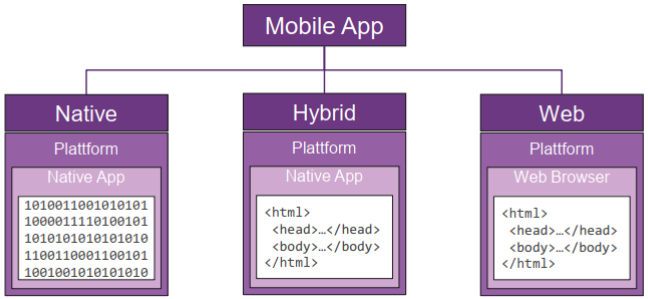
Android

Open Source unter der Apache Lizenz
• Linux Kernel unter GPL 2

Single-Plattform + Native
• Android SDK • iOS SDK
Cross-Plattform + Hybrid
• Cordova • Ionic
Cross-Plattform + Native
• Flutter • Xamarin
Cross-Plattform + Web
• Angular • Vue.js



Vor- und Nachteile der Varianten

Eigenschaft	SP Native	CP Native	CP Hybrid	CP Web
Performance	+++	+++	++	+
Look & Feel[1]	+++	+++	++	+
Zugriff auf Gerätefunktionen[2]	+++	+++	++	+
Portabilität von Code	+	++	+++	+++
Anzahl benötigter Technologien	+	++	+++	+++
Re-Use von existierendem Code	++	++	++	+++
Upgrade Flexibilität[3]	+	+	+	+++
Installationserlebnis[3]	+++	+++	+++	+
Offline Nutzung	+++	+++	+++	++
Gebühren für Veröffentlichung	+	+	+	+++

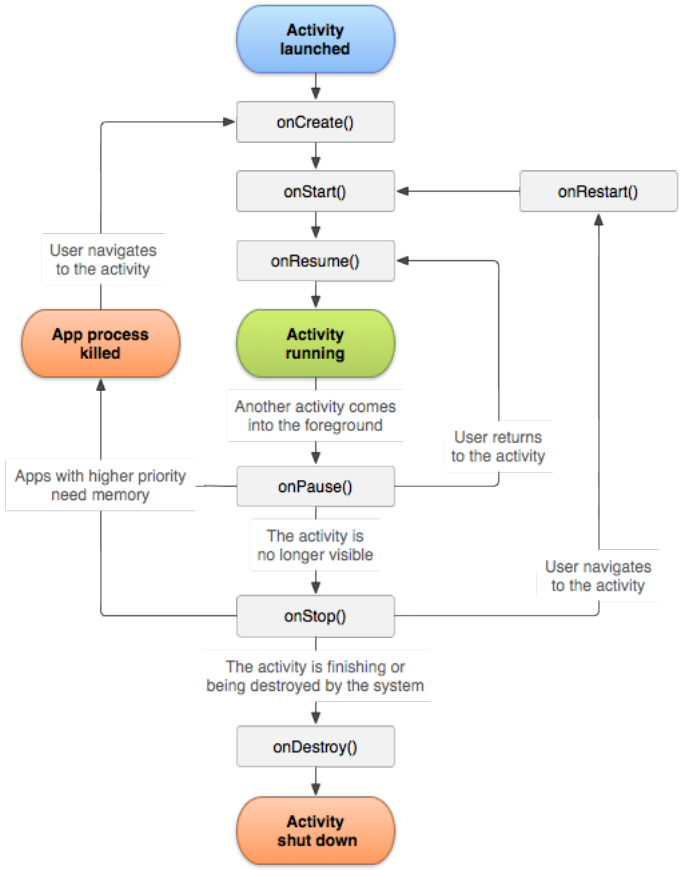
Grundprinzipien

Apps bestehen aus lose gekoppelten, wiederverwendbaren **Komponenten** (**Activities, Content Providers, Services und Broadcast Receivers**). Android hat die Kontrolle über ausgeführte Apps (**Verwaltung des Lebenszyklus, Kommunikation zwischen Komponenten, Terminierung bei Bedarf** (z.B. Speicherknappheit))

Activities

- Activities sind die Grundbausteine einer App
- Eine Activity eine Aufgabe (”Aktivität”), Kontakt suchen, Fotos anschauen
- Jede App enthält 1-n Activities
- Beim App-Start wird die Main Activity von Android erzeugt und ausgeführt
- Registrierung im AndroidManifest.xml nötig
- Activities besitzen eine grafische Oberfläche und verarbeiten Benutzereingaben

Activity Lifecycle



Typische Anwendungsfälle

- Erzeugung des GUI: onCreate()
- Datensicherung: onPause() für schnelle Operationen, ansonsten onStop()
- Dienste wie Lokalisierung aktivieren/deaktivieren: onResume() und onPause()
- Zustand des GUI erhalten, z.B. bei Rotation: onSaveInstanceState() und onRestoreInstanceState()

Intents

- Die Kommunikation zwischen Komponenten erfolgt über Intents (Absicht, Vorhaben)
- Explizit: Aufruf einer definierten Komponente (typischerweise für Komponenten der eigenen App)
- Implizit: Aufruf einer passenden Komponente (typischerweise für Komponenten aus anderen Apps)
- Apps können sich im Android Manifest mit Intent Filters auf implizite Intents registrieren
- Intents werden stets von Android verarbeitet

```
--MainActivity.java--
// Expliziter Intent
Intent secondActivityIntent = new Intent(this,
SecondActivity.class);
startActivity(secondActivityIntent);
// Impliziter Intent
Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.setType("text/plain");
sendIntent.putExtra(Intent.EXTRA_TEXT, "Hey!");
startActivity(sendIntent);

--SecondActivity.java--
Intent intent = this.getIntent();
Bundle extras = intent.getExtras();
int parameter = extras.getInt("myKey");
SecondActivity.java
```

Rückgabewert erhalten via Callback, non-blocking

```
startActivitiyForResult(intent, CODE);
@Override
protected void onActivityResult(int requestCode, int resultCode,
Intent data) { ... }
//Receiving End
this.setResult(Activity.Result_OK); this.finish();
```

Prüfen ob Empfänger vorhanden (Muss nicht immer so sein, Exception)

```
boolean hasReceiver =
intent.resolveActivity(getPackageManager()) != null;
```

AndroidManifest.xml

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

Back Stack

Ausgeführte Activities werden im Back Stack verwaltet
Activities eines Stacks können zu verschiedenen Apps gehören
Dieselbe Activity kann mehrfach im selben Stack enthalten sein

Tasks

Ein Back Stack wird auch Task genannt
Android verwaltet die Ausführung von Tasks
Mittels Overview Screen kann zwischen Tasks gewechselt werden
Bei Bedarf können Activities in neuen Tasks gestartet werden

Prozesse

Jedes APK hat einen eigenen Linux User und einen eigenen Linux Prozess.
Jeder Prozess hat mindestens einen Thread (MainThread). Das schützt vor gegenseitigen Speicherzugriffen

Main Thread

Achtung: nur der Main Thread darf das GUI aktualisieren, sonst Exception
Option 1: Activity.runOnUiThread(Runnable)
Option 2: View.post(Runnable)
Option 3: Handler und Looper

Event Handling im Code

```
final TextView textView = this.findViewById(R.id.text_example);
Button button = this.findViewById(R.id.button_example);
button.setOnClickListener(v -> {
    textView.setText("Button pressed");
});
```

Event Handling im XML

Die Auflösung erfolgt zur Laufzeit via Reflection.
Code-Variante bevorzugen (Trennung von UI und Logik)

```
android:onClick="onExampleButtonClicked
```

Resources

In Android werden alle Dateien, die keinen Code enthalten, als Resources bezeichnet
Zugriff erfolgt über Resource IDs
Zugriff via R-Klasse
In XML können Resources mittels @-Notation abgerufen werden:
@[<package_name>:]<resource_type>/<resource_name>
Neue IDs für UI-Elemente werden mit @+id/ erzeugt

Value-Resources

colors.xml für Farbwerte
dimens.xml für Dimensionen
strings.xml für Texte
styles.xml für Styles

Bei Value-Resources enthält eine Datei mehrere Ressourcen
• Sonst gilt eine Datei = eine Ressource (z.B. Layouts) Empfehlungen
• Veränderliche Werte immer in passenden Files definieren und referenzieren
• Gilt insbesondere für mehrfach verwendete Werte (z.B. Farben, Schriftgrössen, ...)

