



Universität Trier

Bernhard Baltes-Götz

Einführung in die Entwicklung von Android-Apps





2016 (Rev. 160926)

Herausgeber: Zentrum für Informations-, Medien- und Kommunikationstechnologie (ZIMK) an der Universität Trier
Universitätsring 15
D-54286 Trier
WWW: zimk.uni-trier.de
E-Mail: zimk@uni-trier.de
Autor: Bernhard Baltes-Götz
E-Mail: baltes@uni-trier.de
Copyright © 2016; 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 2016 angeboten hat, ist aber auch für das Selbststudium geeignet. In hoffentlich seltenen Fällen enthält der Text noch Formulierungen, die nur für Kursteilnehmer(innen) perfekt passen.

Behandelte Themen

Das Manuskript behandelt u.a. folgende Themen:

- Anwendungsentwicklung in Java mit dem Android Studio 2.x
- Bestandteile einer Android-App:
  - Activity
  - Fragment
  - Intent
  - Service
  - Content Provider
  - Broadcast Receiver
- Ressourcen, Bedienoberfläche
- Multithreading
- Datenbankprogrammierung

Voraussetzungen bei den Leser(innen)[1]

- **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 2016).
- **EDV-Plattform**
  Dass die Leser über ein eigenes Android-Smartphone oder -Tablet verfügen, ist nützlich, nicht unbedingt erforderlich, weil im verwendeten Entwicklungssystem Emulatoren für Android-Geräte zur Verfügung stehen. Zur Softwareentwicklung wird im Kurs bzw. 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.

[1] Zur Vermeidung von sprachlichen Umständlichkeiten beschränkt sich das Manuskript ab jetzt auf die männliche Form.

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 Android-Apps

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

werden dankbar entgegen genommen.[1]

Trier, im September 2016 Bernhard Baltes-Götz

[1] An dieser Stelle geht ein herzliches Dankeschön an Frau Birgit Feltrup für das Aufspüren von zahlreichen Tippfehlern.

Inhaltsverzeichnis

VORWORT 3

INHALTSVERZEICHNIS 5

1 JAVA UND ANDROID 13

1.1 Quellcode, Bytecode und Maschinencode 13

1.2 Android-API 17

1.3 Linux-Unterbau 17

1.4 Android-Versionen und API-Levels 18

1.5 Dokumentation zu Android und seinem Java-API 19

1.6 Übungsaufgaben zu Kapitel 1 19

2 INSTALLATION DER ENTWICKLUNGSUMGEBUNG UNTER WINDOWS 20

2.1 JDK 20

2.2 Android Studio samt Android SDK installieren 20

2.3 Android-Studio zum ersten Mal starten 24

2.4 Die erste Android-App 27

2.5 AVD-Manager 31

2.6 SDK-Manager 36

2.7 Intel(R) Hardware Accelerated Execution Manager (HAXM) 39

2.8 Wichtige Pfade für Einstellungen und Projekte 42

2.9 Android Debug Bridge 42

2.10 Reales Android-Gerät mit der Entwicklungsumgebung verbinden 44
2.10.1 USB-Debugging auf dem Android-Gerät aktivieren 44
2.10.2 Android USB-Treiber installieren 44

2.11 Übungsaufgaben zu Kapitel 2 45

3 ANDROID STUDIO KENNEN LERNEN 46

3.1 Projekt anlegen 46

3.2 Bedienoberfläche entwerfen 49

3.3 Click-Behandlungsmethode für den Schalter erstellen 55

3.4 Sichern und Wiederherstellen 59

3.5 Programm ausführen 60

3.6 Nützliche Funktionen im Android Studio ... 62

3.7 Eigene Apps auf einem beliebigen Android-Gerät installieren ... 62

3.8 Projektkonfiguration einsehen und ändern ... 65

3.9 Kursprojekte öffnen ... 66

3.10 Übungsaufgaben zu Kapitel 3 ... 67

4 KOMPONENTEN UND SYSTEMUMGEBUNG EINER APP ... 69

4.1 Komponenten einer Android-App ... 69

4.2 Manifest ... 70

4.3 Task und Back Stack ... 71

4.4 Apps und Prozesse ... 73

4.5 Sicherheit ... 74

4.6 Übungsaufgaben zu Kapitel 4 ... 75

5 LEBENSZYKLUS EINER AKTIVITÄT ... 76

5.1 Zustände ... 76

5.2 Zustandswechselmethoden ... 78
5.2.1 onCreate() ... 78
5.2.2 onStart() ... 79
5.2.3 onRestoreInstanceState() ... 79
5.2.4 onResume() ... 79
5.2.5 onSaveInstanceState() ... 79
5.2.6 onPause() ... 80
5.2.7 onStop() ... 81
5.2.8 onDestroy() ... 81

5.3 App zur Lifecycle-Demonstration ... 81
5.3.1 Logging ... 82
5.3.2 Protokollausgaben in den Methoden für Activity-Zustandswechsel ... 84
5.3.2.1 Neustart nach Konfigurationsänderung ... 86
5.3.2.2 Neukreation nach Prozessterminierung ... 87

5.4 App-Monitoring im Android Studio ... 88

5.5 Übungsaufgaben zu Kapitel 5 ... 90

6 RESSOURCEN ... 91

6.1 Ressourcentypen ... 91

6.2 Konfigurationsabhängige Ressourcen ... 95

6.3 Ressourcen ansprechen ... 96
6.3.1 Zugriff in einer Ressourcendefinitionsdatei ... 96
6.3.2 Zugriff im Java-Quellcode ... 98
6.3.3 Element-IDs ... 100

6.4 Zeichenfolgen ... 101

6.5 Größenangaben ... 103

6.6 Farben ... 104

6.7 Zeichenbare Ressourcen ... 105
6.7.1 Neun-Feld-Bitmaps ... 105
6.7.2 Zustandslisten ... 107

6.8 Animationen ... 108

6.9 Mediendateien ... 110

6.10 Entspannung und Motivationsstärkung ... 111
6.10.1 Projekt anlegen ... 111
6.10.2 Bedienoberfläche entwerfen ... 111
6.10.3 Click-Behandlungsmethode für den Schalter erstellen ... 114

6.11 Übungsaufgaben zu Kapitel 6 ... 117

7 BEDIENOBERFLÄCHE ... 118

7.1 Layoutdefinition als Ressource ... 119

7.2 Die View - Klassenhierarchie ... 121
7.2.1 Elementare Steuerelementattribute bzw. -eigenschaften ... 122
7.2.1.1 Element-ID ... 122
7.2.1.2 Breite und Höhe ... 122
7.2.1.3 Ränder (Margins) und Einrückung (Padding) ... 123

7.3 Container ... 124
7.3.1 LinearLayout ... 124
7.3.2 RelativeLayout ... 128
7.3.3 FrameLayout ... 130
7.3.4 TableLayout ... 132
7.3.5 GridLayout ... 134

7.4 Ereignisbehandlung ... 138
7.4.1 Elementare und aufbereitete Ereignisse ... 138
7.4.2 Optionen zum Registrieren von Ereignisempfängern ... 139
7.4.2.1 Ereignisbehandlung durch eine anonyme Klasse ... 139
7.4.2.2 Klick-Ereignismethode per XML-Attribut registrieren ... 140
7.4.3 Ereignisverarbeitungskette (unterbrechen) ... 141

7.5 Bedienelemente ... 141
7.5.1 Beschriftungen (TextView) ... 141
7.5.2 Texteingabefelder (EditText) ... 143
7.5.2.1 Optik ... 144
7.5.2.2 Irreguläre Eingaben verhindern ... 145
7.5.2.3 Mehrzeiliges Texteingabefeld ... 147
7.5.2.4 Aktion nach Fertigstellung der Eingabe ... 147
7.5.2.5 Fullscreen-Texteingabe ... 148
7.5.2.6 AutoCompleteTextView ... 149
7.5.2.7 Weitere Attribute für EditText-Elemente ... 150
7.5.3 Aktionsschalter ... 150
7.5.4 Umschalter ... 153
7.5.4.1 Kontrollkästchen ... 155
7.5.4.2 Toggle-Button und Switch ... 156
7.5.4.3 Optionsschalter ... 158
7.5.5 Einfachauswahl per Spinner-Element ... 159
7.5.6 Fortschrittsanzeige ... 161
7.5.7 ListView und ListActivity ... 162

7.6 Menüs ... 165
7.6.1 Menüdeklaration per XML-Datei ... 165
7.6.1.1 Elemente einer Menüdeklaration ... 165
7.6.1.2 Beispiel ... 166
7.6.1.3 Untermenüs ... 167
7.6.1.4 Items gruppieren ... 168
7.6.2 Optionsmenü ... 169
7.6.2.1 Hardware-Menütaste versus App Bar ... 169
7.6.2.2 Zusammenstellen ... 169
7.6.2.3 Menüereignisbehandlung ... 170
7.6.3 Kontextmenü ... 172
7.6.3.1 Flottierendes Kontextmenü ... 172
7.6.3.2 Kontextabhängige Aktionen ... 173

7.7 Übungsaufgaben zu Kapitel 7 ... 173

8 INTENTS UND INTENT-FILTER ... 175

8.1 Bestandteile eines Intent-Objekts ... 176
8.1.1 Name der Komponente ... 176
8.1.2 Aktion ... 177
8.1.3 Daten ... 178
8.1.4 Kategorie ... 179
8.1.5 Extras ... 180
8.1.6 Flags ... 180

8.2 Intent-Filter ... 180
8.2.1 Bestandteile eines Intent-Filters ... 181
8.2.2 data - Element ... 182

8.3 Eigene Activity über einen expliziten Intent starten ... 183
8.3.1 Einfacher Aktivitätsstart ... 184
8.3.2 Extradaten übergeben und in der sekundären Aktivität verwenden ... 185
8.3.3 Rückmeldung der sekundären Aktivität auswerten ... 186

8.4 Fremde Aktivitäten über implizite Intents starten ... 188
8.4.1 Intent-Auflösung ... 188
8.4.2 Aktivitätsstart ohne Rückmeldung ... 188
8.4.3 Rückmeldung der sekundären Aktivität auswerten ... 191

8.5 Intents in Wartestellung ... 192

8.6 Übungsaufgaben zu Kapitel 8 ... 192

9 MULTITHREADING ... 193

9.1 Hintergrund-Thread mit elementaren Mitteln realisieren ... 195

9.2 Helfer-Threads dürfen nicht zur Altlast werden ... 198

9.3 AsyncTask ... 199

9.4 Übungsaufgaben zu Kapitel 9 ... 203

10 DIENSTE 204

10.1 Gestarteter Dienst aus der Klasse IntentService 205
10.1.1 Besonderheiten der Klasse IntentService 205
10.1.2 IntentService in das Projekt aufnehmen 206
10.1.3 Dienst starten und stoppen 208
10.1.4 Ergebniskommunikation per LocalBroadcastManager 209

10.2 Verbundener Dienst 211
10.2.1 Rohling für einen bindungsfähigen Dienst per Entwicklungsumgebung erstellen 212
10.2.2 Bindung herstellen und aufheben 213
10.2.3 Klienten-Server - Kommunikation via Binder-Objekt 214
10.2.4 Service-Ableitung definieren 215

10.3 Vordergrunddienste 218

11 FRAGMENTE 219

11.1 Aufgabenverteilung zwischen Aktivitäten und Fragmenten 220

11.2 Vorschlag für ein AVD-Tablet 221

11.3 Beispiel für die Verwendung von Fragmenten 222

11.4 Fragmente in eine Aktivität integrieren 224

11.5 Lebenszyklusmethoden bei Fragmenten 225

11.6 Fragment-Initialisierung 226

11.7 Fragment-Transaktionen 229
11.7.1 Der Fragment-Manager 230
11.7.2 Fragmente per Programm erstellen und in eine Aktivität aufnehmen 231
11.7.2.1 Animierte Übergänge 232
11.7.2.2 Back Stack für Fragment-Transaktionen 232

11.8 Argument-Bundles und Fabrikmethoden 233

11.9 Fragmente im Single-Pane – Modus 235

12 DATENBANKVERWALTUNG MIT SQLITE 237

12.1 Besonderheiten bei SQLite unter Android 237

12.2 Direktkontakt mit SQLite per ADB 238
12.2.1 SDK-Kommando adb shell 238
12.2.2 Interaktion mit SQLite 240
12.2.3 ADB-Version von PuTTY 241

12.3 Android-Typen zur Datenbankbearbeitung 242
12.3.1 SQLiteDatabase 242
12.3.2 SQLiteOpenHelper 242
12.3.3 Cursor 244

12.4 Ereignistagebuch anlegen und befüllen 245
12.4.1 SQLiteOpenHelper für die Kreation und Aktualisierung der Datenbank 246
12.4.2 Datenbankmodifikation per SQLiteDatabase-Objekt 248
12.4.3 Aktivität zum Erfassen neuer Ereignisse 249

12.5 Ereignisliste aus der Datenbank abrufen und per ListView anzeigen ... 250
12.5.1 CursorAdapter ... 251
12.5.2 Aufgabenverteilung durch die Listenaktivität ... 253
12.5.3 Datenbankabfrage ... 254
12.5.4 SimpleCursorAdapter ... 256

12.6 Tabellenzeilen aktualisieren ... 257

12.7 Tabellenzeilen per Kontextmenü löschen ... 261

13 CONTENT PROVIDER NUTZEN UND ANBIETEN ... 264

13.1 Content Provider fremder Anwendungen nutzen ... 264
13.1.1 Content-URI ... 265
13.1.2 Zugriff via ContentResolver ... 265
13.1.2.1 Abfragen ... 266
13.1.2.2 Modifikationen ... 268
13.1.3 Zugriffsrechte ... 269
13.1.4 Alternative Techniken für den Zugriff auf einen Content Provider ... 270
13.1.5 Dokumentation und Kontrakt-Klasse ... 270

13.2 Content Provider erstellen ... 272
13.2.1 Content-URIs und andere Bestandteile der Schnittstelle definieren ... 272
13.2.2 ContentProvider-Ableitung implementieren ... 273
13.2.2.1 UriMatcher ... 274
13.2.2.2 onCreate() implementieren ... 275
13.2.2.3 query() implementieren ... 276
13.2.2.4 insert() implementieren ... 276
13.2.2.5 delete() implementieren ... 277
13.2.2.6 update() implementieren ... 277
13.2.2.7 getType() implementieren ... 278
13.2.3 Provider-Eintrag im Anwendungsmanifest ... 278
13.2.4 Zugriffsrechte definieren ... 279
13.2.5 Modifikationen der bisherigen Event Diary - Klassen ... 279

13.3 Loader und LoaderManager ... 280
13.3.1 Wichtige Typen im Loader-API ... 281
13.3.2 Loader-Callback - Methoden implementieren ... 281
13.3.3 Loader-Initialisierung und -Aktualisierung ... 282

14 SHARED PREFERENCES ... 285

14.1 Preferences-Ressourcendatei ... 285

14.2 Einstellungsdateien einer App ... 288

14.3 Preferences-Bedienoberfläche ... 290
14.3.1 Einstellungsaktivität bzw. -fragment ... 290
14.3.2 Einstellungsaktivität via Optionsmenü starten ... 292

14.4 Einstellungsdaten per Programm schreiben und lesen ... 293
14.4.1 Die Klasse SharedPreferences ... 293
14.4.2 Einstellungsdaten schreiben ... 294
14.4.3 Einstellungsdaten lesen ... 295

14.5 Sonstige Optionen bei der Einstellungsverwaltung ... 296

15 BROADCAST RECEIVER 297

15.1 Statische Registrierung 297

15.2 Dynamische Registrierung 303

15.3 Broadcast-Nachrichten versenden 304

15.4 Broadcast-Varianten 305
15.4.1 (Un)geordnete Verarbeitung 305
15.4.2 Sticky Broadcasts 306

15.5 Sicherheit 307
15.5.1 Beschränkung der zulässigen Absender 307
15.5.2 Beschränkung der zulässigen Empfänger 309

15.6 Prozessinterne Broadcast-Nachrichten 310

ANHANG 311

A. Lösungsvorschläge zu den Übungsaufgaben 311
Kapitel 1 (Java und Android) 311
Kapitel 3 (Android Studio kennen lernen) 311
Kapitel 4 (Komponenten und Systemumgebung einer App) 311
Kapitel 5 (Lebenszyklus einer Aktivität) 311
Kapitel 6 (Ressourcen) 312
Kapitel 7 (Bedienoberfläche) 312
Kapitel 8 (Intents und Intent-Filter) 312
Kapitel 9 (Multithreading) 313

LITERATUR 315

STICHWORTREGISTER 316

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. 83% erreicht[1] (Prognose für 2016,Tendenz steigend) und zeigt auch auf dem Tablet-Markt eine ähnliche Dominanz.[2] 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).

Als Java im weithin akzeptierten TIOBE -Index zur Verbreitung bzw. Beliebtheit von Programmiersprachen im April 2015 den Platz 1 von C zurückerobert hat, wurde diese Entwicklung auf der TIOBE-Webseite folgendermaßen erklärt:[3]

> 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 gewandelt (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*. Hier ist z.B. eine Java-Methode zu sehen, die einen Bruch mit Zähler und Nenner in zwei Texteingabefeldern einer Android-Bildschirmseite (Aktivität) kürzt:

---

1 http://de.statista.com/statistik/daten/studie/182363/umfrage/prognostizierte-marktanteile-bei-smartphone-betriebssystemen/
2 http://de.statista.com/statistik/daten/studie/242259/umfrage/prognose-zu-marktanteilen-bei-tablets-nach-betriebssystem/
3 http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html (abgerufen am 03.05.2015)

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

    if (z != 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;
    } else
        n = 1;

    etZaehler.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.

Es gibt auch reale Java-Prozessoren, die Bytecode als Maschinensprache verwenden, z.B. die auf mobilen und eingebetteten Systemen stark verbreiteten ARM-Prozessoren mit Jazelle-Erweiterung.[1] Allerdings haben reale Java-Maschinen bisher keine große Bedeutung erlangt.

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

[1] Siehe: http://en.wikipedia.org/wiki/Jazelle

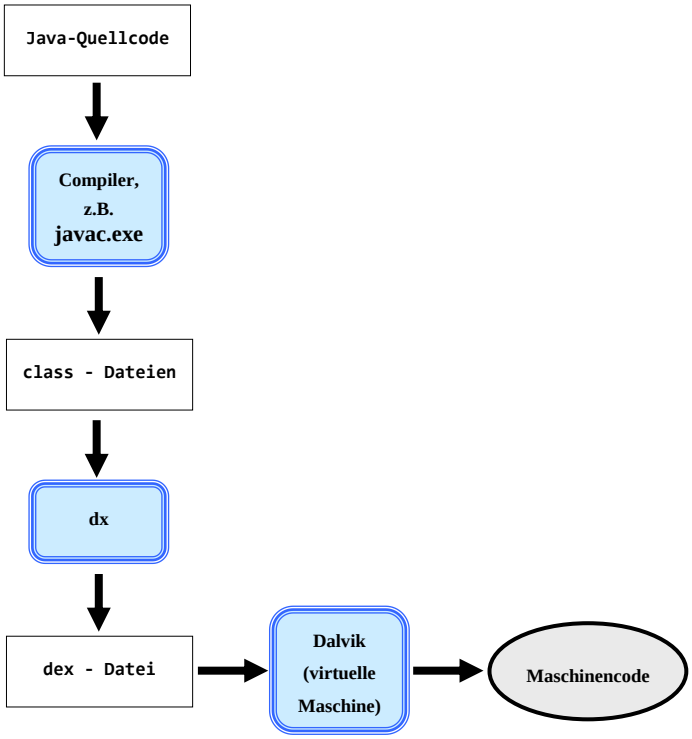
Interpretieren: Bytecode → Maschinencode

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 (**J**ust-**i**n-**T**ime - 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“:[1]

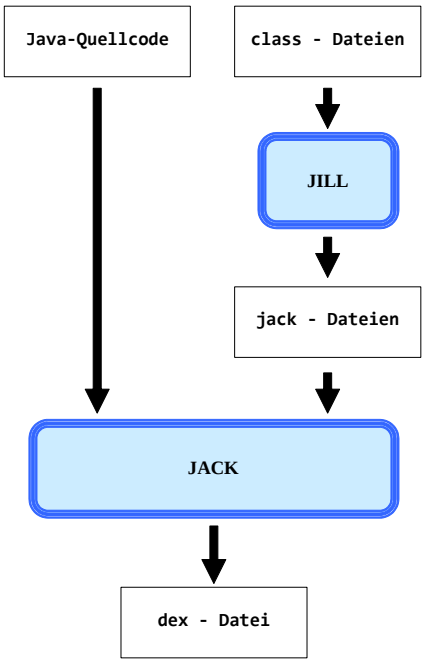
- Schnellere Installationen (‚weil das Ahead-Of-Time - Übersetzen entfällt)
- Einsparung von RAM- und Flash-Speicher

Auch beim Erstellungsprozess mit der **dex**-Datei als Ergebnis sind Änderungen im Gange, die heute schon nutzbar sind und mit der kommenden Android-Version N (erwartet im Herbst 2016) zusammen mit der Version 2.1 des Android-Studios zum Standard werden.[2] Die Werkzeuge **javac** und **dx** werden ersetzt durch Jack und Jill:[3]

- *Java Android Compiler Kit (JACK)* in der Datei **jack.jar**
  Der Compiler übersetzt Quellcode direkt in das dex-Format.
- *Jack Intermediate Library Linker (JILL)*. in der Datei **jill.jar**
  Wenn konventioneller Java-Bytecode (mit Klassenbibliotheken) einzubinden ist, wird der Compiler Jill aktiv und übersetzt die **class**-Dateien in ein Format, das Jack verarbeiten kann (Namenserweiterung für die Ergebnisdateien: **.jack**). Bei der App-Erstellung muss Jill nur dann tätig werden, wenn eine **class**-Datei ergänzt oder aktualisiert worden ist.

Eventuell waren die seit 2010 laufenden juristischen Auseinandersetzungen zwischen den Firmen Google und Oracle ein Grund dafür, im Erstellungsprozess für Android-Apps die Rolle des konventionellen Java-Bytecodes zurückzudrängen.

Der zukünftige App-Erstellungsprozess sieht folgendermaßen aus:



Zu jeder realen Prozessorarchitektur muss eine Android-Runtime erstellt werden, damit dort Dalvik-Bytecode ausgeführt werden kann. Traditionell besonders gut unterstützt sind CPUs mit einem Design der britischen Prozessorschmiede ARM (*Advanced RISC Machines*). Aber auch andere Firmen (z.B. Intel, MIPS) sind darum bemüht, Android auf ihre Prozessoren zu bringen.

---

1 https://developer.android.com/about/versions/nougat/android-7.0.html#jit_aot
2 Für den Jack-Prozess müssen die Build-Tools im Android-SDK eine Version ab 21 besitzen (aktuell: 24).
3 http://www.androidpolice.com/2014/11/30/jack-and-jill-are-googles-new-compilers-for-android-app-developers/

1.2 Android-API

Damit die Programmierer nicht das Rad (und ähnliche Dinge) ständig neu erfinden müssen, bietet jede Java-Plattform eine Standardbibliothek mit fertigen Klassen für nahezu alle Routineaufgaben, die oft als **API** (***A**pplication **P**rogram **I**nterface*) bezeichnet wird. Im Manuskript werden Sie zahlreiche API-Klassen kennen lernen. 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, so dass 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, für Desktop-Anwendungen in Java) besitzt das Android-Java eine deutlich verschiedene Standardbibliothek, um den Besonderheiten von Smartphones und Tablets gerecht werden zu können:

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

Derzeit unterstützt Google bei Android die Programmiersprache Java nur bis zur Version 7. Mit der nächsten, im Herbst 2016 erwarteten Android-Version (API-Level 24, Versionsbezeichnung *N* oder *Android 7*) wird Java 8 unterstützt.

Für die bisherigen Android-Versionen (bis Version 6) hat die Firma Google eine eigene, auf Java 6 basierende Standardbibliothek kreiert. Offenbar ist aber (in 2016) ein Wechsel auf die von der Firma Oracle gemeinsam mit der Open-Source-Community entwickelte **OpenJDK**-Bibliothek im Gange.[1] Es ist für Java- bzw. Android-Entwickler sehr vorteilhaft, wenn das Android-API näher an die Java SE (Java Standard Edition) heranrückt.

1.3 Linux-Unterbau

Android ist keine komplette Betriebssystem-Neuentwicklung, sondern verwendet einen Linux-Kernel als Basis. Jede Android-App läuft in einer eigenen virtuellen Java-Maschine, und diese agiert in einem eigenen Prozess des Linux-Systems.

---

[1] Quelle (abgerufen am 31.03.2016): http://www.heise.de/developer/meldung/Android-N-Googles-Mobilsystem-wird-auf-Open-Source-Java-OpenJDK-aufsetzen-3057453.html

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

http://developer.android.com/guide/topics/manifest/uses-sdk-element.html

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
4.4W	20	KITKAT_WATCH (nur Wearables)
5.0	21	LOLLIPOP
5.1	22	LOLLIPOP_MR1
6.0	23	MARSHMALLOW
7.0	24	NOUGAT

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

http://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	Verwendbare Java-Version
Bis 18	6
Ab 19	7
Ab 24	8

1.5 Dokumentation zu Android und seinem Java-API

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

- Entwicklungswerkzeuge
  http://developer.android.com/sdk/index.html
- Unterrichtseinheiten
  http://developer.android.com/training/index.html
- Dokumentation wichtiger Android-Bestandteile
  http://developer.android.com/guide/components/index.html
- Technische Referenz (Dokumentation der Pakete, Klassen und Interfaces im API)
  http://developer.android.com/reference/packages.html

1.6 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.

2 Installation der Entwicklungsumgebung unter Windows

Zum Erstellen von Android-Apps hat die Firma Google viele Jahre auf die meistverwendete Java-Entwicklungsumgebung Eclipse gesetzt und diese um Android-spezifische Werkzeuge erweitert (Android Developer Tools). Im Jahr 2014 hat Google 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.[1]

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. 8 GB Massenspeicher belegt.

Auf dem Entwicklungsrechner ist auch Arbeitsspeicher sehr willkommen. Unter Windows sollten es mindestens 4 GB sein.

Als die anschließende Beschreibung (im April 2016) erstellt wurde, waren bei Android die Version 6 und beim Android Studio die Version 2.0 aktuell. Es ist zu hoffen, dass die zum Zeitpunkt der Lektüre aktuellen Versionen nur mäßig und oberflächlich von den Beschreibungen abweichen.

2.1 JDK

Weil es sich beim Android Studio um ein Java-Programm handelt, wird zur Ausführung auf dem Entwicklungsrechner eine Java Virtual Maschine (JVM) benötigt. Dabei begnügt sich das Android Studio *nicht* mit einer Java Runtime Environment (JRE), sondern besteht auf dem Java Software Development Kit (JDK). Während sich die JRE auf die Rolle einer Ausführungsumgebung für Java-Programme beschränkt, enthält das JDK zusätzlich etliche Werkzeuge, den Quellcode der JSE-Standardbibliothek und vor allem einen Compiler, der Java-Quellcode in Byte-Code übersetzen kann.

Es ist kein Problem, wenn statt der offiziell vorausgesetzten JDK-Version 7 die aktuelle Version 8 installiert ist.

Über die folgende Webseite der Firma Oracle

http://www.oracle.com/technetwork/java/javase/downloads/index.html

können Sie bei Bedarf ein JDK beziehen, wobei diese Software aus Sicherheitsgründen stets aktuell gehalten werden muss.

Unter Windows 64 wird die 64-Bit - Version des Android-Studios installiert, so dass ein JDK in 64-Bit-Ausführung benötigt wird.

2.2 Android Studio samt Android SDK installieren

Am 07.04.2016 wurde die Datei **android-studio-bundle-143.2739321-windows.exe** mit ca. 1,13 GB über die folgende Webseite zum Herunterladen angeboten:

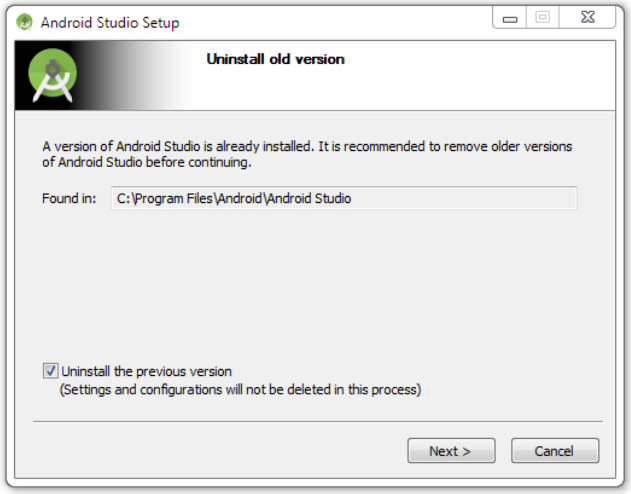
http://developer.android.com/sdk/index.html

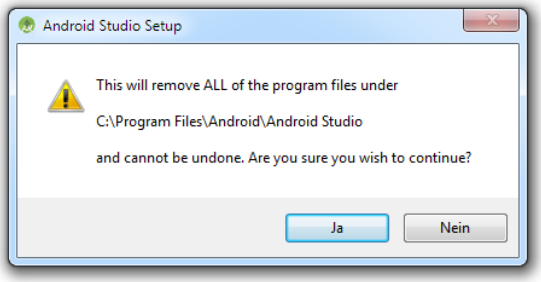
Starten Sie das herunter geladene Programm per Doppelklick, und quittieren Sie ggf. die Anfrage der Windows-Benutzerkontensteuerung mit **Ja**. Dem Vorschlag, eine (im Ordner **C:\Program Files\Android\Android Studio**) vorgefundene Altversion zunächst zu entfernen, sollten Sie zustimmen:

---

[1] 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

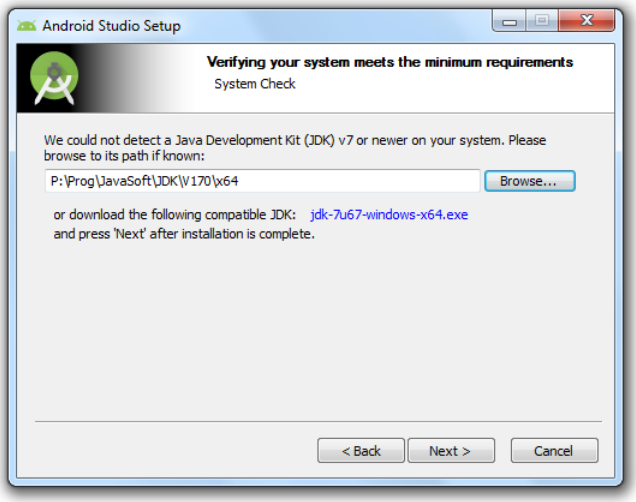




So startet die Installation der aktuellen Version:

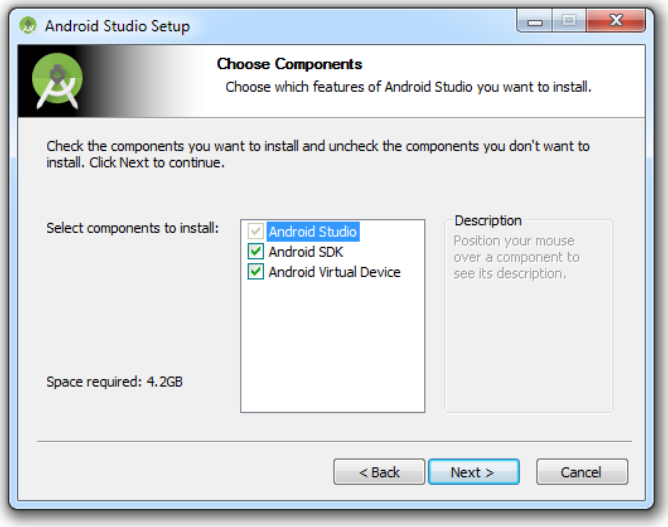


Wenn das Installationsprogramm kein JDK findet, muss man den Ordner einer JDK-Installation (ab Version 7) angeben, z.B.:



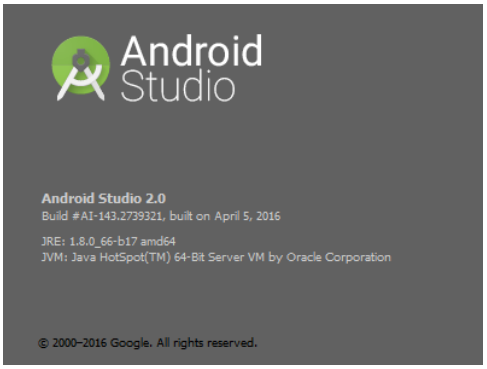
Mit einer Umgebungsvariablen `JAVA_HOME`, die auf einen JDK-Ordner zeigt, lässt sich der Misserfolg des Installationsprogramms bei der JDK-Suche verhindern (siehe Abschnitt 2.3).

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-bundle-143.2739321-windows.exe** ist die Version 2.0 (Build 143.2739321) enthalten:



- **Android SDK**
  Ein Verzicht auf diese Komponente ist nur dann sinnvoll, wenn das SDK auf andere Weise installiert werden soll bzw. bereits installiert ist.[1] Google differenziert zwischen den SDK- und den Platform-Tools.[2]
  - SDK-Tools
    Sie sind unabhängig von einer speziellen Android-Plattform. Dazu gehört z.B. der Android-Emulator. Das Bundle **android-studio-bundle-143.2739321-windows.exe** installiert die Version 25.1.1.
  - Platform-Tools
    Von den Plattform-abhängigen Werkzeugen ist vor allem die **Android Debug Bridge** (ADB) zu erwähnen, die einen Konsolenzugriff auf ein emuliertes oder reales Android-Gerät erlaubt. Das Bundle **android-studio-bundle-143.2739321-windows.exe** installiert die Version 23.1.

  Beim ersten Start des Android Studios kontrolliert ein Setup Wizard die Aktualität der SDK-Installation (siehe Abschnitt 2.3).

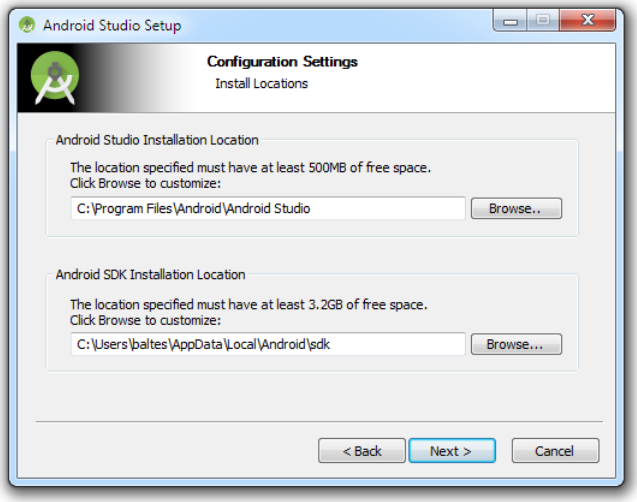
---

[1] Das Android SDK ist über die folgende Webseite als separater Download verfügbar:
http://developer.android.com/sdk/index.html#Other

[2] Eine Dokumentation zu den Werkzeugen findet sich hier:
http://developer.android.com/tools/help/index.html

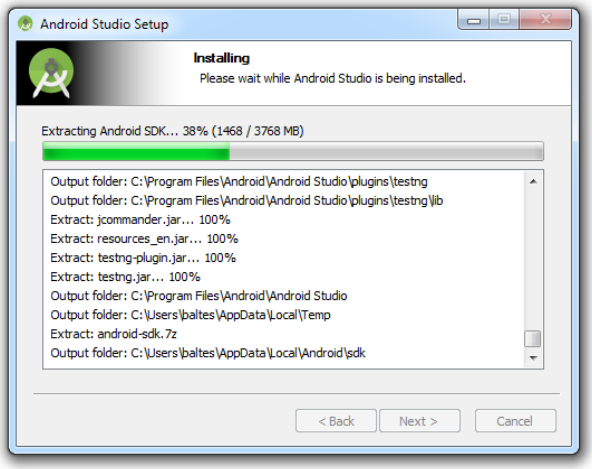
- **Android Virtual Device**
  Das Bundle **android-studio-bundle-143.2739321-windows.exe** installiert die zum Entwickeln erforderlichen Bibliotheken und Dokumente für **Android 6.0 (API-Level 23)** sowie das **Google APIs Intel x86 Atom System Image**. Um dieses System-Image nutzen zu können, muss der Entwicklungsrechner eine CPU mit Virtualisierungstechnik (VT) besitzen, die im BIOS aktiviert ist, und der *Intel(R) Hardware Accelerated Execution Manager* muss installiert werden, was in der Regel beim ersten Start des Android Studios passiert (siehe Abschnitt 2.3). Wenn das **Intel x86 Atom System Image** nicht nutzbar ist, muss per SDK-Manger ein alternatives (ARM-basiertes) System-Image installiert werden (siehe Abschnitt 2.6). In jedem Fall muss per AVD-Manager basierend auf einem installierten System-Image ein virtuelles Android-Gerät eingerichtet werden (siehe Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden.**).

Nach der Bestätigung der Lizenzvereinbarungen werden die Installationsordner erfragt, wobei für das Studio ein Computer-bezogener und für das SDK ein Benutzer-bezogener Installationsort vorschlagen wird:

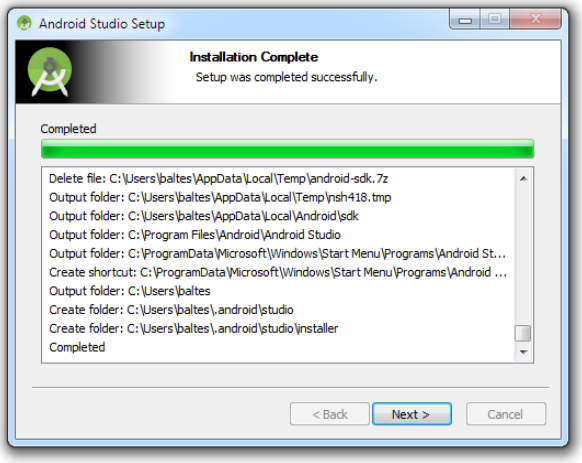


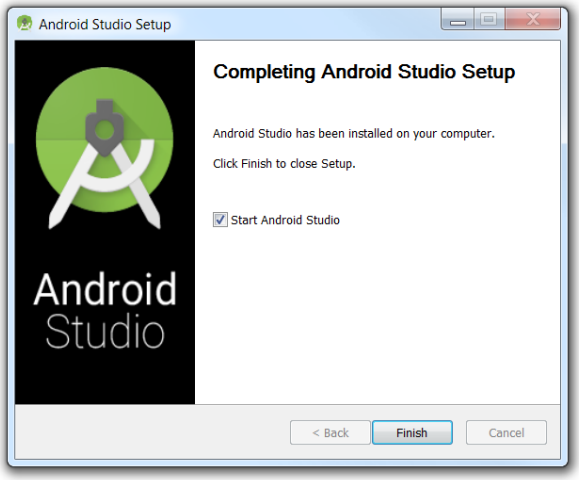
Das SDK in einen Ordner *ohne* Schreibrechte für den regelmäßigen Android Studio - Benutzer zu installieren, ist *nicht* empfehlenswert.

Nach der Wahl bzw. Bestätigung des Startmenüordners werden die Komponenten installiert, wobei vor allem das umfängliche Android SDK für einen Zeitaufwand von mehreren Minuten sorgt:



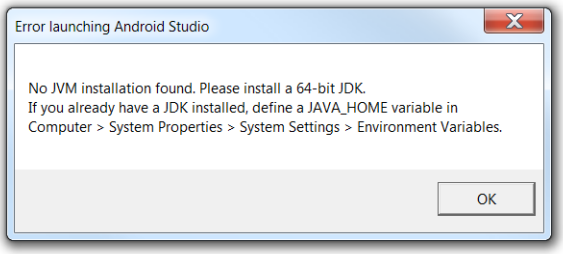
Schließlich das Android-Studio einsatzbereit:[1]



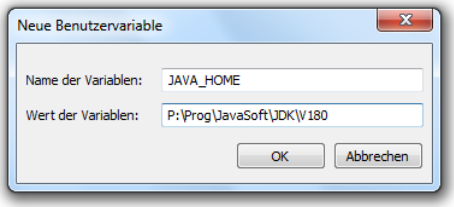


2.3 Android-Studio zum ersten Mal starten

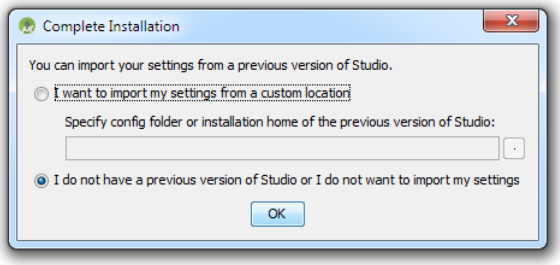
Das Android Studio kann nicht starten, wenn es kein JDK findet:



Wenn eine geeignete JDK-Installation vorhanden ist, lässt sich das Problem durch die Definition der Umgebungsvariablen JAVA_HOME vermeiden, z.B.:



Beim ersten Start des Android-Studios besteht ggf. die Möglichkeit, Einstellungen von einer früheren Studio-Version zu übernehmen:

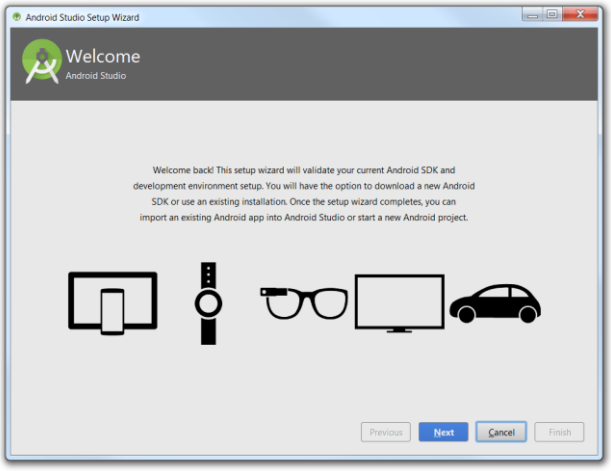


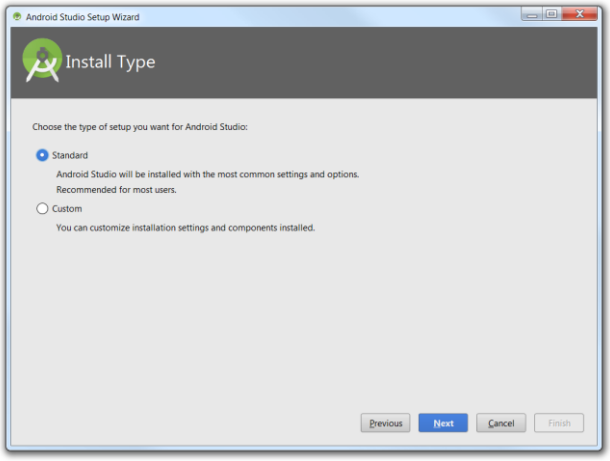
---

[1] Das Setup-Programm hinterlässt die Textdatei **firstrun.data** mit diversen Informationen (z.B. Installationsorte von JDK und Android SDK) im Profilordner des Benutzers, z.B. in:

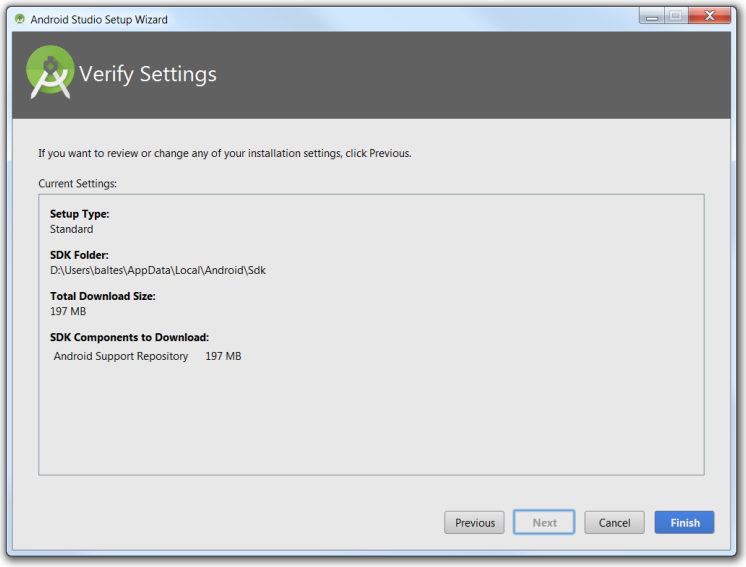
C:\Users\theo\.android\studio\installer

Dann startet entweder der **Android Studio Setup Wizard**:

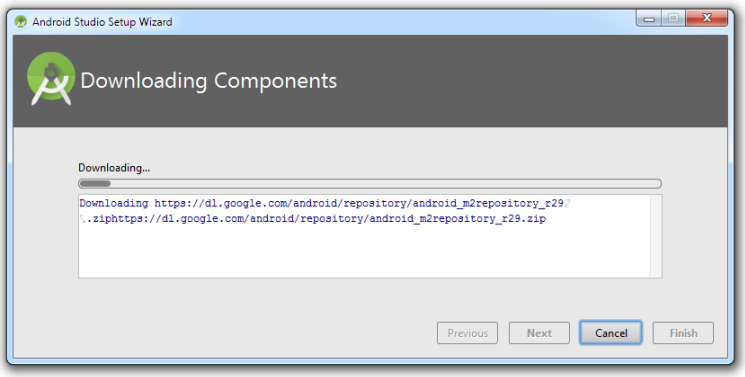




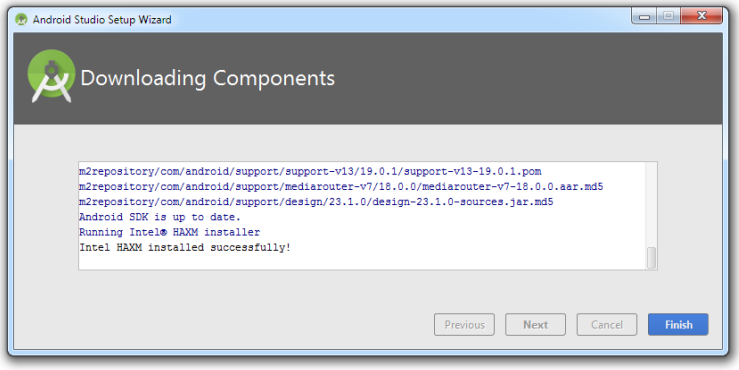
und beschreibt die erforderlichen Ergänzungsinstallationen für das Android-SDK,



oder die Ergänzung der SDK-Installation wird spontan durchgeführt:

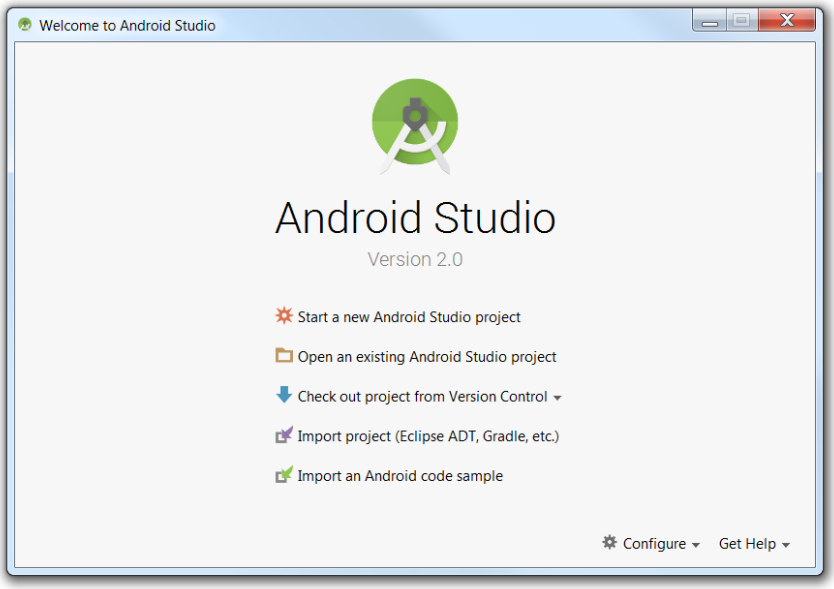


Besitzt der Entwicklungsrechner eine Intel-CPU mit Virtualisierungstechnik VT, dann wird der *Intel(R) Hardware Accelerated Execution Manager (HAXM)* installiert, der für eine beschleunigte Android-Emulation sorgt:



Das Bundle **android-studio-bundle-143.2739321-windows.exe** enthält die HAXM-Version 6.0.1.

Nach der SDK-Aktualisierung bzw. -Ergänzung ist das Android Studio einsatzbereit:

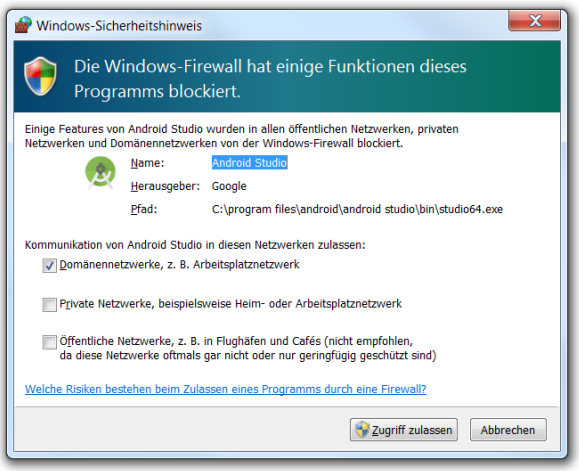


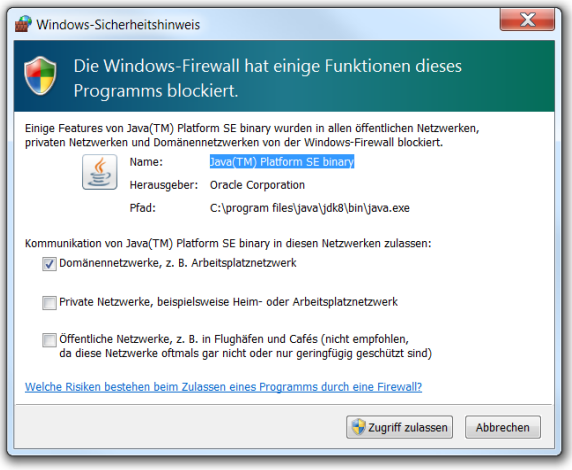
Das Erstellen der ersten App wird in Abschnitt 2.2 beschrieben.

Insgesamt entstehen bei der Installation der Entwicklungsumgebung samt Aktualisierung bzw. Vervollständigung beim ersten Start des Android Studios die folgenden Ordner:

Zweck	Voreingestellter Ordner unter Win-64	Größe
Programm Android Studio 2.0	C:\Program Files\Android\Android Studio	ca. 500 MB
Android SDK	C:\Users\<user>\AppData\Local\Android\sdk	ca. 3,7 GB
Android-Konfiguration (inkl. virtuelle Geräte)	C:\Users\<user>\.android	ca. 1,4 GB
Konfiguration Android Studio 2.0	C:\Users\<user>\.AndroidStudio2.0	ca. 400 MB

Beim Starten des Android Studios bzw. beim Erstellen des ersten Projekts erscheinen eventuell Anfragen der Windows-Firewall bezogen auf das Studio und auf Java:





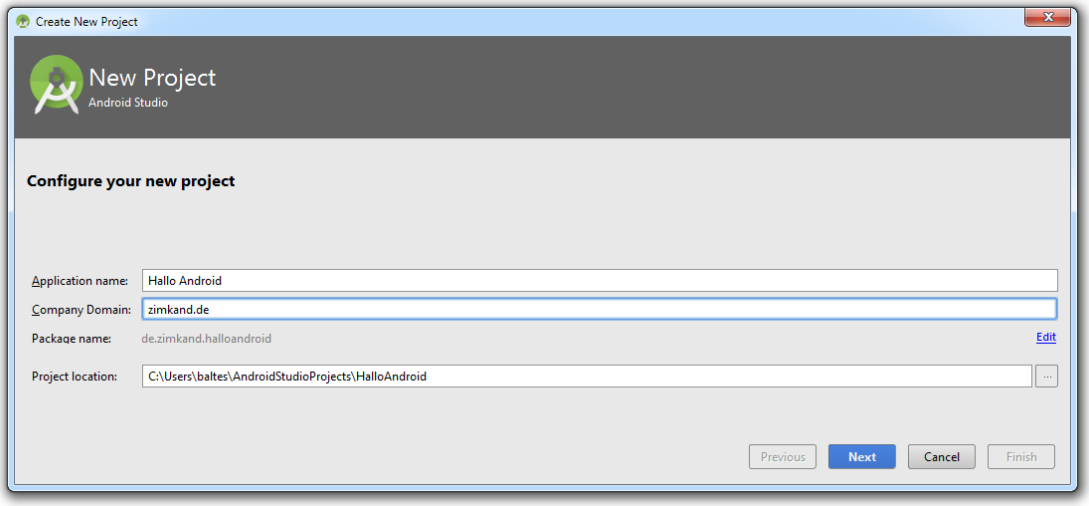
Den Zugriff zu verweigern (alternativlos für Benutzter mit Normalrechten) hat sich bisher *nicht* als nachteilig erwiesen. Wer den **Zugriff zulässt**, geht wohl bei einer Beschränkung auf Domänennetzwerke oder private Netzwerke kein großes Risiko ein und kann seine Entscheidung später über das Systemsteuerungs-Applet der Windows-Firewall revidieren (**Erweiterte Einstellungen > Eingehende Regeln**).

2.4 Die erste Android-App

Um die Funktionstüchtigkeit der Entwicklungsumgebung (inkl. Emulation von Android-Geräten) zu testen, erstellen wir das unvermeidliche *Hallo*-Projekt. 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 u.a. von dem (über **Einstellungen > Apps**) zu startenden 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 müssen 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 **de.zimkand**, 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

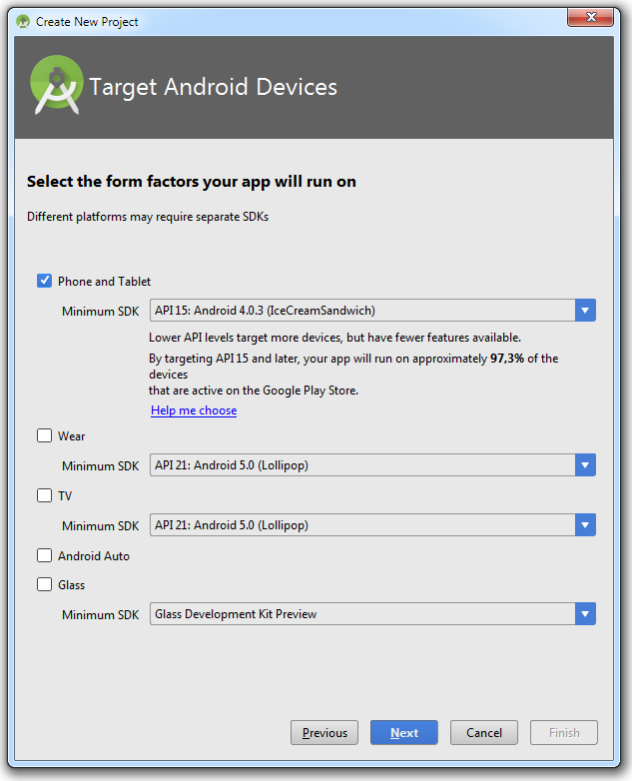
  AndroidStudioProjects

  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:

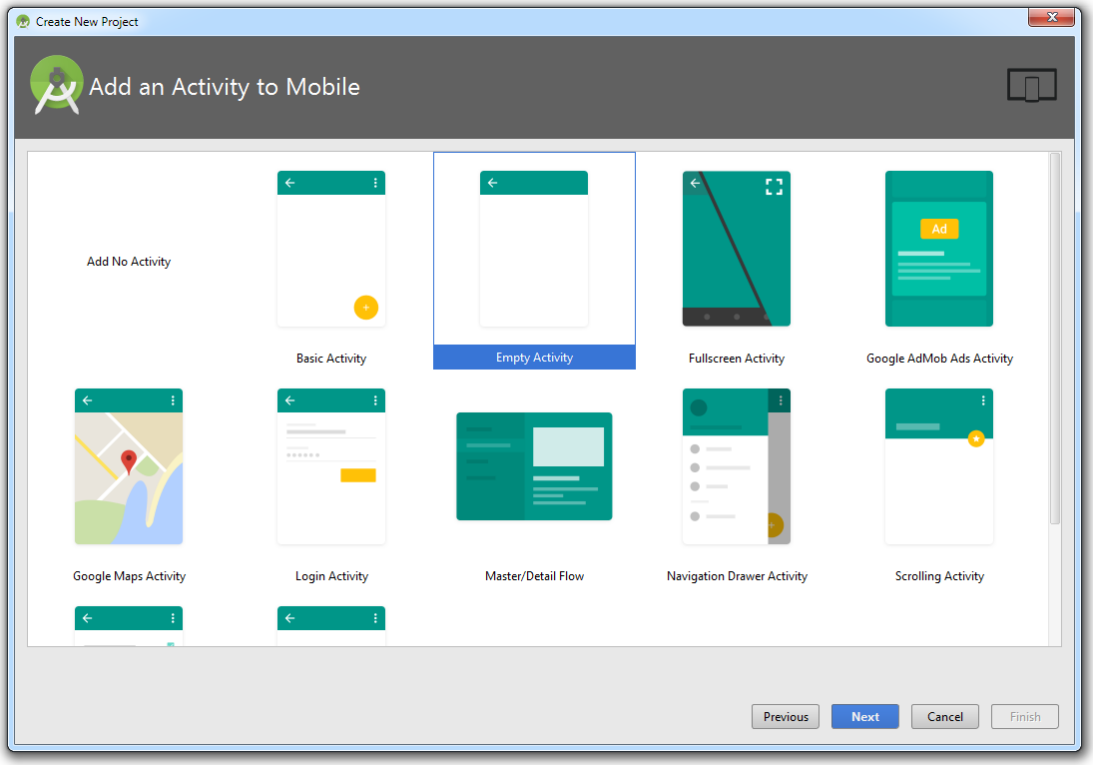
  C:\Users\theo

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

Im nächsten Dialog 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 über 97% der aktuell im Einsatz befindlichen Smartphones und Tablets mit Android unterstützen können:

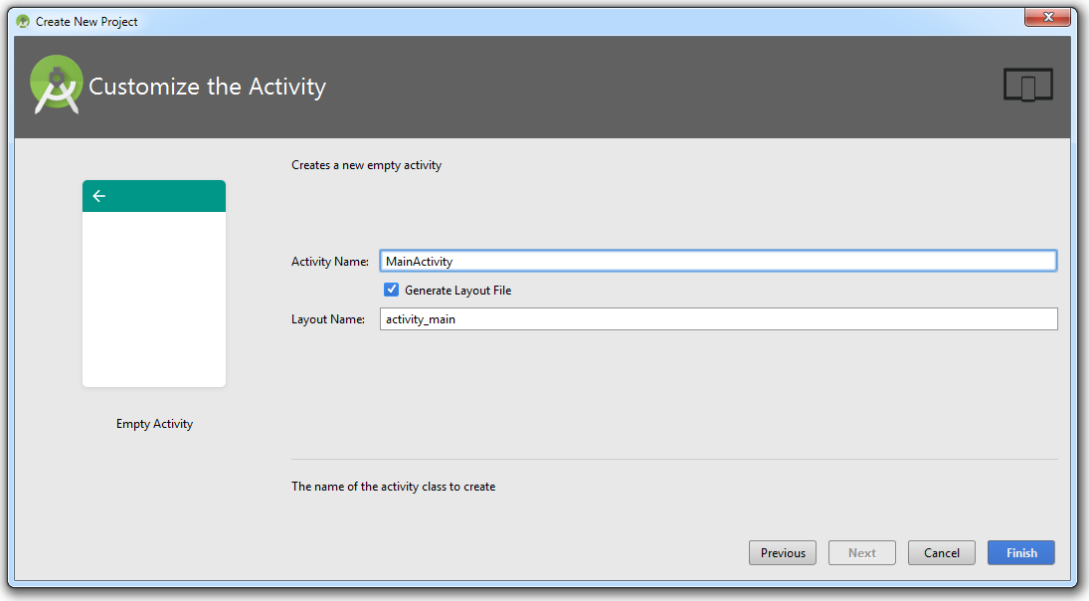


Im nächsten Assistentendialog ist die Voreinstellung **Empty Activity** angemessen:



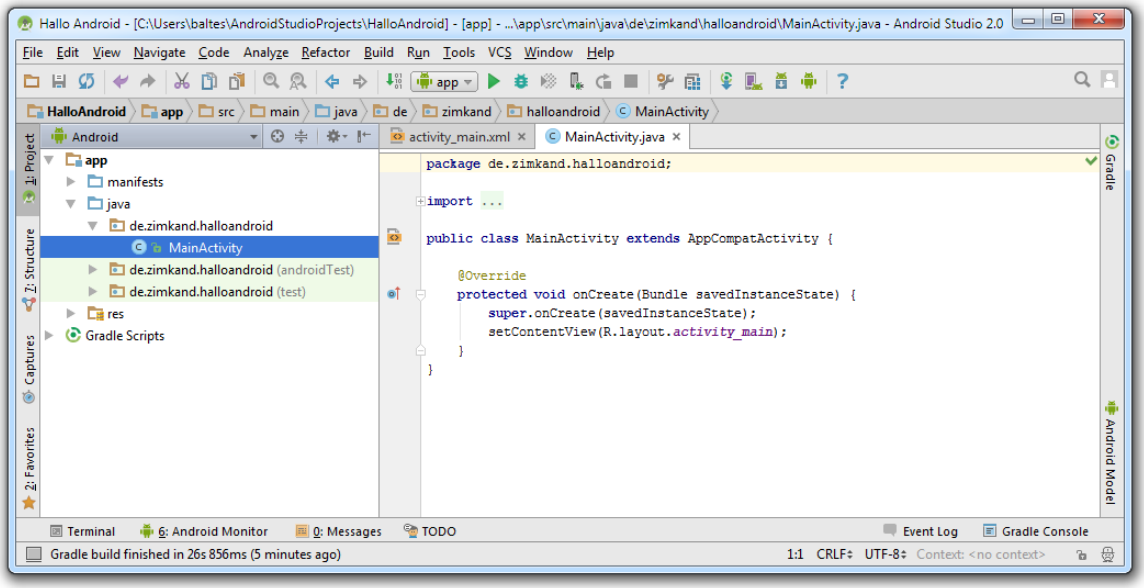
Wir werden uns in Kapitel 4 mit den Activities des Android-Betriebssystems noch ausführlich beschäftigen.

Im letzten Dialog 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**):

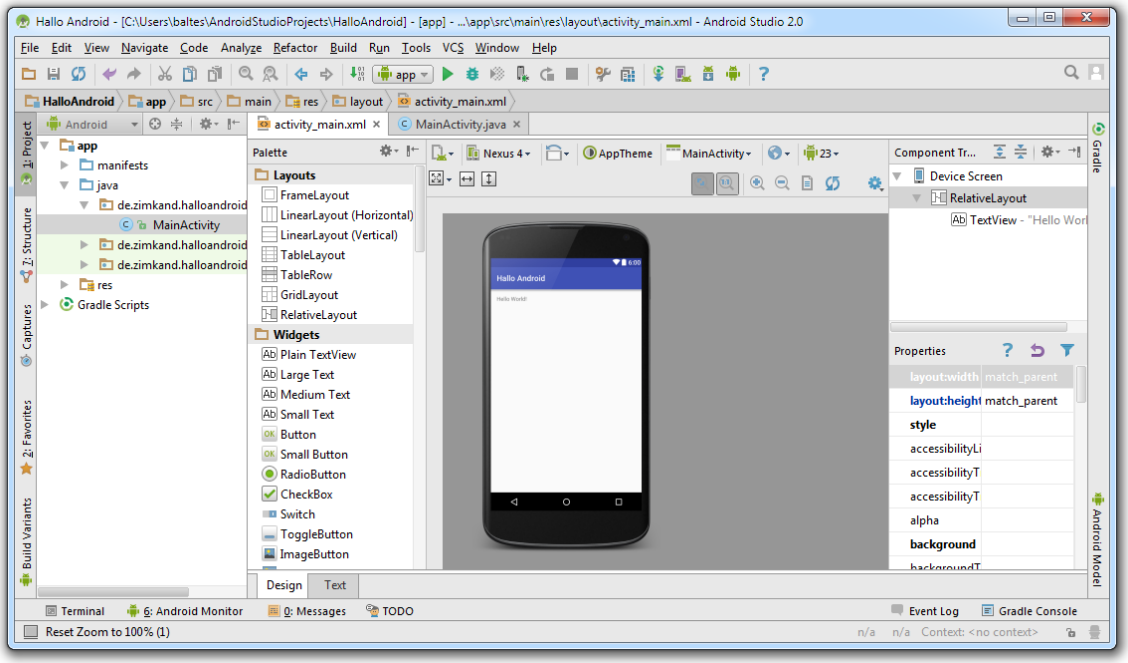


Mit einem Klick auf den Schalter **Finish** veranlassen wir die Erstellung des neuen Projekts, und nach einigen Minuten zeigt das Android Studio den Quellcode der Startaktivität:[1]

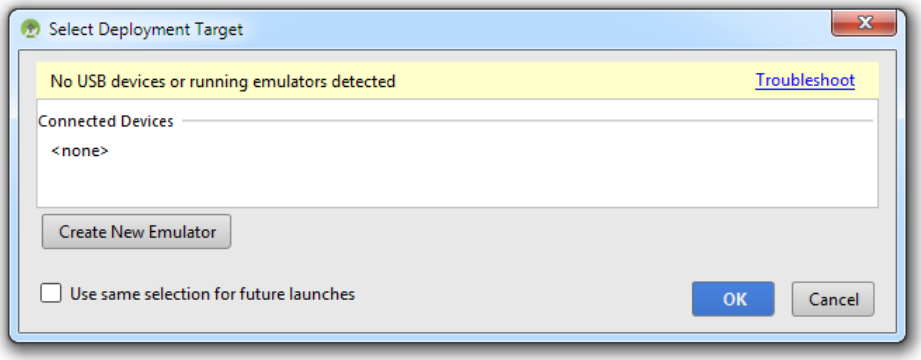
[1] Auf meinem Rechner mit Intel-CPU Core i3 550 dauerte es ca. 3 Minuten.



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:



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 keine virtuellen Geräte definiert sind:



2.5 AVD-Manager

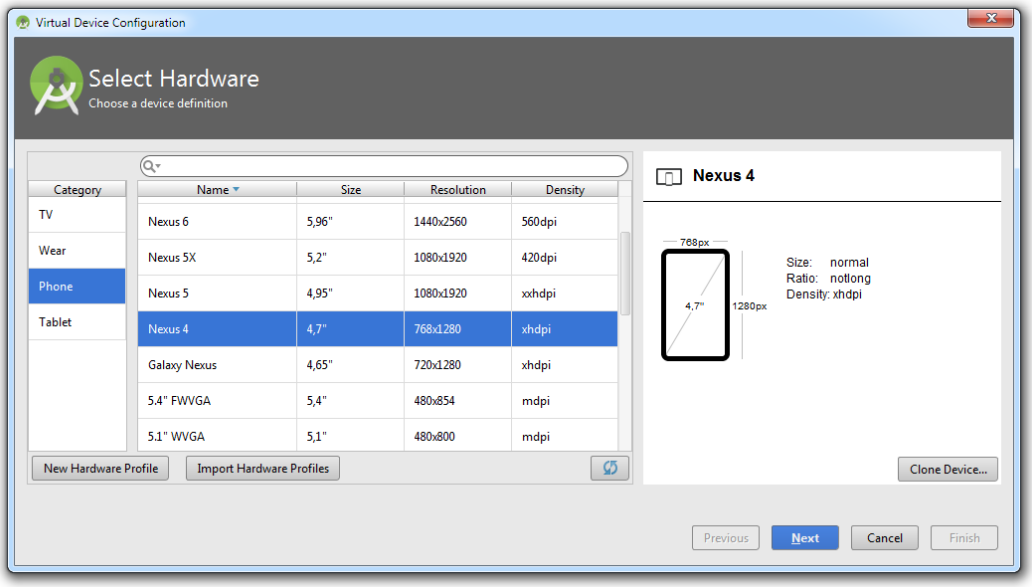
Wir reagieren mit einem Klick auf den Schalter **Create New Emulator**, um per AVD-Manager ein virtuelles Gerät basierend auf dem automatisch installierten x86- System-Image einzurichten.

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 (mit der Telefonie als wichtiger Ausnahme) fast alle Funktionen eines realen Geräts nachahmen.

Den **Android Virtual Device Manager** gibt es in zwei Varianten mit deutlich abweichenden Bedienoberflächen, die folgendermaßen zu starten sind:

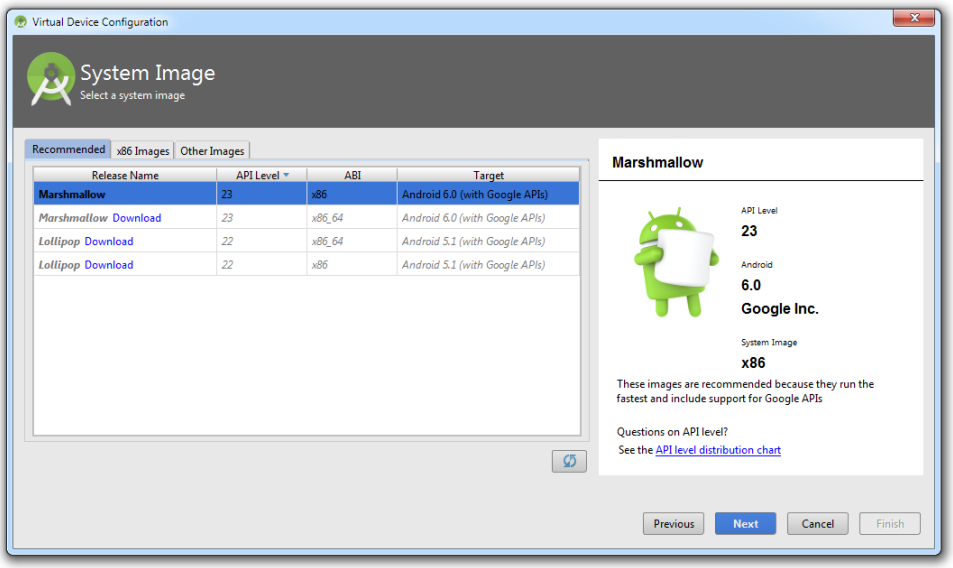
- über die Datei **AVD Manager.exe** im SDK-Installationsordner (z.B. in **C:\Users\theo\AppData\Local\Android\sdk**)
- aus dem Anwendungsfenster Android Studios ...
  - über den Menübefehl **Tools > Android > AVD Manager**
  - über das Symbol  in der Symbolleiste unter der Menüleiste
  - über den Schalter **Create New Emulator** im Dialog **Select Deployment Target**

Wir haben den zuletzt genannten Einstieg in den AVD-Manager verwendet und wählen nun eine relativ bescheidene Hardware für das zu erstellende virtuelle Gerät:

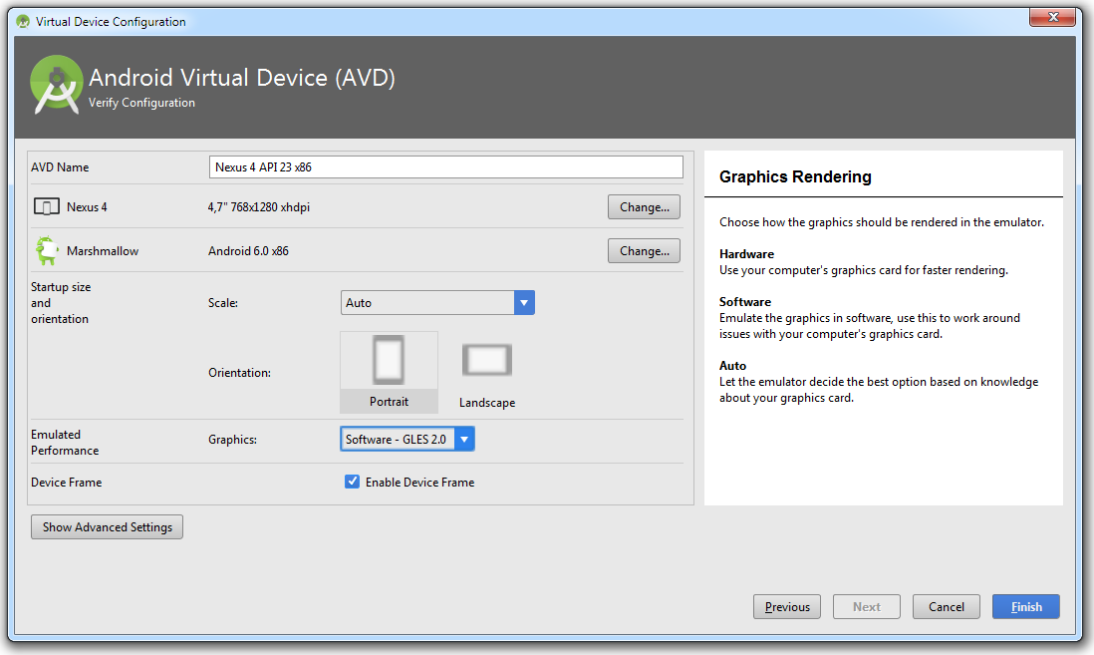


Wenn die GPU (*Graphics Processing Unit*) des Entwicklungsrechners nicht zur Beschleunigung der Grafikausgabe herangezogen werden kann (Hardware-Rendering, siehe unten), dann sollte die Auflösung des emulierten Gerätes nicht über 768 × 1280 hinausgehen.

Als **System Image** akzeptieren wir die Voreinstellung, also das in der SDK-Grundausstattung enthaltene System mit der aktuellen Android-Version 6 (*Marshmallow*, API-Level 23) mit dem **ABI** (*Application Binary Interface*) **x86**:[1]



Anschließend erscheint zum neuen virtuellen Gerät der folgende Einstellungsdialog:



---

[1] Dieses Systemimage (ein Android-System inkl. Linux-Basis) wird bei der in Abschnitt 2.2 beschriebenen Installation auf die Festplatte (oder SSD) befördert und z.B. für den Benutzer `theo` abgelegt i:n

C:\Users\theo\AppData\Local\Android\sdk\system-images\android-23\google_apis\x86

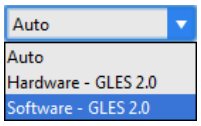
Damit das auf Intel-Technik basierende Image gestartet werden kann, muss der Intel(R) Hardware Accelerated Execution Manager (HAXM) installiert worden sein (vgl. Abschnitt 2.7), was per Voreinstellung nötigenfalls beim ersten Start des Android-Studios passiert (vgl. Abschnitt 2.3).

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.
- **Scale**
  Per Skalierungsfaktor kann man das Fenster eines virtuellen Gerätes z.B. verkleinern:

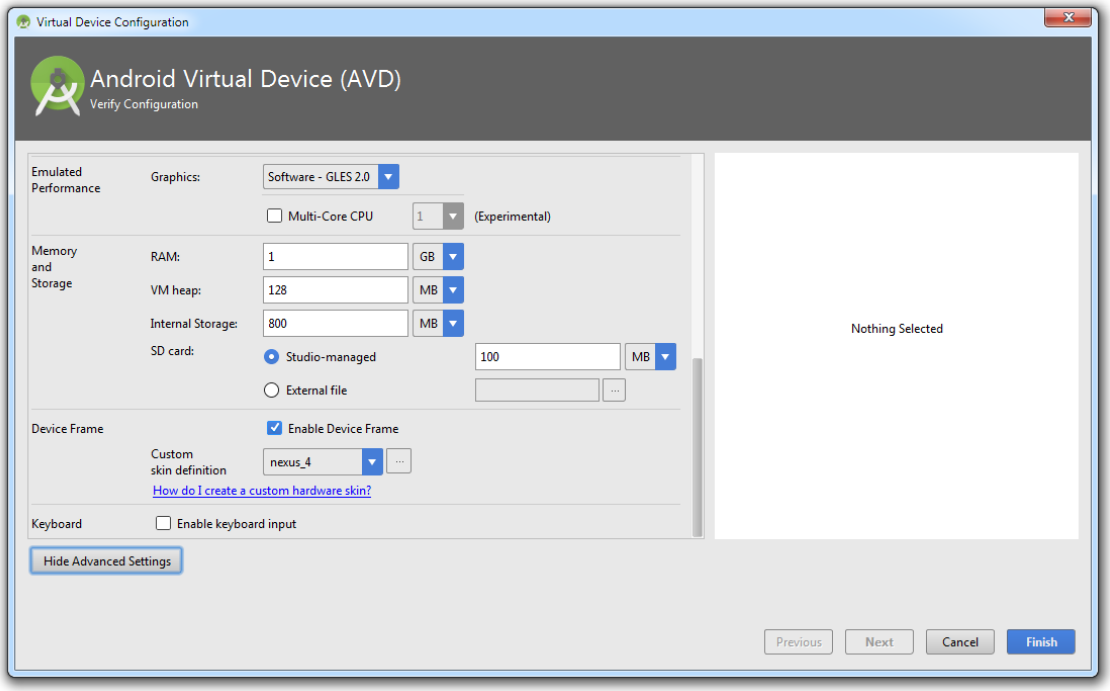
  

  Seit der Android Studio - Version 2.0 hat diese Einstellung an Bedeutung verloren, weil sich die Fenster mit einem emulierten Android-Gerät beliebig in der Größe ändern lassen.
- **Graphics**
  Bei der Grafikausgabe kann man sich zwischen einer Software-Lösung und einer den Grafikprozessor des Wirtsrechners nutzenden Hardware-Lösung entscheiden, oder der Entwicklungsumgebung die Wahl überlassen:

  Wenn der Emulator Startprobleme hat, sollte die Software-Lösung bevorzugt werden.
- **Device Frame**
  Dieses Kontrollkästchen entscheidet darüber, ob im Emulator-Fenster nur die Anzeige, oder auch der Rahmen des nachgeahmten Gerätes erscheinen soll.

