b>
<b>15.1 Preferences-Ressourcendatei.....</b>	<b>375</b>
<b>15.2 Einstellungsdateien einer App.....</b>	<b>379</b>
<b>15.3 Preferences-Bedienoberfläche .....</b>	<b>380</b>
15.3.1    Einstellungsaktivität bzw. -fragment .....	380
15.3.2    Einstellungsaktivität via Optionsmenü starten .....	383

<b>15.4 Einstellungsdaten per Programm schreiben und lesen .....</b>	<b>383</b>
15.4.1 Die Klasse SharedPreferences .....	383
15.4.2 Einstellungsdaten schreiben .....	384
15.4.3 Einstellungsdaten lesen .....	385
<b>15.5 Sonstige Optionen bei der Einstellungsverwaltung .....</b>	<b>386</b>
<b>16 BROADCAST RECEIVER .....</b>	<b>387</b>
<b>16.1 Registrierung per Manifest.....</b>	<b>387</b>
<b>16.2 Broadcast-Intents verarbeiten.....</b>	<b>390</b>
16.2.1 Startschwierigkeiten.....	390
16.2.2 Zeitlich begrenzte Arbeitserlaubnis .....	391
16.2.3 Akzeptiertes Verhalten.....	391
16.2.4 Lebenslauf.....	392
16.2.5 Verbot von Dialogen.....	392
16.2.6 Aufgaben asynchron im Hintergrund ausführen .....	392
<b>16.3 Dynamische Registrierung.....</b>	<b>394</b>
<b>16.4 Broadcast-Nachrichten versenden .....</b>	<b>396</b>
<b>16.5 Broadcast-Varianten .....</b>	<b>397</b>
16.5.1 (Un)geordnete Verarbeitung .....	397
16.5.2 Sticky Broadcasts.....	398
<b>16.6 Sicherheit.....</b>	<b>399</b>
16.6.1 Beschränkung der zulässigen Absender .....	399
16.6.2 Beschränkung der zulässigen Empfänger .....	401
<b>16.7 Prozessinterne Broadcast-Nachrichten .....</b>	<b>401</b>
<b>LITERATUR .....</b>	<b>402</b>
<b>ANHANG .....</b>	<b>403</b>
<b>A. Lösungsvorschläge zu den Übungsaufgaben.....</b>	<b>403</b>
Kapitel 1 (Java und Android) .....	403
Kapitel 4 (Android Studio kennenlernen) .....	403
Kapitel 5 (Komponenten und Systemumgebung einer App) .....	403
Kapitel 6 (Lebenszyklus einer Aktivität) .....	403
Kapitel 7 (Ressourcen) .....	404
Kapitel 8 (Bedienoberfläche) .....	404
Kapitel 9 (Intents und Intent-Filter) .....	404
Kapitel 10 (Multithreading) .....	405
Kapitel 11 (Dienste) .....	405
Kapitel 12 (Fragmente) .....	406
<b>INDEX.....</b>	<b>407</b>

---

# 1 Java und Android

Die von der Firma **Sun Microsystems** (mittlerweile von der Firma **Oracle** übernommen) entwickelte Programmiersprache Java ist zwar mit dem Internet groß geworden, hat sich jedoch mittlerweile als universelle, für vielfältige Zwecke einsetzbare Lösung etabliert. Unter den objektorientierten Programmiersprachen hat Java den größten Verbreitungsgrad, und dieses Paradigma der Softwareentwicklung hat sich praktisch in der gesamten Branche als *state of the art* etabliert.

Es spricht für das Potential von Java, dass diese Programmiersprache als Standard für die Entwicklung von Android-Apps gewählt worden ist. Aktuell trägt Android erheblich zur Attraktivität von Java bei, denn das Smartphone-Betriebssystem der Firma **Google** hat mittlerweile einen Weltmarktanteil von ca. 85% erreicht<sup>1</sup> (Stand 2017, Tendenz steigend) und zeigt auch auf dem Tablet-Markt eine (wenngleich mit ca. 63% weniger ausgeprägte) Dominanz.<sup>2</sup> Außerdem entwickeln sich weitere Einsatzfelder für Android (z. B. Autos, Armbanduhren, Fernseher). Somit ist Android derzeit die am stärksten verbreitete Plattform für klientenseitige Java-Programmierung (Mednieks et al., 2013, S. 41).

Nachdem Java im weithin akzeptierten TIOBE -Index zur Verbreitung bzw. Beliebtheit von Programmiersprachen im April 2015 den Platz 1 von der Programmiersprache C zurückerober hat, wurde diese Entwicklung auf der TIOBE-Webseite folgendermaßen erklärt:<sup>3</sup>

Java is back at the top of the TIOBE index after one and half year. Androids ongoing success is probably the major reason for Java's revival.

Bei der Software-Entwicklung mit Java (so auch für Android) sind drei Säulen beteiligt:

- Die **Programmiersprache**  
Die nach den Regeln der Programmiersprache erstellten Quellcodedateien mit den Klassendefinitionen werden vom Compiler in Bytecode-Dateien übersetzt (siehe Abschnitt 1.1).
- Die **Standardklassenbibliothek** mit ausgereiften Lösungen für (fast) alle Routineaufgaben  
Hier bestehen Unterschiede zwischen Java-Editionen für verschiedene Einsatzfelder.
- Die **Laufzeitumgebung** (virtuelle Java-Maschine) mit zahlreichen Aufgaben bei der Ausführung von Java-Programmen

## 1.1 Quellcode, Bytecode und Maschinencode

Java stellt als Programmiersprache Ausdrucksmittel zur Modellierung von Anwendungsbereichen und zur Formulierung von Algorithmen bereit. Unter einem *Programm* verstehen wir zunächst den vom Entwickler zu formulierenden *Quellcode*. Anschließend ist z. B. eine Java-Methode zu sehen, die einen Bruch mit Zähler und Nenner in zwei Texteingabefeldern einer Android-Bildschirmseite (Aktivität) kürzt:

---

<sup>1</sup> <https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/>

<sup>2</sup> <https://www.statista.com/statistics/273840/global-market-share-of-tablet-operating-systems-since-2010/>

<sup>3</sup> <https://www.tiobe.com/tiobe-index/>

Den ersten Platz im TIOBE-Index hat Java übrigens seit der Rückeroberung im April 2015 bis heute (Juli 2018) gehalten.

```

public void onClick(View v) {
    EditText etZaeher = findViewById(R.id.zaeher);
    EditText etNenner = findViewById(R.id.nenner);
    String sz = etZaeher.getText().toString();
    String sn = etNenner.getText().toString();
    if (sz.length() == 0 || sn.length() == 0)
        return;
    int z = Integer.parseInt(sz);
    int n = Integer.parseInt(sn);

    if (z * n != 0) {
        int rest;
        int ggt = Math.abs(z);
        int divisor = Math.abs(n);
        do {
            rest = ggt % divisor;
            ggt = divisor;
            divisor = rest;
        } while (rest > 0);
        z /= ggt;
        n /= ggt;
        etZaeher.setText(Integer.toString(z));
        etNenner.setText(Integer.toString(n));
    }
}

```

Während *Ihnen* das Lesen und Schreiben von Java-Quellcode bald problemlos gelingt, kann die CPU (*Central Processing Unit*) eines Rechners nur einen maschinenspezifischen Satz von Befehlen verstehen, die als Folge von Nullen und Einsen (= *Maschinencode*) formuliert werden müssen. Die ebenfalls CPU-spezifische Assembler-Sprache stellt eine für Menschen lesbare Form des Maschinencodes dar. Mit dem Assembler- bzw. Maschinenbefehl

**mov eax, 4**

einer CPU aus der x86-Familie wird z. B. der Wert 4 in das EAX-Register (ein Speicherort im Prozessor) geschrieben. Die CPU holt sich einen Maschinenbefehl nach dem anderen aus dem Hauptspeicher und führt ihn aus, heutzutage immerhin mehrere Milliarden Befehle pro Sekunde (*Instructions Per Second, IPS*). Ein Quellcode-Programm muss also erst in Maschinencode übersetzt werden, damit es von einem Rechner ausgeführt werden kann. Dies geschieht bei Java in der Regel (u.a. aus Gründen der Portabilität) in zwei Schritten:

### Kompilieren: Quellcode → Bytecode

Der (z. B. mit einem beliebigen Texteditor verfasste) Quellcode wird vom **Compiler** in einen maschinen-unabhängigen **Bytecode** übersetzt. Dieser besteht aus den Befehlen einer **virtuellen Maschine**, die sich durch ihren vergleichsweise einfachen Aufbau gut auf aktuelle Hardware-Architekturen abbilden lässt. Wenngleich der Bytecode von den heute üblichen Prozessoren noch nicht direkt ausgeführt werden kann, hat er doch bereits die meisten Verarbeitungsschritte auf dem Weg vom Quell- zum Maschinencode durchlaufen. Sein Name geht darauf zurück, dass die Instruktionen der virtuellen Maschine jeweils genau ein Byte (= 8 Bit) lang sind.

Quellcode-Dateien tragen in Java die Namenserweiterung **.java**, Bytecode-Dateien die Erweiterung **.class**.

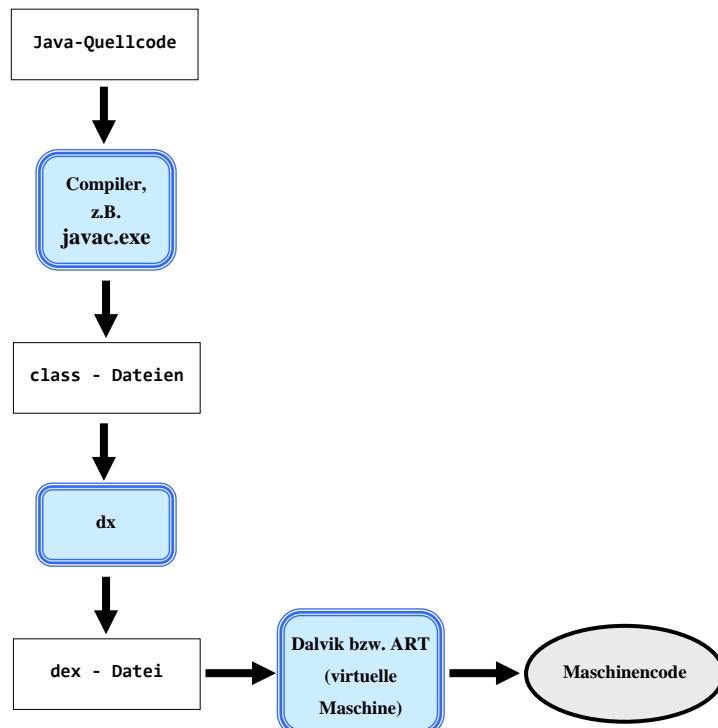
## Interpretieren: Bytecode → Maschinencode

Abgesehen von den seltenen Systemen mit realem Java-Prozessor muss für jede Betriebssystem/CPU - Kombination mit Java-Unterstützung ein (naturgemäß plattformabhängiger) **Interpreter** erstellt werden, der den Bytecode zur Laufzeit in die jeweilige Maschinensprache übersetzt. Man benötigt also für jede reale Maschine eine vom jeweiligen Wirtsbetriebssystem abhängige JVM, um den Java-Bytecode auszuführen.

Mittlerweile kommen bei der Ausführung von Java-Programmen leistungssteigernde Techniken (**Just-in-Time - Compiler**) zum Einsatz, welche die Bezeichnung *Interpreter* fraglich erscheinen lassen. Allerdings ändert sich nichts an der Aufgabe, aus dem plattformunabhängigen Bytecode den zur aktuellen Hardware passenden Maschinencode zu erzeugen.

Bei der Laufzeitumgebung für Android hat die Firma Google eigene Wege beschritten und eine virtuelle Maschine namens **Dalvik** entworfen, die nach dem JIT-Prinzip operiert. Während sich bei Dalvik an der grundlegenden Arbeitsweise im Vergleich zur Java-Runtime der Firma Oracle wenig geändert hat, konnte der Umfang des Bytecodes immerhin ungefähr halbiert werden.

Wie die folgende Abbildung zeigt, startet auch die Erstellung einer Android-App mit der Übersetzung von Java-Quellcode (untergebracht in Textdateien mit der Namenserweiterung **.java**) in konventionellen Java-Bytecode (untergebracht in Binärdateien mit der Namenserweiterung **.class**):



Anschließend übersetzt das Programm **dx** die **class**-Dateien in Dalvik-Bytecode und produziert dabei eine Datei mit der Namenserweiterung **.dex**, die alle Klassen der Anwendung enthält. Zusammen mit anderen Anwendungsbestandteilen wird der Dalvik-Bytecode in einer **APK**-Datei zu den Zielgeräten geliefert, dort installiert und schlussendlich von Dalvik in Maschinencode übersetzt und ausgeführt.

Mit Android 5.x (*Lollipop*) ist Dalvik durch eine Weiterentwicklung namens **ART (Android Runtime)** abgelöst worden, wobei die Übersetzung von Byte- in Maschinencode nicht bei jeder Programmausführung passiert, sondern nach dem **Ahead-Of-Time - Prinzip (AOT)** einmalig bei der Installation, was zu einer flüssigeren Arbeitsweise von Android-Apps führen soll (Becker & Pant

2015, S. 27). Schon mit Android 7 (*Nougat*) ist allerdings wieder ein JIT-Compiler als Ergänzung zum AOT-Compiler im System und sorgt für folgender „Verbesserungen“:<sup>1</sup>

- Schnellere Installationen („weil das Ahead-Of-Time - Übersetzen entfällt“)
- Einsparung von RAM- und Flash-Speicher

Im Zusammenhang mit der Unterstützung von Java 8 hatte Google seit dem Jahr 2014 daran gearbeitet, beim Erstellungsprozess die Werkzeuge **javac** und **dx** durch Alternativen namens Jack und Jill zu ersetzen:<sup>2</sup>

- *Java Android Compiler Kit (JACK)* in der Datei **jack.jar**  
Der Compiler Jack sollte Quellcode direkt in das **dex**-Format übersetzen.
- *Jack Intermediate Library Linker (JILL)*. in der Datei **jill.jar**  
Um konventionellen Java-Bytecode (mit Klassenbibliotheken) einzubinden, sollte der Compiler Jill aktiv werden und die **class**-Dateien in ein Format übersetzen, das Jack verarbeiten kann.

Der Plan wurde aber 2017 wieder aufgegeben, weil sich die Integration der Werkzeuge von Drittanbietern (z. B. zur Code-Verschleierung oder -Optimierung) als problematisch herausgestellt hat.<sup>3</sup>

Nachdem Intel im Jahr 2016 die Entwicklung von Smartphone-Chips gestoppt hat, verwenden die aktuell (im April 2018) erhältlichen Android-Smartphones und -Tablets meist eine CPU mit einem Design der britischen Prozessorschmiede ARM (*Advanced RISC Machines*).<sup>4</sup> Grundsätzlich läuft Android mit seiner Java/Linux - Basis natürlich auch auf x86 - Hardware. Lange gab es mit Remix OS eine Android-Variante für Desktop- und Notebook-Rechner mit x86 - CPU, die aber in 2017 eingestellt wurde.

## 1.2 Standardbibliothek

Damit die Programmierer nicht das Rad (und ähnliche Dinge) ständig neu erfinden müssen, bietet jede Java-Edition eine Standardbibliothek mit fertigen Klassen für nahezu alle Routineaufgaben, die oft als **API (Application Program Interface)** bezeichnet wird. Im Manuskript werden Sie zahlreiche API-Klassen kennenlernen. Eine vollständige Behandlung ist wegen des enormen Umfangs unmöglich und auch nicht erforderlich.

Wir halten fest, dass die Java-Technologie einerseits auf einer Programmiersprache mit einer bestimmten Syntax und Semantik basiert, dass aber andererseits die Funktionalität im Wesentlichen von einer umfangreichen Standardbibliothek beigesteuert wird. Neue Funktionalitäten werden in der Regel durch eine Erweiterung der Standardklassenbibliothek realisiert, sodass hier erhebliche Änderungen stattfinden. Einige Klassen sind mittlerweile schon als *deprecated* (überholt, zurückgestuft, nicht mehr zu benutzen) eingestuft worden. Gelegentlich stehen für eine Aufgabe verschiedene Lösungen aus unterschiedlichen Entwicklungsstadien zur Verfügung.

Im Vergleich zur Java Standard Edition (JSE) besitzt das Android-Java eine deutlich verschiedene Standardbibliothek, um den Besonderheiten von Smartphones und Tablets gerecht werden zu können:

---

<sup>1</sup> [https://developer.android.com/about/versions/nougat/android-7.0.html#jit\\_aot](https://developer.android.com/about/versions/nougat/android-7.0.html#jit_aot)

<sup>2</sup> <http://www.androidpolice.com/2014/11/30/jack-and-jill-are-googles-new-compilers-for-android-app-developers/>

<sup>3</sup> <https://www.heise.de/developer/meldung/Google-verabschiedet-sich-von-der-Jack-Toolchain-fuer-Android-3653696.html>

<sup>4</sup> <https://www.heise.de/newsticker/meldung/Intel-outside-Smartphone-SoCs-gestoppt-3195240.html>

- kleine Displays bei hoher Variabilität von Größe und Seitenverhältnis
- Touchscreen statt Maus
- relativ geringe Prozessorleistung

In den Hintergrund geratene Anwendungen, die nicht mehr mit dem Benutzer interagieren können, stellen in der Regel alle Ressourcen konsumierenden Tätigkeiten ein.

- relativ kleiner Arbeitsspeicher

Bei Speichermando werden in den Hintergrund geratene Anwendungen automatisch aus dem Speicher entfernt, wobei mit der inaktiven Zeit die Wahrscheinlichkeit für einen Rauswurf steigt.

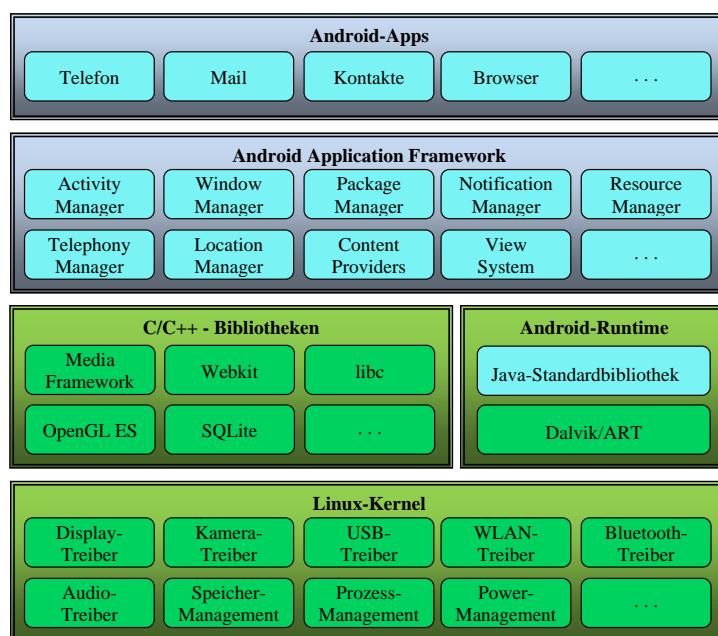
- Zwang zum Energiesparen
- Verfügbarkeit spezieller Sensoren (z. B. Beschleunigung, GPS, Kamera)

Die meisten Bestandteile der JSE-Standardbibliothek sind in weitgehend kompatibler Form auch unter Android anzutreffen. Es kommen notwendigerweise Klassen hinzu, welche die spezielle Hardware und die eigenständige Anwendungslogik von Android unterstützen. Insbesondere enthält Android eine eigene Bibliothek zur Gestaltung von grafischen Bedienoberflächen.

Die von Android verwendete Java-Standardbibliothek basiert seit der Version 7 auf dem **Open-JDK**, das von der Firma Oracle gemeinsam mit der Open-Source - Community entwickelt wird.

### 1.3 Android-Systemarchitektur

Eine virtuelle Java-Maschine und eine Standardbibliothek alleine genügen nicht, um eine Smartphone-Hardware zum Leben zu erwecken. Abgesehen von der eher theoretischen Ausnahme realer Java-Prozessoren benötigt Java stets ein Wirtsbetriebssystem, und die Firma Google hat den Kern des stabilen und quelloffenen Betriebssystems Linux als Basis für Android gewählt. In der folgenden Abbildung wird die Android-Systemarchitektur skizziert:



Durch die Färbung wird zum Ausdruck gebracht, in welcher Programmiersprache die Systembestandteile erstellt wurden:

- Grün gefärbte Komponenten sind in C oder C++ programmiert. Sie gehören zum Bereich der Systemprogrammierung, mit der wir uns nicht beschäftigen werden.

- Blau gefärbte Komponenten sind in Java programmiert. Dazu gehören die von uns erstellten Anwendungsprogramme sowie die Systemumgebung, von der unserer Anwendungsprogramme unterstützt und verwaltet werden.
- Für die Steuerung der Hardware, die Verwaltung von Speicher und Prozessen, für die Sicherheit und vieles mehr ist ein Linux-Kernel zuständig. Dies ist ein stabiler, quelloffener und sehr verbreiteter Betriebssystemkern.
- Über dem Kernel sitzen in C/C++ geschriebene Bibliotheken für elementare Dienstleistungen (z. B. OpenGL zur Grafikausgabe, SQLite zur Datenbankverwaltung). Auf dieser Ebene befindet sich auch die virtuelle Java-Maschine zur Ausführung des Android-spezifischen **dex**-Codes (Dalvik bzw. ART), die in C++ geschrieben wurde. Demgegenüber wurde die Java-Standardbibliothek (mit Klassen wie **System**, **String**, **Socket**) natürlich in Java geschrieben.
- Das Android Application Framework ist der Rahmen (mit Regeln und Dienstleistungen), in dem von uns erstellte Android-Software erfolgreich agieren kann. Es ist dafür gesorgt, dass sich unsere Apps installieren und starten lassen, mit anderen Apps kooperieren können usw. Für seine Dienstleistungen benötigt das Android Application Framework die darunter liegende Schicht von Bibliotheken.
- Zur Anwendungsschicht gehören mit Android ausgelieferte Apps (z. B. zum Telefonieren, zur Verwaltung von Kontakten) sowie die von uns erstellten Apps. Jede Android-App läuft in einer eigenen virtuellen Java-Maschine, und diese agiert in einem eigenen Prozess des Linux-Systems.

#### 1.4 Android-Versionen und API-Levels

Seit dem Erscheinen der ersten Version im Jahr 2008 hat Android eine lange Entwicklungsgeschichte hinter sich gebracht, wobei zahlreiche Versionen und zugehörige Ausbaustufen der Standardbibliothek erschienen sind. Während die Android-Version für die Identifikation gegenüber dem Anwender gedacht ist, wird auf technischer Ebene (z. B. im Programm, in Google Play) das API-Level verwendet, um Kompatibilitätsfragen zu klären. Auf der Webseite

<https://developer.android.com/guide/topics/manifest/uses-sdk-element>

stellt Google die Bezeichnungen gegenüber:

Android-Version	API Level	VERSION_CODE
1.0	1	BASE
1.1	2	BASE_1_1
1.5	3	CUPCAKE
1.6	4	DONUT
2.0	5	ECLAIR
2.0.1	6	ECLAIR_0_1
2.1.x	7	ECLAIR_MR1
2.2.x	8	FROYO
2.3, 2.3.1, 2.3.2	9	GINGERBREAD
2.3.3, 2.3.4	10	GINGERBREAD_MR1
3.0.x	11	HONEYCOMB
3.1.x	12	HONEYCOMB_MR1
3.2	13	HONEYCOMB_MR2
4.0, 4.0.1, 4.0.2	14	ICE_CREAM SANDWICH
4.0.3, 4.0.4	15	ICE_CREAM SANDWICH_MR1
4.1.x	16	JELLY_BEAN
4.2.x	17	JELLY_BEAN_MR1
4.3	18	JELLY_BEAN_MR2
4.4	19	KITKAT

Android-Version	API Level	VERSION_CODE
4.4W	20	KITKAT_WATCH (nur Wearables)
5.0	21	LOLLIPOP
5.1	22	LOLLIPOP_MR1
6.0	23	M (MARSHMALLOW)
7.0	24	N (NOUGAT)
7.1, 7.1.1	25	N_MR1
8.0	26	O (Oreo)
8.1	27	O_MR1

Über die Verbreitung der einzelnen Versionen (gemessen durch die Zugriffe auf Google Play) informiert die folgende Webseite:

<https://developer.android.com/about/dashboards/index.html>

Welcher Java-Sprachumfang für die Android-Entwicklung verwendet werden kann, ist der folgenden Tabelle zu entnehmen:

Android API-Level	Unterstützter Java-Sprachumfang
Bis 18	6
Ab 19	7
Ab 24	8 (teilweise)

## 1.5 Dokumentation zu Android und seinem Java-API

Google bietet im Internet für Android-Entwickler viele Hilfen an:

- Entwicklungswerzeuge  
<https://developer.android.com/studio/>
- Unterrichtseinheiten  
<https://developer.android.com/guide/index.html>
- Dokumentation wichtiger Android-Bestandteile  
<https://developer.android.com/guide/components/activities/index.html>
- Technische Referenz (Dokumentation der Pakete, Klassen und Interfaces im API)  
<https://developer.android.com/reference/packages.html>

## 1.6 Alternativen zu Java bei der Entwicklung von Android-Apps

Java ist die offizielle Programmiersprache für Android, wurde für wesentliche Teile des Betriebssystems verwendet (siehe Abschnitt 1.3) und erhält von Google die stärkste Unterstützung. Sehr wahrscheinlich sind die meisten Apps im Google Play Store in Java geschrieben. Es gibt aber durchaus Alternativen, von denen anschließend einige ohne Anspruch auf Vollständigkeit aufgelistet werden:

- **Kotlin**  
Diese von der Firma JetBrains, die auch für das Android Studio verantwortlich zeichnet, entwickelte Sprache wird wie Java in Bytecode kompiliert und letztlich von einer virtuellen Java-Maschine ausgeführt. Google hat die Kotlin offiziell zur sekundären Entwicklungssprache für Android ernannt. Außerdem wird unter dem Namen *Kotlin/Native* eine Cross-Platform - Entwicklung für Android, iOS und andere Plattformen angeboten. Als neu konzipierte Sprache ist Kotlin sicher moderner als Java und noch ohne Altlasten. Ob sich Kotlin etabliert, ist allerdings keinesfalls sicher (aktueller Rang im TIOBE-Index: 49).
- **C#**

Die Programmiersprache C# ist syntaktisch und hinsichtlich der Anwendungsbreite mit Java vergleichbar. Die mittlerweile von Microsoft übernommene Firma Xamarin unterstützt eine auf C# (und .NET) basierende Entwicklung von plattformübergreifenden Apps für Android, iOS und Windows Phone. Es sind aber Einschränkungen und Probleme im Vergleich zu nativen Android- bzw. iOS-Apps vorhanden, und mit jeder neuen Version der Mobil-Betriebssysteme sind neue Kompatibilitätsprobleme zu erwarten.<sup>1</sup> Die Entwicklung einer Android-App gelingt mit C#/Xamarin sicherlich *nicht* leichter als mit Java/Android Studio. Wer allerdings C# beherrscht und nicht auf Java umsteigen möchte, hat ein starkes Argument für die Verwendung von Xamarin. Eine mit Xamarin entstandene Android-App ist jedoch keinesfalls direkt unter iOS einsetzbar. Die Angaben zum Anteil des wiederverwendbaren Codes schwanken stark mit einem Mittelwert von ca. 70%. Nachdem das Windows-Betriebssystem für Smartphones eingestellt wurde, ist die Motivation der Firma Microsoft zur Weiterentwicklung von Xamarin unsicher.

- **JavaScript-basierte plattformübergreifende Entwicklung**

Von der Firma Google wird ein Multiplattform-SDK namens **Flutter** entwickelt, das mit Googles JavaScript-Dialekt **Dart** arbeitet. Ähnlich ist die von Facebook unterstützte Lösung **React Native** konzipiert, die mit der JavaScript-Variante **TypeScript** arbeitet. Beide Entwicklungssysteme versprechen die Erstellung von Apps, die u.a. unter Android und iOS laufen. Man muss aber damit rechnen, dass die Unterstützung der Platform-spezifischen APIs an Grenzen stößt. Auf der FAQ-Webseite von Flutter wird die Frage

Can I access platform services and APIs like sensors and local storage?  
so beantwortet:<sup>2</sup>

Flutter gives developers out-of-the-box access to *some* platform-specific services and APIs from the operating system.

Auf einer FAQ-Webseite zu React Native heißt es:<sup>3</sup>

On the other hand, a native app is great when we consider using all the features that a platform offers, including such modules as video/audio processing or multithreading.

Eventuell stößt React Native also bereits beim Multithreading an Grenzen, und diese Programmietechnik ist auf einem System mit mehreren CPU-Kernen (also auf praktisch allen modernen Smartphones) für viele Apps unumgänglich. Wie sich im Manuskript zeigen wird, ist in Java-Apps für Android das Multithreading leicht zu realisieren. Bei anspruchsvollen Aufgaben sind die hybriden Lösungen vermutlich darauf angewiesen, Plattform-spezifische Software einzubinden, was dann zu komplexen Konstruktionen führt. Einfache Apps, die im Wesentlichen aus dem Internet bezogene Daten präsentieren, sind mit Flutter und React Native sicher gut zu realisieren, und die plattformübergreifende Architektur spart dann Entwicklungszeit ein.

## 1.7 Übungsaufgaben zu Kapitel 1

1) Welche von den folgenden Aussagen sind richtig bzw. falsch?

1. Die meisten aktuellen CPUs können Java-Bytecode direkt ausführen.
2. Für Android wurde eine angepasste Version der Programmiersprache Java entwickelt.
3. Für Android wurden eine angepasste Standardbibliothek und der besonders kompakte Dalvik-Bytecode entwickelt.

---

<sup>1</sup> Siehe z. B. <https://www.iflexion.com/blog/pros-cons-mobile-development-xamarin/>

<sup>2</sup> <https://flutter.io/faq/>

<sup>3</sup> <https://www.netguru.co/blog/react-native-faq>



---

## 2 Installation der Entwicklungsumgebung unter Windows

Zum Erstellen von Android-Apps hat die Firma Google einige Jahre lang auf die verbreitete Java-Entwicklungsumgebung **Eclipse** gesetzt und diese um Android-spezifische Werkzeuge erweitert (Android Developer Tools). Im Jahr 2014 hat Google jedoch die auf Eclipse basierende Lösung als offizielle Android-IDE durch das **Android Studio** ersetzt, das auf dem Produkt **IntelliJ IDEA Community Edition** der Firma **JetBrains** basiert.<sup>1</sup>

Anschließend wird die Installation der zum Erstellen und Testen von Android-Apps erforderlichen Software beschrieben, wobei aus Zeitgründen nur Windows berücksichtigt werden kann. Insgesamt werden ca. 10 GB Massenspeicher belegt.

Auf dem Entwicklungsrechner ist auch Arbeitsspeicher sehr willkommen. Unter Windows sollten es mindestens 4 GB sein, wobei diese Größe noch kein problemloses Arbeiten ermöglicht, sondern häufig zu Sparmaßnahmen zwingt, um zeitraubende Paging-Aktivitäten des Betriebssystems (Nutzung von Festplatten als virtueller Arbeitsspeicher) und Fehler zu vermeiden. Das gilt insbesondere beim sehr sinnvollen Testen von Apps auf emulierten Android-Geräten (siehe Kapitel 3).

Als die anschließende Beschreibung (im April 2018) erstellt wurde, waren bei Android die Version 8.1 und beim Android Studio die Version 3.1 aktuell. Es ist zu hoffen, dass die zum Zeitpunkt der Lektüre aktuellen Versionen nicht zu stark von den Beschreibungen abweichen.

### 2.1 Android Studio installieren

Weil es sich beim Android Studio um ein Java-Programm handelt, wird zur Ausführung auf dem Entwicklungsrechner eine Java Virtual Maschine (JVM) benötigt. Das Android Studio begnügt sich *nicht* mit einer Java Runtime Environment (JRE), sondern besteht auf einem Java Development Kit (JDK). Während sich eine JRE auf die Rolle einer Ausführungsumgebung für Java-Programme beschränkt, enthält ein JDK zusätzlich etliche Werkzeuge, den Quellcode der Java-Standardbibliothek und vor allem einen Compiler, der Java-Quellcode in Byte-Code übersetzen kann. Seit der Version 2.2 wird zusammen mit dem Android Studio das aktuelle OpenJDK installiert, sodass keine vorherige JDK-Installation vorausgesetzt wird.

Am 30.03.2018 wurde für eine Installation unter Windows die Datei **android-studio-ide-173.4670197-windows.exe** über die folgende Webseite zum Herunterladen angeboten:

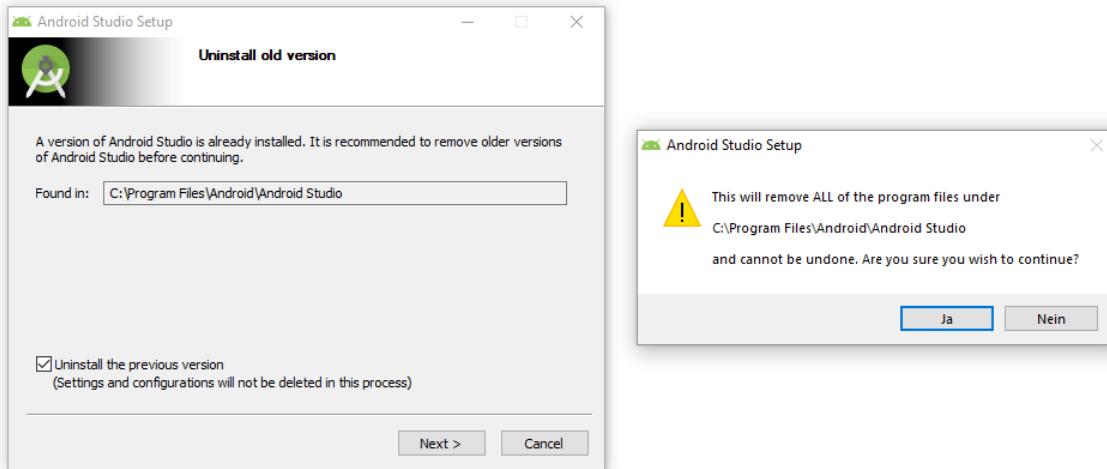
<https://developer.android.com/studio/index.html>

Starten Sie das herunter geladene Programm per Doppelklick, und quittieren Sie ggf. die Anfrage der Windows-Benutzerkontensteuerung mit **Ja**. Dem Vorschlag, eine (im Ordner **C:\Program Files\Android\Android Studio**) vorgefundene Altversion zunächst zu entfernen, sollten Sie zustimmen:

---

<sup>1</sup> Wer Eclipse kennt, findet auf der folgenden Webseite eine Migrationshilfe (z. B. mit einer Gegenüberstellung von Begriffen, Komponenten und Funktionen aus Eclipse und IntelliJ):

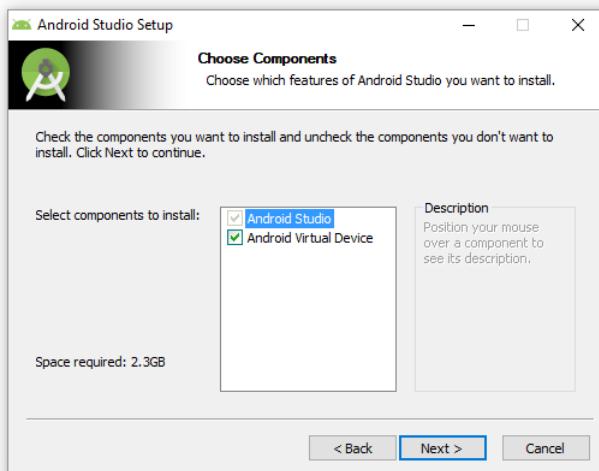
[https://www.jetbrains.com/idea/documentation/migration\\_faq.html](https://www.jetbrains.com/idea/documentation/migration_faq.html)



So startet die Installation der aktuellen Version:



Im nächsten Dialog werden die zu installierenden Komponenten erfragt:



Es stehen folgende Komponenten bereit, die in der Regel auch installiert werden sollten:

- **Android Studio**

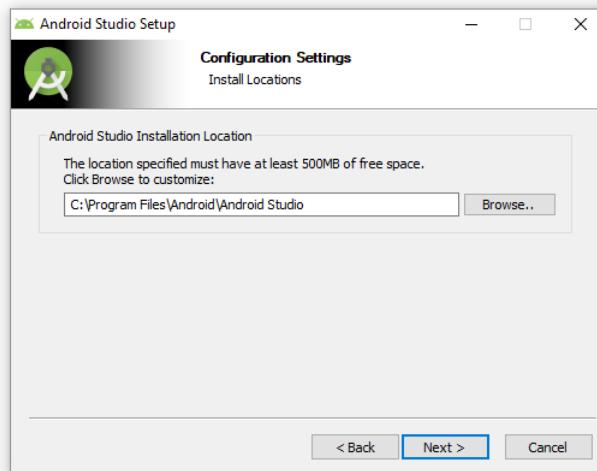
Im Bundle **android-studio-ide-173.4670197-windows.exe** ist die Version 3.1 (Build 173.4670197) enthalten:



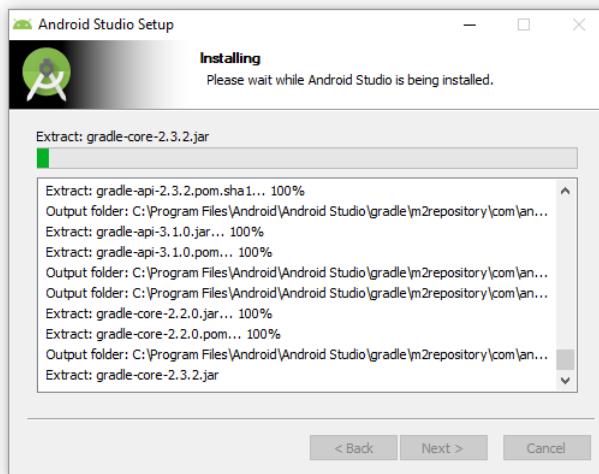
- **Android Virtual Device**

Das Bundle **android-studio-ide-173.4670197-windows.exe** installiert die zum Entwickeln erforderlichen Bibliotheken und Dokumente für **Android 8.1 (API-Level 27)** sowie ein **x86 System Image mit Android 8.1 und Google APIs**. Um ein virtuelles Android-Gerät basierend auf diesem System-Image nutzen zu können, muss der Entwicklungsrechner eine CPU mit Virtualisierungstechnik (VT) besitzen, die im BIOS aktiviert ist. Außerdem muss der *Intel(R) Hardware Accelerated Execution Manager* installiert werden, was in der Regel automatisch bei der Android SDK - Installation geschieht (siehe Abschnitt 2.2). Wenn kein x86-basiertes System-Image nutzbar ist, muss per SDK-Manger ein alternatives (ARM-basiertes) System-Image installiert werden (siehe Abschnitt 3.1). Basierend auf einem installierten System-Image können per AVD-Manager virtuelle Android-Geräte eingerichtet werden (siehe Abschnitt 3.2).

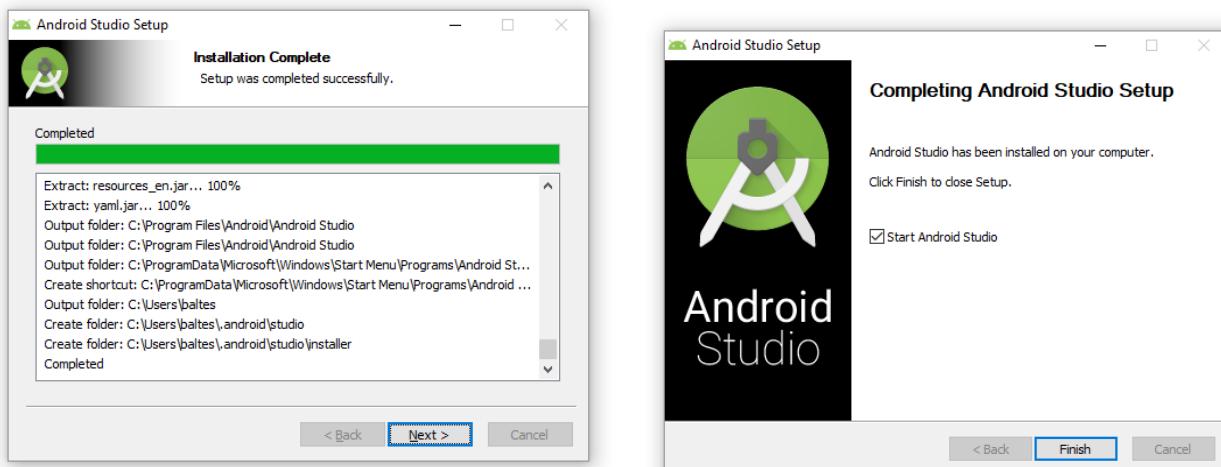
Für das Studio wird ein Computer-bezogener Installationsort vorschlagen:



Nach der Wahl bzw. Bestätigung des Startmenüordners werden die Komponenten installiert:



Schließlich ist das Android-Studio startbereit:



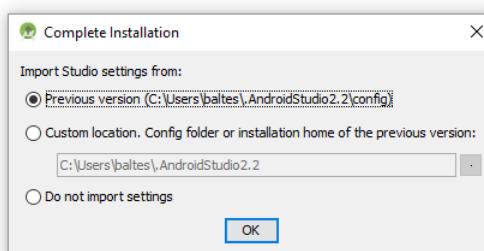
Mit dem Android Studio 3.1 wird am 30.03.2018 das OpenJDK mit der Version 8 (alias 1.8) Update 152 ausgeliefert. Es landet per Voreinstellung im Ordner:

**C:\Program Files\Android\Android Studio\jre**

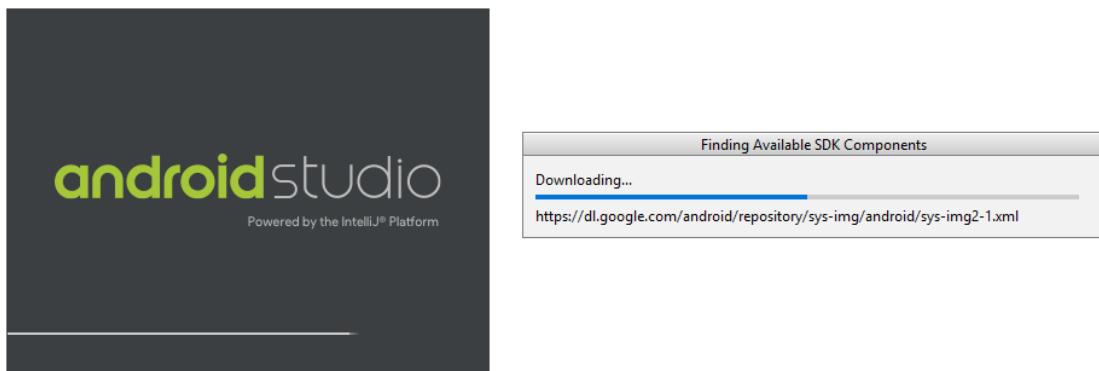
## 2.2 **Android-Studio zum ersten Mal starten und das Android SDK installieren**

Beim ersten Start des Android-Studios wird nötigenfalls das Android-SDK installiert. Dazu wird per Voreinstellung ein Ordner im Windows-Profil des angemeldeten Benutzers verwendet und das Installationsvolumen des Android Studios (ca. 1,3 GB) noch weit übertroffen.

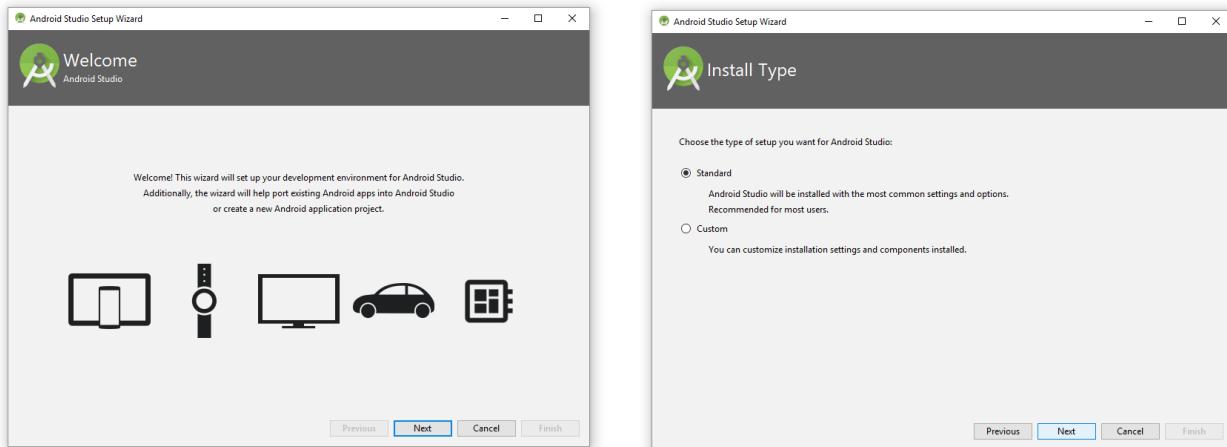
Zunächst besteht ggf. die Möglichkeit, Einstellungen von einer früheren Studio-Version zu übernehmen:



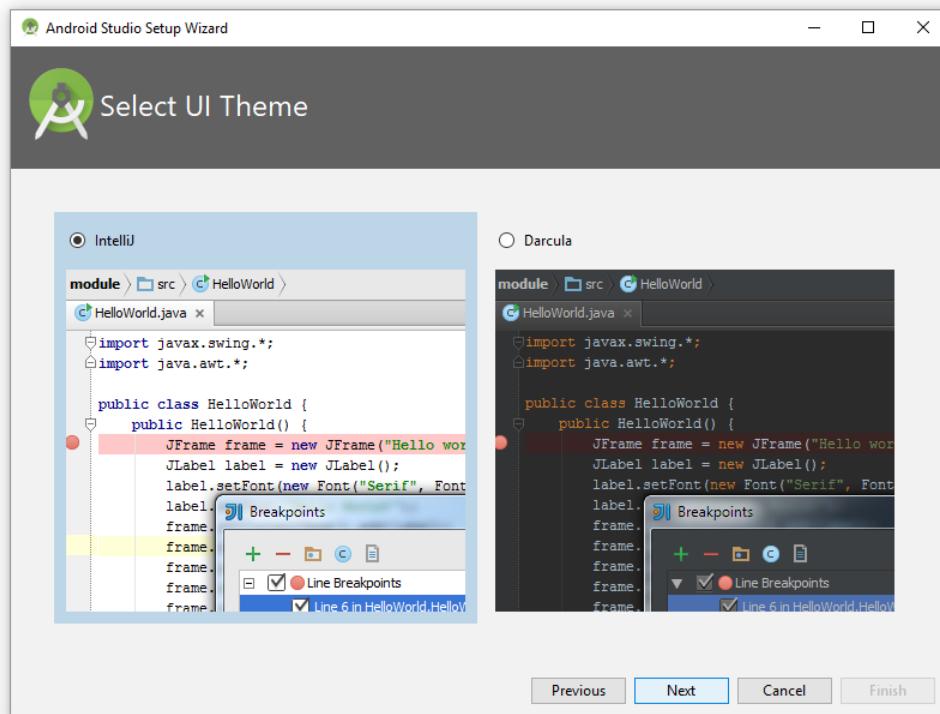
Anschließend werden Android SDK - Komponenten herunter geladen:



Schließlich startet der **Android Studio Setup Wizard**:



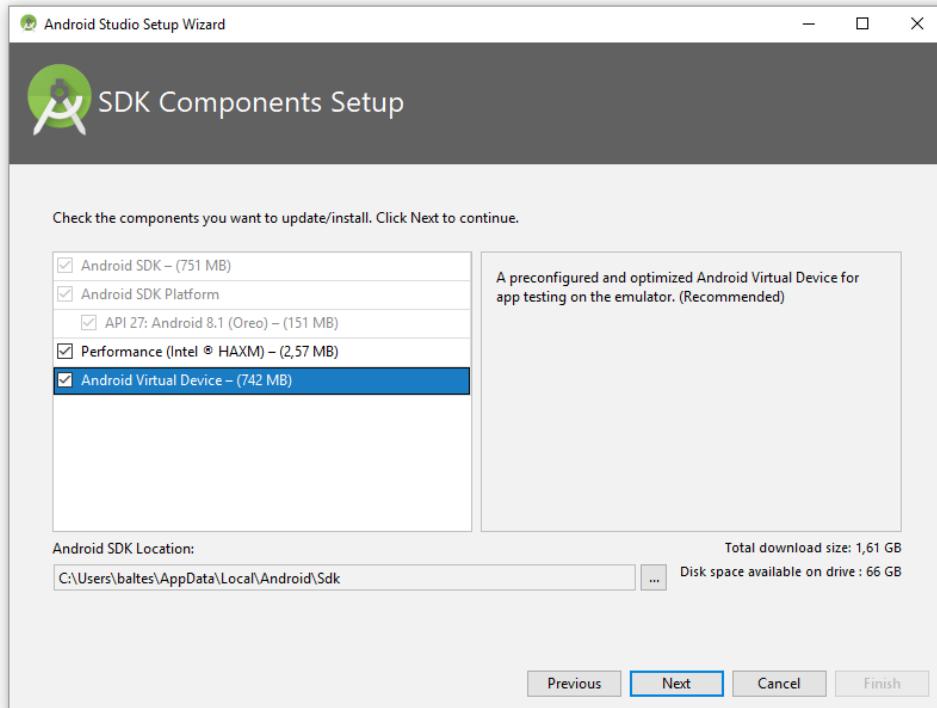
Nach der Entscheidung für eine **Standard**-Installation werden für die Bedienoberfläche des Android Studios zwei UI-Einstellungspakete (Themes) zur Wahl gestellt:



Wer seine Vorliebe für Dracula erst später entdeckt, kann im Android Studio nach **File > Settings > Appearance and Behavior > Appearance > Theme**

das Erscheinungsbild ändern.

Im nächsten Dialog

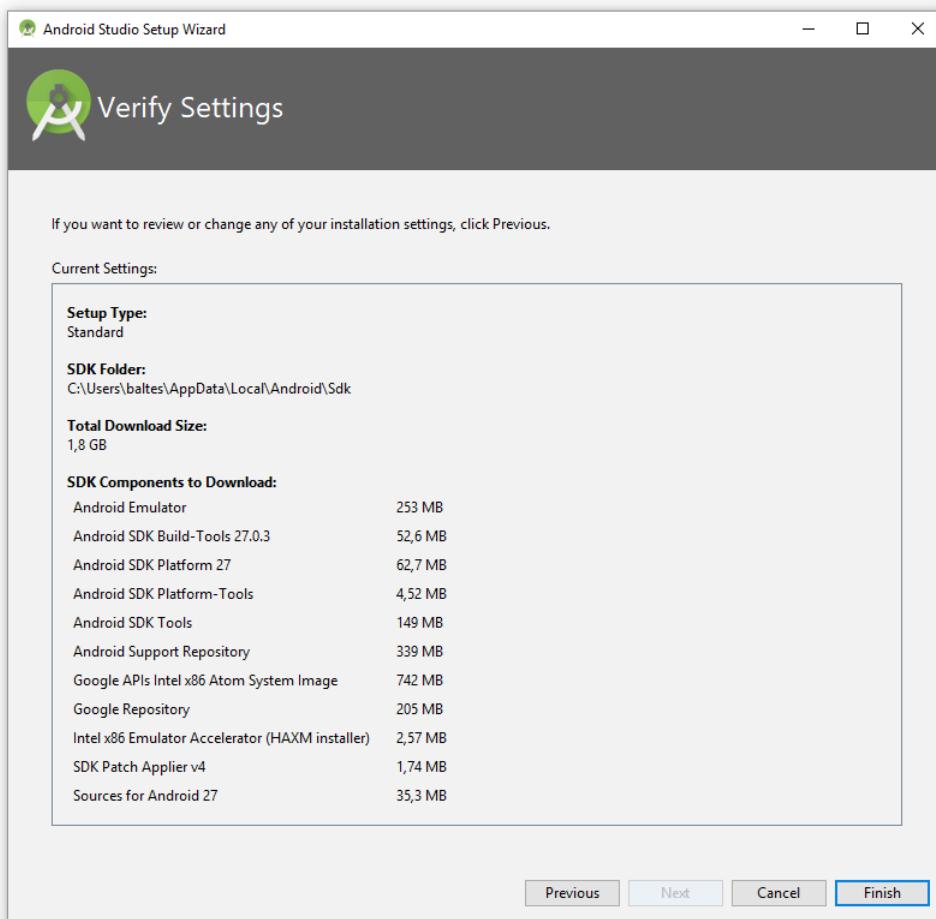


wird der *Intel(R) Hardware Accelerated Execution Manager (HAXM)* angeboten, wenn der Entwicklungsrechner eine Intel-CPU mit Virtualisierungstechnik VT besitzt. Seine Installation sorgt für eine beschleunigte Android-Emulation und ist empfehlenswert. Das Bundle **android-studio-ide-173.4670197-windows.exe** enthält die HAXM-Version 6.2.1.

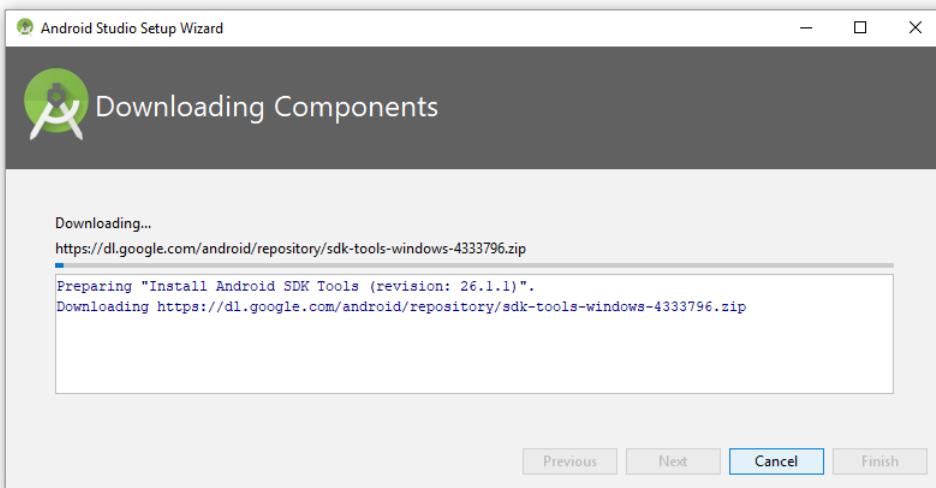
Eine weitere empfehlenswerte Option ist die Installation eines vorkonfigurierten virtuellen Android-Geräts, das die Ausführung von Apps im Emulator erlaubt.

Außerdem kann der (benutzerspezifische!) Installationsordner für das Android-SDK geändert werden. Das SDK in einen Ordner *ohne* Schreibrechte für den regelmäßigen Android Studio - Benutzer zu installieren, ist *nicht* empfehlenswert.

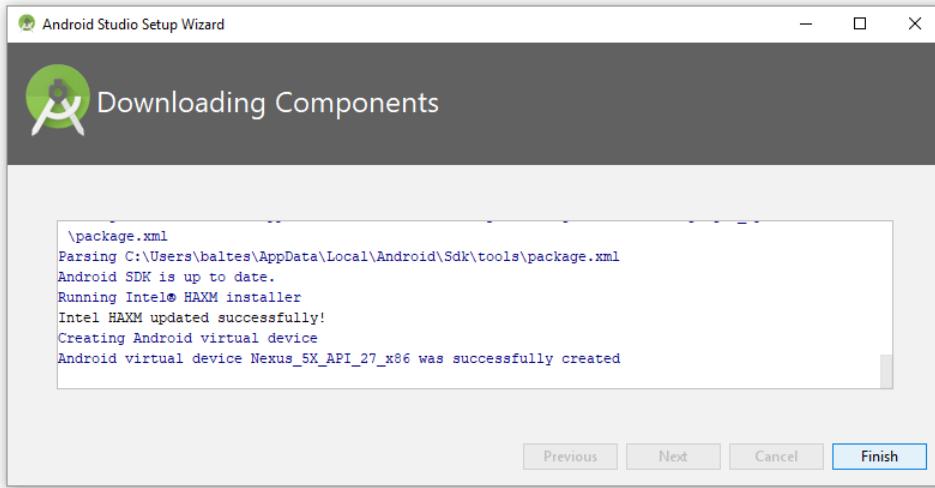
Im nächsten Dialog wird darüber informiert, welche Android-SDK - Komponenten nun heruntergeladen und installiert werden:



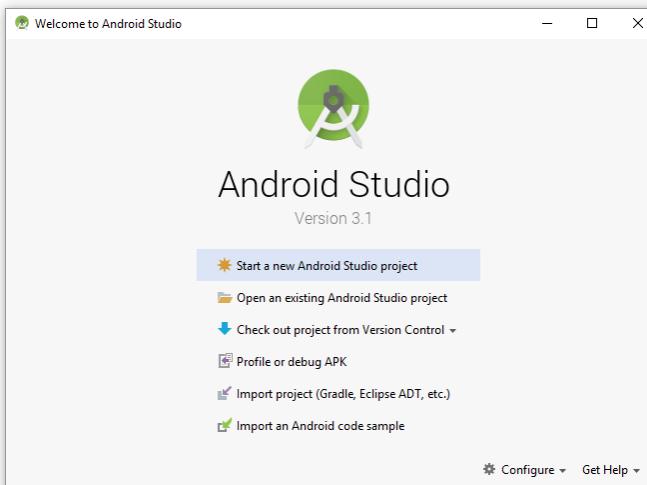
Bis das recht umfängliche Android SDK herunter geladen und installiert ist,



vergehen ca. 10 Minuten. Am Ende steht hoffentlich die erfreuliche Mitteilung, dass u.a. die Hardware-Beschleunigung der Android-Emulation sowie das virtuelle Gerät Nexus 5X (mit dem API-Level 27, für x86-Hardware) erfolgreich installiert worden sind:



Nach der SDK-Installation ist das Android Studio einsatzbereit:



Das Erstellen der ersten App wird in Abschnitt 2.3 beschrieben.

Nach der Installation des Android Studios samt Android SDK und der Erstellung einer ersten Android-App sind unter Windows die folgenden Ordner vorhanden:

Zweck	Voreingestellter Ordner unter Windows	Größe
Programm Android Studio 3.1	C:\Program Files\Android\Android Studio	ca. 1,3 GB
Konfiguration Android Studio 3.1	C:\Users<user>.AndroidStudio3.1	ca. 400 MB
Android SDK	C:\Users<user>\AppData\Local\Android\Sdk	ca. 5 GB
Android SDK - Konfiguration (inkl. virtuelle Geräte)	C:\Users<user>\.android	ca. 1,5 GB
Gradle-Konfiguration	C:\Users<user>\.gradle	ca. 1,5 GB

## 2.3 Die erste Android-App

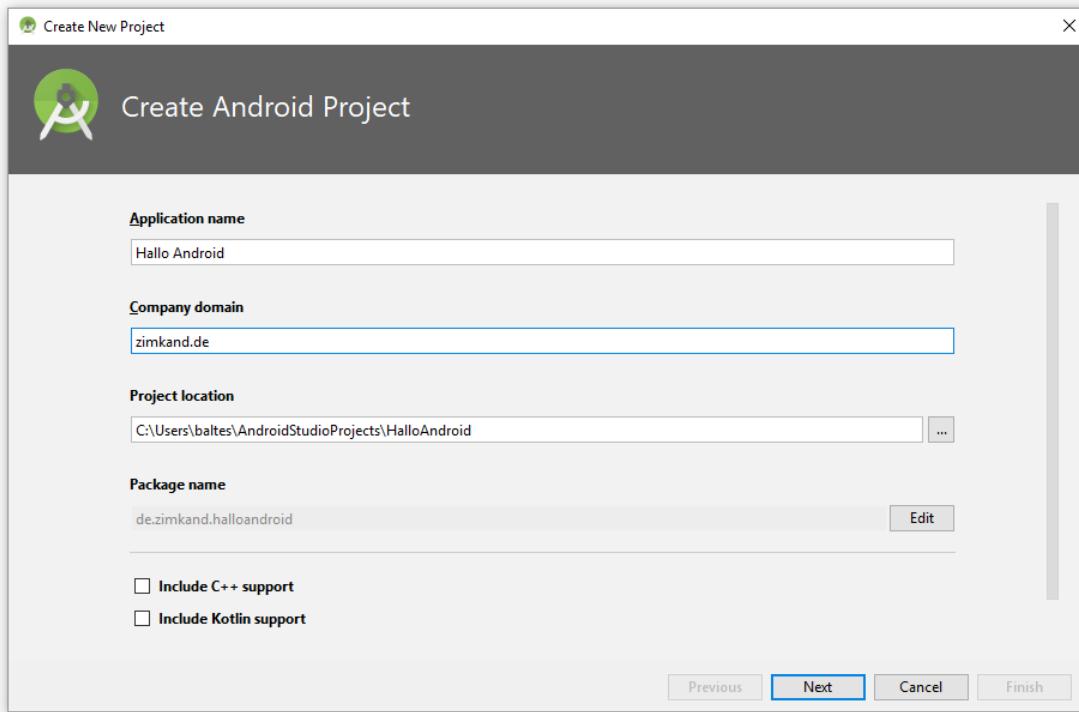
Um die Funktionstüchtigkeit der Entwicklungsumgebung (inkl. Emulation von Android-Geräten) zu testen, erstellen wir das unvermeidliche *Hello*-Projekt.

### 2.3.1 App erstellen

Wählen Sie im Willkommensdialog des Android-Studios das Angebot **Start a new Android Studio project**, oder wählen Sie im Studio-Fenster dem Menübefehl

**File > New Project**

Im ersten Assistentendialog



werden festgelegt:

- **Application name**

Diese Bezeichnung wird nach der Installation auf einem Android-Gerät u.a. vom App-Manager angezeigt und darf Leerzeichen enthalten.

- **Company Domain**

Java-Anwendungen (auch für Android) werden in Pakete verpackt, was u.a. für eine übersichtliche Ablage der Projektdateien in Ordner mit dem Paketnamen sowie für einen abgeschlossenen Namensraum ohne Kollisionsgefahr sorgt. Paketnamen sollen weltweit eindeutig sein. Sofern eine eigene Internet-Domäne vorhanden ist, stellt man dem Anwendungsnamen die Bestandteile des Domänennamens in umgekehrter Reihenfolge voran, um einen eindeutigen Paketnamen zu erzielen. Wir verwenden im Kurs die erfundene Domäne **zimkand.de**, sodass der Assistent im Beispiel als **Package name** vorschlägt: `de.zimkand.halloandroid`. Nach einem Klick auf **Edit** kann der vorgeschlagene Paketname geändert werden.

- **Project location**

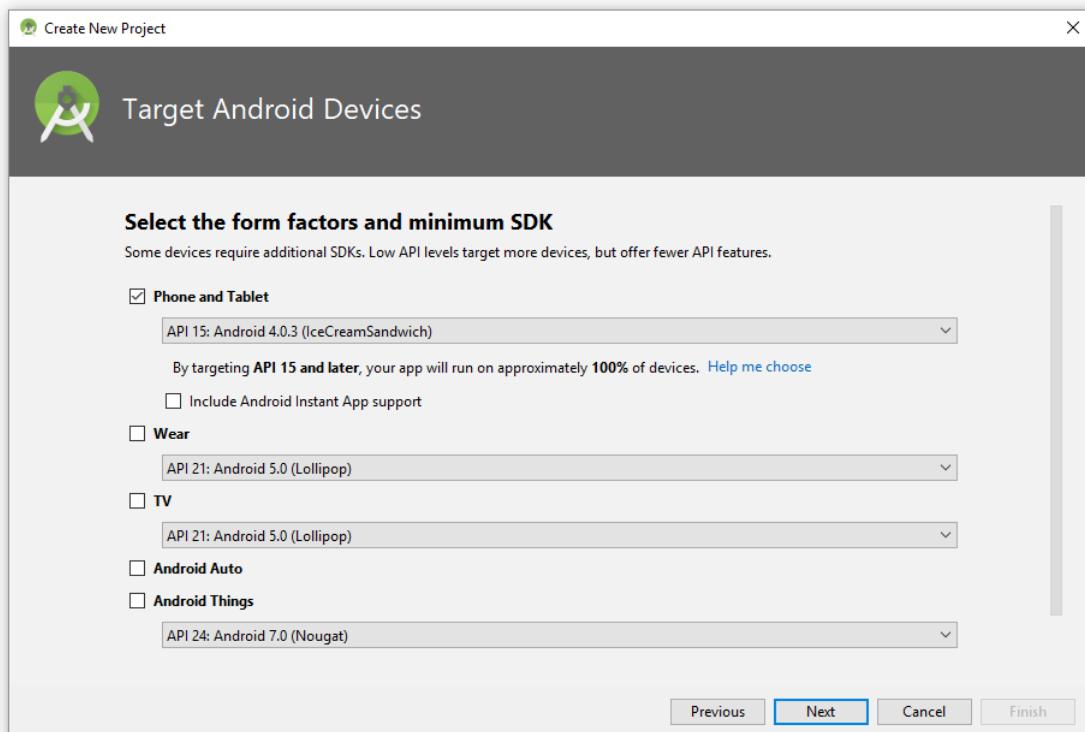
Das Android-Studio legt Projekte unter Windows per Voreinstellung im Ordner **AndroidStudioProjects**

ab, der sich im Stammordner des Benutzerprofils befindet, auf einem PC unter Windows (7, 8.x oder 10) für einen Benutzer namens **theo** z. B. in:

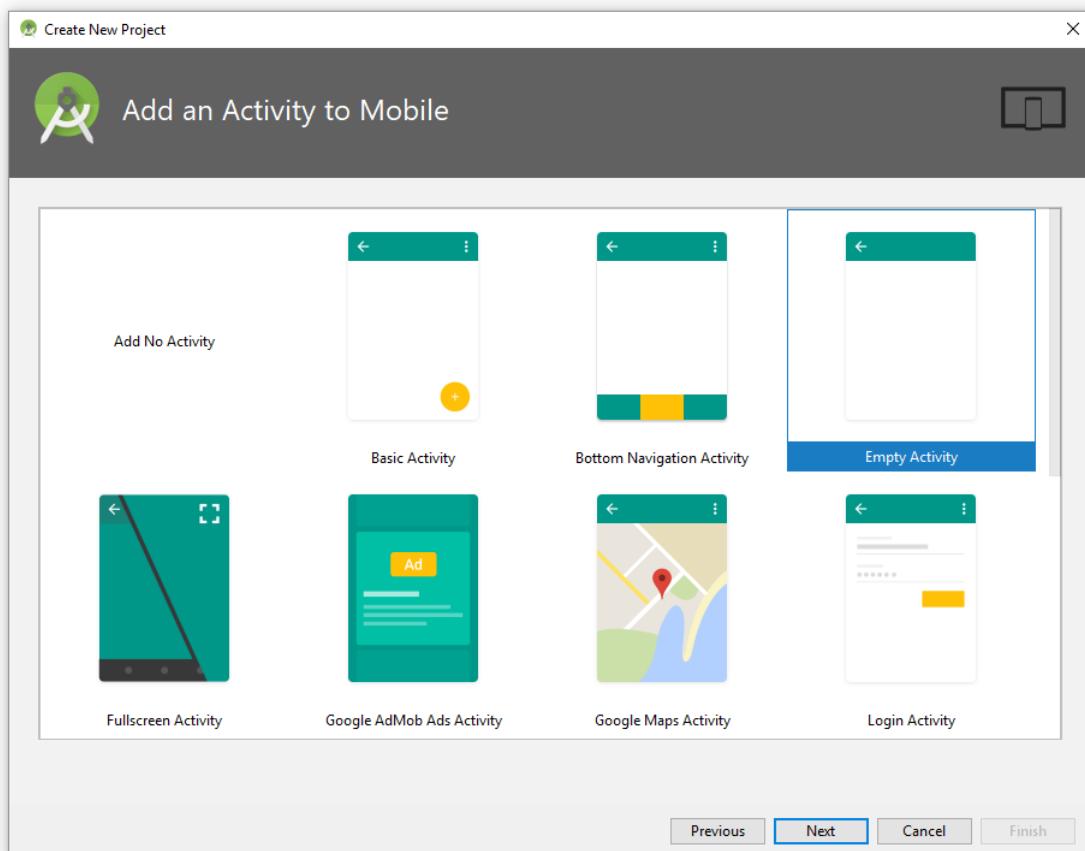
**C:\Users\theo**

Den Projektnamen leitet das Studio aus dem Anwendungsnamen unter Beachtung syntaktischer Restriktionen (z. B. Vermeidung von Leerzeichen) ab.

Im zweiten Assistentendialog geht es darum, auf welchen Plattformen die App laufen soll. Wir akzeptieren den sinnvollen Vorschlag, dass unser Programm auf Smartphones und Tablets mit der minimalen Android-Version 4.0.3 verwendbar sein soll, sodass wir ein halbwegs aktuelles Android-System nutzen und trotzdem praktisch alle aktuell im Einsatz befindlichen Smartphones und Tablets mit Android unterstützen können:

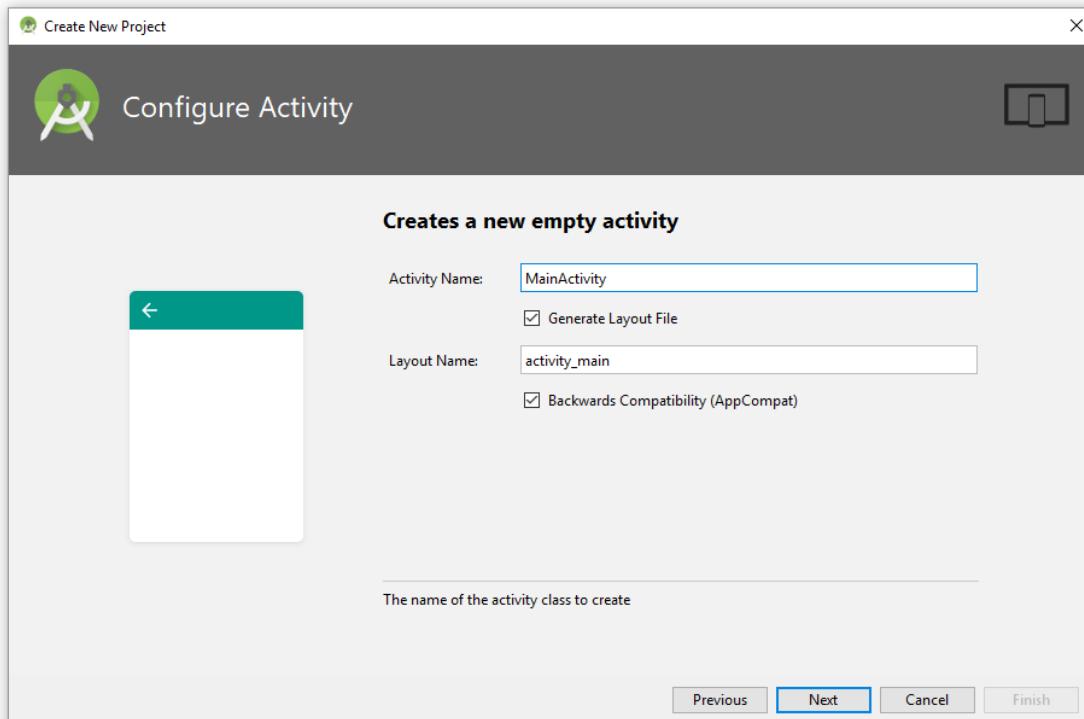


Im dritten Assistentendialog ist die Voreinstellung **Empty Activity** angemessen:

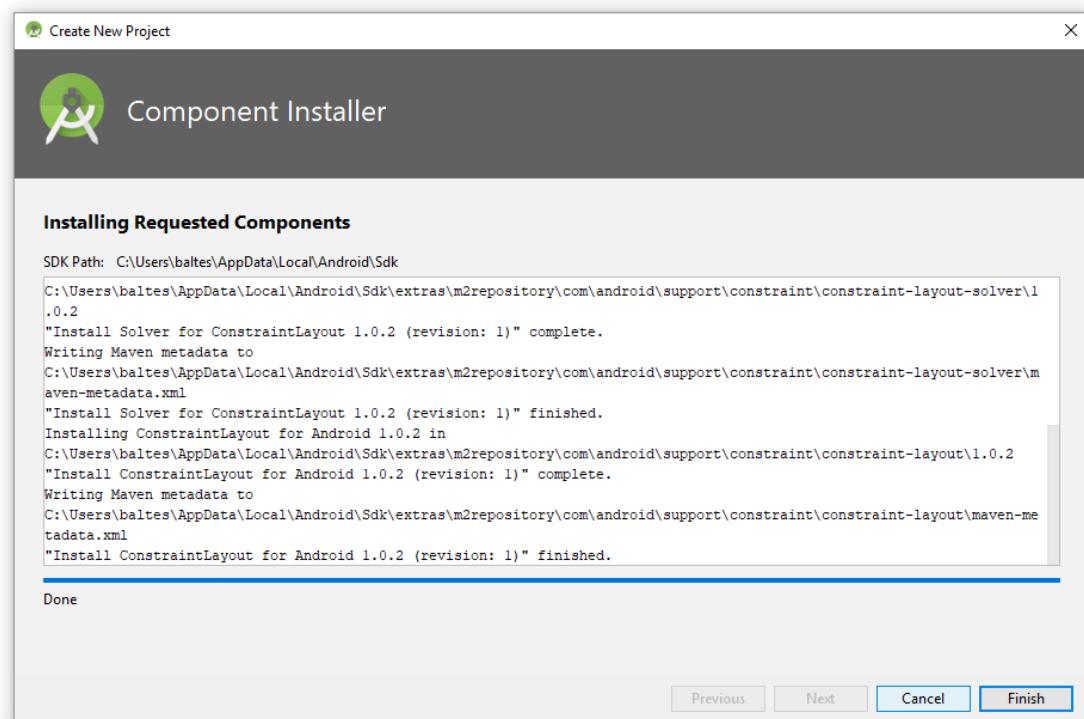


Wir werden uns in Kapitel 5 mit den Activities des Android-Betriebssystems ausführlich beschäftigen.

Im vierten Assistentendialog erhält die Java-Klasse, welche für die Startaktivität bzw. für den Startbildschirm der App zuständig ist, ebenso einen Namen (**Activity name**) wie die XML-Datei, die das Layout der Startaktivität definiert (**Layout name**):



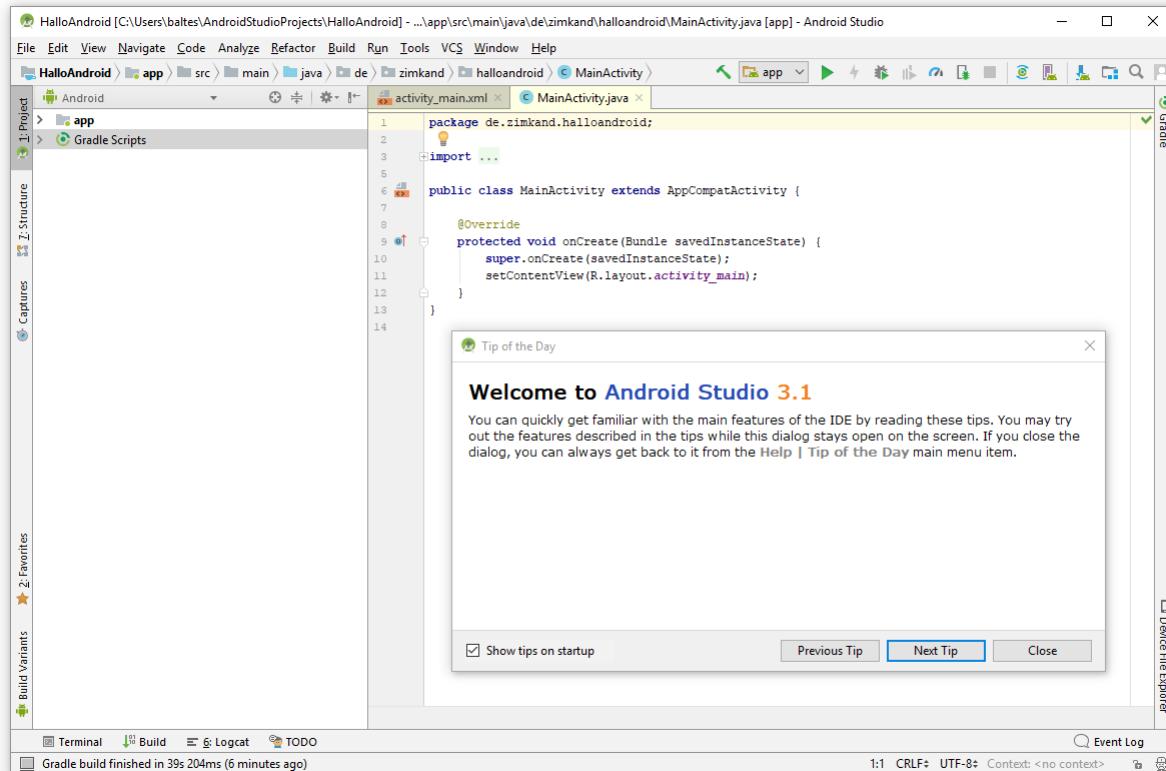
Obwohl wir keine Sonderwünsche geäußert haben, benötigt die Android SDK - Installation doch noch einige Ergänzungen:



Mit einem Klick auf den Schalter **Finish** veranlassen wir die Erstellung des neuen Projekts, und nach einigen Minuten mit intensiver Tätigkeit des Gradle - Build-Systems

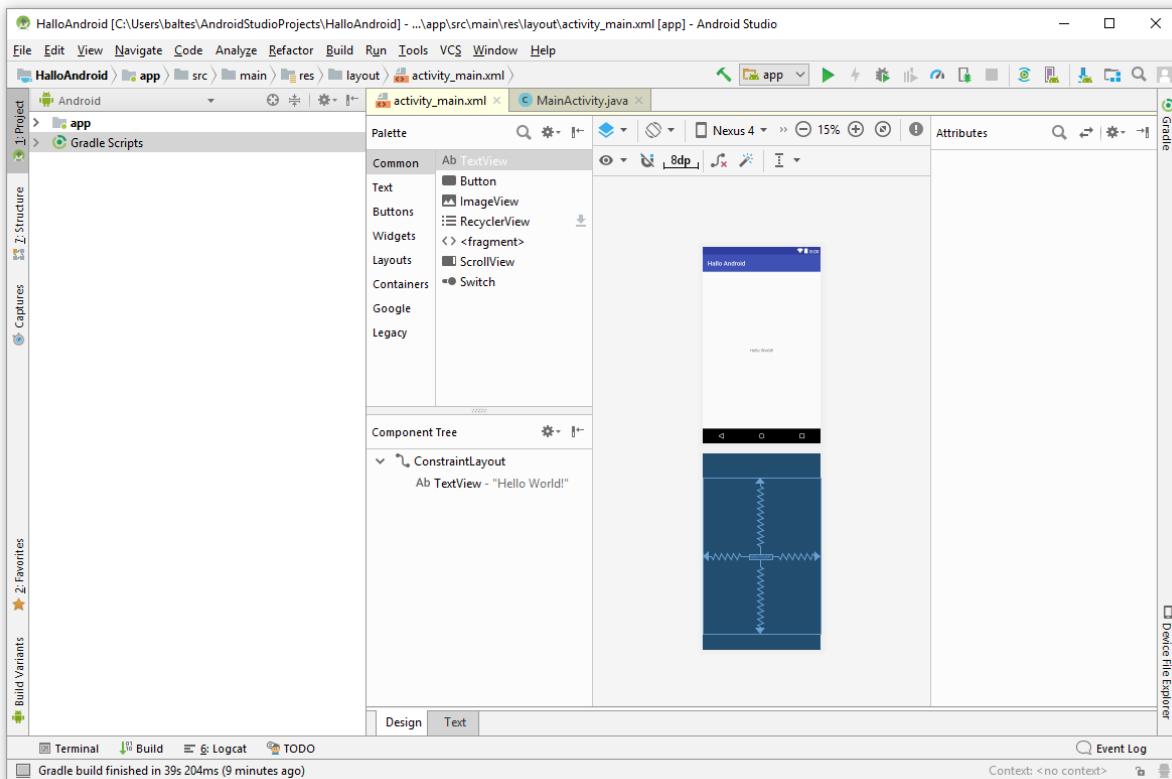


zeigt das Android Studio den Quellcode der Startaktivität:<sup>1</sup>



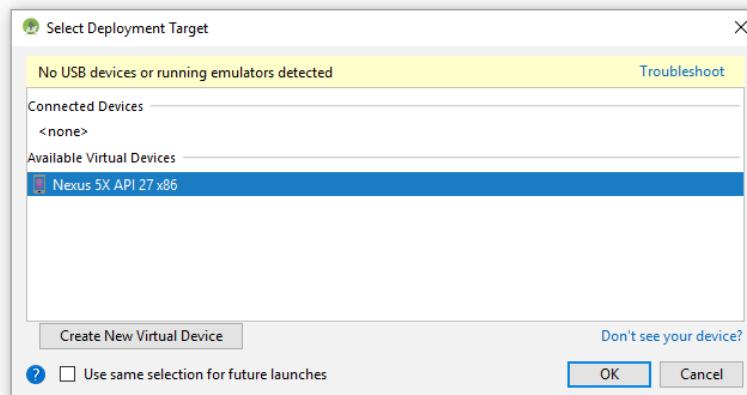
Nach einem Klick auf die Registerkarte mit der XML-Datei, die das Layout der Startaktivität definiert (im Beispiel **activity\_main.xml**), und einer weiteren Wartezeit von ca. 10 Sekunden erscheint in der **Design**-Zone eine Vorschau auf die App:

<sup>1</sup> Auf meinem Rechner mit Intel-CPU Core i3 550 dauerte es ca. 5 Minuten.



### 2.3.2 App im Emulator ausführen

Nach dem Versuch, per Mausklick auf den Schalter mit dem Menübefehl **Run > Run 'app'** oder mit der Tastenkombination + **F10** die Hallo World - App zu starten, erfahren wir, dass keine realen Android-Geräte per USB-Kabel ansprechbar und, aber ein virtuelles Gerät definiert ist:



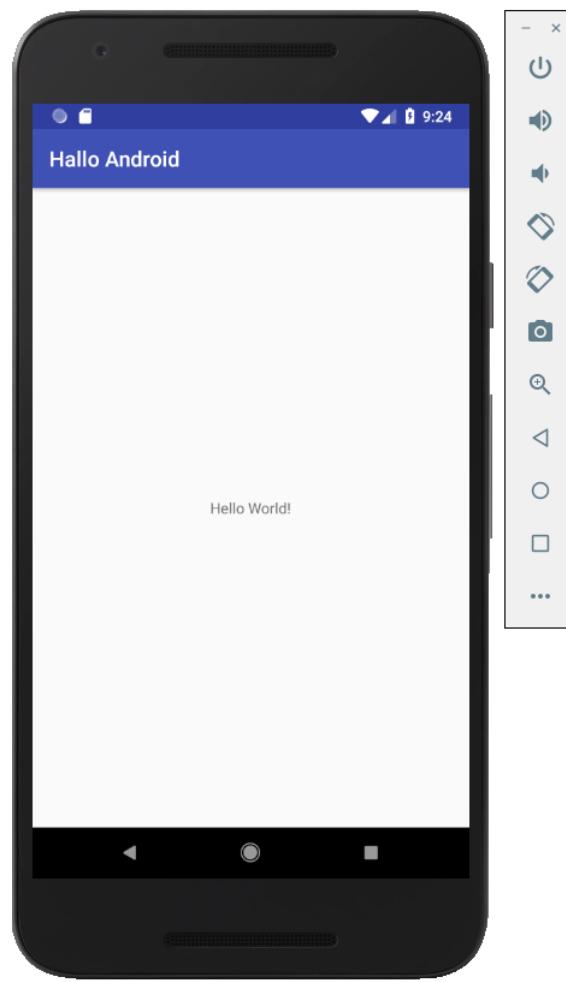
Nach **OK** wird das Android-Betriebssystem auf dem virtuellen Gerät gebootet, dann die **Hello** - App installiert und gestartet:<sup>1</sup>

---

<sup>1</sup> Auf meinem relativ mager ausgestatteten Rechner (4 GB RAM) endete der erste Startversuch nach ca. 2 Minuten mit einem Fehler:

Emulator: Process finished with exit code  
-1073741819 (0xC0000005)

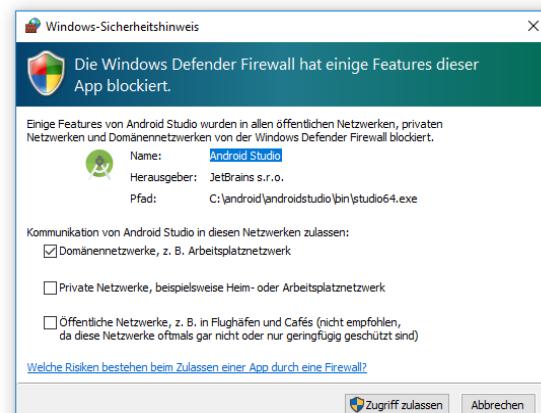
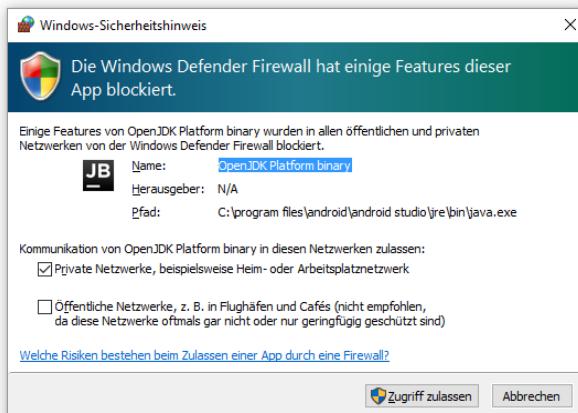
Der zweite Versuch verlief zügig und erfolgreich.



An das Fenster des Emulators ist eine Palette zur Bedienung des virtuellen Geräts angeheftet (z. B. Orientierungswchsel, Anfertigung eines Bildschirmfotos). Es ist sogar möglich, die Größe des Emulator-Fensters durch das Verschieben eines Fensterrandes zu verändern. Detaillierte Hinweise zur Bedienung des Emulators sind hier zu finden:

<https://developer.android.com/studio/run/emulator.html>

Beim Einsatz des Emulators erscheinen eventuell Anfragen der Windows-Firewall bezogen auf den im Android-Studio enthaltenen Java-Interpreter und/oder bezogen auf das Studio selbst:



Den Zugriff mit **Abbrechen** zu verweigern (alternativlos für Benutzer mit Normalrechten) hat sich bisher *nicht* als nachteilig erwiesen. Wer den **Zugriff zulässt**, geht wohl bei einer Beschränkung auf Domänennetzwerke oder private Netzwerke kein großes Risiko ein und kann seine Ent-

scheidung später über das Systemsteuerungs-Applet der Windows-Firewall revidieren (**Erweiterte Einstellungen > Eingehende Regeln**).

## 2.4 Wichtige Pfade für Einstellungen und Projekte

Das **Android SDK** legt den Einstellungsordner **.android** unterhalb eines Basisordners ab, der durch die Umgebungsvariable **ANDROID\_SDK\_HOME** festgelegt werden kann. Fehlt diese Umgebungsvariable, wird unter Windows (7, 8.x, 10) der Stammordner des Benutzerprofils als Basis verwendet, und bei einem Benutzer namens **theo** entsteht:

**C:\Users\theo\.android**

Neben Cache- und Konfigurationsdateien landen hier auch die gemäß Abschnitt 3.2 erstellten virtuellen Android-Geräte:

**C:\Users\theo\.android\avd**

Das **Android Studio** legt Konfigurations- und Projektdateien im Stammordner des Benutzerprofils ab, unter Windows (7, 8.x, 10) für den Benutzer **theo** also unter:

**C:\Users\theo\**

Hier werden folgende Unterordner angelegt:

- Die Benutzer-bezogenen Einstellungen zum Android Studio 3.1 landen in:

**C:\Users\theo\AndroidStudio3.1**

- Die Projekte landen in:

**C:\Users\theo\AndroidStudioProjects**

- Auch das vom Android Studio benutzte Erstellungsprogramm **Gradle** verwendet einen eigenen Konfigurationsordner:

**C:\Users\theo\.gradle**

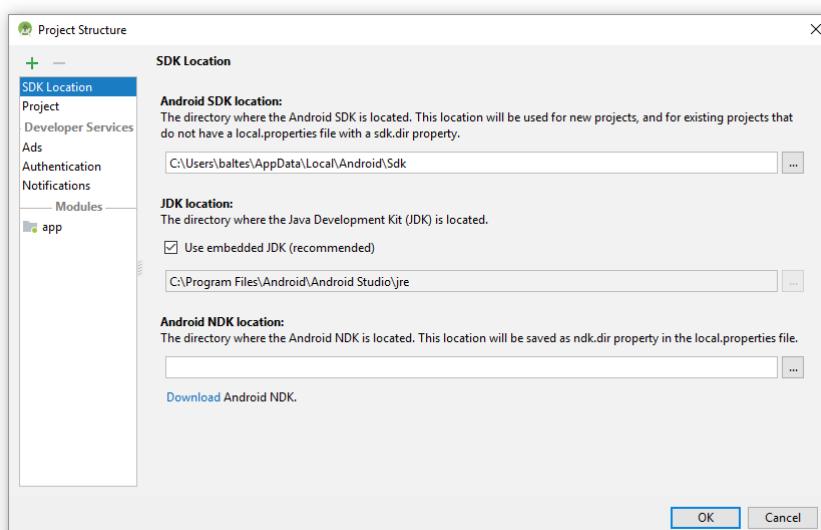
Das Android SDK wird für den Windows-Benutzer **theo** per Voreinstellung hier installiert

**C:\Users\theo\AppData\Local\Android\Sdk**

Dass im Android Studio dieser Ort korrekt eingestellt ist, lässt sich nach dem Menübefehl

**File > Project Structure**

überprüfen:



Im selben Dialog ist auch die **JDK Location** zu finden, wobei es kaum einen Grund gibt, das voreingestellte **Embedded JDK** zu ersetzen.

### **2.5 Übungsaufgaben zu Kapitel 2**

- 1) Führen Sie nach Möglichkeit die in Kapitel 2 beschriebenen Installationen auf Ihrem eigenen Rechner aus.

### 3 Apps auf virtueller und realer Hardware testen

Normalerweise sollte die in Kapitel 2 beschriebene Installation gut über die Bühne gehen und anschließend der Start der ersten App im automatisch installieren virtuellen Android-Gerät gelingen. Die Testumgebung für Android-Software ist allerdings aufwändig und variantenreich, sodass sich eine nähere Beschäftigung damit lohnt.

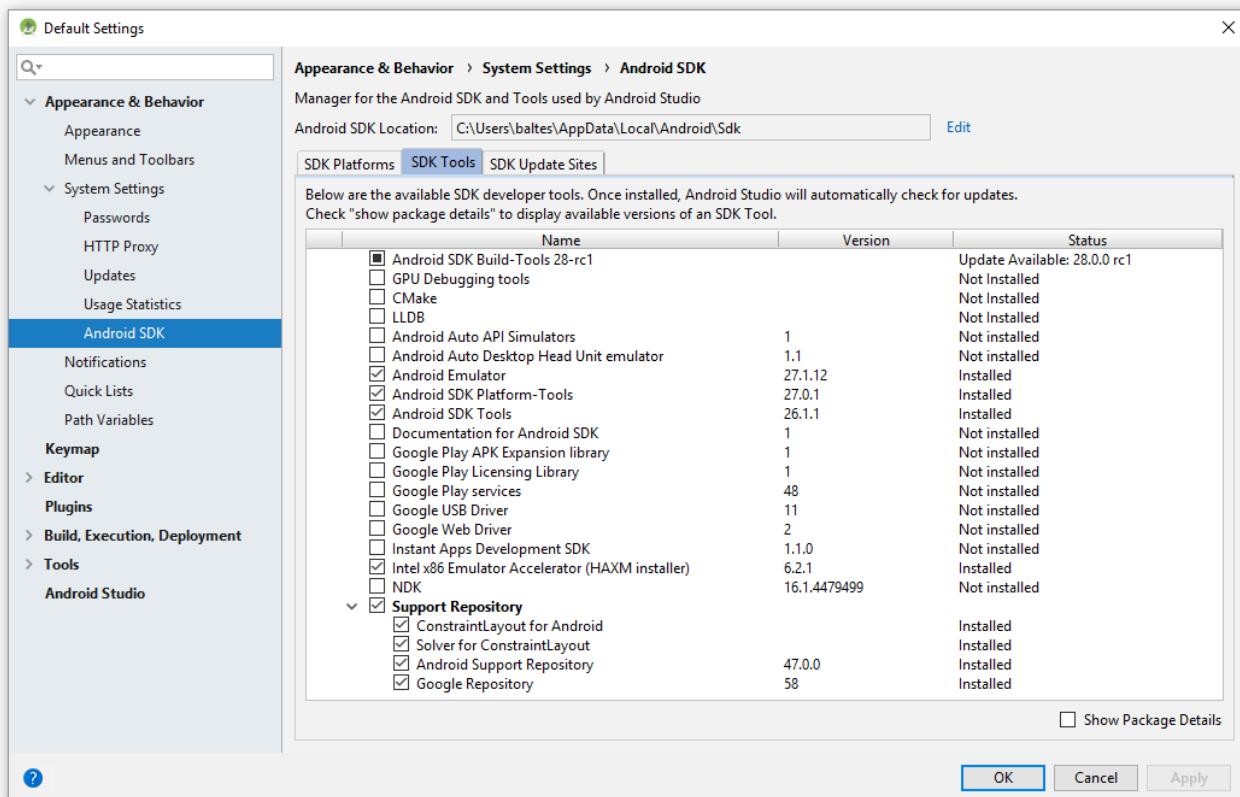
#### 3.1 SDK-Manager

Mit dem SDK-Manager kann man die Android SDK - Installation überprüfen und modifizieren (z. B. ergänzen oder aktualisieren). Insbesondere lassen sich per SDK-Manager zusätzliche Android-Versionen installieren, die auf virtuellen oder realen Android-Geräten zum Testen von Apps verwendet werden sollen. Der SDK-Manager lässt sich aus dem Anwendungsfenster des Android Studios starten ...

- über den Menübefehl **Tools > SDK Manager**
- oder über das Symbol  in der Symbolleiste unter der Menüleiste

##### 3.1.1 SDK-Tools aktualisieren bzw. ergänzen

Nach den in den Abschnitten 2.1 und 2.1 beschriebenen Installationen von Android Studio und Android SDK sind auf dem Registerblatt **SDK Tools** des SDK Managers folgende Komponenten vorhanden:



Es ist sinnvoll, die **Android SDK Build Tools** in der Hoffnung auf einen performanteren und störungsfreien Erstellungsprozess auf die Version 28.0.0 rc1 zu aktualisieren.

Um aus dem Android Studio auf ein per USB-Kabel angeschlossenes Android-Smartphone zugreifen zu können, wird ein USB-Treiber benötigt (siehe Abschnitt 3.5). Der angebotenen **Google USB-Treiber** ist allerdings nur für Smartphones der Firma Google gedacht. Einige andere Hersteller stellen für ihre Geräte ebenfalls einen USB-Treiber zur Verfügung (siehe Abschnitt 3.5).

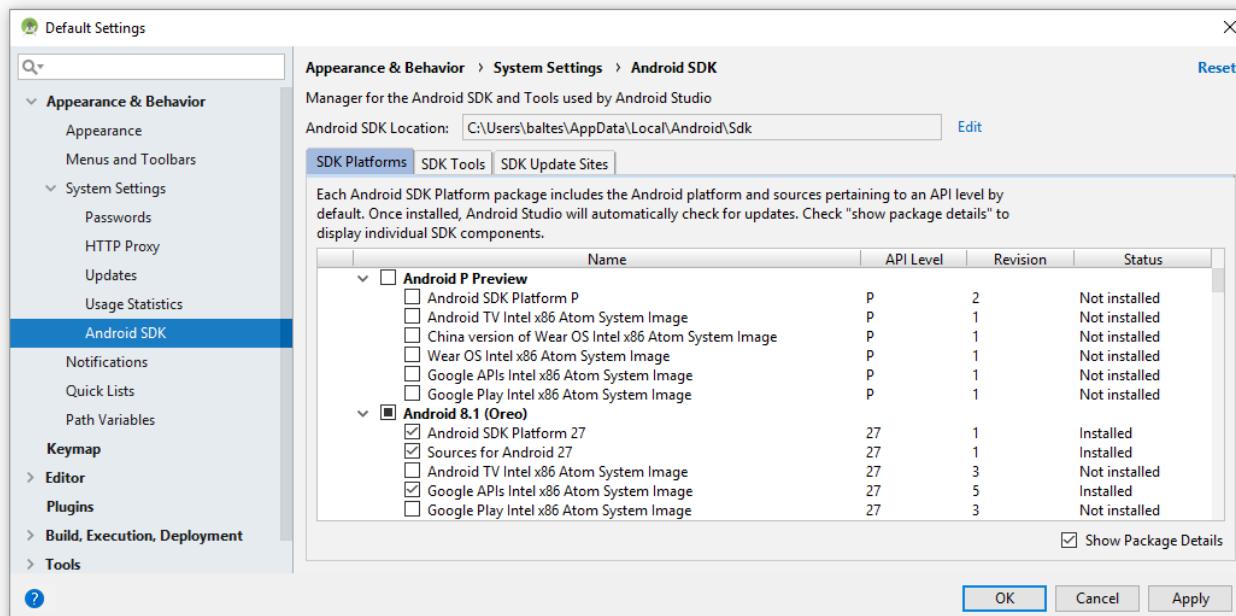
Wählt man im SDK-Manager den **Intel x86 Emulator Accelerator (HAXM installer)**, landet das Installationsprogramm **intelhaxm-android.exe** für diesen Emulator im folgenden Unterordner zum Android SDK:

**...\\sdk\\extras\\intel\\Hardware\_Accelerated\_Execution\_Manager**

Nach **Apply** werden die markierten Ergänzungsinstallationen ausgeführt.

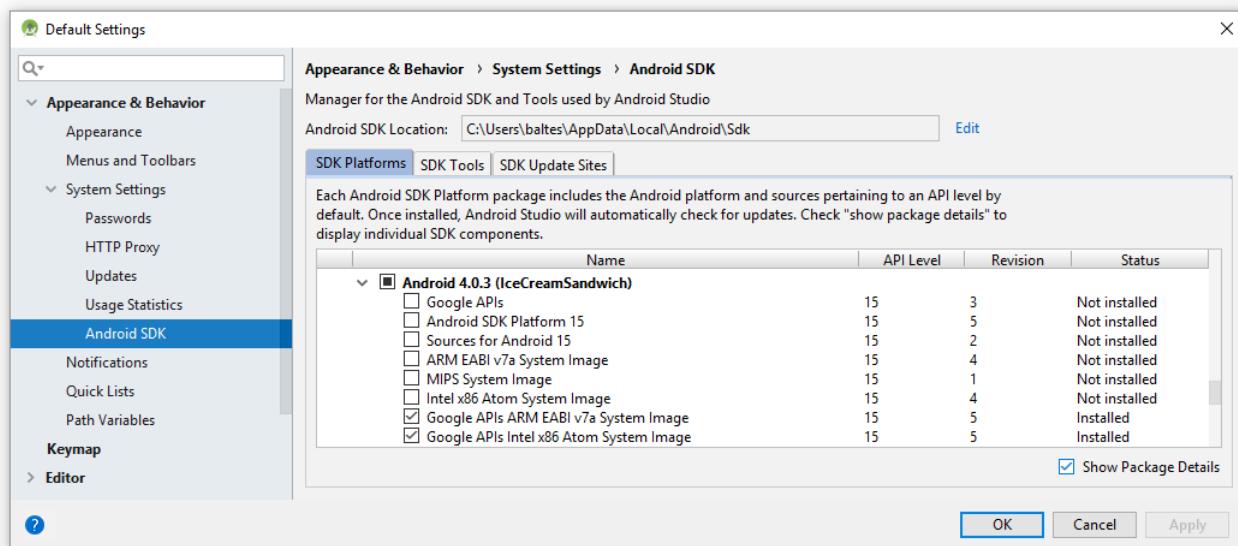
### 3.1.2 SDK-Plattformen aktualisieren bzw. ergänzen

Auf der Registerkarte **SDK Platforms** des SDK Managers sollten die **Package Details** eingeschaltet werden:

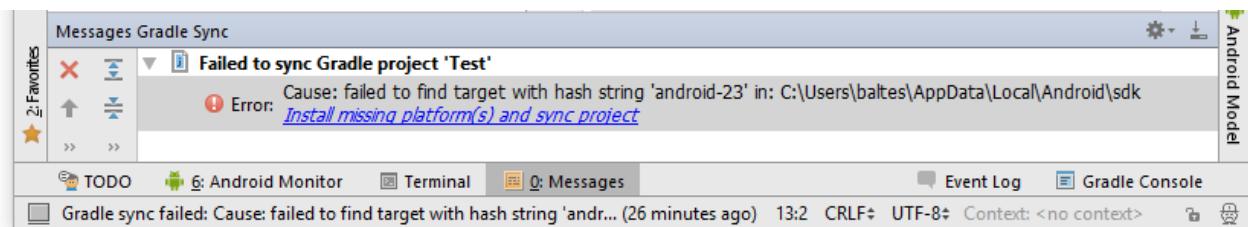


Wenn auf einem Entwicklungsrechner der *Intel(R) Hardware Accelerated Execution Manager (HAXM)* nicht installiert werden kann (vgl. Abschnitt 3.3), sodass kein lauffähiges System-Image als Basis für virtuelle Android-Geräte vorhanden ist, muss ein Image mit der (bei realen Android-Geräten dominierenden) Technik der britischen Prozessorschmiede ARM (*Advanced RISC Machines*) installiert werden (mit oder ohne Google-APIs bzw. Google Play - Zugang). Es wird ohne Hardware-Beschleunigung (also weniger performant) ausgeführt, benötigte aber keine spezielle Prozessortechologie. Bei der Installation von System-Images ist zu bedenken, dass ein Image je nach API-Level (Android-Version) mehr als 1 GB Platz im SDK-Ordner belegen kann.

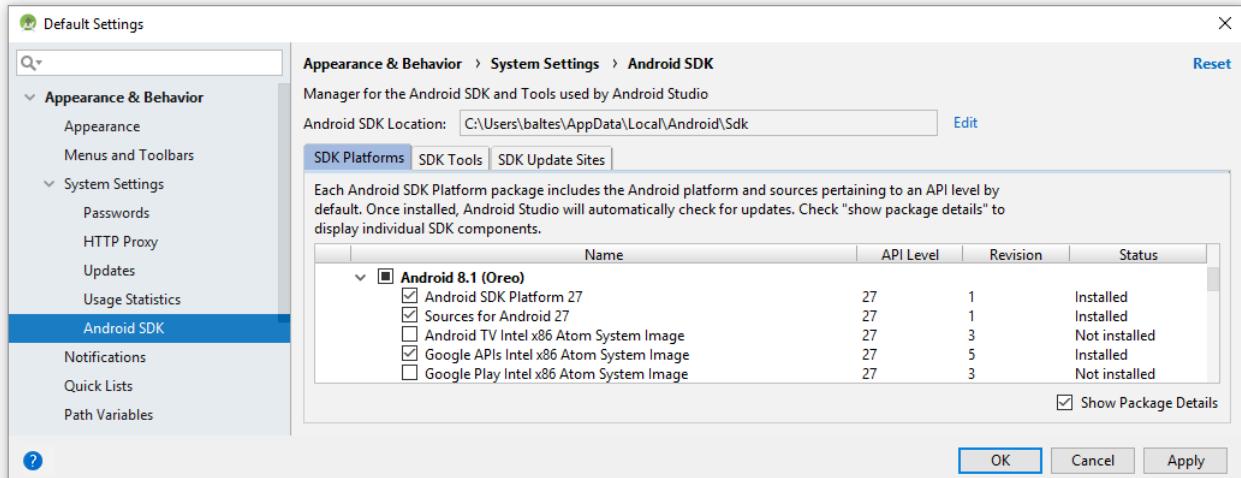
Für die meisten Übungsprogramme werden wir nur Android 4.0.3 (API-Level 15) benötigen. Weil dieses System relativ wenig Hauptspeicher belegt sowie relativ schnell startet, bietet es sich an, System-Images (in Intel- und eventuell auch in ARM-Technik) mit Android 4.0.3 zu installieren. Daher installieren wir die System-Images zum API-Level 15:



Bei einer Android Studio/Android SDK - Installation wird per Voreinstellung wird nur die SDK-Plattform zur jüngsten Android-Version installiert. Wenn Sie nach dieser Regel eine SDK-Version > 8.1 (Oreo, API-Level 27) installiert haben, sehen Sie beim Öffnen der Manuscript-Projekte eine Fehlermeldung nach diesem Muster:



In diesem Fall sollten Sie der Einfachheit halber die SDK-Plattform 27 zusätzlich installieren, z. B.:



Weitere Hinweise zur Verwendung der Kursprojekte folgen in Abschnitt 4.10.

Nach **Apply** werden die markierten Ergänzungsinstallationen ausgeführt.

### 3.2 AVD-Manager

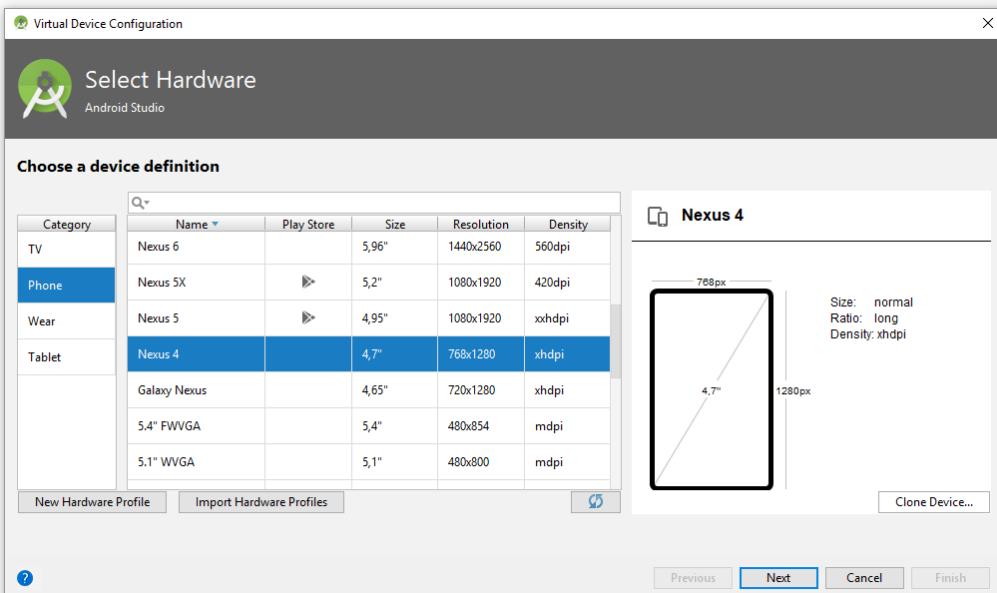
Es ist außerordentlich nützlich bis unverzichtbar, eine App in emulierten (virtuellen) Android-Geräten mit unterschiedlichen Hardware-Merkmalen und Android-Versionen testen zu können. Solche virtuellen Geräte werden mit dem **Android Virtual Device Manager** eingerichtet und können dann im Android Studio als Zielplattform für die Ausführung einer App gewählt werden. Ein virtuelles Gerät kann fast alle Funktionen eines realen Geräts nachahmen.

Man startet den **Android Virtual Device Manager** aus dem Anwendungsfenster des Android Studios ...

- über den Menübefehl **Tools > AVD Manager**,
- über das Symbol  in der Symbolleiste unter der Menüleiste,
- über den Schalter **Create New Virtual Device** im Dialog **Select Deployment Target**, der nach dem Starten einer App (z. B. per Schalter  auftaucht.

#### 3.2.1 Virtuelle Android-Geräte definieren und konfigurieren

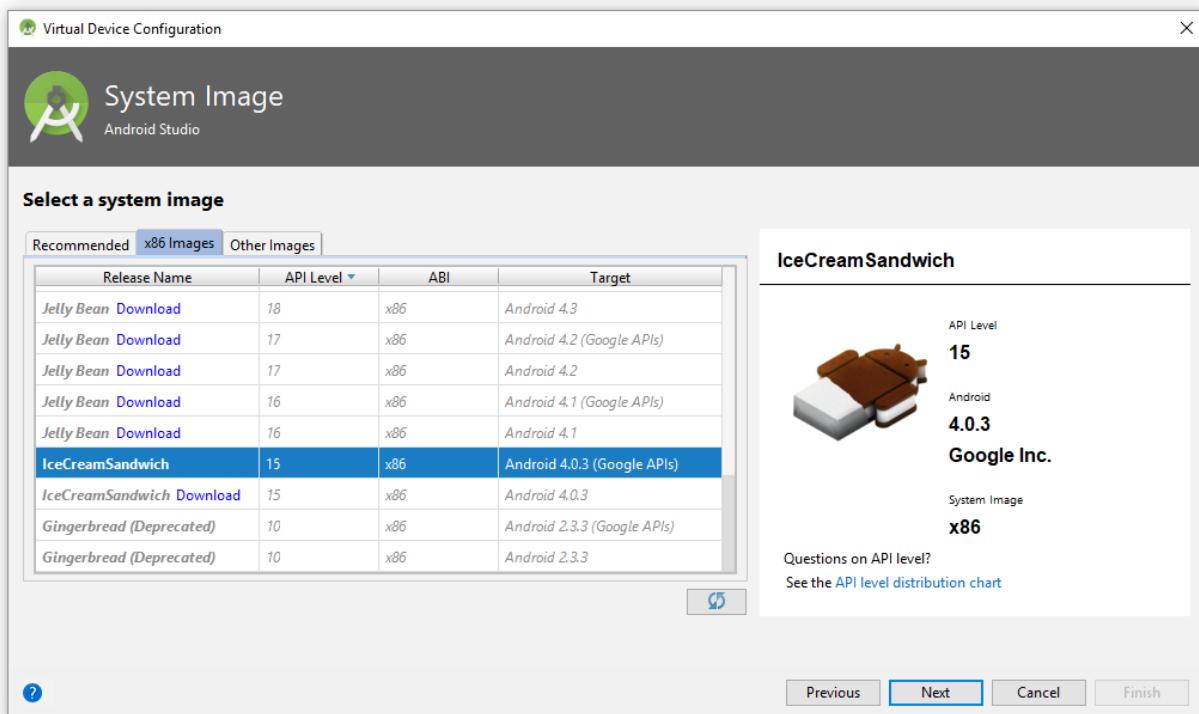
Wir beginnen über den Schalter **Create Virtual Device** damit, ein neues virtuelles Gerät zu erstellen und verwenden dabei eine als **Device Definition** eine relativ bescheidene Hardware:



Im Unterschied zur Gerätedefinition **Nexus 5X**, die beim automatisch installierten AVD-Gerät verwendet wird, ist die Gerätedefinition **Nexus 4** nicht kompatibel mit dem **Play Store**. Hier kann kein Systemabbild installiert werden, das die Play Store - App enthält.

Wenn die GPU (*Graphics Processing Unit*) des Entwicklungsrechners nicht zur Beschleunigung der Grafikausgabe herangezogen werden kann (Hardware-Rendering, siehe unten), dann sollte die Auflösung des emulierten Gerätes nicht über  $768 \times 1280$  hinausgehen.

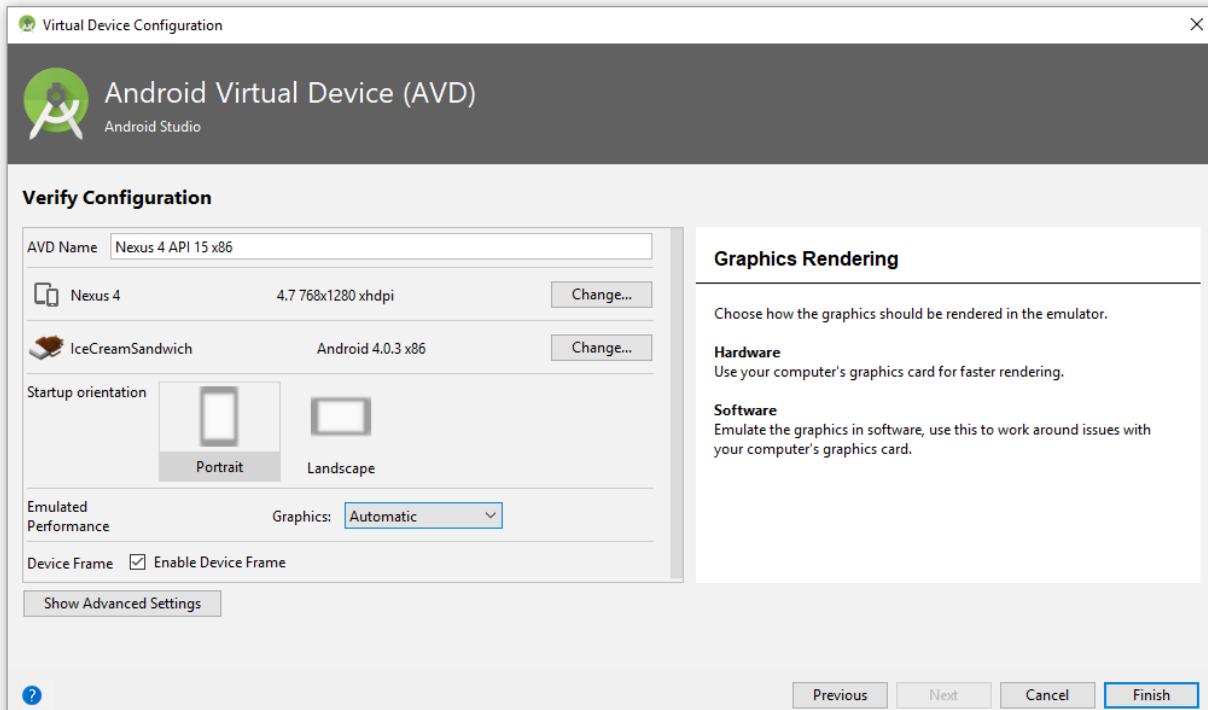
Als **System Image** wählen wir nach einem Klick auf den Schalter **Next** von der Registerkarte **x86 Images** das gemäß Abschnitt 3.1.2 installierte API-Level 15 (mit Google APIs)



Die eingeklammerten Zusätze in den **Target**-Beschreibungen haben folgende Bedeutungen:

- Google APIs  
Es sind Google-Dienste wie Maps und Google+ enthalten.
- Google Play  
Neben den eben genannten Google-Diensten ist auch die Play Store - App enthalten. Für die eben gewählte Gerätedefinition (Nexus 4) ist kein Systemabbild mit Google Play verfügbar.

Nach **Next** erscheint zum neuen virtuellen Gerät der folgende Einstellungsdialog:

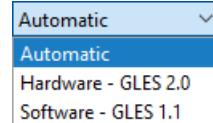


Hier kann man u.a. den **AVD-Namen** ändern und diverse Einstellungen vornehmen:

- Über die **Change**-Schalter kann man die eben vorgestellten Dialoge zur Hardware- bzw. Systemauswahl erneut aufsuchen.

### • Graphics

Bei der Grafikausgabe kann man sich zwischen einer Software-Lösung und einer den Grafikprozessor des Wirtsrechners nutzenden Hardware-Lösung entscheiden, oder der Entwicklungsumgebung die Wahl überlassen:

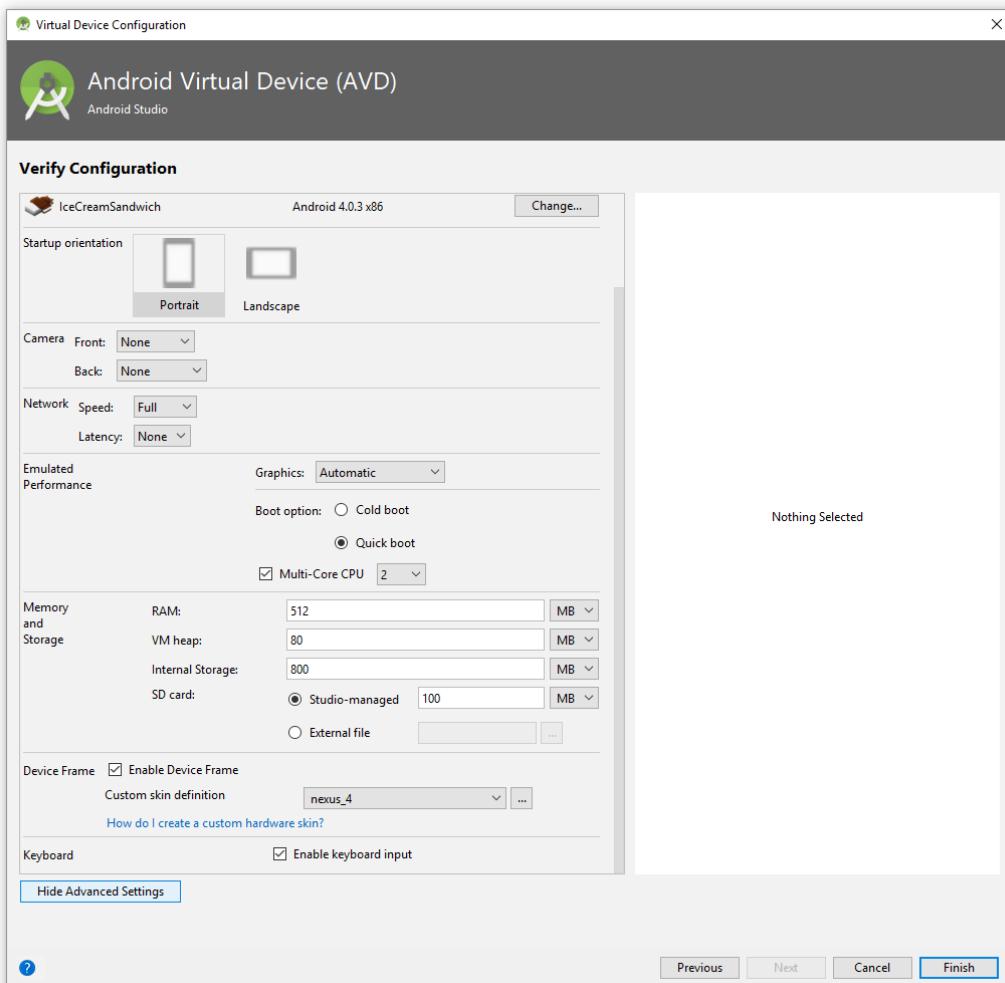


Die Hardware-Beschleunigung der Grafikausgabe ist für die flüssige Bedienbarkeit von virtuellen Android-Geräten von großer Wichtigkeit.

### • Device Frame

Dieses Kontrollkästchen entscheidet darüber, ob im Emulator-Fenster nur die Anzeige, oder auch der Rahmen des nachgeahmten Gerätes erscheinen soll.

Nach einem Klick auf den Schalter **Show Advanced Settings** erscheinen im Konfigurationsdialog



weitere Einstellungsmöglichkeiten, z. B.:

- **Camera**

Wird für die zu testenden Apps keine Kamera benötigt, sollte sie deaktiviert werden, um den Hauptspeicherbedarf des Emulators zu reduzieren

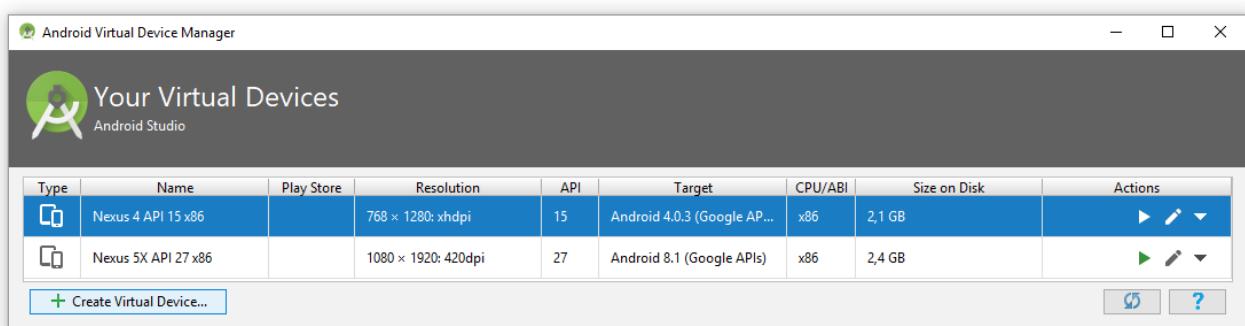
- **RAM**

Steht im Wirtsrechner nur wenig Hauptspeicher zur Verfügung (z. B. 4 GB), sollte die RAM-Ausstattung des Android-Systems möglichst bescheiden gewählt werden.

- **Enable keyboard input**

Ist dieses Kontrollkästchen markiert, benutzt ein virtuelles Android die PC-Tastatur für Eingaben. Wenn Sie (wie bei einem typischen realen Gerät) eine virtuelle Tastatur sehen wollen, müssen Sie in der Gerätekonfiguration die Markierung des Kontrollkästchens **Enable keyboard input** entfernen. Dann erscheint bei Benutzung eines Texteingabefelds eine virtuelle Tastatur.

Nachdem das neue emulierte Gerät per **Finish** erstellt worden ist, erscheint es im AVD-Manager:

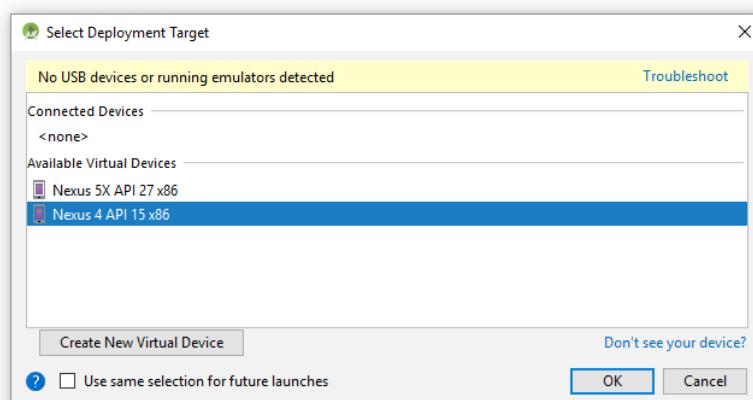


Wird der Emulator zusammen mit einer App gestartet, konkurrieren der Erstellungsprozess und der Emulator um die begrenzte CPU-Kapazität. Daher kann es sich lohnen, den Emulator vor der ersten App-Erstellung zu starten, z. B. so:

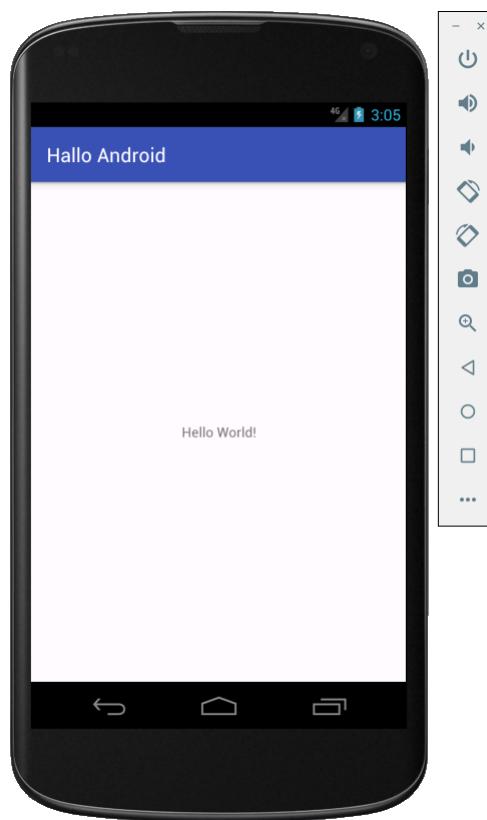
- Aus dem Android Studio per Mausklick auf das Symbol den AVD-Manager starten
- Das gewünschte Gerät per Mausklick auf sein Symbol in der **Actions**-Spalte starten

Klickt man in AVD-Manager auf den Schalter zu einem Gerät, dann erscheint der Einstellungsdialog, den wir oben schon kennengelernt haben.

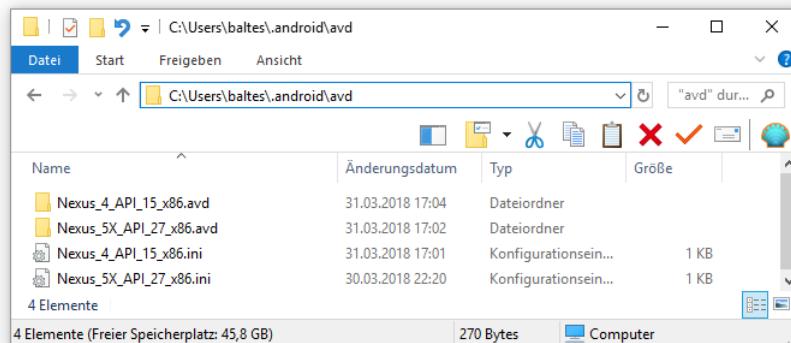
Beim App-Start kann das neu erstellte virtuelle Gerät gewählt werden:



Hier ist die **Hello**-App auf dem sparsamen virtuellen Gerät zu sehen:

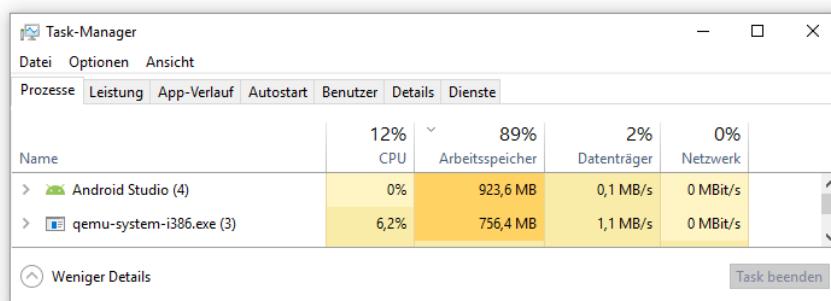


Die virtuellen Geräte werden im Android-Konfigurationsordner abgelegt (vgl. Abschnitt 2.4), z. B.:

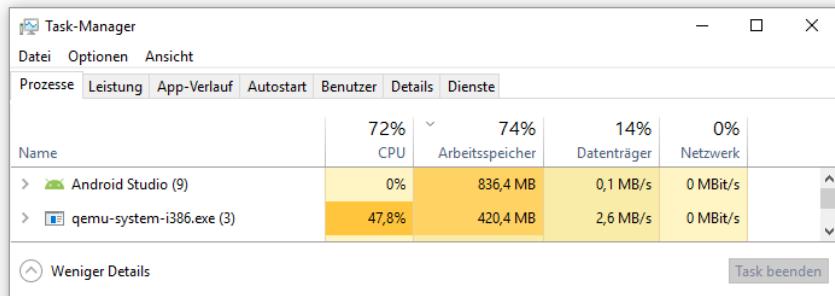


### 3.2.2 Ressourcen-Verbrauch

Die RAM-Ausstattung eines virtuellen Android-Geräts macht sich deutlich im Task-Manager des Windows-Wirtsrechners bemerkbar. Läuft ein virtuelles Gerät mit 1 GB RAM, wird der Hauptspeicher auf einem Wirtsrechner mit insgesamt 4 GB langsam knapp:

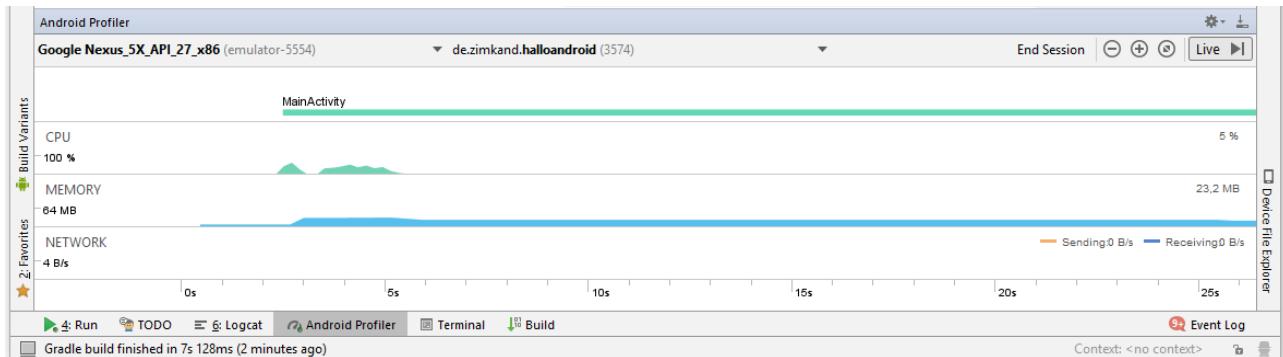


Auf demselben Rechner lässt ein virtuelles Android-System mit 512 RAM mehr Hauptspeicher übrig:



Allerdings bieten die obigen Bildschirmfotos keine verlässlichen Orientierungswerte, weil der RAM-Bedarf des Emulators in Abhängigkeit von der aktuellen Auslastung des virtuellen Gerätes stark variiert.

Über den Ressourcen-Verbrauch *innerhalb* eines virtuellen Android-Systems (ab API-Level 21) informiert das Android Studio im Fenster **Android Profiler**, z. B.:



Der allererste Start eines virtuellen Geräts dauert besonders lang, weil umfangreiche Initialisierungsarbeiten verrichtet werden müssen. Aber auch bei einem fertigen virtuellen Gerät können vom Starten bis zur Bedienbarkeit von Android etliche Sekunden vergehen, wobei die vom Intel Hardware Accelerated Execution Manager (HAXM) unterstützten System-Images deutlich schneller starten als ARM-Images. Wenn ein virtuelles Android-Gerät erst einmal läuft, geht das Starten einer App aus dem Android Studio recht zügig. Daher lässt man den Emulator während der Entwicklungsarbeit laufen. Mit der (leider erst ab API-Level 21 verfügbaren Instant Run - Technik im Android Studio wird der Anwendungsstart nochmals beschleunigt (siehe Abschnitt 4.6).

### 3.3 Intel(R) Hardware Accelerated Execution Manager (HAXM)

Sehr alte CPUs (> 10 Jahre) und solche mit der für Kleinstrechner gedachten Atom-Architektur ausgenommen, besitzt praktisch jede Intel-CPU die Virtualisierungs-Erweiterung VT, sodass Intels Hardware-Beschleunigung für Android genutzt werden kann. Der Intel(R) Hardware Accelerated Execution Managers kann mit Hilfe des SDK-Managers (vgl. Abschnitt 3.1) herunter geladen und installiert werden.<sup>1</sup> Um die interaktive Installation manuell zu starten, setzt man einen Doppelklick auf die herunter geladene und per Voreinstellung im Ordner

...\\sdk\\extras\\intel\\Hardware\_Accelerated\_Execution\_Manager

<sup>1</sup> Der HAXM ist (am 31.03.2018 in der Version 7.0.0) auch auf der folgenden Intel-Webseite verfügbar:

<https://software.intel.com/en-us/android/articles/intel-hardware-accelerated-execution-manager>

Bei meinen Tests ist mir kein Unterschied zwischen den Version 6.2.1 (mit dem Android SDK installiert) und 7.0.0 (von Intel angeboten) aufgefallen.

abgelegte Datei **intelhaxm-android.exe**. Dabei sollte der Installationsrechner mit dem Internet verbunden sein, damit bei Bedarf Updates bezogen werden können.



Die Installation landet im Ordner:

**C:\Program Files\Intel\HAXM**

Im Ordner

**...\\sdk\\extras\\intel\\Hardware\_Accelerated\_Execution\_Manager**

befindet sich auch eine Befehlsdatei zur Durchführung einer automatischen Installation (z. B. in einem Schulungsraum).

Um unter Windows zu überprüfen, ob HAXM tatsächlich installiert und einsatzbereit ist, wechselt man in einem Konsolenfenster zum Ordner

**...\\Sdk\\emulator**

und startet das Programm **emulator.exe** mit dem Kommandozeilenparameter **-accel-check**:

```
C:\ Eingabeaufforderung
C:\Users\baltex\AppData\Local\Android\Sdk\emulator>emulator -accel-check
Warning: Quick Boot / Snapshots not supported on this machine. A CPU with EPT + UG features is
currently needed. We will address this in a future release.
accel:
0
HAXM version 7.0.0 <4> is installed and usable.
accel
```

Um unabhängig von Android SDK den HAXM-Status zu ermitteln, kann unter Windows das Kommando

**sc query intelhaxm**

verwendet werden, z. B.:

```
C:\ Eingabeaufforderung
c:\>sc query intelhaxm
SERVICE_NAME: intelhaxm
        TYPE               : 1  KERNEL_DRIVER
        STATE              : 4  RUNNING
                                <STOPPABLE, NOT_PAUSABLE, IGNORES_SHUTDOWN>
        WIN32_EXIT_CODE    : 0  <0x0>
        SERVICE_EXIT_CODE : 0  <0x0>
        CHECKPOINT        : 0x0
        WAIT_HINT         : 0x0
```

Mit dem Programm **emulator.exe** lässt sich eine virtuelle Maschine auch per Konsole starten, und in einem Fall ist es mir nur auf diese Weise gelungen, die Hardware-Beschleunigung der Grafikausgabe zu aktivieren (Option **-gpu host**):

```
C:\Users\baltes\AppData\Local\Android\Sdk\emulator>emulator -avd Nexus_4_API_15_x86 -gpu host
emulator: WARNING: encryption is off
Warning: Quick Boot / Snapshots not supported on this machine. A CPU with EPT + UG features is currently needed. We will address this in a future release.
emulator: WARNING: Not all modern X86 virtualization features supported, which introduces problems with slowdown when running Android on multicore vCPUs. Setting AVD to run with 1 vCPU core only.
HAX is working and emulator runs in fast virt mode.
audio: Failed to create voice 'goldfish_audio_in'
qemu-system-i386.exe: warning: opening audio input failed
audio: Failed to create voice 'adc'
glClear:470 GL err 0x506
glClear:470 GL err 0x506
glMatrixMode:1543 GL err 0x506
```

Neben zahlreichen Warn- und Fehlermeldungen ohne wesentliche Folgen erhält man bei dieser Startart auch eine explizite Bestätigung der HAXM-Nutzung:

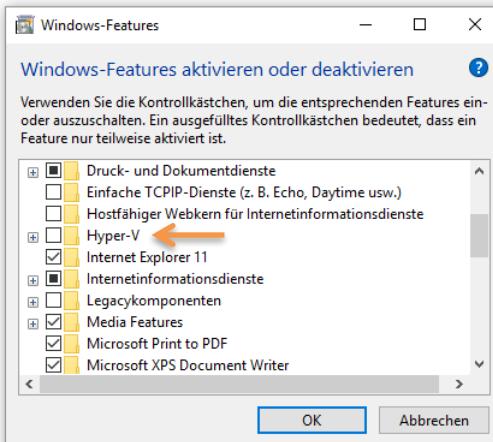
HAX is working and emulator runs in fast virt mode

Beim Start derselben virtuellen Maschine per AVD-Manager oder via ▶ - Schalter im Android Studio blieb die Hardware-Beschleunigung der Grafikausgabe abgeschaltet und die Bedienung des virtuellen Geräts unangenehm zäh.

Ist Hyper-V in Windows aktiviert, kann HAXM nicht installiert werden. Über

### **Systemsteuerung > Programme deinstallieren > Windows-Features aktivieren oder deaktivieren**

lässt sich Hyper-V deaktivieren:<sup>1</sup>

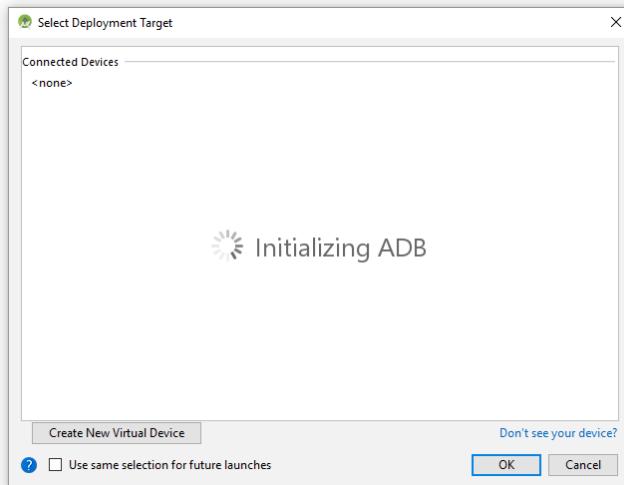


### **3.4 Android Debug Bridge**

Über die Android Debug Bridge (ADB) kann man vom Entwicklungsrechner aus mit einem emulierten oder mit einem per USB oder WLAN angeschlossenen Android-Gerät kommunizieren, um z. B. eine App auf dem Gerät zu installieren oder Debug-Informationen über eine App zu sammeln. Wenn beim Starten der ersten Emulatorinstanz die Server-Komponente der Android Debug Bridge noch nicht läuft, wird sie gestartet, z. B. vom Android Studio:

---

<sup>1</sup> <https://docs.microsoft.com/de-de/xamarin/android/get-started/installation/android-emulator/hardware-acceleration?tabs=vswin>



Unter Windows wird die Datei **adb.exe** gestartet, die sich im folgenden SDK-Unterordner befindet:

**...\\Sdk\\platform-tools**

Man kann den ADB-Server auch per Kommando starten:

```
C:\ Eingabeaufforderung
C:\Users\baltes\AppData\Local\Android\Sdk\platform-tools>adb start-server
* daemon not running; starting now at tcp:5037
* daemon started successfully
```

Das ist im Normalfall nicht erforderlich, weil der Server bei einer Klientenanforderung automatisch gestartet wird. Wenn sich ein Android Gerät nicht per ADB ansprechen lässt (z. B. aus der Entwicklungsumgebung), kann es helfen, den ADB-Server zu beenden, damit er durch die nächste Klientenanforderung neu gestartet wird:

```
C:\ Eingabeaufforderung
C:\Users\baltes\AppData\Local\Android\Sdk\platform-tools>adb kill-server
```

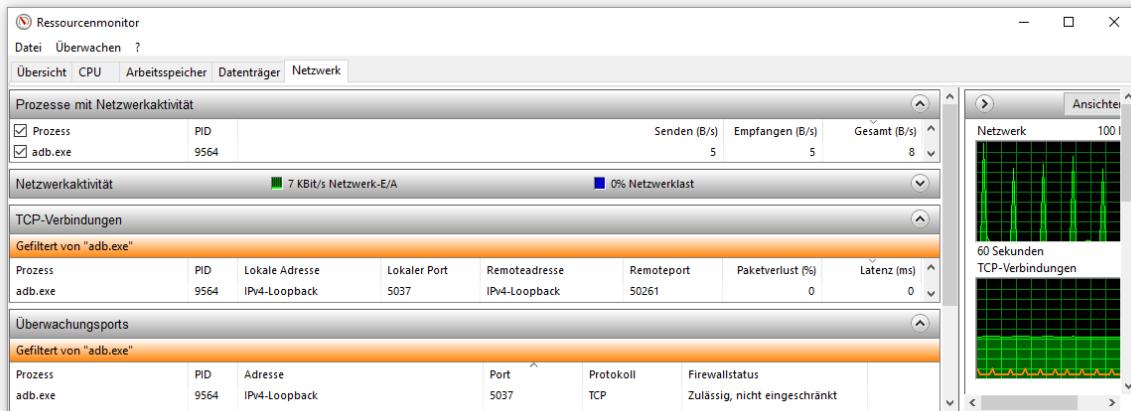
Die wesentlichen ADB-Bestandteile sind:<sup>1</sup>

- Der Server

Er läuft als Hintergrundprozess auf der Entwicklungsmaschine, lauscht am TCP-Port 5037 auf Anfragen von Klienten und kümmert sich um die Kommunikation zwischen den Klienten und den ADB-Dämonen auf den Android-Geräten. Im folgenden Bildschirmfoto des Windows-Ressourcenmanagers sind die Netzwerkaktivitäten des auf der Datei **adb.exe** basierenden Servers zu beobachten:

---

<sup>1</sup> Weitere Hinweise sind hier zu finden: <https://developer.android.com/studio/command-line/adb.html>



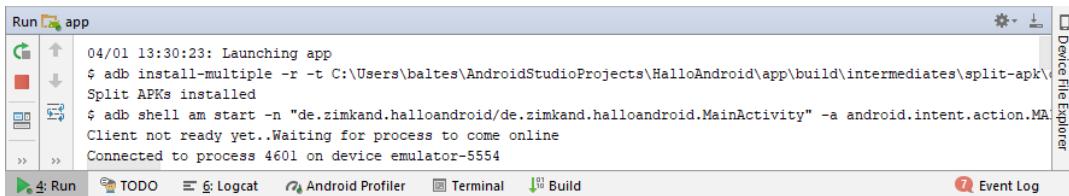
- **Der Geräte-Dämon**

Auf jedem emulierten oder per USB angebundenen Android-Gerät läuft im Hintergrund ein als *ADB-Dämon* bezeichneter Prozess, der die Anforderungen der Klienten bedient.

- **Ein Client**

Er läuft auf der Entwicklungsmaschine und sendet Anforderungen an ein Android-Gerät. Meist kommt dabei die oben schon erwähnte Datei **adb.exe** zum Einsatz, die zahlreiche Kommandos zur Kommunikation mit einem emulierten oder per USB angeschlossenen Android-Gerät verarbeitet kann.

Auch das Android Studio verwendet im Hintergrund diverse **adb**-Kommandos, um z. B. Apps auf einem angeschlossenen Android-Gerät zu installieren und zu starten, um das **logcat**-Fenster der Entwicklungsumgebung mit Statusmeldungen aus dem Android-Gerät zu füllen und für andere Diagnosezwecke (siehe Abschnitt 6.3.1). Im **Run**-Fenster wird dokumentiert, welche Aktionen das Android Studio unternimmt, um eine App zu installieren und zu starten, z. B.:



Das flexible **adb**-Kommando kann auch die aktuell verbundenen Geräte anzeigen, z. B.:



Gelegentlich ist es für einen Entwickler sinnvoll, mit einem Android-Gerät per Konsole direkt zu kommunizieren. Mit dem Kommando **adb shell** lässt sich eine Konsolenverbindung zur ersten angeschlossenen Android-Instanz aufbauen. Anschließend kann man Linux-Kommandos ausführen, um z. B. die Dateisystemeinträge mit **ls -al** auflisten zu lassen:

```
C:\Users\baltes\AppData\Local\Android\Sdk\platform-tools>adb shell
# ls -al
drwxr-xr-x root      root          2018-03-31 23:25 acct
drwxrwx--- system    cache        2018-03-31 18:11 cache
dr-x----- root      root          2018-03-31 23:25 config
lrwxrwxrwx root      root          2018-03-31 23:25 d -> /sys/kernel/debug
drwxrwxr-x system    system       2018-03-31 15:13 data
-rw-r--r-- root      root          116 1970-01-01 00:00 default.prop
drwxr-xr-x root      root          2018-03-31 23:25 dev
lrwxrwxrwx root      root          2200076 1970-01-01 00:00 etc -> /system/etc
-rw-r--r-- root      root          2561 1970-01-01 00:00 init
-rw-r--r-- root      root          3283 1970-01-01 00:00 init.goldfish.rc
-rw-r--r-- root      root          17052 1970-01-01 00:00 init.ranchu.rc
-rw-r--r-- root      root          2018-03-31 23:25 init.rc
drwxr-xr-x root      system       2018-03-31 23:25 mnt
dr-xr-xr-x root      root          2018-03-31 23:25 proc
drwxr----- root      root          2016-02-26 22:39 root
drwxr-x--- root      root          1970-01-01 00:00 sbin
lrwxrwxrwx root      root          2018-03-31 23:25 sdcard -> /mnt/sdcard
drwxr-xr-x root      root          2018-03-31 23:25 storage
drwxr-xr-x root      root          2018-03-31 23:25 sys
drwxr-xr-x root      root          1970-01-01 00:00 system
-rw-r--r-- root      root          272 1970-01-01 00:00 ueventd.goldfish.rc
-rw-r--r-- root      root          272 1970-01-01 00:00 ueventd.ranchu.rc
-rw-r--r-- root      root          3825 1970-01-01 00:00 ueventd.rc
lrwxrwxrwx root      root          2018-03-31 23:25 vendor -> /system/vendor
-rw-r--r-- root      root          1093 1970-01-01 00:00 vold.fstab.goldfish
-rw-r--r-- root      root          1068 1970-01-01 00:00 vold.fstab.ranchu
#
#
```

Um ein bestimmtes Gerät anzusprechen, gibt man seinen Namen als Wert für die Option **-s** an, z. B.:

```
>adb -s emulator-5554 shell
```

Emulatorinstanzen wählen für die Kommunikation mit dem ADB-Server einen TCP-Port mit einer ungeraden Nummer im Bereich ab 5555. Außerdem belegen sie für Konsolenverbindungen noch einen TCP-Port mit der nach unten nächstgelegenen geraden Zahl. Folglich verwendet die erste Emulatorinstanz die beiden folgenden Ports:

5554 für die Konsole

5555 für die Kommunikation mit dem ABD-Server

### **3.5 Reales Android-Gerät per USB mit der Entwicklungsumgebung verbinden**

Manchmal ist ein reales Android-Gerät zum Testen einer App sinnvoller als ein emuliertes Gerät, z. B. weil in der App Sensoren angesprochen werden (z. B. Beschleunigung, Magnetismus). Über die anschließenden Erläuterungen hinausgehende Hinweise finden sich z. B. im Google-Webangebot für Android-Entwickler:

<https://developer.android.com/studio/run/device.html>

#### **3.5.1 USB-Debugging auf dem Android-Gerät aktivieren**

Auf dem Android-Gerät muss das USB-Debugging aktiviert werden, was bis Android 4.1 relativ einfach gelingt:

- Starten Sie die App zur Einstellungsverwaltung.
- Wählen Sie dort ...
  - bei Android-Versionen bis 3.2 die Option **Anwendungen > Entwicklung**,
  - bei Android-Versionen ab 4.0 die Option **Entwickleroptionen**.
- Aktivieren Sie das **USB-Debugging**.

Ab Android 4.2 müssen die Entwickleroptionen erst zugänglich gemacht werden:

- Starten Sie die App zur Einstellungsverwaltung.
- Öffnen Sie die Geräteinformationen (z. B. unter dem Namen **Über dieses Telefon**).
- Tippen Sie siebenmal auf die **Build-Nummer**.
- Wechseln Sie zurück zur Hauptebene der Einstellungs-App, wo nun die **Entwickleroptionen** zu finden sind.

### 3.5.2 Android USB-Treiber installieren

Auf einem Windows-Rechner muss zu einem Android-Gerät ein USB-Treiber installiert werden, damit das Android Studio zu testende Software per ADB (Android Debug Bridge) auf das Gerät schicken kann. Für ein Google-Smartphone (z. B. aus der Nexus- oder Pixel-Serie) eignet sich der per SDK-Manager verfügbare USB-Treiber (vgl. Abschnitt 3.1). Bei aktivierter Option wird der Treiber vom SDK-Manager allerdings lediglich herunter geladen, z. B. unter Windows beim Benutzer **theo** in den Ordner:

**C:\Users\theo\AppData\Local\Android\Sdk\extras\google\usb\_driver**

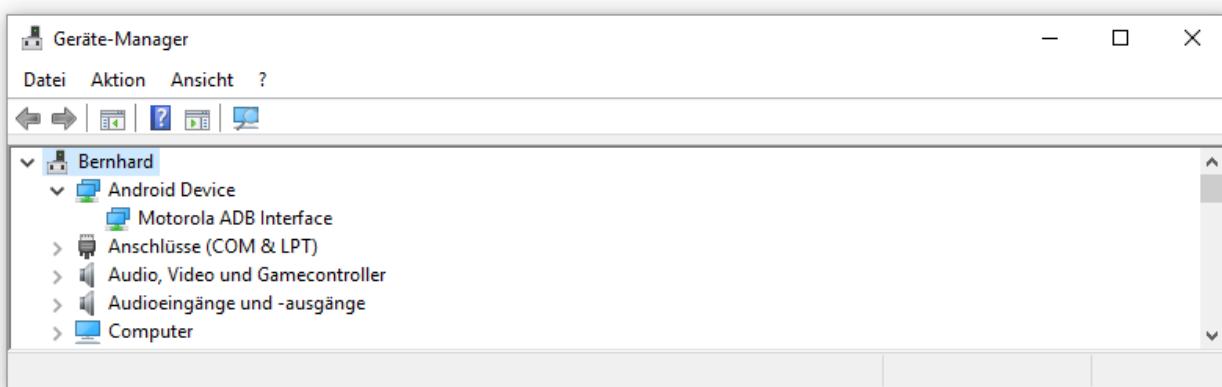
Informationen und Links zu den Treiber-Download-Seiten anderer Hersteller finden sich hier:

<https://developer.android.com/studio/run/oem-usb.html>

Zu einem Motorola Smartphone wird ein Programm namens *Device Manager* angeboten, und sein Installationsprogramm kümmert sich auch um den USB-Treiber für die Android Debug Bridge:



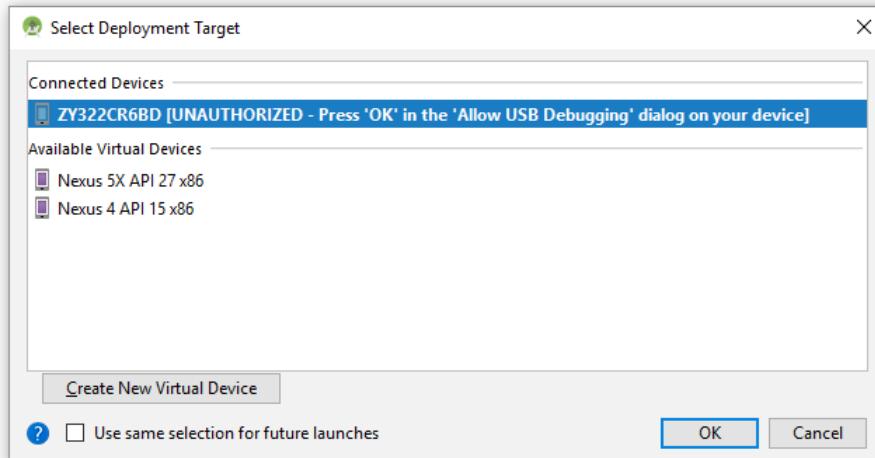
Sobald nach der Installation das Motorola-Smartphone mit aktiviertem USB-Debugging per USB-Kabel mit dem Entwicklungsrechner verbunden wird, erscheint ein **Android Device** im Windows-Gerätemanager:



Auf der oben angegebenen Webseite werden generelle Installationsschritte für die USB-Treiber zu einem Android-Geräten beschrieben.

### 3.5.3 Entwicklungsrechner auf dem Android-Gerät zulassen

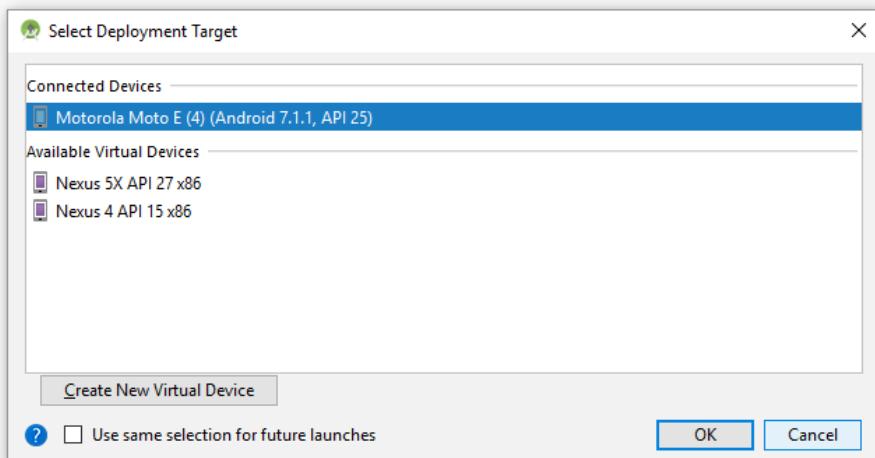
Ab Android 4.2.2 wird aus Sicherheitsgründen einem fremden Rechner die Kooperation mit dem lokalen ADB-Dämon (vgl. Abschnitt 3.4) nur dann erlaubt, wenn die entsprechende Nachfrage auf dem Android-Gerät positiv beantwortet wird. Beim ersten Versuch, mit dem Android Studio eine App auf das Android-Gerät zu übertragen, wird über die Notwendigkeit einer Zulassung informiert:



Das Android-Gerät fragt nach, ob das USB-Debugging zugelassen werden soll, und zeigt der Fingerabdruck des RSA-Schlüssels des Entwicklungsrechners an. Wenn die Abfrage nicht erscheint, hilft eventuell das Stoppen des ADB-Servers:



Nach erfolgter Zulassung klappt die Installation einer App ohne Nachfrage:



Die Autorisierung eines Entwicklungsrechners kann über das Item **USB-Debugging-Autorisierungen aufheben** in den Entwickleroptionen des Android-Geräts widerrufen werden.

### ***3.6 Übungsaufgaben zu Kapitel 3***

- 1) Verbinden Sie nach Möglichkeit ein eigenes Android-Testgerät per USB mit Ihrem Entwicklungsrechner.

## 4 Android Studio kennenlernen

In diesem Abschnitt werden wir eine erste „richtige“ App erstellen, die das Kürzen eines Bruchs ermöglicht. Wenngleich uns dabei unweigerlich Konzepte aus dem Java-Framework einer Android-Anwendung begegnen, verfolgen wir vor allem das Ziel, mit der Bedienung des Android Studios vertraut zu werden.

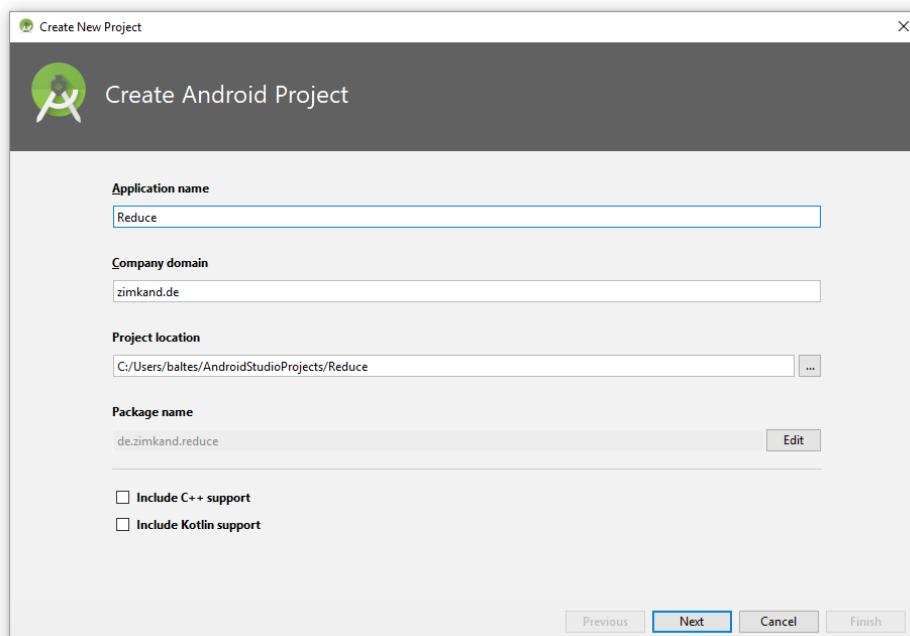
Weil im weiteren Verlauf des Manuskripts sehr oft vom *Android Studio* die Rede sein wird, vereinbaren wir die Abkürzung AS, sprechen also z. B. von *AS-Projekten*.