Dimensionen

dp: Density-independent Pixels
sp: Scale-independent Pixels
px: Pixel
pt: Punkte (1/72 eines physikalischen)
in: Inch
mm: Millimeter
Empfehlungen
- **Schriften immer** in **sp**
- **Alles andere** in **dp**

Qualifier

Die Auslagerung in XML-Dateien dient nicht nur der sauberen Trennung
Resources können in unterschiedlichen Varianten hinterlegt werden
• Texte für verschiedenen Sprachen
• Bilder für verschiedenen Auflösungen
• Layouts für unterschiedliche Gerätetypen
Zur Unterscheidung werden die Verzeichnisnamen mit Qualifiern ergänzt
• Liste der Qualifier
• Qualifier können kombiniert werden
• Reihenfolge der Qualifier muss korrekt sein
• Wizard in Android Studio hilft dabei
Android lädt automatisch die passendsten Ressourcen

Assets

Resources im res/-Ordner können nur via Resource ID zugegriffen werden
Ist der Zugriff auf die Originaldateien oder Dateistruktur nötig, so müssen diese Dateien im assets/-Ordner platziert werden
Via AssetManager-Klasse kann dieser Ordner wie ein Dateisystem gelesen werden
Achtung: die Verwendung von Qualifiers ist nicht möglich

App Manifest

Das AndroidManifest.xml enthält essenzielle Informationen zur App • ID, Name, Version und Logo
• Enthaltene Komponenten
• Hard- und Softwareanforderungen
• Benötigte Berechtigungen

Application ID und Version

package

• Eindeutige Identifikation der App
• Definiert den Namespace für die Anwendung
• Reversed Internet Domain-Format

versionName

• Ein menschenlesbarer String
• Typischerweise Semantic Versioning

versionCode

• Ein positiver Integer für interne Verwendung
• Je höher die Zahl, desto ”neuer” die App
• Unterschiedliche Ansätze zur Inkrementierung

Application-Element

AndroidManifest.xml

```
<application android:name="MyApplication">
<!-- gekürzt -->
</application>
```

• Application ist auch eine Klasse, die den globalen Zustand unserer App hält, und enthält überschreibbare Lifecycle-Methoden
• Wir können optional eine eigene Ableitung von Application registrieren

Rückwärtskompatibilität

API Levels

Das API Level ist eine Zahl, welche die Version der Android API identifiziert
Jedes Level enthält immer alle älteren APIs, ggf. aber deprecated
Um möglichst viele Geräte zu erreichen, sollte der API möglichst niedrig sein
Bestimmte Funktionen sind jedoch nur in neueren API Levels verfügbar
Android Studio unterstützt bei der Wahl des passenden Levels

API Levels im Manifest und Gradle

minSdkVersion gibt an, welche Version das Gerät mindestens haben muss
maxSdkVersion gibt an, welche Version das Gerät maximal haben darf Android ignoriert diesen Wert seit Version 2.0.1 Der Google Play Store verwendet ihn als Filter
Empfehlung: nicht verwenden
targetSdkVersion ist die höchste Version, mit welcher die App getestet wurde
compileSdkVersion gibt an, mit welcher API die App kompiliert wird
minSdkVersion <= targetSdkVersion <= compileSdkVersion

Android Jetpack und AndroidX

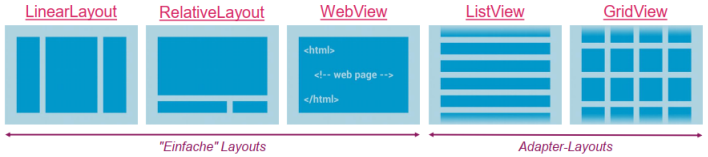
AndroidX ersetzt Android Support Libraries
• AppCompat(Library in Jetpack) macht neue Features auf tieferen API Levels verfügbar
• Grundidee: Verwendung von Elementen aus dem androidx-Namespace anstelle der normalen Android-Komponenten

GUI

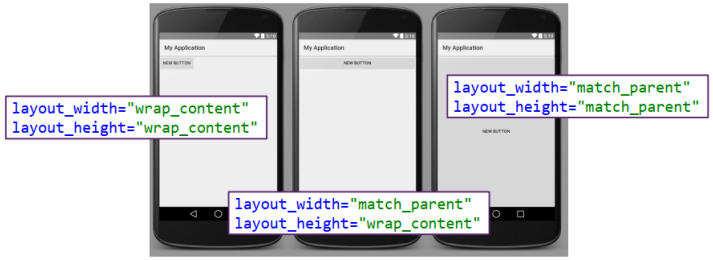
Das GUI kann auf zwei Arten erstellt werden
• Deklarativ: Beschreibung in XML
• Imperativ: Beschreibung im Quellcode
GUI-Elemente werden hierarchisch angeordnet
• View: Widgets (Controls)
• ViewGroup: Layouts (Container)

Die Basisklasse aller GUI-Elemente ist View
ViewGroup-Klassen werden auch Layouts(1) oder Container genannt

Layouts



Layouts können beliebig geschachtelt werden – mit negativem Einfluss auf die Performance
• Bevorzugt sind breite, flache Hierarchien
Auch sind die mit `layout_`-Parametern definiert Werte nur Wünsche an das Eltern-Layout; diese müssen beim Berechnen der realen Grössen ggf. ignoriert werden



`layout_width`: Gewünschte Breite des Childs
`layout_height`: Gewünscht Höhe des Childs Zulässige Werte sind
• `match_parent`: so gross wie möglich
(”am liebsten so gross wie mein Parent-Layout”)
• `wrap_content`: so klein wie möglich
(”nur gerade so gross, dass mein Inhalt Platz hat”)
• Eine Zahl inkl. gültiger Dimension
(eher unüblich; wenn verwendet, dann mit dp)

Linear Layout

Durch Gewichte können die Grössen der Child-Elemente beeinflusst werden
• `android:layout_weight="1"`
• Verwendung im Kombination mit `"wrap_content"`
• Gewichte wirken immer in Richtung der ”Orientierung”
• Kinder ohne Gewicht bekommen minimalen Platz, auf die restlichen Kinder wird die übrige
Fläche anteilsmässig entsprechend der Gewichte aufgeteilt

```
android:layout_margin="20dp" //Mit Layout
android:padding="20dp" //Ohne Layout
```

```
android:orientation="vertical
android:orientation="horizontal"
```

Frame Layout

Kinder werden übereinander angeordnet `android:translationZ`

Relative Layout

Kinder werden relativ zueinander angeordnet `android:layout_toEndOf`
Positions the start edge of this view to the end of the given anchor view ID.
`android:layout_toLeftOf`
Positions the right edge of this view to the left of the given anchor view ID.
`android:layout_toRightOf`
Positions the left edge of this view to the right of the given anchor view ID.
`android:layout_toStartOf`
Positions the end edge of this view to the start of the given anchor view ID.

