

Mobile Programming

Android 4 – Kommunikation & Nebenläufigkeit



kaspar.vongunten@hslu.ch



Inhalt

- Nebenläufigkeit (Concurrency)
 - Der main-Thread & Verhinderung von blockiertem UI
 - Nebenläufige Programmiermodelle
 - AsyncTask
 - Threads
 - JobManager
- Backend-Kommunikation
 - Kommunikation über HTTP
 - HTTP GET/PUT/POST/DELETE
 - Webservices mit XML- und JSON-Daten
 - Typisierte API-Consumption mit Retrofit



<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

Nebenläufigkeit: das Blockierungsproblem

Android und der Main-Thread

- Eine Applikation läuft per Default in genau einem Thread, dem *main*-Thread
 - In diesem main-Thread wird das ganze UI aufgebaut, d.h. **main-Thread = UI-Thread**
- Konsequenz: Wenn main-Thread blockiert ist, friert das UI ein („UI freeze“)... ❄
- Hinweis: UI-Komponenten sind *nicht* Thread-safe!
- D.h. **UI-Zugriff nur aus main-Thread**, sonst Exception:

Caused by: android.view.ViewRootImpl\$CalledFromWrongThreadException: Only the original thread that created a view hierarchy can touch its views.

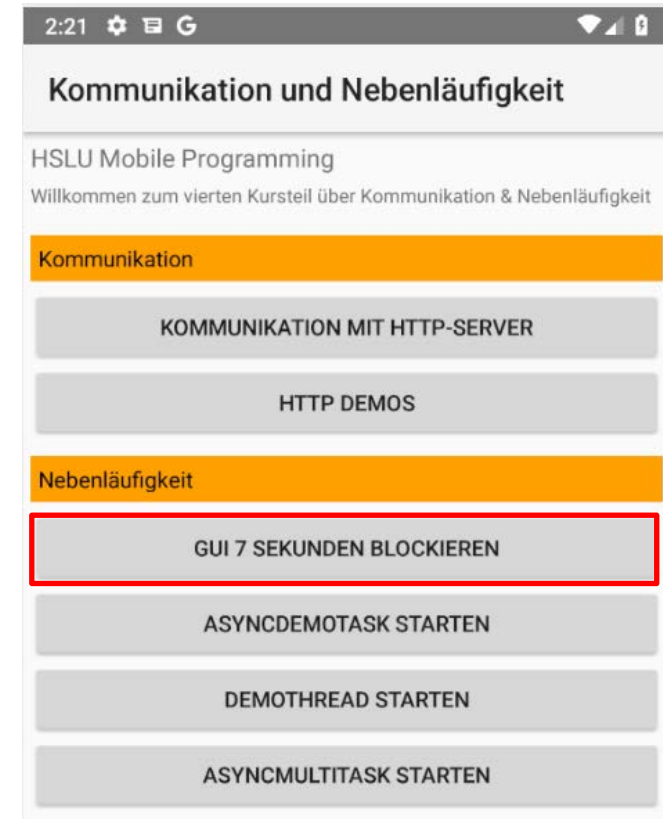
UI & blockierende Methoden...

- Viele Netzwerk-Methoden (und auch andere!) können lange dauern und sind blockierend
 - z.B. `URLConnection.connect()`
 - oder `Bitmap.resize()` oder `Database.open()` ...
- Wird eine solche Methode auf dem main-Thread aufgerufen, so wird dieser blockiert und es werden keine UI-Events mehr verarbeitet
 - D.h. die App reagiert z.B. nicht auf Touch-Events, usw.

 **Das gilt es zu verhindern!**

Demo: Blockierendes UI

- Einfaches Blockieren mittels `Thread.sleep(long time)`
 - in Millisekunden
- Effekt: App reagiert nicht
 - Keine UI-Aktualisierungen
 - Keine Reaktion auf UI-Events
 - „App freeze“ !!

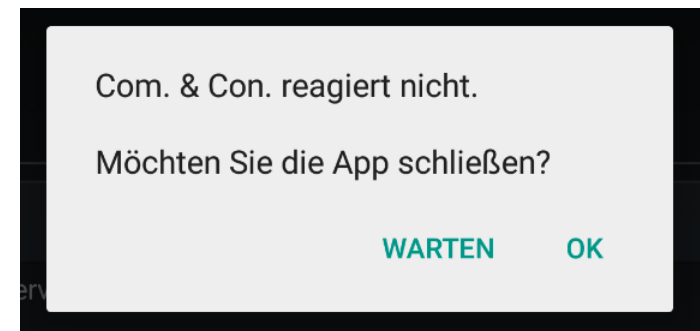


```
public void freeze7Seconds(View view) {  
    try {  
        Thread.sleep(WAITING_TIME_MILLIS);  
    } catch (InterruptedException e) {  
        // ignore  
    }  
}
```

Android-Überwachung: ANR

ANR = Application
Not Responding

- Android System überwacht Ansprechbarkeit (Responsiveness) von Apps
 - Kriterien (Siehe <https://developer.android.com/training/articles/perf-anr.html>)
 - Keine Reaktion auf Input-Event innert 5 Sek
 - Broadcast-Receiver nicht fertig innert 10 Sek
- Möglicher Effekt: ANR-Dialog
 - System-Mechanismus zum stoppen von „bösen“ Apps
- Was tun wir dagegen?



Wie den main-Thread entlasten?

Fragen:

- Ist Hintergrundaufgabe aufschiebbar?
- Hat Task Auswirkungen auf das UI?
- Wartet User auf Resultat?

Herausforderungen:

- Resultat am Ende auf UI-Thread darstellen
- UI noch da?

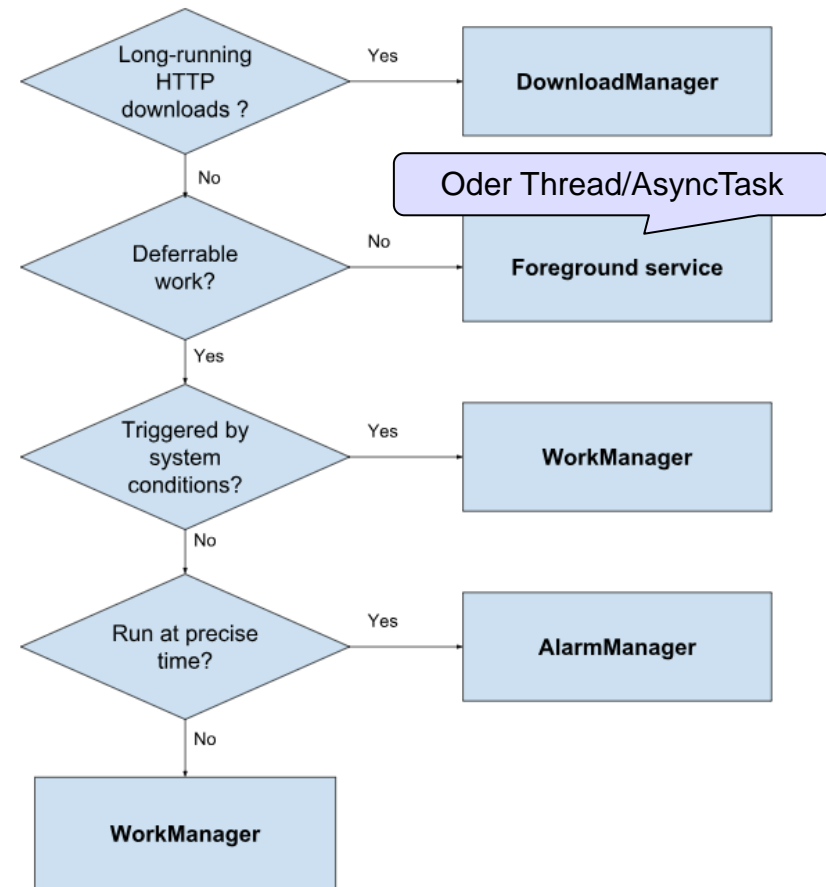


Figure 1. What's the best way to do background work?

<https://developer.android.com/guide/background/>
<https://developer.android.com/topic/performance/threads>

Wenn Aktion nicht aufschiebbar

Kein eigenes Thread-
Handling nötig!

