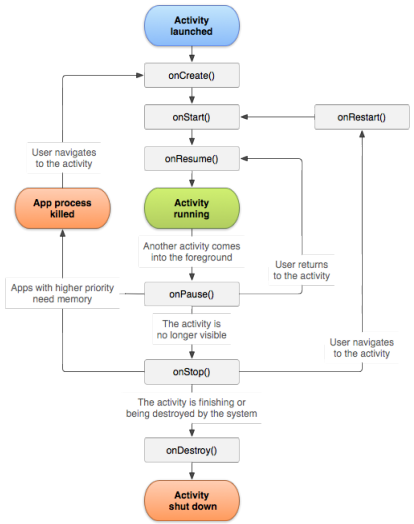
1 Allgemein

- Android ist Open Source unter Apache License
- Java 8 also Lambda, Default Methods aber keine Streams unterstützt
- Keine Registrierung und keine Lizenz, NDA nötig
- Apps bestehen aus **lose gekoppelten**, **wiederverwendbaren** Komponenten wie Activities, Services, Content Provider und Broadcast Receivers
- System hat feste Kontrolle über Applikationen und verwaltet den Lebenszyklus, Kommunikation zwischen Komponenten oder schliesst auch Apps wenn Speicher knapp wird.
- Komponenten anderer Apps können aufgerufen und Systemkomponenten ersetzt werden (Kamera-App, Browser-App)

2 Activities und Lebenszyklus

- App = GUI + Code



- **onCreate(Bundle b)** method, you perform basic application startup logic that should happen only once for the entire life of the activity. For example, your implementation of onCreate() might bind data to lists, associate the activity with a ViewModel
- **onStart()**: For example, this method is where the app initializes the code that maintains the UI.
- **onResume()**: This is the state in which the app interacts with the user. The app stays in this state until something happens to take focus away from the app. Such an event might be, for instance, receiving a phone call, the user's navigating to another activity, or the device screen's turning off.
- **onPause()**: The system calls this method as the first indication that the user is leaving your activity (though it does not always mean the activity is being destroyed); it indicates that the activity is no longer in the foreground (though it may still be visible if the user is in multi-window mode). Use the onPause() method to pause or adjust operations that should not continue (or should continue in moderation) while the Activity is in the Paused state, and that you expect to resume shortly. (save your app data HERE)
- **onStop()**: When your activity is no longer visible to the user, it has entered the Stopped state, and the system invokes the onStop() callback. This may occur, for example, when a newly launched activity covers the entire screen. The system may also call onStop() when the activity has finished running, and is about to be terminated. You should also use onStop() to perform relatively CPU-intensive shutdown operations. For example, if you can't find a more opportune time to save information to a database, you might do so during onStop().
- **onRestart()**: Called after onStop() when the current activity is being re-displayed to the user (the user has navigated back to it).
- **onDestroy()**: onDestroy() is called before the activity is destroyed. The system invokes this callback either because the activity is finishing (due to the user completely dismissing the activity or due to finish() being called on the activity), or the system is temporarily destroying the activity due to a configuration change (such as device rotation or multi-window mode)

```
public class Activity extends Activity {
  protected void onCreate(Bundle savedInstanceState);
  protected void onStart();
  protected void onRestart();
  protected void onResume();
  protected void onPause();
  protected void onStop();
  protected void onDestroy();
}
```

3 Activity Stack und Tasks

Eine Gruppe von Activities in einem Stack nennt man auch Task. Wenn aus einer Activity eine andere Activity gestartet wird, wird sie auf den Stack gelegt. Wenn dann eine andere Activity direkt gestartet wird, bildet diese einen anderen Task und hat einen anderen Stack.

- Activities (und Ressourcen, etc) werden in ein APK gepackt und installiert
- Wird eine Activity aktiv, wird pro APK ein Linux Prozess mit einem Thread gestartet
- Dieser Prozess führt alle Activites die in diesem APK enthalten sind aus
- Jedes APK wird unter einem eigenen Linux User installiert (isolation)
- JARs mit der Erweiterung .apk – JARs wiederum sind ZIP-Dateien
- Libraries, Ressourcen, Assets, Metadaten
- Kompilierte Java-Klassen im DEX-Format
- APKs können direkt installiert werden. Kann Play-Store umgehen. Muss vom User erlaubt sein
- Wird vom Build-System erstellt

4 Fragments Lifecycle

- onAttach: Fragment wird einer Activity hinzugefügt
- onCreateView: UI des Fragments erstellen
- onActivityCreated: wenn Activity onCreate fertig ist
- onDestroyView: Gegenstück zu onCreateView
- onDetach: Gegenstück zu onAttach

5 Intents

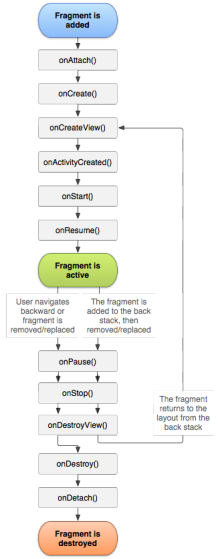
Kommunikation zwischen Komponenten erfolgt über Intents (Absicht, Vorhaben). Intent beschreibt, was gemacht werden soll: **expliziter Aufruf** einer Klasse oder **impliziter Aufruf** der passenden Komponente (Kamera, Email usw.). Bei implizitem Aufruf entscheidet System und ermöglicht **lose Koppelung** von Activities (Austauschbarkeit).
Apps können selbst wiederum Activities zur Verfügung stellen und so auch bestehende Applikationen ersetzen (z.B. alternative Kamera- oder E-Mail-App). Es können auch Daten auf dem Intent gesetzt werden und so der neuen Activity übergeben werden.

```
//Explizit, spezifische App
Intent i = new Intent(this, CalculateActivity.class)
//Implizit, eine Art von App, z.B. Kammera App
Intent i = new Intent(MediaStore.ACTION_IMAGE_CAPTURE)

i.putExtras(bundle);
startActivity(i);
```

```
//in onCreate(...)
Bundle b = getIntent().getExtras();

//generell
b.putInt("abc", 13);
b.putString("d", "hello");
(String) b.get("abc");
b.putSerializable("abc", list);
(List<String>) b.getSerializable("abc");
```

```
Intent dIntent = new Intent(this, DownloadService.class);
dIntent.setData(Uri.parse(fileUrl));
startService(dIntent);
```

```
startActivity(intent); // Kontrolle übergeben
startActivityForResult(intent, SOME_ID); // ink. Rückgabewert

@Override
protected void onActivityResult(int request, int result, Intent data) {
  if (result == Activity.RESULT_OK && request == SOME_ID) {
    // Resultat verarbeiten, Activities mit SOME_ID unterscheiden
    // EditText (Textfeld)
    String from = fromEditText.getText().toString();
  }
}
```



Intents starten nicht nur Activities, sondern werden auch für die Kommunikation mit anderen Komponenten wie Services oder Broadcasts verwendet.
Broadcasts werden über einen Messagebus an alle interessierten Apps im System versendet.
Über Broadcasts werden auch Informationen von Android and die Apps weitergegeben, z.B. wenn die Batterie leer ist oder geladen wird.

