

Android



DI Dr. Heinz Schiffermüller
DI Ursula Riesel
HTBLA-Kaindorf
Abteilung EDVO

Erstellung: Aug. 2017
Letzte Überarbeitung: Sep. 2019

Inhaltsverzeichnis

1	Einführung	2
1.1	Systemarchitektur	2
1.1.1	Die Android Runtime	3
1.1.2	Das Application Framework	3
1.2	Android Installation	4
2	Erste Schritte	5
2.1	Erzeugen eines neuen Projekts mit Android Studio	5
2.2	Die Struktur eines Projekts	7
2.3	Das Erstellen einer einfachen App	8
2.3.1	Das Erstellen der Oberfläche - View	8
2.3.2	Das Einbinden von Text - strings.xml	10
2.3.3	Implementierung der Activity	10
2.4	Der Zugriff auf Ressourcen	13
2.5	Der Zugriff auf Elemente der GUI	14
2.6	Erzeugen von Debug-Informationen	14
2.6.1	Fehlermeldung in der App	15
3	Benutzeroberflächen	16
3.1	Layouts	16
3.1.1	Das LinearLayout	18
3.1.2	Das RelativeLayout	19
3.1.3	Das TableLayout und das GridLayout	20
3.1.4	Das ConstraintLayout	23
3.2	Event-Handling	23
3.3	Views	27
3.3.1	TextView	27
3.3.2	EditText	28
3.3.3	Button	28
3.3.4	ImageButton	29
3.3.5	RadioGroup und RadioButton	29
3.3.6	CheckBox	30
3.3.7	SeekBar	30
3.3.8	Maßeinheiten	31
3.4	Das MVC-Pattern	31
3.5	Events des Touchscreen	31
3.5.1	Long-click Events	31

1 Einführung

Android bezeichnet sowohl ein Betriebssystem als auch eine Software-Plattform für mobile Endgeräte wie Smartphones, Tablets oder Netbooks. Android ist eine Open Source Software, die von einem Konsortium bestehend aus 84 Firmen, der OHA (Open Handset Alliance), an deren Spitze Google steht, unterstützt wird.

Im Juli 2005 hat Google ein kleines Unternehmen namens Android Inc. mit Sitz in Palo Alto übernommen. Es wurde zu Beginn in aller Stille entwickelt, das erste Android SDK wurde 2007 veröffentlicht. Ab ca. 2009 ist die Verbreitung von Android massiv angestiegen.

Android Releases werden traditionell nach den Namen von Süßspeisen (mit alphabetisch aufsteigendem Anfangsbuchstaben) benannt. Wichtig Release-Meilensteine waren:

- **Cupcake** (1.5) - 2009
- **Eclair** (2.0, 2.1) - 2009
- **Gingerbread** (2.3) - 2010
- **Honeycomb** (3.0) - 2011
- **Ice Cream Sandwich** (4.0) - 2011
- **Jelly Bean** (4.1) - 2013
- **KitKat** (4.4) - 2013
- **Lollipop** (5.0) - 2014
- **Marshmallow** (6.0) - 2015
- **Nougat** (7.0) - 2016
- **Oreo** (8.0) - 2017
- **Pie** (9.0) - 2018

1.1 Systemarchitektur

Wichtiger Bestandteil von Android ist die virtuelle Maschine **Dalvik**. Sie führt nahezu alle Programme aus die auf einem Android-System gestartet werden. Der Ablauf ist dabei fast gleich wie unter Java und der JVM (Java Virtual Machine): ein Android Programm wird in Java programmiert und von einem Compiler in ein Dalvik Executable umgewandelt, das dann auf der Dalvik Virtual Machine ausgeführt wird. Seit Android 2.2 steht ein Just-In-Time Compiler zur Verfügung, der den Bytecode zur Laufzeit in ein noch schneller ausführbares Format umwandelt.

Das Android-Betriebssystem basiert auf Linux 2.6, erweitert um eine Reihe von C/C++ Libraries, auf die der Entwickler direkt über das Application-Framework zugreifen kann. Zu den wichtigsten Libraries zählen:

- **System C Library:** ist eine Implementierung der Standard C-Library, die speziell an mobile Endgeräte angepasst wurde.
- **Media Framework:** ermöglicht die Aufnahme und Wiedergabe zahlreicher Audio-, Video- und Grafikformate wie jpg, png, mpeg, mp3 etc.
- **Surface Manager:** kontrolliert die Bildschirmzugriffe und fügt 2D- und 3D-Ausgaben verschiedener Anwendungen zu einem Gesamtbild zusammen.
- **LibWebCore:** Rendering Engine für Webinhalte
- **SGL:** 2D-Grafiklibrary
- **OpenGL/ES:** 3D-Grafiklibrary - Open GL for Embedded Systems
- **FreeType:** Rendering Engine für Bitmap- und Vektorzeichensätze
- **SQLite:** Relationale Datenbank

1.1.1 Die Android Runtime

Besteht aus zwei Bausteinen:

- **Dalvik Virtual Machine**
- **Core Libraries**

Dalvik wurde rein für die Verwendung von mobilen Geräten konzipiert. Die Java .class Dateien werden hierbei mit einem Tool namens **dx** in sog. Dalvik Executables (.dex-Dateien) umgewandelt. .dex Dateien sind Bytecode-Dateien, die allerdings mehrere Java-Klassen enthalten können.

Der zweite wichtige Baustein, die **Core Libraries**, enthalten viele bekannte Java-Libraries wie z.B. `java.lang`, `java.io`, `java.math`, `java.net`, `java.util` uvm.

Für die Programmierung der Benutzeroberfläche, Telefonfunktionen, Multimedia etc. wird das Application Framework verwendet.

1.1.2 Das Application Framework

Das Application Framework stellt eine Reihe von Libraries zur Verfügung z.B. zur Erstellung von Benutzeroberflächen, für den Zugriff auf die Hardwarekomponenten, wie die Kamera, das Netzwerk oder Sensoren.

Ein wichtiges Konzept des Application Frameworks ist, dass Anwendungen ihre Funktionen veröffentlichen können, also auch von anderen Applikationen genutzt werden können.

Kernbestandteile des Application Frameworks sind:

- **Views:** Benutzeroberflächen mit Elementen wie Textfelder, Buttons, Radiobuttons, Listen etc.
- **Content Provider:** ermöglichen den Zugriff auf Daten anderer Programme, bzw. stellen die eigenen Daten zur Verfügung
- **Ressource Manager:** ermöglicht den Zugriff auf lokalisierte Zeichenketten, Grafiken und Layoutdateien.
- **Notification Manager:** ermöglicht Apps den Zugriff auf die Android Statuszeile, oder das Erzeugen von Popup-Nachrichten.
- **Activity Manager:** steuert den Lebenszyklus der Anwendung

1.2 Android Installation


Es empfiehlt sich folgende Umgebung zu installieren:

- Java **JDK 12**
- **Android Studio**
- **Android SDK**


Das Android SDK besteht aus einem Emulator, zahlreichen Tools, den Android Plattformen (d.h. die verschiedenen Android-Versionen), der Dokumentation und einer Reihe von Beispielen.

Bei der Installation empfiehlt sich folgende Reihenfolge:

- Installation des JDK 12
- Download und Installation von Android Studio
URL: <https://developer.android.com/studio>
- Installation des Android SDK: erfolgt über den Component Installer von Android Studio (AS) wenn eine Android Applikation das erste Mal gestartet wird.

Starten des Android **SDK-Manager** durch Klicken auf das  Icon, oder auf *Tools* → **SDK Manager**

Der SDK-Manger dient zur Installation von Android Plattformen, SDK-Tools, Dokumentation.

Starten des **AVD-Manager** (Android Virtual Device) durch Klicken auf das  Icon, oder auf *Tools* → **AVD Manager**.

Das AVD-Tool wird verwendet um ein bestimmtes physikalisches Gerät (also Smartphone) für den Simulator nachzubilden und zu konfigurieren. Dazu gehören Einstellungen für den Emulator-Skin (d.h. die Grafik mit der das Gerät dargestellt wird), die Kamera oder die Android-Version etc.