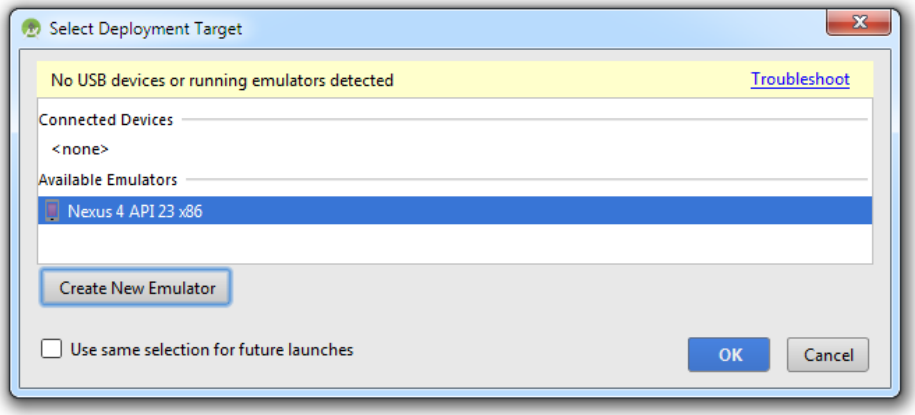
Nach einem Klick auf den Schalter **Show Advanced Settings** erscheinen im Konfigurationsdialog



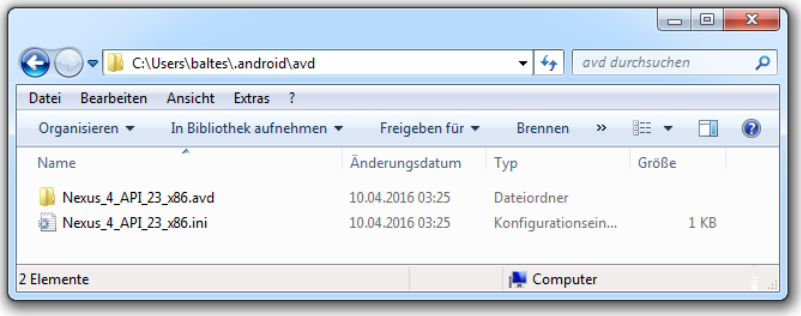
weitere Einstellmöglichkeiten, z.B.:

- **RAM**
  Eine Obergrenze ist die dem Intel(R) Hardware Accelerated Execution Manager (HAXM) eingeräumte Kapazität (vgl. Abschnitt 2.7).
- **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** beseitigen. Dann erscheint bei Benutzung eines Texteingabefelds eine virtuelle Tastatur.

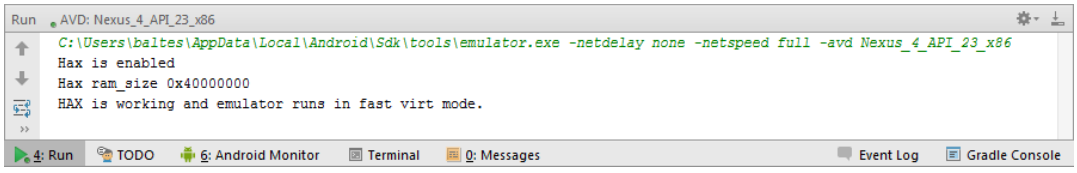
Nun haben wir ein emuliertes Gerät für den Start der ersten Android-App zur Verfügung:



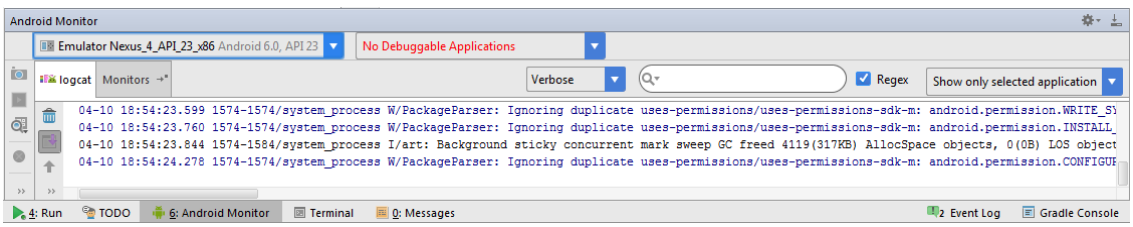
Die virtuellen Geräte werden im Android-Konfigurationsordner abgelegt (vgl. Abschnitt 2.8), z.B.:



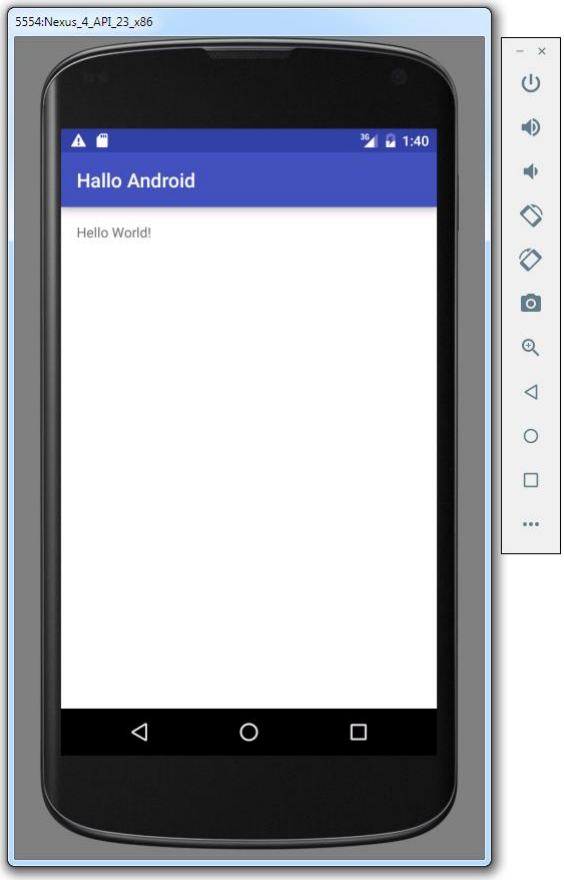
Den Start des Emulators protokolliert das Android Studio im **Run**-Fenster, z.B.:



Wie das virtuelle Android-System auf die Beine kommt, lässt sich im Fenster **Android Monitor** beobachten, z.B.:



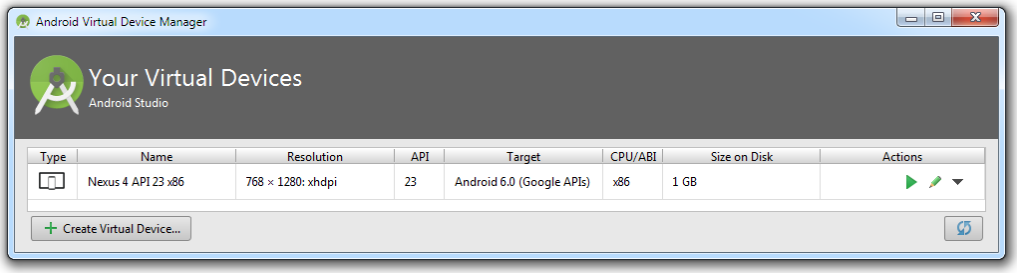
Schließlich erscheint auf dem virtuellen Android-Gerät die *Hallo World* - App:



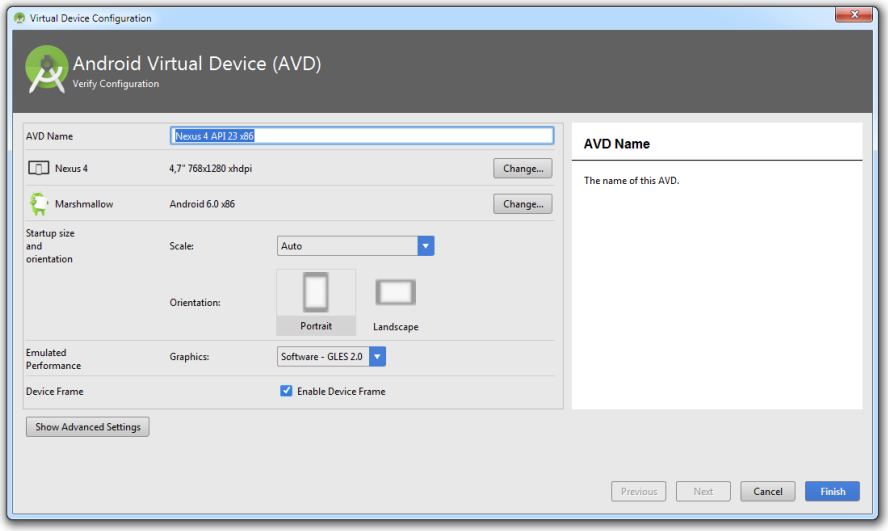
Der allererste Start eines virtuellen Geräts dauert besonders lang, weil umfangreiche Initialisierungsarbeiten verrichtet werden müssen. Aber auch bei einem fertigen virtuellen Gerät können vom Starten bis zur Bedienbarkeit von Android einige Minuten vergehen, wobei die vom Intel Hardware Accelerated Execution Manager (HAXM) unterstützten Geräte deutlich schneller starten. 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.

Seit der Version 2 des Android Studios ist an das Fenster des Emulators eine Palette zur Bedienung des virtuellen Geräts angeheftet (z.B. Orientierungswechsel, Anfertigung eines Bildschirmfotos). Es ist sogar möglich, die Größe des Emulator-Fensters durch das Verschieben eines Fensterrandes zu verändern. Bei einem Orientierungswechsel wird per Voreinstellung die Vordergrundaktivität beendet und neu gestartet, so dass Entwickler testen sollten, wie die eigene App auf Orientierungswechsel reagiert.

Startet man den AVD-Manager aus dem Android Studio z.B. per Mausklick auf das Symbol  in der Symbolleiste unter der Menüleiste, erscheint eine Liste mit den definierten Geräten, z.B.:



Nach einem Klick auf den Schalter erscheint zu einem Gerät der folgende Einstellungsdialog, den wir ober schon kennen gelernt haben:



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 lange vor der ersten App-Erstellung zu starten, z.B. so:

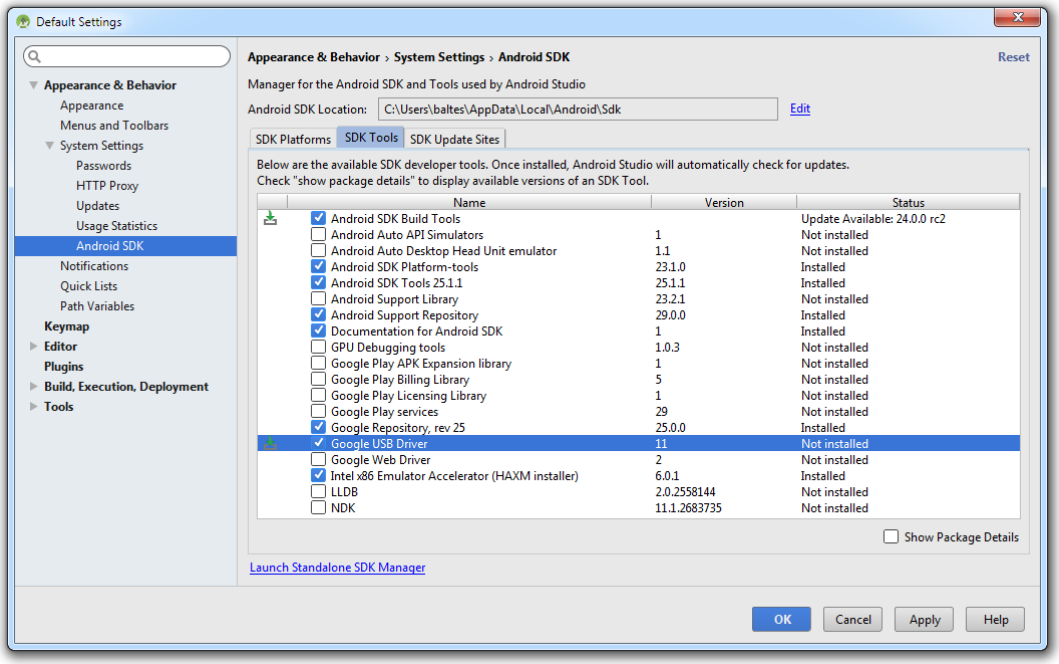
- Aus dem Android Studio per Mausklick auf das Symbol den AVD-Manager starten
- Das gewünschte Gerät per Mausklick auf sein Symbol in der **Actions**-Spalte starten

2.6 SDK-Manager

Mit dem SDK-Manager kann man die Android SDK - Installation überprüfen und modifizieren (z.B. ergänzen oder aktualisieren). Das Programm lässt sich auf unterschiedliche Weise starten, z.B.:

- per Doppelklick auf die Datei **SDK Manager.exe** im SDK-Installationsordner (z.B. in **C:\Users\theo\AppData\Local\Android\sdk**)
- aus dem Anwendungsfenster Android Studios ...
  - über den Menübefehl **Tools > Android > SDK Manager**
  - oder über das Symbol in der Symbolleiste unter der Menüleiste

Wir verwenden anschließend die im Android Studio integrierte Variante des SDK-Managers. Nach den in Abschnitt 2.2 beschriebenen Installationen von Android Studio und Android SDK und den Installationsaktualisierungen beim ersten Start des Android Studios (siehe Abschnitt 2.3) sind auf dem Registerblatt **SDK Tools** folgende Komponenten vorhanden:

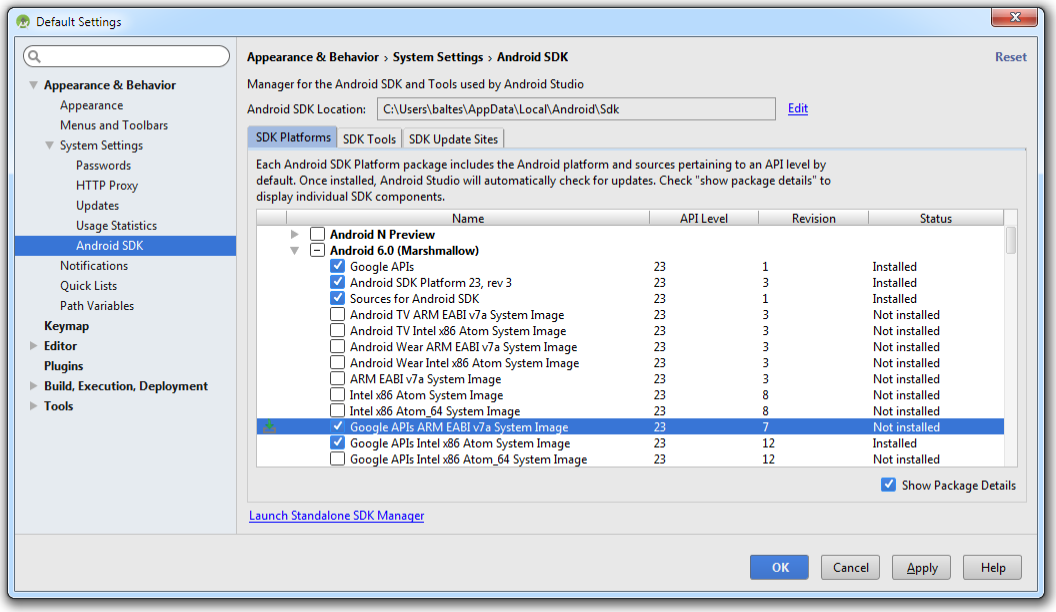


Es ist sinnvoll, die **Android SDK Build Tools** in der Hoffnung auf einen performanteren und störungsarmen Erstellungsprozess auf die Version 24.0.0 rc2 zu aktualisieren. Um aus dem Android Studio oder per ADB-Konsole auf ein per USB-Kabel angeschlossenes Android-Smartphone zugreifen zu können, wird ein USB-Treiber benötigt (siehe Abschnitt 2.10). 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.

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

…\sdk\extras\intel\Hardware_Accelerated_Execution_Manager

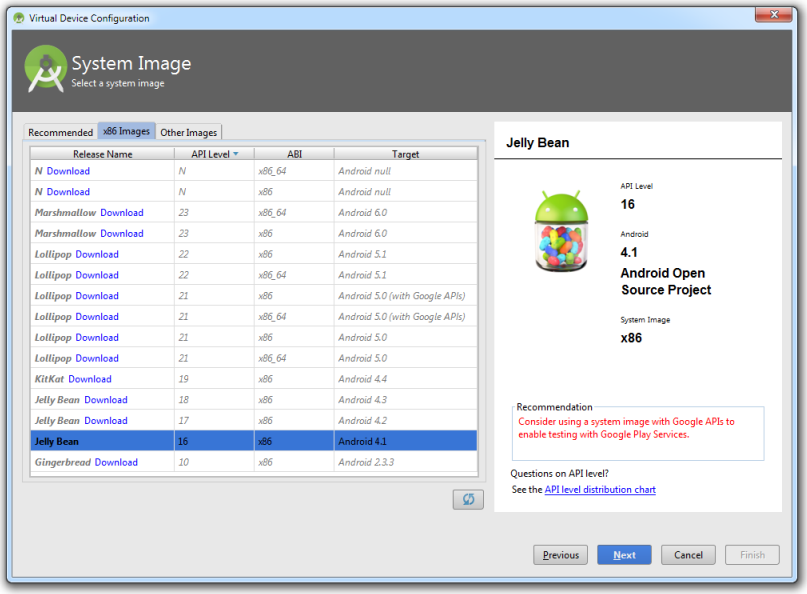
Auf der Registerkarte **SDK Platforms** sollten die **Package Details** eingeschaltet werden:



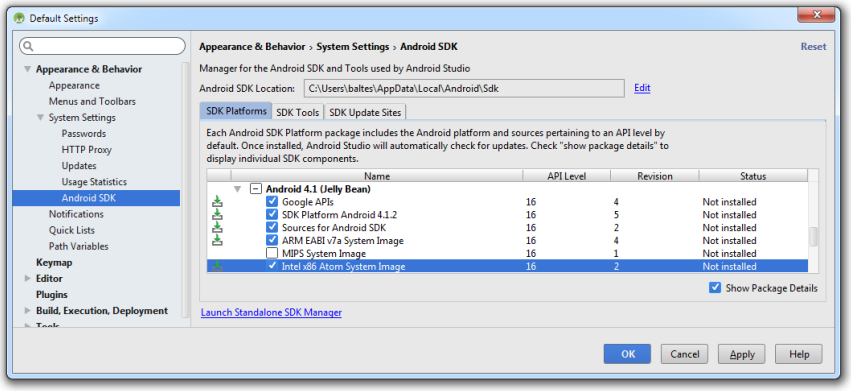
Wenn auf einem Entwicklungsrechner der *Intel(R) Hardware Accelerated Execution Manager (HAXM)* nicht installiert werden kann (vgl. Abschnitt 2.7), so dass kein lauffähiges System-Image als Basis für virtuelle Android-Geräte vorhanden ist, muss ein Image mit der (bei realen Android-

Geräten dominierenden) Technik der britischen Prozessorschmiede ARM (*Advanced RISC Machines*) installiert werden (mit oder ohne Google-APIs). Es wird ohne Hardware-Beschleunigung (also weniger performant) ausgeführt, benötigte aber keine spezielle Prozessortechnologie. 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 das Android 4.0.3 (API-Level 15) benötigen. Weil dieses System relativ wenig Massen- und Hauptspeicher belegt sowie relativ schnell startet, bietet es sich an, die Bibliotheken, Quellen und System-Images (in ARM- und Intel-Technik) zu Android 4.0.3 zu installieren. Die aus dem Android Studio zu startende Variante des AVD-Managers unterstützt allerdings bei den angebotenen **System Images** aus unerfindlichen Gründen das API-Level 15 *nicht* (weder auf x86- noch auf ARM-Basis):[1]

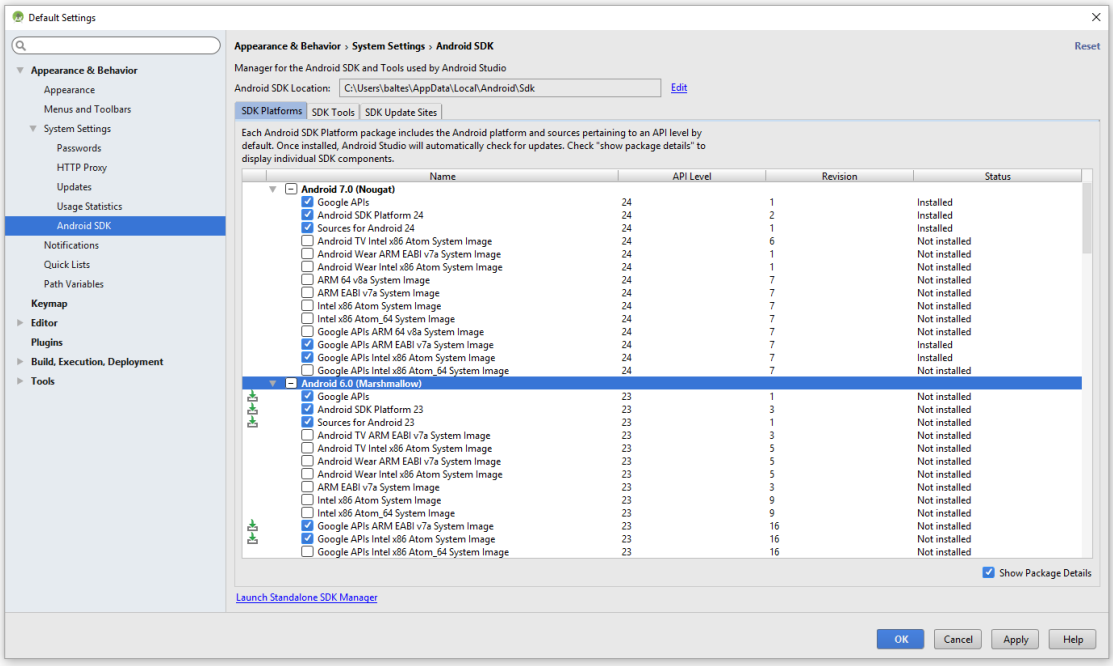


Daher installieren wir per SDK-Manager das API-Level 16 als „Minimalsystem“:

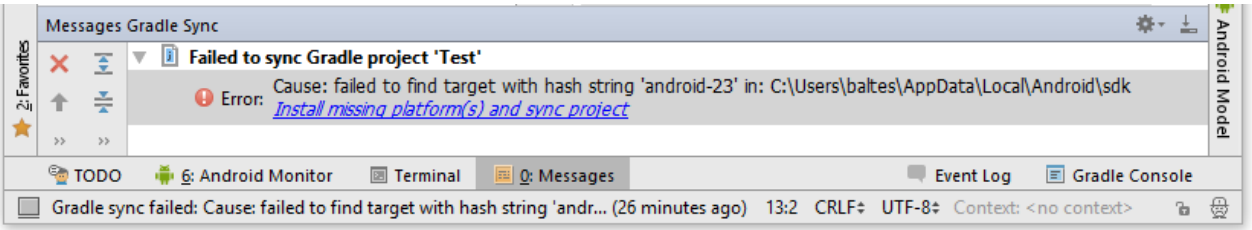


Wenn Sie die Beispielprojekte zum Manuskript problemlos öffnen möchten, sollten Sie per SDK-Manager auch die Plattform Android 6 (Marshmallow, API-Level 23) installieren, z.B. mit der folgenden Ausstattung (zusätzlicher Platzbedarf: leider ca. 7 GB):

[1] Beobachtet mit der Version 2.0 des Android Studios am 16.04.2016



Per Voreinstellung wird nur die SDK-Plattform zur jüngsten Android-Version installiert, und beim Öffnen der Manuskript-Projekte ist die folgende Fehlermeldung zu sehen:



Man kann die Situation durch das Korrigieren von Versionsangeben in der Datei

...\app\gradle.build

beheben, oder der Einfachheit halber die SDK-Plattform 23 installieren.[1] Weitere Hinweise zur Verwendung der Kursprojekte folgen in Abschnitt 3.9.

2.7 Intel(R) Hardware Accelerated Execution Manager (HAXM)

Die aktuelle Version des Intel(R) Hardware Accelerated Execution Managers kann über die folgende Webseite

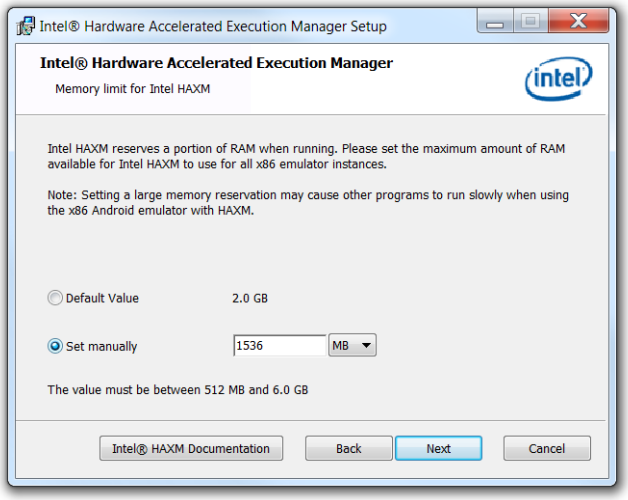
https://software.intel.com/en-us/android/articles/intel-hardware-accelerated-execution-manager

oder mit Hilfe des SDK-Managers (vgl. Abschnitt 2.6) herunter geladen und dann installiert werden. Um die interaktive Installation zu starten, setzt man einen Doppelklick auf die herunter geladene Datei **intelhaxm-android.exe**. Dabei sollte der Installationsrechner mit dem Internet verbunden sein, damit bei Bedarf Updates bezogen werden können.

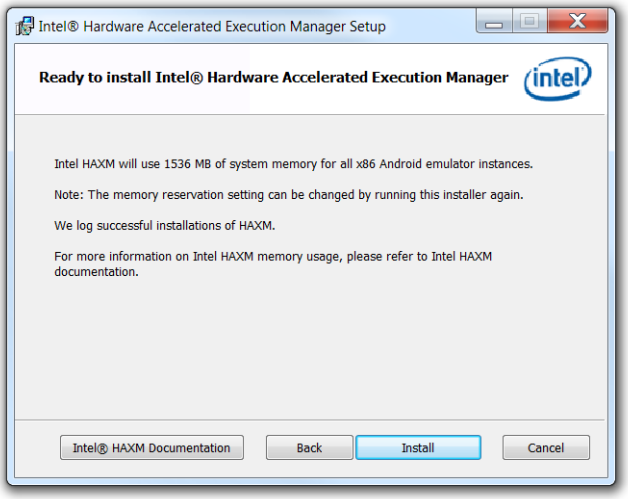
[1] Leider war die bequeme Weitergabe von Projekten an andere Entwickler kein vorrangiges Designziel bei der Konzeption des Android Studios.



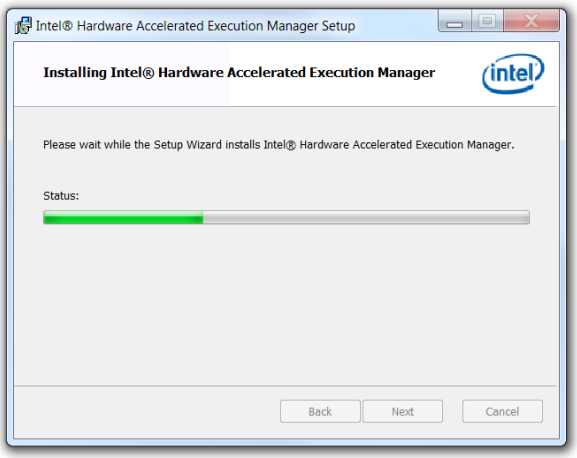
Der vorgeschlagene Hauptspeicherumfang für die Emulation richtet sich nach der vorgefundenen RAM-Ausstattung auf dem Wirtsrechner. Während die Emulation läuft, steht die reservierte Speichermenge für andere Anwendungen nicht zur Verfügung. Mit 1,5 GB sollten auch aktuelle Android-Versionen zufrieden sein:

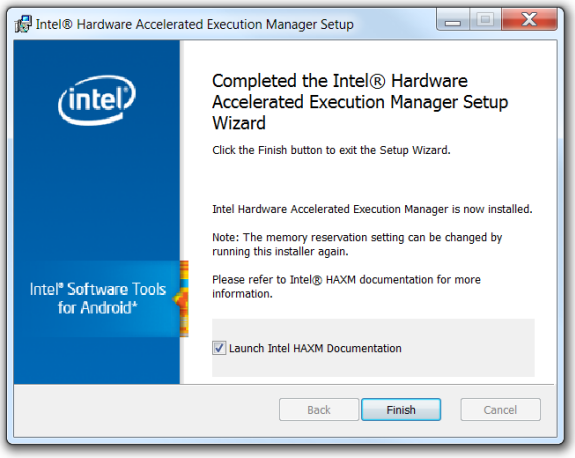


Der nächste Installationsdialog informiert darüber, dass sich die Hauptspeicherausstattung des Emulators durch eine Wiederholung der Installation verändern lässt:



Nach einer erfolgreichen Installation





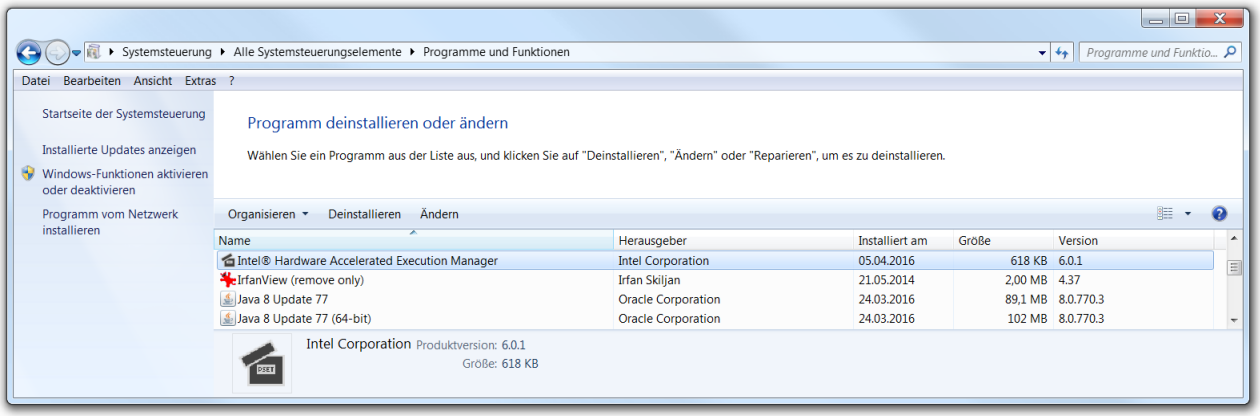
wird optional in einem Browser-Fenster die folgende Webseite geöffnet:

https://software.intel.com/en-us/android

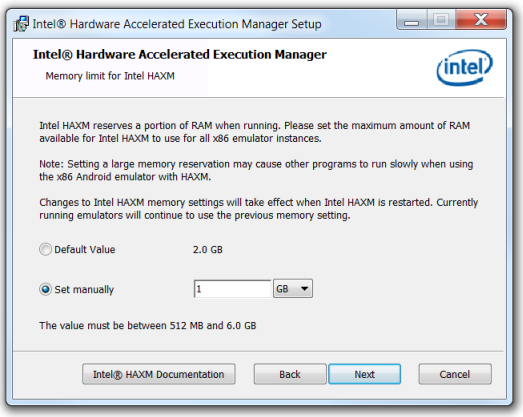
Die Installation landet im Ordner:

C:\Program Files\Intel\HAXM

Um die RAM-Ausstattung der Hardware-beschleunigten Emulation zu ändern, öffnet man das Windows-Systemsteuerungs-Applet **Programme und Funktionen**, markiert den Intel **Hardware Accelerated Execution Manager** und klickt auf **Ändern**:



Nach **Next** und **Change** kann man im folgenden Dialog den gewünschten neuen Wert eintragen:



Dass die HAXM-Version 6.0.1 unter Windows nur durch den zuerst angemeldeten Benutzer seit dem letzten Booten störungsfrei zu benutzen ist, fällt auf einem Personal Computer kaum auf.

2.8 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 **Fehler! Verweisquelle konnte nicht gefunden werden.** 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 landen in:

  C:\Users\theo\.AndroidStudio1.5

- Die Projekte landen in:

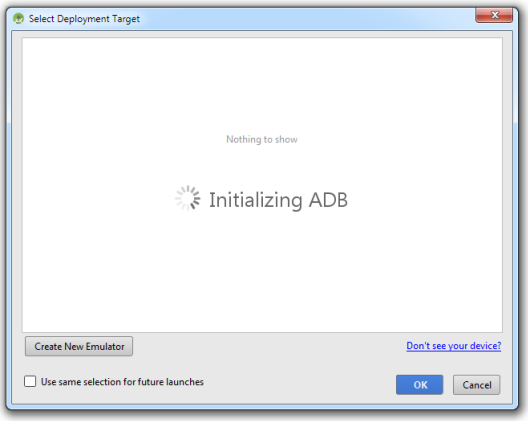
  C:\Users\theo\AndroidStudioProjects

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

  C:\Users\theo\.gradle

2.9 Android Debug Bridge

Über die Android Debug Bridge (ADB) kann man vom Entwicklungsrechner aus mit einem emulierten oder mit einem per USB angeschlossenen Android-Gerät kommunizieren. Wenn beim Starten der ersten Emulatorinstanz die Server-Komponente der Android Debug Bridge noch nicht läuft, wird sie gestartet, z.B. vom Android Studio:



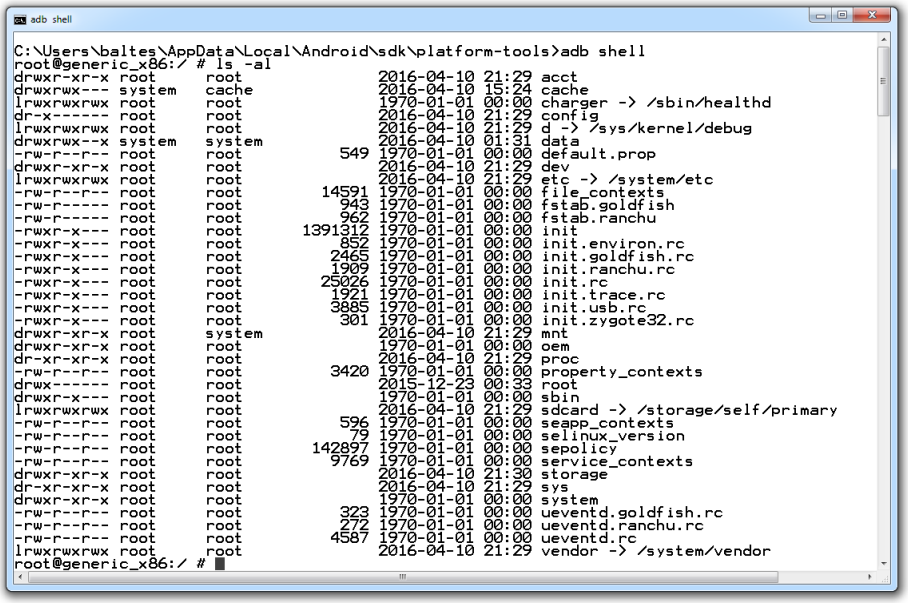
Unter Windows wird die Datei **adb.exe** gestartet, die im folgenden SDK-Unterordner zu finden ist:

<sdk>/platform-tools/

Die wesentlichen ADB-Bestandteile sind:[1]

[1] Weitere Hinweise sind hier zu finden: http://developer.android.com/tools/help/adb.html

- Ein Klient
  Er läuft auf der Entwicklungsmaschine. Mit dem Kommando **adb shell** lässt sich z.B. eine Konsolenverbindung zur ersten aktiven Emulatorinstanz aufbauen:

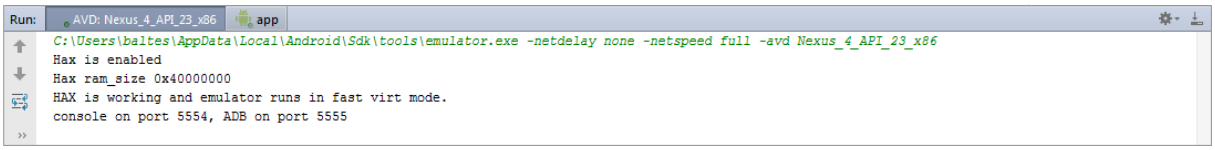


  Auch der im Android Studio enthaltene Android Monitor, der unter anderem das **logcat**-Fenster der Entwicklungsumgebung mit Statusmeldungen aus dem Android-Gerät versorgt und andere Diagnosehilfen bietet (siehe Abschnitt 5.3.1), setzt einen ADB-Klienten ein.
- Der Geräte-Dämon
  Auf jedem emulierten oder per USB angebundenen Android-Gerät läuft im Hintergrund ein als *ADB-Dämon* bezeichneter Prozess, der die Anforderungen der Klienten bedient.
- Der Server
  Er läuft als Hintergrundprozess auf der Entwicklungsmaschine, lauscht am TCP-Port 5037 und kümmert sich um die Kommunikation zwischen den Klienten und den ADB-Dämonen auf den Android-Geräten.

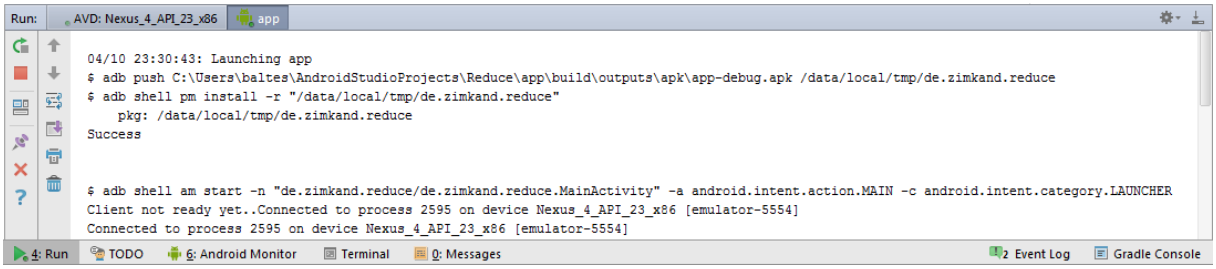
Emulatorinstanzen wählen für die Kommunikation mit dem ADB-Server einen TCP-Port mit einer ungeraden Nummer im Bereich ab 5555. Außerdem belegen sie für Konsolenverbindungen noch einen TCP-Port mit der nach unten nächstgelegenen geraden Zahl. Folglich verwendet die erste Emulatorinstanz die beiden folgenden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

5554 für die Konsole
5555 für die Kommunikation mit dem ABD-Server

So verhält sich auch das virtuelle Android-Gerät, über dessen Start das folgende **Run**-Fenster der Entwicklungsumgebung berichtet:



Auf der **app**-Registerkarte des **Run**-Fensters wird dokumentiert, welche Aktionen das Android Studio unternimmt, um eine App zu installieren und zu starten, z.B.:



2.10 Reales Android-Gerät 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:

http://developer.android.com/tools/device.html

2.10.1 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 **Telefoninfo**).
- Tippen Sie siebenmal auf die **Build-Nummer**.
- Wechseln Sie zurück zur Hauptebene der Einstellungs-App, wo nun die **Entwickleroptionen** zu finden sind.

Ab Android 4.2.2 wird aus Sicherheitsgründen einem fremden Rechner die Kooperation mit dem ADB-Dämon (vgl. Abschnitt 2.9) nur dann erlaubt, wenn die entsprechende Nachfrage auf dem Android-Gerät positiv beantwortet wird. Die Wirksamkeit dieser Schutzmaßnahme hängt natürlich von der Qualität des Sperrbildschirmpassworts ab.

Beim USB-Debugging kann auch der eingestellte **USB-Verbindungstyp** des Android-Geräts eine Rolle spielen.

2.10.2 Android USB-Treiber installieren

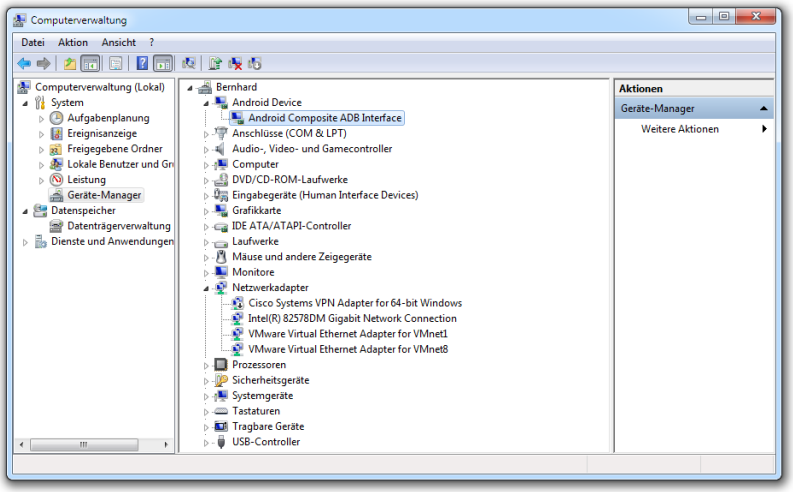
Auf einem Windows-Rechner muss zu einem Android-Gerät ein USB-Treiber installiert werden, damit das Android Studio zu testende Software per ADB (Android Debug Bridge) auf das Gerät schicken kann. Für ein Google-Smartphone (z.B. Nexus 5) eignet sich der per SDK-Manager verfügbare USB-Treiber (vgl. Abschnitt 2.6). 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

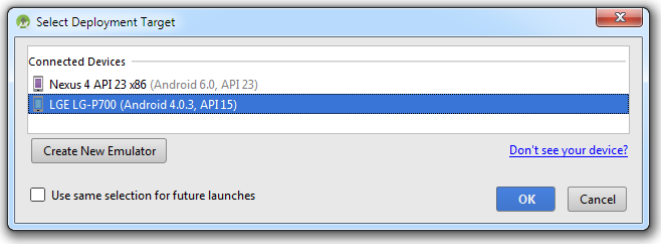
Zum Installieren kann man so vorgehen:

- Android-Gerät per USB-Kabel anschließen, was zu einer unvollständigen Treiberinstallation führt
- Geräte-Manager starten (**Systemsteuerung > System > Geräte-Manager**)
- Aus dem Kontextmenü zum Android-Gerät wählen: **Treibersoftware aktualisieren**
- Im nächsten Dialog antworten: **Auf dem Computer nach Treibersoftware suchen**
- Den oben angegebenen Ordner mit dem USB-Treiber angeben und den weiteren Empfehlungen folgen

Als Ergebnis sollte der erfolgreich installierte Treiber in der Geräteverwaltung erscheinen, z.B.:



Ein per USB angeschlossenes Android-Gerät kann beim App-Start als Ziel gewählt werden, z.B.:



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

http://developer.android.com/tools/extras/oem-usb.html

2.11 Übungsaufgaben zu Kapitel 2

1) Führen Sie die in Kapitel 1 beschriebenen Installationen auf Ihrem eigenen Rechner aus.

3 Android Studio kennen lernen

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

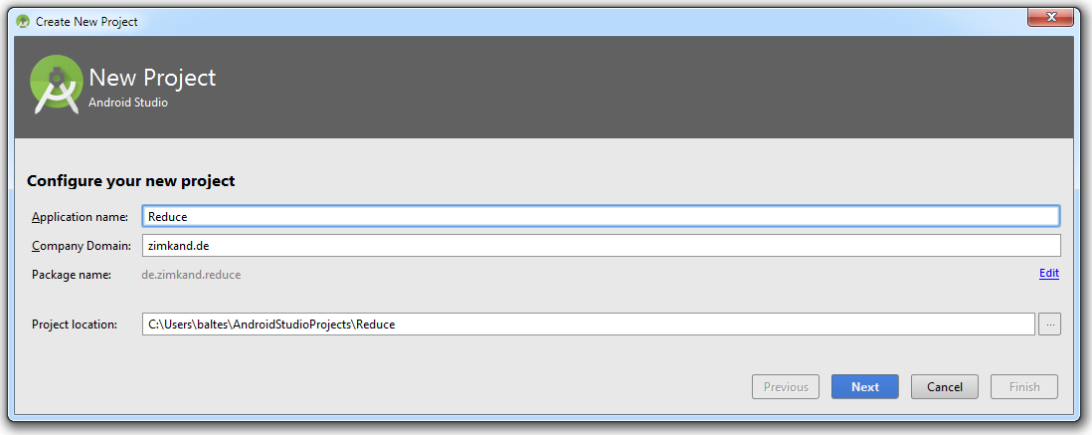
Weil im weiteren Verlauf des Manuskripts sehr oft vom *Android Studio* die Rede sein wird, vereinbaren wir der Kürze halber die nette Abkürzung AS, sprechen also z.B. von *AS-Projekten*.

3.1 Projekt anlegen

Wir legen gemäß der Beschreibung in Abschnitt 2.4 ein neues Projekt an, starten also z.B. bei aktivem Android Studio mit dem Menübefehl

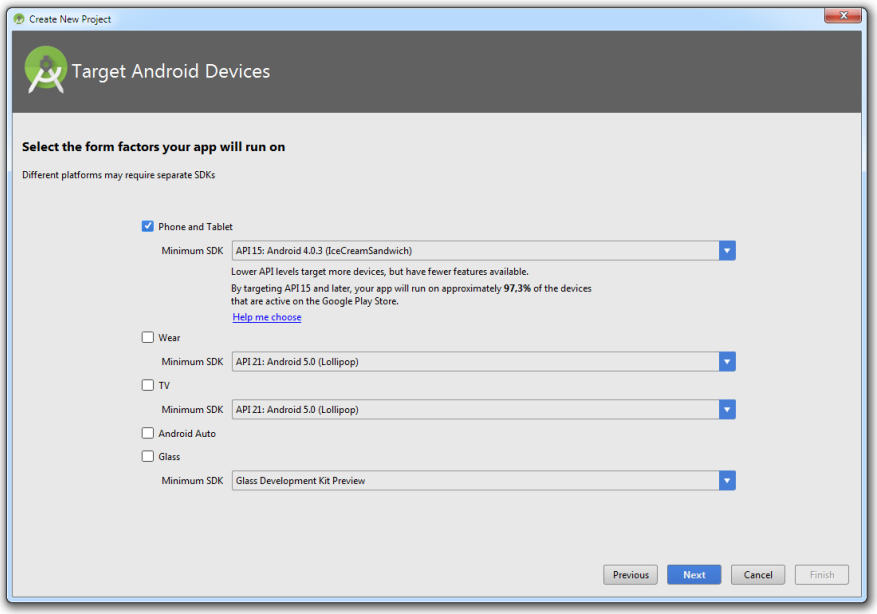
File > New Project

Im ersten Assistentendialog

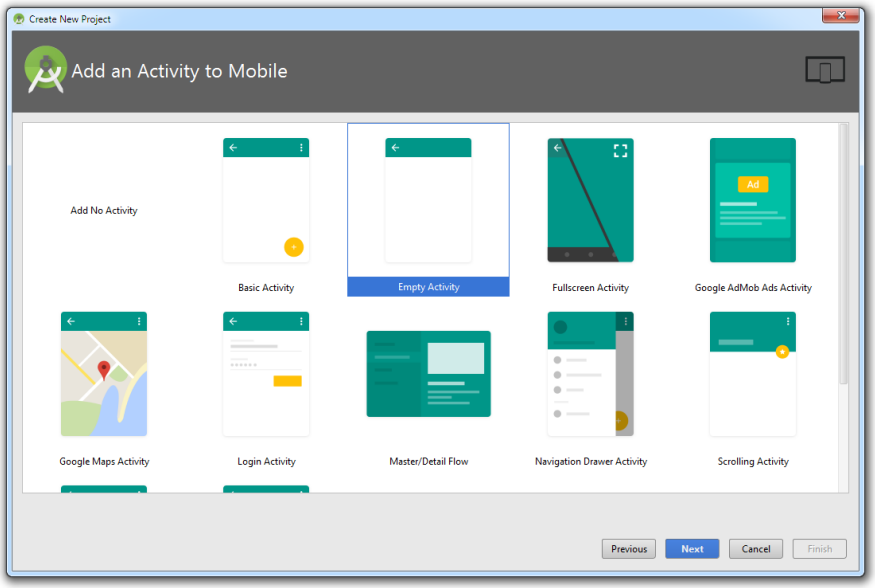


wählen wir `Reduce` als **Application name** und eine erfundene **Company Domain** namens `zimkand.de`.

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:

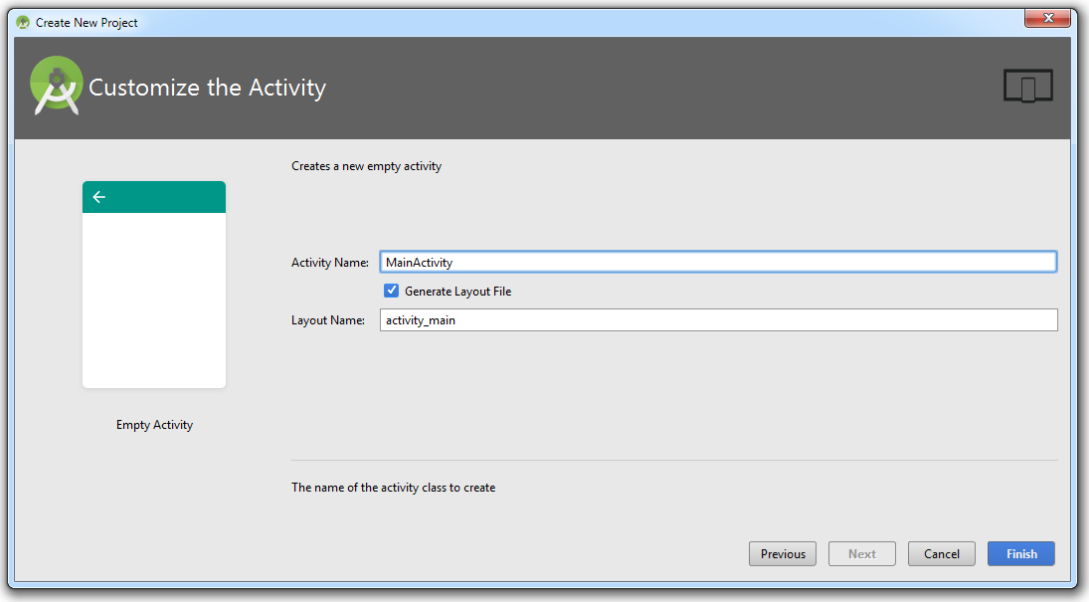


Im nächsten Dialog akzeptieren wir den Vorschlag des Assistenten, in der neuen App gleich eine Activity (eine Bildschirmseite) anzulegen. Weil wir uns um deren Bedienelemente selbst kümmern wollen, behalten wir die Voreinstellung **Empty Activity** bei:

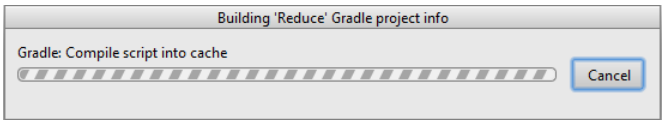


Im letzten Dialog des Assistenten akzeptieren wir ...

- 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.

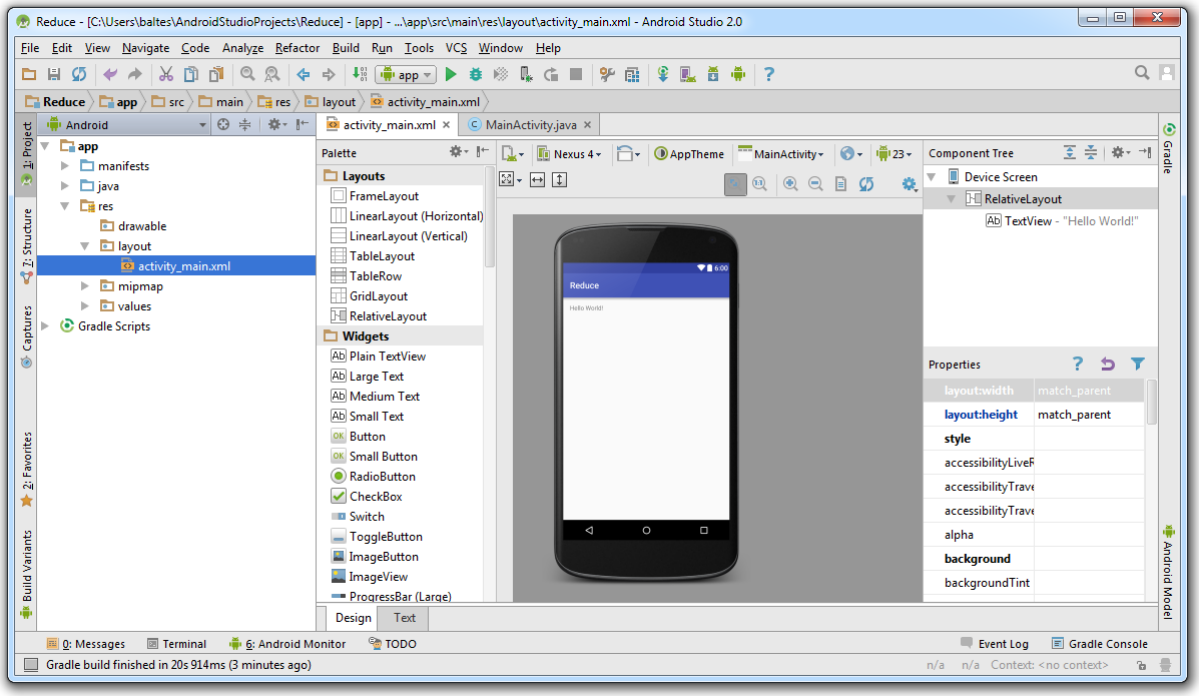


Mit einem Klick auf den Schalter **Finish** veranlassen wir die Erstellung des neuen Projekts. Dabei wird ein so genanntes **Build-System** tätig, das aus Quellcodedateien sowie Ressourcen-Dateien (z.B. im XML-Format zur Beschreibung der Bedienoberfläche einer Aktivität) schlussendlich die **apk**-Datei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



Gradle ist Skript-basiert, und wir könnten das von Google erstellte Skript **build.gradle** ändern, z.B. um schon jetzt die Nutzung der neuen Compiler-Generation („Jack & Jill“) zu erzwingen (vgl. Abschnitt 1.1). In der Regel werden wir aber nicht in den Build-Prozess eingreifen.

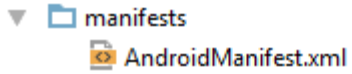
Das neue Projekt wird vom Android Studio in einem eigenen Fenster präsentiert (hier bei aktiver Registerkarte mit der Layout-Definition):



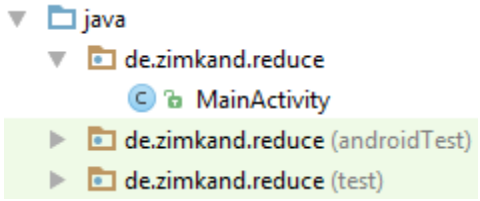
Zunächst macht auch unsere zweite App nichts anderes, als den Text „Hello world!“ zu zeigen.

Im **Project**-Fenster 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.

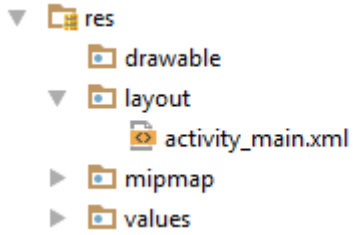
Das Drop-Down - Menü zum **Project**-Fenster im Android Studio bietet mehrere Sichten, von denen für uns nur die beiden folgenden von Bedeutung sind:

- 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:
    

  - **java**
    Hier befinden sich nach Paketen geordnet die Quellcodedateien zu den Klassen 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):

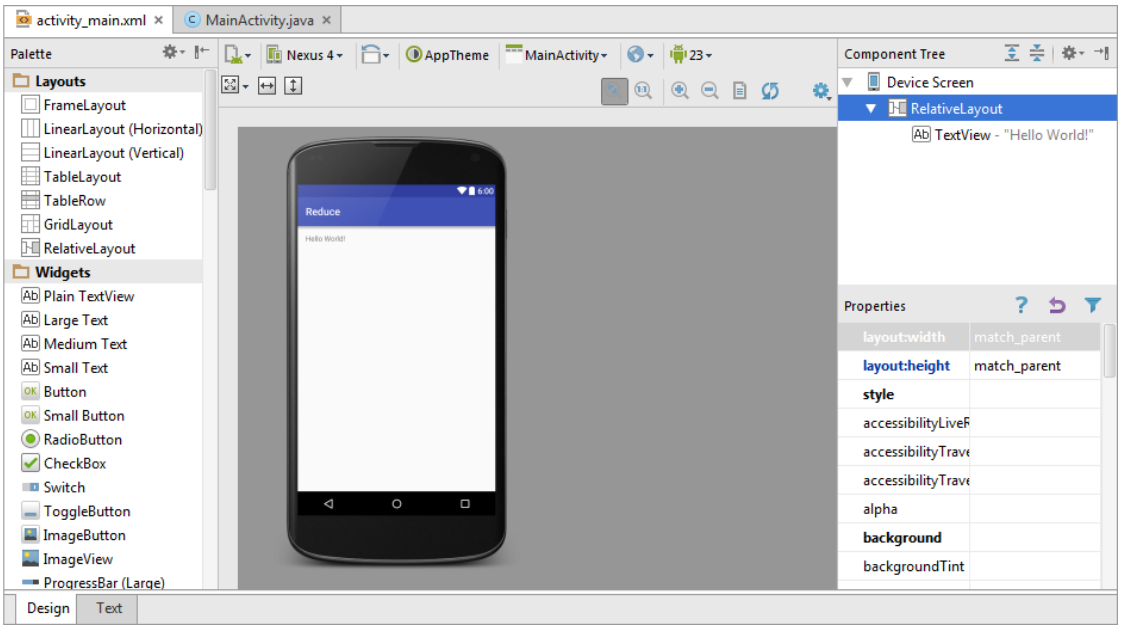


- In der alternativen Sicht **Project** ist die Ordnerstruktur des Dateisystems zu sehen.

In der **Android**-Sicht sind auf der Hauptebene des **Project**-Fensters neben den Modulen auch die **Gradle-Scripts** zu sehen, über die das Gradle-Erstellungssystem aus den Bestandteilen einer App (Manifeste, Quellcode, Ressourcen) eine **APK**-Datei (Android Package) erstellt, die auf den Zielgeräten installiert werden kann. Wir werden kaum Eingriffe in das Build-System vornehmen.

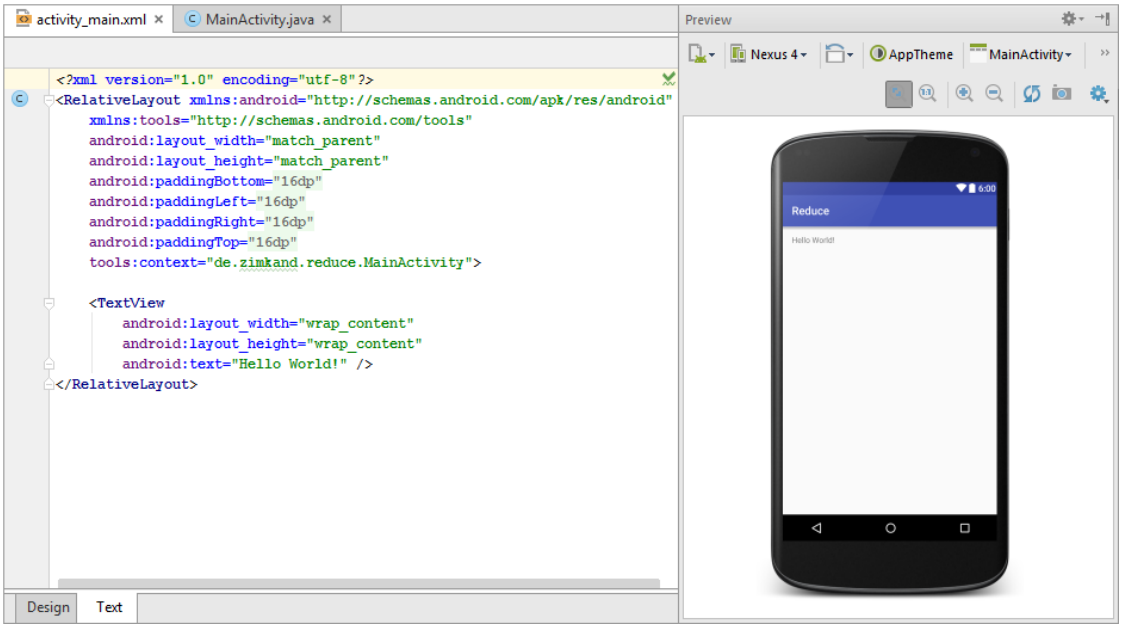
3.2 Bedienoberfläche entwerfen

Das **Project**-Fenster 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 der **Design**-Modus aktiv:



Bei Bedarf können Sie diesen Zustand per Doppelklick auf die Datei **activity_main.xml** herstellen.

Manche Änderungen der GUI-Definition lassen sich allerdings 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 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

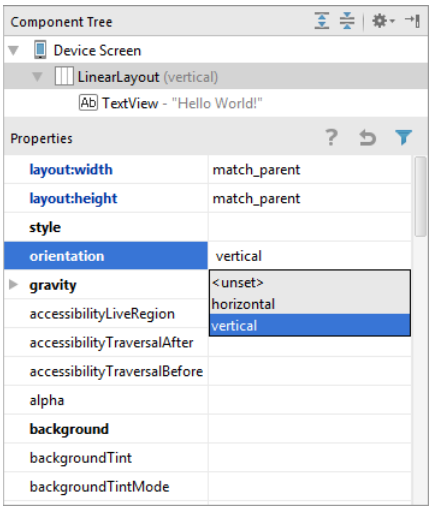
Wie der **Component Tree** der Entwicklungsumgebung zeigt (in der **Design**-Ansicht rechts oben), ist als „Platzanweiser“ für die Steuerelemente (Texteingabefelder, Label, Schalter) ein Objekt der Klasse **RelativeLayout** aktiv. Es ist leistungsfähig, aber etwas knifflig, so dass wir uns den Einstieg erleichtern durch den Wechsel zu einem Layout-Manager der Klasse **LinearLayout** mit

vertikaler Anordnung der Steuerelemente. Wechseln Sie zum **Text**-Modus des Editors, und ersetzen Sie im XML-Code den Elementnamen **RelativeLayout** durch die Alternative **LinearLayout**:

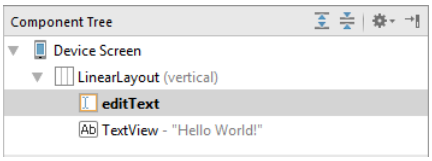
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin"
    tools:context="de.zimkand.reduce.MainActivity">

    <TextView
        android:text="Hello World!"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</LinearLayout>
```

Nach der Rückkehr zum **Design**-Modus ist im **Component Tree** - Fenster der neue Platzanweiser zu sehen, und im **Properties**-Fenster kann man dem **LinearLayout** eine vertikale Orientierung verpassen:[1]



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

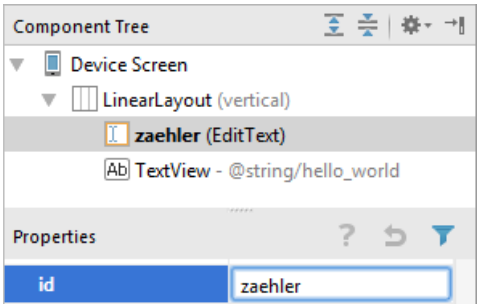


Hier sollen die Benutzer den Zähler des zu kürzenden Bruchs eintragen.

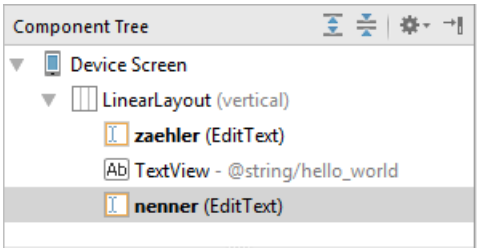
Die Steuerelemente werden im Programm über eine Kennung angesprochen, so dass eine aussagekräftige Benennung zu einem besser lesbaren Quellcode führt. Tragen Sie daher bei markiertem Texteingabefeld im Feld **id** des **Properties**-Fensters den Namen `zaehler` ein:

---

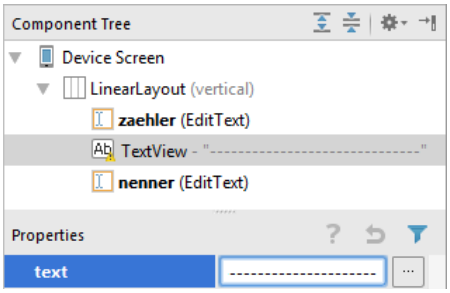
[1] Jetzt ist übrigens im Kontextmenü zum Platzanweiser das Item **Morphing** vorhanden und erlaubt z.B. eine bequeme Rückkehr zum Typ **RelativeLayout**.



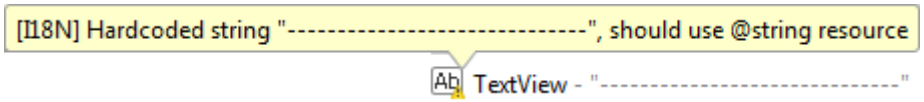
Fügen Sie unter dem vorhandenen **TextView**-Objekt ein weiteres Steuerelement vom Typ **Number (Signed)** ein, und vergeben Sie den Namen `nenner`, so dass im Fenster **Component Tree** der folgende Stand resultiert:



Markieren Sie im Fenster **Component Tree** das **TextView**-Objekt, und tragen Sie im **Properties**-Fenster als **Text** ca. 30 Bindestriche, die (etwas provisorisch) den Bruchstrich darstellen sollen:

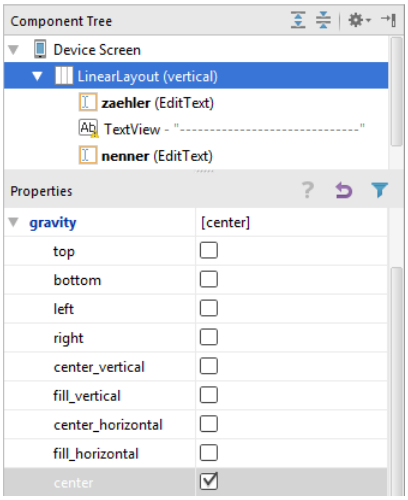


Mit dem kleinen Ausrufezeichen im gelben Warndreieck will uns die Entwicklungsumgebung sagen, dass Zeichenfolgen in Android-Apps nicht direkt kodiert, sondern durch String-Ressourcen (siehe Abschnitt 6.4) realisiert werden sollten, damit die App leicht lokalisiert (an andere Sprachen angepasst) werden kann. Wenn Sie mit Maus auf das Symbol zeigen, erfahren Sie seine Bedeutung:

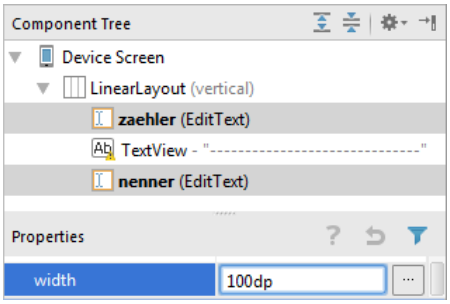


Im konkreten Fall ist eine Übersetzung der Bindestriche aber nicht erforderlich.

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



Reduzieren Sie die Breite der Texteingabefelder für Zähler und Nenner über die Eigenschaft **Width** auf 100 dp:

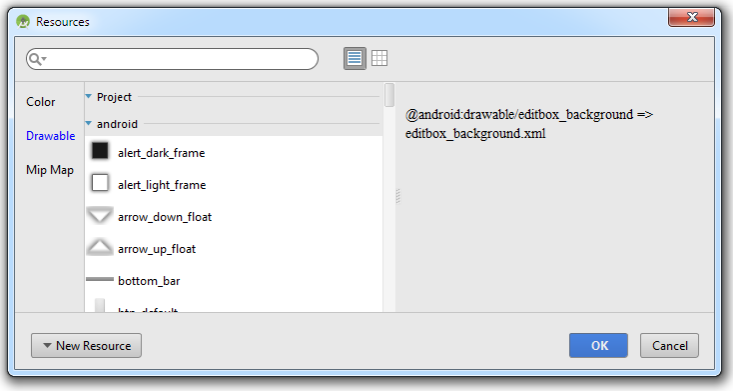


Bei der Maßeinheit **dp** (alias **dip**) handelt es sich um geräteunabhängige Pixel (*device independent pixel*), mit denen wir uns noch etwas näher beschäftigen werden. Ein auflösungsunabhängiges Pixel entspricht einem realen Pixel auf einem Gerät mit einer Auflösung von 160 dpi, hat also eine Größe von ca. 0,15875 mm ($\frac{25{,}4\text{mm}}{160}$). Ein Eingabefeld mit 100 dp ist also ca. 1,59 cm breit.

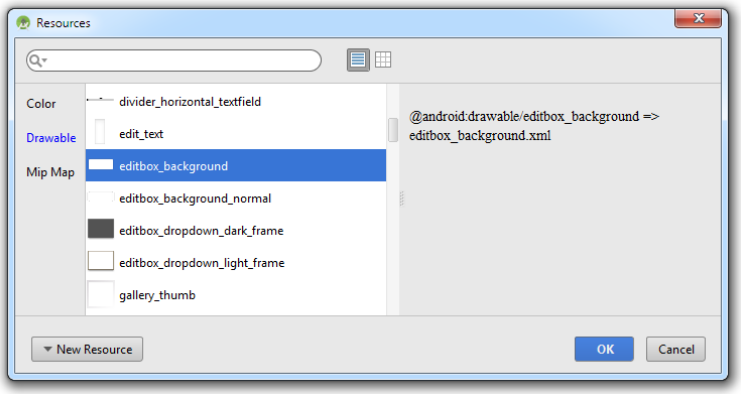
Setzen Sie die Eigenschaft **gravity** für beide Textfelder auf den Wert **center_horizontal**, damit die vom Benutzer eingetragenen Werte horizontal zentriert werden.

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 **Properties**-Fenster auf die Zelle zur Eigenschaft **background** und dann auf den dort erscheinenden Erweiterungsschalter [...], so dass im folgenden Fenster **Resources** nach Wahl der Kategorie **Drawable** aus der Liste **android**



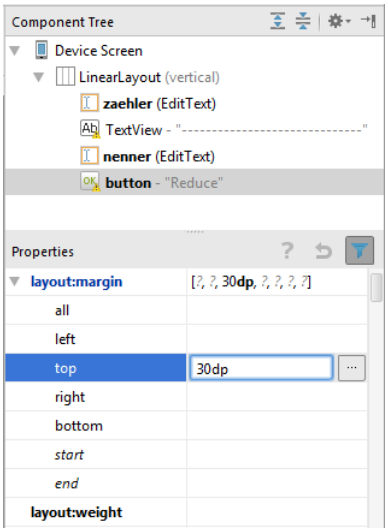
die **editbox_background** gewählt werden kann:



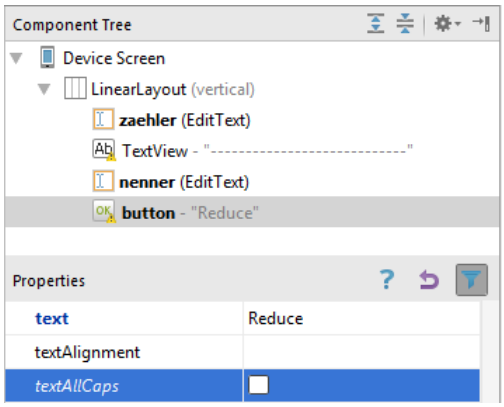
Komplettiert wird unsere Bedienoberfläche durch einen Schalter, mit dem das Kürzen des Bruchs ausgelöst werden soll. Befördern Sie aus dem Paletten-Segment **Widgets** ein Steuerelement vom Typ **Button** an den unteren Rand des Layouts. Ersetzen Sie per **Properties**-Fenster den Vorgabetext *New Button*, 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.

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



Um zu verhindern, dass im laufenden Programm die Schalterbeschriftung komplett in Großschreibung erscheint, setzen wir die Schaltereigenschaft **textAllCaps** auf den Wert **false**:



Das Attribut muss eventuell über den Symbolschalter (**Show expert properties**) zugänglich gemacht werden.

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

3.3 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 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 3.2) in der Datei **activity_main.xml** deklariert haben, erzeugt und angezeigt wird:

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

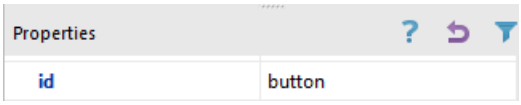
Nun müssen wir den Quellcode der Methode **onCreate()** noch etwas erweitern, damit unser Programm den Erwartungen der Anwender gerecht wird. Zunächst registrieren wir das beim Programmstart erzeugte Objekt der Klasse `MainActivity` als zuständig für die Behandlung von Klick-Ereignissen beim Befehlsschalter. Tragen Sie am Ende der Methode **onCreate()** eine Anweisung ein, die es ermöglicht, den Schalter anzusprechen:[1]

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

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

[1] 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.

- 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 den per Layout-Editor vereinbarten Namen:

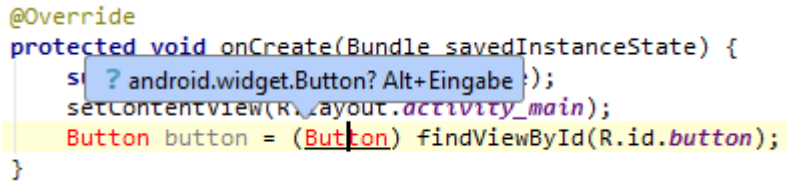


  Bei der Übersetzung hilft eine Klasse namens `R`, die Bestandteil der Anwendung ist und von der Entwicklungsumgebung automatisch generiert wird. Sie besitzt eine interne Klasse namens `id` mit statischen Feldern für die Steuerelemente. Das Feld zum Befehlsschalter ist über `R.id.button` ansprechbar und 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.
- Dem Compiler ist nur bekannt, dass die Methode **findViewById()** als Rückgabe eine Referenz zu einem Objekt der relativ breiten Steuerelementklasse **View** liefert. Damit diese Rückgabe zum Wert der **Button**-Variablen werden kann, müssen wir unser Wissen um die Zugehörigkeit des Steuerelements zur speziellen **View**-Unterklasse Klasse **Button** beisteuern und eine explizite Typumwandlung auf unsere Verantwortung vornehmen.

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

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

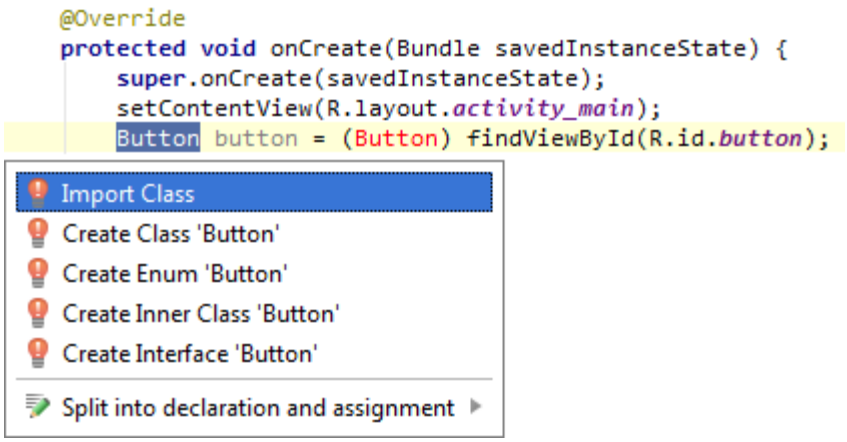
weil es den Bezeichner **Button** nicht auflösen kann. Wenn Sie per Mausklick die Einfügemarke auf den unbekannten Bezeichner setzen, fragt das Studio nach, ob die Klasse **Button** im Paket **android.widget** gemeint ist:



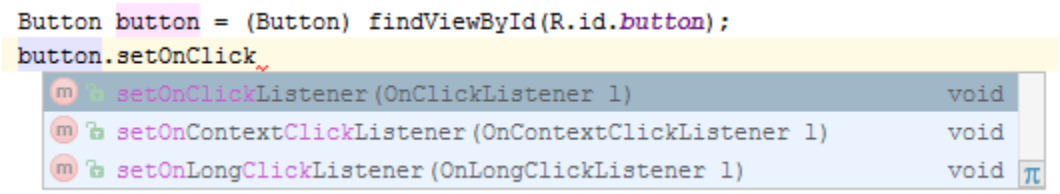
Wenn Sie dies mit der Tastenkombination **Alt+Eingabe** bestätigen, erzeugt das Studio eine Import-Anweisung am Anfang der Klassendefinition, und das Problem ist behoben:

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

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



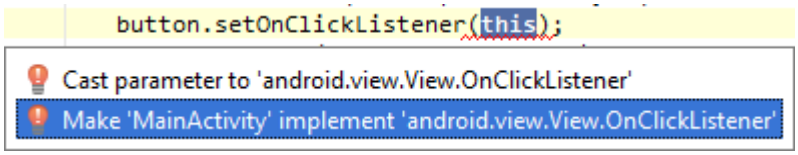
Jetzt verwenden wir die gewonnene Referenz und bitten den Schalter durch Aufruf seiner Methode **setOnClickListener()**, die handelnde Instanz unserer Aktivität (bezeichnet durch das Schlüsselwort **this**) als Klick-Interessenten einzutragen. Weil das Android Studio intelligente Vorschläge zur Code-Vervollständigung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



Es ist unbedingt zu beachten, dass die Registrierungsnachricht an den Befehlsschalter *hinter* den Aufruf **setContentView()** gesetzt wird, weil erst dieser Aufruf den Befehlsschalter erzeugt:

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

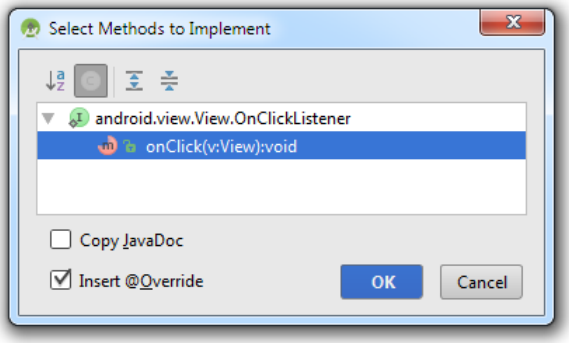
Wir bleiben trotz korrekter Erweiterung der Methode **onCreate()** nicht ohne Kritik, weil die Registrierung zur Benachrichtigung über Klickereignisse nur erlaubt ist für Objekte einer Klasse, die zusichert, alle im Interface **OnClickListener** geforderten Methoden zu beherrschen:



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 Benach-

richtigungen 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 beiden Texteingabefelder für den Zähler und den Nenner bequem ansprechen zu können. Dabei gehen wir genauso vor wie oben beim Befehlsschalter:

```
EditText etZaehler = (EditText) findViewById(R.id.zaehler);
EditText etNenner = (EditText) 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(etZaehler.getText().toString());
int n = Integer.parseInt(etNenner.getText().toString());
```

  In der ersten Anweisung wird das **EditText**-Objekt `etZaehler` 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 den schließlich optimal gekürzten Bruch:

```
if (z != 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;
} else
    n = 1;
```

  Eine Erläuterung zu diesem Kürzungsverfahren finden Sie z.B. in Baltes-Götz & Götz (2016, S. 178).
- Abschließend werden die gekürzten Zahlen von der **Integer**-Methode **toString()** wieder in Zeichenfolgen verwandelt und per **setText()** - Aufruf an die **EditText**-Steuerelemente übergeben:

```
etZaehler.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 = (EditText) findViewById(R.id.zaehler);
    EditText etNenner = (EditText) findViewById(R.id.nenner);
    int z = Integer.parseInt(etZaehler.getText().toString());
    int n = Integer.parseInt(etNenner.getText().toString());

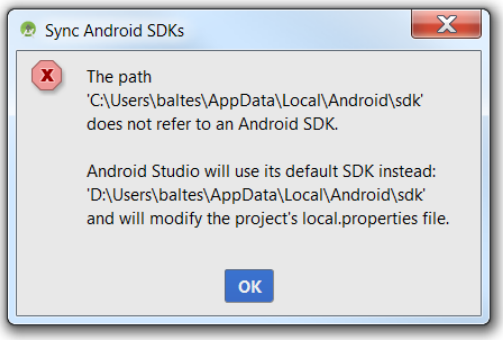
    // größten gemeinsamen Teiler mit dem Euklidischen Algorithmus bestimmen
    if (z != 0) {
        int rest;
        int ggt = Math.abs(z);
        int divisor = Math.abs(n);
        do {
            rest = ggt % divisor;
            ggt = divisor;
            divisor = rest;
        } while (rest > 0);
        z /= ggt;
        n  /= ggt;
    } else
        n = 1;

    etZaehler.setText(Integer.toString(z));
    etNenner.setText(Integer.toString(n));
}
```

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

...\BspUeb\Reduce\Kap3

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



Weil das Android Studio automatisch eine Anpassung der Projektkonfiguration vornimmt, ist das Problem nach einem Klick auf den **OK**-Schalter gelöst.

3.4 Sichern und Wiederherstellen

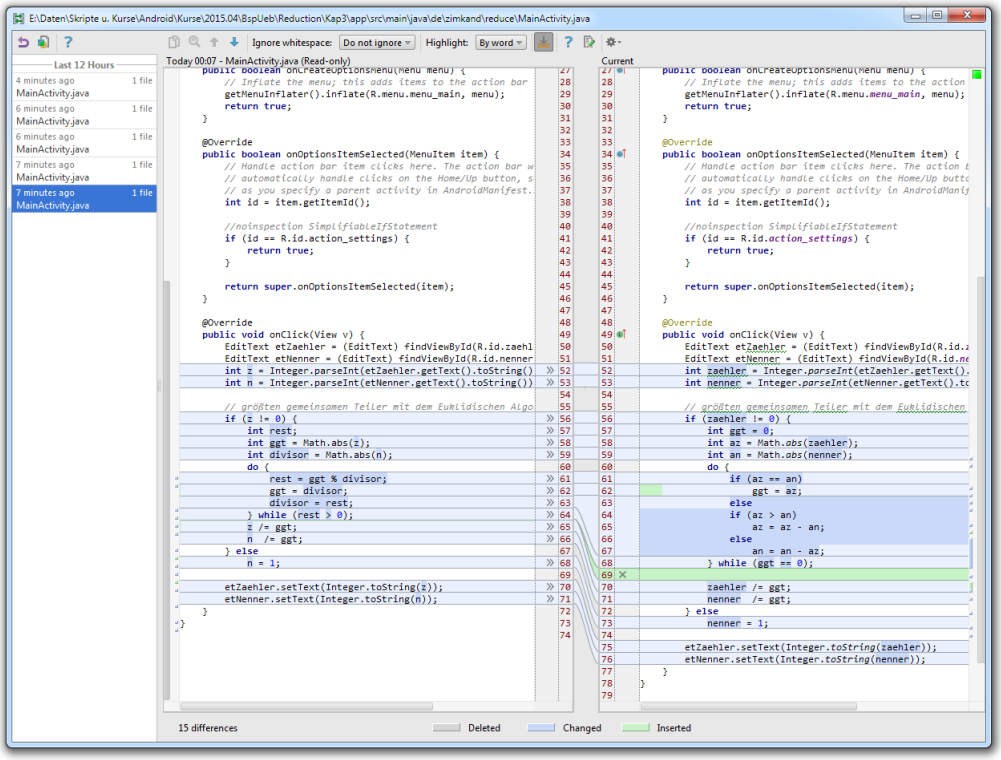
Bei der Arbeit mit dem Android Studio muss man sich um das Sichern von Quellcode und Ressourcen kaum Gedanken machen, weil die Entwicklungsumgebung bei jeder passenden Gelegenheit (z.B. beim Übersetzen) automatisch sichert.

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

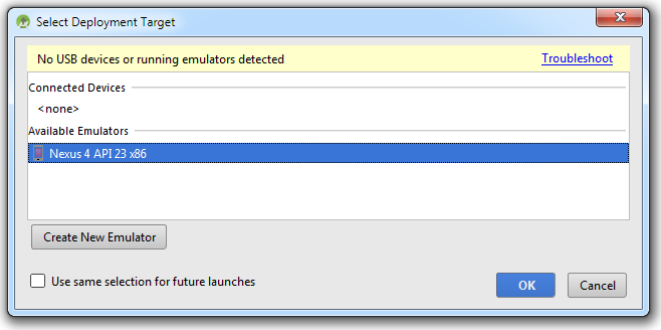
alle Zwischenstände verfügbar, z.B.:



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

3.5 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.:

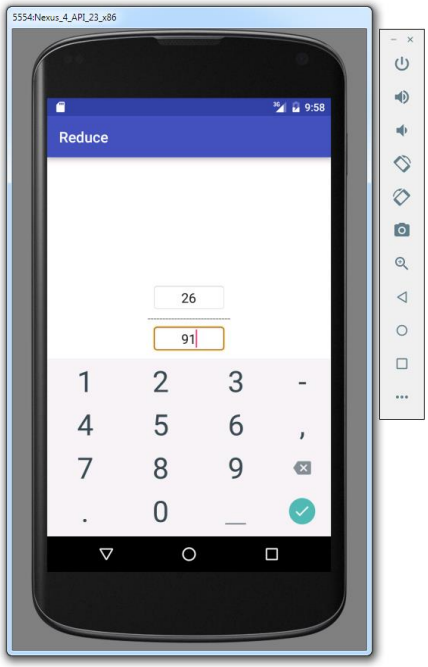


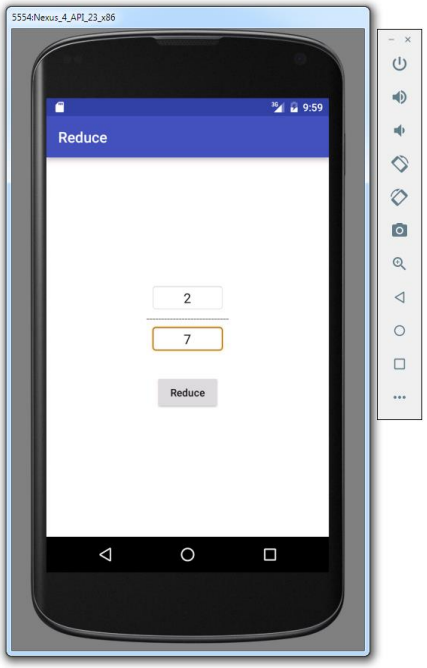
Bei einem virtuellen Gerät erspart man sich durch eine bescheidene Variante mit nicht allzu aktueller Android-Version und relativ kleiner RAM-Ausstattung ...

- 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:

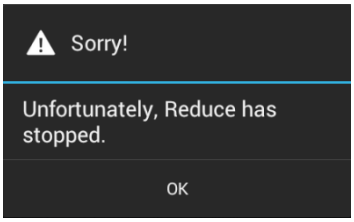




Selbstverständlich lässt sich die App auch auf ein im Debug-Modus am Entwicklungs-PC angeschlossenes Android-Gerät übertragen (vgl. Abschnitt 2.10):



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



Um dies 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());
```

ersetzen durch eine Alternative mit Überprüfung der Voraussetzungen:

```
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"
     . . .
 />
```

3.6 Nützliche Funktionen im Android Studio

In diesem Abschnitt werden einige besonders hilfreiche Funktionen im Android Studio beschrieben:

Hilfe zu einem Begriff bzw. Syntaxbestandteil abrufen
Man setzt die Einfügemarke auf den Begriff und drückt die Tastenkombination **Umschalt+F1**.

