Skript zur Veranstaltung Android Apps

Sommeruni 2018

Toni Draßdo, Alexander Korzec, Nicolas Lehmann,
Ferdous Nasri, Sönke Schmidt

13. August 2018

---

1 Vor dem Workshop

Wir freuen uns darüber, dass du dieses Skript liest und Interesse an unserem Workshop zeigst. Wir möchten dir in einer *offenen und entspannten Atmosphäre* das kleine 1×1 der App-Entwicklung beibringen und eine kleine Starthilfe für deine eigenen Ideen anbieten. Wir möchten dich etwas an die Hand nehmen und mit dir im großen Dschungel der vielfältigen Möglichkeiten von Android auf eine kleine Entdeckungsreise gehen. Wie weit du gehen möchtest und ob diese Reise schon am Ende dieses Absatzes endet, sei dir überlassen. Wir freuen uns auf dein Kommen und laden dich auf jeden Fall herzlich ein!

Falls du die Möglichkeit hast, dann möchten wir dich bitten einen Laptop mit *installiertem Android Studio* mitzunehmen. Unsere Kapazitäten in den Poolräumen sind leider beschränkt und außerdem kannst du in diesem Fall problemlos zu Hause an deinen Apps weiterarbeiten. Android Studio findest du unter dem folgenden Link:

https://developer.android.com/studio/index.html.

Die Installation von Android Studio ist (zumindest unter Windows) einfach und intuitiv. Während des Installationsvorgangs musst du theoretisch nur auf „Weiter“ klicken und viel Geduld haben.

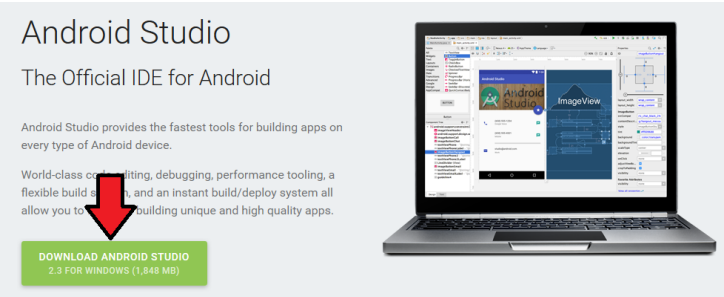


Abbildung 1: Download von Android Studio

Solltest du während der Installation gefragt werden, ob du *HAXM*[1] installieren möchtest, dann kannst du auf „Ja“ klicken. Mit *HAXM* hast du durch Hardwarebeschleunigung bei der Emulation von *Android-ROM*s auf Basis der *x86-Architektur* eine bessere Performance. Auf diese Weise kann man seine Apps bequem testen, ohne sie gleich auf ein physisches Gerät übertragen zu müssen. *HAXM* ist nur auf leistungsstarke Rechner mit einem Intel-Prozessor installierbar.

Falls du *HAXM* nicht installieren kannst, dann steht dir die Möglichkeit offen unter Android Studio ein ganzes Smartphone komplett *softwareseitig* zu emulieren. Wir empfehlen diese Möglichkeit nur, wenn du über einen leistungsstarken Rechner verfügst. Die andere Möglichkeit ist, dass du ein Smartphone/Tablet zum testen deiner Apps verwendest. Einerseits musst du *USB-Debugging* in deinem Gerät aktivieren. Eine Anleitung, wie man *USB-Debugging* aktiviert, findest du unter dem folgenden Link:

`https://mobilsicher.de/schritt-fuer-schritt/usb-debugging-aktivieren`.

Andererseits musst du die *Installation aus unbekannten Quellen* erlauben. Wie du diese Option aktivierst, findest du unter dem Link weiter unten.

`https://www.droidwiki.org/wiki/Unbekannte_Quellen`

Solltest du die Möglichkeit einen Laptop mitzunehmen nicht haben, dann ist das nicht schlimm! Komme einfach vorbei und wir versuchen dich an einen Poolrechner zu setzen.

Wenn du Fragen oder Anmerkungen zu unseren Veranstaltungen oder zu diesem Skript hast, dann kannst du uns darauf (auch persönlich) ansprechen. Wir sind unter der E-Mail-Adresse

`mentoring@mi.fu-berlin.de`

erreichbar. Für alle deine anderen Anliegen gilt natürlich der gleiche Kontaktweg.

2 Der erste Tag

Wir freuen uns, dass du dich entschieden hast, mitzumachen. Am ersten Tag möchten wir dich mit der Entwicklungsumgebung *Android Studio* vertraut machen und eine erste lauffähige App erstellen. Unser Konzept basiert auf „learning by doing“ und wir möchten dir keine langen Tafelvorträge zumuten. Das bedeutet auch, dass wir dir in *Java* nur die nötigsten Sachen zeigen und keine komplette *AlPII*-Veranstaltung zusammenfassen werden.

Gehörst du zu der Gruppe von Leuten, die wenig oder gar keine Java-Kenntnisse haben, dann kannst du auf der folgenden Seite

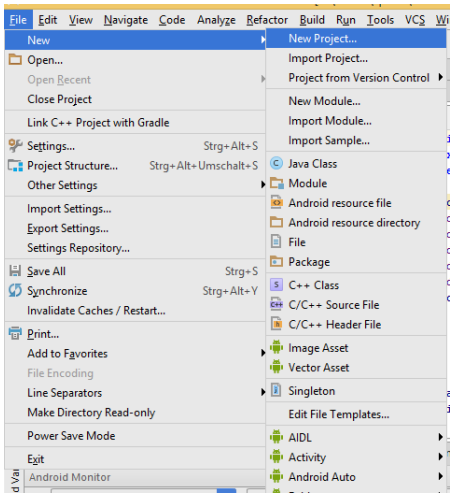
`http://openbook.rheinwerk-verlag.de/javainsel/`.

die Grundlagen erarbeiten, falls du nach dem ersten Tag mehr über Java wissen möchtest.

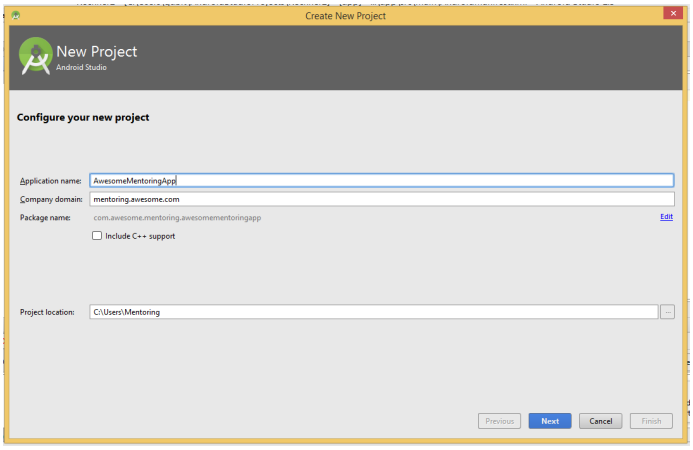
Solltest du stattdessen zu der Gruppe der `l337 h4x0rz` gehören, dann werden unsere Grundlagen für dich mit hoher Wahrscheinlichkeit nicht besonders neu sein. Wir möchten dich um etwas Geduld bitten. Währenddessen kannst du unser Projekt ausbauen oder dir gar ein komplett neues Projekt ausdenken und umsetzen. Wir werden dir mit Rat zur Seite stehen.