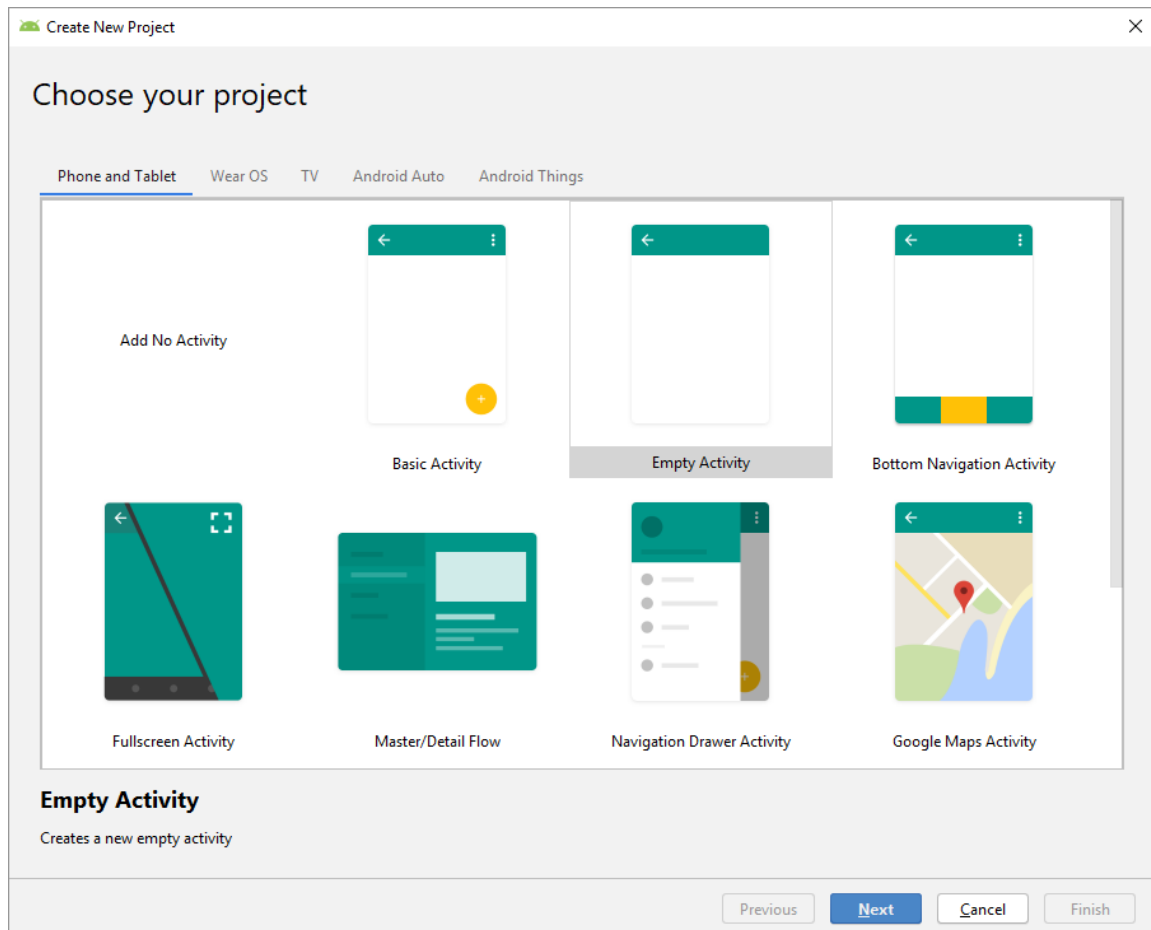
2 Erste Schritte

2.1 Erzeugen eines neuen Projekts mit Android Studio

Ein Projekt enthält alle Artefakte einer Android-Applikation. Dazu gehören Quellcode- und Konfigurations-Dateien, Grafiken, Sounds, Animationen etc..

Erstellen eines neuen Projekts:

→ *File* → *New* → *New Project...*

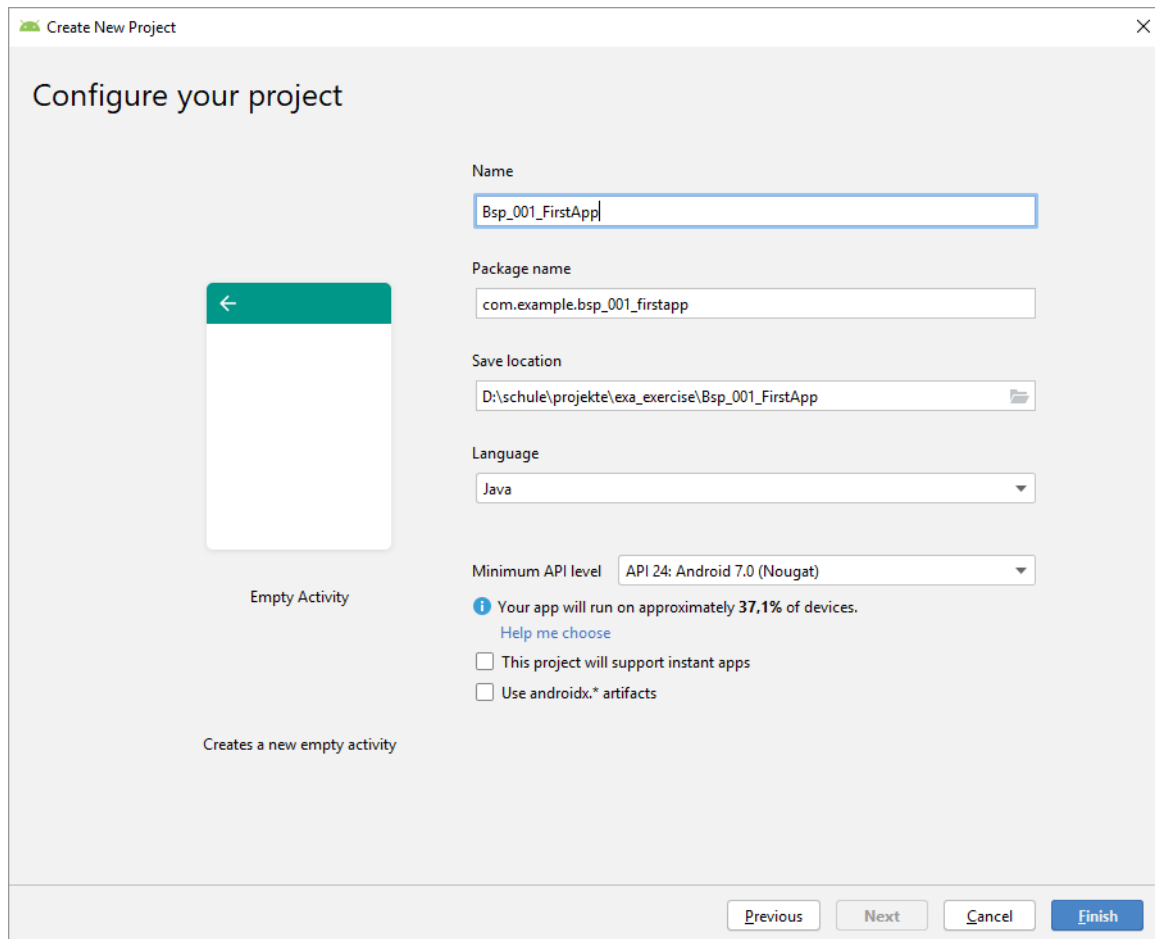


Empty Activity auswählen und auf *Next* klicken.

Im folgenden Konfigurationsfenster werden der Name der App, *Package name*, *Project location* sowie der *Minimum API level* angegeben. Wenn die Applikation am eigenen Handy laufen soll, ist darauf zu achten dass der ausgewählte API Level auch vom Handy unterstützt wird.

Um den aktuellen API Level am Handy zu finden:

→ *Einstellungen* → *Geräteinformationen* → *Softwareinfo*



Der unter *Name* vergebene Name wird später auf dem Gerät bzw. dem Emulator angezeigt. Der unter *Package name* angegebene Name ist für eine spätere Veröffentlichung bei Google Play wichtig, da er dort eindeutig sein muss! Idealerweise folgt man hier den Java Namenskonventionen für packages und gibt den eigenen Domain-Namen, gefolgt vom Namen der App, in umgekehrter Reihenfolge, an.

Also z.B.: `at.htlkaindorf.helloworld`

Die Angabe der Minimum SDK legt fest welche Android Version auf dem Endgerät zumindest installiert sein muss, damit die App lauffähig ist.

Die App kann über den Run-Button (grüner Pfeil) gestartet werden. Es erscheint ein Dialog um die Applikation entweder am Emulator oder am Handy zu starten.

Starten der Applikation am Handy:

Zuerst müssen auf dem Handy die Entwickleroptionen freigeschalten werden. Bei den meisten Handys lässt sich das wie folgt aktivieren:

- *Einstellungen*
- *Geräteinformationen*
- *Softwareinfo*
- 7 Mal auf *Buildnummer* klicken

Zurück zum vorherigen Screen ist nun der Menüpunkt Entwickleroptionen sichtbar. Dort *USB-Debugging* aktivieren.

Weitere Informationen auf: <https://developer.android.com/studio/debug/dev-options.html>

Installieren der USB-Driver am PC um das Handy unter Windows 10 verbinden zu können:

<https://developer.android.com/studio/run/oem-usb#Drivers>

Get OEM-drivers

- ➔ Hersteller suchen und auf Link klicken
- ➔ Driver herunterladen und installieren

Das Handy über USB mit dem PC verbinden.

