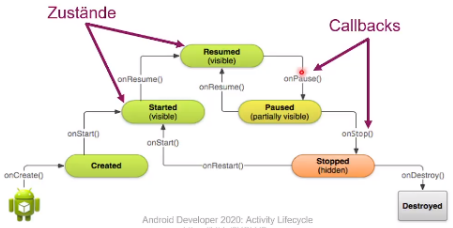
Native / Web /Hybrid

Eigenschaft	SP Native	CP Native	CP Hybrid	CP Web
Performance	+++	+++ / ++	++	+
Natives Aussehen[1]	+++	+++ / ++	++	+
Zugriff auf Gerätefunktionen[2]	+++	+++ / ++	++	+
Portabilität von Code	+	++	+++	+++
Anzahl benötigter Technologien	+	++	+++	+++
Re-Use von existierendem Code	+	++	++	+++
Deployment	App Store	App Store	App Store	Webserver

Callbacks



Ressources

Every Android App has some resources. It exist different kind of resources:

- colors which are used in the app (colors.xml)
- strings which are used (strings.xml)
- styles (styles.xml)
- dimensions / sizes (dimens.xml)

To reference a value from one of the resource file the following syntax is used:

> <Button android:id="@+id/buttonNavigate"
> <!– creates new ID on place –>
> android:text="@string/open$_{2ndactivity}$"
> <!– references open$_{2ndactivity}$ in strings.xml –> />

Dimensions

In the Android SDK exist six diffrent units for dimensions:

- dp: Density-indepentend pixels (das sollte man verwendend)
- sp: scale-independend pixels (das verwenden, wenn man mit Schriftgrössen arbeitet)
- px: Pixels
- pt: Punkt
- in: Inch
- mm: millimeter

You should only use `dp` and `sp` because these two units are the only one which scale with the screen resolution. `sp` should be used if it is used for text size because if the setting `big font size` is enabled on Android the font is scaled. `dp` is not affected by `big font size` option.

Android Manifest The AndroidManifest.xml contains essential information over the app which are important for the operating system. The most important parts of the file are:

- package: a unique ID for the app
- versionName: human readable string. Mostly in the style of 1.0.0
- versionCode: a positive Inter for internal usage. Higher number = new App. Can just be incremented
- minSDKVersion: minimal required SDK Version the smartphone needs
- maxSDKVersion: max required SDK (is ignored)
- targetSDKVersion: on which SDK Version you tested your application

Intents

Intents are used to call other components like an other Activity or an other App. In the Android SDK exists two diffrent kind of intents:

- Explicit intents (An explicit intent is mostly used to call an other component of the own app)
- Implicit intents (An implicit intent is used mostly used call other apps)

An App can register itself for implicit intents inside the AndroidManifest.
To send data to the target component the following functions are used:

- `setData`
- `putExtra() / putExtras()`

AppComat / Compatibility

AppCompat is a part of the Jetpack Library in the AndroidX namepace. It is used to use new Features on old devices. Where possible you should use the AppCompat class instead of the original Android SDK version.

Layouts

The Simple Layouts are used when you know at compile time how many items should be rendered on the Screen. In the Android SDK exits multiple simple layouts. But today only the Jetpacks ConstraintLayout is used. The most important attributes:

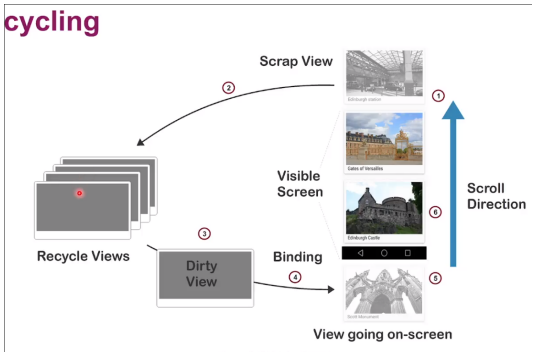
- `layout_width`
  - match$_{parent}$ (as big as possible)
  - match$_{content}$ (as small as possible)
  - dp (not recommended)
- `layout_height`
  - match$_{parent}$ (as big as possible)

  - $\text{match}_{\text{content}}$ (as small as possible)
  - dp (not recommended)

- `padding`
- `layout_margin`

The prefix `layout_` indicates that the argument is just a wish to the parent view element. The parent can ignore / adjust this option to optimize the screen.