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

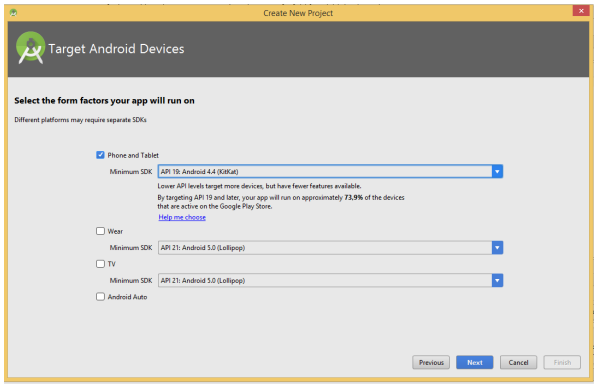
2.1 Erstellung eines neuen Projekts in Android Studio



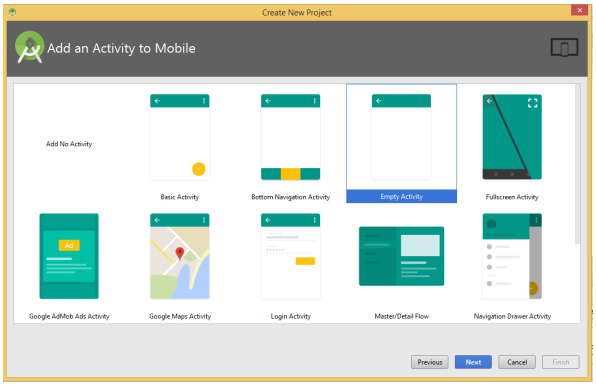
Schritt 1: Unter *File* → *New* auf *New Projects...* klicken.



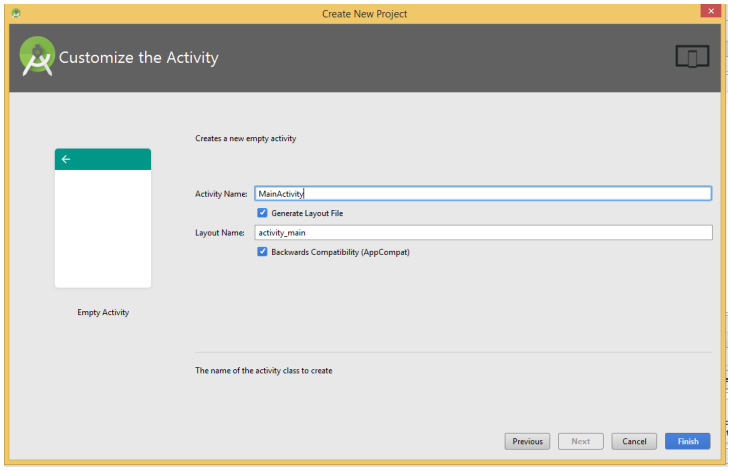
Schritt 2: Benenne deine Applikation passend und klicke auf *Next*.



Schritt 3: Wähle als *Minimum SDK* die *API* 16 (Jelly Bean) aus und klicke auf *Next*.



Schritt 4: Wähle *Empty Activity* aus und klicke auf *Next*.



Schritt 5: Hier klickst du einfach auf *Finish*.

2.2 Layout ändern

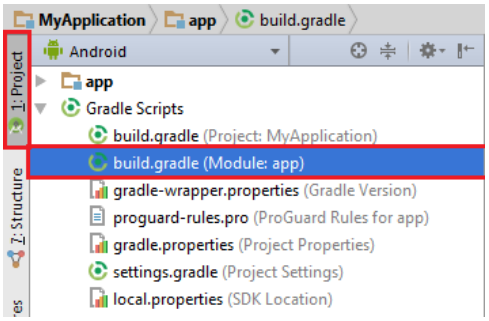
Als wir unseren Taschenrechner programmiert haben, hatten wir Probleme mit der Positionierung der einzelnen Textfelder und Knöpfe. Damit du nicht die gleichen Probleme erlebst, möchten wir dich bitten das *Layout* zu ändern. Wie das genau geht, siehst du in der folgenden kurzen Anleitung:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.example.avoid.myapplication2.MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</android.support.constraint.ConstraintLayout>
```

Schritt 1: Öffne in Android Studio die Datei `activity_main.xml` und finde die rot markierten Bereiche in der *XML-Datei*.

```
<?xml version="1.0" encoding="utf-8"?>
<AbsoluteLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.example.avoid.myapplication2.MainActivity">

</AbsoluteLayout>
```

Schritt 2: Lösche den Code im mittleren roten Kasten im vorherigen Bild vom `TextView`. Ändere die verbleibenden rot markierten Bereiche aus dem vorherigen Bild, so dass sie aussehen, wie die gelben Bereiche in diesem Bild. Wir verwenden das `AbsoluteLayout`. Zwar ist es *deprecated*, aber für den Einstieg reicht es.



Schritt 3: Gehe unter *1: Project* und öffne unter *Gradle Scripts* das Modul im roten Kasten.

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    androidTestCompile('com.android.support.test.espresso:espresso-core:2.2.2', {
        exclude group: 'com.android.support', module: 'support-annotations'
    })
    compile 'com.android.support:appcompat-v7:25.3.1'
    compile 'com.android.support.constraint:constraint-layout:+'
    testCompile 'junit:junit:4.12'
}
```

Schritt 4: Kommentiere die Zeile im roten Kasten mit // aus.

2.3 Kleiner Einblick in die Welt der objektorientierten Programmierung

In unserem Workshop werden wir mit der Programmiersprache *Java* arbeiten. Java ist eine *objektorientierte* Programmiersprache. Stark vereinfacht sind die grundlegenden Begriffe, denen man bei objektorientierten Sprachen immer wieder begegnet, die Folgenden:

- Klasse
- Objekt
- Attribut
- Attributwert
- Methode

Das ausgeführte Programm ist in der Welt der Objektorientierung eine Ansammlung von „konkreten Gegenständen“, welche miteinander interagieren können. Diese werden *Objekte* genannt. Objekte lassen sich durch ihre „Eigenschaften“ und „Fähigkeiten“ beschreiben. Die „Eigenschaften“ nennt man *Attribute* und die „Fähigkeiten“ bezeichnet man als *Methoden*.

Wie beschreiben wir nun als Programmierer Objekte? In diesem Fall müssen wir sogenannte *Klassen* definieren. Klassen sind „Baupläne“, aus denen sich Objekte erzeugen lassen. Das heißt, Klassen sind erst mal *statische* Blaupausen, im Gegensatz zu den sich *dynamisch* verhaltenden Objekten.

Stellen wir uns mal die aufgezeichnete Welt auf eine „spielerische“ Art vor.

