

# Inhalt

- Kotlin
  - Vorstellung Java App + Kotlin Conversion
  - Kodein
  - Room
  - Testing
- 

Day 1

- Databinding
- Functional Programming
- RxKotlin
- Couroutines
- StandardLibs + Extensions

Day 2

# Kotlin – Databinding

## or the universal ViewHolder-Pattern

*„Data binding is a general technique that binds data sources from the provider and consumer together and synchronizes them.“ - Wikipedia*

- Macht „findViewById()“ überflüssig.
- Ermuntert uns UI- und Geschäfts Logik zu trennen
- Weniger UI Code
- Einfache Datensynchronisation zwischen Datenquellen und UI-Elementen
- Man kann Event-Listener direkt im XML binden
- Bietet die Möglichkeit eigene Getter/Setter für std. Attr. zu generieren

## Wie wird Databinding integriert I

### Durch 4 Dinge:

- Aktivieren des Databinding Features im build.gradle
- Dependency hinzufügen:  
**kapt**"com.android.databinding:compiler:\$plugin\_version"
- apply plugin: 'kotlin-kapt'
- Im entsprechenden Layout muss das Root Tag mit einem dem neuen Tag **layout** umschlossen werden

```
android {  
    ...  
    dataBinding {  
        enabled = true  
    }  
}  
  
<?xml version="1.0" encoding="utf-8"?>  
<layout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
    xmlns:app="http://schemas.android.com/apk/res-auto">  
  
    <RelativeLayout  
        xmlns:tools="http://schemas.android.com/tools"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
    ...
```

# Wie wird Databinding integriert II

```
class MainActivity : AppCompatActivity(){
```

```
    override fun onCreate(savedInstanceState: Bundle?) {
```

```
        super.onCreate(savedInstanceState)
```

```
        // The old way
```

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

```
        // val viewFirstName = findViewById<TextView>(T.id.text_view_firstname)
```

```
        // viewLastName = findViewById<TextView>(T.id.text_view_lastname)
```

```
        // The new way
```

```
        val binding : ActivityMainBinding = DataBindingUtil.setContentView(this,R.layout_activity_main)
```

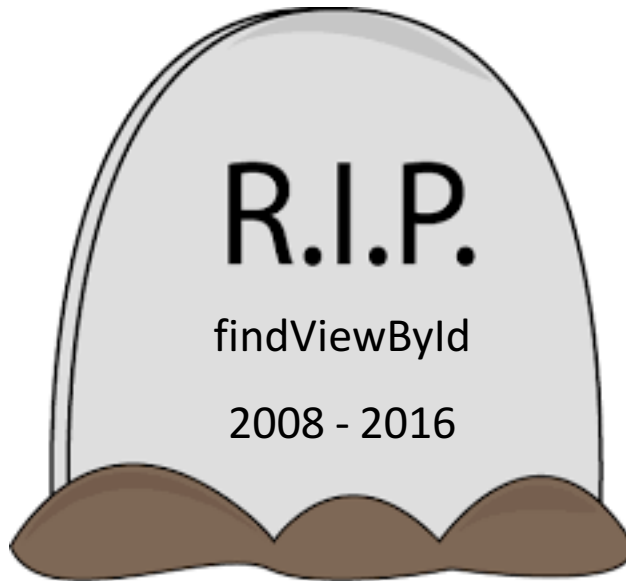
```
        binding.textviewFirstName = "Chris"
```

```
        binding.textviewLastName = "Gauch"
```

```
    }
```

```
}
```

Referenzierung von UI-Elementen  
(Die ID im Layout entspricht dem  
Namen im Binding)



findViewById ist enorm in performant, jeder Aufruf dieser Methode traversiert die gesamte Layout Hierarchie. Beim Databinding geschieht dies nur ein einziges Mal. (Butteknife basiert auf findViewById...)

# Still too much Boiler Plate Code

Bis jetzt haben wir nur „findViewById“ durch den Binding Aufruf ersetzt. Um nun wirkliches DataBinding zu implementieren bietet das Framework zusätzlich die Möglichkeit Klassen direkt im Layout zu referenzieren.