1. Klasse `AsyncTask`

- Einfaches Konstrukt zur Auslagerung von zeitintensiven Aufgaben auf Background-Thread
- Für die meisten Fälle ausreichend

2. Eigene `Thread` Instanzen

- Standardimplementierung von Nebenläufigkeit in Java
- Kann komplex werden: Synchronisierung, Deadlocks, ...

3. Foreground Service

- Hintergrundaktionen, die von User bemerkt werden
- z.B. Music Player

Wie zurück zum UI-Thread?

1. Klasse `AsyncTask`: Vorgesehen durch spezielle Methoden, welche auf `main-Thread` laufen:

- z.B. `onProgressUpdate`, `onPostExecute`, ...
 - Siehe Ausführungen / Demos später

2. Eigener Thread, zwei Möglichkeiten:

- `Activity.runOnUiThread(Runnable action)`
- `View.post(Runnable action)`
- Klasse `android.os.Handler`
 - Benutzt `MessageQueue` von Thread
 - Schauen wir nicht weiter an

Demo folgt

Langandauernde Operationen, z.B. Netzwerk

- Android lässt gewissen Operationen (z.B. Network-API) per Default nicht auf main-Thread zu!

- z.B. Aufruf von `URLConnection.connect()` führt zu **NetworkOnMainThreadException**:

```
android.os.NetworkOnMainThreadException
    at android.os.StrictMode$AndroidBlockGuardPolicy.onNetwork(StrictMode.java:1147)
    at java.net.InetAddress.lookupHostByName(InetAddress.java:418)
    at java.net.InetAddress.getAllByNameImpl(InetAddress.java:252)
```

- Grund: Netzwerk-Kommunikation kann dauern!
 - Netzwerk-Calls nie auf UI-Thread ausführen
 - AsyncTask (oder eigenen Thread) verwenden
 - Und UI-Aktualisierungen in Methoden auf main-Thread!



<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

Nebenläufigkeit: AsyncTask

Die Klasse AsyncTask

```
class MyAsyncTask extends AsyncTask<Params, Progress, Result> {  
    protected Result doInBackground(Params... params) // muss implementiert werden  
    protected void onProgressUpdate(Progress... progress) // optional  
    protected void onPostExecute(Result result) // optional  
    public void execute(Params... params) // starte Task - nur einmal möglich!  
}
```

■ 3 Typ-Parameter (Generics)

- Params = Typ der Input-Elemente, z.B. URL
- Progress = Typ Zwischenresultate, z.B. String (Void falls nicht benutzt)
- Result = Typ des Resultats, z.B. Integer (ggf. Void falls nicht benutzt)

z.B. Links auf Textdateien

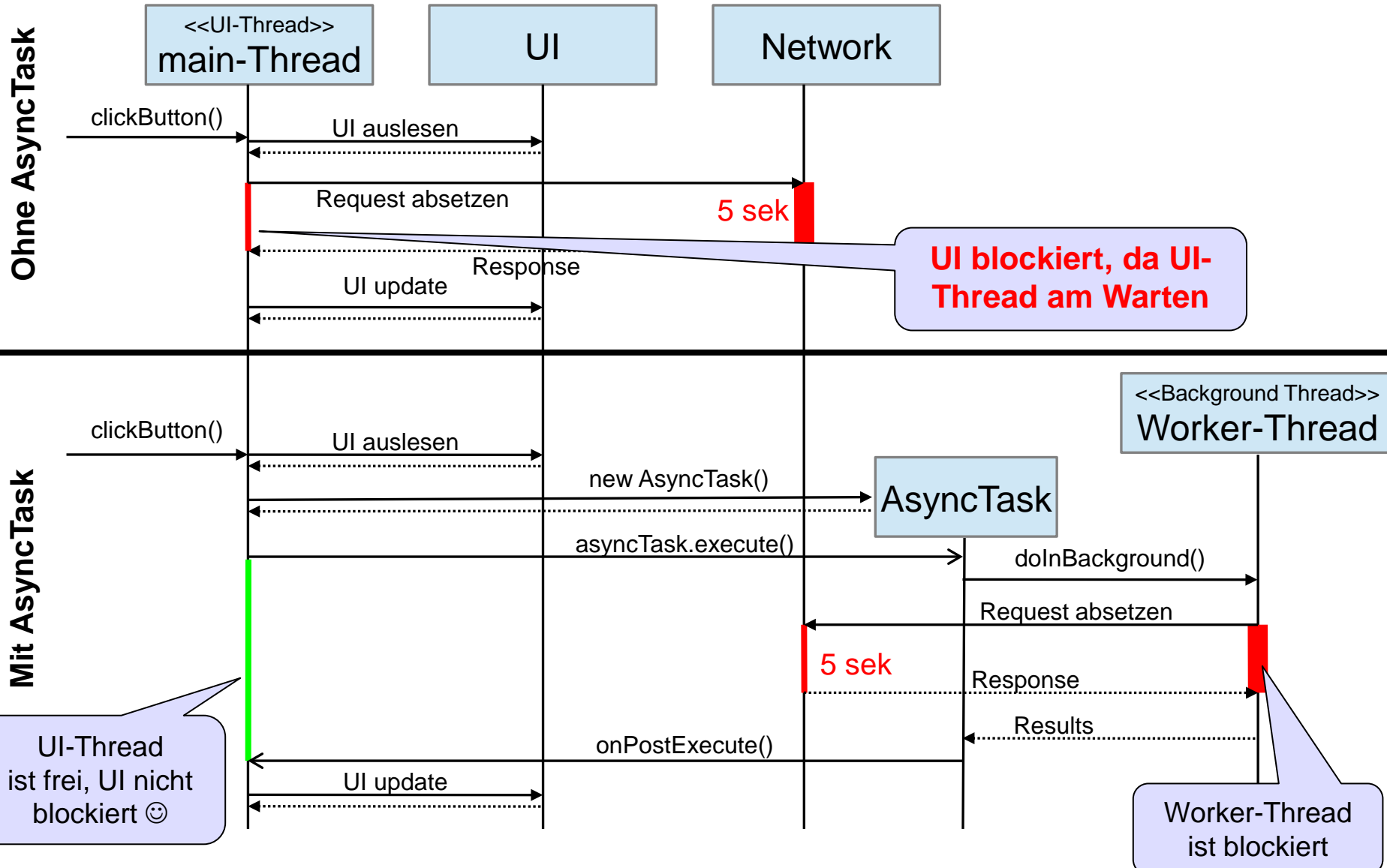
z.B. heruntergeladener Titel

z.B. Anzahl heruntergeladener Titel

■ 3 wichtige Methoden (+ ausführender Thread)

- doInBackground(Params...) - Lange andauernd (Worker-Thread)
- onProgressUpdate(Progress...) - Zwischenresultat verarbeiten (UI-Thread)
- onPostExecute(Result) - Resultat verarbeiten (UI-Thread)

Ablauf ohne & mit AsyncTask



Demo: 7 Sek warten mit eigenem AsyncTask

`public class AsyncDemoTask extends AsyncTask<Integer, Void, String> {`

Parameter-Typ Zwischenresultat-Typ Resultat-Typ

- Hintergrund-Thread 7 Sek blockieren
 - In `AsyncTask.doInBackground(...)`
- Argument vom Typ `Integer`
 - D.h. Übergabe von `Integer[]` an `doInBackground(...)`
- Resultat vom Typ `String`
 - D.h. `doInBackground(...)` gibt `String` zurück
 - Ausgabe in Toast bei Task-Ende
- Nur ein `AsyncTask` soll aktiv sein
 - Ein bisschen Logik dafür selber implementieren...

AsyncTask starten

[AsyncTask läuft...]

UI friert nicht ein! 😊

AsyncTask starten

Resultat = 'Die Parameter waren: 77, 444, 2000, -23, 111'.