Die Objekte, welche wir in Abbildung 3 sehen, sind ein `Eismonster` und ein `Bossmonster`. Sowohl das `Eismonster`, als auch das `Bossmonster` lassen sich mit Eigenschaften, also Attributen, beschreiben. Eine gemeinsame Eigenschaft sind die `Lebenspunkte`. Die Anzahl an `Lebenspunkten` ist jedoch nicht bei beiden gleich. Die konkrete Ausprägung von einem Attribut wird übrigens als *Attributwert* bezeichnet.

Ein `Eismonster` und ein `Bossmonster` haben beide Verhaltensweisen, mit denen sie mit ihrer Umgebung interagieren können. In diesen Verhaltensweisen erkennen wir die *Methoden* wieder. Beide Kämpfer verfügen im Beispiel über die Methode `Angriff` und können sich somit gegenseitig angreifen.

Auf der linken Seite in Abbildung 3, sehen wir die *Klasse* `Basismonster`. Wie hängt nun das `Basismonster` mit dem `Eismonster` und dem `Bossmonster` zusammen? Wir können uns vorstellen, dass das `Eismonster` und das `Bossmonster` konkrete *Exemplare* von der „Blaupause“ `Basismonster` sind.

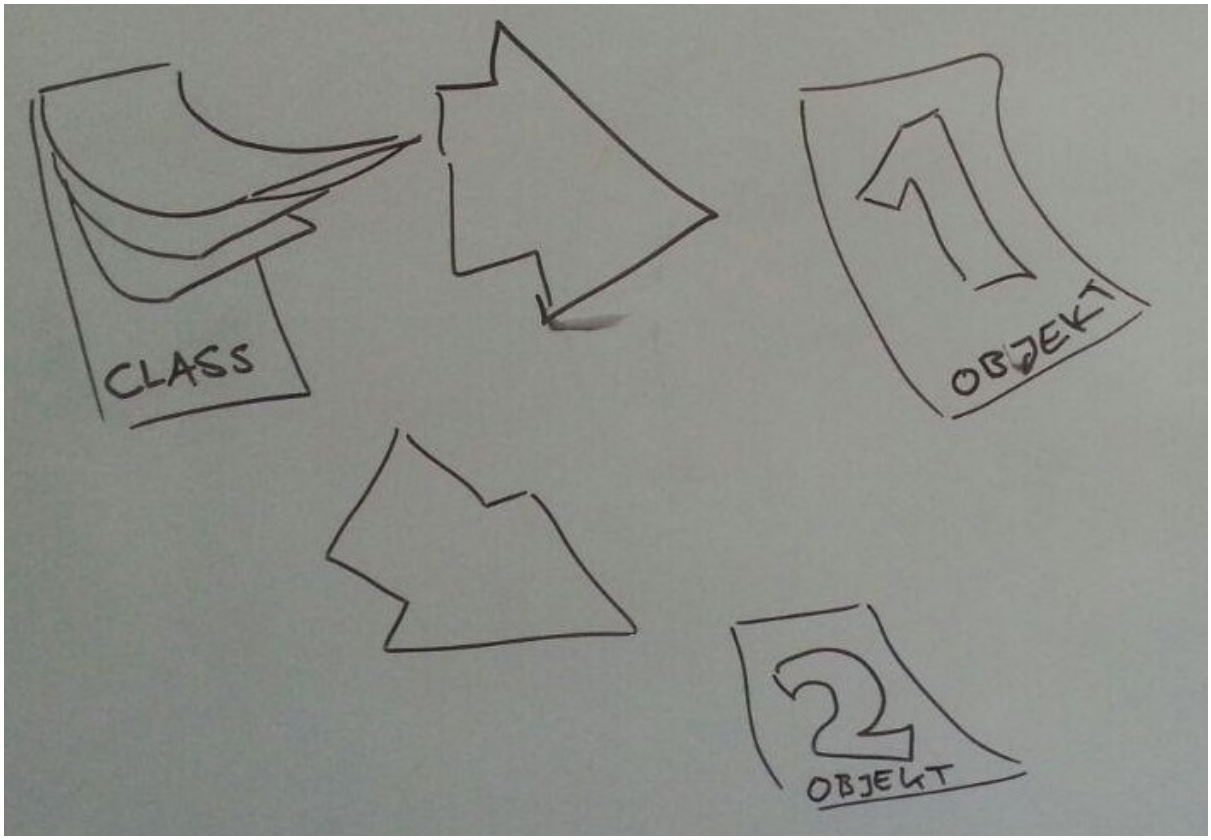


Abbildung 2: Die abgerissenen Notizzettel als Abbild vom Notizblock.

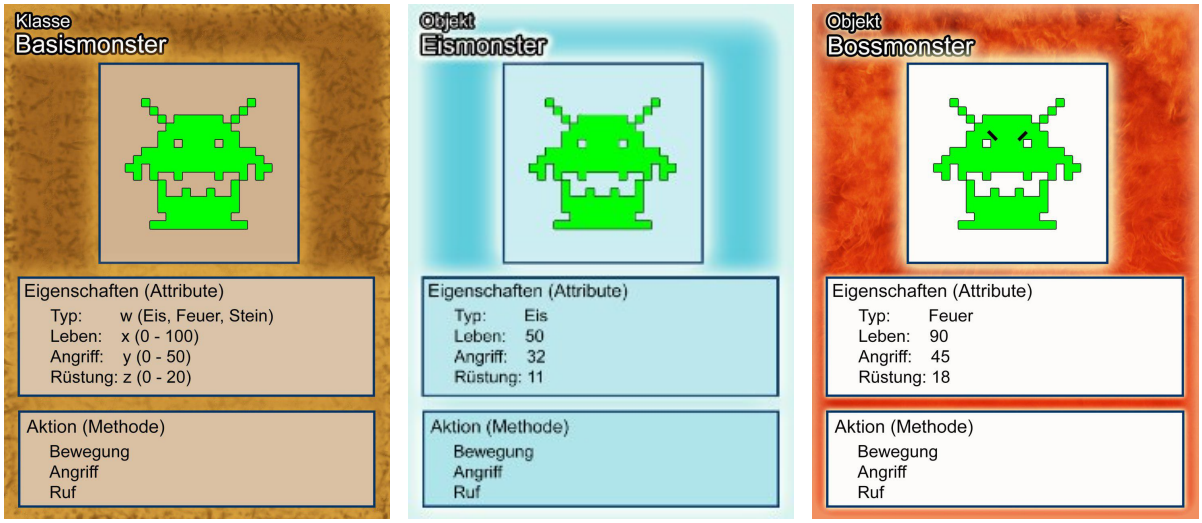
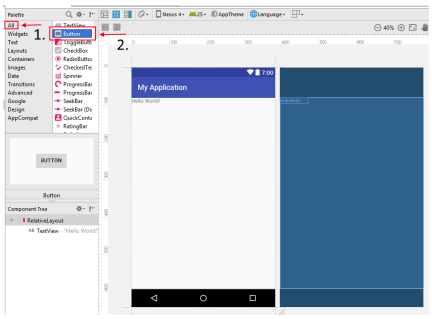


Abbildung 3: Beispiele für Objekte und Klassen.