Die Applikation in AS starten:

- ➔ Unter *Select Deployment Target* wird das verbundene Handy angezeigt
- ➔ Das Handy auswählen
- ➔ OK klicken

2.2 Die Struktur eines Projekts

Mit dem Erstellen des Projekts werden eine Reihe von Dateien und Verzeichnissen erzeugt, die im *Project* Fenster von AS sichtbar sind:

java-Verzeichnis: Hier befinden sich die Java-Quellcode-Dateien. In unserem Beispiel nur die Datei **MainActivity.java**

Hier können neue packages und Java-Klassen eingefügt werden.

res-Verzeichnis: Hier befinden sich Ressourcen die von der App benötigt werden, wie z.B. die Layout-Dateien (**activity_main.xml**), Strings (**strings.xml**), Farben (**colors.xml**), Bilder etc. Die Ressourcen einer App sollten immer in eigenen Dateien, also unabhängig vom Quellcode, verwaltet werden. Das dient dazu um die Schichten der App möglichst unabhängig voneinander zu halten.

generatedJava-Verzeichnis: Hier befinden sich vom ADT erzeugte Java-Dateien (generated Java Files), wie z.B. die Klasse **R.java** die alle wichtigen Ressourcen der App enthält. Der Inhalt dieser Datei setzt sich aus dem Inhalt der Dateien im Verzeichnis **res** zusammen und wird automatisch vom ADT generiert. So werden z.B. in der Datei **strings.xml** aus dem Verzeichnis **res/values** alle für die App notwendigen Strings definiert und vom ADT in **R.java** eingefügt.

manifests-Verzeichnis: Hier befindet sich die **AndroidManifest.xml**-Datei. Sie enthält die zentrale Beschreibung der Anwendung mit allen Bestandteilen der App (Activities, Services, Broadcast Receiver, Content Provider), Hardwarevoraussetzungen, benötigten Android-Versionen etc.

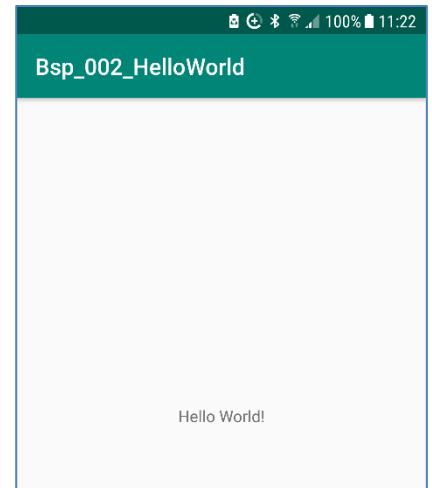
2.3 Das Erstellen einer einfachen App

Erzeuge und starte ein leeres Projekt (empty Project):

Um die GUI zu editieren: Doppelklick auf `res/layout/main_activity.xml`. Diese Datei enthält die vollständige Beschreibung der Oberfläche im XML-Format.

Mit `→ View → Tool Windows → Preview` wird das Vorschau-Fenster angezeigt.

Die Reiter um zwischen Text- und Design-View zu wechseln befinden sich am linken unteren Rand des Fensters.



2.3.1 Das Erstellen der Oberfläche - View

Die App soll mit drei einfachen Controls ausgestattet werden:

- Zwei nicht editierbaren Textfeldern (entspricht einem JLabel)
- Einem Eingabetextfeld das später ausgeblendet wird (entspricht einem JTextField)
- Einem Button, der mit CONTINUE oder FINISH beschriftet wird (entspricht einem JButton)

In unserer App sind Text-, Eingabefeld sowie Button Objekte der **View**-Klassen **TextView**, **EditText** und **Button**, das Layout ist ein Objekt der **ViewGroup**-Klasse **LinearLayout**.

Anders als in Java werden in Android Benutzeroberflächen mit Hilfe von XML-basierten Layout-Dateien beschrieben. Dadurch ist die Oberfläche unabhängig vom Quellcode und kann nachträglich verändert werden ohne dass die App neu kompiliert werden muss.

Die Text-Ansicht zeigt den Quelltext der Datei `main_activity.xml`:

```
<?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=".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>
```

Das Element **<ConstraintLayout>** ist das Wurzelement. Darunter befindet sich das **<TextView>** mit dem der "Hello World!" Text angezeigt wird. Das Constraint Layout ist eines von mehreren möglichen Layouts, die unter Android verfügbar sind.

Ändern des Layouts in der Design-Ansicht: → Component Tree → rechts-Klick auf das aktuelle Layout → *Convert View ...* → Layout auswählen → Apply

Ändere das Layout auf *Linear Layout*. Der "Hello World! " Text wird jetzt in der linken, oberen Ecke angezeigt.

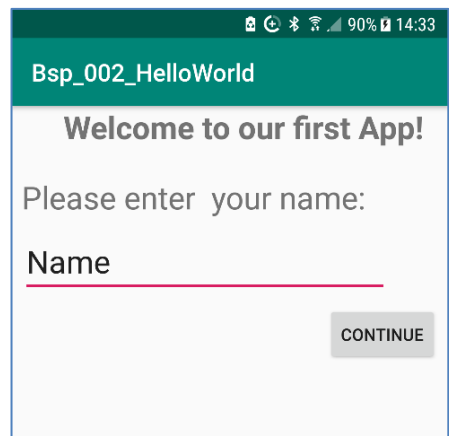
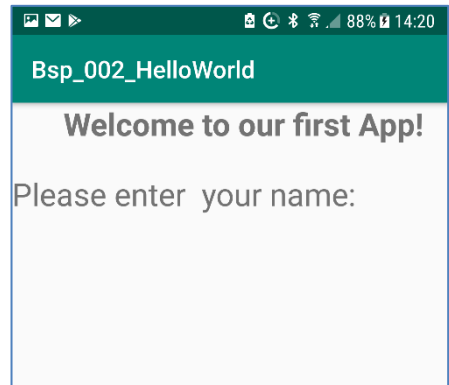
Beim Linear Layout werden alle Elemente/Controls nebeneinander oder übereinander angeordnet. Jedes Control bekommt gleich viel Platz. Mit der Eigenschaft *orientation* des Linear Layouts wird bestimmt ob die Elemente übereinander in einer Spalte - *vertical* - oder nebeneinander in einer Zeile - *horizontal* - angeordnet werden.

TextView Controls sind nicht editierbare Textfelder und damit das Gegenstück zum *JLabel* in Java-Swing. Ändere die Eigenschaften für *text*, *textSize* und *textStyle* entsprechend der Abbildung. Bei der *textSize* wird als Einheit sp - scalable pixel – verwendet, um die Textgröße an die bevorzugte Größe des Benutzers anzupassen.

Füge ein weiteres TextView Element ein und konfiguriere die Elemente entsprechend der Abbildung. Über die Eigenschaft *padding* wird der Abstand zu den anderen Controls bestimmt.

EditText Controls sind editierbare Textfelder (wie *JTextField* unter Java-Swing) damit der User Text eingeben kann. Von der Palette wird dafür das → *Text* → *Plain Text* Element eingefügt.

Zum Schluss wird ein *Button* Control (wie *JButton*) eingefügt. Über die Eigenschaft *layout_gravity* wird das Alignment des Controls bestimmt.



Um auf die einzelnen Controls in der Java-Anwendung zugreifen zu können müssen im Design-View unter id eindeutige Namen vergeben werden. Die Naming-Conventions sind dabei ähnlich wie in Java: eine TextView-Id beginnt mit den Buchstaben *tv*, EditText mit *ed* und Button mit *bt*. In der Datei *main_activity.xml* sollte das TextView-Control wie folgt definiert sein:

```
<TextView
    android:id="@+id/tvHeader"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Welcome to our first App!"
    android:textAlignment="center"
    android:textSize="26sp"
    android:textStyle="bold"/>
```

Nach dem Starten der App kann das Eingabetextfeld bereits benutzt werden, der Continue-Button ist aber noch ohne Funktion.

Tipp:

Um einen Screenshot auf einem Android Smartphone zu erzeugen:

- mit der Hand auf dem Bildschirm von links nach rechts wischen.
- Home- und Power-Button ca 2 Sek. gleichzeitig drücken
- VolumeDown- und Power-Button ca 2 Sek. gleichzeitig drücken
- Über die Spracheingabe: „OK Google“ – „Mache einen Screenshot“

2.3.2 Das Einbinden von Text - `strings.xml`

Anders als in Java, werden Texte/Strings nicht direkt im Quellcode definiert sondern in der eigens dafür vorgesehenen Datei `strings.xml` im Verzeichnis `res/values`

```
<resources>
  <string name="app_name">Bsp_002_HelloWorld</string>
</resources>
```

Wir erweitern die App um zwei weitere Strings, die später in der Programmlogik verwendet werden:

```
<resources>
  <string name="app_name">Bsp_002_HelloWorld</string>
  <string name="hallo">Hallo %1$s - nice to meet you</string>
  <string name="finish">finish</string>
</resources>
```

Das Attribut `name` des `<string>`-Tags wird später im Java Quellcode als Bezeichner verwendet und muss daher innerhalb des Projekts eindeutig sein. Die Zeichenfolge `%1$s` dient als Platzhalter um später den eingegebenen Namen einzufügen.

Nach Abspeichern der Datei `strings.xml` wird anschließend die Datei `R.java` automatisch vom ADT geändert.

2.3.3 Implementierung der Activity

Wegen der geringen Größe eines Smartphone-Bildschirms müssen Handy Apps anders gestaltet werden als Desktop oder Web-Anwendungen. Apps werden in mehrere kleinere Teile/Funktionsblöcke zerlegt. Für jeden dieser Teile wird eine sog. Activity implementiert.

Jeder Activity ist eine Benutzeroberfläche zugeordnet. Die Navigation innerhalb der App wird realisiert indem sich Activities gegenseitig aufrufen. Activities werden auf einem Stack abgelegt, so kann sehr einfach mit einem ZURÜCK-Button die zuvor verwendete Activity wieder angezeigt werden.