5.1 API Level

- API Level sind die für Entwickler relevanten Versionsnummern
- Eine Version der Plattform unterstützt immer auch alle älteren APIs
- Wenn eine Applikation ein Feature einer neueren Android Version verwendet, dann muss die minimale API entsprechend erhöht werden
- Um ein möglichst grosses Publikum zu erreichen, solle der API Level so niedrig wie möglich angesetzt werden
- Allerdings sind bestimmte Funktionen nur in neuen APIs verfügbar. Support Libraries rüsten neue Funktionen auch für alte APIs nach.

5.2 Manifest

Das Manifest umfasst:

- Komponenten der App
- Metadaten (Name, Icon, Versionsnummer)
- Permissions (Internet, kostenpflichtige Anrufe, etc.)
- Anforderungen an die Geräte API

- Wird vom System konsultiert um zu erfahren, ob die App installiert werden kann und welche Permissions diese verwendet
- Das Manifest enthält alle wichtigen Informationen über unsere App
- Alle Actvities der App müssen aufgelistet werden.
- Achtung: Android Studio erweitert das Manifest beim Build um zusätzliche Tags!
- Achtung: Beim Builden des Projekts wird das tatsächliche Manifest generiert.

5.3 Application Klasse

- Parent unserer Activities ist Application
- Application ist auch eine Klasse die den globalen Zustand unserer App hält
- Kann durch eigene Application-Subklasse ersetzt werden

Aus Activities kann mit getApplication() auf die Instanz zugegriffen werden. Hat Lifecycle-Methoden: onCreate(), onLowMemory(), onConfigurationChanged(Configuration newConfig). Das Application-Objekt kann gemeinsam genutzte Daten enthalten. Wird aber jedes mal gekillt, wenn App neustartet (Home Button etc.).

```
<application
   android:name="MyCustomApplication"
   android:icon="@drawable/icon"
   android:label="@string/app_name"
   ...>
```

6 Views

- Deklarativ: Im XML geschrieben (besser, da Trennung)
- Imperativ: Im Java Code geschrieben

Views erhöhen sofort die Komplexität. Sie belegen einen rechteckigen Bereich auf dem Screen. Sind zuständig, um dessen Inhalt zu zeichnen und müssen Events behandeln.
View ist Basisklasse um eine UI zu bauen. Eine Untergruppe der View sind die **Widgets**. Das sind nicht die Widgets des Home-Screens sondern ein Sammelbegriff für alle fix-fertigen Komponenten des UI wie Button aber auch der Packagename dieser Komponenten. Buttons oder Dialoge sind also auch Views.
Eine weitere Untergruppe sind die **ViewGroups**. Die ViewGroup enthält andere Komponenten. Die View ist die Oberklasse aller UI-Komponenten. Das Layout ist eine ViewGroup aber auch wiederum eine View (Vererbung und Composite Pattern).

```
public class MainActivity extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    TextView textView = findViewById(R.id.textView);
  }
}
```

- Android instanziert die obige View activity_main automatisch.
- Die Klasse R (Ressourcen) enthält alle Konstanten für die Layout-Datei.
- Mit setContentView(...) wird das Layout hinzugefügt.

Es können auch eigene Views erstellt werden indem man die Klasse mit @RemoteView annotiert. Der Tag im XML entspricht dann dem Klassenname.

6.1 Layouts

WIMP steht für Windows Icons Menu Pointer.
Es gibt zum Beispiel das **Linear Layout** (Horizontal oder Vertikal), das **Grid Layout** , das **Relative Layout**, das **Table Layout**. Ausserdem gibt es noch das **Framelayout**, das Views überlappend darstellen kann. Auch können eigene Layouts erstellt werden. Das Table Layout ist wie das Gridlayout, kann jedoch nicht scrollen.
Layouts können mit R.layout.activity_main referenziert werden, weil die xml-Datei dann activity_main.xml heisst.
Alle ViewGroups haben das width- und height-Parameter gemeinsam.

```
android:layout_width="match_parent"
android:layout_width="wrap_content"
```

match_parent: so gross wie möglich, also wie der Parent erlaubt
wrap_content: so klein wie möglich, also wie die Kinder erlauben

6.1.1 Linear Layout

Elemente werden Vertikal oder Horizontal angeordnet. Mit android:layout_weight kann ein Gewicht vergeben werden. Die Kinder ohne weight bekommen minimalen Platz, auf die restlichen wird der verfügbare Platz nach Gewicht aufgeteilt. Match_parent schnappt sich den ganzen verfügbaren Platz und alles danach wird aus dem Bild geschoben. Default orientation ist horizontal.


```
<LinearLayout
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:orientation="horizontal">
  <ImageView
    android:layout_width="24dp"
    android:layout_height="24dp"
    android:layout_gravity="center_vertical"
    app:srcCompat="@drawable/ic_bottle" />

  <TextView
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_gravity="center_vertical"
    android:layout_weight="1"
    android:text="Meine Biere" />

  <TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_vertical"
    android:text="0" />
</LinearLayout>
```

android:layout_gravity="right" sowie left und center können das Widget schieben (vertical). top, bottom, center funktionieren nur bei horizontal.

6.1.2 Relative Layout

Das vielseitigste Layout, welches Kinder relative zueinander anordnet.

```
<RelativeLayout xmlns:android="...">
<TextView
android:text="1. Platz"
android:id="@+id/first"
android:layout_centerHorizontal="true"/>
<TextView
android:text="2. Platz"
android:id="@+id/textView2"
android:layout_below="@id/first"
android:layout_toStartOf="@id/first" />
<TextView
android:text="3. Platz"
android:id="@+id/textView3"
android:layout_below="@id/first"
android:layout_toEndOf="@id/first" />
</RelativeLayout>
```

Andere Settings:

- **android:layout_alignRight**: Makes the right edge of this view match the right edge 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_above**: Positions the bottom edge of this view above the given anchor view ID
- **android:layout_toStartOf**: Positions the end edge of this view to the start of the given anchor view ID. Accommodates end margin of this view and start margin of anchor view. (Usually layout_toLeftOf)

6.1.3 Framelayout

Ein Framelayout kann die Kinder übereinander anordnen. Beispiel Live-Kamerabild mit Auslösebutton und Hilfslinien.

6.1.4 ConstraintLayout

Das ConstraintLayout ist konzeptuell mit RelativeLayout verwandt. Default im neuen Android Studio, da für GUI-Builder optimiert.