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

Refactor > Rename

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:



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

3.7 Eigene Apps auf einem beliebigen Android-Gerät installieren

Bei der bisher beschriebenen Vorgehensweise zum Erstellen einer App resultiert schlussendlich eine APK-Datei im folgenden Projektunterordner

...\app\build\outputs\apk

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.

In Android müssen Apps generell mit einem digitalen Zertifikat signiert sein, damit sie installiert werden können. Im Entwicklungsmodus, den wir bisher benutzt haben, und den wir auch im weite-

ren 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.

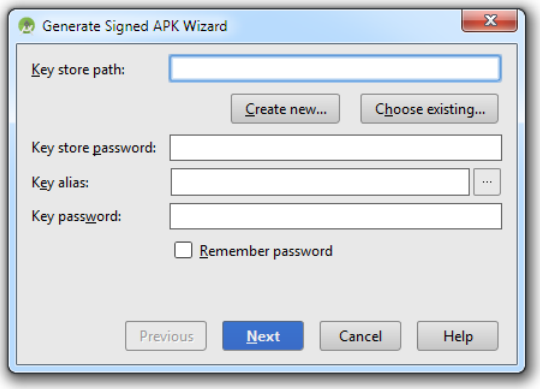
Um die Einschränkungen des Debug-Zertifikats zu vermeiden, muss eine App im **Release-Modus** erstellt und dabei mit einem Zertifikat signiert werden, dessen privater Schlüssel im Besitz des Entwicklers ist. Das zum Signieren im Release-Modus verwendete Zertifikat muss *nicht* von einer Zertifikats-Autorität (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 4.1) beantragen, in einer gemeinsamen Sandbox ausgeführt zu werden und sich damit z.B. wechselseitigen Dateizugriff zu erlauben (siehe Abschnitt 4.5). Solange Sie Ihren privaten Schlüssel nicht preisgeben, kann sich keine fremde App illegitime Zugriffsrechte verschaffen.

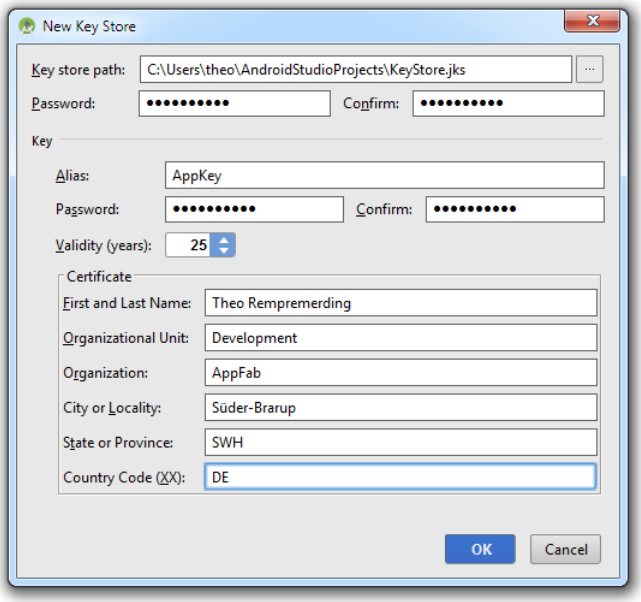
Im Android Studio startet man die Erstellung im Release-Modus mit

Build > Generate Signed APK

Ist noch kein privater Schlüssel vorhanden,



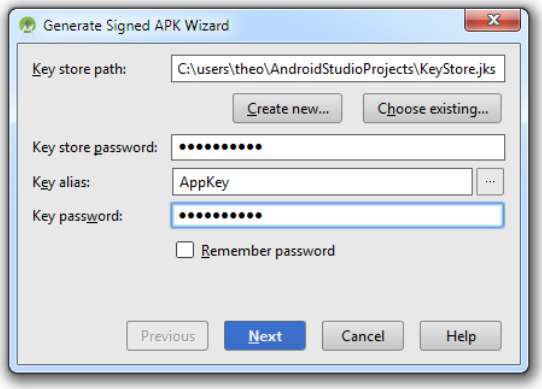
leitet man mit **Create new** die Erstellung eines Schlüsselspeichers und eines Release-Schlüssels ein:



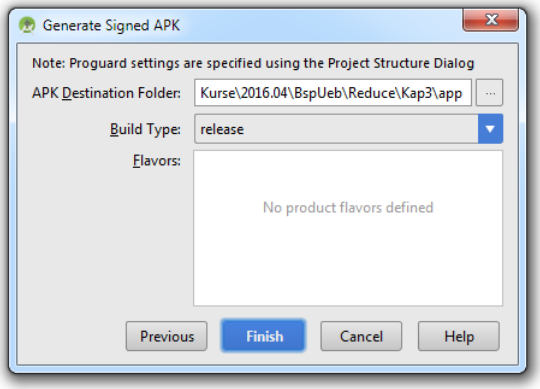
Google empfiehlt eine Laufzeit von mindestens 25 Jahren, um während der gesamten Lebenszeit einer App die Updates mit demselben Schlüssel signieren zu können. Das ermöglicht es den Benutzern, eine installierte App durch Updates bequem zu aktualisieren, statt jeweils eine neue App installieren zu müssen.

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:

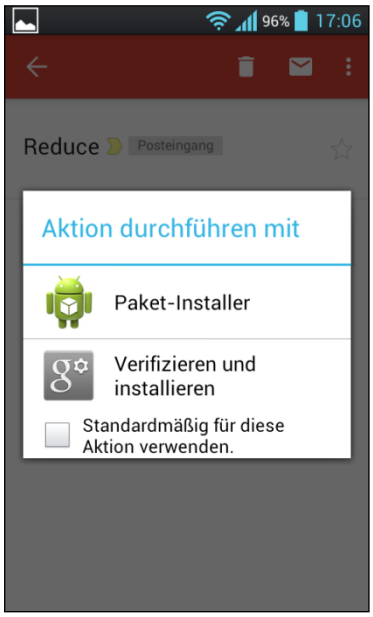


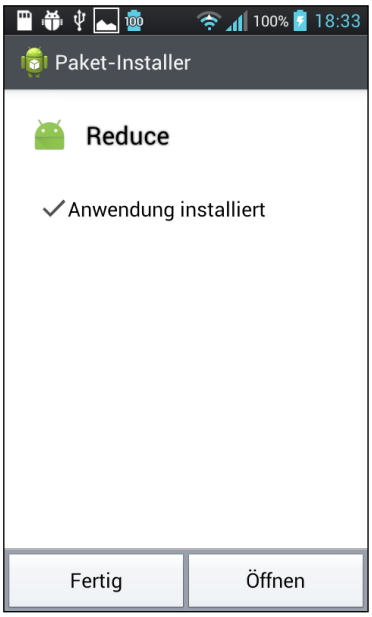
Eine im Release-Modus erstellte App landet per Voreinstellung in der Datei

...\app\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 Quellen**)
- Mail auf dem Zielgerät abrufen und öffnen, Klick auf den Anhang
- **Verifizieren und installieren**:





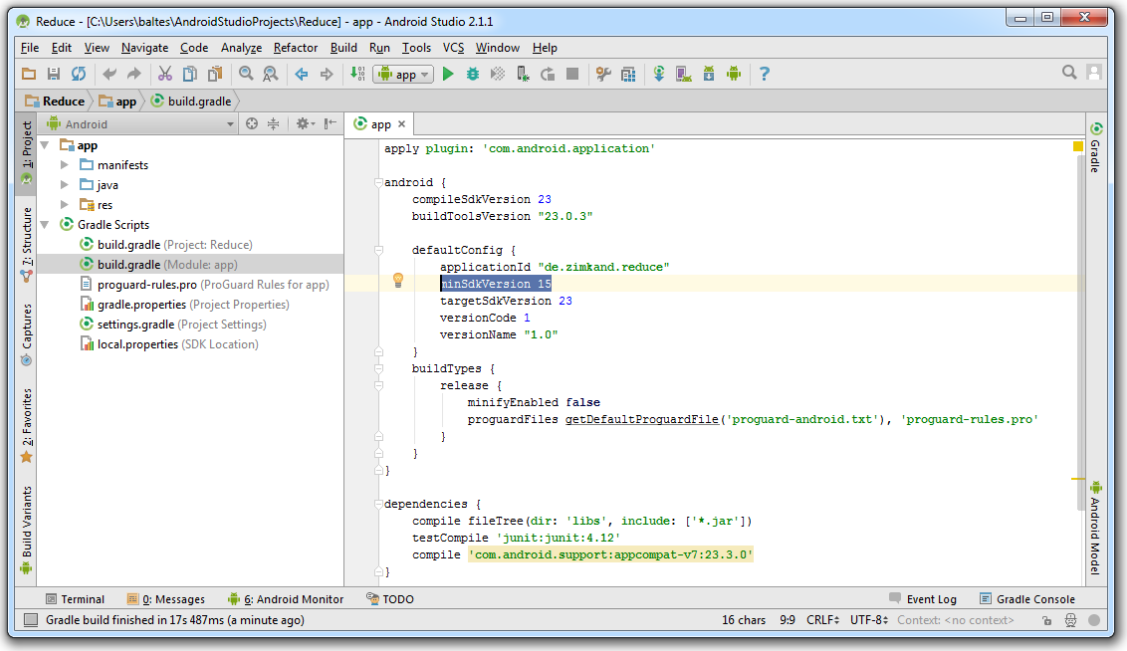
- Auf dem Zielgerät das Installieren von Software aus unbekannter Quelle wieder verbieten

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

http://developer.android.com/tools/publishing/app-signing.html

3.8 Projektkonfiguration einsehen und ändern

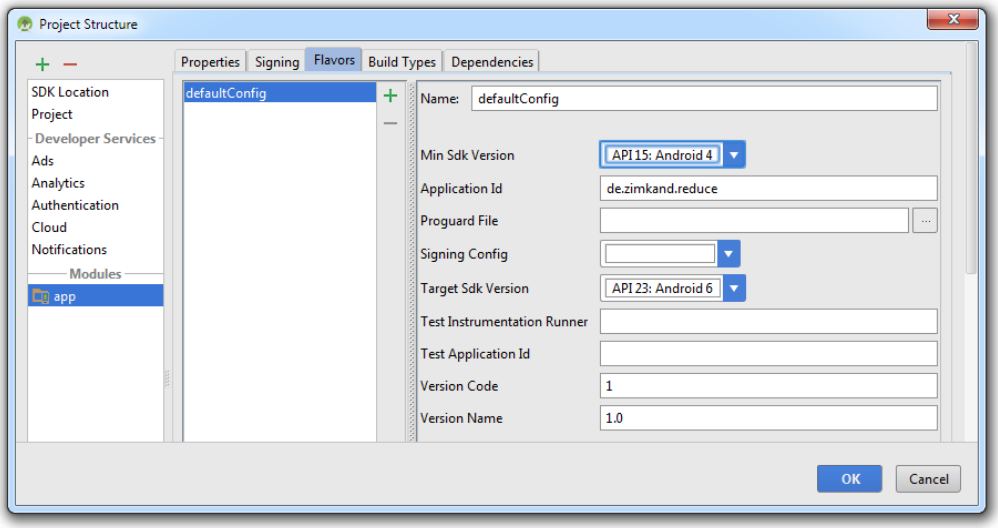
Wichtige Eigenschaften einer App befinden sich in der Modul-Variante der Gradle-Konfigurationsdatei **build.gradle**, z.B. im Attribut **minSdkVersion** die minimal vorausgesetzte Android-Version, die im Assistentendialog zur Anwendungserstellung (vgl. Abschnitt 3.1) angegeben wurde:



Bequemer als eine direkte Änderung dieser Datei ist die Verwendung des via

File > Project Structure > app > Flavors

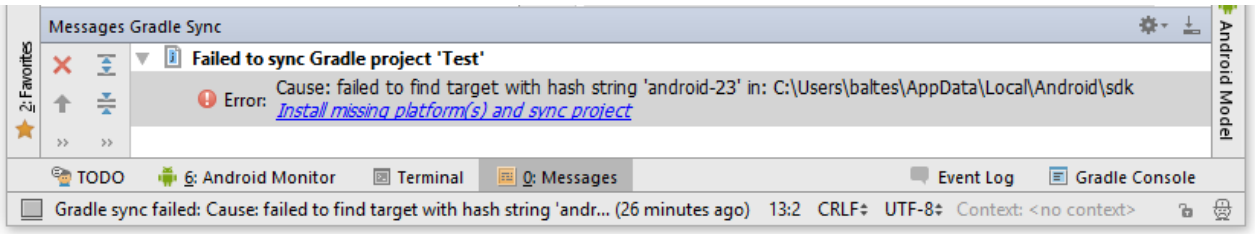
erreichbaren AS-Dialogs, z.B.:



3.9 Kursprojekte öffnen

Obwohl die Android Studio - Entwickler offenbar die Übertragung von Projekten auf andere Rechner nicht vorgesehen haben, sollte es Ihnen (zumindest auf einem Windows-Rechner) gelingen, die Kursprojekte über den simplen Menübefehl **File > Open** zu öffnen. Das wurde zuletzt im September 2016 auf einem PC unter Windows 10 (46 Bit) mit dem Android Studio in der Version 2.2 erfolgreich getestet.

Als wesentliche Voraussetzung muss die SDK-Installation Ihres Rechners die Android-Version 6.0 (Marshmallow, API-Level 23) enthalten. Ansonsten scheitert das Öffnen von Kursprojekten mit der folgenden Fehlermeldung (hier produziert von Android 2.2 auf einem System mit installierter SDK-Plattform 24, ohne die SDK-Plattform 23):



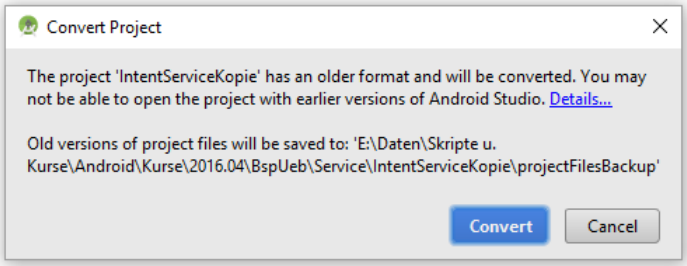
Im Vergleich zur Anpassung von Projektkonfigurationsdateien ist die Installation der SDK-Plattform 6.0 das kleinere Übel (siehe Abschnitt 2.6).

Wie die Kursprojekte zu beziehen sind, wird im Vorwort erläutert. Um das Transportvolumen zu verringern, wurden bei allen Kursprojekten die Inhalte folgender Verzeichnisse gelöscht:

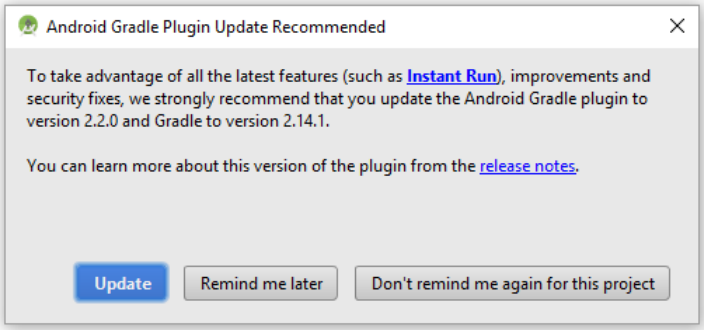
- **...\build**
- **...\app\build**

Das Android Studio sollte beim Öffnen eines Projekts die fehlenden Objekte automatisch neu erstellen.

Einige Belästigungen/Fehlermeldungen Probleme beim Öffnen der Kursprojekte sind leicht beheben. Das Konvertieren von Projekten können Sie bedenkenlos erlauben:

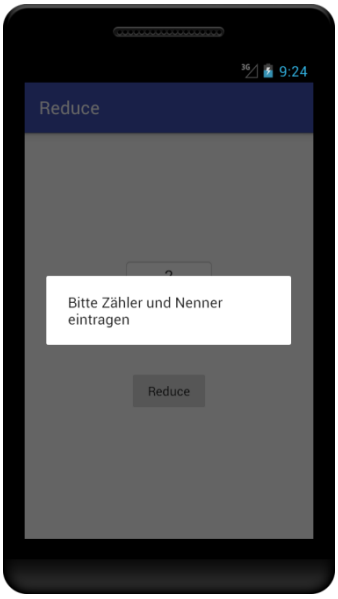


Es erscheint regelmäßig die Empfehlung, das Gradle Plugin zu aktualisieren, der man folgen sollte:



3.10 Übungsaufgaben zu Kapitel 3

1) Die in Kapitel 3 erstellte App stürzt nach der in Abschnitt 3.5 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();
```

4 Komponenten und Systemumgebung einer App

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

4.1 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.[1] 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, und die Startaktivität besitzt Bedienelemente oder Menüs, die das Verzweigen zu speziellen Aktivitäten erlauben. Unsere bisherigen Beispiel-Apps haben sich allerdings auf eine einzige Aktivität beschränkt. Während die Activities vom Anwender als die sichtbaren Bestandteile einer App erlebt werden, arbeiten die anschließend beschriebenen Komponenten im Hintergrund.
- **Service**
  Ein Service (dt.: *Dienst*) hat keine Bedienoberfläche und wird eingesetzt. wenn eine Funktion unabhängig von der im Vordergrund befindlichen Aktivität bzw. App permanent verfügbar sein soll (z.B. MP3-Player, GPS-Tracker). Ein Dienst wird von anderen Anwendungskomponenten gestartet und läuft auch dann noch im Hintergrund weiter, wenn der Benutzer zu einer anderen Anwendung wechselt.
- **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). Er hat keine eigene Bedienoberfläche und darf nur kurzzeitig aktiv werden, kann aber Aktivitäten oder Dienste starten.
- **Content Provider**
  Während unter Windows jede Anwendung mit den Zugriffrechten des angemeldeten Benutzers arbeitet, schottet Android die Anwendungen untereinander ab, so dass eine App nicht auf die von einer anderen App verwalteten Daten zugreifen kann. Will 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, benötigt sie einen Content Provider. Android selbst bietet etliche Content Provider (z.B. für die Kontakte oder Einstellungen). Wir behandeln Content Provider ausführlich in Kapitel 13.

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

---

[1] Die im Herbst 2016 erwartete Android-Version 7 (alias *N*) wird erstmals zwei Aktivitäten nebeneinander auf den Schirm bringen.

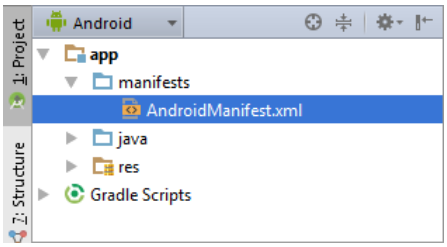
AppCompatActivity (aus dem Paket **android.support.v7.app**) definiert, die wiederum (indirekt) von der Klasse **Activity** (im Paket **android.app**) abstammt:

```
public class MainActivity extends AppCompatActivity implements View.OnClickListener {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button button = (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 gegenwärtig vor allem die **Material Design** - Unterstützung relevant. Das mit der Android-Version 5.0 (API-Level 12) eingeführte Material Design hat die Optik und die Funktionalität der Android-Bedienoberfläche modernisiert.[1]

4.2 Manifest

Damit für ein Programm aus den definierten Komponentenklassen Objekte erzeugt werden können, müssen die Komponenten 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.:[2]



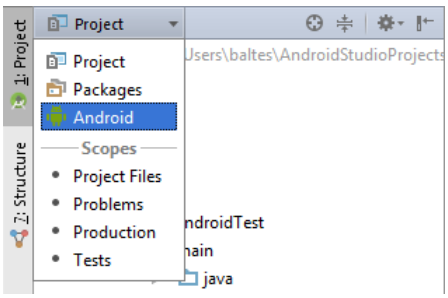
Bislang mussten wir uns nicht explizit mit der App-Manifestdatei beschäftigen, weil die erforderlichen Eintragungen von Assistenten erledigt wurden. Die App-Manifestdatei ist XML-formatiert und enthält ein **application**-Element mit Kindelementen für alle Komponenten des Programms. Anschließend ist das Manifest zu der in Kapitel 3 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"
```

---

1 Siehe https://developer.android.com/training/material/index.html

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



```
    package="de.zimkand.reduce">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        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
```

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

Im weiteren Verlauf von Kapitel 4 konzentrieren wir uns auf die Activities. Später werden auch die restlichen Anwendungskomponenten behandelt.

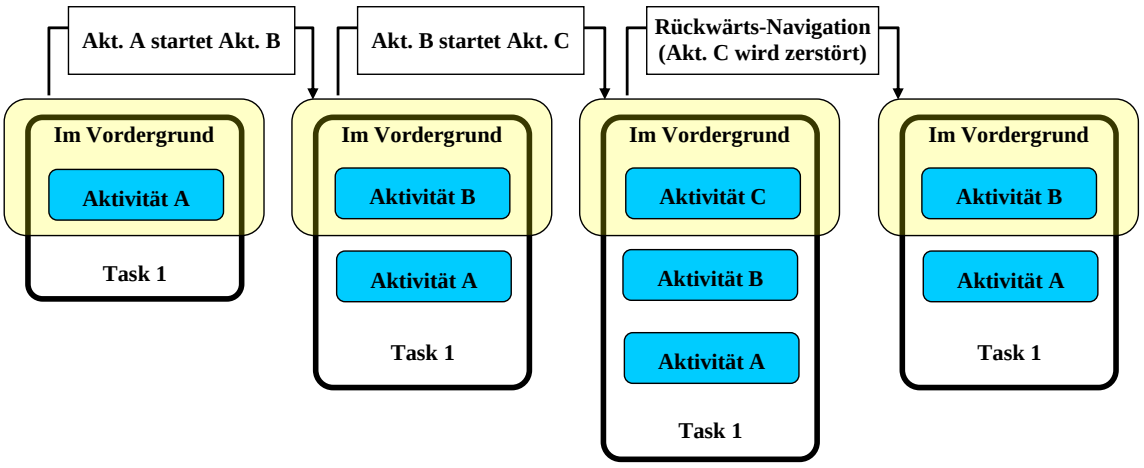
4.3 Task und Back Stack

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.

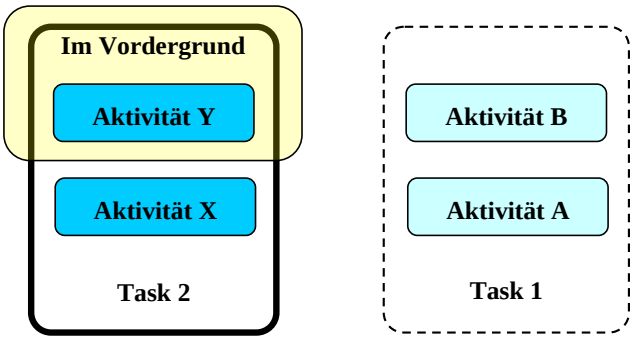
Wenn der Benutzer die Hauptaktivität einer Anwendung über den Startbildschirm oder den App-Starter (engl.: *launcher*) aktiviert und noch keine auf dieser Aktivität basierende Task existiert, legt Android eine neue Task an. Man kann also von *der* Task zu einer Anwendung sprechen.

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. 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:[1]

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



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:



Alle Aktivitäten von Task 1 sind im Zustand *gestoppt* und in Gefahr, bei Speichermangel beendet zu werden (siehe Kapitel 5). Der Back Stack zur Task 1 wird auch dann noch von Android verwaltet, wenn die Aktivitäten des Stapels bereits wegen Speichermangel zerstört worden sind.[1] 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. ▣ beim Nexus 5) oder 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 5 über den Lebenszyklus einer Aktivität behandelt.

Entwickler können (wohlbegründete und gut getestete) Ausnahmen von der normalen Task-Navigation anordnen und z.B. mit dem Wert **true** für das Attribut **alwaysRetainTaskState** zur Hauptaktivität einer App verhindern, das ihr Back Stack (vor dem nächsten System-Neustart) gelöscht wird.[2]

---

[1] Ein längere Zeit (z.B. 30 Minuten) nicht mehr aufgesuchter Stapel soll laut Google-Dokumentation (http://developer.android.com/guide/topics/manifest/activity-element.html) bei der Reaktivierung bis auf die Wurzelaktivität gelöscht werden. Ab der Android-Version 4 ist dieses Verhalten aber *nicht* mehr zu beobachten (siehe http://stackoverflow.com/questions/18847342/does-automatic-task-clearing-occur-in-modern-versions-of-android).

[2] Siehe http://developer.android.com/guide/components/tasks-and-back-stack.html

4.4 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.[1] 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 *sehr selten erforderlich*, die Komponenten einer Anwendung auf mehrere Prozesse zu verteilen. Als Gründe für das Verteilen einer App auf mehrere Prozesse kommen in Frage:

- Stabilität
  Die Stabilität steigt, weil die Konsequenzen eines Programmfehlers auf einen Prozess beschränkt bleiben.
- Verfügbarer Speicher
  Jeder Prozess erhält ein eigenes Heap-Speichersegment, so dass 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 selbst dann noch am Leben zu erhalten, wenn keine Anwendungskomponenten mehr vorhanden sind (siehe z.B. Kapitel 5 zum Lebenszyklus einer Aktivität). Bei Speichermangel ist Android allerdings gezwungen, Prozesse zu zerstören, wobei die Wichtigkeit der Prozesse für den Benutzer über die Reihenfolge der Elimination entscheidet.

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

- **Vordergrundprozesse**
  Gründe für die Zugehörigkeit eines Prozesses zu dieser Kategorie sind z.B.:
  - Hier läuft die Aktivität, die sich gerade im Vordergrund befindet.
  - Hier läuft ein Dienst, der an die Vordergrundaktivität gebunden ist.
  - Hier läuft gerade die **onReceive()** - Methode eines Broadcast Receivers.

  Im Normalfall existieren zu einem Zeitpunkt nur wenige Vordergrundprozesse, und es wird nur selten passieren, dass ein Vordergrundprozess zerstört werden muss.
- **Sichtbare Prozesse**
  Ein solcher Prozess enthält zwar keine Vordergrund-Komponenten, hat aber potentiell Einfluss auf sichtbare Bildschirminhalte. Gründe für die Zugehörigkeit eines Prozesses zu dieser Kategorie sind z.B.:
  - Hier läuft eine Aktivität, die sich zwar nicht mehr im Vordergrund befindet, aber noch sichtbar ist (ihre **onPause()** - Methode wurde ausgeführt).
  - Hier läuft ein Dienst, der an eine sichtbare Aktivität gebunden ist.
- **Dienstprozesse**
  Ein solcher Prozess enthält Dienste, fällt aber nicht in eine von den beiden oberen Kategorien. Es könnte z.B. ein Dienst vorhanden sein, der GPS-Daten zu einer Route aufzeichnet, Dateien in die Cloud sichert oder eine MP3-Datei abspielt. Android geht bei Diensten von einer relativ hohen Relevanz für den Benutzer aus.

---

[1] 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*).

- **Hintergrundprozesse**
  Ein solcher Prozess enthält 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 die betreuten Aktivitäten der letzte Benutzerkontakt zurückliegt.
- **Leere Prozesse**
  In einem leeren Prozess ist eine virtuelle Maschine vorhanden, die aber keine Anwendungskomponenten mehr betreut. Android lässt einen leeren Prozess nach Möglichkeit existieren, damit bei einer möglichen erneuten Nutzung der zugehörigen Anwendung die Zeit für die Prozesskreation eingespart werden kann.

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.[1]

4.5 Sicherheit

Dass bei Android-Apps trotzdem der verzögerten und vielfach 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 via Google Play bezogen, und dort kann Google Schadsoftware aussortieren. An dieser Voreinstellung sollten Anwender nichts ändern.
- Auf Wunsch des Anwenders kann Google die auf einem Android-Gerät bereits installierten Apps regelmäßig überprüfen und ggf. Schäden verhindern. Diese Funktion ist folgendermaßen zu aktivieren:
  - Android < 4.2
    Google-Einstellungen > Sicherheit >
    Gerät nach Sicherheitsbedrohungen durchsuchen
  - Android ab 4.2
    Einstellungen > Google > Sicherheit >
    Gerät nach Sicherheitsbedrohungen durchsuchen
- Android-Apps laufen in einem Sandkasten ab.

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

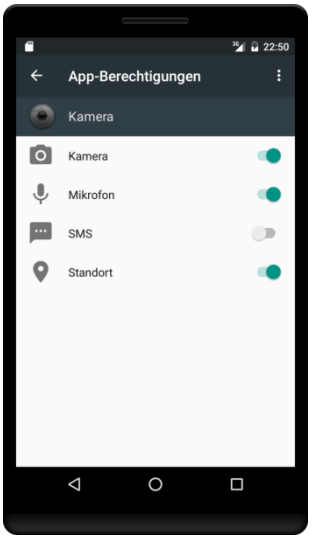
---

1 https://developer.android.com/guide/components/processes-and-threads.html
2 http://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 3.7) können über ihre Manifestdateien (mit dem Attribut **android:sharedUserId**) anfordern, in einem gemeinsamen Sandkasten ausgeführt zu werden, so dass z.B. wechselseitige Dateizugriffsrechte bestehen.[1]

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 bei der Installation genehmigt werden müssen und seit der Android-Version 6 (über **Einstellungen > Apps**) auch später noch administriert (z.B. entzogen) werden können, z.B.:



Wir werden uns in Kapitel 13 damit beschäftigen, wie eine App Berechtigungen zur Nutzung fremder Dienste anfordern und Berechtigungen zur Nutzung eigener Dienste durch andere Apps definieren kann.

4.6 Übungsaufgaben zu Kapitel 4

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.

[1] http://developer.android.com/guide/topics/manifest/manifest-element.html

5 Lebenszyklus einer Aktivität

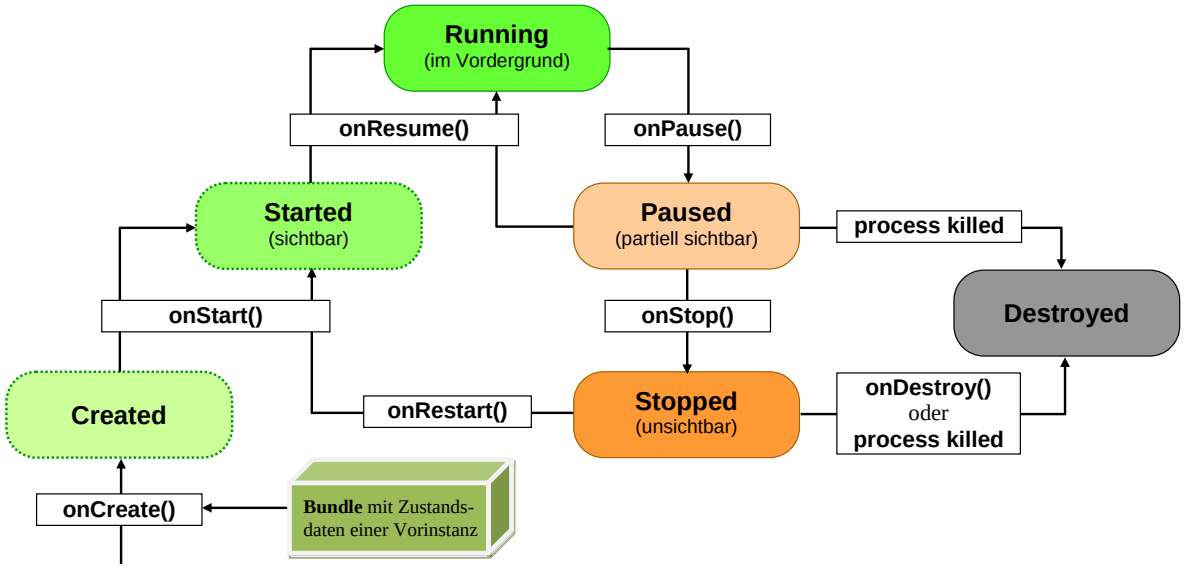
Android greift im Vergleich zu anderen Betriebssystemen stärker in den Lebenslauf einer Anwendung ein. Einzelne Anwendungskomponenten oder komplette Prozesse können stillgelegt und später bei Bedarf neu gestartet werden, wobei z.B. folgende Anlässe beteiligt sein können:

- 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 eine Browser-Activity nach dem Wechsel der Geräteorientierung dieselbe Webseite anzeigt und auf dieselbe Position fokussiert ist. In diesem Kapitel werden wir uns mit dem Lebenszyklus einer Activity befassen.

5.1 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. In der folgenden Abbildung sind die möglichen Zustände einer Aktivität zusammen mit den beim Zustandswechsel beteiligten Methoden bzw. Android-Maßnahmen 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**
  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 und kann in diesem Zustand verweilen, bis der Benutzer zurückkehrt. Bei extremem Ressourcenmangel kann sich das System aber dafür entscheiden, die pausierte Aktivität oder den gesamten Anwendungsprozess mit allen darin enthaltenen Komponenten zu zerstören.

- **Stopped**
  Die Aktivität ist *komplett* durch eine andere Aktivität überlagert. Sie befindet sich funktionstüchtig im Speicher und kann in diesem Zustand verweilen, bis der Benutzer zurückkehrt, oder das System die Aktivität wegen Ressourcenmangel zerstört. Wenn der Benutzer die oberste Aktivität durch Betätigung des Rückwärts-Schalters vom Back Stack abräumt, gelangt sie vom Zustand **Stopped** sofort in den Zustand **Destroyed**.
- **Destroyed**
  Entweder wartet die Aktivitätsinstanz nach dem Aufruf der Methode **onDestroy()** darauf, durch den Garbage Collector der virtuellen Maschine aus dem Speicher entfernt zu werden, oder der Prozess, in dem sie ausgeführt worden war, musste wegen Speichermangel aufgegeben werden. Wenn eine Rückkehr des Benutzers möglich ist, sichert Android die Steuerelementzustände der Aktivitätsinstanz (z.B. die Inhalte von Texteingabefeldern) auf dem Back Stack. Wird eine neue Aktivitätsinstanz erstellt, werden die gespeicherten Zustandsdaten beim Aufruf der Methode **onCreate()** als Parameter vom Typ **Bundle** übergeben. Damit ein Steuerelement (ein Objekt der Klasse **View**) unterstützt werden kann, muss eine Kennung bzw. Element-ID (XML-Attribut **android:id**) vorhanden sein (vgl. Abschnitt 7.2.1.1). Bei einer per Rückwärts-Schalter oder **finish()** - Methode 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.
- **Persistenzdaten**
  Damit sind permanent zu speichernde Daten gemeint (z.B. ein vom Benutzer angelegter Kontakt), die oft an Content Provider übergeben werden.

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 (Müllsammler, siehe unten) 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 gestal-

ten. 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.
- 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 **onStop()** bzw. **onDestroy()** für einzelne Aktivitäten auf, sondern terminiert nach einer Prioritätenliste komplette Prozesse (siehe Abschnitt 4.4).

Es ist zu beachten, dass eine Aktivität bereits nach der Ausführung von **onPause()** von der Eliminierung wegen Ressourcenmangel bedroht ist. Um eine eliminierte Aktivität später reaktivieren zu können, merkt sich Android 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.

5.2 Zustandswechselmethoden

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

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

5.2.1 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. Die Methode erhält als Parameter ein Objekt der Klasse **Bundle**, in dem sich eventuell ein gespeicherter Zustand befindet, der wiederhergestellt werden sollte, ansonsten eine **null**-Referenz.

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

5.2.2 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.

5.2.3 onRestoreInstanceState()

Die in der obigen Abbildung 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. 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 Restaurationen von Zustandsdaten in **onCreate()**.

5.2.4 onResume()

Nach dem Aufruf 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).

5.2.5 onSaveInstanceState()

Die in der obigen Abbildung nicht enthaltene Methode

protected void onSaveInstanceState(Bundle *outState***)**

wird unmittelbar vor oder nach **onPause()** aufgerufen, ...

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

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:

- 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. Durch das Überschreiben von **onBackPressed()** könnte das von Android nicht vorgesehene Sichern der Zustandsdaten einer per Zurück-Schalter verlassenen Aktivität nachgerüstet werden (z.B. unter Verwendung der im Kapitel 14 beschriebenen Shared Preferences).

5.2.6 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).

Man muss damit rechnen, dass der Benutzer nicht mehr zurückkehrt. Daher sind alle Persistenzdaten (z.B. gerade editierte Kontaktdaten) in **onPause()** zu sichern (z.B. durch Übergabe an einen Content Provider). 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.

Nach Ausführung der Methode **onPause()** ist eine Aktivität davon bedroht, bei Speichermangel eliminiert zu werden. Es ist also nicht garantiert, dass die beiden im Lebenszyklus potentiell noch folgenden Methoden **onStop()** und **onDestroy()** tatsächlich aufgerufen werden.

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:[1]

> 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 ein Abbrechen-Schalter angeboten werden, dann ist dieser zusätzlich zu implementieren.

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.

---

[1] http://developer.android.com/reference/android/app/Activity.html

5.2.7 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.

5.2.8 onDestroy()

In der Methode

protected void onDestroy()

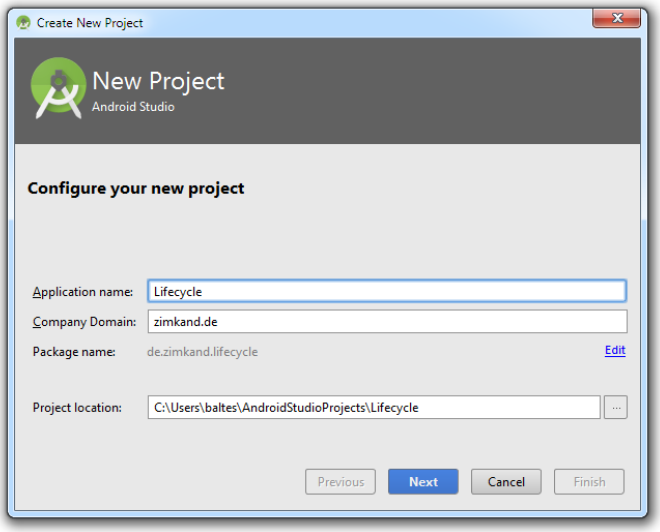
sollten aufwändige Ressourcen freigegeben werden. Google nennt als typische Aufgabe für diese Methode das Stoppen von Threads (siehe Kapitel 9), die mit der zu beendenden Aktivität assoziiert sind. Geht der **onDestroy()** - Aufruf auf einen vorherigen Aufruf der Methode **finish()** zurück, mit der sich eine Aktivität selbst beenden kann, 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 10.2.2).

5.3 App zur Lifecycle-Demonstration

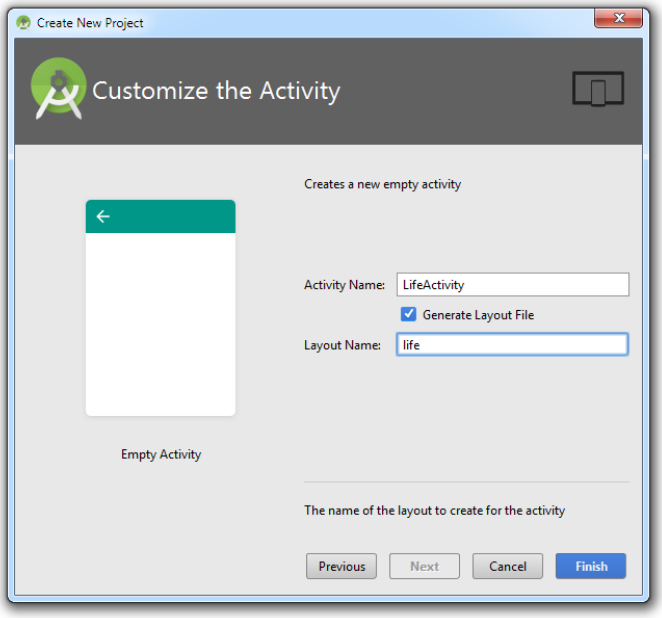
Um Stationen im Lebenszyklus einer Aktivität zu demonstrieren, erstellen wir über

File > New Project

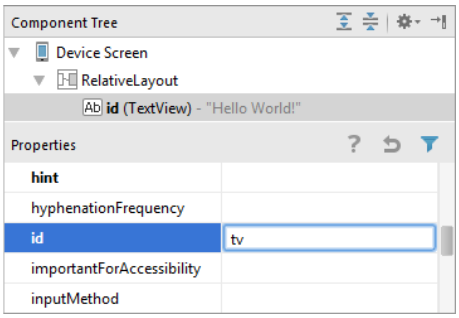
ein neues Projekt mit dem Namen `Lifecycle`:



Wir akzeptieren als **Target Android Devices** wie üblich die Voreinstellung **Phone and Tablet** 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):

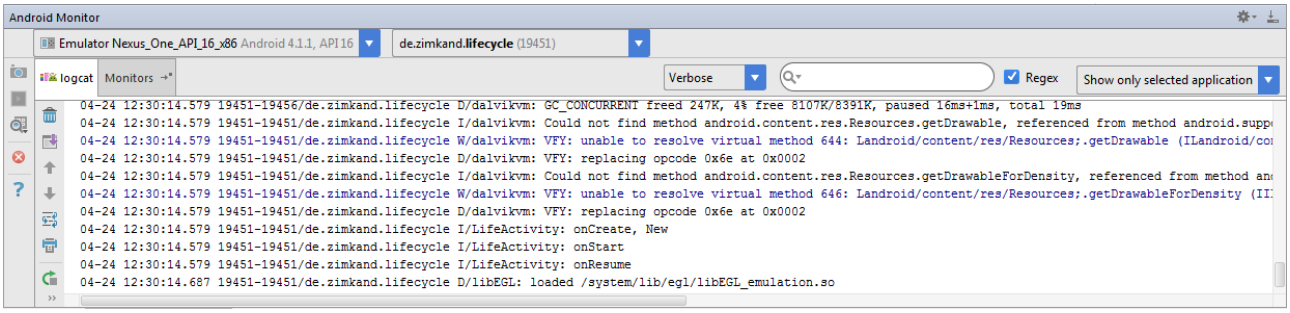


Wir lassen das Projekt mit einem Klick auf **Finish** erstellen und kümmern uns dann im **Design**-Modus des Layout-Editors (zur Bearbeitung der Datei **life.xml**) um die Bedienoberfläche. Hier beschränken wir uns darauf, eine Ressourcen-Kennung für das vorgegebene **TextView**-Objekt zu vergeben:



5.3.1 Logging

Das **logcat**-Registerblatt im **Android Monitor** - Fenster 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:[1]

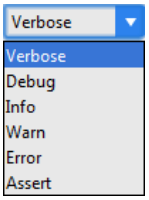


[1] Um das Protokoll auf einem Android-Gerät lesen und in eine Datei sichern zu können, wird eine App wie z.B. *Android System Info* benötigt (Post 2013, S. 285).

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

- Datum
  z.B. `04-24`
- Uhrzeit in GMT
  z.B. `12:30:14.579`
- Prozess-Identifikation
  z.B. `19451-19451/de.zimkand.lifecycle`
- Etikett der Meldung (TAG, siehe unten)
  z.B. `I/LifeActivity`
- Text der Meldung
  z.B. `onResume`

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:



In der **logcat**-Sicht erscheinen zahlreiche Systemmeldungen, und wir können im Programm zusätzliche Ausgaben veranlassen. Dazu genügt der altbekannte Methodenaufruf **println()** an das Objekt **System.out**. Allerdings sind zur Erstellung von **logcat**-Einträgen die statischen Methoden 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. Alle Methoden akzeptieren zwei **String**-Parameter:

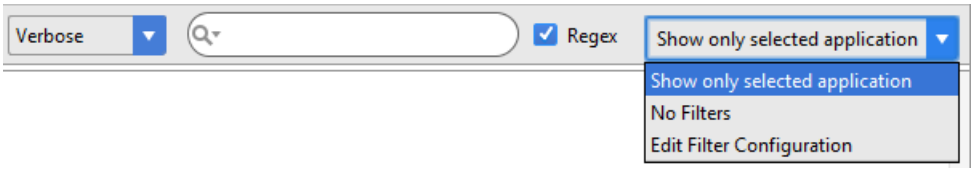
- **tag**
  Über dieses Etikett wird üblicherweise die Herkunft einer Meldung bekannt gegeben, wobei sich oft der Namen der Aktivitätsklasse anbietet. Um Aufwand und Tippfehler einzusparen, ist die Verwendung einer finalisierten **String**-Variablen zu empfehlen, z.B.:
  ```
  private final String TAG = 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.
  Per **System.out** erstellte Ausgaben erhalten das Etikett „**System.out**“.
- **text**
  Der Meldungstext

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

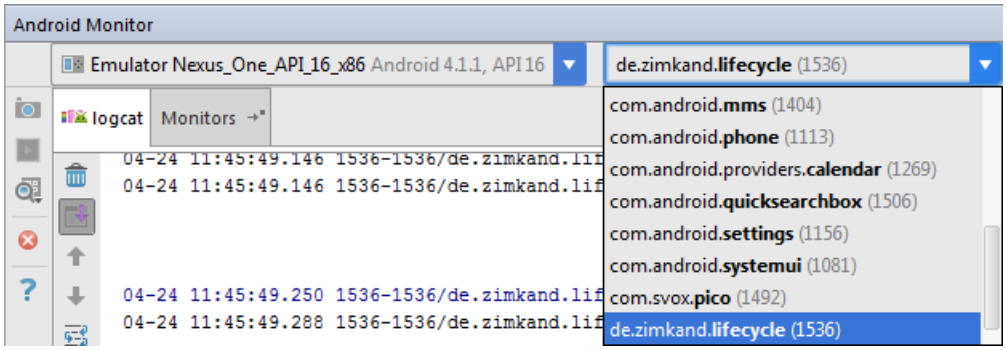
```
Log.i(TAG, "onCreate, " + (savedInstanceState == null ? "New" : "Restored"));
```

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.

In der **logcat**-Sicht 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

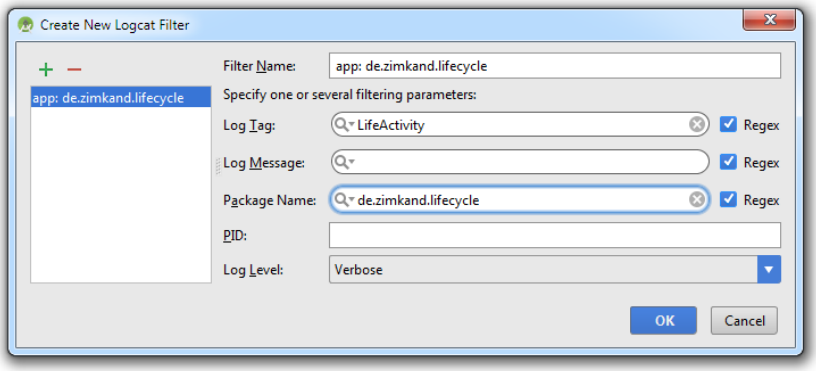


anzuzeigen. Nach dem Start einer App ist diese ausgewählt, was zu einer vorteilhaften Filterung der **logcat**-Ausgaben führt.

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

```
app: de.zimkand.lifecycle
```

mit folgenden Einstellungen:



Indem wir im Feld **Log Tag** den von unserer App beim Aufruf einer **Log**-Methode aufgrund des **tag**-Parameters ausgegebenen Activity-Klassenamen 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.

Über den Schalter lassen sich die Einträge in der **logcat**-Sicht löschen. Wenn die Sicht erwartungswidrig leer bleibt, hilft oft ein Klick auf den **Restart**-Schalter .

5.3.2 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.ActionBarActivity;
import android.os.Bundle;
import android.util.Log;
import android.view.Menu;
import android.view.MenuItem;
import android.widget.TextView;

public class LifeActivity extends ActionBarActivity {

    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 = (TextView) 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");
    }

    @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));
```

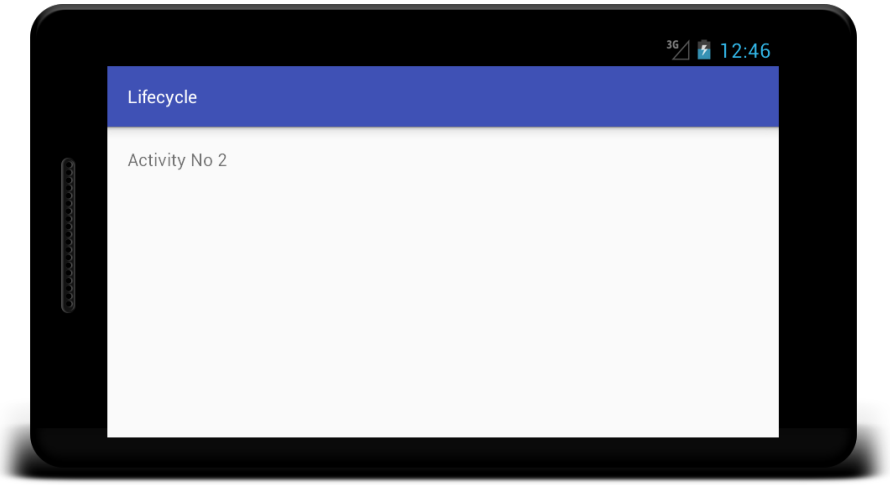
Das komplette Projekt `Lifecycle` ist im folgenden Ordner zu finden

...\BspUeb\Lifecycle

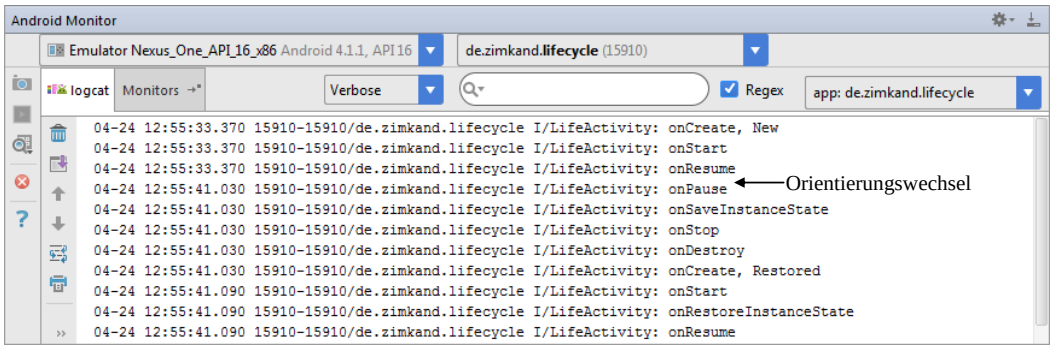
Wir beobachten nun das Activity-Management von Android in verschiedenen Szenarien.

5.3.2.1 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** (vgl. Abschnitt 5.3.1) erhalten wir die nach obigen Erläuterungen zu erwartende Sequenz von Einträgen:

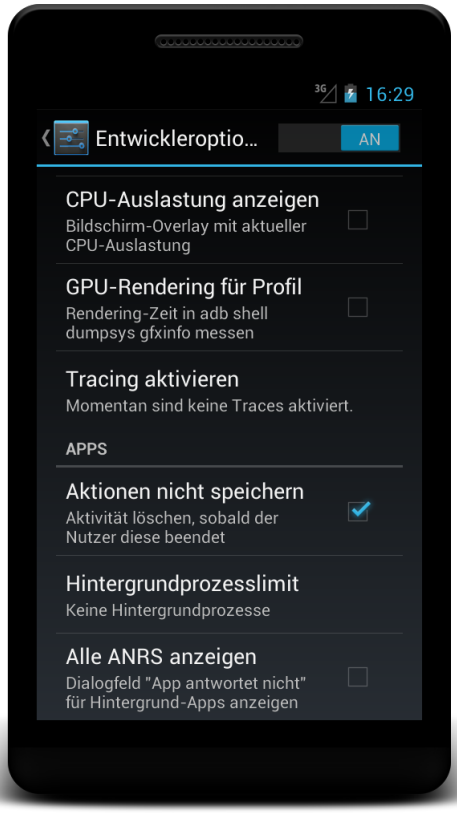


5.3.2.2 Neukreation nach Prozessterminierung

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:[1]

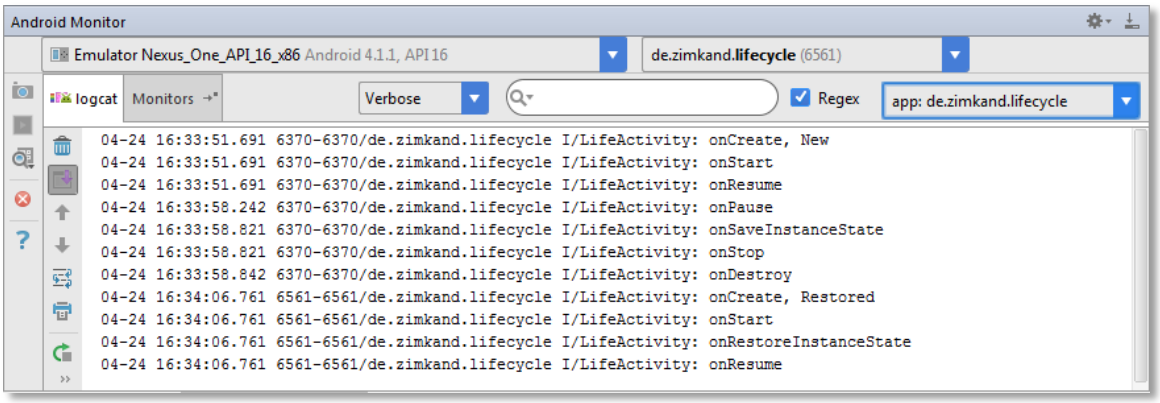
- **Aktionen nicht speichern**
  Ist diese Einstellung aktiviert, wird eine Aktivität zerstört, sobald sie vom Benutzer verlassen wird (z.B. in Richtung Startbildschirm).
- **Hintergrundprozesslimit**
  Wählt man die Option **Keine Hintergrundprozesse**, dann werden leere Prozesse sofort entfernt.

Hier sind die beiden Einstellungen bei einem Smartphone mit Android 4.1.1 (API-Level 16) zu sehen:



Wir starten unsere App, verlassen sie in Richtung Startbildschirm und kehren dann zurück (Liste mit den zuletzt verwendeten Aufgaben aufrufen, `Lifecycle` auswählen). Wie die **logcat**-Einträge

[1] Siehe Abschnitt 2.10.1 wegen der eventuell erforderlichen Aktivierung von Entwickleroptionen

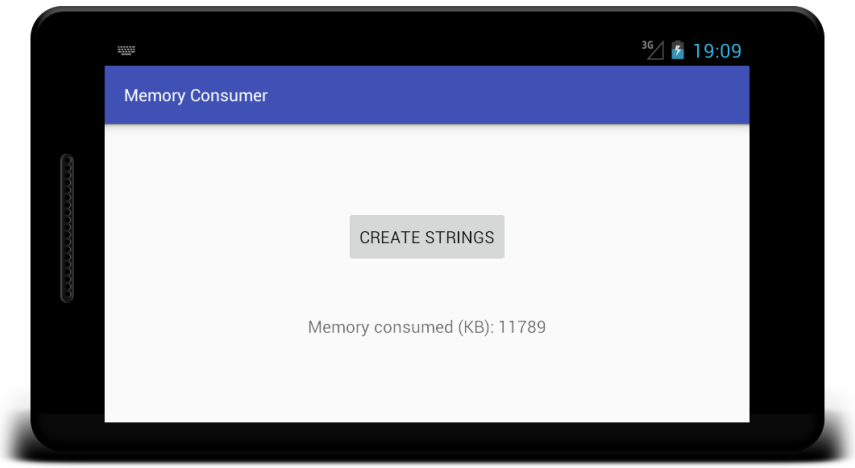


zeigen, ...

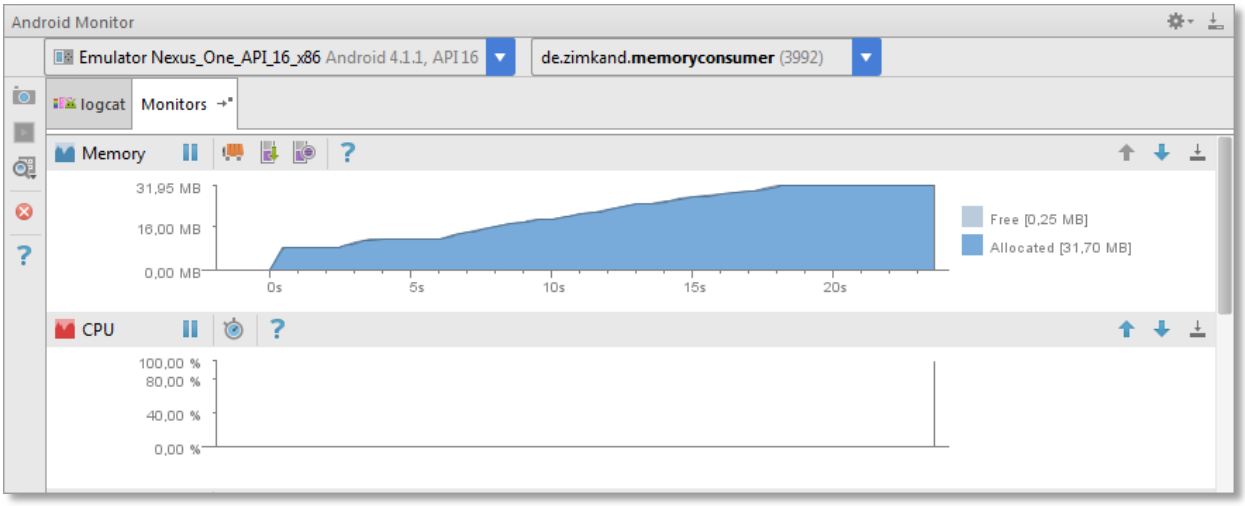
- hat die App durch einen Aufruf der Methode **onDestroy()** Gelegenheit zu einem geordneten Rückzug erhalten,
- ist die Aktivität in einem anderen Prozess restauriert worden.
  Dies ist in der Spalte mit der **PID** (*Process ID*) abzulesen.

5.4 App-Monitoring im Android Studio

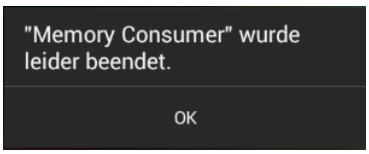
Nachdem wir mit der **logcat**-Sicht im **Android Monitor** ein sehr nützliches Diagnoseinstrument kennen gelernt haben, werfen wir bei dieser Gelegenheit einen Blick auf die benachbarte Registerkarte **Monitors**, die es ermöglicht, den Speicherbedarf, die CPU-Nutzung, die Netzwerkaktivität und die GPU-Nutzung einer App zu beobachten. Wir beschränken uns auf die Speicherverwendung und erstellen eine App namens `Memory Consumer` mit der folgenden Bedienoberfläche:



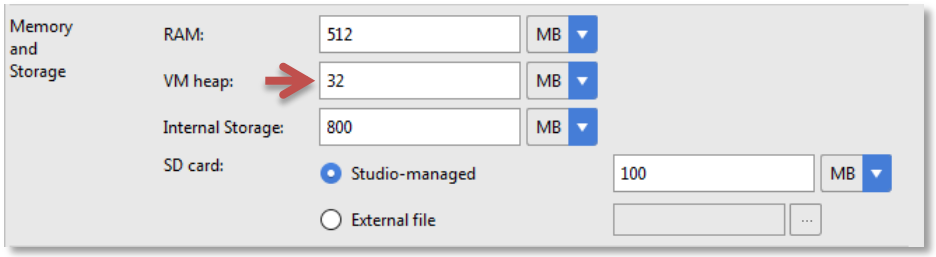
Bei jedem Klick auf den Schalter sollen 10.000 **String**-Objekte erzeugt werden. Das Textfeld zeigt an, wieviel Speicher die App aktuell belegt. Das Android Studio liefert ein Verlaufsdiagramm zur Speichernutzung, das seine Maximalhöhe bei ca. 32 MB erreicht:



Bei diesem Wert wird die Anwendung von Android gestoppt,



weil die verwendete virtuelle Maschine einen auf 32 MB limitierten Heap-Speicher (für Objekte) besitzt:



Eventuell sind Sie am Quellcode zur Speicherverschwendungsaktivität interessiert:

```
package de.zimkand.memoryconsumer;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity implements View.OnClickListener {
    private String [][] as = new String[100][];
    private int index = 0;
    Runtime runtime = Runtime.getRuntime();
    TextView tv;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Button button = (Button) findViewById(R.id.button);
        tv = (TextView) findViewById(R.id.tv);
        long consumed = (runtime.totalMemory() - runtime.freeMemory())/1024;
        tv.setText("Memory consumed (KB): " + Long.toString(consumed));

        button.setOnClickListener(this);
    }
```

```
        @Override
        public void onClick(View v) {
            String[] star = new String[10_000];
            for (int i = 0; i < 10_000; i++)
                star[i] = new String();
            as[index++] = star;
            long consumed = (runtime.totalMemory() - runtime.freeMemory())/1024;
            tv.setText("Memory consumed (KB): " + Long.toString(consumed));
        }
    }
```

Über die statische Methode **runtime()** der Klasse **Runtime** erhalten wir ein Objekt der Klasse **Runtime**, das es der Anwendung erlaubt, mit ihrer Laufzeitumgebung zu kommunizieren. Dieses Objekt liefert über die Methode **totalMemory()** die Größe des aktuell von der virtuellen Maschine verwalteten Speichers. Diese Größe kann auch in Abhängigkeit von Umgebungsbedingungen schwanken. Wir ermitteln zusätzlich über die Methode **freeMemory()** den freien Speicher und berechnen den belegten Speicher als Differenz.

Für den Befehlsschalter realisieren wie die Klickbehandlungsmethode diesmal mit Hilfe einer anonymen Klasse. In **onClick()** werden bei jedem Aufruf 10.000 **String**-Objekte erzeugt. Durch einen als Instanzvariable angelegten Array bestehen Referenzen auf die **String**-Objekte außerhalb der Ereignismethode, so dass die Objekte *nicht* dem Garbage Collector zum Opfer fallen.

Das komplette Projekt `Memory Consumer` ist im folgenden Ordner zu finden

...\BspUeb\Memory Consumer

5.5 Übungsaufgaben zu Kapitel 5

1) Wenn der Benutzer eine Aktivität mit dem Rückwärtsschalter verlässt, wird (indirekt) die Methode **finish()** aufgerufen. In der Zustandswechselmethode **onPaused()**, die bevorzugt zum Speichern von Persistenzdaten verwendet wird, 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 5.1).

2) Welche Lebenszyklus-Methoden werden bei einer im Vordergrund befindlichen Aktivität aufgerufen, wenn der Benutzer das Display aus- und anschließend wieder einschaltet?

6 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 so genannten *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.

6.1 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:[1]

Ressourcentyp	**Beschreibung**	**res-Unterordner** **Dateityp**
Eigenschafts-Animation	Bei dieser Animation wird eine Eigenschaft eines Objekts (z.B. die Hintergrundfarbe) zeitabhängig verändert.	**animator** **XML**
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.	**anim** **XML**
Color State List	Es wird eine Liste von Farben für verschiedene Zustände eines **View**-Objekts (z.B. normal, fokussiert, gedrückt) definiert. Diese Liste kann einem **View**-Objekt wie eine Hintergrundfarbe zugewiesen werden.	**color** **XML**

[1] Weitere Informationen zu den von Android unterstützen Ressourcen-Typen sind hier zu finden:
http://developer.android.com/guide/topics/resources/providing-resources.html
http://developer.android.com/guide/topics/resources/available-resources.html

Ressourcentyp	**Beschreibung**	**res-Unterordner** **Dateityp**
Zeichenbare Ressourcen	Hier geht es um Bitmap-Dateien (in den Formaten PNG, GIF und JPG) sowie andere zeichenbare Ressourcen, z.B.: • Neun-Feld-Dateien (engl. *nine-patch files*) Für eine PNG-Datei werden Bereiche festgelegt, die bei Größenveränderungen unverändert bleiben bzw. in vertikaler und/oder horizontaler Richtung gestreckt werden sollen. • Zustandslisten Für Zustände von Steuerelementen (z.B. normal, gedrückt) wird jeweils eine Bitmap-Datei vereinbart. Weitere Details werden in Abschnitt 6.7 behandelt.	**drawable** 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.	**layout** XML
Menüdefinition	XML-Deklaration für ein Optionen- oder Kontextmenü	**menu** XML
Icons für den Anwendungsstarter (Launcher)	Icons für den Anwendungsstarter (engl. *launcher*) von Android sollten von sonstigen zeichenbaren Ressourcen getrennt abgelegt werden. Das betrifft bei unseren bisherigen Android-Projekten die Datei **ic_launcher.png** mit dem voreingestellten App-Icon. Bei hochaufgelösten Launcher-Icons im Ordner **res/drawable** ist zu befürchten, dass sie von Programmen zur Ressourcen-Optimierung eliminiert werden.[1]	**mipmap** JPG, PNG, etc.
Beliebige Dateien	Hier werden beliebige Dateien abgelegt, die über eine Ressourcen-ID ansprechbar sein sollen (z.B. Audios, Videos).	**raw**
Einfache Ressourcen wie Zeichenfolgen, Dimensionsangaben, Stile	Alle bisher beschriebenen Ressourcen steckten 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 zusammengefasst. Man verwendet allerdings in der Regel für jede Sorte von Einfachressourcen (z.B. Zeichenfolgen, Dimensionsangaben, Stile) jeweils eine eigene XML-Datei (z.B. **strings.xml**, **dimens.xml**, **styles.xml**). Ein Stil fasst eine Reihe von Eigenschaftsausprägungen (z.B. Schriftart, Hintergrundfarbe) zusammen und kann **View**-Objekten zugewiesen werden.	**values** XML

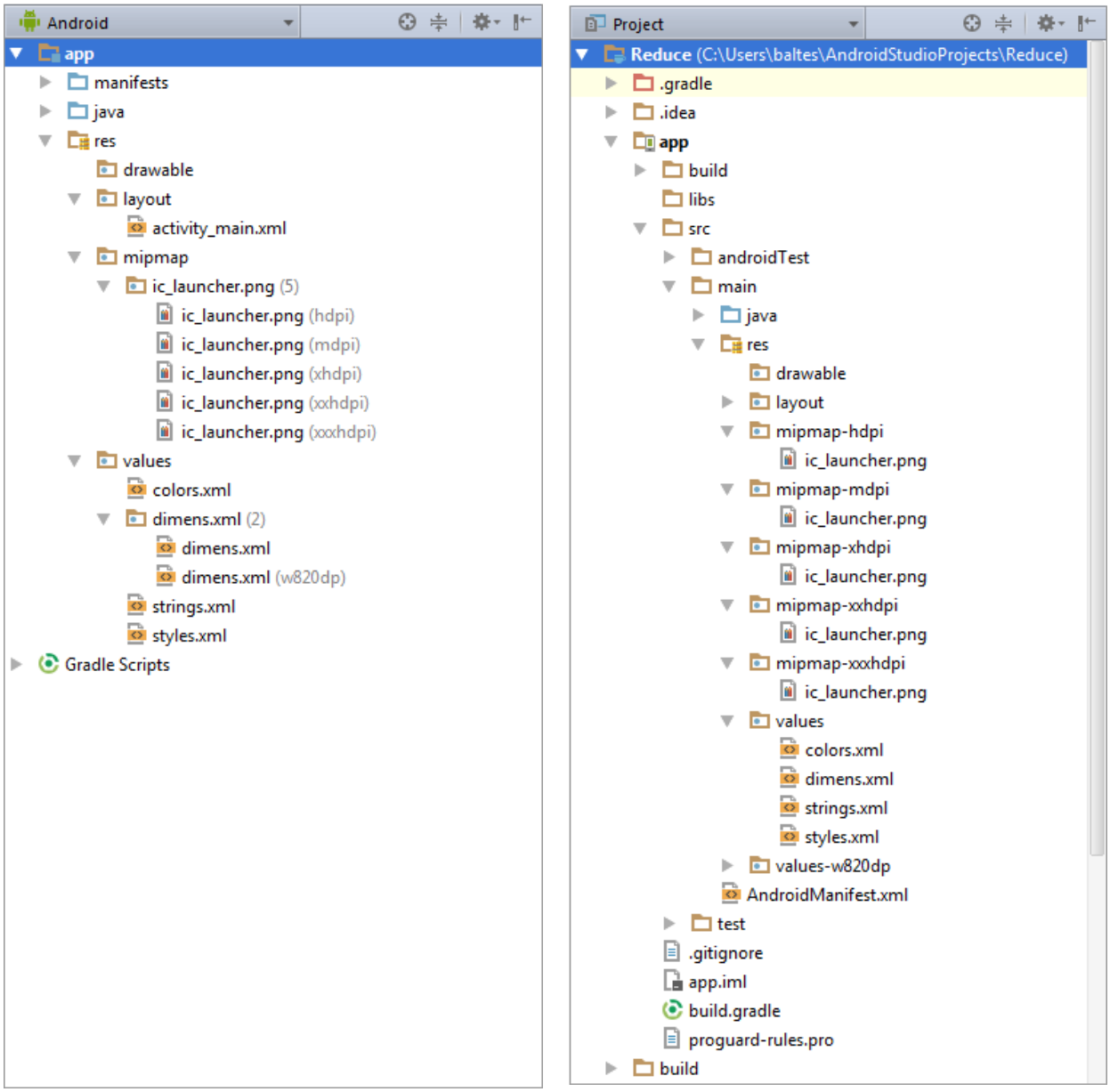
Komplexe Ressourcen (z.B. Animationen, Bitmaps, Layoutdefinitionen) befinden sich in einer eigenen Datei. In der obigen Tabelle sind bis auf die letzte Zeile ausschließlich komplexe Ressourcen enthalten.

[1] Siehe: http://developer.android.com/tools/projects/index.html#mipmap

Das **Project**-Fenster im Android Studio bietet mehrere Sichten, von denen die beiden wichtigsten im folgenden Bildschirmfoto für das in Kapitel 3 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 ein Knoten

res/midmap/ic_launcher.png

mit mehreren gleichnamigen Dateien als Einträgen vorhanden, wobei jeweils in Klammern angegeben ist, für welche Display-Auflösung eine Icon-Datei gedacht ist (vgl. Abschnitt 6.2 zu konfigurationsabhängigen Ressourcen). In der alternativen Sicht **Project** ist die Ordnerstruktur des Dateisystems zu sehen. Dabei zeigt sich, dass sich die Auflösungs-spezifischen Launcher - Icon-Dateien in unterschiedlichen Ordnern befinden:



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) zu verwenden (siehe Abschnitt 6.3), 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 **dimens.xml** aus unserem Projekt `Reduce` enthält z.B. den horizontalen und vertikalen Rand für den Bildschirmauftritt einer Aktivität (zur Einheit **dp** siehe Abschnitt 6.5):

```
<resources>
    <!-- Default screen margins, per the Android Design guidelines. -->
    <dimen name="activity_horizontal_margin">16dp</dimen>
    <dimen name="activity_vertical_margin">16dp</dimen>
</resources>
```

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

```
<?xml version="1.0" encoding="utf-8"?>
<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 und dabei alle Attribute mit Werten versorgen, zu denen ein **item**-Element vorhanden ist. Die **style**-Ressource `AppTheme` dient allerdings dazu, in der Manifestdatei zum Projekt `Reduce` das Erscheinungsbild der 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 6.3). 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 **res/values** (Default-Varianten) oder in einem konfigurationsspezifischen Ordner (siehe Abschnitt 6.2). Es hat sich eingebürgert, für jede Sorte einfacher Ressourcen eine eigene Datei anzulegen und nach dem Typ der enthaltenen Ressourcen zu benennen (z.B. **dimens.xml**, **strings.xml**, **styles.xml**, **colors.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 **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.

6.2 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 und Orientierungen der Hardware
- Unterschiedliche Auflösungen (Pixeldichten) 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 6.1 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. eine Datei **strings.xml** mit den Zeichenfolgen der App im Ordner **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 **res/values-de**.