## Data Tag I

```
<data>  
  <import type="android.view.View"/>  
</data>  
  
<TextView  
  android:text="@{user.lastName}"  
  android:layout_width="wrap_content"  
  android:layout_height="wrap_content"  
  android:visibility="@{user.isAdult ? View.VISIBLE : View.GONE}"/>
```

Simpler Import von Klassen, um z.B.  
auf statische Definition zugreifen zu können.



# All the power to the layout I

## Data Tag – Type Alias

Falls es beim Import von Klassen zu Namenskonflikten kommt, ist es zudem möglich einen type alias zu definieren.

```
<import type="android.view.View"/>  
<import type="com.example.real.estate.View" alias="Vista"/>
```

## Data Tag – Statische Felder

```
<data>  
    <import type="com.example.MyStringUtils"/>  
</data>  
...  
<TextView  
    android:text="@{MyStringUtils.capitalize(user.lastName)}"  
    android:layout_width="wrap_content"
```

Innerhalb der Binding Expression ist es möglich auf stat. Methode zuzugreifen,  
Math. Operationen aufzuführen,  
Switches einzubauen...



# All the power to the layout II a

Some examples:

Tenary Operator:

```
android:text="@{user.lastName != null ? user.lastName : user.name}" />
```

Simple Type Conversion

```
android:text="@{String.valueOf(user.age)}" />
```

Resources and Strings

```
android:padding="@{isBig ? @dimen/bigPadding : @dimen/smallPadding}"
```

String formatting

```
android:text="@{@string/nameFormat(firstName, lastName)}" />
```

Inline plurals

```
android:text="@{@plurals/employee(employeeCount)}"
```

# All the power to the layout II b

## View Attributes

```
<CheckBox
    android:id="@+id/showName"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>
<TextView
    android:text="@{user.firstName}"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:visibility="@{showName.checked ? View.VISIBLE : View.GONE}"
/>
```

# All the power to the layout III

## Variables

Es können eine beliebige Anzahl an Variablen Definitionen innerhalb des Data Tags vorkommen. Jede Variable Beschreibt eine Eigenschaft (Objekt) welches auf dem Layout gesetzt wird, damit es in der Binding Expression verwendet werden kann.

Falls dieses Objekt resp. die Property welches innerhalb der Binding Expression verwendet wird, das Observable Interface oder Teil einer Observable Collection ist, wird die Änderung direkt im Layout referenziert.

```
<data>  
  <variable  
    name="user"  
    type="com.example.myapp.model.User"/>  
</data>
```

```
<TextView  
  android:layout_width="wrap_content"  
  android:layout_height="wrap_content"  
  android:text="@{user.name}"/>
```

# All the power to the layout IV

## One-Way-Binding

```
...
<data>
  <variable
    name="item"
    type="com.example.myapp.model.Item"/>
</data>

...
<TextView
  android:text="@{item.title}"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"/>

...
```

```
data class Item(
    private var _title: String,
    private var _url: String): BaseObservable(){

    val title: String
        @Bindable get(): _title
        set(value){
            _title = value
            notifyPropertyChanged(BR.title)
        }

    Val url: String
        @Bindable get(): _url
        set(value){
            _url = value
            notifyPropertyChanged(BR.url)
        }
}
```

# All the power to the layout V

## One-Way-Binding – The easy way

Jedes mal cutom Getter und Setter zuschreiben ist ziemlich mühselig, daher gibt es im **databinding package** **ObservableField** und das **primitive Gegenstück**. Um diese zu verwenden müssen einfach **public fields** in der Entsprechenden Klasse deklariert werden.

```
import android.databinding.ObservableField
import android.databinding.ObservableInt
```

```
Data class Foo(val bar: ObservableInt,
               var name: ObservableField<String>)
```

One-Way-Binding  
→

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@{foo.name}"/>
```

# All the power to the layout VI

## Two Way Binding - Introduction

Zuerst nochmals ein Beispiel für One-Way-Binding:

<EditText

