Inhaltsverzeichnis

Variationen zum Thema:
Android
Eine Einführung in mobile Anwendungen

von Ralph P. Lano, 1. Auflage

Für wen

Dieses Buch richtet sich an Bachelor-Studierende im fünften oder sechsten Semester. Als Voraussetzung muss man vor allem Java mitbringen. Außerdem sollte man etwas über Algorithmen und Datenstrukturen gehört haben, sonst machen viele der Beispiele keinen Sinn. Auch wird speziell im siebten Kapitel erwartet, dass man bereits eine Netzwerk Vorlesung gehört hat, vor allem die Protokolle IP, TCP, UDP und HTTP sollten bekannt sein. Viele der Beispiele beziehen sich auf meine ersten beiden Bücher [1,2], ohne macht wahrscheinlich die Hälfte dieses Buches keinen Sinn. Es ist also kein alleinstehendes Buch für Android und eigentlich nicht zum Selbststudium gedacht, sondern als Begleitung zu einer Vorlesung.

Von wem

ich bin seit 2011 Professor für Internetprogrammierung und Multimediaapplikationen im Studiengang MediaEngineering an der Technischen Hochschule Nürnberg. Von 2003 bis 2010 war ich Professor für Softwaretechnik und multimediale Anwendungen an der Hochschule Hof, und von 2010 bis 2011 Professor für Media and Computing an der Hochschule für Technik und Wirtschaft Berlin. Ich promovierte 1996 an der University of Iowa zum Thema 'Quantum Gravity: Variations on a Theme'. Von 1996 bis 1997 war ich Postdoctoral Research Associate am Centre for Theoretical Studies des Indian Institute of Science. In der Zeit von 1997 bis 2003 war ich zunächst bei Pearson Education und später bei der Siemens AG in der Softwareentwicklung und dem Projektmanagement tätig.

Über was

das Hauptthema in diesem Buch ist die Android Programmierung. Wir beginnen damit einfache Benutzeroberflächen zu schreiben, erst einmal ohne und dann mit XML Layout Dateien. Wir lernen was ein Intent ist und was man damit alles anstellen kann. Dann werden wir uns etwas ausführlicher mit 2D-Grafik Anwendungen beschäftigen, aber auch mit Toucheingaben und Gesten. Das Lesen von Ressourcen und Assets folgt als nächstes, aber auch das Lesen und Schreiben von internen und externen Speicher wird angesprochen. Die SQLite Datenbanken wird kurz vorgestellt, vertieft mit einigen Content Provider Beispielen. Natürlich werden auch die Sensoren behandelt, beginnend mit den Positions-Sensor, über die Umweltsensoren und schließlich sehen wir uns noch einige Beispiele mit den Linear-, Drehbeschleunigungs- und dem Magnetfeldsensor an.

Nach diesen einführenden Kapiteln geht es dann langsam ans Eingemachte: wir beschäftigen uns ausführlich mit Concurrency (Nebenläufigkeit). Dabei lernen wir Schritt für Schritt worauf man achten muss und wie man die allgemeinen Fallstricke vermeidet. Darauf aufbauend folgt dann die Netzwerkprogrammierung, wobei wir hier ziemlich ins Detail gehen. Es werden die Themen TCP, UDP und JSON behandelt, auch zu Bluetooth und NFC gibt es Beispiele. Als Schmankerl gibts dann TicTacToe als Netzwerkspiel. Im letzten Kapitel werden wir dann mit zahlreichen Beispielen die Multimediamöglichkeiten moderner Androidgeräte ausreizen.

Zusätzlich gibt es noch vier Special Topics: Eines beschäftigt sich mit Performance, genauer Grafikperformance. Dabei geht es aber eher darum ein Gefühl für die Problematik zu entwickeln, und weniger das schnellste Programm aller Zeiten zu schreiben. Ähnlich ist es bei dem Thema Libraries: hier ist Reuse das eigentliche Thema. Services und BroadcastReceivers sind zwar nützlich, aber man benötigt sie eher selten, deswegen sind die in die Special Topics verbannt worden. Und last but not least, folgt ein Special Topic zu Kryptographie. Jeder redet darüber, aber keiner zeigt einem wie es geht. Das kann jetzt aber niemand mehr behaupten.

Wie

lernt man Android Programmierung? Wie alles, durch viel üben! Deswegen ist auch dieses Buch wieder voll mit Übungsbeispielen. Die Veranstaltung so wie ich sie unterrichte besteht aus drei Komponenten: der Vorlesung, der Übung und Hausaufgaben. Die Vorlesung ist zwei Stunden pro Woche und entspricht jeweils dem ersten Teil eines Kapitels im Buch. Ein Kapitel schaffen wir in ca. ein bis zwei Wochen. In den Übungen, die vier Stunden alle zwei Wochen stattfinden, widmen wir uns dann den Projekten. Dabei schaffen wir zwischen zwei und vier der Projekte pro Übung. In der Übung arbeiten die Studierenden in Teams, meist zu zweit, um sich gegenseitig zu helfen. Die Hausaufgaben werden im zweiwöchentlichen Rhythmus bearbeitet und benötigen ca. 4 bis 5 Stunden. Es ist wichtig, dass die Studierenden alleine an der Hausaufgabe arbeiten.

Wo

finde ich die Beispiele und den Quellcode? Die gibt es auf der Webseite zum Buch: www.VariationenZumThema.de. Der Code umfasst knapp 300 Klassen, ca. 20 kLoC (wenn man weiß was das bedeutet). Er dient nur zu Schulungs- und Demonstrationszwecken, und enthält mit Sicherheit zahllose Fehler. Auch wurde er vor allem in Hinblick auf Lesbarkeit und Verständlichkeit geschrieben, deswegen stürzt die eine oder andere App auch mal ohne ersichtliche Gründe ab. Updates, Informationen zur Entwicklungsumgebung, und das Buch in elektronischer Version gibts auch auf der Webseite.

Darf ich

die Beispiele verwenden, oder das Buch kopieren?

Warum

dieses Buch? Seit 1996 programmiere ich Java, und seit ca. 2009 halte ich die Vorlesung "Multimediaapplikationen" in verschiedenen Formen, Studiengängen und Hochschulen. All die Zeit habe ich immer wieder versucht meine Vorlesung an diesem oder jenem Buch zu orientieren. Das Buch von Herrn Burnette [3] kommt am ehesten ran an das was ich suche. Allerdings hat das Buch nur sehr wenige Beispiele und es beschäftigt sich nur mit Android, es fehlen fortgeschrittenere Themen, die man im fünften oder sechsten Semester durchaus ansprechen kann.

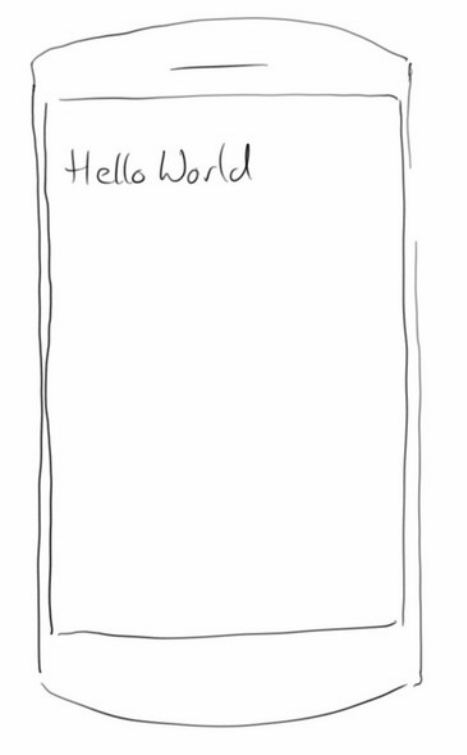
Woher

kommen die Ideen? Mehr als die Hälfte der Ideen basieren auf meinem ersten beiden Büchern [1,2]. Zu empfehlen ist auch das Buch von Ed Burnette [3], das eine sehr schöne Einführung in das Thema liefert, und das Buch von Ian Darwin [4], das gespickt ist mit allen möglichen Beispielen. Der Rest hat sich einfach so ergeben in den Jahren.

Referenzen

[1] R. Lano, Variationen zum Thema: Java: Eine spielerische Einführung, CreateSpace Independent Publishing Platform
[2] R. Lano, Variationen zum Thema: Algorithmen - Eine Einführung anhand von Beispielen, CreateSpace Independent Publishing Platform
[3] Ed Burnette, Hello, Android, O'Reilly, 2009
[4] Ian F. Darwin, ed, Android Cookbook, O'Reilly, 2012

Introduction



Wenn wir uns fragen mit welchem unserer Computer interagieren wir am meisten, dann ist die Antwort ganz klar: unserem Handy. Smartphones sind aus unserem Leben nicht mehr wegzudenken. Sie begleiten uns täglich und überall hin. Deswegen ist auch das zentrale Thema dieses Buches das Programmieren von mobilen Endgeräten. Da weit über 80 Prozent aller mobilen Endgeräte Android als Betriebssystem haben, werden wir uns auch ausschließlich mit selbigem beschäftigten. Das hat auch einen weiteren gewaltigen Vorteil, denn die bevorzugte Programmiersprache für Android ist Java und das kennen wir ja bereits aus den vorherigen Semestern.

Introduction

Microsoft hat Linux immer ein bisschen belächelt. Und auf dem Desktop mag das sogar immer noch berechtigt sein. Aber auf dem Servermarkt war das schon von Anfang an nicht begründet, und auf den mobilen Endgeräten hat Linux seinen wahren Siegeszug angetreten, denn Android ist Linux.

Was die Hardware angeht, denkt man bei Android zunächst an Smartphones und Tablets. Es gibt aber auch Uhren (Smartwatch) und Fernseher (Amazon Fire TV) mit Android als Betriebssystem, und es gibt sogar Android Spielekonsolen, wobei die sich aber bisher nicht durchgesetzt haben.

Glücklicherweise werden wir sehr wenig mit dem zugrunde liegenden Betriebssystem zu tun haben. So wie uns bei der Java Entwicklung für den Desktop und den Server das Betriebssystem eigentlich relativ egal war, so ist uns das auch bei der Entwicklung von Android Programmen relativ egal. Alles was wir wissen müssen, sind die APIs die uns in Java zur Verfügung gestellt werden. Diese APIs sind es auch was "Google Java" und "Oracle Java" unterscheidet, und warum es dieses Buch gibt.

Activity

Für unseren Einstieg in Android beginnen wir mit der *Activity* Klasse. Dabei handelt es sich um die Basisklasse für UI Anwendungen. Eine Activity ist das Äquivalent der Java Swing *JFrame* Klasse oder der ACM *Program* Klasse. Unsere erste Activity ist eine einfache "Hello World" Anwendung:


```
public class SimpleActivity extends Activity {

    @Override
    public void onCreate(Bundle savedState) {
        super.onCreate(savedState);

        LinearLayout ll = new LinearLayout(this);
        ll.setOrientation(LinearLayout.VERTICAL);

        TextView tv = new TextView(this);
        tv.setTextSize(24);
        tv.setText("Hello World!");

        ll.addView(tv);
        setContentView(ll);
    }
}
```

Was früher unsere *init()* Methode war (oder der Konstruktor), heißt jetzt *onCreate()*. Das *LinearLayout* entspricht grob dem *FlowLayout* wie wir es von Swing Anwendungen her kennen. Die Klasse *TextView* entspricht dem *JLabel* von Swing. Der Syntax ist ein klein bisschen anders, aber ansonsten verhalten sich die Klassen fast identisch. Damit wir unsere erste Activity allerdings zu sehen bekommen, benötigen wir noch einen Container, die *Application*.

Application

Für das Android Betriebssystem ist ein Android Programm eine *Application*. Applications werden in der *AndroidManifest.xml* Datei definiert und bestehen in der Regel aus einer oder mehreren Activities. Eine einfache AndroidManifest.xml Datei sieht wie folgt aus:

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

    <uses-sdk android:minSdkVersion="19" />
```

```
        <application android:label="MainApplication" >
            <activity
                android:name="SimpleActivity"
                android:label="Title SimpleActivity" >
                <intent-filter>
                    <action android:name="android.intent.action.MAIN" />
                    <category
                       android:name="android.intent.category.LAUNCHER" />
                </intent-filter>
            </activity>
        </application>

    </manifest>
```

Hier wird zunächst die Application definiert, sie hat den Namen "MainApplication". Innerhalb der Application gibt es eine Activity, unsere "SimpleActivity". Da eine Application aus mehreren Activities bestehen kann, müssen wir noch mitteilen wo es denn losgeht, und das ist was der Intent-Filter macht. Was das Coden angeht sind wir mit unserer ersten Anwendung fertig. War doch gar nicht so schwer.

Android Versionen

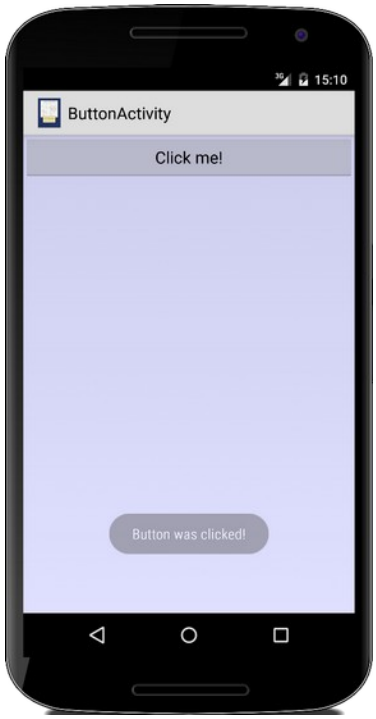
So wie alle Betriebssysteme, entwickelt sich auch Android und deswegen gibt es inzwischen schon einige verschiedene Versionen. Offiziell sind wir inzwischen bei Android 8, auch *Oreo* genannt, angekommen. Wichtig für uns ist das deswegen, weil mit jeder neuen Android Version zusätzliche Features hinzukommen. Wenn wir also in unserem Android Programm bestimmte Features verwenden, die es erst ab einer gewissen Version gibt, müssen wir das im AndroidManifest mitteilen, und das ist was die Zeile macht:

```
        <uses-sdk android:minSdkVersion="19" />
```

Sie besagt, dass unser Programm nur auf Geräten läuft die mindestens das API-Level "19" haben. Auf älteren Geräten lässt sich unser Programm erst gar nicht installieren. API-Level "19" entspricht der Android Version 4.4, was momentan ca. 90% aller Android Geräte beinhaltet [1]. Eine Liste mit allen API-Leveln und den entsprechenden Android Versionen findet man in Referenz [2].

ButtonActivity

Kommen wir zu unserer zweiten Activity. Es geht um die einfachste Form von User-Interaction: wir wollen auf einen Knopf drücken und ein Feedback erhalten. Wir verwenden dafür das *Button* Widget und verbinden es mit einem *OnClickListener*:




```
    public class ButtonActivity extends Activity {

        @Override
        public void onCreate(Bundle savedState) {
            super.onCreate(savedState);

            LinearLayout ll = new LinearLayout(this);
            ll.setLayoutParams(new LayoutParams(
                LayoutParams.MATCH_PARENT,
                LayoutParams.MATCH_PARENT));
            ll.setBackgroundColor(0x200000ff);
            ll.setOrientation(LinearLayout.VERTICAL);

            Button btn = new Button(this);
            btn.setTypeface(Typeface.create(
                "sans-serif", Typeface.NORMAL));
            btn.setText("Click me!");
```

```
            btn.setOnClickListener(new OnClickListener() {
                @Override
                public void onClick(View v) {
                    Toast.makeText(v.getContext(),
                            "Button was clicked!",
                            Toast.LENGTH_SHORT).show();
                }
            });
            ll.addView(btn);

            setContentView(ll);
        }
    }
```

Die *onClick()* Methode wird jedes mal aufgerufen, wenn auf den Knopf gedrückt wird. Dort verwenden wir dann einen *Toast* um dem Nutzer eine kurze Mitteilung zu geben. Ein Toast benötigt eine Referenz zu einem Context, in der Regel die Activity selbst (deswegen *this*):

```
Toast tst = new Toast(this);
tst.makeText(this, "Button was clicked!", Toast.LENGTH_SHORT);
tst.show();
```

Außerdem benötigt er den Text den er anzeigen soll und wie lange der Text angezeigt werden soll. Mittels der *show()* Methode wird der Toast dann angezeigt.

Die Art und Weise wie Buttons und Listener funktionieren ist vollkommen analog zu den Swing Klassen im normalen Java. Lediglich die Namen haben sich ein wenig geändert.

Intent

Das wirklich coole an Android ist, dass wir nicht jedes mal das Rad neu erfinden müssen. Wenn wir z.B. aus unserer Anwendung heraus mal kurz den Browser öffnen wollen, dann müssen wir keinen neuen Browser Code schreiben, sondern wir können den bereits existierenden Browser verwenden. Dafür gibt es die Klasse *Intent*.

Wir wollen eine einfache BrowserActivity schreiben, die aus einem Knopf besteht, und wenn wir auf den Knopf drücken soll sich ein Browser öffnen. Die BrowserActivity ist vollkommen analog zur ButtonActivity, lediglich in der onClick() Methode unterscheidet sie sich:

```
...
public void onClick(View v) {
    Uri uri =
        Uri.parse(
        "http://www.variationenzumthema.de/");
    Intent intent =
        new Intent(Intent.ACTION_VIEW, uri);
    startActivity(intent);
}
...
```

Als erstes generieren wir eine Uri, in diesem Fall eine Webadresse. Die übergeben wir der Intent Klasse mit der Aufforderung *ACTION_VIEW*, was so viel heißt wie "mach mal". Mittels *startActivity()* wird die andere Anwendung dann gestartet. Ist die andere Anwendung fertig, kehren wie wieder zu unserer ursprünglichen Anwendung zurück.

Allgemein kann man jede andere Android Anwendung damit starten. In der Regel beschränkt man sich aber auf die Standard Anwendungen. Mit den folgenden Zeilen, startet man die Standard Telefon Anwendung auf dem Gerät:

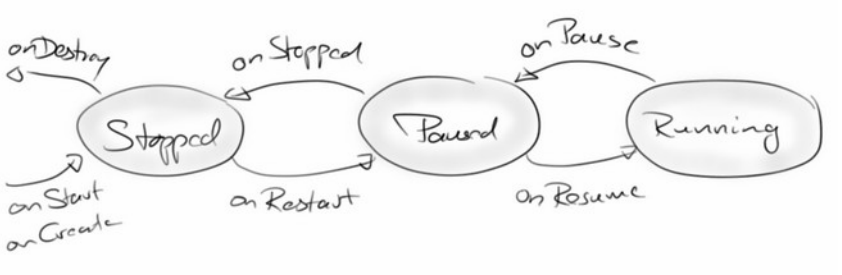
```
Uri uri = Uri.parse("tel:+49 123 456 7890");
Intent intent = new Intent(Intent.ACTION_VIEW, uri);
startActivity(intent);
```

Man kann aber auch eigene Activities starten:

```
Intent intent = new Intent(this, SimpleActivity.class);
startActivity(intent);
```

In den Projekten werden wir sehen, wie wir auf diese Art und Weise SMS versenden, neue Kalendereinträge machen, Barcodes scannen oder Videos aufnehmen können.

Activity Lifecycle



In allen Betriebssystemen mit einer graphischen Benutzeroberfläche, also z.B. Windows, Mac oder Linux, gibt es immer nur ein Fenster das im Vordergrund, also aktiv, ist. Genauso ist das in Android: es kann immer nur eine Activity gerade *active* sein.

Die anderen Activities sind dann im sogenannten *paused* Zustand. Zusätzlich gibt es in Android auch noch den *stopped* Zustand. Dieser ist deswegen notwendig, da manche Android Geräte sehr begrenzte Ressourcen haben, und es manchmal nicht möglich ist alle *paused* Activities im Speicher zu halten. Die werden dann einfach "zwangs-angehalten", also gestoppt. Deswegen gibt es in Android drei Zustände in denen sich eine Activity befinden kann:

- Active / Running
- Paused
- Stopped

Der Wechsel zwischen diesen Zuständen erfolgt über die sogenannten *State Switching* Methoden. Für dieses Wechseln (Switchen) ist der Activity Manager verantwortlich. Der weiß welche gerade die aktive Activity ist, und ist auch für das Neu-Anlegen, das Entfernen und Killen von Activities zuständig. In der Regel wird das Switchen vom Nutzer verursacht, weil er eine andere Activity startet. Es könnten aber auch externe Events sein, wie z.B. ein eingehender Telefonanruf. Der würde dafür sorgen, dass die momentan aktive Activity in den *paused* Zustand übergeht, und die TelefonActivity aktiv wird.

State Switching Methods

Wann immer ein Switching ansteht, sagt uns der Activity Manager kurz vorher Bescheid. Das tut er indem er eine der *State Switching* Methoden in unserer Activity aufruft. Für jeden Zustandswechsel gibt es genau eine Methode:

- **onCreate():** wenn unsere Activity startet.
- **onStart():** kurz bevor unsere Activity sichtbar wird.
- **onResume():** wenn unsere Activity aus dem *paused* Zustand zurückkehrt.
- **onPause():** wenn unsere Activity in den *paused* Zustand wechselt.
- **onStop():** wenn unsere Activity in den *stopped* Zustand wechselt.
- **onRestart():** wenn unsere Activity aus dem *stopped* Zustand zurückkehrt.
- **onDestroy():** kurz bevor die Activity beendet wird.

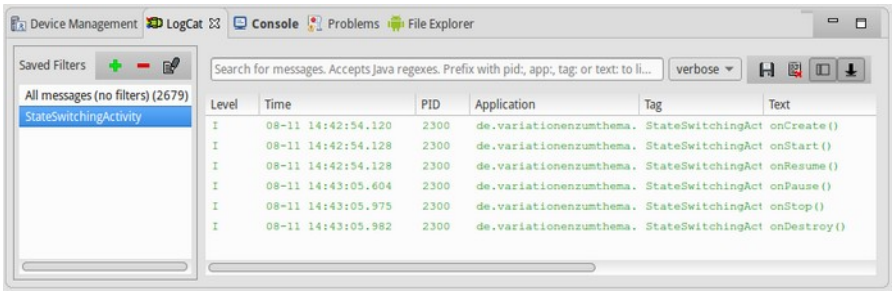
Am einfachsten betrachtet man das Activity Lifecycle Diagramm und dann wird der Zusammenhang sofort klar. Es sei noch angemerkt, dass die Methoden onStop() und onDestroy() nicht zwingend aufgerufen werden. Wenn das Betriebssystem irgendwann einmal ganz dringend Ressourcen benötigt, kann es also durchaus passieren, dass diese beiden Methoden nicht aufgerufen werden. Allerdings, die onPause() wird immer aufgerufen. Deswegen sollte man wichtige Daten immer in der onPause() Methode abspeichern.

StateSwitchingActivity

Schauen wir uns doch die State Switching Methoden mal an. Dazu schreiben wir eine kleine Activity in der wir alle State Switching Methoden überschreiben:

```
public class StateSwitchingActivity
    extends Activity {
    private final String TAG = "StateSwitching";

    protected void onCreate(Bundle savedState) {
        Log.i(TAG, "onCreate()");
        super.onCreate(savedState);
        ...
    }

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

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

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

    protected void onPause() {
        Log.i(TAG,"onPause()");
        super.onPause();
    }

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

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

dabei sehen wir dann folgende Ausgabe im LogCat:



Beim Starten der Activity werden also die Methoden onCreate(), onStart() und onResume() nacheinander aufgerufen, und beim Schließen die Methoden onPause(), onStop() und onDestroy().

Logging

Da wir auf unseren Android Geräten keine Konsole zur Verfügung haben, funktioniert auch ein println() oder eine System.out.println(), wie wir es vielleicht von früher her gewohnt sind, nicht mehr. Dafür gibt es jetzt die Klasse *Log*, die wir gerade oben verwendet haben:

```
Log.i("StateSwitchingActivity", "onCreate()");
```

Hier ist das erste Argument in der Regel der Name der Activity und das zweite die Message die wir ausgeben möchten. Ausgegeben werden die Messages auf dem sogenannten LogCat Fenster in unserer Entwicklungsumgebung.

Es gibt verschiedene Levels von Severity, also wie wichtig oder schwerwiegend eine Message ist. Hier gibt es sechs verschiedene Levels:

- **v():** verbose, wenn man sehr mitteilsam ist, dann kann man *verbose* verwenden, wird aber sehr selten benötigt.
- **d():** debug, wird während der Entwicklungsphase zum Testen verwendet.
- **i():** informational, tut genau das, gibt Informationen aus, die hilfreich sein könnten.
- **w():** warning, sollte für unerwartete oder ungewöhnliche Vorkommnisse verwendet werden.
- **e():** error, verwendet man wenn etwas ernsthaft schief gelaufen ist.

Der sechste ist die *wtf()* Methode für Spaßvögel.

RotationActivity

Wo wir gerade bei den State Switching Methoden sind: Etwas interessantes passiert, wenn wir unser Android Gerät um 90 Grad drehen (im Emulator geht das über Ctrl-F11 oder Ctrl-Cmd-F12, je nach Betriebssystem). Im LogCat sehen wir, dass nacheinander die Methoden

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

aufgerufen werden, d.h., die Activity wird beendet und neu gestartet. Was zur Folge hat, dass alle Nutzerdaten etc. gelöscht werden (es sei denn wir haben sie vorher gespeichert).

Man kann diesen Neustart verhindern mit zwei kleinen Modifikationen. Zunächst muss man im AndroidManifest mitteilen, dass unsere Activity selbst Konfigurationsänderungen handelt:

```
...
<activity
   android:name=".RotationActivity"
   android:label="RotationActivity"
   android:configChanges="keyboardHidden|orientation|screenSize">
</activity>
```

Und natürlich müssen wir sie auch handeln, und das machen wir durch das Überschreiben der *onConfigurationChanged()* Methode:

```
@Override
public void onConfigurationChanged(Configuration newConfig) {
    Log.i(TAG, "onConfigurationChanged()");
    super.onConfigurationChanged(newConfig);
    //setContentView(R.layout.myLayout);
}
```

Wenn wir die Anwendung jetzt testen und die Ausgabe im LogCat ansehen, stellen wir fest, dass die Activity nicht mehr beendet wird. Das ist jetzt nicht die feine Englische, und Google sagt auch, dass man das eigentlich nicht so machen soll, allerdings nennen sie auch keine wirklich einfache, nachvollziehbare Alternative.

Review

Wir haben nicht lange gekleckert und gleich unsere ersten Apps geschrieben. Dabei haben wir wichtige Konzepte kennengelernt, die wir für den Rest des Buches benötigen werden. Wir haben mit der Activity und dem AndroidManifest.xml begonnen. Mit dem TextView und dem Button haben wir unserer ersten UI Widgets verwendet. Der Toast wird für einfache, kurze Nachrichten an den Nutzer verwendet, während die Log Klasse Nachrichten für die Entwickler logt. Auch über die State-Switching Methoden und den Lifecycle den eine typische App durchläuft haben wir etwas gehört. Zum Abschluss haben wir uns noch intensiv mit Intents beschäftigt, mit denen man mit ganz wenig Befehlen bereits relativ viel erreichen kann.

Projekte

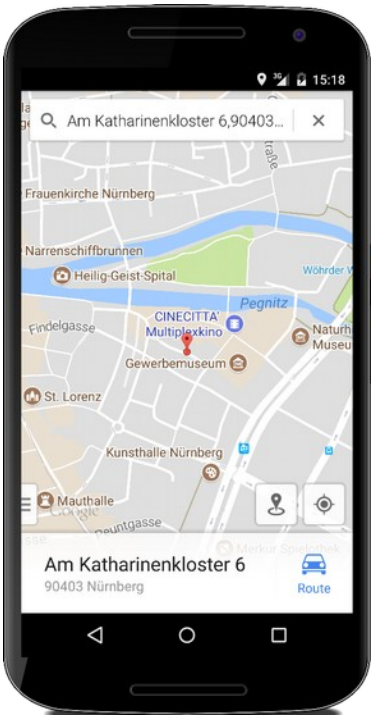
Schauen wir uns mal an was man mit Intents sonst noch so alles machen kann. Wir können z.B. Google Maps starten, SMS und Emails verschicken, oder einen Eintrag im Kalender machen. Auch Barcodes scannen und Videos aufnehmen ist ganz einfach.

GeoActivity

Genauso wie wir den Browser vorhin gestartet haben, können wir auch Google Maps starten, über einen Intent:

```
@Override
public void onClick(View v) {
    Uri uri = Uri.parse("geo:0,0?q="+
        "Am%20Katharinenkloster%206,90403%20"+
        "Nuremberg,Germany");
    Intent intent =
        new Intent(Intent.ACTION_VIEW, uri);
    startActivity(intent);
}
```

Dabei können wir über die URI entweder die direkten Geo-Koordinaten mitgeben, oder wir können Maps über den Query Parameter auch nach einer Adresse suchen lassen.



SMSActivity

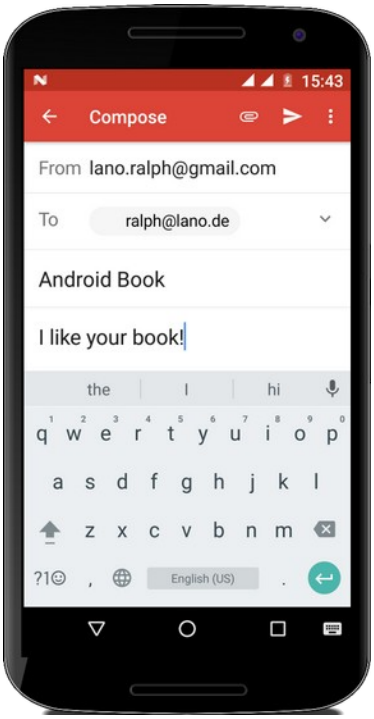
Wenn wir eine SMS versenden wollen, dann können wir das auch über einen Intent tun. Jedem Intent können wir Zusatzinformationen mitgeben:


```
@Override
public void onClick(View v) {
    Intent smsIntent =
        new Intent(Intent.ACTION_SENDTO);
    smsIntent.addCategory(Intent.CATEGORY_DEFAULT);
    smsIntent.putExtra("sms_body",
        "my first sms from an android app!");
    smsIntent.setType("vnd.android-dir/mms-sms");
    smsIntent.setData(
        Uri.parse("sms:" + "0123 456 7890"));
    startActivity(smsIntent);
}
```

Über *putExtra()* können wir zusätzliche Daten mitgeben, in diesem Fall die eigentliche SMS. Über die *setType()* Methode sagen wir, dass wir eine SMS verschicken möchten, was dann Android dazu veranlasst die vorinstallierte SMS Activity aufzurufen. Mit *setData()* teilen wir dann noch die Nummer mit, an die die SMS gehen soll. Wichtig, und das ist bei allen Intents so, die SMS wird nur vorbereitet, das eigentliche Senden der SMS muss immer noch der Nutzer selbst veranlassen. Das ist absichtlich so, wie wir später noch sehen werden.

EmailActivity

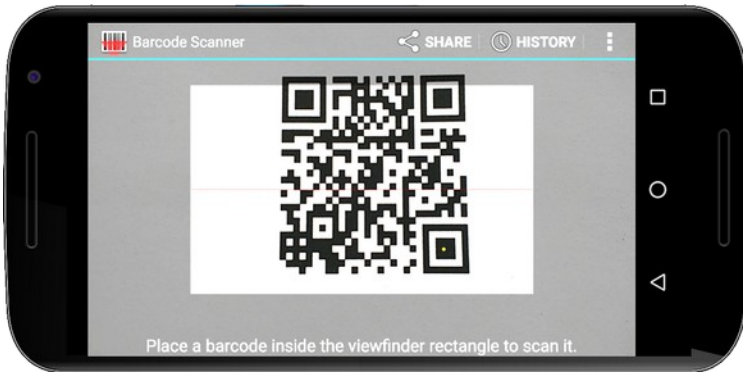
Ganz ähnlich wie SMS können wir auch Emails mit einem Intent versenden. Auch hier können wir dem Intent wieder Zusatzinformationen mitgeben:




```
@Override
public void onClick(View v) {
    Intent emailIntent =
        new Intent(Intent.ACTION_SENDTO,
            Uri.fromParts("mailto",
                "ralph@lano.de", null));

    emailIntent.putExtra(Intent.EXTRA_SUBJECT,
        "Android Book");
    emailIntent.putExtra(Intent.EXTRA_TEXT,
        "I like your book!");

    startActivity(Intent.createChooser(
        emailIntent, "Send email..."));
}
```

Über *putExtra()* geben wir die zusätzliche Daten mit, hier den Betreff und den Text der Email. Der Empfänger der Email wird über die *Uri.fromParts()* Methode mitgeteilt. Die *createChooser()* erlaubt es dem Nutzer aus verschiedenen Emails Clients auszuwählen, falls mehrere installiert sind. Wenn nur einer installiert ist, wird kein Chooser gezeigt.

BarcodeActivity



In den Beispielen die wir bisher gesehen haben, ging es darum über einen Intent eine andere Applikation zu starten und ihr evtl. noch etwas zusätzliche Information mitzugeben. Geht es aber auch umgekehrt, dass wir also Information von einer anderen Applikation erhalten?

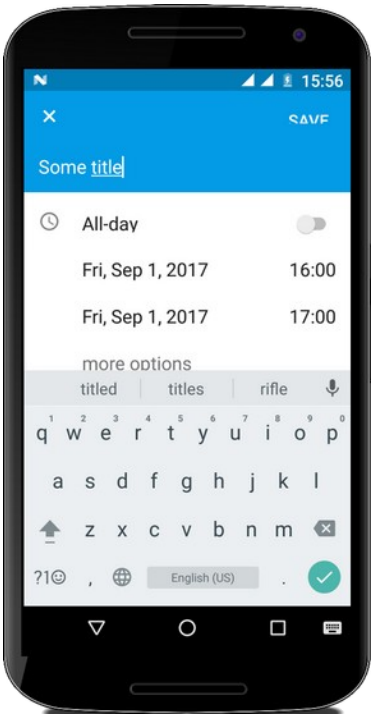
Als Beispiel betrachten wir die BarcodeActivity: dabei geht es darum einen 2D Barcode einzulesen. Da auf den meisten Handys ja bereits eine solche App installiert ist, wäre es doch das einfachste die zu benutzen. Wir verwenden wieder einen Intent, dem wir ein bisschen Zusatzinfos mitgeben, und starten ihn. Dieses mal aber nicht mit der startActivity() Methode, sondern mit der *startActivityForResult()* Methode:

```
public class BarcodeActivity extends Activity {

    private TextView tv;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        ...
        @Override
        public void onClick(View v) {
            Intent intent =
                new Intent("com.google.zxing.client.android.SCAN");
            intent.setPackage("com.google.zxing.client.android");
            intent.putExtra("SCAN_MODE", "QR_CODE_MODE");
            startActivityForResult(intent, 0);
        }
        ...
    }

    public void onActivityResult(int requestCode, int resultCode,
                                 Intent intent) {
        if (requestCode == 0) {
            if (resultCode == RESULT_OK) {
                String contents =
                    intent.getStringExtra("SCAN_RESULT");
                String format =
                    intent.getStringExtra("SCAN_RESULT_FORMAT");
                tv.setText(contents);

            } else if (resultCode == RESULT_CANCELED) {
                Toast.makeText(this, "Scan was canceled",
                    Toast.LENGTH_SHORT).show();
            }
        }
    }
}
```

Der Unterschied ist, dass im ersten Fall nichts passiert wenn die andere Anwendung fertig ist, im zweiten Fall wird aber die *onActivityResult()* Methode in unserer Activity aufgerufen. Man nennt so eine Methode auch einen *Callback*. In unserem Beispiel erhalten wir den Barcode der gescannt wurde.

CalendarActivity

Mittels Intents können wir auch Einträge im Kalender unseres Nutzer vornehmen:




```
@Override
public void onClick(View v) {
    Calendar beginTime = Calendar.getInstance();
    // year, month, day, hour, minute
    beginTime.set(2017, 8, 1, 16, 00);
    long startMillis = beginTime.getTimeInMillis();
    Calendar endTime = Calendar.getInstance();
    endTime.set(2017, 8, 1, 17, 00);
    long endMillis = endTime.getTimeInMillis();

    Intent intent = new Intent(Intent.ACTION_EDIT);
    intent.setType(
        "vnd.android.cursor.item/event");
    intent.putExtra("title", "Some title");
    intent.putExtra("description", "Some text");
    intent.putExtra("beginTime", startMillis);
    intent.putExtra("endTime", endMillis);
    startActivity(intent);
}
```

Wir müssen zunächst Beginn und Ende des Eintrags in Millisekunden berechnen, dann den Typ auf "vnd.android.cursor.item/event" setzen, und den eigentlichen Kalendereintrag wieder mittels *putExtra()* an die Calendar App mitgeben. Auch hier wird der Eintrag nur vorbereitet, der Nutzer muss immer noch seine Zustimmung geben, damit die Eintragung auch wirklich vorgenommen wird.

YouTube, Google Maps and OpenStreetMap

Drei weitere Anwendungsszenarien für Intents sind das Einbinden von YouTube Videos in unsere Anwendung und das Einbinden von Google Maps oder OpenStreetMap. Einen YouTube Video kann man ganz einfach über seine Webadresse einbinden:

```
Uri uri = Uri.parse(
  "http://www.youtube.com/watch?v=6ytGmtmUVSU");
Intent intent = new Intent(Intent.ACTION_VIEW, uri);
startActivity(intent);
```

Bei Google Maps kann man einfach die Adresse, also z.B. "Am Katharinenkloster 6,90403 Nuremberg,Germany" angeben:

```
Uri uri = Uri.parse("http://maps.google.com/?q=Am" +
  "%20Katharinenkloster%206,90403%20Nuremberg,Germany");
Intent intent = new Intent(Intent.ACTION_VIEW, uri);
startActivity(intent);
```

und bei OpenStreetMap übergibt man einfach den Längen- und Breitengrad des Ortes:

```
Uri uri =
  Uri.parse("http://www.openstreetmaps.org/lat=49.452&lon=11.082&zoom=20");
Intent intent = new Intent(Intent.ACTION_VIEW, uri);
startActivity(intent);
```

In allen drei Szenarien wird effektiv der Browser verwendet.

RecordVideoActivity

Als weiteres Beispiel für einen Intent der uns etwas zurückliefert betrachten wir die RecordVideoActivity. Wir möchten einen Video aufnehmen und dafür die ganz normale Kameraanwendung verwenden. Wir starten wieder einen Intent mittels der *startActivityForResult()* Methode:

```
public class RecordVideoActivity extends Activity {
    private TextView tv;

    public void onCreate(Bundle savedState) {
      ...
      public void onClick(View v) {
        Intent intent = new
           Intent(MediaStore.ACTION_VIDEO_CAPTURE);
        startActivityForResult(intent, 1);
      }
      ...
    }

    public void onActivityResult(int requestCode,
        int resultCode, Intent intent) {
        if (requestCode == 1) {
            if (resultCode == RESULT_OK) {
                Uri videoLocation=intent.getData();
                Toast.makeText(this, "Video:" + videoLocation,
                    Toast.LENGTH_LONG).show();

            } else if (resultCode == RESULT_CANCELED) {
                Toast.makeText(this, "Recording was canceled",
                    Toast.LENGTH_SHORT).show();
            }
        }
        super.onActivityResult(requestCode, resultCode, intent);
    }
}
```

Wenn wir dann fertig sind mit unserem Videodreh, wird wieder die *onActivityResult()* Methode aufgerufen mit dem Pfad wo die aufgenommene Videodatei zu finden ist.

NumpadActivity

Über die Methode *putExtra()* können wir anscheinend Daten von einer Activity an eine andere schicken. Was aber noch unklar ist, wie können wir denn aus der anderen Activity auf selbige zugreifen, und könnten wir evtl. wieder Daten zurück an die aufrufende Activity senden?

Betrachten wir das NumpadActivity Beispiel: dabei geht es darum, dass der Nutzer eine Zahl zwischen 1 und 9 auswählen soll. Wir wollen erst einmal Daten (einen int) von der *NumpadDemoActivity* an die *NumpadActivity* senden:

```
public class NumpadDemoActivity extends Activity {
    public void onCreate(Bundle savedState) {
        super.onCreate(savedState);
        int btnId = 42;
        Intent intnt = new Intent(
            getApplicationContext(),
            NumpadActivity.class);
        intnt.putExtra("id", btnId);
        startActivityForResult(intnt, 1);
    }
}
```

Das geht via der *putExtra()* Methode, also wie gehabt.

Wie kommen die Daten denn bei der NumpadActivity an? Wir können uns eine Referenz auf den Intent geben lassen, und mittels *getIntExtra()* darauf zugreifen:

```
public class NumpadActivity extends Activity ... {
    private int btnId = -1;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        btnId = getIntent().getIntExtra("id", -1);
        Log.i(TAG, "id=" + btnId);
        ...
    }
    ...
}
```

Diese "Extras" werden als Key-Value Paare abgespeichert und über den jeweiligen Key können wir auf die Werte zugreifen. Der zweite Parameter in der Methode *getIntExtra()* ist der Default-Wert, falls die aufrufende Activity keinen Wert mitgeliefert hat. Das vermeidet NullPointerExceptions.

So, nun wollen wir aber wieder Daten zurück an die *NumpadDemoActivity* schicken. Der Nutzer hat auf einen der neun Knöpfe gedrückt und in der *onClick()* Methode kreieren wir einfach einen neuen Intent, den wir mit Daten füllen:

```
@Override
public void onClick(View v) {
    int num = ((Button) v).getId();
    Intent intent = new Intent();
    intent.putExtra("id", btnId);
    intent.putExtra("num", num);
    setResult(RESULT_OK, intent);
    finish();
}
```

Mit der *finish()* Methode beenden wir uns selbst, und kehren zur aufrufenden Activity, also *NumpadDemoActivity* zurück.

Und wie kommen wir jetzt an die Daten in der *NumpadDemoActivity* ran? Über die *onActivityResult()* Methode:

```
public class NumpadDemoActivity extends Activity {
    ...
    public void onActivityResult(int requestCode, int resultCode,
        Intent data) {
        super.onActivityResult(requestCode, resultCode, data);
        if (requestCode == 1) {
            if (resultCode == RESULT_OK) {
                int id = data.getIntExtra("id", -1);
                int num = data.getIntExtra("num", -1);
                Log.i(TAG, "id=" + id + ", sNum=" + num);
            }
        }
    }
    ...
}
```

Also, gar nicht so schwer.

Challenges

In diesem Buch geht es nicht nur um Android, sondern auch darum, dass wir jetzt seit mehr als zwei Jahren programmieren, und dass es Zeit wird auch mal etwas fortgeschrittenere Themen anzusprechen.

SEP: Local and Anonymous Classes

In Java gibt es neben den normalen Klassen auch noch zwei weitere: lokale Klassen und anonyme Klassen. Die beiden nennt man manchmal auch innere Klassen im Gegensatz zu den normalen Klassen, die man auch äußere Klassen nennen könnte. Wir haben die anonymen Klassen schon mehrmals verwendet, ohne sie jedoch beim Namen zu nennen (daher der Name), z.B. bei der ButtonActivity:

```
public class ButtonActivity extends Activity {

    public void onCreate(Bundle savedInstanceState) {
        ...
        Button btn = new Button(this);
        btn.setText("Click me!");
        btn.setOnClickListener(new OnClickListener() {

            @Override
            public void onClick(View v) {
                Toast.makeText(v.getContext(),
                        "Button was clicked!",
                        Toast.LENGTH_SHORT).show();
            }
        });
        ...
    }
}
```

Was da innerhalb der *setOnClickListener()* Methode steht ist eine anonyme Klasse. Um zu verstehen, dass es sich dabei um eine anonymen Klasse handelt, wollen wir mal kurz die Metamorphose von einer normalen Klasse in eine lokale Klasse und schließlich in eine anonyme Klasse beobachten.

Normal Class

Wir beginnen mit zwei normalen Klassen, der *ButtonActivity* und der *MyOnClickListener* Klasse:

```
public class ButtonActivity1 extends Activity {

    public void onCreate(Bundle savedInstanceState) {
        ...
        Button btn = new Button(this);
        btn.setText("Click me!");
        btn.setOnClickListener(new MyOnClickListener());
        ...
    }
}

class MyOnClickListener implements OnClickListener {
    @Override
    public void onClick(View v) {
        Toast.makeText(v.getContext(), "Button was clicked!",
            Toast.LENGTH_LONG).show();
    }
}
```

ButtonActivity ist wie gehabt, und MyOnClickListener ist eine eigene Klasse, die das OnClickListener Interface implementiert. Wir sehen, dass wir in der *setOnClickListener()* Methode eine neue Instanz der Klasse MyOnClickListener instanziieren.

Local Class

Der Übergang zur lokalen Klasse ist subtil:

```
public class ButtonActivity2 extends Activity {

    public void onCreate(Bundle savedInstanceState) {
        ...
        Button btn = new Button(this);
        btn.setText("Click me!");
        btn.setOnClickListener(new MyOnClickListener());
        ...
    }

    class MyOnClickListener implements OnClickListener {
        @Override
        public void onClick(View v) {
            Toast.makeText(v.getContext(), "Button was clicked!",
                Toast.LENGTH_LONG).show();
        }
    }
}
```

Alles was sich ändert ist, dass die Klasse MyOnClickListener innerhalb der Klasse ButtonActivity deklariert wurde, also eigentlich wurde nur die geschweifte Klammer etwas anders gesetzt. Die Konsequenzen sind aber schon etwas schwerwiegender: zum Einen kann man die Klasse MyOnClickListener jetzt nur noch mit der Klasse ButtonActivity zusammen benutzen. D.h. sie hat ihre Eigenständigkeit verloren. Der Vorteil dieser Konstruktion ist allerdings, dass eine innere Klasse auf die Instanzvariablen ihrer äußeren Klasse zugreifen kann. Das kann sehr praktisch sein. Sehr häufig macht es auch noch Sinn die Sichtbarkeit der inneren Klasse auf *private* zu setzen.

Anonymous Class

Gibt es keine Notwendigkeit die lokale Klasse beim Namen zu nennen (was sehr häufig der Fall ist), braucht man ihr auch gar nicht erst einen Namen zu geben.

```
public class ButtonActivity3 extends Activity {

    public void onCreate(Bundle savedInstanceState) {
        ...
        Button btn = new Button(this);
        btn.setText("Click me!");
        btn.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                Toast.makeText(v.getContext(), "Button was clicked!",
                    Toast.LENGTH_LONG).show();
            }
        });
        ...
    }
}
```

Man macht hier zwei Dinge: zum Einen ruft man anstelle des Konstruktors von *MyOnClickListener* gleich den Konstruktor von *OnClickListener* auf. Zum Anderen schreibt man die Implementierung der zu überschreibenden Methoden gleich hinter den Konstruktor. Also bei einer anonymen Klasse handelt es sich also um eine lokale Klasse die namenlos ist.

Which is best?

Welche sollte man wann verwenden? Da scheiden sich die Geister. Ganz klar die einfachste, pragmatischste, quick & dirty Lösung ist die anonyme Klasse. Hier ist der Code der zum Button gehört gleich beim Button und man kann auch gleich noch auf die Instanzvariablen der äußeren Klasse zugreifen. Allerdings verletzt die anonyme Klasse das Softwareengineering-Prinzip des Separation of Concerns, d.h., dass eine Klasse nur eine Sache tun sollte, und auch der Zugriff von einer Klasse auf die Instanzvariablen (speziell der privaten) einer anderen, verletzt eigentlich das Prinzip des Information Hiding. Allgemein kann man aber sagen, macht die anonyme Klasse zu viel, dann sollte sie wohl eine lokale Klasse werden. Wird eine lokale Klasse auch in anderen Klassen benötigt, dann sollte sie wohl eher eine normale Klasse werden.

Fragen

1. Erklären Sie den Lebenszyklus (life cycle) einer Android Aktivität.

2. Was sind mögliche Ursachen die Ihre Anwendung aus dem Active / Running-Zustand in den Pause-Status übergehen lassen? (Nennen Sie drei)

3. Zeichnen Sie das Activity Life Cycle Diagramm, mit den drei Zuständen in denen eine Activity sich befinden kann, und nennen Sie auch die wichtigsten State Switching Methoden.

4. In jeder Activity gibt es sieben sogenannte State Switching Methoden:

   onCreate(), onStart(), onPause(), onResume(), onStop(), onRestart(), onDestroy()

   Von diesen werden manchmal die onStop() und die onDestroy() nicht aufgerufen. Geben Sie einen Grund warum das sein könnte und was die für Sie daraus resultierenden Konsequenzen sind.

5. Welche der State-Switching Methoden wird garantiert aufgerufen?

6. Nennen Sie drei Beispiele was man mit einem Intent machen kann.

7. Wie kann man Daten zwischen zwei verschiedenen Intents austauschen?

8. Es gibt zwei Möglichkeiten eine Activity oder einen Intent zu starten:
   - o startActivity()
   - o startActivityForResult()

   Erklären Sie kurz den Unterschied.

9. Im folgenden sehen Sie ein Beispiel für die Verwendung von Intents. Beschreiben Sie kurz, was der jeweilge Code macht.

```
Uri uri = Uri.parse("http://www.ohm-hochschule.de/");
Intent intent = new Intent(Intent.ACTION_VIEW, uri);
startActivity(intent);
```

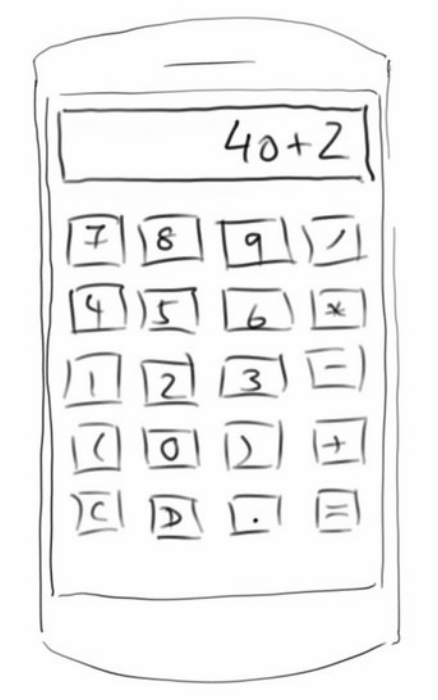
10. Was macht man mit einem Toast?
11. Wofür wird die Datei “AndroidManifest.xml” verwendet? Geben Sie mindestens zwei Beispiele.
12. Warum benötigt man in Android eine Log Klasse?
13. Erklären Sie grob den Unterschied zwischen normalen, lokalen und anonymen Klassen, und wann man welche verwenden sollte.
14. Was ist aus Entwicklersicht zu beachten, wenn das Smartphone vom Nutzer um 90 Grad gedreht wird?

Referenzen

[1] Android version history, https://en.wikipedia.org/wiki/Android_version_history

[2] Codenames, Tags, and Build Numbers, https://source.android.com/source/build-numbers

User Interface



Das zentrale Element fast jeder mobilen Anwendung ist das User Interface. Wenn wir uns an Swing erinnern, dann gab es dort z.B. einen JLabel, JButton, JTextField, JCheckBox, oder JRadioButton. In Android heißen die jetzt TextView, Button, EditText, CheckBox oder RadioButton. Auch Layouts gibt es in Android, und natürlich heißen auch die anders, aber was die Funktionalität angeht, sind sie sehr ähnlich. Was anders ist in Android ist die ungewöhnlich große Anzahl verschiedener Gerätegrößen. Auf der einen Seite gibt es Geräte mit einer Auflösung von 320x240 Pixel, auf der anderen gibt es aber auch welche mit einer 4K Auflösung. Um das unter einen Hut zu bringen hat man sich etwas besonderes einfallen lassen: die XML Layout Dateien.

ButtonActivity

Betrachten wir unsere ButtonActivity aus dem ersten Kapitel. Jetzt allerdings wollen wir das User Interface in einer XML Layout Datei *button_activity.xml* definieren:

```
<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:background="#200000ff"
    android:orientation="vertical" >

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:fontFamily="sans-serif"
        android:text="Click me!"
        android:textAllCaps="false" />

</LinearLayout>
```

Wir erkennen unser LinearLayout wieder und auch unseren "Click me!" Knopf. Die Widgets selbst werden als XML Elemente dargestellt, die Eigenschaften der jeweiligen Widgets werden über XML Attribute festgelegt. Also für jedes XML Element gibt es eine entsprechende Klasse, und für jedes XML Attribut eine entsprechende Methode. Die Verschachtelung, also welche Widgets hierarchisch zu welchen anderen gehören, wird durch die XML Verschachtelung nachempfunden.

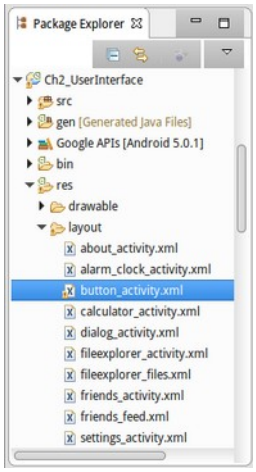
Nun müssen wir dieses XML Layout mit unserem Code verbinden:

```
public class ButtonActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.button_activity);

        Button btn = (Button) findViewById(R.id.button);
        btn.setOnClickListener(new OnClickListener() {

            @Override
            public void onClick(View v) {
                Toast.makeText(v.getContext(),
                    "Button was clicked!",
                    Toast.LENGTH_LONG).show();
            }
        });

    }
}
```

Zunächst sagen wir mit *setContentView(R.layout.button_activity)*, dass der ContentView (also die UI) in der Datei *button_activity.xml* definiert ist. Dann benötigen wir eine Referenz auf unseren Button, und das macht die *findViewById()* Methode: die sucht nach einem View namens "id.button". Den haben wir aber ja oben in der XML Datei mit dem Attribute "@+id/button" festgelegt. D.h. über das id-Attribut können wir aus unserem Code heraus auf die Elemente in der Layout Datei zugreifen.

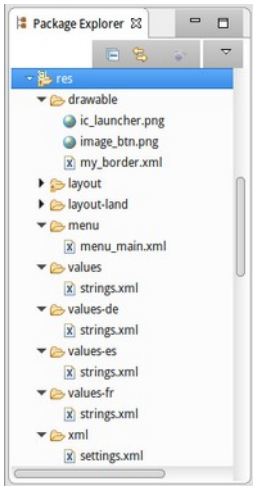
Generell führt die Auslagerung der UI in eine XML Layoutdatei zu viel schlankerem Code. Außerdem erlaubt es eine einfache Internationalisierung (i18n) wie wir gleich noch sehen werden.

Resources und R.java



Die XML Layout Datei ist ein Beispiel für eine Resource. In der Android Entwicklung wird sehr viel über Ressourcen erledigt, ein schönes Beispiel für gutes Software Engineering. Alle Ressourcen finden sich im Verzeichnis */res/* in unserem Android Projekt. Dort finden wir in einem Unterverzeichnis /layout/ alle XML Layout Dateien.

Wir haben mithilfe der Layout Datei eine Trennung von grafischer Benutzeroberfläche und Code erreicht. Diese Trennung von grafischer Benutzeroberfläche und Code hat mehrere Vorteile: wir können die Anordnung der UI Widgets ändern, ohne neuen Code schreiben zu müssen. Wir können verschiedene UIs für verschiedene Gerätegrößen und Formen haben (z.B. Telefone oder Tablets), ohne neuen Code schreiben zu müssen. Und Designer und Entwickler können gleichzeitig arbeiten ohne aufeinander warten zu müssen.



Ressourcen beinhalten aber nicht nur Layout Dateien. Die wichtigsten Ressourcen die man im Verzeichnis */res/* findet sind

- **drawables:** hier finden sich u.a. Icons und Bilddateien,
- **layout:** enthält XML Dateien, die UIs beschreiben,
- **menu:** enthält XML Dateien, die Menus beschreiben,
- **values:** enthält vor allem die *strings.xml* Datei,
- **xml:** enthält XML Dateien, z.B. für eine SettingsActivity.

Aus all diesen Ressourcen erzeugt ein Precompiler (aapt) die Datei *R.java*: die befindet sich im Verzeichnis */gen/* (für generated files) und sieht in etwa so aus:

```
/* AUTO-GENERATED FILE.  DO NOT MODIFY.
 *
 * This class was automatically generated by the
 * aapt tool from the resource data it found.  It
 * should not be modified by hand.
 */

package de.variationenzumthema.android.ch2;

public final class R {
    public static final class drawable {
        public static final int ic_launcher=0x7f020000;
        public static final int image_btn=0x7f020001;
        public static final int my_border=0x7f020002;
    }
    public static final class id {
        public static final int about_content=0x7f080000;
        public static final int textView=0x7f080001;
        public static final int button=0x7f080002;
        public static final int editText=0x7f080003;
        public static final int linearLayout1=0x7f080004;
    }
    public static final class layout {
        public static final int about_activity=0x7f030000;
        public static final int alarm_clock_activity=0x7f030001;
        public static final int button_activity=0x7f030002;
        public static final int calculator_activity=0x7f030003;
        public static final int dialog_activity=0x7f030004;
    }
    public static final class menu {
        public static final int menu_main=0x7f070000;
    }
    public static final class string {
        public static final int app_name=0x7f050000;
```

```
            public static final int hello=0x7f050001;
            public static final int action_settings=0x7f050002;
            public static final int set_alarm=0x7f050003;
        }
        public static final class xml {
            public static final int settings=0x7f040000;
        }
    }
```

Wir sehen zu jedem Unterverzeichnis in */res/* gibt es eine separate lokale Klasse, und zusätzlich gibt es die Klasse *id*, die alle irgendwo mit

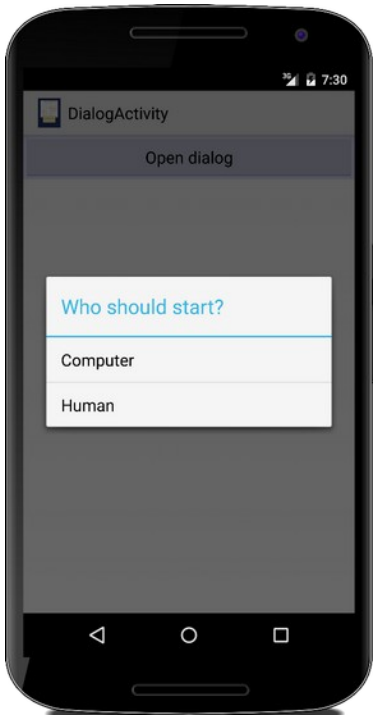
```
android:id="@+id/..."
```

definierten Ids zusammenfasst. Jeder Resource und Id wird eine eindeutige hexadezimale Konstante zugewiesen, die etwas wie eine Adresse aussieht aber keine ist. Im Code werden diese Konstanten dann referenziert, z.B.:

```
setContentView(R.layout.button_activity);
Button btn = (Button) findViewById(R.id.button);
```

und dabei ist *R* die generierte Klasse *R* aus der Datei *R.java*.

Dialog

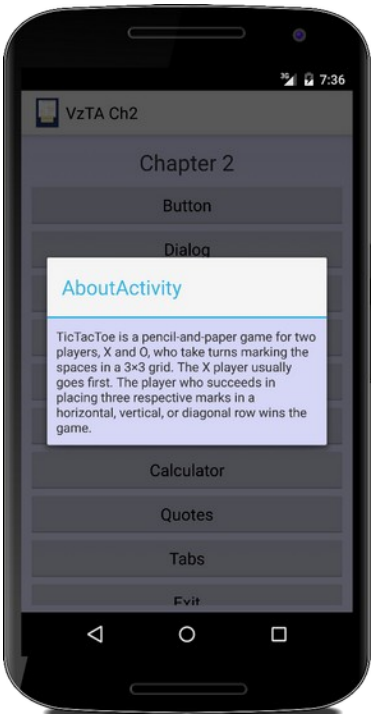


Weil wir gerade dabei sind: nicht selten benötigen wir von unseren Nutzern eine Entscheidung, z.B. bei TicTacToe ist die Frage ob der Mensch oder der Computer den ersten Zug machen soll. Für solche einfachen Abfragen verwenden wir Dialoge. Diese erstellt man am einfachsten mit dem *AlertDialog.Builder*:

```
public class DialogActivity extends Activity {
    ...

    private String[] who_starts = {
        "Computer", "Human" };

    private void openNewGameDialog() {
        new AlertDialog.Builder(this)
            .setTitle("Who should start?")
            .setItems(who_starts,

        new DialogInterface.OnClickListener() {

            @Override
            public void onClick(DialogInterface dialog, int which) {
                Toast.makeText(((Dialog) dialog).getContext(),
                    "Let's start the game, " + which + " will start.",
                        Toast.LENGTH_LONG).show();
            }
        }).show();
    }
}
```

In dem Array *who_starts* legen wir fest welche Optionen angezeigt werden sollen. Über den Parameter *which* wird uns dann mitgeteilt welche Auswahl unser Nutzer getroffen hat.

About



Ein anderes Beispiel wie einem die Layout-Datei das Leben vereinfachen kann ist die AboutActivity. Es geht darum dem Nutzer ein kurzes Feedback oder Hilfe zu geben. Die UI wird dabei in der XML Datei festgelegt:

```
<ScrollView xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="#200000ff"
    android:padding="10dip" >


    <TextView
        android:id="@+id/about_content"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text=
            "TicTacToe is a pencil-and-paper game
            for two players, X and O, who take
            turns marking the spaces in a 3×3 grid.
            The X player usually goes first. The
            player who succeeds in placing three
            respective marks in a horizontal,
            vertical, or diagonal row wins the game." />

</ScrollView>
```

Wir sehen hier das erste Mal den ScrollView, der, falls der Platz für den Text nicht ausreicht, das Scrollen ermöglicht. Wir werden den später noch häufiger verwenden.

Die Activity selbst ist nur vier Zeilen lang:

```
public class AboutActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.about_activity);
    }
}
```

Hätten wir das Gleiche mit Code versucht, wäre das viel hässlicher geworden.

Menu

Wenn wir ein Menu zu unserer Activity hinzufügen wollen, geht das auch wieder am einfachsten über eine XML Datei. In diesem Fall definieren wir eine Datei namens "menu_main.xml" im Verzeichnis /res/menu/ mit folgendem Inhalt:

```
<menu xmlns:android=
    "http://schemas.android.com/apk/res/android" >
    <item
        android:id="@+id/about"
        android:alphabeticShortcut="a"
        android:title="About"/>
    <item
        android:id="@+id/calculator"
        android:alphabeticShortcut="c"
        android:title="Calculator"/>
</menu>
```

Der Inhalt erklärt sich von selbst. Die dazu gehörige Activity ist wieder angenehm kurz:

```
public class MenuActivity extends Activity {

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

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        switch (item.getItemId()) {
        case R.id.about:
            startActivity(new Intent(this, AboutActivity.class));
            return true;
        case R.id.calculator:
            startActivity(new Intent(this, CalculatorActivity.class));
            return true;
        }
        return false;
    }
}
```

Die Verknüpfung zwischen den beiden wird wieder über die generierte Klasse *R* hergestellt.

BorderLayout

Kommen wir kurz auf die Layouts in Android zu sprechen. Die wichtigsten sind die folgenden fünf:

- LinearLayout
- RelativeLayout
- GridLayout
- FrameLayout
- TableLayout

Den ersten haben wir schon kennengelernt. Der zweite, der sogenannte RelativeLayout ist sehr mächtig, allerdings nicht ganz einfach. Deswegen hier ein Beispiel wie man das aus Swing bekannte BorderLayout mittels des RelativeLayout nachbilden kann:

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

    <TextView
        android:id="@+id/north"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:background="#200000ff"
        android:gravity="center_horizontal"
        android:text="North"
        android:textAppearance=
            "@android:style/TextAppearance.Large" />

    <TextView
```

```
            android:id="@+id/south"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_alignParentBottom="true"
            android:background="#200000ff"
            android:gravity="center_horizontal"
            android:text="South"
            android:textAppearance=
                "@android:style/TextAppearance.Large" />

        <TextView
            android:id="@+id/west"
            android:layout_width="wrap_content"
            android:layout_height="match_parent"
            android:layout_above="@id/south"
            android:layout_alignParentLeft="true"
            android:layout_below="@id/north"
            android:gravity="center_vertical"
            android:text="West"
            android:textAppearance=
                "@android:style/TextAppearance.Large" />

        <TextView
            android:id="@+id/east"
            android:layout_width="wrap_content"
            android:layout_height="match_parent"
            android:layout_above="@id/south"
            android:layout_alignParentRight="true"
            android:layout_below="@id/north"
            android:gravity="center_vertical"
            android:text="East"
            android:textAppearance=
                "@android:style/TextAppearance.Large" />

        <TextView
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:layout_above="@id/south"
            android:layout_below="@id/north"
            android:layout_toLeftOf="@id/east"
            android:layout_toRightOf="@id/west"
            android:background="#100000ff"
            android:gravity="center"
            android:text="Center"
            android:textAppearance=
                "@android:style/TextAppearance.Large" />

    </RelativeLayout>
```

Die Elemente definieren ihre Position relativ zueinander über die Ids. Die dazugehörige Activity macht eigentlich gar nichts.

```
    public class BorderLayoutActivity extends Activity {

        @Override
        public void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.border_layout_activity);
        }
    }
```

Den GridLayout werden wir im nächsten Beispiel verwenden, der FrameLayout muss noch bis zu den Fragments warten. Der TableLayout (mein Lieblingslayout) kommt im sechsten Kapitel dran (AllAccelGyro).

Calculator

In diesem Beispiel wollen wir drei Dinge sehen: zunächst natürlich die Verwendung des GridLayout. Dann geht es aber auch darum zu zeigen, dass für manche Sachen die XML Layout Datei besser geeignet ist, andere Dinge aber besser programmatisch gelöst werden. Und schließlich, und das werden wir noch öfter in diesem Buch sehen, war das zweite Semester nicht ganz umsonst.

Wir definieren zunächst das generelle Layout der Anwendung wieder über die XML Datei:

```
<LinearLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:id="@+id/linearLayout1"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="#200000ff"
    android:orientation="vertical"
    android:padding="0dip" >

    <TextView
        android:id="@+id/textView2"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginBottom="5dip"
        android:layout_marginLeft="5dip"
        android:layout_marginRight="5dip"
        android:layout_marginTop="5dip"
        android:background="#ffe0e0e0"
        android:gravity="right"
        android:text="0"
        android:textSize="32dp" />

    <GridLayout
        android:id="@+id/gridLayout1"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" >
    </GridLayout>

</LinearLayout>
```

LinearLayout und TextView sind damit fertig definiert. Allerdings der GridLayout ist noch nicht ganz fertig, da fehlt noch der Inhalt, die Knöpfe. Würden wir diese in der XML Datei definieren, wäre das sehr viel Copy-und-Paste. Besser ist es diese dynamisch zu generieren:

```
public class CalculatorActivity extends Activity implements
View.OnClickListener {

    private final int NR_OF_COLUMNS = 4;

    private String[] btnLabels = {
            "7","8","9","/",
            "4","5","6","*",
            "1","2","3","-",
            "(","0",")","+",
            "C","DEL",".","="};
```

```
    private TextView tv;

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

        getScreenDimensions();

        tv = (TextView) findViewById(R.id.textView2);

        GridLayout gl = (GridLayout) findViewById(R.id.gridLayout1);
        gl.removeAllViews();
        gl.setColumnCount(NR_OF_COLUMNS);
        gl.setRowCount(5);

        Button[] btns = new Button[btnLabels.length];
        for (int i = 0; i < btnLabels.length; i++) {
            btns[i] = new Button(this);
            btns[i].setText(btnLabels[i]);
            btns[i].setLayoutParams( getLayoutParams(width, i) );
            btns[i].setOnClickListener(this);
            gl.addView(btns[i]);
        }
    }
    ...
}
```

Wir holen uns also eine Referenz auf das Gridlayout aus der XML Datei, setzen die Spalten- und Zeilenzahl, und fügen dann die Buttons ein. Jedem Button müssen wir noch ordentliche Layoutparameter verpassen, das machen wir über die Methode *getLayoutParams()*:

```
    private GridLayout.LayoutParams getLayoutParams(int width,int i) {
        GridLayout.LayoutParams param = new GridLayout.LayoutParams();
        // param.height = LayoutParams.WRAP_CONTENT;
        // param.width = LayoutParams.WRAP_CONTENT;
        // param.rightMargin = 5;
        // param.topMargin = 5;
        param.width = width / NR_OF_COLUMNS;
        param.setGravity(Gravity.CENTER);
        param.columnSpec = GridLayout.spec(i % NR_OF_COLUMNS);
        param.rowSpec = GridLayout.spec(i / NR_OF_COLUMNS);
        return param;
    }
```

Die Breite die unser GridLayout zur Verfügung hat, erhalten wir mit der *getScreenDimensions()* Methode:

```
    private void getScreenDimensions() {
        Display display = getWindowManager().getDefaultDisplay();
        Point size = new Point();
        display.getSize(size);
        width = size.x;
        height = size.y;
    }
```

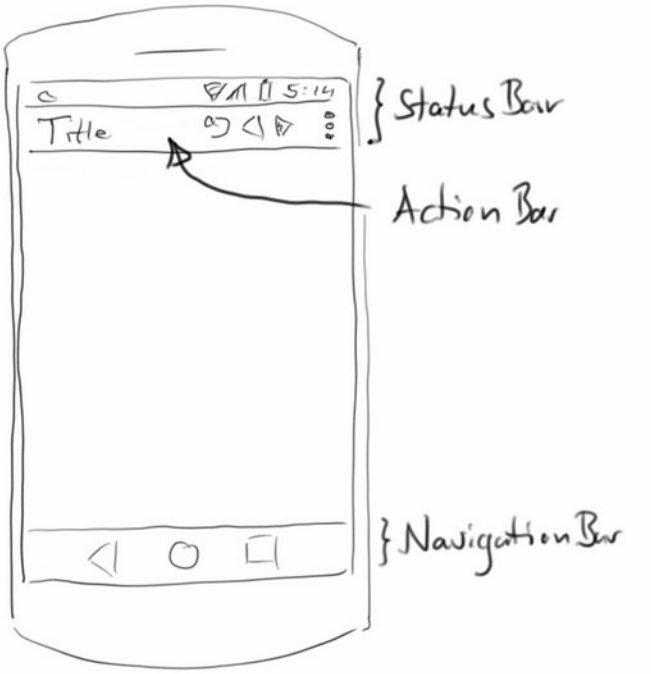
Fehlt nur noch die *onClick()* Methode: wenn eine Ziffer, ein Operator oder eine Klammer eingegeben wird, dann soll die im Display ausgegeben werden, bei einem 'C' soll das Display gelöscht werden, bei dem Drücken von 'DEL', soll das letzte Zeichen gelöscht werden, und beim Drücken des '=' Zeichens soll die Eingabe ausgewertet werden.

```
@Override
public void onClick(View v) {
    String lbl = ((Button) v).getText().toString();
    if ("0123456789()+-*/".contains(lbl)) {
        tv.append(lbl);
    } else if (lbl.equals("C")) {
        tv.setText("");
    } else if (lbl.equals("=")) {
        String infix = tv.getText().toString();
        String postfix = convertFromInfixToPostfix(infix);
        Log.i(TAG, postfix);
        int result = evaluate(postfix);
        tv.setText("" + result);
    } else if (lbl.equals("DEL")) {
        String txt = tv.getText().toString();
        tv.setText(txt.substring(0, txt.length() - 1));
    }
}
```

Die Methoden *convertFromInfixToPostfix()* und *evaluate()* kennen wir noch aus dem vierten Lab des zweiten Semester.

ScreenDimensions

Die Methode *getScreenDimensions()* oben ist nicht ganz korrekt. Denn der uns zur Verfügung stehende Screen Real Estate hängt zum einen natürlich von der Auflösung unseres Bildschirms ab. Aber auch davon ob Statusbar, Actionbar und/oder Navigationbar sichtbar sind.



Die Auflösung unseres Bildschirms erhalten wir mit der *getRealSize()* Methode des Displays:

```
...
String msg = "";
Display display =
    getWindowManager().getDefaultDisplay();
Point size = new Point();
display.getRealSize(size);
msg += "Real size: w=" + size.x + ", h=" + size.y + "\n";
...
```

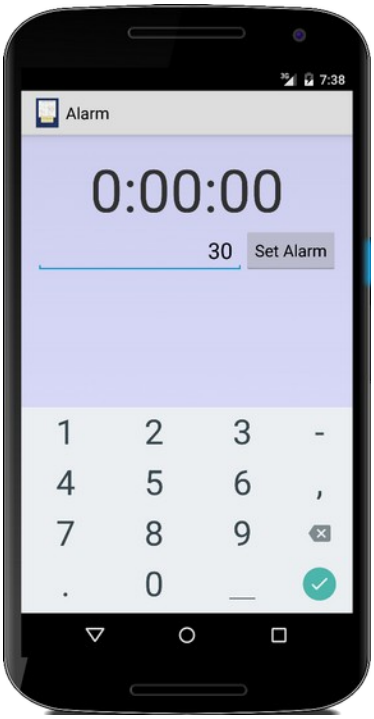
Je nachdem ob Statusbar, Actionbar und/oder Navigationbar sichtbar sind, müssen wir deren Höhe noch von der Höhe des Displays abziehen:

```
...
// status bar
int statusBarHeight = 0;
int resourceId = getResources()
    .getIdentifier("status_bar_height", "dimen", "android");
if (resourceId > 0) {
    statusBarHeight =
        getResources().getDimensionPixelSize(resourceId);
}
msg += "Status Bar: h=" + statusBarHeight + "\n";

// action bar
int actionBarHeight = 0;
final TypedArray styledAttributes =
    this.getTheme().obtainStyledAttributes(
        new int[] { android.R.attr.actionBarSize });
actionBarHeight = (int) styledAttributes.getDimension(0, 0);
styledAttributes.recycle();
msg += "Action Bar: h=" + actionBarHeight + "\n";

// navigation bar
int navigationBarHeight = 0;
resourceId = getResources()
    .getIdentifier("navigation_bar_height", "dimen", "android");
if (resourceId > 0) {
    navigationBarHeight =
        getResources().getDimensionPixelSize(resourceId);
}
msg += "Navigation Bar: h=" + navigationBarHeight;
...
```

Befindet sich unsere App allerdings im Landscape Modus, dann müssen wir die Navigationbar-Höhe von der Breite des Displays abziehen.

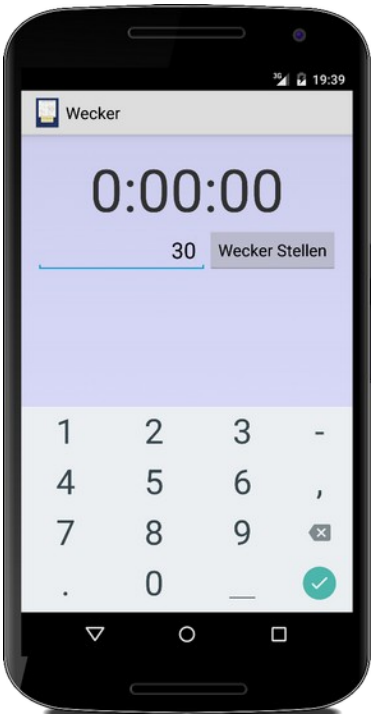
AlarmClock (i18n)

Wir haben inzwischen schon mehrere Vorzüge der Trennung von UI und Code mit Hilfe von XML Dateien gesehen. Ein weiterer ist die Internationalisierung (i18n), also das Anpassen einer Anwendung an verschiedene Sprachen. Die Activity macht wieder mal gar nichts:




```
public class AlarmClockActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedState) {
        super.onCreate(savedState);
        setContentView(
            R.layout.alarm_clock_activity);
    }
}
```

Auch das Layout ist nichts besonderes,




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

    <TextView
        android:id="@+id/textView"
        ...
        android:text="0:00:00" />

    <Button
        android:id="@+id/button"
        ...
        android:text="@string/set_alarm"
        android:textAllCaps="false" />

    <EditText
        android:id="@+id/editText"
        ...
        android:inputType="number"
        android:textSize="24dp" />

</RelativeLayout>
```

Außer, dass der Text des Buttons, "@string/set_alarm", eine Referenz auf einen Eintrag in der Datei */res/values/string.xml* ist:

```
<resources>
    <string name="app_name">Alarm</string>
    <string name="set_alarm">Set Alarm</string>
    <string name="action_settings">Settings</string>
</resources>
```

Wenn wir jetzt unsere Activity in andere Sprachen übersetzen wollen, müssen wir lediglich diese eine Datei übersetzen, für jede unterstützte Sprache eine zusätzliche Datei. Gespeichert werden diese Dateien in unterschiedlichen Verzeichnissen, z.B., die deutsche Übersetzung in dem Verzeichnis */res/values-de/:*

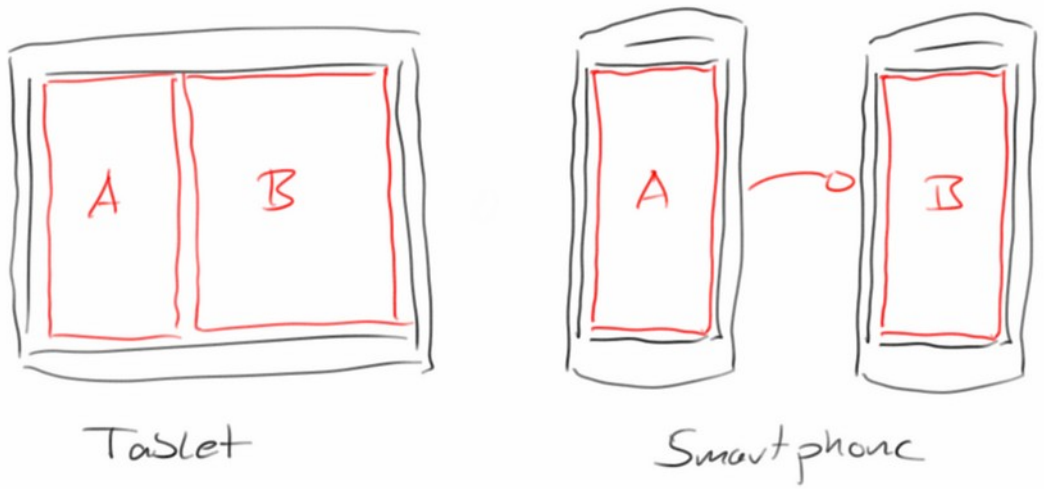
```
<resources>
    <string name="app_name">Wecker</string>
    <string name="set_alarm">Wecker Stellen</string>
    <string name="action_settings">Einstellungen</string>
</resources>
```

An den Systemeinstellungen des Handys erkennt Android welche Sprache der Nutzer eingestellt hat und verwendet diese dann automatisch. Wir müssen gar nichts tun. Wenn wir das testen wollen, müssen wir lediglich in den Systemeinstellungen des Handys eine andere Sprache einstellen.

Quotes

Wenn wir unsere bisherigen Anwendungen mal auf einem Tablet laufen lassen, werden wir schnell feststellen, dass die meisten etwas komisch aussehen. Die verschwenden so viel Platz. Um trotz größerer Bildschirme immer noch ästhetisch einigermaßen ansprechende Apps schreiben zu können, gibt es die Fragments.

Auf einem Tablet kann ich mehr Information anzeigen als auf dem kleinen Bildschirm eines Telefons. Die Idee ist einfach: nimm zwei der kleinen Telefonbildschirme und zeige die auf dem Tablet nebeneinander an.



Als Anwendung wollen wir einen kleinen Zitatebrowser schreiben. Auf dem Handy soll zunächst eine Liste mit Berühmtheiten erscheinen, und wenn man einen dieser Leute anklickt, sollen auf dem nächsten Schirm ein paar Zitate angezeigt werden. Auf dem Tablet hingegen sollen die Berühmtheiten links und die Zitate rechts nebeneinander erscheinen.

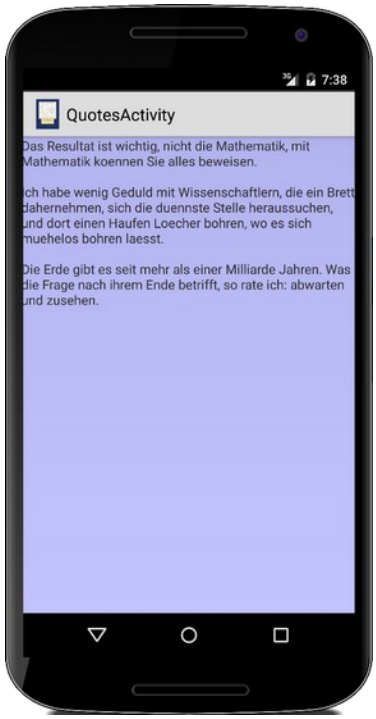
Wir beginnen mit den Layouts: die Activity agiert in diesem Beispiel lediglich als Container für die Fragments. Dabei müssen wir zwei Fälle unterscheiden. Für das Tablet definieren wir das Layout in der Datei quotes_activity.xml im Verzeichnis /res/layout-land/ (*land* steht für Landscape):

```
<LinearLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:id="@+id/frags"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#200000ff"
    android:baselineAligned="false"
    android:orientation="horizontal" >

    <fragment
        android:id="@+id/names_fragment"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="1"
        class="variationenzumthema_ch2.QuotesNamesFragment" />
```

```
        <fragment
            android:id="@+id/quotes_fragment"
            android:layout_width="0dp"
            android:layout_height="match_parent"
            android:layout_weight="3"
            class="variationenzumthema_ch2.QuotesQuotesFragment" />

    </LinearLayout>
```

.

Für das Telefon, definieren wir das Layout auch in einer Datei *quotes_activity.xml*, die ist aber im Verzeichnis /res/layout/ (also da wo es normalerweise immer ist):




```
    <FrameLayout xmlns:android=
        "http://schemas.android.com/apk/res/android"
        android:id="@+id/fragment_container"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="#200000ff" />
```

Wir definieren hier nicht viel, weil wie wir weiter unten sehen werden, auf dem Telefon alles von Hand gemacht werden muss. Wir haben also zwei verschiedene Layoutdateien, eine für das Tablet und eine für das Telefon.

Kommen wir zu den Fragmenten: Für das erste Fragment gibt es keine Layoutdatei, da es sich um ein *ListFragment* handelt und dieses gar nicht verändert wird.

Die Layoutdatei für das zweite Fragment (*quotes_quotes.xml*), verwendet einen ScrollView, wie wir ihn von der AboutActivity her kennen, und einen TextView für die Zitate:

```
    <ScrollView xmlns:android=
        "http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="#200000ff" >

        <TextView
            android:id="@+id/quotes_view"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="Select a name to see quotes!" />

    </ScrollView>
```

Was den Code angeht, beginnen wir mit der einfachsten der drei Klassen, dem QuotesQuotesFragment:

```
    public class QuotesQuotesFragment extends Fragment {

        private String[] quotes;

        @Override
        public View onCreateView(LayoutInflater inflater,
                                 ViewGroup container,
                                 Bundle savedInstanceState) {
            return inflater.inflate(R.layout.quotes_quotes,
                                    container, false);
        }
```

```
    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        if (quotes == null) {
            quotes = getResources().getStringArray(R.array.Quotes);
        }
    }

    public void updateFeedDisplay(int position) {
        TextView tv =
            (TextView) getView().findViewById(R.id.quotes_view);
        tv.setText(quotes[position]);
    }
}
```

In der onCreateView() wird die UI aus der XML Layoutdatei geladen. In der onActivityCreated() füllen wir unser Zitate-Array mit den Zitaten aus der *strings.xml* Datei:

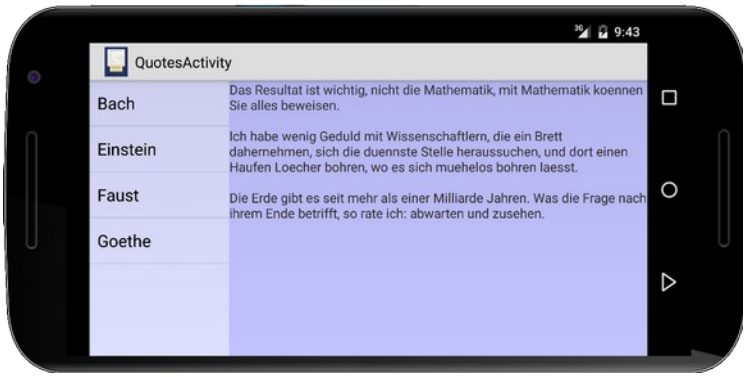
```
<resources>
    <string-array name="Names">
        <item>Bach</item>
        <item>Einstein</item>
        <item>Faust</item>
        <item>Goethe</item>
    </string-array>
    <string-array name="Quotes">
        <item>Alles, was man tun muß, ist die richtige ...</item>
        <item>Das Resultat ist wichtig, nicht die Mathe...</item>
        <item>So soll ich denn mit saurem Schweiss \nEu...</item>
        <item>Wer fremde Sprachen nicht kennt, weiss ni...</item>
    </string-array>
</resources>
```

Schließlich ist da noch die updateFeedDisplay() Methode: hier wird uns mitgeteilt welche der Zitate wir anzeigen sollen. Mitgeteilt wird uns das von der QuotesActivity.

Als nächstes sehen wir uns das QuotesNamesFragment für die Liste der Berühmtheiten an. Wir haben ein Stringarray für die Namen, die wir auch wieder aus der *strings.xml* Datei beziehen. Zusätzlich verknüpfen wir noch diese Namen mit der Liste unseres Fragments (wir sind ja ein ListFragment!):

```
public class QuotesNamesFragment extends ListFragment {

    private String[] names;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        names = getResources().getStringArray(R.array.Names);
        setListAdapter(
            new ArrayAdapter<String>(getActivity(),
                android.R.layout.simple_list_item_activated_1,
                names)
        );
    }

    @Override
    public void onListItemClick(ListView l, View view,
                                int position, long id) {
        ...
    }
    ...
}
```

Falls unser Nutzer jetzt auf einen der Namen klickt wird die *onListItemClick()* Methode aufgerufen, wo wir dann irgendetwas machen könnten.



Wenn das Fragment jetzt ganz für sich alleine leben würde, wären wir fertig. Aber wir wollen ja, dass die Zitate zu den entsprechenden Berühmtheiten im QuotesQuotesFragment angezeigt werden. Leider können wir mit dem QuotesQuotesFragment nicht direkt sprechen, da wir ja gar nicht wissen ob es existiert. Auf dem Tablet schon, auf dem Telefon dagegen nicht.

Deswegen müssen wir den Umweg über die Activity machen, die weiß nämlich Bescheid. Wir müssen also der Activity Bescheid geben können. Das heißt aber nichts anderes als dass wir eine Methode in der Activity aufrufen. Wie macht man sowas softwaretechnisch sauber? Über einen *Callback*.

Callbacks sind in der Regel Methoden, in unserem Fall die *onItemSelected()* Methode der QuotesActivity. Die rufen wir auf, wenn der Nutzer auf einen der Namen klickt. Dazu benötigen wir aber eine Referenz auf die Activity. Die erhalten wir in der *onAttach()* Methode die aufgerufen wird, wenn unser Fragment an die Activty angefügt (attached) wird.

```
public class QuotesNamesFragment extends ListFragment {
    ...
    private SelectionListener callback;

    @Override
    public void onAttach(Activity activity) {
        super.onAttach(activity);
        callback = (SelectionListener) activity;
    }

    @Override
    public void onListItemClick(ListView l, View view,
                                int position, long id) {
        callback.onItemSelected(position);
    }
}
```

Wie stellen wir sicher, dass es in der QuotesActivity auch wirklich die Methode *onItemSelected()* gibt? Das machen wir in dem wir ein Interface definieren, z.B.

```
public interface SelectionListener {
    public void onItemSelected(int position);
}
```

Es könnte auch FritzListener heißen, der Name ist egal, wichtig ist, dass es die Methode onItemSelected() fordert. Das mit dem Callback muss man nicht machen, aber es ist sauberes Softwareengineering.

Nach all der Vorarbeit kommen wir zum Herzen unserer Anwendung der *QuotesActivity*. Wie wir gesehen haben, muss sie das *SelectionListener* Interface und damit die Methode onItemSelected() implementieren.

```
public class QuotesActivity extends Activity
    implements SelectionListener {

    private QuotesNamesFragment namesFragment;
    private QuotesQuotesFragment quotesFragment;
```

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.quotes_activity);

        if (isInSinglePaneMode()) {
            quotesFragment = new QuotesQuotesFragment();
            namesFragment = new QuotesNamesFragment();

            FragmentManager mFragmentManager = getFragmentManager();
            FragmentTransaction fragmentTransaction =
                mFragmentManager.beginTransaction();
            fragmentTransaction.replace(R.id.fragment_container,
                                        namesFragment);
            fragmentTransaction.commit();

        } else {
            quotesFragment=(QuotesQuotesFragment)getFragmentManager()
                .findFragmentById(R.id.quotes_fragment);
        }
    }

    private boolean isInSinglePaneMode() {
        return findViewById(R.id.fragment_container) != null;
    }

    public void onItemSelected(int position) {
        if (isInSinglePaneMode()) {
            FragmentTransaction fragmentTransaction =
                getFragmentManager().beginTransaction();
            fragmentTransaction.replace(R.id.fragment_container,
                                        quotesFragment);
            fragmentTransaction.addToBackStack(null);
            fragmentTransaction.commit();

            getFragmentManager().executePendingTransactions();
        }
        quotesFragment.updateQuotesDisplay(position);
    }
}
```

Die QuotesActivity ist sozusagen der Manager: über die Methode *isInSinglePaneMode()* stellt sie fest, ob sie auf einem Telefon (true) oder einem Tablet (false) läuft.

Tablet is easy: es werden immer beide Fragments angezeigt. Das führt in der onCreate() Methode dazu, dass die beiden Fragmente automatisch erzeugt werden (sind ja in der quotes_activity.xml definiert). Wir benötigen lediglich noch eine Referenz auf das QuotesQuotesFragment.

Beim Telefon ist es etwas komplizierter: da ist ja in der Layoutdatei quotes_activity.xml gar nix definiert. Deswegen legen wir erst einmal beide Fragmente von Hand an, und sagen dem FragmentManager, dass er anstelle von uns (replace) doch das QuotesNamesFragment anzeigen soll. D.h. der Nutzer sieht erst einmal nur die Namen.

Wählt der Nutzer jetzt einen der Namen aus, wird die onItemSelected() Methode aufgerufen. Beim Tablet wieder easy: wir zeigen einfach die Zitate im QuotesQuotesFragment an. Beim Telefon müssen wir das QuotesNamesFragment durch das QuotesQuotesFragment ersetzen. Das geht wieder über den FragmentManager. Wenn jetzt das QuotesQuotesFragment sichtbar ist, können wir dort dann, wie beim Tablet, die richtigen Zitate anzeigen. Das war's.

Nachdem wir gesehen haben, dass das mit den Fragmenten etwas komplizierter ist, verstehen wir vielleicht warum wir im Rest des Buchs selbige nicht so häufig verwenden werden. Übrigens kann man das Telefon auch drehen, dann geht es in den Landscape-Modus.

Tabs

Eine Möglichkeit den begrenzten Platz auf dem Telefon etwas größer zu machen sind Fragments. Eine andere sind Tabs. Die sind um einiges einfacher. Hier gibt es eine Activity und eine Layoutdatei. Wir sehen uns zunächst das Layout an. Die ist vielleicht etwas lang, aber nicht weiter kompliziert:

```
<LinearLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#200000ff"
    android:orientation="vertical" >

    <TabHost
        android:id="@+id/tabHost"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true" >

        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:orientation="vertical" >

            <HorizontalScrollView
                android:layout_width="fill_parent"
                android:layout_height="wrap_content"
                android:fillViewport="true"
                android:scrollbars="none" >

                <TabWidget
                    android:id="@android:id/tabs"
                    android:layout_width="wrap_content"
                    android:layout_height="wrap_content" >
                </TabWidget>
            </HorizontalScrollView>

            <FrameLayout
                android:id="@android:id/tabcontent"
                android:layout_width="match_parent"
                android:layout_height="match_parent" >

                <!-- Tab-0 -->
                <LinearLayout
                    android:id="@+id/tab0"
                    android:layout_width="match_parent"
                    android:layout_height="match_parent"
                    android:background="#100000ff"
                    android:orientation="vertical" >

                    <TextView
                        android:layout_width="match_parent"
                        android:layout_height="match_parent"
                        android:text="Tab 0" />
                </LinearLayout>

                <!-- Tab-1 -->
                <LinearLayout
                    android:id="@+id/tab1"
                    android:layout_width="match_parent"
```

```
                    android:layout_height="match_parent"
                    android:background="#1000ff00"
                    android:orientation="vertical" >

                    <TextView
                        android:layout_width="match_parent"
                        android:layout_height="match_parent"
                        android:text="Tab 1" />
                </LinearLayout>
            </FrameLayout>
        </LinearLayout>
    </TabHost>
</LinearLayout>
```

Der *TabHost* gibt den ganzen Rahmen vor, alles was darin definiert wird gehört zu den Tabs. Das nächste sind die *TabWidgets*: das sind die Reiter mit denen man die verschiedenen Tabs auswählen kann. Die Tabs selbst sind wieder in einem FrameLayout definiert.

Die Activity ist auch ganz einfach. Zunächst basteln wir die UI automatisch aus der Layoutdatei.

```
public class TabsActivity extends Activity {

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

        // ui setup
        final TabHost host = (TabHost) findViewById(R.id.tabHost);
        host.setup();

        // Tab 0
        TabHost.TabSpec spec = host.newTabSpec("Tab Zero");
        spec.setContent(R.id.tab0);
        spec.setIndicator("First Tab");
        host.addTab(spec);

        // Tab 1
        spec = host.newTabSpec("Tab One");
        spec.setContent(R.id.tab1);
        spec.setIndicator("Second Tab");
        host.addTab(spec);

        host.setOnTabChangedListener(new TabHost.OnTabChangeListener()
        {
            public void onTabChanged(String tabId) {
                showToast(host.getCurrentTab());
            }
        });
    }

    private void showToast(int currentTab) {
        Toast.makeText(this,
                       "Tab" + currentTab + " was triggered.",
                       Toast.LENGTH_SHORT).show();
    }
}
```

Danach holen wir uns eine Referenz auf den Tabhost, und fügen einen Tab nach dem anderen zum Host hinzu. Schließlich registrieren wir noch einen OnTabChangeListener, der aufgerufen wird wenn unser Nutzer auf einen der Reiter klickt. Interessant ist vielleicht noch das *final* vor TabHost: das ist nötig, damit wir in der *onTabChanged()* Methode darauf zugreifen dürfen.

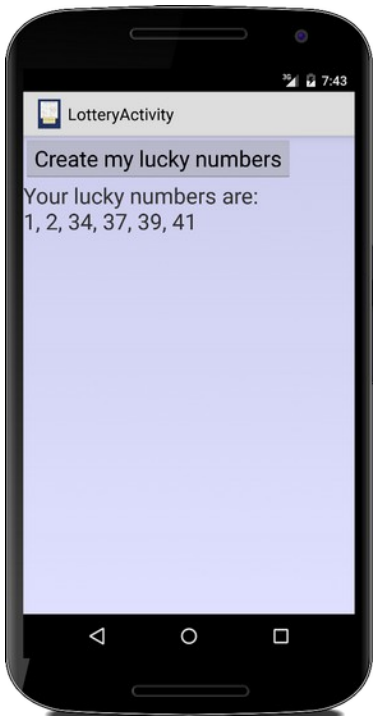
Review

Halten wir noch einmal kurz fest was wir bisher gelernt haben. Wir haben gesehen wie man eine UI in Android über XML Dateien gerätespezifisch festlegen kann. Dabei haben wir auch das erste Mal was über Ressourcen und die R.java Datei gehört. Des weiteren haben wir Beispiele für einen Dialog ein Menu gesehen, sowie verschiedene Layouts kennengelernt. Das Thema Internationalisierung (i18n) haben wir angedeutet, und mit Fragmenten haben wir uns etwas ausführlicher beschäftigt. Abschließend haben wir uns noch kurz die Tabs angeschaut.

Projekte

In den Projekten in diesem Kapitel werden wir viele alte Bekannte aus dem zweiten Semester treffen. Der Sinn dahinter ist zweifach: zum einen wollen wir die verschiedenen Android UI Widgets kennenlernen, gleichzeitig aber vielleicht das eine oder andere wieder auffrischen. Ein paar neue Dinge sind aber auch dabei.

Lottery



Wir wollen einen kleinen Lottozahlengenerator schreiben. Dabei haben wir die Vorarbeit schon im zweiten Semester geleistet: mithilfe von Zufallszahlen und der Set Datenstruktur ist das ganz einfach. Was uns zu tun bleibt ist die Portierung nach Android. Dazu beginnen wir mit dem User Interface, das wir wieder in einer XML Datei definieren, einen Button und einen TextView:

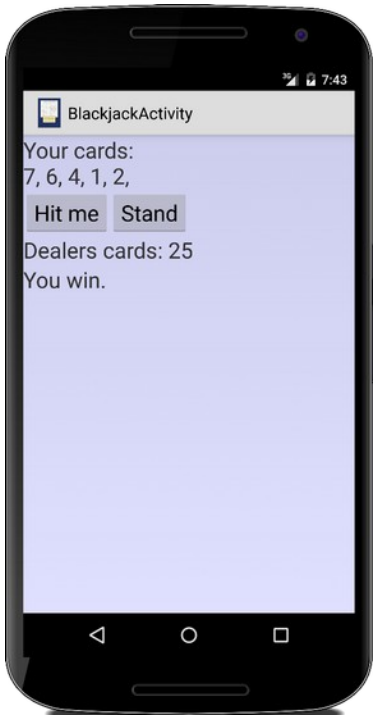
```
<LinearLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#200000ff"
    android:orientation="vertical" >

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:fontFamily="sans-serif"
        android:text="Create my lucky numbers"
        android:textSize="24sp" />

    <TextView
        android:id="@+id/result"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text=""
        android:textSize="24sp" />

</LinearLayout>
```

Im Code setzen wir den ContentView, dann holen wir uns eine Referenz auf den TextView und den Button. Wenn der Knopf gedrückt wurde, erzeugen wir die Lottozahlen und zeigen sie um TextView an.