In der folgenden Tabelle sind wichtige Konfigurationen bzw. Qualifizierer erklärt:[1]

Konfiguration	**Beschreibung**	**Qualifiziererwerte**
Sprache und Region	Die Sprache wird per Doppelbuchstabencode definiert (nach ISO 639-1). Danach folgen optional ein Bindestrich, der Kleinbuchstabe *r* sowie ein Doppelbuchstaben-Regionalcode (nach ISO 3166-1-alpha-2).	Beispiele: **en** **de** **fr-rFR** **fr-rCA**
Bildschirmgröße	Man kann die Bildschirmgröße über die Breite der schmalsten Seite oder über aktuelle Breite definieren. Die Breite der schmalsten Seite spricht man nach dem Muster **sw<N>dp** an (mit *sw* für *smallest width*). Google empfiehlt ... • für 7" - Tablets: **sw600dp** • für 10" - Tablets: **sw720dp** Man könnte z.B. XML-Dateien zur Layoutdefinition in den Ordnern **layout-sw600dp** und **layout-sw720dp** ablegen.	Beispiele: **sw600dp** **sw720dp** **w820dp**

[1] Eine vollständige Liste ist hier zu finden: http://developer.android.com/guide/topics/resources/providing-resources.html

Konfiguration	Beschreibung	Qualifiziererwerte
	Die aktuelle Breite spricht man nach dem Muster **w<N>dp** an. Das Android Studio 2.x legt z.B. im Ordner **values-w820dp** die Datei **dimens.xml** mit Größendefinitionen für Geräte mit einer Breite mindestens 820 dp an. Diese Breite ist auf 7" - und 10" - Tablets im Querformat verfügbar (ca. 960 bzw. 1280 dp).	
Bildschirm-orientierung	Die beiden Werte stehen für den Portrait- bzw. Landschaftsmodus.	**port** **land**
Pixel-Dichte des Displays	Kategorisiert die Displays nach der Pixeldichte:[1] **ldpi**: Low density, ca. 120 dpi. **mdpi**: Medium density, ca. 160 dpi. **hdpi**: High density, ca. 240 dpi **xhdpi**: Extra high density, ca. 320 dpi **xxhdpi**: Extra high density, ca. 480 dpi **nodpi**: Dieser Qualifizierer ist für Bitmaps gedacht, die *nicht* skaliert werden sollen. **tvdpi**: Ist für TV-Geräte gedacht (ca. 213 dpi) und für die meisten Apps irrelevant. Bei Bitmaps im Ordner **res/drawable** geht Android von einer Eignung für die mittlere Auflösung aus und skaliert sie ggf. hoch.	**ldpi** **mdpi** **hpdi** **xhdpi** **xxhdpi** **nodpi** **tvdpi**
API-Level	Über diesen Qualifizierer kann man Ressourcen speziellen API-Levels zuordnen.	Beispiele: **v15** **v23**

Zu einem Ressourcentyp sind *mehrere* Qualifizier erlaubt, die jeweils durch einen Bindestrich getrennt anzugeben sind. Dabei ist die Reihenfolge aus obiger Tabelle einzuhalten (Bach 2012, S. 152), so dass z.B. eine Bitmap (Ressourcentyp **drawable**) zur Verwendung auf hoch aufgelösten Displays durch französisch sprechende Kanadier im folgenden Ordner abzulegen ist:

res/drawable-fr-rCA-hdpi

6.3 Ressourcen ansprechen

Zugriffe auf eine Ressource können an zwei Stellen stattfinden:

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

6.3.1 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.

In unserem Projekt `Reduce` (siehe Kapitel 3) 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** zeigt:

[1] Quelle: http://developer.android.com/design/style/devices-displays.html

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Reduce"
    android:id="@+id/button"
    android:layout_marginTop="30dp"
    android:textAllCaps="false" />
```

Nun korrigieren wir unseren Fehler und ergänzen in der Datei **strings.xml** im Vergleich zum Projektentwicklungsstand in Kapitel 3 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 Layoutdefinitionsdatei **activity_main.xml** (eine komplexe Ressource vom Typ Layout), um die Beschriftung des Befehlsschalters zu vereinbaren:

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/reduce"
    android:id="@+id/button"
    android:layout_marginTop="30dp"
    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 die Einsatzressource. Erforderlich ist der Paketname z.B. bei der häufig praktizierten Verwendung von Ressourcen aus dem Paket **android**:

```
android:divider="@android:drawable/divider_horizontal_dark"
```

Verwendet man in einer Ressourcendefinitionsdatei eine in der Klasse **R.style** im Paket **android** definierte Stilressource, dann müssen im Stilnamen alle Unterstriche durch Punkte ersetzt werden, so dass z.B. die Ressource mit dem Namen **Widget_ProgressBar_Horizontal** folgendermaßen anzusprechen ist:

```
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 kompletten App oder einer Activity zugewiesen wird (zur Definition von Stilen siehe Abschnitt 6.1). Im Vergleich zum oben beschriebenen Ressourcenzugriff startet man mit einem Fragezeichen und kann den einzig zulässigen Ressourcentyp **attr** weglassen:

?[*paket_name*:][attr/]*attribut_name*

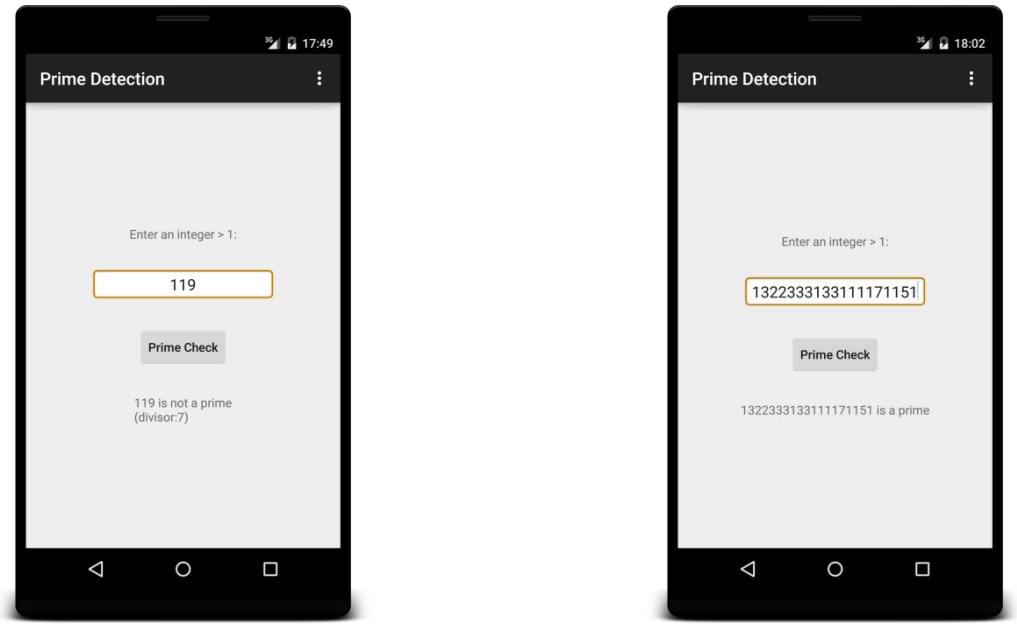
Das folgende Beispiel stammt von der Google-Webseite für Android-Entwickler:[1]

```
<EditText id="text"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:textColor="?android:textColorSecondary"
    android:text="@string/hello_world" />
```

[1] http://developer.android.com/guide/topics/resources/accessing-resources.html

6.3.2 Zugriff im Java-Quellcode

In Abschnitt 6.10 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:[1]

```
<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 Klasse **R** angesprochen, die von der Entwicklungsumgebung (genauer gesagt: vom **Android Asset Packaging Tool**, **AAPT**) aus den Ressourcendefinitionsdateien des Projekts (z.B. **strings.xml**) erstellt wird. Im folgenden Segment aus der **onClick()** - Methode der **Activity** - Klasse in der Primzahlen-App werden mehrere Zeichenfolgen über ihre mit Hilfe der Klasse **R** gebildeten Ressourcen-IDs angesprochen:

```
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));
```

Die Ressourcen-IDs (z.B. `R.string.falseArgument`) bezeichnen in statischen Mitgliedsklassen von **R** statische und finalisierte **int**-Felder, z.B. in der Klasse **R.string**:[2]

---

[1] Falls Sie sich über das nicht-quotierte „>“ - Zeichen in der XML-Datei wundern: An dieser Stelle ist alles erlaubt, außer „<“ und „&“.

[2] 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).

```
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:[1]

```
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);
```

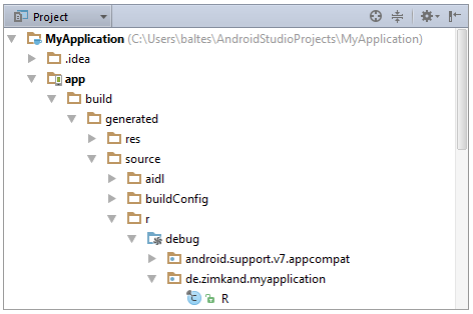
Auch wenn zu einem Ressourcentyp (z.B. **drawable**) konfigurationsabhängige **res**-Unterordner existieren, die im Namen einen oder mehrere Qualifizierer enthalten, wird in der Ressourcen-ID grundsätzlich der Default-Ordnername angegeben. Schließlich soll die Wahl zwischen den alternativen Varianten zu einem Ressourcentyp dem Betriebssystem überlassen werden.

Die Klasse **R** wird beim Übersetzen des Projekts vom AAPT automatisch produziert, so dass manuelle Änderungen dieser Klasse sinnlos sind. Das Android Studio gibt sich einige Mühe, die Klasse **R** vor den neugierigen Blicken der Entwickler zu verstecken:

---

[1] 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);
}
```



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;
        .  .  .
}
```

6.3.3 Element-IDs

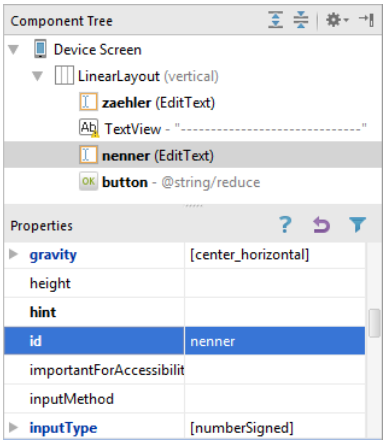
Die eben im Abschnitt 6.3.2 behandelte Klasse **R** enthält auch eine statische Mitgliedsklasse namens **id**, welche in statischen und finalisierten Feldern mit Zugriffsstufe **public** Element-IDs zur Identifikation von Steuerelementen bereit hält, z.B. in unserem Android-Projekt `Reduce`:

```
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 = (EditText) findViewById(R.id.nenner);
```

Vermutlich können Sie sich darin erinnern, dass wir diese Bezeichner im **Design**-Modus der Layout-Bearbeitung per **Properties**-Fenster vereinbart haben, z.B.:



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:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:inputType="numberSigned"
    android:ems="10"
    android:id="@+id/nenner"
    android:gravity="center_horizontal"
    android:singleLine="false"
    android:width="100dp"
    android:background="@android:drawable/editbox_background" />
```

Mit **@+id** wird das AAPT angewiesen, eine neue Element-ID anzulegen.

6.4 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>
```

Nun erstellen wir eine Variante der Datei **strings.xml** mit den deutschen Übersetzungen im Ordner

res/values-de

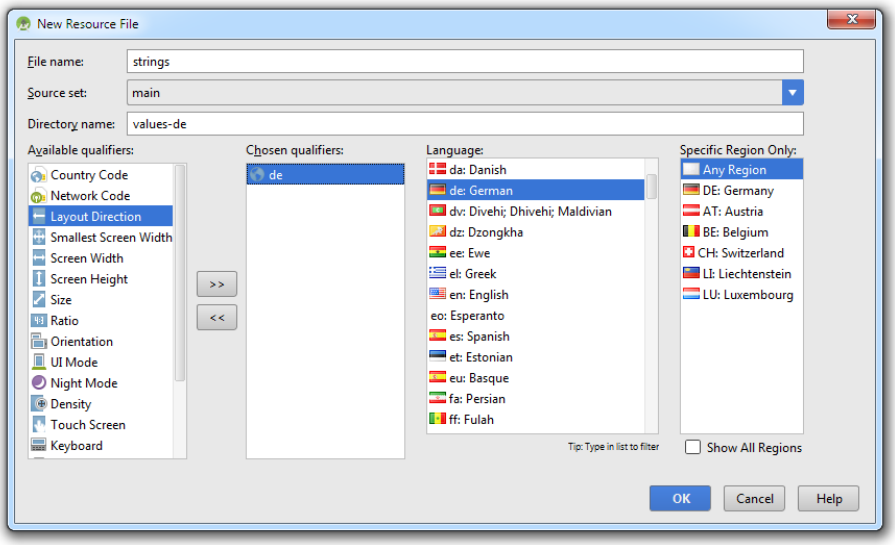
Dazu wählen wir im Android Studio aus dem Kontextmenü zum Knoten

res/values

in der **Android**-Sicht des **Project**-Fensters das Item

New > Values resource file

und tragen im folgenden Dialog

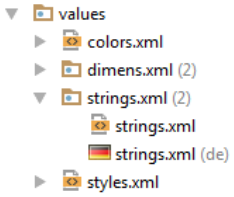


ein:

- **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.

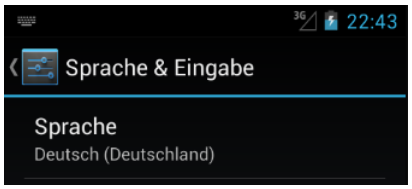
Wir öffnen die nett beflaggte neue **strings.xml** - Variante



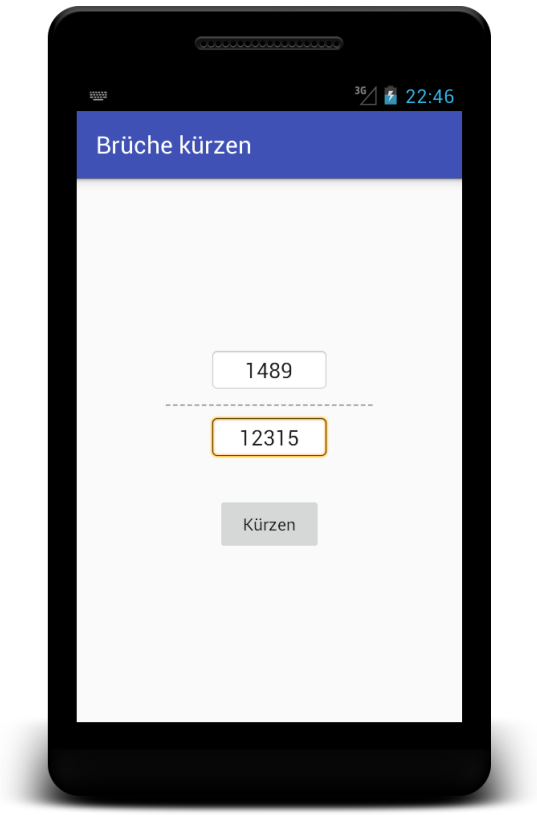
per Doppelklick und übersetzen die Beschriftungen von App und Befehlsschalter, z.B.:

```
<resources>
    <string name="app_name">Brüche kürzen</string>
    <string name="reduce">Kürzen</string>
</resources>
```

Wenn die App anschließend auf einem Android-Gerät mit deutscher Spracheinstellung



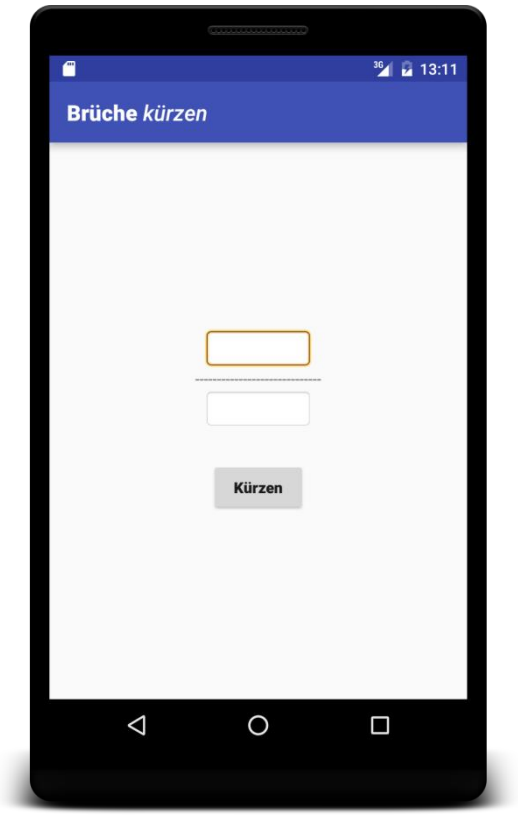
läuft (wählbar über **Settings > Language & input > Language**), dann spricht sie Deutsch:



Mit den von HTML gewohnten Elementen **<b>**, **<i>** und **<u>** für die Auszeichnungen **fett**, *kursiv* und <u>unterstrichen</u> lassen sich Textformatierungen vereinbaren, z.B.:

```
<resources>
    <string name="app_name"><b>Brüche</b> <i>kürzen</i></string>
    <string name="reduce"><b>Kürzen</b></string>
</resources>
```

Im Beispiel wirkt sich die Modifikation auf die Titelzeile der Aktivität und den Schalter aus:



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.[1] Allerdings überraschen die **android**-Ressourcen **yes** und **no** (in Deutsch) mit den Werten *OK* und *Abbrechen*. Etwas mehr Arbeit, aber mehr Kontrolle und eine erheblich größere Zahl an übersetzten Zeichenfolgen hat man durch Übernahme der SDK-Datei **strings.xml** (z.B. **...\sdk\platforms\android-23\data\res\values**) mit den benötigten Sprachvarianten in ein eigenes Projekt.

6.5 Größenangaben

Größenangaben (z.B. Randabstände) sollten in einer Ressourcendatei mit dem **dimens.xml** gesammelt und in der App referenziert werden, um Einheitlichkeit und bequeme Modifizierbarkeit zu erreichen. Ein frisch vom Android Studio 2.x erstelltes Projekt enthält zwei solche Dateien in den Ordnern:

- **.../res/values-w820dp**
  Hier werden Größendefinitionen für Geräte mit einer Breite von mindestens 820dp abgelegt.
- **.../res/values**
  Hier werden Größendefinitionen für schmalere Geräte abgelegt:

[1] Hier sind die in **android.R.string** definierten Ressourcen dokumentiert.

```
<resources>
    <!-- Default screen margins, per the Android Design guidelines. -->
    <dimen name="activity_horizontal_margin">16dp</dimen>
    <dimen name="activity_vertical_margin">16dp</dimen>
</resources>
```

Das Attribut **activity_horizontal_margin** legt z.B. den Abstand des Aktivitätstitels zum linken Display-Rand fest. Es eignet sich dazu, um einem Steuerelement denselben Randabstand zu verpassen und so Linksbündigkeit in Relation zum Aktivitätstitel herzustellen.

Bei Größenangaben zu Elementen einer Android-Bedienoberfläche sind folgende Maßeinheiten zulässig:[1]

px	Ein reales Pixel Diese Maßeinheit ist *nicht* empfehlenswert, weil sich Android-Geräte bzgl. der Pixelgröße stark unterscheiden.
dp **dip**	Ein auflösungsunabhängiges Pixel (*density independent pixel*) 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 **dp** zu empfehlen.
sp	Bei der Einheit **sp** (*scaled pixel*) wird im Unterschied zur Einheit **dp** auch die vom Benutzer bevorzugte Schriftgröße (**Einstellungen > Anzeige > Schriftgröße**) berücksichtigt. Daher sollte die Spezifikation von Schriftgrößen in der Einheit **sp** erfolgen.
pt	Ein **pt** entspricht 1/72 Zoll, also ungefähr 0,35 mm.
in	Ein Inch (Zoll) entspricht ungefähr 25,4 mm.
mm	Millimeter

6.6 Farben

Zur Definition von Farben, die an mehreren Stellen in einer App (z.B. bei mehreren Activities) verwendet werden, sollten Ressourcen 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

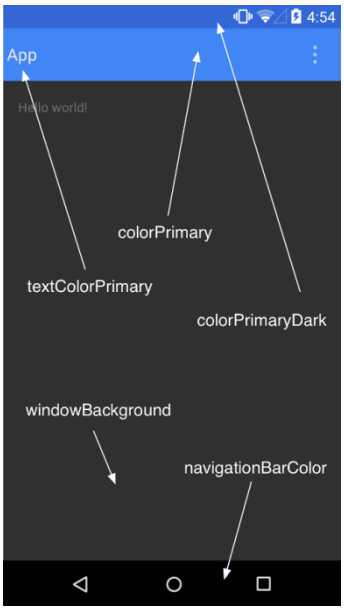
Im folgenden Beispiel wird durch eine Modifikation der Datei **colors.xml** für die Bruchkürzungs-App die Farbe der Titelzeile verändert:

```
<resources>
    <color name="colorPrimary">#7AB53F</color>
    <color name="colorPrimaryDark">#303F9F</color>
    <color name="colorAccent">#FF4081</color>
</resources>
```

Wo die vom Assistenten für neue Apps definierten Farbressourcen Verwendung finden, ist in der folgenden, von einer Google-Webseite für Android-Entwickler übernommen, Abbildung dokumentiert:[2]

---

1 Quelle: http://developer.android.com/guide/topics/resources/more-resources.html#Dimension
2 http://developer.android.com/training/material/theme.html



Im Beispiel resultiert das folgende Ergebnis:



6.7 Zeichenbare Ressourcen

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.[1]

6.7.1 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. Eckgestaltung) zu beschädigen.

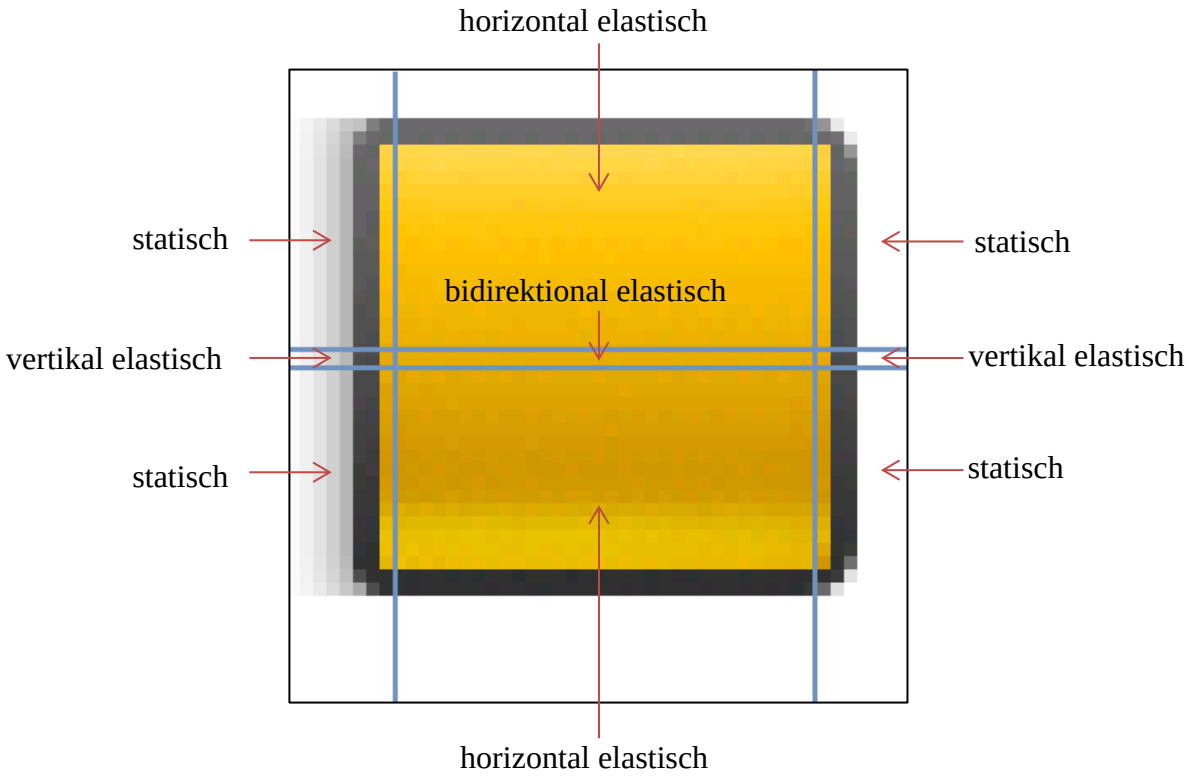
Eine Neun-Feld-Bitmap (engl. *nine-patch bitmap*) entsteht aus einer PNG-Datei, indem neun Rechtecke festgelegt werden, die bei einer Größenänderung ...

- 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

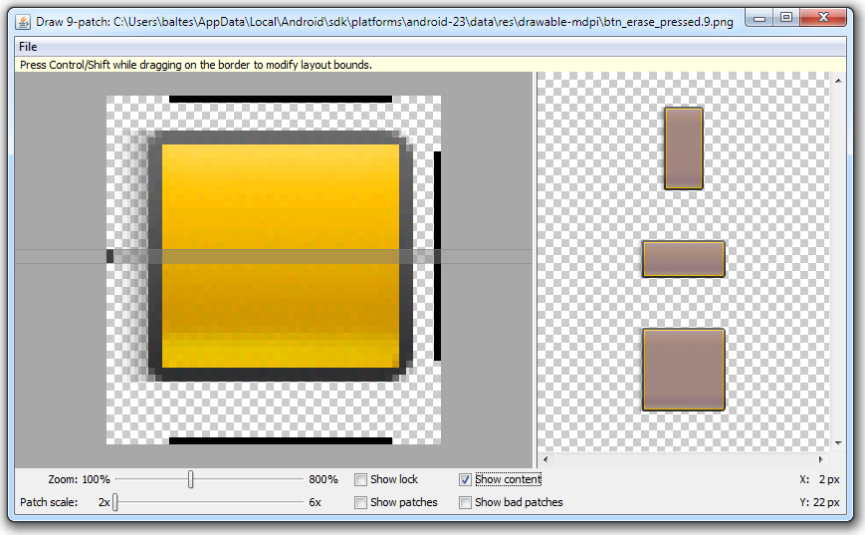
Die Ecken sind statisch, das Zentrum ist bidirektional elastisch, und die restlichen Zonen sind in einer Richtung elastisch:

---

[1] http://developer.android.com/guide/topics/resources/drawable-resource.html



Um diese 9 Zonen zu definieren, erhält eine PNG-Datei einen umlaufenden Rand mit einem Pixel Breite. Mit dem linken Rand wird ein Zeilenbereich ausgewählt, der bei einer vertikalen Vergrößerung gestreckt werden darf:



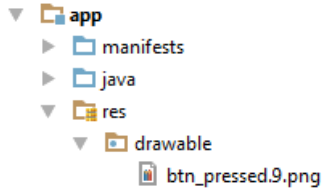
Mit dem oberen Rand wird ein Spaltenbereich ausgewählt, der bei einer horizontalen Vergrößerung gestreckt werden darf.

Die optionalen Markierungen am rechten und unteren Rand tragen *nicht* zur Neun-Feld-Aufteilung bei, sondern definieren den frei zu haltenden Innenrand (engl.: *padding area*) bzw. den Inhaltsbereich (z.B. zur Aufnahme einer Beschriftung). Im rechten Teil des obigen Bildschirmfotos ist für drei Dehnvarianten der Neun-Feld-Fläche der Inhaltsbereich durch eine dunkle Färbung markiert. Zur Erstellung einer Neun-Feld-Datei (mit der Namenserweiterung **.9.png**) aus einer PNG-Datei enthält das Android-SDK im Unterordner **tools** ein als ausführbares Java-Archiv realisiertes Programm namens **draw9patch.jar**, das unter Windows über die im gleichen Ordner befindliche Datei **draw9patch.bat** zu starten ist. Im obigen Bildschirmfoto ist eine Neun-Feld - Beispieldatei aus dem SDK-Unterordner

platforms\android-23\data\res\drawable-mdpi

geöffnet worden, was übrigens ganz bequem per Drag&Drop zu erledigen ist.

Eine Neun-Feld-Datei wird wie jede andere zeichenbare Ressource im Projekt-Unterordner **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:layout_width="match_parent"
    android:layout_height="match_parent"
    android:text="@string/reduce"
    android:id="@+id/button"
    android:textAllCaps="false"
    android:background="@drawable/btn_pressed"
    android:layout_marginTop="30dp"
    android:textSize="30sp" />
```

Um Größenveränderungen zu provozieren, darf im Beispiel der Schalter die volle Breite und Höhe des umgebenden Containers nutzen. Anschließend ist die Neun-Feld-Bitmap stark horizontal gedehnt zu sehen:

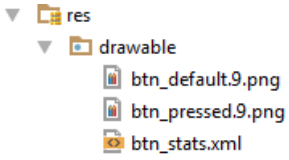


6.7.2 Zustandslisten

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 6.1 erwähnt wurden, lässt sich eine Zustandsliste (engl. *state list*) definieren, die für jeden Zustand eine zeichenbare Ressource (z.B. eine Neun-Feld-Datei) 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>
```

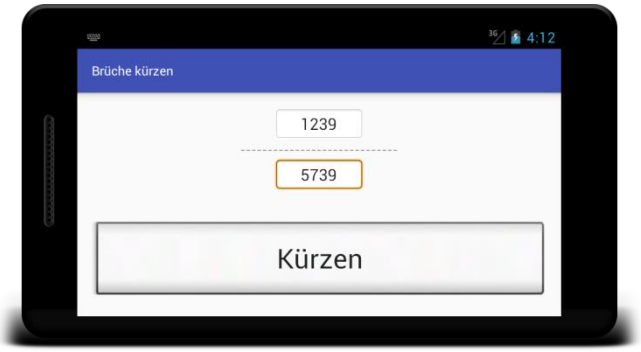
Eine XML-Datei mit einer Zustandsliste wird als zeichenbare Ressource im Ordner **res/drawable** abgelegt, z.B.:



Wird diese Zustandsliste einem Steuerelement als Wert des Attributs **android:background** zugewiesen,

```
<Button
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:text="@string/reduce"
    android:id="@+id/button"
    android:textAllCaps="false"
    android:background="@drawable/btn_stats"
    android:layout_marginTop="30dp"
    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 Bruchkürzungs-App ist im folgenden Ordner zu finden:

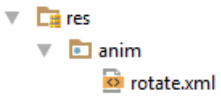
...\BspUeb\Reduce\Abschnitt 6.7

6.8 Animationen

Beim Thema *Animationen* müssen wir uns aus Zeitgründen auf ein einfaches Beispiel beschränken. In der Bruchkürzungs-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 6.1 über Ressourcentypen) zu realisieren, legen wir den Projektordner **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:

```
Animation rotate;
```

In der Methode **onCreate()** wird die Animation unter Verwendung des Ressourcennamens mit Hilfe der statischen Mitgliedsklasse **R.anim** geladen:

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

Schließlich erhält der Befehlsschalter am Ende der Klickereignisbehandlung die Anweisung, die Animation auszuführen:

```
@Override
public void onClick(View v) {
    EditText etZaehler = (EditText) findViewById(R.id.zaehler);
    EditText etNenner = (EditText) findViewById(R.id.nenner);
     . . .
    etZaehler.setText(Integer.toString(z));
    etNenner.setText(Integer.toString(n));

    v.startAnimation(rotate);
}
```

Ausführliche Informationen über Animationen in Android finden sich z.B. im Google-Webangebot für Android-Entwickler.[1]

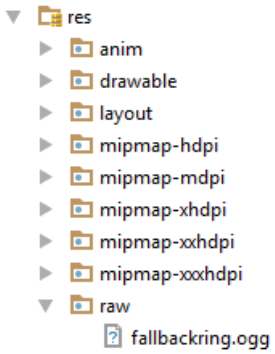
[1] http://developer.android.com/guide/topics/resources/animation-resource.html

6.9 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 Projektordner **res/raw** an und kopieren aus den SDK-Beständen die Datei **fallbackring.ogg**

sdk\platforms\android-23\data\res\raw\fallbackring.ogg

mit einem kurzen Sound im Ogg-Format dorthin:



Im Quellcode der Aktivitätsklasse wird eine Instanzvariable vom Typ **MediaPlayer** definiert:

```
MediaPlayer player;
```

In der Methode **onCreate()** wird das **MediaPlayer**-Objekt kreiert und über die abzuspielende Datei unter Verwendung des Ressourcennamens unter Beteiligung der statischen Migliedsklasse **R.raw** informiert:

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

Schließlich erhält das **MediaPlayer**-Objekt am Ende der Klickereignisbehandlung zum Befehlsschalter die Anweisung, die Soundwiedergabe zu starten:

```
@Override
public void onClick(View v) {
    EditText etZaehler = (EditText) findViewById(R.id.zaehler);
    EditText etNenner = (EditText) findViewById(R.id.nenner);
     . . .
    etZaehler.setText(Integer.toString(z));
    etNenner.setText(Integer.toString(n));

    player.start();
}
```

Ausführliche Informationen zur Klasse **MediaPlayer** bietet z.B. die SDK-Dokumentation.[1] Über die von Android unterstützten Medienformate informiert eine Google-Webseite für Android-Entwickler.[2]

---

1 http://developer.android.com/reference/android/media/MediaPlayer.html
2 http://developer.android.com/guide/appendix/media-formats.html

Das komplette AS-Projekt mit dem aktuellen Entwicklungsstand der Bruchkürzungs-App ist im folgenden Ordner zu finden:

...\BspUeb\Reduce\Abschnitt 6.9

6.10 Entspannung und Motivationsstärkung

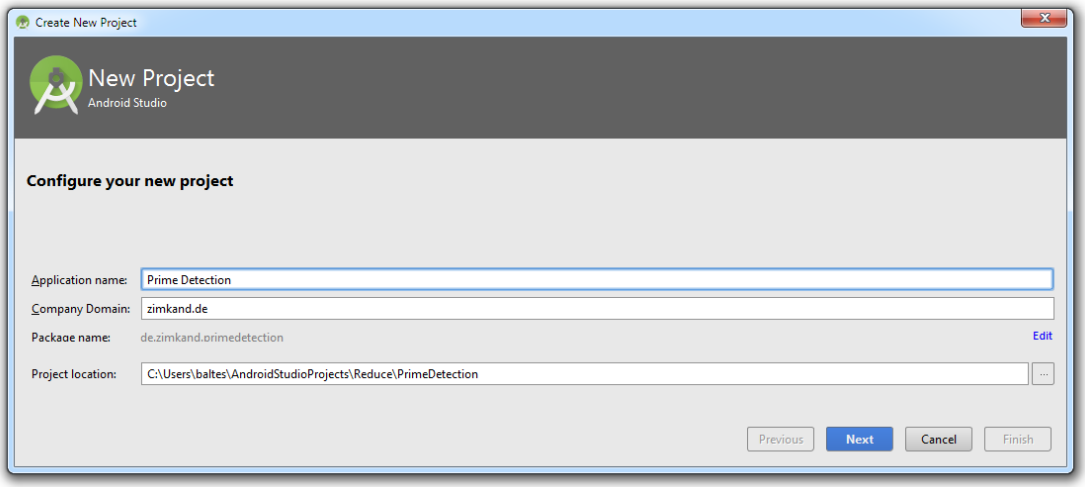
Nach etlichen recht anstrengenden Themen in den Kapiteln 4, 5 und 6 soll dieser Abschnitt zur Entspannung und zur Regeneration Ihrer Motivation beitragen. Wir erstellen eine Android-App, die für eine ganze Zahl größer 1 überprüft, ob es sich um eine Primzahl handelt.

6.10.1 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**:



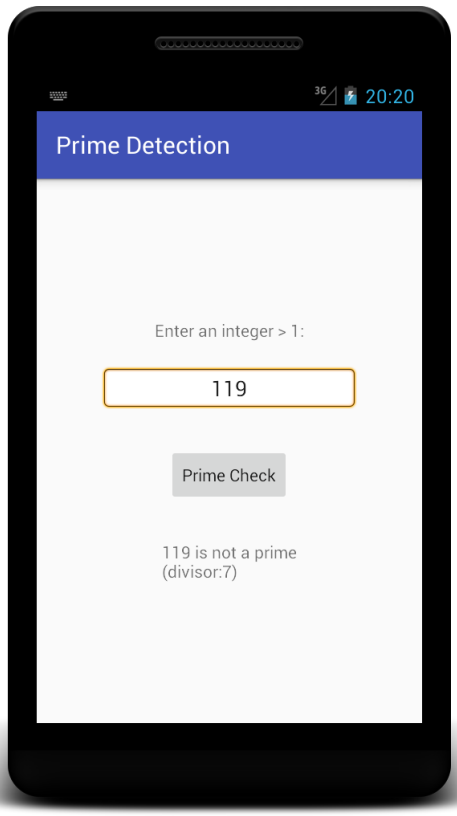
In den beiden folgenden Dialogen akzeptieren wir jeweils die Voreinstellung:

- 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). Nach einem Klick auf den Schalter **Finish** beginnt das Build-System Gradle damit, das Gerüst der neuen App zu erstellen.

6.10.2 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 für die Instruktion
- ein Texteingabefeld für die Primzahlkandidaten
- einen Schalter zur Anforderung der Primzahlprüfung
- ein Label für die Ergebnisausgabe

Auch bei diesem Layout genügt als „Platzanweiser“ für die Steuerelemente ein Objekt der simplen Klasse **LinearLayout**. Wechseln Sie zum **Text**-Modus des Editors, und ersetzen Sie im XML-Code den Elementnamen **RelativeLayout** durch die Alternative **LinearLayout**:

```
<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: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=".MainActivity">

    <TextView android:text="@string/hello_world" android:layout_width="match_parent"
        android:layout_height="wrap_content" />

</LinearLayout>
```

Das Android Studio zeigt übrigens an Stelle der Zugriffe auf anderweitig definierte Ressourcen (z.B. `@dimen/activity_horizontal_margin`, vgl. Abschnitt 6.3.1) in grün hinterlegter Schrift den aufgelösten Wert an (z.B. `"16dp"`).

Nach der Rückkehr zum **Design**-Modus ist im Fenster **Component Tree** der neue Platzanweiser zu sehen, und im **Properties**-Fenster kann man dem **LinearLayout** ...

- für die Eigenschaft **orientation** den Wert **vertical**
- und für die Eigenschaft **gravity** den Wert **center** verpassen.

Das vorhandene **TextView**-Objekt kann bleiben, benötigt aber den neuen Text „Enter an integer > 1“, der selbstverständlich über die Datei **strings.xml** im Projektordner **res/values** als Zeichenfol-

gen-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, so dass die schon in Abschnitt 6.3.2 als Beispiel verwendete Datei entsteht:[1]

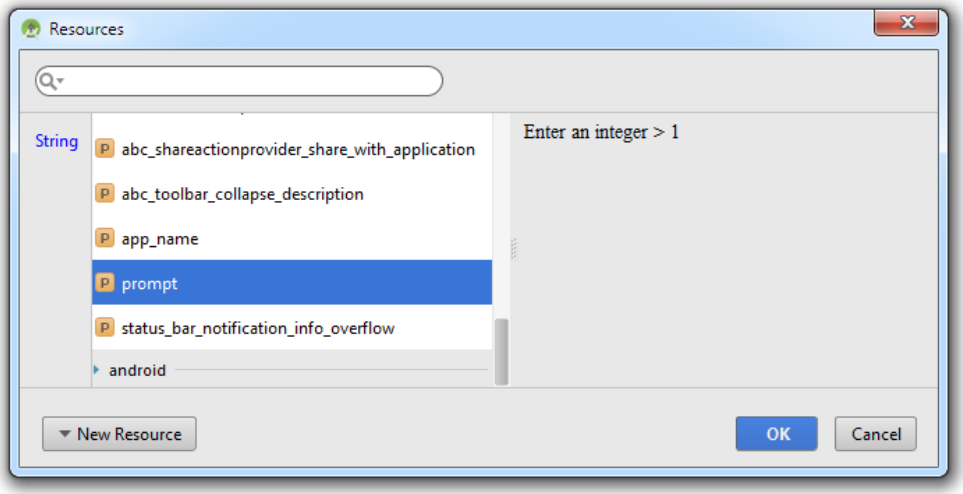
```
<resources>
    <string name="app_name">Prime Detection</string>
    <string name="prompt">Enter an integer > 1:</string>
    <string name="diag">Prime Check</string>
    <string name="prime">is a prime.</string>
    <string name="divisor">divisor:</string>
    <string name="noPrime">is not a prime.</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:

- Nach einem Doppelklick auf das das Element in der Layout-Vorschau erscheint das folgende Fenster

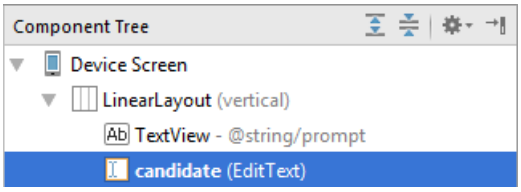


- Nach einem Klick auf den Erweiterungsschalter ... wählt man die passende Zeichenfolge und quittiert mit **OK**:



Wir machen weiter mit dem Texteingabefeld zur Aufnahme der Primzahlkandidaten. Dazu übernehmen wir aus der Abteilung **Text Fields** der **Palette** ein Steuerelement vom Typ **Number**, fügen es per Drag & Drop im Fenster **Component Tree** unter dem vorhandenen **TextView**-Objekt ein und vergeben per **Properties**-Fenster `candidate` als Element-ID:

[1] Falls Sie sich über das nicht-quotierte „>“ - Zeichen in der XML-Datei wundern: An dieser Stelle ist alles erlaubt, außer „<“ und „&“.



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 über die Eigenschaft **ems** mit dem voreingestellten Wert 10 festgelegt. Es ist ein Vielfaches der Breite des Buchstabens *m* anzugeben. Fügen Sie einen oberen Rand hinzu, der für Abstand vom **TextView**-Objekt sorgt (Eigenschaft **layout:margin**, **top**).

Wir verschaffen dem Texteingabefeld eine attraktive Optik durch ein vom Android-System spendiertes Hintergrundbild (siehe Beschreibung in Abschnitt 3.2). Dazu markieren wir das Steuerelement, klicken im **Properties**-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 **Widgets** 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).
- Um zu verhindern, dass im laufenden Programm die Schalterbeschriftung komplett in Großschreibung erscheint, setzen wir die Schaltereigenschaft **textAllCaps** auf den Wert **false**. Diese **expert property** ist erst nach einen Klick auf das Trichter-Symbol erreichbar.
- 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 ein weiteres Objekt **Plain TextView**, das im Programm das Prüfergebnis anzeigen soll und daher `result` als **id** erhält. Sein überflüssiger und störender Initialtext wird gelöscht. Damit ist das zu Beginn des aktuellen Abschnitts beschriebene Designziel erreicht.

6.10.3 Click-Behandlungsmethode für den Schalter erstellen

Wechseln Sie in der Editorzone zum Java-Quellcode der Klasse `PrimeActivity`. Analog zu Abschnitt 3.3 sorgen wir nun 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:

- Tragen Sie am Ende der Methode eine Anweisung ein, die uns eine Ansprachemöglichkeit für den Schalter verschafft:

```
Button 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 Quellcodedatei bekannt gemacht wird.

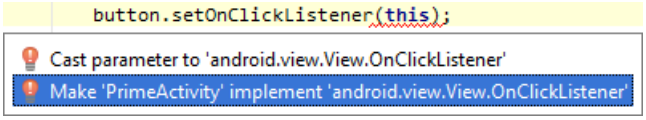
- Bitten Sie den Schalter darum, das Aktivitätsobjekt (angesprochen durch das Schlüsselwort **this**) als Klick-Interessenten zu registrieren:

```
button.setOnClickListener(this);
```

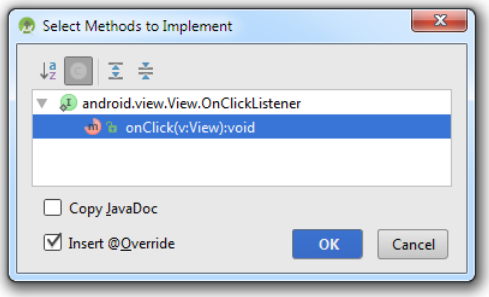
Damit diese Registrierung möglich ist, müssen zwei Voraussetzungen erfüllt sein:

- 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 Einfügemarke auf die reklamierte Syntax (auf das Schlüsselwort **this**) und drücken die Tastenkombination **Alt+Enter**. Aus der Vorschlagsliste wählen wir das zweite Item, um den Klassendefinitionskopf zu erweitern:



Anschließend bietet der folgende Dialog an, einen Rohling für die benötigte Ereignisbehandlungsmethode zu erstellen:



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 **EditText**-Steuerelement am Ende des Layouts (Kennung: `result`) schreiben, so dass wir entsprechende Referenzvariablen benötigen. Weil sich an den Ressourcen-Kennungen bzw. Objektadressen nichts ändert, wäre es eine Zeitverschwendung, die Adressen bei jedem **onClick()** - Aufruf neu zu ermitteln. Diese Aufgabe wird daher in der Methode **onCreate()** erledigt.

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 {
    EditText candidate;
    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 = (Button) findViewById(R.id.button);
    button.setOnClickListener(this);
    candidate = (EditText) findViewById(R.id.candidate);
    result = (TextView) 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)+0.5);
    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 6.10.2 liefert das **EditText**-Objekt garantiert eine positive ganze Zahl mit maximal 18 Stellen. Daher können wir 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 ≤ Wurzel):

Sei die ganze Zahl d (≥ 2) ein echter Teiler der positiven, ganzen Zahl z, d.h. es gibt eine ganze Zahl k (≥ 2) mit

$$z = k \cdot d$$

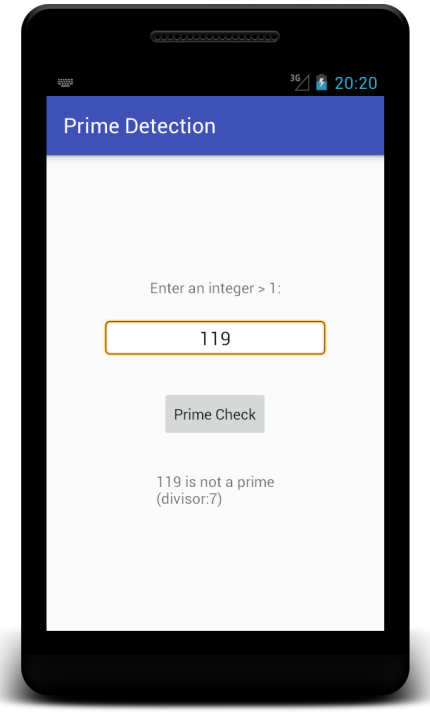
Dann ist auch k ein echter Teiler von z, und es gilt:

$$d \leq \sqrt{z} \quad \text{oder} \quad k \leq \sqrt{z}$$

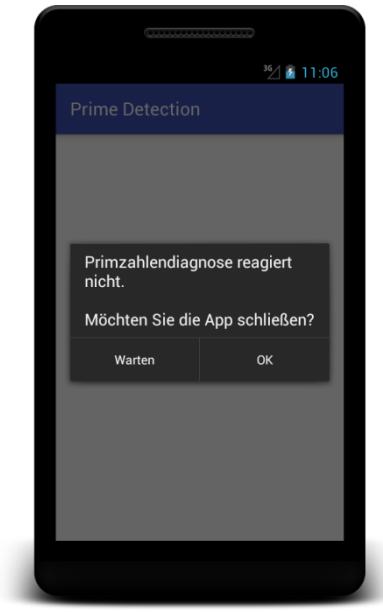
Anderenfalls wäre das Produkt $k \cdot d$ größer als z. Wir haben also folgendes Ergebnis: Wenn eine Zahl z einen echten Teiler hat, dann besitzt sie auch einen echten Teiler kleiner oder gleich $\sqrt{z}$. Wenn man *keinen* echten Teiler kleiner oder gleich $\sqrt{z}$ gefunden hat, kann man die Suche also einstellen, und z ist eine Primzahl.

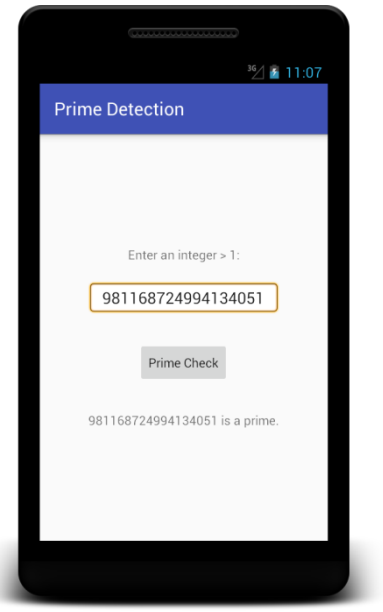
Zur Berechnung der Quadratwurzel verwendet das Beispielprogramm die statische Methode **sqrt()** aus der Klasse **Math**.

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 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 7 und 9) ablaufen lassen, so dass 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.

6.11 Übungsaufgaben zu Kapitel 6

1) Erstellen Sie eine deutsche Version der Zeichenfolgen für die in Abschnitt 6.10 entwickelte Primzahlen-App.

7 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 Sammlung mit Bedienelementen zur Benutzerinteraktion vorhanden (z.B. Befehlsschalter, Kontrollkästchen, Texteingabefelder, Auswahllisten, Menüs). Diese Bedienelemente bezeichnet man als *Steuerelemente*, *widgets* oder *controls*.[1]
- **Ereignisorientierung**
  Das Laufzeitsystem bestimmt als Vermittler oder als Quelle von Ereignissen in erheblichem Maße den Ablauf einer App, indem es Ereignisbehandlungmethoden der App aufruft. Diese Ereignisbehandlungmethoden sind Beispiele für so genannte *Call Back - Routinen*. Ausgelöst werden die Ereignisse in der Regel vom Benutzer, der mit der Hilfe von Eingabegeräten wie Touch Screen oder Tastatur praktisch permanent in der Lage ist, unterschiedliche Wünsche zu artikulieren. Eine App präsentiert mehr oder weniger viele Bedienelemente und wartet die meiste Zeit darauf, dass eine der zugehörigen Ereignisbehandlungmethoden durch ein (meistens) vom Benutzer ausgelöstes Ereignis aufgerufen wird.
- **Single-Thread - Design**
  Alle Ereignisbehandlungmethoden laufen (strikt nacheinander!) im selben Thread (Ausführungsfaden, siehe Kapitel 9) ab, so dass der Zeitaufwand für die Ausführung einer Ereignisbehandlungmethode unbedingt in engen Grenzen bleiben muss. Potentiell zeitaufwändige Aufgaben (z.B. komplexe Berechnungen, Daten von einem Netzwerkdienst besorgen) müssen in einen separaten Thread verlagert werden. In Abschnitt 6.10.3 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 kann andererseits *nur* vom UI-Thread aus zugegriffen werden, so dass 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-Apps gesammelt, die nun systematisiert und abgerundet werden sollen.

[1] 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.

7.1 Layoutdefinition als Ressource

Spätestens seit Kapitel 6 wissen wir, dass unter Android eine Layoutdefinition für eine Aktivität als *Ressource* behandelt wird, die in einer XML-Datei im Projektordner **res/layout** abzulegen ist. Es ist prinzipiell möglich, ein Layout per Java-Quellcode zu definieren (so wie bei der Bedienoberfläche Swing für Java-Desktop-Programme), doch ist die Deklaration via XML wegen der strikten Trennung von Layout und Programmlogik eindeutig zu bevorzugen.[1]

Anschließend ist die (leicht entschlackte) Layoutdefinition zur Activity im Projekt `Prime Detection` zu sehen, die wir in Abschnitt 6.10.2 mit Hilfe der Entwicklungsumgebung erstellt haben:

```
<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: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=".PrimeActivity"
    android:orientation="vertical"
    android:gravity="center"
    android:id="@+id/linLayout">

    <TextView android:text="@string/prompt"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/prompt" />

    <EditText
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:inputType="number"
        android:ems="10"
        android:id="@+id/candidate"
        android:gravity="center_horizontal"
        android:background="@android:drawable/editbox_background"
        android:layout_marginTop="30dp" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/diag"
        android:id="@+id/button"
        android:layout_marginTop="30dp"
        android:layout_marginBottom="30dp"
        android:textAllCaps="false" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/result" />

</LinearLayout>
```

Im Wurzelelement der XML-Datei werden XML-Namensräume definiert:

```
xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:tools="http://schemas.android.com/tools"
```

[1] 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 2016). 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 2012).

Der zu einem XML-Namensraum angegebene URI ist keine reale Internet-Adresse, sondern dient nur zur eindeutigen Identifizierung.[1] Zur Vermeidung von Namenskollisionen wird den meisten Attributen in XML-Dateien aus dem Android-Umfeld ein Namensraumpräfix vorangestellt (im Beispiel: **android:** oder **tools:**).

Im Beispiel ist als Wurzelelement ein Container vom Typ **LinearLayout** vorhanden, der als Kindelemente atomare Steuerelemente aus den folgenden Klassen enthält und linear (im konkreten Fall: vertikal) anordnet:

- **TextView**
- **EditText**
- **Button**

Aus den Elementen der Layoutdefinition entsteht zur Laufzeit eine Hierarchie von Objekten, die zur Klasse **View** oder einer Ableitung gehören (siehe Abschnitt 7.2). 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 so genannten *JavaBeans* zurechnen, weil sie für ihre Eigenschaften Zugriffsmethoden mit genormten Namen bieten (vgl. Abschnitt 13.4.1 in Baltes-Götz & Götz 2016):

- 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()`).

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) 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 guten GUI-Editor besitzt, ist der direkte Kontakt mit der XML-Layoutdefinition nicht unbedingt erforderlich. Allerdings ist oft die direkte Modifikation der (leicht verständlichen) XML-Syntax der schnellste Weg zum Ziel.

Oft ist es erforderlich, für verschieden Gerätegrößen bzw. -orientierungen eigenständige Layoutdefinitionen anzulegen (siehe Abschnitt 6.2 über konfigurationsspezifische Ressourcen). Im Abschnitt 7.3.1 werden wir eine für das Querformat optimierte Alternative zum obigen Layout definieren.

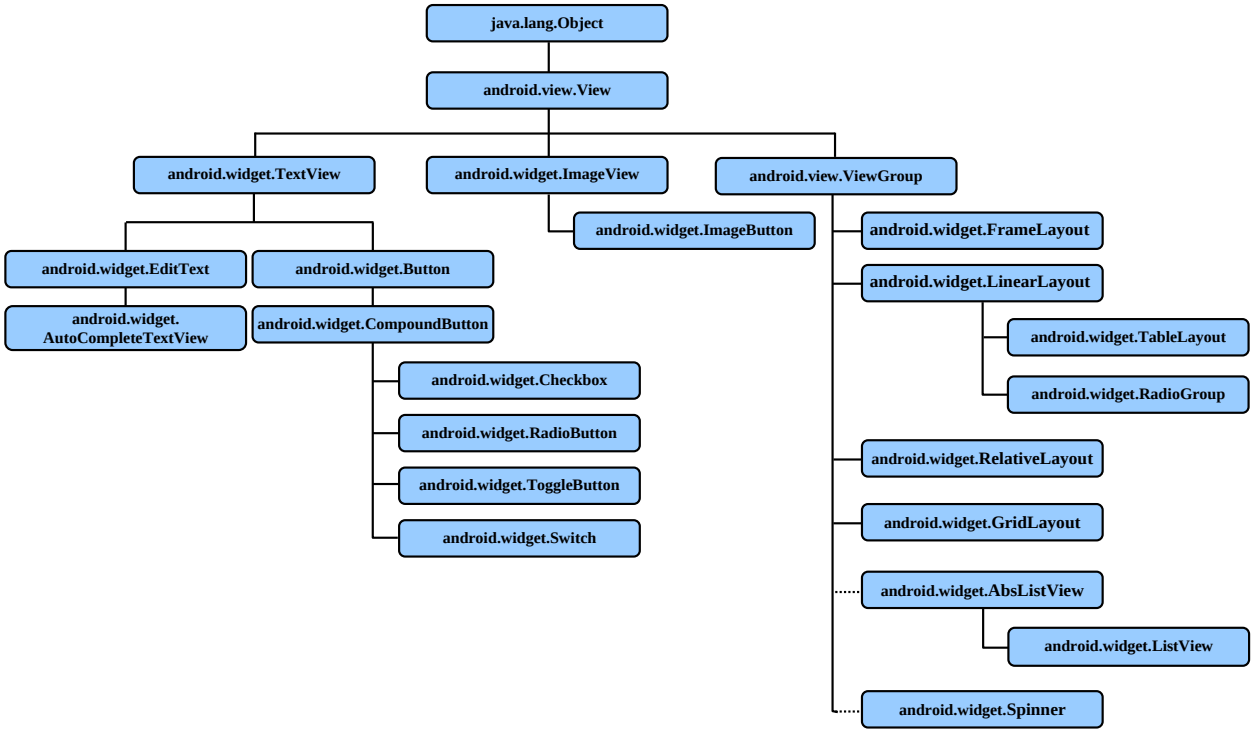
[1] Ein URI (*Universal Resource Identifier*) sieht aus wie ein URL (*Universal Resource Locator*), zeigt aber nicht unbedingt auf ein reales Objekt im Internet.

7.2 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 7.4).
- Sie unterstützen die Verwendung mit Hilfe von visuellen 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:[1]



Während sich die Klassen **View** und **ViewGroup** im Paket **android.view** befinden, gehören die restlichen Klassen zum Paket **android.widget**.

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).

---

[1] Zwischen den Klassen **ViewGroup** und **Spinner** bzw. **AbsListView** wurden der Einfachheit halber Zwischenstufen der Vererbungshierarchie weggelassen.

7.2.1 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.

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 Mitgliedklasse des Container-Typs definiert. Gemeinsame Basisklasse der Layout-Parameter-Klassen ist **ViewGroup.LayoutParams**, und schon die Steuerelement-Basisklasse **View** enthält eine Instanzvariable von diesem Typ. Über das an einem Steuerelement haftende Layout-Parameter-Objekt 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**.

Das **Properties**-Fenster der Entwicklungsumgebung zeigt die XML-Attribute zu einem Steuerelement per Voreinstellung in folgender Anordnung:

- Ganz oben stehen die Layout-Parameter.
- Es folgen wichtige Steuerelementattribute in alphabetischer Reihenfolge.
- Über den Schalter  werden weniger wichtige Attribute ein- bzw. ausgeblendet.

7.2.1.1 Element-ID

Wie ein Steuerelement im Quellcode über seine Element-ID anzusprechen ist, wurde schon in Abschnitt 6.3.3 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 = (EditText) 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 Ressoucendateneien 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 5.1).

7.2.1.2 Breite und Höhe

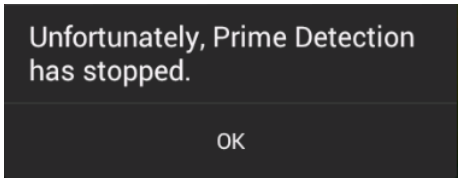
Bei den Attributen

- **android:layout_width** bzw.
- **android:layout_height**

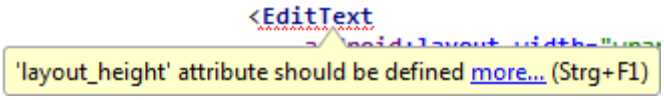
für die Breite bzw. Höhe eines Steuerelements kann man auflösungsabhängige oder (besser!) auflösungsunabhängige Größenangaben machen (vgl. Abschnitt 6.5). Oft sind aber die folgenden Attributausprägungen zu bevorzugen:

- **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 auf diese Weise die gesamte Breite bzw. Höhe des Displays.

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.:



7.2.1.3 Ränder (Margins) und Einrückung (Padding)

Über die **layout_margin**-Attribute schafft man Abstand zwischen einem **View**-Objekt und seinen Nachbarn. Neben dem allseitig wirkenden Attribut **andoid:layout_margin** gibt es die seitenspezifischen Attribute:

- **andoid:layout_marginLeft**
- **andoid:layout_marginRight**
- **andoid:layout_marginTop**
- **andoid:layout_marginBottom**

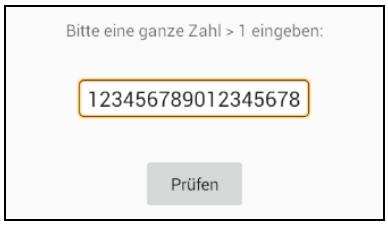
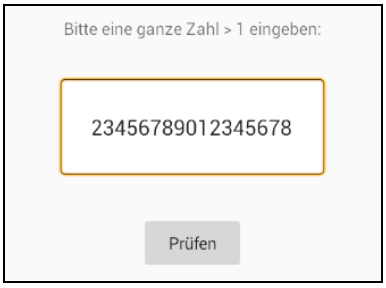
Über die **padding**-Attribute schafft man Abstand zwischen dem Inhalt eines **View**-Objekts und seinem Rand. Neben dem allseitig wirkenden Attribut **andoid:padding** gibt es die seitenspezifischen Attribute:

- **andoid:paddingLeft**
- **andoid:paddingRight**
- **andoid:paddingTop**
- **andoid: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:

layout_margin = "0dp" **padding** = "0dp"	**layout_margin** = "30dp" **padding** = "0dp"	**layout_margin** = "30dp" **padding** = "30dp"
		

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"**).

7.3 Container

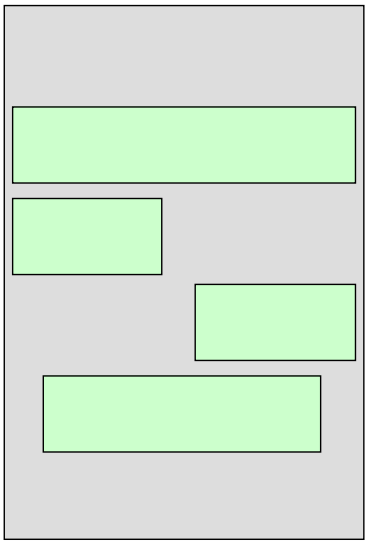
Die Spezialisierungen der Klasse **ViewGroup** können als Container andere **View**- oder **ViewGroup**-Objekte aufnehmen und auf der verfügbaren Fläche anordnen. Zwar gibt es unter Android mit seinem permanenten Fullscreen-Modus für die Container bzw. Layout-Manager weniger Stress als bei einer Java-Desktop-Anwendung, 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

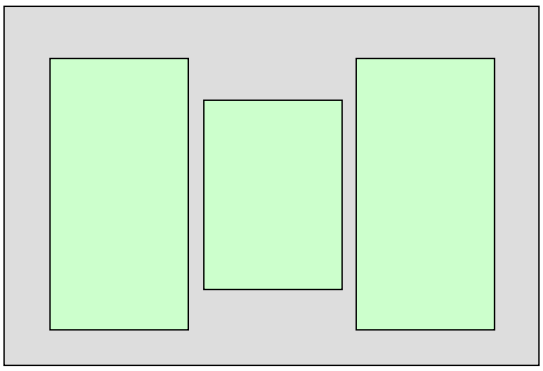
Anschließend werden einige häufig genutzte **ViewGroup**-Ableitungen vorgestellt. Bei einem Activity-GUI mit typischer Komplexität ist es oft erforderlich, Layout-Manager (aus verschiedenen Klassen) geschickt zu verschachteln. Allerdings sollte die Verschachtelung aus Performanzgründen nicht zu tief ausfallen.

7.3.1 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.

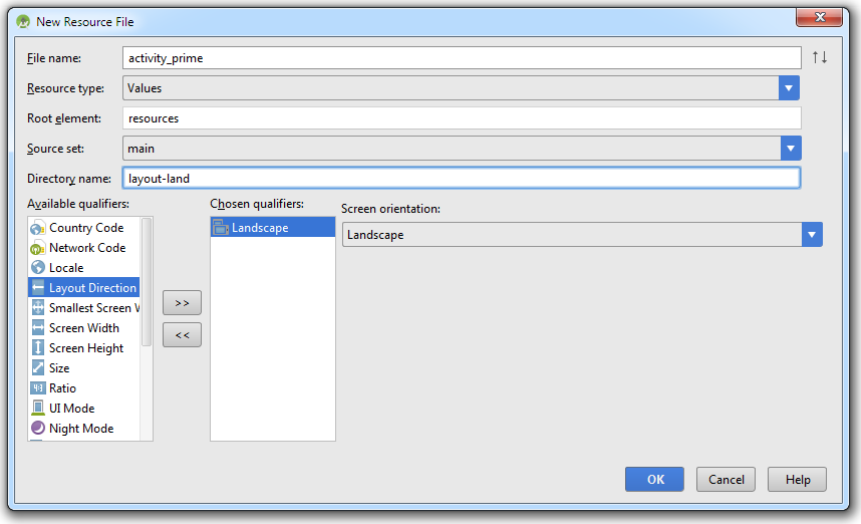
Damit unsere App `Prime Detection` bei einem Gerät mit Landschaftsorientierung so erscheint,



wählen wir aus dem Kontextmenü zum Knoten **res** im Project-Fenster das Item

New > Android resource file

und legen über das Fenster **New Resource File**



im Projektordner **layout-land** eine Datei namens **activity_prime.xml** mit zwei geschachtelten **LinearLayout**-Behältern an:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:gravity="center">

    <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="10dp" android:layout_marginRight="10dp"
            android:inputType="number"
            android:maxLength="18"
            android:minWidth="100dp"
            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="20dp" />
</LinearLayout>
```

Hier sorgt ein äußeres **LinearLayout**-Objekt (das Wurzelelement) 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**).
- Ränder
  Über das Attribut **android:layout_margin** für Kindelemente 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. 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:

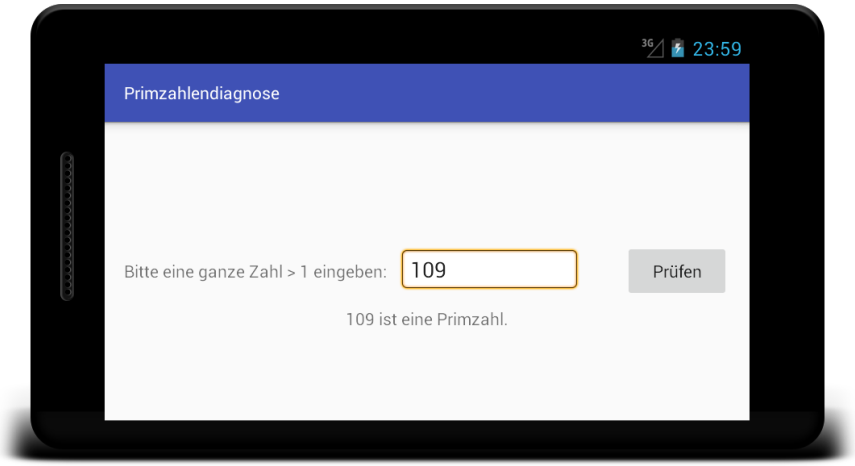


Es wird vielfach empfohlen und von Google auf den Android-Entwicklerseiten auch demonstriert, bei Verwendung von **android:layout_weight** das zugehörige Ausdehnungsattribut auf **0dp** zu setzen, bei einem vertikal orientieren **LinearLayout** also die Höhe:[1]

```
android:layout_height="0dp"
```

7.3.2 RelativeLayout

Verschachtelte **LinearLayout**-Container (wie im vorigen Beispiel) können die Layout-Performance beeinträchtigen und sollten durch eine flache Konstruktion mit *einem* **RelativeLayout**-Container ersetzt werden.[2] 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 einzig verbleibende Layout-Manager erhält den Typ **RelativeLayout**.
- Das **TextView**-Objekt 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"
  ```
- 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"
  ```

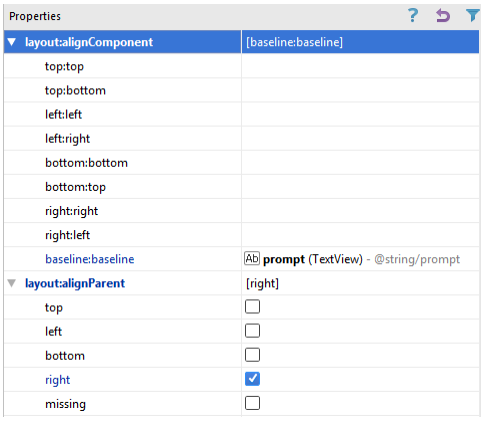
---

[1] https://developer.android.com/guide/topics/ui/layout/linear.html

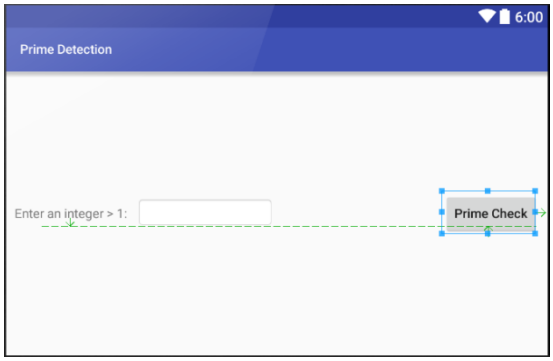
[2] Das empfiehlt Google auf der folgenden Webseite:
http://developer.android.com/training/improving-layouts/optimizing-layout.html

- Der Befehlsschalter wird am rechten Rand des Containers befestigt:
  `android:layout_alignParentRight="true"`
  In vertikaler Richtung orientiert der Befehlsschalter seine Position 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_below="@id/prompt"`

Das Android Studio erleichtert den Umgang mit dem **RelativeLayout** durch die Layout-Attribute im **Properties**-Fenster:



In der Vorschau zeigt das Studio für ein markiertes Element seine Orientierungsbeziehungen an, 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="10dp" android:layout_marginRight="10dp"
        android:inputType="number"
        android:maxLength="18"
        android:minWidth="150dp"
        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="20dp"
        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.

7.3.3 FrameLayout

Dieser simple Layout-Manager platziert die aufgenommenen Elemente nacheinander auf einen Stapel unter Berücksichtigung der folgenden Layout-Parameter:

- **android:layout_width**
- **android:layout_height**
- **android:layout_margin**
- **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

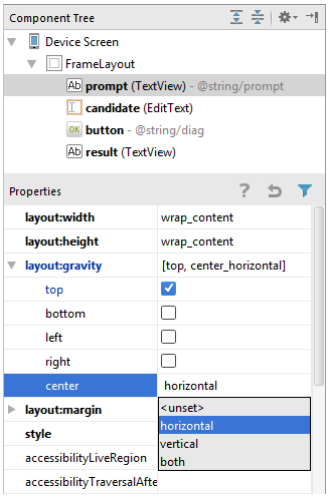
Einige Werte können sinnvoll durch den bitweisen Oder-Operator **|** verknüpft werden (z.B. **top | center_horizontal**). Dieser Operator verarbeitet die Bitmuster von zwei ganzen Zahlen zu einem

neuen Bitmuster, das an der Position *k* genau dann eine Eins enthält, wenn mindestens ein Argument an dieser Stelle eine Eins besitzt, z.B.:[1]

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 **Properties**-Fenster die Ausrichtung eines Steuerelements definieren, z.B.:



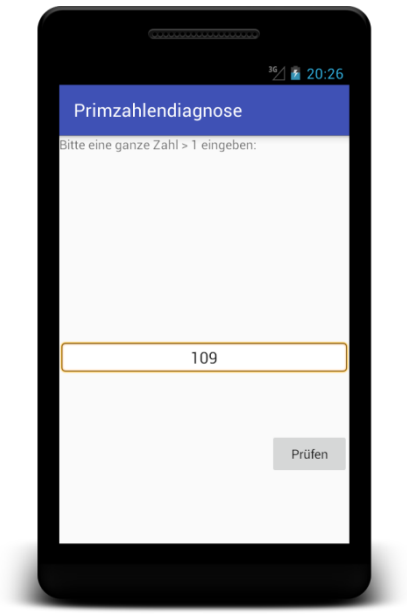
Daraus erstellt die Entwicklungsumgebung den XML-Code, z.B.:

```
android:layout_gravity="center_horizontal|top"
```

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 oben links, das **Button**-Objekt unten rechts mit 75dp Abstand zum unteren Rand, das **EditText**-Objekt (mit `android:layout_width="match_parent"`) in der Mitte sowie das **TextView**-Objekt mit der Ergebnisanzeige am unteren Rand bei horizontaler Zentrierung:

[1] Zur den Werten der Konstanten siehe:
http://developer.android.com/reference/android/widget/LinearLayout.LayoutParams.html#attr_android:layout_gravity



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 völlig ungeeignet. Google empfiehlt, ihn im Allgemeinen nur für ein *einzelnes* Kindelement zu verwenden, z.B. für eine Liste (siehe Abschnitt 7.5.7) oder für ein im Vollbildmodus ablaufendes Video. In dieser Situation ist der extrem niedrige Speicherbedarf des **FrameLayout**-Containers vor Vorteil. Gelegentlich macht sich das **FrameLayout** aber auch dabei nützlich, zwei **View**-Objekte (bei minimalem Programmieraufwand) übereinander zu positionieren, z.B. eine Beschriftung (also ein **TextView**-Objekt) über einem Bild (also einem **ImageView**-Objekt).

7.3.4 TableLayout

Das **TableLayout** positioniert seine Kindelemente auf 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 auch 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 von einzelnen 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**-Wurzelelement 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 7.3.2) 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="10dp" android:layout_marginRight="10dp"
            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="20dp"
        android:gravity="center"/>
</TableLayout>
```

Man hätte dasselbe Ergebnis auch durch das Schachteln von **LinearLayout**-Elementen erreichen können (vgl. Abschnitt 7.3.1). Vorteilhaft ist beim **TableLayout**, dass man für die Kindeelemente keine Breite (**android:layout_width**) und Höhe (**android:layout_height**) angeben muss.

7.3.5 GridLayout

Das mit Android 4 eingeführte **GridLayout** ist der jüngste Platzanweiser für Android-Apps und genügt am ehesten dem Anspruch, ein performanter und flexibler Vielzweck-Layouter zu sein.[1] Allerdings setzt seine erfolgreiche Verwendung eine Beschäftigung mit der Verarbeitung der Layout-Parameter voraus (Jackson 2014, S. 260).

Wie beim **TableLayout** (siehe Abschnitt 7.3.4) 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 ermöglicht, 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.:

[1] http://android-developers.blogspot.de/2011/11/new-layout-widgets-space-and-gridlayout.html

```
<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 zeilendominant, 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 = &quot;fill&quot;\nrowSpan=&quot;2&quot;"
    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:

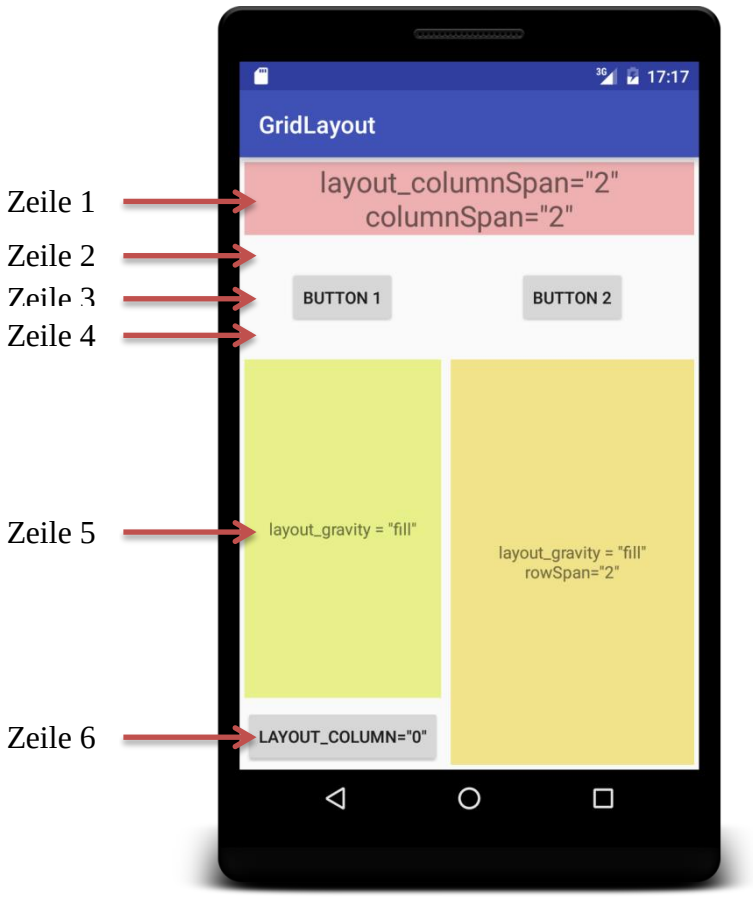
- Erhält das **GridLayout**-Element den Wert **true** für das XML-Attribut **android:useDefaultMargins**, dann orientieren sich die Abstände zwischen den Kindelementen am aktuelle Design-Theme der Anwendung, z.B.:

  ```
  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 Zeilenabstand, ohne in jeder Zelle der betroffenen Spalte bzw. Zeile wiederholt werden zu müssen, z.B.:

  ```
  <Space
      android:layout_height="20dp" />
  ```

Im folgenden Beispiel mit 6 Zeilen und 2 Spalten

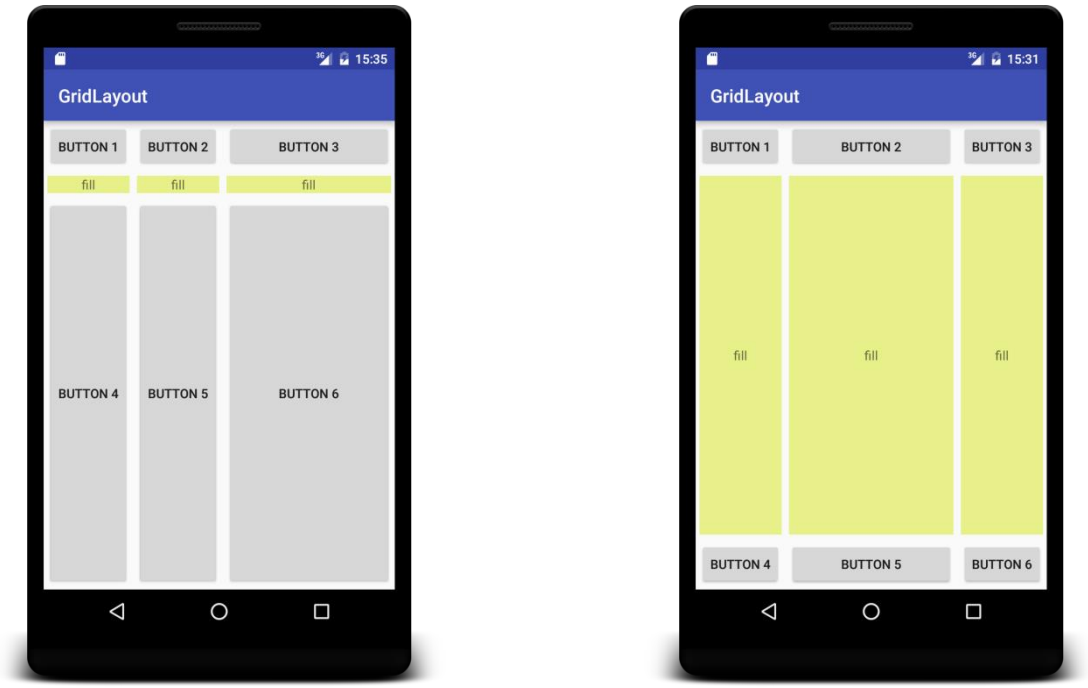


belegt ein **TextView**-Element die beiden Spalten der ersten Zeile. Das **TextView**-Element unten rechts belegt die Zeilen 5 und 6 der rechten Spalte. Vor und hinter der Zeile mit den Button-Elementen befindet sich eine Zeile mit je einem **Space**-Element in der linken Spalte.

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, so dass es analog zum **LinearLayout** möglich ist, andere bzw. mehrere Spalten bzw. Zeilen von Raumreserven profitieren zu lassen, z.B.:[1]

```
<TextView
    android:text="fill"
    android:background="#e8f189"
    android:gravity="center"
    android:layout_gravity="fill"
    android:layout_rowWeight="1" android:layout_columnWeight="1" />
```

Im Beispiel wird nun die *mittlere* Spalte bzw. Zeile gestreckt, wenn alle anderen Spalten bzw. Zeilen das voreingestellt Gewicht 0 behalten (siehe rechte Abbildung oben).

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.

---

[1] Durch die Verwendung des Elements **android.support.v7.widget.GridLayout** und begleitende Maßnahmen sollte das **GridLayout** mit Spalten- bzw. Zeilengewichtung auch in älteren Android-Versionen klappen.

7.4 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.[1]

7.4.1 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:[2]

```
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);
    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.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.:

- **setOnCreateContextMenuListener()**
- **setOnDragListener()**
- **setOnLongClickListener()**

Abgeleitete Klassen kennen weitere Ereignisse.