In der `HelloWorld`-App werden alle Funktionen in einer Activity abgebildet, die in der Klasse `MainActivity.java` implementiert wird. Die Methode `onCreate()` wird verwendet um Benutzeroberflächen aufzubauen und Variablen zu initialisieren.

```
package com.example.bsp_002_helloworld;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
```

```

import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {

    private TextView tvMessage;
    private EditText edInput;
    private Button btFinished;

    private boolean firstClick;

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

        tvMessage = (TextView) findViewById(R.id.tvMessage);
        btFinished = (Button) findViewById(R.id.btContinue);
        edInput = (EditText) findViewById(R.id.edInput);

        firstClick = true;

        btFinished.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                if (firstClick)
                {
                    tvMessage.setText(getString(R.string.hallo,
                                                edInput.getText()));
                    edInput.setVisibility(View.INVISIBLE);
                    btFinished.setText(R.string.finish);
                    firstClick = false;
                }
                else
                {
                    finish();
                }
            }
        });
    }
}

```

Für alle Controls, auf die im Quellcode zugegriffen wird, werden Instanzvariablen erzeugt. Das Laden und Anzeigen aller View-Elemente wird durch die Zeile:

```
setContentView(R.layout.activity_main);
```

realisiert. Der Aufruf sorgt dafür, dass alle Views und ViewGroups, die in `activity_main.xml` definiert wurden, angezeigt werden.

Der Zugriff auf die einzelnen Controls erfolgt mit der Methode `findViewById()`. Dazu wird mittels `R.id.name` auf das Control zugegriffen.

Die Controls werden mit der Methode `setText()` beschriftet, wobei der Zugriff auf die Strings aus der Datei `string.xml` mittels `R.string.name` erfolgt.

Um das Editfeld auszulesen und auf den Buttonklick zu reagieren, muss eine interne, anonyme `OnClickListener` Klasse implementiert werden und das dazugehörige `onClick`-Event überschrieben werden. Dies erfolgt analog zum Swing-Eventhandling in Java.

Die Methode `getString()` wird hier analog zur bekannten Methode `String.format()` verwendet.

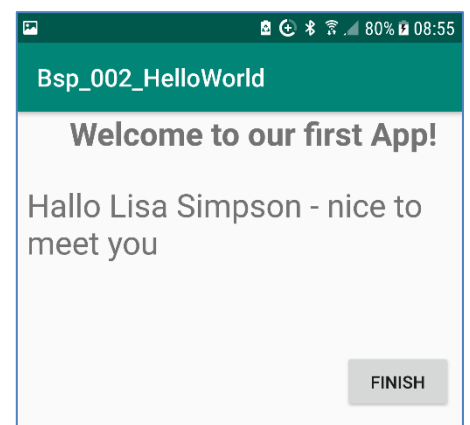
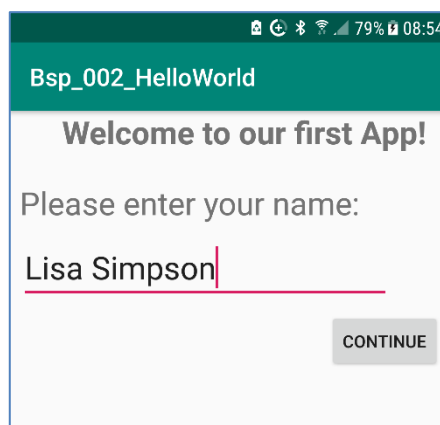
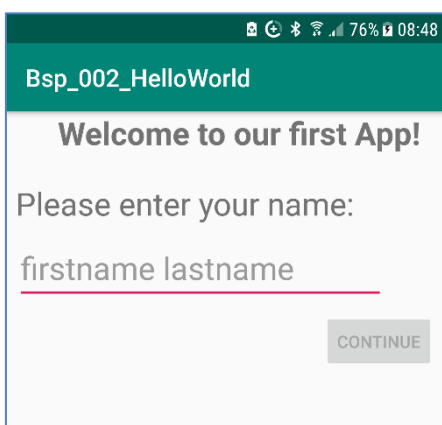
Die App ließe sich noch um folgende kleine Verbesserungen erweitern:

- Um einen Hinweis im EditText-Control anzuzeigen wird ein weiterer String in die Datei `strings.xml` eingefügt und über die Eigenschaft `hint` für das EditText-Control eingebunden.
- Um mehrzeilige Texteingaben zu verhindern muss die Eigenschaft `singleLine` für das `EditText`-Control auf `true` gesetzt werden.
- Um Anfangsbuchstaben automatisch in Großbuchstaben umzuwandeln muss die Eigenschaft `inputType` für das `EditText`-Control auf `textCapWords` gesetzt werden.
- Um zu gewährleisten, dass der Continue-Button erst geklickt werden kann wenn ein Name eingegeben wurde, muss ein `TextChangedListener` in der `onCreate()` eingefügt werden:

```
edInput.addTextChangedListener(new TextWatcher() {
    @Override
    public void beforeTextChanged(CharSequence s, int start, int count, int
after) { }

    @Override
    public void onTextChanged(CharSequence s, int start, int before, int
count) { }

    @Override
    public void afterTextChanged(Editable s) {
        btFinished.setEnabled(s.length() > 0);
    }
});
btFinished.setEnabled(false);
```



Tipp:

Mit Strg-Q wird die Android SDK Documentation für eine selektierte Klasse, Methode etc angezeigt. Beim automatischen Einfügen von Methoden einer überschriebenen Klasse, wie z.B. `TextWatcher` werden die Übergabeparameter entsprechend der API Documentation benannt.

Um die Android API Documentation zu installieren /aktivieren gehe in Android Studio auf:
Tools → *SDK Manager* → *Android SDK*

Im Reiter *SDK-Tools* Die Checkbox *Documentation for Android SDK* anklicken und mit *Apply* installieren.

2.4 Der Zugriff auf Ressourcen

Es gibt verschieden Möglichkeiten auf Ressourcen, die im **res**-Verzeichnis liegen, zuzugreifen. Damit Ressourcen zur Laufzeit zur Verfügung stehen werden in der Klasse **R.java** Konstante angelegt.

Wie schon bei den Strings gezeigt wird in der XML-Datei **strings.xml** ein Key-Value-Paar angelegt, wobei das Attribut **"name"** den Key definiert, der Inhalt des Tags den Value und der Tagname den Datentyp angibt.

Im Verzeichnis **res/values** können weitere Ressourcen in Form von XML-Dateien angelegt werden. So kann z.B. über *New* → *Values Ressource File* die Datei **values.xml** im Verzeichnis **res/values** angelegt werden um einen Boolean und einen Integer-Wert zu definieren:

Datei **res/values/values.xml**:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <bool name="bool1">true</bool>
    <integer name="int1">123</integer>
</resources>
```

Der Zugriff im Java-Quellcode erfolgt über eigene Methoden:

```
public class ZugriffActivity extends Activity {

    private static final String TAG = ZugriffActivity.class.getSimpleName();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        int val = getResources().getInteger(R.integer.int1);
        Log.d(TAG, "Ganze Zahl: "+val);

        Boolean bool = getResources().getBoolean(R.bool.bool1);
        Log.d(TAG, "Boolean: "+bool);
    }
}
```

Die Ausgabe am Logcat wäre:

```
ZugriffActivity:    Ganze Zahl: 123
ZugriffActivity:    Boolean: true
```

Es gibt weiterer getter-Methoden um auf verschiedene andere Datentypen in den Ressourcen-Dateien zuzugreifen. Weitere Informationen zu den Ressource-Typen finden sich im Android-Entwickler-Forum:

<http://developer.android.com/develop/index.html>

2.5 Der Zugriff auf Elemente der GUI

Jedes Element in der XML Layout-Datei kann eine eindeutige ID erhalten, die als String in folgendem Format angegeben wird:


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

Wird die App kompiliert, referenziert die ID auf einen Integer. Das `@+id` Symbol am Beginn des Strings bedeutet, dass der XML-Parser den String als neue Ressource in der Datei `R.java` aufnimmt.

In der Activity-Klasse wird mit der Methode `findViewById()` und dem String der ID auf das Element der View zugegriffen:

```
Button btStart = (Button)findViewById(R.id.btStart);
```

2.6 Erzeugen von Debug-Informationen

Auch bei Android-Projekten kann `System.out.println()` zum Erzeugen von Debug-Informationen verwendet werden. Das Programm muss dazu im Debug-Modus gestartet werden: entweder über den Debug-Button  oder über `RUN · DEBUG`. Die Ausgabe erfolgt auf der Console und im Debug Fenster.

Die Klasse `android.util.Log` enthält die statischen Logging-Methoden `v()`, `d()`, `i()`, `w()` und `e()`, die den Log-Levels *verbose*, *debug*, *info*, *warning* und *error* entsprechen. Übergeben werden an diese Methoden ein Tag-String, der die Quelle des Protokolleintrags kennzeichnet (meist der Name der Klasse oder der Activity), sowie der Ausgabetext:

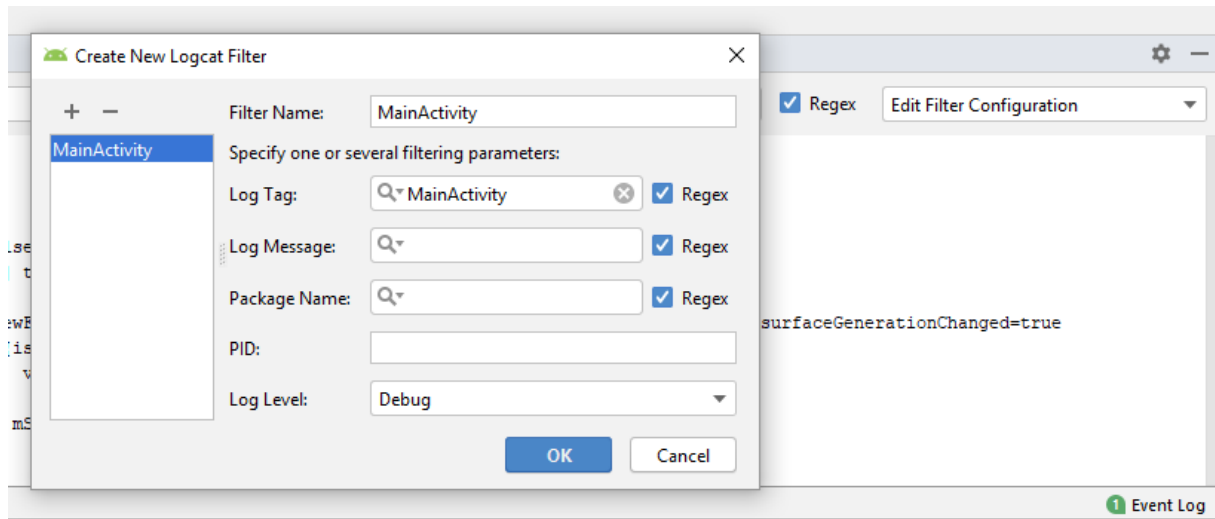
```
private static final String TAG = MainActivity.class.getSimpleName();

...

Log.v(TAG, "Allgemeine Ausgaben");
Log.d(TAG, "Debug-Ausgabe");
Log.i(TAG, "Informationen");
Log.w(TAG, "Warnungen");
Log.e(TAG, "Fehlermeldungen");
```

Alle Log-Informationen werden im Logcat-Fenster zur Laufzeit ausgegeben. Die Ausgabe kann entsprechend dem Level eingegrenzt werden, wobei die Levels hierarchisch angeordnet sind: im Level *verbose* werden auch alle weiteren Levels (*d*, *i*, *w* und *e*) angezeigt, im Level *error* werden nur mehr die *error*-Level Meldungen angezeigt.

Da im Logcat-Fenster sehr viele Meldungen angezeigt werden, können eigene Filter definiert werden: unter EDIT FILTER CONFIGURATION (rechts oben beim Logcat-Fenster) kann ein neuer Filter hinzugefügt werden:



Mit diesem Filter werden nur mehr Meldungen mit dem Log-Tag-String „MainActivity“ und dem Log Level „Debug“ im Logcat angezeigt.

2.6.1 Fehlermeldung in der App

Tritt eine Exception zur Laufzeit auf ist es oft notwendig den Benutzer eine Meldung anzuzeigen. Die einfachste Möglichkeit zur Ausgabe einer kurzen Fehlermeldung am Handy-Display ist die Verwendung der Klasse **Toast**:

```
Toast.makeText(getApplicationContext(), "Error", Toast.LENGTH_LONG).show();
```

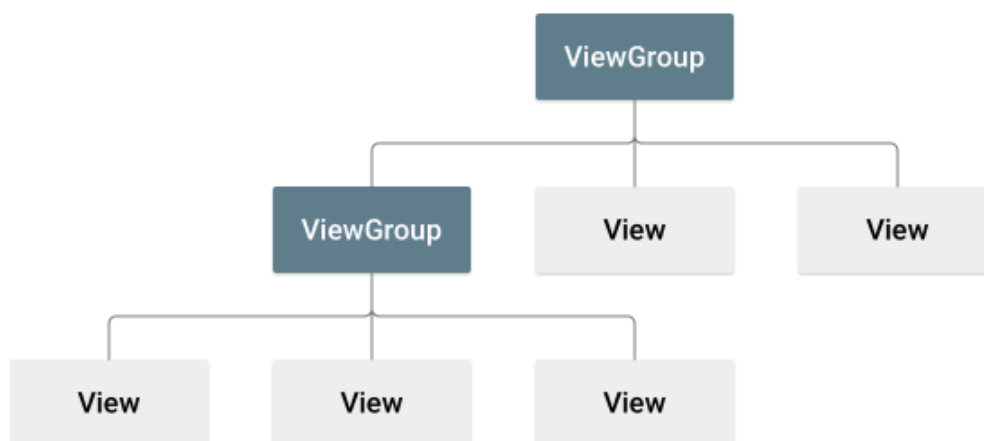
Es erfolgt eine Anzeige mit der Fehlermeldung am Bildschirm. Die Dauer der Anzeige wird durch die Konstanten **LENGTH_SHORT** bzw. **LENGTH_LONG** gesteuert.

3 Benutzeroberflächen

In Android werden Benutzeroberflächen mit Hilfe von *Layouts* definiert. Anders als in Java erfolgt die Beschreibung der GUI mit Hilfe einer XML-Datei. In Android Studio können Oberflächen entweder in der Text-Ansicht mit dem XML-Editor oder in der Design Ansicht mit einem graphischen Editor und einer Palette, die alle Elemente enthält, erstellt werden. Die XML-Dateien die das Layout definieren befinden sich im Verzeichnis **res/layout**.

3.1 Layouts

Layouts sind von der Klasse **ViewGroup** abgeleitet, und werden verwendet um darin UI Elemente, abgeleitet von der Klasse **View**, zu positionieren. Layouts können geschachtelt und kombiniert werden um komplexere Benutzeroberflächen zu erzeugen:



Alle Layouts und die darin verwendeten UI-Elemente (Buttons, Textfelder, ...) werden in Form von XML-Dateien gespeichert. In der Design-Ansicht von Android Studio können Benutzeroberflächen mittels Drag&Drop erstellt werden, die dazugehörige XML-Datei wird automatisch erzeugt.

Android bietet auch die Möglichkeit Layouts und UI-Elemente dynamisch im Java-Quellcode zu erstellen. Die XML-Deklaration wird jedoch ausdrücklich vorgezogen, da es dabei zu einer klaren Trennung zwischen der UI- und der BL-Schicht kommt.

Das Layout bildet das Wurzelement der XML-Datei:

```

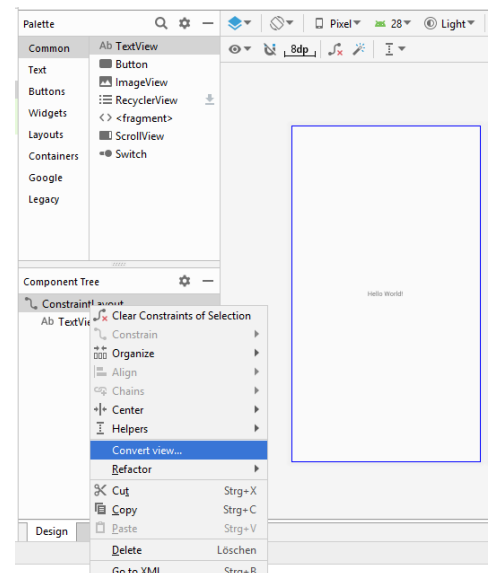
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"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

Das Android SDK stellt eine Reihe von Layouts zur Verfügung:

- **LinearLayout:** Alle Controls werden entweder in horizontaler oder vertikaler Richtung angeordnet. Das vertikale Layout positioniert die UI-Elemente spaltenförmig, das horizontale-Layout zeilenförmig. Jedes Element verfügt über das *weight*-Attribut mit dem die relative Größe des Elements innerhalb des Layouts festgelegt werden kann.
- **RelativeLayout:** Ein sehr flexibles Layout bei dem die Position jedes Elements relativ zu den anderen Elementen bzw. zu den Bildschirmrändern definiert wird.
- **GridLayout** und **TableLayout:** beide Layouts verwenden ein rechteckiges Gitter in dem die Elemente zeilen- und spaltenweise angeordnet werden können. Für diese Layouts empfiehlt es sich den Layout-Editor zu verwenden, statt direkt die XML-Datei zu bearbeiten.
- **ConstraintLayout:** Ein neues von Google entwickeltes Layout, das dem RelativeLayout ähnlich ist, aber flexibler genutzt werden kann und ohne Verschachtelungen auskommt. Es ist das, welches Android Studio als Default Layout verwendet.
- **FrameLayout:** Das einfachste aller Layouts. Jede Child-View wird an ihrer Default-Position, der linken oberen Ecken angezeigt. Da sich mehrere Views innerhalb des Layouts dadurch eventuell überdecken, können die Layout-Parameter *gravity* (Bündigkeit), sowie *width* und *height* zum Positionieren verwendet werden

Alle Layouts sind so aufgebaut, dass keine absoluten Koordinaten verwendet werden, um unterschiedliche Bildschirmgrößen möglichst optimal zu unterstützen.