```
public class LotteryActivity extends Activity {

    private TextView result;

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

        result = (TextView) findViewById(R.id.result);

        Button btnCreateLuckyNumbers =
            (Button) findViewById(R.id.button);
        btnCreateLuckyNumbers.setOnClickListener(
            new View.OnClickListener() {
              public void onClick(View v) {
                String numbers = generateLotteryNumbers();
                result.setText("Your lucky numbers are:\n" + numbers);
              }
            }
        );
    }
    ...
}
```

Da der TextView *result* nur in der Methode onCreate() verwendet wird, könnte er auch eine lokale Variable sein, müsste dann aber *final* sein.

Falls Sie so eine Anwendung im Google Play Store veröffentlichen wollen, sollten Sie unbedingt einen Disclaimer hinzufügen, sonst werden Sie von irgendeinem Dödl verklagt, weil er beim Lotto nix gewonnen hat.



Blackjack

Im Blackjack Beispiel geht es darum mal den RelativeLayout auszuprobieren. Wir brauchen vier TextViews und zwei Buttons. Die Buttons sollen nebeneinander, die Views untereinander angeordnet sein, so wie wir es im Screenshot sehen. Im Code holen wir uns wieder die Referenzen auf alle benötigten Widgets und fügen noch den OnClickListener zu den Knöpfen hinzu.

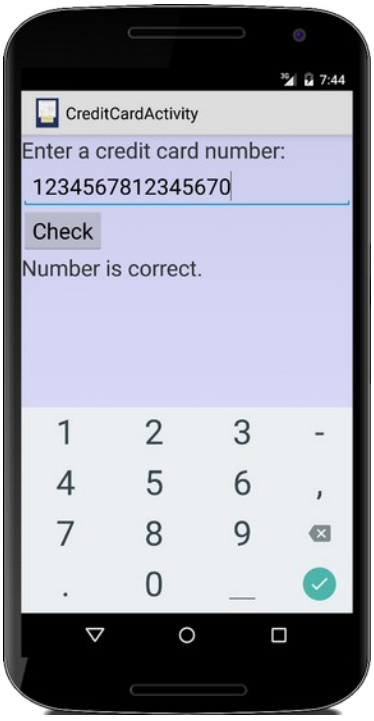
```
public class BlackjackActivity extends Activity {

    private List<Integer> cards =
        new ArrayList<Integer>();

    private TextView tvPlayerCards;
    private TextView tvDealerCards;
    private TextView tvWinOrLoose;

    @Override
    protected void onCreate(Bundle savedState) {
        super.onCreate(savedState);
        setContentView(R.layout.blackjack_activity);

        tvPlayerCards = (TextView) findViewById(R.id.player);
        tvDealerCards = (TextView) findViewById(R.id.dealer);
        tvWinOrLoose = (TextView) findViewById(R.id.win_or_loose);
```

```
            Button btnHit = (Button) findViewById(R.id.btn_hit);
            btnHit.setOnClickListener(new View.OnClickListener() {
                public void onClick(View v) {
                    cards.add(nextRandomInt(1, 11));
                    showPlayerCards();
                }
            });

            Button btnStand = (Button) findViewById(R.id.btn_stand);
            btnStand.setOnClickListener(new View.OnClickListener() {
                public void onClick(View v) {
                    checkResult();
                }
            });

            cards.add(nextRandomInt(1, 11));
            cards.add(nextRandomInt(1, 11));
            showPlayerCards();
        }
        ...
    }
```

Die Logik für das Spiel haben wir ja schon im zweiten Semester geschrieben.

CreditCard

Kreditkarten verwenden den sogenannten Luhn Algorithmus um festzustellen, ob sich bei der eingegebenen Nummer ein Tippfehler eingeschlichen hat. Wir können das in eine kleine App zum Checken von Kreditkarten verwandeln. Dazu brauchen wir zwei TextViews, einen EditText und einen Button:




```
    <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:background="#200000ff"
        android:orientation="vertical" >

        <TextView
            android:id="@+id/label"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Enter a credit card number:"
            android:textSize="24sp" />

        <EditText
            android:id="@+id/editText"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:inputType="number"
            android:textSize="24sp" />

        <Button
            android:id="@+id/button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:fontFamily="sans-serif"
            android:text="Check"
            android:textAllCaps="false"
            android:textSize="24sp" />
```

```
    <TextView
        android:id="@+id/result"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text=""
        android:textSize="24sp" />

</LinearLayout>
```

Interessant ist hier, dass wir dem EditText über *android:inputType="number"* sagen können, dass er nur Zahlen akzeptieren soll. Probieren Sie mal ob er auch Gleitkommazahlen akzeptiert.

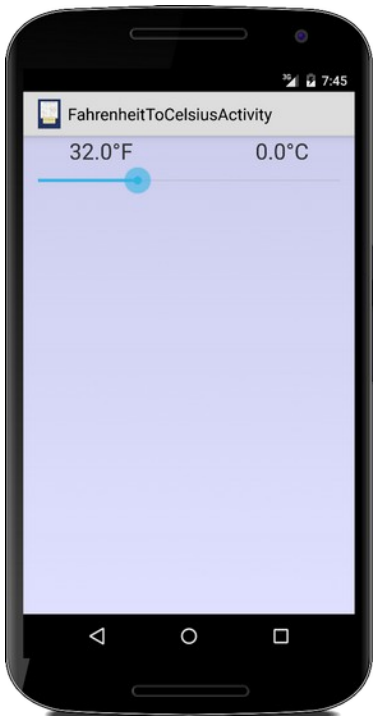
WordGuess

Kommen wir zu Hangman 2.0. Dafür benötigen wir einen TextView, um anzuzeigen welche Buchstaben schon richtig erraten wurden, einen EditText damit der Nutzer raten kann, und einen Button um zu checken ob der Buchstabe im gesuchten Wort ist. Der Code ist auch wieder trivial, zwei neue Dinge können wir aber beim Layout lernen:

```
<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:background="#200000ff"
    android:orientation="vertical" >

    <TextView
        android:id="@+id/word_shown"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:text="mother"
        android:textSize="24sp" />

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center_horizontal"
        android:orientation="horizontal" >

        <EditText
            android:id="@+id/char_entered"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:inputType="text"
            android:maxLength="1"
            android:textSize="24sp" />

        <Button
            android:id="@+id/btn_check_word"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:fontFamily="sans-serif"
            android:text="Check"
            android:textSize="24sp" />
    </LinearLayout>

</LinearLayout>
```

Zum einen sehen wir wie man LinearLayouts ineinander verschachteln kann, z.B. einen vertikal und den anderen horizontal. Zum anderen sehen wir wie man die Eingabe einer EditText auf ein Zeichen beschränken kann. Im nächsten Kapitel lernen wir wie man denn auch das dazugehörige Galgenmännchen zeichnen kann.

FahrenheitToCelsius



Die *SeekBar* Klasse haben wir bisher noch nicht verwendet. Eine schöne Anwendung ist ein kleiner Umrechner der Fahrenheit in Celsius und umgekehrt umwandelt. Man sieht auch schön wieder wie man mittels der Verschachtelung verschiedener Layouts meist an sein Ziel kommt.

Wir wollen in der ersten Zeile links den Fahrenheit TextView und rechts den Celsius TextView haben, darunter soll dann der SeekBar über die ganze Breite der Activity gehen:

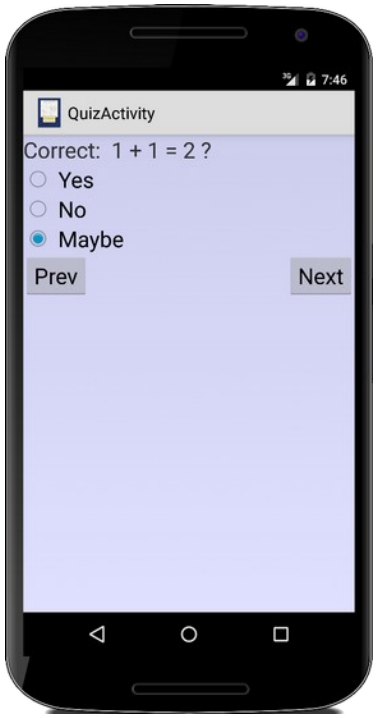
```
<LinearLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#200000ff"
    android:orientation="vertical" >

    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal" >

        <TextView
            android:id="@+id/celsius"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_alignParentRight="true"
            android:layout_alignParentTop="true"
            android:layout_marginRight="50dp"
            android:text="37.8°C"
            android:textSize="24sp" />

        <TextView
            android:id="@+id/fahrenheit"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_alignParentLeft="true"
            android:layout_alignParentTop="true"
            android:layout_marginLeft="50dp"
            android:text="100.0°F"
            android:textSize="24sp" />

    </RelativeLayout>

    <SeekBar
        android:id="@+id/seekbar"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:max="100"
        android:progress="50" />

</LinearLayout>
```

Damit das Ganze auf Inputs des Nutzers reagiert, müssen wir am SeekBar einen OnSeekBarChangeListener anbringen und dessen *onProgressChanged()* Methode überschreiben.

```
public class FahrenheitToCelsiusActivity extends Activity {

    private TextView tvFahrenheit;
    private TextView tvCelsius;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.fahrenheit_to_celsius_activity);

        tvFahrenheit = (TextView) findViewById(R.id.fahrenheit);
        tvCelsius = (TextView) findViewById(R.id.celsius);

        SeekBar seekbar = (SeekBar) findViewById(R.id.seekbar);
        seekbar.setOnSeekBarChangeListener(
          new OnSeekBarChangeListener() {

            @Override
            public void onStopTrackingTouch(SeekBar seekBar) {
            }

            @Override
            public void onStartTrackingTouch(SeekBar seekBar) {
            }

            @Override
            public void onProgressChanged(SeekBar seekBar,
                                          int progress, boolean fromUser) {
              double f = convertProgressToFahrenheit(progress);
              double c = fahrenheitToCelsius(f);

              tvFahrenheit.setText(String.format("%1$,.1f",f)+"°F");
              tvCelsius.setText("" + String.format("%1$,.1f",c)+"°C");
            }
          }
        );

    }

    private int convertProgressToFahrenheit(int progress) {
        return (4 * progress) - 100;
    }

    private double fahrenheitToCelsius(double f) {
        double c = (5.0 / 9.0) * (f - 32.0);
        return c;
    }

}
```

Ursprünglich wollte ich eigentlich den NumberPicker für diese Activity verwenden, der funktioniert aber nicht für negative Zahlen! In Indien wäre das vielleicht o.k., nicht aber bei uns.

Quiz



Wir wollen eine Anwendung für Multiplechoice Quizes erstellen. Im ersten Semester hatten wir uns dazu schon mal Gedanken gemacht. Zunächst überlegen wir uns den Logikteil der Activity. Die Fragen sollen erst einmal aus einem Stringarray kommen. Später könnten die auch von einem Server im Internet kommen. Wir parsen die Strings und laden die Fragen in eine Liste:

```
public class QuizActivity extends Activity {

    private final String[] QUESTIONS = {
        "Correct:  1 + 1 = 2 ?; Yes; No; Maybe",
        "What is 2 ^ 2 ?; 2; 4; 8",
        "A zebra has stripes?; Yes; No" };

    private List<Question> questions;
    private int currentQuestion = 0;

    @Override
    public void onCreate(Bundle savedState) {
        super.onCreate(savedState);

        loadQuestions();
        setContentView(createUI());
    }

    private void loadQuestions() {
        questions = new ArrayList<Question>();
        for (int i = 0; i < QUESTIONS.length; i++) {
            String[] words = QUESTIONS[i].split(";");
            Question q = new Question(words);
            questions.add(q);
        }
    }
    ...
}
```

Die Klasse Question ist so einfach wie möglich, lässt deswegen aber auch einige Softwareengineering-Prinzipien außer Acht:

```
class Question {
    public String question;
    public String[] answers;

    public Question(String[] words) {
        question = words[0];
        answers = words;
    }
}
```

Worum es uns eigentlich geht ist die UI. Die Fragen ändern sich nämlich, deswegen muss mindestens dieser Teil der UI dynamisch erzeugt werden:

```
    private LinearLayout createQuestionView(Question q) {
        LinearLayout ll2 = new LinearLayout(this);
        ll2.setOrientation(LinearLayout.VERTICAL);

        TextView tvQuestion = new TextView(this);
        tvQuestion.setText(q.question);
        tvQuestion.setGravity(Gravity.LEFT);
        tvQuestion.setTextSize(24);
        ll2.addView(tvQuestion);
```

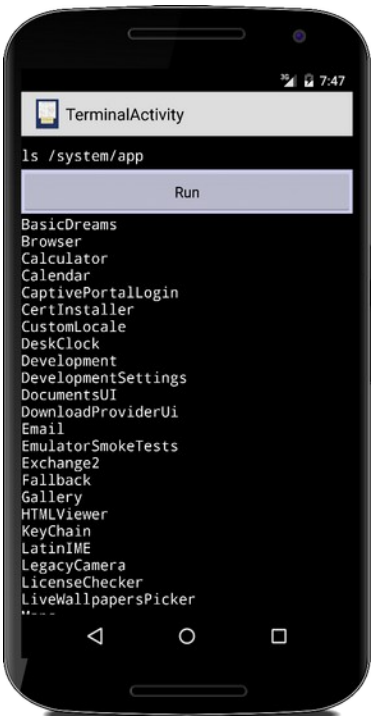
```
            RadioGroup rg = new RadioGroup(this);
            for (int i = 1; i < q.answers.length; i++) {
                RadioButton rb1 = new RadioButton(this);
                rb1.setTextSize(24);
                rb1.setText(q.answers[i]);
                rg.addView(rb1);
            }
            ll2.addView(rg);

            return ll2;
        }
```

Dieses Layout fügen wir dann an entsprechender Stelle in den Rahmen ein. Ob der Teil aus einer XML Datei kommt oder dynamisch erzeugt wird spielt keine Rolle. Wenn man die Erzeugung der UI in eine Methode *createUI()* auslagert, dann muss der Button der zur nächsten Frage führt, folgenden OnClickListener haben:

```
            ...
            btnNext.setOnClickListener(new OnClickListener() {

                @Override
                public void onClick(View v) {
                    Toast.makeText(v.getContext(), "Next!",
                        Toast.LENGTH_SHORT).show();
                    if (currentQuestion < questions.size() - 1) {
                        currentQuestion++;
                        setContentView(createUI());
                    }
                }
            });
            ...
```

und ähnlich für den "Zurück" Knopf.

Terminal



Wir haben ja bereits am Anfang dieses Buchs angedeutet, dass es sich bei Android eigentlich um ein Linux Betriebssystem handelt. Das wollen wir jetzt auch unter Beweis stellen indem wir eine einfach Terminal Anwendung schreiben.

Die UI besteht aus drei Widgets, einem AutoCompleteTextView, einem Button und einem TextView mit Scrollbars.

```
    <LinearLayout xmlns:android=
        "http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="#200000ff"
        android:orientation="vertical" >

        <AutoCompleteTextView
            android:id="@+id/autocomplete"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:background="#ff000000"
            android:fontFamily="monospace"
            android:hint="ls /system/app"
            android:inputType="text"
            android:paddingBottom="5dp"
            android:paddingTop="10dp"
```

```
            android:textColor="#ffffffff"
            android:textSize="16sp" />

        <Button
            android:id="@+id/button"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:fontFamily="sans-serif"
            android:text="Run"
            android:textSize="16sp" />

        <TextView
            android:id="@+id/result"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:background="#ff000000"
            android:fontFamily="monospace"
            android:scrollbars="vertical|horizontal"
            android:text=""
            android:textColor="#ffffffff"
            android:textSize="16sp" />

    </LinearLayout>
```

Der Code ist eigentlich auch nichts besonderes:

```
    public class TerminalActivity extends Activity {

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

            final TextView tv = (TextView) findViewById(R.id.result);
            tv.setMovementMethod(new ScrollingMovementMethod());

            final AutoCompleteTextView et =
                (AutoCompleteTextView) findViewById(R.id.autocomplete);
            String[] commands = new String[] {
                    "ps", "pwd", "cat /proc/cpuinfo",
                    "ls /storage/emulated/0",
                    "cd /storage/emulated/0", "whoami" };
            ArrayAdapter<String> adapter =
                new ArrayAdapter<String>(this,
                    android.R.layout.simple_spinner_item, commands);
            et.setAdapter(adapter);

            Button btn = (Button) findViewById(R.id.button);
            btn.setOnClickListener(new View.OnClickListener() {

                @Override
                public void onClick(View arg0) {
                    String command = et.getText().toString();
                    String output = execute(command);
                    tv.setText(output);
                }
            });

        }
        ...
    }
```

Wir haben das jetzt schon zweimal gesehen, und jetzt schon wieder: das dämliche *final*! Wenn wir es weglassen kompiliert unser Code aber nicht. Wir haben zwei Möglichkeiten:

- wir können aus den zwei Widgets (*tv* und *et*) Instanzvariablen machen, dann ist das final nicht nötig, oder
- wir lassen die zwei Widgets als lokale Variablen, dann ist das final aber zwingend nötig.

Da wir die Widgets aber in keiner anderen Methode verwenden, wäre aus softwaretechnischen Gründen die Variante mit den lokalen Variablen zu bevorzugen. Das final ist deswegen notwendig, da die *onClick()* Methode von einem anderen Thread (dem UI Thread) aus aufgerufen wird. Da wir in der *onClick()* Methode aber auf die beiden Widgets zugreifen, müssen die final sein! Das ist auch der Grund warum der Button nicht final sein muss. Auf den greifen wir ja nicht aus einem anderen Thread zu. Mehr zu Threads in einem späteren Kapitel.

Nach dieser kurzen Ablenkung wollen wir uns aber der eigentlichen Magie in unserem Code widmen, die steckt in der *execute()* Methode:

```
public String execute(String command) {
    StringBuffer output = new StringBuffer();
    Process p;
    try {
        p = Runtime.getRuntime().exec(command);
        p.waitFor();
        BufferedReader reader = new BufferedReader(
            new InputStreamReader(p.getInputStream()));

        String line = "";
        while ((line = reader.readLine()) != null) {
            output.append(line + "\n");
        }

    } catch (Exception e) {
        return e.toString();
    }
    return output.toString();
}
```

Über die Methode *exec()* der Klasse *Process*, die wir von der Runtime bekommen, können wir Linux Kommandos ausführen. Process gibt uns einen InputStream zurück, den wir dann einfach auslesen können. Übrigens hat Process auch einen OutputStream. Man kann damit alles machen, was man auch in einem Terminal oder einer Shell machen kann.

Allerdings hat man sehr eingeschränkte Rechte. Um wenigstens ein bisschen im Dateisystem lesen zu können, müssen wir unserer Anwendung Leseberechtigung geben. Das macht man indem man folgende Zeile

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

in die AndroidManifest.xml Datei einfügt.

TicTacToe

TicTacToe ist auch ein alter Bekannter, bereits aus dem ersten Semester. Damals hatten wir auch schon die Logik implementiert, es bleibt uns also nur noch die Portierung nach Android. Für das Layout macht es Sinn ein GridLayout zu verwenden:

```
<LinearLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:id="@+id/linearLayout1"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="#200000ff"
    android:gravity=
        "center|center_horizontal|center_vertical"
    android:orientation="vertical"
    android:padding="0dip" >

    <GridLayout
        android:id="@+id/gl_tictactoe"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" >
    </GridLayout>

</LinearLayout>
```

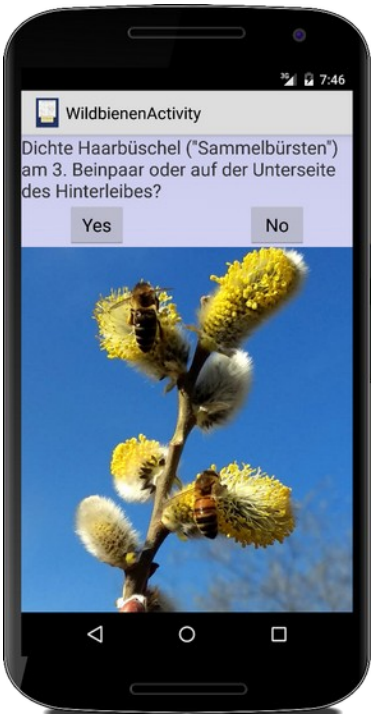
Im Code erzeugen wir die neun Buttons, als ImageButtons:

```
public class TicTacToeActivity extends Activity
                                implements View.OnClickListener {

    private final int NR_OF_COLUMNS = 3;
    private final int NR_OF_ROWS = 3;

    private TicTacToeLogic logic;
    private int currentPlayer = 1;
    private boolean gameOver = false;

    private int width = -1;
    private int height = -1;

    private ImageButton[] imgBtns;

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

        GridLayout gl = (GridLayout) findViewById(R.id.gl_tictactoe);
        gl.removeAllViews();
        gl.setColumnCount(NR_OF_COLUMNS);
        gl.setRowCount(NR_OF_ROWS);

        logic = new TicTacToeLogic();

        getScreenDimensions();

        // Button[] btns = new Button[btnLabels.length];
        imgBtns = new ImageButton[NR_OF_COLUMNS*NR_OF_ROWS];
        for (int i = 0; i < imgBtns.length; i++) {
            imgBtns[i] = new ImageButton(this);
```

```
                imgBtns[i].setId(i);
                imgBtns[i].setAdjustViewBounds(true);
                imgBtns[i].setMaxWidth(width / 3);
                imgBtns[i].setMaxHeight(width / 3);
                imgBtns[i].setImageResource(R.drawable.tictactoe_);
                imgBtns[i].setScaleType(
                    android.widget.ImageView.ScaleType.FIT_CENTER);
                imgBtns[i].setOnClickListener(this);
                gl.addView(imgBtns[i]);
            }
        }

        private void getScreenDimensions() {
            Display display = getWindowManager().getDefaultDisplay();
            Point size = new Point();
            display.getSize(size);
            width = size.x;
            height = size.y;
        }

        @Override
        public void onClick(View v) {
            if (!gameOver) {
                int id = ((ImageButton) v).getId();

                int i = id % NR_OF_COLUMNS;
                int j = id / NR_OF_COLUMNS;
                //Log.i(TAG, "id=" + id + ", i=" + i + ", j=" + j);

                if (logic.isMoveAllowed(currentPlayer, i, j)) {
                    displayPlayer(id);

                    if (logic.isGameOver()) {
                        displayGameOver();
                    } else {
                        computerMakesMove();
                    }

                } else {
                    Toast.makeText(this, "Move not allowed!",
                        Toast.LENGTH_SHORT).show();
                }
            }
        }

        private void displayPlayer(int id) {
            if (currentPlayer == 1) {
                imgBtns[id].setImageResource(R.drawable.tictactoe_x);
                imgBtns[id].setScaleType(
                    android.widget.ImageView.ScaleType.FIT_CENTER);
                currentPlayer = 2;
            } else {
                imgBtns[id].setImageResource(R.drawable.tictactoe_o);
                imgBtns[id].setScaleType(
                    android.widget.ImageView.ScaleType.FIT_CENTER);
                currentPlayer = 1;
            }
        }
        ...
    }
```

Die TicTacToeLogic Klasse übernehmen wir einfach aus dem ersten Semester.

Wildbienen



Im zweiten Semester haben wir uns mit Bäumen und Bienen beschäftigt. Eine der Hausaufgaben war Bienen mit Hilfe eines *Decisiontrees* zu klassifizieren. Falls wir noch die Hausaufgabe aus dem zweiten Semester haben, dann können wir die jetzt in eine App umwandeln.

Falls nicht, können wir uns den Flyer "WILDBIENEN" der Initiative "Deutschland summt!" herunterladen [1]. Auf Seite 5 dieses Flyers sehen wir eine Tabelle mit dem Namen "Wildbienen bestimmen - leicht gemacht!". Daraus können wir einen Entscheidungsbaum machen.

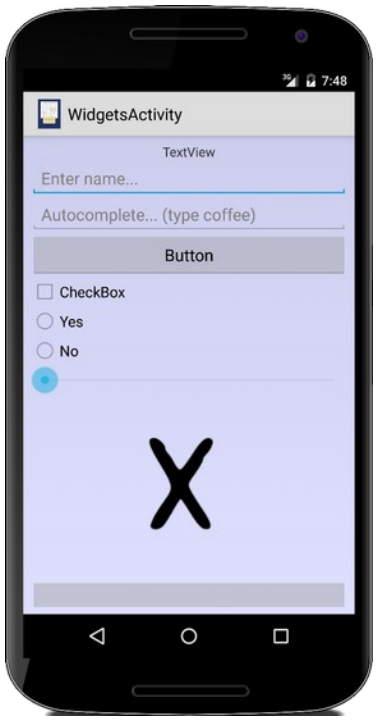
Wir beginnen mit der UI, die ähnelt der FahrenheitToCelsius Activity:

```
<LinearLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#200000ff"
    android:orientation="vertical" >

    <TextView
        android:id="@+id/question"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:lines="3"
        android:text="Wings attached?"
        android:textSize="20sp" />

    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal" >

        <Button
            android:id="@+id/btn_yes"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_alignParentLeft="true"
            android:layout_alignParentTop="true"
            android:layout_marginLeft="50dp"
            android:fontFamily="sans-serif"
            android:text="Yes"
            android:textSize="20sp" />

        <Button
            android:id="@+id/btn_no"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_alignParentRight="true"
            android:layout_alignParentTop="true"
            android:layout_marginRight="50dp"
            android:fontFamily="sans-serif"
            android:text="No"
            android:textSize="20sp" />
    </RelativeLayout>

    <ImageView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:scaleType="centerCrop"
        android:src="@drawable/wildbienen" />
</LinearLayout>
```

Der Code ist auch nicht so kompliziert:

```
public class WildbienenActivity extends Activity {
    private BinaryTree<String> decisions;
    private BinaryNode<String> currentNode;
    private TextView tv;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.wildbienen_activity);

        tv = (TextView) findViewById(R.id.question);
        Button btnYes = (Button) findViewById(R.id.btn_yes);
        btnYes.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                walkThrough("yes");
            }
        });
        Button btnNo = (Button) findViewById(R.id.btn_no);
        btnNo.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                walkThrough("yes");
            }
        });

        createDecisionTree();
        currentNode = (BinaryNode<String>) decisions.root();
        tv.setText(currentNode.getElement());
    }

    private void walkThrough(String anwser) {
        if (anwser.toLowerCase().equals("yes")) {
            currentNode = currentNode.getLeft();
        } else {
            currentNode = currentNode.getRight();
        }
        tv.setText(currentNode.getElement());
    }

    private void createDecisionTree() {
        // create decision tree, left is yes:
        BinaryNode<String> root = new BinaryNode<String>(
            "Dichte Haarbüschel (\"Sammelbürsten\") am 3. " +
            "Beinpaar oder auf der Unterseite des Hinterleibes?");

        BinaryNode<String> wheels =
            new BinaryNode<String>("Wheels attached?");
        root.setLeft(wheels);
        root.setRight(new BinaryNode<String>("Don't Fly!"));

        BinaryNode<String> water =
            new BinaryNode<String>("Water plane?");
        wheels.setLeft(new BinaryNode<String>("Fly!"));
        wheels.setRight(water);

        water.setLeft(new BinaryNode<String>("Fly!"));
        water.setRight(new BinaryNode<String>("Don't Fly!"));

        decisions = new BinaryTree<String>(root);
    }
}
```

Die Klassen BinaryNode und BinaryTree benötigen wir noch. Die haben wir im zweiten Semester verwendet. Wir importieren die einfach als JAR Bibliothek. Dazu müssen wir ein Verzeichnis /libs/ anlegen und die Datei *algorithms.jar* hinein kopieren. Diese müssen wir dann noch zum Buildpath hinzufügen.



Widgets

Welche UI Elemente, sogenannte Widgets, gibt es denn überhaupt? Ne ganze Menge. Hier haben wir mal eine Activity gebastelt die mal die wichtigsten zeigt:

```
...
TextView tv = new TextView(this);
tv.setText("TextView");
tv.setGravity(Gravity.CENTER);
ll.addView(tv);

EditText et = new EditText(this);
et.setHint("Enter name...");
ll.addView(et);

CheckBox cb = new CheckBox(this);
cb.setText("CheckBox");
ll.addView(cb);

Button btn = new Button(this);
btn.setText("Button");
btn.setTypeface( Typeface.create("sans-serif", Typeface.NORMAL));
ll.addView(btn);

RadioButton rb1 = new RadioButton(this);
rb1.setText("Yes");
RadioButton rb2 = new RadioButton(this);
rb2.setText("No");
RadioGroup rg = new RadioGroup(this);
rg.addView(rb1);
rg.addView(rb2);
ll.addView(rg);

SeekBar sb = new SeekBar(this);
ll.addView(sb);

ImageView img = new ImageView(this);
img.setImageResource(R.drawable.tictactoe_x);
ll.addView(img);

ImageButton imgBtn = new ImageButton(this);
imgBtn.setImageResource(R.drawable.tictactoe_o);
ll.addView(imgBtn);

List<String> items = new ArrayList<String>();
items.add("Coffee");
items.add("Espresso");
items.add("Capuccino");
items.add("Latte");
items.add("Cacachino");
ArrayAdapter<String> adapter =
    new ArrayAdapter<String>(this,
        android.R.layout.simple_spinner_item, items);
```

```
Spinner sp = new Spinner(this);
sp.setAdapter(adapter);
ll.addView(sp);

String[] commands =
    new String[] {
        "Coffee", "Espresso", "Capuccino", "Latte", "Cacachino" };
ArrayAdapter<String> adapter2 =
    new ArrayAdapter<String>(this,
        android.R.layout.simple_spinner_item, commands);
AutoCompleteTextView sp2 = new AutoCompleteTextView(this);
sp2.setAdapter(adapter2);
sp2.setHint("Autocomplete... (type coffee)");
ll.addView(sp2);
NumberPicker np = new NumberPicker(this);
np.setMinValue(1);
np.setMaxValue(10);
np.setLayoutParams(new LayoutParams(
    LayoutParams.WRAP_CONTENT, LayoutParams.WRAP_CONTENT));
ll.addView(np);

TimePicker tp = new TimePicker(this);
ll.addView(tp);

DatePicker dp = new DatePicker(this);
dp.setCalendarViewShown(false);
ll.addView(dp);

DatePicker dp2 = new DatePicker(this);
dp2.setSpinnersShown(false);
...
```

Ganz schön viele, oder? Und es gibt noch viel mehr...

IconMenu

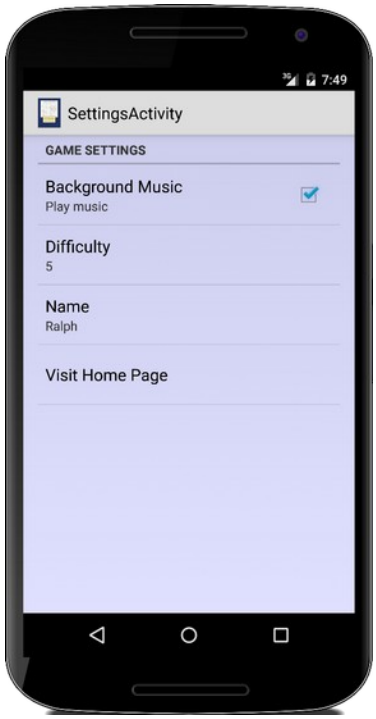
Wir haben ja schon gesehen wie wir ein Menu zu unserer Activity hinzufügen können. Wenn unser Menu nicht zu viele Einträge hat, und wenn genügend Platz zur Verfügung steht ("ifRoom"), dann kann man in der menu.xml Datei auch Icons angeben, die dann in der Titelzeile eingefügt werden.

```
<menu xmlns:android=
    "http://schemas.android.com/apk/res/android"
    xmlns:app=
    "http://schemas.android.com/apk/res-auto" >

    <item
        android:id="@+id/action_rotate_image"
        android:icon=
            "@drawable/ic_replay_black_24dp"
        android:showAsAction="ifRoom"
        android:title="Rotate"/>

    <item
        android:id="@+id/action_previous_image"
        android:icon=
            "@drawable/ic_skip_previous_black_24dp"
        android:showAsAction="ifRoom"
        android:title="Previous"/>
```

```
        <item
            android:id="@+id/action_next_image"
            android:icon="@drawable/ic_skip_next_black_24dp"
            android:showAsAction="ifRoom"
            android:title="Next"/>

    </menu>
```

Diese Bilder müssen im Verzeichnis /res/drawable/ zu finden sein.

Settings



Fast jede Anwendung hat irgendwelche Settings die der Nutzer einstellen kann. Z.B. ob die Hintergrundmusik spielen soll oder nicht, und ähnliches. Hierfür gibt es in Android die *SharedPreferences* wo solche Sachen zentral gespeichert werden können, ohne dass man sich da viel drum kümmern müsste. Es handelt sich um einen einfachen Key-Value Store (also eigentlich eine Map).

Interessant ist in diesem Zusammenhang die sogenannte *PreferenceActivity*: die macht das Anzeigen und Bearbeiten dieser Preferences ziemlich einfach. Denn die Inhalte definiert man einfach in der *settings.xml* Datei im Verzeichnis /res/xml/:

```
    <PreferenceScreen xmlns:android=
        "http://schemas.android.com/apk/res/android" >

        <PreferenceCategory
            android:key="game_settings"
            android:title="Game Settings" >
            <CheckBoxPreference
                android:defaultValue="true"
                android:key="music"
                android:summary="Play music"
                android:title="Background Music" />

            <variationenzumthema_pr2.NumberPickerPreference
                android:defaultValue="5"
                android:key="difficulty"
                android:title="Difficulty" />

            <EditTextPreference
                android:defaultValue="Ralph"
                android:dialogTitle="Enter players name"
                android:key="name"
                android:title="Name" />
        </PreferenceCategory>

        <Preference android:title="Visit Home Page" >
            <intent
                android:action="android.intent.action.VIEW"
                android:data="http://www.variationenzumthema.de" />
        </Preference>

    </PreferenceScreen>
```

Die erinnert so ein bisschen an eine Layout Datei.

Leider ist die PreferenceActivity etwas blöde, deswegen muss man sie erweitern, damit sie wirklich nützlich wird:

```
public class SettingsActivity extends PreferenceActivity
                    implements OnSharedPreferenceChangeListener {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        addPreferencesFromResource(R.xml.settings);

        // nasty trick to change background color:
        View root = ((ViewGroup)
            findViewById(android.R.id.content)).getChildAt(0);
        root.setBackgroundColor(0x200000ff);

        PreferenceManager.setDefaultValues(this,
                                           R.xml.settings, false);
        initSummary(getPreferenceScreen());

        // how to read preferences:
        SharedPreferences sharedPreferences =
            PreferenceManager.getDefaultSharedPreferences(this);
        Map<String, ?> allPreferences = sharedPreferences.getAll();
        for (String key : allPreferences.keySet()) {
            Log.i("SettingsActivity", "key:" + key +
                  ", value=" + allPreferences.get(key));
        }
    }

    @Override
    protected void onResume() {
        super.onResume();
        // Set up a listener whenever a key changes
        getPreferenceScreen().getSharedPreferences()
            .registerOnSharedPreferenceChangeListener(this);
    }

    @Override
    protected void onPause() {
        super.onPause();
        // Unregister the listener whenever a key changes
        getPreferenceScreen().getSharedPreferences()
            .unregisterOnSharedPreferenceChangeListener(this);
    }

    public void onSharedPreferenceChanged(
        SharedPreferences sharedPreferences, String key) {
        updatePrefSummary(findPreference(key));
    }

    private void initSummary(Preference p) {
        if (p instanceof PreferenceGroup) {
            PreferenceGroup pGrp = (PreferenceGroup) p;
            for (int i = 0; i < pGrp.getPreferenceCount(); i++) {
                initSummary(pGrp.getPreference(i));
            }
        } else {
            updatePrefSummary(p);
        }
    }
```

```
    private void updatePrefSummary(Preference p) {
        if (p instanceof ListPreference) {
            ListPreference listPref = (ListPreference) p;
            p.setSummary(listPref.getEntry());

        } else if (p instanceof EditTextPreference) {
            EditTextPreference editTextPref = (EditTextPreference) p;
            if (p.getTitle().toString().toLowerCase()
                                        .contains("password")) {
                p.setSummary("******");

            } else {
                p.setSummary(editTextPref.getText());
            }

        } else if (p instanceof MultiSelectListPreference) {
            EditTextPreference editTextPref = (EditTextPreference) p;
            p.setSummary(editTextPref.getText());

        } else if (p instanceof NumberPickerPreference) {
            NumberPickerPreference listPref =
                (NumberPickerPreference) p;
            p.setSummary("" + listPref.getValue());
        }
    }
}
```

In der Form ist sie aber wirklich hilfreich. Es sei noch angemerkt, dass die Klasse *NumberPickerPreference* ein Beispiel dafür ist, dass man auch seine eigenen DialogPreferences schreiben kann.

FileExplorer

Basierend auf der Quotes Activity wollen wir einen einfachen FileExplorer schreiben. Was das Userinterface angeht, sind die beiden faktisch identisch. Auch das Codegerüst ist gleich.

Etwas anpassen müssen wir den SelectionListener, da jetzt ein Verzeichnis als Ergebnis von dem ListFragment an die Activity übergeben werden soll:

```
public interface SelectionListener {
    public void onDirSelected(File dirName);
}
```

Was das ListFragment angeht, hält es beim FileExplorer die Verzeichnisse:

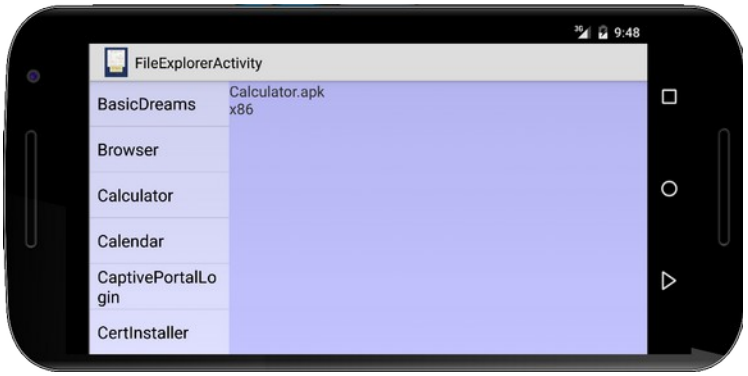
```
public class FileExplorerDirectoryFragment
                            extends ListFragment {

    private SelectionListener callback;
    private File[] dirs;

    @Override
    public void onCreate(Bundle savedState) {
        super.onCreate(savedState);

        dirs = getDirs(new File("/storage/emulated/0"));
```

```
            String[] fileNames = new String[dirs.length];
            for (int i = 0; i < fileNames.length; i++) {
                fileNames[i] = dirs[i].getName();
            }

            if (fileNames != null) {
                Arrays.sort(fileNames);
            }

            setListAdapter(
                new ArrayAdapter<String>(getActivity(),
                    android.R.layout.simple_list_item_activated_1,
                    fileNames));
        }
        ...
    }
```

Wir könnten den ArrayAdapter auch das Array von Verzeichnissen direkt übergeben, nur das sieht dann nicht so hübsch aus. Deswegen der Umweg über das Stringarray mit den Dateinamen.

Beim FileExplorerFileFragment zeigen wir in der Methode *updateFilesDisplay()* die Dateien des gewünschten Verzeichnis in einem TextView an:

```
public class FileExplorerFileFragment
                            extends Fragment {

    @Override
    public View onCreateView(
        LayoutInflater inflater,
        ViewGroup container,
        Bundle savedInstanceState) {
            return inflater.inflate(
                R.layout.fileexplorer_files,
container, false);
    }

    public void updateFilesDisplay(File dirName) {
        TextView mTextView = (TextView)
            getView().findViewById(R.id.fileexplorer_files_tv);
        String[] files = getAllFilesAndDirs(dirName);
        StringBuffer sb = new StringBuffer();
        if (files != null) {
            for (String file : files) {
                sb.append(file + "\n");
            }
        }
        mTextView.setText(sb.toString());
    }

    private String[] getAllFilesAndDirs(File file) {
        String[] filesAndDirs = file.list();
        if (filesAndDirs != null) {
            Arrays.sort(filesAndDirs);
        }
        return filesAndDirs;
    }
}
```



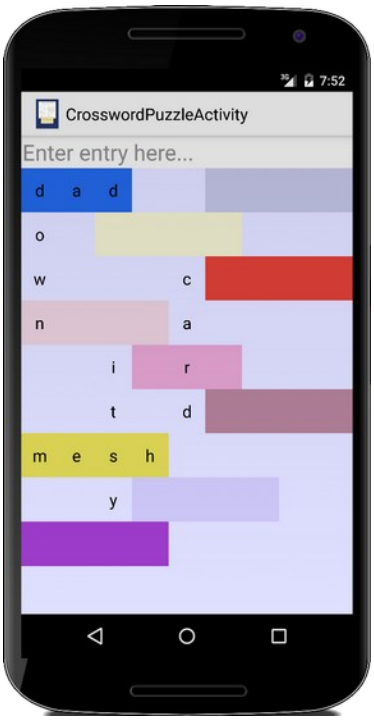
Die FileExplorerActivity ist eins-zu-eins identisch mit der QuotesActivity.

Da wir wieder ein bisschen im Dateisystem lesen wollen, müssen wir unserem FileExplorer Leseberechtigung geben. Wir fügen also wieder die Zeile

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

in unsere AndroidManifest.xml Datei ein.

Challenges



CrosswordPuzzle

Im zweiten Semester haben wir gesehen wie man mit Hilfe der Trie Datenstruktur automatisch Kreuzworträtsel generieren kann. Der Challenge liegt darin, wie man das so auf ein Smartphone portiert, dass es intuitiv benutzbar ist.

Als erstes bietet es sich an, die Wortfelder farblich zu hinterlegen. Klickt man auf eines dieser farbig hinterlegten Felder, dann erscheint in einem Toast der Lösungshinweis für das jeweilige Feld. Ein kleines Problem mit dem farblichen Hinterlegen ist, dass sich ein heilloses Durcheinander ergibt, wenn man nicht zwischen horizontalen und vertikalen Feldern trennt. Deswegen macht es Sinn zwischen einem horizontalen und einem vertikalen Modus zu unterscheiden. Das Umschalten zwischen dem einen und dem anderen kann man durch ein Menu oder einen Extraknopf erreichen. Man kann es aber auch durch das Klicken in ein leeres Feld erreichen.

Research

In diesem Kapitel gibt es ein paar Themen die man noch durch Eigenrecherche vertiefen kann.

Resolutions

Android Geräte kommen in allen möglichen Größen und Formen, die unter anderem auch mit Kürzeln wie ldpi, mdpi, hdpi, xhdpi, usw. bezeichnet werden. Referenz [4] gibt hier etwas Hintergrundinformationen. In der gleichen Referenz wird auch kurz auf den Unterschied zwischen *sp* und *dp* eingegangen.

Styles and Themes

Zu Styles und Themes kann man ein eigenes Buch schreiben. Einen ersten Einstieg liefern aber Referenzen [2] und [3]. Wir werden uns in diesem Buch allerdings auf einen sehr minimalistischen Stil beschränken.

Fragments

Dass Fragments nicht ganz trivial sind, dürfte aus der ungewöhnlichen Codefülle hervorgehen bei den beiden Beispielen die wir betrachtet haben. Es macht daher vielleicht Sinn herauszufinden warum es Fragments überhaupt gibt [5].

Final

Finden Sie heraus warum im Tabs Beispiel der TabHost als *final* deklariert sein muss? Auch bei der Lottery hatten wir eine ähnliche Wahl.

Fragen

1. Erklären Sie wofür die Datei 'R.java' gut ist.

2. Zeichnen Sie bitte die UI die von folgender 'main.xml' Datei erzeugt wird.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <Button
        android:id="@+id/btn_OK"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Open Browser" />

</LinearLayout>
```

3. Betrachten Sie den foglenden Code:

```
public class ButtonActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        // add an event handler to the button:
        View aboutButton = this.findViewById(R.id.btn_OK);

    }
}
```

   Welche Änderungen müssen Sie vornehmen, damit beim Klicken auf den Knopf eine Nachricht in die Log Datei geschrieben wird?

4. Warum verwendet man in Android XML Dateien um die UI zu beschreiben?

5. Nennen Sie drei verschiedene Layouts, die es in Android gibt.

6. Wofür wird die Datei “AndroidManifest.xml” verwendet? Geben Sie mindestens zwei Beispiele.

7. Betrachten Sie folgende UI eines Login Screens:

   Schreiben Sie die 'main.xml' Datei, die eine UI erzeugen würde, die der obigen Skizze möglichst ähnlich sieht.

8. Würden Sie sagen, dass Internationalisierung (i18n) in Android leicht oder schwer ist? Begründen Sie bitte Ihre Aussage.

9. Warum wurden Fragmente mit Android 3 eingeführt? Wofür sind sie gut?

Referenzen

[1] *WILDBIENEN*, Initiative "Deutschland summt!", http://www.deutschland-summt.de/?file=files/media_ds/pdfs/2017/Wildbienen_Folder_23-03-2017.pdf

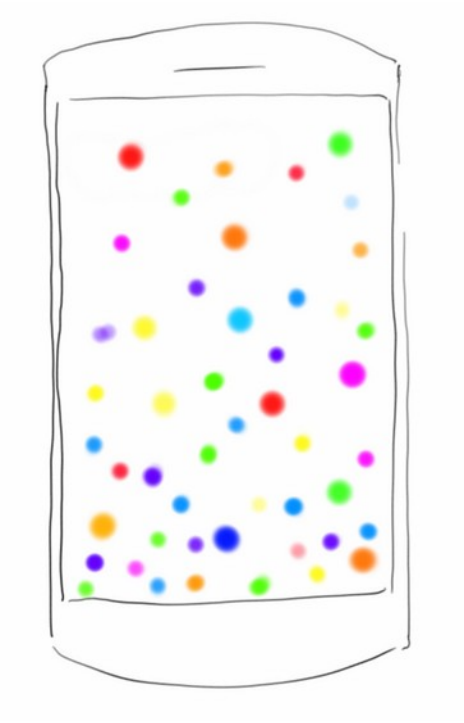
[2] Android - Styles and Themes, www.tutorialspoint.com/android/android_styles_and_themes.htm

[3] Styles and Themes, https://developer.android.com/guide/topics/ui/look-and-feel/themes

[4] Support different pixel densities, https://developer.android.com/training/multiscreen/screendensities

[5] Fragments, https://developer.android.com/guide/components/fragments

Graphics



Zu Benutzeroberflächen gehören nicht nur UI-Widgets, sondern auch 2D-Grafik. In diesem Kapitel wollen wir uns daher vor allem mit 2D-Grafik Anwendungen beschäftigen. Außerdem werden wir sehen wie Tasten- und vor allem Toucheingaben funktionieren, und auch verschiedene Gesten werden wir kennenlernen. Das eine oder andere Beispiel sollte uns aus dem ersten Buch bekannt sein.

View

Bevor wir mit unseren ersten Grafikprogrammen beginnen, müssen wir uns mit der *View* Klasse vertraut machen. Betrachten wir folgendes einfaches Beispiel:

```
public class GraphicsActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(new GraphicsView(this));
    }
}
```

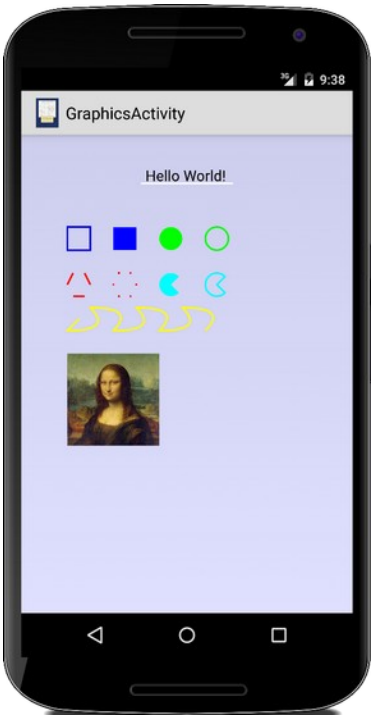
Der Unterschied zum letzten Kapitel ist, dass wir in der *setContentView()* Methode keine Referenz auf eine Resource geben, sondern unsere eigene Klasse, einen View, übergeben. Einen eigenen View zu schreiben ist denkbar einfach: wir erben von der Android View Klasse und überschreiben die *onDraw()* Methode:

```
class GraphicsView extends View {

    public GraphicsView(Context context) {
        super(context);
    }

    @Override
    protected void onDraw(Canvas canvas) {
        ...
    }
}
```

Alles was mit Zeichnen zu tun hat, passiert in der *onDraw()* Methode.

Graphics

Das Grafikmodell von Android unterscheidet sich ein wenig von dem objektorientierten Ansatz, den die ACM-Grafikbibliothek verwendet. Während wir im ACM-Fall GObjects, wie ein GRect oder einen GLabel, zum Zeichnen verwendet haben, verwendet Android die Canvas Klasse mit ihren Zeichenmethoden. Das Canvas Objekt wird uns als Argument in der *onDraw()* Methode übergeben:




```
protected void onDraw(Canvas canvas) {
    setBackgroundColor(0x200000ff);

    Paint paint = new Paint(Paint.ANTI_ALIAS_FLAG);
    paint.setColor(Color.BLACK);
    paint.setTextSize(64f);
    canvas.drawText("Hello World!",
        getWidth() / 2 - 180, 200, paint);
}
```

Wenn wir zum Beispiel Text auf unseren View schreiben wollen, verwenden wir die *drawText()* Methode des Canvas Objekts. Dabei gilt es noch zwei Dinge zu beachten:

- das Farbmodell von Android unterscheidet sich ein wenig von dem in Standard Java: in Android sind die Farben Integers, wobei die vier Bytes für den Alpha-, Rot-, Grün- und Blauwert stehen. Das erklärt die lustig aussehende Zahl "0x200000ff", die ein Integer in hexadezimaler Schreibweise ist. Es gibt auch die üblichen vordefinierten Farben, wie z.B. Color.BLACK. Und wir können natürlich auch unsere eigenen Farben mit der Methode Color.argb(127, 255, 0, 0) erstellen.
- die Klasse Paint: diese wird als Parameter an alle Zeichenmethoden übergeben. Sie spezifiziert Dinge, die Android wissen muss, wenn es Sachen zeichnen soll, wie die Farbe oder Schriftart die es verwenden soll, welche Schriftgröße, und ob geometrische Objekte ausgefüllt werden sollen oder nicht.

Die onDraw() Methode wird von Android jedes Mal aufgerufen, wenn der Bildschirm neu gezeichnet werden muss. Im Allgemeinen sollte man sich angewöhnen, so wenig wie möglich in dieser Methode zu tun, dann läuft unsere Anwendung auch schnell und reibungslos.

Drawing Methods

Betrachten wir die verschiedenen Zeichenmethoden die die Canvas Klasse unterstützt. Die drawText() Methode haben wir ja bereits oben gesehen. Als nächstes sehen wir uns an, wie wir Linien zeichnen:

```
...
paint.setStyle(Paint.Style.STROKE);
paint.setColor(Color.WHITE);
paint.setStrokeWidth(8);
canvas.drawLine(getWidth() / 2 - 200, 210,
    getWidth() / 2 + 200, 210, paint);
...
```

Mit der Methode *setStrokeWidth()* können wir die Dicke der Linie festlegen. Das Zeichnen von Rechtecken, Ovalen und Bögen folgt dem gleichen Muster:

```
...
float x = 200;
float y = 400;
float w = 100;
float h = 100;
paint.setColor(Color.BLUE);
canvas.drawRect(x, y, x + w, y + h, paint);

x += 200;
paint.setStyle(Paint.Style.FILL);
canvas.drawRect(x, y, x + w, y + h, paint);

x += 200;
paint.setColor(Color.GREEN);
canvas.drawOval(new RectF(x, y, x + w, y + h), paint);

x += 200;
paint.setColor(Color.GREEN);
paint.setStyle(Paint.Style.STROKE);
canvas.drawOval(new RectF(x, y, x + w, y + h), paint);

x += 200;
paint.setColor(Color.CYAN);
paint.setStyle(Paint.Style.FILL);
RectF mRectF = new RectF(x, y, x + w, y + h);
float startAngle = 45;
float sweepAngle = 270;
canvas.drawArc(mRectF, startAngle, sweepAngle, true, paint);
```

```
x += 200;
paint.setStyle(Paint.Style.STROKE);
mRectF = new RectF(x, y, x + w, y + h);
canvas.drawArc(mRectF, startAngle, sweepAngle, true, paint);
...
```

Wir können diese Objekte ausfüllen, indem wir den Stil des Paint Objektes auf *Paint.Style.FILL* setzen. Wenn wir nur den Rand dieser Objekte zeichnen wollen, setzen wir den Stil auf *Paint.Style.STROKE*. Interessant ist auch, dass *drawOval()* und *drawArc()* ein RectF als Parameter benötigen.

Zwei weitere interessante Methoden, die in der ACM-Grafikbibliothek keine Entsprechung haben, sind die *drawLines()* und die *drawPoints()* Methode:

```
...
x = 200;
y += 200;
float[] pts = { 200, 650, 225, 600, 275, 600,
                300, 650, 275, 700, 225, 700 };
paint.setColor(Color.RED);
paint.setStyle(Paint.Style.STROKE);
canvas.drawLines(pts, paint);

x += 200;
float[] pts2 = { 400, 650, 425, 600, 475, 600,
                 500, 650, 475, 700, 425, 700 };
canvas.drawPoints(pts2, paint);
...
```

Die zweite macht genau das, was man von ihr erwarten würde. Die erste zeichnet jedoch nur Liniensegmente. Wenn wir nach etwas suchen, das das GPolygon nachahmt, dann ist die *drawPath()* Methode wohl am ähnlichsten:

```
...
x = 200;
y += 250;
Path path = new Path();
path.reset();
path.moveTo(x, y);
for (int i = 0; i < 7; i++) {
    float x0 = x;
    x += 100;
    float y0 = y;
    y += ((i % 2) - 0.5) * 200;
    // path.lineTo(x, y);
    path.quadTo(x, y, x0, y0);
}
// path.close();
paint.setColor(Color.YELLOW);
canvas.drawPath(path, paint);
...
```

Wir beginnen mit einem leeren Pfad (Path), den wir allerdings noch mit *reset()* zurücksetzen müssen. Mit der *moveTo()* Methode fügen wir den Anfangspunkt zu unserem Pfad. Von wo aus wir dann zum nächsten Punkt und danach zu allen anderen Punkten gehen. Für die Verbindungen zwischen den Punkten können wir eine von drei Methoden verwenden:

- **lineTo():** verbindet zwei Punkte einfach mit einer geraden Linie.
- **quadTo():** verwendet eine quadratische Bezierkurve, um zwei Punkte zu verbinden.
- **cubicTo():** verwendet eine kubische Bezierkurve, um zwei Punkte zu verbinden.

Falls wir noch nie etwas von Bézier-Kurven[1] gehört haben, kann man in der Wikipedia nachlesen, wie die im Detail funktionieren.

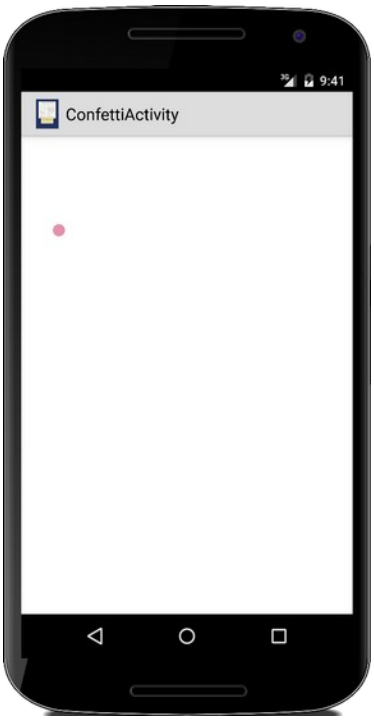
Schließlich können wir auch Bitmaps zeichnen:

```
...
x = 200;
y += 200;
try {
    InputStream is = getAssets().open("Mona_Lisa.jpg");
    Bitmap bitmap = BitmapFactory.decodeStream(is);
    canvas.drawBitmap(bitmap, x, y, null);
} catch (IOException e) {
    Log.e("GraphicsActivity", e.getMessage());
}
...
```

Wir haben also für jedes unserer ACM-Grafikobjekte eine entsprechende Android Canvas Zeichenmethode gefunden:

ACM Grafikobjekt	Android Zeichenmethode
GLabel	drawText()
GLine	drawLine()
GRect	drawRect()
GOval	drawOval()
GArc	drawArc()
GPolygon	drawPath()
GImage	drawBitmap()

Confetti



Um ein Gefühl dafür zu bekommen, wie das Zeichnen in Android funktioniert, beginnen wir mit einem Beispiel aus dem ersten Semester, dem Konfetti-Programm. Die Activity bleibt die gleiche wie oben. Wir müssen uns nur um den View kümmern:

```
class ConfettiView extends View {
    private Paint paint;
    private Random rgen = new Random();

    public ConfettiView(Context context) {
        super(context);
        paint = new Paint(Paint.ANTI_ALIAS_FLAG);
        paint.setStyle(Paint.Style.FILL);
    }

    @Override
    protected void onDraw(Canvas canvas) {
        // create randomly sized oval
        int w = SIZE / 2 + rgen.nextInt(SIZE);
        int x = -SIZE + rgen.nextInt(getWidth());
        int y = -SIZE + rgen.nextInt(getHeight());

        paint.setColor(rgen.nextInt());
        canvas.drawOval(new RectF(x, y, x + w, y + w), paint);

        pause(DELAY);
        invalidate();
    }
}
```

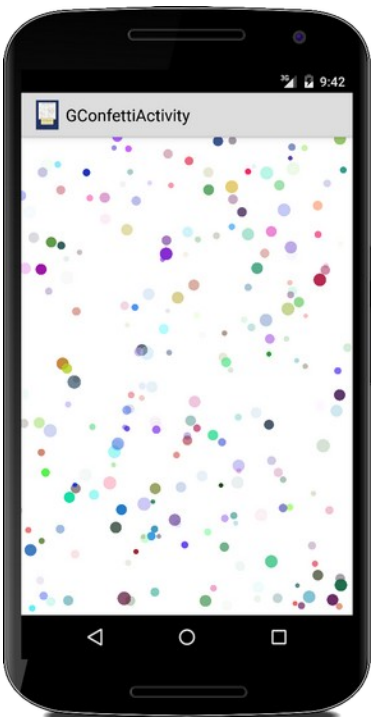
Vergleicht man den Code mit unserem vorherigen Beispiel, so sind die beiden sehr ähnlich: Wir zeichnen ein zufällig gefärbtes Oval an einer zufälligen Position auf dem Bildschirm mit der drawOval() Methode. Da wir mehr als ein Oval zeichnen wollen, pausieren wir ein wenig und rufen dann die *invalidate()* Methode auf. Die sagt dem Bildschirm, dass er sich neu zeichnen soll, also dass die onDraw() Methode wieder aufgerufen werden soll.

Wenn wir den Code ausführen sehen wir, dass er zwar Ovale zeichnet, aber immer wieder die zuvor gezeichneten Ovale löscht. Dies ist ein großer Unterschied zu dem was wir von der ACM Grafikbibliothek her gewohnt sind. Jedes Mal, wenn die onDraw()-Methode aufgerufen wird, wird der Bildschirm gelöscht. Für unser Confetti Programm ist das natürlich nicht sehr hilfreich.

ConfettiBitmap

Eine Lösungsansatz ist sich irgendwie daran zu erinnern, was man schon gezeichnet hat. Eine Möglichkeit ist eine Bitmap zu verwenden. Statt direkt auf die Leinwand zu zeichnen, zeichnen wir in eine Bitmap. Zuerst definieren wir die Bitmap, und da wir sie ja wiederverwenden wollen, tun wir das als Instanzvariable:

```
class ConfettiBitmapView extends View {
    private Bitmap bitmap;
    ...

    protected void onSizeChanged(int w, int h,
                                 int oldw, int oldh) {
        if (bitmap != null) {
            bitmap.recycle();
        }
        bitmap = Bitmap.createBitmap(w, h,
            Bitmap.Config.ARGB_8888);
    }

    @Override
    protected void onDraw(Canvas canvas) {
        ...
        Canvas can = new Canvas(bitmap);
        can.drawOval(new RectF(x, y, x + w, y + w), cPaint);

        canvas.drawBitmap(bitmap, 0, 0, null);
        ...
    }
}
```



Dann erstellen wir in der *onSizeChanged()* Methode eine neue Bitmap. Diese Methode wird immer dann aufgerufen, wenn sich die Größe des Bildschirms ändert, was beim Start der Aktivität und beim Drehen des Bildschirms geschieht. Der Code in der *onDraw()* Methode ist fast identisch, mit dem feinen Unterschied, dass wir das Oval in die Bitmap zeichnen. Aus der Bitmap können wir dann einen Canvas machen, und in den können wir dann ganz normal zeichnen. Zum Schluss müssen wir noch die Bitmap auf den echten Canvas zeichnen. Funktioniert großartig.

GConfetti

Ein anderer Lösungsansatz ist es aus den Confetti Objekte zu machen, so wie die GOvals in der ACM Bibliothek. Für die Confetti müssen wir uns ihre Position und Farbe merken. Also schreiben wir eine Klasse GGOval, mit den folgenden Instanzvariablen:

```
class GGOval {
    private int x, y, w, h;
    private Paint paint;
```

Im Constructor initialisieren wir diese dann:

```
    public GGOval(int x, int y, int w, int h) {
        this.x = x;
        this.y = y;
        this.w = w;
        this.h = h;
        this.paint = new Paint(Paint.ANTI_ALIAS_FLAG);
        this.paint.setStyle(Paint.Style.FILL);
    }
```

Wir benötigen noch eine setColor() Methode, um die Farbe zu setzen:

```
    public void setColor(int color) {
        paint.setColor(color);
    }
```

Und schließlich wäre es noch schön, wenn die Konfetti wüssten, wie sie sich selbst zeichnen sollen, wenn man ihnen einen Canvas gibt:

```
    public void draw(Canvas canvas) {
        canvas.drawOval(new RectF(this.x, this.y, this.x + this.w,
                                  this.y + this.h), paint);
    }
}
```

Das sieht schon sehr nach ACM Graphics Library aus, oder?

Also benutzen wir mal unsere GGOvals. In der GConfettiView Klasse instanziieren wir zunächst eine Liste von *confettis*. Dabei handelt es sich einfach um eine Liste von GGOvals:

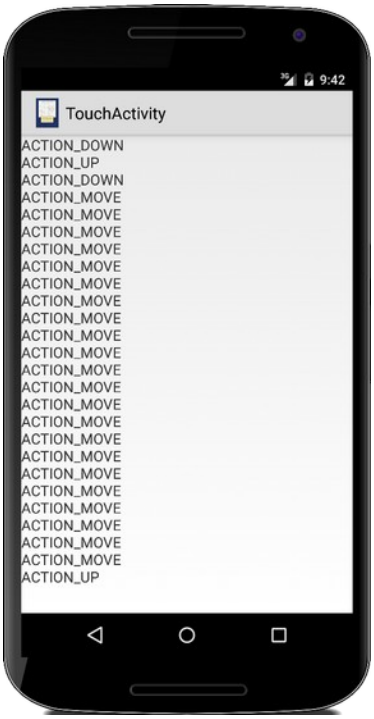
```
class GConfettiView extends View {
    private Random rgen = new Random();
    private List<GGOval> confettis = new ArrayList<GGOval>();

    public GConfettiView(Context context) {
        super(context);
    }

    public void addNewConfetti() {
        // create randomly sized oval
        int width = SIZE / 2 + rgen.nextInt(SIZE);
        int x = -SIZE + rgen.nextInt(getWidth());
        int y = -SIZE + rgen.nextInt(getHeight());
        GGOval oval = new GGOval(x, y, width, width);

        // assign random color
        oval.setColor(rgen.nextInt());

        // add to list
        confettis.add(oval);
    }

    @Override
    protected void onDraw(Canvas canvas) {
        setBackgroundColor(Color.WHITE);

        addNewConfetti();
```

```
            // draw all objects
            for (GGOval ov : confettis) {
                ov.draw(canvas);
            }

            pause(DELAY);
            invalidate();
        }
    }
```

In der onDraw() Methode erstellen wir mit der *addNewConfetti()* Methode bei jedem Aufruf ein neues Konfetti. In dieser Methode wird ein neues zufälliges GGOval erstellt und zu unserer Liste der Confettis hinzugefügt. Danach gehen wir unsere Liste von Konfetti durch und sagen jedem Konfetti, dass es sich auf unseren Canvas zeichnen soll. Das funktioniert auch ganz gut.

Wenn wir wollen, können wir auch alle anderen GObject Klassen aus der ACM-Grafikbibliothek implementieren. Sobald wir das Thema Concurrency abgehandelt haben, ist das auch der Plan: in den Special Topics werden wir die gesamte ACM-Grafikbibliothek implementieren. Das bedeutet, dass fast alle unsere alten GraphicsPrograms aus dem ersten und zweiten Semester jetzt auch auf Android laufen!

Touch



Der einfachste Weg, um Benutzer zur Interaktion mit unserer Anwendung zu bewegen, sind Touch-Events. Alles was wir tun müssen, ist die *onTouchEvent()* Methode zu überschreiben. Das kann man sowohl in der Activity als auch im View tun:

```
public class TouchActivity extends Activity {

    @Override
    public boolean onTouchEvent(MotionEvent event){
        int x = (int) event.getX();
        int y = (int) event.getY();
        switch (event.getAction()) {
        case (MotionEvent.ACTION_CANCEL):
            Log.i(TAG, "ACTION_CANCEL");
            return true;
        case (MotionEvent.ACTION_DOWN):
            Log.i(TAG, "ACTION_DOWN");
            return true;
        case (MotionEvent.ACTION_MOVE):
            Log.i(TAG, "ACTION_MOVE");
            return true;
        case (MotionEvent.ACTION_OUTSIDE):
            Log.i(TAG, "ACTION_OUTSIDE");
            return true;
        case (MotionEvent.ACTION_SCROLL):
            Log.i(TAG, "ACTION_SCROLL");
            return true;
        case (MotionEvent.ACTION_UP):
            Log.i(TAG, "ACTION_UP");
            return true;
        case (MotionEvent.ACTION_POINTER_2_DOWN):
            Log.i(TAG, "ACTION_POINTER_2_DOWN");
            return true;
        default:
            return super.onTouchEvent(event);
        }
    }
}
```

Die nützlichsten Events sind ACTION_DOWN, ACTION_MOVE und ACTION_UP. Interessant ist auch der ACTION_POINTER_2_DOWN Event, der entsteht wenn man mit mehr als nur einem Finger das Display berührt.

Über den MotionEvent Parameter der *onTouchEvent()* Methode erhalten wir die x- und y-Position, an der der Nutzer den Bildschirm berührt hat, aber auch den Zeitpunkt des Ereignisses, *getEventTime()*, und bei einigen Geräten sogar den Druck, *getPressure()*. Normalerweise verwenden wir diese Methoden nicht direkt, statt dessen werden sie vom Gestendetektor bei der Erkennung von Gesten verwendet.

Zusätzlich zu den Touch Events gibt es auch noch die Key Events, an die wir über die *onKeyDown()* Methode rankommen:

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
    Log.i(TAG, "keyCode=" + keyCode);
    return super.onKeyDown(keyCode, event);
}
```

Die wenigsten Smartphone haben heutzutage noch Tastaturen, aber für einige Tablets gibt es Bluetooth Tastaturen und dann macht auch der Key Event wieder Sinn.

Gestures

Wahrscheinlich die bekanntesten Gesten sind der Swipe-Left und der Swipe-Right einer bekannten Dating-App. Das geht sogar soweit, dass ein "left swipe" zur Geste der Ablehnung im digitalen Zeitalter geworden ist. Wie erkennt man also Gesten? Glücklicherweise erledigt Android die meiste Arbeit für uns. Wir verwenden dazu die Klasse *GestureDetector*. In der *onTouchEvent()* Methode senden wir alle Touch Events zur Analyse an den GestureDetector:

```
public class GestureActivity extends Activity {

    private GestureDetector mDetector;

    @Override
    public void onCreate(Bundle savedState) {
        super.onCreate(savedState);
        mDetector = new GestureDetector(
            this, new MyGestureListener());
    }

    @Override
    public boolean onTouchEvent(MotionEvent event){
        this.mDetector.onTouchEvent(event);
        return super.onTouchEvent(event);
    }

    private class MyGestureListener
                extends GestureDetector.SimpleOnGestureListener {
        private final String TAG = "MyGestureListener";

        // onDown should always return true,
        // because all gestures start with onDown
        @Override
        public boolean onDown(MotionEvent event) {
            Log.d(TAG, "onDown");
            return true;
        }
```

```
            @Override
            public boolean onFling(MotionEvent event1, MotionEvent event2,
                                   float velocityX, float velocityY) {
                Log.d(TAG, "onFling");
                return true;
            }

            @Override
            public void onLongPress(MotionEvent event) {
                Log.d(TAG, "onLongPress");
            }

            @Override
            public boolean onScroll(MotionEvent event1,MotionEvent event2,
                                    float distanceX, float distanceY) {
                Log.d(TAG, "onScroll");
                return true;
            }

            @Override
            public void onShowPress(MotionEvent event) {
                Log.d(TAG, "onShowPress");
            }

            @Override
            public boolean onSingleTapUp(MotionEvent event) {
                Log.d(TAG, "onSingleTapUp");
                return true;
            }

            @Override
            public boolean onDoubleTap(MotionEvent event) {
                Log.d(TAG, "onDoubleTap");
                return true;
            }

            @Override
            public boolean onDoubleTapEvent(MotionEvent event) {
                Log.d(TAG, "onDoubleTapEvent");
                return true;
            }

            @Override
            public boolean onSingleTapConfirmed(MotionEvent event) {
                Log.d(TAG, "onSingleTapConfirmed");
                return true;
            }
        }
    }
```

Die Namen der Methoden sind ziemlich selbsterklärend, vielleicht ist der Unterschied zwischen einem Scroll und einem Fling nicht ganz klar:

- **scroll:** die *onScroll()* Methode wird mehrmals aufgerufen, während der Benutzer seinen Finger über den Bildschirm bewegt.
- **fling:** die *onFling()* Methode wird nur einmal am Ende eines Scrollvorgangs aufgerufen.

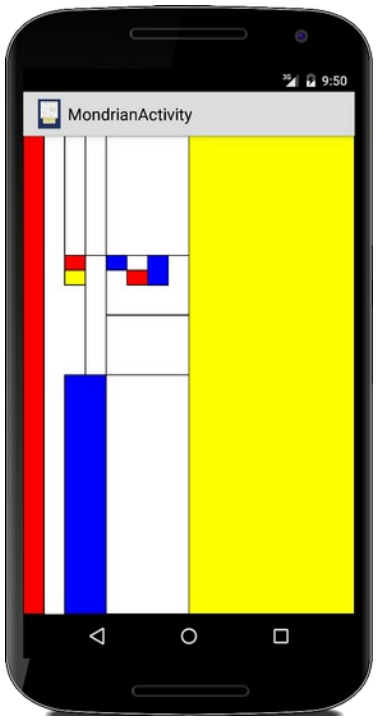
Ein Beispiel für sogenannte Multitouch Gesten werden wir noch in den Challenges sehen.

Review

In diesem Kapitel haben wir uns mit 2D Graphik beschäftigt und dabei die Klassen Color, Paint und Canvas kennengelernt. Wir haben auch gesehen wofür Views gut sind und wie man sie modifiziert. Schließlich haben wir noch ein bisschen was zu Touch und Key Events gehört, und unsere ersten einfachen Gesten erkannt.

Projekte

In diesem Kapitel gibt es nicht all zu viele Projekte, die meisten sind alte Bekannte aus dem ersten und zweiten Semester. Allerdings beschäftigen wir uns in den Special Topics "Graphics Performance" und "Libraries" noch einmal ganz ausführlich mit Graphics.



Mondrian

Im Kapitel zu Rekursion aus dem zweiten Buch haben wir Mondrians gezeichnet. Die Idee war die große Leinwand (canvas) in immer kleinere aufzuteilen und dabei eine der folgenden drei Optionen zu wählen:

- wir teilen die Leinwand horizontal in zwei kleinere Leinwände oder
- wir teilen die Leinwand vertikal in zwei kleinere Leinwände oder
- wir zeichnen ein Rechteck in einer der Farben weiß, rot, blau oder gelb.

Das wiederholen wir bis die Leinwände zu klein sind.

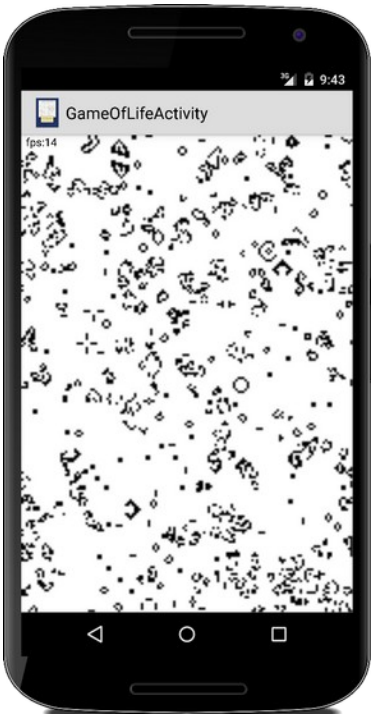
Das Beispiel von damals lässt sich auch recht einfach auf Android übertragen. Wir brauchen wieder eine Activity und einen View. In der onDraw() Methode des Views rufen wir dann die rekusive *drawMondrian()* Methode auf. Bis auf die *drawRectangle()* können wir den Code aus dem zweiten Semester einfach übernehmen.

Was jetzt noch schön wäre, wenn wir durch Berühren des Displays einen neuen Mondrian zeichnen lassen. Das geht ganz einfach, wenn wir im Constructor des Views noch einen OnTouchListener hinzufügen:

```
public MondrianView(Context context) {
    super(context);

    this.setOnTouchListener(new View.OnTouchListener() {
        @Override
        public boolean onTouch(View vw, MotionEvent event) {
            switch (event.getAction()) {
            case MotionEvent.ACTION_DOWN:
                invalidate();
                break;
            }
            return true;
        }
    });
}
```

Die *invalidate()* Methode veranlasst den View sich neu zu zeichnen. Das ist viel besser als die onDraw() Methode direkt aufzurufen (was übrigens auch gar nicht geht).

GameOfLife



Auch Conways Game of Life haben wir schon im zweiten Buch kennengelernt. Kurz zur Wiederholung: Das Universum des Spiel des Lebens ist ein zweidimensionales Gitter aus quadratischen Zellen (GRects), von denen jede in einem von zwei möglichen Zuständen sein kann: lebend (schwarz) oder tot (weiß). Jede Zelle hat acht Nachbarn, und abhängig vom Zustand der Nachbarn entscheidet sich der eigene Zustand in der nächsten Runde nach folgenden Regeln:

- jede lebende Zelle mit weniger als zwei lebenden Nachbarn stirbt (Unter-Bevölkerung),
- jede lebende Zelle mit zwei oder drei lebenden Nachbarn lebt,
- jede lebende Zelle mit mehr als drei lebenden Nachbarn stirbt (Über-Bevölkerung) und
- jede tote Zelle mit genau drei lebenden Nachbarn wird eine lebende Zelle (Fortpflanzung).

Erst einmal können wir einfach den Code aus dem zweiten Semester übernehmen und lediglich beim Zeichnen der kleinen Rechtecke anstelle der GRects die drawRect() Methode verwenden.

Für die Animation verwenden wir ähnlich wie beim GGRect Beispiel die *invalidate()* Methode am Ende der onDraw() Methode. Dann zeichnet sich das Spiel neu mit jedem nächsten Schritt. Der Game Loop findet also eigentlich in der onDraw() Methode statt.

Wir beobachten allerdings, dass die Animation nicht besonders schnell ist. Hier sehen wir das erste Mal ziemlich krass den Performanceunterschied zwischen einer ARM CPU mit 1.5 GHz und einer Intel CPU mit 2.4 GHz. Gefühlt ist die Intel CPU zehnmal schneller.

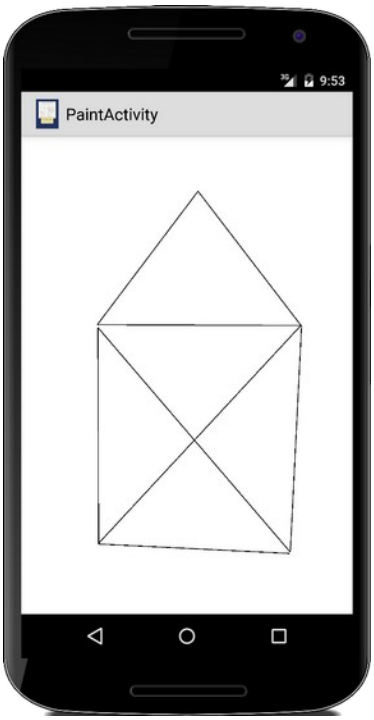
Jammern hilft uns jetzt aber nicht weiter. Im Beispiel Confetti haben wir gesehen wie wir eine Bitmap zum malen verwenden können. Das Gleiche können wir hier auch machen. Wir generieren eine Bitmap die genauso groß ist wie unser cell[][] Array, und verwenden dann einfach die *setPixel()* Methode der Bitmap Klasse, um die Pixel schwarz oder weiß zu malen. Bevor wir die Bitmap dann zeichnen, können wir noch einen kleinen Trick verwenden:

```
Matrix m = new Matrix();
m.setScale((float) mCanvasWidth / SIZE_X,
           (float) mCanvasHeight / SIZE_Y);
canvas.drawBitmap(bitmap, m, null);
```

wir skalieren die Bitmap so, dass sie den gesamten Canvas ausfüllt.

Was ich bisher auch nicht wusste, auch Conway's Game of Life ist wie Karel eine Universal Computing Machine, dazu später mehr unter Research.

Paint



Auch Paint haben wir schon mal gesehen, und zwar als ConnectTheClicks im ersten Semester. Dabei handelt es sich um ein Spiel in dem der Nutzer mit dem Finger eine Stelle berührt, und diese wird dann mit der vorhergehenden durch eine Linie verbunden.

Wenn wir das Spiel wie im ersten Semester implementieren, werden wir das gleiche Problem wie bei dem Beispiel Confetti haben: die alten Linien werden nicht mehr gezeichnet. Hier gibt es zwei Lösungen: entweder wir verwenden wieder eine Bitmap, oder aber, und das ist dieses Mal vielleicht geschickter, merken wir uns alle Punkte in einer Liste als Instanzvariable in der Activity:

```
private ArrayList<Point> points = new ArrayList<Point>();
```

Bei jedem onTouch Event fügen wir dann einen neuen Punkt hinzu und veranlassen einen Redraw mit invalidate():

```
points.add(new Point(x, y));
invalidate();
```

In der onDraw() Methode zeichnen wir dann einfach Linien zwischen den Punkten:

```
protected void onDraw(Canvas canvas) {
    ...
    if (points.size() > 1) {
        Point p0 = points.get(0);
        for (int i = 1; i < points.size(); i++) {
            Point p1 = points.get(i);
            canvas.drawLine(p0.x, p0.y, p1.x, p1.y, color);
            p0 = p1;
        }
    }
}
```

Warum haben wir die Liste von Punkten als Instanzvariable in der Activity und nicht im View gespeichert? Der Grund war weil wir noch etwas vor haben: wir wollen nämlich, dass wenn wir unser Gerät drehen, unsere Zeichnung nicht einfach verschwindet. Die beiden Methoden *onSaveInstanceState()* und *onRestoreInstanceState()*, werden nämlich von Android aufgerufen wenn unser Gerät sich dreht,

```
protected void onSaveInstanceState(Bundle outState) {
    outState.putSerializable("points", points);
}

protected void onRestoreInstanceState(Bundle savedInstanceState) {
    if (savedInstanceState != null) {
        Object obj = savedInstanceState.getSerializable("points");
        if (obj != null) {
            points = (ArrayList<Point>) obj;
        }
    }
}
```

und wir können sie verwenden, um unseren Zustand kurz im *Bundle* zwischenzuspeichern und dann wiederherzustellen. Wie das mit dem *Serializable* genau funktioniert werden wir uns im nächsten Kapitel noch einmal genauer ansehen.

FingerPaint

Das Paint Beispiel schaut doch ziemlich hässlich aus, oder? Inzwischen sind wir ja im 6. Semester, da sollten unsere Programme schon etwas hübscher aussehen. Hübscher heißt aber natürlich auch komplizierter, wenigstens etwas.

Die Klasse Path ist hier die Lösung. Wir haben sie am Anfang des Kapitels schon einmal kurz gesehen.

```
private class FingerPaintView extends View {

    private Paint paint;
    private Path path;

    private float curX, curY;
    private static final float TOUCH_TOLERANCE = 4;
```

```
        public FingerPaintView(Context c) {
            super(c);
            path = new Path();
            paint = new Paint();
            ...
        }

        protected void onDraw(Canvas canvas) {
            canvas.drawPath(path, paint);
        }

        ...
    }
```

In ihr speichern wir unseren Pfad. Die beiden Variablen curX und curY benötigen wir um uns zu merken wo der Nutzer das letzte Mal mit dem Finger war. Die onDraw() Methode ist gespenstisch einfach.

Wo es etwas komplizierter wird ist iń der onTouchEvent() Methode, wir wollen auf die drei Events ACTION_DOWN, ACTION_MOVE und ACTION_UP reagieren:

```
        public boolean onTouchEvent(MotionEvent event) {
            float x = event.getX();
            float y = event.getY();

            switch (event.getAction()) {
            case MotionEvent.ACTION_DOWN:
                touchDown(x, y);
                invalidate();
                break;
            case MotionEvent.ACTION_MOVE:
                touchMove(x, y);
                invalidate();
                break;
            case MotionEvent.ACTION_UP:
                touchUp();
                invalidate();
                break;
            }
            return true;
        }
```

Bei ACTION_DOWN wollen wir uns merken wo der Nutzer getoucht hat, und wollen den Anfang von unserem Pfad auf diesen Punkt legen:

```
        private void touchDown(float x, float y) {
            path.reset();
            path.moveTo(x, y);
            curX = x;
            curY = y;
        }
```

Bei ACTION_MOVE liegt der Hammer begraben:

```
        private void touchMove(float x, float y) {
            float dx = Math.abs(x - curX);
            float dy = Math.abs(y - curY);
            if (dx >= TOUCH_TOLERANCE || dy >= TOUCH_TOLERANCE) {
                path.quadTo(curX, curY, (x + curX) / 2, (y + curY) / 2);
                curX = x;
                curY = y;
            }
        }
```

Zunächst stellen wir fest, ob die Distanz zum vorhergehenden Punkt mindestens vier Pixel ist, nur dann fügen wir ein neues Stück zu unserem Pfad hinzu. Wir könnten hier einfach die Methode lineTo() verwenden, dann sieht das Ganze aber etwas eckig aus, eher wie Kindergarten. Wenn wir aber anstelle die Methode quadTo() verwenden, also Bézier Kurven zeichnen, dann sieht das echt cool aus.

Schließlich sollten wir noch bei ACTION_UP eine Linie zum letzten Punkt zeichnen:

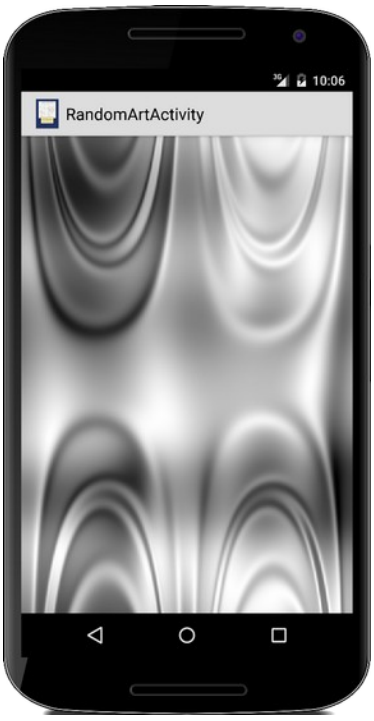
```
        private void touchUp() {
            path.lineTo(curX, curY);
        }
```

Wenn wir noch ein Grid also Hintergrund hinzufügen,

```
    private void drawGrid(Canvas canvas) {
        Paint gridPaint = new Paint();
        gridPaint.setAntiAlias(true);
        gridPaint.setDither(true);
        gridPaint.setColor(GRID_COLOR);
        gridPaint.setStyle(Paint.Style.STROKE);
        gridPaint.setStrokeWidth(1);
        for (int i = 0; i < getWidth(); i += 50) {
            canvas.drawLine(i, 0, i, getHeight(), gridPaint);
        }
        for (int i = 0; i < getHeight(); i += 50) {
            canvas.drawLine(0, i, getWidth(), i, gridPaint);
        }
    }
```

dann fühlt sich die ganze App noch einen Tick professioneller an. Wenn wir jetzt noch unsere Kunstwerke speichern könnten, dann könnten wir damit sogar Geld verdienen.

RandomArt



RandomArt war auch ein Projekt im zweiten Buch wo wir aus Mathematik, in diesem Fall dem Sinus und dem Cosinus, hübsche Graphiken erzeugt haben. Auch dieses Projekt lässt sich einfach in ein Android Projekt umwandeln. Nach unseren obigen Erfahrungen, macht es wahrscheinlich Sinn gleich mit einer Bitmap als grafischer Datenstruktur anzufangen, es sei denn wir wollen wieder etwas länger warten.

Was wir hier aber lernen wollen, ist wie das mit den Swipe-Left und Swipe-Right Gesten funktioniert. Wir wollen bei einer Swipe-Left Geste das erzeugte Bild verwerfen und eine neues erzeugen. Bei einem Swipe-Right hat uns das Bild aber gefallen und wir würden es gerne auf unserer SD Karte speichern.

Bisher haben wir eigentlich nur die Fling Geste kennen gelernt, die unterscheidet aber noch nicht zwischen rechts und links, oder oben und unten. Mit der Klasse *OnSwipeListener* (die nicht Teil der Android API ist) geht das aber ganz einfach. Im Constructor des Views

```
    public RandomArtView(final Context context) {
        super(context);

        setOnTouchListener(new OnSwipeListener(contxt){

            public void onSwipeRight() {
                saveAsPNG();
            }
```

```
            public void onSwipeLeft() {
                Toast.makeText(context, "calculating...",
                    Toast.LENGTH_SHORT).show();

                // needed for toast to become visible:
                Thread.yield();
                pause(100);
                Thread.yield();

                // do a redraw:
                postInvalidate();
            }
        });
    }
```

setzen wir den onTouchListener auf unseren OnSwipeListener. Dort überschreiben wir dann die beiden Methoden die uns interessieren, in dem Fall *onSwipeRight()* und *onSwipeLeftt()*.

Interessant ist vielleicht noch, dass wir hier *postInvalidate()* anstelle von *invalidate()* verwendet haben. Wenn wir nur mal so zum Testen *invalidate()* verwenden, stellen wir fest, dass der Toast unsichtbar bleibt. Also offensichtlich funktioniert es nicht so gut hier. Der Grund dafür ist, dass die Methode onSwipeLeft() eigentlich in einem anderen Thread ausgeführt wird, dem Event-Thread. Der View lebt aber im UI-Thread, und das invalidate() gehört zum UI Thread. Wenn wir also aus einem anderen Thread heraus dem UI-Thread sagen wollen, dass er sich doch neu zeichnen soll, dann müssen wir die *postInvalidate()* Methode verwenden.

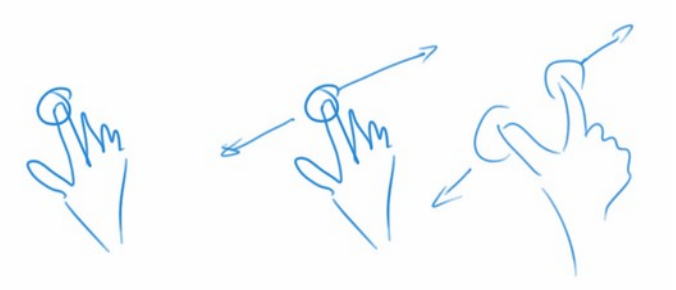
Obwohl wir noch gar nicht gelernt haben, wie man auf die SD-Karte schreibt, ist die *saveAsPNG()* Methode nur drei Zeilen lang und die sind ziemlich selbst erklärend:

```
    private void saveAsPNG() {
        try {
            OutputStream stream =
                new FileOutputStream("/sdcard/RandomArt.png");
            bitmap.compress(CompressFormat.PNG, 0, stream);
            stream.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

Aber im Kapitel Persistence mehr dazu.

Challenges

MultiTouch

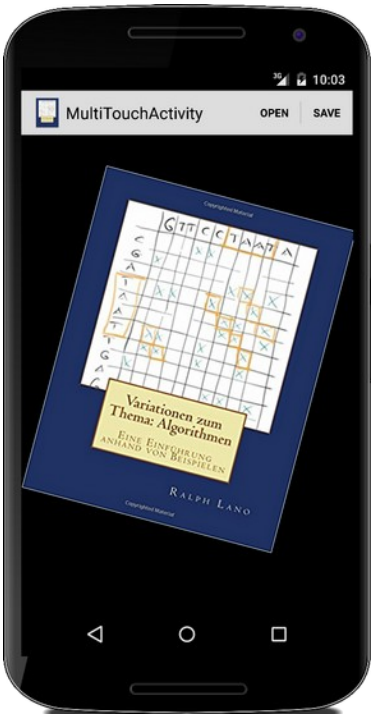


Wenn wir uns nochmal unser Beispiel zu Gesten ansehen, fällt auf, dass die Gesten Zoom und Rotate fehlen. Das wollen wir hier nachholen. Die Idee ist ein Programm zu schreiben, dass Bilder laden kann, diese dann mittels Gesten verschieben, vergrößern, verkleinern und rotieren kann, und schließlich das Ergebnis auch abspeichern kann.

Dazu definieren wir die folgenden Gesten:

- Ein-Finger-Geste: benutzt der Nutzer einen Finger wird das Bild hin- und her verschoben.
- Zwei-Finger-Geste: mit zwei Fingern kann der Nutzer rein- und rauszoomen.
- Drei-Finger-Geste: erlaubt es dem Nutzer das Bild zu drehen.

In der Activty laden und speichern wir das Bild über das Menu. Das Anzeigen des Bildes und das Verarbeiten der Gesten erfolgt im MultiTouchImageView von Jude Pereira [8]. Der ist ein ImageView und implementiert das OnTouchListener Interface:




```
// Copyright Jude Pereira
private class MultiTouchImageView
    extends ImageView
    implements View.OnTouchListener {

        // we can be in one of these 3 states
        private static final int NONE = 0;
        private static final int DRAG = 1;
        private static final int ZOOM = 2;
        private int mode = NONE;

        // these matrices will be used to move
        // and zoom image
        private Matrix matrix = new Matrix();
        private Matrix savedMatrix = new Matrix();

        // remember some things for zooming
        private PointF start = new PointF();
        private PointF mid = new PointF();
        private float oldDist = 1f;
        private float d = 0f;
        private float newRot = 0f;
        private float[] lastEvent = null;

        public MultiTouchImageView(Context context) {
            super(context);
            this.setBackgroundColor(Color.BLACK);
            this.setScaleType(ImageView.ScaleType.MATRIX);
            this.setOnTouchListener(this);
            this.setImageResource(R.drawable.om);
        }

        public void loadImageFromFile(String absolutePath) {
            Bitmap bitmap = BitmapFactory.decodeFile(absolutePath);
            if (bitmap != null) {
                init();
                this.setImageBitmap(bitmap);
            }
        }

        private void init() {
            mode = NONE;

            matrix = new Matrix();
            savedMatrix = new Matrix();

            start = new PointF();
            mid = new PointF();
            oldDist = 1f;
            d = 0f;
            newRot = 0f;
            lastEvent = null;

            setImageMatrix(matrix);
        }
```

```
public boolean onTouch(View v, MotionEvent event) {
    ImageView view = (ImageView) v;
    switch (event.getAction() & MotionEvent.ACTION_MASK) {
    case MotionEvent.ACTION_DOWN:
        savedMatrix.set(matrix);
        start.set(event.getX(), event.getY());
        mode = DRAG;
        lastEvent = null;
        break;
    case MotionEvent.ACTION_POINTER_DOWN:
        oldDist = spacing(event);
        if (oldDist > 10f) {
            savedMatrix.set(matrix);
            midPoint(mid, event);
            mode = ZOOM;
        }
        lastEvent = new float[4];
        lastEvent[0] = event.getX(0);
        lastEvent[1] = event.getX(1);
        lastEvent[2] = event.getY(0);
        lastEvent[3] = event.getY(1);
        d = rotation(event);
        break;
    case MotionEvent.ACTION_UP:
    case MotionEvent.ACTION_POINTER_UP:
        mode = NONE;
        lastEvent = null;
        break;
    case MotionEvent.ACTION_MOVE:
        if (mode == DRAG) {
            matrix.set(savedMatrix);
            float dx = event.getX() - start.x;
            float dy = event.getY() - start.y;
            matrix.postTranslate(dx, dy);
        } else if (mode == ZOOM) {
            float newDist = spacing(event);
            if (newDist > 10f) {
                matrix.set(savedMatrix);
                float scale = (newDist / oldDist);
                matrix.postScale(scale, scale, mid.x, mid.y);
            }
            if (lastEvent != null &&
                event.getPointerCount() == 3) {
                newRot = rotation(event);
                float r = newRot - d;
                float[] values = new float[9];
                matrix.getValues(values);
                float tx = values[2];
                float ty = values[5];
                float sx = values[0];
                float xc = (view.getWidth() / 2) * sx;
                float yc = (view.getHeight() / 2) * sx;
                matrix.postRotate(r, tx + xc, ty + yc);
            }
        }
        break;
    }

    view.setImageMatrix(matrix);
    return true;
}
```

```
            private float spacing(MotionEvent event) {
                float x = event.getX(0) - event.getX(1);
                float y = event.getY(0) - event.getY(1);
                return FloatMath.sqrt(x * x + y * y);
            }

            private void midPoint(PointF point, MotionEvent event) {
                float x = event.getX(0) + event.getX(1);
                float y = event.getY(0) + event.getY(1);
                point.set(x / 2, y / 2);
            }

            private float rotation(MotionEvent event) {
                double delta_x = (event.getX(0) - event.getX(1));
                double delta_y = (event.getY(0) - event.getY(1));
                double radians = Math.atan2(delta_y, delta_x);
                return (float) Math.toDegrees(radians);
            }
        }
```

Der zentrale Trick hier ist, dass wir mit der Methode *setImageMatrix()* eine 3x3 Transformationsmatrix auf den View anwenden können. Die ermöglicht eine Skalierung, Verschiebung und Drehung des Views. Die restlichen Details entnimmt man besser dem Code direkt.

Research

Auch in diesem Kapitel gibt es einige Themen die man noch durch Eigenrecherche vertiefen kann.

Bézier Kurven

Bézier Kurven spielen eine wichtige Rolle in Computergrafik, Animation und auch in Fonts. Lesen Sie nach welche Arten von Bézier Kurven es gibt und wie sie funktionieren [1].

Universal Computing Machine

Wir haben jetzt schon ein paar mal was von einer "Universal Computing Machine" gehört. Was ist das eigentlich? Warum sollte uns das vielleicht interessieren? Sind Menschen Universal Computing Maschinen? Kann eine Universal Computing Maschine Bewusstsein besitzen? Wenn ja, was wären die Konsequenzen?