6.1.5 FlexboxLayout

Angelehnt an CSS Flexbox, Blog Post. Braucht zusätzliche Library, kein GUI-Builder-Support

6.1.6 WebView

Java-Objekte können mit JavaScript angesprochen werden. Kann man JS aktivieren und HTML anzeigen

6.1.7 GridLayout

gravity gilt hier für den Hint. Layout dann für die Ausrichtung im Grid.

```
<GridLayout
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:columnCount="2"
    android:rowCount="5">
       <EditText
        android:layout_row="3"
        android:layout_column="0"
        android:layout_columnSpan="2"
        android:hint="Compose email"
        android:layout_gravity="fill"
        android:gravity="top|left" />
</GridLayout>
```

6.2 Widgets

Es gibt **Eingabe-Widgets** wie Buttons oder Textboxen. Die Aktionen der Benutzer lösen Events aus. Die Widgets haben eine Id also zum Beispiel ein @+id/sendenButton. Im Code kann man sie dann mit R.id.sendenButton referenzieren. Im xml dann mit @id/sendenButton.

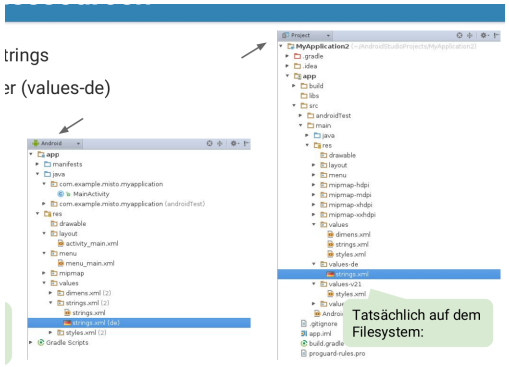
Auf den Eingabefeldern (EditText) kann man die erwartete Eingabeform angeben. So zum Beispiel für Telefonnummern, Email-Adresse, Datum usw. @+ ist das delkarieren und @ für die Nutzung.

```
<EditText
android:id="@+id/phone"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:inputType="phone|textCapSentences|
↪  textAutoCorrect|textPassword" />
```

7 Ressourcen

Es gibt nicht nur R.id sondern auch andere Ressourcen (alles ausserhalb vom Code):

- drawables - Bilder - R.drawables
- menu - Menu - R.drawables
- mipmap - Launcher-Icon der App - R.mipmap
- values - Strings und andere Konstanten. Können verschieden sein zum Beispiel values-de



Wie erwartet, wird pro Ordner eine innere Klasse in R generiert, mit Ausnahme der values (dimens, strings, styles). Strings die externalisiert werden, Farben, Dimensionen werden in XML-Dateien im Ordner values abgelegt. Zum Beispiel in strings.xml `getString(R.string.app_name);` (in Context definiert für Activity und Fragment (kein Context)). Oder in dimens.xml mit @dimen/name referenziert.
Man kann auf die Grösse des Screens Rücksicht nehmen mit zwei Layouts. Man kann da z.B. prüfen, ob ein statisches Fragment vorhanden ist und dann füllen. Sonst nicht. res/layout/main_activity.xml bzw. res/layout-sw600dp/main_activity.xml (600dp wide and bigger)

8 Dimensionen

Grössenangaben von Views erfolgen in **density-independent pixels**: dp oder dip (deprecated). Die Umrechnung erfolgt von inch zu dp: inch * dpi des Bildschirms * (dpi des Bildschirms / 160).
Bei Schriften sind es sp also scale-independent pixels.

9 Events und Eventhandler

Sobald alle Lifecycle Methoden aufgerufen wurden hat unsere App keine aktive Kontrolle mehr. Das Android-Framework hat ein Event-Loop (auch Main-Thread, GUI-Thread genannt), der auf Ereignisse wartet. **Nur der Main-Thread darf das GUI verändern**. Events werden vom User oder Sensoren ausgelöst und von **Event Listeners Interface** (wie View.OnClickListener) empfangen. Kann man auch mit `android:onClick="onBtnClicked"` implementieren (void-Funktion mit Parameter View in Activity/Fragment). Der **View Parameter** entspricht dem Button, der geklickt wurde. Ereignisse werden in einer Message-Queue abgelegt und vom main Thread abgearbeitet.

```
Button button = findViewById(R.id.button);
button.setOnClickListener( v -> { /* Act on event... */ });
```

```
final EditText password = findViewById(R.id.password);
password.addTextChangedListener(new TextWatcher() {
  @Override
  public void afterTextChanged(Editable s) {
    String pw = s.toString();
    if (pw.length() < 8) {
    password.setError("Passwort mind. 8 Zeichen");
    }
  }
  @Override
  public void
  ↪  onTextChanged(CharSequence s, int start, int before, int count){
  }
  @Override
  public void
  ↪  beforeTextChanged(CharSequence s, int start, int count, int after){
  }
});
```

Wenn die View View.OnClickListener implementiert und .setOnClickListener(this) ist.

```
public void onClick(View view) {
  if (view == btn1) { /* Code für btn1 */ }
  else if (view == btn2) { /* Code für btn2 */ }
}
```

10 Fragments

Ein Fragment es ist ein modularer Teil einer Activity mit **eigenem Lebenszyklus**. Ein Fragment kann in mehreren Activities eingebunden werden und eine Activity kann mehrere Fragments beinhalten. Fragments erben von Fragment und ein **LayoutInflater** nimmt XML und instanziert die View-Klasse. Die Umgebende Activity muss auf dem State running sein, sodass das Fragment attached werden kann. **ViewGroup container** ist Layout in der Parent-Activity. Fragments sind **wiederholbare Komponenten**, welche **keine Abhängigkeiten** auf eine Activity haben sollen (Best Practice: Callback Interface für Fragment).
Man kann ein Fragment **statisch oder dynamisch** einbinden. Statisch wird es in XML definiert mit `<fragment android:name="com.example.ManactivityFragment" android:id="..."/>`. Dynamische Fragments lassen sich austauschen. Man fügt dazu ein Framelayout (best practice) als Element ein zum Beispiel `<FrameLayout android:id="@+id/container" />` wo dann die Fragments Platz finden.

```
public class MainActivity extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    getFragmentManager()
      .beginTransaction()
      .add(R.id.container, new MainActivityFragment()); // insert at id
      .commit(); // execute
  }

  private void switchTo(Fragment fragment) {
    Bundle args = new Bundle();
    args.putSerializable("abc", userRegData);
    fragment.setArguments(args);
    getFragmentManager()
      .beginTransaction()
      .replace(R.id.placeholder, fragment)
      .addToBackStack(null);
      .commit();
  }

  public void onBackPressed() {
    if (fragmentManager.getBackStackEntryCount() <= 1) {
      finish(); //beendet Applikation
    } else {
      fragmentManager.popBackStack();
} } }
```

Dann das Fragment:

```
public class MainFragment extends Fragment {
  public interface GoNextClick {void click();} // callback Interface
  private GoNextClick activity;

  public MainFragment() { } // def ctor must exist

  @Override
  public View onCreateView(LayoutInflater inflater, ViewGroup container,
  Bundle savedInstanceState) {
    View v = inflater.inflate(R.layout.fragment_main, container, false);
    //false bedeutet, dass die View nicht in den Container eingefügt
    //werden soll, da das bereits das Fragment macht
    //view.getContext() gibt activity zurück
    UserRegistrationData data = (UserRegistrationData) getArguments()
      .getSerializable("abc");
    return v;
  }
  @Override
  public void onAttach(Context activity) {
    super.onAttach(activity);
    //Besser ein Interface nehmen mit den benötigten Methoden
    //Wenn nicht impl. dann Exception schmeissen
    this.activity = (MainActivity) activity;
  }
  //besser
  @Override
  public void onAttach(Context activity) {
    super.onAttach(activity);
    if (!(activity instanceof GoNextClick)) {
      throw new AssertionError("Activity must implement");
    }
    this.activity = (GoNextClick) activity;
  }
  @Override
  public void onDetach() {
    super.onDetach();
    activity = null;
  } }
```

