

# Zusammenfassung MOBPRO FS2018

Alex Neher

June 27, 2018

## Inhalt

<b>1</b>	<b>Grundlagen</b>	<b>2</b>
1.1	Komponenten . . . . .	2
1.2	Android Manifest . . . . .	2
1.3	Intents . . . . .	2
1.4	Lebenszyklus . . . . .	3
<b>2</b>	<b>Benutzerschnittstellen</b>	<b>4</b>
2.1	Layouts . . . . .	4
2.2	Ressourcen . . . . .	4
2.3	Interaktion mit dem User . . . . .	4
2.3.1	Options Menu . . . . .	4
2.3.2	Toast . . . . .	5
2.3.3	Dialog . . . . .	5
2.3.4	Notifications . . . . .	6
2.4	Adapter . . . . .	6
2.5	Kontextspeicherung . . . . .	6
<b>3</b>	<b>Persistenz</b>	<b>7</b>
3.1	Dateisysteme . . . . .	7
3.2	SQLite . . . . .	8
<b>4</b>	<b>Content Providers</b>	<b>8</b>
<b>5</b>	<b>Kommunikation</b>	<b>9</b>

# 1 Grundlagen

## 1.1 Komponenten

Ein Android-App besteht aus *Komponenten*. Es gibt vier verschiedene Arten von Komponenten:

**Activity:** Eine View. Eine Komponente, die etwas macht und ein UI hat. (Mail-App)

**Service:** Eine Activity ohne UI. Eine Komponente, die etwas im Hintergrund ausführt. (Musik-player im Hintergrund)

**Broadcast Receiver:** Event-Listener, der auf Broadcasts und Intents hört und antwortet

**Content Provider:** Ermöglicht den Datenaustausch zwischen Applikationen (Mail App darf Bilder von der Galerie auswählen)

## 1.2 Android Manifest

Alle Komponenten müssen im **Android Manifest** deklariert werden. Das Manifest enthält alle Komponenten der App, welche Intents gesendet und empfangen werden können, welche Permissions die App benötigt und so weiter

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3 package="ch.hslu.mobpro.firstapp" >
4
5     <application
6         android:allowBackup="false"
7         android:icon="@drawable/ic_launcher"
8         android:label="@string/app_name"
9         android:supportsRtl="true"
10        android:theme="@style/AppTheme" >
11        <activity android:name="ch.hslu.mobpro.firstapp.MainActivity"
12            android:label="FirstApp" >
13            <intent-filter>
14                <action android:name="android.intent.action.MAIN" />
15                <category android:name="android.intent.category.LAUNCHER" />
16            </intent-filter>
17        </activity>
18        <activity android:name="ch.hslu.mobpro.firstapp.LifecycleLogActivity" />
19        <activity android:name="ch.hslu.mobpro.firstapp.QuestionActivity" />
20    </application>
21 </manifest>
```

Listing 1.1: Einfaches Android-Manifest

## 1.3 Intents

Der Wechsel zwischen Komponenten wird mittels **Intents** realisiert. Intents sind eine *offene Kommunikation*. Das heisst, der Sender weiss nicht, ob der Empfänger der Kommunikation überhaupt existiert.

Es wird unterschieden zwischen *impliziten* und *expliziten* Intents. Implizite Intents rufen gezielt eine Klasse auf, während implizite Intents einfach sagen, was getan werden muss (z.B. "ruf mir einen Browser auf, aber es wird nicht spezifiziert, welchen Browser genau).

```

1 Intent myIntent = new Intent(this, Receiver.class);
2 intent.putExtra("msg", "Hello World");
3 startActivity(myIntent);

```

Listing 1.2: Beispiel eines expliziten Intents

```

1 Intent browserCall = new Intent();
2 browserCall.setAction(Intent.ACTION_VIEW);
3 browserCall.setData(Uri.parse("http://www.hslu.ch));
4 startActivity(browserCall);

```

Listing 1.3: Beispiel eines impliziten Intents

```

1 Intent intent = getIntent();
2 String msg = intent.getExtras().getString("msg");
3 displayMessage(msg);

```

Listing 1.4: Empfangen und Auswerten eines Intents

Es kann auch asynchron eine Activity aufgerufen werden, die anschliessend ein Resultat zurückliefert, welches ausgewertet wird:

```

1 //MainActivity
2 Intent intent = new Intent(this, QuestionActivity.class);
3 intent.putExtra("question", "Und wie läuft's so mit der Androidprogrammierung?");
4 startActivityForResult(intent, MY_REQUEST_CODE);
5
6 //QuestionActivity
7 Intent answerData = new Intent();
8 answerData.putExtra("answer", answer);
9 setResult(RESULT_OK, answerData);
10 finish();
11
12 //MainActivity
13 @Override
14 protected void onActivityResult(int requestCode, int resultCode, Intent data) {
15 //resultat verarbeiten
16 }

```

Listing 1.5: Beispiel eines asynchronen Methodenaufrufs

## 1.4 Lebenszyklus

Eine App kann prinzipiell drei states haben:

**running:** App läuft im Fokus und Vordergrund

**paused:** App läuft im Vordergrund aber nicht mehr im Fokus (z.B. wegen Popup)

**stopped:** App läuft im Hintergrund weiter

Es gibt verschiedene EventListener die bei einer Statusänderung aufgerufen werden können:

- onCreate()
- onPause()
- onStart()
- onDestroy()
- onResume()
- onStop()

## 2 Benutzerschnittstellen

### 2.1 Layouts

Layouts sind XML-Dateien, die in der `onCreate()`-Methode einer Activity geladen werden. Es gibt grundsätzlich drei verschiedene Layout-Optionen:

**Linear Layout:** Komponenten werden in Zeilen oder Spalten linear angeordnet

**Constraint Layout:** Komponenten werden "aneinandergebunden". Man kann also sagen "Komp. A ist rechts von B und unterhalb von Komp. C"

**ScrollView:** Möglichkeit für lange Layouts, die länger sind als der Screen. Erlauben nur ein Child (z.B. LinearLayout)

Das Layout kann entweder als XML definiert werden (einfacher und häufiger) oder als Java-Code. Events können direkt ins XML eingebettet werden, oder sie können über die bereits bekannte Methode des Event-Listeners in Java implementiert werden.

```
1 <Button
2     android:id="@+id/main_button_startBrowser"
3     android:layout_width="fill_parent"
4     android:layout_height="wrap_content"
5     android:text="@string/main_button_text_startBrowser"
6     android:onClick="startBrowser" //Einbettung des Event-Listeners
7     android:paddingBottom="@dimen/activity_horizontal_margin"
8 />
```

Listing 2.1: Button-Definition in XML

```
1 Button button = (Button) findViewById(R.id.main_button_startBrowser)
2 button.setOnClickListener(new OnClickListener(){
3     @Override
4     public void onClick(View v){
5         startBrowser();
6     }
7 })
```

Listing 2.2: Button Event-Definition in Java

### 2.2 Ressourcen

Ressourcen wie Strings, Layouts, Bilder, Arrays etc. werden im `/res`-Ordner abgelegt. In der XML-Datei kann über den `@`-Operator darauf zugegriffen werden (`@string/value1`). In Java wird über die automatisch generierte R-Klasse auf Ressourcen zugegriffen (`R.layout.activity_main`). Es können mehrere Layout- oder String-Dateien erstellt werden z.B. für Portrait und Landscape Mode oder für verschiedene Sprachen.

### 2.3 Interaktion mit dem User

#### 2.3.1 Options Menu

Seit Android 3 oder so gibt es rechts oben in einer App normalerweise drei Punkte, über welche das Options-Menu aufgerufen werden kann.

Das Options Menu-Layout wird wie alle anderen Layouts über ein XML definiert, welches anschliessend im `/res`-Ordner abgelegt wird.

```

1 <menu xmlns:android="http://schemas.android.com/apk/res/android"
2   xmlns:tools="http://schemas.android.com/tools" tools:context=".MainActivity">
3   <item
4     android:id="@+id/main_menu_finish"
5     android:title="@string/menu_finish">
6   </item>
7   <item
8     android:id="@+id/main_menu_values"
9     android:title="@string/menu_ShowValues">
10  </item>
11 </menu>

```

Listing 2.3: Beispiel eines Menu Layouts

Das Menu wird anschliessend mit dem MenuInflater 'aufgeblasen'

```

1 @Override
2 public boolean onCreateOptionsMenu (Menu menu){
3   suuper.onCreateOptoinsMenu(menu);
4   MenuInflater inflater = getManuInflater();
5   inflater.inflate(R.menu.menu_main, menu);
6   return true;
7 }

```

Listing 2.4: Beispiel des Menu-Inflatoren

### 2.3.2 Toast

Ein Toast ist eine kleine Meldung auf dem Bildschirm, die dem User etwas mitteilt. Der User kann aber nicht mit dem Toast interagieren.