Constraint Layout

Das modernste und flexibelste Layout in Android
• Teil von Android Jetpack / AndroidX
• Speziell für grosse, komplexe Layouts mit flacher Hierarchie entworfen
• Die Flexibilität bringt leider auch eine steile Lernkurve mit sich
Grundidee ist ähnlich zum Relative Layout
• Definieren von Beziehungen zwischen Views
• Pro View muss mindestens eine horizontale und eine vertikale Einschränkung definiert werden
Guter Support in Android Studio

Animationen

Verwendung der TransitionManager-Klasse
Zwei Layouts nötig: Start und Ende der Animation

```
private boolean displaysFirstLayout = true;
// innerhalb von onCreate()
setContentView(R.layout.activity_constraint);
TextView textViewA = findViewById(R.id.txtA);
ConstraintLayout constraintLayout =
    findViewById(R.id.constraint_layout);
textViewA.setOnClickListener(v -> {
    int targetLayout = displaysFirstLayout ?
                       R.layout.activity_constraint_2 :
                       R.layout.activity_constraint;
    TransitionManager.beginDelayedTransition(constraintLayout);
    ConstraintSet constraintSet = new ConstraintSet();
    constraintSet.load(this, targetLayout);
    constraintSet.applyTo(constraintLayout);
    displaysFirstLayout = !displaysFirstLayout;
});
```

Widget (Controls)

Ein Sammelbegriff für visuelle Elemente des User Interfaces
Basisklasse ist View, nicht Widget

```
<TextView
android:text="TextView"
android:textSize="20sp"
android:textStyle="bold"
android:typeface="monospace"
android:textColor="@android:color/white"
android:background="@color/colorPrimaryDark"
android:drawableEnd="@drawable/ic_emoji"
android:drawableTint="@android:color/white"/>
<ImageView
android:layout_height="80dp"
android:src="@drawable/ic_emoji"
android:scaleType="fitCenter"
android:tint="@color/colorPrimaryDark" />
<Button
android:text="Button"
android:drawableEnd="@drawable/ic_emoji"
android:drawableTint="@color/colorPrimary"/>
<ImageButton
android:layout_height="30dp"
android:src="@drawable/ic_emoji"
android:scaleType="fitCenter"
android:tint="@color/colorPrimaryDark" />
```

<EditText />

EditText dient als Eingabefeld für Texte und Zahlen
android:inputType beeinflusst Verhalten und aussehen
• Format (text, phone, date, textPassword, …)
• Korrekturoptionen (textAutoComplete, textAutoCorrect, …)
• Darstellung (textCapWords, textCapSentences, …)
• Mehrzeiligkeit (textMultiLine)
inputType-Werte sind kombinierbar, z.B.:
`android:inputType="textCapCharacters|textAutoCorrect"`
Abhängig vom Typ wird passendes Keyboard angezeigt

Dazu muss eine Implementierung von TextWatcher als Listener registriert werden:
`myEditText.addTextChangedListener(new TextWatcher() { … })`
Das Interface umfasst 3 Methoden
• beforeTextChanged: wird aufgerufen, bevor der Text geändert wird (wir sehen noch den alten Text)
• onTextChanged: wird aufgerufen, sobald der Text geändert hat (meistens interessiert uns nur diese Methode)
• afterTextChanged: wird aufgerufen, nachdem der Text geändert wurde (hier haben wir noch die Chance den Text anzupassen → Achtung vor Endlosschleife!)
Mit setError() wird eine Nachricht gesetzt, bei jeder Änderung wird diese zurückgesetzt

```
EditText passwordInput = findViewById(R.id.edit_password);
passwordInput.addTextChangedListener(new TextWatcher() {
    // Methoden gekürzt
    @Override
    public void afterTextChanged(Editable editable) {
        if (editable.length() < 8) {
            passwordInput.setError("Passwort zu kurz.");
        }
    }
});
```

Toasts

Einfache Rückmeldung zu einem Vorgang
• Darstellung in einem Popup-Fenster
• Keine Interaktion für Benutzer möglich
• Verschwinden nach kurzer Zeit automatisch

```
Context context = this;
String text = "MGE rocks!";
int duration = Toast.LENGTH_SHORT;
Toast toast = Toast.makeText(context, text, duration);
toast.show();
```

Snackbars

Selber Einsatzzweck wie Toasts
Moderne Alternative für Toasts
Interaktionen mit Benutzer möglich

```
ViewGroup parent = findViewById(R.id.llSystemWidgets);
String text = "MGE rocks!";
int duration = Snackbar.LENGTH_LONG;
String action = "Schliessen";
Snackbar snackbar = Snackbar.make(parent,text, duration);
snackbar.setAction(action, v -> { snackbar.dismiss(); });
snackbar.show();
```

Dialoge

Dialoge erzwingen eine Aktion vom Benutzer
Füllen Screen nicht vollständig
Viele Anpassungsmöglichkeiten

```
AlertDialog dialog;
dialog = new AlertDialog.Builder(this)
.setTitle("Beispiel")
.setMessage("Gutes Beispiel?")
.setCancelable(false)
.setPositiveButton("Ja", (d, id) -> { ... })
.setNegativeButton("Nein", (d, id) -> { ... })
.create();
dialog.show();
```

Notifications

Nocitifcations(Mitteilung ausserhalb aktiver Nutzung, Darstellung an Statusbar / Notification Drawer / Heads-Up Notification / Lock Screen / App Icon Badge). NotificationCompat in AndroidX verwenden

```
// Konstanten und Variablen
final String CID = "MGE_Channel";
final String CNAME = "MGE Notifications";
final String CDESC = "Ein Channel für MGE ";
final int CIMP = NotificationManager.IMPORTANCE_HIGH;
int notificationId = 1;
// Manager aus AndroidX verwenden
NotificationManagerCompat manager;
manager = NotificationManagerCompat.from(this);
// Channel-Erzeugung ab Android 26
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
NotificationChannel channel;
channel = new NotificationChannel(CID, CNAME, CIMP);
channel.setDescription(CDESC);
manager.createNotificationChannel(channel);
}
// Notification erstellen und anzeigen
Notification notification;
notification = new NotificationCompat.Builder(this, CID)
.setSmallIcon(R.drawable.ic_emoji)
.setContentTitle("MGE")
.setContentText("MGE rocks!")
.build();
manager.notify(notificationId++, notification);
```

Menus

API wurde schon oft erweitert
Menu wird als Resource in res/menu definiert

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:app="http://schemas.android.com/apk/res-auto"
xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/menu_1"
    android:title="Menu 1"
    android:icon="@drawable/ic_bulb"
    app:showAsAction="always"/>
    ...
</menu>
```

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.menu_example, menu);
    return true;
}
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.menu_1:
        break;
        ...
    }
    return true;
}
```

ScrollView

- Die ScrollView ist ein spezielles Layout mit nur einem Kind-Element
- Es ergänzt eine Scrollbar und erlaubt das vertikale Scrolling des Inhaltes
- Horizontal Scrolling ist nur mit HorizontalScrollView möglich
- Alternative in AndroidX: NestedScrollView erlaubt beide Richtungen

```
<ScrollView
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <!-- Genau ein Kind hier -->
</ScrollView>
```

Collections

Ein Adapter vermittelt zwischen Darstellung und Datenquelle (Adapter-Pattern)
Dadurch bleibt unser Datenmodell frei von UI-Logik (gutes Software Engineering)

ListView und ArrayAdapter

```
<?xml version="1.0" encoding="utf-8"?>
<ListView xmlns:android="…"
    android:id="@+id/list_example"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
</ListView>
```

```
setContentView(R.layout.activity_main);
String[] data = new String[] { ... };
ArrayAdapter<String> adapter = new ArrayAdapter<>(
        this, android.R.layout.simple_list_item_1,
        android.R.id.text1, data);
ListView listView = findViewById(R.id.list_example);
listView.setAdapter(adapter);
```