### 4.1 Projekt anlegen

Wir legen gemäß der Beschreibung in Abschnitt 2.3 ein neues Projekt an, starten also z. B. bei aktivem Android Studio mit dem Menübefehl

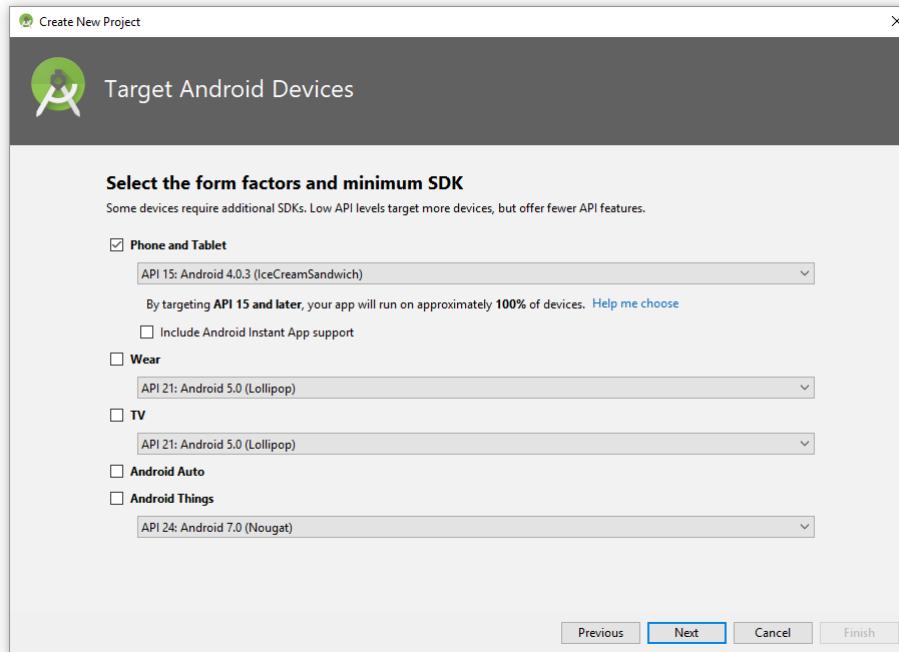
**File > New > New Project**

Im ersten Assistentendialog

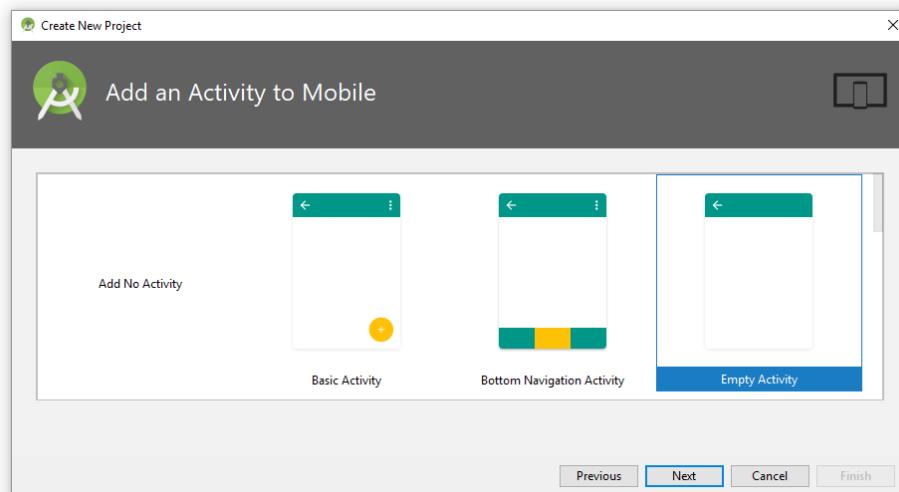


wählen wir Reduce als **Application name** und eine erfundene **Company Domain** namens **zimkand.de**.

Im nächsten Dialog akzeptieren wir den sinnvollen Vorschlag, dass unser Programm auf Smartphones und Tablets mit der minimalen Android-Version 4.0.3 (API-Level 15) verwendbar sein soll:

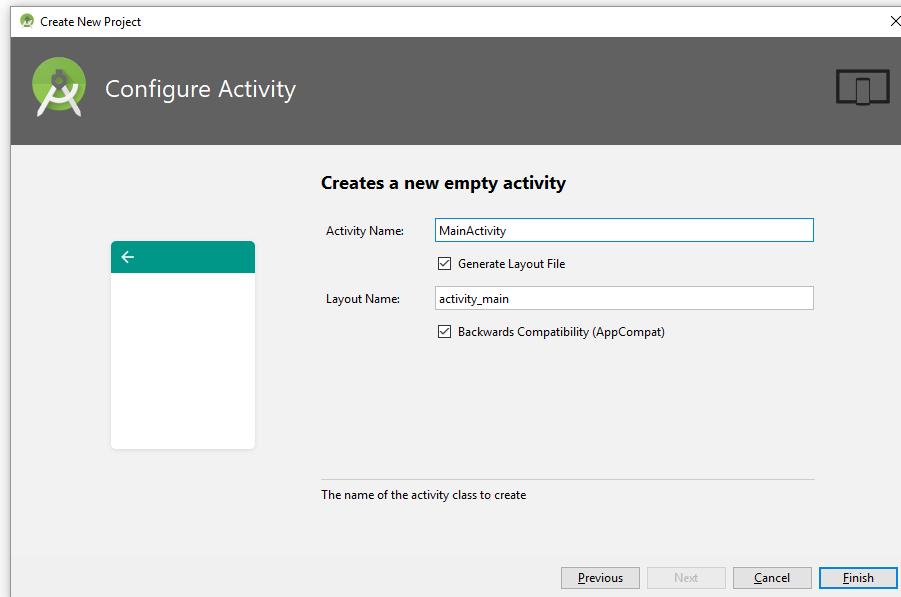


Im nächsten Dialog akzeptieren wir den Vorschlag des Assistenten, in der neuen App gleich eine Activity (eine Bildschirmseite) anzulegen. Weil wir uns um deren Bedienelemente selbst kümmern wollen, behalten wir die Voreinstellung **Empty Activity** bei:

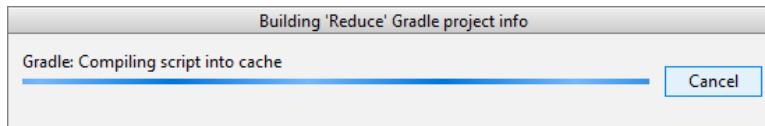


Im letzten Dialog des Assistenten akzeptieren wir ...

- den vorgeschlagenen Namen **MainActivity** für die Java-Klasse zur Aktivität, die der wesentliche Bestandteil unserer App sein wird,
- den vorgeschlagenen Namen **activity\_main.xml** für die Datei, in der das Layout der einzigen Aktivität unserer App definiert wird.

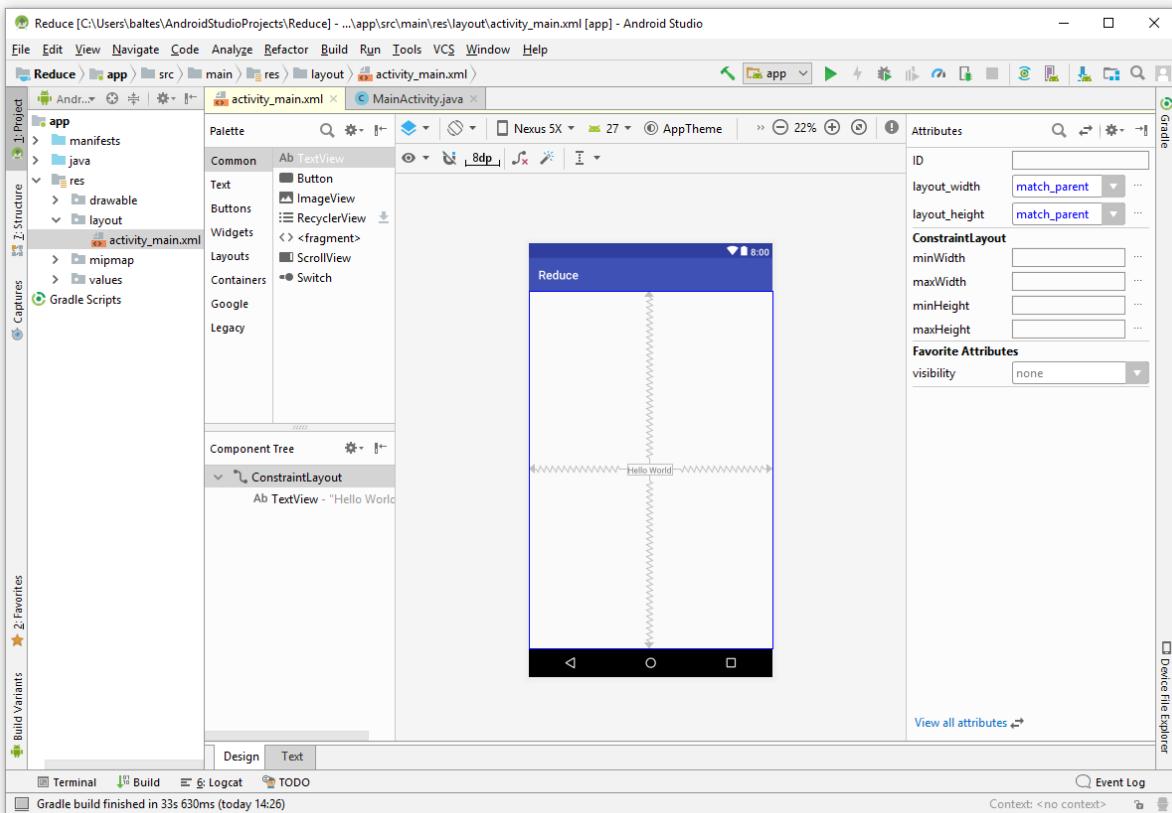


Mit einem Klick auf den Schalter **Finish** veranlassen wir die Erstellung des neuen Projekts. Dabei wird ein sogenanntes **Build-System** tätig, das aus Quellcodedateien sowie Ressourcen-Dateien (z. B. im XML-Format zur Beschreibung der Bedienoberfläche einer Aktivität) schlussendlich die **apk**-Datei zur Anwendung erzeugt. Im Android-Studio ist ein Build-System namens **Gradle** im Einsatz, wie der folgende Tätigkeitsbericht zeigt:



Gradle ist Skript-basiert, und wir könnten das von Google erstellte Skript **build.gradle** ändern. In der Regel werden wir im Kurs aber nicht in den Build-Prozess eingreifen.

Im Fenster mit dem neuen Projekt aktivieren wir die mit **activity\_main.xml** beschriftete Editor-Registerkarte, um die Bedienoberfläche unserer App zu bearbeiten:

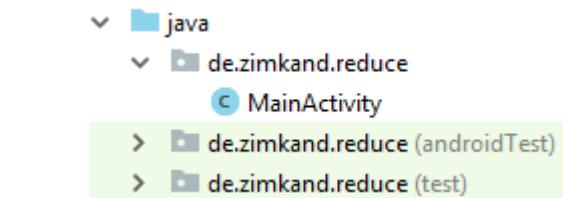


Zunächst macht auch unsere zweite App nichts anderes, als den Text „Hello world!“ zu zeigen.

Im **Project**-Fenster (ab jetzt meist als *Projektexplorer* bezeichnet) ist das Modul **app** mit einer Baumsicht seiner Bestandteile zu sehen. Im Android Studio bzw. in der zugrunde liegenden Entwicklungsumgebung IntelliJ kann ein Projekt mehrere **Module** (vom Typ **App, Test, Library** oder **App Engine**) enthalten. Allerdings werden sich im Kurs alle Projekte auf ein *einziges* Modul vom Typ **App** beschränken, das zudem den Namen **app** trägt.

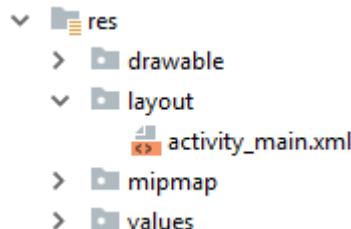
Das Drop-Down - Menü in der Titelzeile des Projektexplorers bietet mehrere Sichten, von denen für uns nur die beiden folgenden von Bedeutung sind:

- Die voreingestellte, oben dargestellte Sicht **Android** löst sich vom Aufbau des Projektordners im Dateisystem zugunsten einer inhaltlich-funktionalen Gliederung. Zu einer App sind drei Hauptbestandteile zu sehen:
  - **manifests**  
Hier befindet sich die Datei **AndroidManifest.xml** mit wichtigen Informationen über die App und ihre Komponenten:
    - ▼ **manifests**
    - AndroidManifest.xml**  - **java**  
Hier befinden sich nach Paketen geordnet die Quellcodedateien mit den Klassen und Schnittstellen der App. Wir interessieren uns zunächst nur für die Klassen der eigentlichen Projektentwicklung (Paketname im Beispiel: `de.zimkand.reduce`) und ignorieren die Pakete mit Testklassen (`de.zimkand.reduce (androidTest)`, `de.zimkand.reduce (test)`):



- **res**

Hier befinden sich die Ressourcen der App (z. B. Layout-Definitionen, Zeichenfolgen):

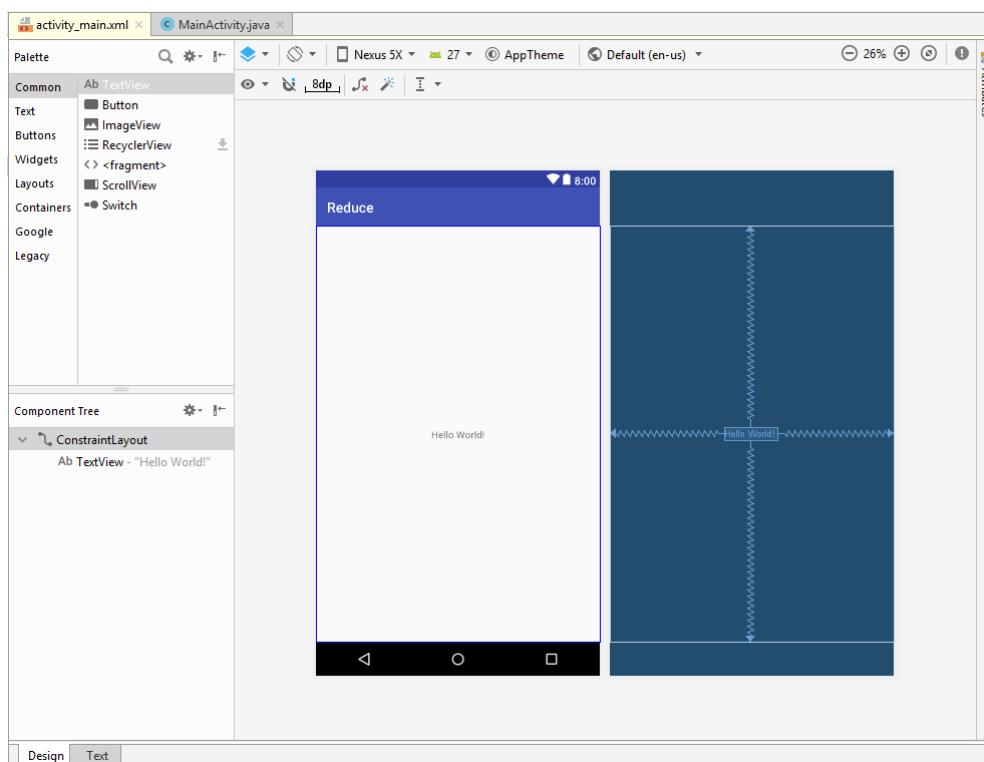


- In der alternativen Sicht **Project** ist die Ordnerstruktur des Dateisystems zu sehen.

In der **Android**-Sicht sind auf der Hauptebene des Projektexplorers neben den Modulen auch die **Gradle-Scripts** zu sehen, über die das Gradle-Erstellungssystem aus den Bestandteilen einer App (Manifest, Quellcode, Ressourcen) eine **apk**-Datei (Android Package) erstellt, die auf den Zielgeräten installiert werden kann.

## 4.2 Bedienoberfläche entwerfen

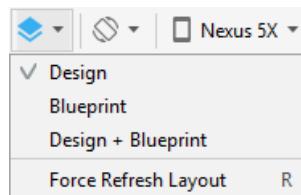
Der Projektexplorer zeigt als wesentlichen app-Bestandteil im Pfad **res/layout** die Datei **activity\_main.xml**, welche die Bedienoberfläche der einzigen Aktivität unserer Anwendung definiert. Wird diese Datei in der Editorzone geöffnet, ist per Voreinstellung die **Design**-Registerkarte aktiv:



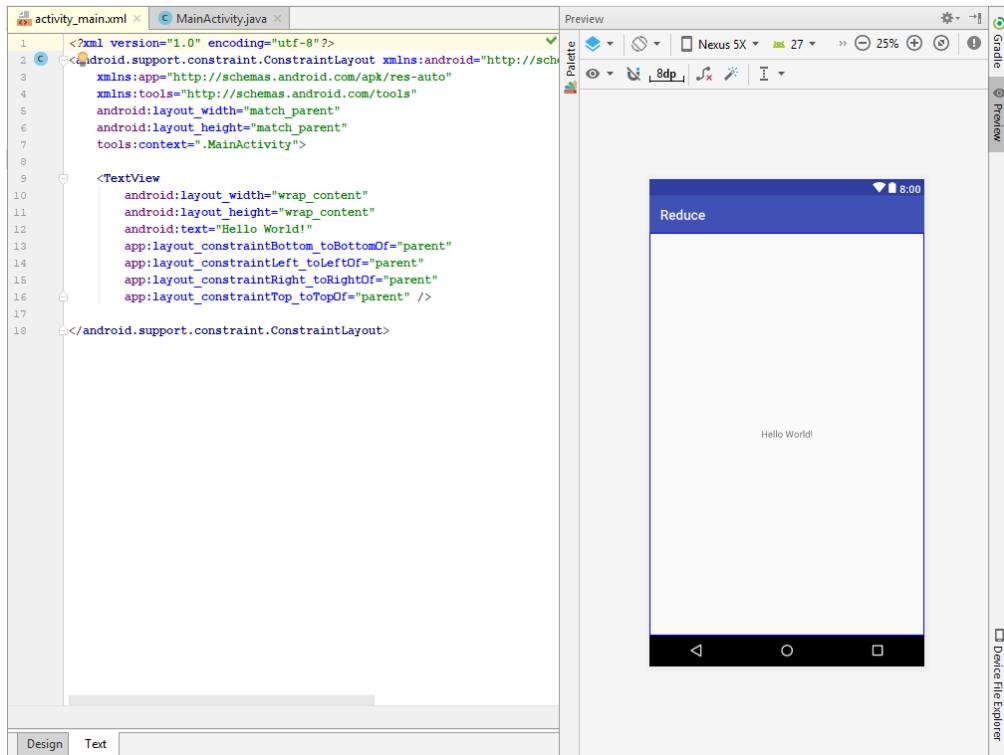
Bei Bedarf können Sie diesen Zustand per Doppelklick auf die Datei **activity\_main.xml** herstellen.

Neben der realitätsnahen **Design**-Ansicht (links) ist die **Blueprint**-Ansicht (rechts) verfügbar, die nur die Grundstruktur der Aktivität zeigt. Wir benötigen diese Ansicht für das aktuelle Beispiel

nicht und eliminieren sie über das Werkzeug zur Gestaltung der Editor-Bedienoberfläche (oben links):

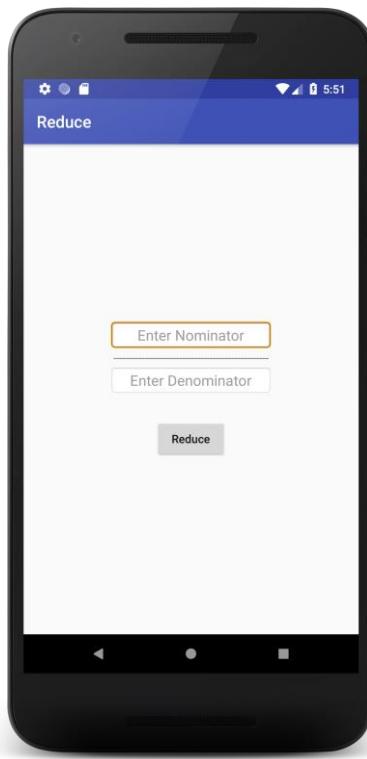


Manche Änderungen der GUI-Definition lassen sich ausschließlich oder bequemer im **Text**-Modus erreichen. Man editiert den XML-Code zur Definition der Bedienoberfläche und kann den Effekt in einem **Preview**-Fenster besichtigen:



Zum Umschalten zwischen dem **Design**- und dem **Text**-Modus befinden sich zwei Laschen in der linken unteren Ecke der Editierzone.

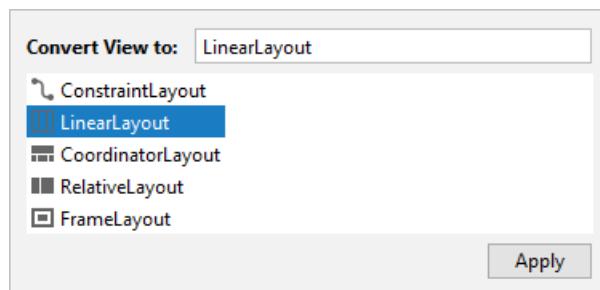
Die zu erstellende Bedienoberfläche unserer App soll ungefähr so aussehen



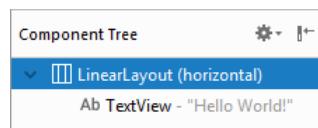
und folgende Steuerelemente enthalten:

- Zwei Texteingabefelder zur Eingabe von Zähler und Nenner eines Bruchs
- Ein Label zur Darstellung des Bruchstrichs
- Einen Schalter zur Anforderung des Kürzens

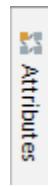
Wie der **Component Tree** der Entwicklungsumgebung zeigt (auf der **Design**-Registerkarte links unten), ist als „Platzanweiser“ für die Steuerelemente (Texteingabefelder, Label, Schalter) ein Objekt der Klasse **ConstraintLayout** aktiv. Es ist leistungsfähig, aber für unsere geplante Anwendung überdimensioniert, sodass wir uns den Einstieg erleichtern durch den Wechsel zu einem Layout-Manager der Klasse **LinearLayout** mit vertikaler Anordnung der Steuerelemente. Wählen Sie aus dem Kontextmenü zum **ConstraintLayout**-Objekt das Item **Convert view**, und entscheiden Sie sich für die Klasse **LinearLayout**:



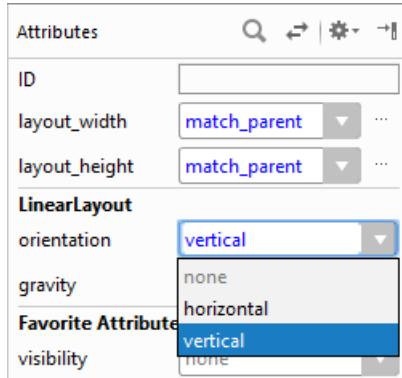
Im **Component Tree** - Fenster ist der neue Platzanweiser zu sehen:



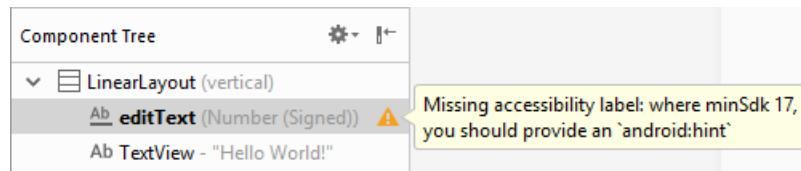
Im **Attributes**-Fenster, das bei Bedarf über die Schaltfläche



am rechten Fensterrand zu öffnen ist, kann man dem **LinearLayout** eine vertikale Orientierung verpassen:

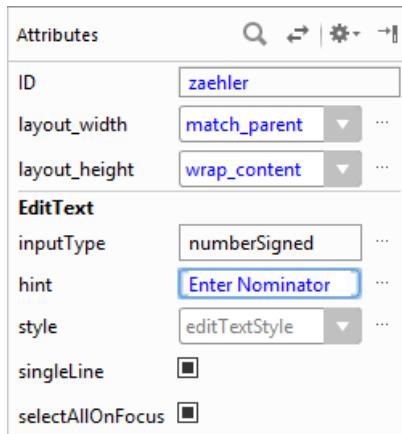


Am linken Rand des Editors befindet sich die **Palette** mit den Steuerelementen. Entnehmen Sie aus der Abteilung **Text** ein Steuerelement vom Typ **Number (Signed)** und fügen Sie es per Drag & Drop im Fenster **Component Tree** über dem vorhandenen **TextView**-Objekt ein:



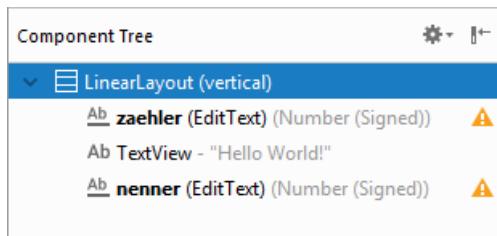
Hier sollen die Benutzer den Zähler des zu kürzenden Bruchs eintragen.

Der Aufforderung, den Benutzer über den Zweck des Texteingabefelds zu informieren, leisten wir Folge durch den folgenden Eintrag im **hint**-Feld des **Attributes**-Fensters:

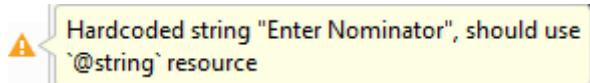


Die Steuerelemente werden im Programm über eine Kennung angesprochen, sodass eine aussagekräftige Benennung zu einem besser lesbaren Quellcode führt. Tragen Sie daher bei markiertem Texteingabefeld im Feld **ID** des **Attributes**-Fensters den Namen **zaehler** ein (siehe oben).

Fügen Sie unter dem vorhandenen **TextView**-Objekt ein weiteres Steuerelement vom Typ **Number (Signed)** ein, vereinbaren Sie einen Erläuterungstext, und vergeben Sie den Namen **nenner**, sodass im Fenster **Component Tree** der folgende Stand resultiert:



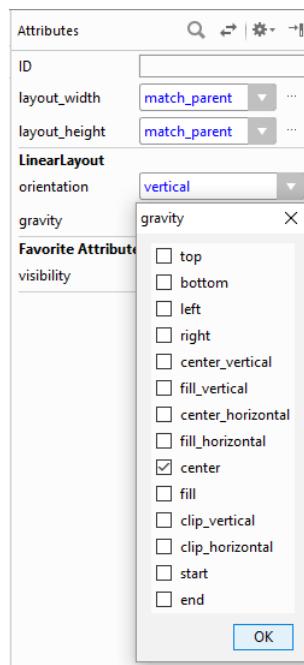
Mit den Warndreiecken will uns die Entwicklungsumgebung sagen, dass Zeichenfolgen in Android-Apps nicht direkt kodiert, sondern durch String-Ressourcen (siehe Abschnitt 7.4) realisiert werden sollten, damit die App leicht internationalisiert werden kann. Wenn Sie mit Maus auf ein Warndreieck zeigen, erfahren Sie seine Bedeutung:



Momentan ignorieren wir den berechtigten Hinweis.

Markieren Sie im Fenster **Component Tree** das **TextView**-Objekt, und tragen Sie im **Attributes**-Fenster als **text** ca. 50 Bindestriche, die (etwas provisorisch) den Bruchstrich darstellen sollen.

Sorgen Sie für eine (vertikale und horizontale) Zentrierung der drei Steuerelemente, indem Sie den Platzanweiser vom Typ **LinearLayout** markieren und dann im **Attributes**-Fenster die Eigenschaft **gravity** auf den Wert **center** setzen:

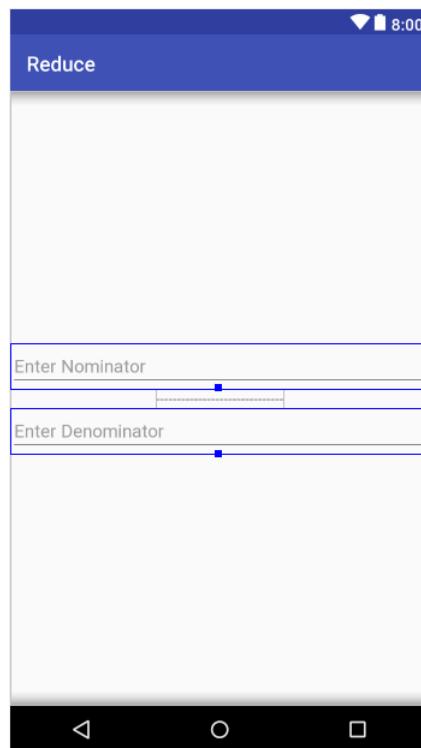


Reduzieren Sie die Breite der Texteingabefelder für Zähler und Nenner, indem Sie als Wert der Eigenschaft **layout\_width** die Vorgabe **match\_parent** durch **wrap\_content** ersetzen. Bei den Texteingabefeldern ist das Attribut **android:ems** auf den Wert 10 voreingestellt,

**android:ems="10"**

und ein **em** ist gerade die Breite des Buchstabens **M** in der aktuellen Schrift. Damit ist der Content angemessen dimensioniert, und der Wert **wrap\_content** für das Attribut **layout\_width** führt zu einer passenden Breite für die Texteingabefelder.

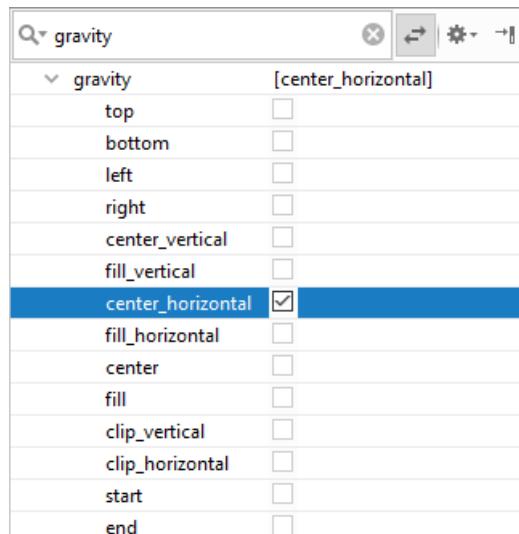
Damit Sie die Eintragung nicht zweimal vornehmen müssen, sollten Sie zunächst in der **Design**-Zone die *beiden* Texteingabefelder markieren, was mit Hilfe der Umschalt-Taste gelingt:



Anschließend kann im **Attributes**-Fenster die Eigenschaft für beide Steuerelemente festgelegt werden:

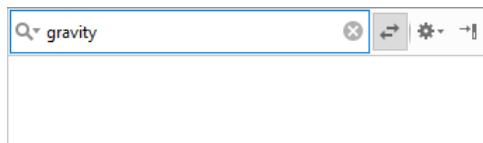


Setzen Sie die Eigenschaft **gravity** für beide Texteingabefelder auf den Wert **center\_horizontal**, damit die vom Benutzer eingetragenen Werte horizontal zentriert werden.

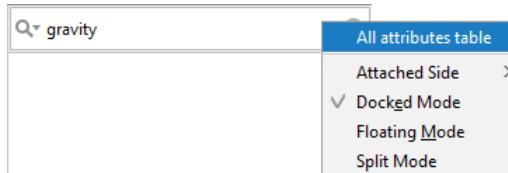


Die Eigenschaft **gravity** muss durch einen Klick auf den Schalter (am oberen Fensterrand) oder auf den Text **View all attributes** (am unteren Fensterrand) zugänglich gemacht werden. Um nicht lange in der umfangreichen Liste aller Eigenschaften stöbern zu müssen, trägt man am besten den Eigenschaftsnamen in das Suchfeld ein.

Wenn die Suche in der vollständigen Liste aller Attribute ohne Ergebnis bleibt,

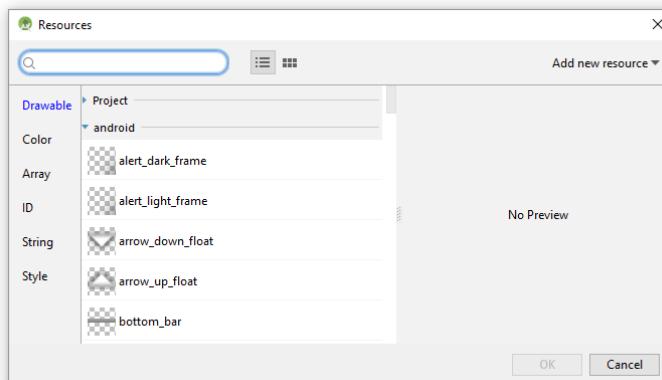


dann müssen Sie aus dem Drop-Down-Menü zum Schalter das Item **All Attributes table** wählen:

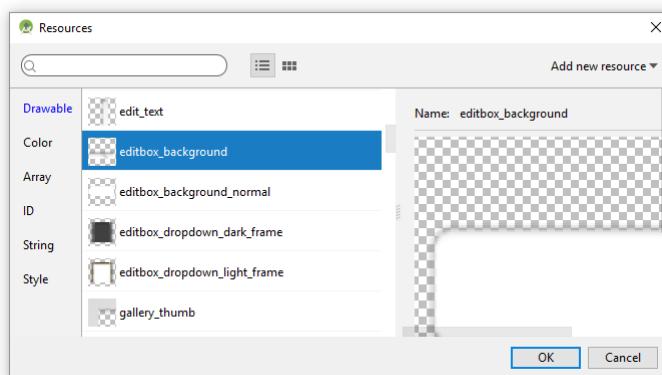


Hätten wir bei den Texteingabefeldern statt des speziellen Typs **Number (Signed)** den allgemeinen Typ **Plain Text** gewählt, könnten bzw. müssten wir nun mit dem Wert **numberSigned** für das Attribut **inputType** dafür sorgen, dass die Benutzer unserer App nur ganze Zahlen mit Vorzeichen eingeben können.

Wir verschaffen den Texteingabefeldern eine attraktive Optik durch ein vom Android-System spendiertes Hintergrundbild. Dazu markieren wir die beiden Steuerelemente, klicken im **Attributes**-Fenster auf die Zelle zur Eigenschaft **background** und dann auf den dort erscheinenden Erweiterungsschalter , sodass im folgenden Fenster **Resources** nach Wahl der Kategorie **Drawable** aus der Liste **android**



die **editbox\_background** gewählt werden kann:

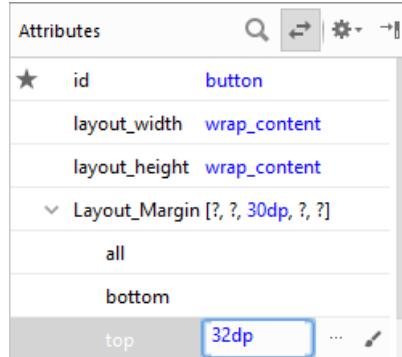


Komplettiert wird unsere Bedienoberfläche durch einen Schalter, mit dem das Kürzen des Bruchs ausgelöst werden soll. Befördern Sie aus dem Paletten-Segment **Buttons** ein Steuerelement vom Typ **Button** an den unteren Rand des Layouts. Ersetzen Sie per **Attributes**-Fenster den Vorgabetext **Button**, z. B. durch **Reduce**, obwohl damit ein Text im XML-Code landet, was die Internationalisierung erschwert.

Als **id** für den Schalter behalten wir die Vorgabe **button** bei.

Ersetzen Sie als Wert der Eigenschaft **layout\_width** die Vorgabe **match\_parent** durch **match\_content**.

Fügen Sie einen oberen Rand ein, um den Schalter von den anderen Bedienelementen abzusetzen:



Um zu verhindern, dass die Schalterbeschriftung komplett in Großschreibung erscheint, muss die Schaltereigenschaft **textAllCaps** auf den Wert **false** gesetzt werden. Per **Attributes**-Fenster ist mir das im Android Studio nicht gelungen. In der **Text**-Ansicht des Editors ist das Ziel problemlos zu realisieren:

```
<Button
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="32dp"
    android:text="Reduce"
    android:textAllCaps="false"/>
```

Damit ist das zu Beginn des aktuellen Abschnitts beschriebene Designziel erreicht.

### 4.3 Click-Behandlungsmethode für den Schalter erstellen

Wir machen uns jetzt daran, eine Java-Methode zu verfassen, die ausgeführt werden soll, wenn der Benutzer den Schalter in unserer App betätigt. Wechseln Sie in der Editorzone zum Quellcode in der Datei **MainActivity.java**. Wenn später ein Benutzer unsere App startet, legt Android ein Objekt der Klasse **MainActivity** an und beauftragt es, die vom Projektassistenten angelegte Methode **onCreate()** auszuführen:

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
}
```

Über die folgende Anweisung wird dafür gesorgt, dass die Hierarchie der GUI-Objekte, die wir mit Hilfe des grafischen GUI-Designers (siehe Abschnitt 4.2) in der Datei **activity\_main.xml** deklariert haben, erzeugt und angezeigt wird:

```
setContentView(R.layout.activity_main);
```

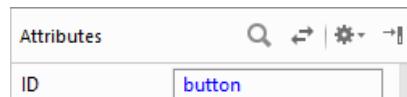
Nun müssen wir den Quellcode der Methode **onCreate()** noch etwas erweitern, damit unser Programm den Erwartungen der Anwender gerecht wird. Zunächst registrieren wir das beim Programmstart erzeugte Objekt der Klasse **MainActivity** als zuständig für die Behandlung von

Klick-Ereignissen beim Befehlsschalter. Tragen Sie am Ende der Methode `onCreate()` eine Anweisung ein, die es ermöglicht, den Schalter anzusprechen:<sup>1</sup>

```
Button button = findViewById(R.id.button);
```

In dieser Zeile stecken einige Lerninhalte zu Java und zur speziellen Android-Umgebung, die uns in den nächsten Kapiteln beschäftigen werden:

- Wir definieren eine Variable namens `button` vom Typ der Steuerelementklasse **Button**.
- Das aktiv handelnde Objekt der Klasse **MainActivity** beherrscht die Methode `findViewById()`, die eine Referenz zum Schalter liefert und dazu als Argument (Parameter) die Kennung des Schalters benötigt (eine ganze Zahl). Wir kennen aber nur die per Layout-Editor vereinbarte **ID**:

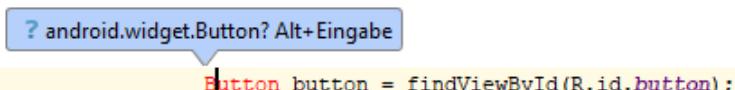


Bei der Übersetzung der **ID** in eine Objektreferenz hilft eine Klasse namens `R`, die Bestandteil der Anwendung ist und von der Entwicklungsumgebung automatisch generiert wird. Sie besitzt eine interne Klasse namens `id` mit statischen Feldern für die Steuerelemente. Das Feld zum Befehlsschalter ist über `R.id.button` ansprechbar und enthält die benötigte Zahl. Weil die Felder in der Klasse `R.id` über den Modifikator `final` gegen Schreibzugriffe geschützt sind, kann hier übrigens auf eine Datenkapselung verzichtet werden.

Das Android Studio kritisiert die neue Anweisung teilweise durch rote Farbe,

```
Button button = findViewById(R.id.button);
```

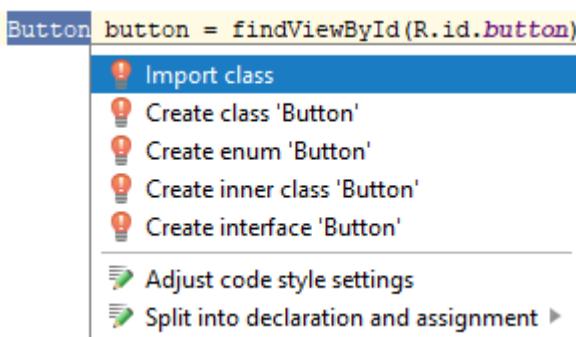
weil es den Bezeichner **Button** nicht auflösen kann. Wenn Sie per Mausklick die Einfügemarkie auf den unbekannten Bezeichner setzen und den Mauszeiger eine kurze Weile über dem Bezeichner verharren lassen, fragt das Android Studio nach, ob die Klasse **Button** im Paket **android.widget** gemeint ist:



Wenn Sie dies mit der Tastenkombination **Alt+Eingabe** bestätigen, dann fügt das Studio eine Import-Anweisung für die Klasse **android.widget.Button** in die Quellcodedatei ein, und das Problem ist behoben:

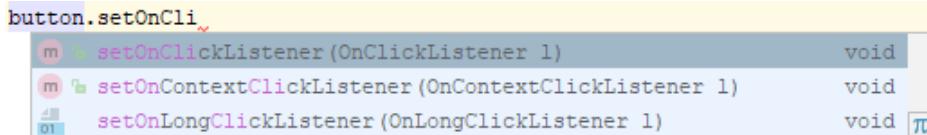
```
import android.widget.Button;
```

Wenn Sie einen kritisierten Bezeichner markieren und dann die Tastenkombination **Alt+Eingabe** drücken, können Sie eventuell zwischen verschiedenen Lösungsvorschlägen wählen, z. B.:

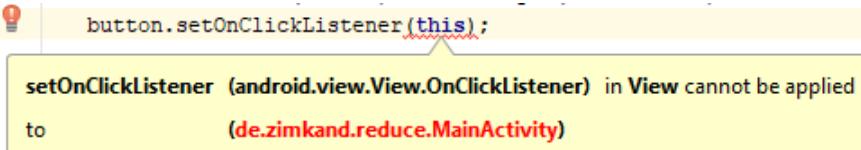


<sup>1</sup> Weil diese Anweisung die Existenz des **Button**-Objekts voraussetzt, muss sie vor allem *hinter* dem Aufruf der Methode `setContentView()` stehen, der das Erstellen der GUI-Objekte veranlasst.

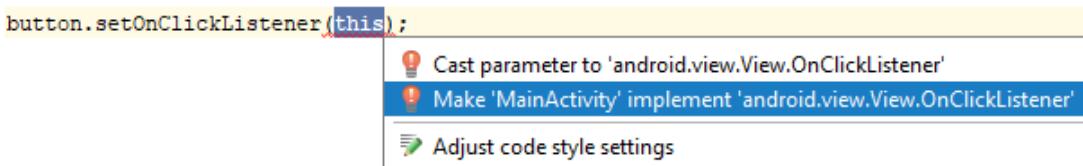
Jetzt verwenden wir die gewonnene Referenz und bitten den Schalter durch Aufruf seiner Methode `setOnClickListener()`, die handelnde Instanz unserer Aktivität (bezeichnet durch das Schlüsselwort `this`) als Klick-Interessenten einzutragen. Weil das Android Studio intelligente Vorschläge zur Code-Vervollständigung macht, kann man sich einige Arbeit und Tippfehler ersparen:



Wir bleiben trotz korrekter Erweiterung der Methode `onCreate()` nicht ohne Kritik,



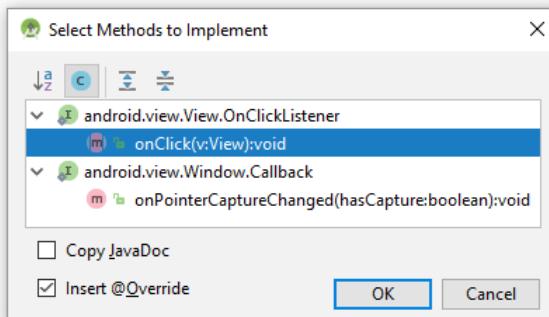
weil die Registrierung zur Benachrichtigung über Klickereignisse nur erlaubt ist für Objekte einer Klasse, die zusichert, alle im Interface `OnClickListener` geforderten Methoden zu beherrschen. Unter den mit **Alt+Eingabe** angeforderten Verbesserungsvorschlägen findet sich an zweiter Stelle die benötigte Änderung für die Kopfzeile der `MainActivity`-Klassendefinition:



Das Studio erweitert die Kopfzeile

```
public class MainActivity extends AppCompatActivity implements View.OnClickListener {
```

und fragt nach, welche Interface-Methoden implementiert werden sollen. Im konkreten Fall ist die Liste kurz, weil das Interface `OnClickListener` lediglich die Methode `onClick()` verlangt:



Nach einer Bestätigung mit **OK** erstellt das Android Studio den folgenden Methodenrohling:

```
@Override
public void onClick(View v) {
}
```

Nun ist dafür gesorgt, dass der Befehlsschalter bei jedem Klick eine Botschaft an das Aktivitätsobjekt schickt, indem er dessen Methode `onClick()` aufruft. Weil die Methode `onClick()` auf Benachrichtigungen sehr passiv reagiert, ist das Programm allerdings noch nutzlos. Wir nehmen die letzte Etappe auf dem Weg zu unserer ersten nützlichen Android-App in Angriff und sorgen für ein sinnvolles Verhalten der Methode `onClick()`:

- Zunächst deklarieren wir Referenzvariablen vom Typ **EditText**, um die Texteingabefelder für den Zähler und den Nenner bequem ansprechen zu können. Dabei gehen wir genauso vor wie oben beim Befehlsschalter:

```
EditText etZahler = findViewById(R.id.zahler);
EditText etNenner = findViewById(R.id.nenner);
```

Wie man das Android Studio auffordert, die Importanweisung für die Klasse **EditText** zu erstellen, wissen Sie mittlerweile.

- Wir deklarieren zwei Variablen vom primitiven Typ **int** und befördern die numerisch interpretierten Inhalte der Texteingabefelder dorthin:

```
int z = Integer.parseInt(etZahler.getText().toString());
int n = Integer.parseInt(etNenner.getText().toString());
```

In der ersten Anweisung wird das **EditText**-Objekt **etZahler** über die Methode **getText()** gebeten, seinen Inhalt abzuliefern. Dabei handelt es sich um ein Objekt vom Typ **Editable**, an das die Botschaft **toString()** gerichtet wird, um eine Text-Repräsentation zu erhalten. Diese kann durch die Methode **parseInt()**, ausgeführt von der Klasse **Integer**, zu einer ganzen Zahl verarbeitet werden, die schließlich der Variablen **z** zugewiesen wird.

- Nun kommt der Auftritt von Herrn Euklid, der einen Algorithmus zur Bestimmung des größten gemeinsamen Teilers (ggT) von zwei ganzen Zahlen entwickelt hat. Indem man den Zähler und den Nenner durch den ggT teilt, erhält man schließlich den optimal gekürzten Bruch:

```
if (z*n != 0) {
    int rest;
    int ggt = Math.abs(z);
    int divisor = Math.abs(n);
    do {
        rest = ggt % divisor;
        ggt = divisor;
        divisor = rest;
    } while (rest > 0);
    z /= ggt;
    n /= ggt;
}
```

Eine Erläuterung zu diesem Kürzungsverfahren finden Sie z. B. in Baltes-Götz & Götz (2018, S. 163).

- Abschließend werden die gekürzten Zahlen von der **Integer**-Methode **toString()** wieder in Zeichenfolgen verwandelt und per **setText()** - Aufruf an die **EditText**-Steuerelemente übergeben:

```
etZahler.setText(Integer.toString(z));
etNenner.setText(Integer.toString(n));
```

Nun ist die Methode **onClick()** komplett, aber noch nicht perfekt, wie sich später herausstellen wird:

```

public void onClick(View v) {
    EditText etZaehler = findViewById(R.id.zaeher);
    EditText etNenner = findViewById(R.id.nenner);
    int z = Integer.parseInt(etZaehler.getText().toString());
    int n = Integer.parseInt(etNenner.getText().toString());

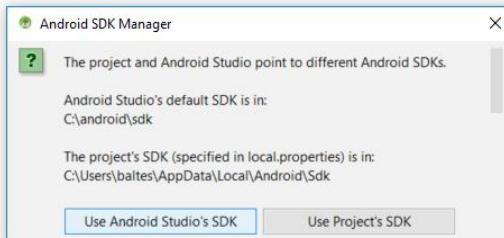
    // größten gemeinsamen Teiler mit dem Euklidischen Algorithmus bestimmen
    if (z*n != 0) {
        int rest;
        int ggt = Math.abs(z);
        int divisor = Math.abs(n);
        do {
            rest = ggt % divisor;
            ggt = divisor;
            divisor = rest;
        } while (rest > 0);
        z /= ggt;
        n /= ggt;
        etZaehler.setText(Integer.toString(z));
        etNenner.setText(Integer.toString(n));
    }
}

```

Das komplette Projekt Reduce auf dem aktuellen Entwicklungsstand ist im folgenden Ordner zu finden

**...\\BspUeb\\Reduce\\Skript**

Beim Öffnen eines kopierten Übungsprojekts erscheint auf Ihrem Rechner eventuell eine Fehlermeldung wegen eines abweichenden Android SDK - Installationsordners, z. B.:

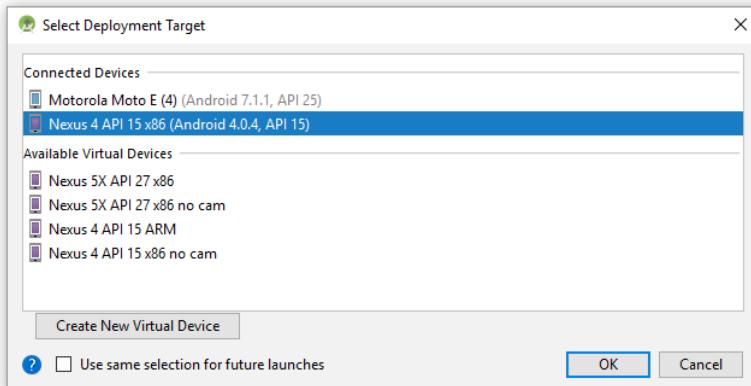


Nach einem Klick auf den Schalter **Use Android Studio SDK** ist das Problem gelöst.

Sollte die Android SDK - Installation auf Ihrem Rechner nicht die Android-Version 8.1 (Oreo, API-Level 27) enthalten, ist das Problem etwas größer (siehe Abschnitt 4.10).

#### 4.4 Programm ausführen

Starten Sie die Reduce-App per Mausklick auf den Schalter , mit dem Menübefehl **Run > Run 'app'** oder mit der Tastenkombination + **F10**. Nachdem die Übersetzung und sonstige Aktivitäten zum Erstellen der App abgeschlossen sind, fragt das Android Studio nach, welches Android-Gerät genutzt oder gestartet werden soll, z. B.:

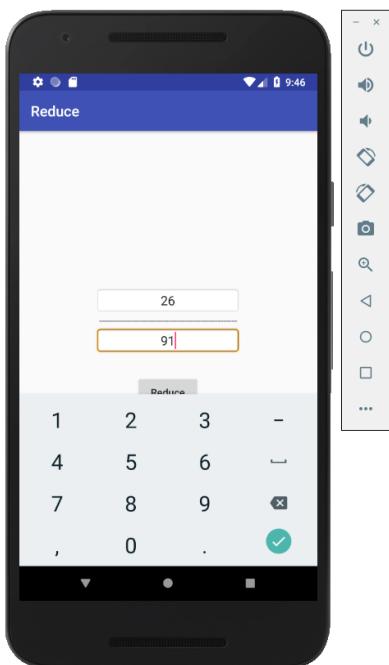


Bei einem virtuellen Gerät erspart man sich durch eine bescheidene Variante mit nicht allzu aktueller Android-Version und relativ kleiner RAM-Ausstattung ...

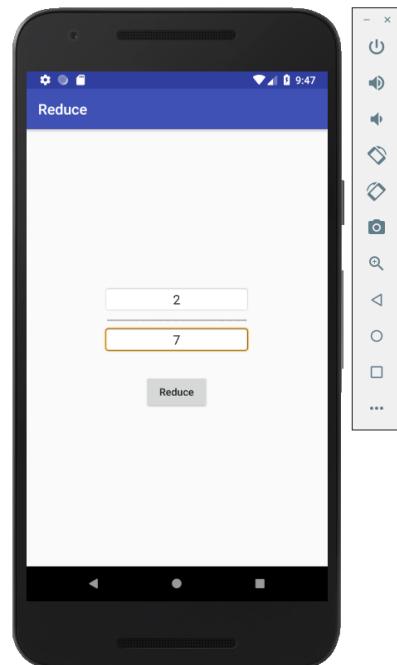
- eine lange Wartezeit beim Starten des virtuellen Geräts
- eine Behinderung anderer Anwendungen durch Speichermangel auf dem Entwicklungsrechner.

Schließlich erleben wir eine wenig imposante, aber doch eindeutig nützliche Anwendung:

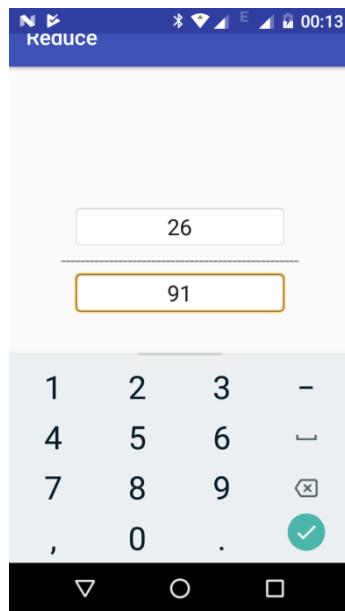
Unangenehme Aufgaben ...



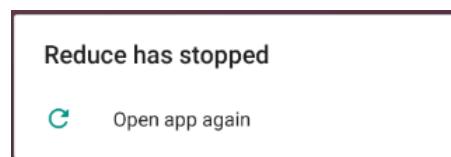
sind nun leicht zu lösen:



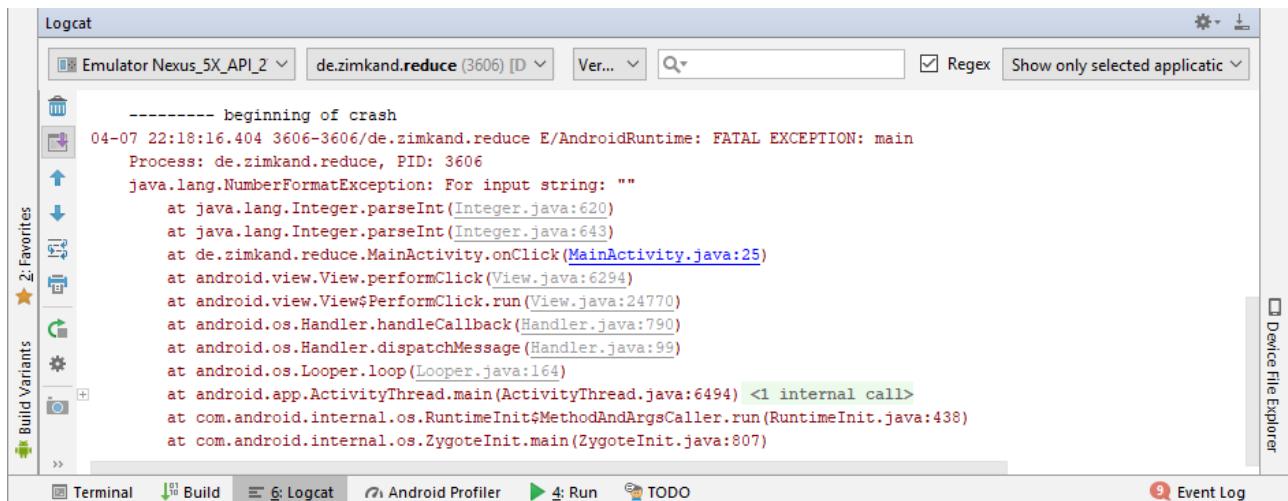
Selbstverständlich lässt sich die App auch auf ein im Debug-Modus am Entwicklungs-PC angeschlossenes Android-Gerät übertragen (vgl. Abschnitt 3.5):



Fehlt beim Kürzen der Zähler oder Nenner, wird das Programm von Android abgebrochen:



Das Android-Studio präsentiert dann im **Logcat**-Fenster die Aufrufersequenz (engl.: *stack trace*) mit den am Unfall beteiligten Methoden:



Ein Klick auf den blau markierten Link führt zu der von uns zu verantwortenden Quellcodezeile.

Um den Unfall zu verhindern, kann man z. B. in der Methode `onClick()` die „optimistischen“ Anweisungen zum Konvertieren von hoffentlich vorhandenen Textfeldinhalten in ganze Zahlen

```
int z = Integer.parseInt(etZaehler.getText().toString());
int n = Integer.parseInt(etNenner.getText().toString());
```

durch eine Alternative mit Überprüfung der Voraussetzungen ersetzen:

```

String sz = etZaehler.getText().toString();
String sn = etNenner.getText().toString();
if (sz.length() == 0 || sn.length() == 0)
    return;
int z = Integer.parseInt(sz);
int n = Integer.parseInt(sn);

```

Es wird kontrolliert, ob die von den Texteingabefeldern angelieferten Zeichenfolgen tatsächlich eine Länge größer als 0 besitzen. Bei vorhandenen Benutzereingaben *muss* es sich um ganze Zahlen handeln, weil wir im GUI-Designer **numberSigned** als **inputType** verwendet haben wie ein Blick in die XML-Datei **activity\_main.xml** bestätigt:

```

<EditText
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:inputType="numberSigned"
    ...
/>

```

## 4.5 Sichern und Wiederherstellen

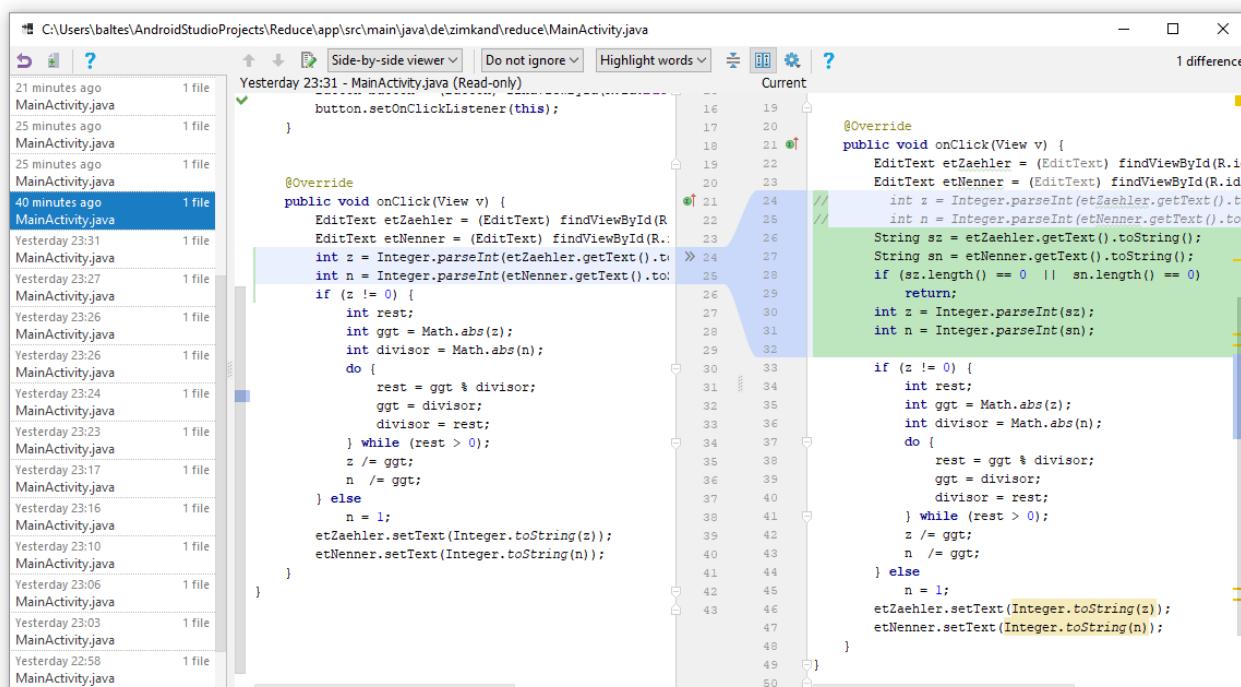
Bei der Arbeit mit dem Android Studio muss man sich um das Sichern von Quellcode oder Resourcen kaum Gedanken machen, weil die Entwicklungsumgebung bei jeder passenden Gelegenheit (z. B. beim Übersetzen) automatisch sichert.

Außerdem ist ein **VCS** (Version Control System) integriert, das lokal arbeiten und mit Cloud-Diensten wie **GitHub** kooperieren kann. Wegen der zahlreichen Funktionen ist ein eigenes Hauptmenü namens **VCS** vorhanden.

Für eine Java- oder XML-Datei sind über

### VCS > Local History > Show History

zahlreiche vorherige Zustände verfügbar, z. B.:



Um zu einem vorherigen Zustand zurück zu kehren, wählt man aus seinem Kontextmenü das Item **Revert**.

Die hinter dem Android Studio stehende Firma JetBrains weist allerdings darauf hin, dass die lokale Historie keine ausgereifte Versionskontrolle ersetzt, denn:

- Die lokale Historie wird gelöscht, wenn eine neue Version des Android Studios installiert wird, oder wenn der Benutzer zur Beseitigung von Problemen den System-Cache der Entwicklungsumgebung löscht (mit dem Menübefehl **File > Invalidate Caches**).
- Weil die maximale Verweilzeit und die Größe der Historie beschränkt sind, existieren dort abgelegte Zwischenstände keinesfalls unbegrenzt.

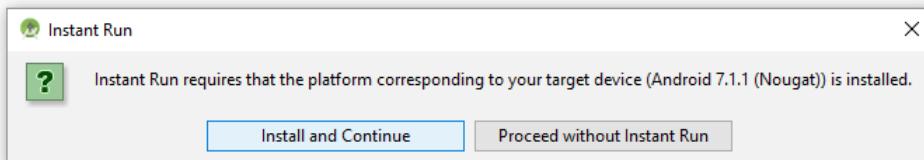
## 4.6 Instant Run

Die **Instant Run** - Technik im Android Studio macht es möglich, nach der regulären Installation einer App auf einem Android-Gerät spätere Änderungen (am Java-Code oder an den Ressourcen) in einem beschleunigten Verfahren (ohne erneute Erstellung und Installation einer kompletten APK-Datei) mit Hilfe des Schalters in Betrieb zu nehmen. Es werden nur die tatsächlich geänderten Klassen oder Ressourcen übertragen. Wenn sich die Manifestdatei nicht geändert hat, ist keine Neuinstallation der App erforderlich, und in günstigen Fällen muss die App nicht einmal neu gestartet werden.<sup>1</sup>

Instant Run setzt folgende Android-Versionen voraus:

- In der Projektdefinition muss als **minSdkVersion** mindestens API-Level 15 angegeben werden.
- Das ausführende Gerät muss mindestens mit API-Level 21 laufen.

Außerdem muss die Android-Version des ausführenden Gerätes auf dem Entwicklungsrechner als SDK-Plattform vorhanden sein. Nötigenfalls werden Sie aufgefordert, einer Erweiterungsinstallationszustimmung zuzustimmen, z. B.:

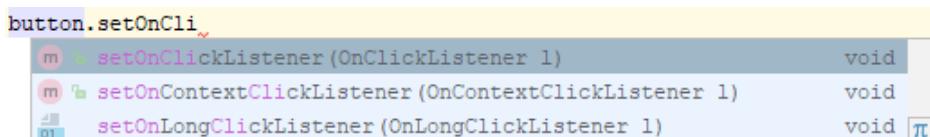


## 4.7 Arbeitshilfen im Android Studio

In diesem Abschnitt werden einige besonders hilfreiche Funktionen im Android Studio beschrieben, die vor allem den Quellcode-Editor betreffen.

### 4.7.1 Syntaxvervollständigung

In Abschnitt 4.3 haben wir beim Erstellen einer Ereignisbehandlungsmethode die Syntaxvervollständigung im Android Studio kennengelernt. Sobald man z. B. einen Punkt hinter eine Klasse oder hinter eine Referenzvariable setzt, erscheint eine Liste mit allen zulässigen Fortsetzungen. Weitere Eingaben reduzieren die Liste, und schließlich ist die passende Vervollständigung leicht per **Enter**-Taste zu wählen:



<sup>1</sup> Details zu Instant Run sind hier zu finden: <https://developer.android.com/studio/run/index.html#instant-run>

Die Syntaxvervollständigung macht Vorschläge für Variablen, Typen, Methoden usw. Sollte sie nicht spontan tätig werden, kann sie mit der folgenden Tastenkombination angefordert werden:

**Strg + Leertaste**

Nach einer Anforderung mit

**Strg + Umschalt + Leertaste**

erscheint eine *smarte* Variante der Syntaxvervollständigung. Indem der erwartete Typ des ganzen Ausdrucks analysiert wird, können Variablen und Methoden kontextabhängig vorgeschlagen werden.

Soll mit Hilfe der Syntaxvervollständigung eine Anweisung nicht erweitert, sondern geändert werden, dann sollte ein Vorschlag nicht per **Enter**-Taste oder Doppelklick, sondern per Tabulatortaste ( ) übernommen werden. Auf diese Weise wird z. B. ein Methodenname *ersetzt*, in dem sich die die Einfügemarke gerade befindet, statt durch Einfügen des neuen Namens ein fehlerhaftes Gebilde zu erzeugen.

Mit der Tastenkombination

**Umschalt + Strg + Enter**

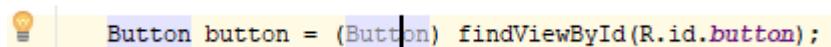
fordert man die Vervollständigung einer Anweisung an, z. B.:

vorher	nachher
button.setOnClickListener(this);	button.setOnClickListener(this);

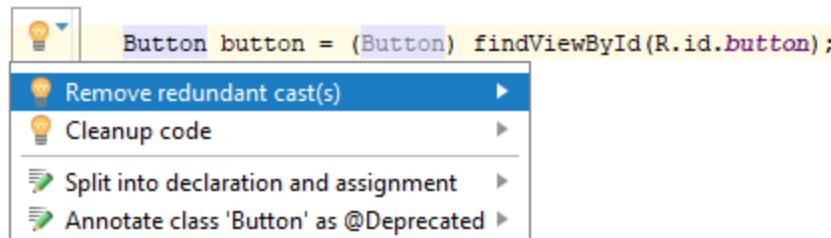
#### 4.7.2 Code-Inspektion und QuickFixes

Das Android Studio führt Code-Inspektionen *on the fly* durch, macht auf (potentielle) Probleme oder Verbesserungsmöglichkeiten aufmerksam und schlägt QuickFix-Korrekturen vor.

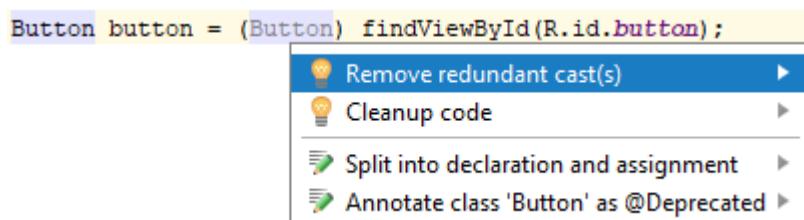
Wenn IntelliJ IDEA eine Code-Änderung vorschlagen möchte, wird das betroffene Element hervorgehoben, und sobald sich die Einfügemarke im Element befindet, erscheint eine gelbe Birne links neben der betroffenen Stelle, z. B.:



Zeigt die Maus auf die Birne, kann ein Drop-Down - Menü mit Korrekturvorschlägen geöffnet werden, z. B.:

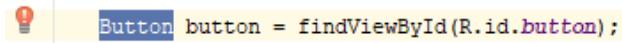


Statt das Drop-Down - Menü zur gelben Birne zu öffnen, kann man auch die Einfügemarke auf das hervorgehobene Element setzen und die Tastenkombination **Alt + Enter** betätigen, um dieselbe Vorschlagsliste zu erhalten:

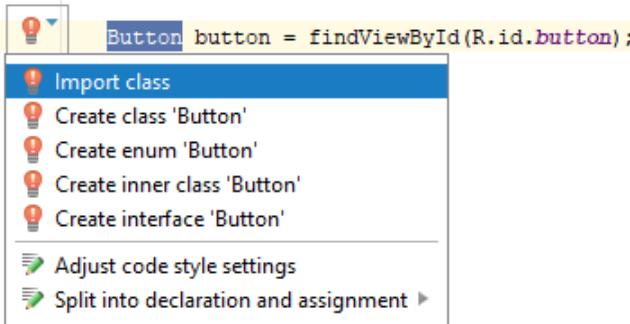


Im Beispiel kann dank erweiterter Compiler-Intelligenz auf die explizite Typumwandlung verzichtet werden.

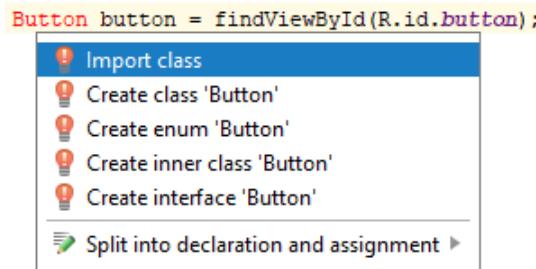
Wird ein vom Android Studio entdeckter *Fehler* markiert, erscheint eine rote Birne links neben der betroffenen Stelle, z. B.:



Zeigt die Maus auf die Birne, kann ein Drop-Down - Menü mit Korrekturvorschlägen geöffnet werden, z. B.:



Statt das Drop-Down - Menü zur roten Birne zu öffnen, kann man auch die Einfügemarke auf den markierten Syntaxbestandteil setzen und die Tastenkombination **Alt + Enter** betätigen, um dieselbe Vorschlagsliste zu erhalten:



Im Beispiel muss die Klasse **Button** entweder mit voll qualifiziertem Namen angesprochen oder in die Quellcodedatei importiert werden:

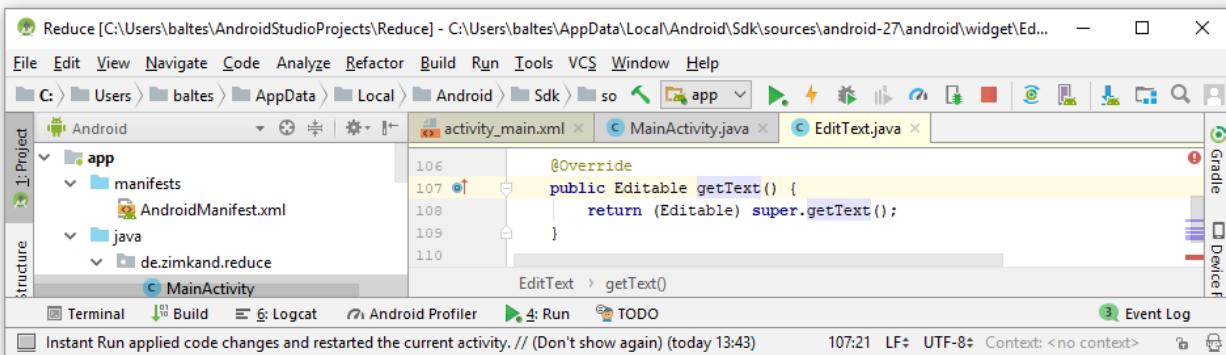
```
import android.widget.Button;
```

### 4.7.3 Orientierungshilfen

Zeigt man bei gedrückter **Strg**-Taste mit dem Mauszeiger auf eine Methode, dann erscheint der Definitionskopf, z. B.:



Setzt man bei gedrückter **Strg**-Taste einen Mausklick auf einen Bezeichner (z. B. Klasse, Variable, Methode), dann springt das Android Studio zur Implementierung des angefragten API-Bestandteils. Nötigenfalls wird der Quellcode der zugehörigen Klasse in ein neues Registerblatt des Editors geladen, z. B.:



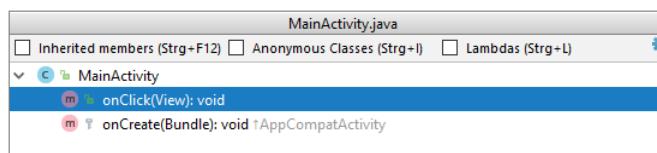
Zum selben Ziel kommt man auch ohne Mausbeteiligung:

- Einfügemarken auf den interessierenden Bezeichner setzen
- Tastenkombination **Strg + B**

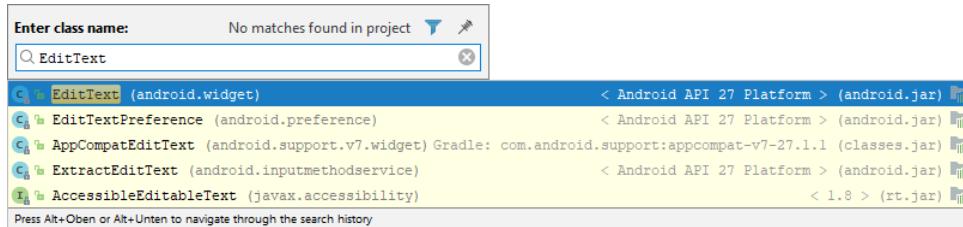
Über die Tastenkombination **Strg + F12** oder mit dem Menübefehl

#### Navigate > File Structure

erhält man einen Dialog mit der Struktur der aktuell im Editor bearbeiteten Datei und kann einen Bestandteil per Mausklick ansteuern:



Um den Quellcode einer beliebigen Klasse anzufordern, trägt man ihren Namen nach der Tastenkombination **Strg + N** in den folgenden Dialog ein:



#### 4.7.4 Refaktorieren

Um z. B. einen Variablen- oder Klassennamen an allen Auftrittsstellen im Projekt zu ändern, setzt man die Einfügemarken auf ein Vorkommen, drückt die Tastenkombination **Umschalt + F6**, ändert die Bezeichnung und quittiert mit der Eingabetaste. Im Menüsysteem ist die benutzte Refaktorfunktion hier zu finden:

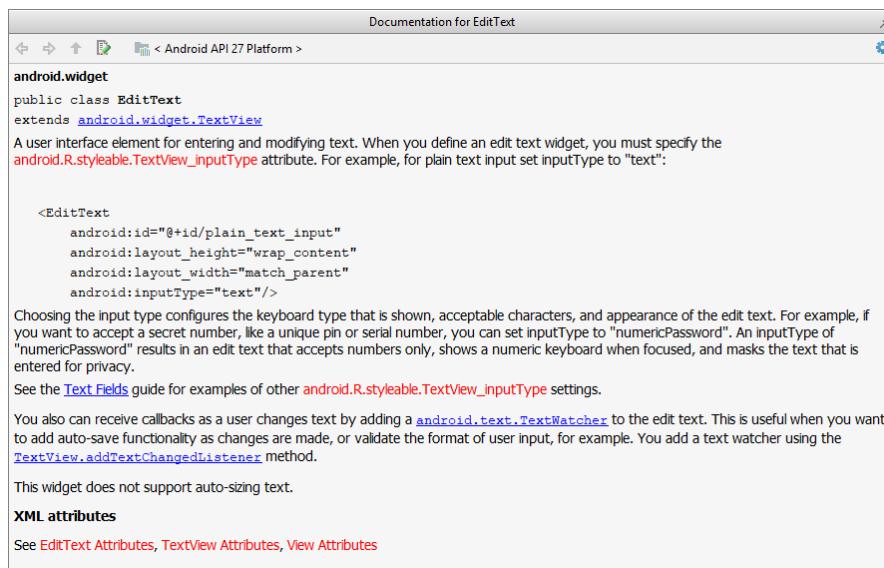
#### Refactor > Rename

Die im **Refactor**-Menü zahlreich vorhandenen weiteren AS-Kompetenzen zur Quellcode-Umgestaltung werden wir im Kurs nicht benötigen.

#### 4.7.5 Javadoc-Information abrufen

Setzt man die Einfügemarken auf ein API-Element (z. B. Klasse oder Methode) und drückt die Tastenkombination **Strg + Q** oder wählt den Menübefehl **View > Quick Documentation**, dann erhält man ein PopUp-Fenster mit der Javadoc-Information zum angefragten Element. Das Fenster

verschwindet bei Fortsetzung der Editorarbeit, sofern es nicht über den Pin-Schalter  fixiert wird, z. B.:



Leider ist es nicht möglich, die externe Dokumentation zu einem API-Element von der Webseite

<https://developer.android.com/reference/packages.html>

über die Tastenkombination **Umschalt + F1** abzurufen.<sup>1</sup> In der IntelliJ-Entwicklungsumgebung für Java klappt die analoge Funktion.

## 4.7.6 Sonstige Hinweise

### 4.7.6.1 Änderungen zurücknehmen

Quellcodeeditor lassen sich die letzten Änderungen zurücknehmen mit der von vielen Programmen gewohnten Tastenkombination **Strg + Z**. Soll eine zurückgenommenen Änderung wiederhergestellt werden, ist die folgende Tastenkombination zu verwenden:

**Strg + Umschalt + Z**

Wer in dieser Situation die (z. B. aus Microsoft Office) gewohnte Tastenkombination **Strg + Y** verwendet, wird vom Löschen der gesamten Zeile überrascht, das natürlich mit **Strg + Z** revidiert werden kann.

### Kommentarstatus für einen Zeilenblock setzen bzw. aufheben

Um einen markierten Zeilenblock als Kommentar zu deklarieren bzw. diese Deklaration aufzuheben, drückt man die **Strg**-Taste zusammen mit der Divisionstaste im numerischen Ziffernblock:

Strg + ÷

Die offiziell beschriebene Tastenkombination klappt nur mit einem US-Tastaturlayout.

<sup>1</sup> Die Funktion ist vorhanden, wenn man per SDK-Manager auf der Registerkarte **SDK Tools** die Option **Documentation for Android SDK** nachinstalliert. Dabei muss man allerdings 1 GB lokalen Speicherplatz aufbringen und erhält vor allem nur die Dokumentation zum API-Level 24 (im April 2018 ist das API-Level 27 aktuell).

#### 4.7.6.2 Android-Bedienoberfläche zurücksetzen

Aufgrund der zahlreichen Möglichkeiten, die AS-Bedienoberfläche zu gestalten, kommt vielleicht irgendwann der Wunsch auf, wieder zur Voreinstellung zurückzukehren, was durch den folgenden Menübefehl zu realisieren ist:

**Window > Restore Default >Layout**

Wer zuvor über

**Window > Store Current Layout as Default**

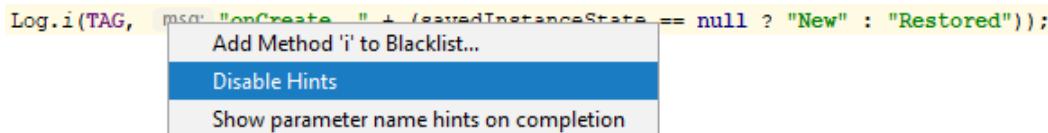
seine bevorzugte Konstellation gesichert hat, kann mit dem eben beschriebenen Menübefehl zu dem gesicherten Layout zurückkehren. Soll in dieser Situation doch der Originalzustand wiederhergestellt werden, hilft das Löschen des AS-Konfigurationsordners (siehe Abschnitt 2.4).

#### 4.7.6.3 Parameternamen im Quellcodeeditor

Das Android Studio zeigt bei manchen Methodenaufrufen für manche Parameter den Namen an, z. B.:

```
Log.i(TAG, msg: "onCreate, " + (savedInstanceState == null ? "New" : "Restored"));
```

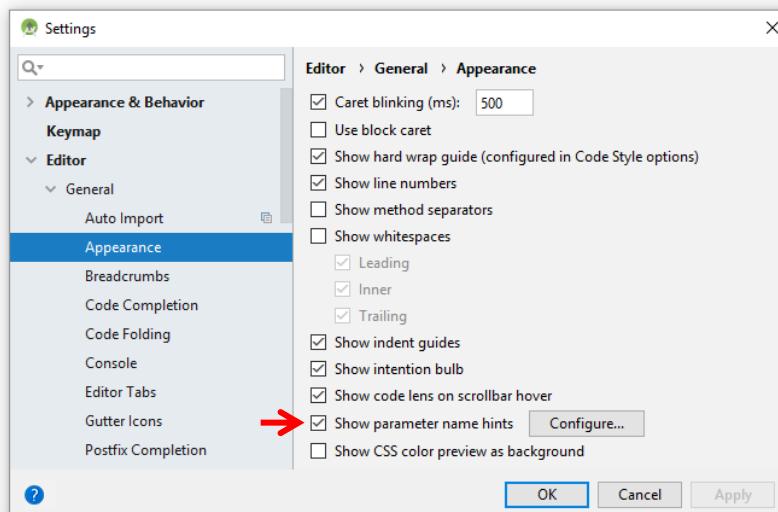
Solche Hinweise können per Kontextmenü entfernt



oder nach dem folgenden Menübefehl

**File > Settings > Editor > General > Appearance**

generell abgeschaltet werden:



#### 4.8 Eigene Apps auf einem beliebigen Android-Gerät installieren

Bei der bisher beschriebenen Erstellung einer App im **Debug**-Modus resultiert schlussendlich eine Datei namens **app-debug.apk** in einem Ordner, der vom Instant Run - Einsatz abhängt:

- Ablage bei Erstellung *ohne* Instant Run:

...|app|build|outputs|apk|debug

- Ablage bei Erstellung *mit* Instant Run:

...|app|build|intermediates|instant-run-apk|debug

Beim Testen während der Entwicklungsarbeit wird die APK-Datei automatisch auf einem emulierten oder per USB-Kabel im Debug-Modus angeschlossenen Gerät installiert und gestartet.

Android-Apps müssen generell mit einem digitalen Zertifikat signiert sein, damit sie installiert werden können. Im Entwicklungsmodus, den wir bisher benutzt haben, und den wir auch im weiteren Kursverlauf meist verwenden werden, kommt ein Debug-Zertifikat zum Einsatz, das 365 Tage gültig ist (ab Android Studio - Installation). Es befindet sich unter Windows für einen Benutzer namens **theo** in der Datei:

**C:\Users\theo.android\debug.keystore**

Ist das Debug-Zertifikat abgelaufen, scheitert die App-Erstellung. Zur Lösung dieses Problems, muss lediglich die Datei mit dem Debug-Zertifikat gelöscht werden. Allerdings können vorhandene Apps mit dem neuen Zertifikat nicht aktualisiert, sondern nur ersetzt werden.

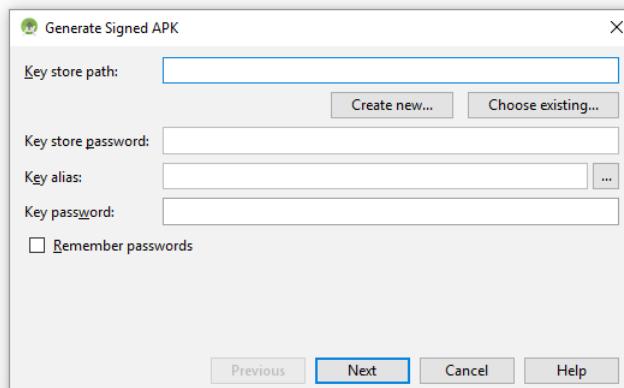
Um die Einschränkungen des Debug-Zertifikats zu vermeiden, muss eine App im **Release**-Modus erstellt und dabei mit einem Zertifikat signiert werden, dessen privater Schlüssel im Besitz des Entwicklers ist. Das zum Signieren im Release-Modus verwendete Zertifikat muss *nicht* von einer Zertifikats-Autorität (CA) signiert sein. Offenbar zieht Google die Option eines signierten Zertifikats überhaupt nicht ernsthaft in Betracht. Für eine App-Verteilung über Google Play darf die Laufzeit des zum Signieren einer App verwendeten Zertifikats nicht vor dem 22.10.2033 enden, und eine kommerzielle Zertifikats-Autorität wird eine solche Laufzeit wohl kaum anbieten.

Generell geht es beim Signieren von Android-Apps primär darum, den Autor zu identifizieren. Vom selben Autor stammende Apps können über ihre Manifest-Dateien (siehe Abschnitt 5.1) beantragen, in einer gemeinsamen Sandbox ausgeführt zu werden und sich damit z. B. wechselseitigen Dateizugriff zu erlauben (siehe Abschnitt 5.5). Solange Sie Ihren privaten Schlüssel nicht preisgeben, kann sich keine fremde App illegitime Zugriffsrechte verschaffen.

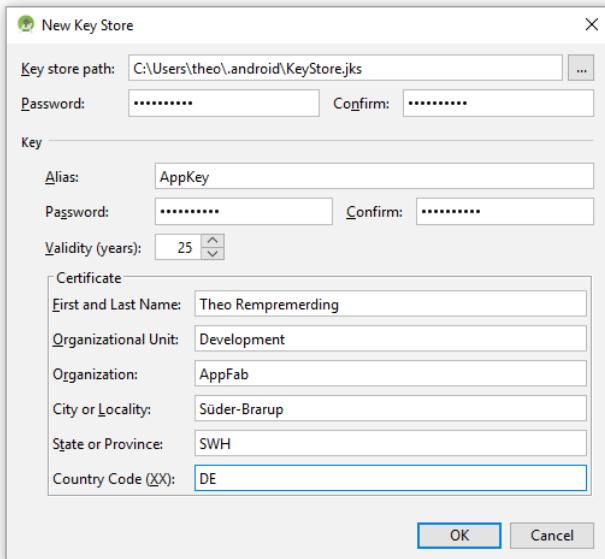
Im Android Studio startet man die Erstellung im Release-Modus mit

**Build > Generate Signed APK**

Ist noch kein privater Schlüssel vorhanden,



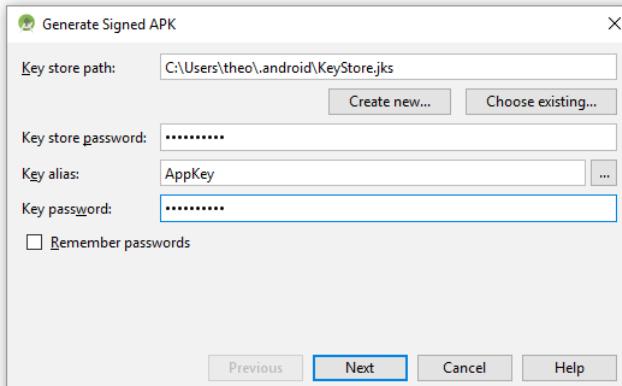
leitet man mit **Create new** die Erstellung eines Schlüsselspeichers und eines Release-Schlüssels ein:



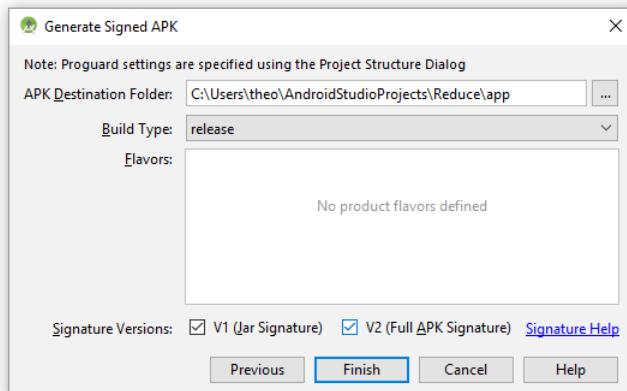
Google empfiehlt eine Laufzeit von mindestens 25 Jahren, um während der gesamten Lebenszeit einer App die Updates mit demselben Schlüssel signieren zu können. Das ermöglicht es den Benutzern, eine installierte App durch Updates bequem zu aktualisieren, statt jeweils eine neue App installieren zu müssen.

Die privaten Schlüssel unter Kontrolle zu halten, ist aus nahe liegenden Gründen enorm wichtig. Mit einem entwendeten Schlüssel kann ein böswilliger Entwickler Anwendungen erstellen, welche die Sicherheit von mit demselben Schlüssel signierten Anwendungen und damit die Sicherheit der Anwender aushebeln. Eine App des rechtmäßigen Schlüsselinhabers kann z. B. über ein angebliches Update durch eine böswillige Alternative ersetzt werden.

Der neue Schlüssel wird zum Signieren der App benutzt:



Abschließend kann der Ausgabeordner zur Kenntnis genommen bzw. bestimmt werden:

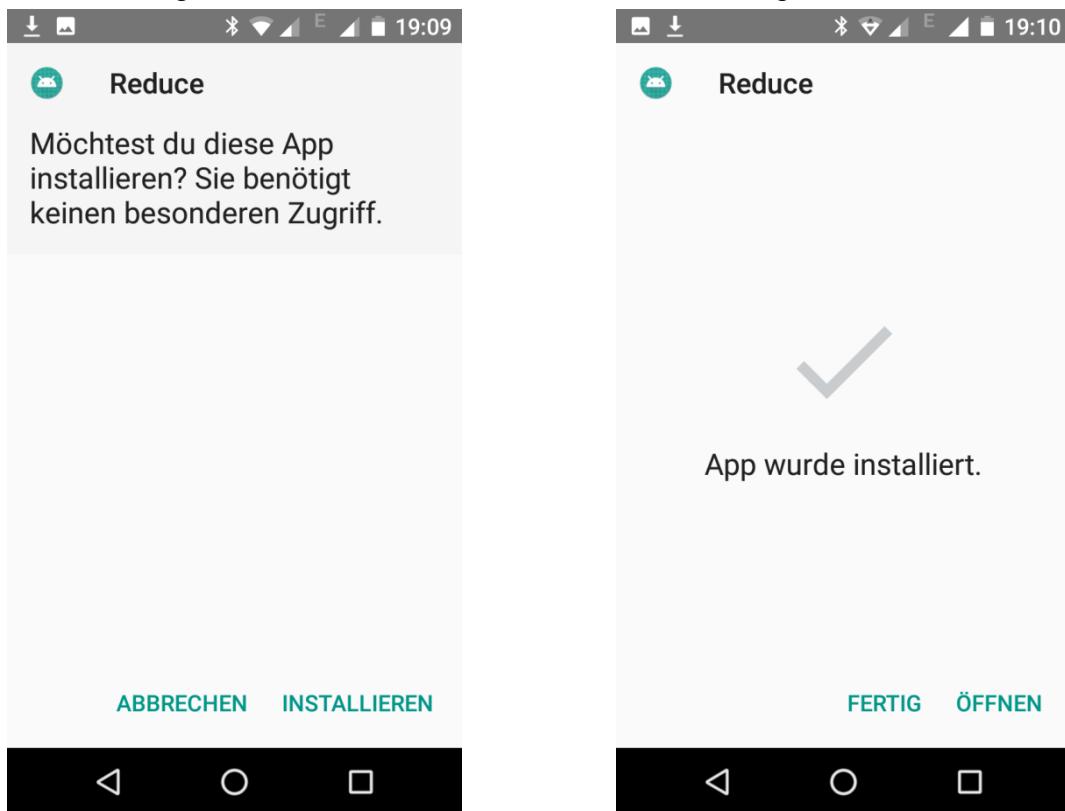


Mit Android 7 wurde eine neue Signierungstechnik mit verstärktem Schutz gegen die nicht-autorisierte Veränderung einer **apk**-Datei eingeführt. Sofern bei der Erstellung einer App keine Probleme auftreten, spricht nichts dagegen, beide Signierungsverfahren zu aktivieren.

Eine im Release-Modus erstellte App landet per Voreinstellung in der Datei  
**...\\app\\release\\app-release.apk**

Um eine eigene App auf einem Android-Gerät eines Interessenten zu installieren, das nicht mit dem Entwicklungsrechner verbunden ist, muss man nicht den Weg über Google Play wählen. Eine kleine Lösung sieht so aus:

- Die im Release-Modus erstellte **apk**-Datei per Mail-Anhang an den Interessenten schicken
- Auf dem Zielgerät **ausnahmsweise und vorübergehend** das Installieren von Software aus unbekannter Quelle erlauben (**Einstellungen > Sicherheit > Unbekannte Herkunft**)
- Mail auf dem Zielgerät abrufen und öffnen, Klick auf den Anhang:



- Auf dem Zielgerät das Installieren von Software aus unbekannter Quelle wieder verbieten

Weitere Details zum Signieren von Android-Apps finden sich z. B. im Google-Webangebot für Android-Entwickler:

<https://developer.android.com/studio/publish/app-signing>

## 4.9 Projektkonfiguration einsehen und ändern

Wichtige Eigenschaften einer App befinden sich in der Modul-Variante der Gradle-Konfigurationsdatei **build.gradle**, z. B. im Attribut **minSdkVersion** die minimal vorausgesetzte Android-Version, die im Assistentendialog zur Anwendungserstellung (vgl. Abschnitt 4.1) angegeben wurde:

```

apply plugin: 'com.android.application'

android {
    compileSdkVersion 27
    defaultConfig {
        applicationId "de.zimkand.reduce"
        minSdkVersion 15
        targetSdkVersion 27
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
    }
}

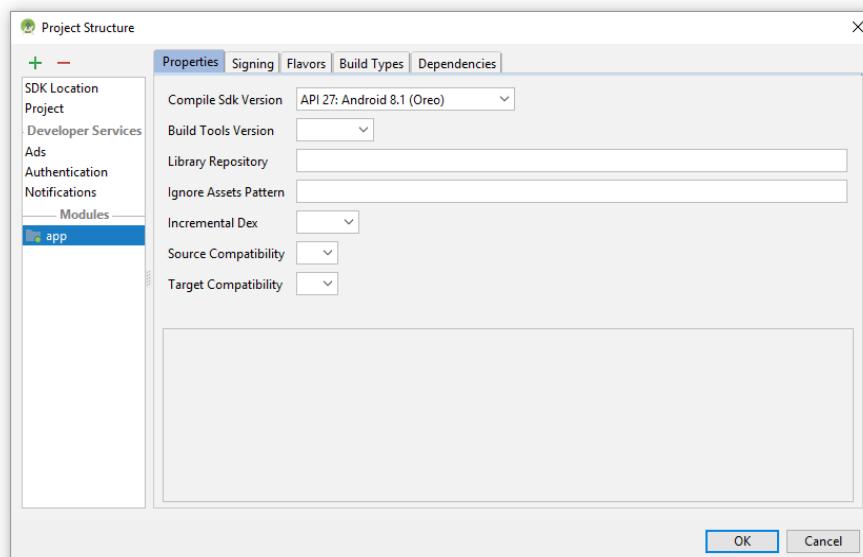
android{} > defaultConfig{}

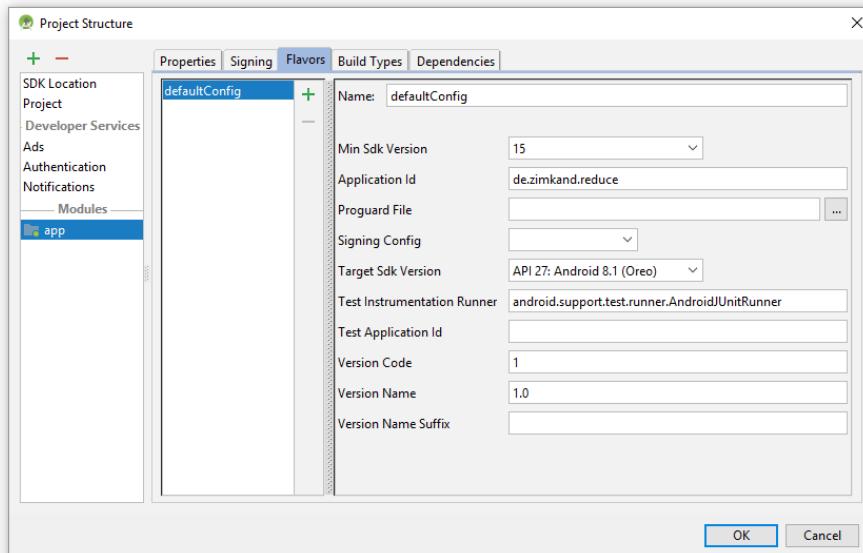
```

Statt die Datei **build.gradle** direkt zu editieren, kann man die Einstellungen nach

**File > Project Structure > app**

auch im folgenden Dialog vornehmen:





Für die Kompatibilität einer App sind drei Einstellungen von großer Bedeutung:

- **compileSdkVersion**

Diese Einstellung informiert das Build-System Gradle darüber, mit welcher SDK-Version eine App kompiliert werden muss. Diese SDK-Version muss natürlich auf dem Entwicklungsrechner installiert sein. Man verwendet in der Regel die neueste SDK-Version.

Die Versionen des **dx**-Compilers und anderer Erstellungswerkzeuge (z. B. **AAPT** (Android Asset Packaging Tools) für die Ressourcen-Verwaltung) leiten sich aus der SDK-Version ab. Sind für eine SDK-Version mehrere Versionen der Erstellungswerkzeuge erschienen und installiert, kann optional per **buildToolsVersion** eine konkrete Version eingestellt werden, statt die neueste installierte Version zu akzeptieren, z. B.:

**buildToolsVersion '27.0.3'**

- **minSdkVersion**

Diese Einstellung legt das von einer App vorausgesetzte minimale API-Level fest. Das **lint**-Werkzeug im Android Studio überwacht den Code auf die Verwendung von API-Bestandteilen, die im minimal vorausgesetzten SDK nicht vorhanden sind, z. B.:<sup>1</sup>

```
Toolbar myToolbar = (Toolbar) findViewById(R.id.my_toolbar);
```

Class requires API level 21 (current min is 15): android.widget.Toolbar [more...](#) (Strg+F1)

Durch die Verwendung der Support Library kann eine App möglichst viele aktuelle Android-Features nutzen (z. B. das mit der Android-Version 5.0 (API-Level 21) eingeführte Material Design) und trotzdem die **minSdkVersion** niedrig halten (z. B. bei API-Level 15).<sup>2</sup> Im Beispiel besteht die Lösung darin, nicht die Klasse **Toolbar** aus dem Paket **android.widget** zu verwenden, sondern die gleichnamige Klasse aus dem Paket **android.support.v7.widget**. Eine App kann zur Laufzeit das vorhandene API-Level feststellen und sein Funktionsangebot entsprechend anpassen.

<sup>1</sup> Die Fehlermeldung zu provozieren, gelang im Beispiel allerdings nur mit der seit Android 8.0 (API-Level 26) eigentlich überflüssigen expliziten Typumwandlung (**Toolbar**).

<sup>2</sup> <https://developer.android.com/topic/libraries/support-library/>

- **targetSdkVersion**

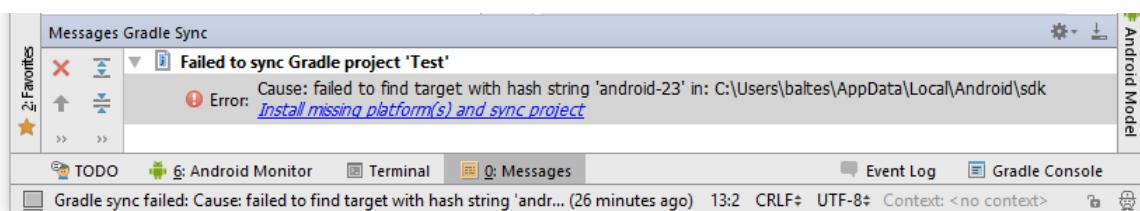
Hier ist das höchste API-Level anzugeben, für das eine App getestet worden ist, in dem sich eine App also korrekt verhalten kann. Generell haben die Android-Designer großen Wert auf die Vorwärts-Kompatibilität gelegt, und das Erscheinen einer neuen Android-Version bringt für eine vorhandene App normalerweise keine Kompatibilitätsprobleme. Mit den Verhaltensregeln der neuen Android-Version wird eine App nur dann konfrontiert, wenn sie eine passende **targetSdkVersion** besitzt. Z. B. wird das mit Android 6.0 (API-Level 23) eingeführte Berechtigungsmodell nicht auf Apps mit einer **targetSdkVersion** < 23 angewendet. Natürlich erwarten Benutzer von einer App, dass sie durch ein Update an die aktuelle Android-Version angepasst wird.

Im Unterschied zur **compileSdkVersion** landen die **minSdkVersion** und die **targetSdkVersion** in der Manifestdatei einer App (siehe Abschnitt 5.2).

## 4.10 Kursprojekte öffnen

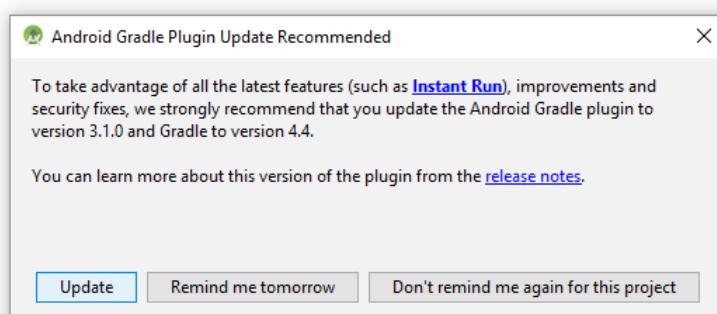
Obwohl die Android Studio - Entwickler offenbar die Übertragung von Projekten auf andere Rechner nicht vorrangig unterstützen, sollte es (zumindest auf einem Windows-Rechner) gelingen, die Kursprojekte über den simplen Menübefehl **File > Open** zu öffnen.

Als wesentliche Voraussetzung muss die SDK-Installation Ihres Rechners die Android-Version 8.1 (Oreo, API-Level 27) enthalten. Ansonsten scheitert das Öffnen von Kursprojekten mit einer Fehlermeldung analog zum folgenden Beispiel:



Das fehlende SDK wird nach einem Klick auf den Link automatisch installiert.

Ein vorgeschlagenes **Update** der Gradle-Software zur Projekterstellung sollte man akzeptieren, z. B.:

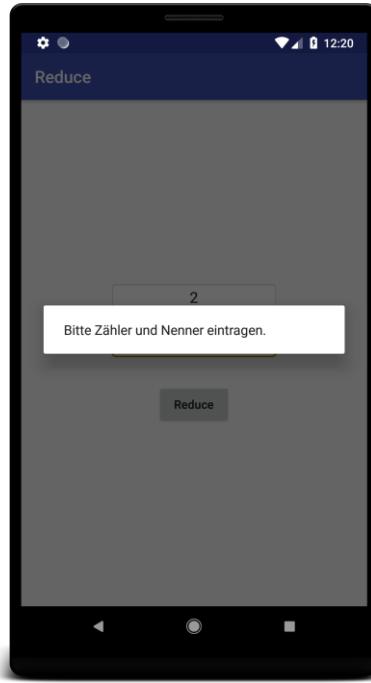


Wie die Kursprojekte zu beziehen sind, wird im Vorwort erläutert. Um das Transportvolumen zu verringern, wurde bei allen Kursprojekten der Ordner ...\\app\\build gelöscht. Das Android Studio erstellt beim Öffnen eines Projekts die fehlenden Objekte automatisch neu.

## 4.11 Übungsaufgaben zu Kapitel 4

1) Die in Kapitel 4 erstellte App stürzt nach der in Abschnitt 4.4 vorgenommenen Verbesserung der Methode `onClick()` nicht mehr ab, wenn das Kürzen für einen unvollständig beschriebenen Bruch

angefordert wird. Momentan passiert in diesem Fall einfach nichts. Erweitern Sie die Methode `onClick()` so, dass der Benutzer in einem Meldungsfenster über die korrekte Benutzung des Programms informiert wird, z. B.:



Hinweise:

- Erzeugen Sie ein Objekt der Klasse **AlertDialog.Builder** (aus dem Paket **android.support.v7.app**):.  
`AlertDialog.Builder builder = new AlertDialog.Builder(this);`
- Beauftragen Sie dieses Objekt, den Meldungstext festzulegen und das Dialogfenster (ein Objekt der Klasse **AlertDialog**) zu erstellen:  
`builder.setMessage("Bitte Zähler und Nenner eintragen.");  
AlertDialog dialog = builder.create();`  
Für Zeichenfolgen werden wir bald **String**-Ressourcen verwenden, damit unsere Apps leicht in andere Sprachen übersetzt werden können.
- Lassen Sie das **AlertDialog**-Objekt auftreten:  
`dialog.show();`

---

## 5 Komponenten und Systemumgebung einer App

In diesem Kapitel werden die Komponenten einer Android-App und ihre Systemumgebung vorgestellt.

### 5.1 Komponenten einer Android-App

In Android besteht eine App aus mehreren Komponenten, die darauf angelegt sind, auch von anderen Apps benutzt zu werden. So stellt z. B. die Standardanwendung zur Adressverwaltung einige von ihren Komponenten auch anderen Anwendungen zur Verfügung, die z. B. auf diesem Weg eine E-Mail - Adresse zu einem Kontakt ermitteln können. Selbstverständlich entscheidet letztlich der Anwender (bis zur Android-Version 5 bei der Installation, seit der Version 6 auch später) darüber, welche Rechte eine App haben soll.

Eine Android-App kann Komponenten folgenden Typs enthalten:

- **Activity**

Eine Activity (dt.: *Aktivität*) präsentiert eine Bildschirmseite mit Bedienelementen für eine bestimmte Tätigkeit (z. B. Liste mit Kontakten einsehen, Details zu einem Kontakt bearbeiten). Meist belegt die Bedienoberfläche einer Aktivität die komplette Display-Fläche.<sup>1</sup> Gelegentlich befindet sie sich in einem Dialogfenster, das vor dem Fenster einer übergeordneten Aktivität erscheint und nur einen Teil der Anzeigefläche belegt.

In der Regel enthält eine App *mehrere* Aktivitäten, zwischen denen der Benutzer zur Erledigung bestimmter Aufgaben mit Hilfe von Bedienelementen wechseln kann. Beim App-Start erscheint die Haupt- oder Startaktivität. Unsere bisherigen Beispiel-Apps haben sich allerdings auf eine einzige Aktivität beschränkt. Während die Aktivitäten vom Anwender als die sichtbaren Bestandteile einer App erlebt werden, arbeiten die anschließend beschriebenen Komponenten im Hintergrund.

- **Service**

Ein Service (dt.: *Dienst*) führt länger laufende Aufgaben im Hintergrund aus und hat keine Bedienoberfläche. Er kann von einer anderen Anwendungskomponente (von einer Activity, von einem Broadcast Receiver oder von einem anderen Service) *gestartet* werden (über die Methode **startService()**), um eine einzelne Aufgabe auszuführen. In diesem Fall beendet sich der Dienst in der Regel selbst, sobald seine Aufgabe (z. B. ein Download oder eine Synchronisation) erledigt ist. Ein gestarteter Dienst läuft weiter, wenn der Benutzer zu einer anderen Aktivität wechselt. In der Regel findet nach dem Start keine weitere Steuerung durch den Initiator statt. Statt einen Dienst zu starten, kann sich eine Anwendungskomponente an einen Dienst *binden* (über die Methode **bindService()**), um anschließend mit ihm zu interagieren (z. B. zur Steuerung einer Musikwiedergabe). Ein gebundener Dienst kann auch mehreren Klienten dienen und endet mit der letzten Bindung.

- **Broadcast Receiver**

Ein Broadcast Receiver (dt.: *Rundrufempfänger*) kann auf Rundrufnachrichten reagieren, die vom System oder von Anwendungen stammen (z. B. niedriger Akkuladezustand, WLAN-Verbindung hergestellt, Flugmodus beendet). Er hat keine eigene Bedienoberfläche und darf nur kurzzeitig aktiv werden, kann aber Aktivitäten oder Dienste starten und erlaubt somit den Anwendungsstart zu passenden Gelegenheiten.

---

<sup>1</sup> Seit der Android-Version 7 können auch zwei Aktivitäten gleichzeitig auf einem Display (neben- oder übereinander) präsent sein (geteilte Bildschirmsicht).

- **Content Provider**

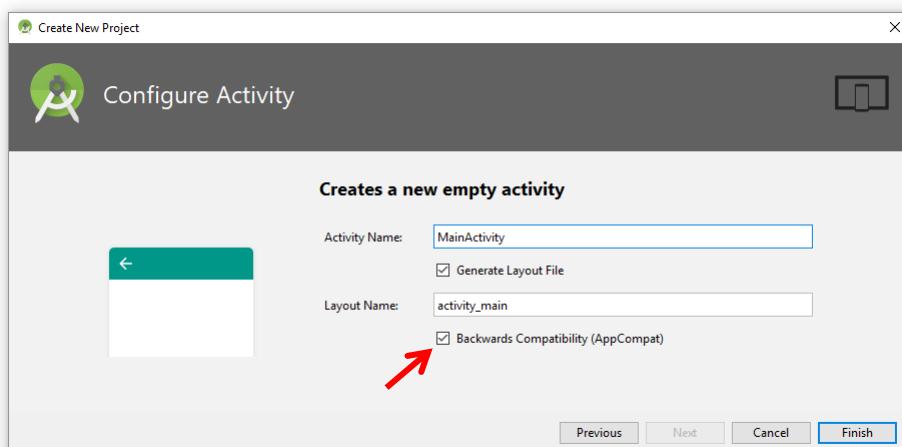
Während unter Windows jede Anwendung mit den Zugriffrechten des angemeldeten Benutzers arbeitet, schottet Android die Anwendungen untereinander ab, sodass eine App nicht auf die von einer anderen App verwalteten Daten zugreifen kann. Soll eine App die von ihr (z. B. in einer Datenbank, in einer Datei oder im Internet) verwalteten Daten für kontrollierte Zugriffe durch andere Apps zur Verfügung stellen, muss ein sogenannter *Content Provider* erstellt werden. Auch Android selbst enthält etliche Content Provider (z. B. für die Kontakte oder Einstellungen).

Zu den Komponententypen existieren im Android-API Basisklassen, von denen die in unseren eigenen Projekten verwendeten Komponenten abgeleitet werden. In unserem Beispielprogramm zum Kürzen von Brüchen wurde eine Aktivitätsklasse ausgehend von der Basisklasse

**AppCompatActivity** (aus dem Paket **android.support.v7.app**) definiert, die wiederum (indirekt) von der Klasse **Activity** (im Paket **android.app**) abstammt:

```
public class MainActivity extends AppCompatActivity implements View.OnClickListener {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button button = findViewById(R.id.button);
        button.setOnClickListener(this);
    }
    . .
}
```

Indem eigene Aktivitätsklassen *nicht* direkt von der Klasse **Activity**, sondern von der Klasse **AppCompatActivity** abgeleitet werden, kann eine App moderne Android-Techniken auch auf Geräten mit älteren Betriebssystemversionen nutzen. Für Projekte mit der im Kurs durchweg bevorzugten minimalen Android-Version 4.0.3 (API-Level 15) ist z. B. die **Material Design** - Unterstützung relevant. Das mit der Android-Version 5.0 (API-Level 21) eingeführte Material Design hat die Optik und die Funktionalität der Android-Bedienoberfläche modernisiert.<sup>1</sup> Für die Wahl der Abwärtskompatibilität entscheidet man sich durch ein (per Voreinstellung markiertes) Kontrollkästchen des Assistenten für neue Aktivitäten, z. B.:



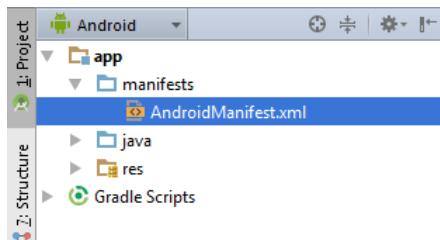
Man benutzt dabei eine sogenannte *Support Library*, im konkreten Fall die **v7 appcompat library** in der zum aktuellen SDK passenden Version 27.1.1.<sup>2</sup> Um die Verwaltung der daraus für eine App resultierenden Abhängigkeit kümmert sich das Gradle-Erstellungssystem.

<sup>1</sup> <https://developer.android.com/design/material/index.html>

<sup>2</sup> <https://developer.android.com/topic/libraries/support-library/features>

## 5.2 Manifest

Die Komponenten einer Android-App müssen in der Datei **AndroidManifest.xml** deklariert werden. Diese Datei ist im **Project**-Fenster des Android Studios (bei aktiver Sicht **Android**) im Ordner **manifests** zu finden, z. B.:<sup>1</sup>



Bislang mussten wir uns nicht explizit mit der Manifestdatei beschäftigen, weil die erforderlichen Eintragungen von Assistenten erledigt wurden. Die Manifestdatei ist XML-formatiert und enthält ein **application**-Element mit Kindelementen für alle Komponenten des Programms. Anschließend ist das Manifest zu der in Kapitel 4 erstellen Bruchkürzungs-App mit einer Aktivität als einziger Komponente zu sehen:

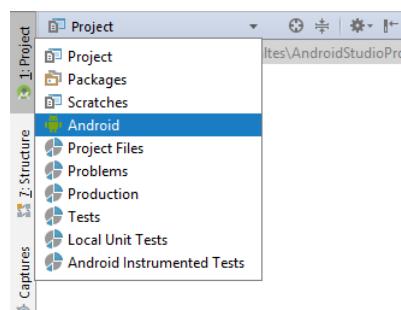
```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="de.zimkand.reduce">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Im Attribut **android:name** zum **activity**-Element ist der Name der realisierenden Klasse gefragt. Statt einen voll qualifizierten Namen inkl. Paketname anzugeben,

**de.zimkand.reduce.MainActivity**

---

<sup>1</sup> Wenn Sie im **Project**-Fenster den Ordner **manifests** vermissen, müssen Sie wahrscheinlich nur als Sicht für das **Project**-Fenster die Variante **Android** wählen:



kann man (wie im Beispiel) durch einen einleitenden Punkt vor dem Klassennamen den im Attribut **package** des **manifest**-Wurzelements angegebenen Paketnamen ansprechen.

Über **intent-filter** - Elemente werden die Kompetenzen einer Anwendungskomponente beschrieben. Im Beispiel finden sich die obligatorischen Angaben, um **MainActivity** zur Startaktivität (engl.: *default activity*) der App zu erklären.

Im weiteren Verlauf von Kapitel 5 konzentrieren wir uns auf die Activities. Später werden auch die restlichen Anwendungskomponenten behandelt.

In der Manifestdatei müssen neben den Anwendungskomponenten noch weitere Daten zu einer Anwendung deklariert werden, z. B.

- Paketname
- Benötigte Berechtigungen (z. B. Zugriff auf das Internet oder das Adressbuch)

In der vom Android Studio präsentierten Anwendungsmanifestdatei kein **uses-sdk** - Element zur Deklaration des minimal erforderlichen API-Levels.<sup>1</sup> Diese (z. B. vom Google Play Store benötigte) Angabe fügt das Gradle-Build-System aus dem Skript **build.gradle** in die APK-Datei mit der fertigen Anwendung ein. Diese erweiterte Version der Datei **AndroidManifest.xml** ist für eine im Debug-Modus erstellte App (vgl. Abschnitt 4.8) im Projektordner

**...\\app\\build\\intermediates\\manifests\\full\\debug** zu finden, z. B. für die Bruchkürzungs-App:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="de.zimkand.reduce"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="15"
        android:targetSdkVersion="27" />

    <application
        android:allowBackup="true"
        android:debuggable="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme" >
        <activity android:name="de.zimkand.reduce.MainActivity" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

<sup>1</sup> Eine entsprechende Forderung mit dem Muster

```
<manifest ... >
    <uses-feature android:name="android.hardware.camera.any"
                  android:required="true" />
    <uses-sdk android:minSdkVersion="7" android:targetSdkVersion="19" />
    ...
</manifest>
```

findet sich z. B. hier:

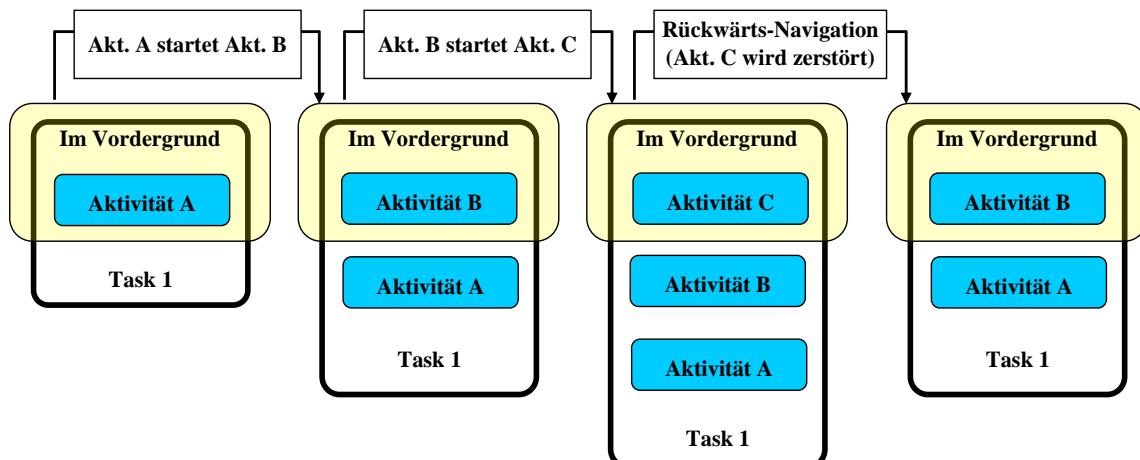
<https://developer.android.com/guide/components/fundamentals.html>

### 5.3 Task und Back Stack

Eine Reihe von nacheinander in einem Arbeitsablauf gestarteten Aktivitäten bilden eine **Task** (dt.: *Aufgabe*), und Android verwaltet für jede Task einen LIFO-Stapel (*Last-In-First-Out*), der als *Back Stack* bezeichnet wird. Häufig gehören die Aktivitäten in einem Stapel zu selben App, doch ist es in Android keinesfalls ungewöhnlich, dass auch Aktivitäten fremder Apps einbezogen werden, die bestimmte Teilaufgaben (z. B. ein Foto aufnehmen) erledigen.

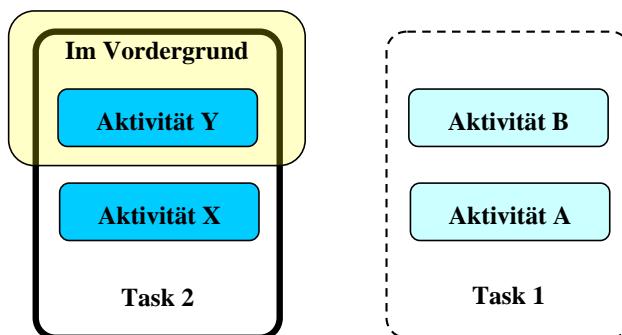
Wenn der Benutzer die Hauptaktivität einer Anwendung über eine Verknüpfung auf dem Startbildschirm oder per App-Starter (engl.: *launcher*) aktiviert und noch keine auf dieser Aktivität basierende Task existiert, legt Android eine neue Task an. Man kann also von *der Task* zu einer Anwendung sprechen.

Wird aus der ersten Aktivität heraus eine weitere gestartet, landet diese oben auf dem Stapel und kann mit dem Benutzer interagieren. Die überlagerte Aktivität wird **gestoppt**, und Android speichert den Zustand ihrer Bedienoberfläche für den Fall, dass der Benutzer zu dieser Aktivität zurückkehrt. In der folgenden Abbildung wächst der Stapel einer neuen Task auf drei Aktivitäten an, bevor der Benutzer per Rückwärts-Schalter die oberste Aktivität vom Stapel entfernt:<sup>1</sup>



Die entfernte Aktivität wird **zerstört**, und der Zustand ihrer Bedienoberfläche wird *nicht* gespeichert.

Eventuell wechselt der Benutzer anschließend per Home-Schalter zum Startbildschirm, um eine andere Aufgabe zu beginnen. Dann gerät der Back Stack zur ersten Task komplett in den Hintergrund:



<sup>1</sup> Das Vorbild für die Abbildung stammt von der Webseite:  
<http://developer.android.com/guide/components/tasks-and-back-stack.html>

Alle Aktivitäten von Task 1 sind im Zustand **gestoppt** und in Gefahr, bei Speichermangel beendet zu werden (siehe Kapitel 6). Der Back Stack zur Task 1 wird auch dann noch von Android verwaltet, wenn die Aktivitäten des Stapels bereits wegen Speichermangel zerstört worden sind. Wird das System herunter gefahren, gehen allerdings alle Aufgaben bzw. Stapel verloren.

Wechselt der Benutzer zum Startbildschirm und wählt erneut die Startaktivität der ersten (im Hintergrund befindlichen) Task, dann wird keine neue Task gestartet, sondern die vorhandene (als kompletter Stapel) in den Vordergrund geholt. Eine andere Möglichkeit für die Rückkehr zu einer Task bietet die Liste mit den zuletzt verwendeten Aufgaben, die auf einem Android-Gerät über einen speziellen Schalter (z. B.  ) oder (bei älteren Gräten) durch einen längeren Druck auf die Home-Taste angefordert werden kann.

Die Anwender erwarten, einen reaktivierten Stapel exakt im zuvor verlassenen Zustand vorzufinden. Welche Aufgaben die App-Entwickler beim Speichern und Restaurieren von Aktivitätszuständen zu erledigen haben, wird im Kapitel 6 über den Lebenszyklus einer Aktivität behandelt.

Entwickler können Ausnahmen von der normalen Task-Navigation anordnen. Eventuell soll z. B. ausgehend vom Stapel

$$A \rightarrow B \rightarrow C$$

das Starten der Aktivität B *nicht* zum Stapel

$$A \rightarrow B \rightarrow C \rightarrow B$$

führen, sondern zum Stapel

$$A \rightarrow B$$

Dazu ist ein **Intent**-Objekt mit dem Flag **FLAG\_ACTIVITY\_CLEAR\_TOP** an die Methode **startActivity()** zu übergeben. Dann wird die im Stapel vorhandene Instanz der Aktivität B reaktiviert, statt eine neue Instanz zu erstellen und auf den Stapel aufzulegen.

Arbeitet ein Android-Gerät im Mehrfenster-Modus (geteilte Bildschirmsicht, möglich ab Android 7.0 bzw. API-Level 24), dann besitzt jedes Fenster seine eigene Task-Verwaltung.

## 5.4 Apps und Prozesse

Unter Android wird jede Anwendung in einer eigenen virtuellen Java-Maschine ausgeführt, die als eigenständiger Prozess des zugrunde liegenden Linux-Betriebssystems läuft.<sup>1</sup> In diesem Anwendungsprozess laufen per Voreinstellung alle Komponenten einer Anwendung. Eine zu startende Komponente (z. B. eine Aktivität) wird also ...

- in einem neu erzeugten Prozess mit ausgeführt, wenn zu Ihrer Anwendung noch kein Prozess existiert,
- oder im bereits vorhandenen Prozess der Anwendung ausgeführt.

Es ist möglich, aber *selten erforderlich*, die Komponenten einer Anwendung auf mehrere Prozesse zu verteilen. Als Gründe für das Verteilen einer App auf mehrere Prozesse kommen in Frage:

---

<sup>1</sup> Aus Abschnitt 1.1 ist bereits bekannt, dass bis zur Android-Version 4.4 eine DVM (*Dalvik Virtual Machine*) zum Einsatz kommt und ab Android 5 eine ART (*Android Runtime*).

- Stabilität  
Die Stabilität steigt, weil die Konsequenzen eines Programmfehlers auf einen Prozess beschränkt bleiben.
- Verfügbarer Speicher  
Jeder Prozess erhält ein eigenes Heap-Speichersegment, sodass der für eine Anwendung verfügbare Speicher durch Verwendung von mehreren Prozessen erheblich gesteigert werden kann.

Einen Prozess (samt der virtuellen Maschine) zu starten, verursacht viel Aufwand. Daher versucht Android, Prozesse nach Möglichkeit auch dann noch am Leben zu erhalten, wenn sie aktuell nicht benötigt werden. Bei Speichermangel ist Android allerdings gezwungen, Prozesse zu zerstören, wobei die Wichtigkeit der Prozesse für den Benutzer über die Reihenfolge der Elimination entscheidet.

Es ist eine Besonderheit von Android im Vergleich zu anderen Betriebssystemen, dass die Entscheidung über das Prozessende *nicht* von der Anwendung selbst und auch nicht vom Benutzer getroffen wird.

Die Prozesse werden in Kategorien eingeteilt, die anschließend nach absteigender Wichtigkeit beschrieben werden:

- **Vordergrundprozess (engl.: *foreground process*)**

Gründe für die Zugehörigkeit eines Prozesses zu dieser Kategorie sind:

- Hier läuft die Aktivität, die sich gerade im Vordergrund befindet (mit **onResume()** als der zuletzt ausgeführten Lebenszyklusmethode, vgl. Abschnitt 6.2.4).
- Hier läuft gerade die **onReceive()** - Methode eines Broadcast Receivers.
- Hier läuft ein Dienst, der gerade eine Rückrufmethode ausführt (**onCreate()**, **onStart()** oder **onDestroy()**).

Im Normalfall existieren zu einem Zeitpunkt nur wenige Vordergrundprozesse, und es wird nur sehr selten (bei extremem Speichermangel) passieren, dass ein Vordergrundprozess zerstört werden muss.

- **Sichtbarer Prozess (engl.: *visible process*)**

Ein solcher Prozess enthält zwar keine Vordergrund-Aktivität, hat aber potentiell Einfluss auf sichtbare Bildschirminhalte. Gründe für die Zugehörigkeit eines Prozesses zu dieser Kategorie sind z. B.:

- Er enthält eine Aktivität, die sich zwar nicht mehr im Vordergrund befindet, aber noch sichtbar ist (die **onPause()** - Methode wurde ausgeführt, die **onStop()** - Methode aber noch nicht).
- Er enthält einen sogenannten *Vordergrunddienst*, der mit **startForeground()** gestartet wird und ein Symbol in die Statuszeile setzt, um auf eine Nachricht hinzuweisen.

- **Dienstprozess (engl.: *service process*)**

Ein solcher Prozess enthält einen mit **startService()** gestarteten Dienst, der z. B. einen Down- oder Upload-Auftrag im Hintergrund ausführt. Android geht bei Diensten von einer relativ hohen Relevanz für den Benutzer aus. Ein länger als 30 Minuten aktiver Dienst kann allerdings die Herabstufung zum Hintergrundprozess nicht verhindern. Seit Android 8 werden die von einer App gestarteten Dienste beendet, wenn sich die App seit mehreren Minuten im Hintergrund befindet (siehe Abschnitt 11.2.1).

- **Hintergrundprozess (engl.: *cached process*)**

Ein solcher Prozess enthält oft Aktivitäten, die für den Benutzer nicht mehr sichtbar sind und ihre **onStop()** - Methodenaufrufe absolviert haben. Oft sind zahlreiche Hintergrundprozesse vorhanden, und Android zerstört sie nach einer MRU-Regel (*Most Recently Used*). Das Todesrisiko steigt also, je länger für eine Aktivität der letzte Benutzerkontakt zurückliegt. Bei einem gut laufenden Android-System werden ausschließlich Hintergrundprozesse von der Speicherverwaltung abgeräumt.

Enthält ein Prozess B eine Komponente, die von einem anderen Prozess A benutzt wird, dann hat B mindestens die Priorität von A. Diese Konstellation liegt dann vor, wenn ...

- Prozess B einen Dienst enthält, an den eine Komponente von Prozess A gebunden ist,
- Prozess B einen Content Provider enthält, der von einer Komponente im Prozess A benutzt wird.

Nachdem der Prozess einer Anwendung gestartet worden ist, wird zunächst das **Anwendungsobjekt** instanziert. Es ist per Voreinstellung von der Klasse **Application** im Paket **android.app**, doch kann auch eine Ableitung dieser Klasse vorgenommen und über das Attribut **android:name** im **application**-Element der Manifestdatei deklariert werden. Aufgabe des Anwendungsobjekts ist die Verwaltung von anwendungsglobalen Zuständen. In den Kursbeispielen wird sich aber kein Anlass für einen direkten Zugriff auf das Anwendungsobjekt oder für die Definition einer **Application**-Ableitung ergeben.

Weitere Details sind auf Googles Entwickler-Webseite zum Thema zu finden.<sup>1</sup>

## 5.5 Sicherheit

Dass bei Android-Systemen trotzdem der verzögerten und bei älteren Geräten oft komplett unmöglichen Updates und trotz der hohen Verbreitung von Android und der entsprechend hohen Attraktivität für kriminelle Elemente vergleichsweise wenige Schäden durch Sicherheitslöcher auftreten, liegt wohl nicht zuletzt an den folgenden Ursachen:

- Android-Apps werden in der Regel ausschließlich über den Google Play Store bezogen, und dort kann Google Schadsoftware aussortieren. An dieser Voreinstellung sollten Anwender nichts ändern.
- Mit einer als *Google Play Protect* bezeichneten Technik können die auf einem Android-Gerät bereits installierten Apps regelmäßig überprüft werden. Diese Funktion ist folgendermaßen zu aktivieren:
  - Android < 4.2

**Google-Einstellungen > Sicherheit >  
Google Play Protect**

- Android ab 4.2

**Einstellungen > Google > Sicherheit >  
Google Play Protect**

- Android-Apps laufen in einem abgeschotteten Sandkasten ab, was gleich näher erläutert wird.

Android nutzt die Sicherheitsmechanismen des zugrunde liegenden Linux-Betriebssystems und sperrt die Apps folgendermaßen in einen Sandkasten ein, um Schäden durch böswillige Apps einzudämmen:<sup>2</sup>

---

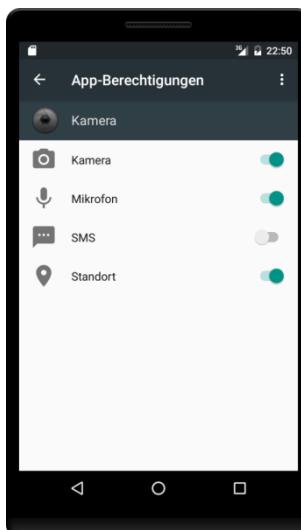
<sup>1</sup> <https://developer.android.com/guide/components/activities/process-lifecycle.html>

<sup>2</sup> <https://developer.android.com/guide/components/fundamentals.html>

- Jede App läuft mit einem eigenen Linux-Benutzerkonto und kann nur die Rechte dieses Kontos nutzen.
- Insbesondere werden Dateisystemrechte so gesetzt, dass jede App auf ihren eigenen Bereich im Linux-Dateisystem eingeschränkt ist.
- Für jede App ist eine eigene virtuelle Maschine (DVM bzw. ART) zuständig, die in einem eigenen Linux-Prozess ausgeführt wird. Damit kann eine App nicht auf den Hauptspeicher fremder Apps zugreifen.

Nur zwei mit demselben Zertifikat signierte Apps (vgl. Abschnitt 4.8) können über ihre Manifestdateien (mit dem Attribut **android:sharedUserId**) anfordern, in einem gemeinsamen Sandkasten ausgeführt zu werden, sodass z. B. wechselseitige Dateizugriffsrechte bestehen.<sup>1</sup>

Damit eine App die Dienste des Betriebssystems (z. B. Zugriff auf das Internet) oder die Dienste einer anderen App (z. B. Zugriff auf einen Content Provider wie das Adressbuch) nutzen kann, muss sie in ihrer Manifestdatei entsprechende Ansprüche anmelden, die vom Anwender genehmigt werden müssen und seit der Android-Version 6 (über **Einstellungen > Apps**) auch später noch administriert (z. B. entzogen) werden können, z. B.:



Android realisiert das *Prinzip der minimalen Berechtigung* (engl.: *principle of least privilege*), so dass eine App die meisten Rechte erst beantragen muss.

Wir werden uns in Kapitel 14 damit beschäftigen, wie eine App Berechtigungen zur Nutzung fremder Dienste anfordern und Berechtigungen zur Nutzung eigener Dienste durch andere Apps definieren kann.

## 5.6 Übungsaufgaben zu Kapitel 5

1) Welche von den folgenden Aussagen sind richtig bzw. falsch?

1. Alle Aktivitäten im Back Stack einer Task müssen zur selben App gehören.
2. Alle Aktivitäten im Back Stack einer Task werden im selben Linux-Prozess ausgeführt.
3. Ein Android-Entwickler kann dafür sorgen, dass mehrere von ihm signierte Apps in derselben Sandbox ausgeführt werden.

---

<sup>1</sup> <https://developer.android.com/guide/topics/manifest/manifest-element.html>

## 6 Lebenszyklus einer Aktivität

Android greift im Vergleich zu anderen Betriebssystemen stärker in den Lebenslauf einer Anwendung ein. Einzelne Anwendungskomponenten oder komplett Prozesse können aus dem Hauptspeicher entfernt und später bei Bedarf neu gestartet werden, wobei z. B. folgende Anlässe in Frage kommen:

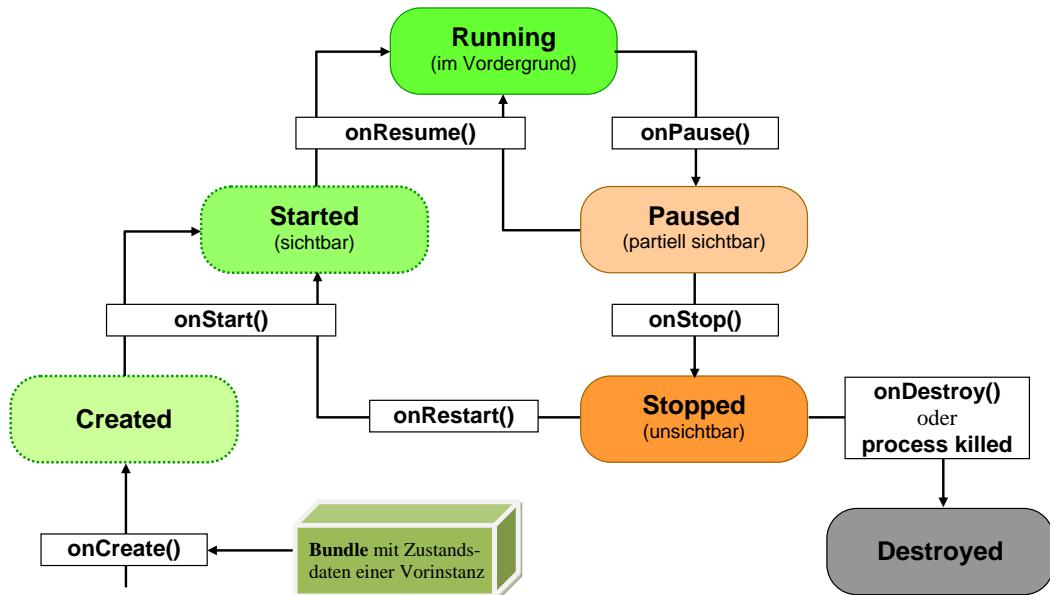
- Mangel an Ressourcen (vor allem Hauptspeicher)
- Wechsel der Geräteorientierung zwischen Hoch- und Querformat

Programmierer müssen auf solche Eingriffe vorbereitet sein und auf entsprechende Signale korrekt reagieren, damit die Anwender von Ärger (z. B. durch Daten- und Zeitverlust) verschont bleiben. Die Anwender ahnen nichts von den „dramatischen Ereignissen“ im Hintergrund, erwarten aber z. B., dass Aktivität zur Bearbeitung von Notizen nach dem Wechsel der Geräteorientierung denselben Text anzeigt und auf dieselbe Position fokussiert ist.

Im Vergleich zu anderen Betriebssystemen (z. B. Linux, MacOS X oder Windows) sind die Benutzer beim Anwendungs- bzw. Speicher-Management wenig beteiligt. Insbesondere sollen sie sich normalerweise nicht um das Beenden von Aktivitäten kümmern (siehe Gargenta & Nakamura 2014, Kap. 5). Damit hat der Aktivitäts-Manager in Android einiges zu tun, und für eine Android-Aktivität resultiert ein recht komplexer Lebenszyklus, mit dem wir uns nun befassen.

### 6.1 Zustände

Aus dem Benutzerverhalten und dem Activity - Management von Android resultieren für eine Aktivität verschiedene Zustände, wobei die Übergänge durch den Aufruf bestimmter Methoden gekennzeichnet sind. Wie Sie bereits wissen, wird in der frühen Startphase die Methode **onCreate()** aufgerufen, die für Initialisierungsarbeiten zuständig ist (z. B. für die Erstellung der Hierarchie von Bedienelementen). In der folgenden Abbildung sind die möglichen Zustände einer Aktivität zusammen mit den beim Zustandswechsel beteiligten Methoden bzw. Systemeingriffen für Geräte mit einer Android-Version ab 3.0 zu sehen:



Während die Zustände **Created** und **Started** schnell durchlaufen werden, kann sich eine Aktivität längere Zeit in den folgenden Zuständen befinden:

- **Running** (alias: **Resumed**)

Die Aktivität befindet sich im Vordergrund, hat den Eingabefokus und kann mit dem Anwender kommunizieren. In diesem Zustand befindet sich die Aktivität zwischen den Aufrufen von **onResume()** und **onPause()**.

- **Paused**

In diesem Zustand befindet sich eine Aktivität nach der Rückkehr der Methode **onPause()**. Die Aktivität ist *partiell* durch eine andere Aktivität überlagert, die entweder transparent ist oder nicht den gesamten Bildschirm belegt (z. B. ein Meldungsfenster), und kann keine Benutzereingaben mehr empfangen. Sie befindet sich funktionstüchtig im Speicher, bis der Benutzer zurückkehrt oder der Destruktionsprozess fortschreitet mit der Lebenszyklusmethode **onStop()**.<sup>1</sup> Eine Aktivität in diesem Zustand kann durchaus aktiv bleiben, obwohl die Zustandsbezeichnung Passivität suggeriert. Ein sinnloser Energieverbrauch muss allerdings unterbleiben.

- **Stopped**

In diesem Zustand befindet sich eine Aktivität nach der Rückkehr der Methode **onStop()**. Die Aktivität ist *komplett* durch eine andere Aktivität überlagert. Sie befindet sich funktionsfähig im Speicher und kann in diesem Zustand verweilen, bis der Benutzer zurückkehrt, oder der Destruktionsprozess fortschreitet. Eine Aktivität in diesem Zustand kann durchaus aktiv bleiben, obwohl die Zustandsbezeichnung Passivität suggeriert. Ein sinnloser Energieverbrauch muss allerdings unterbleiben. Im Zustand **Stopped** kann eine Aktivität (zusammen mit dem kompletten Prozess der Anwendung) vom System wegen Speichermangel zerstört werden. Wenn der Benutzer die Vordergrundaktivität durch Betätigung des Rückwärts-Schalters vom Back Stack abräumt, gelangt sie vom Zustand **Stopped** sofort in den Zustand **Destroyed**. Dasselbe passiert bei einem Orientierungswechsel zwischen Hoch- und Querformat.

- **Destroyed**

Entweder wartet die Aktivitätsinstanz nach dem Aufruf der Methode **onDestroy()** darauf, vom Garbage Collector der virtuellen Maschine aus dem Speicher geworfen zu werden, oder der Anwendungsprozess, in dem sie ausgeführt worden war, musste wegen Speichermangel zerstört werden. Wenn eine Rückkehr des Benutzers möglich ist, sichert Android die Steuerelementzustände der Aktivitätsinstanz (z. B. die Inhalte von Texteingabefeldern). Wird eine neue Instanz derselben Aktivität erstellt, werden die gespeicherten Zustandsdaten beim Aufruf der Methode **onCreate()** als Parameter vom Typ **Bundle** übergeben. Damit ein Steuerelement (ein Objekt der Klasse **View**) bei der Sicherung und Wiederherstellung unterstützt werden kann, muss es eine Kennung bzw. eine Element-ID (XML-Attribut **android:id**) besitzen (vgl. Abschnitt 8.2.2.3). Bei einer per Rückwärts-Schalter oder durch einen anderweitig initiierten **finish()** - Aufruf vom Back Stack entfernten Aktivitätsinstanz entfällt die Sicherung der Steuerelementzustände.

Im Zusammenhang mit der Terminierung einer Aktivität ist das Sichern ihrer Daten von Bedeutung, wobei zu unterscheiden ist zwischen:

- **Zustandsdaten**

Das betrifft die Bedienoberfläche (z. B. Inhalt eines Texteingabefelds) und die Instanzvariablen des Aktivitätsobjekts. Die Zustandsdaten sind zu sichern für den Fall, dass der Benutzer zu einer Aktivität zurückkehrt. Die in diesem Fall anzulegende Nachfolgerinstanz sollte dem untergegangenen Vorbild möglichst weitgehend entsprechen.

---

<sup>1</sup> Bei extremem Ressourcenmangel kann sich ein Android-System mit einer Version < 3.0 dafür entscheiden, die pausierte Aktivität zusammen mit dem kompletten Anwendungsprozess zu zerstören. Da wir in unseren Apps grundsätzlich (wie vom Android Studio empfohlen) die minimale Android-Version 4.0.3 (API-Level 15) voraussetzen, brauchen wir uns um Geräte mit einer Android-Version < 3.0 nicht zu kümmern.

- **Persistenzdaten**

Damit sind permanent zu speichernde Daten (z. B. ein vom Benutzer angelegter Kontakt) und Einstellungen (z. B. die vom Benutzer gewählte Schriftgröße) gemeint. Während Daten meist an einen Content Provider (siehe Kapitel 14) übergeben werden, dienen zum Konser vieren von Einstellungen oft die Shared Preferences (siehe Kapitel 15).

Beim Ableben einer Aktivität sind zwei Mechanismen zu unterscheiden, wobei im ersten Fall ein Objekt und im zweiten Fall ein Prozess betroffen ist (siehe z. B. Mednieks et al 2013, S. 340f).

- a) **Zerstörung von gestoppten Aktivitäts-Objekten**

Wenn von der **Zerstörung** einer bereits gestoppten Aktivität gesprochen wird, ist in der Regel folgender Ablauf gemeint:

- Die Aktivität erhält den Aufruf **onDestroy()**.
- Nach Rückkehr des Aufrufs verwirft Android seine Referenz auf das Aktivitätsobjekt (setzt sie auf **null**). Nun sollten keine Referenzen auf die Aktivität mehr vorhanden sein, denn es ist davon abzuraten, irgendwo im Programm Aktivitätsreferenzen aufzubewahren. Damit wird früher oder später der Garbage Collector (Müllsampler) der virtuellen Java-Maschine tätig und entfernt das Objekt aus dem Speicher.

Als Gründe für dieses Geschehen kommen z. B. in Frage:

- Die Aktivität befindet sich im Vordergrund, und der Benutzer betätigt den **Zurück-Schalter**. In diesem Fall wird nach **onStop()** sofort die Methode **onDestroy()** aufgerufen, und Android sichert den Zustand der Steuerelemente *nicht*.
- Die Aktivität befindet sich im Vordergrund, und es findet eine **Konfigurationsänderung** statt, die einen neuen Aufbau der Bedienoberfläche erzwingt. Neben seltenen Ereignissen wie einer Änderung der Ländereinstellung oder der Schriftgröße kommt vor allen ein Orientierungswechsel (vom Hoch- zum Querformat oder umgekehrt) in Betracht. Bei einer Konfigurationsänderung zerstört Android die Aktivität und startet im selben Prozess eine neue Instanz, um dem Programmierer Gelegenheit zu geben, die Bedienoberfläche neu zu gestalten. Ein Orientierungswechsel kann allerdings vom Benutzer per Gerätekonfiguration oder vom Programmierer verhindert werden.
- Eine Aktivität kann sich durch Aufruf ihrer **finish()** - Methode selbst beenden, was in speziellen Situationen sinnvoll ist. In der Regel sollte man **finish()** nicht verwenden, sondern stattdessen die gleich zu beschreibenden Methoden für Zustandswechsel implementieren und dem System die Aktivitätsverwaltung überlassen. Die **finish()** - Methode wird auch bei Betätigung des Zurück-Schalters aufgerufen.
- Das System benötigt mehr Ressourcen für die Vordergrundaktivität und zerstört eine gestoppte Aktivität.

- b) **Terminierung von Prozessen**

Bei einem gravierenden Ressourcenmangel hält sich Android nicht mit den Aufrufen von **onDestroy()** für einzelne Aktivitäten auf, sondern terminiert nach einer Prioritätenliste komplett Prozesse (siehe Abschnitt 5.4).

Es ist zu beachten, dass eine Aktivität nach der Ausführung von **onStop()** von der Eliminierung wegen Ressourcenmangel bedroht ist. Falls eine Rückkehr des Benutzers möglich ist, merkt sich Android für eine eliminierte Aktivität den Zustand der Steuerelemente und die Position der Aktivität in ihrem Back Stack. Wenn der Programmierer sinnvolle Überschreibungen der Zustandswechselmethoden erstellt (z. B. mit Konservierung der Werte von Instanzvariablen), dann bemerkt der Benutzer bei seiner Rückkehr zur Aktivität nichts davon, dass es sich um eine komplett neue Instanz handelt, die eventuell in einem anderen Prozess ausgeführt wird.

Wird eine Aktivität im Rahmen des geplanten Programmablaufs terminiert (z. B. Benutzer betätigt den Zurück-Schalter, Aktivität beendet sich selbst per **finish()**), dann speichert Android *keine* Zustandsdaten.

## 6.2 Zustandswechselmethoden

Anschließend werden die Aufgaben der Zustandswechselmethoden beschrieben. Diese Methoden dürfen keinesfalls vom Programmierer aufgerufen werden. Es handelt sich um sogenannte *Callback-Methoden*, die für den Aufruf durch den Aktivitäts-Manager in Android bestimmt sind.

Abgeleitete Klassen müssen in ihren Überschreibungen der Zustandswechselmethoden die jeweilige Basisklassenvariante aufrufen, was meist zu Beginn geschieht.

### 6.2.1 onCreate()

Die Methode

**protected void onCreate(Bundle savedInstanceState)**

wird von Android beim Erstellen einer Aktivitätsinstanz aufgerufen und *muss* implementiert werden. Hier werden die Bestandteile der Aktivität initialisiert. Insbesondere wird durch einen Aufruf der Methode **setContentView()** die Bedienoberfläche der Aktivität erstellt, wobei eine Resource ID für das in einer XML-Datei definierte Layout zu übergeben ist, z. B.:

```
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_main);  
    Button button = findViewById(R.id.button);  
    button.setOnClickListener(this);  
}
```

Ziemlich regelmäßig sind in **onCreate()** auch Aufrufe der Methode **findViewById()** anzutreffen, die Referenzen auf wichtige, im Layout der Aktivität definierte Steuerelemente liefert, die in Methoden der Aktivitätsklasse angesprochen werden sollen.

Die Methode **onCreate()** erhält als Parameter ein Objekt der Klasse **Bundle**, in dem sich eventuell der gespeicherte Zustand einer vorher (z. B. wegen Speichermangel oder Orientierungswechsel) zerstörten Vorgängerinstanz befindet, ansonsten eine **null**-Referenz.

### 6.2.2 onStart()

Die Methode

**protected void onStart()**

wird aufgerufen, wenn die Aktivität sichtbar wird. Hier sind Maßnahmen mit Relevanz für die Gültigkeit der Bedienoberfläche zu ergreifen. Ist z. B. ein Broadcast Receiver bei der Aktualisierung der Bedienoberfläche beteiligt, dann sollte seine Registrierung in **onStart()** vorgenommen werden.

Die Methode **onRestart()** ist für den Fall gedacht, dass bei der Reaktivierung einer zuvor gestoppten Aktivität im Vergleich zur Methode **onStart()** zusätzliche Maßnahmen erforderlich sind, was praktisch nie der Fall ist.

### 6.2.3 onRestoreInstanceState()

Die in der obigen Abbildung (siehe Abschnitt 6.1) *nicht* enthaltene Methode

**protected void onRestoreInstanceState(Bundle savedInstanceState)**

wird nach **onStart()** aufgerufen und erhält (wie **onCreate()**) ein **Bundle**-Objekt mit den zuvor durch die Methode **onSaveInstanceState()** (siehe unten) gesicherten Zustandsdaten einer (z. B. wegen Speicherlager oder Orientierungswechsel zerstörten) Vorgängerinstanz. Die Implementation der **Activity**-Basisklassen im Android-API sorgt für eine Wiederherstellung der Steuerelementzustände. Weil **onRestoreInstanceState()** nur dann aufgerufen wird, wenn ein **Bundle**-Objekt vorhanden ist, kann die in **onCreate()** erforderliche Existenzprüfung entfallen.

In der Regel verzichtet man darauf, **onRestoreInstanceState()** zu überschreiben und erledigt eigenverantwortlich vorzunehmende Restaurierungen von Zustandsdaten in **onCreate()**.

### 6.2.4 onResume()

Nach der Rückkehr der Methode

**protected void onResume()**

kann die Aktivität mit dem Benutzer interagieren. Bei einem Spiel wird man hier z. B. die regelmäßige Ausführung einer Methode zur Neuberechnung und -anzeige des Spielfelds in Gang setzen. Weil **onResume()** auch nach jeder Rückkehr aus dem pausierten Zustand aufgerufen wird, sind jetzt alle Ressourcen zu initialisieren, die in **onPause()** freigegeben wurden (z. B. die Kamera).

### 6.2.5 onSaveInstanceState()

Die in der obigen Abbildung (siehe Abschnitt 6.1) *nicht* enthaltene Methode

**protected void onSaveInstanceState(Bundle outState)**

wird aufgerufen, ...

- wenn eine Aktivitätsinstanz von der Bildfläche verschwindet und damit von Zerstörung bedroht ist,
- und dem eventuell nach der Zerstörung zurückkehrenden Benutzer in der neu zu erzeugenden Aktivitätsinstanz der verlassene Zustand wieder angeboten werden soll.

Der Aufrufzeitpunkt ist nicht exakt determiniert:<sup>1</sup>

- Bis zur Android-Version 8.x (Oreo) erfolgt der Aufruf vor oder nach **onPause()**, auf jeden Fall aber vor **onStop()**.
- Bei späteren Android-Versionen wird der Aufruf nach **onStop()** erfolgen.

Von der Basisklassenvariante (definiert in der Klasse **Activity**) werden die Zustände aller Bedienelemente gesichert, die eine Kennung (XML-Attribut **android:id**) besitzen. Sollen weitere Zustandsinformationen (z. B. Werte von Instanzvariablen) gesichert werden, muss die Methode überschrieben werden. Den obligatorischen Aufruf der Basisklassenvariante setzt man in der Regel ans Ende der Überschreibung.

Die Methode **onSaveInstanceState()** erhält als Parameter das **Bundle**-Objekt, das von Android zum Sichern von Zustandsdaten verwendet und bei einer späteren Wiederherstellung der Aktivität an **onCreate()** sowie an **onRestoreInstanceState()** übergeben wird.

In folgenden Fällen wird **onSaveInstanceState()** *nicht* aufgerufen:

---

<sup>1</sup> [https://developer.android.com/reference/android/app/Activity.html#onSaveInstanceState\(android.os.Bundle\)](https://developer.android.com/reference/android/app/Activity.html#onSaveInstanceState(android.os.Bundle))

- Wenn eine Aktivitätsinstanz per **finish()** beendet wurde.
- Wenn eine Aktivitätsinstanz vom Benutzer per Zurück-Schalter verlassen wurde.  
In diesem Fall wird die **Activity**-Methode **onBackPressed()** aufgerufen, deren Standardimplementation darin besteht, **finish()** aufzurufen.

Weil **onSaveInstanceState()** nicht unter allen Umständen aufgerufen wird, eignet sich die Methode *nicht* zum Sichern von Persistenzdaten. Für diese wichtige Aufgabe bieten sich die folgenden Gelegenheiten an:

- Eine Aktivität kann bereits im Zustand **Running** Daten speichern, z. B. beim Verlassen eines Texteingabefeldes.
- In der oft aufgerufenen Methode **onPause()** sollten *keine* aufwändigen Sicherungen durchgeführt werden. Ab Android 3.0 wird nach **onPause()** garantiert auch noch **onStop()** aufgerufen, und diese Methode ist dazu geeignet, aufwändige Sicherungen durchzuführen.

## 6.2.6 onPause()

Die Methode

### **protected void onPause()**

wird aufgerufen, wenn die Aktivität (teilweise) durch eine andere Aktivität überlagert wird, und keine weitere Interaktion mit dem Benutzer möglich ist.

Alle nur im Vordergrund sinnvollen Tätigkeiten der Activity müssen jetzt eingestellt werden, z. B. die Aktualisierung einer Spieloberfläche. Vor allem sind Tätigkeiten zu unterbrechen, die ...

- CPU-Zeit oder Energie verbrauchen (z. B. Animationen, GPS-Abfragen),
- Ressourcen blockieren (z. B. die Kamera).

Geht der **onPause()** - Aufruf auf einen vorherigen Aufruf der Methode **finish()** zurück, dann liefert die Methode **isFinishing()** den Wert **true**, und eine Rückkehr zur Aktivität ist ausgeschlossen.

Es wird oft empfohlen, Persistenzdaten (z. B. gerade editierte Kontaktdaten) in **onPause()** zu sichern (z. B. durch Übergabe an einen Content Provider). Solche Empfehlungen finden sich vor allem in älteren Anleitungen, weil eine Aktivität vor Android 3.0 nach der Ausführung von **onPause()** davon bedroht war, bei Speichermangel eliminiert zu werden. Seit Android 3.0 ist aber garantiert, dass nach **onPause()** auch noch **onStop()** ausgeführt wird. Da wir in unseren Apps grundsätzlich (wie vom Android Studio empfohlen) die minimale Android-Version 4.0.3 (API-Level 15) voraussetzen, brauchen wir uns um Geräte mit einer Android-Version < 3.0 nicht zu kümmern. Google rät davon ab, aufwändige Arbeiten wie das Sichern von Benutzerdaten sowie Netz- oder Datenbankzugriffe in **onPause()** durchzuführen und empfiehlt die Methode **onStop()** für diese Arbeiten.<sup>1</sup> Das Speichern von Persistenzdaten kann durchaus bereits im Zustand **Running** erfolgen, z. B. beim Verlassen eines Texteingabefeldes.

Auch bei Benutzung des Zurück-Schalters müssen Persistenzdaten gesichert werden. Der Zurück-Schalter ist definitiv *nicht* als Abbrechen-Schalter konzipiert, was die Google-Dokumentation für Entwickler unmissverständlich beschreibt:<sup>2</sup>

Note this implies that the user pressing BACK from your activity does *not* mean "cancel" -- it means to leave the activity with its current contents saved away.

Soll eine Abbrechen-Option angeboten werden, dann ist diese zusätzlich zu implementieren.

---

<sup>1</sup> <https://developer.android.com/guide/components/activities/activity-lifecycle.html>

<sup>2</sup> <http://developer.android.com/reference/android/app/Activity.html>

In **onPause()** einen Dialog zu präsentieren, ist absolut unvereinbar mit der Android-Bedienungslogik, denn ein Anlass für den **onPause()** - Aufruf kann z. B. ein ankommender Anruf sein.

### 6.2.7 onStop()

Die Methode

#### **protected void onStop()**

wird aufgerufen, wenn eine Aktivität komplett von der Bildfläche verschwindet. Hier sollten Tätigkeiten gestoppt werden, die bei einer unsichtbaren Aktivität nutzlos sind. Ist z. B. ein Broadcast Receiver bei der Aktualisierung der Bedienoberfläche beteiligt, so sollte seine Registrierung in **onStop()** aufgehoben und in **onStart()** wieder restauriert werden.

Nach der Rückkehr von **onStop()** ist eine Aktivität im Risiko, bei Speichermangel vom System samt Anwendungsprozess zerstört zu werden. Daher ist es oft erforderlich, in **onStop()** Persistenzdaten zu sichern.

### 6.2.8 onDestroy()

In der Methode

#### **protected void onDestroy()**

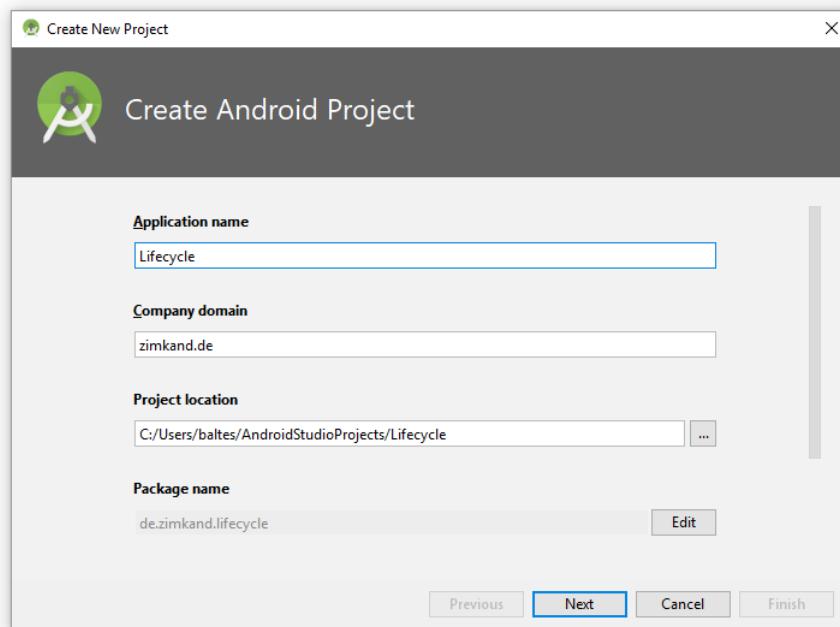
sollten aufwändige Ressourcen freigegeben werden. Google nennt als typische Aufgabe für diese Methode das Stoppen von Threads (siehe Kapitel 10), die mit der zu beendenden Aktivität assoziiert sind. Geht der **onDestroy()** - Aufruf auf einen vorherigen Aufruf der Methode **finish()** zurück, mit der sich eine Aktivität selbst beenden kann, dann liefert die Methode **isFinishing()** den Wert **true**. Ob der aktuelle Aufruf von **onDestroy()** auf einen Konfigurationswechsel zurückgeht, erfährt man durch einen Aufruf der **Activity**-Methode **isChangingConfigurations()** (siehe Beispiel in Abschnitt 11.3.2).