DemoThread starten

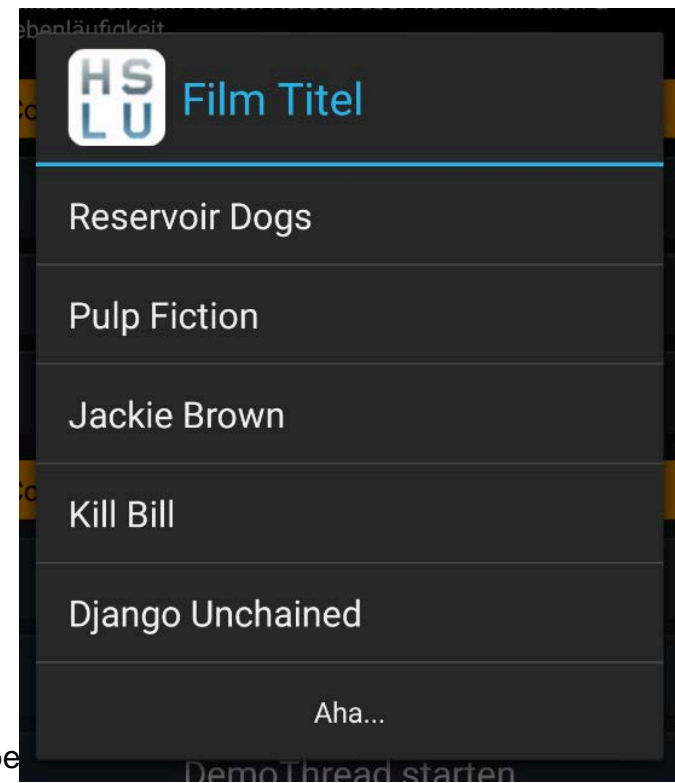
AsyncDemoTask läuft schon!

Demo: AsyncTask mit Zwischenresultaten (Progress)

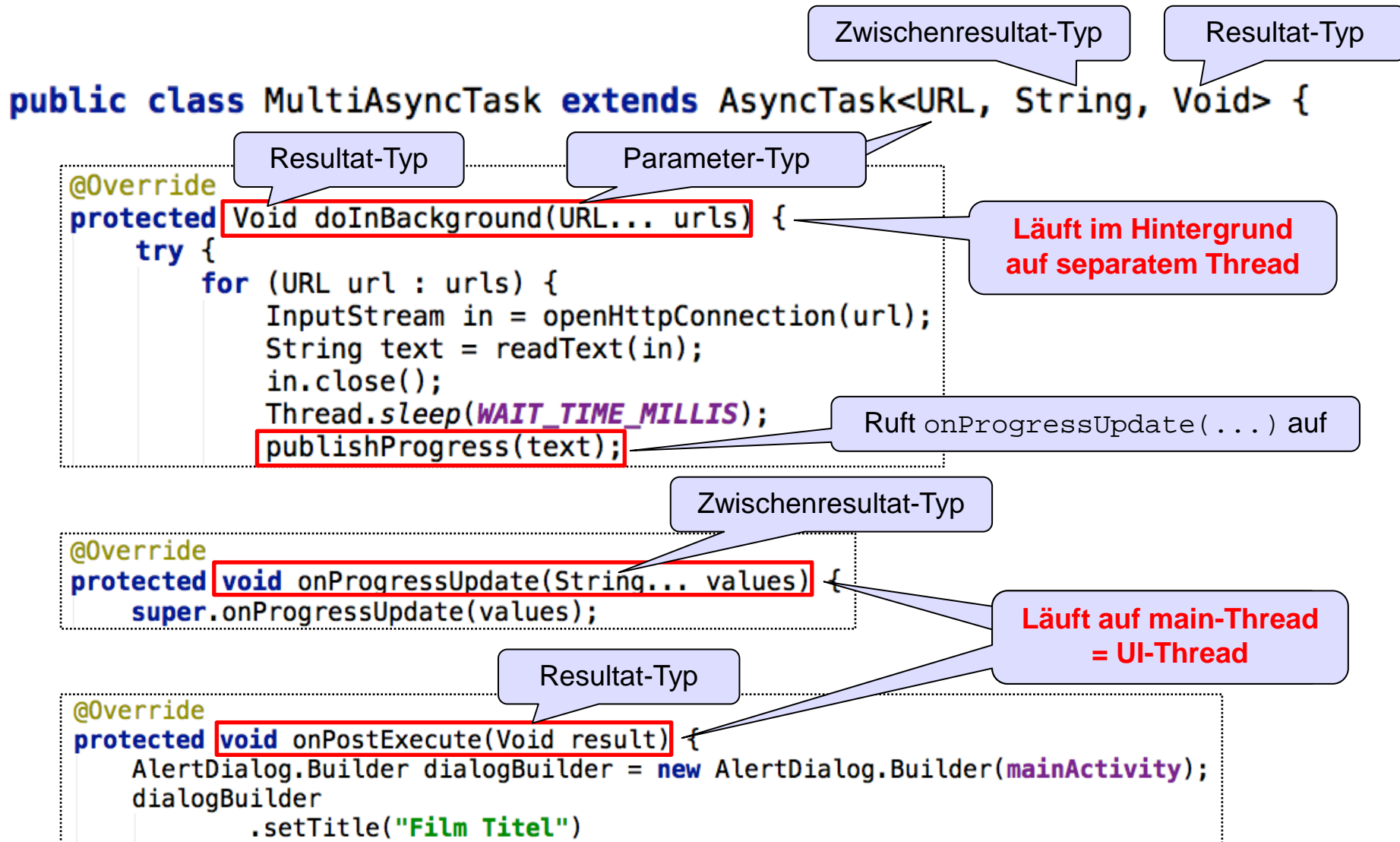
- Idee: Mehrere Textdateien von Server holen:
 - Zwischenresultate in Toast:
 - Benutzen Methoden
 - `publishProgress(...)`
 - `onProgressUpdate(...)`
 - Am Ende Dialog mit allen Texten (Filmtiteln):
 - `onPostExecute(...)`
- Daten-URL: `http://wherever.ch/hslu/titleX.txt`
 - für X von 0..4

AsyncMultiTask starten

DemoThread starten
Neuer Titel: 'Pulp Fiction'.



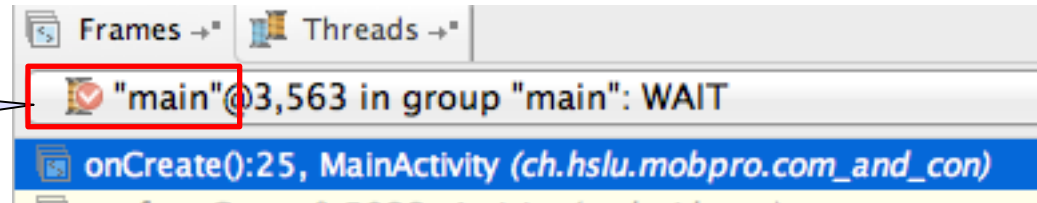
Code-Ausschnitte "Demo mit Zwischenresultaten"



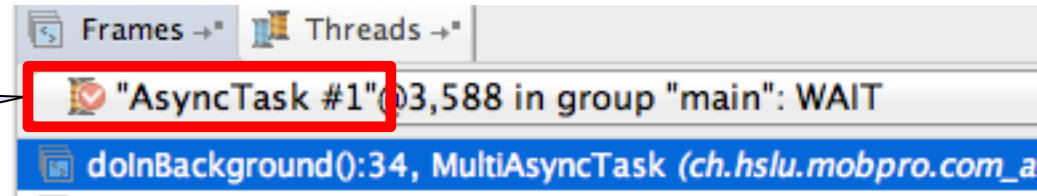
AsyncTask: Methoden & Threads...

...hier leistet der
Debugger gute
Dienste! ☺

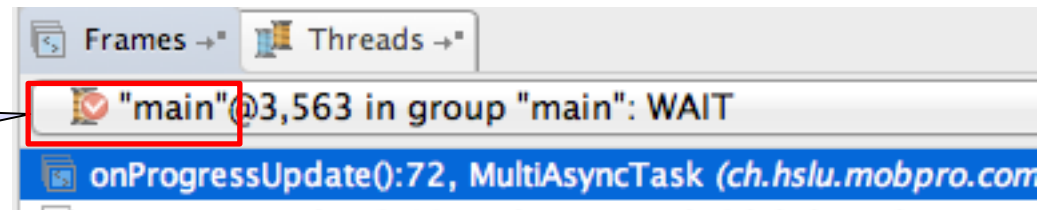
MainActivity läuft im
main-Thread (= UI-Thread)



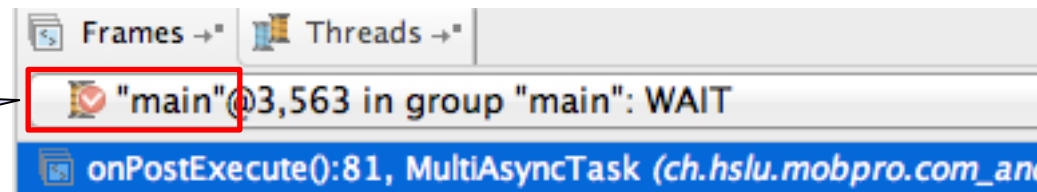
AsyncTask.doinBackground()
läuft in eigenem Thread



AsyncTask.onProgressUpdate()
läuft im main-Thread



AsyncTask.onPostExecute()
läuft im main-Thread



AsyncTasks: Parallele Ausführung oder nicht?