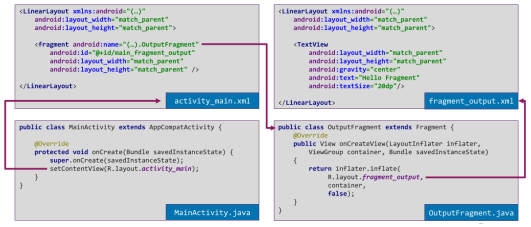
- The LinearLayout in Android
- The FrameLayout in Android
- The RelativeLayout in Android
- Jetpacks ConstraintLayout



Fragments

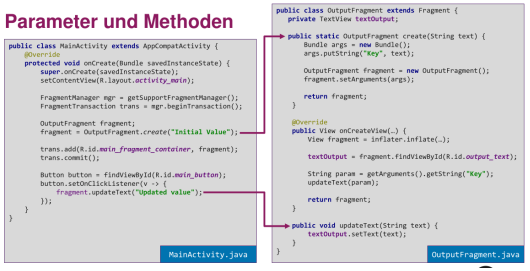
Sometimes you want to use the same screen in multiple activities. For example a list of items which is faded in. This list of item can not be implemented as a Android SDK Activity. But with a **Fragment** it is possible. A fragment consists of a XML and Java file.

Use only the `androidx.fragment.app*` classes.




```
<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>
```
activity_main.xml

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

        FragmentManager mgr = getSupportFragmentManager();
        FragmentTransaction trans = mgr.beginTransaction();

        OutputFragment fragment = new OutputFragment();
        trans.add(R.id.main_fragment_container, fragment);
        trans.commit();
    }
}
```
MainActivity.java



How to hand over parameters to a fragment

Use always `Bundle` to set parameters for the Fragment (What are Fragments in the Android SDK) Otherwise, the values getting lost when the fragment is updated. It is recommended to create a Factory Function to create the fragment.

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; } }

Styles in Android

```
<TextView
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:text="Element 1"
   style="@style/HeaderText" />

<TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Element 2"
    style="@style/HeaderText.Big"/>
```
layout.xml

```
<style name="HeaderText">
    <item name="android:textSize">24sp</item>
    <item name="android:background">#ff9999</item>
    <item name="android:padding">8dp</item>
    <item name="android:layout_margin">8dp</item>
    <item name="android:gravity">center</item>
</style>

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

```
<resources>
  <style name="AppTheme" parent="…">
    <item name="android:textViewStyle">@style/MyText</item>
  </style>

  <style name="MyText">
    <item name="android:textSize">24sp</item>
    <item name="android:background">#ff9999</item>
    <item name="android:padding">8dp</item>
    <item name="android:layout_margin">8dp</item>
    <item name="android:gravity">center</item>
  </style>
</resources>
```
styles.xml

Material Design
Material is the metaphor:

- inspired by the pysical world
- surface should be like paper and ink

Bold, graphic, intentional:

- based on prinicples of print medias
- hierarchy, grid, fonts, colors

Motion provides meaning:

- motion means action
- sestrained, subtle use

Permissions
In Android exists two kinds of permission:

- normal, requested by the system
- dangerous, requested by the user

Permissions can be revoked by the user at **any** time. Since API 30 it is possible that the system revokes the permission automatically. **Important**: Check always access on proceeded APIs. If not a `SecurityException` could be thrown.

