

Projektarbeit

Alexander Lange

28. Februar 2022

Zusammenfassung

Dieser Bericht erläutert die Umgestaltung der Prüfplan-Applikation der Fachhochschule Bielefeld. Es wurden dabei einige Änderungen sowohl im Frontend als auch im Backend vorgenommen. Ziel war es zum einen, die Nutzererfahrung durch eine schönere Applikation zu verbessern und zum andern die Weiterarbeit durch einen besser strukturierten und dokumentierten Code zu verbessern. Dabei wurde viel Wert auf eine gute Programmlogik und überschaubare Klassen und Methoden gelegt, sowie die Trennung von verschiedenen Aufgaben mit Hilfe des Model-View-ViewModel-Patterns.

Inhaltsverzeichnis

1 Frontend	5
1.1 Themen	5
1.1.1 Attribute	5
1.1.2 Farben	6
1.1.3 Darkmode	7
1.1.4 Themendefinition	7
1.2 Filter	8
1.2.1 Implementierung	9
1.2.2 Dialog Fenster	10
1.2.3 Suchleiste	12
1.3 Navigation	12
1.3.1 Neuimplementierung des Fragmentwechsels	14
1.3.2 TabLayout	16
1.3.3 ViewPager	18
1.4 Layouts	19
1.4.1 Alle Prüfungen	19
1.4.2 Favoriten	20
1.4.3 Einstellungen	22
1.4.4 Studiengänge Verwalten	25
2 Model-View-ViewModel-Pattern	27
2.1 Modell	27
2.1.1 Room	27
2.1.2 Retrofit2	29
2.1.3 Repository	30
2.1.4 SharedPreferences Repository	31
2.2 ViewModel	31
2.2.1 Coroutines	31
2.2.2 Verwendung mehrerer ViewModels	33
2.3 View	33
2.3.1 ViewModelProvider	33
2.3.2 LiveData Observer	34
3 CalendarIO	35
3.1 Smartphone Kalender	35
3.2 Events	35
3.2.1 InsertionType	35
3.2.2 Einfügen	35

3.2.3	Update	35
3.2.4	Löschen	35
3.3	Id Management	35
4	Update Manager	36
5	Push Service	37
6	Background Worker	38
7	Themenattribute	40
8	Codeausschnitte	41
8.1	Filter	41
8.1.1	Filervalidierung	41
8.2	Suchleiste	42
8.3	Retrofit2	42
8.3.1	GSON Objekte	42
8.4	ViewModel	43
8.4.1	updateEntry	43
9	Kotlin	45
9.1	Vorteile gegenüber Java	45
9.2	Variablen	45
9.2.1	Definition	46
9.2.2	Null-Sicherheit	46
9.3	Methoden	47
9.4	Datenstrukturen	47
9.4.1	Array	48
9.4.2	ArrayList	49
9.4.3	Set	49
9.4.4	Map	49
9.4.5	List	49
9.5	Schleifen	50
9.5.1	While	50
9.5.2	For	50
9.5.3	Foreach	51
9.6	Verzweigungen	51
9.6.1	If-Verzweigung	51
9.7	Klassen	52
9.7.1	Definition	52
9.7.2	Konstruktoren	52
9.7.3	Vererbung	54
9.8	Erweiterungsmethoden	55
9.9	Object und Companion	55
9.10	Interfaces	56
9.10.1	Anonymes Interface	57
9.10.2	Vereinfachung	58
9.11	Coroutines	59
9.11.1	Suspend	59
9.11.2	Coroutine Scope	60

9.12	Vorteile gegenüber Java	62
9.13	Variablen	62
9.13.1	Definition	62
9.13.2	Null-Sicherheit	62
9.14	Methoden	63
9.15	Datenstrukturen	64
9.15.1	Array	64
9.15.2	ArrayList	65
9.15.3	Set	65
9.15.4	Map	66
9.15.5	List	66
9.16	Schleifen	66
9.16.1	While	66
9.16.2	For	67
9.16.3	Foreach	67
9.17	Verzweigungen	67
9.17.1	If-Verzweigung	67
9.18	Klassen	68
9.18.1	Definition	68
9.18.2	Konstruktoren	69
9.18.3	Vererbung	70
9.19	Erweiterungsmethoden	71
9.20	Object und Companion	71
9.21	Interfaces	72
9.21.1	Anonymes Interface	74
9.21.2	Vereinfachung	74
9.22	Coroutines	76
9.22.1	Suspend	76
9.22.2	Coroutine Scope	76
10	Weiterarbeit	79
10.1	Farbthemen	79
10.2	Strings	80
10.3	Bezeichnungen	80
10.4	MVVM	80
10.4.1	Model	80
10.4.2	ViewModel	80
10.4.3	View	80
10.5	Dokumentation	81
11	Lifecycles	82
11.1	Activity-Lifecycle	83
11.2	Fragment-Lifecycle	84

Einleitung

Die Digitalisierung der Welt schreitet voran. Das Smartphone ist mittlerweile zu einem Grundgegenstand eines jeden Menschen geworden. Tag für Tag werden neue Apps entwickelt, die einem das Leben einfacher machen sollen. Im Zuge dessen entwickelt die Fachhochschule Bielefeld eine eigene App, mit der sich jeder Student seinen eigenen Prüfungsplan zusammen basteln kann. Im Zentrum dabei steht eine Übersicht über alle anstehenden Prüfungen, von denen sich der Student die für Ihn relevanten auswählen kann, und bei Bedarf werden diese auch automatisch mit dem Kalender synchronisiert.

Ziel dieser Arbeit war es, die bereits bestehende App zu verbessern. Zum einen sollte die Programmiersprache von Java auf Kotlin geändert werden, zum anderen sollte die Benutzerfreundlichkeit und die Weiterarbeit angenehmer gestaltet werden.

Damit eine App benutzerfreundlich ist, muss auf einige Sachen geachtet werden. Dazu zählen:

- Übersichtlichkeit
- Personalisierbarkeit
- Verständlichkeit

Um diese Punkte zu erfüllen, wurden die einzelnen Seiten neu aufgebaut und unnötige Sachen wurden entfernt. Zudem wurden einige neue Einstellungsmöglichkeiten hinzugefügt, mit denen der Nutzer die App für sich persönlich anpassen kann. Dazu zählen unter anderem verschiedene Farbthemen und ein Darkmode.

Um die Weiterarbeit zu erleichtern, wurde der Programmcode neu strukturiert. Es wurde das sogenannte Model-View-ViewModel-Pattern umgesetzt, wodurch verschiedene Anwendungsbereiche wie Datenzugriffe, App Logik und User Interface von einander abgekoppelt werden, und es wurde eine umfangreiche Dokumentation erstellt.

In den folgenden Kapitel gibt es zuerst eine kleine Einführung in die Programmiersprache Kotlin, anschließend werden die Veränderungen im Backend und im Frontend näher erläutert. Zum Schluss gibt es noch ein paar kurze Anmerkungen und Regeln zur Weiterarbeit an dem Projekt, damit die App auch weiterhin gut strukturiert bleibt.

Kapitel 1

Frontend

Da die Applikation vor der Veränderung sehr unübersichtlich aufgebaut war, wurden einige Veränderungen vorgenommen, um die Benutzererfahrung zu steigern. Dabei wurde viel Wert auf ein einfaches Design gelegt, in dem die Wichtigen Elemente gut hervorgehoben werden. Zudem wurde die Navigation angepasst um eine flüssigere Interaktion zu ermöglichen.

1.1 Themen

Farben sind ein essenzieller Bestandteil einer jeden App. Eine schlechte Farbgebung kann einen großen Einfluss auf die Benutzererfahrung haben. Für ein gutes Design sollten wenige verschiedene Farben eingesetzt werden, welche sich auch gut kombinieren lassen. Da jeder Mensch sich mit anderen Farben wohl fühlt, wurden verschiedene Themen hinzugefügt zwischen denen der Benutzer sich ein Farbeschema aussuchen kann. Um dies zu erreichen, mussten allgemeine Farbattribute definiert werden, welche die zuvor fest implementierten Farben ersetzen und somit eine dynamische Farbgebung der App zulassen.

1.1.1 Attribute

Attribute stellen allgemein gültige Parameter dar, die für die gesamte App verwendet werden können. Dabei sind deren Werte für jedes Thema verschieden. Es gibt von Android Studio bereits einige Attribute, es wurden allerdings noch weitere hinzugefügt um für mehr Abwechslung zu sorgen. Die neu hinzugefügten Attribute sind in der Datei **attrs.xml** definiert. Sie bilden das Grundgerüst für die verschiedenen Themen. Innerhalb der XML-Dateien kann mit dem Befehl `?attr/` auf die verschiedenen Attribute zugegriffen werden. Für den Zugriff im Programmcode wurde die Funktion 1.1 in der Klasse **Utils** erstellt, die es ermöglicht, eine Farbe über die Attribut Id zu beziehen. Auf die Attribut Id kann mit dem Befehl `R.attr.` zugegriffen werden.

```
1 fun getColorFromAttr(@AttrRes attrColor:Int, context: Context,  
2                     typedValue: TypedValue= TypedValue(), resolveRes:Boolean=  
3                     true): Int{  
2     context.theme.resolveAttribute(attrColor,typedValue,  
3         resolveRes)  
3     return typedValue.data
```

} Codeausschnitt 1.1: Zugriff auf eine Farbe anhand der Attribut Id

1.1.2 Farben

Zuvor bestand die Applikation aus einer festen Farbe für die Navigationsleiste und die Actionbar, sowie verschiedene fest definierten Farben für die einzelnen grafischen Elemente. Das neue Farbschema besteht aus einer Primärfarbe, die das grundlegende Aussehen bestimmt, einer Akzentfarbe um einzelne Elemente hervorzuheben sowie einer Hintergrundfarbe. Um für ein wenig Abwechslung zu sorgen, gibt es neben der Primärfarbe noch eine hellere und eine dunklere Version. Für jede dieser Farben ist auch noch eine Vordergrundfarbe definiert, um Text auf den Elementen gut sichtbar darstellen zu können. Das gesamte Farbschema ist im Anhang 7 aufgelistet. Die Abbildungen 1.1 und 1.2 zeigen den Unterschied zwischen dem neuen und dem alten Thema.



Abbildung 1.1: Aussehen der App vor der Implementierung des neuen Farbschemas



Abbildung 1.2: Aussehen der App nach der Implementierung des neuen Farbschemas

Die Farben werden in der Datei **colors.xml** definiert. Hier werden alle Farben festgelegt, welche verwendet werden um später die Themen zu bilden. Grund dafür ist die Implementierung eines Darkmodes, welche in Abschnitt 1.1.3 erklärt wird. In der Datei **colors.xml** werden allerdings noch keine Attribute de-

finiert, dies erfolgt erst bei der Erstellung des Themas in der Datei **styles.xml**. Näheres dazu im Abschnitt 1.1.4.