## 6.3 App zur Lifecycle-Demonstration

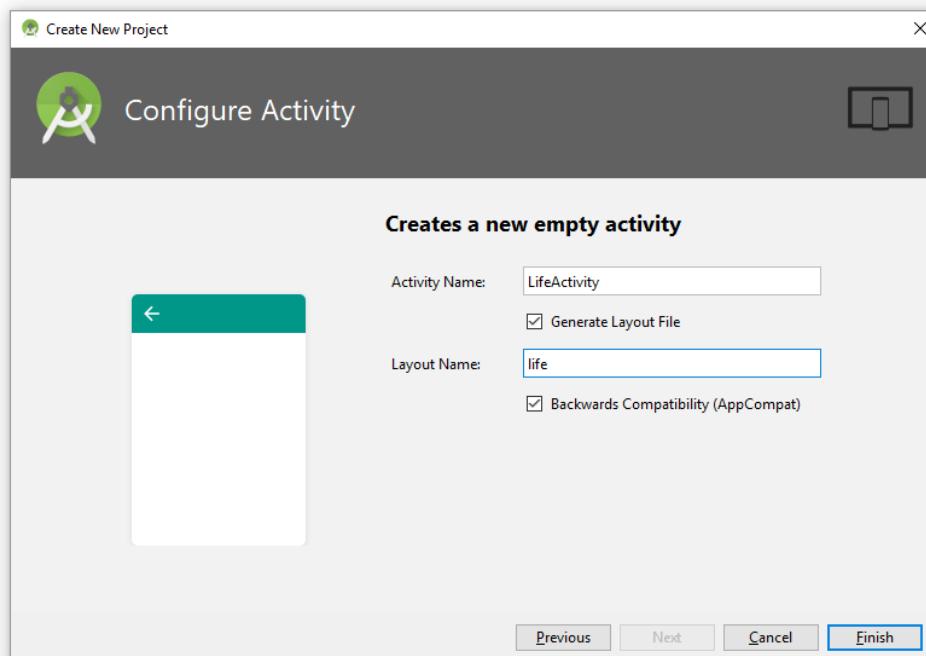
Um Stationen im Lebenszyklus einer Aktivität zu demonstrieren, erstellen wir über

**File > New Project**

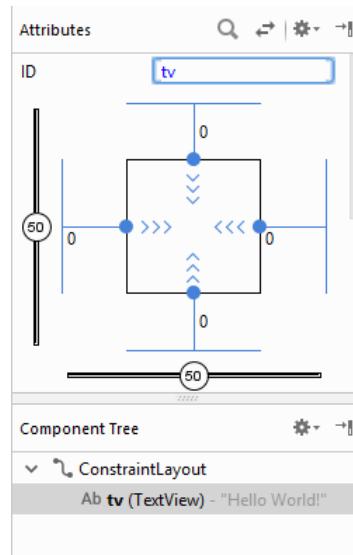
ein neues Projekt mit dem Namen **Lifecycle**:



Wir akzeptieren als **Target Android Devices** wie üblich die Voreinstellung **Phone and Tablet** mit dem minimalen API-Level 15 und lassen eine **Empty Activity** anlegen, deren Konfiguration wir diesmal zu Übungszwecken modifizieren. Wir verwenden **LifeActivity** als **Activity Name** (als Namen für die Java-Klasse der Aktivität) und **life** als **Layout Name** (als Namen für die XML-Datei, welche das Layout der Aktivitätsbedienoberfläche definiert):

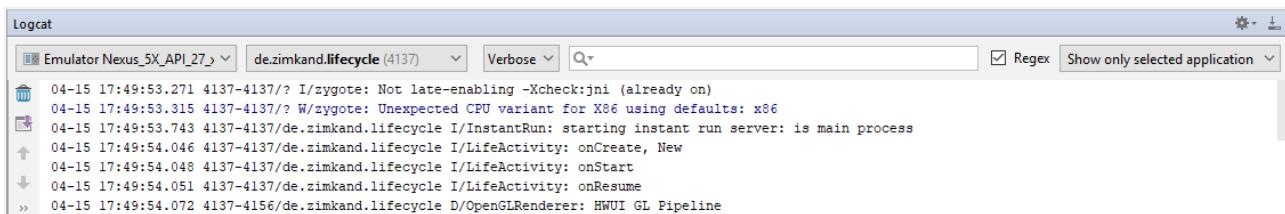


Wir lassen das Projekt mit einem Klick auf **Finish** erstellen und kümmern uns dann im **Design**-Modus des Layout-Editors (zur Bearbeitung der Datei **life.xml**) um die Bedienoberfläche. Hier beschränken wir uns darauf, eine Ressourcen-Kennung (**ID**) für das vorgegebene **TextView**-Objekt zu vergeben:



### 6.3.1 Logging

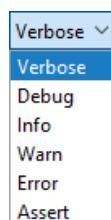
Das **Logcat**-Fenster im Android Studio zeigt die System-Logs eines emulierten oder per USB verbundenen Android-Geräts an. Diese sind zur Beobachtung einer App und bei der Fehleranalyse außerordentlich nützlich:



Bedeutung der Spalten auf dem **logcat**-Registerblatt:

- Datum  
z. B. **04-15**
- Uhrzeit in GMT  
z. B. **17:49:54.046**
- Prozess-Identifikation  
z. B. **4137-4137/de.zimkand.lifecycle**
- Log-Kategorie und Etikett (tag) der Meldung (siehe unten)  
z. B. **I/MainActivity**
- Meldungstext  
z. B. **onCreate, New**

Die **logcat**-Einträge werden nach Schweregrad bzw. Anlass klassifiziert. Über das Drop-Down - Menü mit dem voreingestellten Wert **Verbose** kann man entscheiden, was angezeigt werden soll:



Im **Logcat**-Fenster erscheinen zahlreiche Systemmeldungen, und wir können in unseren Programmen zusätzliche Ausgaben veranlassen. Dazu genügt der altbekannte Methodenaufruf **println()** an das Objekt **System.out**. Allerdings sind zur Erstellung von **Logcat**-Einträgen die statischen Me-

thoden der Klasse **Log** aus dem Paket **android.util** zu bevorzugen, wobei über den Methodennamen (**v()**, **d()**, **i()**, **w()**, **e()**) die gewünschte Log-Kategorie gewählt wird. Mit dem Methode **i()** erzeugt man z. B. Meldungen aus der Kategorie **Info**.

Alle **Log**-Ausgabemethoden akzeptieren zwei **String**-Parameter:

- **tag**

Über dieses Etikett wird üblicherweise die Herkunft einer Meldung bekannt gegeben, wobei sich oft der Namen der Aktivitätsklasse anbietet. Um Aufwand und Tippfehler einzusparen, ist die Verwendung einer finalisierten **String**-Variablen zu empfehlen, z. B.:

```
private final String TAG = this.getClass().getSimpleName();
```

Um den Klassennamen der aktuellen Aktivität zu ermitteln, wird das handelnde Aktivitätsobjekt (angesprochen per **this**) über die Methode **getClass()** um eine Referenz auf das die eigene Klasse repräsentierende Objekt gebeten. Dieses Klassenobjekt beherrscht die Methode **getSimpleName()**, die als Rückgabe den Klassennamen (ohne Paketangabe) liefert.<sup>1</sup>

- **text**

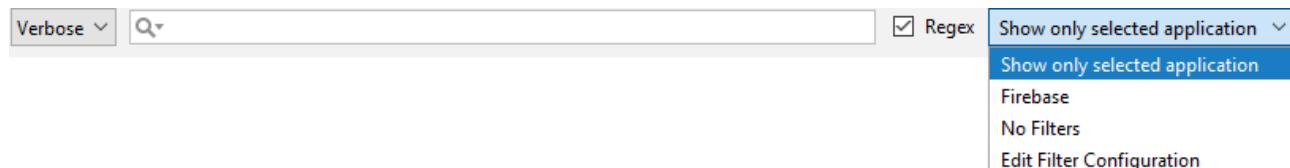
Der Meldungstext

Im folgenden Aufruf der Methode **Log.i()** wird die Ausführung der **Activity**-Methode **onCreate()** dokumentiert:

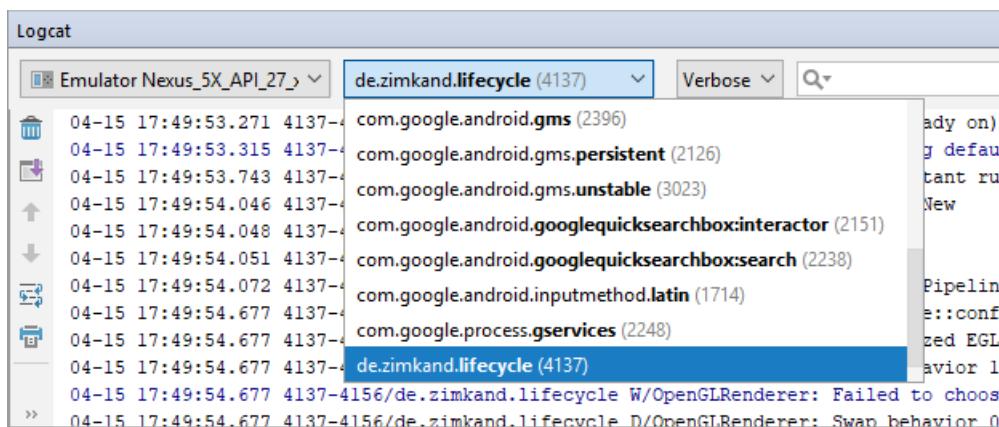
```
Log.i(TAG, "onCreate, " + (savedInstanceState == null ? "New" : "Restored"));
```

An **onCreate()** wird über den Parameter vom Typ **Bundle** entweder ein Objekt übergeben, das den gesicherten Zustand einer von Android zuvor eliminierten Aktivitätsinstanz enthält, oder die **null**-Referenz. Durch den Konditionaloperator wird im Beispiel für eine passende Ausgabe gesorgt.

Im **Logcat**-Fenster kann man die Meldungen nicht nur auf eine Log-Kategorie beschränken, sondern auch nach diversen Kriterien filtern. Im Drop-Down-Menü am rechten Rand



ist die (voreingestellte) Option verfügbar, ausschließlich die Meldungen der ausgewählten App



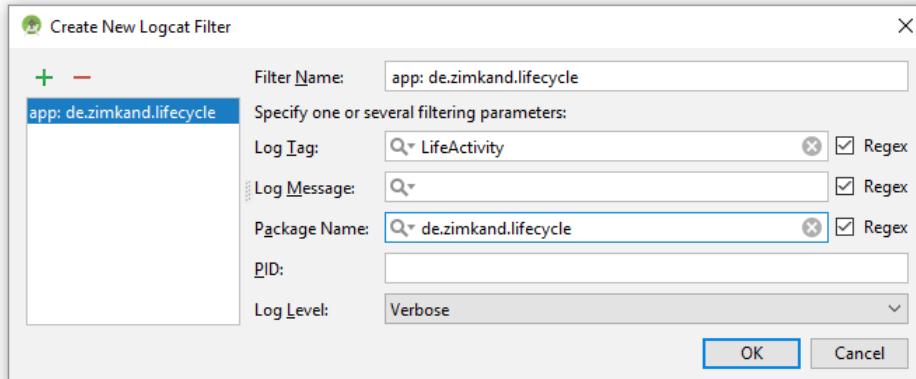
anzuzeigen. Nach dem Start einer App ist diese ausgewählt, was zu einer vorteilhaften Filterung der **Logcat**-Ausgaben führt.

Das Item **Edit Filter Configuration** im Drop-Down-Menü mit den Filteroptionen öffnet einen Dialog, der eine Auswahl der Meldungen nach diversen Kriterien erlaubt. Wir definieren einen Filter mit dem Namen

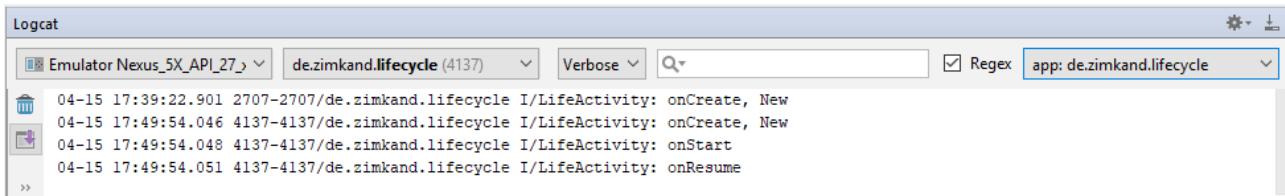
<sup>1</sup> Per **System.out** erstellte Ausgaben erhalten das Etikett „**System.out**“.

app: de.zimkand.lifecycle

mit den folgenden Einstellungen:

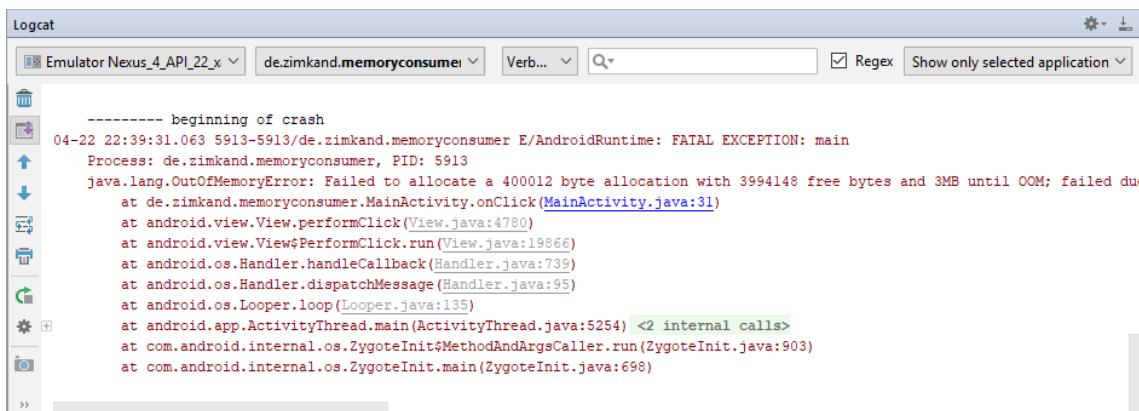


Indem wir im Feld **Log Tag** den von unserer App beim Aufruf einer **Log**-Methode aufgrund des **tag**-Parameters ausgegebenen Activity-Klassennamen eintragen und außerdem noch den Paketnamen unserer App angeben (Feld **Package Name**), ist dafür gesorgt, dass in der **logcat**-Sicht ausschließlich die von uns veranlassten Meldungen erscheinen, z. B.:



Über den Schalter lassen sich die Einträge im **Logcat**-Fenster löschen. Wenn die Sicht erwartungswidrig leer bleibt, hilft eventuell ein Klick auf den **Restart**-Schalter .

Es ist keinesfalls generell günstig, die **Logcat**-Einträge auf selbst erzeugte Meldungen zu beschränken. Zeigt ein **Logcat**-Fenster alle die eigene App betreffenden Meldungen an, erfährt man z. B. nach einem Laufzeitfehler die Fehlerursache und die Aufrufersequenz an der Fehlerstelle. Diese Informationen sind für die Fehlerdiagnose extrem nützlich, z. B. nach einem **OutOfMemoryError**:



### 6.3.2 Protokollausgaben in den Methoden für Activity-Zustandswechsel

Wir überschreiben in unserer Aktivitäts-Klassendefinition die Methoden für kritische **Activity**-Lebensereignisse und veranlassen jeweils einen Protokolleintrag:

```
package de.zimkand.lifecycle;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.util.Log;
import android.widget.TextView;

public class LifeActivity extends AppCompatActivity {

    private final String TAG = this.getClass().getSimpleName();
    private static int nAct=0;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.life);
        Log.i(TAG, "onCreate, " + (savedInstanceState == null ? "New" : "Restored"));
        TextView tv = findViewById(R.id.tv);
        tv.setText("Activity No "+Integer.toString(++nAct));
    }

    @Override
    protected void onStart() {
        super.onStart();
        Log.i(TAG, "onStart");
    }

    @Override
    protected void onRestoreInstanceState(Bundle savedInstanceState) {
        super.onRestoreInstanceState(savedInstanceState);
        Log.i(TAG, "onRestoreInstanceState");
    }

    @Override
    protected void onResume() {
        super.onResume();
        Log.i(TAG, "onResume");
    }

    @Override
    protected void onPause() {
        super.onPause();
        Log.i(TAG, "onPause" + (this.isFinishing() ? "(Finishing)" : ""));
    }

    @Override
    protected void onSaveInstanceState(Bundle savedInstanceState) {
        super.onSaveInstanceState(savedInstanceState);
        Log.i(TAG, "onSaveInstanceState");
    }

    @Override
    protected void onStop() {
        super.onStop();
        Log.i(TAG, "onStop");
    }

    @Override
    protected void onRestart() {
        super.onRestart();
        Log.i(TAG, "onRestart");
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        Log.i(TAG, "onDestroy");
    }
}
```

In der ersten Anweisung der Lebenszyklus-Methoden wird jeweils die überschriebene Basisklassenmethode aufgerufen, z. B.:

```
@Override
protected void onStart() {
    super.onStart();
    Log.i(TAG, "onStart");
}
```

Wir werden gelegentlich dafür sorgen, dass während der Nutzung unserer Aktivität (nacheinander) verschiedene Instanzen entstehen. Um diese fortlaufend nummerieren zu können, definieren wir das statische (der Klasse zugeordnete) Feld `nAct` vom primitiven Typ `int`:

```
private static int nAct=0;
```

In der Methode `onCreate()`, die beim Erstellen einer neuen Aktivitätsinstanz abläuft, wird `nAct` inkrementiert und zur Aktualisierung der `TextView`-Anzeige verwendet:

```
tv.setText("Activity No "+Integer.toString(++nAct));
```

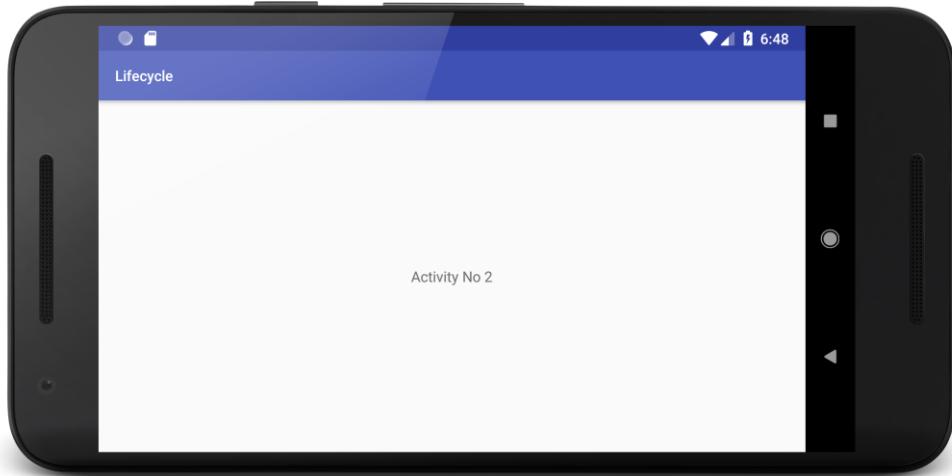
Das komplette Projekt `Lifecycle` ist im folgenden Ordner zu finden

`...\\BspUeb\\Lifecycle`

Wir beobachten nun das Activity-Management von Android in verschiedenen Szenarien.

### 6.3.2.1 Neustart nach Konfigurationsänderung

Wir starten die App auf einem Smartphone im Hochformat und führen über das Symbol  in der Palette zum emulierten Smartphone einen Orientierungswechsel durch. Anschließend zeigt die App dieses Bild,



und im gefilterten **Logcat**-Fenster (vgl. Abschnitt 6.3.1) erhalten wir die nach obigen Erläuterungen zu erwartende Sequenz von Einträgen:

Time	Process	Log Message
04-15 17:49:54.046	4137-4137/de.zimkand.lifecycle	I/LifeActivity: onCreate, New
04-15 17:49:54.048	4137-4137/de.zimkand.lifecycle	I/LifeActivity: onStart
04-15 17:49:54.051	4137-4137/de.zimkand.lifecycle	I/LifeActivity: onResume
04-15 18:42:19.956	4137-4137/de.zimkand.lifecycle	I/LifeActivity: onPause ← Orientierungswechsel
04-15 18:42:19.961	4137-4137/de.zimkand.lifecycle	I/LifeActivity: onSaveInstanceState
04-15 18:42:20.126	4137-4137/de.zimkand.lifecycle	I/LifeActivity: onStop
04-15 18:42:20.127	4137-4137/de.zimkand.lifecycle	I/LifeActivity: onDestroy
04-15 18:42:20.483	4137-4137/de.zimkand.lifecycle	I/LifeActivity: onCreate, Restored
04-15 18:42:20.484	4137-4137/de.zimkand.lifecycle	I/LifeActivity: onStart
04-15 18:42:20.485	4137-4137/de.zimkand.lifecycle	I/LifeActivity: onRestoreInstanceState
04-15 18:42:20.488	4137-4137/de.zimkand.lifecycle	I/LifeActivity: onResume

Die neue Aktivitätsinstanz lebt im selben Prozess (mit der Nummer 4137).

### 6.3.2.2 Neukreation nach Prozessterminierung

Um das Verhalten unserer App nach einer Prozessterminierung beobachten zu können, ohne durch das Starten von zahlreichen Apps einen Speichermangel provozieren zu müssen, verwenden wir zwei Android-Entwickleroptionen:<sup>1</sup>

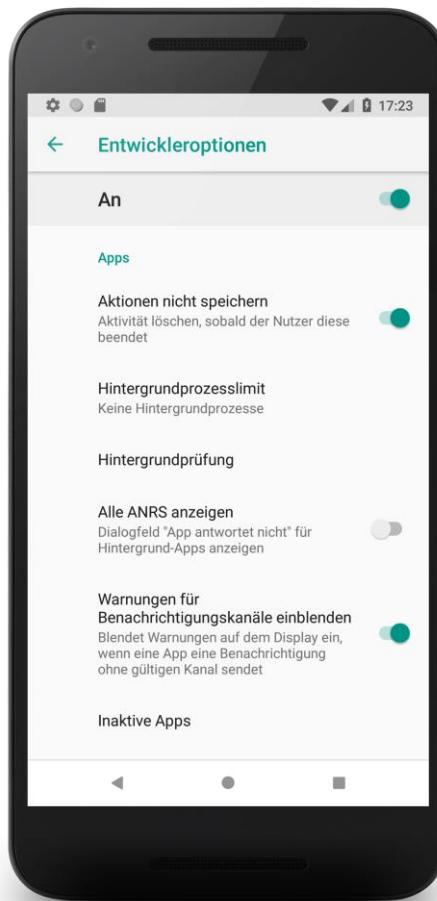
- **Aktionen nicht speichern**

Ist diese Einstellung aktiviert, wird eine Aktivität zerstört, sobald sie vom Benutzer verlassen wird (z. B. in Richtung Startbildschirm).

- **Hintergrundprozesslimit**

Wählt man die Option **Keine Hintergrundprozesse**, dann werden leere Hintergrundprozesse sofort entfernt (vgl. Abschnitt 5.4).

Hier sind die beiden Einstellungen bei einem Smartphone mit Android 8.1 (API-Level 27) zu sehen:



Wir starten unsere App, verlassen sie in Richtung Startbildschirm und kehren dann zurück (z. B. über die Liste mit den zuletzt verwendeten Aufgaben aufrufen). Wie die **Logcat**-Einträge

```

04-16 17:03:30.281 6282-6282/de.zimkand.lifecycle I/LifeActivity: onCreate, New
04-16 17:03:30.285 6282-6282/de.zimkand.lifecycle I/LifeActivity: onStart
04-16 17:03:30.290 6282-6282/de.zimkand.lifecycle I/LifeActivity: onResume
04-16 17:14:35.953 6282-6282/de.zimkand.lifecycle I/LifeActivity: onPause
04-16 17:14:36.551 6282-6282/de.zimkand.lifecycle I/LifeActivity: onSaveInstanceState
04-16 17:14:36.555 6282-6282/de.zimkand.lifecycle I/LifeActivity: onStop
04-16 17:14:51.599 12377-12377/de.zimkand.lifecycle I/LifeActivity: onCreate, Restored
04-16 17:14:51.602 12377-12377/de.zimkand.lifecycle I/LifeActivity: onStart
04-16 17:14:51.605 12377-12377/de.zimkand.lifecycle I/LifeActivity: onRestoreInstanceState
04-16 17:14:51.610 12377-12377/de.zimkand.lifecycle I/LifeActivity: onResume

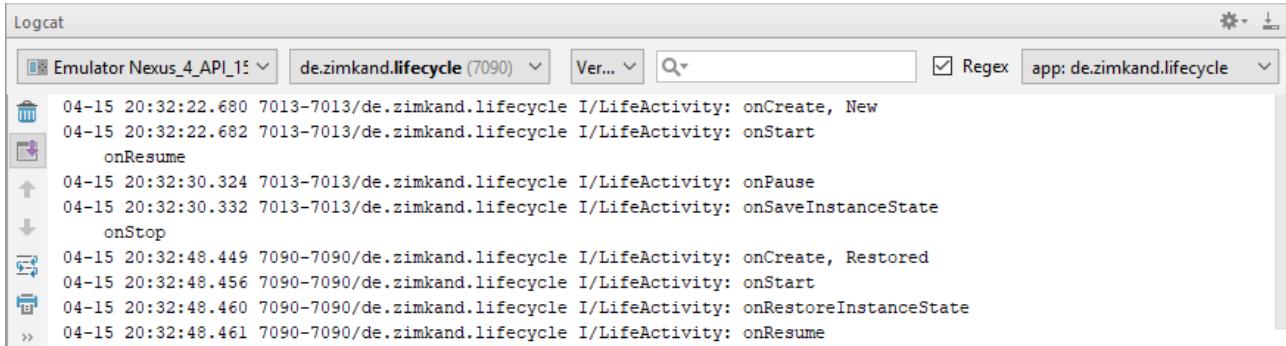
```

<sup>1</sup> Siehe Abschnitt 3.5.1 wegen der eventuell erforderlichen Aktivierung von Entwickleroptionen

zeigen, ...

- hat die App offenbar *keinen* Aufruf der Methode **onDestroy()** erhalten, weil der gesamte Anwendungsprozess zerstört worden ist,
- ist die Aktivität in einem anderen Prozess restauriert worden, was in der Spalte mit der Prozess-Identifikation abzulesen ist.

Auf einem Rechner mit Speichermangel (getestet mit 4 GB) tauchen im **Logcat**-Fenster gelegentlich unvollständige Einträge auf, z. B.:



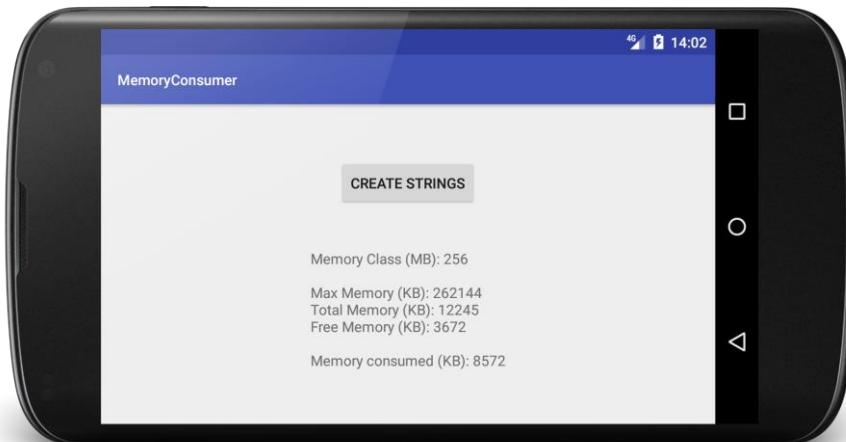
```

Logcat
Emulator Nexus_4_API_15 de.zimkand.lifecycle (7090) Ver... Q Regex app: de.zimkand.lifecycle
04-15 20:32:22.680 7013-7013/de.zimkand.lifecycle I/LifeActivity: onCreate, New
04-15 20:32:22.682 7013-7013/de.zimkand.lifecycle I/LifeActivity: onStart
    onResume
04-15 20:32:30.324 7013-7013/de.zimkand.lifecycle I/LifeActivity: onPause
04-15 20:32:30.332 7013-7013/de.zimkand.lifecycle I/LifeActivity: onSaveInstanceState
    onStop
04-15 20:32:48.449 7090-7090/de.zimkand.lifecycle I/LifeActivity: onCreate, Restored
04-15 20:32:48.456 7090-7090/de.zimkand.lifecycle I/LifeActivity: onStart
04-15 20:32:48.460 7090-7090/de.zimkand.lifecycle I/LifeActivity: onRestoreInstanceState
>> 04-15 20:32:48.461 7090-7090/de.zimkand.lifecycle I/LifeActivity: onResume

```

## 6.4 Speicherverbrauch einer App

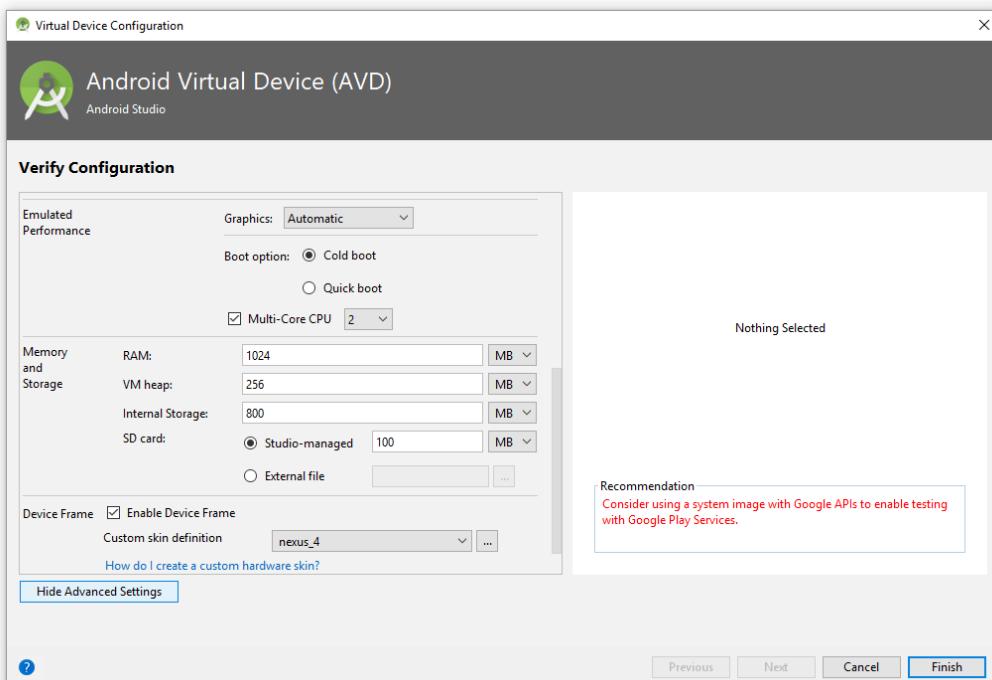
Nachdem wir in Abschnitt 6.3 mit dem **Logcat**-Fenster ein sehr nützliches Diagnoseinstrument kennengelernt haben, werfen wir bei dieser Gelegenheit einen Blick auf das benachbarte Fenster **Android Profiler**, das es ermöglicht, den Speicherbedarf, die CPU-Nutzung und die Netzwerkaktivität einer App zu beobachten. Wir beschränken uns auf die Speicherverwendung und erstellen eine App namens **MemoryConsumer** mit der folgenden Bedienoberfläche:



Bei jedem Klick auf den Schalter werden ca. 4 MB Heap-Speicher belegt. Eine **TextView**-Komponente zeigt die Rückgaben von diversen Speicherinformationsmethoden an, die gleich erläutert werden.

#### 6.4.1 Speicherinformationsmethoden

Solange ein Objekt existiert, befinden es sich mit all seinen Instanzvariablen in dem als **Heap** (deutsch: *Haufen*) Bereich des programmeigenen Arbeitsspeichers.<sup>1</sup> Wieviel Heap-Speicher einer App zur Verfügung steht, hängt u.a. von der RAM-Ausstattung des Android-Gerätes ab. Weil Android auf Multitasking ausgelegt ist, also den Simultanbetrieb von mehreren Apps unterstützt, wird einer einzelnen App natürlich nicht der gesamte RAM-Speicher zur Verfügung gestellt. Bei einem emulierten Android-Gerät lässt sich per AVD-Manager die maximale Heap-Größe einer App einstellen, wobei allerdings Nebenbedingungen dafür sorgen, dass nicht jeder eingestellte Wert akzeptiert wird. Im folgenden Beispiel werden als **VM heap** die Werte 256 und 512 akzeptiert:



Bei einem Objekt der Klasse **ActivityManager** kann eine App ihre aktuelle Speicherklasse (engl.: *memory class*) erfragen, wobei es sich um einen **int**-Wert mit der maximalen Größe des Heap-Speichers in MB handelt. Eine Aktivität kann sich über die Methode **getSystemService()** eine Referenz auf ein Objekt der Klasse **ActivityManager** verschaffen, um es über die Methode **getMemoryClass()** nach dem maximalen Hauptspeicher zu fragen, z. B.:

```
private ActivityManager am;
private int memoryClass;

@Override
protected void onCreate(Bundle savedInstanceState) {
    . . .
    am = (ActivityManager) getSystemService(ACTIVITY_SERVICE);
    memoryClass = am.getMemoryClass();
    . . .
}
```

Dieselbe Information (diesmal in Bytes) erhält man auch von einem Objekt der Klasse **Runtime**, das über die statische Methode **getRuntime()** der Klasse **Runtime** (im Paket **java.lang**) verfügbar ist,

```
private Runtime runtime = Runtime.getRuntime();
```

<sup>1</sup> Neben dem Heap-Speicher benötigt ein Programm noch weitere Speicherbereiche für seine Variablen. Solange eine Methode ausgeführt wird, befinden sich ihre lokalen Variablen in einem Bereich des programmeigenen Arbeitsspeichers, den man als **Stack** (deutsch: *Stapel*) bezeichnet.

nach Befragen über die Methode **maxMemory()**, z. B.:

```
long max = runtime.maxMemory()/1024;
```

Wenn eine Aktivität bei einer Speicheranforderung das Limit überschreitet, wird sie mit einem Laufzeitfehler vom Typ **OutOfMemoryError** beendet.

Wieviel Heap-Speicher einer App zugebilligt worden ist, verrät die **Runtime**-Methode **totalMemory()**, z. B.:

```
long total = runtime.totalMemory()/1024;
```

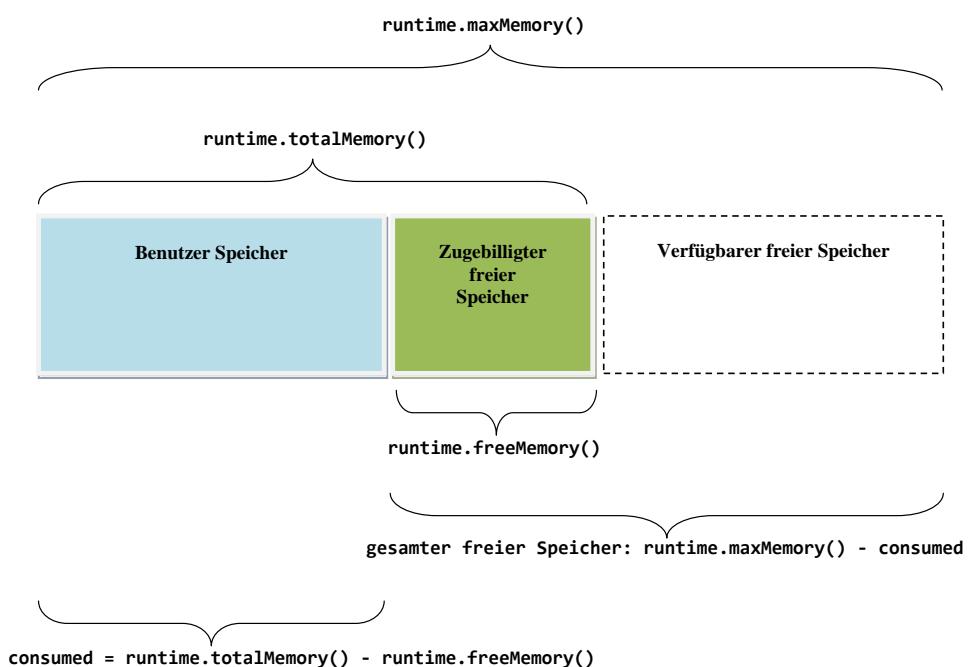
Der zugebilligte Speicher ist in der Regel noch nicht komplett belegt. Von der **Runtime**-Methode **freeMemory()** erfährt man die aktuelle Größe der Reserve, z. B.:

```
long free = runtime.freeMemory()/1024;
```

Aus dem zugebilligten und dem noch freien Speicher lässt sich errechnen, wieviel Heap-Speicher eine App aktuell belegt, z. B.:

```
long consumed = (runtime.totalMemory() - runtime.freeMemory())/1024;
```

In der folgenden Abbildung sind die Speicherbeiche bzw. die Rückgaben der beschriebenen Abfragemethoden zu sehen:<sup>1</sup>



#### 6.4.2 Beispielprogramm zur Speicherverschwendug

Es folgt der Quellcode zur App **MemoryConsumer**, welche die beschriebenen Speicherinformationsmethoden verwendet:

<sup>1</sup> Das Vorbild für die Abbildung stammt von:

<https://stackoverflow.com/questions/3571203/what-are-runtime-getruntime-totalmemory-and-freememory>

```

package de.zimkand.memoryconsumer;

import android.app.ActivityManager;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.*;

public class MainActivity extends AppCompatActivity implements View.OnClickListener {
    private String[][] as = new String[1000][];
    private int index = 0;
    private Runtime runtime = Runtime.getRuntime();
    private TextView tv;
    private ActivityManager am;
    private int memoryClass;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button button = findViewById(R.id.button);
        tv = findViewById(R.id.tv);
        button.setOnClickListener(this);
        am = (ActivityManager) getSystemService(ACTIVITY_SERVICE);
        memoryClass = am.getMemoryClass();
        showMemory();
    }

    @Override
    public void onClick(View v) {
        String[] star = new String[100_000];
        for (int i = 0; i < 100_000; i++)
            star[i] = new String();
        as[index++] = star;
        showMemory();
    }

    private void showMemory() {
        long max = runtime.maxMemory()/1024;
        long total = runtime.totalMemory()/1024;
        long free = runtime.freeMemory()/1024;
        long consumed = (runtime.totalMemory() - runtime.freeMemory())/1024;
        tv.setText("Memory Class (MB): " + Long.toString(memoryClass) +
                   "\n\nMax Memory (KB): " + Long.toString(max) +
                   "\nTotal Memory (KB): " + Long.toString(total) +
                   "\nFree Memory (KB): " + Long.toString(free) +
                   "\n\nMemory consumed (KB): " + Long.toString(consumed));
    }
}

```

Die Klickbehandlungsmethode `onClick()` zum Befehlsschalter wird (wie im Bruchkürzungsprограмm, siehe Abschnitt 4.3) vom Aktivitätsobjekt ausgeführt. Diese Methode erzeugt einen (ziemlich sinnlosen) Array mit 100.000 leeren **String**-Objekten und belegt ca. 4 MB Heap-Speicher. Durch einen als Instanzvariable angelegten Array mit **String**-Arrays als Elementen bestehen Referenzen auf die **String**-Objekte außerhalb der Ereignismethode, sodass die Objekte *nicht* dem Garbage Collector zum Opfer fallen. Über das etwas umständlich definierte Objekt zur Speicherverschwendungen sollten die Leser nicht allzu intensiv nachdenken. Jedenfalls erhöht die App bei jedem Klick ihren Heap-Speicherverbrauch um ca. 4 MB.

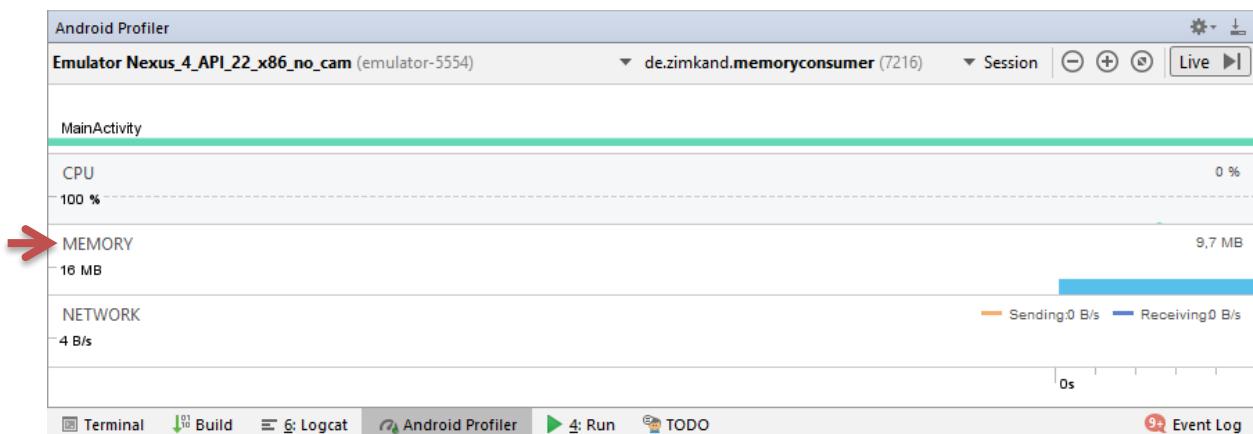
Wir werden in Abschnitt 6.4.3 den Heap-Speicherverbrauch der App mit dem Profiler im Android Studio beobachten und schließlich in Abschnitt 6.4.4 durch eine unerfüllbare Speicheranforderung einen Laufzeitfehler vom Typ **OutOfMemoryError** provozieren, der zur Beendigung des Programms durch Android führt.

Das komplette Projekt **MemoryConsumer** ist im folgenden Ordner zu finden

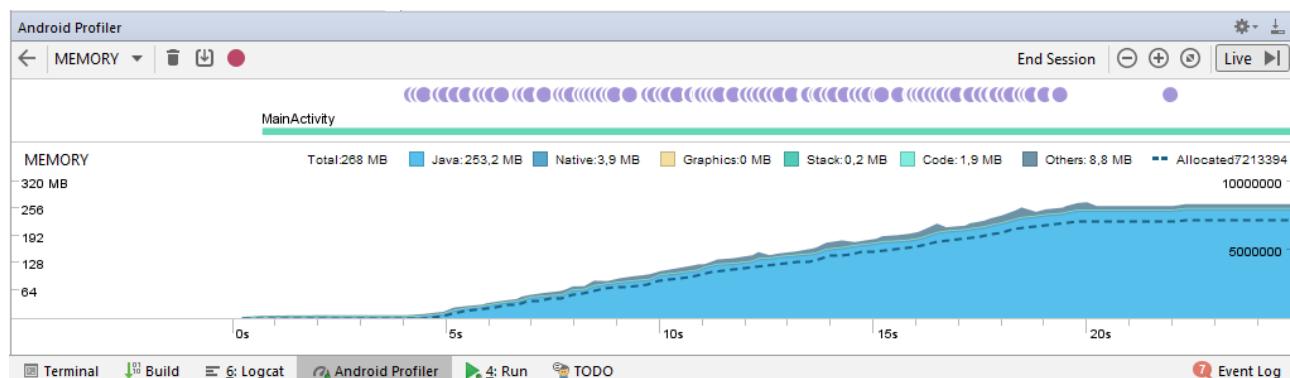
...\\BspUeb\\MemoryConsumer

#### 6.4.3 Speicherbedarf per Profiler beobachten

Nachdem wir die App **MemoryConsumer** auf einem Android-Gerät mit dem minimalen API-Level 21 gestartet haben, wählen wir sie im Fenster **Android Profiler** über ihren Paketnamen aus

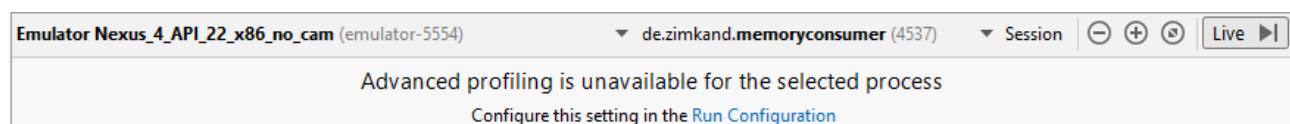


und klicken dann auf **MEMORY**, um die Speicherverwendung im Detail zu beobachten. Das Android Studio liefert ein Verlaufsdiagramm zur Speichernutzung durch die **MainActivity**, und durch fleißige Klicks auf den Schalter **CREATE STRINGS** in unserer App steigt die Heap-Größe (siehe Kategorie **Java** im Diagramm) bis auf ca. 256 MB (= **maxMemory()** - Rückgabe):

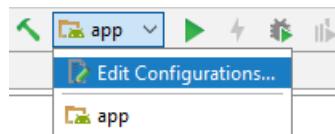


Auf einem emulierten Gerät mit Android 8.1 (API-Level 27) wurde die App unbenutzbar zäh und benötigte ca. eine Minute für die Reaktion auf eine Schalterbetätigung, wenn sie unter Beobachtung durch den **Profiler** stand.

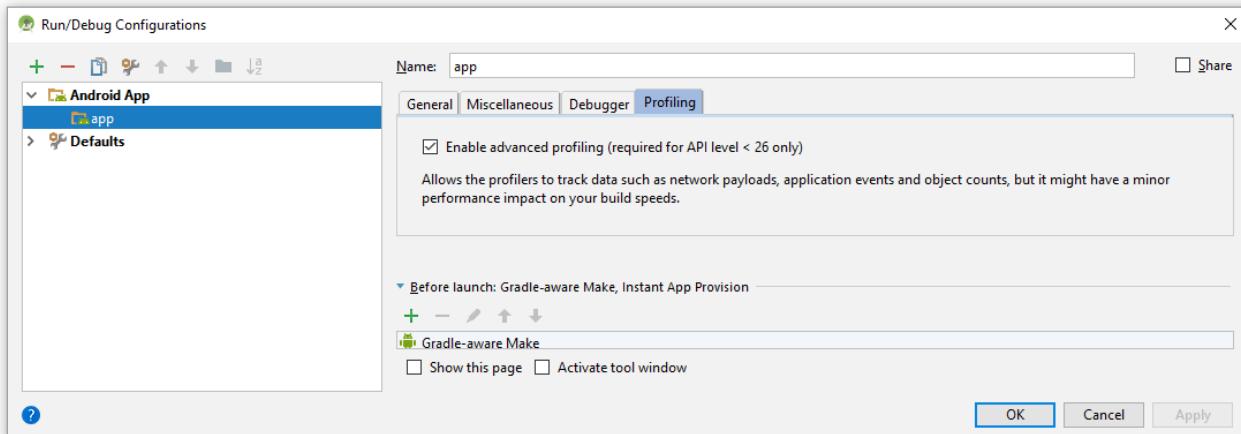
Läuft eine zu beobachtende App auf einem Gerät mit API-Level < 26, ist zunächst kein **Advanced Profiling** möglich:



In dieser Situation öffnet man die **Run**-Konfiguration zur App

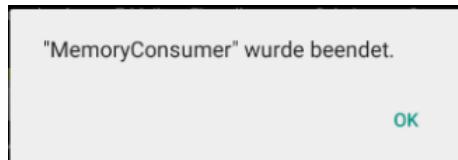


und markiert auf dem Registerblatt **Profiling** das Kontrollkästchen zum **advanced profiling**:



#### 6.4.4 OutOfMemoryError

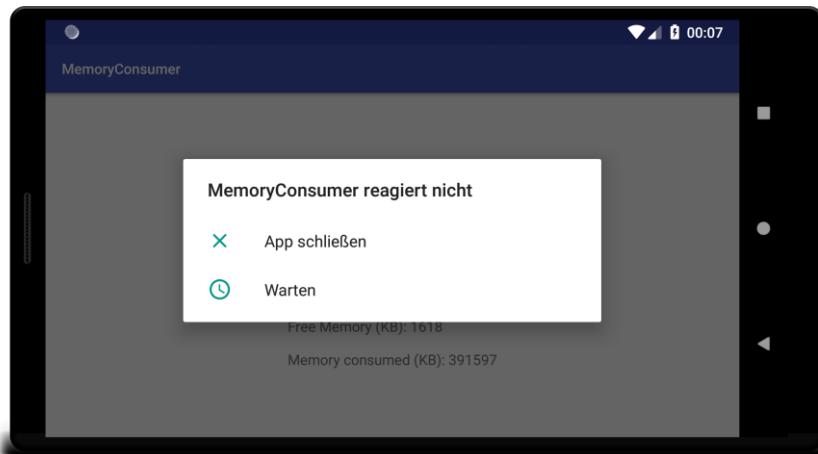
Wenn das Beispielprogramm mit seinen kumulierten Speicheranforderungen über das (z. B. mit der **ActivityManager** - Methode **getMemoryClass()** ermittelbare) Maximum hinausgeht, wird es von Android gestoppt. Bei Android 5.1 geschieht dies schnell und „gnadenlos“:



Im Logcat-Fenster wird ein **OutOfMemoryError** dokumentiert:

```
Process: de.zimkand.memoryconsumer, PID: 5351
java.lang.OutOfMemoryError: Failed to allocate a 400012 byte allocation with 3951588 free
bytes and 3MB until OOM; failed due to fragmentation (required contiguous free 401408 bytes
where largest contiguous free 0 bytes)
    at de.zimkand.memoryconsumer.MainActivity.onClick(MainActivity.java:28)
    at android.view.View.performClick(View.java:4780)
    at android.view.View$PerformClick.run(View.java:19866)
    at android.os.Handler.handleCallback(Handler.java:739)
    at android.os.Handler.dispatchMessage(Handler.java:95)
    at android.os.Looper.loop(Looper.java:135)
    at android.app.ActivityThread.main(ActivityThread.java:5254)
    at java.lang.reflect.Method.invoke(Native Method)
    at java.lang.reflect.Method.invoke(Method.java:372)
    at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:903)
    at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:698)
```

Auf einem emulierten Smartphone mit Android 8.1 läuft der Abgang etwas anders. Nach der Überschreitung der Speichergrenze bringt eine weitere Anforderung das Betriebssystem zur ANR-Fehlermeldung (Application Not Responding):



Die App verbleibt aber noch etliche Minuten im Hauptspeicher, während das System mehrere, im **Logcat**-Fenster dokumentierte Versuche startet, per Garbage Collecting (GC) freien Speicher zu gewinnen. Als Grund für den GC-Einsatz wird explizit eine über das erlaubte Maximum hinausgehende Heap-Anforderung angegeben (**GC Alloc**):

```

Starting a blocking GC Alloc
04-27 00:10:11.597 4656-4656/de.zimkand.memoryconsumer I/zygote: WaitForGcToComplete blocked Alloc on
HeapTrim for 2.160s
    Starting a blocking GC Alloc
04-27 00:10:11.597 4656-4661/de.zimkand.memoryconsumer I/zygote: WaitForGcToComplete blocked Alloc on
HeapTrim for 2.074s
    Starting a blocking GC Alloc
04-27 00:10:11.605 4656-4656/de.zimkand.memoryconsumer I/zygote: Waiting for a blocking GC Alloc
04-27 00:10:12.886 4656-4667/de.zimkand.memoryconsumer I/zygote: Clamp target GC heap from 388MB to 384MB
    Background concurrent copying GC freed 81922(1792KB) AllocSpace objects, 0(0B) LOS objects, 0% free,
382MB/384MB, paused 38us total 1.288s
04-27 00:10:12.886 4656-4665/de.zimkand.memoryconsumer I/zygote: WaitForGcToComplete blocked Alloc on
HeapTrim for 1.288s

```

Nach erfolglosen Bemühungen, per GC den Heap-Speicherbedarf wieder unter die Grenze (hier: 384 MB) zu bringen, kommt es schließlich doch zum Abbruch mit einem **OutOfMemoryError**:

```

04-27 00:10:14.130 4656-4656/de.zimkand.memoryconsumer D/AndroidRuntime: Shutting down VM
04-27 00:10:14.152 4656-4662/de.zimkand.memoryconsumer I/zygote:
Thread[3,tid=4662,WaitingInMainSignalCatcherLoop,Thread*=0xef50d800,peer=0x13081028,"Signal Catcher"]:
reacting to signal 3
04-27 00:10:14.159 4656-4656/de.zimkand.memoryconsumer E/AndroidRuntime: FATAL EXCEPTION: main
    Process: de.zimkand.memoryconsumer, PID: 4656
    java.lang.OutOfMemoryError: OutOfMemoryError thrown while trying to throw
    OutOfMemoryError; no stack trace available
04-27 00:10:14.160 4656-4665/de.zimkand.memoryconsumer E/System: java.lang.OutOfMemoryError:
    OutOfMemoryError thrown while trying to throw OutOfMemoryError; no stack trace available

```

## 6.5 Übungsaufgaben zu Kapitel 6

- 1) Wenn der Benutzer eine Aktivität mit dem Rückwärtsschalter verlässt, wird (indirekt) die Methode **finish()** aufgerufen. In der Zustandswechselmethode **onPause()** kann dieser explizite Abgang über den Rückgabewert **true** der Methode **isFinishing()** festgestellt werden. Demonstrieren Sie das bitte durch eine **Logcat**-Ausgabe (analog zum Beispielprogramm in Abschnitt 6.3.2.1).
- 2) Welche Lebenszyklus-Methoden werden bei einer im Vordergrund befindlichen Aktivität aufgerufen, wenn der Benutzer das Display aus- und anschließend wieder einschaltet?

## 7 Ressourcen

In Android ist es möglich und sehr zu empfehlen, die in der Programmiersprache Java realisierte Anwendungslogik strikt vom Erscheinungsbild einer App zu trennen. Das Erscheinungsbild wird bestimmt von sogenannten *Ressourcen*. Unter diesem Begriff subsumiert man Dateien mit Layoutdefinitionen, Zeichenfolgen, Menüdefinitionen, Animationsdefinitionen, Bitmaps und sonstige Binärdateien (z. B. mit Audios) usw. Die strikte Trennung von Code und Ressourcen hat gravierende Vorteile:

- Code und Ressourcen können unabhängig voneinander gewartet werden (eventuell durch **verschiedene Personen** mit speziellen Kompetenzen).
- Es ist leicht, alternative Ressourcen für **verschiedene Konfigurationen** (z. B. Display-Orientierungen, Sprachen) bereitzustellen, wobei das Ressourcensystem von Android die Verwaltung übernimmt, also insbesondere bei einer Ressource zwischen den verfügbaren Alternativen wählt.

Bei den in einer App verwendeten *Zeichenfolgen* lassen sich Verfahren und Vorteile der Externalisierung von Ressourcen besonders gut nachvollziehen:

- Wenn sich alle Zeichenfolgen in einer vom Code getrennten Datei befinden, ist die Übersetzung in andere Sprachen leicht möglich, wobei sprachkompetente Personen eingesetzt werden können, die keine Programmierkenntnisse benötigen.
- Auf den Android-Geräten der Endbenutzer kann das Betriebssystem dafür sorgen, dass die zur lokalen Spracheinstellung am besten passenden Zeichenfolgen verwendet werden.

### 7.1 Ressourcentypen

Die Ressourcen einer App werden im Projektordner **res** abgelegt. Dort existieren Unterordner, die jeweils Ressourcen eines bestimmten Typs aufnehmen (z. B. **drawable**, **layout**, **mipmap**, **values**).

In der folgenden Tabelle mit den wichtigsten Ressourcentypen finden sich einige Sorten, mit denen wir noch keine Erfahrungen gesammelt haben:<sup>1</sup>

Ressourcentyp	Beschreibung	res-Unterordner Dateityp
Eigenschafts-Animation	Bei dieser Animation wird eine Eigenschaft eines Objekts (z. B. die Hintergrundfarbe) zeitabhängig verändert.	<b>animator</b> <b>XML</b>
View-Animation	Hier ist zwischen der Tween- und der Frame-Animation zu unterscheiden. Bei der Tween-Animation wird ein grafisches Objekt transformiert (z. B. gedreht, verzerrt). Bei der Frame-Animation wird eine Sequenz von Bildern vorgeführt.	<b>anim</b> <b>XML</b>
Color State List	Es wird eine Liste von Farben für verschiedene Zustände eines <b>View</b> -Objekts (z. B. normal, fokussiert, gedrückt) definiert. Diese Liste kann einem <b>View</b> -Objekt wie eine Hintergrundfarbe zugewiesen werden.	<b>color</b> <b>XML</b>

<sup>1</sup> Weitere Informationen zu den von Android unterstützten Ressourcen-Typen sind hier zu finden:  
<https://developer.android.com/guide/topics/resources/providing-resources.html>  
<https://developer.android.com/guide/topics/resources/available-resources.html>

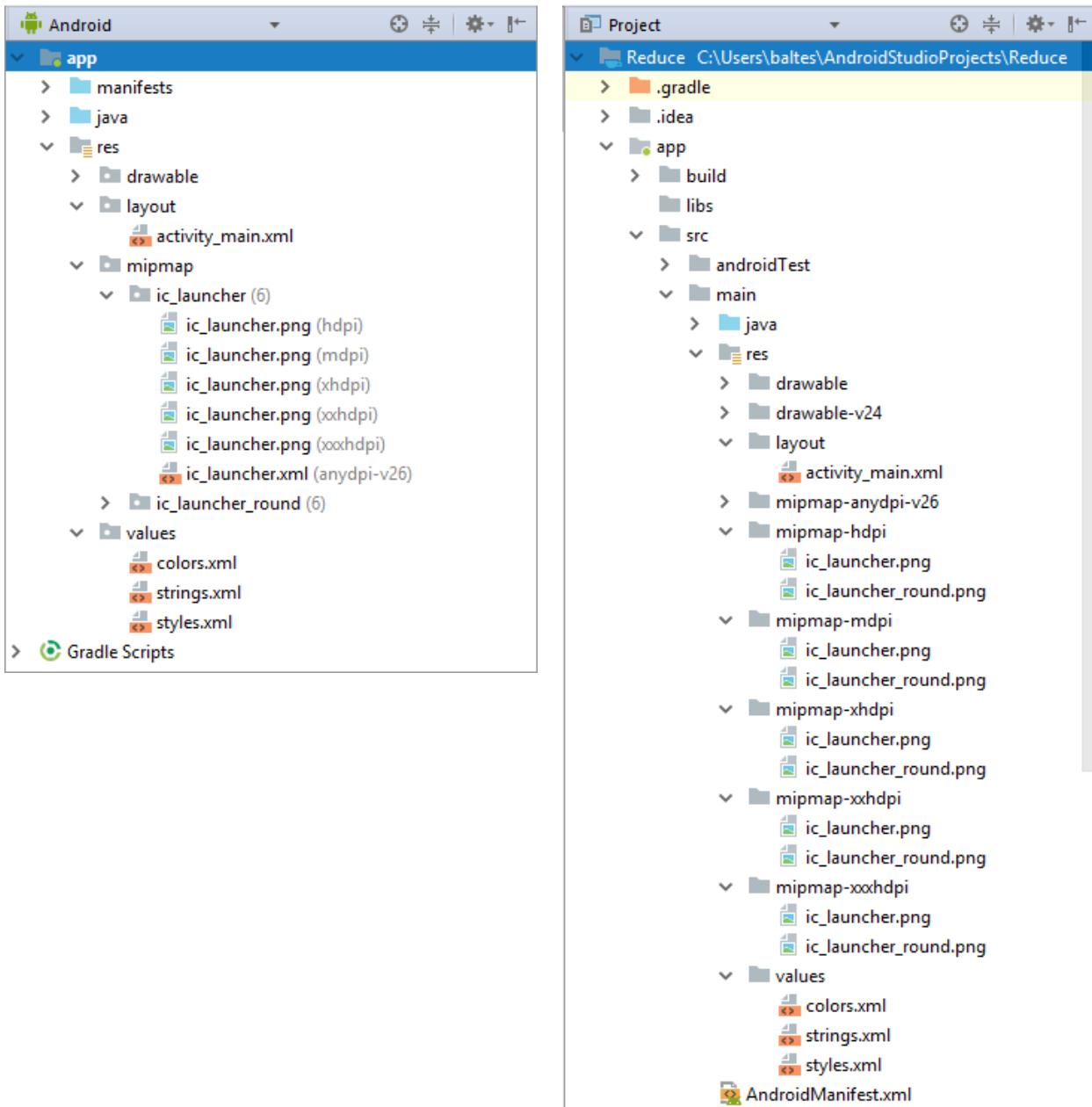
Ressourcentyp	Beschreibung	res-Unterordner Dateityp
Zeichenbare Ressourcen	<p>Hier geht es um Bitmap-Dateien (in den Formaten PNG, GIF und JPG) sowie andere zeichenbare Ressourcen, z. B.:</p> <ul style="list-style-type: none"> <li>• Neun-Feld-Dateien (engl. <i>nine-patch files</i>) Für eine PNG-Datei werden Bereiche festgelegt, die bei Größenveränderungen unverändert bleiben bzw. in vertikaler und/oder horizontaler Richtung gestreckt werden sollen.</li> <li>• Zustandslisten Für Zustände von Steuerelementen (z. B. normal, gedrückt) wird jeweils eine Bitmap-Datei vereinbart.</li> </ul> <p>Weitere Details werden in Abschnitt 7.7 behandelt.</p>	<b>drawable</b> JPG, PNG, etc.
Layoutdefinition	Das Layout einer Activity wird in einer XML-Datei definiert. Bei der Erstellung lassen wir uns meist vom GUI-Designer der Entwicklungsumgebung unterstützen.	<b>layout</b> XML
Menüdefinition	XML-Deklaration für ein Optionen- oder Kontextmenü	<b>menu</b> XML
Icons für den Anwendungsstarter (Launcher)	Icons für den Anwendungsstarter (engl. <i>launcher</i> ) von Android sollten von sonstigen zeichenbaren Ressourcen getrennt abgelegt werden. Wir arbeiten bisher mit den vom Android Studio automatisch erstellten Dateien <b>ic_launcher.png</b> und <b>ic_launcher_round.png</b> . Welche Datei tatsächlich als App-Icon verwendet wird, entscheidet das Zielsystem. Den in Android-Ressourcen häufig anzutreffenden Namensanfang „ <b>ic_</b> “ tragen Dateien mit Icons.	<b>mipmap</b> JPG, PNG, etc.
Beliebige Dateien	Hier werden beliebige Dateien abgelegt, die über eine Ressourcen-ID ansprechbar sein sollen (z. B. Audios, Videos).	<b>raw</b>
Einfache Ressourcen wie Zeichenfolgen, Dimensionsangaben, Stile	Alle bisher beschriebenen Ressourcen stecken als sogenannte <i>komplexe Ressourcen</i> jeweils in einer eigenen Datei (z. B. vom Typ XML oder PNG). Für einfache Ressourcen (z. B. statische Farbwerte, Zeichenfolgen) jeweils eine eigene Datei zu verwenden, wäre sehr unpraktisch. Stattdessen werden mehrere Werte in einer XML-Datei mit einem <b>resources</b> -Wurzelement zusammengefasst. Man verwendet allerdings in der Regel für jede Sorte von Einfachressourcen (z. B. Zeichenfolgen, Dimensionsangaben, Stile) jeweils eine eigene XML-Datei (z. B. <b>strings.xml</b> , <b>dimens.xml</b> , <b>styles.xml</b> ). Ein Stil fasst eine Reihe von Eigenschaftsausprägungen (z. B. Schriftart, Hintergrundfarbe) zusammen und kann <b>View</b> -Objekten zugewiesen werden.	<b>values</b> XML

Komplexe Ressourcen (z. B. Animationen, Bitmaps, Layoutdefinitionen) befinden sich in einer eigenen Datei. In der obigen Tabelle sind bis auf die letzte Zeile ausschließlich komplexe Ressourcen enthalten.

Der Projektexplorer im Android Studio bietet mehrere Sichten, von denen die beiden wichtigsten in den folgenden Bildschirmotos für das in Kapitel 4 erstellte Projekt **Reduce** zu sehen sind. Die voreingestellte, links dargestellte Sicht **Android** löst sich vom Aufbau des Projektordners im Dateisystem zugunsten einer inhaltlich-funktionalen Gliederung. Z. B. ist für das Launcher-Icon der App in der traditionellen rechteckigen Variante ein Knoten

### **res/mipmap/ic\_launcher**

mit mehreren gleichnamigen Dateien als Einträgen vorhanden, wobei jeweils in Klammern angegeben ist, für welche Display-Auflösung eine Icon-Datei gedacht ist (vgl. Abschnitt 7.2 zu konfigurationsabhängigen Ressourcen). In der alternativen Sicht **Project** ist die Ordnerstruktur des Dateisystems zu sehen. Dabei zeigt sich, dass sich die Icon-Dateien für den Launcher in Auflösungspezifischen Ordnern befinden:



Seit der Version 7.1 unterstützt Android auch **runde Icons**, und Google empfiehlt den Entwicklern, die Icon-Ressourcen in doppelter Ausfertigung zu liefern, um alle Android-Versionen optimal un-

terstützen zu können.<sup>1</sup> In der Manifestdatei (vgl. Abschnitt 5.2) sollte dementsprechend das **application**-Element die Attribute **android:icon** und **android:roundIcon** besitzen, z. B.:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="de.zimkand.reduce">
    <application
        android:icon="@mipmap/ic_launcher"
        android:roundIcon="@mipmap/ic_launcher_round"
        . . .
    </application>
</manifest>
```

Die **Android**-Ansicht des Projektexplorers zeigt bei empfehlungskonform erstellten Apps im Knoten **res/mipmap** die beiden Unterknoten **ic\_launcher** und **ic\_launcher\_round**. Auf den Endgeräten wählt das Ressourcensystem von Android die passende Variante, z. B.:

Das Icon zu unserer Reduce-App auf einem Gerät mit Android 5.1 (API 22)



Das Icon zu unserer Reduce-App auf einem Gerät mit Android 8.1 (API 27)



Seit der Version 8.0 beherrscht Android **adaptive Icons**, die es einem unterstützenden Launcher ermöglichen, die Icons an den lokal bevorzugten Look anzupassen und z. B. zwischen einer rechteckigen oder runden Form zu wählen.<sup>2</sup> Weil adaptive Icons eine deckend kolorierte Hintergrundschicht und eine Vordergrundschicht mit transparenten Bildanteilen besitzen, werden außerdem optische Effekte wie Schatten (siehe obigen Screenshot mit einem Android 8.1 - Startbildschirm) und Animationen ermöglicht.

In den vom Android Studio erstellten Projekten sind die adaptiven Icons folgendermaßen realisiert:

- Der Projektexplorer enthält die Knoten **app/res/mipmap/ic\_launcher** und **app/res/mipmap/ic\_launcher\_round** mit verschiedenen auflösungsabhängigen Qualifizierern (vgl. Abschnitt 7.2). Dort befinden sich PNG-Dateien mit rechteckigen und runden Icons, die von Android-Versionen mit einem API-Level < 26 verwendet werden sollen.
- Für das API-Level ab 26 enthalten die genannten Knoten mit dem Qualifizier **anydpi-v26** die XML-Dateien **ic\_launcher.xml** bzw. **ic\_launcher\_round.xml** mit einem Wurzelement namens **adaptive-icon**, z. B.:

```
<adaptive-icon xmlns:android="http://schemas.android.com/apk/res/android">
    <background android:drawable="@drawable/ic_launcher_background" />
    <foreground android:drawable="@drawable/ic_launcher_foreground" />
</adaptive-icon>
```

<sup>1</sup> <https://developer.android.com/about/versions/nougat/android-7.1.html>

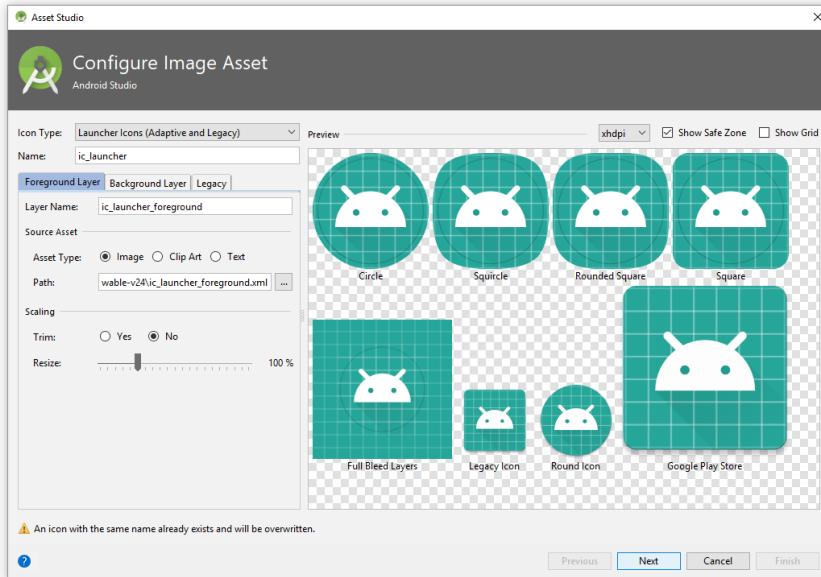
<sup>2</sup> [https://developer.android.com/guide/practices/ui\\_guidelines/icon\\_design\\_adaptive](https://developer.android.com/guide/practices/ui_guidelines/icon_design_adaptive)

- Die im XML-Element **adaptive-icon** als Hintergrund- bzw. Vordergrunddefinition vereinbarten XML-Dateien befinden sich im Projektexplorer-Knoten **app/res/drawable**.

Wir werden im Kurs keine Zeit finden, die vom Android Studio zu unseren Apps erzeugten Icon-Ressourcen näher zu erforschen und zu modifizieren. Wer hier kreativ werden möchte, findet in der Entwicklungsumgebung über den Menübefehl:

**File > New > Image Asset**

das passende Werkzeug mit Unterstützung für moderne Icon-Varianten:



In jedem Fall ist ein Launcher-Icon eine komplexe Ressource und in einer eigenen Datei untergebracht. Beim Zugriff auf komplexe Ressourcen sind die Dateinamen (ohne Extension) als Ressourcen-IDs zu verwenden (siehe Abschnitt 7.3), und bei den Dateinamen sind folgende Regeln einzuhalten:

- Die Menge der erlaubten Zeichen ist beschränkt: „a-z0-9\_“. Großbuchstaben sind verboten.
- Das erste Zeichen darf keine Ziffer sein, damit aus dem Dateinamen ein gültiger Java-Bezeichner entsteht.
- Bei zusammengesetzten Namen sollte ein trennender Unterstrich verwendet werden (siehe Beispiele).

Einfache Ressourcen (z. B. Größenangaben, Zeichenfolgen, Stildefinitionen) stecken in typspezifischen XML-Elementen (z. B. **dimen**, **string**, **style**), die sich als Kinder eines **resources**-Wurzelelements in einer XML-Datei befinden. Die vom Assistenten für neue Projekte angelegte Datei **strings.xml** aus unserem Projekt Reduce enthält z. B. die **String**-Ressource **app\_name**:<sup>1</sup>

```
<resources>
    <string name="app_name">Reduce</string>
</resources>
```

Auch die folgende Ressource vom Typ **style** ist eine Assistentenproduktion aus dem Projekt Reduce:

---

<sup>1</sup> Wir haben der Einfachheit halber bisher darauf verzichtet, weitere **String**-Ressourcen für in der Bedienoberfläche enthaltene Zeichenfolgen anzulegen, werden diesen Mangel aber gleich beheben

```
<resources>
    <!-- Base application theme. -->
    <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
        <!-- Customize your theme here. -->
        <item name="colorPrimary">@color/colorPrimary</item>
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
        <item name="colorAccent">@color/colorAccent</item>
    </style>
</resources>
```

Auf einem Basisstil aufbauend erhalten mehrere **item**-Elemente einen Namen und einen Wert. Beim GUI-Design kann man einem **View**-Objekt eine **style**-Ressource zuweisen, wobei alle Attribute des Objekts mit Werten versorgt werden, zu denen ein **item**-Element vorhanden ist. Die **style**-Ressource **AppTheme** dient allerdings dazu, in der Manifestdatei zum Projekt Reduce das Erscheinungsbild der gesamten App festzulegen:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="de.zimkand.reduce">
    <application
        android:theme="@style/AppTheme">
        .
        .
    </application>
</manifest>
```

Beim Zugriff auf einfache Ressourcen ist das **name**-Attribut aus der Definition zu verwenden (siehe Abschnitt 7.3). Das Android Studio bildet die Bezeichner meist nach den Regeln, die eben für die Namen von Dateien mit komplexen Ressourcen genannt wurden, macht gelegentlich aber Ausnahmen. Grundsätzlich ist jeder Java-Bezeichner erlaubt.

Die XML-Dateien mit einfachen Ressourcen befinden sich im Ordner **app/res/values** (Default-Varianten) oder in einem konfigurationsspezifischen Ordner (siehe Abschnitt 7.2). Es hat sich eingebürgert, für jede Sorte einfacher Ressourcen eine eigene Datei anzulegen und nach dem Typ der enthaltenen Ressourcen zu benennen (z. B. **colors.xml**, **strings.xml**, **styles.xml**).

Bei Farben treten komplexe *und* einfache Ressourcen auf: Während für eine Ressource vom Typ *Color State List* eine eigene XML-Datei im Ordner **res/color** angelegt wird, landen alle einfachen (statischen) Farbdefinitionen gemeinsam in der Datei **colors.xml** im Ordner **app/res/values**.

XML-Dateien sollten korrekterweise mit der folgenden Header-Zeile eingeleitet werden:

```
<?xml version="1.0" encoding="utf-8"?>
```

Die Assistenten im Android Studio halten sich jedoch nicht systematisch an diese Regel, und ein Verstoß bleibt ohne Folgen. Im Manuskript wird die Header-Zeile ab jetzt aus Platzgründen meist weggelassen.

## 7.2 Konfigurationsabhängige Ressourcen

Ein wesentlicher Grund für die nachdrückliche Empfehlung zur Externalisierung von Ressourcen ist die Bedingungsvielfalt, in der sich eine Android-App bewähren muss, z. B.:

- Unterschiedliche Länder- und Regionseinstellungen
- Unterschiedliche Größen, Seitenverhältnisse, Orientierungen und Auflösungen der Displays
- Unterschiedliche Android-Versionen

Wenn Ressourcen von Programmcode getrennt sind, kann das Android-Ressourcensystem die zur aktuellen Konfiguration passenden Varianten wählen.

Die Tabelle in Abschnitt 7.1 nennt zu jedem Ressourcentyp den Ordner für die *voreingestellte* Variante. Diese kommt für beliebige Konfigurationen (z. B. für beliebige Sprachen) zum Einsatz, wenn keine spezifische Variante verfügbar ist. Für konfigurationsspezifische Ressourcen (z. B. eine Datei **strings.xml** mit den Zeichenfolgen in Französisch) sind Ordnernamen zu verwenden, die nach dem Ressourcentyp noch **Qualifizierer** enthalten (z. B. **values-fr**). Das Ressourcensystem von Android versucht, zur Konfiguration des aktuellen Geräts passende Ressourcen zu finden. Wenn dies misslingt, verwendet es die voreingestellten Ressourcen. Daher *muss* z. B. bei einer App mit externalisierten Zeichenfolgen eine Datei namens **strings.xml** im Ordner **app/res/values** vorhanden sein, wobei in der Regel die englische Sprache verwendet wird. Übersetzungen in andere Sprachen können optional in den konfigurationsspezifischen Ordnern angeboten werden, z. B. eine Datei **strings.xml** mit deutschen Übersetzungen im Ordner **app/res/values-de**.

In der folgenden Tabelle sind wichtige Konfigurationen bzw. Qualifizierer erklärt:<sup>1</sup>

Konfiguration	Beschreibung	Qualifiziererwerte
Sprache und Region	Die Sprache wird per Doppelbuchstabencode definiert (nach ISO 639-1). Danach folgen optional ein Bindestrich, der Kleinbuchstabe <i>r</i> sowie ein Doppelbuchstabencode-Regionalcode (nach ISO 3166-1-alpha-2).	Beispiele: <b>en</b> <b>de</b> <b>fr-rFR</b> <b>fr-rCA</b>
Bildschirmgröße	Man kann die Bildschirmgröße über die Breite der schmalsten Seite oder über die aktuelle Breite definieren. Die Breite der schmalsten Seite spricht man nach dem Muster <b>sw&lt;N&gt;dp</b> an (mit <i>sw</i> für <i>smallest width</i> ). Google empfiehlt z. B.: <sup>2</sup> <ul style="list-style-type: none"> <li>• für 5" - Smartphones: <b>sw480dp</b></li> <li>• für 7" - Tablets: <b>sw600dp</b></li> <li>• für 10" - Tablets: <b>sw720dp</b></li> </ul> Man könnte z. B. XML-Dateien zur Layoutdefinition in den Ordnern <b>layout-sw600dp</b> und <b>layout-sw720dp</b> ablegen. Die aktuelle Breite spricht man nach dem Muster <b>w&lt;N&gt;dp</b> an, z. B.: <b>w820dp</b> .	Beispiele: <b>sw600dp</b> <b>sw720dp</b> <b>w820dp</b>
Bildschirmorientierung	Die beiden Werte stehen für den Portrait- bzw. Landschaftsmodus.	<b>port</b> <b>land</b>

<sup>1</sup> Eine vollständige Liste ist hier zu finden:

<https://developer.android.com/guide/topics/resources/providing-resources.html>

<sup>2</sup> <https://developer.android.com/training/multiscreen/screensizes.html>

Konfiguration	Beschreibung	Qualifiziererwerte
Pixel-Dichte des Displays	Weil die Pixeldichten von neuen Android-Geräten stetig zunehmen, wird die Definition von passenden Abkürzungen allmählich schwierig: <b>mdpi:</b> Medium density, ca. 160 dpi. <b>hdpi:</b> High density, ca. 240 dpi <b>xhdpi:</b> Extra high density, ca. 320 dpi <b>xxhdpi:</b> Extra high density, ca. 480 dpi <b>xxxhdpi:</b> ca. 640 dpi  Bei Bitmaps im Ordner <b>res/drawable</b> geht Android von einer Eignung für die mittlere Auflösung aus und skaliert sie ggf. hoch. Wir werden uns meist darauf beschränken, <b>mdpi</b> -Ressourcen zu erstellen.	<b>mdpi</b> <b>hdpi</b> <b>xhdpi</b> <b>xxhdpi</b> <b>xxxhdpi</b>
API-Level	Über diesen Qualifizierer kann man Ressourcen speziellen API-Levels zuordnen.	Beispiele: <b>v15</b> <b>v27</b>

Zu einem Ressourcentyp sind *mehrere* Qualifizierer erlaubt, die jeweils durch einen Bindestrich getrennt anzugeben sind. Dabei ist die Reihenfolge aus obiger Tabelle einzuhalten (Bach 2012, S. 152), sodass z. B. eine Bitmap (Ressourcentyp **drawable**) zur Verwendung auf hoch aufgelösten Displays durch französisch sprechende Kanadier im folgenden Projektexplorer-Knoten (in der **Android**-Sicht) anzulegen ist:

**app/res/drawable-fr-rCA-hdpi**

### 7.3 Ressourcen ansprechen

Zugriffe auf bereits definierte Ressourcen können an zwei Stellen erforderlich werden:

- in einer Ressourcendefinitionsdatei im XML-Format
- im Java-Quellcode

#### 7.3.1 Plattform-Ressourcen

Zu den bereits definierten Ressourcen gehören auch die in Android verfügbaren Plattform-Ressourcen. Einen Überblick liefert die Klasse **R** aus dem Paket **android** mit ihren diversen statischen Mitgliedsklassen (z. B. **R.strings**, **R.colors**).<sup>1</sup> Hier befinden sich statische und finalisierte **int**-Felder mit Ressourcen-IDs, die einen Zugriff auf die Plattform-Ressourcen im Java-Quellcode ermöglichen (siehe Abschnitt 7.3.3).

Definiert sind die Plattform-Ressourcen in XML-Dateien, die in einem **Sdk**-Ordner zu finden sind, z. B.:

**C:\Users\theo\AppData\Local\Android\Sdk\platforms\android-27\data\res**

#### 7.3.2 Zugriff in einer Ressourcendefinitionsdatei

Für einfache Ressourcen (z. B. Zeichenfolgen, Größenangaben) ist beim Zugriff das **name**-Attribut aus dem definierenden XML-Element zu verwenden. Für komplexe Ressourcen (z. B. Layoutdefinitionen, Bitmaps) ist beim Zugriff der Dateiname (ohne Extension) zu verwenden.

---

<sup>1</sup> <https://developer.android.com/reference/android/R.html>

In unserem Projekt **Reduce** (siehe Kapitel 4) haben wir bei der Beschriftung des Befehlsschalters bisher gegen den Rat des Android Studios gehandelt und eine „hart kodierte Zeichenfolge“ verwendet, wie ein Blick in die Layout-Definitionsdatei **activity\_main.xml** (eine komplexe Ressource, siehe Projektexplorer-Knoten **app/res/layout**) zeigt:

```
<Button
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="32dp"
    android:text="Reduce"
    android:textAllCaps="false"/>
```

Nun korrigieren wir unseren Fehler und ergänzen in der Ressourcendefinitionsdatei **strings.xml** (siehe Projektexplorer-Knoten **app/res/values**) im Vergleich zum Projektentwicklungsstand in Kapitel 4 eine Zeichenfolgen-Ressource mit dem Namen **reduce**:

```
<resources>
    <string name="app_name">Reduce</string>
    <string name="reduce">Reduce</string>
</resources>
```

Diese verwenden wir in der Layout-Definitionsdatei **activity\_main.xml**, um die Beschriftung des Befehlsschalters zu vereinbaren:

```
<Button
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="32dp"
    android:text="@string/reduce"
    android:textAllCaps="false"/>
```

Generell ist bei dieser Zugriffsart die Ressourcenbezeichnung folgendermaßen zu bilden:

**@[paket\_name:]ressourcen\_typ/ressourcen\_name**

Bei der hier gewählten Syntaxbeschreibung schließen die eckigen Klammern optionale Bestandteile ein. Der Paketname kann entfallen, wenn die verwendete Ressource aus demselben Paket stammt wie der Einsatzort. Erforderlich ist der Paketname z. B. bei der häufig praktizierten Verwendung von Ressourcen aus dem Paket **android**:

```
android:background="@android:drawable/editbox_background"
```

Eine Besonderheit besteht bei den Plattform-Ressourcen vom Typ **style** oder **theme** (definiert in den Dateien **styles.xml** bzw. **themes.xml**). Die Namen der in der Klasse **R.style** im Paket **android**<sup>1</sup> vorhandenen statischen **int**-Variablen zu diesen Ressourcen weichen von den Ressourcennamen in den XML-Dateien ab. Aus den Variablennamen in **R.style** (z. B.

**Widget\_ProgressBar\_Horizontal**) entstehen die in den XML-Dateien definieren und daher beim Zugriff zu verwendenden Ressourcennamen, indem alle Unterstriche durch Punkte ersetzt werden, z. B.:

```
style="@android:style/Widget.ProgressBar.Horizontal"
```

Beim GUI-Design bietet es sich oft an, ein Attribut für ein Steuerelement identisch mit dem entsprechenden Attribut aus dem aktuellen **Thema** (engl.: *theme*) zu wählen. Ein Thema ist ein Stil, der einer App oder einer Activity zugewiesen wird (zur Definition von Stilen siehe Abschnitt 7.1). Im Vergleich zum oben beschriebenen Ressourczugriff startet man mit einem Fragezeichen und kann den einzige zulässigen Ressourcentyp **attr** weglassen:

---

<sup>1</sup> <https://developer.android.com/reference/android/R.style.html>

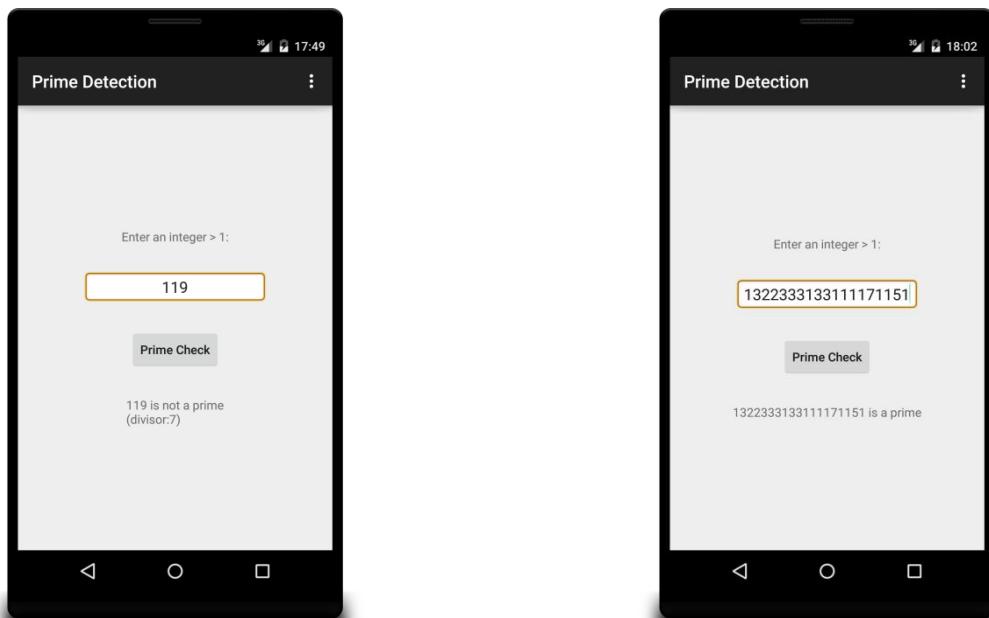
?[*paket\_name*:][*attr*/]*attribut\_name*

Das folgende Beispiel stammt von der Google-Webseite für Android-Entwickler:<sup>1</sup>

```
<EditText id="text"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:textColor="?android:textColorSecondary"
    android:text="@string/hello_world" />
```

### 7.3.3 Zugriff im Java-Quellcode

In Abschnitt 7.10 werden wir eine App zur Primzahlendiagnose erstellen, wobei die Beschriftung einer zum Ergebnisbericht dienenden **TextView**-Komponente bedingungsabhängig festgelegt werden muss, z. B.:



Wir werden geeignete Textbausteine als Ressourcen in der Datei **strings.xml** definieren:<sup>2</sup>

```
<resources>
    <string name="app_name">Prime Detection</string>
    <string name="prompt">Enter an integer > 1:</string>
    <string name="diag">Prime Check</string>
    <string name="prime">is a prime.</string>
    <string name="divisor">divisor:</string>
    <string name="noPrime">is not a prime.</string>
    <string name="falseArgument">is not an integer > 1.</string>
</resources>
```

Im Quellcode werden diese Ressourcen mit Hilfe der projekteigenen Klasse **R** angesprochen, die von der Entwicklungsumgebung (genauer gesagt: vom **Android Asset Packaging Tool, AAPT**) aus den Ressourcendefinitionsdateien des Projekts (im Beispiel: **strings.xml**) erstellt wird. Im folgenden Segment aus der **onClick()** - Methode der **Activity** - Klasse in der Primzahlen-App werden mehrere Zeichenfolgen mit Hilfe der Klasse **R** über ihre Ressourcen-IDs angesprochen:

<sup>1</sup> <http://developer.android.com/guide/topics/resources/accessing-resources.html>

<sup>2</sup> Falls Sie sich über das nicht-quotierte „>“ - Zeichen in der XML-Datei wundern (im **string**-Element **falseArgument**): An dieser Stelle ist alles erlaubt, außer „<“ und „&“.

```

if (cand <= 1)
    result.setText(s + " " + getString(R.string.falseArgument));
else
    if (df)
        result.setText(s + " " + getString(R.string.noPrime) + "\n(" +
                      getString(R.string.divisor) + Long.toString(i)+ ")");
    else
        result.setText(s + " " + getString(R.string.prime));

```

Bei den Ressourcen-IDs handelt es sich um die Werte von statischen und finalisierten **int**-Feldern (z. B. `R.string.falseArgument`) in statischen Mitgliedsklassen von **R**, z. B. in der Klasse **R.string**.<sup>1</sup>

```

public final class R {
    ...
    public static final class string {
        ...
        public static final int app_name=0x7f0b0011;
        public static final int diag=0x7f0b0012;
        public static final int divisor=0x7f0b0013;
        public static final int falseArgument=0x7f0b0014;
        public static final int hello_world=0x7f0b0015;
        public static final int noPrime=0x7f0b0016;
        public static final int prime=0x7f0b0017;
        ...
    }
    ...
}

```

Für den Zugang zu den Ressourcen ist prinzipiell ein Objekt aus der Klasse **Resources** zuständig. Seine Adresse ist für ein Aktivitätsobjekt über die Methode `getResources()` in Erfahrung zu bringen. Im folgenden Beispiel wird das **Resources**-Objekt aufgefordert, eine Zeichenketten-Ressource zu beschaffen:

```
String s1 = getResources().getString(R.string.diag);
```

Im konkreten Fall einer Zeichenketten-Ressource beherrscht das Aktivitätsobjekt allerdings eine Methode `getString()`, die bei geringerem Schreibaufwand dasselbe Ergebnis liefert:<sup>2</sup>

```
String s1 = getString(R.string.diag);
```

Analog verwendet man in vielen anderen Fällen Ressourcen-IDs ohne Direktkontakt mit der Klasse **Resources**. In der `onCreate()` - Methode unserer Apps wird der Methode `setContentView()` die Kennung einer Layout-Ressource übergeben, um daraus die Bedienoberfläche zu erstellen, z. B.:

```
setContentView(R.layout.activity_main);
```

Eine Bitmap - Ressourcen-ID eignet sich als Parameter für die Methode `setImageResource()` der Klasse **ImageView**, z. B.:

```
ImageView ball = new ImageView(this);
ball.setImageResource(R.drawable.ball);
```

<sup>1</sup> Eine statische Mitgliedsklasse wird der Übersichtlichkeit halber in eine Hüllenklasse verpackt und muss daher mit einem Doppelnamen angesprochen werden (z. B. `R.string`), ist aber ansonsten wie eine Top-Level - Klasse zu verwenden (siehe z. B. Baltes-Götz & Götz 2016, Abschnitt 4.9.1.2).

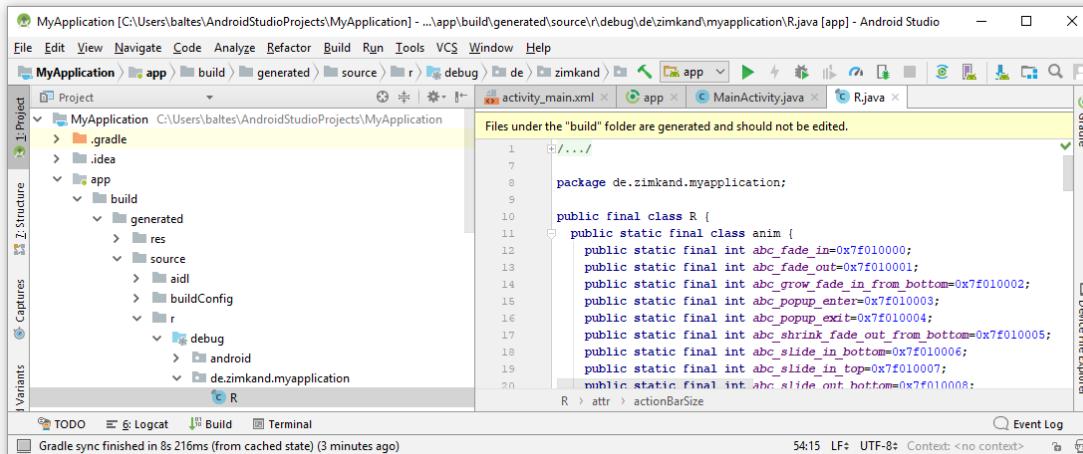
<sup>2</sup> Ein Blick in den Quellcode der Klasse **Context**, von der die Klasse **Activity** die Methoden `getResource()` und `getString()` erbt, zeigt, dass auch `getString()` auf das **Resources**-Objekt zugreift:

```

public final String getString(int id) {
    return getResources().getString(id);
}
```

Auch wenn zu einem Ressourcentyp (z. B. **drawable**) konfigurationsabhängige **res**-Unterordner existieren, die im Namen einen oder mehrere Qualifizierer enthalten, wird in der Ressourcen-ID grundsätzlich der Default-Ordnername angegeben. Schließlich soll die Wahl zwischen den alternativen Varianten zu einem Ressourcentyp dem Betriebssystem überlassen werden.

Die Klasse **R** wird beim Übersetzen des Projekts vom AAPT automatisch produziert, sodass manuelle Änderungen dieser Klasse sinnlos sind. Das Android Studio gibt sich einige Mühe, die Klasse **R** vor den neugierigen Blicken der Entwickler zu verstecken:



Die Feldnamen in den statischen Mitgliedsklassen von **R** stimmen bei einfachen Ressourcen (z. B. Zeichenfolgen) mit dem **name**-Attribut aus dem definierenden XML-Element überein (siehe z. B. Klasse **R.string**), bei komplexen Ressourcen (z. B. Layoutdefinition) mit dem Dateinamen (ohne Extension), z. B.:

```

public static final class layout {
    . . .
    public static final int activity_main=0x7f040016;
    . . .
}

```

### 7.3.4 Steuerelement-IDs

Die eben im Abschnitt 7.3.3 behandelte Klasse **R** enthält auch eine statische Mitgliedsklasse namens **id**, welche in statischen und finalisierten Feldern mit Zugriffsstufe **public** Element-IDs zur Identifikation von Steuerelementen bereit hält, z. B. in unserem Android-Projekt Reduce:

```

public final class R {
    . . .
    public static final class id {
        . . .
        public static final int button=0x7f090041;
        public static final int nenner=0x7f090040;
        public static final int zaehler=0x7f09003f;
        . . .
    }
    . . .
}

```

Wir benutzen diese Element-IDs z. B. als Parameter für die **Activity**-Methode **findViewById()**, um eine Referenz zu einem Steuerelement zu erhalten:

```
EditText etNenner = findViewById(R.id.nenner);
```

Vermutlich können Sie sich darin erinnern, dass wir diese Bezeichner im **Design**-Modus der Layout-Bearbeitung per **Attributes**-Fenster vereinbart haben.

Wer die XML-Layoutdefinition zu einer Aktivität direkt editieren möchte, muss dort eine spezielle Syntax zur Vereinbarung von Element-IDs verwenden, z. B.:

```
<EditText
    android:id="@+id/nenner"
    android:layout_width="200dp"
    android:layout_height="wrap_content"
    android:background="@android:drawable/editbox_background"
    android:ems="10"
    android:gravity="center_horizontal"
    android:hint="Enter Denominator"
    android:inputType="numberSigned" />
```

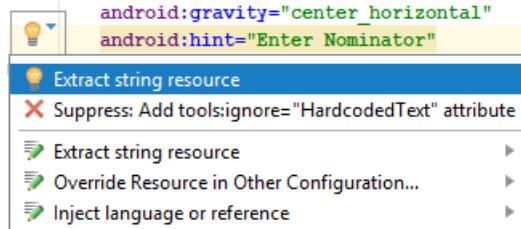
Mit `@+id` wird das AAPT angewiesen, eine neue Element-ID anzulegen.

## 7.4 Zeichenfolgen

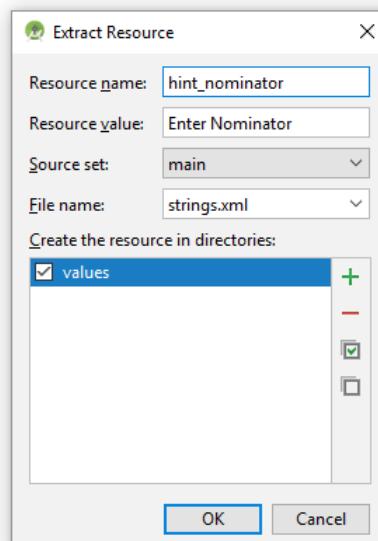
Im Project Reduce sind mittlerweile in der Datei `strings.xml` die beiden folgenden Zeichenfolgen-Ressourcen definiert:

```
<resources>
    <string name="app_name">Reduce</string>
    <string name="reduce">Reduce</string>
</resources>
```

Wir erstellen auch für die Hinwestexte zum Zähler bzw. Nenner jeweils eine Zeichenfolgenresourse und lassen uns dabei vom Android Studio per QuickFix unterstützen (vgl. Abschnitt 4.7.2), z. B.:



Nach Wahl des Vorschlags **Extract string resource** muss nur noch ein Name gewählt werden, z. B.:



In `strings.xml` resultiert der neue Zustand:

```
<resources>
    <string name="app_name">Reduce</string>
    <string name="reduce">Reduce</string>
    <string name="hint_nominator">Enter Nominator</string>
    <string name="hint_denominator">Enter Denominator</string>
</resources>
```

Nun erstellen wir eine Variante der Datei **strings.xml** mit den deutschen Übersetzungen im Projektexplorerknoten

### app/res/values-de

Zunächst wird an diesem Beispiel das generelle Vorgehen zum Erstellen einer konfigurationsspezifischen Ressourcendatei beschrieben. Später wird sich zeigen, dass im Android Studio zur Erstellung von sprachspezifischen Zeichenfolgen-Ressourcen noch ein besseres Werkzeug verfügbar ist.

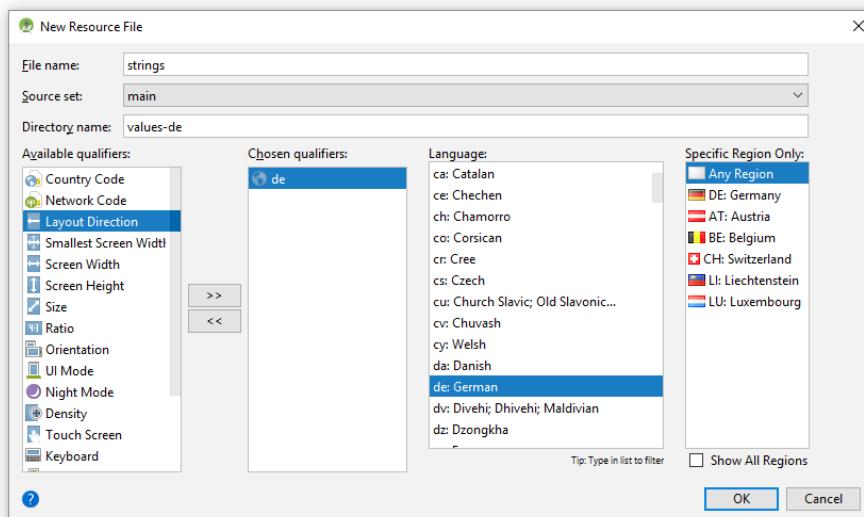
Wir wählen aus dem Kontextmenü zum Knoten

### app/res/values

in der **Android**-Sicht des Projektexplorers das Item

#### New > Values resource file

und tragen im folgenden Dialog

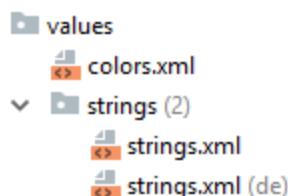


ein:

- **strings** als **File name** (Die Erweiterung **xml** wird automatisch angehängt.)
- **values-de** als **Directory name**

Statt den **Directory name** zu schreiben, kann man von den **Available qualifiers** die Option **Locale** mit dem Schalter **>>** aktivieren und dann bei **Language** und optional auch bei **Region** ein Listenelement auswählen.

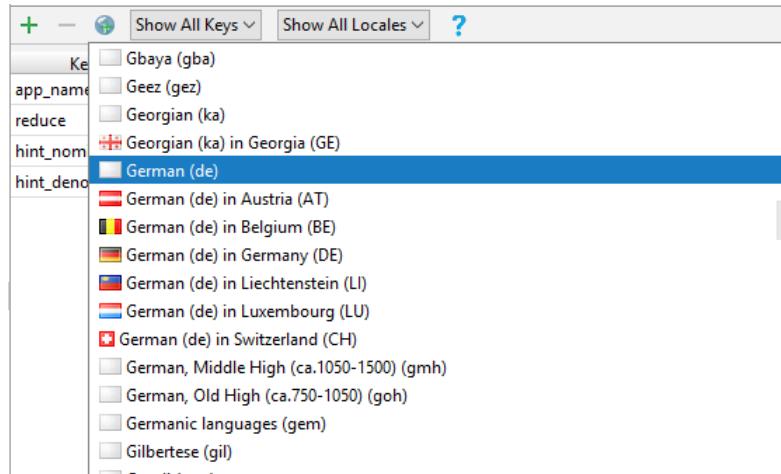
Nun kann man die neue **strings.xml** - Variante



per Doppelklick öffnen und übersetzte Zeichenfolgen eintragen.

Mit Hilfe des **Übersetzungseditors** im Android Studio lässt sich die Erstellung einer neuen Sprachversion für die Datei **strings.xml** erheblich vereinfachen:

- Öffnen Sie die Datei **strings.xml** per Doppelklick im Editor, und klicken Sie auf den Link **Open editor** in der rechten oberen Fensterecke.
- Nach einem Klick auf die Weltkugel können Sie eine neue Sprachversion ergänzen, z. B.:



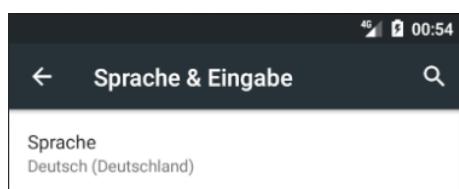
- Anschließend trägt man die Übersetzungen ein, z. B.:

Key	Resource Folder	Untranslatable	Default Value	German (de)
app_name	app\src\main\res	<input type="checkbox"/>	Reduce	Brüche kürzen
reduce	app\src\main\res	<input type="checkbox"/>	Reduce	Kürzen
hint_nominator	app\src\main\res	<input type="checkbox"/>	Enter Nominator	Zähler eintragen
hint_denominator	app\src\main\res	<input type="checkbox"/>	Enter Denominator	Nenner eintragen

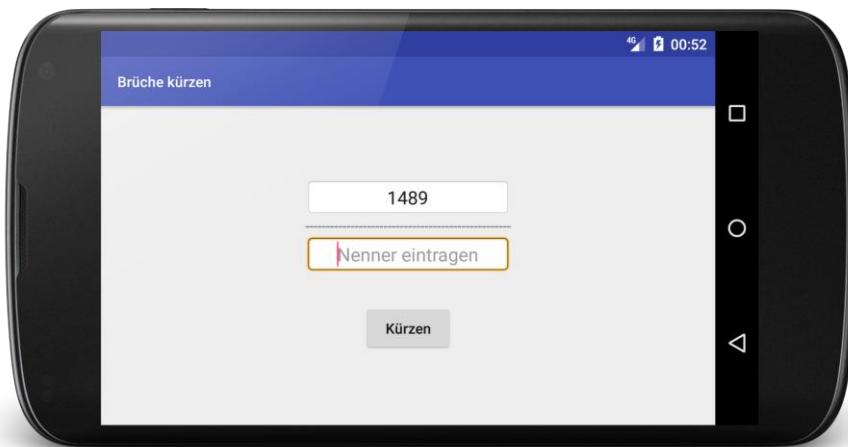
In der deutschen Sprachversion von **strings.xml** zeigt sich das erwartete Ergebnis:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">Reduce</string>
    <string name="reduce">Kürzen</string>
    <string name="hint_nominator">Zähler eintragen</string>
    <string name="hint_denominator">Nenner eintragen</string>
</resources>
```

Wenn die App anschließend auf einem Android-Gerät mit deutscher Spracheinstellung



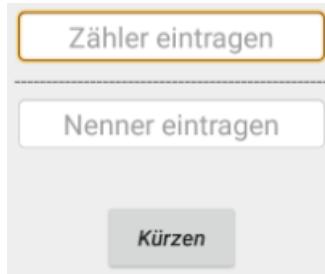
läuft (wählbar über **Settings > Language & input > Language**), dann spricht sie Deutsch:



Mit den von HTML gewohnten Elementen **<b>**, **<i>** und **<u>** für die Auszeichnungen **fett**, **kursiv** und **unterstrichen** lassen sich in **strings.xml** Textformatierungen vereinbaren, um z. B.

```
<string name="reduce"><i>Kürzen</i></string>
```

eine kursive Schalterbeschriftung zu erreichen:



Bei oft benötigten Standardbeschriftungen kann man etwas Aufwand sparen und **string**-Ressourcen aus dem Paket **android** (inkl. Übersetzungen) verwenden, z. B. mit der Ansprache

```
@android:string/cancel
```

Auf diese Weise sind u.a. die Ressourcen **ok**, **paste** (dt. Wert: *Einfügen*) und **search\_go** (dt. Wert: *Suchen*) verfügbar. Allerdings überraschen die **android**-Ressourcen **yes** und **no** mit den Werten *OK* und *Cancel*. Wie bereits erwähnt, sind die Definitionen der Plattform-Ressourcen in einem **Sdk**-Ordner zu finden, z. B.:

```
C:\Users\theo\AppData\Local\Android\Sdk\platforms\android-27\data\res
```

## 7.5 Größenangaben

Größenangaben (z. B. Randabstände) sollten in einer Ressourcendatei mit dem Namen **dimens.xml** gesammelt werden, um Einheitlichkeit und bequeme Modifizierbarkeit zu erreichen. Bei Größenangaben zu Elementen einer Android-Bedienoberfläche sind folgende Maßeinheiten zulässig:<sup>1</sup>

---

<sup>1</sup> Quelle: <http://developer.android.com/guide/topics/resources/more-resources.html#Dimension>

<b>px</b>	Ein reales Pixel Diese Maßeinheit ist <i>nicht</i> empfehlenswert, weil sich Android-Geräte bzgl. der Pixelgröße stark unterscheiden.
<b>dp</b> <b>dip</b>	Ein auflösungsunabhängiges Pixel ( <i>density independent pixel</i> ) entspricht einem realen Pixel auf einem Gerät mit einer Auflösung von 160 dpi, hat also eine Größe von ca. 0,16 mm: $\frac{25,4\text{mm}}{160} = 0,15875\text{mm}$ Bei Größen- bzw. Abstandsangaben zu Steuerelementen ist diese Einheit <b>dp</b> zu empfehlen.
<b>sp</b>	Bei der Einheit <b>sp</b> ( <i>scaled pixel</i> ) wird im Unterschied zur Einheit <b>dp</b> auch die vom Benutzer bevorzugte Schriftgröße ( <b>Einstellungen &gt; Anzeige &gt; Schriftgröße</b> ) berücksichtigt. Daher sollte die Spezifikation von Schriftgrößen in der Einheit <b>sp</b> erfolgen.
<b>pt</b>	Ein <b>pt</b> entspricht 1/72 Zoll, also ungefähr 0,35 mm.
<b>in</b>	Ein Inch (Zoll) entspricht ungefähr 25,4 mm.
<b>mm</b>	Millimeter

## 7.6 Farben

Zur Definition von Farben, die an mehreren Stellen in einer App (z. B. in mehreren Aktivitäten) verwendet werden, sollten Ressourcen in der Datei **colors.xml** definiert und an den betroffenen Stellen referenziert werden. Zur Definition einer Farbe wird das von HTML gewohnte vierkanalige ARGB-Schema verwendet (Transparenz-Rot-Grün-Blau), wobei der Alphakanal zur Transparenzdefinition weggelassen werden darf. Für jeden Kanal sind hexadezimale Werte von 00 bis FF erlaubt, wobei folgende Formate zulässig sind:

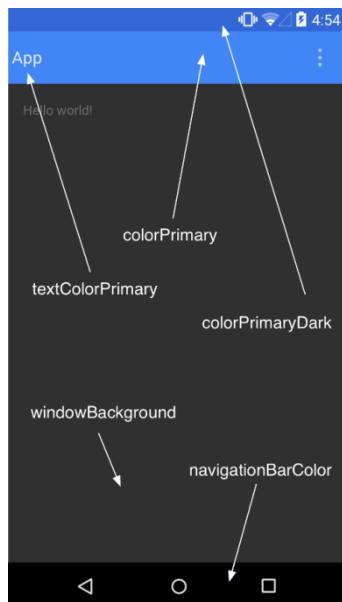
**#RGB**  
**#ARGB**  
**#RRGGBB**  
**#AARRGGBB**

Bei Verwendung der Kurzschreibweise wird jede hexadezimale Ziffer implizit verdoppelt, sodass z. B. #F00 für #FF0000 steht.

Im folgenden Beispiel wird durch eine Modifikation der vom Android Studio erstellten Datei **colors.xml** für die Bruchkürzungs-App die Farbe der Titelzeile verändert:

```
<resources>
    <color name="colorPrimary">#7AB53F</color>
    <color name="colorPrimaryDark">#303F9F</color>
    <color name="colorAccent">#FF4081</color>
</resources>
```

Wo diese Farbressourcen Verwendung finden, ist in der folgenden, von einer Google-Webseite für Android-Entwickler übernommen, Abbildung dokumentiert:



Im Beispiel resultiert das Ergebnis:



## 7.7 Zeichenbare Ressourcen

Mit *zeichenbaren Ressourcen* (engl.: *drawable resources*) sind Grafiken gemeint, die von einer App auf das Display gezeichnet werden. Von den vielfältigen Möglichkeiten zur Definition von zeichenbaren Ressourcen werden in diesem Abschnitt behandelt:

- Neun-Feld-Bitmaps
- Zustandslisten

Weitere Details sind z. B. in der Google-Online-Dokumentation für Android-Entwickler zu finden.<sup>1</sup>

### 7.7.1 Neun-Feld - Bitmaps

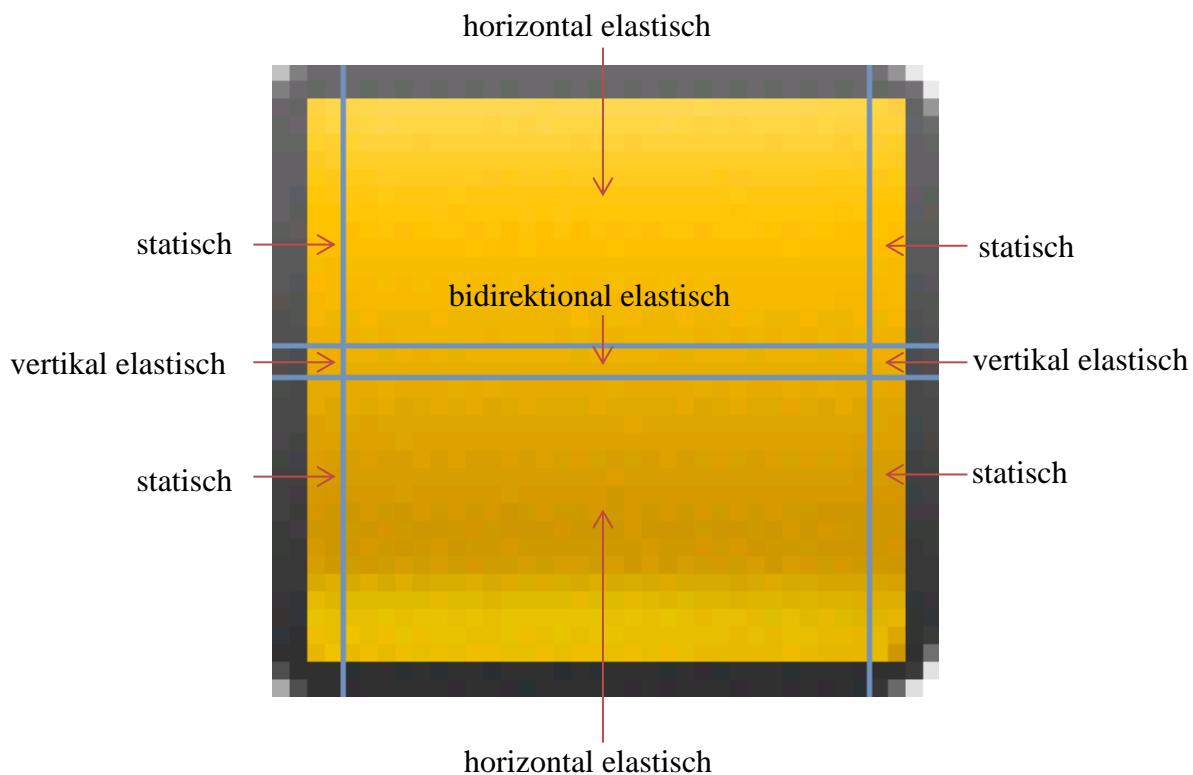
Zur Anpassung einer App an verschiedene Größenverhältnisse (z. B. bei einem Orientierungswechsel) müssen oft rechteckige Flächen (z. B. der Hintergrund eines Schalters) gedehnt oder gestaucht werden. Während das bei einfarbigen Flächen problemlos möglich ist, kann es bei einer gemusterten Fläche zu unästhetischen Verzerrungen kommen. Mit den Neun-Feld - Bitmaps wird das Ziel realisiert, eine gemusterte Fläche vergrößern oder verkleinern zu können, ohne zentrale Merkmale des Musters (z. B. die Eckengestaltung) zu beschädigen.

Eine Neun-Feld - Bitmap (engl. *nine-patch bitmap*) entsteht aus einer PNG-Datei, indem neun Rechtecke festgelegt werden, die bei einer Größenänderung ...

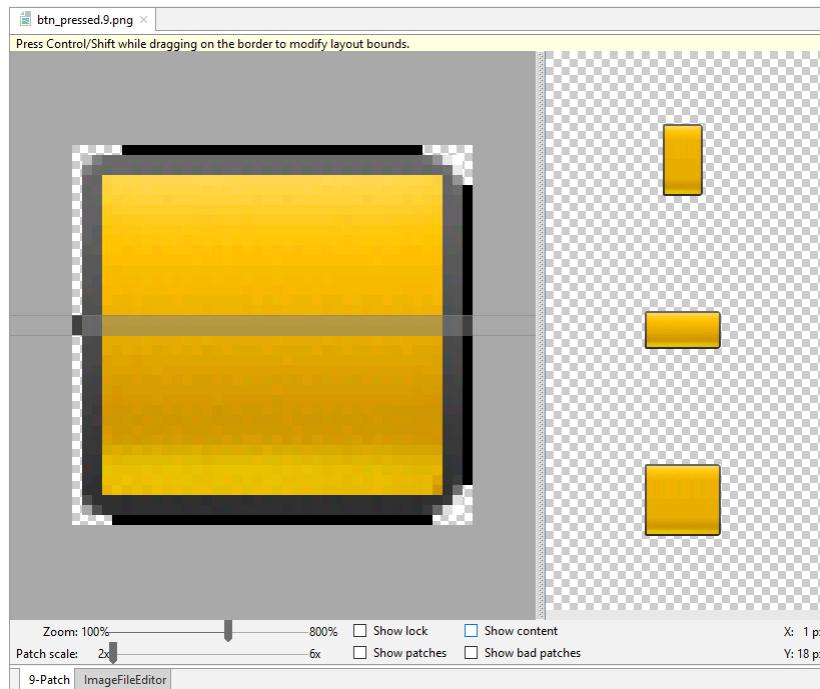
- statisch bleiben
- oder nur in vertikaler Richtung gestreckt bzw. gestaucht werden
- oder nur in horizontaler Richtung gestreckt bzw. gestaucht werden
- oder in beiden Richtungen gestreckt bzw. gestaucht werden

<sup>1</sup> <http://developer.android.com/guide/topics/resources/drawable-resource.html>

Die Ecken sind statisch, das Zentrum ist bidirektional elastisch, und die restlichen Zonen sind in einer Richtung elastisch:



Um diese 9 Zonen zu definieren, erhält eine PNG-Datei einen umlaufenden Rand mit einem Pixel Breite. Mit dem linksseitigen Rand wird ein Zeilenbereich ausgewählt, der bei einer vertikalen Vergrößerung gestreckt werden darf:

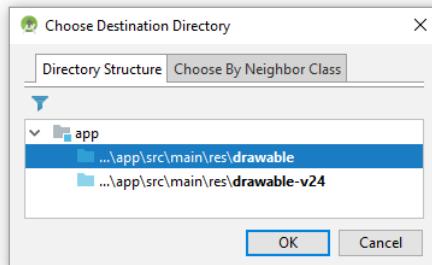


Mit dem oberen Rand wird ein Spaltenbereich ausgewählt, der bei einer horizontalen Vergrößerung gestreckt werden darf.

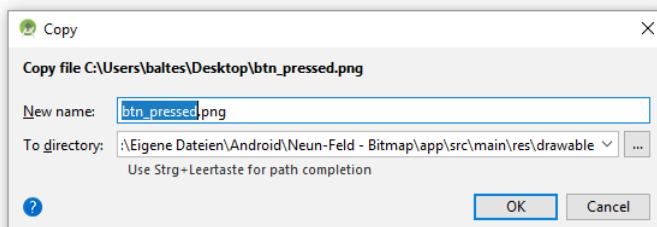
Die optionalen Markierungen am rechten und am unteren Rand tragen *nicht* zur Neun-Feld - Aufteilung bei, sondern definieren den freizuhaltenden Innenrand (engl.: *padding area*) bzw. den Inhaltsbereich (z. B. zur Aufnahme einer Beschriftung).

Zur Erstellung einer Neun-Feld - Bitmap mit dem Android Studio kopiert man zunächst die zugrundeliegende PNG-Datei in den Projektordnerknoten **app/res/drawable**, was unter Windows z. B. so zu erledigen ist:

- Per Windows-Explorer die zu importierende PNG-Datei in die Zwischenablage befördern.
- Im Projektexplorer den Zielknoten nötigenfalls aufklappen.
- Aus dem Kontextmenü zum **Zielknoten** das Item **Paste** wählen, um die Importdatei aus der Zwischenablage zu entnehmen.
- Den auflösungsabhängigen Zielordner wählen:

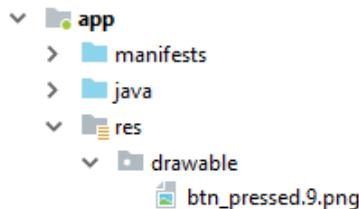


- Das Kopieren bestätigen:



Anschließend wählt man aus dem Kontextmenü zur aufgenommenen PNG-Datei das Item **Create 9-Patch file**, akzeptiert für die neue Datei den vorgeschlagenen Ordner **drawable** und wählt einen Namen (mit der Extension **.9.png**). Nach einem Doppelklick auf die neue Datei erscheint ein spezialisierte 9-Patch - Editor, der einen umlaufenden Rand mit einem Pixel Breite ergänzt, sodass die oben beschriebenen Zonendefinitionen vorgenommen werden können (siehe obiges Bildschirmfoto).

Eine Neun-Feld - Bitmap wird wie jede andere zeichenbare Ressource im Projektexplorerknoten **app/res/drawable** oder in einer auflösungsspezifischen Variante abgelegt, z. B.:



Diese zeichenbare Ressource kann einem Steuerelement als Wert für das Attribut **android:background** zugewiesen werden, z. B.:

```
<Button
    android:id="@+id/button"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_margin="32dp"
    android:background="@drawable	btn_pressed"
    android:text="@string/reduce"
    android:textAllCaps="false"
    android:textSize="32sp" />
```

Um Größenveränderungen zu provozieren, darf im Beispiel der Schalter die volle Breite des umgebenden Containers und die gesamte verfügbare Höhe nutzen (Wert **match\_parent** für die Attribute **layout\_width** und **layout\_height**). Außerdem hat der Schalter ....

- einen allseitigen Rand von 32 dp
- sowie eine Beschriftung mit der Größe 32 sp

erhalten. So sieht die Neun-Feld - Bitmap bei starker horizontaler Dehnung aus:



Das komplette AS-Projekt mit dem aktuellen Entwicklungsstand der Bruchkürzung-App ist im folgenden Ordner zu finden:

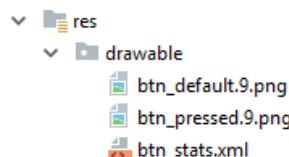
**...\\BspUeb\\Ressourcen\\Neun-Feld - Bitmap**

### 7.7.2 Zustandslisten

Bei der Hintergrundbemalung eines Schalters ist eine Anpassung an den aktuellen Zustand (z. B. normal, fokussiert, gedrückt) üblich. Analog zu den Color State Listen, die in Abschnitt 7.1 erwähnt wurden, lässt sich eine Zustandsliste definieren, die für jeden Zustand eine zeichenbare Ressource (z. B. eine Neun-Feld - Bitmap) definiert, z. B.:

```
<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:state_pressed="true"
          android:drawable="@drawable	btn_pressed"/> <!-- pressed -->
    <item android:drawable="@drawable	btn_default" /> <!-- default -->
</selector>
```

Eine XML-Datei mit einer Zustandsliste wird als zeichenbare Ressource im Ordner **res/drawable** abgelegt, z. B.:



Wird diese Zustandsliste einem Steuerelement als Wert des Attributs **background** zugewiesen,

```
<Button
    android:id="@+id/button"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_margin="32dp"
    android:background="@drawable/btn_stats"
    android:text="@string/reduce"
    android:textAllCaps="false"
    android:textSize="30sp" />
```

durchsucht Android für jeden Zustand die Liste der **item**-Elemente von oben nach unten und wählt die erste passende Ressource.

Anschließend ist für eine Variante der Bruchrechnungs-App mit der obigen Zustandsliste als Hintergrunddekoration für den Befehlsschalter der normale Zustand zu sehen:



Das komplette AS-Projekt mit dem aktuellen Entwicklungsstand der Bruchkürzung-App ist im folgenden Ordner zu finden:

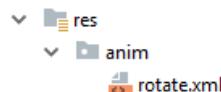
**...\\BspUeb\\Ressourcen\\Zustandsliste**

## 7.8 Animationen

Beim Thema *Animationen* müssen wir uns aus Zeitgründen auf ein einfaches Beispiel beschränken. In der Bruchkürzung-App soll sich der Schalter nach erfolgter Rechnung (am Ende der Klick-Behandlung) einmal komplett um seinen Schwerpunkt drehen:



Um diese Tween-Animation (vgl. Abschnitt 7.1 über Ressourcentypen) zu realisieren, legen wir den Projektexplorerknoten **app/res/anim** und darin die Datei **rotate.xml**



mit einer Animationsdefinition an:

```
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <rotate
        android:duration="1000"
        android:fromDegrees="0"
        android:toDegrees="360"
        android:pivotX="50%"
        android:pivotY="50%" />
</set>
```

Hier wird eine Rotation mit folgenden Attributen vereinbart:

- **android:duration**  
Dauer in Millisekunden
- **android:fromDegrees**  
Startwinkel
- **android:toDegrees**  
Endwinkel
- **android:pivotX, android:pivotY**  
Mit der Angabe 50% für die die X- und die Y-Koordinate des Drehpunkts wird eine Drehung um den Schwerpunkt veranlasst.

Im Quellcode der Aktivitätsklasse wird eine Instanzvariable vom Typ **Animation** definiert:

```
private Animation rotate;
```

In der Methode **onCreate()** wird die Animation unter Verwendung des Ressourcennamens (= Dateiname ohne Extension) mit Hilfe der statischen Mitgliedsklasse **R.anim** geladen:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Button button = findViewById(R.id.button);
    button.setOnClickListener(this);
    rotate = AnimationUtils.loadAnimation(this, R.anim.rotate);
}
```

Schließlich erhält der Befehlsschalter am Ende einer erfolgreichen Klickereignisbehandlung die Anweisung, die Animation auszuführen:

```
@Override
public void onClick(View v) {
    EditText etZahler = findViewById(R.id.zahler);
    EditText etNenner = findViewById(R.id.nenner);
    String sz = etZahler.getText().toString();
    String sn = etNenner.getText().toString();
    . . .
    int z = Integer.parseInt(sz);
    int n = Integer.parseInt(sn);
    if (z*n != 0) {
        . . .
        v.startAnimation(rotate);
    }
}
```

Ausführliche Informationen über Animationen in Android finden sich z. B. im Google-Webangebot für Android-Entwickler.<sup>1</sup>

Das komplette AS-Projekt mit dem aktuellen Entwicklungsstand der Bruchkürzungs-App ist im folgenden Ordner zu finden:

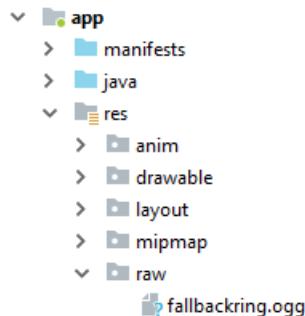
...\\BspUeb\\Ressourcen\\Animation

## 7.9 Mediendateien

Auch beim Umgang mit Medien (Audios, Videos) müssen wir uns aus Zeitgründen auf ein einfaches Beispiel beschränken. In der Bruchkürzungs-App soll der Befehlsschalter bei Betätigung eine akustische Rückmeldung geben. Dazu legen wir den Projektexplorerknoten **app/res/raw** an und kopieren aus den Android-SDK - Beständen die Datei **fallbackring.ogg**

...\\Sdk\\platforms\\android-27\\data\\res\\raw\\fallbackring.ogg

mit einem kurzen Sound im Ogg-Format dorthin:



Im Quellcode der Aktivitätsklasse wird eine Instanzvariable vom Typ **MediaPlayer** definiert:

**private MediaPlayer player;**

In der Methode **onCreate()** wird das **MediaPlayer**-Objekt kreiert und über die abzuspielende Datei unter Verwendung des Ressourcennamens unter Beteiligung der statischen Mitgliedsklasse **R.raw** informiert:

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Button button = findViewById(R.id.button);
    button.setOnClickListener(this);
    player = MediaPlayer.create(this, R.raw.fallbackring);
}
  
```

Schließlich erhält das **MediaPlayer**-Objekt am Ende einer erfolgreichen Klickereignisbehandlung zum Befehlsschalter die Anweisung, die Soundwiedergabe zu starten:

---

<sup>1</sup> <http://developer.android.com/guide/topics/resources/animation-resource.html>

```

@Override
public void onClick(View v) {
    EditText etZaehler = findViewById(R.id.zaeher);
    EditText etNenner = findViewById(R.id.nenner);
    String sz = etZaehler.getText().toString();
    String sn = etNenner.getText().toString();
    . . .
    int z = Integer.parseInt(sz);
    int n = Integer.parseInt(sn);
    if (z*n != 0) {
        . . .
        player.start();
    }
}

```

Ausführliche Informationen zur Klasse **MediaPlayer** bietet z. B. die SDK-Dokumentation.<sup>1</sup> Über die von Android unterstützten Medienformate informiert eine Google-Webseite für Android-Entwickler.<sup>2</sup>

Das komplette AS-Projekt mit dem aktuellen Entwicklungsstand der Bruchkürzungs-App ist im folgenden Ordner zu finden:

...\\BspUeb\\Ressourcen\\Medien

## 7.10 Entspannung und Motivationsstärkung

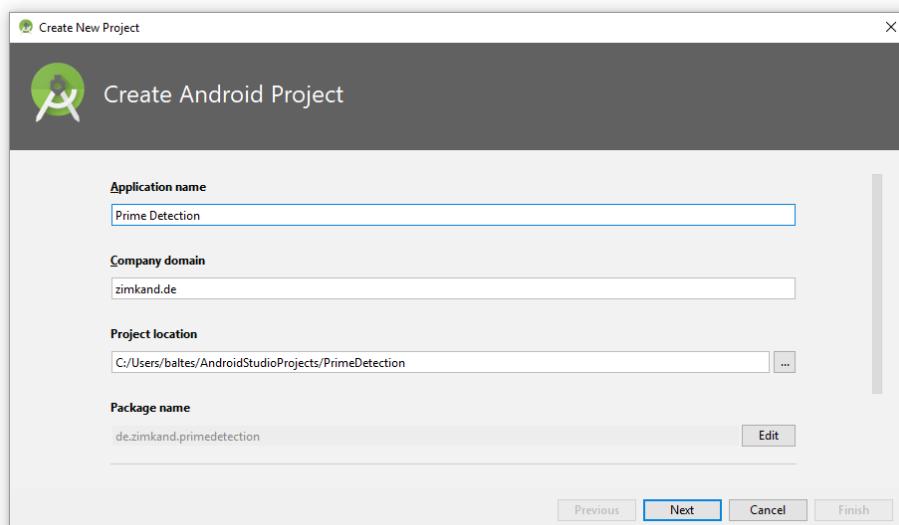
Nach etlichen recht anstrengenden Themen in den Kapiteln 5, 6 und 7 soll dieser Abschnitt zur Entspannung und zur Regeneration Ihrer Motivation beitragen. Wir erstellen eine Android-App, die für eine ganze Zahl größer 1 überprüft, ob es sich um eine Primzahl handelt.

### 7.10.1 Projekt anlegen

Wir legen ein neues Projekt an, starten also z. B. bei aktivem Android Studio mit dem Menübefehl

**File > New Project**

Im ersten Assistentendialog wählen wir **Prime Detection** als **Application name** und **zimkand.de** als **Company domain**:



<sup>1</sup> <http://developer.android.com/reference/android/media/MediaPlayer.html>

<sup>2</sup> <http://developer.android.com/guide/appendix/media-formats.html>

In den beiden folgenden Dialogen akzeptieren wir jeweils die Voreinstellung:

- Das Programm soll auf Smartphones und Tablets mit der minimalen Android-Version 4.0.3 verwendbar sein.
- Der Assistent soll eine **Empty Activity** anlegen.

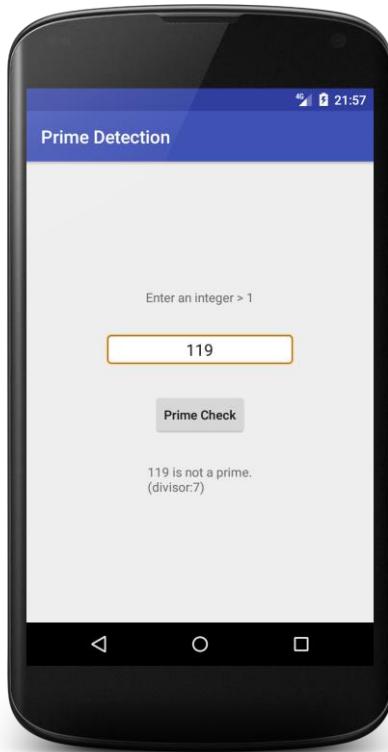
Im letzten Dialog wählen wir den Namen **PrimeActivity** für die Java-Klasse zur Aktivität und akzeptieren ansonsten die Voreinstellungen, z. B. ...

- den Namen **activity\_prime** für das Layout der Aktivität
- Abwärtskompatibilität durch die Wahl der Basisklasse **AppCompatActivity** für unsere Aktivität

Nach einem Klick auf den Schalter **Finish** beginnt das Build-System Gradle damit, das Gerüst der neuen App zu erstellen.

### 7.10.2 Bedienoberfläche entwerfen

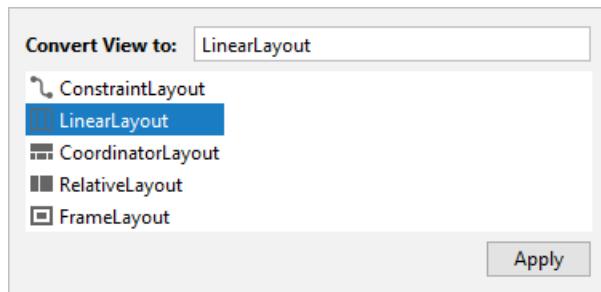
Öffnen Sie die Layoutdefinitionsdatei **activity\_prime.xml** zur Startaktivität im **Design**-Modus. Die zu erstellende Bedienoberfläche soll ungefähr so aussehen



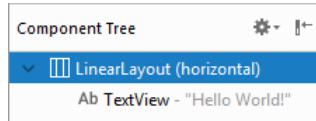
und folgende Steuerelemente (Controls) enthalten:

- ein Label (aus der Klasse **TextView**) für die Instruktion
- ein Texteingabefeld (aus der Klasse **EditText**) für den Primzahlkandidaten
- einen Schalter (aus der Klasse **Button**) zur Anforderung der Primzahlprüfung
- ein Label (aus der Klasse **TextView**) für die Ergebnisausgabe

Auch bei diesem Layout genügt als „Platzanweiser“ für die Steuerelemente ein Objekt der simplen Klasse **LinearLayout**. Wechseln Sie nötigenfalls zum **Design**-Modus des Layout-Editors, wählen Sie im **Component Tree** aus dem Kontextmenü zum vorhandenen **ConstraintLayout**-Manager das Item **Convert view**, und entscheiden Sie sich für die Klasse **LinearLayout**:



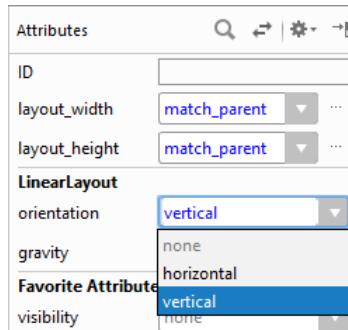
Im **Component Tree** - Fenster ist der neue Platzanweiser zu sehen:



Geben Sie dem **LinearLayout** im **Attributes**-Fenster, das bei Bedarf über die Schaltfläche



am rechten Fensterrand zu öffnen ist, eine vertikale Orientierung:



Setzen Sie außerdem die Eigenschaft **gravity** den Wert **center**, damit die im Layout enthaltenen Steuerelemente horizontal und vertikal zentriert werden.

Das vorhandene **TextView**-Objekt kann bleiben, benötigt aber den neuen Text „Enter an integer > 1“, der selbstverständlich über die Datei **strings.xml** im Projektexplorerknoten **app/res/values** als Zeichenfolgen-Ressource vereinbart werden sollte. Wir öffnen die Datei **strings.xml** per Doppelklick und sorgen dort für passende Beschriftungen (auch für weitere, noch anzulegende Steuerelemente):

- Zur Verwendung für das **TextView**-Objekt wird die folgende Zeichenfolgenressource definiert:
 

```
<string name="prompt">Enter an integer > 1</string>
```
- Dann werden noch weitere Zeichenfolgeressourcen ergänzt, sodass die schon in Abschnitt 7.3.3 als Beispiel verwendete Datei entsteht:<sup>1</sup>

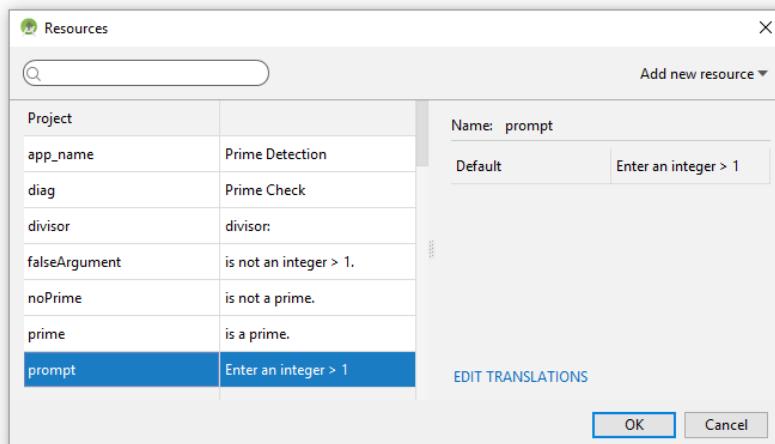
---

<sup>1</sup> Falls Sie sich über das nicht-gequerte „>“ - Zeichen in der XML-Datei wundern (Inhalt zum Element **prompt**): An dieser Stelle ist alles erlaubt, außer „<“ und „&“.

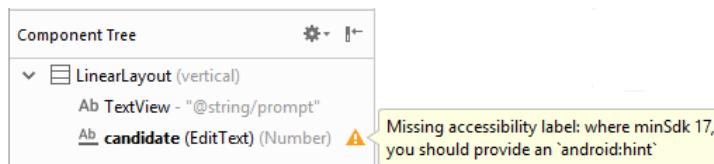
```
<resources>
    <string name="app_name">Prime Detection</string>
    <string name="prompt">Enter an integer > 1:</string>
    <string name="diag">Prime Check</string>
    <string name="prime">is a prime.</string>
    <string name="divisor">divisor:</string>
    <string name="noPrime">is not a prime.</string>
    <string name="falseArgument">is not an integer > 1.</string>
</resources>
```

Das **TextView**-Steuerelement kann auf folgende Weise mit der Zeichenfolgen-Ressource **prompt** versorgt werden:

- Element (im **Component Tree** oder in der Vorschau) markieren
- Nach einem Klick auf den Erweiterungsschalter zur Eigenschaft **text** im Attributes-Fenster die passende Zeichenfolge wählen und mit **OK** quittieren:



Wir machen weiter mit dem Texteingabefeld zur Aufnahme des Primzahlkandidaten. Dazu übernehmen wir aus der Abteilung **Text** der **Palette** ein Steuerelement vom Typ **Number**, fügen es per Drag & Drop im **Component Tree** unter dem vorhandenen **TextView**-Objekt ein und vergeben per **Attributes**-Fenster **candidate** als Element-ID:



Die Anmerkung zum fehlenden Hinweis dürfen wir diesmal ignorieren, weil eine Eingabeinstrukturion per **TextView**-Element vorhanden ist.

Das **EditText**-Objekt soll maximal 18 Ziffern erlauben, damit bei der Primzahldiagnose unter Verwendung des Datentyps **long** (maximaler Wert: 9223372036854775807) kein Überlauf auftritt. Dazu erhält seine Eigenschaft **maxLength** den Wert 18.

Setzen Sie die Eigenschaft **gravity** für das Textfeld auf den Wert **center\_horizontal**, damit der vom Benutzer eingetragene Wert horizontal zentriert erscheint. Die Breite des Textfelds wird übrigens per Voreinstellung über die Eigenschaft **ems** mit dem voreingestellten Wert 10 festgelegt:

**android:ems="10"**

Es ist ein Vielfaches der Breite des Buchstabens  $m$  anzugeben. Setzen Sie das Attribut **layout\_width** des Texteingabefelds auf den Wert **wrap\_content**. Fügen Sie einen oberen Rand von 32dp hinzu, der für Abstand vom **TextView**-Objekt sorgt (Eigenschaft **Layout\_margin, top**).<sup>1</sup>

Wir verschaffen dem Texteingabefeld eine attraktive Optik durch ein vom Android-System spendiertes Hintergrundbild (siehe Beschreibung in Abschnitt 4.2). Dazu markieren wir das Steuerelement, klicken im **Attributes**-Fenster auf die Zelle zur Eigenschaft **background** und dann auf den dort erscheinenden Erweiterungsschalter . Es öffnet sich das Fenster **Resources**, und wir wählen von den **android**-Ressourcen vom Typ **Drawable** die **editbox\_background** (eine Neun-Feld - Bitmap).

Weiter geht es mit dem Befehlsschalter, der die Primzahlenprüfung auslösen soll:

- Befördern Sie aus dem Paletten-Segment **Buttons** ein Steuerelement vom Typ **Button** an den unteren Rand des Layouts.
- Zur Beschriftung wird die Zeichenfolgen-Ressource **diag** verwendet (siehe obige Beschreibung zum **TextView**-Objekt).
- Setzen Sie die Eigenschaft **layout\_width** auf den Wert **wrap\_content**.
- Um zu verhindern, dass im laufenden Programm die Schalterbeschriftung komplett in Großschreibung erscheint, setzen wir die Schaltereigenschaft **textAllCaps** auf den Wert **false**, was erfahrungsgemäß (vgl. Abschnitt 4.2) einen Ausflug zur **Text**-Ansicht und eine manuelle Eintrag in der XML-Datei erfordert:  
**android:textAllCaps="false"**
- Als **ID** für den Schalter behalten wir die Vorgabe **button** bei.
- Fügen Sie einen oberen und einen unteren Rand ein, um den Schalter von den umgebenden Bedienelementen abzusetzen (Eigenschaft **Layout\_margin, top, bottom**).

An den unteren Rand des Layouts setzen wir aus der Paletten-Abteilung **Text** ein weiteres Objekt aus der Klasse **TextView**, das im Programm das Prüfergebnis anzeigen soll und daher **result** als **ID** erhält. Der überflüssige Initialtext wird gelöscht, und als **layout\_width** eignet sich der Wert **wrap\_content**.

Damit ist das zu Beginn des aktuellen Abschnitts beschriebene Designziel erreicht.

### 7.10.3 Click-Behandlungsmethode für den Schalter erstellen

Wechseln Sie in der Editorzone zum Java-Quellcode der Klasse **PrimeActivity**. Analog zu Abschnitt 4.3 sorgen wir dafür, dass die Benutzer nach einem Klick auf den Schalter die erwartete Auskunft erhalten.

Zunächst registrieren wir in der Methode **onCreate()** das handelnde Objekt unserer Aktivitätsklasse als zuständig für die Behandlung von Klick-Ereignissen beim Befehlsschalter:

---

<sup>1</sup> Sehr prinzipientreue Entwickler werden den Randabstand natürlich über eine Dimensions-Ressource integrieren (vgl. Abschnitt 7.5).

- Tragen Sie am Ende der Methode eine Anweisung ein, die uns eine Ansprachemöglichkeit für den Schalter verschafft:

```
Button button = findViewById(R.id.button);
```

Sorgen Sie nach Aufforderung durch das Android Studio dafür, dass die Klasse **Button** durch eine Importanweisung am Anfang der Quellcode datei bekannt gemacht wird.

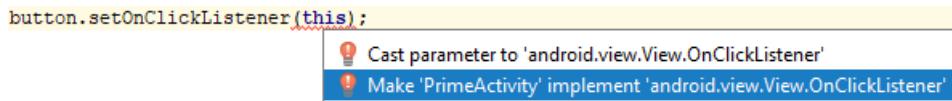
- Bitten Sie den Schalter darum, das Aktivitätsobjekt (angesprochen durch das Schlüsselwort **this**) als Klick-Interessenten zu registrieren:

```
button.setOnClickListener(this);
```

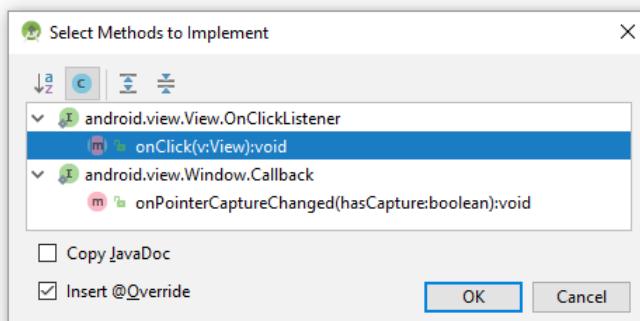
Damit diese Registrierung möglich ist, müssen zwei Voraussetzungen erfüllt sein:

- Die Aktivitätsklasse muss in ihrem Definitionskopf versichern, das Interface **OnClickListener** zu implementieren.
- Die Aktivitätsklasse muss ihr Versprechen einlösen und die Interface-Methode **onClick()** implementieren.

Die fällige Routinearbeit kann zum Teil vom Android Studio erledigt werden. Wir setzen die Einfügemarkierung auf die reklamierte Syntax (auf das Schlüsselwort **this**) und drücken die Tastenkombination **Alt+Enter**. Aus der erscheinenden Vorschlagsliste wählen wir das zweite Item, um den Klassendefinitionskopf zu erweitern:



Anschließend bietet der folgende Dialog an, einen Rohling für die benötigte Ereignisbehandlungs methode zu erstellen:<sup>1</sup>



Nach dem Quittieren mit **OK** ergänzt das Android Studio in der Klassendefinition:

```
@Override
public void onClick(View v) {
}
```

In dieser Methode werden wir den Text aus dem **EditText**-Steuerelement mit dem Primzahlkandidaten (Kennung: **candidate**) lesen und das Prüfergebnis in das **TextView**-Steuerelement am Ende des Layouts (Kennung: **result**) schreiben, sodass wir entsprechende Referenzvariablen benötigen. Weil sich an den Ressourcen-Kennungen bzw. Objektadressen nichts ändert, wäre es eine Zeitverschwendug, die Adressen bei jedem **onClick()** - Aufruf neu zu ermitteln. Diese Aufgabe wird daher in der Methode **onCreate()** erledigt.

---

<sup>1</sup> Hinter dem Angebot, die Methode **onPointerCaptureChanged()** zu implementieren, steckt die mit Android 8 eingeführte und nur für Rechner mit einer Maus als Eingabegerät (z. B. Chrome OS - Rechner, die Android-Apps ausführen) relevante Möglichkeit, alle Mausereignisse an ein Steuerelement zu kanalisieren. Wir können diese Option getrost ignorieren.

Weil die Referenzvariablen in mehreren Methoden (`onCreate()`, `onClick()`) benötigt werden, definieren wir sie als Instanzvariablen. Als Namen verwenden wir die Steuerelement-Kennungen:

```
public class PrimeActivity extends ActionBarActivity implements View.OnClickListener {
    private EditText candidate;
    private TextView result;
    . . .
}
```

In der Methode `onCreate()` wird die Verbindung zu den Steuerelementen der Bedienoberfläche hergestellt:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_prime);
    Button button = findViewById(R.id.button);
    button.setOnClickListener(this);
    candidate = findViewById(R.id.candidate);
    result = findViewById(R.id.result);
}
```

Jetzt muss noch die Klick-Ereignisbehandlungsmethode für den Schalter komplettiert werden, damit ein funktionstüchtiges Programm entsteht:

```
@Override
public void onClick(View arg0) {
    boolean df = false;
    long cand, i, mdc;
    String s = candidate.getText().toString();
    if (s.length() == 0)
        return;
    cand = Long.parseLong(s);
    mdc = (int) Math.sqrt(cand);
    for (i = 2; i <= mdc; i++)
        if (cand % i == 0) {
            df = true;
            break;
        }
    if (cand <= 1)
        result.setText(s + " " + getString(R.string.falseArgument));
    else
        if (df)
            result.setText(s + " " + getString(R.string.noPrime) + "\n" +
                          getString(R.string.divisor) + Long.toString(i) + ")");
        else
            result.setText(s + " " + getString(R.string.prime));
}
```

Aufgrund unserer GUI-Konfiguration in Abschnitt 7.10.2 (`inputType = "number",  
maxLength="18"`) liefert das `EditText`-Objekt garantiert eine nichtnegative ganze Zahl mit maximal 18 Stellen. Daher können wir auf eine Validierung der Eingabe verzichten.

Der Algorithmus zur Primzahlendiagnose prüft schlicht in einer `for`-Schleife von 2 bis zum größten möglichen Teiler per Modulo-Operator, ob sich der Kandidat durch den aktuellen Wert der Laufvariablen restfrei teilen lässt. Es sollte vielleicht noch erläutert werden, warum die Suche nach einem Teiler des Primzahlkandidaten bei seiner Wurzel enden kann (genauer: bei der größten ganzen Zahl  $\leq$  Wurzel):

Sei die ganze Zahl  $d$  ( $\geq 2$ ) ein echter Teiler der positiven, ganzen Zahl  $z$ , d.h. es gibt eine ganze Zahl  $k$  ( $\geq 2$ ) mit

$$z = k \cdot d$$

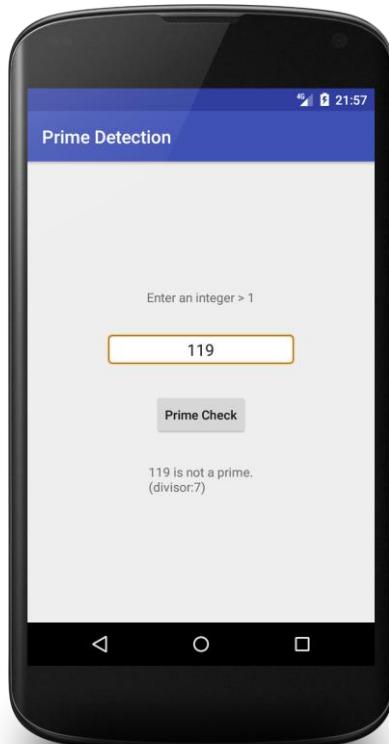
Dann ist auch  $k$  ein echter Teiler von  $z$ , und es gilt:

$$d \leq \sqrt{z} \text{ oder } k \leq \sqrt{z}$$

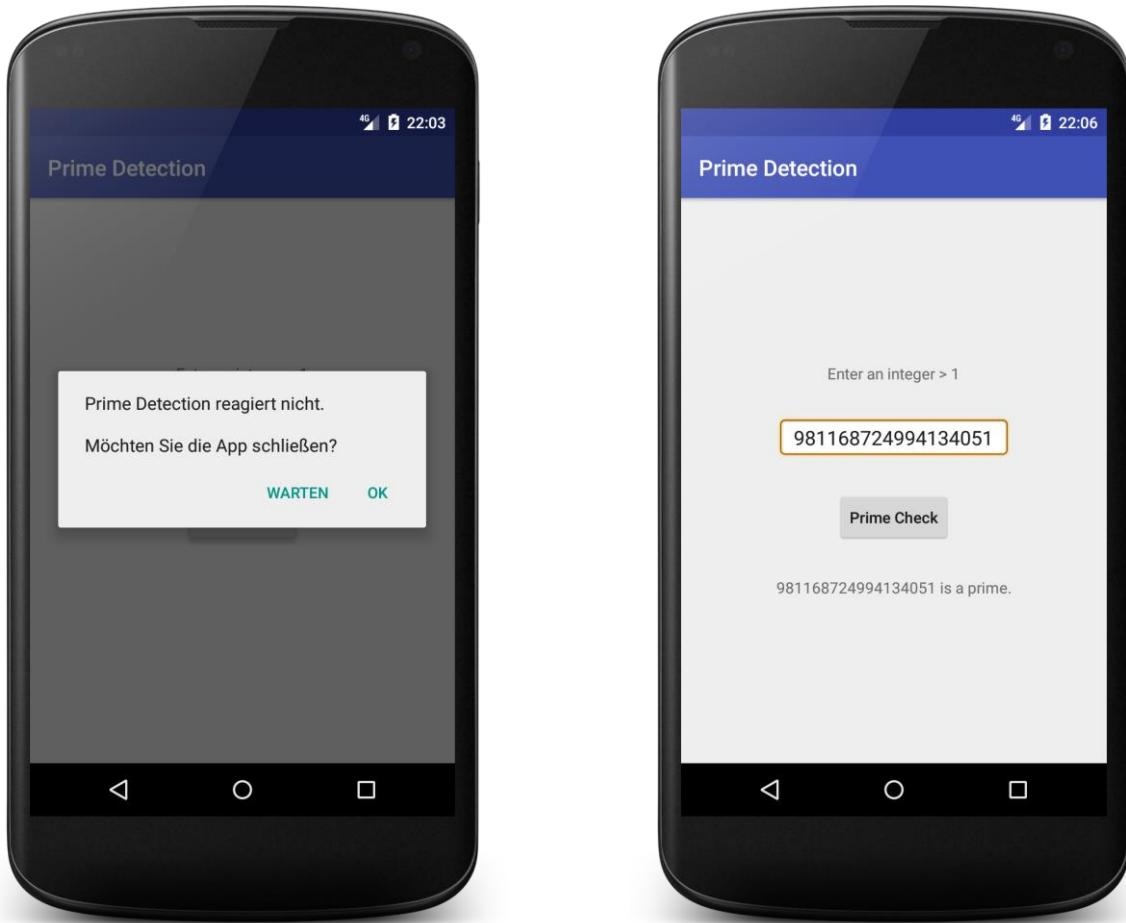
Andernfalls wäre das Produkt  $k \cdot d$  größer als  $z$ . Wir haben also folgendes Ergebnis: Wenn eine Zahl  $z$  einen echten Teiler hat, dann besitzt sie auch einen echten Teiler kleiner oder gleich  $\sqrt{z}$ . Wenn man *keinen* echten Teiler kleiner oder gleich  $\sqrt{z}$  gefunden hat, kann man die Suche also einstellen, und  $z$  ist eine Primzahl.

Zur Berechnung der Quadratwurzel verwendet das Beispielprogramm die statische Methode `sqrt()` aus der Klasse **Math**.

Die erste Bewährungsprobe übersteht das Programm ohne Kritikpunkte:



Sobald *größere* Primzahlzahlkandidaten (z. B. 981168724994134051) zu untersuchen sind, steigt der Zeitaufwand für die `onClick()` - Methode jedoch so stark an, dass Android mit einer ANR-Fehlermeldung (*Application Not Responding*) reagiert, wenn der Benutzer während der Bearbeitung ungeduldig erneut auf den Schalter klickt:



Wenn der Benutzer unserem Programm vertraut, wird er für sein **WARTEN** belohnt.

Unsere App kann kaum mit einer guten Bewertung rechnen, weil wir eine potentiell sehr zeitaufwändige Methode im UI-Thread (vgl. die Kapitel 8 und 10) ablaufen lassen, sodass die Bedienoberfläche länger als 5 Sekunden nicht mehr auf Eingaben reagiert und Android den ANR-Fehler meldet. Im Kapitel über Multithreading werden wir die Berechnung in einen separaten Thread auslagern und so das Problem beheben.

### 7.11 Übungsaufgaben zu Kapitel 7

- 1) Erstellen Sie eine deutsche Version der Zeichenfolgen für die in Abschnitt 7.10 entwickelte Primzahlen-App.

---

## 8 Bedienoberfläche

Mit Android wird die Gruppe der Java - GUI-Systeme (mit JavaFX, Swing, AWT, SWT etc.) um ein weiteres Mitglied erweitert. Allen Java - GUI-Systemen sind folgende Konstruktionsprinzipien gemeinsam:

- **Bibliothek mit Bedienelementen**

Es ist eine umfangreiche Bibliothek mit Bedienelementen zur Benutzerinteraktion vorhanden (z. B. Befehlsschalter, Kontrollkästchen, Texteingabefelder, Auswahllisten, Menüs).

Diese Bedienelemente bezeichnet man als *Steuerelemente*, *widgets* oder *controls*.<sup>1</sup> Es handelt sich um Java-Objekte mit der Besonderheit, dass sich auf das Display zeichnen und mit dem Benutzer interagieren.

- **Ereignisorientierung**

Das Laufzeitsystem bestimmt als Vermittler oder als Quelle von Ereignissen in erheblichem Maß den Ablauf einer Aktivität, indem es Ereignisbehandlungsmethoden der Aktivität aufruft. Diese Ereignisbehandlungsmethoden sind Beispiele für sogenannte *Call Back - Routinen*. Ausgelöst werden die Ereignisse in der Regel durch den Benutzer, der mit der Hilfe von Eingabegeräten wie Touch Screen oder Tastatur praktisch permanent in der Lage ist, unterschiedliche Wünsche zu artikulieren. Eine Aktivität präsentiert mehr oder weniger viele Bedienelemente und wartet die meiste Zeit darauf, dass eine der zugehörigen Ereignisbehandlungsmethoden durch ein vom Benutzer ausgelöstes Ereignis aufgerufen wird.

- **Single-Thread - Design**

Alle Ereignisbehandlungsmethoden laufen (strikt nacheinander!) im selben Thread (Ausführungsfaaden, siehe Kapitel 10) ab, sodass der Zeitaufwand für die Ausführung einer Ereignisbehandlungsmethode unbedingt in engen Grenzen bleiben muss. Potentiell zeitaufwändige Aufgaben (z. B. komplexe Berechnungen, Daten von einem Netzwerkdienst oder von einer Datenbank abrufen) müssen in einen separaten Thread verlagert werden. In Abschnitt 7.10.3 wurde demonstriert, dass der Aufruf einer zeitaufwändigen Methode im Ereignisbehandlungs-Thread (ab jetzt kurz: *UI-Thread*) zu einem ANR-Fehler führt (*Application Not Responding*). Auf die Steuerelemente einer Aktivität kann andererseits *nur* vom UI-Thread aus zugegriffen werden, sodass die aus einem anderen Thread initiierten Änderungen der Bedienoberfläche über Ereignisse im UI-Thread abgewickelt werden müssen. Zur Entscheidung für das Single-Thread – Design haben folgende Vorteile für die Bedienoberfläche geführt (vgl. Mednieks et al. 2013, S. 195):

- Hohe Performanz  
Die Flüssigkeit der Bedienoberfläche profitiert davon, dass keine Thread-Synchronisation erforderlich ist.
- Garantierte Konsistenz  
Bei parallel ablaufenden Ereignisbehandlungen wären inkonsistente Zustände der Bedienoberfläche schwer zu verhindern.