In Android Studio ist das Constraint Layout das Default Layout. Das Ändern erfolgt im Component Tree Fenster: rechts-Klick auf das Layout → Context Menü → Convert View. Es öffnet sich ein Dialog in dem das Layout geändert werden kann.



Um das Layout zur Laufzeit zu erzeugen, muss in der Methode `onCreate()` der **Activity** Klasse die Methode `setContentView()` mit der XML-Layout-Datei aufgerufen werden:

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

3.1.1 Das LinearLayout

Im **LinearLayout** werden alle UI-Elemente in Form einer Zeile (**horizontal**) oder einer Spalte (**vertical**) über die Eigenschaft **android:orientation** angeordnet:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:gravity="right"
/>
```

Für die Eigenschaften **layout_width** und **layout_height** werden die Werte **match_parent** und **wrap_content** verwendet. Bei **match_parent** passt sich die Breite oder Höhe an die Größe des Parent-Objekts an. Das Parent-Objekt ist beim Wurzelement der Bildschirm, bei einem geschachtelten Element die Größe des darüberliegenden Elements. Bei **wrap_content** ergibt sich die Größe aus der Summe der Einzelgrößen der enthaltenen Elemente.

Über die Eigenschaften **layout_gravity** wird die Bündigkeit der Elemente definiert: **left**, **right**, **center**, **top**, **bottom**, bzw. zusammengesetzt in der Form: **top|left**, ...

Mit der Eigenschaft **layout_weight** lässt sich die Größe eines Elements in Relation zu den anderen Elementen gewichten: Mit dem Wert 0 wird nur der benötigte Platz verwendet, mit einem größeren Wert wird mehr Platz für das Element beansprucht. Elemente die den gleichen Wert haben, bekommen auch gleich viel Platz.

Beispiel:

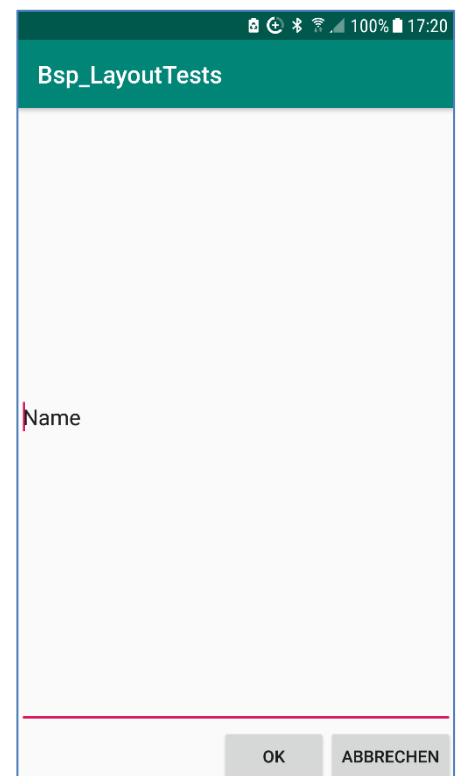
Im folgenden Beispiel werden zwei **LinearLayouts** geschachtelt um zwei nebeneinander angeordnete Buttons und ein **EditText**-Element anzuzeigen. Mit den zuvor beschriebenen Eigenschaften lassen sich unterschiedliche Aufteilungen erzielen:

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity">

    <EditText
        android:id="@+id/editText"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="Name" />

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

        <Button
            android:id="@+id/button2"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="0"
            android:text="@android:string/ok" />
```

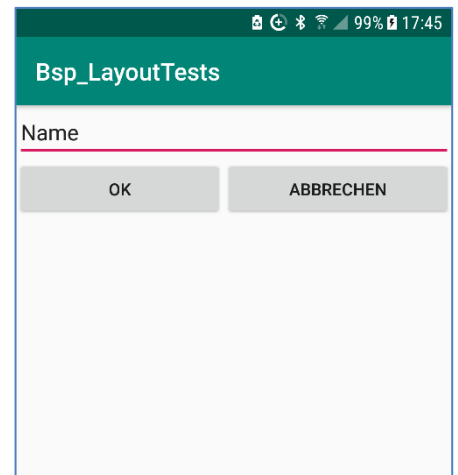


```

<Button
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_weight="0"
    android:text="@android:string/cancel" />
</LinearLayout>
</LinearLayout>

```

Nur durch Änderung des **layout_weight** beim EditText-Element von 1 auf 0 und bei den Button-Elementen von 0 auf 1 ergibt sich das rechts abgebildete Layout →



Mit der Eigenschaft **gravity** wird im Unterschied zu **layout_gravity** die Bündigkeit des Inhalts des Elements angegeben, also ob z.B. der Text innerhalb eines Buttons links- oder rechtsbündig angezeigt wird.

3.1.2 Das RelativeLayout

Im **RelativeLayout** werden Views in Relation zueinander angeordnet. Da durch dieses Konzept keine Verschachtelungen notwendig sind, wird die App performanter.

```

<RelativeLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

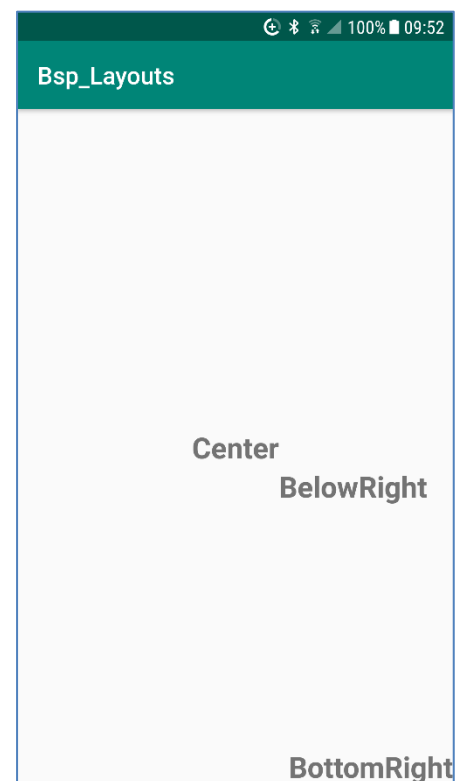
    <TextView
        android:id="@+id/textView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:text="Center"
        android:textSize="24sp"
        android:textStyle="bold" />

    <TextView
        android:id="@+id/textView2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/textView1"
        android:layout_toRightOf="@id/textView1"
        android:text="BelowRight"
        android:textSize="24sp"
        android:textStyle="bold" />

    <TextView
        android:id="@+id/textView3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentRight="true"
        android:layout_alignParentBottom="true"
        android:text="BottomRight"
        android:textSize="24sp"
        android:textStyle="bold" />

</RelativeLayout>

```



Eigenschaften die mit **true** oder **false** angegeben werden, um die Position der View in der Parent-View zu bestimmen:

Eigenschaft	Bedeutung
<code>layout_alignParentTop</code>	Bündigkeit oben im Parent
<code>layout_alignParentBottom</code>	Bündigkeit unten im Parent
<code>layout_alignParentLeft</code>	Bündigkeit links im Parent
<code>layout_alignParentRight</code>	Bündigkeit rechts im Parent
<code>layout_centerVertical</code>	vertikal im Parent zentriert
<code>layout_centerHorizontal</code>	horizontal im Parent zentriert
<code>layout_centerInParent</code>	vertikal und horizontal im Parent zentriert

Eigenschaften um die Position der View in Relation zu einer anderen View zu spezifizieren:

Eigenschaft	Bedeutung
<code>layout_below</code>	unterhalb
<code>layout_above</code>	oberhalb
<code>layout_toRightOf</code>	rechts
<code>layout_toLeftOf</code>	links

Eigenschaften um die Bündigkeit der View in Relation zu einer anderen View zu spezifizieren:

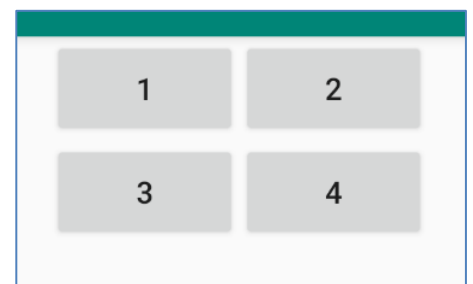
Eigenschaft	Bedeutung
<code>layout_alignTop</code>	oben bündig
<code>layout_alignBottom</code>	unten bündig
<code>layout_alignLeft</code>	linksbündig
<code>layout_alignRight</code>	rechtsbündig

3.1.3 Das TableLayout und das GridLayout

Beim **TableLayout** werden die Elemente zeilenweise eingefügt. Eine Zeile wird mit Hilfe des **TableRow** Tags definiert. Um 4 Buttons gitterförmig anzuordnen werden zwei Zeilen eingefügt:

```
<TableLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TableRow
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:gravity="center">
        <Button
            android:id="@+id/bt1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="1"/>
        <Button
            android:id="@+id/bt2"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="2"/>
    </TableRow>
    <TableRow
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:gravity="center">
        <Button
            android:id="@+id/bt3"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="3"/>
        <Button
            android:id="@+id/bt4"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="4"/>
    </TableRow>
</TableLayout>
```



```

</TableRow>
<TableRow
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center">
    <Button
        android:id="@+id/bt3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="3"/>
    <Button
        android:id="@+id/bt4"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="4"/>
    </TableRow>
</TableLayout>

```


Die Anzahl der Spalten im Layout entspricht der Anzahl der Spalten der **TableRow** mit den meisten Elementen.