Ein eigener ArrayAdapter

```
setContentView(R.layout.activity_main);
ArrayList<User> data = UserManager.getUsers();
UsersAdapter adapter = new UsersAdapter(this, data);
ListView listView = findViewById(R.id.list_example);
listView.setAdapter(adapter);

public class User {
    public String name;
    public int age;
    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
public class UsersAdapter extends ArrayAdapter<User> {
    public UsersAdapter(ctx c, ArrayList<User> users) {
        super(ctx, 0, users);
    }
    @Override
    public View getView(int pos, View view, ViewGroup parent) {
        if (view == null) {
            Context context = getContext();
            LayoutInflater inflater = LayoutInflater.from(context);
            view = inflater.inflate(
                android.R.layout.simple_list_item_2,
                parent, false);
        }
        TextView text1 = view.findViewById(android.R.id.text1);
        TextView text2 = view.findViewById(android.R.id.text2);
        User user = getItem(pos);
        text1.setText(user.name);
        text2.setText(user.age + " Jahre");
        return view;
    }
}
```

Viewholder

Es gibt noch zwei weitere, teure Operationen in unserem Beispiel

```
TextView text1 = view.findViewById(android.R.id.text1);
TextView text2 = view.findViewById(android.R.id.text2);
```

Effizienter wäre es, diese Objekt-Referenzen pro erzeugter View zu speichern
Genau dies ist die Idee hinter dem View Holder-Pattern

```
if (view == null) {
    //Inflate wie vorher
    viewHolder = new ViewHolder();
    viewHolder.text1 = view.findViewById(android.R.id.text1);
    viewHolder.text2 = view.findViewById(android.R.id.text2);
    view.setTag(viewHolder);
} else {
    viewHolder = (ViewHolder)convertView.getTag();
}
```

RecyclerView

Die RecyclerView ist eine moderne Alternative zu ListView und GridView
- Integriertes View-Recycling
- Erzwungene Verwendung von View Holdern
- Weniger Overhead im eigenen Code
- Layout-Flexibilität durch LayoutManager
- Sie ist, ihr ahnt es schon, Teil von AndroidX
- Die Verwendung wird von Google empfohlen

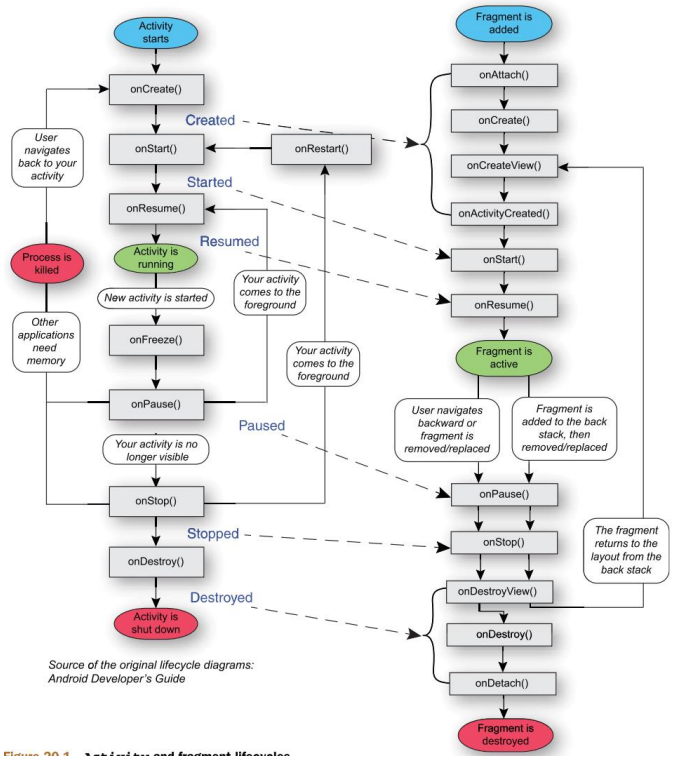
```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/recycler_view"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
</androidx.recyclerview.widget.RecyclerView>
```

```
    //MainActivity
    setContentView(R.layout.activity_recyclerview);
    RecyclerView recyclerView = findViewById(R.id.recycler_view);
    RecyclerView.LayoutManager layoutManager;
    layoutManager = new LinearLayoutManager(this);
    recyclerView.setLayoutManager(layoutManager);
    ArrayList<User> data = UserManager.getUsers();
    UsersAdapter adapter = new UsersAdapter(data);
    recyclerView.setAdapter(adapter);

public class UsersAdapter extends RecyclerView.Adapter<ViewHolder>
    private ArrayList<User> users;
    @Override
    public ViewHolder onCreateViewHolder(ViewGroup parent, int vt)
        Context context = parent.getContext();
        LayoutInflater inflater = LayoutInflater.from(context);
        View view = inflater.inflate(android.R.layout.simple_list_
        parent,false);
        return new ViewHolder(view,
            view.findViewById(android.R.id.text1),
            view.findViewById(android.R.id.text2)
        );
    }
    @Override
    public void onBindViewHolder(ViewHolder holder, int position)
        User user = this.users.get(position);
        holder.text1.setText(user.name);
        holder.text2.setText(user.age + " Jahre");
    }
    @Override
    public int getItemCount() {
        return this.users.size();
    }
}

private class ViewHolder {
    TextView text1;
    TextView text2;
}
```

Fragments

Activities können nicht kombiniert werden – dafür aber Fragments Zusätzliche Callbacks gegenüber Activity
- onAttach: Fragment an Activity angehängt
- onCreateView: UI des Fragments erstellen
- onActivityCreated: Activity wurde erzeugt
- onDestroyView: Gegenstück zu onCreateView

• onDetach: Gegenstück zu onAttach



Statische Einbindung

```
<fragment android:name="(…).OutputFragment"
android:id="@+id/main_fragment_output"
android:layout_width="match_parent"
android:layout_height="match_parent" />
```

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

Dynamische Einbindung

Vorteil: Austauschbarkeit, keine direkten Abhängigkeiten zu Activities

```
<LinearLayout xmlns:android="(…)"
android:layout_width="match_parent"
android:layout_height="match_parent">
    <FrameLayout android:id="@+id/main_fragment_container"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
</LinearLayout>
```

```
FragmentManager mgr = getSupportFragmentManager();
FragmentTransaction trans = mgr.beginTransaction();
OutputFragment fragment = new OutputFragment();
trans.add(R.id.main_fragment_container, fragment);
trans.commit();
```

Kommunikation

Parameter & Methoden: Activity -> Fragment
Callback: Activity <- Fragment

```
fragment = OutputFragment.create("Initial Value");
...
button.setOnClickListener(v -> {
    fragment.updateText("Updated value");});
```

```
public class OutputFragment extends Fragment {
    private TextView textOutput;
    public static OutputFragment create(String text) {
        Bundle args = new Bundle();
        args.putString("Key", text);
        OutputFragment fragment = new OutputFragment();
        fragment.setArguments(args);
        return fragment;
    }
    @Override
    public View onCreateView(...) {
        View fragment = inflater.inflate(...);
        textOutput = fragment.findViewById(R.id.output_text);
        String param = getArguments().getString("Key");
        updateText(param);
        return fragment;
    }
    public void updateText(String text) {
        textOutput.setText(text);
    }
}
```

Callback Interfaces

```
public class MainActivity extends AppCompatActivity
implements OutputFragmentCallback {
...
public interface OutputFragmentCallback {
    void onTextTapped(String text);
}
...
public class OutputFragment extends Fragment {
    private OutputFragmentCallback callback;
    @Override
    public void onAttach(Context context) {
        super.onAttach(context);
        try {
            callback = (OutputFragmentCallback) context;
        } catch (ClassCastException e) {
            throw new ClassCastException("...");
        }
    }
    @Override
    public View onCreateView(...) {
        View fragment = inflater.inflate(...);
        textOutput = fragment.findViewById(R.id.output_text);
        textOutput.setOnClickListener(v -> {
            callback.onTextTapped("…");
        });
        return fragment;
    }
}
```

Fragmente verschachteln

Unterschied: `getChildFragmentManager()` anstelle von `getSupportFragmentManager()`

Wann sind Fragments sinnvoll?