Conway's Game of Life

Conways Spiel des Lebens ist uns das erste Mal im ersten Semester über den Weg gelaufen. Interessant ist dazu neben dem Wikipedia Artikel [2] auch ein Interview mit Conway selbst auf Numberphile [3] und schließlich ein bemerkenswertes YouTube-Video [4]. Conways Spiel des Lebens erscheint fast trivial, aber seine philosophischen Implikationen sind unermesslich. Erstens zeigt es, dass die Grundlage des Lebens, die Reproduktion, ein einfaches Regelwerk ist. Und zweitens zeigt es, dass dieselben Grundregeln die Basis einer Universal Computing Machines sind. Was Sie vielleicht in Ihrer Doktorarbeit beweisen könnten, dass das Spiel des Lebens auch Bewusstsein besitzt.

Cellular Automata

Copyright (c) 2005 Richard Ling [7]

Verwandt mit Comways Spiel des Lebens sind Cellular Automata. Eine kleine Einführung finden Sie im Wikipedia Artikel [5]. Insbesondere das Unterkapitel "Elementare zelluläre Automaten" ist wichtig. Stephen Wolfram hat in seinem Buch [6] diese ausführlich untersucht. Bei einem dieser Automaten, nämlich Regel 110, handelt es sich auch, wie das Spiel des Lebens, um eine "Universal Computing Machine".

Wenn Sie nun nach Zeichen von Außerirdischen suchen, schauen Sie nicht weiter als nach rechts: es ist ganz klar, dass diese Kegelschnecke die Regel 110 anzeigt. Offensichtlich wurde ihre DNA von Außerirdischen programmiert, um genau dieses Muster zu erzeugen und uns auf ihre Existenz hinzuweisen. Wenn wir die DNA dieses Tieres genauer analysieren würden, würden wir mit Sicherheit Anweisungen finden, wie wir sie erreichen können. Die Außerirdischen, nicht die Kegelschnecken.

Fragen

1. Erläutern Sie wofür die drei Klassen “Color”, “Paint” und “Canvas” gut sind, evtl. mit Beispiel.
2. Nennen Sie vier der Methoden, die verwendet werden können, um auf einen Canvas zu zeichnen.
3. Welche Methode müssen Sie überschreiben um auf Touch Events reagieren zu können?
4. Nennen Sie eine Geste die sehr einfach mit Androids GestureDetector Klasse umzusetzen ist.
5. Beschreiben Sie bitte in Worten was der folgende Code zeichnet:

```
public class GraphicsActivity extends Activity {
        public void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView( new GraphicsView(this) );
        }
}

class GraphicsView extends View {
        public GraphicsView(Context context) {
                super(context);
        }

        protected void onDraw(Canvas canvas) {
                setBackgroundColor(Color.BLACK);

                Paint cPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
                cPaint.setStyle(Paint.Style.STROKE);
                cPaint.setColor(Color.RED);
                cPaint.setStrokeWidth(2);
                cPaint.setTextSize(24f);

                canvas.drawLine(40, 60, 150, 60, cPaint);
                canvas.drawText("Hi there!", 50, 50, cPaint);
       }
}
```

Referenzen

[1] Bézier curve, https://en.wikipedia.org/wiki/Bézier_curve

[2] Conways Spiel des Lebens, https://de.wikipedia.org/wiki/Conways_Spiel_des_Lebens

[3] Inventing Game of Life - Numberphile, https://www.youtube.com/watch?v=R9Plq-D1gEk

[4] Life in life, https://www.youtube.com/watch?v=xP5-iIeKXE8

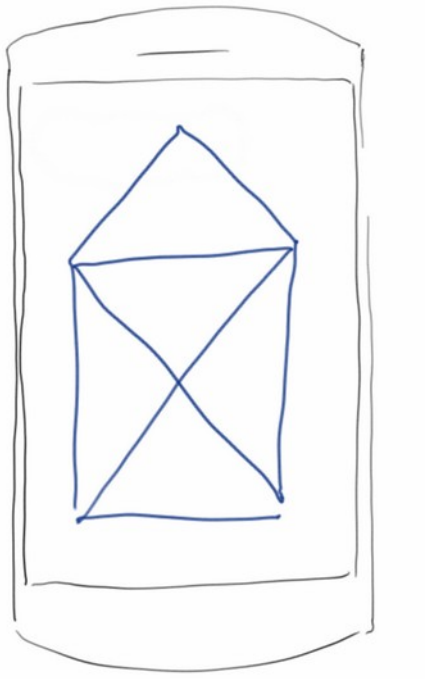
[5] Cellular automaton, https://en.wikipedia.org/wiki/Cellular_automaton

[6] A New Kind of Science, by Stephen Wolfram, http://www.wolframscience.com/nks/

[7] Richard Ling, A Textile cone snail (Conus textile), https://commons.wikimedia.org/wiki/File:Textile_cone.JPG

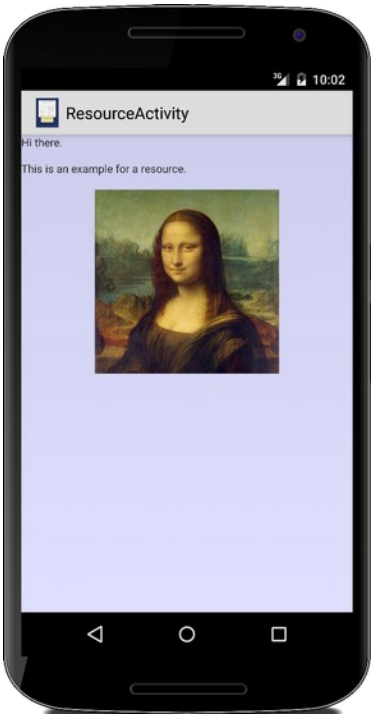
[8] Jude Pereira, Multi Touch in Android – Translate, Scale, and Rotate, https://judepereira.com/blog/multi-touch-in-android-translate-scale-and-rotate/

Persistence



Dieses Kapitel befasst sich mit der Datenpersistenz, d.h. dem Lesen und Schreiben von Daten in einen persistenten Speicher, normalerweise der SD-Karte. Abhängig von unseren Bedürfnissen gibt es verschiedene Möglichkeiten dies zu tun. Das können Dateien sein, die wir nur lesen wollen und die mit der App verpackt ausgeliefert werden sollen. Oder wir wollen kleinere oder größere Datenmengen auf die SD-Karte schreiben. Manchmal wollen wir aber auch mit strukturierten Daten arbeiten, in der Regel Datenbanken. Und schließlich gibt es auf Android noch die sogenannten "Content Provider": dabei handelt es sich um bereits bestehenden Datenbanken, die beispielsweise Kalender- und Kontaktinformationen des Nutzers speichern.

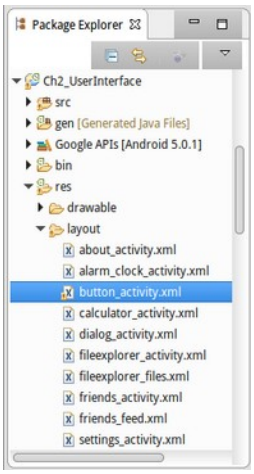
Resources



Wenn wir für unsere App ein paar Bilder oder kleinere Textdateien benötigen, also Dateien die wir nur lesen wollen, dann bietet sich das /res/raw/ Verzeichnis dafür an. Das Verzeichnis kann jede Art von Datei enthalten. Diese Dateien werden mit in der apk-Datei verpackt, d.h. sie werden mit unserer Anwendung ausgeliefert. Das ist zwar praktisch, bedeutet aber auch, dass die Größe unserer apk-Datei mit der Anzahl und Größe der Dateien die wir einbinden auch zunimmt.

Als kleine Anwendung wollen wir aus einer Textdatei names "test.txt" lesen:

```
InputStream is =
getResources().openRawResource(R.raw.text);
BufferedReader r =
    new BufferedReader(new InputStreamReader(is));
StringBuilder total = new StringBuilder();
String line;
while ((line = r.readLine()) != null) {
    total.append(line).append('\n');
}
```

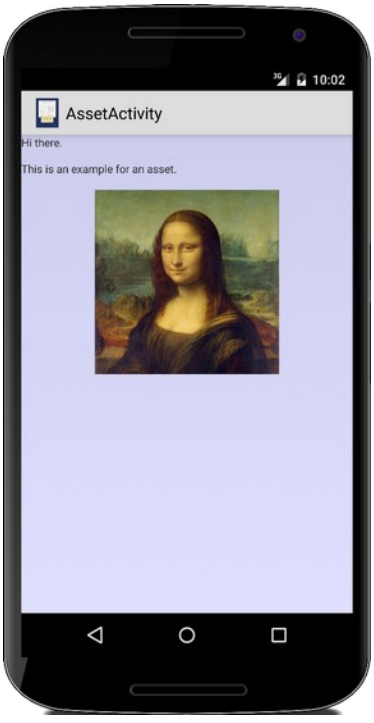


Auch auf binäre Dateien können wir so zugreifen:

```
InputStream is2 =
getResources().openRawResource(R.raw.mona_lisa);
Drawable d = Drawable.createFromStream(is2, null);
ImageView img = new ImageView(this);
img.setImageDrawable(d);
img.getLayoutParams().height = 800;
```

Ein kleiner Nachteil bei der Verwendung von Ressourcen ist, dass Dateien im Verzeichnis /raw/ keine Großbuchstaben oder Leerzeichen enthalten dürfen. Ein Vorteil ist hingegen, dass die Namen der Ressourcen bei der Kompilierung überprüft werden, so dass Tippfehler ausgeschlossen sind.

Assets

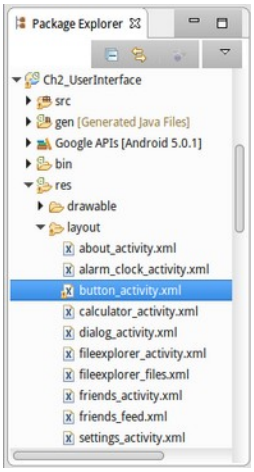


Assets sind den Ressourcen sehr ähnlich, sie werden auch zum Lesen von schreibgeschützten Daten verwendet, die sich aber in dem Verzeichnis /assets/ befinden. Sie werden auch mit der apk-Datei ausgeliefert. Das Asset Verzeichnis ist mehr wie ein Dateisystem und erlaubt eine Verzeichnisstruktur, was hilfreich ist, wenn wir viele Dateien haben. Außerdem folgen die Dateinamen den üblichen Namenskonventionen.

Der Zugriff auf Dateien im Asset-Verzeichnis ist fast gleich wie bei den Ressourcen:

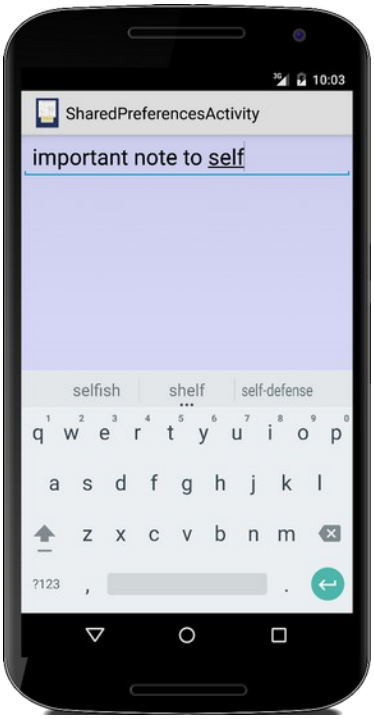
```
InputStream is = getAssets().open("text.txt");
...
InputStream is2 =
getAssets().open("Mona_Lisa.jpg");
...
```

Der feine Unterschied ist, dass wir Strings statt Konstanten verwenden, um auf die Dateien zuzugreifen. Das bedeutet zwar, dass der Compiler nicht mehr auf Tippfehler prüfen kann, aber es gibt uns mehr Flexibilität.



Der Vorteil von Ressourcen und Assets ist, dass sie zusammen mit dem apk ausgeliefert und verteilt werden. Das ist aber auch ihr größter Nachteil. Und wir können nur lesend auf sie zugreifen.

Shared Preferences



Häufig müssen wir in unseren Apps einfach nur ein paar Schlüssel-Wert Paare speichern. Z.B. Einstellungen in unserer App oder einen Highscore in einem Spiel. Dafür kann man die Klasse SharedPreferences verwenden. Mit ihr kann man primitive Datentypen wie booleans, floats, ints, longs , aber auch Strings speichern.

Im Prinzip sind die SharedPreferences nichts anderes wie eine HashMap in der wir ja auch nur Key-Value Paare speichern. Als kleines Beispiel lesen wir einen String zu Beginn unserer Anwendung in der *onCreate()* Methode. Bevor unsere Anwendung beendet wird, also in der *onStop()* Methode speichern wir dann den evtl. neuen Wert wieder ab.

```
@Override
protected void onCreate(Bundle savedInstanceState)
{
    ...
    SharedPreferences prefs =
        getPreferences(MODE_PRIVATE);
    String txt = prefs.getString("KEY", "default");
    ...
}

@Override
protected void onStop() {
    ...
    SharedPreferences prefs = getPreferences(MODE_PRIVATE);
    SharedPreferences.Editor editor = prefs.edit();
    editor.putString("KEY", et.getText().toString());
    editor.commit();
    ...
}
```

Natürlich sollten SharedPreferences nur für kleinere Datenmengen verwendet werden.

Man kann auch im *Bundle* kleine Datenmengen speichern, gesehen haben wir das im Paint Beispiel des zweiten Kapitels.

Internal Storage

In Android hat jede Anwendung ein spezielles privates Verzeichnis (/data/data/packagename), das bei der Installation erstellt wird. Dieses befindet sich auf der internen SD Karte. Man kann da alles mögliche dauerhaft speichern. Als kleines Beispiel wollen wir die Zahl 42 in die Datei "test.data" schreiben und lesen:

```
FileOutputStream fos =
    openFileOutput("test.data", MODE_PRIVATE);
fos.write( 42 );
fos.close();
```

```
FileInputStream fis = openFileInput("test.data");
int i = fis.read();
et.setText(""+i);
fis.close();
```

Für den Zugriff auf das private Verzeichnis stehen zwei weitere interessante Methoden zur Verfügung. Mit *fileList()* können wir alle Dateien und Unterverzeichnisse auflisten und mit *deleteFile()* können wir Dateien löschen. Da der interne Speicher auf den meisten Geräten etwas eingeschränkt ist, sollten wir keine sehr großen Dateien in den internen Speicher ablegen.

External Storage

Wenn wir größere Datenmengen haben, oder wenn unsere Daten nicht privat sind, also z.B. wenn wir unsere Daten mit anderen Anwendungen teilen möchten, dann können wir die externe SD-Karte verwenden. Am Code ändert sich fast nichts:

```
FileOutputStream fos = new
FileOutputStream("/sdcard/test.data");
```

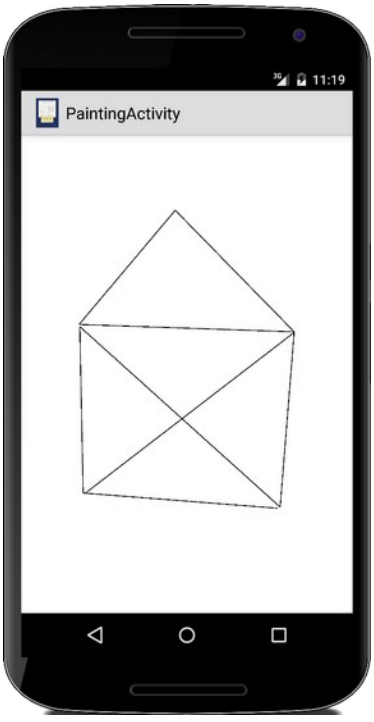
Analog für den InputStream.

Ein Unterschied besteht allerdings: unsere Anwendung muss um Erlaubnis fragen, damit sie auf die externe SD-Karte zugreifen kann. In der AndroidManifest.xml Datei müssen wir folgende Zeile einfügen:

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

Normalerweise verwendet man das private Verzeichnis für Daten, die kleiner als ein oder zwei Megabyte sind. Für größere Datenmengen sollte man die externe SD-Karte verwenden.

Serialization



Im zweiten Kapitel haben wir schon mal mit den Thema *Serialization* zu tun gehabt. Kurz gesagt geht es darum Objekte zu speichern, z.B. auf der Festplatte. Das Gegenstück dazu ist *Deserialisieren*, also ein Objekt wieder von der Festplatte zu laden. Das geht überraschend einfach mit den ObjectOutput- und ObjectInputStream Klassen.

Als Beispiel wollen wir unsere Kunstwerke (PaintingActivity) aus Kapitel drei serialisieren. Die Kunstwerke sind als Punkte in einer Liste gespeichert:

```
public class PaintingActivity extends Activity {

    private List<Point> points =
        new ArrayList<Point>();

    ...
}
```

also müssen wir lediglich die Liste speichern, und das geht überraschend einfach:

```
FileOutputStream fos = openFileOutput(FILE_NAME, MODE_PRIVATE);
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(points);
oos.close();
fos.close();
```

Wir benötigen einen OutputStream, z.B. einen FileOutputStream. Diesen übergeben wir an einen ObjectOutputStream als Parameter. Das Speichern selbst geht dann über den Aufruf der Methode *writeObject()*. Wenn wir diese Zeilen in die onPause() Methode integrieren, werden unsere Kunstwerke jedes mal gespeichert wenn wir unsere Activity beenden.

Der umgekehrte Vorgang, also das Deserialisieren, geht komplett analog:

```
FileInputStream fis = openFileInput(FILE_NAME);
ObjectInputStream ois = new ObjectInputStream(fis);
Object obj = ois.readObject();
points = (List<Point>) obj;
ois.close();
fis.close();
```

Hier benötigen wir also einen InputStream den wir an den ObjectInputStream übergeben, und per *readObject()* lesen wir dann unser Objekt wieder ein. Da der ObjectInputStream nicht wissen kann um was für ein Objekt es sich handelt, müssen wir ihm das noch sagen, deswegen der Cast. Wenn wir diese Zeilen in die onResume() Methode integrieren, werden unsere Kunstwerke jedes mal geladen wenn die Activity neu startet.

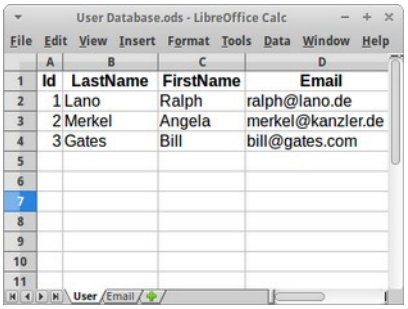
Sind alle Klassen serialisierbar? Interessanterweise nein. Z.B. die Android Klasse "android.graphics.Point" die wir in der ursprünglichen Version der PaintingActivity verwendet haben ist nicht serialisierbar. Deswegen müssen wir unsere eigene Point Klasse schreiben. Das hört sich komplizierter an als es ist. Unsere Klasse muss lediglich das *Serializable* Interface implementiert:

```
class Point implements Serializable {
    public int x;
    public int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Komischerweise muss man beim Serializable Interface gar keine Methode überschreiben. Ist ein bisschen wie Magie (Reflection)!

Eine Anmerkung noch: innere Klassen kann man nicht serialisieren, bzw. man kann sie nur zusammen mit ihren äußeren Klassen serialisieren.

Database



Obwohl viele Leute höllischen Respekt vor Datenbanken haben, sind die eigentlich gar nicht so schwer. Für uns ist eine Datenbank erst einmal ein paar Excel Tabellen. Als Beispiel beginnen wir mit einer Tabelle für Nutzer, wir nennen sie mal "User". Ein User hat einen Nachnamen, einen Vornamen und eine Email Adresse. In der Tabelle werden daraus Spalten (Columns). Die einzelnen Einträge für die Leute sind dann die Reihen (Rows). Bei Datenbank Tabellen fügt man dann meist noch eine Spalte für die "Id" ein: eine Id ist einfach ein Zähler, der bei 1 anfängt und für jeden neuen Eintrag um eins erhöht wird. Damit hat jeder Eintrag eine eigene Id und es kann nie doppelte Ids geben. Was aber passieren könnte, dass es zwei Leute mit dem gleichen Namen gibt. Gibt's ja auch in echt.

SQL

Die "Structured Query Language", kurz SQL [1], ist die Sprache die die meisten Datenbanken sprechen. Es ist eine etwas ältere Sprache (so wie Altgriechisch, sieht man daran, dass alles groß geschrieben ist und die Befehle immer weniger als acht Buchstaben haben), aber sie funktioniert immer noch recht gut.

Als erstes ist es immer eine gute Idee evtl. existierende Tabellen zu löschen. Das geht mit

```
DROP TABLE Users;
```

ganz einfach. Sollte man natürlich nur machen, wenn da keine wichtigen Daten drin waren, die sind nämlich sonst futsch.

Danach legen wir eine neue Tabelle an. Hierfür gibt es das "CREATE TABLE" Kommando:

```
CREATE TABLE Users (
    Id INTEGER not null,
    LastName VARCHAR(255) not null,
    FirstName VARCHAR(255),
    Email VARCHAR(255),
    PRIMARY KEY (Id)
);
```

Wir sagen also, dass die Tabelle "Users" heißen soll, dass es eine Spalte Id geben soll, die eine Ganzzahl sein soll und die immer einen Wert haben muss (not null). Dann deklarieren wir noch die anderen Spalten. Dabei heißt "VARCHAR(255)" soviel wie ein String der Länge 255 auf Altgriechisch. Am Ende sagen wir noch, dass "Id" der Primary Key sein soll.

Nachdem wir die Tabelle angelegt haben, wollen wir auch ein paar Daten einfügen, und das geht folgendermaßen:

```
INSERT INTO Users (Id, LastName, FirstName, Email)
    VALUES (1,'Lano','Ralph','ralph@lano.de');
```

Ganz wichtig sind die Apostrophen, auch *Single Quotes* genannt. Das sind also die geraden Striche die von oben nach unten gehen, und sich weder leicht nach links, noch leicht nach rechts lehnen. Hört sich lustig an, ist aber wichtig: besonders beim Kopieren aus PDF Dateien passiert es nicht selten, dass die Apostrophen sich leicht zu neigen beginnen.

Schließlich wollen wir mal schauen was inzwischen in unserer Tabelle alles drin steht, das geht mit:

```
SELECT * FROM Users;
```

Es gibt dann noch ein DELETE und ein UPDATE Kommando, aber im Prinzip war's das schon.

SQLite

Teil von Android ist eine Datenbank namens SQLite [2]. Sie ist relativ einfach zu bedienen: Für jede Tabelle, die wir in unserer Datenbank haben wollen, erstellen wir eine Klasse, z.B. die *FriendsTable*:

```
import static android.provider.BaseColumns._ID;

public class FriendsTable extends SQLiteOpenHelper {
        public static final String TABLE_NAME = "friends";
        public static final String LAST_NAME = "lastName";
        public static final String FIRST_NAME = "firstName";
        public static final String EMAIL = "email";

        public FriendsTable(Context context, String name,
                            CursorFactory factory, int version) {
                super(context, name, factory, version);
        }
        ...
}
```

Die Klasse erweitert die *SQLiteOpenHelper* Klasse, was uns den Zugriff auf die SQLite Datenbank erleichtert, d.h. um die Low-Level Details müssen wir uns nicht kümmern. Es macht Sinn, Tabellen- und Spaltennamen als Konstanten zu deklarieren, das vermeidet Typos.

Was wir noch tun müssen, ist die beiden Methoden *onCreate()* und *onUpdate()* zu überschreiben. Die erste wird benötigt, da wir beim ersten Start unseres Programms alle Tabellen in der Datenbank anlegen müssen. Die zweite wird benötigt, wenn wir unsere Tabellen ändern, d.h. eine neue Spalte hinzufügen, eine entfernen oder ihre Eigenschaften ändern wollen. Normalerweise will man bei einem Update nicht alle vorhandenen Daten löschen, aber in unserem einfachen Fall ist es genau das, was wir tun:

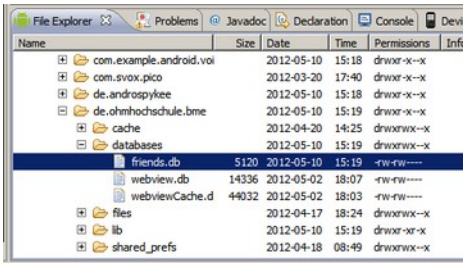
```
@Override
public void onCreate(SQLiteDatabase db) {
        db.execSQL("CREATE TABLE " + TABLE_NAME + " ("
                + _ID + " INTEGER PRIMARY KEY AUTOINCREMENT, "
                + LAST_NAME + " TEXT NOT NULL,"
                + FIRST_NAME + " TEXT NOT NULL,"
                + EMAIL + " TEXT NOT NULL);");
}

@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion,
                      int newVersion) {
        db.execSQL("DROP TABLE IF EXISTS " + TABLE_NAME);
        onCreate(db);
}
```

Wenn wir genau hinsehen, erkennen wir unsere SQL Kommandos CREATE und DROP.

Creating the Database

Als erstes müssen wir eine Datenbank erstellen. Das machen wir am besten in der onCreate() Methode unserer Activity:

```
public class DatabaseActivity extends Activity {

    private static final String DATABASE_NAME =
        "friends.db";
    private static final int DATABASE_VERSION = 6;

    private FriendsTable friends;

    @Override
    public void onCreate(Bundle savedState) {

        friends = new FriendsTable(this,
            DATABASE_NAME, null, DATABASE_VERSION);
        ...
    }
}
```



Wenn wir eine neue FriendsTable erstellen, prüft das System, ob die Datenbank "friends" bereits existiert. Wenn das der Fall ist, dann wird es diese einfach öffnen. Ist das nicht der Fall, wird eine neue Datei namens "friends.db" im Verzeichnis "/data/data/package_name/databases" erstellt. Die gesamte Datenbank wird in dieser einen Datei gespeichert, was natürlich bedeutet, dass die Größe unserer Datenbank durch die Größe die maximale Dateigröße begrenzt ist. Man kann sich die Datenbankdatei in der FileExplorer Ansicht ansehen.

Noch etwas: Jedes Mal, wenn wir eine unserer Tabellen ändern, müssen wir eine neue Datenbankversion anlegen, d.h. die Konstante 'DATABASE_VERSION' sollte um Eins erhöht werden.

Inserting Data

Sobald die Datenbank fertiggestellt ist, können wir Daten einfügen. Da unsere FriendsTable Klasse die SQLiteOpenHelper Klasse erweitert, gibt es die *getWriteableDatabase()* Methode. Diese gibt uns ein SQLiteDatabase Objekt zurück, mit dem wir dann Daten in unsere Tabelle einfügen können:

```
private void addFriend(String lastName, String firstName, String email){
        SQLiteDatabase db = friends.getWritableDatabase();
        ContentValues values = new ContentValues();
        values.put(FriendsTable.LAST_NAME, lastName);
        values.put(FriendsTable.FIRST_NAME, firstName);
        values.put(FriendsTable.EMAIL, email);
        db.insert(FriendsTable.TABLE_NAME, null, values);
}
```

Falls etwas schief geht, wird eine Exception ausgelöst.

Reading Data

Schließlich müssen wir in der Lage sein, aus unserer Datenbanktabelle zu lesen. Auch hier verwenden wir das SQLiteDatabase Objekt, allerdings in der read-only Version. Die Syntax bezieht sich auf die Syntax der SQL SELECT-Anweisung:

```
private Cursor getFriends() {
        SQLiteDatabase db = friends.getReadableDatabase();
        String[] FROM = { _ID, FriendsTable.LAST_NAME,
                          FriendsTable.FIRST_NAME, FriendsTable.EMAIL, };
        Cursor cursor = db.query(FriendsTable.TABLE_NAME, FROM,
                        null, null, null, null,
                        FriendsTable.LAST_NAME + " DESC");
        startManagingCursor(cursor);
        return cursor;
}
```

Die Methode *query()* hat dabei Platz für folgende Parameter: den Tabellennamen, die Selection, die Selection Argumente, GroupBy, Having und OrderBy Argumente.

Die Methode *getFriends()* gibt uns einen Cursor zurück, der das Ergebnis der Abfrage enthält. Der Cursor ist ein Iterator, mit dem wir über die zurückgegebenen Werte iterieren können:

```
private void showFriends(Cursor cursor) {
        StringBuffer sb = new StringBuffer();
        while (cursor.moveToNext()) {
                long id = cursor.getLong(0);
                sb.append(id).append(": ");
                sb.append(cursor.getString(1)).append(", ");
                sb.append(cursor.getString(2)).append(", ");
                sb.append(cursor.getString(3)).append("\n");
        }
        textView1.setText( sb.toString() );
}
```

Das war's.

Updating and Deleting

Zwei weitere Operationen, die wir ausführen können, sind das Aktualisieren und Löschen vorhandener Einträge. Die folgenden zwei Methoden zeigen, wie man das macht:

```
private void deleteFriend(int rowID) {
        SQLiteDatabase db = friends.getWritableDatabase();
        String where = _ID + "=" + rowID;
        db.delete(FriendsTable.TABLE_NAME, where, null);
}

private void updateFriend(int rowID, String lastName,
                          String firstName, String email) {
        SQLiteDatabase db = friends.getWritableDatabase();
        ContentValues values = new ContentValues();
        values.put(FriendsTable.LAST_NAME, lastName);
        values.put(FriendsTable.FIRST_NAME, firstName);
        values.put(FriendsTable.EMAIL, email);
        String where = _ID + "=" + rowID;
        db.update(FriendsTable.TABLE_NAME, values, where, null);
}
```

Raw Queries

Falls wir die Sprache SQL beherrschen, können wir auch reines SQL mit der Methode *rawQuery()* ausführen:

```
private long getNumberOfFriendsWithRawQuery() {
    SQLiteDatabase db = friends.getReadableDatabase();
    String[] FROM = { _ID, FriendsTable.LAST_NAME,
                      FriendsTable.FIRST_NAME, FriendsTable.EMAIL, };
    Cursor cursor = db.rawQuery(
        String.format(
            "select count(*) from %s", FriendsTable.TABLE_NAME),
        null);
    cursor.moveToFirst();
    long count = cursor.getLong(0);
    cursor.close();
    return count;
}
```

Dies ist nicht unbedingt empfehlenswert, aber eine Option.

Content Provider

Kommen wir kurz zu den Content-Providern: Bei den Content-Providern handelt es sich um bereits vorgefertigte Datenbanken die mit Android kommen. Man könnte natürlich auch seinen eigenen Content-Provider schreiben, aber normalerweise verwendet man die bereits existierenden, als da sind:

- **ContactsContract:** erlaubt Zugriff auf die Kontakte.
- **CalendarContract:** erlaubt Zugriff auf den Kalender.
- **MediaStore:** listet alle Mediendateien auf dem Gerät.
- **Browser:** erlaubt Zugriff auf Lesezeichen, Browserverlauf und Suchanfragen.
- **CallLog:** enthält Informationen über die Anrufhistorie.
- **UserDictionary:** erlaubt Zugriff auf benutzerdefinierte Wörter.
- **Settings:** erlaubt Zugriff auf die Geräteeinstellungen.

Wir werden uns die ersten drei in den Projekten näher ansehen.

ORM

Wir sind inzwischen im 21. Jahrhundert und leben in einer objekt-orientierten Welt. Auf eine Datenbank mittels SQL zuzugreifen ist vielleicht aus historischen Gründen noch interessant (oder falls Sie pro Zeile Code bezahlt werden), aber normalerweise würde man versuchen, ein Objekt-Relational Framework (ORM) zu verwenden. Das erledigt für uns den größten Teil der Fußarbeit und wir können uns auf die Kopfarbeit konzentrieren.

Als Beispiel werfen wir kurz einen Blick auf das Sugar ORM Framework [3]. Es ist ziemlich einfach einzurichten und es ist Open Source. Um das Sugar ORM Framework nutzen zu können, müssen wir zwei Bibliotheken, sugar-1.4.jar und guava-19.0.jar, zu unserem Projekt hinzufügen. Als zugrundeliegende Datenbank wird SQLite verwendet.

Configuration

Bevor wir Sugar ORM verwenden können, müssen wir noch die folgenden Zeilen zur AndroidManifest Datei hinzufügen:

```
...
<activity
    android:name=
        "variationenzumthema_ch4.ORMActivity"
    android:label="ORMActivity" >
</activity>
<meta-data
    android:name="DATABASE"
    android:value="orm_sugar_example.db" />
<meta-data
    android:name="VERSION"
    android:value="2" />
<meta-data
    android:name="QUERY_LOG"
    android:value="true" />
<meta-data
    android:name="DOMAIN_PACKAGE_NAME"
    android:value="variationenzumthema_ch4" />
...
```

Das Paket sollte dasjenige sein, in dem sich unsere ORMActivity befindet.

POJO

Zuerst müssen wir die Klasse deklarieren, die wir in der Datenbank speichern wollen, hier die Klasse *Book*:

```
public class Book extends SugarRecord {
    @Unique
    String isbn;
    String title;
    String author;

    public Book() {
    }

    public Book(String isbn, String title, String author) {
        this.isbn = isbn;
        this.title = title;
        this.author = author;
    }
}
```

Sugar ORM erstellt daraus eine Datenbanktabelle namens BOOK mit einer Spalte für jede Instanzvariable. Sugar verwendet Reflection, deswegen müssen wir einen Standardkonstruktor zur Verfügung stellen, auch wenn er nichts tut.

Writing and Reading

Schreiben und Lesen mit Sugar ORM ist sehr einfach: wir müssen lediglich unseren Code mit den Methoden *SugarContext.init()* und *SugarContext.terminate()* umgeben:

```
SugarContext.init(this);

// create new book:
Book book1 = new Book("1530651840", "Java", "Ralph Lano");
book1.save();

// load entity:
Book book2 = Book.findById(Book.class, 1);

SugarContext.terminate();
```

Der Rest geht wie folgt: Wir sagen unserem Buch einfach, dass es sich mit *save()* speichern soll, und wenn wir nach einem Buch suchen, verwenden wir die *findById()* Methode.

Sugar ORM ermöglicht auch komplexere Abfragen:

```
List<Book> books = Book.findWithQuery(
                    Book.class,
                    "SELECT * FROM Book WHERE isbn LIKE ?",
                    "153%");
```

Wenn dies an Stored Procedures erinnert, dann ist es nicht verwunderlich, denn das ist es auch.

Wir können auch Bücher löschen,

```
Book book2 = Book.findById(Book.class, 1);
book2.delete();
```

und updaten,

```
Book bookOne = new Book("1530651840", "Java", "Ralph P. Lano");
bookOne.update();
```

Wenn wir alle Bücher brauchen, dann gibt es dafür die *findAll()* Methode:

```
Iterator<Book> booksIt = Book.findAll(Book.class);
```

Auch das Einfügen von mehreren Büchern in einem Rutsch ist möglich:

```
List<Book> books = new ArrayList<>();
books.add(new Book("1530651840", "Java", "Ralph P. Lano"));
books.add(new Book("1545467463", "Algorithmen", "Ralph Lano"));
books.add(new Book("1537765469", "Internet", "Ralph P. Lano"));
books.add(new Book("1724526138", "Android", "Ralph P. Lano"));
SugarRecord.saveInTx(books);
```

Sugar ORM ist nicht ganz so leistungsfähig wie Hibernate oder JDO, aber es unterstützt One-To-One und One-To-Many Beziehungen. Also für die meisten unserer täglichen Bedürfnisse, ist es gut genug. Wenn wir etwas Ausgefalleneres brauchen, dann gibt es z.B. Room von Google [4] oder auch greenDAO [5] ist einen Blick wert.

Online Storage

Eine weiterer Ort unsere Daten zu speichern, ist Online. Natürlich kann man sich da selbst was überlegen, z.B. mit einem RESTful Webservice. Aber es gibt auch andere interessante Lösungen. Da ist z.B. Firebase von Google [6], das im Grunde eine Datenbank in der Cloud ist. Ein kleines Problem mit den Online-Speichern ist, was passiert, wenn wir offline sind? Nun, die Realm-Datenbank [7] versucht dieses Problem zu lösen.

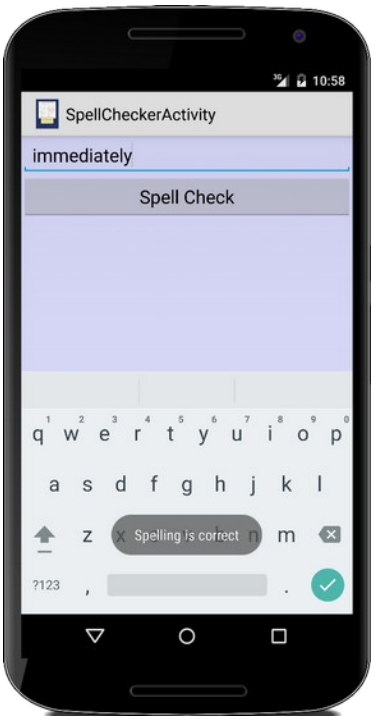
Review

In diesem Kapitel haben wir gelernt wie wir Daten permanent speichern können. Dabei haben wir gesehen wie man mit Ressourcen und Assets Dateien als Teil der App ausliefern kann. Wir haben SharedPreferences kennengelernt und auch gesehen wie man auf die interne und externe SD Karte zugreifen kann. Schließlich haben wir gesehen wie man mit der eingebauten Datenbank strukturierte Daten bearbeiten kann. Wir haben Content-Provider kurz angesprochen und auch mit ORM haben wir uns kurz beschäftigt.

Projekte

In den Projekten wollen wir zunächst Ressourcen und Assets vertiefen, dazu erinnern wir uns an ein paar Beispielen aus dem zweiten Buch, wie z.B. dem SpellChecker oder dem Wörterbuch. Als anspruchsvolleres Beispiel sehen wir uns auch noch einmal den UBahn-Navigator an. Ein einfacher Texteditor demonstriert das Lesen und Schreiben auf die externe SD Karte. Und danach folgen noch ein paar Beispiele zu ContentProvidern wo wir auf den Kalender, die Kontakte und die MusikProvider zugreifen.

SpellChecker



Die UI des SpellChecker besteht aus einem EditText und einem Button. Nachdem der Nutzer ein Wort eingegeben hat und auf den "Spell Check" Knopf gedrückt hat, checken wir ob das eingegebene Wort in unserem Wörterbuch zu finden ist. Über einem Toast geben wir dem Nutzer dann Rückmeldung.

Wenn wir noch mal kurz im zweiten Semester nachsehen, dort haben wir dafür ein HashSet verwendet. Das HashSet befüllen wir mit allen Wörter der englischen Sprache,

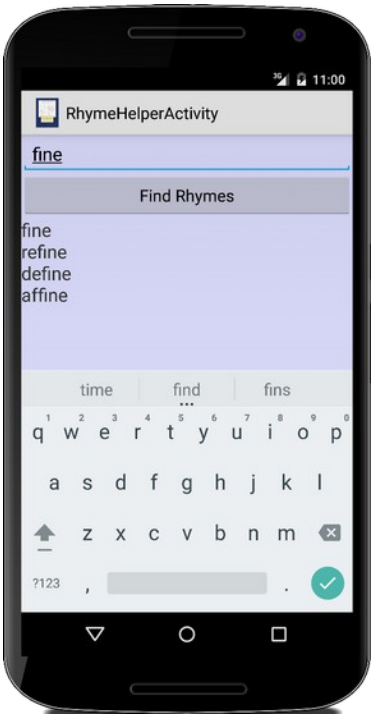
```
Set<String> words =
    buildIndexFromFile("dictionary_en_de.txt");
```

und verwenden dann die *contains()* Methode um festzustellen ob das Wort richtig geschrieben wurde:

```
String word = readLine("Enter word to check: ");
if (words.contains(word.toLowerCase())) {
    println("Spelling is correct.");
} else {
    println("Spelling is NOT correct.");
}
```

Das Wörterbuch könnten wir entweder als Resource oder als Asset speichern.

RhymeHelper



Ganz analog zum SpellChecker können wir auch unsere Reim-Anwendung aus dem zweiten Semester auf Android portieren. Für die UI benötigen wir einen EditText, einen Button und einen TextView. Die Logik können wir eins-zu-eins aus dem zweiten Semester übernehmen.

Die Datenstruktur die wir damals verwendet haben war der *Trie*. Wenn wir also Worte suchen die sich auf "cool" reimen, dann suchen wir nach allen Worten die auf "ool" enden. Der Trick ist die Worte falsch herum in dem Trie zu speichern.

Wir instanziieren den *Trie*:

```
private SimpleTrie trie = new SimpleTrie();
```

und in den Trie schreiben wir einfach unser gesamtes Wörterbuch:

```
loadLexiconFromFile("dictionary_en_de.txt");
```

Beim Einfügen in den Trie,

```
trie.add( reverseString(en.toLowerCase()) );
```

achten wir aber darauf, dass wir alle Worter falsch herum einfügen. Wenn wir dann nach Reimen suchen, ist das ganz einfach:

```
String word = readLine("Enter word to rhyme: ");
for (String s : trie.nodesWithPrefix( reverseString(word) )) {
    println( reverseString(s) );
}
```

Ganz perfekt ist die Anwendung nicht, denn sie findet nur exakte Reime. Besser wäre es wahrscheinlich einen abgewandelten SoundEx oder Metaphone Algorithmus zu verwenden.

Languages

Wir wollen unser Mehr-Sprachen-Übersetzungsprogramm aus dem zweiten Semester aufs Handy portieren. Die UI besteht wieder aus einem EditText und einem Button. Für die Anzeige der Übersetzungen bietet sich die Möglichkeit einen ListView zu verwenden, um die verschiedenen Sprachen gleichzeitig anzuzeigen.

Wir verwenden eine Map die einen String als Key hat (das englische Wort) und als Value eine List von Strings (die Übersetzungen). Das Wörterbuch können wir entweder als Resource oder als Asset speichern.

UniversalTranslator

Die Idee hinter dem UniversalTranslator ist es, einen Satz aus dem Englischen in eine (fast) beliebige andere Sprache zu übersetzen. Allerdings nur Wort für Wort. Wir nehmen den Logik Code vom obigen Beispiel. Auch die UI ist ganz ähnlich: einen EditText zum Eingeben, einen Spinner mit dem wir die Sprache auswählen und einen TextView um die Übersetzung anzuzeigen. Das Wörterbuch können wir entweder als Resource oder als Asset speichern.

Subway

Die Subway Activity besteht aus zwei Spinnern, einem Button und einem TextView. Der Nutzer wählt den Abfahrts- und Zielbahnhof über die beiden Spinner aus. Über den Button wird dann die Suche nach der kürzesten Strecke ausgelöst. Das Ergebnis wird dann im TextView angezeigt. Die Datei mit den Verbindungsdaten können wir entweder als Resource oder als Asset speichern.

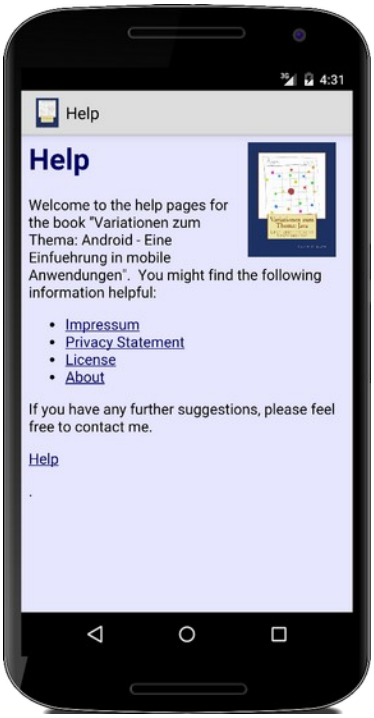
Bzgl der Logik bedienen wir uns wieder dem Beispiel aus dem zweiten Semester.

Synonyms

Beim Synonymbrowser geht es darum ausgehend von einem Wort dessen Synonyme in einer Liste anzuzeigen. Klickt der Nutzer dann auf eines dieser Synonyme, wird dieses Wort als Ausgangswort genommen. Auf diese Art kann man ganz interessante Reisen durch den Synonymgraphen machen.

Die UI ist identisch zu der in der Languages Activity: ein EditText, ein Button und ein ListView. Die Logik wie man durch den Synonymgraphen navigiert kommt wieder aus dem zweiten Semester. Die Datei mit den Synonympaaren können wir entweder als Resource oder als Asset speichern.

HelpPages



Jede App benötigt ein Impressum, evtl. sogar eine Datenschutzerklärung, u.U. vielleicht auch noch Hilfeseiten. Hier gibt es verschiedene Möglichkeiten das umzusetzen. Man könnte eine eigene UI schreiben. Der Nachteil ist, dass das bei Änderungen nicht einfach zu pflegen ist. Man könnte die Seiten als HTML Seiten im Internet hosten. Der Nachteil hier ist, dass die Seiten offline dann nicht zur Verfügung stehen.

Eine dritte Möglichkeit ist, das WebView Widget zu verwenden. Der Vorteil dieser Variante ist, dass wir für unseren Website und für unsere App die gleichen Inhalte pflegen können. Das WebView Widget, das auch Androids hauseigenen Browser verwendet, kann HTML und CSS darstellen. Die Webseiten selbst, inklusive Stylesheets und Bilder, laden wir aus dem Resource Verzeichnis (/res/raw/).

Damit auch Links funktionieren, müssen wir unsere HTML Seiten ein klein wenig modifizieren. Z.B. die index.html, sieht wie folgt aus:

```
<html>
  ...
  <link rel="stylesheet" type="text/css"
   href="style.css">
  ...
  <img alt="" src="img0.jpg"
   style="width: 96px; height: 125px" />
  ...
  <a href="int:0">Impressum</a>
  <a href="int:1">Privacy Statement</a>
  <a href="int:2">License</a>
  ...
</html>
```

Also Links zu Stylesheets und Bildern sind einfach relative Links. Allerdings Links zu anderen lokalen Seiten, ändern wir in "int:0" um. Dabei sind die Seiten einfach mit 0, 1, 2, ... durchnummeriert, und "int" steht für intern.

In der onCreate() Methode unserer *HelpPagesActivity* initialisieren wir den WebView:

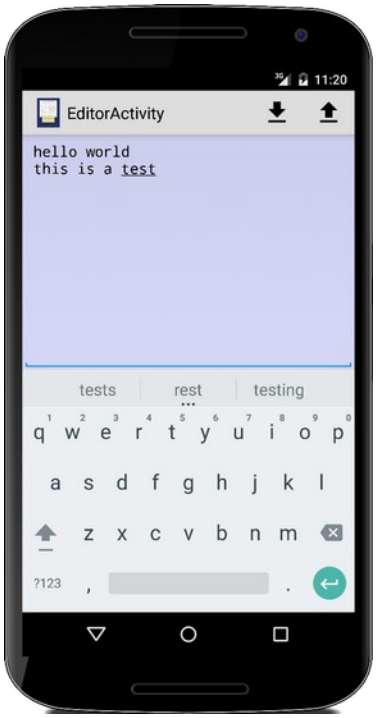
```
public class HelpPagesActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        this.setTitle("Help");
        WebView webView = new WebView(this);
        webView.setWebViewClient(new HelpPagesWebViewClient());
        String html = readFile("");
        webView.loadDataWithBaseURL("file:///android_res/raw/",
                                    html, "text/html", "UTF-8", null);
        webView.getSettings().setBuiltInZoomControls(true);
        webView.getSettings().setSupportZoom(true);
        setContentView(webView);
    }
```

Die *readFile()* Methode liest eine gewünschte Datei aus dem Resource Verzeichnis:

```
    private String readFile(String pageNr) {
        Field f = null;
        ...
        f = R.raw.class.getField("index" + pageNr);

        StringBuffer sb = new StringBuffer();
        InputStream htmlStream =
            getResources().openRawResource(f.getInt(null));
        BufferedReader is = new BufferedReader(
            new InputStreamReader(htmlStream, "UTF8"));
        String line;
        while ((line = is.readLine()) != null) {
            sb.append(line);
        }
        is.close();
        ...
        return sb.toString();
    }
```

Interessant ist noch die HelpPagesWebViewClient Klasse: sie sorgt dafür, dass die internen Links funktionieren, indem sie die Methode *shouldOverrideUrlLoading()* überschreibt:

```
    private class HelpPagesWebViewClient extends WebViewClient {
        @Override
        public boolean shouldOverrideUrlLoading(WebView view,
                                                String url) {
            if (url.startsWith("int:")) {
                String html = readFile(url.substring(4));
                view.loadDataWithBaseURL("file:///android_res/raw/",
                    html, "text/html", "UTF-8", null);
                view.getSettings().setBuiltInZoomControls(true);
                view.getSettings().setSupportZoom(true);
            }
            return true;
        }
    }
}
```



Editor

Wie schwer ist es denn einen kleine Texteditor zu schreiben? Im Prinzip besteht doch ein Texteditor nur aus einem EditText und zwei Knöpfen zum Öffnen und Speichern. Beim EditText müssen wir ein paar Attribute setzen, damit er das macht was er soll:

```
<EditText
    android:id="@+id/editText"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fontFamily="monospace"
    android:gravity="top|left"
    android:inputType="textMultiLine"
    android:scrollHorizontally="false"
    android:scrollbars="vertical"
    android:text=""
    android:textSize="16sp" />
```

Die beiden Knöpfe kann man am einfachsten über ein IconMenu wie wir es im zweiten Kapitel gemacht haben implementieren.

Um Dateien zu suchen verwenden wir am besten die Klasse *FileChooser* die dessen Autor Roger Keays freundlicherweise unter die Public Domain Lizenz gestellt hat [8]. Diese müssen wir in der onCreate() initialisieren:

```
public class EditorActivity extends Activity
    implements FileSelectedListener {

    private EditText et;
    private FileChooser fc;
    private File currentFile;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.editor_activity);

        et = (EditText) findViewById(R.id.editText);

        currentFile =
            android.os.Environment.getExternalStorageDirectory();
        fc = new FileChooser(this, currentFile.getAbsolutePath());
        fc.setFileListener(this);
    }
    ...

}
```

Interessant ist auch der kleine Trick, wie wir das Android Betriebssystem fragen können, wo denn genau bitte der External Storage ist. Der ist nämlich nicht bei allen Geräten an der gleichen Stelle, wenn er denn überhaupt vorhanden ist. Mit der *showDialog()* Methode können wir den FileChooser dann anzeigen lassen,

```
fc.showDialog();
```

der dann, wenn der Nutzer eine Datei ausgewählt hat, die Methode *fileSelected()* aufruft (hat damit zu tun, dass wir oben das *FileSelectedListener* Interface implementiert haben):

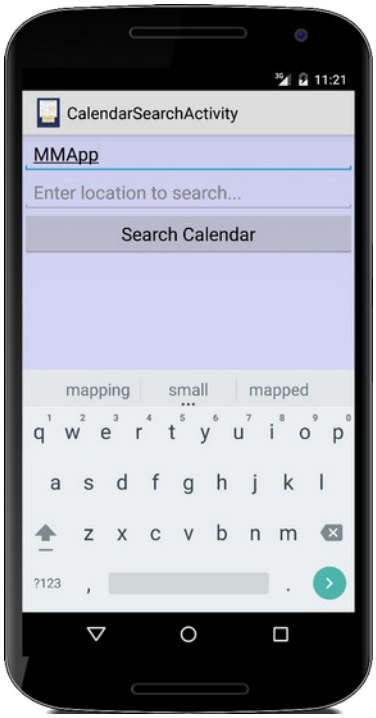
```
public void fileSelected(File file) {
    currentFile = file;
    try {
        StringBuffer sb = new StringBuffer();
        BufferedReader br = new BufferedReader(
            new FileReader(file));

        while (true) {
            String line = br.readLine();
            if (line == null)
                break;
            sb.append(line + "\n");
        }
        br.close();
        et.setText(sb);

    } catch (Exception e) {
        Log.e("EditorActivity", e.getMessage());
    }
}
```

Kommen wir zum Speichern: dafür lässt sich der *AlertDialog.Builder* gut verwenden. Wir haben ihn schon mal in der DialogActivity im zweiten Kapitel gesehen. Jetzt geben wir ihm aber einen EditText, was wir mit dem Aufruf von *dialog.setView()* bewirken:

```
private void openSaveFileDialog() {
    AlertDialog.Builder dialog = new AlertDialog.Builder(this);
    dialog.setTitle("Save");

    final EditText input = new EditText(this);
    input.setText(currentFile.getAbsolutePath());
    dialog.setView(input);

    dialog.setPositiveButton("Save",
        new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog,int which){
                String fileName = input.getText().toString();
                saveFile(fileName);
            }
        }
    );
    dialog.setNegativeButton("Cancel",
        new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog,int which){
                dialog.cancel();
            }
        }
    );
    dialog.show();
}
```

Der Rest ist ziemlich selbsterklärend, wir sind ja schon im 6. Semester, oder?

CalendarSearch



Um uns ein bisschen mit den ContentProvidern anzufreunden wollen wir eine kleine Calendarsuche implementieren. Aus Sicht von Android sind ContentProvider nichts anderes als Datenbanken. Das ist auch beim Calendar nicht anders. Wir verwenden also die ganz normale *query()* Methode wie wir sie bei der Datenbank kennen gelernt haben. Die Query gibt uns dann einen Cursor zurück, über den wir dann iterieren, und das war's auch schon:

```
private String searchTitleFromEventTable(
                                        String title) {
    String titles = "";

    String[] projection = {
        "_id",
        CalendarContract.Events.TITLE,
        CalendarContract.Events.EVENT_LOCATION,
        CalendarContract.Events.DTSTART,
        CalendarContract.Events.DTEND,
    };

    Uri uri = CalendarContract.Events.CONTENT_URI;
    String selection =
        CalendarContract.Events.TITLE + " LIKE ? ";
    String[] selectionArgs = new String[] {title};

    Cursor cur = getContentResolver().query(uri, projection,
        selection, selectionArgs, null);

    while (cur.moveToNext()) {
        String titl = cur.getString(
          cur.getColumnIndex(CalendarContract.Events.TITLE));
        titl += ":";
        titl += cur.getString(
          cur.getColumnIndex(CalendarContract.Events.EVENT_LOCATION));
        titles += titl + "\n";
    }
    cur.close();
    return titles;
}
```

Effektiv wird aus dem obigen Code eine ganz normale SQL Abfrage gemacht. Wir können die Analogie mal im Einzeln aufdröseln:

- **uri:** hierüber wird festgelegt in welcher Tabelle wir suchen wollen,
- **projection:** gibt an welche Spalten uns interessieren,
- **selection:** das ist die eigentliche Abfrage, wonach wir suchen,
- **selectionArgs:** sind die Argumente für die *selection*, wie bei einer Stored Procedure, und
- **null:** hier könnten noch zusätzliche Argumente kommen, wie z.B. ORDER BY.

Also in SQL übersetzt entspricht der obige Code in etwa:

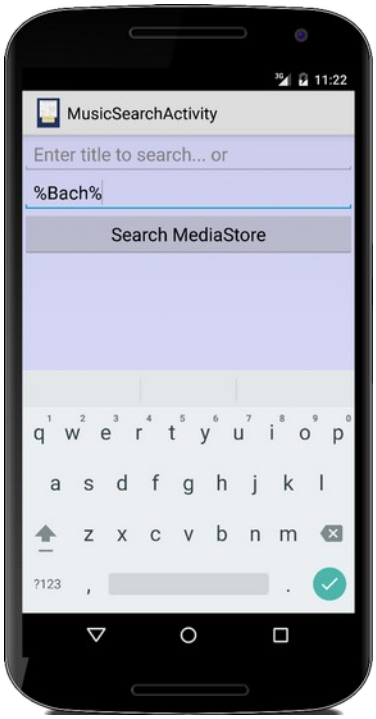
```
SELECT TITLE, EVENT_LOCATION, DTSTART, DTEND FROM
    CalendarContract.Events WHERE TITLE LIKE title;
```

Eigentlich ist SQL gar nicht so schlecht, effizienter auf jeden Fall.

Ah, eine Sache noch: wir wollen ja nicht, dass jeder auf unsere Kalenderdaten zugreifen darf, deswegen müssen wir den Nutzer um Erlaubnis fragen, ob wir auf seinen Kalender zugreifen dürfen. Dazu müssen wir die folgende Zeile noch im AndroidManifest hinzufügen:

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

MusicSearch



Android hat auch eine Datenbank für Medien, den *MediaStore*: beim Booten oder beim Einfügen einer SD Karte scannt Android alle Dateien und sucht nach Audio-, Video- und Bilddateien. Was es dann so an Metadaten gefunden hat speichert es im MediaStore ab.

Um den MediaStore mal verwendet zu haben, wollen wir nach einem Künstler in den Audiodateien suchen:

```
private String searchArtistFromMediaStore(
                                        String artist) {
    String artists = "";

    String[] projection = {
        "_id",
        MediaStore.Audio.Media.TITLE,
        MediaStore.Audio.Media.ALBUM,
        MediaStore.Audio.Media.ARTIST,
        MediaStore.Audio.Media.COMPOSER,
    };

    Uri uri =
       MediaStore.Audio.Media.EXTERNAL_CONTENT_URI;
    String selection =
       MediaStore.Audio.Media.ARTIST + " LIKE ? ";
    String[] selectionArgs = new String[] { artist };

    Cursor cur = getContentResolver().query(uri, projection,
                                     selection, selectionArgs, null);

    while (cur.moveToNext()) {
        String titl = cur.getString(
            cur.getColumnIndex(MediaStore.Audio.Media.TITLE));
        titl += ":" + cur.getString(
            cur.getColumnIndex(MediaStore.Audio.Media.ARTIST));
        titl += ":" + cur.getString(
            cur.getColumnIndex(MediaStore.Audio.Media.COMPOSER));
        artists += titl + "\n";
    }
    cur.close();

    return artists;
}
```

Der Code ist komplett analog zu dem vom vorherigen Beispiel, der einzige Unterschied ist, dass wir jetzt in der Tabelle MediaStore.Audio suchen.

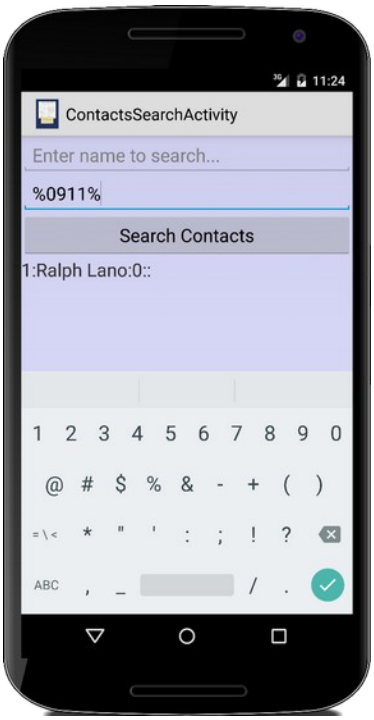
Eine interessante Anmerkung sei hier vielleicht, dass eine App keine Berechtigungen benötigt um auf den MediaStore zuzugreifen. Aus Sicht eines Entwicklers ist das natürlich praktisch. Aus Sicht des Nutzers vielleicht nicht so, denn der MediaStore speichert auch Thumbnails.

Übrigens, den Prozess des Durchstöperns kann man mit folgenden Zeilen anstoßen:

```
sendBroadcast(
    new Intent(
      Intent.ACTION_MEDIA_MOUNTED,
      Uri.parse("file://" + Environment.getExternalStorageDirectory())
    )
);
```

Das dauert allerdings ein bisschen (Stunden) bis es fertig ist.

ContactsSearch



Als dritte Anwendung wollen wir mal in den Kontakten suchen. Das geht zwar schon recht gut mit der normalen Suche, aber was nicht geht ist nach Teilen zu suchen. Also z.B. nach Telefonnummern die mit "0911" losgehen.

Wie viele von den ContentProvidern, besteht auch die ContactsContract Datenbank aus mehreren Tabellen. Um jetzt nach Telefonnummern suchen zu können, müssen wir erst mal wissen wessen Telefonnummer. Deswegen gehen wir einfach mal alle Kontakte in unserer Datenbank durch:

```
private String searchTelephoneInContacts(
                                    String telephone) {
    String numbers = "";

    String[] projection = {
        ContactsContract.Contacts._ID,
        ContactsContract.Contacts.DISPLAY_NAME,
        ContactsContract.Contacts.HAS_PHONE_NUMBER,
        ContactsContract.Contacts.TIMES_CONTACTED,
    };

    Uri uri =
        ContactsContract.Contacts.CONTENT_URI;
    ContentResolver cr = getContentResolver();
    Cursor cur = cr.query(uri, projection, null, null, null);

    while (cur.moveToNext()) {
        String id = cur.getString(cur.getColumnIndex(
            ContactsContract.Contacts._ID));
        String displayName = cur.getString(cur.getColumnIndex(
            ContactsContract.Contacts.DISPLAY_NAME));
        Integer hasPhone = cur.getInt(cur.getColumnIndex(
            ContactsContract.Contacts.HAS_PHONE_NUMBER));
        Integer timesContacted = cur.getInt(cur.getColumnIndex(
            ContactsContract.Contacts.TIMES_CONTACTED));

        ...
    }
    cur.close();

    return numbers;
}
```

Die obigen Zeilen entsprechen einem "SELECT * FROM contacts;", wenn man in SQL denken mag.

Hier sehen wir zwei interessante Informationen: *hasPhone* und *timesContacted*. Die erste benötigen wir gleich wenn wir nach Telefonnummern suchen wollen. Die zweite ist insofern interessant, als dass Android anscheinend zählt, wie häufig wir mit jemandem Kontakt haben. Man will ja nichts Schlimmes denken, aber in Zeiten von Edward Snowden, fragt man sich ist das wirklich notwendig?

Aber machen wir weiter: es macht natürlich nur Sinn bei Leuten nach Telefonnummern zu suchen, wenn sie welche haben. Falls sie aber eine haben, dann suchen wir in der Telefonnummer Tabelle (CommonDataKinds.Phone) nach unserer gesuchten Nummer. Und da wir fit in SQL sind, verwenden wir den "LIKE" Operator, d.h. wir können mit "%" auch Teilsuchen ausführen:

```
...
String phone = null;
if (hasPhone > 0) {
    String selection =
        ContactsContract.CommonDataKinds.Phone.CONTACT_ID;
    selection += " = ? AND ";
    selection += ContactsContract.CommonDataKinds.Phone.NUMBER;
    selection += " LIKE ?";

    String[] selectionArgs = new String[] { id, telephone };

    Cursor cp = cr.query(
        ContactsContract.CommonDataKinds.Phone.CONTENT_URI, null,
                                    selection, selectionArgs, null);
    if (cp != null && cp.moveToFirst()) {
        // while (cp.moveToNext()) {
        phone = cp.getString( cp.getColumnIndex(
            ContactsContract.CommonDataKinds.Phone.NUMBER));
    }
    cp.close();
}

if (phone != null && phone.length() > 0) {
    String titl = id;
    titl += ":" + displayName;
    titl += ":" + timesContacted;
    titl += ":" + phone;
    numbers += titl + "\n";
}
...
```

Ziemlich cool, oder? Auch das Stöpern durch die Kontakte ist nur möglich, wenn wir uns die Erlaubnis dazu vom Nutzer holen. Dazu müssen wir folgende Zeile in die AndroidManifest.xml Datei einfügen:

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

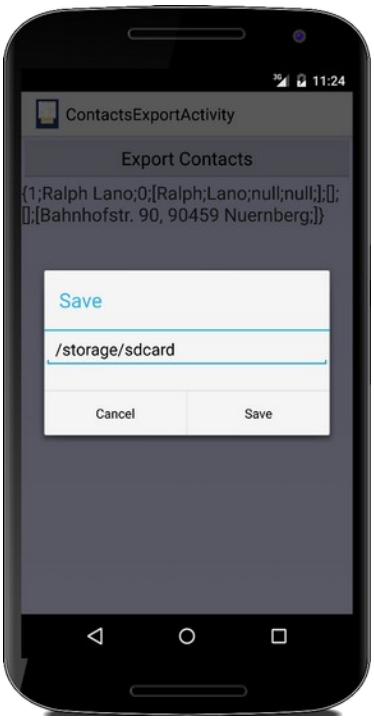
Von daher ist es vielleicht gar nicht so schlimm, wenn Android mitzählt wie häufig wir Kontakt mit jemandem haben. WhatsApp und Facebook freuen sich natürlich auch über diese Info.

ContactsExport

Ab und zu möchte man vielleicht mal einen Backup von seinen Kontaktdaten machen. Man kann jetzt natürlich im Play Store nach irgendeiner App suchen die das macht, aber da weiß man nie genau was die App dann noch so mit den eigenen Daten macht.

Deswegen wollen wir eine Anwendung schreiben, die alle Kontaktdaten ausliest und in eine Datei schreibt. Als Format verwenden wir JSON [9], da es uns erlaubt die Struktur in den Daten zu erhalten. Das Vorgehen ist ähnlich wie im letzten Beispiel: erst müssen wir uns alle Kontakte in der ContactsContract.Contacts Tabelle holen, und danach in den verschiedenen Untertabellen nach den Details suchen, als da sind:

- CommonDataKinds.Email: für die Emails,
- CommonDataKinds.StructuredName: für Vorname, Nachname, Titel, etc.,
- CommonDataKinds.StructuredPostal: für die Addressen und
- CommonDataKinds.Phone: für die Telefonnummern.



Jede dieser Untertabellen kann mehr als einen Datensatz enthalten, es soll ja Leute geben die mehr als eine Telefonnummer haben. Der Code sieht jetzt etwas lang aus, aber jeder einzelne Schritt ist nachvollziehbar:

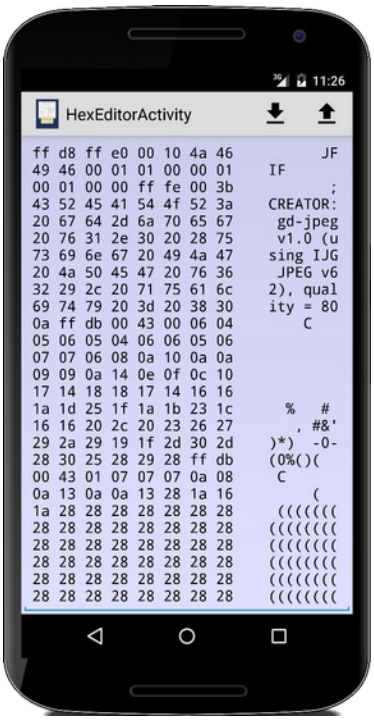
```
        private String getAllContacts() {
            String allContacts = "";

            String[] projection = {
                ContactsContract.Contacts._ID,
                ContactsContract.Contacts.DISPLAY_NAME,
                ContactsContract.Contacts.HAS_PHONE_NUMBER,
                ContactsContract.Contacts.TIMES_CONTACTED,
            };

            Uri uri = ContactsContract.Contacts.CONTENT_URI;
            String orderBy = ContactsContract.Contacts.DISPLAY_NAME + " ASC";

            ContentResolver cr = getContentResolver();
            Cursor cur = cr.query(uri, projection, null, null, orderBy);

            while (cur.moveToNext()) {
                String id = cur.getString(cur.getColumnIndex(
                    ContactsContract.Contacts._ID));
                String displayName = cur.getString(cur.getColumnIndex(
                    ContactsContract.Contacts.DISPLAY_NAME));
                Integer hasPhone = cur.getInt(cur.getColumnIndex(
                    ContactsContract.Contacts.HAS_PHONE_NUMBER));
                Integer timesContacted = cur.getInt(cur.getColumnIndex(
                    ContactsContract.Contacts.TIMES_CONTACTED));

                String email = "[";
                Cursor ce = cr.query(
                  ContactsContract.CommonDataKinds.Email.CONTENT_URI, null,
                  ContactsContract.CommonDataKinds.Email.CONTACT_ID + " = ?",
                  new String[] { id }, null);

                // if (ce != null && ce.moveToFirst()) {
                while (ce.moveToNext()) {
                    email += ce.getString(ce.getColumnIndex(
                        ContactsContract.CommonDataKinds.Email.DATA));
                    email += ";";
                }
                ce.close();
                email += "]";

                String structuresName = "[";

                String whereName =
                   ContactsContract.Data.MIMETYPE + " = ? AND " +
                   ContactsContract.CommonDataKinds.StructuredName.CONTACT_ID+
                   " = ?";

                String[] whereNameParams = new String[] {
ContactsContract.CommonDataKinds.StructuredName.CONTENT_ITEM_TYPE,id};

                Cursor cn = cr.query(
                  ContactsContract.Data.CONTENT_URI, null,
                  whereName, whereNameParams,
                  ContactsContract.CommonDataKinds.StructuredName.GIVEN_NAME);
```

```
            while (cn.moveToNext()) {
              structuresName += cn.getString(cn.getColumnIndex(
                ContactsContract.CommonDataKinds.StructuredName.GIVEN_NAME));
              structuresName += ";" + cn.getString(cn.getColumnIndex(
                ContactsContract.CommonDataKinds.StructuredName.FAMILY_NAME));
              structuresName += ";" + cn.getString(cn.getColumnIndex(
                ContactsContract.CommonDataKinds.StructuredName.MIDDLE_NAME));
              structuresName += ";" + cn.getString(cn.getColumnIndex(
                ContactsContract.CommonDataKinds.StructuredName.PREFIX));
              structuresName += ";";
            }
            cn.close();
            structuresName += "]";

            String address = "[";
            Cursor ca = cr.query(
                ContactsContract.CommonDataKinds.StructuredPostal.CONTENT_URI,
                null,
                ContactsContract.CommonDataKinds.StructuredPostal.CONTACT_ID
                + " = ?",
                new String[] { id }, null);

            // if (ca != null && ca.moveToFirst()) {
            while (ca.moveToNext()) {
                address += ca.getString(ca.getColumnIndex(
                    ContactsContract.CommonDataKinds.StructuredPostal.DATA));
                address += ";";
            }
            ca.close();
            address += "]";

            String phone = "[";
            if (hasPhone > 0) {
                Cursor cp = cr.query(
                   ContactsContract.CommonDataKinds.Phone.CONTENT_URI, null,
                   ContactsContract.CommonDataKinds.Phone.CONTACT_ID + " = ?",
                   new String[] { id }, null);

                // if (cp != null && cp.moveToFirst()) {
                while (cp.moveToNext()) {
                    phone += cp.getString(cp.getColumnIndex(
                        ContactsContract.CommonDataKinds.Phone.NUMBER));
                    phone += ";";
                }
                cp.close();
            }
            phone += "]";

            String titl = "{" + id;
            titl += ";" + displayName;
            titl += ";" + timesContacted;
            titl += ";" + structuresName;
            titl += ";" + phone;
            titl += ";" + email;
            titl += ";" + address;
            allContacts += titl + "}\n";
          }
          cur.close();

          return allContacts;
        }
```

In dem Beispiel sehen wir auch, wie man mit *orderBy* sortieren kann. Die App ist jetzt nicht die schnellste, wenn man aber einen StringBuffer anstelle der vielen Strings verwenden würde, wäre das Ganze bestimmt zehnmal schneller.

Challenges

HexEditor



Basierend auf unserer Editor Activity kann man relativ einfach auch einen HexEditor implementieren. Anstelle eines FileReaders müssen wir aber einen FileInputStream in der *fileSelected()* Methode verwenden, da wir es ja mit binären Daten zu tun haben:

```
public void fileSelected(File file) {
    try {
        FileInputStream fis =
            new FileInputStream(file);

        StringBuffer sbHex = new StringBuffer();
        StringBuffer sbAscii = new StringBuffer();
        int count = 0;

        int content;
        while ((content = fis.read()) != -1) {
            if (content > 31 && content < 127) {
                sbAscii.append((char) content);
            } else {
                sbAscii.append(' ');
            }

            byte b = (byte) content;
            char c1 = DIGITS[(b >> 4) & 0xf];
            char c2 = DIGITS[b & 0xf];
            sbHex.append(c1);
            sbHex.append(c2);
            sbHex.append(' ');

            count++;
            if (count == NR_OF_BYTES_PER_LINE) {
                sbHex.append("   " + sbAscii + "\n");
                sbAscii = new StringBuffer();
                count = 0;
            }
        }

        fis.close();

        et.setText(sbHex.toString());

    } catch (Exception e) {
        Log.e("HexEditorActivity", e.getMessage());
    }
}
```

Und analog in der *saveFile()* Methode.

ContactsImport

Uns fehlt noch das Gegenstück zu ContactsExport mit dem wir die Kontaktdaten wieder importiert können. Wir müssen also die JSON Datei parsen, vielleicht mit dem JSONObject Parser aus dem Netzwerkkapitel, da sparen wir uns viel Arbeit. Danach fügen wir die Kontakte einfach mithilfe des ContactsContracts wieder ein.

Fragen

1. Wenn Sie Ihr Android-Gerät ausschalten, gehen alle flüchtigen Daten (d.h. Daten die im RAM sind) verloren. Nennen Sie drei Möglichkeiten, um Daten auf einem Android Gerät zu persistieren, also so zu speichern, dass sie auch nach dem Ausschalten des Geräts wieder hergestellt werden können**.**

2. Was ist der Vorteil von Ressourcen und Assets und was ist deren Nachteil?

3. Wofür verwendet man Shared Preferences?

4. Was unterscheidet Internal Storage von External Storage?

5. Wie heißt die Datenbank die Teil von Android ist?

6. Sie können die Daten Ihrer Applikation sowohl in einer lokalen Datenbank als auch auf dem Dateisystem speichern. Geben Sie je ein Beispiel wann das eine und wann das andere Sinn macht.

7. Wie können Sie Zugriff auf die Bookmarks des Browser bekommen?

8. Nennen Sie drei der standard Content Providers die es auf jedem Android gerät gibt.

9. Erklären Sie kurz, welche Information jeder der folgenden Content Provider bereitstellt:
   - ContactsContract:
   - MediaStore:
   - Browser:
   - CallLog:
   - UserDictionary:
   - Settings:

10. Was kann alles schief gehen wenn Sie mit Dateien arbeiten (öffnen, lesen, schreiben)?

11. Welche Klasse verwenden Sie um die Größe einer Datei festzustellen?

12. Erklären Sie kurz was Serialisierung ist, und welche der Stream Klassen Sie dafür verwenden müssen.

13. Nehmen Sie an Sie müssten ein Objekt der Klasse Point auf die SD Karte speichern, z.B. mit einem FileOutputStream. Erläutern Sie wie Sie das tun würden einmal mit Serialization und einmal ohne Serialization.

```
class Point {
    public int x;
    public int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Referenzen

[1] SQL, en.wikipedia.org/wiki/SQL

[2] Using SQLite Database with Android, www.codeproject.com/Articles/119293/Using-SQLite-Database-with-Android

[3] Sugar ORM, satyan.github.io/sugar/index.html

[4] Room Persistence Library, https://developer.android.com/topic/libraries/architecture/room

[5] greenDAO, greenrobot.org/greendao/

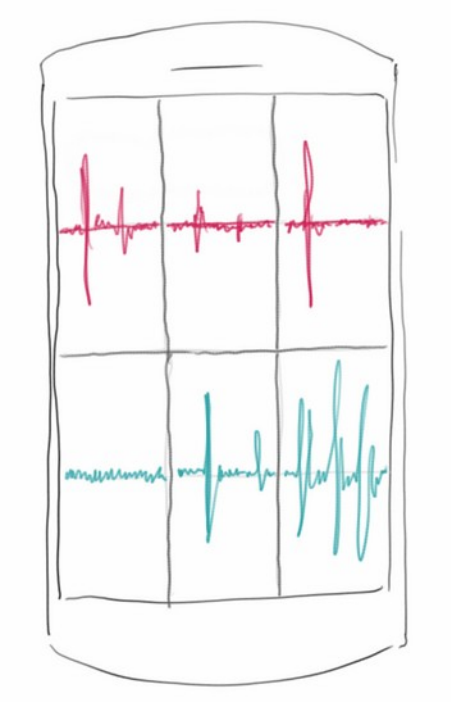
[6] Firebase Android Codelab, https://codelabs.developers.google.com/codelabs/firebase-android/#0

[7] Realm Database, https://realm.io/products/realm-database

[8] Roger Keays, FileChooser, http://www.ninthavenue.com.au/simple-android-file-chooser

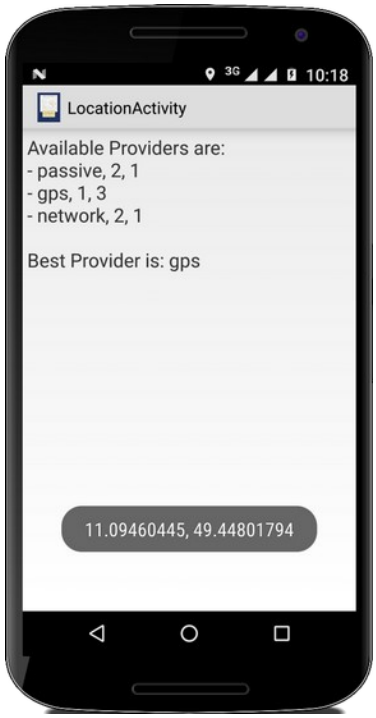
[9] JSON, https://en.wikipedia.org/wiki/JSON

Sensors



Die meisten Android-Geräte haben Sensoren, manche sogar ganz viele. Im Prinzip unterscheidet man zwischen drei verschiedenen Arten von Sensoren: den Positions-, den Umwelt- und den Bewegungs-Sensoren. Von den Positions-Sensoren, meist GPS, erhält man seine Position in Längen- und Breitengraden, manchmal auch die Höhe über dem Meeresspiegel. Mit den Umweltsensoren kann man Helligkeit, Temperatur und je nach Gerät manchmal auch Luftdruck messen. Bewegungssensoren messen Linear- und Drehbeschleunigung. Auch der Magnetfeldsensor gehört zu den Bewegungssensoren. Technisch müsste man wohl sowohl das Mikrofon als auch die Kamera als Sensoren bezeichnen. Sogar die Batterie ist als Sensor zu gebrauchen: man kann mit ihr Temperatur messen.

Location



Beginnen wir mit einem der wichtigsten Sensoren dem GPS Sensor [3]. Praktisch jedes Handy hat einen. Man verwendet ihn meistens um festzustellen wo man sich befindet. Man kann ihn aber auch verwenden um seine Geschwindigkeit und in Abwesenheit eines Magnetfeldsensors auch seine Richtung zu bestimmen, grob wenigstens.

Hat man keinen GPS ist noch nicht alles verloren, Android hat nämlich noch zwei andere Möglichkeiten zu bestimmen wo es sich befindet. Zum einen über das Mobilfunknetz: jedes Handy verbindet sich mit dem Mobilfunknetz über die Mobilfunkmasten. Da diese Masten aber eindeutige Identifikationsnummern haben, und außerdem bekannt ist, wo sich welcher Mast befindet, kann man auf diese Art und Weise ganz grob die Position bestimmen. Es muss sich eben in der Nähe dieses einen Masten befinden. Zum anderen kann man auch über das Wifi-Netz feststellen wo jemand ist, wenn man weiß wo der Wifi-Accesspoint steht.

Die Genauigkeit der verschiedenen Methoden ist ganz unterschiedlich: GPS ist die genaueste, mit einer Auflösung von ca. 5 bis 10 Metern. Es kostet aber auch den meisten Strom. Die Ortsbestimmung über das Mobilfunknetz ist viel ungenauer, in Städten manchmal auf ca. 100 Meter genau, auf dem Land sind es eher Kilometer. Aber es kostet keine zusätzliche Energie. Wifi ist irgendwo dazwischen, allerdings funktioniert es eher selten.

Wie weiß man jetzt welche LocationProvider denn überhaupt zur Verfügung stehen? Man fragt einfach den *LocationManager*:

```
protected void onCreate(Bundle savedInstanceState) {
    ...

    LocationManager locationManager =
        (LocationManager) getSystemService(LOCATION_SERVICE);

    // list all providers
    String msg = "Available Providers are:\n";
    List<String> providers = locationManager.getAllProviders();
    for (String providerName : providers) {
        LocationProvider provider =
            locationManager.getProvider(providerName);
        msg += "- " + provider.getName() + ", ";
        msg += provider.getAccuracy() + ", ";
        msg += provider.getPowerRequirement() + "\n";
    }
    tv.setText(msg);

    // get best provider
    Criteria criteria = new Criteria();
    criteria.setAccuracy(Criteria.ACCURACY_FINE);
    String bestProvider =
        locationManager.getBestProvider(criteria, true);
    tv.append("\nBest Provider is: " + bestProvider);
}
```

Will man noch wissen welcher denn der beste Provider ist, dann verwendet man die *Criteria* Klasse, und sagt z.B. man möchte einen Provider mit der ACCURACY_FINE.

Nachdem wir wissen welchen unserer LocationProvider wir wollen, ist der nächste Schritt diesen zu verwenden. Eine Möglichkeit ist das LocationListener Interface zu implementieren und die *onLocationChanged()* Methode zu überschreiben:

```
public class LocationActivity extends Activity
                              implements LocationListener {
    protected void onCreate(Bundle savedInstanceState) {
        ...
        locationManager.requestLocationUpdates(bestProvider, 1000,
                                               1, this);
    }

    public void onLocationChanged(Location location) {
        String msg = "" + location.getLongitude() + ", " +
            location.getLatitude();
        Toast.makeText(this, msg, Toast.LENGTH_LONG).show();
    }

    public void onPause() {
        locationManager.removeUpdates(this);
        super.onPause();
    }
}
```

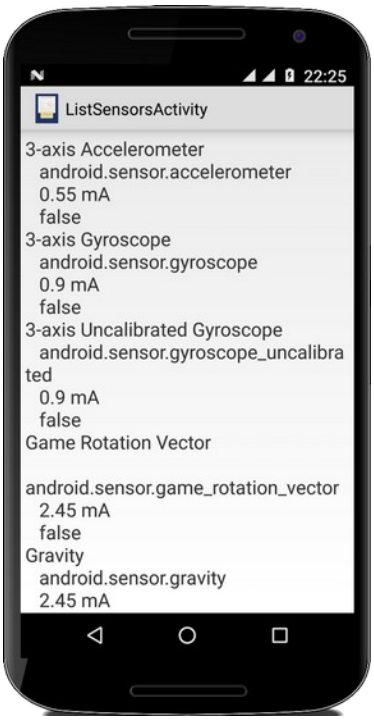
Damit der LocationListener aber weiß welchen der vielen Provider er denn verwenden soll, müssen wir ihm das mitteilen mit der *requestLocationUpdates()* Methode: hier sagen wir welchen Provider er verwenden soll (bestProvider), wie oft er die Position updaten soll (alle 1000 ms) und was so die Distanzänderungen sind, die uns interessieren (1 Meter). Man sollte immer bedenken, dass der GPS Sensor ziemlich viel Strom frisst, deswegen je seltener wir ihn verwenden, desto seltener müssen wir auch unser Handy wieder aufladen. Auch was wir nicht vergessen dürfen ist, den GPS Sensor wieder auszuschalten, wenn wir ihn nicht mehr brauchen. Das machen wir in der onPause() Methode oben.

Damit das Program auch funktioniert, muss natürlich GPS eingeschaltet sein und man muss den Nutzer um Erlaubnis fragen. Das machen wir wie üblich in der AndroidManifest Datei:

```
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

List Sensors

Was gibt es denn noch für Sensoren, außer dem GPS Sensor? Am besten fragt man den SensorManager, der weiß nämlich Bescheid:




```
SensorManager mSensorManager =
  (SensorManager) getSystemService(SENSOR_SERVICE);
List<Sensor> sensorList =
  mSensorManager.getSensorList(Sensor.TYPE_ALL);

String msg = "";
for (Sensor sensor : sensorList) {
    String type = "";
    if (Build.VERSION.SDK_INT >= 20) {
        type = sensor.getStringType();
    } else {
        type = "" + sensor.getType();
    }
    msg += sensor.getName() + "\n";
    msg += "   " + type + "\n";
    msg += "   " + sensor.getPower() + " mA\n";
    if (Build.VERSION.SDK_INT >= 21) {
      msg += "   " + sensor.isWakeUpSensor() +"\n";
    }
}
TextView tv = (TextView) findViewById(R.id.textview);
tv.setText(msg);
```

Interessant im Beispiel oben ist vielleicht wie wir mit *Build.VERSION.SDK_INT* feststellen, welche Version von Android unser Nutzer verwendet, und abhängig davon Methoden aufrufen, die es erst ab einer gewissen Version gibt.

Environmental

Betrachten wir die Umwelt-Sensoren etwas näher. Das grobe Gerüst ist bei allen Sensoranwendungen gleich:


```
public class SensorActivity extends Activity
                    implements SensorEventListener {

    private SensorManager mSensorManager;

    @Override
    public final void onCreate(Bundle savedState) {
        ...
        mSensorManager = (SensorManager)
          getSystemService(Context.SENSOR_SERVICE);
    }

    @Override
    public final void onSensorChanged(
                                SensorEvent event) {
        switch (event.sensor.getType()) {
        case Sensor.TYPE_LIGHT:
          float illuminance_in_lx =event.values[0];
          ...
          break;
        }
    }

    @Override
    protected void onResume() {
        super.onResume();

        Sensor mLight =
            mSensorManager.getDefaultSensor(Sensor.TYPE_LIGHT);
        if (mLight != null) {
            mSensorManager.registerListener(this, mLight,
                                    SensorManager.SENSOR_DELAY_NORMAL);
        }
    }

    @Override
    protected void onPause() {
        super.onPause();
        mSensorManager.unregisterListener(this);
    }

    @Override
    public final void onAccuracyChanged(Sensor sensor, int accuracy) {
    }
}
```

Wir erweitern den *SensorEventListener* und implementieren dessen Methoden *onSensorChanged()* und *onAccuracyChanged()*, wobei wir praktisch nur onSensorChanged() verwenden werden. Dann benötigen wir einen SensorManager über den wir unsere Sensoren bekommen. Was dann noch wichtig ist, dass wir in der onResume() unsere Sensoren registrieren und in der onPause() wieder deregistrieren.

Im Beispiel oben haben wir den Licht-Sensor verwendet, den fast jedes Android Gerät hat. Außerdem findet man noch folgende Sensoren mit abnehmender Häufigkeit:

- TYPE_LIGHT: misst die Luminosität in Lux,
- TYPE_PROXIMITY: misst die Entfernung in cm,
- TYPE_TEMPERATURE: misst die Temperatur in Grad Celsius, und
- TYPE_PRESSURE: misst den Luftdruck in mbar.

Den jeweilgen Wert des Sensors erhalten wir in der onSensorChanged() Methode über den SensorEvent Parameter.

Wie immer bei Sensoren, verbrauchen die meisten viel Strom. Wir können darauf Einfluss nehmen, wenn wir die Sensoren nicht so häufig verwenden. Einstellen können wir das über den letzten Parameter der *registerListener()* Methode:

- SENSOR_DELAY_NORMAL: entspricht 200.000 Mikrosekunden,
- SENSOR_DELAY_GAME: entspricht 20.000 Mikrosekunden,
- SENSOR_DELAY_UI: entspricht 60.000 Mikrosekunden, und
- SENSOR_DELAY_FASTEST: entspricht 0 Mikrosekunden.

Man kann aber auch einfach einen Wert von Hand übergeben, natürlich in Mikrosekunden.

Motion

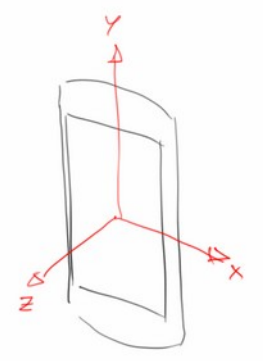
Kommen wir zu den Bewegungs-Sensoren. Es gibt da im Prinzip drei Arten:

- TYPE_ACCELEROMETER: misst Beschleunigung in m/s2,
- TYPE_GYROSCOPE: misst Rotationsbewegungen in rad/s, und
- TYPE_MAGNETIC_FIELD: misst Magnetfelder in Mikrotesla.

Fast alle Android Geräte haben einen Beschleunigungssensor. In letzter Zeit ist der Gyroskopsensor auch fast in jedem Gerät zu finden, der Magnetfeldsensor ist allerdings eher selten. D.h. vor allem mit Herstellungskosten zu tun.

Die Verwendung ist komplett analog zu den Umwelt-Sensoren, der einzige Unterschied liegt darin, dass wir jetzt drei Werte erhalten:

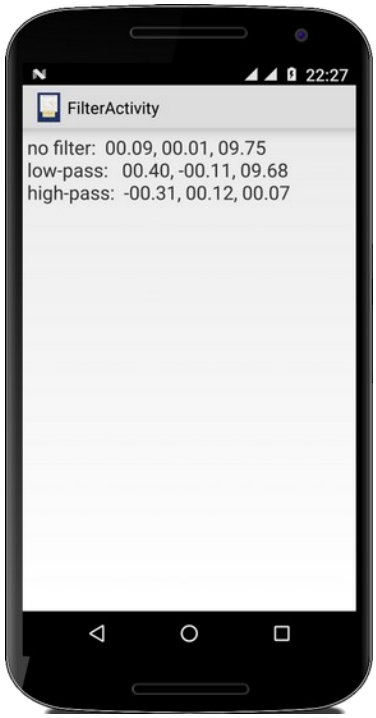
```
public final void onSensorChanged(SensorEvent evnt)
{
  switch (evnt.sensor.getType()) {
  case Sensor.TYPE_ACCELEROMETER:
    float[] acceleration_in_m_per_s2 =event.values;
    msg[0] = "ACCEL:  " +
      df.format(acceleration_in_m_per_s2[0]) +", "+
      df.format(acceleration_in_m_per_s2[1]) +", "+
      df.format(acceleration_in_m_per_s2[2]) +
      " m/s2\n";
    break;
  ...
  }
}
```



Je einer für die x-, die y- und die z-Richtung. Die Richtungen sind dabei relativ zum Gerät definiert. Die z-Richtung kommt aus dem Gerät heraus, die y-Richtung zeigt nach oben, und die x-Richtung zeigt nach rechts.

Wer gerne wissen möchte wie diese Sensoren wirklich funktionieren, findet in dem GoogleTechTalk von David Sachs eine gelungene Zusammenfassung [1].

Filter



Bei den drei Bewegungssensoren, die wir gerade kennengelernt haben, handelt es sich um Hardwaresensoren. Was wir bekommen sind die ungefilterten Rohdaten. Das sieht man daran, dass die Werte, speziell beim Beschleunigungssensor, relativ wild hin- und herspringen. Für viele Anwendungen ist das eher störend und man möchte die Daten *filtern*. Die am häufigsten verwendeten Filter sind der Tiefpass (low-pass) und der Hochpass (high-pass).

Der Tiefpass ist im Prinzip ein gleitender Durchschnitt. Beim Beschleunigungssensor bewirkt er, dass die schnellen Schwankungen verschwinden, und alles was übrig bleibt ist die konstante Beschleunigung der Schwerkraft. Der Name kommt daher, dass der Tiefpass niedere Frequenzen ungehindert durchlässt, während er für hohe Frequenzen, also sich schnell ändernde Werte, nahezu undurchlässig ist. Er lässt also *tief*e Frequenzen *pass*ieren.

```
private float lowPass(float current, float average)
{
    return average * LOW_PASS_FACTOR +
           current * (1- LOW_PASS_FACTOR);
}
```

Der Hochpass ist das Gegenstück dazu und ist für niedere Frequenzen undurchlässig, dafür kommen die hohen Frequenzen nahezu ungehindert durch. Beim Beschleunigungssensor bedeutet das, dass die Schwerkraftkomponente herausgefiltert wird und lediglich die schnellen Hin- und Herbewegungen übrig bleiben. Allerdings bleibt das Rauschen (noise) erhalten, was für die meisten Anwendungen ein Problem darstellt.

```
private float highPass(float current, float average) {
    return current - average;
}
```

Die Grenzfrequenz wird durch die beiden Konstanten bestimmt:

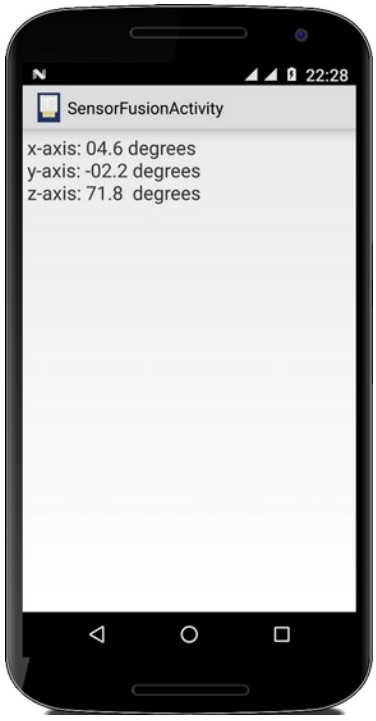
```
private final float LOW_PASS_FACTOR = 0.8f;
private final int TIME_RESOLUTION = 100;
```

Dabei hat der LOW_PASS_FACTOR einen Wert zwischen 0 und 1, während die TIME_RESOLUTION meist in Millisekunden gemessen wird. Im Code sieht das dann so aus:

```
private long lastTime;
private float[] avg_acceleration = new float[3];

public final void onSensorChanged(SensorEvent event) {
    long currentTime = System.currentTimeMillis();
    if (currentTime - lastTime > TIME_RESOLUTION) {
        lastTime = currentTime;

        switch (event.sensor.getType()) {
        case Sensor.TYPE_ACCELEROMETER:
            float[] acceleration_in_m_per_s2 = event.values;

            // low-pass
            for (int i = 0; i < 3; i++) {
              avg_acceleration[i] = lowPass(
                acceleration_in_m_per_s2[i], avg_acceleration[i]);
            }
```

```
                    // high-pass
                    float[] high_pass = new float[3];
                    for (int i = 0; i < 3; i++) {
                      high_pass[i] = highPass(
                        acceleration_in_m_per_s2[i], avg_acceleration[i]);
                    }

                    msg[0] = "no filter:  "
                        + df.format(acceleration_in_m_per_s2[0]) + ", "
                        + df.format(acceleration_in_m_per_s2[1]) + ", "
                        + df.format(acceleration_in_m_per_s2[2]) + "\n";
                    msg[0] += "low-pass:   "
                        + df.format(avg_acceleration[0])
                        + ", " + df.format(avg_acceleration[1]) + ", "
                        + df.format(avg_acceleration[2]) + "\n";
                    msg[0] += "high-pass:  " + df.format(high_pass[0])
                        + ", " + df.format(high_pass[1]) + ", "
                        + df.format(high_pass[2]) + "\n";
                    break;
                ...
            }
        }
```

Sensor Fusion



Wir haben vorhin von Hardwaresensoren gesprochen. Das deutet an, dass es auch Softwaresensoren gibt. Das Problem mit den Hardwaresensoren ist, dass sie einmal ganz schön rauschen und zum andern auch nicht besonders genau sind. Der Klassiker ist Indoor-Navigation basierend auf dem Beschleunigungssensor. Es funktioniert einfach nicht (wobei die Gründe hier eher vom Zweifachintegral stammen und weniger von der Ungenauigkeit des Sensors [1]).

Die Idee hinter den Softwaresensoren ist nun, die Hardwaresensoren gegeneinander auszuspielen, also aus mehreren Hardwaresensoren einen Softwaresensor zu erzeugen. Glücklicherweise macht Android das für uns, sonst müssten wir uns noch mit Kalman Filtern beschäftigen [2]. Auf den meisten Android Geräten gibt es die folgenden Softwaresensoren:

- TYPE_GRAVITY
- TYPE_LINEAR_ACCELERATION
- TYPE_GAME_ROTATION_VECTOR
- TYPE_ROTATION_VECTOR

Die beiden ersten machen faktisch genau das Gleiche was wir gerade bei den Filtern gemacht haben: TYPE_GRAVITY enthält den Schwerkraftteil der Beschleunigung, sagt uns also wo unten ist, und TYPE_LINEAR_ACCELERATION ist das was übrig bleibt wenn man TYPE_GRAVITY von den Rohdaten (TYPE_ACCELERATION) abzieht.

Wenn man weiß wo die Gyroskopdaten herkommen, wundert man sich, dass man die für irgendetwas verwenden kann. Trotzdem gehört der Gyroskopsensor zu den besseren. Man kann ihn aber noch besser machen, wenn man ihn über "Sensor Fusion" mit dem Beschleunigungssensor kombiniert [1]. Was dann rauskommt ist der TYPE_GAME_ROTATION_VECTOR. Man kann ihn relativ gut verwenden um Rotationen zu messen und auch um die Ausrichtung des Handys im Raum zu erhalten.

Allerdings hat der TYPE_GAME_ROTATION_VECTOR einen leichten Drift, er bleibt also nicht stabil, sondern driftet immer ein bisschen (das kommt von dem Gyrosensor). Hat man aber das Glück in seinem Handy auch noch einen Magnetfeldsensor zu haben, kann man diesen Drift (auch wieder über Sensor Fusion) korrigieren. Das ist dann der TYPE_ROTATION_VECTOR. Die mit Abstand saubersten Daten liefert TYPE_ROTATION_VECTOR. D.h. wann immer dieser Sensor zur Verfügung steht, empfiehlt es sich ihn zu verwenden.

Verwendet werden die Softwaresensoren genauso wie die Hardwaresensoren. Die Werte die beim TYPE_GAME_ROTATION_VECTOR zurückkommen sind normalisiert zwischen -1 und +1, wenn man sie mit 180 multipliziert, dann erhält man Grad.