1.1.3 Darkmode

Ein weiteres Feature, welches hinzugefügt wurde, ist der Darkmode. Dieser bildet für jedes Thema eine alternative, die aus überwiegend dunklen Farben besteht. Dies ist zum einen schonender für die Augen, zum anderen kann es aber auch beim Energiesparen helfen. Android Studio bietet von sich aus bereits eine Möglichkeit, einen Darkmode umzusetzen. Ausgangspunkt dafür ist der Ordner **values-night**. In diesem können alternative XML-Dateien angelegt werden, welche beim Wechsel zum Darkmode die entsprechenden Dateien ersetzen. In diesem Fall wurde eine neue **colors.xml** Datei angelegt, in welcher allen Farben neue Werte zugewiesen werden. In beiden **colors.xml** Dateien die Namen übereinstimmen. Der Unterschied zwischen aktiviertem und nicht aktiviertem Darkmode ist in den Abbildungen 1.3 und 1.4 zu sehen.



Abbildung 1.3: Aussehen der App ohne Darkmode

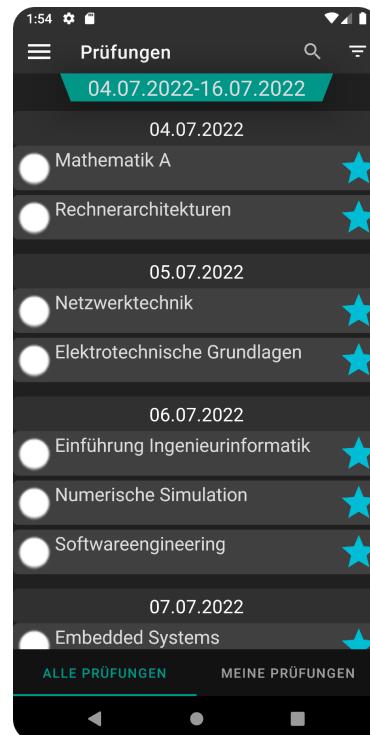


Abbildung 1.4: Aussehen der App mit Darkmode

1.1.4 Themendefinition

Mit Hilfe der Attribute und Farben können die verschiedenen Themen erstellt werden. Dies passiert in der Datei **styles.xml**. Dort kann für jedes Thema ein neuer *style* erstellt werden, welcher den Attributen die entsprechenden Farben

zuweist. Für alle Themen wurde zudem ein *BaseTheme* erstellt, in welchem Farben festgelegt wurden, die für alle Themen gleich sein sollen. Dieses Thema muss jedes neue Thema als *parent* erben. Ein Beispiel für die Implementierung eines Themas ist in Abbildung 1.2 gezeigt

```

1 <style name="Theme.AppTheme_1" parent="BaseTheme">
2     <item name="themeName">@string/Theme1_ThemeName</item>
3
4     <item name="colorPrimary">@color/Theme1.colorPrimary</
5         item>
6     <item name="colorOnPrimary">@color/Theme1.
7         colorOnPrimary</item>
8
9     <item name="colorPrimaryDark">@color/Theme1.
10        colorPrimaryDark</item>
11     <item name="colorOnPrimaryDark">@color/Theme1.
12        colorOnPrimaryDark</item>
13
14     <item name="colorPrimaryLight">@color/Theme1.
15        colorPrimaryLight</item>
16     <item name="colorOnPrimaryLight">@color/Theme1.
17        colorOnPrimaryLight</item>
18
19     <item name="colorAccent">@color/Theme1.colorAccent</
20         item>
21     <item name="colorOnAccent">@color/Theme1.colorOnAccent</
22         item>
23
24     <item name="colorBackground">@color/Theme1.
25        colorBackground</item>
26     <item name="colorOnBackground">@color/Theme1.
27        colorOnBackground</item>
28
29     <item name="actionMenuTextColor">@color/Theme1.
30        colorOnPrimaryDark</item>
31
32 </style>
```

Codeausschnitt 1.2: Implementierung eines Themas

1.2 Filter

Ein weiteres Problem bei der vorherigen App war die Unübersichtlichkeit. Ein Beispiel ist die Suche nach einer bestimmten Prüfung. Zuvor gab es hierfür ein eigenes Fenster, indem nach bestimmten Kriterien gesucht werden konnte. Zusätzlich gab es auch ein Fenster, in welchem nach einem bestimmten Wahlmodul gesucht werden konnte. Um beides zu vereinfachen, wurde die Suche durch einen Filter ersetzt. Die Grundidee war dabei, dass der Benutzer Im Filter verschiedene Kriterien auswählen kann und im Anschluss nur noch diejenigen Prüfungen angezeigt werden, die diesen Kriterien entsprechen. Eine weitere Motivation dabei war, dass der Filter sowohl für die Favoriten als auch die gesamten Prüfungen gelten soll. Das heißt, wenn der Benutzer einen Filter eingestellt hat, soll dieser beibehalten werden, auch wenn er von der Gesamtübersicht zu den Favoriten wechselt und umgekehrt.

1.2.1 Implementierung

Um diesen Anforderungen gerecht zu werden, wurde eine statische Filterklasse erstellt (siehe Anhang 9.9). Diese ist daher für die gesamte App gültig.

Kriterien

Im Filter gibt es für jedes Kriterium eine Property. Diese werden als nullbare Properties deklariert, wobei auf null gesetzte Werte als nicht gefiltert angesehen werden (zum Beispiel ÄlleÄuswahl bei Modulen). Die Kriterien nach denen gefiltert werden kann sind die folgenden:

- Name des Moduls
- Name des Studiengangs
- Datum
- Name des Erstprüfers
- Semester

FilterChangedListener

Damit der Filter einen dynamischen Einfluss auf die App hat, wurde ein FilterChangedListener hinzugefügt. Dieser ist eine Liste aus Funktionen, welche von überall aus der App hinzugefügt werden können. Diese Funktion besitzen weder Eingabe noch Rückgabe Parameter. Wenn sich ein Wert im Filter ändert, werden alle Funktionen in dieser Liste ausgeführt. Auf diese Weise kann zum Beispiel eine Liste sofort aktualisiert werden, sobald sich der Filter ändert. Aufgerufen wird dieser Listener aus angepassten Mutatormethoden der Properties. Ein Beispiel ist im Ausschnitt 1.3 zu sehen.

```
1 var modulName: String? = null
2     set(value) {
3         field = value
4         filterChanged()
5     }
```

Codeausschnitt 1.3: Beispiel einer Mutatormethode im Filter für den Modulnamen

Validierung

Um die Filterung an sich zu vereinfachen, wurden Validierungsmethoden implementiert, dessen Aufgabe es ist, für eine oder mehrere Prüfungen zu testen, ob diese dem Filter entsprechen. Diese Methoden sind im Anhang 8.1.1 in Abbildung 8.1 zu sehen.

Reset und Userfilter

Um den Filter wieder zurück zu setzen, gibt es eine Reset Methode, welche alle Properties auf ihren Normalzustand zurück setzt, also den Zustand, in dem keine

Prüfung aussortiert werden würde. Des weiteren gibt es in der Klasse **MainActivity** die Methode *userFilter*, welche den Filter auf einen dem Benutzer entsprechenden Normalzustand zurücksetzt. Dafür wird der Filter zurückgesetzt und der Studiengang wird auf den Hauptstudiengang des Benutzers gesetzt. Diese Methode wird aufgerufen wenn die **MainActivity** gestartet wird, wenn der Benutzer im Filter den Reset Button drückt (siehe Abschnitt 1.2.2) oder wenn das selbe Fragment, in dem sich der Benutzer befindet, erneut aufgerufen wird.

1.2.2 Dialog Fenster

Um den Benutzer mit dem Filter interagieren zu lassen, wurde ein Dialogfenster implementiert, welches über die Actionbar geöffnet werden kann. Das Fenster im ganzen ist in Abbildung 1.5 zu sehen. Im folgenden werden die einzelnen Auswahlmöglichkeiten kurz erläutert.

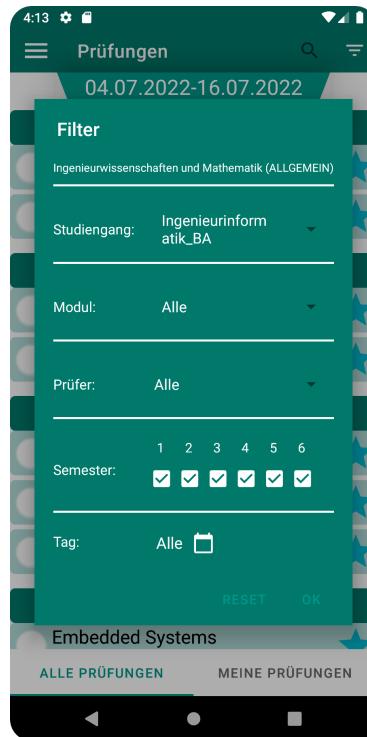


Abbildung 1.5: Dialogfenster des Filters

Modulname, Studiengangsname, Erstprüfer

Für die Kriterien Modulname, Studiengangsname und Erstprüfer gibt es in dem Dialog jeweils ein Dropdown Menü, in welchem dem Benutzer alle Möglichkeiten angezeigt werden können und aus denen er dann einen Wert auswählen kann.

Semester

Für die Semester Auswahl gibt es für jedes Semester (1-6) eine Checkbox, wo der Benutzer einstellen kann, aus welchen Semestern er die Prüfungen sehen möchte.

Kalender

Um einen bestimmten Tag auszuwählen, kann der Benutzer über ein Kalendericon ein weiteres Dialogfenster öffnen. Dieses zeigt einen Kalender, aus welchem der Benutzer einen Tag auswählen kann. Die Auswahl ist dabei auf die aktuelle Prüfungsphase beschränkt, Tage vorher oder nachher stehen nicht zur Auswahl. Zusätzlich stellt der Dialog Buttons zum Speichern oder Abbrechen der Auswahl, sowie der Auswahl aller möglichen Tage zur Verfügung. Das Dialogfenster ist in Abbildung 1.6 zu sehen.