API-Doku-Auszug von `AsyncTask.execute(Params... params):`

*Note: this function schedules the task on a queue for a single background thread or pool of threads depending on the platform version. When first introduced, AsyncTasks were executed serially on a single background thread. Starting with DONUT, this was changed to a pool of threads allowing multiple tasks to operate in parallel. Starting HONEYCOMB, tasks are back to being executed on a single thread to avoid common application errors caused by parallel execution. **If you truly want parallel execution, you can use the `executeOnExecutor(Executor, Params...)` version of this method with `THREAD_POOL_EXECUTOR`; however, see commentary there for warnings on its use.***

Schlussfolgerungen:

1. AsyncTasks werden standardmässig (seriell) durch einen einzelnen Worker-Thread ausgeführt
2. Thread-Pool (parallele Ausführung) durch Angabe Executor möglich
3. **Implementierungs-Details (ggf. auch relevante!) können ändern...**

Die Plattform

ÜBERSICHT

VERSIONEN

TECHNOLOGIE

BIBLIOTHEKEN

KOTLIN

Versionen

Pie

Übersicht

[Funktionen und APIs](#)

Verhaltensänderungen für alle Apps
Verhaltensänderungen für Apps, die auf API 28 oder höher ausgerichtet sind

Migration zu Android 9

Energieverwaltung

Einschränkungen für Nicht-SDK-Schnittstellen

Oreo

Nougat

Marshmallow

Lollipop

KitKat

Dashboards

Geräte

Wear OS

Android TV

Android Auto

Android Things

Chrome OS-Geräte

Android Developers > Plattform > Versionen

Android 9 features and APIs

Wichtiges Kapitel 😊

Neue Features und Änderungen

Frühere Versionen

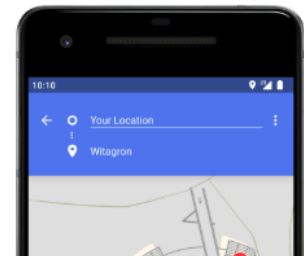
Android 9 (API level 28) introduces great new features and capabilities for users and developers. This document highlights what's new for developers.

To learn about the new APIs, read the [API diff report](#) or visit the [Android API reference](#). Also be sure to check out [Android 9 Behavior Changes](#) to learn about areas where platform changes may affect your apps.

Indoor positioning with Wi-Fi RTT

Android 9 adds platform support for the IEEE 802.11mc Wi-Fi protocol—also known as *Wi-Fi Round-Trip-Time* (RTT)—to let you take advantage of indoor positioning in your apps.

On devices running Android 9 with hardware support, your apps can use the [RTT APIs](#) to measure the distance to nearby RTT-capable Wi-Fi access points (APs). The device must have location services enabled and Wi-Fi



Apropos API-Changes...

Behavior changes: all apps

Inhalt ^

- Power management
- Privacy changes
- Restrictions on use of non-SDK interfaces
- Security behavior changes
 - Device security changes
 - Cryptographic changes
 - Android secure encrypted files are no longer supported
- Updates to the ICU libraries
- Android Test changes
- Java UTF decoder
- Hostname verification using a certificate
- Network address lookups can cause network violations
- Socket tagging
- Reported amount of available bytes in socket
- More detailed network capabilities reporting for VPNs
- Files in xt_qtaguid folder are no longer available to apps
- FLAG_ACTIVITY_NEW_TASK requirement is now enforced
- Screen rotation changes
- Apache HTTP client deprecation affects apps with non-standard ClassLoader
- Enumerating cameras

Android 9 (API level 28) introduces a number of changes to the Android system. The following behavior changes apply to *all apps* when they run on the Android 9 platform, regardless of the API level that they are targeting. All developers should review these changes and modify their apps to support them properly, where applicable to the app.

For changes that only affect apps that target API level 28 or higher, see [Behavior changes: apps targeting](#)

Framework security changes

Android 9 includes several behavior changes that improve your app's security. These changes take effect only if your app targets API level 28 or higher.

Network TLS enabled by default

If your app targets Android 9 or higher, the `isCleartextTrafficPermitted()` method returns `false` by default. If your app needs to enable cleartext for specific domains, you must explicitly set `cleartextTrafficPermitted` to `true` for those domains in your app's [Network Security Configuration](#).

Web-based data directories separated by process

In order to improve app stability and data integrity in Android 9, the system separates web data by directory among [multiple processes](#). Typically, such data directories include cookies, form data, and other persistent and temporary storage related to web browsing.

In most cases, your app should use classes from the `android.webkit` package, such as `WebView` and `CookieManager`, in only one process. For example, you should move all `Activity` objects that use a `WebView` into the same process. You can more strictly enforce the "one process only" rule by calling `disableWebView()` in your app's other processes. This call prevents `WebView` from being initialized in those other processes by mistake, even if it's being called from a dependent library.

If your app must use instances of `WebView` in more than one process, you must assign a unique data directory suffix for each process, using the `WebView.setDataDirectorySuffix()` method, before using a given instance of `WebView` in that process. This method places web data from each process in its own directory within your app's data directory.

★ **Note:** Even if you use `setDataDirectorySuffix()`, the system doesn't share cookies and other web data across your app's process boundaries. If multiple processes in your app need access to the same web data, you need to copy it between those processes yourself. For example, you can call `getCookie()` and `setCookie()` to manually transfer cookie data from one process to another.

Android P(ie), API-28

Beispiel für einen *Breaking Change*: Dazu gleich noch mehr



<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

Nebenläufigkeit: Threads

Java-Threads: Kurze Repetition

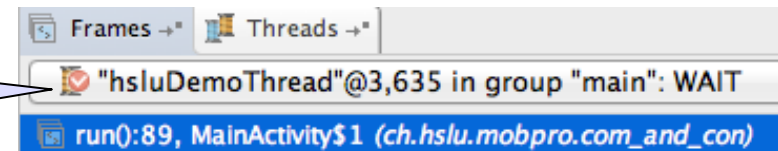
- `java.lang.Thread` implements `Runnable`
 - Muss ein `Runnable` gesetzt haben (übergeben im Konstruktor) oder selber `run()` implementieren
 - Wichtigste Methoden:
 - `run()`
 - `start()`
 - `sleep(long)`
 - `isAlive()`
- Interface `java.lang.Runnable` (oder Lambda)
 - Eine Methode: `run()`
 - Implementierung des relevanten nebenläufigen Codes

Verwendung von Threads

```
public void startAsyncDemoTask(View view) {
    if ((asyncDemoTask == null) || (asyncDemoTask.getStatus() == FINISHED)) {
        // only start a new task if there is no one running yet...
        asyncDemoTask = new AsyncDemoTask(this);
        asyncDemoTask.execute(1,2,3,4);
    } else {
        Toast.makeText(this, "AsyncDemoTask läuft schon!", LENGTH_SHORT).show();
    }
}
```

Thread-Name

Thread-Name wird im
Debugger angezeigt



```
private Thread createWaitThread(final Button button) {
    return new Thread("hsluDemoThread") {
        @Override
        public void run() {
            try {
                Thread.sleep(WAITING_TIME_SHORT);
                MainActivity.this.runOnUiThread(new Runnable() {
                    @Override
                    public void run() {
                        button.setText(getResources().getString(R.string.main_concurrencyThread));
                        Toast.makeText(MainActivity.this, "Demo Thread beendet!", LENGTH_SHORT);
                    }
                });
            }
        }
    };
}
```

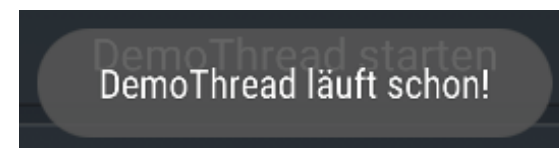
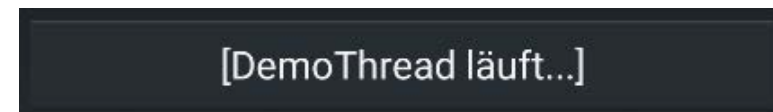
Blockierung!