```
public final void onSensorChanged(SensorEvent event) {
    String msg = "";
    switch (event.sensor.getType()) {
    case Sensor.TYPE_GAME_ROTATION_VECTOR:
        float[] angle = event.values;
        msg =  "x-axis: " + df.format(angle[0] * 180) + " degrees\n";
        msg += "y-axis: " + df.format(angle[1] * 180) + " degrees\n";
        msg += "z-axis: " + df.format(angle[2] * 180) + " degrees\n";
        break;
    ...
}
```

StepCounter

Ein anderes schönes Beispiel für einen Software-Sensor ist der Step-Counter Sensor. Wie üblich registrieren wir den Sensor in der onResume() Methode mittels:

```
Sensor mSensor =
    mSensorManager.getDefaultSensor(
        Sensor.TYPE_STEP_COUNTER);
mSensorManager.registerListener(this, mSensor,
    SensorManager.SENSOR_DELAY_NORMAL);
```

und die gelaufenen Schritte erhalten wir dann ab und zu in der onSensorChanged() Methode:

```
public final void onSensorChanged(
                                SensorEvent event) {
    float steps = event.values[0];
    tv.setText("Nr of steps: " + steps);
}
```

Das Schrittezählen funktioniert aber nur so lange unsere Activity im Vordergrund ist. Sobald die Activity nicht mehr aktiv ist, werden auch keine Schritte mehr gezählt. Später werden wir sehen, wie wir das mit Services, die im Hintergrund laufen, lösen können.

Review

So weit haben wir uns ein bisschen mit Sensoren beschäftigt. Mit dem LocationManager haben wir unsere Position bestimmt. An alle anderen Sensordaten kommen wir über den SensorManager. Als Beispielanwendungen haben wir uns als erstes eine Liste aller vorhandenen Sensoren geben lassen, und danach auf den Lichtsensor und den Beschleunigungssensor zugegriffen. Wir haben ganz kurz über Filter gesprochen, und schließlich die Softwaresensoren angesprochen.

Projekte

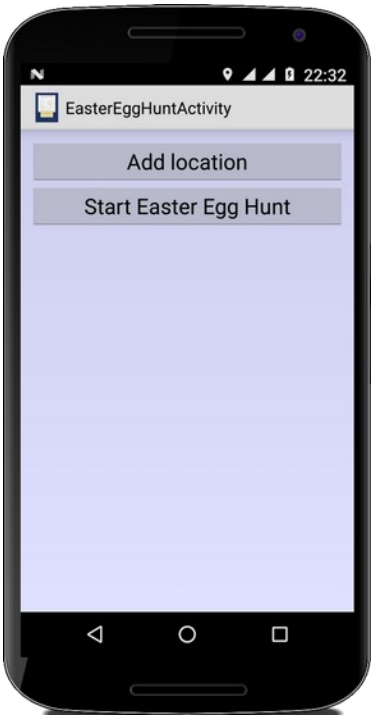
Was kann man mit den Sensoren denn so alles machen? Zunächst können wir an einer kleinen Konkurrenz von Google Maps arbeiten. Auch das Ostereierfinden ist natürlich mit GPS viel einfacher als ohne (speziell wenn man ein großes Grundstück hat). Wie man mit der Batterie Temperatur messen kann ist auch mal interessant zu wissen. Neben Erdbebenmessungen, können wir mit den Sensoren auch Alter und Manneskraft bestimmen. Ein paar Spiele sind auch unterhaltsam, z.B. unseren alten Freund BrickBreaker über Lagesensor steuern. Von Zufallszahlen haben wir schon seit zwei Jahren nichts mehr gehört, wird wieder mal Zeit. Und ein Kompass und ein "Metall"-Detektor bilden dann den Abschluss.

OpenStreetMap

Im ersten Kapitel haben wir ja bereits gesehen, wie man eine geografische Position in OpenStreetMap anzeigen kann, indem man einfach den Längen- und Breitengrad eines Ortes übergibt:

```
Uri uri = Uri.parse(
    "http://www.openstreetmaps.org/?" +
    "lat=49.452&lon=11.082&zoom=20");
Intent intent =
    new Intent(Intent.ACTION_VIEW, uri);
startActivity(intent);
```

Inzwischen wissen wir aber auch wie wir mit dem GPS Sensor die Position unseres Handys bestimmen können. Wenn wir die beiden also kombinieren, können wir ganz einfach unsere momentane Position in OpenStreetMap anzeigen. Zu unserem eigenen Google Maps fehlt jetzt nur noch Dijkstra...



Easter Egg Hunt

Die "Digital Natives" suchen ihre Ostereier natürlich nur noch mit Handy (Karel natürlich auch). Die App besteht aus zwei Modi: dem OsterhasenModus und dem SuchModus.

Der OsterhasenModus ist für den Osterhasen: während er die Eier versteckt, clickt er auf einen Knopf "Add location" jedes Mal wenn er ein Osterei versteckt hat. Dabei wird die momentane Position in einer Liste von Locations gespeichert:

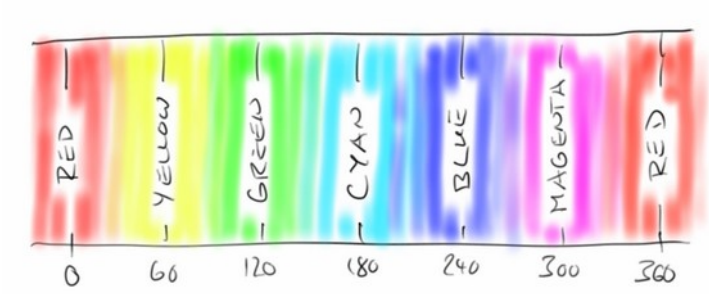
```
private List<Location> locations =
    new ArrayList<Location>();
```

Ist der Osterhase fertig mit dem Verstecken, clickt auf "Start Easter Egg Hunt" und übergibt das Handy den Kids. Die App wechselt dann in den SuchModus, was eigentlich nur bedeutet, dass die Buttons unsichtbar gemacht werden:

```
btnAddLocation.setVisibility(View.INVISIBLE);
btnDone.setVisibility(View.INVISIBLE);
```

Im SuchModus zeigen wir die Entfernung zu den verschiedenen Locations farblich an: blau ist weit weg und rot ist ganz nah:

```
hsv[0] = distanceInMeters * DISTANCE_FACTOR;
ll.setBackgroundColor(Color.HSVToColor(hsv));
```



Am besten verwendet man dazu den Farbton (Hue) [4] des HSV-Farbraum Modells. Die Werte des Farbtons gehen von 0 bis 360.

Wir können also ganz einfach die Distanz mit einer Konstanten (z.B. 6) multiplizieren, um eine Farbe zu erhalten. Das Array *hsv* ist eine Instanzvariable und wird mit den folgenden Werten initialisiert:

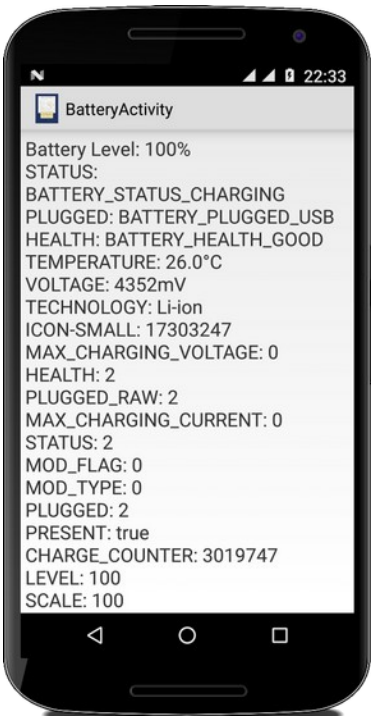
```
private float[] hsv = { 240f, 1f, 1f };
```

Was wir noch wissen müssen ist, wie wir die Distanz zwischen zwei Locations erhalten. Das geht am einfachsten mit der Methode *distanceTo()*:

```
currentEasterEgg = 0;
float distanceInMeters =
    currentLocation.distanceTo(locations.get(currentEasterEgg));
```

Die Instanzvariable *currentEasterEgg* ist ein Zähler, den wir immer um eins erhöhen, wenn ein Osterei gefunden wurde. Wie wissen wir, dass ein Osterei gefunden wurde? Eine Möglichkeit wäre einfach zu sagen wenn die Distanz weniger als fünf Meter ist, dann ist das Osterei gefunden. Das ist zwar nicht ganz realistisch, aber da die Auflösung des GPS Sensors nicht viel besser als ein paar Meter ist, macht alles andere eigentlich keinen Sinn. Happy Easter!

Battery



Die Sensoren für Licht und Proximity haben wir ja schon gesehen. Interessant wäre natürlich noch ein Temperatursensor. Leider haben die wenigsten Geräte einen. Aber wir wären keine angehenden Ingenieure wenn wir so einfach aufgeben würden. Es stellt sich nämlich heraus, dass fast alle Android Geräte eine Batterie haben. Und nahezu alle modernen Batterien haben einen Temperatursensor zwecks Überladeschutz. Wäre doch cool wenn wir die Batterie nach ihrer Temperatur fragen könnten.

Den Zugriff auf die Batteriedaten erhält man über einen Intent:

```
IntentFilter batteryIntentFilter =
   new IntentFilter(Intent.ACTION_BATTERY_CHANGED);
Intent batteryIntent =
   registerReceiver(null, batteryIntentFilter);
```

Dem stellt man einfach Fragen, z.B. nach der Temperatur:

```
int temperature = batteryIntent.getIntExtra(
   BatteryManager.EXTRA_TEMPERATURE, 0);
float celsius = ((float) temperature) / 10;
```

oder der Batteriespannung:

```
int voltage = batteryIntent.getIntExtra(
   BatteryManager.EXTRA_VOLTAGE, 0);
```

Will man alles über die Batterie wissen, dann kann man durch die ganzen Extras iterieren:

```
Bundle extras = batteryIntent.getExtras();
for (String key : extras.keySet()) {
    sb.append(key.toUpperCase() + ": ");
    sb.append(extras.get(key) + "\n");
}
```

Eine kleine Anmerkung, die Temperatur der Batterie ist natürlich immer etwas höher als die Umgebungstemperatur. Denn sowohl beim Entladen (also sobald unser Handy an ist), aber besonders auch beim Aufladen, finden ja chemische Prozesse innerhalb der Batterie statt, die für eine erhöhte Temperatur sorgen.

Eine andere Möglichkeit an Batteriedaten zu kommen ist über den *BatteryManager*. Leider ist der ziemlich nutzlos, denn außer den momentanen Ladezustand, liefert der praktisch keine nützliche Information und er funktioniert erst ab Android Version 21 so richtig:

```
private String getBatteryLevel() {
    String msg;
    BatteryManager bm = (BatteryManager) getSystemService(
                                                    BATTERY_SERVICE);
    if (Build.VERSION.SDK_INT >= 21) {
        int capacity = bm.getIntProperty(
            BatteryManager.BATTERY_PROPERTY_CAPACITY);
        msg = "Battery Level: " + capacity + "%\n";
    } else {
        msg = "Battery Level not supported!\n";
    }
    return msg;
}
```

Earth Quake

Man kann mit Smartphones auch Erdbeben detektieren [5]. Dazu benutzt man einfach den Beschleunigungssensor und sucht nach bestimmten Mustern. Uns genügt aber ein ganz einfacher Erdbebendetektor: sobald die Werte des Beschleunigungssensor einen bestimmten Schwellwert überschreiten, geben wir Alarm mittels eines Toasts.

Als erstes definieren wir den Schwellwert für Erdbeben:

```
private final double EARTHQUAKE_THRESHOLD = 0.1;
```

Außerdem benötigen wir noch eine Instanzvariable für die letzten Werte des Beschleunigungssensors:

```
private double[] accelLast = new double[3];
```

Jetzt müssen wir lediglich in der *onSensorChanged()* Methode feststellen, um wieviel sich die Beschleunigung im Vergleich zum letzten Mal geändert hat:

```
double maxDelta = 0;
for (int i = 0; i < 3; i++) {
    double accel = event.values[i];
    double delta = accelLast[i] - accel;
    accelLast[i] = accel;
    if (delta > maxDelta) {
        maxDelta = delta;
    }
}
```

Wenn diese zu groß ist geben wir Alarm mittels eines Toasts:

```
if (maxDelta > EARTHQUAKE_THRESHOLD) {
    Toast.makeText(this, "Earthquake detected!",
        Toast.LENGTH_SHORT).show();
}
```

Funktioniert aber nur wenn wir unser Smartphone auf festen Untergrund stellen, sonst gibt unsere App die ganze Zeit Alarm.

Age

Wir wollen eine kleine App schreiben mit der man das Alter einer Person bestimmen kann. Oben in unserer Erdbeben App haben wir effektiv einen Hochpass verwendet, also nur hohe Frequenzen durchgelassen, da uns nur schnelle Änderungen interessiert haben. Für unsere Altersbestimmungs App interessiert uns aber eher das durchschnittliche Zittern, wir wollen also auf die Deltas einen Tiefpass anwenden.

Wir holen uns also wie oben die Beschleunigungswerte, aber uns interessiert nur deren absoluter Wert (sonst kommt beim Durchschnittbilden null raus):

```
for (int i = 0; i < 3; i++) {
    double accel = event.values[i];
    double delta = Math.abs(accelLast[i] - accel);
    accelLast[i] = accel;
    // low pass filter
    deltaAvg[i] =
        (ATTENUATION * deltaAvg[i] + delta) /
        (ATTENUATION + 1);
    checkForAge();
}
```

Danach schicken wir die *deltas* durch den Tiefpass-Filter. Ein guter Wert für ATTENUATION ist 10.

Wie machen wir jetzt aus dem Zittern das Alter? Wir multiplizieren einfach mit 100. Nicht sehr wissenschaftlich, aber ist ja nur ne App. Damit die Daten auf dem Display nicht wild hin- und herspringen, macht es evtl. noch Sinn die Anzeige nur einmal die Sekunde upzudaten:

```
private void checkForAge() {
    long thisTime = System.currentTimeMillis();
    if (thisTime - lastTime > DELAY) {
        double maxDeltaAvg =
            Math.max(deltaAvg[2], Math.max(deltaAvg[1], deltaAvg[0]));
        int age = (int) (maxDeltaAvg * 100);
        tv.setText("You are about: " + age + " years old!");
        lastTime = thisTime;
    }
}
```

Ich bin laut meiner App 42, so fühl ich mich auch...

Hau den Lukas

Nach einer wissenschaftlichen Analyse zum Balzgehabe baiuwarischer Ureinwohner [6], kommt man recht schnell auf die Idee für eine App, die sich in Bayern bestimmt gut verkaufen ließe, und mit der man nebenher auch noch den Handyverkauf ankurbeln könnte.

Im Gegensatz zur Erdbeben App, wo wir uns nur für die kleinsten Beschleunigungen interessiert haben, interessieren uns bei der Hau-den-Lukas App nur die größten Beschleunigungen. Dazu berechnen wir den Betrag der momentanen Beschleunigung:

```
double accel =
    Math.sqrt(event.values[0] * event.values[0]
    + event.values[1] * event.values[1]
    + event.values[2] * event.values[2]);
```

Abhängig von der Höhe der Beschleunigung geben wir dann dem Ureinwohner Feedback. Dazu definieren wir zwei Arrays als Instanzvariablen (eine TreeMap mit umgekehrter Sortierfolge wäre natürlich die bessere Datenstruktur),

```
private final int[] accelRequired = { 250, 200, 150, 100, 50, 0 };
private final String[] ranks = {
    "Weltmeister", "Weibaheld", "Haderlump",
    "Anfänger", "G'schaftl Huaba", "Schlappschwanz" };
```

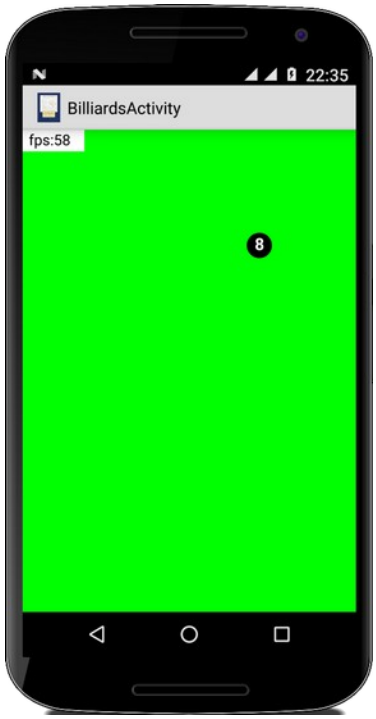
und vergleichen die momentane Beschleunigung *accel* mit den im *accelRequired[]* Array vordefinierten Werten:

```
int i = 0;
for (; i < ranks.length; i++) {
    if (accel > accelRequired[i])
        break;
}
tv.append(ranks[i]);
```

Damit ermitteln wir dann den Titel für unseren Ureinwohner.

Ganz fertig sind wir noch nicht: das Problem ist, dass die Methode *onSensorChanged()* mehrere Male in der Sekunde aufgerufen wird. Deshalb wird unser Rang nur ganz kurz angezeigt, bevor er wieder überschrieben wird. Wir benötigen also noch eine Instanzvariable *oldAccel*, und nur wenn die neue Beschleunigung höher war als die alte, zeigen wir diese an.

Billiards



Wir können Sensordaten natürlich auch für Spiele verwenden. Als Starter nehmen wir unser Billiard Programm aus dem ersten Semester. Die Idee ist, dass wir die Daten vom Beschleunigungssensor nehmen und damit den Ball bewegen.

Da wir in den GraphicsProgrammen die onCreate() ausgelagert haben, ist es am besten den SensorManager in unserer init() Methode zu initialisieren:

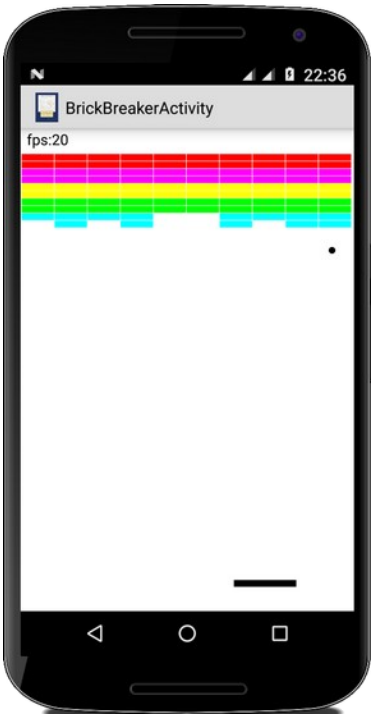
```
private SensorManager mSensorManager;

public void init() {
    mSensorManager = (SensorManager)
        getSystemService(Context.SENSOR_SERVICE);
}
```

Die Methoden onResume(), onPause() und onAccuracyChanged() sind genauso wie in den vorhergehenden Beispielen.

Und wie bewegt sich jetzt der Ball? Das ist überraschend einfach, wir müssen lediglich vx und vy in der onSensorChanged() Methode anpassen:

```
public final void onSensorChanged(
                                SensorEvent event) {
    switch (event.sensor.getType()) {
    case Sensor.TYPE_ACCELEROMETER:
        float[] acceleration_in_m_per_s2 = event.values;
        vx -= acceleration_in_m_per_s2[0];
        vy += acceleration_in_m_per_s2[1];
        break;
    default:
        return;
    }
}
```

BrickBreaker



Auch beim BrickBreaker lässt sich der Beschleunigungssensor verwenden, in diesem Fall als sogenannter Lagesensor. Basierend auf dem Beschleunigungssensor wollen wir, dass sich der Paddle bewegt.

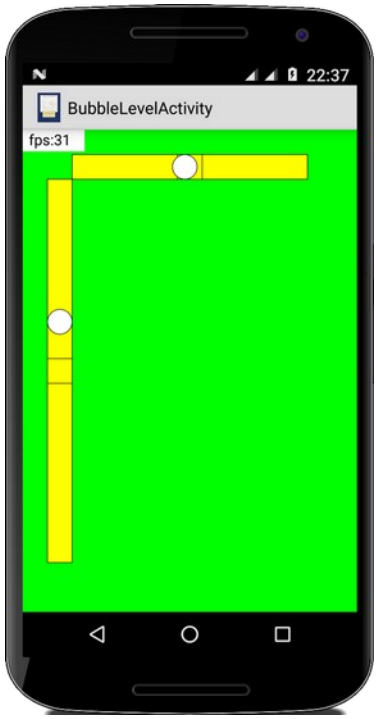
Zunächst machen wir die gleichen Anpassungen wie beim Billiardsbeispiel was den SensorManager und die onResume(), onPause() und onAccuracyChanged() Methoden angeht.

Dann müssen wir uns überlegen wie wir das mit dem Paddle machen. Eine Möglichkeit ist es dem Paddle eine eigene Geschwindigkeit zu geben und basierend auf dieser Geschwindigkeit, das Paddle zu bewegen:

```
private double paddleSpeed = 0;

private void movePaddle() {
    if (paddle != null) {
      double xP = paddle.getX();
      if (xP < 0.0) {
        paddleSpeed = 0;
        paddle.setX(1);
      } else if (xP > getWidth() - 2*BALL_RADIUS) {
        paddleSpeed = 0;
        paddle.setX(getWidth() - 2*BALL_RADIUS -1);
      }
      paddle.move((int) paddleSpeed, 0);
    }
}
```

Dann müssen wir lediglich noch *paddleSpeed* in der onSensorChanged() Methode anpassen:

```
public final void onSensorChanged(SensorEvent event) {
    switch (event.sensor.getType()) {
    case Sensor.TYPE_ACCELEROMETER:
        float[] acceleration_in_m_per_s2 = event.values;
        paddleSpeed -= acceleration_in_m_per_s2[0];
        break;
    default:
        return;
    }
}
```



Es dürfte ziemlich offensichtlich sein, dass wir hiermit eine ganze Klasse von Handyspielen abgedeckt bekommen.

BubbleLevel

Als letzte Anwendung für den Lagesensor schreiben wir eine Wasserwaagen App. Damit das Ganze hübsch aussieht implementieren wir es als GraphicsProgram. Unsere Bubbles sind einfach GOvals, eine für die horizontale und die andere für die vertikale Ausrichtung, die wir jeweils mittig platzieren:

```
private GOval bubbleHor;
private GOval bubbleVer;

private void setup() {
    setBackground(Color.GREEN);

    bubbleHor = new GOval(BUBBLE_SIZE, BUBBLE_SIZE);
    bubbleHor.setFillColor(Color.WHITE);
    bubbleHor.setFilled(true);
    add(bubbleHor, (getWidth() - BUBBLE_SIZE) / 2, PADDING);
```

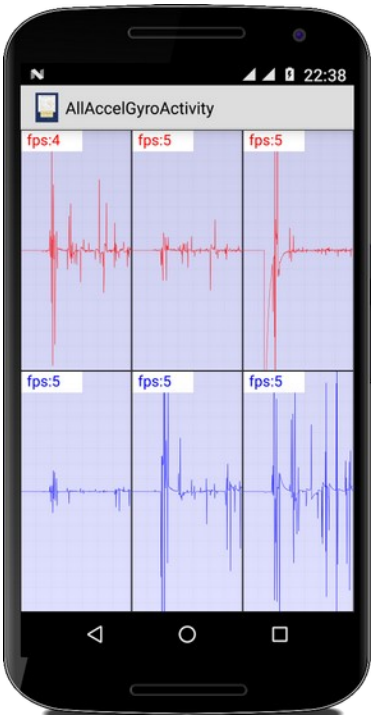
```
        bubbleVer = new GOval(BUBBLE_SIZE, BUBBLE_SIZE);
        bubbleVer.setFillColor(Color.WHITE);
        bubbleVer.setFilled(true);
        add(bubbleVer, PADDING, (getHeight() - BUBBLE_SIZE) / 2);
    }
```

Die Positionierung der Luftblasen erfolgt in der onSensorChanged() Methode:

```
    public final void onSensorChanged(SensorEvent event) {
        switch (event.sensor.getType()) {
        case Sensor.TYPE_ACCELEROMETER:
            float[] acceleration_in_m_per_s2 = event.values;
            if (bubbleHor != null) {
                int x = (int) ((getWidth() - BUBBLE_SIZE) / 2
                    + acceleration_in_m_per_s2[0] * SENSITIVITY);
                bubbleHor.setX(x);
                int y = (int) ((getHeight() - BUBBLE_SIZE) / 2
                    - acceleration_in_m_per_s2[1] * SENSITIVITY);
                bubbleVer.setY(y);
            }
            break;
        default:
            return;
        }
    }
```

Wenn man will, kann man noch im Hintergrund je ein GRect für das Gehäuse der Wasserwaage setzen. (Warum man bei x ein Plus machen muss und bei y ein Minus ist mir auch nicht ganz klar.)

AllAccelGyro



Wenn man sich Gedanken zu einer neuen App macht die Sensordaten verwenden soll, ist es hilfreich, wenn man die Sensordaten vorher mal visualisiert. Das wollen wir jetzt tun: eine App schreiben, welche die Werte des Beschleunigungssensors und des Gyroskops visualisiert.

Zu Beginn registrieren wir in der onResume() Methode Beschleunigungssensor und Gyroskop:

```
    protected void onResume() {
        super.onResume();
        Sensor mAccelerometer =
            mSensorManager.getDefaultSensor(
                Sensor.TYPE_ACCELEROMETER);
        if (mAccelerometer != null) {
            mSensorManager.registerListener(
                this,
                mAccelerometer,
                SensorManager.SENSOR_DELAY_NORMAL);
        }
        Sensor mGyroscope =
            mSensorManager.getDefaultSensor(
                Sensor.TYPE_GYROSCOPE);
        if (mGyroscope != null) {
            mSensorManager.registerListener(
                this,
                mGyroscope,
                SensorManager.SENSOR_DELAY_NORMAL);
        }
    }
```

GraphView

Zum Visualisieren verwenden wir die Klasse *GraphView*. Es handelt sich dabei um einen ganz normalen View, der einen Context als Übergabeparameter im Konstruktor erwartet. Die Verwendung der Klasse ist so einfach wie möglich gehalten:

```
GraphView gv = new GraphView(this);

gv.setStyle(GraphView.GraphStyle.LINE);
gv.setColor(Color.BLUE);
gv.setStrokeWidth(1);

gv.setMin(-0.1f);
gv.setMax(0.1f);
gv.addDataPoint(accelLastAvg[i]);

gv.postInvalidate();
```

Was die verschiedenen Stile angeht, kennt er HISTOGRAM, POINT und LINE. Die Daten werden wie bei einem Oszilloskop angezeigt, d.h. neue Datenpunkte werden von rechts kommend eingefügt.

TableLayout

Für diese App eignet sich das *TableLayout* am besten. Ähnlich wie Tabellen in HTML bestehen die Tabellen im TableLayout aus Reihen, *TableRow*, in die man dann einen oder mehrere Views einfügen kann. Das TableLayout selbst ist ganz einfach zu definieren:

```
TableLayout.LayoutParams tableParams =
    new TableLayout.LayoutParams(
        TableLayout.LayoutParams.MATCH_PARENT,
        TableLayout.LayoutParams.MATCH_PARENT);

TableLayout tl = new TableLayout(this);
tl.setBackgroundColor(0x200000ff);
tl.setLayoutParams(tableParams);
setContentView(tl);
```

Interessanter sind die LayoutParams der TableRow, denn hier können wir die gewünschte Breite und Höhe angeben, die ausnahmsweise sogar respektiert werden:

```
TableRow.LayoutParams rowParams =
    new TableRow.LayoutParams(
        TableRow.LayoutParams.MATCH_PARENT,
        TableRow.LayoutParams.WRAP_CONTENT);
rowParams.width = width / NR_OF_COLUMNS;
rowParams.height = height / NR_OF_ROWS;
```

dabei sind *width* und *height* Instanzvariablen, die wir über die Methode *getScreenSize()* ermittelt haben (Kapitel zwei).

Jetzt können wir die zwei TableRows definieren, in die wir je drei GraphViews einfügen:

```
gvs = new GraphView[6];

TableRow tr1 = new TableRow(this);
for (int i = 0; i < 3; i++) {
    gvs[i] = new GraphView(this);
    gvs[i].setLayoutParams(rowParams);
    tr1.addView(gvs[i]);
}
tl.addView(tr1);
```

```
TableRow tr2 = new TableRow(this);
for (int i = 3; i < 6; i++) {
    gvs[i] = new GraphView(this);
    gvs[i].setLayoutParams(rowParams);
    tr2.addView(gvs[i]);
}
tl.addView(tr2);
```

Sensor Data

Im einfachsten Fall zeigen wir einfach die Rohdaten an:

```
public void onSensorChanged(SensorEvent event) {
    switch (event.sensor.getType()) {
    case Sensor.TYPE_ACCELEROMETER:
        for (int i = 0; i < 3; i++) {
            double accel = event.values[i];
            gvs[i].addDataPoint(accel);
            gvs[i].postInvalidate();
        }
        break;
    case Sensor.TYPE_GYROSCOPE:
        for (int i = 0; i < 3; i++) {
            double gyro = event.values[i];
            gvs[i + 3].addDataPoint(gyro);
            gvs[i + 3].postInvalidate();
        }
        break;
    default:
        return;
    }
}
```

Wenn wir aber auf den dritten Beschleunigungssensor achten, stellen wir fest, dass der einen Wert von 9.8 hat, und deswegen gar nicht angezeigt wird. Hier gibt es zwei Möglichkeiten: entweder Minimum und Maximum im GraphView für diesen Sensor zu ändern, oder einen Hochpass als Filter einzusetzen.

RandomGenerator

Am Anfang des zweiten Buchs haben wir ein bisschen was über Pseudo-Zufallszahlen gelernt und dabei auch den Lehmer Algorithmus kennen gelernt. Alle Pseudo-Zufallszahlen Algorithmen haben das gleiche Problem: kann man die Seed erraten, also den ersten Wert, dann kann man alle folgenden Zahlen berechnen. Da in der Regel die Uhrzeit als Seed verwendet wird, gelingt es der NSA und ähnlichen Organisationen sehr häufig die meisten Verschlüsselungen zu knacken.

Da wir aber inzwischen auf die Sensordaten unseres Smartphones zugreifen können, können wir der NSA ein Schnippchen schlagen: wir benutzen einfach den Beschleunigungssensor und das Gyroskop um die Seed zu generieren. Vorab sollten wir aber noch zwei Dinge beachten: Natürlich sollte unser Smartphone nicht starr rumliegt, sondern ein bisschen bewegen sollte es sich schon. Und was den z-Beschleunigungssensor angeht, wird der immer mit einer '9' beginnen (es sei denn wir sind auf dem Mond oder Mars).

Für unsere Anwendung definieren wir zwei Instanzvariablen:

```
private float[] acceleration_in_m_per_s2;
private float[] rotation_in_rad_per_s;
```

Die werden in der *onSensorChanged()* Methode immer auf die aktuellen Werte gesetzt.

In der onClick() Methode des "Init Seed from Sensors" Buttons erzeugen wir dann eine *seedA* aus den Beschleunigungsdaten

```
float seedA = acceleration_in_m_per_s2[0];
seedA += acceleration_in_m_per_s2[1];
// gravity is to predictable:
float gravity = acceleration_in_m_per_s2[2];
// get rid of first 3 digits
gravity = gravity * 100 - (int) (gravity * 100);
seedA += gravity;
```

und eine *seedB* aus den Gyroskopdaten

```
float seedB = rotation_in_rad_per_s[0];
seedB += rotation_in_rad_per_s[1];
seedB += rotation_in_rad_per_s[2];
```

Von den beiden Seeds nehmen wir dann noch den absoluten Wert

```
seedA = Math.abs(seedA);
seedB = Math.abs(seedB);
```

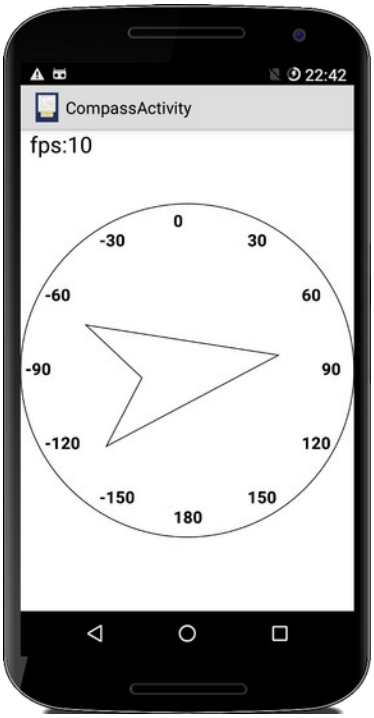
Wir sollten darauf achten, dass beide Seeds größer als Null sind, denn ansonsten deutet das auf ein Problem mit den Sensordaten hin:

```
if (seedA > 0 && seedB > 0) {
    // get exponent:
    int expA = (int) Math.log10(seedA);
    int expB = (int) Math.log10(seedB);
    seedA = (float) (seedA * Math.pow(10, -expA));
    seedB = (float) (seedB * Math.pow(10, -expB));
    Log.i("seedA: ", "" + seedA);
    Log.i("seedB: ", "" + seedB);

    // now make them big
    long seed = (long) (seedB * 100000 * 100000)
        + (long) (seedA * 100000);

    rgen.setSeed((int) seed);
    tv.setText("" + seed);

} else {
    Toast.makeText(v.getContext(), "Bad seeds, try again!",
        Toast.LENGTH_SHORT).show();
}
```

Der Trick mit dem Logarithmus führt dazu, dass führende Nullen verschwinden. Die enthalten keinerlei Information und sind eher schädlich. Dann kombinieren wir die beiden Seeds und benutzen sie als Eingangsseed für unseren Lehmeralgorithmus.

Man könnte jetzt der Versuchung erliegen alle Zufallszahlen mittels Sensordaten erzeugen zu lassen und auf den Lehmer Algorithmus komplett zu verzichten. Davon würde aber jeder abraten, der ein Studium der Mathematik hinter sich hat.

Compass



Überraschend wenige Smartphones haben einen Magnetfeldsensor, speziell im unteren Preissegment. Sollte unser Smartphone aber einen haben, dann können wir diesen natürlich verwenden um einen Kompass zu programmieren.

Um uns nicht allzu viel Arbeit mit der UI zu machen, setzen wir auf Reuse: für die analoge Gradanzeige verwenden wir unser Projekt *AnalogClock* aus dem ersten Semester, wo wir einfach die Uhrzeiten durch Gradangaben ersetzen:

```
private void drawFace() {
    face = new GOval(SIZE, SIZE);
    add(face, 0, OFFSET);
}

private void drawAngles() {
    for (int i = -5; i <= 6; i++) {
        GLabel digit = new GLabel("" + i * 30);
        digit.setFont("Times New Roman-bold-"
            + FONT_SIZE);
        double radians = 2 * Math.PI * i / 12;
        double radius = SIZE - 80;
        double x = -30 + SIZE / 2
            + Math.sin(radians) * radius / 2;
        double y = 10 + SIZE / 2
            - Math.cos(radians) * radius / 2;
        add(digit, x, y + OFFSET);
    }
}
```

Für unsere Kompassnadel verwenden wir unser geliebtes *GSpaceShip*, nur ein bisschen größer:

```
private void drawSpaceShip(float degrees) {
    if (face != null) {
        if (ship != null) {
            remove(ship);
        }
        ship = new GSpaceShip();
        ship.rotate(-degrees);
        add(ship, SIZE / 2, SIZE / 2 + OFFSET);
    }
}
```

Kommen wir zum Sensor: wir könnten zwar den Hardwaresensor *TYPE_MAGNETIC_FIELD* verwenden, allerdings viel einfacher ist der TYPE_ROTATION_VECTOR Sensor. Das ist ein Softwaresensor, der eine Kombination aus drei Hardwaresensoren ist und den Vorteil hat, dass zum einen die Schwerkraft verwendet wird um die z-Achse festzulegen, und zum anderen kleinere Schwankungen mittels Kalmanfilter bereinigt wurden.

Wir registrieren also den TYPE_ROTATION_VECTOR Sensor in der onResume() Methode:

```
protected void onResume() {
    super.onResume();
    Sensor mMagnet = mSensorManager.getDefaultSensor(
                                        Sensor.TYPE_ROTATION_VECTOR);
    if (mMagnet != null) {
        mSensorManager.registerListener(this, mMagnet,
            SensorManager.SENSOR_DELAY_NORMAL);
    } else {
        Toast.makeText(this, "Your device has no magnetic sensors!",
            Toast.LENGTH_SHORT).show();
    }
}
```

und in der *onSensorChanged()* Methode zeichnen wir dann die Kompassnadel:

```
public void onSensorChanged(SensorEvent event) {
    float degrees = event.values[2] * 180;
    Log.i("degrees=", "" + degrees);
    drawSpaceShip(degrees);
}
```

Ein Kompass mit Digitalanzeige wäre natürlich viel einfacher gewesen.

MetalDetector

Eine Anwendung bei der wir den Hardwaresensor *TYPE_MAGNETIC_FIELD* direkt verwenden können ist ein Metalldetektor, genauer ein Eisendetektor (oder was sonst noch so magnetische Felder verursacht, wie z.B. Gleichstrom). Wir registrieren unseren Magnetfeld-Sensor in der onResume() Methode:

```
protected void onResume() {
    super.onResume();

    Sensor mMagnet =
        mSensorManager.getDefaultSensor(
            Sensor.TYPE_MAGNETIC_FIELD);
    if (mMagnet != null) {
        mSensorManager.registerListener(
            this,
            mMagnet,
            SensorManager.SENSOR_DELAY_NORMAL);
    } else {
        Toast.makeText(this,
            "Your device has no magnetic sensors!",
            Toast.LENGTH_SHORT).show();
    }
}
```

Bei der Auswertung der Sensordaten, müssen wir das Erdmagnetfeld beachten, dass ja immer vorhanden ist. Um es los zu werden, können wir einen Hochpass verwenden. Zusätzlich interessieren uns ganz schnelle Änderungen eigentlich auch nicht, deswegen verwenden wir zusätzlich auch noch einen Tiefpass. Natürlich müssen wir bei der Wahl unserer beiden Konstanten etwas vorsichtig sein, denn sonst kommen gar keine Daten mehr durch. Aber mit den folgenden beiden Werten

```
private final float LOW_PASS_FACTOR = 0.8f;
private final int TIME_RESOLUTION = 100;
```

scheint es ganz gut zu funktionieren. Die onSensorChanged() Methode sieht dann wie folgt aus:

```
public final void onSensorChanged(SensorEvent event) {
    long currentTime = System.currentTimeMillis();

    if (currentTime - lastTime > TIME_RESOLUTION) {
        lastTime = currentTime;

        float[] field_strength_in_micro_tesla = event.values;

        // low-pass
        for (int i = 0; i < 3; i++) {
          magneticBackground[i] = lowPass(
            field_strength_in_micro_tesla[i], magneticBackground[i]);
        }
```

```
            // high-pass filter
            float[] highPass = new float[3];
            for (int i = 0; i < 3; i++) {
              highPass[i] = highPass(
                field_strength_in_micro_tesla[i], magneticBackground[i]);
            }

            msg = "x: " + df.format(highPass[0]) + "\n";
            msg += "y: " + df.format(highPass[1]) + "\n";
            msg += "z: " + df.format(highPass[2]) + "\n";
            TextView tv = (TextView) findViewById(R.id.textview);
            tv.setText(msg);
        }
    }
```

Wobei *lastTime* und *magneticBackground* zwei Instanzvariablen sind. Stellt sich die Frage, könnte man mit einem 50 Hz Filter auch Wechselstrom nachweisen?

Challenges

Spy

Eine interessante Beobachtung ist, dass man auf die Sensordaten ohne besondere Berechtigungen zugreifen darf. Anders ausgedrückt, wir als Nutzer unseres Phones, können einer Anwendung nicht verbieten auf die Sensoren zu zugreifen (ausgenommen natürlich der GPS Sensor). Man könnte nun meinen, dass sei ja nicht weiter schlimm, was kann man denn mit den Sensordaten schon anfangen? Nun, Forscher der Stanford University haben gezeigt [7], dass man das Gyroskop oder den Beschleunigungssensor effektiv als Mikrofon für Frequenzen von unterhalb 200 Hz verwenden kann. Verschwörungstheoretiker würden das als Beweis werten, dass Google von NSA Mitarbeitern unterwandert ist. Uns ist das aber egal, wir wollen das einfach mal in Aktion sehen.

Zunächst sollten wir den Beschleunigungssensor registrieren:

```
protected void onResume() {
    super.onResume();

    mSensorManager.registerListener(this,
        mSensorManager.getDefaultSensor(
            Sensor.TYPE_ACCELEROMETER),
        SensorManager.SENSOR_DELAY_FASTEST);
}
```

(auf manchen Geräten ist das Gyroskop besser geeignet). In der UI gibt es zwei Knöpfe, einen zum Aufnehmen und einen zum Abspielen:

```
btnAccel = (Button) findViewById(R.id.btnAccel);
btnAccel.setOnClickListener(new OnClickListener() {

    @Override
    public void onClick(View v) {
        rawPointer = 0;
        btnAccel.setEnabled(false);
    }
});
```

```
btnAgain = (Button) findViewById(R.id.btnAgain);
btnAgain.setOnClickListener(new OnClickListener() {

    @Override
    public void onClick(View v) {
        getMinMax();
        short[] buffer = convertToAudio();
        playAudio(buffer);
    }
});
```

Die Aufnahme findet in der onSensorChanged() Methode statt:

```
private int rawPointer = TIME * FREQUENCY / SCALE;
private float[] rawData = new float[TIME * FREQUENCY / SCALE];

public void onSensorChanged(SensorEvent event) {
    if (rawPointer < TIME * FREQUENCY / SCALE) {
        float[] values = event.values;
        float delta = 0;
        for (int i = 0; i < 3; i++) {
            delta += values[i] - oldValues[i];
            oldValues[i] = values[i];
        }
        // we do not want the first value:
        if (delta < 9) {
            rawData[rawPointer] = delta;
            rawPointer++;
        }

    } else {
        getMinMax();
        mTextView1.setText("size: " + rawData.length
            + ", min=" + min + " ,max=" + max);
        btnAccel.setEnabled(true);
    }
}
```

Die Sensordaten werden in dem Instanzarray *rawData* gespeichert. Der *rawPointer* ist ein Pointer in das *rawData* Array. Wenn wir am Ende des Arrays angelangt sind, dann hört die Aufnahme auf. Die Länge der Aufnahme wird durch die drei Konstanten

```
private final int SCALE = 80;    // should be a gcd of FREQUENCY
private final int FREQUENCY = 8000;
private final int TIME = 2; // seconds
```

definiert. Dabei sollte FREQUENCY nicht verändert werden. TIME ist ungefähr die Länge der Aufnahme in Sekunden, und SCALE hat einen Wert irgendwo zwischen 10 und 100, abhängig davon wie schnell das Gerät die Sensordaten liefert.

Die Methode *getMinMax()* wird benötigt um die Sensordaten zu skalieren, es findet einfach den kleinsten und den größten Wert in dem Sensordatenarray:

```
private float min, max;
private void getMinMax() {
    min = Float.MAX_VALUE;
    max = Float.MIN_VALUE;
    for (int i = 0; i < rawData.length; i++) {
        float value = rawData[i];
        if (value > max) {
            max = value;
        }
```

```
            if (value < min) {
                min = value;
            }
        }
    }
```

Damit lassen sich die Sensordaten dann in Audiodaten konvertieren,

```
    private short[] convertToAudio() {
        short[] buffer = new short[TIME * FREQUENCY];
        float maxMinusMin = max - min;
        for (int i = 0; i < TIME * FREQUENCY; i++) {
            // scale between 0 and 1:
            float v = (rawData[i / SCALE] - min) / maxMinusMin;
            // scale to max range of short:
            short vv = (short) (2 * Short.MAX_VALUE * (v - 0.5));
            buffer[i] = vv;
        }
        return buffer;
    }
```

die wir dann ganz einfach abspielen können:

```
    private void playAudio(short[] buffer) {
        int audioLength = buffer.length;
        AudioTrack audioTrack = new AudioTrack(AudioManager.STREAM_MUSIC,
    FREQUENCY,
                AudioFormat.CHANNEL_CONFIGURATION_MONO,
    AudioFormat.ENCODING_PCM_16BIT, audioLength,
                AudioTrack.MODE_STREAM);
        audioTrack.play();
        audioTrack.write(buffer, 0, audioLength);
    }
```

Unsere App ist natürlich nicht perfekt, für Stimme oder Musik ist sie nicht geeignet. Wenn wir aber Morse-artig auf den Tisch klopfen (und das Gerät auf dem Tisch liegt), dann kann man das Klopfen hören. Wir haben also ein Mikrofon für niederfrequente Geräusche. Übrigens ist das auch der Grund warum man seine Festplatte nicht anschreien soll [8].

RotationVectorDemo (TYPE_GAME_ROTATION_VECTOR)

Wir haben den Softwaresensor TYPE_GAME_ROTATION_VECTOR noch nicht angesprochen. Im Gegensatz zum Sensor TYPE_ROTATION_VECTOR kommt er ohne Magnetfeldsensor aus, funktioniert also auf viel mehr Endgeräten. Einen kleinen Nachteil hat er aber schon, es gibt immer einen kleinen Drift, abhängig vom Gerät.

Ein wunderschönes Beispiel von Google, die RotationVectorDemo [9] zeigt sehr schön visuell wie man den Sensor benutzt und was mit dem Drift gemeint ist. Als kleines Zusatzfeature sieht man auch wie man mit OpenGL in Android arbeiten kann.

Research

In diesem Kapitel gibt es noch zwei interessante Themen, die man noch etwas vertiefen könnte.

Sensor Fusion

Wir haben den Begriff "Sensor Fusion" ein paar mal erwähnt. Dazu gibt es einen sehr interessanten GoogleTechTalk von David Sachs [1], den man sich mal anschauen sollte. Überhaupt gibt es ne ganze Menge cooler GoogleTechTalks.

Kalman Filter

"Sensor Fusion" verwendet unter anderem Kalman Filter. Als angehende Ingenieurin schadet es nichts mal den Artikel in der Wikipedia zu überfliegen [2].

Fragen

1. Auf den meisten Android Geräten gibt es mehrere Möglichkeiten um die eigene Position (location) zu bestimmen, manche mit höherer andere mit geringerer Auflösung. Nennen Sie drei verschiedene Möglichkeiten.

2. Benötigt man besondere Erlaubnisse (permissions) um auf die Positionsdaten (location) eines Gerätes zugreifen zu können, oder anders ausgedrückt, bedarf es besonderer Permissions wenn die Applikation auf dem Gerät installiert wird?

3. Warum ist der Dalvik Debug Monitor Server (DDMS) wichtig wenn man Apps entwickelt, die den GPS Sensor benötigen?

4. Welche Sensoren findet man auf den meisten normalen Android Geräten? (Nennen Sie mindestens 5)

5. Was versteht man unter dem Begriff Sensor Fusion, und warum ist Sensor Fusion relevant für Android Entwickler?

6. Filter werden sehr häufig zusammen mit Sensoren verwendet, dabei kommen vor allem der Hochpass- und der Tiefpassfilter vor. Nennen Sie ein Beispiel wann Sie einen Hochpass- und wann Sie einen Tiefpassfilter einsetzen würden.

7. Überlegen Sie sich eine Beispielapp die Sie mit dem Beschleunigungssensor entwickeln könnten.

Referenzen

[1] David Sachs, GoogleTechTalks, Sensor Fusion on Android Devices: A Revolution in Motion Processing, https://www.youtube.com/watch?v=C7JQ7Rpwn2k

[2] Kalman filter, https://en.wikipedia.org/wiki/Kalman_filter

[3] Global Positioning System, en.wikipedia.org/wiki/Global_Positioning_System

[4] Hue, https://en.wikipedia.org/wiki/Hue

[5] Qingkai Kong, et.al., Smartphone-based Networks for Earthquake Detection, https://pdfs.semanticscholar.org/554a/f252d80c8cefc5de0a8f511a3427f8b9bcb9.pdf

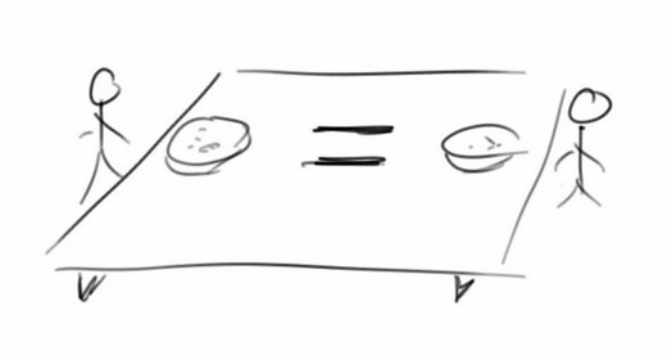
[6] Hau den Lukas auf der Wiesn, https://www.youtube.com/watch?v=_AOdyo3tCwc

[7] Michalevsky Y., Boneh D., and Nakibly G., Gyrophone: Recognizing Speech From Gyroscope Signals, https://crypto.stanford.edu/gyrophone/files/gyromic.pdf

[8] Bryan Cantrill, Shouting in the Datacenter, https://www.youtube.com/watch?v=tDacjrSCeq4

[9] RotationVectorDemo.java, https://android.googlesource.com/platform/development/+/master/samples/ApiDemos/src/com/example/android/apis/os/RotationVectorDemo.java

Concurrency



Auf unseren Computern und Telefonen sieht es so aus, als ob viele Dinge gleichzeitig passieren, mehrere geöffnete Fenster oder Aktivitäten, im Hintergrund laufende Dienste, gleichzeitige Downloads, etc. Es ist für uns so natürlich, dass wir es nicht einmal bemerken. Moderne Prozessoren haben mehrere Kerne, 4 bis 8 ist heutzutage die Norm, und die Zahl steigt mit jeder neuen Prozessorgeneration. Noch extremer ist das bei Grafikkarten bei denen die GPUs sogar mehrere tausend "Kerne", sogenannte Shader, haben. Die Frage die sich für uns stellt ist, wie nutzen wir diese zusätzlichen Kerne?

In unserer ganz normalen Java Anwendung laufen bereits mehrere Threads ohne unser Zutun: In einem einfachen Java-Konsolenprogramm gibt es mindestens den Haupt-Thread und den Garbage-Collector-Thread. Für eine typische Java-UI-Anwendung mit Swing haben wir zusätzlich den UI-Thread und einen Thread zur Verwaltung der Events. Diese "Standard Threads" merken wir normalerweise nicht einmal (es sei denn, sie tun nicht, was sie sollen).

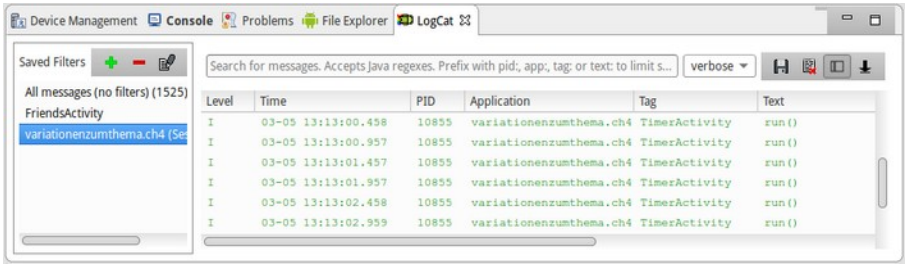
In diesem Kapitel geht es darum zu lernen, wie wir unsere eigenen Threads schreiben können und wie man allgemeine Fallstricke vermeidet. Die Multi-Thread-Programmierung ist nicht ganz trivial wie wir gleich sehen werden.

Timer and TimerTask

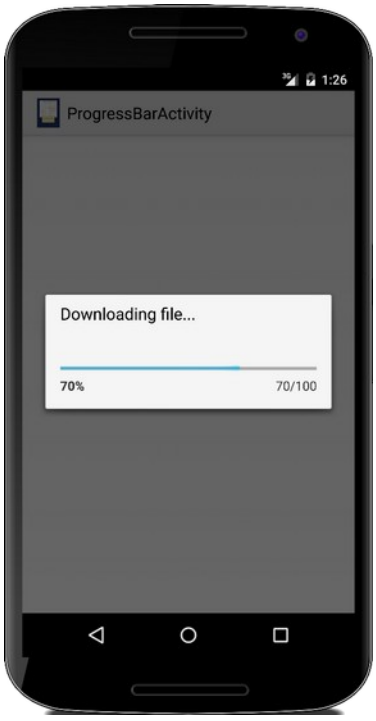
Der einfachste Weg, um ein paar einfache Aufgaben im Hintergrund zu erledigen, ist mit Androids *Timer* und *TimerTask* Klassen. Grundsätzlich sagt der TimerTask, was zu tun ist und der Timer sagt, wann es zu tun ist. D.h. wir verwenden den Timer, um einen TimerTask zu starten. Der folgende Code schreibt alle zwei Sekunden eine kleine Meldung in die Logdatei:

```
public class TimerActivity extends Activity {

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        long delay = 3000;  // delay (ms) before task is executed
        long period = 2000; // time (ms) between successive executions
        Timer timer = new Timer();
        timer.schedule(new TimerTask() {
            @Override
            public void run() {
                Log.i("TimerActivity", "run()");
            }
        }, delay, period);

        pause(20000); // wait 20 secs
        timer.cancel();
    }
}
```

Es ist interessant festzustellen, dass der Task weiterläuft, auch wenn wir die TimerActivity in den Hintergrund schicken. Erst beim Beenden der Application wird der Task beendet. Im LogCat sehen wir dass der Task kontinuierlich weiterläuft:



Wenn eine Anwendung läuft, verbraucht sie Strom. Das gilt auch für die TimerActivity, selbst wenn sie im Hintergrund läuft. Wir sollten daher in der onPause() Methode dafür sorgen, dass evtl. laufende Threads angehalten werden, um die Batterie unseres Nutzers zu schonen. Für den Timer macht das die *cancel()* Methode.



AsyncTask

Der TimerTask ist einfach zu handhaben und tut was er soll. Allerdings, darf man aus der run() Methode des TimerTasks nicht auf UI Elemente zugreifen. Warum, sei dahin gestellt, wichtig ist, dass der große Bruder des TimerTasks, der *AsyncTask* das darf.

Ein klein wenig komplizierter ist der AsyncTask schon. Aber wir haben eben den Vorteil, dass wir auch auf die Benutzeroberfläche zugreifen können. Eine typische Anwendung ist z.B. ein Fortschrittsbalken. Wir beginnen mit einem sehr einfachen Beispiel, das lediglich aus einem ProgressDialog besteht:

```
public class ProgressBarActivity extends Activity {
    private ProgressDialog progressDialog;
    private Context context;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        this.context = this;

        progressDialog = new ProgressDialog(this);
        progressDialog.setCancelable(true);
        progressDialog.setMessage("Downloading file...");
        progressDialog.setProgressStyle(
            ProgressDialog.STYLE_HORIZONTAL);
        progressDialog.setProgress(0);
        progressDialog.setMax(100);
        progressDialog.show();

        new DownloadTask().execute("I am the parameter");
    }
    ...
}
```

Um den AsyncTask zu starten, rufen wir einfach seine *execute()* Methode auf, mit oder ohne Parametern. In unserem Beispiel simulieren wir einen Download, der nur etwas Zeit verschwendet:

```
private class DownloadTask extends AsyncTask<String, Integer, String>{

    @Override
    protected void onPreExecute() {
        Log.i("DownloadTask.onPreExecute()", "starting...");
    }

    @Override
    protected String doInBackground(String... params) {
        String response = params[0] + " - I am the return value.";
        for (int i = 1; i <= 10; i++) {
            try {
                Thread.sleep(1000);
                publishProgress(i * 10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        return response;
    }

    @Override
    protected void onProgressUpdate(Integer... values) {
        Log.i("DownloadTask.onProgressUpdate()",
              "percent=" + values[0]);
        progressDialog.setProgress(values[0]);
    }

    @Override
    protected void onPostExecute(String result) {
        Log.i("DownloadTask.onPostExecute()",
              "done: result=" + result);
        progressDialog.dismiss();
        Toast.makeText(context, result, Toast.LENGTH_SHORT).show();
    }
}
```

Der Syntax von AsyncTask ist etwas gewöhnungsbedürftig. Zunächst fällt die spitze Klammer mit den Datenypen String, Integer und String auf: diese geben an, welche Datentypen als Parameter und Rückgabewerte in den Methoden *doInBackground(), onProgressUpdate()* und *onPostExecute()* verwendet werden.

Der erste, in diesem Fall ein String, bezieht sich auf den Typ des Parametertyps der Methode doInBackground(). Dies ist auch der gleiche der an die execute() Methode übergeben wird, die von der Activity aus aufgerufen wird. Der zweite, in diesem Fall ein Integer, ist der Parametertyp der Methode onProgressUpdate(). Und der letzte, in diesem Fall ein String, ist der Parametertyp der Methode onPostExecute(). Das klingt komplizierter als es ist.

Wenn wir also die execute() Methode in der Activity aufrufen, wird ein neuer Task gestartet und dann dessen doInBackground() Methode aufgerufen. Hier läuft der Task als eigener Thread. Solange wir die doInBackground() Methode nicht verlassen, lebt der Task. Verlässt der Task aber die doInBackground() Methode, dann wird automatisch die onPostExecute() Methode aufgerufen, und das war's.

Aber der Sinn der Übung war ja ursrpünglich, dass wir den Fortschrittsbalken im UI-Thread verändern können: das geschieht mit der Methode *publishProgress()*. Die rufen wir in der doInBackground() Methode auf. Die publishProgress() Methode ruft indirekt die onProgressUpdate() auf, und die wiederum darf auf den UI-Thread zugreifen.

Also im AsyncTask dürfen sowohl die onProgressUpdate() als auch die onPostExecute() auf den UI-Thread zugreifen. Allerdings die doInBackground() darf das nicht, dort passiert aber in der Regel die ganze Arbeit.

Threads

Wenn weder TimerTask noch AsyncTask unser Problem lösen, müssen wir zu schwereren Waffen greifen: den Threads. Im Prinzip verwenden sowohl der TimerTask als auch der AsyncTask Threads, nur wir sehen das nicht.

Es gibt zwei Möglichkeiten aus einer beliebigen Klasse einen Thread zu machen. Die erste ist einfach von der *Thread* Klasse zu vererben:

```
public class MyFirstThread extends Thread {

    public void run() {
        ...
    }
}
```

Die zweite ist das *Runnable* Interface zu implementieren:

```
public class MySecondThread implements Runnable {

    public void run() {
        ...
    }
}
```

In beiden Fällen müssen wir die *run()* Methode überschreiben, und ja das ist die gleiche *run()* Methode wie wir sie seit Karel kennen. Solange wir uns innerhalb der run() Methode befinden, lebt unser Thread. Sobald wir die run() Methode verlassen, ist er tot.

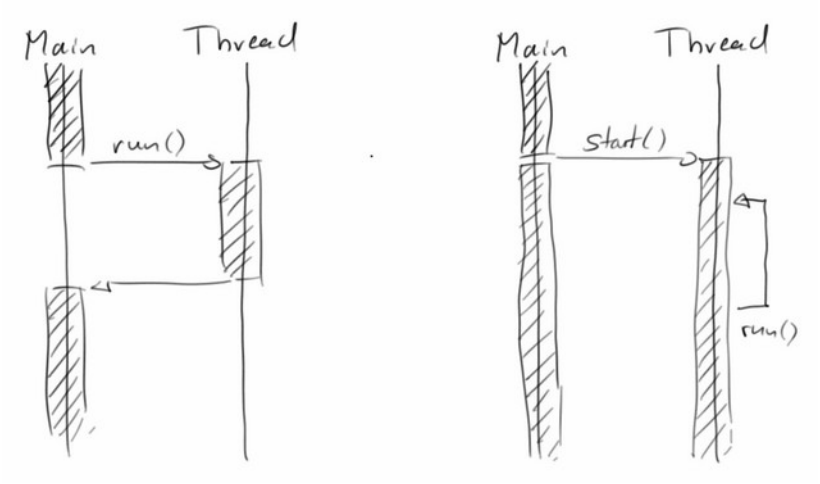
Wie der Mensch durchläuft das Leben eines Threads drei Phasen, er wird geboren, er lebt und er stirbt irgendwann. Und ähnlich wie bei Menschen, wenn man einmal tot ist, war's das. Soll heißen, man kann Threads nicht wiederbeleben.

Was wir allerdings noch nicht geklärt haben: wie werden denn Threads geboren? Das macht die *start()* Methode:

```
MyFirstThread th1 = new MyFirstThread();
th1.start();

MySecondThread mst = new MySecondThread();
Thread th2 = new Thread(mst);
th2.start();
```

Im ersten Fall, wo wir vererben, können wir einfach die start() Methode direkt aufrufen. Im zweiten Fall, übergeben wir das Runnable Objekt an ein Thread Objekt, und starten dieses dann.

Der Unterschied zwischen dem Aufruf der start()-Methode und dem Aufruf der run()-Methode kann mit einem Sequenzdiagramm schön visualisiert werden:



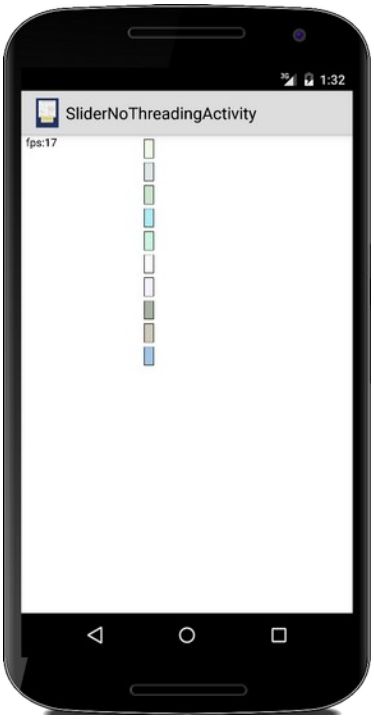
Zu Lebzeiten können sich Threads in verschiedenen Zuständen befinden:

- Running
- Waiting, Sleeping, Blocked
- Ready to run

Der "Thread Scheduler" ist für die Verwaltung der Threads zuständig. Er sorgt dafür, dass jeder Thread mal zum Laufen kommt, und managt auch die Zustände der Threads. Ist natürlich selbst auch ein Thread.

SliderNoThreadingActivity

Sehen wir uns mal ein einfaches Beispiel an. Dazu betrachten wir erst einmal die Klasse *Slider*, bei der es sich einfach um ein GRect handelt:




```
private class Slider extends GRect {

    public Slider(int size, int color) {
        super(size / 2, size);
        setFilled(true);
        setFillColor(color);
    }
}
```

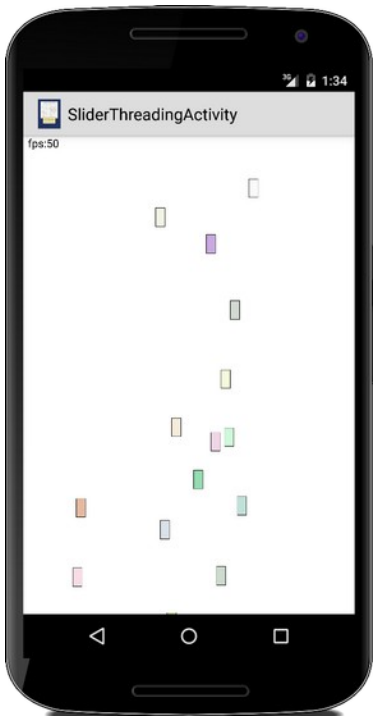
Als nächstes erstellen wir zehn von diesen Sliders und lassen sie einfach von links nach rechts über den Bildschirm gleiten. Bisher haben wir das immer mit unserem *game loop* gemacht:

```
public class SliderNoThreadingActivity extends GraphicsProgram {
    ...

    public void run() {
        Slider[] sliders = createSliders();

        // game loop
        while (true) {
            for (int i = 0; i < sliders.length; i++) {
                sliders[i].move(STEP, 0);
            }
            pause(DELAY);
        }
    }
    ...
}
```

Wir haben also eine riesige Schleife, in der wir jedem der Slider sagen, dass er sich bewegen soll. Dann warten wir ein wenig und wiederholen das Ganze. Das ist das einfache Leben, wir haben nur einen Thread.

SliderThreadingActivity

Im obigen Beispiel sind die Sliders ziemlich dumm. Man muss ihnen sagen was sie tun sollen, nämlich sich zu bewegen. Wäre es nicht cool, wenn die Slider so klug wären, sich selbst zu bewegen? Um unsere Slider schlau zu machen, müssen wir einfach nur das *Runnable* Interface implementieren:




```
private class Slider extends GRect
                     implements Runnable {

    public Slider(int size, int color) {
        super(size / 2, size);
        setFilled(true);
        setFillColor(color);
    }

    public void run() {
        while (true) {
            pause(DELAY);
            move(STEP, 0);
        }
    }
}
```

Sobald wir sagen, dass unsere Klasse das Runnable Interface implementiert, müssen wir auch die *run()* Methode überschreiben, die zu diesem Interface gehört. In der run() Methode, haben wir eine Schleife, die unserem GameLoop sehr ähnlich sieht. Jeder Slider bewegt sich jetzt von ganz alleine.

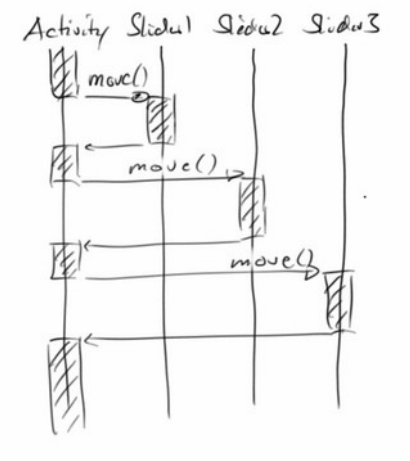
Aber, wie machen wir die Sliders lebendig? Hier kommt die Thread Klasse ins Spiel:

```
public class SliderThreadingActivity extends GraphicsProgram {

    private final int SLIDER_SIZE = 80;
    private final int DELAY = 40;
    private final int STEP = 5;

    private RandomGenerator rgen = new RandomGenerator();

    public void mousePressed(int x, int y) {
        // create a new slider
        Slider slider = new Slider(SLIDER_SIZE, rgen.nextColor());
        add(slider, 0, rgen.nextDouble(0, getHeight()));

        // run the slider in a new Thread
        Thread sliderThread = new Thread(slider);
        sliderThread.start();
    }
    ...
}
```

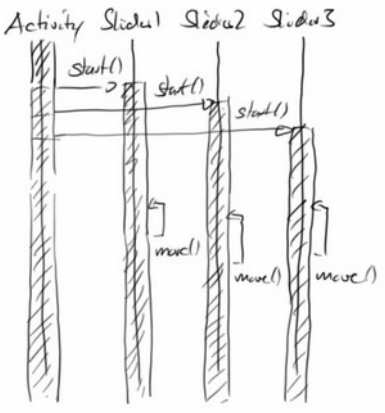
Wir übergeben eines der Runnable Objekte (also einen Slider) an den Konstruktor der Thread Klasse und rufen dann einfach dessen start() Methode auf. Danach wird der Thread zum Leben erweckt und macht sein eigenes Ding. Ist ziemlich einfach.

Sequence Diagram

Um den feinen Unterschied zwischen den beiden Ansätzen oben zu sehen, ist ein Sequenzdiagramm sehr hilfreich. Im Sequenzdiagramm werden die Objekte nach rechts und die Zeit nach unten angetragen. Dabei sieht man wie die verschiedenen Objekte untereinander kommunizieren. Im Beispiel links sehen wir einen dicken Balken der den momentan laufenden Thread anzeigt, den einzigen Thread. Im Beispiel rechts dagegen gibt es mehrere dicke Balken, jedes Mal, wenn ein neuer Thread erstellt wird, fügen wir einen neuen hinzu.



Single Thread

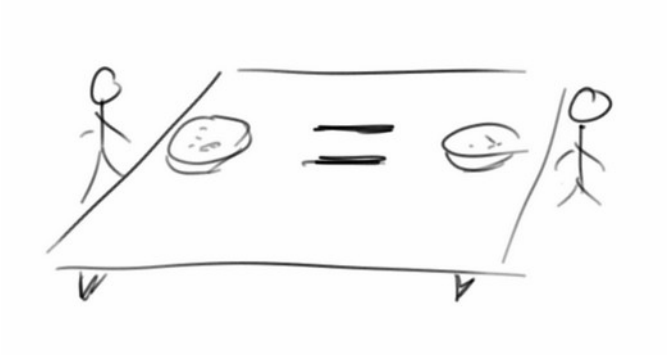


Several Threads

Jeder dieser neuen Threads macht sein eigenes Ding. Wir sehen auch, dass es außer beim Erstellen, sehr wenig Interaktion zwischen den verschiedenen Threads gibt, und das ist eine gute Sache.

Für moderne CPUs mit mehreren Kernen bedeutet die Verwendung mehrerer Threads, dass jeder Thread auf einem eigenen Kern laufen kann. Im Gegensatz dazu verwenden wir im Single-Thread-Modus nur einen der Kerne, die anderen tun nichts. Das ist Verschwendung. Wenn wir also unsere Programme beschleunigen wollen, sollten wir Threads häufiger verwenden.

Dining Philosophers



Also, warum hat jeder so viel Angst vor Threads, scheint bisher ziemlich harmlos zu sein. Nun, es gibt ein berühmtes Beispiel, das sehr schön visualisiert, was alles schief gehen kann: es nennt sich das *Philosophenproblem*, oder auf Englisch "Dining Philosophers" [1].

Die Geschichte geht so: Es waren einmal zwei chinesische Philosophen, die an einem Tisch saßen. Sie philosophierten den ganzen Tag und die einzige Ablenkung die sie hatten, war ab und zu zu essen. Jeder Philosoph bekam eine Schüssel Reis. Allerdings gab es nur zwei Essstäbchen. Jetzt ist es unmöglich Reis mit nur einem Stäbchen zu essen (zumindest für unsere Philosophen). Um zu überleben, muss ein Philosoph also beide Stäbchen bekommen.

Grundsätzlich gibt es da drei Möglichkeiten:

- Der erste Philosoph nimmt sehr schnell beide Stäbchen, isst den Reis und behält die Stäbchen für sich. Die Folge ist, dass es ihm gut geht, aber sein Mitphilosoph wird verhungern.
- Der erste Philosoph nimmt sehr schnell beide Stäbchen, isst den Reis und legt die Stäbchen wieder auf den Tisch. Sein Mitphilosoph nimmt die Stäbchen, isst. Beiden geht es gut und sie können fleißig weiter philosophieren.
- Jeder nimmt ein Stäbchen und behält es. Dann werden beide verhungern.

Es ist klar, dass das Problem in der Verteilung der Stäbchen liegt. Wenn jeder Philosoph sein eigenes Paar Essstäbchen hätte, würden beide glücklich bis ans Ende ihrer Tage leben. Aber weil es eine gemeinsame Resource gibt, die Stäbchen, und diese Resource für das Überleben unerlässlich ist, könnte es zu Problemen kommen.

Diese Probleme haben Namen und die häufigsten sind die folgenden vier:

- **Race Condition:** Ein Thread hat die kritische Resource und gibt sie nicht zurück. Philosoph 1 hat beide Stäbchen, isst weiter und gibt die Stäbchen nicht zurück.
- **Starvation:** Ein Thread kann nicht laufen, weil er auf eine kritische Resource wartet. Philosoph 2 wartet auf die Stäbchen und verhungert.
- **Dead Lock:** Ein Thread hat einen Teil einer kritischen Ressource, braucht aber einen anderen Teil, um mit seiner Aufgabe fortzufahren. Philosoph 1 hat ein Stäbchen und Philosoph 2 hat das andere. Aber man braucht beide Stäbchen, um zu essen (und zu überleben).
- **Lock Starvation:** Das passiert, wenn eine kritische Ressource von einem Thread gesperrt ist und somit kein anderer Thread darauf zugreifen kann, was bedeutet, dass der andere Thread verhungert. Wenn einer der Philosophen ein Stäbchen behält und es nicht zurückgibt, dann verhungert der andere.

Im Allgemeinen führt Thread-unsicherer Code zu diesen Problemen. Also müssen wir lernen, wie man Thread-Save Code schreibt!

Was ist also die Lösung in unserem Philosophenproblem? Die Lösung besteht aus zwei Teilen: Erstens, wenn ein Philosoph beide Stäbchen bekommt, sollte er sie nach dem Essen zurückgeben und den anderen im Idealfall benachrichtigen. Zweitens, falls er nur eines der Stäbchen bekommt, aber nicht das andere, sollte er dieses zurückgeben, ein wenig warten, und es erneut versuchen.

PhilosopherProgram

Sehen wir uns das Dining Philosopher Problem mal aus der Nähe an: dazu implementieren wir das Problem als normale Java Konsolenanwendung. Wir verwenden Standard Java und nicht Android, da wir uns auf das wesentliche konzentrieren wollen. Wir fangen mit unseren Stäbchen an: Das ist eine sehr einfache Klasse, sie hat nur einen Namen als Instanzvariable:

```
    private class ChopStick {
        private String name;

        public ChopStick(String name) {
            this.name = name;
        }

        public String getName() {
            return name;
        }
    }
```

Auch das Hauptprogramm ist ziemlich einfach. Es ist ein Standard-Java-Programm. Wir erstellen zwei Essstäbchen, fügen sie zu einer Liste hinzu, und wir erstellen zwei Philosophen mit ihren jeweiligen Threads:

```
public class PhilosopherProgram {

    private List<ChopStick> chopSticks;

    public PhilosopherProgram() {
        // create list and add two chop sticks:
        chopSticks = Collections.synchronizedList(
            new ArrayList<ChopStick>());
        chopSticks.add(new ChopStick("A"));
        chopSticks.add(new ChopStick("B"));
        // create two philosophers:
        Philosopher p1 = new Philosopher("1", chopSticks);
        Philosopher p2 = new Philosopher("2", chopSticks);
        Thread t1 = new Thread(p1);
        Thread t2 = new Thread(p2);
        t1.start();
        t2.start();
    }

    public static void main(String[] args) {
        PhilosopherProgram pp = new PhilosopherProgram();
    }
    ...
}
```

Wichtig ist, dass unsere Liste von zwei Chopsticks, *chopSticks*, die gemeinsame Resource ist. Deshalb haben wir die Liste mit *synchronizedList()* zu einer synchronisierten Liste gemacht. Wann immer wir eine Liste zwischen Threads teilen, müssen wir sicherstellen, dass es sich um eine synchronisierte Version handelt.

Die *Philosopher* Klasse selbst implementiert das Runnable Interface, d.h. sie ist ein Thread. Zusätzlich hat sie einen Namen, eine Referenz auf die gemeinsame Ressource, die *chopSticks*, und sie hat einen Lebenszähler, die *liveForce*. Ist die *liveForce* auf Null, ist der Philosoph tot.

```
    private class Philosopher implements Runnable {
        private String name;
        private List<ChopStick> chopSticks;
        private int liveForce = 5;

        public Philosopher(String name, List<ChopStick> chopSticks) {
            this.name = name;
            this.chopSticks = chopSticks;
        }


        @Override
        public void run() {
            ...
        }
    }
```

Was uns im Weiteren interessiert ist zu sehen, was alles in der run() Methode schief gehen kann.

Philosopher v.1

Beginnen wir mit dem einfachsten Ansatz. Unser Programm soll solange weiter laufen, solange unsere Philosophen noch am Leben sind, also noch *liveForce* haben. Bei jeder Iteration verlieren unsere Philosophen ein Leben. Nur wenn sie beide Stäbchen haben, können sie essen und damit ihre liveForce wieder auffüllen. Wenn alle Leben aufgebraucht sind, sind sie tot.

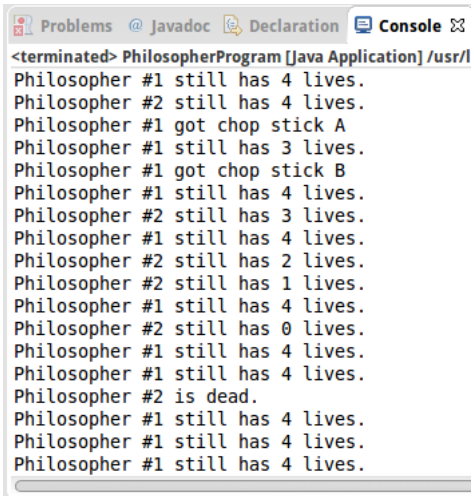
```
private void run() {
    ChopStick cs1 = null;
    ChopStick cs2 = null;
    while (liveForce > 0) {
        liveForce--;
        System.out.println("Philosopher #" + name + " still has "
                + liveForce + " lives.");

        // wait between 0 and DELAY milliseconds
        pause(new Random().nextInt(DELAY));


        // try to get both chop sticks
        if (cs1 == null) {
            if (chopSticks.size() > 0) {
                cs1 = chopSticks.remove(0);
                System.out.println("Philosopher #" + name
                        + " got chop stick " + cs1.getName());
            }
        } else {
            if (chopSticks.size() > 0) {
                cs2 = chopSticks.remove(0);
                System.out.println("Philosopher #" + name
                        + " got chop stick " + cs2.getName());
            }
        }

        // do we have both chop sticks?
        if ((cs1 != null) && (cs2 != null)) {
            liveForce = 5;
        }
    }
    System.out.println("Philosopher #" + name + " is dead.");
}
```

Was die Stäbchen angeht, versuchen wir zuerst, das ersten Stäbchen zu bekommen. Sobald wir das erste haben, versuchen wir das zweite zu bekommen.

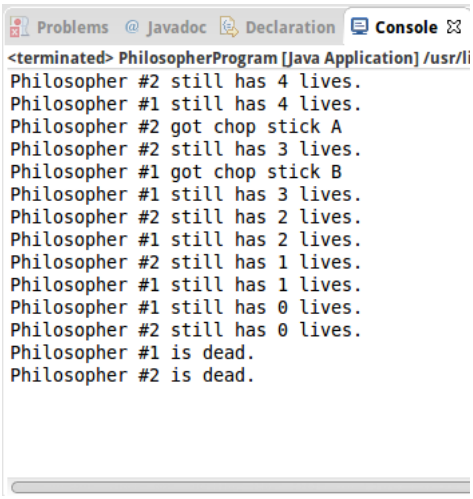
Das mag jetzt etwas komisch erscheinen, aber es ist meist nicht ganz trivial Threading Probleme zu entdecken. Deswegen übertreiben wir hier ein wenig, damit die Probleme deutlich zu Tage treten. Das ist auch der Grund warum wir oben die Pause eingefügt haben. Die Probleme existieren ohne die Pause, sie treten dann aber nur seltener auf. (Für manch einen mag das auch schon eine Lösung sein...)

Wenn wir diesen Code ein paar Mal ausführen, bemerken wir zwei verschiedene Ergebnisse. Die meiste Zeit werden beide Philosophen sterben, denn jeder bekommt nur ein Stäbchen. Das ist der *Dead Lock*, der zum Verhungern (*Starvation*) führt. Aber ab und zu bekommt der eine Philosoph beide Stäbchen (*Race Condition*) und der andere Philosoph wird sterben (*Starvation*). Es passiert aber nie, dass beide Philosophen überleben. Entweder stirbt einer oder es sterben beide.

Man sieht das beispielhaft an der folgende Ausgabe im LogCat:



Race Condition



Starvation

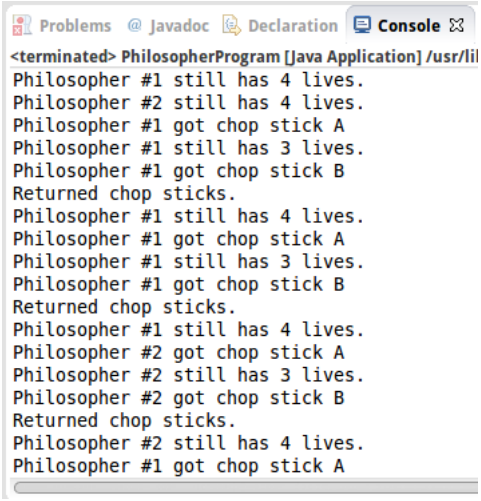
Philosopher v.2

Offensichtlich haben wir zwei Probleme. Wir wollen das zweite (die *Race Condition*) zuerst lösen: wenn wir anständige Philosophen sind, dann müssen wir die Stäbchen nach dem Essen zurückgeben:

```
private void run() {
    ...

        // do we have both chop sticks?
        if ((cs1 != null) && (cs2 != null)) {
            liveForce = 5;

            // return chop sticks
            chopSticks.add(cs1);
            chopSticks.add(cs2);
            cs1 = null;
            cs2 = null;
            System.out.println("Returned chop sticks.");
        }

    ...
}
```

Wir haben das Problem der *Race Condition* gelöst. Das ist also eine gute Sache. Allerdings nur für uns, nicht für die Philosophen. Denn wenn wir den Code ein paar Mal ausführen, stellen wir fest, dass eigentlich immer beide Philosophen sterben, nur dauert es jetzt länger.

Im LogCat sieht das so aus:



Philosopher v.2

Synchronization

Wie lösen wir das Deadlock Problem, wenn also jeder Philosoph nur ein Stäbchen hat? Eine Idee wäre, nachdem wir den ersten Chop-Stick haben, zu versuchen den zweiten zu bekommen. Wenn wir es schaffen, gut. Wenn wir es nicht schaffen, könnten wir unseren ersten Chop-Stick zurücklegen und von vorne anfangen. Das sieht erst einmal nach einer guten Lösung aus. Es ist auch besser als die anderen Lösungsansätze die wir bisher versucht haben. Aber wenn wir den Code ausführen, sterben unsere Philosophen immer noch.

Wir könnten noch ein paar andere Sachen versuchen, aber es verschiebt nur das eigentliche Problem. Wir brauchen eine radikal andere Lösung: Was, wenn wir dem anderen Philosophen verbieten könnten, an die Stäbchen zu kommen? Wir bauen eine Barriere auf dem Tisch, die es uns erlaubt, beide Stäbchen zu greifen. Dann essen wir, und nachdem wir fertig sind, legen wir die Stäbchen zurück und entfernen die Barriere. Der andere kann das Gleiche tun. Dieses "Barrierebauen" nennt man auch *Synchronisation*.

Philosopher v.3

Unser Code mit Synchronisation sieht so aus:

```
private void run() {
    ...
        // try to get both chop sticks, lock the other guy(s) out
        synchronized (chopSticks) {
            if (chopSticks.size() > 0) {
                cs1 = chopSticks.remove(0);
                System.out.println("Philosopher #" + name
                        + " got chop stick " + cs1.getName());
            }

            if (chopSticks.size() > 0) {
                cs2 = chopSticks.remove(0);
                System.out.println("Philosopher #" + name
                        + " got chop stick " + cs2.getName());
            }
        }
    ...
}
```

Der synchronisierte Block wird wie eine Zeile Code ausgeführt, d.h. alle anderen Threads (und Philosophen) müssen warten, bis wir mit unserem Codeblock fertig sind. Das ist die Holzhammer Methode, sie ist nicht sehr elegant, aber sie löst unser Deadlock Problem. Wenn unsere Philosophen die Stäbchen nach dem Essen zurückgeben, dann werden beide mal dran kommen und beide überleben.

Philosopher v.4

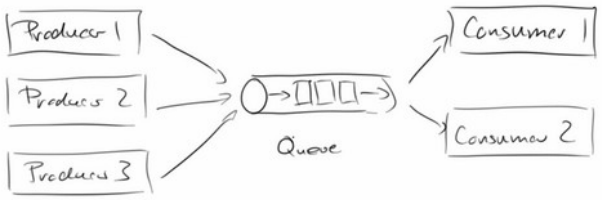
Dass wir den Ansatz oben die "Holzhammer Methode" genannt haben, deutet darauf hin, dass es anscheinend auch eine vornehmere Methode gibt. Die wollen wir uns jetzt ansehen. In unserer dritten Version verwendeten wir den so genannten Polling-Ansatz: Der Philosoph, der die Stäbchen nicht bekommt, muss immer wieder versuchen, sie zu bekommen, das ist Polling. Polling ist nicht sehr effizient. Anstelle es immer wieder zu versuchen und zu versuchen, wäre es doch besser zu warten (*wait*), wenn wir wissen, dass jemand anderes die Stäbchen hat. Wenn aber der andere fertig ist, sollte er uns doch gefälligst benachrichtigen (*notify*). Das ist es, was der folgende Code tut:

```
public void run() {
    ...

        // try to get both chop sticks
        synchronized (chopSticks) {
            if (chopSticks.size() < 2) {
                try {
                    chopSticks.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            } else {
                cs1 = chopSticks.remove(0);
                System.out.println("Philosopher #" + name
                        + " got chop stick " + cs1.getName());
                cs2 = chopSticks.remove(0);
                System.out.println("Philosopher #" + name
                        + " got chop stick " + cs2.getName());
            }
        }

        // do we have both chop sticks?
        if ((cs1 != null) && (cs2 != null)) {
            liveForce = 5;
            // now return the chop sticks:
            synchronized (chopSticks) {
                chopSticks.add(cs1);
                chopSticks.add(cs2);
                cs1 = null;
                cs2 = null;
                System.out.println("Returned chop sticks.");
                chopSticks.notifyAll();
            }
        }
    ...
}
```

Wir beginnen damit, mit *synchronized* die anderen Philisophen auszusperren. Dann checken wir, ob zwei Stäbchen verfügbar sind. Wenn nicht, dann warten wir. Wenn ja, schnappen wir uns beide und essen. Sobald wir mit dem Essen fertig sind, geben wir beide Stäbchen zurück und benachrichtigen die anderen, dass die Stäbchen wieder verfügbar sind.

Hypothetisch hätten wir auch einen riesigen synchronisierten Codeblock erstellen und alles darin machen können. Aber das ist eine schlechte Angewohnheit und anfällig für alle möglichen Probleme. In der Regel sollten synchronisierte Blöcke so klein wie möglich sein. Das haben wir im obigen Code getan. Wenn wir unseren Code jetzt testen, sehen wir, dass unsere beiden Philosophen glücklich bis ans Ende ihrer Tage leben werden.

Producer - Consumer Problem



Sehr oft treffen wir auf ein Szenario, in dem ein Thread Nachrichten an einen anderen Thread sendet. Dies wird auch als das Producer-Consumer Problem bezeichnet [2]. Ein Thread produziert also immer die Nachrichten und der andere Thread verbraucht sie.

Ein Beispiel ist ein Webserver: der lauscht immer auf eingehende Requests. Normalerweise leitet dieser Listener-Thread diese eingehenden Nachrichten einfach an andere Threads weiter, um die Arbeit zu erledigen. Denn wenn der Listener Thread auch die Arbeit machen würde, könnte ein Teil der neu eingehenden Nachricht verloren gehen.

Die beiden Threads teilen sich eine gemeinsame Ressource, die Requests. Wie wir bei unseren Philosophen gesehen haben, könnte das zu Problemen führen. Da dies ein Problem ist, das sehr häufig auftritt, haben sich Leute eine Lösung ausgedacht, die *BlockingQueue* [3]. Wir sehen uns die blockierende Warteschlange gleich mal im Code an. In unserer Activity erstellen wir eine BlockingQueue und je einen *Producer* und *Consumer* Thread:

```
public class ProducerConsumerActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // can keep at most 10 elements:
        BlockingQueue<String> queue =
            new ArrayBlockingQueue<String>(10);

        Producer p = new Producer(queue);
        Consumer c = new Consumer(queue);
        Thread t1 = new Thread(p);
        Thread t2 = new Thread(c);
        t1.start();
        t2.start();
    }
    ...
}
```

Im Allgemeinen kann es mehr als einen Consumer und auch mehr als einen Producer geben. In unserem Fall sind die Nachrichten nur Strings, aber das kann alles mögliche sein, sogar binäre Daten.

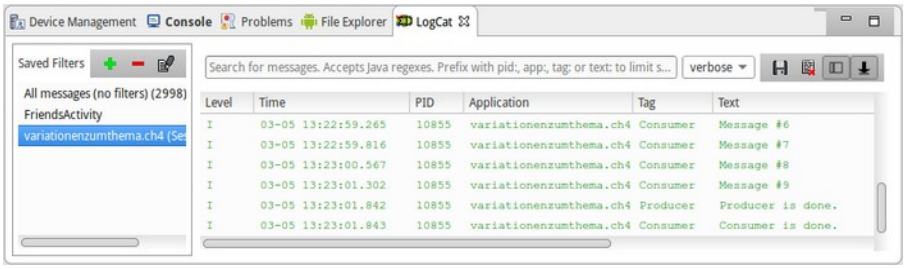
Der Producer produziert die Nachrichten und fügt sie in die Warteschlange ein:

```
        private class Producer implements Runnable {
            private BlockingQueue<String> queue;

            public Producer(BlockingQueue<String> queue) {
                this.queue = queue;
            }

            @Override
            public void run() {
                try {
                    // send ten messages on average one per second
                    for (int i = 0; i < 10; i++) {
                        queue.put("Message #" + i);
                        pause(DELAY / 2 + rnd.nextInt(DELAY / 2));
                    }

                    // end the consumer thread
                    queue.put("quit");
                    Log.i("Producer", "Producer is done.");

                } catch (InterruptedException e) {
                    Log.i("Producer", e.getMessage());
                }

            }
        }
```

Der Konsument nimmt sie aus der Warteschlange und konsumiert sie:

```
        private class Consumer implements Runnable {
            private BlockingQueue<String> queue;

            public Consumer(BlockingQueue<String> queue) {
                this.queue = queue;
            }



            public void run() {
                try {
                    // receive messages
                    String msg;
                    while (!(msg = queue.take()).equals("quit")) {
                        Log.i("Consumer", msg);
                    }

                    Log.i("Consumer", "Consumer is done.");

                } catch (InterruptedException e) {
                    Log.i("Consumer", e.getMessage());
                }
            }
        }
```

Wirklich nicht so schwer. Im LogCat sieht das dann so aus:



Wenn wir mit BlockingQueues in Java arbeiten, haben wir verschiedene Implementierungen zur Auswahl: ArrayBlockingQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedTransferQueue, PriorityBlockingQueue, und SynchronousQueue. Und es gibt auch verschiedene Methoden zum Schreiben, Lesen und Prüfen der Warteschlange. Man sollte hier erst einmal die Dokumentation lesen [3] und danach eine bewusste Entscheidung treffen.

Review

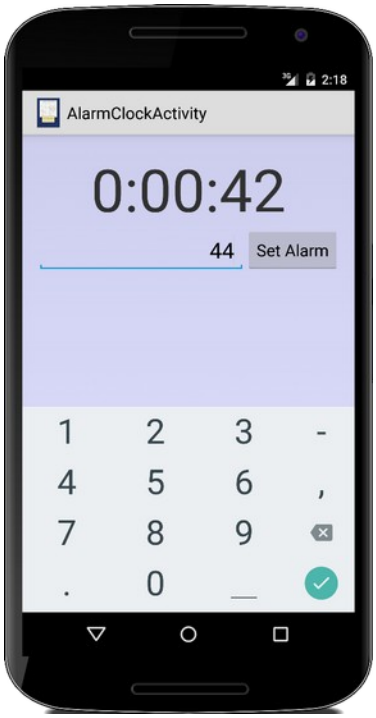
In diesem Kapitel haben wir zunächst was zu TimerTask und AsyncTask erfahren. Dann haben wir gesehen wie man Threads erstellt und ausführt, aber auch die Probleme kennen gelernt die dabei auftreten können wie Deadlock, Race-Condition und Starvation. Gelöst haben wir diese Problem mit Synchronisation. Und schließlich haben wir noch das Producer-Consumer Problem mit einer BlockingQueue gelöst.

Projekte

In den Projekten vertiefen wir den AsyncTask mit zwei Beispielen. Anschließend werden wir sehen wie Grafikprogramme von mehreren Threads profitieren können, dabei werden wir vor allem auch das Thema Synchronisation vertiefen. Zum Schluss sehen wir noch wie wir explizit von den vielen CPU Cores in unserem Smartphones Nutzen ziehen können.

AlarmClock

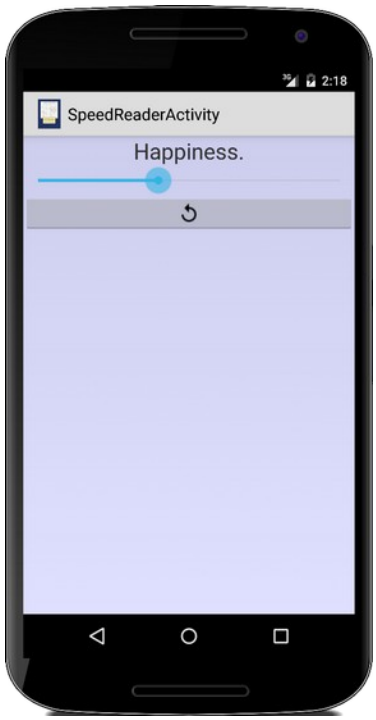
Eine erste Anwendung für den AsyncTask ist ein Wecker. Die UI können wir aus dem ersten Semester borgen. Was neu ist ist der AlarmClockTask. Der wird über die onClick() gestartet, und wir sagen über die execute() Methode auch wann der Alarm losgehen soll:




```
public class AlarmClockActivity extends Activity {

    ...
    // UI stuff
        btnStart.setOnClickListener(
            new View.OnClickListener() {
                public void onClick(View v) {
                    ...
                    long alarmTime = ...;
                    srt = new AlarmClockTask();
                    srt.execute(alarmTime);
                }
            }
        );
    ...


    // Params, Progress, Result
    private class AlarmClockTask extends AsyncTask<Long, Long, String>
    {

        @Override
        protected String doInBackground(Long... params) {
            long alarmTime = params[0];
            while (true) {
                long remainingTime =
                    alarmTime - System.currentTimeMillis();
                if (remainingTime <= 0)
                    break;
                publishProgress(remainingTime);
                pause(DELAY);
            }
            return "done";
        }

        @Override
        protected void onProgressUpdate(Long... values) {
            face.setText(convertSecondsInTime(values[0]));
        }

        @Override
        protected void onPostExecute(String result) {
            // 100% volume
            ToneGenerator toneG =
                new ToneGenerator(AudioManager.STREAM_ALARM, 100);
            // 1000 ms
            toneG.startTone(
                ToneGenerator.TONE_CDMA_ALERT_CALL_GUARD, 1000);
        }
        ...
    }
}
```

Die *alarmTime* wird nämlich der doInBackground() Methode als Parameter übergeben. Über publishProgress() wird dann in der onProgressUpdate() die UI geupdated. Wenn die Zeit abgelaufen ist, wird die onPostExecute() aufgerufen, und dort erzeugen wir über den ToneGenerator einen Alarmton. Interessant ist vielleicht noch zu beobachten, was passiert wenn unsere Activty in den Hintergrund geht, oder wenn das Smartphone ausgeschaltet wird.

SpeedReader



Es gibt eine Vielzahl von Techniken wie man seine Lesegeschwindigkeit erhöhen kann [4], eine davon ist die App die wir gleich schreiben werden. Auch hier benötigen wir wieder einen AsyncTask um auf die UI zugreifen zu können.

Die UI besteht aus einem TextView für den zu lesenden Text, einer SeekBar für die Geschwindigkeit, und einem Button um das Ganze noch mal zu wiederholen.