11 ListView

```
<ListView
android:layout_width="match_parent"
android:layout_height="match_parent"
android:id="@+id/listView"/>
```

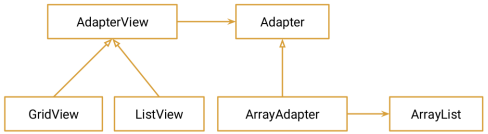
In onCreate(...) oder onCreateView(...):

```
//einfache Varianten (this ist activity sonst getActivity())
ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,
↪  android.R.layout.simple_list_item_1, myStringArray);
ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,
↪  R.layout.myLayout, R.id.usedTextLabel, myStringArray);

//Beispiel:
MyAdapter adapter = new MyAdapter(this, R.layout.myLayout, moduleList);

listView.setAdapter(adapter);
//parent = listview, view = getView(...), id = getItemId() vom Adapter
listView.setOnItemClickListener((parent, view, position, id) -> {
  Module module = (Module) parent.getItemAtPosition(position);
  module.setSelected(!module.isSelected());
  dataAdapter.notifyDataSetChanged();
});
```

```
public class MyAdapter extends extends ArrayAdapter<Module>
  public MyCustomAdapter(Context context, int rowLayout,
      ArrayList<Module> moduleList) {
    super(context, rowLayout, moduleList);
    //save modulelist for getView(...)
  }
  @Override
  public View getView(int position, View convertView, ViewGroup parent) {
    if (convertView == null) {
      LayoutInflater layoutInflater = (LayoutInflater) getActivity()
        .getSystemService(Context.LAYOUT_INFLATER_SERVICE);
      convertView = layoutInflater.inflate(R.layout.rowlayout, null);
    }

    final Module module = moduleList.get(position);
    TextView tw = (TextView) convertView.findViewById(R.id.textView);
    CheckBox cb = (CheckBox) convertView.findViewById(R.id.checkBox);
    cb.setOnClickListener((View v) -> {});
    tw.setText("(" + module.getCode() + ")");

    return convertView;
  }
  @Override
  public int getCount() {return list.size();}
}
```

12 Recycler View

Der **Adapter** adaptiert eine Liste oder einen anderen Speicher. Der **ViewHolder** zwischenspeichert die Views. Der **Layoutmanager** arrangiert die Views. Der **ItemDecoration** kann Dekoration um die Elemente oder Trennlinien machen. Der **ItemAnimator** kann hinzufügen, entfernen und scrolling von Elementen animieren.



Den Adapter kann man aktualisieren mit:

```
notifyDataSetChanged(); // Everything will be reloaded
notifyItemChanged(positionToUpdate); // better
notifyItemInserted(insertPosition); // more specific
notifyItemRemoved(removePosition); // more specific
```

```
public class
↪  NotesAdapter extends RecyclerView.Adapter<NotesAdapter.ViewHolder> {

  public class ViewHolder extends RecyclerView.ViewHolder {
    public View itemRoot;
    public TextView textView;

    public ViewHolder(View itemRoot, TextView textView) {
      super(itemRoot);
      this.itemRoot = itemRoot;
      this.textView = textView;
    }
  }

  private Notes notes;
  private ItemSelectionListener selectionListener;
  public NotesAdapter(Notes notes, ItemSelectionListener sl) {
    this.notes = notes;
    this.selectionListener = sl;
  }

  @Override
  public ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
    LayoutInflater layinf = LayoutInflater.from(parent.getContext());
    View v = layinf.inflate(R.layout.rowlayout, parent, false);
    TextView textView = (TextView) v.findViewById(R.id.textView);
    ViewHolder viewHolder = new ViewHolder(v, textView);
    return viewHolder;
  }

  @Override
  public void onBindViewHolder(ViewHolder holder, final int position) {
    final Note note = notes.get(position);
    holder.textView.setText(note.getTitle());
    holder.itemRoot.setOnClickListener((View
    ↪  v) -> selectionListener.onItemSelected(position));
  }

  @Override
  public int getItemCount() { return notes.size();}
}
```

```
public class NotesListFragment extends Fragment {

  private ItemSelectionListener itemSelectionCallback = null;
  private RecyclerView recyclerView;
  private LinearLayoutManager layoutManager;
  private NotesAdapter adapter;

  @Override
  public View onCreateView(LayoutInflater inf, ViewGroup container,
    Bundle saveinst) {

    View rootView = inf.inflate(R.layout.fragment, container, false);

    recyclerView = rootView.findViewById(R.id.recyclerView);

    // Eine Optimierung, wenn sich die Displaygroesse der Liste nicht
    // ändern wird.
    recyclerView.setHasFixedSize(true);

    layoutManager = new LinearLayoutManager(getActivity());
    recyclerView.setLayoutManager(layoutManager);

    Application app = (Application) getActivity().getApplication();

    adapter = new NotesAdapter(app.getNotes(), itemSelectionCallback);
    recyclerView.setAdapter(adapter);

    return rootView;
  }
}
```

13 Menus

Das Context Menu erscheint, wenn man z.B. lange auf ein Listentry drückt. Die ActionBar ist die oberste Leiste und kann ein Optionsmenu (oft ganz rechts) beinhalten.

13.1 Optionsmenu

In der Activity :

```
public boolean onCreateOptionsMenu(Menu menu) {
  //groupId, ItemId-Zahl, Sort, Name
  menu.add(0, START_MENU_ITEM, 0, "Start");
  menu.add(0, SUBMIT_MENU_ITEM, 0, "Submit");

  // ODER wenn im xml erstellt
  getMenuInflater().inflate(R.menu.menu_main, menu);

  return true;
}

public boolean onOptionsItemSelected(MenuItem item) {
  switch (item.getItemId()) {
    case START_MENU_ITEM: //Konstante frei wählbar
      return true; //bedeutet behandelt
    case SUBMIT_MENU_ITEM: //Wenn in XML definiert R.id...
      return true;
  }
  return super.onOptionsItemSelected(item);
}
```

Kann man auch in einem XML definieren `<menu> <item android:title="" /></menu>` in res/menu/ menu_main.xml
In Fragments:

```
public class MainActivityFragment extends Fragment {
  @Override
  public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
    inflater.inflate(R.menu.menu_main, menu);
  }
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setHasOptionsMenu(true); // Imporant otherwise not inflated
  }
```

13.2 Toolbar

In einem layout:

```
<android.support.v7.widget.Toolbar
android:id="@+id/toolbar"
android:layout_width="match_parent"
android:layout_height="wrap_content">
</android.support.v7.widget.Toolbar>
```

```
public class MainActivity extends AppCompatActivity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Toolbar toolbar = findViewById(R.id.toolbar);
    setSupportActionBar(toolbar); // Important part
} }
```