```
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="@{ user.firstName}"  
    android:id="@+id/firstName" />
```

Bei Verwendung dieser Nomenklatur, müsste man z.B. Mittels dem Callback „afterTextChanged“ die Änderung des Users zurück in das Model schreiben.

Einfache geht es mit dem **Two-way-binding Operator** `@={}` (eingeführt in AS 2.1) welcher automatisch das Model mit den Änderungen des Objektes updated.

Leider funktioniert das Two-Way-Binding nicht für jedes Attribut, nur jene welche eine Event Notifikation der Änderung Auslösen. Dies beinhaltet so gut wie jede Input Komponente von Android

# All the power to the layout VII

## Event Handling- Introduction

Das DataBinding Framework bietet 3 Möglichkeiten um ein Event Listener zu den Layouts hinzuzufügen:

- Listener Objects
- Method References
- Lambda Expressions

# All the power to the layout VIII

## Event Handling- Listener Objects

Für **jeden View** mit einem Listener welche durch eine **set\*** (setOnClickListener) Methode (im Gegensatz zu add\* addOnLayoutChanged) hinzugefügt wird, kann der Listener auf folgende Art referenziert werden.

```
<View android:onClick="@{callbacks.clickListener}" .../>
```

Wobei der Listener mit einem Getter oder einem public field zur Verfügung gestellt werden muss:

```
class Callbacks {  
    var clickListener: View.OnClickListener? = null  
}
```



# All the power to the layout IX

## Event Handling- Method References

Mit Methoden Referenzen ist es möglich eine statische oder Instanz Methode an jeden Callback zu binden, solange **diese Methode die selben Parameter und den selben Rückgabe Wert hat wie der Callback** an den sie gebunden sind.

```
<EditText  
    android:afterTextChanged="@{callbacks::nameChanged}" .../>
```

```
class Callbacks {  
    fun nameChanged(editable: Editable) {  
        //...  
    }  
}
```

Wichtig ist hier, das das Attribut „afterTextChanged“ den gleichen Namen hat, wie diejenige Methode im korrespondieren Listener (TextWatcher.afterTextChanged(e: Ediable?)

## All the power to the layout X

### Event Handling- Lambda Expressions

Es ist ebenfalls möglich eine Lambda Expression zur Verfügung zu stellen und dieser jedmöglichen Parameter zu übergeben.

```
<EditText  
    android:afterTextChanged="@{(e)->callbacks.textChanged(user, e)}"  
... />
```

```
class Callbacks {  
    fun textChanged(user: User, editable: Editable) {  
        if (user.hasName()) {  
            //...  
        } else {  
            //...  
        }  
    }  
}
```

Wichtig ist hier die Anzahl der Parameter welche man übergibt. Es **gilt ALLE** oder **keine**, aber man kann nicht mixen.

```
<EditText  
    android:afterTextChanged="@{()->callbacks.textChanged(user)}"  
... />
```

# All the power to the layout X

Event Handling- Lambda Expressions and View References

<EditText

```
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:id="@+id/firstName"  
    android:text="@={user.firstName}" />
```

<CheckBox

```
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:onCheckedChangeListener="@{()->handler.checked(firstName)}" />
```

# Binding Adapter I

## Ein ColorPicker

```
class ColorPicker : View (private val color: Int){  
    private var color: Int = 0;  
    public setColor(color: Int) {  
        this.color = color;  
        invalidate();  
    }  
    public getColor(): Int {  
        return color;  
    }  
    public addListener(listener: OnColorChangeListener ) {  
        //...  
    } public removeListener(listener: OnColorChangeListener ) {  
        //...  
    }  
}
```

```
<com.example.myapp.ColorPicker  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    app:color="@{color}"/>
```

Das Setzen des Attributes „color“ braucht in diesem speziellen Fall keinen speziellen Code, da das Framework automatisch nach einem Setter mit dem selben Namen wie des Attributes sucht.

# Binding Adapter II

Binding Adapters bieten einem die Möglichkeit komplexe Setter zu implementieren. Komplex bedeutet in diesem Fall das jede erdenkbare Transformation / Logik ausgeführt werden kann.

```
<ImageView  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    app:imageUrl="@{product.imageUrl}"/>
```

Das DataBinding Framework würde bei dieser Definition auf der Klasse ImageView nach einer Methode „setImageUrl“ („set“+ Attribut-Name) suchen und diese natürlich nicht finden. Hier kommen nun Binding Adapter ins Spiel

```
@BindingAdapter("imageUrl")  
public setImageUrl(imageView : ImageView, url : String?) {  
    if (url == null or url.isEmpty()) {  
        Picasso.with(imageView.getContext()).load(R.drawable.placeholder).into(imageView)  
    } else {  
        Picasso.with vm(imageView.getContext()).load(url).into(imageView)  
    }  
}
```

# Binding Adapter III

Die Binding Adapter Annotation nimmt den Attribut Namen als Parameter entgegen (Für Attribute des Android -Namespace muss der voll-qualifizierte Name inkl. „android“ angegeben werden). Der erste Parameter ist dabei immer der Typ des Ziel Views. Der zweite ist der Wert welcher gesetzt werden soll.