```
public class SpeedReaderActivity extends Activity {
    private final int MAX_SPEED = 1000;
    private final int INITIAL_SPEED = 500;

    private String text =
        "We hold these truths to be self-evident, "
        + "that all men are created equal, "
        + "that they are endowed by their Creator "
        + "with certain unalienable Rights, "
        + "that among these are Life, Liberty "
        + "and the pursuit of Happiness.";
    ...
}
```

Auch hier wird der AsyncTask gestartet, wenn wir auf den Button klicken. In der execute() Methode übergeben wir dieses Mal aber den Text der vorgelesen werden soll. Der SpeedReaderTask selbst besteht lediglich aus zwei Methoden:

```
    // Params, Progress, Result
    private class SpeedReaderTask
        extends AsyncTask<String, String, String> {

        protected String doInBackground(String... params) {
            String text = params[0];
            String[] words = text.split(" ");
            for (String word : words) {
                pause(delay);
                publishProgress(word);
            }
            return "done";
        }

        protected void onProgressUpdate(String... values) {
            String word = values[0];
            tv.setText(word);
        }
    }
```

Confetti

Das Konfetti-Programm aus dem ersten Semester soll das erste Opfer unserer neuen Superpowers werden: Anstelle einfach dumm herunterzufallen, sollen die Confetti sich auch noch ein bisschen zufällig hin und her bewegen. In unserem ursprünglichen Programm waren die Confetti zufällig gefärbte GOvals. Ähnlich wie wir GRects in der SliderThreadingActivity zu Sliders gemacht haben, verwandeln wir nun die GOvals in Confettis, die das Runnable Interface implementieren:

```
private class Confetti extends GOval
                       implements Runnable {

    public Confetti(int width, int col) {
        super(width, width);
        setFilled(true);
        setFillColor(col);
    }

    @Override
    public void run() {
        // animate the slide across the screen
        for (int i = 0; i < HEIGHT / STEP; i++) {
            pause(DELAY_MOVE);
            int x = (int) (Math.random() * 2 - 1);
            move(x, STEP);
        }
    }
}
```

In der run() Methode bewegen wir das Confetti um den Betrag STEP nach unten und lassen es auch ein kleines bisschen nach links oder rechts wandern. In unserem GraphicsProgram erstellen wir neue Confetti wie im ersten Semester, und machen dann aber daraus Threads:

```
public class ConfettiActivity extends GraphicsProgram {
    ...

    public void run() {
        waitForTouch();
        HEIGHT = getHeight();
        while (true) {
            // create a new random confetti
            int width = rgen.nextInt(SIZE / 2, SIZE);
            int col = rgen.nextColor();
            double x = rgen.nextDouble(-SIZE / 2, getWidth());
            double y = rgen.nextDouble(-SIZE / 2, 100);
            Confetti confetti = new Confetti(width, col);
            add(confetti, x, y);

            // run the confetti in a new Thread
            Thread confettiThread = new Thread(confetti);
            confettiThread.start();

            pause(DELAY_CREATION);
        }
    }
    ...
}
```

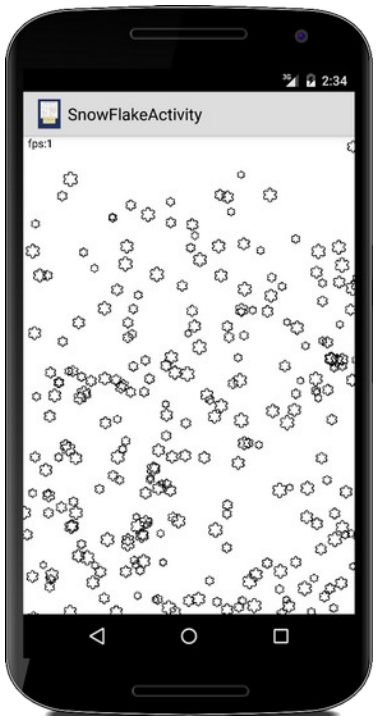
Observations

Der Code, wie er ist, läuft gut, aber wenn wir ihn eine Weile laufen lassen, werden wir einige interessante Beobachtungen machen. Wir stellen fest, dass sich das ganze Konfetti am Boden sammelt. Das sieht gut aus, aber es ist ein Problem: Das sind alles tote Threads, und der Müll wird nicht eingesammelt (garbage collected). In einem lang laufenden Programm führt dies früher oder später zu Speicherproblemen.

Wenn wir die Anzahl der pro Sekunde erzeugten Konfetti erhöhen, dann friert das Programm irgendwann einfach ein. In einer ersten Vermutung könnten wir denken, dass das damit zu tun hat, dass wir mehr Konfetti haben, als wir pro Sekunde zeichnen können. Aber das ist nicht richtig. Der wahre Grund liegt in der sehr schlampigen Art und Weise, wie wir die draw() Methode der GObjects in der onDraw() Methode der GView Klasse aufrufen.

Noch etwas anderes: Lassen wir die App ein wenig laufen. Dann schicken wir die App in den Hintergrund, indem wir etwas anderes auf unserem Handy starten und ein wenig warten. Dann kehren wir zu unserer Confetti App zurück: wenn wir genau hinsehen, bemerken wir vielleicht kurz den alten Zustand der Benutzeroberfläche, und dann tauchen auf einmal ganz viele neue Confettis auf. Das heißt, obwohl der UI-Thread nicht lief, wurden weiterhin im Hintergrund Confettis generiert. Jedes Mal, wenn wir einen Thread starten, einschließlich des Haupt-Threads, läuft der so lange weiter, wie er in seiner run() Methode ist. Daher ist unser Ansatz, eine Endlosschleife innerhalb der run() Methode durchzuführen, nicht besonders schlau.

Snowflakes



Am Ende von Kapitel vier im zweiten Buch sahen wir die KochSnowflake. Stellt sich heraus, wenn wir die Confetti im obigen Programm durch Schneeflocken ersetzen, bekommt das Ganze eine weihnachtliche Atmosphäre. Eine Möglichkeit, eine Klasse SnowFlake zu erstellen, ist mithilfe des GPolygon. Wenn das mit GPolygonen aber nicht funktioniert, kann man alternativ auch nach Bildern von Schneeflocken suchen und die GImage-Klasse verwenden.

```
private class SnowFlake extends GPolygon
                        implements Runnable {
    ...
    public SnowFlake(int size) {
        super();
        createKochSnowflake(0, 0, size,
                            NR_OF_ITERATIONS);
    }

    private void createKochSnowflake(int x, int y,
                  int length, int nrOfIterations) {
        drawKochLine(x, y, length, 0,
                     nrOfIterations);
        drawKochLine(x + length, y, length, -120,
                     nrOfIterations);
        double x1 = x + length * Math.cos(-60 * Math.PI / 180);
        double y1 = y - length * Math.sin(-60 * Math.PI / 180);
        drawKochLine(x1, y1, length, 120, nrOfIterations);
    }

    private void drawKochLine(double x0, double y0, double length,
                              double angle, int nrOfIterations) {
        // base case:
        if (nrOfIterations == 0) {
            double x1 = x0 + length * Math.cos(angle * Math.PI / 180);
            double y1 = y0 - length * Math.sin(angle * Math.PI / 180);
            addVertex((int) x1, (int) y1);
            return;
```

```
            // recursive case:
            } else {
                double len = length / 3;
                double ang = angle;
                drawKochLine(x0, y0, len, ang + 0, nrOfIterations - 1);
                double x1 = x0 + len * Math.cos(ang * Math.PI / 180);
                double y1 = y0 - len * Math.sin(ang * Math.PI / 180);
                drawKochLine(x1, y1, len, ang + 60, nrOfIterations - 1);
                ang = ang + 60;
                double x2 = x1 + len * Math.cos(ang * Math.PI / 180);
                double y2 = y1 - len * Math.sin(ang * Math.PI / 180);
                drawKochLine(x2, y2, len, ang - 120, nrOfIterations - 1);
                ang = ang - 120;
                double x3 = x2 + len * Math.cos(ang * Math.PI / 180);
                double y3 = y2 - len * Math.sin(ang * Math.PI / 180);
                drawKochLine(x3, y3, len, ang + 60, nrOfIterations - 1);
            }
        }
    }
```

Fireworks

Eine weitere schöne Anwendung von Multi-Threading ist ein Feuerwerksprogramm. Ein Feuerwerk besteht aus Raketen, die explodieren, und dann folgen kleine Lichter den Gesetzen der Schwerkraft. Wir ignorieren die Rakete und kümmern uns nur um die kleinen Lichter nach der Explosion:

```
public class FireWorksActivity
                        extends GraphicsProgram {
    public void run() {
        ...
        while (true) {
            startExplosion();
            pause(DELAY);.
        }
    }
    ...
}
```

Bei der Explosion erzeugen wir eine feste Anzahl von Lichtern (GLight), jedes mit der gleichen Farbe, alle an der gleichen Stelle beginnend, aber jedes mit einer anderen Richtung:

```
    private void startExplosion() {
        int col = rgen.nextBrightColor();
        int x = rgen.nextInt(0, getWidth());
        int y = rgen.nextInt(0, getHeight());
        double angle = Math.random(); // start with a random angle
        double deltaAngle = 2.0 * Math.PI / NR_LIGHTS;
        for (int i = 0; i < NR_LIGHTS; i++) {
            double vx = SPEED * Math.cos(angle);
            double vy = SPEED * Math.sin(angle);
            GLight light = new GLight(vx, vy, col);
            add(light, x, y);

            Thread thread = new Thread(light);
            thread.start();

            angle += deltaAngle;
        }
    }
```

Aus jedem Licht wird ein eigener Thread. D.h. bei vielen Explosionen gibt es viele Threads! Die *GLight* Klasse selbst ist trivial:

```
private class GLight extends GOval implements Runnable {
    private static final int DELAY = 40;
    private static final double GRAVITY = 0.05;

    double vx;
    double vy;

    public GLight(double vx, double vy, int col) {
        super(SIZE, SIZE);
        setFillColor(col);
        setFilled(true);
        this.vx = vx;
        this.vy = vy;
    }

    @Override
    public void run() {
        // animate the light
        for (int i = 0; i < 100; i++) {
            pause(DELAY);
            move((int) vx, (int) vy);
            vy = vy + GRAVITY;
        }
        // make the lights invisible:
        setFillColor(Color.BLACK);
    }
}
```

Observations

Mit unserer Fireworks Activity können wir noch ein paar zusätzliche Dinge lernen. Wenn wir die Anwendung etwas länger laufen lassen, dann stellen wir fest, dass die Framerate mit der Zeit abnimmt. Warum ist das so? Weil keines der Lichtobjekte vom Bildschirm entfernt wird. (Wenn wir die Anzahl der Feuerwerke, die wir pro Sekunde erzeugen, erhöhen, wird der Effekt früher sichtbar.)

Um das Ganze etwas näher zu untersuchen, sehen wir uns mal an wie viel Speicher unsere Anwendung verbraucht. Mit der folgenden Methode können wir den Speicherverbrauch ermitteln und im Log ausgeben:

```
private ActivityManager.MemoryInfo getAvailableMemory() {
    ActivityManager activityManager =
        (ActivityManager) this.getSystemService(ACTIVITY_SERVICE);
    ActivityManager.MemoryInfo memoryInfo =
        new ActivityManager.MemoryInfo();
    activityManager.getMemoryInfo(memoryInfo);
    return memoryInfo;
}
```

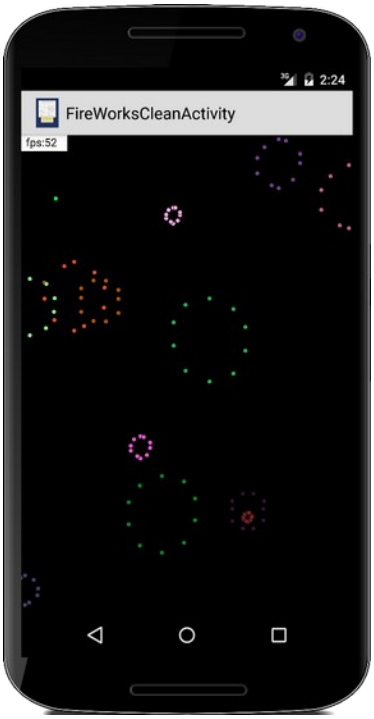
Nicht überraschend stellen wir fest, dass der verfügbare Speicher immer weniger wird. Ab und zu schlägt der Garbage Collector zu, aber er kann nicht wirklich viel tun, da wir für immer an unseren Lichtern festhalten, was zur Folge hat, dass uns früher oder später der Speicher ausgeht. Wieder können wir den Effekt beschleunigen, indem wir die folgenden beiden Zeilen zu unserer GLight Klasse hinzufügen:

```
private class Light extends GOval implements Runnable {
    ...
    byte[] oneKiloByte;

    public Light(double vx, double vy, int col) {
        ...
        oneKiloByte = new byte[1024];
    }
    ...
}
```

Die Änderung macht unsere Lichter ein wenig "schwerer" aus Sicht des Speichers.

Also auch die Fireworks Activity hat das gleiche Problem wie Confetti: Wenn wir die App ein wenig laufen lassen, sie dann in den Hintergrund schicken, und schließlich wieder zu unserer App zurückkehren, bemerken wir wieder, dass ganz kurz der alten Zustand der Benutzeroberfläche sichtbar ist, bevor dann ganz viele neue Lichter auftauchen, und je nachdem wie lange wir gewartet haben, friert die App ein.

Cleanup after Yourself



Wenigstens einmal sollten wir ein Programm schreiben, das seinen Müll auch wieder aufräumt. Grundsätzlich müssen wir die Lichter aus unserem Programm entfernen. Dies geschieht mit der remove() Methode der Klasse GraphicsProgram. Aber woher wissen wir, wann wir ein Licht entfernen müssen? Wenn das Licht fertig ist, verlässt es die run() Methode. Danach ist der Licht-Thread tot. Glücklicherweise können wir einen Thread mit der *isAlive()* Methode fragen, ob er tot oder lebendig ist.

Wir sammeln also alle Threads erst einmal in einer Map:

```
public class FireWorksCleanActivity
                        extends GraphicsProgram {
    private Map<Thread, Light> allLights =
        new HashMap<Thread, Light>();

    private void startExplosion() {
        ...
        for (int i = 0; i < NR_LIGHTS; i++) {
            ...
            Light light = new Light(vx, vy, col);
            add(light, x, y);

            Thread thread = new Thread(light);
            thread.start();

            allLights.put(thread, light);
            ...
        }
    }
    ...
}
```

Dann müssen wir in regelmäßigen Abständen unsere Liste durchgehen,

```
    public void run() {
        ...
        while (true) {
            ...
            cleanupLightsNoLongerUsed();
            ...
        }
    }
```

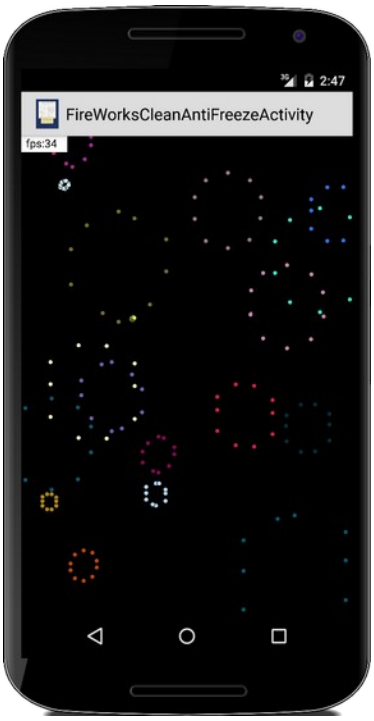
und nach toten Threads suchen:

```
private void cleanupLightsNoLongerUsed() {
    List<Thread> toBeRemoved = new ArrayList<Thread>();
    for (Thread t : allLights.keySet()) {
        if (!t.isAlive()) {
            Light l = allLights.get(t);
            // remove light from UI:
            remove(l);
            toBeRemoved.add(t);
        }
    }
    for (Thread t : toBeRemoved) {
        allLights.remove(t);
    }
}
```

Wir müssen zuerst die Lichter vom Bildschirm entfernen und anschließend auch aus unserer Map. Letzteres ist etwas tricky: während wir über eine Liste oder Map iterieren, dürfen wir keine Elemente aus der Liste entfernen. Deshalb brauchen wir eine temporäre Liste, um die Elemente zu sammeln, die entfernt werden sollen (*toBeRemoved*), und wenn wir alle gefunden haben, können wir sie dann am Ende entfernen.

Funktioniert es? Testen wir es: wir erhöhen wieder das "Gewicht" jeder Leuchte, indem wir die Größe des *oneKiloByte* Arrays um den Faktor zehn erhöhen. Wir werden feststellen, dass die App jetzt viel länger läuft. Was zeigt, dass wir gute Arbeit geleistet haben. Aber irgendwann (in meinem Fall, wenn das *oneKiloByte* etwa ein Megabyte groß ist) wird man eine wirklich seltsame Fehlermeldung bekommen: Der Müllsammler (Garbage Collector) gibt auf: viel zu viel zum Aufräumen.

AntiFreeze



Haben wir unser "Freeze"-Problem behoben? Nein, es ist noch da. Der Grund ist, dass unsere run() Methode weiterläuft, auch wenn unsere Anwendung in den Hintergrund geht:

```
public void run() {
    ...
    while (true) {
        startExplosion();
        pause(DELAY);
        cleanupLightsNoLongerUsed();
        Log.i("FireWorksActivity",
            "availMem=" +
            getAvailableMemory().availMem +
            ", nr of lights=" +
            allLights.size());
    }
}
```

Das sehen wir, wenn wir die Anzahl der Lichter beobachten: Obwohl unser Prozess im Hintergrund läuft, läuft er weiter und erhöht die Anzahl der Lichter. Wenn wir dann zurückkehren, laufen so viele Threads, dass unser UI-Thread nie genug Zeit hat, mit dem Zeichnen all der Lichter nachzukommen.

Was wir also tun müssen, ist unseren Hauptthread zu unterbrechen, wenn die Anwendung in den Hintergrund geht. Dafür überschreiben wir die Methode *onPause()*, die wird ja aufgerufen kurz bevor unsere Activity in den Hintergrund geschickt wird. In ihr teilen wir unseren Threads mit, dass sie schlafen sollen:

```
private Object mPauseLock;
private boolean mPaused;

protected void onPause() {
    synchronized (mPauseLock) {
        mPaused = true;
    }
    super.onPause();
}
```

Offensichtlich müssen wir unsere Threads wieder aufwecken, wenn wir aus dem Winterschlaf zurückkehren. Das geschieht in der onResume() Methode:

```
protected void onResume() {
    synchronized (mPauseLock) {
        mPaused = false;
        mPauseLock.notifyAll();
    }
    super.onResume();
}
```

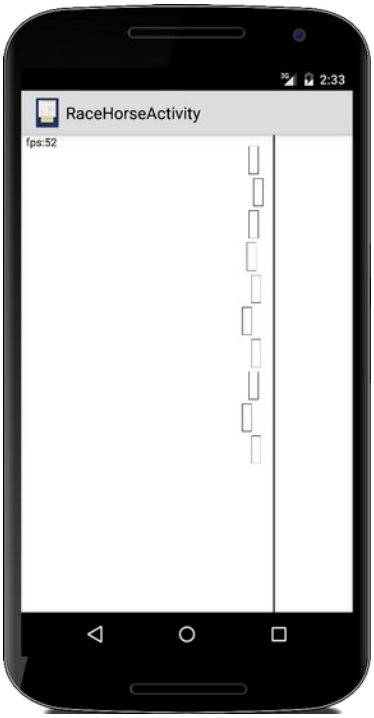
Und in unserem GameLoop müssen wir prüfen, ob wir pausieren sollen oder nicht:

```
public void run() {
    ...
    while (true) {
        startExplosion();
        pause(DELAY);
        cleanupLightsNoLongerUsed();
        Log.i("FireWorksActivity",
                "availMem=" + getAvailableMemory().availMem +
                ", nr of lights=" + allLights.size());

        synchronized (mPauseLock) {
            while (mPaused) {
                try {
                    mPauseLock.wait();
                } catch (InterruptedException e) {
                }
            }
        }
    }
}
```

Sie haben wahrscheinlich bemerkt, dass wir jedes Mal, wenn wir versuchen, die *mPaused* Variable zu lesen oder zu ändern, sorgfältig auf ein Sperrobjekt synchronisiert haben (*mPauseLock*). Und wir verwenden dieses Sperrobjekt, um die Methoden *wait()* und *notifyAll()* aufzurufen. Das Sperrobjekt übernimmt die Sperrverwaltung.

Wenn wir jetzt unsere Anwendung testen, wird sie funktionieren. Auch nach einer Woche im Hintergrund, lief meine App noch einwandfrei. Es gab ein kleines Speicherleck, das aber auch durch eine andere Anwendung verursacht worden sein könnte. Für mich ist eine Woche gut genug.

RaceHorses



In diesem letzten Beispiel wollen wir ein kleines Wettrennen durchführen, zum Beispiel mit Pferden [5]. Offensichtlich sind unsere Pferde GRects, was sonst. Wie immer beginnen wir mit der Aktivity, wir erstellen zehn Pferde, platzieren sie auf dem Bildschirm, verwandeln sie in Threads und starten sie:

```
public class RaceHorseActivity
                extends GraphicsProgram {

    public void run() {
        boolean[] isThereAWinner = new boolean[1];
        isThereAWinner[0] = false;

        // create ten horses
        RaceHorse[] rhs =
            new RaceHorse[NR_OF_HORSES];
        for (int i = 0; i < rhs.length; i++) {
            rhs[i] = new RaceHorse(
                            HORSE_SIZE / 2,
                            HORSE_SIZE,
                            isThereAWinner);
            add(rhs[i],
                100, 50 + ((HORSE_SIZE + 20) * i));
        }

        GThickLine finishLine =
            new GThickLine(600 - 2, 0, 600 - 2, getHeight(), 2);
        add(finishLine);

        // turn them into threads
        Thread[] ths = new Thread[NR_OF_HORSES];
        for (int i = 0; i < rhs.length; i++) {
            ths[i] = new Thread(rhs[i]);
        }

        // start them of
        for (int i = 0; i < rhs.length; i++) {
            ths[i].start();
        }
    }
```

Die Sache, die ein wenig knifflig ist, wie wissen wir, wer gewonnen hat? Es gibt da verschiedene Möglichkeiten, aber die eleganteste ist die folgende: wir deklarieren eine boolesche Variable *isThereAWinner*, setzen sie anfänglich auf false und übergeben sie an alle Pferde. Wenn ein Pferd die Ziellinie erreicht, prüft es den Wert dieser Variablen. Wenn der Wert *false* ist, dann ändert es ihn auf *true*, so dass alle anderen Pferde sehen, dass schon jemand vor ihnen im Ziel war:

```
private class RaceHorse extends GRect implements Runnable {
    private boolean[] isThereAWinner;

    public RaceHorse(double w, double h, boolean[] isThereAWinner) {
        super((int) w, (int) h);
        this.isThereAWinner = isThereAWinner;
    }

    @Override
    public void run() {
        for (int i = 0; i < 50; i++) {
            move(10, 0);
            pause(rgen.nextInt(50, 150));
        }
```

```
            // reached finish, check if I am first
            if (!isThereAWinner[0]) {
                isThereAWinner[0] = true;
                //pause(200):
                setFilled(true);
                setFillColor(Color.RED);
            }
            invalidate();
        }
    }
```

Warum haben wir ein Array anstelle einer stink-normalen booleschen Variable verwenden? Der Grund dafür ist, dass primitive Datentypen als Kopie übergeben werden (pass-by-value), während Objekte und Arrays als Referenz übergeben werden (pass-by-reference). In unserem Beispiel macht es keinen Sinn mit einer Kopie zu arbeiten, wir müssen den Wert des Originals verändern. (Man könnte weiter fragen, warum nicht die Wrapper-Klasse Boolean anstelle des Arrays verwenden? Schließlich werden Objekte ja als Referenz übergeben. Nun, versuchen Sie es und Sie werden sehen, dass die Wrapper-Klassen zu nichts gut sind).

Wenn wir unser Programm starten, scheint alles gut zu laufen. Aber erinnern wir uns an die Philosophen und ihrer gemeinsamen Ressource, dem Chopstick? Wir haben hier auch eine gemeinsame Ressource, das *isThereAWinner* Array. Könnte das ein Problem sein? Darauf können wir wetten. Alles, was wir tun müssen, ist die Zeile "pause(200)" zu entkommentieren. Auf einmal gibt es mehr als nur ein Pferd das denkt, dass es gewonnen hat. Eine klassische *Race-Condition*! (Jetzt wissen wir, woher der Name kommt).

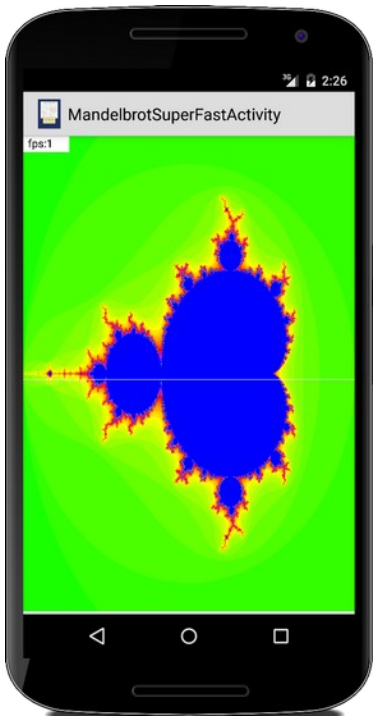
Wie lösen wir das Problem? Richtig, durch richtige Synchronisation. Wir müssen sicherstellen, dass der gesamte if-Block als eine Einheit ausgeführt wird, deshalb umgeben wir ihn mit einem *synchronized* Block:

```
        synchronized (isThereAWinner) {
            if (!isThereAWinner[0]) {
                pause(200);
                isThereAWinner[0] = true;
                setFilled(true);
                setFillColor(Color.RED);
            }
        }
```

Das erfüllt den Zweck.

Challenges

MandelbrotSuperFast



In dem Special Topic "Graphics Performance" haben wir unser Mandelbrot Beispiel aus dem ersten Semester etwas beschleunigt. Die Frage ist aber geht es noch schneller? Die meisten CPUs haben ja inzwischen mehr als einen Core, Quadcores sind heute schon fast die Regel. Deswegen wollen wir hier unser Mandelbrot Programm parallelisieren.

Bevor wir aber damit beginnen, wollen wir erst einmal wissen wie viele Prozessoren uns denn zur Verfügung stehen:

```
int nrCPUs =
    Runtime.getRuntime().availableProcessors();
```

Falls wir ein etwas älteres oder billigeres Smartphone haben, und wir feststellen dass wir nur einen Prozessor haben, dann können wir uns die folgende Arbeit natürlich sparen, das wird nichts bringen.

Unser Programm ist nahezu identisch mit dem Mandelbrot Beispiel aus dem Special Topic Kapitel. Was sich aber ändert, ist wie wir die Arbeit aufteilen:

```
class MandelbrotView extends View {
    ...
    private final int NR_OF_THREADS = 16;
    private boolean[] isDone =
        new boolean[NR_OF_THREADS];
    private int[][] bitMapArray2;

    protected void onSizeChanged(int w, int h, int oldw, int oldh) {
        ...
        bitMapArray2 =
            new int[NR_OF_THREADS][mCanvasWidth *
                                   mCanvasHeight / NR_OF_THREADS];
    }
```

Was wir noch klären müssen: wie viele Threads sollen wir denn nehmen? Naiv würde man sagen, so viele wie wir CPU-Cores haben. Das geht allerdings davon aus, dass jeder Thread gleich viel Arbeit bekommt. Bei unserem Mandelbrot Beispiel (und bei vielen ähnlichen Beispielen) sind manche Streifen aber viel rechenintensiver als andere. Deswegen macht es Sinn, die Arbeit in noch kleinere Stücke aufzuteilen.

In unserem Beispiel wählen wir 16 Threads. Für jeden dieser Threads legen wir ein eigenes Bitmap Array an. Außerdem haben wir noch ein kleines boolsches Array *isDone*, mit dem wir feststellen können, ob schon alle fertig sind.

In der onDraw() Methode starten wir die 16 Threads, und jeder bekommt einen kleinen Streifen des Mandelbrotsets zur Bearbeitung:

```
public void onDraw(Canvas canvas) {
    double dy = (yMax - yMin) / NR_OF_THREADS;
    double y = yMin;
    for (int i = 0; i < NR_OF_THREADS; i++) {
        MandelbrotTask task =
            new MandelbrotTask(i, bitMapArray2[i]);
        task.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR,
                (double) mCanvasWidth,
                (double) mCanvasHeight / NR_OF_THREADS,
                xMin, xMax, y, y + dy);
        y += dy;
    }
```

Beim MandelbrotTask handelt es sich um einen AsyncTask. Der bekommt im Constructor seine *id* übergeben und den Teil des Bitmap Arrays an dem er arbeiten soll. Interessant ist hier, dass wir nicht die Methode *execute()* wie sonst aufrufen, sondern die Methode *executeOnExecutor()*. Diese ist zwingend notwendig, damit die AsyncTasks auch wirklich auf separaten Prozessoren laufen.

Über das *isDone[]* Array wissen wir, ob alle Threads fertig sind, so lange warten wir:

```
    // make sure all threads are done
    boolean done = false;
    while (!done) {
        done = true;
        for (int i = 0; i < isDone.length; i++) {
            if (!isDone[i]) {
                done = false;
            }
        }
        pause(2);
    }
```

Dann kleben wir die einzelnen Streifen wieder zusammen:

```
    int dHeight = mCanvasHeight / NR_OF_THREADS;
    int height = 0;
    for (int i = 0; i < NR_OF_THREADS; i++) {
        bitmap.setPixels(bitMapArray2[i], 0, mCanvasWidth,
                0, height, mCanvasWidth,
                mCanvasHeight / NR_OF_THREADS);
        height += dHeight;
    }
```

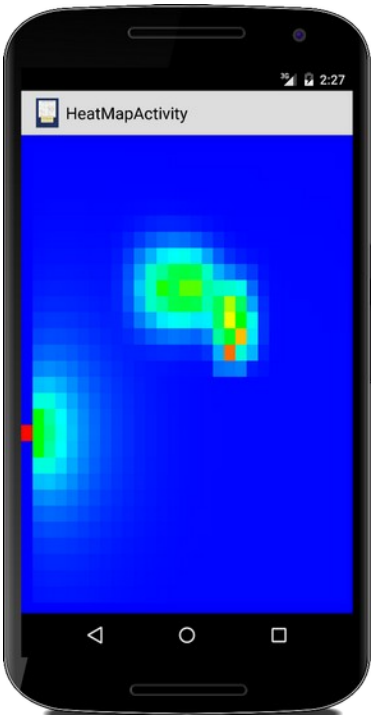
und zeichnen die Bitmap auf unserem Canvas:

```
    canvas.drawBitmap(bitmap, new Rect(0, 0,
            bitmap.getWidth(), bitmap.getHeight()),
            new Rect(0, 0, bitmap.getWidth(), bitmap.getHeight()),
            null);
    ...
}
```

Fehlt noch der MandelbrotTask: der besteht im Prinzip nur aus der *doInBackground()* Methode, in der wir die Mandelbrot Menge berechnen:

```
class MandelbrotTask extends AsyncTask<Double, Void, Long> {
    private int myIndex = -1;
    private int[] bitMapArray;

    public MandelbrotTask(int indx, int[] bitMapArray) {
        this.myIndex = indx;
        this.bitMapArray = bitMapArray;
    }
```

```
        protected Long doInBackground(Double... params) {
            long startTime = System.currentTimeMillis();

            int mCanvasWidth = (int) params[0].doubleValue();
            int mCanvasHeight = (int) params[1].doubleValue();
            double xMin = params[2];
            double xMax = params[3];
            double yMin = params[4];
            double yMax = params[5];

            // draw pixels in bitmap
            double xStep = (xMax - xMin) / mCanvasWidth * 1;
            double yStep = (yMax - yMin) / mCanvasHeight * 1;
            for (double x = xMin; x < xMax; x += xStep) {
                int i = (int)
                    (((x - xMin) * mCanvasWidth) /(xMax - xMin));
                for (double y = yMin; y < yMax; y += yStep) {
                    int j = (int)
                        (((y - yMin) * mCanvasHeight) /(yMax - yMin));
                    bitMapArray[j * mCanvasWidth + i] =
                        function(x, y);
                }
            }

            isDone[myIndex] = true;
            return (System.currentTimeMillis() - startTime);
        }

        private int function(double x0, double y0) {
            double x = 0.0;
            double y = 0.0;
            int iteration = 0;
            while (x * x + y * y < 4 && iteration < MAX_ITERATION) {
                double xtemp = x * x - y * y + x0;
                y = 2 * x * y + y0;
                x = xtemp;
                iteration++;
            }
            return RAINBOW_COLORS[
                iteration % RAINBOW_NR_OF_COLORS];
        }
    }
```



Interessant sind die Frameraten die wir jetzt erhalten, ja nach Anzahl der Prozessoren die unser Smartphone hat, erhöht sich die Geschwindigkeit proportional. Wenn wir jetzt noch zoomen könnten, wäre es perfekt.

HeatMap

Bei der HeatMap Activity geht es darum die Wärmeleitungsgleichung [6] zu visualisieren. Die Wärmeleitungsgleichung beschreibt wie sich die Temperatur orts- und zeitabhängig ändert. Das hört sich vielleicht kompliziert an, sieht aber cool aus.

Als erstes zerlegen wir unseren Raum, also den Bildschirm, in lauter kleine Rechtecke, unser *data* Array:

```
public class HeatMapActivity extends Activity
                            implements Runnable {
    private float[][] data =
        new float[NR_ROWS][NR_COLUMNS];
```

Dann müssen wir ein paar Konstanten definieren

```
    public static final int DELAY = 50;
    public static final int NR_ROWS = 30;
    public static final int NR_COLUMNS = 30;
    public static final int DATA_MAX_VALUE = 100;
    public static final float TIME_DISSIPATION_FACTOR = 0.9f;
    public static final float SPACE_DISSIPATION_FACTOR = 0.1f;
```

Die ersten drei sind selbsterklärend, DATA_MAX_VALUE ist der maximale Wert den unsere *data* Werte annehmen können, und die zwei letzten Konstanten kommen von der Heatequation. Die können wir beliebig setzen, aber bei den vorgegebenen Werten sieht es am hübschesten aus. Und wir benötigen noch einen View,

```
    private HeatMapView gv;
```

da wir das Ganze ja visualisieren wollen. In der onCreate() initialisieren wir den View und starten uns als Thread:

```
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        gv = new HeatMapView(this, data);
        setContentView(gv);

        new Thread(this).start();
    }
```

Die run() Methode tut fast nichts:

```
    public void run() {
        while (true) {
            applyHeatDissipation();
            pause(DELAY);
        }
    }
```

Die ganze Arbeit passiert in der *applyHeatDissipation()* Methode, effektiv ist das die Wärmeleitungsgleichung in Java, verstehen müssen wir das nicht:

```
    private void applyHeatDissipation() {
        float[][] data2 = new float[NR_ROWS][NR_COLUMNS];

        for (int i = 1; i < NR_COLUMNS - 1; i++) {
            for (int j = 1; j < NR_ROWS - 1; j++) {
                float data0 = data[i][j];
                float delta = 0;
                for (int m = -1; m < 2; m++) {
                    for (int n = -1; n < 2; n++) {
                        float d = data[i + m][j + n] - data0;
                        if (d > 0) {
                            delta += d;
                        }
                    }
                }
                data2[i][j] = delta * SPACE_DISSIPATION_FACTOR;
            }
        }
```

```
            for (int i = 1; i < NR_COLUMNS - 1; i++) {
                for (int j = 1; j < NR_ROWS - 1; j++) {
                    data[i][j] = data[i][j] * TIME_DISSIPATION_FACTOR;
                    data[i][j] += data2[i][j];
                    if (data[i][j] < 0) {
                        data[i][j] = 0;
                    }
                }
            }
        }
```

Fehlt noch der View und wir sind fertig:

```
        class HeatMapView extends View {
            private float[] hsv = { 240f, 1f, 1f };
            private float[][] data;
            private Paint cPaint;

            public HeatMapView(Context context, float[][] data) {
                super(context);
                this.data = data;
                cPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
                cPaint.setStyle(Paint.Style.FILL);
            }

            @Override
            protected void onDraw(Canvas canvas) {
                float dwidth = getWidth() / NR_COLUMNS + 1;
                float dheight = getHeight() / NR_ROWS + 1;
                for (int i = 0; i < NR_COLUMNS; i++) {
                    for (int j = 0; j < NR_ROWS; j++) {
                        float x = i * dwidth;
                        float y = j * dheight;
                        hsv[0] = (DATA_MAX_VALUE - data[i][j]) * 240
                                                        / DATA_MAX_VALUE;
                        int col = Color.HSVToColor(hsv);
                        cPaint.setColor(col);
                        canvas.drawRect(x, y, x + dwidth, y + dheight,
                                                                cPaint);
                    }
                }
            }
        }
```

Der View stellt einfach nur das *data[]* Array bildlich dar.

Wenn wir das so laufen lassen, passiert noch nichts, alles was wir sehen ist ein blauer Bildschirm. Was noch fehlt ist eine Hitzequelle, und die erzeugen wir über den TouchEvent in der Activity:

```
        public boolean onTouchEvent(MotionEvent event) {
            float dwidth =
                this.getWindow().getDecorView().getWidth() /NR_COLUMNS +1;
            float dheight =
                this.getWindow().getDecorView().getHeight() /NR_ROWS +1;
            int i = (int) (event.getX() / dwidth);
            int j = (int) (event.getY() / dheight);
            data[i][j] = 100;
            return super.onTouchEvent(event);
        }
```

Hübsch, oder?

Research

Zu Threading gibt es ganze Bücher, die sich nur diesem Thema widmen. Soll heißen, was wir hier besprochen haben, ist nur eine erste, oberflächliche Einführung. Man kann sich aber wenigstens das Dining Philosophers Problem mal etwas näher ansehen.

Dining Philosophers Problem

Unser alter Freund, Herr Dijkstra, hat sich das Problem der Abendessenden Philosphen als Hausaufgabe für seine Studierenden ausgedacht [1]. Die ursprüngliche Version ist etwas komplizierter als meine, deswegen sollten wir uns das Original mal durchlesen.

Fragen

1. Geben Sie ein Beispiel wofür man einen AsyncTask verwenden könnte. Was unterscheidet ihn von normalen Threads?

2. In der Vorlesung haben wir uns die Problematik des Multithreading mit einem Beispiel verdeutlicht. In dem Beispiel rannten Pferde (durch schwarze Rechtecke dargestellt) um die Wette. Jenes Pferd welches als erstes über die Ziellinie rannte, vollführte einen Siegestanz (es wurde rot anstelle von schwarz). Meist hat das auch funktioniert, aber manchmal kam es zu einem Fehler der mit Multithreading zu tun hatte. Wie machte sich der Fehler bemerkbar? Wie haben wir das Problem gelöst?

3. Betrachten Sie die Klasse 'Horse':

   ```
   public class Horse {
       ...
   }
   ```

   Nehmen Sie an, dass die Klasse bereits komplett implementiert wurde. Ihre Aufgabe besteht lediglich darin, die Klasse so zu modifizieren, dass sie als separater Thread laufen kann. Welches Interface müssen Sie implementieren und welche Methode müssen Sie überschreiben?

4. In der Vorlesungen haben wir das Problem der abend-essenden Philosophen behandelt (dining philosophers). Erinnern Sie sich an die Geschichte, und benutzen Sie die Analogie um zu erklären was eine 'Race Condition' (auf deutsch 'kritischer Wettlauf') ist, was 'Starvation' (auf deutsch 'Verhungern') bedeutet und was man unter 'Dead Lock' (auf deutsch 'Verklemmung') versteht.

5. Was ist der Unterschied zwischen einem *TimerTask* und einem *AsyncTask*?

6. Wenn ein Thread mal tot ist, können Sie ihn dann wiederbeleben?

7. Es gibt zwei Möglichkeiten aus einer beliebigen Klasse einen Thread zu machen. Die erste ist einfach von der Thread Klasse zu vererben:

```
public class MyFirstThread extends Thread {
    public void run() {
        ...
    }
}
```

   Die zweite ist das Runnable Interface zu implementieren:

```
public class MySecondThread implements Runnable {
    public void run() {
        ...
    }
}
```

   Welche ist zu bevorzugen und warum?

8. Was ist die generelle Idee hinter dem Producer-Consumer Pattern?

.

Referenzen

[1] Dining philosophers problem, https://en.wikipedia.org/wiki/Dining_philosophers_problem

[2] Producer–consumer problem, https://en.wikipedia.org/wiki/Producer–consumer_problem

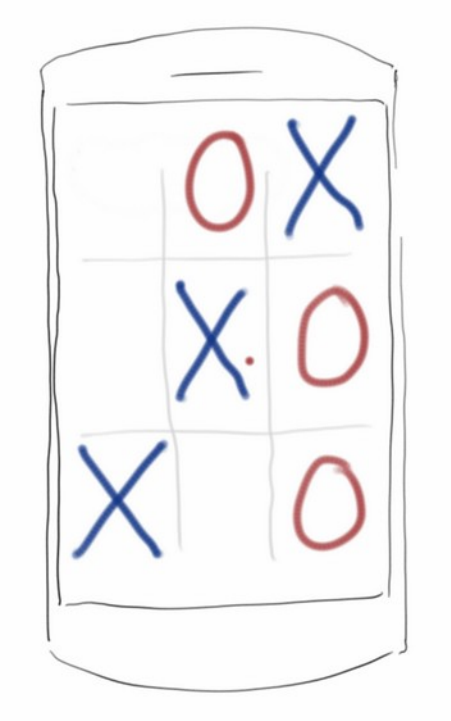
[3] Interface BlockingQueue<E>, https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html

[4] Speed reading, https://en.wikipedia.org/wiki/Speed_reading

[5] Programming Methodology, CS106A, von Mehran Sahami, https://see.stanford.edu/Course/CS106A

[6] Heat equation, https://en.wikipedia.org/wiki/Heat_equation

Networking



Was wäre ein Smartphone ohne eine Internetverbindung? Ziemlich langweilig wenn man nicht surfen, whatsappn oder Online spielen könnte. Deswegen schreiben wir in diesem Kapitel einen einfachen Browser, wir laden Webseiten herunter und auch ein eigener Server ist auf dem Programm. Mit einer kleinen Chatapplikation lernen wir auch wofür man TCP und UDP verwenden kann und was der Unterschied ist. JSON steht auf dem Menü und wir implementieren einen Wifi- und einen Netzwerkscanner. Dann verbringen wir noch ein bisschen Zeit mit Bluetooth. Als Schmankerl gibts dann unser TicTacToe als Netzwerkspiel und wir steuern unseren Laptop mit dem Handy fern.

In diesem Kapitel wird es auch ziemlich threadig, soll heißen, dass fast jedes Programm mit Threads arbeitet. Es kann also nix schaden ab und zu mal im Kapitel fünf nachzusehen wenn was unklar ist. Sobald man Netzwerksachen macht, benötigt man viele Permissions. Will man sich es einfach machen, dann erlaubt man mal alles was irgendwo mal gebraucht werden könnte:

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.BLUETOOTH" />
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
<uses-permission android:name="android.permission.NFC" />
<uses-permission android:name=
                            "android.permission.ACCESS_WIFI_STATE" />
<uses-permission android:name=
                            "android.permission.CHANGE_WIFI_STATE" />
<uses-permission android:name="android.permission.WRITE_SETTINGS" />
```

Das ist natürlich nicht besonders nutzerfreundlich, aber wenn WhatsApp das kann, können wir das alle mal. Wir machen das ja nur aus didaktischen Gründen, WhatsApp aus wirtschaftlichen. Ach ja, für alle Beispiele in diesem Kapitel muss natürlich Wifi eingeschaltet sein.

WebView

Als erstes wollen wir ein bisschen im Web browsen. Wie das mit einem Intent geht, haben wir bereits im ersten Kapitel gesehen. Hier wollen wir einen *WebView* verwenden. Dabei handelt es sich um einen View der HTML und CSS darstellen kann, und der auch bei Androids hauseigenem Browser verwendet wird. Neben einem EditText und Button Widget, fügen wir den WebView in unsere Layoutdatei ein:

```
<LinearLayout ... >
    ...
    <WebView
        android:id="@+id/webView"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
</LinearLayout>
```

Wenn wir nur möchten, dass der WebView eine Webseite lädt und anzeigt, dann genügen die Zeilen,

```
WebView webView = (WebView)
findViewById(R.id.webView);
webView.loadUrl("http://www.google.com/");
```

Sobald der Nutzer aber auf irgendeinen Link klicken würde, dann würde Android den Default-Browser öffnen und dort den Link anzeigen. Um das zu verhindern, müssen wir die *setWebViewClient()* Methode des WebViews aufrufen und ihr einen WebViewClient übergeben:

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.webview_activity);

    final EditText et = (EditText) findViewById(R.id.editText);

    final WebView webView = (WebView) findViewById(R.id.webView);
    webView.setWebViewClient(new WebViewClient());

    Button btn = (Button) findViewById(R.id.btn);
    btn.setOnClickListener(new View.OnClickListener() {
```

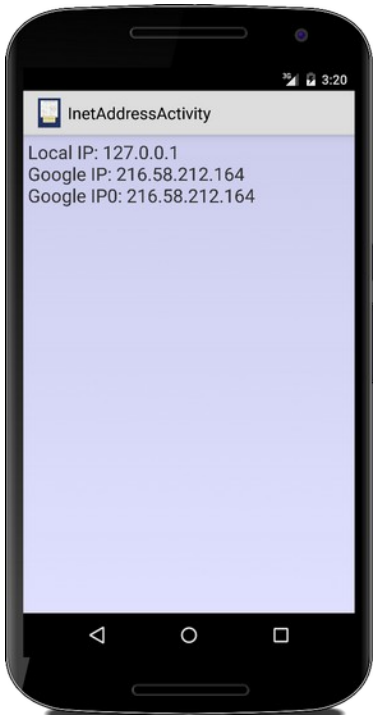
```
            @Override
            public void onClick(View v) {
                String sUri = et.getText().toString();
                if (!sUri.startsWith("http://")) {
                    sUri = "http://" + sUri;
                }
                webView.loadUrl(sUri);
                webView.requestFocus();
            }
        });
    }
```

Im Unterschied zu dem Browserbeispiel aus dem ersten Kapitel, benötigt unser Beispiel noch die Erlaubnis ins Internet gehen zu dürfen. Das erledigen wir durch die folgende Zeile in unserer AndroidManifest Datei:

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

Interessant ist vielleicht noch, dass wir die Klasse WebViewClient auch erweitern können, und dann das Verhalten unseres Browsers voll unter Kontrolle haben. Wie das geht haben wir im Projekt *HelpPages* im vierten Kapitel gesehen.

InetAddress



Wenn wir im Browser "www.google.com" eingeben, dann wissen wir, dass wir auf den Google Server wollen. Unser Smartphone kann aber erst mal mit der Domain "www.google.com" nicht viel anfangen [1]. Es braucht die IP Adresse. Dieses Umwandeln von Domain nach IP Adresse und umgekehrt macht in Java die *InetAddress* Klasse:

```
String networkInfo = "";
InetAddress localAdr = InetAddress.getLocalHost();
networkInfo += "Local IP: " +
localAdr.getHostAddress();

InetAddress remoteAdr =
InetAddress.getByName("www.google.com");
networkInfo += "\nGoogle IP: " +
remoteAdr.getHostAddress();

InetAddress[] remoteAdrs =
InetAddress.getAllByName("www.google.com");
for (int i = 0; i < remoteAdrs.length; i++) {
    networkInfo += "\nGoogle IP" + i + ": ";
    networkInfo += remoteAdrs[i].getHostAddress();
}
```

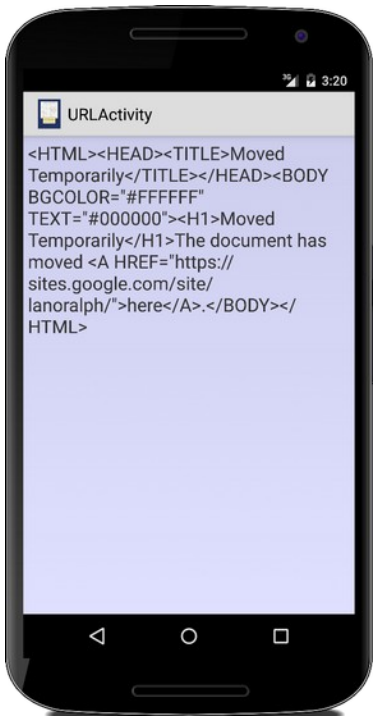
Für einen Reverse-DNS-Lookup verwendet man die *getHostName()* Methode, die macht aus einer IP Adresse wieder einen Domainnamen. Braucht man eher selten.

Damit der obige Code ohne weiteres funktioniert, müssen wir zwei Dinge tun: zunächst müssen wir wieder die Erlaubnis haben ins Internet gehen zu dürfen, aber wir müssen zusätzlich noch eine *ThreadPolicy* setzen [2]:

```
StrictMode.ThreadPolicy policy =
    new StrictMode.ThreadPolicy.Builder().permitAll().build();
StrictMode.setThreadPolicy(policy);
```

Tun wir das nicht, bekommen wir eine *NetworkOnMainThreadException*. Aus irgendeinem Grund hat es Android nicht so gerne, dass wir aus dem Main Thread heraus auf's Netzwerk zugreifen. Natürlich kann man jetzt immer einen Extra-Thread für's Netzwerk starten. Unsere Beispiele würden dadurch aber so unübersichtlich und unlesbar, dass wir aus didaktischen Gründen darauf verzichten werden. Deswegen müssen wir in all unseren Programmen immer diese zwei Zeilen am Anfang der onCreate() einfügen.

URL



Sehr häufig wollen wir nur eine simple HTML Seite runterladen. Dafür verwendet man die *URL* Klasse. Die URL Klasse erlaubt es uns eine HTTP-Verbindung (connection) zu einem Webserver aufzubauen, und sie liefert uns einen InputStream:

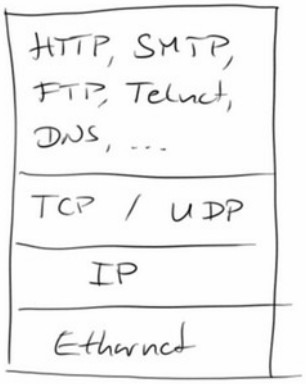
```
public static String getWebpage(String address) {
    try {
        URL url = new URL(address);
        HttpURLConnection con =
          (HttpURLConnection) url.openConnection();
        InputStream is = con.getInputStream();
        BufferedReader br = new BufferedReader(
            new InputStreamReader(is, "UTF-8"));
        String content = "";
        while (true) {
            String line = br.readLine();
            if (line == null)
                break;
            content += line;
        }
        br.close();
        con.disconnect();

        return content;

    } catch (Exception e) {
        e.printStackTrace();
    }
    return null;
}
```

Dieser InputStream ist der gleiche wie wir ihn vom Lesen von Dateien her kennen, dort war es ein FileInputStream. Deswegen ist der Rest des Codes auch identisch mit dem aus der EditorActivity im vierten Kapitel. Verwenden können wir unsere Methode dann ganz einfach:

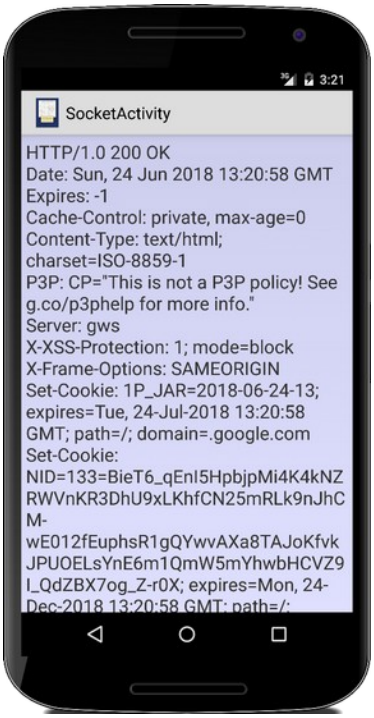
```
String html = getWebpage("http://www.lano.de"));
```

Mit der URL Klasse kann man noch viel mehr machen als einfach nur Dateien runterzuladen, aber für den Anfang soll das genügen.

Noch mal ganz kurz zur Erinnerung: In der Internetprotokollfamilie [29] gibt es effektiv vier Schichten, dabei ist die unterste die Hardwareschicht. Darüber folgt die IP Schicht mit der sich zwei Computer im Internet finden. In der TCP/UDP Schicht (Transportschicht) wird dann zwischen einem verbindungsorientierten oder verbindungslosen Protokoll gewählt. In der Anwendungsschicht wird schließlich das Protokoll gewählt mit dem sich die beiden Computer unterhalten wollen.



Socket



Die URL Klasse baut eine TCP Verbindung auf, lädt die gewünschte Seite von einem Server, und schließt die Verbindung wieder. Wenn wir nur eine Seite laden wollen dann genügt das auch. Manchmal möchte man aber mehr als eine Seite laden, oder man möchte die Verbindung offen halten. Dann benötigen wir einen *Socket*, genauer eine TCP Socket [3].

Die Socket Klasse ist relativ einfach zu verwenden: man gibt ihr eine IP Adresse und einen Port [4]. Dann sagt man ihr, dass sie sich verbinden soll:

```
SocketAddress sockaddr = new
InetSocketAddress("www.google.com", 80);
Socket socket = new Socket();
socket.connect(sockaddr, TIMEOUT);
Log.i("SocketActivity",
    ""+socket.getLocalAddress());
Log.i("SocketActivity",
    ""+socket.getRemoteSocketAddress());
```

Wenn es einen interessiert, kann man mit *getLocalAddress()* die eigene Adresse erfahren und mit *getRemoteSocketAddress()* die Adresse des Servers.

Steht unsere TCP Verbindung, können wir mit dem Server reden. Wir müssen natürlich die Sprache des Servers sprechen. Der Google Server spricht HTTP [5], und deswegen schreiben wir in den OutputStream:

```
OutputStream os = socket.getOutputStream();
os.write("GET / \r\n".getBytes());
os.flush();
```

Das "GET / \r\n" sagt soviel wie "Gib mir doch Deine Einstiegsseite". Die *flush()* Methode schickt die Daten schon mal los, sonst würde der Socket nämlich warten, ob wir noch mehr zu sagen haben.

Jetzt antwortet uns der Server, deswegen müssen wir zuhören, und das machen wir mit dem InputStream, den wir vom Socket bekommen:

```
InputStream is = socket.getInputStream();
while (true) {
    int len = is.read();
    if (len == -1)
        break;
    tv.append(""+(char) len);
}
```

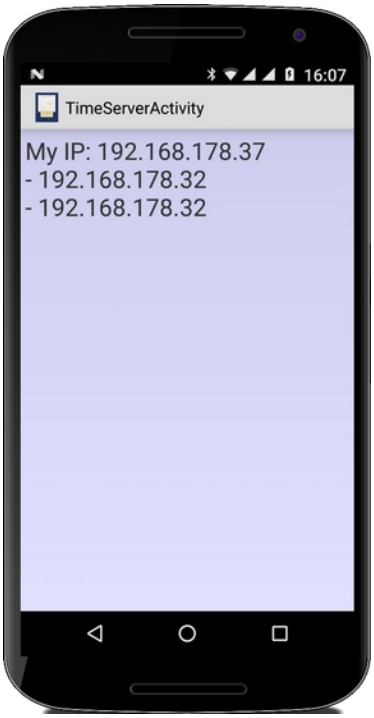
Wir lesen vom Socket solange bis nichts mehr kommt. Wir könnten jetzt noch eine zweite Anfrage schicken, und so weiter.

Wenn wir dann fertig sind, müssen wir alle Türen wieder zu machen, und derer sind drei:

```
os.close();
is.close();
socket.close();
```

Und das wars. Natürlich muss man um den ganzen Code einen dicken try-catch Block schreiben, weil da kann ja alles mögliche schief laufen.

TimeServer



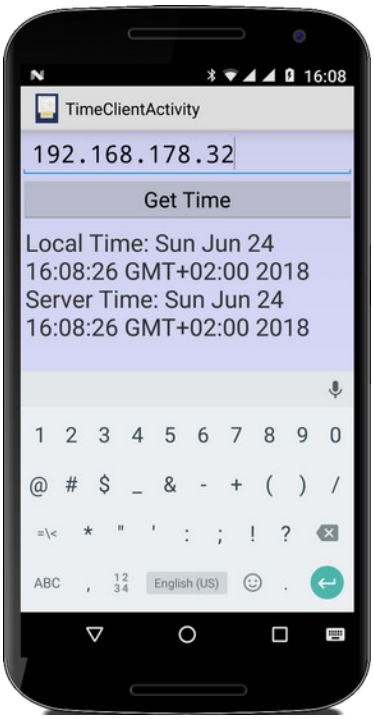
Nach dem Client Socket wollen wir uns jetzt den Server Socket ansehen, der ist auch nicht viel schwieriger. Wir wollen einen kleinen Server schreiben, der aktuelles Datum und Uhrzeit ausgibt. Dazu benötigen wir erst einmal einen *ServerSocket*:

```
ServerSocket server = new ServerSocket(3737);
while (true) {
    Socket socket = server.accept();

    OutputStream os = socket.getOutputStream();
    String daytime = new Date().toString();
    os.write(daytime.getBytes());

    os.close();
    socket.close();
}
```

Der hört auf einem bestimmten Port, in unserem Fall 3737. Dann wartet der Server in der *accept()* Methode solange bis jemand etwas von ihm will. Man nennt das auch einen "blocking" Call. Versucht jetzt irgend ein Client sich mit dem Server zu verbinden, dann liefert die *accept()* Methode einen ganz normalen Client Socket als Rückgabewert. Mit dem können wir wie im Socket Beispiel oben arbeiten. In diesem Fall schicken wir einfach Datum und Uhrzeit zurück.

Bei Servern ist es üblich, da sie blockieren, dass man sie in einem separaten Thread startet:

```
public class TimeServerActivity extends Activity implements Runnable {

    public void onCreate(Bundle savedInstanceState) {
        ...
        Thread th = new Thread(this);
        th.start();
    }

    public void run() {
        ... server code ...
    }
}
```



Und das ist der Teil, der Server etwas komplizierter macht, aber nicht viel.

TimeClient

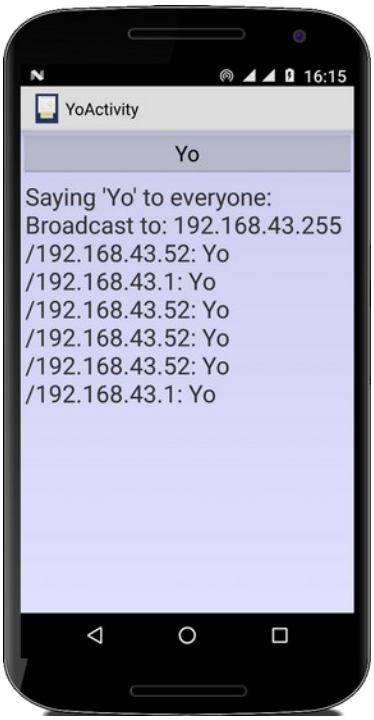
Jetzt haben wir einen Server, fehlt natürlich noch der Client. Der ist fast identisch zu unserem vorigen Socket Beispiel:

```
String daytime = "";
SocketAddress sockaddr =
    new InetSocketAddress(IP, 3737);
Socket socket = new Socket();
socket.connect(sockaddr, TIMEOUT);

InputStream is = socket.getInputStream();
while (true) {
    int len = is.read();
    if (len == -1)
        break;
    daytime += (char) len;
}
```

```
is.close();
socket.close();
```

Hypothetisch könnte man auch einen Browser als Client nehmen, aber die Browser sprechen nur HTTP. Dazu später mehr.

Yo



Neben dem TCP Protokoll gibt es auch noch das UDP Protokoll [6]. Da das UDP Protokoll nicht verbindungsorientiert ist, ist es sogar einfacher als das TCP Protokoll. Und, mit dem UDP Protokoll kann man auch Broadcasts versenden, soll heißen man kann eine Nachricht an alle schicken.

In unserer Yo Anwendung [7] wollen wir ein einfaches "Yo" per UDP Broadcast versenden. Dazu benötigen wir einen *DatagramSocket*, also einen UDP Sockel:

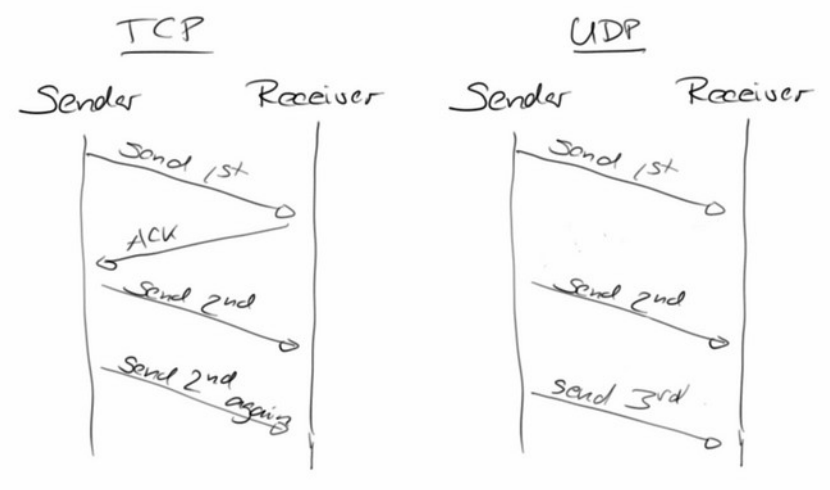
```
private void sendYoToEveryone() {
    try {
        byte[] data = "Yo".getBytes();
        DatagramPacket theOutput =
            new DatagramPacket(data, data.length,
                Util.getLocalBroadcastAddress(),
                YO_PORT);
        DatagramSocket theSocket =
            new DatagramSocket();
        theSocket.send(theOutput);

    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Über den schicken wir dann ein *DatagramPacket*, also ein UDP Datenpaket. Man kann das entweder an eine bestimmte IP Adresse schicken, oder eben als Broadcast, das macht die Methode *getLocalBroadcastAddress()* der Util Klasse. DatagramPackets können nicht beliebig groß sein: maximal gehen 65507 Byte, garantiert sind aber nur 512 Byte. Das hat mit dem evtl. Umverpacken von Datenpacketen in Routern, Bridges und Switches zu tun.

Kommen wir zum UDP Server: auch der sollte wieder als separater Thread laufen. Im Gegensatz zu TCP, wird bei UDP für Client und Server der gleiche Sockel verwendet. Der einzige Unterschied ist, dass wir im ersten Fall die Methode *send()* verwenden und im zweiten die Methode *receive()*:

```
public void run() {
    byte[] buffer = new byte[MAX_PACKET_SIZE];
    DatagramSocket server = new DatagramSocket(YO_PORT);
    DatagramPacket packet = new DatagramPacket(buffer, buffer.length);

    while (true) {
        server.receive(packet);
        String yoMessage =
            new String(packet.getData(), 0, packet.getLength());
        final String msg = "" + packet.getAddress() + ": "+ yoMessage;

        // reset the length for the next packet
        packet.setLength(buffer.length);
    }
}
```

Im Gegensatz zur *accept()* Methode, blockiert die *receive()* Methode nicht. D.h. es kann durchaus passieren, dass uns das eine oder andere Packet entwischt, wenn wir nicht gerade zuhören. Das ist der Nachteil von UDP.



Zwei Anmerkungen noch: es stellt sich heraus, dass in einem kabelgebunden Netzwerk so gut wie keine UDP Packet verloren gehen, komischerweise verschwinden aber überraschend viele UDP Packete wenn sie über Wifi geschickt werden. Und leider gibt es einige Smartphones die keine UDP Broadcasts können, bzw. diese unterbinden, z.B. einige Samsung Handys tun sich da anscheinend schwer.

Review

Wir haben die Grundlagen der Netzwerkprogrammierung mit Android Geräten gelegt. Mit dem WebView Widget können wir HTML Seiten anzeigen, mit der URL Klasse Webseiten herunterladen, und mit der InetAddress Klasse DNS Anfragen verschicken. Dann haben wir gesehen wie wir mit der Socket und der ServerSocket Klasse TCP Verbindungen herstellen können, sowohl clientseitig als auch serverseitig. Und im letzten Beispiel haben wir UDP Packete verschickt und empfangen mit Hilfe der DatagramPacket und DatagramSocket Klassen.

Projekte

Was wir bisher gesehen haben, haut einen noch nicht so vom Hocker. Aber wir haben die Grundlagen gelegt für eine ganze Reihen von interessanten Anwendungen. Dazu gehören Netzwerkscanner, Server- und Chatanwendungen, ein bisschen Bluetooth und natürlich ein Spiel. Am coolsten ist aber wahrscheinlich die RemoteDesktopClient Anwendung.

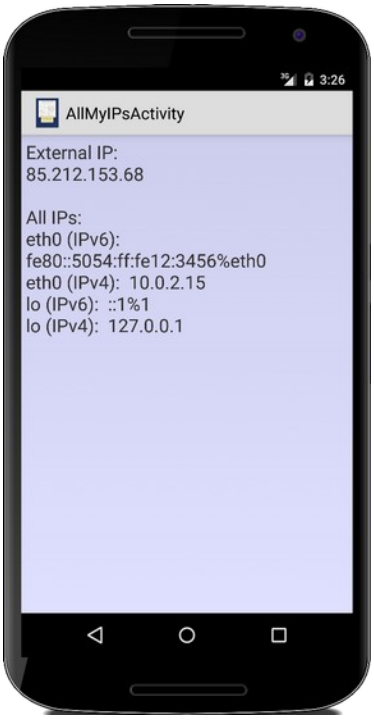
NetworkScanner

Eine interessante Methode der InetAddress Klasse ist die *isReachable()* Methode: diese schickt einen ICMP Request an eine bestimmte Adresse, macht also eine 'ping' Anfrage [8]. Wir können uns das für einen kleine Network Scanner zu Nutze machen:

```
public void run() {
    ...
    InetAddress myIP = Util.getMyLocalIpAddress();
    byte[] localAddresses = myIP.getAddress();
    for (int i = 0; i < 256; i++) {
        localAddresses[3] = (byte) i;
        final InetAddress address =
          InetAddress.getByAddress(localAddresses);

        if (address.isReachable(TIMEOUT)) {
            Log.i("NetworkScannerActivity",
                  address.getHostAddress());
        }
    }
    ...
}
```

In dem Code oben gehen wir einfach alle lokalen Adressen durch und schauen ob irgendjemand antwortet. Danach wissen wir welche Rechner es in unserem lokalen Netz gibt (falls diese auf einen 'ping' antworten). Interessanterweise scheint das nur im lokalen Netz zu funktionieren.

AllMyIPs



Solange wir als Client unterwegs sind, kann es uns eigentlich egal sein, was unsere IP Adresse ist. Wenn wir allerdings einen Server auf unserem Smartphone hosten wollen, dann müssten wir schon wissen was unsere IP ist, denn sonst kann sich ja niemand mit unserem Server verbinden.

Stellt sich heraus, die meisten Smartphones haben mehr als eine IP Adresse. Das kommt daher, dass man sich ja einmal über Wifi mit dem Internet verbinden kann, aber natürlich auch über das mobile Datennetz. Deswegen müssen wir uns zunächst alle Netzwerkkarten (nics) unseres Gerätes geben lassen, und dann von jeder dieser Karten die IP Adressen:

```
private String getAllLocalIpAddresses() {
    String sIPs = "";
    ...
    for (Enumeration<NetworkInterface> nics =
         NetworkInterface.getNetworkInterfaces();
         nics.hasMoreElements();) {

        NetworkInterface nic = nics.nextElement();

        for (Enumeration<InetAddress> ips =
             nic.getInetAddresses();
             ips.hasMoreElements();) {
            InetAddress ip = ips.nextElement();
            sIPs += nic.getName();
            if (InetAddressUtils.isIPv4Address(ip.getHostAddress())) {
                sIPs += " (IPv4):";
            } else {
                sIPs += " (IPv6):";
            }
            sIPs += "  " + ip.getHostAddress() + "\n";
        }
```

```
        }
        ...
        return sIPs;
    }
```

Je nachdem wie wir uns mit dem Internet verbinden, liefert diese Methode verschiedene Resultate.

Interessant ist allerdings, dass es sich bei all diesen Adressen um lokale Adressen im Sinne von IP handelt, d.h. sie beginnen entweder mit "192.168.y.z" oder mit "10.x.y.z". Wenn wir unsere wirkliche externe IP Adresse erfahren wollen, müssen wir einen Server draussen im Internet fragen:

```
    private String getMyExternalIP() {
        String webpage =
    Util.getWebpage("http://wikimusicapp.appspot.com/myip");
        if (webpage != null) {
            String[] words = webpage.split(" ");
            return words[3];
        }
        return "No external IP available";
    }
```

Wenn ich das auf meinem Handy laufen lasse, dann ist meine externe IP Adresse die "80.187.122.215". Wir können nachsehen, wem die Adresse gehört. Es stellt sich heraus, dass das eine Adresse der Telekom ist, also unseres mobilen Dienstleisters. Im Gegensatz zu unserem DSL oder Kabelanschluss, teilen wir die Adresse mit vielen anderen Leuten. Deswegen können wir leider keinen externen Server auf unseren mobilen Endgeräten hosten. Wenn wir uns allerdings auf Wifi und das lokale Netzwerk beschränken geht das sehr wohl.

TimeClientNIST

Will man die genaue Uhrzeit wissen, dann kann man beim National Institute of Standards and Technology (NIST) nachfragen [9]. Die betreiben nämlich unter der Adresse "time.nist.gov" einen Server, der sowohl das Time Protocol (RFC-868) auf Port 37 als auch das Daytime Protocol (RFC-867) auf Port 13 zur Verfügung stellt [10]. Mit einer kleinen Modifikation können wir unsere SocketActivity dafür verwenden:

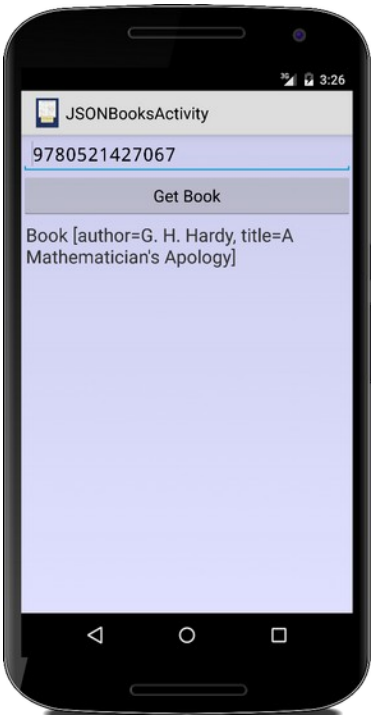
```
    private long getTimeFromNIST() {
        long time = 0;
        ...
        SocketAddress sockaddr =
            new InetSocketAddress("time.nist.gov",
                                  PORT_TIME);
        Socket socket = new Socket();
        socket.connect(sockaddr, TIMEOUT);

        InputStream is = socket.getInputStream();

        byte[] buffer = new byte[4];
        int len = is.read(buffer, 0, 4);
        time = 4294967296l +
                  Util.byteArrayToBigEndianInt(buffer);

        is.close();
        socket.close();
        ...
        return time;
    }
```

Die Zahl die wir bekommen ist die Zeit in Sekunden, die seit dem 1. Januar 1900 vergangen ist. Wenn wir das mit unserer lokalen Zeit vergleichen wollen,

```
private long getLocalTime() {
    long localTime = (new Date().getTime() -
                      new Date(0, 0, 1).getTime()) / 1000;
    return localTime;
}
```

müssen wir noch 3600 abziehen, da sich die Zeit auf UTC bezieht [11].

JSONBooks



Viele Webservices im Netz verwenden die JavaScript Object Notation, kurz JSON [12], so auch Google Books [13]. Wenn wir im Browser auf folgende Adresse gehen,

```
https://www.googleapis.com/books/v1/volumes?
q=isbn:9780521427067
```

dann bekommen wir folgendes JSON zu sehen:

```
{
 "kind": "books#volumes",
 "totalItems": 1,
 "items": [
  {
   "kind": "books#volume",
   ...
   "volumeInfo": {
    "title": "A Mathematician's Apology",
    "authors": [
     "G. H. Hardy"
    ],
   }
   ...
  }
 ]
}
```

Man kann das Ganze jetzt von Hand parsen oder man kann es sich etwas einfacher machen und die Java Klasse *JSONObject* verwenden. Zunächst einmal laden wir das JSON mit der URL Klasse herunter:

```
String url =
    "https://www.googleapis.com/books/v1/volumes?q=isbn:" + isbn;
String json = Util.getWebpage(url);
Book bk = new Book(json);
```

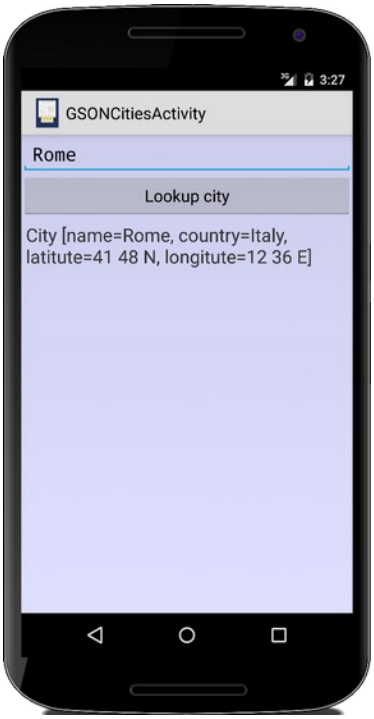
Im Constructor der Book Klasse, verwenden wir dann die JSONObject Klasse:

```
class Book {
    private String title;
    private String author;

    public Book(String json) {
        try {
            JSONObject jsonObject = new JSONObject(json);
            JSONArray items = jsonObject.getJSONArray("items");
            JSONObject book1 = (JSONObject) items.get(0);

            JSONObject volumeInfo =
                (JSONObject) book1.get("volumeInfo");
```

```
                title = volumeInfo.getString("title");

                JSONArray authors = volumeInfo.getJSONArray("authors");
                author = (String) authors.get(0);

            } catch (JSONException e) {
                e.printStackTrace();
            }
        }

        public String toString() {
            return "Book [author=" + author + ", title=" + title + "]";
        }
    }
```

Wir müssen natürlich grob wissen, wie unser JSON aufgebaut ist, aber ansonsten geht das Parsen relativ einfach von der Hand.

GSONCities



Noch einfacher geht das Ganze mit der *Gson* Klasse von Google [14]. Dazu schauen wir das Cities Beispiel aus dem fünften Semester noch einmal an. Als Webservice können wir es unter

```
http://wikimusicapp.appspot.com/cities?city=Rome
```

erreichen. Dieser Webservice liefert uns das folgende JSON:

```
{'country':'Italy','name':'Rome',
 'latitute':'41 48 N','longitute':'12 36 E'}
```

Wenn wir nun eine Klasse *City* wie folgt deklarieren (die Instanzvariablen müssen identisch mit den Bezeichnern im JSON sein),

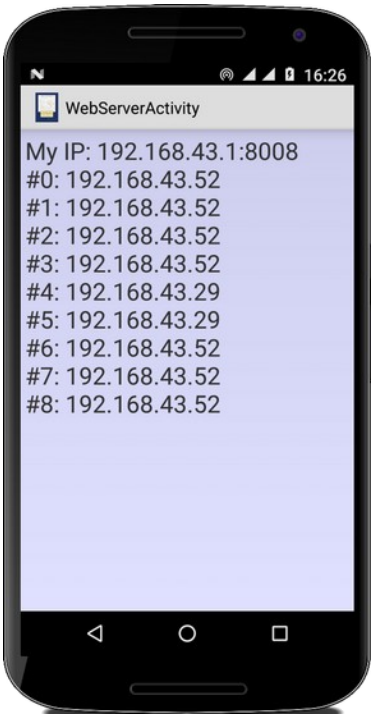
```
class City {
    private String country;
    private String name;
    private String latitute;
    private String longitute;

    public City() {
    }

    public String toString() {
        ...
    }
}
```

dann erlaubt uns Gson ohne große Umschweife daraus eine Klasse zu machen:

```
String url =
    "http://wikimusicapp.appspot.com/cities?city=" + cityName;
String json = Util.getWebpage(url);

Gson gson = new Gson();
City city = gson.fromJson(json, City.class);
```

Einfacher geht's nicht.

WebServer



Inzwischen haben wir alles was nötig ist, um einen eigenen Webserver zu schreiben. Wie beim TimeServer verwenden wir einen ServerSocket. Idealerweise sollte der auf dem HTTP Port 80 hören, aber auf Linux Systemen (und Android ist ein Linux System) darf nur Root auf Ports unterhalb von 1024 zugreifen. Deswegen verwenden wir irgendeinen Port der größer ist. Den ServerSocket starten wir, wie gehabt, in einem separaten Thread:

```
public void run() {
    ...
    int threadNr = 0;
    ServerSocket server = new ServerSocket(8008);
    while (isRunning) {
        Socket socket = server.accept();
        (new ConnectionThread(++threadNr,
                              socket)).start();
    }
    server.close();
    ...
}
```

Neu ist jetzt, dass wir einen zweiten Thread starten, den ConnectionThread, sobald sich jemand mit uns verbindet. Der Grund dafür ist, dass unser Server ja mehr als nur eine Seite und mehr als nur einen Client bedienen soll. Wenn also ein Request reinkommt, dann lassen wir einen neuen Thread diesen Request beantworten, der Server selbst kann aber sofort wieder auf neue Requests antworten. Andernfalls, müssten nämlich neue Requests immer erst mal warten bis der alte fertig ist. Das Internet wäre so ziemlich nutzlos, wenn man bei Amazon mit seinem Einkauf warten müsste bis der andere fertig ist. Da kann ich ja gleich in den Laden um die Ecke gehen und dort in der Schlange warten.

Sehen wir uns als nächstes den *ConnectionThread* mal etwas genauer an: Im Constructor bekommen wir eine Referenz auf den Sockel zum Client, sowie einen Zähler, der uns sagt wie viele Users schon bei uns waren:

```
class ConnectionThread extends Thread {
    private int threadNr;
    private Socket socket;

    public ConnectionThread(int threadNr, Socket socket) {
        this.threadNr = threadNr;
        this.socket = socket;
    }

    public void run() {
        try {
            OutputStream out = socket.getOutputStream();
            String http = createHTTPResponse();
            out.write(http.getBytes());
            out.flush();
            out.close();
            socket.close();
            socket = null;

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    ...
}
```

Der ConnectionThread erweitert die Thread Klasse. Wir hätten natürlich auch das Runnable Interface implementieren können, aber wir wollen ab und zu ja mal was Neues lernen. In der run() Methode geht es zu wie gehabt: wir holen uns den OutputStream vom Socket und schreiben unsere Daten in den Stream. Da das HTTP Protokoll in seiner einfachsten Version zustandslos ist, machen wir den Socket gleich wieder zu. Da wir nur einen einfachen Webserver schreiben wollen, der immer nur die gleiche Seite zurückliefert, interessiert uns der InputStream nicht.

Kommen wir zum HTTP, also zur *createHTTPResponse()* Methode: die HTTP Response besteht aus dem HTTP Header und der Payload, in unserem Fall dem HTML:

```
private String createHTTPResponse() {
    String html = createDummyHTML();
    String httpHeader =
        createHTTPHeader(html.length(), "text/html");
    return httpHeader + html;
}
```

Das HTML erzeugen wir in der *createDummyHTML()* Methode, die natürlich beliebiges HTML erzeugen könnte:

```
private String createDummyHTML() {
    String html = "<html><body>"
            + "<h1>Hello from WebServerActivity!</h1>"
            + "<p>You are visitor number "
            + threadNr + ".</p>" + "</body></html>";
    return html;
}
```

Den HTTP Header erzeugen wir in der *createHTTPHeader()* Methode:

```
private String createHTTPHeader(long contentLength,
                                        String mimeType) {
    String httpHeader = "HTTP/1.0 200 OK\r\n"
            + "Server: WebServerActivity 1.0\r\n"
            + "Content-length: " + contentLength + "\r\n"
            + "Content-type: " + mimeType
            + "\r\n\r\n";
    return httpHeader;
}
```

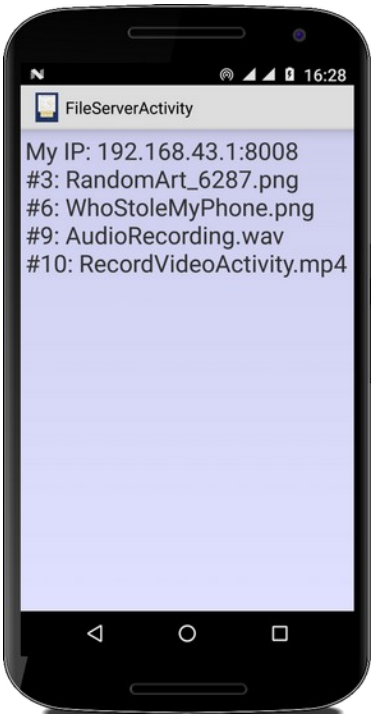
Wie so ein HTTP Header auszusehen hat kann man im RFC 2616 [15] nachlesen. Effektiv was der Header sagt ist:

1. Alles ist o.k.: "HTTP/1.0 200 OK";
2. ich heiße: "WebServerActivity 1.0";
3. es kommen jetzt *contentLength* Bytes an Daten, mach Dich bereit;
4. und die Daten sind vom Typ "text/html".

Ganz wichtig sind auch die zwei CRLF, also "\r\n\r\n", damit wird das Ende des HTTP Headers markiert, und der Browser weiß dann, was danach kommt sind die Daten.

Eine kleine Anmerkung noch zu unserem Zähler *threadNr*: der ist natürlich nur hübsches Beiwerk und eigentlich gar nicht nötig. Interessant ist er aber trotzdem: wenn wir nämlich eine Seite mit Firefox herunterladen, dann funktioniert der Zähler wie gewünscht, bei jedem Versuch erhöht sich der Zähler um eins. Verwenden wir aber den Chrome Browser, dann erhöht sich der Count um zwei. D.h. Chrome Nutzer verursachen eine doppelt so hohe Last auf unserem Server wie Firefox Nutzer. Wir sollten natürlich von Chrome Nutzern daher auch doppelt so hohe Gebühren verlangen. Ganz so schlimm ist es aber nicht: Chrome möchte neben dem eigentlich Request immer noch gerne ein Favicon [16] haben. Sobald wir ihm ein Favicon schicken, hört er auf danach zu fragen. Wir schicken ihm aber keins, geht ihn doch nix an was unser Favicon ist.

FileServer



Wie kann man denn Bilder zwischen zwei Smartphones oder einem Smartphone und einem Laptop einfach austauschen? Richtig, wir modifizieren unseren WebServer ein klein wenig und schon geht das. Alles was notwendig ist, ist ein kleiner InputStream im ConnectionThread:

```
public void run() {
    ...
    byte[] rawData;

    // what do you want?
    InputStream in = socket.getInputStream();
    BufferedReader br = new BufferedReader(
                        new InputStreamReader(in));
    String request = br.readLine();
    String[] parts = request.split(" ");
    String whatDoYouWant = parts[1];
    Log.i("FileServerActivity",
          "request: " + whatDoYouWant);

    File dir = android.os.Environment.
                    getExternalStorageDirectory();
    final File downloadFile =
        new File(dir, whatDoYouWant);

    if (!"/".equals(whatDoYouWant) && downloadFile.exists()) {
        rawData = getRawData(downloadFile);
    } else {
        String content = getDirectoryContent(dir);
        String header = createHTTPHeader(content.length(),
                                                "text/html");
        rawData = (header + content).getBytes();
    }

    // here it is:
    OutputStream out = socket.getOutputStream();
    out.write(rawData);
    out.flush();

    out.close();
    in.close();
    socket.close();
    socket = null;
    ...
}
```

Wir bereiten zunächst ein Bytearray für die Rohdaten vor. Dann fragen wir was unser Nutzer denn haben möchte. Die Information bekommen wir aus dem HTTP GET Request der vom Browser kommt, welcher in der Regel wie folgt aussieht:

```
GET /filename HTTP/1.1
```

Das was nach dem GET kommt ist die Datei die der Nutzer haben möchte.

Wie weiß denn unser Nutzer was er gerne haben möchte? Bei der ersten Anfrage geben wir ihm einfach eine Liste von Dateien mit der Methode *getDirectoryContent()*:

```
private String getDirectoryContent(File dir) {
    String content = "<html><body>";
    File[] files = dir.listFiles();
    for (File file : files) {
        if (file.isFile()) {
            content += "<a href='" + file.getName() + "'>";
            content += file.getName() + "</a><br/>";
        }
    }
    content += "</body></html>";
    return content;
}
```

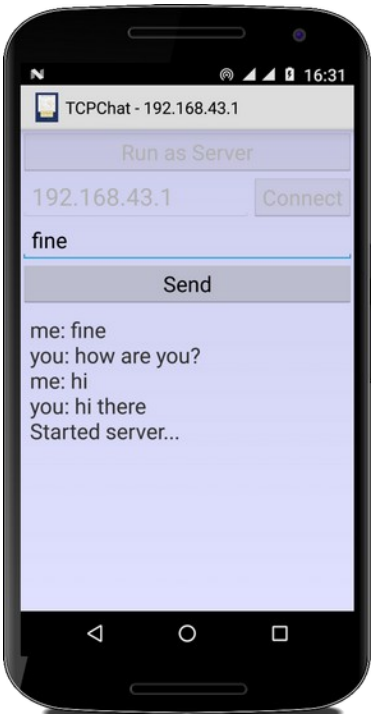
Schließlich brauchen wir noch die Methode *getRawData()* um die Datei in Binärform an den Clientbrowser zu schicken:

```
private byte[] getRawData(File downloadFile) {
    String mimeType = "application/octet-stream"; // "text/html"
    if (downloadFile.getAbsolutePath().endsWith(".png")) {
        mimeType = "image/png";
    }

    byte[] data = new byte[(int) downloadFile.length()];
    try {
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        FileInputStream fis = new FileInputStream(downloadFile);
        int read = 0;
        while ((read = fis.read(data)) != -1) {
            baos.write(data, 0, read);
        }
        fis.close();
    } catch (Exception e) {
        e.printStackTrace();
    }

    String header =
        createHTTPHeader(downloadFile.length(), mimeType);
    byte[] head = header.getBytes();

    byte[] buffer = new byte[header.length() +
        (int) downloadFile.length()];
    System.arraycopy(head, 0, buffer, 0, head.length);
    System.arraycopy(data, 0, buffer, head.length, data.length);

    return buffer;
}
```

Browser verstehen verschiedene MIME Types [17]: z.B. HTML ("text/html"), Bilder ("image/png") oder eben Binärdaten ("application/octet-stream").

Besonders sicher ist unser FileServer natürlich nicht, jeder der im lokalen Netz ist und unsere IP kennt, kann Dateien von unserem Handy runterladen. Da müsste man sich noch was überlegen.

TCPChat



Mit unserer FileServer Actiity können wir Dateien austauschen, wie wäre es denn wenn wir Textnachrichten (oder auch Sprachnachrichten) austauschen könnten? Hier wollen wir zwei Versionen implementieren, erst eine One-To-One Version die TCP verwendet, und danach eine One-To-Many, die UDP verwendet. Die Nachrichten sollen "live" übertragen werden, d.h. sobald man auf "Send" klickt soll die Nachricht gesendet werden und beim Gegenüber sofort angezeigt werden. Dazu benötigen wir eine stehende TCP Verbindung. D.h. im Gegensatz zu früher, lassen wir die TCP Verbindung offen, und schließen sie erst wieder wenn unsere Unterhaltung beendet ist.

Zwei Dinge machen die TCPChatActivity etwas komplizierter als unsere bisherigen Anwendungen: zunächst wissen wir nie genau wann unser Gegenüber eine Nachricht schickt. Das lösen wir durch einen separaten Server-Thread, der auf eingehende Nachrichten wartet. Daraus ergibt sich aber eine anderes Problem: ein Nicht-UI Thread darf eigentlich nicht auf unsere UI zugreifen. Das muss er aber, sonst sehen wir ja nicht was der Andere getippt hat. Man könnte das durch ein *runOnUiThread()* lösen, die bessere Lösung ist aber das mit einem AsyncTask zu lösen:

```
class ConnectionTask
        extends AsyncTask<Void, byte[], Boolean> {

    private Socket socket;
    private InputStream is;
    private OutputStream os;

    @Override
    protected Boolean doInBackground(Void... params) {
        ...
    }

    @Override
    protected void onProgressUpdate(byte[]... values) {
        ...
    }
}
```

Erinnern wir uns, die *doInBackground()* Methode wird über die *execute()* Methode aufgerufen, in ihr "lebt" unser AsyncTask. In der *onProgressUpdate()* können wir dann auf den UI Thread zugreifen. Der ConnectionTask ist am besten eine lokale Klasse, dann kann er nämlich ohne große Probleme auf Instanzvariablen der übergeordneten UI Klasse zugreifen.

Bevor wir aber den AsyncTast starten können, müssen wir eine TCP Verbindung zwischen den beiden Gesprächspartnern aufbauen. Dazu ist es zwingend notwendig, dass einer von beiden einen TCP Server in Form eines ServerSockets startet (es sei denn man hat wie WhatsApp einen zentralen Server irgendwo im Internet stehen [18]). In unserer App gibt es daher einen Knopf "Run as Server", der den Server startet. Der andere muss sich jetzt mit dem Server verbinden, dazu muss er natürlich die IP Adresse des Servers kennen. Je nachdem ob wir als Server oder Client starten, müssen wir das dem ConnectionTask mitteilen. Das geht am einfachsten über die *runAsServer* Instanzvariable:

```
public class TCPChatActivity extends Activity {
    private boolean runAsServer = false;
    private EditText editMessage;
    private TextView textConversation;
    ...
}
```

Um den ConnectionTask zu starten verwenden wir:

```
connectionTask.execute();
```

Abhängig davon ob wir als Server oder als Client unterwegs sind, passieren verschiedene Dinge in der doInBackground() Methode des ConnectionTasks:

```
protected Boolean doInBackground(Void... params) {
    if (runAsServer) {
        ServerSocket server = null;
        server = new ServerSocket(PORT);
        socket = server.accept();

    } else {
        String sIP = editServerIP.getText().toString();
        SocketAddress sockaddr = new InetSocketAddress(sIP, PORT);
        socket = new Socket();
        socket.connect(sockaddr, TIMEOUT);
    }

    if (socket.isConnected()) {
        is = socket.getInputStream();
        os = socket.getOutputStream();
        receiveMessages();
    } else {
        Log.e(getLocalClassName(),
              "socket not connected, should not happen...");
    }
    ...
}
```

Im ersten Fall initialisieren wir einen ServerSocket und lassen ihn auf eingehende Verbindungen warten, im anderen Fall stellen wir eine Verbindung als Client her. Wenn die Verbindung erfolgreich hergestellt wurde, erhalten wir sowohl auf Server- als auch auf Clientseite einen Socket. Von dem lassen wir uns sowohl InputStream als auch OutputStream geben, und verarbeiten diese dann in der *receiveMessages()* Methode, die identisch ist für Server und Client:

```
private void receiveMessages() throws IOException {
    byte[] buffer = new byte[BUFFER_SIZE];
    while (true) {
        int read = is.read(buffer, 0, BUFFER_SIZE);
        if (read == -1)
            break;
        // need to make copy, since this is used in other thread:
        byte[] temp = new byte[read];
        System.arraycopy(buffer, 0, temp, 0, read);
        publishProgress(temp);
    }
    Log.e(getLocalClassName(), "Done: should not happen... ");
}
```

Über die *publishProgress()* Methode zeigen wir die ankommenden Nachrichten im UI Thread an:

```
protected void onProgressUpdate(byte[]... values) {
    if (values.length > 0) {
        textConversation.setText("you: "
            + new String(values[0]) + "\r\n"
            + textConversation.getText());
    }
}
```

Wir sind fast fertig: was noch fehlt ist das Senden von Nachrichten. Dazu implementieren wir die *sendMessage()* Methode im ConnectionTask:

```
public void sendMessage() {
    String message = editMessage.getText().toString();
    try {
        if (socket.isConnected()) {
            os.write(message.getBytes());
        } else {
            Log.e(getLocalClassName(),
                  "sendMessage(): Socket is closed");
        }
    } catch (Exception e) {
        Log.e(getLocalClassName(), "sendMessage(): " + e);
    }
}
```

Wir greifen dazu auf das EditText Widget im UI Thread zu und schicken den Inhalt über den OutputStream an unseren Gesprächspartner. Das Senden selbst wird durch einen Klick auf den "Send" Button in der UI ausgelöst:

```
...
btnSend = (Button) findViewById(R.id.btnSend);
btnSend.setOnClickListener(new OnClickListener() {
    public void onClick(View v) {
        connectionTask.sendMessage();
        textConversation.setText("me: "
            + editMessage.getText().toString() + "\r\n"
            + textConversation.getText());
    }
});
...
```

Tada.

UDPChat

Die UDPChatActivity ist etwas einfacher als die TCP Variante. Das hat damit zu tun, dass wir nicht zwischen Client und Server unterscheiden müssen. Auch benötigen wir nicht die IP Adresse der anderen, da wir einfach einen Broadcast an alle im lokalen Netz schicken.

Schauen wir uns zunächst die *sendMessage()* Methode in der UDP Version an:

```
private void sendMessage() {
    try {
        String sMessage =
            et.getText().toString();
        DatagramSocket ds =
            new DatagramSocket();
        InetAddress serverAddr =
            Util.getLocalBroadcastAddress();
        DatagramPacket dp =
            new DatagramPacket(
                sMessage.getBytes(),
                sMessage.length(),
                serverAddr, PORT);
        ds.setBroadcast(true);
        ds.send(dp);
```

```
            } catch (Exception e) {
                Log.e(getLocalClassName(), "sendMessage(): " + e);
            }
        }
```

Die sieht zwar ein klein wenig komplizierter aus, aber im Gegensatz zur TCP Version benötigen wir keine Referenzen auf InputStream, OutputStream oder Socket.

Ähnlich verhält es sich beim Empfangen der Nachrichten, das wir in der *doInBackground()* Methode erledigen:

```
    protected Boolean doInBackground(Void... params) {
        ...
        byte[] buffer = new byte[MAX_PACKET_SIZE];
        DatagramPacket packet =
            new DatagramPacket(buffer, buffer.length);
        DatagramSocket socket = null;

        socket = new DatagramSocket(PORT);
        while (true) {
            ...
            socket.receive(packet);

            InetAddress senderIP = packet.getAddress();
            String senderName = "" + senderIP.getAddress()[3];

            byte[] temp = new byte[packet.getLength()];
            System.arraycopy(buffer, 0, temp, 0, temp.length);
            publishProgress(senderName.getBytes(), temp);
            ...
        }
        ...
        return result;
    }
```

Da es sich um einen Boadcast handelt, wollen wir den Sender der Nachricht noch anhand des letzten Bytes seiner IP Adresse, *senderIP.getAddress()[3]*, identifizieren.

GameServer

Will man ein Netzwerkspiel implementieren, dann müssen sich die Spieler erst einmal finden und verbinden. Es gibt zwar von Google die Wi-Fi Peer-to-Peer API [19], die scheint aber nicht viel zu taugen. Deswegen implementieren wir mal kurz unsere eigene Version.

Nehmen wir an wir wollen ein Spiel für zwei Spieler schreiben, z.B. TicTacToe. Wir könnten, wie im Webserver Beispiel, einen ServerSocket starten und auf eingehende Verbindungen warten. Dazu müsste der Client aber die IP Adresse des Servers kennen und das ist ja so 20. Jahrhundert.

Im 21. Jahrhundert starten wir einen UDP Server,

```
    public void run() {
        ...
        byte[] buffer = new byte[MAX_PACKET_SIZE];
        DatagramSocket server =
            new DatagramSocket(GAME_SERVER_PORT);
        DatagramPacket packet =
            new DatagramPacket(buffer, buffer.length);
```

```
            while (true) {
                ...
                server.receive(packet);
                InetAddress address = packet.getAddress();
                final String msg = address.getHostAddress();
                ...
```

der auf ankommende UDP Packete hört. Die Spieler senden einfach nur einen UDP Broadcast. Kommt jetzt ein Packet von einem Spieler, dann können wir über *getAddress()* die IP Adresse des Spielers ermitteln. Wir warten also bis alle Spieler sich bemerkbar gemacht haben,

```
                playerSet.add(address);
                if (playerSet.size() >= NR_OF_PLAYERS_PER_TEAM) {
                    int i = 0;
                    InetAddress[] addresses =
                        new InetAddress[NR_OF_PLAYERS_PER_TEAM];
                    for (InetAddress playerAddress : playerSet) {
                        if (i < NR_OF_PLAYERS_PER_TEAM) {
                            addresses[i] = playerAddress;
                        }
                        i++;
                    }
                    establishConnectionBetweenPlayers(addresses);
                    playerSet.clear();
                }
                // reset the length for the next packet
                packet.setLength(buffer.length);
            }
            ...
        }
```

und speichern die IP Adressen der Spieler in dem HashSet *playerSet* ab. Wenn wir alle Spieler haben, dann teilen wir über die *establishConnectionBetweenPlayers()* Methode allen Spielern die gegenseitigen IP Adressen mit:

```
        private void establishConnectionBetweenPlayers(
                                        final InetAddress[] addresses) {
            for (int i = 0; i < addresses.length; i++) {
                final InetAddress inetAddress = addresses[i];
                Thread th = new Thread(new Runnable() {

                    public void run() {
                        sendAddressesToClient(inetAddress, addresses);
                    }

                    private void sendAddressesToClient(
                                        final InetAddress inetAddress,
                                        final InetAddress[] addresses) {
                        try {
                            SocketAddress sockaddr =
                                new InetSocketAddress(inetAddress,
                                                      GAME_SERVER_PORT);
                            Socket socket = new Socket();
                            socket.connect(sockaddr, TIMEOUT);

                            OutputStream os = socket.getOutputStream();
                            ObjectOutputStream oos =
                                new ObjectOutputStream(os);
                            oos.writeObject(addresses);
                            oos.close();
                            os.close();
                            socket.close();
```

```
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            });
            th.start();
        }
    }
```

Diese Methode ist ein bisschen fies: sie startet so viele Threads wie es Spieler gibt und schickt jedem Spieler die IP Adressen der anderen. Die Spieler müssen natürlich zuhören.