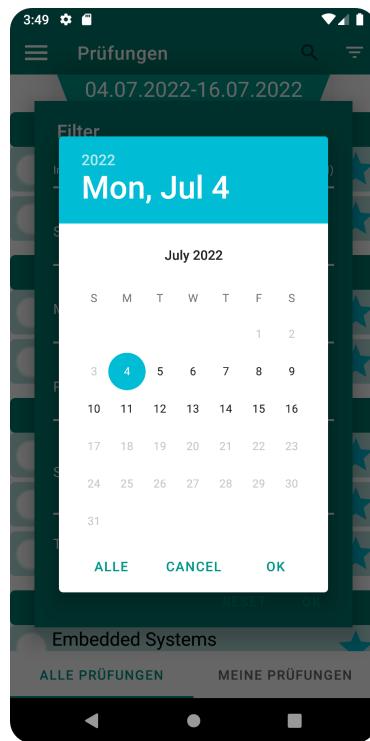


Abbildung 1.6: Kalendar zur Auswahl eines bestimmten Tages im Filter

Filter Schließen

Wenn der Benutzer seine Auswahl beendet hat, kann er entweder über einen OK Button den Dialog schließen oder über den Reset Button den *userFilter* aus Abschnitt 1.2.1 wiederherstellen.

1.2.3 Suchleiste

Zusätzlich zum Dialogfenster gibt es in der ActionBar noch eine Suchmöglichkeit, mit der der Benutzer eine Prüfung nach einem Modulnamen suchen kann. Dies ist aus jedem Fragment in der **MainActivity** möglich. Nach Bestätigung der Suche wird jedes mal automatisch das Fragment **ExamOverviewFragment** aufgerufen, wobei der Filter auf den eingegebenen Modulnamen gesetzt wurde. Um diese Suche zu erleichtern, wurde der Suchleiste eine Autovervollständigung übergeben, die nach der Eingabe von zwei oder mehr Buchstaben alle Module anzeigen, die der Eingabe entsprechen könnten. Dies ist in Abbildung 1.7 gezeigt.

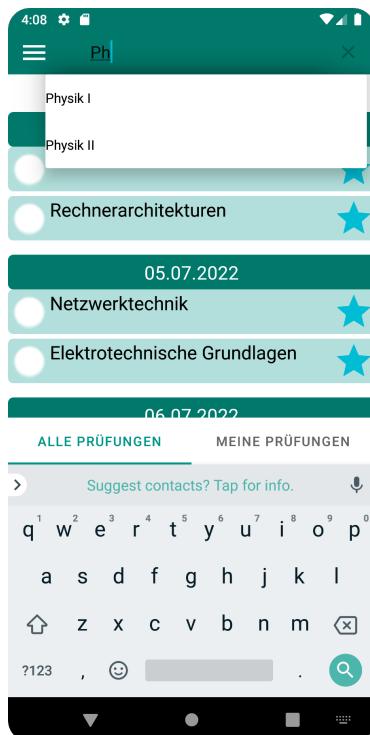


Abbildung 1.7: Suchleiste mit Autovervollständigung

1.3 Navigation

Eine weitere Verbesserung wurde an der Navigation vorgenommen. Zuvor gab es einen Navigationdrawer, durch den der Benutzer zu jeder Seite wechseln konnte, sowie eine Bottomnavigation, durch die der Benutzer zwischen den gesamten Prüfungen, der Suche, den Favoriten und der Wahlmodulssuche wechseln konnte. Abbildungen 1.8 und 1.9 zeigen den Ausgangspunkt, an dem gearbeitet wurde.

Diese Navigation war allerdings sehr undurchsichtig und aufwändig und hatte wenig Struktur. Eine erste Verbesserung wurde durch das verschieben des Suchfensters und der Wahlmodulsuche in den Filter, welcher über die ActionBar erreichbar ist, erzielt. Dadurch wurde die Bottom Navigation bereits einfacher. Der Plan war es, ein Fenster für die Prüfungsansichten zu erstellen, in welchem

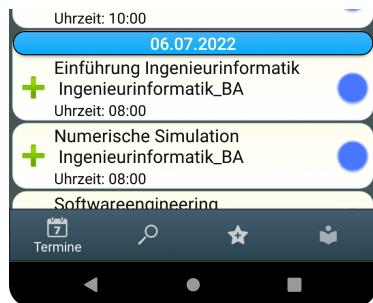


Abbildung 1.8: Aussehen der alten Bottom Navigation

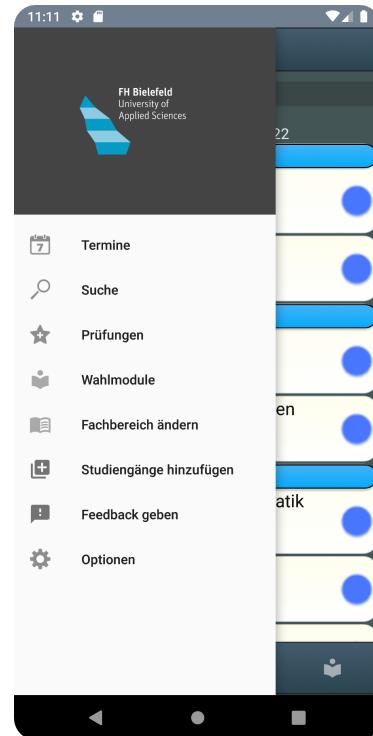


Abbildung 1.9: Aussehen des alten Navigation Drawers

der Benutzer durch die Bottom Navigation zwischen allen Prüfungen und seinen Favoriten hin und her wechseln und alle weiteren Seiten sollen über den Navigation Drawer erreichbar sein. Die Bottom Navigation bleibt dabei allerdings dauerhaft aktiv, um eine schnellere Rückkehr zur Prüfungsübersicht zu ermöglichen. Abbildungen 1.10 und 1.11 zeigen die Navigation nach der Veränderung.

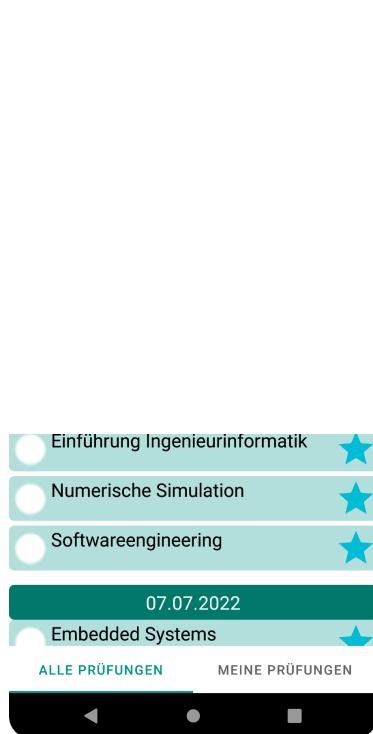


Abbildung 1.10: Aussehen der neuen Bottom Navigation



Abbildung 1.11: Aussehen des neuen Navigation Drawers

1.3.1 Neuimplementierung des Fragmentwechsels

Ein weiteres Problem bei der Navigation war die Implementierung für das wechseln zwischen den einzelnen Fragmenten. Die Umsetzung erfolgte zuvor über große Switch-Case Strukturen, in denen viel wiederholdender Code verwendet wurde. Da jeder Fragmentwechsel mit ein paar kleinen Ausnahmen identisch ist, wurde dafür eine Verallgemeinerung vorgenommen. Der Codeausschnitt 1.4 zeigt einen Ausschnitt aus der alten Umsetzung.

```

1 case R.id.navigation_diary:
2         header.setTitle(getApplicationContext()
3             .getString(R.string.title_exam));
4         recyclerView.setVisibility(View.
5             INVISIBLE);
6         calendar.setVisibility(View.GONE);
7         btnSearch.setVisibility(View.GONE);
8         dl.closeDrawer(GravityCompat.START);
9         ft.replace(R.id.frame_placeholder, new
10             Favoritenfragment());
11        ft.commit();
12
13        return true;
14
15 case R.id.navigation_settings:
16         header.setTitle(getApplicationContext()
17             .getString(R.string.title_settings)
18             );
19         recyclerView.setVisibility(View.
20             INVISIBLE);
21         calendar.setVisibility(View.GONE);
22         btnSearch.setVisibility(View.GONE);
23         dl.closeDrawer(GravityCompat.START);
24         ft.replace(R.id.frame_placeholder, new
25             Optionen());
26         ft.commit();
27
28        return true;

```

Codeausschnitt 1.4: Ausschnitt der Switch-Case Verzweigung der alten Navigation

Dabei ist die Wiederholung von bestimmten Codezeilen gut zu erkennen. Die einzigen Unterschiede dabei sind der Titel und der Aufruf des neuen Fragments in den Zeilen 2,13,7 und 18. Die Verallgemeinerung des Fragmentwechsels wurde in zwei Schritten durchgeführt. Um den Titel auszulagern, wurde für jedes Fragment in der **MainActivity** eine Elternklasse hinzugefügt, welche als abstrakte Methode den Namen des Fragmentes zurück liefern muss. Diese Klasse ist im Ausschnitt 1.5 zu sehen.

```

1 abstract class MainActivityFragment: Fragment() {
2     abstract fun getName(context: Context): String
3 }

```

Codeausschnitt 1.5: Elternklasse für alle Fragment innerhalb der MainActivity

Um zu einem neuen Fragment wechseln zu können, wurde in der **MainActivity** eine neue Funktion *changeFragment* implementiert. Diese bekommt als Eingabeparameter das Fragment, zu dem gewechselt werden soll und kümmert sich anschließen um den entsprechenden Wechsel. In Abbildung 1.6 ist diese Funktion zu sehen. Die Umsetzung des ViewPages aus Kapitel 1.3.3 sorgt zwar für ein wenig mehr Komplexität, die Funktion ist allerdings gut erweiterbar und einfach zu verwenden.

```

1 fun changeFragment(fragment: MainActivityFragment): Boolean {
2     activity_main_toolbar.title = fragment.getName(this)
3     drawer_layout.closeDrawer(GravityCompat.START)
4     activity_main_textview_current_period_timestamp.
5         visibility = View.VISIBLE
6     when {
7         fragment :: class == ExamOverviewFragment :: class -> {
8             activity_main_placeholder.visibility = View.
9                 INVISIBLE
10            activity_main_viewpager.visibility = View.
11                VISIBLE
12            activity_main_viewpager.setCurrentItem(0, true)
13            activity_main_viewpager.invalidate()
14        }
15        fragment :: class == FavoriteOverviewFragment :: class -> {
16            activity_main_placeholder.visibility = View.
17                INVISIBLE
18            activity_main_viewpager.visibility = View.
19                VISIBLE
20            activity_main_viewpager.setCurrentItem(1, true)
21            activity_main_viewpager.invalidate()
22        }
23        else -> {
24            val ft = supportFragmentManager.
25                beginTransaction()
26            activity_main_viewpager.visibility = View.
27                INVISIBLE
28            activity_main_textview_current_period_timestamp
29                .visibility = View.GONE
30            activity_main_placeholder.visibility = View.
31                VISIBLE
32            ft.replace(R.id.activity_main_placeholder,
33                fragment)
34            ft.commit()
35        }
36    }
37    return true
38 }