Callback: Benachrichtigung des UI-
Thread (= main-Thread) durch
Activity.runOnUiThread(Runnable)

Besser: Java 1.8 aktivieren
und Lambda verwenden!

Demo: WaitThread

- Auf Knopfdruck wird im Hintergrund 7 Sek gewartet
 - UI wird NICHT blockiert, da auf Hintergrund-Thread 😊
- Wichtig: UI-Anpassungen nur auf UI-Thread
 - Zurück auf main-Thread mit `Activity.runOnUiThread`
- Weiter Implementiert
 - Änderung Button-Titel
 - Mechanismus, dass max. 1 Wait-Thread läuft






<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

(Backend-) Kommunikation über HTTP

Backend-Kommunikation

- Viele Apps kommunizieren mit einem Server im Hintergrund, welcher z.B. Daten hält oder Business-Logik bereitstellt oder User authentisiert  **Backend**
- Kommunikation mit dem Backend findet i.d.R. über ein (REST) HTTP-API statt
- Datenformat oft JSON, seltener XML

HTTP: Hyper Text Transfer Protocol

Was heisst „zustandslos“?

- Zustandsloses Kommunikationsprotokoll
 - Transport über TCP/IP
 - Request/Response Muster (Anfrage/Antwort)
 - Anfragemethoden: GET, PUT, POST, DELETE
 - Nachrichten bestehen aus zwei Teilen: Header + Body
 - 0..n Headers: Key-Value Paare
 - Body (Content): beliebig (typischerweise Text)
 - Mit jeder Antwort liefert Server einen Statuscode
 - z.B. 200 = OK
(http://developer.android.com/reference/java/net/URLConnection.html#HTTP_OK)

HTTP Requests absetzen

- Standard-Klassen `URL` und `URLConnection` erlauben das Absetzen von HTTP-Requests mit Java-Bordmitteln
 - Verwendung mühsam, uraltes API (letztes Jahrhundert)
- Besser: Eine HTTP-Client-Library verwenden
 - «Headless Browser»
 - Empfohlen: `OkHttpClient`, <http://square.github.io/okhttp/>
- Gradle Dependency

```
implementation 'com.squareup.okhttp3:okhttp:3.13.1'
```

```
implementation 'com.squareup.okhttp3:logging-interceptor:3.12.1'
```

Verwendung OkHttpClient (Text)

Client mit Logging
erzeugen
(ausschalten in
produktiver App!)

```
// HttpClient erzeugen und konfigurieren
HttpLoggingInterceptor logger = new HttpLoggingInterceptor();
logger.setLevel(HttpLoggingInterceptor.Level.BODY);
OkHttpClient client = new OkHttpClient.Builder()
    .addInterceptor(logger)
    .build();
```

POST-Request
Beispiel (mit Body)

```
// GET Request konfigurieren (GET = default)
Request request = new Request.Builder()
    // .post(RequestBody.create(MediaType.parse("text/json"), "{...}"))
    .url("http://www.wherever.ch/hslu/loremIpsum.txt")
    .build();
```

```
Response response = client.newCall(request).execute();
if (response.isSuccessful()) {
    return response.body().string();
}
```

Zugriff auf Body (Content)
erst nach Prüfung
Response-Code

```
String error = String.format("ERROR: Request failed with %s %s",
    response.code(), response.message());
Log.e("HttpService", error);
```

Zugriff auf Response-Metadaten

HTTP: Cleartext Traffic erlauben

Erinnere: API-Changes für Android 9 von früherer Folie!

- Seit Android 9 (API 28) ist Kommunikation über HTTP (unverschlüsselt) per default unterbunden

```
java.io.IOException: Cleartext HTTP traffic to wherever.ch not permitted
    at com.android.okhttp.HttpHandler$CleartextURLFilter.checkURLPermitted(HttpHandler.java:115)
    at com.android.okhttp.internal.huc.HttpURLConnectionImpl.execute(HttpURLConnectionImpl.java:458)
    at com.android.okhttp.internal.huc.HttpURLConnectionImpl.connect(HttpURLConnectionImpl.java:127)
    at ch.hslu.mobpro.com_and_con.MultiAsyncTask.openHttpConnection(MultiAsyncTask.java:51)
    at ch.hslu.mobpro.com_and_con.MultiAsyncTask.doInBackground(MultiAsyncTask.java:34)
    at ch.hslu.mobpro.com_and_con.MultiAsyncTask.doInBackground(MultiAsyncTask.java:19)
    at android.os.AsyncTask$2.call(AsyncTask.java:333)
```

- Manchmal aber nicht möglich oder nicht erwünscht!
Kann «clear text traffic» in Manifest explizit erlauben

```
<application
    android:icon="@mipmap/ic_launcher"
    android:label="Kommunikation und Nebenläufigkeit"
    android:theme="@style/AppTheme"
    android:usesCleartextTraffic="true"
    tools:ignore="GoogleAppIndexingWarning">
```


(Lokales) Testen der Backend-Kommunikation

- Testen mit lokalem Server ist möglich!
 - Netzwerkkommunikation kann mit dem Emulator wie auch mit einem HW-Gerät getestet werden, ohne dass Server im Internet erreichbar sein müssen
- Vorgehen
 1. Webserver/Service auf Entwicklermaschine installieren
 2. Android-Applikation starten und verbinden
 - a) Im Emulator bezeichnet **10.0.2.2** die IP-Adresse des Hostrechners, auf dem der Emulator läuft
 - b) Auf HW-Gerät muss auf die IP-Adresse des Hostrechners verbunden werden (gleiches WLAN)

Demo: Kommunikation mit lokalem HTTP-Server

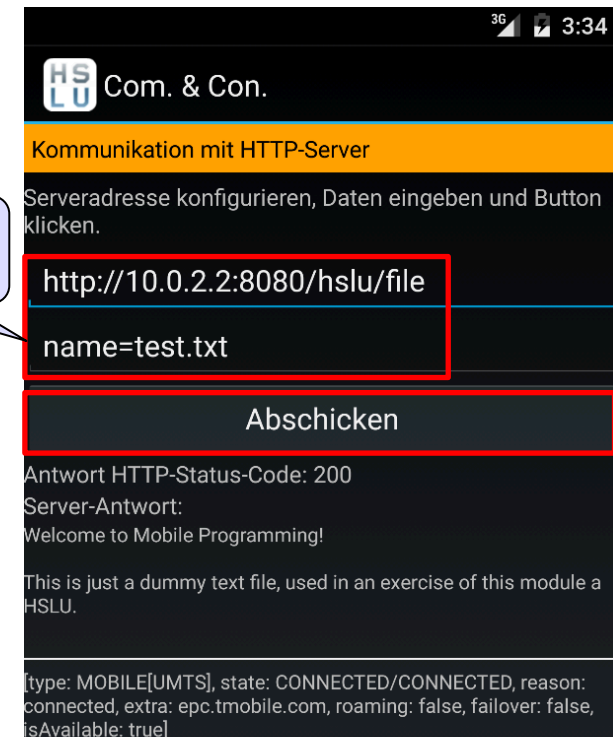
- Lokalen HTTP-Server starten
 - Siehe `SimpleHttpServer.jar` im Ilias (inkl. Code!)
 - Zwei Pfade
 - `/hslu/dummy`
 - Liefert Dummy-Response zurück
 - `/hslu/file`
 - Erwartet GET-Parameter `name=<filename>`

Probieren Sie
`homer.jpg` 😊

- HTTP GET-Request schicken
 - `http://10.0.2.2:8080/hslu/file`
 - `http://x.y.z.v:8080/hslu/dummy`