Eine Anmerkung, wahrscheinlich ist es besser den GameServer nicht als Activity zu starten, sondern als Service. Oder man lässt den Server mitspielen, soll heißen er ist auch einer der Clients. Macht den Code aber ein klein wenig komplizierter.

GameClient

Kommen wir zum Gegenpart: dem GameClient. Der schickt erst einmal einen UDP Broadcast, was er da schickt ist egal. Deswegen verwenden wir einfach unsere *sendYoToEveryone()* von früher:

```
public class GameClientActivity extends Activity
                                implements Runnable {
    public void onCreate(Bundle savedState) {
        ...

        sendYoToEveryone();

        Thread th = new Thread(this);
        th.start();
    }
    ...
}
```

Sobald wir unser "Yo" verschickt haben, müssen wir auf Nachricht vom Server warten, dazu starten wir lokal einen TCP ServerSocket,

```
    public void run() {
        ...
        // then wait for TCP answer from server
        ServerSocket server = null;
        server = new ServerSocket(GAME_SERVER_PORT);
        Socket socket = server.accept();

        InputStream is = socket.getInputStream();
        ObjectInputStream ois = new ObjectInputStream(is);
        final InetAddress[] addresses =
            (InetAddress[]) ois.readObject();
        ois.close();
        is.close();
        socket.close();
        server.close();
        ...
    }
```

und warten darauf, dass uns der GameServer kontaktiert. Danach haben wir alle Adressen der anderen Spieler und können mit dem eigentlichen Spiel beginnen. Ob wir da dann mit TCP oder UDP kommunizieren, hängt von der Spielart ab. Bei einem rundenbasierten Spiel wie TicTacToe, würde ich die TCP Variante wählen, bei einem schnelleren Actionspiel wahrscheinlich eher die UDP Variante.

GameClient2

In dem Beispiel oben läuft der Server im lokalen Netz und verwendet UDP Broadcasts, damit sich die Spieler finden. Leider gibt es einige Smartphones die keinen UDP Broadcasts können. Alternativ kann man natürlich einen Server im Internet hosten. Dabei sind aber zwei Dinge zu beachten:

1. wir können keinen UDP Broadcast ins Internet schicken, wir müssen also die Adresse des Servers im Internet kennen und verbinden uns ganz normal über einen HTTP Request;
2. ein Server im Internet kann sich nicht einfach mit einem Computer in einem lokalen Netz verbinden. Es gibt zwar Websockets, die sind aber nicht ganz trivial und in unserem Fall auch gar nicht notwendig.

Mit dem Hintergedanken des Reuse (z.B. für TicTacToe später), lagern wir den Code der mit dem Verbindungsaufbau zu tun hat in die Klasse *ConnectionBroker* aus. Anstelle einen "Yo"-Broadcast zu schicken, schicken wir einen HTTP Request mittels der URL Klasse an den Server (in unserem Fall "http://wikimusicapp.appspot.com/"):

```
public class ConnectionBroker extends Thread {
    private final int GAME_SERVER2_PORT = 3236;
    private final int TIMEOUT = 1000; // in ms

    private String myLocalIP;
    private Socket socket;
    private boolean isSocketAvailable = false;
    private boolean isServer = false;

    public void run() {
        ...
        myLocalIP = Util.getMyLocalIpAddress().getHostAddress();
        URL url = new URL(
            "http://wikimusicapp.appspot.com/gameserver?internalIP="
            + myLocalIP);

        HttpURLConnection con =
            (HttpURLConnection) url.openConnection();

        BufferedReader br = new BufferedReader(
            new InputStreamReader(con.getInputStream()));
        String data = br.readLine();
        br.close();

        if (data != null) {
            if (data.trim().equals("{}")) {
                startServerAndWaitToBeContacted();
            } else {
                connectToWaitingPlayer(data);
            }
        }
        ...
    }
}
```

Das Ganze muss als separater Thread laufen, da es ja unter Umständen etwas dauern kann bis wir einen Partner finden.

Hier gibt es jetzt zwei Möglichkeiten:

1) Wenn sich noch nicht alle Spieler angemeldet haben, müssen wir warten. Das ist genauso wie im GameServer Beispiel oben: wir starten einen lokalen TCP ServerSocket und warten bis uns jemand kontaktiert:

```
private void startServerAndWaitToBeContacted() {
    try {
        ServerSocket server = null;
        server = new ServerSocket(GAME_SERVER2_PORT);
        socket = server.accept();
        isSocketAvailable = true;

        server.close();

    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

2) Alle anderen Spieler haben sich angemeldet, wir sind der letzte: dann gibt uns der Server die lokalen Adressen der anderen Spieler als JSON:

```
{[192.168.32.23, 192.168.32.42]}
```

Die warten ja darauf von uns kontaktiert zu werden, was wir in der *connectToWaitingPlayer()* Methode tun:

```
private void connectToWaitingPlayer(String data) {
    String ipServer = getIPAddressToConnectTo(data);
    try {
        SocketAddress sockaddr =
            new InetSocketAddress(ipServer, GAME_SERVER2_PORT);
        socket = new Socket();
        socket.connect(sockaddr, TIMEOUT);
        isSocketAvailable = true;

    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Hier gehen wir von einem Zweispieler-Spiel aus. Bei mehr Spielern müssten wir natürlich mit jedem Spieler eine separate Verbindung aufbauen oder wieder über UDP Broadcasts arbeiten. Ach ja, die *getIPAddressToConnectTo()* Methode fehlt noch:

```
private String getIPAddressToConnectTo(String data) {
    StringTokenizer st = new StringTokenizer(data, "{[, ]}");
    while (st.hasMoreTokens()) {
        String ip = st.nextToken();
        if (!ip.equals(myLocalIP)) {
            return ip;
        }
    }
    return null;
}
```

Diesen ConnectionBroker können wir jetzt problemlos in unserer Activity verwenden:

```
public class GameClient2Activity extends Activity
                                 implements Runnable {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        ...
        Thread th = new Thread(this);
        th.start();
    }

    @Override
    public void run() {
        String myLocalIP =
            Util.getMyLocalIpAddress().getHostAddress();

        // establish connection to other player
        ConnectionBroker broker = new ConnectionBroker();
        broker.start();
        while (!broker.isSocketAvailable()) {
            pause(200);
        }

        // get InputStream and OutputStream
        try {
            final Socket socket = broker.getSocket();
            InputStream is = socket.getInputStream();
            OutputStream os = socket.getOutputStream();
            ...

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Auch hier starten wir wieder einen separaten Thread, der den ConnectionBroker startet und wartet bis sich ein Partner gefunden hat. Sobald *isSocketAvailable()* true ist, haben wir einen Partner und können dann mit dem Spiel beginnen.

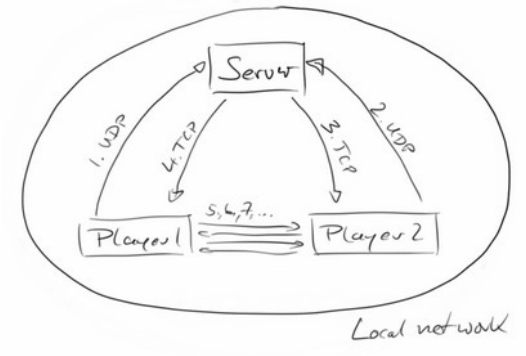
Noch kurz zum Server: eine Möglichkeit ist diesen als Java Servlet zu implementieren [20]. Der Code ist nicht besonders komplex, ist aber auch nicht besonders sicher:

```
public class GameServer extends HttpServlet {

    private final int NR_OF_PLAYERS_PER_TEAM = 2;
    private Map<String, Set<String>> clients =
        new HashMap<String, Set<String>>();

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response) throws IOException {

        String internalIP = request.getParameter("internalIP");

        if (internalIP != null
            && internalIP.length() > 8
            && internalIP.length() < 16) {

            String clientAddresses = "{";
            String externalIP = request.getRemoteAddr();
```

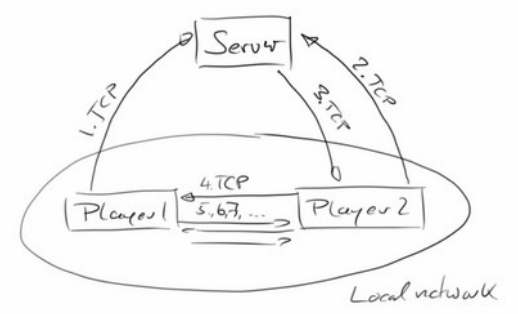
```
                if (clients.containsKey(externalIP)) {
                    Set<String> playerSet = clients.get(externalIP);
                    playerSet.add(internalIP);
                    if (playerSet.size() >= NR_OF_PLAYERS_PER_TEAM) {
                        clientAddresses += playerSet.toString();
                        clients.remove(externalIP);
                    }

                } else {
                    Set<String> playerSet = new HashSet<String>();
                    playerSet.add(internalIP);
                    clients.put(externalIP, playerSet);
                }
                clientAddresses += "}";

                PrintWriter out = response.getWriter();
                out.println(clientAddresses);
            }
        }
    }
```

Der Server wird nur kurz zum Austausch der lokalen IP Adressen verwendet, danach spielt er keine Rolle mehr.

Die folgenden Grafiken sollen den Unterschied zwischen GameClient und GameClient2 veranschaulichen:



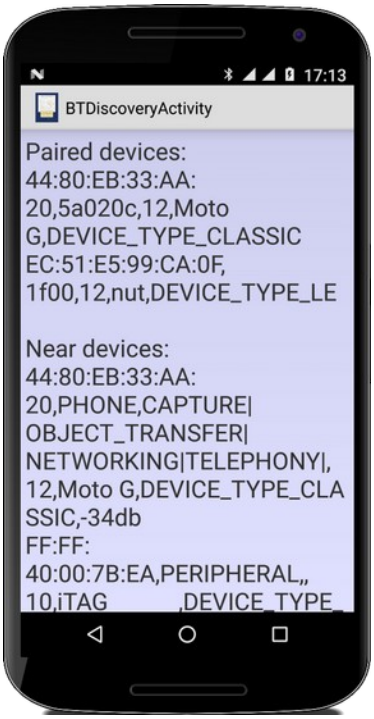
GameClient



GameClient2

Im ersten Fall läuft alles im internen Netzwerk ab, es ist also keine Internetverbindung notwendig. Im zweiten Fall, steht der Server im Internet. Er wird allerdings nur zum Finden der anderen Spieler verwendet. Im eigentlich Spiel spielt er dann keine Rolle mehr.

BTDiscovery



Fast alle Smartphones heutzutage haben Bluetooth [21]. Bluetooth wird wie Wifi zur drahtlosen Datenübertragung verwendet. Am bekanntesten sind wahrscheinlich Bluetooth Headsets und Lautsprecher. Es sendet im 2,4 GHz Band, was manchmal zu Problemen führt, wenn man Wifi auch im 2,4 GHz Band betreibt (Wifi kann man auch im 5,8 GHz Band betreiben, dann gibt es keine Störungen mehr). In der Regel ist die Reichweite von Bluetooth geringer, da es mit geringerer Leistung sendet, das muss aber nicht sein.

Als erstes Beispiel wollen wir mal sehen, was es denn so an Bluetooth Geräten in der Nähe gibt. Dafür verwenden wir den *BluetoothAdapter*:

```
public class BTDiscoveryActivity extends Activity {
    ...
    private BluetoothAdapter btAdapter;
    private BroadcastReceiver btReciever;

    @Override
    public void onCreate(Bundle savedState) {
        ...

        btAdapter =
            BluetoothAdapter.getDefaultAdapter();

        if (btAdapter.isDiscovering()) {
            btAdapter.cancelDiscovery();
        }

        Set<BluetoothDevice> pairedDevices =
            btAdapter.getBondedDevices();
        ...

        btReciever = new BluetoothReceiver();
        IntentFilter filter =
            new IntentFilter(BluetoothDevice.ACTION_FOUND);
        this.registerReceiver(btReciever, filter);
        btAdapter.startDiscovery();
    }
    ...
}
```

Zunächst einmal sagt uns der BluetoothAdapter mit welchen Geräten wir in der Vergangenheit ein "Pairing" durchgeführt haben. "Pairing" ist wichtig, da man sich nur mit gepairten Geräten verbinden kann. Allerdings heißt das nicht, dass diese Geräte gerade in der Nähe sind. Diese Information erhalten wir mit dem *BluetoothReceiver*:

```
class BluetoothReceiver extends BroadcastReceiver {
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        if (BluetoothDevice.ACTION_FOUND.equals(action)) {
            BluetoothDevice device = intent.getParcelableExtra(
                                    BluetoothDevice.EXTRA_DEVICE);
            short rssi = intent.getShortExtra(
                    BluetoothDevice.EXTRA_RSSI, Short.MIN_VALUE);

            BluetoothClass bc = device.getBluetoothClass();
            String mdc = BT_DEVICES.get(bc.getMajorDeviceClass());
            String services = getBTServices(bc);
            String sbc = "" + mdc + "," + services;
```

```
                    final String tmp = "" + device.getAddress() + ","
                            + sbc + "," + device.getBondState() + ","
                            + device.getName() + ","
                            + BT_DEVICE_TYPES[device.getType()] + ","
                            + rssi + "db";
                    tv.append(tmp + "\n");
                }
            }
        }
```

Hier erhalten wir die MAC Adresse der Geräte, ob sie mit uns verbunden sind, ihren Namen, und die momentane Signalstärke. Was den Typ angeht unterscheidet Android zwischen den folgenden Typen:

```
"DEVICE_TYPE_UNKNOWN", "DEVICE_TYPE_CLASSIC",
"DEVICE_TYPE_LE", "DEVICE_TYPE_DUAL"
```

Ein Gerät ist also entweder klassisch oder "Low Energy", manche sind auch beides.

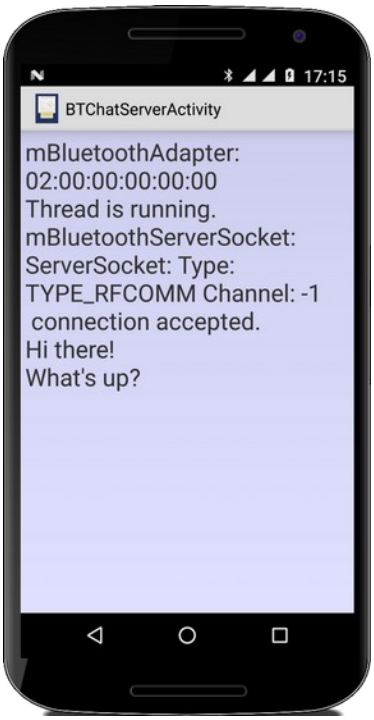
Interessant sind auch noch die Arten von Geräten die es gibt (BT_DEVICES). In der Android Dokumentation [22] finden sich:

- AUDIO_VIDEO
- COMPUTER
- HEALTH
- IMAGING
- MISC
- NETWORKING
- PERIPHERAL
- PHONE
- TOY
- UNCATEGORIZED
- WEARABLE

Und zusätzlich können folgende Services zu Verfügung gestellt werden (BT_SERVICES):

- AUDIO
- CAPTURE
- INFORMATION
- LIMITED_DISCOVERABILITY
- NETWORKING
- OBJECT_TRANSFER
- POSITIONING
- RENDER
- TELEPHONY

Wenigstens sind das die Services die Android verwenden kann.



BTChatServer

Bevor wir irgendetwas mit Bluetooth machen, müssen die beiden Geräte, die miteinander reden sollen, erst einmal gepairt werden. Das macht man über die Bluetooth Einstellung auf dem Smartphone. Da wir hier einen Chat zwischen zwei Smartphones implementieren wollen, muss man das natürlich auf beiden Geräten tun.

Für unseren Bluetooth ChatServer benötigen wir natürlich wieder Zugriff auf den *BluetoothAdapter*:

```
public class BTChatServerActivity extends Activity
                                   implements
```

```
Runnable {
    ...
    private BluetoothAdapter mBluetoothAdapter;
    private BluetoothServerSocket mBluetoothServerSocket;
    private BluetoothSocket socket;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        ...
        mBluetoothAdapter = BluetoothAdapter.getDefaultAdapter();
        Thread th = new Thread(this);
        th.start();
    }
}
```

Da es sich um einen Server handelt, muss das Ganze wieder als eigener Thread laufen. In dessen *run()* Methode starten wir einen *BluetoothServerSocket*:

```
    public void run() {
        ...
        mBluetoothServerSocket =
            mBluetoothAdapter.listenUsingRfcommWithServiceRecord(
                "BTChatActivity_Server",
                UUID.fromString(MY_UUID));

        while (true) {
            ...
            BluetoothSocket socket = mBluetoothServerSocket.accept();
            Log.i("BTChatServerActivity", " connection accepted.");

            if (socket != null) {
                InputStream is = socket.getInputStream();
                OutputStream os = socket.getOutputStream();
                while (isRunning) {
                    final int data = is.read();
                    os.write(data);
                    runOnUiThread(new Runnable() {
                        @Override
                        public void run() {
                            tv.append("" + (char) data);
                        }
                    });
                }
            }
            ...
        }
    }
```

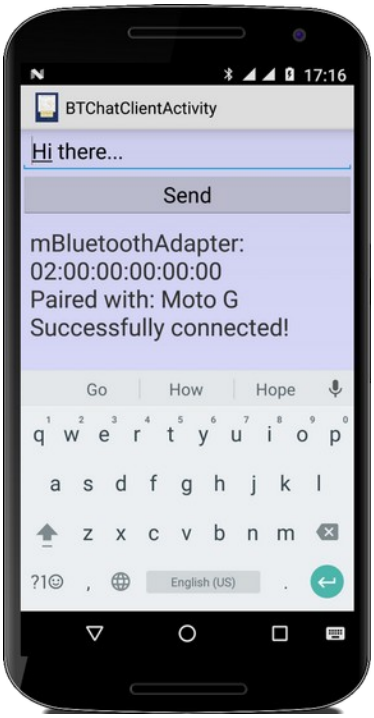
Danach warten wir in der *accept()* Methode ganz geduldig auf hereinkommende Verbindungsanfragen. Der Code der folgt ist eins zu eins identisch mit dem was wir bereits bei TCP gesehen haben: wir haben je einen InputStream und einen OutputStream, vom einen lesen wir, in den anderen schreiben wir. In dem Beispiel gerade, haben wir einen einfachen Echo-Server implementiert [23].

Eine Anmerkung noch zur UUID: oben haben wir folgende verwendet:

```
BASE_BLUETOOTH_UUID = "-0000-1000-8000-00805F9B34FB";
MY_UUID = "00420042" + BASE_BLUETOOTH_UUID;
```

Es scheint so zu sein, dass man den vorderen Teil frei wählen kann, der hintere Teil aber vorgegeben ist. So richtig zu verstehen scheint das aber niemand [24].

BTChatClient



Kommen wir zum Bluetooth ChatClient: Zunächst benötigen wir wieder den BluetoothAdapter,

```
public class BTChatClientActivity extends Activity{

    private BluetoothAdapter mBluetoothAdapter;
    private BluetoothSocket socket;
    private OutputStream os;

    @Override
    public void onCreate(Bundle savedState) {
        ...
        mBluetoothAdapter =
            BluetoothAdapter.getDefaultAdapter();
        tv.append("mBluetoothAdapter: " +
            mBluetoothAdapter.getAddress() + "\n");

        BluetoothDevice mBluetoothDevice =
            findPairedBluetoothDevice("Moto");

        if (mBluetoothDevice != null) {
            ...
            socket =
                mBluetoothDevice.createRfcommSocketToServiceRecord(
                    UUID.fromString(BTChatServerActivity.MY_UUID));
            socket.connect();
            os = socket.getOutputStream();
            os.write("Hi there!\n".getBytes());
            ...
        } else {
            tv.append("No device found!\n");
        }
    }
```

Dann suchen wir unter den gepairten Devices eines mit dem wir uns verbinden wollen, hier ist es das "Moto" Gerät. Haben wir das Gerät gefunden, dann lassen wir uns den Sockel geben und dessen OutputStream. Wichtig ist, dass die UUID die gleiche ist wie die des Servers.

Die Methode *findPairedBluetoothDevice()* ist ziemlich einfach,

```
    private BluetoothDevice findPairedBluetoothDevice(String name) {
        BluetoothDevice mBluetoothDevice = null;
        Set<BluetoothDevice> pairedDevices =
            mBluetoothAdapter.getBondedDevices();
        for (BluetoothDevice device : pairedDevices) {
            if (device.getName().trim().startsWith(name)) {
                mBluetoothDevice = device;
            }
        }
        return mBluetoothDevice;
    }
```

das könnte man aber auch besser machen.

In der momentanen Version, schickt nur der Client Nachrichten an den Server. Aber mit dem was wir aus den Projekt zum TCP-Chat gelernt haben, ist es ein leichtes auch Nachrichten zu empfangen.

BLEScanner

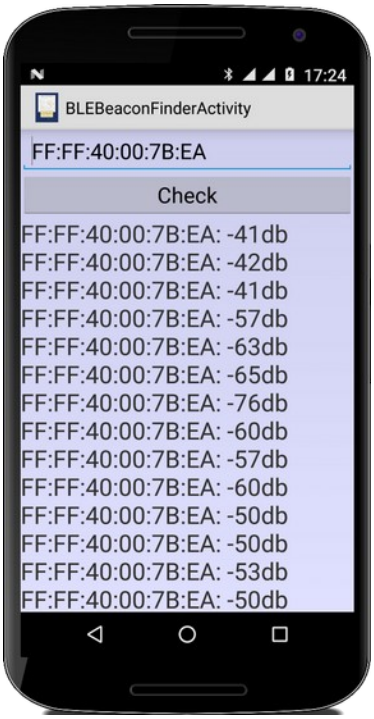
Bluetooth Low Energy [25] ist Teil der Bluetooth 4.0 Spezifikation und viele der neureren Geräte verwenden es. Der Vorteil ist, dass es viel stromsparender sein soll. Prinzipiell funktioniert Bluetooth LE sehr ähnlich: wir benötigen wieder einen BluetoothAdapter, allerdings müssen wir den explizit vom System anfordern:

```
public class BLEScannerActivity extends Activity {
    ...
    private BluetoothAdapter mBluetoothAdapter;
    private BluetoothAdapter.LeScanCallback
                                    mLeScanCallback;

    @Override
    public void onCreate(Bundle savedState) {
        ...
        final BluetoothManager bluetoothManager =
           (BluetoothManager) getSystemService(
                        Context.BLUETOOTH_SERVICE);

        mBluetoothAdapter =
            bluetoothManager.getAdapter();

        mLeScanCallback =
            new BluetoothAdapter.LeScanCallback() {

            @Override
            public void onLeScan(final BluetoothDevice device,
                                 int rssi, byte[] scanRecord) {
                String address = device.getAddress();
                String name = device.getName();
                int strength = rssi;
                Log.i("BLEScannerActivity",
                      address + ": " + name + ", " + strength);
            }
        };

        mBluetoothAdapter.startLeScan(mLeScanCallback);
        ...
    }
}
```

Dann definieren wir einen *LeScanCallback*, der aufgerufen wird, wenn Bluetooth LE Geräte gefunden wurden. Vorher müssen wir aber den Scan mit *startLeScan()* starten.

Eine Besonderheit an Bluetooth LE ist, dass wir die Permissions ACCESS_COARSE_LOCATION und/oder ACCESS_FINE_LOCATION anfordern müssen. Ohne findet der Scanner keine der LE Geräte.

BLEBeaconFinder



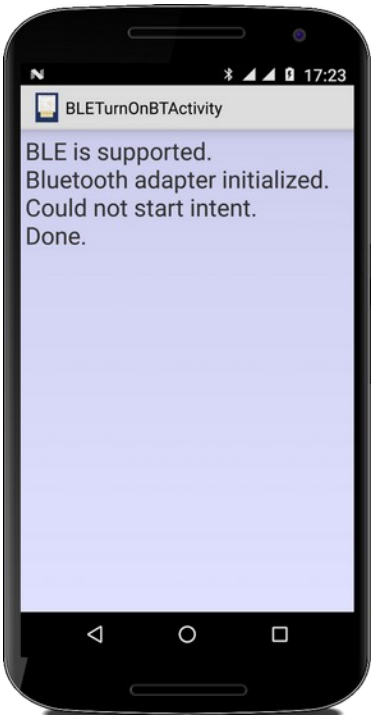
Eine hübsche Anwendung für Bluetooth LE ist ein Beacon Finder. Bluetooth Beacons kann man für ein, zwei Euro in China bestellen. Wenn man sich langweilt, kann man die dann in seinem Haus verstecken und mit unserer BeaconFinder Activity wieder finden. Ist ein bisschen wie Ostereier suchen.

Der Code ist fast eins-zu-eins identisch mit unserem BLEScanner. Hier starten wir den Scan aber erst wenn auf den Knopf gedrückt wurde, und nur der Beacon mit der richtigen MAC Adresse soll angezeigt werden:

```
public class BLEBeaconFinderActivity
                                extends Activity {
    ...
    @Override
    public void onCreate(Bundle savedState) {
        ...
        final EditText et =
            (EditText) findViewById(R.id.editText);
        et.setText("FF:FF:40:00:7B:EA");

        mLeScanCallback =
            new BluetoothAdapter.LeScanCallback() {

            @Override
            public void onLeScan(final BluetoothDevice device,
                                 int rssi, byte[] scanRecord) {
                String mac = et.getText().toString().trim();
                if (mac.equals(device.getAddress().toUpperCase())) {
                    String tmp = "" + device.getAddress() + ": "
                                 + rssi + "db";
                    tv.setText(tmp + "\n" + tv.getText());
                }
            }
        };

        Button btn = (Button) findViewById(R.id.button);
        btn.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                final String mac = et.getText().toString().trim();
                mBluetoothAdapter.startLeScan(mLeScanCallback);
            }
        });

        ...
    }
    ...
}
```

Auch dieses Beispiel könnte man für Indoor Location verwenden, mehr dazu später.

BLETurnOnBT

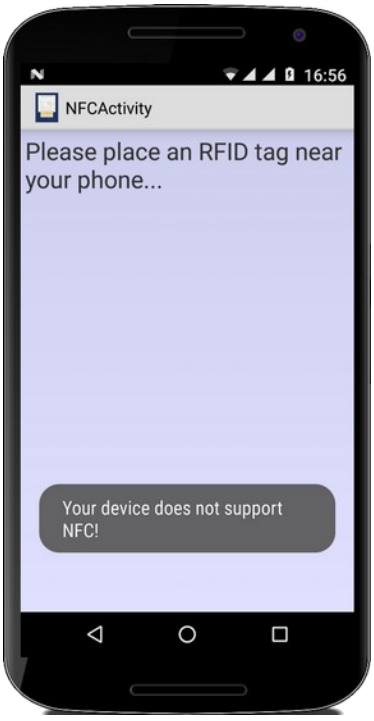


Unser letztes Bluetooth Beispiel soll noch zeigen wie wir Bluetooth über einen Intent einschalten können. Zunächst holen wir uns wieder eine Referenz auf den BluetoothManager:

```
public class BLETurnOnBTActivity extends Activity {

    private final int REQUEST_ENABLE_BT = 42;
    private TextView tv;
    private BluetoothAdapter mBluetoothAdapter;

    @Override
    public void onCreate(Bundle savedState) {
        ...
        final BluetoothManager bluetoothManager =
            (BluetoothManager) getSystemService(
                        Context.BLUETOOTH_SERVICE);

        mBluetoothAdapter =
            bluetoothManager.getAdapter();

        if (mBluetoothAdapter == null ||
            !mBluetoothAdapter.isEnabled()) {
            Intent enableBtIntent =
                new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
            startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
        }
    }

    public void onActivityResult(int requestCode, int resultCode,
                                                    Intent intent) {
        if (requestCode == REQUEST_ENABLE_BT &&
            resultCode == RESULT_OK) {
            tv.append("Everything is fine!\n");
            ...
        }
        super.onActivityResult(requestCode,
                               resultCode, intent);
    }
}
```

Dann starten wir einen Intent mit der Action "BluetoothAdapter.ACTION_REQUEST_ENABLE" und warten auf das Resultat. Falls alles o.k. ist, wurde Bluetooth eingeschaltet.

NFC



Einige Android Smartphones beherrschen auch die Near-Field Kommunikation, kurz NFC [26]. Obwohl hauptsächlich für elektronische Bezahlung beworben, kann man damit auch andere interessante Dinge machen. Z.B. kann man zwischen zwei Smartphones kommunizieren und, was noch viel nützlicher ist, man kann RFID Tags auslesen [27], und abhängig vom Gerät, sogar beschreiben. Wir wollen sie aber nur lesen.

Als erstes müssen wir im AndroidManifest um Erlaubnis fragen:

```
<uses-permission android:name="android.permission.NFC" />
<uses-feature android:name="android.hardware.nfc"
              android:required="true" />
```

Außerdem startet man eine NFCActivity über einen Intent, auch das geschieht in der AndroidManifest Datei:

```
<activity android:name="variationenzumthema_pr7.NFCActivity" >
    <intent-filter>
        <action
          android:name="android.nfc.action.TAG_DISCOVERED" />
        <category
          android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>
```

Hält man also sein Smartphone in die Nähe eines RFID Tags, dann wird unsere Activity gestartet.

Kommen wir zur Activity selbst. Zunächst einmal sollten wir in der onCreate() feststellen ob unser Gerät überhaupt NFC unterstützt und wenn ja ob es auch eingeschaltet ist:

```
public void onCreate(Bundle savedInstanceState) {
    ...
    NfcAdapter nfcAdapter = NfcAdapter.getDefaultAdapter(this);
    if (nfcAdapter == null) {
        Toast.makeText(this, "Your device does not support NFC!",
            Toast.LENGTH_LONG).show();
    } else if (!nfcAdapter.isEnabled()) {
        Toast.makeText(this, "You need to enable NFC!",
            Toast.LENGTH_LONG).show();
    }
}
```

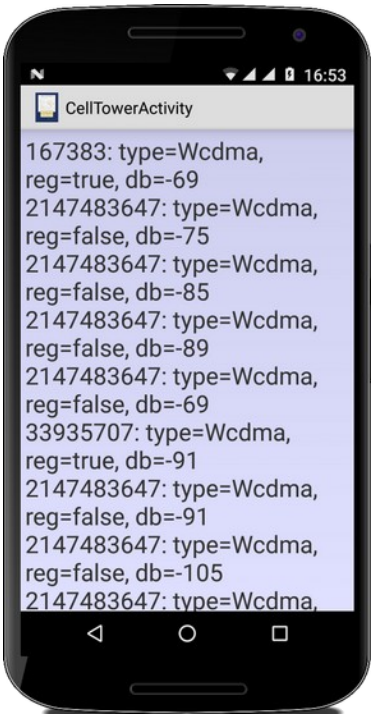
Sollte beides der Fall sein, dann können wir ein RFID Tag mit folgendem Code auslesen:

```
protected void onResume() {
    super.onResume();

    Intent intent = getIntent();
    String action = intent.getAction();

    if (NfcAdapter.ACTION_TAG_DISCOVERED.equals(action)) {
        Tag tag = intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);
        if (tag != null) {
            String tmp = "Id: "
                + Util.byteArrayToHexString(tag.getId()) + "\n\n";

            tmp += "Supported NFC Technologies:\n";
            String[] technologies = tag.getTechList();
            for (int i = 0; i < technologies.length; i++) {
                tmp += "- " + technologies[i] + "\n";
            }

            tv.setText(tmp);
        }
    }
}
```

In dem Beispiel geben wir nur die ID des Tags aus und welche Technologie das Tag verwendet. Das ist eigentlich ziemlich nutzlos. Interessant sind aber aktive RFID Tags: die haben Reichweiten bis zu mehreren hundert Metern, und über die Signalstärke kann man die grobe Entfernung des Tags abschätzen. Das kann zum Finden von Gegenständen ganz nützlich sein. Übrigens sowohl der Reisepass als auch der Personalausweis enthalten einen RFID Tag.

Jetzt können wir endlich testen ob die teure Geldbörse mit RFID-Schutz auch ihr Geld wert war.

CellTower

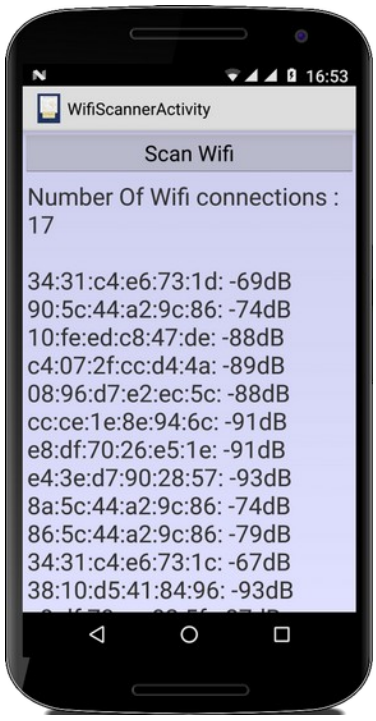


Wir haben im Sensor-Kapitel gesehen, dass man neben dem Signal der GPS Satelliten auch die Mobilfunkmasten zur groben Ortsbestimmung verwenden kann. Das passiert eigentlich alles im Hintergrund und man muss sich da nicht selbst darum kümmern. Trotzdem ist es interessant mal zu sehen welche Funkmasten denn in der Nähe sind. Das kann man mit dem *TelephonyManager*:

```
TelephonyManager telephonyManager =
    (TelephonyManager) this.getSystemService(
                        Context.TELEPHONY_SERVICE);

List<CellInfo> cells =
telephonyManager.getAllCellInfo();
for (CellInfo cellInfo : cells) {
    String type = "none";
    CellSignalStrength strength = null;
    boolean registered = false;
    int latitude = 0;
    int longitude = 0;
    int cellid = 0;
    int celllac = 0;

    if (cellInfo instanceof CellInfoCdma) {
        type = "Cdma";
        CellInfoCdma ci = (CellInfoCdma) cellInfo;
        strength = ci.getCellSignalStrength();
        registered = ci.isRegistered();

        CellIdentityCdma identity = ci.getCellIdentity();
        cellid = identity.getNetworkId();
        latitude = identity.getLatitude();
        longitude = identity.getLongitude();

    } else if (cellInfo instanceof CellInfoGsm) {
        type = "Gsm";
        CellInfoGsm ci = (CellInfoGsm) cellInfo;
        strength = ci.getCellSignalStrength();
        registered = ci.isRegistered();

        CellIdentityGsm identity = ci.getCellIdentity();
        cellid = identity.getCid();
        celllac = identity.getLac();

    } else if (cellInfo instanceof CellInfoLte) {
        type = "Lte";
        CellInfoLte ci = (CellInfoLte) cellInfo;
        strength = ci.getCellSignalStrength();
        registered = ci.isRegistered();

        CellIdentityLte identity = ci.getCellIdentity();
        cellid = identity.getCi();
        // celllac = identity.getLac();

    } else if (cellInfo instanceof CellInfoWcdma) {
        type = "Wcdma";
        CellInfoWcdma ci = (CellInfoWcdma) cellInfo;
        strength = ci.getCellSignalStrength();
        registered = ci.isRegistered();
```

```
            CellIdentityWcdma identity = ci.getCellIdentity();
            cellid = identity.getCid();
            celllac = identity.getLac();

        } else {
            Log.e("CellTowerActivity", "Unknown CellInfo subclass.");
        }

        if (strength != null) {
            int db = strength.getDbm();
            String msg = "" + cellid + ": ";

            msg += "type=" + type + ", ";
            msg += "reg=" + registered + ", ";
            msg += "db=" + db + "\n";
            tv.append(msg);
            Log.i("CellTowerActivity", "" + cellInfo.toString());
        }
    }
```

Je nach verwendetem Mobilfunkstandard, also z.B. Cdma, Gsm, Lte oder Wcdma, erhält man unterschiedliche Informationen. Was man aber immer bekommt, ist die Signalstärke und bei welchem Masten sich unser Telefon registriert hat. In der Regel ist das der mit dem stärksten Signal. Wenn man jetzt z.B. eine Liste der normalen Mobilfunkmasten hätte, und man auf einmal einen neuen entdeckt, der sich vielleicht sogar noch bewegt, könnte man mit dieser Information IMSI-Catcher entdecken [28]. Oder wenn man den Ort aller Mobilfunkmasten kennt, kann man auch ein eigenes Navigationssystem entwickeln.



WifiScanner

Mit dem WifiScanner erhalten wir eine Liste von verfügbaren Wifi Accesspoints. Dafür verwenden wir den *WifiManager*:

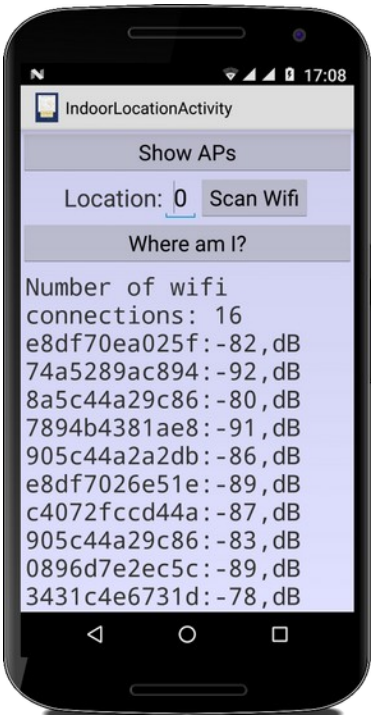
```
public class WifiScannerActivity extends Activity {

    private TextView tv;
    private WifiManager wifiManager;

    @Override
    protected void onCreate(Bundle savedState) {
        ...
        wifiManager = (WifiManager)
            getSystemService(Context.WIFI_SERVICE);

        if (wifiManager.isWifiEnabled() == false) {
            wifiManager.setWifiEnabled(true);
        }

        WifiReceiver wifiReceiver =
            new WifiReceiver();

        registerReceiver(wifiReceiver, new IntentFilter(
                        WifiManager.SCAN_RESULTS_AVAILABLE_ACTION));
        wifiManager.startScan();
        ...
    }
    ...
```

Mit *startScan()* starten wir den Scan. Das Resultat erhalten wir über einen BroadcastReceiver, die lokale Klasse *WifiReceiver*:

```
class WifiReceiver extends BroadcastReceiver {

    public void onReceive(Context c, Intent intent) {
        StringBuilder sb = new StringBuilder();
        List<ScanResult> wifiScanResultList =
            wifiManager.getScanResults();
        sb.append("Number Of Wifi connections :"
                  + wifiScanResultList.size() + "\n\n");
        for (ScanResult result : wifiScanResultList) {
            String bssid = result.BSSID;
            int signalLevel = result.level;
            sb.append(bssid + ": " + signalLevel + "dB\n");
        }
        tv.setText(sb);
    }
}
}
```

Das mag erst einmal ziemlich nutzlos erscheinen, aber im nächsten Beispiel sehen wir, wofür das gut sein kann. Denn es gibt überraschend viele Accesspoints.

IndoorLocation



Bei einer gewissen Mindestzahl von Wifi Accesspoints kann man unseren WifiScanner zur groben Positionsbestimmung verwenden, auch innerhalb von Gebäuden. Das ist zwar nicht super-genau, aber den Raum in dem man sich befindet kriegt man auf jeden Fall raus.

Die Idee ist, einmal durchs Haus zu laufen und an verschiedenen Stellen die Signalstärke der verfügbaren Accesspoints zu messen. Das speichern wir dann in einer Map:

```
// String is bssid, first integer is location-id,
// second is signal-strength
Map<String, Map<Integer, Integer>> accessPoints;
```

dabei ist der Key die BSSID des AccessPoints und der Value ist wieder eine Map, bei der jetzt der Key der Ort war an dem wir eine Messung durchgeführt haben, und der Value ist die Signalstärke an dem Ort.

Wenn wir jetzt wissen wollen wo wir sind, dann verwenden wir unseren WifiScanner und vergleichen die gefunden Accesspoints und Signalstärken mit unseren gespeicherten Werten. Daraus können wir dann den wahrscheinlichen Ort bestimmen. Dabei verwenden wir ein einfaches Voting, das funktioniert überraschend gut. Dieses Voting ist in die Klasse *WifiLocationManager* ausgelagert.

Unsere IndoorLocation Activity ist analog zum WifiScanner aufgebaut. Über den WifiManager starten wir Scans, und die Resultate kommen wieder im WifiReceiver an. Zusätzlich benötigen wir aber noch den WifiLocationManager:

```
public class IndoorLocationActivity extends Activity {
    ...
    private STATE state = STATE.IDLE;
    private WifiLocationManager wifiLocMangr;
    ...
}
```

Weiter benötigen wir eine Zustandsvariable *state*, die in einem von drei möglichen Zuständen sein kann:

```
enum STATE {
    IDLE, SCANNING_MODE, LOCATION_MODE
}
```

Bei IDLE passiert gar nichts. Wenn wir gerade durchs Haus laufen und Signalstärken messen, sind wir im SCANNING_MODE. Und wenn wir versuchen unsere Position zu bestimmen, dann sind wir im LOCATION_MODE.

Die UI ist trivial: z.B. der Knopf der den Scan startet, hat den folgenden OnClickListener:

```
Button btnScanWifi =
    (Button) this.findViewById(R.id.buttonScanWifi);
btnScanWifi.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        state = STATE.SCANNING_MODE;
        wifiManager.startScan();
        tv.setText("Starting Scan...");
    }
});
```

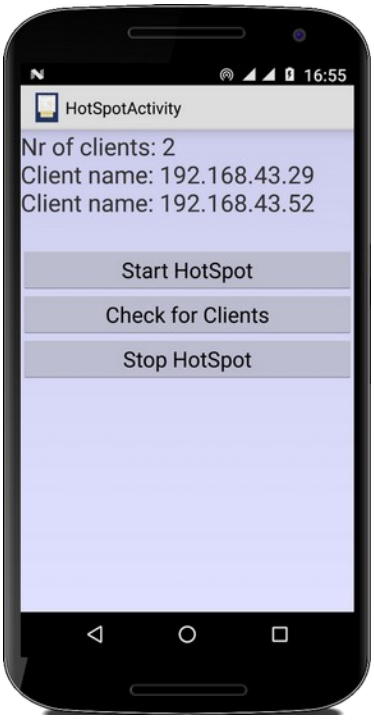
Der Knopf für die Ortsbestimmung ist vollkommen analog. Interessant sind die Änderungen an unserem *WifiReceiver*:

```
class WifiReceiver extends BroadcastReceiver {

    public void onReceive(Context c, Intent intent) {
        if (state == STATE.SCANNING_MODE) {
            List<ScanResult> wifiScanResultList =
                wifiManager.getScanResults();
            int loctn =
                Integer.parseInt(edt.getText().toString());
            wifiLocMangr.addScanResults(wifiScanResultList,
                                                    loctn);
            tv.setText(wifiLocMangr.listAccessPoints());

        } else if (state == STATE.LOCATION_MODE) {
            List<ScanResult> wifiScanResultList =
                wifiManager.getScanResults();
            TreeMap<Integer, Integer> votes =
                wifiLocMangr.getLocationEstimates(
                                        wifiScanResultList);

            // print votes
            String msg = "";
            for (int vote : votes.keySet()) {
                msg += "loc: " + votes.get(vote)
                       + ", count: " + vote + "\n";
            }
            tv.setText(msg);

        }
        state = STATE.IDLE;
    }
}
```

Wir unterscheiden jetzt zwischen den zwei Zuständen, im ersten Fall rufen wir die *addScanResults()* Methode des WifiLocationManagers auf, im anderen die *getLocationEstimates()* Methode.

Vermutlich könnte man mit einem Kalman Filter oder einem Neuronalen Netz die Genauigkeit um einiges erhöhen, aber wir wissen ja noch gar nicht wie das geht. Kommt im nächsten Buch.

HotSpot



Für die eine oder andere Anwendung wäre es ganz cool wenn man das Smartphone in einen Wifi Hotspot verwandeln könnte. Das ist mit der Klasse *HotSpotAPManager* von Nick Russler und Ahmet Yueksektepe sogar ganz einfach.

Zunächst benötigen wir eine HotSpotAPManager Instanzvariable, die wir in der onResume() initialisieren:

```
public class HotSpotActivity extends Activity {

    private HotSpotAPManager wifiApManager = null;

    protected void onResume() {
        super.onResume();
        wifiApManager = new HotSpotAPManager(this);
    }

    protected void onPause() {
        super.onPause();
        wifiApManager = null;
    }
    ...
}
```

Danach können wir mit

```
wifiApManager.setWifiApEnabled(null, true);
```

den Hotspot starten, mit

```
wifiApManager.setWifiApEnabled(null, false);
```

den Hotspot anhalten, und mit

```
List<String> clients = wifiApManager.getClientList();
```

können wir feststellen, wer sich denn alles mit unserem Hotspot verbunden hat. Natürlich benötigt man spezielle Permissions um einen Hotspot starten zu können. Ich hätte mal vermutet, dass es die folgenden drei sind:

```
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
<uses-permission android:name="android.permission.CHANGE_WIFI_STATE" />
<uses-permission android:name="android.permission.WRITE_SETTINGS" />
```

Idealerweise hat der Nutzer vorher schon mal einen Hotspot eingerichtet.

Challenges

RemoteDesktopClient

In dem java.awt Packet gibt es eine sehr interessante Klasse namens *Robot*: mit der kann man Screenshots machen, aber man kann auch die Maus auf dem Computer bewegen und sogar Tastatureingaben simulieren. Mit der App RemoteDesktopClient wollen wir also unseren Computer vom Handy aus steuern. Das kann z.B. ganz praktisch sein für eine Powerpoint Präsentation wenn wir keine von den teuren Fernbedienungen haben. Oder wenn wir in den Ferien sind und zu Hause auf unserem Computer was nachschauen wollen (wobei das nicht sehr schlau ist ohne entsprechende Sicherheitsvorkehrungen).

Als erstes schreiben wir das Programm das auf unserem normalen Desktop-Computer läuft. Das geht sowohl auf Windows, Mac als auch Linux. Es handelt sich um ein ganz einfaches Programm das aus einer *main()* heraus gestartet wird:

```
package variationenzumthema_pr7;
...
public class RemoteControlServer {
    private int PORT = 1778;

    enum COMMAND {
        GET_SCREENSHOT, MOUSE_MOVE, MOUSE_CLICKED
    }

    public static void main(String[] args) {
        int port = 1778;
        if (args.length == 1) {
            port = Integer.parseInt(args[0]);
        }
        RemoteControlServer rcs = new RemoteControlServer(port);
    }
    ...
```

Im Constructor läuft unser eigentliches Programm:

```
    public RemoteControlServer(int PORT) {
        this.PORT = PORT;
        ...
        ServerSocket echod = new ServerSocket(PORT);
        while (true) {
            Socket socket = echod.accept();
            System.out.println("connection established.");

            InputStream in = socket.getInputStream();
            DataInputStream din = new DataInputStream(in);

            int command = din.readInt();
            switch (COMMAND.values()[command]) {

            case GET_SCREENSHOT:
                sendScreenshot(socket);
                break;
```

```
                case MOUSE_MOVE:
                    int x = din.readInt();
                    int y = din.readInt();
                    mouseMove(x, y);
                    break;

                case MOUSE_CLICKED:
                    x = din.readInt();
                    y = din.readInt();
                    mouseClick(x, y);
                    break;

                default:
                    System.out.println("Unknown command.");
                    break;
                }

                in.close();
                socket.close();
            }
            ...
        }
    }
```

Wir haben einen ServerSocket und warten auf Kommandos die in Form von Integern hereinkommen. In unserem Beispiel haben wir drei Kommandos implementiert: GET_SCREENSHOT, MOUSE_MOVE und MOUSE_CLICKED. Die folgenden drei Methode zeigen wie man diese Kommandos mit der Robot Klasse umsetzt:

```
    private void mouseClick(int x, int y) throws AWTException {
        Robot robot = new Robot();
        robot.mouseMove(x, y);
        robot.mousePress(InputEvent.BUTTON1_MASK);
        robot.mouseRelease(InputEvent.BUTTON1_MASK);
    }

    private void mouseMove(int x, int y) throws AWTException {
        Robot robot = new Robot();
        robot.mouseMove(x, y);
    }

    private void sendScreenshot(Socket socket) throws Exception {
        OutputStream out = socket.getOutputStream();

        Robot robot = new Robot();
        Dimension screenSize =
            Toolkit.getDefaultToolkit().getScreenSize();
        BufferedImage image = robot.createScreenCapture(
                                            new Rectangle(screenSize));
        ImageIO.write(image, "png", out);

        out.close();
    }
```

Zum Starten des Programms gibt man am Desktop Computer auf einer Konsole oder Terminal folgendes ein:

```
java variationenzumthema_pr7.RemoteControlServer 1778
```

No rocket science here.

Kommen wir zur Android App, der *RemoteDesktopClientActivity*: Zunächst einmal handelt es sich dabei um ein GraphicsProgram:

```
public class RemoteDesktopClientActivity extends GraphicsProgram {
    private final int PORT = 1778;
    private final String IP = "192.168.178.54";
    private final int TIMEOUT = 1000;
    private final double SCALE = 1.0;

    private GImage img;
    private int oldX, oldY;

    enum COMMAND {
        GET_SCREENSHOT, MOUSE_MOVE, MOUSE_CLICKED
    }

    @Override
    public void init() {
        StrictMode.ThreadPolicy policy =
            new StrictMode.ThreadPolicy.Builder().permitAll().build();
        StrictMode.setThreadPolicy(policy);
        addMouseListeners();
    }
    ...
}
```

Aus Faulheit sind IP und Port fest gecoded. In der init() wird die ThreadPolicy gesetzt und der MouseListener hinzugefügt (was glaub ich gar nicht mal notwendig ist). Die run() Methode haut jetzt auch niemanden vom Hocker:

```
    public void run() {
        waitForTouch();
        while (true) {
            sendRequestToServer(COMMAND.GET_SCREENSHOT, 0, 0);
            if (img != null) {
                img.scale(SCALE);
                add(img);
            }
            pause(500);
        }
    }
```

Wir holen uns einfach einmal alle halbe Sekunde einen ScreenShot vom Server, skalieren es so damit es auf den Handybildschirm passt und zeigen es an. Die Methode *sendRequestToServer()* schickt die verschiedenen Requestarten an den Server,

```
    private void sendRequestToServer(COMMAND cmd, int x, int y) {
        ...
        SocketAddress sockaddr = new InetSocketAddress(IP, PORT);
        Socket socket = new Socket();
        socket.connect(sockaddr, TIMEOUT);

        OutputStream out = socket.getOutputStream();
        DataOutputStream dos = new DataOutputStream(out);
        dos.writeInt(cmd.ordinal());

        switch (cmd) {

        case GET_SCREENSHOT:
            getScreenShotFromServer(socket);
            break;
```

```
        case MOUSE_MOVE:
            dos.writeInt(x);
            dos.writeInt(y);
            break;

        case MOUSE_CLICKED:
            dos.writeInt(x);
            dos.writeInt(y);
            break;

        default:
            System.out.println("Unknown command.");
            break;
        }

        out.close();
        socket.close();
        ...
    }
```

und nur im Fall des Screenshots will sie auch wieder was haben:

```
    private void getScreenShotFromServer(Socket socket) {
        Bitmap image = null;
        try {
            InputStream is = socket.getInputStream();
            image = BitmapFactory.decodeStream(is);
            is.close();

        } catch (Exception e) {
            System.out.println(e.toString());
        }

        if (image != null) {
            if (img != null) {
                // allow garbage collector to do its work
                remove(img);
                img = null;
            }
            img = new GImage(image);
        }
    }
```

Die Methode ist deswegen etwas komplizierter da wir so viele Bilder bekommen, dass wir unbedingt den Garbage Collector bei seiner Arbeit unterstützen müssen. Sonst stürzt unsere App nach ein paar Sekunden mit einer OutOfMemory Fehlermeldung ab.

Was jetzt noch fehlt sind die Mausgeschichten:

```
    public void mouseMoved(int x, int y) {
        x = (int) (x / SCALE);
        y = (int) (y / SCALE);
        sendRequestToServer(COMMAND.MOUSE_MOVE, x, y);
    }

    public void mousePressed(int x, int y) {
        oldX = (int) (x / SCALE);
        oldY = (int) (y / SCALE);
    }
```

```
    public void mouseReleased(int x, int y) {
        x = (int) (x / SCALE);
        y = (int) (y / SCALE);
        int dx = oldX - x;
        int dy = oldY - y;
        // moved at less than 4 pixel
        if (dx * dx + dy * dy <= 16) {
            sendRequestToServer(COMMAND.MOUSE_CLICKED, x, y);
        }
    }
```

Viola.

TicTacToeActivity

Erinnern wir uns an das TicTacToe Spiel aus dem ersten Semester: wäre doch cool wenn wir das jetzt endlich über's Netzwerk spielen könnten. Mit dem ConnectionBroker aus dem GameClient2 Beispiel ist das auch gar nicht so schwer.

Zunächst benötigen wir neben den bereits existierenden Instanzvariablen noch die folgenden,

```
public class TicTacToeActivity
                        extends GraphicsProgram {
    ...
    private int me = 1;

    private boolean gameHasStarted = false;
    private ObjectOutputStream oos;

    public void init() {
        StrictMode.ThreadPolicy policy =
            new StrictMode.ThreadPolicy.Builder().
                                permitAll().build();
        StrictMode.setThreadPolicy(policy);
    }

    ...
}
```

und in der *init()* sollten wir wieder darauf achten die ThreadPolicy zu setzen. In der *run()* Methode stellen wir die Verbindung zum anderen Spieler her:

```
    public void run() {
        waitForTouch();
        setup();

        // establish connection to other player
        ConnectionBroker broker = new ConnectionBroker();
        broker.start();
        while (!broker.isSocketAvailable()) {
            pause(200);
        }
        showToastInUIThread("Game has started!");

        // who starts?
        if (broker.isServer()) {
            me = 2;
        }
        ...
```

Dabei müssen wir festsetzen welcher Spieler die Nummer 1 und welcher die Nummer 2 ist. Wir benutzen dafür die Methode *isServer()* vom Broker.

Als nächstes benötigen wir den InputStream und den OutputStream. Da wir ja nie wissen wann der andere Spieler seinen Zug macht, müssen wir alles was vom anderen Spieler kommt in einem eigenen Thread, dem *InConnectionThread*, auslagern:

```
            // now get InputStream and OutputStream
            final Socket socket = broker.getSocket();
            try {
                InConnectionThread in =
                    new InConnectionThread(socket.getInputStream());
                in.start();

                oos = new ObjectOutputStream(socket.getOutputStream());
                gameHasStarted = true;
                if (currentPlayer == me) {
                    showToastInUIThread("Your move!");
                } else {
                    showToastInUIThread(
                    "Wait for the other player to make the first move!");
                }

            } catch (IOException e) {
                e.printStackTrace();
            }
        }
```

Der InConnectionThread ist eine lokale Klasse,

```
    class InConnectionThread extends Thread {
        private InputStream is;

        public InConnectionThread(InputStream is) {
            StrictMode.ThreadPolicy policy = new StrictMode.ThreadPolicy.
                                            Builder().permitAll().build();
            StrictMode.setThreadPolicy(policy);

            this.is = is;
        }

        @Override
        public void run() {
            try {
                ObjectInputStream ois = new ObjectInputStream(is);
                while (true) {
                    Point p = (Point) ois.readObject();
                    displayPlayer(p.x, p.y, currentPlayer);
                }

            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
```

Über den InputStream bekommen wir die Position auf die der andere Spieler geklickt hat.

Die Klasse *Point* ist trivial:

```
public class Point implements Serializable {
    public int x;
    public int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Muss aber eine eigene Klasse und serialisierbar sein.

Wir sind fast fertig: in der *mousePressed()* Methode müssen wir noch ein paar kleine Modifikationen vornehmen:

```
public void mousePressed(int x, int y) {
    if (gameHasStarted) {
        if (currentPlayer == me) {
            int i = x / CELL_WIDTH;
            int j = y / CELL_WIDTH;

            if (TicTacToeLogic.isMoveAllowed(currentPlayer,i,j)){
                displayPlayer(i, j, currentPlayer);
                sendMoveToOtherPlayer(i, j);
            }

            if (TicTacToeLogic.isGameOver()) {
                displayGameOver();
            }

        } else {
            Toast.makeText(getApplicationContext(),
                "It is not your turn!",
                Toast.LENGTH_SHORT).show();
        }
    } else {
        Toast.makeText(getApplicationContext(),
            "Other player still needs to join!",
            Toast.LENGTH_SHORT).show();
    }
}
```

Und zwar müssen wir erst mal feststellen, ob das Spiel überhaupt schon begonnen hat. Soll heißen, der Spieler darf erst dann klicken, wenn sich ein Partner gefunden hat. Und natürlich darf er nur klicken wenn er wirklich an der Reihe ist. Falls der Spielzug denn erlaubt ist, zeigen wir ihn mit *displayPlayer()* wie gehabt an, aber wir müssen dem anderen Spieler natürlich noch mitteilen welchen Zug wir gemacht haben, das macht die *sendMoveToOtherPlayer()* Methode:

```
private void sendMoveToOtherPlayer(int i, int j) {
    try {
        Point p = new Point(i, j);
        oos.writeObject(p);
        oos.flush();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Niemand hat gesagt, dass Multiplayerspiele einfach sind... aber machbar sind sie schon.

Research

Auch in diesem Kapitel gibt es viele Themen die man noch durch Eigenrecherche vertiefen kann, genannt seien die folgenden drei.

MIME

Die "Multipurpose Internet Mail Extensions", kurz MIME, sind extrem wichtig wenn wir irgendetwas im Internet machen. Die wichtigste ist wohl "text/html" oder "image/png", aber es gibt noch ganz viele andere. Auch hier ist das Internet eine gute Möglichkeit sich über MIME und die MIME Types einen Überblick zu verschaffen.

RFC 2616

RFC 2616 [2] ist die Spezifikation des HTTP Protokolls. Wir sollten uns die Spezifikation mal ansehen, nach bekannten Namen unter den Autoren suchen, und im Inhaltsverzeichnis mal nach Request und Response suchen, sowie identifizieren welche HTTP Methoden und welche HTTP Status Codes es gibt.

JSON

Wenn wir öfter mal mit Webservices zu tun haben, dann wird uns immer wieder die JavaScript Object Notation, kurz JSON begegnen [12]. Deswegen macht es Sinn sich da mal einzulesen.

Fragen

1. Nennen Sie ein Beispiel, wofür man die WebView Klasse verwenden kann.

2. Für welche der folgenden Szenarien benötigen Sie die Permission :

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

   Mehrere Antworten können richtig sein.
   O wenn ein externer Browser aus Ihrer App gestartet wird
   O wenn Sie die WebView Klasse in Ihrer App verwenden

3. Geben Sie ein Beispiel für JSON und erklären Sie wofür man es verwendet.

4. UDP hat Broadcast, TCP nicht. Warum ist das wichtig wenn Sie ein Multiplayer Spiel implementieren wollen?

5. Wenn Sie versuchen einen Webserver auf einem Android Gerät laufen zu lassen, welches durch ein 3G Provider (z.B. O2 oder Vodaphone) mit dem Internet verbunden ist, kommt es zu Problemen. Erklären Sie bitte was der Grund dafür ist. (Hinweis: Bei diesen Providern ist die IP Adresse Ihres Gerätes von der Form 10.x.y.z).

6. Jedes Smartphone hat in der Regel mindestens drei verschiedene IP Adressen. Können Sie näher erläutern warum?

7. Wofür verwendet man die InetAddress Klasse und wofür die URL Klasse?

8. Erklären Sie kurz was die folgenden Klassen tun:
   - O InetAddress
   - O URL
   - O Socket
   - O Datagram

9. Mit welcher der nachfolgenden Java Klassen geht das Herunterladen von Internet Webseiten am einfachsten?
   - O InetAddress
   - O Socket
   - O URL

10. Was müssen Sie an der folgende Klasse ändern, damit sie serialisierbar wird?

```
class Point {
    public int x;
    public int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Referenzen

[1] Domain Name System, https://en.wikipedia.org/wiki/Domain_Name_System

[2] StrictMode, https://developer.android.com/reference/android/os/StrictMode

[3] Transmission Control Protocol, https://en.wikipedia.org/wiki/Transmission_Control_Protocol

[4] Internet Protocol, https://en.wikipedia.org/wiki/Internet_Protocol

[5] Hypertext Transfer Protocol, https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

[6] User Datagram Protocol, https://en.wikipedia.org/wiki/User_Datagram_Protocol

[7] Yo (app), https://en.wikipedia.org/wiki/Yo_(app)

[8] Internet Control Message Protocol, https://en.wikipedia.org/wiki/Internet_Control_Message_Protocol

[9] National Institute of Standards and Technology, https://en.wikipedia.org/wiki/National_Institute_of_Standards_and_Technology

[10] NIST Internet time service, https://www.nist.gov/pml/time-and-frequency-division/services/internet-time-service-its

[11] Coordinated Universal Time, https://en.wikipedia.org/wiki/Coordinated_Universal_Time

[12] JSON, https://en.wikipedia.org/wiki/JSON

[13] Google Books APIs, https://developers.google.com/books/

[14] Gson, https://en.wikipedia.org/wiki/Gson

[15] RFC2616, https://www.w3.org/Protocols/rfc2616/rfc2616.html

[16] Favicon, https://en.wikipedia.org/wiki/Favicon

[17] Multipurpose Internet Mail Extensions (MIME), https://en.wikipedia.org/wiki/MIME

[18] WhatsApp, https://en.wikipedia.org/wiki/WhatsApp

[19] Create P2P connections with Wi-Fi, https://developer.android.com/training/connect-devices-wirelessly/wifi-direct

[20] Java servlet, https://en.wikipedia.org/wiki/Java_servlet

[21] Bluetooth, https://en.wikipedia.org/wiki/Bluetooth

[22] android.bluetooth, https://developer.android.com/reference/android/bluetooth/package-summary

[23] Echo Protocol, https://en.wikipedia.org/wiki/Echo_Protocol

[24] Android: How do bluetooth UUIDs work?, https://stackoverflow.com/questions/13964342/android-how-do-bluetooth-uuids-work

[25] Bluetooth Low Energy, https://en.wikipedia.org/wiki/Bluetooth_Low_Energy

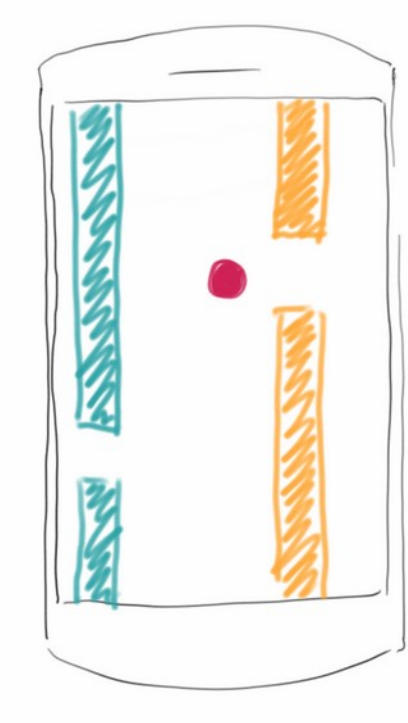
[26] Near-field communication, https://en.wikipedia.org/wiki/Near-field_communication

[27] Radio-frequency identification, https://en.wikipedia.org/wiki/Radio-frequency_identification

[28] IMSI-catcher, https://en.wikipedia.org/wiki/IMSI-catcher

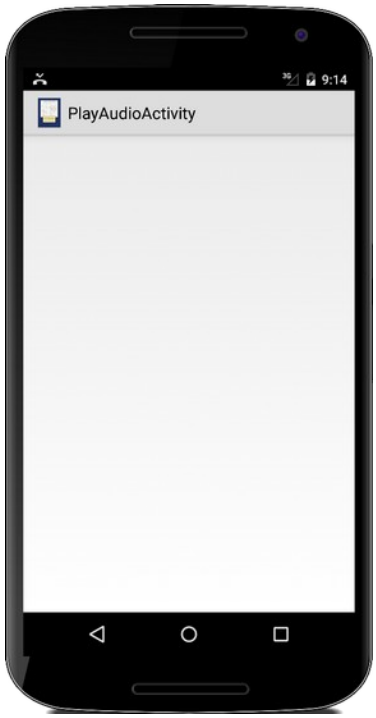
[29] Internet protocol suite, https://en.wikipedia.org/wiki/Internet_protocol_suite

Multimedia



Wir haben schon im Kapitel zu Sensoren gesehen, dass moderne Android Geräte viel mehr sind als nur dumme Telefone mit einer Internetverbindung. In diesem Kapitel werden wir uns mit den Multimediafähigkeiten von Android Geräten beschäftigen. Natürlich kann man die zum Abspielen von MP3 Dateien und Videos verwenden, aber mit dem Mikrofon und der Kamera als "Sensoren", eröffnen sich auf einmal ganz neuen Anwendungsszenarien. In diesem Kapitel werden wir daher zwar die Grundlagen der Multimediamöglichkeiten kennen lernen, aber auch etwas über den Tellerrand hinaus blicken.

PlayAudio



Um Audiodateien abzuspielen verwenden wir den *MediaPlayer*:

```
MediaPlayer player =
    MediaPlayer.create(this, R.raw.trap_beat);
player.start();
```

Der MediaPlayer kann mit Audiodateien die z.B. im Resourceverzeichnis */raw/* sind arbeiten, aber auch mit Assets oder ganz normalen Dateien auf der internen oder externen SD Karte:

```
String path = Environment.
    getExternalStorageDirectory().getPath()
    + "/Music/trap_beat.mp3";
MediaPlayer.setDataSource(path);
```

In der Regel sollte man vor dem *start()* noch die *prepare()* Methode aufrufen.

Der MediaPlayer hat ganz viele Methoden, die für uns interessantesten sind:

- **pause():** pausiert das Abspielen, mit start() geht's wieder weiter.
- **stop():** beendet das Abspielen.
- **isPlaying():** sagt einem, ob gerade was abgespielt wird.
- **getDuration():** gibt an wie lange ein Stück dauert, in Millisekunden.
- **getCurrentPosition():** gibt die momentane Abspielposition, in Millisekunden.
- **seekTo(int msec):** damit kann man die Abspielposition setzen, auch in Millisekunden.
- **setLooping(boolean looping):** lässt den MediaPlayer das Stück immer wieder wiederholen.
- **setVolume(float leftVolume, float rightVolume):** setzt die Lautstärke.

Wichtig ist noch zu wissen wie man den MediaPlayer wieder ausschaltet, das ist nämlich nicht ganz trivial:

```
protected void onDestroy() {
    if (player != null) {
        player.stop();
        player.release();
        player = null;
    }
    super.onDestroy();
}
```

Wir müssen also zunächst die *stop()* Methode aufrufen. Danach müssen wir unbedingt die *release()* Methode aufrufen, und schließlich macht es Sinn mit *player = null* dem GarbageCollector mitzuteilen, dass er jetzt aufräumen darf. Der Grund dafür ist, dass es sich beim MediaPlayer eigentlich um eine C++ Klasse mit einem ganz dünnen Java Wrapper handelt. Die *release()* Methode macht nichts anderes als den Destructor der C++ Klasse aufzurufen. Wenn wir das nicht machen, then "all hell breaks loose", wie die Amerikaner zu sagen pflegen.

SimpleSoundGenerator

Eine andere Möglichkeit Töne zu generieren ist mit dem *AudioTrack*. Dabei handelt es sich um eine Klasse die Arrays von 16-Bit Werten (shorts) abspielt. Zu verwenden ist die Klasse denkbar einfach:

```
AudioTrack audioTrack = new AudioTrack(
    AudioManager.STREAM_MUSIC, SAMPLE_RATE,
    AudioFormat.CHANNEL_OUT_MONO,
    AudioFormat.ENCODING_PCM_16BIT,
    2 * data.length, AudioTrack.MODE_STATIC);
audioTrack.flush();
audioTrack.write(data, 0, data.length);
audioTrack.play();
```

Wichtig ist dabei ob man Stereo oder Mono abspielen will und die SAMPLE_RATE, z.B.

```
private final int SAMPLE_RATE = 44100;
```

entspricht der Rate mit der typischerweise CDs aufgenommen werden. Mögliche Werte sind 8000, 11025, 16000, 22050 und 44100.

Zum Testen können wir mit der folgenden Methode einen Sinuston einer vorgegebenen Frequenz erzeugen:

```
private final double FREQUENCY = 440;

private short[] generateSound(int bufferSize) {
    short[] data = new short[bufferSize];
    double factor = 2 * Math.PI / (SAMPLE_RATE / FREQUENCY);
    for (int i = 0; i < data.length; i++) {
        data[i] = (short) (Short.MAX_VALUE * Math.sin(factor * i));
    }
    return data;
}
```

Auch beim AudioTrack handelt es sich eigentlich um eine C++ Klasse, d.h., wir müssen nach dem Aufruf der stop() Methode noch die *release()* Methode aufrufen, damit der AudioTrack ordentlich aufgeräumt wird.

RecordAudio

Natürlich können wir nicht nur Töne erzeugen, sondern wir können auch welche aufnehmen, mit der *AudioRecord* Klasse. Sie funktioniert ähnlich wie die AudioTrack Klasse:

```
AudioRecord audioRecord = new AudioRecord(
    MediaRecorder.AudioSource.MIC, SAMPLE_RATE,
    AudioFormat.CHANNEL_IN_MONO,
    AudioFormat.ENCODING_PCM_16BIT, bufferSize);

audioRecord.startRecording();

int length = audioRecord.read(buffer,0,bufferSize);

audioRecord.stop();
audioRecord.release();
```

In der Regel würden wir das Mikrofon als Quelle verwenden, es gibt aber auch andere Quellen, z.B. ein Telefongespräch. Wichtig ist der Buffer, der muss eine Mindestgröße haben, die können wir aber erfragen:

```
int bufferSize = AudioRecord.getMinBufferSize(
    SAMPLE_RATE,
    AudioFormat .CHANNEL_IN_MONO,
    AudioFormat.ENCODING_PCM_16BIT);
short[] buffer = new short[bufferSize];
```

Wenn wir längere Aufnahmen machen möchten, müssen wir um das *read()* einfach eine Schleife basteln:

```
// record for roughly 2 seconds:
for (int i = 0; i < 20; i++) {
    int length = audioRecord.read(buffer, 0, bufferSize);
    for (int j = 0; j < length; j++) {
        dos.writeShort(buffer[j]);
    }
}
```

In diesem Beispiel schreiben wir die Daten einfach in einen DataOutputStream, den wir vorher angelegt haben:

```
File f = new File(Environment.getExternalStorageDirectory(),
"AudioRecording.pcm");
FileOutputStream fos = new FileOutputStream(f);
BufferedOutputStream bos = new BufferedOutputStream(fos);
DataOutputStream dos = new DataOutputStream(bos);
```

Das so erzeugte File ist eine "PCM" Datei, enthält also die Audiodaten im Rohformat. Wir könnten sie natürlich mit der AudioTrack Klasse wieder abspielen. Der MediaPlayer (und andere Musikplayer) verstehen das PCM Format allerdings nicht. Das macht eigentlich wenig Sinn, denn das Windows WAV Format ist nichts anderes als das PCM Format mit einem Header versehen. Den kann man auch selbst anfügen, wie das gemacht wird kann man auf StackOverflow nachlesen [1].

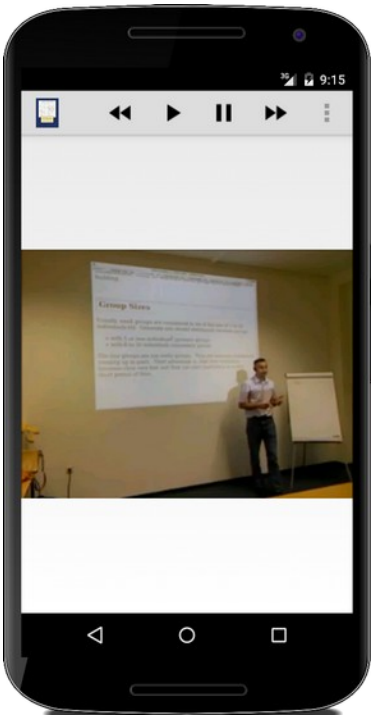
Eine Anmerkung noch, natürlich müssen wir den Nutzer um Erlaubnis fragen, ob wir Audioaufnahmen machen dürfen:

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

PlayVideo

Das Abspielen von Videos ist ähnlich einfach wie das Abspielen von Audios. Zunächst definieren wir die UI und verwenden das *VideoView* Widget:

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

    <VideoView
        android:id="@+id/videoView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_gravity="center" />

</LinearLayout>
```



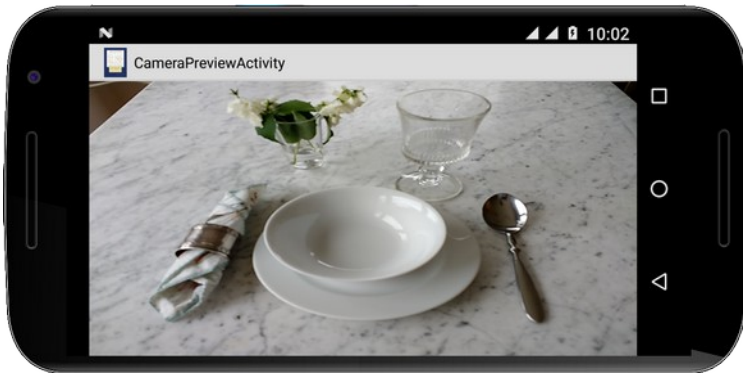
Dem VideoView muss man lediglich mitteilen wo die Videodatei zu finden ist,

```
VideoView video = (VideoView) findViewById(R.id.videoView);
//video.setVideoPath("/sdcard/socialweb.mp4");
Uri uri = Uri.parse("android.resource://" + getPackageName()
                    + "/" + R.raw.socialweb);
video.setVideoURI(uri);
video.start();
```

und mit *start()* wird das Video abgespielt. Der VideoView hat die gleichen Methoden wie der MediaPlayer was das Abspielen, Pausieren, usw. angeht.

Was allerdings signifikant schwieriger ist, ist Videos zu finden, die der VideoView auch abspielen kann. Zunächst sieht es nämlich so aus, wie wenn er jede MP4 Datei abspielen könnte. Dem ist aber nicht so. Man findet dazu erfreulich wenig Dokumentation und auch die Gerätehersteller scheinen da Unterschiede zu machen. Auf jeden Fall nach langwieriger Suche und gefühlt zehn Viren und Trojanern die ich mir dabei auf meinem Windowsrechner geholt habe, habe ich es geschafft eine Datei zu konvertieren.

CameraPreview



Fast alle Android Geräte haben Cameras heutzutage. Sehen wir uns mal an wie wir darauf zugreifen können. Auch hier müssen wir erst wieder die UI spezifizieren, dieses Mal verwenden wir den *SurfaceView*:

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

    <SurfaceView
        android:id="@+id/surfaceView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:gravity="center_horizontal" />

</LinearLayout>
```

Im Code holen wir uns eine Referenz auf selbigen,

```
SurfaceView surface =(SurfaceView)this.findViewById(R.id.surfaceView);
SurfaceHolder holder = surface.getHolder();
```

und lassen uns seinen *SurfaceHolder* geben (keine Ahnung was das ist). Den *holder* brauchen wir, weil wir nämlich einen Callback dranhängen wollen:

```
holder.addCallback(new SurfaceHolder.Callback(){
    private Camera mCamera;

    @Override
    public void surfaceChanged(SurfaceHolder holder, int format,
                                            int width, int height) {
        // do nothing
    }
```

```
        @Override
        public void surfaceCreated(SurfaceHolder holder) {
            try {
                mCamera = Camera.open();
                mCamera.setPreviewDisplay(holder);
                mCamera.startPreview();
            } catch (Exception e) {
                Log.e("CameraPreviewActivity", e.getMessage());
            }
        }

        @Override
        public void surfaceDestroyed(SurfaceHolder holder) {
            mCamera.stopPreview();
            mCamera.release();
            mCamera = null;
        }
    });
```

Dieser SurfaceHolder.Callback hat zwei Methoden die uns interessieren: die *surfaceCreated()* wird aufgerufen wenn der SurfaceView das erste Mal angezeigt wird. Hier verbinden wir die Kamera mit dem SurfaceView über den Holder. Die Kamera weiß dann was zu tun ist. Die Methode *surfaceDestroyed()* wird aufgerufen wenn wir fertig sind, hier müssen wir wieder ordentlich aufräumen.

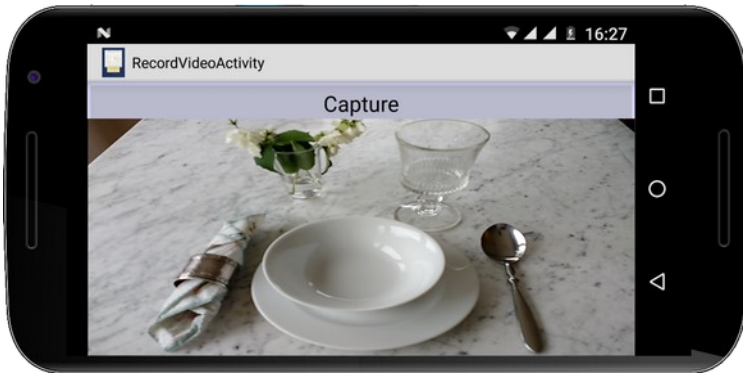
Natürlich möchte man nicht, dass jede beliebige Anwendung einfach auf die Kamera zugreifen darf, ohne dass der Nutzer zugestimmt hat. Deswegen muss man im AndroidManifest noch um Erlaubnis fragen:

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

Auch scheint der Preview im Landscape Modus besser zu funktionieren, deswegen sollte man vielleicht noch die Orientierung der App auf Landscape fixieren:

```
<activity
    android:name="variationenzumthema_ch8.CameraPreviewActivity"
    android:screenOrientation="landscape" />
```

RecordVideo



Unsere CameraPreview Activity zeigt zwar an was die Kamera sieht, aufnehmen tut sie aber nix. Dazu braucht man die *MediaRecorder* Klasse. Die ist ne Diva und will gut behandelt werden. Man kann die Details in [2] nachlesen, aber grob geht es wie folgt: erst mal muss man alles vorbereiten:

```
private boolean prepareVideoRecorder(String fileName) {
    mMediaRecorder = new MediaRecorder();

    // Step 1: Unlock and set camera to MediaRecorder
    mCamera.unlock();
    mMediaRecorder.setCamera(mCamera);

    // Step 2: Set sources
    mMediaRecorder.setAudioSource(
                                MediaRecorder.AudioSource.CAMCORDER);
    mMediaRecorder.setVideoSource(MediaRecorder.VideoSource.CAMERA);
```

```
        // Step 3: Set a CamcorderProfile (requires API Level 8 or higher)
        mMediaRecorder.setProfile(CamcorderProfile.get(
                                        CamcorderProfile.QUALITY_HIGH));

        // Step 4: Set output file
        mMediaRecorder.setOutputFile(fileName);

        // Step 5: Set the preview output
        mMediaRecorder.setPreviewDisplay(holder.getSurface());

        // Step 6: Prepare configured MediaRecorder
        try {
            mMediaRecorder.prepare();
        } catch (IllegalStateException e) {
            Log.d(getLocalClassName(),
                "IllegalStateException preparing MediaRecorder: "
                + e.getMessage());
            releaseMediaRecorder();
            return false;
        } catch (IOException e) {
            Log.d(getLocalClassName(),
                "IOException preparing MediaRecorder: " + e.getMessage());
            releaseMediaRecorder();
            return false;
        }
        return true;
    }
```

Danach kann man dann mit

```
mMediaRecorder.start();
```

loslegen. Mit *stop()* hört's dann wieder auf, und am Ende sollte man wieder aufräumen:

```
private void releaseMediaRecorder() {
    if (mMediaRecorder != null) {
        mMediaRecorder.reset(); // clear recorder configuration
        mMediaRecorder.release(); // release the recorder object
        mMediaRecorder = null;
        mCamera.lock(); // lock camera for later
    }
}
```

Natürlich will der SurfaceHolder auch gut behandelt werden, und die Kamera sowieso, deswegen ufert der Code dann doch etwas aus.

TextToSpeech

Mit der immer größeren Popularität von Siri und Alexa hat sich die Erwartungshaltung der Nutzer bzgl. Sprachausgabe und Spracherkennung von Apps geändert. Interessanterweise ist das gar nicht so schwer zu implementieren. Wir beginnen mit der Sprachausgabe.

Zunächst müssen wir sicherstellen, dass unser Gerät die richtigen Einstellungen hat: über die Accessibility-Settings (Language Settings in älteren Geräten) muss die Sprachausgabe eingeschaltet sein. Manchmal muss man aus dem Google Play Store auch die "Google Text-to-speech" App nachinstallieren.

Die eigentliche Arbeit macht dann die *TextToSpeech* Klasse. Diese initialisiert man, gibt ihr noch mit in welcher Sprache sie sprechen soll, und dann lässt man sie einfach drauf los plappern:

```
TextToSpeech tts =
    new TextToSpeech(getApplicationContext(),
        new TextToSpeech.OnInitListener() {
            @Override
            public void onInit(int status) {
                if (status != TextToSpeech.ERROR) {
                    tts.setLanguage(Locale.US);

                    tts.speak("hi there.",
                        TextToSpeech.QUEUE_FLUSH, null, null);
                    tts.speak("how are you?",
                        TextToSpeech.QUEUE_ADD, null, null);

                } else {
                    Log.e("TextToSpeechActivity",
                          "No speech engine available.");
                }
            }
    }
);
```

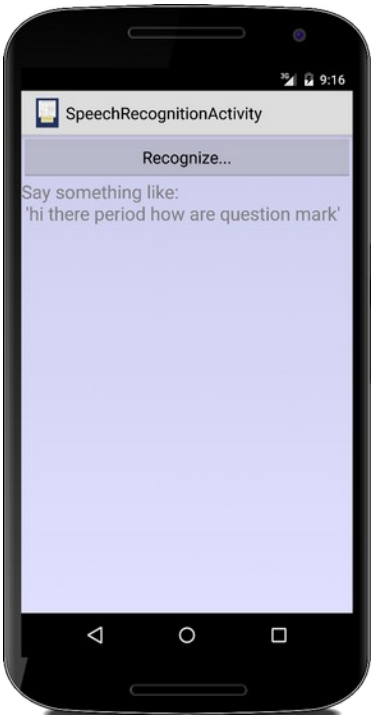
Wenn wir den TextToSpeech nicht mehr benötigen, dann sollten wir ihn anhalten, *stop()*, und herunterfahren, *shutdown()*:

```
@Override
public void onPause() {
    if (tts != null) {
        tts.stop();
        tts.shutdown();
        tts = null;
    }
    super.onPause();
}
```

SpeechRecognition

Das Gegenstück zu Sprachausgabe ist Spracherkennung. Auch das ist überraschend einfach mittels eines Intents, dem *RecognizerIntent*:

```
Intent intent = new Intent(
  RecognizerIntent.ACTION_RECOGNIZE_SPEECH);
intent.putExtra(
  RecognizerIntent.EXTRA_LANGUAGE_PREFERENCE,"en");
intent.putExtra(
  RecognizerIntent.EXTRA_LANGUAGE_MODEL,

RecognizerIntent.LANGUAGE_MODEL_FREE_FORM);
intent.putExtra(
  RecognizerIntent.EXTRA_MAX_RESULTS, 3);

startActivityForResult(intent, REQUEST_CODE);
```



Mit startActivityForResult() starten wir den Intent, und warten auf das Resultat:

```
protected void onActivityResult(int requestCode, int resultCode,
                                                    Intent data) {
    if (requestCode == REQUEST_CODE) {
        if (resultCode == RESULT_OK) {
            ArrayList<String> matches =
                data.getStringArrayListExtra(
                    RecognizerIntent.EXTRA_RESULTS);
            tv.setText(matches.get(0));
        } else {
            tv.setText("Something went wrong: " + resultCode);
        }
    } else {
        super.onActivityResult(requestCode, resultCode, data);
    }
}
```

In den Extras des *data* Intent finden wir dann, was die Spracherkennung erkannt hat. Da die Erkennung nicht immer hundertprozentig funktioniert, liefert uns der Intent gleich mehrere mögliche Alternativen, deswegen bekommen wir eine Liste. Das erste Element dieser Liste, ist das wahrscheinlichste, deswegen benutzt man selbiges in der Regel.

Noch zwei Anmerkungen: in den älteren Android-Versionen benötigt man noch eine aktive Internetverbindung, in den neueren geht die Spracherkennung aber auch offline. Und ähnlich wie bei der Sprachausgabe kann nicht jedes Handy automatisch Spracherkennung: manchmal muss man diese einschalten oder erst installieren. Wenn aber die normale Google-Suche mit Spracherkennung funktioniert, dann funktioniert auch unser Code.

Review

Wir haben gesehen wie wir Audiodateien mit dem MediaPlayer bzw. der Klasse AudioTrack abspielen können. Audioaufnahmen erledigen wir mit der AudioRecord Klasse. Für das Abspielen von Videodateien verwenden wir den SurfaceView, für die Aufnahme den MediaRecorder. Außerdem haben wir gesehen wie wir mit dem SurfaceView und der Camera Klasse einen Livefeed der Kamera anzeigen können. Schließlich haben wir noch mit der Klasse TextToSpeech und dem Intent ACTION_RECOGNIZE_SPEECH Sprachsynthese respektive Spracherkennung betrieben.

Projekte

Nach dieser kurzen Einführung in die grundlegenden Multimedia-Klassen wird es jetzt richtig interessant. Zum Aufwärmen erzeugen wir ein paar Töne und sehen uns das mit den Filtern noch einmal an. Interessanter wird es mit dem Mikrofon: da versuchen wir uns an einem Schnarchdetektor und einem Equalizer. Die Echo App ist schon anspruchsvoller und obwohl nicht perfekt ist auch die Sonar App ziemlich cool. Mit der Fourieranalyse wird es noch interessanter, wenn wir mit FlappyBall lernen Melodien zu pfeifen, wir unser Klavier endlich mal ordentlich stimmen können und uns ein Live-Spektrogramm unserer Stimme ansehen können. Mit der Kamera messen wir dann Distanzen, nehmen Fotos auf, schreiben einen Surveillance Service und faken eine 360 Grad Kamera. Dann kommt eine kleine Demonstration von Augmented Reality und FaceDetection ist mit Android eine triviale Übung. Mit Hilfe von Text-to-Speech erzeugen wir dann AudioBooks, aus unserem alten Freund ELIZA machen wir einen ChatBot, und schließlich wird noch diktiert. Allerdings aus unserem alten Handy eine echte IP Kamera zu machen, schießt wahrscheinlich den Vogel ab.

TuneGenerator

Wenn mal gerade keine Stimmgabel zur Hand ist, und wir unsere Gitarre oder unser Klavier schnell stimmen müssen, dann können wir das mit unserer TuneGenerator App. Wir gehen mal davon aus, dass wir eine "Gleichstufige Stimmung" [3] wollen, dann sind nämlich die Frequenzen der Töne sehr einfach zu berechnen [4]:

```
private double getPitch(int key) {
    double base = Math.pow(2.0, 1.0 / 12.0);
    double pitch = 440.0 *
        Math.pow(base, (key - 49));
    return pitch;
}
```

Dabei ist der Kammerton (A4 in USA) die 49te Taste von links auf einem normalen Klavier. Wenn wir jetzt noch wissen wollen wie der Ton heißt (im amerkanischen System), dann verwenden wir die Methode *getPitchName()*:

```
private final String[] pitchNames =
    { "C", "C#", "D", "D#", "E", "F",
      "F#", "G", "G#", "A", "A#", "B" };

private String getPitchName(int key) {
    String pitchName = pitchNames[(key + 8) % 12];
    pitchName += ((key + 8) / 12);
    return pitchName;
}
```

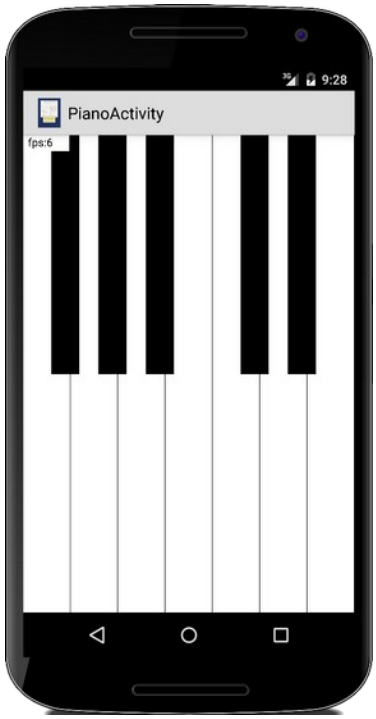
Wie wir einen Sinuston einer bestimmten Frequenz erzeugen, haben wir ja bereits in der *generateSound()* Methode im SimpleSoundGenerator weiter oben gesehen. Wir können aber noch einen Schritt weitergehen und kontinuierlich einen Ton erzeugen. Dazu muss der Sound allerdings in einem separaten Thread abgespielt werden:

```
public class TuneGeneratorActivity extends Activity
                                    implements Runnable {
    ...

    @Override
    public void onCreate(Bundle savedInstanceState) {
        ...
        new Thread(this).start();
    }

    @Override
    public void run() {
        bufferSize = AudioTrack.getMinBufferSize(
            SAMPLE_RATE, AudioFormat.CHANNEL_OUT_MONO,
            AudioFormat.ENCODING_PCM_16BIT);
        audioTrack = new AudioTrack(
            AudioManager.STREAM_MUSIC, SAMPLE_RATE,
            AudioFormat.CHANNEL_OUT_MONO,
            AudioFormat.ENCODING_PCM_16BIT,
            2 * bufferSize, AudioTrack.MODE_STREAM);
        audioTrack.flush();
```

```
            int step = 0;
            while (true) {
                if (isPlaying) {
                    generateSound(step++);
                    audioTrack.write(data, 0, data.length);
                    if (step == 1) {
                        audioTrack.play();
                    }
                }
            }
        }
        ...
    }
```

Dabei ist interessant, dass die *play()* Methode nur einmal aufgerufen wird, und wir danach die Audiodaten kontinuierlich über die *write()* Methode an den AudioTrack liefern.

Piano



Wir haben ja bereits im ersten Semester eine kleines Pianoprogramm geschrieben. Mit unserer selbstgeschriebenen ACM Graphicslibrary können wir das natürlich ganz einfach in eine App umwandeln. Nur das Abspielen der Töne müssen wir noch implementieren.