Eintrittspunkt -> zwingend Activity

Styling

Widgets können im XML gestylt werden. Führt aber zu Code-Duplizierung, Inkonsistenzen und Unübersichtlichkeit. Mit Styles können wir Formatierungen wiederverwendbar machen

```
<style name="HeaderText.Big">
    <item name="android:textSize">40sp</item>
</style>
```

Theme

Themes sind spezielle Styles, die für eine ganze App oder einzelne Activities gelten

```
<style name="AppTheme" parent="…">
    <item name="android:textViewStyle">@style/MyText</item>
</style>
<style name="MyText">
    <item name="android:textSize">24sp</item>
    ...
</style>
```

Variante 1: Manifest

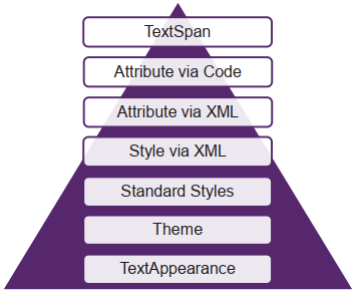
- Für die ganze App (application)
- Für einzelne Activities (activity)

```
<application ... android:theme="@style/AppTheme">
    <activity ... android:theme="@style/AnotherAppTheme" />
</application>
```

Variante 2: Activity-Code

- Innerhalb von onCreate()
- Wichtig: vor setContentView()

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    setTheme(R.style.AnotherAppTheme);
    setContentView(R.layout.activity_styling);
}
```



Material

Eine Design Language ist eine Hilfestellung für den Designprozess. Beschreibt wie Teile aussehen und sich verhalten. Ziel ist ein systemweites, konsistentes und benutzbares Look and Feel.
Material Design ist eine Design Language von Google mit physikalischem Material als Vorbild. Oberflächen erinnern an Papier und Tinte. Materialen reflektieren und werfen Schatten. Basiert auf Print-Medien. Bewegungen bedeuten Aktionen.
Material ist immer 1dp dick. Material hat eine unendliche Auflösung. Material kann sich verändern und bewegen. Die Flughöhe definiert, welches Element welche überlagert.
8dp-Raster

```
<style name="AppTheme" parent="...">
    <item name="colorPrimary">...</item>
    <item name="colorPrimaryDark">...</item>
    <item name="colorAccent">...</item>
</style>
```

Styles für Texte

```
style="@style/TextAppearance.MaterialComponents.Headline3"/>
```

—

Berechtigung

• Apps dürfen nur Aktionen ausführen, die andere Dienste nicht negativ beeinflussen (Sandbox)
• Vor riskante Operationen müssen Berechtigungen eingeholt werden
Es gibt zwei Arten von Berechtigungen:
Normal: werden durch das System erteilt
Gefährlich: werden durch den Benutzer erteilt

Zugriff auf System APIs: Internet, WiFi, Bluetooth
Zugriff auf sensitive Daten: Telefonbuch, Kalender
Zugriff auf Hardware: Kamera, Lokalisierung

Empfehlung: vor jedem Zugriff auf geschützte APIs Status überprüfen
Andernfalls droht SecurityException
Best Practices
- Nur anfordern, was auch benötigt wird
- Im Kontext der Verwendung anfordern
- Transparente Erklärungen
- Abbruch ermöglichen
- Verweigerung berücksichtigen

Manifest

Benötigte Berechtigungen müssen im Manifest deklariert werden
- Knoten <uses-permission
- Obergrenze für API-Level definierbar

Hinweise auf benötigte Features für Filterung im Google Play Store
- Knoten <uses-feature>
- Optional, aber empfohlen
- required="true" → notwendig (Standard)

```
private static final int CALLBACK_CODE = 1;
String permission = Manifest.permission.CALL_PHONE;
// Aktuellen Status prüfen (AndroidX)
int status = ContextCompat.checkSelfPermission(this, permission);
if (status != PackageManager.PERMISSION_GRANTED) {
    if (shouldShowRequestPermissionRationale(permission)) {
    // Erklärung für Benutzer anzeigen (Wozu nötig?)
    }
    // Berechtigung beim Benutzer anfordern
    requestPermissions(
    new String[]{ permission },
    CALLBACK_CODE);
}

@Override
public void onRequestPermissionsResult(
int requestCode, String[] permissions, int[] results) {
    if (requestCode != CALLBACK_CODE)
        return;
    if (results.length == 0)
        return; // Anfrage abgebrochen
    if (results[0] == PackageManager.PERMISSION_GRANTED) {
        // Berechtigung erteilt
    } else {
        // Berechtigung verweigert
    }
}
```

Persistenz

Speicherarten

Interner Speicher

- Stets verfügbar
- Geschützter Speicherbereich pro App
- Speicherplatz begrenzt
- Für app-interne Daten

Externer Speicher

- Nicht immer verfügbar
- Oft ein Wechseldatenträger
- Emulation durch Android möglich
- Speicherplatz begrenzt (aber meist grösser)
- Primär für geteilte Daten

App-spezifische Dateien

- Eigene, proprietäre Datenformate
- Interner oder externer Speicher
- Bei Deinstallation der App gelöscht
- Geschützt vor fremdem Zugriff

```
// Schreiben
File folder = getFilesDir();
File file = new File(folder, "my_file.txt");
String input = "MGE Beispiel";
FileOutputStream outputStream = new FileOutputStream(file);
outputStream.write(input.getBytes());
outputStream.close();
// Dateien anzeigen
for(File fileInFolder : folder.listFiles()) {
    Log.d("MGE.V05", "File: " + fileInFolder.getName());
}
// Lesen
int length = (int) file.length();byte[] bytes = new byte[length];
FileInputStream inputStream = new FileInputStream(file);
inputStream.read(bytes);
inputStream.close();
String output = new String(bytes);
```

Preferences

- Key-Value-Paare
- Interner Speicher
- Bei Deinstallation der App gelöscht
- Keine Berechtigungen nötig
- Zugriff via SharedPreferences-Objekte

```
String file = "ch.ost.rj.mge.v05.myapplication.preferences";
String key1 = "my.key.1";
String key2 = "my.key.2";
String key3 = "my.key.3";
int mode = Context.MODE_PRIVATE;
// Objekt abholen
SharedPreferences preferences;
preferences = getSharedPreferences(file, mode);
// Schreiben
SharedPreferences.Editor editor = preferences.edit();
editor.putString(key1, "MGE Beispiel");
editor.putBoolean(key2, true);
editor.putInt(key3, 42);
editor.commit();
// Lesen
String value1 = preferences.getString(key1, "default");
boolean value2 = preferences.getBoolean(key2, false);
int value3 = preferences.getBoolean(key3, 0);
```

Content Providers

Datenquellen für andere Apps(Kalender, Kontakte, Medien, etc.)
Client-Server Modell
Client: Content Resolver
Server: Content Provider

Content Resolvers

```
Cursor cursor = getContentResolver().query(uri, projection,
selection, selectionArgs, sortOrder);
```

Medien

• Teilbare Bilder, Videos und Musik
• Externer Speicher
• Bleiben bei Deinstallation der App erhalten
• Zugriff via MediaStore (Content Provider), Berechtigung nötig

Dokumente

• Teilbare Dokumente wie PDF, ZIP, etc • Externer Speicher
• Bleiben bei Deinstallation der App erhalten
• Zugriff via Storage Access Framework
• Auswahl von Dokumenten via "Picker", keine Berechtigung nötig