```

Codeausschnitt 1.6: Methode zum wechseln zu einem anderen Fragment

1.3.2 TabLayout

Um für noch weiteren Komfort bei der Nutzung der App zu sorgen, wurde es dem Benutzer ermöglicht, zwischen der Übersicht aller Prüfungen und seinen Favoriten durch ein Wischen des Bildschirmes hin und her zu wechseln. Dafür wurde die Bottom Navigation durch ein TabLayout ersetzt. Ein TabLayout verwaltet mehrere Fragmente in Form von Tabs, zwischen denen über eine Navigation ähnlich der Bottom Navigation hin und her gewechselt werden kann. Das TabLayout ist aus mehreren Gründen eine gute Alternative zur Bottom Navigation. Ein Vorteil der hier verwendet wird ist die Tatsache, dass das TabLayout erkennt, wenn ein Tab erneut ausgewählt wurde (Reselected). Dies findet Anwendung beim UserFilter aus Kapitel 1.2.1. Der Hauptgrund für die Verwendung eines TabLayouts ist allerdings, dass in diesen ein ViewPager eingebaut werden kann, der es ermöglicht, durch ein Wischen über die Seite zwischen den einzelnen Tabs zu wechseln. Der ViewPager ist in Kapitel 1.3.3 näher erläutert. Bevor ein Ta-

bLayout verwendet werden kann, muss dieses zu aller erst in das Layout der **MainActivity** eingebaut werden. Dies ist im Ausschnitt 1.7 gezeigt.

```
1      <com.google.android.material.tabs.TabLayout  
2          android:id="@+id/activity_main_tab_layout"  
3              android:layout_width="match_parent"  
4                  android:layout_height="50dp"  
5                      android:layout_alignParentBottom="true"  
6                  >  
7      </com.google.android.material.tabs.TabLayout>
```

Codeausschnitt 1.7: Einbettung des TabLayouts in das Layout der MainActivity

Anschließend kann das TabLayout in der entsprechenden Activity Klasse initialisiert werden. Dies ist in Abbildung 1.8 zu sehen. Zuvor muss allerdings der ViewPager initialisiert worden sein. Verwendet wird dabei der sogenannte *TabLayoutMediator*, welcher sich um die Initialisierung des TabLayouts, inklusive des ViewPagers, kümmert. Der Block nach dem Funktionsaufruf beschreibt die Strategie, wie des TabLayout auf bestimmte Sachen reagieren soll. In Zeile 5 - 8 in Abbildung 1.8 werden die Namen für die einzelnen Tabs, die in der Navigationsleiste angezeigt werden, festgelegt. Anschließend wird dem TabLayout noch ein Listener zugefügt, welcher bestimmt was passiert, wenn ein Tab gewählt, wiedergewählt oder abgewählt wurde. Dabei werden die einzelnen Tabs durch eine Position voneinander unterschieden.

```

1 private fun initTabLayout(){
2     TabLayoutMediator(activity_main_tab_layout ,
3         activity_main_viewpager
4     ) {
5         tab, position ->
6             tab.text = when(position){
7                 1->resources.getString(R.string.tab_layout_exam)
8                 else->resources.getString(R.string.
9                     tab_layout_exam_overview)
10            }
11            activity_main_tab_layout.addTabSelectedListener(
12                object:TabLayout.OnTabSelectedListener{
13                    override fun onTabReselected(tab: TabLayout.Tab
14                        ?) {
15                        userFilter()
16                        when(tab?.position){
17                            1->changeFragment(
18                                FavoriteOverviewFragment())
19                            else->{
20                                changeFragment(ExamOverviewFragment
21                                    ())
22                            }
23                        }
24                    }
25                    override fun onTabSelected(tab: TabLayout.Tab?)
26                        {
27                            when(tab?.position){
28                                1->changeFragment(
29                                    FavoriteOverviewFragment())
30                                else->changeFragment(
31                                    ExamOverviewFragment())
32                            }
33                        }
34                    }
35                }.attach()
36            }
37        }
38    }
39}

```

Codeausschnitt 1.8: Methode zum wechseln zu einem anderen Fragment

1.3.3 ViewPager

Damit durch wischen zwischen den Tabs gewechselt werden kann, muss dem TabLayout ein ViewPager übergeben werden. Ein ViewPager muss ebenfalls in das Layout mit eingebaut werden, wie es im Ausschnitt 1.9 gezeigt ist.

```

1      <androidx.viewpager2.widget.ViewPager2
2          android:id="@+id/activity_main_viewpager"
3          android:layout_width="match_parent"
4          android:layout_height="match_parent"
5          android:layout_below="@+id/activity_main_toolbar"
6          android:layout_above="@+id/activity_main_tab_layout"
7      />

```

Codeausschnitt 1.9: Einbettung des ViewPagers in das Layout der MainActivity

Anschließend muss dem ViewPager ein FragmentStateAdapter übergeben werden. Dieser muss mindestens zwei Methode überschreiben. Eine die die Anzahl an Tabs zurückgibt und eine die aus einer Position ein Fragment erzeugt. Die Implementierung ist im Ausschnitt 1.10 zu sehen.

```

1 class MainFragmentPagerAdapter(fragmentActivity : FragmentActivity) : FragmentStateAdapter(fragmentActivity) {
2     override fun getItemCount(): Int {
3         return 2
4     }
5     override fun createFragment(position: Int): Fragment {
6         return when(position){
7             1-> FavoriteOverviewFragment()
8             else-> ExamOverviewFragment()
9         }
10    }
11 }

```

Codeausschnitt 1.10: Methode zum wechseln zu einem anderen Fragment

Anschließend kann der ViewPager in das TabLayout mit eingebaut werden, wies es die Abbildung 1.8 zeigt. Damit der ViewPager allerdings einwandfrei verwendet werden kann, musste das *SwipeToDelete* Feature wieder entfernt werden. Dieses hatte es dem Benutzer ermöglicht, durch ein wischen über einen Prüfplaneintrag diesen zu favorisieren oder zu entfavorisieren. Allerdings wurde dadurch die Wischgeste für den ViewPager blockiert, da stattdessen über den Prüfplaneintrag gewischt wurde.

1.4 Layouts

Neben der Navigation wurden auch die einzelnen Fragmente an sich überarbeitet. Dabei wurde wert auf Übersichtlichkeit und Design gelegt.

1.4.1 Alle Prüfungen

Die Liste, welche alle Prüfungen anzeigt, wurde mehr oder weniger gelassen wie sie war. Allerdings wurde der Kalender oberhalb der Liste, wessen Zweck durch den neuen Filter ersetzt wurde, entfernt. Zudem wurde der Prüfungszeitraum in die MainActivity verschoben, sodass er sowohl für alle Prüfungen als auch für die Favoriten sichtbar ist und wurde gut sichtbar unterhalb der ActionBar

platziert. Die Abbildungen 1.12 und 1.13 zeigen einen vorher/nachher Vergleich des Fragmentes für alle Prüfungen.



Abbildung 1.12: Aussehen des Fragments für Alle Prüfungen vor der Veränderung



Abbildung 1.13: Aussehen des Fragments für Alle Prüfungen nach der Veränderung

Ebenfalls zu erkennen ist die Umpositionierung des Favorisiericons. Dieses wurde auf die rechte Seite verschoben, da der größte Teil der Bevölkerung Rechtshänder sind und für diese daher das Favorisieren von Prüfungen erleichtert wird, vor allem wenn die App mit nur einer Hand bedient wird.

1.4.2 Favoriten

Ebenfalls eine Veränderung erhalten hat die Übersicht über die Favoriten. Diese werden in der neuen Version in einem CardView angezeigt, welches als Designelement einfacher und schöner aussieht.

Card View

Eine *Card* ist eine gute Möglichkeit, um gebündelte Daten schön und Strukturiert darzustellen. Dabei sollte jede *Card* unabhängig und nicht mit anderen *Cards* verbunden, oder in mehrere kleinere *Cards* aufgeteilt werden können[4]. Der Unterschied zwischen der alten und der neuen Darstellung ist in den Abbildungen 1.14 und 1.15 gezeigt.

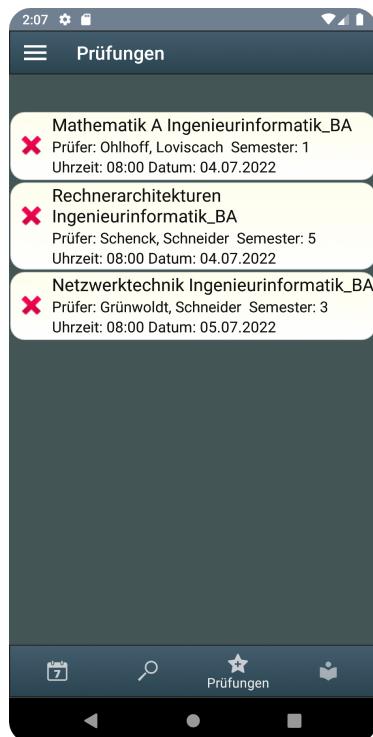


Abbildung 1.14: Aussehen der Favoriten vor der Veränderung



Abbildung 1.15: Aussehen der Favoriten nach der Veränderung

Neben den Informationen über die Prüfung, besitzt jede *Card* zusätzlich ein Icon in der oberen linken Ecke, welches Auskunft gibt ob die Prüfung in den Kalender eingetragen wurde oder nicht.

Kontext Menü

Damit mit den Favoriten interagiert werden kann, wurde zudem ein Kontextmenü hinzugefügt, welches geöffnet werden kann, wenn der Benutzer länger auf eine Prüfung drückt. Dieses Menü gibt dem Benutzer die Möglichkeit, die Prüfung aus den Favoriten zu entfernen und sie in den Kalender einzufügen oder zu entfernen, je nach dem ob sie bereits im Kalender vorhanden ist oder nicht. Das Menü ist in Abbildung 1.16 zu sehen.