Wir könnten den MediaPlayer verwenden und ähnlich wie im ersten Semester vorgefertigte Soundfiles abspielen. Aber wir können auch die AudioTrack Klasse verwenden. Das gibt uns viel mehr Kontrolle. Wie wir gerade gesehen haben, sollte die Tonerzeugung in einem separate Thread laufen. Das können wir zusammenfassen in der Klasse *Player*, die als eigenständiger Thread läuft:

```
    public class Player implements Runnable {
        private int SAMPLE_RATE = 16000;
        private int bufferSize = 0;
        public boolean isPlaying = false;
        private final BlockingQueue<short[]> queue;

        public Player(BlockingQueue<short[]> queue,
                      int SAMPLE_RATE, int bufferSize) {
            this.queue = queue;
            this.SAMPLE_RATE = SAMPLE_RATE;
            this.bufferSize = bufferSize;
        }

        public void run() {
            try {
                isPlaying = true;
                AudioTrack audioTrack = new AudioTrack(
                    AudioManager.STREAM_MUSIC,
                    SAMPLE_RATE, AudioFormat.CHANNEL_OUT_MONO,
                    AudioFormat.ENCODING_PCM_16BIT, bufferSize,
                    AudioTrack.MODE_STREAM);
                audioTrack.flush();
                audioTrack.play();

                while (isPlaying) {
                    short[] data = queue.take();
                    audioTrack.write(data, 0, data.length);
                }

                audioTrack.stop();
                audioTrack.release();
```

```
            } catch (InterruptedException e) {
                Log.i("Player", e.getMessage());
            }
        }
    }
```

Der Player bekommt seine Daten über eine BlockingQueue geliefert (Producer-Consumer), er ist der Consumer, und spielt einfach die Daten ab, wie sie über die BlockingQueue reinkommen. Gefüttert wird die BlockingQueue vom Producer, unserem GraphicsProgram. Dort werden die Daten in der *mousePressed()* Methode erzeugt und in die BlockingQueue geschrieben:

```
public void mousePressed(int x, int y) {
    GObject obj = getElementAt(x, y);
    if (obj != null) {
        for (int i = 0; i < keys.length; i++) {
            if (obj == keys[i]) {
                try {
                    double pitch = getPitch(pitches[i] + 40);
                    short[] data = generateSound(pitch);
                    queue.put(data);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

WaveformGenerator

Wenn einem Sinuswellen zu langweilig sind, dann kann man natürlich auch beliebige andere Wellenformen erzeugen, z.B. Sinus-, Rechteck-, Sägezahn-, Dreieck- oder Treppenwellenformen. Dazu müssen wir lediglich die *generateSound()* Methode aus dem SimpleSoundGenerator etwas anpassen:

```
private void generateSound(int bufferSize) {
    data = new short[bufferSize];
    double factor =
        2 * Math.PI / (SAMPLE_RATE / FREQUENCY);
    for (int i = 0; i < data.length; i++) {
        data[i] = (short) (Short.MAX_VALUE * 0.95 *
            function(factor, i));
    }
}
```

wobei die *function()* jetzt eben nicht mehr nur eine einfache Sinusfunktion ist, sondern je nach dem Wert der Variablen *waveForm* eine der gewünschten Wellenformen:

```
private double function(double factor, int i) {
    switch (waveForm) {
    case 1:
        return functionSquare(factor, i);
    case 2:
        return functionSawTooth(factor, i);
    case 3:
        return functionTriangle(factor, i);
    case 4:
        return functionStaircase(factor, i);
```

```
        default:
            return functionSin(factor, i);
        }
    }
```

Hier ein paar kleine Codebeispiele wie man die verschiedenen Wellenformen erzeugen kann:

```
    private double functionSin(double factor, int i) {
        return Math.sin(i * factor);
    }

    private double functionSquare(double factor, int i) {
        double x = i * factor;
        x = x % (2 * Math.PI);
        if (x < Math.PI) {
            return -1;
        } else {
            return +1;
        }
    }

    private double functionSawTooth(double factor, int i) {
        double x = i * factor;
        x = x % (2 * Math.PI);
        return x / (Math.PI) - 1;
    }

    private double functionTriangle(double factor, int i) {
        double x = i * factor;
        x = x % (2 * Math.PI);
        if (x < Math.PI) {
            return 2 * x / (Math.PI) - 1;
        } else {
            return 2 * (2 * Math.PI - x) / (Math.PI) - 1;
        }
    }

    private double functionStaircase(double factor, int i) {
        double x = i * factor;
        x = x % (2 * Math.PI); // 0..2*PI
        x = x / (2 * Math.PI); // 0..1
        x = x * 10; // 0..9.999
        int s = (int) x; // 0..9
        s = s - 5; // -5..4
        return s / 5.0;
    }
```

Natürlich kann man die Wellenformen auch noch grafisch in einem GraphView darstellen, oder wir verbinden den Kopfhörerausgang mit einem Oszilloskop.

AudioFilter

Im Kapitel zu Sensoren haben wir von Hochpass- und Tiefpassfiltern gehört. Noch mal kurz zur Erinnerung, ein Hoch*pass* lässt die hohen Frequenzen *pass*ieren, ist aber für niedere Frequenzen undurchlässig, und beim Tiefpass ist es umgekehrt. Das kann man natürlich sehr schön demonstrieren mit Audiodateien.

Wie kommen wir aber an die Daten in einer Audiodatei? Für die Standard-Windows-Wave Dateien ist das relativ einfach. Wir öffnen sie mit einem DataInputStream,

```
InputStream is = getResources().openRawResource(R.raw.trap_beat2);
BufferedInputStream bis = new BufferedInputStream(is);
DataInputStream dis = new DataInputStream(bis);
readWavHeader(dis);
```

und in *readWavHeader()* entfernen wir einfach die ersten 44 Bytes:

```
private void readWavHeader(DataInputStream dis) throws IOException {
    for (int i = 0; i < 11; i++) {
        dis.readInt();
    }
}
```

Danach lesen wir dann Shorts mit *readShort(dis)* aus dem DataInputStream:

```
while (dis.available() > 0) {
    short[] data = new short[bufferSize];
    int i = 0;
    while (dis.available() > 0 && i < bufferSize) {
        data[i] = readShort(dis);
        i++;
    }
    ...
```

Der Grund warum wir die Methode *readShort(dis)* benötigen hat damit zu tun, dass Windows "little endian" ist [5] und wir die zwei Bytes des Shorts umdrehen müssen:

```
private short readShort(InputStream in) throws IOException {
    return (short) (in.read() | (in.read() << 8));
}
```

Nachdem wir die Daten jetzt eingelesen haben, müssen wir sie filtern, je nach Modus entweder die niederfrequenten oder die hochfrequenten:

```
    ...
    short[] filtered = null;
    switch (mode) {
    case 1:
        LOW_PASS_FACTOR = 0.3f;
        filtered = filterLowFrequencies(data);
        break;
    case 2:
        LOW_PASS_FACTOR = 0.9f;
        filtered = filterHighFrequencies(data);
        break;
    default:
        filtered = data;
        break;
    }

    audioTrack.write(filtered, 0, filtered.length);
}
```

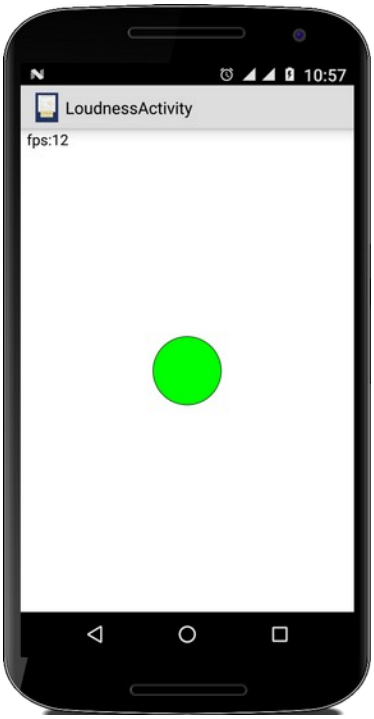
Die beiden Methoden wenden dabei die Filter auf die Daten an:

```
private short[] filterLowFrequencies(short[] recordingData) {
    short[] highPassArray = new short[recordingData.length];
    short avg = 0;
    for (int i = 0; i < recordingData.length; i++) {
        avg = lowPass(recordingData[i], avg);
        highPassArray[i] = highPass(recordingData[i], avg);
    }
    return highPassArray;
}

private short[] filterHighFrequencies(short[] recordingData) {
    short[] lowPassArray = new short[recordingData.length];
    short avg = 0;
    for (int i = 0; i < recordingData.length; i++) {
        avg = lowPass(recordingData[i], avg);
        lowPassArray[i] = avg;
    }
    return lowPassArray;
}
```

wobei *lowPass()* und *highPass()* die gleichen sind wie im Sensorkapitel:

```
private short lowPass(short current, short average) {
    return (short) (average * LOW_PASS_FACTOR + current * (1 -
LOW_PASS_FACTOR));
}

private short highPass(short current, short average) {
    return (short) (current - average);
}
```

Loudness



Wir haben schon lange keine Spiele mehr programmiert, es wird wieder Zeit. Bei Loudness geht es darum die Größe eines Balls (GOval) von der Lautstärke abhängig zu machen.

Wir schreiben also ein GraphicsProgram und im *setupUI()* fügen wir einen Ball hinzu:

```
public class LoudnessActivity extends
GraphicsProgram {
    ...
    public void run() {
        waitForTouch();
        setupUI();
        // game loop...
    }

    private void setupUI() {
        setBackground(Color.WHITE);

        ball = new GOval(BALL_MIN_SIZE,
                                    BALL_MIN_SIZE);
        ball.setFillColor(Color.GREEN);
        ball.setFilled(true);
        add(ball, getWidth() / 2, getHeight() / 2);
    }
}
```

Die Größe des Balls soll von der Lautstärke abhängen, also

```
private void changeBallSize(int loudness) {
    int size = loudness * getWidth() / MAX_LOUDNESS +
                                                    BALL_MIN_SIZE;

    int x = (getWidth() - size) / 2;
    int y = (getHeight() - size) / 2;
    ball.setBounds(x, y, size, size);
}
```

Nun kommen wir zur entscheidenden Frage, wie ermitteln wir die Lautstärke? Wir nehmen einfach den Durchschnitt der Absolutwerte der Rohdaten:

```
private int calculateLoudness(short[] copy) {
    double average = 0;
    for (int i = 0; i < copy.length; i++) {
        average += Math.abs(copy[i]);
    }
    average /= copy.length;
    return (int) average;
}
```

und die Rohdaten kommen von unserer AudioRecord Klasse:

```
public void run() {
    ...
    short[] data = new short[bufferSize];
    AudioRecord audioRecord = new AudioRecord(...);
    audioRecord.startRecording();

    while (isRecording) {
        int length = audioRecord.read(data, 0, bufferSize);
        int loudness = calculateLoudness(data);
        changeBallSize(loudness);
        pause(DELAY);
    }
    ...
}
```

SnoringService

Bevor ich die SnoringApp hatte, hat sich meine Frau immer beschwert, dass ich so laut schnarche. Aber jetzt seitdem ich die SnoringApp habe, kann auch meine Frau wieder ruhig schlafen. Die SnoringApp: ist ihren Preis wert, ich kann mir mein Leben gar nicht mehr ohne vorstellen.

Die SnoringApp ist in Prinzip nichts anderes als die Loudness Activity oben, mit dem Unterschied, dass wir ab einer gewissen Lautstärke einen Alarm starten (oder ein Vibrieren, Elektroschock, oder was auch immer notwendig ist um den Schnarchenden aufzuwecken):

```
if (averageLoudness > LOUDNESS_THRESHOLD) {
    isRecording = false;
    audioRecord.stop();

    // start alarm
    Intent intent = new Intent(this,
        SnoringAlarmActivity.class);
    startActivity(intent);
}
```

Eine subtile Änderung gibt es aber doch: da die Loudness Activity ein GraphicsProgram ist, wird im Hintergrund ein eigener Thread gestartet (deswegen ja auch die run() Methode). Für unsere Games ist das notwendig, damit der GameLoop unabhängig vom UI Thread, also dem Zeichnen ist. Bei der Aufnahme und dem Abspielen von Audio ist das ähnlich: das soll eigentlich in einem separaten Thread laufen. Da wir für die SnoringApp ja kein GraphicsProgram verwenden, müssen wir das selbst machen:

```
public class SnoringServiceActivity extends Activity
                                    implements Runnable {
    ...

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        new Thread(this).start();
    }

    @Override
    public void run() {
        ...
    }
}
```

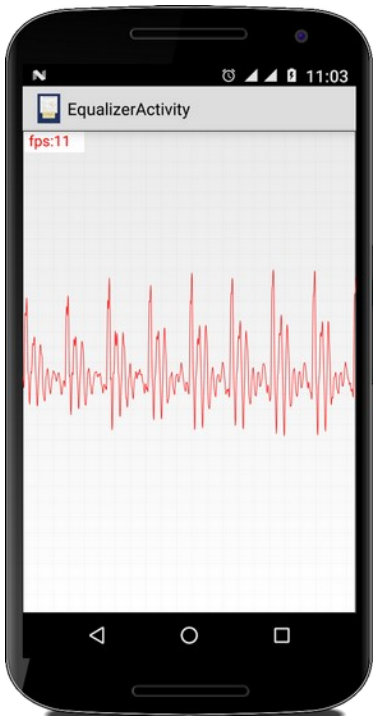
Die SnoringAlarmActivity ist einfach unsere PlayAudio Activity mit einem Knopf zum Ausschalten des Alarms. Für den Alarm selbst können wir natürlich irgendeine Audiodatei verwenden, oder aber wir benutzen den *RingtoneManager*:

```
Uri alert =
    RingtoneManager.getDefaultUri(RingtoneManager.TYPE_ALARM);
player = new MediaPlayer();
player.setDataSource(context, alert);
player.setAudioStreamType(AudioManager.STREAM_ALARM);
player.prepare();
player.start();
```

Für Testzwecke implementieren wir das als Activity, viel sinnvoller ist es aber das in einen Service umzuwandeln. Das hat vor allem Vorteile für den Ladezustand unserer Batterie.

Equalizer

Im Kapitel zu Sensoren haben wir die Klasse *GraphView* kennengelernt. Die kann man natürlich auch verwenden um unsere Audiodaten zu visualisieren, also einen Equalizer zu bauen. Genau wie in den zwei Apps oben, verwenden wir die AudioRecord Klasse,




```
public void run() {
    ...
    AudioRecord audioRecord = new AudioRecord(...);
    audioRecord.startRecording();

    while (isRecording) {
        int length =
            audioRecord.read(data, 0, bufferSize);

        for (int j = 0; j < data.length; j++) {
            gv.addDataPoint(data[j]);
        }
        gv.postInvalidate();
    }
    audioRecord.stop();
    ...
}
```

und schicken die Daten an einen GraphView, den wir in der *onCreate()* initialisiert haben:

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    gv = new GraphView2(this);
    gv.setMin(Short.MIN_VALUE);
    gv.setMax(Short.MAX_VALUE);
    gv.setStyle(GraphView2.GraphStyle.LINE);
    gv.setColor(Color.RED);
    gv.setStrokeWidth(1);
    setContentView(gv);

    new Thread(this).start();
}
```

Easy.

AudioRecorder

Es gibt zahlreiche Apps im Google Play Store die die Funktion eines Diktiergerätes haben. Mit den beiden Klassen AudioRecord und AudioTrack können wir so eine App auch selbst umsetzen.

Wir beginnen mit dem Aufnahmeteil unserer Anwendung: die Anwendung soll einen Button haben mit dem wir die Aufnahme starten und stoppen können,

```
public class AudioRecorderActivity extends Activity
{
    private boolean isRecording = false;

    @Override
    public void onCreate(Bundle savedState) {
        super.onCreate(savedInstanceState);
        setContentView(
            R.layout.audio_recorder_activity);

        final Button btn =
            (Button) findViewById(R.id.btnRecord);
        btn.setOnClickListener(
            new OnClickListener() {

            @Override
            public void onClick(View v) {
                if (isRecording) {
                    isRecording = false;
                    btn.setText("Record");
                } else {
                    isRecording = true;
                    btn.setText("Stop recording");

                    Date date = new Date();
                    SimpleDateFormat dateFormat =
                        new SimpleDateFormat("yyyyMMdd_HHmmss");
                    String fileName = dateFormat.format(date ".pcm";
                    Recorder recorder = new Recorder(fileName);
                    new Thread(recorder).start();
                }
            }
        });
        ...
    }
}
```

Die Aufnahme selbst lagern wir in die innere Klasse *Recorder* aus, die ein eigener Thread ist:

```
class Recorder implements Runnable {
    private String fileName;

    public Recorder(String fileName) {
        this.fileName = fileName;
    }

    @Override
    public void run() {
        File f = new File(
            Environment.getExternalStorageDirectory(), fileName);

        try {
            BufferedOutputStream bos = new BufferedOutputStream(
                new FileOutputStream(f));
            DataOutputStream dos = new DataOutputStream(bos);

            int bufferSize = AudioRecord.getMinBufferSize(
                SAMPLE_RATE, AudioFormat.CHANNEL_IN_MONO,
                AudioFormat.ENCODING_PCM_16BIT);
            short[] buffer = new short[bufferSize];

            AudioRecord audioRecord = new AudioRecord(
                MediaRecorder.AudioSource.MIC, SAMPLE_RATE,
                AudioFormat.CHANNEL_IN_MONO,
                AudioFormat.ENCODING_PCM_16BIT, bufferSize);

            audioRecord.startRecording();

            while (isRecording) {
                int length=audioRecord.read(buffer,0,bufferSize);
                for (int j = 0; j < length; j++) {
                    dos.writeShort(buffer[j]);
                }
            }

            audioRecord.stop();
            audioRecord.release();

            dos.close();
            bos.close();

            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    words.add(0, fileName);
                    ((ArrayAdapter) lv.getAdapter()).
                                        notifyDataSetChanged();
                }
            });

        } catch (Exception e) {
            Log.i(getLocalClassName(), e.getMessage());
        }
    }
}
```

Die Klasse Recorder muss deswegen eine innere Klasse sein, damit sie auf die Instanzvariable *isRecording* zugreifen kann. Interessant ist hier auch wie wir aus der Klasse Recorder auf den UI Thread mittels der *runOnUiThread()* Methode zugreifen: wenn wir fertig sind mit der Aufnahme, soll die neue Aufnahme ja in der UI angezeigt werden. Man hätte das auch mit einem AsyncTask tun können, aber ab und zu wollen wir ja auch mal was Neues lernen.

Kommen wir zu dem Teil der mit dem Abspielen zu tun hat. Hier ist es nicht nötig einen separaten Thread zu implementieren (es sei denn man möchte das Abspielen unterbrechen können). In der *onCreate()* fügen wir dafür noch einen ListView hinzu, der die bereits gemachten Aufnahmen auflisten soll:

```
@Override
public void onCreate(Bundle savedInstanceState) {
    ...

    words = getFilesWithExtension(".pcm");

    lv = (ListView) findViewById(R.id.listview);
    ArrayAdapter<String> adapter =
        new ArrayAdapter<String>(this,
            android.R.layout.simple_spinner_item, words);
    lv.setAdapter(adapter);
    lv.setOnItemClickListener(new OnItemClickListener() {

        @Override
        public void onItemClick(AdapterView<?> parent, View view,
                                int position, long id) {
            String fileName = (String)
              ((ArrayAdapter) lv.getAdapter()).getItem(position);
            playAudio(fileName);
        }
    });
}
```

Die *getFilesWithExtension()* Methode listet einfach alle Dateien mit einer bestimmten Endung auf:

```
private List<String> getFilesWithExtension(final String extension)
{
    List<String> words = new ArrayList<String>();
    File f = Environment.getExternalStorageDirectory();
    File[] files = f.listFiles(new FilenameFilter() {
        public boolean accept(File dir, String name) {
            return name.toLowerCase().endsWith(extension);
        }
    });
    for (int i = 0; i < files.length; i++) {
        words.add(0, files[i].getName());
    }
    return words;
}
```

Das Einzige was jetzt noch fehlt ist die *playAudio()* Methode:

```
private void playAudio(String fileName) {
    File f = new File(
        Environment.getExternalStorageDirectory(), fileName);

    try {
        BufferedInputStream bis = new BufferedInputStream(
            new FileInputStream(f));
        DataInputStream dis = new DataInputStream(bis);
```

```
                int bufferSize = AudioTrack.getMinBufferSize(SAMPLE_RATE,
                    AudioFormat.CHANNEL_OUT_MONO,
                    AudioFormat.ENCODING_PCM_16BIT);
                AudioTrack audioTrack = new AudioTrack(
                    AudioManager.STREAM_MUSIC,
                    SAMPLE_RATE, AudioFormat.CHANNEL_OUT_MONO,
                    AudioFormat.ENCODING_PCM_16BIT, bufferSize,
                    AudioTrack.MODE_STREAM);

                audioTrack.flush();
                audioTrack.play();

                while (dis.available() > 0) {
                    short[] buffer = new short[bufferSize];

                    int i = 0;
                    while (i < bufferSize && dis.available() > 0) {
                        buffer[i] = dis.readShort();
                        i++;
                    }

                    audioTrack.write(buffer, 0, bufferSize);
                }

                audioTrack.stop();
                audioTrack.release();

                dis.close();
                bis.close();

            } catch (Exception e) {
                Log.i(getLocalClassName(), e.getMessage());
            }
        }
```

Diese öffnet die ausgewählte PCM Datei, liest die Daten in einen Buffer und sendet den Buffer an den AudioTrack. Sieht komplizierter aus als es ist.

Echo

Ein App aus dem Bereich der Psychoakkustik ist die EchoActivity. Hier geht es darum sich selbst zu hören, allerdings um ein paar Zehntel Millisekunden zeitversetzt. Das Interessante dabei: man fängt an zu stottern, bzw. hört sich wie ein Idiot an. Der Effekt funktioniert allerdings nur mit Kopfhörer und Mikrofon.

Für die App verwenden wir zum einen die Player Klasse die wir bereits in unserer Piano Activity verwendet haben. Zusätzlich benötigen wir das Gegenstück dazu, die Recorder Klasse, die komplett analog dazu aufgebaut ist:

```
public class Recorder implements Runnable {
    private int SAMPLE_RATE = 16000;
    private int bufferSize = 0;
    public boolean isRecording = false;
    private final BlockingQueue<short[]> queue;

    public Recorder(BlockingQueue<short[]> queue,
                    int SAMPLE_RATE, int bufferSize) {
        this.queue = queue;
        this.SAMPLE_RATE = SAMPLE_RATE;
        this.bufferSize = bufferSize;
    }
```

```
        @Override
        public void run() {
            try {
                isRecording = true;
                short[] data = new short[bufferSize];
                AudioRecord audioRecord = new AudioRecord(
                    MediaRecorder.AudioSource.MIC, SAMPLE_RATE,
                    AudioFormat.CHANNEL_IN_MONO,
                    AudioFormat.ENCODING_PCM_16BIT, bufferSize);

                audioRecord.startRecording();

                while (isRecording) {
                    int length = audioRecord.read(data, 0, bufferSize);
                    // make a copy and add to queue
                    short[] copy = new short[bufferSize];
                    System.arraycopy(data, 0, copy, 0, copy.length);
                    queue.put(copy);
                }

                audioRecord.stop();
                audioRecord.release();

            } catch (Exception e) {
                Log.i("Recorder", e.getMessage());
            }
        }
    }
```

Das Einzige was hier besonders ist, dass wir eine Kopie des *data* Arrays machen. Das ist notwendig, da es sich ja bei einem Array um einen Referenzdatentypen handelt. Es ist nicht auszuschließen, dass die AudioRecord Klasse diese Referenz weiterverwendet um da neue Daten reinzuschreiben, was die alten überschreiben würde. Wenn wir aber eine Kopie haben, dann kann uns das egal sein.

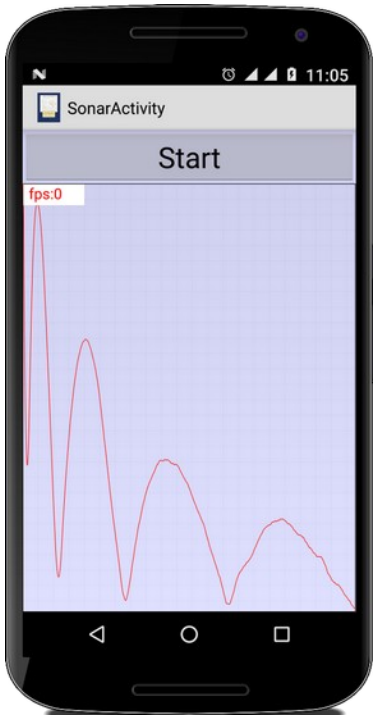
Mit diesen Vorarbeiten wird die eigentlich Echo Activity ganz einfach:

```
    public class EchoActivity extends Activity {
        ...
        private int bufferFactor = 5;

        private Player player;
        private Recorder recorder;
        private BlockingQueue<short[]> queue;

        @Override
        public void onCreate(Bundle savedInstanceState) {
            ...
            Button btn = (Button) findViewById(R.id.button);
            btn.setOnClickListener(new OnClickListener() {

                @Override
                public void onClick(View v) {
                    int minBufferSize = AudioRecord.getMinBufferSize(
                        SAMPLE_RATE,
                        AudioFormat.CHANNEL_IN_MONO,
                        AudioFormat.ENCODING_PCM_16BIT);
                    bufferSize = (minBufferSize * (bufferFactor + 2)) / 2;

                    queue = new ArrayBlockingQueue<short[]>(10);
```

```
                    player = new Player(queue, SAMPLE_RATE, bufferSize);
                    new Thread(player).start();

                    recorder =
                        new Recorder(queue, SAMPLE_RATE, bufferSize);
                    new Thread(recorder).start();
                }
            });
        }
        ...
    }
```

Recorder und Player folgen dem klassischen Producer-Consumer Pattern, über die ArrayBlockingQueue werden die Audiodaten vom Recorder an den Player geschickt. Die Verzögerung von ein paar Zehntel Millisekunden wird über die Größe des Buffers erzeugt: da der Player die Daten ja nicht direkt Byte für Byte, sondern immer Paketweise bekommt, hängt der Delay von der Größe der Datenpakete ab. Je größer die Pakete, desto länger der Delay.

Sonar



Als nächstes wollen wir eine App schreiben mit der man U-Boote finden kann. Im Ernst, mit dieser App kann man grob die Distanz zwischen dem Smartphone und der nächsten großen Wand finden. Mit etwas mehr Aufwand, kann man sogar noch genauere Messungen machen.

Die Idee ist ganz einfach: man sendet ein kurzes Geräusch und wartet auf das Echo. Je länger das Echo benötigt, desto weiter ist etwas weg. Da sich Schall mit einer Geschwindigkeit von ca. 330 m/s ausbreitet, kann man daraus die Entfernung berechnen.

Bevor wir mit dem Coden anfangen können, müssen wir uns aber Gedanken machen bzgl. der Frequenz mit der wir senden und der Samplingrate.

Nehmen wir an wir hätten eine Samplingrate von 1 Sample pro Sekunde. Dann könnten wir bestenfalls Objekte lokalisieren die 330 Meter weg sind. Das ist ziemlich nutzlos, denn je weiter etwas weg ist, desto schwächer ist natürlich auch das Echo. Deswegen wollen wir eine Samplingrate die so hoch wie möglich ist: bei 44100 Samples pro Sekunde ist unsere zeitliche Auflösung bei 0.02 Millisekunden, was ca. 0.7 cm entspricht.

Kommen wir zur Frequenz mit der wir unser Signal aussenden sollten: wenn wir mal 330 Hz als Beispiel nehmen, dann bedeutet das, dass wir 330 Schwingungen pro Sekunde haben. Da sich unser Signal aber auch mit 330 m/s ausbreitet, bedeutet das, dass "eine Welle" etwa einen Meter lang ist. Daraus erkennen wir, dass je höher die Frequenz, desto besser auch unsere Auflösung ist. Können wir eine beliebig hohe Frequenz wählen? Nein leider nicht, da gibt uns das Abtasttheorem von Nyquist-Shannon [6] ein oberes Limit: die Frequenz kann maximal die Hälfte der Samplingrate sein. Also höchstens 22500 Hz.

Beginnen wir mit dem Coden: die App ist eine einfache Activity. Wir haben einen Knopf mit dem wir den Sonar starten und einen GraphView der uns das Resultat anzeigt:

```
public class SonarActivity extends Activity {
    private final int NR_OF_WAVES = 4;
    private final double FREQUENCY = 8800;
    private final int SAMPLE_RATE = 44100;
    private int waveForm = 1;

    private GraphView2 gv;
```

```
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.sonar_activity);

        Button btnStart = (Button) findViewById(R.id.btnStart);
        btnStart.setOnClickListener(new OnClickListener() {

            @Override
            public void onClick(View v) {
                int bufferSize = SAMPLE_RATE / 10;
                Recorder recorder = new Recorder(bufferSize);
                new Thread(recorder).start();

                playSound();
            }
        });

        gv = (GraphView2) findViewById(R.id.graphview);
        gv.setMin(Short.MIN_VALUE);
        gv.setMax(Short.MAX_VALUE);
        gv.setStyle(GraphView2.GraphStyle.LINE);
        gv.setColor(Color.RED);
        gv.setStrokeWidth(1);
    }
}
```

Wenn der Knopf gedrückt wird, starten wir als erstes die Aufnahme als eigenen Thread. Wir nehmen für eine Zehntelsekunde auf (SAMPLE_RATE / 10), was einer Auflösung von 33 Metern entspricht. Erst nachdem die Aufnahme läuft, erzeugen wir mit playSound() ein kurzes Geräusch. Die playSound() Methode haben wir aus dem WaveformGenerator geborgt:

```
    private void playSound() {
        double time = NR_OF_WAVES / FREQUENCY;
        int bufferSize = (int) (time * SAMPLE_RATE) + 1;
        generateSound(bufferSize, waveForm);

        audioTrack = new AudioTrack(AudioManager.STREAM_MUSIC,
                SAMPLE_RATE, AudioFormat.CHANNEL_OUT_MONO,
                AudioFormat.ENCODING_PCM_16BIT, 2 * data.length,
                AudioTrack.MODE_STATIC);
        audioTrack.flush();
        audioTrack.write(data, 0, data.length);
        audioTrack.play();
    }
```

Hier gibt es zwei Größen, die wir festlegen müssen: die *NR_OF_WAVES* und die *waveForm*. Da muss man ein bisschen probieren, bei mir haben vier Wellen, also NR_OF_WAVES = 4, gut funktioniert und als waveForm haben sich Rechteckwellen als am besten geeignet herausgestellt.

Sehen wir uns jetzt den Recorder an: der sieht genauso aus wie unsere anderen Recorder Threads, allerdings machen wir nur eine Aufnahme, und die Daten schicken wir dann an die *evaluateDistances()* Methode zum Auswerten:

```
class Recorder implements Runnable {
    private int bufferSize = 0;

    public Recorder(int bufferSize) {
        this.bufferSize = bufferSize;
    }
```

```
        @Override
        public void run() {
            try {
                short[] data = new short[bufferSize];
                AudioRecord audioRecord = new AudioRecord(
                    MediaRecorder.AudioSource.MIC,
                    SAMPLE_RATE, AudioFormat.CHANNEL_IN_MONO,
                    AudioFormat.ENCODING_PCM_16BIT, bufferSize);
                audioRecord.startRecording();

                int length = audioRecord.read(data, 0, bufferSize);

                short[] copy = new short[bufferSize];
                System.arraycopy(data, 0, copy, 0, copy.length);

                audioRecord.stop();
                audioRecord.release();
                audioRecord = null;

                evaluateDistances(copy);

            } catch (Exception e) {
                Log.i(getLocalClassName(), e.getMessage());
            }

        }
    }
```

In der *evaluateDistances()* Methode, könnten wir jetzt unsere Rohdaten einfach im GraphView anzeigen:

```
        private void evaluateDistances(short[] recordingData) {
            double[] crossCorrelationArray =
                crossCorrelation(recordingData);
            gv.setMin(min);
            gv.setMax(max);
            for (int i = maxK; i < gv.getSize() + maxK; i++) {
                gv.addDataPoint(crossCorrelationArray[i]);
            }
            gv.postInvalidate();
        }
```

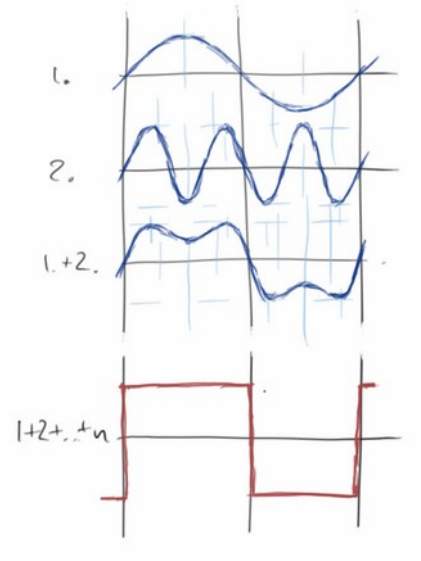
Es stellt sich aber heraus, dass man da nicht wirklich viel sieht, außer ein paar kleinen Bumps. Allerdings wenn man mal kurz was in der Wikipedia zu Cross-Correlation liest [7] (oder beim Professor Carl in der Vorlesung aufgepasst hat),

```
        private double min, max;
        private int maxK = 0;

        private double[] crossCorrelation(short[] recordingData) {
            min = Double.MAX_VALUE;
            max = Double.MIN_VALUE;
            double[] crossCorrelationArray =
                new double[recordingData.length];
            for (int i = 0; i < recordingData.length - data.length; i++) {
                double tmpi = 0;
                for (int j = 0; j < data.length; j++) {
                    tmpi += Math.abs(data[j] * recordingData[i + j]);
                }
                crossCorrelationArray[i] = tmpi;
                min = Math.min(min, crossCorrelationArray[i]);
```

```
                if (max < crossCorrelationArray[i]) {
                    max = crossCorrelationArray[i];
                    maxK = i;
                }
            }
            return crossCorrelationArray;
        }
```

dann kann man auf einmal das Hauptsignal und das erste Echo ganz klar erkennen. Bei einer Samplingrate von 44100 Hz entspricht ein Pixel auf dem GraphView in etwa 0.7 cm. Oder da die Linien im Grid des GraphView 50 Pixel auseinanderliegen, was in etwa 37 cm entspricht. So, jetzt können wir, wenn unser Smartphone wasserdicht ist, auf einem großen Ozean nach U-Booten suchen...

Fast Fourier Transform (FFT)



Obertöne [8] geben jedem musikalischen Instrument seinen charakteristischen Klang. Auch der Klang einer Stimme, wie z.B. der von Freddie Mercury [9], wird nicht unerheblich von den Obertönen beeinflusst. Es gibt sogar das reine "Overtone Singing" [10], was auch für den charakteristischen Klang des Kehlgesangs aus Tuwa [11] verantwortlich ist.

Hauptsächlich verantwortlich für den Klang ist das Verhältnis der verschiedenen Obertöne zueinander, vor allem deren Lautstärke, auch Amplitude genannt. Sind diese bekannt, kann man den Klang eines Instruments dadurch erzeugen in dem man einfach den Grundton und alle seine Obertöne im entsprechenden Verhältnis aufaddiert.

Der umgekehrte Prozess, wie man also einen Klang in seine Obertöne zerlegt, nennt man Fourier-Transformation [12]. Wie das genau funktioniert braucht uns eigentlich nicht zu interessieren, wir können aber unsere Mathematiker Freunde mal fragen, die werden dann auf einmal sehr redselig.

In den nächsten vier Beispielen wollen wir uns mit ein paar sehr praktischen Anwendung der Fourier-Transformation beschäftigen. Für unsere Experimente verwenden wir die FFT Klasse aus dem Projekt MEAPsoft der Columbia University [13], welcher wiederum auf Code von Douglas L. Jones basiert.

Die Klasse FFT ist relativ einfach zu benutzen. Zunächst rufen wir den Konstruktor auf, dem wir sagen müssen wie groß unser Audiobuffer ist:

```
bufferSize = nextpow2(bufferSize);
FFT fft = new FFT(bufferSize);
```

dabei ist eine Eigenheit des FFT, dass die *buffersize* eine Zweierpotenz sein muss. Die Anwendung ist dann denkbar einfach:

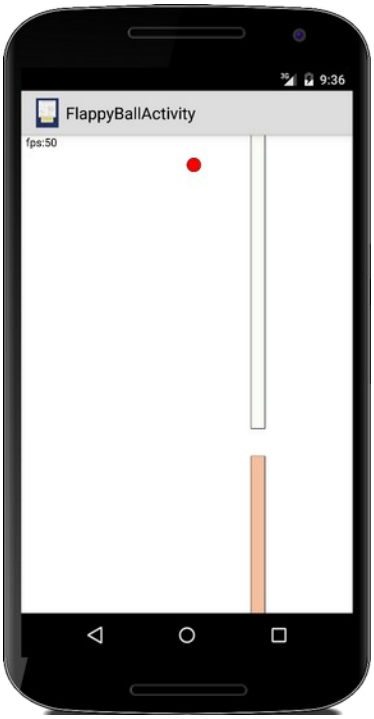
```
...
short[] data = new short[bufferSize];
audioRecord.read(data, 0, bufferSize);
double[] magn = fft.doSimpleFFT(data);
```

Wir nehmen die Audiodaten wie gewohnt in einem Array auf, und schicken diese dann an die Methode *doSimpleFFT()* der Klasse FFT. Das Array *magn* das wir zurückbekommen enthält das Fourierspektrum. Dazu bekommen wir auch noch gleich die kleinste und größte Amplitude mit *getMin()* und *getMax()* geliefert. Außerdem können wir die Frequenz des lautesten Tons, *maxPitch*, bestimmen:

```
double LOWEST_FREQUENCY = SAMPLE_RATE / (double) bufferSize;
maxPitch = fft.getMaxK() * LOWEST_FREQUENCY;
```

Häufig interessiert uns nur dieser *maxPitch*, manchmal aber auch das gesamte Spektrum.

FlappyBall

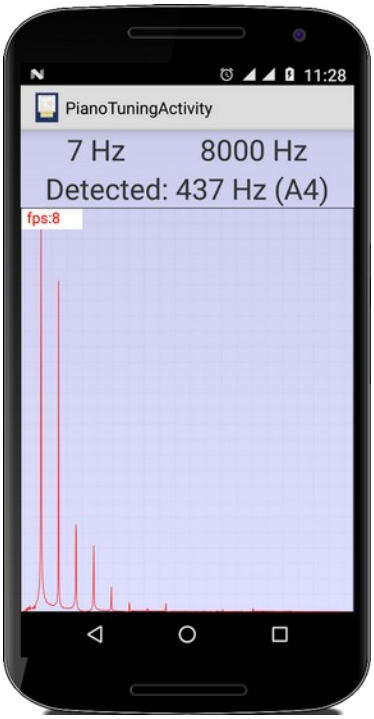


Eine erste einfache Anwendung für unsere FFT Klasse ist FlappyBall. Die Idee ist, dass die Höhe des Balls durch Singen oder Pfeifen beeinflusst wird. Wir nehmen den Code aus dem ersten Semester und ändern lediglich die *moveBall()* Methode:

```
private final int MAX_FREQUENCY = 2000;
private final int MIN_FREQUENCY = 400;
private final int LOW_PASS_FACTOR = 20;

private double pitch = MIN_FREQUENCY;
private double pitchAvg = MIN_FREQUENCY;

private void moveBall() {
    if (pitch > MIN_FREQUENCY &&
        pitch < MAX_FREQUENCY) {
        pitchAvg = (LOW_PASS_FACTOR * pitchAvg
            + pitch) / (LOW_PASS_FACTOR + 1);
        double y = getHeight() - BALL_DIAM
            - ((pitchAvg - MIN_FREQUENCY)
            / (MAX_FREQUENCY - MIN_FREQUENCY))
            * getHeight();

        ball.setLocation(getWidth() / 2, (int) y);
    }
}
```

Interessant ist hier die Instanzvariable *pitch*: die wird nämlich im *Recorder* Thread verändert. Die *Recorder* Klasse ist ganz ähnlich aufgebaut wie im *Echo* Beispiel oben: die BlockingQueue benötigen wir nicht, bei der *buffersize* müssen wir wie oben darauf achten, dass es eine Zweierpotenz sein muss, und im Recording-Loop führen wir die Fourier-Transformation durch:

```
while (isRecording) {
    int length = audioRecord.read(data, 0, bufferSize);
    double[] magn = fft.doSimpleFFT(data);
    double LOWEST_FREQUENCY = SAMPLE_RATE / (double) bufferSize;
    pitch = fft.getMaxK() * LOWEST_FREQUENCY;
}
```



Am Ende setzen wir die Instanzvariable *pitch* auf den Wert der lautesten Frequenz. Wenn man jetzt die Lücke in den Wänden nicht zufällig erzeugt würde, sondern basierend auf einer Melodie, könnte man mit unserem Programm sogar spielerisch das Pfeifen lernen.

PianoTuning

Die PianoTuningActivity ist ganz ähnlich wie die FlappyBall App, denn uns interessiert hauptsächlich die Frequenz des lautesten Tons. Aus dieser Frequenz können wir dann die Klaviertaste ermitteln die gedrückt wurde (wenn das Klavier einigermaßen gestimmt ist). Im TuneGenerator Beispiel haben wir ja schon die Methoden *getPitch()* und *getPitchName()* gesehen. Was wir noch brauchen ist das Gegenstück zu getPitch() und zwar *getKey()*:

```
private int getKey(double pitch) {
    double base = Math.pow(2.0, 1.0 / 12.0);
    double k = (Math.log(pitch) - Math.log(440.0))
             / Math.log(base) + 49;
    if (k >= 0 && k <= 100) {
        return (int) Math.round(k);
    }
    return 0;
}
```

Diese wandelt eine gegebene Frequenz (pitch) in eine Taste auf dem Klavier um.

Was wir aber außerdem machen können ist das Fourierspektrum anzeigen. Wir nehmen also das Array *magn*, das wir bisher ignoriert haben, und schicken es an einen GraphView:

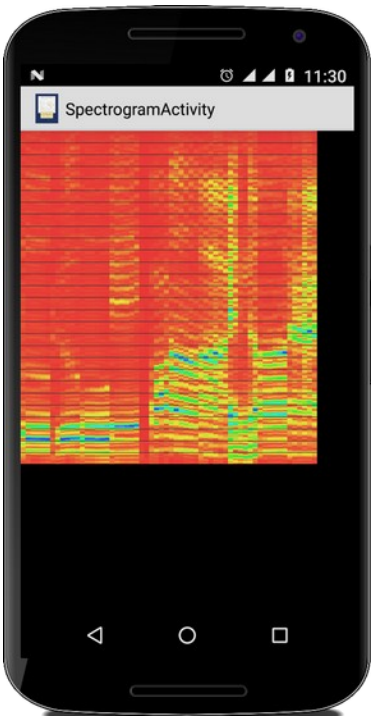
```
GraphView2 gv = (GraphView2) findViewById(R.id.graphview);
...

gv.reset();
gv.setMax(fft.getMax());
gv.setMin(0);
int len = Math.min(gv.getSize(), magn.length);
for (int j = 0; j < len; j++) {
    gv.addDataPoint(magn[j]);
}
// move to the left
int delta = gv.getSize() - len;
for (int j = 0; j < delta - 10; j++) {
    gv.addDataPoint(0);
}
gv.postInvalidate();
...
```

Interessant ist die Framerate: auf meinem Moto G4 läuft das Ganze mit ca. 10 fps, ohne irgendwelche besonderen Optimierungen. Impressive.

Wenn wir die App an einem echten Klavier ausprobieren, stellen wir fast, dass sie für die hohen Töne wunderbar funktioniert. Allerdings bei den tiefen liegt sie häufig daneben. Das hat damit zu tun, dass nicht immer der Grundton der lauteste ist. Unsere App sucht aber nur nach dem lautesten.

Spectrogram

Im letzten Beispiel haben wir das Fourierspektrum einfach als Graph dargestellt, der sich mit der Zeit ändert. Man kann es aber auch so darstellen, dass man die Amplituden des Spektrums mit Farben kodiert, z.B. rot für kleine Amplituden und blau für große Amplituden. Wenn man diese farbigen Linien dann nebeneinander fortlaufend weiterschreibt, dann nennt man das ein Spektrogramm [14].

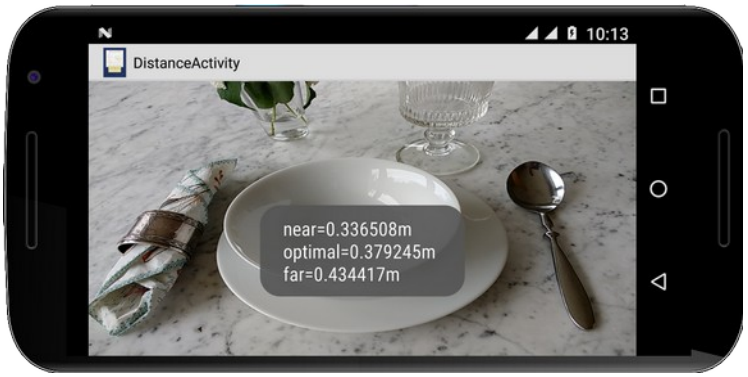


Der Code dafür basiert auf der PianoTuningActivity: in einem eigenen Thread nehmen wir vom Mikrofon auf, machen die Fourieranalyse und zeichnen das Spektrum in einen *SpectrumView*. Der SpectrumView nimmt das Spektrum das der FFT liefert und macht daraus jeweils ein kleines GImage. Von diesen GImages merkt er sich die letzten zehn Stück und zeichnet diese bei jedem Redraw. Die Details kann man im Code nachsehen, der nicht ganz trivial ist. Dass das überhaupt funktioniert spricht zum einen für die Leistungsfähigkeit moderner Android Geräte, aber auch für das Android Betriebssystem.

SoundWave

SoundWave [15] ist ein Forschungsprojekt von Microsoft. Dabei geht es darum mit Hilfe des Dopplereffekts Gesten zu erkennen. Dazu erzeugt man einen konstanten Ton und nimmt gleichzeitig mit dem Mikrofon auf. Dann fuchtelt man mit der Hand ein bisschen hin und her, und man kann sofort erkennen, dass sich die Amplitude das aufgenommenen Signals ändert. Auch das Verhältnis der Obertöne ändert sich. Anscheinend kann man daraus mit etwas Fantasie und Magie (= künstliche Intelligenz) Gesten erkennen. Vielleicht macht es noch Sinn die Frequenzen außerhalb des hörbaren Bereichs zu wählen.

Distance



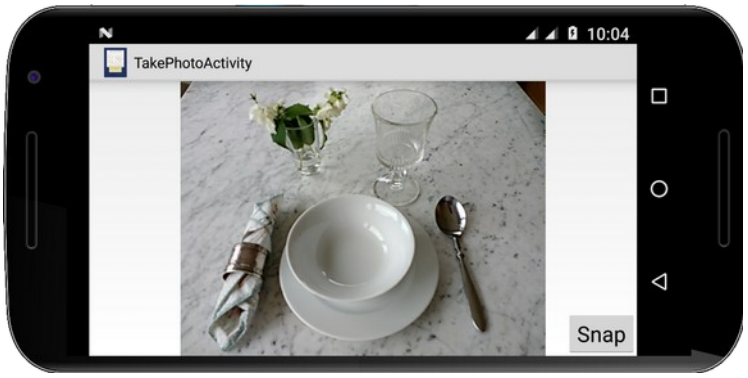
Bei den Camera.Parametern gibt es eine interessante Methode: *getFocusDistances()*. Dies würde darauf schließen lassen, dass man mit der Kamera Distanzen messen kann. Sehen wir uns das mal an.

Wir nehmen einfach unsere CameraPreview Activtiy und machen ein paar kleine Modifikationen in der *surfaceCreated()* Methode:

```
public void
surfaceCreated(SurfaceHolder holder) {
    try {
        mCamera = Camera.open();

        mCamera.autoFocus(new AutoFocusCallback() {
            @Override
            public void onAutoFocus(boolean success, Camera camera) {
                float distances[] = new float[3];
                camera.getParameters().getFocusDistances(distances);
                Log.i("DistanceActivity", "distances are: near="
                    + distances[0] + "m, optimal="
                    + distances[1] + "m, far=" + distances[2] + "m");
            }
        });

        Camera.Parameters params = mCamera.getParameters();
        params.setFocusMode(
                    Camera.Parameters.FOCUS_MODE_CONTINUOUS_PICTURE);
        mCamera.setParameters(params);

        mCamera.setPreviewDisplay(holder);
        mCamera.startPreview();
    } catch (Exception e) {
        Log.e("DistanceActivity", e.getMessage());
    }
}
```

Dabei gibt die Kamera einem nur einen groben Bereich, also eine Abschätzung zwischen *near*, *optimal* und *far* an. Aber immerhin.

TakePhoto



Im Beispiel CameraPreview haben wir ja schon gesehen wie man auf die Kamera zugreifen und das Kamerabild in einem SurfaceView anzeigen kann. Was wir aber noch nicht gesehen haben, wie wir einfache Schnappschüsse machen können.

Wir beginnen mit den Instanzvariablen, wir brauchen natürlich eine Referenz auf die Kamera, dann wollen wir das geschossene Bild ja anzeigen, deswegen einen ImageView, und wir benötigen auch noch eine *SurfaceTexture*:

```
private Camera camera;
private ImageView iv;
private SurfaceTexture surfaceTexture;
```

Die SurfaceTexture wird von der Kamera anstelle des SurfaceViews benötigt. Anscheinend ist die Kamera ohne nicht glücklich.

In der onCreate() basteln wir wie üblich unsere UI zusammen:

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.take_photo_activity);

    iv = (ImageView) this.findViewById(R.id.imageView);

    Button btn = (Button) this.findViewById(R.id.btnSnap);
    btn.setOnClickListener(new OnClickListener() {

        @Override
        public void onClick(View v) {
            takePicture();
        }
    });

    surfaceTexture = new SurfaceTexture(42);
}
```

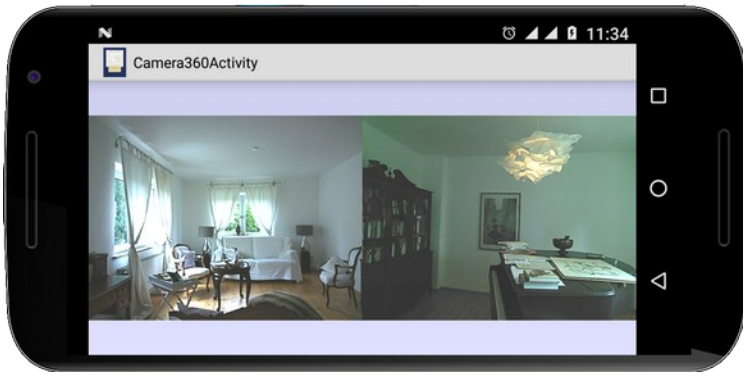
Nichts besonderes hier, die 42 ist absolut willkürlich. Interessant wird jetzt die *takePicture()* Methode: wir wählen die Kamera aus, setzen die SurfaceTexture und die Camera.Parameter:

```
private void takePicture() {
    try {
        if (camera == null) {
            camera = Camera.open(0);
            camera.setPreviewTexture(surfaceTexture);

            Camera.Parameters params = camera.getParameters();
            params.setFlashMode(Camera.Parameters.FLASH_MODE_OFF);
            params.setPictureSize(640, 480);
            params.setPictureFormat(ImageFormat.JPEG);
            camera.setParameters(params);
        }

        camera.startPreview();
        camera.takePicture(null, null, new PictureCallback() {
```

```
                @Override
                public void onPictureTaken(byte[] data, Camera camera) {
                    Bitmap bitmap = BitmapFactory.decodeByteArray(data,
                                                            0, data.length);
                    iv.setImageBitmap(bitmap);

                    camera.stopPreview();
                }

            });

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

Danach beginnen wir mit dem Preview, und machen ein Bild. Sobald das Bild fertig ist (dauert ein bisschen), wird die Methode *onPictureTaken()* des *PictureCallback* aufgerufen. Und dort können wir dann aus den Rohdaten, *data*, eine Bitmap erzeugen, die wir dann im ImageView anzeigen. Easy.

Camera360



Es gibt ja inzwischen ziemlich teure 360 Grad Kameras zu kaufen. Was eigentlich bescheuert ist, weil ja heutzutage fast jedes Smartphone zwei Kameras hat, eine nach vorne und eine nach hinten. Man müsste doch einfach beide Kameras gleichzeitig ansprechen und schon hat man eine 360 Grad Kamera.

Stellt sich heraus, dass man auf fast keinem Smartphone beide Kameras gleichzeitig ansprechen kann. Das ist aber kein Problem für uns, wir sprechen einfach eine nach der anderen an, abwechselnd. Der Aufbau ist fast identisch mit dem TakePhoto Projekt, wir haben lediglich zwei ImageViews.

Damit die Bilder kontinuierlich angezeigt werden, machen wir aus unserer Activity einen Thread, und in dessen *run()* Methode nehmen wir dann abwechselnd ein Bild mit der Kamera Nummer 0 (back-facing) und der mit der Nummer 1 (front-facing) auf:

```
public class Camera360Activity extends Activity implements Runnable {
    ...
    private int cameraId = 0;
    private boolean isCameraBusy = false;


    @Override
    public void run() {
        while (true) {
            if (!isCameraBusy) {
                cameraId++;
                cameraId %= 2;
                isCameraBusy = true;
                takePicture();
            }
            pause(10);
        }
    }
```

```
        private void takePicture() {
            try {
                camera = Camera.open(cameraId);
                ...
            }
        }
```

Schon wieder 300 Euro gespart.

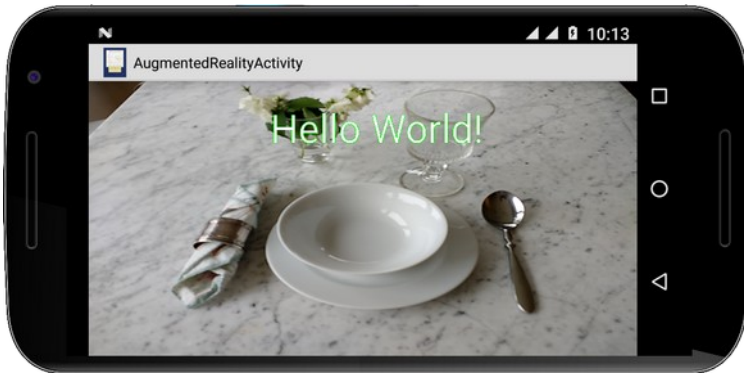
SurveillanceService

Eine weitere Anwendung für unsere *takePicture()* Methode ist der SurveillanceService. Die Idee ist recht einfach: einmal in der Stunde, Minute oder Sekunde ein Bild machen und auf der SD-Karte speichern. Dazu müssen wir lediglich die *onStartCommand()* Methode unseres Services überschreiben:

```
    public int onStartCommand(Intent intent,
                              int flags, int startId) {
        picturesDir =
            new File(Environment.
                getExternalStoragePublicDirectory(
                    Environment.DIRECTORY_PICTURES),
                    "SurveillanceService");
        picturesDir.mkdirs();

        // delay in ms before task is executed
        long delay = 0;
        // time in ms between successive executions
        long period = 60 * 1000;

        timer = new Timer();
        timer.schedule(new TimerTask() {
            @Override
            public void run() {
                takePicture();
            }
        }, delay, period);

        // if we get killed, restart
        return START_STICKY;
    }
```

Natürlich benötigen wir noch die zugehörige Activity zum Starten des Services. Wenn man möchte kann man in dieser Activity auch gleich die gemachten Bilder anzeigen oder einen Movie daraus machen (siehe IPCam).

AugmentedReality



AugmentedReality ist momentan der große Hype, deswegen wollen wir uns das mal etwas näher ansehen. Wir beginnen damit unsere CameraPreview Activtiy erst einmal auseinanderzunehmen, und alles was mit dem SurfaceView zu tun hat in eine eigene Klasse *CameraView* zu schreiben:

```
private class CameraView extends SurfaceView
                         implements SurfaceHolder.Callback {
    private Camera mCamera;

    public CameraView(Context context) {
        super(context);

        SurfaceHolder holder = this.getHolder();
        holder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
        holder.addCallback(this);
    }

    @Override
    public void surfaceCreated(SurfaceHolder holder) {
        try {
            mCamera = Camera.open();
            mCamera.setPreviewDisplay(holder);
            mCamera.startPreview();
        } catch (Exception e) {
            Log.e("CameraView", e.getMessage());
        }
    }

    @Override
    public void surfaceChanged(SurfaceHolder holder, int format,
                                              int width, int height) {
        // do nothing
    }

    @Override
    public void surfaceDestroyed(SurfaceHolder holder) {
        mCamera.stopPreview();
        mCamera.release();
        mCamera = null;
    }
}
```

CameraView ist also sowohl ein SurfaceView als auch ein SurfaceHolder.Callback.

Als nächstes schreiben wir eine ganz triviale Klasse *OverlayView*, die ein einfacher View ist und lediglich einen Text mit "Hello World!" anzeigt:

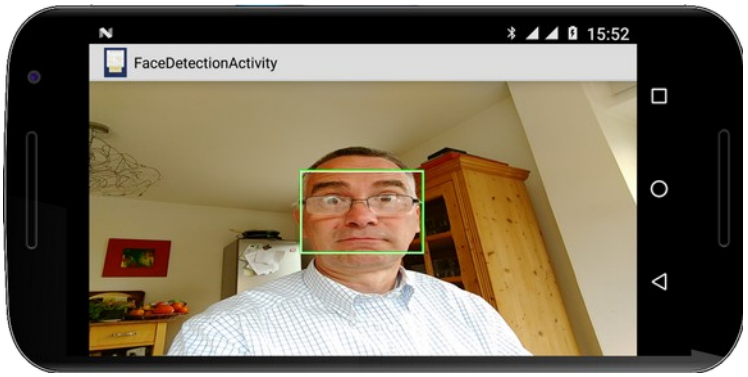
```
private class OverlayView extends View {

    public OverlayView(Context context) {
        super(context);
    }

    @Override
    protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);

        Paint paint = new Paint(Paint.ANTI_ALIAS_FLAG);
        paint.setColor(Color.BLACK);
        paint.setTextSize(64f);
        canvas.drawText("Hello World!",
                        getWidth() / 2 - 180, 200, paint);
    }
}
```

Wenn wir jetzt diese beiden Views übereinander legen, dann haben wir Augmented Reality. Mit einem FrameLayout ist das total trivial:

```
public class AugmentedRealityActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        FrameLayout fl = new FrameLayout(this);
        fl.setLayoutParams(new LayoutParams(LayoutParams.MATCH_PARENT,
                                            LayoutParams.MATCH_PARENT));
        setContentView(fl);

        CameraView cameraView = new CameraView(this);
        fl.addView(cameraView);

        OverlayView overlayView = new OverlayView(this);
        fl.addView(overlayView);
    }
}
```

Tada.

FaceDetection



Als kleines Schmankerl machen wir aus unserer AugmentedReality App eine FaceDetection App. Wir müssen nur ein paar Zeilen ändern.

Wir beginnen mit dem Erkennen von Gesichtern. Das kann Android ganz allein, und zwar über einen *FaceDetectionListener*: den definieren wir in der *onCreate()* Methode unserer Activity:

```
public void onCreate(Bundle
savedInstanceState) {
    ...
    faceDetectionListener = new FaceDetectionListener() {
        @Override
        public void onFaceDetection(Face[] faces, Camera camera) {
            detectedFaces = faces;
            overlayView.invalidate();
        }
    };
}
```

Den FaceDetectionListener müssen wir jetzt noch an die Kamera anschließen, und das machen wir in der *surfaceCreated()* Methode des *CameraView*:

```
public void surfaceCreated(SurfaceHolder holder) {
    try {
        mCamera = Camera.open(1);
        mCamera.setPreviewDisplay(holder);
        mCamera.startPreview();

        mCamera.setFaceDetectionListener(faceDetectionListener);
        mCamera.startFaceDetection();

    } catch (Exception e) {
        Log.e("CameraView", e.getMessage());
    }
}
```

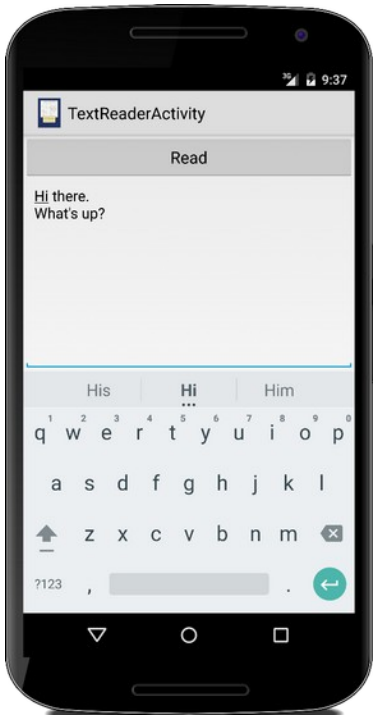
Schließlich zeichnen wir einfach grüne Rechtecke wo die Gesichter detektiert wurden in der *onDraw()* Methode unserer *OverlayView* Klasse:

```
private class OverlayView extends View {
    ...
    protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);

        if (detectedFaces != null) {
            Paint paint = new Paint(Paint.ANTI_ALIAS_FLAG);
            paint.setStyle(Paint.Style.STROKE);
            paint.setStrokeWidth(8);
            paint.setColor(Color.GREEN);

            float scaleX = (float) (getWidth() / 2000.0);
            float scaleY = (float) (getHeight() / 2000.0);

            for (int i = 0; i < detectedFaces.length; i++) {
                Rect rect = detectedFaces[i].rect;

                float left = (1000 - rect.left) * scaleX;
                float top = (1000 + rect.top) * scaleY;
                float right = (1000 - rect.right) * scaleX;
                float bottom = (1000 + rect.bottom) * scaleY;

                canvas.drawRect(left, top, right, bottom, paint);
            }

        }
    }
}
```

Eigentlich erschreckend einfach.

TextReader

Im TextToSpeech Beispiel war der Text der gesprochen wurde im Code vorgegeben. Schöner ist es natürlich, wenn der Nutzer bestimmen kann was gesprochen wird. Dazu modifizieren wir das TextToSpeech Beispiel indem wir einen Button und einen EditText hinzufügen. Sobald auf den Knopf gedrückt wird, lassen wir den TextToSpeech Engine vorlesen was geschrieben wurde:




```
...
final EditText et = (EditText)
findViewById(R.id.edittext);

Button btn = (Button) findViewById(R.id.button);
btn.setOnClickListener(new View.OnClickListener() {

    @Override
    public void onClick(View v) {
        tts.speak(et.getText().toString(),
            TextToSpeech.QUEUE_FLUSH, null, null);
    }
});
...
```

Easy.

AudioBookCreator

Im Beispiel TextReader haben wir gerade gesehen wie einfach es ist die Sprachausgabe von Android zu benutzen. Natürlich kann man sie auch verwenden um ganze Bücher in AudioBooks zu verwandeln.

Im Prinzip könnten wir das ganze Buch einfach als einen String einlesen und dann an den TextToSpeech Engine übergeben. Das Problem damit ist, dass sich das etwas komisch anhört. Der Hauptpunkt der stört: zwischen Absätzen erwartet man eine etwas längere Pause. Die macht der TextToSpeech Engine aber nicht. Deswegen müssen wir da etwas nachhelfen.

Als erstes lesen wir unser Buch in eine Liste, dabei entspricht jeder Absatz einem Eintrag in der Liste:

```
private List<String> readFromResource() {
    List<String> text = new ArrayList<String>();
    StringBuilder total = new StringBuilder();
    try {
        InputStream is = getResources().
            openRawResource(R.raw.tom_sawyer);
        BufferedReader r = new BufferedReader(
            new InputStreamReader(is));

        String line;
        while ((line = r.readLine()) != null) {
            if (line.length() > 0) {
                total.append(line);
            } else {
                String t = total.toString();
                if (t.length() > 0) {
                    text.add(t);
                }
                total = new StringBuilder();
            }
        }
        // OBOB
        String t = total.toString();
        if (t.length() > 0) {
            text.add(t);
        }

    } catch (IOException e) {
        e.printStackTrace();
    }
    return text;
}
```

In der *onCreate()* unserer Activity rufen wir dann diese Methode auf,

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    textList = readFromResource();

    tts = new TextToSpeech(getApplicationContext(),
        new TextToSpeech.OnInitListener() {
```

```
            @Override
            public void onInit(int status) {
                if (status != TextToSpeech.ERROR) {
                    tts.setLanguage(Locale.US);

                    setProgressListener();
                    doSpeak();

                } else {
                    Log.i("AudioBookCreatorActivity",
                          "No speech engine available.");
                }
            }
        });
    }
```

In der *onInit()* Methode fügen wir jetzt aber einen *UtteranceProgressListener* hinzu und rufen dann ein erstes Mal die *doSpeak()* Methode auf. Die *setProgressListener()* Methode sieht wie folgt aus:

```
    private void setProgressListener() {
        tts.setOnUtteranceProgressListener(
            new UtteranceProgressListener() {

            @Override
            public void onDone(String utteranceId) {
                if (textListIndex < textList.size()) {
                    doSpeak();
                }
            }

            @Override
            public void onStart(String utteranceId) {
            }

            @Override
            public void onError(String utteranceId) {
            }

        });
    }
```

Die *onDone()* Methode des UtteranceProgressListener wird dann aufgerufen wenn der Engine mit dem Reden einer Utterance gerade fertig wurde. Deswegen müssen wir die *doSpeak()* Methode einmal vorher aufrufen. Die *doSpeak()* Methode wiederum ist trivial:

```
        private void doSpeak() {
            pause(DELAY);
            final String speech = textList.get(textListIndex);
            tts.speak(speech, TextToSpeech.QUEUE_FLUSH, null, speech);
            textListIndex++;
        }
```

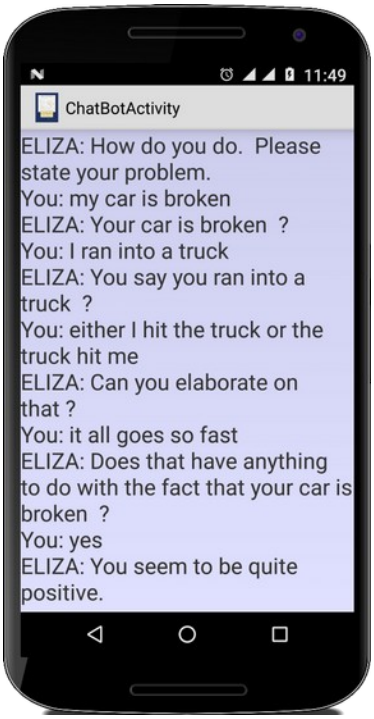
Die gewünschte Pause zwischen den Absätzen im Buch erzeugen wir durch den Aufruf von *pause()* am Anfang der *doSpeak()* Methode.

Jetzt wäre es natürlich noch schön, wenn wir die Bücher nicht nur vorlesen könnten, sondern auch in einer Datei speichern könnten. Da müssen wir einfach anstelle von *tts.speak()* die folgende Zeile schreiben:

```
    tts.synthesizeToFile(speech, null, new File(DESTINATION_FILE +
    textListIndex + ".wav"), speech);
```

Cool, oder?

ChatBot



Im ersten Semester haben wir ja bereits kurz die Bekanntschaft von ELIZA gemacht [16]. Das Programm damals war rein textbasiert. Mit unseren neugefunden Möglichkeiten können wir jetzt aber eine Anwendung schreiben die sprachgebunden ist. Soll heißen, ELIZA spricht mit uns und hört uns zu.

Unsere ChatBotActivity ist eine ganz normale Activity. Nachdem wir ELIZA initialisiert haben,

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

    tv = (TextView)
        this.findViewById(R.id.textView);

    initEliza();

    String me = "Hello";
    String eliz = eliza.processInput(me);
    tv.setText(eliz + "\n");

    initTTS(eliz);
}
```

starten wir den TextToSpeech Engine und lassen ihn den ersten Satz sagen:

```
private void initTTS(final String msg) {
    tts = new TextToSpeech(this, new TextToSpeech.OnInitListener() {
        @Override
        public void onInit(int status) {
            if (status == TextToSpeech.SUCCESS) {
                tts.setLanguage(Locale.US);
                setProgressListener();
                tts.speak(msg, TextToSpeech.QUEUE_FLUSH, null, msg);

            } else {
                Log.i(getApplicationContext(),
                      "No speech engine available.");
                tts.shutdown();
            }
        }
    });
}
```

Wie im AudioBookCreator arbeiten wir auch hier wieder mit einem UtteranceProgressListener. Nachdem der erste Satz gesagt ist, hören wir zu:

```
private void setProgressListener() {
    tts.setOnUtteranceProgressListener(
        new UtteranceProgressListener() {

        @Override
        public void onDone(String utteranceId) {
            startListening();
        }

        @Override
        public void onStart(String utteranceId) {
        }
```

```
        @Override
        public void onError(String utteranceId) {
        }
    });
}
```

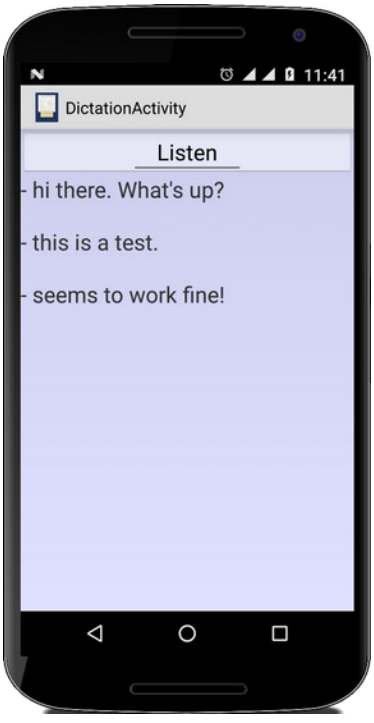
In der *startListening()* Methode starten wir einfach die Spracherkennung wie in unserem TextReader:

```
private void startListening() {
    Intent intent =
        new Intent(RecognizerIntent.ACTION_RECOGNIZE_SPEECH);
    intent.putExtra(RecognizerIntent.EXTRA_LANGUAGE_PREFERENCE, "en");
    intent.putExtra(RecognizerIntent.EXTRA_LANGUAGE_MODEL,
        RecognizerIntent.LANGUAGE_MODEL_FREE_FORM);
    intent.putExtra(RecognizerIntent.EXTRA_MAX_RESULTS, 3);

    startActivityForResult(intent, REQUEST_CODE);
}
```

und in der onActivityResult() Methode,

```
protected void onActivityResult(int requestCode,
                                int resultCode, Intent data) {
    if (requestCode == REQUEST_CODE && resultCode == RESULT_OK) {
        ArrayList<String> matches = data.getStringArrayListExtra(
                                    RecognizerIntent.EXTRA_RESULTS);
        tv.append(me + "\n");
        String eliz = eliza.processInput(me);
        tts.speak(eliz, TextToSpeech.QUEUE_FLUSH, null, eliz);
        tv.append(eliz + "\n");
    }
    super.onActivityResult(requestCode, resultCode, data);
}
```

lassen wir erst ELIZA den Input verarbeiten und anschließend per Sprachsynthese ausgeben. Jetzt haben wir immer jemanden zum Reden wenn uns langweilig ist, WhatsApp ade.



Dictation

Die meisten Diktier-Apps aus dem Google Play Store sind ziemlich langweilig: sie nehmen einfach das Gesagte auf und speichern es als Sound File ab. Mit etwas Spracherkennung, können wir das viel besser.

Unsere App besteht aus einem Button und einem TextView. Wenn wir auf den Button das erste Mal drücken soll die Spracherkennung beginnen, beim zweiten Mal soll sie wieder aufhören. Deswegen bietet sich hier vielleicht ein ToggleButton an. Was allerdings noch cool wäre, wenn man irgendwie diesen komischen Google Dialog während der Spracherkennung wegbekommen könnte. Kann man.

Als erstes muss unsere App ein *RecognitionListener* Interface implementieren, mit all den dazugehörigen Methoden:

```
public class DictationActivity extends Activity
    implements RecognitionListener {

    private TextView tv;
    private SpeechRecognizer recognizer;
    ...
}
```

und wir benötigen einen *SpeechRecognizer* als Instanzvariable.

Dann müssen wir das Ganze in der *onCreate()* initialisieren:

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.dictation_activity);
    tv = (TextView) findViewById(R.id.textView);

    // init speech recognition engine
    if (SpeechRecognizer.isRecognitionAvailable(this)) {
        recognizer = SpeechRecognizer.createSpeechRecognizer(this);
        recognizer.setRecognitionListener(this);

        final Intent recognizerIntent =
            new Intent(RecognizerIntent.ACTION_RECOGNIZE_SPEECH);
        recognizerIntent.putExtra(
            RecognizerIntent.EXTRA_LANGUAGE_PREFERENCE, "en");
        recognizerIntent.putExtra(
            RecognizerIntent.EXTRA_LANGUAGE_MODEL,
            RecognizerIntent.LANGUAGE_MODEL_FREE_FORM);
        recognizerIntent.putExtra(
            RecognizerIntent.EXTRA_MAX_RESULTS, 3);

        ToggleButton btn =
            (ToggleButton) findViewById(R.id.toggleButton);
        btn.setOnCheckedChangeListener(
            new CompoundButton.OnCheckedChangeListener() {
            @Override
            public void onCheckedChanged(CompoundButton buttonView,
                                         boolean isChecked) {
                if (isChecked) {
                    recognizer.startListening(recognizerIntent);
                } else {
                    recognizer.stopListening();
                }
            }
        });
    } else {
        Log.i("Dictation", "No speech recognition engine installed!");
    }
}
```

Wir schauen zuerst nach, ob es überhaupt Spracherkennung gibt. Danach initialisieren wir unseren *recognizer*, und sagen ihm noch wer ihm zuhört. Dann bereiten wir wie üblich einen RecognizerIntent vor. Diesen starten wir allerdings nicht als Intent, sondern über die *startListening()* Methode des *recognizer*. Startet man den *recognizer* auf diese Art und Weise, dann muss man ihn aber auch von Hand wieder anhalten, deswegen die *stopListening()* Methode.

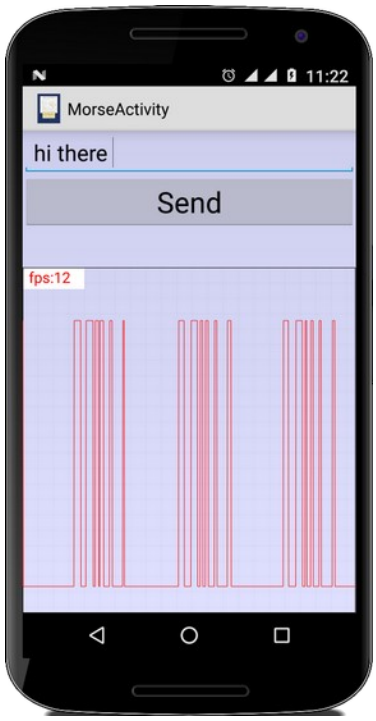
Nachdem wir die *stopListening()* Methode aufgerufen haben, macht die Spracherkennung ihre Arbeit, und wenn sie fertig ist, wird die Methode *onResults()* des *RecognitionListener* aufgerufen:

```
@Override
public void onResults(Bundle results) {
    ArrayList<String> matches = results.getStringArrayList(
                                SpeechRecognizer.RESULTS_RECOGNITION);
    String text = "";
    for (String match : matches) {
        text += match + "\n";
    }
    tv.append("- " + text + "\n");
}
```

Und das war's eigentlich schon.

Challenges

Morse



Jeder hat schon mal etwas von Morsezeichen gehört, insbesondere das "SOS" Signal, drei kurz, drei lang, drei kurz, ist wohl jedem bekannt [17]. Der Morsecode wurde eigentlich für Telegrafen erfunden um über lange Distanzen zu kommunizieren. Er ist nicht besonders effektiv und auch nicht sehr tolerant was seine Fehleranfälligkeit angeht, aber dafür ist er sehr einfach. Wir wollen den Morsecode verwenden, damit zwei Android Geräte miteinander kommunizieren können. Dass das Thema ganz aktuell ist, zeigt z.B. der Artikel [18]: es gibt nämlich sogenannte akustische Cookies die Verwendung finden um personalisierte Werbeanzeigen einzublenden. Die basieren genau auf diesem Prinzip.

Als erstes benötigen wir eine Möglichkeit aus Strings Morsezeichen zu erzeugen und umgekehrt aus Morsezeichen wieder Strings. Dafür verwenden wir die *MorseStateMachine* Klasse, die genau das macht:

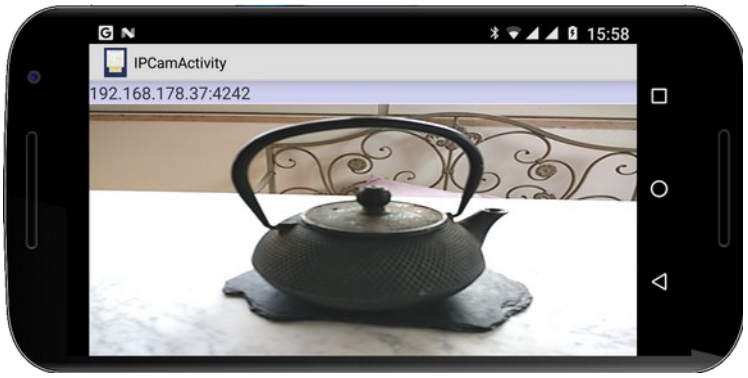
```
// init
MorseStateMachine msm = new MorseStateMachine();

// encode
String msg = "MORSE CODE";
String morse =
msm.convertStringToMorseCode(msg.toUpperCase());
System.out.println(morse);

// decode
msm.addNewString("_______===_===___===_===_===" +
                 "___=_===_=___=_=_=___=_______===_=_===_=___==");
System.out.println(msm.getMorseMessage());
msm.addNewString("=_===_===___===_=_=___=_______");
System.out.println(msm.getMorseMessage());
```

Dabei bedeutet der Unterstrich, '_', kein Signal, und das Gleichheitszeihen, '=', ein Signal. Laut Wikipedia soll ein Dah dreimal so lange sein wie ein Dit [19].

Das weitere Vorgehen ist ganz ähnlich wie bei unserer Sonar Anwendung: der Nutzer gibt in einem EditText einen Text ein, aus dem macht die MorseStateMachine dann '_' und '=', und die senden wir dann wie bei der Sonar Anwendung als kleine Wellenpakete. Auf der empfangenden Seite, wird dann aus dem empfangenen Signal wieder '_' und '=', je nachdem ob ein Ton ankommt oder nicht. Und mit der MorseStateMachine machen wir daraus wieder Text, den wir dann anzeigen.

Interessanterweise haben die uralt akustischen Modems [20] nach genau diesem Prinzip funktioniert. Wobei allerdings andere Kodierungs- und Kompressionsverfahren eingesetzt wurden, und man überhaupt viel Gehirnschmalz aus der Signalverarbeitung braucht, damit das Ganze ordentlich und mit hohen Datenübertragungraten funktioniert.

IPCam



Zum krönenden Abschluss wollen wir aus unserem Smartphone eine IP Kamera machen. Das ist gar nicht so schwer, wir müssen lediglich unsere TakePhoto App mit unserer WebServerActivity App aus dem letzten Kapitel kombinieren.

Bevor wir aber loslegen, müssen wir uns noch kurz die RFC1341 ansehen [21]: dort wird MIME definiert, und speziell im Kapitel 7.2 wird der Multipart Content-Type spezifiziert:

```
Content-Type: multipart/mixed; boundary=gc0p4Jq0M2Yt08jU534c0p
```

Der besagt so viel wie, dass jetzt mehrere Teile kommen (*multipart*) und dass diese Teile durch den String "gc0p4Jq0M2Yt08jU534c0p" begrenzt (*boundary*) werden [22]. Diese Boundary wird verwendet, um die verschiedenen Teile voneinander trennen zu können, d.h. wann immer der String

```
--gc0p4Jq0M2Yt08jU534c0p
```

in den Daten vorkommt, weiß man jetzt kommt der nächste Teil. Wie weiß man, dass man fertig ist? Dazu sendet man einfach

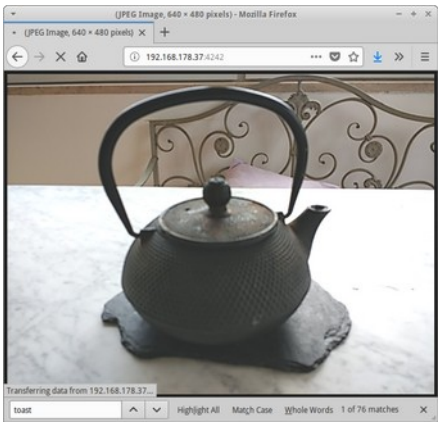
```
--gc0p4Jq0M2Yt08jU534c0p--
```

also, zwei Dashes vor dem Boundary String und zwei danach. Der Boundary String ist beliebig, sollte aber nicht in den eigentlichen Daten vorkommen. Interessanterweise verstehen fast alle Browser den Multipart Content-Type.

Kommen wir zum Code. Wie bereits oben angedeutet, nehmen wir die TakePhoto App und machen ein paar kleine Änderungen. Als erstes implementieren wir das *Runnable* Interface, und am Ende der *onCreate()* starten wir unseren Thread:

```
public class TakePhotoActivity extends Activity implements Runnable {

    public void onCreate(Bundle savedInstanceState) {
        ...
        TextView tv = (TextView) findViewById(R.id.textView);
        tv.setText(Util.getMyLocalIpAddress().getHostAddress()
                  + ":" + PORT);
        ...
        Thread th = new Thread(this);
        th.start();
    }
}
```

Danach müssen wir die run() Methode implementieren. Hier kopieren wir zunächst die run() Methode aus der WebServerActivity:



```
public void run() {
    try {
        ServerSocket server =
            new ServerSocket(PORT);
        while (isRunning) {
            Socket socket = server.accept();
            (new ConnectionThread(
                ++threadNr, socket)).start();
        }
        server.close();
```

```
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

Für unsere IPCam macht es keinen Sinn für jede neue Connection einen neuen Thread zu starten, denn die Connection soll ja offen bleiben. An die Stelle setzen wir daher den folgenden Code:

```
    final OutputStream out = socket.getOutputStream();

    // send first part of header
    String httpHeader1 = "HTTP/1.0 200 OK\r\n"
            + "Content-type: multipart/x-mixed-replace; "
            + "boundary=--BoundaryString\r\n\r\n";
    out.write(httpHeader1.getBytes("ASCII"));

    // send image stream
    while (isRunning && socket.isConnected()) {
        camera.takePicture(null, null, new PictureCallback() {

            @Override
            public void onPictureTaken(byte[] data, Camera camera) {
                try {
                    // send second part of header
                    String httpHeader2 = "--BoundaryString\r\n"
                        + "Content-type: image/jpeg\r\n"
                        + "Content-length: " + data.length + "\r\n\r\n";
                    out.write(httpHeader2.getBytes("ASCII"));
                    out.write(data);
                    out.flush();

                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        });
        pause(DELAY);
    }

    out.close();
    socket.close();
    socket = null;
```

Und das war's auch schon. Wenn wir jetzt mit einem Browser in unserem lokalen Netzwerk auf die angezeigt IP Adresse gehen, dann bekommen wir den Livefeed der Smartphone Kamera angezeigt. Die Pause im Code ist notwendig, weil wir unser Smartphone damit doch an seine Grenzen bringen. Bei meinem Handy funktioniert das noch mit einem DELAY von 300 Millisekunden, mehr geht aber nicht.

Camera2Preview

Eine Sache noch: bei unseren bisherigen Camera Programmen gab es immer diese komische Warnung: "The type *Camera* is deprecated", und man wurde ermahnt man solle doch die *Camera2* Klasse verwenden. In diesem Projekt haben wir das CameraPreview Projekt mit der Camera2 Klasse implementiert. Die Camera2 Klasse kann zwar viel mehr als die normale Camera Klasse, allerdings bläht sich der Code derart auf, dass es für ein Buch zur Einführung in die Android Progammierung nicht mehr geeignet ist.

Research

Auch in diesem Kapitel kann man sich noch weiterbilden wenn man möchte.

Endian

Arbeitet man auf dem Byte Level, dann passieren häufig komische Fehler, die auf der unterschiedlichen Endianness [5] von CPUs und Betriebssystemen zu tun haben. Prinzipiell gibt es Little-Endian und Big-Endian. Wir sollten den Unterschied kennen.

Fourier

Häufig wird Fourier-Transformation in der Mathe3 Vorlesung geskippt. Das ist eigentlich sehr schade. Natürlich muss man nicht in kleinsten Detail wissen wie sie funktioniert, aber grob verstehen sollte man sie schon und vielleicht ein bisschen was in Referenz [12] dazu nachlesen. Nicht nur im Audiobereich, auch die Videokompression, z.B. Jpeg, verwendet häufig eine Variante der Fourier-Transformation.

Nyquist–Shannon

Das Nyquist-Shannon-Abtasttheorem [6] ist der Grund warum CDs mit einer Sampling Rate von 44 kHz aufgenommen werden. Es ist Zeit mehr darüber zu erfahren.

Cross-Correlation

Wie kommt man in einem verrauschten Signal an die "guten" Daten? Es gibt viele unterschiedliche Möglichkeiten, aber eine mit der man anfangen sollte ist die Cross-Correlation [7], die wir für die Sonar App verwendet haben.

Fragen

1. Nennen Sie drei Medientypen die man mit dem Android MediaPlayer abspielen kann.

2. Ist es besser den MediaPlayer in der *onCreate()* oder der *onResume()* zu erzeugen? Warum?

3. Wenn Sie den MediaPlayer nicht mehr benötigen, sollten Sie die folgenden Schritte befolgen:

```
if ( player != null ) {
    player.stop();
    player.release();
    player = null;
}
```

   Warum ist das so?

Referenzen

[1] Recording .Wav with Android AudioRecorder, https://stackoverflow.com/questions/17192256/recording-wav-with-android-audiorecorder

[2] Capturing videos, https://developer.android.com/guide/topics/media/camera#capture-video

[3] Gleichstufige Stimmung, https://de.wikipedia.org/wiki/Gleichstufige_Stimmung

[4] Frequenzen der gleichstufigen Stimmung, https://de.wikipedia.org/wiki/Frequenzen_der_gleichstufigen_Stimmung

[5] Endianness, https://en.wikipedia.org/wiki/Endianness

[6] Nyquist–Shannon sampling theorem, https://en.wikipedia.org/wiki/Nyquist–Shannon_sampling_theorem

[7] Cross-correlation, https://en.wikipedia.org/wiki/Cross-correlation

[8] Overtone, https://en.wikipedia.org/wiki/Overtone

[9] The Secrets Behind Freddie Mercury's Legendary Voice, https://www.youtube.com/watch?v=p3MjsrMNCbU

[10] Polyphonic Overtone Singing, https://www.youtube.com/watch?v=vC9Qh709gas

[11] Tuvan throat singing, https://en.wikipedia.org/wiki/Tuvan_throat_singing

[12] Fast Fourier transform, https://en.wikipedia.org/wiki/Fast_Fourier_transform

[13] MEAPsoft, Columbia University, http://www.ee.columbia.edu/~ronw/code/MEAPsoft/doc/html/FFT_8java-source.html

[14] Spectrogram, https://en.wikipedia.org/wiki/Spectrogram

[15] SoundWave, https://www.microsoft.com/en-us/research/project/soundwave-using-the-doppler-effect-to-sense-gestures/

[16] ELIZA, https://en.wikipedia.org/wiki/ELIZA

[17] Morse code, https://en.wikipedia.org/wiki/Morse_code

[18] SoniControl: App soll vor akustischen Cookies schützen, https://www.heise.de/newsticker/meldung/SoniControl-App-soll-vor-akustischen-Cookies-schuetzen-4059259.html

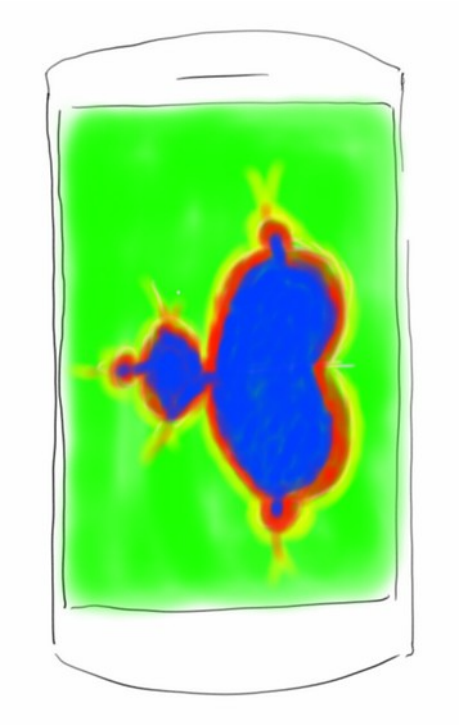
[19] Morsezeichen, https://de.wikipedia.org/wiki/Morsezeichen#Zeitschema_und_Darstellung

[20] Acoustic coupler, https://en.wikipedia.org/wiki/Acoustic_coupler

[21] MIME (Multipurpose Internet Mail Extensions), https://tools.ietf.org/html/rfc1341

[22] The Multipart Content-Type, https://www.w3.org/Protocols/rfc1341/7_2_Multipart.html

Special Topic:
Graphics Performance



Eine Frage, die sich oft bei der Grafikprogrammierung stellt: geht's schneller? Obwohl es keine allgemein gültige Antwort gibt, wollen wir hier einige der zugrunde liegenden Regeln kennen lernen. Dazu sehen wir uns vier verschiedenen Methoden an, wie man 50.000 GRects zeichnen kann.

In allen vier Fällen verwenden wir die folgende Aktivität,


```
public class FastGRectActivity extends Activity {
    private final int SIZE = 40;
    private final int DELAY = 40;

    @Override
    public void onCreate(Bundle savedState) {
        super.onCreate(savedState);

        View gv = new FastGRectView(this);
        setContentView(gv);
    }
}
```

Lediglich in den Views unterscheiden sie sich. Alle vier Views haben aber das gleiche Gerüst:

```
class FastGRectView extends View {
    private Paint paint;
    private Random rgen = new Random();

    public FastGRectView(Context context) {
        super(context);

        paint = new Paint(Paint.ANTI_ALIAS_FLAG);
        paint.setStyle(Paint.Style.FILL);
        paint.setStrokeWidth(1);
        paint.setTextSize(48f);
    }

    @Override
    protected void onDraw(Canvas canvas) {
        long startTime = System.currentTimeMillis();

        for (int k = 0; k < 50000; k++) {
            ...
        }

        long time = System.currentTimeMillis() - startTime;
        paint.setColor(Color.WHITE);
        canvas.drawRect(0, 0, 250, 70, paint);
        paint.setColor(Color.BLACK);
        canvas.drawText("time:" + time, 20, 50, paint);
    }
}
```

Der Konstruktor initialisiert alle Instanzvariablen und das Zeichnen erfolgt in der *onDraw()* Methode. Um die Zeit zu messen, verwenden wir die Methode *System.currentTimeMillis()*. Generell sollte so wenig wie möglich in der *onDraw()* Methode passieren, denn das kostet ja Zeit. Deswegen haben wir das *Paint* Objekt bereits im Konstuktor erzeugt.

Reference Case: Simple Canvas Drawing Methods

Wir beginnen mit der einfachsten Art und Weise Rechtecke zu zeichnen, mit der Methode *canvas.drawRect()*:

```
for (int k = 0; k < 50000; k++) {
    int w = rgen.nextInt(SIZE);
    int x = rgen.nextInt(getWidth());
    int y = rgen.nextInt(getHeight());
```

```
            paint.setColor(rgen.nextInt());
            canvas.drawRect(new RectF(x, y, x + w, y + w), paint);
        }
```

Das ist unser Referenzpunkt. Auf meinem Motorola G4 benötigt dieser Code ca. 1570ms.

Simple Canvas Drawing with a GRect

Als erste Modifikation führen wir eine *GRect* Klasse ein.

```
        private class GRect {
            protected int x, y, w, h;
            private Paint paint;

            public GRect(int x, int y, int w, int h) {
                this.x = x;
                this.y = y;
                this.w = w;
                this.h = h;
                this.paint = new Paint(Paint.ANTI_ALIAS_FLAG);
                paint.setColor(Color.BLUE);
                paint.setStrokeWidth(1);
                paint.setStyle(Paint.Style.FILL);
            }

            public void setColor(int color) {
                paint.setColor(color);
            }

            public void draw(Canvas canvas) {
                canvas.drawRect(new RectF(this.x, this.y, this.x + this.w,
                    this.y + this.h), paint);
            }
        }
```

Diese Klasse verwendet Instanzvariablen für den Zustand und sie hat auch eine *draw()* Methode, die für das Zeichnen verantwortlich ist.

In der View Klasse sieht die *onDraw()* Methode jetzt etwas komplizierter aus, aber nur, weil wir die Erzeugung der GRect-Objekte von dem eigentlichen Zeichnen trennen:

```
        protected void onDraw(Canvas canvas) {
            // seperate creation (this part takes about 1250ms)
            GRect[] rects = new GRect[50000];
            for (int k = 0; k < 50000; k++) {
                // create randomly sized rect
                int w = rgen.nextInt(SIZE);
                int x = rgen.nextInt(getWidth());
                int y = rgen.nextInt(getHeight());

                rects[k] = new GRect(x, y, w, w);
                rects[k].setColor(rgen.nextInt());
            }

            // from drawing (this part also takes about 1250ms)
            long startTime = System.currentTimeMillis();
            for (int k = 0; k < 50000; k++) {
                rects[k].draw(canvas);
            }

            ...
        }
```

Der Grund für diese Trennung ist, dass unter normalen Umständen die meisten Objekte nur einmal erstellen würden und dann nur gelegentlich ein paar hinzugefügt oder entfernt würden. Das hat aber nichts mit der Grafikleistung zu tun.

Naiv würde man erwarten, dass dieser Ansatz deutlich langsamer sein müsste als unser erster Ansatz. Erstens, weil wir Objekte verwenden und die Erstellung von Objekten teuer ist. Zweitens, weil der Code komplizierter aussieht, was in der Regel mit Langsamkeit übersetzt wird. Aber das Gegenteil ist der Fall: Dieser Code benötigt nur 1250ms, d.h. er ist ca. 20% schneller. Wir lernen also, dass Objektorientierung, wenn richtig angewendet, zu einer Leistungssteigerung führen kann. (Der Grund liegt wahrscheinlich am Prozessor-Cache.)

Canvas Drawing using a Bitmap

Auf fast allen Rechnern sind die einfachen Array Operationen deutlich optimiert. Für die Grafikprogrammierung bedeutet das, dass man bei dem Arbeiten mit Bitmaps idealerweise mit dem zugrunde liegenden Pixel-Array arbeitet. Um das auszuprobieren schreiben wir unseren View etwas um:

```
class FastGRectView extends View {
    ...
    private int mCanvasWidth;
    private int mCanvasHeight;
    private Bitmap bitmap;
    private int[] bitMapArray;

    ...

    protected void onSizeChanged(int w, int h, int oldw, int oldh) {
        if (bitmap != null) {
            bitmap.recycle();
        }
        bitmap = Bitmap.createBitmap(w, h, Bitmap.Config.ARGB_8888);
        mCanvasWidth = w;
        mCanvasHeight = h;
        bitMapArray = new int[mCanvasWidth * mCanvasHeight];
    }

    ...
}
```

Wir deklarieren eine Bitmap und ihr Pixel-Array als Instanzvariablen. Zusätzlich benötigen wir die *onSizeChanged()* Methode, in der wir die Bitmap und das Array initialisieren.

Die *onDraw()* Methode erfährt folgende Änderungen:

```
for (int k = 0; k < 50000; k++) {
    int w = rgen.nextInt(SIZE);
    int x = rgen.nextInt(getWidth() - SIZE);
    int y = rgen.nextInt(getHeight() - SIZE);

    drawRect(x, y, w, w, rgen.nextInt());
}
bitmap.copyPixelsFromBuffer(IntBuffer.wrap(bitMapArray));

canvas.drawBitmap(bitmap, 0, 0, null);
```

In der drawRect() Methode zeichnen wir in das Pixel-Array. Die Methode *bitmap.copyPixelsFromBuffer()* erlaubt es uns, aus dem Array eine Bitmap zu machen und dann mit der Methode *drawBitmap()* des Canvas die Bitmap auf dem Bildschirm zu zeichnen.

Die *drawRect()* Methode ist ein wenig primitiv, da es sich um eine Manipulation auf Byte-Ebene handelt:

```
private void drawRect(int x, int y, int w, int h, int color) {
    int len = bitMapArray.length;
    for (int i = 0; i < w; i++) {
        for (int j = 0; j < h; j++) {
            int idx = (y + j) * mCanvasWidth + (x + i);
            if (idx < len) {
                bitMapArray[(y + j) * mCanvasWidth + (x + i)] = color;
            }
        }
    }
}
```

Und, wie lange dauert es? Nun, es dauert 740ms. Das ist doppelt so schnell wie unser ursprünglicher Ansatz.

Canvas Drawing using a Bitmap and a GRect

Können wir es noch besser machen? Betrachtet man unser vorheriges Beispiel, gibt es Hoffnung. Auch hier lagern wir unsere Datenhaltung und das Zeichnen in eine GRect Klasse aus:

```
private class GRect {
    protected int x, y, w, h;
    private int color;

    public GRect(int x, int y, int w, int h) {
        this.x = x;
        this.y = y;
        this.w = w;
        this.h = h;
        this.color = Color.BLUE;
    }

    public void setColor(int color) {
        this.color = color;
    }

    public void draw(int[] bitMapArray) {
        int len = bitMapArray.length;
        for (int i = 0; i < w; i++) {
            for (int j = 0; j < h; j++) {
                int idx = (y + j) * mCanvasWidth + (x + i);
                if (idx < len) {
                    bitMapArray[(y + j) * mCanvasWidth + (x + i)] =
                        color;
                }
            }
        }
    }
}
```

Wir stellen fest, dass sie fast identisch zu der vorherigen GRect-Klasse ist.

Das gleiche gilt für die *onDraw()* Methode:

```
...
// from drawing (this part also takes about 1250ms)
long startTime = System.currentTimeMillis();
for (int k = 0; k < 50000; k++) {
    rects[k].draw(bitMapArray);
}
bitmap.copyPixelsFromBuffer(IntBuffer.wrap(bitMapArray));
```

```
// draw bitmap on canvas
canvas.drawBitmap(bitmap, 0, 0, null);
...
```

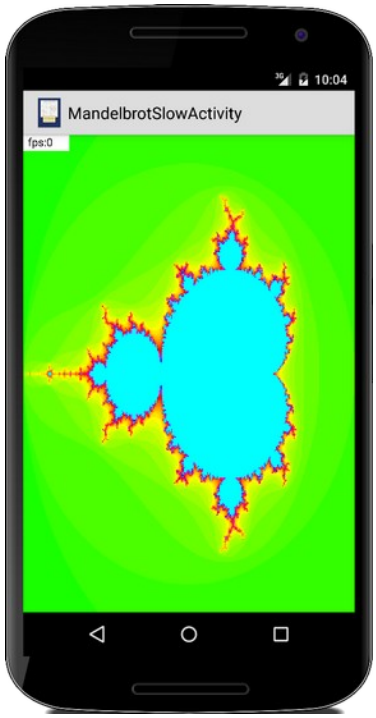
Der einzige Unterschied ist, dass wir das Pixel-Array an die *draw()* Methode der GRects übergeben und dass wir die *drawBitmap()* Methode des Canvas Objekts am Ende aufrufen.

Ist es wirklich schneller? 490ms ist die Zeit, die es dauerte. Das ist dreimal schneller als unser ursprünglicher Ansatz! Wenn wir z.B. 25 Bilder pro Sekunde in unserem Spiel haben wollen, bedeutet das, dass wir ungefähr 2000 GRect Objekte pro Frame zeichnen können.

Lassen Sie sich also nicht von Leuten täuschen, die sagen, dass Objekte oder die Objektorientierung langsam ist. Ist sie komplizierter? Nun, am Anfang sieht es so aus. Aber auch das hängt von der Verkapselung ab. Im nächsten Kapitel sehen wir, wie man all dies in eine Bibliothek umwandelt, die der ACM-Grafikbibliothek ähnelt. Versuchen Sie danach, ein Spiel wie BrickBreaker mit den beiden verschiedenen Ansätzen zu programmieren und treffen Sie dann ein Urteil.

Noch ein Hinweis: In unserer Version der ACM-Grafikbibliothek haben wir den Bitmap-Ansatz nicht verwendet. Sie fragen sich vielleicht, warum? Nun, erstens, wenn wir deutlich weniger als 2000 Objekte zeichnen, vielleicht ein paar hundert, dann ist das Speichern nicht ganz so wichtig, weil die Methode bitmap.copyPixelsFromBuffer() etwa 10ms dauert. Zusätzlich bietet die Klasse Canvas viele praktische Methoden wie drawOval(), drawLines() und drawText(), um nur einige zu nennen. Die Implementierung all dieser Elemente in unserem Bitmap-Ansatz nimmt viel Entwicklungszeit in Anspruch. Aber wenn unsere Programme mit vielen tausend Objekten arbeiten müssten, dann würden wir mit Sicherheit auf den Bitmap-Ansatz zurückkommen.

Mandelbrot



Mandelbrot Fraktale haben wir schon im zweiten Buch gesehen. Dabei wird einfach die mathematische Gleichung

```
z_n+1 = z_n * z_n + c
```

grafisch dargestellt. Den Code können wir fast eins-zu-eins aus dem zweiten Semester übernehmen. Allerdings sollten wir vieleicht erst einmal in Vierer-Schritten durch die Iteration gehen, es stellt sich nämlich heraus, dass es gefühlt ewig dauert bis unser Bild fertig ist.

