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

AOSP
Android is OpenSource and is called **AOSP**. Google and the other developers normally use the AOSP and add their own services on top of the System. The problem is that if updates happen in the AOSP this often is not deployt to the end user. This would be to much work for the big companies to update their system. Often only big version changes / security patches are applied. This is the reason for a big version fragmentation.

Activity
Activities are the basic building blocks for creating Android Apps. One activity should only do one tasks. The MainActicity is the first Activity started when the app is opened (similar to a main function in Java / Kotlin). The MainActivity is defined in the AndroidManifest. An Activity consists of an XML-file (Layout) and a Java / Kotlin file (logic).

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

Callbacks

Android Developer 2020: Activity Lifecycle
https://bit.ly/2YChVSn
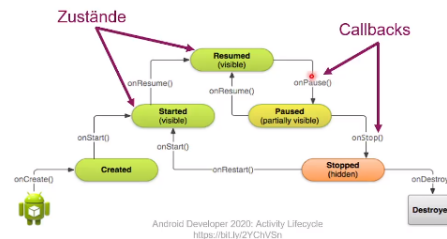
Ressources
Every Android App has some resources. It exist different kind of resources:

- 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: !!Source 1!!

The R Class
The `R` class in the Android SDK is generated during the build. With this class all the IDs of the Ressources in the Androids SDK can be access. The ID is often used to get the Java / Kotlin element to interact with: !!source 2!!

Reach on Event
To respond to a event **listeners** are used: !!source 3!!

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.

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()`

Explicit Intent

An explicit intent is mostly used to call an other component of the own app. As an example to switch the activity:

!!source 4!!

Implicit Intent

An implicit intent is used mostly used call other apps. For example when my app needs to take a picture it can ask the operating system to open an other app to take this photo. Thanks to this concept not every app has to implement everything by itself.

It is important to check if a receiver is installed which can handle the intent. If no receiver is installed and you send your intent, the app will crash with an exception.

!!source 5!!

Gestartete Apps TODO

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)
  - $match_{content}$ (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.

Linear Layout

In the LinearLayout all children are arranged vertically or horizontally.

- `android:orientation=vertical`
- `andoird:orientation=horizontal`
- `android:layout_weight=xx`
  - the larger the number the wider the bar
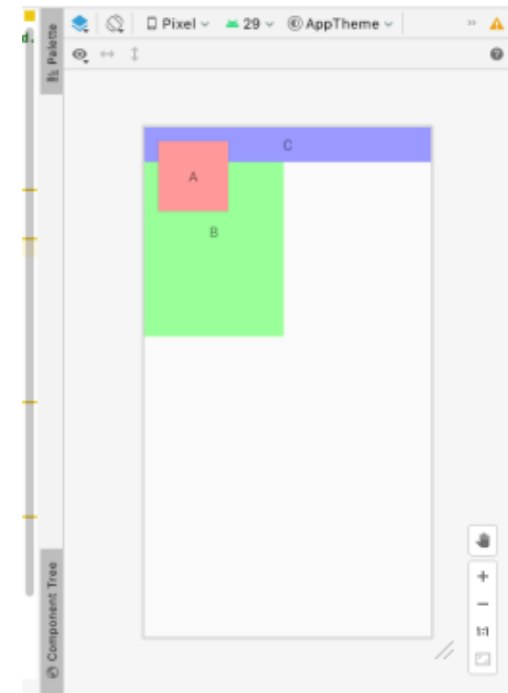 - use it only with `wrap_content`

FrameLayout

In the FrameLayout the children are stacked on each other. As an example a photo app, in the background the live image and on top the control widgets.

Adjustment in the height:

- the later in the XML the higher the control
- set the height manually with `android:translationZ`

RelativeLayout

In the RelativeLayout the elements are arranged relatively to each other. This is often used for a efficient alternative for nested Layouts (Nested Layouts in Android).

2

Jetpacks ConstraintLayout

Today the ConstraintLayout should be used for The Simple Layouts in Android. The ConstraintLayout is similar to the RelativeLayout (The RelativeLayout in Android). The whole Screen is designed using constraints.

Android Studio generated good code (XML) for the ConstraintLayout. The size `0dp` means that the size should be calculated from the constraints.

The ConstraintLayout is **not** in the `android:` namespace. It is in the XML namespace `app:`.

Adapter Layout The AdapterLayout can render a potential infinite set of information. It is not known how many items are available during compile time. For this the Jetpacks RecyclerView is used.

The RecyclerView is an element from the Jetpack Library (Android Jetpack). As the name says it recycles views (The Adapter Layouts in Android).