Wir haben in einigen Beispielprojekten (z. B. *Reduce*, *Prime Detection*) schon Erfahrungen beim Erstellen von Bedienoberflächen für Android-Aktivitäten gesammelt, die nun systematisiert und abgerundet werden sollen.

---

<sup>1</sup> Die Wortkombination *widget* aus *window* und *gadgets* steht für ein *Fenstergerät*. Von den hier gemeinten Steuerelementen sind die gleichnamigen speziellen Android-Apps mit Auftritt auf dem Startbildschirm zu unterscheiden.

## 8.1 Layoutdefinition per XML

Spätestens seit Kapitel 7 wissen wir, dass in Android eine Layoutdefinition für eine Aktivität als *Ressource* behandelt wird, die in einer XML-Datei im Projektordner **app/res/layout** abzulegen ist. Es ist auch möglich, ein Layout per Java-Quellcode zu definieren (so wie bei der Bedienoberfläche Swing für Java-Desktop-Programme).<sup>1</sup> Die Deklaration via XML ist aber wegen der folgenden Vorteile eindeutig zu bevorzugen:

- Die strikte Trennung von Layout und Programmlogik erleichtert die Entwicklung, vor allem bei Beteiligung von Programmier- *und* Grafikspezialisten.
- Die Unterstützung von verschiedenen Konfigurationen (z. B. Displaygrößen und -orientierungen) durch angepasste Layoutdefinitionen ist sehr einfach.

Der Aufbau einer Bedienoberfläche durch eine Hierarchie von verschachtelten Elementen, die entweder direkt mit dem Benutzer interagieren (**View**-Elemente) oder unsichtbar im Hintergrund zur Verwaltung von Elementen dienen ( **ViewGroup**-Elemente), ist vielen Entwicklern aus der Gestaltung von Webseiten per HTML bekannt.

Anschließend ist die Layoutdefinition zur Activity im Projekt **Prime Detection** zu sehen, die wir in Abschnitt 7.10.2 mit Hilfe der Entwicklungsumgebung erstellt haben:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center"
    android:orientation="vertical"
    tools:context=".PrimeActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/prompt" />

    <EditText
        android:id="@+id/candidate"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="32dp"
        android:background="@android:drawable/editbox_background"
        android:ems="10"
        android:gravity="center_horizontal"
        android:inputType="number"
        android:maxLength="18" />
```

---

<sup>1</sup> Mittlerweile hat auch in die Java-Standard-Edition mit *JavaFX* eine GUI-Technik Einzug gehalten, die eine strenge Trennung von GUI-Design (bevorzugt in XML zu erstellen) und Programmlogik unterstützt (siehe z. B. Baltes & Baltes-Götz 2018). Microsoft ist diesen Weg beim *Windows Presentation Framework* (WPF) für die .NET-Plattform mit der *Extended Application Markup Language* (XAML) schon etwas früher gegangen (siehe z. B. Baltes-Götz 2017).

```

<Button
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginBottom="32dp"
    android:layout_marginTop="32dp"
    android:text="@string/diag"
    android:textAllCaps="false" />

<TextView
    android:id="@+id/result"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

</LinearLayout>

```

Als Wurzelement der XML-Datei kommt meist ein **ViewGroup**-Objekt (ein Layout-Manager) zum Einsatz, doch ist auch ein **View**-Objekt erlaubt (siehe Abschnitt 8.2 zur Klassenhierarchie mit den Klassen **View** und **ViewGroup**).

Im Wurzelement werden XML-Namensräume definiert:

```

xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:app="http://schemas.android.com/apk/res-auto"
xmlns:tools="http://schemas.android.com/tools"

```

Der zu einem XML-Namensraum angegebene URI ist keine reale Internet-Adresse, sondern dient nur zur eindeutigen Identifizierung von Attributen.<sup>1</sup> Zur Vermeidung von Namenskollisionen wird den meisten Attributen in XML-Dateien aus dem Android-Umfeld ein (per Doppelpunkt abgeschlossener) Namensraumpräfix vorangestellt (im Beispiel: **android**, **app** oder **tools**).

Im Beispiel ist als Wurzelement ein Container vom Typ **LinearLayout** vorhanden, der als Kind-Elemente atomare Steuerelemente aus den folgenden Klassen enthält und linear (im konkreten Fall: vertikal) anordnet:

- **TextView**
- **EditText**
- **Button**

Aus den Elementen der XML-Layoutdefinitionsdatei entsteht beim Erstellen des Projekts eine Layout-Ressource, und beim Laden einer Layout-Ressource zur Laufzeit (meist in der **onCreate()**-Methode einer Aktivität) entsteht eine Hierarchie von Objekten, die zur Klasse **View** oder zu einer Ableitung gehören (siehe Abschnitt 8.2).

Den Attributen der XML-Elemente entsprechen *Eigenschaften* der Klassen. Zu einer Eigenschaft gehört nicht unbedingt eine einzelne Instanzvariable, aber in der Regel ein Paar von Methoden für den lesenden bzw. schreibenden Zugriff auf die Eigenschaft. Man kann die Klassen aus der **View**-Hierarchie den sogenannten *JavaBeans* zurechnen, weil sie für ihre Eigenschaften Zugriffsmethoden mit genormten Namen bieten:

- Auf die Einleitung durch *get* bzw. *set* folgt der Name der Eigenschaft (z. B. **getText()**, **setText()**).
- Bei Eigenschaften mit dem Datentyp **boolean** wird die Abfragemethode mit *is* eingeleitet (z. B. **isChecked()**).

---

<sup>1</sup> Ein URI (*Universal Resource Identifier*) sieht aus wie ein URL (*Universal Resource Locator*), zeigt aber nicht unbedingt auf ein reales Objekt im Internet.

Beim „Aufpumpen“ der Aktivitäts-Bedienoberfläche aus einer Layout-Ressource wird ein Objekt der Klasse **LayoutInflater** tätig, das wir meist indirekt durch einen Aufruf der **Activity**-Methode **setContentView()** beauftragen. Dieser Methode ist als Parameter die Ressourcen-ID der Layoutdefinitionsdatei (im Beispiel: **activity\_prime.xml**) zu übergeben, die aus einer statischen Variablen der vom AAPT (Android Asset Packaging Tool) automatisch erstellten Klasse **R.layout** entnommen wird, wobei der Variablenname mit dem Dateinamen ohne Namenserweiterung übereinstimmt. In der Regel findet der **setContentView()** - Aufruf im Rahmen der **Activity**-Initialisierungsmethode **onCreate()** statt, z. B.:

```
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_prime);  
    . . .  
}
```

Weil das Android Studio einen sehr guten Layout-Editor besitzt, ist der direkte Kontakt mit der XML-Layoutdefinition nicht unbedingt erforderlich. Allerdings ist manchmal die direkte Modifikation der (leicht verständlichen) XML-Syntax der schnellste Weg zum Ziel.

Oft ist es erforderlich, für verschiedene Gerätegrößen bzw. -orientierungen eigenständige Layoutdefinitionen anzulegen (siehe Abschnitt 7.2 über konfigurationsspezifische Ressourcen). Im Abschnitt 8.3.1 werden wir z. B. eine für das Querformat optimierte Alternative zum obigen Layout definieren.

## 8.2 Steuerelementklassen

### 8.2.1 Die View - Klassenhierarchie

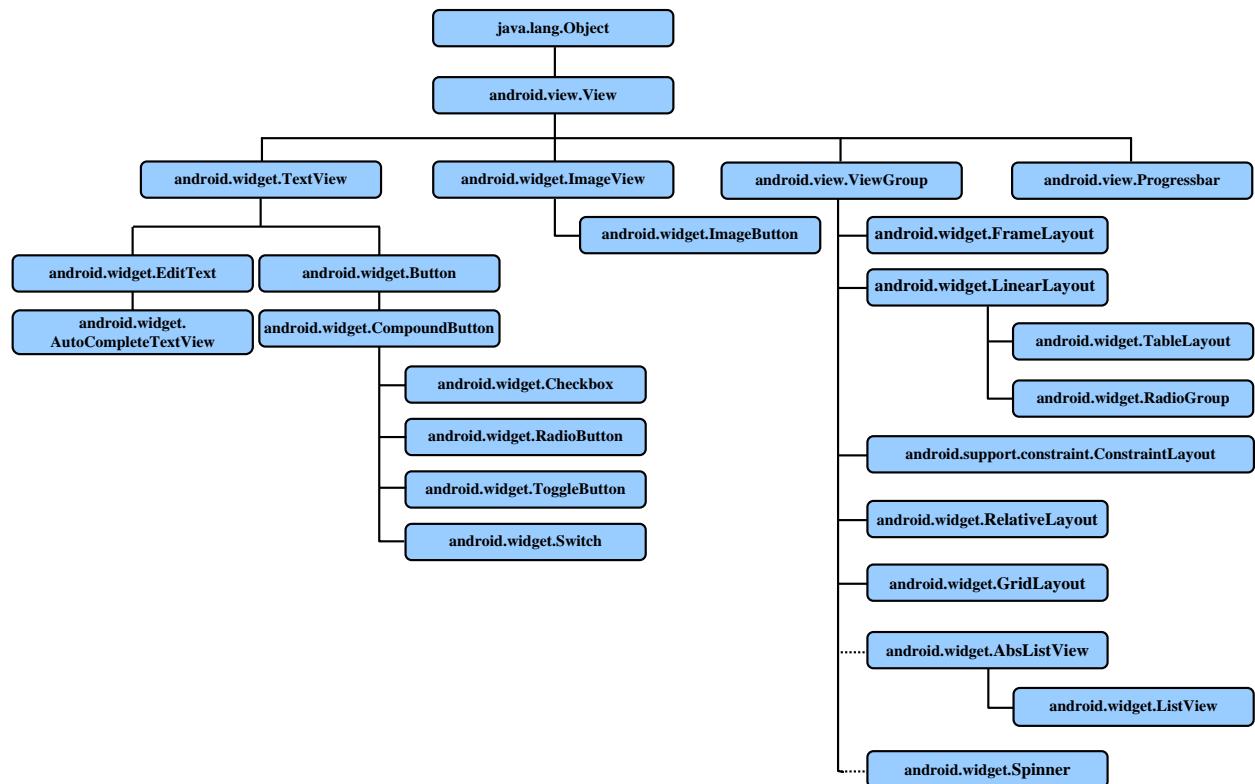
Die Steuerelemente zur Realisation einer Bedienoberfläche sind Objekte von Klassen, die im Unterschied zu gewöhnlichen Klassen (z. B. **String**) einige zusätzliche Kompetenzen besitzen:

- **Visuelle Steuerelemente** treten auf dem Bildschirm in Erscheinung und kommunizieren mit dem Benutzer (reagieren z. B. auf Touch- oder Tastatur-Ereignisse).
- Aus elementaren **Ereignissen** (z. B. Beginn bzw. Ende einer Touch Screen - Berührung) erstellen sie anwendungsrelevante Ereignisse (z. B. Schalter betätigt), über die sich andere Objekte informieren lassen können (siehe Abschnitt 8.4).
- Sie unterstützen die Verwendung mit Hilfe von **Entwicklungsumgebungen**. Durch die systematische Verfügbarkeit von Methoden zum Lesen und Setzen von Eigenschaften und die Einhaltung der JavaBeans-Benennungsregeln kann eine Entwicklungsumgebung z. B. eine Tabelle mit den Eigenschaftsausprägungen zur Entwurfszeit anbieten.

In der folgenden Abbildung sehen Sie die Abstammungsverhältnisse der im Kurs bis zum Ende des aktuellen Kapitels behandelten Steuerelementklassen:<sup>1</sup>

---

<sup>1</sup> Zwischen den Klassen **ViewGroup** und **Spinner** bzw. **AbsListView** wurden der Einfachheit halber Zwischenstufen der Vererbungshierarchie weggelassen.



Während sich die Klassen **View** und **ViewGroup** im Paket **android.view** befinden, gehören die meisten anderen Klassen zum Paket **android.widget**. Eine Ausnahme stellt das top-aktuelle **ConstraintLayout** im Paket **android.support.constraint** dar.

Jedes Objekt aus der Klasse **View** oder einer **View**-Ableitung repräsentiert eine rechteckige Fläche auf dem Bildschirm.

Von **ViewGroup** abstammende Klassen können als Container für **View**-Objekte fungieren. Die Bedienoberfläche einer Activity besteht aus einer Hierarchie von **View**-Objekten.

Wenn die Bedienoberfläche neu gezeichnet werden muss, wird die Aufforderung entlang der Äste des **View**-Baums bis zu den Blättern propagiert, die dann für ihre Darstellung sorgen. Android beherrscht allerdings einige Techniken, um überflüssigen Aufwand wie z. B. die Neuausgabe von unveränderten Display-Bereichen zu vermeiden (siehe Mednieks et al. 2013, S. 193).

## 8.2.2 Elementare Steuerelementattribute bzw. -eigenschaften

Alle Steuerelemente haben zahlreiche Attribute, von denen die wichtigsten in diesem Abschnitt besprochen werden. Einem XML-Attribut entspricht in der Regel ein Paar aus einer `get`- und einer `set`-Methode, die den Zugriff per Programm, also während der Laufzeit erlauben. Ob die Eigenschaft (engl. *property*) hinter einem Paar von Zugriffsmethoden durch eine einzelne Instanzvariable realisiert wird, liegt nach dem objektorientierten Prinzip der Datenkapselung im Verborgenen und spielt für den Anwendungsprogrammierer keine Rolle.

### 8.2.2.1 Layout-Parameter

Welche Attribute zur Positionierung und Dimensionierung eines Steuerelements verfügbar sind, hängt wesentlich vom Typ des enthaltenden Containers ab. Man spricht hier von den *Layout-Parametern*, und diese sind in einer Mitgliedsklasse des Container-Typs definiert. Gemeinsame Basisklasse der Layout-Parameter - Klassen ist **ViewGroup.LayoutParams**.

Jedes View-Objekt besitzt eine Instanzvariable namens **mLayoutParams**,

```
protected ViewGroup.LayoutParams mLayoutParams;
```

die auf den zuständigen Layout-Manager zeigt:

```
public void setLayoutParams(ViewGroup.LayoutParams params) {
    if (params == null) {
        throw new NullPointerException("Layout parameters cannot be null");
    }
    mLayoutParams = params;
    resolveLayoutParams();
    if (mParent instanceof ViewGroup) {
        ((ViewGroup) mParent).onSetLayoutParams(this, params);
    }
    requestLayout();
}
```

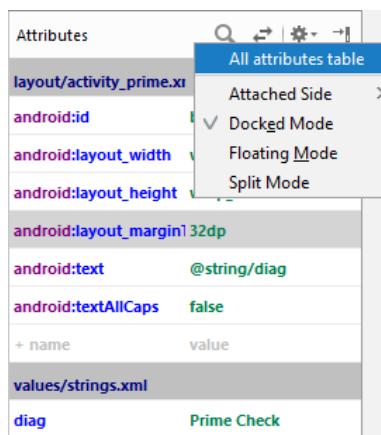
So erfährt der Container, wie das Element positioniert und dimensioniert werden möchte.

Die zu Layout-Parametern gehörigen XML-Attribute besitzen einen Namen mit der Einleitung „**layout\_**“, z. B. **layout\_width**.

### 8.2.2.2 Attributes-Fenster des Layout-Editors im Android Studio

Das **Attributes**-Fenster des Layout-Editors im Android Studio zeigt die XML-Attribute zu einem Steuerelement per Voreinstellung in folgender Anordnung:

- Ganz oben stehen die **ID** sowie die Layout-Parameter **layout\_width** und **layout\_height**, die (bis auf wenige Ausnahmen) für *jedes* Element definiert werden müssen.
- Es folgen wichtige Steuerelementattribute nach Klassen in der Vererbungshierarchie geordnet.
- Über den Schalter werden weniger wichtige Attribute ein- bzw. ausgeblendet. Wenn nach einem Wechsel zur vollständigen Liste nur die beim markierten Steuerelement modifizierten Attribute zu sehen sind, müssen Sie aus dem Drop-Down-Menü zum Schalter das Item **All Attributes table** wählen:



### 8.2.2.3 Steuerelement-ID

Wie ein Steuerelement im Quellcode über seine Element-ID anzusprechen ist, wurde schon in Abschnitt 7.3.4 erklärt. Eine Activity beherrscht dazu die Methode **findViewById()**, der als Parameter eine Element-ID zu übergeben ist. Diese entsteht aus dem Attribut **android:id**, das einem Steuerelement in der Layoutdefinition zugewiesen wird.

Hier ist ein Beispiel aus der Layoutdefinition

`android:id="@+id/candidate"`

bzw. aus dem Activity-Quellcode der App `Prime Detection` zu sehen:

```
candidate = findViewById(R.id.candidate);
```

Zwischen dem textuellen Wert für das XML-Attribut **android:id** und der **int**-wertigen Element-ID, die durch eine statische Variable der Klasse **R.id** anzusprechen ist, vermittelt das AAPT (Android Asset Packaging Tool), das die Ressourcendateien (also auch die XML-Layoutdefinitionen) kompiliert.

Wenn eine Aktivität von der Displayoberfläche verdrängt wird und später zurückkehrt, speichert und restauriert Android automatisch die Zustandsdaten aller Steuerelemente, die eine Kennung besitzen (siehe Abschnitt 6.1).

#### 8.2.2.4 Breite und Höhe

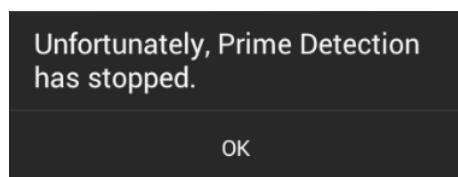
Die Layout-Attribute

- **android:layout\_width** bzw.
- **android:layout\_height**

für die Breite bzw. Höhe eines Steuerelements *müssen* mit Werten versorgt werden (Ausnahme: **TableLayout**, siehe Abschnitt 8.3.5). Man kann auflösungsabhängige oder (besser!) auflösungsunabhängige Größenangaben machen (vgl. Abschnitt 7.5). Oft sind aber die folgenden Attributausprägungen zu bevorzugen:

- **wrap\_content**  
Das **View**-Objekt soll gerade so breit bzw. hoch werden, dass sein Inhalt angezeigt werden kann.
- **match\_parent** (frühere, mittlerweile abgewertete Bezeichnung: **fill\_parent**)  
Das **View**-Objekt soll die gesamte Breite bzw. Höhe des übergeordneten Containers einnehmen. Ein Top-Level - Container beansprucht z. B. auf diese Weise die gesamte Breite bzw. Höhe des Displays.
- **match\_constraint** (nur beim **ConstraintLayout**)  
Ein von einem **ConstraintLayout**-Manager verwaltetes **View**-Objekt soll bei Beachtung der beteiligten Restriktionen (z. B. Layout-Gewichte) die gesamte freie Breite bzw. Höhe des Containers in Anspruch nehmen. Wird ein Steuerelement z. B. links- und rechtsseitig am Container-Rand verankert, dann führt der Wert **match\_constraint** für das Attribut **layout\_width** dazu, dass die gesamte Breite verwendet wird. Für die von einem **ConstraintLayout**-Manager verwalteten **View**-Objekte sollte auf keinen Fall die Dimensionsangabe **match\_parent** verwendet werden.<sup>1</sup>

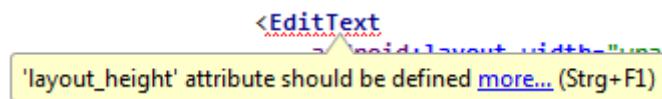
Fehlt für ein Steuerelement die Breiten- oder Höhenangabe, stürzt die App beim Berechnen der **View**-Ausdehnungen ab:



Das wird bei Ihren Apps vermutlich nur selten passieren, weil das Android Studio vorausschauend warnt, z. B.:

---

<sup>1</sup> <https://developer.android.com/reference/android/support/constraint/ConstraintLayout>



### 8.2.2.5 Ränder (Margins) und Einrückung (Padding)

Über die **layout\_margin**-Attribute bringt man ein View-Objekt auf Abstand zu seinen Nachbarn oder zu den Container-Rändern. Neben dem allseitig wirkenden Attribut **android:layout\_margin** gibt es die seitenspezifischen Attribute:

- **android:layout\_marginLeft**
- **android:layout\_marginRight**
- **android:layout\_marginTop**
- **android:layout\_marginBottom**

Über die **padding**-Attribute schafft man Abstand zwischen dem Inhalt eines View-Objekts und seinem Rand. Neben dem allseitig wirkenden Attribut **android:padding** gibt es die seitenspezifischen Attribute:

- **android:paddingLeft**
- **android:paddingRight**
- **android:paddingTop**
- **android:paddingBottom**

Anschließend sind bei einem **EditText**-Objekt mit

**android:layout\_width = android:layout\_height = "wrap\_content"**

und mit Nachbarn *ohne* Randangabe die Effekte bei einer Vergrößerung von Innen- und Außenrand zu sehen:

<b>layout_margin = "0dp" padding = "0dp"</b>	<b>layout_margin = "32dp" padding = "0dp"</b>	<b>layout_margin = "32dp" padding = "32dp"</b>
<p>Bitte eine ganze Zahl &gt; 1 eingeben: 123456789012345678 Prüfen</p>	<p>Bitte eine ganze Zahl &gt; 1 eingeben: 123456789012345678 Prüfen</p>	<p>Bitte eine ganze Zahl &gt; 1 eingeben: 23456789012345678 Prüfen</p>

In der dritten Variante hat der horizontale Platz *nicht* gereicht, um bei Beachtung von Außen- und Innenrand den kompletten Inhalt darzustellen (gemäß **android:layout\_height = "wrap\_content"**).

## 8.3 Container

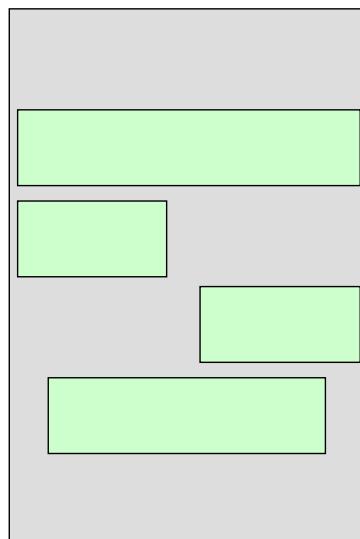
Die Spezialisierungen der Klasse **ViewGroup** können als Container andere View- oder **ViewGroup**-Objekte aufnehmen und auf der verfügbaren Fläche anordnen. Zwar gibt es unter Android mit seinem mehr oder weniger permanenten Fullscreen-Modus für die Container bzw. Layout-Manager weniger Stress als bei einer Java-Desktop-Anwendung, wo jedes Fenster jederzeit beliebig in der Breite und Höhe geändert werden kann, doch besteht auch bei einer Android-App reichlich Anpassungsbedarf:

- Bei einem Orientierungswechsel zwischen Hoch- und Querformat
- Beim Einsatz auf Geräten mit verschiedenen Display-Größen
- Weil Zeichenfolgen sinnvollerweise übersetzt werden und dabei gelegentlich ihre Länge erheblich ändern, muss eine flexible UI-Definition verhindern, dass Überlagerungen auftreten.

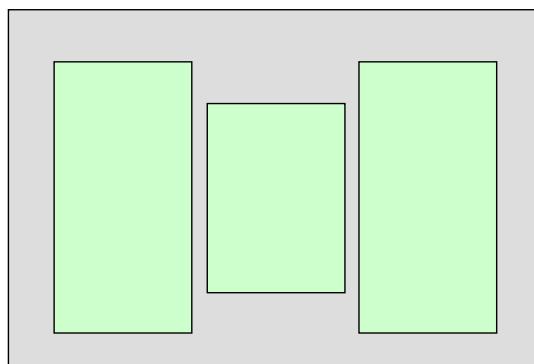
Anschließend werden einige häufig genutzte **ViewGroup**-Ableitungen vorgestellt. Die Layout-Manager dürfen verschachtelt werden, was die Flexibilität der GUI-Gestaltung erhöht. Allerdings sollte die Verschachtelung aus Performanzgründen nicht zu tief ausfallen. Moderne Layout-Manager (z. B. **ConstraintLayout** und **RelativeLayout**) zeichnen sich dadurch aus, dass auch komplexe Bedienoberfläche ohne die Notwendigkeit einer Verschachtelung definiert werden können. Bei einer einfachen Aktivität ist also gegen die Verwendung eines einfachen Layout-Managers (z. B. **LinearLayout**) nichts einzuwenden. Bevor man zu komplizierten Verschachtelungen greift (mit mehr als zwei Ebenen), sollte man unbedingt auf einen leistungsfähigen Layout-Manager umstellen. Aufgrund der guten Unterstützung durch den Layout Editor im Android Studio sind auch leistungsfähige Layout Manager leicht zu beherrschen. Das gilt vor allem für das **ConstraintLayout**, das folglich als Alternative zum **LinearLayout** in Kurs sehr oft zum Einsatz kommen wird,

### 8.3.1 LinearLayout

Das **LinearLayout** ordnet die enthaltenen Elemente vertikal oder horizontal an (gesteuert über das Attribut **android:orientation**). Bei vertikaler Anordnung belegt jedes Kindelement eine Zeile:

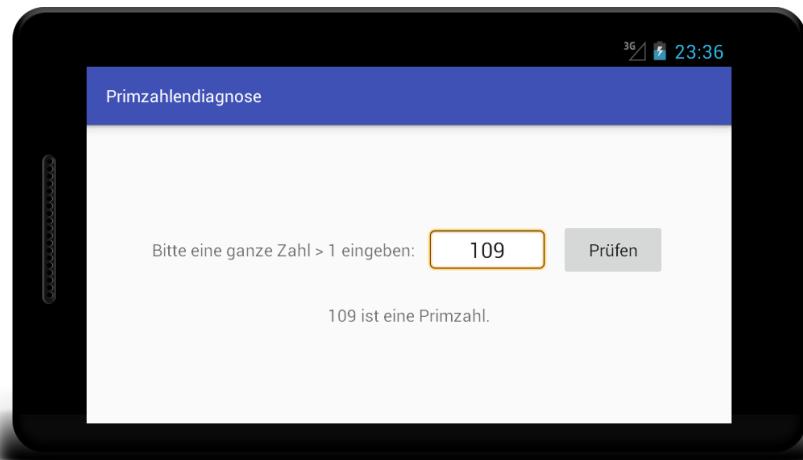


Bei horizontaler Anordnung belegt jedes Kindelement eine Spalte:



Weil das **LinearLayout** ebenso simpel wie flexibel ist, wird es am häufigsten verwendet. In unseren bisherigen Beispielprogrammen haben wir die vertikale Anordnung der Kindelemente bevorzugt.

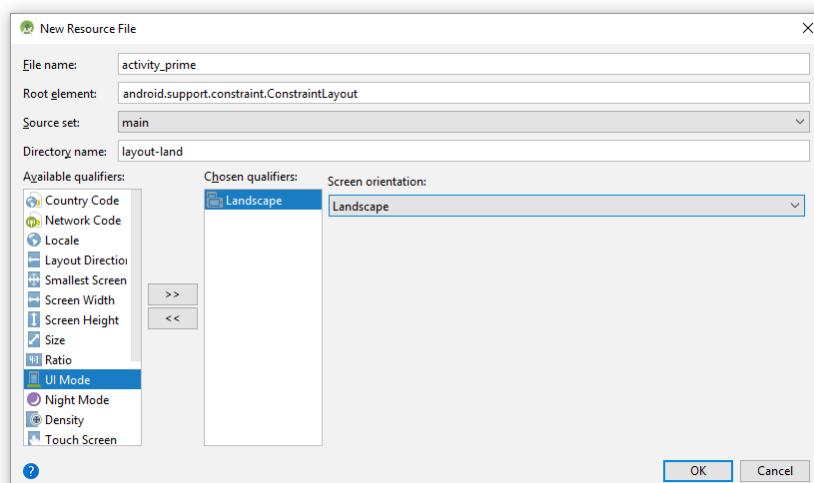
Damit unsere App Prime Detection bei einem Gerät mit Landschaftsorientierung so erscheint (Instruktion, Eingabefeld und Befehlsschalter nebeneinander, Ergebnisanzeige darunter),



wählen wir aus dem Kontextmenü zum Knoten **app/res/layout** im Project-Fenster das Item

### New > Layout resource file

und legen über das Fenster **New Resource File**



unter dem Namen Datei namens **activity\_prime.xml** (die Extension ist erlaubt, aber nicht erforderlich) eine Layout-Definition mit dem vorgeschlagenen **Root element** **android.support.constraint.ConstraintLayout** im Projektordner **layout-land** an. Um den korrekten Qualifizierer im Ordnernamen zu ermitteln, ...

- markiert man in der Liste **Available qualifiers** die Option **Orientation** und klickt an den Transportschalter **>>**.
- Dann wählt man **Landscape** als **Screen orientation**.

In der Datei **activity\_prime.xml** erstellen wir ein Layout mit zwei geschachtelten **LinearLayout**-Behältern:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center"
    android:orientation="vertical"
    tools:context=".PrimeActivity">
```

```

<LinearLayout
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="horizontal"
    android:gravity="center">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/prompt" />

    <EditText
        android:id="@+id/candidate"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginLeft="8dp"
        android:layout_marginRight="8dp"
        android:inputType="number"
        android:maxLength="18"
        android:background="@android:drawable/editbox_background"
        android:gravity="center_horizontal">
    </EditText>

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/diag"
        android:textAllCaps="false" />
</LinearLayout>

<TextView
    android:id="@+id/result"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="32dp" />
</LinearLayout>

```

Hier sorgt ein äußeres **LinearLayout**-Objekt (das Wurzelement) mit seiner vertikalen Orientierung dafür, dass folgende View-Objekte untereinander stehen:

- Ein inneres **LinearLayout**-Objekt mit *horizontaler* Orientierung, das die Eingabeaufforderung (vom Typ **TextView**), das Eingabefeld (vom Typ **EditText**) und den Befehlsschalter (vom Typ **Button**) als Kindelemente enthält.
- Die Ergebnisausgabe (vom Typ **TextView**)

Weil das äußere **LinearLayout**-Objekt für das Attribut **android:gravity** den Wert **center** besitzt, sind seine Kindelemente in vertikaler und horizontaler Richtung zentriert, sofern sie keine individuelle Ausrichtung wünschen.

Ein **LinearLayout**-Objekt respektiert folgende Darstellungswünsche der Kindelemente:

- Ausrichtung bzw. Anziehung  
Über das Attribut **android:layout\_gravity** können die Kindelemente ...
  - bei einem vertikal orientierten **LinearLayout** ihre Ausrichtung in horizontaler Richtung wählen (sinnvolle Werte: **left**, **right**, **center**),
  - bei einem horizontal orientierten **LinearLayout** ihre Ausrichtung in vertikaler Richtung wählen (sinnvolle Werte: **top**, **bottom**, **center**).

Weil die Größe eines Steuerelements durch die Gravitation *nicht* beeinflusst wird, sind manche Kombinationen sinnlos (z. B.: **left | right**).<sup>1</sup>

---

<sup>1</sup> Das Zeichen | steht für die bitweise ODER-Operation (siehe Abschnitt 8.3.4).

- **Ränder**  
Über das Attribut **android:layout\_margin** sorgt man für passende Abstände (siehe **EditText** - Element im Beispiel).

Beachten Sie den Unterschied zwischen den beiden Attributen:

- **android:layout\_gravity**  
Damit wird die Ausrichtung (Gravität) eines Kindelements in einem umgebenden Container geregelt.
- **android:gravity**  
Damit wird bei einem Container die Ausrichtung (Gravität) aller Kindelemente geregelt. Bei einem **EditText** - Objekt beeinflusst das Attribut die Textausrichtung.

Merkhilfe: Von den beiden Attributen wirkt dasjenige mit dem Namensbestandteil „out“ nach Außen, das andere nach innen. Weil ein Layout-Manager gleichzeitig Kindelement und Container sein kann (siehe Beispiel), sind hier beide Attribute relevant.

Wenn ein vertikal orientiertes **LinearLayout** seinen Kindern über das **gravity**-Attribut das horizontale Zentrieren empfiehlt, ein Kind hingegen über sein **layout\_gravity**-Attribut eine linksseitige Anziehung bevorzugt, dann dominiert der Wunsch des Kindes.

Ein **LinearLayout** kann den nach Berücksichtigung der Ausdehnungswünsche aller Kindelemente verbleibenden Platz auf die Kindelemente verteilen. Welchen Größenzuwachs ein Kindelement erfährt, hängt vom Attribut **android:layout\_weight** ab, das per Voreinstellung den Wert 0 besitzt. Um das Attribut **android:layout\_weight** im Beispiel nutzen zu können, wurde beim inneren **LinearLayout** - Container der Wert zum Attribut **android:layout\_width** von **wrap\_content** auf **match\_parent** geändert, um verteilbaren Platz zu schaffen:

**android:layout\_width="match\_parent"**

Wenn im Landschafts-Layout des **Prime Detection** - Beispiels das **TextEdit**-Objekt als Wert für das Attribut **android:layout\_weight** den Wert 1 erhält,

**android:layout\_weight="1"**

während die übrigen Elemente im selben Container weiterhin das voreingestellte Gewicht 0 haben, konsumiert das Texteingabefeld den ungenutzten horizontalen Platz:



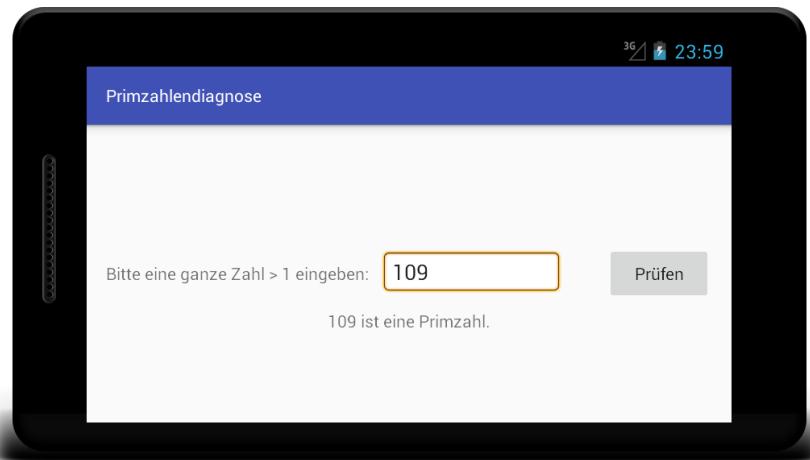
Es wird vielfach empfohlen und von Google auf den Android-Entwicklerseiten auch demonstriert, bei Verwendung von **android:layout\_weight** das zugehörige Ausdehnungsattribut auf **0dp** zu setzen, z. B.:<sup>1</sup>

**android:layout\_width="0dp"**

<sup>1</sup> <https://developer.android.com/guide/topics/ui/layout/linear.html>

### 8.3.2 RelativeLayout

Verschachtelte **LinearLayout**-Container (wie im vorigen Beispiel) können die Layout-Performance beeinträchtigen und sollten durch eine flache Konstruktion mit *einem* Layout-Container ersetzt werden.<sup>1</sup> Bevor Google im Jahr 2016 das **ConstraintLayout** vorgestellt hat, war für diesen Zweck das **RelativeLayout** der eindeutige Favorit. Beim **RelativeLayout** können Steuerelemente in horizontaler und vertikaler Richtung am umgebenden Container und/oder an anderen (durch ihre **id** bezeichneten) Steuerelementen ausgerichtet werden. Das folgende Landschafts-Layout für unsere App Prime Detection



wird so realisiert:

- Der einzige verbleibende Layout-Manager erhält den Typ **RelativeLayout**.
- Das **TextView**-Objekt mit der Instruktion erhält die **id** **prompt**:

`android:id="@+id/prompt"`

Es wird vertikal in Bezug auf den Container zentriert

`android:layout_centerVertical="true"`

und am linken Rand des Containers befestigt:

`android:layout_alignParentLeft="true"`

Außerdem erhält es einen linken Rand:

`android:layout_marginLeft="@dimen/activity_horizontal_margin"`

Statt einer festen Größenangabe in den Quellcode aufzunehmen, wird eine Ressource verwendet, die in der Datei **dimens.xml** (im Projektordner **app/res/values**) definiert wird:

```
<resources>
    <dimen name="activity_horizontal_margin">16dp</dimen>
</resources>
```

- Das **EditText**-Objekt orientiert seine vertikale Position am **TextView** mit der Instruktion:

`android:layout_alignBaseline="@id/prompt"`

Horizontal setzt es sich rechts neben die Instruktion:

`android:layout_toRightOf="@id/prompt"`

---

<sup>1</sup> Das empfiehlt Google auf der folgenden Webseite:  
<http://developer.android.com/training/improving-layouts/optimizing-layout.html>

Weil das Attribut **android:layout\_width** den Wert **wrap\_content** hat, kann die Breite des Texteingabefelds über das Attribut **android:ems** festgelegt werden:

```
android:ems="10"
```

Schließlich erhält das **EditText**-Objekt einen linken und einen rechten Rand:

```
android:layout_marginLeft="@dimen/activity_horizontal_margin"
    android:layout_marginRight="@dimen/activity_horizontal_margin"
```

- Der Befehlsschalter wird am rechten Rand des Containers befestigt:

```
    android:layout_alignParentRight="true"
```

In vertikaler Richtung orientiert der Befehlsschalter seine Position ebenfalls am Instruktionstext:

```
    android:layout_alignBaseline="@id/prompt"
```

Außerdem erhält der Schalter einen rechten Rand:

```
    android:layout_marginRight="@dimen/activity_horizontal_margin"
```

- Das **TextView**-Objekt mit dem Prüfresultat wird horizontal im Container zentriert

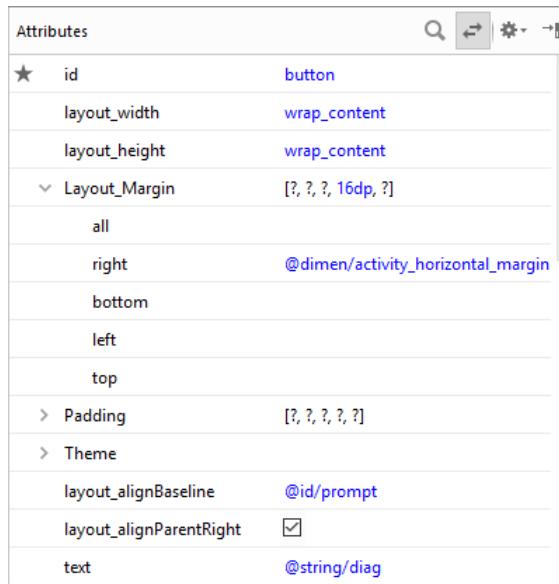
```
    android:layout_centerHorizontal="true"
```

und in vertikaler Richtung (mit einem Abstand) unter das **prompt** - Element platziert:

```
    android:layout_marginTop="24dp"
```

```
    android:layout_below="@id/prompt"
```

Das Android Studio erleichtert den Umgang mit dem **RelativeLayout** durch das **Attributes**-Fenster, z. B.:



Im Beispiel ist es gelungen, mit dem **RelativeLayout** die gewünschten relativen Anordnungen ohne Layout-Schachtelung zu erzielen. Es folgt die komplette Beschreibung des Landschafts-Layouts für die App **Prime Detection**:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <TextView
        android:id="@+id/prompt"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/prompt"
        android:layout_alignParentLeft="true"
        android:layout_centerVertical="true"
        android:layout_marginLeft="@dimen/activity_horizontal_margin" />

    <EditText
        android:id="@+id/candidate"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginLeft="@dimen/activity_horizontal_margin"
        android:layout_marginRight="@dimen/activity_horizontal_margin"
        android:inputType="number"
        android:maxLength="18"
        android:ems="10"
        android:background="@android:drawable/editbox_background"
        android:layout_alignBaseline="@id/prompt"
        android:layout_toRightOf="@id/prompt">
    </EditText>

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/diag"
        android:textAllCaps="false"
        android:layout_alignParentRight="true"
        android:layout_alignBaseline="@id/prompt"
        android:layout_marginRight="@dimen/activity_horizontal_margin"/>

    <TextView
        android:id="@+id/result"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="24dp"
        android:layout_centerHorizontal="true"
        android:layout_below="@id/prompt" />
</RelativeLayout>
```

Dem **RelativeLayout** fehlt die aus dem **LinearLayout** bekannte Möglichkeit, ungenutzten Platz gesteuert über das Attribut **android:layout\_weight** auf die am ehesten profitierenden Steuerelemente zu verteilen. Selbstverständlich ist es erlaubt, **LinearLayout** - und **RelativeLayout** - Behälter zu verschachteln.

Im **ConstraintLayout** ist die die dynamische Restplatzverteilung auf privilegierte Steuerelemente wieder möglich. Außerdem wird es durch den Layout-Editor im Android Studio weitaus besser unterstützt als das **RelativeLayout**. Folglich sollte für neue Projekte mit komplexem Layout das **ConstraintLayout** verwendet werden.

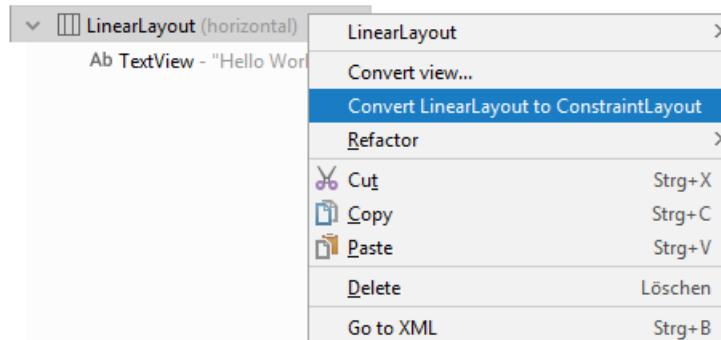
### 8.3.3 ConstraintLayout

Seit 2016 empfiehlt Google für komplexe Layouts die Klasse **ConstraintLayout**, die dank Support-Repository für Android-Versionen ab 2.3 (API-Level 9), also für praktisch alle aktuell im Einsatz befindlichen Android-Geräte verfügbar ist. Ihr besonderer Vorteil besteht in der perfekten Unterstützung durch den Layout-Editor im Android Studio, die ein direktes Editieren der XML-Datei weitgehend überflüssig macht.

Legt das Android Studio 3.1 eine neue Aktivität an, dann verwendet die Layout-Datei per Voreinstellung ein Wurzelement aus der Klasse **android.support.constraint.ConstraintLayout**:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
    .
    .
</android.support.constraint.ConstraintLayout>
```

Ein vorhandenes **LinearLayout** lässt sich per Kontextmenü in ein **ConstraintLayout** konvertieren:



Jedes Element in einem **ConstraintLayout** benötigt mindestens eine horizontale und eine vertikale Beschränkung, wobei in Frage kommen:

- Befestigung an einem Container-Rand
- Ausrichtung in Bezug auf ein anderes Element
- Befestigung an einer unsichtbaren Orientierungslinie

Wird ein Steuerelement per Drag & Drop von der **Palette** auf die **Design**-Zone des Layout-Editors gezogen, erscheint es dort mit quadratischen Anfassern zur Größenänderung in den Ecken und mit runden Anfassern zur Definition von Restriktionen an den vier Seiten, z. B.:



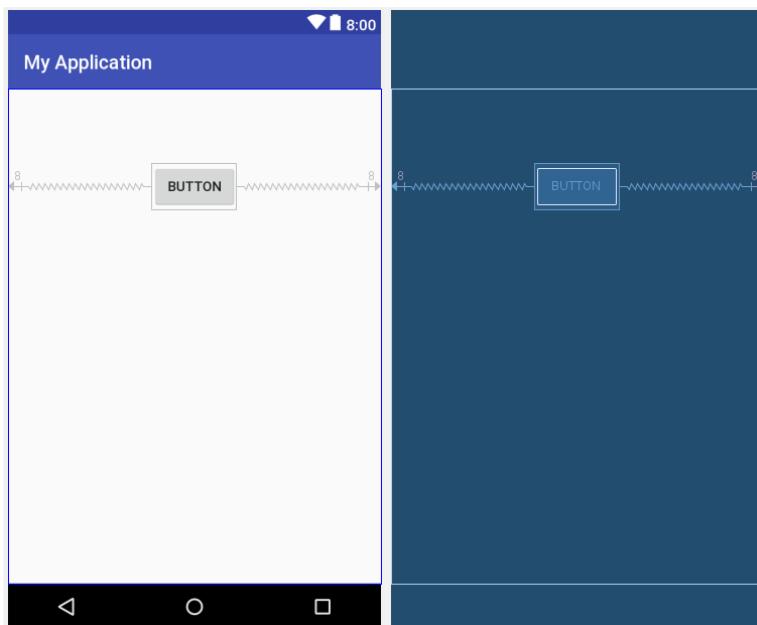
Über die quadratischen Anfasser in den Ecken kann die Größe des Elements geändert werden. Allerdings entstehen so fixierte Werte, die nach einer Änderung der Container-Größe (z. B. wegen eines Orientierungswechsels) vermutlich nicht mehr passen.

Wir verwenden gleich die runden Anfasser, um Restriktionen zu vereinbaren. Weil das automatische Erstellen von Restriktionen (siehe Abschnitt 8.3.7) aus didaktischen Gründen zunächst unerwünscht ist, schalten wir es über den Schalter in der Symbolleiste des Layout-Editors ab.

Im Layout-Editor bietet das Android Studio über das Drop-Down-Menü zum Schalter als Ansichten zur Wahl:

- nur **Design** (linke Seite im folgenden Bildschirmfoto)
- nur **Blueprint** (rechte Seite im folgenden Bildschirmfoto)
- Beide

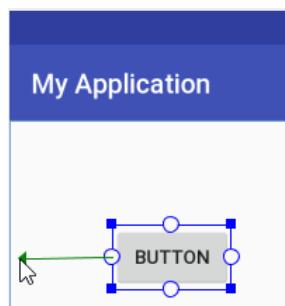
Häufig bieten beide Ansichten dieselbe Information, z. B.:



Daher ist es nur selten erforderlich, beide Ansichten zu aktivieren.

#### 8.3.3.1 Constraints in Bezug auf den Container

Um ein Steuerelement an einem Container-Rand zu verankern, setzt man einen Mausklick auf den zugewandten runden Anfasser, hält die Maustaste gedrückt und zieht den grünen Pfeil bis zum Container-Rand:

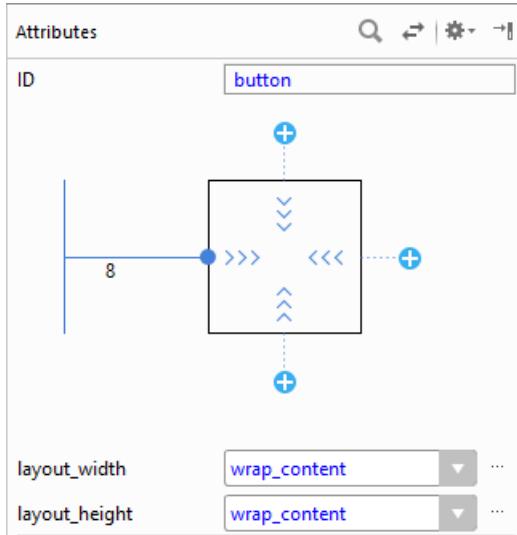


Im Beispiel hält das **Button**-Objekt nach der Verankerung zum Container-Rand den voreingestellten Abstand von 8dp, zu dem per Symbolleiste 8dp mit Wirksamkeit für zukünftig eingefügte Steuerelemente) eine Alternative eingestellt werden kann:



Dass alle Abstände ein Vielfaches von 8 dp sind, liegt am empfohlenen Grundlinienraster im Material Design von Android.<sup>1</sup>

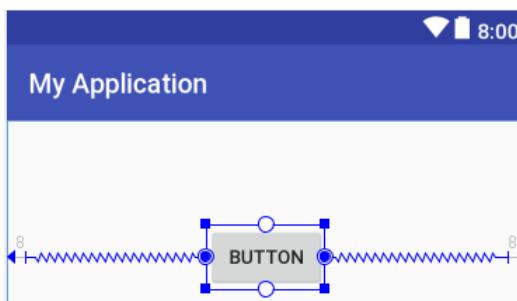
Der nur beim **ConstraintLayout** am oberen Rand des **Attributes**-Fensters vorhandene **View Inspector**



zeigt, ...

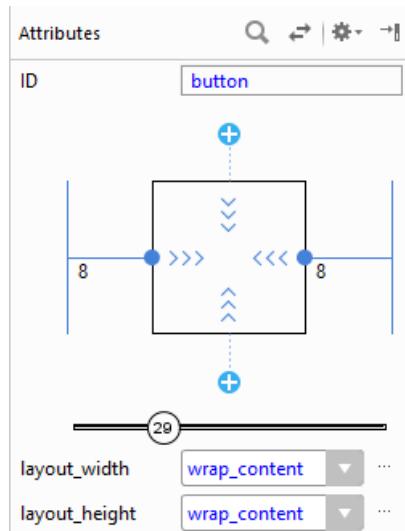
- dass eine Verankerung am linken Container-Rand besteht,
- dass der Randabstand 8 dp beträgt.

Wird die Gegenseite des Steuerelements am gegenüber liegenden Container-Rand verankert, dann können bei der voreingestellten Größe des Steuerelements (z. B. **layout\_width = wrap\_content**) die beiden Randabstände von 8 dp nicht gleichzeitig eingehalten werden. Das von beiden Seiten einer identischen Anziehungskraft ausgesetzte Steuerelement landet in der Mitte, z. B.:

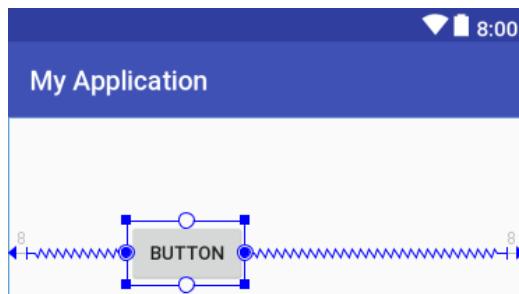


Im View Inspector erlaubt ein Schieberegler mit Prozentangabe, die Kräftebalance zu verändern,

<sup>1</sup> <https://developer.android.com/distribute/best-practices/develop/use-material-design>

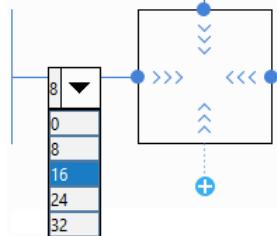


sodass sich das Steuerelement aus der Mittellage herausbewegt:



Der View Inspector erlaubt noch weitere Layout-Eingriffe:

- Über den Schalter kann eine weitere Verankerung vorgenommen werden.
- Über den Schalter kann eine Verankerung gelöst werden.
- Der Randabstand zu einer Verankerung lässt sich per Drop-Down - Werkzeug verändern:

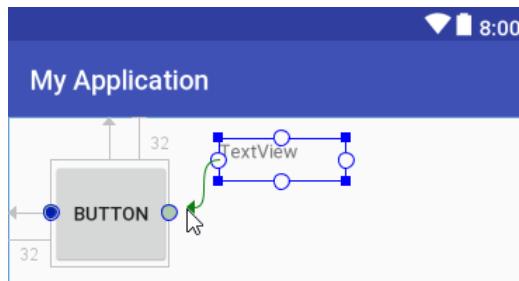


- In dem das Steuerelement darstellenden Quadrat befinden sich Symbole für die Breite und Höhe des Steuerelements. Per Mausklick auf ein Symbol wechselt man zum nächsten Zustand aus der folgenden Serie:
  - >>> **wrap\_content**  
Die Breite bzw. Höhe des Steuerelements orientiert sich am darzustellenden Inhalt.
  - fixierter Wert mit der aktuellen Breite bzw. Höhe
  - match\_constraint  
Das Steuerelement konsumiert den nach Beachtung von Restriktionen (z. B. Randabständen) verfügbaren Platz.

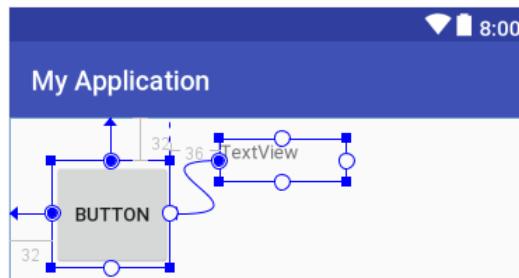
### 8.3.3.2 Constraints in Bezug auf andere Steuerelemente

Ein Steuerelement kann seine vertikale oder horizontale Koordinate von einem anderen Steuerelement übernehmen, wobei in der Regel für einen Abstand zwischen den beiden Elementen gesorgt

wird. Im folgenden Beispiel wird der linksseitige Constraint-Anfasser eines **TextView**-Objekts auf den Verankerungspunkt eines **Button**-Objekts gezogen:



Dadurch übernimmt das **TextView**-Objekt seine horizontale Koordinate vom **Button**-Objekt, indem es (mit Abstand) rechts an den **Button** andockt:

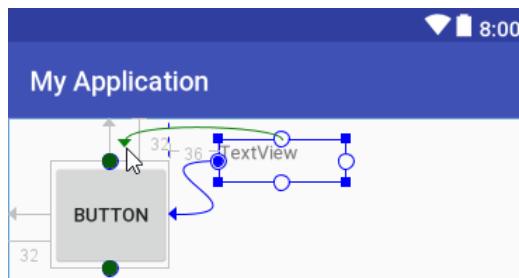


Hier ist die Umsetzung der Restriktion in XML-Attribute zu sehen:

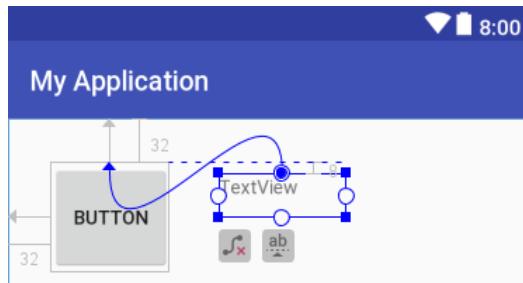
```
<TextView  
    android:id="@+id/textView5"  
    android:layout_width="94dp"  
    android:layout_height="32dp"  
    android:layout_marginLeft="36dp"  
    android:layout_marginStart="36dp"  
    android:text="TextView"  
    app:layout_constraintStart_toEndOf="@+id/button"  
    tools:layout_editor_absoluteY="16dp" />
```

Das Attribut **tools:layout\_editor\_absoluteY** ist nur für die Editor-Phase gedacht. Wenn dem **TextView**-Objekt keine vertikale Koordinate zugewiesen wird, erscheint es im laufenden Programm am oberen Display-Rand.

Eine sinnvolle Möglichkeit, die vertikale Koordinate des **TextView**-Objekts festzulegen, besteht in der vertikalen Ausrichtung in Bezug auf das **Button**-Objekt. Dazu wird der obere Constraint-Anfasser des **TextView**-Objekts auf den oberen Verankerungspunkt des **Button**-Objekts gezogen:

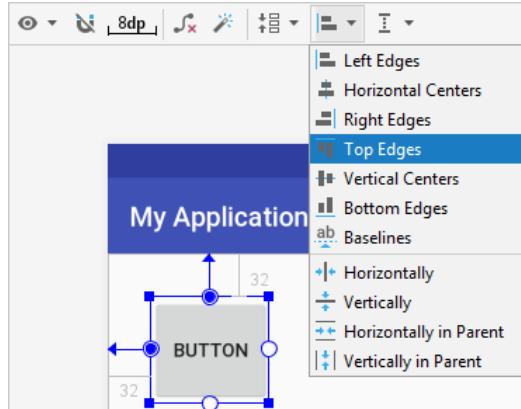


Dasselbe Ergebnis



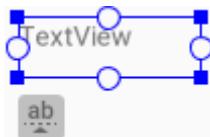
kann man auch so erreichen:

- Beide Steuerelemente markieren
- Aus dem Drop-Down-Menü zum Schalter das Item **Top Edges** wählen:

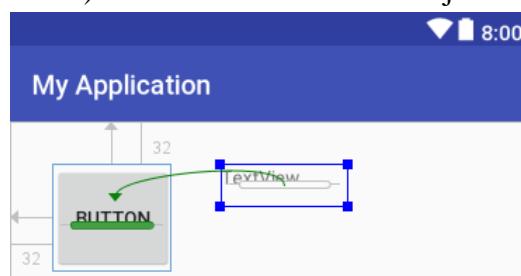


Ist für zwei Steuerelemente eine Beschriftung vorgesehen (z. B. bei den Steuerelementklassen **Button** und **TextView**), dann kommt eine vertikale **Ausrichtung der Beschriftungsgrundlinien** in Frage. Im Beispiel kann man so vorgehen:

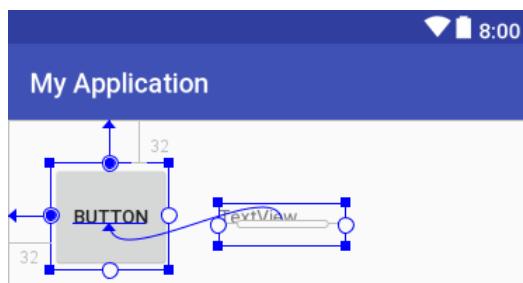
- Wird das **TextView**-Objekt markiert, erscheint darunter das Symbol zur Einleitung einer Grundlinienausrichtung:



- Nach einem Klick auf dieses Symbol, wird die Grundlinie des **TextView**-Objekts angezeigt, und man kann von dort eine Drag&Drop - Aktion starten, um eine vertikale Ausrichtungsbeziehung zur (grün blinkenden) Grundlinie des **Button**-Objekts herstellen:



Dasselbe Ergebnis



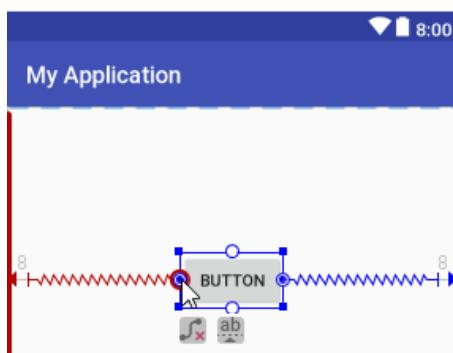
lässt sich nach dem Markieren der beiden Steuerelemente auch über das Item **Baselines** im Drop-Down-Menü zum Schalter erreichen.

#### 8.3.3.3 *match\_constraint*

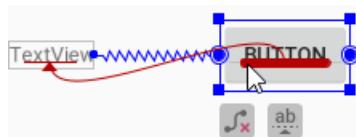
Mit dem Wert **match\_constraint** für die Attribute **layout\_width** oder **layout\_height** wird ein per **ConstraintLayout**-Manager verwaltetes View-Objekt aufgefordert, unter Beachtung der beteiligten Restriktionen die gesamte freie Breite bzw. Höhe des Containers in Anspruch zu nehmen. Wird ein Steuerelement z. B. links- und rechtsseitig am Container-Rand verankert, dann führt der Wert **match\_constraint** für das Attribut **layout\_width** dazu, dass die gesamte Breite verwendet wird. Für die von einem **ConstraintLayout**-Manager verwalteten View-Objekte soll auf keinen Fall die Dimensionsangabe **match\_parent** verwendet werden.<sup>1</sup>

#### 8.3.3.4 *Restriktionen aufheben*

Um eine Restriktion aufzuheben, setzt man einen Mausklick auf ihren Ausgangspunkt. Befindet sich der Mauszeiger über diesem Punkt, ist scheint die Restriktion komplett in roter Farbe:

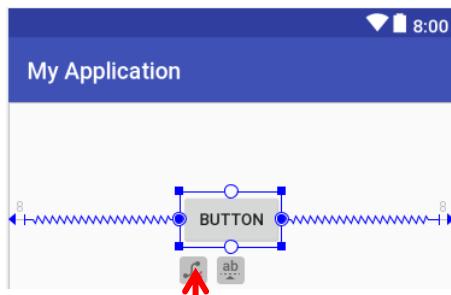


Eine Grundlinienausrichtung kann per Mausklick auf die rot blinkende Grundlinie aufgehoben werden:



Um *alle* Restriktionen *eines Steuerelements* zu löschen, markiert man es und klickt dann auf das unter dem Steuerelement erschienene Symbol , z. B.:

<sup>1</sup> <https://developer.android.com/reference/android/support/constraint/ConstraintLayout>

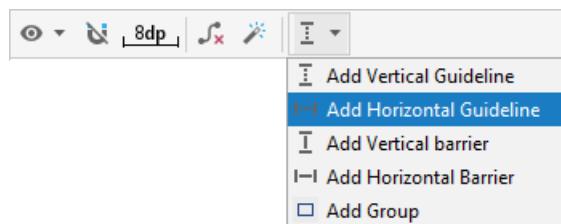


Um *alle* Restriktionen im gesamten **ConstraintLayout** zu löschen, klickt man auf den Schalter  in der Symbolleiste:

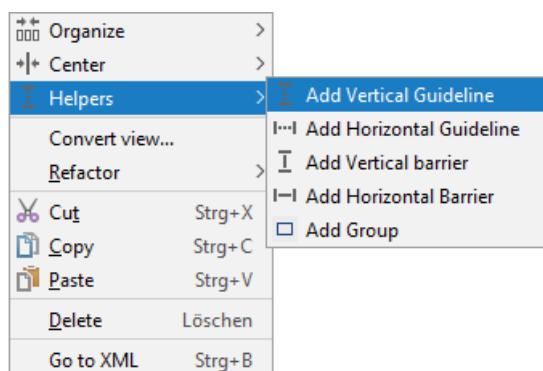


### 8.3.3.5 Ausrichtung an Orientierungslinien oder Barrieren

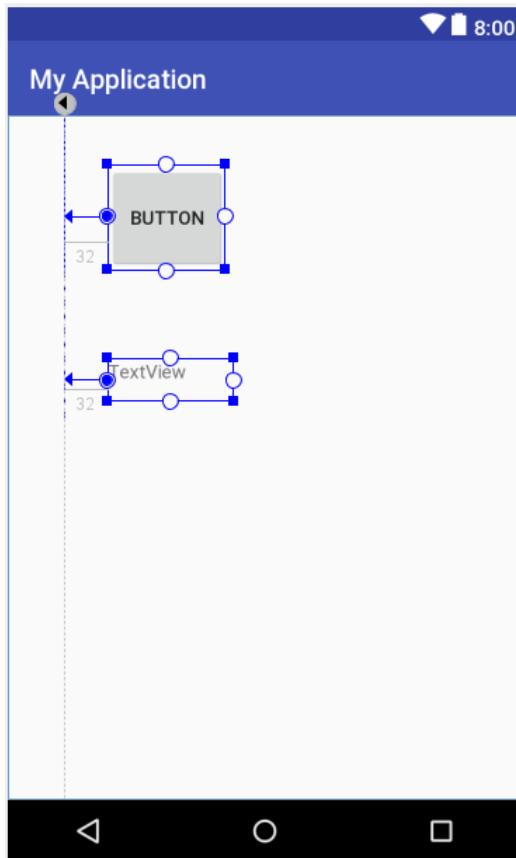
Über das Drop-Down - Menü zum Schalter  in der Symbolleiste



oder per Kontextmenü zur Designzone



kann man eine horizontale oder vertikale Orientierungslinie (engl.: *guideline*) einzeichnen, die im späteren Programm nicht zu sehen ist, aber als Bezug für eine vertikale oder horizontale Restriktion dienen kann, z. B.:

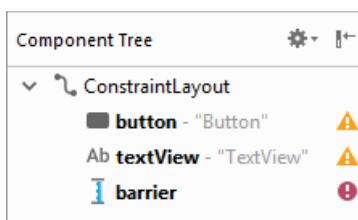


Die Orientierungslinie erscheint in der XML-Layout-Deklaration als Element vom Typ **android.support.constraint.Guideline**:

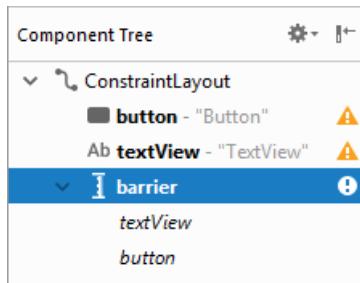
```
<android.support.constraint.Guideline
    android:id="@+id/guideline"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    app:layout_constraintGuide_begin="43dp" />
```

Sie kann in der Designphase verschoben werden, woraufhin sich alle angedockten Elemente bewegen. Weil die Orientierungslinie eine Kennung erhält, kann sie auch im laufenden Programm verschoben werden.

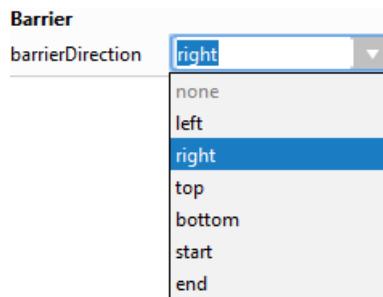
Über die oben genannten Menüs lässt sich auch eine horizontale oder vertikale **Barriere** einfügen. Sie kann wie eine Orientierungslinie zum Andocken verwendet werden, hat aber keine eigenständige Position, sondern übernimmt die in einer ausgewählten Richtung maximale Position der zugeordneten Steuerelemente. Wird im Beispiel eine vertikale Barriere eingefügt, ist dies zunächst nur im **Component Tree** zu bemerken:



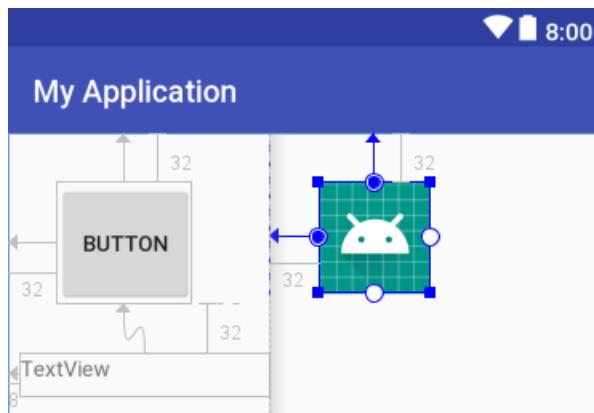
Man ordnet der Barriere per Drag&Drop im **Component Tree** Steuerelemente unter, z. B.



Per **Attributes**-Fenster kann der Barriere eine Richtung zugeordnet werden, z. B.:



Im Beispiel kann man die Barriere nun sehen und zum Andocken von Steuerelementen verwenden:



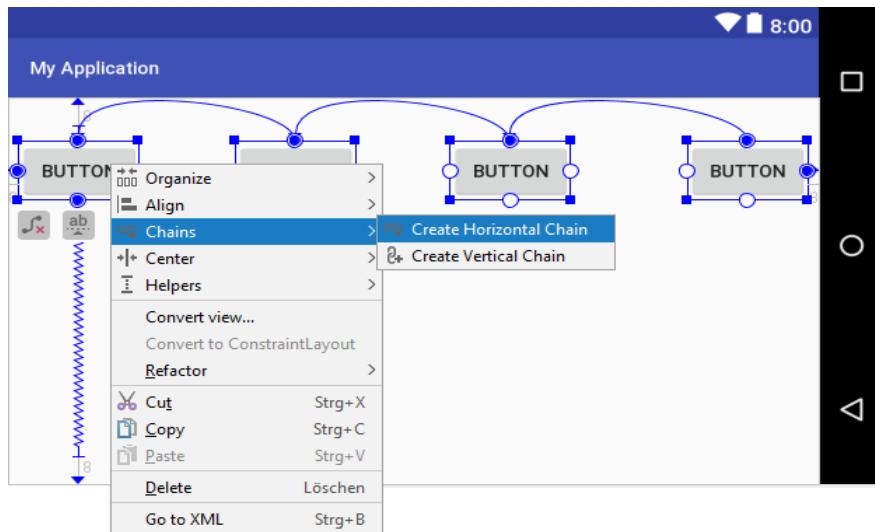
Ein wesentlicher Nutzen der Barriere besteht in der automatischen Reaktion auf größenvariable Steuerelemente. Ist z. B. bei einem **TextView**-Objekt mit lokalisierten Texten für das Attribut **layout\_width** der Wert **wrap\_content** eingestellt, dann wird sich bei einem Wechsel der Sprache vermutlich die Breite ändern. Mit einer Barriere wird verhindert, dass durch die Größenzunahme benachbarte Steuerelemente tangiert werden.

#### 8.3.3.6 Steuerelementketten mit Platzaufteilung

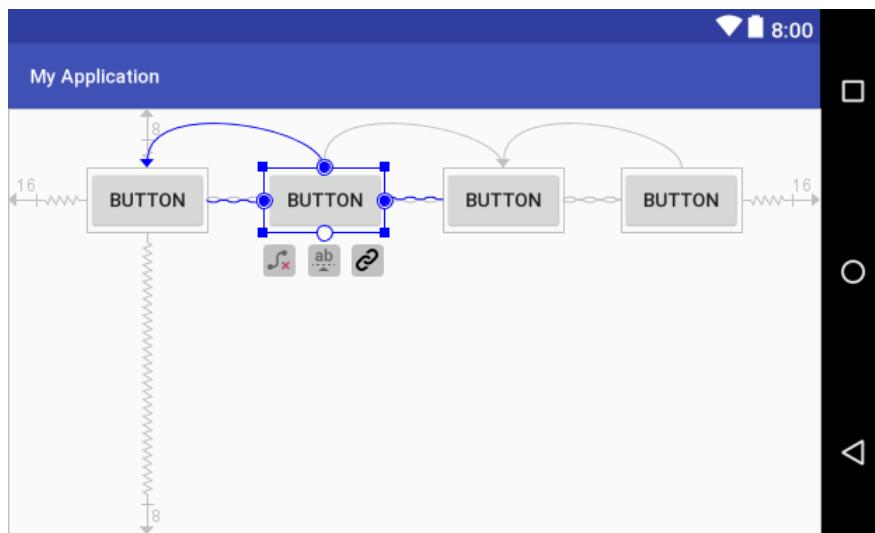
Aus mehreren markierten Steuerelementen erstellt man eine horizontal oder vertikal orientierte Kette mit einem veränderbaren Kriterium für die Platzaufteilung. Im folgenden Beispiel wird aus 4 markierten **Button**-Objekten über das Kontextmenü-Item

**Chains > Create Horizontal Chain**

eine horizontal orientierte Kette gebildet:

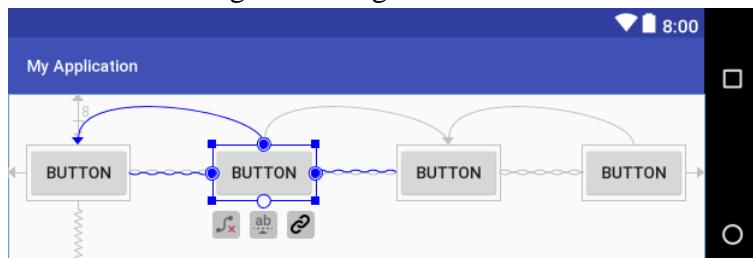


Durch wiederholte Mausklicks auf das KettenSymbol unter einem markierten Kettenglied

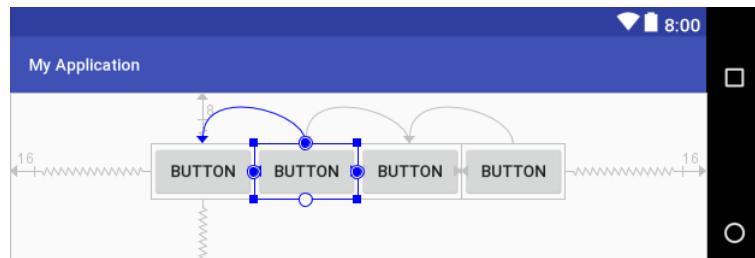


wechselt man den Platzaufteilungsmodus:

- **Gleichmäßige Verteilung** (= Voreinstellung)  
Die Steuerelemente werden gleichmäßig verteilt (siehe oben).
- **Gleichmäßige Verteilung innerhalb**  
Die äußeren Glieder werden (mit eingestelltem Abstand am Container-Rand angedockt. Die restlichen Steuerelemente werden gleichmäßig verteilt:



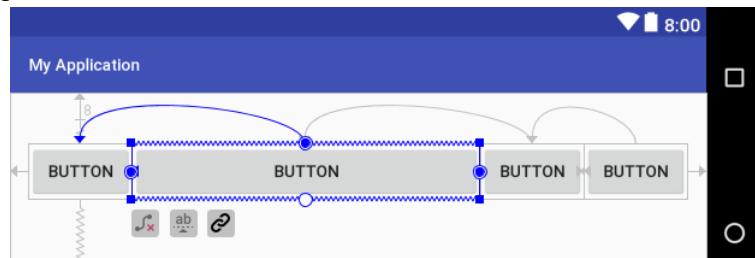
- **Gepackt**



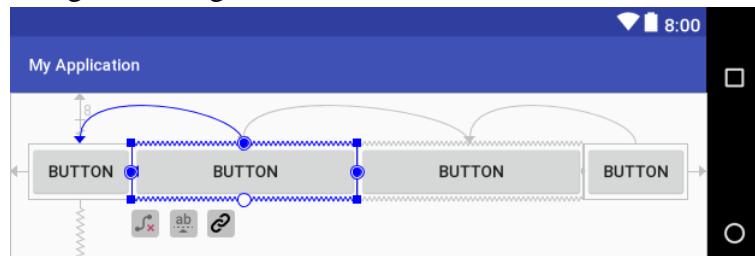
Die Steuerelemente folgen (mit definierten Abständen) unmittelbar aufeinander und sind gemeinsam zentriert.

- **Gewichtet**

Besitzt ein Steuerelement beim Attribut **layout\_width** den Wert **match\_constraint**, dann belegt es den ungenutzten Platz:



Haben *mehrere* Steuerelemente beim Attribut **layout\_width** den Wert **match\_constraint**, dann entscheiden deren Werte beim Attribut **layout\_constraintHorizontal\_weight** über die Platzverteilung. Im folgenden Beispiel haben die beiden mittleren Schalter denselben positiven Wert und die beiden äußeren Schalter den Wert 0, sodass die mittleren Schalter den überschüssigen Platz gleichmäßig unter sich aufteilen:

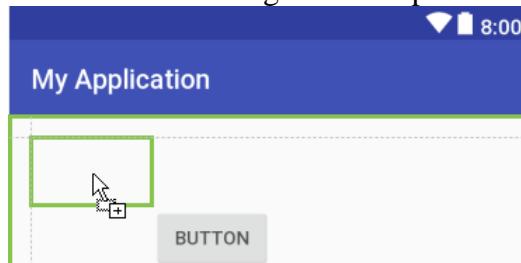


Bei vertikal orientierten Ketten geht man analog vor bzw. verwendet analog benannte Attribute.

### 8.3.3.7 Automatisch erstellte Constraints

Ist die **Autoconnect**-Funktion aktiv (erkennbar am Schalter in der Symbolleiste des Layout-Editors), dann wird ein neu eingefügtes Steuerelement automatisch an Container-Rändern verankert, wenn der Editor von einer geplanten Position ausgeht, z. B. ...

- weil das Steuerelement an einer durch Führungslinien empfohlenen Position abgelegt wird:

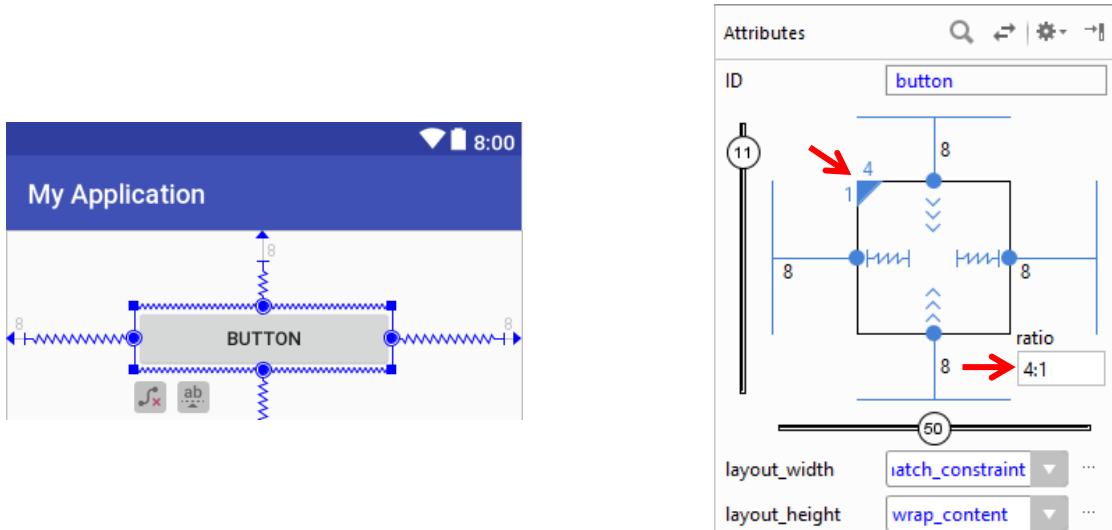


- weil das Element an einer horizontal und/oder vertikalen zentralen Position abgelegt wird.

Als Ausgangsbasis für eine manuelle Layout-Gestaltung kann der Editor über das Werkzeug aufgefordert werden, Restriktionen für alle vorhandene Steuerelemente vorzunehmen.

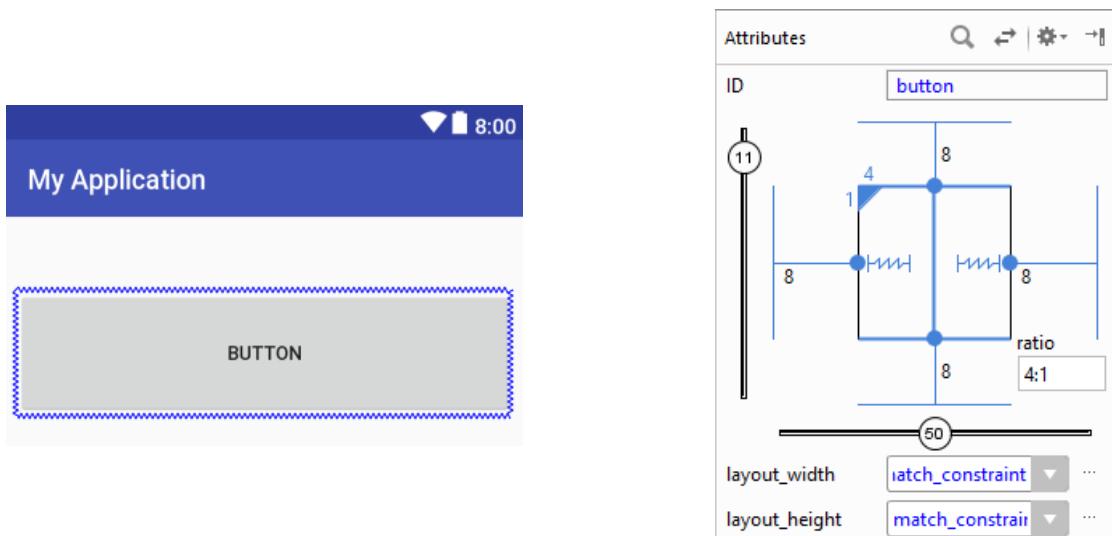
### 8.3.3.8 Größenangabe per Verhältnis

Ist für ein Steuerelement entweder **layout\_width** oder **layout\_height** auf den Wert **match\_constraint** (maximaler Raumgewinn nach Beachtung der Restriktionen) eingestellt, dann zeigt der View Inspector in der linken oberen Ecke des Element-Stellvertreters ein Dreieck und außerdem unten rechts ein Textfeld mit dem aktuellen Seitenverhältnis, z. B.:



Im Beispiel wird die Höhe eines **Button**-Objekts durch seinen Inhalt bestimmt (**layout\_height = wrap\_content**), und die Breite ist auf das 4-fache der Höhe festgelegt. Dieses Seitenverhältnis bleibt z. B. auch nach einem Orientierungswechsel gültig.

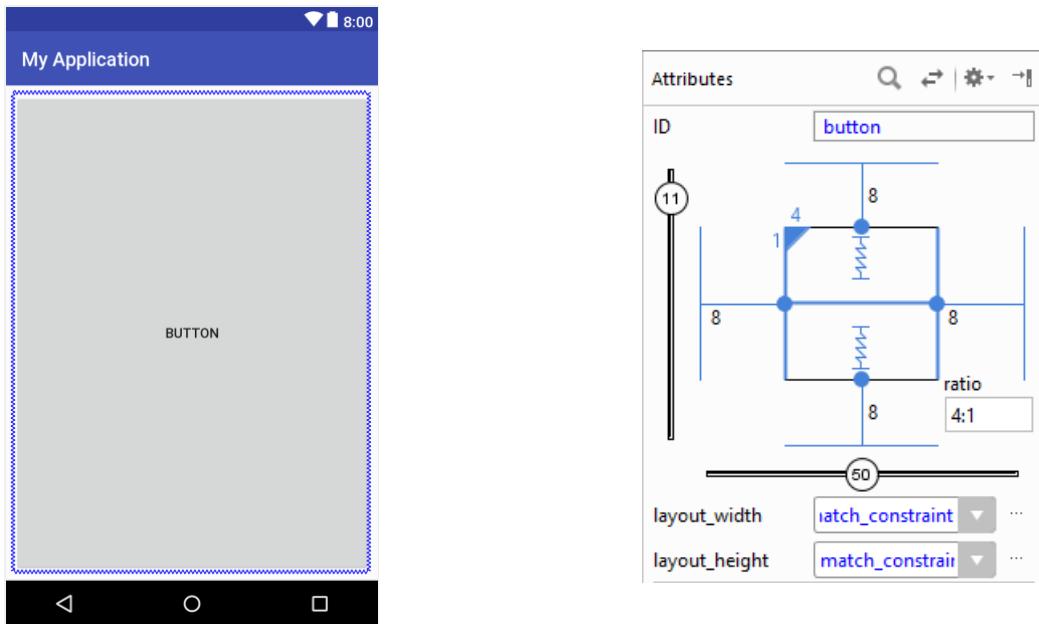
Sind **layout\_width** und **layout\_height** auf den Wert **match\_constraint**, also auf maximalen Raumgewinn eingestellt, dann kann über das Dreieck festgelegt werden, welche Dimension die absolute Größe des Steuerelements festlegt, während die andere Dimension ihre Größe am geforderten Verhältnis orientieren muss. In der folgenden Variante des Beispiels konsumiert der Schalter die maximale Breite, und die Höhe ergibt sich aus dem Größenverhältnis:



Der Inspector zeigt für die vertikale Dimension eine durchgezogene Linie, und in der XML-Layout-Deklaration erhält das Attribut **app:layout\_constraintDimensionRatio** als Wert eine Verhältnisangabe mit vorangestelltem **h**:

```
app:layout_constraintDimensionRatio="h,4:1"
```

Nach einem Mausklick auf das Dreieck belegt der Befehlsschalter die gesamte Display-Fläche, weil nun die Höhe den maximalen Wert annimmt, und die Breite mit mäßigem Erfolg den 4-fachen Wert der Höhe anstrebt:



Der Inspector zeigt für die horizontale Dimension eine durchgezogene Linie, und in der XML-Layout-Deklaration erhält das Attribut **app:layout\_constraintDimensionRatio** als Wert eine Verhältnisangabe mit vorangestelltem w:

`app:layout_constraintDimensionRatio="w,4:1"`

### 8.3.4 FrameLayout

Dieser simple Layout-Manager platziert die aufgenommenen Elemente nacheinander auf einen Stapel unter Berücksichtigung der folgenden Layout-Parameter:

- **android:layout\_width** (siehe Abschnitt 8.2.2.4)
- **android:layout\_height** (siehe Abschnitt 8.2.2.4)
- **android:layout\_margin** (siehe Abschnitt 8.2.2.5)
- **android:layout\_gravity**

Mit dem Attribut **android:layout\_gravity** wird für ein Steuerelement festgelegt, von welcher Seite es angezogen werden soll. Erlaubte Werte sind:

- **top, bottom, left, right**
- **center\_vertical, center\_horizontal**  
Vertikales bzw. horizontales Zentrieren
- **center**  
Vertikales *und* horizontales Zentrieren

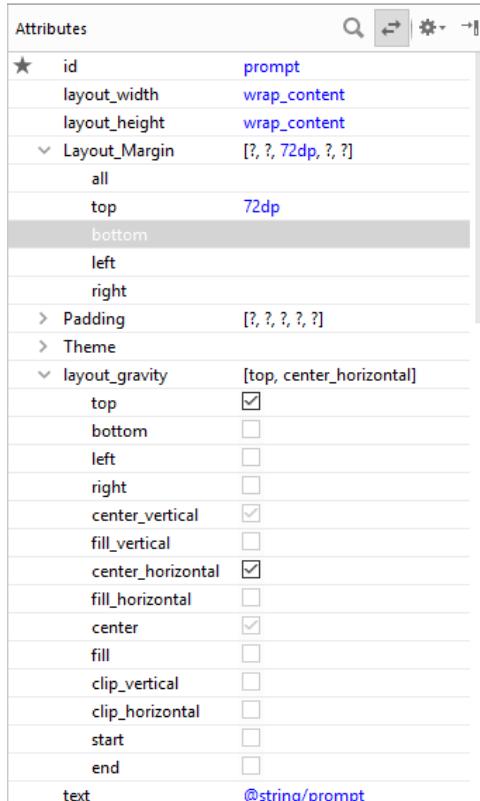
Einige Werte können sinnvoll durch den bitweisen Oder-Operator | verknüpft werden (z. B. **top | center\_horizontal**). Dieser Operator verarbeitet die Bitmuster von zwei ganzen Zahlen zu einem neuen Bitmuster, das an der Position *k* genau dann eine Eins enthält, wenn mindestens ein Argument an dieser Stelle eine Eins besitzt, z. B.:<sup>1</sup>

<sup>1</sup> Zur den Werten der Konstanten siehe:  
[http://developer.android.com/reference/android/widget/LayoutParams.html#attr\\_android:layout\\_gravity](http://developer.android.com/reference/android/widget/LayoutParams.html#attr_android:layout_gravity)

android:layout_gravity	Hexadezimaler Wert	Bitmuster (Typ: int)
top	0x30	0000 0000 0000 0000 0000 0000 0011 0000
center_horizontal	0x01	0000 0000 0000 0000 0000 0000 0000 0001
top   center_horizontal	0x31	0000 0000 0000 0000 0000 0000 0011 0001

Im Beispiel liefert der | - Operator dasselbe Ergebnis wie die Addition, was im Allgemeinen aber *nicht* der Fall ist.

Ein Android-Entwickler muss aber keine Bit-Berechnungen durchführen, sondern kann per **Attributes**-Fenster die Ausrichtung eines Steuerelements definieren, z. B.:



Daraus erstellt die Entwicklungsumgebung den XML-Code, z. B.:

```
<TextView
    android:id="@+id/prompt"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="top|center_horizontal"
    android:layout_marginTop="72dp"
    android:text="@string/prompt" />
```

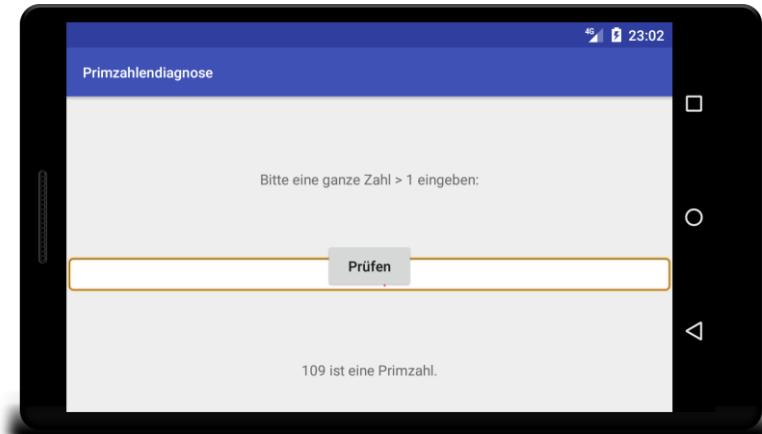
Weil die Größe eines Steuerelements in einem **FrameLayout**-Container durch die Gravitation *nicht* beeinflusst wird, sind manche Kombinationen sinnlos (z. B.: **top | bottom**).

In der folgenden Variante des Primzahlenprogramms residiert ...

- das **TextView**-Objekt mit der Instruktion in der oberen Display-Hälfte (mit 72dp Abstand zum oberen Rand) bei horizontaler Zentrierung,
- das **Button**-Objekt in der unteren Display-Hälfte (mit 72dp Abstand zum unteren Rand) bei horizontaler Zentrierung,
- das **EditText**-Objekt (mit dem Wert **match\_parent** für das Attribut **android:layout\_width**) vertikal zentriert,
- das **TextView**-Objekt mit der Ergebnisanzeige unten (mit 32dp Abstand zum unteren Rand) bei horizontaler Zentrierung.



In Abhängigkeit von den gewünschten Größen- und Positionen der verwalteten Elemente kann es zu Überlappungen kommen, was hier nach einem Wechsel zum Querformat passiert ist:



Offenbar ist der **FrameLayout**-Container für diese App wenig ungeeignet. Google empfiehlt, das **FrameLayout** nur für ein *einzelnes* Kindelement zu verwenden, z. B. für eine Liste (siehe Abschnitt 8.5.7) oder für ein im Vollbildmodus ablaufendes Video. In dieser Situation ist der extrem niedrige Speicherbedarf des **FrameLayout**-Containers vor Vorteil.

### 8.3.5 TableLayout

Das **TableLayout** setzt seine Kindelemente in die Zellen einer Tabelle und besitzt eine deutliche Verwandtschaft mit dem HTML-Element `<table>`. Zur Definition einer Zeile ist ein Kindelement vom Typ **TableRow** zu verwenden, das dem HTML-Element `<tr>` entspricht. Anders als eine HTML-Tabelle zeichnet ein **TableLayout** keine Rahmen.

Hinsichtlich Verwendung und Funktionalität zeigt ein **TableRow** - Element viele Gemeinsamkeiten mit dem horizontal orientierten **LinearLayout**. Die eingefüllten **View**-Elemente werden sukzessive auf die Spalten verteilt. Die Spaltenzahl der Tabelle wird durch die Zeile mit der größten Spaltenzahl festgelegt.

Hinsichtlich der Verteilung der Kindelemente auf die Zellen eines **TableRow** - Zeilen-Containers sind Liberalisierungen im Vergleich zum sukzessiven Besetzen der Zellen möglich:

- Man kann Zellen leer lassen

Für die Elemente in den Zellen kann das Attribut **android:layout\_column** auf einen 0-basierten Wert gesetzt werden, z. B.:

****android:layout\_column="1"****

Die Zellelemente müssen in der Reihenfolge ihrer räumlichen Anordnung eingefügt werden. Fehlt das Attribut **layout\_column**, wird der Spaltenindex automatisch inkrementiert.

- Ein **View**-Element kann sich über mehrere Spalten ausdehnen, z. B.:

****android:layout\_span="2"****

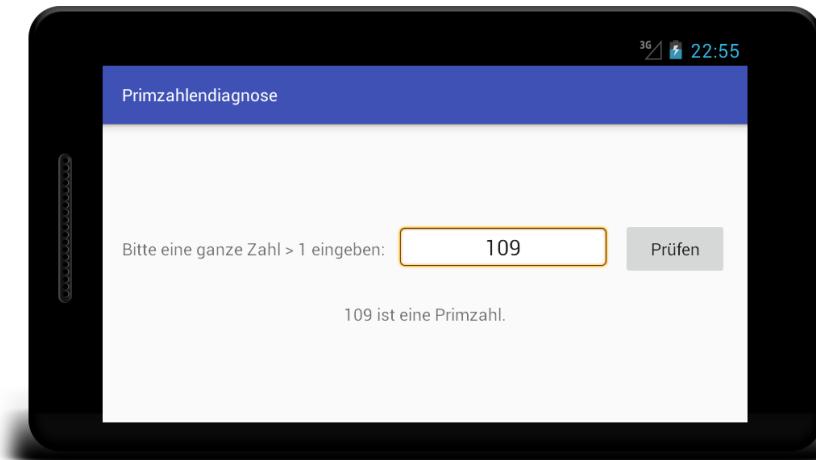
Wird als Zeile statt eines **TableRow**-Elements ein beliebiges **View**-Element eingefügt, erstreckt sich dieses über die komplette Zeile. Auf diese Weise lassen sich z. B. Trennlinien realisieren. Aus dem folgenden Element der Basisklasse **View** mit einer Höhe von 1dp und schwarzer Hintergrundfarbe resultiert eine dünne Linie zwischen zwei Zeilen:

```
<View  
        android:layout_height="1dp"  
        android:background="#000000" />
```

Für die Größenverhältnisse in einem **TableLayout** gelten die folgenden Regeln:

- Die Breite der Tabelle wird durch den umgebenden Container (bzw. das Display) festgelegt.
- Über die Breite einer Spalte entscheidet per Voreinstellung die Zeile mit dem breitesten Inhalt in dieser Spalte.
- Über das XML-Attribut **android:stretchColumns** oder die Methode **setColumnStretchable()** lassen sich Spalten auswählen, die gedehnt werden dürfen, um den verfügbaren horizontalen Platz zu nutzen.
- Über das XML-Attribut **android:shrinkColumns** oder die Methode **setColumnShrinkable()** lassen sich Spalten auswählen, die verkleinert werden dürfen, um fehlenden horizontalen Platz zu beschaffen.
- **TableLayout**-Kindelemente haben beim Attribut **layout\_width** grundsätzlich den Wert **match\_parent**.

Die relativ einfachen Anforderungen beim Landschafts-Layout für unseren Primzahlendetektor leistet das **TableLayout** problemlos:



In der ersten Zeile befindet sich ein **TableRow**-Element, das ein **TextView**, ein **EditText** und ein **Button**-Element enthält. Über den Wert 1 für das Attribut **android:stretchColumns** zum **TableLayout**-Wurzelement wird dafür gesorgt, dass die Spalte mit dem **EditText**-Element in horizontaler Richtung den unbelegten Platz konsumiert. In der zweiten Zeile befindet sich (ohne

**TableRow**-Hülle das **TextView**-Element mit dem Ergebnis, das demzufolge horizontal zentriert erscheint.

Im Vergleich zum **RelativeLayout** (vgl. Abschnitt 8.3.2) ist das Ergebnis besser (wegen der Restplatzverwertung) und der Aufwand geringer:

```
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:stretchColumns="1"
    android:gravity="center">

    <TableRow>
        <TextView
            android:id="@+id/prompt"
            android:text="@string/prompt"
            android:layout_marginLeft="@dimen/activity_horizontal_margin" />

        <EditText
            android:id="@+id/candidate"
            android:layout_marginLeft="@dimen/activity_horizontal_margin"
            android:layout_marginRight="@dimen/activity_horizontal_margin"
            android:inputType="number"
            android:maxLength="18"
            android:background="@android:drawable/editbox_background"
            android:gravity="center">
        </EditText>

        <Button
            android:id="@+id/button"
            android:text="@string/diag"
            android:textAllCaps="false"
            android:layout_marginRight="@dimen/activity_horizontal_margin"/>
    </TableRow>

    <TextView
        android:id="@+id/result"
        android:layout_marginTop="@dimen/activity_vertical_margin"
        android:gravity="center"
        android:freezesText="true"/>
</TableLayout>
```

Vorteilhaft ist beim **TableLayout**, dass man für die Kindelemente keine Breite (**android:layout\_width**) und Höhe (**android:layout\_height**) angeben muss.

### 8.3.6 GridLayout

Das mit Android 4 eingeführte **GridLayout** ist ein performanter und flexibler Vielzweck-Layouter. Allerdings setzt seine erfolgreiche Verwendung eine Beschäftigung mit der Verarbeitung der Layout-Parameter voraus (Jackson 2014, S. 260).

Wie beim **TableLayout** (siehe Abschnitt 8.3.5) wird eine Matrixanordnung der Kindelemente ermöglicht. Dabei hat das **GridLayout** u.a. die folgenden Vorteile:

- Ein Element darf sich nicht nur über mehrere Spalten, sondern auch über mehrere Zeilen erstrecken.
- Es sind keine zwischengeschalteten Layout-Manager für die Zeilen erforderlich, und von dieser flacheren Hierarchie profitieren die Layout-Performanz und die Speichereffizienz.

Wenn ein Kindelement keinen Ortswunsch äußert, wird eine automatische Platzierung aufgrund der Container-Attribute **android:orientation**, **android:rowCount** und **android:columnCount** vorgenommen, z. B.:

```
<GridLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:columnCount="2" >
    .
    .
</GridLayout>
```

Bei der voreingestellten horizontalen Orientierung erfolgt die Verteilung der Steuerelemente zeilen-dominant, d.h. zuerst wird die erste Zeile von links nach rechts gefüllt, dann die zweite Zeile etc.

Bei der vertikalen Orientierung erfolgt die Verteilung der Steuerelemente spaltendominant, d.h. zuerst wird die erste Spalte von oben nach unten besetzt, dann die zweite Spalte etc.

Die Kindelemente in einem **GridLayout**-Container können über die Attribute **android:layout\_row** und **android:layout\_column** eine Matrixzelle ansteuern und außerdem per **android:columnSpan** bzw. **android:rowSpan** eine Erstreckung über mehr als eine Spalte bzw. Zeile und so einen beliebigen rechteckigen Bereich benachbarter Zellen beantragen, z. B.:

```
<TextView
    android:text="layout_gravity = "fill"\nrowSpan=\"2\""
    android:background="#f1e389"
    android:layout_column="1" android:layout_row="4"
    android:layout_rowSpan="2"
    android:gravity="center"
    android:layout_gravity="fill" />
```

Für Abstände zwischen den Steuerelementen kann auf unterschiedliche Weise gesorgt werden:

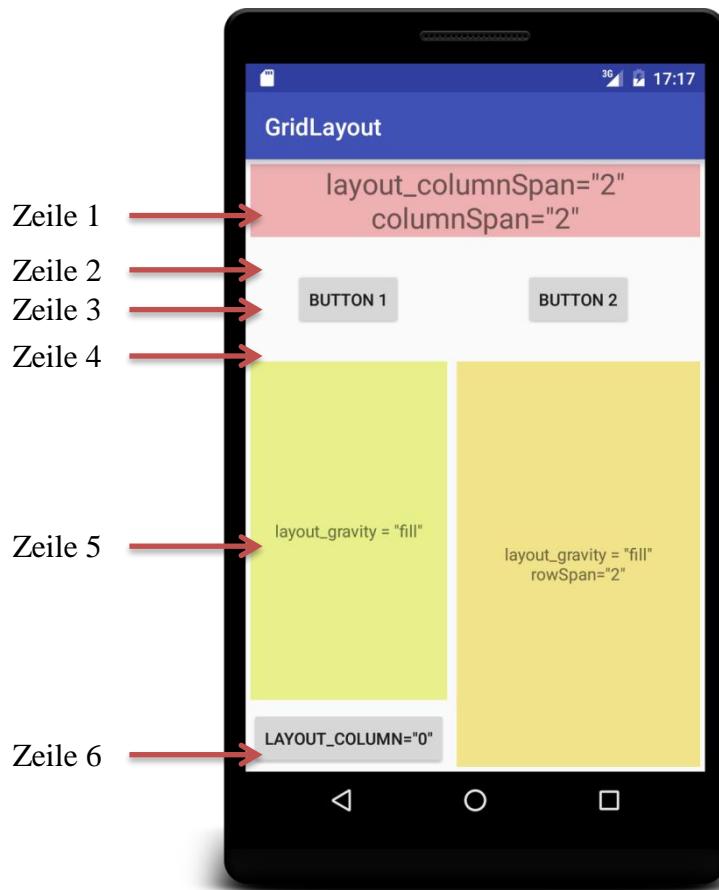
- Erhält das **GridLayout**-Element den Wert **true** für das XML-Attribut **android:useDefaultMargins**, dann orientieren sich die Abstände zwischen den Kindelementen am aktuelle Design-Theme der Anwendung, z. B.:  
**android:useDefaultMargins="true"**

Die Kindelemente können über die Attribute **android:layout\_marginLeft** etc. Wünsche äußern, die ggf. die allgemeine Einstellung dominieren.

- Das unsichtbare Abstandshalter-Steuerelement **Space** sorgt für einen Spalten- oder Zeilen-abstand, ohne in jeder Zelle der betroffenen Spalte bzw. Zeile wiederholt werden zu müssen, z. B.:

```
<Space
    android:layout_height="16dp" />
```

Im folgenden Beispiel mit 6 Zeilen und 2 Spalten

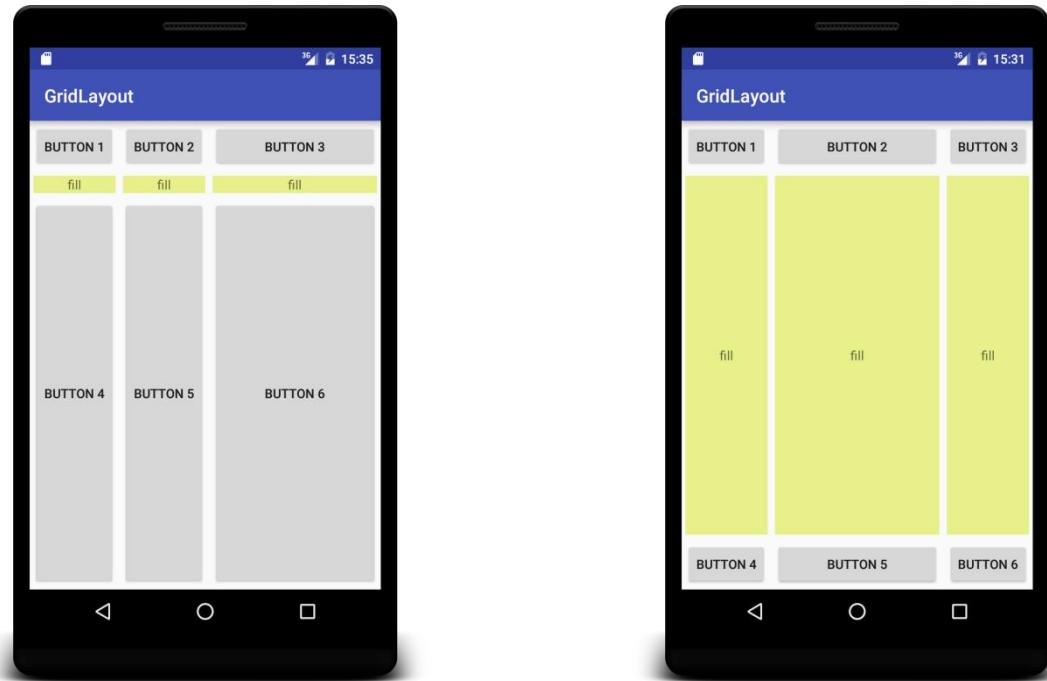


belegt ein **TextView**-Element die beiden Spalten der ersten Zeile. Das **TextView**-Element unten rechts belegt die Zeilen 5 und 6 der rechten Spalte. Vor und hinter der Zeile mit den **Button**-Elementen befindet sich eine Zeile mit je einem **Space**-Element in der linken Spalte.

Bei der Restraumverwertung ist die ursprüngliche **GridLayout**-Variante (aus Android 4) wenig flexibel und bevorzugt die Ränder (rechts bzw. unten). Im folgenden Beispiel ist für alle Bedienelemente über das XML-Attribut **android:layout\_gravity** angeordnet, in horizontaler und in vertikaler Richtung den verfügbaren Platz vollständig auszufüllen:

**android:layout\_gravity="fill"**

Im Ergebnis werden die rechte Spalte und die untere Zeile gestreckt (siehe linke Abbildung):



Seit der API-Version 21 sind die XML-Attribute **android:layout\_columnWeight** und **android:layout\_rowWeight** verfügbar, sodass es analog zum **LinearLayout** möglich ist, andere bzw. mehrere Spalten bzw. Zeilen von Raumreserven profitieren zu lassen, z. B.:

```
<TextView
    android:text="fill"
    android:background="#e8f189"
    android:gravity="center"
    android:layout_gravity="fill"
    android:layout_weight="1" android:layout_columnWeight="1" />
```

Im Beispiel wird nun die *mittlere* Spalte bzw. Zeile gestreckt, wenn alle anderen Spalten bzw. Zeilen das voreingestellte Gewicht 0 behalten (siehe rechte Abbildung).

Im Zusammenhang mit der Restraumverwertung über Gewichte spielt der Begriff der *Flexibilität* von Steuerelementen und Spalten bzw. Zeilen eine wichtige Rolle:

- Ein Steuerelement ist *flexibel*, wenn sein Attribut **android:layout\_gravity** einen Wert erhalten hat, z. B.:
 

```
        android:layout_gravity="fill"
        android:layout_gravity="center_horizontal"
```

 oder ein Spalten- bzw. Zeilengewicht vergeben worden ist, z. B.:
 

```
        android:layout_columnWeight="1"
        android:layout_rowWeight="1"
```
- Eine Spalte bzw. Zeile ist flexibel, wenn alle darin enthaltenen Elemente flexibel sind.

Damit eine Spalte bzw. Zeile bei der Verwertung von Restraum ihre Breite bzw. Höhe ändern kann, muss sie flexibel sein. Soll eine Spalte bzw. Zeile ihre Ausdehnung beibehalten, genügt die Anwesenheit *eines* nicht-flexiblen Elements.

## 8.4 Ereignisbehandlung

Ereignisse werden in eine Warteschlange nach dem FIFO-Prinzip (First In, First Out) gestellt und nacheinander abgearbeitet. Wegen der Single-Thread - Architektur des UI-Systems laufen die Ereignisbehandlungsmethoden strikt nacheinander ab: Solange eine Ereignisbehandlungsmethode läuft, findet keine weitere Ereignisbehandlung statt.

Wer über die Erläuterungen in diesem Abschnitt hinaus noch weitere Informationen zur Ereignisbehandlung benötigt, wird z. B. auf Googles Webseiten für Android-Entwickler fündig.<sup>1</sup>

### 8.4.1 Elementare und aufbereitete Ereignisse

Ein Touch-Ereignis wird an denjenigen Endknoten des **View**-Baums ausgeliefert, in dessen Rechteck die Berührung stattgefunden hat. Hier wird die Ereignisbehandlungsmethode **onTouchEvent()** der Klasse **View** aufgerufen, die per Parameter ein Ereignisobjekt aus der Klasse **MotionEvent** erhält. Der primäre *event handler* **onTouchEvent()** beurteilt das Elementarereignis bzw. eine Serie von Elementarereignissen und ruft z. B. unter bestimmten Bedingungen (Berührung im Rechteck des Steuerelements, Kontakt beendet ohne vorheriges Verlassen des Rechtecks) die **View**-Methode **performClick()** auf:<sup>2</sup>

```
public boolean performClick() {
    final boolean result;
    final ListenerInfo li = mListenerInfo;
    if (li != null && li.mOnClickListener != null) {
        playSoundEffect(SoundEffectConstants.CLICK);
        li.mOnClickListener.onClick(this);
        result = true;
    } else {
        result = false;
    }

    sendAccessibilityEvent(AccessibilityEvent.TYPE_VIEW_CLICKED);

    notifyEnterOrExitForAutoFillIfNeeded(true);

    return result;
}
```

Aus einer Serie von elementaren Touch-Ereignissen wird also unter Umständen ein Klick-Ereignis. Ist eine Tastatur vorhanden, führt auch ein Druck auf die Enter-Taste zu einem Klick-Ereignis, wenn eine Schaltfläche in diesem Moment den Eingabefokus besitzt. Sofern beim betroffenen **View**-Objekt ein Interessent für Klick-Ereignisse registriert ist,

`li != null && li.mOnClickListener != null`

wird dessen **onClick()** - Methode (als sekundärer event handler) von **performClick()** aufgerufen.

Neben dem Klick-Ereignis bietet die Klasse **View** noch viele andere Ereignisse an, erkennbar an der Existenz einer **setOnXxxListener()** - Methode, z. B.:

- **setOnContextMenuListener()**
- **setOnDragListener()**
- **setOnLongClickListener()**

Abgeleitete Klassen kennen weitere Ereignisse.

<sup>1</sup> <http://developer.android.com/guide/topics/ui/ui-events.html>

<sup>2</sup> Das Quellcodesegment stammt aus der Klasse **View** im Paket **android.view** der Android-Version 8.1 (API-Level 27).

Auch eigene Klassen können Ereignisse bzw. Registrierungsmethoden anbieten, um die asynchrone Kommunikation mit anderen Klassen zu unterstützen. Eine Klasse könnte per Listener-Technik Interessenten über das Ergebnis einer längeren Recherche oder Berechnung informieren.

Wir haben als Nutzer der GUI-Bibliothek zwei Möglichkeiten, in Reaktion auf Ereignisse ins Spiel zu kommen:

- Wenn man an einem aufbereiteten Ereignis interessiert ist (z. B. Klick beim Schalter zur Anforderung einer Primzahlendiagnose im Beispielprogramm **Prime Detection**), dann ist es sinnvoll, die originale Steuerelementklasse zu verwenden und einen Empfänger für das aufbereitete Ereignis zu registrieren.
- Wenn man an Elementarereignissen interessiert ist, muss man die betroffene Steuerelementklasse (z. B. **Button**) überbauen und in der Ableitung die geerbten Ereignisbehandlungsmethoden (z. B. **onTouchEvent()**) überschreiben. Dieser Aufwand ist angemessen, wenn Ereignisreaktionen nahe beim Steuerelement stattfinden sollen (z. B. eine spezielle optische, akustische oder haptische Reaktion).

Auf das Überschreiben von Ereignisbehandlungsmethoden in **View**-Ableitungen werden wir im Kurs verzichten. Im Zusammenhang mit dem Registrieren von Ereignisinteressenten behandeln wir nun Alternativen zum bisher in unseren Apps eingesetzten Verfahren.

#### 8.4.2 Optionen zum Registrieren von Ereignisempfängern

Wir haben bisher Klick-Ereignisse von den Aktivitäts-Objekten behandeln lassen und dazu ...

- im Kopf der Aktivitätsklassendefinition angekündigt, das (in der Klasse **View** definierte) Interface **OnClickListener** zu implementieren, z. B.:

```
public class PrimeActivity extends ActionBarActivity  
    implements View.OnClickListener {...}
```

- die im Interface geforderte Method **onClick()** in der Aktivitätsklasse implementiert
- und das Aktivitätsobjekt beim **View**-Objekt als Click-Listener registriert, z. B.:

```
Button button = findViewById(R.id.button);  
button.setOnClickListener(this);
```

Man kann nur *einen* **OnClickListener** bei einem **View**-Objekt registrieren. Bei einem Aufruf der Methode **setOnClickListener()** wird ein bereits vorhandener Listener ggf. ersetzt.

Zu jedem Ereignis (z. B. **Click**, **LongClick**) gehört ein Interface, das von einem Ereignisempfänger implementiert werden muss (z. B. **OnClickListener**, **OnLongClickListener**). Im Android-API beschränken sich die Event Listener - Interfaces meist auf eine Methode (z. B. **onClick()**, **onLongClick()**).

##### 8.4.2.1 Ereignisbehandlung durch eine anonyme Klasse

Alternativ kann man der Registrierungsmethode als Aktualparameter ein Objekt einer anonymen, spontan definierten Klasse übergeben, z. B.:

```

private CheckBox cbBold;
    ...
cbBold = findViewById(R.id.cbBold);
    ...
cbBold.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener() {
    @Override
    public void onCheckedChanged(CompoundButton cb, boolean isChecked) {
        Typeface typeface = tvSample.getTypeface();
        int style = typeface.getStyle();
        Typeface actTFace = rbSans.isChecked() ? Typeface.SANS_SERIF : Typeface.SERIF;
        if (isChecked)
            tvSample.setTypeface(actTFace,
                (style & Typeface.ITALIC) != 0 ? Typeface.BOLD_ITALIC : Typeface.BOLD);
        else
            tvSample.setTypeface(actTFace,
                (style & Typeface.ITALIC) != 0 ? Typeface.ITALIC : Typeface.NORMAL);
    }
});
```

Hier wird beim Steuerelement `cbBold` aus der Klasse **CheckBox** (siehe Abschnitt 8.5.4.1) ein **CompoundButton.OnCheckedChangeListener** (ein Interessent für Änderungen des Markierungszustands) registriert, wobei nicht nur das horchende Objekt dynamisch erzeugt, sondern die gesamte Klasse an Ort und Stelle definiert wird. Die in der anonymen Klasse zur Schriftverwaltung verwendeten Methoden `get Typeface()`, `getStyle()` und `set Typeface()` und weitere Details werden in Abschnitt 8.5.4.1 erläutert.

Eine solche **anonyme Klasse** hat die Möglichkeit, in ihren Methoden auf Felder und Methoden der umgebenden Klasse zuzugreifen, was im Beispiel allerdings nicht erforderlich ist.

Einige Eigenschaften von anonymen Klassen (siehe z. B. Baltes-Götz & Götz 2018, Abschnitt 12.1.1.2):

Definition und Instantiierung finden in einem `new`-Operanden statt, wobei im Konstruktorauftruf der fehlende Klassename durch den Namen der implementierten Schnittstelle oder den Namen der beerbten Basisklasse vertreten wird. Es folgt ein Klassendefinitionsblock, der wie üblich durch geschweifte Klammern zu begrenzen ist. Im Beispiel wird die Schnittstelle **CompoundButton.OnCheckedChangeListener** angegeben und deren (einige) Methode `onCheckedChanged()` implementiert. Es kann nur eine einzige Instanz erzeugt werden. Werden mehrere Instanzen benötigt, ist eine anonyme Klasse ungeeignet.

- Weil der Klassename fehlt, sind keine Konstruktoren möglich. Über die Instanzinitialisierer ist jedoch eine Ersatzlösung verfügbar.
- Statische Methoden sind verboten, und statische Variablen müssen finalisiert sein.
- Der Compiler erzeugt auch für eine anonyme Klasse eine eigene **class**-Datei, in deren Namen der Bezeichner für die umgebende Klasse eingeht. Bei einer umgebenden Klasse mit dem Namen **CheckRadioActivity** landet die erste anonyme Klasse in der Datei **CheckRadioActivity\$1.class**.

#### 8.4.2.2 Ereignisbehandlung durch einen Lambda-Ausdruck

In der Programmiersprache Java können seit der Version 8 die sogenannten *Lambda-Ausdrücke* dazu verwendet werden, *Boilerplate-Code* zu vermeiden (siehe z. B. Baltes-Götz & Götz 2018, Abschnitt 12.1). Mit dieser abwertenden Bezeichnung sind Syntaxfragmente gemeint, die sich häufig in weitgehend identischer Form wiederholen und daher lästig werden. Die seit Java 8 mögliche Realisierung einer Ereignisbehandlungsmethode durch einen Lambda-Ausdruck enthält im Vergleich zur traditionellen Lösung durch eine anonyme Klasse weniger monoton und umständlich zu wiederholenden Boilerplate-Code z. B.:

```

cbBold.setOnCheckedChangeListener((cb, isChecked) -> {
    Typeface typeface = tvSample.getTypeface();
    int style = typeface.getStyle();
    Typeface actTFace = rbSans.isChecked() ? Typeface.SANS_SERIF : Typeface.SERIF;
    if (isChecked)
        tvSample.setTypeface(actTFace,
            (style & Typeface.ITALIC) != 0 ? Typeface.BOLD_ITALIC : Typeface.BOLD);
    else
        tvSample.setTypeface(actTFace,
            (style & Typeface.ITALIC) != 0 ? Typeface.ITALIC : Typeface.NORMAL);
});
```

Hinter den Kulissen bleibt im Wesentlichen alles beim Alten, wobei der Compiler viele Routinearbeiten übernimmt:

- Er kennt den Parametertyp **CompoundButton.OnCheckedChangeListener** von **setOnCheckedChangeListener()** und akzeptiert einen Lambda-Ausdruck, der die erforderliche Methode **onCheckedChanged()** realisiert, sodass sich eine passende anonyme Klasse erstellen lässt.
- Die im Lambda-Ausdruck vor dem Pfeil (->) angegebenen Parameter müssen vom passenden Typ sein. Man kann jedoch auf eine Typangabe verzichten, wobei der Compiler die Typen aus der Interface-Definition entnimmt.
- Die vom Code-Block produzierte Rückgabe muss vom passenden Typ sein. Im Beispiel hat die Interface-Methode **onCheckedChanged()** den Rückgabetyp **void**, und eine **return**-Anweisung mit Rückgabe als Bestandteil des Lambda-Ausdrucks führt zu einer Fehlermeldung:

return 13;  
Unexpected return value

- Wenn das GUI-Framework später (nach einem Mausklick auf den Schalter) die **CompoundButton.OnCheckedChangeListener** - Methode **onCheckedChanged()** aufruft, wird der Code im Lambda-Ausdruck ausgeführt.

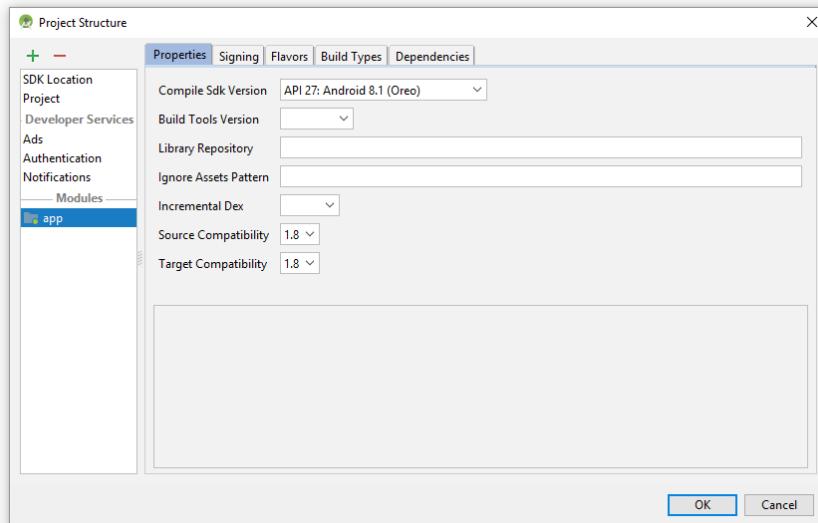
Damit Lambda-Ausdrücke oder andere Bestandteile von Java 8 in einem Android-Programm verwendet werden können, muss man nach dem Menübefehl

### File > Project Structure

im folgenden Dialog auf der Registerkarte **Properties** für das Modul app die

- **Source Compatibility** und die
- **Target Compatibility**

auf den Wert 1.8 setzen:



Alternativ kann man in der Modul-bezogenen Konfigurationsdatei **build.gradle** das Java-Kompatibilitätsniveau direkt setzen:

```
compileOptions {
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
}
```

#### 8.4.2.3 Klick-Ereignismethode per XML-Attribut registrieren

Die bisher beschriebenen Verfahren zum Registrieren von Ereignisinteressenten sind auf beliebige Ereignisse anwendbar. Wenn es nur darum geht, das Klick-Ereignis eines Steuerelements (z. B. vom Typ **Button**) durch die Activity-Klasse zu behandeln, lässt sich auf folgende Weise der Aufwand reduzieren:

- Man implementiert in der Activity-Klasse eine Ereignisbehandlungsmethode mit ...
  - dem Zugriffsmodifikator **public**,
  - exakt einem Parameter, der den Typ **View** besitzt
  - und einem beliebigen Namen.

Beispiel:

```
public void checkPrime(View v) {
    . . .
}
```

- In der Layout-Definition erhält die Ereignisquelle das Attribut **android:onClick** mit dem Namen der Methode als Wert, z. B.:

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/diag"
    android:id="@+id/button"
    android:layout_marginBottom="32dp"
    android:layout_marginTop="32dp"
    android:textAllCaps="false"
    android:onClick="checkPrime" />
```

#### 8.4.3 Ereignisverarbeitungskette (unterbrechen)

Zu einem Ereignis gehört ein Interface, das den Definitionskopf der Behandlungsmethode vorschreibt. In den bisherigen Beispielen war der Rückgabetyp **void** verlangt. Beim **LongClick** - Er-

eignis, das von einem **View**-Objekt nach einem längeren Touch gefeuert wird, ist hingegen der Rückgabetyp **boolean** vorgeschrrieben.

Sind bei einem **View**-Objekt eine **Click**- und eine **LongClick**-Behandlungsmethode registriert, läuft nach einer längeren Berührung durch den Benutzer folgende Ereignisverarbeitung ab:<sup>1</sup>

- Es wird zuerst die **LongClick**-Behandlungsmethode aufgerufen.
- Kehrt der Aufruf mit dem Wert **false** zurück, wird anschließend die **Click**-Behandlungsmethode aufgerufen.
- Kehrt der Aufruf der **LongClick**-Behandlungsmethode hingegen mit dem Wert **true** zurück, gilt das Ereignis als erledigt, und es wird keine weitere Behandlungsmethode aufgerufen.

## 8.5 Bedienelemente

### 8.5.1 Beschriftungen (TextView)

Die Klasse **TextView** wird für statische oder variable Beschriftungen in der Bedienoberfläche verwendet. Sie bietet die passende Gelegenheit zur Beschäftigung mit den in Android verfügbaren Schriftarten, und auch noch einige andere **TextView**-Optionen verdienen eine kurze Behandlung. Außerdem ist **TextView** die Basisklasse des **EditText**-Steuerelements, das für Texteingabefelder verwendet wird. Viele Attribute, die in Abschnitt 8.5.2 im Zusammenhang mit dem Texteingabefeld (Klasse **EditText**) beschrieben werden, sind schon in der Klasse **TextView** definiert, werden aber hier selten benötigt.

#### 8.5.1.1 Schriftfamilien und -attribute

Mit dem XML-Attribut **android:typeface** oder der korrespondierenden Methode **setTypeface()** kann man die Schriftart auf einen von den folgenden Werten setzen: **sans**, **serif**, **monospace**, **normal** (Voreinstellung: **sans**), z. B.:

```
android:typeface="serif"
```

Mit der folgenden **setTypeface()** - Überladung setzt man im Java-Quellcode die Schriftart und die Schriftauszeichnung:

```
public void setTypeface(Typeface tf, int style)
```

Den ersten Parameter kann man über **Typeface**-Objekte versorgen, die eine Schriftart repräsentieren und durch öffentliche, statische und finalisierte Felder der Klasse **Typeface** ansprechbar sind:

- **Typeface.SANS\_SERIF**  
Serifenfreie Proportionalschrift
- **Typeface.SERIF**  
Proportionalschrift mit Serifen
- **Typeface.MONOSPACE**  
Gleichabständige Schrift

Den zweiten Parameter kann man über die **int**-wertigen **Typeface**-Konstanten aus der folgenden Tabelle versorgen, um einen Schriftstil zu wählen:

---

<sup>1</sup> Quelle: [http://www.techotopia.com/index.php/An\\_Overview\\_and\\_Example\\_of\\_Android\\_Event\\_Handling](http://www.techotopia.com/index.php/An_Overview_and_Example_of_Android_Event_Handling)

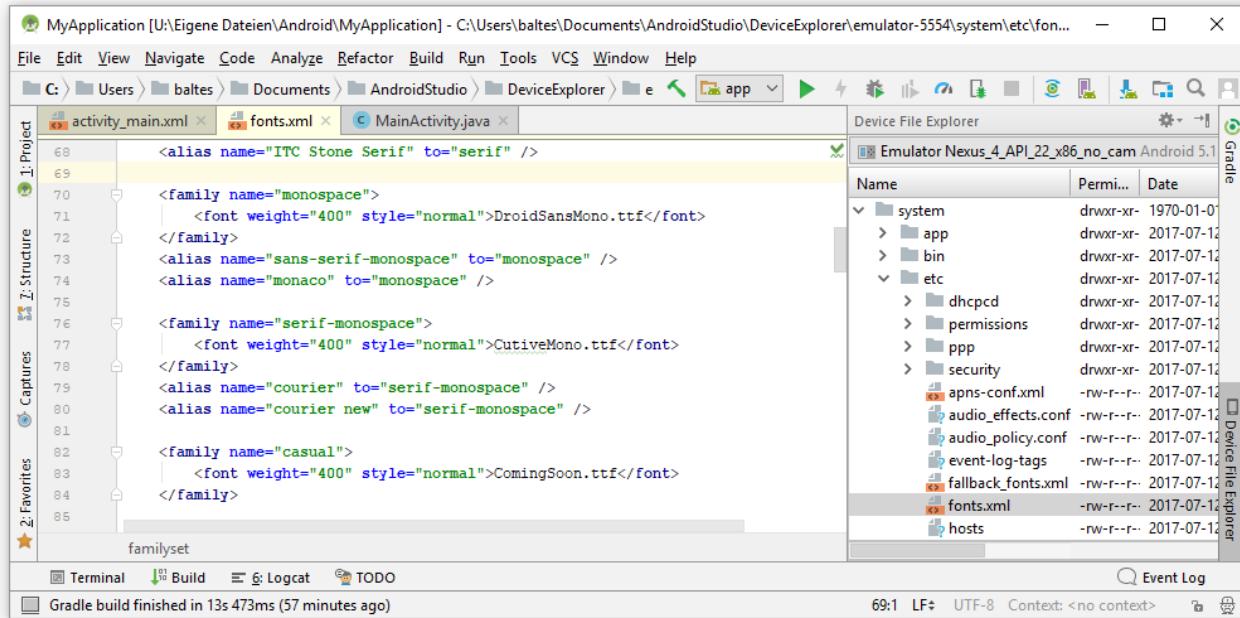
int-Wert	Stil	Typeface-Konstante
0	Standard	NORMAL
1	Fett	BOLD
2	Kursiv	ITALIC
3	Fett + Kursiv	BOLD_ITALIC

Über die Methode **get Typeface()** erhält man von einem **TextView**-Objekt eine Referenz auf die verwendete Schriftart. Das zurück gelieferte **Typeface**-Objekt lässt sich über **getStyle()** nach der Schriftauszeichnung befragen, wobei der int-wertigen Rückgabe die Kodierung aus der obigen Tabelle zugrunde liegt.

Mit dem XML-Attribut **android:textStyle** lassen sich zur Entwurfszeit die folgenden Textauszeichnungen setzen: **normal**, **bold**, **italic**. Um **bold** und **italic** gleichzeitig zu setzen, gibt man beide Auszeichnungen durch den Operator | getrennt an:

**android:textStyle="bold|italic"**

Die oben präsentierte Liste mit systemweit in Android verfügbaren Schriftfamilien ist nicht üppig. Auf einem konkreten Android-Gerät sind in der Regel mehr Schriftfamilien verfügbar, was ein Blick in die Datei **/system/etc/fonts.xml** belegt. Bei einem emulierten Gerät lässt sich der Inhalt dieser Datei bequem über den **Device File Explorer** im Android Studio einsehen, bei einem Smartphone mit Android 5.1:



Eine Anwendung steht es frei, für den eigenen Gebrauch weitere Schriftarten zu installieren.

Mit dem XML-Attribut **android:textSize** oder der korrespondierenden Methode **setTextSize()** setzt man die Textgröße, wobei die Maßeinheit **sp** (vgl. Abschnitt 7.5) verwendet werden sollte, z. B.:

**android:textSize="15sp"**

### 8.5.1.2 Attribut `freezesText`

Ein **TextView**-Objekt mit *statischem* Inhalt muss im Quellcode nicht angesprochen werden, sodass eine Element-ID überflüssig ist. Wenn ein **TextView**-Objekt zu Anzeige variabler Beschriftungen dient (und dementsprechend auch eine Element-ID besitzt), ist zu beachten, dass die automatische Sicherung und Wiederherstellung des Inhalts im Rahmen der **Activity** - Lebenszyklus-Methoden **onSaveInstanceState()** und **onRestoreInstanceState()** (siehe Abschnitt 6) nur dann klappt, wenn in der Layoutdefinition das Attribut **android:freezesText** den Wert **true** erhält,

```
<TextView  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:freezesText="true"  
    android:id="@+id/result" />
```

oder der analoge Effekt mit der Methode **setFreezesText()** erreicht wird.

Aus der App zur Primzahlendiagnose (siehe Abschnitt 7.10) kennen wir die Methode **setText()**, mit der man den von einem **TextView**-Objekt anzuzeigenden Text im Programm ändern kann. Das Gegenstück **getText()**, das den aktuell angezeigten Text liefert, befindet sich ebenfalls im Handlungsrepertoire der Klasse **TextView**, wird aber hier sehr viel seltener benötigt als bei der Ableitung **EditText** (siehe Abschnitt 8.5.2).

### 8.5.1.3 Attribut `autoLink`

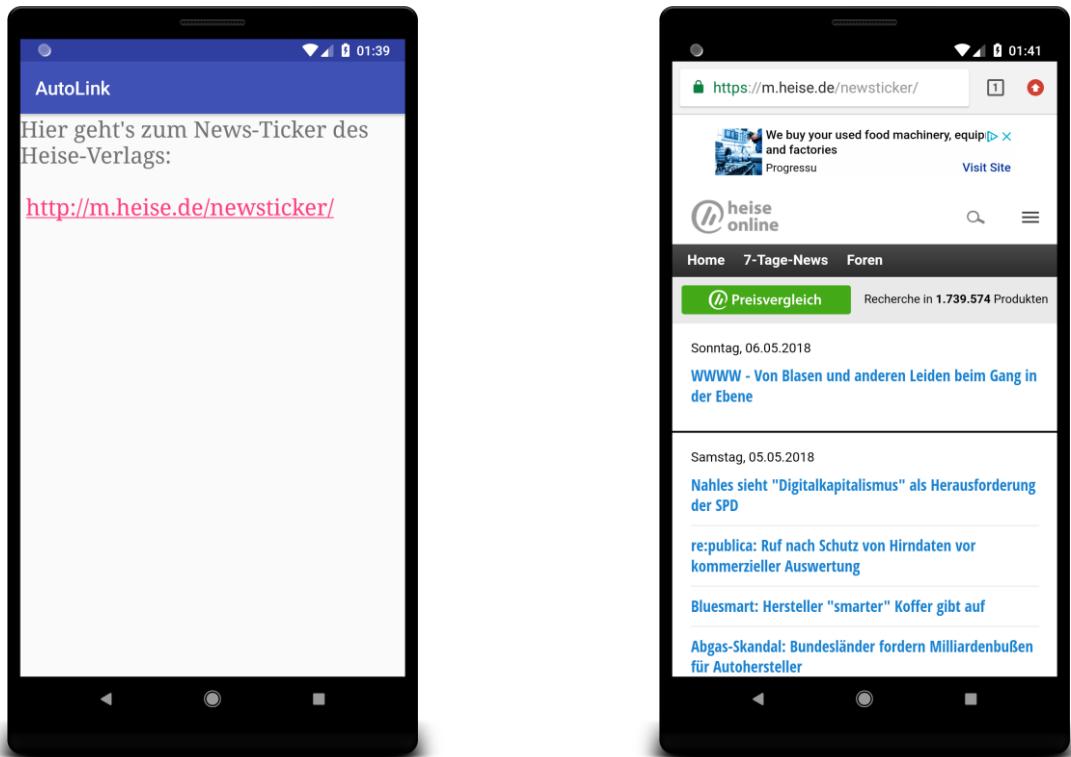
Über das XML-Attribut **android:autoLink** oder die korrespondierende Methode **setAutoLinkMask()** sorgt man dafür, dass im Text enthaltenen Internet-Adressen (z. B. Mail-Adressen oder URLs) erkannt und in klickbare Links umgesetzt werden. Aufgrund des **autoLink**-Attributs zum folgenden **TextView**-Steuerelement

```
<TextView  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="@string/link_demo"  
    android:typeface="serif"  
    android:textSize="24sp"  
    android:autoLink="web" />
```

wird die **string**-Ressource `link_demo`

```
<resources>  
    <string name="app_name">AutoLink</string>  
    <string name="link_demo">Hier geht's zum News-Ticker des Heise-Verlags:\n\n  
        https://m.heise.de/newsticker/</string>  
</resources>
```

wie im linken Bildschirmfoto angezeigt, und ein Klick auf den Link öffnet die verlinkte Seite im Browser:



Nebenbei demonstriert die im Programm genutzte **string**-Ressource, dass man einen Zeilenwechsel per Newline-Escape-Sequenz (**\n**) anfordern kann.

### 8.5.2 Texteingabefelder (EditText)

Zur Erfassung von (ein- oder mehrzeiligen) Texteingaben verwendet man ein Objekt der von **TextView** abstammenden Klasse **EditText** und deklariert es in der Regel über ein <**EditText**> - Element in der Layoutdefinition. Das Steuerelement besitzt etliche Kompetenzen als Texteditor inklusive der Zwischenablagen-Kooperation über ein Kontextmenü, das per Langklick angefordert wird.

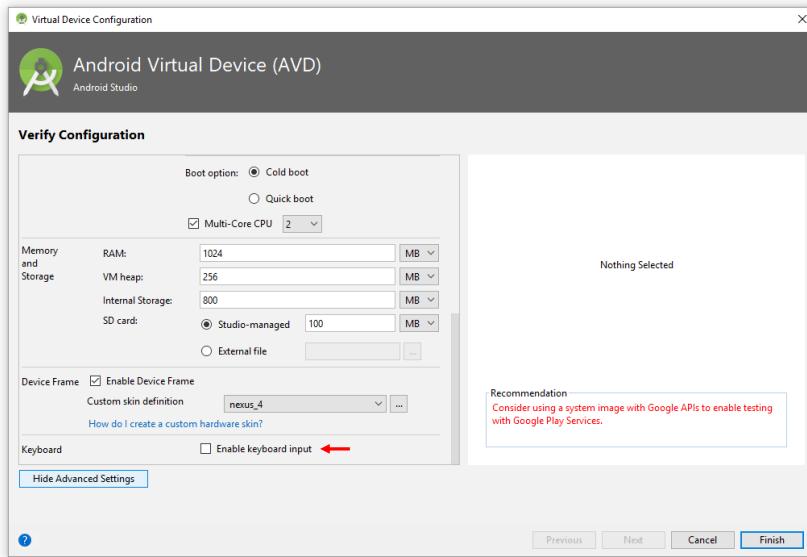
Bei einem Android-Gerät mit Touchscreen hat ein Touch auf ein **EditText**-Objekt zur Folge, dass

...

- eine virtuelle Tastatur erscheint,
- und alle Eingaben zum getroffenen Element wandern.

Wenn die virtuelle Tastatur zum Beenden der Eingabe keine spezielle Taste (z. B. **Fertig**, siehe unten) anbietet, kann sie mit dem Rückwärtsschalter zum Verschwinden gebracht werden.

Damit ein emuliertes Android-Gerät wie ein typisches Smartphone mit einer *virtuellen* Tastatur arbeitet, statt die reale Tastatur des Wortsrechners zu unterstützen, darf in seinem Eigenschaftsdialog (im **Android Virtual Device Manager** über die Schaltersequenz und **Show Advanced Settings** erreichbar) das Kontrollkästchen zum Aktivieren der Hardware-Tastatur *nicht* markiert sein:

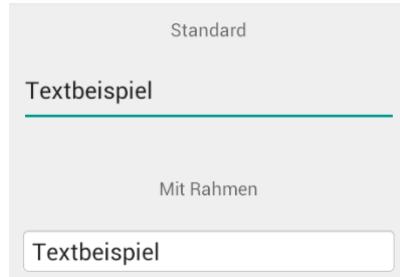


### 8.5.2.1 Optik

Über das Attribut **android:background** unter Verwendung einer vom Android-System spendierten Neun-Feld - Bitmap (vgl. Abschnitt 7.7.1)

**android:background="@android:drawable/editbox\_background"**

kann man den Texteingabefeldern eine attraktive (gerahmte) Optik verschaffen, was wir schon mehrfach getan haben (siehe Beschreibung auf Seite 53).



### 8.5.2.2 Irreguläre Eingaben verhindern

Über das XML-Attribut **android:inputType** kann man für ein Texteingabefeld die zulässigen Zeichen beschränken, wobei sich die virtuelle Tastatur automatisch entsprechend anpasst. Das erspart dem Benutzer Zeitverlust durch Fehlversuche, und für den Entwickler reduziert sich der Aufwand bei der Kontrolle der Benutzereingaben. Im Beispielprogramm **Prime Detection** haben wir auf diese Weise ein Texteingabefeld realisiert, dessen virtuelle Tastatur



nur positive ganze Zahlen zulässt:

```
<EditText
    android:id="@+id/candidate"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:gravity="center_horizontal"
    android:inputType="number"
    android:maxLength="18"
    android:ems="10"
    android:background="@android:drawable/editbox_background"
    android:layout_marginTop="24dp"
    android:enabled="true" />
```

Das Attribut **android:inputType** erlaubt noch andere Werte mit Einfluss auf das Layout der virtuellen Tastatur, z. B.

- **numberSigned, numberDecimal**

Es erscheint eine Tastatur, mit der sich neben Ziffern auch ein Minuszeichen (als erstes Zeichen) bzw. ein Dezimaltrennzeichen eingeben lässt. Sollen negative Dezimalzahlen erlaubt sein, muss man beide Werte über den | - Operator verknüpfen:

**android:inputType="numberDecimal|numberSigned"**

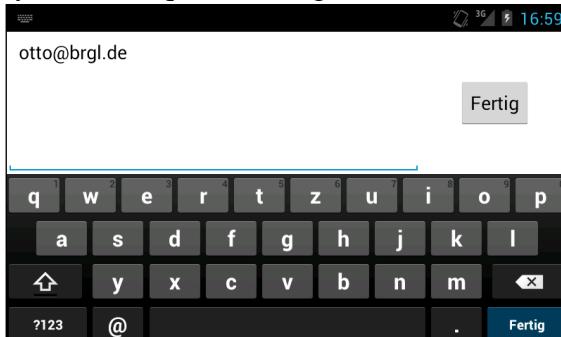
Im Android Studio kann man statt des XML-Editors natürlich auch das **Attributes**-Fenster verwenden, um die erlaubten Zeichen für ein **EditText**-Steuerelement festzulegen.

- **phone**

Man erhält das Tastaturlayout einer Telefon-App.

- **textEmailAddress**

Man erhält ein Tastaturlayout zur bequemen Eingabe einer Mail-Adresse, z. B.:



- **date, time**

Von der Verwendung dieser Eingabetypen muss abgeraten werden, weil viele Hersteller von Android-Geräten die Tastatur-Anwendung durch eine Eigenkreation ersetzen und dabei die Unterstützung der Eingabetypen **date** und **time** vernachlässigen. Oft erscheint ein numerisches Tastatur-Layout ohne Trennzeichen für Datums- bzw. Zeitbestandteile. Zur Erfassung von Datums- und Zeitangaben sollte statt eines **EditText**-Elements mit Eingabetyp ein Standarddialog vom Type **DatePicker** bzw. **TimePicker** verwendet werden.

Einige **inputType** - Werte steuern die *Verarbeitung* der Tastatureingaben, z. B.

- **textPassword**

Ein eingetipptes Zeichen wird kurz angezeigt und dann durch einen Punkt ersetzt.

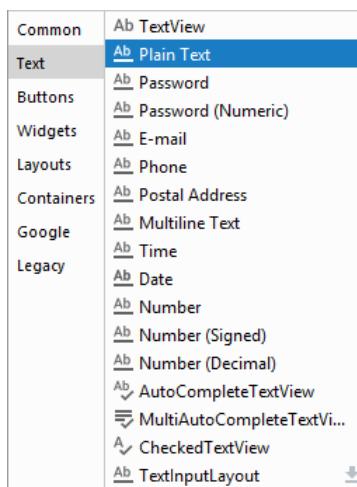
- **textCapCharacters**

Alle Eingaben werden als Großbuchstaben interpretiert.

- **textNoSuggestions**

Die Wordergänzungsvorschläge während der Eingabe werden abgeschaltet.

Für viele Eingabetypen bietet das Android Studio in der Palette **Text** des grafischen GUI-Designers ein vorkonfiguriertes Element:



Über die Methode **setInputType()** lassen sich Eingaberestriktionen per Programm (also zur Laufzeit) setzen oder aufheben.

Mit dem Attribut **android:maxLength** lässt sich die Anzahl erlaubter Zeichen zur Entwurfszeit beschränken. Zur Ausführungszeit ist dies durch die Methode **setFilters()** möglich, die zudem mit Hilfe geeigneter Objekte vom Typ **InputFilter** auch eine Konvertierung von Eingabezeichen vornehmen kann, um z. B. eine Beschränkung auf Großbuchstaben durchzusetzen.

### 8.5.2.3 Mehrzeiliges Texteingabefeld

Ein Texteingabefeld arbeitet im Einzeilenmodus, sobald sein Attribut **android:inputType** einen Wert erhält. In dieser Situation besitzt die zugehörige virtuelle Tastatur *keine* - Taste (siehe Abschnitt 8.5.2.2). Ist kein **inputType**-Wert vorhanden, unterstützt ein **EditText**-Steuerelement auch mehrzeilige Texte, und die zugehörige virtuelle Tastatur besitzt eine - Taste, z. B.:



Je nach verfügbarem Platz vergrößert sich das Steuerelement in vertikaler Richtung nach Betätigung der - Taste. Über das Attribut **android:lines** kann eine initial sichtbare Anzahl von Zeilen angefordert werden, wobei die tatsächliche Zahl durch den verfügbaren Platz begrenzt ist (speziell bei Anwesenheit der virtuellen Tastatur), z. B.:

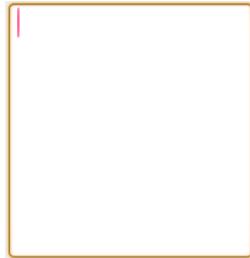
**android:lines="13"**

Außerdem sind in dieser Situation die folgenden Attribute von Interesse:

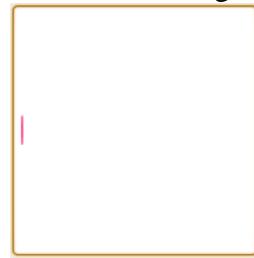
- **android:gravity**

Mit dem Wert **top** sorgt man dafür, dass die Texteingabe oben links beginnt:

**android:gravity="top"**



Voreinstellung



- **android:scrollbars**

Hat das Attribut den Wert **vertical**, erscheint beim vertikalen Wischen am rechten Rand des Textfelds ein Balken mit Positions- und Längeninformation zum Text:

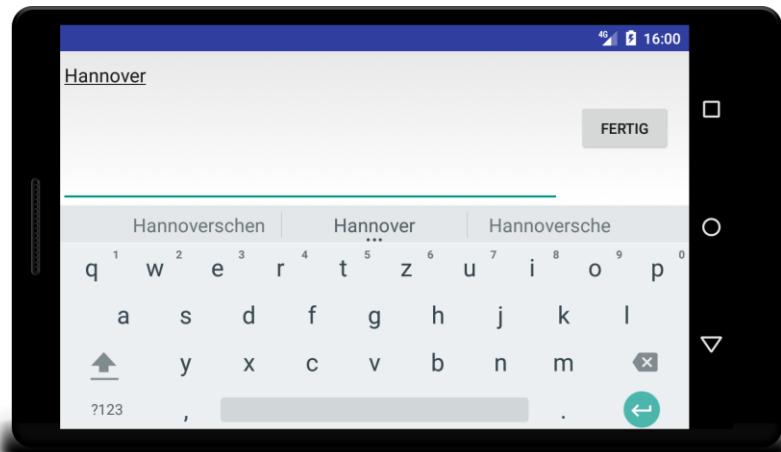


Beispiel:

```
<EditText
    android:id="@+id/et"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginBottom="8dp"
    android:layout_marginTop="8dp"
    android:background="@android:drawable/editbox_background"
    android:ems="10"
    android:gravity="top"
    android:lines="8"
    android:scrollbars="vertical" />
```

#### 8.5.2.4 Fullscreen-Texteingabe

Ist zu wenig Platz vorhanden, um eine Anwendung und die virtuelle Tastatur gemeinsam anzuzeigen (z. B. bei einem Smartphone im Querformat), dann schaltet Android auf die Fullscreen-Texteingabe um, wobei außer der virtuellen Tastatur nur das gerade zu bedienende Texteingabefeld zu sehen ist, z. B.:

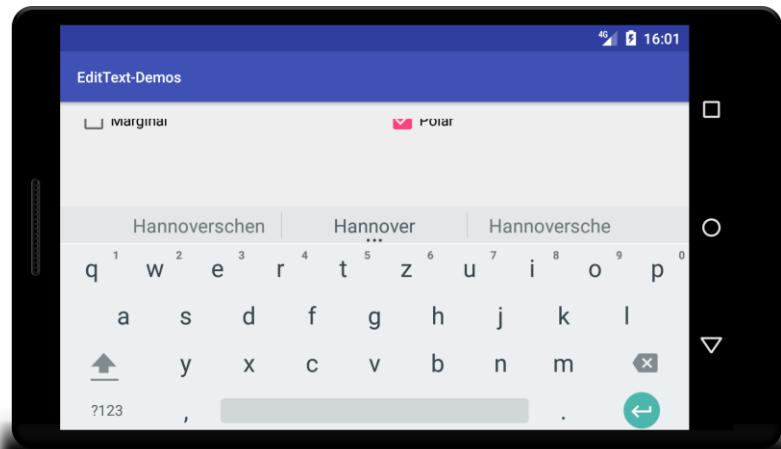


Neben dem Textfeld erscheint ein Schalter, dessen Beschriftung über das Attribut **android:imeActionButton** festgelegt werden kann.

Mit dem Wert **flagNoExtractUi** für das Attribut **android:imeOptions** lässt sich der Fullscreen-Texteingabemodus verhindern:

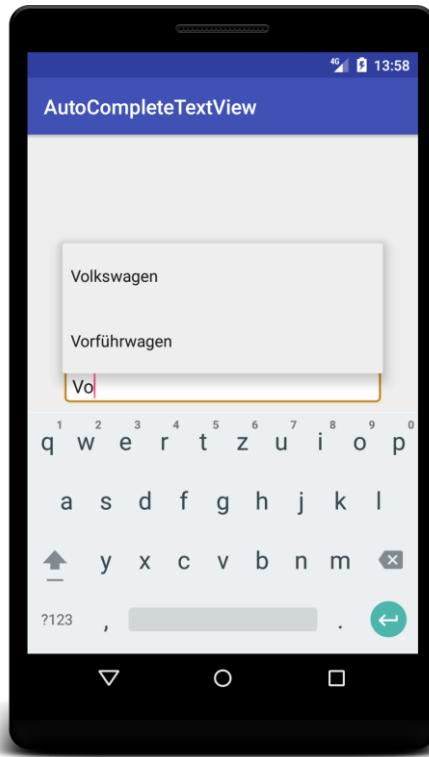
```
android:imeOptions="flagNoExtractUi"
```

Das ist aber oft nicht sinnvoll, z. B.:



### 8.5.2.5 AutoCompleteTextView

Wenn für einen zu erfragenden Text die möglichen Antworten meist aus einer begrenzten Liste stammen, dann kann die Eingabe durch ein Objekt der Klasse **AutoCompleteTextView** erleichtert werden. Wie bei der Code-Vervollständigung einer Programmentwicklungsumgebung erscheint zu einem Textanfang eine Drop-Down-Liste mit möglichen Vervollständigungen, aus der per Touch gewählt werden kann, z. B.:



Es ist dem Benutzer aber auch erlaubt, die Vorschläge ignorieren und einen beliebigen Text einzutragen.

Im Beispiel wird zur Realisation der Autovervollständigung ein **String**-Array mit deutschen Automarken angelegt. Daraus entsteht ein Objekt der generischen Klasse **ArrayAdapter<String>**, wobei im Konstruktor die Ressource **android.R.layout.simple\_dropdown\_item\_1line** das Layout festlegt. Dieser Adapter wird dem **AutoCompleteTextView**-Objekt zugewiesen:

```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        String[] germanCars = new String[]
                {"Volkswagen", "Vorführwagen", "BMW", "Mercedes", "Opel", "Ford", "Porsche", "Audi"};
        ArrayAdapter<String> adapter = new ArrayAdapter<>(this,
                android.R.layout.simple_dropdown_item_1line, germanCars);

        AutoCompleteTextView autoComplete = findViewById(R.id.autoCompleteTV);
        autoComplete.setAdapter(adapter);
    }
}
```

### 8.5.2.6 Weitere Attribute für EditText-Elemente

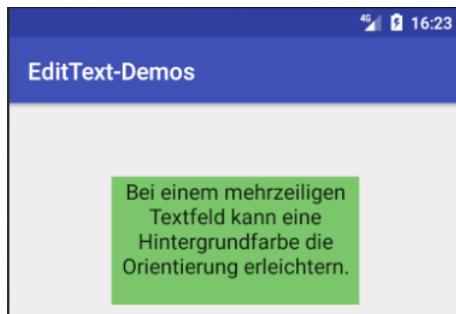
Mit dem Attribut **android:hint** vereinbart man einen Hinweistext, der in grauer Schriftfarbe im Texteingabefeld erscheint und beim Eintippen des ersten Zeichens verschwindet (siehe Beispiel in Abschnitt 8.5.2.4).

Hat das Attribut **android:layout\_width** den Wert **wrap\_content**, kann die Breite des Texteingabefelds über das Attribut **android:ems** festgelegt werden, z. B.:

**android:ems="5"**

Die Breite wird so gewählt, dass von der eingestellten Schriftart entsprechend viele Zeichen mit der maximalen Breite (nämlich M's) hineinpassen. Das Attribut **android:ems** erlaubt also eine Größenangabe in Relation zur eingestellten Schriftart.

Bei einem mehrzeiligen Texteingabefeld ist die in Abschnitt 8.5.2.1 beschriebene Rahmung besonders nützlich, weil die verfügbare Eingabezone erkennbar wird. Alternativ kann eine Hintergrundfarbe den Eingabebereich anzeigen, z. B.:



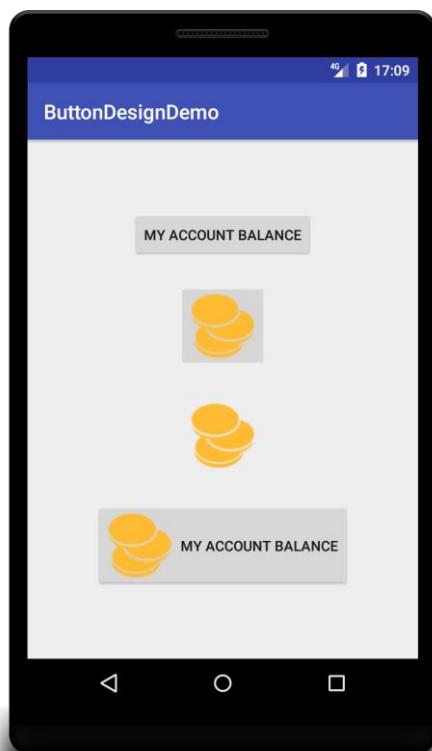
Man vereinbart die Hintergrundfarbe über das Attribut **android:background**, z. B.:

**android:background="#7BC66B"**

### 8.5.3 Schaltflächen

Mit einer Schaltfläche können die Benutzer eine Aktion auslösen oder ein Formular quittieren. Mit der Klasse **Button** (im Paket **android.widget**) zur Realisation von Schaltflächen haben wir schon mehrfach Bekanntschaft gemacht und insbesondere die Optionen zur Klick-Ereignisbehandlung kennengelernt (siehe Abschnitt 8.4), sodass in diesem Abschnitt nur einige Gestaltungsoptionen erwähnt werden sollen.

In der folgenden Bedienoberfläche (mit einem vertikal orientierten **LinearLayout**)



sind vier Schalter-Varianten zu sehen:

- **Button** (1. Schalter im Beispiel)

Das **Button**-Element ist weitgehend selbsterklärend:

```
<Button
    android:id="@+id/button1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/account"
    android:contentDescription="@string/account" />
```

Über das Attribut **android:contentDescription** vereinbart man einen Text, der als Bedienungserleichterung für Personen mit Sehbehinderung durch geeignete Software vorgelesen werden kann. Bei oft benötigten Standardbeschriftungen (z. B. **OK**, **Cancel**) kann man etwas Aufwand sparen und **string**-Ressourcen aus dem Paket **android** (inkl. Übersetzungen) verwenden (siehe Abschnitt 7.4).

- **ImageButton** (2. Schalter im Beispiel)

Ein Schalter aus der Klasse **ImageButton** zeigt statt einer Beschriftung ein Bild, das in der XML-Deklaration als zeichenbare Ressource über das Attribut **android:src** vereinbart wird:

```
<ImageButton
    android:id="@+id/button2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="24dp"
    android:src="@drawable/ic_coins_1"
    android:contentDescription="@string/account" />
```

Wie man eine Bitmap-Datei in ein AS-Projekt aufnimmt, wird gleich beschrieben.

- Schalter ohne Rand bzw. Hintergrund (3. Schalter im Beispiel)

Speziell bei einem **ImageButton** kommt der randlose Stil in Frage. In der XML-Deklaration wird dem **style**-Attribut (Achtung: ohne **android**-Präfix) das Attribut **borderlessButtonStyle** aus dem aktuellen Thema (engl.: *theme*) zugewiesen (vgl. Abschnitt 7.3.1 zu diesem Ressourcen-Zugriff):

```
<ImageButton
    android:id="@+id/button3"
    style="?android:borderlessButtonStyle"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@+id/button2"
    android:layout_marginTop="24dp"
    android:src="@drawable/ic_coins_1"
    android:contentDescription="@string/account" />
```

Damit der randlose Stil bei einem **ImageButton** voll zur Geltung kommt, sollte ein Bild mit transparenter Hintergrundfarbe verwendet werden. Bei einer PNG-Datei eine Farbe als transparent zu deklarieren, gelingt z. B. mit der Windows-Freeware IrfanView.

- Schalter mit Beschriftung *plus* Symbol (4. Schalter im Beispiel)

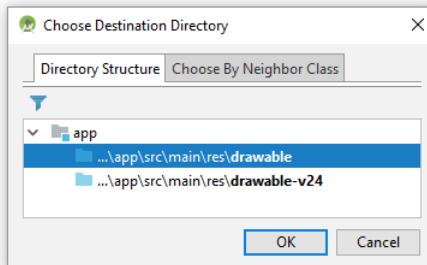
Sollen auf einem Schalter eine Beschriftung *und* ein Symbol erscheinen, verwendet man die Klasse **Button**, vereinbart die Beschriftung über das Attribut **android:text** und das Symbol über ein positionsspezifisches Attribut (z. B. **android:drawableLeft**):

```
<Button
    android:id="@+id/button4"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="24dp"
    android:drawableLeft="@drawable/ic_coins_1"
    android:drawablePadding="8dp"
    android:text="@string/account"
    android:contentDescription="@string/account"/>
```

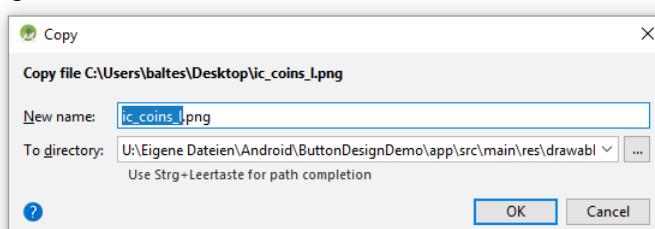
Für einen passenden Abstand zwischen Text und Bild sorgt das Attribut **android:drawable-Padding**.

Um eine PNG-Datei in ein AS-Projekt aufzunehmen, kann man unter Windows so vorgehen:

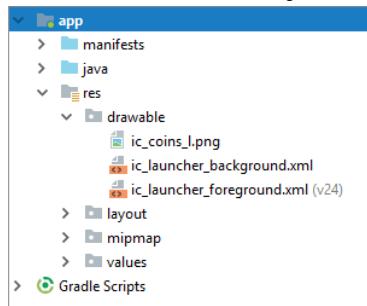
- Per Windows-Explorer die zu importierende PNG-Datei in die Zwischenablage befördern.
- Im Projektexplorer den folgenden Knoten aufklappen:  
**app/res/drawable**
- Aus dem Kontextmenü zum **drawable**-Knoten das Item **Paste** wählen, um die Importdatei aus der Zwischenablage zu entnehmen.
- Den auflösungsabhängigen Zielordner wählen:



- Das Kopieren bestätigen:



- Schließlich erscheint die aufgenommene Datei im Projektexplorer:



Eine Grundausstattung an Symbolen enthält das Android-SDK in versionsspezifischen Unterordnern, z. B. in:

**...\\sdk\\platforms\\android-27\\data\\res\\drawable-mdpi**

In Abschnitt 7.7.2 wurde beschrieben, wie eine Schalterdekoration durch zustandsabhängige Neun-Feld - Bitmaps realisiert wird. Auf den Google-Webseiten für Android-Entwickler wird beschrie-

ben, wie sich der Hintergrund eines Schalters über eine Ressource vom Typ *Color State List* (vgl. Abschnitt 7.1) gestalten lässt.<sup>1</sup>

#### 8.5.4 Umschalter

In diesem Abschnitt werden Klassen für Umschalter vorgestellt, die von der Basisklasse **CompoundButton** und indirekt von der Klasse **Button** abstammen (siehe Stammbaum in Abschnitt 8.2):

- Über ein **Kontrollkästchen** können die Benutzer eine Option ein- oder ausschalten. Es wird durch die Klasse **CheckBox** realisiert, z. B.:



Als alternative Darstellungsarten stehen zur Verfügung:

- Toggle-Schalter (Klasse **ToggleButton**)

Es ist eine Schaltfläche mit zwei alternierenden Beschriftungen vorhanden. Die aktuelle Beschriftung zeigt an, welcher Zustand durch das nächste Klickereignis erreicht wird, z. B.:

Vor API-Level 21



Ab API-Level 21



Eine horizontale Linie am unteren Rand signalisiert durch ihre Farbe den aktuellen Schaltzustand.

- Switch-Schalter (Klasse **Switch**)

Es ist ein Schieberegler mit den beiden Zuständen *Aus* (Position links) und *Ein* (Position rechts) vorhanden, z. B.:

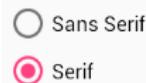
Vor API-Level 21



Ab API-Level 21



- Soll der Benutzer von mehreren Optionen *genau eine* wählen (Single Choice), verwendet man Steuerelemente aus der Klasse **RadioButton** in einem Container aus der Klasse **RadioGroup**, z. B.:

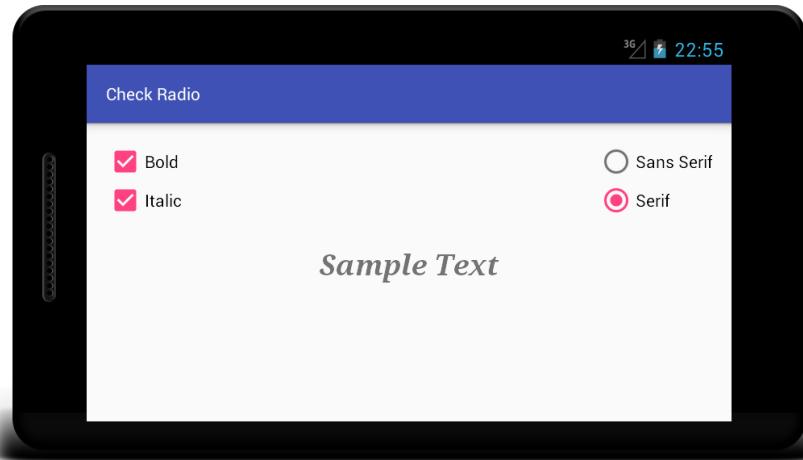


Ändert sich der Zustand eines Umschalters, erhalten registrierte Ereignisempfänger die Botschaft **onCheckedChanged()**. Dies ist die einzige Methode im Interface **OnCheckedChangeListener**, das in der Klasse **CompoundButton** definiert ist.

In folgendem Programm kann für den Text in einem **TextView**-Steuerelement über ein Optionsfeld zwischen den beiden Schriftarten **Sans Serif** und **Serif** gewählt werden. Außerdem lassen sich über Kontrollkästchen die Schriftauszeichnungen **fett** und **kursiv** unabhängig voneinander ein- bzw. ausschalten:

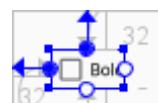
---

<sup>1</sup> Siehe: <http://developer.android.com/guide/topics/ui/controls/button.html>

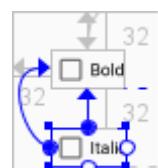


Das Layout verwendet ein Wurzelement aus der Klasse **ConstrainedLayout** mit folgenden Kindelementen:

- Das Kontrollkästchen für die Fett-Auszeichnung ist links und oben am Container mit einem Abstand von 32 dp angeheftet:



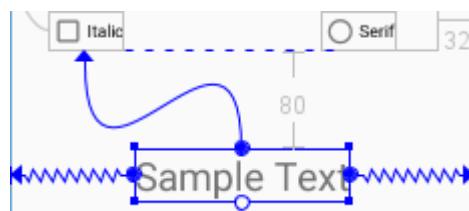
- Das Kontrollkästchen für die Kursiv-Auszeichnung übernimmt die horizontale Koordinate vom Kontrollkästchen für die Fett-Auszeichnung, ist also mit seinem Kollegen linksbündig ausgerichtet. Auch die vertikale Koordinate leitet die kursiv-**CheckBox** aus der Position der fett-**CheckBox** ab, indem es einen Abstand von 32dp zum unteren Rand der fett-**CheckBox** einhält:



- Die beiden Optionsschalter stecken in einer **RadioGroup**, die vom **LinearLayout** abstammt und zusätzliche Leistungen zur Verwaltung ihrer Kindelemente erbringt (siehe Abschnitt 8.5.4.3). Die **RadioGroup** ist am rechten Container-Rand mit dem Abstand 32 dp befestigt und übernimmt die vertikale Koordinate vom Kontrollkästchen, ist also oben mit diesem ausgerichtet:



- Das **TextView**-Element ist am linken und rechten Container-Rand befestigt, was wegen identischer Anziehungskräfte zu einer horizontalen Zentrierung führt. Seine vertikale Position erhält das **TextView**-Element, indem es zum unteren Rand der kursiv-**CheckBox** einen Abstand von 80 dp einhält:



Hier ist die vollständige Layoutdefinition zu sehen:

```
<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <CheckBox
        android:id="@+id/cbBold"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginLeft="32dp"
        android:layout_marginStart="32dp"
        android:layout_marginTop="32dp"
        android:text="@string/bold"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <CheckBox
        android:id="@+id/cbItalic"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="32dp"
        android:text="@string/italic"
        app:layout_constraintStart_toStartOf="@+id/cbBold"
        app:layout_constraintTop_toBottomOf="@+id/cbBold" />

    <RadioGroup
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginRight="32dp"
        android:checkedButton="@+id/rbSans"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="@+id/cbBold" >

        <RadioButton
            android:id="@+id/rbSans"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/sans" />

        <RadioButton
            android:id="@+id/rbSerif"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginTop="32dp"
            android:text="@string/serif" />
    </RadioGroup>

    <TextView
        android:id="@+id/tvSample"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="80dp"
        android:text="@string/sample"
        android:textSize="32sp"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/cbItalic" />
</android.support.constraint.ConstraintLayout>
```

Das AS-Projekt ist im folgenden Ordner zu finden:

...\\BspUeb\\UI\\Controls\\CheckRadio

### 8.5.4.1 Kontrollkästchen

In der Aktivitätsklasse des Beispielprogramms werden Instanzvariablen für die **CheckBox** - Objekte deklariert und in der Methode **onCreate()** initialisiert:

```
private CheckBox cbBold, cbItalic;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_check_radio);
    cbBold = findViewById(R.id.cbBold);
    cbItalic = findViewById(R.id.cbItalic);
    . . .
}
```

Zur Behandlung der Zustandswechselereignisse wird im Beispielprogramm für die beiden Kontrollkästchen jeweils eine anonyme Klasse definiert. Den Quellcode der Klasse zu **cbBold** kennen Sie schon aus Abschnitt 8.4.2.1. Es folgt das analoge Gegenstück zu **cbItalic**:

```
cbItalic.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener() {
    @Override
    public void onCheckedChanged(CompoundButton cb, boolean isChecked) {
        Typeface typeface = tvSample.getTypeface();
        int style = typeface.getStyle();
        Typeface actTFace = rbSans.isChecked() ? Typeface.SANS_SERIF : Typeface.SERIF;
        if (isChecked)
            tvSample.setTypeface(actTFace,
                (style & Typeface.BOLD) != 0 ? Typeface.BOLD_ITALIC : Typeface.ITALIC);
        else
            tvSample.setTypeface(actTFace,
                (style & Typeface.BOLD) != 0 ? Typeface.BOLD : Typeface.NORMAL);
    }
});
```

Die Methode **onCheckedChanged()** erfragt beim **TextView**-Objekt **tvSample** mit **getTypeface()** die Schriftart. Das zurück gelieferte **Typeface**-Objekt repräsentiert die Schriftart und lässt sich über **getStyle()** nach der Schriftauszeichnung befragen, wobei der **int**-wertigen Rückgabe folgende Kodierung zugrunde liegt:

int-Wert	Stil	Typeface-Konstante
0	Standard	NORMAL
1	Fett	BOLD
2	Kursiv	ITALIC
3	Fett + Kursiv	BOLD_ITALIC

Mit der Methode **setTypeface()** kann man einer **TextView**-Komponente eine Schriftartfamilie und eine Schriftauszeichnung verordnen (vgl. Abschnitt 8.5.1). Allerdings ist es *nicht* möglich, per **setTypeface()** mit dem vorhandenen **Typeface**-Objekt als erstem Parameter die Auszeichnung wieder auf NORMAL zu setzen. Daher wird im Beispiel der erste **setTypeface()** - Parameter über eine Konstante der Klasse **TypeFace** geliefert und die passende Schriftart über den Schaltzustand des **RadioButton**-Objekts **rbSans** (vgl. Abschnitt 8.5.4.3) ermittelt.<sup>1</sup>

Je nach Markierung des Kontrollkästchens (Wert des Parameters **isChecked**) wird mit der **TextView**-Methode **setTypeface()** die *kursiv*-Auszeichnung für den Beispieltext in **tvSample** gesetzt oder aufgehoben. Dabei ist die **fett**-Auszeichnung zu berücksichtigen. Um deren Wert zu er-

<sup>1</sup> Viele Webseiten empfehlen, der ersten **setTypeFace()** - Parameter auf **null** zu setzen, wenn nur die Auszeichnung geändert werden soll. Wie ein Blick in den Quellcode der Klasse **TextView** zeigt, ist diese Technik nicht sinnvoll - mit dem Wert 0 (Auszeichnung NORMAL) als zweitem Parameter. Die **TextView**-Methode **getTypeface()** liefert anschließend die Rückgabe **null**, was im Beispielprogramm zu einer **NullPointerException** führt.

mitteln, wird im folgenden Ausdruck der Bit-orientierte Operator **&** verwendet (siehe Baltes-Götz & Götz 2018, Abschnitt 3.5.6):

```
(style & Typeface.BOLD) != 0
```

Der Operator **&** erzeugt aus den beiden **int**-wertigen Argumenten ein Bitmuster, das genau dann an der Stelle *k* eine 1 enthält, wenn *beide* Argumentmuster an dieser Stelle eine 1 besitzen und andernfalls eine 0. In **Typeface.BOLD** ist nur ein einziges Bit ungleich 0, und insgesamt ist der obige Ausdruck genau dann **true**, wenn in der aktuellen Schrift die **fett**-Auszeichnung eingeschaltet ist. Die restliche Logik des Konditionaloperators dürfte klar sein.

#### 8.5.4.2 Toggle-Button und Switch

Um Darstellungsalternativen zur Klasse **CheckBox** zu demonstrieren, ersetzen wir im Umschalter-Beispielprogramm das erste Kontrollkästchen durch ein Objekt der Klasse **ToggleButton** und das zweite durch ein Objekt aus der Klasse **Switch**. Im Java-Quellcode der Aktivitätsklasse sind im Wesentlichen die Klassenzugehörigkeiten zu ändern:

```
private ToggleButton toggleBold;
private Switch switchItalic;
. .
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_check_radio);
    toggleBold = findViewById(R.id.toggleBold);
    switchItalic = findViewById(R.id.switchItalic);
. .
}
```

In der Layout-Definition sind neben den Elementnamen die Beschriftungsattribute zu ändern. Beim **ToggleButton** wird das Attribut **android:text** ersetzt durch die Attribute **android:textOff** und **android:textOn** mit den Beschriftungen für den passiven bzw. für den aktiven Zustand:

```
<ToggleButton
    android:id="@+id/toggleBold"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginLeft="32dp"
    android:layout_marginStart="32dp"
    android:layout_marginTop="32dp"
    android:textOff="@string/textBoldOff"
    android:textOn="@string/textBoldOn"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

In der Beschriftung für den passiven Zustand wird mitgeteilt, dass beim nächsten Klick ein Wechsel in den aktiven Zustand erfolgt:

```
<string name="textBoldOff">Bold On</string>
```

Beim **Switch**-Objekt informiert das Attribut **android:text** (wie bei einem **CheckBox**-Objekt) über den Zweck des Umschalters:



Ältere Android-Versionen (vor API-Level 21) besitzen Beschriftungen für den aktuellen Schaltzustand:



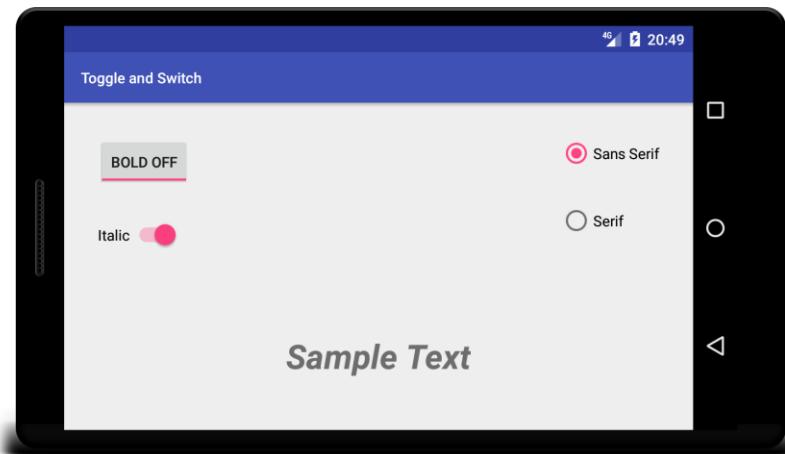
Weil wir alle Android-Versionen ab API-Level 15 unterstützen wollen, versorgen wir in der Layout-Deklaration auch die zugehörigen Attribute **android:textOff** und **android:textOn** mit Werten:

```
<Switch
    android:id="@+id/switchItalic"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="32dp"
    android:text="@string/italic"
    android:textOff="@string/textItalicOff"
    android:textOn="@string/textItalicOn"
    app:layout_constraintStart_toStartOf="@+id/toggleBold"
    app:layout_constraintTop_toBottomOf="@+id/toggleBold" />
```

Bei den Beschriftungen für den passiven bzw. aktiven Zustand muss man seine Kreativität nicht strapazieren:

```
<string name="textItalicOff">Off</string>
<string name="textItalicOn">On</string>
```

Die Bedienbarkeit des Umschalter-Beispielprogramms profitiert nicht unbedingt von der Ersetzung der Kontrollkästchen durch ein **ToggleButton**- bzw. **Switch**-Objekt:



#### 8.5.4.3 Optionsschalter

Bei zusammengehörigen **RadioButton**-Objekten muss dafür gesorgt werden, dass maximal (oder genau) ein Schalter markiert (eingerastet) ist. Dazu steckt man die Umschalter in einen Container vom Typ **RadioGroup**, der von **LinearLayout** abstammt, z. B.:

```
<RadioGroup
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginRight="32dp"
    android:checkedButton="@+id/rbSans"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toTopOf="@+id/cbBold" >

    <RadioButton
        android:id="@+id/rbSans"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/sans" />
```

```

<RadioButton
    android:id="@+id/rbSerif"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="32dp"
    android:text="@string/serif" />
</RadioGroup>

```

Im Beispiel wird dafür gesorgt, dass beim Programmstart der Schalter zur Schriftart **Sans Serif** eingerastet ist:

```
    android:checkedButton="@+id/rbSans"
```

Für die beiden Optionsschalter des Umschalter-Demonstrationsprogramms

```

private RadioButton rbSans, rbSerif;
. . .
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_check_radio);
    rbSans = findViewById(R.id.rbSans);
    rbSerif = findViewById(R.id.rbSerif);
    rbSans.setOnCheckedChangeListener(this);
    rbSerif.setOnCheckedChangeListener(this);
. . .
}

```

übernimmt das Aktivitätsobjekt die Ereignisbehandlung, wozu seine Klasse mit der Methode **onCheckedChanged()** ausgerüstet wird:

```

public class CheckRadioActivity extends AppCompatActivity
    implements CompoundButton.OnCheckedChangeListener {
. . .
@Override
public void onCheckedChanged(CompoundButton rb, boolean isChecked) {
    if (!isChecked)
        return;
    int style = tvSample.getTypeface().getStyle();
    if (rb == rbSans)
        tvSample.setTypeface(Typeface.SANS_SERIF, style);
    else
        tvSample.setTypeface(Typeface.SERIF, style);
}
. . .
}

```

In dieser Methode kommen die in Abschnitt 8.5.4.1 vorgestellten Schriftverwaltungsmethoden **getTypeface()**, **getStyle()** und **setTypeface()** zum Einsatz.