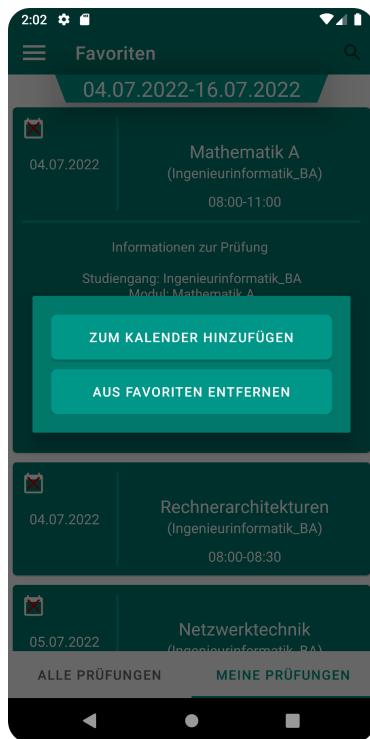


Abbildung 1.16: Kontextmenü eines Favoriten

1.4.3 Einstellungen

Eine weitere große Veränderung wurde an den Einstellungen vorgenommen. Die Abbildungen 1.17 und 1.18 stellen den Unterschied zur früheren Version dar.



Abbildung 1.17: Aussehen der Einstellungen vor der Veränderung

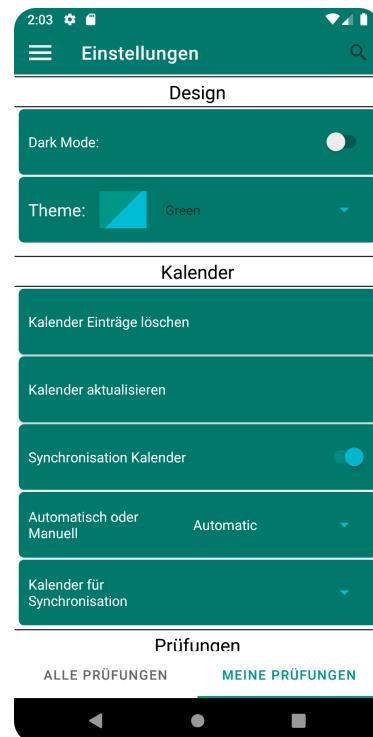


Abbildung 1.18: Aussehen der Einstellungen nach der Veränderung

Die Einstellungen wurden in verschiedene Gruppen aufgeteilt und voneinander getrennt. Außerdem sind noch weitere Einstellungen hinzugekommen, wie unter anderem für das Thema und den Darkmode (siehe Kapitel 1.1 und 1.1.3). Und für den Fall, dass eine Einstellung einen Neustart der App erfordert, wird dies über eine Snackbar am unteren Bildschirmrand angezeigt. Die Snackbar ist in Abbildung 1.19 gezeigt. Sie bietet außerdem die Möglichkeit über den "NeustartenButton" die App automatisch neu starten zu lassen.

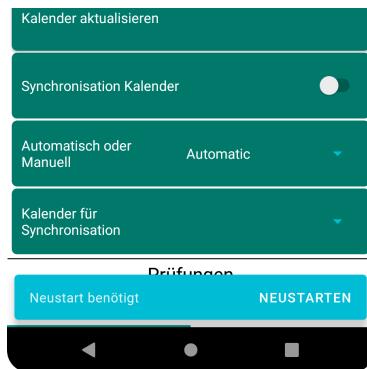


Abbildung 1.19: Snackbar in den Einstellung zur Anzeige, dass ein Neustart benötigt wird

Die Implementierung der Snackbar ist im Ausschnitt 1.11 zu sehen.

```

1      .setAction(requireContext().resources.getString
2          (R.string.settings_snackbar_restart)){
3      val pid = Process.myPid()
4      Process.killProcess(pid)
5      dirty = false
6  }
7      .setBackgroundTint(Utils.getColorFromAttr(R.
8          attr.colorAccent, requireContext()))
9      .setTextColor(Utils.getColorFromAttr(R.attr.
10         colorOnAccent, requireContext()))
11     .setActionTextColor(Utils.getColorFromAttr(R.
12         attr.colorOnAccent, requireContext()))
13     .show()

```

Codeausschnitt 1.11: Implementierung der Snackbar zum Neustarten lassen der App

Zu guter Letzt wurde die Datenschutzerklärung und das Impressum in die Einstellungen mit eingebaut. Diese werden in einem eigenen Fragment als Fließtext angezeigt. Da die Texte in einer eigenen Textdatei vorliegen, wurde eine Methode in der Klasse **Utils** implementiert, die diese Textdateien ausliest und als String zurück gibt. Diese Implementierung ist im Ausschnitt 1.12 gezeigt.

```

1 fun readTextFile(context: Context, @RawRes textResource:Int):  

2     String {  

3         val inputStream = context.resources.openRawResource(  

4             textResource)  

5         val reader = BufferedReader(InputStreamReader(  

6             inputStream))  

7         var string: String?  

8         val stringBuilder: StringBuilder = StringBuilder()  

9         while (true) {  

10             string = reader.readLine()  

11             try {  

12                 if (string==null) break  

13             } catch (ex:Exception){  

14                 Log.e("Utils",ex.stackTraceToString())  

15             }  

16             stringBuilder.append(string).append("\n")  

17         }  

18         inputStream.close()  

19         return stringBuilder.toString()  

20     }

```

Codeausschnitt 1.12: Methode zum einlesen einer Textdatei

1.4.4 Studiengänge Verwalten

Das Fragment zum ändern der gewählten Studiengänge wurde ebenfalls angepasst. Ursprünglich war es nur möglich, Studiengänge die nicht dem Hauptstudiengang entsprachen, hinzu zu wählen oder ab zu wählen. Der Hauptstudiengang konnte nur beim Ändern der Fakultät mit verändert werden. In der neuen Version ist es jetzt möglich, den Hauptstudiengang dort zu ändern, wo die Studiengänge hinzugefügt oder entfernt werden konnten. Der Vergleich ist in den Abbildungen 1.20 und 1.21 zu sehen.

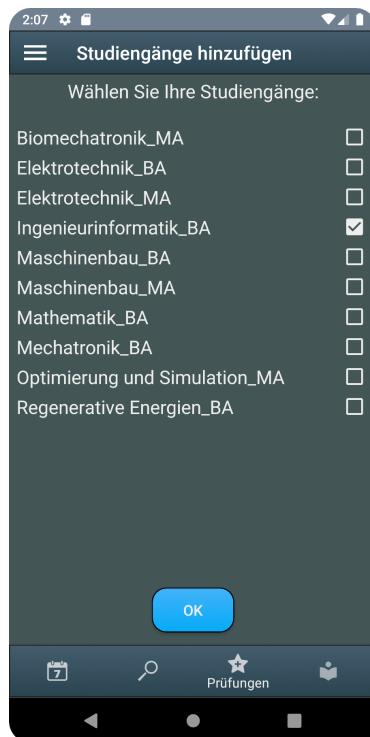


Abbildung 1.20: Aussehen der Studiengangverwaltung vor der Veränderung

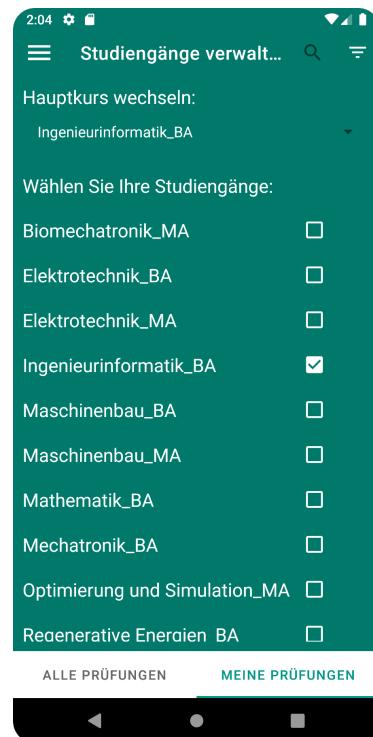


Abbildung 1.21: Aussehen der Studiengangverwaltung nach der Veränderung

Beim ändern des Hauptstudienganges wird außerdem darauf geachtet, den gewählten Hauptstudiengang, falls noch nicht geschehen, zu den gewählten Studiengängen hinzuzufügen.

Kapitel 2

Model-View-Viewmodel-Pattern

Neben den Veränderungen im Frontend wurden auch am Backend einige Änderungen vorgenommen. Ein großes Problem war, dass fast der gesamte Code in den Activities und Fragmenten ausgeführt wurde. Dies sorgt allerdings für eine große Unordnung und macht es zudem sehr schwer, weitere Änderungen an der App vorzunehmen. Um diesem Problem entgegen zu wirken, wurde das sogenannte Model-View-Viewmodel (MVVM) Pattern umgesetzt. Durch dieses wird der Code in drei Bereiche aufgeteilt.

Der erste Teil des Patterns ist das Modell. In diesem werden alle Zugriffe auf die Datenbanken und weitere externe Ressourcen durchgeführt. Dies schließt die Room Datenbank, das Rest Interface und dies Shared Preferences mit ein.

Der zweite Teil ist die View. In dieser findet die Verwaltung der grafischen Elemente der App statt. Dazu zählen Benutzerinteraktionen wie die Navigation oder Buttons aber auch das Anzeigen von Informationen und Dialogen.

Der letzte Teil des Patterns ist das ViewModel. Dieses bildet die Schnittstelle zwischen dem Modell und der View, beinhaltet allerdings auch jegliche weitere Programmlogik, die unabhängig von der View oder dem Modell ist wie zum Beispiel die Kalender Schnittstelle.

2.1 Modell

Wie bereits erwähnt dient das Modell dazu, den Zugriff zu den externen Datenspeichern wie zum Beispiel den Datenbanken herzustellen. Das Ziel ist es dabei, den Zugriff auf eine Art und Weise zu Vereinfachen, sodass eine Veränderung in der Datenbank keine Änderung an der View oder dem ViewModel nötig hat.

2.1.1 Room

Im Zusammenhang mit der App werden zwei Datenbanken verwendet. Die erste ist die Room Datenbank. Diese ist eine lokale Datenbank nach dem SQLite

Standard, dessen Zweck es ist, ein Abbild der externen Datenbank auf dem Smartphone des Benutzers zu speichern, um eine Benutzung der App ohne eine bestehende Internetverbindung zu ermöglichen.

Entitäten

Die bereits existierenden Entitäten wurden nicht weiter verändert, es wurde allerdings die Fakultät als neue Entität hinzugefügt. Die Fakultäten wurden zuvor in den Shared Preferences abgelegt, was zu komplizierten Abfragen geführt hatte.