You have to implement a ViewHolder and an Adapter The ViewHolder inherits from `RecyclerView.ViewHolder` and stores the reference to the view elements. The Adapter inherits from `RecyclerView.Adapter<T>` is used to build from the data an appropriate view. The Adapter must override the following functions:

- `onCreateViewHolder()`: creates a new ViewHolder with references
- `onBindViewHolder()`: attach new content to the ViewHolder
- `getItemCount()`: the size of the underlying collection

cycling
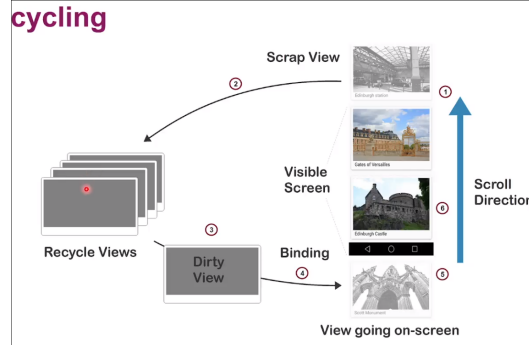
Nested Layout But this should be avoided if possible because this has massive negative impacts on the performance. *Wide and flat hierarchy.*

Widgets

All Widgets in the Android SDK are in the Namespace `android.widget` and inherit from `View`.

- TextView
- Imageview
- Button
- ImageButton
- EditText
- Checkboxes
- Picker (for Example Date picker)
- Floating Action Button (often place at the bottom)
- Radio Buttons
- Seek Bar (slider bar)
- Switches
- Rating Bar
- Spinne (Combobox / Dropdown)

There exits also UI-Widget with out XML. These widgets are generated from the OS over an API.

- Toasts
- Snackbars
- Dialoge
- Notifications

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.
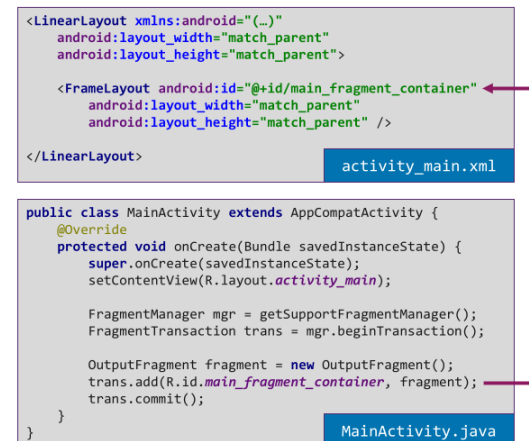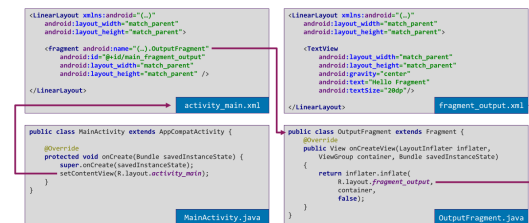
Parameter und Methoden

```java
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;
        fragment = OutputFragment.create("Initial Value");

        trans.add(R.id.main_fragment_container, fragment);
        trans.commit();

        Button button = findViewById(R.id.main_button);
        button.setOnClickListener(v -> {
            fragment.updateText("Updated value");
        });
    }
}
```
MainActivity.java

```java
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);
    }
}
```
OutputFragment.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. !!source 6!!