```
<ImageView  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    app:imageUrl="@{product.imageUrl}"  
    app:placeholder="@{@drawable/reddit_placeholder}" />
```

Wenn ein Adapter mehrere Attribute verwaltet,  
So müssen diese in einer Liste angegeben werden,  
dabei muss die Reihenfolge mit Derjenigen im xml  
übereinstimmen.

```
@BindingAdapter(value={"imageUrl", "placeholder"}, requireAll=false)  
public setImageUrl(imageView: ImageView, url: String?,placeholder: Drawable?) {  
    if (url == null or url.isEmpty()) {  
        Picasso.with(imageView.getContext()).load(placeholder).into(imageView)  
    } else {  
        Picasso.with(imageView.getContext()).load(url).into(imageView)  
    }  
}
```

## Exercise 01

- Füge auf dem Login Screen einen neuen TextView hinzu welcher aussagt wie stark das Passwort ist
  - Binde mittels one-way or two-way Binding diesen neuen TextView an den Passwort EditText
  - Die zu ändernden Stellen sind mit „*Excercise-DataBinding-01*“ markiert.
- 
- *Es gibt hier nicht nur eine Lösung...*

# Inverse - Binding Adapter I

Do you remember the ColorPicker ?

```
class ColorPicker : View (private val color: Int){
    private int color;
    public setColor(color: Int) {
        this.color = color;
        invalidate();
    }
    public getColor(): Int {
        return color;
    }
    public addListener(listener: OnColorChangeListener ) {
        //...
    } public removeListener(listener: OnColorChangeListener ) {
        //...
    }
}
```

```
<com.example.myapp.ColorPicker
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:color="@{color}"/>
```

zu

```
<com.example.myapp.ColorPicker
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:color="@={color}"/>
```



# Inverse - Binding Adapter III

Was wenn der Rückgabe Wert nicht übereinstimmt, oder man noch etwas mehr tun möchte als nur eine Getter aufzurufen ?

```
object ColorPickerBindingAdapters {  
    @InverseBindingAdapter(attribute = "color")  
    fun getColor(view: ColorPicker): Int {  
        return convertColorToInt(view.getColor())  
    }  
  
    @BindingAdapter("color")  
    fun setColor(view: ColorPicker, color: Int) {  
        view.setColor(convertIntToColor(color))  
    }  
}
```

**Wichtig: Vermeidet endlos Schleifen !**

```
@BindingAdapter("color")  
fun setColor(view: ColorPicker, color: Int) {  
    if (color != view.getColor()) {  
        view.setColor(color)  
    }  
}
```

# Two ViewHolder Patterns ?

Der RecyclerView hat 2 wichtige Methoden welche benutzt werden um die Daten an die Views zu binden:

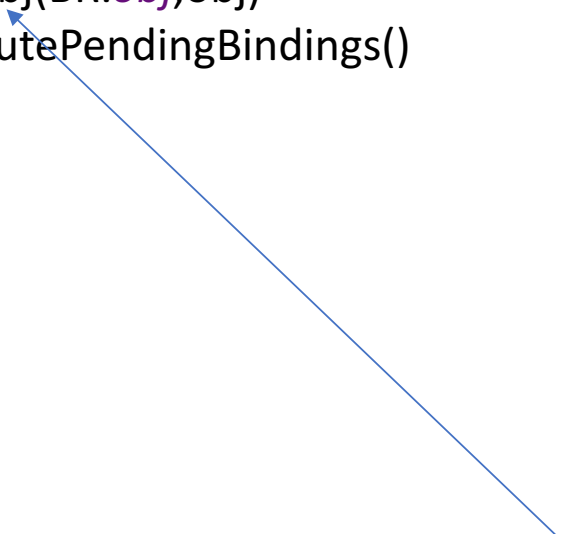
**fun** onCreateViewHolder(parent: ViewGroup, viewType: Int): RecyclerView.ViewHolder

**fun** onBindViewHolder(holder: RecyclerView.ViewHolder, position: Int)

Idelarweise würden nur die generierte Binding Klasse von der Funktion onCreateViewHolder zurückgegeben werden, (ist im Prinzip beides DataBinding) allerdings erbt diese generierte Klasse nicht von RecyclerView.ViewHolder, daher Muss die Binding Klasse vom ViewHolder gehalten werden.

# ViewHolder – Binding I