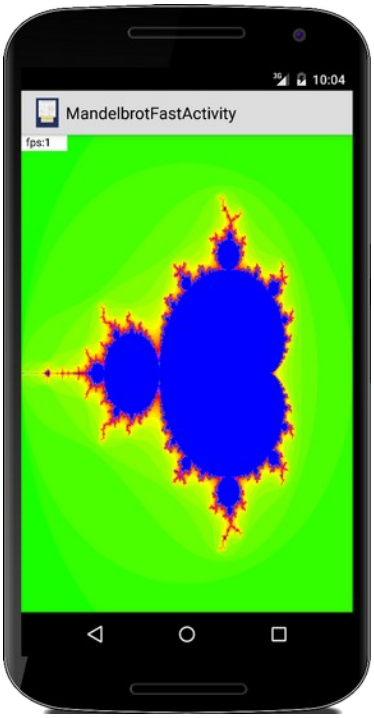
Ähnlich wie beim GameOfLife Beispiel kann man das Ganze fast um den Faktor zehn beschleunigen, wenn man mit einem Array und einer Bitmap arbeitet, anstelle der *drawPoint()* oder *drawRect()* Methode der Canvas Klasse. Dazu definieren wir eine *bitmap* und ein *bitMapArray* als Instanzvariablen:

```
private Bitmap bitmap;
private int[] bitMapArray;
```

In der *onSizeChanged()* Methode initialisieren wir die beiden

```
protected void onSizeChanged(int w, int h, int
oldw, int oldh) {
    if (bitmap != null) {
        bitmap.recycle();
    }
    bitmap = Bitmap.createBitmap(w, h, Bitmap.Config.ARGB_8888);
    mCanvasWidth = w;
    mCanvasHeight = h;
    bitMapArray = new int[mCanvasWidth * mCanvasHeight];
}
```

Und in der *onDraw()* Methode berechnen wir das Array, machen daraus dann eine Bitmap und zeichnen die Bitmap auf den Canvas:

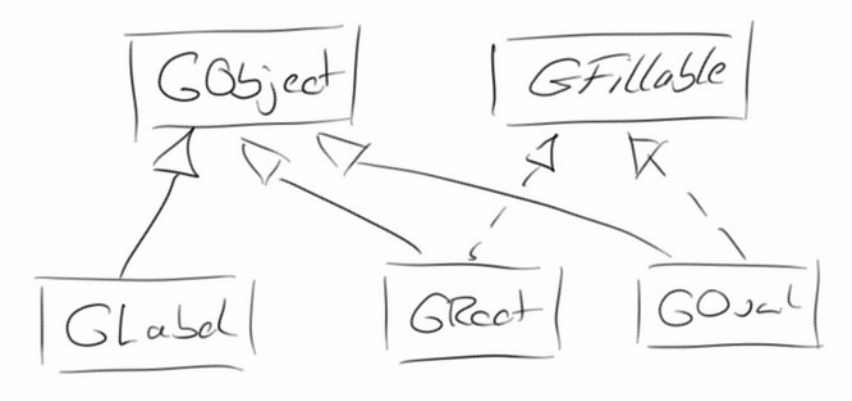



```
public void onDraw(Canvas canvas) {

    double xStep = (xMax - xMin) /mCanvasWidth *1;
    double yStep = (yMax - yMin) /mCanvasHeight *1;
    for (double x = xMin; x < xMax; x += xStep) {
        int i = (int) (((x - xMin) * mCanvasWidth)
            / (xMax - xMin));
        for (double y = yMin; y <yMax; y +=yStep) {
            int j = (int) (((y - yMin)
                * mCanvasHeight) / (yMax - yMin));
            bitMapArray[j * mCanvasWidth + i] =
                function(x, y);
        }
    }
    bitmap.setPixels(bitMapArray, 0, mCanvasWidth,
        0, 0, mCanvasWidth, mCanvasHeight);

    canvas.drawBitmap(bitmap, new Rect(0, 0,
        bitmap.getWidth(), bitmap.getHeight()),
        new Rect(0, 0, bitmap.getWidth(),
            bitmap.getHeight()), null);
}
```

Das geht jetzt schon viel schneller, ca. ein Bild pro Sekunde.

Geht es noch schneller? Die meisten Android CPUs haben inzwischen mehr als einen Core, Quadcores sind heute schon fast die Regel. Bisher haben wir aber in all unseren Programmen immer nur einen dieser Cores verwendet. Wenn wir es schaffen die anderen auch arbeiten zu lassen, dann können wir unser Programm noch mal um einiges schneller machen, je nachdem wie viele Cores unser Gerät hat. Wie das geht sehen wir im Kapitel zu Concurrency.

Special Topic: Libraries



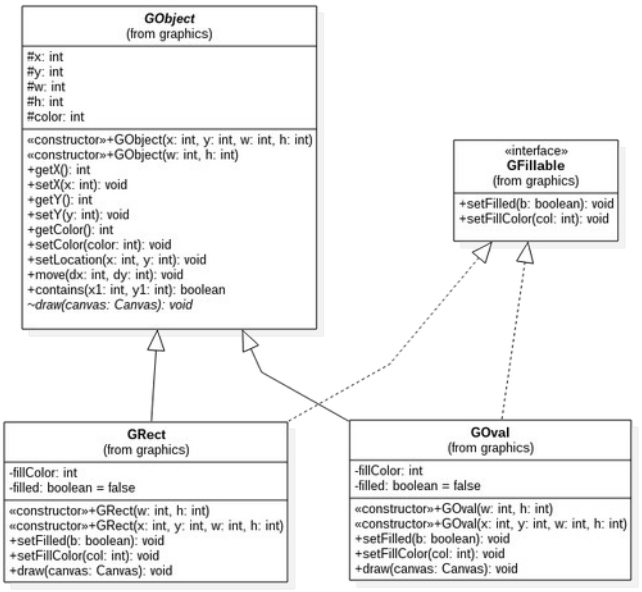
Wiederverwendung ist das Thema in diesem Kapitel. Wir haben gesehen, dass wir unsere alten ACM Grafikprogramme mit ein paar Modifikationen auf Android portieren können. Die Frage ist, können wir das auch ohne Änderungen? Die Idee ist also, eine *acm.jar* Datei für Android zu erstellen.

Requirements for GRect and GOval

Sehen wir uns zunächst die Anforderungen für GRect, GOval und auch GObject an. Wir werfen dazu einen Blick auf BrickBreaker und fragen uns: welche Methoden benötigen wir, damit wir dieses Spiel implementieren können? Geht man Zeile für Zeile den Code von BrickBreaker durch, kommt folgende Liste heraus:

- getX() und getY()
- getWidth() und getHeight()
- move(dx, dy) und setLocation(x, y)
- setColor(col)
- setFilled(b) und setFillColor(col)
- und zwei Constructoren der Form GRect(w,h) und GRect(x,y,w,h)

Das ist also unsere Minimalanforderung. Aus der lässt sich das Klassendiagramm rechts ableiten.



GObject

Beginnen wir mit der GObject Klasse. Da außer der draw() Methode praktisch alle Methoden zwischen GRect und GOval identisch sind, macht es Sinn die in einer Elternklasse, GObject, zu implementieren. Die *draw()* Methode allerdings muss unterschiedlich sein, deswegen deklarieren wir sie als *abstract*:

```
abstract void draw(Canvas canvas);
```

Dies bedeutet, dass alle Klassen, die von GObject erben (wie GRect und GOval), diese Methode implementieren müssen. Es bedeutet aber auch das die Klasse GObject als *abstract* deklariert werden muss. In den jeweiligen Kinderklassen, also GRect und GOval, müssen wir dann diese Methode überschreiben.

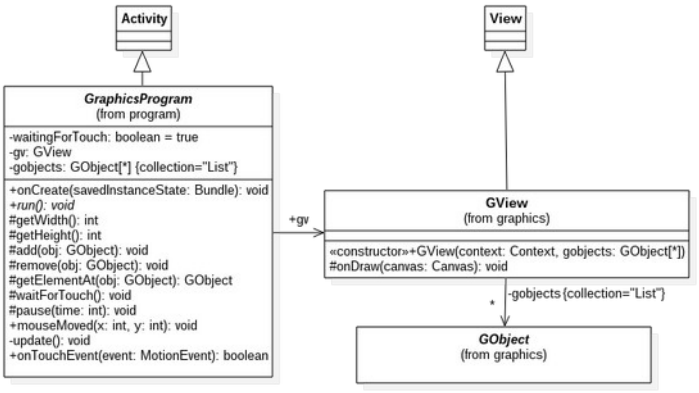
GFillable

Wir hätten die Methoden *setFilled()* und *setFillColor()* auch in der GObject Klasse implementieren können. Allerdings gibt es Klassen, wie z.B. GLabel und GLine, für die diese Methoden keinen Sinn machen. Deswegen lagern wir diese in ein Interface aus, das GFillable Interface. Der Nachteil von Interfaces in Java ist, dass sie keinen Code beinhalten dürfen. Deswegen muss die eigentliche Implementierung dieser Methoden in den Klasse GRect und GOval passieren (Duplication of Code). Das ist der Preis den man dafür zahlen muss, dass es in Java keine Mehrfachvererbung gibt.

GraphicsProgram

Der nächste Schritt ist das GraphicsProgram von ACM. Wiederum werfen wir einen Blick auf BrickBreaker und sehen, dass wir mindestens die folgenden Methoden benötigen:

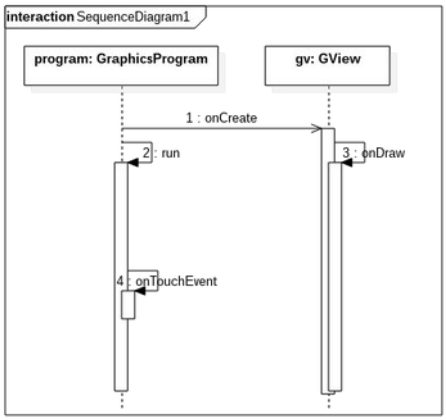
- run()
- waitForClick()
- pause(t)
- getElementAt(x, y)
- setSize(w, h)
- add(o)
- mouseMoved()
- und addMouseListeners().



Dies bestimmt im Wesentlichen, wie unsere GraphicsProgram Klasse aussehen muss.

Es gibt eine wichtige Sache, die wir beachten müssen: In Android wird das Zeichen in der onDraw() Methode der View Klasse durchgeführt. Touch Events hingegen werden an die onTouchEvent() Methode der Activity Klasse delegiert. Das bedeutet, dass das Zeichnen der GRects und GOvals in der GView-Klasse erfolgen muss, während die Bewegung der GObjecte aufgrund von GameLoop und/oder Touch Events, in der GraphicsProgram-Klasse erfolgen muss. Da beide Klassen also auf die GObjekte zugreifen müssen, müssen wir sie zwischen beiden Klassen teilen (shared Resource).

Multi-Threading



Bei Animationen und Spielen mit der ACM-Bibliothek waren wir uns dessen nicht bewusst, aber eigentlich geschahen zwei Dinge gleichzeitig: Einerseits war es der Game Loop der ständig gelaufen ist und unabhängig davon wurden die GObjects gezeichnet. Wir müssen dies nun von Hand implementieren, und der einzige Weg, dies zu tun, ist mit Threads zu arbeiten: Ein Thread macht den Game Loop (in GraphicsProgram) und der andere das Zeichnen (in GView).

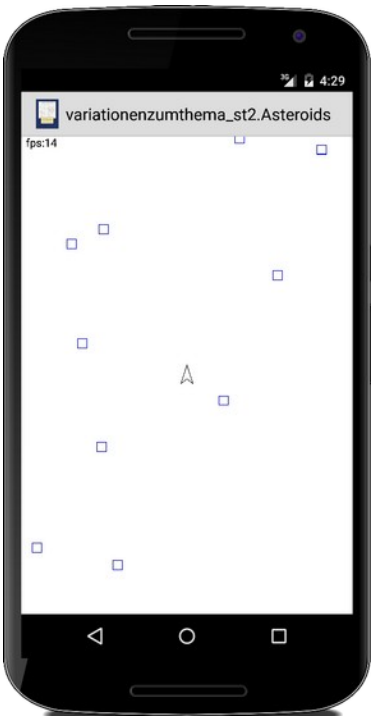
Was passiert, ist am einfachsten mit einem Sequenzdiagramm zu sehen:

Solange zwei Threads keine gemeinsame Resource teilen, kann nicht viel schief gehen. Aber in unserem Fall teilen sich die beiden Threads eine gemeinsame Ressource, die Liste der GObjects. Diese Liste muss mit Vorsicht behandelt werden, und das ist es, was die folgende Zeile tut:

```
private List<GObject> gobjects =
    Collections.synchronizedList(new ArrayList<GObject>());
```

Es besagt im Grunde, dass jeder Zugriff auf diese Liste synchronisiert werden muss.

Library - Reuse



Wir können jetzt alle unsere Grafikprogramme aus dem ersten und zweiten Semester auf Android portieren. Dabei haben wir zwei Möglichkeiten.

Entweder wir kopieren den Quellcode für die Grafikklassen zu unseren Grafikprogrammen und kompilieren beides zusammen.

Oder wir erstellen aus den Grafikklassen eine jar-Bibliothek, ähnlich der *acm.jar* Bibliothek. Diese sogenannte "*acm_graphics.jar*" müssten wir dann anstelle der *acm.jar* Bibiliothek verwenden, und schon laufen unsere alten Programme auf Android.

Um mit Eclipse eine jar-Bibiliothek zu erstellen, beginnt man mit einem Standard-Java-Projekt. Dorthin kopiert man die Java Quelldateien von acm_graphics. Nun müssen wir einen Verweis auf die Datei android.jar (zu finden z.B. in /Android/Sdk/platforms/android-21/) im Java Build Path hinzufügen. Dann müssten unsere Klassen eigentlich kompilieren. Zum Generieren der jar Datei, klickt man im Package Explorer mit der rechten Maustaste und wählt Export, dann Java->JAR-Datei und klickt auf Weiter. Auf dem nächsten Bildschirm wählt man nur die Dateien aus, die man auch in der jar Datei haben möchten. Als nächstes wählt man das Exportziel, etwa AppA_AcmLibrary/lib/acm_android.jar. Meist gibt es jetzt noch ein paar nichtssagende Fehlermeldungen, die meist nicht viel zu bedeuten haben. Man schaut einfach in die jar-Datei (es ist nur eine ZIP-Datei) und stellt sicher, dass sie das enthält, was sie soll. Arbeitet man mit Android Studio, dann ist das Erstellen von jar-Bibliotheken etwas komplizierter, aber nicht viel [1].

Referenzen

[1] How to make a .jar out from an Android Studio project, stackoverflow.com/questions/21712714/how-to-make-a-jar-out-from-an-android-studio-project

Special Topic: Services

Bisher haben wir nur Activitys kennen gelernt. Für gefühlt 99% aller Anwendungen genügen diese auch. Es gibt aber eine handvoll Anwendungen, bei denen es praktisch wäre, wenn sie im Hintergrund weiterlaufen würden. Zwei Anwendungen mit denen wir schon zu tun hatten sind der MusicPlayer und der StepCounter. Bei beiden erwarten wir eigentlich, dass sie im Hintergrund weiterlaufen, auch wenn wir eine andere Activity im Vordergrund haben. Genau für diese Fälle gibt es Services. Außerdem lernen wir noch die BroadcastReceiver kennen. Diese werden in der Regel von Systemevents getriggert, und haben ähnlich wie Services kein Userinterface.

SimpleService

Der einfachste Service muss lediglich die Klasse *Service* erweitern und die Methode *onBind()* implementieren:

```
public class SimpleService extends Service {

    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }
}
```

Natürlich ist er so total nutzlos. Wie bei Activities haben auch Services die *onCreate()* und *onDestroy()* Methoden,

```
    ...
    public void onCreate() {
        Log.i("SimpleService", "onCreate()");
        super.onCreate();
    }

    public void onDestroy() {
        Log.i("SimpleService", "onDestroy()");
        super.onDestroy();
    }
    ...
```

die wir zum Initialisieren und Aufräumen überschreiben können. Wirklich interessant ist die *onStartCommand()* Methode, denn hier wird der Service gestartet:

```
    ...
    public int onStartCommand(Intent intent, int flags, int startId) {
        Log.i("SimpleService", "onStartCommand()");

        // if we get killed, restart
        // return START_STICKY;

        // Don't automatically restart this Service if it is killed
        return START_NOT_STICKY;
    }
    ...
```

Alles was den Service ausmacht, passiert eigentlich hier (ein bisschen ähnlich wie bei der run() Methode von Threads). Interessant ist der Rückgabeparameter dieser Methode. Wenn wir den Wert *START_NOT_STICKY* zurückgeben, dann ist der Service damit beendet. Geben wir aber den Wert *START_STICKY* zurück, dann wird Android den Service immer wieder starten (meistens wenigstens), selbst wenn es ihn mal aus Platzmangel abgeschossen hat.

Im AndroidManifest müssen wir den Service noch bekannt machen, das geht über die folgenden Zeilen:

```
...
<service
    android:name="variationenzumthema_st3.SimpleService"
    android:exported="false" />
...
```

Jetzt bleibt noch die Frage, wie starten wir denn unseren Service? Eine Möglichkeit ist, das über eine Activity zu tun. Dort definieren wir einen Intent:

```
public class SimpleServiceActivity extends Activity {

    protected void onCreate(Bundle savedInstanceState) {
        ...
        final Intent simpleServiceIntent =
            new Intent(getApplicationContext(), SimpleService.class);
        ...
        startService(simpleServiceIntent);
        ...
        stopService(simpleServiceIntent);
        ...
    }
    ...
}
```

über den wir dann mittels *startService()* den Service starten und mittels *stopService()* beenden.

Wie wissen wir ob ein Service schon läuft? Das geht entweder auf dem Gerät über die Einstellungen -> Developer options -> Running services, oder mit der folgenden Methode:

```
private boolean isServiceRunning(Class<?> serviceClass) {
    ActivityManager manager =
        (ActivityManager) getSystemService(Context.ACTIVITY_SERVICE);

    for (RunningServiceInfo service :
         manager.getRunningServices(Integer.MAX_VALUE)) {
        if (serviceClass.getName().equals(
                                    service.service.getClassName())) {
            return true;
        }
    }
    return false;
}
```

MusicService

Als erste nützliche Anwendung für einen Service wollen wir einen MusicService implementieren, der die Musik weiterspielt, auch wenn unsere MusicServiceActivity nicht mehr im Vordergrund ist.

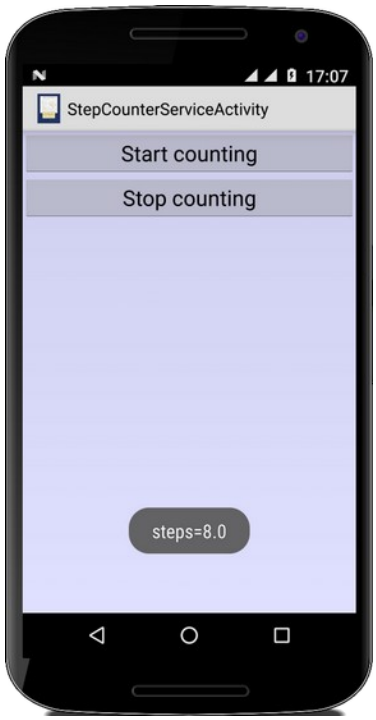
Dazu definieren wir die Instanzvariable *player* vom Typ MediaPlayer und implementieren die *onStartCommand()* und *onDestroy()* Methoden:

```
public int onStartCommand(Intent intent, int flags,
                                        int startId) {
    if (player == null) {
        player = MediaPlayer.create(this,
                                    R.raw.trap_beat);
        player.start();

    } else {
        if (player.isPlaying()) {
            player.seekTo(0);
        } else {
            player.start();
        }
    }
    return START_NOT_STICKY;
}
```

```
public void onDestroy() {
    if (player != null) {
        player.stop();
        player.release();
    }
    super.onDestroy();
}
```

Gestartet wird der Service über die *MusicServiceActivity*, die identisch zur SimpleServiceActivity ist.

StepCounterService



Damit der StepCounter-Sensor Schritte zählt, muss immer ein Listener definiert sein. Wenn wir dafür eine Activity verwenden, dann werden nur so lange Schritte gezählt wie diese Activity im Vordergrund ist. Öffnen wir aber eine andere Activity oder ist unser Gerät ausgeschaltet, dann werden keine Schritte gezählt, was natürlich nicht besonders nützlich ist. Deswegen wollen wir in dieser Anwendung einen Service die Schritt zählen lassen.

Dazu mischen wir unseren Code aus der *StepCounterActivity* aus Kapitel sechs mit dem *SimpleService* oben:

```
public class StepCounterService
    extends Service
    implements SensorEventListener {
    ...
}
```

Da die Service Klasse keine onResume() und onPause() Methode hat, schieben wir den Code zum Registrieren des StepCounter Sensors in die *onStartCommand()* Methode und zum Deregistrieren in die *onDestroy()* Methode.

Ein User Interface hat ein Service natürlich keines, deswegen lassen wir alles was mit dem TextView zu tun hat weg. Interessanterweise können Services aber toasten, d.h. immer wenn die *onSensorChanged()* Methode aufgerufen wird, also ein Schritt erkannt wurde, zeigen wir die Anzahl der Schritte in einem Toast an:

```
public final void onSensorChanged(SensorEvent event) {
    float steps = event.values[0];
    Toast.makeText(this, "steps=" + steps,
        Toast.LENGTH_SHORT).show();
}
```

Zum Starten unseres Services benötigen wir wieder eine *StepCounterServiceActivity*, die identisch zur SimpleServiceActivity ist.

StepCounterBinderService

Wenn uns das stört, dass wir alle paar Schritte "getoastet" werden, dann müssen wir irgendwie auf den Service und seine Daten zugreifen können, damit wir die dann in einer Activity anzeigen können. Dazu müssen wir den Service an die Activity "binden".

Als Erstes schreiben wir eine Klasse namens *StepBinder* vom Typ Binder,

```
public class StepCounterBinderService
    extends Service
    implements SensorEventListener {

    private final IBinder mBinder =
        new StepBinder();
```

```
        public class StepBinder extends Binder {

            StepCounterBinderService getService() {
                return StepCounterBinderService.this;
            }
        }
        ...
    }
```

die die Methode *getService()* enthält, welche uns einfach eine Referenz auf eine Instanz unseres Services gibt. In der *onBind()* Methode, die wir jetzt das erste Mal verwenden, geben wir einfach eine Referenz auf den Binder zurück:

```
        public IBinder onBind(Intent intent) {
            return mBinder;
        }
```

Damit haben wir also einen Referenz auf unseren Service, und können auf all seine public Methoden zugreifen. Deswegen wenn wir noch eine Methode namens

```
    public float getNumberOfSteps() {
        return steps;
    }
```

haben, können wir aus unsere Activity auf die Schritte zugreifen.

Kommen wir zur Activity: Hier müssen wir uns mit Hilfe der Methode *bindService()* mit dem Service verbinden,

```
        protected void onStart() {
            super.onStart();
            Intent intent = new Intent(this,
                StepCounterBinderService.class);
            bindService(intent, mConnection, Context.BIND_AUTO_CREATE);
        }

        protected void onStop() {
            super.onStop();
            unbindService(mConnection);
        }
```

und natürlich später die Verbindung auch wieder mit *unbindService()* aufheben. Um jetzt über den Binder an den Service ran zu kommen, benötigen wir noch die *mConnection*, eine ServiceConnection die wir in der *onCreate()* definieren:

```
        private StepCounterBinderService mService;
        private ServiceConnection mConnection;

        public void onCreate(Bundle savedInstanceState) {
            ...
            mConnection = new ServiceConnection() {

                @Override
                public void onServiceConnected(ComponentName className,
                                               IBinder service) {
                    StepBinder binder = (StepBinder) service;
                    mService = binder.getService();
                }
```

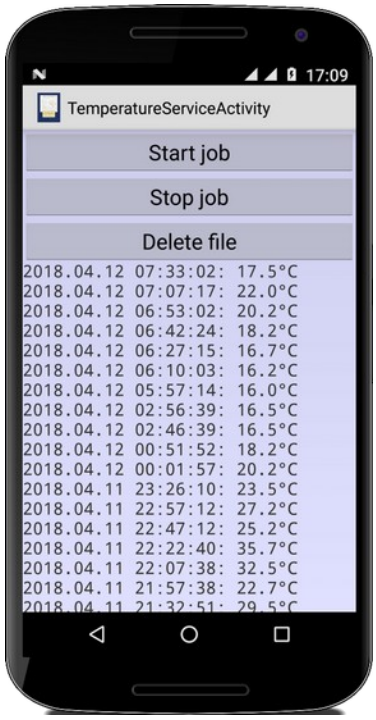
```
            @Override
            public void onServiceDisconnected(ComponentName arg0) {
                // do nothing
            }
        };
        ...
}
```

Jetzt haben wir mit *mService* eine Referenz auf den Service und können dort alle Methoden aufrufen die public sind, z.B. die *getNumberOfSteps()* Methode.

Wenn wir Daten von der Activity an den Service übergeben wollen, dann geht das am besten beim Starten des Services: der Service wird ja über einen Intent gestartet, und dem Intent können wir ja Extras mitgeben, wie wir bereits im ersten Kapitel gesehen haben.

Offensichtlich ist das Ganze nicht ganz trivial, deswegen sollte man sich zweimal überlegen ob die direkte Kommunikation zwischen Activty und Service wirklich nötig ist. Im nächsten Beispiel sehen wir nämlich, dass es auch einfacher geht.

TemperatureService



Wenn wir z.B. die Temperatur in einem Raum kontinuierlich überwachen wollen, dann bietet es sich an einen Service dafür zu verwenden. Da wir aber die Temperatur nicht im Sekundentakt benötigen, sondern nur ein paar Mal pro Stunde, eignet sich ein *JobService* dafür am besten. Wie wir Temperaturen mit der Batterie messen können haben wir ja im Sensor Kapitel bereits gesehen.

Ein JobService hat eine *onStartJob()* und eine *onStopJob()* Methode,

```
public class TemperatureService extends JobService
{
    @Override
    public boolean onStartJob(JobParameters params)
    {
        ...
        return true;
    }

    @Override
    public boolean onStopJob(JobParameters params)
    {
        ...
        return false;
    }
}
```

die genau das machen was man von ihnen erwartet.

Gestartet wird der JobService in der Regel aus einer Activity. Dazu sagen wir ihm welche JobService-Klasse gestartet werden soll, und in welchem REFRESH_INTERVAL der Service aufgerufen werden soll:

```
public class TemperatureServiceActivity extends Activity {

    private final int JOB_ID = 42;
    private final int REFRESH_INTERVAL = 15 * 60 * 1000;

    private JobScheduler mJobScheduler;
```

```
    public void onCreate(Bundle savedInstanceState) {
        ...
        mJobScheduler = (JobScheduler)
            getSystemService(Context.JOB_SCHEDULER_SERVICE);
        ...

        ComponentName componentName = new ComponentName(
            getApplicationContext(), TemperatureService.class);
        JobInfo.Builder jobBuilder =
            new JobInfo.Builder(JOB_ID, componentName);
        jobBuilder.setPeriodic(REFRESH_INTERVAL);
        // jobBuilder.setExtras(bundle).build();
        JobInfo jobInfo = jobBuilder.build();

        if (mJobScheduler.schedule(jobInfo) ==
                                        JobScheduler.RESULT_SUCCESS) {
            Log.i("", "Success");
        } else {
            Log.i("", "Failure");
        }

        ...
        mJobScheduler.cancelAll();
        ...
    }
}
```

Bei neureren Android Systemen kann man nur Intervalle von mindestens 15 Minuten verwenden. Wie wir im Screenshot sehen können, scheint der JobService aber sogar damit Probleme zu haben, wenigstens nachts über.

Fast hätte ich es vergessen, in der AndroidManifest Datei müssen wir noch um Erlaubnis fragen, ob wir einen Job schedulen dürfen:

```
<service
    android:name="variationenzumthema_st3.TemperatureService"
    android:permission="android.permission.BIND_JOB_SERVICE" />
```

SimpleReceiver

Eine andere Möglichkeit Services zu starten ist mittels eines *BroadcastReceiver*. BroadcastReceiver sind relativ unscheinbar und einfach zu programmieren. Man muss lediglich die Methode *onReceive()* überschreiben:

```
public class SimpleReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
        Vibrator v = (Vibrator)
            context.getSystemService(Context.VIBRATOR_SERVICE);
        v.vibrate(500);
        Toast.makeText(context, "Hi from SimpleReceiver!",
            Toast.LENGTH_SHORT).show();
    }
}
```

Gestartet werden BroadcastReceiver durch Systemevents, die man in der AndroidManifest Datei festlegt:

```
<receiver
    android:name="variationenzumthema_st3.SimpleReceiver"
    android:enabled="true"
    android:exported="true" >
    <intent-filter>
        <action android:name=
            "android.intent.action.ACTION_POWER_CONNECTED" />
    </intent-filter>
</receiver>
```

Wenn man die Liste der möglichen Systemevents durchgeht, und man auch nur das kleinste Fünkchen Überwachungsphobie hat, wird einem ganz anders. Z.B. gibt es da:

```
ACTION_POWER_CONNECTED
ACTION_SHUTDOWN
BATTERY_LOW
BOOT_COMPLETED
DATA_SMS_RECEIVED
DOCK_EVENT
DREAMING_STARTED
HEADSET_PLUG
MEDIA_MOUNTED
NEW_OUTGOING_CALL
PACKAGE_INSTALL
SCREEN_OFF
USER_PRESENT
...
```

Da man als Enduser weder merkt, dass ein Receiver läuft, noch bei der Installation darauf hingewiesen wird, hat man noch einen weiteren Grund an manche Verschwörungstheorie zu glauben.

Eine Sache noch: natürlich ist es ganz wichtig, dass der Nutzer einem explizit erlaubt, dass man den Vibrator verwenden darf:

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

Eigentlich lächerlich.

Wenn Sie jemanden zum Wahnsinn treiben wollen, dann starten Sie aus einem Service heraus (der von einem Receiver gestartet wird) zu zufälligen Zeiten den Vibrator. Bis die Leute raus finden wo das Vibrieren herkommt, sind die längst in der Klapse.

MusicReceiver

Um mal zu sehen, wie man von einem Receiver heraus einen Service starten kann, sehen wir uns kurz das folgende Beispiel an:

```
public class MusicReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
        Intent musicServiceIntent =
            new Intent(context, MusicService.class);
        context.startService(musicServiceIntent);
    }
}
```

In dem Fall starten wir unseren MusicService von oben. Natürlich müssen wir den Receiver wieder in der AndroidManifest Datei mit einem Systemevent verbinden. Hypothetisch könnte die Musik jetzt in einer Endlosschleife laufen (*player.setLooping(true)*), und der Nutzer wird die Musik nur durch Deinstallieren der App wieder los (oder über die Developer Optionen den Service abschießen).

NotificationReceiver

Der MusicReceiver oben war ein Beispiel wie man es nicht machen sollte. Denn eigentlich sollte unser Nutzer immer die Möglichkeit haben, einen Service zu beenden. Dazu müsste er aber wissen, dass ein Service läuft. Das macht man am besten über *Notifications*. Notifications sind kleine Icons, die in der Statuszeile angezeigt werden, über die der Nutzer mit unserer Anwendung interagieren kann.

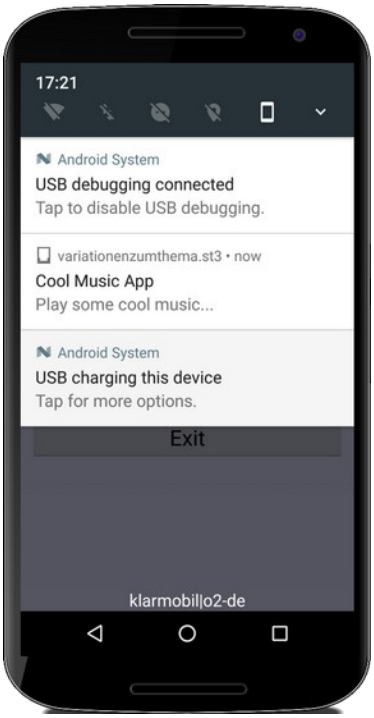
Wir beginnen wieder mit dem Receiver,

```
public class NotificationReceiver
                      extends BroadcastReceiver {

    @Override
    public void onReceive(Context context,
                          Intent intent) {
        Intent musicServiceIntent =
            new Intent(context,
                       NotificationService.class);
        context.startService(musicServiceIntent);
    }
}
```

Der Receiver started den *NotificationService*, der vom Code her identisch mit unserem MusicService oben ist, der einzige Unterschied sind die folgenden Zeilen:

```
public class NotificationService extends Service {

    private final int NOTIFICATION_ID = 42;
    ...

    @Override
    public void onCreate() {
        ...
        Intent notificationIntent =
            new Intent(this, NotificationActivity.class);
        PendingIntent pendingIntent =
            PendingIntent.getActivity(this, 0, notificationIntent, 0);
        Notification notification =
            new Notification.Builder(this)
                .setSmallIcon(R.drawable.notification_icon)
                .setContentTitle("Cool Music App")
                .setContentText("Play some cool music...")
                .setContentIntent(pendingIntent)
                .build();

        startForeground(NOTIFICATION_ID, notification);
    }

    ...
}
```

Wir basteln hier also eine Notification, versehen sie mit einem eigenen Icon (kann nur transparent-weiß sein), einem Titel und Text, sowie einem Intent, der gestartet werden soll, wenn der Nutzer auf die Notification klickt. In unserem Fall ist das die NotificationActivity:




```
public class NotificationActivity extends Activity
{

    @Override
    public void onCreate(Bundle savedState) {
        super.onCreate(savedState);

        final Intent notificationServiceIntent =
            new Intent(getApplicationContext(),
                       NotificationService.class);

        LinearLayout ll = new LinearLayout(this);
        ll.setBackgroundColor(0x200000ff);
        ll.setOrientation(LinearLayout.VERTICAL);

        Button btn = new Button(this);
        btn.setText("End NotificationService");
        btn.setOnClickListener(
            new OnClickListener() {

            @Override
            public void onClick(View v) {
                stopService(notificationServiceIntent);
                Toast.makeText(v.getContext(),
                    "NotificationService stopped",
                    Toast.LENGTH_LONG).show();
            }
        });

        ll.addView(btn);
        setContentView(ll);
    }
}
```

Diese Activity sollte mindestens dem Nutzer erlauben, den Service zu beenden. Dann ist alles gut.

PhoneCallReceiver

Zum Abschluss noch ein kleines Schmankerl: auf Stack Overflow habe ich den AbstractPhoneCallReceiver gefunden. Das ist ein klassischer BroadcastReceiver so wie wir sie gerade verwendet haben. Interessant ist, dass er auf das Telefon hört, deswegen braucht er auch zwei Permissions,

```
<uses-permission android:name=
    "android.permission.READ_PHONE_STATE" />
<uses-permission android:name=
    "android.permission.PROCESS_OUTGOING_CALLS" />
```

die eine für eingehende und die andere für ausgehende Telefonate. Das schöne an dem AbstractPhoneCallReceiver ist, dass er uns die Schmutzarbeit abnimmt, und wir müssen lediglich die Methoden überschreiben, die uns interessieren, als da sind:

```
public class PhoneCallReceiver extends AbstractPhoneCallReceiver {

    @Override
    protected void onIncomingCallStarted(Context context,
                                         String number, Date start) {
        Toast.makeText(context, "onIncomingCallStarted(): "+number,
            Toast.LENGTH_LONG).show();
        //recordAudio();
    }

    @Override
    protected void onOutgoingCallStarted(Context context,
                                         String number, Date start) {
        Toast.makeText(context, "onOutgoingCallStarted(): "+number,
            Toast.LENGTH_LONG).show();
    }

    @Override
    protected void onIncomingCallEnded(Context context, String number,
                                       Date start, Date end) {
        Toast.makeText(context, "onIncomingCallEnded(): "+number,
            Toast.LENGTH_LONG).show();
        //playAudio();
    }

    @Override
    protected void onOutgoingCallEnded(Context context, String number,
                                       Date start, Date end) {
        Toast.makeText(context, "onOutgoingCallEnded(): "+number,
            Toast.LENGTH_LONG).show();
    }

    @Override
    protected void onMissedCall(Context context, String number,
                                Date start) {
        Toast.makeText(context, "onMissedCall(): "+number,
            Toast.LENGTH_LONG).show();
    }
}
```

Wenn wir also jetzt beim Telefonieren zuhören wollen, müssen wir mit der Audioaufnahme in der *onIncomingCallStarted()* Methode beginnen und in der *onIncomingCallEnded()* Methode wieder aufhören. Wie das mit der Audioaufnahme funktioniert haben wir ja im Multimedia Kapitel besprochen.

WhoStoleMyPhoneService

Die Idee hinter *WhoStoleMyPhoneService* ist ganz einfach: wir wollen ein Bild mit der Selfie-Kamera machen, wann immer eine der folgenden Aktionen passiert:

ACTION_SCREEN_ON, ACTION_SCREEN_OFF oder ACTION_USER_PRESENT.

Dazu schreiben wir einen Service der in der *onStartCommand()* Methode einen BroadcastReceiver registriert, der auf diese Aktionen hört:

```
public class WhoStoleMyPhoneService extends Service {
    private BroadcastReceiver mReceiver;
```

```
        @Override
        public int onStartCommand(Intent intent, int flags, int startId) {
            IntentFilter intentFilter =
                new IntentFilter(Intent.ACTION_SCREEN_ON);
            intentFilter.addAction(Intent.ACTION_SCREEN_OFF);
            intentFilter.addAction(Intent.ACTION_USER_PRESENT);
            mReceiver = new WhoStoleMyPhoneReceiver();
            registerReceiver(mReceiver, intentFilter);

            return START_STICKY;
        }
```

Der BroadcastReceiver ist eine lokale Klasse, die in der *onReceive()* Methode einfach ein Bildchen macht:

```
        private class WhoStoleMyPhoneReceiver extends BroadcastReceiver {
            private Camera camera;
            private SurfaceTexture surfaceTexture;

            @Override
            public void onReceive(Context context, Intent intent) {
                if (intent.getAction().equals(Intent.ACTION_USER_PRESENT))
                {
                    surfaceTexture = new SurfaceTexture(42);
                    takePicture();
                    Toast.makeText(context, "Smile!",
                        Toast.LENGTH_SHORT).show();
                }
            }
            ...
        }
    }
```

Idealerweise wird das Bildchen nicht lokal, sondern im Internet gespeichert, sonst hat das Ganze ja wenig Sinn. Und natürlich muss der Service noch irgendwie gestartet werden.

Fragen

1. Geben Sie ein Beispiel wofür man typischerweise die Service Klasse und wofür die BroadcastReceiver Klasse verwenden könnte.

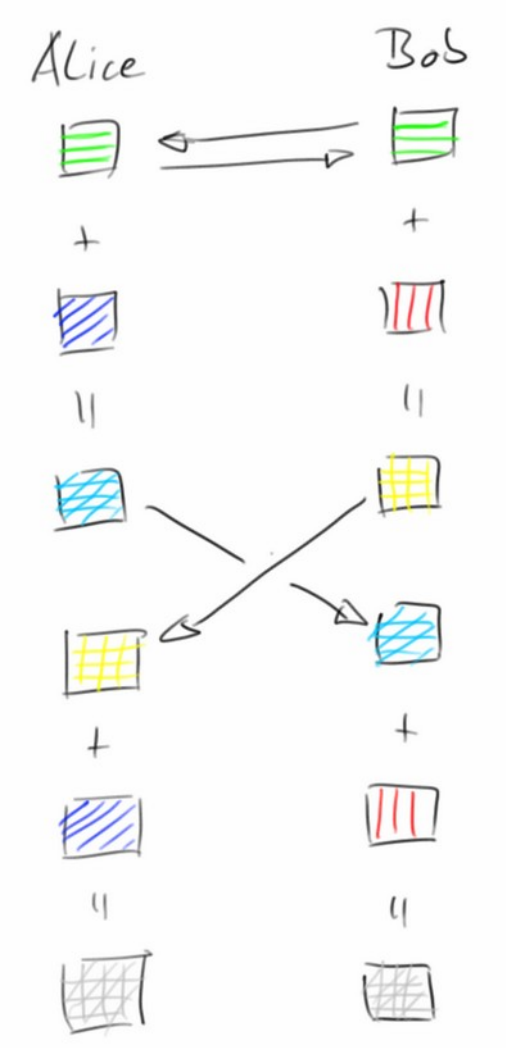
2. In diesem Kapitel haben Sie von den sogenannten BroadcastReceivern gehört. Wofür sind diese nützlich, geben Sie vielleicht ein Beispiel.

3. In Android gibt es vier Arten von Applikationen:
   - Activity
   - Service
   - BroadcastReceiver
   - ContentProvider

   Beschreiben Sie jede kurz und geben Sie je ein Beispiel.

4. Was müssen Sie tun, damit ihre Programm startet wenn der Systemevent "ACTION_POWER_CONNECTED" ausgelöst wird.

Special Topic: Cryptography



Die Ursprünge der Cryptography liegen schon bei den alten Griechen [1] und sogar davor: wie sende ich eine Nachricht an jemanden, ohne dass andere diese Nachricht lesen können. Schon im ersten Semester haben wir die Caesar Cipher [2] kennen gelernt, die Julius Caesar verwendet haben soll, um mit seinen Generälen zu kommunizieren. Zur Cryptography gehört aber noch viel mehr als nur Verschlüsselung.

Die Grundlagen dafür haben wir im zweiten Semester gelegt: Ganz wichtig sind gute Zufallszahlen, genauer Pseudo-Zufallszahlen. Praktisch alle modernen Verschlüsselungsverfahren beruhen auf Pseudo-Zufallszahlengeneratoren. Außerdem haben wir das Konzept der Hashfunktion kennengelernt als wir mit der HashMap und dem HashSet zu tun hatten. Auch in der Cryptography gibt es Hashfunktion, sogenannte Crypto-Hashverfahren. Und auch die Idee der Prüfsumme (Checksum), wie wir sie beim Luhn-Algorithmus für Creditkarten kennen gelernt haben, werden wir wieder benötigen. Deswegen haben wir eigentlich alle Grundlagen um uns jetzt ein bisschen mit modernen Verfahren der Cryptography zu beschäftigen.

An dieser Stelle vielleicht noch ein kleines Wort der Vorsicht: mit Java ist die Verwendung von kryptografischen Algorithmen sehr einfach geworden. Und auch dieses zugegebener Maßen sehr oberflächliche Kapitel gibt vielleicht den Eindruck, dass das Ganze gar nicht so kompliziert ist. Das ist aber ein falscher Eindruck, denn der Teufel liegt wirklich im Detail und in der richtigen Verwendung des richtigen Algorithmus für die jeweilige Anwendung. Wer Cryptography ernsthaft anwenden will, kommt nicht drum herum sich mit der Materie tiefer auseinanderzusetzen. Einen sehr schönen Einstieg liefert die Veranstaltung "Online Cryptography" von Dan Boneh [3].

Wenn wir mit kryptografischen Algorithmen arbeiten, werden wir es sehr häufig mit binären Daten, vor allem Bytearrays, zu tun haben. Um diese binären Daten menschenlesbar zu machen, haben sich zwei Verfahren etabliert: einmal die hexadezimale Schreibweise und zum anderen die Darstellung als Base64.

Hex

In der hexadezimale Schreibweise teilt man die 8-bit eines Bytes einfach in die oberen und die unteren vier Bit. Da man mit vier Bit Zahlen zwischen 0 und 15 darstellen kann, werden diese einfach auf die Ziffern 0-9 und die ersten sechs Buchstaben des Alphabets A-F abgebildet:

```
char[] HEX_VALUES = { '0', '1', '2', '3', '4', '5', '6', '7',
                      '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' };
```

Wenn wir jetzt ein Byte haben nehmen wir uns erst einmal die unteren vier Bit vor, die wir erhalten indem wir eine bitweise AND Operation auf unser Byte ausführen:

```
byte tmp = 153;
byte lowerFourBits = tmp & 0x0F;
char hex1 = HEX_VALUES[ lowerFourBits ];
```

Dann kommen die oberen vier Bit dran: dazu verwenden wir den *unsigned right shift operator* ">>>", der einfach eine Null links einfügt und alle übrigen Bits um eins nach rechts verschiebt. Das unterste Bit geht dabei verloren. Wenn wir das vier mal machen, dann haben wir die oberen vier Bits an der Stelle der ehemaligen unteren vier Bits, und links stehen vier Nullen. Der Rest geht dann wie oben:

```
byte tmp = 153;
byte higherFourBits = tmp >>> 4;
char hex2 = HEX_VALUES[ higherFourBits ];
```

Das Ganze können wir dann in einer kompakten Methode zusammenfassen, die wir häufiger verwenden werden und die uns aus einem Byte-Array einen Hex-String macht:

```
public static String byteArrayToHexString(byte[] buffer) {
    char[] hex = new char[buffer.length * 2];
    int tmp;
    for (int j = 0; j < buffer.length; j++) {
        tmp = buffer[j] & 0xFF;
        hex[j * 2] = HEX_VALUES[tmp >>> 4];
        hex[j * 2 + 1] = HEX_VALUES[tmp & 0x0F];
    }
    return new String(hex);
}
```

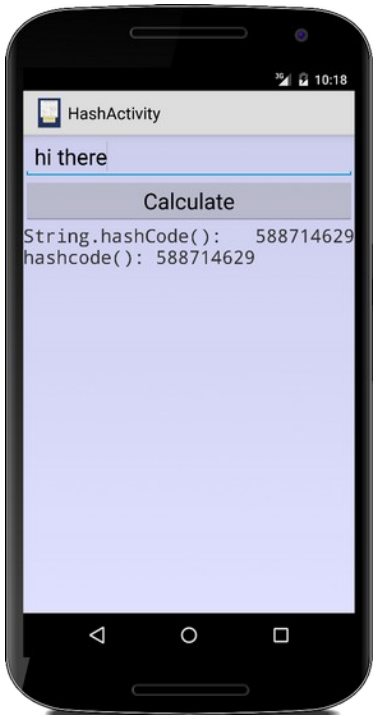
Dazu gibt es auch die umgekehrte Methode, *hexStringToByteArray(hexString)*, die aus einem Hex-String wieder ein Byte-Array macht.

Base64

Die Darstellung von Binärdaten als Hex-String ist nicht besonders effektiv und verschwendet eigentlich viel Platz. Wir haben ja noch die ganzen anderen Buchstaben, die kleinen und die großen, und auch noch einige Sonderzeichen. Deswegen hat sich das Base64 Verfahren etabliert [4], das relativ effektiv mit Platz umgeht und trotzdem lesbar bleibt. Die Details müssen wir nicht verstehen, lediglich wie man es verwendet:

```
byte[] ba = new byte[8];
String base64 = Base64.encodeToString(ba, Base64.DEFAULT);

String base64 = "CB1bmQgUXVhcms=";
byte[] ba = Base64.decode(base64, Base64.DEFAULT);
```

Vielleicht sollten man noch andeuten, dass es sich bei Base64 um eine Kodierung handelt, nicht um eine Verschlüsselung.



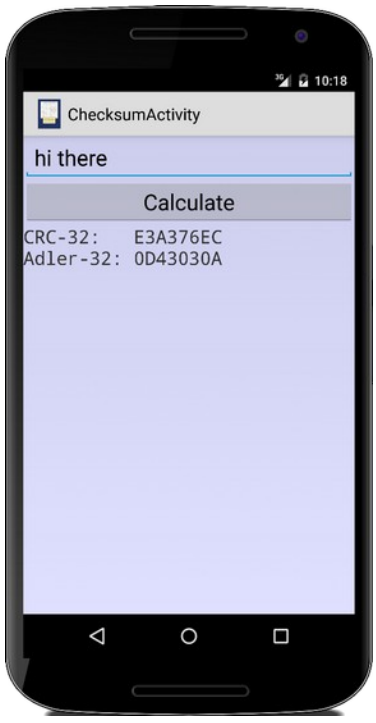
Hash

In Java hat jede Klasse eine *hashCode()* Methode. Z.B. für die Klasse String sieht diese grob so aus:

```
public int hashCode(char[] value) {
    int h = 0;
    for (int i = 0; i < value.length; i++) {
        h = 31 * h + value[i];
    }
    return h;
}
```

Es wird also aus einem beliebig großen String eine 32-Bit Zahl gemacht. Verwendet wird die Methode in den Klassen HashMap und HashSet damit diese möglichst effektiv Daten lesen und schreiben können.

Was sind denn die Anforderung an so eine Hash-Funktion? Zum Einen soll sie möglichst schnell sein, und zum Andern soll es möglichst selten zu Kollisionen kommen, also dass für zwei unterschiedliche Strings der gleiche Hashwert raus kommt. Speziell die zweite Anforderung ist für Crypthashfunktionen ganz wichtig.



Checksum

Kommen wir zu den Prüfsummen. Bei Prüfsummen geht es eigentlich weniger darum, dass sie super-schnell sind, sondern eher darum, dass sie Fehler finden. Z.B. beim Luhn-Algorithmus für Kreditkarten, den wir im zweiten Semester kennen gelernt haben, geht es darum Tippfehler und Zahlendreher aufzudecken. Ähnliches tun auch die Algorithmen für die ISBN und IBAN Nummern.

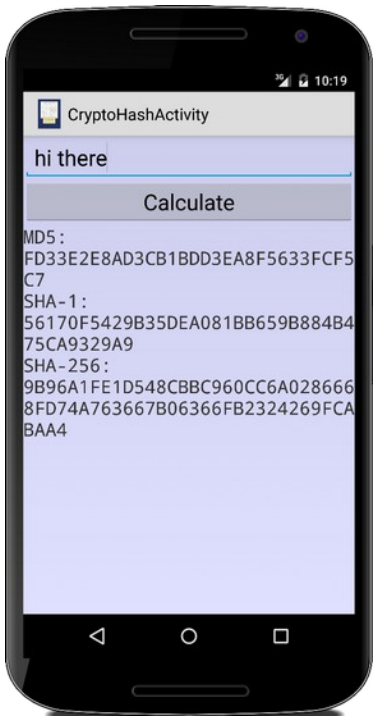
Diese Algorithmen funktionieren ganz gut für Werte fester Länge, wie z.B. Kreditkartennummern u.ä., die ja immer 16 Zahlen lang sind. Wenn wir es aber mit Bytearrays von unterschiedlicher Länge oder sogar mit Datenströmen zu tun haben, dann wird sehr häufig die *CRC-32* oder auch die *Adler-32* Checksum verwendet [5,6]. Z.B. sowohl im IP-Protokoll, als auch im TCP-Protokoll wird die CRC Prüfsumme verwendet um Fehler in der Datenübertragung festzustellen [7].

Beide Algorithmen sind im *java.util.zip* Paket von Java implementiert und ganz einfach zu verwenden:

```
byte bytes[] = "Hello world".getBytes();
Checksum checksum = new CRC32();
checksum.update(bytes, 0, bytes.length);
System.out.println("Check sum is: " + checksum.getValue());
```

Sie sind sehr schnell und entdecken einfache Fehler.

CryptoHash



Bei kryptografischen Hashfunktionen [8] handelt es sich um spezielle Prüfsummen, die zu den normalen Eigenschaften einer Prüfsumme zusätzlich noch explizit folgende haben sollten:

- es muss praktisch unmöglich sein vom Hashwert auf den Wert der Ausgangsdaten zu schließen,
- kleine Änderungen in den Ausgangsdaten sollen zu großen Änderungen im Hashwert führen, und
- es soll quasi unmöglich sein, zwei Sätze von Ausgangsdaten zu finden, die zum gleichen Hashwert führen.

Keiner der Algorithmen die wir bisher gesehen haben hat diese Eigenschaften, aber kryptografischen Hashfunktionen haben sie.

Wofür verwendet man kryptografischen Hashfunktionen aber nun? Die drei wichtigsten sind:

- zum Speichern von Passwörtern in Datenbanken,
- um festzustellen ob Daten verändert wurden, oder
- zum Identifizieren von Dateien oder Daten.

Neuerdings mit Cryptowährungen kommt auch noch die Idee "proof-of-work" hinzu, dazu später mehr.

In der Verwendung sind die kryptografischen Hashfunktionen ganz einfach. Man holt sich erst den Algorithmus seiner Wahl, z.B. den *MD5*, führt ihm dann mittels *update()* die Daten zu, und verdaut das Ganze am Ende mit der *digest()* Methode:

```
MessageDigest md = MessageDigest.getInstance("MD5");
md.update( byteArray );
byte[] hashCode = md.digest();
```

Der *hashCode* den man dann erhält ist je nach Algorithmus zwischen 128 Bit (MD5) und 512 Bit (SHA-512) lang. Die vier wichtigsten Cryptohash Algorithmen sind:

- MD5
- SHA-1
- SHA-2 in den Versionen SHA-256 und SHA-512 [9]
- bcrypt [10]

Die ersten beiden sollte man aber heutzutage nicht mehr verwenden, da sie nicht mehr als sicher gelten.

Rainbow Tables

Betrachten wir das Speichern von Passwörtern in Datenbanken etwas näher: Die Passwörter dort im Klartext zu speichern ist ziemlicher Unsinn, denn sollte mal jemand in unsere Datenbank hacken, dann hat er alle Passwörter. Passiert überraschend häufig [11]. Deswegen speichert man den Cryptohashwert des Passwortes in der Datenbank. Nehmen wir an unser Passwort ist das übliche, also "123456", dann ist dessen MD5 Wert der Hex-String:

```
E10ADC3949BA59ABBE56E057F20F883E
```

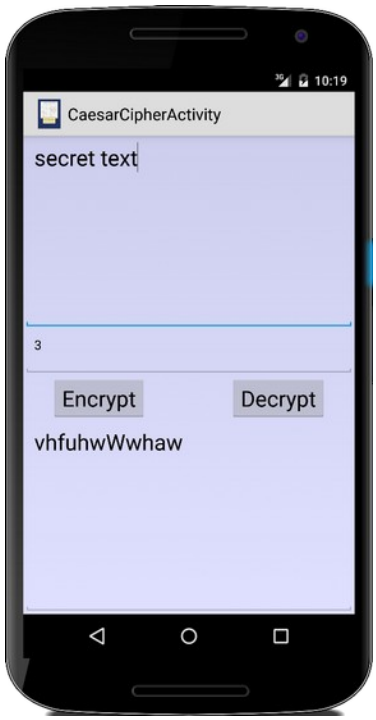
Wegen der oben genannten Eigenschaften der Cryptohashfunktionen kann man das Passwort nur durch Brute-Force wieder herausfinden, soll heißen, man muss alle möglichen String-Kombination ausprobieren. Nun würde man denken das macht doch keiner. Aber dem ist nicht so: angefangen damit hat die NSA, und inzwischen gibt's diese sogenannten *Rainbow Tables* im Internet zum runter laden.

Wie können wir der NSA und anderen Hackern aber die Suppe versalzen? Richtig, mit Salz:

```
MessageDigest md = MessageDigest.getInstance("MD5");
md.update( byteArray );

byte[] salt = new byte[64];
new SecureRandom().nextBytes(salt);
md.update(salt);

byte[] hashCode = md.digest();
```

Wir müssen uns natürlich unser Salz merken und an einer sicheren Stelle aufbewahren, was nicht ganz trivial ist. Aber damit sind unsere Passwörter dann auch vor Brute-Force Attacken sicher.

CaesarCipher



Das erste Verschlüsselungsverfahren stammt von den alten Griechen [1] und wird heute häufig als Caesar Cipher bezeichnet [2]. Eigentlich ist es nichts anderes als Addition kombiniert mit unserem alter Freund dem Restwertoperator, angewendet auf Buchstaben. Wir haben eine einfache Version bereits im ersten Semester geschrieben, jetzt machen wir daraus eine App.

Man wählt als erstes einen Schlüssel, das ist eine Zahl zwischen 1 und 25:

```
int key = 12;
```

Zur Verschlüsselung addiert man den Schlüssel dann zu dem Buchstaben den man verschlüsseln möchte und wendet danach den Restwertoperator darauf an:

```
private char encryptChar(char c, int key) {
    int d = c - 'a';
    int e = d + key;  // encryption happens here
    int f = e % 26;
    char g = (char) (f + 'a');
    return g;
}
```

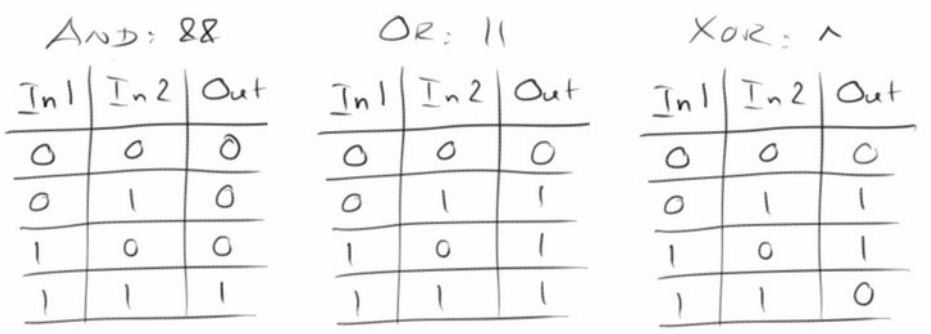
Das macht man dann für alle Buchstaben (die am besten alle klein geschrieben sind). Für die Entschlüsselung dreht man das Ganze einfach um, man subtrahiert also den Schlüssel.

Damit man also Nachrichten verschlüsseln und entschlüsseln kann, benötigt man den Schlüssel. Hier bei Caesar ist das eine Zahl zwischen 1 und 25. Das ändert sich auch bei den modernen Verfahren nicht, die Zahl wird nur größer. Jeder der den Schlüssel hat, kann die Nachrichten entschlüsseln [12]. Deswegen ist das Problem aller Verschlüsselungsverfahren, wie tauscht man den Schlüssel so aus, dass niemand anderes ihn zu sehen bekommt. Das Problem wurde erst 1976 von Diffie und Hellman gelöst.

Eine Anmerkung noch, die Caesar Cipher ist natürlich nicht besonders sicher, und lässt sich heute ganz einfach mit statistischen Verfahren knacken.

XORCipher

Kommen wir zur Mutter aller modernen Verschlüsselungsverfahren, der XOR Cipher. Auch die haben wir schon im ersten Semester kennen gelernt, damals haben wir die Mona Lisa mit dem Taj verheiratet (Steganographie). Zur Erinnerung, die Wahrheitstabellen für die drei logischen Operationen AND, OR und XOR, sehen wie folgt aus:



Das sieht jetzt erst mal recht langweilig aus und scheint mit Verschlüsselung nicht besonders viel zu tun zu haben. Allerdings hat die XOR Operation eine sehr interessante Eigenschaft: Nehmen wir den Buchstaben 'a', dessen ASCII Wert ist 97. Dann nehmen wir irgendeine Zufallszahl, z.B. 42, und die beiden verknüpfen wir mit der XOR Operation:

```
  0110 0001 (97)
^ 0010 1010 (42)
-----------
= 0100 1011 (75)
```

Wenn wir jetzt dieses Resultat, die 75, mit der 42 noch mal per XOR verknüpfen,

```
  0100 1011 (75)
^ 0010 1010 (42)
-----------
= 0110 0001 (97)
```

dann kommt da wieder unser 'a' raus! Wenn wir das mit dem vergleichen was wir gerade in der Caesar Cipher gesehen haben, dann sieht das verdammt nach Verschlüsselung aus: das erste Mal haben wir verschlüsselt, das zweite Mal entschlüsselt, und der Schlüssel ist die 42.

One-Time Pad

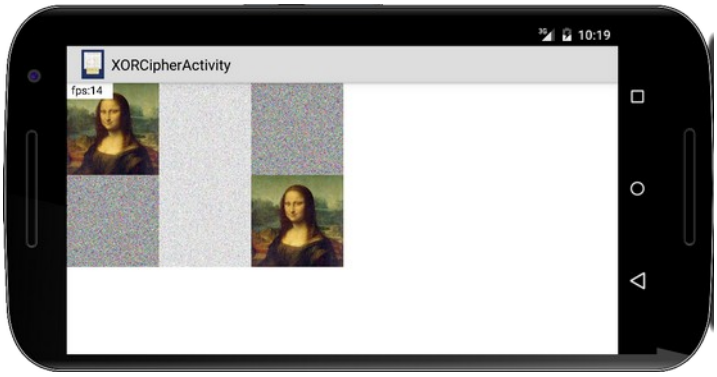
Wie gut unser XOR-Verschlüsselung ist, hängt nur von der Länge und der Zufälligkeit des Schlüssels ab. Ist der Schlüssel 8 Bit lang, wie gerade, dann ist die XOR Cipher keinen Deut besser als die Caesar Cipher. Ist die Länge des Schlüssels aber genauso lang wie die Nachricht, und ist der Schlüssel vollkommen zufällig, dann nennt man diese Art der Verschlüsselung die One-Time Pad Verschlüsselung (Einmalverschlüsselung) [13]. Diese ist absolut sicher, d.h. kann nicht geknackt werden. Das Problem ist allerdings der Schlüssel, der ist nämlich ziemlich groß (mindestens genauso groß wie die Nachricht), und muss irgendwann vorher mal zwischen Sender und Empfänger ausgetauscht worden sein.

Lehmer

Aber mit Zufallszahlen kennen wir uns aus (wenigstens wenn wir im zweiten Semester aufgepasst haben): da gab es nämlich den Herrn Lehmer mit seinen Pseudozufallszahlen. Die einfache Idee: man verwendet anstelle des One-Time Pads Pseudozufallszahlen. Wie stellt man aber sicher, dass Sender und Empfänger die gleichen Pseudozufallszahlen verwenden? Wenn wir uns erinnern, sind die Pseudozufallszahlen in Lehmer's Algorithmus (und allen andern Pseudozufallszahlenalgorithmen) eindeutig durch die sogenannte *Seed* festgelegt. D.h., wir müssen dem Empfänger lediglich unsere Seed mitteilen, damit er die Nachricht entschlüsseln kann. Also ist die Seed der Schlüssel in diesem Fall.

Eigentlich ist Cryptography nichts anderes als Potenzieren, mit dem Restwertoperator richtig hantieren und ein bisschen XOR.

Mona Lisa

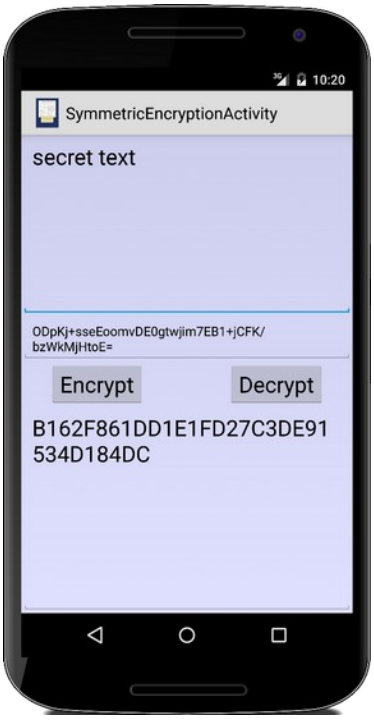


Als kleine Anwendung für die XOR Cipher wollen wir die Mona Lisa verschlüsseln. Der Code ist eigentlich identisch mit dem aus dem ersten Semester: lediglich das zweite Bild das wir als Schlüssel verwenden ist jetzt nicht das Taj, sondern ein Bild voller Zufallszahlen:

```
private GImage
createRandomImage(int width,
int height, int seed) {
    rgen.setSeed(seed);

    int[][] array = new int[width][height];
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            array[i][j] = rgen.nextInt();
        }
    }
    return new GImage(array);
}
```

Wir müssen natürlich das gleiche Zufallsbild für Ver- und Entschlüsselung nehmen. Sonst wird das nix.

SymmetricEncryption



Kommen wir zu den heute gängigen symmetrischen Verschlüsselungsverfahren. Sie heißen symmetrisch, weil für die Verschlüsselung der gleiche Schlüssel verwendet wird wie für die Entschlüsselung. Praktisch alle symmetrischen Verschlüsselungsverfahren sind irgendwo Abwandlungen der XOR-Cipher. Es wird in der Regel ein etwas besserer Zufallszahlengenerator verwendet, die Schlüssel (*Seeds*) sind meist etwas länger, und vor allem beim Padding ist man vorsichtiger [14].

Bekannte symmetrische Verfahren sind

- AES
- IDEA
- Blowfish
- Skipjack
- DES
- TripleDES

Dabei wurden DES und TripleDES von der NSA entwickelt, und man kann sich vorstellen wie sicher die sind. Am häufigsten sieht man heutzutage den AES Algorithmus mit einer Schlüssellänge von 256 Bit. Aber auch der IDEA wird öfter verwendet. Blowfish ist auch nicht schlecht. Welche Algorithmen zur Verfügung stehen hängt stark von der Android Version und teilweise auch vom Hersteller ab und dem Land in dem man lebt.

Nachdem wir den Algorithmus ausgewählt haben, müssen wir noch ein Padding wählen. Was ist Padding? Bei all den Algorithmen oben handelt es sich um sogenannt *Block-Ciphers*. D.h. es wird immer ein Block mit x-Bytes verschlüsselt, wobei x meist die Länge des Schlüssels ist. Was ist aber, wenn unsere Nachricht nicht genau x-Byte lang ist, sondern etwas länger oder kürzer? Dann wir der Rest aufgefüllt (padded). Und dieses Auffüllen ist sehr problematisch. Füllt man z.B. mit lauter Nullen auf, dann ist das nicht sehr schlau, denn ein möglicher Attacker kann dann in den verschlüsselten Daten Regelmässigkeiten entdecken und daraus auf den Schlüssel schließen. Deswegen ist das Padding ganz wichtig, und man muss es richtig machen. Wer Details dazu wissen möchte, sollte sich die Vorlesung von Dan Boneh [16] oder Jonathan Katz [17] ansehen.

Beim Padding wählt man erst Mal den Modus Operandi (mode of operation), da gibt es ECB, CBC, CFB und OFB. ECB sollte man nicht mehr verwenden, CFB und OFB sind eher für Stream-Ciphers [15]. Danach wählt man noch die Länge des Padding, und da gibt es vor allem PKCS5Padding und PKCS7Padding. Auch hier sollte man ersteren nicht verwenden, wenn möglich.

Sehen wir uns das Ganze mal in der Praxis an: als erstes legen wir unseren Algorithmus und das Padding fest, z.B.:

```
private final String ALGORITHM = "AES";
private final String PADDING = "AES/ECB/PKCS7Padding";
```

wir haben hier ECB gewählt, weil wir nett zur NSA sein wollen. Sonst wär's ja unfair.

Dann benötigen wir einen Schlüssel, den wir mit der *KeyGenerator* Klasse erzeugen:

```
KeyGenerator kgen = KeyGenerator.getInstance(ALGORITHM);
kgen.init(256); // use a 256 bit length
SecretKey secretKey = kgen.generateKey();
```

Diesen Schlüssel verwenden wir sowohl für die Verschlüsselung als auch für die Entschlüsselung (symmetrisch), wir müssen ihn uns merken, und irgendwie unbemerkt zum Empfänger bringen. Da der Schlüssel eigentlich binär íst (ein Bytearray) macht man diese meist mittels HEX oder Base64 menschenlesbar, z.B.:

```
String base64 = Base64.encodeToString(
                            secretKey.getEncoded(), Base64.DEFAULT);
```

Das Verschlüsseln geht dann wie folgt:

```
String msg = "Hello World!";
Cipher cipher = Cipher.getInstance(PADDING);
cipher.init(Cipher.ENCRYPT_MODE, secretKey);
byte[] encrypted = cipher.doFinal(msg.getBytes("UTF-8"));
```

Wir holen uns eine Cipher mit dem richtigen Padding. Danach initialisieren wir die Cipher mit dem Schlüssel, und sagen ihr noch, dass sie verschlüsseln soll. Das Verschlüsseln selbst passiert dann in der *doFinal()* Methode. Der verschlüsselte Text wird in dem Bytearray *encrypted* gespeichert.

Der Empfänger erhält das Bytearray *encrypted* mit dem verschlüsselten Text. Den Schlüssel hat er früher auf einem anderen Weg irgendwie erhalten. Die Entschlüsselung verläuft nun vollkommen analog (obwohl es digital ist ;-):

```
Cipher cipher = Cipher.getInstance(PADDING);
cipher.init(Cipher.DECRYPT_MODE, secretKey);
byte[] decrypted = cipher.doFinal(encrypted);
String msg = new String(decrypted, "UTF-8");
```

Und mehr ist da nicht.

Symmetrische Verschlüsselungsverfahren sind sehr sicher und auch sehr schnell. Sie haben aber alle ein gemeinsames Problem: den Schlüsselaustausch. Wie können zwei Parteien, die sich vorher noch nie gesehen haben einen Schlüssel austauschen?

PassPhraseEncryption

Bevor wir das Schlüsselaustauschproblem lösen, kommen wir erst noch zu einem anderen Problem: So ein 256 Bit AES Schlüssel sieht wenn man ihn Base64 encoded so aus:

```
5iWRLweSee65JYRd6dnAkJsezKpg9kwyITRGeINfzPE=
```

Das kann sich doch keiner merken. Deswegen gibt es die PassPhrase-Verschlüsselung: man verwendet also ein Passwort oder besser einen ganzen Satz zur Verschlüsselung. Denn je länger das Passwort, desto sicherer die Verschlüsselung.

Als erstes müssen wir einen Algorithmus auswählen,

```
private final String ALGORITHM =
    "PBEWithMD5AndDES";
```

Auf Android Geräten ist die Auswahl an PBE Algorithmen nicht besonders groß, außer dem zweiten hört sich keiner von denen besonders sicher an (und ausgerechnet der zweite funktioniert auf meinem Handy nicht):

- PBEWithMD5AndDES
- PBKDF2WithHmacSHA1
- PBEWithMD5AndDES
- PBEWithMD5AndTripleDES

Falls man hier nach etwas sichererem sucht, dann sollte man sich mal Bouncy Castle ansehen [18].

Zur Verschlüsselung benötigen wir den Passphrase. Aus dem erzeugen wir mittels der *SecretKeyFactory* einen Schlüssel:

```
String passPhrase =
 "@ v3ry l0n9 (0mp1!(a+3d pa$$phra$3, w!+h num83r$ and $p3(!a1 (har$";

KeySpec keySpec =
    new PBEKeySpec(passPhrase.toCharArray(), salt, 1024, 256);
SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(ALGORITHM);
SecretKey secretKey = keyFactory.generateSecret(keySpec);

AlgorithmParameterSpec algSpec =
    new PBEParameterSpec(salt, nrOfIterations);

Cipher cipher = Cipher.getInstance(ALGORITHM);
cipher.init(Cipher.ENCRYPT_MODE, secretKey, algSpec);
byte[] encrypted = cipher.doFinal(msg.getBytes());
```

Neu hier ist das *Salz*, das wir bereits bei den Rainbow-Tables gesehen haben: es versalzt der NSA die Suppe:

```
salt = new byte[64];
new SecureRandom().nextBytes(salt);
```

Zusätzlich haben wir den Parameter *nrOfIterations*, den wir z.B. auf 19 setzen:

```
int nrOfIterations = 19;
```

Er besagt wie oft das Salz in die Suppe gestreut wird. Je häufiger desto salziger die Suppe, desto länger dauert es aber auch bis die Suppe fertig ist.

Zum Entschlüsseln benötigen wir wieder den Passphrase, aber auch das Salz und die *nrOfIterations*:

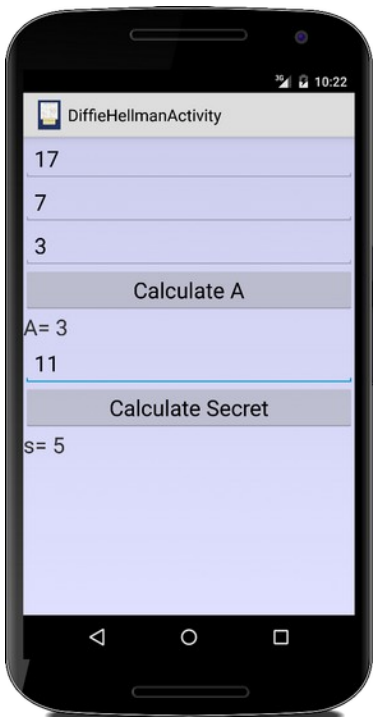
```
String passPhrase =
 "@ v3ry l0n9 (0mp1!(a+3d pa$$phra$3, w!+h num83r$ and $p3(!a1 (har$";

KeySpec keySpec =
    new PBEKeySpec(passPhrase.toCharArray(), salt, 1024, 256);
SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(ALGORITHM);
SecretKey secretKey = keyFactory.generateSecret(keySpec);

AlgorithmParameterSpec algSpec =
    new PBEParameterSpec(salt, nrOfIterations);

Cipher cipher = Cipher.getInstance(ALGORITHM);
cipher.init(Cipher.DECRYPT_MODE, secretKey, algSpec);
byte[] decrypted = cipher.doFinal(encrypted);
String msg = new String(decrypted, "UTF-8");
```

Man liest nun häufig im Internet, dass man das Salz und die Anzahl der Iterationen bedenkenlos mit der verschlüsselten Nachricht übermitteln kann. Mir persönlich scheint das aber sehr komisch. Würd ich nicht machen.

Allgemein muss man sagen, PassPhrase Verschlüsselung ist zwar praktisch von der Handhabbarkeit und Nutzerfreundlichkeit, aber es ist bei weitem nicht so sicher wie die normale, symmetrische Verschlüsselung die wir oben gesehen haben.

Eine Frage: haben Sie einen Passwortmanager auf Ihrem Handy? Haben Sie mal darauf geachtet von welcher Firma dieser hergestellt wurde? Haben Sie gecheckt ob diese Firma vielleicht eine Verbindung zur NSA hat? Sinn würde das schon machen, oder? Aber jetzt haben wir ja unseren eigenen.

Diffie - Hellman



Wir haben es schon mehrmals angesprochen: Das Problem des Schlüsselaustausches. Es wurde 1976 von Diffie und Hellman gelöst [19]. Der Algorithmus ist etwas nicht trivial, aber im Grunde wissen wir schon genug, um ihn zu verstehen.

Prime

Wir müssen uns erst einmal an die Primzahlen aus der Schule erinnern: Primzahlen [20] sind Zahlen, die nur durch die Eins und durch sich selbst geteilt werden können. Die ersten paar Primzahlen sind:

```
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,
47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, ...
```

Co-Prime

Zu Beginn des zweiten Semesters lernten wir den Euklid-Algorithmus kennen, um den größten gemeinsamen Teiler (gcd) zwischen zwei Zahlen zu finden. Für die Mathematiker ist eine Zahl 'a' teilerfremd (coprime) mit einer Zahl 'n', wenn gilt [21]

```
gcd(a, n) = 1
```

Das bedeutet eigentlich 'a' und 'n' haben keinen größten gemeinsamen Teiler.
Das ist es, was teilerfremd bedeutet. Betrachten wir zwei Beispiele: teilerfremd von n = 5 sind:

```
1, 2, 3, und 4
```

oder teilerfremd von n = 8 sind:

```
1, 3, 5, und 7.
```

Congruent

Als nächstes müssen wir lernen, was kongruent für einen Mathematiker bedeutet [22]: Zwei Zahlen 'a' und 'b' gelten als *kongruent modulo* 'n', wenn

```
a = b % n
```

wobei '%' unser geliebter Modulo-Operator ist. Als Beispiel kann man an die Uhr denken: wir sagen z.B., dass 3:00 Uhr nachmittags gleichbedeutend mit 15:00 Uhr ist, denn

```
3 = 15 % 12
```

d.h. die Zahlen 3 und 15 sind *kongruent modulo* 12. Ist doch gar nicht so schwer. Mathematiker verwenden nur lustige Worte für einfache Konzepte.

Primitive Root Modulo n

Nun wirds aber noch lustiger: wir sagen, dass eine Zahl 'g' eine *Primitivwurzel modulo 'n'* ist,

1. wenn jede Zahl 'a' die teilerfremd von 'n' ist,
2. auch noch *kongruent* zu 'g' hoch 'k' modulo 'n' ist.

Zur Veranschaulichung wählen wir n = 5. Zuerst müssen wir also alle Zahlen 'a' finden, die zu 5 teilerfremd sind. Das haben wir oben schon gemacht:

```
1, 2, 3, und 4.
```

Zweitens, müssen wir folgendes überprüfen: für jede von diesen Zahlen 'a' müssen wir jetzt ein 'g' finden, das zu irgendeiner willkürlichen Potenz 'k' genommen und davon modulo 'n', wieder eines dieser 'a' ergibt. Nur wenn da wieder alle 'a' vorkommen (also keins verloren gegangen ist), dann ist 'g' eine Primitivwurzel modulo 'n'. Versuchen wir es also mit der Zahl g = 2 und n = 5:

```
2 ^ 1 % 5 = 2
2 ^ 2 % 5 = 4
2 ^ 3 % 5 = 3
2 ^ 4 % 5 = 1
```

Also 2 ist auf jeden Fall eine Primitivwurzel modulo 5. Mal sehen, ob g = 4 eine ist:

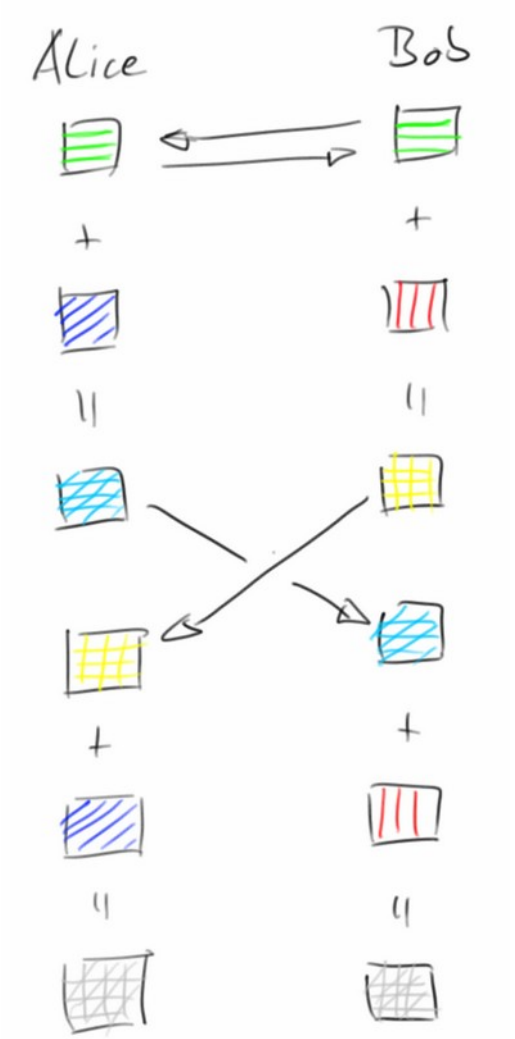
```
4 ^ 1 % 5 = 4
4 ^ 2 % 5 = 1
4 ^ 3 % 5 = 4
4 ^ 4 % 5 = 1
```

Wir sehen also, dass 4 keine Primitivwurzel modulo 5 ist. In der Wikipedia [23] finden wir die Liste der Primitivwurzel für die ersten hundert n:

```
...
n = 17: 3, 5, 6, 7, 10, 11, 12, 14
n = 19: 2, 3, 10, 13, 14, 15
n = 23: 5, 7, 10, 11, 14, 15, 17, 19, 20, 21
...
```

Aber wir könnten sie auch selbst berechnen. Jetzt sind wir alle bereit für Diffie-Hellman.

Diffie and Hellman



Der von Diffie und Hellman vorgeschlagene Algorithmus führt zu einem gemeinsamen Geheimnis, ohne dass das Geheimnis selbst ausgetauscht werden muss. Stattdessen kann man es berechnen. Dieses Geheimnis kann man dann als Schlüssel für die weitere, verschlüsselte Kommunikation verwenden.

Um zu sehen, wie der Algorithmus funktioniert, denken wir uns zwei Personen, Alice und Bob. Alice und Bob wollen geheime Nachrichten austauschen und benötigen dafür einen gemeinsamen geheimen Schlüssel.

1. Als erstes müssen sich Alice und Bob auf zwei Zahlen 'p' und 'g' einigen, mit der Einschränkung, dass 'p' eine Primzahl und 'g' eine Primitivwurzel modulo 'p' ist. Von dem, was wir gerade gelernt haben, könnten p = 17 und g = 7 gute Zahlen sein. Die Zahlen 'p' und 'g' sind öffentlich, d.h. jeder kann sie sehen.

2. Als nächstes wählen Alice und Bob jeweils eine geheime Zahl. Nehmen wir an, Alice wählt a = 3 und Bob wählt b = 5. Diese geheimen Zahlen behalten sie nur für sich und teilen sie mit niemandem.

3. Aus diesen geheimen Zahlen berechnen sie dann neue Zahlen 'A' und 'B', mit der folgenden Formel

   ```
   A = g^a % p = 7^3 % 17 = 3     und
   B = g^b % p = 7^5 % 17 = 11
   ```

   Diese beiden Zahlen teilen sie wieder miteinander. Auch hier gilt, jeder kann diese beiden Zahlen sehen.

4. Nun können beide den gemeinsamen Schlüssel berechnen: Alice benutzt ihre geheime Zahl 'a' und Bobs öffentliche Zahl 'B', während Bob seine geheime Zahl 'b' und Alices öffentliche Zahl 'A' benutzt:

   ```
   s = B^a % p = 11^3 % 17 = 5     und     s = A^b % p = 3^5 % 17 = 5
   ```

   Wir sehen, dass beide Geheimnisse 's' gleich sind. 's' ist der gemeinsame Schlüssel zwischen Alice und Bob. Für einen Lauscher ist es praktisch unmöglich, dieses Geheimnis aus den gegebenen öffentlichen Informationen zu berechnen.

Warum funktioniert das? Der Grund dafür ist, dass für die gegebene Konstruktion stets gilt:

```
(g^a % p)^b % p = (g^b % p)^a % p
```

aber nur, wenn 'p' eine Primzahl und 'g' eine Primitivwurzel modulo 'p' ist. Es gibt noch eine weitere Einschränkung: die Zahlen a, b und p sollten sehr große Zahlen sein, nur dann ist es sehr schwer, diesen Algorithmus mit roher Gewalt zu knacken.

AsymmetricEncryption

Ein Jahr nach Diffie und Hellman fanden die drei Forscher Ron Rivest, Adi Shamir and Leonard Adleman mit dem nach ihnen benannten RSA Algorithmus den ersten Algorithmus der asymmetrischen Art [24]. Asymmetrisch deshalb, weil es hier zwei Schlüssel gibt: einen zum Verschlüsseln, dem Public-Key und einen zum Entschlüsseln, dem Private-Key. Praktisch die gesamte Internetverschlüsselung beruht auf diesen beiden Verfahren, Diffie-Hellman und RSA.

Ähnlich wie Diffie-Hellman beruht der RSA Algorithmus auf der Tatsache, dass es rechnerisch sehr aufwändig ist (d.h. lange dauert) das Produkt zweier großer Primzahlen zu faktorisieren. Die Details sind nicht super kompliziert, sogar einfacher als bei Diffie-Hellman, man muss halt drauf kommen [25].

Wie verwendet man jetzt so einen asymmetrischen Verschlüsselungsalgorithmus? Erst einmal muss man wieder den Algorithmus und das Padding auswählen:

```
private final String ALGORITHM =
    "RSA"; // "ElGamal"
private final String PADDING =
    "RSA/ECB/PKCS1Padding"; // "ElGamal"
```

Weder ECB noch PKCS1Padding sind jetzt das Gelbe vom Ei, aber wenn man nichts anderes hat, nimmt man was man kriegt. Außer dem RSA Algorithmus gibt es noch den *ElGamal* Algorithmus. Soll auch nicht schlecht sein.

Als nächstes müssen wir Schlüssel erzeugen, in diesem Fall ein Paar:

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance(ALGORITHM);
kpg.initialize(2048, new SecureRandom());
KeyPair pair = kpg.generateKeyPair();
```

Das KeyPair besteht aus zwei Schlüsseln, dem *public* und dem *private* Key:

```
PublicKey publicKey = keyPair.getPublic();
PrivateKey privateKey = keyPair.getPrivate();
```

Der public Key, darf und soll öffentlich sein, also jeder kann ihn sehen. Wenn also Alice an Bob eine Nachricht schicken möchte, dann benötigt sie Bob's public Key. Damit verschlüsselt sie ihre Nachricht an Bob:

```
Cipher cipher = Cipher.getInstance(PADDING);
cipher.init(Cipher.ENCRYPT_MODE, publicKey);
byte[] encrypted = cipher.doFinal(msg.getBytes("UTF-8"));
```

Bob erhält die *encrypted* Nachricht von Alice, und entschlüsselt sie mit seinem private Key:

```
Cipher cipher = Cipher.getInstance(PADDING);
cipher.init(Cipher.DECRYPT_MODE, privateKey);
byte[] decrypted = cipher.doFinal(encrypted);
String msg = new String(decrypted, "UTF-8");
```

Den private Key darf niemand außer Bob sehen. Denn nur mit ihm kann man Nachrichten entschlüsseln.

Also, mit dem public Key kann man nur Verschlüsseln und mit dem private Key nur Entschlüsseln. Wie schickt jetzt aber Bob Nachrichten an Alice? Dafür muss Alice ihr eigenes Schlüsselpaar erzeugen, und davon ihren public Key an Bob schicken. D.h. bevor wir an jemanden verschlüsselte Nachrichten schicken können müssen wir den public Key der Person haben mit der wir kommunizieren wollen. Im Internet gibt es dafür spezielle Server [26], wobei die Frage ist in wieweit man denen wirklich trauen kann. Denn wenn ich die NSA wäre, würde ich so einen Keyserver hosten...

DigitalSignature

Ein interessantes Nebenprodukt der asymmetrischen Verschlüsselung ist die digitale Signatur [27]. Mit einer digitalen Signatur kann man drei Dinge erreichen:

- Authentication: man kann nachweisen wer eine Nachricht gesendet hat,
- Non-Repudiation: der Autor kann nicht leugnen Urheber der Nachricht zu sein und
- Integrity: die Nachricht wurde nicht verändert.

Die Integrität einer Nachricht kann man auch mit einer Cryptohashfunktion (wie SHA-256) nachweisen, die ersteren beiden aber nicht. Das geht nur mit der digitale Signatur.

Es gibt zwei Algorithmen mit denen man digitale Signaturen erstellen kann, DSA [28] und RSA. Wir beginnen wie üblich mit der Auswahl von Algorithmus und Padding:

```
private final String ALGORITHM = "DSA";
private final String PADDING = "SHA1withDSA";
```

Als nächstes benötigen wir wieder ein KeyPair:

```
KeyPairGenerator kpg =
    KeyPairGenerator.getInstance(ALGORITHM);
kpg.initialize(1024, new SecureRandom());
KeyPair pair = kpg.generateKeyPair();
```

Auch hier gilt wieder: der public Key ist öffentlich, also für jeden einsehbar, der private Key sollte nie in fremde Hände gelangen.

Anstelle zu Verschlüsseln, *signieren* wir die Daten um die es geht, allerdings mit unserem private Key:

```
Signature dsaSign = Signature.getInstance(PADDING);
dsaSign.initSign(privateKey);
dsaSign.update(msg.getBytes());
byte[] signature = dsaSign.sign();
```

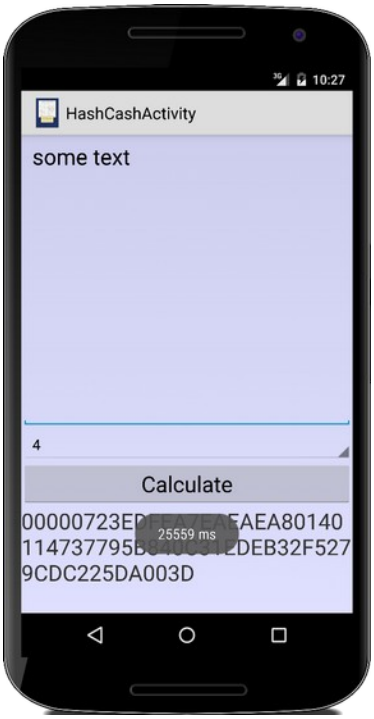
Die Signatur hat ähnlich wie ein Cryptohash eine feste Länge, und auch ansonsten hat sie die gleichen Eigenschaften, die wir auch schon bei den Cryptohashes geschätzt haben.

Wie verifiziert man nun eine Signatur? Zunächst braucht man die Originaldaten, also die *msg*. Außerdem benötigt man den public Key. Der ist ja öffentlich zugänglich, also kann jeder die Signatur verifizieren:

```
Signature dsaSign = Signature.getInstance(PADDING);
dsaSign.initVerify(publicKey);
dsaSign.update(msg.getBytes());
boolean verifies = dsaSign.verify(signature);
```

Wenn da *true* rauskommt, dann passt alles. Kommt aber *false* raus, hat entweder jemand anderes die Daten erzeugt oder die Daten sind verändert worden. D.h. also beim digitalen Signaturverfahren werden die Daten nicht verschlüsselt. Die Daten sind für jeden einsehbar. Alles was wir mit der Signatur sagen ist, dass wir die Daten fabriziert haben, und dass die Daten von niemandem verändert wurden.

HashCash



Wir alle werden ständig mit Spam-Mails bombardiert. Wie könnte man solche E-Mails verhindern? Wenn sie etwas kosten würden. Das würde das Problem lösen, denn für die Spammer wäre es einfach zu teuer. Aber niemand will wirklich für das Versenden von E-Mails bezahlen. Aber was, wenn es nur ein bisschen Rechenpower kostet? Für den normalen Nutzer wäre es kein Problem zehn Sekunden zu warten, um eine E-Mail zu versenden. Aber für Spammer wäre das ein Problem. Denn wenn man eine Millionen E-Mails versenden will, würde das 115 Jahre dauern.

Was hat das jetzt mit Kryptographie zu tun? Nun das Berechnen eines Cryptohashes (z.B. eines SHA-256) dauert ungefähr eine halbe Millisekunde. Das ist die erste Eigenschaft die sich HashCash zunutze macht.

Die zweite ist, wenn wir einen Buchstaben zu unserer Email hinzufügen, dann ändert sich der Cryptohashwert. Wir können jetzt solange einen Buchstaben hinzufügen, bis der Hashwert mit der Ziffer '0' beginnt. Im Schnitt ist dazu die Berechnung von acht Cryptohashes notwendig. Also sind wir schon bei vier Millisekunden.

Wir können auch verlangen, dass die ersten beiden Ziffern Null sein sollen. Die Berechnung wird dann noch länger dauern. Je mehr Nullen wir wollen, desto länger dauert es. Das ist was gemeinhin unter dem Begriff "Proof-of-Work" [30] bekannt ist. Auf aktueller Hardware dauert es etwa zehn Sekunden, einen Cryptohash zu finden, der mit fünf Nullen beginnt, und genau das macht Hashcash [29]. Diese Idee des Proof-of-Work ist auch für Cryptocoins wie Bitcoin relevant [31].

SSL, HTTPS: Mixed Encryption

Zum Schluss sie noch angemerkt, dass asymmetrische Verschlüsselung nicht ganz so perfekt ist wie sie zu sein scheint. Zum einen ist sie viel langsamer als symmetrische Verschlüsselung. Des weiteren stellt sich das Management der Schlüssel als etwas komplizierter heraus. Z.B. wenn ich eine Nachricht an mehrere Personen verschicken möchte, dann benötige ich den öffentlichen Schlüssel von jeder Person an der die Nachricht gehen soll. Und asymmetrische Verfahren sind anfällig für kryptographische Angriffe, insbesondere bei langen Nachrichten.

Deswegen verwendet man in der Praxis (z.B. ssl, ssh, https) gemischte Verfahren. Bei der gemischten Verschlüsselung verwenden wir zunächst eine asymmetrische Verschlüsselung, tauschen dann einen privaten Schlüssel aus und kommunizieren danach nur noch symmetrisch über den privaten Schlüssel. Damit haben wir das beste aus beiden Welten.

Research

Zum Thema Kryptografie gibt es wieder ganz viel zu recherchieren.

Concepts

Im Zusammenhang mit Verschlüsselung tauchen einige Begriffe immer wieder auf:

- Authentifizierung (authentication)
- Authentizität (authenticity)
- Verschlüsselung (encryption)
- Integrität einer Nachricht (integrity of a message)
- Vertraulichkeit (confidentiality)
- Unleugbarkeit (non-repudiation)

Falls Ihnen die Begriffe nicht geläufig sind, sollten Sie die vielleicht mal kurz bei Wikipedia nachschlagen.

Keystore

Ein riesen Problem ist das Speichern von Schlüsseln. Dazu gibt es unter Android etwas das man den *Keystore* nennt. Prinzipiell eine gute Idee, aber in Wirklichkeit ziemlich nutzlos. Lesen Sie dazu die folgenden drei Posts:

- https://developer.android.com/training/articles/keystore.html
- https://nelenkov.blogspot.de/2012/05/storing-application-secrets-in-androids.html
- https://nelenkov.blogspot.de/2012/04/using-password-based-encryption-on.html

Zertifikate

Wie weiß man wem man trauen kann? Im Internet wird das durch Zertifikate geregelt. Dazu macht es Sinn so zum Einstieg sich mal den Artikel dazu in der Wikipedia durchzulesen [33].

Man-in-the-Middle

Oben bei Asymmetrischer Verschlüsselung habe ich angedeutet, dass es Sinn machen würde, wenn die NSA Keyserver hosten würde. Um das zu verstehen sollte man mal nachlesen was unter einer Man-in-the-Middle Attacke [32] zu verstehen ist.

Fragen

1. Wofür verwendet man normalerweise einen MessageDigest wie den MD5 oder SHA-1?

2. Was ist ein MD5?

3. Wie könnten Sie die Caesar-Verschlüsselung knacken?

4. Im Zusammenhang mit Verschlüsselungsverfahren ist es immer wichtig einen guten Zufallszahlengenerator zu haben. Warum?

5. Was ist das Hauptproblem symmetrischer Verschlüsselungsverfahren?

6. Warum könnte die NSA Interesse haben einen Keyserver zu hosten?

7. Warum sollte man keine asymmetrische Verschlüsselung für sehr lange Nachrichten verwenden?

Referenzen

[1] Skytale, https://de.wikipedia.org/wiki/Skytale

[2] Caesar cipher, https://en.wikipedia.org/wiki/Caesar_cipher

[3] Online Cryptography Course, Dan Boneh, Stanford University, https://crypto.stanford.edu/~dabo/courses/OnlineCrypto/

[4] Base64, https://en.wikipedia.org/wiki/Base64

[5] Cyclic redundancy check, https://en.wikipedia.org/wiki/Cyclic_redundancy_check

[6] Adler-32, https://en.wikipedia.org/wiki/Adler-32

[7] IPv4, https://en.wikipedia.org/wiki/IPv4#Header_Checksum

[8] Cryptographic hash function, https://en.wikipedia.org/wiki/Cryptographic_hash_function

[9] SHA-2, https://en.wikipedia.org/wiki/SHA-2

[10] bcrypt, https://en.wikipedia.org/wiki/Bcrypt

[11] Twitter ruft nach Sicherheitspanne zum Passwortwechsel auf, https://www.heise.de/newsticker/meldung/Twitter-ruft-nach-Sicherheitspanne-zum-Passwortwechsel-auf-4041199.html

[12] Enigma machine, https://en.wikipedia.org/wiki/Enigma_machine

[13] One-time pad, https://en.wikipedia.org/wiki/One-time_pad

[14] Padding (cryptography), https://en.wikipedia.org/wiki/Padding_(cryptography)

[15] Block cipher mode of operation, https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#Common_modes

[16] Online Cryptography Course, Dan Boneh, Stanford University, https://crypto.stanford.edu/~dabo/courses/OnlineCrypto/

[17] Jonathan Katz Coursera: Cryptography, https://www.coursera.org/course/cryptography

[18] Bouncy Castle (cryptography), https://en.wikipedia.org/wiki/Bouncy_Castle_(cryptography)

[19] Diffie-Hellman key exchange, https://en.wikipedia.org/wiki/Diffie–Hellman_key_exchange

[20] Prime number, https://en.wikipedia.org/wiki/Prime_number

[21] Coprime integers, https://en.wikipedia.org/wiki/Coprime_integers

[22] Congruence relation, https://en.wikipedia.org/wiki/Congruence_relation

[23] Primitive root modulo n, https://en.wikipedia.org/wiki/Primitive_root_modulo_n

[24] Public-key cryptography, https://en.wikipedia.org/wiki/Public-key_cryptography

[25] RSA (cryptosystem), https://en.wikipedia.org/wiki/RSA_(cryptosystem)

[26] Key server (cryptographic), https://en.wikipedia.org/wiki/Key_server_(cryptographic)

[27] Digital signature, https://en.wikipedia.org/wiki/Digital_signature

[28] Digital Signature Algorithm, https://en.wikipedia.org/wiki/Digital_Signature_Algorithm

[29] Hashcash, https://en.wikipedia.org/wiki/Hashcash

[30] Proof-of-work system, https://en.wikipedia.org/wiki/Proof-of-work_system

[31] Bitcoin, https://en.wikipedia.org/wiki/Bitcoin

[32] Man-in-the-middle attack, https://en.wikipedia.org/wiki/Man-in-the-middle_attack

[33] Public key certificate, https://en.wikipedia.org/wiki/Public_key_certificate

Epilogue

Hat länger gedauert als gedacht.

Alphabetical Index

Projekte