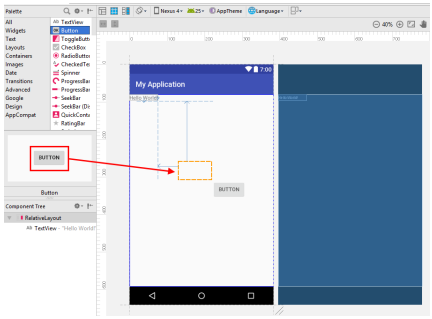
Wozu ist eine solche Situation praktisch? Wir können uns vorstellen, dass das `Eismonster` mit einem Angriffswert von 32 einen geringeren Schaden verursacht, als das `Bossmonster` mit einem Angriffswert von 45. Wir können uns jedoch auch vorstellen, dass wir für die Kalkulation vom Schaden bei einem Angriff in beiden Fällen den gleichen Schadensalgorithmus verwenden möchten, bloß mit anderen Parametern. Den Schadensalgorithmus müssen wir als Programmierer in der Klasse `Basismonster` spezifizieren. Dadurch nutzen die beiden Objekte `Eismonster` und `Bossmonster` den gleichen Schadensalgorithmus und brauchen lediglich ihre *Attributwerte* als Parameter einsetzen.

2.4 Wie klicke ich mir einen funktionierenden Knopf zusammen?

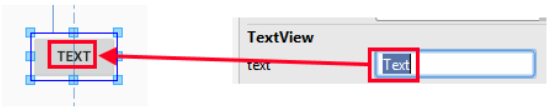
Wir zeigen dir den Umgang mit Android Studio, indem wir einen funktionsfähigen `Button` erstellen.



Schritt 1: Wähle bei der *Palette* einen `Button` aus.



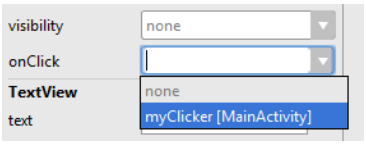
Schritt 2: Ziehe einen `Button` mittels „*drag & drop*“ auf die Oberfläche.



Schritt 3: Gib deinem `Button` einen schicken Namen.

```
public void myClicker (View v) {

    EditText zahl1 = (EditText) findViewById(R.id.editText);
    EditText zahl2 = (EditText) findViewById(R.id.editText2);
    EditText ergebnis = (EditText) findViewById(R.id.editText3);

    String s1 = zahl1.getText().toString();
    String s2 = zahl2.getText().toString();

    Integer i1 = Integer.parseInt(s1);
    Integer i2 = Integer.parseInt(s2);

    Integer rechnung = i1 + i2;

    String loesung = String.valueOf(rechnung);

    ergebnis.setText(loesung);
}
```

Schritt 4: Ziehe auf die gleiche Weise mehrere `EditText` Objekte auf die Oberfläche und schreibe eine `myClicker()` Methode, die so ähnlich aussieht, wie oben im Bild.



Schritt 5: Wähle beim `Button` bei der Eigenschaft *onClick* die Methode `myClicker()` aus.

3 Der zweite Tag

Neues Projekt, neues Glück! Bevor du voller Tatendrang an die Arbeit gehst, möchten wir dir zunächst einige fortgeschrittene Aspekte von Android zeigen.

Zuerst möchten wir dir etwas über `Activities` erzählen. Eine `Activity` stellt im Allgemeinen eine Bildschirmseite dar und eine App lässt sich ganz grob als eine Menge von `Activities` auffassen. Daher sind `Activities` ein wichtiges Grundkonzept von Android und unserer Ansicht nach lohnt es sich, wenigstens von deren Existenz zu wissen.

Später möchten wir dir etwas Input zur *Sensorik* geben. Die Vielfalt der Sensoren am Smartphone, und die sich dadurch potenziell ergebenden Nutzerinteraktionen, lassen kaum Wünsche für die Realisierung deiner Ideen offen.

Hast du dir seit dem letzten Mal überlegt, was du heute tun möchtest? Solltest du schon eine Idee haben, dann ist das Klasse! Nachdem wir eine kleine Einführung zur Sensorik gegeben haben, kannst du gerne anfangen an deinem Projekt zu arbeiten. Falls du keine Idee hast, dann werden wir dir einige Projekte vorschlagen.

3.1 Noch eine kurze Vorbemerkung

Wir vermuten, dass du in den nächsten beiden Tagen mit Sachen arbeiten wirst, welche sogar uns unbekannt sind. In diesem Fall werden wir mit dir gemeinsam nach einer Lösung suchen. Wir möchten dir außerdem den folgenden Link ans Herz legen:

`https://developer.android.com/guide/index.html`.

Diese Seite enthält gut geschriebene, praktische *API Guides* und wir werden ebenfalls, bei eventuellen Problemen, dort Hilfe suchen. Da zum Programmieren einfach das Lesen von Dokumentationen gehört, möchten wir dich darum bitten, bei Sachen, welche wir nicht im Skript erläutern, dich zunächst eigenständig im Internet nach Lösungen zu erkunden.

3.2 Über `Activities` und `Intents`

Wir haben schon erwähnt, dass eine App im Grunde als eine lose Ansammlung von `Activities` gesehen werden kann. Sicher hast du schon bemerkt, dass in der Datei `MainActivity.java` von deinem Taschenrechner „mysteriöse“ Methoden namens `onCreate()` usw. stehen. Dieses Geheimnis wollen wir immerhin ansatzweise lüften.

Wahrscheinlich ist dir bekannt, dass in einem gewöhnlichem Java-Programm die `main()` Methode als einziger *Einstiegspunkt* dient. Das heißt, dass in der gegebenen Situation immer die Methode `main()` als Erstes aufgerufen wird. Im Gegensatz dazu gibt es bei einer Android-App in der Regel mehrere Einstiegspunkte.

Hier mal ein einfaches Beispiel, um es sich klar zu machen. Angenommen wir haben auf einem Smartphone eine gewöhnliche Kontakte-App. Einerseits kannst du die Kontakte-App starten, indem du in deinem Launcher auf das Icon dieser App klickst. Andererseits kannst du deine Kontakte-App auch von einer anderen App aus starten. Man kann sich zum Beispiel vorstellen,

dass man von einer Messenger-App aus mittels bestimmter Aktionen die Kontakte-App aufruft, welche die Telefonnummer einer konkreten Person anzeigt. Die Oberflächen, welche beim ersten Start der Kontakte-App angezeigt werden, sind nicht identisch. In der ersten Situation haben wir den Startbildschirm der Kontakte-App erzeugt und in der zweiten Situation war unser Einstiegspunkt gleich ein Bildschirm mit einem konkreten Kontakt.