13.3 Toast

```
Toast toast
↪  = Toast.makeText(getActivity(), "Hello MGE!", Toast.LENGTH_SHORT);
//offset x und y = 0
toast.setGravity(Gravity.TOP| Gravity.LEFT, 0, 0);
toast.show();
```

13.4 Snackbar

```
private void mkSnack() {
  Snackbar s = Snackbar.make(tView, "Hello MGE!", Snackbar.LENGTH_LONG);
  s.setAction("Again!", (View v) -> {
    mkSnack();
  });
  s.show();
}
```

13.5 Diverse Widgets

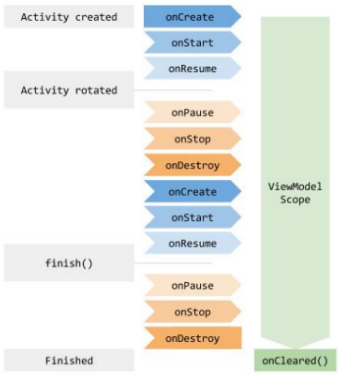
```
<android.support.design.widget.FloatingActionButton
android:src="@drawable/plus"
android:onClick="onPlusClicked" />

<!-- Damit Hinweistext
↪  erhalten bleibt beim Edit (EditText einfügen mit android:hint): -->
<android.support.design.widget.TextInputLayout ...></...>

<!-- Zusammenklappende Toolbar: -->
<android.support.design.widget.CollapsingToolbarLayout
```

```
//TabLayout und ViewPager:
ViewPager viewPager = findViewById(R.id.viewpager);
ViewPagerAdapter adapter = new
ViewPagerAdapter(getSupportFragmentManager());
adapter.addFragment(new ListFragment(), "CALLS");
adapter.addFragment(new ListFragment(), "CHATS");
adapter.addFragment(new ListFragment(), "CONTACTS");
viewPager.setAdapter(adapter);
TabLayout tabLayout = findViewById(R.id.tabs);
tabLayout.setupWithViewPager(viewPager);
```

```
<!-- Neu
↪  laden und in onCreate(...) dann .setOnRefreshListener(() -> ...) -->
<android.support.v4.widget.SwipeRefreshLayout ...>
<android.support.v7.widget.RecyclerView .../>
</android.support.v4.widget.SwipeRefreshLayout>
```

14 Persistenz

14.1 ViewModel



Sind vom Lebensyzklus der Activity oder des Fragemnts entkoppelt. Die Daten werden aber nicht persistiert. Mehrere Activity oder Fragments können das gleiche ViewModel benutzen. Das ViewModel muss von ViewModel erben. Das this ist die FragmentActivity oder das Fragment. Es wird dann nur ein ViewModel pro Activity erstellt.

```
public class MyViewModel extends ViewModel {
  // some Methods and states.
}

public class Activi extends Activity {
  public void onCreate(Bundle savedInstanceState) {
    MyViewModel
    ↪  model = ViewModelProviders.of(this).get(MyViewModel.class);
  } // per Activity a instance will be created
}
```

14.2 App-Daten Persistieren

```
//PREFS_NAME ist ein bliebiger String
SharedPreferences
↪  settings = getSharedPreferences("settings", MODE_PRIVATE);
SharedPreferences.Editor editor = settings.edit();

// false ist der Default  Wert
editor.putBoolean("MyPreferenceKey", false);
editor.getInt("abc", 1);
editor.getString("abc", "");
editor.commit();
```

14.3 File Storage

14.3.1 Interner Speicher

```
FileOutputStream fos = openFileOutput(FILENAME, Context.MODE_PRIVATE);
fos.write("File Content".getBytes());
fos.close();
```

14.3.2 Externer Speicher

Braucht Permission im Android Manifest.

```
File path = Environment.getExternalStoragePublicDirectory
↪  (Environment.DIRECTORY_PICTURES);
File file = new File(path, "HSR_Cat.png");
```

14.4 SQLite Storage

```
public class DBHelper extends SQLiteOpenHelper {
  private static final int DATABASE_VERSION = 2;
    DBHelper(Context context) {
    super(context, DATABASE_NAME, null, DATABASE_VERSION);
  }
  @Override
  public void onCreate(SQLiteDatabase db) {
    db.execSQL("CREATE TABLE ...;");
  }
  @Override
  public
  ↪  void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
  }
}

DBHelper helper = new DBHelper(this);
SQLiteDatabase db = helper.getReadableDatabase();
db.execSQL("SELECT * FROM ...;");
```

14.5 onSaveInstanceState

```
public class MainActivity extends AppCompatActivity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
  }

  @Override
  protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
} }
```

Speichert alle Views, die eine ID haben. Kann man auch mit putString(String key, String value) oder mit putInt(...) erweitern. onSaveInstanceState wird nicht immer ausgeführt wenn z.B. gekillt oder mit Back-Button verlassen wird. Konsequenz: App-Daten immer in onPause sichern.

15 Hintergrundtasks

Operationen die lange laufen müssen in einem separaten Thread laufen. Da App sonst nicht mehr auf Usereingaben reagiert.

```
//funktioniert nicht, wenn man UI-Elemente anpassen muss
Thread thread = new Thread(runnable);
thread.start();
```

Wenn man UI-Elemente anpassen muss (simple Variante):

```
  public void onClick(View v) {
    Runnable runnable = () -> {
      final Bitmap bitmap = download("http://slow.hsr.ch/hsr_cat.bmp");
      imageView.post(() -> imageView.setImageBitmap(bitmap));
    };
    Thread thread = new Thread(runnable);
    thread.start();
  }
```

15.1 AsyncTask

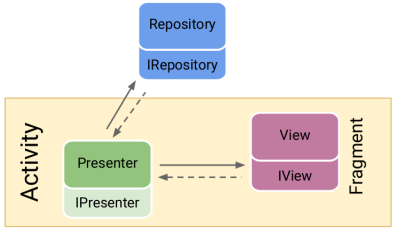
```
class DownloadBitmapTask extends AsyncTask<String, Integer, Bitmap> {
  @Override
  protected void onPreExecute() {
    super.onPreExecute(); // Wird im UI-Thread ausgeführt
  }
  @Override
  protected Bitmap doInBackground(String... params) {
    //Wenn Integer nicht Void
    //Kann man in einem Loop aufrufen für z.B. Progressbar
    publishProgress(10);

    return download(params[0]); // Wird in einem eigenen Thread ausgeführt
  }
  @Override
  protected void onProgressUpdate(Integer... values) {
    progressBar.setProgress(values[0]); // Wenn Integer nicht Void
  }
  @Override
  protected void onPostExecute(Bitmap bitmap) {
    imageView.setImageBitmap(bitmap);
  }
}

new DownloadBitmapTask().execute( "http://slow.ch/cat.bmp");
```

oder AsyncTask, der eingebettet in FileActivity ist:

```
private class mTask extends AsyncTask <String, Void, List<WordCount>> {
  final FileHolder holder;
  private mTask(FileHolder holder) {
    this.holder = holder;
  }
  @Override
  protected List <WordCount> doInBackground(String...strings) {
    return analyzeText(strings[0]);
  }
  @Override
  protected void onPostExecute(List < WordCount > wordCounts) {
    super.onPostExecute(wordCounts);
    WordCountResult result = new WordCountResult(holder, wordCounts);

    Intent showResultIntent
    ↪   = new Intent(FileActivity.this, WordListActivity.class);
    Bundle bundle = new Bundle();
    bundle.putSerializable(KEY_WORD_RESULT, result);
    showResultIntent.putExtras(bundle);
    //oder context in Constructor speichern
    FileActivity.this.startActivity(showResultIntent);
  }
}
```