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.

---

[1] http://developer.android.com/guide/topics/ui/ui-events.html

[2] Das Quellcodesegment stammt aus der Klasse **View** im Paket **android.view** der Android-Version 6.0 (API-Level 23).

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**) beerben 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.

7.4.2 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 = (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**). ImAndroid-API beschränken sich die Event Listener - Interfaces meist auf eine Methode (z.B. **onClick()**, **onLongClick()**).

7.4.2.1 Ereignisbehandlung durch eine anonyme Klasse

Alternativ kann man der Registrierungsmethode als Aktualparameter ein Objekt einer anonymen, spontan definierten Klasse übergeben, z.B.:

```
CheckBox cbBold;

cbBold = (CheckBox) 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 7.5.4.1) ein **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 Klasse zur Schriftverwaltung verwendeten Methoden **getTypeface()**, **getStyle()** und **setTypeface()** und weitere Details werden in Abschnitt 7.5.4.1 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 2016, Abschnitt 12.1.1.2):

- Definition und Instantiierung finden in einem **new**-Operanden statt, wobei im Konstruktoraufruf der fehlende Klassenname 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 (einzige) Methode **onCheckedChanged()** implementiert.

  Es kann nur eine einzige Instanz erzeugt werden. Werden mehrere Instanzen benötigt, ist eine anonyme Klasse ungeeignet.
- Weil der Klassenname 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**.

Das Android Studio bemüht sich, die syntaktische Komplexität von anonymen Klassen zu verstecken, und verwendet eine von den Lambda-Ausdrücken in Java 8 inspirierte Darstellung (vgl. Baltes & Baltes-Götz 2016, Abschnitt 12.1):

```
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);
});
```

7.4.2.2 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 etwas reduzieren:

- Man implementiert in der Activity-Klasse eine Ereignisbehandlungsmethode mit …
  - dem Zugriffsmodifikator **public**,
  - dem Rückgabetyp **void**,
  - einem Parameter vom Typ **View**
  - 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="30dp"
    android:layout_marginTop="30dp"
    android:textAllCaps="false"
    android:onClick="checkPrime" />
```

7.4.3 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** - Ereignis, das von einem **View**-Objekt nach einem längeren Touch gefeuert wird, ist hingegen der Rückgabetyp **boolean** vorgeschrieben.

Sind bei einem **View**-Objekt eine **Click**- *und* eine **LongClick**-Behandlungsmethode registriert, läuft nach einer längeren Berührung durch den Benutzer folgende Ereignisverarbeitung ab:[1]

- 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.

7.5 Bedienelemente

In diesem Abschnitt werden einfache Bedienelemente behandelt, die wir teilweise schon kennen. Mit komplexeren Exemplaren (z.B. **ListView**) werden wir uns später beschäftigen.

7.5.1 Beschriftungen (TextView)

Von der Steuerelementklasse **TextView**, die zur Ausgabe von Beschriftungen dient, ist wenig zu berichten. Ein **TextView**-Objekt mit statischem Inhalt muss im Quellcode nicht angesprochen werden, so dass 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 5) nur dann klappt, wenn in der Layoutdefinition das Attribut **android:freezesText** den Wert **true** erhält,

---

[1] Quelle: http://www.techotopia.com/index.php/An_Overview_and_Example_of_Android_Event_Handling

```
<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 6.10) 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 7.5.2).

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** (per Voreinstellung gleich **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.MONSPACE**
  Gleichabständige Schrift

Den zweiten Parameter kann man über die **int**-wertigen **Typeface**-Konstanten aus der folgenden Tabelle versorgen, um einen Schriftstil zu wählen:

int-Wert	Stil	**Typeface**-Konstante
0	Standard	NORMAL
1	**Fett**	BOLD
2	*Kursiv*	ITALIC
3	***Fett + Kursiv***	BOLD_ITALIC

Über die Methode **getTypeface()** 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"
```

Mit dem XML-Attribut **android:textSize** oder der korrespondierenden Methode **setTextSize()** setzt man die Textgröße, wobei die Maßeinheit **sp** (vgl. Abschnitt 6.5) verwendet werden sollte, z.B.:

```
android:textSize="15sp"
```

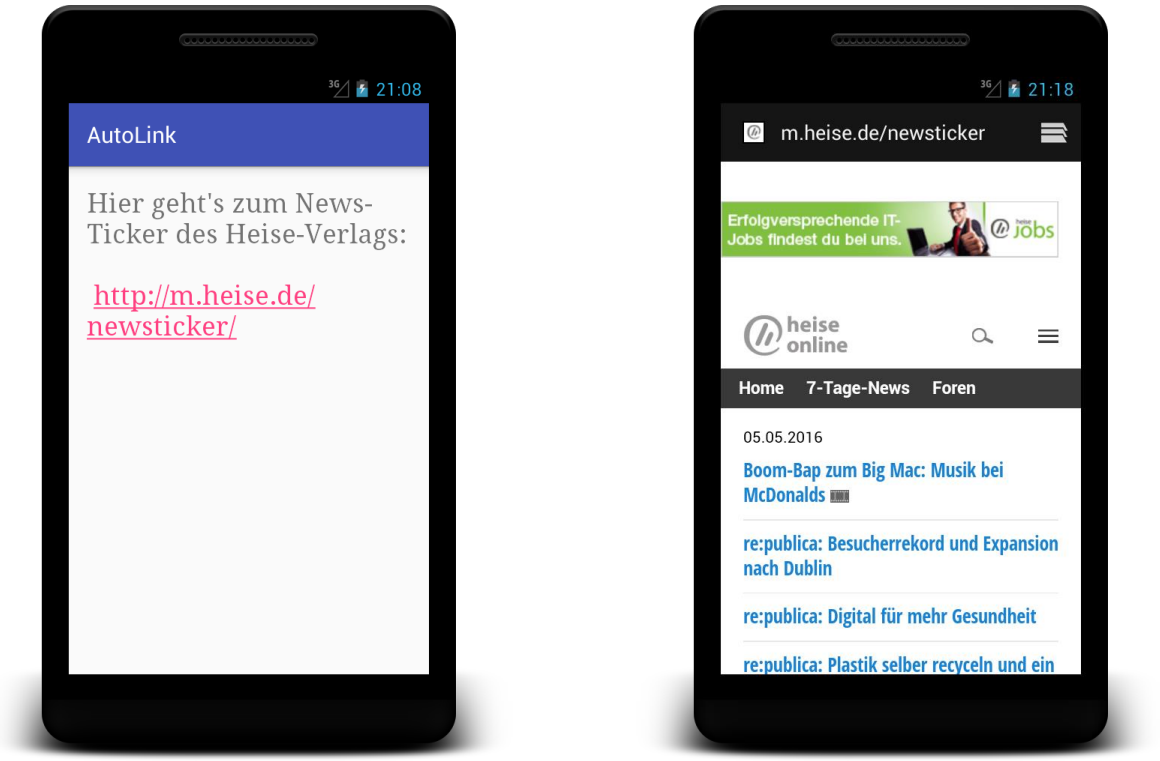
Über das XML-Attribut **android:autoLink** oder die korrespondierende Methode **setAutoLinkMask()** sorgt man dafür, dass im Text enthaltenen Internet-Adressen (z.B. Mail-Adressen oder URLs) erkannt und in klickbare Links umgesetzt werden. Aufgrund des **autoLink**-Attributs zum folgenden **TextView**-Steuerelement

```
<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
        http://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.

Viele XML-Attribute, die anschließend im Zusammenhang mit dem Texteingabefeld (Klasse **EditText**) beschrieben werden (z.B. **android:ems**, **android:lines**), sind schon in der **EditText**-Basisklasse **TextView** definiert, werden aber hier selten benötigt.

7.5.2 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>** - Ele-

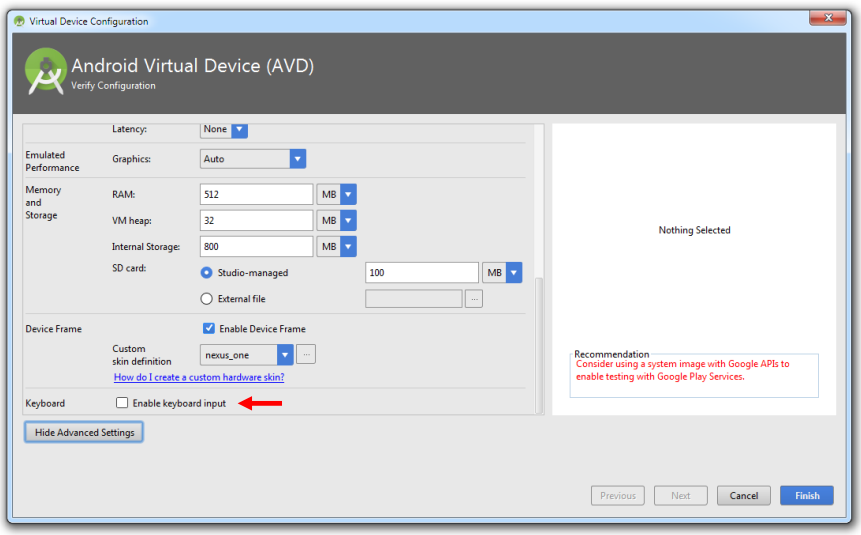
ment in der Layoutdefinition. Das Steuerelement besitzt etliche Kompetenzen als Texteditor inklusive der Zwischenablagen-Kooperation über ein Kontextmenü, das per Langklick angefordert wird.

Bei einem Android-Gerät mit Touchscreen hat ein Touch auf ein **EditText**-Objekt zur Folge, dass …

- eine virtuelle Tastatur erscheint,
- und das getroffene Element den Eingabefokus erhält.

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 Wirtsrechners 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:

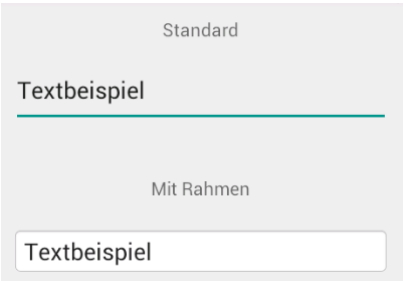


7.5.2.1 Optik

Über das Attribut **android:background** unter Verwendung einer vom Android-System spendierten Neun-Feld-Bitmap (vgl. Abschnitt 6.7.1)

```
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).



7.5.2.2 Irreguläre Eingaben verhindern

Über das XML-Attribut **android:inputType** kann man 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

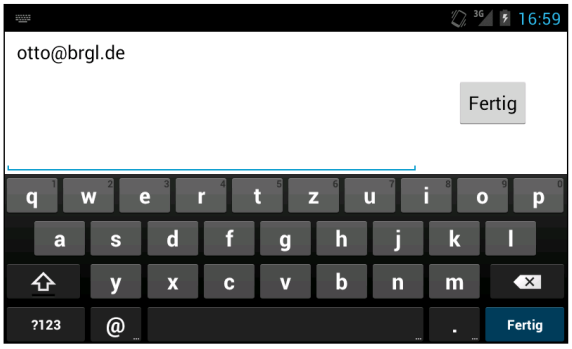


nur positive ganze Zahlen zulässt:

```
<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="20dp"
    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 **Properties**-Fenster verwenden, um die erlaubten Zeichen für ein **EditText**-Steuerelement festzulegen.
- **phone**
  Man erhält das Tastaturlayout einer Telefon-App.
- **textEmailAdress**
  Man erhält ein Tastaturlayout zur bequemen Eingabe einer Mail-Adresse:

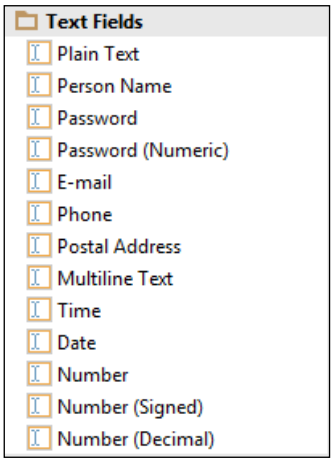


- **date**, **time**
  Von der Verwendung dieser Eingabetypen muss abgeraten werden, weil viele Hersteller von Android-Geräten die Tastatur-Anwendung durch eine Eigenkreation ersetzen und dabei die Unterstützung der Eingabetypen **date** und **time** vernachlässigen. Oft erscheint ein numerisches Tastatur-Layout 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 Fields** des grafischen GUI-Designers ein eigenes Item:



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.

7.5.2.3 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 7.5.2.2).

Um bei fehlendem **inputType**-Wert den Einzeilenmodus zu erzwingen, setzt man das Attribut **android:singleLine** auf den Wert **true**:

```
android:singleLine="true"
```

Um bei einem vorhandenen **inputType**-Wert eine mehrzeilige Texteingabe zu ermöglichen, schreibt man den Wert **textMultiLine** zusätzlich in das Attribut **inputType**, z.B.:

```
android:inputType="text|textMultiLine"
```

Im Mehrzeilenmodus besitzt die virtuelle Tastatur eine ↵ - Taste



Außerdem sind in dieser Situation die folgenden Attribute von Interesse:

- **android:lines**
  Legt fest, wie viele Zeilen sichtbar sein sollen, wobei die Anzahl der tatsächlich sichtbaren Zeilen durch den verfügbaren Platz begrenzt ist (speziell bei Anwesenheit der virtuellen Tastatur).
- **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_marginTop="20dp"
    android:ems="10"
    android:lines="8"
    android:scrollbars="vertical"
    android:background="@android:drawable/editbox_background" />
```

7.5.2.4 Aktion nach Fertigstellung der Eingabe

Zu einem Texteingabefeld mit **text** als Wert für das Attribut **android:inputType** erscheint in der unteren rechten Ecke der virtuellen Tastatur ein Schalter mit angepasster Beschriftung und entsprechendem Verhalten:

- Fertig oder ↵
  Folgt auf das Texteingabefeld im aktuellen Layout *kein* weiteres Texteingabefeld, dann erscheint ...
  - beim einzeiligen Textfeld die Taste Fertig, welche die Texteingabe beendet
  - beim mehrzeiligen Textfeld die Taste ↵, die ...
    - beim einfachen Klick eine neue Zeile einfügt
    - beim Langklick die Texteingabe beendet
- Weiter oder ↵
  Folgt auf das Texteingabefeld im aktuellen Layout noch ein weiteres Texteingabefeld, dann erscheint
  - beim einzeiligen Textfeld die Taste Weiter, welche die Texteingabe im aktuellen Feld beendet und beim nächsten Texteingabefeld fortsetzt
  - beim mehrzeiligen Textfeld die Taste ↵, die ...
    - beim einfachen Klick eine neue Zeile einfügt
    - beim Langklick die Texteingabe im aktuellen Feld beendet und im nächsten Texteingabefeld fortsetzt

  Beim einzeiligen Textfeld kann mit dem Wert **actionDone** für das Attribut **android:imeOptions**

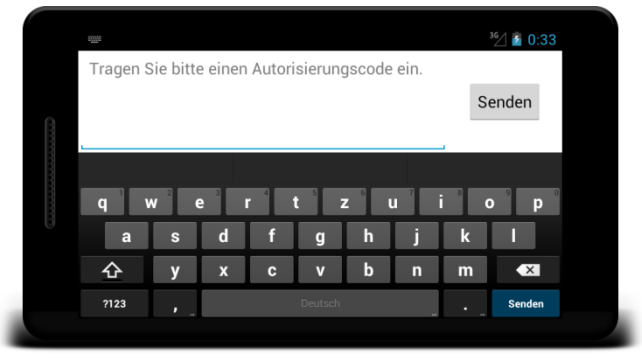
  ```
  android:imeOptions="actionDone"
  ```

  trotz folgendem Textfeld die Taste Fertig angefordert werden, die ...
  - beim einfachen Klick die Texteingabe beendet
  - beim Langklick die Texteingabe im aktuellen Feld beendet und im nächsten Texteingabefeld fortsetzt

Es sind noch weitere Aktionen als Werte für das Attribut **android:imeOptions** möglich (z.B. **actionSend**), wobei ein **OnEditorActionListener** beim **EditText**-Element registriert werden muss, der über das Ereignis informiert werden möchte und geeignet zu reagieren weiß.[1]

7.5.2.5 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:imeActionLabel** festgelegt werden kann.

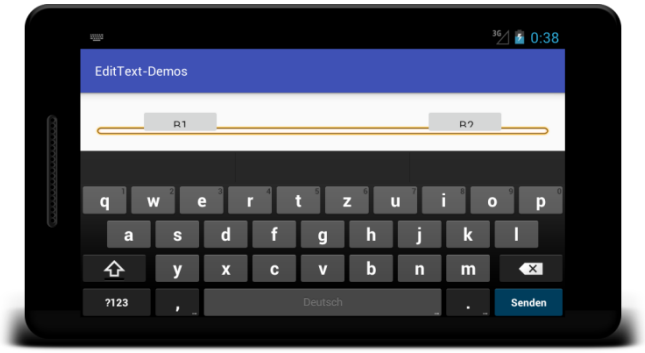
Mit dem Wert **flagNoExtractUi** für das Attribut **android:imeOptions** lässt sich der Fullscreen-Texteingabemodus verhindern:

---

[1] Siehe: http://developer.android.com/guide/topics/ui/controls/text.html

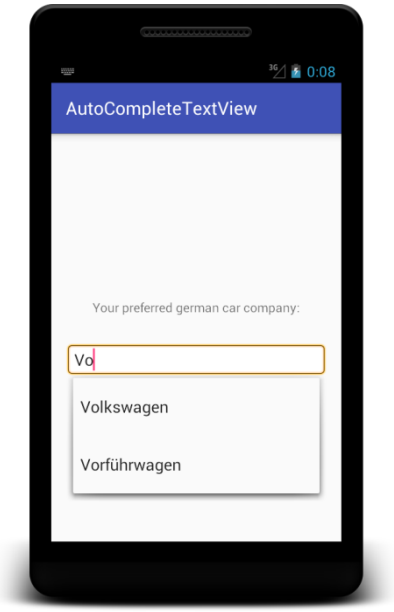
```
android:imeOptions="flagNoExtractUi"
```

Das ist aber oft nicht sinnvoll, z.B.:



7.5.2.6 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 Klick 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<String>(this,
              android.R.layout.simple_dropdown_item_1line, germanCars);

      AutoCompleteTextView autoComplete =
        (AutoCompleteTextView) findViewById(R.id.autoCompleteTextView);
      autoComplete.setAdapter(adapter);
  }
}
```

7.5.2.7 Weitere Attribute für EditText-Elemente

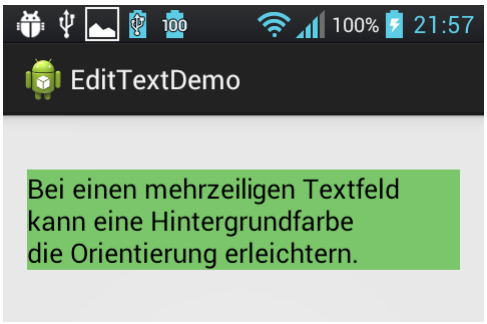
Mit dem Attribut **android:hint** vereinbart man einen Hinweistext, der in grauer Schriftfarbe im Texteingabefeld erscheint und beim Eintippen des ersten Zeichens verschwindet (siehe Beispiel in Abschnitt 7.5.2.5).

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 7.5.2.1 beschriebene Rahmung besonders nützlich, weil die verfügbare Eingabezone erkennbar wird. Alternativ kann eine Hintergrundfarbe den Eingabebereich anzeigen, z.B.:



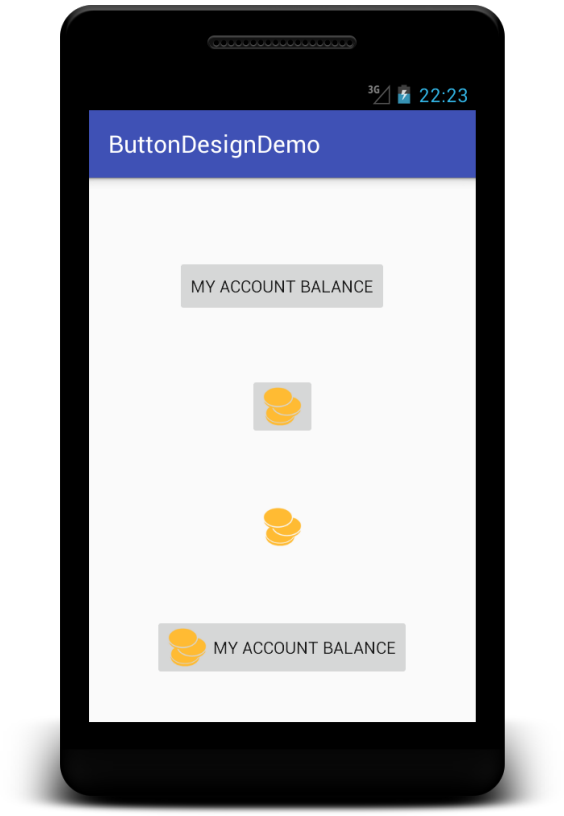
Man vereinbart die Hintergrundfarbe über das Attribut **android:background**, z.B.:

```
android:background="#7BC66B"
```

7.5.3 Aktionsschalter

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 kennen gelernt (siehe Abschnitt 7.4), so dass in diesem Abschnitt nur einige Gestaltungsoptionen erwähnt werden sollen.

Im folgenden Layout



sind neben einem Standard-**Button** mit Beschriftung noch drei 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:layout_alignParentTop="true"
    android:layout_centerHorizontal="true"
    android:layout_marginTop="50dp"
    android:text="@string/account" />
```

  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 6.4).

- **ImageButton** (2. Schalter im Beispiel)
  Ein Schalter aus der Klasse **ImageButton** zeigt statt einer Beschriftung ein Bild, das in der XML-Deklaration über das Attribut **android:src** vereinbart wird:

```
<ImageButton
    android:id="@+id/button2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerHorizontal="true"
    android:layout_below="@+id/button1"
    android:layout_marginTop="50dp"
    android:src="@drawable/ic_coins_l"
    android:contentDescription="@string/account" />
```

  Wie man eine Bitmap-Datei in ein AS-Projekt aufnimmt, wird gleich beschrieben. Mit dem Attribut **android:contentDescription** wird ein erläuternder Text zur Unterstützung sehbehinderter Menschen vereinbart.

- 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 6.3.1 zu diesem Ressourcen-Zugriff):

```
<ImageButton
    android:id="@+id/button3"
    style="?android:borderlessButtonStyle"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerHorizontal="true"
    android:layout_below="@+id/button2"
    android:layout_marginTop="50dp"
    android:src="@drawable/ic_coins_1"
    android:contentDescription="@string/account" />
```

- 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:

```
<Button
    android:id="@+id/button4"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerHorizontal="true"
    android:layout_below="@+id/button3"
    android:layout_marginTop="50dp"
    android:drawablePadding="5dp"
    android:drawableLeft="@drawable/ic_coins_1"
    android:text="@string/account" />
```

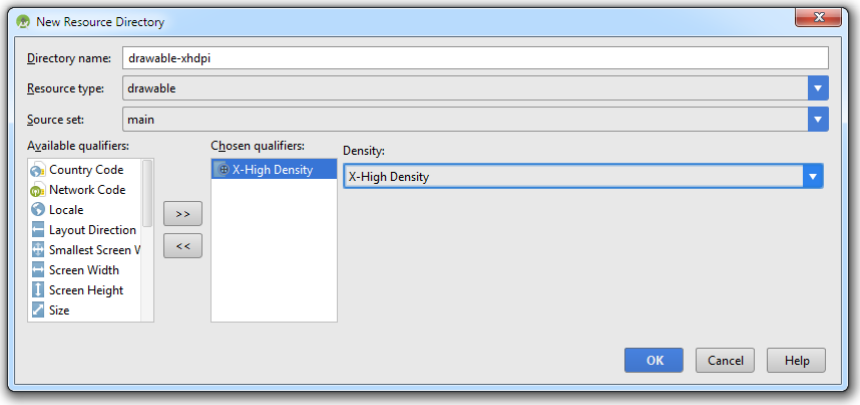
  Für einen passenden Abstand zwischen Text und Bild sorgt das Attribut **android:drawablePadding**.

Um eine PNG-Datei in ein AS-Projekt aufzunehmen, kann man so vorgehen:

- In der **Project**-Sicht des Projektordners das Verzeichnis

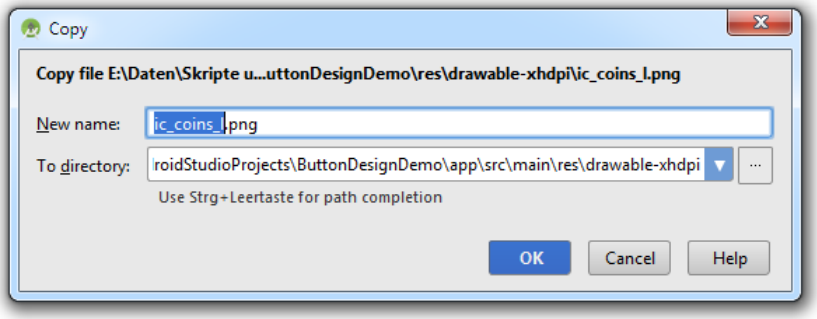
  app/src/main/res

  öffnen
- Falls zur Auflösung der Importdatei noch kein **drawable**-Ordner existiert (z.B. **drawable-xhdpi**), wählt man aus dem Kontextmenü zu **app/src/main/res** das Item **New > Android resource directory**, so dass der folgende Dialog erscheint:

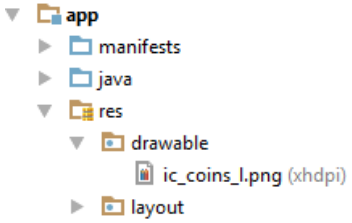


  Hier wählt man **drawable** als **Resource type** und **Density** in der Liste **Available qualifiers**. Nach Wahl der gewünschten Auflösung (**Density**), kann mit **OK** die Kreation des neuen Ressourcen-Ordners angefordert werden.

- Per Windows-Explorer befördert man die zu importierende PNG-Datei in die Zwischenablage.
- Aus dem Kontextmenü zum auflösungsspezifischen **drawable**-Ordner wählt man das Item **Paste**, um die Importdatei aus der Zwischenablage zu entnehmen.
- Nach **OK** muss noch im folgenden Dialog das Kopieren bestätigt werden:



- Schließlich erscheint die aufgenommene Datei im Projektordner und ist auch in der meist bequemeren **Android**-Sicht zu sehen, z.B.:



- Niedrigere Auflösungen sollten von Android durch automatische Neuberechnungen versorgt werden.

Eine Grundausstattung an Symbolen enthält das Android-SDK in versionsspezifischen Unterordnern, z.B.:

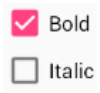
…\sdk\platforms\android-23\data\res\drawable-xxhdpi

Auf den Google-Webseiten für Android-Entwickler wird beschrieben, wie sich der Hintergrund eines Schalters über eine Ressource vom Typ *Color State List* (vgl. Abschnitt 6.1) gestalten lässt.[1]

7.5.4 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 7.2):

- 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.:

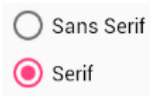


[1] Siehe: http://developer.android.com/guide/topics/ui/controls/button.html

- Switch-Schalter (Klasse **Switch**)
  Es ist ein Schieberegler mit den beiden Zuständen *Aus* (Position links) und *Ein* (Position rechts) vorhanden, z.B.:

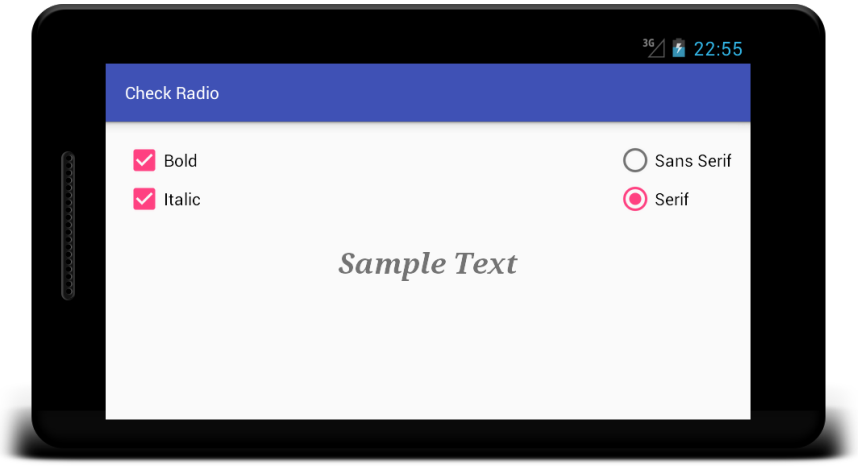


- Soll der Benutzer von mehreren Optionen *genau eine* wählen, verwendet man Steuerelemente aus der Klasse **RadioButton** in einem Container aus der Klasse **RadioGroup**, z.B.:



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 zwei Kontrollkästchen die Schriftauszeichnung und über ein Optionsfeld zwischen den beiden Schriftarten **Sans Serif** und **Serif** gewählt werden:



Das Layout verwendet ein Wurzelelement aus der Klasse **RelativeLayout** mit folgenden Kindelementen:

- Das Kontrollkästchen für die Fett-Auszeichnung ist voreinstellungsgemäß links und oben am übergeordneten Container angeheftet.
- Das Kontrollkästchen für die Kursiv-Auszeichnung ist voreinstellungsgemäß am linken Rand des Containers angeheftet und außerdem unter dem Fett-Kontrollkästchen positioniert.
- Die beiden Optionsschalter stecken in einer **RadioGroup**, die vom **LinearLayout** abstammt und zusätzliche Leistungen zur Verwaltung ihrer Kindelemente erbringt (siehe Abschnitt 7.5.4.3). Im elterlichen Container ist die **RadioGroup** rechts (explizit) und oben (implizit) angeheftet.
- Das **TextView**-Element ist in Bezug auf den **RelativeLayout** - Container horizontal zentriert und außerdem (mit einem Abstand) unter der **RadioGroup** positioniert.

Hier ist die vollständige Layoutdefinition zu sehen:

```
<RelativeLayout 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:padding="@dimen/activity_vertical_margin" >
```

```
    <CheckBox
        android:id="@+id/cbBold"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/bold" />

    <CheckBox
        android:id="@+id/cbItalic"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/cbBold"
        android:text="@string/italic" />

    <RadioGroup
        android:id="@+id/radioGroup"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentRight="true" >

        <RadioButton
            android:id="@+id/rbSans"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:checked="true"
            android:text="@string/sans" />

        <RadioButton
            android:id="@+id/rbSerif"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/serif" />
    </RadioGroup>

    <TextView
        android:id="@+id/tvSample"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/radioGroup"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="20dp"
        android:text="@string/sample"
        android:textSize="24sp" />
</RelativeLayout>
```

Das AS-Projekt ist im folgenden Ordner zu finden:

…\BspUeb\GUI\CheckRadio

7.5.4.1 Kontrollkästchen

In der Aktivität des Beispielprogramms werden Instanzvariablen für die **CheckBox** - Objekte deklariert und in der Methode **onCreate()** initialisiert:

```
CheckBox cbBold, cbItalic;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_check_radio);
    cbBold = (CheckBox) findViewById(R.id.cbBold);
    cbItalic = (CheckBox) 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 7.4.2.1. 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 7.5.1). 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 7.5.4.3) ermittelt.[1]

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 ermitteln, wird im folgenden Ausdruck der Bit-orientierte Operator **&** verwendet (siehe Baltes-Götz & Götz 2016, 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 anderenfalls eine 0. Insgesamt ist der obige Ausdruck genau dann **true**, wenn die **fett**-Auszeichnung eingeschaltet ist. Die restliche Logik des Konditionaloperators dürfte klar sein.

7.5.4.2 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 Aktivität sind im Wesentlichen die Klassenzugehörigkeiten zu ändern:

---

[1] 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.

```
ToggleButton toggleBold;
Switch switchItalic;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_check_radio);

    toggleBold = (ToggleButton) findViewById(R.id.toggleBold);
    switchItalic = (Switch) 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 Beschriftungen für den passiven und den aktiven Zustand:

```
<ToggleButton
    android:id="@+id/toggleBold"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textOff="@string/textBoldOff"
    android:textOn="@string/textBoldOn"/>
```

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:



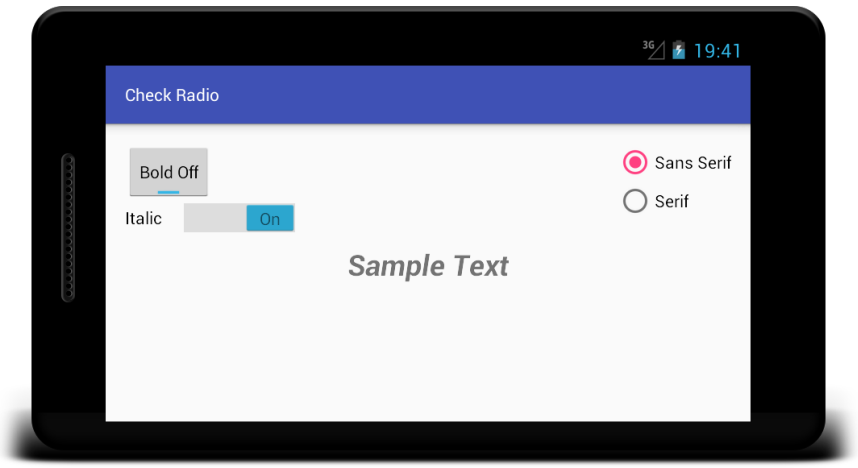
Es wird ergänzt durch Beschriftungen für den passiven und den aktiven Zustand:

```
<Switch
    android:id="@+id/switchItalic"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/textItalic"
    android:textOff="@string/textItalicOff"
    android:textOn="@string/textItalicOn" />
```

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:



7.5.4.3 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:id="@+id/radioGroup"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentRight="true" >

    <RadioButton
        android:id="@+id/rbSans"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:checked="true"
        android:text="@string/sans" />

    <RadioButton
        android:id="@+id/rbSerif"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/serif" />
</RadioGroup>
```

Für die beiden Optionsschalter des Umschalter-Demonstrationsprogramms

```
RadioButton rbSans, rbSerif;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_check_radio);
    rbSans = (RadioButton) findViewById(R.id.rbSans);
    rbSerif = (RadioButton) findViewById(R.id.rbSerif);
    rbSans.setOnCheckedChangeListener(this);
    rbSerif.setOnCheckedChangeListener(this);
      . . .
}
```

übernimmt das Aktivitätsobjekt die Ereignisbehandlung, wozu die Klasse mit der Methode **onCheckedChanged()** ausgerüstet wird:

```
@Override
public void onCheckedChanged(CompoundButton rb, boolean isChecked) {
    if (!rb.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 7.5.4.1 vorgestellten Schriftverwaltungsmethoden **getTypeface()**, **getStyle()** und **setTypeface()** zum Einsatz.[1]

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

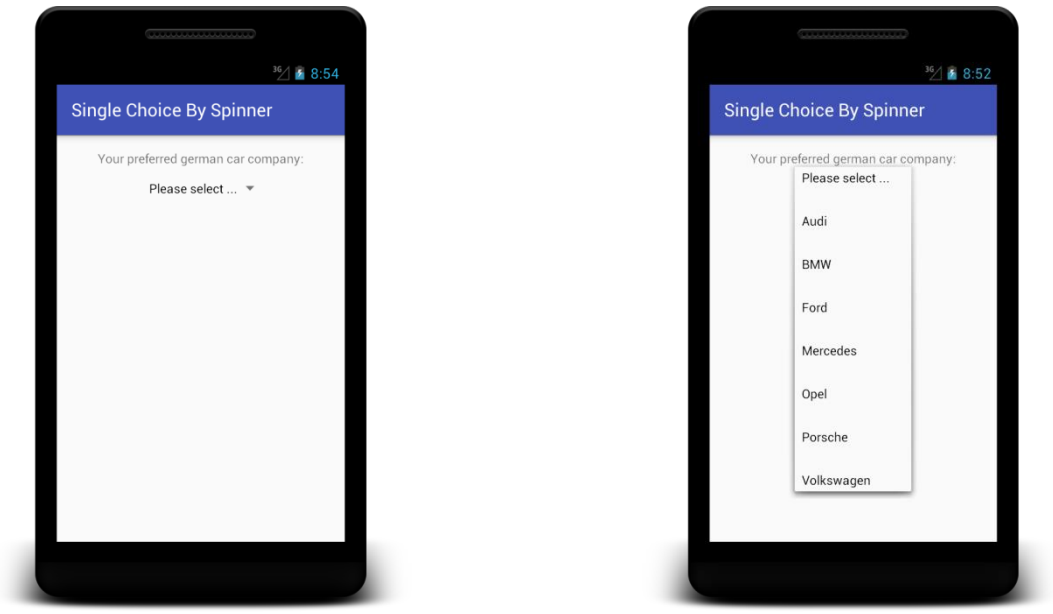
Weil die Methode nur im ersten Fall arbeiten soll, wird zu Beginn mit **isChecked()** der Wechseltyp festgestellt und die Ereignisbehandlung ggf. abgebrochen.

7.5.5 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, so dass wenig Validierungsaufwand entsteht.

Anschließend ist ein **Spinner**-Element im geschlossenen (links) und im geöffneten Zustand (rechts) zu sehen:



[1] Die Schriftart **Typeface.MONOSPACE** einzubeziehen, scheiterte daran, dass diese Schriftart auf Befragen mit der Methode **getStyle()** keine korrekte Stilinformation geliefert hat (zum Übersetzen verwendete SDK-Version: 23).

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.

Zur Realisation eines **Spinner**-Elements kann man weitgehend analog zum **AutoCompleteTextView**-Steuerelement (siehe Abschnitt 7.5.2.6) vorgehen und einen **ArrayAdapter<String>** per Java-Quellcode explizit zuweisen (siehe Lösungsvorschlag zur Übungsaufgabe 1 in Abschnitt 7.7). Bei der von Google empfohlenen Strategie erfährt das XML-Element **Spinner** von den Wahlalternativen über das Attribut **android:entries**, das auf eine Ressource von Typ **String**-Array zeigt:[1]

```
<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 6.1), die zusammen mit anderen einfachen Ressourcen in einer XML-Datei im Zweig **res/values** innerhalb eines gemeinsamen **<resources>** - Wurzelelements definiert wird, z.B. in der Datei **strings.xml**:

```
<resources>
    <string name="app_name">Single Choice By Spinner</string>
    <string name="prompt">Your preferred german car company:</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>
```

Um die Benutzerwahl verarbeiten zu können, verbindet man das **Spinner**-Element im Aktivitätsquellcode mit einer Referenzvariablen und vereinbart über die Methode **setOnItemSelectedListener()** ein Objekt vom Interfacetyp **AdapterView.OnItemSelectedListener** zur Verarbeitung der Wahl:

```
public class MainActivity extends AppCompatActivity
                          implements  AdapterView.OnItemSelectedListener {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Spinner spinner = (Spinner) findViewById(R.id.spinner);
        spinner.setOnItemSelectedListener(this);
    }
    . . .
}
```

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:

---

[1] Siehe http://developer.android.com/guide/topics/ui/controls/spinner.html

```
@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("Your Selection:\n" + text + " (Pos. " +
                           Integer.toString(position) + ")");
        AlertDialog dialog = builder.create();
        dialog.show();
    }
}
```

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:
  ```
  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\GUI\Spinner

7.5.6 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 übliche Weise:

```
private ProgressBar progressBar;
  .  .  .
protected void onCreate(Bundle savedInstanceState) {
     .  .  .
    progressBar = (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:[1]

```
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 der im Beispiel verwendete Wert **match_parent**.

Bei Verwendung des 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.:

```
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"
```

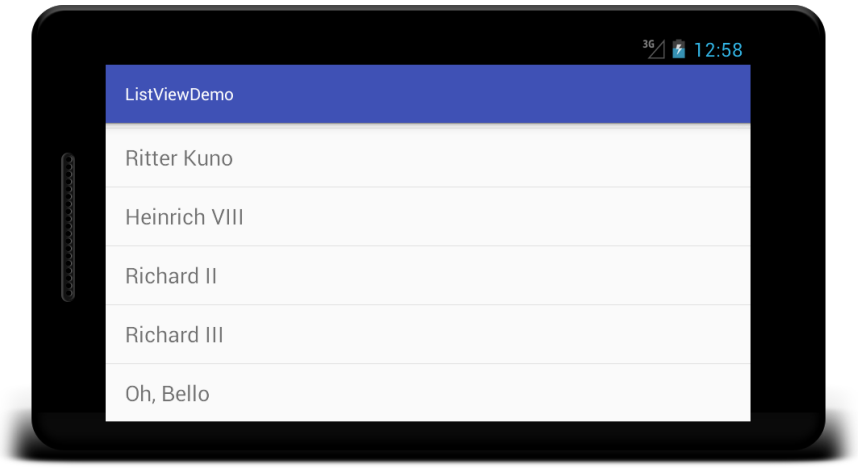
Im Abschnitt 9.3 folgt ein ausführliches Beispiel zur Verwendung eines Fortschrittsbalkens.

7.5.7 ListView und ListActivity

Eine sehr verbreitete Aufgabe für eine Android-Activity besteht darin, eine zu durchstöbernde vertikale Liste von Items (z.B. Kontakte, Apps, Musiktitel) 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 konzipiert ist und dabei zusätzliche Kompetenzen besitzt. Die Listenelemente werden von einem Objekt der Klasse **Adapter** zugeliefert, das sie z.B. aus einer Datenbank (vgl. Kapitel 12) oder aus einem Array beschafft. Für das folgende Beispiel

[1] Siehe Abschnitt 6.3.1 zu den Punkten im Ressourcennamen



sorgt eine simple Layoutdefinition

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <ListView
        android:id="@+id/list"
        android:layout_height="wrap_content"
        android:layout_width="match_parent">
    </ListView>
</LinearLayout>
```

und der folgende Aktivitäts-Quellcode:

```
package de.zimkand.listviewdemo;
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 = (ListView) findViewById(R.id.list);

        final String[] TITLES = {"Heinrich IV", "Alfons, der Viertel vor Zwölfte",
         "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 = (ListView) findViewById(R.id.list);
```

Der Adapter vom Typ **ArrayAdapter<String>** entsteht aus einem **String**-Array und einer Layoutdefinition in der Klasse **R.layout** im Pakt **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 8 beschäftigen werden.

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);

        final String[] TITLES = {"Heinrich IV", "Alfons, der Viertel vor Zwölfte",
         "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 reguläre Aktivitätsbasisklasse beerben und das simple Layout mit **ListView**-Element konstruieren.

7.6 Menüs

Seit der Version 3 bietet Android drei Typen von Menüs:

- Optionsmenü und App Bar zu einer Aktivität
  In das Optionsmenü gehören Aktionen, welche die 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 von einer zunächst als *Action Bar* und später als *App Bar* bezeichneten Leiste übernommen. Dabei blieb die Grundlogik der Menü-APIs unverändert.
- Kontextmenü
  Das Kontextmenü erscheint nach einem Langklick auf ein Element und bietet Aktionen mit Relevanz für dieses Element (z.B. Löschen, Speichern).
- Popup-Menü
  Das Popup-Menü ähnelt stark dem traditionellen, flottierenden Kontextmenü. Eventuell wurde das Popup-Menü mit Android 3 eingeführt als Kompensation zur gleichzeitig erlassenen Design-Direktive, das flottierende Kontextmenü durch die sogenannten *kontextabhängigen Aktionen* zu ersetzen (siehe Abschnitt 7.6.3.2). Kontextabhängige Aktionen erscheinen nämlich nicht *neben* einem betroffenen Element (wie das flottierende Kontextmenü), sondern am oberen Display-Rand und überlagern dort die App Bar.

Mit Popup-Menüs werden wir uns aus Zeitgründen nicht beschäftigen. Dieser Menü-Typ wird wie seine beiden Alternativen in der Online-Dokumentation für Android-Entwickler beschrieben.[1]

7.6.1 Menüdeklaration per XML-Datei

Analog zum Vorgehen bei einer **View**-Hierarchie sollte man ein Menü nicht per Quellcode aufbauen, sondern (im Projektordner **.../res/menu**) eine XML-formatierte Menü-Ressource erstellen, die dann im Programm zu einem Objekt der Klasse **Menu** inflationiert wird.

7.6.1.1 Elemente einer Menüdeklaration

Aus dem **menu**-Wurzelelement der XML-Datei entsteht im laufenden Programm ein Container aus der Klasse **Menu**, der Elemente aus der Klasse **MenuItem** aufnimmt. Das Wurzelelement einer XML-Menüdefinition kann als Kindelemente enthalten:

- **item**-Elemente, die jeweils ein Menüitem repräsentieren
- **group**-Elemente, welche Menüitems zu Gruppen mit gemeinsamen Eigenschaften 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 der Menüitems bestimmen.

---

[1] Siehe https://developer.android.com/guide/topics/ui/menus.html

- **android:showAsAction**
  Soll das Item am rechten Rand in der App Bar erscheinen? Mögliche Werte:
  - **ifRoom**
    Das Item erscheint in der App Bar, wenn dort genügend Platz vorhanden ist.
  - **never** (= Voreinstellung)
    Das Item erscheint nach einem Klick auf das Überlaufsymbol  am rechten Rand der App Bar.

  Später demonstrieren zwei Varianten eines Beispielprogramms, wie sich die beiden Attributausprägungen auswirken.

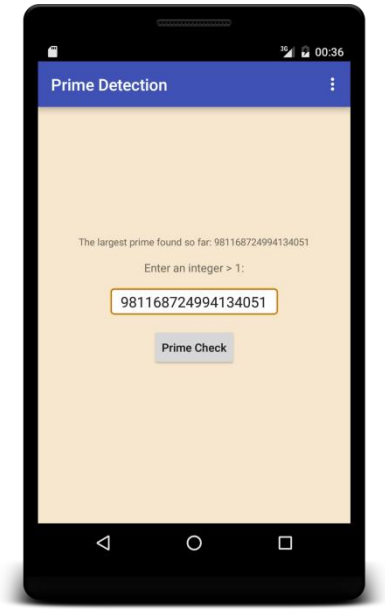
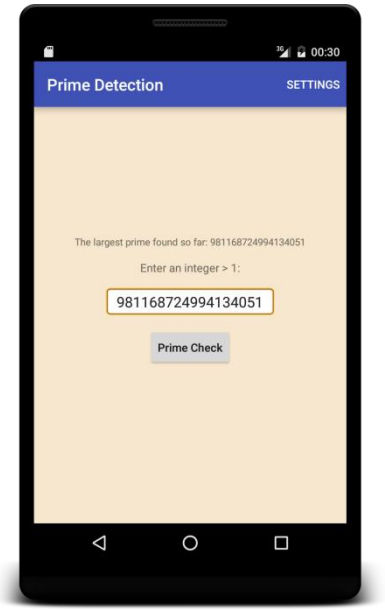
7.6.1.2 Beispiel

Lässt man das Android Studio in der Version 2.x ein neues Projekt mit einer **Basic Activity** anlegen, resultiert im Projektordner **...\res\menu** per Voreinstellung die folgende XML-Datei **menu_main.xml** mit einem minimalen Optionsmenü. Es ist nur ein einziges Item vorhanden, das offenbar die Programmeinstellungen zugänglich machen soll:

```
<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>
```

Das einzige Menüitem erhält eine ID und einen Titel. Mit dem Attribut **android:orderInCategory** wird die Auftrittsreihenfolge der Menüitems festgelegt, was bei einem einzigen Exemplar natürlich irrelevant ist. Über das Attribut **app:showAsAction** wird darüber entschieden, ob das Menüitem in der App Bar erscheinen soll, wenn dort genügend Platz vorhanden ist. Das folgende Beispielprogramm (vgl. Kapitel 14) benötigt tatsächlich in seinem Optionsmenü nur ein Item zum Starten der Einstellungsverwaltung. Es demonstriert den Auftritt des Optionsmenüs in Abhängigkeit vom Attribut **showAsAction**:[1]

[1] Weil wir auf Empfehlung des AS-Assistenten für neue Projekte unsere Aktivitätsklassen von **AppCompatActivity** ableiten, ist das Attribut **showAsAction** aus den **app** - Namensraum zu verwenden. Anderenfalls wäre das Attribut **showAsAction** aus den **android** - Namensraum die richtige Wahl.

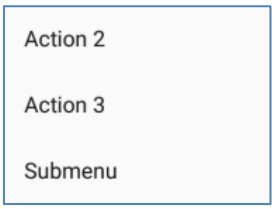
app:showAsAction="never"	app:showAsAction="ifRoom"
	

7.6.1.3 Untermenüs

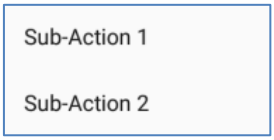
Um ein **Untermenü** zu realisieren, wird in ein **item**-Element ein **menu**-Element eingefügt, das wiederum beliebig viele **item**-Elemente aufnehmen kann, z.B. im folgenden Optionsmenü:

```
<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/action1"
        android:icon="@drawable/android"
        android:title="Action 1"
        app:showAsAction="ifRoom" />
    <item
        android:id="@+id/action2"
        android:title="Action 2" />
    <item
        android:id="@+id/submenu1"
        android:title="Submenu">
        <menu>
            <item
                android:id="@+id/subaction1"
                android:title="Sub-Action 1" />
            <item
                android:id="@+id/subaction2"
                android:title="Sub-Action 2" />
        </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:

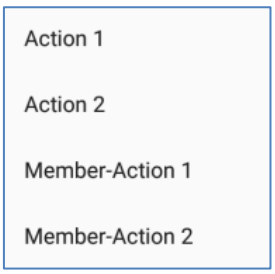


7.6.1.4 Items gruppieren

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/subaction1"
                android:title="Member-Action 1" />
            <item
                android:id="@+id/subaction2"
                android:title="Member-Action 2" />
    </group>
</menu>
```

Von dieser Gruppierung ist im entfalteten Menü nichts zu erkennen:



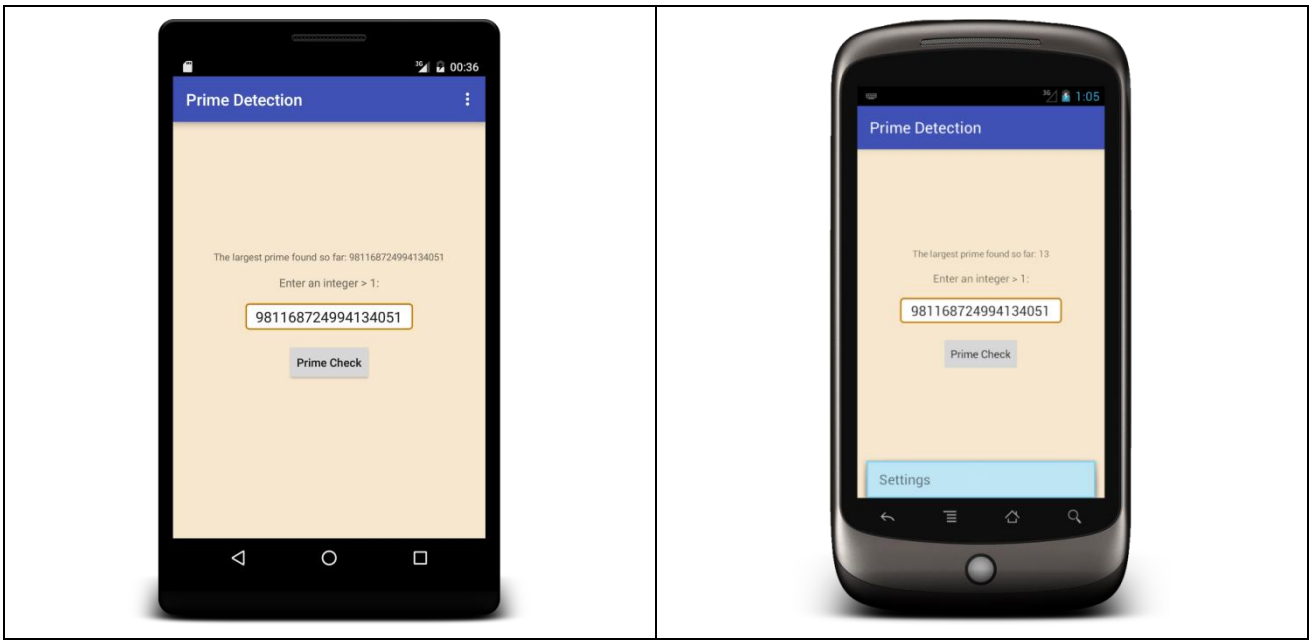
Allerdings können die Items einer Gruppe einer gemeinsamen Behandlung unterzogen werden, z.B. mit den folgenden **Menu**-Methoden:

- **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.

7.6.2 Optionsmenü

7.6.2.1 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.:



7.6.2.2 Zusammenstellen

Zum Füllen des Optionsmenüs sind die Aktivität und alle darin geladenen Fragmente (siehe Kapitel 11) 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 bevölkert, z.B.:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.menu_main, menu);
    return true;
}
```

Seit der Android-Version 3 wird die Methode **onCreateOptionsMenu()** beim Starten der Aktivität aufgerufen, um die App Bar passend ausstatten zu können.

Für Menüanpassungen per Programm stehen u.a. die folgenden **Menu**-Methoden zur Verfügung:

- **public void removeItem(int** *id***)**
  Ein Item wird entfernt.
- **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.

Ein Einsatz dieser Methoden kommt z.B. in Frage, wenn mehrere Aktivitäten viele Optionsmenüitems gemeinsam haben und daher von einer gemeinsamen Basisklasse abgeleitet werden, um

Code-Wiederholungen zu vermeiden. Erforderliche Menüanpassungen in einer speziellen Aktivität lassen sich dann folgendermaßen erledigen:

- die geerbte Methode **onCreateOptionsMenu()** überschreiben
- zu Beginn die Basisklassenvariante aufrufen
- obige Methoden zur Anpassung des Menüs verwenden

Um das Optionsmenü zur Laufzeit ereignisorientiert zu ändern, stehen seit Android 3 folgende Methoden bereit:

- **invalidateOptionsMenu()**
  Diese Aktivitätsmethode erklärt das Optionsmenü für ungültig.
- **onPrepareOptionsMenu(Menu** *menu***)**
  Nachdem das Optionsmenü als ungültig erklärt worden ist, wird die Methode **onPrepareOptionsMenu(Menu** *menu***)** für die Aktivität selbst sowie für jedes Fragment aufgerufen. Per Parameter wird das zu modifizierende Menü übergeben.

7.6.2.3 Menüereignisbehandlung

Wenn der Benutzer ein Optionsmenüitem wählt (eventuell per App Bar), 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 Optionenmenü:

```
<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:icon="@drawable/android"
        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 Menüereignisbehandlungsmethode

```
@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:



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. Anderenfalls sollte sie die überschriebene Basisklassenvariante aufrufen und deren Rückgabe durchreichen (siehe Beispiel).

Sind Fragmente vorhanden (siehe Kapitel 11), 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/android"
    android:title="Action 1"
    android:onClick="onAction1Click"
    app:showAsAction="ifRoom" />
```

Es kommt eine Methode der Aktivität mit folgenden Eigenschaften in Frage:

- Zugriff **public**
- Ein Parameter vom Typ **MenuItem**, keine weiteren Parameter

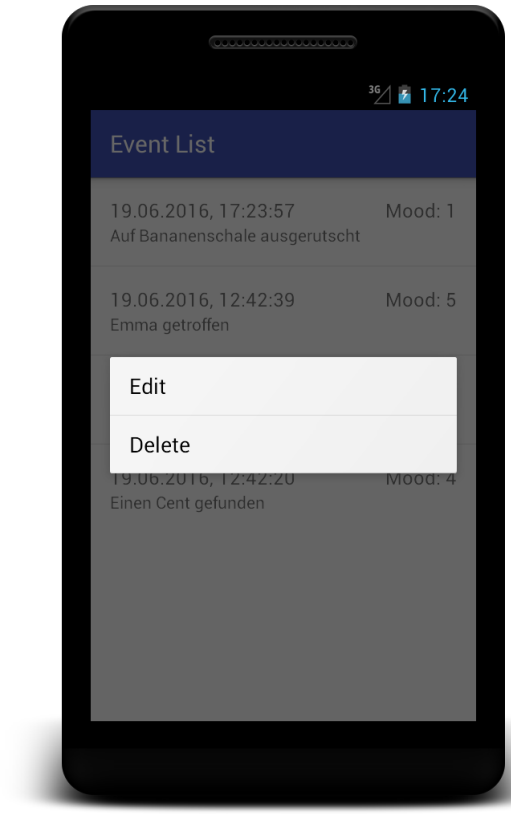
Ist eine Aktivität über die Methode **onOptionsItemSelected()** *und* über das Attribut **android:onClick** auf die Behandlung einer Menüentscheidung vorbereitet, dann gewinnt die Deklaration in der XML-Datei.

7.6.3 Kontextmenü

Ein Kontextmenü kann zu jedem **View**-Objekt eingerichtet werden, doch sind Container wie **ListView** (vgl. Abschnitt 7.5.7) besonders relevante Einsatzorte.

7.6.3.1 Flottierendes Kontextmenü

Um in einer Aktivität oder in einem Fragment für ein Steuerelement ein flottierendes 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, z.B.:

  ```
  ListView lv = (ListView) findViewById(R.id.listView);
  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 Menü-Ressource geschieht, z.B.:

  ```
  @Override
  public void onCreateContextMenu(ContextMenu contMenu, View v,
                                  ContextMenu.ContextMenuInfo contextMenuInfo) {
      super.onCreateContextMenu(contMenu, v, contextMenuInfo);
      getMenuInflater().inflate(R.menu.event_list_context_menu, contMenu);
  }
  ```

  Die Methode erfährt per Parameter, für welches **View**-Objekt das Kontextmenü angefordert wurde, und erhält über ein Objekt vom Typ **ContextMenu.ContextMenuInfo** genauere In-

formationen. Abkömmlinge der Klasse **AdapterView** liefern ein Objekt der Klasse **AdapterView.AdapterContextMenuInfo**, das in der öffentlichen Instanzvariablen **id** den Indexwert der vom Kontextmenü betroffenen Tabellenzeile aufbewahrt.

- 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 **getItem()** nach seiner Kennung, z.B.:

```
@Override
public boolean onContextItemSelected(MenuItem item) {
    AdapterView.AdapterContextMenuInfo acmi =
        (AdapterView.AdapterContextMenuInfo) item.getMenuInfo();
    switch (item.getItemId()) {
        case R.id.evliconit_edit:
            editEntry(acmi.id);
            return true;
        case R.id.evliconit_del:
            delEntry(acmi.id);
            return true;
        default:
            return super.onContextItemSelected(item);
    }
}
```

Nach einer erfolgreichen Behandlung sollte die Methode den Wert **true** zurückmelden. Anderenfalls sollte sie die überschriebene Basisklassenvariante aufrufen und deren Rückgabe durchreichen.

7.6.3.2 Kontextabhängige Aktionen

An Stelle des traditionellen Kontextmenüs empfiehlt Google die so genannten *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. Informationen zu kontextabhängigen Aktionen bietet z.B. die Online-Dokumentation für Android-Entwickler.[1]

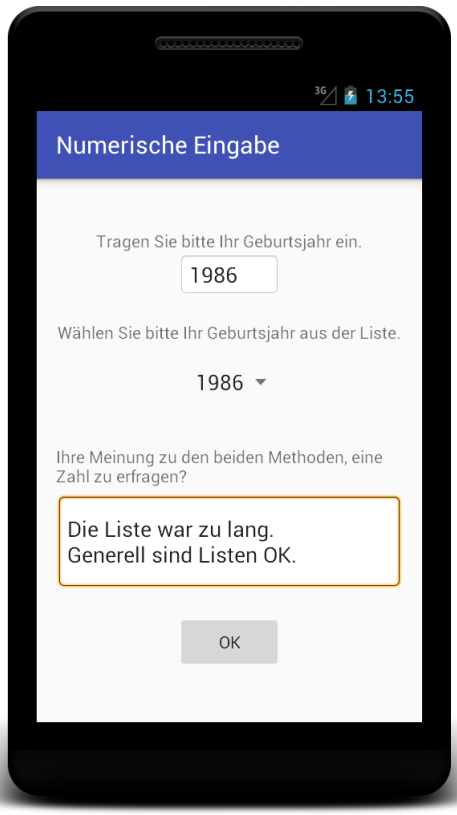
7.7 Übungsaufgaben zu Kapitel 7

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:

[1] Siehe https://developer.android.com/guide/topics/ui/menus.html



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 **GridLayout** erlauben im Unterschied zu den anderen Containern 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.

8 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).

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** *auftrag***)** verwendet werden.
- **Services**
  Mit der Methode **startService(Intent** *auftrag***)** lässt sich ein Service beauftragen, einen Einzelauftrag im Hintergrund auszuführen. Auch für die Methode **bindService()** wird ein **Intent**-Objekt benötigt. Ein gebundener Service bleibt aktiv, solange der aufrufende Kontext (meist eine Aktivität) existiert.
- **Broadcast Receiver**
  Mit der Methode **sendBroadcast(Intent** *ereignis***)** wird ein Rundruf an alle registrierten Empfänger versendet.

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.

Explizite Intents enthalten den voll qualifizierten Klassennamen (inkl. Paketname) der ausführenden Komponente (Activity oder Service) und dienen hauptsächlich zur Kommunikation *innerhalb* einer Anwendung.

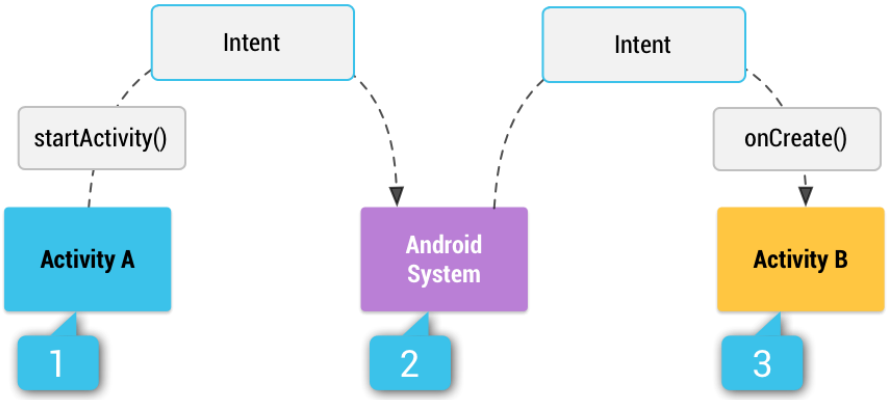
Dienstleistungen *fremder* Apps lassen sich über **implizite** Intents nutzen. Statt eines Komponentennamens enthalten sie eine Auftragsbeschreibung, im Wesentlichen bestehend aus einer **Aktion** und den zu verarbeitenden **Daten**. Wer als Auftragnehmer in Frage kommt, ermittelt Android durch den Vergleich …

- der Intent-Eigenschaften (siehe Abschnitt 8.1)
- mit den Intent-Filtern der installierten Apps (siehe Abschnitt 8.2).

In der App-Manifestdatei sind die Verarbeitungskapazitäten einer Komponente durch einen oder auch durch mehrere Intent-Filter beschrieben.

In der folgenden Abbildung beschreibt Google den Ablauf beim Starten einer Aktivität über einen *impliziten* Intent:[1]

[1] Quelle: http://developer.android.com/guide/components/intents-filters.html



Aktivität A verpackt einen Wunsch 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), benötigt er keine konkreten Informationen über den Auftragnehmer. Welche Aktivität durch Vermittlung des Betriebssystems letztlich tätig wird, entscheidet sich erst zur Laufzeit. Wenn mehrere Auftragnehmer in Frage kommen, erhält der Benutzer eine Wahlmöglichkeit.

8.1 Bestandteile eines Intent-Objekts

8.1.1 Name der Komponente

Wird als Auftragnehmer eine Anwendungskomponente explizit genannt (spezifiziert durch das Paket und die Klasse), dann resultiert ein explizites Intent-Objekt und es findet keine weitere Auswertung von Intent-Bestandteilen zur Ermittlung einer passenden Komponente statt. Apps verwenden explizite Intents hauptsächlich dazu, ihre eigenen Aktivitäten oder Dienste zu starten.

Bei der Erstellung eines expliziten **Intent**-Objekts (Paket: **android.content**) 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 Methode einer anonymen Klasse stammt, wird im ersten Parameter dem Schlüselwort **this** der Name der umgebenden **Activity**-Klasse vorangestellt, um das Aktivitätsobjekt anzusprechen:

```
Intent secAct = new Intent(MainActivity.this, SecondaryActivity.class);
```

Aus den beiden **Intent**-Konstruktorparametern 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-Kopmponente 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.

8.1.2 Aktion

Die (auszuführende oder per Broadcast zu berichtende) Aktion wird per **String**-Objekt angegeben, wobei in Frage kommen:

- In der Klasse **Intent** oder anderen Systemklassen (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 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";
  ```

Nach Möglichkeit sollten zur Spezifikation einer Aktion die in Framework-Klassen definierten Konstanten verwendet werden.

Man kann die Aktion per **Intent**-Konstruktor 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:[1]

[1] Quelle: https://developer.android.com/reference/android/content/Intent.html

Intent-Feldname	Ziel	Zweck
ACTION_MAIN	Activity	Aktivität als Basis für eine neue Aufgabe starten
ACTION_VIEW	Activity	Angegebene Daten (z.B. einen Kontakt) anzeigen
ACTION_EDIT	Activity	Modifikation der angegebenen Daten ermöglichen
ACTION_PICK	Activity	Wahl eines Items aus den angegebenen Daten ermöglichen, Auswahl zurückliefern
ACTION_WEB_SEARCH	Activity	Beginnt der Suchtest mit *http* oder *https*, 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.

Von der auszuführenden Aktion hängen die weiteren Bestandteile eines Intents wesentlich ab.

8.1.3 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[1] 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 das **Uri**-Objekt und/oder den MIME-Typ der zu bearbeitenden Daten festlegen:

- **setData(Uri** *daten***)** setzt das **Uri**-Objekt mit der Adresse der Daten
- **setType(String** *typ***)** setzt den MIME-Typ der Daten
  Die Methode **setData()** kann überflüssig sein, weil z.B. zur **ACTION_SEND** per Extras-Bestandteil (siehe Abschnitt 8.1.5) der zu versendende Text geliefert wird. In diesem Fall genügt es, per **setType()** den MIME-Typ zu deklarieren.
- **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.

Die für unsere Zwecke wesentlichen URI-Bestandteile sind:

*schema***://***host***:***port*/*pfad***?***abfrage*

Beispiele:

```
http://www.source.com/download/data/chart.svg
content://com.android.contacts/contacts
```

[1] Siehe z.B.http://www.iana.org/assignments/media-types/media-types.xhtml

Beim Schema **content** ist klar, dass die Daten durch einen auf dem lokalen Gerät installierten Content Provider (siehe Kapitel 13) verwaltet werden. In diesem Fall kann Android den MIME-Typ problemlos ermitteln.

Aktion und Daten definieren zusammen, was getan werden soll, z.B.[1]

Aktion	Per URI beschriebene Daten	Resultierender Auftrag
ACTION_VIEW	content://com.android.contacts/contacts	Liste mit 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 Standardbibliotheksklassen 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);
startActivity(abook);
```

8.1.4 Kategorie

Eine Intent-Kategorie wird per **String** angegeben und enthält Informationen über die Beschaffenheit der Komponente, welche den Auftrag erledigen kann. Zu einem Intent sind beliebig viele Kategorien erlaubt, doch kommen die meisten Intents ohne Kategorienangabe aus.

Zwei 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 4.3) und dem Anwendungsstarter des Systems bekannt sein.

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).

---

[1] Die Beispiele stammen von der Webseite http://developer.android.com/reference/android/content/Intent.html

8.1.5 Extras

Über Schlüssel-Wert - Paare vereinbart man Informationen, welche die ausführende Komponente benötigt (z.B. den Empfänger und den Betreff beim Versenden einer Mail). Man kann die Informationen einzeln durch Aufrufe einer Überladung der Methode **putExtra()** (mit zwei Parametern für den Schlüssel und den Wert) vereinbaren, z.B.:

```
newContact.putExtra(ContactsContract.Intents.Insert.NAME, "Kurt Saar");
```

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 so genannten *Kontrakt-Klasse* fündig (vgl. Abschnitt 13.1.5). Wird z.B. das Android-Adressbuch per **content**-URI angesprochen, dann finden sich die Schlüsselnamen in der Klasse **ContactsContract.Intents.Insert**.[1]

8.1.6 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);
```

8.2 Intent-Filter

Um für eine App-Komponente die Fähigkeiten zur Verarbeitung *impliziter* Intents zu deklarieren, erhält das zugehörige Element der App-Manifestdatei ein **intent-filter** - Kindelement (oder auch mehrere). Ein impliziter Intent wird einer Komponente nur dann zugestellt, wenn seine Anforderungen bzgl.

- auszuführender Aktion,
- zu verarbeitender Daten
- und 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.

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. Dementsprechend sollten für einen Service auch keine Intent-Filter definiert werden.

Eine Komponente ohne Intent-Filter kann nur explizite Intents erhalten.

---

[1] Dies ist eine statische Mitgliedsklasse von **Intents**, die wiederum als statische Mitgliedsklasse in **ContactsContract** definiert ist.

Die Verwendbarkeit einer Komponente durch fremde Apps wird durch das Manifest-Attribut **android:exported** gesteuert, wobei folgende Regeln gelten:

- Sind Intent-Filter vorhanden, hat das Attribut **android:exported** die Voreinstellung **true**, so dass die Aktivität auch von fremden Apps genutzt werden kann.
- Sind *keine* Intent-Filter vorhanden, hat das Attribut **android:exported** die Voreinstellung **false**, so dass 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.

8.2.1 Bestandteile eines Intent-Filters

Das **intent-filter** - Element zu einer Komponente definiert 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>
  ```
  Wie das Beispiel zeigt, ist zur Bezeichnung einer Aktion nicht eine Konstante aus der Klasse **Intent** anzugeben, sondern der vollständige Inhalt, z.B.:
  Falsch: **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 8.2.2).
- **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()** verwendeter 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, das die Firma Google in ihrer Online-Dokumentation zur Android-Programmierung verwendet:[1]

```
<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 unterscheiden.

8.2.2 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:[2]

<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:

[1] Siehe http://developer.android.com/reference/android/content/Intent.html
[2] Vgl. http://developer.android.com/guide/topics/manifest/data-element.html

- **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. **http**). Das für Android spezifische Schema **content** wird verwendet für den Zugriff auf Daten, die durch einen lokal installierten Content Provider verwaltet werden (siehe Kapitel 13). 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 also 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.
- **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. Viele Intent-Filter beschränken sich im **data**-Element auf die Angabe eines MIME-Typs (siehe Beispiel am Ende von Abschnitt 8.2.1).

8.3 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.

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**
- Namen festlegen und Klick auf **Finish**

Das Android Studio legt eine Klasse für die neue Aktivität an und trägt diese in die Manifestdatei der Anwendung ein, z.B.:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="de.zimkand.myapplication">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        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>
        <activity android:name=".SecondaryActivity"></activity>
    </application>
</manifest>
```

Obligatorisch ist nur der Name der realisierenden Klasse. Zusätzlich kann z.B. ein im Action- bzw. App Bar anzuzeigender Titel angegeben werden:

```
<activity
    android:name="de.zimkand.explintentsim.SecondaryActivity"
    android:label="@string/title_activity_secondary" >
</activity>
```

8.3.1 Einfacher Aktivitätsstart

Aus einer Aktivität (z.B. aus einer Startaktivität mit Wahlmöglichkeit) 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 = (Button) findViewById(R.id.startSecAct);
btnStartSecAct.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        Intent secAct = new Intent(MainActivity.this, SecondaryActivity.class);
        startActivity(secAct);
    }
});
```

Weil im Beispiel ein Objekt einer anonymen Klasse als Ereignisempfänger am Werk ist, wird über den Namen der umgebenden Klasse und das Schlüsselwort **this** das **Activity**-Objekt ermittelt.

Um 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);
  ```

Der Aufruf von **startActivity()** kehrt sofort zurück, und in der **onClick()** - Methode des Beispiels könnten weitere Anweisungen stehen, sogar weitere Aufrufe der Methode **startActivity()**. Unabhängig davon ist die primäre Aktivität mittlerweile im Back Stack der aktiven Task gelandet (vgl. Abschnitt 4.3). Nach Beendigung der Methode **onClick()** erhält die primäre Activity die in Abschnitt 5 beschriebenen Lebenszyklus-Methodenaufrufe (**onPause()**, **onStop()**).

8.3.2 Extradaten übergeben und in der sekundären Aktivität verwenden

Soll die sekundäre Aktivität Daten zur näheren Auftragsbeschreibung erhalten, kann man diese mit der Intent-Methode **putExtra()** als Schlüssel-Wert - Paare im **Intent**-Objekt unterbringen (vgl. Abschnitt 8.1.5), z.B.:

```
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;

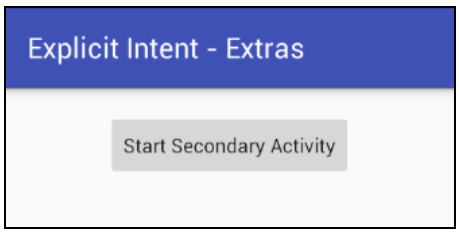
import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.widget.TextView;

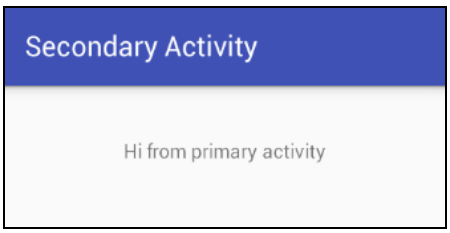
public class SecondaryActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_secondary);

        Bundle extras = getIntent().getExtras();
        TextView etMess = (TextView) findViewById(R.id.etMess);
        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:





8.3.3 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 so genannten *Request-Code*, besitzt, z.B.:

```
private int REQUEST_CODE_EXPLICIT = 12;
startActivityForResult(secAct, REQUEST_CODE_EXPLICIT);
```

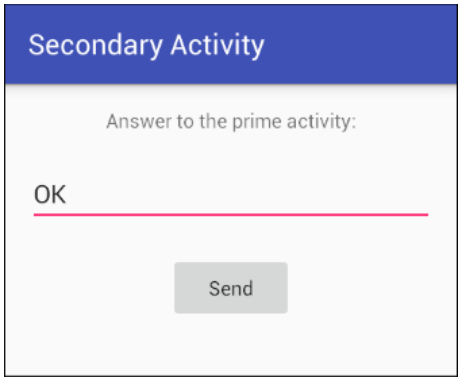
Außerdem muss der Auftraggeber die **Activity**-Methode **onActivityResult()** überschreiben, die bei Beendigung der sekundären Aktivität aufgerufen wird. Sie erhält per Parameter folgende Informationen:

- 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 Result-Code, mit dem die sekundäre Aktivität den Ausgang ihrer Bemühungen kategorisiert
- Ein **Intent**-Objekt
  Android benutzt also 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 im statischen Feld `EXTRA_MESS`:

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent intent) {
    if (requestCode == REQUEST_CODE_EXPLICIT) {
        TextView tv = (TextView) findViewById(R.id.tvAnswer);
        if (resultCode == RESULT_OK)
            tv.setText(getString(R.string.answer) + " " +
                       intent.getStringExtra(EXTRA_MESS));
        else
            tv.setText(getString(R.string.answer) + " " +
                       getString(R.string.neg_answer));
    }
}
```

Um eine Benutzeräußerung entgegen nehmen zu können, erhält die sekundäre Aktivität ein erweitertes Layout mit Texteingabefeld und Befehlsschalter:



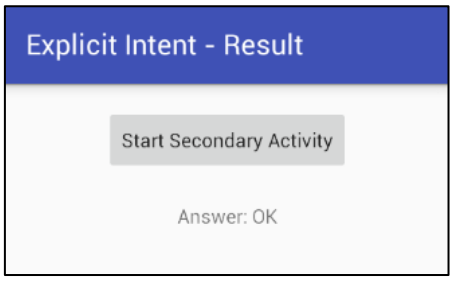
Nun bleibt noch zu beschreiben, wie die sekundäre Aktivität den Result-Code und das **Intent**-Retour-Objekt erstellt. Im Beispiel geschieht dies in der **onClick()** - Methode zum Befehlsschalter:

```
Button btnSend = (Button) 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 besprochene Beispielprogramm ist im folgenden Ordner zu finden:

...\BspUeb\Intent\Explizit\Result

8.4 Fremde Aktivitäten über implizite Intents starten

Schon die Android-Grundausstattung bietet zahlreiche Helfer, die man in eigenen Apps nutzen kann. Auf der folgenden Webseite

http://developer.android.com/guide/components/intents-common.html

findet sich Lösungen für viele Standardaufgaben (z.B. Foto aufnehmen, E-Mail versenden, Position auf einer Landkarte anzeigen).

8.4.1 Intent-Auflösung

Bei einem impliziten Intent ermittelt Android (durch Befragen seines Package-Managers), welche Komponenten als Empfänger in Frage kommen. Ein Intent-Filter muss drei Tests bestehen, bevor die zugehörige Komponente ein **Intent**-Objekt zur Erledigung 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 (per **Uri**-Objekt und/oder MIME-Typ (aus dem **Uri**-Objekt erschlossen oder explizit deklariert) in seiner Unterstützungsliste enthalten.[1]

8.4.2 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:

---

[1] Auf der folgenden Webseite ist das recht umfängliche Regelwerk für den Data-Kompatibilitätstest zu finden:
http://developer.android.com/guide/components/intents-filters.html#DataTest

- 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	`content`
Host	`com.android.contacts`
Pfad	`contacts`

  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 8.1.3)

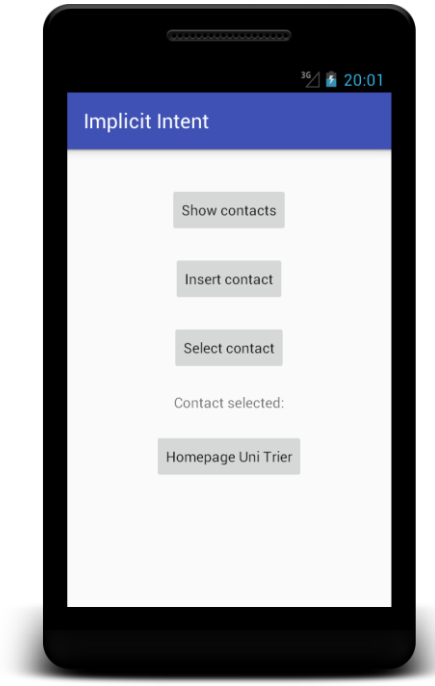
  ```
  ContactsContract.Contacts.CONTENT_URI
  ```

In einem einfachen Beispielprogramm erstellen wir das **Intent**-Objekt in der **onClick()** - Methode zu einem Befehlsschalter:

```
Button btnViewContacts = (Button) 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);
    startActivity(abook);
  }
});
```

Das **Intent**-Objekt dient als Parameter der Methode **startActivity()**, die Android auffordert, eine zum Auftrag passende Aktivität zu suchen und zu starten.

Kurze Zeit nach einem Klick auf den initiierenden 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.:

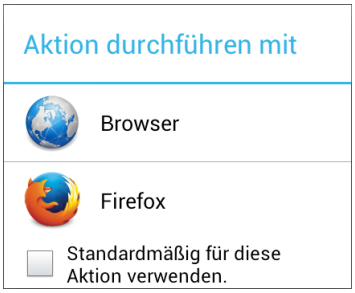
```
Intent abook = new Intent();
abook.setAction(Intent.ACTION_VIEW);
abook.setData(ContactsContract.Contacts.CONTENT_URI);
```

Im nächsten Beispiel wird ein Browser mit der Anzeige einer Webseite beauftragt:

```
Button btnViewUT = (Button) findViewById(R.id.viewUT);
btnViewUT.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        Intent viewUT = new Intent(Intent.ACTION_VIEW,
                                   Uri.parse("http://www.uni-trier.de/"));
        startActivity(viewUT);
    }
});
```

Das im **Intent**-Konstruktor verwendete **Uri**-Objekt wird mit der statischen **Uri**-Methode **parse()** aus einer Zeichenfolge erstellt.

Findet der Package-Manager *mehrere* Activities, die sich als Auftragsnehmer für geeignet halten, darf der Benutzer entscheiden, z.B.:

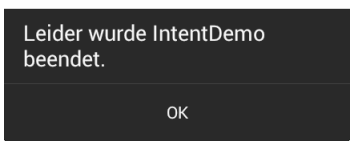


Soll dem Benutzer zu einem **Intent**-Objekt unabhängig von einer früheren Entscheidung für eine standardmäßig zu verwendende App auf jeden Fall ein Auswahldialog vorgelegt werden, erstellt man zum ursprünglichen **Intent**-Objekt ein neues mit der **Intent**-Methode **createChooser()** und startet dieses, z.B.:

```
Intent origInt = new Intent();
  .  .  .
Intent chooser = Intent.createChooser(origInt, "@string/title_chooser");
startActivity(chooser);
```

Im zweiten **createChooser()** - Parameter wird ein Titel für den Nachfragedialog festgelegt.

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 eine passende Aktivität kennt, z.B.:

```
Intent powerContacts = new Intent(Intent.ACTION_POWER_CONNECTED,
                                  ContactsContract.Contacts.CONTENT_URI);
if (powerContacts.resolveActivity(MainActivity.this.getPackageManager()) != null)
    startActivity(powerContacts);
```

Das als Parameter für **resolveActivity()** benötigte **PackageManager** - Objekt ermittelt man mit der **Context**-Methode **getPackageManager()**. Weil das Quellcodesegment aus einer anonymen Klasse stammt, wird über den Namen der umgebenden Klasse und das Schlüsselwort **this** das **Activity**-

Objekt ermittelt. Es beherrscht die Methode **getPackageManager()**, weil die Klasse **Activity** von der Klasse **Context** abstammt.

Im folgenden Beispiel wird das Android-Adressbuch um einen neuen Kontakt erweitert, wobei ein **Intent**-Objekt mit Extradaten zum Einsatz kommt:

```
Button btnInsertContact = (Button) 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");
        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.