Styles in Android

```xml
<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

```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

```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

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

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

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
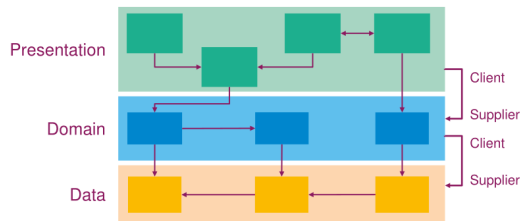
Obeserver Pattern

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
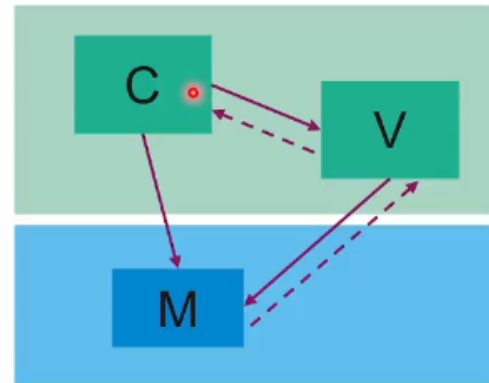
Application class

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.
!!source 7!!
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 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.
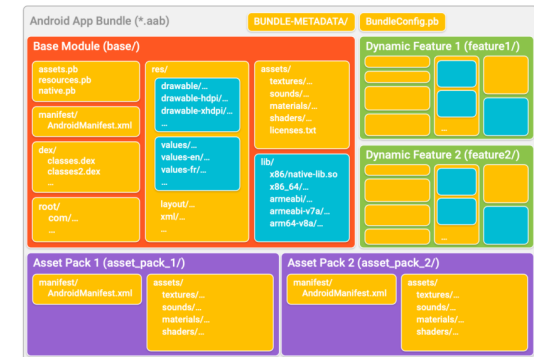
Android Build Process
First all files from the project must be compiled to a APK file (). The resulting APK file (What is a APK file?) is uploaded to the Google Play Store. The Installed App is executed. If the code is already translated to Machine Code it is executed straight away. Else you compile 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 ().
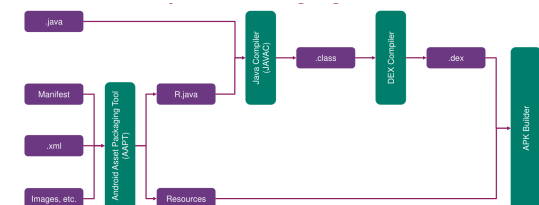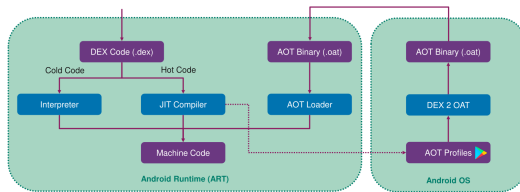
View Binding

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

MVVM

Sehr relevant für Android & WPF
Mehr in Wochen 7 + 12
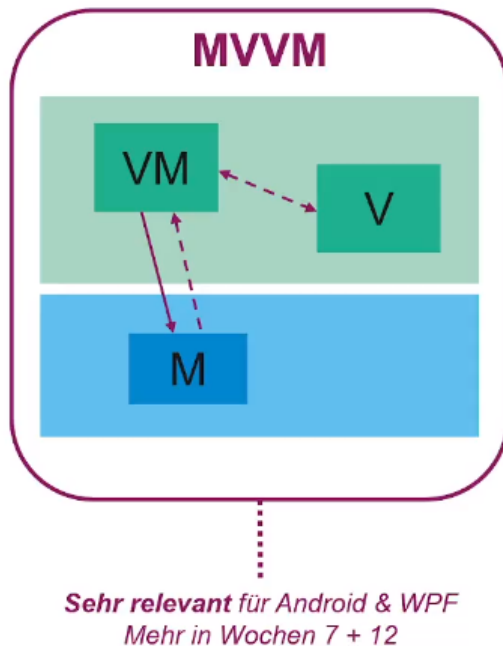
Lifecycle Aware Component

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 Lifecylce-Owner)

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 Code

Source 1 - Access Value from Resource

```
<Button
    android:id="@+id/buttonNavigate" <!-- creates new ID on place -->
    android:text="@string/open_2nd_activity" <!-- references open_2nd_activity in strings.xml --> />
```

Source 2 - R Class example

```
var button = (Button)this.findViewById(R.id.btnLogIn);
```

Source 3 - Reach on an Event using listeners

```
final TextView textView = this.findViewById(R.id.text_example);
Button button = this.findViewById(R.id.button_example);

button.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        textView.setText("Button pressed");
```

```
        }
    }
);

// Java 8 (Lambda)
button.setOnClickListener(v -> { /**/ });
```

Source 4 - Explicit Intent

```
// SecondActivity.kt
class SecondActivity : AppCompatActivity() {
    companion object {
        private const val MESSAGE_KEY = "MESSAGE";
        fun createIntent(context: Context, message: String): Intent {
            val intent = Intent(context, SecondActivity::class.java)
            intent.putExtra(MESSAGE_KEY, message)
            return intent
        }
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_second)

        val textBox = findViewById<TextView>(R.id.textOutput)
        val message = intent.extras?.getString(MESSAGE_KEY)
        textBox.text = message
    }
}


// Some other activity
startActivity(SecondActivity.createIntent(this, "Hallo␣Welt"))
```

Source 5 - Implicit Intent

```
val intent = Intent(Intent.ACTION_VIEW, Uri.parse("https://www.ost.ch"))
val hasReceiver = intent.resolveActivity(getPackageManager()) != null;


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

Source 6 - Fragment with parameters

```
public class OutputFragment extends Fragment {
    private TextView textOutput;
    public static OutputFragment create(String text) {
        Bundle args = new Bundle();
        args.putString("Key", text);
```

```java
        OutputFragment fragment = new OutputFragment();
        fragment.setArguments(args);
        return fragment;
    }
}
```

Source 7 - The application class

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

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

    // .. more callbacks
}
```