```
class MyViewHolder(val binding:ItemBinding):  
    RecyclerView.ViewHolder(binding.root){  
    open fun bind(obj: Item){  
        binding.setObj(BR.obj,obj)  
        binding.executePendingBindings()  
    }  
}
```



**Anmerkung:** Das **executePendingBindings()** ist wichtig, Dies forciert die Bindings sofort durchzugehen, anstatt bis zum nächsten Frame zuwarten. Da der RecyclerView direkt im Anschluss an onBindViewHolder() den View vermisst, kann es sein, dass wenn executePendingBindings() nicht ausgeführt wird, der View noch mit den „alten“ Binding Daten gemessen wird, und daher die falsche Größe berechnet wird.

**setObj()** ist typisiert, der Name leitet sich aus dem Name des Bindings ab.

## ViewHolder – Binding II

```
fun onCreateViewHolder(parent: ViewGroup, viewType: Int): MyViewHolder {  
    val inflater = LayoutInflater.from(parent.context)  
    val itemBinding = ItemBinding.inflate(inflater, parent, false)  
    return MyViewHolder(itemBinding)  
}
```

```
fun onBindViewHolder(holder: MyViewHolder, position: Int) {  
    val item = getItemForPosition(position)  
    holder.bind(item)  
}
```

Der obige Ansatz reduziert den Code schon ziemlich. Es müssen keine Views referenziert werden oder manuell Werte gesetzt werden, aber wäre es nicht schön einen universellen ViewHolder zu haben ?

# The Universal ViewHolder I

```
class MyViewHolder(private val binding: ViewDataBinding : RecyclerView.ViewHolder(binding.getRoot())) {  
  
    public bind(obj: Object) {  
        binding.setVariable(BR.obj, obj)  
        binding.executePendingBindings()  
    }  
}
```

Wir verwenden den alg. Typ **ViewDataBinding** und nicht den Item spezifischen. Ebenso rufen wir die **typ-unspezifische** Methode **setVariable()** und nicht **setObj()** Methode auf. Dieser Ansatz bedingt aber das in der entsprechenden Layout Datei eine Variable Namens „**obj**“ definiert wird.

```
<data>  
    <variable name="obj" type="Item"/>  
...
```

Der Typ von Item spielt dabei keine Rolle, da wir ja die typ-unspezifische Methode **setVariable()** verwenden.

# The Universal ViewHolder II

```
public abstract class MyBaseAdapter : RecyclerView.Adapter<MyViewHolder>() {  
    public MyViewHolder onCreateViewHolder(ViewGroup parent,int viewType) {  
        LayoutInflater inflater = LayoutInflater.from(parent.getContext());  
        ViewDataBinding binding = DataBindingUtil.inflate(inflater, viewType, parent, false);  
        return new MyViewHolder(binding);  
    }  
    public void onBindViewHolder(MyViewHolder holder,int position) {  
        Object obj = getObjForPosition(position);  
        holder.bind(obj);  
    }  
    @Override  
    public int getItemViewType(int position) {  
        return getLayoutIdForPosition(position);  
    }  
    protected abstract Object getObjForPosition(int position);  
    protected abstract int getLayoutIdForPosition(int position);  
}
```

# Exercise 02

- Ändere die derzeitigen Recyclerviews (Detail und Overview) so ab das sie mittels Databinding die ViewHolder darstellen.
- Betroffene Klassen
  - Alle Klassen im Package `ch.zuehlke.sbb.reddit.features.news.overview.adapter.impl`
  - OverviewFragment und DetailFragment
  - Die beiden Layout Dateien:
    - `item_detail.xml`
    - `Item_overview`
- Auch hier gilt es gibt nicht nur eine Lösung.