Jede dieser Oberflächen in unserer Kontakte-App stellt eine `Activity` dar. `Activities` können andere `Activities` über `Intents` aufrufen. `Intents` stellen abstrakte Beschreibungen von Operationen dar, die eine `Activity` einer anderen `Activity` übermitteln kann. Zum Beispiel könnte die Kontakte-App über einen `Intent` eine Kamera-App anweisen ein Bild über die Frontkamera aufzunehmen (Selfie :D).

Darüber hinaus befindet sich eine `Activity` während ihrer Lebensdauer in einem gewissen Zustand. Zum Beispiel kann eine `Activity` gerade auf dem Bildschirm angezeigt oder pausiert und in den Hintergrund verlagert werden.

3.2.1 Lebenszyklus einer `Activity`

Sobald eine `Activity` ihren Zustand ändert, wird mindestens eine der folgenden Methoden aufgerufen:

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

Bei unserem Taschenrechner haben wir nur die Methode `onCreate()` wirklich verwendet. Eventuell ist es nützlich weitere Methoden mit Leben zu füllen. Zum Beispiel kannst du eine Methode `onDestroy()` definieren in der deine `Activity` letzte Sicherungen macht, bevor sie ins Gras beißt.

Falls du mehr über *App Components*, wie zum Beispiel `Activities`, wissen möchtest, dann kannst du dich auf der Seite

`https://developer.android.com/guide/components/index.html`

informieren. In Abbildung 4 sieht man die möglichen Zustände einer `Activity` und welche Methoden bei welchen *Übergängen* im „Zustandsdiagramm“ aufgerufen werden.

3.3 Sensorik

In Android werden alle Sensoren unter einem *Systemdienst* namens `SensorManager` verwaltet. Auf diesen Systemdienst können wir mithilfe der Methode `getSystemService()` von `Context` mit dem Argument `Context.SYSTEM_SERVICE` zugreifen.

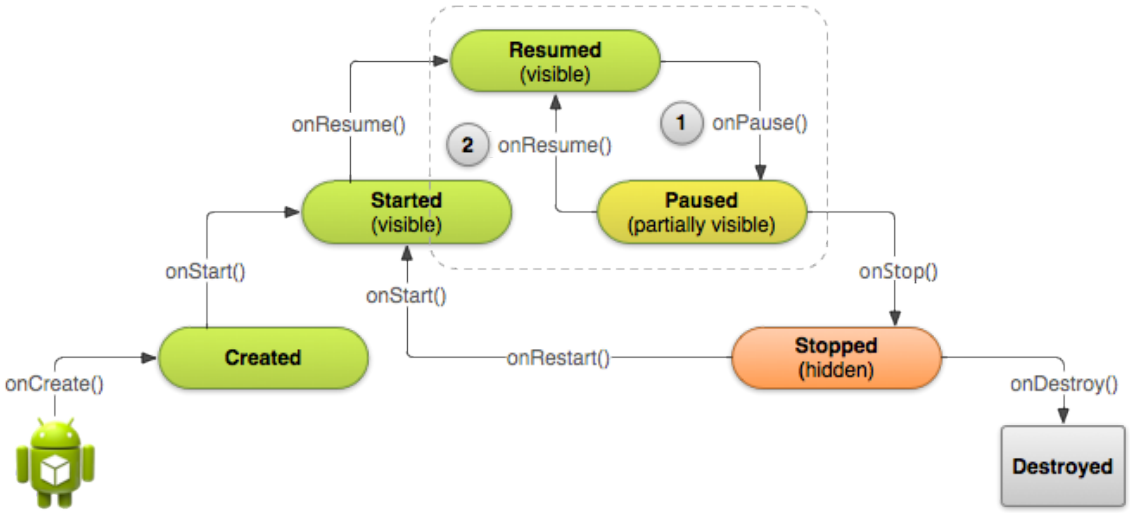


Abbildung 4: Der *Lebenszyklus* einer `Activity` als „Zustandsdiagramm“, Quelle: https://en.wikipedia.org/wiki/File:Android_application_life_cycle.png

```
SensorManager mngr =
    (SensorManager) context.getSystemService(Context.SYSTEM_SERVICE);
```

Weiter oben siehst du die Initialisierung eines *Objekts* vom Typ `SensorManager`. Nun musst du aus dem `SensorManager` einen spezifischen `Sensor` auswählen. Möchtest du zum Beispiel Werte aus dem *Beschleunigungssensor* auslesen, dann musst du auf `mngr` die Methode `getDefaultSensor()` mit dem Argument `Sensor.TYPE_ACCELEROMETER` aufrufen.

```
Sensor snr = mngr.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
```

Zwar haben wir nun einen konkreten Sensor, jedoch ist dieser erstmal in unserem Programm gar nicht aktiviert. Wir müssen nun einen `SensorEventListener` für unseren Sensor `snr` schreiben. Dafür definierten wir uns zunächst ein Objekt vom Typ `SensorEventListener`.

```
SensorEventListener listener = new SensorEventListener() {

    @Override
    public void onSensorChanged(SensorEvent event) {}

    @Override
    public void onAccuracyChanged(Sensor sensor, int accuracy) {}

}
```

Interessant ist für uns die Methode `onSensorChanged()`. Diese Methode bekommt ein Objekt `event` vom Typ `SensorEvent`. In Android gibt ein Objekt von diesem Typ eine Änderung der Messwerte von einem Sensor an. Zum Beispiel, wenn wir unser Smartphone kippen, wird gerade in diesem Moment ein `SensorEvent` mit den entsprechenden Messwerten erzeugt und an die Methode `onSensorChanged()` übergeben. Das heißt, dass du gerade dort deinen Quellcode schreiben sollst, wenn du auf die eben beschriebene externe Einwirkung reagieren möchtest.

Zum Schluss registrieren wir `listener` und `snr` bei `mngr`. Dafür müssen wir dem `SensorManager` `mngr` mittels der Methode `registerListener()` Bescheid geben.

```
mngr.registerListener(listener, snr, SensorManager.SENSOR_DELAY_UI);
```

Mit dem letzten *Argument* gibst du die *Datenrate* an, mit der die Daten von deinem Sensor ankommen sollen. Hier siehst du eine Auflistung einiger Einstellungen, in aufsteigender Sortierung, bezüglich der Datenrate:

- `SENSOR_DELAY_NORMAL`
- `SENSOR_DELAY_UI`
- `SENSOR_DELAY_GAME`
- `SENSOR_DELAY_FASTEST`

Während die Einstellung `SENSOR_DELAY_UI` für gewöhnliche UI-Interaktionen ausreicht, bietet sich für Spiele die Einstellung `SENSOR_DELAY_GAME` an.