Bei jedem Markierungswechsel wird die Ereignisbehandlungsmethode *zweimal* aufgerufen:

- Für das **RadioButton**-Objekt, das die Markierung erhalten hat
- Für das **RadioButton**-Objekt, das die Markierung verloren hat

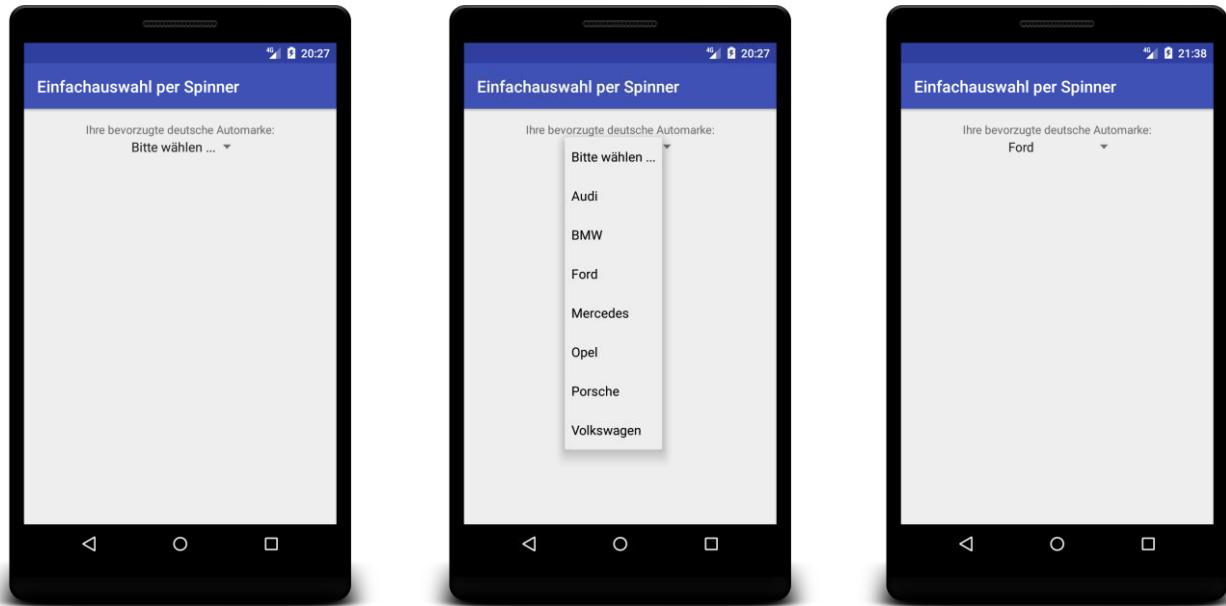
Weil die Methode nur im ersten Fall arbeiten soll, wird zu Beginn mit **isChecked()** der Wechseltyp festgestellt und die Ereignisbehandlung ggf. abgebrochen.

### 8.5.5 Einfachauswahl per Spinner-Element

Bei einer Einfachauswahl (z. B. aus einer Liste von Automarken) ist oft ein Steuerelement vom Typ **Spinner**, das eine Dropdown-Liste realisiert, eine günstige Lösung im Vergleich zu den potentiellen Alternativen **RadioGroup** und **TextEdit**:

- Wie beim Texteingabefeld wird wenig Platz benötigt.
- Wie bei der **RadioButton**-Serie sind ungültige Daten ausgeschlossen, sodass wenig Validierungsaufwand entsteht.

Anschließend ist ein **Spinner**-Element im initialen Zustand (links), im geöffneten Zustand (Mitte) und im geschlossenen Zustand nach einer vorherigen Wahl (rechts) zu sehen:



Als erste (im Startzustand sichtbare) Option sollte eventuell (wie im Beispiel) eine Aufforderung zur Wahl präsentiert werden, damit nicht bei einem unbeachteten **Spinner**-Element die oben stehende echte Option als gewählt betrachtet wird.

Ein **Spinner**-Steuerelement kann per XML deklariert und dabei durch das Attribut **android:entries**, das auf eine Ressource von Typ **String**-Array zeigt, über die Wahlalternativen informiert werden:

```
<Spinner
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/spinner"
    android:entries="@array/german_cars" />
```

Ein **String**-Array ist eine *einfache* Ressource (vgl. Abschnitt 7.1), die zusammen mit anderen einfachen Ressourcen in einer XML-Datei im Zweig **res/values** innerhalb eines gemeinsamen **<resources>** - Wurzelelements definiert wird. Im Beispiel wird eine **String**-Array - Ressource namens **german\_cars** in der Datei **strings.xml** definiert:

```

<resources>
    <string name="app_name">Single Choice By Spinner</string>
    <string name="prompt">Your preferred german car company:</string>
    <string name="selection">Your selection:</string>
    <string-array name="german_cars">
        <item>Please select ...</item> <item>Audi</item>
        <item>BMW</item> <item>Ford</item>
        <item>Mercedes</item> <item>Opel</item>
        <item>Porsche</item> <item>Volkswagen</item>
    </string-array>
</resources>

```

Wenn die **String**-Array - Ressource zu einem **Spinner**-Steuerelement erst zur Laufzeit festgelegt werden soll, verzichtet man auf das XML-Attribut **android:entries** und erstellt ein Objekt der konkretisierten generischen Klasse **ArrayAdapter<CharSequence>** über die statische **ArrayAdapter**-Methode **createFromResource()**, z. B.:<sup>1</sup>

```
ArrayAdapter<CharSequence> adapter = ArrayAdapter.createFromResource(this,
    R.array.german_cars, android.R.layout.simple_spinner_dropdown_item);
```

Im dritten Parameter wird das Layout für die dynamisch zu erzeugenden Elemente des Drop-Down - Menüs festgelegt. Anschließend ist der Effekt von zwei Aktualparameterwerten (jeweils als Konstante der Klasse **android.R.layout** ausgedrückt) zu sehen:

**android.R.layout.simple\_spinner\_item**      **android.R.layout.simple\_spinner\_dropdown\_item**



Das **ArrayAdapter<CharSequence>** - Objekt wird schließlich über die Methode **setAdapter()** dem **Spinner**-Steuerelement zugeordnet, das zuvor mit einer Referenzvariablen verbunden wurde:

```
Spinner spinner = findViewById(R.id.spinner);
spinner.setAdapter(adapter);
```

Wie man einen **ArrayAdapter<String>** auf das Basis eines **String**-Arrays erstellt, war in Abschnitt 8.5.2.5 zu sehen.

Um die Benutzerwahl verarbeiten zu können, vereinbart man für das **Spinner**-Steuerelement über die Methode **setOnItemSelectedListener()** ein Objekt vom Interface Typ **AdapterView.OnItemSelectedListener** zur Verarbeitung der Wahl:

---

<sup>1</sup> **CharSequence** ist ein Interface, das u.a. von der Klasse **String** implementiert wird. In Java verwendet man **CharSequence**, um eine Festlegung auf eine spezielle Klasse (wie z. B. **String**) zu vermeiden. Im aktuellen Beispiel ist die Verwendung von **CharSequence** bequem, weil es der Rückgabetyp der Methode **createFromResource()** ist.

```

public class MainActivity extends AppCompatActivity
    implements AdapterView.OnItemSelectedListener {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Spinner spinner = findViewById(R.id.spinner);
        spinner.setOnItemSelectedListener(this);
    }

    @Override
    public void onItemSelected(AdapterView<?> parent, View view, int position, long id) {
        if (position != 0) {
            String text = ((TextView) view).getText().toString();
            AlertDialog.Builder builder = new AlertDialog.Builder(this);
            builder.setMessage(getString(R.string.selection)+"\n" + text +
                    " (Pos. " + Integer.toString(position) + ")");
            AlertDialog dialog = builder.create();
            dialog.show();
        }
    }

    @Override
    public void onNothingSelected(AdapterView<?> parent) {
    }
}

```

Das Interface **AdapterView.OnItemSelectedListener** schreibt zwei Methoden vor:

- **public void onItemSelected(AdapterView<?> parent, View view, int position, long id)**
- **public void onNothingSelected(AdapterView<?> parent)**

Im Beispiel beschränken wir uns darauf, in der Methode **onItemSelected()** die Benutzerwahl per Message-Dialog zu bestätigen.

Leider wird die Methode **onItemSelected()** auch unmittelbar nach der Kreation des Steuerelements aufgerufen, was im Beispiel aufgrund der Bedingung (**position != 0**) nicht auffällt. Um den meist unerwünschten ersten Aufruf unauffällig zu gestalten, kann man so vorgehen:

- In einer Instanzvariablen wird der Initialisierungszustand des **Spinner**-Elements notiert:  
`private boolean spinnerInit = false;`
- Stellt die Methode **onItemSelected()** fest, dass die Variable **spinnerInit** noch den Wert **false** hat, beschränkt sie sich darauf, diesen Wert zu ändern:  
`if (!spinnerInit) {
 spinnerInit = true;
 return;
}`

Das AS-Projekt zum aktuellen Beispiel ist im folgenden Ordner zu finden:

`...\\BspUeb\\UI\\Controls\\Spinner`

### 8.5.6 Fortschrittsanzeige

Um dem Benutzer über eine laufende Bearbeitung und nach Möglichkeit auch über den bereits erzielten Bearbeitungsfortschritt zu informieren, setzt man ein Steuerelement aus der Klasse **ProgressBar** ein:

```
<ProgressBar
    android:id="@+id/progressBar"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
```

Für die Zugriffsmöglichkeit per Programm sorgt man auf die folgende Art:

```
private ProgressBar progressBar;
. . .
protected void onCreate(Bundle savedInstanceState) {
    . . .
    progressBar = findViewById(R.id.progressBar);
}
```

Per Voreinstellung erscheint als Fortschrittsanzeige ein rotierender Kreis, der keine Information über die zu erwartende Bearbeitungsdauer bietet:



Eine informative horizontale Fortschrittsanzeige



lässt sich über das XML-Attribut **style** erreichen:<sup>1</sup>

```
style="@android:style/Widget.ProgressBar.Horizontal"
```

Aus dem Wert **wrap\_content** für das XML-Attribut **android:layout\_width** ergibt sich ein recht schmaler Fortschrittsbalken:



Daher empfiehlt sich im Kontext mit einem **LinearLayout** der Wert **match\_parent** bzw. im Kontext mit einem **ConstraintLayout** der Wert **match\_constraint**.

Selbstverständlich existieren nahezu unbegrenzt viele Varianten zur Gestaltung des Fortschrittsbalkens. Anschließend ist eine durch den Wert **4dp** für das Attribut **android:layout\_height**

```
android:layout_height="4dp"
```

realisierte schmale Variante zu sehen:



Bei Verwendung des horizontalen Balkenstils kann der Benutzer mit Hilfe der Methode **setProgress()** über den Bearbeitungsfortschritt informiert werden, z. B.:

```
progressBar.setProgress(progress);
```

Per Voreinstellung werden der Bearbeitungsbeginn mit dem Wert 0 und die vollständige Bearbeitung mit dem Wert 100 signalisiert. Ein alternativer Maximalwert lässt sich über das XML-Attribut **android:max** oder mit der wobei mit der **ProgressBar**-Methode **setMax()** vereinbaren.

Soll eine Fortschrittsanzeige nur temporär zu sehen sein, kann sie mit der **ProgressBar**-Methode **setVisibility()** aus- bzw. eingeschaltet werden, z. B.:

---

<sup>1</sup> Siehe Abschnitt 7.3.2 zu den Punkten im Ressourcennamen

```
progressBar.setVisibility(ProgressBar.GONE);
progressBar.setVisibility(ProgressBar.VISIBLE);
```

Den initialen Zustand eines **ProgressBar**-Steuerelements kann man auch über das XML-Attribut **android:visibility** beeinflussen, z. B.:

```
android:visibility="gone"
```

Eine halbwegs sinnvolle Demonstration des Fortschrittsbalkens erfordert eine Anwendung mit Hintergrundaufgabe und damit einen Vorgriff auf das nicht mehr allzu ferne Kapitel 10 über Multithreading. Wir definieren die folgende Klasse **ProgressDemoTask** als Ableitung der Klasse **AsyncTask<String, Integer, Integer>**:

```
private class ProgressDemoTask extends AsyncTask<String, Integer, String> {

    @Override
    protected void onPreExecute() {
        button.setEnabled(false);
        progressBar.setProgress(0);
        progressBar.setVisibility(ProgressBar.VISIBLE);
    }

    @Override
    protected String doInBackground(String... strings) {
        int i = 0;
        while (i < 100) {
            if (this.isCancelled())
                return null;
            i++;
            try {
                Thread.sleep(100);
            } catch(InterruptedException ie) {Thread.currentThread().interrupt();}
            publishProgress(i*1);
        }
        return null;
    }

    @Override
    protected void onProgressUpdate(Integer... progress) {
        progressBar.setProgress(progress[0]);
    }

    @Override
    protected void onPostExecute(String res) {
        progressBar.setVisibility(ProgressBar.GONE);
        button.setEnabled(true);
    }
}
```

Die Logik von Android sorgt dafür, dass beim Starten der Hintergrundaufgabe zunächst die **ProgressDemoTask**-Methode **onPreExecute()** ausgeführt wird. Dabei wird der Befehlsschalter deaktiviert. Diese Technik ist offenbar in vielen Situationen nützlich, also ein guter Kandidat für das Langzeitgedächtnis eines Android-Entwicklers:

```
button.setEnabled(false);
```

Außerdem wird in **onPreExecute()** der Fortschrittsbalken auf den Wert 0 gesetzt und sichtbar gemacht:

```
progressBar.setProgress(0);
progressBar.setVisibility(ProgressBar.VISIBLE);
```

Anschließend beginnt die `asynchron` in einem Hintergrund-Thread ausgeführte Methode `doInBackground()` damit, in 100 Schritten jeweils 100 Millisekunden zu schlafen und dann den Fortschrittsbalken um den Wert 1 zu steigern. Letzteres geschieht durch einen Aufruf der Methode `publishProgress()`, die wiederum für einen Aufruf der Methode `onProgressUpdate()` sorgt, die im UI-Thread ausgeführt wird und daher auf den Fortschrittsbalken zugreifen darf.

Nach Erledigung seiner Aufgabe führt das `ProgressDemoTask`-Objekt noch die Methode `onPostExecute()` mit Abschlussarbeiten aus:

- Der Fortschrittsbalken wird zum Verschwinden gebracht,  
`progressBar.setVisibility(ProgressBar.GONE);`
- und der Befehlsschalter wird wieder aktiviert:  
`button.setEnabled(true);`

`ProgressDemoTask` wird als Mitgliedsklasse der folgenden Aktivitätsklasse `MainActivity` definiert:

```
package de.zimkand.progressbardemo;

import android.os.AsyncTask;
. . .
import android.widget.ProgressBar;

public class MainActivity extends AppCompatActivity implements View.OnClickListener {

    private ProgressBar progressBar;
    private Button button;
    private ProgressDemoTask task;

    private class ProgressDemoTask extends AsyncTask<String, Integer, String> {
        . . .

        @Override
        protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_main);
            progressBar = findViewById(R.id.progressBar);
            button = findViewById(R.id.button);
            button.setOnClickListener(this);
        }

        @Override
        public void onClick(View v) {
            task = new ProgressDemoTask();
            task.execute();
        }

        @Override
        protected void onDestroy() {
            super.onDestroy();
            if (task != null)
                task.cancel(true);
        }
    }
}
```

Wenn die Aktivität (z. B. bei einem Orientierungswechsel) von Android zerstört wird (also den `onDestroy()` - Aufruf erhält), muss die `AsyncTask` per `cancel()` - Aufruf aufgefordert werden, den Betrieb einzustellen:

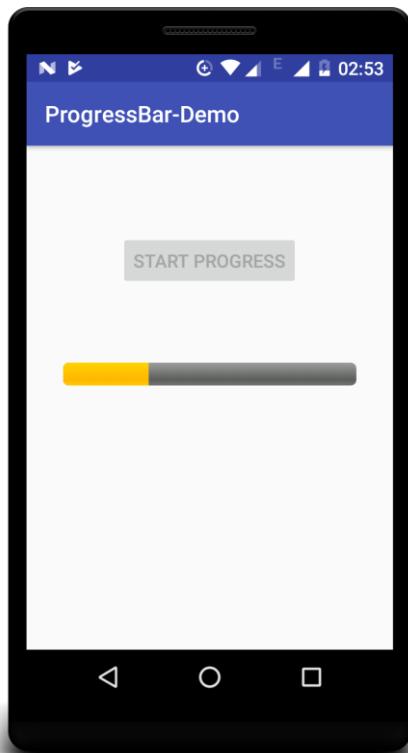
```
if (task != null)
    task.cancel(true);
```

Im Beispiel ist dafür gesorgt, dass sie es tatsächlich tut (siehe Quellcode der Methode `doInBackground()`):

```
if (this.isCancelled())
    return null;
```

Ohne diese Aufräumaktion behält das `ProgressDemoTask`-Objekt eine implizite Referenz auf das bei seiner Erstellung umgebende Aktivitätsobjekt. Folglich kann das mittlerweile zerstörte Aktivitätsobjekt nicht vom Garbage Collector aus dem Speicher entfernt werden, und es kommt es einer Speicherverschwendungen. Zu diesen Details der Multithreading-Programmierung mit Hilfe der Klasse `AsyncTask` siehe Abschnitt 10.4.

So sieht die Bedienoberfläche aus:



Nach einem Klick auf den Schalter wird ein Objekt der Klasse `ProgressDemoTask` erzeugt und über die Methode `execute()` beauftragt, in einem separaten Thread eine Aufgabe auszuführen.

Das AS-Projekt zum aktuellen Beispiel ist im folgenden Ordner zu finden:

...\\BspUeb\\UI\\Controls\\ProgressBar

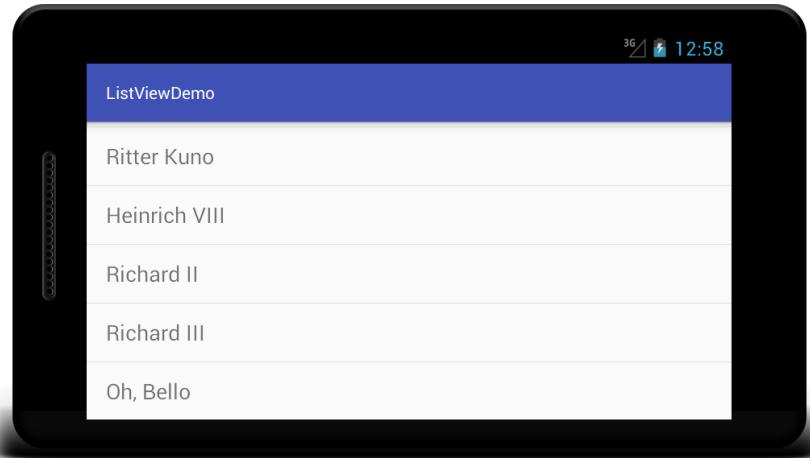
Wie man z. B. an der Reaktion der Anwendung auf einen Orientierungswechsel bemerkt, sind noch nicht alle Anforderungen an eine sinnvolle Multithreading-Lösung erfüllt. Später folgen realistischere Beispiele zur Verwendung eines Fortschrittsbalkens.

### 8.5.7 ListView und ListActivity

Eine sehr verbreitete Aufgabe für eine Android-Activity besteht darin, eine zu durchstöbernde Liste von vertikal angeordneten Items (Kontakten, Musiktiteln etc.) zu präsentieren. Nach Wahl eines Items durch den Benutzer erscheint in der Regel eine andere Activity zur Detailpräsentation bzw. -bearbeitung.

Zur Lösung dieser Routineaufgabe ist mit der Klasse `ListView` ein Container verfügbar, der (ähnlich wie die Klassen `RadioGroup` und `Spinner`) für die Verwaltung spezieller Kindelemente kon-

zipiert ist und entsprechende Zusatzkompetenzen besitzt. Die Listenelemente werden durch ein Objekt vom Typ **Adapter** geliefert, das sie z. B. aus einer Datenbank (vgl. Kapitel 13) oder aus einem Array beschafft. Für das folgende Beispiel



sorgt eine simple Layoutdefinition

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <ListView
        android:id="@+id/list"
        android:layout_height="wrap_content"
        android:layout_width="match_parent">
    </ListView>
</FrameLayout>
```

und der folgende Aktivitäts-Quellcode:

```
package de.zimkand.listview;

import android.support.v7.app.AlertDialog;
. . .

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        ListView lv = findViewById(R.id.list);

        String[] titles = {"Heinrich IV", "Alfons, der Vierter vor Zwölften",
            "Heinrich V", "Ritter Kuno", "Heinrich VIII", "Richard II", "Richard III",
            "Oh, Bello", "Richard IV", "Richard V", "Der Kaufmann aus Venedig",
            "Othello", "König Lear"};

        lv.setAdapter(new ArrayAdapter<String>(this,
            android.R.layout.simple_list_item_activated_1, titles));
        lv.setVerticalScrollBarEnabled(true);
        lv.setOnItemClickListener(new ListView.OnItemClickListener() {
            @Override
            public void onItemClick(AdapterView<?> av, View v, int position, long id) {
                AlertDialog.Builder builder = new AlertDialog.Builder(MainActivity.this);
                builder.setMessage("Ihre Wahl: " + ((TextView)v).getText().toString());
                AlertDialog dialog = builder.create();
                dialog.show();
            }
        });
    }
}
```

Auf gewohnte Weise wird eine Referenz zu dem **ListView**-Objekt hergestellt, das aus der Layoutdefinition resultiert:

```
ListView lv = findViewById(R.id.list);
```

Der Adapter vom Typ **ArrayAdapter<String>** entsteht aus einem **String**-Array und einer Layoutdefinition in der Klasse **R.layout** im Paket **android**:

```
lv.setAdapter(new ArrayAdapter<String>(this,
    android.R.layout.simple_list_item_activated_1, titles));
```

Nachdem das **ListView**-Objekt einen Adapter und damit seine Items erhalten hat, wird noch ein vertikaler Rollbalken und eine Klickbehandlungsmethode ergänzt, die sich auf die Anzeige einer Meldung beschränkt, da wir uns mit **Intent**-Technik zum Starten weiterer Aktivitäten erst im folgenden Kapitel 9 beschäftigen werden.

Das AS-Projekt zum aktuellen Beispiel ist im folgenden Ordner zu finden:

...\\BspUeb\\UI\\Controls\\ListView

Das Android-SDK enthält mit der **ListActivity** eine Klasse, die ein **ListView**-Objekt als einziges Steuerelement enthält, sodass man keine Layoutdefinition benötigt. Damit reduziert sich der Aufwand für das obige Beispiel auf die folgende Java-Quellcodedatei:

```
package de.zimkand.listactivity;

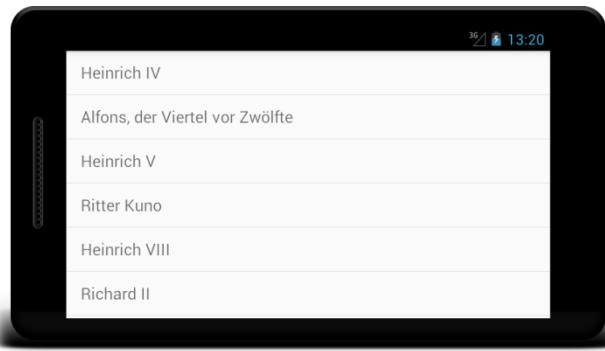
import android.app.ListActivity;
. . .

public class MainActivity extends ListActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        String[] titles = {"Heinrich IV", "Alfons, der Viertel vor Zwölften",
            "Heinrich V", "Ritter Kuno", "Heinrich VIII", "Richard II", "Richard III",
            "Oh, Bello", "Richard IV", "Richard V", "Der Kaufmann aus Venedig",
            "Othello", "König Lear"};

        setListAdapter(new ArrayAdapter<String>(this,
            android.R.layout.simple_list_item_activated_1, titles));
        ListView lv = getListView();
        lv.setVerticalScrollBarEnabled(true);
        lv.setOnItemClickListener(new ListView.OnItemClickListener() {
            @Override
            public void onItemClick(AdapterView<?> av, View v, int position, long id) {
                AlertDialog.Builder builder = new AlertDialog.Builder(MainActivity.this);
                builder.setMessage("Ihre Wahl: " + ((TextView)v).getText().toString());
                AlertDialog dialog = builder.create();
                dialog.show();
            }
        });
    }
}
```

Es fällt allerdings leicht negativ auf, dass dem **ListActivity**-Objekt die Titelzeile (**ActionBar**) fehlt:



Wer sie vermisst, sollte also besser die Klasse **AppCompatActivity** beerben und das simple Layout mit **ListView**-Element konstruieren (siehe oben).

Wenn bei einer **ListView**-Anwendung die Performanz zum Problem wird, empfiehlt Google einen Wechsel zur Klasse **RecyclerView**, die durch geschickte Wiederverwendung von sogenannten *view holder* - Objekten die Anzahl von zeitaufwändigen Objektcreationen reduziert und damit auch für sehr lange Listen geeignet ist. Der erhöhte Aufwand im Vergleich zur Klasse **ListView** wird noch durch weitere Vorteile belohnt.<sup>1</sup>

## 8.6 Menüs

Als Grundausstattung mit Menüformen bietet Android seit der Version 3:<sup>2</sup>

- **Optionsmenü**

In das Optionsmenü einer Aktivität gehören Angebote, welche die Aktivität bzw. Anwendung insgesamt betreffen (z. B. Einstellungen ändern, Hilfe öffnen, Suche starten). Mit der Android-Version 3 wurde der dedizierte Menü-Schalter aus dem Pflichtenheft für Android-Geräte gestrichen. Seine Aufgabe wurde durch die zunächst als *Action Bar* und später als *App Bar* bezeichneten Leiste übernommen, die sich meist am oberen Rand des Displays befindet. Hier zeigen sich neben dem Titel der Aktivität besonders wichtige Items aus dem Optionsmenü, während die restlichen Menüitems durch das Überlaufsymbol ☰ am rechten Rand der App Bar zugänglich sind. Trotz aller Umbaumaßnahmen an den Android-Bedienelementen blieb die Grundlogik des Optionsmenüs unverändert.

- **Kontextmenü**

Das Kontextmenü erscheint nach einem Langklick auf ein Element der Bedienoberfläche und enthält Aktionen zur Modifikation des Elements (z. B. Löschen, Editieren). Bei der Realisation konkurrieren das traditionelle, *schwebende Kontextmenü* und die neuerdings von Google empfohlenen *kontextabhängigen Aktionen*.

- **Popup-Menü**

Das Popup-Menü ähnelt optisch dem traditionellen, schwebenden Kontextmenü. Es wird zwar durch ein Steuerelement geöffnet (z. B. per Klick oder Tap), dient aber nicht (wie das Kontextmenü) zur Bearbeitung des initiierenden Elements, sondern anderen Aufgaben. So kann z. B. nach Betätigung eines Schalters, der das Teilen eines Inhalts anfordert, ein Popup-Menü erscheinen, um das zu verwendende soziale Netzwerk zu erfragen.

Als weitere Navigationsoption bieten einige Apps (z. B. die Einstellungs-App von Android) ein per Wischgeste vom linken Bildschirmrand nach innen oder per Hamburger-Symbol am linken Rand der App Bar zu öffnendes Schubladenmenü (engl. *navigation drawer*). Es wird von Google in den Richtlinien für das Material Design empfohlen für die seitliche (laterale) Navigation auf derselben

---

<sup>1</sup> <https://developer.android.com/guide/topics/ui/layout/recyclerview#java>

<sup>2</sup> <https://developer.android.com/guide/topics/ui/menus>

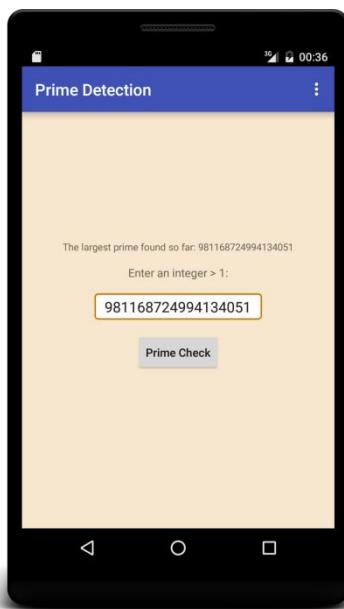
Hierarchieebene.<sup>1</sup> In der Praxis werden Options- und Schubladenmenü oft für ähnliche Zwecke verwendet (als Hauptmenü). Leider ist die technische Realisation des Schubladenmenüs relativ aufwändig, sodass es in unserem Einführungskurs nicht behandelt werden kann.<sup>2</sup>

## 8.6.1 Optionsmenü

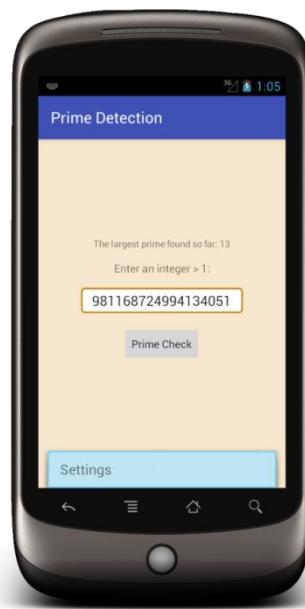
### 8.6.1.1 Hardware-Menütaste versus App Bar

Wie die anschließend präsentierten Bildschirmfotos einer App zur Primzahlendiagnose zeigen, erscheint zum Optionsmenü auf einem modernen Android-Gerät *ohne* Hardware-Menütaste am rechten Rand der App Bar das Überlaufsymbol ☰. Auf einem älteren Gerät *mit* Hardware-Menütaste (z. B. ☰) zeigt die Titelzeile *keinen* Hinweis auf das Optionsmenü. Nach einem Klick auf die Menütaste erscheint das Menü am unteren Display-Rand, z. B.:

Optionsmenü per App Bar



Optionsmenü per Hardware-Taste



### 8.6.1.2 Menüdeklaration per XML-Datei

Analog zum Vorgehen bei einer Layout-Definition sollte man ein Menü nicht per Quellcode aufbauen, sondern (im Projektexplorerknoten **app/res/menu**) eine XML-formatierte Menü-Ressource erstellen, die dann im Programm zu einem Objekt der Klasse **Menu** inflationiert wird, z. B.:

<sup>1</sup> <https://material.io/design/navigation/understanding-navigation.html#lateral-navigation>

<sup>2</sup> Hier befindet sich die Google-Anleitung zur Realisation des Schubladenmenüs:

<https://developer.android.com/training/implementing-navigation/nav-drawer#java>

```

<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context="de.zimkand.menudemo.MainActivity">
    <item
        android:id="@+id/settings"
        android:icon="@drawable/ic_settings"
        android:title="Settings"
        app:showAsAction="ifRoom" />
    <item
        android:id="@+id/layout"
        android:title="Layout" />
    <group
        android:id="@+id/group"
        android:title="Group">
        <item
            android:id="@+id/info"
            android:title="Info" />
        <item
            android:id="@+id/help"
            android:title="Help" />
    </group>
</menu>

```

#### 8.6.1.2.1 Elemente einer Menüdeklaration

Aus dem **menu**-Wurzelement der XML-Datei entsteht im laufenden Programm ein Container aus der Klasse **Menu**, der Elemente aus der Klasse **MenuItem** aufnimmt. Das Wurzelement einer XML-Menüdefinition kann als Kindelemente enthalten:

- **item**-Elemente, die jeweils ein Menüitem repräsentieren
- **group**-Elemente, welche Menüitems zu Gruppen mit gemeinsamen Eigenschaften (z. B. Sichtbarkeit, (De)aktivierung) zusammenfassen

Die wesentlichen Attribute eines **item**-Elements sind:

- **android:id**  
Anhand dieser Kennung kann ein Menüitem im Quellcode identifiziert werden.
- **android:icon**
- **android:title**
- **android:orderInCategory**  
Über dieses Attribut kann man die Reihenfolge bestimmen, mit der die Items im Menü erscheinen.

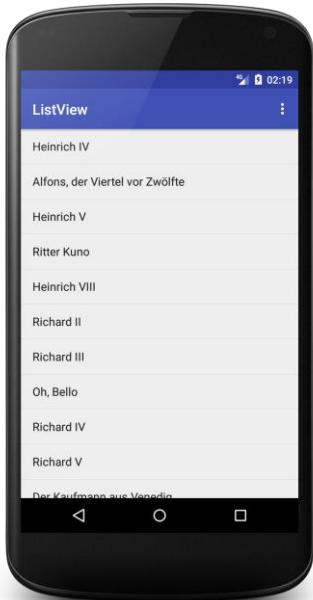
- **android:showAsAction**

Soll das Item (über ein eigenständiges Symbol) in der App Bar erscheinen? Mögliche Werte:

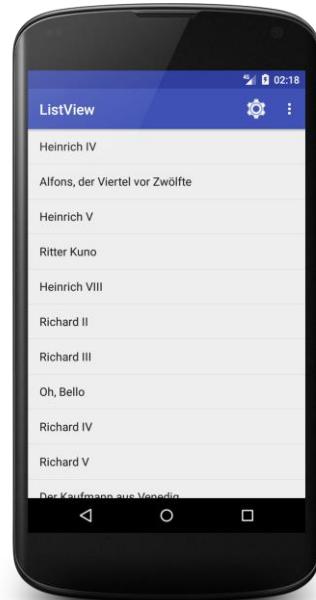
- **ifRoom**  
Das Item erscheint in der App Bar, wenn dort genügend Platz vorhanden ist.
- **always**  
Das Item soll unbedingt in der App Bar erscheinen. Wird der Auftritt für mehrere Items erzwungen, kann es zu Überlappungen kommen.
- **withText**  
Besitzt ein Item mit App Bar - Auftritt ein Icon, wird per Voreinstellung ausschließlich das Icon angezeigt. Durch den Wert **withText**, der per | - Operator mit den Werten **ifRoom** und **always** kombiniert werden kann, sorgt man dafür, dass ggf. auch der **title** angezeigt wird.
- **never** (= Voreinstellung)  
Das Item erscheint nach einem Tap auf das Symbol : im Überlaufmenü.

Das folgende Beispielprogramm verwendet die obige Menüdefinition und variiert dabei den Wert des Attributs **app:showAsAction** für das Item **Settings**:

**app:showAsAction="never"**



**app:showAsAction="ifRoom"**



In der rechten Variante hat das Item **Settings** einen eigenständigen Auftritt in der App Bar (per Symbol), während die restlichen Items per Überlaufmenü zugänglich sind.

Das im Beispiel für das Item **Settings** verwendete Icon

**android:icon="@drawable/ic\_settings"**

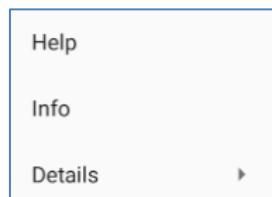
wurde aus SDK-Beständen als **drawable**-Ressource in das Projekt übernommen (siehe z. B. Abschnitt 8.5.3 zum Importverfahren).

### 8.6.1.2.2 Untermenüs

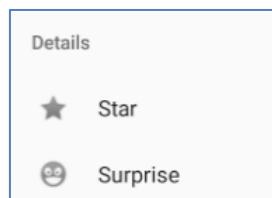
Um ein *Untermenü* zu realisieren, wird in ein **item**-Element ein **menu**-Element eingefügt, das wiederum mehrere **item**-Elemente aufnimmt, z. B. im folgenden Optionsmenü:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context="de.zimkand.menudemo.MainActivity">
    <item
        android:id="@+id/help"
        android:icon="@drawable/ic_menu_help"
        android:title="Help" />
    <item
        android:id="@+id/info"
        android:icon="@drawable/ic_menu_info_details"
        android:title="Info" />
    <item
        android:id="@+id_submenu"
        android:title="Details">
        <menu>
            <item
                android:id="@+id/star"
                android:icon="@drawable/ic_menu_star"
                android:title="Star" />
            <item
                android:id="@+id/surprise"
                android:icon="@drawable/ic_menu_emoticons"
                android:title="Surprise" />
        </menu>
    </item>
</menu>
```

Nach einem Klick auf das Überlaufsymbol ☰ mit den drei Punkten erscheint im Beispiel das Optionsmenü:



Nach einem Klick auf das dritte Item wird das Optionsmenü durch das folgende Untermenü ersetzt:



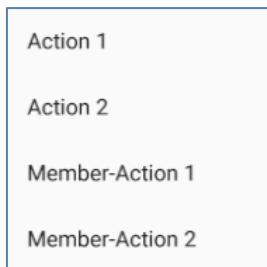
In Abschnitt 8.6.1.6 wird ein Trick vorgestellt, mit dem man Android dazu bringen kann, auch auf der Hauptebene des Optionsmenüs Icons zu zeigen.

### 8.6.1.2.3 Item-Gruppierung

Mit dem **group**-Element kann man eine Gruppe von Menüitems bilden, was im folgenden Optionsmenü demonstriert wird:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context="de.zimkand.menudemo.MainActivity">
    <item
        android:id="@+id/action1"
        android:title="Action 1" />
    <item
        android:id="@+id/action2"
        android:title="Action 2" />
    <group
        android:id="@+id/group"
        android:title="Group">
        <item
            android:id="@+id/maction1"
            android:title="Member-Action 1" />
        <item
            android:id="@+id/maction2"
            android:title="Member-Action 2" />
    </group>
</menu>
```

Von dieser Gruppierung ist im entfalteten Menü nichts zu erkennen:



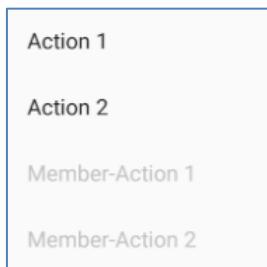
Allerdings können die Items einer Gruppe einer gemeinsamen Behandlung unterzogen werden, z. B. mit den folgenden **Menu**-Methoden:

- **public void setGroupVisible(int group, boolean visible)**  
Alle Items in der Gruppe werden (un)sichtbar gemacht.
- **public void setGroupEnabled(int group, boolean enabled)**  
Alle Items in der Gruppe werden (de)aktiviert.

In der folgenden Situation sind die Items in der Gruppe durch die Anweisung

```
menu.setGroupEnabled(R.id.group, false);
```

deaktiviert worden:

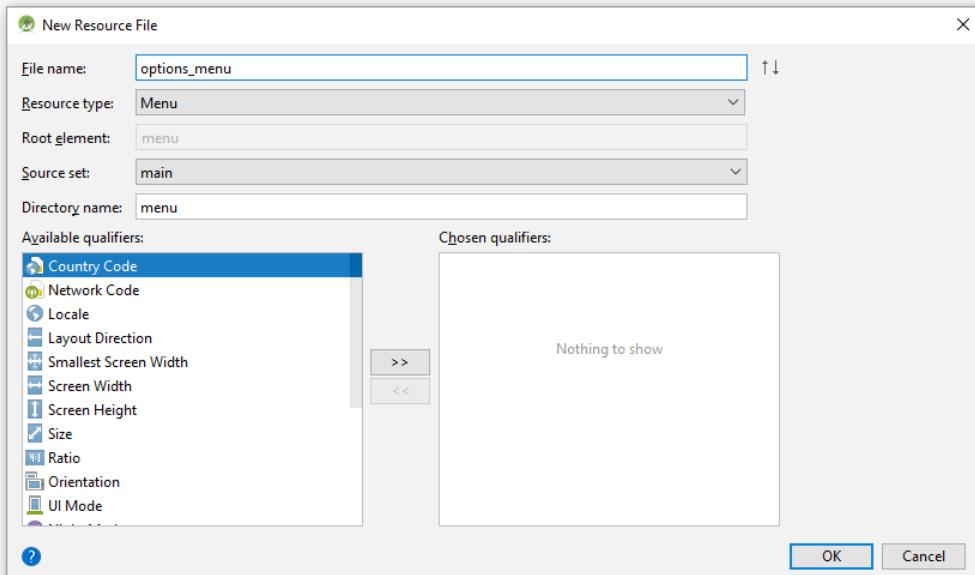


### 8.6.1.3 Menü-Ressource einfügen

Wir öffnen eine Kopie des ListView-Beispielprogramms aus Abschnitt 8.5.7, wählen aus dem Kontextmenü zum Knoten **app/res** im Projektexplorer das Item

**New > Android Resource file**

und erstellen im folgenden Fenster



die Datei **options\_menu.xml** mit **Menu** als **Resource type**. Nach dem Quittieren des Dialogs erscheint die Menüdefinition im Projektexplorer und wird im Editor geöffnet.

Wir beschäftigen uns zwar gerade mit generellen Eigenschaften der Android-Menüs, benötigen aber ein Beispiel, und dabei werden einige Details zum Optionsmenü (siehe Abschnitt 8.6.1) vorweggenommen. Wir fügen im **Design**-Modus aus der **Palette** ein **Menu Item** per Drag&Drop ein. Zur Platzierung kann man die Vorschau



oder den **Component Tree** verwenden.

Via **Attributes**-Fenster verpassen wir dem Menüitem:

- eine **id**
- einen **title**

Wie ein Wechsel zum **Text**-Modus zeigt, ist die folgende Menüdefinition entstanden:

```
<menu xmlns:app="http://schemas.android.com/apk/res-auto"
      xmlns:android="http://schemas.android.com/apk/res/android">

    <item
        android:id="@+id/settings"
        android:title="Settings" />
</menu>
```

Zum Befüllen des Optionsmenüs sind die Aktivität und alle darin geladenen Fragmente (siehe Kapitel 12) berechtigt. In der Regel verläuft das Befüllen so, dass eine Aktivität oder ein Fragment die

Methode **onCreateOptionsMenu()** überschreibt und durch das Inflationieren einer XML-Menüressource den per Parameter gelieferten **Menu**-Container befüllt, z. B.:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.menu_main, menu);
    return true;
}
```

#### 8.6.1.4 Optionsmenü zur Laufzeit anpassen

Für Menüanpassungen per Programm stehen u.a. die folgenden **Menu**-Methoden zur Verfügung:

- **public void removeItem(int id)**  
Ein Item wird entfernt, z. B.:  
`menu.removeItem(R.id.info);`  
Durch die folgende Anweisung wird ein Item aus einem *Untermenü* entfernt:  
`menu.findItem(R.id.details).getSubMenu().removeItem(R.id.star);`
- **public void removeGroup(int id)**  
Eine Itemgruppe wird entfernt.
- **public void clear()**  
Alle Items werden entfernt.
- **public void add(CharSequence title)**  
Ein Item wird ergänzt.

Statt ein Item über die **Menu**-Methode **removeItem()** zu entfernen, kann man es über die **MenuItem**-Methode **setVisible()** unsichtbar machen, z. B.:

```
menu.findItem(R.id.info).setVisible(false);
```

Ein Einsatz dieser Methoden kommt z. B. dann in Frage, wenn mehrere Aktivitäten viele Optionsmenüitems gemeinsam haben und von einer gemeinsamen Basisklasse abgeleitet werden, um Code-Wiederholungen zu vermeiden. Erforderliche Menüanpassungen in einer speziellen Aktivität lassen sich dann folgendermaßen erledigen:

- die geerbte **Activity**-Methode **onCreateOptionsMenu()** überschreiben
- zu Beginn die Basisklassenvariante aufrufen
- obige Methoden zur Anpassung des Menüs verwenden

Um das Optionsmenü zur Laufzeit zu ändern, sollte die **Activity**-Methode **onPrepareOptionsMenu()** verwendet werden:

```
public boolean onPrepareOptionsMenu(Menu menu)
```

Sie wird aufgerufen:

- nach dem Aufruf der Methode **onCreateOptionsMenu()**
- bei jedem Klick auf das Überlaufsymbol ☰ am rechten Rand der App Bar
- nachdem das Optionsmenü durch einen Aufruf der **Activity**-Methode **invalidateOptionsMenu()** als ungültig erklärt worden ist.

Sind in der App Bar Menüitems sichtbar, die kontextabhängig modifiziert werden sollen, muss der erforderliche Aufruf von **onPrepareOptionsMenu()** durch einen Aufruf von **invalidateOptionsMenu()** veranlasst werden.

### 8.6.1.5 Menüereignisbehandlung

Wenn der Benutzer ein Optionsmenüitem wählt, dann wird die Aktivitäts- bzw. Fragment-Methode **onOptionsItemSelected()** aufgerufen und per Parameter über das gewählte Item informiert. Zur Demonstration definieren wir ein einfaches und sinnfreies Optionsmenü:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context="de.zimkand.menudemo.MainActivity">
    <item
        android:id="@+id/action1"
        android:title="Action 1"
        app:showAsAction="ifRoom" />
    <item
        android:id="@+id/action2"
        android:title="Action 2" />
    <item
        android:id="@+id/action3"
        android:title="Action 3" />
</menu>
```

In der folgenden Menüereignis-Behandlungsmethode

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.action1:
            Toast.makeText(this, "Action 1", Toast.LENGTH_SHORT).show();
            return true;
        case R.id.action2:
            Toast.makeText(this, "Action 2", Toast.LENGTH_SHORT).show();
            return true;
        case R.id.action3:
            Toast.makeText(this, "Action 2", Toast.LENGTH_SHORT).show();
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

beschränken wir uns darauf, dem Benutzer eine Rückmeldung über das gewählte Item zu geben, indem ein **Toast**-Meldungsfenster für kurze Zeit erscheint, z. B.:



Die mit der statischen **Toast**-Methode **makeText()** erzeugte Meldung wird mit **show()** zum Auftritt aufgefordert, wobei die Anzeigedauer vom dritten **makeText()** - Parameter abhängt:

```
Toast.makeText(this, "Action 1", Toast.LENGTH_SHORT).show();
```

Nach einer erfolgreichen Menüereignisbehandlung sollte die Methode **onOptionsItemSelected()** den Wert **true** zurückmelden. Andernfalls sollte sie nach einer Empfehlung von Google die über-

schriebene Basisklassenvariante aufrufen und deren Rückgabe durchreichen (siehe Beispiel).<sup>1</sup> Von der voreingestellten Implementation wird dabei **false** zurückgemeldet.

Sind Fragmente vorhanden (siehe Kapitel 12), dann wird die folgende Aufrufreihenfolge realisiert:

- Zuerst wird die **onOptionsItemSelected()** - Methode der Aktivität aufgerufen.
- Dann kommen die Fragmente in der Reihenfolge der Aufnahme dran.
- Sobald eine Methode die Rückgabe **true** liefert, stoppt die Aufrufsequenz.

Seit Android 3 kann ein Menüitem auch über das Attribut **android:onClick** seines XML-Deklarationselements mit einer Klickbehandlungsmethode verknüpft werden, z. B.:

```
<item
    android:id="@+id/action1"
    android:icon="@drawable/ic_action1"
    android:title="Action 1"
    android:onClick="onAction1Click"
    app:showAsAction="ifRoom" />
```

Es kommt eine Methode der Aktivität mit folgenden Eigenschaften in Frage:

- Zugriffsstufe **public**
- Ein Parameter vom Typ **MenuItem**, keine weiteren Parameter

Ist eine Aktivität über die Methode **onOptionsItemSelected()** und über das Attribut **android:onClick** auf die Behandlung einer Menüentscheidung vorbereitet, dann gewinnt die Deklaration in der XML-Datei.

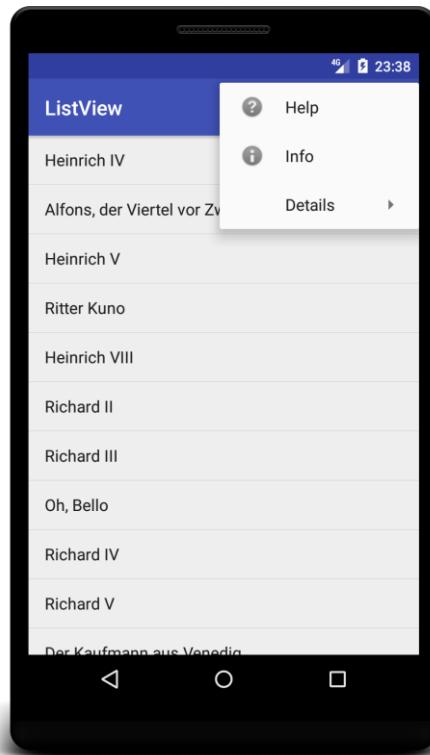
#### 8.6.1.6 Icons auf der Hauptebene des Optionsmenüs

Beim Optionsmenü werden zu den Items der Hauptebene die vereinbarten Icons *nicht* angezeigt, z. B.:



Wer gegen Googles Design-Vorstellungen das folgende Verhalten

<sup>1</sup> <https://developer.android.com/guide/topics/ui/menus>



durchsetzen möchte, muss etwas Aufwand betreiben. Durch die folgende Menüdefinition wird die Hauptebene in eine Unterebene verfrachtet:

```

<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context="de.zimkand.menudemo.MainActivity">
    <item
        android:id="@+id/dummy"
        app:showAsAction="always"
        android:icon="@drawable/ic_menu_3dots_transparent"
        android:title="">
        <menu>
            <item
                android:id="@+id/help"
                android:icon="@drawable/ic_menu_help"
                android:title="Help" />
            <item
                android:id="@+id/info"
                android:icon="@drawable/ic_menu_info_details"
                android:title="Info" />
            <item
                android:id="@+id/submenu"
                android:title="Details">
                <menu>
                    <item
                        android:id="@+id/star"
                        android:icon="@drawable/ic_menu_star"
                        android:title="Star" />
                    <item
                        android:id="@+id/surprise"
                        android:icon="@drawable/ic_menu_emoticons"
                        android:title="Surprise" />
                </menu>
            </item>
        </menu>
    </item>
</menu>

```

Das Einstiegsitem erhält ...

- beim Attribut **app:showAsAction** den Wert **always**:  
`app:showAsAction="always"`
- ein Icon mit drei weißen Punkten und einem transparenten Hintergrund  
`android:icon="@drawable/ic_menu_3dots_transparent"`  
Zur Erstellung genügen z. B. das Windows-Hilfsprogramm Paint und die Freeware IrfanView.
- eine leere Beschriftung:  
`android:title=""`

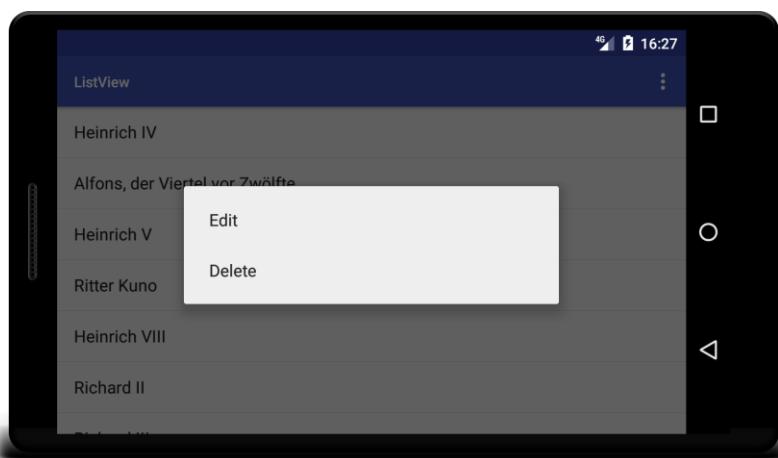
Bei dynamischen Änderungen eines so konstruierten Menüs ist zu beachten, dass sich die Hauptebenen-Items in einem Untermenü befinden.

## 8.6.2 Kontextmenü

Ein Kontextmenü kann zu jedem **View**-Objekt eingerichtet werden, doch sind Container wie **ListView** (vgl. Abschnitt 8.5.7) besonders relevante Einsatzorte. Bei der Realisation konkurrieren das traditionelle, schwebende Kontextmenü und die neuerdings von Google empfohlenen kontextabhängigen Aktionen.

### 8.6.2.1 Schwebendes Kontextmenü

Um in einer Aktivität oder in einem Fragment für ein Steuerelement ein schwebendes Kontextmenü wie im folgenden Beispiel zu realisieren,



sind folgende Schritte erforderlich:

- Das Steuerelement durch einen Aufruf der **Activity**- bzw. **Fragment**-Methode **registerForContextMenu()** für die Anzeige eines Kontextmenüs registrieren. Im folgenden Beispiel wird ein komplettes **ListView**-Objekt registriert:

```
ListView lv = findViewById(R.id.list);
registerForContextMenu(lv);
```

Als Parameter ist an **registerForContextMenu()** eine Referenz auf das zu versorgende Steuerelement zu übergeben.
- Die **Activity**- bzw. **Fragment**-Methode **onCreateContextMenu()** implementieren. Sie wird vor jeder Anzeige des Kontextmenüs aufgerufen und eignet sich dazu, für eine passende Ausstattung mit Menüitems zu sorgen, was in der Regel durch das Inflationieren einer Menüressource geschieht, z. B.:

```

@Override
public void onCreateContextMenu(ContextMenu contMenu, View v,
                               ContextMenu.ContextMenuItemInfo contextMenuItemInfo) {
    super.onCreateContextMenu(contMenu, v, contextMenuItemInfo);
    getMenuInflater().inflate(R.menu.context_menu, contMenu);
}

```

Die Methode erfährt über den zweiten Parameter, für welches **View**-Objekt das Kontextmenü angefordert wurde, und erhält über den dritten Parameter ein Objekt vom Typ **ContextMenu.ContextMenuItemInfo** genauere Informationen. Zu Abkömmlingen der Klasse **AdapterView<T extends Adapter>** wird ein Objekt der Klasse **AdapterView.AdapterContextMenuInfo** geliefert, das in der öffentlichen Instanzvariablen **id** den Indexwert der vom Kontextmenü betroffenen Tabellenzeile aufbewahrt.

Zu Beginn der Methode wird einer üblichen Praxis folgend die überschriebene Basisklassenvariante aufgerufen.<sup>1</sup>

Im Beispiel wird die folgende Kontextmenüdefinition verwendet:

```

<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/conit_edit"
          android:title="Edit" />
    <item android:id="@+id/conit_del"
          android:title="Delete" />
</menu>

```

- Die **Activity**- bzw. **Fragment**-Methode **onContextItemSelected()** implementieren. Sie wird aufgerufen, wenn der Benutzer ein Item aus dem Kontextmenü gewählt hat. Um das Item zu identifizieren, fragt man das mitgelieferte Parameterobjekt mit der Methode **getitemId()** nach seiner Kennung, z. B.:

```

@Override
public boolean onContextItemSelected(MenuItem item) {
    AdapterView.AdapterContextMenuInfo acmi =
        (AdapterView.AdapterContextMenuInfo) item.getMenuInfo();
    switch (item.getItemId()) {
        case R.id.conit_edit:
            Toast.makeText(this, "Edit with "+String.valueOf(acmi.id),
                         Toast.LENGTH_SHORT).show();
            return true;
        case R.id.conit_del:
            Toast.makeText(this, "Del with "+String.valueOf(acmi.id),
                         Toast.LENGTH_SHORT).show();
            return true;
        default:
            return super.onContextItemSelected(item);
    }
}

```

Gehört das Kontextmenü zu einer Ableitung der Klasse **AdapterView<T extends Adapter>**, dann liefert die vom Parameterobjekt beherrschte Methode **getMenuInfo()** ein Objekt der Klasse **AdapterView.AdapterContextMenuInfo**, das in der öffentlichen Instanzvariablen **id** den Indexwert des vom Kontextmenü betroffenen Elements aufbewahrt. Nach einer erfolgreichen Behandlung sollte die Methode den Wert **true** zurückmelden. Andernfalls sollte sie die überschriebene Basisklassenvariante aufrufen und deren Rückgabe durchreichen.

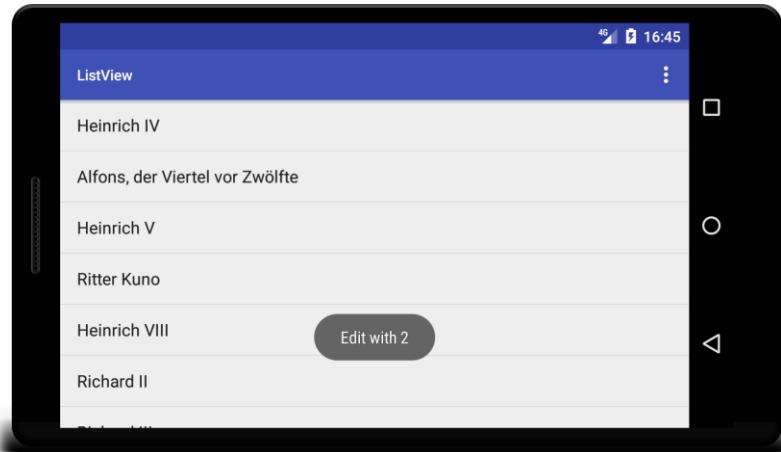
Sind Fragmente vorhanden (siehe Kapitel 12), dann wird die folgende Aufrufreihenfolge realisiert:

---

<sup>1</sup> In der Methode **onCreateContextMenu()** der Klasse **Activity** (im Paket **android.app**) passiert auf dem API-Level 27 (Android 8.1) allerdings nichts (siehe <https://developer.android.com/guide/topics/ui/menus>).

- Zuerst wird die **onContextItemSelected()** - Methode der Aktivität aufgerufen.
- Dann kommen die Fragmente in der Reihenfolge der Aufnahme dran.
- Sobald eine Methode die Rückgabe **true** liefert, stoppt die Aufrufsequenz.

Im Beispiel findet nach der Wahl eines Kontextmenüitems statt der versprochenen Aktion lediglich eine Toast-Anzeige statt (vgl. Abschnitt 8.6.1.5):



Bald werden wir in der Lage sein, aufgrund einer Kontextmenüwahl z. B. per Intent-Objekt eine andere Aktivität zu starten, die ein Editieren des Eintrags erlaubt.

Das gesamte Beispielprogramm befindet sich im AS-Projekt

**...\\BspUeb\\UI\\Menu\\ListView with Floating Context Menu**

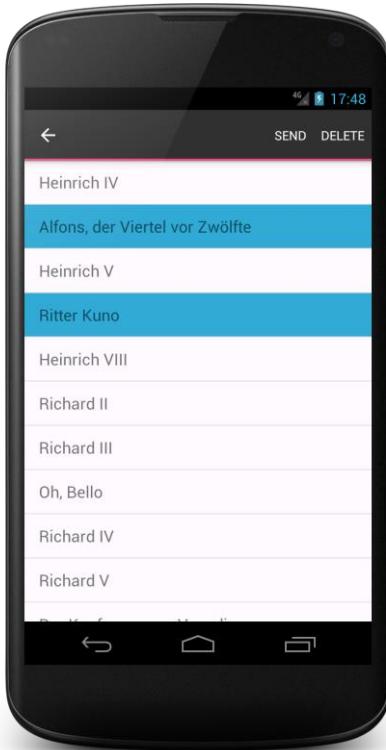
### 8.6.2.2 Kontextabhängige Aktionen

An Stelle des traditionellen schwebenden Kontextmenüs empfiehlt Google die sogenannten *kontextabhängigen Aktionen* (engl.: *contextual action mode*). Nachdem der Kontextaktionsmodus (z. B. durch einen Langklick) gestartet worden ist, übernimmt die sogenannte **Context Action Bar** (CAB) den Platz der App Bar und bietet kontextabhängige Aktionen an. Besonders nützlich ist die Möglichkeit, kontextabhängige Aktionen (z. B. das Löschen) auf *mehrere*, simultan markierte Elemente eines **ListView**- oder **GridView**-Steuerelements anzuwenden.

Wir ersetzen in der App aus dem letzten Abschnitt das schwebende Kontextmenü durch kontextabhängige Aktionen und übernehmen dabei eine Lösung von einer Google-Webseite für Android-Entwickler.<sup>1</sup> Im Endergebnis kann per Langklick auf ein Listenelement die Context Action Bar aktiviert werden, die den Platz der App Bar übernimmt. Wie das anschließend präsentierte Bildschirmfoto zeigt, haben die kontextabhängigen Aktionen zwei erhebliche Vorteile gegenüber dem schwebenden Kontextmenü:

- Es wird kein Teil der Aktivitätsoberfläche verdeckt.
- Der Benutzer kann eine beliebige Teilmenge von zu bearbeitenden Listenelementen markieren:

<sup>1</sup> <https://developer.android.com/guide/topics/ui/menus.html>



Nachdem ein Item aus der CAB gewählt und die zugehörige Ereignisbehandlung abgeschlossen wurde, kehrt man in der Regel zur App Bar zurück. Weitere Möglichkeiten zum Beenden des CAB-Modus sind:

- Alle Markierungen aufheben
- Touch auf den Linkspfeil am linken CAB-Rand
- Betätigung der generellen Rückwärtstaste

Bei den Klassen **ListView**, **GridView** oder einer anderen Ableitung der Klasse **AbsListView** sind nur zwei Schritte erforderlich, um kontextabhängige Aktionen für eine flexibel wählbare Teilmenge der Listenelemente zu ermöglichen.

Zunächst wird der Wahlmodus auf **ListView.CHOICE\_MODE\_MULTIPLE\_MODAL** gesetzt, z. B.:

```
lv.setChoiceMode(ListView.CHOICE_MODE_MULTIPLE_MODAL);
```

Der zweite Schritt ist etwas aufwändiger und besteht darin, dem **List-** oder **GridView-** Steuerelement ein Objekt vom Typ **AbsListView.MultiChoiceModeListener** durch einen Aufruf der Methode **setMultiChoiceModeListener()** zur Verfügung zu erstellen. Im Beispiel stammt das Parameterobjekt aus einer anonymen Klasse:

```
lv.setMultiChoiceModeListener(new AbsListView.MultiChoiceModeListener() { ... });
```

Dieses Objekt ist für die kontextabhängigen Aktionen zuständig und muss einige Rückrufmethoden beherrschen, die nun beschrieben werden.

In der einmalig aufgerufenen Methode **onCreateActionMode()** wird das kontextabhängige Aktionsmenü in der Regel durch das Inflationieren einer XML-Menüdefinition befüllt, z. B.:

```
public boolean onCreateActionMode(ActionMode mode, Menu menu) {
    getMenuInflater().inflate(R.menu.context_menu, menu);
    return true;
}
```

Mit dem Rückgabewert entscheidet man darüber, ob der kontextabhängige Aktionsmodus gestartet oder abgebrochen werden soll.

Vor jeder CAB-Präsentation wird die Methode **onPrepareActionMode()** aufgerufen, die eine Anpassung der Menüzusammenstellung ermöglicht. Im Beispiel wird nur durch den Rückgabewert **false** darüber informiert, dass *keine* Änderung vorgenommen worden ist:

```
public boolean onPrepareActionMode(ActionMode mode, Menu menu) {
    return false;
}
```

Mit dem Rückgabewert entscheidet man darüber, ob die CAB geändert worden ist.

In der Methode **onDestroyActionMode()** kann man darauf reagieren, dass der Benutzer den CAB-Modus verlassen hat:

```
@Override
public void onDestroyActionMode(ActionMode mode) {
```

In der Methode **onItemCheckedStateChanged()** kann man darauf reagieren, dass der Benutzer die Menge der markierten Elemente geändert hat. Eine mögliche Reaktion besteht z. B. darin, die Anzeige der Anzahl markierter Elemente in der CAB-Titelzeile zu ändern. Im Beispiel passiert nichts:

```
@Override
public void onItemCheckedStateChanged(ActionMode mode, int position,
                                      long id, boolean checked) {
```

Schließlich muss kodiert werden, was bei Wahl einer Aktion passieren soll, wobei wir uns im Beispiel auf Andeutungen per Toast-Anzeige beschränken (vgl. Abschnitt 8.6.1.5):

```
@Override
public boolean onActionItemClicked(ActionMode mode, MenuItem item) {
    switch (item.getItemId()) {
        case R.id.conit_send:
            Toast.makeText(MainActivity.this, "Send selected items",
                         Toast.LENGTH_SHORT).show();
            mode.finish();
            return true;
        case R.id.conit_del:
            Toast.makeText(MainActivity.this, "Delete selected items",
                         Toast.LENGTH_SHORT).show();
            mode.finish();
            return true;
        default:
            return false;
    }
}
```

Welche Aktion gewählt wurde, ist vom **MenuItem**-Parameterobjekt über einen Aufruf der Methode **getItemId()** zu erfahren. Nach Ausführung einer Aktion wird in der Regel der CAB-Modus beendet:

```
mode.finish();
```

Mit dem Rückgabewert informiert man Android darüber, ob das Ereignis als behandelt gelten soll.

Hier ist der vollständige **setMultiChoiceModeListener()** - Aufruf zu sehen:

```

lv.setMultiChoiceModeListener(new AbsListView.MultiChoiceModeListener() {
    @Override
    public boolean onCreateActionMode(ActionMode mode, Menu menu) {
        MenuInflater inflater = mode.getMenuInflater();
        inflater.inflate(R.menu.context_menu, menu);
        return true;
    }
    @Override
    public boolean onPrepareActionMode(ActionMode mode, Menu menu) {
        return false;
    }
    @Override
    public boolean onActionItemClicked(ActionMode mode, MenuItem item) {
        switch (item.getItemId()) {
            case R.id.conit_send:
                Toast.makeText(MainActivity.this, "Send selected items",
                    Toast.LENGTH_SHORT).show();
                mode.finish();
                return true;
            case R.id.conit_del:
                Toast.makeText(MainActivity.this, "Delete selected items",
                    Toast.LENGTH_SHORT).show();
                mode.finish();
                return true;
            default:
                return false;
        }
    }
    @Override
    public void onDestroyActionMode(ActionMode mode) {
    }

    @Override
    public void onItemCheckedStateChanged(ActionMode mode, int position, long id,
                                         boolean checked) {
    }
});

```

Das gesamte Beispielprogramm befindet sich im AS-Projekt

**...\\BspUeb\\UI\\ListView with Contextual Action Mode**

Kontextabhängige Aktionen sind zwar im Zusammenhang mit einem **ListView**- oder **GridView**-Steuerelement besonders attraktiv, doch kommen sie auch bei einem einzelnen Bedienelement in Frage.<sup>1</sup>

### 8.6.3 PopUp-Menü

Das PopUp-Menü ähnelt optisch dem schwebenden Kontextmenü, soll aber *nicht* dazu verwendet werden, Änderungen am auslösenden Steuerelement vorzunehmen. Man nutzt es häufig dazu, um Ausführungsdetails zu einer vom Benutzer gewählten Aktion zu erfragen. Man kann z. B. per PopUp-Menü die verfügbaren sozialen Netzwerke zur Auswahl anbieten, nachdem sich ein Benutzer per Schaltfläche dafür entschieden hat, ein Foto mit anderen zu teilen. Gelegentlich (z. B. in der GMail-App) wird ein PopUp-Menü durch ein Überlaufsymbol ☰ geöffnet, das mit einem konkreten Element (z. B. mit einer Mail) verbunden ist. Das PopUp-Menü sollte dann Optionen anbieten, die sich auf das konkrete Element beziehen, ohne es zu verändern.<sup>2</sup>

---

<sup>1</sup> <https://developer.android.com/guide/topics/ui/menus>

<sup>2</sup> <https://developer.android.com/guide/topics/ui/menus>

Wir verwenden ein einfaches Beispiel mit einem Schalter, dessen Klickbehandlungsmethode ein PopUp-Menü öffnet. Um den Schalter mit seiner Klickbehandlungsmethode zu verbinden, verwenden wir der Abwechslung halber die in Abschnitt 8.4.2.3 beschriebene Technik, ...

- in der Aktivitätsklasse eine Methode mit ...
  - der Zugriffsstufe **public**,
  - und einem Parameter vom Typ **View**
 zu definieren,
- in der Layout-Definition den Methodennamen als Wert des Schalter-Attributs **android:onClick** zu verwenden, z. B.:
   
**android:onClick="showSharePopup"**

In der Klickbehandlungsmethode wird ein Objekt der Klasse **PopupMenu** erstellt und dem Schalter zugeordnet:

```
public void showSharePopup(View v) {
    PopupMenu popup = new PopupMenu(this, v);
    getMenuInflater().inflate(R.menu.popup_menu, popup.getMenu());
    popup.setOnMenuItemClickListener(this);
    popup.show();
}
```

Das Menü wird unter Verwendung der folgenden XML-Menüdefinition

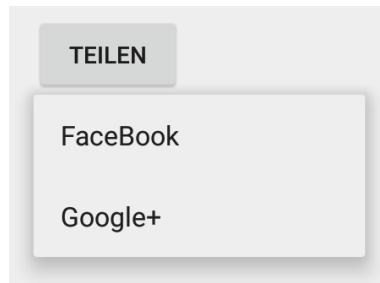
```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/facebook"
        android:title="FaceBook" />
    <item
        android:id="@+id/google_plus"
        android:title="Google+" />
</menu>
```

entfaltet. Bevor das **PopupMenu**-Objekt per **show()** - Methode zum Auftritt gebeten wird, erfährt es noch, dass die Aktivität die **OnMenuItemClickListener**-Rolle übernimmt.

Um dieser Rolle gerecht zu werden, muss sich die Aktivitätsklasse in ihrem Definitionskopf zu dem Interface **PopupMenu.OnMenuItemClickListener** bekennen und die somit erforderliche Methode **onMenuItemClick()** implementieren:

```
@Override
public boolean onMenuItemClick(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.facebook:
            Toast.makeText(this, "Facebook", Toast.LENGTH_SHORT).show();
            return true;
        case R.id.google_plus:
            Toast.makeText(this, "Google+", Toast.LENGTH_SHORT).show();
            return true;
        default:
            return false;
    }
}
```

Nach einem Klick auf den Schalter erscheint das Popup-Menü:



Das Beispielprogramm befindet sich im AS-Projekt

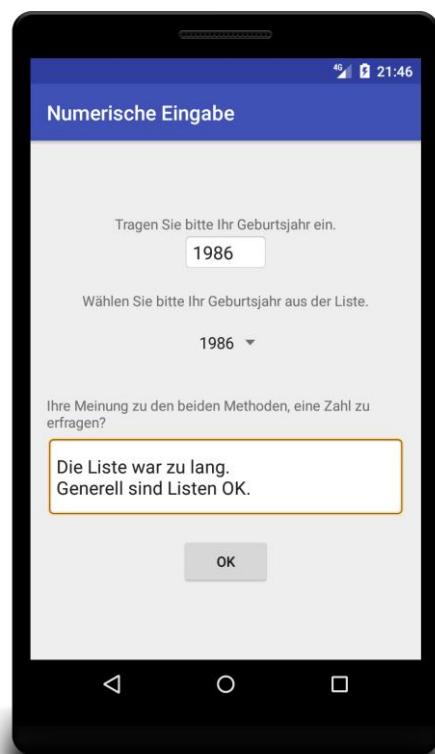
**...\\BspUeb\\UI\\Menu\\PopUp Menu**

### **8.7 Übungsaufgaben zu Kapitel 8**

1) Erstellen Sie eine App, die das Geburtsjahr des Anwenders (zulässige Werte: von 1900 bis 1999) auf zweierlei Weise erfasst:

- Per **EditText**-Element
- Per **Spinner**-Element

Prüfen Sie jeweils, ob eine Antwort in erlaubten Wertebereich vorliegt. Die App soll ungefähr die folgende Bedienoberfläche anbieten:



2) Welche von den folgenden Aussagen sind richtig bzw. falsch?

1. Bei einem **EditText**-Steuerelement muss der Mehrzeilenmodus mit dem Wert **textMultiLine** für das Attribut **android:inputType** erzwungen werden.
2. Das **LinearLayout** und das **ConstraintLayout** erlauben über Gewichtsattribute eine flexible Restraumverteilung auf Zeilen bzw. Spalten.
3. Welche Attribute zur Positionierung für ein Steuerelement zur Verfügung stehen, hängt vom umgebenden Container ab.
4. Das **ConstraintLayout** wird vom Layout-Editor im Android Studio besser unterstützt als jeder andere Layout-Manager.

## 9 Intents und Intent-Filter

Objekte der Klasse **Intent** im Paket **android.content** dienen zur Kommunikation zwischen den Komponenten in einem Android-System, innerhalb einer Anwendung und auch über Anwendungsgrenzen hinweg. Sie kommen zum Einsatz, ...

- um einen **Auftrag zu erteilen**,
- um über ein **Ereignis** (meist auf Systemebene) zu informieren (z. B. WLAN aktiviert).

In der englischen Sprache ist *intent* die zulässige Kurzform von *intention* (dt.: *Absicht, Vorhaben*).

Als Auftragnehmer für ein **Intent**-Objekt kommen folgende App-Komponenten in Frage:

- **Activities**

Zum Starten einer neuen Aktivität kann z. B. die Methode **startActivity(Intent intent)** verwendet werden.

- **Services**

Wenn eine Komponente (z. B. eine Aktivität) über die Methode **bindService(Intent service, ServiceConnection conn, int flags)** eine Client-Server - Beziehung zu einem Dienst aufbaut, wird ein **Intent**-Objekt als Parameter benötigt. Mit der Methode **startService(Intent service)** lässt sich ein Service beauftragen, einen Einzelauftrag im Hintergrund selbstständig auszuführen (z. B. einen Datei-Download).

- **Broadcast Receiver**

Mit der Methode **sendBroadcast(Intent intent)** wird ein Rundruf an alle registrierten Empfänger gesendet.

Alle in der Aufzählung genannten Methoden sind in der Klasse **Context** definiert, von der u.a. die Klassen **Activity** und **Service** (indirekt) abstammen.

Wir beschränken uns im aktuellen Kapitel auf die Verwendung von Intents bei der Kommunikation zwischen Aktivitäten, weil Services und Broadcast Receiver noch nicht behandelt worden sind.

**Explizite** Intents enthalten den voll qualifizierten Klassennamen (inkl. Paketname) der ausführenden Komponente (Activity oder Service) und dienen hauptsächlich zur Kommunikation *innerhalb* einer Anwendung.

Dienstleistungen *fremder* Apps lassen sich über **implizite** Intents nutzen. Statt eines Komponentennamens enthalten sie eine Auftragsbeschreibung, im Wesentlichen bestehend aus einer **Aktion** und den zu verarbeitenden **Daten**. Wer als Auftragnehmer in Frage kommt, ermittelt Android durch den Vergleich ...

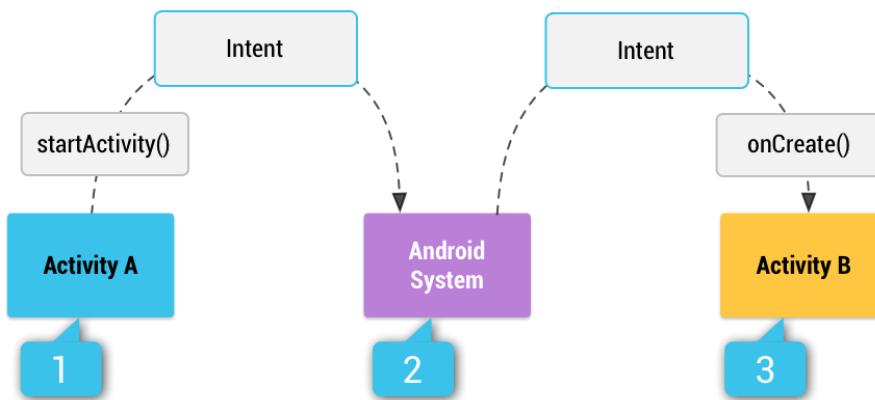
- der Intent-Eigenschaften (siehe Abschnitt 9.1)
- mit den Intent-Filtern der installierten Apps (siehe Abschnitt 9.2).

In der App-Manifestdatei sind die Verarbeitungskapazitäten einer Komponente durch einen oder durch mehrere Intent-Filter beschrieben.

In der folgenden Abbildung beschreibt Google den Ablauf beim Starten einer Aktivität über einen *impliziten* Intent:<sup>1</sup>

---

<sup>1</sup> <https://developer.android.com/guide/components/intents-filters>



Aktivität A verpackt einen Auftrag in einen Intent und übergibt diesen durch Aufruf der Methode `startActivity()` an das Betriebssystem. Android ermittelt eine passende Aktivität, startet diese und übergibt dabei den Intent. Kommen mehrere Auftragnehmer in Frage, erhält der Benutzer einen Auswahldialog.

Mit dem Intent-System realisiert Android eine Kooperation zwischen Apps nach dem Prinzip der **späten Bindung**. Wenn ein App-Programmierer eine fremde Activity zur Erledigung einer Teilaufgabe einspannen will (z. B. Kontakt anrufen, Mail versenden, aktuelle Position in einer Landkarte anzeigen), benötigt er keine konkreten Informationen über den Auftragnehmer. Welche Aktivität durch Vermittlung des Betriebssystems letztlich tätig wird, entscheidet sich erst zur Laufzeit.

## 9.1 Bestandteile eines Intent-Objekts

### 9.1.1 Name der Komponente

Wird als Auftragnehmer eine Anwendungskomponente explizit spezifiziert (über Paket und Klasse), dann resultiert ein *explizites Intent*-Objekt, und es findet keine weitere Auswertung von Intent-Bestandteilen zur Ermittlung des Auftragsnehmers statt. Apps verwenden explizite Intents hauptsächlich dazu, ihre eigenen Aktivitäten oder Dienste zu starten.

Bei der Erstellung eines expliziten Intent-Objekts per Konstruktor sind zwei Parameter im Spiel:

`public Intent(Context packageContext, Class<?> cls)`

Als **Context** gibt man in der Regel das aktuelle **Activity**-Objekt an. Im zweiten Parameter ist ein Objekt gefragt, das die explizit zu beauftragende Klasse beschreibt. Man erhält die benötigte Referenz z. B. durch den Klassennamen mit angehängtem Schlüsselwort **class**.

Weil die folgende Anweisung aus einer Ereignisbehandlungsmethode in einer *anonymen* Klasse stammt, wird im ersten Parameter dem Schlüsselwort **this** der Name der umgebenden **Activity**-Klasse vorangestellt, um das Aktivitätsobjekt anzusprechen:

```
Intent secAct = new Intent(MainActivity.this, SecondaryActivity.class);
```

Aus den beiden Parametern der vorgestellten **Intent**-Konstruktorüberladung entsteht ein Objekt vom Typ **ComponentName**, das eine Android-Komponente (über ein Paket und eine Klasse) identifiziert. Statt per **Intent**-Konstruktor kann die Komponente auch durch eine von den folgenden **Intent**-Methoden festgelegt werden:

- **public Intent setClass(Context packageContext, class<?> class)**

Im folgenden Beispiel wird ein leeres **Intent**-Objekt erzeugt und dann analog zum oben vorgeführten Konstruktoraufruf durch den Kontext und das Klassenobjekt über die Auftragnehmer-Komponente infomiert:

```
Intent secAct = new Intent();
secAct.setClass(MainActivity.this, SecondaryActivity.class);
```

- **public Intent setComponent(ComponentName component)**

Die Methode **setComponent()** erwartet ein Parameterobjekt vom Typ **ComponentName**, z. B.:

```
Intent secAct = new Intent();
secAct.setComponent(new ComponentName(MainActivity.this,
SecondaryActivity.class));
```

Im Beispiel wird der folgende **ComponentName**-Konstruktor verwendet:

```
public ComponentName(Context package, Class<?> class)
```

Es sind auch **ComponentName**-Konstruktorüberladungen mit **String**-Parametern vorhanden, z. B.:

```
public ComponentName(String package, String class)
```

Bei Ihrer Verwendung kann der Compiler allerdings keine Existenzprüfung vornehmen, so dass es ggf. zu einem Laufzeitfehler kommt.

Ein zu startender Service sollte aus Sicherheitsgründen auf jeden Fall explizit benannt werden. Bei einem impliziten Intent wäre unsicher, welcher Dienst tätig wird, was zudem ohne Kontrollmöglichkeit durch den Benutzer passieren würde.

### 9.1.2 Aktion

Implizite Intents enthalten statt eines Komponentennamens eine Auftragsbeschreibung, die im Wesentlichen aus einer Aktion und den zu verarbeitenden Daten besteht. Eine Aktion ...

- soll von einer anderen Anwendungskomponente (Activity oder Service) ausgeführt werden
- oder hat stattgefunden und soll nun per Broadcast bekanntgegeben werden.

Die Aktion eines Intents wird per **String**-Objekt angegeben, wobei in Frage kommen:

- In der Klasse **Intent** oder anderen SDK-Klassen (z. B. **Settings**) vordefinierte Konstanten  
In der Klasse **Intent** ist z. B. die Aktion **ACTION\_VIEW** definiert, mit der eine Komponente aufgefordert wird, Daten (z. B. eine Liste mit Kontakten, ein Bild oder eine Webseite) anzuzeigen:

```
public static final String ACTION_VIEW = "android.intent.action.VIEW";
```

- Selbst definierte und durch eigene Anwendungskomponenten auszuführende Aktionen  
Bei der Definition ist darauf zu achten, den Paketnamen der App als Präfix zu verwenden, z. B.:

```
static final String ACTION_TUWAS = "de.zimkand.intentdemo.action.TUWAS";
```

Man kann die Aktion über eine geeignete **Intent**-Konstruktorüberladung oder mit der Methode **setAction()** festlegen, z. B.:

```
abook.setAction(Intent.ACTION_VIEW);
```

In der folgenden Tabelle sind wichtige, in der Klasse **Intent** definierte Aktionen aufgelistet:<sup>1</sup>

---

<sup>1</sup> <https://developer.android.com/reference/android/content/Intent>

Intent-Feldname	Ziel	Zweck
ACTION_MAIN	Activity	Aktivität als Basis für eine neue Aufgabe starten
ACTION_VIEW	Activity	Angegebene Daten anzeigen (z. B. einen Kontakt oder ein Foto)
ACTION_EDIT	Activity	Modifikation der angegebenen Daten ermöglichen
ACTION_SEND	Activity	Daten versenden (z. B. als Mail)
ACTION_PICK	Activity	Wahl eines Items aus den angegebenen Daten ermöglichen, Auswahl zurückliefern
ACTION_WEB_SEARCH	Activity	Beginnt der Suchtext mit <i>http</i> oder <i>https</i> , wird eine Webseite geöffnet, sonst startet die Google-Suche.
ACTION_POWER_CONNECTED	Broadcast Receiver	Informiert darüber, dass eine externe Stromversorgung angeschlossen worden ist
ACTION_DATE_CHANGED	Broadcast Receiver	Für Datumswechsel interessiert sich z. B. eine App, die Erinnerungen versenden soll.
ACTION_HEADSET_PLUG	Broadcast Receiver	Der Kopfhörer ist eingesteckt oder entfernt worden.

In der Klasse **Settings** sind Aktionen definiert, die bestimmte Seiten der Einstellungs-App starten, z. B. ACTION\_SECURITY\_SETTINGS, ACTION\_USER\_DICTIONARY\_SETTINGS.

Von der auszuführenden Aktion hängen die weiteren Bestandteile eines Intents wesentlich ab.

### 9.1.3 Daten

Zur Beschreibung der zu verarbeitenden Daten kommen in Frage:

- Adresse der Daten (URI)  
Die Adressierung erfolgt über ein Objekt der Klasse **Uri**, das einen URI (*Uniform Resource Identifier*) darstellt.
- MIME-Typ der Daten  
Durch einen aus Typ und Subtyp bestehenden MIME-Typ (z. B. **text/html**) wird die Beschaffenheit der zu verarbeitenden Daten beschrieben, wobei neben standardisierten Typen<sup>1</sup> auch Anbieter-spezifische Typen möglich sind. Android schafft es oft, aber nicht immer, den MIME-Typ aus dem **Uri**-Objekt zu erschließen. Es kann z. B. erforderlich sein, den MIME-Typ explizit zu definieren, wenn es sich bei einer Internet-Adresse im **Uri**-Objekt z. B. um eine Bild- oder eine Audiodatei handeln könnte.

Mit den folgenden **Intent**-Methoden kann man den URI und/oder den MIME-Typ der zu bearbeitenden Daten festlegen:

- **setData(Uri daten)** setzt das **Uri**-Objekt mit der Adresse der Daten  
Mit **setData()** gibt man z. B. eine zu öffnende Webseite an. Das Parameterobjekt vom Typ **Uri** muss aber nicht unbedingt die vollständige Adresse eines Dokuments enthalten, sondern kann sich z. B. darauf beschränken, ein Protokoll bzw. Schema festzulegen (siehe unten).
- **setType(String typ)** setzt den MIME-Typ der Daten
- **setDataAndType(Uri daten, String typ)** setzt den URI und den MIME-Typ der Daten  
Die beiden zuerst genannten Methoden nacheinander aufzurufen, hat *nicht* denselben Effekt wie ein Aufruf der Methode **setDataAndType()**, weil sie sich in ihrer Wirkung gegenseitig aufheben. Die automatische Ableitung des Datentyps aus dem URI wird durch die explizite Typdeklaration abgeschaltet.

<sup>1</sup> Siehe z. B.<http://www.iana.org/assignments/media-types/media-types.xhtml>

Ein Intent kann zur Beschreibung der zu bearbeitenden Daten enthalten:

- nur einen URI
- nur einen Datentyp (einen MIME-Typ)
- beides
- keines von beiden

Einige Beispiele sollen die etwas abstrakte Darstellung konkretisieren und dabei die Vielfalt der möglichen Datenspezifikationen illustrieren. Alle Beispiele werden später in `startActivity()` - Aufrufen verwendet.

Im ersten Beispiel soll per **ACTION\_SEND** ein einfacher Text (MIME-Typ **text/plain**) über ein beliebiges soziales Netzwerk versendet werden. Dabei wird der Text als Extras-Bestandteil in das **Intent**-Objekt eingefügt (siehe Abschnitt 9.1.5). Es ist kein **Uri**-Objekt mit einer Datenadresse beteiligt, sodass kein `setData()` - Aufruf benötigt wird:

```
Intent intent = new Intent(Intent.ACTION_SEND);
intent.setType("text/plain");
intent.putExtra(Intent.EXTRA_TEXT, "Zu versender Text");
```

Im zweiten Beispiel soll eine geographische Position auf einer Karte angezeigt werden. Per `setData()` wird ein **Uri**-Objekt mit Protokollangabe und Position in den Intent eingefügt, während kein MIME-Typ beteiligt ist:

```
Intent intent = new Intent(Intent.ACTION_VIEW);
intent.setData(Uri.parse("geo:49.747355,6.687724"));
```

Im letzten Beispiel wird die Aufnahme eines neuen Adressbucheintrags beantragt (Aktion **ACTION\_INSERT**). Das Adressbuch wird über ein **Uri**-Objekt angesprochen, aus dem der MIME-Typ von Android abgeleitet werden kann. Über `putExtra()` -Aufrufe beschreibt man den neuen Eintrag:

```
Intent intent = new Intent(Intent.ACTION_INSERT);
intent.setData(Uri.parse("content://com.android.contacts/contacts"));
intent.putExtra(ContactsContract.Intents.Insert.NAME, "Kurt Saar");
```

Bei der Ausgestaltung des **Intent**-Objekts für einen konkreten Zweck hilft eine Liste mit Lösungen für typische Anwendungsfälle. Google präsentiert eine solche Liste z. B. auf der folgenden Webseite:

<https://developer.android.com/guide/components/intents-common>

Im weiteren Verlauf des Abschnitts folgen noch einige Details zur Datenspezifikation durch URIs. Die für unsere Zwecke wesentlichen URI-Bestandteile sind:

*schema://host:port/pfad?abfrage*

Beispiele:

```
https://www.source.com/download/data/chart.svg
mailto:
content://com.android.contacts/contacts
```

Beim Schema **content** ist klar, dass die Daten durch einen auf dem lokalen Gerät installierten Content Provider (siehe Kapitel 14) verwaltet werden. In diesem Fall kann Android den MIME-Typ problemlos ermitteln.

Aktion und Daten definieren zusammen, was getan werden soll, z. B.:<sup>1</sup>

---

<sup>1</sup> Die Beispiele stammen von der Webseite <https://developer.android.com/reference/android/content/Intent>

Aktion	Per URI beschriebene Daten	Resultierender Auftrag
ACTION_VIEW	content://com.android.contacts/contacts	Liste mit allen Kontakten anzeigen
ACTION_VIEW	content://com.android.contacts/contacts/1	Kontakt Nr. 1 anzeigen
ACTION_INSERT	content://com.android.contacts/contacts	Neuen Kontakt einfügen

Eine häufig verwendete **Intent**-Konstruktorüberladung besitzt Parameter für die Aktion und die per URI spezifizierten Daten:

```
public Intent(String action, Uri uri)
```

Häufig benötigte **Uri**-Objekte sind über Konstanten von SDK-Klassen ansprechbar. So existiert z. B. in der internen Klasse **android.provider.ContactsContract.Contacts** eine statische, finalisierte und öffentliche **Uri**-Referenzvariable namens **CONTENT\_URI**, die auf ein **Uri**-Objekt zum Android-Adressbuch zeigt:

```
content://com.android.contacts/contacts
```

Im folgenden Code-Segment wird ein implizites **Intent**-Objekt erzeugt, das die Anzeige der Kontakte im Android-Adressbuch veranlasst. Es wird als Parameter der Methode **startActivity()** verwendet, die Android auffordert, eine Aktivität zu ermitteln und zu starten, die den Auftrag ausführen kann:

```
Intent abook = new Intent(Intent.ACTION_VIEW, ContactsContract.Contacts.CONTENT_URI);
if (abook.resolveActivity(getApplicationContext()) != null)
    startActivity(abook);
```

Um eine App-Terminierung mit Ausnahmefehler zu vermeiden, wird im Beispiel der Startauftrag nur dann erteilt, wenn der Package-Manager von Android bestätigt hat, dass mindestens eine als Auftragnehmer geeignete Aktivität installiert ist (vgl. Abschnitt 9.4.2).

#### 9.1.4 Kategorien

Eine Intent-Kategorie wird per **String**-Objekt angegeben und enthält Informationen über die Beschaffenheit der Komponente, welche den Auftrag erledigen kann. Zu einem Intent sind beliebig viele Kategorien erlaubt, doch ist es bei den meisten Intents nicht erforderlich, eine Kategorie explizit anzugeben.

Relevante Kategorien sind:

- **CATEGORY\_BROWSABLE**  
Es kommt nur eine Activity in Frage, die durch einen Browser aufgerufen werden kann, um die von einem URI referenzierten Daten anzuzeigen (z. B. ein Bild).
- **CATEGORY\_LAUNCHER**  
Die ausführende Activity muss als Ausgangspunkt einer Task taugen (vgl. Abschnitt 5.3) und dem Anwendungsstarter des Systems bekannt sein.
- **CATEGORY\_DEFAULT**  
Ein als Parameter von **startActivity()** oder **startActivityForResult()** verwendeter Intent deklariert implizit **CATEGORY\_DEFAULT**. Diese Kategorie muss also *nicht* explizit gesetzt werden.

Zu diesen Kategorien sind **String**-Konstanten in der Klasse **Intent** definiert, z. B.:

```
public static final String CATEGORY_BROWSABLE = "android.intent.category.BROWSABLE";
```

Zur Vergabe einer Kategorie verwendet man die **Intent**-Methode **addCategory()**.

Aufgrund der bisher genannten Eigenschaften (Komponente, Aktion, Daten, Kategorie) kann Android die zu startende Komponente bestimmen. Die anschließend beschriebenen **Intent**-Bestandteile

tragen *nicht* zur Ermittlung der Zielkomponente bei, sondern enthalten Informationen für die ausführende Komponente (Extras) oder das Betriebssystem (Flags).

### 9.1.5 Extras

Über Schlüssel-Wert - Paare vereinbart man Informationen, welche die ausführende Komponente benötigt (z. B. den Empfänger, den Betreff und den Text beim Versenden einer Mail). Man kann die Informationen einzeln durch Aufrufe einer Überladung der **Intent**-Methode **putExtra()** (mit zwei Parametern für den Schlüssel und den Wert) vereinbaren, z. B.:

```
newContact.putExtra(ContactContract.Inserts.Insert.NAME, "Kurt Saar");
```

Von der Methode **putExtra()** existieren diverse Überladungen, die sich beim Datentyp für den zweiten Parameter unterscheiden. Man kann z.B. auch ein komplettes, serialisiertes Objekt per Intent übertragen (siehe Beispiel in Abschnitt 13.5).

Eine Alternative besteht darin, mehrere Schlüssel-Wert - Paare in einem **Bundle**-Objekt unterzubringen, das dem Intent durch einen Aufruf der Methode **putExtras()** hinzugefügt wird.

Bei der Benennung der Extras bietet sich oft ein Rückgriff auf Konstanten der Klasse **Intent** an, die am Namensanfang „EXTRA\_“ zu erkennen sind, z. B.

```
sendIntent.putExtra(Intent.EXTRA_TEXT, textMessage);
```

Damit steigt die Chance, dass der Auftragnehmer die angebotenen Extras richtig interpretiert.

Wenn die Daten von einem Content Provider aus der Android-Grundausstattung verwaltet werden, dann wird man bei der Suche nach den Schlüsselnamen in einer sogenannten *Kontrakt-Klasse* fünfzig (vgl. Abschnitt 14.1.9). Wird z. B. das Android-Adressbuch per **content-URI** angesprochen, dann finden sich die Schlüsselnamen in der internen Klasse **ContactContract.Insert**.<sup>1</sup>

Bei vielen Aktionen werden die beteiligten Daten (z. B. ein per **ACTION\_SEND** zu versendender Text) als Extras-Bestandteil in das **Intent**-Objekt eingefügt. Dann ist kein **Uri**-Objekt mit einer Datenadresse beteiligt, und es wird kein **setData()** - Aufruf benötigt (siehe Abschnitt 9.1.5). Bei der Aufnahme eines neuen Adressbucheintrags (Aktion **ACTION\_INSERT**) sind z. B. Extras mit den Merkmalen des neuen Eintrags *und* das **Uri**-Objekt zum Adressbuch beteiligt.

### 9.1.6 Flags

Mit der **Intent**-Methode **setFlags()** signalisiert man dem Betriebssystem gewünschte Ausführungsmodalitäten, z. B. mit dem (als Konstante in der Klasse **Intent** definierten) **FLAG\_ACTIVITY\_NO\_HISTORY** die Anweisung, eine gestartete Aktivität *nicht* in die Liste der zuletzt verwendeten Aktivitäten aufzunehmen:

```
newContact.setFlags(Intent.FLAG_ACTIVITY_NO_HISTORY);
```

## 9.2 Intent-Filter

Um für eine App-Komponente die Fähigkeiten zur Verarbeitung *impliziter* Intents zu deklarieren, erhält das zugehörige Element der App-Manifestdatei **AndroidManifest.xml** ein **intent-filter** - Kindelement (oder auch mehrere). Ein impliziter Intent wird einer Komponente nur dann zugestellt, wenn seine Anforderungen bzgl.

---

<sup>1</sup> Dies ist eine statische Mitgliedsklasse von **Intents**, die wiederum als statische Mitgliedsklasse in **ContactContract** definiert ist.

- der auszuführenden Aktion,
- der zu verarbeitender Daten
- und der Komponentenbeschaffenheit

von einem Intent-Filter der Komponente erfüllt werden.

Für Aktivitäten müssen die Intent-Filter in der App-Manifestdatei deklariert werden. Für einen Broadcast Receiver stellt die dynamische (De-)registrierung über die Methoden **registerReceiver()** bzw. **unregisterReceiver()** eine erlaubte und oft sinnvolle Alternative dar, damit die Empfangsbereitschaft z. B. nur dann besteht, wenn sich eine bestimmte Aktivität im Vordergrund befindet.

Ein Service sollte aus Sicherheitsgründen *nicht* über implizite Intents angesprochen werden, damit keine Vagheit bzgl. der tatsächlich ausgeführten Software auftritt, zumal der Benutzer nicht sieht, welcher Service startet. Seit Android 5 (API-Level 21) führt der Aufruf von **bindService()** mit einem impliziten Intent zu einem Ausnahmefehler. Dementsprechend sollten für einen Service auch keine Intent-Filter definiert werden.

Eine Komponente ohne Intent-Filter kann nur *explizite* Intents erhalten.

Die Verwendbarkeit einer Komponente durch *fremde* Apps wird durch das Attribut **android:exported** in ihrem Manifest-Element gesteuert, wobei folgende Regeln gelten:

- Sind Intent-Filter vorhanden, hat das Attribut **android:exported** die Voreinstellung **true**, sodass die Aktivität auch von fremden Apps genutzt werden kann.
- Sind *keine* Intent-Filter vorhanden, hat das Attribut **android:exported** die Voreinstellung **false**, sodass die Aktivität von fremden Apps *nicht* genutzt werden kann. Zu dieser Voreinstellung hat wohl die Annahme der Android-Designer geführt, dass der für explizite Intents erforderliche Klassenname in der Regel nicht öffentlich bekannt ist.

Über die später noch zu behandelnden Berechtigungen (engl.: *permissions*) lässt sich regeln, welche anderen Apps eine Komponente starten dürfen.<sup>1</sup>

### 9.2.1 Bestandteile eines Intent-Filters

Die **intent-filter** - Elemente zu einer Komponente definieren die akzeptablen impliziten Intents mit Kindelementen folgenden Typs:

- **action**

Für jede ausführbare Aktion (z. B. **ACTION\_VIEW**, **ACTION\_EDIT**) ist ein **action**-Element anzugeben, z. B.:

```
<intent-filter>
    <action android:name="android.intent.action.VIEW" />
    ...
</intent-filter>
```

Ist eine Komponente auch für die Verwendung durch fremde Apps gedacht, dann sollte nach Möglichkeit zur Spezifikation einer Aktion eine Definition aus einer SDK-Klasse verwendet werden. In der SDK-Klasse **Intent** ist z. B. die Aktion **ACTION\_VIEW** definiert:

```
public static final String ACTION_VIEW = "android.intent.action.VIEW";
```

Wie der obige Intent-Filter zeigt, ist zur Bezeichnung einer Aktion nicht eine Konstante (z. B. aus der Klasse **Intent**) anzugeben, sondern ein **String**-Literal (der vollständige Inhalt der Konstante), z. B.:

<sup>1</sup> <https://developer.android.com/guide/topics/permissions/defining>

Falsch: **android.content.Intent.ACTION\_VIEW**, **Intent.ACTION\_VIEW**  
 Richtig: **android.intent.action.VIEW**

- **data**

Welche Daten verarbeitet werden können, wird per URI und/oder MIME-Typ beschrieben.  
 Weil bei den Attributen des **data**-Elements etliche Details zu behandeln sind, geschieht dies in einem eigenen Abschnitt (mit der Nummer 9.2.2).

- **category**

Im Attribut **android:category** wird eine unterstützte Intent-Kategorie genannt. Ein potentieller Empfänger von impliziten Intents muss auf jeden Fall ein Element mit **CATEGORY\_DEFAULT** enthalten:

```
<category android:name="android.intent.category.DEFAULT"/>
```

Begründung: Ein als Parameter von **startActivity()** oder **startActivityForResult()** verwendetes Intent deklariert implizit **CATEGORY\_DEFAULT**. Damit eine Komponente als Empfänger für ein Intent in Frage kommt, müssen alle (expliziten oder impliziten) Kategorien des Intents in ihrem Intent-Filter enthalten sein.

Die Intent-Kategorien sind in der Klasse **Intent** (Paket: **android.content**) als Konstanten deklariert, z. B.:

```
public static final String CATEGORY_DEFAULT =
    "android.intent.category.DEFAULT";
```

In einem Intent-Filter wird dem **name**-Attribut des **category**-Elements *nicht* der Name, sondern der Wert der Konstanten (als **String**-Literal) zugewiesen.

Damit die Hauptaktivität einer App vom Anwendungsstarter angeboten wird, ist in einem Intent-Filter neben der **ACTION\_MAIN** auch die **CATEGORY\_LAUNCHER** anzugeben:

```
<intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
```

In einem Intent-Filter sind auch mehrere Elemente einer Sorte erlaubt, wobei die Komponente dann alle Kombinationen beherrschen muss (z. B. **ACTION\_EDIT** mit allen deklarierten MIME-Typen).

Anschließend wird die App-Manifestdatei zum Beispielprogramm **Note Pad** auszugsweise (eingeschränkt auf die Activity **NoteEditor**) wiedergegeben. Dieses Beispielprogramm wird von der Firma Google in ihrer Online-Dokumentation zur Android-Programmierung beschrieben:<sup>1</sup>

---

<sup>1</sup> <https://developer.android.com/reference/android/content/Intent>

```

<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.android.notepad">
    <application android:icon="@drawable/app_notes"
        android:label="@string/app_name">

        <activity class=".NoteEditor" android:label="@string/title_note">
            <intent-filter>
                <action android:name="android.intent.action.VIEW" />
                <action android:name="android.intent.action.EDIT" />
                <category android:name="android.intent.category.DEFAULT" />
                <data android:mimeType="vnd.android.cursor.item/vnd.google.note" />
            </intent-filter>

            <intent-filter>
                <action android:name="android.intent.action.INSERT" />
                <category android:name="android.intent.category.DEFAULT" />
                <data android:mimeType="vnd.android.cursor.dir/vnd.google.note" />
            </intent-filter>
        </activity>
    </application>
</manifest>

```

Es sind zwei Intent-Filter vorhanden, die sich hinsichtlich der unterstützten Aktionen und hinsichtlich der jeweils zulässigen Daten (per MIME-Typ beschrieben) unterscheiden.

### 9.2.2 data-Element

Ein **data**-Kindelement zu einem **intent-filter** - Element beschreibt über einen MIME-Typ (Attribut **android:mimeType**) und/oder über einen URI (restliche Attribute), welche Daten die Komponente verarbeiten kann:<sup>1</sup>

```

<data
    android:scheme="string"
    android:host="string"
    android:port="string"
    android:path="string"
    android:pathPrefix="string"
    android:pathPattern="string"
    android:mimeType="string" />

```

Erläuterung der Attribute:

- **android:scheme**

Das Schema gibt den Typ des URIs an und diktiert, welche Bestandteile zu folgen haben. Ohne Schema sind alle anderen URI-Attribute bedeutungslos. Häufig ist mit dem Schema ein Übertragungsprotokoll assoziiert (z. B. **https**). Das für Android spezifische Schema **content** wird verwendet für den Zugriff auf Daten, die durch einen lokal installierten Content Provider verwaltet werden (siehe Kapitel 14). Weil in Android die Schemaidentifikation von der Groß-/Kleinschreibung abhängt, sollten für das Schema ausschließlich Kleinbuchstaben verwendet werden.

Ist ein MIME-Typ (z. B. **video/\***) deklariert (siehe unten), aber kein Schema, dann sind die Schemata **content** und **file** implizit gesetzt. Android geht davon aus, dass Daten des angegebenen Typs verarbeitet werden können, die von einem lokalen Content Provider oder aus einer Datei auf dem lokalen Gerät stammen.

---

<sup>1</sup> <https://developer.android.com/guide/topics/manifest/data-element>

- **android:host**

Es ist der Name eines Servers oder (beim Schema **content**) der Name eines lokalen Content Providers anzugeben. Die **host**-Angabe ist nur dann wirksam, wenn auch das **scheme**-Attribut vorhanden ist. Es wird empfohlen, beim **host**-Attribut ausschließlich Kleinbuchstaben zu verwenden.

- **android:port**

Über eine Portnummer wird ein spezieller Dienst auf einem Server angesprochen. Die Angabe einer Portnummer ist nur dann wirksam, wenn auch ein Schema und ein Server angegeben werden.

- **android:path**

Ein komplett angegebener Pfad muss vollständig mit dem Pfad im Intent übereinstimmen. Bei einem Content Provider - URI (Schema **content**), ist als Pfad ein Tabellenname anzugeben.

- **android:pathPrefix**

Dieses Attribut ist zu verwenden, wenn der Filter zu allen Intents passen soll, deren Pfad mit einer bestimmten Zeichenfolge beginnt.

- **android:pathPattern**

Dieses Attribut ist zu verwenden, wenn beim Vergleich mit dem Pfad des **Uri**-Objekts im Intent Jokerzeichen zu verwenden sind:

- \* Steht für beliebig viele Wiederholungen des unmittelbar vorangegangenen Zeichens
  - .\* Steht für eine beliebige Zeichenfolge, z. B.:
- ```
<data android:scheme="http" android:host="www.google.com"
      android:pathPattern="/calendar/hosted/.*/event" />
```

- **android:mimeType**

Bei der Angabe eines MIME-Typs (z. B. **image/png**) kann der Subtyp durch das Jokerzeichen **\*** ersetzt werden (z. B. **image/\***). Es wird empfohlen, den MIME-Typ in Kleinbuchstaben zu schreiben.

Ein **intent-filter** - Element darf mehrere **data**-Kindelemente enthalten.

Am häufigsten kommen Intent-Filter vor, die zur Datenspezifikation einen MIME-Type angeben (oder mehrere), aber keinen URI (siehe Beispiel am Ende von Abschnitt 9.2.1). Die zu verarbeitenden Daten stammen nämlich in der Regel von einem (lokal installierten) Content Provider, und Android geht bei jeder Anwendungskomponente davon aus, dass sie Daten von einem Content Provider beziehen kann. Folglich ist keine URI-Spezifikation erforderlich.

### **9.3 Eigene Activity über einen expliziten Intent starten**

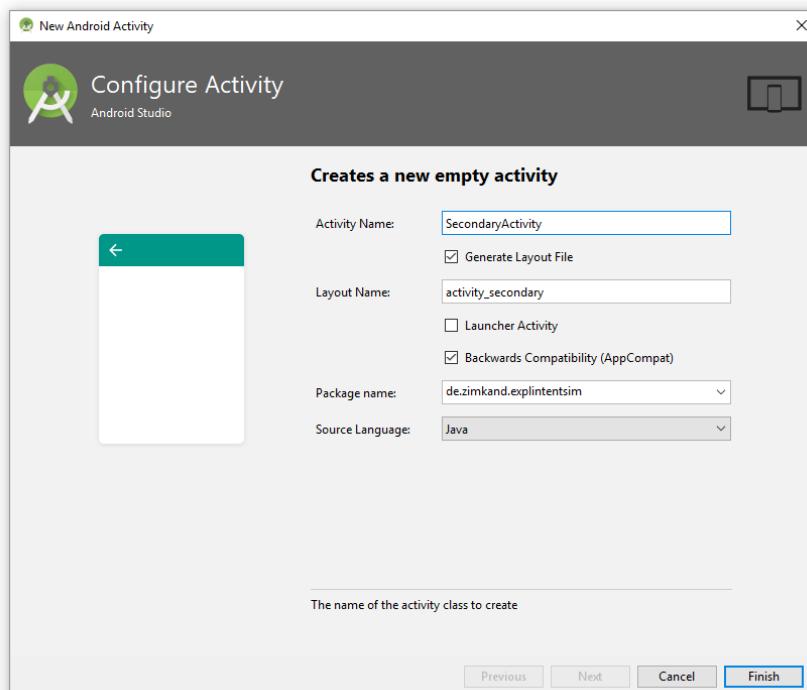
Über einen expliziten Intent sollten ausschließlich *eigene* Anwendungskomponenten gestartet werden. Man sollte sich nie darauf verlassen, dass eine fremde App installiert ist und eine Komponente mit den vermuteten Namen besitzt. Fremde Komponenten sollten über einen impliziten Intent mit Spezifikation von Aktion und Daten angesprochen werden.

Wir starten ein neues AS-Projekt mit einer leeren Aktivität, die vom Assistenten wie gewohnt in der Manifestdatei als Start- bzw. Hauptaktivität der Anwendung deklariert wird, z. B.:

```
<activity
    android:name="de.zimkand.explintentsim.MainActivity"
    android:label="@string/app_name" >
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

Um in einem AS-Projekt eine zusätzliche Aktivität anzulegen, kann man so vorgehen:

- Kontextmenü zum Paket der Anwendung im **Project**-Fenster öffnen
- Auswahl von **New > Activity > Empty Activity**
- im folgenden Fenster einen **Activity Name** wählen



und auf **Finish** klicken

Das Android Studio legt eine Klasse für die neue Aktivität an und trägt diese in die Manifestdatei der Anwendung ein, z. B.:

```

<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="de.zimkand.explintentsim">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme">
        <activity
            android:name=".MainActivity"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".SecondaryActivity"></activity>
    </application>
</manifest>

```

Obligatorisch ist nur der Name der realisierenden Klasse. Zusätzlich kann z. B. ein in der App Bar anzugebender Titel angegeben werden:

```

<activity
    android:name=".SecondaryActivity"
    android:label="@string/title_activity_secondary" >
</activity>

```

### 9.3.1 Einfacher Aktivitätsstart

Aus einer Aktivität (z. B. aus der Startaktivität) eine sekundäre Aktivität derselben App über ein explizites **Intent**-Objekt zu starten, ist eine simple Angelegenheit. Man erstellt ein **Intent**-Objekt über die folgende Konstruktor-Überladung

**public Intent (Context packageContext, Class<?> class)**

mit den beiden folgenden Parametertypen:

- **Context**

Hier gibt man in der Regel als Aktualparameter über das Schlüsselwort **this** das handelnde **Activity**-Objekt an.

- **Class<?>**

Hier gibt man das **Class**-Objekt zur Klasse an, welche die sekundäre Aktivität realisiert. Eine einfache Möglichkeit, die benötigte Referenz zu gewinnen, besteht darin, auf den Klassennamen durch einen Punkt getrennt das klein geschriebene Schlüsselwort **class** folgen zu lassen.

Anschließend startet man die sekundäre Aktivität mit der Methode **startActivity()**, die das **Intent**-Objekt als Aktualparameter erhält.

Im folgenden Code-Segment wird eine sekundäre Aktivität in der **onClick()** - Ereignismethode zu einem Befehlsschalter gestartet:

```
Button btnStartSecAct = findViewById(R.id.startSecAct);
btnStartSecAct.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        Intent secAct = new Intent(MainActivity.this, SecondaryActivity.class);
        startActivity(secAct);
    }
});
```

Weil im Beispiel ein Objekt einer anonymen Klasse als Click-Ereignisempfänger am Werk ist, wird das **Activity**-Objekt über den Namen der umgebenden Klasse plus Schlüsselwort **this** angesprochen.

Um ausnahmsweise eine Aktivität einer *fremden* Anwendung über ein explizites **Intent**-Objekt zu starten, kann man so vorgehen:

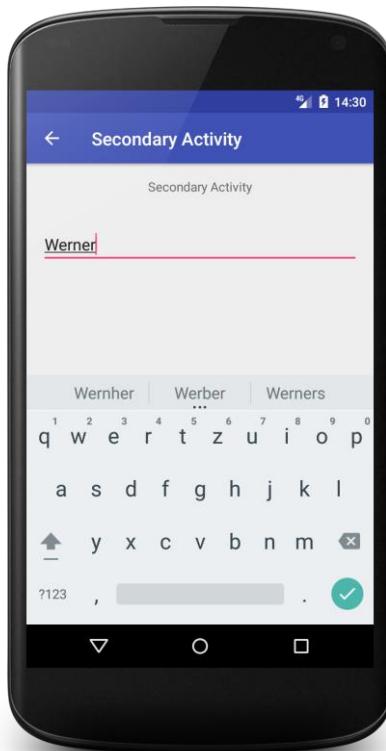
- Leeres **Intent**-Objekt erstellen:  
`Intent intent = new Intent();`
- Paket (Anwendung) und Aktivitätsklasse über ein Objekt der Klasse **ComponentName** und die Methode **setComponent()** festlegen:  
`String paket = "de.zimkand.brdemo";
String klasse = "de.zimkand.brdemo.MainActivity";
intent.setComponent(new ComponentName(paket, klasse));`
- Aktivität starten:  
`startActivity(intent);`

Ein Aufruf von **startActivity()** kehrt sofort zurück, und in der **onClick()** - Methode des Beispiels könnten weitere Anweisungen stehen. Unabhängig davon ist die primäre Aktivität mittlerweile im Back Stack der aktiven Task gelandet (vgl. Abschnitt 5.3). Nach Beendigung der Methode **onClick()** erhält die primäre Aktivität die in Kapitel 6 beschriebenen Lebenszyklus-Methodenaufrufe (**onPause()**, **onStop()**).

### 9.3.2 Navigation innerhalb der eigenen App

Besteht eine App aus mehreren, hierarchisch angeordneten Aktivitäten, dann sollte die von Android verwaltete Rückwärtssnavigation ergänzt werden durch einen Aufwärtsschalter in der App Bar von untergeordneten Aktivitäten. Er orientiert sich nicht an der zeitlichen Sequenz von Aktivitäten, sondern bewirkt die sofortige Rückkehr zur vorgeordneten (elterlichen) Aktivität.

Damit das aktuelle, sehr simple Beispiel zumindest ansatzweise den Unterschied zwischen der Rückwärts- und die Aufwärtssnavigation demonstrieren kann, wird in die sekundäre Aktivität noch ein **EditText**-Steuerelement eingebaut:



Seine Nutzung ruft die Bildschirmtastatur hervor, sodass die generelle Rückwärtstaste sich nicht unmittelbar auf die Aktivitätensequenz der aktuellen Aufgabe bezieht (vgl. Abschnitt 5.3), sondern zunächst die virtuelle Tastatur beendet. Vergleichbare „Nebentätigkeiten“ der Rückwärtstaste sind übrigens:

- Beendigung von schwebenden Fenstern (Popup-Menüs, Dialogboxen)
- Beendigung der Context Action Bar (vgl. Abschnitt 8.6.2.2)

In dieser Lage bewirkt der Aufwärtsschalter eine sofortige Rückkehr zur festgelegten elterlichen Aktivität.

Diese Navigationsoption ist z. B. auch dann sinnvoll, wenn eine Aktivität (z. B. zuständig für die Einstellungen einer App) aus verschiedenen anderen Aktivitäten erreicht werden kann und die Aufwärtsnavigation zuverlässig zur Startaktivität führen soll, während die Rückwärtsnavigation den Benutzer zur vorherigen Aktivität bringt.

Um die Aufwärtsnavigation über einen Linkspfeil am linken Rand der App Bar (siehe obiges Bildschirmfoto) zu ermöglichen, ist für die betroffene Aktivität ein Eintrag in der Manifestdatei vorzunehmen. Ab Android 4.1 (API Level 16) ist das **activity**-Element das Attribut **android:parentActivityName** einzufügen, z. B.:

```
<activity
    android:name=".SecondaryActivity"
    android:label="@string/title_activity_secondary"
    android:parentActivityName=".MainActivity">
    ...
</activity>
```

Bei Android-Version  $\leq 4.0$  muss ...

- die Support Library integriert werden, was wir routinemäßig tun,
- und ein geeignetes **meta-data** - Element in das **activity**-Element aufgenommen werden.

Weil wir alle Android-Versionen ab 4.0.3 (API-Level 15) unterstützen wollen, verwenden wir im Beispiel das folgende **activity**-Element:

```

<activity
    android:name=".SecondaryActivity"
    android:label="@string/title_activity_secondary"
    android:parentActivityName=".MainActivity">
    ...
    <meta-data
        android:name="android.support.PARENT_ACTIVITY"
        android:value=".MainActivity" />
</activity>

```

### 9.3.3 Extradaten übergeben und in der sekundären Aktivität verwenden

Soll die sekundäre Aktivität Daten zur näheren Auftragsbeschreibung erhalten, bringt man diese mit der **Intent**-Methode **putExtra()** als Schlüssel-Wert - Paare im **Intent**-Objekt unter (vgl. Abschnitt 9.1.5), z. B.:

```

static final String EXTRA_MESS = "mess";
...
Intent secAct = new Intent(MainActivity.this, SecondaryActivity.class);
secAct.putExtra(EXTRA_MESS, getString(R.string.send_string));
startActivity(secAct);

```

Die aufgerufene Aktivität kann mit der Methode **getIntent()** auf das beim Start erhaltene **Intent**-Objekt zugreifen, was hier in der Methode **onCreate()** geschieht:

```

package de.zimkand.explintentext;

import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.widget.TextView;

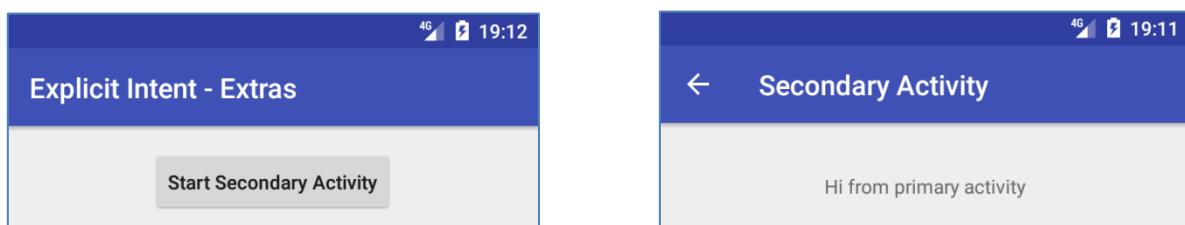
public class SecondaryActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_secondary);
        TextView etMess = findViewById(R.id.etMess);

        Bundle extras = getIntent().getExtras();
        String def = extras.getString(MainActivity.EXTRA_MESS);
        if (def.length() != 0)
            etMess.setText(def);
    }
}

```

Mit der Methode **getExtras()** gewinnt man aus dem **Intent**-Objekt ein **Bundle**-Objekt mit den Extradaten, aus dem sich die mit einem Schlüssel verbundenen Werte entnehmen lassen. Offensichtlich benötigt der Programmierer der sekundären Aktivität eine genaue Beschreibung des erhaltenen **Intent**-Objekts.

Im Beispiel zeigt die sekundäre Aktivität die übergebene Zeichenfolge an:



### 9.3.4 Rückmeldung der sekundären Aktivität auswerten

Soll die sekundäre Aktivität ein Ergebnis zurückmelden, muss man sie mit der Methode **startActivityForResult()** starten, die als zusätzlichen Parameter im Vergleich zur oben verwendeten Methode **startActivity()** einen **int**-wertigen Parameter, den sogenannten *Request-Code*, besitzt, z. B.:

```
private int REQUEST_CODE_EXPLICIT = 12;
...
startActivityForResult(secAct, REQUEST_CODE_EXPLICIT);
```

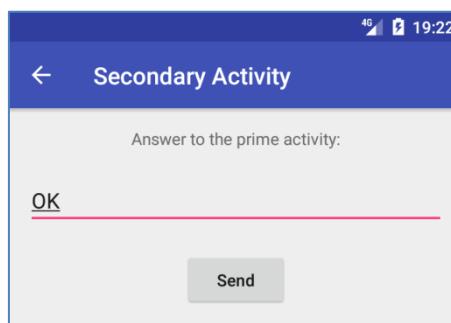
Außerdem muss der Auftraggeber die **Activity**-Methode **onActivityResult()** überschreiben, die bei Beendigung der sekundären Aktivität aufgerufen wird. Sie erhält per Parameter folgende Informationen:

- Den Request-Code, mit dem die sekundäre Aktivität gestartet worden war  
Eine primäre Aktivität, die mehrere sekundäre Aktivitäten startet, kann so die Rückmeldungen unterscheiden.
- Den **int**-wertigen Result-Code, mit dem die sekundäre Aktivität den Ausgang ihrer Bemühungen kategorisiert
- Ein **Intent**-Objekt  
Android benutzt auch für den rückwärts gerichteten Informationstransfer ein **Intent**-Objekt.