Die **UserDao** Klasse wurde ebenfalls angepasst. Es wurden unnötige Abfragen entfernt und neu benötigte hinzugefügt. Zusätzlich wurde bei den *Insert* Queries eine Konflikt Strategie definiert, um Probleme mit bereits in der Datenbank vorhandenen Daten zu vermeiden. Probleme wären dabei zum Beispiel, dass zum Beispiel Benutzerauswahlen wie eine favorisierte Prüfung oder ein favorisierter Studiengang verloren gehen würden, sollten die Datensätze ersetzt werden. Es soll ebenfalls verhindert werden, dass Daten mehrfach in der Datenbank vorkommen. Daher wurde die Konflikt Strategie sofern eingestellt, dass Datensätze von bereits vorhandenen Daten ignoriert werden. Damit diese Datensätze trotzdem ein Update bekommen können wurde im **BaseViewModel** aus Kapitel 2.2 eine entsprechende Methode *updateEntry* implementiert, welche eine Prüfung in der Datenbank ersetzt und gleichzeitig den Favoritenstatus erhält. Diese Methode ist im Anhang 8.4.1 zu finden.

LiveData

Eine weitere Neuerung in der Room Database sind die sogenannten LiveData Objekte. Bei einem LiveData Objekt handelt es sich um eine observierbare Klasse zum speichern von Daten. Das heißt es kann ein Observer übergeben werden, der von der Klasse benachrichtigt wird, sobald sich die Daten innerhalb der Klasse ändern. Dies ist in sofern nützlich als das auf diese Weise die App sofort mitbekommt, sollten an den Daten relevante Änderungen geschehen sein. Die Abbildungen 2.1 und 2.2 zeigen den Unterschied zwischen einem Query mit und ohne die Verwendung von LiveData. Auffällig ist dabei das fehlende Schlüsselwort *suspend*. Dieses sorgt in Kotlin dafür, dass eine Methode zu einer asynchronen Methode wird, welche nur innerhalb einer anderen Suspend Methode oder einer Coroutine ausgeführt werden kann (mehr dazu im Anhang 9.22). Der Vorteil ist also, das auf die LiveData Objekt zugegriffen werden kann ohne zuvor eine Coroutine zu starten, was eine wichtige Eigenschaft für die Nutzung in der View ist. Dort werden nämlich keine Coroutines verwendet. Dies ist in Kapitel 2.3.2 näher erläutert.

```

1 @Query("SELECT * FROM TestPlanEntry as t INNER JOIN Course as c
        ON c.courseName LIKE t.course WHERE c.chosen = 1 ORDER BY
        date, termin, module")
2     fun getAllEntriesForChosenCoursesByDateLiveData(): LiveData
        <List<TestPlanEntry>>

```

Codeausschnitt 2.1: SQL Abfrage mit Hilfe eines LiveData Objektes

```

1 @Query("SELECT * FROM TestPlanEntry ORDER BY date, termin,
        module")
2     suspend fun getAllEntries(): List<TestPlanEntry>?

```

Codeausschnitt 2.2: SQL Abfrage ohne Hilfe eines LiveData Objektes

2.1.2 Retrofit2

Eine weitere verwendete Datenbank ist die über ein Rest Interface erreichbare Datenbank auf dem Server der Fachhochschule Bielefeld. Für den Zugriff wurde zuvor die API **Retrofit** verwendet. Diese wurde nun durch die neue **Retrofit2** API ersetzt, die den Aufwand für den Zugriff stark reduziert.

Interface

Ähnlich wie bei der Room Database, erfolgt der Zugriff mit Retrofit2 über ein Interface, in welchem die Queries definiert sind. Im Ausschnitt 2.3 ist eine solche Query zu sehen. Das Schlüsselwort *GET* signalisiert dabei, dass es sich um eine Selektion und nicht um eine Manipulation handelt. Nach den *GET* wird in den Klammern die URL zu den Daten angegeben. Dabei können über die geschweiften Klammern Eingabeparameter eingebaut werden. Die festen URLs die für die Zugriffe verwendet werden sind als Konstanten in der Datei **URLs.kt** abgelegt. Über das Dollarzeichen kann auf diese Konstanten zugegriffen werden.

```

1 @GET("$entriesRelativeUrl/{ppSemester}/{pTermin}/{pYear}/{pIds
    }/")
2     suspend fun getEntries(
3         @Path("ppSemester") ppSemester: String,
4         @Path("pTermin") pTermin: String,
5         @Path("pYear") pYear: String,
6         @Path("pIds") pIds: String): List<GSOneEntry>

```

Codeausschnitt 2.3: Datenbank Query mit Retrofit2

Json

Wie in Abbildung 2.3 zu sehen, besteht die Rückgabe einer Query aus einer Liste aus GSON Objekten. Dies sind eigens definierte Data Klassen, welche die empfangenen Daten von der Restschnittstelle abspeichern. Die Attributnamen sollten dabei mit denen der Restschnittstelle übereinstimmen. Die Data Klassen sind in den Anhängen 8.3.1, 8.3.1 und 8.3.1 zu finden.

RetrofitHelper

Um das Retrofit Interface zu verwenden, muss zunächst ein Retrofit Objekt erstellt werden. Hierfür ist die Klasse **RetrofitHelper** zuständig. Diese hat eine Methode um das Retrofitobjekt zu erzeugen, zu initialisieren und zurückzugeben. Die Methode dafür ist im Ausschnitt 2.4 zu sehen.

```
1 fun getInstance(): Retrofit {
2     val gson = GsonBuilder().setLenient().create()
3     return Retrofit.Builder()
4         .baseUrl(serverResourcesUrl).addConverterFactory(
5             GsonConverterFactory.create(gson)).build()
}
```

Codeausschnitt 2.4: Methode zur Deklarierung und Initialisierung eines Retrofit Objektes

Dabei wird als *baseUrl* eine URL übergeben, die für alle Zugriffe identisch ist. Zusätzlich muss auch noch ein entsprechender Konverter übergeben werden, welcher die empfangenen Daten in die entsprechenden Klassen konvertiert. Da die Daten als JSON Format empfangen werden, wird der GSON-Konverter verwendet. Dieser wird mit der Methode *setLenient* zuvor so initialisiert, dass er verschiedene Formatierungen akzeptiert. Nach der Erstellung des Retrofit Objektes, kann der Zugriff auf die Restschnittstelle mit Hilfe des Retrofit Interfaces eingerichtet werden. Dies passiert allerdings im Repository aus Kapitel 2.1.3.

2.1.3 Repository

Um die Datenbankzugriffe von den ViewModels und den Views klar zu trennen, wird ein Repository implementiert, welches sich um jegliche Zugriffe auf die Datenbanken kümmert. Auf diese Weise greifen die ViewModels und die Views nicht direkt über die entsprechenden Interfaces, sondern nur über das entsprechende Repository auf die Datenbanken zu. Wenn also Änderungen an den Interfaces vorgenommen werden müssen, muss also nur das Repository angepasst werden. Dies sorgt für eine gute Wartbarkeit und Skalierbarkeit.

Damit das Repository auf die Daten Zugreifen kann, müssen zuerst die Interfaces eingerichtet werden. In Ausschnitt 2.5 ist dieser Prozess zu sehen.

```
1 private var localDataSource: UserDao = AppDatabase.
2     getAppDatabase(context).userDao()
3 private var remoteDataSource =
4     RetrofitHelper.getInstance().create(RetrofitInterface::
5         class.java)
```

Codeausschnitt 2.5: Einrichten des UserDao Interfaces und des Retrofit Interfaces

Anschließend können über die Interfaces die Abfragen definiert werden. In der Regel werden die Methoden aus den Interfaces einfach nochmal aufgerufen. Dies passiert allerdings in einem neuen CoroutineContext mit Hilfe von *withContext*. Bei der Funktion handelt es sich danach zwar immer noch um eine Suspend Funktion, dank der Nutzung von *withContext* können die Funktionen danach allerdings Seriell innerhalb einer anderen Coroutine ausgeführt werden. Dies ist von Vorteil wenn in einer Funktion mehrere dieser Funktionen voneinander

Abhängig sind. Ohne die Nutzung von `withContext` würden alle Funktionen parallel zu einander ablaufen und es könnte keine serielle Kette gebildet werden wo eine Funktion mit den Daten einer anderen Funktion weiter arbeitet [geeksforgeekswithcontext2020]. `WithContext` wird zusammen mit dem IO Dispatcher verwendet, welcher optimiert ist für Netz- und Laufwerklastige Aufgaben (siehe im Anhang 9.22.2).

Neben den einfachen neu aufgerufenen Methoden gibt es außerdem auch noch Methoden, die nicht durch eines der Interface umgesetzt werden konnten, wie es zum Beispiel bei der Methode `fetchFaculties` der Fall ist. In diesem Fall sind diese Methoden dann ebenfalls im Repository implementiert.

2.1.4 SharedPreferences Repository

Ein weiteres Repository im Modell ist das `SharedPreferencesRepository`. Dieses, wie der Name schon vermuten lässt, kümmert sich um jede Interaktion mit den SharedPreferences. Dies hat zum einen den Vorteil, dass die vielen Zugriffe der früheren Appversion verallgemeinert werden können und somit deutlich an Redundanz gespart wird. Zum anderen wird die Fehleranfälligkeit bei der Nutzung stark reduziert, die durch die Nutzung von String Indikatoren für die einzelnen Daten entstehen können. Da auf Daten in den SharedPreferences über eine eigens definierte Id angesprochen werden, könnte es schnell zu Fehlern kommen, wenn bei der Id ein Tippfehler passiert oder eine falsche Id verwendet wird. Diesem Problem wird mit dem neuen Repository entgegengewirkt. Es gibt vier verschiedene Dateien, in denen bestimmte Daten abgelegt sind.

- `periodInformation` - Informationen zur aktuellen Prüfperiode
- `settings` - App Einstellungen des Benutzers
- `userSelection` - Auswahlen des Benutzers, zum Beispiel Hauptstudiengang oder Fakultät
- `calendarEntries` - Im Kalender enthaltene Prüfplaneinträge kombiniert mit der entsprechenden Event Id

Da es sich bei den Zugriffen auf die Shared Preferences nicht um asynchrone Zugriffe handelt, müssen die entsprechenden Methoden nicht als Suspend Funktionen definiert werden.

2.2 ViewModel

Damit die View mit dem Modell kommunizieren kann, wird eine weitere Schnittstelle benötigt. Das ViewModel erfüllt diesen Job, da es die Kommunikation mit dem Modell auf eine Weise vereinfacht, dass die View darauf zugreifen kann. Der Vorteil eines ViewModel ist, dass die Daten erhalten bleiben, auch wenn die UI neu aufgebaut wird, wie es zum Beispiel beim Wechsel vom Portrait in den Landscape Modus der Fall ist.

2.2.1 Coroutines

Ein großes Problem welches das ViewModel lösen muss ist, dass der Zugriff auf das Modell in einigen Fällen nur über eine Coroutine möglich ist.