```
private static final int CREATE_DOCUMENT_CODE = 1;
private static final int OPEN_DOCUMENT_CODE = 2;
private static final String FILE_NAME = "my_file.txt";
private static final String FILE_TYPE = "text/plain";
// Intent zum Schreiben eines Dokuments
Intent intent = new Intent(Intent.ACTION_CREATE_DOCUMENT);
intent.addCategory(Intent.CATEGORY_OPENABLE);
intent.setType(FILE_TYPE);
intent.putExtra(Intent.EXTRA_TITLE, FILE_NAME);
startActivityForResult(intent, CREATE_DOCUMENT_CODE);
// Intent zum Öffnen eines Dokuments
Intent intent = new Intent(Intent.ACTION_OPEN_DOCUMENT);
```

```
intent.addCategory(Intent.CATEGORY_OPENABLE);
intent.setType(FILE_TYPE);
startActivityForResult(intent, OPEN_DOCUMENT_CODE);

@Override
public void onActivityResult(int req, int res, Intent data) {
    super.onActivityResult(req, res, data);
    switch(req) {
        case CREATE_DOCUMENT_CODE:
            if (res == Activity.RESULT_OK) {
                Uri uri = data.getData();
                // Mit Content Resolver Uri verarbeiten
            }
            break;
        case OPEN_DOCUMENT_CODE:
            if (res == Activity.RESULT_OK) {
                Uri uri = data.getData();
                // Mit Content Resolver Uri verarbeiten
            }
    }
}
```

Datenbanken

- Strukturierte Daten
- Interner Speicher
- Bei Deinstallation der App gelöscht
- Keine Berechtigungen nötig
- Zugriff via SQLite API oder Room aus AndroidX / Jetpack

Datenbanken – Room

- ORM (Object Relational Mapping)
- Weniger Code nötig (z.B. Mapping)
- Überprüfung von Queries zur Compile-Zeit
- Kompatibilität mit Jetpack-Komponenten

```
@Entity
public class Entry {
    @PrimaryKey(autoGenerate = true)
    public int id;
    @ColumnInfo
    public String content;
}

@Dao
public interface EntryDao {
    @Query("SELECT * FROM entry")
    List<Entry> getEntries();
    @Insert
    void insert(Entry entry);
    @Delete
    void delete(Entry entry);
}

@Dao
public interface EntryDao {
    @Query("SELECT * FROM entry")
    List<Entry> getEntries();
    @Insert
    void insert(Entry entry);
    @Delete
    void delete(Entry entry);
}

// Erzeugung DB-Objekt (MainActivity.java)
EntryDatabase db = Room.databaseBuilder(
this, EntryDatabase.class, "room.db").build();
EntryDao dao = db.entryDao();
// Daten einfügen
Entry entry = new Entry();
entry.content = "MGE Vorlesung";
dao.insert(entry);
// Daten auslesen
List<Entry> entries = dao.getEntries();
for (Entry entry : entries) {
    Log.d(null, + entry.id + " | " + entry.content);
}
// Aufräumen
db.close();
```

Debugging

- Device File Explorer in Android Studio

Hardwarezugriff

Sensor Framework

- Framework für verschiedene Sensoren
- `SensorManager` als Einstiegspunkt
- `Sensor` als Repräsentant für realen Sensor
- `SensorEvent` enthält Werte des Sensors
- `SensorEventListener` für Callbacks

Verzögerung beeinflusst Energieverbrauch • `SENSOR_DELAY_FASTEST` (0ms)
- `SENSOR_DELAY_GAME` (20ms)
- `SENSOR_DELAY_UI` (60ms)
- `SENSOR_DELAY_NORMAL` (200ms)

Änderung der Genauigkeit
- `SENSOR_STATUS_ACCURACY_HIGH`
- `SENSOR_STATUS_ACCURACY_MEDIUM`
- `SENSOR_STATUS_ACCURACY_LOW`
- `SENSOR_STATUS_ACCURACY_UNRELIABLE`

```
// Bei Sensor für Änderungen registrieren
String service = Context.SENSOR_SERVICE;
int type = Sensor.TYPE_LIGHT;
int delay = SensorManager.SENSOR_DELAY_NORMAL;
SensorManager mgr = (SensorManager)getSystemService(service);
Sensor sensor = mgr.getDefaultSensor(type);
mgr.registerListener(this, sensor, delay);
// Implementierung von SensorEventListener
@Override
public void onSensorChanged(SensorEvent sensorEvent) {
float lux = sensorEvent.values[0];
    Log.d(null, lux + " lux");
}
@Override
public void onAccuracyChanged(Sensor sensor, int i) { }
```

Vibration -> Verwendung der Klasse Vibrator -> Berechtigung nötig (VIBRATE). Sensor-Listener sollten in onPause() und onResume() abgemeldet/angemeldet werden

```
String service = Context.VIBRATOR_SERVICE;
int maxAmplitude = 255;
Vibrator vibrator = (Vibrator) getSystemService(service);
// Ab API 1
vibrator.vibrate(500);
Vibrator.cancel();
// Ab API 26
long[] durs = new long[]{ 500, 500, 500, 500, 500 };
int[] amps = new int[]{ 50, 100, 150, 200, 255 };
VibrationEffect effect;
effect = VibrationEffect.createOneShot(500, 255);
effect = VibrationEffect.createWaveform(durs, amps, -1);
vibrator.vibrate(effect);
// Ab API 29
int effectId = VibrationEffect.EFFECT_DOUBLE_CLICK;
effect = VibrationEffect.createPredefined(effectId);
vibrator.vibrate(effect);
```

Status der Internet-Verbindung

Klasse ConnectivityManager
Üblich sind zwei Kanäle (Mobile & Wifi)
- Android nutzt automatisch den besten Kanal
- Höhere Geschwindigkeit
- Bessere Signalqualität
- Kein Roaming Needs `ACCESS_NETWORK_STATE`