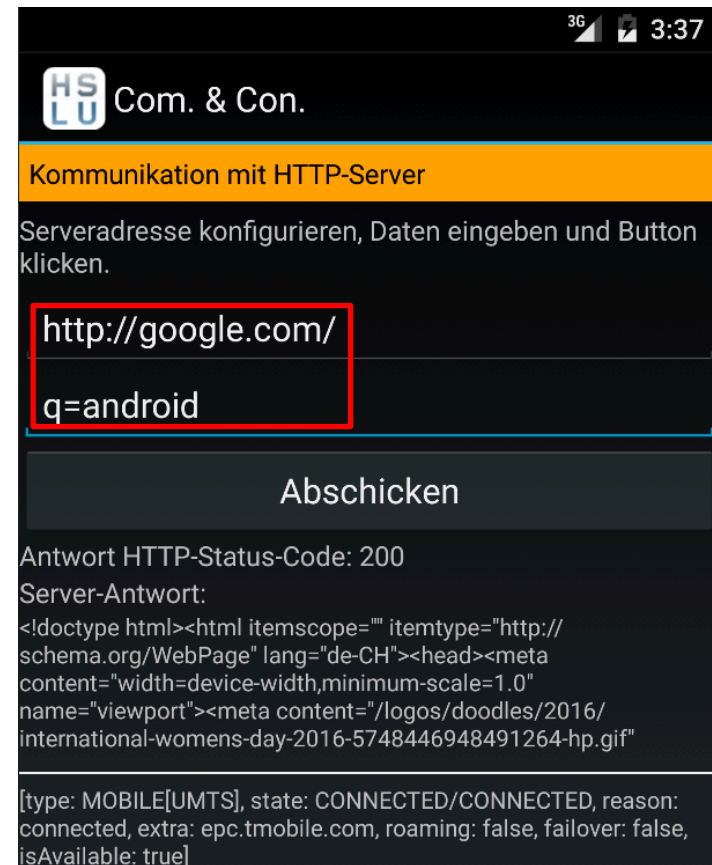
HW-Gerät

Emulator



Demo: HTTP-Aufruf beliebiger HTTP-Server

- z.B. google.com
 - Query: q=android
 - Entspricht Browser-Aufruf
`https://www.google.com/?q=android`
- Erzeugung GET-Parameter?
 - Selber zusammenbauen!
- Netzwerkstatus?
 - Klasse: `android.net.ConnectivityManager`
 - Methode: `getActiveNetworkInfo()`



Manifest: Berechtigungen

- Für Internet-Zugriff (und Netzwerkstatus) muss Berechtigung vorliegen (resp. deklariert sein)
- Bekannter Mechanismus: Im Manifest eintragen

```
<uses-permission android:name="android.permission.INTERNET" />  
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

Verwendung OkHttpClient (Binäre Daten)

```
Request request = new Request.Builder()  
    .url("http://www.wherever.ch/hslu/homer.jpg")  
    .build();
```

Gleich wie Text, aber binäre Resource

```
Response response = client.newCall(request).execute();  
if (response.isSuccessful()) {  
    return BitmapFactory.decodeStream(response.body().byteStream());  
} else {  
    logError(response);  
}
```

Zugriff auf Body
als InputStream

```
response.close();
```

Response schliessen nach Gebrauch
(aber erst nach Konsumation Body!)

Stream in Bild umwandeln

Die Klasse `BitmapFactory` transformiert die Bytes verschiedener Bildertypen (JPG, PNG, GIF) in ein `Bitmap`-Objekt. Dieses kann dann auf einer beliebigen `ImageView` als Input gesetzt werden.

```
InputStream in = openHttpConnection(urls[0]);  
Bitmap image = BitmapFactory.decodeStream(in);  
in.close();  
ImageView imageView = (ImageView) findViewById(R.id.image);  
imageView.setImageBitmap(image);
```

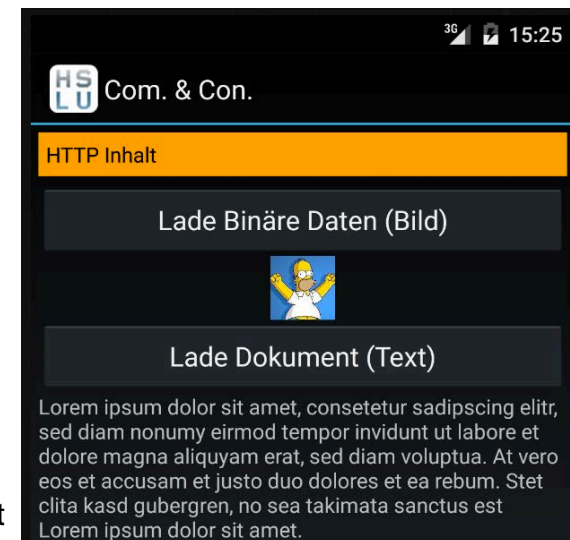
Lade Binäre Daten (Bild)



Demo: Bild & Text per HTTP laden & darstellen (Ü4)

- **Text:** `http://wherever.ch/hslu/loremIpsum.txt`
 - Oder lokal vom *SimpleHttpServer*
`/hslu/file/test.txt`
 - Dargestellt in `TextView`

- **Bild:** `http://wherever.ch/hslu/homer.jpg`
 - Oder lokal vom *SimpleHttpServer*
`/hslu/file/homer.jpg`
 - Dargestellt in `ImageView`





<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

JSON-Webservices mit Retrofit konsumieren

«Definition» eines HTTP-Webservice

- Oft gesehen: Gesamte Information (d.h. Pfad und gewünschte Operation) in URI verpackt:
 - z.B.: GET http://foobar.io/?action=delete&itemId=23
 - Verwendet typischerweise nur eine bis wenige HTTP-Methoden (typisch: nur GET und ggf. POST)
 - Verwendet also auch GET um etwas zu löschen oder um Daten zu schicken (als Query-Param-Value)
- Besser: REST Semantik verwenden, d.h. HTTP-Methoden für gewünschte CRUD Operation verwenden
 - Siehe nächste Folie

REST-ful Webservices

- Webservice auf der Basis von HTTP
- Grundidee (in purer Form)
 - Base-URL = Ressourcensammlung (<http://directory.com/contacts/>)
oder einer einzelnen Resource (<http://directory.com/contacts/17>)
 - HTTP-Methode = Operation auf Daten (GET, PUT, POST, DELETE)
 - Antwort-Datenformat = XML, JSON, ...

Wir erinnern uns:
Content-Provider!

Resource	GET	PUT	POST	DELETE
Collection URI, such as http://directory.com/contacts/	List the URIs and perhaps other details of the collection's members.	Replace the entire collection with another collection.	Create a new entry in the collection. The new entry's URL is usually returned by the operation.	Delete the entire collection.
Element URI, such as http://directory.com/contacts/17	Retrieve a representation of the addressed collection member, expressed in an appropriate media type.	Replace the addressed member of the collection, or if it doesn't exist, create it.	Treat the addressed member as a collection in its own right and create a new entry in it.	Delete the addressed member of the collection.

Quelle: http://en.wikipedia.org/wiki/Representational_state_transfer#Example

