Zusammenfassung MOBPRO FS2018

Alex Neher

June 28, 2018

Inhalt

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. (Musikplayer 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

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="ch.hslu.mobpro.firstapp" >

   <application
      android:allowBackup="false"
      android:icon="@drawable/ic_launcher"
      android:label="@string/app_name"
      android:supportsRtl="true"
      android:theme="@style/AppTheme" >
      <activity android:name="ch.hslu.mobpro.firstapp.MainActivity"
         android:label="FirstApp" >
         <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
         </intent-filter>
      </activity>
      <activity android:name="ch.hslu.mobpro.firstapp.LifecycleLogActivity" />
      <activity android:name="ch.hslu.mobpro.firstapp.QuestionActivity" />
   </application>
</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).

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

Listing 1.2: Beispiel eines expliziten Intents

```
Intent browserCall = new Intent();
broserCall.setAction(Intent.ACTION_VIEW);
browserCall.setData(Uri.parse("http://www.hslu.ch));
startActivity(browserCall);
```

Listing 1.3: Beispiel eines impliziten Intents

```
Intent intent = getIntent();
String msg = intent.getExtras().getString("msg");
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:

```
//mainActivity
Intent intent = new Intent(this, QuestionActivity.class);
intent.putExtra("question", "Und wie läufts so mit der Androidprogrammierung?");
startActivityForResult(intent, MY_REQUEST_CODE);

//QuestionActivity
Intent answerData = new Intent();
answerData.putExtra("answer", answer);
setResult(RESULT_OK, answerData);
finish();

//MainActivity
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
//resultat verarbeiten
}
```

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()`
- `onStopped()`

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 "aneinadergebunden". 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 entwder 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.

```
<Button
   android:id="@+id/main_button_startBrowser"
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"
   android:text="@string/main_buttontext_startBrowser"
   android:onClick="startBrowser" //Einbettung des Event-Listeners
   android:paddingBottom="@dimen/activity_horizontal_margin"
/>
```

Listing 2.1: Button-Definition in XML

```
Button button = (Button) findViewById(R.id.main_button_startBrowser)
button.setOnClickListener(new OnClickListener(){
   @Override
   public voidonClick(View v){
      startBrowser();
   }
})
```

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 anschliessned im `/res`-Ordner abgelegt wird.

```xml
<menu xmlns:android="http://schemas.android.com/apk/res/android"
   xmln:tools="http://schemas.android.com/tools" tools:context=".MainActivity">
   <item
      android:id="@+id/main_menu_finish"
      android:title="@string/menu_finish">
   </item>
   <item
      android:id="@+id/main_menu_values"
      android:title="@string/menu_ShowValues">
   </item>
</menu>
```

Listing 2.3: Beispiel eines Menu Layouts

Das Menu wird anschliessend mit dem `MenuInflater` 'aufgeblasen'

```
@Override
pulic boolean onCreateOptionsMenu (Menu menu){
   suuper.onCreateOptoinsMenu(menu);
   MenuInflater inflater = getManuInflater();
   inflater.inflate(R.menu.menu_main, menu);
   return true;
}
```

Listing 2.4: Beispiel des Menu-Inflators

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.

```
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.

```
AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setTitle(R.string.dialog_fire_missiles_title)
   .setMessage(R.string.dialog_fire_missiles)
   .setPositiveButton(R.string.fire, new DialogInterface.OnClickListener() {
      public void onClick(DialogInterface dialog, int id) {
         // FIRE ZE MISSILES!
      }
   })
   .setNegativeButton(R.string.cancel, new DialogInterface.OnClickListener() {
      public void onClick(DialogInterface dialog, int id) {
         // User cancelled the dialog
      }
   });
AlertDialog dialog = builder.create();
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.

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

```
<Spinner
   android:id="@+id/main_spinner"
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:entries="@array/itCourses" /> //Füllt das Array in den Spinner
</Spinner>

//in array.xml
<resources>
   <string-array name="itCourses">
      <item>MOBPRO</item>
   </string-array>
</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:

```
//Speichern
@Override
protected void onSaveInstanceState(Bundle outState){
   outState.putInt(KEY, value);
   super.onSaveInstanceState(outState);
}

//Abrufen
@Override protected void onRestoreInstanceState(Bundle savedInstanceState){
   super.onRestoreInstanceState(savedInstanceState);
   value = savedInstanceState.getInt(KEY);
}
```

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)`. Beliebig viele Präferenzen pro App mit je einem eigenen Namen

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

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

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

3.1 Dateisysteme

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

```
Writer writer = null;
try{
   writer = new BufferedWriter(new FileWriter(outFile));
   writer.write(text);
   return true;
} catch (final IOException ex){
   //catch Exception
}
```

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

```
<ises-permission android:name="android.permission.WRITE_EXTERAL_STORAGE"/>
```