Für eine Auflistung von Sensoren wirf mal einen Blick hier rein:

https://developer.android.com/guide/topics/sensors/sensors_overview.html

3.3.1 Beschleunigungssensor - Implementierung

Zwar wissen wir nun, wie wir einen Sensor aktivieren können, aber noch nicht, wie man mit einem Sensor umgeht. Weiter unten haben wir ein minimales Beispiel für den Beschleunigungssensor konstruiert. Dabei ist `tv` ein Objekt vom Typ `TextView`, welches beim Kippen vom Smartphone seine Position in Richtung der x-Koordinate ändert.

```
@Override
public void onSensorChanged(SensorEvent event) {
    //Der sensor aus ankommendem event
    Sensor sns2 = event.sensor;
    //Ist das der Sensortyp den wir suchen?
    if(sns2.getType() == Sensor.TYPE_ACCELEROMETER) {
        float x = event.values[0];
        float y = event.values[1];
        float z = event.values[2];
        final TextView tv  = (TextView) findViewById(R.id.txtv1);
        tv.setX(tv.getX() - x);
    }
}
```

3.3.2 Bemerkungen zum `SensorEventListener`

Während der Einrichtung von unserem Beschleunigungsensor, haben wir das Verhalten implementiert, indem wir mittels der Methode `registerListener()` den Beschlenigungssensor zusammen mit einem Objekt vom Typ `SensorEventListener` beim `SensorManager mngr` registriert haben. Die Entwickler, die sich diese Herangehensweise überlegt haben, haben dabei an das *Publish/Subscribe-Pattern* gedacht:

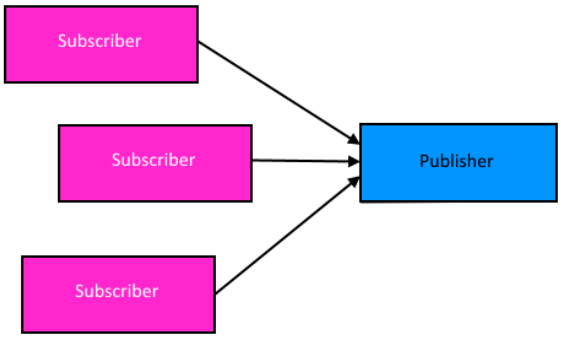


Abbildung 5: Eine Skizze des *Publish/Subscribe-Patterns.*

Beim *Publish/Subscribe-Pattern* gibt es zwei *Akteure*:

- Den *Publisher*, welcher Daten hält. Diese Daten können sich im Laufe der Zeit ändern.
- Mehrere *Subscriber*, die sich für die Daten vom *Publisher* interessieren und auf Änderungen der Daten beim *Publisher* eventuell reagieren möchten.

Wie funktioniert dieses *Entwurfsmuster* in der Praxis? Stellen wir uns mal einen *Publisher* und zwei *Subscriber* vor. Angenommen, der eine *Subscriber* stellt die Daten als Kreisdiagramm und der andere als Histogramm dar, wie in Abbildung 6. Wenn sich die Daten beim *Publisher* ändern, dann sind die Darstellungen der beiden *Subscriber* nicht mehr aktuell. Beide *Publisher* müssen diese Tatsache irgendwie mitbekommen.

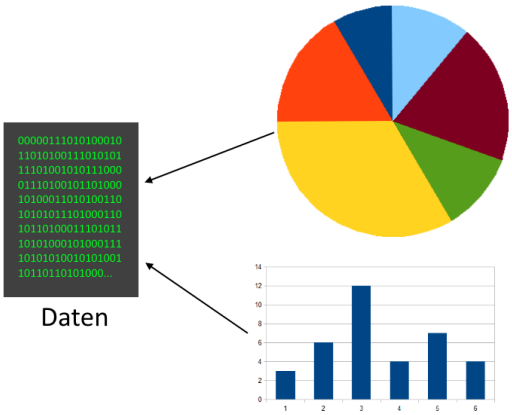


Abbildung 6: Die Daten vom *Publisher* werden von den *Subscribern* unterschiedlich dargestellt.

Beim *Publish/Subscribe-Pattern* informiert der *Publisher* seine *Subscriber* über die Änderung seiner Daten. Entweder der *Publisher* sendet sofort die veränderten Daten oder die *Subscriber* müssen die Daten nun aktiv nochmal anfordern.

Auf diese Weise ersparen wir uns die zyklischen Abfragen der einzelnen „Subscriber“ beim „Publisher“. Wieso möchten wir gerade das vermeiden? Stelle dir mal vor, dass du zu Hause regelmäßig zu deinem Telefon gehen musst, um zu überprüfen, ob dich gerade jemand anruft. Man kann leicht einsehen, dass dies alles andere als angenehm ist.

4 Der dritte Tag

Hast du schon mal den Geburtstag deines besten Freundes oder deiner besten Freundin vergessen? Das ist eigentlich kein Weltuntergang, aber wirklich schön ist dies eher nicht. Ab heute wird sich alles ändern, denn wir werden einen *Geburtstagsreminder* programmieren, der dich an die Geburtstage deiner Familienmitglieder und Freunde erinnert. Vorausgesetzt, dein Handy hat etwas Akku und ist angeschaltet ;)

Die Implementierung des Geburtstagsreminders haben wir in drei Schritte eingeteilt:

1. Anzeigen von Benachrichtigungen
2. Ausführung eines Dienstes im Hintergrund
3. Die Implementierung der Programmlogik

Wir konzentrieren uns vor allem auf die ersten zwei Punkte. Der dritte Punkt besteht eher aus „Implementierungsdetails“ und hat mehr mit Java zu tun, als mit Android selbst.

4.1 Benachrichtigungen

Interessant für unsere Zwecke ist die Klasse `NotificationCompat`. Mittels dieser können wir eine `Notification` absetzen. Dafür musst du die folgende Zeile in die Datei `build.gradle` (auf dem module-level) in den Bereich `dependencies` schreiben:

```
dependencies {
  //some Code
  compile "com.android.support:support-v4:25.0.2"   //This is for
  //notifications. The last number (here 25.0.2) should be the same as your
  //buildToolsVersion on top
}
```

Außerdem muss eventuell die Version angepasst werden. Anstatt `25.0.2` schreibst du die Version rein, die in `buildToolsVersion` steht. Diese findest du in der selben Datei unter dem Abschnitt `android`. Das sieht in etwa so aus:

```
android {
  //some Code
  buildToolsVersion "25.0.2"
  //some Code
}
```

Betrachte nun den folgenden Code:

```
package com.x.y;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.app.NotificationManager;
import android.support.v4.app.NotificationCompat;
import android.view.View;
import android.content.Context;

public class MainActivity extends AppCompatActivity {
```

```
12    @Override
13    protected void onCreate(Bundle savedInstanceState) {
14      super.onCreate(savedInstanceState);
15      setContentView(R.layout.activity_main);
16    }

18    public void sendNotification(View view) {
19      NotificationCompat.Builder mBuilder = new NotificationCompat.Builder(this);

21      Intent intent = new Intent(Intent.ACTION_VIEW,
22          Uri.parse("https://www.google.com/"));
23      PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, intent, 0);
24      mBuilder.setContentIntent(pendingIntent);

26      mBuilder.setSmallIcon(R.drawable.preferences_system_notifications_icon);
27      mBuilder.setContentTitle("My notification titel");
28      mBuilder.setContentText("Click me to open google in your browser.");

30      NotificationManagerCompat notificationManager =
31          NotificationManagerCompat.from(this);
32      notificationManager.notify(001, mBuilder.build());
33    }

35  }
```

Die Klasse `NotificationCompat.Builder` hilft uns dabei eine Benachrichtigung zu erzeugen. Zuerst setzen wir die Eigenschaften einer Benachrichtigung, wie z.B.