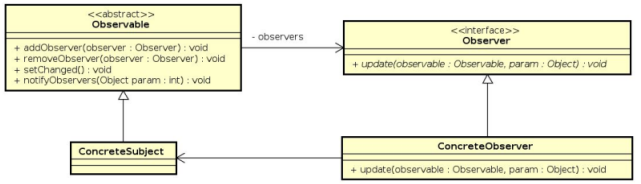
16 Patterns und Architektur

- Die **Multitier Architecture** soll Abhängigkeiten auf unterliegende Schichten beschränken und Zyklen verhindern.
- Die **Presentation** is zuständig für die Darstellung und die Interaktion mit dem Benutzer. Ist dadurch typischerweise sehr stark an UI-Toolkit gebunden. Hat Zugriff auf Domain.
- Die **Domain** ist die Businesslogik und Domainklassen ohne UI Funktionalität. Die Domain soll nicht abhängig von Android sein. Die Domain hat wenig externe Abhängigkeiten und ist einfach zu testen.
- **Data** implementiert die Speicherung der Daten. Stellt diese Dienste der Domain zur Verfügung.
- An den Schnittstellen der Layer wird gerne mit Interfaces und Fassaden gearbeitet um diese zu entkoppeln.
- Wenn keine Zyklen erlaubt sind, wie kann die Domain Presentation dann die Presentation benachrichtigen wenn der Zustand eines Domainobjekts geändert hat? **Observer Pattern**. Subject welches beobachtet wird (z.B. Model). Observer welcher beobachtet (z.B. View). Observer melden sich beim Subject an, damit das Subject den Observer doch bitte informiert, sobald was geschehen ist. Observer kennt Subject ziemlich gut, aber das Subject muss nichts über den konkreten Observer wissen.
- Beim **MVC** liest die View das Model. Der **Controller** manipuliert das Model und bekommt Benachrichtigungen von der View. Es ist also kein Zyklus: Listener oder Observer bzw. Benachrichtigung ist der gestrichelte Pfeil.




```
private final LiveData<List<Wish>> myWishlist;
myWishlist.setValue(new ArrayList<Wish>(...));
```

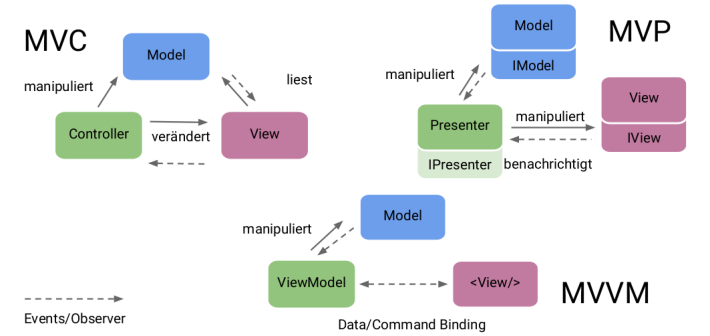
```
model = ViewModelProviders.of(this) .get(MainViewModel.class);
model.getMyWishlist().observe(this, this::updateWishlistCount);

private void updateWishlistCount(List<Wish> wishes) {
myWishlistCount.setText( String.valueOf(wishes.size()));
}
```



notify in zwei Schritten: setChanged(); und notifyObservers(...);. Ohne setChanged() wird nichts benachrichtigt.

16.1 MVC, M. V. Presenter, M. V. View-Model

Controller durch Activity ersetzen.

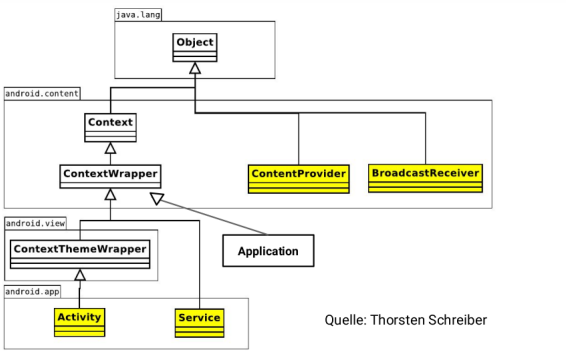


- Interfaces für View, Model (sogenannte Repositories) und Presenter (optional)
- Fragment implementiert View-Interface
- Activity instanziert Presenter, verbindet Presenter mit Repository-Model und View

16.2 Observer-Pattern

```
public class StocksAdapter
↪  extends RecyclerView.Adapter<...> implements Observer {
  private ArrayList<Stock> dataset;
  public StocksAdapter(ArrayList<Stock> stocks) {
    dataset = stocks;
    stocks.get(0).addObserver(this);
  }
  @Override
  public void update(Observable observable, Object data) {
    int position = dataset.indexOf(observable);
    notifyItemChanged(position);
  }
}

public class Stock extends Observable {
  private double price;
  // Konstruktor, Getter
  private void setNewRandomPrice() {
    price += new Random().nextDouble();
    setChanged();
    notifyObservers();
  }
}

LiveData<List<Wish>> myWishlist = new LiveData<List<Wish>>();
myWishlist.setValue(new ArrayList<Wish>(...));

//Observer ist this (also Fragment, Activity):
MainViewModel
↪  model = ViewModelProviders.of(this).get(MainViewModel.class);
model.getMyWishlist().observe(this, this::updateWishlistCount);
```

17 Context

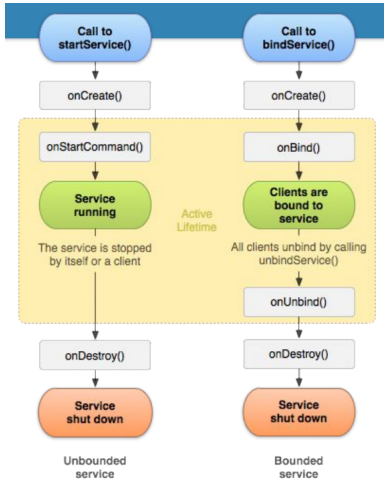


Quelle: Thorsten Schreiber

Mit einer Context-Instanz (also typischerweise die Activity) können wir:

- Neue Views erstellen (Kontext muss als Parameter mitgegeben werden)
- Auf System-Services zugreifen `context.getSystemService(LAYOUT_INFLATER_SERVICE)`;
- Die Applikationsinstanz erhalten
- Neue Activities starten (mittels Intent)
- Preferences lesen und schreiben
- Services starten