```
[
  {
    "sf": "HTTP",
    "lfs": [
      {
        "lf": "hypertext transfer protocol",
        "freq": 6,
        "since": 1995,
        "vars": [
          {
            "lf": "Hypertext Transfer Protocol",
```

- Acronym-Service:

- Pretty-Print: Browser Plugin oder

- **Dictionary (Word Definitions):**

- Pretty-Print: Direkt im Browser 😊

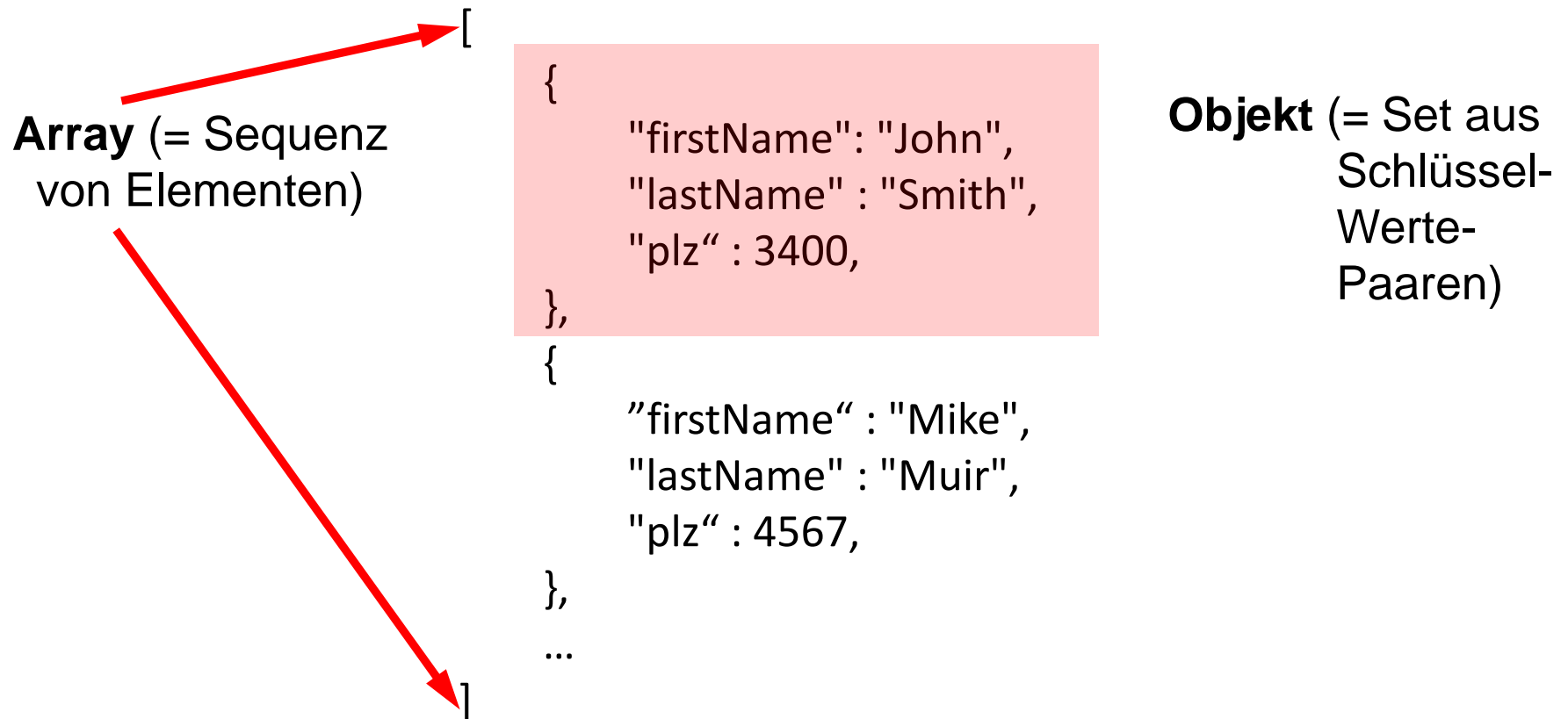
```

- <WordDefinition>
  <Word>android</Word>
- <Definitions>
  - <Definition>
    <Word>android</Word>
  - <Dictionary>
    <Id>gcide</Id>
    - <Name>
      The Collaborative International Dictionary of English v.0.44
    </Name>
  </Dictionary>
- <WordDefinition>
  Android \An"droïd\ ([a^]n"droïd), ||Androides \An*"droi"des\ ([a

```

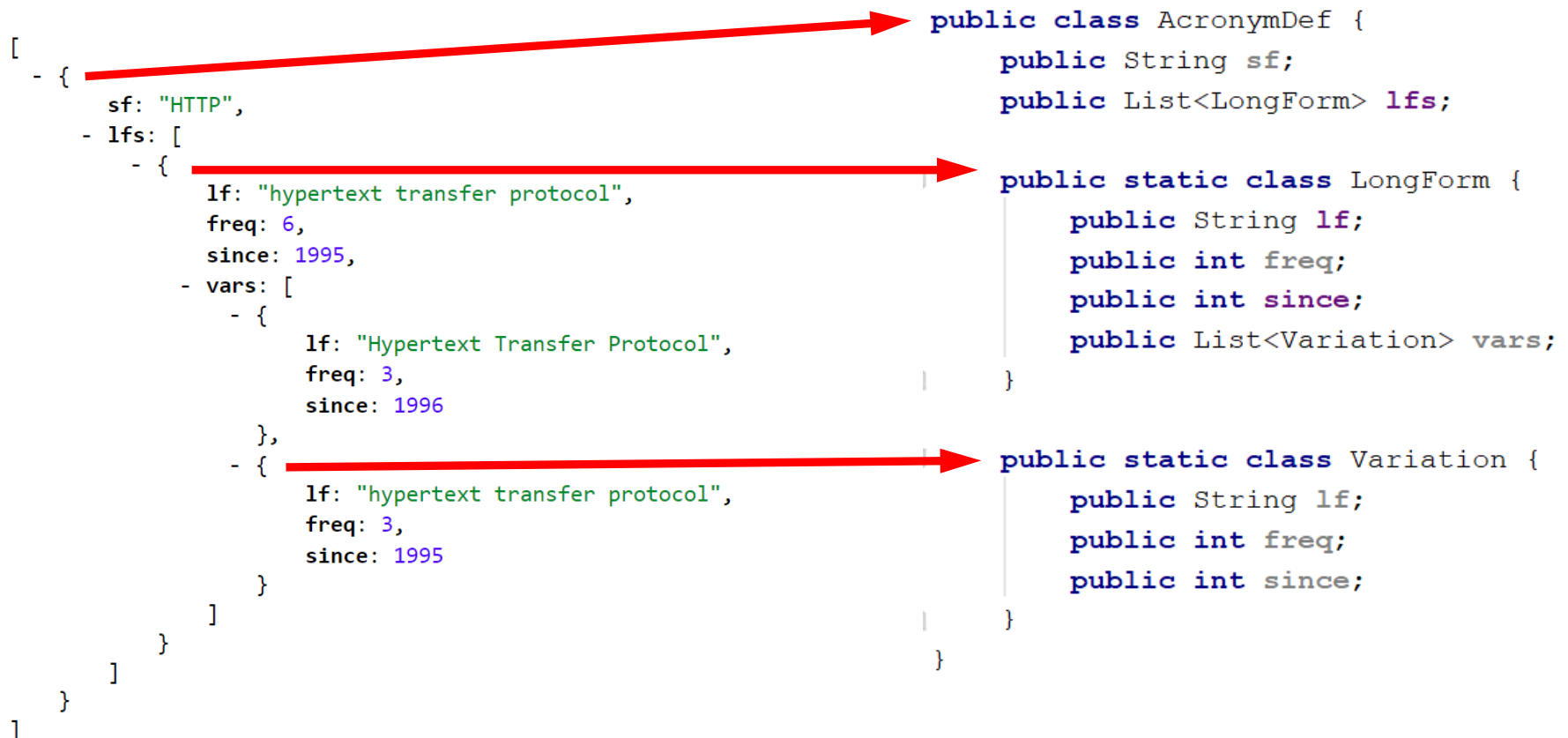
Mini-Exkurs: Datenformat JSON

Mehr zum Standard und Format: <http://json.org>



Json vs. Java

Service Doku unter: <http://www.nactem.ac.uk/software/acromine/rest.html>



JSON Parsing mit Gson

- Gson ist ein JSON-to-Java-Mapper
 - Bildet JSON-Strukturen auf äquivalente Java-Klassen ab (ähnlich ORM, wie bei Room gesehen)
- Beispiel

```
String url = "http://www.nactem.ac.uk/.../dictionary.py?sf=HTTP";  
OkHttpClient client = new OkHttpClient();  
Request request = new Request.Builder().url(url).build();  
Response response = client.newCall(request).execute();  
String json = response.body().string();
```

Etwas Trickserei, weil eine Liste von Definitionen zurückgegeben wird, aber in Java keine Runtime-Typ einer generischen Liste definiert werden kann

```
Gson gson = new Gson();  
Type listType = new TypeToken<List<AcronymDef>>(){}.getType();  
List<AcronymDef> definitions = new Gson().fromJson(json, listType);
```

Retrofit

- Aufrufe von Hand absetzen und JSON von Hand mappen ist immer noch recht «low level»
- Analog zu den Dao-Interfaces in Room für DB-Zugriff möchten wir das Backend als Interface abstrahieren, um HTTP-Call mit Java-Call zu ersetzen
- Dafür gibt es Retrofit: <https://square.github.io/retrofit/>

```
public interface AcronymService {  
  
    @GET("dictionary.py")  
    Call<List<AcronymDef>> getDefinitionsOf(@Query("sf") String sf);  
  