Mit der Eigenschaft **android:stretchColumns** im **TableLayout**-Tag kann der Index oder die Indizes der Spalten angegeben werden, die über die Breite des Bildschirms gestreckt werden. Der Index ist 0-basierend:

```

<TableLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:stretchColumns="0,1">
    <!-- stretch first and second column -->
    <TableRow
        ...

```



1	2
3	4

Weitere Beispiele:

```

android:stretchColumns="1">    <!-- stretch the second column -->
android:stretchColumns="*">    <!-- stretch all columns -->

```

Weitere Eigenschaften:

android:collapseColumns="0"	Um die Spalte mit dem Index 0 unsichtbar zu machen
android:layout_span="2"	Um ein Element auf zwei Spalten auszudehnen

Beim **GridLayout** werden mit den Eigenschaften **columnCount** und **rowCount** die Anzahl an Zeilen und Spalten definiert. Alle Views werden in der Reihenfolge eingefügt. Das gleiche Beispiel wie vorher lässt sich mit **GridLayout** wie folgt realisieren:

```

<GridLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:columnCount="2"
    android:rowCount="2">

    <Button
        android:id="@+id/bt1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="1" />
    <Button
        android:id="@+id/bt2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="2" />
    <Button
        android:id="@+id/bt3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="3" />
    <Button
        android:id="@+id/bt4"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="4" />
</GridLayout>

```

Um Spalten über die gesamte Breite des Layouts zu strecken, wird die Eigenschaft `layout_columnWeight` verwendet. Es werden alle Elemente der gleichen Spalte mit dieser Eigenschaft ausgestattet:

```

<Button
    android:id="@+id/bt1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_columnWeight="1"
    android:text="1" />
<Button
    android:id="@+id/bt2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="2" />
<Button
    android:id="@+id/bt3"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_columnWeight="1"
    android:text="3" />
<Button
    android:id="@+id/bt4"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="4" />

```

3.1.4 Das ConstraintLayout

ToDo

3.2 Event-Handling

Das Android Event Management baut auf folgenden Konzepten auf:

Event-Listener:

Event-Listener sind Interfaces, die eine oder mehrere Callback-Methoden zu einem Event definieren. Diese Methoden müssen implementiert werden und werden aufgerufen wenn das dazugehörige Event registriert wurde und zur Laufzeit durch eine Benutzeraktion ausgelöst wird.

Registrierung des Event-Listeners:

Jeder Event-Listener muss sich bei einem oder mehreren UI-Elementen registrieren um tatsächlich zur Laufzeit aufgerufen zu werden. Zur Registrierung werden Methoden der UI-Element-Klassen verwendet: `setOnXxxListener()` oder `addOnXxxListener()`.

Event-Handler:

Sind die Methoden aus dem Listener-Interface, die ausimplementiert werden und zur Laufzeit ausgeführt werden, wenn das entsprechende Event ausgelöst wird.

Für jedes UI-Element können unterschiedliche Events verwendet werden. Um Event-Listener zu registrieren und die Handler-Methode zu implementieren gibt es vier verschiedene Varianten:

- (1) Verwendung einer inneren, anonymen Klasse oder einer Lambda Expression in der Activity Klasse
- (2) Verwendung einer inneren Klasse in der Activity Klasse
- (3) Die Activity-Klasse implementiert das Listener Interface
- (4) Der Name der Handler-Methode wird in der `activity_main.xml` Datei definiert, die Methode wird direkt in der Activity-Klasse implementiert

Beispiel:

Für einen Button wird ein Event-Handler implementiert, der auf einen Button-Klick reagieren soll. Dazu muss mit Hilfe der Methode `setOnClickListener()` ein `OnClickListener` für diesen `Button` registriert werden. Als Event-Handler wird die Methode `onClick()` implementiert. Die folgenden Code-Snippets zeigen alle drei Varianten:

Variante 1:

Verwendung einer inneren, anonymen Klassen um ein `onClick`-Event für einen Button zu registrieren und implementieren:

```
public class MainActivity extends AppCompatActivity {
    private Button btStart;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        btStart = (Button) findViewById(R.id.btStart);
    }
}
```



```

    btStart.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            Button bt = (Button)v;
            Log.i(TAG, "Click-Event on Button " + bt.getText());
        }
    });
    ...
}

```

Die Parameter View v der `onClick`-Methode enthält eine Referenz auf das UI-Element, das Auslöser des Click-Event ist, in unserem Fall der Button.

Die Registrierung-Methoden für die Listener-Interfaces sind nach folgendem Muster aufgebaut:

- `setOnXxxListener()` - wenn nur eine abstrakte Methode vorhanden ist. Hier kann auch eine Lambda-Expression zur Implementierung verwendet werden.
- `addOnXxxListener()` - wenn mehr als eine abstrakte Methode vorhanden ist.

Sind in der Handler-Methode mehrere Anweisungen notwendig, empfiehlt es sich, aus Gründen der Übersichtlichkeit, den Code aus der inneren Klasse in eine Methode der `Activity`-Klasse zu transferieren:

```

public class MainActivity extends AppCompatActivity {
    private Button btStart;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        btStart = (Button)findViewById(R.id.btStart);
        btStart.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                onStart(v);
            }
        });
    }

    public void onStart(View v) {
        Button bt = (Button)v;
        Log.v(TAG, "Click-Event on Button"+ bt.getText());
    }
}

```

Variante 2:

Verwendung einer inneren Klassen:

```

public class MainActivity extends AppCompatActivity {
    private Button btStart;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        btStart = (Button)findViewById(R.id.btStart);
        btStart.setOnClickListener(new MyOnClickListener());
        ...
    }
}

```

```

class MyOnClickListener implements OnClickListener {
    @Override
    public void onClick(View v) {
        Button bt = (Button)v;
        Log.i(TAG, "Click-Event on Button " + bt.getText());
    }
}

```

Hier implementiert die Klasse **MyOnClickListener** das **OnClickListener**-Interface und wird als innere Klasse, also als Klasse innerhalb einer Klasse, in der **MainActivity.java** implementiert. Die Registrierung erfolgt durch Übergabe einer Instanz der inneren Klasse.

Der Vorteil dieser Variante ist, dass die **onCreate()**-Methode sehr schlank und übersichtlich bleibt.

Ein weiterer Vorteil ergibt sich wenn mehrere Elemente, den gleichen Event-Handler verwenden. Im folgenden Beispiel registrieren 4 Buttons (**bt1**, ... **bt4**) den gleichen Listener:

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
    MyOnClickListener myCL = new MyOnClickListener();
    bt1.setOnClickListener(myCL);
    bt2.setOnClickListener(myCL);
    bt3.setOnClickListener(myCL);
    bt4.setOnClickListener(myCL);
    ...
}

```

Variante 3:

Die Activity-Klasse implementiert das Listener-Interface:

```

public class MainActivity extends AppCompatActivity
    implements View.OnClickListener {

    private Button btStart;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        btStart = (Button)findViewById(R.id.btStart);
        btStart.setOnClickListener(this);
        ...
    }

    @Override
    protected void onClick(View v) {
        Log.v(TAG, "Click-Event has been called");
    }
}

```

Bei der Registrierung wird mit dem **this**-Pointer eine Referenz auf die Activity-Klasse übergeben.

Variante 4:

Definition der Handler-Methode in der Layout-Datei `main_activity.xml`:

```
<Button
    android:id="@+id/btStart"
    android:layout_width="wrap_content"
    android:layout_height="match_parent"
    android:layout_weight="1"
    android:onClick="onStart"
    android:visibility="visible" />
```

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
    }

    @Override
    protected void onStart(View v) {
        Log.v(TAG, "Click-Event has been called");
    }
}
```

Diese Variante kann allerdings nur für `onClick`-Events verwendet werden.

3.3 Views

Jedes UI-Element auf einer Android GUI ist eine View und damit von der Klasse **View** abgeleitet.

Android unterstützt eine Vielzahl von vordefinierten Views:

- **TextView** zur Anzeige von Text
- **EditText** zur Eingabe von Text
- **Button** zum Klicken
- **Checkbox** und **RadioButton**
- **ScrollView** für scrollbaren, mehrzeiligen Text
- **ImageView** zur Anzeige von Bildern

Views können unterschiedlich konfiguriert werden. Allgemeine Eigenschaften von Views sind:

- **layout_width** und **layout_height** um die Größe der View im Layout festzulegen
- **width** und **height** um die Größe des Inhalts der View festzulegen
- **id** um auf die View innerhalb der Activity zugreifen zu können

Zu jedem Element gibt es eine große Anzahl von XML-Attributen zum Konfigurieren des Elements. Die meisten Attribute sind selbsterklärend und können sehr einfach über die Design-Ansicht in Android Studio gesetzt werden. In der folgenden Übersicht werden deshalb nur wenige Attribute beschrieben.