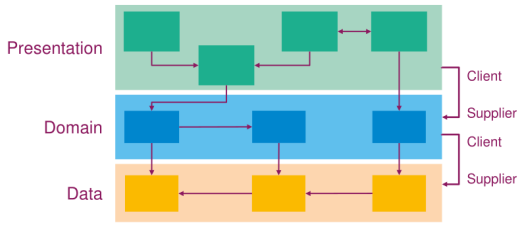
Persistence
In Android exits 5 APIs to store something on the storage:

- App-sepcific files
  - anything
  - internal / external
  - File API
- Preferences
  - Key-Value, simple value
  - internal
  - SharedPreferences
- Database
  - often domainobjects
  - internal
  - SQLite / Room
- Medien
  - Images, Videos, Music
  - external
  - MediaStore
- Dokumente
  - Documents (PDF, ZIP)
  - external
  - Storage Access Framework

Sensor Framework
For accessing the sensors in a Android based smartphone you use the Sensor Framework. All sensors use the same API:

- `SensorManager`, entry point
- `Sensor`, represent the real sensor
- `SensorEvent`, contains the values
- `SensorEventListener`, used for callbacks

Layer Architecture



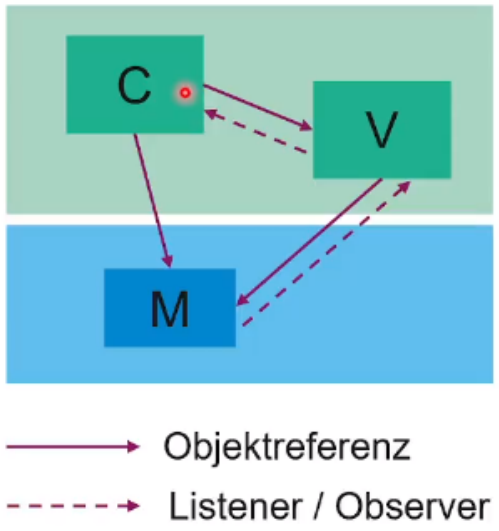
Obeserver Pattern
The goal of the Observer Pattern is to resolve a cyclic dependency. The Pattern consists of two objects:

- **Subject**: is monitored (e.g. a model)
- **Observer**: monitors the subject (e.g. a view)

The observer register it self on the subject. The only requirement is that the subject implements a specific interface (Java, C#). Using this approach the domain does not have to need anything from the view (excepted the interface).

MVC
The MVC consist of three parts:

- Model: contains the data
- View: reads data from the model and render it
- Controller: coordination between View and Model



Application class
Every Android App has a `application` node in the AndroidManifest file. During the start of the app a instance of the class is created. It is possible to inherit from this class.

```
<application android:name=".MyApplication">
<!– Ower Components –> </application>

public class MyApplication extends Application { @Override
public void onCreate() { super.onCreate(); }
// .. more callbacks }
```

Using a custom application class you can:

- create object which should be initialized only once
- creating Singleton objects
- access to global objects
- monitor life cycle of all activities
  - implemented interface
  - `Application.ActivityLifecycleCallbacks`

Application Lifce Cycle
The Application class in the Android SDK has several life cycle methods:

- onCreate
- onTerminate
  - on real devices never called
- onConfigurationChanged
  - on system configuration changes
- onLowMemory
  - hint for a possible termination of the app
- onTrimMemory
  - on moments where it is possible to clean up
  - e.g. app is in background

Context
With a `Context` object it is possible to get access to various services and resources of an app. Every `Context` object has a limited life time. After the life time is over all resources requested by the context are freed.
Attention: The Application context has only restricted possibilities for UI interactions

	Application	Activity	Service	ContentProvider	BroadcastReceiver
Show a Dialog	NO	YES	NO	NO	NO
Start an Activity	NO[1]	YES	NO[1]	NO[1]	NO[1]
Layout Inflation	NO[2]	YES	NO[2]	NO[2]	NO[2]
Start a Service	YES	YES	YES	YES	YES
Bind to a Service	YES	YES	YES	YES	NO
Send a Broadcast	YES	YES	YES	YES	YES
Register BroadcastReceiver	YES	YES	YES	YES	NO[3]
Load Resource Values	YES	YES	YES	YES	YES

Broadcasts
Broadcasts are normal intents (What is an Intent and for what it is used in the Android SDK?) and are used to exchange message between apps. There are two kinds of Broadcasts:

- lokal broadcasts: exchange inside the app
- global broadcasts: Global Broadcasts in Android

You can register your self for a broadcast (What are Broadcasts in Android?) in two ways:

- statically in the AndroidManifest (not recommended / deprecated)
- dynamically in the code (Dynamic registration for Broadcasts in Android)

Global Broadcast
Global broadcasts are used to exchange messages inside the whole system. Android it self sends global broadcasts. For example when:

- the system is booted
- network connection is lost
- a SMS is received

Services
In Android (Android Open Source Project) Services are used to start Tasks in the background. For example:

- download data from a web page
- play music

The Services is executed by the **Main-Thread**. *Important*: The lifecycle of a service is independent of the App and has no UI (some times a Notification).
Started Service
Started Services is one type of Services (For what are Services used in Android?) in Android (Android Open Source Project). This kind of service is used for unique actions for example for a download of files from a webpage.
Attention: This service runs potential for ever. To stop the service:

- stopped by service itself: `stopSelf()`
- stopped by application: `stopService()`
- stopped by Android

Bound Service

Bound Services is one kind of services (For what are Services used in Android?) in Android (Android Open Source Project). It is used for task which are long lasting. For example a music player.

The communication between the app and the service is similar to a client/server communication. The service can have multiple clients. If the last client disconnects from the service it is stopped.

APK Files

In Android (Android Open Source Project) Apps are installed from a so called APK file (Android Package). This APK files are just a normal zip file.

APK Splitting

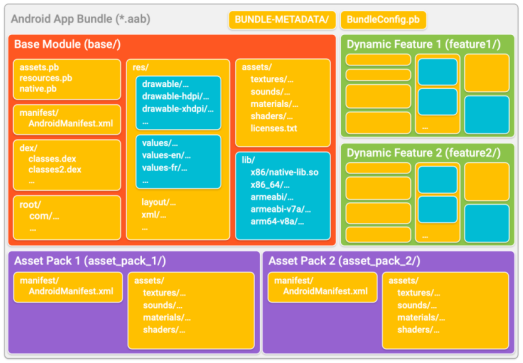
In the Google Play Store the size of the APK file (What is a APK file?) is limited to 100 MB. To overcome this limitation you can split your APK file in multiple smaller APKs

APK Expansion Files

In the Google Play Store the size of the APK file (What is a APK file?) is limited to 100 MB. To overcome this limitation you can move your resources (e.g. videos) in a Exapnsion File (.OBB). Play Store allows 2x2GB.

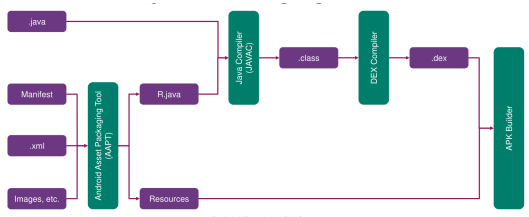
What is the ABB?

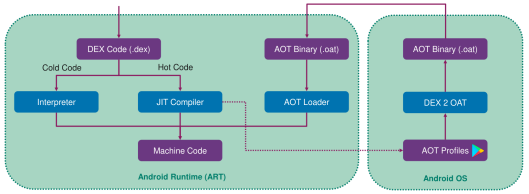
The Android App Bundle (AAB) is the successor of the APK (What is a APK file?). AAB is a container for all the content from your app. You upload the the Container and Google builds the APK dynamically.



Android Build Process

First all files from the project must be compiled to a APK file (). The resulting APK file (What is a APK file?) is uploaded to the Google Play Store. The Installed App is executed. If the code is already translated to Machine Code it is executed straight away. Else you compile it to and upload the compiled part to the Play Store. Next time a user needs this part of code is does not need to compile it by itself ().