8.4.3 Rückmeldung der sekundären Aktivität auswerten

Die in Abschnitt 8.3.3 vorgeführte Rückmeldetechnik unter Verwendung der Methode **startActivityForResult()** klappt grundsätzlich 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, so dass eine Weiterverarbeitung der erhaltenen Daten (z.B. Kontaktdaten) möglich ist. Details werden wir im Zusammenhang mit Datenbank- bzw. Abfragetechniken kennen lernen (siehe Kapitel 12).

Im folgenden Beispiel wird die Auswahl-Aktion angefordert, und die zu verarbeitenden Daten werden über das **Uri**-Objekt zum Android-Adressbuch spezifiziert:

```
Button btnSelectContact = (Button) 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(getPackageManager()) != 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 = (TextView) 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 8.4 erstellte Beispielprogramm ist im folgenden Ordner zu finden:

...\BspUeb\Intent\Implizit

8.5 Intents in Wartestellung

Man kann ein explizites **Intent**-Objekt, das einen Aufruf einer eigenen Anwendungskomponente bewirkt, in ein Objekt der Klasse **PendingIntent** verpacken und einer fremden App zur Verfügung stellen, die es später mit den Rechten der eigenen App 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.[1]

8.6 Übungsaufgaben zu Kapitel 8

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.

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.

[1] https://developer.android.com/reference/android/app/PendingIntent.html

9 Multithreading

Wie Sie bereits wissen, werden per Voreinstellung alle Komponenten einer Android-App von derselben 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**, so dass 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 Rückreise per Browser zu ermitteln, sofern das Mobilfunknetz verfügbar ist.

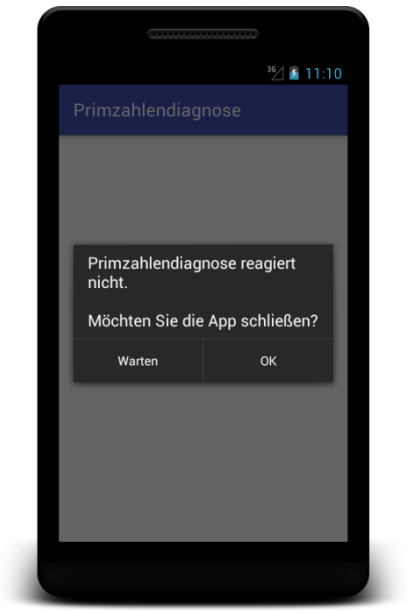
Begrenzt werden die Multitasking-Fähigkeiten eines Android-Geräts nur durch den verfügbaren Speicher und die CPU-Leistung. Wie in Abschnitt 4.4 zu erfahren war, kann Android bei Speichermangel Prozesse nach einer Prioritätenliste terminieren, wobei dann alle enthaltenen Anwendungskomponenten betroffen sind.

Weil in der Regel mehr Prozesse vorhanden sind als CPU-Kerne, besteht 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 *preemtiven* 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.

Im Prozess einer Android-App laufen per Voreinstellung 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 7.4). Die Bedienoberfläche darf nicht durch zeitaufwändige Methodenaufrufe (z.B. mit einem Netzwerkzugriff, einer Datenbankabfrage 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. Wenn eine App 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.:



Solche Auftritte sind peinlich und geschäftsschädigend. Zeitaufwändige Arbeiten müssen aus dem UI-Thread heraus gehalten und in einem **Hintergrund-Thread** ausgeführt werden.

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 die CPU-Kerne auszulasten und Zeit zu sparen. Mittlerweile (2016) 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 auf einem Gerät mit Quad-Core - CPU deutlich schneller laufen als auf einem Single-Core - Gerät. Mittlerweile verwenden 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.
- 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 7 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. Im Zusammenhang mit dem Multithreading sind also zwei Regeln zu beachten:

- Der UI-Thread (alias: Haupt-Thread) darf nicht durch aufwändige Methodenaufrufe belastet werden. Diese müssen in Hintergrund-Threads ausgelagert werden.
- Zugriffe auf grafische Bedienelemente sind nur aus dem UI-Thread erlaubt, so dass ein Hintergrund-Thread seine Änderungswünsche über den UI-Thread kanalisieren muss.

Analoge Regeln gelten auch für die meisten anderen grafischen Bedienoberflächen (z.B. JavaFX, Swing).

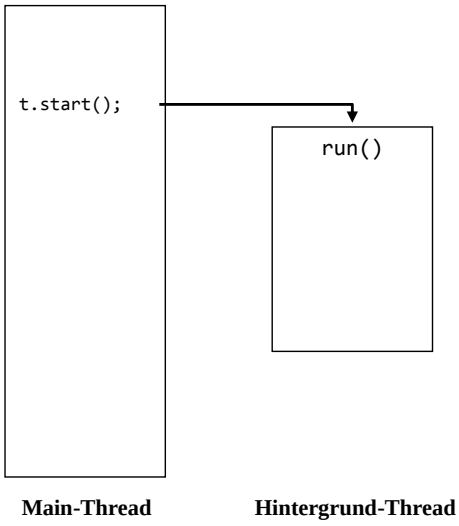
Während jeder *Prozess* einen eigenen Adressraum besitzt, laufen die *Threads* eines Programms im selben Adressraum ab, so dass sie gelegentlich auch als *leichtgewichtige Prozesse* bezeichnet werden. Sie haben einen gemeinsamen Heap-Speicher, wohingegen jeder Thread als selbständiger Kontrollfluss bzw. Ausführungsfaden aber einen eigenen Stack-Speicher benötigt.

9.1 Hintergrund-Thread mit elementaren Mitteln realisieren

Ein Thread wird in Java durch ein Objekt aus der Klasse **Thread** oder aus einer abgeleiteten Klasse realisiert. Eine Option besteht darin, von **Thread** eine anonyme Klasse 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.:

```
public void onClick(View arg0) {
    t = new Thread() {
        public void run() {
            // Im Hintergrund auszuführende Tätigkeiten
        }
    };
    t.start();
}
```

Nach dem Starten 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.

Mit der eben skizzierten Technik erstellen wir nun eine Variante des Beispielprogramm `Prime Detection`, das auch bei großen Primzahlkandidaten (trotz der mathematisch simplen Prüfmethode) 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 die Kommunikation zwischen dem Hintergrund-Thread, der irgendwann sein Ergebnis melden möchte, und dem UI-Thread, der als einziger Thread berechtigt ist, Änderungen an den **View**-Objekten der Bedienoberfläche vorzunehmen.

In der **onClick()** - Methode zum Befehlsschalter wird derselbe vorübergehend stillgelegt, damit bis zur Erledigung des Auftrags keine weiteren Aufträge abgeschickt werden können. Dann wird ein **Thread**-Objekt erstellt mit einer **run()** - Methode, welche die Zeichenfolge mit dem Primzahl-Kandidaten vom **EditText**-Steuerelement bezieht und anschließend den aus Abschnitt 6.10.3 bekannten Algorithmus ausgeführt:

```
private Thread t;
private Handler h = new Handler();
  .  .  .
@Override
public void onClick(View view) {
    button.setEnabled(false);
    t = new Thread() {
        @Override
        public void run() {
            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)+0.5);
            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);

            h.post(new Runnable() {
                @Override
                public void run() {
                    result.setText(sResult);
                    button.setEnabled(true);
                }
            });
        }
    };
    t.start();
}
```

Das fertige **Thread**-Objekt 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, so dass die **run()** - Methode des Objekts baldmöglichst im Haupt-Thread ausgeführt wird. In dieser Methode 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:[1]

---

[1] 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();
    }
}
```

```
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.[1]

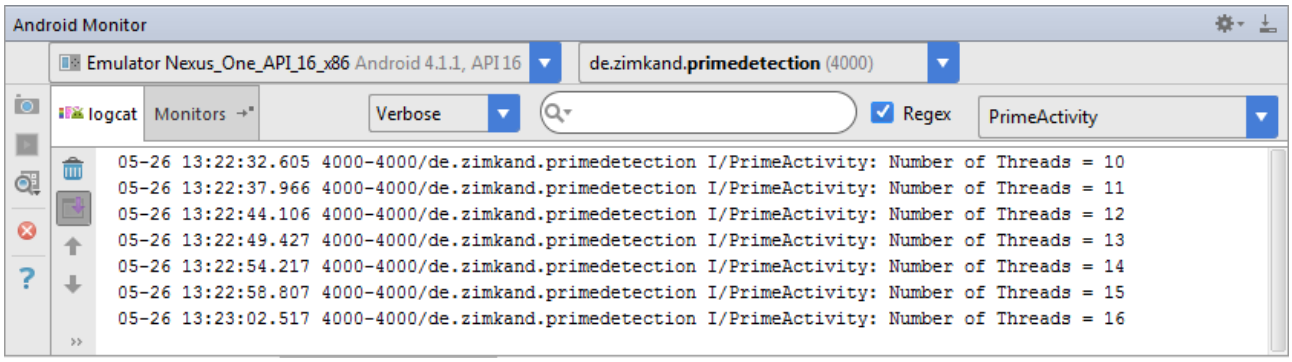
9.2 Helfer-Threads dürfen nicht zur Altlast werden

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 5 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.

Um diesen groben Programmierfehler zu demonstrieren, ergänzen wir in der **onCreate()** - Methode des Beispielprogramms aus Abschnitt 9.1 zwei Anweisungen, welche die Anzahl der im Prozess der App aktiven Threads ins **logcat**-Fenster der Entwicklungsumgebung schreiben:

```
final String TAG = this.getClass().getSimpleName();
int numberOfThreads = Thread.getAllStackTraces().keySet().size();
Log.i(TAG, "Number of Threads = " + String.valueOf(numberOfThreads));
```

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:



Zur Beseitigung des Problems müssen wir die Activity-Methode **onDestroy()** überschreiben und 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:

---

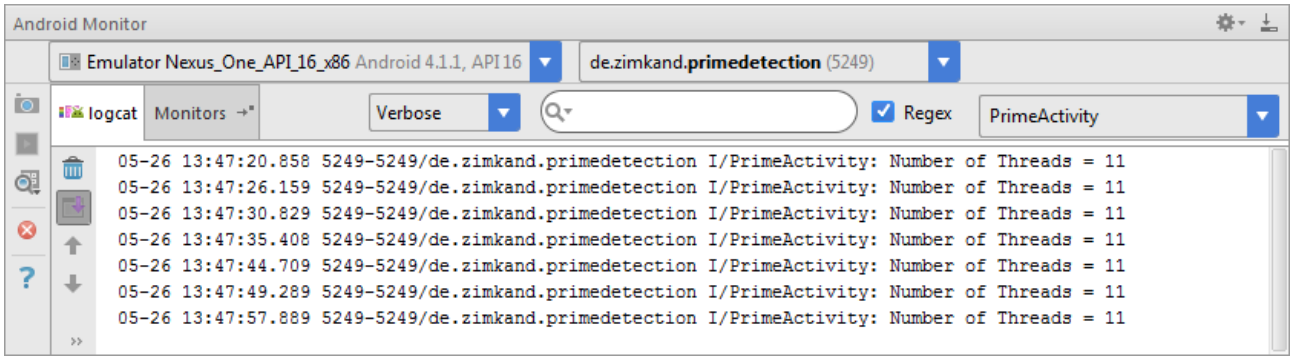
[1] Siehe http://developer.android.com/reference/android/os/Handler.html

```
@Override
protected void onDestroy() {
    super.onDestroy();
    if (t != null)
        t.interrupt();
}
```

Ein freundlicher und 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 nicht mehr zu Thread-Altlasten:



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

9.3 AsyncTask

Die Klasse **AsyncTask** aus der Android-Standardbibliothek vereinfacht die asynchrone Aufgabenabwicklung …

- ohne Belastung des UI-Threads
- mit anschließender Ergebnisanzeige (und optionaler zwischenzeitlicher Fortschrittsanzeige) im UI-Thread

Auf den Android-Entwicklerseiten findet sich die folgende Charakterisierung und Einsatzempfehlung zur Klasse **AsyncTask**:[1]

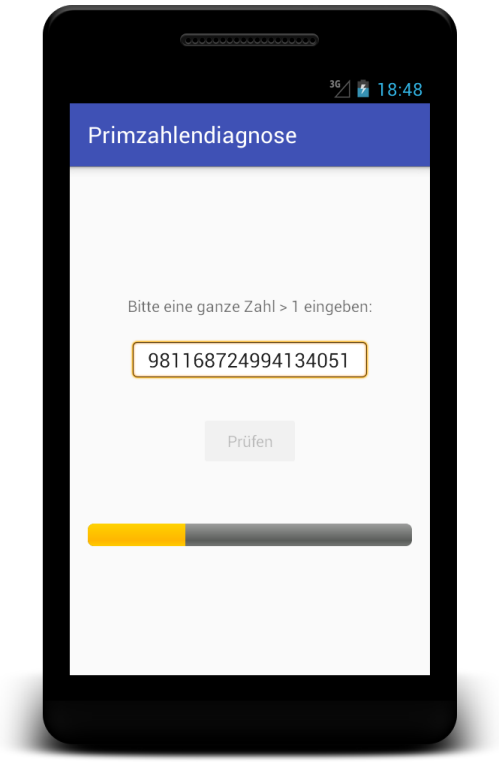
[1] Quelle: http://developer.android.com/reference/android/os/AsyncTask.html, abgerufen am 26.05.2016

> 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 des **Java Concurrent APIs** fehlt in diesem Kurs die Zeit (siehe z.B. Baltes-Götz & Götz 2016, Kapitel 16).

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 und liefern somit das in Abschnitt 7.5.6 versprochene Beispiel zur Verwendung des **ProgressBar**-Steuerelements nach. 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 `PrimeActivityAsyncTask` erhalten soll:

```
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)+0.5);
        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);
    }
}
```

AsyncTask ist abstrakt und generisch definiert,

```
public abstract class AsyncTask<Params, Progress, Result> {
  . . .
}
```

und wir müssen bei der Angabe der Basisklasse drei Typparameter konkretisieren:

- **Params**
  Dies ist der beim Starten einer asynchronen Aufgabenbearbeitung (siehe unten) an die Methode **execute()** zu übergebende Parametertyp.
- **Progress**
  Die Methode **onProgressUpdate()** zur Meldung des Bearbeitungsfortschritts hat einen Parameter vom Typ **Progress**.
- **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.[1] 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:

```
@Override
protected void onPreExecute() {
    result.setText("");
    button.setEnabled(false);
    progressBar.setVisibility(ProgressBar.VISIBLE);
    progressBar.setProgress(0);
}
```

- **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. Falls per **execute()** *mehrere* Einzelarbeiten in Auftrag gegeben wurden, kann dank Serienparameter eine entsprechende Anzahl von Zwischenergebnissen geliefert werden (z.B. Download-Stände für mehrere Dateien). 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:

[1] https://developer.android.com/reference/android/os/AsyncTask.html

```
@Override
protected void onProgressUpdate(Integer... progress) {
    progressBar.setProgress(progress[0]);
}
```

- **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);
}
```

Auch bei Verwendung der Klasse **AsyncTask** müssen wir aktiv dafür sorgen, dass beim Zerstören einer Aktivität die von ihr initiierten und noch aktiven Hintergrundaktivitäten beendet werden (vgl. Abschnitt 9.2). 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 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;
```

Das komplette AS-Projekt mit der im aktuellen Abschnitt beschriebenen `Prime Detection` - Variante ist im folgenden Ordner zu finden:

...\BspUeb\Multithreading\PrimeDetectionAT

9.4 Übungsaufgaben zu Kapitel 9

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

1. Ein **AsyncTask**-Objekt kann nur einmal ausgeführt werden.
2. Wenn Android eine Aktivität zerstört (z.B. bei einem Orientierungswechsel), werden alle von der Aktivität gestarteten Threads automatisch beendet.
3. Erhält ein Thread die Botschaft **interrupt()**, wird er von Android beendet.

10 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.[1] 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 eine Dienst starten.

Ein weiterer Grund für die Definition eines Dienstes ist das Ziel, Funktionalität für andere Anwendungen anzubieten.

Zwar kann auch eine im Hintergrund befindliche Aktivität weiterarbeiten (sinnvollerweise unschädlich für die Bedienoberfläche in einem eigenen Thread), doch ist ein Hintergrundprozess bei Ressourcen-Verknappung weit eher von der Schließung bedroht als ein Dienstprozess (vgl. Abschnitt 4.4). Wir werden im Primzahlenbeispiel dafür sorgen, dass eine langwierige Diagnose in einem Dienst ausgeführt wird und z.B. einen Aktivitätsneustart wegen Orientierungswechsel übersteht.

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. Erheblich seltener kommt ein **entfernter Dienst** zum Einsatz (engl.: *remote service*), der in einem eigenen Prozess ausgeführt wird. Wir beschränken uns auf den lokalen Dienst.

Eine weitere Unterscheidung betrifft die Betriebsart von Diensten:

- **Gestartete Dienste**
  Ein über die **Context**-Methode **startService()** aktivierter Dienst kann im Hintergrund potentiell unbegrenzt lange existieren, also auch seinen Urheber überleben. Mögliche Gründe für seine Beendigung:
  - Der Dienst beendet sich selbst über die Methode **stopSelf()**, nachdem er seine Aufgabe (z.B. einen Datei-Download) erledigt hat.
  - Der Dienst wird von einer anderen Anwendungskomponente über die Methode **stopService()** beendet.
  - Der Dienst wird (samt Prozess) wegen Speichermangel vom System beendet.

  In der Regel verfolgt ein gestarteter Dienst nur *einem* Zweck und bietet kein API mit verschiedenen Methoden an. In besonders einfachen Fällen ist keine Ergebnisrückmeldung an den Aufrufer erforderlich.
- **Gebundene Dienste**
  Über die **Context**-Methode **bindService()** kann sich ein Klient an einen Dienst binden und diesen *Server* bis zum expliziten Lösen der Verbindung durch **unbindService()** nutzen. Als Klient kommt eine Aktivität, ein anderer Dienst oder ein Content Provider in Frage (nicht aber ein Broadcast Receiver). Es ist erlaubt, dass sich mehrere Klienten mit einem Dienst verbinden. Befinden sich Klient und Server in derselben Anwendung und im selben Prozess, dann kann der Klient die öffentlichen 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 Speichermangel besteht

[1] Wegen solcher Apps, die natürlich auch einen Haupt-Thread besitzen, sollte man eigentlich nicht von *UI-Thread* einer App sprechen.

In der Praxis werden die Unterschiede zwischen den beiden Service-Einsatzszenearien 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 Methode erlaubt das Starten des Dienstes durch andere Komponenten.
  - **onBind()**
    Diese Methode ermöglicht die Verbindungsaufnahme durch andere Komponenten.
- Auch ein gestarteter Dienst kann Ergebnisse an einen Aufrufer melden. Er erlaubt jedoch keinen Methodenaufruf durch Klienten.

Wird ein Dienst zunächst per **startService()** gestartet wird und später auch noch Bindungen erhält, dann müssen für ein reguläres, nicht durch Speichermangel bedingtes Ende folgende Bedingungen erfüllt sein:

- Der Dienst erhält eine Aufforderung zum Beenden (**stopService()** oder **stopSelf()**).
- Alle Bindungen werden gelöst.

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**). Google empfiehlt allerdings nachdrücklich, die von **Service** abgeleitete Klasse **IntentService** zu beerben, wenn ein startfähiger Dienst realisiert werden soll.

10.1 Gestarteter Dienst aus der Klasse IntentService

Über den aktuellen Abschnitt hinausgehende Informationen zur Nutzung der Klasse **IntentService** bietet z.B. die Online-Dokumentation für Android-Entwickler.[1]

10.1.1 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, so dass 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 einem Dienst generell relevanten Rückrufmethoden **onCreate()**, **onStartCommand()**, **onBind()** und **onDestroy()** 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 ein Klient 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:

---

[1] https://developer.android.com/training/run-background-service/index.html
https://developer.android.com/guide/components/services.html

- **START_REDELIVER_INTENT**
  Der Prozess wird neu gestartet, sobald ausreichend Speicher verfügbar ist, und erhält einen **onStartCommand()** - Aufruf mit dem letzten **Intent**-Objekt.
- **START_STICKY**
  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**
  Der Prozess wird *nicht* neu gestartet.

Die **onStartCommand()** - Überschreibung der Klasse **IntentService** liefert per Voreinstellung die Rückgabe **START_NOT_STICKY**. Über die **IntentService**-Methode **setIntentRedelivery()**

public void setIntentRedelivery(boolean *enabled***)**

lässt sich aber erreichen, dass **onStartCommand()** die alternative Rückgabe **START_REDELIVER_INTENT** liefert.

Die **onBind()** - Überschreibung der Klasse **IntentService** liefert die Rückgabe **null**, so dass keine Bindung möglich ist. Um dieses Verhalten zu ändern, muss **onBind()** überschrieben werden.

Neben der Rückrufmethode **onHandleIntent()** muss in einer **IntentService**-Ableitung noch ein öffentlicher und parameterfreier Konstruktor erstellt werden, so dass sich der Aufwand in engen Grenzen hält.

Wie es bei einem hohen Automatisierungsgrad zu erwarten ist, bestehen Einschränkungen im Vergleich zur generellen **Service**-Klasse:

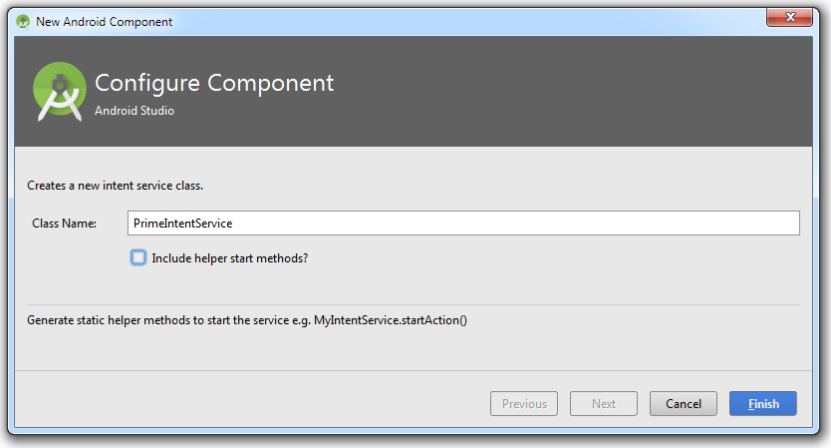
- Es kann nur *ein* Arbeits-Thread genutzt werden. Dieser führt die Aufträge nacheinander aus. Eintreffende **onStartCommand()** - Aufrufe werden nacheinander an die Methode **onHandleIntent()** weitergeleitet.
- Eine laufende Auftragsbearbeitung kann nicht gestoppt werden.

10.1.2 IntentService in das Projekt aufnehmen

Wir ersetzen im Primzahlenprogramm den Hintergrund-Thread durch einen gestarteten Service. Um die Hilfe der Entwicklungsumgebung bei der Definition der Service-Klasse in Anspruch zu nehmen, wählen wie aus dem Kontextmenü zum Paketeintrag im **Project**-Fenster den Befehl:

New > Service > Service (IntentService)

Wir legen einen Klassennamen fest und verzichten auf die **helper start methods**:



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 **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:

```
@Override
protected void onHandleIntent(Intent intent) {
    String strCand = intent.getExtras().getString("CAND");
    Context context = PrimeIntentService.this;
    LocalBroadcastManager locBcMan = LocalBroadcastManager.getInstance(this);
    Intent returnIntent = new Intent("de.zimkand.primedetection.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)+0.5);
    widthSeg = Math.max(10_000, 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);
}
```

10.1.3 Dienst starten und stoppen

Gestartet wird der **IntentService** in der Klickbehandlungsmethode der Aktivität über ein explizites **Intent**-Objekt mit der Extrainformation 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, die **onStartCommand()** - Methode des Dienstes aufzurufen. Falls der Dienst noch nicht existiert, wird er zunächst erstellt und erhält einen Aufruf der Methode **onCreate()**.

Weil sich der **IntentService** nach Erledigung eines Auftrags selbst stoppt, und außerdem eine laufende Auftragsbearbeitung nicht unterbrochen werden kann, muss sich die Aktivität *nicht* darum kümmern, den von ihr gestarteten Dienst mit **stopService()** zu beenden.

Wenn ein Dienst einen **onStartCommand()** - Aufruf erhalten hat, wird er vom System nur bei Speichermangel beendet. Für ein reguläres Ende ist ein **stopSelf()** - oder ein **stopService()** - Aufruf erforderlich.

Anschließend wird die in der Methode **onHandleIntent()** benutzte Technik zur Rückmeldung der Ergebnisse an die Aktivität beschrieben.

10.1.4 Ergebniskommunikation per LocalBroadcastManager

Google empfiehlt, zur Meldung von Ergebnissen an die beauftragende Komponente einen **LocalBroadcastManager** zu verwenden, sofern nur Komponenten der eigenen App informiert werden sollen.[1]

Wir erhalten ein **LocalBroadcastManager**-Objekt über die statische Methode **getInstance()** der Klasse **LocalBroadcastManager**:

```
LocalBroadcastManager locBcMan = LocalBroadcastManager.getInstance(this);
```

Versendet wird ein **Intent**-Objekt mit einer selbst definierten Aktion:

```
Intent returnIntent=new Intent("de.zimkand.primedetection.BROADCAST_ACTION");
```

Ein **Bundle**-Objekt

```
Bundle messBundle = new Bundle();
```

wird mit Extrainformationen gefüttert und in das **Intent**-Objekt eingefügt, z.B.:

```
messBundle.putFloat("progress", 0.0f);
messBundle.putString("result", res);
returnIntent.putExtras(messBundle);
```

Schließlich wird der **LocalBroadcastManager** gebeten, den Intent zu versenden:

```
locBcMan.sendBroadcast(returnIntent);
```

Damit sich die Aktivität beim **LocalBroadcastManager** (z.B. in ihrer Lebenszyklusmethode **onCreate()**) als Broadcast Receiver registrieren lassen kann,

```
@Override
protected void onCreate() {
      .  .  .
    receiver =  new PrimeEvalReceiver();
    IntentFilter intentFilter =
        new IntentFilter("de.zimkand.primedetection.BROADCAST_ACTION");
    LocalBroadcastManager.getInstance(this).registerReceiver(receiver, intentFilter);
}
```

sind einige Vorbereitungen nötig.

Im ersten Parameter der Methode **registerReceiver()** ist ein Objekt einer von **BroadcastReceiver** abstammenden Klasse zu übergeben. Wir definieren eine Ableitung als innere Klasse und deklarieren eine Instanzvariable

---

[1] https://developer.android.com/training/run-background-service/report-status.html

```
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 Methode **onReceive()** wertet das per Parameter erhaltene **Intent**-Objekt aus und nimmt entsprechende Änderungen an der Bedienoberfläche vor (Befehlsschalter `button`, Ergebnisanzeige im **TextView**-Objekt `result`, Fortschrittsbalken `progressBar`). Den Fortschrittsbalken sichtbar zu machen, ist nach einem Aktivitätsneustart und bereits laufender Primzahlendiagnose relevant.

Im zweiten Parameter von **registerReceiver()** ist ein **IntentFilter**-Objekt (vgl. Abschnitt 8.2) zu übergeben, wobei wir uns auf die Angabe der Broadcast-Aktion des **Intent**-Objekts beschränken können, das der **LocalBroadcastManager** in der Klasse `PrimeIntentService` versendet.

Ganz nebenbei machen wir erste Erfahrungen mit dem Broadcast Receiver, einer noch nicht offiziell behandelten Android-Komponente (siehe Kapitel 15).

Als Ergebnis der Mühe läuft nun im Beispielprogramm die Primzahlendiagnose in einem Hintergrund-Thread ab, der in einem Service agiert, also z.B. bei einem Orientierungswechsel ungestört weiterläuft, während die im Vordergrund befindliche Aktivität zerstört und neu gestartet wird. 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, so dass die Benutzer eine aufwändige Diagnose nicht neu starten müssen:



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).

Das komplette AS-Projekt mit der im aktuellen Abschnitt beschriebenen `Prime Detection` - Variante ist im folgenden Ordner zu finden:

...\BspUeb\Service\Service\IntentService

10.2 Verbundener Dienst

Obwohl die Klasse **IntentService** für das Primzahlenprogramm gut geeignet erscheint, ersetzen wir sie nun zu Übungszwecken durch einen gebundenen Service und lernen dabei ein nicht ganz triviales Framework kennen.

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 Dienst, der Musikstücke abspielt, und einer Activity, die sich temporär 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 dem eines gestarteten Dienstes mit dem Auftrag, eine Datei zu einem Cloud-Server zu übertragen.

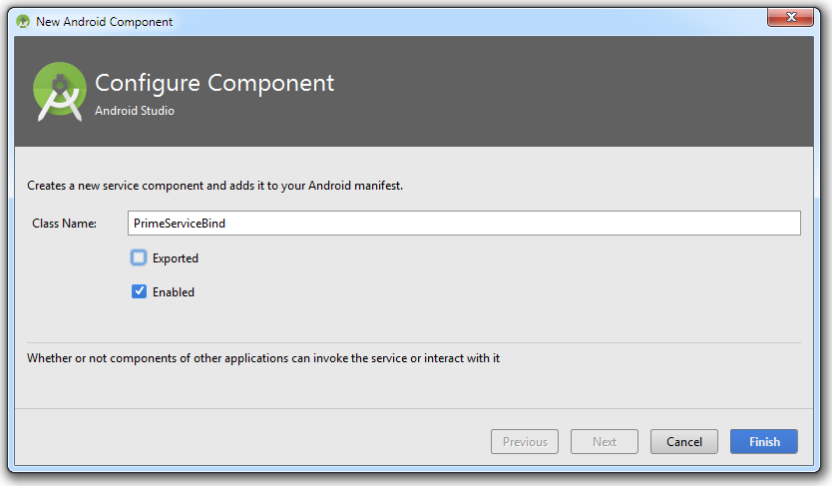
Damit ein Musikdienst auch dann aktiv bleibt, wenn keine Bindung an die Activity besteht, sollte er mit **startService()** gestartet werden und zusätzlich Verbindungen mit **bindService()** erlauben.

10.2.1 Rohling für einen bindungsfähigen Dienst per Entwicklungsumgebung erstellen

Um die Starthilfe der Entwicklungsumgebung bei der Definition der Service-Klasse in Anspruch zu nehmen, wählen wie aus dem Kontextmenü zum Paketeintrag im **Project**-Fenster den Befehl:

New > Service > Service

Wir legen einen Klassennamen fest und entfernen die Markierung beim Kontrollkästchen **Exported**, so dass 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=".PrimeServiceBind"
            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 im Manifest keine Intent-Filter vorhanden (empfohlen!), 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.

10.2.2 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***)**

Wir binden das **Service**-Objekt in der Lebenszeitmethode **onCreate()** der Aktivität ein:

```
Intent intent = new Intent(this, PrimeServiceBind.class);
bindService(intent, serviceConnection, Context.BIND_AUTO_CREATE);
```

Wir übergeben an **bindService()** ein **Intent**- sowie ein **ServiceConnection**-Objekt (siehe unten) und signalisieren mit der **int**-wertigen Konstanten **Context.BIND_AUTO_CREATE** im dritten Parameter, 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 (bound && !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.:

```
06-10 02:38:41.148 1551-1551/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, so dass kein Speicherleck (engl.: *memory leak*) vorliegt. Benutzer werden von der im **logcat**-Fenster der Entwicklungsumgebung erscheinenden Fehlermeldung *nicht* irritiert.

Ob aktuell eine Serviceverbindung besteht, merkt sich das Aktivitätsobjekt in der **boolean**-Instanzvariablen `bound`.

10.2.3 Klienten-Server - Kommunikation via Binder-Objekt

Von den möglichen Verfahren zur Kommunikation eines Klienten mit einem eingebundenen Dienst[1] (**Binder**-Ableitung, **Messenger**, **AIDL**) behandeln wir nur die einfachste Variante, deren Voraussetzungen aber fast immer gegeben sind: Klient und Dienst müssen zur selben Anwendung gehören und im selben Prozess laufen.

Bei einer Klienten-Server - Kommunikation per **Binder**-Objekt, sind folgende Typen beteiligt:

- **ServiceConnection**
  In der Methode **bindService()** zur Verbindungsaufnahme (siehe oben) muss der Klient ein Objekt einer die Schnittstelle **ServiceConnection** implementierenden Klasse übergeben. Dazu ist die Methode **onServiceConnected()** zu implementieren:

  public void onServiceConnected(ComponentName *className***, IBinder** *service***)**

  Sie wird aufgrund des klientenseitigen **bindService()** - Aufrufs beim Verbindungsaufbau 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 Aktivitätsklasse eine anonyme **ServiceConnection**-Implementation:

```
private ServiceConnection serviceConnection;
  .  .  .
serviceConnection = new ServiceConnection() {
    @Override
    public void onServiceConnected(ComponentName className, IBinder service) {
        PrimeServiceBind.PrimeBinder binder = (PrimeServiceBind.PrimeBinder) service;
        primeService = binder.getService();
        primeService.setHandler(handler);
        bound = true;
    }
    @Override
    public void onServiceDisconnected(ComponentName arg0) {
        bound = false;
    }
};
```

  In der Schnittstelle **ServiceConnection** ist außerdem die Methode **onServiceDisconnected()** vorgeschrieben. Über ihren Aufruf erfährt der Klient davon, dass der Service nicht mehr zur Verfügung steht. Ein möglicher Grund für den Aufruf von **onServiceDisconnected()** ist der vorangegangene Aufruf von **unbindService()** durch den Klienten.

- **Binder**
  Bei der im Beispiel bevorzugten, relativ einfachen Kommunikationstechnik liefert die **Service**-Lebenszyklusmethode **onBind()** ein Objekt aus einer **Binder**-Ableitung. Befinden sich Klient und Server in derselben Anwendung, kann sich eine **Binder**-Ableitung darauf beschränken, in einer (z.B. als `getService()` bezeichneten) Methode die Adresse des **Service**-Objekts zu veröffentlichen, damit der Klient anschließend die öffentlichen **Service**-Methoden direkt aufrufen kann. Unsere **Service**-Klasse definiert die folgende, von **Binder**- am stammende innere Klasse `PrimeBinder`

```
public class PrimeBinder extends Binder {
    public PrimeServiceBind getService() {
        return PrimeServiceBind.this;
    }
}
```

  und liefert in der Lebenszyklusmethode **onBind()** ein Objekt dieser Klasse:

```
@Override
public IBinder onBind(Intent intent) {
```

[1] https://developer.android.com/guide/components/bound-services.html

```
        return new PrimeBinder();
    }
```

- **Handler**
  Unsere **Service**-Klasse soll Aufträge asynchron in einem eigenen Thread erledigen und benötigt daher eine Möglichkeit, das Arbeitsergebnis nach Fertigstellung an den Klienten zu melden. Dazu erstellt unsere Aktivität eine Ableitung der Klasse **Handler**, die sich um die Nachrichtenwarteschlange des UI-Threads kümmert. Wir müssen die **Handler**-Methode **handleMessage()** überschreiben, die beim Eintreffen einer Nachricht für den UI-Thread aufgerufen wird. Somit haben wir die Möglichkeit, auf Nachrichten an den UI-Thread anwendungsadäquat zu reagieren, bevor wir durch Aufruf der überschriebenen Basisklassenmethode die Standardbehandlung auslösen:

```
final Handler handler = new Handler() {
    @Override
    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);
            return;
        }
        String ergebnis = bundle.getString("result");
        result.setText(ergebnis);
        button.setEnabled(true);
        progressBar.setVisibility(ProgressBar.GONE);
        super.handleMessage(message);
    }
};
```

  Die **Service**-Klasse erhält über ihre Methode **setHandler()**

```
public void setHandler(Handler handler) {
    this.handler = handler;
}
```

  eine Referenz auf das **Handler**-Objekt (siehe **ServiceConnection**-Methode **onServiceConnected()**) und kann daher per **sendMessage()** Nachrichten an den UI-Thread verschicken. Diese führen zum Aufruf der **Handler**-Methode **handleMessage()** im UI-Thread, so dass 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.

10.2.4 Service-Ableitung definieren

In der **Service**-Klasse `PrimServiceBind` 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 Klient 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 10.2.3). Man *muss*

die Methode **onBind()** implementieren, kann aber durch den Rückgabewert **null** eine Bindung verhindern.

Nach Auflösung aller Bindungen wird ein ausschließlich gebundener (nicht gleichzeitig auch gestarteter) Dienst automatisch vom System zerstört. Bei Speichermangel kann auch einem verbundenen Dienst dieses Schicksal widerfahren. Die Methode **onDestroy()** ist der richtige Ort für die Freigabe von Ressourcen (z.B. Threads, registrierten Broadcast Receivern). Im Beispiel führen wir die Primzahlendiagnose natürlich in einem eigenen Thread aus, der in **onDestroy()** beendet werden muss, damit es nicht zu der in Abschnitt 9.2 beschriebenen Ansammlung von Altlast-Threads kommen kann.

Die Lebenszyklusmethode **onCreate()**, die ggf. beim Erstellen des Dienstes (noch vor **onBind()** und **onStartCommand()**) vom System aufgerufen wird, ist im Beispiel nicht erforderlich.

Damit der Primzahlendienst genutzt werden kann, erstellen wir die Methode `evalCandidate()`, die einen **Thread** zur Durchführung der Primzahlendiagnose startet:

```
public void evalCandidate(String cand) {
    this.cand = cand;
    t = new Thread(this);
    t.start();
}
```

In der vom `PrimServiceBind`-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() {
    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) + 0.5);
    widthSeg = Math.max(10_000, 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);
        }
    }
```

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.[1]

In das **Bundle**-Objekt werden Name-Wert-Paare per **putFloat()** bzw. **putString()** eingefügt, z.B.:

```
bundle.putFloat("progress", i/(float)mdc*100);
bundle.putString("result", res);
```

Per **setData()** wird das **Bundle** in die **Message** eingefügt,

```
message.setData(bundle);
```

die per **sendMessage()** - Aufruf an den vom Aktivitätsobjekt zur Verfügung gestellten **Handler** auf die Reise geht:

```
handler.sendMessage(message);
```

Die Klasse `PrimServiceBind` 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 `PrimeServiceBind`-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.evalCandidate(s);
    }
}
```

---

[1] Bei Wiederverwendung desselben **Message**-Objekts kommt es zum Laufzeitfehler:

```
java.lang.IllegalStateException: The specified message queue synchronization barrier token
has not been posted or has already been removed.
```

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.[1]

Das komplette AS-Projekt mit der im aktuellen Abschnitt beschriebenen `Prime Detection` - Variante ist im folgenden Ordner zu finden:

...\BspUeb\Service\Service\LocalServiceBind

10.3 Vordergrunddienste

Ein Dienst kann und soll mit der Methode **startForeground()** gestartet werden, wenn seine Tätigkeit vom Benutzer „mit Interesse wahrgenommen“ wird (z.B. bei einem Programm zur Musikwiedergabe). In diesem Fall muss ein Eintrag für die Benachrichtigungszeile erstellt werden und es dem Benutzer ermöglichen, Kontakt mit der Anwendung aufzunehmen (meist durch Starten einer Activity). Dabei kommen die in Abschnitt 8.5 kurz erwähnten Intents in Wartestellung (engl.: *pending intents*) zum Einsatz. Ein Einstieg in dieses Thema ist über die Online-Dokumentation zu Android möglich.[2]

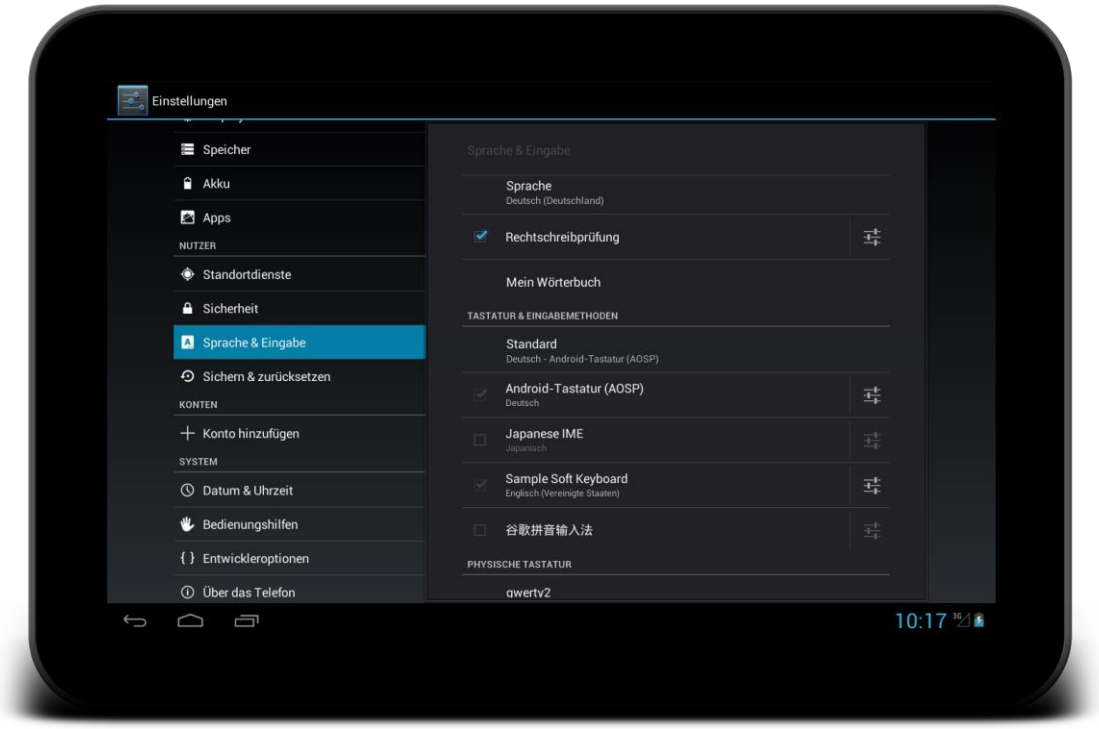
[1] https://developer.android.com/guide/components/services.html
https://developer.android.com/guide/components/bound-services.html

[2] https://developer.android.com/guide/components/services.html

11 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 Listenelement. Klickt der Benutzer in der Übersichtaktivität auf ein Listenelement, dann wird per Intent die Detailsaktivität gestartet. Für den Wechsel von der Details- zur Übersichtaktivität ist der Rückwärtsschalter zu verwenden.

Auf einem Tablet (mit einer Bildschirmdiagonale 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 Einstellungs-App vorgeführt wird:



Über Bedienelemente im linken Fragment, das eine Liste mit den Einstellungskategorien präsentiert, kann der Inhalt des rechten Fragments, das die Einstellungen zur gewählten Kategorie präsentiert, verändert werden. Seit Android 4 stehen Fragmente auch auf Smartphones zur Verfügung.

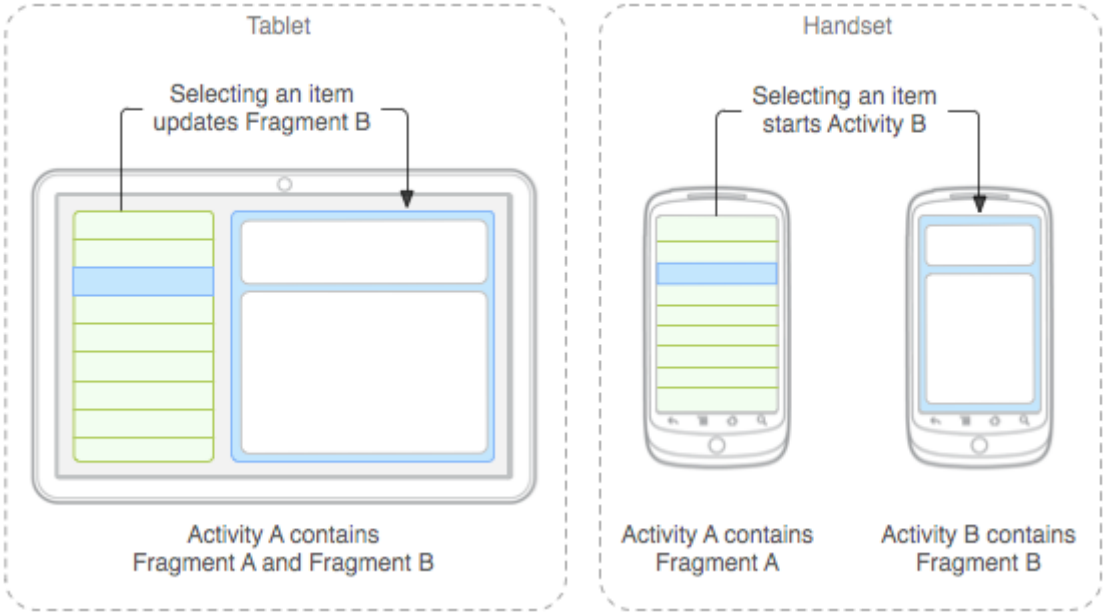
Sicher ist die ergonomische Unterstützung eines Tablet-Displays, das üblicherweise im Landschaftsformat betrieben wird, auch mit traditionellen Android-Mitteln möglich, z.B. mit einem horizontal orientierten **LinearLayout** - Container auf oberster Ebene und entsprechend positionierten Bedienelementen (z.B. ein **ListView**-Element auf der linken Seite und ein Container mit den Elemente 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, so dass ein App-Entwickler zur Unterstützung *beider* Gerätegattungen viel Parallelarbeit zu leisten hat.

Außer der rationellen 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.

11.1 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 Aktivität A auf das Listenfragment und startet nach Auswahl eines Listenelements die Aktivität B, welche das Detailsfragment einbindet:[1]



Das wesentliche Motiv für die Erweiterung von Android um Fragmente bestand darin, unterschiedliche Display-Größen bei minimalem Zusatzaufwand für die App-Entwickler 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. 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 11.7.2.2).
- **ListFragment**
  Diese Klasse unterstützt die Verwaltung von Listen (analog zur Klasse **ListActivity**). Wir werden sie gleich in einem Beispielprogramm verwenden.
- **PreferenceFragment**
  Diese Klasse unterstützt die Verwaltung einer Liste von **Preference**-Objekten (analog zur Klasse **PreferenceActivity**).

[1] http://developer.android.com/guide/components/fragments.html

- **WebViewFragment**
  Diese Klasse unterstützt die Anzeige von Webinhalten durch eine **WebView**-Komponente.

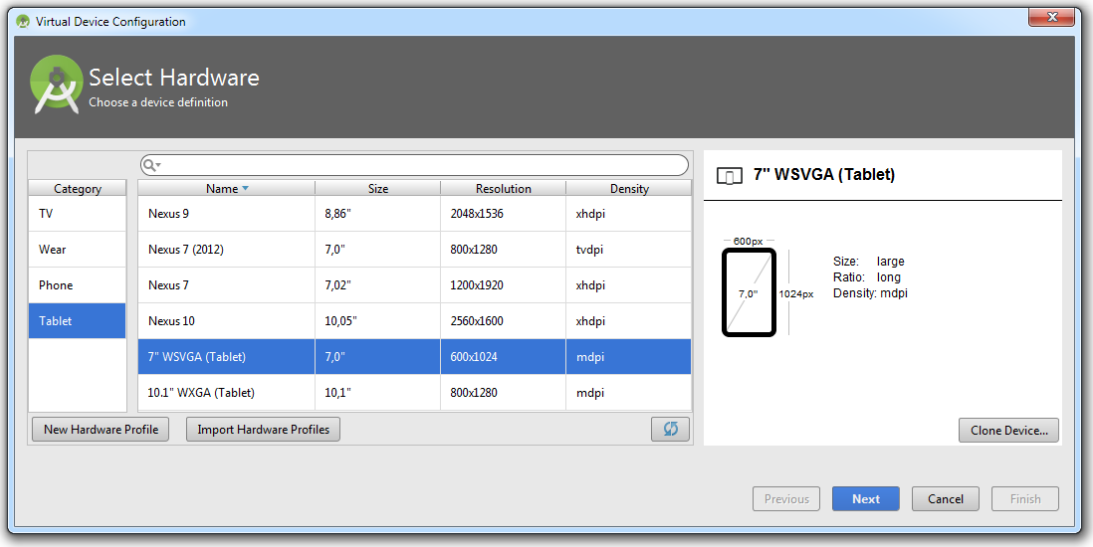
Zum Verhältnis bzw. zur Kooperation von Aktivitäten und Fragmenten ist zu sagen:

- Ein Fragment ist eine Aktivitätskomponente mit Anwendungslogik und eigenen Lebenszyklusmethoden, die in der Regel auch eine Bedienoberfläche präsentiert. Weil es *keine Anwendungs*komponente 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.
- 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, so dass der Benutzer per Rückwärtsschalter dorthin zurückkehren kann.

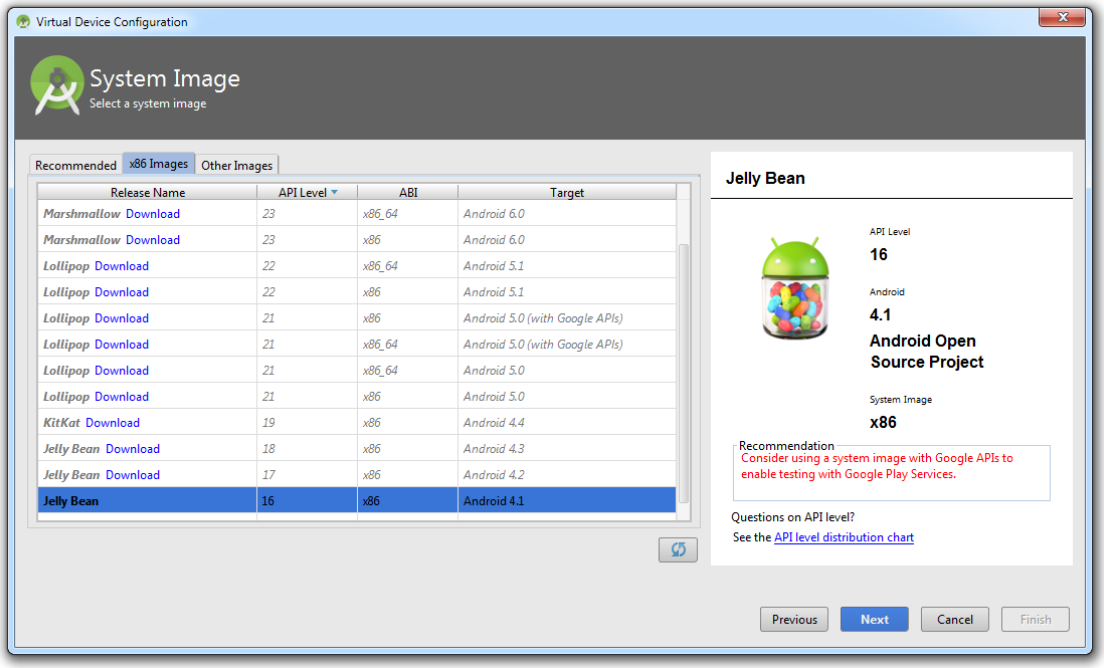
11.2 Vorschlag für ein AVD-Tablet

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)**:

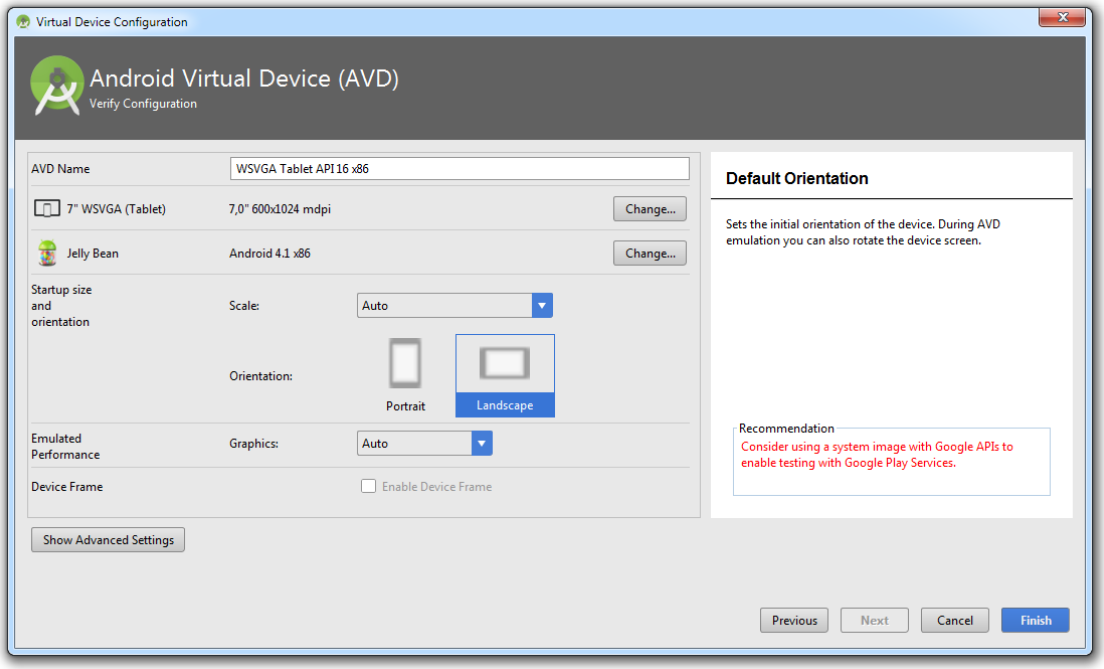


- Wählen Sie ein Ressourcen schonendes Android-System ab **API-Level** 16, z.B.:



Falls der *Hardware Accelerated Execution Manager* von Intel installiert ist (vgl. Abschnitt 2.7), sollte ein System-Image mit Intel-Technik verwendet werden.

- Wählen Sie einen **AVD-Namen** und die **Landscape**-Orientierung:



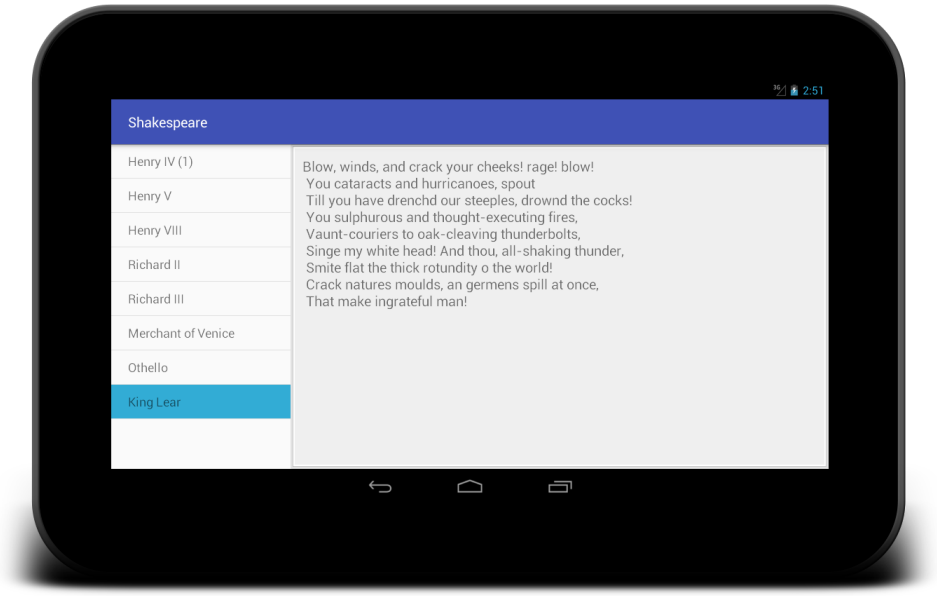
11.3 Beispiel für die Verwendung von Fragmenten

Im aktuellen Kapitel 11 verwenden wir zur Demonstration eine abgewandelte Version des von Google auf der Entwickler-Webseite zu Fragmenten beschriebenen Beispielprogramms, das Shakespeare-Text anzeigen kann.[1] In einer Liste werden die Titel von 8 Dramen 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)

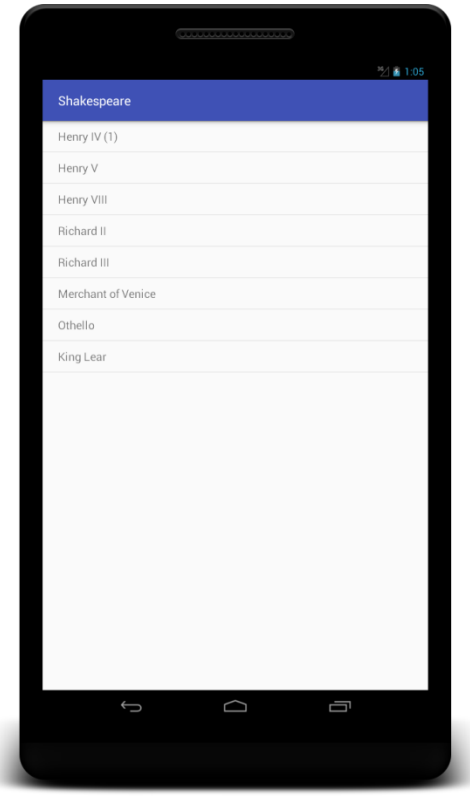
[1] https://developer.android.com/guide/components/fragments.html

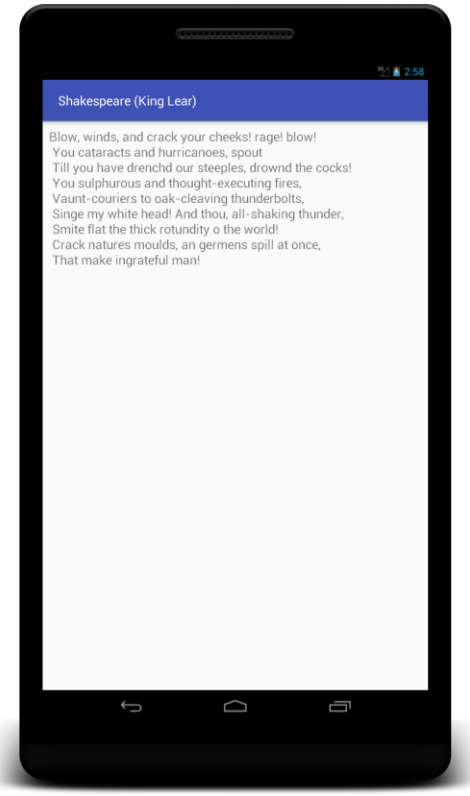
und die Textanzeige (rechts) simultan durch *eine* Aktivität mit *zwei* Fragmenten angezeigt. Im Hochformat werden jedoch *zwei* Aktivitäten verwendet, die jeweils *ein* Fragment enthalten, das die Titelliste bzw. den Text zum gewählten Titel anzeigt.

So arbeitet die fertige App, deren Innenleben Sie im weiteren Verlauf von Kapitel 11 kennen lernen werden, auf einem Tablet im Landschaftsformat:



Im Portraitformat sind zwei verschiedene Aktivitäten im Einsatz, die jeweils nur *ein* Fragment beherbergen:





Das komplette AS-Projekt mit dem Beispielprogramm ist im folgenden Ordner zu finden:

...\BspUeb\Fragmente\Shakespeare

11.4 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 6.2 zu konfigurationsabhängigen Ressourcen). In der Layoutdefinition für das **Landschaftsformat** (abzulegen im Projektordner **layout-land**, vgl. Abschnitt 7.3.1) werden die beiden in Abschnitt 11.3 beschriebenen Fragmente in einem horizontal orientierten **LinearLayout** - Container (siehe Abschnitt 7.3.1) 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 werden die beiden alternativ verwendbaren Techniken für die Fragmentintegration demonstriert:

- Das Fragment mit einer Liste von Shakespeare-Dramen wird analog zu einem **View**-Objekt durch ein **fragment** - Element repräsentiert (mit *klein* geschriebenen Namensanfang!). Dessen **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 eines einzelnen Textes wird später per Programm in den **FrameLayout** - Container (vgl. Abschnitt 7.3.3) mit der Element-ID `details` eingefügt. Wie ein Fragment per Programm explizit erzeugt und per Fragment-Transaktion an einer vorbereiteten Layout-Stelle eingefügt wird, ist in Abschnitt 11.7 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 7.3.1).

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-Datei auskommen.

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 11.5 beschriebenen Lebenszyklusmethoden **onCreate()**, **onActivityCreated()** und **onCreateView()** 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 11.9)

11.5 Lebenszyklusmethoden bei Fragmenten

Ein Fragment besitzt eigene Lebenszyklusmethoden, wobei nur wenige Unterschiede im Vergleich zu einer Aktivität bestehen (vgl. Abschnitt 5). Google empfiehlt, in einer eigenen **Fragment**-Ableitung mindestens die folgenden Methoden zu überschreiben:[1]

- **onCreate()**
  Diese Methode wird von Android beim Erstellen einer Fragmentinstanz aufgerufen und eignet sich zum Initialisieren von Instanzvariablen, die den Zustand des Fragments definieren.
- **onActivityCreated()**
  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.
- **onCreateView()**
  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 Wurzelelement für die Steuerelemente des Fragments dient. Bei einem Fragment ohne Bedienoberfläche, liefert man die Rückgabe **null**.
- **onSaveInstanceState()**
  Diese Methode wird unmittelbar vor oder nach **onPause()** 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 werden muss. Die Zustände der Steuerelemente eines Fragments (z.B. Inhalt eines Texteingabefelds) werden (wie bei Aktivitäten) automatisch

[1] https://developer.android.com/guide/components/fragments.html

gesichert und restauriert, sofern sie eine Kennung besitzen (vgl. Abschnitt 5.1).[1] Um andere Zustandsinformationen (z.B. in Instanzvariablen) muss sich der Programmierer kümmern.

- **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. Es ist nicht garantiert, dass die im Lebenszyklus potentiell noch folgenden Methoden **onStop()** und **onDestroy()** tatsächlich aufgerufen werden. Nach Ausführung der Methode **onPause()** ist ein Fragment davon bedroht, bei Speichermangel zusammen mit der kompletten App eliminiert zu werden.

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.
- **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 11.7.2.2).
- **Destroyed**
  Für die Zerstörung eines Fragments kommen zwei Gründe in Frage:
  - Die umgebende Aktivität ist zerstört worden.
  - Das Fragment wurde aus der (sichtbaren) umgebenden Aktivität entfernt und dabei *nicht* auf dem Fragment-Stack der Aktivität abgelegt (siehe Abschnitt 11.7.2.2).

11.6 Fragment-Initialisierung

Im Shakespeare-Beispielprogramm wird die Fragment-Klasse zur Präsentation der Titelliste aus der SDK-Klasse **ListFragment** abgeleitet:

```
public class TitlesFragment extends ListFragment
                            implements FragmentManager.OnBackStackChangedListener {
    private boolean mDualPane;
    private int mCurCheckPosition = 0;
    private FragmentManager fm;
    private boolean uselessStackState = true;
```

[1] Bei einem **TextView**-Steuerelement klappt die automatische Sicherung und Wiederherstellung nur dann, wenn in der Layoutdefinition das Attribut **android:freezesText** den Wert **true** erhält.

```
    @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;

        if (savedInstanceState != null) {
            mCurCheckPosition = savedInstanceState.getInt("curChoice", 0);
        }

        if (mDualPane) {
            getListView().setChoiceMode(ListView.CHOICE_MODE_SINGLE);
            showDetails(mCurCheckPosition);
        }
    }

    @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) {
        showDetails(position);
    }

    @Override
    public void onBackStackChanged() {
        DetailsFragment details =
                          (DetailsFragment) fm.findFragmentById(R.id.details);
        if (details != null)
            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 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***, List<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 **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-Dramen) aus einer Ressource 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 11.4).

```
mDualPane = getActivity().findViewById(R.id.details) != null;
```

Im Dual-Pane - Modus soll das aktuell gewählte Listenelement hervorgehoben dargestellt werden. Dazu muss mit der **ListView**-Methode **setChoiceMode()** der passende Modus **ListView.CHOICE_MODE_SINGLE** gesetzt werden:

```
if (mDualPane) {
    getListView().setChoiceMode(ListView.CHOICE_MODE_SINGLE);
    showDetails(mCurCheckPosition);
}
```

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.

Im Dual-Pane - Modus sind das Listen- und das Detailsfragment simultan zu sehen. Die `TitlesFragment`-Methode `showDetails()` sorgt für die sofortige Anzeige des Detailsfragments (siehe Abschnitt 11.7).

Weil ein Listenfragment eine Bedienoberfläche anbietet, muss seine Klasse die Methode **onCreateView()** beherrschen, die zum Instanzieren der Steuerelementhierarchie dient (vgl. Abschnitt 11.5). Sie wird vor der ersten Anzeige des Fragments aufgerufen und muss ein **View**-Objekt abliefern, das als Wurzelelement 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:

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                         Bundle savedInstanceState) {
    return inflater.inflate(com.android.internal.R.layout.list_content,
            container, false);
}
```

Android liefert der Methode **onCreateView()** per Parameter einen **LayoutInflater** (vgl. Abschnitt 7.1) und außerdem:

- den Container in der Aktivität, der das Fragment aufnehmen wird
- Ein **Bundle**-Objekt, das ggf. Informationen über die Vorgängerinstanz des Fragment-Objekts enthält

Wird (wie in der Klasse **ListFragment**) die **LayoutInflater**-Methode **inflate()** 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 7.1). Beim **boolean**-Parameter in Position 3 muss unbedingt der Wert **false** übergeben werden, womit die Verwaltung des abgelieferten **View**-Wurzelelements 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 11.8).

11.7 Fragment-Transaktionen

Im Shakespeare-Beispielprogramm ist das Listenfragment dafür verantwortlich, bei Bedarf das Detailsfragment ins Spiel zu bringen. Das geschieht …

- im Dual-Pane - Modus bei der Initialisierung des Listenfragments durch die `TitlesFragment`-Methode **onActivityCreated()** (siehe Abschnitt 11.6) 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, ohne dass die explizite Registrierung eines Ereignisempfängers erforderlich wäre. Wir überschreiben diese Methode in unserer Klasse `TitlesFragment` folgendermaßen:

```
@Override
public void onListItemClick(ListView list, View v, int position, long id) {
    showDetails(position);
}
```

Die hier sowie in **onActivityCreated()** aufgerufene private `TitlesFragment`-Methode `showDetails()` sorgt für einen umgebungsgerechten Auftritt des Detailsfragments:

```
private void showDetails(int index) {
    mCurCheckPosition = index;
    if (mDualPane) {
        getListView().setItemChecked(index, true);
        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;
}
```

Im Dual-Pane - Modus wird zunächst veranlasst, dass unmittelbar nach dem Start der App (aufgrund des `showDetails()` - Aufrufs mit dem Argument 0) das erste Listenelement markiert ist, weil das in dieser Lage parallel sichtbare Detailsfragment den ersten Shakespeare-Text anzeigen wird:

```
getListView().setItemChecked(index, true);
```

Diese Anweisung ist überflüssig (aber unschädlich), wenn `showDetails()` durch `onListItemClick()` aufgerufen worden ist.

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 noch kein Detailsfragment, was bei einer neu gestarteten App der Fall ist.
- Das vorhandene Detailsfragment zeigt nicht den aktuell gewählten Shakespeare-Text an.

Dabei spielt der nun zu behandelnde Fragment-Manager eine zentrale Rolle.

11.7.1 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 11.4).
- Wurde ein Fragment per Programm in die Aktivität aufgenommen, ist die ID des heimatlichen 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.[1]

11.7.2 Fragmente per Programm erstellen und in eine Aktivität aufnehmen

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:

```
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 Fabrikmethode namens `newInstance()`, die per Parameter die Nummer des gewünschten Textes entgegen nimmt und als Rückgabe eine Referenz auf ein neues Objekt ihres Typs liefert:

```
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 **FramentManager** 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.setCustomAnimations(android.R.animator.fade_in, android.R.animator.fade_out);
ft.replace(R.id.details, details);
ft.commit();
```

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 **FragmentTransaction**-Methoden zu verwenden:

---

[1] Wie Sie aus Abschnitt 7.3.3 wissen, eignet sich der **FrameLayout**-Container zur Aufnahme von genau einem Steuerelement.

- **add()**
  Wie das Verhalten des Beispielprogramms beim Start im Querformat zeigt, klappt die Methode **replace()** auch dann, wenn das zu ersetzende Fragment noch *nicht* existiert.
- **remove()**

11.7.2.1 Animierte Übergänge

Optional lässt sich für die Fragment-Transaktion eine Animation vereinbaren. Mit dem Vorschlag aus dem originalen Google-Quelltext

```
ft.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE);
```

zeigte sich allerdings *keine* Animation. Die Methode **setCustomAnimations()** führt unter Verwendung von Animations-Definitionen aus dem Ressourcen-Arsenal des Betriebssystems (siehe oben) 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);
```

11.7.2.2 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 Fragment-Stapel 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 Fragment-Stapel mehrere „Scheiben“, bewirkt die Rückwärts-Taste zunächst die sukzessive Rückabwicklung der Fragment-Transaktionen, bevor schließlich die Aktivität verlassen wird.

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.

In unserem Shakespeare-Beispiel ist (wie in vielen ähnlichen Programmen) dafür zu sorgen, dass im Querformat-Modus bei der Rückkehr zu einem früheren Fragment 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);
    if (details != null)
        getListView().setItemChecked(details.getShownIndex(), true);
}
```

Wenn das Details-Fragment vorhanden, also der Dual-Pane - Modus aktiv ist, wird das Details-Fragment nach dem angezeigten Text befragt und die Markierung des **ListView**-Steuerelements im **ListFragment** entsprechend gesetzt.

In der Methode **onActivityCreated()** wird das `TitlesFragment`-Objekt beim Fragment-Manager als Interessent für **BackStackChanged**-Ereignisse angemeldet:

```
fm = getFragmentManager();
fm.addOnBackStackChangedListener(this);
```

11.8 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 `Titles-Fragment` basiert `DetailsFragment` auf **Fragment**:

```
public class DetailsFragment extends 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, wäh-

rend die Werte von Instanzvariablen verloren gehen, wenn sich der Programmierer nicht selbst darum kümmert.

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* funktionstüchtige Implementierung der Lebenszyklusmethode **onCreateView()** erbt (vgl. Abschnitt 11.5), muss sie die Methode **onCreateView()** überschreiben, um ihre **View**-Hierarchie zu erstellen.[1]

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.[2] 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:

```
<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>
         .  .  .
```

---

[1] Die von **Fragment** geerbte Methode **onCreateView()** liefert an ihren Aufrufer statt eines Wurzelelements für die Steuerelemente des Fragments lediglich **null** zurück:

```
public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container,
                         Bundle savedInstanceState) {
    return null;
}
```

[2] Die Berechnung des Innenrandes übernehmen wir ausnahmsweise ohne großes Grübeln aus der Vorlage.

```
        <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.

11.9 Fragmente im Single-Pane – Modus

Soll eine App sowohl auf großen als auch auf kleinen Display-Flächen ergonomisch agieren, muss sie auch das klassische Kartenstapel-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 `showDetails()` via **onListItemClick()** aufgerufen. Die Methode erkennt den Single-Pane - Betrieb und startet eine neue Aktivität über das in Abschnitt 8.3 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 = null;
    if (savedInstanceState == null) {
        details = new DetailsFragment();
        details.setArguments(getIntent().getExtras());
        getFragmentManager()
                .beginTransaction()
                .add(android.R.id.content, details, "defrag")
                .commit();
    } else
        details = (DetailsFragment) getFragmentManager().findFragmentByTag("defrag");
    setTitle(getTitle() +
       " ("+getResources().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, so dass die im Back Stack darunter liegende Hauptaktivität aktiv wird.

Wenn die Detailsaktivität im Portrait-Modus (ohne Vorgänger-Instanz) neu entsteht, dann wird …

- ein neues Detailsfragment erstellt,[1]
- per **setArguments()** mit dem Name-Wert - Paar aus dem **Intent**-**Bundle** versorgt
- und per Fragment-Transaktion (siehe Abschnitt 11.7) ins Layout der Detailsaktivität aufgenommen.

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 Wurzelelement 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.

Nach einer Neukreation der Activity im Portrait-Modus muss kein neues Fragment erzeigt werden. Weil das alte nach der Nummer des angezeigten Textes befragt werden soll, muss seine Adresse über die **FragmentManager**-Methode **findFragmentByTag()** ermittelt werden:

```
details = (DetailsFragment) getFragmentManager().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.

---

[1] Die `DetailsFragment`-Fabrikmethode **newInstance()** statt des Konstruktors zu benutzen, wäre aufwändiger, weil
- erst aus dem **Intent**-**Bundle** die Textnummer (ein **int**-Wert) extrahiert werden müsste,
- um diese anschließend in **newInstance()** wieder in ein Bundle für das Fragment zu verpacken.

Bei der gewählten Lösung wird das **Intent**-**Bundle** ohne Aus- und Einpacken direkt in das Fragment kopiert.

12 Datenbankverwaltung mit 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 einordnen kann. Wenn sich eine Kontakte-App nicht nur um Einzelpersonen 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 DBMS integriert.

Einige fortschrittliche Datenbanktechniken, über die z.B. Becker & Pant (2015, S. 265f) informieren, können aus Zeitgründen in diesem Kapitel nicht behandelt werden:

- Prepared Statements
  Werden mehrere SQL-Kommandos benötigt, die sich nur bei einzelnen Parametern (z.B. Attributausprägungen) unterscheiden, sollte ein Objekt vom Typ **SQLiteStatement** erstellt werden. Es enthält ein SQL-Kommando mit Fragezeichen an Stelle konkreter Attributausprägungen und wird zur Vorübersetzung an SQLite gesendet, so dass bei wiederholter Ausführung ein Leistungsgewinn resultiert.
- Transaktionen
  So bezeichnet man zusammengehörige Datenbankmodifikationen, die nur vollständig oder überhaupt nicht ausgeführt werden dürfen.

12.1 Besonderheiten bei SQLite unter Android

SQLite ist ein schlankes, ohne Konfigurationsaufwand einsetzbares relationales DBMS. Aufgrund der guten Integration in Android ist SQLite über komfortable Klassen leicht zu nutzen. Die im Kurs für emulierte Android-Geräte bevorzugte Android-Version 4.1 (API-Level 16) enthält die 2012 erschienene SQLite-Version 3.7.11.

Ein wichtiges Designziel bei SQLite war die Robustheit unter widrigen Umständen (z.B. Stromausfall, Speichermangel, rabiate Prozessbeendigung durch das Betriebssystem), so dass praktisch nie eine Datenbank verloren geht. Über die Hälfte des SQLite-Codes soll sich mit Tests beschäftigen (Mednieks et al. 2013, S. 296), was durchaus als Orientierungswert für andere Projekte mit einem hohen Qualitätsanspruch taugt.

Nähere Informationen zur Verwendung von SQLite bietet die folgende Webseite:

https://www.sqlite.org/docs.html

In SQLite kommt eine komplette Datenbank in einer *einzigen* 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/

Unter Android ist für Primärschlüssel einer Tabelle der Name **_id** vorgeschrieben, wenn man über einen **Cursor** (siehe unten) auf die Tabelle zugreifen will.

Für jeden, in einer beliebigen SQLite-Datenbankspalte gespeicherten Wert kann eine von den folgenden **Speicherklassen** verwendet werden:

- **NULL**
- **INTEGER**
  Vorzeichenbehaftete Ganzzahl, für die je nach Größe 1, 2, 3, 4, 6 oder 8 Bytes aufgewendet werden
- **REAL**
  IEEE-Gleitkommazahl mit 8 Bytes Speicheraufwand
- **TEXT**
  Text in der Datenbankkodierung (UTF-8, UTF-16BE oder UTF-16LE), der den individuell erforderlichen Speicherplatz belegt. SQLite ignoriert bei SQL-typischen Felddeklarationen wie z.B. `VARCHAR(255)` die Längenangabe.
- **BLOB**
  Ein BLOB (*Binary Large Object*) ist ein Array mit Elementen vom Typ **byte**.

In einer Zelle einer Datenbankspalte lassen sich Daten aus einer beliebigen Speicherklasse unterbringen, jedoch besitzt jede Datenbankspalte eine **Typaffinität**, und SQLite unternimmt dementsprechende Konvertierungsversuche. Es sind folgende Affinitäten möglich:

- **TEXT**
- **NUMERIC**
- **INTEGER**
- **REAL**
- **NONE**

Wird z.B. der Text "3.0e+1" in eine Zelle mit der Affinität **INTEGER** eingefügt, dann wird der Wert 30 abgespeichert. Mit den Typaffinitäten wird bezweckt, dass SQLite trotz Speicherklassen-Minimalismus eine möglichst hohe Kompatibilität zu anderen Datenbankverwaltungssystemen besitzt. Die Typaffinität einer Datenbankspalte wird aus dem Namen ihres deklarierten Datentyps abgeleitet, so dass z.B. aus dem deklarierten Typ `VARCHAR(255)` die Affinität **TEXT** resultiert. Weitere Details zu den Datentypen in der SQLite-Version 3 finden sich in der Online-Dokumentation.[1]

12.2 Direktkontakt mit SQLite per ADB

Beim ersten Kontakt mit SQLite verwenden wir eine konsolenorientierte Interaktion und lernen dabei ein wertvolles Werkzeug zur Fehlersuche kennen, das sich nicht nur bei Datenbankanwendungen nutzen lässt, um hinter die Android-Kulissen zu blicken.

12.2.1 SDK-Kommando adb shell

Wir verwenden die **Android Debug Bridge** (ADB), um eine Konsolenverbindung zu einem emulierten Android-Gerät herzustellen (vgl. Abschnitt 2.9):

[1] https://www.sqlite.org/datatype3.html

- 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:



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 sind ein Bindestrich und die vom emulierten Gerät verwendete Konsolenportnummer anzuhängen, z.B.:[1]

```
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:



Leider funktionieren Steuertasten (z.B. Backspace) nicht wie gewohnt, sondern produzieren (neben der korrekten Funktion) auch Steuerzeichen, z.B.:



Glücklicherweise tritt das Problem bei der Interaktion mit **SQLite** *nicht* auf. In Abschnitt 12.2.3 wird eine Lösung für das Problem beschrieben.