}
```

Retrofit Konfiguration und Aufruf

```
// Factory
Retrofit retrofit = new Retrofit.Builder()
    .client(new OkHttpClient())
    .addConverterFactory(GsonConverterFactory.create())
    .baseUrl("http://www.nactem.ac.uk/software/acromine/")
    .build();
```

Verwende Gson als Mapper

Base-URL (Präfix für alle erzeugten Services)

```
// Service-Instanz (nur einmal erzeugen)
AcronymService service = retrofit.create(AcronymService.class);
```

Typischerweise ein Service pro «Domäne» (vgl. DAO)

```
// Aufruf
Response<List<AcronymDef>> response =
    service.getDefinitionsOf("http").execute();
if (response.isSuccessful()) {
    return response.body();
}
```

Wiederverwenden!

Response-Verarbeitung gleich wie OkHttpClient

Retrofit: Dependencies

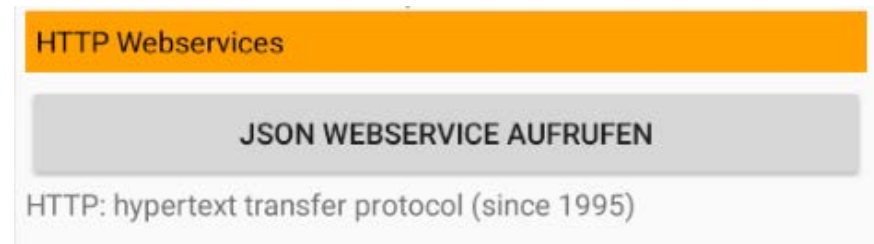
- Gradle

```
implementation 'com.squareup.retrofit2:retrofit:2.5.0'  
implementation 'com.squareup.retrofit2:converter-gson:2.5.0' // JSON mapper
```

- Retrofit basiert auf OkHttp, diese Library ist somit automatisch vorhanden
 - Kann verwendete OkHttpClient-Instanz vor dem Setzen auf der Retrofit-Factory noch konfigurieren (z.B. für Logging)

Demo: JSON-Service mit Retrofit konsumieren (Ü4)

- Service-URL:
`http://www.nactem.ac.uk/software/acromine/dictionary.py`
- Argument:
`sf=HTTP`
- Logging in Konsole:



```
2019-03-11 18:04:50.197 D/OkHttp: --> GET http://www.nactem.ac.uk/software/acromine/dictionary.py?sf=HTTP
2019-03-11 18:04:50.198 D/OkHttp: --> END GET
2019-03-11 18:04:51.016 D/OkHttp: <-- 200 OK http://www.nactem.ac.uk/software/acromine/dictionary.py?sf=HTTP (818ms)
2019-03-11 18:04:51.017 D/OkHttp: Date: Mon, 11 Mar 2019 17:04:53 GMT
2019-03-11 18:04:51.017 D/OkHttp: Server: Apache/2.2.15 (Scientific Linux)
2019-03-11 18:04:51.017 D/OkHttp: Connection: close
2019-03-11 18:04:51.018 D/OkHttp: Transfer-Encoding: chunked
2019-03-11 18:04:51.018 D/OkHttp: Content-Type: text/plain; charset=UTF-8
2019-03-11 18:04:51.047 D/OkHttp: [{ "sf": "HTTP", "lfs": [{ "lf": "hypertext transfer protocol", "freq": 6, "since": 1995, "vars":
[{"lf": "Hypertext Transfer Protocol", "freq": 3, "since": 1996}, {"lf": "hypertext transfer protocol", "freq": 3, "since": 1995}]}]}]
2019-03-11 18:04:51.047 D/OkHttp: <-- END HTTP (231-byte body)
```

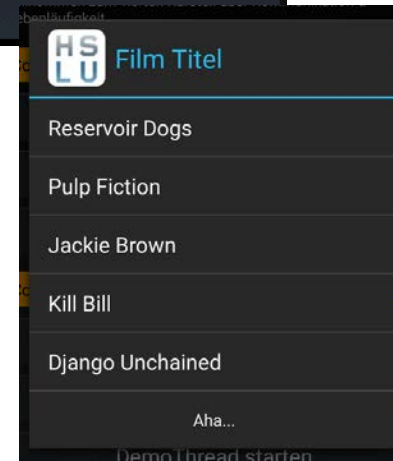
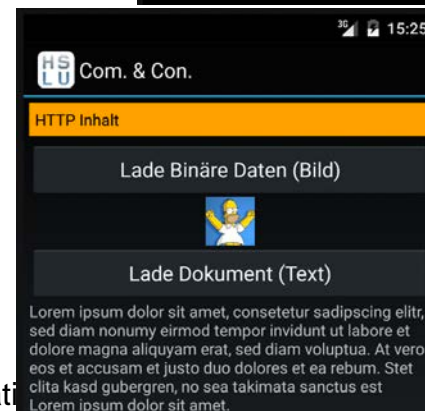
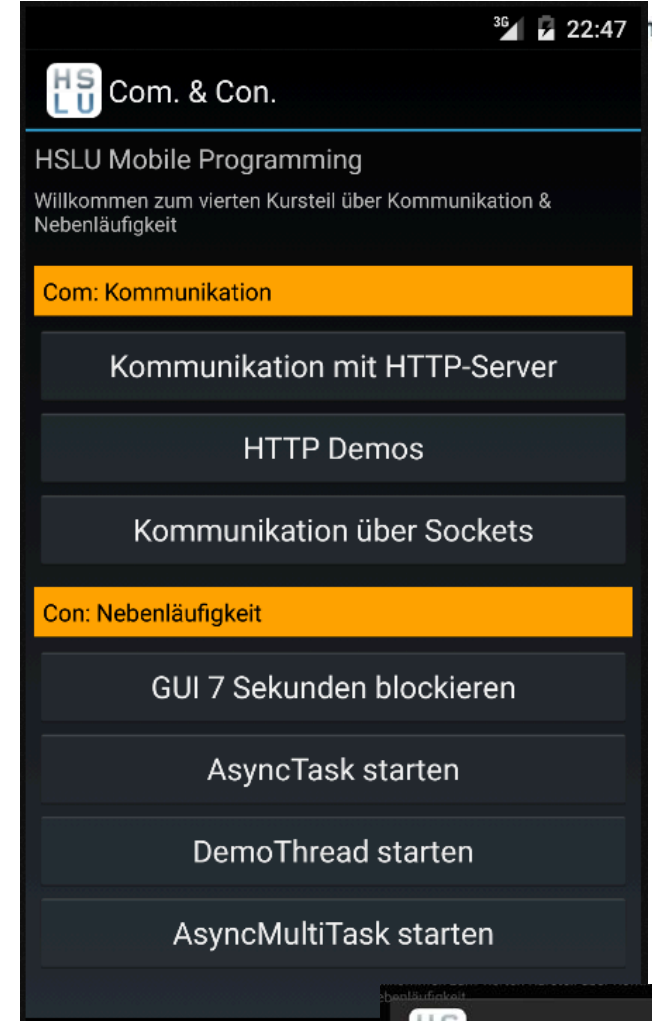


<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

Übung 4

Zur Übung 4

- HTTP-Kommunikation
 - Bild- und Textdokument
 - JSON-Daten mit Retrofit
- Nebenläufigkeit
 - „Blockier“-Button
 - Warten im AsyncTask
 - Warten in eigenem Thread
 - MultiAsyncTask
 - Mehrere Texte von Server
 - Toast: Zwischenresultat
 - Endanzeige in Dialog



Zur Übung 4: Server Code

- Vorgegebener Java Code im Ilias: SimpleHttpServer
 - als IntelliJ IDEA Projekt
 - als ausführbare jar-Datei (`java -jar SimpleHttpServer.jar`)

```
n0002065:ComAndCon-Server-Code taarnold$ java -jar SimpleHttpServer.jar 12345  
Simple HTTP server is up and running: 0:0:0:0:0:0:0:12345
```

- SimpleHttpServer
 - Zwei Kontexte
 - `hslu/dummy/` - Liefert DummyResponse (Text)
 - `hslu/file/` - mit Parameter „file=test.txt“ oder „file=homer.jpg“
 - Per Default auf Port 8080 (mit Argument auf bel. Port)