18 Services



- Für Aufgaben, die im Hintergrund ablaufen sollen oder deren Abarbeitung das UI zu lange blockieren würde wie Musik, Netzwerk laden oder rechenintensive Aufgaben.
- Haben keine Activity
- können gestartet werden. Die Erledigung einmaliger Aufgaben (z.B. Download).
- Ein Service wird mit startService(intent) gestartet.
- Der Service läuft im Hintergrund und wird nicht gestoppt*, auch wenn der Anwender die App wechselt oder der startende Context zerstört wird.
- Hintergrund meint nicht in einem anderen Thread, sondern dass der Service auch dann weiterläuft wenn der User die App wechselt!
- AsyncTask für Entkoppelung von Main-Thread. Service um Aufgabe vom Context zu entkoppeln.
- können gebunden werden. Client-Server ähnliche Kommunikation über eine längere Zeitdauer.
- Ein Service wird mit bindService(intent, ...) gebunden.
- Es ist möglich, dass mehrere Clients gebunden sind
- Wenn alle Clients unbindService(...) aufgerufen haben, wird der Service zerstört.
- Services die immer laufen sollen, auch wenn die App nicht aktiv ist, müssen eine Notification anzeigen
- arbeiten im UI-Thread der Applikation, sind also keine eigenen Threads
- müssen in der Manifest-Datei deklariert werden.

```
<application>
<service
android:name=".LocalService"
android:exported="false" /> <!-- nur eigene App -->
</application>
```

```
Intent intent = new Intent(this, LocalService.class);
intent.putExtra("KEY1", "Value to be used by the service");
startService(intent);
stopService()
//Im Service
stopSelf();
```

18.1 Intent Service

Intent Service benutzt eigenen Worker-Thread. Stoppt den Service nachdem alle Intents abgearbeitet wurden. Enthält keine Referenz auf die Activity. Resultat mit Broadcast oder mit PendingIntent mitteilen.

```
public class HelloIntentService extends IntentService {
  public HelloIntentService() {
    super("HelloIntentService");
  }
  @Override
  protected void onHandleIntent(Intent intent) {
     Bundle bundle = intent.getExtras();
     FileHolder fileHolder = (FileHolder) bundle.get("abc");
  }
}
```

In der Activity

```
Bundle bundle = new Bundle();
bundle.putSerializable(KEY_FILE_HOLDER, holder);
Intent intent = new Intent(this, WordCountService.class);
intent.putExtras(bundle);

startService(intent);
```

18.2 Bound Service

Beim Aufruf von bindService() erhält der Client ein Interface, um mit dem Service zu kommunizieren. Nachdem alle Clients unbindService() aufgerufen haben, wird der Service beendet.

```
public class LocalService extends Service {
  private final IBinder binder = new LocalBinder();
  public class LocalBinder extends Binder {
    LocalService getService() {
      return LocalService.this;
    }
  }
  @Override
  public IBinder onBind(Intent intent) {
    return binder;
  }
  public int getRandomNumber() {
    return new Random().nextInt(100);
  }
}
```

Client (Activity):

```
Intent intent = new Intent(this, LocalService.class);
bindService(intent, connection, Context.BIND_AUTO_CREATE);

private ServiceConnection connection = new ServiceConnection() {
  //ComponentName ist Activity/Fragment/Service etc.
  //Binder ist Base interface for a remotable object (rpc)
  @Override
  public void onServiceConnected(ComponentName clsName, IBinder binder) {
    LocalService.LocalBinder
    ↪  myBinder = (LocalService.LocalBinder) binder;
    LocalService myService = myBinder.getService();
    int random = myService.getRandomNumber();
  }
  @Override
  public void onServiceDisconnected(ComponentName name) { }
};
```

19 Content Provider

- Stellt Daten prozessübergreifend zur Verfügung
- Client-Server Modell
- Provider-Client und Provider kommunizieren über standardisierte Schnittstelle um Daten auszutauschen
- Android stellt Daten über Content-Provider zur Verfügung: Kalender, Kontakte, ...
- Der Content Provider kann Permissions setzen, um den Zugriff auf die Daten einzuschränken oder zu schützen
- Client muss die erforderlichen Permissions im Manifest angeben
- Nur erforderlich wenn: Anderen Apps strukturierte Daten oder Dateien zur Verfügung stehen sollen.

20 Broadcast Receiver

Das System verschickt Meldungen per Broadcast. Meldungen werden als Intent verschickt, Action bestimmt Ereignistyp. Eine Meldung enthält Informationen über ein bestimmtes Ereignis wie Akku schwach oder SMS empfangen oder gebootet usw.. Broadcast Receiver können registriert werden, um bestimmte Meldungen zu erhalten. Apps können auch Meldungen per Broadcast verschicken.
Statisch registrierbar:

```
<receiver android:name=".TimeChangedReceiver">
  <intent-filter>
    <action android:name="android.intent.action.TIME_SET" />
  </intent-filter>
</receiver>
```

Dynamisch registrieren in onResume():

```
LocalBroadcastManager lbm = LocalBroadcastManager.getInstance(this);
// Intent.ACTION_BOOT_COMPLETED
IntentFilter filter = new IntentFilter("image");
//Hier vielleicht die View/Activity übergeben:
MyBroadcastReceiver receiver = new MyBroadcastReceiver();
lbm.registerReceiver(receiver, filter);
```

Abmelden in onPause():

```
LocalBroadcastManager lbm = LocalBroadcastManager.getInstance(this);
lbm.unregisterReceiver(receiver);
```

Der Receiver kann eine Referenz auf eine View/Activity erhalten, damit er das UI verändern kann:

```
private class MyBroadcastReceiver extends BroadcastReceiver {
  public MyBroadcastReceiver(ImageView v) {...}
  @Override
  public void onReceive(Context context, Intent intent) {
    String data = intent.getExtras().getString("data");
  }
}
```

In einem IntentService gibt es den ApplicationContext():

```
LocalBroadcastManager
↪  lbm = LocalBroadcastManager.getInstance(getApplicationContext());
Intent i = new Intent("image");
i.putExtra("path", f.getPath());
lbm.sendBroadcast(i);
```

21 Sensoren

- SensorManager als Einstiegspunkt für die verfügbaren Sensoren
- Sensor als Repräsentant für einen konkreten Sensor
- SensorEventListener um sich für Updates von Sensordaten zu registrieren
- SensorEvent um die Sensordaten (Werte, Zeitpunkt) auszulesen.

```
public class MainActivity
↪  extends AppCompatActivity implements SensorEventListener {
  private TextView textView;
  private SensorManager sensorManager;
  // Implementieren für onSensorChanged
  private Sensor lightSensor;
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    //...
    textView = (TextView) findViewById(R.id.textView);
    sensorManager
    ↪  = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
    //Besser: prüfen ob Sensor vorhanden!
    lightSensor = sensorManager.getSensorList(Sensor.TYPE_LIGHT).get(0);
  }
  @Override
  protected void onResume() {
    super.onResume();
    sensorManager.registerListener(this,
    ↪  lightSensor, SensorManager.SENSOR_DELAY_NORMAL);
  }
  @Override
  //Wir wollen nur Sensordaten empfangen wenn die App im Vordergrund läuft
  protected void onPause() {
    super.onPause();
    sensorManager.unregisterListener(this);
  }
  @Override
  public void onSensorChanged(SensorEvent event) {
    //Je nach Sensor unterschiedliche Daten
    textView.setText(String.format("Helligkeit:
    ↪  %.0f", event.values[0]));
  }
}
```

22 Dependency Injection

22.1 Dagger 2

Ein Dependency-Injection Framework ist Dagger.