- das Symbol (`setSmallIcon(...)`),
- den Titel (`setContentTitle(...)`),
- den eigentlichen Inhalt (`setContentText(...)`) und
- die Aktion, die ausgeführt werden soll, wenn der Benutzer auf die Benachrichtigung klickt (`setContentIntent(...)`).

Bei der Gelegenheit sehen wir `Intents` in der Praxis. Wir erinnern uns, dass `Intents` der Kommunikation zwischen `Activities` dienen. Sobald wir auf die Benachrichtigung klicken, öffnet sich in einem Browser die Seite `www.google.com`. Sind die Parameter festgelegt, kann die Benachrichtigung über `build()` erstellt werden. Das Anzeigen der Benachrichtigung bewerkstelligen wir über die Klasse `NotificationManagerCompat` über die Methode `notify(...)`.

Nun kannst du einen `Button` auf die *UI* ziehen und über das `OnClick` Attribut festlegen, dass beim Drücken des Buttons die Methode `sendNotification(...)` ausgeführt werden soll. Falls alles korrekt implementiert ist, wird eine Benachrichtigung erscheinen, wenn auf den `Button` geklickt wird.

4.2 Services

Aktuell ist es nicht möglich den Geburtstagsreminder im *Hintergrund* laufen zu lassen. Das ist potenziell ziemlich ungünstig, weil wir erst die App öffnen müssen, damit wir an die Geburtstage

erinnert werden und wahrscheinlich erfahren, dass wir wieder einmal einen Geburtstag verschlafen haben. Hier können wir uns `Services` zunutze machen. Ein Beispiel für einen Service ist weiter unten zu sehen.

```
1  package com.x.y
2
3  import android.app.Service;
4  import android.content.Intent;
5  import android.media.MediaPlayer;
6  import android.os.IBinder;
7  import android.widget.Toast;
8
9  public class MyService extends Service {
10
11   MediaPlayer myPlayer;
12
13   @Override
14   public IBinder onBind(Intent intent) {
15     return null;
16   }
17
18   @Override
19   public void onCreate() {
20     myPlayer = MediaPlayer.create(this, R.raw.sun);
21     //eine mp3-Datei mit dem namen "sun" muss im
22     //"raw" Ordner vom "res" Ordner liegen!
23     myPlayer.setLooping(false);
24   }
25
26   @Override
27   public void onStart(Intent intent, int startid) {
28     myPlayer.start();
29   }
30
31   @Override
32   public void onDestroy() {
33     myPlayer.stop();
34   }
35
36 }
```

Das Verhalten von diesem Service ist relativ simpel. Die Methoden `onCreate()`, `onStart()` und `onDestroy()` sind aus dem zweiten Tag bekannt, wo wir über `Activities` gesprochen haben. Sobald der Service erstellt wird, wird Musik im Hintergrund abgespielt. Das heißt, erstellt eine `Activity` diesen `Service`, können wir von der App wieder auf den Hauptbildschirm navigieren und die Musik wird immer noch abgespielt. Analog wird das Abspielen der Musik gestoppt, wenn der Service terminiert wird.

Hinweis: `R.raw.sun` bezieht sich auf eine Audiodatei, die wir importiert haben. Eventuell musst du dies auch tun und anschließend die *ID* anpassen, damit alles erst läuft.

Schreibe in die Klasse aus Kapitel 4.1 die Methoden

```
1    public void onClick_start(View src) {
2      startService(new Intent(this, MyService.class));
3    }
```

```
5   public void onClick_stop(View src) {
6     stopService(new Intent(this, MyService.class));
7   }
```

und erstelle auf der *UI* zwei weitere `Buttons`. Die beiden Klassen kommunizieren wieder über `Intents`, wie du schon sicherlich bemerkt hast. Durch die Aufrufe der Methoden `startService(...)` und `stopService(...)`, können wir einen `Service` starten und stoppen.

Verändere wieder die `onClick` Attribute der `Buttons` so, dass die beiden neu hinzugekommenen Methoden angesprochen werden können über die `Buttons`. Nun kannst du auf Knopfdruck Musik abspielen lassen.

Schließlich muss in eurer Manifest Datei unter dem Abschnitt `application` folgender Code stehen:

```
1 <service
2             android:name=".MyService"
3             android:enabled="true"
4             android:exported="true"></service>
```

Eventuell hat *Android Studio* dies beim Erstellen des Services schon übernommen.

4.3 Implementierung der Logik

Hier ist es definitiv von Vorteil, wenn du mittlerweile etwas Java kannst. Die Logik werden wir gemeinsam implementieren im Workshop. Für alle anderen Leser, die nicht beim Workshop dabei sind, sei dieser Schritt als Übungsaufgabe nahegelegt.

Der mühsamste Schritt hier ist die persistente Speicherung und das Einlesen der Geburtstage. Dabei gibt es vier verschiedene Möglichkeiten Daten zu speichern, welche hier

https://developer.android.com/guide/topics/data/data-storage

aufgelistet werden. Wir werden die Geburtstage intern im Verzeichnis

`/data/data/<Paketname der App>`

speichern, wo ein Nutzer in der Regel nicht einfach so Zugriff auf die Datei hat (*Internal file storage*). Selbstverständlich existieren noch weitere mögliche Implementierungen. Zum Beispiel kannst du die Datei in dem Verzeichnis speichern, wo deine eigenen Dateien liegen (External file storage) oder in einer Datenbank.

4.4 Eine neue Activity starten

Bisher haben wir nur eine einzige `Activity` verwendet und bloß erwähnt, dass `Activities` andere `Activities` erzeugen können. Wir finden diese Behauptung hat zumindest ein kurzes Beispiel verdient. Schließlich ist auf Dauer eine App mit nur einer `Activity` nicht wirklich reizvoll.

Als erstes erstellen wir eine neue Activity. Klicke dafür mit der rechten Maustaste im Projekt Fenster auf der linken Seite auf den Namen deines Projektes. In dem nun geöffnetem Fenster wählst du `New` aus. Recht weit unten siehst du einen Punkt der `Activity` heißt. Wenn du nun mit der Maus darüber gehst, siehst du die ganzen Activitys, wähle am besten `Empty Activity` aus. Wir haben den Namen dieser neuen Activity einfach mal `SecondActivity` genannt. Du kannst sie aber natürlich so nennen wie du magst. Durch diese Art des Hinzufügens einer Activity musst du nichts weiter machen, das Programm fügt für dich automatisch die Abhängigkeiten bei Gradle und in dem Android Manifest für dich hinzu.