Scope

Damit eine Coroutine ausgeführt werden kann, wird ein CoroutineScope benötigt (siehe Anhang 9.22.2). Vom ViewModel gibt es dafür bereits eine Lösung, und zwar das *ViewModelScope*. Dieses Scope ist an das ViewModel gebunden, und wird daher abgebrochen, wenn das ViewModel beendet wird. Dies ist hilfreich um unnötige Ressourcen zu sparen, die durch nicht benötigte Coroutines entstehen könnten. Ein Beispiel für die Nutzung des ViewModelScopes ist im Ausschnitt 2.6 zu sehen.

```
1 fun insertCourses(courses: List<Course>) {
2     viewModelScope.launch(coroutineExceptionHandler) {
3         repository.insertCourses(courses)
4     }
5 }
```

Codeausschnitt 2.6: Verwendung des ViewModelScopes

Exception Handler

Sollte während einer Coroutine eine Exception geworfen werden, so muss diese natürlich abgefangen werden um die App vor einem Absturz zu bewahren. Zu diesem Zweck wird jeder Coroutine ein *coroutineExceptionHandler* mit gegeben, welcher diese Aufgabe erfüllt. Der Exception Handler wird jedes mal aufgerufen sobald eine Exception in der Coroutine gefangen wurde und ist in dieser Version der App so definiert, dass er nur die Fehlermeldung ausgibt.

Mutable LiveData

Da die Berechnungen innerhalb der Coroutines asynchron zum Rest der App ausgeführt werden, können diese Funktionen keine Werte direkt zurückgeben. Dies ist allerdings ein Problem, wenn die View zum Beispiel auf eine Liste von Daten aus der Datenbank zugreifen möchte. Eine Möglichkeit dabei ist die Nutzung der LiveData Querys aus der Room Datenbank (siehe Kapitel 2.1.1). Da allerdings nicht jede Aufgabe auf diese Weise erledigt werden kann, zum Beispiel wenn Daten noch ausgewertet werden müssten, muss auf eine Alternative zurückgegriffen werden. In diesem Fall werden die **MutableLiveData** Objekte verwendet. Bei diesen handelt es sich ebenfalls um LiveData Objekte, denen allerdings zur Laufzeit neue Werte gegeben werden können. Dies ist bei den normalen LiveData Objekten nicht der Fall. Ausschnitt 2.7 zeigt ein Beispiel für die Initialisierung und Verwendung eines MutableLiveData Objektes. In dem Beispiel sollen alle Studiengänge zu der ausgewählten Fakultät gefunden werden. Da hierzu allerdings zuerst die gewählte Fakultät benötigt wird und dies nicht in die UserDao der Room Database eingebaut werden kann, muss dafür eine MutableLiveData Liste verwendet werden. Die Werte werden dabei über die Methode *postValue* an das MutableLiveData Objekt übergeben.

```
1 val liveCoursesForFaculty = MutableLiveData<List<Course>?>()
2 fun getCourses() {
3     viewModelScope.launch {
4         val courses = getSelectedFacultyId()?.let {
5             getCoursesByFacultyId(it)
6         }
7         liveCoursesForFaculty.postValue(courses)
8     }
9 }
```

```
    7 }
```

Codeausschnitt 2.7: Initialisierung und Verwendung eines MutableLiveData Objektes

2.2.2 Verwendung mehrerer ViewModels

Damit die Redundanz der App verringert wird wurde für alle Activities und Fragmente ein übergeordnetes ViewModel ,das **BaseViewModel**, implementiert, welches Aufgaben löst die für alle Activities und Fragmente relevant sein könnten. Dazu zählen zum Beispiel allgemeine Zugriffe auf das Modell, sowie auf den Kalender. Da es aber häufig sein kann, dass diese Methoden nicht ausreichen, wenn zum Beispiel ein Element eine gefilterte Liste bekommen soll, ist für jede Activity und jedes Fragment ein eigenes ViewModel implementiert, welches für die speziellen Aufgaben vorgesehen ist. Jedes ViewModel erbt dabei vom **BaseViewModel**. Dieses Prinzip sorgt zum einen für mehr Flexibilität aber auch für mehr Ordnung.

2.3 View

Der letzte Teil des Patterns ist die View. Diese kümmert sich um alle grafischen Elemente der App. Im Vordergrund dabei stehen die Fragmente und Activities. In ihnen werden die grafischen Elemente verwaltet. Dabei werden zum Beispiel die Daten aus dem Modell auf dem Bildschirm angezeigt oder Benutzereingaben bekommen eine Funktion.

2.3.1 ViewModelProvider

Damit die View auf das entsprechende ViewModel zugreifen kann und somit Zugriff auf das Modell oder alle weitere Funktionen des ViewModels erhält, muss dieses über einen ViewModelProvider eingebunden werden. Dieser Vorgang ist im Ausschnitt 2.8 an einem Beispiel gezeigt. Dabei muss als Owner jeweils der aktuelle Context mit übergeben werden und anstelle von *MainViewModel::class.java* muss die Klasse des passenden ViewModels übergeben werden. Zusätzlich muss auch noch eine *ViewModelFactory* mit übergeben werden. Deinen Implementierung ist im Ausschnitt 2.9 gezeigt.

```
1 viewModel = ViewModelProvider(  
2     this, //Owner  
3     ViewModelFactory(application)  
4 ) [MainViewModel::class.java]
```

Codeausschnitt 2.8: Einbinden eines ViewModels mit Hilfe eines ViewModelProviders

```
1 class ViewModelFactory(application: Application) :  
    ViewModelProvider.AndroidViewModelFactory(application)
```

Codeausschnitt 2.9: ViewModelFactory zum einbinden eines ViewModels in ein Fragment oder eine Activity

Das ViewModel sollte dabei als globales Attribut deklariert werden, damit von der gesamten Activity oder dem gesamten Fragment zugegriffen werden kann.

2.3.2 LiveData Observer

Damit von der View auf die Daten im ViewModel Zugegriffen werden kann die als LiveData verfügbar sind, muss diesen LiveData Objekten ein Observer übergeben werden. Dieser wird jedes mal aufgerufen sobald sich die entsprechenden Daten verändern. Der Ausschnitt 2.9 zeigt die Definition eines solchen Observers an einem Beispiel. Ein Problem ist, dass bei der Verwendung des Observers innerhalb eines Fragmentes es schnell dazu kommen kann, dass dieser mehrfach dem LiveData Objekt hinzugefügt werden kann. Das liegt daran, dass der Observer in der Regel in der `onCreateView` Methode erstellt wird, da er meistens in Zusammenhang mit einem UI Element benötigt wird. Diese Methode wird im Lifecycle des Fragmentes allerdings jedes mal aufgerufen, nachdem die Ansicht zerstört wurde. Dies ist zum Beispiel der Fall wenn einmal von dem Fragment weg und wieder zurück navigiert wurde. Der komplette Lifecycle des Fragmentes ist im Anhang 11.2 zu finden. Um dieses Problem zu beheben kann dem Observer als Owner der `viewLifecycleOwner` übergeben werden. Dieser sorgt dafür, dass der Observer nur ein einziges mal dem LiveData Objekt hinzugefügt wird, egal wie oft die Methode aufgerufen wird.

```
1 viewModel.liveFavorites.observe(  
2     viewLifecycleOwner // Owner  
3 ) { entryList ->  
4     if (entryList != null) {  
5         recyclerViewFavoriteAdapter.updateContent(  
6             Filter.validateList(entryList))  
7     }  
}
```

Codeausschnitt 2.10: Beispiel für die Definition eines Observers für ein LiveData Objekt

Im Anschluss an die Methode `observe` muss implementiert werden, wie mit den veränderten Daten umgegangen werden soll. Dies wird dann jedes mal ausgeführt, sobald sich etwas in den Daten verändert.

Kapitel 3

CalendarIO

3.1 Smartphone Kalender

3.2 Events

3.2.1 InsertionType

3.2.2 Einfügen

3.2.3 Update

3.2.4 Löschen

3.3 Id Management

Kapitel 4

Update Manager

Kapitel 5

Push Service

Kapitel 6

Background Worker

Fazit

Kapitel 7

Themenattribute

- themeName
- colorPrimary
- colorOnPrimary
- colorPrimaryLight
- colorOnPrimaryLight
- colorPrimaryDark
- colorOnPrimaryDark
- colorAccent
- colorOnAccent
- colorBackground
- colorOnBackground
- actionMenuTextColor

Kapitel 8

Codeausschnitte

8.1 Filter

8.1.1 Filtervalidierung

```
1 fun validateFilter(entry: TestPlanEntry?): Boolean {
2     if (entry == null) {
3         return false
4     }
5     if (modulName != null && entry.module?.lowercase() ?.
6         startsWith(
7             modulName?.lowercase() ?: entry.module?.
8                 lowercase() ?: "-1"
9         ) == false
10    ) {
11        return false
12    }
13    if (entry.course?.lowercase()?.startsWith(
14        courseName?.lowercase() ?: entry.course?.
15            lowercase() ?: "-1"
16        ) == false
17    ) {
18        return false
19    }
20    if (datum != null) {
21        val sdf = SimpleDateFormat("yyyy-MM-dd", Locale.
22            getDefault())
23        val date = entry.date?.let { sdf.parse(it) }
24        val comp = date?.date?.let { Date(date.year, date.
25            month, it, 0, 0, 0) }?:return false
26        if (!datum!!.atDay(comp)) {
27            return false
28        }
29    }
30    if (entry.firstExaminer?.lowercase()?.startsWith(
31        examiner?.lowercase() ?: entry.firstExaminer?.
32            lowercase() ?: "-1"
33        ) == false
34    ) {
35        return false
36    }
37    if (entry.semester != null && !semester[entry.semester
38        !!.toInt().minus(1)] {
```

```

32         return false
33     }
34     return true
35 }
36
37 fun validateList(list: List<TestPlanEntry>): List<TestPlanEntry>
38     > {
39     val ret = mutableListOf<TestPlanEntry>()
40     list.forEach {
41         if (validateFilter(it)) {
42             ret.add(it)
43         }
44     }
45     return ret
46 }
```

Codeausschnitt 8.1: Methoden zum Validieren von einer oder mehreren Prüfungen

8.2 Suchleiste

8.3 Retrofit2

8.3.1 GSON Objekte

GSONCourse

```

1 data class GSONCourse(
2     val CourseName: String,
3     val CourseShortName: String,
4     val FKFBID: String,
5     val SGID: String
6 )
```

Codeausschnitt 8.2: GSON Klasse zum Speichern der Daten eines Studienganges von der Restschnittstelle

GSONEntry

```

1 data class GSONEntry(
2     val FirstExaminer: String?,
3     val SecondExaminer: String?,
4     val Form: String?,
5     val Semester: String?,
6     val Date: String?,
7     val Module: String?,
8     val CourseName: String?,
9     val Termin: String?,
10    val ID: String?,
11    val Room: String?,
12    val CourseId: String?,
13    val Status: String?,
14    val Hint: String?,
15    val Color: String?,
16    val Timestamp: String?
```