Alle UI-Elemente unterstützen eine größere Anzahl von Listener-Interfaces. Die wichtigsten davon sind:

Listener-Interface	Event-Handler	ausgelöst durch
onClickListener	onClick (View v)	einen einfachen Klick
onLongClickListener	onLongClick (View v)	einen längeren Klick (> 1 Sek.)
TextWatcher	afterTextChanged (Editable s)	bei jeder Änderung des Textes
	beforeTextChanged (CharSequence s, int start, int cnt, int after)	bei einer Änderung der cnt Zeichen ab dem Index start
	onTextChanged (CharSequence s, int start, int before, int cnt)	die cnt Zeichen ab dem Index start wurden geändert
onTouchListener	onTouch (View v, MotionEvent event)	ein Touch-Event
onDragListener	onDrag (View v, DragEvent event)	ein Drag-Event

Die Registrierung des **TextWatcher**-Interfaces erfolgt über die Methode **addTextChangedListener**.

3.3.1 TextView

Ein **TextView** Element wird verwendet um ein- oder mehrzeiligen Text anzuzeigen. In der Palette ist das Element unter *Text* → *TextView* zu finden.

```
<TextView
    android:id="@+id/tvHello"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerInParent="true"
```

```

    android:text="@string/hello"
    android:textSize="24sp"
    android:textStyle="bold" />

```

Die Größe des Textes wird über die Eigenschaft **textSize** definiert. Hier wird empfohlen als Einheit **sp** - scaled pixels - zu verwenden.

Der Text kann direkt als String eingegeben werden, oder wie im Beispiel oben, über die Datei **strings.xml**.

Die **id** sollte für alle Views vergeben werden, um in der Klasse **MainActivity.java** auf die View zugreifen zu können. Dort können die Eigenschaften über getter- und setter-Methoden verwendet bzw. geändert werden.

3.3.2 EditText

Das **EditText** Element wird verwendet, damit der Benutzer Text eingeben kann. In der Palette ist das Element unter *Text* → *Plain Text*, *Text* → *Password* usw. zu finden.

```

<EditText
    android:id="@+id/editText"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:ems="10"
    android:inputType="number"
    android:textSize="30sp" />

```

Mit der Eigenschaft **inputType** kann bestimmt werden, welche Werte der Benutzer eingeben darf. Wurde z.B. **inputType="number"** gesetzt, so können zur Laufzeit nur Zahlen eingegeben werden. Die meisten in der Palette angezeigten **EditText**-Elemente unterscheiden sich nur durch diese Eigenschaft.

Die Eigenschaft **ems** definiert die Breite des Elements in Abhängigkeit der ausgewählten Font. Als Referenz wird der Buchstabe m verwendet. Der Wert 10 bedeutet daher, dass die Breite des Feldes groß genug ist, um 10 Mal den Buchstaben 'm' einzugeben.

3.3.3 Button

Das **Button** Element ist in der Palette unter *Buttons* → *Button* zu finden.

```

<Button
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:onClick="onStartClick"
    android:text="Start" />

```

Mit der Eigenschaft **onClick** kann eine Handler-Methode registriert werden, die in der Activity-Klasse implementiert werden muss.

Der Button kann außer einer Beschriftung auch Icons enthalten, die über die drawable-Eigenschaften **drawableLeft**, **drawableRight**, ... gesetzt werden.

3.3.4 ImageButton

Verhält sich gleich wie ein `Button`, mit dem Unterschied, dass statt des Textes nur ein Image angezeigt wird.

```
<ImageButton
    android:id="@+id/starButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:srcCompat="@android:drawable/btn_star_big_on" />
```

3.3.5 RadioGroup und RadioButton

Die `RadioGroup`, wird auf der Oberfläche positioniert, die dazugehörigen `RadioButton`-Elemente darin eingefügt.

```
<RadioGroup
    android:id="@+id/optionGroup"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="vertical">

    <RadioButton
        android:id="@+id/rbOptionA"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Option A" />

    <RadioButton
        android:id="@+id/rbOptionB"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Option B" />
</RadioGroup>
```

Über die Eigenschaft `orientation` der `ButtonGroup` kann eine horizontale oder vertikale Ausrichtung der Radiobuttons definiert werden.

Gleich wie beim `Button` kann die `onClick`-Eigenschaft verwendet werden.

Wird in der Activity-Klasse ein `onCheckedChangeListener` auf die `ButtonGroup` gesetzt, so reagieren alle Radiobuttons in der Gruppe auf dieses Event:

```
btGroup = (RadioGroup) findViewById(R.id.optionGroup);
btGroup.setOnCheckedChangeListener(new RadioGroup.OnCheckedChangeListener()
{
    @Override
    public void onCheckedChanged(RadioGroup group, int checkedId) {
        RadioButton rb = (RadioButton) findViewById(checkedId);
        Log.i(TAG, "clicked on " + rb.getText().toString());
    }
});
```

An die Methode `onCheckedChanged` wird neben der `RadioGroup` auch die ID des angeklickten Radiobuttons übergeben, auf den somit leicht zugegriffen werden kann.

Alternativ lässt sich ein `onCheckedChangeListener` für jeden einzelnen Radiobutton registrieren.

<i>Listener-Interface</i>	<i>Event-Handler</i>	<i>ausgelöst durch</i>
<code>OnCheckedChangeListener</code>	<code>onCheckedChanged</code> (<code>RadioGroup group</code> , <code>int checkedId</code>)	einen einfachen Klick

3.3.6 CheckBox

Die `CheckBox`, verhält sich ähnlich wie ein `RadioButton` ohne `ButtonGroup`.

```
<CheckBox
    android:id="@+id/cbEnglish"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="English" />
```

Am häufigsten wird für Checkboxes ein `OnClickListener` verwendet.

3.3.7 Seekbar

Die `SeekBar` entspricht dem Slider in Java. Über die Eigenschaften `Min`, `Max` und `Progress` werden der kleinste Wert, der größte Wert und das Inkrement bestimmt

```
<SeekBar
    android:layout_width="300dp"
    android:layout_height="wrap_content"
    android:max="36"
    android:min="12"
    android:progress="2"
    android:progressTint="#FF9800"
    android:thumbTint="#FF9800"
    android:verticalScrollbarPosition="left" />
```

Mit den Eigenschaften `progressTint` und `thumbTint` lassen sich die Farben von Progress und Thumb einstellen.

Über die Methode `setOnSeekBarChangeListener` lässt sich das `Listener_interface` registrieren:

<i>Listener-Interface</i>	<i>Event-Handler</i>	<i>ausgelöst</i>
<code>OnSeekBarChangeListener</code>	<code>onProgressChanged</code> (<code>SeekBar sb</code> , <code>int progress</code> , <code>boolean fromUser</code>)	bei jeder Änderung der Seekbar
	<code>onStartTrackingTouch</code> (<code>SeekBar sb</code>)	am Beginn des Drag-Prozesses
	<code>onStopTrackingTouch</code> (<code>SeekBar sb</code>)	am Ende des Drag-Prozesses

3.3.8 Maßeinheiten

Für Längenangaben, wie Breite, Höhe und Schriftgröße können in Android verschiedene Einheiten verwendet werden:

dp oder dip (density independent pixel)	Einheit die sich an der physikalische Dichte, also der Auflösung des Bildschirms, orientiert. Die Elemente werden auf Smartphones mit unterschiedlicher Auflösung und Größe, im Verhältnis gleich groß dargestellt.
sp (scale independent pixel)	Einheit die ähnlich wie dp ist, mit dem Unterschied, dass zusätzlich die vom Benutzer eingestellte Schriftgröße zur Skalierung verwendet wird. Diese Einheit sollte ausschließlich für Schriften verwendet werden.
px (pixel)	Einheit die sich an den Pixel orientiert - d.h. bei unterschiedlicher Auflösung der Smartphones werden die Elemente unterschiedliche groß dargestellt .

Damit die Benutzeroberfläche auf Bildschirmen unterschiedlicher Auflösung und Größe gleich angezeigt wird sollten immer **dp**-Einheiten, bzw. **sp**-Einheiten für Schriftgrößen verwendet werden.

3.4 Das MVC-Pattern

Um eine klare Trennung zwischen den verschiedenen Teilen einer Android Applikation zu erreichen wird das Model-View-Control Pattern, kurz MVC-Pattern, eingesetzt. Die einzelnen Schichten sind wie folgt definiert:

- **Model** - enthält die Geschäftslogik (den Business-Layer) der App. Dazu gehören z.B. Java-Beans Klassen sowie weitere Klassen, die Programmlogik enthalten.
- **View** - ist zuständig für die grafische Darstellung der App: also die Benutzeroberfläche und der Darstellung der Daten aus dem Modell. In Android wird das in der Datei `activity_main.xml` realisiert.
- **Controller** - ist das Bindeglied zwischen der View und Model. Der Controller wird benachrichtigt von Benutzeraktionen und initiiert Updates auf dem Modell und auf der View. Die Rolle des Controllers übernimmt die Klasse `MainActivity.java`.

3.5 Events des Touchscreen

Android stellt eine Reihe grundlegender Events

3.5.1 Long-click Events

Die `CheckBox`, verhält sich ähnlich wie ein `RadioButton` ohne `ButtonGroup`.

```
<CheckBox
    android:id="@+id/cbEnglish"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="English" />
```