```

1 Toast toast = Toast.makeText(context, "Toast Bsp", Toast.LENGTH_LONG).show();

```

Listing 2.5: Toast-Beispiel

### 2.3.3 Dialog

Der Dialog oder Alert ist ein Popup, der dem User eine Information mitteilt und er darauf reagieren muss.

```

1 AlertDialog.Builder builder = new AlertDialog.Builder(this);
2 builder.setTitle(R.string.dialog_fire_missiles_title)
3   .setMessage(R.string.dialog_fire_missiles)
4   .setPositiveButton(R.string.fire, new DialogInterface.OnClickListener() {
5     public void onClick(DialogInterface dialog, int id) {
6       // FIRE ZE MISSILES!
7     }
8   })
9   .setNegativeButton(R.string.cancel, new DialogInterface.OnClickListener() {
10    public void onClick(DialogInterface dialog, int id) {
11      // User cancelled the dialog
12    }
13  });
14 AlertDialog dialog = builder.create();
15 dialog.show();

```

Listing 2.6: Alert-Beispiel

### 2.3.4 Notifications

Kommen später

## 2.4 Adapter

Adapter nehmen, wie bereits aus APPE/VSK bekannt, Daten, konvertieren sie in ein anderes Format und übergeben sie dem Zielkomponenten.

```
1 String[] myArray = new String[]{"Fanta", "Cola", "Eistee"};
2 ArrayAdapter<String> adapter = new ArrayAdapter<String>(this.
    android.R.layout.of.spinner, myArray);
3 this.setListAdapter(adapter);
```

Listing 2.7: Array-Adapter um String Array in Spinner zu füllen

Alternativ kann man sich den Adapter auch sparen und das direkt im XML des Komponenten (z.B. Spinner) machen:

```
1 <Spinner
2     android:id="@+id/main_spinner"
3     android:layout_width="match_parent"
4     android:layout_height="wrap_content"
5     android:entries="@array/itCourses" /> //Füllt das Array in den Spinner
6 </Spinner>
7
8 //in array.xml
9 <resources>
10     <string-array name="itCourses">
11         <item>MOBPRO</item>
12     </string-array>
13 </resources>
```

Listing 2.8: Spinner-Layout mit direkten Füllen

## 2.5 Kontextspeicherung

Der Zustand der App geht verloren, wenn die App gestoppt oder pausiert wird, oder auch wenn z.B die Bildschirmorientierung geändert wird. Man kann aber bestimmte Daten kurzzeitig im Memory speichern und sie anschliessend wieder abrufen:

```
1 //Speichern
2 @Override
3 protected void onSaveInstanceState(Bundle outState){
4     outState.putInt(KEY, value);
5     super.onSaveInstanceState(outState);
6 }
7
8 //Abrufen
9 @Override protected void onRestoreInstanceState(Bundle savedInstanceState){
10     super.onRestoreInstanceState(savedInstanceState);
11     value = savedInstanceState.getInt(KEY);
12 }
```

Listing 2.9: Abspeichern und Abrufen von Values im Memory

## 3 Persistenz

Es wird grundsätzlich zwischen drei Arten von Präferenzen unterschieden:

**Default Shared Preferences:** `getDefaultSharedPreferences(this)`. Für die gesamte App

**Shared Preferences:** `getSharedPreferences(name, mode)`. Beliebige viele Präferenzen pro App mit je einem eigenen Namen

**Private Preferences:** `getPreferences(mode)`. Für die aktuelle Aktivität

```
1 final SharedPreferences preferences = getPreferences(MODE_PRIVATE);
2 final int newResumeCount = preferences.getInt(COUNTER_KEY, 0)+1;
3 final SharedPreferences.Editor editor = preferences.edit();
4 editor.putInt(COUNTER_KEY, newResumeCount);
5 editor.apply();
```

Listing 3.1: Holen des ResumeCount und um eins erhöht wieder abspeichern

### 3.1 Dateisysteme

Dateien können privat oder öffentlich sein. Private Dateien sind ausschliesslich aus der Applikation heraus oder über Content Providers zugreifbar und werden im Applikationsverzeichnis abgelegt. Öffentliche Daten werden auf der SD-Karte oder dem internen Filesystem abgelegt, wo jeder Zugriff drauf haben kann.