View Binding

View Binding is way to access the View Elements. It's benefit over the traditional way are:

- type safety
- null safety
- no `findViewById()` calls

View Binding (What is View Binding in an Android Application?) needs to be enabled in Gradle. The Binding class is generated during the build. The name of the Binding class is build from the Layout name in Camel Case with `Binding` as suffix:

- `activity_main.xml` → `ActivityMainBinding`

Data Binding

With Data Binding you can access data from inside the XML. This is achieved with the Observer Pattern. The Layout is the Observer of the Data.
Data Binding (What is Data Binding in an Android Application?) has to be enabled in the Gradle settings. If you want to observe your class see How to make your Class observable for Data Binding?.

How to make Class Observable

Attention: A object witch is used with Data Binding (What is Data Binding in an Android Application?) is *not* automatically observable. You have to change the data source in the following way:

- *observable fields*: this should be used for a few values only
- *observable objects*: this should be used if the whole class should be observed

  - you have to mark with the Annotation `@Bindable` the getters.

How to bind an Event
First you have to enable Data Binding (How to use Data Binding in your Android Application?). Then you have two options:

- *method references*: references directly to the function with a fitting signature
- *listener bindings*: allows the usage of expressions before making a call

One-way vs. two-way
A One-Way binding is Data Binding (What is Data Binding in an Android Application?) or View Binding (What is View Binding in an Android Application?) exclusive. You only use one of them. But often you want a Two-Way Binding. If the data changes the view changes, and if the user changes the UI the data are changed.

Risk of Data Binding
Common risks with Data Binding (What is Data Binding in an Android Application?):

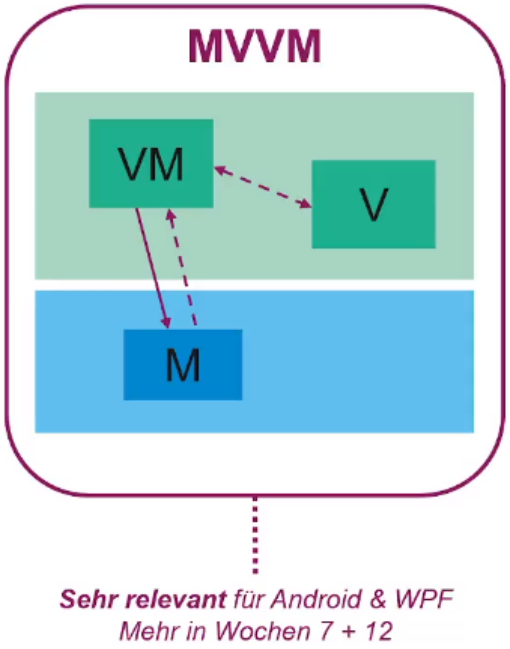
- Pollute the the Model with Android SDK Details
- to much logic in the layout
- difficult to debug
- takes longer for the compilation (I think that is not that bad)
- possible occurrence of invisible observers (What is a Invisible Observer?)

MVVM
The MVVM Pattern consists of three parts:

- *Model*: contains the Domain / business logic
- *View*: contains the graphical UI and user input
- *View Model*: contains the logic form the UI and ensures the communication between Model and View
  - This is often achived over Data Binding
  - What is Data Binding in an Android Application?
  - Data Binding in WPF

The benefits of the MVVM pattern are:

- View Model is easy to test because it does not contain any UI classes
- the View has only visual tasks
- Changes in the Model do not directly affect the View



Lifecycle Aware Component

A Lifecycle Aware Component reacts on state changes of **a other component**. Using this approach the state logic moves from the owner to the observer.

To implement a Lifecylce Aware Component you need two concepts:

- *LifecylceOwner*: manage the state in a `Lifecylce` object
- *LifecylceObserver*: observes the Lifecycle object (and with this the LifecylceOwner)

LiveData Class LiveData is a lifecycle-aware observable (What is a Lifecycle Aware Component?, Observer Pattern) from the Android SDK. The benefit that is also solves the Invisible Observer Problem (What is a Invisible Observer?).

It's an alternative to Observable Fields / Observable Objects (How to make your Class observable for Data Binding?) and is possible to use it as a source for Data Binding (What is Data Binding in an Android Application?) The Observer is notified if enabled and are removed if it is stopped (What is a Invisible Observer?)

1 End