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

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

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

Die Berechtigungen werden über eine Callback-Methode ausgewertet:

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

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:

```
dbAdapter = new DbAdapter(this);
dbAdapter.open();
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 eindeuten 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:

```
public void showSMSList(final View view){
   final Cursor cursor = getContentResolver().query(
      Telephony.Sms.Inbox.CONTENT_URI,
      new String[]{
         Telephony.Sms.Inbox._ID, //SMS-Id
         Telephony.Sms.Inbox.BODY //SMS-Text
      },
      null, //selection
      null, //selection args
      null  //sort order
   );
}
```

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

5.1 HTTP

Die Kommunikation über HTTP läuft über `java.net.HttpURLConnection`, der aber davon abhängt, dass man bereits eine URL (`java.net.URL`) hat, auf die er verbinden kann:

```
URL url = new URL("www.hslu.ch");
HttpURLConnection httpConnection = (HttpURLConnecton) url.openConnection();
httpConnection.setInstanceFollowRedirects(true);
httpConnection.connect();
if(httpConnection.getResponseCode() == HttpURLConnection.HTTP_OK){
   InputStream content = httpConnection.getInputStream();
   //content verarbeiten
}
```

Exkurs: Stream zu Text verarbeiten
Mithilfe eines `BufferedReader` kann der `InputStream` der HTTP-Connection zeilenweise ausgelesen werden:

```
private String readText(InputStream in) throws IOException{
   StringBuilder text = new StringBuilder();
   String line;
   BufferedReader reader = new BufferedReader(new InputStreamReader(in));
   while((line = reader.readLine()) != null){
      text.append(line);
      text.append("\n");
   }
   reader.close();
   return text.toString();
}
```

5.2 Sockets

Im Gegensatz zu HTTP sind Sockets **zustandsbehaftet**, sie merken sich also den Kontext (z.B. Logins). Die Kommunikation basiert auch auf keinem Protokoll wie HTTP, sondern man sendet einfach Datenströme hin und her. Ebenfalls bleibt die Client-Server Verbindung so lange offen, bis sie einer der beiden abbricht.

Der Ablauf ist prinzipiell so, dass der Server einen Socket hat, auf welchen alle Clients verbinden (quasi eine 'Eingangshalle'). Sobald sich ein Client verbunden hat, wird ein Client-Socket erstellt und der Client wird auf diesen Socket weitergeleitet. Somit ist der Hauptsocket wieder frei für weitere ankommende Verbindungen.

Ein Socket wird mittels einer IP-Adresse und einem Port erstellt:

```
socket = new socket(IP, Port);
writer = new BufferedWriter(new OutputStreamWriter(socket.getOutputStream()));
reader = new BufferedReader(new InputStramReader(socket.getInputStream));
```

Listing 5.1: Erstellung eines Sockets und dessen Reader/Writer

6 Nebenläufigkeit

Java ist zwar multithreaded, aber per Default läuft eine App trotzdem nur auf einem Thread: dem **main-Thread**, der auch für das UI verantwortlich ist. Soll heissen, wenn der main-Thread blockiert ist, tut das UI keinen Mucks mehr → UI-Freeze.

Wenn man nun etwas machen will, was evtl. ein bisschen länger dauern könne und man den UI-Freeze verhindern will, gibt es grundsätzlich zwei Methoden:

AsyncTask: Die Operation wird gestartet und im Hintergrund ausgeführt. Der main-Thread wird benachrichtigt, sobald ein Resultat vorliegt. Keine eigenen Threads und somit kein nerviges Thread-Handling

Eigene Threads: Man kreiert eigene Threads, so dass die Applikation multithreaded läuft und der main-Thread so nicht überlastat wird. Manchmal notwendig, da der main-Thread z.B. Network-Connections gar nicht zulässt.

6.1 AsyncTask

Der AsyncTask ist in drei Methoden aufgeteilt:

`doInBackground(Params...)` Lange andauernder Task im Worker-Thread

onProgressUpdate(Progress...) Verarbeitung des Zwischenresultates im main-Thread

onPostExecute(Result) Verarbeitung des Endresultats im main-Thread

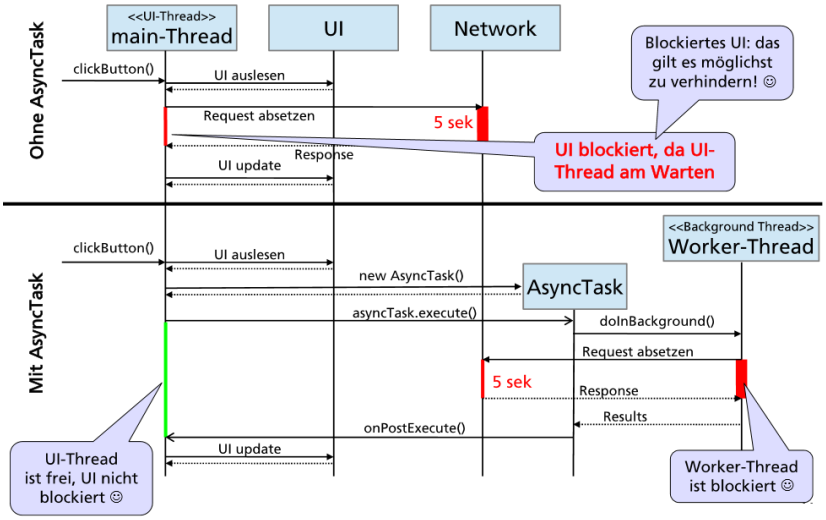

Abb. 6.1: Vergleich mit und ohne AsyncTask