```
1 Writer writer = null;
2 try{
3     writer = new BufferedWriter(new FileWriter(outFile));
4     writer.write(text);
5     return true;
6 } catch (final IOException ex){
7     //catch Exception
8 }
```

Listing 3.2: File schreiben

Um aufs PUBLIC Filesystem (SD-Karte) zugreifen zu können, muss zuerst eine Berechtigung dafür erlangt werden. Alle Berechtigungen, die eine App benötigt, werden im Android Manifest festgelegt

```
1 <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

Listing 3.3: Berechtigung im Android Manifest für das Lesen von der SD-Karte

```
1 int grant = checkSelfPermission(Manifest.permission.WRITE_EXTERNAL_STORAGE);
2 if (grant != PackageManager.PERMISSION_GRANTED) {
3     requestPermissions(new String[]{Manifest.permission.WRITE_EXTERNAL_STORAGE}, 23);
4 } else {
5     writeSDCard();
6 }
```

Listing 3.4: Checken, ob man die Berechtigung hat und wenn nicht, Berechtigung anfragen

Die Berechtigungen werden über eine Callback-Methode ausgewertet:

```

1 public void onRequestPermissionsResult(int requestCode, String[] permissions, int[]
  grantResults){
2     switch (requestCode){
3         case 24:
4             if(grantResults.length > 0 && grantResults[0] !=
              PackageManager.PERMISSION_GRANTED){
5                 Toast.makeText(this, "Permission " + permissions[0] + " denied!",
                  Toast.LENGTH_SHORT).show();
6             } else {
7                 readSDCard();
8             }

```

Listing 3.5: Verarbeitung der Permission-Grants

## 3.2 SQLite

SQLite ist ein open source relationales Datenbank-Management System, welches für Android optimiert ist. Man hat pro App beliebig viele Datenbanken, jedoch nur eine Datei pro Datenbank. Über der DB gibt es noch eine Abstraktionsebene, den sog. *Room*. Der Programmierer greift jedoch nur via `dbAdapter()` auf die Datenbank bzw. den Room zu:

```

1 dbAdapter = new DbAdapter(this);
2 dbAdapter.open();
3 Note note = dbAdapter.getNote(17);

```

Listing 3.6: Anwendung des dbAdapters

## 4 Content Providers

Wie bereits im ersten Kapitel erwähnt, ermöglicht der Content Provider den Austausch von Daten über Applikationsgrenzen hinweg. Dies indem er mittels eindeutigen URIs auf Items (`content://anwendung/gruppe/item`) oder Verzeichnisse (`content://anwendung/gruppe`) zugreift. Das Android-OS liefert bereits einige in-house Content Provider wie z.B. Kontakte, Kalender o.ä.

Der Zugriff auf solche Content Providers läuft immer über den `ContentResolver` (`Context.getContentResolver()`) und gibt stets einen *Cursor* zurück:

```

1 public void showSMSList(final View view){
2     final Cursor cursor = getContentResolver().query(
3         Telephony.Sms.Inbox.CONTENT_URI,
4         new String[]{
5             Telephony.Sms.Inbox._ID, //SMS-Id
6             Telephony.Sms.Inbox.BODY //SMS-Text
7         },
8         null, //selection
9         null, //selection args
10        null //sort order
11    );
12 }

```

Listing 4.1: Auslesen aller SMS mittels Content Provider

Man kann sich auch einen eigenen Content Provider schreiben, wenn man auf andere Daten zugreifen will, die nicht von den systemeigenen Providern abgedeckt werden.



- Klasse muss von `android.content.ContentProvider` abgeleitet sein
- Klasse muss bei App-Start initiiert werden (z.b. in der `onCreate()`-Methode)
- CRUD-Methoden (zumindest die, die benötigt werden) müssen implementiert werden
- Es muss entschieden werden, ob der Provider exportiert werden soll oder nicht (exportiert = auch andere Anwendungen können auf ihn zugreifen und Daten holen)

## 5 Kommunikation