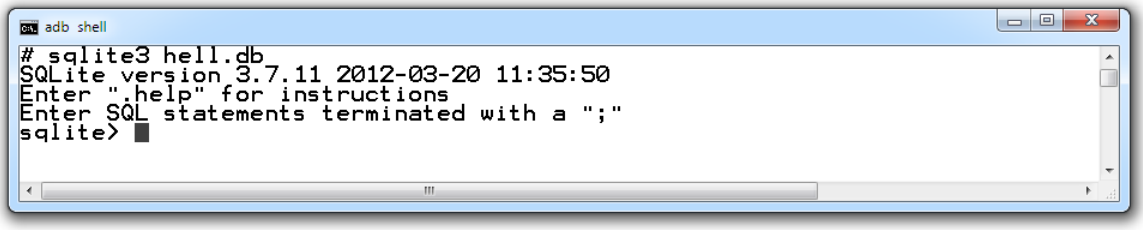
---

[1] Die Konsolenportnummer eines emulierten Gerätes wird im **Run**-Fenster der Entwicklungsumgebung protokolliert, z.B.:

```
Connected to process 1606 on device Nexus_One_API_16_x86 [emulator-5554]
```

12.2.2 Interaktion mit SQLite

Starten Sie eine Interaktion mit **SQLite** zur Bearbeitung der Datenbank **hell.db** über das Kommando **sqlite3** mit dem Datenbanknamen als Argument:



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

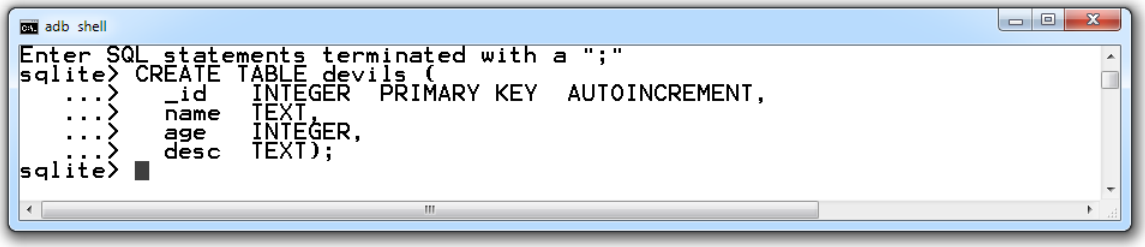
  INTEGER PRIMARY KEY AUTOINCREMENT

  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.

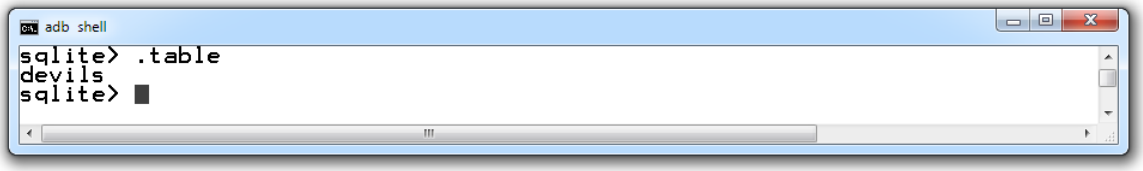
Wird das komplette Kommando

```
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:



Über das SQLite-Kommando **.table** (mit einem einleitenden Punkt!) erhält man eine Liste mit den Tabellen der Datenbank:



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;
```

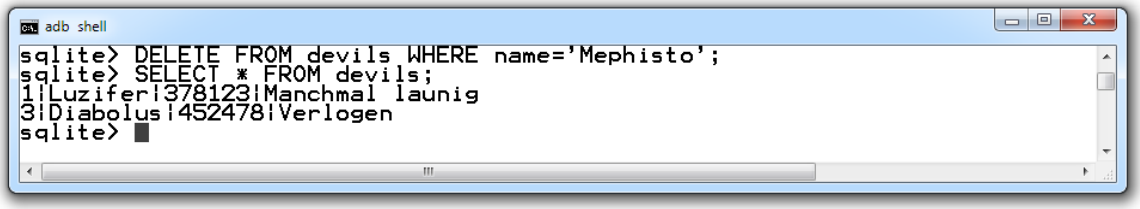
unter Verwendung der Klausel **ORDER BY** mit der Spezifikation **DESC** werden die Namen und Altersangaben der Teufel mit einem Alter von minimal 200.000 Jahren nach Alter absteigend sortiert ausgegeben:



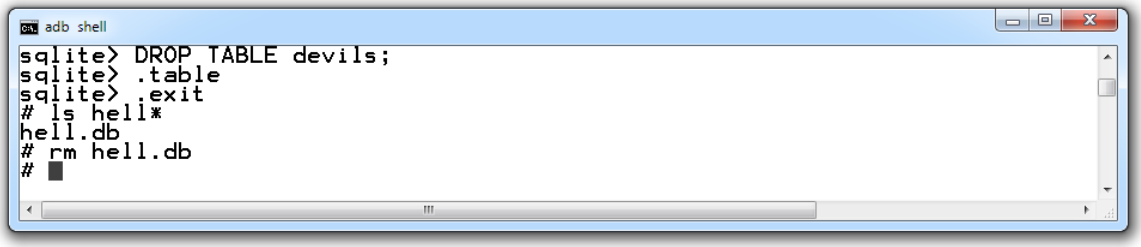
Mit dem Kommando **UPDATE** lassen sich Werte ändern, z.B.

```
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.:



Per **DROP TABLE** kann eine Tabelle gelöscht werden, z.B.:



Mit dem SQLite-Kommando **.exit** 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.

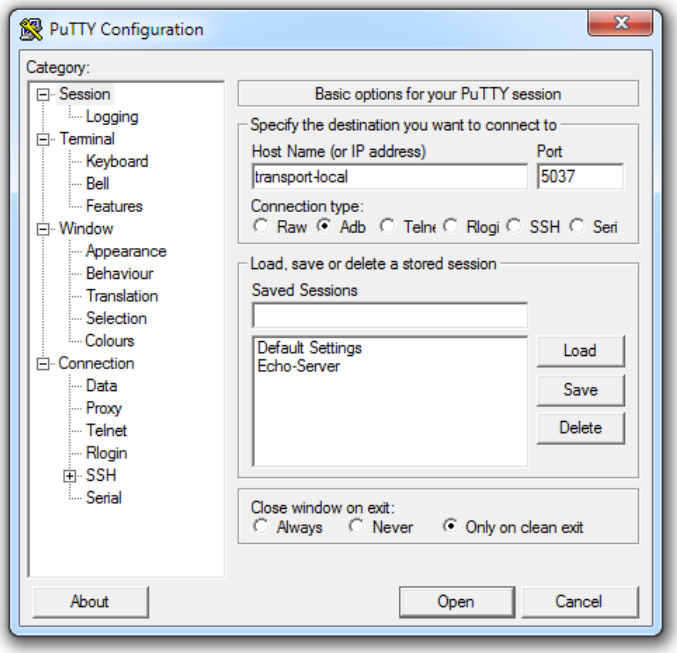
12.2.3 ADB-Version von PuTTY

Wer häufiger mit Android-Geräten per Konsole interagieren möchte, braucht (zumindest unter Windows) einen Ersatz für die per **adb shell** gestartete Konsole, die auf Steuertasten wie Backspace unpraktisch reagiert (siehe Abschnitt 12.2.1). Auf der folgenden Webseite

http://kyleknowledge.blogspot.de/2014/11/adb-enhanced-putty-replacement-for-adb.html

wird eine um ADB-Kompetenzen erweiterte Version des unter Windows beliebten Terminal-Programms PuTTY angeboten. Mit einem laufenden virtuellen Android-Gerät nimmt man folgendermaßen Kontakt auf:

- PuTTY per Doppelklick auf die Datei **putty.exe** starten
- **Adb** als **Connection type** wählen
- `transport-local` als **Host Name** eintragen und den vorgegebenen Port 5037 belassen:



- Klick auf **Open**

Nun kann man nach Herzenslust Tippfehler produzieren und korrigieren, z.B.:



12.3 Android-Typen zur Datenbankbearbeitung

12.3.1 SQLiteDatabase

Ein Objekt der Klasse **SQLiteDatabase** (Paket **android.database.sqlite**) repräsentiert die Datenbank im Programm. Mit der **SQLiteDatabase**-Methode **execSQL()** lassen sich beliebige SQL-Kommandos ausführen, insbesondere auch Datendefinitionskommandos (wie z.B. **CREATE TABLE**). Die **SQLiteDatabase**-Methoden **query()**, **insert()**, **update()** und **delete()** kapseln SQL-Abfrage- bzw. Modifikationskommandos.

12.3.2 SQLiteOpenHelper

Die abstrakte Klasse **SQLiteOpenHelper** (Paket **android.database.sqlite**) spielt im Android-Datenbank-Framework eine wichtige Rolle bei der Erstellung und bei der Aktualisierung einer Datenbank. Eine eigene Ableitung muss folgende Methoden implementieren:

- **onCreate()**
  Diese Methode wird beim Zugriff auf eine nicht existente Datenbank von Android aufgerufen. Hier definiert man das Datenbankschema (Namen der Tabellen, Namen und Datentypen der Tabellenspalten) mit SQL-Kommandos wie **CREATE TABLE**.
- **onUpgrade()**
  Diese Methode wird aufgerufen, wenn die Version einer angetroffenen Datenbank kleiner ist als die dem **SQLiteOpenHelper**-Konstruktor genannte Version. In diesem Fall muss **onUpgrade()** in der Lage sein, alle früheren, bei Kunden anzutreffenden Datenbankversionen verlustfrei in die aktuelle Version zu überführen.

Optional kann zusätzlich die Methode **onDowngrade()** implementiert werden. Sie arbeitet analog zu **onUpgrade()** und wird aufgerufen, wenn eine App auf eine jüngere Datenbankversion trifft. Das passiert z.B. dann, wenn ein Anwender zu einer früheren App-Version zurückgekehrt ist. Fehlt in einer solchen Situation eine **onDowngrade()** - Überschreibung, wirft die Basisklassenvariante eine **SQLiteException**.

Eine Android-App muss eine Datenbankdatei nicht selbst erstellen. An die **onCreate()** - Methode wird per Parameter eine neue, leere Datenbank zum Initialisieren übergeben.

Über die **SQLiteOpenHelper**-Methoden **getReadableDatabase()** und **getWritableDatabase()** mit einem **SQLiteDatabase**-Objekt als Rückgabe stellt man eine Datenbankverbindung her, wobei die Datenbank erstellt und/oder geöffnet wird:

- Existiert eine Datenbank mit der geforderten Version, wird diese geöffnet.
- Existiert keine Datenbank, wird unter Verwendung der **SQLiteOpenHelper**-Methode **onCreate()** eine Datenbank erstellt.
- Existiert eine Datenbank mit einer falschen Version, ist die **SQLiteOpenHelper**-Methode **onUpgrade()** oder die Methode **onDowngrage()** gefordert, das Problem zu lösen. Welche Version erforderlich ist bzw. beim Erstellen einer neuen Datenbankdatei dort zu dokumentieren ist, erfährt ein **SQLiteOpenHelper** -Objekt per Konstruktor (siehe Abschnitt 12.4.1).

Das von **getReadableDatabase()** bzw. **getWritableDatabase()** bereitgestellte **SQLiteDatabase**-Objekt bleibt fest mit dem **SQLiteOpenHelper**-Objekt assoziiert, und seine Adresse wird zur Beschleunigung späterer Zugriffe gespeichert. Weil die erste Verbindungsaufnahme mit einer Datenbank einen erheblichen Zeitaufwand verursachen kann, sollte sie möglichst *nicht* im Haupt-Thread stattfinden.

Die Methoden **getReadableDatabase()** und **getWritableDatabase()** liefern in der Regel dieselbe Referenz. Wenn eine vorhandene Datenbank nicht im Schreibmodus geöffnet werden kann (z.B. wegen Speichermangel), ...

- dann öffnet **getReadableDatabase()** die Datenbank im Lesemodus,
- während **getWritableDatabase()** eine **SQLiteException** wirft.

Zu einer Datenbank sollte es in der Regel nur *ein* **SQLiteOpenHelper**-Objekt und damit *eine* Datenbankverbindung geben. Dann garantiert das eine beteiligte **SQLiteDatabase**-Objekt die Thread-Sicherheit, indem es Datenbankzugriffe aus verschiedenen Threads serialisiert.[1] Weil eine SQLite-Datenbank zwar nur einen schreibenden, aber mehrere lesende Zugriffe erlaubt, kann es sinnvoll sein, mehrere **SQLiteOpenHelper**-Objekte (Datenbankverbindungen) zu verwenden. Von zwei konkurrierenden Schreibzugriffen wird aber einer scheitern, wobei kein Ausnahmefehler gemeldet, sondern nur eine **logcat**-Fehlermeldung geschrieben wird.[2]

---

[1] Dies besagt der am 26.06.2016 abgerufene Stack Overflow - Beitrag von Kevin Galligan:
http://stackoverflow.com/questions/2493331/what-are-the-best-practices-for-sqlite-on-android

[2] Diese Information stammt aus einem Stack Overflow - Beitrag von Kevin Galligan (siehe vorherige Fußnote).

Für eine **SQLiteOpenHelper**-Ableitung, die zum Öffnen einer bestimmten Datenbank vorgesehen ist, sollte sichergestellt werden, dass nur *ein* Objekt der Klasse erzeugt werden kann. Dieses so genannte **Singleton**-Muster ist z.B. durch eine im statischen Kontext verwaltete **SQLiteOpenHelper**-Instanz zu realisieren (siehe Abschnitt 12.4.1).

In der Android-Entwicklerszene wird viel darüber diskutiert, wann ein **SQLiteOpenHelper**-Objekt und damit die zugehörige Datenbankverbindung geschlossen werden sollte. Solange eine Anwendung nur *eine* Datenbankverbindung (nur ein **SQLiteOpenHelper**-Objekt) verwendet, kann nach einer von Kevin Galligan vorgetragenen Argumentation auf das Schließen dieser Verbindung verzichtet werden.[1]

Neben dem Singleton-Muster gibt es einen weiteren Lösungsansatz zur Vermeidung der Mehrfachkreation von **SQLiteOpenHelper**-Objekten:[2] Man verpackt die SQLite-Datenbank in einen Content Provider (siehe Kapitel 13). Diese eigentlich für den Datenaustausch zwischen verschiedenen Anwendungen gedachte Technik kann innerhalb der eigenen Anwendung dazu dienen, Datenbankzugriffe elegant zu verwalten und auch noch asynchron (außerhalb des Haupt-Threads) auszuführen.

12.3.3 Cursor

Als Rückgabe der **SQLiteDatabase**-Methode **query()**, die das SQL-Kommando **SELECT** kapselt, erhält man ein Objekt einer Klasse, die das Interface **Cursor** im Paket **android.database** implementiert. Dieses Objekt repräsentiert die Tabelle mit dem Abfrageergebnis und erlaubt einen Zugriff über einen Zeiger. Anschließend werden einige **Cursor**-Methoden kurz 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 Ergebniszeile 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. Bei einer großen Ergebnistabelle kann der *erste* Aufruf der **Cursor**-Methode **getCount()** lange dauern und zu einem ANR-Fehler führen, wenn er im UI-Thread stattfindet. Als Alternative bietet sich die statische Methode **queryNumEntries()** der Klasse **DatabaseUtils** im Paket **android.database** an.
- **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:

---

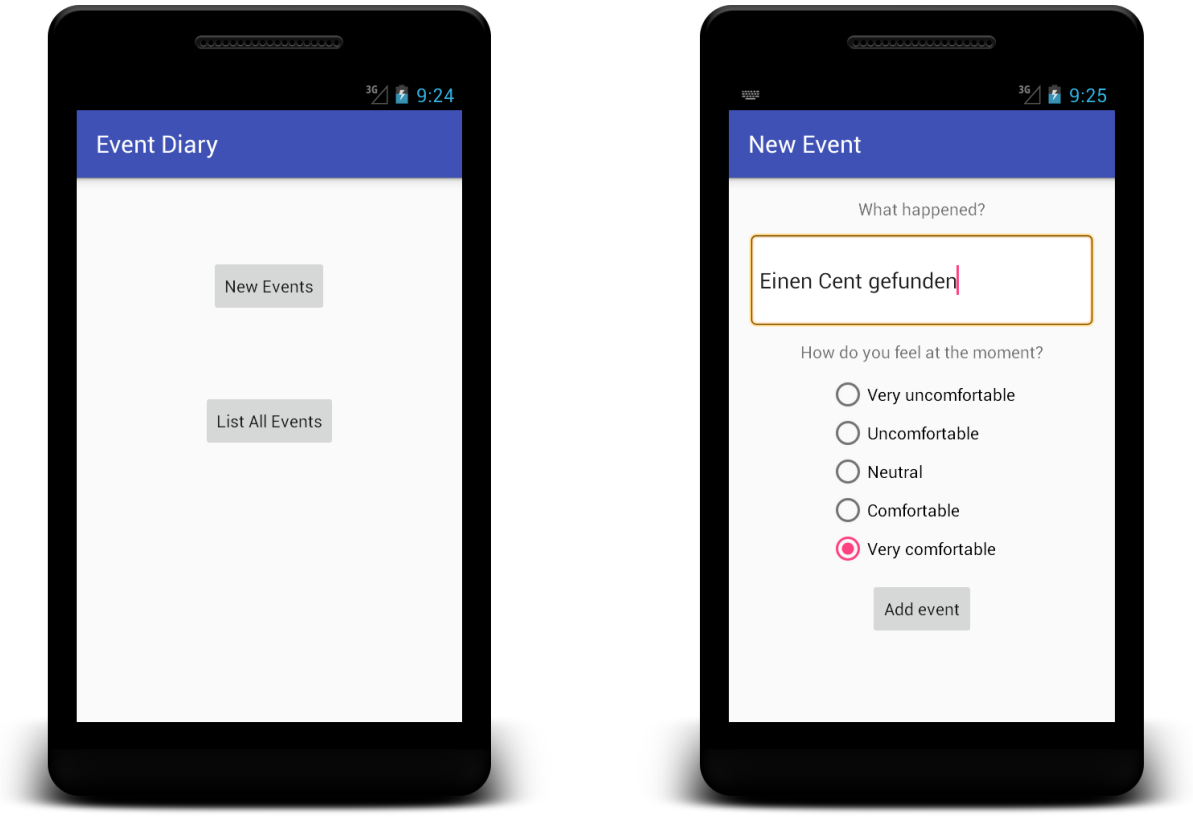
[1] Siehe http://touchlabblog.tumblr.com/post/24474750219/single-sqlite-connection

[2] Zu Details siehe den folgenden, am 15.09.2015 abgerufenen Beitrag von Alex Lockwood:
http://www.androiddesignpatterns.com/2012/05/correctly-managing-your-sqlite-database.html

  - **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.

12.4 Ereignistagebuch anlegen und befüllen

Wir erstellen 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 Künneth (2012, Kapitel 11) eingeflossen.

Um eine neue, leere Aktivität in ein Projekt einzufügen, wählt man aus dem Kontextmenü zum Paket mit den Klassen des Projekts (z.B. `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>
```

12.4.1 SQLiteOpenHelper für die Kreation und Aktualisierung der Datenbank

Für die Kreation und Aktualisierung der SQLite-Datenbank **eventdiary.db** soll im Programm ein Objekt der von **SQLiteOpenHelper** abstammenden Klasse `EventDiaryOpenHelper` zuständig sein. Um diese Klasse im Projekt anzulegen, wählen wir aus dem Kontextmenü zum Paket mit den Klassen des Projekts (z.B. `de.zimkand.eventdiary`) den Eintrag

New > Java Class

In statischen **String**-Feldern der Klasse `EventDiaryOpenHelper` wird das Datenbankschema hinterlegt:

```
public static final String TABLE_NAME_EVENTS = "events";
public static final String ROWID = "_id";
public static final String COL_NAME_DATETIME = "datetime";
public static final String COL_NAME_EVENT = "event";
public static final String COL_NAME_MOOD = "mood";
```

Diese Felder sind als **public** deklariert, damit die Schemainformationen von beliebigen Klassen der Anwendung verwendet werden können. Der generell sinnvolle Vorschlag, zur allgemein zugänglichen Aufbewahrung der Schemainformationen eine spezielle Klasse anzulegen, die zudem für jede Datenbanktabelle eine eigene interne Klasse besitzt, wird im Beispiel mangels realem Nutzen *nicht* umgesetzt.[1]

Zwei private statische **String**-Felder der Klasse `EventDiaryOpenHelper` enthalten die SQL-Kommandos zum Erstellen bzw. Löschen der Datenbanktabelle mit den zu sammelnden Benutzererlebnissen. Diese Informationen werden in den `EventDiaryOpenHelper`-Methoden **onCreate()** bzw. **onUpgrade()** benötigt, die bei Bedarf von Android für die Erstellung bzw. Aktualisierung der Datenbank verwendet werden. Dies geschieht beim ersten Aufruf der **SQLiteOpenHelper**-Methoden **getWritableDatabase()** bzw. **getReadableDatabase()**, die ein Objekt der Klasse **SQLiteDatabase** abliefern.

Hier sind die wesentlichen Bestandteile im `EventDiaryOpenHelper`-Quellcode zu sehen:

```
package de.zimkand.eventdiary;

import android.content.ContentValues;
  .  .  .
import android.widget.Toast;

public class EventDiaryOpenHelper extends SQLiteOpenHelper {

    private static Context context;
    private static final String DB_NAME = "eventdiary.db";
    private static final int DB_VERSION = 1;

    private static EventDiaryOpenHelper instance;
```

[1] Siehe z.B. https://developer.android.com/training/basics/data-storage/databases.html

```
    public static final String TABLE_NAME_EVENTS = "events";
    public static final String ROWID = "_id";
    public static final String COL_NAME_DATETIME = "datetime";
    public static final String COL_NAME_EVENT = "event";
    public static final String COL_NAME_MOOD = "mood";

    private static final String CREATE_TABLE_EVENTS =
        "CREATE TABLE " + TABLE_NAME_EVENTS + " (" +
                ROWID + " INTEGER PRIMARY KEY AUTOINCREMENT, " +
                COL_NAME_DATETIME + " INTEGER NOT NULL, " +
                COL_NAME_EVENT + " TEXT NOT NULL, " +
                COL_NAME_MOOD + " INTEGER NOT NULL);";

    private static final String DROP_TABLE_EVENTS =
            "DROP TABLE IF EXISTS " + TABLE_NAME_EVENTS + ";";

    public static synchronized EventDiaryOpenHelper getHelper(Context parContext) {
        if (instance == null)
            instance = new EventDiaryOpenHelper(parContext);
        return instance;
    }

    private EventDiaryOpenHelper(Context parContext) {
        super(parContext, DB_NAME, null, DB_VERSION);
        context = parContext;
    }

    public void insert(String event, int mood, long datetime) {
        try {
            SQLiteDatabase db = getWritableDatabase();
            ContentValues values = new ContentValues();
            values.put(COL_NAME_DATETIME, datetime);
            values.put(COL_NAME_EVENT, event);
            values.put(COL_NAME_MOOD, mood);
            db.insertOrThrow(TABLE_NAME_EVENTS, null, values);
            Toast.makeText(context, R.string.inserted, Toast.LENGTH_SHORT).show();
        } catch (SQLiteException e) {
            Toast.makeText(context, R.string.insert_failed,
                           Toast.LENGTH_SHORT).show();
        }
    }

    public Cursor query() {
        SQLiteDatabase db = getReadableDatabase();
        return db.query(TABLE_NAME_EVENTS, null, null, null, null, null,
                        COL_NAME_DATETIME + " DESC");
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(CREATE_TABLE_EVENTS);
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        db.execSQL(DROP_TABLE_EVENTS); // Scherz!!
        onCreate(db);
    }
}
```

Um das Singleton-Muster zu realisieren, wird ein privates und statisches Feld vom Typ `EventDiaryOpenHelper` verwendet:

```
private static EventDiaryOpenHelper instance;
```

Der `EventDiaryOpenHelper`-Konstruktor ist aufgrund des Zugriffsmodifikator **private** nur klassenintern verfügbar:

```
private EventDiaryOpenHelper(Context parContext) {
    super(parContext, DB_NAME, null, DB_VERSION);
    context = parContext;
}
```

Es wird eine statische und öffentlich verfügbare Fabrikmethode angeboten, die das *eine* erwünschte `EventDiaryOpenHelper`-Objekt anlegt, wenn es noch nicht existiert, und ansonsten die Adresse des bereits vorhandenen Objekts liefert:

```
public static synchronized EventDiaryOpenHelper getHelper(Context parContext) {
    if (instance == null)
        instance = new EventDiaryOpenHelper(parContext);
    return instance;
}
```

Die Datenbankdatei wird nötigenfalls automatisch angelegt beim Versuch, per **getWritableDatabase()** eine Möglichkeit zum schreibenden Zugriff zu erlangen (siehe `EventDiaryOpenHelper`-Methode `insert()`). Erforderliche Informationen über die anzulegende Datei (z.B. Name und Version) erfährt das `EventDiaryOpenHelper`-Objekt per Konstruktor. Anschließend wird zum Initialisieren der Datenbank die `EventDiaryOpenHelper`-Methode **onCreate()** aufgerufen. Dort wird über die **SQLiteDatabase**-Methode **execSQL()** das SQL-Kommando **CREATE TABLE** ausgeführt. Beim Zusammensetzen von SQL-Kommandos per Plus-Operator ist darauf zu achten, dass die Leerzeichen relevant sind für die Einhaltung der Syntaxregeln.

Die Methode **onUpgrade()** ist im Beispiel nur provisorisch ausgeführt und darf auf keinen Fall so bleiben, weil bei jeder Änderung des Datenbankschemas eine vorhandene Datenbanktabelle gelöscht wird.

Für das Beispielprogramm hat die Erfahrung gezeigt, dass bei Ausführung von **onCreate()** sowie **insert()** (mit der beim ersten Aufruf potentiell zeitkritischen Methode **getWritableDatabase()**) im UI-Thread keine Beeinträchtigung feststellbar ist. Daher wird die generelle Empfehlung, Datenbankzugriffe mit Hilfe der Klassen **AsyncTask** (vgl. Abschnitt 9.3) oder **IntentService** (vgl. Abschnitt 10.1) in einem eigenen Thread asynchron auszuführen, *nicht* umgesetzt.

12.4.2 Datenbankmodifikation per SQLiteDatabase-Objekt

Das `EventDiaryOpenHelper`-Objekt wirkt nicht direkt auf die SQLite-Datenbank ein, sondern beauftragt dazu ein Objekt der Klasse **SQLiteDatabase**. Mit der Methode `insert()` kann das `EventDiaryOpenHelper`-Objekt beauftragt werden, eine neue Zeile in die Datenbanktabelle `events` zu schreiben, wozu mit der **SQLiteOpenHelper**-Methode **getWritableDatabase()** eine Referenz auf das fest mit dem `EventDiaryOpenHelper` assoziierte **SQLiteDatabase**-Objekt beschafft wird. Die Werte des neuen Falles für die Attribute `event`, `mood` und `datetime` landen zunächst in einem Objekt der Klasse **ContentValues**, um dann als dritter Parameter der **SQLiteDatabase**-Methode **insertOrThrow()** in die Datenbank geschrieben zu werden. Im ersten Parameter des **insertOrThrow()** - Kommandos

```
db.insertOrThrow(TABLE_NAME_EVENTS, null, values);
```

wird die Tabelle gewählt. Der zweite Parameter (namens *nullColumnHack*) ist nur in einer sehr speziellen Situation relevant und kann meist mit **null** versorgt werden.[1]

Wenn **getWritableDatabase()** bzw. **insertOrThrow()** auf einen Fehler bei der Datenbankbearbeitung stoßen, werfen sie ein Ausnahmeobjekt aus der Klasse **SQLException**. Daher findet ihr Aufruf im Rahmen einer **try-catch** - Anweisung statt.[2]

Um den Benutzer über das gelungene oder gescheiterte Einfügen einer neuen Tabellenzeile zu informieren, zeigt die `EventDiaryOpenHelper`-Methode `insert()` ein **Toast**-Meldungsfenster an, 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(context, R.string.inserted, Toast.LENGTH_SHORT).show();
```

12.4.3 Aktivität zum Erfassen neuer Ereignisse

Nun kommen wird zur Beschreibung der Klasse `NewEventActivity`, die das Erfassen von neuen Ereignissen erlaubt. In ihrer Lebenszyklusmethode **onCreate()** wird mit Hilfe der das Singleton-Muster realisierenden statischen `EventDiaryOpenHelper`-Fabrikmethode eine Referenz zum (neuen oder schon vorhandenen) `EventDiaryOpenHelper`-Objekt erworben:

```
dbConnection = EventDiaryOpenHelper.getHelper(this);
```

Dann wird eine anonyme Klasse als Klick-Handler für den Befehlsschalter definiert:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_new_event);

    dbConnection = EventDiaryOpenHelper.getHelper(this);
```

---

[1] Ein leeres **ContentValues**-Objekt verstößt gegen die SQL-Regel, dass beim Einfügen einer neuen Datenbankzeile mindestens eine Spalte einen Wert erhalten muss. Als Wert des Parameters *nullColumnHack* kann man den Namen einer Ersatzspalte mit erlaubtem **null**-Wert angeben. Erhält der **insert()** - Aufruf ein leeres **ContentValues**-Objekt, wird die Ersatzspalte explizit auf **null** gesetzt, so dass eine neue Tabellenzeile entsteht.

[2] Weil **SQLException** von **RuntimeException** abstammt, ist die Ausnahmebehandlung freiwillig. Zu Details bei der Ausnahmebehandlung siehe Baltes-Götz & Götz, 2016 (Kapitel 11).

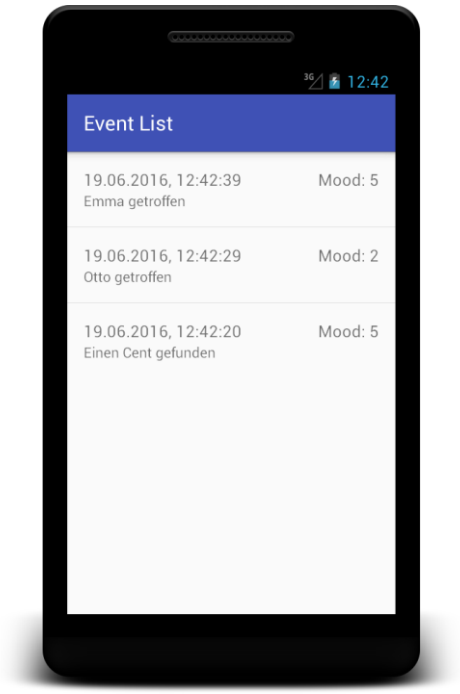
```
        Button btnAddEvent = (Button) findViewById(R.id.btnAddEvent);
        btnAddEvent.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                EditText event = (EditText)findViewById(R.id.etEvent);
                String s = event.getText().toString();
                RadioGroup radioGroup = (RadioGroup) findViewById(R.id.radioGroup);
                int rbId = radioGroup.getCheckedRadioButtonId();
                View rb = radioGroup.findViewById(rbId);
                int mood = radioGroup.indexOfChild(rb) + 1;
                if (s.length() > 0  &&  mood >= 1  &&  mood <= 5) {
                    dbConnection.insert(event.getText().toString(), mood,
                                        System.currentTimeMillis());
                    radioGroup.clearCheck();
                    event.setText("");
                } else
                    Toast.makeText(NewEventActivity.this, R.string.incomplete,
                                   Toast.LENGTH_SHORT).show();
            }
        });
    }
```

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.

12.5 Ereignisliste aus der Datenbank abrufen und per ListView anzeigen

Wir erstellen eine weitere Aktivität mit der Aufgabe, eine Liste mit allen bisher vom Benutzer erfassten Ereignissen anzuzeigen:



Dazu wählen wir aus dem Kontextmenü zum Paket mit den Klassen des Projekts (z.B. `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>
```

Prinzipiell eignet sich die Basisklasse **ListActivity**, die bereits etliche Kompetenzen bzw. Ausstattungen zur Präsentation einer Liste besitzt (z.B. ein **ListView**-Steuerelement). Allerdings besteht eine Besonderheit der Klasse **ListActivity** im Vergleich zu der bisher (aufgrund der Empfehlung des AS-Assistenten für neue Aktivitäten) verwendeten Basisklasse **AppCompatActivity** darin, dass *keine* Titelzeile erscheint. Daher verwenden wir die Basisklasse **AppCompatActivity** und erstellen eine simple Layout-Definition mit **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>
```

12.5.1 CursorAdapter

Für die Beschaffung der Daten zu einem **ListView**-Steuerelement sorgt ein Objekt aus einer Klasse, die das Interface **ListAdapter** (im Paket **android.widget**) erfüllt. In unserem Fall stammen die Daten aus einem **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 static final DateFormat FORMAT_DATE = DateFormat.getDateInstance();
    private static final DateFormat FORMAT_TIME = DateFormat.getTimeInstance();
    private LayoutInflater inflator;
    private int colDateTime = 1, colEventDesc = 2, colMood = 3;

    public EventDiaryAdapter(Context context, Cursor cursor) {
        super(context, cursor, CursorAdapter.FLAG_REGISTER_CONTENT_OBSERVER)
        inflator = LayoutInflater.from(context);
    }
```

```
    @Override
    public View newView(Context context, Cursor cursor, ViewGroup parent) {
        return inflator.inflate(R.layout.activity_event_entry, null);
    }

    @Override
    public void bindView(View view, Context context, Cursor cursor) {
        TextView tvDateTime = (TextView) view.findViewById(R.id.datetime);
        TextView tvMood = (TextView) view.findViewById(R.id.moodValue);
        TextView tvDesc = (TextView) view.findViewById(R.id.eventDesc);
        long timeMillis = cursor.getLong(colDateTime);
        date.setTime(timeMillis);
        tvDateTime.setText(FORMAT_DATE.format(date) + ", " +
                           FORMAT_TIME.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.[1] Anschließend 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:

```
<RelativeLayout 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: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.eventdiary.EventListActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/datetime"
        android:layout_alignParentLeft="true"
        android:textSize="16sp" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/moodValue"
        android:text="@string/moodvalue"
        android:layout_alignParentRight="true"
        android:textSize="16sp" />
```

[1] Siehe http://developer.android.com/reference/android/widget/CursorAdapter.html

```
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textAppearance="?android:attr/textAppearanceSmall"
            android:id="@+id/eventDesc"
            android:layout_below="@+id/datetime"
            android:layout_alignParentLeft="true" />

    </RelativeLayout>
```

- **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.

12.5.2 Aufgabenverteilung durch die Listenaktivität

In der **onCreate()** - Methode der Klasse `EventListActivity` werden folgende Objekte miteinander verbunden

- Das Objekt unserer **SQLiteOpenHelper**-Ableitung `EventDiaryOpenHelper`
  Gemäß Singleton-Muster gelangen wir über die statische Fabrikmethode `getHelper()` an eine Referenz auf das einzige `EventDiaryOpenHelper`-Objekt, das die Verbindung zur SQLite-Datenbank **eventdiary.db** herstellt. Es kennt das Datenbankschema und sorgt durch seine Methoden **onCreate()** und **onUpgrade()**, die bei Bedarf von Android aufgerufen werden, für die Erstellung bzw. Aktualisierung der Datenbank. Mit dem `EventDiaryOpenHelper`-Objekt ist ein **SQLiteDatabase**-Objekt fest assoziiert, das die Verbindung zur Datenbank herstellt und Methoden wie **query()**, **update()** und **insert()** beherrscht, die SQL-Kommandos kapseln.
- Ein Objekt der Klasse **Cursor**
  Als Rückgabe der **SQLiteDatabase**-Methode **query()**, die das SQL-Kommando **SELECT** kapselt, erhält man ein Objekt der Klasse **Cursor** (siehe Abschnitt 12.5.3). Es repräsentiert die Tabelle mit dem Abfrageergebnis und wird dem Adapter-Konstruktor per Parameter übergeben.
- Ein Objekt unserer **CursorAdapter**-Ableitung `EventDiaryAdapter`
  Dieses Objekt verbindet die Daten im **Cursor**-Objekt mit dem **ListView**-Steuerelement (siehe Abschnitt 12.5.1). Es wird bei der Konstruktion angewiesen, sich als Beobachter beim **Cursor**-Objekt registrieren zu lassen. Wenn der **CursorAdapter** von einer Veränderung der **Cursor**-Daten erfährt, fordert er das **ListView**-Objekt auf, die Anzeige zu aktualisieren.

Es folgt der erfreulich kurze Quellcode der Klasse `EventListActivity`:

```
package de.zimkand.eventdiary;

import android.database.Cursor;
  .  .  .
import android.widget.SimpleCursorAdapter;

public class EventListActivity extends AppCompatActivity {

    private EventDiaryOpenHelper dbHelper;
    private Cursor cursor;
    private EventDiaryAdapter adapter;
```

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_event_list);
        ListView lv = (ListView) findViewById(R.id.listView);
        dbHelper = EventDiaryOpenHelper.getHelper(this);
        cursor = dbHelper.query();
        startManagingCursor(cursor);
        adapter = new EventDiaryAdapter(this, cursor);
        lv.setAdapter(adapter);
    }
}
```

Durch den Methodenaufruf

```
startManagingCursor(cursor);
```

wird es der Aktivität überlassen, das **Cursor**-Objekt im Rahmen ihrer eigenen 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.

Mittlerweile sind die Methoden **startManagingCursor()**, **deactivate()** und **requery()** als abgewertet (engl.: *deprecated*) gekennzeichnet, sollten also nicht mehr verwendet werden. Weil sie Datenbankabfragen im UI-Thread zur Folge haben, kann die Bedienoberfläche behindert werden. 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 beiden genannten Probleme verfügbar. Weil diese Lösung einen Content Provider benötigt, wird sie erst in Abschnitt 13.3 vorgestellt.

12.5.3 Datenbankabfrage

In der Klasse `EventDiaryOpenHelper` ist die Methode `query()` nachzurüsten, die alle Zeilen der Tabelle `events` in absteigender chronologischer Sortierung abruft und in einem **Cursor**-Objekt abliefert:

```
public Cursor query() {
    SQLiteDatabase db = getReadableDatabase();
    return db.query(TABLE_NAME_EVENTS, null, null, null, null, null,
                    COL_NAME_DATETIME + " DESC");
}
```

Das geschieht durch Vermittlung der in Abschnitt 12.3.1 vorgestellten Klasse **SQLiteDataBase**, deren **query()** - Methode letztlich ein **SELECT**-Kommando an die SQLite-Datenbank übermittelt und die Ergebnistabelle als **Cursor**-Objekt liefert.

Neben dem Tabellennamen im ersten Parameter sind bei Verwendung der folgenden **query()** - Überladung

public Cursor query(String *table***, String[]** *projection***, String** *selection***,**
String[] *selectionArgs***, String** *groupBy***, String** *having***, String** *orderBy***)**

etliche weitere Parameter beteiligt, von denen im Beispiel wenig Gebrauch gemacht wird:

- **String[]** *projection*
  Im zweiten Parameter kann eine Liste mit den tatsächlich benötigten Spalten angegeben werden. Durch die folgende Variablendeklaration mit Initialisierung wird ein möglicher Aktualparameterwert hergestellt:
  ```
  String[] projection = {COL_NAME_ID, COL_NAME_DATETIME,
                         COL_NAME_EVENT, COL_NAME_MOOD};
  ```
  Der Primärschlüssel (Name in der Datenbank: **_id**) *muss* in die Abfrage einbezogen werden, wenn die Ergebnistabelle mit Hilfe der Klasse **CursorAdapter** an ein **ListView**-Steuerelement gebunden werden soll.[1] Im Beispiel werden die Spaltennamen aus Konstanten der Klasse `EventDiaryOpenHelper` gelesen. Setzt man den zweiten **query()** - Parameter auf **null**, dann werden *alle* Spalten der Tabelle einbezogen.
- **String** *selection*
  Über eine Bedingung kann die Menge der abgerufenen Zeilen eingeschränkt werden. Es ist die Syntax der SQL-Klausel **WHERE** zu verwenden, wobei das Schlüsselwort **WHERE** aber wegzulassen ist. Im folgenden Beispiel werden die Ereignisse mit einer vom Benutzer gewählten Stimmungslage angefordert:
  ```
  return db.query(TABLE_NAME_EVENTS, null, COL_NAME_MOOD+"=?",
                  new String[]{userInput}, null, null, null);
  ```
  Was an Stelle des ?-Platzhalters einzusetzen ist, wird im nächsten Parameter festgelegt. Man spricht hier von einer *parametrisierten Abfrage*. Setzt man den dritten Parameter auf **null**, dann werden *alle* Zeilen in der Tabelle abgerufen.
- **String[]** *selectionArgs*
  Es ist ein **String**-Array mit einer zur Liste der Fragezeichen im *selection*-Parameter korrespondierenden Liste von Zeichenfolgen anzugeben.
- **String** *groupBy*
  Mit der Syntax der SQL-Klausel **GROUP BY** gruppiert man die Tabelle nach den angegebenen Spalten (allgemein: Ausdrücken), wobei die Schlüsselwörter **GROUP BY** wegzulassen sind. Für jede Wertekombination der Gruppierungsvariablen entsteht (maximal) *eine* Ergebniszeile. Oft verwendet man die Gruppierung zusammen mit einer Auswertungsfunktion wie MAX oder MEAN und erhält z.B. für die Abteilungen eines Unternehmens (= Werte der Gruppierungsvariablen) jeweils das mittlere Gehalt.
- **String** *having*
  Für eine angeforderte Zeilengruppierung (siehe Parameter *groupBy*) kann festgelegt werden, welche Gruppen in der Ergebnismenge enthalten sein sollen. Es ist die Syntax der SQL-Klausel **HAVING** zu verwenden, wobei das Schlüsselwort **HAVING** aber wegzulassen ist.
- **String** *sortOrder*
  Mit der Syntax der SQL-Klausel **ORDER BY** kann die gewünschte Sortierung angefordert werden, zu verwenden, wobei die Schlüsselwörter **ORDER BY** wegzulassen sind, z.B.:
  ```
  return db.query(TABLE_NAME_EVENTS, null, COL_NAME_EVENT+" LIKE '%Otto%'",
                  null, null, null, COL_NAME_DATETIME + " DESC");
  ```

Es ist strikt zu vermeiden, bei einer Abfrage den Parameter *selection* nach dem folgenden Muster

```
return db.query(TABLE_NAME_EVENTS, null, COL_NAME_MOOD+"='"+userInput+"'",
                null, null, null, null);
```

unter Verwendung von *Benutzerreingaben* zu konstruieren, weil böswillige Eingaben durch eine so genannte **SQL-Injektion** zu einem illegitimen Informationstransfer führen können. Das oft beschriebene Angriffsszenario lässt sich auch mit unserer Ereignisdatenbank demonstrieren, wenngleich dabei die Realitätsnähe etwas verloren geht. Wir stellen uns vor, dass ein Benutzer nur die Ereignisse zu genau *einer* Stimmungslage sehen und den Wert (von 1 bis 5) wählen darf, was unge-

---

[1] Siehe http://developer.android.com/reference/android/widget/CursorAdapter.html

schickt per Texteingabe geschehen soll. Um die Beschränkung auf *eine* Stimmungslage aufzuheben, könnte ein gerissener Benutzer folgende Eintragung in das Textfeld für den Stimmungswert vornehmen:

```
' OR '1'='1
```

Insgesamt resultiert dann ungefähr das folgende **SELECT**-Kommando:

```
SELECT * FROM events WHERE mood='' OR '1'='1'
```

Wegen der Tautologie im eingeschmuggelten Teil des logischen Ausdrucks ist die **WHERE**-Bedingung immer wahr, und der Benutzer sieht die Ereignisse zu *allen* Stimmungswerten. In der Originalversion des Beispiels wird auf analoge Weise eine Passwortüberprüfung ausgehebelt (siehe z.B. Smith 2014).

Beim **query()** - Aufruf mit einer parametrisierten Anfrage werden die Benutzerbeiträge strikt als Daten behandelt und können keine SQL-Syntax einschleusen. Weil es keine Tabellenzeile mit dem Stimmungswert

```
' OR '1'='1
```

gibt, sieht der neugierige Benutzer eine *leere* Ereignisliste.

Ein erfolgreicher **query()** - Aufruf liefert ein **Cursor**-Objekt, das vor der ersten Zeile positioniert ist. Als spezielle **query()** - Ergebnisse sind möglich:

- Leere Ergebnismenge
  In Abhängigkeit vom Füllzustand der Tabelle und von der verwendeten **WHERE**-Klausel kann eine leere Ergebnismenge resultieren. Über die Anzahl der Zeilen im **Cursor**-Objekt informiert dessen Methode **getCount()**.
- Ausnahmefehler
  Enthält der **query()** - Aufruf z.B. einen fehlerhaften Tabellennamen, dann resultiert eine **android.database.sqlite.SQLiteException**.

Was über die **query()** - Parameter an SQLite übermittelt wird, sind letztlich die Bestandteile eines **SELECT**-Kommandos, und erfahrene SQL-Programmierer werden eventuell die Alternative **rawQuery()** bevorzugen, weil dabei das **SELECT**-Kommando nahezu in der gewohnten Form verwendet werden kann.

Das vollständige AS-Projekt auf dem aktuellen Stand befindet sich im Ordner

…\BspUeb\SQLite\EventDiary.1

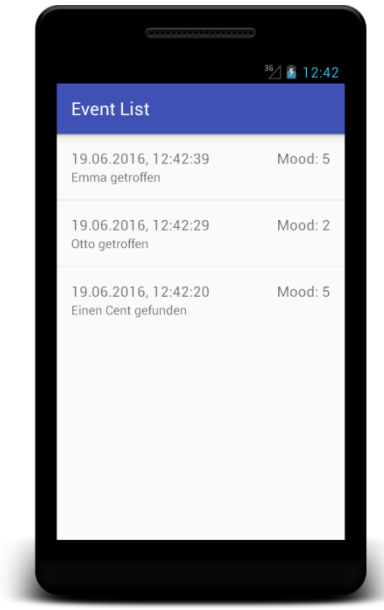
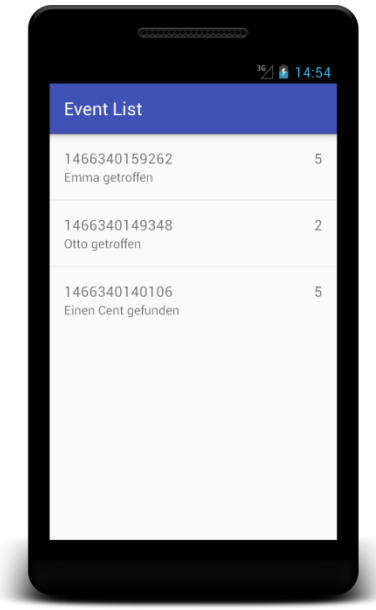
12.5.4 SimpleCursorAdapter

Bestehen die **ListView**-Elemente ausschließlich aus **TextView**-Objekten, dann kann man auf eine eigene **CursorAdapter**-Ableitung sowie das Implementieren der abstrakten Methoden **newView()** und **bindView()** verzichten und die Klasse **SimpleCursorAdapter** verwenden. Dem empfohlenen Konstruktor[1] übergibt man neben einem **Cursor**-Objekt und einer Layoutdefinition eine Liste mit den darzustellenden Tabellenspalten und eine korrespondierende Liste mit den zu versorgenden **ListView**-Objekten. Durch die folgende Variante der **onCreate()** - Methode aus unserer Klasse `EventListActivity` wird die Verwendung der Klasse **SimpleCursorAdapter** demonstriert:

[1] Siehe http://developer.android.com/reference/android/widget/SimpleCursorAdapter.html

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_event_list);
    ListView lv = (ListView) findViewById(R.id.listView);
    dbHelper = EventDiaryOpenHelper.getHelper(this);
    cursor = dbHelper.query();
    startManagingCursor(cursor);
    CursorAdapter sca = new SimpleCursorAdapter(this, R.layout.activity_event_entry,
            cursor,
            new String[] {EventDiaryOpenHelper.COL_NAME_DATETIME,
                    EventDiaryOpenHelper.COL_NAME_MOOD,
                    EventDiaryOpenHelper.COL_NAME_EVENT},
            new int[] {R.id.datetime, R.id.moodValue, R.id.eventDesc},
            0);
    lv.setAdapter(sca);
}
```

In unserem Beispiel bestehen die Listenelemente tatsächlich nur aus **TextView**-Objekten. Allerdings haben wir die implementierte **CursorAdapter**-Methode **bindView()** u.a. dazu genutzt, aus den in der Datenbank gespeicherten Millisekunden eine ansehnliche Datums- und Zeitanzeige zu erstellen:

CursorAdapter-Lösung (**bindView()** implementiert)	**SimpleCursorAdapter**-Lösung
	

12.6 Tabellenzeilen aktualisieren

Wir erweitern das seit Abschnitt 12.4 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 = (ListView) findViewById(R.id.listView);
lv. setOnItemClickListener(this);
```

In der Behandlungsmethode **onItemClick()**

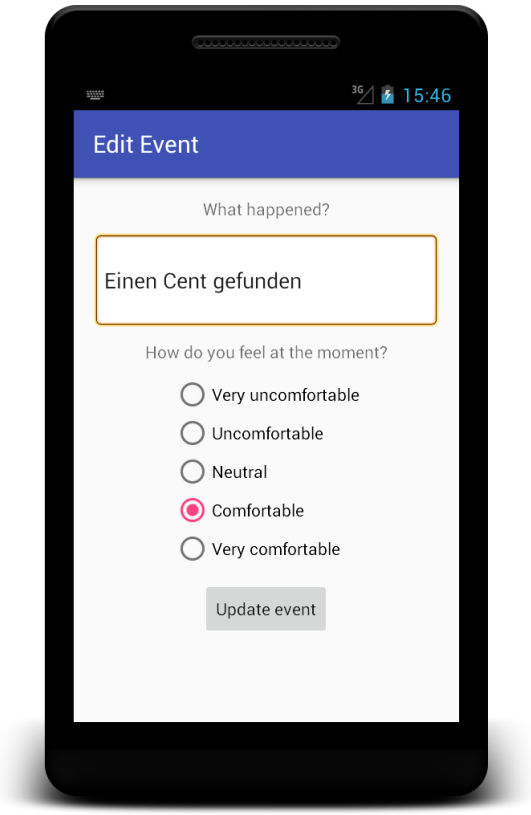
```
@Override
public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
    editEntry(id);
}
```

wird die private Methode `editEntry()` aufgerufen:

```
private void editEntry(long rowid) {
    Intent intent = new Intent(this, EditEventActivity.class);
    intent.putExtra("rowid", rowid);
    startActivity(intent);
}
```

Hier wird eine Aktivität zum Editieren eines Eintrags gestartet und per Extrainformation (vgl. Abschnitt 8.1.5) über die Nummer der zu bearbeitenden Tabellenzeile informiert.

Für die neue Funktionalität brauchen wir keine neue Aktivität, sondern können die vorhandene `NewEventActivity` (siehe Abschnitt 12.4) 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 `EditEvent-Activity`), die das Einfügen von neuen Ereignissen genauso beherrscht wie das Ändern von alten Ereignissen:

```
package de.zimkand.eventdiary;

import android.database.Cursor;
  .  .  .
import android.widget.Toast;

public class EditEventActivity extends AppCompatActivity {

    private boolean insertMode = true;
    private long rowid;
    private int colDateTime = 1, colEventDesc = 2, colMood = 3;
    private EventDiaryOpenHelper dbConnection;
```

```
    private Cursor cursor;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_edit_event);
        final RadioGroup radioGroup = (RadioGroup) findViewById(R.id.radioGroup);
        final EditText event = (EditText)findViewById(R.id.etEvent);
        Button btnEvent = (Button) findViewById(R.id.btnEvent);
        dbConnection = EventDiaryOpenHelper.getHelper(this);

        Bundle extras = getIntent().getExtras();
        if (extras != null) {
            insertMode = false;
            rowid = extras.getLong("rowid");
            cursor = dbConnection.query(rowid);
            if (!cursor.moveToFirst())
                this.finish();
            event.setText(cursor.getString(colEventDesc));
            int mood = cursor.getInt(colMood);
            ((RadioButton) radioGroup.getChildAt(mood - 1)).setChecked(true);
            btnEvent.setText(getString(R.string.update_event));
        } else
            this.setTitle(getString(R.string.title_activity_new_event));

        btnEvent.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                String s = event.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) {
                        dbConnection.insert(event.getText().toString(), mood,
                                            System.currentTimeMillis());
                        radioGroup.clearCheck();
                        event.setText("");
                    } else
                        dbConnection.update(rowid, event.getText().toString(), mood);
                } else
                    Toast.makeText(EditEventActivity.this, R.string.incomplete,
                                   Toast.LENGTH_SHORT).show();
            }
        });
    }
}
```

Im Änderungsmodus werden die initial anzuzeigenden Daten aus der Datenbank gelesen. Zunächst wird die Zeilennummer aus dem erhaltenen **Intent**-Objekt entnommen:

```
Bundle extras = getIntent().getExtras();
rowid = extras.getLong("rowid");
```

Dann wird das `EventDiaryOpenHelper`-Objekt per `query()` - Aufruf gebeten, die Daten der Ausgangszeile in ein **Cursor**-Objekt zu übertragen:

```
cursor = dbConnection.query(rowid);
if (!cursor.moveToFirst())
    this.finish();
event.setText(cursor.getString(colEventDesc));
int mood = cursor.getInt(colMood);
cursor.close();
((RadioButton) radioGroup.getChildAt(mood - 1)).setChecked(true);
btnEvent.setText(getString(R.string.update_event));
```

Das von `query()` gelieferte **Cursor**-Objekt ist *vor* dem ersten Eintrag positioniert. Bevor die Spalten der hoffentlich im **Cursor**-Objekt enthaltenen Zeile per **getString()** und **getInt()** ausgelesen

werden können, muss der Zeiger zum ersten Eintrag bewegt werden. Wenn dies scheitert, wird die Aktivität beendet. Nachdem die Ereignisdetails aus dem **Cursor**-Objekt entnommen worden sind, wird es geschlossen.

Die `EventDiaryOpenHelper`-Methode `query()` wird um einen Parameter vom Typ **long** erweitert, um sie anweisen zu können, entweder *alle* Zeilen der Ereignistabelle (Parameterwert < 0) oder eine bestimmte Zeile zu beschaffen:

```
public Cursor query(long rowid) {
    SQLiteDatabase db = getReadableDatabase();
    if (rowid < 0)
        return db.query(TABLE_NAME_EVENTS, null, null, null,
                        null, null, COL_NAME_DATETIME + " DESC");
    else
        return db.query(TABLE_NAME_EVENTS, null, "_id=?",
                        new String[]{Long.toString(rowid)}, null, null, null);
}
```

Man hätte prinzipiell die Ausgangsdaten für die Aktualisierungsaktivität über **Intent**-Extras von der Listenaktivität übernehmen und dabei einen Datenbankzugriff einsparen können. Allerdings hätte das lästigen Programmieraufwand verursacht. Das **ListView**-Objekt enthält nicht die Datenbankfelder selbst, sondern daraus abgeleitete Zeichenfolgen erhalten, z.B. die Zeichenfolge „Mood: 3“ statt des **int**-Werts 3.

Zur Übertragung der geänderten Daten in die zugehörige Tabellenzeile rufen wir die neu zu erstellende `EventDiaryOpenHelper`-Methode `update()` auf:

```
dbConnection.update(rowid, event.getText().toString(), mood);
```

In der `update()` - Implementierung

```
public void update(long rowid, String event, int mood) {
    try {
        SQLiteDatabase db = getWritableDatabase();
        ContentValues values = new ContentValues();
        values.put(COL_NAME_EVENT, event);
        values.put(COL_NAME_MOOD, mood);
        db.update(TABLE_NAME_EVENTS, values, "_id=?",
                  new String[]{Long.toString(rowid)});
        Toast.makeText(context, R.string.updated, Toast.LENGTH_SHORT).show();
    } catch (SQLiteException e) {
        Toast.makeText(context, R.string.update_failed, Toast.LENGTH_SHORT).show();
    }
}
```

wird ein **ContentValues** - Objekt mit den zu ändernden Werten gefüllt. Für den eigentlichen Datenbankzugriff ist die **update()** - Methode der Klasse **SQLiteDatabase** zuständig. Sie erhält …

- im ersten Parameter den Namen der Tabelle
- im zweiten Parameter den **ContentValues** -Objekt
- im dritten Parameter eine **WHERE**-Klausel mit Platzhaltern, wobei das Schlüsselwort **WHERE** selbst entfällt
- im vierten Parameter einen **String**-Array mit einer zur Platzhalterliste korrespondierenden Liste von Argumenten

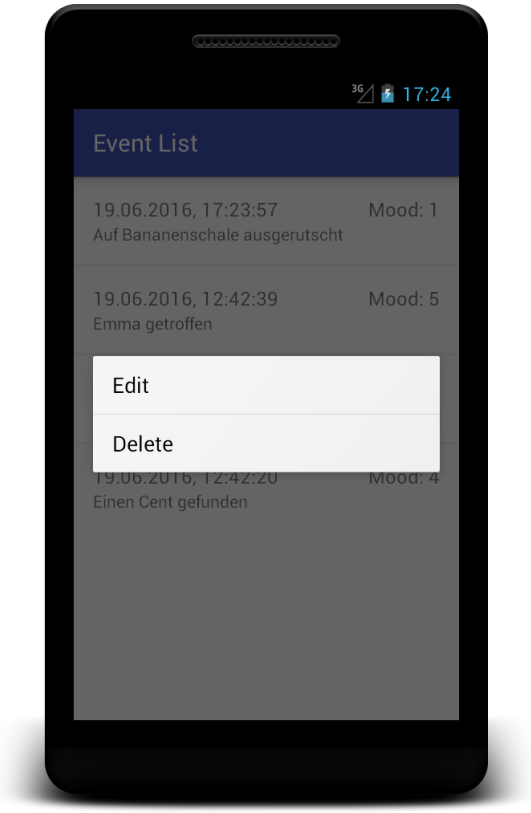
In unserem Beispiel enthält die **WHERE**-Klausel **`"_id=?"`** nur einen Platzhalter für der Wert des Primärschlüssels, und im letzten **update()** - Parameter wird etwas mühsam ein passender **String**-Array konstruiert.

Außer den Arbeiten am Quellcode der Klassen `EditEventActivity` und `EventDiary-OpenHelper` sind noch einige Erweiterungen in der Ressourcendatei **strings.xml** vorzunehmen, um den Projektzustand im folgenden Ordner zu erreichen:

…\BspUeb\SQLite\EventDiary.2

12.7 Tabellenzeilen per Kontextmenü löschen

Unser Ereignistagebuch soll dem Benutzer auch die Möglichkeit bieten, Einträge zu löschen. Zur Realisation versorgen in der `EvenListActivity` das **ListView**-Steuerelement mit einem flottierenden Kontextmenü (vgl. Abschnitt 7.6.3.1), das durch einen Langklick auf das Steuerelement hervorgelockt werden kann:



Um in einer Aktivität oder in einem Fragment für ein Steuerelement ein flottierendes Kontextmenü zu realisieren, sind folgende Schritte erforderlich (vgl. Abschnitt 7.6.3.1):

- Das Steuerelement durch einen Aufruf der **Activity**- bzw. **Fragment**-Methode **registerForContextMenu()** für die Anzeige eines Kontextmenüs registrieren, z.B.:
  ```
  ListView lv = (ListView) findViewById(R.id.listView);
  registerForContextMenu(lv);
  ```
- Die **Activity**- bzw. **Fragment**-Methode **onCreateContextMenu()** 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()** 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.ContextMenuInfo contextMenuInfo) {
    super.onCreateContextMenu(contMenu, v, contextMenuInfo);
    getMenuInflater().inflate(R.menu.event_list_context_menu, contMenu);
}
```

Es folgt die im Projektordner **/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>
```

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();
    switch (item.getItemId()) {
        case R.id.evliconit_edit:
            editEntry(acmi.id);
            return true;
        case R.id.evliconit_del:
            dbHelper.delete(acmi.id);
            cursor.requery();
            return true;
        default:
            return super.onContextItemSelected(item);
    }
}
```

Hier sind zu ermitteln:

- Indexwert der Tabellenzeile 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 etwas Hintergrundwissen. Das Parameterobjekt liefert über die Methode **getMenuInfo()** ein Objekt aus der Klasse **AdapterView.AdapterContext-MenuInfo**, wobei eine explizite Umwandlung in diesen Typ erforderlich ist. Dieses Objekt bewahrt in der öffentlichen Instanzvariablen **id** den Indexwert der vom Kontextmenü betroffenen Tabellenzeile auf.

Ist eine Tabellenzeile zu löschen, wird die neu zu erstellende `EventDiaryOpenHelper`-Methode `delete()` mit dem Indexwert als Parameter aufgerufen. Die folgende Methode

```
public void delete(long rowid) {
    try {
        SQLiteDatabase db = getWritableDatabase();
        db.delete(TABLE_NAME_EVENTS, "_id=?", new String[]{Long.toString(rowid)});
        Toast.makeText(context, R.string.deleted, Toast.LENGTH_SHORT).show();
    } catch (SQLiteException e) {
        Log.i(TAG, "SQLiteException "+ e.getMessage());
        Toast.makeText(context, R.string.delete_failed, Toast.LENGTH_SHORT).show();
    }
}
```

verwendet für den eigentlichen Datenbankzugriff die **delete()** - Methode der Klasse **SQLiteDatabase**. Diese erhält ...

- im ersten Parameter den Namen der Tabelle
- im zweiten Parameter eine **WHERE**-Klausel mit ?-Platzhalterzeichen (ohne das Schlüsselwort **WHERE**)
- im dritten Parameter einen **String**-Array mit einer zur Platzhalterliste korrespondierenden Liste von Argumenten

In unserem Beispiel enthält die **WHERE**-Klausel "`_id=?`" nur *einen* Platzhalter für den Wert des Primärschlüssels, und im dritten Parameter wird in passender **String**-Array konstruiert.

In der Methode **onContextItemSelected()** wird das **Cursor**-Objekt nach dem Löschen der Tabellenzeile per **requery()** aufgefordert, seinen Datenbestand zu aktualisieren:

```
case R.id.evliconit_del:
    dbHelper.delete(acmi.id);
    cursor.requery();
    return true;
```

Bei einer großen Datenbanktabelle sollten derartige Operationen *nicht* im UI-Thread ausgeführt werden. Mit Hilfe der Klasse **CursorLoader** gelingt eine überzeugende Lösung für Laden der aktualisierten Daten. Weil dabei ein Content Provider ins Spiel kommt, wird die Lösung erst in Kapitel 13 vorgestellt.

Weil die `EventListActivity` in ihrer **onCreate()** - Methode per **startManagingCursor()** aufgefordert worden ist, sich im Zusammenhang mit kritischen Lebensereignissen um das **Cursor**-Objekt zu kümmern, findet ein **requery()** - Aufruf auch dann statt, wenn die Aktivität aus dem gestoppten Zustand zurückkehrt.

Weil der `EventDiaryAdapter` bei der Konstruktion beauftragt worden ist, sich beim Cursor-Objekt als Beobachter registrieren zu lassen (`CursorAdapter.FLAG_REGISTER_CONTENT_OBSERVER`) führt ein **requery()** schließlich zur Anzeige der aktuellen Ereignisliste.

Am Ende eines **case**-Blocks muss die **switch**-Anweisung verlassen werden, damit es nicht zum „Durchfall“ in den nächsten **case**-Block kommt. Im Fall der Methode **onContextItemSelected()** bietet es sich an, am Ende eines **case**-Blocks per **return** die Methode zu verlassen. Dabei ist zu beachten, dass die Methode den Rückgabetyp **boolean** besitzt. Mit dem Rückgabewert **true** wird signalisiert, dass keine weitere Reaktion auf das Kontextmenüitem erfolgen soll (z.B. in Bezug auf ein übergeordnetes Steuerelement). Mit dem Rückgabewert **false** wird signalisiert, dass die Verarbeitung der Kontextmenüwahl entlang der Verarbeitungskette fortgesetzt werden soll (vgl. Abschnitt 7.4.3).

Das vollständige AS-Projekt auf dem aktuellen Stand befindet sich im Ordner

…\BspUeb\SQLite\EventDiary.3

13 Content Provider nutzen und anbieten

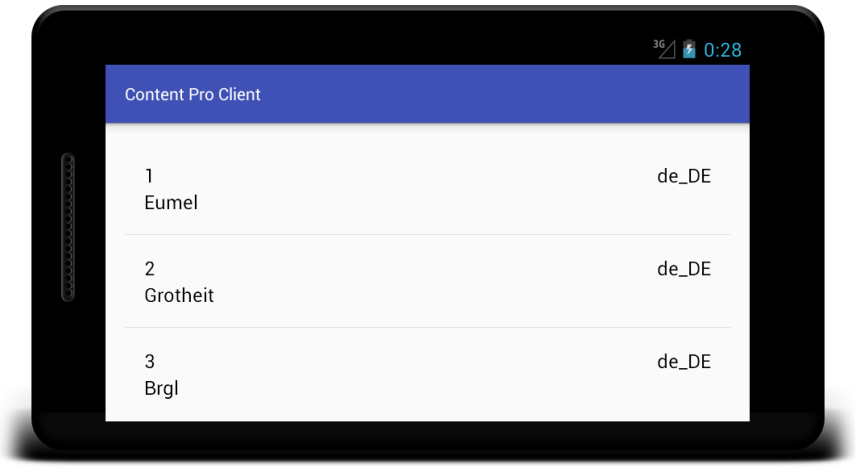
Um fremden Anwendungen den Zugriff auf eigene Daten zu ermöglichen, implementiert man eine als *Content Provider* bezeichnete Anwendungskomponente. Diese stellt den Klienten meist eine oder mehrere Tabellen mit strukturierten Daten zur Verfügung. Ob diese Daten in einer Datenbank abgelegt sind, oder aus einer alternativen Quelle stammen, ist ein für Klienten irrelevantes Implementierungsdetail. Neben strukturierten Daten (Abfrageergebnissen) können auch Dateiinhalte veröffentlicht werden (z.B. eine Fotosammlung).

Selbstverständlich können wir auch in eigenen Anwendungen die Content Provider fremder Anwendungen nutzen, wobei das Betriebssystem selbst einige Datenbestände anbietet (z.B. Kontakte, Termine).

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.

13.1 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 Adressen und Termine). In diesem Abschnitt übernehmen wir ein Beispiel aus der Android-Online-Dokumentation und greifen auf das von Android verwaltete Wörterbuch mit den benutzereigenen Erweiterungen des Standardwörterbuchs zu.[1] Auf einem Smartphone im Querformat sieht das Beispielprogramm so aus:

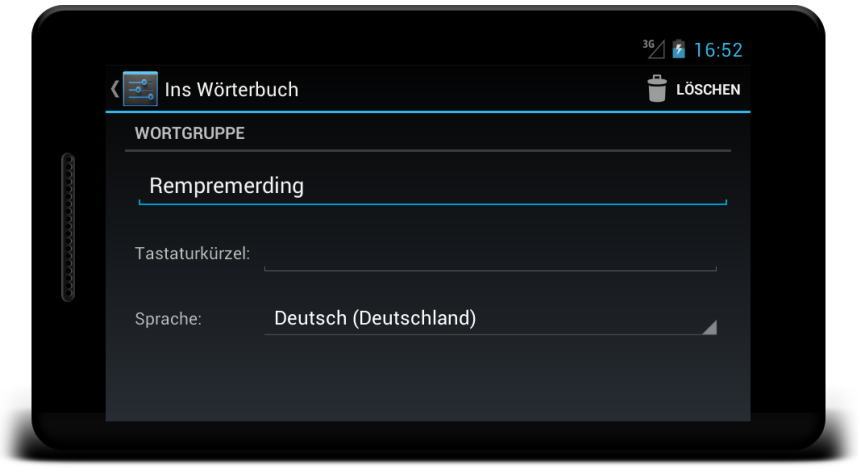


Ist das Benutzerwörterbuch auf dem zum Testen verwendeten Android-Gerät noch leer, kann es über die folgende Befehlssequenz

Einstellungen > Sprache und Eingabe > Mein Wörterbuch

gefüllt werden, z.B.:

[1] Hier ist das Beispiels mit vielen weiteren Details zu finden (abgerufen am 19.07.2015): http://developer.android.com/guide/topics/providers/content-provider-basics.html



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 13.3 wird demonstriert, wie man asynchrone Provider-Zugriffe durchführt, die in einem Hintergrund-Thread ablaufen und den UI-Thread *nicht* belasten.

13.1.1 Content-URI

Zur Beschreibung der bei einem Content Provider angeforderten Daten wird ein **Uri**-Objekt mit einem **content**-Schema verwendet. Um z.B. beim Provider für das benutzerspezifische Wörterbuch eines Android-Geräts den dritten Eintrag anzufordern, ist ein auf der folgenden Zeichenfolge basierendes **Uri**-Objekt zu verwenden:

Aufbau:	Schema	Provider	Pfad	ID
Beispiel:	`content://`	`user_dictionary/`	`words/`	`3`

Man kann das benötigte **Uri**-Objekt über die statische **Uri**-Methode **parse()** aus einer Zeichenfolge erstellen, z.B.:

```
Uri uri = Uri.parse("content://user_dictionary/words");
```

Sinnvoller ist die Verwendung einer vom Entwickler des Content Providers veröffentlichten Zeichenfolgevariablen, z.B.:

```
Uri uri = UserDictionary.Words.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(UserDictionary.Words.CONTENT_URI, 4);
```

13.1.2 Zugriff via ContentResolver

Zugriffe auf einen Content Provider zeigen eine große Ähnlichkeit mit den in Kapitel 12 beschriebenen Datenbankzugriffen. Als Neuerung kommt ein Objekt der Klasse **ContentResolver** ins Spiel.

Außerdem sind Zugriffsrechte relevant (siehe Abschnitte 13.1.3 und 13.2.4):

- Der Provider kann in seiner Manifestdatei für das Lesen und Schreiben seiner Daten eine Berechtigung definieren.
- Der Klient muss diese Rechte in seiner Manifestdatei deklarieren, und der Benutzer muss vor der Installation zustimmen.