Damit du alles gleich ausprobieren kannst, empfehlen wir dir einen neuen Button in deiner bisherigen *UI* zu erstellen. Füge deiner `Activity` aus 4.1 die Methode

```
1   public void onClick_new(View src) {
2     Intent i = new Intent(this, SecondActivity.class);
3     startActivity(intent);
4   }
```

hinzu. Vergesse nicht diese Methode mit dem neu erstellten `Button` zu verknüpfen. Unsere automatisch erstellte neue Activity mit dem Namen `SecondActivity`, sieht wie folgt aus:

```
1  package com.x.y;
2
3  import android.support.v7.app.AppCompatActivity;
4  import android.os.Bundle;
5
6  public class SecondActivity extends AppCompatActivity {
7
8    @Override
9    protected void onCreate(Bundle savedInstanceState) {
10     super.onCreate(savedInstanceState);
11     setContentView(R.layout.activity_main);
12   }
13
14 }
```

Im Grunde genommen sehen wir weiter oben nur eine „nackte“ `Activity`. Du kannst nun gerne ausprobieren, ob alles funktioniert.

4.5 MVC

Ein *Architekturmuster* beschreibt die Organisation und Interaktion zwischen einzelnen Komponenten einer Anwendung. Im Grunde ist das *MVC-Pattern* ein bewährter Vorschlag von erfahrenen Entwicklern die eigene Software zu strukturieren und geschickt in Einzelteile zu zerlegen.

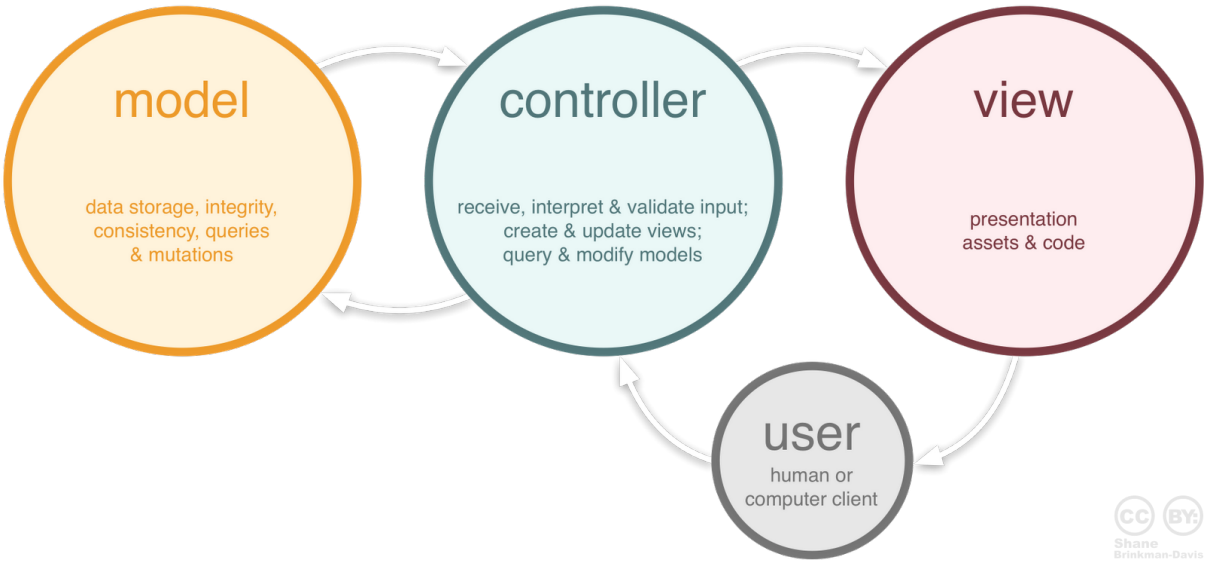


Abbildung 7: Das MVC-Pattern, Quelle: https://basicsofwebdevelopment.files.wordpress.com/2015/01/mvc_role_diagram.png

In diesem Fall haben wir im *MVC-Pattern* die Komponenten *Model*, *View* und *Controller*. Auf diese einzelnen Komponenten möchten wir nun näher eingehen, anhand des Beispiels eines Musikplayers.

4.5.1 Das Model

Im *Model* beschreiben wir ausschließlich die *Logik* unserer Anwendung. Bei unserem Taschenrechner wäre das *Model* der Code, der die verschiedenen Operationen auf Operanden durchführt, welche das *Model* vom *Controller* erhält.

4.5.2 Die View

Die *View* ist die Visualisierung von unserem *Model*. Genauer gesagt meinen wir die äußere Erscheinung einer Applikation ohne die Funktionen der einzelnen Objekte auf dem Bildschirm, wie z.B. `Button`, `EditText` usw..

4.5.3 Der Controller

Der *Controller* ist dafür zuständig auf äußere Ereignisse zu reagieren und als Vermittler zwischen *Model* und *View* zu fungieren. Sobald du mit einem `Button` oder einem `EditText` interagierst, erhält der *Controller* deine Interaktion in Form eines `Events`. Dieses `Event` wird vom *Controller* zum *Model* weitergeleitet. Ergebnisse von Berechnungen vom *Model* werden vom *Controller* wieder entgegengenommen und der *Controller* aktualisiert eventuell die *View* mit den neuen Daten aus dem *Model*.

Angenommen, du hast deinen Taschenrechner nach dem *MVC-Pattern* strukturiert: Kurz gefasst würde die Eingabe zweier Zahlen und derer Addition wie folgt funktionieren:

- Du gibst die Zahlen ein und drückst auf „=“ → Controller.
- Die Zahlen werden addiert → Model.
- Der Controller bekommt das Ergebnis und aktualisiert die View.

4.6 Kleine Bitte für den vierten Tag

Super! Die ersten Schritte in deiner Karriere als Appentwickler hast du nun gemeistert. Wir haben auch klein angefangen und finden, dass du dich über deinen bisherigen Erfolg freuen darfst :)

Wir werden uns nun einem neuen, etwas anspruchsvollerem Projekt zuwenden. Wir regen dich dazu an, dir ein eigenes Projekt zu überlegen, an dem du am vierten und letzten Tag arbeiten möchtest. Wir empfehlen dir ein nicht zu ambitioniertes Projekt auszusuchen. Suche dir am besten ein Projekt aus, welches du in *vier* Stunden fertigstellen kannst mit deinem aktuellen Kenntnisstand.

5 Der vierte Tag

Am heutigen Tag werden wir dir keine neuen Sachen zeigen. Du darfst die verbleibenden vier Stunden dieses Workshops nutzen, um dein Projekt voranzubringen. Bei Fragen, fragen! :)

Abbildung 8: Viel Spaß!, Quelle: https://cdn.pixabay.com/photo/2015/12/18/23/33/luck-1099292_960_720.jpg

Wir bedanken uns bei dir für deine Teilnahme und würden uns über eine ehrliche Rückmeldung zur Veranstaltung sehr freuen.

6 Literaturhinweise

- *Goll, Joachim: Architektur- und Entwurfsmuster der Softwaretechnik : Mit lauffähigen Beispielen in Java. Berlin Heidelberg New York: Springer-Verlag, 2014.*

- *Mednieks, Zigurd; Meike, G. Blake; Dornin, Laird; Nakamura, Masumi: Programming Android. 2. Aufl.. Sebastopol: "O'Reilly Media, Inc.", 2012.*