```
String service = Context.CONNECTIVITY_SERVICE;
ConnectivityManager manager;
manager = (ConnectivityManager) getSystemService(service);
// Aktive Verbindung prüfen
NetworkInfo activeNetwork = manager.getActiveNetworkInfo();
if (activeNetwork != null) {
    int type = activeNetwork.getType();
    Log.d(null, "Active connection: " + type);
}
// Verbindungen prüfen
for (Network network : manager.getAllNetworks()) {
    NetworkInfo info = manager.getNetworkInfo(network);
    boolean state = info.isConnected();
    if (info.getType() == ConnectivityManager.TYPE_WIFI) {
        Log.d(null, "WiFi is connected: " + state);
    }
    if (info.getType() == ConnectivityManager.TYPE_MOBILE) {
        Log.d(null, "Mobile is connected: " + state);
    }
}
```

Kamera

- Option 1: Apps via Intents starten
- Option 2: Camera-API
- Option 3: camera2-API
- Option 4: CameraX-API (AndroidX)

Vorteile von Option 1:
- Keine Berechtigungen nötig
- Weniger Logik und Komplexität

Vorteile von Optionen 2 bis 4
- Kamera Teil der eigenen App
- Mehr Möglichkeiten und Kontroll

Architektur

- Presentation
  - Darstellung und Interaktion mit Benutzer
  - Typischerweise stark an UI-Toolkit gebunden
  - Zugriff auf Domain-Schicht
- Domain
  - Businesslogik und Domänenklassen
  - Keine UI-Funktionalität
  - Wenig externe Abhängigkeiten
  - Einfach zu testen
- Data
  - Speicherung der Daten
  - Stellt Daten der Domain zur Verfügung
  - Auch Persistenz oder Datenhaltung genannt

Observer Pattern
• Zwei Rollen
• Subject (wird beobachtet, z.B. Model-Klasse)
• Observer (beobachtet, z.B. View-Klasse)

Model-View-Controller
• Model beinhaltet die Daten (Java-Klassen)
• View liest die Daten des Modells und zeigt diese an (View, Adapter)
• Controller erhält Events der View und manipuliert das Model (Activity, Fragment)

Application

- Der Eltern-Knoten unserer Komponenten im Manifest heisst application
- Zu diesem Knoten wird beim Start eine Instanz der Application-Klasse erzeugt
- Eigene Ableitungen der Klasse sind möglich
  * Einmalige Initialisierungen
  * Erzeugung von Singleton-Objekten
  * Zugriff auf globale Objekte
- Lifecycle-Methoden werden von Android zu bestimmten Zeitpunkten aufgerufen

- onCreate()
  - Einmalig nach dem Start der App
  - Vor Erzeugung anderer App-Komponenten
- onTerminate()
  - Wird auf realen Geräten nie aufgerufen
  - Reminder: Apps können stets beendet werden
- onConfigurationChanged(newConfig)
  - Bei Änderungen der System-Konfiguration
  - Parameter enthält neue Konfiguration
  - Beispiel: Rotation des Gerätes
- onLowMemory()
  - Bei Speicherknappheit des Systems
  - Hinweis auf mögliche Terminierung der App
- onTrimMemory(level)
  - Bei geeigneten Momenten für Aufräumaktionen
  - Parameter gibt Hinweise auf Auslöser
  - Beispiel: App geht in den Hintergrund (`TRIM_MEMORY_UI_HIDDEN`)

Via Application-Klasse kann der Lebenszyklus aller Activities überwacht werden

Context

Context ist eine abstrakte Klasse der SDK mit über 50 Ableitungen (!), beispielsweise
• Activity
• Application

- Context-Objekte haben eine eingeschränkte Lebensdauer
  - Angeforderte Ressourcen werden mit dem zugehörigen Context freigegeben
  - Beispiel: Activity-Context via this
- Das Application Context-Objekt ist während der ganzen App-Lebensdauer gültig
  - Abholen via Context.getApplicationContext()
  - Eingeschränkt verwendbar für UI-Aktionen
  - Vorsicht vor Memory Leaks

Broadcasts

Austausch von Meldungen zwischen Apps Zwei Arten von Broadcasts:
• **Lokal**: innerhalb einer App • **Global**: innerhalb des ganzen Systems

Broadcasts empfangen
Statische Registrierung im Manifest vs. **Dynamische Registrierung** via Context. Implizite Broadcasts können nur von dynamischen Registrierungen empfangen werden, wegen einer Änderung im API 26. Explizite Broadcasts können via statische und dynamische Registrierung emfpangen werden.
Broadcasts versenden

- Broadcasts sind normale Intent-Objekte
- Die Action im Intent definiert den Ereignistyp
- Parameter sind als Extras möglich

Für lokale Nachrichten immer LocalBroadcastManager verwenden

Best Practices

- Dynamische Registrierung bevorzugen
- Lokale Broadcasts bevorzugen
- Keine sensitiven Daten in Broadcasts übermitteln
- App ID in eigene Broadcast-Actions integrieren
- Schnelle Rückkehr aus onReceive()
- Falls App nicht aktiv: keine Activity starten

Services

Ab API 30 werden die normalen Java-Klassen für Background Threads empfohlen

- Ausführung von Aktionen im Hintergrund
- Lebenszyklus unabhängig von App
- Kein oder reduziertes UI (Notification)
- Werden auf Main-Thread ausgeführt

Started Services

- Für einmalige Aktionen
- Laufen potentiell endlos weiter

Bound Services

- Für Aufgaben über längere Zeitdauer
- Client-Server ähnliche Kommunikation
- Austausch von Daten fester Bestandteil
- Mehrere Clients gleichzeitig möglich
- Nach Verbindungsende zu letztem Client wird der Service automatisch gestoppt

Build & Deployment

Android führt anstelle von Byte Code optimierten DEX-Code aus

APK

- Apps werden aus APK-Dateien installiert
- APKs aus dem Play Store sind signiert
- Limit des Google Play Store: 100 MB
- Optimierung 1: APK Splitting
- Optimierung 2: APK Expansion Files

AAB

- Nachfolger des APK-Formats
- Container mit jeglichen Inhalten eines Apps
- Dynamische Erzeugung von APKs
- Google in Besitz des Signaturschlüssels
- Ab 2021 zwingend im Google Play Store

Binding

Unterschied zu View Binding View Binding: Vereinfacht Zugriff View vs Data Binding: Vereinfacht Zugriff Data

View Binding

Vereinfacht den Zugriff auf View-Elemente
Namensgebung der Klassen:
• Layout-Name als Camel Case + Binding
• `activity_main.xml` -> ActivityMainBinding

```
android {
  buildFeatures {
    viewBinding true
  }
}
```

```
public class MainActivity extends AppCompatActivity {
    private ActivityMainBinding binding;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    LayoutInflater inflater = getLayoutInflater();
    binding = ActivityMainBinding.inflate(inflater);
    setContentView(binding.getRoot());
    binding.buttonHello.setOnClickListener(v -> {});
    }
}
```

Data Binding

Erlaubt im XML Zugriff auf Objekte

```
public class MainActivity extends AppCompatActivity {
    private ActivityMainBinding binding;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        binding = DataBindingUtil.setContentView(
        this, R.layout.activity_main);
        User user = new User("Thomas", "Kälin");
        binding.setUser(user);
    }
}
```

```
<layout xmlns:android="…">
    <data>
        <variable name="user" type="ch.ost.rj.mge.v07.User"/>
    </data>
    <LinearLayout
```

```
    android:layout_width="match_parent"
    android:layout_height="match_parent">
        <TextView
        android:id="@+id/first"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@{user.firstName}" />
    </LinearLayout>
</layout>
```

Event Handling

```
<data>
    <variable name="handler" type="(…).EventHandler"/>
</data>
...
<Button
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:text="Method Reference"
android:onClick="@{handler::doSomething}"/>
<Button
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:text="Listener Binding"
android:onClick='@{v -> handler.doSomething(v, "...")}'/>
```

```
public class EventHandler {
    public void doSomething(View view) {
    }
    public void doSomething(View view, String text) {
    }
}
```

Ein in Data Bindings verwendetes Objekt wird **nicht** automatisch observierbar
Für die automatische Aktualisierung der View muss die Datenquelle angepasst werden Observable Fields für einfache Datentypen Observable Objects für eigene Klassen

```
// Observable Fields
public class User {
    public final ObservableField<String> firstName = new ...();
    public final ObservableField<String> lastName = new ...();
    public final ObservableInt age = new ...();
}

public class User extends BaseObservable {
...
    notifyPropertyChanged(BR.firstName);
```

Two-Way Bindings

Konsistenz von Model und View Notation: = vor der Binding Expression
`android:value="@={user.age}"`
Chancen
- Schlankere Activities und Fragmente
- Ermöglicht eigene MVVM-Implementierung
- Bessere Testbarkeit des Codes

Risiken
- Model mit Android-Details verunreinigt
- Zu viel Logik im Layout (Expression Language)
- Bei Fehlern erschwertes Debugging
- Erhöhter Zeitbedarf für Kompilierung
- Gefahr für ”unsichtbare Observer”

MVVM

- Das Model enthält Daten- und Domänenklassen (Businesslogik)
- Die View umfasst die grafische Benutzeroberfläche & Benutzereingaben
- Das ViewModel enthält die Logik des UI und vermittelt zwischen Model und View
- ViewModel ist einfacher zu testen. View kümmert sich rein um visuelles. Änderungen am Model haben keine direkte Auswirkung auf die View.