```
                                        Param, Zwischenresultat, Resultat
public class MultiAsyncTask extends AsyncTask<URL, String, Void>
   @Override
   protected Void doInBackground(URL... urls){  //läuft im Worker-Thread
      try{
         for(URL url : urls){
            InputStram in = openHttpConnection(url);
            String text = readText(in);
            in.close();
            Thread.sleep(WAIT_TIME_MILLIS);
            publishingProgress(text) //ruft onProgressUpdate auf
         }
      } catch (IOException){
         //do stuff
      }
   }

   @Override
   protected void onProgressUpdate(String... values){ //läuft auf main-Thread
      super.onProgressValues(values);
   }

   @Override
   protected void onPostExecution(Void result){  //läuft im main-Thread
      AlertDialog.Builder dialogBuilder = new AlertDialog.Builder(mainActivity);
      dialogBiulder
         .//create Dialog with data from URL
   }

}
```

6.2 Threads

Threads sollten (theoretisch) bereits bekannt sein von PRG2 (glaub...), und APPE:

Die Thread Klasse implementiert das Interface `Runnable`, muss also im Konstruktor ein Runnable übergeben werden oder die Methoden `run()`, `start()`, `sleep(long)`, `isAlive()` selbst implementieren

```
public void startDemoThread(View view){
   final Button button = (Button) view;
   if((demoThread == null) || !(demoThread.isAlive())){
      demoThread = createWaitThread(button);
      demoThread.start();
      button.setText("DemoThread läuft..");
   } else {
      Toast.makeText(this, "DemoThrad läuft schon", Toast.LENGTH_SHORT).show();
   }
}

private Thread createWaitThread(final Button button){
   return new Thrad("hsluDemoThread"){ //<-- Thread-Name
      @Override
      public void run(){
         final Runnable doneRunnable = new Runnable(){
            @Override
```

```
18              public void run(){
19                 button.setText("DemoThread starten");
20              }
21          };
22          try{
23              Thread.sleep(WAITING_TIME_MILLI) //<-- main-Thread blockiert
24              MainActivity.this.runOnUiThread(doneRunnable); //
                    Callback-Benachrichtigung des main-Threads
25          }
26       }
27    }
28 }
```

7 Services

Wie bereits in Kap. 1 erklärt, ist ein Service eigentlich eine Activity, die ohne UI, sprich im Hintergrund durchgeführt wird. Ebenfalls bietet ein Service die Möglichkeit, gewisse Funktionalitäten als API zu exportieren und sie so anderen Apps anzubieten.

Ein Service läuft aber weder auf einem eigenen Thread, noch auf einem eigenen Prozess (ausser man definiert ihn explizit so, z.B. für lange Operationen)

Services können entweder über `startService()` (asynchron) oder `bindService()` (synchron) aufgerufen werden:

`startService()`	`bindService()`
`onCreate()` Bei Erzeugung	`onCreate()` Bei Erzeugung
`onStartCommand()` Auftragsbehandlung	`onBind()` Bei Verbindung mit Komponente
	`onUnbind()` Bei Beednung der Verbindung
`onDestroy()` Bei Beendung (durch System, Service, Applikation oder System)	`onDestroy()` Bei Beendung (durch System, Service, Applikation oder System)

```
1  //Service starten/Auftrag erteilen
2  Intent musicService = new Intent(this, MyMusicService.class);
3  musicService.putExtra("audioFileUrl", "http://jurassicpark.org/matingturtles.flac");
4  startService(musicService);
5
6  //Service stoppen
7  stopService(new Intent(this, MyMusicService.class));
8
9  //Service implementieren
10 public class MyMusicService extends Service{
11    @Override
12    public int onStartCommand(Intent intent, int flags, int startId){
13       startMusicStreamingThreadIfNotRunning();
14       String audioFileUrl = intent getStringExtra("audioFileUrl");
15       schedulePlay(audioFileUrl);
16       return START_STICKY;
17    }
18 }
```