Wenn die primäre Aktivität im Beispiel von der erfolgreichen Tätigkeit der sekundären Aktivität erfährt, entnimmt sie aus dem erhaltenen **Intent**-Objekt über die Methode **getStringExtra()** den Wert zum Schlüssel, der im statischen Feld **EXTRA\_MESSAGE** definiert ist:

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent intent) {
    if (requestCode == REQUEST_CODE_EXPLICIT) {
        TextView tv = findViewById(R.id.tvAnswer);
        if (resultCode == RESULT_OK)
            tv.setText(getString(R.string.answer) + " " +
                    intent.getStringExtra(EXTRA_MESSAGE));
        else
            tv.setText(getString(R.string.answer) + " " +
                    getString(R.string.neg_answer));
    }
}
```

Um eine Benutzeräußerung entgegennehmen zu können, erhält die sekundäre Aktivität ein erweitertes Layout mit Texteingabefeld und Befehlsschalter:



Es bleibt noch zu beschreiben, wie die sekundäre Aktivität den Result-Code und das **Intent**-Retour-Objekt erstellt. Im Beispiel geschieht dies in der **onClick()** - Methode zum Befehlsschalter:

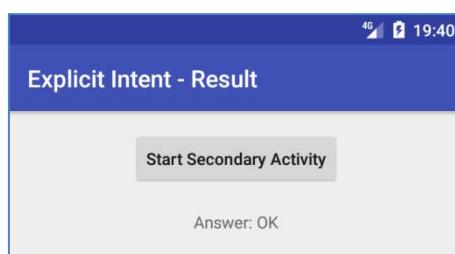
```

Button btnSend = findViewById(R.id.send);
btnSend.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View arg0) {
        Intent intent = new Intent();
        intent.putExtra(MainActivity.EXTRA_MESS, etMess.getText().toString());
        setResult(RESULT_OK, intent);
        finish();
    }
});
```

In ein zunächst leeres **Intent**-Objekt wird mit **putExtra()** ein Schlüssel-Wert - Paar eingefügt. Durch die Methode **setResult()** wird ...

- mit dem ersten Parameter der Result-Code festgelegt
- und mit dem zweiten Parameter das **Intent**-Objekt in das Rücksendepaket gesteckt.

Anschließend beendet sich die sekundäre Aktivität mit der Methode **finish()**, und die primäre Aktivität führt ihre **onActivityResult()** - Methode aus:



Das im aktuellen Abschnitt beschriebene Beispielprogramm ist im folgenden Ordner zu finden:

**...\\BspUeb\\Intent\\Explizit\\Result**

## 9.4 Fremde Aktivitäten über implizite Intents starten

Schon die Android-Grundausstattung bietet zahlreiche Helfer, die man in eigenen Apps nutzen kann. Auf der folgenden Webseite

<https://developer.android.com/guide/components/intents-common>

finden sich Lösungen für viele Standardaufgaben (z. B. Foto aufnehmen, E-Mail versenden, Position auf einer Landkarte anzeigen).

### 9.4.1 Intent-Auflösung

Bei einem impliziten Intent ermittelt Android (durch Befragen seines Package-Managers), welche Komponenten als Empfänger in Frage kommen. Ein Intent-Filter (siehe Abschnitt 9.2) muss drei Tests bestehen, bevor die zugehörige Komponente ein **Intent**-Objekt zur Erledigung bzw. zur Kenntnisnahme erhalten kann:

- Action-Test  
Die Aktion des **Intent**-Objekts muss im Intent-Filter aufgelistet sein.
- Category-Test  
*Jede* Kategorie des **Intent**-Objekts muss im Intent-Filter aufgelistet sein.
- Data-Test  
Der Intent-Filter muss die Datenspezifikation aus dem Intent in seiner Unterstützungsliste enthalten.

Der Data-Test ist komplex, weil die Datenspezifikation per **Uri**-Objekt und/oder per MIME-Typ (aus dem **Uri**-Objekt erschlossen oder explizit deklariert) erfolgend kann. Um das Regelwerk übersichtlich zu halten, zerlegen wir den Data-Test in einen URI- und einen MIME-Test.

So wird festgestellt, ob ein Intent-Filter den URI-Test für einen Intent besteht:

- Enthält ein Intent-Filter nur ein Schema, dann passen alle Intent-URIs mit diesem Schema (unabhängig von Host und Pfad).
- Enthält ein Intent-Filter ein Schema und einen Host, dann passen alle URIs mit diesem Schema und diesem Host (unabhängig vom Pfad).
- Enthält ein Intent-Filter ein Schema, einen Host und einen Pfad, dann passt ein URI nur dann, wenn Schema, Host und Pfad übereinstimmen.
- Enthält ein Intent-Filter einen Pfad mit Wildcard-Stern (\*), dann wird der Pfad-Test entsprechend liberalisiert.

Enthält ein Intent-Filter eine MIME-Angabe mit Wildcard-Stern (\*), dann wird der MIME-Test für ein **Intent**-Objekt entsprechend liberalisiert, z. B.:

```
<data android:mimeType="image/*" />
```

Regeln für den gesamten Data-Test, den ein Intent-Filter bestehen muss, damit die zugehörige Aktivität als Auftragnehmer bzw. der zugehörige Broadcast Receiver als Empfänger für einen Intent in Frage kommt:

- Bei einem Intent ohne Datenspezifikation besteht ein Filter den Data-Test genau dann, wenn er ebenfalls keine Datenspezifikation enthält. Ohne URI und MIME-Typ kommt z. B. ein Intent zur Definition eines Alarms aus, der zu einer bestimmten Zeit mit einer bestimmten Nachricht ausgeführt werden soll:<sup>1</sup>

```
public void createAlarm(String message, int hour, int minutes) {
    Intent intent = new Intent(AlarmClock.ACTION_SET_ALARM)
        .putExtra(AlarmClock.EXTRA_MESSAGE, message)
        .putExtra(AlarmClock.EXTRA_HOUR, hour)
        .putExtra(AlarmClock.EXTRA_MINUTES, minutes);
    if (intent.resolveActivity(getApplicationContext()) != null) {
        startActivity(intent);
    }
}
```

Man verwendet die Aktion **android.intent.action.SET\_ALARM** (im Beispiel über eine Konstante der Klasse **AlarmClock** angesprochen) und ergänzt den Zeitpunkt sowie die Nachricht über Extras. Ein zugehöriger Filter kann so aussehen:

```
<intent-filter>
    <action android:name="android.intent.action.SET_ALARM" />
    <category android:name="android.intent.category.DEFAULT" />
</intent-filter>
```

- Enthält ein Intent einen URI, aber keinen MIME-Typ (weder explizit definiert, noch aus dem URI erschlossen), dann ist der Data-Test genau dann erfolgreich, wenn der URI-Test bestanden wird, und im Filter kein MIME-Typ definiert ist. Im folgenden Beispiel wird Android beauftragt, eine Aktivität zum Formulieren und Versenden einer Mail zu starten:<sup>2</sup>

---

<sup>1</sup> Das Beispiel stammt von der Seite: <https://developer.android.com/guide/components/intents-filters>

<sup>2</sup> Das Beispiel stammt von der Seite: <https://developer.android.com/guide/components/intents-common>

```
private void composeEmail(String[] addresses, String subject) {
    Intent intent = new Intent(Intent.ACTION_SENDTO);
    intent.setData(Uri.parse("mailto:"));
    intent.putExtra(Intent.EXTRA_EMAIL, addresses);
    intent.putExtra(Intent.EXTRA_SUBJECT, subject);
    if (intent.resolveActivity(getApplicationContext()) != null)
        startActivity(intent);
}
```

Hier ist ein passender Intent-Filter zu sehen:

```
<intent-filter>
    <action android:name="android.intent.action.SENDTO" />
    <data android:scheme="mailto" />
    <category android:name="android.intent.category.DEFAULT" />
</intent-filter>
```

- Enthält ein Intent einen MIME-Typ, aber keine URI-Bestandteile, dann ist der Data-Test genau dann erfolgreich, wenn der MIME-Test bestanden wird, und im Filter keine URI-Bestandteile definiert sind. Mit Hilfe des folgenden Intent-Objekts (ohne URI, mit MIME-Typ) wird Android aufgefordert, einen Text durch eine passende Aktivität versenden zu lassen:

```
Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.putExtra(Intent.EXTRA_TEXT, "Zu versendender Text");
sendIntent.setType("text/plain");
if (sendIntent.resolveActivity(getApplicationContext()) != null)
    startActivity(sendIntent);
```

Hier ist ein passender Intent-Filter zu sehen:

```
<intent-filter>
    <action android:name="android.intent.action.SEND" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="text/plain" />
</intent-filter>
```

- Enthält ein Intent einen MIME-Typ und einen URI, dann ist der Data-Test genau dann bestanden, wenn sowohl der MIME- als auch der URI-Test erfolgreich verlaufen. Enthält ein Intent einen MIME-Typ und als Schema entweder **content** oder **file**, dann ist der URI-Test auch dann bestanden, wenn der Filter nur einen MIME-Typ angibt. Android geht davon aus, dass eine Komponente Daten von passendem Typ von einem (auf demselben Gerät installierten) Content Provider beziehen oder aus einer lokalen Datei lesen kann.

#### 9.4.2 Aktivitätsstart ohne Rückmeldung

Im ersten Beispiel lassen wir uns die Kontakte im Android-Adressbuch auflisten. Wir können davon ausgehen, dass die auf jedem Android-Gerät vorhandene Kontakte-App unseren Auftrag erledigen kann und über entsprechend konfigurierte Intent-Filter verfügt. Allerdings müssen wir nicht etwa diese App (und ihre Activities) kennen, sondern lediglich ein implizites Intent-Objekt mit den folgenden Informationen erstellen:

- Aktion

Die in der Klasse **Intent** definierte statische **String**-Konstante **ACTION\_VIEW** beschreibt die gewünschte Aktion.

- Daten

Zur Beschreibung der Daten verwenden wir einen URI mit folgenden Bestandteilen:

Schema	<b>content</b>
Host	<b>com.android.contacts</b>
Pfad	<b>contacts</b>

Damit wird der systemseitige Content Provider zur Verwaltung der Kontakte angesprochen. Man kann das benötigte Uri-Objekt aus einer Zeichenfolge aufbauen,

```
Uri.parse("content://com.android.contacts/contacts")
```

oder ein äquivalentes vordefiniertes Uri-Objekt verwenden (vgl. Abschnitt 9.1.3)

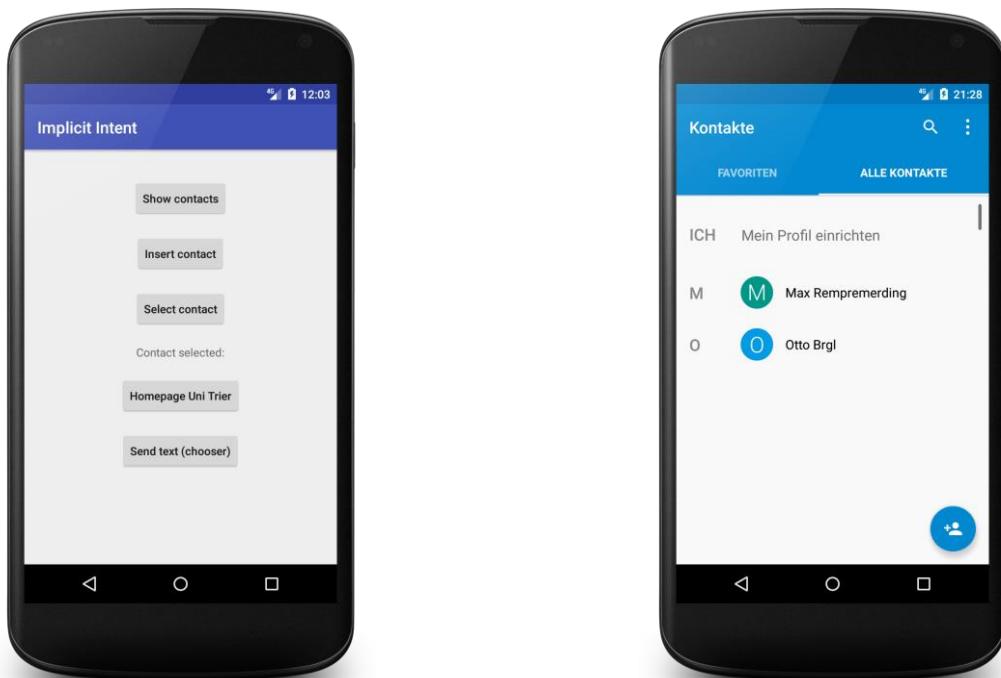
```
ContactsContract.Contacts.CONTENT_URI
```

In einem einfachen Beispielprogramm erstellen wir das **Intent**-Objekt in der **onClick()** - Methode zu einem Befehlsschalter:

```
Button btnViewContacts = findViewById(R.id.viewContacts);
btnViewContacts.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        Intent abook = new Intent(Intent.ACTION_VIEW,
                ContactsContract.Contacts.CONTENT_URI);
        if (abook.resolveActivity(getApplicationContext()) != null)
            startActivity(abook);
    }
});
```

Das **Intent**-Objekt dient als Parameter der Methode **startActivity()**, die Android auffordert, eine zum Auftrag passende Aktivität zu starten.

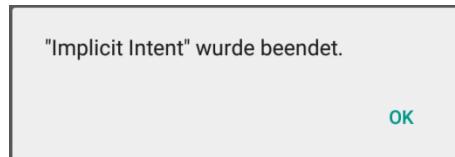
Kurze Zeit nach einem Klick auf den initierenden Schalter des Beispielprogramms erscheint die Kontakte-App mit der gewünschten Anzeige, z. B.:



Statt einen **Intent**-Konstruktor mit Parametern für Aktion und Daten zu verwenden, kann man auch ein leeres **Intent**-Objekt erzeugen und es anschließend über Methodenaufrufe ausstatten, z. B.:

```
Intent abook = new Intent();
abook.setAction(Intent.ACTION_VIEW);
abook.setData(ContactsContract.Contacts.CONTENT_URI);
```

Existiert zu einem impliziten Intent auf dem aktuellen Gerät keine passende Aktivität, wird der allzu optimistische Auftraggeber von Android beendet, z. B.:



Daher ist es ratsam, vor dem Aufruf von `startActivity()` mit der Intent-Methode `resolveActivity()` zu überprüfen, ob der Package-Manager zum aktuellen Kontext mindestens eine passende Aktivität kennt, z. B.:

```
Intent powerContacts = new Intent(Intent.ACTION_POWER_CONNECTED,
                                    ContactsContract.Contacts.CONTENT_URI);
if (powerContacts.resolveActivity(getApplicationContext()) != null)
    startActivity(powerContacts);
```

Das als Parameter für `resolveActivity()` benötigte **PackageManager** - Objekt ermittelt man mit der **Context**-Methode `getPackageManager()`.

Im folgenden Beispiel wird das Android-Adressbuch um einen neuen Kontakt erweitert, wobei ein **Intent**-Objekt mit Extradaten zum Einsatz kommt:

```
Button btnInsertContact = findViewById(R.id.insertContact);
btnInsertContact.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        Intent newContact = new Intent(Intent.ACTION_INSERT,
  ContactsContract.Contacts.CONTENT_URI);
        newContact.putExtra(ContactsContract.Intents.Insert.NAME, "Kurt Saar");
        if (newContact.resolveActivity(getApplicationContext()) != null)
            startActivity(newContact);
    }
});
```

Das **Intent**-Objekt verlangt die **INSERT**-Aktion, die auf das Android-Adressbuch angewendet werden soll, und ist damit noch nicht vollständig spezifiziert. Die Datenbankfelder des neuen Kontakts werden über `putExtra()` - Aufrufe mit Werten versorgt. Die erforderlichen Schlüsselnamen (die Namen der Datenbankfelder) können über statische Variablen der (doppelt) eingeschachtelten Klasse `ContactsContract.Intents.Insert` angesprochen werden.

Im nächsten Beispiel wird ein Browser mit der Anzeige einer Webseite beauftragt:

```
Button btnViewUT = findViewById(R.id.viewUT);
btnViewUT.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        Intent viewUT = new Intent(Intent.ACTION_VIEW,
                                   Uri.parse("https://www.uni-trier.de/"));
        if (viewUT.resolveActivity(getApplicationContext()) != null)
            startActivity(viewUT);
    }
});
```

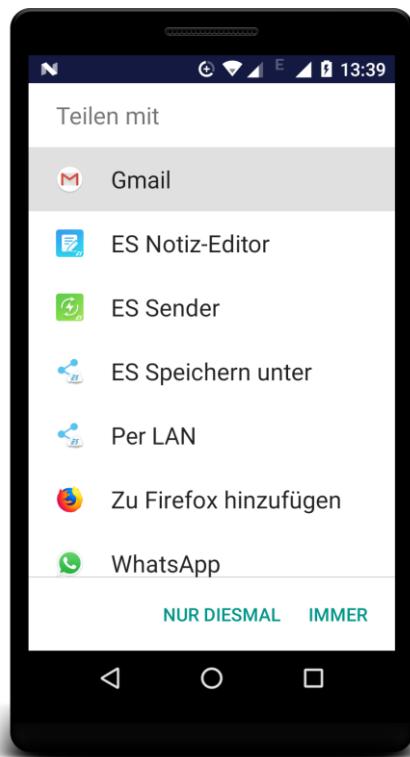
Das im **Intent**-Konstruktor verwendete **Uri**-Objekt wird mit der statischen **Uri**-Methode `parse()` aus einer Zeichenfolge erstellt.

### 9.4.3 Wahl zwischen alternativen Intent-Auftragnehmern

Findet der Package-Manager *mehrere* Activities, die sich als Auftragsnehmer für geeignet halten, darf der Benutzer entscheiden. Durch die folgende Click-Behandlungsmethode wird ein Text versendet:

```
public void onClick(View v) {
    Intent sendIntent = new Intent();
    sendIntent.setAction(Intent.ACTION_SEND);
    sendIntent.putExtra(Intent.EXTRA_TEXT, "Please notice this message.");
    sendIntent.setType("text/plain");
    if (sendIntent.resolveActivity(getApplicationContext()) != null)
        startActivity(sendIntent);
}
```

Sind mehrere Aktivitäten mit passendem Intent-Filter vorhanden, überlässt Android dem Benutzer die Wahl:

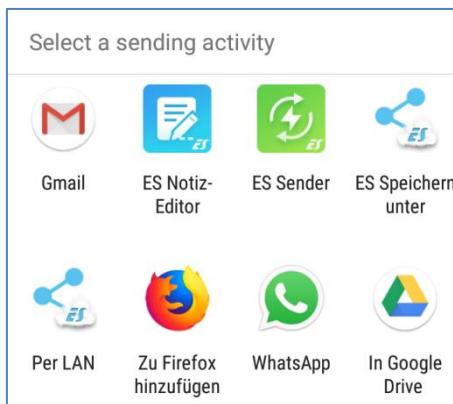


Hat sich der Benutzer für eine dauerhafte Zuordnung entschieden, wird bei nächster Gelegenheit die zuständige Aktivität automatisch verwendet.

Soll dem Benutzer zu einem **Intent**-Objekt unabhängig von einer früheren Entscheidung für eine standardmäßig zu verwendende Aktivität auf jeden Fall ein Auswahldialog vorgelegt werden, erstellt man zum ursprünglichen **Intent**-Objekt ein neues mit der **Intent**-Methode **createChooser()** und startet dieses, z. B.:

```
public void onClick(View v) {
    Intent sendIntent = new Intent();
    sendIntent.setAction(Intent.ACTION_SEND);
    sendIntent.putExtra(Intent.EXTRA_TEXT, "Please notice this message.");
    sendIntent.setType("text/plain");
    startActivityForResult(Intent.createChooser(sendIntent, getText(R.string.title_chooser)));
}
```

Im zweiten **createChooser()** - Parameter wird ein Titel für den folgenden, garantiert erscheinenden Nachfragedialog festgelegt:

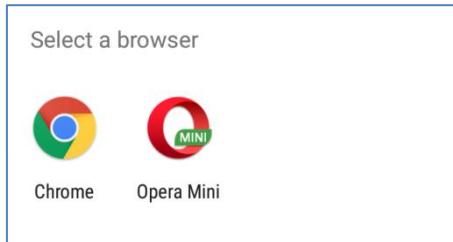


Wenn kein potentieller Intent-Auftragnehmer verfügbar ist, kommt es abweichend vom `startActivity()`-Aufruf *ohne* Chooser-Beteiligung *nicht* zur Beendigung der App mit einem Ausnahmefehler, sondern der Benutzer wird entsprechend informiert. Folglich ist *keine* Absicherung über die in Abschnitt 9.4.2 beschriebene **PackageManager**-Methode `resolveActivity()` erforderlich.

Es ist es mir übrigens *nicht* gelungen, für die Anzeige einer Webseite durch einen Browser auf analoge Weise

```
Intent viewUT = new Intent(Intent.ACTION_VIEW,
                           Uri.parse("http://www.uni-trier.de/"));
Intent chooser = Intent.createChooser(viewUT, getString(R.string.title_chooser));
startActivity(chooser);
```

den Auftritt eines Auswahldialogs zu erzwingen. Unter Android 7.1 und 8.1 ist der folgende Dialog



nur dann erschienen, wenn der Benutzer noch *keinen* voreingestellten Browser gewählt hatte. Ansonsten wurde trotz Chooser-Einsatz ohne Nachfrage der Standard-Browser verwendet.

#### 9.4.4 Rückmeldung der sekundären Aktivität auswerten

Die in Abschnitt 9.3.4 vorgeführte Rückmeldetechnik unter Verwendung der Methode `startActivityForResult()` klappt auch bei fremden Aktivitäten. I.A. kennt man die per Intent-Auflösung beteiligten Activities und deren Kommunikationsgepflogenheiten nicht, doch sollte z. B. ein Vergleich des Result-Codes mit der Konstanten **RESULT\_OK** verlässlich sein. Häufig (z. B. bei Verwendung von systemseitigen Content Providern) ist aber das Rückmeldeverhalten gut dokumentiert, sodass eine Weiterverarbeitung der erhaltenen Daten (z. B. Kontaktdaten) möglich ist. Details werden wir im Zusammenhang mit Datenbank- bzw. Abfragetechniken kennen lernen (siehe Kapitel 13).

Im folgenden Beispiel wird die Auswahl-Aktion angefordert, und die zu verarbeitenden Daten werden über das **Uri**-Objekt zum Android-Adressbuch spezifiziert:

```

Button btnSelectContact = findViewById(R.id.selectContact);
btnSelectContact.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        Intent intent = new Intent(Intent.ACTION_PICK,
                                    ContactsContract.Contacts.CONTENT_URI);
        if (intent.resolveActivity(getApplicationContext()) != null)
            startActivityForResult(intent, REQUEST_CODE_PICK);
    }
});
```

Es wird eine Aktivität der Adressbuch-App von Android tätig, und wir entnehmen dem Result-Code, ob der Benutzer einen Adressbucheintrag gewählt, oder das Adressbuch per Rückwärtsschalter verlassen hat. Im positiven Fall extrahieren wir mit noch zu erlernenden Datenbanktechniken den Namen des gewählten Adressbucheintrags und präsentieren diesen in einer **TextView**-Komponente:

```

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent intent) {
    if (requestCode == REQUEST_CODE_PICK) {
        TextView tv = findViewById(R.id.tvSelected);
        if (resultCode == RESULT_OK) {
            String[] projection = {ContactsContract.CommonDataKinds.Phone.DISPLAY_NAME};
            Cursor cursor = getContentResolver().query(intent.getData(), projection,
                null, null, null);
            cursor.moveToFirst();
            String name = cursor.getString(0);
            tv.setText(this.getString(R.string.selected)+" "+name);
            cursor.close();
        } else
            tv.setText(this.getString(R.string.selected)+" "+getString(R.string.neg_answer));
    }
}
```

Das im Verlauf von Abschnitt 9.4 erstellte Beispielprogramm ist im folgenden Ordner zu finden:

...\\BspUeb\\Intent\\Implizit

## 9.5 Intents in Wartestellung

Als Entwickler einer App, die zur Erleichterung der anschließenden Erläuterungen als *Mapp* bezeichnet werden soll, kann man ein explizites **Intent**-Objekt, das einen Aufruf einer Mapp-Anwendungskomponente bewirkt, in ein Objekt der Klasse **PendingIntent** verpacken und einer fremden App zur Verfügung stellen, die es später mit den Rechten der Anwendung Mapp verwenden kann. Auf diese Weise wird z. B. ein vollberechtigter **Intent**-Einsatz über den heruntergezogenen Vorhang der Benachrichtigungszeile ermöglicht. Weitere Details zu schwebenden Intents finden sich z. B. bei Staudemeyer (2013, S. 68f) und in der SDK-Dokumentation.<sup>1</sup>

## 9.6 Übungsaufgaben zu Kapitel 9

- 1) In der Klasse **Settings** im Paket **android.provider** sind einige Aktionen definiert, die das direkte Öffnen von speziellen Einstellungsformularen per Intent erlauben. Integrieren Sie z. B. die **ACTION\_LOCALE\_SETTINGS** in eine eigene Anwendung.

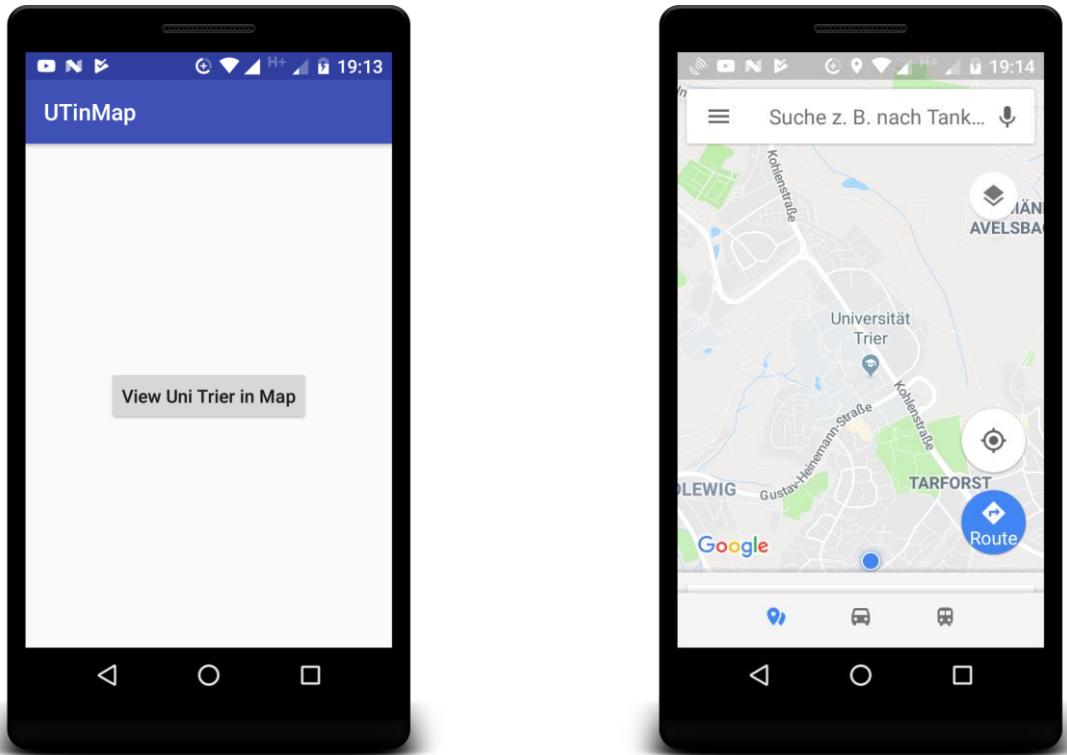
---

<sup>1</sup> <https://developer.android.com/reference/android/app/PendingIntent.html>

2) Welche von den folgenden Aussagen sind richtig bzw. falsch?

1. Eine Aktivität ohne Intent-Filter kann nicht per Intent gestartet werden.
2. Um in einer eigenen App fremde Komponenten (z. B. das Android-Adressbuch) über implizite Intents nutzen zu können, müssen bei der Installation der eigenen App entsprechende Rechte beantragt werden.

3) Erstellen Sie eine App, die auf Knopfdruck den Standort der Universität Trier in einer Karte anzeigt, z. B.:



Geographische Koordinaten der Universität Trier (Gebäude E):

- Breitengrad: 49.747355
- Längengrad: 6.687724

Auf einem emulierten Android-Gerät können die Kartendienste von Google nur dann genutzt werden, wenn ein Systemabbild zum Einsatz kommt, das die Google APIs enthält (vgl. Abschnitt 3.2 zum AVD-Manager).

---

## 10 Multithreading

Wie Sie bereits wissen, werden per Voreinstellung alle Komponenten einer Android-App von der selben virtuellen Maschine (DVM oder Nachfolger ART) ausgeführt, die als Linux-Prozess läuft. Abweichungen von dieser Voreinstellung (z. B. Verteilung der Komponenten einer App auf mehrere Prozesse, Ausführung von Komponenten aus verschiedenen Apps im selben Prozess) sind per App-Manifestdatei möglich (Komponenten-Attribut **android:process**), aber nur selten erforderlich.

Das Gespann aus Linux und Android beherrscht **Multitasking**, sodass mehrere Prozesse parallel ausgeführt werden können. Man kann also z. B. auf einer Wanderung, die von einem GPS-Tracker permanent aufgezeichnet wird, parallel mit einer anderen App Radio hören. Während mindestens zwei Apps im Hintergrund tätig sind, spricht nichts dagegen, die Busverbindung für die Rückkreise per Browser zu ermitteln, sofern das Mobilfunknetz verfügbar ist.

Begrenzt werden die Multitasking-Fähigkeiten eines Android-Geräts nur durch den verfügbaren Speicher und die CPU-Leistung. Wie in Abschnitt 5.4 zu erfahren war, kann Android bei Speicher mangel Prozesse nach einer Prioritätenliste terminieren, wobei dann alle enthaltenen Anwendungs komponenten betroffen sind.

Weil in der Regel mehr Prozesse als CPU-Kerne vorhanden sind, besteht teilweise nur Quasi Parallelität, wobei den Prozessen der Zugang zu einem CPU-Kern reihum gewährt wird. Weil dabei Linux bzw. Android die Oberhoheit haben und nicht auf die Kooperationsbereitschaft der Prozesse angewiesen sind, spricht man von einem *preemptiven* Multitasking, wie es auch bei anderen modernen Betriebssystemen üblich ist. Sofern nur wenige CPU-Kerne vorhanden sind, reduziert sich zwar die Ausführungsgeschwindigkeit jedes Programms im Vergleich zum Solobetrieb, doch ist in den meisten Anwendungen ein flüssiges Arbeiten möglich.

Als Ergänzung zum gerade beschriebenen *Multitasking*, das ohne Zutun der Anwendungsprogrammierer vom Betriebssystem bewerkstelligt wird, ist es oft sinnvoll oder gar unumgänglich, auch *innerhalb* einer Anwendung nebenläufige *Ausführungsfäden* zu realisieren, wobei man hier vom **Multithreading** spricht. Bei einem Internet-Browser muss man z. B. nach dem Anstoßen eines längeren Downloads nicht untätig den Fortschrittsbalken anstarren, sondern kann parallel mit einem anderen Tab arbeiten.

Das aktuelle Kapitel kann nur einen Einstieg in die komplexe Materie der Multithreading Programmierung unter Android bieten. Wir beschäftigen uns überwiegend mit Hintergrund-Threads zur Unterstützung einer im Vordergrund befindlichen Aktivität. Nicht behandelt werden u.a.:

- Intensives Multithreading mit zahlreichen, lange aktiven Threads  
Hier bewährt sich die Klasse **ThreadPoolExecutor**.<sup>1</sup>
- Multithreading in Diensten  
Selbstverständlich ist das Multithreading auch für die Dienst-Komponenten in Android- Apps hochrelevant (siehe Kapitel 11).
- Geplante Aufgaben  
Für diesen Zweck sind im Laufe der Android-Entwicklung die Klassen **AlarmManager**, **JobScheduler** und **WorkManager** entwickelt worden.<sup>2</sup>

Weiterführende Informationen finden sich z. B. bei Göransson (2014), Meike (2016) und in der Online-Dokumentation für Android-Entwickler.

---

<sup>1</sup> <https://developer.android.com/reference/java/util/concurrent/package-summary>

<sup>2</sup> <https://developer.android.com/topic/performance/scheduling>

## 10.1 Main-Thread und Multithreading

Im Prozess einer Android-App laufen per Voreinstellung alle App-Komponenten und damit alle Methoden im selben Thread, der als **Main-Thread** bezeichnet wird. Manchmal wird er **UI-Thread** genannt, weil hier auch die Ereignisverarbeitung und die Bildschirmausgabe stattfinden (vgl. Abschnitt 8.4). Die Methoden zur Behandlung von UI-Ereignissen sind Beispiele für sogenannte *Call Back - Routinen*. Ausgelöst werden die Ereignisse in der Regel durch den Benutzer, der mit der Hilfe von Eingabegeräten wie Touch Screen oder Tastatur praktisch permanent in der Lage ist, unterschiedliche Wünsche zu artikulieren. Eine Aktivität präsentiert mehr oder weniger viele Bedienelemente und wartet die meiste Zeit darauf, dass eine der zugehörigen Ereignisbehandlungsmethoden durch ein vom Benutzer ausgelöstes Ereignis aufgerufen wird. Neben dem Benutzer kommen auch das Laufzeitsystem und andere Anwendungen als Ereignisquellen in Frage. Vom Laufzeitsystem werden z. B. die in Abschnitt 6.2 beschriebenen Lifecycle-Methoden für Aktivitäten zu bestimmten Gelegenheiten aufgerufen.

Die Bedienoberfläche darf nicht durch zeitaufwändige Methodenaufrufe (z. B. mit einem Netzwerkzugriff, einer Datenbankinitialisierung oder einer komplexen Berechnung) behindert oder gar blockiert werden. Wir haben bei unserem Beispielprogramm **Prime Detection** beobachten können bzw. müssen, dass bei einer zeitaufwändigen Ereignisbearbeitung (im Beispiel: Beurteilung einer sehr großen Primzahl) die Bedienoberfläche blockiert wird (siehe Abschnitt 7.10.3). Wenn eine Aktivität auf eine (weitere) Benutzereingabe innerhalb von ca. 5 Sekunden nicht reagiert, präsentiert Android eine ANR-Fehlermeldung (*Application Not Responding*) und gibt dem Benutzer die Möglichkeit, die Anwendung zu beenden, z. B.:<sup>1</sup>



Solche Auftritte sind peinlich und geschäftsschädigend. Auch kürzere Blockaden der Bedienoberfläche sind unbedingt zu vermeiden, indem zeitaufwändige Arbeiten aus dem UI-Thread heraus gehalten und in einem Hintergrund-Thread ausgeführt werden. Zwei Zitate aus der Android-Dokumentation des Herstellers können dazu dienen, den Begriff *zeitaufwändig* zu konkretisieren:<sup>2 3</sup>

If the main thread cannot finish executing blocks of work within 16ms, the user may observe hitching, lagging, or a lack of UI responsiveness to input.

<sup>1</sup> Ein weiterer Anlass für eine ANR-Fehlermeldung: Ein **BroadcastReceiver** braucht länger als 10 Sekunden, um die **onReceive()** - Methode abzuschließen (siehe unten).

<sup>2</sup> <https://developer.android.com/topic/performance/threads>

<sup>3</sup> <https://developer.android.com/guide/background/>

In general, anything that takes more than a few milliseconds should be delegated to a background thread.

Die Multithreading-Technik kommt nicht nur dann in Frage, wenn eine Anwendung mehrere Aufgaben quasi-gleichzeitig erledigen soll. Sind auf einem Gerät *mehrere* CPU-Kerne verfügbar, dann sollten aufwändige Einzelaufgaben (z. B. das Rendern einer 3D-Ansicht) in Teilaufgaben zerlegt werden, um alle CPU-Kerne einzuspannen und Zeit zu sparen. Mittlerweile (2018) sind bei einem Android-Gerät 4 Kerne guter Standard, und Multi-Core - CPUs erhöhen den Druck auf die Software-Entwickler, per Multithreading für gut skalierende Anwendungen zu sorgen, die alle CPU-Kerne nutzen. Dementsprechend verwenden mittlerweile fast alle nicht-trivialen Android-Anwendungen die Multithreading-Programmierung (Mednieks et al. 2013, S. 121).

Beim Multithreading ist allerdings eine sorgfältige Einsatzplanung erforderlich, denn:

- Thread-Wechsel sind mit einem gewissen Zeitaufwand verbunden und sollten daher nicht zu häufig stattfinden.
- Das Laufzeitsystem wird durch die Verwaltung von Threads zusätzlich belastet.
- Jeder Thread belegt minimal 64KB Speicher.<sup>1</sup>
- Oft erfordert das Synchronisieren von Threads einige Aufmerksamkeit beim Programmierer. Hier kommt es oft zu Fehlern, die zudem aufgrund variabler Ergebnisse schwer zu analysieren sind.

Wie wir aus Kapitel 8 wissen, wurde die Android-Bedienoberfläche aus Performanz- und Konsistenzgründen für den Single-Thread - Betrieb entworfen. Weil Zugriffe auf die Steuerelemente der Bedienoberfläche (z. B. zur Aktualisierung einer **TextView**-Ausgabe) nur aus dem UI- bzw. Main-Thread erlaubt sind, muss ein anderer Thread seine Änderungswünsche als Nachrichten in die Warteschlange des UI-Threads einstellen. Ein direkter UI-Zugriff aus einem anderen Thread führt zu einer Beendigung der App mit Ausnahmefehler oder zu einem irregulären Verhalten (z. B. unterbleibende Anzeige einer wichtigen Information).

Im Zusammenhang mit dem Multithreading sind also *zwei* Regeln zu beachten:

- Der UI-Thread (alias: Haupt-Thread) darf nicht durch aufwändige Methodenaufrufe (mit einem Zeitbedarf von mehr als 16 Millisekunden) belastet werden. Solche Methodenaufrufe müssen in einem Hintergrund-Threads ausgeführt werden.
- Zugriffe auf Bedienelemente sind nur aus dem UI-Thread erlaubt, sodass ein Hintergrund-Thread seine Änderungswünsche über den UI-Thread kanalisiert muss.

Analoge Regeln gelten übrigens auch für andere grafische Bedienoberflächen (z. B. JavaFX, Swing, Windows Presentation Foundation).

Während jeder *Prozess* einen eigenen Adressraum besitzt, laufen die *Threads* eines Programms im selben Adressraum ab, sodass sie gelegentlich auch als *leichtgewichtige Prozesse* bezeichnet werden. Sie haben einen gemeinsamen Heap-Speicher, wohingegen jeder Thread als selbstständiger Kontrollfluss bzw. Ausführungsfaden aber einen eigenen Stack-Speicher benötigt.

---

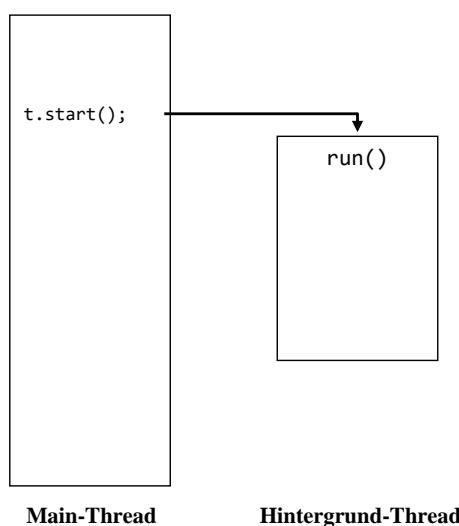
<sup>1</sup> <https://developer.android.com/topic/performance/threads>

## 10.2 Hintergrund-Thread mit elementaren Mitteln realisieren

Ein Thread lässt sich in Java auf elementare Weise durch ein Objekt aus der Klasse **Thread** oder aus einer abgeleiteten Klasse realisieren. Um in einer Ereignisbehandlungsmethode eine Hintergrundtätigkeit anzustoßen, bietet es sich an, eine anonyme Klasse von **Thread** abzuleiten und die **run()** - Methode, die im Hintergrund ausgeführt werden soll, zu überschreiben. Mit dem **start()** - Befehl an das Objekt dieser Klasse sorgt man dafür, dass die **run()** - Methode in einem separaten Thread ausgeführt wird, z. B.:

```
private Thread t;
. . .
public void onClick(View arg0) {
    t = new Thread() {
        @Override
        public void run() {
            // Im Hintergrund auszuführende Tätigkeiten
        }
    };
    t.start();
}
```

Nach dem Start befindet sich ein Thread im Zustand **ready** und wartet auf die Zuteilung eines CPU-Kerns, um seine **run()** - Methode und alle von dort direkt oder indirekt aufgerufenen Methoden auszuführen:



Ein aktiver Thread befindet sich im Zustand **running**. Die virtuelle Maschine verwaltet die Threads in enger Zusammenarbeit mit dem Wirtsbetriebssystem, wobei ein Thread zwischen den Zuständen **ready** und **running** wechselt.

Sobald seine **run()** - Methode abgearbeitet ist, endet ein Thread. Er befindet sich dann im Zustand **terminated** und kann *nicht* erneut gestartet werden.

Über die statische Methode **setThreadPriority()** der Klasse **Process** im Paket **android.os** lässt sich die Priorität des aktuell ausgeführten Threads beeinflussen und somit die Bearbeitungszeit für eine Aufgabe beeinflussen.<sup>1</sup> Google empfiehlt für Hintergrund-Threads den durch die Konstante **THREAD\_PRIORITY\_BACKGROUND** in der Klasse **Process** definierten Wert:<sup>2</sup>

```
android.os.Process.setThreadPriority(android.os.Process.THREAD_PRIORITY_BACKGROUND);
```

<sup>1</sup> <https://developer.android.com/topic/performance/threads>

<sup>2</sup> <https://developer.android.com/training/multiple-threads/define-runnable>

Android unterscheidet zwischen der Vorder- und der Hintergrundgruppe von Threads und verwendet ca. 95% der CPU-Kapazität für die Threads in der Vordergrundgruppe. Per Voreinstellung erbt ein neu gestarteter Thread die Gruppenzugehörigkeit und die Priorität des Threads, aus dem er gestartet worden ist.

Mit der eben skizzierten Technik erstellen wir nun eine Variante des Beispielprogramms zur Primzahlendiagnose, das auch bei großen Kandidaten keinen ANR-Fehler (*Application Not Responding*) produziert. Es sind zwei Aufgaben zu lösen:

- Wir erstellen einen Hintergrund-Thread und lassen in dessen `run()` - Methode den Algorithmus ausführen.
- Wir organisieren den Informationstransfer vom Hintergrund-Thread, der irgendwann sein Ergebnis melden möchte, zum UI-Thread, der als einziger Thread berechtigt ist, Änderungen an den `View`-Objekten der Bedienoberfläche vorzunehmen.

In der `onClick()` - Methode des Befehlsschalters zum Starten der Primzahlendiagnose wird der Schalter durch einen Aufruf der Methode `setEnabled()` vorübergehend deaktiviert, damit bis zur Erledigung eines Auftrags keine weiteren Aufträge abgeschickt werden können. Dann wird ein Thread-fähiges Objekt erstellt mit einer `run()` - Methode, welche die Zeichenfolge mit dem Primzahlkandidaten vom `EditText`-Steuerelement bezieht und anschließend den aus Abschnitt 7.10.3 bekannten Algorithmus ausgeführt:

```
private EditText candidate;
private TextView result;
private Button button;
private Thread t;
private Handler h = new Handler();

.

@Override
public void onClick(View view) {
    button.setEnabled(false);
    t = new Thread() {
        @Override
        public void run() {
            android.os.Process.setThreadPriority(android.os.Process.THREAD_PRIORITY_BACKGROUND);
            String sCandidate = candidate.getText().toString();
            final String sResult;
            boolean df = false;
            long cand, i, mdc;

            if (sCandidate.length() == 0)
                return;
            cand = Long.parseLong(sCandidate);
            mdc = (int) Math.sqrt(cand);
            for (i = 2; i <= mdc; i++)
                if (cand % i == 0) {
                    df = true;
                    break;
                }
            if (cand <= 1)
                sResult = sCandidate + " " + getString(R.string.falseArgument);
            else
                if (df)
                    sResult = sCandidate + " " + getString(R.string.noPrime) + "\n" +
                        getString(R.string.divisor) + Long.toString(i) + ")";
                else
                    sResult = sCandidate + " " + getString(R.string.prime);
        }
    }.start();
    h.post(new Runnable() {
        public void run() {
            result.setText(sResult);
        }
    });
}
```

```

    h.post(new Runnable() {
        @Override
        public void run() {
            result.setText(sResult);
            button.setEnabled(true);
        }
    });
}
t.start();
}

```

Das Objekt aus der anonymen **Thread**-Ableitung wird schließlich gestartet:

```
t.start();
```

Um seine (im lokalen **String**-Objekt **sResult** zwischengespeicherten) Ergebnisse im **TextView**-Objekt **result** zu präsentieren, beauftragt der Hintergrund-Thread ein Objekt der Klasse **Handler**. Es wird per Deklaration mit Initialisierung im Haupt-Thread erzeugt und ist daher mit dessen Nachrichtenwarteschlange verbunden:

```
private Handler h = new Handler();
```

Über seine Methode **post()** kann man das **Handler**-Objekt beauftragen, ein **Runnable**-Objekt (ein Objekt einer Klasse, die das Interface **Runnable** implementiert), in die Nachrichtenwarteschlange des Haupt-Threads zu stellen, sodass die **run()** - Methode des Objekts baldmöglichst im Haupt-Thread ausgeführt wird.

Um das Interface **Runnable** zu erfüllen, muss eine Klasse lediglich eine Methode namens **run()** mit folgenden Eigenschaften implementieren:

- Die Methode ist öffentlich zugänglich (**public**), hat den Rückgabetyp **void** und keine Parameter.
- Sie besitzt keine **throws**-Klausel zum Deklarieren einer Ausnahme.

Man kann übrigens auch den Hintergrund-Thread selbst auf der Basis einer das Interface **Runnable** implementierenden Klasse realisieren und hat dabei im Vergleich zu der oben gewählten Lösung den Vorteil, statt **Thread** eine beliebige Basisklasse verwenden zu können.

In der **run()** - Methode, die im Haupt-Thread ausgeführt werden soll, veröffentlichen wir das Prüfergebnis und reaktivieren den Befehlsschalter:

```

h.post(new Runnable() {
    @Override
    public void run() {
        result.setText(sResult);
        button.setEnabled(true);
    }
});

```

Während einer langwierigen Primzahlendiagnose sperrt die aktuelle Version unserer App den Befehlsschalter, erlaubt dem Benutzer aber, den nächsten Auftrag in das Texteingabefeld zu schreiben, z. B.:



Den Informationstransfer vom Hintergrund-Thread in den UI-Thread können wir auch mit der **Activity**-Methode **runOnUiThread()**

```
PrimeActivity.this.runOnUiThread(new Runnable() {
    @Override
    public void run() {
        result.setText(sResult);
        button.setEnabled(true);
    }
});
```

oder mit der **View**-Methode **post()** bewerkstelligen:<sup>1</sup>

```
result.post(new Runnable() {
    @Override
    public void run() {
        result.setText(sResult);
        button.setEnabled(true);
    }
});
```

Eine weitere, momentan von uns nicht benötigte Funktion der Klasse **Handler** besteht übrigens darin, Nachrichten und **Runnable**-Objekte nach einem *Zeitplan* ausführen zu lassen.<sup>2</sup>

Weil der Hintergrund-Thread auf einer anonymen, innerhalb einer Aktivitätsmethode definierten Klasse basiert, können wir in seiner **run()** - Methode bequem über Instanzvariablen der Aktivitätsklasse auf die Steuerelemente zugreifen. Wenn in einem Hintergrund-Thread *keine* Referenzen auf die Steuerelemente vorhanden sind, besteht eine sinnvolle Strategie darin, eine **Handler**-Ableitung zu definieren und dort die Methode **handleMessage()** zu überschreiben. Dann kann sich der Hintergrund-Thread auf die Übermittlung von Sachinformationen (z. B. Zwischenstände, Endergebnis) in

<sup>1</sup> Wie ein Blick in den Quellcode der Android-Klassen **Activity** bzw. **View** zeigt, kommt dabei ebenfalls die **Handler**-Methode **post()** zum Einsatz, z. B.:

```
public final void runOnUiThread(Runnable action) {
    if (Thread.currentThread() != mUiThread) {
        mHandler.post(action);
    } else {
        action.run();
    }
}
```

<sup>2</sup> <https://developer.android.com/reference/android/os/Handler>

**handleMessage()** - Aufrufen beschränken, während sich der **Handler** um die Konsequenzen für Bedienelemente kümmert. Diese Lösung wird in Abschnitt 11.3.3 verwendet.

### 10.3 Altlasten durch schlecht programmierte Helper-Threads

Wenn eine Activity zerstört wird, der zugehörige Prozess aber weiter existiert, liegt es in der Verantwortung des Programmierers, alle von der Activity gestarteten und noch aktiven Hintergrund-Threads zu beenden. Aus Abschnitt 6 wissen wir, dass eine Activity bei einer Konfigurationsänderung (z. B. bei einem Orientierungswechsel) per Voreinstellung zerstört und dann im selben Linux-Prozess neu gestartet wird. Zur Activity gehörige Hintergrund-Threads müssen in der Lebenszyklus-Methode **onDestroy()** beendet werden, weil sie anderenfalls bis zum Ende ihrer **run()** - Methode weiterlaufen und das System belasten.

Noch schlimmer ist, dass Hintergrund-Threads oft eine implizite oder explizite Referenz auf das Aktivitäts-Objekt besitzen und folglich den Garbage Collector daran hindern, die Aktivität nach Beendigung ihrer Methode **onDestroy()** aus dem Speicher zu entfernen. Zu einer impliziten Referenz auf das Aktivitätsobjekt kommt es z. B. dann, wenn Threads auf einer nicht-statischen Mitgliedsklasse der Aktivität oder (wie im Beispiel) auf einer anonymen, in einer Aktivitätsmethode definierten Klasse basieren. Wie wir aus Abschnitt 6.4.3 wissen, ist der Speicherverbrauch einer Aktivität nach oben nur durch den für die gesamte App verfügbaren Speicher begrenzt. Somit kann das Blockieren des Garbage Collectors eine Speicherverschwendungen bzw. ein Speicherleck (engl. *memory leak*) von erheblichem Ausmaß zur Folge haben.

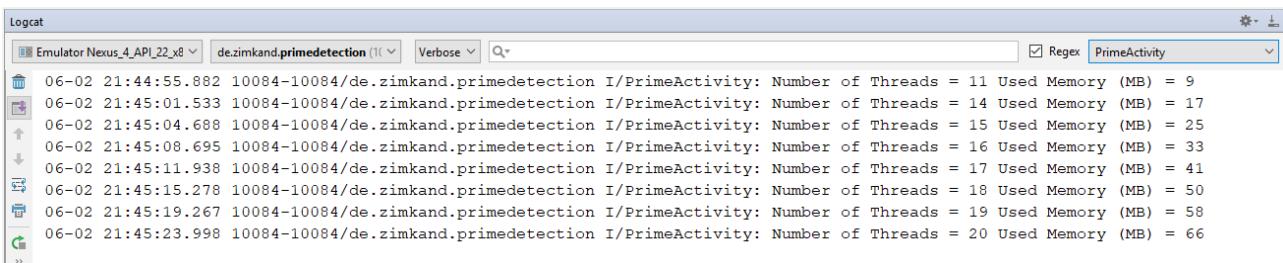
Um die CPU-Zeit - und Speicherverschwendungen durch einen schlecht programmierten Helper-Thread zu demonstrieren, erhöhen wir im Beispielprogramm aus Abschnitt 10.2 den Heap-Speicherbedarf der Aktivität durch eine sinnlose Instanzvariable um 8 MB:

```
private long[] memLoad = new long[1048576];
```

Außerdem erweitern wir die **onCreate()** - Methode um Anweisungen, welche die Anzahl der im Prozess der App aktiven Threads sowie den Speicherverbrauch (vgl. Abschnitt 6.4.1 zur Ermittlung) ins **Logcat**-Fenster schreiben:

```
String TAG = this.getClass().getSimpleName();
Runtime runtime = Runtime.getRuntime();
int numberOfThreads = Thread.getAllStackTraces().keySet().size();
long consumed = (runtime.totalMemory() - runtime.freeMemory())/1024;
Log.i(TAG, "Number of Threads = " + String.valueOf(numberOfThreads) +
    " Used Memory (KB) = " + String.valueOf(consumed));
```

Wenn der Hintergrund-Thread durch eine große Primzahl längere Zeit beschäftigt ist, während seiner Tätigkeit durch Orientierungswechsel ein Neustart der Aktivität provoziert und anschließend die abgebrochene Primzahlendiagnose jeweils neu gestartet wird, dann zeigt das **Logcat**-Fenster eine kontinuierlich steigende Anzahl von Threads sowie einen wachsenden Speicherverbrauch:



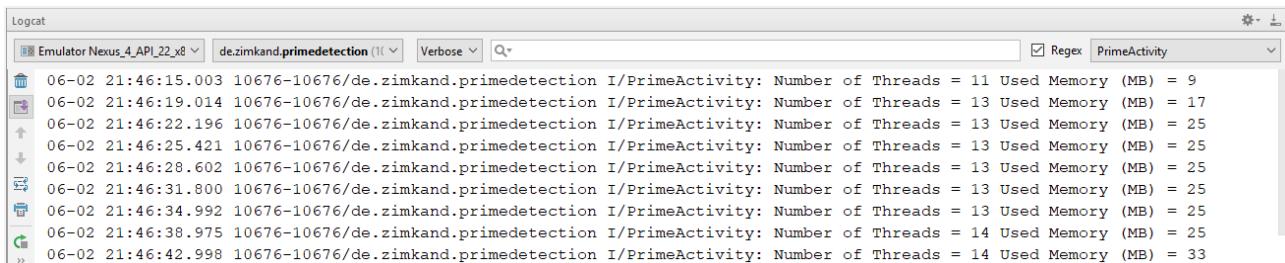
Zur Beseitigung des Problems müssen wir die Activity-Methode **onDestroy()** überschreiben und dort den Primzahlen-Thread terminieren. Zum Beenden eines Threads sollte die veraltete (abgewertete) Methode **stop()** *nicht* mehr verwendet werden. Stattdessen fordert man den Thread mit der Methode **interrupt()** auf, seine Tätigkeit einzustellen:

```
@Override
protected void onDestroy() {
    super.onDestroy();
    if (t != null)
        t.interrupt();
}
```

Ein kompetenter Thread-Programmierer prüft regelmäßig über die Methode **isInterrupted()**, ob die **run()** - Methode baldigst verlassen und somit der Thread beendet werden sollte. In unserem Beispiel ist dieser Test in der **for**-Schleife zur Prüfung der Teiler-Kandidaten angebracht:

```
for (i = 2; i <= mdc; i++) {
    if (Thread.currentThread().isInterrupted())
        return;
    if (cand % i == 0) {
        df = true;
        break;
    }
}
```

Nun führen Orientierungswechsel-bedingte Neustarts der Aktivität nicht mehr zu einer Steigerung der Thread-Zahl, und der Garbage Collector kann seines Amtes walten:



Einen gelegentlichen (temporären) Anstieg des Speicherverbrauchs durch unsere App können wir nicht verhindern, weil die Garbage Collector - Einsätze von der Laufzeitumgebung gesteuert werden.

Im Beispiel ist es angemessen, den Hintergrund-Thread *nicht* schon in **onStop()**, sondern erst in **onDestroy()** zu beenden. Eventuell vertreibt sich der Benutzer während einer langwierigen Primzahlendiagnose die Wartezeit mit einer anderen App und rechnet damit, dass die Analyse im Hintergrund fortgesetzt wird.

Das komplette AS-Projekt mit der im aktuellen Abschnitt beschriebenen **Prime Detection** - Variante ist im folgenden Ordner zu finden:

...\\BspUeb\\Multithreading\\PrimeDetectionThread

## 10.4 AsyncTask

Die Klasse **AsyncTask** aus dem Android-SDK vereinfacht die asynchrone Aufgabenabwicklung ...

- ohne Belastung des UI-Threads
- mit anschließender Ergebnisanzeige (und optionaler zwischenzeitlicher Fortschrittsanzeige) im UI-Thread

Auf den Android-Entwicklerseiten der Firma Google findet sich die folgende Charakterisierung und Einsatzempfehlung zur Klasse **AsyncTask**:<sup>1</sup>

<sup>1</sup> <https://developer.android.com/reference/android/os/AsyncTask>

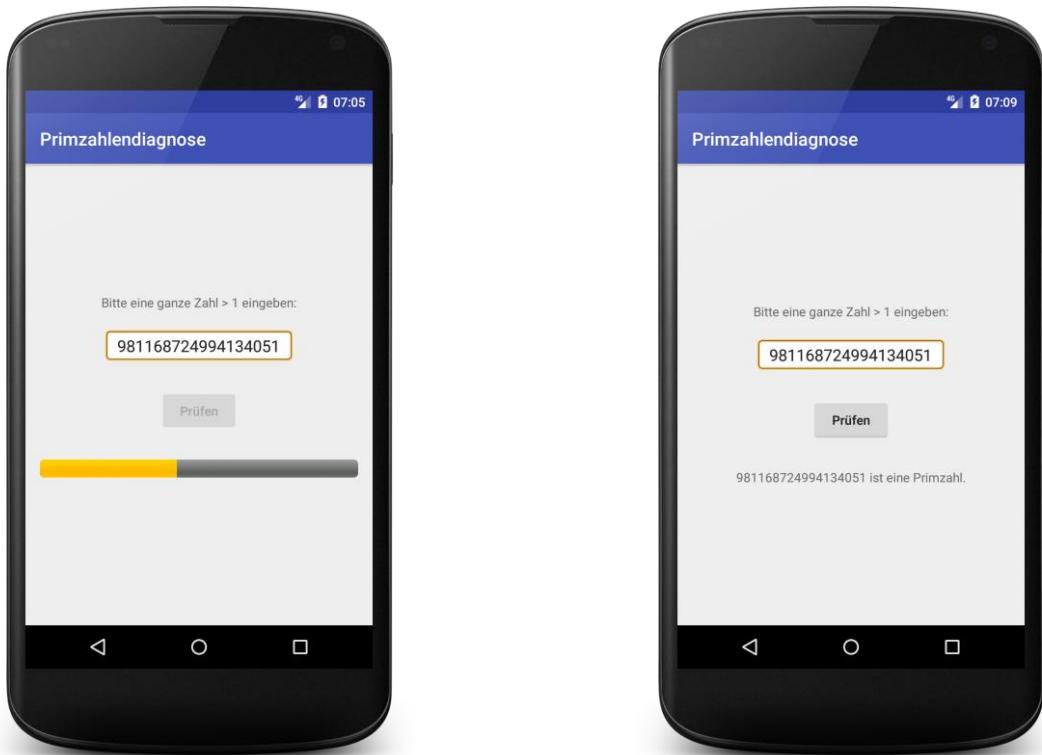
AsyncTask is designed to be a helper class around `Thread` and `Handler` and does not constitute a generic threading framework. AsyncTasks should ideally be used for short operations (a few seconds at the most.) If you need to keep threads running for long periods of time, it is highly recommended you use the various APIs provided by the `java.util.concurrent` package such as `Executor`, `ThreadPoolExecutor` and `FutureTask`.

Für eine Beschreibung der genannten Klassen aus dem Paket `java.util.concurrent` fehlt in diesem Kurs die Zeit (siehe z. B. Baltes-Götz & Götz 2018, Kapitel 15 sowie die Online-Dokumentation für Android-Entwickler<sup>1</sup>).

Eine in ihren Konsequenzen leicht einzuschätzende Restriktion der **AsyncTask**-Lösung besteht darin, dass (seit API-Level 11) alle **AsyncTask**-Objekte einer App denselben Thread benutzen. Sollen in einer Aktivität mehrere Hintergrundaufgaben abgewickelt werden, sind also Blockaden zu befürchten. Werden mehrere, simultan arbeitende Threads benötigt, sollte eine Lösung auf Basis der Klasse **ThreadPoolExecutor** angestrebt werden.

Für unser **Prime Detection** - Demonstrationsprogramm ist die Klasse **AsyncTask** nicht perfekt geeignet, weil die Prüfung bei einer sehr großen Primzahl auf einem Gerät mit schwacher CPU-Leistung durchaus mehrere Minuten dauern kann. Wir demonstrieren trotzdem die **AsyncTask**-Verwendung und leiten dazu eine eigene Klasse aus dieser abstrakten Basisklasse ab.

Damit die Benutzer bei einer langen Wartezeit nicht die Geduld verlieren, bauen wir eine Fortschrittsanzeige ein (vgl. Abschnitt 8.5.6 zum **ProgressBar**-Steuerelement). So verhält sich die erweiterte Version des Primzahlendetektors:



Um in der **AsyncTask** - Ableitung bequem auf die Instanzvariablen der Aktivitätsklasse (z. B. zu den Steuerelementen) zugreifen zu können, erstellen wir in der Aktivitätsklasse **PrimeActivity** eine innere Klasse, die den Namen **AsyncPrimeTask** erhalten soll:

<sup>1</sup> <https://developer.android.com/reference/java/util/concurrent/package-summary>

```

private class AsyncPrimeTask extends AsyncTask<String, Integer, String> {

    private Context context = PrimeActivity.this;

    @Override
    protected void onPreExecute() {
        result.setText("");
        button.setEnabled(false);
        progressBar.setVisibility(ProgressBar.VISIBLE);
        progressBar.setProgress(0);
    }

    @Override
    protected String doInBackground(String... arg) {
        boolean df = false;
        long cand, i, mdc, widthSeg;
        int nSeg = 100;
        if (arg[0].length() == 0)
            return null;
        cand = Long.parseLong(arg[0]);
        mdc = (int) Math.sqrt(cand);
        widthSeg = Math.max(10_000, mdc/nSeg);
        for (i = 2; i <= mdc; i++) {
            if (this.isCancelled())
                return null;
            if (i % widthSeg == 0)
                publishProgress((int)(i/(float)mdc*100));
            if (cand % i == 0) {
                df = true;
                break;
            }
        }
        if (cand <= 1)
            return arg[0] + " " + context.getString(R.string.falseArgument);
        else
            if (df)
                return arg[0] + " " + context.getString(R.string.noPrime) + "\n(" +
                    context.getString(R.string.divisor) + Long.toString(i)+ ")";
            else
                return arg[0] + " " + context.getString(R.string.prime);
    }

    @Override
    protected void onProgressUpdate(Integer... progress) {
        progressBar.setProgress(progress[0]);
    }

    @Override
    protected void onPostExecute(String res) {
        result.setText(res);
        progressBar.setVisibility(ProgressBar.GONE);
        button.setEnabled(true);
    }
}

```

Die Klasse **AsyncTask** ist abstrakt und generisch definiert,

```

public abstract class AsyncTask<Params, Progress, Result> {
    . . .
}

```

und wir müssen bei der Angabe der **AsyncPrimeTask**-Basisklasse

```

private class AsyncPrimeTask extends AsyncTask<String, Integer, String>
drei Typparameter konkretisieren, wobei nur Referenztypen erlaubt sind:

```

- **Params**

Beim Starten einer asynchronen Aufgabenbearbeitung (siehe unten) ist an die Methode **execute()** ein Serienparameter mit diesem Elementtyp zu übergeben.

- **Progress**

Dies ist der Parametertyp der Methode **onProgressUpdate()** zur Meldung des Bearbeitungsfortschritts.

- **Result**

Von diesem Typ ist die Rückgabe der im Hintergrund-Thread ausgeführten Methode **doInBackground()**.

Wird ein Typparameter *nicht* benötigt, weil z. B. die Methode **onProgressUpdate()** zur Meldung des Bearbeitungsfortschritts *nicht* implementiert werden soll, gibt man an dieser Position **Void** als Typaktualparameter an.

Die wichtigsten Methoden der Klasse **AsyncTask** sind:

- **public final AsyncTask<Params, Progress, Result> execute(Params... params)**

Mit dieser Methode wird die Aufgabenbearbeitung initiiert. Der Aufruf erfolgt im UI-Thread und kehrt praktisch sofort zurück. Weil ein Serienparameter verwendet wird, kann man eine variable Anzahl von Einzelaufträgen übergeben. Als Rückgabe liefert das beauftragte Objekt eine Referenz auf sich selbst. Weil die Methode als **final** definiert ist, darf sie nicht überschrieben werden. Für jedes **AsyncTask**-Objekt ist nur *ein* **execute()** - Aufruf erlaubt.<sup>1</sup> Beim wiederholten Einsatz einer **AsyncTask**-Ableitung muss also jeweils ein neues Objekt erzeugt werden.

Das Beispielprogramm enthält einen **execute()** - Aufruf in der **onClick()** - Methode zum Befehlsschalter:

```
@Override
public void onClick(View view) {
    String s = candidate.getText().toString();
    if (s.length() != 0) {
        apt = new AsyncPrimeTask();
        apt.execute(s);
    }
}
```

- **protected void onPreExecute()**

Diese Methode läuft vor dem Start der Hintergrundaktivität *im UI-Thread* ab und kann für vorbereitende Maßnahmen verwendet werden. Im Primzahlenbeispielprogramm wird das vorherige Prüfergebnis beseitigt, der Befehlsschalter zur Anforderungen weiterer Prüfungen deaktiviert sowie die Fortschrittsanzeige sichtbar gemacht und initialisiert.<sup>2</sup>

```
@Override
protected void onPreExecute() {
    result.setText("");
    button.setEnabled(false);
    progressBar.setVisibility(ProgressBar.VISIBLE);
    progressBar.setProgress(0);
}
```

Weil die überschriebene Basisklassenvariante keine Anweisungen enthält, sparen wir uns deren Aufruf.

---

<sup>1</sup> <https://developer.android.com/reference/android/os/AsyncTask.html>

<sup>2</sup> Im Beispiel hätte man auf die Methode **onPreExecute()** verzichten und die UI-Vorbereitungen in der Methode **onClick()** vornehmen können.

- **protected Result doInBackground(Params... params)**

Diese in der Basisklasse abstrakt definierte Methode muss auf jeden Fall implementiert werden. Sie leistet die eigentliche Arbeit und wird in einem Hintergrund-Thread ausgeführt. Weil ein Serienparameter verwendet wird, kann man beim Start der Aufgabenbearbeitung per **execute()** eine variable Anzahl von Einzelaufträgen übergeben (z. B. mehrere Dateien zum Herunterladen). Im Beispiel enthält **doInBackground()** den praktisch unveränderten Code aus der früheren **onClick()** - Behandlungsmethode zum Befehlsschalter des Primzahlendiagnoseprogramms.

- **protected void onProgressUpdate(Progress... values)**

Diese Methode läuft *im UI-Thread* ab und dient zur Anzeige des Bearbeitungsfortschritts. Auslöser für die Meldung des Bearbeitungsfortschritts ist ein Aufruf der Methode **publishProgress()**, der meist in der Methode **doInBackground()** erfolgt. Im Beispielprogramm wird der Fortschrittsbalken aktualisiert:

```
@Override
protected void onProgressUpdate(Integer... progress) {
    progressBar.setProgress(progress[0]);
}
```

Weil die überschriebene Basisklassenvariante keine Anweisungen enthält, sparen wir uns deren Aufruf.

- **protected void onPostExecute(Result result)**

Diese Methode wird nach Beendigung von **doInBackground()** aufgerufen und erhält per Parameter deren Rückgabewert. Weil **onPostExecute()** im UI-Thread läuft, kann die Methode uneingeschränkt zur Ergebnisanzeige verwendet werden.

Im Beispielprogramm zeigen wir das Ergebnis der Primzahlendiagnose an, lassen den Fortschrittsbalken verschwinden und reaktivieren den Befehlsschalter:

```
@Override
protected void onPostExecute(String res) {
    if (res != null)
        result.setText(res);
    progressBar.setVisibility(ProgressBar.GONE);
    button.setEnabled(true);
}
```

Weil die überschriebene Basisklassenvariante keine Anweisungen enthält, sparen wir uns deren Aufruf.

Auch bei Verwendung der Klasse **AsyncTask** müssen wir dafür sorgen, dass beim Zerstören einer Aktivität eine von ihr initiierte und noch laufende Hintergrundaktivität beendet wird (vgl. Abschnitt 10.3). Dazu rufen wir (meist im Rahmen der **Activity**-Lebenszyklus-Methode **onDestroy()**) die  **AsyncTask**-Methode **cancel()** auf, z. B.:

```
@Override
protected void onDestroy() {
    super.onDestroy();
    if (apt != null)
        apt.cancel(true);
}
```

Damit eine Unterbrechung tatsächlich klappt, müssen wir in der  **AsyncTask**-Methode **doInBackground()** zu passenden Gelegenheiten über die Methode **isCancelled()** prüfen, ob ein Ausstieg gewünscht ist, und passend reagieren, z. B.:

```
if (this.isCancelled())
    return null;
```

Ohne diese Aufräumaktion bleibt das **AsyncPrimeTask**-Objekt aktiv und behindert den weiteren Betrieb, weil alle  **AsyncTask**-Aufträge einer App durch denselben Thread erledigt werden. Zudem

besitzt es eine implizite Referenz auf das bei seiner Erstellung beteiligte Aktivitätsobjekt, weil **AsyncPrimeTask** als nicht-statische Mitgliedsklasse von **PrimeActivity** definiert worden ist.<sup>1</sup> Folglich kann ein zerstörtes Aktivitätsobjekt nicht vom Garbage Collector aus dem Speicher entfernt werden, und es kommt es einem Speicherleck (engl.: *memory leak*).

Mit Hilfe der in Abschnitt 10.3 beschriebenen Diagnosetechnik (Aktivität auf aufgeblähtem Heap-Speicherbedarf und **Log**-Ausgaben in **onCreate()**) lässt sich nachweisen, dass bei einem Verzicht auf den **cancel()** - Aufruf in der **onDestroy()** - Methode der Klasse **PrimeActivity** z. B. Orientierungswechsel zu einer Speicherverschwendungen führen:

```

Logcat
Emulator Nexus_4_API_22_x8 de.zimkand.primedetection [1] Verbose Regex PrimeActivity
06-03 08:19:05.945 15780-15780/de.zimkand.primedetection I/PrimeActivity: Number of Threads = 11 Used Memory (MB) = 9
06-03 08:19:12.466 15780-15780/de.zimkand.primedetection I/PrimeActivity: Number of Threads = 14 Used Memory (MB) = 17
06-03 08:19:16.431 15780-15780/de.zimkand.primedetection I/PrimeActivity: Number of Threads = 14 Used Memory (MB) = 25
06-03 08:19:20.428 15780-15780/de.zimkand.primedetection I/PrimeActivity: Number of Threads = 14 Used Memory (MB) = 33
06-03 08:19:23.697 15780-15780/de.zimkand.primedetection I/PrimeActivity: Number of Threads = 14 Used Memory (MB) = 42
06-03 08:19:27.746 15780-15780/de.zimkand.primedetection I/PrimeActivity: Number of Threads = 14 Used Memory (MB) = 50
06-03 08:19:31.691 15780-15780/de.zimkand.primedetection I/PrimeActivity: Number of Threads = 14 Used Memory (MB) = 58
06-03 08:19:36.469 15780-15780/de.zimkand.primedetection I/PrimeActivity: Number of Threads = 14 Used Memory (MB) = 66
06-03 08:19:40.399 15780-15780/de.zimkand.primedetection I/PrimeActivity: Number of Threads = 14 Used Memory (MB) = 74
06-03 08:19:44.382 15780-15780/de.zimkand.primedetection I/PrimeActivity: Number of Threads = 14 Used Memory (MB) = 82
06-03 08:19:49.185 15780-15780/de.zimkand.primedetection I/PrimeActivity: Number of Threads = 14 Used Memory (MB) = 90
>>

```

Die Anzahl der Threads schwilkt durch den Programmierfehler *nicht* an, weil Android für alle **AsyncTask**-Einsätze stets denselben Thread verwendet. Es entsteht jedoch ein extrem lästiger Auftragsstau, weil die asynchronen Aufgaben alle im selben Thread nacheinander ausgeführt werden. Eine neu initiierte Aufgabe wird also erst dann ausgeführt, denn alle im Stau befindlichen und völlig nutzlos gewordenen Aufgaben abgearbeitet sind.

Das komplette AS-Projekt mit der im aktuellen Abschnitt beschriebenen **Prime Detection** - Variante ist im folgenden Ordner zu finden:

...\\BspUeb\\Multithreading\\PrimeDetection AsyncTask

## 10.5 Nachrichtenübermittlung zwischen verschiedenen Threads

Soll ein Thread Nachrichten erhalten können, richtet man für ihn eine Nachrichtenwarteschlange ein, die als verkettete Liste realisiert und durch ein Objekt der Klasse **android.os.MessageQueue** modelliert wird. Eine Nachricht wird mit Hilfe der Klasse **android.os.Handler** in die Warteschlange eingestellt. Für die sequentielle Abarbeitung der Nachrichten nach dem Prinzip **first in, first out** (mit der Möglichkeit zur Vereinbarung einer verzögerten Ausführung) sorgt ein Objekt der Klasse **android.osLooper**. Es ermittelt die zur Bearbeitung anstehende Nachricht und übermittelt sie an den denselben **Handler**, der die Nachricht in die Warteschlange eingestellt hat.

Für den Haupt-Thread existiert in jedem Fall ein **Looper** und eine Nachrichtenwarteschlange, über die z.B. auch die Lebenszyklus-Methodenaufrufe für Aktivitäten und Fragmente Dienste abgewickelt werden. Es kann auch für jeden anderen Thread eine Nachrichtenwarteschlange eingerichtet werden (siehe Göransson 2014, S. 49f).

<sup>1</sup> Google beschreibt das Problem auf der folgenden Webseite  
<https://developer.android.com/topic/performance/threads>

und empfiehlt zur Vermeidung:

- Die implizite Referenz auf ein umgebendes Aktivitätsobjekt sollte vermieden werden (z. B. durch Verwendung einer *statischen* Mitgliedsklasse).
- Nun wird eine alternative Möglichkeit zur Aktualisierung der Bedienoberfläche durch die asynchrone Aufgabe benötigt. Mit Hilfe der Klasse **WeakReference<MainActivity>** wird die Verwendung einer expliziten Referenz vermieden. Ein Objekt dieser Klasse kann eine Referenz auf das Aktivitätsobjekt aufbewahren, ohne die Beseitigung der Aktivität durch den Garbage Collector zu verhindern.

Wir werden diese Empfehlungen in einem späteren Beispiel umsetzen (siehe Abschnitt 11.3.3.3).

Enthält die Warteschlange eines Threads keine ausführbare Nachricht (mit einer Ausführungszeit, die nicht in der Zukunft liegt), dann ist der Thread passiv (engl.: *idle*). Zum mindesten beim Haupt-Thread sollte dies der Normalzustand sein, damit der Thread auf neu eintreffende Nachrichten verzögerungsfrei reagieren kann. Ein Thread kann sich vom System über das Eintreten in den idle-Zustand informieren, lassen, um diese Situation zur Ausführung von weniger wichtigen Arbeiten auszunutzen (Göransson 2014, S. 52ff).

Wir haben die Klasse **Handler** schon in Abschnitt 10.2 kennengelernt und ihre Methode **post()** dazu benutzt, um von einem Hintergrund-Thread aus ein **Runnable**-Objekt in die Nachrichtenwarteschlange des Haupt-Threads zu stellen, um die **run()** - Methode dieses Objekts im Haupt-Thread ausführen zu lassen.

Wird im Konstruktor-Aufruf für ein neues **Handler**-Objekt kein **Looper** angegeben, dann wird das **Handler**-Objekt mit dem **Looper** des aktuellen Threads verbunden, was beim UI-Thread stets möglich ist, z.B.:

```
private Handler h = new Handler();
```

Besitzt der aktuelle Thread keine Nachrichtenwarteschlange, kommt es zu einem Ausnahmefehler.

Zu einem Thread können mehrere **Handler** existieren und Nachrichten in die Warteschlange einfügen. Steht eine Nachricht zur Bearbeitung an, wird sie vom **Looper** an den **Handler** übergeben, der sie eingefügt hat. Das Einfügen und die Verarbeitung einer Nachricht finden in der Regel in verschiedenen Threads statt, doch ist es z.B. möglich, dass der UI-Thread Nachrichten in die eigene Warteschlange stellt, die möglichst bald oder auch mit einer Verzögerung ausgeführt werden sollen.

Das in Abschnitt 10.2 beschriebene Beispielprogramm setzt für eine aufwändige Berechnung einen Hintergrund-Thread ein, der eine Referenz zu einem **Handler** des UI-Threads besitzt und diesen per **post()** - Methode

```
public final boolean post(Runnable runnable)
```

auffordert, ein **Runnable**-Objekt als Nachricht in die Warteschlange des UI-Threads zu stellen:

```
h.post(new Runnable() {
    @Override
    public void run() {
        result.setText(sResult);
        button.setEnabled(true);
    }
});
```

Äquivalent zur **Handler**-Methode **post()** arbeiten:

- die **Activity**-Methode **runOnUiThread()**
- die **View**-Methode **post()**

Neben einer Nachricht vom Typ **Runnable** kann ein **Handler** auch eine Nachricht vom Typ **Message** in die Warteschlange einreihen, wobei die Methode **sendMessage()** zu verwenden ist:

```
public final boolean sendMessage(Message message)
```

Ein **Message**-Objekt ist ein Container für diverse Informationen.<sup>1</sup> Im folgenden Codesegment, das zu einem später vorgestellten Beispielprogramm gehört, wird ein **Bundle**-Objekt per **setData()** in ein **Message**-Objekt eingefügt, das anschließend per **sendMessage()** in die Warteschlange des zu einem **Handler**-Objekt gehörenden Threads gestellt wird:

---

<sup>1</sup> <https://developer.android.com/reference/android/os/Message>

```
Bundle bundle = new Bundle();
bundle.putFloat("progress", i / (float) mdc * 100);
Message message = new Message();
message.setData(bundle);
handler.sendMessage(message);
```

Weil **Message**-Objekte in einer App potentiell zahlreich benötigt werden, hebt Android diese Objekte in einem Message-Pool zur Wiederverwendung auf, um Aufwand bei der Objektkreation zu sparen. Wer sich an diesem Recycling-System beteiligen möchte, verzichtet auf die Neukreation und verwendet stattdessen über die statische **Handler**-Methode **obtain()** ein **Message**-Objekt aus dem Pool, z.B.:

```
Message message = Message.obtain();
```

Während eine Nachricht vom Typ **Runnable** ohne große Mitwirkung des zuständigen Handlers in dessen Thread ausgeführt wird, ist zur Verarbeitung einer Nachricht vom Typ **Message** mit ihren potentiellen zahlreichen Details einige Sachkunde erforderlich. Eine solche Nachricht wird an die **Handler**-Methode **handleMessage()** übergeben, und zur angemessenen Behandlung muss diese Methode überschrieben werden. Das ist in einer eigenen, von **Handler** abgeleiteten Klasse möglich:

```
private MyHandler handler;
.
.
.
private static class MyHandler extends Handler {
    .
    .
    @Override
    public void handleMessage(Message message) {
        .
        .
        super.handleMessage(message);
    }
}
.
.
.
handler = new MyHandler(this);
```

Alternativ kann eine Klasse mit beliebigem Stammbaum (z.B. eine Aktivität) das in der Klasse **Handler** definierte Interface **Handler.Callback**

```
public interface Callback {
    public boolean handleMessage(Message msg);
}
```

implementieren und als Parameter für den **Handler**-Konstruktor zum Einsatz kommen (siehe Göransson 2014, S. 67).

## 10.6 Übungsaufgaben zu Kapitel 10

1) Welche von den folgenden Aussagen sind richtig bzw. falsch?

1. Reagiert der UI-Thread 5 Sekunden lang nicht auf eine Benutzereingabe, erscheint die ANR-Fehlermeldung. Kürzere Blockaden sind akzeptabel.
2. Ein **AsyncTask**-Objekt kann nur einmal ausgeführt werden.
3. Wenn Android eine Aktivität zerstört (z. B. bei einem Orientierungswechsel), dann werden alle von der Aktivität gestarteten Threads automatisch beendet.
4. Ruft man für einen Thread die Methode **interrupt()** auf, dann wird der Thread von Android beendet.
5. Ruft man für eine asynchrone Aufgabe (für ein Objekt einer **AsyncTask**-Ableitung) die Methode **cancel()** auf, dann wird die Aufgabe von Android beendet.
6. In einer App werden alle **execute()** - Aufrufe an irgendein **AsyncTask**-Objekt im selben Thread ausgeführt (also nacheinander).



---

## 11 Dienste

Funktionalitäten einer App an, die auch dann erledigt werden sollen, wenn sich die App *nicht* im Vordergrund befindet (z. B. GPS-Aufzeichnung während einer Wanderung, Datensicherung in die Cloud, (Internet-)Radio oder Abspielen von Musik), müssen über Dienste realisiert werden. Es sind sogar Android-Apps möglich und sinnvoll, die überhaupt keine Activity (keine Bedienoberfläche) besitzen, sondern ausschließlich Services, Broadcast Receiver und Content Provider.<sup>1</sup> Eine solche App kann sich z. B. per Broadcast Receiver über den Systemstart (also über die durchgeführte Aktion `android.intent.action.BOOT_COMPLETED`) informieren lassen besitzen und daraufhin einen Dienst starten.

Zwar kann auch eine im Hintergrund befindliche Aktivität weiterarbeiten, doch ist ein Hintergrundprozess bei einer Ressourcen-Verknappung weit eher von der Schließung bedroht als ein Dienstprozess (vgl. Abschnitt 5.4). Wir werden im Primzahlenbeispiel dafür sorgen, dass die Primzahlendiagnose in einem Dienst ausgeführt wird. Damit steigt die Chance dafür, dass eine Diagnose fortgesetzt wird, nachdem die initiiierende Aktivität in den Hintergrund geraten ist. Außerdem wird die Primzahlendiagnose vom Lebenszyklus der initiiierenden Aktivität entkoppelt und läuft auch dann weiter, wenn die Aktivität z. B. wegen eines Konfigurationswechsels zerstört und neu gestartet wird. Ein Dienst läuft bei einem Konfigurationswechsel ungestört weiter.

Soll in einer Aktivität eine zeitaufwändige, aus dem UI-Thread herauszuhaltende Aufgabe ausgeführt werden, bestehen zwei Optionen:

- Direkte Verwendung eines zusätzlichen Threads (siehe Kapitel 10)
- Beauftragung eines Dienstes, der schließlich die Aufgabe in einem Extra-Thread abwickelt

Für die Wahl des Lösungswegs gilt die folgende Empfehlung: Die aufwändigere Lösung per Dienst ist zu bevorzugen, wenn die Hintergrundbearbeitung auch dann fortgeführt werden soll, wenn die Aktivität nicht mehr mit dem Benutzer interagieren kann.

Im Regelfall wird ein Dienst im Prozess der Anwendung ausgeführt und folglich als **lokaler Dienst** (engl.: *local service*) bezeichnet. Er startet im Haupt-Thread der Anwendung, muss also unbedingt einen zusätzlichen Thread einsetzen, um eine Behinderung der Bedienoberfläche zu vermeiden. Seltener kommt ein **entfernter Dienst** zum Einsatz (engl.: *remote service*), der in einem eigenen Prozess oder im Prozess einer anderen App ausgeführt wird. Wir beschränken uns auf den lokalen Dienst.

Eine weitere Unterscheidung betrifft die Betriebsart von Diensten:

- **Vordergrunddienst**  
Ein Vordergrunddienst verrichtet eine für den Benutzer wahrnehmbare Tätigkeit (z. B. Abspielen von Musik) und besitzt ein permanent erreichbares Bedienelement in der Benachrichtigungszone von Android.
- **Gestarteter Dienst (Hintergrunddienst)**  
Ein über die **Context**-Methode `startService()` aktivierter Dienst kann unter einer Android-Version < 8 im Hintergrund potentiell unbegrenzt lange existieren, also auch seinen Urheber überleben. Mögliche Gründe für seine Beendigung:

---

<sup>1</sup> Wegen solcher Apps, die natürlich auch einen Haupt-Thread besitzen, sollte man eigentlich nicht von *UI-Thread* einer App sprechen.

- Der Dienst beendet sich selbst über die **Service**-Methode **stopSelf()**, nachdem er seine Aufgabe (z. B. einen Datei-Download) erledigt hat.
- Der Dienst wird von einer anderen Anwendungskomponente über die **Context**-Methode **stopService()** beendet.
- Der Dienst wird (samt Prozess) wegen Speichermangel vom System beendet.

Ein gestarteter Dienst bietet *kein* API mit Methoden an, die durch andere Komponenten aufgerufen werden können. Er erledigt in der Regel nur *eine* Aufgabe (z. B. Daten in eine Datenbank sichern) und beendigt sich anschließend. In besonders einfachen Fällen ist keine Ergebnisrückmeldung an den Aufrufer erforderlich. Um Behinderungen der Vordergrundanwendung zu vermeiden und den Energieverbrauch zu reduzieren, bestehen seit Android 8 Einschränkungen für gestartete Dienste (siehe Abschnitt 11.2.1).

- **Gebundener Dienst**

Über die **Context**-Methode **bindService()** kann sich ein Klient (z. B. eine Aktivität) an einen Dienst binden und diesen *Server* bis zum expliziten Lösen der Verbindung durch **unbindService()** nutzen. Ein gebundener Dienst bietet ein API mit Methoden an, die durch Klienten aufgerufen werden können. Als Klient kommt eine Aktivität, ein anderer Dienst oder ein Content Provider in Frage (nicht aber ein Broadcast Receiver). Es ist erlaubt, dass mehrere Klienten simultan mit einem Dienst verbunden sind. Befinden sich Klient und Server in derselben Anwendung und im selben Prozess, dann kann der Klient die (öffentliche oder im selben Paket) sichtbaren Methoden des Dienstes aufrufen. Es ist aber auch eine Kommunikation über Prozessgrenzen möglich (*inter process communication, IPC*). Ein gebundener Dienst wird automatisch vom System gestoppt, ...

- wenn kein Klient mehr verbunden ist,
- wenn ein drastischer Speichermangel besteht

In der Praxis werden die Unterschiede zwischen einem gestarteten und einem gebundenen Dienst etwas verwischt:

- Ein Dienst kann das Starten unterstützen *und* zusätzlich auch Bindungen erlauben. Dazu ist es lediglich erforderlich, die *beiden* folgenden Rückrufmethoden zu implementieren:
  - **onStartCommand()**  
Diese Rückrufmethode ist involviert, wenn ein Dienst über die **Context**-Methode **startService()** durch andere Komponenten gestartet wird.
  - **onBind()**  
Diese Rückrufmethode ist involviert, wenn sich andere Komponenten über die **Context**-Methode **bindService()** mit einem Dienst verbinden.
- Auch ein gestarteter Dienst ohne Bindungsmöglichkeit kann Ergebnisse an einen Aufrufer melden. Er unterstützt jedoch keinen Methodenaufruf durch Klienten (stellt kein API zur Verfügung).

Wenn ein Dienst zunächst per **startService()** gestartet wird und später auch noch Bindungen per **bindService()** erhält, dann müssen für ein reguläres, nicht durch Speichermangel bedingtes, Ende folgende Bedingungen erfüllt sein:

- Alle Bindungen sind gelöst.
- Der Dienst erhält eine Aufforderung zum Beenden (**stopSelf()** oder **stopService()**).

Ein Dienst muss wie jede andere Anwendungskomponente in der Manifestdatei einer Anwendung deklariert werden.

Basisklasse für alle Dienste ist die Klasse **Service** (im Paket **android.app**). Nach Möglichkeit sollten aber die im SDK bereits vorhandenen Ableitungen (z. B. **IntentService**) als Basis für eigene Klassendefinitionen verwendet werden.

## 11.1 Lebenszyklusmethoden bei Diensten

Generell sind bei einem Dienst die folgenden Lebenszyklusmethoden aus der Klasse **Service** (im Paket **android.app**) relevant:

- **public void onCreate()**  
In dieser Methode werden einmalig erforderliche Initialisierungen vorgenommen. Sie wird für *jeden* Dienst aufgerufen (unabhängig von der Betriebsart).
- **public int onStartCommand(Intent intent, int flags, int id)**  
Diese Methode wird aufgerufen, nachdem eine andere Komponente den Dienst mit **startService()** gestartet hat. Ein gestarteter Dienst läuft (abgekoppelt vom Lebenszyklus einer initiiерenden Komponente) potentiell unbegrenzt lange (zumindest vor Android 8, siehe Abschnitt 11.2.1). Der Dienst muss sich entweder selbst beenden über die **Service**-Methode **stopSelf()**, oder er muss durch eine andere Komponente beendet werden über die **Context**-Methode **stopService()**. Soll für einen Dienst nur die *gebundene* Betriebsart möglich sein, muss **onStartCommand()** nicht implementiert werden.
- **public abstract IBinder onBind(Intent intent)**  
Diese Methode wird aufgerufen, nachdem eine andere Komponente mit **bindService()** die Verbindungsaufnahme eingeleitet hat. Durch das von **onBind()** ablieferierte Objekt, das die Schnittstelle **IBinder** erfüllt, wird es der initiiерenden Komponente ermöglicht, Methoden des Dienstes aufzurufen (siehe Abschnitt 11.3.3 zu den Details). Als abstrakte Methode muss **onBind()** implementiert werden, kann sich jedoch bei einem Dienst ohne Bindungsoption auf die Rückgabe von **null** beschränken.
- **public void onDestroy()**  
Die Methode **onDestroy()** ist der richtige Ort für Aufräumungsarbeiten (z. B. Threads beenden, Broadcast Receiver oder Ereignisempfänger abmelden). Sie wird für *jeden* Dienst aufgerufen (unabhängig von der Betriebsart).

## 11.2 Gestarteter Dienst aus der Klasse IntentService

An unseren bisherigen Lösungen zur Primzahlendiagnose in einem Hintergrund-Thread (siehe Kapitel 10) ist u.a. zu bemängeln, dass bei einem Orientierungswechsel und dem damit verbundenen Neustart der Aktivität eine laufende Diagnose, die durchaus mehrere Minuten dauern kann, abgebrochen wird. Über ein **IntentService**-Objekt gelingt es, die im Hintergrund laufende Aufgabenbearbeitung vom Lebenszyklus der Aktivität zu entkoppeln. Ein Dienst übersteht den Orientierungswechsel unbeschadet.

Über den aktuellen Abschnitt hinausgehende Informationen zur Nutzung der Klasse **IntentService** bietet z. B. die Online-Dokumentation für Android-Entwickler.<sup>1</sup>

### 11.2.1 Mit Android 8 eingeführte Restriktionen für gestartete Dienste

Gestartete Dienste stehen mittlerweile im Verdacht, CPU-Zeit und Energie zu verschwenden und so die im Vordergrund befindliche App zu behindern sowie die Akku-Laufzeit zu reduzieren. Davon ist auch die Klasse **IntentService** betroffen. Google empfiehlt daher, gestartete Dienste nach Möglichkeit durch sogenannte **geplante Aufgaben** zu ersetzen. Auf der Webseite über geplante Aufgaben in Android heißt es recht deutlich:<sup>2</sup>

Use started services only as a last resort. The Android platform may not support started services in the future.

<sup>1</sup> <https://developer.android.com/training/run-background-service/index.html>  
<https://developer.android.com/guide/components/services.html>

<sup>2</sup> <https://developer.android.com/topic/performance/scheduling>

Seit Android 8 ist einer App das Starten und Verwenden von Diensten nur noch erlaubt, wenn sie sich im Vordergrund befindet oder erst vor einigen Minuten in den Hintergrund verdrängt wurde. Nach Ablauf der Gnadenfrist von einigen Minuten betrachtet Android die App als idle (dt.: *im Leerlauf befindlich*). Von ihr gestartete Hintergrunddienste werden angehalten, so als ob sie einen Aufruf der **Service**-Methode **stopSelf()** erhalten hätten.<sup>1</sup>

While an app is in the foreground, it can create and run both foreground and background services freely. When an app goes into the background, it has a window of several minutes in which it is still allowed to create and use services. At the end of that window, the app is considered to be *idle*.

Von diesen Einschränkungen sind zunächst nur Apps mit einer **targetSdkVersion** ab 26 (Android 8) betroffen. Allerdings bietet die Einstellungs-App von Android 8 dem Benutzer die Möglichkeit, die Restriktionen für beliebige Apps zu aktivieren.

### 11.2.2 Besonderheiten der Klasse IntentService

Für einfache Hintergrundaufgaben empfiehlt Google die von **Service** abgeleitete Klasse **IntentService**, die im Vergleich zu ihrer Basisklasse einige Vorteile bietet:

- Die Arbeitsaufträge werden automatisch in einem eigenen Thread ausgeführt, sodass es nicht zur Behinderung des UI- bzw. Main-Threads kommt.
- Ein **IntentService**-Objekt beendet sich selbst, wenn es seine Arbeit erledigt hat.
- Es ist explizit erwünscht, *ausschließlich* die in **IntentService** definierte Rückrufmethode **onHandleIntent()** zu überschreiben. Die in der Basisklasse **Service** definierten und für einen Dienst generell relevanten Rückrufmethoden **onCreate()**, **onStartCommand()**, **onBind()** und **onDestroy()** (vgl. Abschnitt 11.1) sollen explizit *nicht* überschrieben werden. Wer sie doch überschreibt, muss (außer bei **onBind()**) unbedingt die Basisklassenvariante aufrufen.

Von der strikt empfohlenen Zurückhaltung beim Überschreiben von geerbten Rückrufmethoden ist auch die bei gestarteten Diensten generell obligatorische Methode **onStartCommand()** betroffen:

**public int onStartCommand(Intent intent, int flags, int id)**

Das System ruft diese Methode auf, wenn eine Komponente mit **startService()** beantragt hat, den Dienst zu starten. Android übergibt an **onStartCommand()** den initiierenden Intent und erfährt per Rückgabewert, wie nach einem durch Speichermangel erzwungenen Prozessende verfahren werden soll. Für wichtige Rückgabewerte sind Konstanten in der Klasse **Service** definiert:

- **START\_STICKY** (int-Wert: 1)  
Der Prozess wird neu gestartet, sobald ausreichend Speicher verfügbar ist, erhält aber einen **onStartCommand()** - Aufruf mit dem Wert **null** für den Parameter **Intent**.
- **START\_NOT\_STICKY** (int-Wert: 2)  
Der Prozess wird *nicht* neu gestartet.
- **START\_REDELIVER\_INTENT** (int-Wert: 3)  
Der Prozess wird neu gestartet, sobald ausreichend Speicher verfügbar ist, und erhält einen **onStartCommand()** - Aufruf mit dem letzten **Intent**-Objekt.

Die **onStartCommand()** - Überschreibung der Klasse **IntentService** liefert per Voreinstellung die Rückgabe **START\_NOT\_STICKY**. Über die **IntentService**-Methode **setIntentRedelivery()**

**public void setIntentRedelivery(boolean enabled)**

---

<sup>1</sup> <https://developer.android.com/about/versions/oreo/background>

lässt sich aber erreichen, dass **onStartCommand()** die alternative Rückgabe **START\_REDELIVER\_INTENT** liefert.

Die **onBind()** - Überschreibung der Klasse **IntentService** liefert die Rückgabe **null**, sodass keine Bindung möglich ist. Um dieses in der Regel erwünschte Verhalten zu ändern, muss man **onBind()** überschreiben.

Neben der Rückrufmethode **onHandleIntent()** muss in einer **IntentService**-Ableitung noch ein öffentlicher und parameterfreier Konstruktor erstellt werden, sodass sich der Aufwand in engen Grenzen hält.

Wie es bei einem hohen Automatisierungsgrad zu erwarten ist, bestehen bei **IntentService** Einschränkungen im Vergleich zur generellen **Service**-Klasse:

- Es kann nur *ein* Arbeits-Thread genutzt werden. Dieser führt die Aufträge nacheinander aus.
- Eine laufende Auftragsbearbeitung kann nicht gestoppt werden. Ein Aufruf der **Context**-Methode **stopService()** für einen **IntentService** bleibt folgenlos.

Statt **IntentService** die generellere Klasse **Service** zu überben, um eine für den gestarteten Betrieb geeignete Dienstklasse zu definieren, lohnt sich vor dann, ...

- wenn mehrere Aufträge simultan abgewickelt werden sollen,<sup>1</sup>
- wenn es möglich sein soll, einen laufenden Auftrag abzubrechen.

### 11.2.3 IntentService in ein Projekt aufnehmen

Wir starten mit der ursprünglichen Version des Primzahlenprogramms im Ordner

...|BspUeb|PrimeDetection

und ergänzen im Layout eine Fortschrittsanzeige (vgl. Abschnitt 8.5.6)

```
<ProgressBar  
    android:id="@+id/progressBar"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    style="@android:style/Widget.ProgressBar.Horizontal"  
    android:layout_marginLeft="32dp"  
    android:layout_marginRight="32dp"  
    android:visibility="gone" />
```

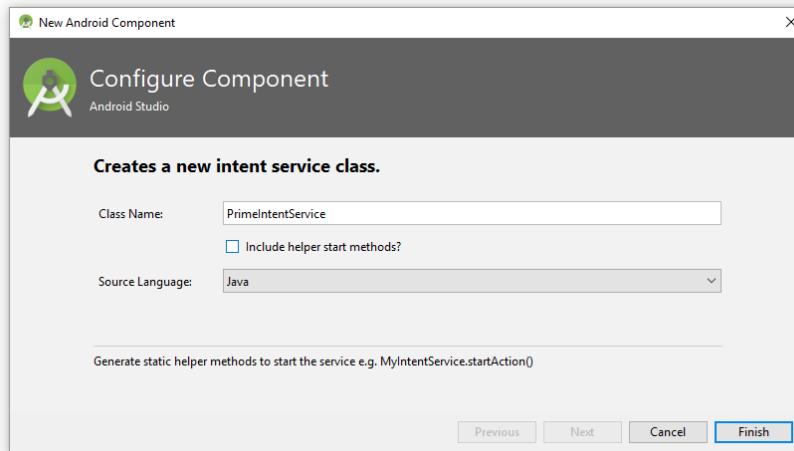
Um die Hilfe der Entwicklungsumgebung bei der Definition der Service-Klasse in Anspruch zu nehmen, wählen wie aus dem Kontextmenü zum Paketeintrag im Projektexplorer den Befehl:

**New > Service > Service (IntentService)**

Wir legen einen Klassennamen fest und verzichten auf die **helper start methods**:

---

<sup>1</sup> Hier ist ein Beispiel der Firma Google zu finden: <https://developer.android.com/guide/components/services>



Nach **Finish** wird die Klassendatei angelegt und ein Manifesteintrag erstellt:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="de.zimkand.primedetection">
    <application . . .
        <activity android:name=".PrimeActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <service
            android:name=".PrimeIntentService"
            android:exported="false"></service>
    </application>
</manifest>
```

Obligatorisch ist lediglich das Attribut **android:name**, das den Namen der realisierenden Klasse erhält. Sind zu einem Dienst im Manifest keine Intent-Filter vorhanden (empfohlen!), dann hat das Attribut **android:exported** ohnehin die Voreinstellung **false**.

Aus der Klassendefinition entfernen wir alle Assistentenkreationen bis auf den obligatorischen Konstruktor, der durch den Aufruf der Basisklassenvariante für eine Benennung des automatisch angelegten Threads sorgt, und den Rumpf der obligatorischen Überschreibung von **onHandleIntent()**:

```
package de.zimkand.primedetection;

import android.app.IntentService;
import android.content.Intent;

public class PrimeIntentService extends IntentService {
    public PrimeIntentService() {
        super("PrimeIntentService");
    }

    @Override
    protected void onHandleIntent(Intent intent) {
    }
}
```

Wir realisieren in **onHandleIntent()** den bekannten Algorithmus zur Primzahlensuche, wobei der Kandidat aus dem per Parameter gelieferten **Intent**-Objekt entnommen wird. Zur Ergebniskommunikation wird der in Abschnitt 11.2.5 beschriebene **LocalBroadcastManager** verwendet:

```

@Override
protected void onHandleIntent(Intent intent) {
    String strCand = intent.getExtras().getString("CAND");

    Context context = this;
    LocalBroadcastManager locBcMan = LocalBroadcastManager.getInstance(this);
    Intent returnIntent = new Intent(PrimeActivity.BROADCAST_ACTION);
    Bundle messBundle = new Bundle();

    boolean df = false;
    long cand, i, mdc, widthSeg;
    int nSeg = 100;
    if (strCand.length() == 0)
        return;
    cand = Long.parseLong(strCand);
    mdc = (int) Math.sqrt(cand);
    widthSeg = Math.max(1000, mdc/nSeg);
    for (i = 2; i <= mdc; i++) {
        if (i%widthSeg == 0){
            messBundle.putFloat("progress", i/(float)mdc*100);
            returnIntent.putExtras(messBundle);
            locBcMan.sendBroadcast(returnIntent);
        }
        if (cand % i == 0) {
            df = true;
            break;
        }
    }
    String res;
    if (cand <= 1)
        res = strCand + " " + context.getString(R.string.falseArgument);
    else if (df)
        res = strCand + " " + context.getString(R.string.noPrime) + "\n" +
              context.getString(R.string.divisor) + Long.toString(i)+ ")";
    else
        res = strCand + " " + context.getString(R.string.prime);
    messBundle.putFloat("progress", 0.0f);
    messBundle.putString("result", res);
    returnIntent.putExtras(messBundle);
    locBcMan.sendBroadcast(returnIntent);
}

```

#### 11.2.4 Dienst starten und stoppen

Gestartet wird der **IntentService** in der Click-Behandlungsmethode der Aktivität über ein explizites **Intent**-Objekt mit der Extraintformation über den zu untersuchenden Wert:

```

@Override
public void onClick(View view) {
    String s = candidate.getText().toString();
    result.setText("");
    button.setEnabled(false);
    progressBar.setProgress(0);
    progressBar.setVisibility(ProgressBar.VISIBLE);
    Intent intent = new Intent(this, PrimeIntentService.class);
    intent.putExtra("CAND", candidate.getText().toString());
    startService(intent);
}

```

Der **startService()** - Aufruf kehrt sofort zurück und veranlasst Android, ...

- nötigenfalls Falls ein Objekt der Dienstklasse zu erstellen und seine Methode **onCreate()** aufzurufen,
- die **onStartCommand()** - Methode des Dienstobjekts aufzurufen.

Bei einem **IntentService**-Objekt wird anschließend die Methode **onHandleIntent()** aufgerufen, um den Auftrag zu erledigen.

Im Allgemeinen kommen folgende Anlässe für die Beendigung eines gestarteten Dienstes in Frage:

- Der Dienst beendet sich nach Erledigung seines Auftrags selbst durch die **Service**-Methode **stopSelf()**.
- Der Dienst wird von einer anderen Anwendungskomponente über die **Context**-Methode **stopService()** beendet.
- Der Dienst wird von Android wegen Speichermangel beendet.
- Seit Android 8 werden die von einer App gestarteten Dienste beendet, wenn sich die App seit mehreren Minuten im Hintergrund befindet (siehe Abschnitt 11.2.1).

Bei einem Objekt der Klasse **IntentService** müssen wir uns nicht um die Terminierung kümmern, weil sich ein **IntentService** nach Erledigung seines Auftrags selbst stoppt, und außerdem eine laufende Auftragsbearbeitung nicht unterbrochen werden kann.

Generell kann und soll man sich bei einem Objekt der Klasse **IntentService** darauf beschränken, die Methode **onHandleIntent()** zu überschreiben, die automatisch in einem eigenen Thread ausgeführt wird. Außerdem wird ein öffentlicher und Parameterfreier Konstruktor benötigt.

Anschließend wird die im Beispiel benutzte Ergebniskommunikation per **LocalBroadcastManager** beschrieben.

### 11.2.5 Ergebniskommunikation per LocalBroadcastManager

Google empfiehlt, in einem gestarteten Dienst zur Meldung von Ergebnissen an die beauftragende Komponente einen **LocalBroadcastManager** zu verwenden, sofern nur Komponenten in der eigenen App informiert werden sollen.<sup>1</sup>

Wir erhalten ein **LocalBroadcastManager**-Objekt über die statische Methode **getInstance()** der Klasse **LocalBroadcastManager**:

```
LocalBroadcastManager locBcMan = LocalBroadcastManager.getInstance(this);
```

Als Transportvehikel für die zu übertragende Information wird ein implizites **Intent**-Objekt mit einer selbst definierten Aktion versendet:

```
Intent returnIntent = new Intent(PrimeActivity.BROADCAST_ACTION);
```

Es wird ein **Bundle**-Objekt

```
Bundle messBundle = new Bundle();
```

mit Extrainformationen gefüttert und in das **Intent**-Objekt eingefügt, z. B.:

```
messBundle.putFloat("progress", i/(float)mdc*100);
returnIntent.putExtras(messBundle);
```

Schließlich wird der **LocalBroadcastManager** gebeten, das **Intent**-Objekt zu versenden:

```
locBcMan.sendBroadcast(returnIntent);
```

In unserem Beispiel soll eine Aktivität (ein Objekt der Klasse **PrimeActivity**) das per Rundruf versendete **Intent**-Objekt empfangen. Daher lässt die Aktivität in ihrer Lebenszyklusmethode **onCreate()** einen Broadcast Receiver beim **LocalBroadcastManager** registrieren,

---

<sup>1</sup> <https://developer.android.com/training/run-background-service/report-status.html>

```

@Override
protected void onCreate() {
    ...
    receiver = new PrimeEvalReceiver();
    IntentFilter intentFilter = new IntentFilter(BROADCAST_ACTION);
    LocalBroadcastManager.getInstance(this).registerReceiver(receiver, intentFilter);
}

```

Im ersten Parameter der Methode **registerReceiver()** ist ein Objekt einer von **BroadcastReceiver** abstammenden Klasse zu übergeben. Wir definieren die Ableitung **PrimeEvalReceiver** als innere Klasse zur Aktivität und deklarieren eine Instanzvariable vom Typ **PrimeEvalReceiver**:

```

private PrimeEvalReceiver receiver;
...
private class PrimeEvalReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        Bundle bundle = intent.getExtras();
        float progress = bundle.getFloat("progress");
        if (progress != 0.0f) {
            button.setEnabled(false);
            progressBar.setVisibility(ProgressBar.VISIBLE);
            progressBar.setProgress((int)progress);
            return;
        }
        String ergebnis = bundle.getString("result");
        result.setText(ergebnis);
        button.setEnabled(true);
        progressBar.setVisibility(ProgressBar.GONE);
    }
}

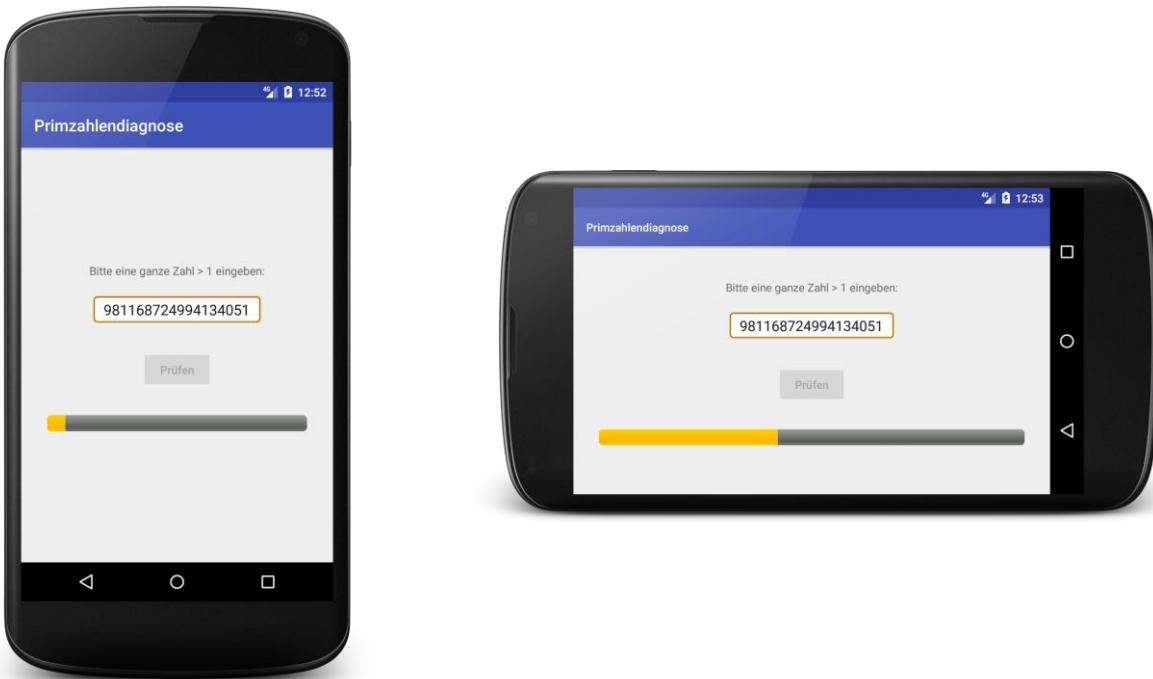
```

Die in der Basisklasse abstrakt definierte und damit obligatorisch zu überschreibende Methode **onReceive()** wertet das per Parameter erhaltene **Intent**-Objekt aus und nimmt entsprechende Änderungen an der Bedienoberfläche vor (Verfügbarkeit des Befehlsschalters **button**, Ergebnisanzeige im **TextView**-Objekt **result**, Fortschrittsbalken **progressBar**). Den Fortschrittsbalken sichtbar zu machen und den Befehlsschalter zu deaktivieren, ist nach einem Aktivitätsneustart und bereits laufender Primzahlendiagnose relevant.

Im zweiten Parameter von **registerReceiver()** ist ein **IntentFilter**-Objekt (vgl. Abschnitt 9.2) zu übergeben, wobei wir uns auf die Angabe der Broadcast-Aktion des **Intent**-Objekts beschränken können, das der **LocalBroadcastManager** in der Klasse **PrimeIntentService** versendet.

Im aktuellen Beispiel machen wir übrigens erste Erfahrungen mit dem Broadcast Receiver, einer noch nicht offiziell behandelten Android-Komponente (siehe Kapitel 16).

Als Ergebnis der Mühe läuft nun im Beispielprogramm die Primzahlendiagnose in einem Hintergrund-Thread, der von einem Service gestartet wird. Während die im Vordergrund befindliche Aktivität bei einem Orientierungswechsel zerstört und neu gestartet wird, läuft der Dienst unbeschadet weiter, und es gibt keinen Grund, seinen Extra-Thread zu stoppen. Weil die neu gestartete Aktivität einen Broadcast Receiver mit demselben **IntentFilter** registriert, erfährt sie die aktuellen Ergebnisse des im Hintergrund ungestört tätigen Threads und zeigt diese an, sodass der Benutzer eine aufwändige Diagnose *nicht* neu starten muss:



Im Beispiel wird der Broadcast Receiver nur beim Zerstören der Aktivität über die Methode **unregisterReceiver()** abgemeldet:

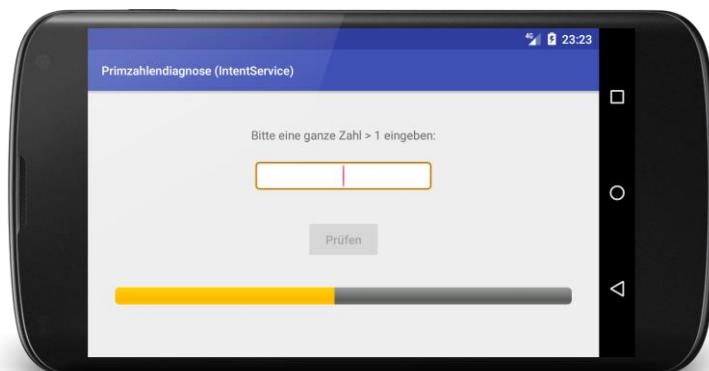
```
@Override
protected void onDestroy() {
    super.onDestroy();
    LocalBroadcastManager.getInstance(this).unregisterReceiver(receiver);
}
```

Im gestoppten Zustand empfangene Aktualisierungen sorgen nach der Rückkehr in den sichtbaren Zustand für eine aktuelle Anzeige (z. B. mit dem im Hintergrund erarbeiteten Prüfergebnis).

Bei einem **IntentService** stellt sich die Frage *nicht*, ob der Dienst mit **stopService()** beendet werden soll, wenn der Benutzer z. B. die initiiierende Aktivität per Rückwärtsschalter verlässt:

```
if (!isChangingConfigurations()) {
    stopService(new Intent(this, PrimeIntentService.class));
}
```

Eine Einschränkung der Klasse **IntentService** besteht bekanntlich darin, dass sich eine laufende Bearbeitung nicht abbrechen lässt. Das führt zu einem unerwünschten Verhalten des Beispielprogramms, wenn es vom Benutzer per Rückwärtstaste verlassen wird. In diesem Fall wird die Aktivität zerstört, doch der nicht zu stoppende **IntentService** läuft weiter. Startet der Benutzer das Programm anschließend neu, empfängt der Broadcast Receiver die Nachrichten des weiterhin aktiven Dienstes, und der Benutzer sieht einen unmotivierten Fortschrittsbalken, z. B.:



Das komplette AS-Projekt mit der im aktuellen Abschnitt beschriebenen Prime Detection - Variante ist im folgenden Ordner zu finden:

...\\BspUeb\\Service\\IntentService

### 11.3 Gebundener Dienst

Obwohl die Klasse **IntentService** für das Primzahlenprogramm gut geeignet ist, ersetzen wir sie nun zu Übungszwecken durch einen gebundenen Service und lernen dabei ein nicht ganz triviales Framework kennen. Ein Dienst für die gebundene Betriebsart macht deutlich mehr Implementationsaufwand als ein Dienst für die gestartete Betriebsart. Unser Beispiel bleibt allerdings hinter den Möglichkeiten eines gebundenen Dienstes zurück:

- Es findet keine Nutzung durch *mehrere* Klienten statt.
- Der einzige Klient erteilt keine simultan zu bearbeitenden Aufträge.

Gebundene Dienste eignen sich für folgende Einsatzszenarien:

- Andere Komponenten sollen die Möglichkeit zur Interaktion mit dem Dienst haben. Damit ist gemeint, dass Klienten nicht (wie beim Service-Start) einen Auftrag einreichen und eventuell irgendwann eine Rückmeldung erhalten, sondern permanent Methoden des Dienstes aufrufen können, die i.A. eine Rückgabe liefern.
- Der Dienst soll über IPC (*interprocess communication*) auch für Komponenten anderer Apps nutzbar sein.

Ein typisches Beispiel für den zuerst genannten Einsatzzweck ist eine App mit einem gebundenen Dienst, der Musikstücke abspielt, und einer Activity, die sich mit dem Dienst verbindet und dann diverse Methoden des Dienstes mit sofortiger Ausführung und Rückmeldung aufrufen muss, z. B.:

- Titel des aktuellen Musikstücks abfragen und wählen
- Lautstärke abfragen und einstellen
- Klangfarbe abfragen und einstellen

Dieses Einsatzszenario unterscheidet sich deutlich vom eines gestarteten Dienstes, der z. B. den Auftrag erhält, eine Datei zu einem Cloud-Server zu übertragen.

Ein gebundener Dienst wird vom System gestoppt, ...

- wenn kein Klient mehr verbunden ist  
Ein Klient kann seine bestehende Bindung durch einen Aufruf der **Context-Methode unbindService()** beenden.
- wenn ein gravierender Speichermangel besteht

#### 11.3.1 Rohling für einen bindungsfähigen Dienst per Entwicklungsumgebung erstellen

Wir starten erneut mit der ursprünglichen Version des Primzahlenprogramms im Ordner

...\\BspUeb\\PrimeDetection

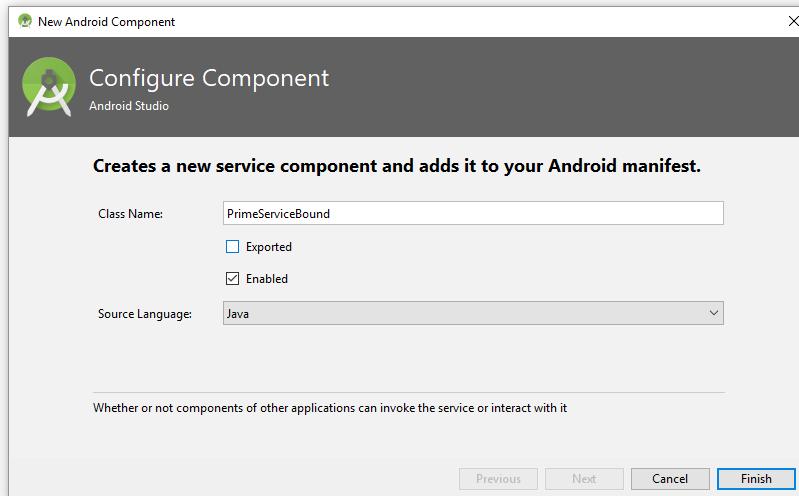
und ergänzen im Layout eine Fortschrittsanzeige (vgl. Abschnitt 8.5.6)

```
<ProgressBar
    android:id="@+id/progressBar"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    style="@android:style/Widget.ProgressBar.Horizontal"
    android:layout_marginLeft="32dp"
    android:layout_marginRight="32dp"
    android:visibility="gone" />
```

Um die Starthilfe der Entwicklungsumgebung bei der Definition der Service-Klasse in Anspruch zu nehmen, wählen wie aus dem Kontextmenü zum Paketeintrag im Projektexplorer den Befehl:

### New > Service > Service

Wir legen einen Klassennamen fest und entfernen die Markierung beim Kontrollkästchen **Exported**, sodass der Service von fremden Apps *nicht* genutzt werden kann (auch nicht über explizite Intents):



Nach **Finish** wird die Quellcodedatei der neuen Klasse angelegt und ein Manifesteintrag für den Dienst erstellt:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="de.zimkand.primedetection">

    <application . . .
        <activity android:name=".PrimeActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <service
            android:name=".PrimeServiceBound"
            android:enabled="true"
            android:exported="false"></service>
    </application>
</manifest>
```

Obligatorisch ist lediglich das Attribut **android:name**, das den Namen der realisierenden Klasse erhält. Das Attribut **android:enabled** mit der Voreinstellung **true** legt fest, ob ein Dienst verfügbar ist, d.h. ob Android auf Anforderung Objekte des Dienstes instanzieren soll. Die Einstellung kann mit Hilfe der Klasse **PackageManager** per Programm geändert werden. Sind zu einem Dienst (wie empfohlen!) im Manifest keine Intent-Filter vorhanden, dann hat das Attribut **android:exported** ohnehin die Voreinstellung **false**. Mit der Definition von Berechtigungen (Permissions) zur Benutzung eines Dienstes werden wir uns später beschäftigen.

### 11.3.2 Bindung herstellen und aufheben

Um ein **Service**-Objekt einzubinden, wird ein **Intent**-Objekt und ein Aufruf der **Context**-Methode **bindService()** benötigt:

```
public boolean bindService(Intent service, ServiceConnection connection, int flags)
```

Im Beispiel bindet sich die Aktivität des Primzahlenprogramms in ihrer Lebenszeitmethode **onCreate()** an das **Service**-Objekt aus der Klasse **PrimeServiceBound**:

```
Intent intent = new Intent(this, PrimeServiceBound.class);
bindService(intent, serviceConnection, Context.BIND_AUTO_CREATE);
```

Wir übergeben an **bindService()** ein explizites **Intent**-Objekt sowie ein **ServiceConnection**-Objekt (siehe unten) und signalisieren im dritten Parameter mit der **int**-werten Konstanten **Context.BIND\_AUTO\_CREATE**, dass Android den Dienst automatisch starten soll, falls er noch nicht läuft. Wenn dieses Flag fehlt, und der Service *nicht* läuft, bleibt es bei diesem inaktiven Zustand, und **bindService()** endet mit der Rückgabe **false**.

In der Lebenszyklusmethode **onDestroy()** wird die Verbindung zum Service durch einen Aufruf den **Context**-Methode **unbindService()** wieder aufgehoben:

```
@Override
protected void onDestroy() {
    super.onDestroy();
    if (!isChangingConfigurations())
        unbindService(serviceConnection);
}
```

Es gilt zu verhindern, dass bei einer Aktivitätszerstörung im Zusammenhang mit einem Konfigurationswechsel auch der (nicht an andere Komponenten gebundene) Dienst untergeht. Dabei könnte das aufwändig erarbeitete Zwischenergebnis einer Primzahlenprüfung verloren gehen. Ob der aktuelle Aufruf von **onDestroy()** auf einen Konfigurationswechsel zurückgeht, erfährt man durch einen Aufruf der **Activity**-Methode **isChangingConfigurations()**.

Android nimmt zur Laufzeit per Ausnahmefehler kritisch Stellung, wenn beim Ableben einer Activity eine Serviceverbindung besteht und nicht in **onStop()** oder **onDestroy()** aufgehoben wird, z. B.:

```
06-17 18:44:40.812 9667-9667/de.zimkand.primedetection E/ActivityThread:
Activity de.zimkand.primedetection.PrimeActivity has leaked ServiceConnection
```

Dieses „Fehlverhalten“ erlaubt sich das Beispielprogramm nur bei einem Orientierungswechsel. Wenn das nach dem Orientierungswechsel neu erstellte Activity-Objekt die Methode **bindService()** aufruft, wird es mit dem vorhandenen **Service**-Objekt verbunden, sodass kein Speicherleck (engl.: *memory leak*) vorliegt. Benutzer werden von der im **logcat**-Fenster der Entwicklungsumgebung erscheinenden Fehlermeldung *nicht* irritiert.

Wird ein Aktivitätsobjekt zerstört, das sich zuvor mit einem Dienst verbunden hat, wird die Bindung *nicht* automatisch aufgehoben. Das wäre bei einer Zerstörung wegen eines Orientierungswechsels auch nicht wünschenswert, weil dabei ein Dienst ohne weitere Bindung dabei ebenfalls zerstört würde.

In seinem Buch über Android 4 empfiehlt Allen (2012, S. 413ff) einen erheblichen Aufwand, um das Überleben einer Bindung während einer Konfigurationsänderung sicherzustellen:

- Seiner Meinung nach sollten die Methoden **bindService()** und **unbindService()** nicht vom Aktivitätsobjekt, sondern vom zugehörigen Kontext ausgeführt werden, z. B.:
 

```
getApplicationContext().bindService(intent, serviceConnection, Context.BIND_AUTO_CREATE);
```

 Die Forderung klingt plausibel, doch fehlt die Begründung, wie sich das ausführende Objekt auf den Effekt der Methoden **bindService()** und **unbindService()** auswirkt.
- Außerdem soll durch eine Überschreibung der bislang nicht diskutierten Aktivitäts-Lebenszyklusmethode
 

```
public Object onRetainNonConfigurationInstance()
```

 dafür gesorgt werden, dass **ServiceConnection**-Objekt aus der alten (durch Konfigurationswechsel untergehenden) Aktivitätsinstanz in die neue Instanz zu übertragen, damit von der neuen Aktivitätsinstanz im **bindService()** - Aufruf dasselbe **ServiceConnection**-Objekt verwendet wird. Die Forderung klingt noch plausibler wie die obige, doch zeigt die bisherige Erfahrung, dass der erhebliche Aufwand überflüssig ist.

### 11.3.3 Klienten-Server - Kommunikation via Binder-Objekt und Handler

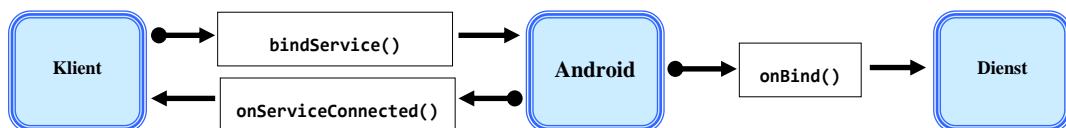
Von den möglichen Verfahren, einem Klienten den Aufruf von öffentlichen Methoden eines verbundenen Dienstes zu ermöglichen, behandeln wir nur die einfachste Variante, die mit einer Ableitung der Klasse **Binder** aus dem Paket **android.os** arbeitet.<sup>1</sup> Ihre Voraussetzungen sind meist erfüllt: Klient und Dienst müssen zur selben Anwendung gehören und im selben Prozess laufen. Für die Übertragung von Zwischen- und Endergebnissen zum Klienten verwenden wir eine Ableitung der Klasse **Handler** aus dem Paket **android.os**. Insgesamt sind folgende Typen beteiligt:

#### 11.3.3.1 ServiceConnection

In der Methode **bindService()** zur Verbindungsaufnahme (siehe oben) muss der Klient ein Objekt einer Klasse übergeben, welche die Schnittstelle **ServiceConnection** implementiert, also die Methode **onServiceConnected()** implementiert:

```
public void onServiceConnected(ComponentName className, IBinder service)
```

Sie wird aufgrund des klientenseitigen **bindService()** - Aufrufs beim Verbindungsauftakt vom System aufgerufen und erhält per Parameter ein Objekt vom Typ **IBinder**, das der Service in seiner Lebenszyklusmethode **onBind()** geliefert hat. Erst durch den Aufruf von **onServiceConnected()** erfährt der Klient von der etablierten Verbindung zum **Service**.



Wir definieren in unserer Aktivität eine innere Klasse, die das Interface **ServiceConnection** implementiert:

<sup>1</sup> <https://developer.android.com/guide/components/bound-services.html>

```

private MyServiceConnection serviceConnection;
. . .
private class MyServiceConnection implements ServiceConnection {
    @Override
    public void onServiceConnected(ComponentName className, IBinder service) {
        PrimeServiceBound.PrimeBinder binder = (PrimeServiceBound.PrimeBinder) service;
        primeService = binder.getService();
        bound = true;
        if (primeService.isRunning()) {
            button.setEnabled(false);
            primeService.setHandler(handler);
        }
    }

    @Override
    public void onServiceDisconnected(ComponentName component) {
        bound = false;
    }
}

```

In der Schnittstelle **ServiceConnection** ist neben **onServiceConnected()** noch die Methode **onServiceDisconnected()** vorgeschrieben. Über ihren Aufruf erfährt der Client davon, dass der Dienst nicht mehr zur Verfügung steht, z. B. weil er einen Laufzeitfehler verursacht hat oder wegen Speichermangels zerstört worden musste. Diese Methode wird *nicht* aufgerufen, nachdem ein Client mit **unbindService()** seine Bindung an den Dienst beendet hat. Bei einem App-internen Dienst ist nicht mit einem Aufruf der Methode **onServiceDisconnected()** zu rechnen, weil bei den genannten Ursachen die gesamte App betroffen ist.

Ob aktuell eine Serviceverbindung besteht, merkt sich das Aktivitätsobjekt in der **boolean**-Instanzvariablen **bound**.

### 11.3.3.2 Binder

Für die prozessinterne Kommunikation zwischen einem Client und einem gebundenen lokalen Dienst ist die **Binder**-Technik zu bevorzugen. Die **Service**-Lebenszyklusmethode **onBind()** liefert ein Objekt aus einer von **Binder** (im Paket **android.os**) abstammenden Klasse, das der Client als Parameter der Rückrufmethode **onServiceConnected()** erhält. Der Client kann das **Binder**-Objekt zum Aufruf von öffentlichen Methoden nutzen, die in der **Binder**- oder in der **Service**-Ableitung definiert sind. Android ruft die Methode **onBind()** nur bei der ersten Klientenanforderung (per **bindService()**) auf. Alle weiteren Klienten, die sich mit demselben Dienst verbinden möchten, erhalten ohne erneuten **onBind()** - Aufruf dasselbe **Binder**-Objekt. Meist kann sich die **Binder**-Ableitung darauf beschränken, in einer (z. B. als **getService()** bezeichneten) Methode die Adresse des **Service**-Objekts zu veröffentlichen, damit der Client anschließend die öffentlichen Methoden des Dienstes direkt aufrufen kann. Unsere **Service**-Klasse definiert die folgende von **Binder** abstammende innere Klasse **PrimeBinder**

```

public class PrimeBinder extends Binder {
    public PrimeServiceBound getService() {
        return PrimeServiceBound.this;
    }
}

```

und liefert in der Lebenszyklusmethode **onBind()** ein Objekt dieser Klasse:

```

@Override
public IBinder onBind(Intent intent) {
    return new PrimeBinder();
}

```

Ein Klient erhält das **PrimeBinder**-Objekt als Parameter der **ServiceConnection**-Methode **onServiceConnected()** (siehe Abschnitt 11.3.3.1).

### 11.3.3.3 Ergebniskommunikation über Message-Objekte

Unsere **Service**-Klasse soll Aufträge asynchron in einem eigenen Thread erledigen und benötigt daher eine Möglichkeit, den aktuellen Bearbeitungsstand sowie das Arbeitsergebnis auf zulässige Weise in der Bedienoberfläche darzustellen. Dazu erstellt unsere Aktivität eine Ableitung der Klasse **Handler**, die sich um die Nachrichtenwarteschlange des UI-Threads kümmert. Wir überschreiben die **Handler**-Methode **handleMessage()**, die beim Eintreffen einer Nachricht für den UI-Thread aufgerufen wird (vgl. Abschnitt 10.5). Somit haben wir die Möglichkeit, auf Nachrichten an den UI-Thread anwendungsadäquat zu reagieren:

```
private static class MyHandler extends Handler {
    private WeakReference<PrimeActivity> wRef;

    public MyHandler(PrimeActivity target) {
        wRef = new WeakReference<>(target);
    }

    @Override
    public void handleMessage(Message message) {
        PrimeActivity pa = wRef.get();
        if (pa != null)
            pa.updateUI(message);
    }
}
```

Die **Service**-Klasse erhält über ihre Methode **setHandler()**

```
public void setHandler(Handler handler) {
    this.handler = handler;
}
```

eine Referenz auf das **Handler**-Objekt und kann daher per **sendMessage()** Nachrichten an den UI-Thread verschicken. Diese führen zum Aufruf der **Handler**-Methode **handleMessage()** im UI-Thread, sodass sich die Botschaften des **Service**-Objekts unfallfrei auf die Bedienoberfläche der Aktivität auswirken können. Wir werden auf dem beschriebenen Kanal Aktualisierungen für den Fortschrittsbalken und das Endergebnis transportieren.

Es wird einiger Aufwand betrieben, um gewöhnliche („harte“) Referenzen auf das (z. B. bei einer Konfigurationsänderung von Vernichtung bedrohte) Aktivitätsobjekt aus **Handler**-Objekten herauszuhalten. Das **Handler**-Objekt ist dem **Service**-Objekt bekannt, das die **Handler**-Methode **sendMessage()** aufrufen muss. Wenn indirekt auch eine Referenz auf das Aktivitätsobjekt im Dienst enthalten wäre, könnte ein zerstörtes Aktivitätsobjekt nicht vom Garbage Collector abgeräumt werden. Es würde ein sogenanntes *Speicherleck* (engl.: *memory leak*) entstehen. Auf folgende Weise wird verhindert, dass eine „harte“ Referenz auf das Aktivitätsobjekt in das **Handler**-Objekt gelangt:

- Die **Handler**-Ableitung wird als *statische* Mitgliedsklasse definiert.
- Für den erforderlichen Zugriff auf die Aktivität wird ein Instanzobjekt aus der Klasse **WeakReference<PrimeActivity>** verwendet. Ein Objekt dieser Klasse kann eine Referenz auf das Aktivitätsobjekt aufbewahren, ohne die Beseitigung der Aktivität durch den Garbage Collector zu verhindern. Folglich wird in der Aktivitätsklasse eine Hilfsmethode benötigt:

```

private void updateUI(Message message) {
    Bundle bundle = message.getData();
    float progress = bundle.getFloat("progress");
    if (progress != 0.0f) {
        progressBar.setVisibility(ProgressBar.VISIBLE);
        progressBar.setProgress((int)progress);
    } else {
        String ergebnis = bundle.getString("result");
        result.setText(ergebnis);
        button.setEnabled(true);
        progressBar.setVisibility(ProgressBar.GONE);
    }
}

```

Wird das Risiko eines Speicherlecks missachtet, lässt sich die **Handler**-Ableitung einfacher realisieren, z.B.:

```

private class MyHandler extends Handler {
    This Handler class should be static or leaks might occur (de.zimkand.primedetection.PrimeActivity.MyHandler) more... (Strg+F1)
    public void handleMessage(Message message) {
        Bundle bundle = message.getData();
        float progress = bundle.getFloat("progress");
        if (progress != 0.0f) {
            progressBar.setVisibility(ProgressBar.VISIBLE);
            progressBar.setProgress((int)progress);
        } else {
            String ergebnis = bundle.getString("result");
            result.setText(ergebnis);
            button.setEnabled(true);
            progressBar.setVisibility(ProgressBar.GONE);
        }
    }
}

```

In dieser Situation warnt das Android Studio allerdings durch eine farbliche Hinterlegung des Klassennamens mit zugehöriger Einblendung vor dem Problem. Die Einblendung liefert der im Studio integrierte Code-Inspektor namens **lint**, der übrigens auch nach dem folgenden Menübefehl tätig wird:

#### Analyze > Inspect Code

Wir haben übrigens auch in Abschnitt 10.2 ein mit der Nachrichtenwarteschlange des UI-Threads verbundenes Objekt der Klasse **Handler** verwendet, um Steuerelemente aus einem anderen Thread heraus zu aktualisieren. Dort bestanden keine Memory Leak - Argumente gegen einen Hintergrund-Thread auf der Basis einer anonymen Klasse, sodass wir in der **run()** - Methode bequem über Variablen der Aktivitätsklasse auf die Steuerelemente zugreifen konnten. Folglich genügte die **Handler**-Methode **post()** für unsere Zwecke, und wir haben auf eine **Handler**-Ableitung mit Überschreibung der Methode **handleMessage()** verzichtet.

### 11.3.4 Service-Ableitung definieren

In der **Service**-Klasse **PrimServiceBound** zur Primzahlenanwendung überschreiben wir die Lebenszyklusmethoden **onBind()** und **onDestroy()**:

```
@Override
public IBinder onBind(Intent intent) {
    return new PrimeBinder();
}

@Override
public void onDestroy() {
    if (t != null)
        t.interrupt();
}
```

Die Methode **onBind()** wird vom System aufgerufen, wenn ein Client durch den Aufruf von **bindService()** versucht, den Dienst einzubinden. Durch eine Rückgabe vom Typ **IBinder** wird es dem Aufrufer ermöglicht, mit dem Dienst zu kommunizieren (siehe Abschnitt 11.3.3).

Nach Auflösung aller Bindungen wird ein ausschließlich gebundener (nicht gleichzeitig auch gestarteter) Dienst automatisch vom System zerstört. Bei Speichermando kann auch einem verbundenen Dienst dieses Schicksal widerfahren. Die Methode **onDestroy()** ist der richtige Ort für Aufräumungsarbeiten. Im Beispiel führen wir die Primzahlendiagnose in einem eigenen Thread aus, der in **onDestroy()** beendet werden muss, damit es nicht zu der in Abschnitt 10.3 beschriebenen Ansammlung von Altlast-Threads kommen kann.

Die Lebenszyklusmethode **onCreate()**, die beim Erstellen des Dienstes (noch vor **onBind()** und **onStartCommand()**) vom System aufgerufen wird, ist im Beispiel irrelevant und muss daher nicht überschrieben werden.

Damit der Primzahlendienst genutzt werden kann, erstellen wir die Methode **evalCandidate()**, die einen **Thread** zur Durchführung der Primzahlendiagnose startet:

```
public boolean evalCandidate(String cand) {
    if (running)
        return false;
    else {
        this.cand = cand;
        t = new Thread(this);
        t.start();
        return true;
    }
}
```

Außerdem kann der Dienst in der Methode **isRunning()** darüber informieren, ob er gerade mit einer Primzahlendiagnose beschäftigt ist:

```
public boolean isRunning() {
    return running;
}
```

In der vom **PrimServiceBound**-Objekt ausgeführten **run()** - Methode des Threads kommt der schon mehrfach verwendete Algorithmus zur Primzahlendiagnose zum Einsatz. Außerdem wird die Ergebnismeldung per **Handler** vorgenommen:

```

@Override
public void run() {
    running = true;
    Bundle bundle = new Bundle();
    Message message;
    boolean df = false;
    long cand, i, mdc, widthSeg;
    int nSeg = 100;
    if (this.cand.length() == 0)
        return;
    cand = Long.parseLong(this.cand);
    mdc = (int) Math.sqrt(cand);
    widthSeg = Math.max(1000, mdc / nSeg);
    for (i = 2; i <= mdc; i++) {
        if (Thread.currentThread().isInterrupted())
            return;
        if (i % widthSeg == 0 && (handler != null)) {
            bundle.putFloat("progress", i / (float) mdc * 100);
            message = new Message();
            message.setData(bundle);
            handler.sendMessage(message);
        }
        if (cand % i == 0) {
            df = true;
            break;
        }
    }
    String res;
    if (cand <= 1)
        res = this.cand + " " + getString(R.string.falseArgument);
    else if (df)
        res = this.cand + " " + getString(R.string.noPrime) + "\n(" +
              getString(R.string.divisor) + Long.toString(i) + ")";
    else
        res = this.cand + " " + getString(R.string.prime);
    if (handler != null) {
        bundle.putFloat("progress", 0.0f);
        bundle.putString("result", res);
        message = new Message();
        message.setData(bundle);
        handler.sendMessage(message);
    }
    running = false;
}

```

Es werden Objekte der Klassen **Message** und **Bundle** (beide im Paket **android.os**) erstellt:

```

Bundle bundle = new Bundle();
Message message;
. . .
message = new Message();

```

Während das **Bundle**-Objekt in der Methode `run()` mehrfach verwendet werden kann, hat es sich beim **Message**-Objekt als notwendig erwiesen, für jede Nachrichtenübermittlung an den UI-Thread ein neues Objekt zu verwenden.<sup>1</sup>

In das **Bundle**-Objekt werden Name-Wert-Paare per `putFloat()` bzw. `putString()` eingefügt, z. B.:

```
bundle.putFloat("progress", i/(float)mdc*100);
```

Per `setData()` wird das **Bundle** in die **Message** eingefügt,

---

<sup>1</sup> Bei Wiederverwendung desselben **Message**-Objekts kommt es zum Laufzeitfehler:

```
java.lang.IllegalStateException:
{ when=-6h11m56s20ms what=0 target=de.zimkand.primedetection.PrimeActivity$MyHandler }
This message is already in use.
```

```
message.setData(bundle);
```

die per **sendMessage()** - Aufruf an den vom Aktivitätsobjekt zur Verfügung gestellten **Handler** auf die Reise geht:

```
handler.sendMessage(message);
```

Die Klasse **PrimServiceBound** erlaubt es, das **Handler**-Objekt über ihre Methode **setHandler()** zu wechseln. So kann sich das nach einem Orientierungswechsel neu erstellte Aktivitätsobjekt als Empfänger für die Nachrichten des laufenden Dienstes registrieren lassen.

Die Aktivitätsklasse nutzt die **PrimeServiceBound**-Methode **evalCandidate()** in der Klick-Ereignisbehandlung zum Befehlsschalter:

```
@Override
public void onClick(View view) {
    String s = candidate.getText().toString();
    if (s.length() != 0 && bound) {
        result.setText("");
        button.setEnabled(false);
        progressBar.setProgress(0);
        progressBar.setVisibility(ProgressBar.VISIBLE);
        primeService.setHandler(handler);
        primeService.evalCandidate(s);
    }
}
```

Im Vergleich zu der in Abschnitt 11.2 vorgestellten Programmvariante mit der Primzahlendiagnose in einem gestarteten Dienst (aus der Klasse **IntentService**) lässt sich die im aktuellen Abschnitt Variante mit einem gebundenen Dienst deutlich mehr Zeit, was an der besseren Threading-Lösung durch die SDK-Klasse **IntentService** liegen mag.<sup>1</sup>

Weitere Details zur Verwendung von gebundenen Diensten finden sich z. B. bei Becker & Pant (2015, S. 175ff) sowie in Googles Online-Dokumentation für Android-Entwickler.<sup>2</sup>

Das komplette AS-Projekt mit der im aktuellen Abschnitt beschriebenen **Prime Detection** - Variante ist im folgenden Ordner zu finden:

...\\BspUeb\\Service\\LocalBoundService - Handler

## 11.4 Vordergrunddienste

Ein Dienst kann und sollte mit der **Context**-Methode **startForegroundService()** gestartet werden, wenn seine Tätigkeit vom Benutzer wahrnehmbar ist (z. B. bei einem Programm zur Musikwiedergabe). In diesem Fall muss ein permanenter Eintrag für die Benachrichtigungszeile erstellt werden, sodass der Benutzer Kontakt mit dem Dienst aufnehmen kann (meist durch Starten einer Activity). Dabei kommen die in Abschnitt 9.5 kurz erwähnten Intents in Wartestellung (engl.: *pending intents*) zum Einsatz. Es wird kaum passieren, dass ein Vordergrunddienst wegen Speicherangel terminiert wird. Weitere Informationen über Vordergrunddienste finden sich z. B. in der Online-Dokumentation zu Android.<sup>3</sup>

---

<sup>1</sup> Außerdem ist aufgefallen, dass es beim Start des Programms aus dem Android Studio 3.1 zu einem Stack Overflow - Fehler kommt, wenn in der Run-Konfiguration das Advanced Profiling eingeschaltet ist. Das Problem wird vermutlich bald durch ein Update der Entwicklungsumgebung behoben.

<sup>2</sup> <https://developer.android.com/guide/components/services.html>  
<https://developer.android.com/guide/components/bound-services.html>

<sup>3</sup> <https://developer.android.com/guide/components/services.html>

## 11.5 Übungsaufgaben zu Kapitel 11

1) Welche von den folgenden Aussagen sind richtig bzw. falsch?

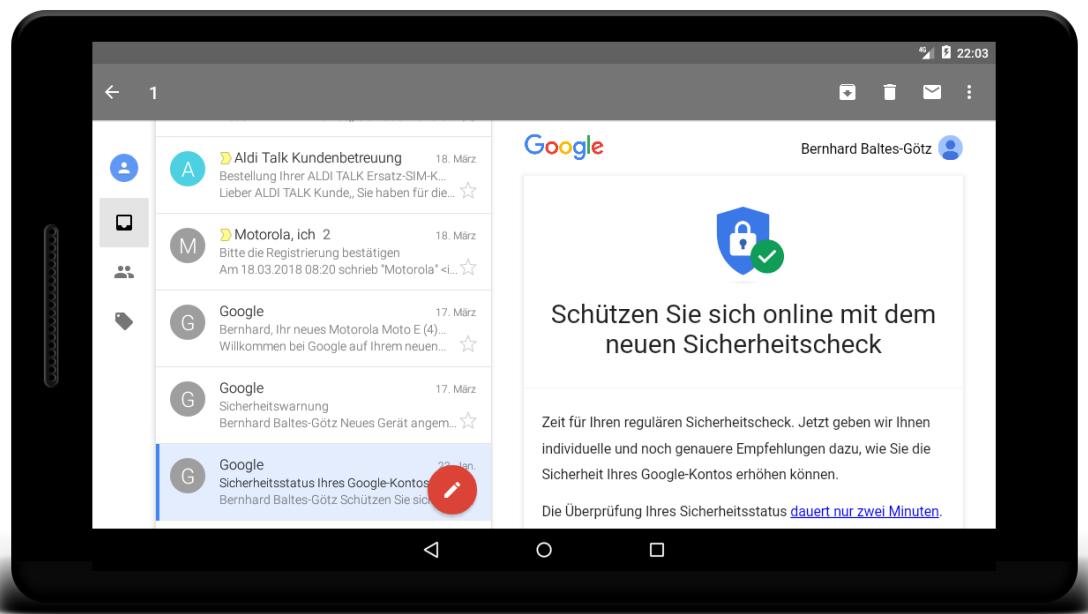
1. In der Manifestdatei sollten Intent-Filter zu einem Dienst definiert werden.
2. Seit der Android-Version 8 gelten Einschränkungen für gestartete Dienste.
3. Bei Verwendung der Service-Ableitung **IntentService** ist es explizit erwünscht, ausschließlich die in **IntentService** definierte Rückrufmethode **onHandleIntent()** zu überschreiben.
4. Ein gestarteter Dienst (ohne Bindungsoption) ist nicht in der Lage, mehrere Aufträge simultan auszuführen.
5. Ein Dienst in der gebundenen Betriebsart (kein **onStartCommand()** - Aufruf erfolgt) erhält einen **onDestroy()** - Aufruf, sobald alle Klienten ihre Bindung gelöst haben.

2) Ersetzen Sie in der Primzahlen-App aus Abschnitt 11.3 die Ergebniskommunikation per **Handler** durch die Abschnitt 11.2 verwendete Kommunikation per **LocalBroadcastManager**.

## 12 Fragmente

Das ursprüngliche UI-Design von Android orientiert sich an **Smartphones** und arbeitet wegen der begrenzten Bedienoberfläche nach dem **Stapelprinzip**. Eine App hält für jede Teilaufgabe eine Activity mit den benötigten Steuerelementen bereit, und diese läuft im Vollbildmodus. Es gibt z. B. oft eine Activity zur Präsentation einer Liste (von Kontakten, Medien oder sonstigen Elementen) sowie eine zweite Activity zur Anzeige bzw. Bearbeitung der Details zu einem gewählten Listen-element. Klickt der Benutzer in der Übersichtaktivität auf ein Listenelement, dann wird per Intent die Detailsaktivität gestartet. Für den Wechsel von der Details- zur Übersichtaktivität kann z. B. der Rückwärtsschalter verwendet werden.

Auf einem **Tablet** (mit einer Bildschirmdiagonale von ca. 7 bis 12 Zoll) ist aber genügend Platz vorhanden, um die Übersicht- *und* die Detailsanzeige *simultan* präsent zu halten und damit die Bedienbarkeit der Anwendung zu verbessern. Android hat mit der Version 3 (API-Level 11), die Tablet-Geräten vorbehalten war, mit dem **Fragment** eine Aktivitätskomponente eingeführt, die es erleichtert, Bedienelemente für verschiedene Teilaufgaben auf *einer* Bildschirmseite präsent zu halten, was hier mit der GMail-App vorgeführt wird:



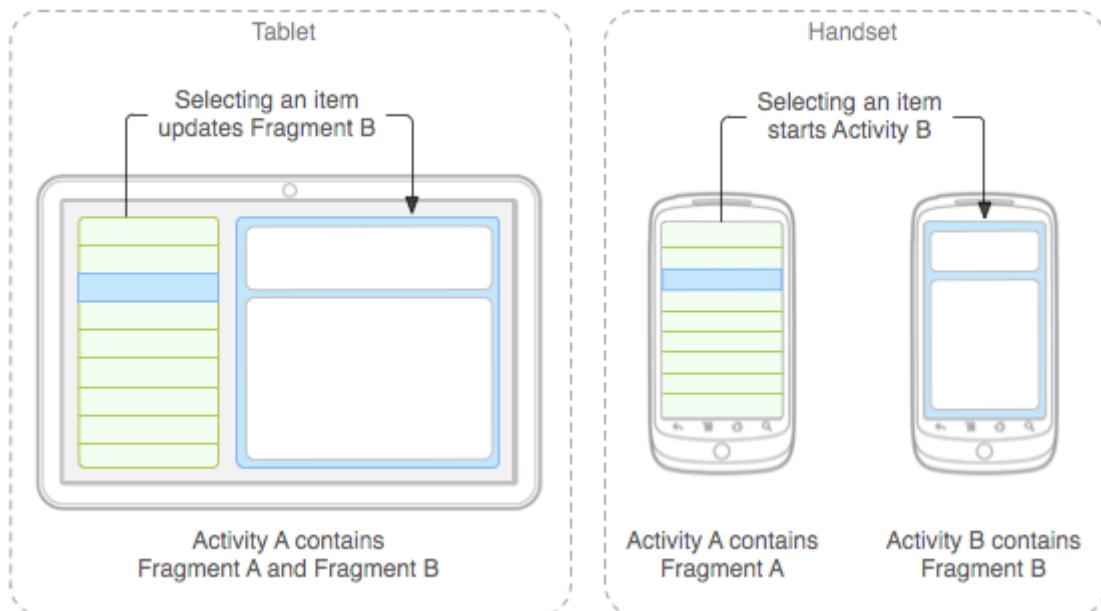
Über die Liste auf der linken Seite kann eine Mail gewählt werden, die auf der rechten Seite angezeigt werden soll. Seit Android 4 stehen Fragmente auch auf Smartphones zur Verfügung.

Sicher ist die ergonomische Unterstützung eines Tablet-Displays, das üblicherweise im Landschaftsformat betrieben wird, auch mit traditionellen Android-Mitteln möglich, z. B. mit einem horizontal orientierten **LinearLayout** - Container auf oberster Ebene und entsprechend positionierten Bedienelementen (z. B. ein **ListView**-Element auf der linken Seite und ein Container mit den Elementen für die Detailanzeige auf der rechten Seite). Allerdings ist in diesem Fall *eine* Activity für *alle* Bedienelemente zuständig, und diese Activity ist für den Einsatz auf einem Smartphone ziemlich ungeeignet, sodass ein App-Entwickler zur Unterstützung *beider* Gerätekategorien viel Parallelarbeit zu leisten hat.

Außer der rationalen Unterstützung verschiedener Displayformate bietet die Fragment-Technik auch die Option zur **dynamischen Anpassung der Bedienoberfläche** an Benutzerwünsche, indem Fragmente im laufenden Betrieb aufgenommen, entfernt oder ersetzt werden können. Zusätzlich kann eine Aktivität ein eigenes Back Stack - Segment verwalten und dort Fragment-Konfigurationen ablegen, zu denen der Benutzer später per Back-Taste zurückkehren kann.

## 12.1 Aufgabenverteilung zwischen Aktivitäten und Fragmenten

Mit einem Fragment ist es möglich, Teile der Bedienelemente *samt zugehöriger Anwendungslogik* in einer Klasse zusammenzufassen, die als Modul in verschiedene Aktivitäten eingebunden werden kann. In der folgenden Abbildung, die von Googles Webseite für Android-Entwickler stammt, ist eine App zu sehen, die für eine Listen- und eine Detailansicht jeweils ein Fragment verwendet. Auf einem Tablet kommt eine Aktivität A zum Einsatz, die *beide* Fragmente einbindet. Auf einem Smartphone beschränkt sich die Aktivität A auf das Listenfragment und startet nach Auswahl eines Listenelements die Aktivität B, die das Detailsfragment einbindet:<sup>1</sup>



Das wesentliche Motiv für die Erweiterung von Android um Fragmente bestand darin, unterschiedliche Display-Größen bei minimalem Zusatzaufwand unterstützen zu können.

Um für eine eigene App ein Fragment zu realisieren, leitet man eine neue Klasse aus der Basisklasse **Fragment** oder einer bereits bestehenden Unterklasse ab. Weil die Klasse **Fragment** im Paket **android.app** ab API-Level 28 als abgewertet (engl.: *deprecated*) gilt, sollte die gleichnamige Klasse aus dem Paket **android.support.v4.app** (also aus der Support Library) verwendet werden.

Das Android-SDK enthält einige **Fragment**-Ableitungen, die bei Standardaufgaben eine günstige Ausgangsbasis für eigene Klassen darstellen:

- **DialogFragment**  
Wird eine Dialogbox als Fragment realisiert statt über Methoden der Klasse **Activity**, kann sie auf dem Back Stack - Segment der Aktivität abgelegt und folglich vom Benutzer später wieder angesteuert werden (siehe Abschnitt 12.7.3).
- **ListFragment**  
Diese Klasse zeigt analog zur Klasse **ListActivity** eine Liste von Items an, die von einem Adapter (z. B. **ArrayAdapter**, **SimpleCursorAdapter**) verwaltet werden. Wenn bei einer **ListFragment**-Anwendung aufgrund des Listenumfangs die Performanz zum Problem wird, sollte nach einer Empfehlung von Google ein Fragment mit einem Objekt der Klasse **RecyclerView** im Layout verwendet werden. Wir werden gleich in einem Beispielprogramm die deutlich einfachere **ListFragment**-Lösung nutzen und dabei auf kein Performanz-Problem stoßen.

<sup>1</sup> <https://developer.android.com/guide/components/fragments>

- **PreferenceFragmentCompat**

Diese Klasse präsentiert eine Liste von **Preference**-Objekten (analog zur Klasse **PreferenceActivity**).

- **WebViewFragment**

Diese Klasse unterstützt die Anzeige von Webinhalten durch eine **WebView**-Komponente.

Zum Verhältnis bzw. zur Kooperation von Aktivitäten und Fragmenten ist zu sagen:

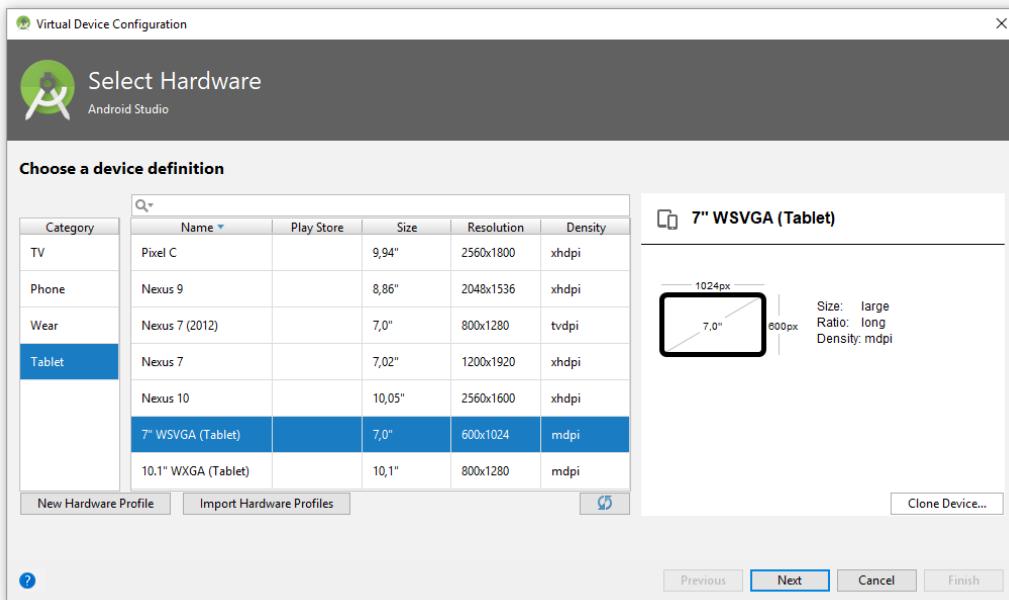
- Ein Fragment ist eine Aktivitätskomponente mit Anwendungslogik und eigenen Lebenszyklusmethoden, die in der Regel auch eine Bedienoberfläche präsentiert. Weil es *keine Anwendungskomponente* ist, muss ein Fragment *nicht* in die App-Manifestdatei eingetragen werden.
- Ein Fragment kann nur als Bestandteil einer Aktivität tätig werden. Google verwendet in seiner Entwicklerdokumentation die Bezeichnung *sub activity*. Eine Fragmentklasse kann in mehreren Aktivitätsklassen genutzt werden. Ein Fragmentobjekt ist jedoch fest mit dem umgebenden Aktivitätsobjekt verbunden.
- In einer Aktivität lassen sich auch *mehrere* Fragmente verwenden, um z. B. eine Mehrzonen-Bedienoberfläche (engl. *multi-pane UI*) zu realisieren. Selbstverständlich kann eine Aktivität neben Fragmenten weiterhin auch Steuerelemente einsetzen, um ihre Funktionalität zu realisieren.
- Der Lebenszyklus eines Fragments ist an den Lebenszyklus der umgebenden Aktivität gekoppelt. Muss z. B. die Aktivität pausieren, so gilt dies auch für ihre Fragmente. Wird eine Aktivität zerstört, so ereilt ihre Fragmente dasselbe Schicksal.
- Die Einbindung eines Fragments in eine Aktivität kann statisch über die Layoutdefinitionsdatei der Aktivität erfolgen oder dynamisch per Programm (ähnlich wie bei **View**-Objekten).
- Wenn eine Aktivität ihre Bedienoberfläche durch Fragment-Transaktionen modifiziert (Fragmente hinzufügt, entfernt oder ersetzt), kann sie die einzelnen Zustände in dem von ihr verwalteten Back Stack - Segment konservieren, sodass der Benutzer per Rückwärtsschalter dorthin zurückkehren kann.

## 12.2 Vorschlag für ein AVD-Tablet

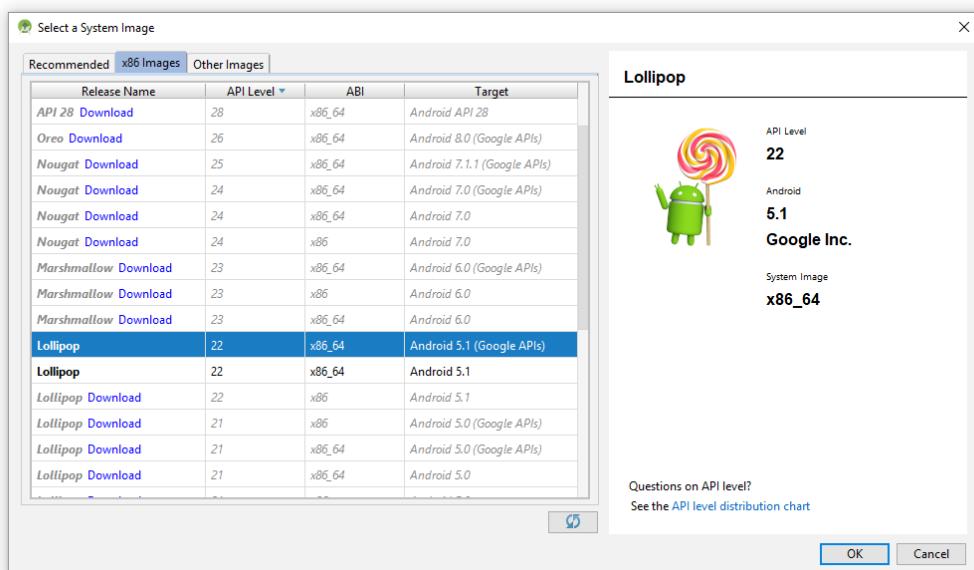
Der Start eines emulierten Tablets im Rahmen unserer Entwicklungsumgebung kann wegen der größeren Auflösung erhebliche Zeit in Anspruch nehmen. Mit dem folgenden Bauvorschlag sollte das nicht passieren:

- Öffnen Sie aus dem Android Studio den AVD-Manager, und beginnen Sie per Mausklick auf den Schalter **Create Virtual Device** die Definition eines virtuellen Tablets.

- Wählen Sie **Tablet** als **Category** und ein Gerät mit möglichst geringer Auflösung, z. B. **7" WSVGA (Tablet)**:

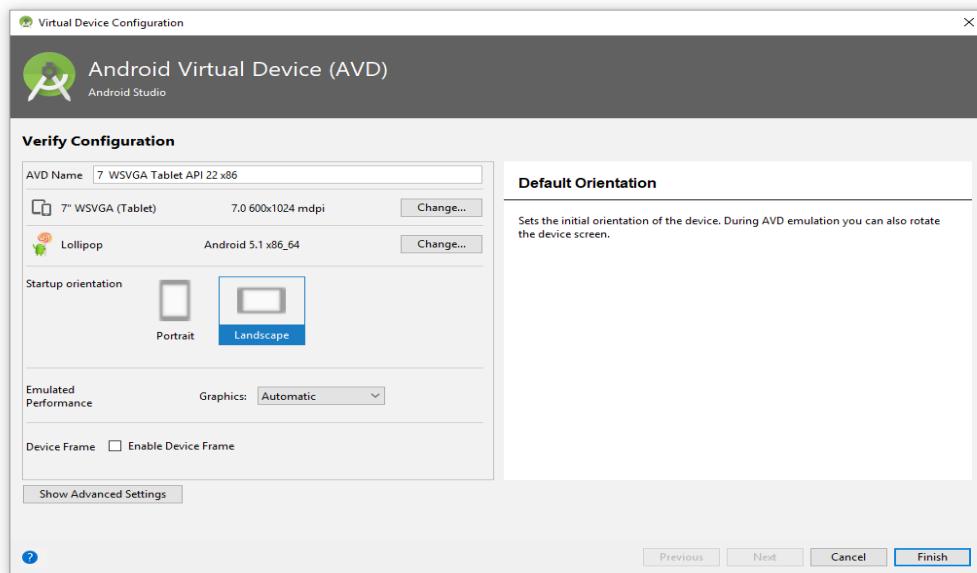


- Wählen Sie im nächsten Dialog ein Ressourcen schonendes **System Image** ab **API-Level 15**, z. B.:

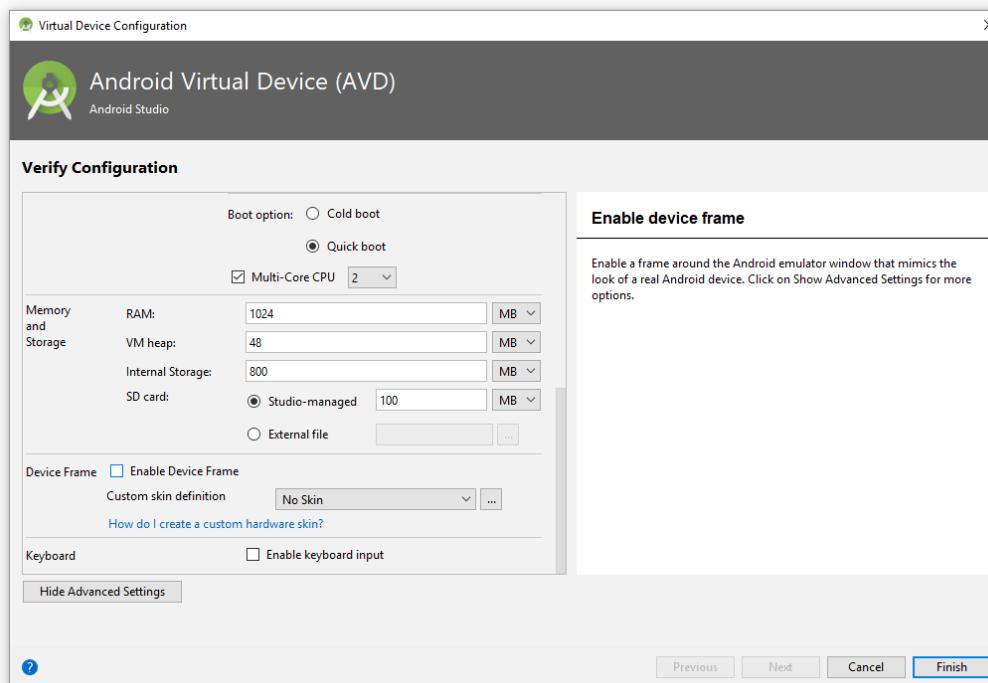


Falls der *Hardware Accelerated Execution Manager* von Intel installiert ist (vgl. Abschnitt 3.3), sollte ein System-Image mit x86 - Technik verwendet werden.

- Wählen Sie im nächsten Dialog einen **AVD-Namen** und **Landscape** als **Startup Orientation**:



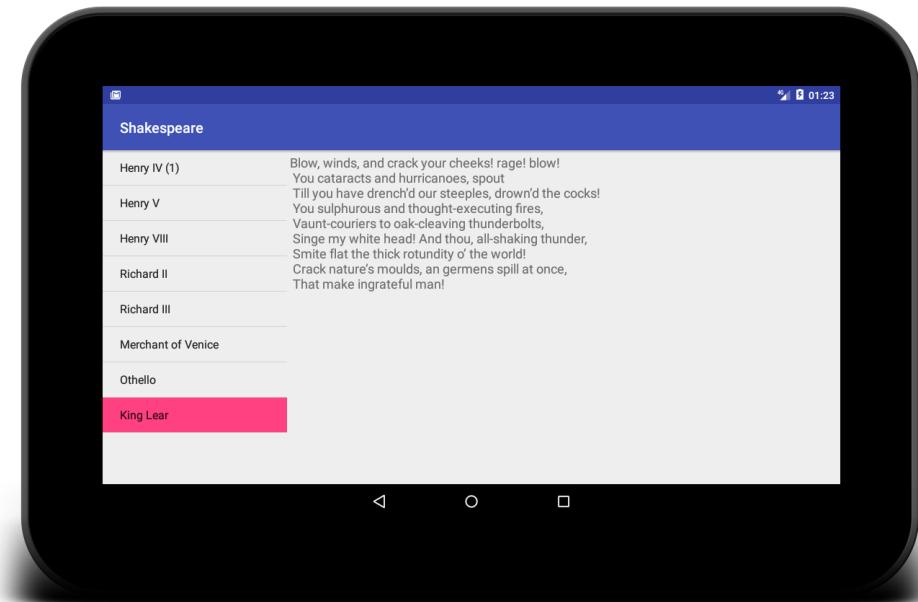
- Nach einem Klick auf den Schalter **Show Advanced Settings** kann man z. B. die RAM-Ausstattung beeinflussen und die Markierung beim Kontrollkästchen **Enable keyboard input** entfernen, damit im emulierten Gerät bei Verwendung eines Texteingabefelds (wie bei einem realen Android-Gerät) eine virtuelle Tastatur auf dem Bildschirm erscheint:



### 12.3 Beispiel für die Verwendung von Fragmenten

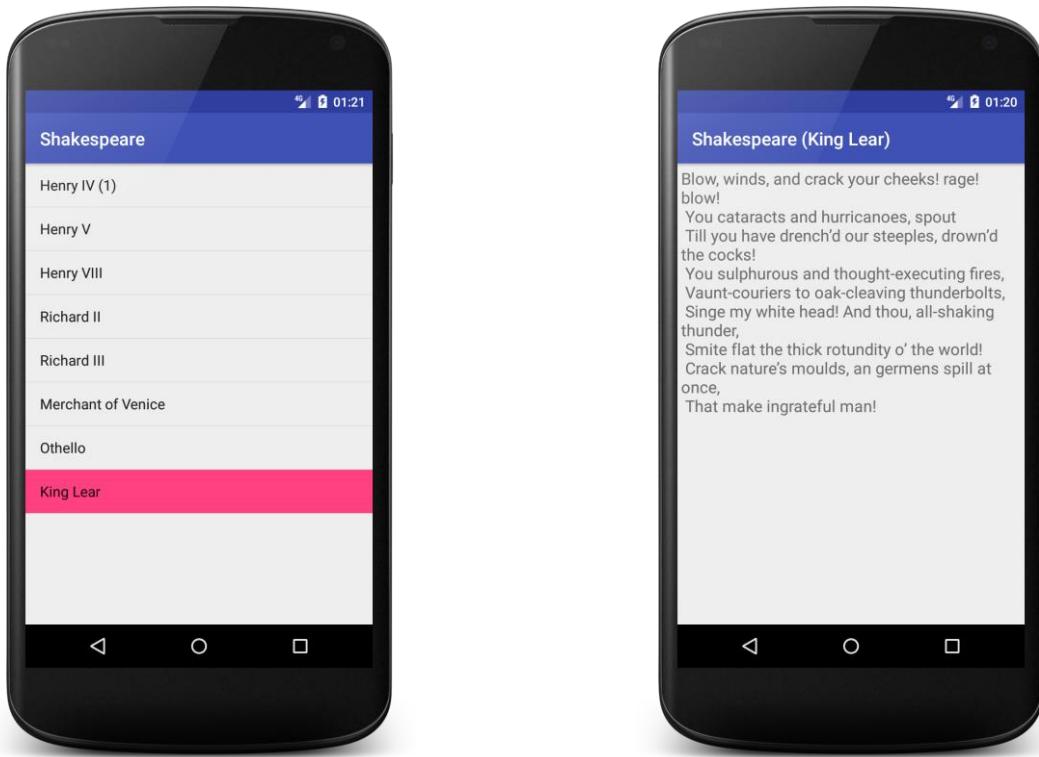
Im aktuellen Kapitel 12 verwenden wir zur Demonstration eine abgewandelte Version des von Google auf der Entwickler-Webseite zu Fragmenten beschriebenen Beispielprogramms, das Shakespeare-Text anzeigen kann.<sup>1</sup> In einer Liste werden die Titel von 8 Stücken angezeigt, und nach Wahl eines Titels erscheint ein Textauszug. Für die Auswahlliste und die Textanzeige ist jeweils ein Fragment zuständig. Auf beliebigen Android-Geräten werden im Querformat die Titelliste (links) und die Textanzeige (rechts) simultan durch *eine* Aktivität mit *zwei* Fragmenten angezeigt. Im Hochformat werden jedoch *zwei* Aktivitäten verwendet, die jeweils *ein* Fragment enthalten, das die Titelliste bzw. den Text zum gewählten Titel anzeigt.

So arbeitet die fertige App, deren Innenleben Sie im weiteren Verlauf von Kapitel 12 kennenlernen werden, auf einem Tablet im Landschaftsformat:



Im Portraitformat sind zwei verschiedene Aktivitäten im Einsatz, die jeweils nur *ein* Fragment beherbergen:

<sup>1</sup> <https://developer.android.com/guide/components/fragments.html>



Das komplette AS-Projekt mit dem Beispielprogramm ist im folgenden Ordner zu finden:

**...\\BspUeb\\Fragmente\\Shakespeare**

#### 12.4 Fragmente in eine Aktivität integrieren

Im Beispielprogramm wird die Klasse **StartActivity** aus der vom Android Studio bei gewünschter (und voreingestellter) Kompatibilität inkl. API-Level 15 gewählten Klasse **AppCompatActivity** abgeleitet. Der Quellcode beschränkt sich auf die Lebenszyklusmethode **onCreate()**:

```
public class StartActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.fragment_layout);
    }
}
```

Hier wird das Layout aus der Definitionsdatei **fragment\_layout.xml** expandiert, die in zwei konfigurationsspezifischen Varianten vorliegt (siehe Abschnitt 7.2 zu konfigurationsabhängigen Resourcen). In der Layoutdefinition für das **Landschaftsformat** (abzulegen im Projektordner **...\\app\\src\\main\\res\\layout-land**, vgl. Abschnitt 8.3.1) werden die beiden in Abschnitt 12.3 beschriebenen Fragmente in einem horizontal orientierten **LinearLayout** - Container (siehe Abschnitt 8.3.1) untergebracht:

```

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <fragment class="de.zimkand.fragmentdemo.TitlesFragment"
        android:id="@+id/titles"
        android:layout_height="match_parent"
        android:layout_width="0dp"
        android:layout_weight="1" />

    <FrameLayout android:id="@+id/details"
        android:layout_height="match_parent"
        android:layout_width="0dp"
        android:layout_weight="3"
        android:background="?android:attr/detailsElementBackground" />
</LinearLayout>

```

Dabei kommen die beiden alternativ verwendbaren Techniken für die Fragmentintegration zum Einsatz:

- Das Fragment mit einer Liste von Shakespeare-Stücken wird analog zu einem **View**-Objekt durch ein **fragment** - Element repräsentiert (mit *klein* geschriebenen Namensanfang!). Desse **ns class**-Attribut benennt die Klasse, welche das Fragment realisiert. Im Beispiel kommt die Klasse **TitlesFragment** zum Einsatz. Über seine Element-ID kann ein Fragment zur Laufzeit identifiziert werden.
- Das Fragment zur Anzeige einer Textpassage aus dem gewählten Shakespeare-Stück wird zur Laufzeit per Programm in den **FrameLayout** - Container (vgl. Abschnitt 8.3.4) mit der Element-ID **details** eingefügt, wenn die App im Landschaftsmodus läuft. Wie ein Fragment per Programm explizit erzeugt und per Fragment-Transaktion an einer vorbereiteten Layout-Stelle eingefügt wird, ist in Abschnitt 12.7 zu erfahren.

Durch die Ausdehnungsattribute **layout\_weight** und **layout\_width** wird dafür gesorgt, dass die verfügbare horizontale Ausdehnung im Verhältnis 1:3 auf die beiden Elemente (bzw. Fragmente) aufgeteilt wird (vgl. Abschnitt 8.3.1).

In der Layoutdefinition für das **Portraitformat** erscheint ausschließlich das Listenfragment:

```

<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <fragment class="de.zimkand.fragmentdemo.TitlesFragment"
        android:id="@+id/titles"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
</FrameLayout>

```

Bei der Aktivität zur Detailsanzeige im Portraitmodus werden wir ohne Layout-Definitionsdatei auskommen und die **View**-Hierarchie über Java-Anweisungen erstellen.

Unsere App unterstützt die beiden Orientierungen (Portrait vs. Landschaft) jeweils durch ein spezielles layout, kümmert sich aber z.B. nicht über die Displaygröße. Daher verhält sich die App auf Smartphones und Tablets identisch:

- Im Portraitformat ist entweder die Liste der Shakespeare-Stücke *oder* eine Textpassage aus einem Stück zu sehen.
- Im Landschaftsformat sind die Liste *und* eine Textpassage zu sehen.

Trifft Android beim Laden der Layoutdefinition zu einer Aktivität auf ein **fragment**-Element, wird ein Objekt der angegebenen Klasse erzeugt. Damit dies gelingen kann, muss die Fragmentklasse

einen **parameterfreien** Konstruktor mit Zugriffsmodifikator **public** besitzen. Dies ist z. B. der Fall, wenn in der Klasse *kein* expliziter Konstruktor definiert und somit der Standardkonstruktor verfügbar ist. Weil der Standardkonstruktor die Schutzstufe der Klasse übernimmt, muss diese als **public** definiert sein.

Im weiteren Verlauf ruft Android die in Abschnitt 12.5 beschriebenen Lebenszyklusmethoden des neuen **Fragment**-Objekts auf.

Eine Aktivität muss sich nicht (wie die **StartActivity** im Beispiel) auf die Rolle eines passiven Containers für aktive Fragmente beschränken, sondern kann z. B. ...

- in Fragmenten aufgetretene Ereignisse verarbeiten und an andere Fragmente weiterleiten
- Fragment-Transaktionen ausführen (siehe Abschnitt 12.9)

## 12.5 Lebenszyklusmethoden bei Fragmenten

Ein Fragment besitzt eigene Lebenszyklusmethoden, wobei nur wenige Unterschiede im Vergleich zu einer Aktivität bestehen (vgl. Abschnitt 6). Es folgt eine Liste der wichtigsten Lebenszyklusmethoden in der Reihenfolge, in der sie von Android für eine neue **Fragment**-Instanz bis zur Interaktionsfähigkeit mit dem Benutzer aufgerufen werden:<sup>1</sup>

- **public void onAttach(Context context)**  
Diese Methode wird aufgerufen, wenn eine Fragmentinstanz erstmals mit ihrer Aktivität assoziiert wird.
- **public void onCreate(Bundle savedInstanceState)**  
Diese Methode eignet sich zum Initialisieren von Instanzvariablen, die den Zustand des Fragments definieren.
- **public View onCreateView(LayoutInflater inflater,  
                        ViewGroup container, Bundle savedInstanceState)**  
Bei einem Fragment *mit* Bedienoberfläche wird diese Methode vor der ersten Anzeige aufgerufen, wenn die Objekte der Steuerelementhierarchie zu instanzieren sind. Sie muss als Rückgabe ein **View**-Objekt liefern, das als Wurzelement für die Steuerelemente des Fragments dient. Wird die Klasse **ListFragment** als Basisklasse für ein Fragment verwendet, kann man auf das Implementieren der Methode **onCreateView()** verzichten und das Erbstück aus der Basisklasse **ListFragment** verwenden, das ein **ListView**-Objekt abliefer (siehe Beispiel in Abschnitt 12.6). Bei einem Fragment ohne Bedienoberfläche, liefert man als **onCreateView()** - Rückgabe den Wert **null**.
- **public void onActivityCreated(Bundle savedInstanceState)**  
Für die **Fragment**-Methode **onCreate()** ist nicht garantiert, dass zum Zeitpunkt ihres Aufrufs die umgebende Aktivität bereits vollständig initialisiert ist (Künneth 2012, S. 114). Wenn bei der **Fragment**-Initialisierung auf die umgebende Aktivität zugegriffen werden muss, ist die Methode **onActivityCreated()** zu bevorzugen, die garantiert erst nach Beendigung der **Activity**-Methode **onCreate()** aufgerufen wird.
- **public void onStart()**  
Diese Methode wird aufgerufen, wenn das Fragment sichtbar wird.
- **public void onResume()**  
Diese Methode wird aufgerufen, wenn das Fragment mit dem Benutzer interagieren kann.

Wird eine von diesen Methoden überschrieben, muss die Basisklassenvariante aufgerufen werden, was meist in der ersten Anweisung geschieht.

---

<sup>1</sup> <https://developer.android.com/reference/android/support/v4/app/Fragment>

Die Pflicht, eine überschriebene Basisklassenvariante aufzurufen, gilt (mit Ausnahme von `onSaveInstanceState()`) auch für die bei einer (temporären) Deaktivierung beteiligten **Fragment**-Lebenszyklusmethoden, die anschließend in der Reihenfolge aufgelistet sind, in der sie von Android aufgerufen werden:

- **public void onPause()**

Diese Methode wird aufgerufen, wenn das Fragment (teilweise) überlagert wird, und keine weitere Interaktion mit dem Benutzer möglich ist. Man muss damit rechnen, dass der Benutzer nicht mehr zurückkehrt und alle Persistenzdaten in `onPause()` sichern. Seit Android 3.0 ist garantiert, dass nach `onPause()` auch noch `onStop()` aufgerufen wird (vgl. Abschnitt 6.2.6).

- **public void onSaveInstanceState(Bundle state)**

Diese Methode wird irgendwann vor `onDestroy()` aufgerufen und dient zum Sichern von Zustandsinformationen. Das beim Sichern entstehende **Bundle**-Objekt wird den Methoden `onCreate()`, `onActivityCreated()` und `onCreateView()` als Parameter übergeben, wenn das Fragment später wiederhergestellt wird. Die Zustände der Steuerelemente eines Fragments (z. B. Inhalt eines Texteingabefelds) werden (wie bei Aktivitäten) automatisch gesichert und restauriert, sofern sie eine Kennung besitzen (vgl. Abschnitt 6.1).<sup>1</sup> Um andere Zustandsinformationen (z. B. in Instanzvariablen) muss sich der Programmierer kümmern. Die Methode `onSaveInstanceState()` wird *nicht* aufgerufen, wenn der Benutzer das Fragment oder die komplette Aktivität per Rückwärtstaste verlässt.

- **public void onStop()**

Diese Methode wird aufgerufen, wenn das Fragment nicht mehr sichtbar ist. Hier beendet man Abläufe, die in `onStart()` gestartet worden sind. Nach Ausführung der Methode `onStop()` ist ein Fragment davon bedroht, bei Speicherzugang zusammen mit der kompletten App eliminiert zu werden. Daher ist es oft erforderlich, in `onStop()` Persistenzdaten zu sichern.

- **public void onDestroy()**

In dieser Methode sollten aufwändige Ressourcen freigegeben werden. Google nennt als typische Aufgabe für diese Methode das Stoppen von Threads (siehe Kapitel 10), die mit dem Fragment assoziiert sind.

- **public void onDetach()**

Diese Methode wird aufgerufen, wenn die Assoziation einer Fragmentinstanz mit ihrer Aktivität aufgehoben wird.

Ein Fragment, das als Komponente in eine Aktivität aufgenommen wurde, kann sich wie sein Gastgeber abgesehen von kurzen Übergangsphasen in folgenden Zuständen befinden:

- **Running (alias: Resumed)**

Die umgebende Aktivität ist im Zustand **Running**, und das Fragment ist sichtbar.

- **Paused**

Die umgebende Aktivität ist teilweise durch eine andere Aktivität überlagert und kann keine Benutzereingaben mehr empfangen. Die neue Vordergrundaktivität ist entweder teilweise transparent oder belegt nicht den gesamten Bildschirm.

---

<sup>1</sup> Bei einem **TextView**-Steuerelement klappt die automatische Sicherung und Wiederherstellung nur dann, wenn in der Layoutdefinition das Attribut `android:freezesText` den Wert `true` erhält.

- **Stopped**

Das Fragment ist nicht mehr sichtbar, wobei zwei Gründe möglich sind:

- Die umgebende Aktivität ist komplett durch eine andere Aktivität überlagert worden (befindet sich im Zustand **Stopped**).
- Das Fragment wurde aus der (sichtbaren) umgebenden Aktivität entfernt *und* dabei auf dem Fragment-Stack der Aktivität abgelegt (siehe Abschnitt 12.7.3).

- **Destroyed**

Für die Zerstörung eines Fragments kommen zwei Gründe in Frage:

- Die umgebende Aktivität wird zerstört.
- Das Fragment wurde aus der (sichtbaren) umgebenden Aktivität entfernt und dabei *nicht* auf dem Fragment-Stack der Aktivität abgelegt (siehe Abschnitt 12.7.3).

## 12.6 ListFragment-Initialisierung

Im Shakespeare-Beispielprogramm wird die Fragment-Klasse zur Präsentation der Titelliste aus der SDK-Klasse **ListFragment** im Paket **android.support.v4.app** (also in der Support Library) abgeleitet:<sup>1</sup>

```
import android.support.v4.app.ListFragment;
import android.support.v4.app.FragmentManager;
import android.support.v4.app.FragmentTransaction;
.

public class TitlesFragment extends ListFragment
    implements FragmentManager.OnBackStackChangedListener {
    private boolean mDualPane;
    private int mCurCheckPosition;
    private FragmentManager fm;
    private boolean uselessStackState = true;

    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);

        fm = getFragmentManager();
        fm.addOnBackStackChangedListener(this);

        setListAdapter(new ArrayAdapter<String>(getActivity(),
            android.R.layout.simple_list_item_activated_1,
            getResources().getStringArray(R.array.titles)));

        mDualPane = getActivity().findViewById(R.id.details) != null;
        getListView().setChoiceMode(ListView.CHOICE_MODE_SINGLE);

        if (savedInstanceState != null)
            mCurCheckPosition = savedInstanceState.getInt("curChoice", 0);

        if (mDualPane) {
            if (savedInstanceState == null)
                getListView().setItemChecked(mCurCheckPosition, true);
            showDetails(mCurCheckPosition);
        }
    }
}
```

---

<sup>1</sup> Die Klasse **ListFragment** im Paket **android.app** ist ab API-Level 8 herabgestuft (engl.: *deprecated*).