```
public class UserActivity extends AppCompatActivity {
    private ActivityUserBinding binding;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        User user = new User("Thomas", "Kälin", 36);
        UserViewModel viewModel = new UserViewModel(user);
        binding = ActivityUserBinding.inflate(...);
        binding.setVm(viewModel);
        setContentView(binding.getRoot());
    }
}

public class UserViewModel {
    private final User user;
    public final ObservableField<String> name = new O...<>();
    public final ObservableInt age = new ObservableInt();
        public ViewModelObservableFields(User user) {
        this.user = user;
        name.set(user.name);
        age.set(user.age);
    }
    public void incrementAge() {
        int newAge = age.get() + 1;
        age.set(newAge);
    }
    public void save() {
        user.name = name.get();
        user.age = age.get();
    }
}
```

Unsere MVVM-Implementierung ist elegant, besitzt aber zwei Probleme:
1. Data Bindings auf Observable Fields bleiben auch dann aktiv, wenn die zugehörige Activity gerade nicht im Vordergrund ist
2. Bei einer Rotation des Gerätes wird das View Model neu erzeugt – die darin verwalteten Daten gehen somit verloren

LifecycleObserver

```
public class MyLocationListener implements LifecycleObserver {
    private final Lifecycle lifecycle;
    private boolean enabled = false;
    public MyLocationListener(Lifecycle lifecycle,
    Consumer<Location> callback) {
        this.lifecycle = lifecycle;
        this.lifecycle.addObserver(this);
    }
    @OnLifecycleEvent(ON_START)
    void start() {
        if (enabled) {
            // Positionsbestimmung starten
        }
    }
    @OnLifecycleEvent(ON_STOP)
    void stop() {
        // Positionsbestimmung stoppen
    }
    public void enable() {
        enabled = true;
        if (lifecycle.getCurrentState().isAtLeast(STARTED)) {
            start();
        }
    }
} Location.java
```

LiveData

LiveData ist ein lifecycle-aware Observable Alternative zu Observable Fields und Observable Objects
Als Datenquelle für Data Bindings verwendbar

Observer werden …
…nur benachrichtigt, wenn diese aktiv sind (STARTED, RESUMED)
…entfernt, wenn diese gestoppt wurden (DESTROYED)

```
//Viemodel
public final MutableLiveData<String> name =
new MutableLiveData<>();

// OnCreate
ViewModel vm = new ViewModel();
vm.name.observe(this, name -> { ... } );
```

MVVM im Eigenbau – Version 2

```
public class UserActivity extends AppCompatActivity {
    private ActivityUserBinding binding;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        User user = new User("Thomas", "Kälin", 36);
        UserViewModel viewModel = new UserViewModel(user);
        binding = ActivityUserBinding.inflate(...);
        binding.setVm(viewModel);
        binding.setLifecycleOwner(this);
        setContentView(binding.getRoot());
    }
}

public class UserViewModel {
    private final User user;
    public final MutableLiveData<String> name = new M...<>();
    public final MutableLiveData<Integer> age = new M...<>();
    public ViewModelObservableFields(User user) {
        this.user = user;
        name.setValue(user.name);
        age.setValue(user.age);
    }
    public void incrementAge() {
        int newAge = age.getValue() + 1;
        age.setValue(newAge);
    }
```

```
    public void save() {
        user.name = name.getValue();
        user.age = age.getValue();
    }
}
```

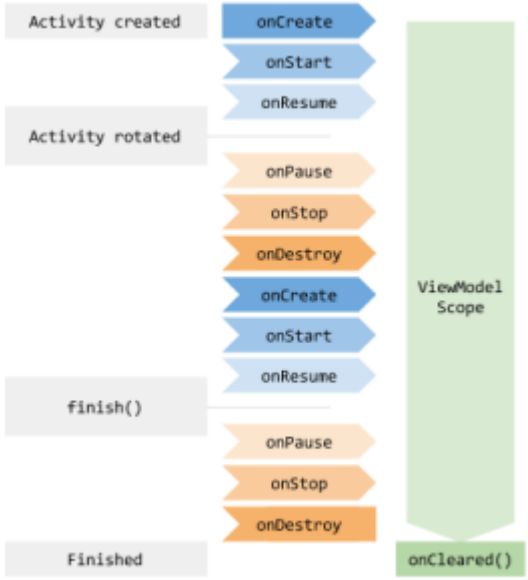
LiveData muss für die Observer im XML (Data Bindings) den zugehörigen LifecycleOwner kennen

ViewModel

Die abstrakte Klasse ViewModel löst unser letztes Problem (Rotation des Gerätes) ViewModel-Objekte hängen von LifecycleObjekten ab

- Mehrfache Erzeugung des ViewModel mit dem selben Lifecycle-Objekt liefert Singletont
- Erst bei Zerstörung des Lifecycle-Objektes wird auch das ViewModel-Objekt freigegeben

An App-Lebenszeit geknüpfte View Models sind möglich (via Application-Objekt) (Basisklasse AndroidViewModel)



MVVM im Eigenbau – Version 3

```
public class UserActivity extends AppCompatActivity {
    private ActivityUserBinding binding;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        User user = new User("Thomas", "Kälin", 36);
        UserViewModelFactory factory = new Use...Factory(user);
        UserViewModel viewModel = new ViewModelProvider(
        this, factory).get(UserViewModel.class);
        binding = ActivityUserBinding.inflate(...);
        binding.setVm(viewModel);
        binding.setLifecycleOwner(this);
        setContentView(binding.getRoot());
    }
}
public class UserViewModel extends ViewModel {
    // Rest wie zuvor
}
public class UserViewModelFactory
            implements ViewModelProvider.Factory {
    private final User user;
    public UserViewModelFactory(User user) {
        this.user = user;
    }
    @Override
    public <T extends ViewModel> T create(Class<T> class) {
        return (T) new UserViewModel(user);
    }
}
```

Lifecycle-Aware-Components reagieren auf Änderungen. Zustandslogik verschiebt sich vom Owner zum Observer.
Interaktion zwischen Fragemnts kann sehr komplex werden (viele Callbacks). ViewModels können helfen aber Fragments verlieren die Unabhängigkeit.

	Woche 07	Woche 02	Woche 05
	ViewModel	Saved instance state	Persistent storage
Storage location	in memory	serialized to disk	on disk or network
Survives configuration change	Yes	Yes	Yes
Survives system-initiated process death	No	Yes	Yes
Survives user complete activity dismissal/onFinish()	No	No	Yes
Data limitations	complex objects are fine, but space is limited by available memory	only for primitive types and simple, small objects such as String	only limited by disk space or cost / time of retrieval from the network resource
Read/write time	quick (memory access only)	slow (requires serialization/deserialization and disk access)	slow (requires disk access or network transaction)

Android Developer 2020: Saving States!

- Welche Methoden werden beim Starten der App aufgerufen?
  - `onCreate` → `onStart` → `onResume`
- Welche Methoden werden beim Screen des Screens aufgerufen?
  - `onPause` → `onStop`
- Welche Methoden werden beim Entsperren des Screens aufgerufen?
  - `onRestart` → `onStart` → `onResume`
- Welche Methoden werden einer Orientierungsänderung (Portrait zu Landscape) aufgerufen?
  - `onPause` → `onStop` → `onDestroy` → `onCreate` → `onStart` → `onResume`
- Welche Methoden werden beim Drücken des «Zurück Buttons» aufgerufen?
  - `onPause` → `onStop` → `onDestroy`

Übungen

«Schriftgrösse» nur einfluss auf sp
Starten einer Acitvity, welche nicht existiert, wirft eine Exception

```
Button browserButton = findViewById(R.id.buttonBrowser);
browserButton.setOnClickListener(v -> {
    Intent intent = new Intent(Intent.ACTION_VIEW,
                    Uri.parse("http://www.ost.ch"));
    if (intent.resolveActivity(getPackageManager()) != null) {
        startActivity(intent);
    }
});
```

Wenn onClick Events im XML deklariert werden, werden die zur Laufzeit via Reflection auf eine gleichnamige Methode gemappt. Führt zu Exceptions zur Laufzeit bei falschem Namen.

```
final Runnable updateUi = new Runnable() {
// wie gehabt
};
Runnable background = new Runnable() {
@Override
public void run() {
    Thread.sleep(3000);
    Looper looper = Looper.getMainLooper();
    Handler handler = new Handler(looper);
    handler.post(updateUi);
}};
Thread thread = new Thread(background);
thread.start();
```