- Ein **Modul** instanziert unsere Klassen (z.B. Repository) die wir injecten wollen
- Eine **Komponente** fasst Module zusammen und ist zuständig für die Injection
- Eine Klasse lässt sich über die Komponenten ihre Abhängigkeiten injecten

Setup Code:

```
@Module
public class BeersRepositoryModule {
  @Provides
  @Singleton
  public BeersRepository getBeersRepository() {
    return new BeersRepository();
  }
}

@Singleton
@Component(modules = {BeersRepositoryModule.class})
public interface BeerProComponent {
  //für jede Klasse, die wir injecten wollen.
  void inject(MainActivity activity);
}

public class Application extends android.app.Application {
  BeerProComponent component;
  //In der Application erstellen wir die Komponente
  public void onCreate() {
    component = DaggerBeerProComponent.builder().build();
  }
  public BeerProComponent getComponent() {
    return component;
  }
}

public class MainActivity extends AppCompatActivity {
  @Inject
  BeersRepository beersRepository;
  //Mit @Inject annotierte Felder werden von der Komponente gesetzt
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    ...
    ((Application) getApplication()).getComponent().inject(this);
    ...
  }
}
```

Vorteil: einfache Testbarkeit: Modul oder Applikation austauschen. Nachtei: Bei Fehlkonfiguration gibt es NullpointerExceptions.

22.2 Butter Knife

Wie dagger aber für Views.

```
@BindView(R.id.password) EditText password;
@BindString(R.string.login_error) String loginErrorMessage;
@OnClick(R.id.submit)
void submit() {
  ...
}

@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.simple_activity);
  ButterKnife.bind(this); // Important
  ...
}
```

22.3 Data Binding

View Binding erspart uns eine gewisse Tipparbeit. Datbinding ermöglicht es, dass man direkt auf das Objekt zugreifen kann.

```
<layout xmlns:android="http://schemas.android.com/apk/ res/android">
  <data>
    <variable name="user" type="ch.hsr.User"/>
    <variable name="presenter" type="com.android.example.Presenter" />
  </data>
  <RelativeLayout>
    <TextView android:text="@{user.firstName}" />
    <TextView android:text="@{user.lastName}" />
  </RelativeLayout>
</layout>

<!--
android:text="@{String.valueOf(index + 1)}"
android:visibility="@{age < 13 ? View.GONE : View.VISIBLE}"
android:onClick="@{() -> presenter.onSaveClick(task)}"\\
android:onClick="@{controller::onButtonSaveClicked}"
```

```
android:addTextChangedListener="@{user.lastNameWatcher}"
-->
```

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

Es gibt auch noch die

```
public
↪  final ObservableField<String> last = new ObservableField<String>();
last.addOnPropertyChangedCallback(
↪  Observable.OnPropertyChangedCallback callback);
last.set("abc");
last.get();
```

22.4 Lombok

"Project Lombok is a java library that automatically plugs into your editor and build tools, spicing up your java. Never write another getter or equals method again."

- @Data für Datenklassen
- @Value für Valueklassen
- @FieldNameConstants

@Data is a convenient shortcut annotation that bundles the features of @ToString, @EqualsAndHashCode, @Getter / @Setter and @RequiredArgsConstructor together: In other words, @Data generates all the boilerplate that is normally associated with simple POJOs (Plain Old Java Objects) and beans: getters for all fields, setters for all non-final fields, and appropriate toString, equals and hashCode implementations that involve the fields of the class, and a constructor that initializes all final fields, as well as all non-final fields with no initializer that have been marked with @NonNull, in order to ensure the field is never null.
@Data is like having implicit @Getter, @Setter, @ToString, @EqualsAndHashCode and @RequiredArgsConstructor annotations on the class (except that no constructor will be generated if any explicitly written constructors already exist).
@Value is the immutable variant of @Data; all fields are made private and final by default, and setters are not generated. The class itself is also made final by default, because immutability is not something that can be forced.
The @FieldNameConstants annotation generates an inner type which contains 1 constant for each field in your class; either string constants (fields marked public static final, of type java.lang.String) or if you prefer, an enum type with 1 value for each field. @FieldNameConstants is useful for various marshalling and serialization frameworks.

23 Material Design

- Eine Design Language ist also eine Hilfestellung für den Designprozess. Beschreibt, wie die Teile einer Applikation aussehen und sich verhalten sollen.Teilweise klare Regeln, aber auch Empfehlungen und Beispiele wie Farbschema, Icons usw.
- Ziel ist, ein konsistentes und benutzbares Look-and-feel im ganzen Android zu erreichen. Wie Coding Guidelines ist es auch empfehlenswert Human-Interface Guidelines im Projekt festzulegen.
- Materialien sind geometrische Formen (bzw. Schnipsel aus Papier) mit einer Dicke von exakt 1dp. UI-Elemente haben unterschiedliche Höhen. Alle Materialien werden an einem 8dp Grid ausgerichtet. Grundelement ist das Paper, das an- und übereinander gelegt werden kann.
- Farbauswahl: grosse Palette mit Farben und Abstufungen (Beispiele). Empfehlung: 3 Farbtöne der Primärpalette und eine Akzentfarbe aus einer zweiten Palette auszuwählen
- Animationen helfen die Illusion aufrecht zu erhalten, dass wir die Materialien auf dem Screen (durch eine Glasscheibe) direkt manipulieren.
- Patterns geben Hilfestellung bei oft auftretenden Problemen. Beispiel: Wie gibt man Feedback zu falschen Usereingaben?
- Umsetzung kann man direkt auf jedem Control machen. Oder man lagert es auf die styles aus.
- Views können keine eigenen Themes haben, aber Theme Overlays,

```
<!-- res/values/styles.xml -->
<style name="MyButtonStyle">
<item name="android:background">#ff85e1</item>
<item name="android:height">32dp</item>
<item name="android:minWidth">64dp</item>
<item name="android:padding">8dp</item>
</style>

<Button style="@style/MyButtonStyle" />
```

```
<!-- Manifest -->
<application
...
android:theme="@style/AppTheme" >
<activity
android:name=".MainActivity"
android:theme="@style/AppTheme" >

<!-- styles.xml -->
<style name="AppTheme" parent="Theme.AppCompat.Light.NoActionBar">
<!-- Hier können wir Theme-Einstellungen überschreiben -->
</style>
```