```

@Override
public void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    outState.putInt("curChoice", mCurCheckPosition);
}

@Override
public void onListItemClick(ListView l, View v, int position, long id) {
    mCurCheckPosition = position;
    showDetails(position);
}

@Override
public void onBackStackChanged() {
    DetailsFragment details =
        (DetailsFragment) fm.findFragmentById(R.id.details);
    getListView().setItemChecked(details.getShownIndex(), true);
}

private void showDetails(int index) {
    . . .
}

```

Ein Objekt der Klasse **ListFragment** enthält ein Steuerelement der Klasse **ListView**, das als vertikal rollbarer Container für **TextView**-Elemente dient. In den **TextView**-Elementen werden Zeichenfolgen dargestellt, die ein **Adapter**-Objekt zuliefert.

Im Beispielprogramm erfolgt die Initialisierung des **ListFragment**-Objekts in der Methode **onActivityCreated()**, für die garantiert ist, dass zum Zeitpunkt ihres Aufrufs die **onCreate()** - Methode der umgebenden Aktivität bereits vollständig ausgeführt ist. Somit ist es z. B. möglich, über die **Fragment**-Methode **getActivity()** eine Referenz zum Aktivitätsobjekt zu erhalten.

Als Adapter (Lieferant für die **ListView**-Elemente) kommt im Beispiel ein Objekt der konkretisierten generischen Klasse **ArrayAdapter<String>** zum Einsatz:

```

setListAdapter(new ArrayAdapter<String>(getActivity(),
    android.R.layout.simple_list_item_ACTIVATED_1,
    getResources().getStringArray(R.array.titles)));

```

Es wird eine Konstruktorüberladung

**ArrayAdapter(Context context, int resource, String[] objects)**

mit drei Parametern verwendet:

- *context*  
Den Kontext liefert im Beispiel die umgebende Aktivität, welche das Fragment mit der Methode **getActivity()** ermittelt.
- *resource*  
Hier ist die Ressourcen-ID zu einer Layoutdefinitionsdatei mit einem **TextView**-Element anzugeben. Beim Befüllen der Liste werden die **TextView**-Objekte mit Attributen aus diesem Layout erzeugt. Im Beispiel stammt die Ressourcen-ID bzw. Layoutdefinitionsdatei aus dem Arsenal des Betriebssystems.
- *objects*  
Im Beispiel werden die Listenelemente (Titel von Shakespeare-Stücken) aus einer Resource vom Typ **string-array** bezogen:

```

<resources>
    <string name="app_name">Shakespeare</string>

    <string-array name="titles">
        <item>Henry IV (1)</item>
        <item>Henry V</item>
        <item>Henry VIII</item>
        <item>Richard II</item>
        <item>Richard III</item>
        <item>Merchant of Venice</item>
        <item>Othello</item>
        <item>King Lear</item>
    </string-array>
    .
    .
</resources>

```

Für seine weitere Tätigkeit (speziell für seine Reaktion auf eine Elementauswahl) muss das Listenfragment wissen, ob das Detailsfragment in *derselben* Aktivität erscheinen soll („Dual-Pane - Modus“) oder in einer *separaten* Aktivität („Single-Pane - Modus“). Dazu wird überprüft, ob ein **View**-Objekt mit der Element-ID **details** existiert. So wird im Landscape-Layout der **FrameLayout**-Container bezeichnet, der das Details-Fragment aufnehmen soll (siehe Abschnitt 12.4).

```
mDualPane = getActivity().findViewById(R.id.details) != null;
```

Das aktuell gewählte Listenelement soll hervorgehoben dargestellt werden. Dazu muss mit der **ListView**-Methode **setChoiceMode()** der passende Modus **ListView.CHOICE\_MODE\_SINGLE** gesetzt werden:

```
getListView().setChoiceMode(ListView.CHOICE_MODE_SINGLE);
```

Außerdem sorgt diese Einstellung dafür, dass sich ein zerstörtes und wiederbelebtes **ListView**-Objekt dank der fürsorglichen Behandlung durch das Betriebssystem daran erinnern kann, welches Element in der Vorgängerinstanz gewählt war.

Das **ListView**-Objekt behält zwar über einen Aktivitätsneustart hinweg seine Markierung bei, liefert aber mit der Methode **getSelectedItemPosition()** nach der aktuellen Wahl befragt stets den unbrauchbaren Wert -1. Daher speichert das Listen-Fragment die aktuelle Wahl in der Instanzvariablen **mCurCheckPosition**, die in der Lebenszyklusmethode **onSaveInstanceState()** in ein **Bundle**-Objekt gesichert wird, das Android ggf. einer Nachfolgerinstanz übergibt:

```

@Override
public void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    outState.putInt("curChoice", mCurCheckPosition);
}

```

In **onActivityCreated()** wird ggf. aus dem übergebenen **Bundle**-Objekt die aktuelle Wahl ermittelt:

```

if (savedInstanceState != null)
    mCurCheckPosition = savedInstanceState.getInt("curChoice", 0);

```

Im Landschaftsmodus wird in **onActivityCreated()** die private **TitlesFragment**-Methode **showDetails()** aufgerufen, damit im Details-Fragment der aktuell gewählte Text erscheint (zur Methode **showDetails()** siehe Abschnitt 12.7). Bei einer neu gestarteten App (**Bundle**-Parameter von **onActivityCreated()** ist **null**) wird im Details-Fragment der erste Text angezeigt (**mCurCheckPosition** hat den Initialisierungswert 0), und das erste Listenelement wird hervorgehoben:

```

if (mDualPane) {
    if (savedInstanceState == null)
        getListView().setItemChecked(mCurCheckPosition, true);
    showDetails(mCurCheckPosition);
}

```

Weil ein Listenfragment eine Bedienoberfläche anbietet, muss seine Klasse die Methode **onCreateView()** beherrschen, die zum Instanzieren der Steuerelementhierarchie dient (vgl. Abschnitt 12.5). Sie wird vor der ersten Anzeige des Fragments aufgerufen und muss ein **View**-Objekt abliefern, das als Wurzelement für die Steuerelemente des Fragments dient. Im konkreten Fall kann die Klasse **TitlesFragment** auf das Implementieren der Methode **onCreateView()** verzichten und das Erbstück aus ihrer Basisklasse **ListFragment** verwenden, das ein **ListView**-Objekt liefert.

Android liefert an die Methode **onCreateView()**

```
public View onCreateView(LayoutInflater inflater, ViewGroup cont, Bundle savedInstanceState)
```

per Parameter:

- einen **LayoutInflater** (vgl. Abschnitt 8.1)
- den **ViewGroup**-Container in der Aktivität, der das Fragment aufnehmen wird,
- ein **Bundle**-Objekt, das ggf. Informationen über die Vorgängerinstanz des Fragment-Objekts enthält.

Wird die **LayoutInflater**-Methode **inflate()**

```
public View inflate(int resource, ViewGroup root, boolean attachToRoot)
```

zur Kreation der **onCreateView()** - Rückgabe verwendet, ist als erster Aktualparameter eine Layout-Ressource zu übergeben. Zum Erstellen einer eigenen Layout-Ressource geht man genauso vor wie bei einer Aktivität (vgl. Abschnitt 8.1). Beim **boolean**-Parameter in Position 3 muss unbedingt der Wert **false** übergeben werden, womit die Verwaltung des abgelieferten **View**-Wurzelements dem Fragment-Framework überlassen wird, statt es fest an den übergeordneten Container zu binden (siehe Mednieks et al. 2013, S. 234).

Statt eine Layout-Ressource und die Methode **inflate()** zu verwenden, kann man die Bedienoberfläche des Fragments auch per Programm definieren. So werden wir beim Fragment für die Textanzeige im Shakespeare-Beispiel vorgehen (siehe Abschnitt 12.8).

## 12.7 Fragment-Transaktionen

Im Shakespeare-Beispielprogramm ist das Listenfragment auch dafür verantwortlich, bei Bedarf das Detailsfragment ins Spiel zu bringen bzw. zu aktualisieren. Das geschieht ...

- im Dual-Pane - Modus bei der Initialisierung des Listenfragments durch die **TitlesFragment**-Methode **onActivityCreated()** (siehe Abschnitt 12.6) und nach der Wahl eines Listenelements,
- im Single-Pane - Modus nach der Wahl eines Listenelements.

Bei Wahl eines Listenelements wird die **ListFragment**-Methode **onListItemClick()** ausgeführt, wobei wir uns aufgrund der **ListFragment**-Klassendefinition nicht selbst um die Registrierung des Ereignisempfängers kümmern müssen. Wir überschreiben die Methode **onListItemClick()** in unserer Klasse **TitlesFragment** folgendermaßen:

```
@Override
public void onListItemClick(ListView list, View v, int position, long id) {
    mCurCheckPosition = position;
    showDetails(position);
}
```

Die hier sowie in **onActivityCreated()** aufgerufene private **TitlesFragment**-Methode **showDetails()** sorgt für einen umgebungsgerechten Auftritt des Detailsfragments:

```
private void showDetails(int index) {
    if (mDualPane) {
        DetailsFragment details = (DetailsFragment) fm.findFragmentById(R.id.details);
        if (details == null || details.getShownIndex() != index) {
            details = DetailsFragment.newInstance(index);
            FragmentTransaction ft = fm.beginTransaction();
            ft.setCustomAnimations(android.R.animator.fade_in, android.R.animator.fade_out);
            ft.replace(R.id.details, details);
            if (!uselessStackState)
                ft.addToBackStack(null);
            ft.commit();
        }
    } else {
        Intent intent = new Intent();
        intent.setClass(getActivity(), DetailsActivity.class);
        intent.putExtra("index", index);
        startActivity(intent);
    }
    uselessStackState = false;
}
```

Ist der Dual-Pane - Modus aktiv, wird in folgenden Fällen dafür gesorgt, dass ein neues Detailsfragment erstellt und im **FrameLayout** auf der rechten Display-Seite angezeigt wird:

- Es existiert kein Detailsfragment, was bei einer neu gestarteten App der Fall ist. In diesem Fall wird **showDetails()** in **onActivityCreated()** mit dem Parameterwert 0 aufgerufen, so dass der erste Text erscheint.
- Beim Aufruf von **showDetails()** aus **onListItemClick()** zeigt das vorhandene Detailsfragment nicht den aktuell gewählten Shakespeare-Text an.

Dabei spielt der nun zu behandelnde Fragment-Manager eine zentrale Rolle.

### 12.7.1 Der Fragment-Manager

Um die Bedingungen zu überprüfen und ggf. die erforderlichen Fragment-Transaktionen durchführen zu können, wird der zuständige **FragmentManager** ermittelt, wozu in den Klassen **Activity** und **Fragment** die Methode **getFragmentManager()** bereitsteht:

```
private FragmentManager fm;
. . .
fm = getFragmentManager();
```

Mit der **FragmentManager**-Methode **findFragmentById()** wird versucht, eine Referenz zum bereits vorhandenen, bei einem früheren **showDetails()** - Aufruf erstellten, Detailsfragment zu erhalten:

```
DetailsFragment details = (DetailsFragment) fm.findFragmentById(R.id.details);
```

Die zum Projekt gehörige Klasse **DetailsFragment** wird gleich vorgestellt. Als Argument für die Methode **findFragmentById()** ist eine Element-ID anzugeben:

- Bei einer Fragment-Integration über das **fragment**-Element einer Layout-Definition ist dessen ID anzugeben (vgl. Abschnitt 12.4).
- Wurde ein Fragment per Programm in die Aktivität aufgenommen, ist die ID des Containers anzugeben. In unserem Fall ist diese Variante zu verwenden. Die benötigte Element-ID gehört zum **FrameLayout**-Container, den wir in die Layout-Definition für den Landschaftsmodus zur Aufnahme des Details-Fragments eingebaut haben.<sup>1</sup>

### 12.7.2 Die Klasse FragmentTransaction

Wenn von **findFragmentById()** eine **null**-Referenz zurückgeliefert wird, oder das vorhandene Fragment auf Befragen mit **getShownIndex()** mitteilt, dass es *nicht* den korrekten Shakespeare-Text anzeigt, ist eine Fragment-Transaktion erforderlich:<sup>2</sup>

```
if (details == null || details.getShownIndex() != index) {  
    . . .  
}
```

Zunächst wird ein neues Objekt der Klasse **DetailsFragment** erstellt, das den Text zum aktuell gewählten Listenelement anzeigen soll. Für solche Aufgaben bieten viele Klassen einen parametrisierten Konstruktor an, was Fragment-Klassen in der Regel aber *nicht* tun. Stattdessen ist oft eine statische Fabrikmethode mit äquivalenter Funktionalität vorhanden. Die Klasse **DetailsFragment** besitzt eine solche Fabrikmethode namens **newInstance()**, die per Parameter die Nummer des gewünschten Textes entgegennimmt und als Rückgabe eine Referenz auf ein neues **DetailsFragment** liefert (zur Definition der Methode **newInstance()** siehe Abschnitt 12.8):

```
details = DetailsFragment.newInstance(index);
```

Bei einer Fragment-Transaktion kann man eine (oder auch mehrere) von den folgenden Maßnahmen ausführen:

- ein Fragment in eine Aktivität aufnehmen
- ein Fragment entfernen
- ein Fragment durch ein anderes ersetzen

Um eine Fragment-Transaktion einzuleiten, ruft man die Methode **beginTransaction()** der Klasse **FragmentManager** auf und erhält als Rückgabe ein Objekt der Klasse **FragmentTransaction**, das für die weiteren Schritte zuständig ist. Im Beispiel ersetzen wir mit der **FragmentTransaction** - Methode **replace()**

```
public FragmentTransaction replace (int containerViewId, Fragment fragment)
```

das Fragment im Container, den die Element-ID im ersten Parameter anspricht, durch die vom zweiten Parameter referenzierte Neukreation:

```
FragmentTransaction ft = fm.beginTransaction();  
ft.replace(R.id.details, details);  
ft.commit();
```

Ggf. werden durch einen **replace()** - Aufruf auch *mehrere* aktuell im Container vorhandene Fragmente ersetzt. Wie das Verhalten des Beispielprogramms beim Start im Querformat zeigt, klappt die Methode **replace()** auch dann, wenn das zu ersetzende Fragment noch *nicht* existiert.

---

<sup>1</sup> Wie Sie aus Abschnitt 8.3.4 wissen, eignet sich der **FrameLayout**-Container zur Aufnahme von genau einem Steuerelement.

<sup>2</sup> Im Dual-Pane - Modus könnte man den Wechsel zu einem anderen Text auch *ohne* Fragment-Transaktion realisieren. Allerdings wäre das performantere Programm etwas komplizierter, und die bequeme Rückwärtsnavigation über den Aktivitäts-lokalen Back Stack würde entfallen.

Um alle vorbereiteten Änderungen in die Tat umzusetzen, ruft man die **FragmentTransaction**-Methode **commit()** auf.

Für die beiden anderen Fragment-Transaktionen aus der obigen Liste sind die folgenden **Fragment-Transaction**-Methoden zu verwenden:

- **public FragmentTransaction add(int containerViewId, Fragment fragment)**  
Zu **add()** sind etliche Überladungen vorhanden.
- **public FragmentTransaction remove(Fragment fragment)**

### 12.7.3 Back Stack für Fragment-Transaktionen

Soll dem Anwender die Möglichkeit geboten werden, eine Fragment-Transaktion per Rückwärts-Schalter zu widerrufen, befördert man sie mit der **FragmentTransaction**-Methode **addToBackStack()** auf den lokalen Back Stack der Aktivität, z. B.:

```
if (!uselessStackState)
    ft.addToBackStack(null);
```

Man kann die Transaktion per **String**-Parameter benennen, oder mit dem Wert **null** auf diese Option verzichten. Die Anweisung ist vor dem **commit()** - Aufruf einzufügen. Befinden sich auf dem lokalen Back Stack der Aktivität mehrere „Scheiben“, bewirkt die Rückwärts-Taste zunächst die sukzessive Rückabwicklung der Fragment-Transaktionen, bevor schließlich die Aktivität verlassen wird.

Entfernt man bei einer Transaktion ein Fragment, ohne **addToBackStack()** aufzurufen, dann wird das Fragment unwiederbringlich zerstört. Mit einem konservierenden **addToBackStack()** - Aufruf wird das Fragment lediglich gestoppt, und der Benutzer kann per Rückwärtsnavigation die Transaktion widerrufen.

Im Beispiel verhindert eine Bedingung, dass der beim Anwendungsstart durchlaufene Zustand vor der ersten Detailsanzeige auf den Back Stack gerät und zum nutzlosen Zwischenstopp bei der Rückwärtsnavigation wird.

Im Shakespeare-Beispiel ist dafür zu sorgen, dass im Querformat-Modus bei der Rückkehr zu einem früheren Details-Fragment (also zu einem früheren Text) die Markierung im **ListView**-Steuerelement korrespondierend wandert. Um dieses Verhalten zu erzielen, implementiert die Klasse **TitlesFragment** die Schnittstelle **FragmentManager.OnBackStackChangedListener**

```
public class TitlesFragment extends ListFragment
    implements FragmentManager.OnBackStackChangedListener {
    . .
}
```

und realisiert die geforderte Rückrufmethode **onBackStackChanged()** folgendermaßen:

```
@Override
public void onBackStackChanged() {
    DetailsFragment details = (DetailsFragment) fm.findFragmentById(R.id.details);
    getListView().setItemChecked(details.getShownIndex(), true);
}
```

Das restaurierte Details-Fragment wird nach dem angezeigten Text befragt, und die Markierung des **ListView**-Steuerelements im **ListFragment**-Objekt wird entsprechend gesetzt.

In der Methode **onActivityCreated()** wird das **TitlesFragment**-Objekt beim Fragment-Manager als Interessent für **BackStackChanged**-Ereignisse angemeldet:

```
fm = getFragmentManager();
fm.addOnBackStackChangedListener(this);
```

### 12.7.4 Animierte Übergänge

Optional lässt sich für eine Fragment-Transaktion eine Animation vereinbaren. Mit dem Vorschlag aus dem originalen Google-Quelltext

```
ft.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE);
```

zeigt sich allerdings keine (beeindruckende) Animation. Die Methode **setCustomAnimations()** führt unter Verwendung von Animations-Definitionen aus dem Ressourcen-Arsenal des Betriebssystems zum erwarteten Ergebnis, wenn sie *vor* der Methode **replace()** ausgerufen wird, z. B.:

```
ft.setCustomAnimations(android.R.animator.fade_in,
                      android.R.animator.fade_out);
```

## 12.8 Argument-Bundles und Fabrikmethoden

Nachdem schon mehrfach von der Klasse **DetailsFragment** der Rede war, wird es Zeit, einen Blick auf deren Definition zu werfen. Im Unterschied zur **ListFragment**-Ableitung **TitleFragment** basiert **DetailsFragment** auf der Klasse **Fragment** im Paket **android.support.v4.app** (in der Support Library):

```
public class DetailsFragment extends android.support.v4.app.Fragment {

    public static DetailsFragment newInstance(int index) {
        DetailsFragment fragment = new DetailsFragment();
        Bundle args = new Bundle();
        args.putInt("index", index);
        fragment.setArguments(args);
        return fragment;
    }

    public int getShownIndex() {
        return getArguments().getInt("index", 0);
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                           Bundle savedInstanceState) {
        if (container == null)
            return null;

        ScrollView scroller = new ScrollView(getActivity());
        TextView text = new TextView(getActivity());
        int padding = (int)TypedValue.applyDimension(TypedValue.COMPLEX_UNIT_DIP,
   4, getActivity().getResources().getDisplayMetrics());
        text.setPadding(padding, padding, padding, padding);
        scroller.addView(text);
        text.setText(getResources().getStringArray(R.array.dialogues)[getShownIndex()]);
        text.setTextSize(TypedValue.COMPLEX_UNIT_SP, 18);

        return scroller;
    }
}
```

Will eine andere Klasse ein **DetailsFragment**-Objekt erzeugen, das einen bestimmten Shakespeare-Text anzeigt, kann sie die statische Fabrikmethode **newInstance()** benutzen und über einen **int**-wertigen Parameter den gewünschten Text bestimmen. Das in **newInstance()** mit dem parameterfreien Konstruktor erstellte **DetailsFragment**-Objekt wird über die **Fragment**-Methode **setArguments()** mit einem **Bundle**-Objekt ausgestattet. Darin befindet sich (über den Namen **index** ansprechbar) die Nummer des darzustellenden Textes.

Eventuell fragen Sie sich, warum man nicht einfach eine Instanzvariable verwendet, um die Konfigurationsinformation im neuen **DetailsFragment**-Objekt zu speichern. Die Antwort ergibt sich daraus, wie Android die Fragmente von zerstörten und später neu angelegten Aktivitäten behandelt: Bei dieser Prozedur bleiben die **Bundle**-Objekte mit den Argumenten der Fragmente erhalten, während die Werte von Instanzvariablen verloren gehen, wenn sich der Programmierer nicht selbst darum kümmert. Im Beispielprogramm spielt dieses automatische Konservieren und Restaurieren von Fragment-Argumenten allerdings keine Rolle.

Das Argument-**Bundle** zu einem Fragment ist übrigens verschieden vom **Bundle**-Objekt, das die Methoden **onCreate()**, **onActivityCreated()** und **onCreateView()** bei der Erstellung des Nachfolgers zu einem zerstörten Fragment per Parameter erhalten. In der Klasse **TitlesFragment** haben wir die Methode **onSaveInstanceState()** überschrieben, um eine Instanzvariable per **Bundle**-Objekt in das Nachfolger-Fragment zu transferieren:

```
@Override
public void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    outState.putInt("curChoice", mCurCheckPosition);
}
```

Das Gegenstück **getArguments()** von **setArguments()** wird in der **DetailsFragment**-Methode **getShownIndex()** benutzt, welche die Nummer des dargestellten Textes liefert.

Weil die Klasse **DetailsFragment** (im Unterschied zur Klasse **TitlesFragment**) *keine* funktionsstüchtige Implementierung der Lebenszyklusmethode **onCreateView()** erbt (vgl. Abschnitt 12.5), muss sie die Methode **onCreateView()** überschreiben, um ihre **View**-Hierarchie zu erstellen.<sup>1</sup>

In der **onCreateView()** - Überschreibung wird ein spezieller Fall vorab behandelt. Wenn Android die Startaktivität bei einem Orientierungswechsel vom Quer- zum Hochformat neu erstellt, werden die beiden zuvor vorhandenen Fragmente ebenfalls neu erzeugt. Dabei wird auch die **DetailsFragment** - Methode **onCreateView()** aufgerufen, erhält jedoch (als zweiten Parameter) statt einer Referenz auf den elterlichen Container den Wert **null**. Der **FrameLayout** - Container, in dem sich das Detailsfragment im Landschaftsmodus befand, existiert nämlich nicht mehr. In dieser Situation beendet sich **onCreateView()** spontan mit der Rückgabe **null**. Es würde keinen großen Schaden anrichten, die Steuerelementhierarchie des Fragments zu erstellen, wäre aber sinnlos, weil diese nicht zu sehen wäre.

Wird beim Aufruf von **onCreateView()** ein Container zur Aufnahme des Fragments mitgeliefert, legt die Methode ein **TextView**-Objekt an. Hier begegnet uns erstmals die mehrfach erwähnte Möglichkeit, Steuerelemente per Programm zu erzeugen, statt sie in einer Layoutdatei zu deklarieren. Dem **TextView**-Konstruktor ist als Kontext die umgebende Aktivität zu übergeben, wobei die benötigte Referenz leicht über die **Fragment**-Methode **getActivity()** zu ermitteln ist. Das **TextView**-Objekt erhält per **setPadding()** einen Innenrand und per **setText()** den gewünschten Inhalt.<sup>2</sup> Dazu ermittelt die **DetailsFragment**-Methode **getShownIndex()** über das Argument-**Bundle** im Fragment, das auch eine Zerstörung und Neuerstellung überlebt, die Textnummer und verwendet diese als Index für die **string-array** - Ressource **R.array.dialogues** mit den Shakespeare-Texten:

---

<sup>1</sup> Die von **Fragment** geerbte Methode **onCreateView()** liefert an ihren Aufrufer statt eines Wurzelements für die Steuerelemente des Fragments lediglich **null** zurück:

```
public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container,
                        Bundle savedInstanceState) {
    return null;
}
```

<sup>2</sup> Die Berechnung des Innenrandes übernehmen wir ausnahmsweise ohne großes Grübeln aus der Vorlage.

```

<resources>
    <string name="app_name">Shakespeare</string>

    <string-array name="titles">
        <item>Henry IV (1)</item>
        .
        .
        <item>King Lear</item>
    </string-array>

    <string-array name="dialogues">
        <item>
            So shaken as we are, so wan with care,\n
            .
            .
            In forwarding this dear expedience.
        </item>
        .
        .
        <item>
            Blow, winds, and crack your cheeks! rage! blow!\n
            You cataracts and hurricanoes, spout\n
            Till you have drench\'d our steeples, drown\'d the cocks!\n
            You sulphurous and thought-executing fires,\n
            Vaunt-couriers to oak-cleaving thunderbolts,\n
            Singe my white head! And thou, all-shaking thunder,\n
            Smite flat the thick rotundity o\' the world!\n
            Crack nature\'s moulds, an germens spill at once,\n
            That make ingrateful man!
        </item>
    </string-array>
</resources>

```

Per **setTextSize()** wird die Schriftgröße in der Einheit **sp** (*scaled pixel*) festgelegt. Um ein vertikales Rollen zu ermöglichen, wird ein Objekt der Klasse **ScrollView** erzeugt und per **addView()** mit dem **TextView** - Objekt befüllt. Dieses **ScrollView**-Objekt wird als **View**-Wurzel des Fragments an den **onCreateView()** - Aufrufer geliefert.

## 12.9 Fragmente im Single-Pane – Modus

Soll eine App sowohl auf großen als auch auf kleinen Display-Flächen ergonomisch agieren, muss sie auch das klassische Stapel-UI unterstützen. Wird die Hauptaktivität des Shakespeare-Beispielprogramms auf einem Gerät im Portrait-Modus (neu) gestartet, erstellt Android (wie im Dual-Pane - Betrieb) aufgrund des **fragment**-Elements in der Layoutdefinitionsdatei ein Objekt der Klasse **TitlesFragement** und ruft dessen Methode **onActivityCreated()** auf. Diese stellt fest, dass kein **FrameLayout** mit der Element-ID **details** zur Aufnahme des Detailsfragments vorhanden ist, und verzichtet auf den Aufruf der Methode **showDetails()**. Wählt der Benutzer im Listenfragment ein Element, wird **onListItemClick()** und darin **showDetails()** aufgerufen. Diese Methode erkennt den Single-Pane - Betrieb und startet eine neue Aktivität über das in Abschnitt 9.3 beschriebene **Intent**-Verfahren:

```

Intent intent = new Intent();
intent.setClass(getActivity(), DetailsActivity.class);
intent.putExtra("index", index);
startActivity(intent);

```

Das zunächst leer erstellte **Intent**-Objekt erfährt per **setClass()**, welche Klasse den Auftrag ausführen soll. Außerdem wird ein Name-Wert - Paar mit der Information über den anzuzeigenden Text in das **Bundle** zum **Intent**-Objekt eingefügt.

Es folgt der erfreuliche knappe Quellcode zur Detailsaktivität:

```

public class DetailsActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        if (getResources().getConfiguration().orientation==Configuration.ORIENTATION_LANDSCAPE) {
            finish();
            return;
        }

        DetailsFragment details;
        if (savedInstanceState == null) {
            details = DetailsFragment.newInstance(getIntent().getExtras().getInt("index", 0));
            getSupportFragmentManager().beginTransaction()
                .add(android.R.id.content, details, "defrag").commit();
        } else
            details = (DetailsFragment) getSupportFragmentManager().findFragmentByTag("defrag");

        setTitle(getTitle() +
            " ("+getStringArray(R.array.titles)[details.getShownIndex()]+")");
    }
}

```

In ihrer Initialisierungsmethode **onCreate()** überprüft die Detailsaktivität, ob sich das Gerät im Landschaftsmodus befindet. Dieser Fall tritt ein, wenn bei aktiver Detailsaktivität ein Wechsel zum Landschaftsmodus stattfindet. In diesem Fall beendet sich die Detailsaktivität, sodass die im Back Stack darunter liegende Startaktivität in den Vordergrund gelangt.

Wenn die Detailsaktivität im Portrait-Modus (ohne Vorgänger-Instanz) neu entsteht, dann wird ...

- ein neues Detailsfragment erstellt, wobei die Nummer des anzuzeigenden Textes aus dem **Intent-Bundle** der Aktivität entnommen wird
- und per Fragment-Transaktion (siehe Abschnitt 12.7) ins Layout der Detailsaktivität aufgenommen.

Weil wir die Anfrage nach dem **FragmentManager**

```

getSupportFragmentManager().beginTransaction()
    .add(android.R.id.content, details, "defrag").commit();

```

an die Aktivität richten, ist die Methode **getSupportFragmentManager()** zu verwenden, um ein Objekt der Klasse **FragmentManager** aus dem Paket **android.support.v4.app** (aus der Support Library) zu erhalten. In Abschnitt 12.6 haben wir die Anfrage **getFragmentManager()** an ein Objekt aus der Klasse **ListFragment** gerichtet.

Die verwendete Überladung der **FragmentTransaction**-Methode **add()**

```
public FragmentTransaction add (int containerViewId, Fragment fragment, String tag)
```

erhält...

- als ersten Parameter die von Android bezogene **int**-Konstante **android.R.id.content**, die auf das Wurzelement im **View**-Baum der Aktivität zeigt. Somit wird die Bildschirmfläche der Aktivität zum Container für das einzufügende Fragment.
- als zweiten Parameter das aufzunehmende Fragment
- als dritten Parameter eine Kennzeichnung des Fragments, über die es später identifiziert werden kann.

Wenn die Aktivität nach einer Zerstörung durch Android im Portrait-Modus restauriert wird, muss kein neues Fragment erzeugt werden.<sup>1</sup> Weil das alte Fragment nach der Nummer des angezeigten Textes befragt werden soll, muss seine Adresse über die **FragmentManager**-Methode **findFragmentByTag()** ermittelt werden:

```
details = (DetailsFragment) getSupportFragmentManager().findFragmentByTag("defrag");
```

Mit der folgenden Anweisung am Ende der **onCreate()** - Methode wird die Titelzeile der Aktivität um eine Herkunftsangabe zum angezeigten Text erweitert:

```
setTitle(getTitle() +  
    " (" + getResources().getStringArray(R.array.titles)[details.getShownIndex()] + ")");
```

Auch im Single-Pane - Modus leisten die beiden Fragmentklassen die wesentliche Arbeit, und das Projekt braucht dank Fragment-Technik nur sehr wenige Code-Zeilen, um zwei deutlich verschiedene Betriebsarten zu unterstützen.

Schließlich muss die Activity zur Details-Anzeige noch in die App-Manifestdatei eingetragen werden.

## 12.10 Übungsaufgaben zu Kapitel 12

1) Welche von den folgenden Aussagen sind richtig bzw. falsch?

1. Eine Fragment-Transaktion wird stets auf dem lokalen Back Stack der Aktivität abgelegt, sodass sie rückgängig gemacht werden kann, wenn der Benutzer den Rückwärtsschalter betätigt.
2. Wird ein Fragment von Android zerstört (z.B. bei einem Orientierungswechsel), so wird ein (mit **setArguments()** eingefügtes) Argument-Bundle automatisch gerettet und in ein Nachfolger-Fragment aufgenommen.
3. Das Layout eines Fragments kann wie das Layout einer Aktivität durch das Inflationieren einer XML-Layoutdefinitionsdatei oder durch Java-Anweisungen erstellt werden.

---

<sup>1</sup> Um die Restaurierung einer zerstörten Details-Aktivität mit dem Android Studio zu simulieren, muss lediglich die Instant Run - Funktion (siehe Abschnitt 4.6) verwendet werden, während sich die Details-Aktivität im Vordergrund befindet.

## 13 Datenbankverwaltung mit Room und SQLite

Viele Apps benötigen für die durch Benutzerfleiß gesammelten Daten eine Persistenzlösung. Eine gelungene Android-App erledigt das Öffnen und Sichern ihrer Daten automatisch. Der Benutzer sollte stets die richtigen Daten zur Hand haben und darauf vertrauen dürfen, dass seine Daten sicher sind (Mednieks et al. 2013, 308ff).

Häufig sind *strukturierte* Daten zu speichern, z. B. die Kontaktpersonen des Benutzers mit zahlreichen Attributen wie Name, Mail-Adresse, Telefonnummer etc. Es liegen identisch aufgebaute Datensätze vor, die man als Zeilen in eine Tabelle eintragen kann. Wenn sich eine Kontakte-App nicht nur um Personen kümmert, sondern z. B. auch Termine verwaltet, bietet sich eine Datenverwaltung in *mehreren* Tabellen an. Wenn eine App strukturierte Daten in einer Tabelle oder in mehreren Tabellen persistent (über den aktuelle Sitzung hinaus) speichern muss, ist die Beauftragung eines Datenbankmanagementsystems (DBMS) gegenüber der Speicherung in selbst verwalteten Dateien zu bevorzugen.

Heute arbeitet praktisch jedes DBMS mit der **relationalen Datenbankstruktur**, und der Zugriff auf relationale Datenbanken ist über die Abfragesprache SQL weitgehend standardisiert. In Android ist mit **SQLite** ein an die mobile Umgebung gut angepasstes, schlankes, ohne Konfigurationsaufwand einsetzbares relationales DBMS integriert. Ein wichtiges Designziel bei SQLite war die Robustheit unter widrigen Umständen (z. B. Stromausfall, Speicherzugang, rabierte Prozessbeendigung durch das Betriebssystem), sodass praktisch nie eine Datenbank verloren geht. Über die Hälfte des SQLite-Quellcodes soll sich mit Tests beschäftigen (Mednieks et al. 2013, S. 296), was durchaus als Orientierungswert für andere Projekte mit einem hohen Qualitätsanspruch taugt.

In SQLite kommt eine komplette Datenbank in einer *einzigem* Datei unter. Wir werden später im Rahmen eines Projekts mit dem Paketnamen `de.zimkand.eventdiary` die Datenbankdatei `eventdiary.db` erstellen. Diese befindet sich auf einem Android-Gerät im folgenden Ordner:

`/data/data/de.zimkand.eventdiary/databases/`

Auf der Google I/O - Entwicklerkonferenz 2017 wurden unter dem Namen **Android Architecture Components** eine Reihe von innovativen Bibliotheken für die Android-Entwicklung vorgestellt. Dazu gehört unter dem Namen **Room Persistence Library** (Kurzbezeichnung: **Room**) auch eine **ORM-Lösung (Object Relational Mapper)**. Diese Abstraktionsschicht über SQLite ermöglicht einen Datenbankzugriff im objektorientierten Programmierstil, ohne die Leistungsfähigkeit von SQLite einzuschränken.

Im Vergleich zur weiterhin möglichen direkten Verwendung von SQLite bestehen erhebliche Vorteile, z. B.:

- Das Erstellen von SQL-Kommandos wird dem Entwickler weitgehend abgenommen, sodass Datenbankanwendungen ...
  - schnell,
  - mit wenig Fehlern
  - und mit wenig lästigem Routine-Code (engl.: *boilerplate code*)erstellt werden können.
- Wenn doch noch SQL-Kommandos erforderlich sind (z.B. beim **SELECT**-Kommando), dann findet eine Überprüfung der SQL-Syntax beim Übersetzen (also im Android Studio) statt, was die Entwicklung beschleunigt und die Gefahr von Laufzeitfehlern minimiert.
- Eine durchgehend objektorientierte Softwareentwicklung ist gegenüber der bei direkter SQLite-Verwendung anzutreffenden Java/SQL - Kombination zu bevorzugen. ORM-

Lösungen sind auch auf anderen Software-Plattformen Standard für die professionelle Entwicklung.<sup>1</sup>

- Per Voreinstellung untersagt Room Datenbankzugriffe im UI-Thread, um eine Blockade der Bedienoberfläche zu vermeiden. Allerdings kann man diese Voreinstellung abschalten, was wir der Einfachheit halber tun werden.<sup>2</sup>

Google empfiehlt den Datenbankzugriff per Room mit allem Nachdruck (Hervorhebung aus dem Original):<sup>3</sup>

Because Room takes care of these concerns for you, we **highly recommend** using Room instead of SQLite.

### 13.1 Benötigte Typen für den Datenbankzugriff via Room

In diesem Abschnitt wird erläutert, welche Klassen und Schnittstellen für den Zugriff auf eine Datenbankdatei via Room benötigt werden. Wir beschränken uns auf den einfachen und häufig anzu treffenden Fall einer Datenbank mit einer *einzigsten* Tabelle.

Das Room-Framework verwendet in erheblichem Umfang die *deklarative* Programmierung, wobei Annotationen aus dem Paket **android.arch.persistence.room** eine entscheidende Rolle spielen.<sup>4</sup>

Über die anschließende Darstellung hinausgehende Informationen finden sich z.B. in der Googles Online-Dokumentation für Android-Entwickler.<sup>5</sup>

#### 13.1.1 Entities

Als *Entität* bezeichnet man eine persistente Klasse, deren Objekte in der Datenbank gespeichert werden sollen. Sie wird auf eine Datenbanktabelle abgebildet, wobei jedem Objekt der Entität eine Zeile der Tabelle entspricht. Die Werte eines Objekts für die Instanzvariablen der persistenten Klasse, werden auf die Tabellenspalten abgebildet. Room legt eine Datenbanktabelle und deren Schema (Namen und Typen der Attribute bzw. Spalten) nötigenfalls passend zur Definition der zugehörigen Entitätsklasse automatisch an.

Zur Room-Entitätsklasse wird eine gewöhnliche Java-Klasse durch die anschließend geschriebenen Besonderheiten:

---

<sup>1</sup> In der Java Standard Edition, die von der Firma Oracle betreut wird, kommt als ORM-Lösung das *Java Persistence API* (JPA) zum Einsatz. Microsoft schlägt im .NET-Framework mit dem *Entity Framework* die Brücke zwischen der objektorientierten Programmierung und relationalen Datenbanken.

<sup>2</sup> In Kapitel 10 haben Sie Multithreading-Lösungen kennengelernt, die Sie in praxisgerechten Apps mit Datenbankzugriff verwenden können.

<sup>3</sup> <https://developer.android.com/training/data-storage/room/>

<sup>4</sup> Über Java-Annotationen informiert z.B. der Abschnitt 9.5 in Baltes-Götz & Götz (2018).

<sup>5</sup> <https://developer.android.com/training/data-storage/room/>

### 13.1.1.1 Annotation @Entity

Der Klasse ist die Annotation **@Entity** voranzustellen, z.B.:

```
@Entity(tableName = "users")
public class User {
    @PrimaryKey
    private int id;
    ...
}
```

Optional sind Annotationselemente möglich, z.B.:

- **tableName**

Durch dieses Annotationselement mit dem Datentyp **String** kann ein Tabellenname festgelegt werden, um die Voreinstellung (Tabellenname = Klassenname) abzuändern (siehe Beispiel).

- **primaryKeys**

Das Annotationselement **primaryKeys** mit dem Datentyp **String[]** ist zu verwenden, wenn *mehrere* Instanzvariablen *gemeinsam* den Primärschlüssel bilden, z.B.:

```
@Entity(primaryKeys = {"firstName", "lastName"})
```

### 13.1.1.2 Öffentlich zugängliche Felder

Eine Entitätsklasse enthält (außer Konstruktoren) nur Instanzvariablen und zugehörige Methoden zum Lesen und Schreiben durch fremde Klassen (Getter und Setter nach der Java-Beans-Konvention), z.B.:

```
@Entity(tableName = "users")
public class User {
    @PrimaryKey
    private int id;

    @ColumnInfo(name = "first_name")
    private String firstName;

    @ColumnInfo(name = "last_name")
    private String lastName;

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }

    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

Es ist möglich, aber im Sinne der Datenkapselung unerwünscht, öffentlich zugängliche Felder zu verwenden und auf Getter/Setter zu verzichten.

Per Voreinstellung übernimmt Room als Namen der Tabellenspalten die Namen der Instanzvariablen. Über die Annotation **@ColumnInfo** lassen sich alternative Spaltennamen wählen (siehe Beispiel).

Außerdem lässt sich über die Annotation **@Ignore** ein Feld von der Persistenz ausschließen, z.B.:

```
@Ignore  
private String passwd;
```

### 13.1.1.3 Primärschlüssel

Für jede Entitätsklasse bzw. Tabelle muss ein Primärschlüssel gewählt werden, der eine eindeutige Identifikation der Zeilen erlaubt. Meist wird über die Annotation **@PrimaryKey** ein einzelnes Feld ausgezeichnet, z.B.:

```
@PrimaryKey  
private int id;
```

Bei einem ganzzahligen Primärindex kann man über das Annotationselement **autoGenerate** mit dem Datentyp **boolean** dafür sorgen, dass der Wert für eine neu eingefügte Tabellenzeile durch automatisches Inkrementieren ermittelt wird.

```
@PrimaryKey(autoGenerate = true)  
private int id;
```

Was zu tun ist, wenn *mehrere* Instanzvariablen *gemeinsam* den Primärschlüssel bilden, war schon in Abschnitt 13.1.1.1 zu sehen.

### 13.1.2 DAO (Data Access Object)

Die beim **Data Access Object** benötigten Verhaltenskompetenzen definiert man über eine Schnittstelle oder eine abstrakte Klasse mit der Annotation **@Dao**. Hier werden durch Schnittstellenmethoden bzw. durch abstrakte Instanzmethoden die benötigten SELECT-, INSERT-, UPDATE- und DELETE-Operationen beschrieben. Die Implementationen der Methoden erstellt Room automatisch, was wohl letzte Zweifel an der Effizienz des Datenbankzugriffs per Room-Framework im Vergleich zur direkten SQLite-Programmierung ausräumen dürfte.

Eine Methode zur Realisation einer INSERT-, UPDATE- oder DELETE-Operation akzeptiert Parameter mit dem (Element-)Typ einer Entitätsklasse, wobei pro Parameter ...

- ein einzelnes Objekt,
- ein Array von Objekten
- oder eine Kollektion (z.B. Liste) von Objekten

angegeben werden kann.

Sind laut Aktualparameterliste *mehrere* Objekte zu bearbeiten, dann führt Room automatisch eine **Transaktion** durch, d.h.:

- entweder werden alle Einzelaufträge ausgeführt,
- oder die Datenbank bleibt unverändert.

### 13.1.2.1 INSERT

Im folgenden Beispiel aus der Online-Dokumentation für Android-Entwickler wird die Schnittstelle MyDAO definiert und per **@Dao**-Annotation als Grundlage für das Data Access Objekt deklariert:<sup>1</sup>

```
@Dao
public interface MyDAO {
    @Insert
    void insertUser(User user);

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    void insertUsers(User... users);

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    void insertUsers(List<User> users);
}
```

Die aufgrund der **@Insert** - Annotation von Room zu implementierende Methode **insertUser()** sichert ein einzelnes Objekt der Entitätsklasse **User**. Wird statt **void** der Rückgabetyp **long** angegeben, dann liefert die Room-Implementation die Zeilennummer des neu eingefügten Objekts. Wie bei allen anderen Methoden wurde der Zugriffsmodifikator **public** bewusst weggelassen, weil in einer Java-Schnittstelle die Zugriffsstufe **public** voreingestellt ist.

Von den beiden Überladungen der Methode **insertUsers()** besitzt die erste einen **varargs**-Parameter, akzeptiert also beliebig viele **User**-Objekte als durch Komma getrennte Aktualparameter. An die zweite Überladung ist eine Liste von **User**-Objekten zu übergeben. Statt **void** der Rückgabetyp **long[]** bzw. **List[Long]** verwendet werden, damit die Room-Implementation die Zeilennummern der eingefügten Objekte liefert.

Beide Überladungen besitzen eine **@Insert** - Annotation mit dem Element **onConflict** (Datentyp: **int**), und dessen Wert **OnConflictStrategy.REPLACE** (eine Konstante aus der Annotation **OnConflictStrategy**) sorgt dafür, dass beim Einfügen eine bereits vorhandene Tabellenzeile mit demselben Wert für den Primärschlüssel ersetzt wird.

### 13.1.2.2 UPDATE

Erhält in einer DAO-Schnittstelle eine Methode die Annotation **@Update**, dann resultiert eine Room-Implementation, welche die zu den Parameterobjekten gehörigen Datenbankzeilen aktualisiert, z.B.:

```
@Dao
public interface MyDAO {
    . . .
    @Update
    void updateUsers(User... users);
    . . .
}
```

Wenn keine zu aktualisierende Datenbankzeile (mit passendem Primärschlüssel) existiert, wird die Datenbank nicht verändert. Wird als Rückgabetyp **int** statt **void** verwendet, liefert die Room-Implementation die Anzahl der geänderten Zeilen.

---

<sup>1</sup> <https://developer.android.com/training/data-storage/room/accessing-data>

### 13.1.2.3 DELETE

Erhält in einer DAO-Schnittstelle eine Methode die Annotation `@Delete`, dann resultiert eine Room-Implementation, welche die zu den Parameterobjekten gehörigen Datenbankzeilen löscht, z.B.:

```
@Dao
public interface MyDAO {
    ...
    @Delete
    void deleteUsers(User... users);
    ...
}
```

Wenn keine zu löschenende Datenbankzeile (mit passendem Primärschlüssel) existiert, wird die Datenbank nicht verändert. Wird als Rückgabetyp `int` statt `void` verwendet, liefert die Room-Implementation die Anzahl der gelöschten Zeilen.

### 13.1.2.4 SELECT

Erhält in einer DAO- Schnittstelle eine Methode die Annotation `@Query`, dann resultiert eine Room-Implementation, die eine Datenbankabfrage ausführt und einen Array mit einer Entitätsklasse als Elementtyp zurückliefert. Über das `@Query` - Annotationselement mit dem (nicht anzugebenden) Namen `value` und dem Datentyp `String` gibt man das **SELECT**-Kommando an. Im ersten Beispiel ist eine simple Abfrage zu sehen, die alle Zeilen der Tabelle `users` liefern soll. An dieser Stelle soll die bemerkenswerte Fähigkeit des Room-Frameworks, bereits beim Übersetzen SQL-Fehler zu entdecken, demonstriert werden (vgl. Abschnitt 13.1.1.1 zum korrekten Tabellennamen):

```
@Query("SELECT * FROM user")
User[] loadAllUsers();
```

Der Fehler ist schnell behoben:

```
@Dao
public interface MyDAO {
    ...
    @Query("SELECT * FROM users")
    User[] loadAllUsers();
    ...
}
```

Um eine parametrisierte Abfrage zu erhalten, definiert man in der Schnittstellenmethode passende Parameter und verwendet diese mit einem vorangestellten Doppelpunkt im **SELECT**-Kommando, z.B.:

```
@Query("SELECT * FROM users WHERE first_name = :fn")
User[] loadAllUsersWithFirstName(String fn);
```

Bei Abfragen, die mehrere Treffer liefern können, ist neben einem Array als Rückgabe auch eine Liste erlaubt, z.B.:

```
@Query("SELECT * FROM users WHERE first_name = :fn")
List<User> loadAllUsersWithFirstName(String fn);
```

Bei Abfragen, die maximal *einen* Treffer liefern können, verwendet man eine Entitätsklasse als Rückgabetyp, z.B.:

```
@Query("SELECT * FROM users WHERE id = :userid")
User getUser(long userid);
```

### 13.1.3 RoomDatabase

Um eine Datenbank per Room-Framework in ein Programm zu integrieren, müssen wir noch eine abstrakt definierte eigene Klasse von **RoomDataBase** im Paket **android.arch.persistence.room** ableiten und mit der Annotation **@Database** ausstatten, z.B.:

```
@Database(entities = {User.class}, version = 1)
public abstract class AppDatabase extends RoomDatabase {
    public abstract MyDAO getUserDAO();
}
```

Die **@Database** - Annotation hat zwei wichtige Elemente:

- **entities**

Als Wert für dieses Element vom Datentyp **Class[]** wird ein Array mit den **Class**-Dateien zu den Entitätsklassen erwartet. Im folgenden Beispiel ist eine Datenbank mit zwei Tabellen bzw. Entitätsklassen im Spiel:

```
@Database(entities = {User.class, Book.class}, version = 1)
```

- **version**

Die Versionsangabe spielt eine wichtige Rolle, um Kompatibilitätsprobleme zu verhindern.

In der abstrakten **RoomDataBase**-Ableitung ist eine abstrakte Methode zu deklarieren, die keinen Parameter besitzt und ein Objekt vom eigenen DAO-Typ abliefernt.

Um Room aufzufordern, die abstrakten Methoden in der DAO-Schnittstelle und der **RoomDataBase**-Ableitung zu implementieren und ein nutzbares Data Access Object zu erstellen, kann man wie im folgenden Beispiel verfahren:

```
MyDAO myDAO = Room.databaseBuilder(getApplicationContext(),
        AppDatabase.class, "users.db")
    .allowMainThreadQueries()
    .build()
    .getUserDAO();
```

Hier wird zunächst die statische Methode **databaseBuilder()** aus der Klasse **Room** aufgerufen, welche den Anwendungskontext, die **Class**-Datei zur eigenen **RoomDataBase**-Ableitung sowie den Namen der Datenbankdatei als Parameter erhält und im Beispiel ein Objekt der Klasse **RoomDataBase.Builder<AppDatabase>** ab liefert.

An dieses Objekt ergeht die Aufforderung, das Verbot von Datenbankzugriffen im Haupt-Thread aufzuheben. Als Rückgabe der zuständigen Methode **allowMainThreadQueries()** liefert das angesprochene Objekt eine Referenz auf sich selbst, sodass man gleich die Methode **build()** aufrufen kann, welche die Datenbank kreiert bzw. initialisiert. Man erhält als Rückgabe ein Objekt der eigenen **RoomDataBase**-Ableitung, von dem man das Data Access Object anfordern kann. Wie bequem man dann über die DAO-Schnittstellenmethoden auf die Datenbank zugreifen kann, wird das Beispiel in Abschnitt 13.2 zeigen.

### 13.1.4 Typkonverter

Besitzt eine Entitätsklasse Instanzvariablen vom Typ einer Klasse, dann sind Typkonverter für die Abbildung in eine Datenbanktabelle erforderlich. Das gilt z.B. für eine Instanzvariable vom Typ **Date**, den SQLite nicht kennt:

```
@ColumnInfo(name = "datetime")
private Date dateTIme;
```

Weil ein **Date**-Objekt Datum und Uhrzeit letztlich in einer **long**-Variablen mit der Anzahl der Millisekunden seit dem 1. Januar, 1970, 00:00:00 GMT speichert, kann ein **Date**-Objekt auf einfache

Weise in einer SQLite-Datenbank abgelegt werden. Bei den erforderlichen Konvertierungen kann man zudem auf bewährte Methoden im Java-API zurückgreifen.

Man definiert eine Klasse mit Konvertierungsmethoden für beide Richtungen, welche die Annotation **@TypeConverter** erhalten, z.B.:

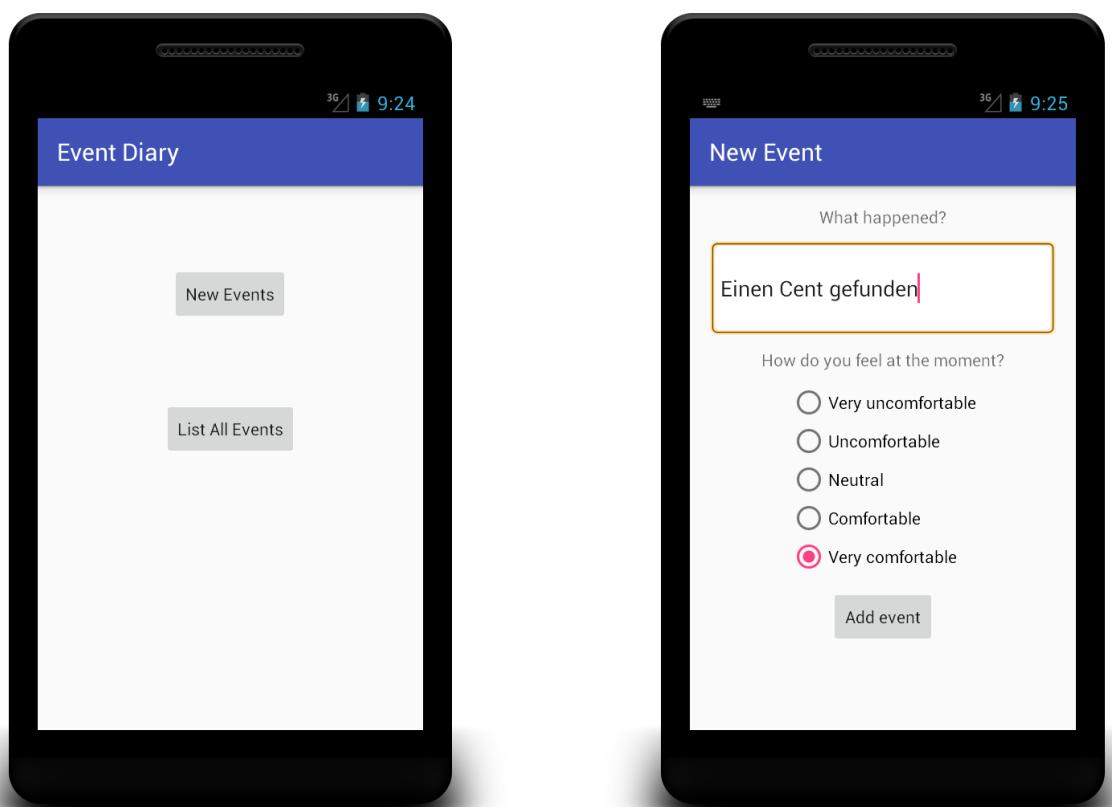
```
public class DateLongConverter {  
    @TypeConverter  
    public long convertDateToLong(Date date) {  
        return date.getTime();  
    }  
  
    @TypeConverter  
    public Date convertLongToDate(long time) {  
        return new Date(time);  
    }  
}
```

Der eigenen **RoomDataBase**-Ableitung (vgl. Abschnitt 13.1.3) müssen Typkonverter durch die Annotation **@TypeConverters** bekannt gegeben werden. Dazu erhält das Annotationselement mit dem (nicht anzugebenden) Namen **value** und dem Datentyp **Class<?>[]** eine Initialisierungsliste mit den **Class**-Objekten zu den Typkonvertern, z.B.:

```
@Database(entities = {Event.class}, version = 1)  
@TypeConverters({DateLongConverter.class})  
public abstract class AppDatabase extends RoomDatabase {  
    public abstract EventDAO getEventDAO();  
}
```

## 13.2 Ereignistagebuch anlegen und befüllen

Wir erstellen nun eine App, mit der sich ein Ereignistagebuch befüllen und anzeigen lässt. In der Startaktivität (linkes Bildschirmfoto) werden die beiden Funktionen angeboten:



Über den oberen Schalter auf dem Startbildschirm ist eine Aktivität erreichbar, die Ereignisbeschreibungen entgegen nimmt (rechtes Bildschirmfoto). Der Benutzer liefert eine Beschreibung zu einem aktuellen Ereignis und die resultierende Stimmung. In der Datenbank werden zusätzlich auch Datum und Uhrzeit festgehalten. In das Beispiel sind einige Ideen aus Künneß (2012, Kapitel 11) eingeflossen.

Wir gestalten ein einfaches **LinearLayout** für die Startaktivität:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center_horizontal"
    android:orientation="vertical"
    tools:context=".MainActivity">

    <Button
        android:id="@+id/btnEvents"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="48dp"
        android:text="@string/new_event"
        android:textAllCaps="false" />

    <Button
        android:id="@+id/btnList"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="64dp"
        android:text="@string/show_list"
        android:textAllCaps="false" />
</LinearLayout>
```

Dieses Layout mit den beiden Schaltern wird in der Lebenszyklusmethode **onCreate()** aus der Layoutdefinitionsdatei geladen:

```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

Bevor in **onCreate()** der Datenbankzugriff initialisiert werden kann, müssen die in Abschnitt 13.1 beschriebenen Typen definiert werden.

### 13.2.1 Room in die Abhängigkeitsliste des Moduls aufnehmen

Um die erforderliche Room Persistence Library nutzen zu können, muss diese in die Abhängigkeitsliste des Projekt-Moduls aufgenommen werden. Öffnen Sie dazu die Modul-Variante der Datei **build.gradle**, und ergänzen Sie die folgenden Zeilen

```
def room_version = "1.1.1"
implementation "android.arch.persistence.room:runtime:$room_version"
annotationProcessor "android.arch.persistence.room:compiler:$room_version"
```

im Block **dependencies** auf, z.B.:

```

dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'com.android.support:appcompat-v7:27.1.1'
    implementation 'com.android.support.constraint:constraint-layout:1.1.2'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'com.android.support.test:runner:1.0.2'
    androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.2'
    def room_version = "1.1.1"
    implementation "android.arch.persistence.room:runtime:$room_version"
    annotationProcessor "android.arch.persistence.room:compiler:$room_version"
}

```

Lassen Sie das Projekt anschließend synchronisieren, z.B. über den neu eingeblendeten und gelb hinterlegten Link in der oberen rechten Fensterecke.

### 13.2.2 Entity Event

Wir definieren eine Entitätsklasse namens **Event** für die zu verwaltenden und zu persistierenden Ereignisse. Dazu legen wir über den folgenden Befehl aus dem Kontextmenü zum Paket `de.zimkand.eventdiary` im Projektxplorer eine neue Klasse an:

#### New > Java Class

Die Klasse **Event** erhält die Annotation `@Entity` mit dem optionalen **String**-Element **tableName** zur Benennung der zugehörigen Datenbanktabelle:

```

@Entity(tableName = "events")
public class Event {
}

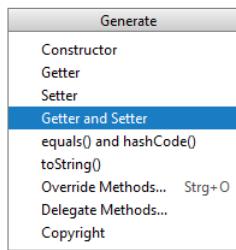
```

Per Voreinstellung erhält die Datenbanktabelle den Namen der Entitätsklasse.

Ein Ereignis benötigt die folgenden Attribute bzw. Instanzvariablen:

- Eine ganzzahlige Kennung mit dem Namen **id** und dem Datentyp **long** für ein ereignisreiches Leben  
Dieses Feld soll als Primärschlüssel der Tabelle dienen und dabei automatisch generiert werden. Dies erreichen wir durch eine Annotation `@PrimaryKey`, deren **boolean**-Element **autoGenerate** den Wert **true** erhält. Außerdem soll die Datenbankspalte über die Annotation `@ColumnInfo` mit dem Element **name** abweichend von der Instanzvariablen benannt werden. Insgesamt ergibt sich die folgende Deklaration:  
`@PrimaryKey(autoGenerate = true)
@ColumnInfo(name = "eventid")
private long id;`
- Datum und Zeit werden für ein Ereignis in der Instanzvariablen **dateTime** vom Typ **Date** festgehalten:  
`@ColumnInfo(name = "datetime")
private Date dateTime;`
- In der **String**-Instanzvariablen **desc** wird eine Ereignisbeschreibung hinterlegt, und in der **int**-Instanzvariable **mood** soll eine Stimmungsbeurteilung auf einer Skala von 1 bis 5 vorgenommen werden:  
`private String desc;
private int mood;`

Weil wir private Felder angelegt haben, werden für Room öffentliche Zugriffsmethoden zum Lesen und Schreiben benötigt. Um diese sogenannten Getter und Setter vom Android Studio erstellen zu lassen, wählen wir aus dem Kontextmenü zur Klassendefinition das Item **Generate** und aus dem folgenden PopUp-Menü das Item **Getter und Setter**:



Schließlich wird noch ein Konstruktor für die Entitätsklasse zur Verfügung gestellt:

```
public Event(Date dateTime, String desc, int mood) {
    this.dateTime = dateTime;
    this.desc = desc;
    this.mood = mood;
}
```

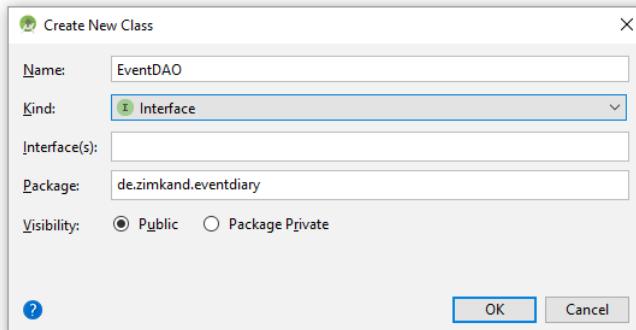
Im Konstruktor wird das Feld `id` *nicht* initialisiert, weil sein Wert von der Datenbank generiert wird.

### 13.2.3 Schnittstelle EventDAO als Grundlage für das Data Access Object

Wir definieren eine Schnittstelle namens `EventDAO`, um die benötigten und von Room zu implementierenden Datenbankzugriffsmethoden zu deklarieren. Dazu legen wir über den folgenden Befehl aus dem Kontextmenü zum Paket `de.zimkand.eventdiary` im Projektexplorer

**New > Java Class**

eine neue Schnittstelle an:



Die Schnittstelle `EventDAO` erhält die Annotation `@Dao`:

```
@Dao
public interface EventDAO {
```

Wir benötigen die folgenden Datenbankzugriffsmethoden:

- Durch eine Abfragemethode mit dem Namen `getEvents()` und mit dem Rückgabetyp `List<Event>` werden alle Zeilen in der Ereignistabelle angefordert:

```
@Query("SELECT * FROM events")
List<Event> getEvents();
```

Das erforderliche `SELECT`-Kommando wird als `String`-Wert des Standardelements `value` in der Annotation `@Query` an die Methode geheftet.

- Durch die Methode `insert()` mit dem Rückgabetyp **void** werden die per `varargs`-Parameter benannten Event-Objekte in die Ereignistabelle eingefügt:

```
@Insert(onConflict = OnConflictStrategy.REPLACE)  
void insert(Event... events);
```

In der Annotation `@Insert` wird durch den Wert **OnConflictStrategy.REPLACE** für das Element **onConflict** dafür gesorgt, dass ggf. vorhandene Datenbankzeilen mit einer identischen Kennung ersetzt werden.

- Durch die Methode `update()` mit dem Rückgabetyp **void** werden die per `varargs`-Parameter benannten Zeilen in der Ereignistabelle aktualisiert:

```
@Update  
void update(Event... events);
```

Der Methode wird die Annotation `@Update` vorangestellt.

- Durch die Methode `delete()` mit dem Rückgabetyp **void** werden die per `varargs`-Parameter benannten Zeilen aus der Ereignistabelle gelöscht:

```
@Delete  
void delete(Event... events);
```

Der Methode wird die Annotation `@Delete` vorangestellt.

Wer sich für die Implementation der Schnittstelle **EventDAO** durch das Room-Framework interessiert, sollte sich die Klasse `EventDAOImpl.java` im folgenden Projektordner ansehen:

```
...|app|build|generated|source|apt|debug|de|zimkand|eventdiary
```

### 13.2.4 Typkonverter

Weil die Event-Instanzvariable `dateTime` vom Typ der Klasse **Date** ist und daher nicht direkt in einer SQLite-Tabelle abgelegt werden kann, benötigen wir eine Klasse, die sich um die Typumwandlung in beiden Richtungen kümmert (siehe Abschnitt 13.1.4). Dazu legen wir über den folgenden Befehl aus dem Kontextmenü zum Paket `de.zimkand.eventdiary` im Projektexplorer eine neue Klasse namens `DateLongConverter` an:

#### New > Java Class

Weil diese Klasse schon in Abschnitt 13.1.4 als Beispiel aufgetreten ist, sind hier keine weiteren Erläuterungen erforderlich.

### 13.2.5 RoomDatabase-Ableitung

Wir benötigen schließlich noch eine projektspezifische, abstrakte Ableitung der Klasse **RoomDatabase** aus dem Paket `android.arch.persistence.room` (siehe Abschnitt 13.1.3). Dazu legen wir über den folgenden Befehl aus dem Kontextmenü zum Paket `de.zimkand.eventdiary` im Projektexplorer eine neue Klasse namens `EventDatabase` an:

#### New > Java Class

Die Klasse `EventDatabase` erhält die Annotation `@Database` mit Elementen für die Liste der Entitätsklassen und die Version:

```

@Database(entities = {Event.class}, version = 1)
@TypeConverters({DateLongConverter.class})
public abstract class EventDatabase extends RoomDatabase {
    private static EventDatabase instance;

    public abstract EventDAO getEventDAO();

    public static synchronized EventDatabase getInstance(Context context) {
        if (instance == null) {
            instance = Room.databaseBuilder(context, EventDatabase.class,
                "eventdiary.db")
                .allowMainThreadQueries()
                .build();
        }
        return instance;
    }
}

```

In einer weiteren Annotation zur Klasse `EventDatabase` wird der Typkonverter zum Datums/Zeit - Feld benannt.

Die abstrakt zu definierende `EventDatabase`-Methode `getEventDAO()` hat keinen Parameter und liefert ein Objekt vom Typ der DAO-Schnittstelle.

Weil jede `RoomDatabase`-Instanz einen hohen Aufwand verursacht, wird durch das sogenannte *Singleton-Muster* dafür gesorgt, dass in der App nur *ein* `EventDatabase`-Objekt entsteht.<sup>1</sup> Dazu wird die Methode `getInstance()` angeboten, die eine Referenz auf das einzige `EventDatabase`-Objekt liefert und das Objekt nötigenfalls erstellt.

Um einen störungsfreien Multithreading-Betrieb sicherzustellen, wird die Methode `getInstance()` synchronisiert, sodass sie nicht in mehreren Threads gleichzeitig ausgeführt werden kann. Was simultane Datenbankzugriffe aus mehreren Threads betrifft, verlassen wir uns auf die Thread-Sicherheit der zugrundeliegenden SQLite-Datenbank.<sup>2</sup>

Wer sich für die Implementation der Klasse `EventDatabase` durch das Room-Framework interessiert, sollte sich die Klasse `EventDatabaseImpl.java` im folgenden Projektordner ansehen:

...\\app\\build\\generated\\source\\apt\\debug\\de\\zimkand\\eventdiary

In der überschriebenen Methode `getEventDAO()` wird per Singleton-Muster dafür gesorgt, dass ein `EventDatabase` - Objekt nur *ein* `EventDAO`- Objekt erzeugt:

```

@Override
public EventDAO getEventDAO() {
    if (_eventDAO != null) {
        return _eventDAO;
    } else {
        synchronized(this) {
            if(_eventDAO == null) {
                _eventDAO = new EventDAO_Impl(this);
            }
        }
        return _eventDAO;
    }
}

```

<sup>1</sup> <https://developer.android.com/training/data-storage/room/>

<sup>2</sup> <https://www.sqlite.org/faq.html>

Das in mehreren Aktivitäten unserer Ereignistagebuch-Anwendung benötigte EventDAO-Objekt soll in der Startaktivität verfügbar gemacht werden. Wir deklarieren eine private und statische Referenzvariable zum Data Access Object

```
private static EventDAO eventDAO;
```

und initialisieren diese in der Lebenszyklusmethode **onCreate()**:

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    eventDAO = EventDatabase.getInstance(this).getEventDAO();

    . . .

}
```

Damit andere Aktivitäten die Datenbankverbindung bequem nutzen können, wird eine öffentliche und statische Methode erstellt, die eine Referenz auf das Data Access Object liefert:

```
public static EventDAO getEventDAO() {
    return eventDAO;
}
```

### 13.3 Ereignisliste aus der Datenbank abrufen und per ListView anzeigen

#### 13.3.1 Aktivität zum Anzeigen der Ereignisliste

Wir erstellen nun eine Aktivität namens **EventListActivity**, die eine Liste aller Ereignisse anzeigt. Um eine neue, leere Aktivität in ein Projekt einzufügen, wählt man im Projektxplorer aus dem Kontextmenü zum Paket mit den Klassen des Projekts (`de.zimkand.eventdiary`) den Eintrag

**New > Activity > Empty Activity**

Als **Activity Name** eignet sich z. B. **EventListActivity**. Für den Eintrag in der Manifestdatei sorgt das Android Studio:

```
<activity
    android:name=".EventListActivity"
    android:label="@string/title_activity_event_list" >
</activity>
```

Im Beispiel wird eine **String**-Ressource zur Benennung der Aktivität verwendet.

Das Layout der Aktivität beschränkt sich auf ein **ListView**-Steuerelement:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <ListView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/listView" />
</LinearLayout>
```

In der der Lebenszyklusmethode **onCreate()** der **EventListActivity**

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_event_list);

    EventDAO eventDAO = MainActivity.getEventDAO();
    List<Event> allEvents = eventDAO.getEvents();
    if (allEvents.size() == 0) {
        eventDAO.insert(new Event(new Date(), "5 Cent gefunden", 5));
        allEvents = eventDAO.getEvents();
    }

    ArrayAdapter adapter = new EventArrayAdapter(this, R.layout.activity_event_entry,
  allEvents);
    ListView lv = findViewById(R.id.listView);
    lv.setAdapter(adapter);
}

```

übertragen wir aus der Datenbanktabelle alle Ereignisse in das Objekt `allEvents` vom Typ `List<Event>`. Wir besorgen uns eine Referenz zum DAO bei der Startaktivität und lassen die Abfrage in der EventDAO-Methode `getEvents()` ausführen:

```

EventDAO eventDAO = MainActivity.getEventDAO();
List<Event> allEvents = eventDAO.getEvents();

```

Im Fall einer noch leeren Ereignistabelle fügen wir mit der EventDAO-Methode `insert()` ein Ereignis ein

```
eventDAO.insert(new Event(new Date(), "5 Cent gefunden", 5));
```

und wiederholen dann die Abfrage.

Damit das `ListView`-Steuerelement in der `EventListActivity` - Bedienoberfläche die Ereignisse im `List<Event>` - Kollektionsobjekt `allEvents` anzeigen kann, muss noch ein vermittelndes Objekt aus einer selbst definierten Adapterklasse tätig werden, die anschließend erläutert wird:

```

EventArrayAdapter adapter = new EventArrayAdapter(this,
   R.layout.activity_event_entry, allEvents);
ListView lv = findViewById(R.id.listView);
lv.setAdapter(adapter);

```

### 13.3.2 Adapter-Klasse für das ListView-Steuerelement

Wir legen im Paket `de.zimkand.eventdiary` unter dem Namen `EventArrayAdapter` eine weitere Klasse an und leiten diese von `ArrayAdapter<Event>` ab:

```

public class EventArrayAdapter extends ArrayAdapter<Event> {
    private SimpleDateFormat sdf = new SimpleDateFormat(
        "EEE, dd MMM yyyy, HH:mm:ss");
    private Context context;
    private int layout;
    private List<Event> events;

    public EventArrayAdapter(Context context, int layout, List<Event> events) {
        super(context, layout, events);
        this.context = context;
        this.layout = layout;
        this.events = events;
    }
}

```

```
@Override  
public View getView(int position, View convertView, ViewGroup parent) {  
    Event event = events.get(position);  
    if (convertView == null)  
        convertView = LayoutInflater.from(context).inflate(layout, null);  
    TextView tvDateTime = convertView.findViewById(R.id.dateTime);  
    TextView tvMood = convertView.findViewById(R.id.moodValue);  
    TextView tvDesc = convertView.findViewById(R.id.eventDesc);  
    Date date = event.getDateTime();  
    tvDateTime.setText(sdf.format(date));  
    tvMood.setText(context.getString(R.string.moodvalue) + " " +  
        String.valueOf(event.getMood()));  
    tvDesc.setText(event.getDesc());  
    return convertView;  
}  
}
```

Neben einem Objekt zur Datums- und Zeitformatierung erhält die Klasse Instanzvariablen für ...

- den Anwendungs-**Context**
- die Ressourcen-Kennung zu einer Layout-Definition für die Darstellung eines einzelnen Ereignisses
- das **List<Event>** - Kollektionsobjekt mit den Ereignissen

Im **EventArrayAdapter** - Konstruktor werden die drei beschriebenen Instanzvariablen initialisiert. Außerdem werden die Initialisierungswerte an einen Basisklassenkonstruktor weitergeleitet.

Die wichtigste Rolle des Adapters besteht darin, in der Rückrufmethode **getView()** für jedes Listen-Element das anzugebende **View**-Objekt zu liefern. Falls die an **getView()** als zweiter Parameter (Name: **convertView**) gelieferte **View**-Hierarchie noch nicht existiert, wird sie durch Inflationieren der dem **EventArrayAdapter** bekannten Layout-Ressource erstellt:

```
if (convertView == null)  
    convertView = LayoutInflater.from(context).inflate(layout, null);
```

Die Layout-Definition für ein Ereignis (mit den Attributen Datum/Zeit, Stimmung, Beschreibung) wird durch die folgende XML-Datei **event.xml** mit einem **ConstraintLayout**-Wurzelement beschrieben:

```

<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="de.zimkand.eventdiary.EventListActivity">

    <TextView
        android:id="@+id/dateTime"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginLeft="16dp"
        android:layout_marginStart="16dp"
        android:layout_marginTop="16dp"
        android:textSize="16sp"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        tools:text="dateTime" />

    <TextView
        android:id="@+id/moodValue"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginEnd="16dp"
        android:layout_marginRight="16dp"
        android:layout_marginTop="16dp"
        android:text="@string/moodvalue"
        android:textSize="16sp"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <TextView
        android:id="@+id/eventDesc"
        android:layout_width="wrap_content"
        android:layout_height="20dp"
        android:layout_marginLeft="16dp"
        android:layout_marginStart="16dp"
        android:layout_marginTop="8dp"
        android:textAppearance="?android:attr/textAppearanceSmall"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/dateTime"
        tools:text="eventDesc" />

</android.support.constraint.ConstraintLayout>

```

Die **TextView**-Elemente im Ereignis-Layout werden mit Texten versorgt, die aus dem aktuellen Ereignis

```
Event event = events.get(position);
```

extrahiert werden:

```

Event event = events.get(position)
TextView tvDateTime = convertView.findViewById(R.id.dateTime);
TextView tvMood = convertView.findViewById(R.id.moodValue);
TextView tvDesc = convertView.findViewById(R.id.eventDesc);
Date date = event.getDate();
tvDateTime.setText(sdf.format(date));
tvMood.setText(context.getString(R.string.moodvalue) + " " +
String.valueOf(event.getMood()));
tvDesc.setText(event.getDesc());

```

Zum Starten der **EventListActivity** über einen expliziten Intent wird in der **onCreate()** - Methode der Startaktivität für den unteren Befehlsschalter eine passende **onClick()** - Methode erstellt:

```
Button btnList = findViewById(R.id.btnList);
btnList.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        Intent intent = new Intent(MainActivity.this, EventListActivity.class);
        startActivity(intent);
    }
});
```

## 13.4 Neue Ereignisse erfassen

Wir erstellen nun eine Aktivität namens `NewEventActivity`, mit der die Benutzer neue Ereignisse erfassen können. Um eine neue, leere Aktivität in ein Projekt einzufügen, wählt man im Projekt-explorer aus dem Kontextmenü zum Paket mit den Klassen des Projekts (`de.zimkand.eventdiary`) den Eintrag

**New > Activity > Empty Activity**

Als **Activity Name** eignet sich z. B. `NewEventActivity`. Für den Eintrag in der Manifestdatei sorgt das Android Studio:

```
<activity
    android:name=".NewEventActivity"
    android:label="@string/title_activity_new_event" >
</activity>
```

Das Layout der Activity ist länglich, aber simpel:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center_horizontal"
    android:orientation="vertical"
    android:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:paddingTop="16dp"
    tools:context="de.zimkand.eventdiary.NewEventActivity">

    <TextView
        android:id="@+id/textQuest"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="16dp"
        android:text="@string/desc_new_event" />
    <EditText
        android:id="@+id/etDesc"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="16dp"
        android:background="@android:drawable/editbox_background"
        android:lines="3"
        android:scrollbars="vertical" />
    <TextView
        android:id="@+id/textMood"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="16dp"
        android:text="@string/desc_mood" />
```

```

<RadioGroup
    android:id="@+id/radioGroup"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="16dp">
    <RadioButton
        android:id="@+id/rbMood1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/mood1" />
    <RadioButton
        android:id="@+id/rbMood2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/mood2" />
    <RadioButton
        android:id="@+id/rbMood3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/mood3" />
    <RadioButton
        android:id="@+id/rbMood4"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/mood4" />
    <RadioButton
        android:id="@+id/rbMood5"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/mood5" />
</RadioGroup>
<Button
    android:id="@+id	btnAddEvent"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="16dp"
    android:text="@string/add_event"
    android:textAllCaps="false" />
</LinearLayout>

```

In der Lebenszyklusmethode **onCreate()** der NewEventActivity

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_new_event);
    final RadioGroup radioGroup = findViewById(R.id.radioGroup);
    eventDAO = MainActivity.getEventDAO();

    Button btnAddEvent = findViewById(R.id.btnAddEvent);
    btnAddEvent.setOnClickListener(new View.OnClickListener() {
        public void onClick(View v) {
            EditText desc = findViewById(R.id.etDesc);
            String s = desc.getText().toString();
            int rbId = radioGroup.getCheckedRadioButtonId();
            View rb = radioGroup.findViewById(rbId);
            int mood = radioGroup.indexOfChild(rb) + 1;
            if (s.length() > 0 && mood >= 1 && mood <= 5) {
                eventDAO.insert(new Event(new Date(), s, mood));
                radioGroup.clearCheck();
                desc.setText("");
            } else
                Toast.makeText(NewEventActivity.this, R.string.incomplete,
                    Toast.LENGTH_SHORT).show();
        }
    });
}

```

besorgen wir uns eine Referenz auf das DAO:

```
eventDAO = MainActivity.getEventDAO();
```

In der **onClick()** - Methode zum Befehlsschalter **btnAddEvent** beauftragen wir das DAO, ein neu erstelltes Event-Objekt in die Datenbanktabelle einzufügen:

```
eventDAO.insert(new Event(new Date(), s, mood));
```

Zugegebenermaßen ist die letzte Anweisung mit dem sehr kurzen Auftritt eines Event-Objekts nicht dazu geeignet, die Vorteile einer ORM-Lösung zu illustrieren.

In der **onClick()** - Methode wird folgendermaßen der per Optionsschalter geäußerte Stimmungswert ermittelt:

- Über die **RadioGroup** - Methode **getCheckedRadioButtonId()** wird die Kennung des eingerasteten **RadioButton**-Schalters ermittelt.
- Daraus lässt sich mit Hilfe der Methode **findViewById()** eine Referenz auf das Objekt gewinnen.
- Schließlich erfährt man über die Methode **indexOfChild()**, welche laufende Nummer der markierte Schalter innerhalb der Gruppe besitzt, und kennt somit den Stimmungswert.

Zum Starten der **NewEventActivity** über einen expliziten Intent wird in der **onCreate()** - Methode der Startaktivität für den oberen Befehlsschalter eine passende **onClick()** - Methode erstellt:

```
Button btnEvents = findViewById(R.id.btnEvents);
btnEvents.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        Intent intent = new Intent(MainActivity.this, NewEventActivity.class);
        startActivity(intent);
    }
});
```

Das vollständige AS-Projekt auf dem aktuellen Stand befindet sich im Ordner

...\\BspUeb\\SQLite\\EventDiary.1

### 13.5 Tabellenzeilen aktualisieren

Wir erweitern das seit Abschnitt 13.2 erstellte Ereignistagebuch um die Möglichkeit, erfasste Ereignisse nachträglich zu verändern. Dazu erhält das im **EventListActivity**-Objekt enthaltene **ListView**-Steuerelement eine Methode zur Behandlung von Klicks auf seine Elemente:

```
ListView lv = findViewById(R.id.listView);
lv.setOnItemClickListener(this);
```

Die Aktivität erklärt sich zum **AdapterView.OnItemClickListener**

```
public class EventListActivity extends AppCompatActivity
    implements AdapterView.OnItemClickListener {
    . . .
}
```

und implementiert die erforderliche Behandlungsmethode **onItemClick()**:

```
@Override
public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
    Event event = (Event)adapter.getItem(position);
    editEntry(event, position);
}
```

Die Methode ermittelt über den Parameter **position** und die **ArrayAdapter<E>** - Methode **getItem()** das zu bearbeitende Event-Objekt und übergibt es samt seiner Position im Adapter an die private Methode **editEntry()**, die sich um die weitere Bearbeitung kümmert.

In der Methode `editEntry()` wird eine Aktivität zum Editieren eines Ereignisses über einen expliziten Intent gestartet (vgl. Abschnitt 9.1.5). Über Extrainformationen werden das komplette Event-Objekt in serialisierter Form und seine Nummer im ListView-Adapter übergeben:

```
private void editEntry(Event event, int position) {
    Intent intent = new Intent(this, EditEventActivity.class);
    intent.putExtra("event", event);
    intent.putExtra("position", position);
    startActivityForResult(intent, REQUEST_CODE_EDIT);
}
```

Damit die folgende `putExtra()` - Überladung

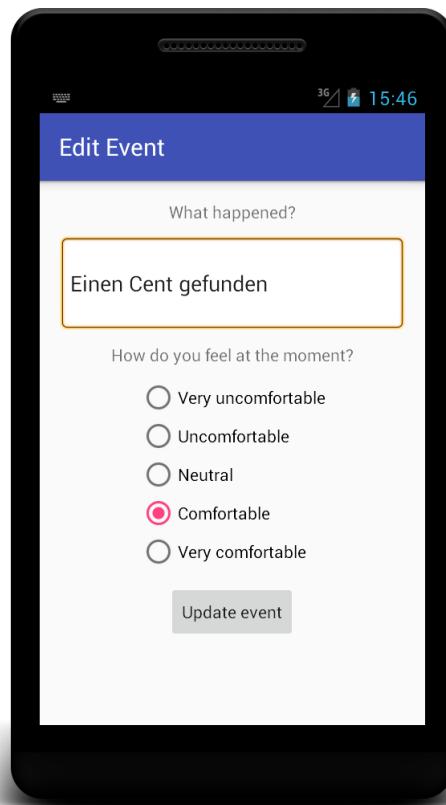
```
public Intent putExtra (String name, Serializable value)
```

genutzt werden kann, muss die Klasse `Event` die Schnittstelle `Serializable` implementieren:

```
public class Event implements Serializable {
    . . .
}
```

Sie gibt durch diese Marker-Schnittstelle zu erkennen, dass sie nichts gegen das Serialisieren ihrer Objekte einzuwenden hat, muss aber keine Schnittstellenmethoden implementieren.

Für die neue Funktionalität brauchen wir keine neue Aktivität, sondern können die vorhandene `NewEventActivity` (siehe Abschnitt 13.2) so erweitern, dass sie auch zum Editieren vorhandener Einträge taugt. Im Layout sind lediglich die Titelzeile und die Beschriftung des Befehlsschalters zu ändern, was per `setTitle()` bzw. `setText()` geschehen kann:



Mit einigen Fallunterscheidungen entsteht eine Aktivität (mit dem neuen Namen `EditEventActivity`), die das Einfügen von neuen Ereignissen genauso beherrscht wie das Ändern von alten Ereignissen:

```
package de.zimkand.eventdiary;

import android.support.v7.app.AppCompatActivity;
...
import java.util.Date;

public class EditEventActivity extends AppCompatActivity {

    private EventDAO eventDAO;
    private boolean insertMode = true;
    private Event event;
    private int position;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_edit_event);
        final RadioGroup radioGroup = findViewById(R.id.radioGroup);
        EditText desc = findViewById(R.id.etDesc);
        Button btnEvent = findViewById(R.id.btnEvent);

        eventDAO = MainActivity.getEventDAO();

        Bundle extras = getIntent().getExtras();
        if (extras != null) {
            insertMode = false;
            event = (Event) extras.getSerializable("event");
            position = extras.getInt("position");
            desc.setText(event.getDesc());
            int mood = event.getMood();
            ((RadioButton) radioGroup.getChildAt(mood - 1)).setChecked(true);
        } else {
            this.setTitle(getString(R.string.title_activity_new_event));
            btnEvent.setText(getString(R.string.add_event));
        }

        btnEvent.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                EditText desc = findViewById(R.id.etDesc);
                String s = desc.getText().toString();
                int rbId = radioGroup.getCheckedRadioButtonId();
                View rb = radioGroup.findViewById(rbId);
                int mood = radioGroup.indexOfChild(rb) + 1;
                if (s.length() > 0 && mood >= 1 && mood <= 5) {
                    if (insertMode) {
                        eventDAO.insert(new Event(new Date(), s, mood));
                        radioGroup.clearCheck();
                        desc.setText("");
                    } else {
                        event.setDesc(s);
                        event.setMood(mood);
                        eventDAO.update(event);
                        Intent intent = new Intent();
                        intent.putExtra("position", position);
                        intent.putExtra("desc", s);
                        intent.putExtra("mood", mood);
                        setResult(RESULT_OK, intent);
                        finish();
                    }
                } else
                    Toast.makeText(EditEventActivity.this, R.string.incomplete,
                        Toast.LENGTH_SHORT).show();
            }
        });
    }
}
```

Ein zu änderndes Ereignis wird aus dem mitgelieferten **Intent**-Objekt entnommen:

```
Bundle extras = getIntent().getExtras();
if (extras != null) {
    insertMode = false;
    event = (Event) extras.getSerializable("event");
    position = extras.getInt("position");
    desc.setText(event.getDesc());
    int mood = event.getMood();
    ((RadioButton) radioGroup.getChildAt(mood - 1)).setChecked(true);
} else {
    this.setTitle(getString(R.string.title_activity_new_event));
    btnEvent.setText(getString(R.string.add_event));
}
```

Die Aktualisierung eines Falles ist per Room-Technik auf beeindruckend einfache Weise zu bewerkstelligen:

```
event.setDesc(s);
event.setMood(mood);
eventDAO.update(event);
```

Die Aktivität **EditEventActivity** liefert den neuen Ereigniszustand an ihren Aufrufer zurück

```
Intent intent = new Intent();
intent.putExtra("position", position);
intent.putExtra("desc", s);
intent.putExtra("mood", mood);
setResult(RESULT_OK, intent);
finish();
```

zur Verarbeitung in der Rückrufmethode **onActivityResult()**:

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent intent) {
    if (requestCode == REQUEST_CODE_EDIT && resultCode == RESULT_OK && intent != null) {
        Bundle extras = intent.getExtras();
        int position = extras.getInt("position");
        Event event = (Event) adapter.getItem(position);
        event.setDesc(extras.getString("desc"));
        event.setMood(extras.getInt("mood"));
        adapter.notifyDataSetChanged();
    }
}
```

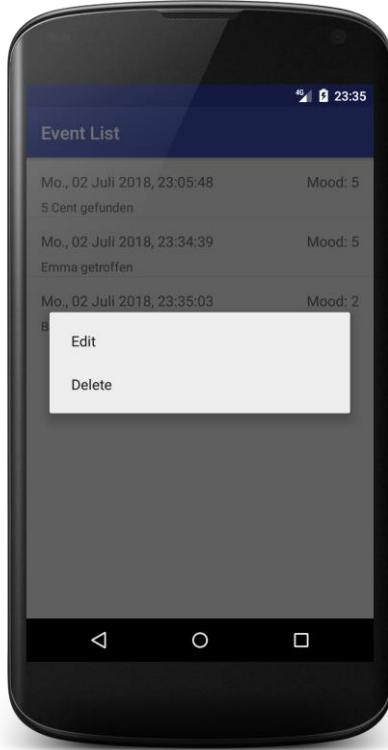
Diese Methode aktualisiert die im Adapter enthaltene Ansicht des editierten Ereignisses und sorgt über die **ArrayAdapter<E>** - Methode **notifyDataSetChanged()** für eine aktualisierte Anzeige im **ListView**-Objekt.

Außer den Arbeiten am Quellcode von **EditEventActivity** und **Event** sind noch einige Erweiterungen in der Ressourcendatei **strings.xml** vorzunehmen, um den Projektzustand im folgenden Ordner zu erreichen:

**...\\BspUeb\\SQLite\\EventDiary.2**

### 13.6 Tabellenzeilen per Kontextmenü löschen

Unser Ereignistagebuch soll dem Benutzer auch die Möglichkeit bieten, Einträge zu löschen. Zur Realisation versorgen in der `EventListActivity` das `ListView`-Steuerelement mit einem flottierenden Kontextmenü, das durch einen Langklick auf das Steuerelement geöffnet werden kann:



Um in einer Aktivität oder in einem Fragment für ein Steuerelement ein flottierendes Kontextmenü zu realisieren, sind folgende Schritte erforderlich (vgl. Abschnitt 8.6.2.1):

- Das Steuerelement ist durch einen Aufruf der **Activity-** bzw. **Fragment-**Methode `registerForContextMenu()` für die Anzeige eines Kontextmenüs registrieren, z. B.:
 

```
ListView lv = findViewById(R.id.ListView);
registerForContextMenu(lv);
```
- Die **Activity-** bzw. **Fragment-**Methode `onCreateContextMenu()` ist zu implementieren. Sie wird vor *jeder* Anzeige des Kontextmenüs aufgerufen und eignet sich dazu, für eine passende Ausstattung des Menüs zu sorgen.
- Die **Activity-** bzw. **Fragment-**Methode `onContextItemSelected()` ist zu implementieren. Sie wird aufgerufen, wenn der Benutzer ein Item aus dem Kontextmenü gewählt hat.

Weil im Beispiel das Kontextmenü stets identisch aufgebaut sein soll, kann sich die Methode `onCreateContextMenu()` darauf beschränken, das Kontextmenü aus einer XML-Definitionsdatei per `MenuInflater` zu expandieren:

```
@Override
public void onCreateContextMenu(ContextMenu contMenu, View v,
                                ContextMenu.ContextMenuItemInfo contextMenuItemInfo) {
    super.onCreateContextMenu(contMenu, v, contextMenuItemInfo);
    getMenuInflater().inflate(R.menu.event_list_context_menu, contMenu);
}
```

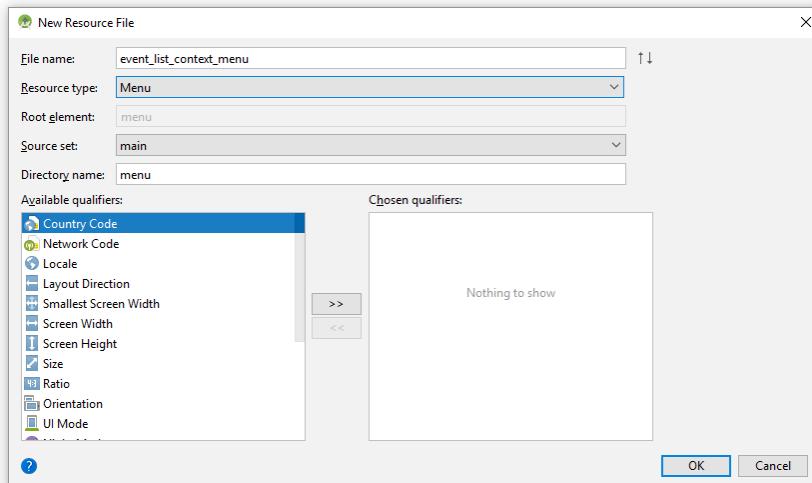
Es folgt die im Projektordnerknoten `app/res/menu` abzulegende Menüdefinitionsdatei `event_list_context_menu.xml`, die zwei Menüitems mit Kennung und Titel enthält:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/evliconit_edit"
        android:title="@string/evliconit_edit" />
    <item android:id="@+id/evliconit_del"
        android:title="@string/evliconit_del" />
</menu>
```

Um diese Datei anzulegen, wählt man aus dem Kontextmenü zum Projektexplorerknoten **app/res** den Befehl

### New > Android Resource File

und ergänzt im folgenden Dialog



den Dateinamen sowie den Ressourcen-Typ.

Wenn der Benutzer das Kontextmenü geöffnet und ein Item gewählt hat, wird die Methode **onContextItemSelected()** aufgerufen:

```
@Override
public boolean onContextItemSelected(MenuItem item) {
    AdapterView.AdapterContextMenuInfo acmi =
        (AdapterView.AdapterContextMenuInfo) item.getMenuInfo();
    Event event = (Event)adapter.getItem(acmi.position);
    switch (item.getItemId()) {
        case R.id.evliconit_edit:
            editEntry(event, acmi.position);
            return true;
        case R.id.evliconit_del:
            eventDAO.delete(event);
            adapter.remove(event);
            adapter.notifyDataSetChanged();
            return true;
        default:
            return super.onContextItemSelected(item);
    }
}
```

Hier sind zu ermitteln:

- Das **Event**-Objekt hinter dem Listenelement, für das ein Kontextmenü angefordert wurde
- Item, das aus dem Kontextmenü gewählt wurde

Während die zweite Frage über den mitgelieferten Parameter vom Typ **MenuItem** leicht zu klären ist, erfordert die Klärung der ersten Frage einigen Aufwand:

- Das Parameterobjekt liefert über die Methode **getMenuInfo()** ein Objekt aus der Klasse **AdapterView.AdapterContextMenuInfo**, wobei eine explizite Umwandlung in diesen Typ erforderlich ist. Dieses Objekt bewahrt in der öffentlichen Instanzvariablen **id** den Indexwert der vom Kontextmenü betroffenen Listenelements auf.
- Über die **ArrayAdapter<E>** - Methode **getItem()** erhalten wir Beispiel das **Event**-Objekt, das hinter dem Listenelement mit einem bestimmten Indexwert steckt.

Um ein Ereignis zu löschen, verwenden die **EventDAO**-Methode **delete()**:

```
eventDAO.delete(event);
```

Anschließend sorgen wir folgendermaßen

```
adapter.remove(event);
adapter.notifyDataSetChanged();
```

für eine sofortige Aktualisierung der **ListView**-Anzeige:

- Das gelöschte Ereignis wird aus dem **ArrayAdapter** entfernt.
- Der **ArrayAdapter** wird über die Methode **notifyDataSetChanged()** aufgefordert, das **ListView**-Objekt zu informieren.

Um die Änderung der Ereignisbeschreibung zu ermöglichen, übergeben wir das Ereignis samt seiner Position im Adapter an die private Methode **editEntry()**, die sich um die weitere Bearbeitung kümmert (siehe Abschnitt 13.5).

Das vollständige AS-Projekt auf dem aktuellen Stand befindet sich im Ordner

```
...\\BspUeb\\SQLite\\EventDiary.3
```

## 13.7 Weitere wichtige Room-Optionen

Einige sehr relevante Room-Optionen können aus Zeitgründen in diesem Manuskript nicht behandelt werden:

- **Migration**

Damit die Benutzer bei einer Änderung des Datenbankschemas keine Datenverluste erleiden, muss eine Migrationsklasse definiert werden.<sup>1</sup>

- **LiveData<T>**

In einer Room-Abfragemethode (definiert in der DAO-Schnittstelle) kann ein Rückfragetyp unter Verwendung der Klasse **LiveData<T>** aus dem Paket **android.arch.lifecycle** konstruiert werden, um eine automatische Anpassung der Bedienoberfläche an geänderte Daten zu erreichen.<sup>2</sup>

- **RxJava2**

In einer Room-Abfragemethode (definiert in der DAO-Schnittstelle) kann ein Rückfragetyp unter Verwendung von Klassen aus der RxJava2 - Bibliothek verwendet werden (z. B. **Flowable<T>**). Im Ergebnis ist die Abfrage nicht nur beobachtbar (mit einer automatischen Anpassung der Bedienoberfläche an geänderte Daten, wie bei **LiveData<T>**), sondern wird auch asynchron ausgeführt (ohne Belastung des UI-Threads).<sup>3</sup>

<sup>1</sup> <https://developer.android.com/reference/android/arch/persistence/room/migration/Migration>

<sup>2</sup> <https://developer.android.com/topic/libraries/architecture/livedata>

<sup>3</sup> <https://medium.com/google-developers/room-rxjava-acb0cd4f3757>

### 13.8 Direktkontakt mit SQLite per ADB-Konsole

Leute mit großer Neugier oder speziellen Anwendungsproblemen können konsolenorientiert mit SQLite interagieren. Die gleich vorgestellte ADB-Konsole (*Android Debug Bridge*) lässt sich nicht nur bei Datenbankanwendungen dazu verwenden, um hinter die Android-Kulissen zu blicken.

#### 13.8.1 SDK-Kommando adb shell

Wir verwenden die **Android Debug Bridge** (ADB), um eine Konsolenverbindung zu einem emulierten Android-Gerät herzustellen (vgl. Abschnitt 3.4):

- Starten Sie ein virtuelles Android-Gerät und eine Windows-Eingabeaufforderung.
- Wechseln Sie in der Eingabeaufforderung zum Unterordner **platform-tools** der SDK-Installation.
- Schicken Sie das Kommando **adb shell** ab, um eine Konsolenverbindung zum emulierten Android-Gerät herzustellen:

```
C:\ Eingabeaufforderung - adb shell
C:\Users\baltes\AppData\Local\Android\Sdk\platform-tools>adb shell
root@generic_x86_64:/ # _
```

Sind mehrere emulierte Geräte aktiv, ist zwischen dem Kommandonamen **adb** und der Betriebsart **shell** noch die Option **-s** mit dem Gerätenamen einzufügen. An den Namensanfang **emulator** sind ein Bindestrich und die vom emulierten Gerät verwendete Portnummer anzuhängen, z. B.:

`adb -s emulator-5554 shell`

Eine Liste der verbundenen Geräte liefert der Befehl:

`adb devices`

Ist nur *ein* emuliertes Gerät aktiv, erhält man die Ausgabe:

```
List of devices attached
emulator-5554    device
```

Wechseln Sie in der Linux-Konsole zum Ordner **/data/local/tmp** auf dem emulierten Gerät:

```
C:\ Eingabeaufforderung - adb shell
C:\Users\baltes\AppData\Local\Android\Sdk\platform-tools>adb shell
root@generic_x86_64:/ # cd /data/local/tmp
root@generic_x86_64:/data/local/tmp # _
```

#### 13.8.2 Interaktion mit SQLite

Starten Sie eine Interaktion mit **SQLite** zur Bearbeitung der Datenbank **hell.db** über das Kommando **sqlite3** mit dem Datenbanknamen als Argument:

```
C:\ Eingabeaufforderung - adb shell
root@generic_x86_64:/ # cd /data
root@generic_x86_64:/data # sqlite3 hell.db
SQLite version 3.8.6.1 2015-05-21 17:24:32
Enter ".help" for usage hints.
sqlite> _
```

Wenn die Datenbankdatei im aktuellen Ordner noch nicht existiert, wird sie automatisch angelegt.

Erstellen Sie mit dem Kommando **CREATE TABLE** eine neue Tabelle mit dem Namen **devils** und den folgenden Feldern:

- **\_id**

Weil dieses Feld als ganzzahliger Primärschlüssel fungieren soll, erhält es die Attribute  
**INTEGER PRIMARY KEY AUTOINCREMENT**

Mit dem optionalen Schlüsselwort **AUTOINCREMENT** wird die Wiederverwendung von frei gewordenen Zeilennummern verhindert.

- **name**

Dieses Feld mit Datentyp **TEXT** soll die Namen der Teufel aufnehmen.

- **age**

In dieses Feld mit Datentyp **INTEGER** werden die Altersangaben zu den Teufeln geschrieben.

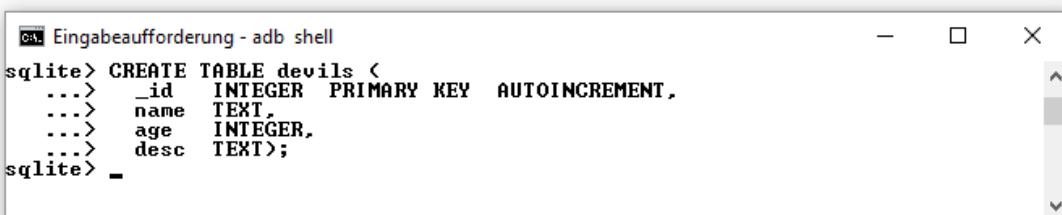
- **desc**

Dieses Feld mit Datentyp **TEXT** soll Beschreibungen zu den Teufeln aufnehmen.

Wird das komplette Kommando

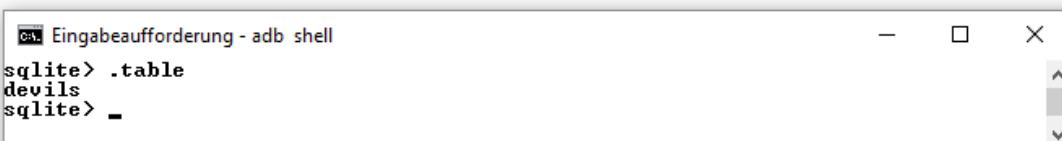
```
CREATE TABLE devils (
    _id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT,
    age INTEGER,
    desc TEXT);
```

zeilenweise eingegeben, fordert SQLite so lange eine Fortsetzung, bis es ein Semikolon als Kommandoterminator gesehen hat:



```
Eingabeaufforderung - adb shell
sqlite> CREATE TABLE devils <
...>     _id INTEGER PRIMARY KEY AUTOINCREMENT,
...>     name TEXT,
...>     age INTEGER,
...>     desc TEXT>;
sqlite> -
```

Über das SQLite-Kommando **.table** (kein SQL-Kommando, daher der einleitende Punkt und *kein* terminierendes Semikolon) erhält man eine Liste mit den Tabellen der Datenbank:



```
Eingabeaufforderung - adb shell
sqlite> .table
devils
sqlite> -
```

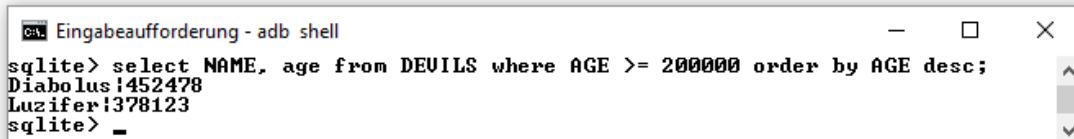
Nun fügen wir über **INSERT**-Kommandos einige Teufel in die Datenbank ein:

```
INSERT INTO devils VALUES(1, 'Luzifer', 378123, 'Manchmal launig');
INSERT INTO devils (name, age) VALUES('Mephisto', 174332);
INSERT INTO devils (name, age, desc) VALUES('Diabolus', 452478, 'Verlogen');
```

Mit dem folgenden **SELECT**-Kommando

```
SELECT name, age FROM devils WHERE age >= 200000 ORDER BY age DESC;
```

unter Verwendung der Klausel **ORDER BY** mit der Spezifikation **DESC** werden die Namen und Altersangaben der Teufel mit einem Alter von minimal 200.000 Jahren nach Alter absteigend sortiert ausgegeben:



```
sqlite> select NAME, age from DEVILS where AGE >= 200000 order by AGE desc;
Diabolus:452478
Luzifer:378123
sqlite> -
```

Wie man sieht, ist in SQL-Kommandos die Groß-/Kleinschreibung irrelevant.

Mit dem Kommando **UPDATE** lassen sich Werte ändern, z. B.

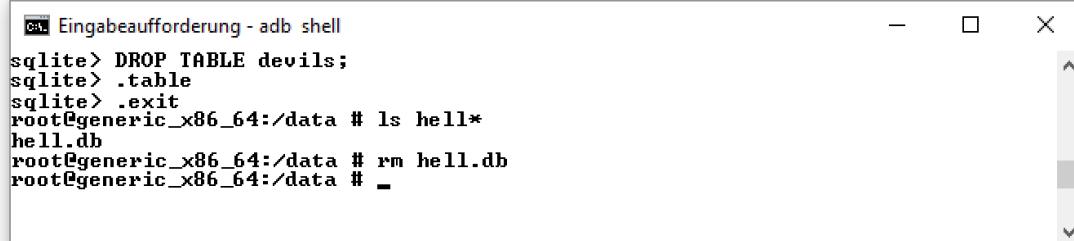
```
UPDATE devils SET desc='Wie die Faust aufs Auge' WHERE name='Mephisto';
```

Mit dem Kommando **DELETE** lassen sich Zeilen aus der Tabelle löschen, z. B.:



```
sqlite> DELETE FROM devils WHERE name='Mephisto';
sqlite> SELECT * FROM devils;
1:Luzifer:378123:Manchmal launig
3:Diabolus:452478:Verlogen
sqlite> -
```

Per **DROP TABLE** kann eine Tabelle gelöscht werden, z. B.:



```
sqlite> DROP TABLE devils;
sqlite> .table
sqlite> .exit
root@generic_x86_64:/data # ls hell*
hell.db
root@generic_x86_64:/data # rm hell.db
root@generic_x86_64:/data # -
```

Mit dem SQLite-Kommando **.exit** (kein SQL-Kommando, daher der einleitende Punkt und *kein* terminierendes Semikolon) beendet man die Interaktion mit der Datenbank. Im Beispiel wird abschließend mit dem Linux-Kommando **rm** die Datenbankdatei gelöscht. Das kann bei der realen App-Entwicklung durchaus sinnvoll sein, wenn eine Datenbankdatei mit störenden Eigenschaften den Anwendungsstart verhindert.

Weitere **SELECT**-Optionen (z. B. **JOIN**, **GROUP**) werden von Becker & Pant (2015, S. 259f) beschrieben.

## 14 Content Provider nutzen und anbieten

Um fremden Anwendungen den Zugriff auf eigene Daten zu ermöglichen, erstellt man eine *Content Provider* - Anwendungskomponente. Diese stellt den Klienten meist eine oder mehrere Tabellen mit strukturierten Daten zur Verfügung. Ob diese Daten in einer Datenbank abgelegt sind, oder aus einer alternativen Quelle (z.B. aus einem Array) stammen, ist ein für Klienten irrelevantes Implementierungsdetail. Neben strukturierten Daten (Abfrageergebnissen) können auch Dateien veröffentlicht werden (z.B. eine Fotosammlung).

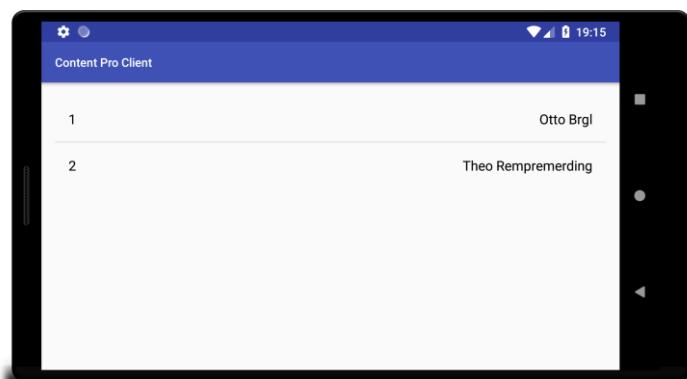
Selbstverständlich können wir in eigenen Anwendungen die Content Provider fremder Anwendungen nutzen, wobei das Betriebssystem selbst einige Datenbestände anbietet (z.B. Kontakte, Kalendereinträge, SMS-Nachrichten).<sup>1</sup>

Über Content Provider und ihre Klienten kommt in Android ein Datenaustausch zwischen Anwendungen unter der Kontrolle eines Berechtigungssystems zu Stande, wobei die erforderliche Inter-Prozess - Kommunikation von Android erledigt wird.

### 14.1 Content Provider fremder Anwendungen nutzen

Über Content Provider in fremden Anwendungen stehen viele Datenbestände bereit, die man in eigenen Anwendungen nutzen kann (z.B. die von Android verwalteten Kontakte und Termine). In diesem Abschnitt wagen wir uns an den Kontakte-Provider, der von großem Interesse ist, aber auch eine relativ komplexe Struktur besitzt. Das Content Provider - Standardbeispiel in vielen Lehrtexten und auch in der (im Juli 2018) aktuellen Google-Dokumentation zu Android ist das Benutzerwörterbuch. Es ist zum Einstieg in das Thema sehr gut geeignet, hat aber den entscheidenden Nachteil, dass es ab API 23 für Apps nicht mehr ansprechbar ist.<sup>2</sup>

Auf einem Smartphone im Querformat sieht das Beispielprogramm zur Nutzung des Kontakte-Providers so aus:



Der Einfachheit werden im aktuellen Abschnitt die Provider- und damit letztlich die Datenbankzugriffe im UI-Thread (also synchron) ausgeführt, was generell wegen des potentiell hohen Zeitaufwands und der damit drohenden ANR-Fehler (*Application Not Responding*) zu vermeiden ist. In Abschnitt 14.3 wird demonstriert, wie man asynchrone Provider-Zugriffe durchführt, die in einem Hintergrund-Thread ablaufen und den UI-Thread *nicht* belasten.

Für Zugriffe auf einen Content Provider ist ein Objekt der Klasse **ContentResolver** zu verwenden. Es ermöglicht:

<sup>1</sup> Siehe Liste in: <https://developer.android.com/reference/android/provider/package-summary>

<sup>2</sup> <https://developer.android.com/reference/android/provider/UserDictionary>

- Abfragen (siehe Abschnitt 14.1.3)  
Im typischen Fall erhält man eine Tabelle mit Ergebniszeilen, die durch ein Objekt vom Typ **Cursor** (siehe Abschnitt 14.1.2) repräsentiert wird.
- Modifikationen (siehe Abschnitt 14.1.5)  
In eine Tabelle eines Content Providers können wie bei einer Datenbanktabelle neue Zeilen eingetragen sowie vorhandene Zeilen geändert oder gelöscht werden.

Beim Zugriff auf einen Content Provider sind Zugriffsrechte relevant:

- In der Anwendungs-Manifestdatei können zu einer Provider-Komponente Berechtigungen für das Lesen und Schreiben seiner Daten definiert werden, wenn die per Voreinstellung bestehende freie Verfügbarkeit nicht sinnvoll ist (siehe Abschnitt 14.2.4).
- Eine zugriffswillige fremde App muss diese Rechte in ihrer eigenen Manifestdatei beantragen, und der Benutzer muss diesem Antrag zustimmen, bevor die App installiert werden kann (siehe Abschnitt 14.1.7).

### 14.1.1 Content-URI

Zur Identifikation der Daten, die von einem Content Provider bezogen oder dort verändert werden sollen, dient ein **Uri**-Objekt mit einem **content**-Schema (als Protokoll-Angabe). Um z.B. beim Kontakte-Provider eines Android-Geräts den dritten Eintrag anzufordern, ist ein auf der folgenden Zeichenfolge basierendes **Uri**-Objekt zu verwenden:

Aufbau:	Schema	Provider (Autorität)	Pfad	ID
<b>Beispiel:</b>	<code>content://</code>	<code>com.android.contacts/</code>	<code>contacts/</code>	<code>3</code>

Am Providerbestanteil erkennt Android, welcher Systembestandteil bzw. welche App für ein **Uri**-Objekt mit **content**-Schema zuständig ist. Bei der Installation einer App mit Content Provider wird diese Zeichenfolge in der Manifestdatei hinterlegt (siehe Abschnitt 14.2.3).

Man kann das benötigte **Uri**-Objekt über die statische **Uri**-Methode **parse()** aus einer Zeichenfolge erstellen, z.B.:

```
Uri uri = Uri.parse("content://com.android.contacts/contacts");
```

Sinnvoller ist die Verwendung einer vom Entwickler des Content Providers veröffentlichten Zeichenfolgevariablen, z.B.:

```
Uri uri = ContactsContract.Contacts.CONTENT_URI;
```

Wenn bei einem vorhandenen **Uri**-Objekt hinter die Pfadangabe eine Kennung gesetzt werden soll, um einen individuellen Eintrag anzusprechen, eignet sich die statische Methode **withAppendedId()** der Klasse **ContentUris** (im Paket **android.content**), z.B.:

```
Uri Uri = ContentUris.withAppendedId(ContactsContract.Contacts.CONTENT_URI, 4);
```

### 14.1.2 Cursor

Bei einer Abfrage an einen Content Provider bzw. an das vermittelnde **ContentResolver**-Objekt (siehe Abschnitt 14.1.3) liefert die zu verwendende **query()** - Methode ein Objekt aus einer das Interface **Cursor** implementierenden Klasse. Wer bei Datenbankzugriffen unter Android statt der von Google stark präferierten und in Kapitel 13 beschrieben Room Persistence Library noch die traditionellen Klassen **SQLiteDatabase** und **SQLiteOpenHelper** benutzt, ist mit dem Datentyp **Cursor** wohlvertraut, weil die **SQLiteDatabase**-Methode **query()** ein Objekt als Rückgabe liefert, das die Verhaltenskompetenzen der Schnittstelle **Cursor** besitzt. Im Kurs begegnet uns die Schnittstelle **Cursor** nun erstmals als Rückgabetyp der **ContentResolver**-Methode **query()**.

Ein Objekt aus einer das Interface **Cursor** implementierenden Klasse (ab jetzt auch kurz als **Cursor**-Objekt bezeichnet) repräsentiert die **Tabelle mit dem Abfrageergebnis** und erlaubt über einen Zeiger einen Zugriff auf die Zeilen. Anschließend werden einige **Cursor**-Methoden vorgestellt:

- **void moveToFirst(), void moveToNext(), void moveToLast()**  
Der Zeiger wird zur ersten, nächsten bzw. letzten Zeile bewegt. Weil der Zeiger initial vor der ersten Zeile steht, muss der Zeiger mit **moveToFirst()** bewegt werden, um die erste Ergebnissezeile ansprechen zu können.
- **int getCount()**  
Man erhält die Anzahl der Zeilen im Abfrageergebnis. Wenn eine Abfrage keine Treffer erbracht hat, liefert **getCount()** die Rückgabe 0.
- **int getColumnIndex(String columnName)**  
Diese Methode liefert zu einem Spaltennamen seinen Index oder -1, wenn keine Spalte mit diesem Namen vorhanden ist.
- **int getType(int index)**  
Für die Spalte mit dem angegebenen Index wird ihr Datentyp gemeldet, wobei die folgenden **Cursor**-Konstanten als Rückgabewerte auftreten:
  - **FIELD\_TYPE\_NULL** (int-Wert: 0)
  - **FIELD\_TYPE\_INTEGER** (int-Wert: 1)
  - **FIELD\_TYPE\_FLOAT** (int-Wert: 2)
  - **FIELD\_TYPE\_STRING** (int-Wert: 3)
  - **FIELD\_TYPE\_BLOB** (int-Wert: 4)
 Ein BLOB (*Binary Large Object*) ist ein Array mit Elementen vom Typ **byte**.
- **double getDouble(int index), int getInt(int index), String getString(int index), etc.**  
Im günstigen Fall erhält man den Inhalt der Spalte mit dem angegebenen Index als Wert vom gewünschten Typ. Das Verhalten der Methoden bei ungültigem Indexwert, falschem Datentyp oder Zelleninhalt außerhalb des Wertebereichs ist implementationsabhängig.

### 14.1.3 Abfragen

Beim lesenden Zugriff auf einen Content Provider ist eine **query()** - Anfrage an das in jeder **Context**-Ableitung (also insbesondere in jeder Aktivität) über die Methode **getContentResolver()** erreichbare Objekt der Klasse **ContentResolver** zu richten, z.B.:

```
Cursor cursor = getContentResolver().query(uri, projection, null, null, null);
```

Man erhält ein **Cursor**-Objekt zurück, das eine Ergebnistabelle repräsentiert (vgl. Abschnitt 14.1.2). Neben dem **Uri**-Objekt im ersten Parameter (siehe Abschnitt 14.1.1) sind bei Verwendung der folgenden **query()** - Überladung

```
public Cursor query (Uri uri, String[] projection, String selection,
                     String[] selectionArgs, String sortOrder)
```

etliche weitere Parameter beteiligt:

- **String[] projection**  
Im zweiten Parameter kann eine Liste mit den tatsächlich benötigten Tabellenspalten angegeben werden. Im Beispiel sollen zu einem Kontakt seine Kennung und sein Anzeigename im Abfrageergebnis enthalten sein:

```
String[] projection = {ContactsContract.Contacts._ID,
                      ContactsContract.Contacts.DISPLAY_NAME};
```

Die Kennung bzw. der Primärschlüssel (mit der Spaltenbezeichnung **\_id**) muss in die Abfrage einbezogen werden, wenn die Ergebnistabelle mit Hilfe der Klasse **CursorAdapter**

genutzt und z.B. an ein **ListView**-Steuerelement gebunden werden soll.<sup>1</sup> Setzt man den Parameter *projection* auf **null**, dann werden *alle* Spalten in der Tabelle des Content Providers abgerufen. Die Spaltennamen sind der Provider-Dokumentation zu entnehmen und vorzugsweise über Konstanten aus der Kontraktklasse zum Provider (vgl. Abschnitt 14.1.9, im Beispiel: **ContactsContract.Contacts**) anzusprechen.

- **String selection**

Über eine Bedingung kann die Menge der abgerufenen Zeilen eingeschränkt werden. Es ist die Syntax der Klausel **WHERE** aus dem SQL-Kommando **SELECT** zu verwenden, wobei das Schlüsselwort **WHERE** selbst aber wegzulassen ist. Im folgenden Beispiel wird die Abfrage auf Kontakte mit einem bestimmten Anzeigennamen eingeschränkt:

```
String whereClause = ContactsContract.RawContacts.DISPLAY_NAME_PRIMARY + " = ?";
String[] whereArgs = new String[] {"Otto Brgl"};
Cursor cursor = contentResolver.query(
    ContactsContract.RawContacts.CONTENT_URI,
    projection, whereClause, whereArgs, null);
```

Was an Stelle der **?**-Platzhalter einzusetzen ist, wird im nächsten Parameter festgelegt. Man spricht hier von einer *parametrisierten Abfrage*. Setzt man den *selection*-Parameter auf **null**, dann werden *alle* Zeilen in der Tabelle des Content Providers abgerufen.

- **String[] selectionArgs**

Es ist ein **String**-Array mit einer zur Liste der Fragezeichen im *selection*-Parameter korrespondierenden Liste von Zeichenfolgen anzugeben.

- **String sortOrder**

Mit der Syntax der Klausel **ORDER BY** aus dem SQL-Kommando **SELECT** kann die gewünschte Sortierung der Ergebnistabelle angefordert werden, wobei die Schlüsselwörter **ORDER BY** wegzulassen sind, z.B.:

```
ContactsContract.Contacts.DISPLAY_NAME + " ASC"
```

Ein erfolgreicher **query()** - Aufruf liefert ein **Cursor**-Objekt, das *vor* der ersten Zeile der Ergebnistabelle positioniert ist. Als problematische **query()** - Ergebnisse sind möglich:

- Leere Ergebnismenge

In Abhängigkeit vom Füllzustand der Quelltabelle und von der verwendeten **WHERE**-Klausel kann eine leere Ergebnismenge resultieren. Über die Anzahl der Zeilen im **Cursor**-Objekt informiert dessen Methode **getCount()**.

- **null**

Die Rückgabe **null** tritt z.B. in folgenden Fällen auf:

- Das **Uri**-Objekt ist fehlerhaft, indem es z.B. eine fehlerhafte Schemaangabe oder keine Autorität enthält. Solche Probleme sind ausgeschlossen, wenn man das **Uri**-Objekt über eine vom Entwickler des Content Providers veröffentlichten Zeichenfolgevariable anspricht.
- Der Content Provider ist nicht installiert.
- Im Prozess des Content Providers ist ein Ausnahmefehler aufgetreten.

- Ausnahmefehler

Enthält z.B. das **Uri**-Objekt einen fehlerhaften Tabellennamen, dann resultiert eine **IllegalArgumentException**. Besteht keine Berechtigung zum Zugriff, dann wird eine **SecurityException** geworfen.

Ein praxistaugliches Programm muss auf alle problematischen **query()** - Ergebnisse sinnvoll reagieren.

---

<sup>1</sup> <https://developer.android.com/reference/android/widget/CursorAdapter>

Das von einem erfolgreichen **query()** - Aufruf gelieferte **Cursor**-Objekt wird häufig an ein **ListView**-Steuerelement gebunden, um die Ergebnistabelle zu präsentieren. Als Vermittler eignet sich ein Objekt der Klasse **SimpleCursorAdapter**, z.B.:

```
int[] tvIds = {R.id.id, R.id.name};
SimpleCursorAdapter adapter = new SimpleCursorAdapter(getApplicationContext(),
    R.layout.list_view_element, cursor, projection, tvIds, 0);
ListView lv = findViewById(R.id.listView);
lv.setAdapter(adapter);
```

Im Beispiel wird die folgende **SimpleCursorAdapter** - Konstruktorüberladung verwendet:

```
public SimpleCursorAdapter(Context context, int layout, Cursor cursor,
    String[] projection, int[] tvIds, int flags)
```

Als zweiten Parameter ist die Ressourcen-Kennung zur Layout-Beschreibung für ein einzelnes Listenelement anzugeben. Im Beispiel kommt das folgende **ConstraintLayout** mit zwei **TextView**-Steuerelementen zum Einsatz, welche die Elementkennungen **id**, und **name** besitzen:

```
<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin"
    tools:context="de.zimkand.contentproclient.MainActivity"
    android:id="@+id/constraintLayout">
    <TextView
        android:id="@+id/id"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textColor="@android:color/black"
        android:textSize="16sp"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
    <TextView
        android:id="@+id/word"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="8dp"
        android:textColor="@android:color/black"
        android:textSize="16sp"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/id" />
</android.support.constraint.ConstraintLayout>
```

Im **tvIds**-Parameter des **SimpleCursorAdapter** - Konstruktors ist eine zur Projektion analog aufgebaute Liste von **TextView**-Steuerelementen anzugeben.

Anschließend ist der Quellcode für eine App zu sehen, die eine Liste mit den Kontakten des Benutzers präsentiert. Der unerwartet große Umfang des Quellcodes resultiert vor allem aus dem seit Android 6 gestiegenen Aufwand bei der Verwaltung von Zugriffsrechten (vgl. Abschnitt 14.1.7):

```

package de.zimkand.contentproclient;

import android.content.ContentResolver;
...
import android.widget.SimpleCursorAdapter;

public class MainActivity extends AppCompatActivity
    implements ActivityCompat.OnRequestPermissionsResultCallback {

    private final String TAG = "ContentProClient";
    private final int PERMISSIONS_REQUEST_READ_CONTACTS = 1;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // Nötigenfalls die READ-Berechtigung anfordern
        if (ContextCompat.checkSelfPermission(this,
            Manifest.permission.READ_CONTACTS) != PackageManager.PERMISSION_GRANTED)
            ActivityCompat.requestPermissions(this, new String[]{Manifest.permission.READ_CONTACTS},
                PERMISSIONS_REQUEST_READ_CONTACTS);
        else
            showContacts();
    }

    private void showContacts() {
        String[] projection = {ContactsContract.Contacts._ID,
            ContactsContract.Contacts.DISPLAY_NAME};
        Cursor cursor = getContentResolver().query(ContactsContract.Contacts.CONTENT_URI,
            projection, null, null, null);
        if (cursor != null){
            if (cursor.getCount() > 0 ) {
                int[] tvIds = {R.id.id, R.id.name};
                SimpleCursorAdapter adapter = new SimpleCursorAdapter(
                    getApplicationContext(), R.layout.list_view_element,
                    cursor, projection, tvIds, 0);
                ListView lv = findViewById(R.id.listView);
                lv.setAdapter(adapter);
            }
        }
    }

    @Override
    public void onRequestPermissionsResult(int requestCode,
        String permissions[], int[] grantResults) {
        switch (requestCode) {
            case PERMISSIONS_REQUEST_READ_CONTACTS: {
                if (grantResults.length > 0
                    && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
                    showContacts();
                } else
                    Toast.makeText(this, "Read-Permission not granted", Toast.LENGTH_SHORT).show();
                return;
            }
        }
    }
}

```

Statt die **Cursor**-Daten per Adapter an ein **ListView**-Steuerelement zu binden, können sie auch extrahiert und anderweitig genutzt werden. Um die Datentypen in einer Provider-Tabelle zu ermitteln, kann man die Dokumentation konsultieren (siehe Abschnitt 14.1.9) oder die **Cursor**-Methode **getType()** mit einer Spaltennummer als Aktualparameter aufrufen.

Zu jedem unterstützten Content-URI nennt ein Provider einen MIME-Typ, den man durch die **ContentResolver**-Methode **getType()** (mit einem Parameter vom Typ **Uri** und einer Rückgabe vom Typ **String**) erfragen kann. Beim Kontakte-Provider führt die Anfrage

```
getContentResolver().getType(ContactsContract.Contacts.CONTENT_URI)
```

zur Auskunft:

```
vnd.android.cursor.dir/contact
```

#### 14.1.4 Struktur des Kontakte-Providers in Android

Der Kontakte-Provider in Android muss zu einer Person diverse Informationsquellen (Konten) mit jeweils zahlreichen Einzelinformationen (z.B. Namensbestandteile, Telefonnummern, Mail-Adressen) zusammenführen und hat daher eine komplexe Struktur.<sup>1</sup>

Es sind drei Tabellen zu unterscheiden:

- **contacts**

In dieser Tabelle steht eine Zeile für einen Kontakt. Sie wird per Aggregation aus der Tabelle **raw\_contacts** gewonnen, die für einen Kontakt *mehrere* Zeilen enthalten kann, die aus verschiedenen Konten des Smartphone-Besitzers (z.B. Google-Mail, Twitter) stammen. Wir richten zwar *Anfragen* an die Tabelle **contacts** (siehe Abschnitt 14.1.3), nehmen hier aber *keine direkten* Veränderungen vor.

- **raw\_contacts**

Hier steht eine Zeile für die Informationen zu einem Kontakt, die aus einem bestimmten Konto des Smartphone-Besitzers (z.B. Google-Mail, Twitter) stammen. Wichtige Spalten in der Tabelle **raw\_contacts** sind:

<b>_ID</b>	Kennung des Rohkontakts
<b>CONTACT_ID</b>	Kennung des Kontakts, zu dem der Rohkontakt gehört
<b>ACCOUNT_TYPE</b>	Dienstanbieter des Kontos, aus dem die Daten stammen (z.B. com.google, com.twitter)
<b>ACCOUNT_NAME</b>	Name des Kontos (z.B. brgl@google.com, brgl_von_hameln) Es ist möglich, dass ein Smartphone-Besitzer <i>mehrere</i> Konten vom selben Typ hat (z.B. mehrere Google-Konten)
<b>DELETED</b>	Der Kontakte-Provider synchronisiert seine Daten in der Regel mit Kontoanbietern im Internet. Ein als gelöscht markierter Rohkontakt verschwindet, sobald der zugehörige Synchronisations-Adapter den Rohkontakt vom Server gelöscht hat.

- **data**

Hier befinden sich die Daten zu einem Rohkontakt, wobei in einer Zeile Daten eines bestimmten Typs stehen (z.B. Namensbestandteile, Mail-Adressen, Foto). Folglich ...

- sind die Zeilen unterschiedlich aufgebaut,
- und zu einem Rohkontakt gehören mehrere Zeilen.

Wichtige Spalten in der Tabelle **data** sind:

<b>RAW_CONTACT_ID</b>	Kennung des Rohkontakts, zu dem die Daten gehören
<b>MIMETYPE</b>	Typ der in einer Zeile enthaltenen Daten
<b>DATA1 - DATA15</b>	Diese 15 generischen Spalten enthalten je nach MIMETYPE unterschiedliche Daten. Außerdem sind in Abhängigkeit vom MIMETYPE Aliasnamen für die Spalten vorhanden.

<sup>1</sup> <https://developer.android.com/guide/topics/providers/contacts-provider>

### 14.1.5 Modifikationen

Für modifizierende Zugriffe auf einen **ContentProvider** stellt ein **ContentResolver** die Methoden **insert()**, **update()** und **delete()** zur Verfügung. Diese Methoden stehen auch beim Kontakte-Provider zur Verfügung, und wir werden sie gleich auf die Kontakte-Tabellen **raw\_contacts** und **data** (siehe Abschnitt 14.1.4) anwenden. Für die Praxis empfiehlt Google die performantere sogenannte *Stapel-Modifikation* (siehe Abschnitt 14.1.8), wobei aber hinsichtlich der Grundlogik beim Providerzugriff keine Unterschiede bestehen.<sup>1</sup>

#### insert()

Um per App einen neuen Kontakt aufzunehmen, gehen wir in zwei Schritten vor:

- Zunächst wird ein neuer Rohkontakt aufgenommen. Wie Sie aus Abschnitt 14.1.4 wissen, ist es nicht möglich, auf direktem Weg eine Zeile in die **contacts**-Tabelle aufzunehmen.
- Dann werden wir einen Anzeigennamen für den neuen Rohkontakt in die **data**-Tabelle eintragen.

Weil der neue Kontakt zu keinem Konto gehört, bleiben die **raw\_contacts**-Tabellenspalten **ACCOUNT\_TYPE** und **ACCOUNT\_NAME** unversorgt. Die Spalten **\_ID** und **CONTACT\_ID** werden automatisch betreut, sodass wir mit einem leeren **ContentValues**-Objekt arbeiten können. Es wird neben dem **Uri**-Objekt zur **raw\_contacts**-Tabelle des Providers an die **ContentResolver**-Methode **insert()** übergeben:

```
ContentValues values = new ContentValues();
Uri rawContactUri =
    getContentResolver().insert(ContactsContract.RawContacts.CONTENT_URI, values);
long rawContactId = ContentUris.parseId(rawContactUri);
```

Von **insert()** erhalten wir als Rückgabe ein **Uri**-Objekt zum eingefügten Rohkontakt. Daraus extrahieren wir die anschließend benötigte **RAW\_CONTACT\_ID**.

Im zweiten Schritt tragen wir Informationen zum neuen Rohkontakt in die **data**-Tabelle des Kontakte-Providers ein. Wie in Abschnitt 14.1.4 zu erfahren war, kann eine **data**-Tabellenzeile diverse Informationen aufnehmen (z.B. Anzeigename, Telefonnummern). Der MIMETYPE für eine Zeile mit Namensbestandteilen befindet sich in der folgenden Konstanten:

```
ContactsContract.CommonDataKinds.StructuredName.CONTENT_ITEM_TYPE
```

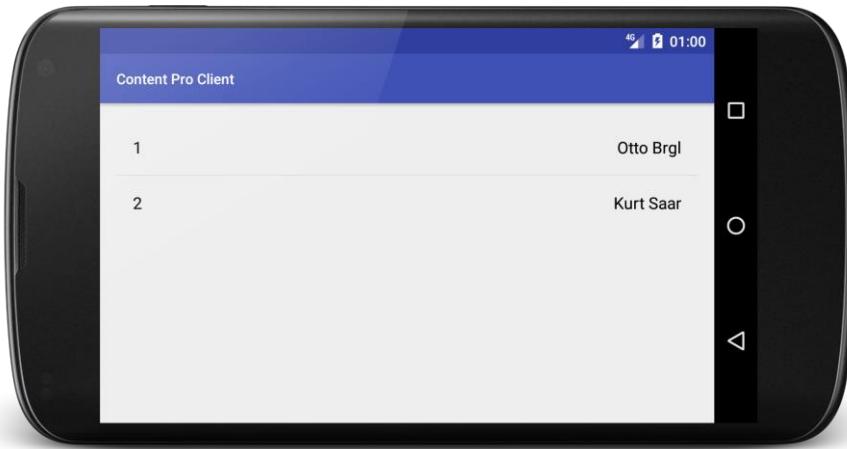
Eine **data**-Zeile mit diesem MIMETYPE hat 9 Spalten mit den Namen **DATA1** bis **DATA9**, die jeweils zur Aufnahme eines bestimmten Namensbestandteils vorgesehen sind. In der Spalte **DATA1** mit dem Aliasnamen **DISPLAY\_NAME** landet der Anzeigename. In den folgenden Anweisungen werden für ein neues **ContentValues**-Objekt durch **put()** - Aufrufe eine **RAW\_CONTACT\_ID**, ein **MIMETYPE** und ein **DISPLAY\_NAME** festgelegt. Das **ContentValues**-Objekt wird neben dem **Uri**-Objekt zur **data**-Tabelle des Providers an die **ContentResolver**-Methode **insert()** übergeben:

```
values.put(ContactsContract.Data.RAW_CONTACT_ID, rawContactId);
values.put(ContactsContract.Data.MIMETYPE,
          ContactsContract.CommonDataKinds.StructuredName.CONTENT_ITEM_TYPE);
values.put(ContactsContract.CommonDataKinds.StructuredName.DISPLAY_NAME, "Kurt Saar");
getContentResolver().insert(ContactsContract.Data.CONTENT_URI, values);
```

Wie die erweiterte App zeigt, hat Android aus dem eingefügten Rohkontakt einen neuen Eintrag in der **contacts**-Tabelle erstellt:

---

<sup>1</sup> <https://developer.android.com/guide/topics/providers/contacts-provider>



### update()

Um Einträge in einer Tabelle eines Content Providers zu ändern, verwendet man wie beim Einfügen einer neuen Zeile ein **ContentValues**-Objekt:

- Man muss nur die zu verändernden Attribute einfügen (mit der **ContentValues**-Methode **put()**).
- Um ein Attribut zu löschen, verwendet man die **ContentValues**-Methode **putNull()**.

Um die Menge der zu ändernden Tabellenzeilen zu definieren, verwendet man wie bei einer Abfrage die Parameter *selection* und *selectionArgs* (vgl. Abschnitt 14.1.3). Schließlich werden das **Uri**-Objekt zum **ContentProvider**, das **ContentValues**-Objekt und die Auswahlparameter an die **ContentResolver**-Methode **update()** übergeben.

Wir ändern nun den gerade eingefügten Rohkontakt und verwenden für die erforderliche Auswahl der **row\_contacts**-Zeile der Bequemlichkeit halber die seit dem Einfügen von Kurt Saar (siehe oben) bekannte **RAW\_CONTACT\_ID**. Im Zusammenhang mit dem Löschen eines Rohkontakts werden wir uns der Aufgabe stellen, die **RAW\_CONTACT\_ID** aus anderen Informationen zu ermitteln.

Um den Anzeigennamen des Rohkontakts zu ändern, setzen wir das vorhandene **ContentValues**-Objekt zurück und fügen den gewünschten Anzeigennamen ein:

```
values.clear();
values.put(ContactContract.CommonDataKinds.StructuredName.DISPLAY_NAME, "Ludger Saar");
```

Wir verwenden eine parametrisierte Auswahl mit einer Selektions-Zeichenfolge und einem **String[]** - Array mit Selektionsargumenten:

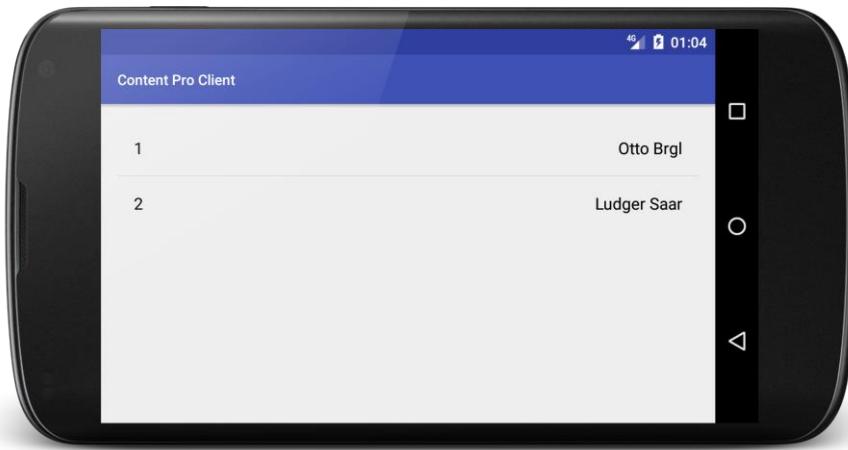
```
String where;
String[] whereArgs;
where = ContactContract.Data.RAW_CONTACT_ID + " = ?";
whereArgs = new String[] {String.valueOf(rawContactId)};
```

An die **ContentResolver**-Methode **update()** werden das **Uri**-Objekt zur **data**-Tabelle des Providers, das **ContentValues**-Objekt sowie sie Selektionsparameter übergeben:

```
int rowsChanged;
rowsChanged = getContentResolver().update(ContactContract.Data.CONTENT_URI, values,
   where, whereArgs);
```

Als Rückgabe erhalten wir die Anzahl der geänderten Tabellenzeilen.

Wie die erweiterte App zeigt, hat Android die Änderungen am Rohkontakt in die **contacts**-Tabelle übernommen:



### **delete()**

Um eine Zeile aus einer Tabelle eines Content Providers zu *löschen*, verwendet man die **ContentResolver**-Methode **delete()** und definiert die Menge der betroffenen Tabellenzeilen wie bei den **ContentResolver**-Methoden **query()** und **update()** über die Auswahlparameter *selection* und *selectionArgs*.

Wir löschen aus der **raw\_contacts**-Tabelle des Kontakte-Providers den ersten Eintrag mit dem Anzeigenamen Ludger Saar. Um die **RAW\_CONTACT\_ID** zu ermitteln,

```
rawContactId = getRawContactIdByName("Ludger", "Saar");
```

verwenden wir die folgende Hilfsmethode, die mit einer Abfrage (siehe Abschnitt 14.1.3) arbeitet und aus dem erhaltenen **Cursor**-Objekt die letzte (und hoffentlich einzige) Zeile verwendet:<sup>1</sup>

```
private long getRawContactIdByName(String givenName, String familyName) {
    ContentResolver contentResolver = getContentResolver();
    String[] projection = {ContactsContract.RawContacts._ID};
    String displayName = givenName + " " + familyName;
    String whereClause = ContactsContract.RawContacts.DISPLAY_NAME_PRIMARY + " = ?";
    String[] whereArgs = new String[] {displayName};

    Cursor cursor = contentResolver.query(ContactsContract.RawContacts.CONTENT_URI,
        projection, whereClause, whereArgs, null);
    long rawContactId = -1;
    if(cursor!=null) {
        int queryResultCount = cursor.getCount();
        if(queryResultCount > 0) {
            cursor.moveToLast();
            rawContactId = cursor.getLong(cursor.getColumnIndex(ContactsContract.RawContacts._ID));
        }
    }
    return rawContactId;
}
```

Die Auswahl des zu löschen Rohkontakts erfolgt nach bewährtem Rezept:

```
where = ContactsContract.RawContacts._ID + " = ?";
whereArgs = new String[] {String.valueOf(rawContactId)};
```

Das zur **raw\_contacts**-Tabelle des Kontakte-Providers gehörige **Uri**-Objekt muss darüber informiert werden, dass *kein* Synchronisations-Adapter im Spiel ist, damit die Löschung nicht nur vor-gemerkt, sondern sofort ausgeführt wird:

```
Uri uri = ContactsContract.RawContacts.CONTENT_URI;
uri.buildUpon().appendQueryParameter(ContactsContract.CALLER_IS_SYNCADAPTER, "false");
```

---

<sup>1</sup> Die wesentlichen Ideen zur folgenden Methode stammen von:

<https://www.dev2qa.com/how-to-update-delete-android-contacts-programmatically/>

Schließlich kann die **ContentResolver**-Methode **delete()** aufgerufen werden:

```
rowsChanged = getContentResolver().delete(uri, where, whereArgs);
```

Als Rückgabe vom Typ **int** erhält man die Anzahl der gelöschten Zeilen.

### 14.1.6 SQL-Injektion

Es ist strikt zu vermeiden, bei einer Abfrage oder Modifikation den Parameter *selection* nach dem folgenden Muster

```
String where = ContactsContract.RawContacts.DISPLAY_NAME_PRIMARY +
    "='" + userInput + "'";
long anzahl = getContentResolver().delete(ContactsContract.RawContacts.CONTENT_URI,
    where, null);
```

unter Verwendung von *Benutzereingaben* zu konstruieren, weil böswillige oder ungeschickte Eingaben durch eine so genannte **SQL-Injektion** zu einem unerwünschten Verhalten führen können. Das Szenario lässt sich auch in unserem Beispiel demonstrieren, wenngleich die Realitätsnähe zu wünschen übrig lässt. Wir stellen uns vor, dass der Benutzer mit einer Aktivität spielt, die das Löschen eines Kontaktes erlaubt und dass er dabei versehentlich die folgende Eingabe macht:

```
' OR '1'='1
```

Insgesamt resultiert ungefähr das folgende **DELETE**-Kommando:

```
DELETE FROM raw_contacts WHERE display_name=' ' OR '1'='1'
```

Wegen der Tautologie im rechten Teil des logischen Ausdrucks ist die **WHERE**-Bedingung immer wahr, und es werden *alle* Rohkontakte gelöscht.

Beim **delete()** - Aufruf mit einer parametrisierten Anfrage werden die Benutzerbeiträge strikt als Daten behandelt und können keine SQL-Syntax einschleusen. Weil vermutlich der Anzeigename

```
' OR '1'='1
```

nicht vorkommt, bleibt die Kontaktedatenbank unverändert.

### 14.1.7 Zugriffsrechte

Damit die eben präsentierten Abfragen bzw. Modifikationen tatsächlich möglich sind, muss eine Klienten-Anwendung in ihrer Manifestdatei das Recht zum lesenden bzw. zum schreibenden Zugriff auf den Kontakte-Provider über **uses-permission** -Elemente beantragen, z. B.:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="de.zimkand.contentproclient" >

    <uses-permission android:name="android.permission.READ_CONTACTS"/>
    <uses-permission android:name="android.permission.WRITE_CONTACTS"/>

    <application
        . . .
    </application>

</manifest>
```

Die vom Entwickler des Providers festgelegte Bezeichnung für ein Zugriffsrecht ist der Dokumentation zu entnehmen. Zu den von Android definierten Zugriffsrechten sind die Bezeichnungen in der SDK-Klasse **Manifest.permission** (im Paket: **android**) dokumentiert. Hier befinden sich z.B. die folgenden **String**-Konstanten:

- **public static final String READ\_CONTACTS**  
Inhalt: `android.permission.READ_CONTACTS`
- **public static final String WRITE\_CONTACTS**  
Inhalt: `android.permission.WRITE_CONTACTS`

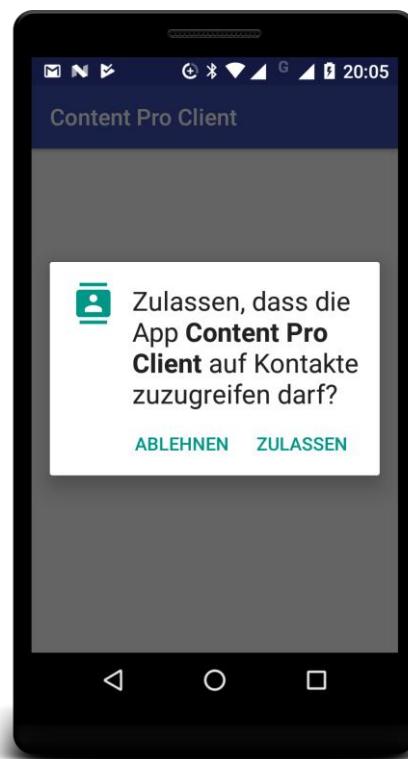
Bei der Installation einer App müssen die von ihr angeforderten Rechte durch den Benutzer genehmigt werden.

Eine weitere Voraussetzung für die Nutzbarkeit eines Content Providers durch fremde Anwendungen besteht darin, dass im **provider** - Element der Manifestdatei das Attribut **android:exported** den Wert **true** besitzen muss (= Voreinstellung).

Ab Android 6 muss eine benötigte Berechtigung nicht nur in der Manifestdatei deklariert, sondern auch einmalig zur Laufzeit beim Benutzer angefordert werden. Unser Beispielprogramm stellt über die statische **ContextCompat**-Methode **checkSelfPermission()** fest, ob das Recht zum Schreibzugriff auf den Kontakte-Provider besteht und fordert bei negativer Auskunft dieses Recht über die statische **ActivityCompat**-Methode **requestPermissions()** an:

```
if (ContextCompat.checkSelfPermission(this, Manifest.permission.WRITE_CONTACTS)
    != PackageManager.PERMISSION_GRANTED)
    ActivityCompat.requestPermissions(this,
        new String[] {Manifest.permission.WRITE_CONTACTS}, PERMISSIONS_REQUEST_WRITE_CONTACTS);
else
    modifyContacts();
```

Daraufhin erhält der Benutzer die folgende Anfrage:



Wie das Bildschirmfoto zeigt, erkundigt sich Android nicht speziell nach dem Schreibrecht, sondern allgemein nach dem Zugriffsrecht. Das Schreib- und das Leserecht für den Kontakte-Provider befinden sich in derselben **Berechtigungsgruppe** (engl. *permission group*), und Android (ab Version 6 bzw. API-Level 23) reagiert folgendermaßen auf die Anforderung von Berechtigungen aus einer Gruppe:

- Wurde einer App noch keine Berechtigung aus der Gruppe zugebilligt, dann erkundigt sich Android beim Benutzer.
- Hat eine App bereits eine Berechtigung aus der Gruppe erhalten, dann erfolgt *keine weitere* Nachfrage beim Benutzer.

Um auf die Antwort reagieren zu können, implementiert unsere Aktivität das Interface

```
public class MainActivity extends AppCompatActivity
    implements ActivityCompat.OnRequestPermissionsResultCallback {
    . . .
}
```

und überschreibt die Rückrufmethode **onRequestPermissionsResult()**:<sup>1</sup>

```
@Override
public void onRequestPermissionsResult(int requestCode,
                                       String permissions[], int[] grantResults) {
    switch (requestCode) {
        case PERMISSIONS_REQUEST_WRITE_CONTACTS: {
            // If request is cancelled, the result arrays are empty.
            if (grantResults.length > 0
                && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
                modifyContacts();
            } else
                Toast.makeText(this, "Write-Permission not granted", Toast.LENGTH_SHORT).show();
            return;
        }
        case PERMISSIONS_REQUEST_READ_CONTACTS: {
            // If request is cancelled, the result arrays are empty.
            if (grantResults.length > 0
                && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
                showContacts();
            } else
                Toast.makeText(this, "Read-Permission not granted", Toast.LENGTH_SHORT).show();
            return;
        }
    }
}
```

#### 14.1.8 Alternative Techniken für den Zugriff auf einen Content Provider

In der Online-Dokumentation zu den Content-Provider - Grundlagen informiert Google über einige alternative Techniken für den Zugriff auf einen Content Provider:<sup>2</sup>

- Stapelbetrieb  
Man kann eine Anzahl von Content Provider - Operationen in einem Objekt vom Typ **ArrayList<ContentProviderOperation>** zusammenstellen und dann mit der **ContentResolver**-Methode **applyBatch()** im Stapelbetrieb ausführen lassen.
- Asynchrone Ausführung  
Über die asynchrone Ausführung von Content Provider - Zugriffen mit Hilfe der Klasse **CursorLoader** informiert der Abschnitt 14.3.
- Zugriff mit temporärer Berechtigung für einen konkreten Content-URI  
Eine Anwendung ohne permanentes (bei der Installation erworbenes) Zugriffsrecht für einen Content Provider kann ein temporäres (bis zur Beendigung der Anwendungsinstantz gültiges) Zugriffsrecht für einen konkreten Content-URI erhalten (eine URI Permission). Das geschieht über die folgenden Flags

<sup>1</sup> Vorbild für diese Methode: <https://developer.android.com/training/permissions/requesting#handle-response>

<sup>2</sup> <https://developer.android.com/guide/topics/providers/content-provider-basics#AltForms>

- `Intent.FLAG_GRANT_READ_URI_PERMISSION`
- `Intent.FLAG_GRANT_WRITE_URI_PERMISSION`

zu einem **Intent**-Objekt, das einer Aktivität beim `startActivity()` - Aufruf übergeben oder als Ergebnis eines `startActivityForResult()` - Aufrufs zurückgeliefert wird. Im folgenden Beispiel wird die Anzeige der per **Uri**-Objekt angesprochenen Provider-Daten durch eine ansonsten nicht berechtigte Aktivität erlaubt:

```
Intent uriIntent = new Intent(Intent.ACTION_VIEW);
uriIntent.setData(contentUri);
uriIntent.addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);
startActivity(uriIntent);
```

Weitere Details finden sich in der Online-Dokumentation für Android-Entwickler.<sup>1</sup>

- Beauftragung einer berechtigten und kompetenten Aktivität  
Eine Anwendung ohne Zugriffsrecht kann zur Realisation eines bestimmten Zwecks (z.B. wiederkehrenden Termin im Kalender eintragen) über einen passenden Intent eine berechtigte und kompetente Aktivität beauftragen, die sich in den Vordergrund schiebt und mit dem Benutzer interagiert. Diese Zugriffart hat den Vorteil, dass komplexe Datenbankeinträge nur von kompetenten Aktivitäten vorgenommen werden.

#### 14.1.9 Dokumentation und Kontrakt-Klasse

Informationen für die erfolgreiche Nutzung eines Content Providers sollten vom Entwickler in einer so genannten *Kontrakt-Klasse* gesammelt und zur Verfügung gestellt werden, z.B.:

- Content-URIs
- Spaltennamen zu den Tabellen
- Intent-Aktionen für den Erwerb temporärer URI-Rechte oder für die Delegation von Aufgaben an berechtigte Aktivitäten (siehe Abschnitt 14.1.8)

Zum Kontakte-Provider erfüllt die Klasse **ContactsContract** im Paket **android.provider** mit ihren zahlreichen statischen inneren Klassen (z.B. **Contacts**, **RawContacts**, **Data**), diesen Zweck. In Abschnitt 14.1.1 haben wir aus der Klasse **ContactsContract.Contacts** die Konstante **CONTENT\_URI** als Referenz auf das **Uri**-Objekt zur **contacts**-Tabelle verwendet:

```
Uri uri = ContactsContract.Contacts.CONTENT_URI;
```

Content Provider sollten auf Befragen mit `getType()` zu jedem URI einen MIME-Typ nennen, um über die Beschaffenheit der gelieferten Daten zu informieren. Ein MIME-Typ besteht aus Typ und Subtyp (z.B. **text/html**), wobei standardisierte Typen genutzt oder Provider-spezifische Typen definiert werden können. Für strukturierte (Datenbank-analoge) Daten werden unter Android generell die folgenden MIME-Typen verwendet:

- Für eine Ergebnistabelle mit potentiell mehreren Zeilen:  
**vnd.android.cursor.dir**
- Für eine einzelne Zeile:  
**vnd.android.cursor.item**

Die Subtypen sind Provider-spezifisch. Auf den Webseiten für Android-Entwickler wird empfohlen, auf die Einleitung **vnd** einen weltweit eindeutigen Namen (z.B. abgeleitet aus dem Domänen-namen) und eine Bezeichnung des Inhalts (z.B. den Namen einer Datenbanktabelle) folgen zu lassen:<sup>2</sup>

---

<sup>1</sup> <https://developer.android.com/guide/topics/permissions/overview>

<sup>2</sup> <https://developer.android.com/guide/topics/providers/content-provider-creating#TableMIMETypes>

- URI `content://com.example.app.provider/tablone/vnd.android.cursor.dir/vnd.com.example.provider.tablone`
- URI `content://com.example.app.provider/tablone/1/vnd.android.cursor.item/vnd.com.example.provider.tablone`

## 14.2 Content Provider erstellen

Über einen Content Provider kann eine App eigene Daten für fremde Anwendungen verfügbar machen. Eigene Daten können angeboten werden als:

- Dateien  
Wenn ein Provider Medien anbietet, bietet sich zur Datenablage eine Kombination von Dateien und Datenbanktabellen an.
- Strukturierte Daten in Tabellenform  
Zur Speicherung bzw. Verwaltung der Tabellen bietet sich eine SQLite-Datenbank an (siehe Kapitel 13). Diese Standardlösung wird z.B. von allen Android-internen Content Providern verwendet.