13.1.2.1 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 12.3.3). Neben dem **Uri**-Objekt im ersten Parameter (siehe Abschnitt 13.1.1) 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 den Wörtern auch die eindeutige Kennung und die zugehörige Sprach/Region - Kombination angefordert werden:
  ```
  String[] projection = {UserDictionary.Words._ID,
                         UserDictionary.Words.LOCALE,
                         UserDictionary.Words.WORD};
  ```
  Die Kennung bzw. der Primärschlüssel (mit der Spaltenbezeichnung **_ID**) muss in die Abfrage einbezogen werden, wenn die Ergebnistabelle mit Hilfe der Klasse **CursorAdapter** an ein **ListView**-Steuerelement gebunden werden soll.[1] Setzt man den zweiten Parameter auf **null**, dann werden *alle* Spalten in der Tabelle des Content Providers abgerufen. Die Spaltennamen sind der Provider-Dokumentation zu entnehmen.
- **String** *selection*
  Über eine Bedingung kann die Menge der abgerufenen Zeilen eingeschränkt werden. Es ist die Syntax der SQL-Klausel **WHERE** zu verwenden, wobei das Schlüsselwort **WHERE** selbst aber wegzulassen ist. Im folgenden Beispiel werden die Wörter mit einer vom Benutzer gewählten Startzeichenfolge angefordert:
  ```
  cursor = getContentResolver().query(uri, projection,
     UserDictionary.Words.WORD+" LIKE ?", new String[] {userInput+"%"}, 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. Im obigen Beispiel wird das SQL-Platzhaltersymbol **%** verwendet, um alle Wörter mit einer bestimmten Startzeichenfolge anzufordern.

---

[1] Siehe http://developer.android.com/reference/android/widget/CursorAdapter.html (abgerufen am 19.07.2015)

- **String** *sortOrder*
  Mit der Syntax der SQL-Klausel **ORDER BY** kann die gewünschte Sortierung angefordert werden, wobei die Schlüsselwörter **ORDER BY** wegzulassen sind, z.B.:
  ```
  UserDictionary.Words.WORD+" DESC"
  ```

Mit der SQL-Injektion und ihrer Vermeidung durch die strikte Anwendung von parametrisierten Abfragen haben wir uns schon in Abschnitt 12.5.3 beschäftigt.

Ein erfolgreicher **query()** - Aufruf liefert ein **Cursor**-Objekt, das *vor* der ersten Zeile der Ergebnistabelle positioniert ist. Als spezielle **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**
  Enthält z.B. das **Uri**-Objekt eine fehlerhafte Schemaangabe (z.B. `contents`), dann resultiert der Rückgabewert **null**.
- Ausnahmefehler
  Enthält z.B. das **Uri**-Objekt einen fehlerhaften Tabellennamen, dann resultiert eine **IllegalArgumentException**.

Eine typische Verwendung der Ergebnistabelle in einem **Cursor**-Objekt besteht in der Bindung an ein **ListView**-Steuerelement, wobei oft ein Objekt der Klasse **SimpleCursorAdapter** (siehe Abschnitt 12.5.4) als Vermittler zum Einsatz kommt, z.B.:

```
int[] items = {R.id.id, R.id.locale, R.id.word};
SimpleCursorAdapter adapter = new SimpleCursorAdapter(getApplicationContext(),
            R.layout.list_view_element, cursor, projection, items, 0);
ListView lv = (ListView) findViewById(R.id.listView);
lv.setAdapter(adapter);
```

Im vorletzten Parameter des verwendeten **SimpleCursorAdapter** - Konstruktors ist eine zur Projektion analog aufgebaute Liste von **TextView**-Steuerelementen anzugeben.

Anschließend ist der eben abschnittsweise beschriebene Quellcode für eine sehr simple App, die eine Liste mit den Einträgen im Benutzerwörterbuch präsentiert, im Überblick zu sehen:

```
package de.zimkand.contentproclient;

import android.content.ContentValues;
  .  .  .
import android.widget.Toast;

public class MainActivity extends AppCampatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Uri uri = UserDictionary.Words.CONTENT_URI;
        String[] projection = {UserDictionary.Words._ID,
                               UserDictionary.Words.LOCALE,
                               UserDictionary.Words.WORD};

        Cursor cursor = getContentResolver().query(uri, projection, null, null, null);
        startManagingCursor(cursor);
```

```
            int[] tvIds = {R.id.id, R.id.locale, R.id.word};
            SimpleCursorAdapter adapter = new SimpleCursorAdapter(getApplicationContext(),
                R.layout.list_view_element, cursor, projection, tvIds, 0);
            ListView lv = (ListView) findViewById(R.id.listView);
            lv.setAdapter(adapter);
        }
    }
```

Für die im Beispiel verwendete, aber von Google als abgewertet deklarierte Methode **startManagingCursor()**, welche der Aktivität die Verwaltung des **Cursor**-Objekts (inkl. Schließen) überträgt (vgl. Abschnitt 12.5.2), wird in Abschnitt 13.3 ein sinnvoller Ersatz vorgestellt.

Statt die **Cursor**-Daten per Adapter an ein **ListView**-Steuerelement zu binden, können sie auch extrahiert und anderweitig genutzt werden. Während im Wörterbuchbeispiel alle Spalten des vom Provider bezogenen **Cursor**-Objekts den Datentyp **String** haben, können im Allgemeinen z.B. auch die anderen von SQLite unterstützten Datentypen auftreten (siehe Abschnitt 12.3.3). Um die Datentypen in einer Provider-Tabelle zu ermitteln, kann man die Dokumentation konsultieren (siehe Abschnitt 13.1.5) 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 Benutzerwörterbuch-Provider führt die Anfrage

```
getContentResolver().getType(UserDictionary.Words.CONTENT_URI)
```

zur Auskunft:

```
vnd.android.cursor.dir/vnd.google.userword
```

13.1.2.2 Modifikationen

Für modifizierende Zugriffe auf einen **ContentProvider** stellt ein **ContentResolver** die Methoden **insert()**, **update()** und **delete()** zur Verfügung. Diese sind weitgehend analog zu den korrespondierenden **SQLiteDatabase**-Methoden für modifizierende Zugriffe auf Datenbanken zu verwenden (siehe Abschnitte 12.4, 12.6 und 12.7).

Um das Benutzerwörterbuch um einen neuen Eintrag zu erweitern, erstellen wir ein **ContentValues**-Objekt und versorgen durch **put()** - Aufrufe die Spalten der **Words**-Tabelle mit passenden Werten. Schließlich werden das **Uri**-Objekt zum **ContentProvider** und das fertige **ContentValues**-Objekt als Parameter an die **ContentResolver**-Methode **insert()** übergeben:

```
Uri uri = UserDictionary.Words.CONTENT_URI;
ContentValues values = new ContentValues();
values.put(UserDictionary.Words.APP_ID, "ContentProClient");
values.put(UserDictionary.Words.LOCALE, "de_DE");
values.put(UserDictionary.Words.WORD, "Rempremerding");
values.put(UserDictionary.Words.FREQUENCY, "13");
Uri nUri = getContentResolver().insert(uri, values);
```

Als Rückgabe erhält man von **insert()** ein **Uri**-Objekt zum neuen Eintrag.

Um Einträge im Benutzerwörterbuch zu *ändern*, verwendet man wie beim Einfügen eines neuen Eintrags 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 13.1.2.1). Schließlich werden das **Uri**-

Objekt zum **ContentProvider**, das **ContentValues**-Objekt und die Auswahlparameter an die **ContentResolver**-Methode **update()** übergeben, z.B.:

```
ContentValues values = new ContentValues();
values.put(UserDictionary.Words.WORD, "Rempremmerding");
values.putNull(UserDictionary.Words.APP_ID);
String where = UserDictionary.Words.WORD + " LIKE ?";
String[] whereArgs = {"Rempremer%"};
int rowsChanged = getContentResolver().update(uri, values, where, whereArgs);
```

Als Rückgabe vom Typ **int** erhält man die Anzahl der geänderten Zeilen.

Um Einträge aus dem Benutzerwörterbuch zu *löschen*, verwendet man die **ContentResolver**-Methode **delete()** und definiert dabei die Menge der betroffenen Tabellenzeilen wie bei den **ContentResolver**-Methoden **query()** und **update()** über die Auswahlparameter *selection* und *selectionArgs*, z.B.:

```
String where = UserDictionary.Words.WORD + " LIKE ?";
String[] whereArgs = {"Remprem%"};
int rowsChanged = getContentResolver().delete(uri, where, whereArgs);
```

Als Rückgabe vom Typ **int** erhält man die Anzahl der gelöschten Zeilen.

13.1.3 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 das Benutzerwörterbuch über **uses-permission** -Elemente beantragen:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="de.zimkand.contentproclient" >

   <uses-permission android:name="android.permission.READ_USER_DICTIONARY" />
   <uses-permission android:name="android.permission.WRITE_USER_DICTIONARY" />

   <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 Klasse **Manifest.permission** (Paket: **android**) dokumentiert. Hier befinden sich z.B. die folgenden **String**-Konstanten:

- **public static final String READ_USER_DICTIONARY**
  Wert: **android.permission.READ_USER_DICTIONARY**
- **public static final String WRITE_USER_DICTIONARY**
  Wert: **android.permission.WRITE_USER_DICTIONARY**

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).

13.1.4 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:[1]

- 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 13.3.
- 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 Anwendungsinstanz gültiges) Zugriffsrecht für einen konkreten Content-URI erhalten und muss dabei folgendermaßen vorgehen:
  - Über die Methode **startActivityForResult()** mit passendem Intent-Parameter wird eine berechtigte Aktivität gestartet.
  - Diese liefert dem Antragsteller ein Ergebnis-Intent mit enthaltenem Content-URI, das einen speziellen Provider-Zugriff ermöglicht.
- 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.

13.1.5 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 13.1.4)

Zum Benutzerwörterbuch-Provider eines Android-Geräts, dessen Angebot sich auf die **Words**-Tabelle beschränkt, erfüllt die statische Mitgliedsklasse **UserDictionary.Words** diesen Zweck. In Abschnitt 13.1.1 haben wir aus dieser Klasse die Konstante **CONTENT_URI** als Referenz auf das **Uri**-Objekt zur **Words**-Tabelle verwendet:

```
Uri uri = UserDictionary.Words.CONTENT_URI;
```

Außerdem definiert die Klasse zu jeder Spalte der **Words**-Tabelle eine Konstante samt Dokumentationskommentar:

[1] http://developer.android.com/guide/topics/providers/content-provider-basics.html#AltForms

```
public static final String _ID = BaseColumns._ID;

/**
 * The word column.
 * <p>TYPE: TEXT</p>
 */
public static final String WORD = "word";

/**
 * The frequency column. A value between 1 and 255. Higher values imply higher frequency.
 * <p>TYPE: INTEGER</p>
 */
public static final String FREQUENCY = "frequency";

/**
 * The locale that this word belongs to. Null if it pertains to all
 * locales. Locale is as defined by the string returned by Locale.toString().
 * <p>TYPE: TEXT</p>
 */
public static final String LOCALE = "locale";

/**
 * The uid of the application that inserted the word.
 * <p>TYPE: INTEGER</p>
 */
public static final String APP_ID = "appid";

/**
 * An optional shortcut for this word. When the shortcut is typed, supporting IMEs should
 * suggest the word in this row as an alternate spelling too.
 */
public static final String SHORTCUT = "shortcut";
```

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 Domänennamen) und eine Bezeichnung des Inhalts (z.B. den Namen einer Datenbanktabelle) folgen zu lassen.[1] Für die **Uri**-Objekte zur **Words**-Tabelle des Benutzerwörterbuch-Providers definiert die Klasse **UserDictionary.Words** die folgenden MIME-Typen:

- URI **content://user_dictionary/words/**
  vnd.android.cursor.dir/vnd.google.userword
- URI **content://user_dictionary/words/1**
  vnd.android.cursor.item/vnd.google.userword

[1] http://developer.android.com/guide/topics/providers/content-provider-creating.html#TableMIMETypes

13.2 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 12). 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 besonders einfach realisieren (vgl. Abschnitt 13.3).

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 12.4 vorgestellten Ereignisdatenbank fungiert.

Wer weitere Informationen zur Erstellung von Content Providern benötigt, wird z.B. in der Online-Dokumentation zu Android fündig.[1]

13.2.1 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 13.1.1):

- 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:[2]
  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* (engl. *authority*) des URIs angegeben.
- 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`

---

[1] http://developer.android.com/guide/topics/providers/content-provider-creating.html

[2] Beim systemseitigen Content Provider **user_dictionary**, dem wir in Abschnitt 13.1.1 begegnet sind, wird diese Regel *nicht* beachtet.

Im Allgemeinen kann ein Provider *mehrere* Tabellen anbieten. Es dürfen Pfadnamen mit mehreren Segmenten verwendet werden, was der Übersicht dienen kann.

- Zeilennummer
  Zum Zugriff auf eine einzelne Tabellenzeile ist zusätzlich deren Nummer anzugeben, z.B.:
  ```
  content://de.zimkand.eventdiary.provider/events/7
  ```
  Diese wird per Konvention mit der meist als **_ID** bezeichneten Primärindexspalte der Tabelle abgeglichen.[1]

Bei der Konzeption eines Content Providers zur Präsentation von strukturierten Daten sind also festzulegen:

- Ein Name (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 die Anwendung zum Content Provider auch Intents behandeln, sind deren Aktionen, Extras und Flags zu definieren (vgl. Abschnitt 8.1).

Alle zur Verwendung eines Content Providers erforderlichen Informationen sollten in einer Kontrakt-Klasse definiert werden (vgl. Abschnitt 13.1.5).

13.2.2 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 13.1.2):[2]

- **public abstract boolean onCreate()**
  Diese Initialisierungsmethode wird von Android aufgerufen, sobald der Provider (z.B. aufgrund des ersten Zugriffs durch einen **ContentResolver**) erstellt worden ist. Weil die Methode im Main-Thread ausgeführt wird, darf sie keine aufwändigen Operationen durchführen.
- **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 13.1.2.1).
- **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 13.1.2.2).
- **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 13.1.2.2).
- **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 13.1.2.2).

---

[1] Damit ein als **Cursor**-Objekt von einem Content Provider bezogenes Abfrageergebnis in einem **ListView**-Steuerelement präsentiert werden kann, muss eine als **_ID** bezeichnete Spalte vorhanden sein.

[2] Zu jeder **ContentProvider**-Zugriffsmethode existiert in der Klasse **ContentResolver** ein Gegenstück mit identischer Signatur.

- **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. Sofern sich die veröffentlichten Daten in einer SQLite-Datenbank befinden, sind allerdings in der **ContentProvider**-Ableitung *keine* Maßnahmen zur Sicherung der Thread-Sicherheit erforderlich, solange nur ein *einziges* **SQLiteOpenHelper**-Objekt und damit ein einziges **SQLiteDatabase**-Objekt im Spiel sind. Dann kümmern sich SQLite und die Klasse **SQLiteDatabase** um die Thread-Sicherheit.[1]

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** transportiert werden muss, was nur bei den folgenden Ausnahmeklassen möglich ist:

- **IllegalArgumentException**
  Diese Ausnahmeklasse bietet sich z.B. an, wenn der Klient ein einen fehlerhaften **Uri**-Parameter sendet.
- **NullPointerException**
  Diese Ausnahmeklasse kommt in Frage, wenn ein Problem beim Datenbankzugriff aufgetreten ist.

Um einen Rohling für den Content Provider mit provisorischen Implementationen für alle abstrakten **ContentProvider**-Methoden in ein AS-Projekt einzufügen, wählen wir 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`.

13.2.2.1 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 per **switch**-Anweisung verarbeitet werden können. Weil pro **ContentProvider** nur *ein* Projekt dieser Klasse 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";
private static final UriMatcher uriMatcher;
static {
    uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
    // Typ 1 für den URI de.zimkand.eventdiary.provider/events
    uriMatcher.addURI(AUTHORITY, EventDiaryOpenHelper.DB_TABLE_EVENTS, 1);
    // Typ 2 für URIs, die eine Zeile der events-Tabelle ansprechen
    uriMatcher.addURI(AUTHORITY, EventDiaryOpenHelper.DB_TABLE_EVENTS +"/#", 2);
}
```

---

[1] Siehe den am 26.06.2016 abgerufenen Stack Overflow - Beitrag von Kevin Galligan auf der Seite:
http://stackoverflow.com/questions/2493331/what-are-the-best-practices-for-sqlite-on-android

Aufgrund dieser Initialisierung durch die beiden **addsURI()** - 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 **addsURI()** - Aufrufs wird Platzhalterzeichen **#** für Zahlen verwendet. Durch den Platzhalter ***** kann im *path*-Parameter eine beliebige Zeichenfolge vertreten werden.

In der `EventProvider`-Methode **query()** wird ein zur **UriMatcher**-Klassifikation passender Aufruf der **SQLiteDatabase**-Methode **query()** formuliert:

```
@Override
public Cursor query(Uri uri, String[] projection, String selection,
                    String[] selectionArgs, String sortOrder) {
    Cursor result;
    SQLiteDatabase db = dbHelper.getReadableDatabase();
    switch (uriMatcher.match(uri)) {
        case 1:
            result = db.query(EventDiaryOpenHelper.DB_TABLE_EVENTS, projection,
                              selection, selectionArgs, null, null, sortOrder);
            break;
        case 2:
            result = db.query(EventDiaryOpenHelper.DB_TABLE_EVENTS, projection,
              "_id=?", new String[]{uri.getLastPathSegment()}, null, null, null);
            break;
        default:
            throw new IllegalArgumentException("URI invalid");
    }
    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.

13.2.2.2 onCreate() implementieren

Die Methode **onCreate()**

public abstract boolean onCreate ()

kann sich darauf beschränken, eine Referenz auf das als Solist tätige Objekt aus unserer Klasse `EventDiaryOpenHelper` zu beschaffen (vgl. Abschnitt 12.4.1):

```
@Override
public boolean onCreate() {
    dbHelper = EventDiaryOpenHelper.getHelper(getContext());
    return true;
}
```

Durch das Singleton-Muster wird sichergestellt, dass genau *ein* Objekt der Klasse `EventDiaryOpenHelper` und damit genau *eine* Verbindung zu der SQLite-Datenbank existiert, sodass SQLite und die Klasse **SQLiteDatabase** für Thread-Sicherheit sorgen können.[1]

13.2.2.3 query() implementieren

In der **query()** - Implementation unseres Content Providers (siehe Abschnitt 13.2.2.1) wird ein zur **UriMatcher**-Klassifikation passender Aufruf der **SQLiteDatabase**-Methode **query()** formuliert. Als Rückgabe ist ein **Cursor**-Objekt zu liefern, das eventuell keine Zeilen enthält, also auf Befragen mit **getCount()** den Wert 0 meldet. Auf eine ungültige URI-Angabe sollte **query()** z.B. mit einer **IllegalArgumentException** reagieren

13.2.2.4 insert() implementieren

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 13.1.2.2), wobei für unversorgte Spalten im Datenbankschema ein Voreinstellungswert existieren sollte. Im folgenden Beispiel wird für den Datenbankzugriff die **SQLiteDatabase**-Methode **insertOrThrow()** verwendet:

```
@Override
public Uri insert(Uri uri, ContentValues values) {
    long newRow;
    if (uriMatcher.match(uri) == 1) {
        try {
            SQLiteDatabase db = dbHelper.getWritableDatabase();
            newRow = db.insertOrThrow(EventDiaryOpenHelper.DB_TABLE_EVENTS, null, values);
        } catch (NullPointerException exp) {
            throw new NullPointerException("Database Error");
        }
        Toast.makeText(getContext(), R.string.inserted, Toast.LENGTH_SHORT).show();
        return ContentUris.withAppendedId(uri, newRow);
    } else
        throw new IllegalArgumentException("URI invalid");
}
```

Auf ein ungeeignetes **Uri**-Objekt reagiert die Methode mit einer **IllegalArgumentException**. Schlägt der Datenbankzugriff fehl, wird eine **NullPointerException** geworfen. Wir oben erläutert wurde, können nur Ausnahmeobjekte aus diesen beiden Klassen über Prozessgrenzen hinweg zum Aufrufer transportiert werden.

---

[1] Aufmerksame Leser werden sich vielleicht fragen, ob dieses Ziel den Singleton-Aufwand tatsächlich erforderlich macht. Schließlich wird z.B. in der Klasse `EventProvider` mit weniger Aufwand dafür gesorgt, dass genau *ein* Memberobjekt der Klasse **UriMatcher** entsteht (siehe Abschnitt 13.2.2.1). Unter Software-Entwicklern wird oft darüber diskutiert, ob die beiden folgenden Lösungen gleichwertig sind:

- Singleton-Muster (siehe Abschnitt 12.4.1)
- Statische, öffentliche und finalisierte Referenz auf ein beim Laden der Klasse erstelltes Objekt, kein öffentlicher Konstruktor

Tatsächlich sorgen beide Lösungen für die Beschränkung auf *ein* Objekt und für Thread-Sicherheit. Auch beim Ziel, die Projektkreation möglichst „faul“ und spät (engl.: *lacy*) durchzuführen, gibt es oft keinen Unterschied, weil Klassen generell nur bei Bedarf geladen werden. Allerdings ist die Singleton-Lösung flexibler, weil die Getter-Methode geändert werden kann, ohne die Syntax zugreifender Klassen ungültig zu machen. Im konkreten Fall ist mit einer solchen Änderung aber kaum zu rechnen, und wir hätten in der Klasse `EventDiaryOpenHelper` statt des Singleton-Musters auch die die alternative, etwas einfachere Lösung wählen können.

Das bei einem erfolgreichen Aufruf zu liefernde **Uri**-Objekt zur neuen Tabellenzeile wird über die Methode **withAppendedId()** der Klasse **ContentUris** erstellt.

13.2.2.5 delete() implementieren

Die **ContentProvider**-Methode **delete()**

public abstract int delete (Uri *uri***, String** *selection***, String[]** *selectionArgs***)**

löscht aus der per **Uri**-Objekt spezifizierten Tabelle alle Objekte, die der Auswahlbedingung entsprechen, welche über die Parameter *selection* und *selectionArgs* formuliert wird (siehe Abschnitt 12.5.3). In unserer Klasse `EventProvider` wird die Aufforderung zum Löschen an das die Datenbank vertretende ein **SQLiteDatabase**-Objekt weitergeleitet:

```
@Override
public int delete(Uri uri, String selection, String[] selectionArgs) {
    int nDel;
    if (uriMatcher.match(uri) == 1) {
        SQLiteDatabase db = dbHelper.getWritableDatabase();
        nDel = db.delete(EventDiaryOpenHelper.DB_TABLE_EVENTS, selection, selectionArgs);
        Toast.makeText(getContext(), R.string.deleted, Toast.LENGTH_SHORT).show();
    } else
        throw new IllegalArgumentException("URI invalid");
    return nDel;
}
```

13.2.2.6 update() implementieren

Die **ContentProvider**-Methode **update()**

public abstract int update (Uri *uri***, ContentValues** *values***,**
String *selection***, String[]** *selectionArgs***)**

aktualisiert in der per **Uri**-Objekt spezifizierten Tabelle alle Objekte, die der Auswahlbedingung entsprechen (siehe Abschnitt 12.5.3), wobei ein **ContentValues**-Objekt die neuen Werte enthält (siehe Abschnitt 13.1.2.2). Wenn ein Content Provider eine Datenbanktabelle zugänglich macht, wird er die Aktualisierungsoperation vom zuständigen **SQLiteDatabase**-Objekt ausführen lassen, z.B.:

```
@Override
public int update(Uri uri, ContentValues values, String selection, String[] selectionArgs) {
    int nChanged;
    if (uriMatcher.match(uri) == 1) {
        SQLiteDatabase db = dbHelper.getWritableDatabase();
        nChanged = db.update(EventDiaryOpenHelper.DB_TABLE_EVENTS, values,
                             selection, selectionArgs);
        Toast.makeText(getContext(), R.string.updated, Toast.LENGTH_SHORT).show();
        return nChanged;
    } else
        throw new IllegalArgumentException("URI invalid");
}
```

An den Aufrufer wird die Zahl der geänderten Datenbankzeilen zurückgemeldet.

13.2.2.7 getType() implementieren

Content Provider sollten zu jedem unterstützten URI einen MIME-Typ festlegen, um über die Beschaffenheit der gelieferten Daten zu informieren (vgl. Abschnitt 13.1.5). Die folgende Methode stammt aus dem NotePad-Beispiel der Firma Google und wurde der Einfachheit halber leicht abgewandelt:[1]

```
@Override
public String getType(Uri uri) {
    switch (uriMatcher.match(uri)) {
        case 1:
            return "vnd.android.cursor.dir/vnd.google.note";
        case 2:
            return "vnd.android.cursor.item/vnd.google.note";
        default:
            throw new IllegalArgumentException("Unknown URI " + uri);
    }
}
```

Im `EventProvider` - Beispiel wird auf die Definition und Rückmeldung von MIME-Typen zu der Ereignistabelle verzichtet:

```
@Override
public String getType(Uri uri) {
    return null;
}
```

13.2.3 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:[2]

- **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 13.2.1). Android kann mit den Informationen aus diesem Attribut eine Tabelle mit den installierten Providern aufbauen und bei einer **ContentResolver**-Anfrage die zuständige Anwendungskomponente ausfindig machen.
- Mit etlichen Attributen lässt sich die Behandlung und die Verfügbarkeit des Providers steuern, z.B.:

---

[1] Quelle: https://android.googlesource.com/platform/development/+/master/samples/NotePad/

[2] Weitere Details sind z.B. in der Online-Dokumentation zu Android zu finden:
http://developer.android.com/guide/topics/manifest/provider-element.html

  - **android:enabled**
    Dieses Attribut mit der Voreinstellung **true** legt fest, ob der Provider verfügbar ist, d.h. ob Android beim ersten Zugriff automatisch ein Objekt des Providers instanziieren soll. Die Einstellung kann mit Hilfe der Klasse **PackageManager** per Programm geändert werden.
  - **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 13.2.4.

13.2.4 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 13.2.2) Rechte auf Provider-, Tabellen oder Zeilenebene gesetzt werden. Wir beschränken uns auf die Definition von Lese- und Schreibrechten auf Providerebene.[1]

Für das Lese- und das Schreibrecht zum `EventProvider` wird jeweils ein weltweit eindeutiger Name festgelegt und dem Attribut **android:readPermission** bzw. **android:readPermission** des **provider** - Elements der Manifestdatei zugewiesen:

```
<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:

```
<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. Werden neben dem generellen Recht auch spezifische Rechte definiert (also **android:readPermission** bzw. **android:readPermission**), dann dominieren letztere.

13.2.5 Modifikationen der bisherigen Event Diary - Klassen

Nachdem unser Ereignistagebuch nunmehr über einen Content Provider verfügt, sollte diese Komponente auch von den anwendungseigenen Aktivitäten genutzt werden, um unsinnigen Doppelaufwand beim Datenzugriff zu vermeiden. Folglich werden in der Klasse `EventDiaryOpenHelper` alle Datenbankzugriffsmethoden gestrichen, weil die Klasse `EventProvider` diese Leistungen

---

[1] Weitere Details sind z.B. in der Online-Dokumentation zu Android zu finden:
http://developer.android.com/guide/topics/providers/content-provider-creating.html#Permissions

erbringen soll. In den Klassen `EventListActivity` und `EditEventActivity` müssen die Datenbankzugriffsmethoden auf die Nutzung des **ContentResolver**-Objekts umgestellt werden. Das macht keine große Mühe, wie z.B. die Methode **onCreate()** der Klasse `EditEventActivity` zeigt:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_edit_event);

    final RadioGroup radioGroup = (RadioGroup) findViewById(R.id.radioGroup);
    Button btnEvent = (Button) findViewById(R.id.btnEvent);
    EditText event = (EditText) findViewById(R.id.etEvent);

    int colEventDesc = 2, colMood = 3;
    final Cursor cursor;

    Bundle extras = getIntent().getExtras();
    if (extras != null) {
        insertMode = false;
        rowid = extras.getLong("rowid");
        cursor = getContentResolver().query(EventProvider.CONTENT_URI, null, "_id=?",
                                            new String[]{Long.toString(rowid)},null);
        if (!cursor.moveToFirst())
            this.finish();
        event.setText(cursor.getString(colEventDesc));
        int mood = cursor.getInt(colMood);
        cursor.close();
        ((RadioButton) radioGroup.getChildAt(mood - 1)).setChecked(true);
        btnEvent.setText(getString(R.string.update_event));
    } else
        this.setTitle(getString(R.string.title_activity_new_event));

    btnEvent.setOnClickListener(new View.OnClickListener() {
      .  .  .
    });
}
```

Das vollständige AS-Projekt auf dem aktuellen Stand befindet sich im Ordner

…\BspUeb\Content Provider\EventDiary.4

13.3 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 12 und im bisherigen Verlauf von Kapitel 13 vorgestellten Datenbankzugrifftechnik 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 in diesem Zusammenhang 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.

13.3.1 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.
- **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**
  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.

13.3.2 Loader-Callback - Methoden implementieren

Zur Kommunikation zwischen einem Klienten (Aktivität oder Fragment) und seinem **LoaderManager** dienen die Methoden der Schnittstelle **LoaderManager.LoaderCallbacks**, 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 Loader-Callback-Methoden an:

```
public class EventListActivity extends AppCompatActivity
      implements AdapterView.OnItemClickListener, LoaderManager.LoaderCallbacks<Cursor> {
  .  .  .
}
```

Ist ein zu initialisierendes **Loader<Cursor>** - Objekt (vgl. Abschnitt 13.3.3) noch nicht 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 13.1.2.1):

```
@Override
public Loader<Cursor> onCreateLoader(int id, Bundle args) {
    return new CursorLoader(this,
            EventProvider.CONTENT_URI, null, null, null,
            EventDiaryOpenHelper.DB_COL_DATETIME +" DESC");
}
```

Weil bei unserer Anwendung nur *eine* Loader-Kennung im Spiel ist, müssen wir den ersten 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 Überladung wird daraufhin das in der `EventListActivity` tätige Adapterobjekt per **swapCursor()** beauftragt, den frisch erstellten Cursor zu verwenden:

```
@Override
public void onLoadFinished(Loader<Cursor> loader, Cursor data) {
    adapter.swapCursor(data);
}
```

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(Loader<Cursor> loader) {
    adapter.swapCursor(null);
}
```

13.3.3 Loader-Initialisierung und -Aktualisierung

In der `EventListActivity`-Methode **onCreate()** verzichten wir 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.PRO_COL_DATETIME +" DESC");
//        startManagingCursor(cursor);
```

Stattdessen erhält das **ListView**-Steuerelement in der `EventListActivity` ein Adapterobjekt *ohne* Cursor (mit **null** als Wert für den zweiten Konstruktor-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 13.3.2).

Der **LoaderManager** der Aktivität wird in **onCreate()** mit der Methode **initLoader()** dazu aufgefordert, für ein initialisiertes **Loader<Cursor>** - Objekt zu sorgen:

```
getLoaderManager().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 Loader-Callback-Methode **onCreateLoader()** weitergereicht wird (siehe Abschnitt 13.3.2). Mit dem obligatorischen dritten Aktualparameter ist das Objekt zu benennen, das die Loader-Callback - 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 Loader-Callback - Methode **onCreateLoader()** aufgerufen.

So sieht die modernisierte Methode im Überblick aus:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_event_list);

    lv = (ListView) findViewById(R.id.listView);
    lv. setOnItemClickListener(this);
    registerForContextMenu(lv);

    adapter = new EventDiaryAdapter(this, null);
    lv.setAdapter(adapter);
    getLoaderManager().initLoader(0, null, this);
}
```

Wird das Loader-API in einem Fragment verwendet, eignet sich die Lebenszyklusmethode **onActivityCreated()** am besten dazu, einen Loader zu initialisieren.

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(EventProvider.CONTENT_URI, "_id=?",
                                         new String[]{Long.toString(acmi.id)});
if (numDel > 0)
    getLoaderManager().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)
        getLoaderManager().getLoader(0).onContentChanged();
}
```

Das vollständige AS-Projekt auf dem aktuellen Stand befindet sich im Ordner

…\BspUeb\Content Provider\EventDiary.5

14 Shared Preferences

Zur Verwaltung von Programmeinstellungen stellt Android ein API bereit, das …

- für Benutzer eine einheitliche Bedienoberfläche zur Bearbeitung von Einstellungen bietet (inkl. Systemeinstellungen)
- und dem Programmierer sehr viele Routinearbeiten abnimmt (z.B. Gestaltung der Bedienoberfläche, lesende und schreibende Dateizugriffe).

Die als *Shared Preferences* bezeichneten 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 taugen die Shared Preferences als Daten-Persistenzlösung mit folgenden Einschränkungen:

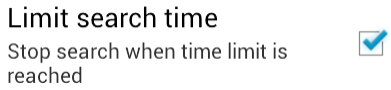
- Es geht um eine begrenzte Datenmenge.
- Ausschließlich erlaubte Datentypen: **boolean**, **float**, **int**, **long**, **String** und **Set<String>**

Wenn es etwa um eine Persistenzlösung für drei Zeichenfolgen und zwei Zahlen geht, sind die Shared Preferences deutlich handlicher als eine Datenbanklösung oder eine Dateiaus- und -eingabe.

14.1 Preferences-Ressourcendatei

Android verwendet zur Verwaltung einer Einstellung jeweils ein Objekt aus der Klasse **Preference** (im Namensraum **android.preference**) oder einer Ableitung. Häufig verwendete **Preference**-Ableitungen sind:

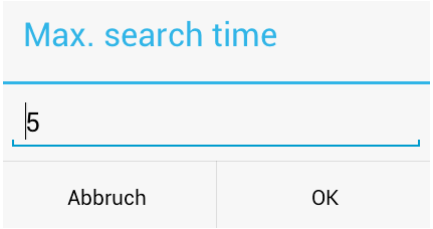
- **CheckBoxPreference**
  Der Benutzer sieht ein Kontrollkästchen und legt mit seiner Wahl einen Wert mit Datentyp **boolean** fest, z.B.:



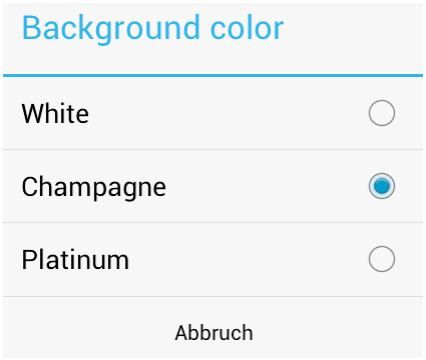
- **SwitchPreference**
  Der Benutzer sieht einen Umschalter und legt mit seiner Wahl einen Wert mit Datentyp **boolean** fest, z.B.:



- **EditTextPreference**
  In der Einstellungsliste entsteht ein Item, das nach dem Antippen ein Dialogfeld zur Texteingabe erscheinen lässt, so dass schließlich eine Zeichenfolge resultiert. Welche Eingaben hier möglich sind, kann der Programmierer wie bei einem **EditText**-Steuerelement festlegen. Im folgenden Beispiel sind nur positive, ganze Zahlen erlaubt:



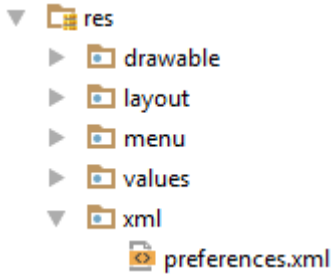
- **ListPreference**
  In der Einstellungsliste entsteht ein Item, das nach Auswahl ein Optionsfeld (eine Liste von Optionsschaltern) erscheinen lässt, z.B.:



Die Wahl des Benutzers wird als Zeichenfolge gespeichert.

- **MultiSelectListPreference**
  Diese Klasse unterstützt eine Liste mit Mehrfachwahl.
- **PreferenceCategory**, **PreferenceScreen**
  Diese Klassen unterstützen die strukturierte Präsentation von Einstellungen in der Bedienoberfläche. Man kann Einstellungen in Gruppen oder untergeordneten Einstellungsseiten zusammenfassen.

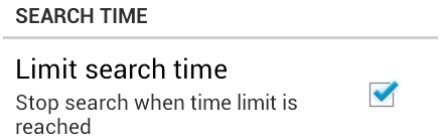
Zwar kann man Einstellungsobjekte per Programm erzeugen, doch ist in der Regel (wie bei **View**-Objekten) die Deklaration per XML-Datei zu bevorzugen. Dazu legt man nötigenfalls im Projekt-Ordner **res** einen Unterordner mit dem Namen **xml** an und erstellt dort eine XML-Datei, die traditionell (aber nicht verpflichtend) den Namen **preferences.xml** erhält:



Als XML-Wurzel ist ein **PreferenceScreen** - Element zu verwenden, und bei den Kindelementen sind die oben genannten Typen erlaubt.

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 14.4).
- **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>
```

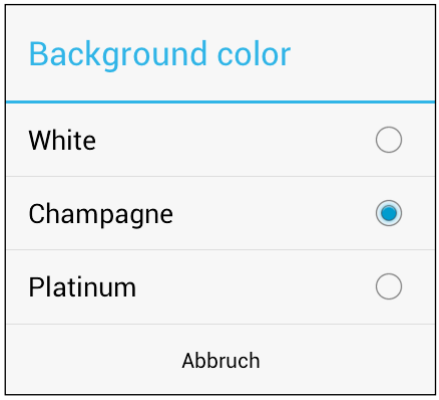
Über das **CheckBoxPreference**-Element `limitSearchTime` können Benutzer den Primzahlendetektor veranlassen, die Prüfung eines Kandidaten nach Überschreitung einer maximalen Zeit abzubrechen.[1] Per Voreinstellung ist die Suchzeitbeschränkung inaktiv.

Nur bei aktivierter 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 das Programms bestimmen:

[1] Das Programm so zu erweitern, dass es die Suchzeitbeschränkung tatsächlich beachtet, bleibt dem Leser überlassen.



In den **ListPreference**-Attributen **android:entries** bzw. **android:entryValues** mit den für Menschen lesbaren Optionsnamen bzw. den dahinter stehenden Werten

```
android:entries="@array/bgColor"
android:entryValues="@array/bgColorValues"
```

werden Ressourcen vom Typ Array angegeben, die in der Datei **...\res\values\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>
```

Weitere Details zur Einstellungs-Ressourcendeklaration sind auf Googles Webseite für Android-Entwickler zu finden.[1]

14.2 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 Hilfe des Android-SDK - Werkzeugs **monitor.exe**, das sich auf einem Windows-Rechner mit 64-Bit-Architektur per Voreinstellung im Ordner

C:\Users*user*\AppData\Local\Android\sdk\tools\lib\monitor-x86_64

befindet, kann man das Dateisystem eines virtuellen Android-Gerätes besonders bequem inspizieren und sogar Dateien zwischen dem Entwicklungsrechner und dem Android-Gerät austauschen.[2] Das Werkzeug lässt sich aus dem Android Studio über den Menübefehl

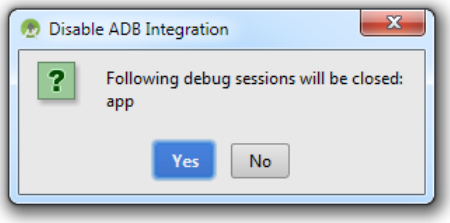
---

[1] Sie z.B. http://developer.android.com/guide/topics/ui/settings.html

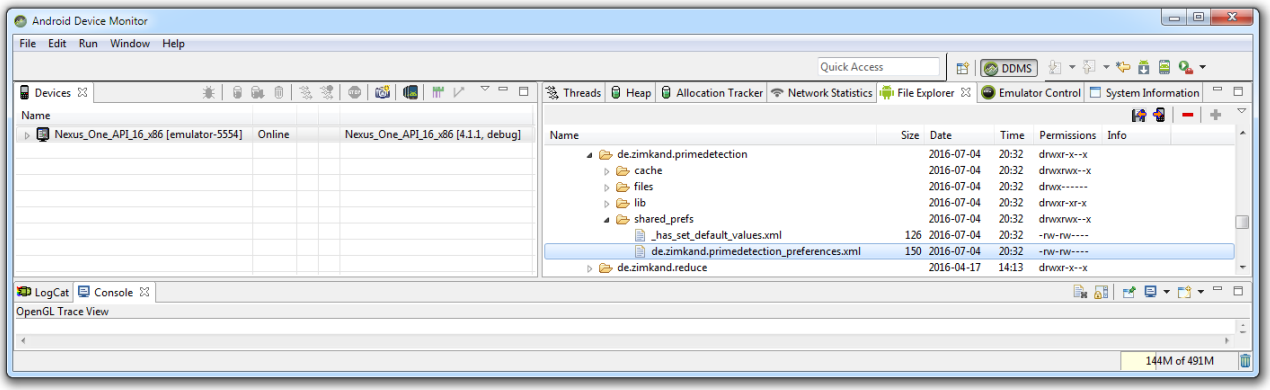
[2] Reale Android-Geräte gewähren aus verständlichen Gründen weniger großzügige Einblicke in ihr Dateisystem. Hier lässt sich z.B. der Ordner **data**, für den wir uns momentan interessieren, *nicht* öffnen.

Tools > Android > Android Device Monitor

und mit dem Symbol (links neben dem Fragezeichensymbol des Hilfesystems) starten. Eine aktive Debug-Sitzung muss gestoppt werden, damit die via Android Debug Bridge für den Device Monitor frei wird:[1]



Im nun startenden **Android Device Monitor** markieren wir nötigenfalls das gewünschte Gerät, wählen rechts die Registerkarte **File Explorer** und steuern den oben genannten Einstellungsordner an. Hier ist das Ergebnis für die App im Paket **de.zimkand.primedetection** zu sehen:



Per Symbolschalter können wir ...

- eine Datei löschen
- eine Datei vom Androiden zum PC befördern
- eine Datei vom PC in den aktuell markierten Ordner auf dem Androiden befördern

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 14.3). Diese basieren auf der in Abschnitt 14.1 beschriebenen Einstellungsdatei.
- **maxPrimeStore.xml**
  Hier speichert das Programm (ohne intentionale Beteiligung des Benutzers) die größte bisher entdeckte Primzahl (siehe Abschnitt 14.4).

Um diese Dateien mit den Mitteln des Android-Betriebssystems wieder los zu werden, kann ein Benutzer auf der App-Seite der Systemeinstellungen ...

---

1 Wenn der Device Monitor erst einmal läuft, kann man im Android Studio über den Menübefehl

Tools > Android > Enable ADB Integration

die ADB (Android Debug Bridge) wieder übernehmen und wie gewohnt mit virtuellen Maschinen arbeiten. Der Device Monitor zeigt das Dateisystem des Android-Gerätes weiterhin korrekt an, ist aber eventuell nicht mehr vollständig funktionstüchtig.

- die **Daten löschen**



- oder die Anwendung **deinstallieren**

14.3 Preferences-Bedienoberfläche

14.3.1 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 14.2).

Soll eine App noch Android-Version vor dem API-Level 11 unterstützen, ist zur Erstellung der Einstellungs-Bedienoberfläche die von **Activity** abgeleitete Klasse **PreferenceActivity** zu verwenden. Setzt eine App die Android-Version 3 (das API-Level 11) voraus, sollte nach den Empfehlungen von Google die von **Fragment** abgeleitete Klasse **PreferenceFragment** verwendet werden.[1] Weil unser Beispielprogramm nur Android-Geräte ab API-Level 15 unterstützen soll, realisieren wir die Einstellungen über die Basisklasse **PreferenceFragment**. Zusätzlich benötigen wir eine Activity als Rahmen für das Einstellungsfragment. Um die Anzahl der Quellcodedateien in Grenzen zu halten, definieren wir die **PreferenceFragment**-Ableitung als innere Klasse der Aktivität:

[1] http://developer.android.com/guide/topics/ui/settings.html

```
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);
       }
   }

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);

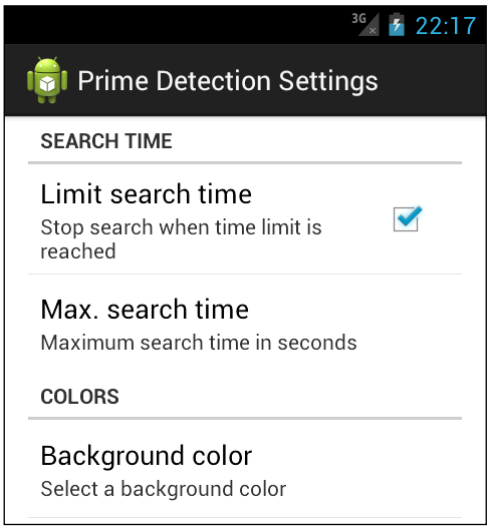
       getFragmentManager().beginTransaction()
       .replace(android.R.id.content, new SettingsFragment())
       .commit();
   }
}
```

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 beiden Klassen 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 Einstellungsaktivität* noch nicht vorhanden, wird sie unter Verwendung der Voreinstellungswerte aus der Preferences-Ressourcendatei (siehe Abschnitt 14.1) 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 **setDefaultValues()** der Klasse **PreferenceManager** auf, z.B.:

```
PreferenceManager.setDefaultValues(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, so dass eine vorhandene Einstellungsdatei unverändert bleibt. 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 14.2). Eine erneute Ausführung der Methode **setDefaultValues()** eignet sich übrigens *nicht* dazu, Voreinstellungswerte wiederherzustellen. Es werden lediglich fehlende Einstellungen ergänzt, vorhandene behalten jedoch ihren aktuellen Wert.

Wirft man mit dem in Abschnitt 14.4.1 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. auch nach Rückkehr von der Einstellungsaktivität zur Hauptaktivität des Programms abläuft:

```
@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 14.4 zu erläuternden **SharedPreferences** - Methode **getString()**. Aus der gewonnen Zeichenfolge erstellt die statische **Color**-Methode **parseColor()** einen **int**-Wert, der als Parameter für die **View**-Methode **setBackgroundColor()** dient.

14.3.2 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 7.6). Lässt man das Android Studio in der Version 2.x ein neues Projekt mit einer **Basic Activity** anlegen, resultiert im Projek-

tordner **...\res\menu** per Voreinstellung die folgende XML-Datei **menu_main.xml** zur Deklaration eines Optionsmenüs, das unserem momentanen Bedarf zufälligerweise 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 **Fehler! Verweisquelle konnte nicht gefunden werden.**) starten wir bei passender Item-ID die Einstellungsaktivität über ein explizites **Intent**-Objekt (vgl. Abschnitt 8.3):

```
@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);
   }
}
```

14.4 Einstellungsdaten per Programm schreiben und lesen

14.4.1 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 Variable 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 aktuell (API-Level ≤ 23) ausschließlich der Wert **MODE_PRIVATE** sinnvoll ist: Er reserviert den Zugriff für die eigene App (oder für eine App mit derselben User-ID).

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 14.1) und über die **PreferenceFragment**-Methode **addPreferences-FromResource()** (vgl. Abschnitt 14.3.1) 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:

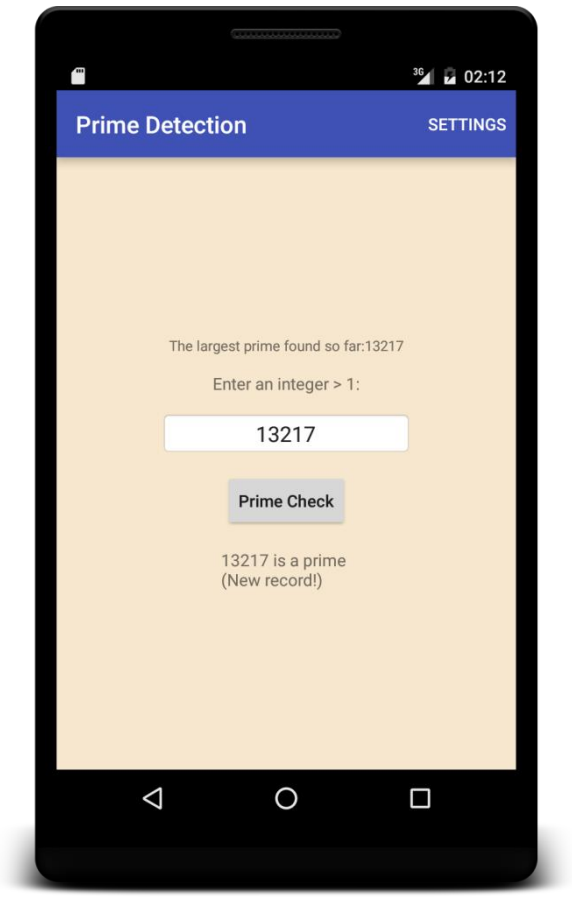
de.zimkand.primedetection_preferences.xml

Im folgenden Beispiel wird der Kontext von der Activity geliefert:

```
private SharedPreferences spMaxPrime, spDefaultPrefs;
 . . .
spDefaultPrefs = PreferenceManager.getDefaultSharedPreferences(this);
```

14.4.2 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 14.4.1 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>
```

14.4.3 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 14.4.1)

```
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 Schüssel anzugeben, was im Beispiel durch eine finalisierte Variable 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 14.3.1 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\SharedPreferences

Der Einfachheit halber wird *nicht* die mit einem Dienst arbeitende Variante der Primzahlendiagnose verwendet (siehe Kapitel 10), sondern die Variante mit einem **AsyncTask**-Objekt im Rahmen der primären Aktivität (siehe Abschnitt 9.3).

14.5 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, registrieren lassen. Dessen Methode **onSharedPreferenceChanged()** wird aufgerufen, sobald sich eine vom **SharedPreferences**-Objekt verwaltetet 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, so dass sie auf allen Android-Geräten eines Benutzers verfügbar sind (siehe Anuzzi et al. 2014, S. 296).

15 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 **sendBroadast()**). Somit spielen Broadcast Receiver eine wichtige Rolle bei der Kommunikation über Prozessgrenzen hinweg.[1]

Das Betriebssystem informiert z.B. per Rundruf über folgende Ereignisse:

- Das WLAN wurde (de-)aktiviert.
- Das Display wurde aus- oder eingeschaltet.
- Eine externe Stromversorgung wurde angeschlossen bzw. getrennt.

Beim Eintreffen eines passenden Broadcast-Intents wird die **onReveive()** - Methode eines registrieren Empfängers aufgerufen.

Die Registrierung eines Broadcast Receivers kann statisch über ein **receiver**-Element im Anwendungsmanifest oder dynamisch über die **Context**-Methode **registerReceiver()** erfolgen (siehe Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden.**).

Broadcast Receiver haben keine eigene Bedienoberfläche, können sich aber doch bemerkbar machen:

- über die Benachrichtigungsleiste am oberen Bildschirmrand (im Manuskript nicht behandelt)
- über Toast-Meldungen (siehe z.B. Seite 171)

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, so dass z.B. ein beim Aufruf der Methode **startActivity()** verwendetes **Intent**-Objekt nicht an einen Broadcast Receiver übermittelt wird.

In nächsten Abschnitt werden neben der statischen Registrierung auch einige generelle Eigenschaften der Broadcast Receiver behandelt.

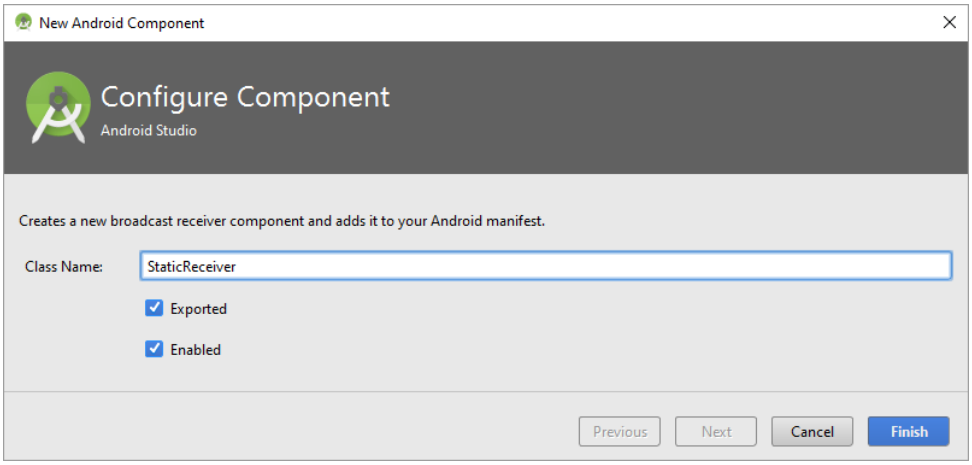
15.1 Statische Registrierung

Wir legen ein neues AS-Projekt mit einer leeren Aktivität an, und wählen anschließend aus dem Kontextmenü zum Paketeintrag im **Project**-Fenster den Befehl:

New > Other > Broadcast Receiver

Es erscheint ein Assistent mit einem einzigen Fenster, das den Namen der neu anzulegenden Klasse erfragt:

[1] Aber auch bei der Kommunikation innerhalb einer App kommen Broadcast Receiver zum Einsatz, was schon in Abschnitt 10.1.4 vorgeführt wurde.



Der Assistent liefert eine rudimentäre Klassendefinition für den Broadcast Receiver

```
package de.zimkand.brstatisch;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;

public class StaticReceiver extends BroadcastReceiver {
    public StaticReceiver() {
    }

    @Override
    public void onReceive(Context context, Intent intent) {
    }
}
```

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.brstatisch">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <receiver
            android:name=".StaticReceiver"
            android:enabled="true"
            android:exported="true">
        </receiver>
    </application>
</manifest>
```

Unser Receiver stammt von der Klasse **BroadcastReceiver** im Pakt **android.content** ab und muss die abstrakt definierte Basisklassenmethode **onReceive()** überschreiben. Wir beschränken uns darauf, in einer Toast-Nachricht die per **getAction()** zu erfahrende Aktion des empfangenen Toast-Objekts anzuzeigen:

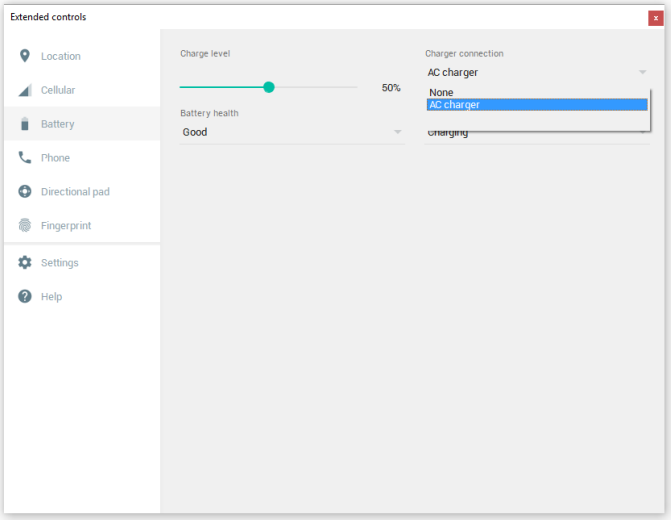
```
@Override
public void onReceive(Context context, Intent intent) {
    Toast.makeText(context, intent.getAction(), Toast.LENGTH_LONG).show();
}
```

Im **receiver**-Element der Manifestdatei fügen wir einen **intent-filter** (vgl. Abschnitt 8.2) mit passendem **action**-Element ein, um den Empfänger für Systemnachrichten über den Anschluss einer externen Stromversorgung zu registrieren:

```
<receiver
    android:name=".StaticReceiver "
    android:enabled="true"
    android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.ACTION_POWER_CONNECTED" />
    </intent-filter>
</receiver>
```

Im Android-Emulator lässt sich über die Symbolpalette ein Fenster mit erweiterten Bedienelementen anfordern, das u.a. die Möglichkeit bietet, die Verbindung mit einem Ladegerät zu simulieren:





Auf diese Weise können wir mit einem emulierten Gerät testen, ob unser statisch registrierter Broadcast Receiver (auch bei inaktiver App!) die Systemnachrichten erhält und verarbeitet, z.B.:



Reagiert ein korrekt entwickelter Receiver *nicht* auf passende Broadcast Intents, dann kommen vor allem zwei Ursachen in Frage:

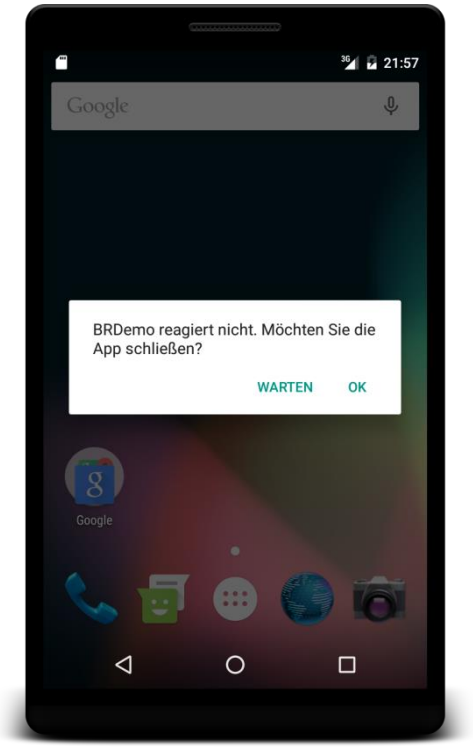
- Seine Anwendung befindet sich im gestoppten Zustand. Um diese Ursache werden wir uns gleich kümmern.
- Fehlende Rechte (siehe Abschnitt 15.5)

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 15.3). 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* > Stoppen 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 5).

Die **onReceive()** - Methode sollte maximal 10 Sekunden lang aktiv bleiben, weil sonst mit einem Einschreiten des Betriebssystems zu rechnen ist.[1] Bei Tests mit virtuellen Smartphones unter Android 4.1.1 oder 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:



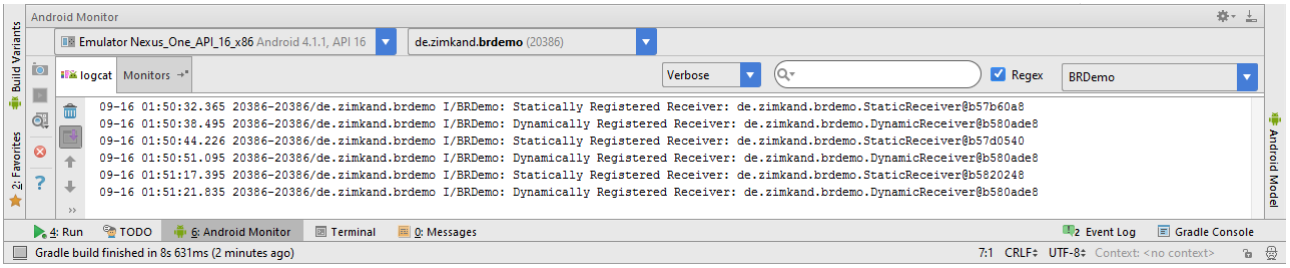
Andernfalls wurde der Anwendungsprozess nach ca. 60 Sekunden zerstört. Wenn eine lang laufende **onReceive()** - Methode eine Vordergrundaktivität daran hindert, auf Bedienelemente zu reagieren, ist bereits nach 5 Sekunden mit einer ANR-Fehlermeldung zu rechnen.

[1] https://developer.android.com/reference/android/content/BroadcastReceiver.html#ProcessLifecycle

Nach Beendigung der Methode **onReceice()** 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 ist:



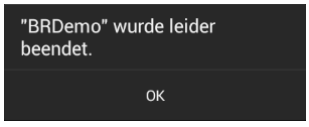
Bei einem *dynamisch* registrierten Receiver (siehe Abschnitt 15.2) werden hingegen mehrere **onReceive()** - Aufrufe vom *selben* Objekt ausgeführt, sofern die registrierende Aktivität nicht zwischenzeitlich zerstört und neu erstellt worden ist.

Die Objekte eines statisch registrierten Receivers werden nach Ablauf der **onReceive()** - Methode als ungültig betrachtet. Enthält der Prozess einer App neben einem statisch registrierten Receiver keine weiteren Komponenten, besitzt er nach Beendigung der **onReceive()** - Methode eine sehr kleine Priorität und dementsprechend ein relativ hohes Risiko, bei Speichermangel eliminiert zu werden (vgl. Abschnitt 4.4).

Daher darf in **onReceive()** 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:



Wenn ein statisch registrierter Receiver einen Dialog initiieren soll, kann das über einen beauftragten Dienst geschehen.

Es wird davon abgeraten, aus **onReceive()** einen Hintergrund-Thread (z.B. per **AsyncTask**) zu starten, weil das am Ergebnis interessierte Receiver-Objekt bei Erledigung der Aufgabe eventuell seine **onReceive()** - Methode längst beendet hat und nicht mehr existiert.[1] Das Problem lässt sich vermeiden, indem aus **onReceive()** ein Dienst beauftragt wird. Dazu kann ...

[1] https://developer.android.com/reference/android/content/BroadcastReceiver.html

- mit **startService()** ein Dienst neu gestartet
- oder mit **peekService()** ein bereits laufender Dienst angesprochen werden kann.

Es ist jedoch *nicht* sinnvoll, einen Dienst mit **bindService()** einzubinden.

Seit dem API-Level 11 kann ein statisch registrierter Receiver aus **onReceive()** eine Hintergrundoperation veranlassen und nach Beendigung von **onReceive()** auf das Ergebnis der Hintergrundverarbeitung reagieren. Im Unterschied zur Hintergrundbearbeitung mit Dienstbeteiligung ist jedoch die gesamte Operation derselben Laufzeitbeschränkung unterworfen wie die Methode **onReceive()** selbst, so dass als maximal mögliche Bearbeitungszeit nur 10 Sekunden 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();
```

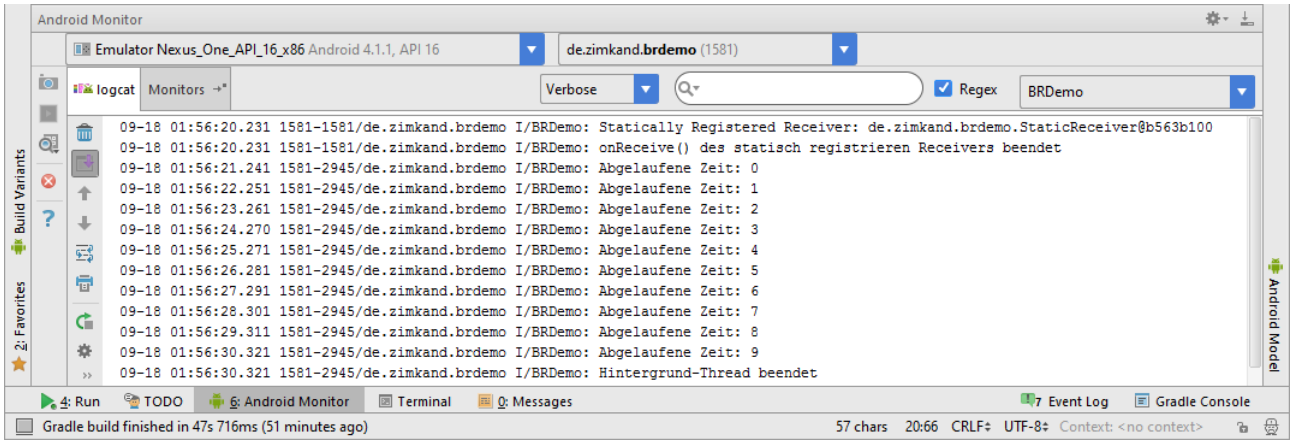
Innerhalb der garantierten maximalen Lebensdauer bleibt das Receiver-Objekt über das Ende der Methode **onReceive()** hinaus so lange erhalten, bis die **finish()** - Methode des **PendingResult**-Objekts aufgerufen wird.

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 registrieren Receivers beendet");
    }
}
```

aktiv wird, entstehen im **logcat**-Fenster der Entwicklungsumgebung die folgenden Protokolleinträge:



15.2 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 und ihre Anzeige aktuell halten muss.
- Für einige Broadcast-Intents des Betriebssystems sind ausschließlich dynamisch registrierte Receiver erlaubt, um den Akku zu schonen. Ein Beispiel ist das einmal pro Minute gefeuerte Ereignis **Intent.ACTION_TIME_TICK**.

Besonderheiten eines dynamisch registrierten Receivers im Vergleich zum statisch (per Anwendungsmanifest) registrierten Receivers 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 meist 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 seine **onReceive()** - Methode nicht in der erlaubten Zeitspanne (nominell 10, empirisch 60 Sekunden) endet, wird dasselbe Verhalten bei einem dynamisch registrierten Receiver toleriert (beobachtet bei virtuellen Smartphones unter Android 4.1.1 und 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 Activity zu demonstrieren. Wir verwenden eine **BroadcastReceiver**-Ableitung, die sich nur durch den Klassennamen von der in Abschnitt 15.1 verwendeten Variante unterscheidet:

```
package de.zimkand.brdemo;

import android.content.BroadcastReceiver;

  .  .  .

public class DynamicReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
    }
}
```

Die Hauptaktivität der App erzeugt in ihrer Lebenszyklusmethode **onCreate()** ein Objekt der Receiver-Klasse 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);
    }
}
```

Wird eine Aktivität zerstört (z.B. nach einer Rückwärtsnavigation des Benutzers), die eine dynamische Broadcast Receiver - Registrierung vorgenommen, aber nicht aufgehoben hat, dann beschwert sich Android über eine Speicherverschwendung (*has leaked*), z.B.:

```
09-24 16:01:33.884 9561-9561/de.zimkand.brdemo E/ActivityThread: Activity
de.zimkand.brdemo.MainActivity has leaked IntentReceiver
de.zimkand.brdemo.DynamicReceiver@b579b458 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.

Wenn ein Service (vgl. Kapitel 10) 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.

15.3 Broadcast-Nachrichten versenden

Broadcast-Intents stammen größtenteils vom System, doch können sich auch Apps am Broadcast-basierten Kommunikationsverfahren beteiligen, was in diesem Abschnitt demonstriert wird. Zum Versenden einer Nachricht im normalen, asynchronen Modus dienen die beiden 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 15.5).

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 8). Grundsätzlich ist es aber auch möglich, mit einem expliziten **Intent**-Objekt per „Broadcast“ einen *einzigen*, 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 8.1.1):

```
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 15.1) 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.

15.4 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.

15.4.1 (Un)geordnete Verarbeitung

Beim oben beschriebenen Versandt mit der Context-Methode **sendBroadcast()** erfolgt die Zustellung an mehrere registrierte Empfänger ohne definierte Reihenfolge (eventuell auch gleichzeitig). Außerdem kann ein Empfänger keine Informationen an Nachfolger weitergeben und die Auslieferungssequenz nicht stoppen.

Nach dem Versandt 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.

15.4.2 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 Versandt eines Broadcast-Intents mit der Methode **sendStickyBroadcast()** (erforderliche Berechtigung: **android.permission.BROADCAST_STICKY**) den permanenten Verbleib im System anzuordnen, so dass die Nachricht auch an später registrierte Receiver ausgeliefert wird.[1] 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, so dass sie in eigenen Apps nicht mehr verwendet werden sollte (siehe Müller 2015, S. 34).

Die von vielen Apps abonnierte Broadcast-Nachricht **Intent.ACTION_BATTERY_CHANGED** mit dem aktuellen Ladezustand des Akkus wird jedoch z.B. in Android 6 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
```

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:[2]

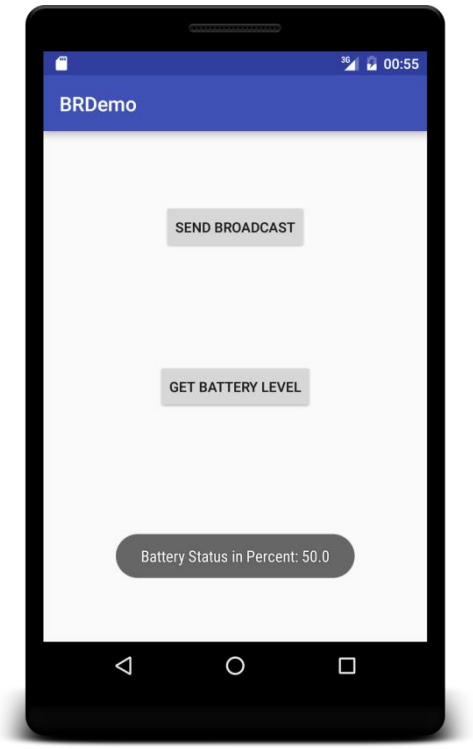
---

[1] Deutsche Übersetzung von *sticky*: *klebrig*

[2] Idee übernommen von https://developer.android.com/training/monitoring-device-state/battery-monitoring.html

```
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**.[1]

15.5 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.

15.5.1 Beschränkung der zulässigen Absender

Soll ausnahmsweise 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.:

[1] https://developer.android.com/reference/android/content/Context.html

```
<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**, so dass die Aktivität auch von fremden Apps genutzt werden kann (siehe Abschnitt 8.2).

Für Receiver, die nur zur anwendungsinternen Kommunikation dienen sollen, ist eine spezielle Technik verfügbar, die eine bessere Performanz bietet und keine Sicherheitsprobleme kennt (siehe Abschnitt 0).

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, so dass 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 reglementieren. 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 android:priority="1">
                . . .
            </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.explbroadcast">
    <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** *broadcastPermission***, 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.

15.5.2 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 angesprochen 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 zu verwenden, z.B.:

```
sendBroadcast(intent, "de.zimkand.permission.BROADCAST");
```

Empfangsberechtigt sind nur solche Apps, welche die erforderliche Berechtigung über ein **uses-permission** - Element im Anwendungsmanifest erfolgreich beantragt haben (siehe Abschnitt 15.5.1).

Verwendet der Sender eine App-eigene Berechtigung, muss er diese in seiner Manifestdatei definieren, 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>
```

15.6 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 anwendungsinternen Kommunikation mit Hilfe der Klasse **LocalBroadcastManager** haben wir schon in Abschnitt 10.1.4 kennen gelernt. Dort meldet ein Dienst seine Arbeitsergebnisse per Broadcast-Intent, und eine Aktivität derselben App zeigt diese Ergebnisse in ihrer Bedienoberfläche an.

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 3 (Android Studio kennen lernen)

Aufgabe 1

Ein Lösungsvorschlag ist im folgenden Ordner zu finden:

...\BspUeb\Reduce

Kapitel 4 (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.

Kapitel 5 (Lebenszyklus einer Aktivität)

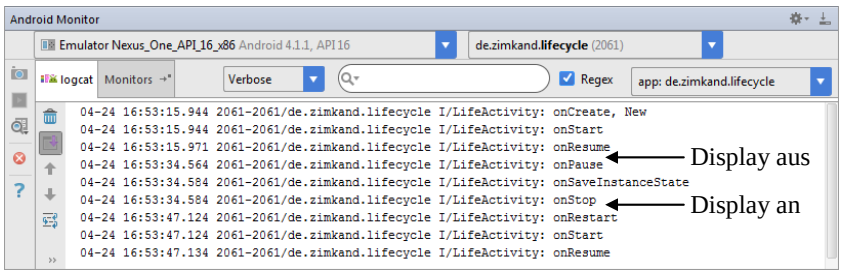
Aufgabe 1

Ausgehend vom Beispielprogramm in Abschnitt 5.1 genügt eine minimale Änderung in der Methode **onPause()**:

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

Aufgabe 2

Wie das folgende **logcat**-Protokoll (erstellt mit der in Abschnitt 5.3 beschriebenen App) demonstriert, wird die beim Abschalten des Displays die im Vordergrund befindliche App gestoppt und nach dem Einschalten des Displays wieder gestartet:



Diese Beobachtungen stammen von einem Smartphone im Hochformat. Auf einem Smartphone, das sich vor dem Abschalten des Displays im Querformat befindet und einen Sperrbildschirm besitzt, treten Orientierungswechsel auf. Wie wir aus Abschnitt 5.3.2.1 wissen, führt jeder Orientierungswechsel zur Zerstörung und Neuerstellung der Vordergrundaktivität.

Kapitel 6 (Ressourcen)

Aufgabe 1

Das Projekt `PrimeDetection` auf dem aktuellen Entwicklungsstand ist im folgenden Ordner zu finden:

...\BspUeb\PrimeDetection

Kapitel 7 (Bedienoberfläche)

Aufgabe 1

Ein Lösungsvorschlag ist im folgenden Ordner zu finden:

...\BspUeb\GUI\Numerische Eingabe

Aufgabe 2

1. **Falsch**
   Siehe Abschnitt 7.5.2.3
2. **Richtig**
3. **Richtig**
   Eine Erläuterung zu den so genannten *Layout-Parametern* findet sich in Abschnitt 7.2.1.

Kapitel 8 (Intents und Intent-Filter)

Aufgabe 1

Ein Lösungsvorschlag ist im folgenden Ordner zu finden:

...\BspUeb\Intent\Settings

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.

Kapitel 9 (Multithreading)

Aufgabe 1

1. **Richtig**
2. **Falsch**
3. **Falsch**
   Es wird lediglich das Interrupt-Signal gesetzt. Ein kooperativer Thread beachtet dieses Signal regelmäßig und stellt ggf. seine Tätigkeit ein.

Literatur

Anuzzi, J. Jr., Darcey, L. & Conder, S. (2014). *Introduction to Android Application Development* (4. ed.). Upper Saddle River, NJ: Addison-Wesley.

Baltes-Götz, B. (2012). *Einführung in das Programmieren mit C# 4.0*. Online-Dokument: https://www.uni-trier.de/index.php?id=22777

Baltes-Götz, B. & Götz, J. (2016). *Einführung in das Programmieren mit Java 8*. Online-Dokument: http://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

Gosling, J., Joy, B., Steele, G., Bracha, G. & Buckley, A. (2011). *The Java Language Specification. Java SE 7 Edition*. Online-Dokument: http://download.oracle.com/javase/7/specs/jls/JLS-JavaSE7.pdf

Jackson, W. (2014). *Android Apps for Absolute Beginners* (3rd ed.). New York: Apress.

Louis, D. & Müller. P. (2014). *Android*. München: Hanser.

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.

Müller, D. (2015). Untersuchung zur Broadcastsicherheit von Android-Apps. Online-Dokument: http://www.informatik.uni-bremen.de/~sohr/papers/BachelorThesisDaniel.pdf

Post, U. (2013). *Android-Apps entwickeln für Einsteiger* (3. Aufl.). Bonn: Galileo Press.

Smith, G.J. (2014). Analysis and Prevention of Code-Injection Attacks on Android OS. *Graduate Theses and Dissertations*. http://scholarcommons.usf.edu/etd/5391 (abgerufen am 06.09.2015)

Staudemeyer, J. (2013). *Android Programmierung* (2. Aufl.). Köln: O'Reilly.

Stichwortregister

.

.exit ... 243
.table ... 242

A

AAPT ... 98
ABI ... 32
Activity ... 69
ADB ... 41, 44
addCategory() ... 180
addsURI() ... 278
addToBackStack() ... 233
Ahead-Of-Time ... 15
Akku ... 309
AlertDialog ... 68
andoid
  layout_gravity ... 127
android
  exported ... 310
Android Asset Packaging Tool ... 98
Android Debug Bridge ... 41, 240
Android Virtual Device Manager ... 31
AndroidManifest.xml ... 70
Animation ... 110, 233
Animationen ... 109
Anonyme Klassen ... 140
ANR-Fehler ... 118, 119, 194
Anwendungsobjekt ... 74
AOT ... 15
API ... 17
API-Level ... 18
APK ... 15, 49
AppCompatActivity ... 70
Application ... 74
apply()
  SharedPreferences ... 298
ARGB ... 105
ARM ... 16, 38
ArrayAdapter<String> ... 165
ART ... 15
Assembler ... 14
AsyncTask ... 200
AutoCompleteTextView ... 150
autoLink ... 143
AVD-Manager ... 31

B

Back Stack ... 71
  Fragmente ... 233
Beans ... 121
beginTransaction() ... 232
Berechtigungen
  Broadcast Receiver ... 311
  Content Provider ... 272
Binary Large Object ... 240
BIND_AUTO_CREATE ... 214
Binder ... 215
bindService() ... 214
bindView() ... 255
BLOB ... 240
borderlessButtonStyle ... 153
Broadcast Receiver ... 69, 300
BroadcastReceiver ... 210
build.gradle ... 65
Build-System ... 47
Bundle ... 78, 80, 83, 181, 218, 226, 234
Button ... 151
Bytecode ... 14

C

Call Back - Routinen ... 119
Callback-Methoden ... 78
CATEGORY_BROWSABLE ... 180
CATEGORY_DEFAULT ... 182
CATEGORY_LAUNCHER ... 180
CheckBox ... 154
Cloud Save ... 299
Color ... 295
commit() ... 232
  SharedPreferences ... 298
Compiler ... 14
ComponentName ... 177, 186, 308
CompoundButton ... 154
content ... 184
content (URI) ... 268
Content Provider ... 69, 180, 184, 190, 267
contentDescription ... 152
ContentProvider ... 276
ContentResolver ... 268
Content-URI ... 268
ContentUris ... 268
ContentValues ... 250, 271
controls ... 119
CPU ... 14
CREATE TABLE (SQL) ... 242
createChooser() ... 191
Cursor ... 246, 262
CursorAdapter ... 254
CursorLoader ... 284

D

Dalvik ... 15
DatabaseUtils ... 246
Datenbanken ... 239

DELETE (SQL) 243
delete()
ContentProvider 280
ContentResolver 272
Destroyed 77
DialogFragment 221
Dienst 205
Dienstprozesse 73, 205
dip 53
doInBackground() 203
dp 53
DROP TABLE (SQL) 243
Dropdown-Liste 160
dx 15
E
Eclipse 20
Editable 58
EditText 58, 144
Eigenschaften 121
Einfache Ressourcen 94
Element-IDs 100
ems 151
Entwickleroptionen 87
Ereignisse 138
execSQL() 250
execute()
AsyncTask 203
Explizite Intents 176
Expliziter Intent 184
exported 182
Extras
Intent 181
F
Fabrikmethode 232, 234
fill_parent 124
findFragmentById() 231
findFragmentByTag() 237
findViewById() 56, 78, 100
finish() 188
FLAG_EXCLUDE_STOPPED_PACKAGES
..... 308
FLAG_INCLUDE_STOPPED_PACKAGES
..... 303, 308
Flags 181
Flexibilität 137
Fortschrittsanzeige 162
Fragment 293
Fragmente 220
FragmentManager 231
FragmentTransaction 232
FrameLayout 131
freezesText 142
Fullscreen-Texteingabe 149
G
Gebundene Dienste 205
Gestartete Dienste 205
Gestoppter Zustand
einer App 303
getActivity() 228, 235
getArguments() 235
getDefaultSharedPreferences() 297
getExtras() 187
getIntent() 186
getIntExtra() 310
getLastPathSegment() 278
getPackageManager() 191
getReadableDatabase() 245
getSharedPreferences() 296
getString() 99
getStringExtra() 188
getType() 246
getTypeface() 143
getWritableDatabase() 245
GitHub 59
goAsync() 305
Gradle 41, 47, 65
Gradle-Scripts 49
gravity 127
GridLayout 135
Größenangaben 104
GROUP BY
SQL 258
H
Handler 197, 216
HAVING
SQL 258
HAXM 25
Heap 89, 196
hint 151
Hintergrundfarbe 151
Hintergrundprozesse 74
I
IBinder 217
IllegalArgumentException 279
ImageButton 152
Implizite Intents 176
inflate() 255
initLoader() 285
inputType 145, 148
INSERT (SQL) 243
insert()
ContentProvider 279
ContentResolver 271
insertOrThrow() 250

IntelliJ IDEA Community Edition 20
Intent 236
Intent.ACTION_TIME_TICK 306
IntentFilter 211
Intent-Filter 181
Intent-Filter
  Broadcast Receiver 302
Intents 176
IntentService 206
Interpreter 15
interrupt() 199
invalidateOptionsMenu() 171
isChangingConfigurations() 81, 214
isFinishing() 80
isInterrupted() 200
J
JACK 16
Java Beans 121
JAVA_HOME 21, 24
JavaFX 120
Jazelle 14
JetBrains 20
JILL 16
K
Klassen
  anonyme 140
Komplexe Ressourcen 92
Kontextmenü 263
Kontrakt-Klasse 273
Kontrollkästchen 154, 156
L
layout_column 133
layout_gravity 128, 131
layout_height 123
layout_margin 124
layout_span 133
layout_weight 128
layout_width 123
LayoutInflater 121, 230, 255
Layout-Parameter 123
Leere Prozesse 74
LinearLayout 50, 113, 125
ListActivity 165, 253
ListFragment 221
ListView 163
Loader-API 283
LocalBroadcastManager 210
Log 83
LogCat 82
Lokaler Dienst 205
M
Main-Thread 194
makeText() 172
margins 124
Maschinencode 14
match() 277
match_parent 124
maxLength 147
MediaPlayer 110
Mediendateien 110
Message 218
MIME-Typ 179, 271
Module 48
MotionEvent 138
Multitasking 194
Multithreading 194, 245
N
Neun-Feld-Bitmap 106, 145
newView() 255
nine-patch bitmap 106
O
onActivityCreated() 286
onActivityResult() 187
onBackPressed() 80
onBind() 216
onCheckedChanged() 155
OnCheckedChangeListener 155
onContentChanged() 286
onContextItemSelected() 174, 264
onCreate() 78
  ContentProvider 278
  SQLiteOpenHelper 245
onCreateContextMenu() 173, 264
onCreateLoader() 284
onCreateOptionsMenu () 170
onCreateView() 230
onDestroy() 81, 199
onDowngrade() 245
onListItemClick() 230
onLoaderReset() 285
onLoadFinished() 285
onOptionsItemSelected() 171, 296
onPause() 80, 306
onPostExecute() 204
onPreExecute() 203
onPrepareOptionsMenu(Menu) 171
onProgressUpdate() 203
onReceive() 301
onRestoreInstanceState() 79
onResume() 79, 306
onSaveInstanceState() 79

onServiceConnected() ... 215
onStart() ... 79
onStop() ... 81
onUpgrade() ... 245
Optionsschalter ... 159
ORDER BY
SQL ... 258
ORDER BY (SQL) ... 243
P
PackageManager ... 191
padding ... 124
parse() ... 191, 268
parseColor() ... 295
Passwort ... 147
Paused ... 76
peekService() ... 305
PendingIntent ... 193
performClick() ... 138
Permissions
Broadcast Receiver ... 311
Content Provider ... 272
PID ... 88
post() ... 198
Preemtives Multitasking ... 194
Preference ... 288
PreferenceFragment ... 221, 293
PreferenceManager ... 295, 297
Primzahlen ... 117
Process ID ... 88
ProgressBar ... 162, 201
publishProgress() ... 203
putExtra() ... 181, 192
PuTTY ... 243
Q
Qualifizierer ... 95
query() ... 246, 256
ContentProvider ... 278, 279
ContentResolver ... 269
SQLiteDataBase ... 257
R
R 98
R.java ... 99
RadioButton ... 155
RadioGroup ... 155, 159, 160
rawQuery() ... 259
Refaktorisierung ... 62
registerForContextMenu() ... 173, 264
registerReceiver() ... 306, 309
Relationale Datenbanken ... 239
RelativeLayout ... 50, 129
Release-Modus ... 63
Request-Code ... 187
resolveActivity() ... 191
Resources ... 99
Ressourcen ... 91
Ressourcen-IDs ... 98
Running ... 76
runOnUiThread() ... 198
S
Sandkasten ... 74
Schaltflächen ... 151
ScrollView ... 236
SDK-Manager ... 36
sendBroadcast() ... 307
sendOrderedBroadcast() ... 308
sendStickyBroadcast() ... 309
Service ... 69, 205
ServiceConnection ... 215
setArguments() ... 234
setAutoLinkMask() ... 143
setBackgroundColor() ... 295
setColumnShrinkable() ... 134
setColumnStretchable() ... 134
setComponent() ... 186, 308
setContentView() ... 78, 121
setData() ... 218
setFilters() ... 147
setFlags() ... 181
setFreezesText() ... 142
setInputType() ... 147
setMax() ... 163
setOnItemSelectedListener() ... 161
setPackage() ... 312
setProgress() ... 163
setText() ... 142
setTextSize() ... 143, 236
Settings ... 193
setTypeface() ... 142, 157
setVisibility() ... 163
Shared Preferences ... 288
shared_prefs ... 291
SharedPreferences ... 296
showAsAction ... 167
shrinkColumns ... 134
Sicherheit ... 74
Sichtbare Prozesse ... 73
SimpleCursorAdapter ... 259, 270
Singleton-Muster ... 246
Space ... 136
Späte Bindung ... 177
Speicherklassen (SQLite) ... 240
Spinner ... 160
SQL ... 239
SQL-Injektion ... 258

SQLite 239
sqlite3 242
SQLiteDatabase 244, 248, 250
SQLiteDataBase 257
SQLiteOpenHelper 244, 250
Stack 196
startActivity() 185, 190
startActivityForResult() 187, 286
startForeground() 219
startManagingCursor() 256, 285
startService() 305
Steuerelemente 119
Sticky Broadcast 309
Stopped 77
stretchColumns 134
string-array 229
swapCursor()swapCursor() 285
Switch 158
T
TableLayout 133
Task 71
textAllCaps 54, 115
TextEdit 160
textMultiLine 148
textSize 143
textStyle 143
TextView 142
Thema 97
Thread 196
Toast-Meldungsfenster 172, 251
ToggleButton 158
Tween-Animation 109
Typaffinität (SQLite) 240
typeface 142
Typeface 142
U
UI-Thread 194
Umschalter 154
unbindService() 214, 215
Uniform Resource Identifier 179
UPDATE (SQL) 243
update()
  ContentProvider 280
  ContentResolver 271
Uri 190, 268
URI 275
UriMatcher 277
Uri-Objekt 179, 191, 268
UserDictionary.Words 273
uses-permission 272, 311
V
VCS 59
ViewGroup 122, 125
virtuellen Maschine 14
Vordergrundprozesse 73
W
WebViewFragment 222
WHERE
  SQL 257
withAppendedId() 268
wrap_content 124
Z
Zustände einer Aktivität 76