Zwar sind Content Provider in erster Linie für die Verwendung durch andere Anwendungen gedacht, doch kann es auch Gründe für die Nutzung durch andere Komponenten der *eigenen* App geben. Z.B. lässt sich auf diese Weise ein asynchroner Datenzugriff realisieren (vgl. Abschnitt 14.3).

Wir werden uns in diesem Abschnitt auf Content Provider zur Präsentation von strukturierten Daten beschränken und ein Beispiel konstruieren, das als Schnittstelle zu der in Abschnitt 13.2 vorgestellten Ereignisdatenbank fungiert.

Wer weitere Informationen zur Erstellung von Content Providern benötigt, wird z.B. in der Online-Dokumentation zu Android fündig.<sup>1</sup>

### 14.2.1 Content-URIs und andere Bestandteile der Schnittstelle definieren

Bei jedem Zugriff auf die Daten eines Content Providers ist ein **Uri**-Objekt an den **ContentResolver** zu übergeben. Ein URI (*Uniform Resource Identifier*) besitzt folgende Bestandteile, um den Provider und die Daten zu identifizieren (vgl. Abschnitt 14.1.1):

- Typ (Schema)  
Jeder zu einem Content Provider gehörige URI startet mit:  
**content://**
- Name des Providers (Autorität)  
Damit ein in der Android-Welt eindeutiger Name entsteht, sollte die folgende Bildungsregel verwendet werden:

***reverse-domain-name.application-name.provider-name***

Für den letzten Bestandteil empfiehlt Google die schlichte Bezeichnung *provider*, so dass z.B. für den Content Provider zum Ereignistagebuch der folgende Name resultiert:

**de.zimkand.eventdiary.provider**

Gemäß Internet-Terminologie wird mit dem Provider-Namen die *Autorität* des URIs angegeben.

---

<sup>1</sup> <https://developer.android.com/guide/topics/providers/content-provider-creating>

- Pfad

Der Pfadname zeigt auf eine Datenbanktabelle oder auf eine Datei. Für die Ereignistabelle im Content Provider zum Ereignistagebuch eignet sich z.B. die Bezeichnung `events`, so dass insgesamt der folgende URI resultiert:

```
content://de.zimkand.eventdiary.provider/events
```

Im Allgemeinen kann ein Provider *mehrere* Tabellen anbieten. Es dürfen Pfadnamen mit mehreren Segmenten verwendet werden.

- Zeilennummer

Zum Zugriff auf eine einzelne Tabellenzeile ist zusätzlich deren Nummer anzugeben, z.B.:

```
content://de.zimkand.eventdiary.provider/events/7
```

Diese wird in der Regel mit der als `_ID` bezeichneten Primärindexspalte der Tabelle abgeglichen.<sup>1</sup>

Bei der Konzeption eines Content Providers zur Präsentation von strukturierten Daten sind also festzulegen:

- Der Name des Providers (die Autorität)
- Für mindestens eine Tabelle der Name und für jede Tabellenspalte der Datentyp und der Name
- Für jeden akzeptierten URI ein MIME-Type

Soll eine Anwendung für den indirekten Zugang zu einem Content Provider auch von Aktivitäten auszuführende Intents unterstützen, dann sind deren Aktionen, Extras und Flags zu definieren (vgl. Abschnitt 9.1).

Alle zur Verwendung eines Content Providers erforderlichen Informationen sollten in einer Kontrakt-Klasse definiert werden (vgl. Abschnitt 14.1.9).

### 14.2.2 ContentProvider-Ableitung implementieren

Um den Klienten Zugriff auf strukturierte Daten zu gewähren, ist eine Ableitung der abstrakten Klasse **ContentProvider** zu definieren. Dabei sind sechs abstrakte Methoden zu implementieren, die mit Ausnahme der Methode `onCreate()` von einem klientenseitigen **ContentResolver**-Objekt aufgerufen werden (vgl. Abschnitt Fehler! Verweisquelle konnte nicht gefunden werden.):<sup>2</sup>

- **public abstract boolean onCreate()**

Diese Initialisierungsmethode wird von Android beim Anwendungsstart im Main-Thread aufgerufen. Aufwändige Operationen (z.B. Öffnen einer Datenbank, Migration auf eine neue Datenbankversion) können den Anwendungsstart dementsprechend verzögern. Solche Operationen bis zur ersten Verwendung aufzuschieben, hat folgende Vorteile:<sup>3</sup>

- Der Anwendungsstart wird nicht verzögert oder gar durch Probleme bei der Initialisierung verhindert.
- Findet kein Provider-Zugriff statt, entfällt der Initialisierungsaufwand komplett.

---

<sup>1</sup> Damit ein als **Cursor**-Objekt von einem Content Provider bezogenes Abfrageergebnis mit Hilfe der Klasse **CursorAdapter** genutzt und z.B. an ein **ListView**-Steuerelement gebunden werden kann, muss eine als `_id` bezeichnete Spalte vorhanden sein.

<sup>2</sup> Zu jeder **ContentProvider**-Zugriffsmethode existiert in der Klasse **ContentResolver** ein Gegenstück mit identischer Signatur.

<sup>3</sup> [`https://developer.android.com/reference/android/content/ContentProvider.html#onCreate\(\)`](https://developer.android.com/reference/android/content/ContentProvider.html#onCreate())

- **public abstract Cursor query()**  
Klienten (**ContentResolver**) können eine Abfrage vornehmen und erhalten ein **Cursor**-Objekt als Rückgabe. Über Parameter wählen die Klienten eine Tabelle, eine Auswahl von Spalten bzw. Zeilen sowie eine Sortierung (vgl. Abschnitt 14.1.3).
- **public abstract Uri insert()**  
Klienten können eine neue Zeile in eine Tabelle einfügen und erhalten das **Uri**-Objekt zur neuen Zeile als Rückgabe (vgl. Abschnitt 14.1.4).
- **public abstract int update()**  
Klienten können Zeilen modifizieren und erhalten die Anzahl der betroffenen Zeilen als Rückgabe. Über Parameter wählen die Klienten eine Tabelle, eine Auswahl von Zeilen sowie die neuen Spaltenwerte (vgl. Abschnitt 14.1.4).
- **public abstract int delete()**  
Klienten können Zeilen löschen und erhalten die Anzahl der betroffenen Zeilen als Rückgabe. Über Parameter wählen die Klienten eine Tabelle und eine Auswahl von Zeilen (vgl. Abschnitt 14.1.4).
- **public abstract String getType()**  
Klienten können den MIME-Typ zu einem URI erfragen.

Mit Ausnahme von **onCreate()** müssen alle Methoden Thread-sicher sein, weil sie eventuell aus verschiedenen Threads aufgerufen werden. Verwendet ein Provider eine SQLite-Datenbank profitieren seine Datenbankzugriffsmethoden von der Thread-Sicherheit des SQLite-Datenbankmanagementsystems.<sup>1</sup>

Soll auf eine Funktionalität verzichtet werden (z.B. auf das Löschen von Tabellenzeilen), kann man die betroffene Methode „leer“ implementieren und eine passende Rückgabe liefern (z.B. den Wert 0 bei der Methode **delete()**).

Mit dem Aufrufer per Ausnahmeobjekt zu kommunizieren, ist nur eingeschränkt möglich, weil das Ausnahmeobjekt vom Prozess mit dem **ContentProvider** zum Prozess mit dem **ContentResolver** übertragen (dort neu geworfen) werden muss, was nur bei den folgenden Ausnahmeklassen möglich ist:<sup>2</sup>

- **IllegalArgumentException**  
Diese Ausnahmeklasse bietet sich z.B. an, wenn der Klient einen fehlerhaften **Uri**-Parameter sendet.
- **NullPointerException**

Um einen Rohling für den Content Provider mit provisorischen Implementationen für alle abstrakten **ContentProvider**-Methoden in ein Android Studio - Projekt einzufügen, wählen wir im Projektexplorer aus dem Kontextmenü zum Paket mit den Klassen des Projekts (z.B. `de.zimkand.eventdiary`) den Eintrag

#### New > Other > Content Provider

Als Klassenname eignet sich `EventProvider`.

<sup>1</sup> <https://www.sqlite.org/faq.html>

<sup>2</sup> <https://developer.android.com/guide/topics/providers/content-provider-creating>

#### 14.2.2.1 Room Persistence Library

Wir übernehmen die Datenbanktechnik von der bisherigen Ausbaustufe des Ereignistagebuchprojekts (siehe Abschnitt 13.2), müssen aber die DAO-Schnittstelle um Methoden mit einem **Cursor**-Objekt als Rückgabe erweitern:<sup>1</sup>

```
@Dao
public interface EventDAO {
    ...
    @Query("SELECT * FROM events")
    Cursor selectAll();

    @Query("SELECT * FROM events WHERE _id = :rowid")
    Cursor selectById(long rowid);

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    void insert(Event... events);

    @Query("DELETE FROM " + EventProvider.TABLE_EVENTS + " WHERE " +
           EventProvider.COL_ID + " = :rowid")
    int deleteById(long rowid);

    @Query("SELECT COUNT(*) FROM " + EventProvider.TABLE_EVENTS)
    int count();
}
```

Wir kommen um diese Methoden nicht herum, weil die **ContentProvider**-Methode `query()` obligatorisch ein **Cursor**-Objekt liefern muss. Allerdings klingen die folgenden Zeilen aus der Room-Dokumentation von Google leicht beunruhigend:

**Caution:** It's highly discouraged to work with the Cursor API because it doesn't guarantee whether the rows exist or what values the rows contain. Use this functionality only if you already have code that expects a cursor and that you can't refactor easily.

In der `EventDAO`-Definition ergänzen wir außerdem die Methode `deleteById()`, um die Implementation der **ContentProvider**-Methode `delete()` vereinfachen zu können (vgl. Abschnitt 14.2.2.6).

Auch die Entitätsklasse `Event` benötigt Anpassungen zur Unterstützung des Content Providers.

---

<sup>1</sup> Die Zusammenführung der älteren Content Provider - Technik mit der topaktuellen Room Persistence Library orientiert sich an <https://github.com/googlesamples/android-architecture-components/tree/master/PersistenceContentProviderSample>

```

@Entity(tableName = "events")
public class Event {

    @PrimaryKey(autoGenerate = true)
    @ColumnInfo(name = "_id")
    private long id;

    . . .

    @Update
    int update(Event... events);

    . . .

    public void setValues(ContentValues values) {
        if (values.containsKey(EventProvider.COL_DATETIME)) {
            dateTime = new Date(values.getAsLong(EventProvider.COL_DATETIME));
        }
        if (values.containsKey(EventProvider.COL_DESC)) {
            desc = values.getAsString(EventProvider.COL_DESC);
        }
        if (values.containsKey(EventProvider.COL_MOOD)) {
            mood = values.getAsInteger(EventProvider.COL_MOOD);
        }
    }
}

public static Event fromContentValues(ContentValues values) {
    Event event = new Event();
    event.setValues(values);
    return event;
}
}

```

Wie Sie mittlerweile wissen, spielt die Klasse **Cursor** bei der Verwendung eines Content Providers eine zentrale Rolle als Rückgabetyp der Methode **query()**. Damit ein als **Cursor**-Objekt von einem Content Provider bezogenes Abfrageergebnis mit Hilfe der Klasse **CursorAdapter** genutzt und z.B. an ein **ListView**-Steuerelement gebunden werden kann, muss eine Spalte mit dem Namen **\_id** vorhanden sein. Ansonsten kommt es zu der folgenden Fehlermeldung:

```
error: There is a problem with the query: [SQLITE_ERROR] SQL error or missing database (no such column: _id)
```

Daher erhält in der zur Entitätsklasse **Event** gehörenden Datenbanktabelle die als Primärschlüssel dienende Spalte den Namen **\_id**.

```

@PrimaryKey(autoGenerate = true)
@ColumnInfo(name = "_id")
private long id;

```

In der Methode **update()** wurde der Rückgabetyp **void** durch die Alternative **int** ersetzt (siehe Abschnitt 13.1.2.2).

Außerdem ergänzen wir die Methoden **setValues()** und **fromContentValues()** um die Implementation der **ContentProvider**-Methoden **update()** und **insert()** zu vereinfachen (vgl. Abschnitt 14.2.2.7 und 14.2.2.5)

### 14.2.2.2 UriMatcher

Die Datenzugriffsmethoden der Klasse **ContentProvider** erhalten ein **Uri**-Parameterobjekt, das von unterschiedlicher (eventuell auch von defekter) Gestalt sein kann. Bei der Beurteilung und Differenzierung hilft die Klasse **UriMatcher**. Ein entsprechend instruiertes Objekt dieser Klasse bildet per **match()** - Methode gültige URIs auf ganze Zahlen ab, die später bequem in einer **switch**-Anweisung verwendet werden können. Weil pro **ContentProvider** nur *ein* Projekt der Klasse **UriMatcher** benötigt wird, eignet sich ein statisches Member-Objekt. Das folgende Codesegment aus der **ContentProvider**-Ableitung **EventProvider** zum Ereignistagebuchprojekt enthält einen statischen Initialisierer, der beim Laden der Klasse ausgeführt wird und das **UriMatcher**-Objekt initialisiert:

```
private static final String AUTHORITY = "de.zimkand.eventdiary.provider";
public static final String TABLE_EVENTS = "events";
private static final UriMatcher uriMatcher;
private static final int EVENTS_DIR = 1;
private static final int EVENTS_ITEM = 2;

static {
    uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
    uriMatcher.addURI(AUTHORITY, TABLE_EVENTS, EVENTS_DIR);
    uriMatcher.addURI(AUTHORITY, TABLE_EVENTS +"/#", EVENTS_ITEM);
}
```

Aufgrund dieser Initialisierung durch die beiden **addURI()** - Aufrufe liefert die **match()** - Methode des **UriMatcher**-Objekts eine 1, wenn die gesamte Ereignistabelle angesprochen wird. Bezieht sich ein URI auf eine einzelne Tabellenzeile, dann liefert **match()** eine 2.

Im *path*-Parameter des zweiten **addURI()** - Aufrufs wird Platzhalterzeichen # für Zahlen verwendet. Durch den Platzhalter \* kann im *path*-Parameter eine beliebige Zeichenfolge vertreten werden.

In der **EventProvider**-Methode **query()** wird eine zur **UriMatcher**-Klassifikation passende **EventDAO**-Methode aufgerufen:

```
@Override
public Cursor query(Uri uri, String[] projection, String selection,
                     String[] selectionArgs, String sortOrder) {
    Cursor result;
    switch (uriMatcher.match(uri)) {
        case EVENTS_DIR:
            result = eventDAO.selectAll();
            break;
        case EVENTS_ITEM:
            result = eventDAO.selectById(Long.parseLong(uri.getLastPathSegment()));
            break;
        default:
            throw new IllegalArgumentException("URI invalid: " + uri);
    }
    return result;
}
```

Von den Methoden der Klasse **Uri** zur bequemen Verarbeitung von URIs wird im Beispiel die Methode **getLastPathSegment()** verwendet, um den letzten URI-Bestandteil abzuspalten.

### 14.2.2.3 *onCreate()*

Die Methode **onCreate()**

```
public abstract boolean onCreate()
```

kann sich darauf beschränken, das als Solist tätige Objekt aus der Klasse **EventDatabase** um eine Referenz auf das (ebenfalls nur einmal vorhandene) **EventDAO**-Objekt zu bitten (vgl. Abschnitt 13.2.5):

```
@Override
public boolean onCreate() {
    eventDAO = EventDatabase.getInstance(getContext()).getEventDAO();
    return true;
}
```

### 14.2.2.4 *query()*

Die **query()** - Implementation unseres Content Providers verwenden folgenden Methodenkopf

```
public abstract Cursor query(Uri uri, String[] projection, String selection,
                             String[] selectionArgs, String sortOrder)
```

und ruft passend zur **UriMatcher**-Klassifikation die Methode **selectAll()** oder die Methode **selectById()** des **EventDAO**-Objekts auf (siehe Abschnitt 14.2.2.2). Als Rückgabe ist ein **Cursor**-Objekt zu liefern, das eventuell keine Zeilen enthält, also auf Befragen mit **getCount()** die Anzahl 0 meldet. Auf eine ungültige URI-Angabe reagiert unsere **query()** - Überschreibung mit einer **IllegalArgumentException**.

### 14.2.2.5 *insert()*

Die **ContentProvider**-Methode **insert()**

```
public abstract Uri insert(Uri uri, ContentValues values)
```

fügt in die per **Uri**-Objekt spezifizierte Tabelle ein neues Objekt mit den per **ContentValues**-Objekt festgelegten Werten ein (siehe Abschnitt 14.1.4). Unsere Implementation erzeugt mit Hilfe der **Event**-Methode **fromContentValues()** ein neues **Event**-Objekt und fügt es mit Hilfe der **EventDAO**-Methode **insert()** in die Datenbanktabelle ein:

```
public Uri insert(Uri uri, ContentValues values) {
    switch (uriMatcher.match(uri)) {
        case EVENTS_DIR:
            Event event = Event.fromContentValues(values);
            eventDAO.insert(event);
            Toast.makeText(getApplicationContext(), R.string.inserted, Toast.LENGTH_SHORT).show();
            return ContentUris.withAppendedId(uri, event.getId());
        case EVENTS_ITEM:
            throw new IllegalArgumentException("Insertion with ID not allowed: " + uri);
        default:
            throw new IllegalArgumentException("URI invalid: " + uri);
    }
}
```

Auf ein ungeeignetes **Uri**-Objekt reagiert die Methode mit einer **IllegalArgumentException**. Das bei einem erfolgreichen Aufruf zu liefernde **Uri**-Objekt zur neuen Tabellenzeile wird über die Methode **withAppendedId()** der Klasse **ContentUris** erstellt.

#### 14.2.2.6 `delete()`

Die **ContentProvider**-Methode `delete()`

```
public abstract int delete(Uri uri, String selection, String[] selectionArgs)
```

löscht ...

- entweder aus der per **Uri**-Objekt spezifizierten Tabelle alle Zeilen, die der Auswahlbedingung entsprechen, welche über die Parameter *selection* und *selectionArgs* formuliert wird (siehe Abschnitt 14.1.3)
- oder die per **Uri**-Objekt mit Tabellen- und Kennungsangabe spezifizierte Einzelzeile.

Unsere Klasse **EventProvider** beherrscht nur das Löschen einer einzelnen Zeile und verwendet dazu die Methode `deleteById()` der Klasse **EventDAO** (siehe Abschnitt 14.2.2.1):

```
@Override
public int delete(Uri uri, String selection, String[] selectionArgs) {
    switch (uriMatcher.match(uri)) {
        case EVENTS_DIR:
            throw new IllegalArgumentException("Cannot delete without ID: " + uri);
        case EVENTS_ITEM:
            int count = eventDAO.deleteById(ContentUrис.parseId(uri));
            if (count > 0)
                Toast.makeText(getApplicationContext(), R.string.deleted, Toast.LENGTH_SHORT).show();
            return count;
        default:
            throw new IllegalArgumentException("URI invalid: " + uri);
    }
}
```

#### 14.2.2.7 `update()`

Die **ContentProvider**-Methode `update()`

```
public abstract int update(Uri uri, ContentValues values,
                           String selection, String[] selectionArgs)
```

aktualisiert ...

- entweder in der per **Uri**-Objekt spezifizierten Tabelle alle Zeilen, die der Auswahlbedingung entsprechen, welche über die Parameter *selection* und *selectionArgs* formuliert wird (siehe Abschnitt 14.1.3)
- oder die per **Uri**-Objekt mit Tabellen- und Kennungsangabe spezifizierte Einzelzeile.

Die neuen Werte liefert ein **ContentValues**-Objekt.

Unsere Klasse **EventProvider** beherrscht nur die Aktualisierung einer einzelnen Zeile und verwendet dazu die **Event**-Methode `setValues()` sowie die **EventDAO**-Methode `update()`:

```

@Override
public int update(Uri uri, ContentValues values, String selection, String[]
selectionArgs) {
    switch (uriMatcher.match(uri)) {
        case EVENTS_DIR:
            throw new IllegalArgumentException("Cannot update without ID: " + uri);
        case EVENTS_ITEM:
            Event event = eventDAO.getEvent(ContentUris.parseId(uri));
            event.setValues(values);
            int count = eventDAO.update(event);
            if (count > 0)
                Toast.makeText(getContext(), R.string.updated, Toast.LENGTH_SHORT).show();
            return count;
        default:
            throw new IllegalArgumentException("URI invalid: " + uri);
    }
}

```

An den Aufrufer wird die Zahl der geänderten Datenbankzeilen zurückgemeldet, wobei von unserer Implementation nur die Werte 0 und 1 zu erwarten sind.

#### 14.2.2.8 `getType()`

Ein Content Provider sollte über die Methode

```
public abstract String getType(Uri uri)
```

zu jedem unterstützten URI einen MIME-Typ liefern, um über die Beschaffenheit der Daten zu informieren (vgl. Abschnitt 14.1.9). Auch unsere Klasse `EventProvider` erfüllt diese Pflicht:

```

@Override
public String getType(Uri uri) {
    switch (uriMatcher.match(uri)) {
        case EVENTS_DIR:
            return "vnd.android.cursor.dir/vnd" + AUTHORITY + "." + TABLE_EVENTS;
        case EVENTS_ITEM:
            return "vnd.android.cursor.item/vnd" + AUTHORITY + "." + TABLE_EVENTS;
        default:
            throw new IllegalArgumentException("Unknown URI: " + uri);
    }
}

```

#### 14.2.3 Provider-Eintrag im Anwendungsmanifest

Content Provider müssen wie Activities und Services im Anwendungsmanifest deklariert werden. Lässt man einen Content Provider vom Android Studio angelegen, wird das erforderliche `provider`-Element automatisch in die Manifestdatei eingetragen, z.B. beim `EventProvider`:

```

<provider
    android:name=".EventProvider"
    android:authorities="de.zimkand.eventdiary.provider"
    android:enabled="true"
    android:exported="true" >
</provider>

```

Hier sind u.a. die folgenden Informationen für das Betriebssystem enthalten:<sup>1</sup>

---

<sup>1</sup> Weitere Details sind z.B. in der Online-Dokumentation zu Android zu finden:

<https://developer.android.com/guide/topics/manifest/provider-element>

- **android:name**  
In diesem Attribut wird der Name der implementierenden Klasse genannt.
- **android:authorities**  
Mit diesem Attribut wird der in Content-URIs zu verwendende Name des Providers festgelegt (vgl. Abschnitt 14.2.1). Android baut mit den Informationen aus diesem Attribut eine Tabelle mit den installierten Providern auf und ermittelt bei einer **ContentResolver**-Anfrage die zuständige Anwendungskomponente.
- Mit etlichen Attributen lässt sich die Behandlung und die Verfügbarkeit des Providers steuern, z.B.:
  - **android:enabled**  
Dieses Attribut mit der Voreinstellung **true** legt fest, ob der Provider verfügbar ist, d.h. ob Android beim App-Start ein Objekt des Providers instanzieren soll.
  - **android:exported**  
Dieses Attribut legt fest, ob *andere* Anwendungen auf den Provider zugreifen dürfen, wobei für den Zugriff auf einen öffentlichen (exportierten) Provider ggf. Zugriffrechte erforderlich sind.
- Zugriffsrechte  
Mit der Definition von Zugriffsrechten für fremde Anwendungen beschäftigen wir uns anschließend in Abschnitt 14.2.4.

#### 14.2.4 Zugriffsrechte definieren

Wenn ein Content Provider in der Anwendungs-Manifestdatei über den Wert **true** zum Attribut **android:exported** als öffentlich deklariert wurde, bestehen für beliebige Anwendungen volle Schreib- und Leserechte. Um diesen Zustand zu ändern, können im **provider** - Element der Manifestdatei (vgl. Abschnitt 14.2.2) Rechte auf Provider-, Tabellen oder Zeilenebene gesetzt werden. Wir beschränken uns auf die Definition von Lese- und Schreibrechten auf Providerebene. Weitere Details sind z.B. in der Online-Dokumentation zu Android zu finden:<sup>1</sup>

Für das Lese- und das Schreibrecht zum **EventProvider** wird jeweils ein möglichst weltweit eindeutiger Name festgelegt und dem Attribut **android:readPermission** bzw.

**android:readPermission** des **provider** - Elements der Manifestdatei zugewiesen, z.B.:

```
<provider
    android:name=".EventProvider"
    android:authorities="de.zimkand.eventdiary.provider"
    android:readPermission="de.zimkand.eventdiary.provider.permission.READ"
    android:writePermission="de.zimkand.eventdiary.provider.permission.WRITE"
    android:enabled="true"
    android:exported="true" >
</provider>
```

Damit eine fremde App lesend *und* schreibend auf unseren Provider zugreifen kann, muss sie in **uses-permission** - Elementen ihrer Manifestdatei die von uns definierten Rechte anmelden in der Hoffnung, dass der Anwender zustimmt und damit die Installation ermöglicht (vgl. Abschnitt 14.1.7):

```
<uses-permission android:name="de.zimkand.eventdiary.provider.permission.READ" />
<uses-permission android:name="de.zimkand.eventdiary.provider.permission.WRITE" />
```

Alternativ kann man für einen Provider mit dem Attribut **android:permission** das kombinierte Recht zum Lesen und zum Schreiben definieren, z.B.:

---

<sup>1</sup> <https://developer.android.com/guide/topics/providers/content-provider-creating#Permissions>

```
    android:permission="de.zimkand.eventdiary.provider.permission.READ_WRITE"
```

Werden neben dem generellen Recht auch spezifische Rechte definiert (also **android:readPermission** bzw. **android:readPermission**), dann dominieren letztere.

## 14.2.5 Modifikationen der bisherigen Event Diary - Klassen

### 14.2.5.1 Datenbankzugriffe nur noch im Content Provider

Nachdem unser Ereignistagebuch nunmehr über einen Content Provider verfügt, sollte diese Komponente auch von den anwendungseigenen Aktivitäten genutzt werden, um unsinnigen Doppelaufwand beim Datenbankzugriff zu vermeiden. Folglich werden in den Klassen **EventListActivity** und **EditEventActivity** die Datenbankzugriffe auf die Nutzung des **ContentResolver**-Objekts umgestellt. Das macht keine große Mühe, wie z.B. die Methode **onCreate()** der Klasse **EventListActivity** zeigt:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_event_list);

    lv = findViewById(R.id.listView);
    lv.setOnItemClickListener(this);
    registerForContextMenu(lv);

    cursor = getContentResolver().query(EventProvider.CONTENT_URI, null, null, null,
   EventProvider.COL_DATETIME + " DESC");
    if (cursor.getCount() == 0) {
        ContentValues values = new ContentValues();
        values.put(EventProvider.COL_DESC, "5 Cent gefunden");
        values.put(EventProvider.COL_MOOD, 5);
        values.put(EventProvider.COL_DATETIME, System.currentTimeMillis());
        getContentResolver().insert(EventProvider.CONTENT_URI, values);
        cursor.requery();
    }

    startManagingCursor(cursor);
    adapter = new EventDiaryAdapter(this, cursor);
    lv.setAdapter(adapter);
}
```

Durch den Methodenaufruf

```
startManagingCursor(cursor);
```

wird die Aktivität beauftragt, das als Parameter übergebene **Cursor**-Objekt im Rahmen ihrer Lebenszyklusmethoden zu verwalten:

- Der Cursor wird deaktiviert (über die **Cursor**-Methode **deactivate()**), wenn die Aktivität gestoppt wird.
- Der Cursor wird geschlossen (über die **Cursor**-Methode **close()**), wenn die Aktivität zerstört wird.
- Kehrt die Aktivität aus dem gestoppten Zustand zurück, erhält der Cursor (im UI-Thread!) einen **requery()** - Aufruf, um sich aktuelle Daten zu besorgen.

Nachdem die Methode **onCreate()** der Klasse **EventListActivity** in eine als leer vorgefundene Datenbanktabelle zu Demonstrationszwecken ein Ereignis eingefügt hat, veranlasst sie durch die **Cursor**-Methode **requery()** eine Wiederholung der Abfrage:

```
cursor.requery();
```

Die Methoden **startManagingCursor()**, **deactivate()** und **requery()** sind als abgewertet gekennzeichnet, weil sie Datenbankaktivitäten im UI-Thread vornehmen und damit potentiell die flüssige Bedienbarkeit der App gefährden. Außerdem verliert ein von der Aktivität verwalteter Cursor seine Daten bei einem Konfigurationswechsel, was vermeidbare Datenbankabfragen zur Folge hat. Mit dem sogenannten **Loader-API** und insbesondere mit der Klasse **CursorLoader** ist eine sinnvolle Lösung für die genannten Probleme verfügbar, die gleich in Abschnitt 14.3 vorgestellt wird.

#### 14.2.5.2 CursorAdapter

Für die Beschaffung der Daten zu einem **ListView**-Steuerelement sorgt generell ein Objekt aus einer Klasse, die das Interface **ListAdapter** (im Paket **android.widget**) erfüllt. Der im aktuellen Kapitel für den Datenzugriff verwendete Content Provider liefert als **query()** - Rückgabe ein **Cursor**-Objekt, und folglich ist die abstrakte Klasse **CursorAdapter** als Basis für die projektspezifische Ableitung **EventDiaryAdapter** sehr gut geeignet:

```
package de.zimkand.eventdiary;

import android.content.Context;
. . .
import java.util.Date;

public class EventDiaryAdapter extends CursorAdapter {
    private final Date date = new Date();
    private SimpleDateFormat sdf = new SimpleDateFormat("EEE, dd MMM yyyy, HH:mm:ss");
    private LayoutInflater inflater;
    private int colDateTime = 1, colEventDesc = 2, colMood = 3;

    public EventDiaryAdapter(Context context, Cursor c) {
        super(context, c, CursorAdapter.FLAG_REGISTER_CONTENT_OBSERVER);
        inflater = LayoutInflater.from(context);
    }

    @Override
    public View newView(Context context, Cursor cursor, ViewGroup parent) {
        return inflater.inflate(R.layout.entry, null);
    }

    @Override
    public void bindView(View view, Context context, Cursor cursor) {
        TextView tvDateTime = view.findViewById(R.id.dateTime);
        TextView tvMood = view.findViewById(R.id.moodValue);
        TextView tvDesc = view.findViewById(R.id.eventDesc);
        long timeMillis = cursor.getLong(colDateTime);
        date.setTime(timeMillis);
        tvDateTime.setText(sdf.format(date));
        tvMood.setText(context.getString(R.string.moodvalue) + " " +
                      Integer.toString(cursor.getInt(colMood)));
        tvDesc.setText(cursor.getString(colEventDesc));
    }
}
```

Im **EventDiaryAdapter**-Konstruktor wird der empfohlene Basisklassenkonstruktor aufgerufen, wobei im dritten Aufrufparameter durch die **int**-wertige Konstante **CursorAdapter.FLAG\_REGISTER\_CONTENT\_OBSERVER** dafür gesorgt wird, dass der Adapter von einer Änderung bei den **Cursor**-Daten erfährt und das **ListView**-Objekt auffordert, die Anzeige zu aktualisieren.<sup>1</sup> An-

---

<sup>1</sup> Siehe <https://developer.android.com/reference/android/widget/CursorAdapter>

schließend wird das in der gleich zu beschreibenden Methode **newView()** benötigte **LayoutInflater**-Objekt ermittelt.

Es sind zwei abstrakte **CursorAdapter**-Methoden zu implementieren:

- **newView()**

Diese Methode liefert eine **View**-Wurzel mit untergeordneten Anzeigeelementen für die Daten, auf die der **Cursor** zeigt. In der Regel definiert man für eine Zeile der Ergebnistabelle ein Layout per XML-Datei und gewinnt daraus über die **inflate()** - Methode der Klasse **LayoutInflater** eine **View**-Hierarchie. In unserem Beispiel enthält die Layoutdefinition für ein Ereignis drei **TextView**-Elemente zur Anzeige von Datum/Zeit, Stimmung und Ereignisbeschreibung:

```
<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="de.zimkand.eventdiary.EventListActivity">

    <TextView
        android:id="@+id/dateTime"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginLeft="16dp"
        android:layout_marginStart="16dp"
        android:layout_marginTop="16dp"
        android:textSize="16sp"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        tools:text="dateTime" />

    <TextView
        android:id="@+id/moodValue"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginEnd="16dp"
        android:layout_marginRight="16dp"
        android:layout_marginTop="16dp"
        android:text="@string/moodvalue"
        android:textSize="16sp"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <TextView
        android:id="@+id/eventDesc"
        android:layout_width="wrap_content"
        android:layout_height="20dp"
        android:layout_marginLeft="16dp"
        android:layout_marginStart="16dp"
        android:layout_marginTop="8dp"
        android:textAppearance="?android:attr/textAppearanceSmall"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/dateTime"
        tools:text="eventDesc" />

</android.support.constraint.ConstraintLayout>
```

- **bindView()**

Diese Methode besorgt vom **Cursor** (also aus der Ergebnistabelle einer Datenbankabfrage) mit Methoden wie **getInt()**, **getLong()** oder **getString()** die Daten der aktuellen Zeile und übergibt sie an die Elemente des per **newView()** erstellten Layouts. Sie wird für *jedes* Listenelement aufgerufen.

Das vollständige AS-Projekt auf dem aktuellen Stand befindet sich im Ordner

**...\\BspUeb\\Content Provider\\EventDiary.4**

### **14.3 Loader und LoaderManager**

Durch das in Android 3.0 eingeführte **Loader-API** wird das asynchrone Laden von Daten unterstützt, sodass es leicht fällt, Datenbankzugriffe im UI-Thread zu vermeiden. Geschieht ein Datenbankzugriff über einen Content Provider einer fremden App, dann sollte auf jeden Fall eine asynchrone Lösung realisiert werden. Dabei leistet die Klasse **CursorLoader** wertvolle Dienste. Weil sie generell einen Content Provider als Kooperationspartner benötigt, kann es auch bei einer ausschließlich anwendungsinternen Datenbanknutzung sinnvoll sein, einen Content Provider einzuspannen. Im Unterschied zu der im Kapitel 13 und im bisherigen Verlauf von Kapitel 14 vorgestellten Datenbankzugriffstechnik bietet das Loader-API neben dem asynchronen Zugriff weitere Vorteile:

- Man muss sich nicht mehr um das lästige Schließen von **Cursor**-Objekten kümmern und kann auf die abgewerteten Methoden **startManagingCursor()** und **requery()** verzichten.
- Nach einem Konfigurationswechsel wird die Verbindung zu einem vorhandenen **Cursor**-Objekt automatisch wiederhergestellt, sodass die Datenbankabfrage nicht wiederholt werden muss.

Asynchrone Datenbankzugriffe per **Loader**-API sind auf *Abfragen* beschränkt. Sollen zeitaufwändige Schreiboperationen (von Typ **UPDATE**, **INSERT** oder **DELETE**) im Hintergrund ausgeführt werden, kann z.B. eine von **AsyncTask** abgeleitete Klasse eingesetzt werden.

#### **14.3.1 Wichtige Typen im Loader-API**

Wenn eine Aktivität oder ein Fragment Datenbankzugriffe per Loader-API vornehmen soll, dann sind die folgenden Klassen bzw. Schnittstellen beteiligt:

- **Loader<D>**

Für das asynchrone Laden von Daten sind Ableitungen der generischen Basisklasse **Loader<D>** zuständig. Von einer **Loader<Cursor>** - Instanz wird erwartet, die Daten aufzubewahren, wenn die zugehörige Aktivität bzw. das zugehörige Fragment gestoppt wird, so dass die Daten *nicht* neu geladen werden müssen, wenn der Anwender zurückkehrt.

- **AsyncTaskLoader<Cursor>**

In der Regel wird nicht die Klasse **Loader<Cursor>** beerbt, sondern ihre Ableitung **AsyncTaskLoader<Cursor>**. Bei dieser Klasse kommt für das asynchrone Laden der Daten ein Objekt der Klasse **AsyncTask<Params, Progress, Result>** zum Einsatz, die in Abschnitt 10.4 vorgestellt wurde.

- **CursorLoader**

Diese im Zusammenhang mit dem Loader-API in der Regel verwendete Ableitung der Klasse **AsyncTaskLoader<Cursor>** kooperiert mit dem **ContentResolver** einer Anwendung, taugt also zum Zugriff auf die von einem **ContentProvider** verwalteten Daten. Aus einer erfolgreichen Operation resultiert ein **Cursor**-Objekt. Soll kein **ContentProvider**, sondern eine alternative Datenquelle im Loader-API verwendet werden, muss eine eigene **Loader<D>** - Ableitung erstellt werden.

- **LoaderManager**

Pro Aktivität bzw. Fragment ist *ein* Objekt aus einer Ableitung der abstrakten Klasse **LoaderManager** dafür zuständig, eine oder mehrere **Loader**-Instanz(en) zu verwalten.

- **LoaderManager.LoaderCallbacks<Cursor>**

In dieser Schnittstelle sind die von einer Aktivität bzw. von einem Fragment geforderten Methoden zur Kooperation mit einem **LoaderManager** definiert. Um eine neue Loader-Instanz zu erstellen, ist z.B. die Callback-Methode **onCreateLoader()** zu implementieren.

Die Klassen **Loader**, **CursorLoader** und **LoaderManager** sollten aus der Support-Library importiert werden:

```
import android.support.v4.content.Loader;
import android.support.v4.content.CursorLoader;
import android.support.v4.app.LoaderManager;
```

### 14.3.2 LoaderManager.LoaderCallbacks<Cursor> - Methoden implementieren

Zur Kommunikation zwischen einem Klienten (Aktivität oder Fragment) und seinem **LoaderManager** dienen die Methoden **onCreateLoader()**, **onLoadFinished()** und **onLoaderReset()** der Schnittstelle **LoaderManager.LoaderCallbacks<Cursor>**, die vom **LoaderManager** zu passenden Gelegenheiten aufgerufen werden. Um in der **EventListActivity** der Ereignistagebuchanwendung das Loader-API zu verwenden, kündigen wir im Klassendefinitionskopf das Implementieren der **LoaderCallbacks**-Methoden an:

```
public class EventListActivity extends AppCompatActivity
    implements AdapterView.OnItemClickListener, LoaderManager.LoaderCallbacks<Cursor> {
    . . .
}
```

Ist beim Aufruf der **LoaderManager**-Methode **initLoader()** (siehe Abschnitt 14.3.3) noch kein **Loader<Cursor>** - Objekt vorhanden, ruft der **LoaderManager** die Methode **onCreateLoader()** auf. Unsere Überschreibung liefert ein **CursorLoader** - Objekt, das als Konstruktor-Parameter alle für einen Aufruf der **ContentResolver**-Methode **query()** erforderlichen Argumente erhält (vgl. z.B. Abschnitt 14.1.3):<sup>1</sup>

---

<sup>1</sup> Das Android Studio hat bei der Erstellung von Methodenrümpfen zur Einlösung der Schnittstellen-Verpflichtungen zwei Annotationen verwendet, die uns im Kurs noch nicht begegnet sind:

- **@NonNull**

Durch die Annotation der gesamten Methode **onCreateLoader()** mit **@NonNull** wird zum Ausdruck gebracht, dass bei einem Aufruf die Rückgabe **null** nicht akzeptabel ist, dass der Programmierer also auf jeden Fall ein **Loader<Cursor>** - Objekt liefern muss.

- **@Nullable**

Durch die Annotation des **Bundle**-Parameters der Methode **onCreateLoader()** mit **@Nullable** wird zum Ausdruck gebracht, dass hier der Aktualparameterwert **null** akzeptabel ist, dass der Programmierer also damit rechnen muss.

```

@NonNull
@Override
public Loader<Cursor> onCreateLoader(int id, @Nullable Bundle args) {
    return new CursorLoader(this,
        EventProvider.CONTENT_URI, null, null, null,
        EventProvider.COL_DATETIME + " DESC");
}

```

Weil bei unserer Anwendung nur *eine* Loader-Kennung im Spiel ist, müssen wir den ersten **onCreateLoader()** - Parameter nicht beachten, und in dem als zweiter Parameter übergebenen **Bundle**-Objekt sind keine relevanten Argumente für die **CursorLoader**-Konstruktion zu erwarten.

Die im Loader enthaltene Abfrage wird vom Framework ausgeführt, sobald die Methode **onCreateLoader()** zurückkehrt (Becker & Pant 2015, S. 320).

Sobald ein **Loader<Cursor>** - Objekt das erfolgreiche Laden meldet, ruft der **LoaderManager** die Methode **onLoadFinished()** auf. In unserer Überschreibung wird daraufhin das in der **EventListActivity** tätige Adapterobjekt per **swapCursor()** beauftragt, den frisch erstellten Cursor zu verwenden:

```

@Override
public void onLoadFinished(@NonNull Loader<Cursor> loader, Cursor cursor) {
    adapter.swapCursor(cursor);
}

```

Durch den **swapCursor()** - Aufruf wird die Nutzung eines eventuell vorhandenen alten **Cursor**-Objekts beendet, so dass dieses Objekt vom Loader-Framework gelöscht werden kann.

Ist ein **Loader<Cursor>** - Objekt nicht mehr verfügbar, z.B. weil es über die **LoaderManager**-Methode **destroyLoader()** zerstört worden ist, ruft der **LoaderManager** die Methode **onLoaderReset()** auf. Hier sollte die Anwendung den Zugriff auf den **Cursor** des Loaders beenden:

```

@Override
public void onLoaderReset(@NonNull Loader<Cursor> loader) {
    adapter.swapCursor(null);
}

```

### 14.3.3 Loader-Initialisierung und -Aktualisierung

Wir stellen die **EventListActivity** auf die Verwendung des Loader-APIs um und verzichten in der Methode **onCreate()** darauf, über die **ContentResolver**-Methode **query()** ein **Cursor**-Objekt zu füllen und zur Verwaltung an die veraltete (abgewertete) **Activity**-Methode **startManagingCursor()** zu übergeben:

```

//     cursor = getContentResolver().query(EventProvider.CONTENT_URI,null,null,null,
//   EventProvider.COL_DATETIME + " DESC");
//     startManagingCursor(cursor);

```

Stattdessen erhält das **ListView**-Steuerelement in der **EventListActivity** ein Adapterobjekt *ohne* Cursor (mit **null** als Wert für den zweiten Konstruktoren-Parameter):

```

adapter = new EventDiaryAdapter(this, null);
lv.setAdapter(adapter);

```

In der vom **LoaderManager** aufgerufenen Methode **onLoadFinished()** wird der Adapter eine Referenz auf ein gebrauchsfertiges, in einem Hintergrund-Thread gefülltes **Cursor**-Objekt erhalten (siehe Abschnitt 14.3.2).

Der **LoaderManager** der Aktivität wird in **onCreate()** mit der Methode **initLoader()** dazu aufgefordert, für ein initialisiertes **Loader<Cursor>** - Objekt zu sorgen:

```
getSupportLoaderManager().initLoader(0, null, this);
```

Über den ersten Parameter erhält der Loader eine eindeutige Kennung. Weil unsere Anwendung nur *einen* Loader verwendet, spielt diese Kennung keine große Rolle. Als zweiter Parameter kann optional ein **Bundle**-Objekt mit Argumenten für die Konstruktion des Loaders übergeben werden, das an die **LoaderCallbacks<Cursor>** - Methode **onCreateLoader()** weitergereicht wird (siehe Abschnitt 14.3.2). Mit dem obligatorischen dritten Aktualparameter ist das Objekt zu benennen, das die **LoaderCallbacks<Cursor>** - Methoden ausführen soll. In unserem Beispiel übernimmt das Aktivitätsobjekt diesen Job.

Wenn beim Aufruf von **initLoader()** bereits ein Loader mit der Kennung im ersten Parameter existiert, wird dieses Objekt weiterverwendet. Ist noch *kein* Loader mit dieser Kennung vorhanden, wird die **LoaderCallbacks<Cursor>** - Methode **onCreateLoader()** aufgerufen.

So sieht die modernisierte Methode **onCreate()** - im Überblick aus:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_event_list);

    lv = findViewById(R.id.listView);
    lv.setOnItemClickListener(this);
    registerForContextMenu(lv);

    adapter = new EventDiaryAdapter(this, null);
    lv.setAdapter(adapter);
    getSupportLoaderManager().initLoader(0, null, this);
}
```

Wird das Loader-API in einem Fragment verwendet, eignet sich die Lebenszyklusmethode **onActivityCreated()** am besten dazu, die Loader-Initialisierung anzustoßen.

Ein **CursorLoader** bemerkt *nicht*, wenn sich der zuvor abgefragte Datenbestand geändert hat, kann aber mit der Methode **onContentChanged()** aufgefordert werden, sein **Cursor**-Objekt zu aktualisieren. Wir nutzen diese Möglichkeit in der **EventListActivity**-Methode **onContextItemSelected()**, um die Ereignisliste zu aktualisieren, nachdem ein Eintrag per Kontextmenü gelöscht worden ist:

```
int numDel = getContentResolver().delete(
    ContentUris.withAppendedId(EventProvider.CONTENT_URI, acmi.id), null, null);
if (numDel > 0)
    getSupportLoaderManager().getLoader(0).onContentChanged();
```

Analog ist dafür zu sorgen, dass die Cursor-Daten nach einer **UPDATE**-Operation aktualisiert werden. Wir starten in der **EventListActivity**-Methode **editEntry()** die für das Ändern von Ereignissen zuständige **EditEventActivity** mit der Methode **startActivityForResult()** und erstellen die CallBack-Methode **onActivityResult()**, die nach Durchführung einer **UPDATE**-Operation aufgerufen wird. Dort wird der **CursorLoader** mit der Methode **onContentChanged()** aufgefordert, seine Daten zu aktualisieren:

```
private final int REQUEST_CODE_EDIT = 1;

private void editEntry(long rowid) {
    Intent intent = new Intent(this, EditEventActivity.class);
    intent.putExtra("rowid", rowid);
    startActivityForResult(intent, REQUEST_CODE_EDIT);
}

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent intent) {
    if (requestCode == REQUEST_CODE_EDIT)
        getSupportFragmentManager().getLoader(0).onContentChanged();
}
```

Das vollständige AS-Projekt auf dem aktuellen Stand befindet sich im Ordner

**...\\BspUeb\\Content Provider\\EventDiary.5**

## 15 Einstellungen

Die Verwaltung von Einstellungen wird in Android durch das Preference-API unterstützt. Es bietet für Benutzer eine einheitliche Bedienoberfläche zur Bearbeitung von Einstellungen (inkl. Systemeinstellungen) nimmt dem Programmierer viele Routinearbeiten ab (z. B. Gestaltung der Bedienoberfläche, lesende und schreibende Dateizugriffe). Die Einstellungen landen als Name-Wert - Paare in XML-Dateien, die von Android verwaltet werden. Dort überdauern sie den aktuellen Anwendungsprozess und auch einen Systemneustart.

Neben dem eigentlichen Zweck der Einstellungsverwaltung taugt das Preference-API auch als Daten-Persistenzlösung mit den folgenden Einschränkungen:

- Es geht um eine begrenzte Datenmenge.
- Ausschließlich erlaubte Datentypen: **boolean, float, int, long, String** und **Set<String>**

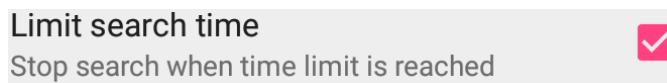
Wenn es etwa um eine Persistenzlösung für drei Zeichenfolgen und zwei Zahlen geht, ist das Preference-API deutlich handlicher als eine Datenbanklösung oder eine Dateiaus- und -eingabe.

### 15.1 Preferences-Ressourcendatei

Android verwendet zur Verwaltung einer Einstellung jeweils ein Objekt aus der Klasse **Preference** (im Namensraum **android.preference**) oder aus einer Ableitung. Häufig verwendete **Preference**-Ableitungen sind:

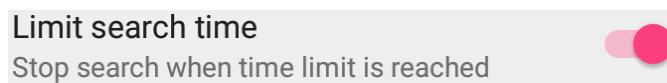
- **CheckBoxPreference**

Der Benutzer sieht ein Kontrollkästchen und legt mit seiner Wahl einen Wert mit Datentyp **boolean** fest, z. B.:



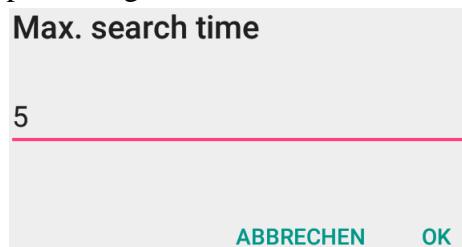
- **SwitchPreference**

Der Benutzer sieht einen Umschalter und legt mit seiner Wahl einen Wert mit Datentyp **boolean** fest, z. B.:



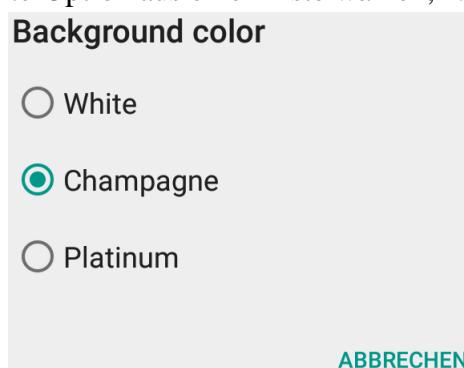
- **EditTextPreference**

Mit diesem Element wird eine Zeichenfolge erfasst, wobei die erlaubten Zeichen wie bei einem **EditText**-Steuerelement über das Attribut **android:inputType** festgelegt werden. Im folgenden Beispiel sind nur positive, ganze Zahlen erlaubt:



- **ListPreference**

Der Benutzer kann *genau eine* Option aus einer Liste wählen, z. B.:



Die Wahl wird als Zeichenfolge gespeichert.

- **MultiSelectListPreference**

Diese Klasse unterstützt eine Liste mit Mehrfachwahl.

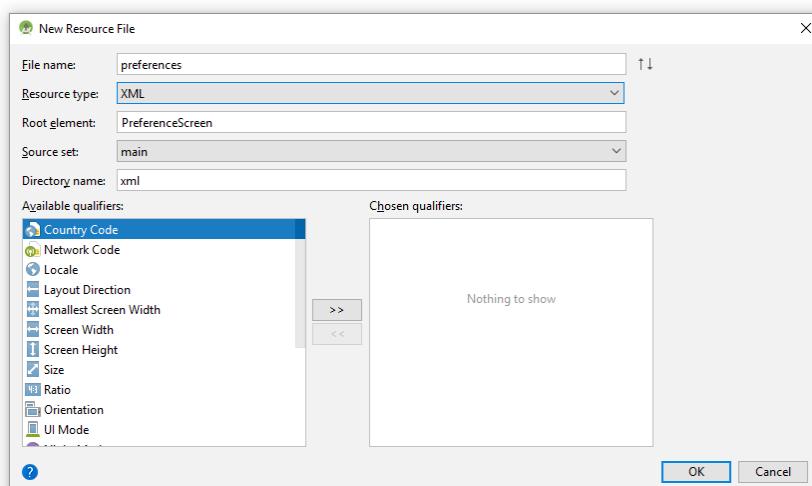
- **PreferenceCategory, PreferenceScreen**

Diese Klassen unterstützen die strukturierte Präsentation von Einstellungen. Man kann Einstellungen in Gruppen oder untergeordneten Einstellungsseiten zusammenfassen.

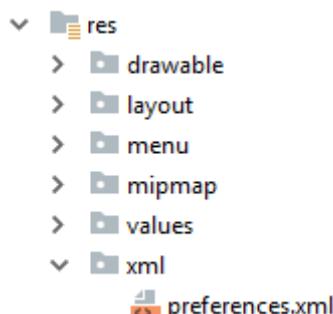
Zwar kann man Einstellungsobjekte per Programm erzeugen, doch ist in der Regel (wie bei **View**-Objekten) die Deklaration per XML-Datei zu bevorzugen. Dazu wählt man aus dem Kontextmenü zum Projektexplorerknoten **app/res** das Item

### New > Android Ressource File

und legt im folgenden Dialog für die neue Datei den gebräuchlichen (aber nicht vorgeschriebenen) Namen **preferences**, den Typ **XML**, das Wurzelement **PreferenceScreen** sowie den Ordnernamen **xml** fest:



Das Ergebnis im Projektexplorer:



Zur Gestaltung der Einstellungsseite kann man (wie bei einer Bedienoberfläche) entweder die XML-Datei direkt editieren oder einen grafischen Designer benutzen.

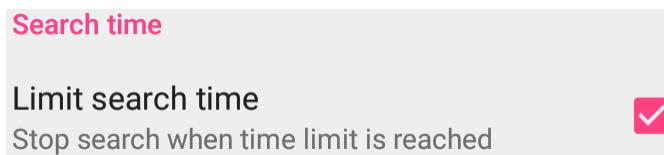
Wichtige Attribute für die **Preference**-Elemente:

- **android:key**

Jede Einstellung wird als Schlüssel-Wert - Paar gespeichert, und im Programm über den Schlüssel angesprochen (siehe Abschnitt 15.4).

- **android:title**

Mit diesem Attribut legt man fest, wie eine Einstellung, eine Kategorie oder eine untergeordnete Einstellungsseite in der Bedienoberfläche betitelt sein soll. Im folgenden Beispiel sind die Titel zu einer Kategorie (Search time) und zu einer Einstellung (Limit search time) zu sehen:



- **android:summary**

Über dieses Attribut ergänzt man den Titel durch eine ausführlichere Beschreibung (im Beispiel: Stop search when time limit is reached).

- **android:defaultValue**

Nach Möglichkeit sollten alle Einstellungen einen Voreinstellungswert erhalten.

Das folgende Beispiel realisiert drei Einstellungen für unser Beispielprogramm zur Primzahlendiagnose:

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">

    <PreferenceCategory android:title="@string/pref_catSearchTime">
        <CheckBoxPreference
            android:key="limitSearchTime"
            android:title="@string/pref_limitSearchTime"
            android:summary="@string/pref_limitSearchTimeSumm"
            android:defaultValue="false" />

        <EditTextPreference
            android:dependency="limitSearchTime"
            android:key="maxSearchTime"
            android:title="@string/pref_maxSearchTime"
            android:summary="@string/pref_maxSearchTimeSumm"
            android:inputType="number"
            android:defaultValue="5" />
    </PreferenceCategory>

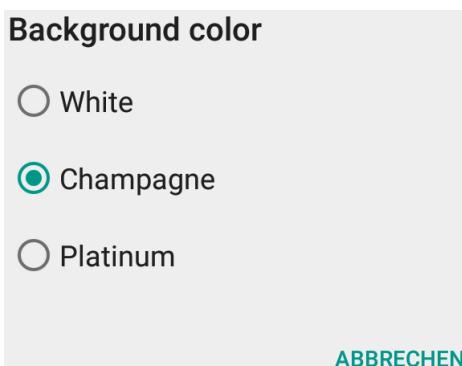
    <PreferenceCategory android:title="@string/pref_catBackgroundColor">
        <ListPreference
            android:key="bgColor"
            android:title="@string/pref_bgColor"
            android:summary="@string/pref_bgColorSumm"
            android:entries="@array/bgColor"
            android:entryValues="@array/bgColorValues"
            android:defaultValue="@string/bgColorDefault"/>
    </PreferenceCategory>
</PreferenceScreen>
```

Über das **CheckBoxPreference**-Element `limitSearchTime` können Benutzer den Primzahlendetektor veranlassen, die Prüfung eines Kandidaten nach Überschreitung einer maximalen Zeit abzubrechen.<sup>1</sup> Per Voreinstellung ist die Suchzeitbeschränkung inaktiv.

Nur bei aktiver Suchzeitbeschränkung (siehe Attribut **android:dependency**) ist die **EditTextPreference**-Einstellung `maxSearchTime` verfügbar. Über das Attribut **android:inputType** wird dafür gesorgt, dass die Benutzer nur eine positive ganze Zahl für die maximale Suchzeit in Sekunden eingeben können.

Für die beiden zusammengehörigen Einstellungen zur Suchzeitbeschränkung wurde der Übersichtlichkeit halber eine eigene Kategorie eingerichtet.

Über das **ListPreference**-Element `bgColor` können die Benutzer die Hintergrundfarbe des Programms bestimmen:



Den **ListPreference**-Attributen **android:entries** bzw. **android:entryValues** werden die für Menschen lesbaren Optionsnamen bzw. die dahinter stehenden Werte zugewiesen:

```
android:entries="@array/bgColor"
android:entryValues="@array/bgColorValues"
```

Im Beispiel geschieht dies unter Verwendung von Ressourcen vom Typ Array, die in der Datei **arrays.xml** folgendermaßen deklariert werden:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="bgColor">
        <item>White</item>
        <item>Champagne</item>
        <item>Platinum</item>
    </string-array>
    <string-array name="bgColorValues">
        <item>#FFFFFF</item>
        <item>#F7E7CE</item>
        <item>#E5E4E2</item>
    </string-array>
</resources>
```

Diese Ressourcendatei wird im Projektexplorerknoten **app/res/values** angelegt.

Weitere Details zur Einstellungs-Ressourcendeklaration sind auf Googles Webseite für Android-Entwickler zu finden.<sup>2</sup>

---

<sup>1</sup> Das Programm so zu erweitern, dass es die Suchzeitbeschränkung tatsächlich beachtet, bleibt dem Leser überlassen.

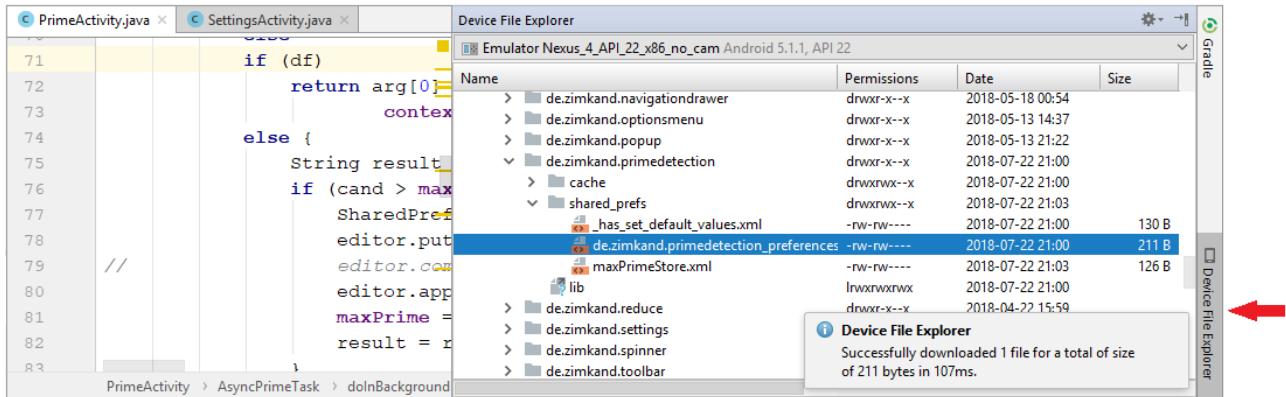
<sup>2</sup> <https://developer.android.com/guide/topics/ui/settings>

## 15.2 Einstellungsdateien einer App

Alle von einer App per Bedienoberfläche angebotenen oder ausschließlich per Programm verwendeten Einstellungsdaten werden in XML-Dateien abgelegt. Mit dem Platzhalter *package* für den Paketnamen der App befinden sich die Einstellungsdateien im folgenden Ordner des Android bzw. Linux-Dateisystems:

**/data/data/package/shared\_prefs**

Mit dem **Device File Explorer** (zu aktivieren über eine Schaltfläche am rechten Fensterrand) kann man das Dateisystem eines virtuellen Android-Gerätes bequem inspizieren und sogar Dateien zwischen dem Entwicklungsrechner und dem Android-Gerät austauschen:<sup>1</sup>



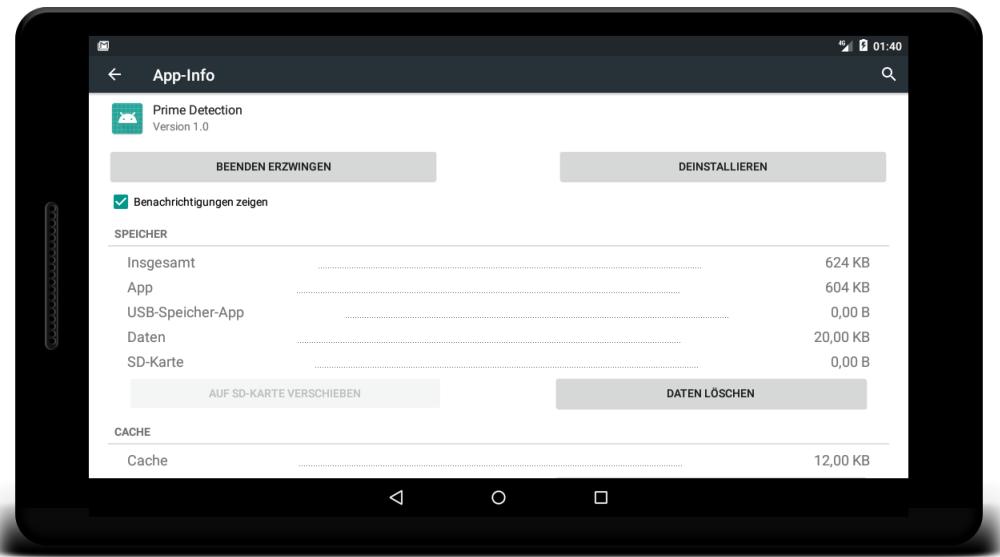
Die als Beispiel dienende neue Version des Primzahlendetektors verwendet *zwei* Einstellungsdateien:

- **de.zimkand.primedetection\_preferences.xml**  
Hier landen die Einstellungen, die Benutzer per Bedienoberfläche einsehen und verändern können (siehe Abschnitt 15.3). Diese basieren auf der in Abschnitt 15.1 beschriebenen Einstellungsdatei.
- **maxPrimeStore.xml**  
Hier speichert das Programm (ohne intentionale Beteiligung des Benutzers) die größte bisher entdeckte Primzahl (siehe Abschnitt 15.4).

Um diese Dateien mit den Mitteln des Android-Betriebssystems wieder los zu werden, kann ein Benutzer auf der App-Seite der Systemeinstellungen ...

<sup>1</sup> Reale Android-Geräte gewähren aus verständlichen Gründen weniger großzügige Einblicke in ihr Dateisystem. Hier lässt sich z.B. der Ordner **data**, für den wir uns momentan interessieren, *nicht* öffnen.

- die **Daten löschen**



- oder die Anwendung **deinstallieren**

### 15.3 Preferences-Bedienoberfläche

#### 15.3.1 Einstellungsaktivität bzw. -fragment

Das Preference-API ist primär dazu konzipiert, die in einer Ressourcen-Datei (gelegentlich auch in *mehreren* Ressourcen-Dateien) definierten Einstellungen über eine standardisierte Bedienoberfläche für Benutzer zugänglich zu machen. Vom Benutzer vorgenommene Änderungen werden automatisch in einer XML-formatierten Datei im **shared\_prefs** - Ordner der App gespeichert (siehe Abschnitt 15.2).

Setzt eine App mindestens die Android-Version 3 (das API-Level 11) voraus, sollte nach einer Empfehlung von Google zur Verwaltung der Einstellungen die von **Fragment** abgeleitete Klasse **PreferenceFragment** verwendet werden.<sup>1</sup> Weil unser Beispielprogramm das API-Level 15 voraussetzt, folgen wir dieser Empfehlung. Zusätzlich benötigen wir eine Activity als Rahmen für das Einstellungsfragment (vgl. Abschnitt 12.4). Um die Anzahl der Quellcodedateien in Grenzen zu halten, definieren wir die **PreferenceFragment**-Ableitung als innere Klasse der Aktivität:

```
package de.zimkand.primedetection;

import android.os.Bundle;
...
import android.preference.PreferenceManager;

public class SettingsActivity extends AppCompatActivity {

    public static class SettingsFragment extends PreferenceFragment {
        @Override
        public void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            addPreferencesFromResource(R.xml.preferences);
        }
    }
}
```

<sup>1</sup> <https://developer.android.com/guide/topics/ui/settings>

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    getFragmentManager().beginTransaction()
        .replace(android.R.id.content, new SettingsFragment())
        .commit();
}
}

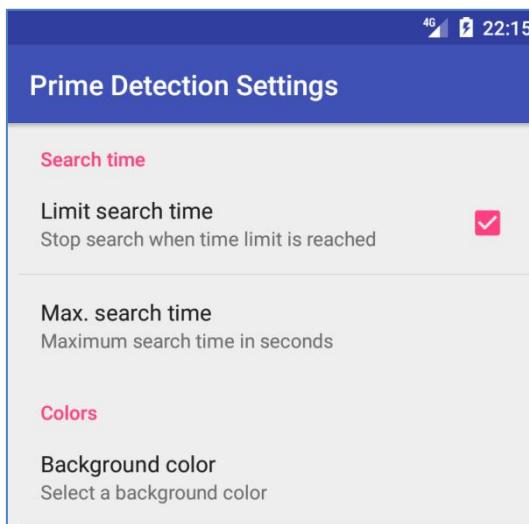
```

Weil die **PreferenceFragment**-Ableitung die Modifikatoren **public** und **static** besitzt, benimmt sie sich wie eine öffentliche Top-Level - Klasse, was für die Fragment-Transaktion erforderlich ist.

In den beiden Klassen **SettingsActivity** und **SettingsFragment** müssen wir lediglich die Methode **onCreate()** überschreiben. Die **PreferenceFragment**-Ableitung muss sich *nicht* um ihre **View**-Hierarchie kümmern, sondern lediglich die Methode **addPreferencesFromResource()** aufrufen:

```
addPreferencesFromResource(R.xml.preferences);
```

Aus einer Einstellungs-Deklarationsdatei entsteht so mit sehr wenig Programmieraufwand eine Bedienoberfläche zur Verwaltung der Einstellungen:



Ist die Datei

**package\_preferences.xml**

mit den Benutzereinstellungen beim Start der *Einstellungaktivität* noch nicht vorhanden, wird sie unter Verwendung der Voreinstellungswerte aus der Preferences-Ressourcendatei (siehe Abschnitt 15.1) angelegt.

In der Regel ist es sinnvoll, die Datei **package\_preferences.xml** bereits *vorher* explizit anzulegen. Dazu ruft man (z. B. in der **onCreate()** - Methode der *Startaktivität*) die statische Methode **setDefaults()** der Klasse **PreferenceManager** auf, z. B.:

```
PreferenceManager.setDefaults(this, R.xml.preferences, false);
```

Im ersten Parameter gibt man den Anwendungskontext an und im zweiten Parameter den Namen der Preferences-Ressourcendatei. Der **boolean**-Parameter an Position drei entscheidet darüber, ob die Initialisierung wiederholt ausgeführt werden soll (Wert **true**) oder nur einmalig (Wert **false**). In der Regel wird der Wert **false** gewählt. Eine bereits erfolgte Initialisierung erkennt Android an der Existenz der Datei **\_has\_set\_default\_values.xml** im Ordner **/data/data/package/shared\_prefs** (siehe Bildschirmfoto in Abschnitt 15.2).

Eine erneute Ausführung der Methode **setDefaultValues()** eignet sich übrigens *nicht* dazu, die Vor-einstellungswerte zu restaurieren. Es werden lediglich fehlende Einstellungen ergänzt, vorhandene behalten jedoch ihren aktuellen Wert. Um die Voreinstellungswerte zu restaurieren, ...

- verschafft man sich eine Referenz zum Editieren der voreingestellten Einstellungsdatei (vgl. Abschnitt 15.4.2),
- löscht mit **clear()** und **apply()** die Einstellungen
- und ruft die Methode **setDefaultValues()** mit dem Wert **true** für den dritten Parameter auf.

Es folgt ein Beispiel für die komplette Prozedur (, die im Beispielprogramm nicht enthalten ist):<sup>1</sup>

```
SharedPreferences pref = PreferenceManager.getDefaultSharedPreferences(this);
SharedPreferences.Editor editor = pref.edit();
editor.clear();
editor.apply();
PreferenceManager.setDefaultValues(this, R.xml.preferences, true);
```

Wirft man mit dem in Abschnitt 15.4.1 beschriebenen Verfahren einen Blick in die von Android angelegte Datei mit den Einstellungen eines Benutzers, dann erhält man für das Beispielprogramm nach dem Einschalten der Suchzeitenbegrenzung das folgende Bild:

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
<string name="maxSearchTime">5</string>
<string name="bgColor">#FFFFFF</string>
<boolean name="limitSearchTime" value="true" />
</map>
```

Insbesondere zeigt sich, dass ...

- für Einstellungen aus den Klassen **EditTextPreference** und **ListPreference** eine Zeichenfolge abgespeichert wird,
- für eine Einstellung aus der Klasse **CheckBoxPreference** der Datentyp **boolean** verwendet wird.

Während die Einstellungen zur Suchzeitbeschränkung im Beispielprogramm derzeit noch folgenlos bleiben, wird die vom Benutzer gewünschte Hintergrundfarbe tatsächlich realisiert. Dies geschieht in der Methode **onResume()**, die z. B. nach Rückkehr von der Einstellungsaktivität zur Hauptaktivität des Programms abläuft:

```
private SharedPreferences spDefaultPrefs;
...
spDefaultPrefs = PreferenceManager.getDefaultSharedPreferences(this);
...
@Override
protected void onResume() {
    super.onResume();
    View root = ((ViewGroup) findViewById(android.R.id.content)).getChildAt(0);
    String s = spDefaultPrefs.getString(BG_COLOR, DEF_BG_COLOR);
    root.setBackgroundColor(Color.parseColor(s));
}
```

Nach Ermittlung des Wurzel-Containers der Activity erfolgt ein lesender Zugriff auf die Einstellungsdatei mit der in Abschnitt 15.4 zu erläuternden **SharedPreferences** - Methode **getString()**.

---

<sup>1</sup> Zur Klärung der nicht ganz präzisen Beschreibung auf der Google-Seite  
<https://developer.android.com/reference/android/preference/PreferenceManager.html>  
 hat die folgende Stack Overflow - Seite beigetragen (Beitrag von Nermeen):  
<https://stackoverflow.com/questions/13762248/reset-default-values-of-preference>

Aus der gewonnenen Zeichenfolge erstellt die statische **Color**-Methode **parseColor()** einen **int**-Wert, der als Parameter für die **View**-Methode **setBackgroundColor()** dient.

### 15.3.2 Einstellungsaktivität via Optionsmenü starten

Wir müssen noch dafür sorgen, dass die Benutzer unseres Programms die Einstellungsaktivität starten können und realisieren dazu ein Optionsmenü (siehe Abschnitt 8.6). Lässt man das Android Studio in der Version 3.x ein neues Projekt mit einer **Basic Activity** anlegen, resultiert im Projektexplorerknoten **app/res/men** per Voreinstellung die folgende XML-Datei **menu\_main.xml** zur Deklaration eines Optionsmenüs, das unserem momentanen Bedarf gut entspricht:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context="de.zimkand.primedetection.PrimeActivity">
    <item
        android:id="@+id/action_settings"
        android:title="@string/action_settings"
        android:orderInCategory="100"
        app:showAsAction="never" />
</menu>
```

In der Aktivitätsmethode **onOptionsItemSelected()** zur Behandlung einer Menüitem-Auswahl (vgl. Abschnitt 8.6.1.5) starten wir bei passender Item-ID die Einstellungsaktivität über ein explizites **Intent**-Objekt (vgl. Abschnitt 9.3):

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.action_settings:
            Intent settIntent = new Intent(this, SettingsActivity.class);
            startActivity(settIntent);
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

## 15.4 Einstellungsdaten per Programm schreiben und lesen

### 15.4.1 Die Klasse Sharedpreferences

Unabhängig von bzw. ergänzend zu der oben beschriebenen Inspektion und Modifikation von Einstellungen durch den Benutzer lassen sich Einstellungsdateien per Programm bearbeiten, was uns eine sehr bequeme Möglichkeit zur persistenten Verwaltung von kleineren Datenmengen liefert.

Beim Zugriff auf Einstellungsdaten per Programm spielt die Klasse **Sharedpreferences** eine wichtige Rolle. Jede Einstellungsdatei wird von einem eindeutig bestimmten Objekt der Klasse **Sharedpreferences** betreut, das den lesenden und den schreibenden Zugriff auf die Einstellungsdatei ermöglicht.

Von der **Context**-Methode **getSharedPreferences()** erhalten wir das **Sharedpreferences**-Objekt zur Einstellungsdatei mit dem im ersten Parameter angegebenen Namen:

```
public Sharedpreferences getSharedPreferences(String name, int mode)
```

Im folgenden Beispiel wird die Aktivität angesprochen und der Dateiname über eine finalisierte Instanzvariable festgelegt:

```
private final String MAX_PRIME_STORE = "maxPrimeStore";
private SharedPreferences spMaxPrime, spDefaultPrefs;
...
spMaxPrime = getSharedPreferences(MAX_PRIME_STORE, MODE_PRIVATE);
```

Wenn noch keine Datei mit dem angegebenen Namen existiert, wird sie beim ersten Schreibzugriff erstellt.

Im zweiten **getSharedPreferences()** - Parameter ist ein Modus anzugeben, wobei ausschließlich der Wert **MODE\_PRIVATE** (int-Wert 0) sinnvoll ist: Er reserviert den Zugriff für die eigene App (oder für eine App mit derselben User-ID). Alle anderen technisch möglichen Ausprägungen sind als *abgewertet* gekennzeichnet.

Von der statischen Methode **getDefaultSharedPreferences()** der Klasse **PreferenceManager**

```
public static SharedPreferences getDefaultSharedPreferences(Context context)
```

erhalten wir ein **SharedPreferences**-Objekt zur Einstellungsdatei mit den über eine XML-Datei deklarierten (vgl. Abschnitt 15.1) und über die **PreferenceFragment**-Methode **addPreferencesFromResource()** (vgl. Abschnitt 15.3.1) ins Spiel gebrachten Einstellungen. Diese Datei trägt einen fest vorgegebenen Namen, der aus dem Paketnamen durch Anhängen von **\_preferences.xml** entsteht, was z. B. beim Paketnamen **de.zimkand.primedetection** zum folgenden Ergebnis führt:

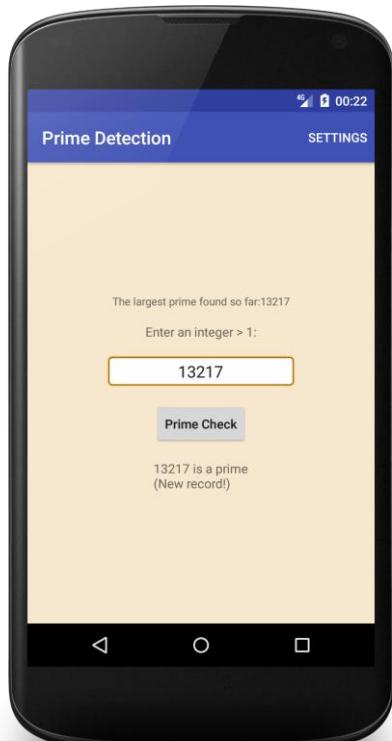
**de.zimkand.primedetection\_preferences.xml**

Im folgenden Beispiel wird der Kontext von der Activity geliefert:

```
private SharedPreferences spMaxPrime, spDefaultPrefs;
...
spDefaultPrefs = PreferenceManager.getDefaultSharedPreferences(this);
```

#### 15.4.2 Einstellungsdaten schreiben

Wir erweitern den Primzahlendetektor um eine sportliche Note und merken uns die maximale bisher gefundene Primzahl, z. B.:



Übertrifft eine erkannte Primzahl die größte bisher gefundene, wird das neue Maximum mit Hilfe des **SharedPreferences** - Objekts `spMaxPrime` in die verbundene Einstellungsdatei geschrieben. Dazu wird mit der **SharedPreferences** - Methode `edit()` ein Objekt aus der inneren Klasse **SharedPreferences.Editor** engagiert und beauftragt, das neue Maximum (in der Variablen `cand`) mit der zum Datentyp passenden Methode `putLong()` als Wert zum Schlüssel `maxPrime` in die Einstellungsdatei zu schreiben:

```
private final String MAX_PRIME = "maxPrime";  
.  
.  
.  
SharedPreferences.Editor editor = spMaxPrime.edit();  
editor.putLong(MAX_PRIME, cand);  
editor.apply();
```

Mit der Methode `apply()` veranlasst man den Editor, die vorbereiteten Schreiboperationen auszuführen. Um zeitaufwändige, den UI-Thread störende Dateizugriffe zu vermeiden, wird zunächst nur das **SharedPreferences** - Objekt im Arbeitsspeicher geändert, während Dateizugriffe asynchron erfolgen (z. B. im Zusammenhang mit Lebenszyklusmethoden). Ein potentieller Nachteil dieser Arbeitsweise besteht darin, dass Fehler bei der asynchronen Sicherung nicht an den Initiator gemeldet werden. Im Unterschied zu `apply()` bewirkt die ältere, immer noch gebräuchliche Methode `commit()` ein sofortiges Schreiben in die Einstellungsdatei.

Wirft man mit dem in Abschnitt 15.4.1 beschriebenen Verfahren einen Blick in die Einstellungsdatei, zeigt sich im Beispiel der folgende Inhalt:

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>  
<map>  
<long name="maxPrime" value="13217" />  
</map>
```

### 15.4.3 Einstellungsdaten lesen

Um aus einer Einstellungsdatei den Wert zu einem Schlüssel zu lesen, besorgen wir uns eine Referenz auf das zuständige **SharedPreferences**-Objekt (siehe Abschnitt 15.4.1)

```
spMaxPrime = getSharedPreferences(MAX_PRIME_STORE, MODE_PRIVATE);
```

und verwenden die zum Datentyp des Schlüssels passende `get`-Methode. In der folgenden Anweisung wird auf die „High score“ - Datei des Beispielprogramms zugegriffen:

```
maxPrime = spMaxPrime.getLong(MAX_PRIME, 0);
```

Im ersten Parameter mit Datentyp **String** ist der Schlüssel anzugeben, was im Beispiel durch eine finalisierte Instanzvariable geschieht. Im zweiten Parameter ist ein Ersatzwert anzugeben für den Fall, dass kein aktueller Wert zu ermitteln ist, weil die Einstellungsdatei nicht existiert oder keinen Schlüssel mit dem angegebenen Namen enthält.

Ein Lesezugriff des Beispielprogramms auf die Datei **de.zimkand.primedetection\_preferences.xml** war schon in Abschnitt 15.3.1 zu sehen:

```
String s = spDefaultPrefs.getString(BG_COLOR, DEF_BG_COLOR);
```

Das komplette AS-Projekt mit dem aktuellen Entwicklungsstand des Primzahlendetektors ist im folgenden Ordner zu finden:

...\\BspUeb\\Preferences

Der Einfachheit halber wird *nicht* die mit einem Dienst arbeitende Variante der Primzahlendiagnose verwendet (siehe Kapitel 11), sondern die Variante mit einem **AsyncTask**-Objekt im Rahmen der primären Aktivität (siehe Abschnitt 10.4).

### 15.5 Sonstige Optionen bei der Einstellungsverwaltung

Anuzzi et al. (2014, Kapitel 11) erwähnen bzw. beschreiben weitere Optionen zur Einstellungsverwaltung, z. B.:

- Beobachter bei einem **SharedPreferences**-Objekt registrieren  
Über die **SharedPreferences**-Methode **registerOnSharedPreferenceChangeListener()** kann man ein Objekt aus einer Klasse, die das Interface **SharedPreferences.OnSharedPreferenceChangeListener** implementiert, als Beobachter registrieren lassen. Dessen Methode **onSharedPreferenceChanged()** wird aufgerufen, sobald sich eine vom **SharedPreferences**-Objekt verwaltet Einstellung ändert (siehe Anuzzi et al. 2014, S. 285).
- Sichern in der Google-Cloud  
Ein als *Cloud Save* bezeichneter Dienst der Firma Google macht es möglich, Einstellungen in der Cloud zu sichern, sodass sie auf allen Android-Geräten eines Benutzers verfügbar sind (siehe Anuzzi et al. 2014, S. 296).

---

## 16 Broadcast Receiver

Mit dem Broadcast Receiver lernen wir in diesem Kapitel eine weitere Komponente von Android-Apps kennen. Objekte einer aus **BroadcastReceiver** (im Paket **android.content**) abgeleiteten Klasse lassen sich als Empfänger von bestimmten **Intent**-Objekten registrieren, die vom System oder von App-Komponenten (Aktivitäten oder Diensten) versandt („ausgestrahlt“) werden (z. B. mit der Methode **sendBroadcast()**). Somit spielen Broadcast Receiver eine wichtige Rolle bei der Kommunikation über Prozessgrenzen hinweg. Aber auch bei der Kommunikation innerhalb einer App kommen Broadcast Receiver zum Einsatz, was schon in Abschnitt 11.2.5 zu sehen war.

Das Betriebssystem informiert z. B. per Rundruf über folgende Ereignisse:

- Der Neustart des Systems ist abgeschlossen.
- Das WLAN wurde (de-)aktiviert.
- Eine externe Stromversorgung wurde angeschlossen bzw. getrennt.

Beim Eintreffen eines passenden Broadcast-Intents wird die **onReceive()** - Methode eines registrierten Empfängers aufgerufen.

Die Registrierung eines Broadcast Receivers kann statisch über ein **receiver**-Element im Anwendungsmanifest erfolgen oder dynamisch über die **Context**-Methode **registerReceiver()**. Seit der Android-Version 8 (API-Level 26) ist allerdings die Möglichkeit zur statischen Registrierung eines Receivers für implizite Broadcast-Intents nur noch bei wenigen, vom System stammenden Nachrichten erlaubt bzw. wirksam.

Broadcast Receiver haben keine eigene Bedienoberfläche, können sich aber doch bemerkbar machen:

- über die Benachrichtigungsleiste am oberen Bildschirmrand (im Manuscript nicht behandelt)
- über Toast-Meldungen (siehe z. B. Seite 216)

In beschränkten zeitlichen Umfang (von ca. 10 Sekunden) dürfen Broadcast Receiver aufgrund der per Intent übermittelten Informationen Datenverarbeitungen im Hintergrund ausführen.

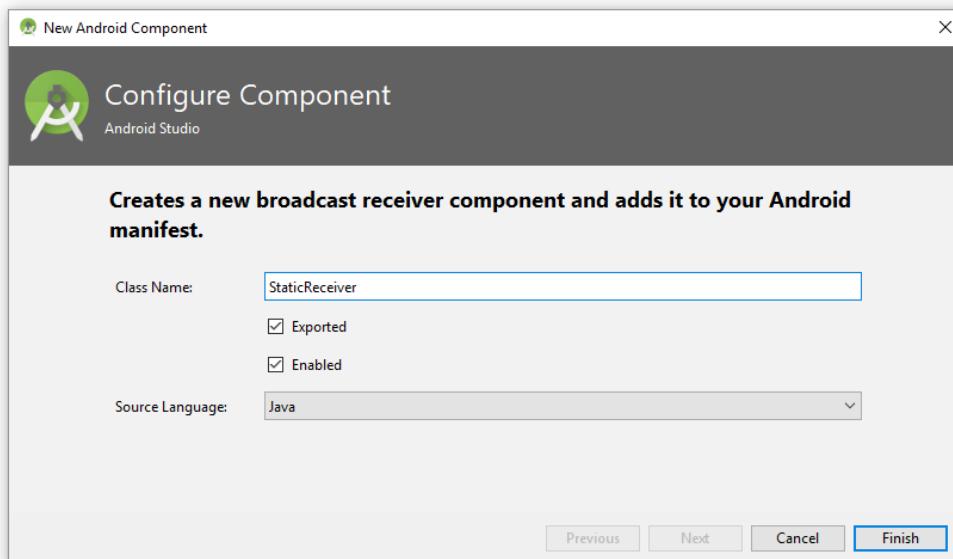
Wir haben Intents schon mehrfach als kommunikative Hilfsmittel kennengelernt, z. B. beim Starten von Aktivitäten. Die verschiedenen mit Intents arbeitenden Android-Systeme sind aber streng getrennt, sodass z. B. ein beim Aufruf der Methode **startActivity()** verwendetes **Intent**-Objekt nicht an einen Broadcast Receiver übermittelt wird.

### 16.1 Registrierung per Manifest

Wir legen ein neues AS-Projekt mit einer leeren Aktivität an, und wählen anschließend aus dem Kontextmenü zum Paketeintrag im Projektexplorer den Befehl:

**New > Other > Broadcast Receiver**

Es erscheint ein Assistent mit einem einzigen Fenster, das den Namen der neu anzulegenden Klasse erfragt:



Der Assistent liefert eine rudimentäre Klassendefinition für den Broadcast Receiver

```
package de.zimkand.brdemo;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;

public class StaticReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        throw new UnsupportedOperationException("Not yet implemented");
    }
}
```

und nimmt eine statische Registrierung des Receivers in der Manifestdatei vor:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="de.zimkand.brdemo">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <receiver
            android:name=".StaticReceiver"
            android:enabled="true"
            android:exported="true"></receiver>
    </application>
</manifest>
```

Unser Receiver stammt von der Klasse **BroadcastReceiver** im Paket **android.content** ab und muss die abstrakt definierte Basisklassenmethode **onReceive()** überschreiben. Wir beschränken uns da-

rauf, in einer Toast-Nachricht die per `getAction()` zu erfahrende Aktion des empfangenen **Intent**-Objekts anzuseigen:

```
@Override
public void onReceive(Context context, Intent intent) {
    Toast.makeText(context, intent.getAction(), Toast.LENGTH_LONG).show();
}
```

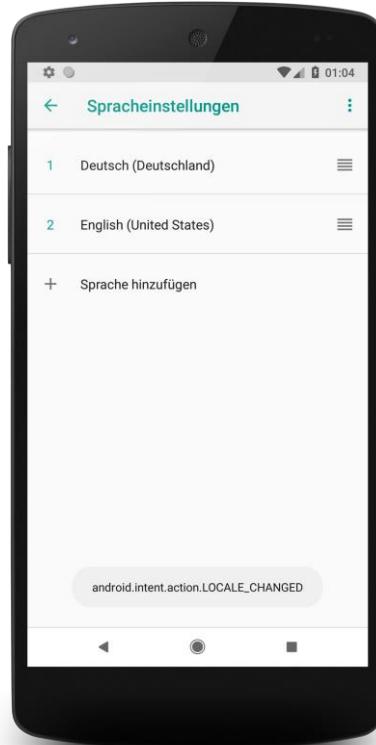
Im **receiver**-Element der Manifestdatei fügen wir einen **intent-filter** (vgl. Abschnitt 9.2) mit passendem **action**-Element ein, um den Empfänger für Systemnachrichten über den Wechsel der Benutzersprache zu registrieren:

```
<receiver
    android:name=".StaticReceiver"
    android:enabled="true"
    android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.LOCALE_CHANGED"/>
    </intent-filter>
</receiver>
```

Enthält ein **receiver**-Element keinen Intent-Filter, dann kann der zugehörige Receiver nur durch einen expliziten Broadcast-Intent (mit Angabe der ausführenden Komponente, ohne Aktionsbeschreibung) angesprochen werden.

Wird die **onReceive()** - Methode aufgrund einer *statischen* Registrierung aufgerufen, dann wird nach ihrer Beendigung das **BroadcastReceiver**-Objekt verworfen. In dieser Lage sollte kein Hintergrund-Thread mit Ergebnisrückmeldung aus **onReceive()** gestartet werden, weil seine Ergebnisse mit großer Wahrscheinlichkeit verloren gehen.

Das folgende Bildschirmfoto zeigt eine Toast-Anzeige durch den aktiven Broadcast-Receiver des Beispielprogramms auf einem virtuellen Gerät mit Android 8.1 (API-Level 27):



Um einen statisch registrierten Broadcast Receiver aus einer momentan nicht aktiven App über ein Ereignis zu informieren, muss ein Prozess zur App kreiert werden. Dies verursacht einen relevanten Zeit- und Energieaufwand, der sich zudem oft nicht lohnt. Daher sind die Android-Designer seit

einiger Zeit bestrebt, den durch statisch registrierte Receiver verursachten Aufwand zu reduzieren. Mit dem Ziel, die Akku-Laufzeit zu verlängern, ist es seit Android 7 nicht mehr möglich, für Broadcast-Intents mit den folgenden Aktionen einen Receiver statisch zu registrieren:

**ACTION\_NEW\_PICTURE, ACTION\_NEW\_PICTURE, ACTION\_NEW\_VIDEO**

Seit Android 8 (API-Level 26) ist die statische Registrierung von Receivern für implizite Broadcast-Intents fast komplett wirkungslos. Von der Funktionseinschränkung sind lediglich Systemnachrichten ausgenommen, von den z.B. im API-Level 27 definierten 191 Systemnachrichten aber bei weitem nicht alle, sondern nur ca. 30 sorgfältig ausgewählte Nachrichten.<sup>1</sup>

Von den genannten Beschränkungen sind per Voreinstellung nur Apps mit einer **targetSdkVersion** ab 24 bzw. 26 betroffen. Allerdings bietet die Einstellungs-App von Android 8 dem Benutzer die Möglichkeit, die Restriktionen für beliebige Apps zu aktivieren.

Als Ersatzlösungen für das statischen Registrieren von Broadcast Receivern stehen zur Verfügung:

- die dynamische Registrierung (siehe Abschnitt 16.3)
- bedingungsabhängige geplante Aufgaben (im Manuskript noch nicht enthalten)<sup>2</sup>

## 16.2 Broadcast-Intents verarbeiten

### 16.2.1 Startschwierigkeiten

Reagiert ein korrekt entwickelter Receiver *nicht* auf passende Broadcast Intents, dann kommen vor allem die folgenden Ursachen in Frage:

- Seine Anwendung befindet sich im gestoppten Zustand. Um diese Ursache werden wir uns gleich kümmern.
- Die installierte Android-Version ab API-Level 26 ignoriert die statische Receiver-Registrierung (siehe Abschnitt 16.1).
- Fehlende Rechte (siehe Abschnitt 16.6)

Wenn sich eine App im **gestoppten Zustand** befindet, dann können ihre Broadcast Receiver nur Intents empfangen, die mit dem **FLAG\_INCLUDE\_STOPPED\_PACKAGES** versendet worden sind (vgl. Abschnitt 16.4). Ein Programm befindet sich im gestoppten Zustand:

- Unmittelbar nach der Installation, vor dem ersten Starten durch den Benutzer
- Nachdem das Programm über **Einstellungen > Apps > Konkrete App > Beenden erzwingen** gestoppt worden ist.

Ein Programm verlässt den gestoppten Zustand, wenn es einmal vom Benutzer gestartet wird. Beachten Sie den gravierenden Unterschied zwischen einer gestoppten *Anwendung* und einer *Aktivität* im gestoppten Zustand (siehe Kapitel 6).

---

<sup>1</sup> Eine Liste mit den Aktionsbezeichnungen zu den Android-Rundrufnachrichten befindet sich im Android-SDK - Ordner, für das API-Level 27 z.B. in der Datei:

`...\Sdk\platforms\android-27\data\broadcast_actions.txt`

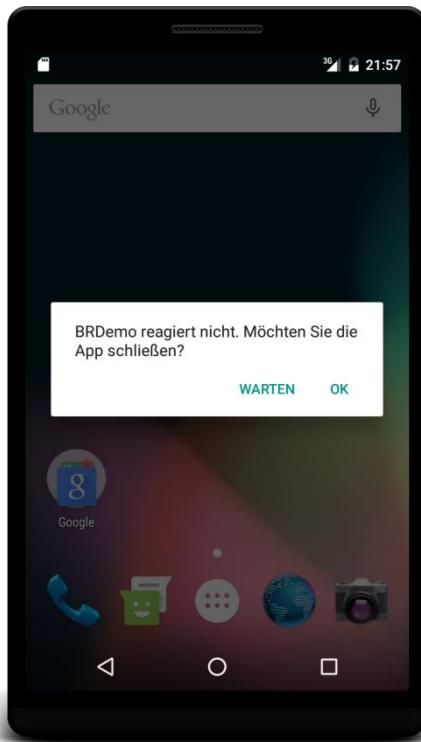
Auf der folgenden Webseite sind die Broadcast-Ausnahmen dokumentiert, für die auch in Android 8 noch Receiver statisch registriert werden dürfen (z.B. ACTION\_BOOT\_COMPLETED, ACTION\_LOCALE\_CHANGED):

<https://developer.android.com/guide/components/broadcast-exceptions>

<sup>2</sup> <https://developer.android.com/topic/performance/scheduling>

### 16.2.2 Zeitlich begrenzte Arbeitserlaubnis

Die `onReceive()` - Methode eines Broadcast Receivers sollte maximal 10 Sekunden lang aktiv bleiben, weil sonst mit einem Einschreiten des Betriebssystems zu rechnen ist.<sup>1</sup> Bei Tests mit einem virtuellen Smartphone unter Android 6.0 hat das Betriebssystem allerdings erst nach ca. 60 Sekunden reagiert. Es erschien eine ANR-Fehlermeldung, wenn eine Aktivität derselben App im Vordergrund war, oder wenn über die Android-Entwickleroptionen angeordnet worden war, dass ANR-Fehlermeldungen auch für Hintergrund-Apps erscheinen sollen:



Andernfalls wurde der Anwendungsprozess nach ca. 60 Sekunden zerstört. Wenn eine im UI-Thread laufende `onReceive()` - Methode eine Vordergrundaktivität daran hindert, auf Benutzerwünsche zu reagieren, ist bereits nach 5 Sekunden mit einer ANR-Fehlermeldung zu rechnen.

### 16.2.3 Akzeptiertes Verhalten

Ein Broadcast Receiver besitzt keine Bedienoberfläche und darf auch keine Dialogboxen öffnen (siehe Abschnitt 16.2.5); er kann sich aber per Benachrichtigungszeile oder durch Toast-Einblendungen bemerkbar machen.

Auf keinen Fall sollte ein Broadcast Receiver eine Aktivität starten.<sup>2</sup> Man stelle sich die verblüffte bis verärgerte Reaktion des Benutzers vor, wenn aufgrund eines Broadcast-Intents mehrere registrierte Empfänger jeweils eine Aktivität starten würden.

Zur Ausführung von Hintergrundtätigkeiten, die aus dem UI-Thread herausgehalten werden müssen, bieten sich einem Broadcast Receiver einige Optionen an (siehe Abschnitt 16.2.6).

<sup>1</sup> <https://developer.android.com/reference/android/content/BroadcastReceiver.html#ProcessLifecycle>

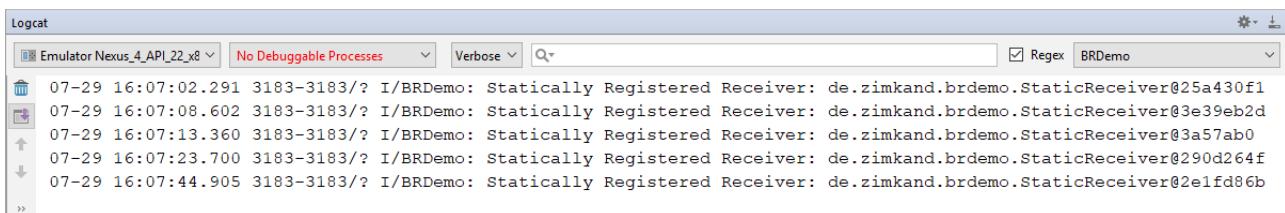
<sup>2</sup> <https://developer.android.com/guide/components/broadcasts>

### 16.2.4 Lebenslauf

Nach Beendigung der Methode `onReceive()` wird ein Objekt eines statisch registrierten Receivers als ungültig betrachtet. Zur Demonstration erweitern wir die `onReceive()` - Methode des Beispiels um eine **Logcat**-Ausgabe, aus der sich die Identität des handelnden Objekts entnehmen lässt:

```
@Override
public void onReceive(Context context, Intent intent) {
    Log.i("BRDemo", "Statically Registered Receiver: " + this.toString());
    Toast.makeText(context, intent.getAction(), Toast.LENGTH_LONG).show();
}
```

Wie das folgende Protokoll zeigt, wird jeder `onReceive()` - Aufrufe durch ein anderes, jeweils von Android neu erzeugtes **BroadcastReceiver**-Objekt ausgeführt, sodass die Aufbewahrung von Zuständen in Instanzvariablen sinnlos wäre:



Bei einem *dynamisch* registrierten Receiver (siehe Abschnitt 16.3) werden hingegen mehrere `onReceive()` - Aufrufe vom *selben* Objekt ausgeführt, sofern die registrierende Aktivität nicht zwischenzeitlich zerstört und neu erstellt worden ist.

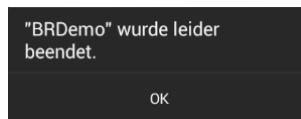
Weil die Objekte eines statisch registrierten Receivers nach Ablauf der `onReceive()` - Methode als ungültig betrachtet werden, hat der zugehörige Prozess in dieser Situation eine sehr kleine Priorität und dementsprechend ein relativ hohes Risiko, bei Speichermangel eliminiert zu werden (vgl. Abschnitt 5.4), wenn keine weiteren Anwendungskomponenten vorhanden sind.

### 16.2.5 Verbot von Dialogen

In der `onReceive()` - Methode eines statisch registrierten Receivers darf kein Dialog (z. B. aus der Klasse `AlertDialog`) gestartet werden. Bei der Ausführung der folgenden Methode

```
@Override
public void onReceive(Context context, Intent intent) {
    AlertDialog.Builder builder = new AlertDialog.Builder(context);
    builder.setMessage(intent.getAction());
    AlertDialog dialog = builder.create();
    dialog.show();
}
```

kommt es zu einem Laufzeitfehler:



### 16.2.6 Aufgaben asynchron im Hintergrund ausführen

Die `onReceive()` - Methode wird im Haupt-Thread gestartet und muss aufwändige Bearbeitungen in einen separaten Thread verlagern. Wie Sie bereits wissen, sollte aus der `onReceive()` - Methode eines *statisch* registrierten Receivers kein Hintergrund-Thread mit Ergebnisrückmeldung aufgerufen werden, weil der Receiver nach Ablauf von `onReceive()` verworfen wird, und daher die Ergebnisse des Threads mit großer Wahrscheinlichkeit verloren gehen. Seit dem API-Level 11 kann ein statisch registrierter Receiver jedoch aus `onReceive()` eine Hintergrundoperation veranlassen und *nach* Be-

endigung von **onReceive()** auf das Ergebnis der Hintergrundverarbeitung reagieren. Es bleibt jedoch dabei, dass für einen Broadcast-Intent nur 10 Sekunden Bearbeitungszeit garantiert sind.

Man ruft in **onReceive()** die **BroadcastReceiver**-Methode **goAsync()** auf und erhält als Rückgabe ein Objekt der Klasse **BroadcastReceiver.PendingResult**, z. B.:

```
final BroadcastReceiver.PendingResult result = goAsync();
```

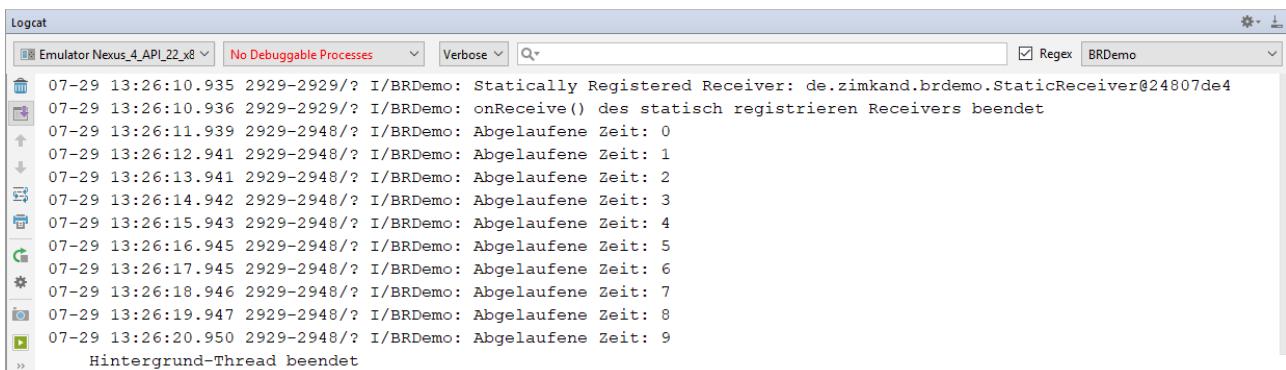
Innerhalb der garantierten Lebensdauer bleibt das Receiver-Objekt über das Ende der Methode **onReceive()** hinaus so lange erhalten, bis die **finish()** - Methode des **PendingResult**-Objekts aufgerufen wird.

Wenn der folgendermaßen definierte **BroadcastReceiver**

```
public class StaticReceiver extends BroadcastReceiver {
    private void done() {
        Log.i("BRDemo", "Hintergrund-Thread beendet");
    }

    @Override
    public void onReceive(Context context, Intent intent) {
        Log.i("BRDemo", "Statically Registered Receiver: " + this.toString());
        final BroadcastReceiver.PendingResult result = goAsync();
        new Thread() {
            @Override
            public void run() {
                for (int i = 0; i < 10; i++)
                    try {
                        Thread.sleep(1000);
                        Log.i("BRDemo", "Abgelaufene Zeit: " + Integer.toString(i));
                    } catch (Exception ex) {}
                done();
                result.finish();
            }
        }.start();
        Log.i("BRDemo", "onReceive() des statisch registrierten Receivers beendet");
    }
}
```

aktiv wird, entstehen im geeignet gefilterten **Logcat**-Fenster der Entwicklungsumgebung die folgenden Protokolleinträge:<sup>1</sup>



Wenn die für einen Broadcast Receiver garantierte Arbeitszeit von 10 Sekunden eventuell nicht ausreicht, und länger dauernde Aktionen initiiert werden sollen, bieten sich folgende Lösungen an:

<sup>1</sup> In der letzten Protokollzeile fehlen aus unbekannten Gründen einige technische Details.

- Aus **onReceive()** kann mit **peekService()** ein bereits laufender Dienst angesprochen werden. Es ist jedoch nicht sinnvoll, ...
  - mit **startService()** einen Dienst neu zu starten
  - oder mit **bindService()** einen Dienst einzubinden.
- Ab Android 5 (API-Level 21) steht für asynchron auszuführende Aufgaben die Klasse **JobScheduler** bereit, die in vielen Situationen einen Broadcast Receiver ersetzen kann.<sup>1</sup> Für ältere Android-Versionen empfiehlt Google als Kompatibilitätsebene die Open Source - Lösung **Firebase JobDispatcher**.<sup>2</sup>

### 16.3 Dynamische Registrierung

Im Unterschied zu den übrigen Komponenten einer Android-App (Activity, Service, Content Provider) muss ein Broadcast Receiver nicht unbedingt im Anwendungsmanifest registriert werden. Dies kann auch per Programm über die **Context**-Methode **registerReceiver()** geschehen. Mögliche Gründe für die dynamische Registrierung sind:

- Ein Receiver soll nur dann ansprechbar sein, wenn sich eine Aktivität im Vordergrund befindet.
- Für einige Broadcast-Intents des Betriebssystems sind seit jeher ausschließlich dynamisch registrierte Receiver erlaubt, um den Akku zu schonen. Ein Beispiel ist das einmal pro Minute gefeuerte Ereignis **Intent.ACTION\_TIME\_TICK**. Wie in Abschnitt 16.1 zu erfahren war, ist seit der Android-Version 8 (API-Level 26) die statische Registrierung von Broadcast Receivern für implizite Broadcast-Intents fast komplett unwirksam.

Besonderheiten eines dynamisch registrierten Receivers im Vergleich zum statisch (per Anwendungsmanifest) registrierten Receiver sind:

- Der Receiver persistiert über das Ende eines **onReceive()** - Methodenaufrufs hinaus. Er verarbeitet also in der Regel mehrere Signale und kann zwischen seinen Auftritten Zustände in Instanzvariablen aufbewahren.
- Beim statisch registrierten Receiver ist das Betriebssystem für den Lebenszyklus eines Objekts verantwortlich. Es wird automatisch erzeugt und nach Ablauf seiner **onReceive()** - Methode wieder verworfen. Demgegenüber muss ein Objekt eines dynamisch registrierten Receivers per Programm erstellt, mit der **Context**-Methode **registerReceiver()** aktiviert und mit der **Context**-Methode **unregisterReceiver()** wieder deaktiviert werden. Ist eine Activity der Initiator, wird die Receiver-Registrierung z.B. in der Lebenszyklusmethode **onResume()** vorgenommen und in der Methode **onPause()** wieder aufgehoben.
- Aus der **onReceive()** - Methode eines dynamisch registrierten Receivers lassen sich Dialoge (z. B. aus der Klasse **AlertDialog**) starten.
- Während Android bei einem statisch registrierten Receiver einschreitet, wenn die Verarbeitung eines Broadcast-Intents nicht in der erlaubten Zeitspanne (nominell 10, empirisch 60 Sekunden) endet, wird dasselbe Verhalten bei einem dynamisch registrierten Receiver toleriert (beobachtet bei virtuellen Smartphone unter Android 6.0). Dabei spielt es keine Rolle, ob während der **onReceive()** - Ausführung eine Aktivität der eigenen App im Vordergrund ist oder nicht.

Allmählich wird es Zeit, die dynamische Registrierung eines Broadcast Receivers durch eine Aktivität konkret zu demonstrieren. Wir verwenden eine **BroadcastReceiver**-Ableitung, die sich nur unwesentlich von der in Abschnitt 16.1 verwendeten Variante unterscheidet:

<sup>1</sup> <https://developer.android.com/topic/performance/scheduling>

<sup>2</sup> <https://github.com/firebase/firebase-jobdispatcher-android#user-content-firebase-jobdispatcher->

```

package de.zimkand.brdemo;

import android.content.BroadcastReceiver+
. . .

public class DynamicReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        Toast.makeText(context, intent.getAction(), Toast.LENGTH_LONG).show();
    }
}

```

Die Hauptaktivität der App erzeugt in ihrer Lebenszyklusmethode **onCreate()** ein Objekt der Klasse **DynamicReceiver** sowie einen Intent-Filter zur Systemnachricht über das Ende des Ladestroms:

```

dynamicReceiver = new DynamicReceiver();
intentFilter = new IntentFilter(Intent.ACTION_POWER_DISCONNECTED);

```

Wenn die Aktivität in den Vordergrund gelangt (also in der Methode **onResume()**) bzw. den Platz an der Sonne verliert (also in der Methode **onPause()**), wird die Registrierung des Receivers vorgenommen bzw. aufgehoben:

```

package de.zimkand.brdemo;

import android.content.IntentFilter;
. . .

public class MainActivity extends AppCompatActivity {
    private DynamicReceiver dynamicReceiver;
    private IntentFilter intentFilter;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        dynamicReceiver = new DynamicReceiver();
        intentFilter = new IntentFilter(Intent.ACTION_POWER_DISCONNECTED);
    }

    @Override
    protected void onResume() {
        super.onResume();
        registerReceiver(dynamicReceiver, intentFilter);
    }

    @Override
    protected void onPause() {
        super.onPause();
        unregisterReceiver(dynamicReceiver);
    }
}

```

Wenn bei fehlendem **unregisterReceiver()** - Aufruf in **onPause()** die Hauptaktivität zerstört wird (z. B. aufgrund einer Rückwärtsnavigation des Benutzers), dann beschwert sich Android über ein Speicherleck, weil eine dynamische Broadcast Receiver - Registrierung vorgenommen, aber nicht aufgehoben wurde, z. B.:

```

07-29 16:52:13.865 2835-2835/de.zimkand.brdemo E/ActivityThread: Activity
de.zimkand.brdemo.MainActivity has leaked IntentReceiver
de.zimkand.brdemo.DynamicReceiver@35ff2c27 that was originally registered here. Are
you missing a call to unregisterReceiver()?

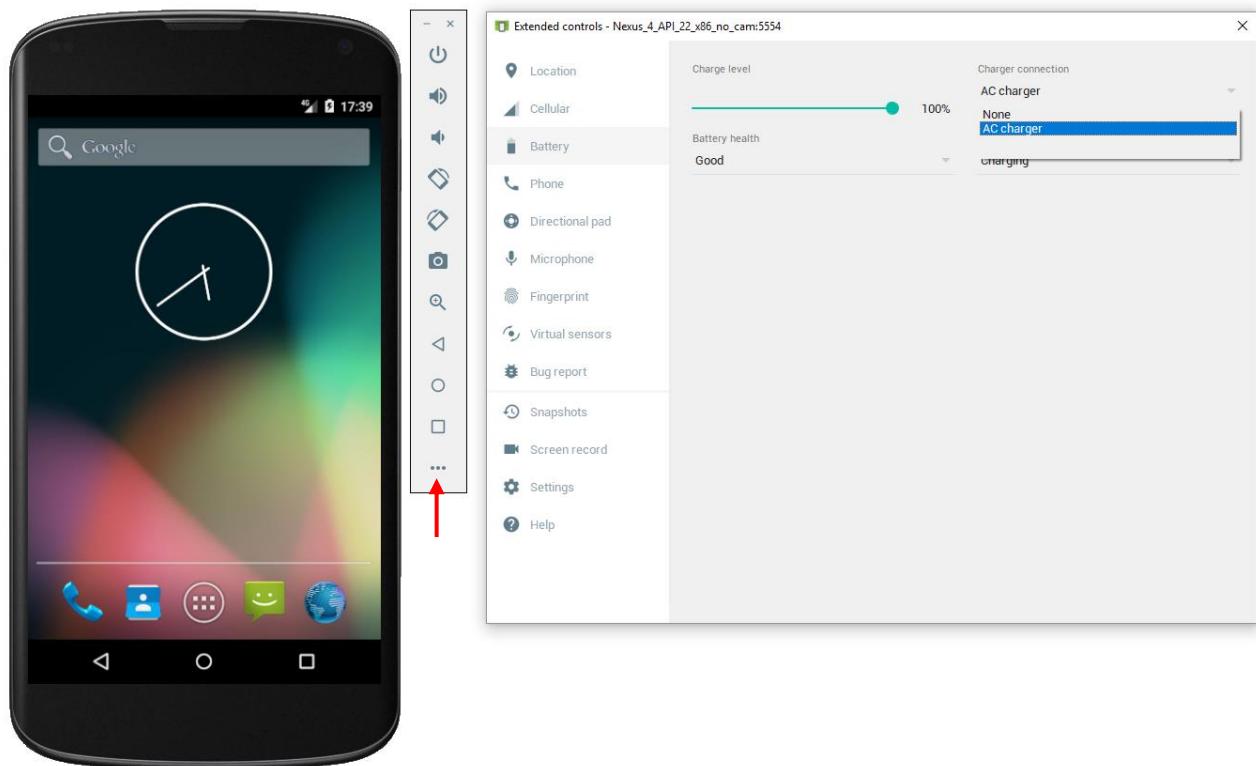
```

Der vergessene Broadcast Receiver hält die verworfene Aktivität im Speicher, bis irgendwann der gesamte Prozess eliminiert wird.

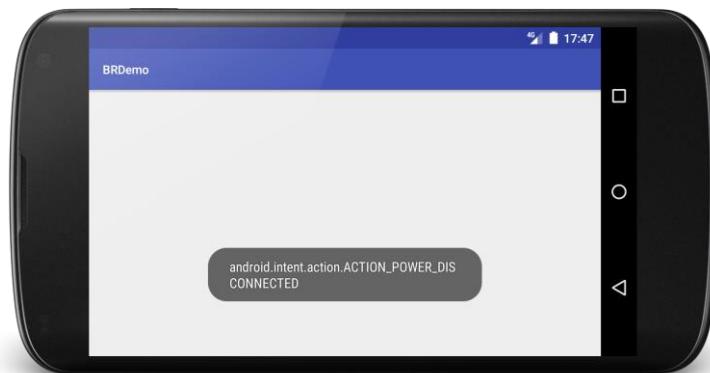
Nimmt eine Aktivität eine dynamische Receiver-Registrierung in ihrer Lebenszyklusmethode `onStart()` bzw. `onCreate()` vor, dann ist die „Gegenmethode“ `onStop()` bzw. `onDestroy()` der geeignete Ort, um die Registrierung wieder aufzuheben.

Wenn ein Service (vgl. Kapitel 11) einen Broadcast Receiver dynamisch registriert, muss diese Registrierung ebenfalls in einer geeigneten Lebenszyklus-Methode (z. B. `stopSelf()` oder `stopService()` bei einem gestarteten Service) aufgehoben werden.

Im Android-Emulator lässt sich übrigens über die Symbolpalette ein Fenster mit erweiterten Bedienelementen anfordern, das u.a. die Möglichkeit bietet, die Aufnahme bzw. Beendigung der Verbindung zu einer Stromquelle zu simulieren:



Auf diese Weise können wir mit einem emulierten Gerät testen, ob unser dynamisch registrierter Broadcast Receiver die Systemnachricht über das Ende des Ladestroms erhält und verarbeitet, z.B.:



#### 16.4 Broadcast-Nachrichten versenden

Broadcast-Intents stammen größtenteils vom System, doch können sich auch Apps am Broadcast-basierten Kommunikationsverfahren beteiligen, was in diesem Abschnitt demonstriert werden soll.

Zum Versenden einer Nachricht im normalen Modus dienen die beiden folgenden Überladungen der **Context-Methode sendBroadcast()**.

```
public void sendBroadcast(Intent intent)
public void sendBroadcast(Intent intent, String receiverPermission)
```

Neben dem **Intent**-Objekt im obligatorischen ersten Parameter kann optional im zweiten Parameter der Name einer Berechtigung angegeben werden, die ein *Empfänger* besitzen muss, um die Nachricht zu erhalten (siehe Abschnitt 16.6).

In der folgenden Klickbehandlungsmethode zu einem Befehlsschalter wird ein Broadcast-Intent verschickt, dessen Aktion über den eingetretenen Feierabend informiert:

```
@Override
public void onClick(View v) {
    sendBroadcast(new Intent("de.zimkand.intent.action.FINITO"));
}
```

Diverse installierte Apps mögen auf diese Nachricht reagieren (z. B. den E-Mail-Abruf von bestimmten Konten abschalten).

Per Broadcast für die anwendungsübergreifende Kommunikation versendete **Intent**-Objekte sind meist vom *impliziten* Typ (siehe Kapitel 9). Grundsätzlich ist es aber auch möglich, mit einem expliziten **Intent**-Objekt per „Broadcast“ einen *einzigsten*, über seinen Paket- und Klassennamen identifizierten und statisch registrierten Receiver anzusprechen. Im folgenden Beispiel werden das Paket (die Anwendung) und die Receiver-Klasse über ein Objekt der Klasse **ComponentName** und die Methode **setComponent()** festgelegt (vgl. Abschnitt 9.1.1):

```
Intent intent = new Intent();
String paket = "de.zimkand.brdemo";
String klasse = "de.zimkand.brdemo.StaticReceiver";
intent.setComponent(new ComponentName(paket, klasse));
sendBroadcast(intent);
```

Dass man den Paket- und den Klassennamen eines Receivers einer *fremden* App kennt, ist allerdings eher ungewöhnlich.

Ob ein Broadcast-Intent auch Apps im gestoppten Zustand (siehe Abschnitt 16.1) aktivieren soll, wird durch zwei Flags geregelt:

- **FLAG\_EXCLUDE\_STOPPED\_PACKAGES**  
Dieses von Android per Voreinstellung an alle Broadcast-Intents angeheftete Flag verhindert die Zustellung an gestoppte Anwendungen.
- **FLAG\_INCLUDE\_STOPPED\_PACKAGES**  
Die Zustellung an gestoppte Anwendungen wird erlaubt.

Sind beide Flags gesetzt, gewinnt das INCLUDE-Flag.

## 16.5 Broadcast-Varianten

Auch wenn das Emittieren von Rundrufen durch eigene Apps vorläufig nicht geplant ist, lohnt sich eine Beschäftigung mit den dabei verfügbaren Varianten, weil diese auch für Broadcast Receiver relevant sind.

### 16.5.1 (Un)geordnete Verarbeitung

Beim oben beschriebenen Versand mit der **Context-Methode sendBroadcast()** erfolgt die Zustellung an die registrierten Empfänger ohne definierte Reihenfolge (eventuell auch gleichzeitig). Außerdem kann ein Empfänger keine Informationen an Nachfolger weitergeben und die Auslieferungssequenz nicht stoppen.

Nach dem Versand mit der **Context**-Methode **sendOrderedBroadcast()** erfolgt die Zustellung hingegen sequentiell, wobei sich die Reihenfolge an den (z. B. per Manifest-Attribut **android:priority** festgelegten) Prioritäten der Receiver orientiert, z. B.:

```
<intent-filter android:priority="1">
    <action android:name="de.zimkand.intent.action.FINITO" />
</intent-filter>
```

Haben mehrere Receiver identische Prioritäten, dann entscheidet der Zufall.

Jeder Receiver kann ...

- über die Methode **isOrderedBroadcast()** feststellen, ob eine empfangene Nachricht per **sendOrderedBroadcast()** versandt worden ist,
- per **setResultCode()**, **setResultData()** und **setResultExtras()** Informationen an seinen Nachfolger weitergeben,
- per **getresultCode()**, **getresultData()** und **getresultExtras()** die von seinem Vorgänger übermittelten Informationen auswerten,
- die Verarbeitung per **abortBroadcast()** beenden.

Eine spezielle **sendOrderedBroadcast()** - Überladung macht es dem Sender möglich, einen eigenen Broadcast Receiver in die finale Position zu bringen, um die Verarbeitungsergebnisse der Receiver-Kette auswerten zu können.

### 16.5.2 Sticky Broadcasts

Ist ein dynamisch zu registrierender Receiver beim Versenden eines Broadcast-Intents „offline“, verpasst er die Nachricht. Das kann bei wichtigen Systemnachrichten durchaus problematisch sein. Zur Lösung des Problems bietet Android die Möglichkeit, durch den Versand eines Broadcast-Intents mit der Methode **sendStickyBroadcast()** (erforderliche Berechtigung:

**android.permission.BROADCAST\_STICKY**) den permanenten Verbleib im System anzugeben, sodass die Nachricht auch an später registrierte Receiver ausgeliefert wird.<sup>1</sup> Eine Sticky-Nachricht verschwindet, ...

- wenn sie durch eine Intent-Filter - identische (entschieden durch die **Intent**-Methode **filterEquals()**) ersetzt wird,
- wenn sie mit **removeStickyBroadcast()** zurückgezogen wird,
- beim Neustart des Systems.

Seit Android 5 (API-Level 21) ist die Methode **sendStickyBroadcast()** allerdings aus Sicherheitsgründen als **deprecated** gekennzeichnet, sodass sie in eigenen Apps nicht mehr verwendet werden sollte (siehe Müller 2015, S. 34).

Die Broadcast-Nachricht **Intent.ACTION\_BATTERY\_CHANGED** mit dem aktuellen Ladezustand des Akkus wird jedoch auch in Android 8.1 (API-Level 27) weiterhin vom System im Sticky-Modus versendet. Trotz der generellen Sicherheitsbedenken gegen die Methode **sendStickyBroadcast()** besteht hier kein Risiko, weil eine gewöhnliche App trotz der beanspruchten Berechtigung **android.permission.BROADCAST\_STICKY** daran gehindert wird, die spezielle Broadcast-Nachricht **Intent.ACTION\_BATTERY\_CHANGED** zu versenden:

```
java.lang.SecurityException: Permission Denial: not allowed to send broadcast
    android.intent.action.BATTERY_CHANGED
```

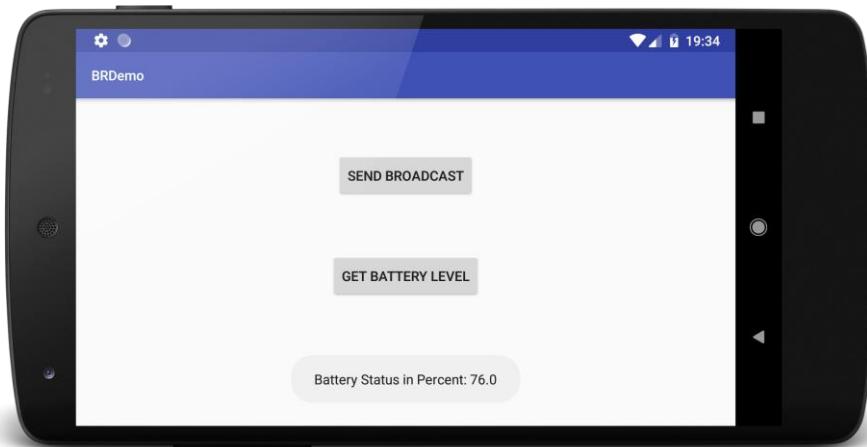
---

<sup>1</sup> Deutsche Übersetzung von *sticky*: klebrig

Ist ein Broadcast-Intent bekanntermaßen sticky (z. B. **Intent.ACTION\_BATTERY\_CHANGED**), dann kann man das zuletzt gesendete Exemplar als Rückgabe eines **registerReceiver()** - Aufrufs mit passendem **IntentFilter**-Parameter abrufen und darf dabei im ersten Parameter **null** statt eines **BroadcastReceiver**-Objekts angeben. Im folgenden Quellcodesegment wird diese Technik demonstriert:<sup>1</sup>

```
Intent batteryIntent = this.registerReceiver(null,
    new IntentFilter(Intent.ACTION_BATTERY_CHANGED));
int level = batteryIntent.getIntExtra(BatteryManager.EXTRA_LEVEL, -1);
int scale = batteryIntent.getIntExtra(BatteryManager.EXTRA_SCALE, -1);
float batteryPct = (level/(float)scale)*100;
Toast.makeText(this, "Battery Status in Percent: "+Float.toString(batteryPct),
    Toast.LENGTH_LONG).show();
```

Der prozentuale Ladezustand wird aus dem erhaltenen **Intent**-Objekt per **getIntExtra()** ermittelt und dann per **Toast** angezeigt:



Weitere Details zum Sticky-Broadcast finden sich in der Dokumentation zur Klasse **Context**.<sup>2</sup>

## 16.6 Sicherheit

Das Broadcast-System in Android dient hauptsächlich zur anwendungsübergreifenden Kommunikation. Doch zu viel Offenheit kann ein Sicherheitsrisiko darstellen, und Android bietet Mechanismen, den Kreis der zulässigen Kommunikationsteilnehmer zu beschränken.

### 16.6.1 Beschränkung der zulässigen Absender

Soll ein statisch deklarierter Broadcast Receiver ausschließlich für Komponenten der eigenen App ansprechbar sein, setzt man im **receiver**-Element der Manifestdatei das Attribut **android:exported** auf den Wert **false**, z. B.:

```
<receiver
    android:name=".StaticReceiver"
    android:enabled="true"
    android:exported="false"
    <intent-filter>
        <action android:name="de.zimkand.intent.action.FINITO" />
    </intent-filter>
</receiver>
```

Sind Intent-Filter vorhanden, hat das Attribut **android:exported** per Voreinstellung den Wert **true**, sodass die Aktivität auch von fremden Apps genutzt werden kann (siehe Abschnitt 9.2).

<sup>1</sup> Idee übernommen von <https://developer.android.com/training/monitoring-device-state/battery-monitoring.html>

<sup>2</sup> <https://developer.android.com/reference/android/content/Context.html>

Für Rundrufe, die nur zur anwendungsinternen Kommunikation dienen sollen, unterstützt Android über die Klasse **LocalBroadcastManager** eine spezielle Technik, die eine bessere Performanz bietet und keine Sicherheitsprobleme kennt (siehe Abschnitt 16.7).

Das Attribut **android:enabled** mit der Voreinstellung **true** legt fest, ob ein statisch registrierter Broadcast Receiver generell aktiviert ist, d.h. ob Android auf Anforderung Objekte des Receivers instanzieren soll. Diese Einstellung lässt sich mit Hilfe der Klasse **PackageManager** ändern, sodass man auch den statisch registrierten Receiver per Programm (de)aktivieren kann.

Statt einen Receiver für *alle* fremden Apps zu sperren, lässt sich über das Berechtigungssystem von Android der Zugriff differenzierter regeln. Man kann bei der Registrierung eines Broadcast Receivers festlegen, dass nur Absender im Besitz einer Berechtigung akzeptiert werden.

Bei einem statisch registrierten Receiver wird die geforderte Berechtigung im Anwendungsmanifest deklariert, z. B.:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="de.zimkand.brdemo">
    <permission android:name="de.zimkand.permission.BROADCAST" />
    <application . . . >
        <activity android:name=".MainActivity">
            . .
        </activity>
        <receiver android:name=".StaticReceiver" android:enabled="true"
            android:exported="true"
            android:permission="de.zimkand.permission.BROADCAST">
            <intent-filter>
                . .
            </intent-filter>
        </receiver>
    </application>
</manifest>
```

Das **permission**-Element ist erforderlich, um eine App-eigene Berechtigung zu definieren. Diese kann dann wie eine vorgegebene (z. B. systemseitige) Berechtigung im Attribut **android:permission** des **receiver**-Elements als schützende Barriere verwendet werden.

Eine Broadcast-Nachricht bleibt ohne Effekt auf den Empfänger, wenn der Absender (Klient) die erforderliche Berechtigung nicht in seinem Anwendungsmanifest per **uses-permission** - Element erfolgreich beantragt hat, z. B.:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="de.zimkand.explicitbroadcast">
    <uses-permission android:name="de.zimkand.permission.BROADCAST" />
    <application
        . .
    </application>
</manifest>
```

Auch bei der dynamischen Registrierung eines Receivers kann man dafür sorgen, dass nur Absender im Besitz einer Berechtigung akzeptiert werden. Dazu ist die folgende Überladung der Methode **registerReceiver()** zu verwenden:

```
public Intent registerReceiver (BroadcastReceiver receiver,
                               IntentFilter filter, String senderPermission, Handler scheduler)
```

Erfolgt der Schutz des Receivers über eine App-eigene Berechtigung, so ist diese im Anwendungsmanifest zu definieren (siehe oben). Über den letzten Parameter kann man dafür sorgen, dass der Receiver *nicht* im Haupt-Thread der Anwendung ausgeführt wird. Von dieser Möglichkeit wird im folgenden Beispiel (wie im gesamten Kurs) kein Gebrauch gemacht:

```

private DynamicReceiver dynamicReceiver
dynamicReceiver = new DynamicReceiver();
. . .
registerReceiver(dynamicReceiver,
    new IntentFilter("de.zimkand.intent.action.FINITO"),
    "de.zimkand.permission.BROADCAST", null);

```

Ist ein dynamisch registrierter Receiver durch eine Berechtigung geschützt, benötigt übrigens auch die eigene App ein **uses-permission** - Manifestelement für den Zugriff.

### 16.6.2 Beschränkung der zulässigen Empfänger

Per Voreinstellung ist eine Broadcast-Nachricht durch *jede* App zu empfangen. Seit Android 4 kann man mit der **Intent**-Methode **setPackage()** dafür sorgen, dass nur Receiver aus einer einzigen Anwendung (mit dem passenden **package**-Attribut im Manifest) durch einen Broadcast-Intent ange- sprochen werden, z. B.:

```

Intent intent = new Intent("de.zimkand.intent.action.FINITO");
intent.setPackage("de.zimkand.brdemo");
sendBroadcast(intent);

```

Eine weitere Möglichkeit, den Empfängerkreis für eine Nachricht einzuschränken, besteht darin, in der versendenden Methode (z. B. **sendBroadcast()** oder **sendOrderedBroadcast()**) einen **Permission**-Parameter anzugeben, z. B.:

```
sendBroadcast(intent, "de.zimkand.permission.BROADCAST");
```

Empfangsberechtigt sind nur Apps, welche die erforderliche Berechtigung über ein **uses- permission** - Element im Anwendungsmanifest erfolgreich beantragt haben (siehe Abschnitt 16.6.1).

Verwendet der Sender eine App-eigene Berechtigung, muss er diese in seiner Manifestdatei definie- ren, z. B.:

```

<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="de.zimkand.bcsender">
    <permission android:name="de.zimkand.permission.BROADCAST" />
    <application . . . >
        . . .
    </application>
</manifest>

```

## 16.7 Prozessinterne Broadcast-Nachrichten

Sind bei einer Broadcast-Kommunikation ausschließlich anwendungseigene Komponenten beteiligt, sollte die Klasse **LocalBroadcastManager** verwendet werden. Man muss dem Thema Sicherheit keine Aufmerksamkeit widmen und steigert die Performanz. Eine ausführliche Anwendung der an- wendungsinternen Kommunikation mit Hilfe der Klasse **LocalBroadcastManager** haben wir schon in Abschnitt 11.2.5 kennengelernt. Dort meldet ein Dienst seine Arbeitsergebnisse per Broadcast- Intent, und eine Aktivität derselben App zeigt diese Ergebnisse in ihrer Bedienoberfläche an. Ein Receiver für die anwendungslokale Broadcast-Kommunikation muss auf jeden Fall dynamisch re- gistriert werden.

---

## Literatur

- Allen, G. (2012). *Beginning Android 4*. New York, NY: Apress.
- Baltes-Götz, B. (2017). *Einführung in das Programmieren mit C# 6.0*. Online-Dokument: <https://www.uni-trier.de/index.php?id=22777>
- Baltes-Götz, B. & Götz, J. (2018). *Einführung in das Programmieren mit Java 9*. Online-Dokument: <https://www.uni-trier.de/index.php?id=22787>
- Becker, A. & Pant, M. (2015). *Android 5*. Heidelberg: dpunkt-Verlag.
- Felker, D. & Burton, M. (2013). *Android App Entwicklung für DUMMIES* (2. Aufl.). Weinheim: Wiley-VCH
- Garganta, M. & Nakamura, M. (2014). *Learning Android* (2<sup>nd</sup> ed.). Sebastopol, CA: O'Reilly.
- Göransson, A. (2014). *Efficient Android Threading*. Sebastopol, CA: O'Reilly
- Koller, D. (2012). *Android-Apps programmieren*. Haar: Franzis.
- Künneth, T. (2012). *Android 4. Apps entwickeln mit dem Android SDK* (2. Aufl.). Bonn: Galileo Press.
- Mednieks, Z., Dornin, L., Meike, G. B. & Nakamura, M. (2011). *Programming Android*. Sebastopol, CA: O'Reilly.
- Mednieks, Z., Dornin, L., Meike, G. B. & Nakamura, M. (2013). *Android Programmierung* (2. Aufl.). Köln: O'Reilly.
- Meike, G.B. (2016). *Android Concurrency*. Boston, MA: Pearson Education.
- Post, U. (2013). *Android-Apps entwickeln für Einsteiger* (3. Aufl.). Bonn: Galileo Press.

---

## Anhang

### A. Lösungsvorschläge zu den Übungsaufgaben

#### Kapitel 1 (Java und Android)

##### Aufgabe 1

1. Falsch

2. Falsch

Zur Programmierung von Android-Anwendungen kommt die reguläre Programmiersprache Java zum Einsatz. Allerdings hinkt der unter Android unterstützte Versionstand der Java-Entwicklung leicht hinterher.

3. Richtig

#### Kapitel 4 (Android Studio kennenlernen)

##### Aufgabe 1

Ein Lösungsvorschlag ist im folgenden Ordner zu finden:

...\\BspUeb\\Reduce\\Alert

#### Kapitel 5 (Komponenten und Systemumgebung einer App)

##### Aufgabe 1

1. Falsch

2. Falsch

Die Aktivitäten in einer Task können zu verschiedenen Apps gehören, und die Komponenten einer App werden stets in einem eigenen Prozess (und von einer eigenen virtuellen Maschine) ausgeführt.

3. Richtig

Alle Apps müssen mit demselben privaten Schlüssel signiert werden (siehe Abschnitt 4.8).

#### Kapitel 6 (Lebenszyklus einer Aktivität)

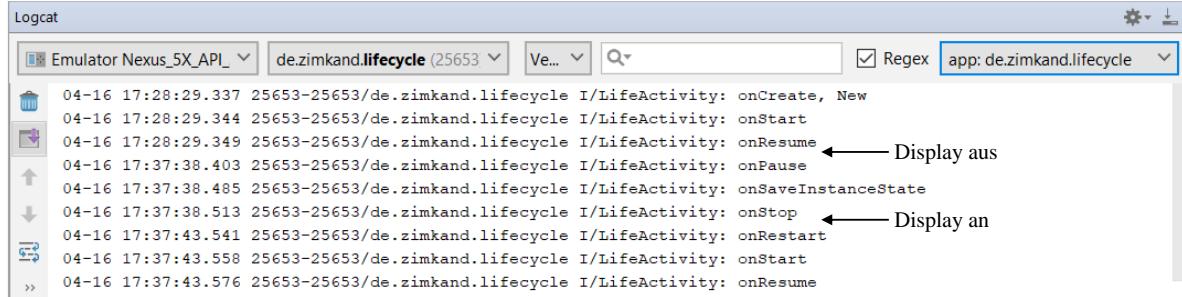
##### Aufgabe 1

Ausgehend vom Beispielprogramm in Abschnitt 6.1 genügt eine minimale Änderung in der Methode `onPause()`:

```
@Override  
protected void onPause() {  
    super.onPause();  
    Log.i(TAG, "onPause" + (this.isFinishing() ? "(Finishing)" : ""));  
}
```

## Aufgabe 2

Bei einem emulierten Android-Gerät lässt sich das Display z. B. über den Schalter  auf der rechtsseitig am Gerät angehefteten Bedienpalette oder mit der Tastenkombination **Strg+7** ausschalten.<sup>1</sup> Wie das folgende **Logcat**-Protokoll (erstellt mit der in Abschnitt 6.3 beschriebenen App) demonstriert, wird die beim Abschalten des Displays die im Vordergrund befindliche App gestoppt und nach dem Einschalten des Displays wieder gestartet:



The screenshot shows the Android Logcat interface with the following log entries:

```

Logcat
Emulator Nexus_5X_API de.zimkand.lifecycle (25653)
04-16 17:28:29.337 25653-25653/de.zimkand.lifecycle I/LifeActivity: onCreate, New
04-16 17:28:29.344 25653-25653/de.zimkand.lifecycle I/LifeActivity: onStart
04-16 17:28:29.349 25653-25653/de.zimkand.lifecycle I/LifeActivity: onResume ← Display aus
04-16 17:37:38.403 25653-25653/de.zimkand.lifecycle I/LifeActivity: onPause ← Display an
04-16 17:37:38.485 25653-25653/de.zimkand.lifecycle I/LifeActivity: onSaveInstanceState
04-16 17:37:38.513 25653-25653/de.zimkand.lifecycle I/LifeActivity: onStop ← Display an
04-16 17:37:43.541 25653-25653/de.zimkand.lifecycle I/LifeActivity: onRestart
04-16 17:37:43.558 25653-25653/de.zimkand.lifecycle I/LifeActivity: onStart
04-16 17:37:43.576 25653-25653/de.zimkand.lifecycle I/LifeActivity: onResume

```

## Kapitel 7 (Ressourcen)

### Aufgabe 1

Das Projekt **PrimeDetection** auf dem aktuellen Entwicklungsstand ist im folgenden Ordner zu finden:

**...\\BspUeb\\PrimeDetection**

## Kapitel 8 (Bedienoberfläche)

### Aufgabe 1

Ein Lösungsvorschlag ist im folgenden Ordner zu finden:

**...\\BspUeb\\UI\\Controls\\Numerische Eingabe**

### Aufgabe 2

#### 1. Falsch

Siehe Abschnitt 8.5.2.3

#### 2. Richtig

#### 3. Richtig

Eine Erläuterung zu den sogenannten *Layout-Parametern* findet sich in Abschnitt 8.2.2.

#### 4. Richtig

## Kapitel 9 (Intents und Intent-Filter)

### Aufgabe 1

Ein Lösungsvorschlag ist im folgenden Ordner zu finden:

**...\\BspUeb\\Intent\\Settings**

---

<sup>1</sup> Auf einem Rechner mit lediglich 4 GB Hauptspeicher ließ sich das Display abschalten, aber anschließend nicht wieder einschalten (beobachtet bei Android Studio 3.1).

**Aufgabe 2****1. Falsch**

Für den Start durch einen expliziten Intent ist kein Intent-Filter erforderlich.

**2. Falsch**

Per Intent wird der Benutzer an eine fremde App vermittelt, und die eigene App ist vorübergehend aus dem Spiel.

**Aufgabe 3**

Ein Lösungsvorschlag ist im folgenden Ordner zu finden:

...\\BspUeb\\Intent\\UTinMap

**Kapitel 10 (Multithreading)****Aufgabe 1****1. Falsch**

Im UI-Thread gestartete Methoden sollten nicht länger als 16 Millisekunden für die Ausführung benötigen.

**2. Richtig****3. Falsch****4. Falsch**

Es wird lediglich das Interrupt-Signal gesetzt. Ein kooperativer Thread beachtet dieses Signal regelmäßig und stellt ggf. seine Tätigkeit ein.

**5. Falsch**

Es wird lediglich Signal gesetzt. Eine korrekt programmierte asynchrone Aufgabenklasse beachtet dieses Signal regelmäßig und stellt ggf. ihre Tätigkeit ein.

**6. Richtig****Kapitel 11 (Dienste)****Aufgabe 1****1. Falsch**

Ein Dienst sollte aus Sicherheitsgründen auf keinen Fall über einen impliziten Intent gestartet werden, damit keine Vagheit bzgl. der tatsächlich ausgeführten Software besteht. Seit Android 5 (API-Level 21) führt der Aufruf von **bindService()** mit einem impliziten Intent zu einem Ausnahmefehler. Dementsprechend sollte zu einem Dienst auch kein Intent-Filter definiert werden.

**2. Richtig****3. Richtig****4. Falsch**

Diese Einschränkung gilt nur für die besonders bequem verwendbare Klasse **IntentService**.

**5. Richtig****Aufgabe 2**

Ein Lösungsvorschlag ist im folgenden Ordner zu finden:

...\\BspUeb\\Service\\LocalBoundService - BR

**Kapitel 12 (Fragmente)****Aufgabe 1****1. Falsch**

Eine Fragment-Transaktion wird nur dann konserviert, wenn vor dem **commit()** - Aufruf an das **FragmentTransaction**-Objekt dessen Methode **addToBackStack()** aufgerufen wird.

**2. Richtig****3. Richtig**

---

# Index

.	
.exit.....	342
.table .....	341
<b>@</b>	
@ColumnInfo .....	317, 323
@Dao .....	317, 324
@Database .....	320, 325
@Delete.....	319, 325
@Entity .....	316, 323
@Ignore.....	317
@Insert.....	318, 325
@NonNull .....	371
@Nullable .....	371
@PrimaryKey .....	317, 323
@Query .....	319, 324
@TypeConverter.....	321
@TypeConverters .....	321
@update .....	318, 325
<b>A</b>	
AAPT .....	114
Activity.....	75
ActivityCompat .....	354
ActivityManager .....	99
Adaptive Icons .....	108
ADB .....	36, 40
addCategory().....	233
addToBackStack().....	308
addURI() .....	362
Ahead-Of-Time .....	3
Akku.....	398
Aktion eines Intents .....	230
Aktivitäts-Manager .....	84, 96
AlertDialog.....	74
andoid	
layout_gravity .....	148
android	
exported.....	399
inputType .....	375
Android Architecture Components .....	314
Android Asset Packaging Tool .....	114
Android Debug Bridge.....	36, 340
Android Profiler .....	98
Android Virtual Device Manager .....	29
AndroidManifest.xml.....	77
Animation.....	127, 309
Animationen.....	126
Anonyme Klassen .....	175
ANR-Fehler.....	136, 138, 255
Anwendungsobjekt.....	82
AOT .....	3
API.....	4
API-Level .....	6
apk.....	47
APK .....	3
AppCompatActivity.....	76, 130
app-debug.apk.....	68
Application .....	82
apply()	
SharedPreferences .....	385
applyBatch() .....	355
ARGB .....	121
ARM .....	4, 27
ArrayAdapter<E> .....	328, 333
ArrayAdapter<String>.....	207
ArrayList<ContentProviderOperation>....	355
ART .....	3
Assembler .....	2
AsyncTask .....	262
AutoCompleteTextView.....	188
autoGenerate .....	317
autoLink .....	181
AVD-Manager .....	29
<b>B</b>	
Back Stack .....	79
Fragmente .....	308
Barriere .....	161
Beans.....	140
beginTransaction().....	307
Berechtigungen	
Broadcast Receiver .....	400
Content Provider.....	353, 366
Berechtigungsgruppe .....	354
BIND_AUTO_CREATE .....	283
Binder .....	285
bindService().....	283
bindView() .....	370
Boilerplate-Code.....	176
borderlessButtonStyle .....	190
Broadcast Receiver .....	75, 387
BroadcastReceiver .....	279
build.gradle .....	71, 322
Build-System .....	45
buildToolsVersion .....	72
Bundle .....	87, 88, 93, 234, 243, 289, 301, 309
Button .....	189
Bytecode .....	2
<b>C</b>	
C#.....	8
cached process .....	82
Call Back - Routinen .....	138

Callback-Methoden .....	87
CATEGORY_BROWSABLE .....	233
CATEGORY_DEFAULT .....	236
CATEGORY_LAUNCHER .....	233
CheckBox .....	192
CheckBoxPreference .....	375
checkSelfPermission() .....	354
Cloud Save .....	386
Color .....	383
commit() .....	308
SharedPreferences .....	385
Compiler .....	2
compileSdkVersion .....	72
ComponentName .....	229, 241, 397
CompoundButton .....	192
ConstraintLayout .....	49
ConstraintLayout .....	153, 329, 347
ContactsContract .....	356
content .....	232, 237
content (URI) .....	344
Content Provider ....	76, 86, 232, 237, 248, 343
contentDescription .....	190
ContentProvider .....	358
ContentResolver .....	343
Content-URI .....	344
ContentUris .....	344
ContentValues .....	350
ContextCompat .....	354
controls .....	138
CPU .....	2
CREATE TABLE (SQL) .....	341
createChooser() .....	250
Cursor .....	344
CursorAdapter .....	368
CursorLoader .....	371
<b>D</b>	
Dalvik .....	3
DAO .....	317, 324
Dart .....	8
Data Access Object .....	317, 324
databaseBuilder() .....	320
Date .....	320, 323
Datenbanken .....	314
Debug-Modus .....	68
DELETE (Room) .....	319
DELETE (SQL) .....	342
delete()	
ContentProvider .....	364
ContentResolver .....	352
Destroyed .....	85
DialogFragment .....	293
Dienst .....	271
Dienstprozesse .....	81, 271
doInBackground() .....	204, 266
drawable resources .....	122
DROP TABLE (SQL) .....	342
Dropdown-Liste .....	199
dx .....	3
<b>E</b>	
Eclipse .....	10
Editable .....	57
EditText .....	57, 182
EditTextPreference .....	375
Eigenschaften .....	140
Einfache Ressourcen .....	109
Element-IDs .....	116
ems .....	151, 189
Entität .....	315
Entity .....	323
Entwickleroptionen .....	97
Ereignisse .....	174
execute() .....	205
AsyncTask .....	265
Explizite Intents .....	228
Expliziter Intent .....	238
exported .....	235
Extras	
Intent .....	234
<b>F</b>	
Fabrikmethode .....	307, 309
Farben .....	121
fill_parent .....	144
findFragmentById() .....	306
findFragmentByTag() .....	313
findViewById() .....	55, 87, 116
finish() .....	245
Firebase JobDispatcher .....	394
FLAG_EXCLUDE_STOPPED_PACKAGES .....	397
FLAG_INCLUDE_STOPPED_PACKAGES .....	390, 397
Flags .....	234
Flexibilität .....	173
Flutter .....	8
Fortschrittsanzeige .....	202
Fragment .....	380
Fragmente .....	292
FragmentManager .....	306
FragmentTransaction .....	307
FrameLayout .....	166
freeMemory() .....	100
freezesText .....	181
Fullscreen-Texteingabe .....	187

**G**

GC Alloc ..... 104  
Gebundene Dienste ..... 272  
Gestartete Dienste ..... 271  
Gestoppter Zustand  
    einer App ..... 390  
getActivity() ..... 303, 310  
getArguments() ..... 310  
getDefaultSharedPreferences() ..... 384  
getExtras() ..... 243  
getFragmentManager() ..... 306  
getIntent() ..... 243  
getIntExtra() ..... 399  
getItemId() ..... 220, 223  
getLastPathSegment() ..... 362  
getMemoryClass() ..... 99  
getPackageManager() ..... 249  
getRuntime() ..... 99  
getSharedPreferences() ..... 383  
getString() ..... 115  
getStringExtra() ..... 244  
getSystemService() ..... 99  
getType() ..... 345  
getTypeface() ..... 180  
GitHub ..... 61  
goAsync() ..... 393  
Google Play Protect ..... 82  
Gradle ..... 24, 45, 71  
Gradle-Scripts ..... 47  
gravity ..... 148  
GridLayout ..... 170  
GridView ..... 222  
Größenangaben ..... 120

**H**

handleMessage() ..... 269  
Handler ..... 259, 286  
HAXM ..... 15  
Heap ..... 99, 256  
hint ..... 188  
Hintergrundfarbe ..... 189  
Hintergrundprozesse ..... 82  
Hyper-V ..... 36

**I**

IBinder ..... 288  
IllegalArgumentException ..... 363  
ImageButton ..... 190  
Implizite Intents ..... 228  
inflate() ..... 369  
initLoader() ..... 372  
inputType ..... 183  
INSERT (Room) ..... 318  
INSERT (SQL) ..... 341

**insert()**

    ContentProvider ..... 363  
    ContentResolver ..... 350  
Instant Run ..... 62  
IntelliJ IDEA Community Edition ..... 10  
Intent  
    Extras ..... 234  
    Kategorien ..... 233  
Intent ..... 228  
    Aktion ..... 230  
    Daten ..... 231  
Intent ..... 311  
Intent.ACTION\_TIME\_TICK ..... 394  
IntentFilter ..... 279  
Intent-Filter ..... 234  
Intent-Filter  
    Broadcast Receiver ..... 389  
IntentService ..... 273, 274  
Interpreter ..... 3  
interrupt() ..... 261  
invalidateOptionsMenu() ..... 215  
IrfanView ..... 190  
isChangingConfigurations() ..... 90, 283  
isFinishing() ..... 89  
isInterrupted() ..... 262

**J**

JACK ..... 4  
Java Beans ..... 140  
JavaFX ..... 139  
JetBrains ..... 10  
JILL ..... 4  
JobScheduler ..... 394

**K**

Klassen  
    anonyme ..... 175  
Komplexe Ressourcen ..... 106  
Kontextmenü ..... 337  
Kontrakt-Klasse ..... 356  
Kontrollkästchen ..... 192, 195  
Kotlin ..... 7

**L**

Lambda-Ausdruck ..... 176  
launcher ..... 79  
layout\_column ..... 169  
layout\_gravity ..... 149, 166  
layout\_height ..... 144  
layout\_margin ..... 145  
layout\_span ..... 169  
layout\_weight ..... 149, 299  
layout\_width ..... 144  
LayoutInflator ..... 141, 305, 369  
Layout-Parameter ..... 142

LifeData ..... 339  
 LinearLayout ..... 49, 130, 146  
 lint-Werkzeug ..... 72, 287  
 ListActivity ..... 207  
 ListFragment ..... 293  
 ListPreference ..... 376  
 ListView ..... 205, 222, 327  
 Loader-API ..... 370  
 LoaderManager ..... 371  
 LocalBroadcastManager ..... 278  
 Log ..... 93  
 LogCat ..... 92  
 Lokaler Dienst ..... 271  
**M**  
 Main-Thread ..... 255  
 makeText() ..... 216  
 margins ..... 145  
 Maschinencode ..... 2  
 match() ..... 362  
 match\_constraint ..... 144, 159  
 match\_parent ..... 144  
 maxLength ..... 185  
 maxMemory() ..... 100  
 MediaPlayer ..... 128  
 Mediendateien ..... 128  
 Mehrzeilige Texte ..... 185  
 memory leak ..... 286  
 Message ..... 289  
 Message-Pool ..... 269  
 MIME-Typ ..... 231, 349  
 minSdkVersion ..... 62, 72  
 Module ..... 46  
 MotionEvent ..... 174  
 MultiSelectListPreference ..... 376  
 Multitasking ..... 254  
 Multithreading ..... 254  
**N**  
 Neun-Feld - Bitmap ..... 122, 183  
 newView() ..... 369  
 nine-patch bitmap ..... 122  
 notifyDataSetChanged() ..... 336, 339  
**O**  
 obtain() ..... 269  
 onActivityCreated() ..... 373  
 onActivityResult() ..... 244, 336  
 onBackPressed() ..... 89  
 onBind() ..... 273, 288  
 onCheckedChanged() ..... 192  
 OnCheckedChangeListener ..... 192  
 onConflict ..... 325  
 onContentChanged() ..... 373  
 onContextItemSelected() ..... 220, 337

onCreate()  
     Activity ..... 87  
     ContentProvider ..... 363  
     Service ..... 273  
 onCreateActionMode() ..... 222  
 onCreateContextMenu() ..... 219, 337  
 onCreateLoader() ..... 371  
 onCreateOptionsMenu () ..... 215  
 onCreateView() ..... 305  
 onDestroy()  
     Activity ..... 90  
     Service ..... 273  
 onHandleIntent() ..... 274, 276  
 onItemClick() ..... 333  
 onListItemClick() ..... 305  
 onLoaderReset() ..... 372  
 onLoadFinished() ..... 372  
 onOptionsItemSelected() ..... 216, 383  
 onPause() ..... 89, 394  
 onPostExecute() ..... 266  
 onPreExecute() ..... 203, 265  
 onPrepareOptionsMenu(Menu) ..... 215  
 onProgressUpdate() ..... 204, 266  
 onReceive() ..... 279, 388  
 onRestart() ..... 87  
 onRestoreInstanceState() ..... 88  
 onResume() ..... 88, 394  
 onRetainNonConfigurationInstance ..... 284  
 onSaveInstanceState() ..... 88  
 onServiceConnected() ..... 284  
 onStart() ..... 87  
 onStartCommand() ..... 273  
 onStop() ..... 90  
 Optionsschalter ..... 197  
 ORDER BY (SQL) ..... 342  
 OutOfMemoryError ..... 100, 103  
**P**  
 PackageManager ..... 249  
 padding ..... 145  
 parse() ..... 249, 344  
 parseColor() ..... 383  
 Passwort ..... 185  
 Paused ..... 85  
 peekService() ..... 394  
 PendingIntent ..... 252  
 performClick() ..... 174  
 permission group ..... 354  
 Permissions  
     Broadcast Receiver ..... 400  
     Content Provider ..... 353, 366  
 Persistente Klasse ..... 315  
 PopupMenu ..... 225

---

post() .....	259, 260, 268	RxJava2.....	339
Handler.....	268	<b>S</b>	
View .....	268	Sandkasten .....	82
Preemtives Multitasking .....	254	Schaltflächen.....	189
Preference.....	375	ScrollView .....	311
Preference-API.....	375	SDK-Manager.....	26
PreferenceFragment .....	380	SELECT (Room) .....	319
PreferenceFragmentCompat.....	294	sendBroadcast() .....	397
PreferenceManager .....	381, 384	sendMessage() .....	268
Primärschlüssel .....	317	sendOrderedBroadcast() .....	398
primaryKeys.....	316	sendStickyBroadcast() .....	398
Primzahlen .....	135	Serializable .....	334
Profiler .....	98	Service .....	75, 271
ProgressBar .....	202, 263	ServiceConnection.....	284
Projektion .....	345	setAction() .....	230
publishProgress() .....	266	setArguments() .....	309
putExtra() .....	234, 249	setAutoLinkMask().....	181
<b>Q</b>		setBackgroundColor().....	383
Qualifizierer .....	111	setColumnShrinkable() .....	169
query()		setColumnStretchable() .....	169
ContentProvider .....	362, 363	setComponent().....	241, 397
ContentResolver.....	345	setContentView() .....	87, 141
Quick-Fixes .....	63	setData() .....	289
<b>R</b>		setFilters() .....	185
R 114		setFlags() .....	234
R.java .....	116	setFreezesText() .....	181
RadioButton .....	192	setInputType() .....	185
RadioGroup.....	192, 197, 199	setMax() .....	202
React Native .....	8	setOnClickListener() .....	56
RecyclerView .....	208, 293	setOnItemSelectedListener() .....	200
Refaktorieren.....	65	setPackage() .....	401
registerForContextMenu() .....	219, 337	setProgress() .....	202
registerReceiver() .....	394, 399	setText() .....	181
Relationale Datenbanken .....	314	setTextSize() .....	180, 311
RelativeLayout .....	150	setThreadPriority() .....	257
Release-Modus .....	68	Settings .....	252
requery() .....	367	setTypeface() .....	179, 195
Request-Code .....	244	setVisibility() .....	202
requestPermissions() .....	354	shared_prefs .....	379
resolveActivity() .....	249	SharedPreferences .....	383
Resources .....	115	shrinkColumns .....	169
Ressourcen .....	105	Sicherheit .....	82
Ressourcen-ID .....	112	Sichtbare Prozesse .....	81
Ressourcen-IDs .....	115	SimpleCursorAdapter .....	347
Room .....	314	Singleton-Muster .....	326
Room Persistence Library .....	360	Space .....	171
RoomDatabase .....	320, 325	Späte Bindung .....	229
Runde Icons .....	107	Speicherleck .....	286
Runnable .....	259	Spinner .....	199
Running .....	85	SQL .....	314
runOnUiThread() .....	260, 268	SQL-Injektion .....	353
Runtime .....	99	SQLite .....	314

sqlite3 ..... 340  
Stack ..... 99, 256  
stack trace ..... 60  
startActivity() ..... 240, 248  
startActivityForResult() ..... 244, 373  
startForegroundService() ..... 290  
startManagingCursor() ..... 372  
Steuerelemente ..... 138  
Sticky Broadcast ..... 398  
Stopped ..... 85  
stopService() ..... 275, 280  
stretchColumns ..... 169  
string-array ..... 303  
Support Library ..... 72, 76  
swapCursor() ..... 372  
SwitchPreference ..... 375  
Switch-Schalter ..... 192, 196

**T**

TableLayout ..... 168  
tableName ..... 316  
targetSdkVersion ..... 73, 274, 390  
Task ..... 79  
textAllCaps ..... 54, 133  
TextEdit ..... 199  
textSize ..... 180  
textStyle ..... 180  
TextView ..... 181  
Thema ..... 113  
Thread ..... 257  
THREAD\_PRIORITY\_BACKGROUND ..... 257  
Thread-Sicherheit ..... 326  
Toast-Meldungsfenster ..... 216  
ToggleButton ..... 192, 196  
Transaktion ..... 317  
Tween-Animation ..... 127  
typeface ..... 179  
Typeface ..... 179  
Typkonverter (Room) ..... 320

**U**

Überlaufsymbol ..... 208, 209  
UI-Thread ..... 255  
Umschalter ..... 192  
unbindService() ..... 281, 283, 285  
Uniform Resource Identifier ..... 231  
Untermenü ..... 212, 215  
UPDATE (Room) ..... 318  
UPDATE (SQL) ..... 342  
update()  
    ContentProvider ..... 364  
    ContentResolver ..... 351  
Uri ..... 248, 344  
URI ..... 357  
URI Permission ..... 355  
UriMatcher ..... 362  
Uri-Objekt ..... 231, 249, 344  
uses-permission ..... 353, 400  
uses-sdk ..... 78

**V**

VCS ..... 61  
View Inspector ..... 155  
ViewGroup ..... 142, 145  
virtuellen Maschine ..... 2  
Vordergrundprozesse ..... 81

**W**

WeakReference ..... 267  
WebViewFragment ..... 294  
withAppendedId() ..... 344  
wrap\_content ..... 144

**X**

Xamarin ..... 8

**Z**

Zeichenbare Ressourcen ..... 122  
Zugriffsrechte ..... 353, 366  
Zustände einer Aktivität ..... 85  
Zustandslisten ..... 125