17)

Codeausschnitt 8.3: Gson Klasse zum Speichern der Daten eines Prüfplaneintrages von der Restschnittstelle

GSONUuid

```
1 class GSONUuid {  
2     @SerializedName("uuid")  
3     @Expose  
4     var uuid: String? = null  
5 }
```

Codeausschnitt 8.4: Gson Klasse zum Speichern der Daten einer UUID von der Restschnittstelle

8.4 ViewModel

8.4.1 updateEntry

```
1 open fun updateEntry(old: TestPlanEntry, new: TestPlanEntry){  
2     viewModelScope.launch(coroutineExceptionHandler) {  
3         new.favorite = old.favorite  
4         repository.deleteEntry(old)  
5         repository.insertEntry(new)  
6     }  
7 }
```

Codeausschnitt 8.5: Ersetzen eines Prüfplaneintrages in der Datenbank mit erhalten des Favoritenstausses

```
1 private fun initSearchView(menu: Menu) {  
2     //Hole die Suchleiste aus dem Actionmenu (in der  
3     //ActionBar)  
4     val search: SearchView = menu.findItem(R.id.  
        menu_item_search).actionView as SearchView  
5     //Hole Referenz zum Autocomplete Objekt aus der  
     //Suchleiste  
6     val searchAutoComplete: SearchAutoComplete = search.  
        findViewById(R.id.search_src_text)  
7     //Observer fuer alle Pruefplaneinträge, wenn sich die  
     //Eintraege aendern soll die Autovervollstaendigung  
     //angepasst werden  
8     viewModel.liveEntriesOrdered.observe(this) { entryList  
9         ->  
10         val list: MutableList<String> = mutableListOf()  
11         //Fuege fuer jeden Pruefplaneintrag den Modulnamen der  
         //Autovervollstaendigung hinzu  
12         entryList?.forEach { entry -> list.add(entry.module  
             ?: "") }  
13         //Uebergebe die Autovervollstaendigung als Adapter  
         //dem zustaendigen Objekt  
14         searchAutoComplete.setAdapter(  
             SimpleSpinnerAdapter(  
                 this,
```

```

15         R.layout.
16             layout_simpler_spinner_adapter_item,
17             list
18         )
19     }
20
21     //Setze den OnClickListener fuer die
22     //Autovervollstaendigung. Wenn ein Eintrag ausgewaehlt
23     //wurde, fuege diesen in die Suchleiste und bestaetige
24     //die Suche (Submit ueber "true" Parameter)
25     search.AutoComplete.setOnItemClickListener { adapterView
26         , _, i, _ ->
27         search.setQuery(adapterView.getItemAtPosition(i).
28             toString(), true)
29     }
30     //Setze Listener fuer Nutzerinteraktionen mit der
31     //Suchleiste
32     search.setOnQueryTextListener(object : SearchView.
33         OnQueryTextListener {
34         //Wenn Text eingefuegt wurde, mache nichts
35         override fun onQueryTextChange(text: String?):
36             Boolean {
37             return true
38         }
39         //Wenn bestaetigt wurde, setzte den Filter und wechsle
40         //zur Pruefungsuebersicht
41         override fun onQueryTextSubmit(text: String?):
42             Boolean {
43             Filter.reset()
44             Filter.modulName = if (text.isNullOrEmpty())
45                 null else text
46             changeFragment(ExamOverviewFragment())
47             return true
48         }
49     })
50 }

```

Codeausschnitt 8.6: Implementierung der Suchleiste

Kapitel 9

Kotlin

Kotlin ist eine Programmiersprache, die 2016 von dem Unternehmen Jetbrains in der Version 1.0 veröffentlicht wurde und 2017 von Google zur offiziellen Programmiersprache für Android erklärt wurde.[6] [3]

Genauso wie Java werden Kotlinprogramme in der JVM (Java-Virtual-Machine) ausgeführt. Dies sorgt für gute Kompatibilität dieser beiden Sprachen und macht so den Umstieg deutlich einfacher. Ein entscheidender Vorteil ist zum Beispiel, das Javaklassen in Kotlinklassen eingebunden werden können. Dies erspart viel Arbeit, da keine neuen Klassen programmiert werden müssen und viel mit dem alten Wissen weiter gemacht werden kann. Zusätzlich gibt es auch Tools, zum Beispiel von Android Studio, die es einem ermöglichen, bestehenden Javacode automatisch in Kotlincode umzuwandeln. Dies funktioniert zum einen für die Dateien im eigenen Projekt, aber auch mit einkopiertem Code aus externen Quellen.

9.1 Vorteile gegenüber Java

Kotlin bietet aber auch einige Vorteile gegenüber Java. Dazu gehören

- Nullpointer Sicherheit
- Weniger Codezeilen (dadurch allerdings erschwerte Lesbarkeit)

[1]. Die Nullpointer Sicherheit wird in Kapitel 9.13.2 näher erläutert. Die Codeverkürzung wird dadurch erzielt, dass vieles unnötige weggelassen werden können. Dazu gehört zum Beispiel das Semikolon und die Getter-und Setter Methoden. Zudem gibt es viele Möglichkeiten zur Vereinfachung, wie zum Beispiel SAM aus Kapitel 9.21.2 oder die Nullüberprüfungen aus Kapitel 9.13.2.

9.2 Variablen

Kotlin ist eine statisch-typisierte Programmiersprache. Das bedeutet, dass der Datentyp von Variablen im Gegensatz zu dynamisch-typisierten Sprachen nicht der Laufzeit, sondern bereits bei der Kompilierung festgelegt wird. Daher muss bei der Deklarierung von Variablen der Datentyp festgelegt werden wie es in Kapitel 9.13.1 erklärt ist, aber auch Verkürzungstechniken wie

9.2.1 Definition

Variablen in Kotlin werden über die Schlüsselwörter *var* und *val* definiert. Dabei steht *var* für eine Variable, der zur Laufzeit neue Werte zugewiesen werden können. Variablen die mit *val* definiert wurden, erhalten nur einmal bei der Erzeugung einen Wert und können danach nur noch gelesen werden. Der Codeausschnitt in Abbildung 9.28 zeigt verschiedene Deklarations-und Initialisierungsmöglichkeiten von Variablen in Kotlin. Datentypen werden, wie in Zeile 1,2 zu sehen, mit der Erweiterung *:Typ* festgelegt. Wenn der Variable bei der Deklaration sofort ein Wert zugewiesen wird, kann die Typisierung auch weglassen werden, da der Compiler aus der Zuweisung den Datentyp automatisch bestimmt.

```
1  var a:Int = 3
2  val b = 2
3  a = 2
```

Codeausschnitt 9.1: Variablen

9.2.2 Null-Sicherheit

Eine Variable kann als Null-Sicher definiert werden. Das bedeutet, dass diese Variable den Wert *null* annehmen kann, ohne das beim Zugriff eine Nullpointer-Exception geworfen wird. Um dies zu ermöglichen, wird hinter den Datentyp ein Fragezeichen angehängt, wie es im Codeausschnitt 9.29 zu sehen ist. Der Zugriff auf eine Nullable-Variable muss allerdings immer mit einer Nullüberprüfung stattfinden. Dafür gibt es verschiedene Möglichkeiten. Die einfachste Möglichkeit ist in Zeile 4 zu sehen. Durch den Operator *?.* wird eine Nullüberprüfung durchgeführt und die Methode *Split()* wird nur ausgeführt, wenn *a* einen Wert besitzt. Sollte mit dem Ergebnis der Methode weiter gearbeitet werden, so muss für jede zusammenhängende Methode ebenfalls eine Nullüberprüfung durchgeführt werden, wie es in Zeile 6 zu sehen ist. Mit dem Operator *?:* wird in dem Fall eine Alternative angegeben, sollte die Variable *b* null enthalten. Mit dem *!!*-Operator in Zeile 5 wird die Nullable-Variable *a* für den Ausdruck in eine Not-Nullable-Variable umgewandelt. Wenn dies getan wird, muss allerdings vorher sichergestellt werden, dass die Variable zu keiner Zeit den Wert *null* annehmen kann oder es muss eine manuelle Nullüberprüfung durchgeführt werden. Für den Fall, dass eine Methode nur ausgeführt werden soll, wenn der Eingabeparameter nicht *null* ist, gibt es die *let*-Operation aus Zeile 7. Die Methode in den geschweiften Klammern wird nur ausgeführt wenn die aufrufende Variable einen Wert besitzt. Auf die Variable wird dann in den Klammern mit dem Schlüsselwort *it* zugegriffen.

```
1  var a:String? = null
2  print(a)
3  a = "Hello World!"
4  val b = a?.Split(" ")
5  val c = a!!.Split(" ")
6  print(b?:"Leeres Array")
7  b?.let{print(it)}
```

Codeausschnitt 9.2: Umgang mit Nullable-Variablen

9.3 Methoden

Methoden in Kotlin, werden mit dem Schlüsselwort *fun* generiert. Der folgende Methodenname ist frei wählbar. In den Klammern werden Parameter festgelegt und nach den Klammern kann mit dem Ausdruck *:Datentyp* ein Rückgabedatentyp für die Methode festgelegt werden. Im Beispiel 9.30 sind ein paar Varianten aufgeführt. In Zeile 1 ist eine Methode ohne Rückgabewert und ohne Parameter zu sehen. In Zeile 10 ist eine Methode zu sehen, die zwei mögliche Eingabeparameter und einen Rückgabeparameter besitzt. Der zweite Eingabeparameter kann allerdings auch weggelassen werden da für ihn ein Default-Wert festgelegt wurde. In Zeile 15 ist eine Methode mit einer variablen Anzahl an Eingabeparameter definiert. Das heißt der Methode können beliebig viele Werte von einem Datentyp übergeben werden und die Methode kann dann über diese Werte iterieren. Feste und variable Parameter können auch kombiniert werden, wie es in Zeile 23 der Fall ist.

```
1 fun main() {
2     val a = add(1,2)
3     val b = add(1)
4     print(a.toString())
5     val c = add(1,2,3,4,5)
6     print(b.toString())
7     addAndPrint(1,2)
8 }
9
10 fun add(wert1:Int, wert2:Int=0):Int{
11     val sum = wert1+wert2
12     return sum
13 }
14
15 fun add(vararg werte:Int):Int{
16     var sum = 0
17     for(num in werte){
18         sum += num
19     }
20     return sum
21 }
22
23 fun add2(wert1:Int, vararg weitere:Int):Int{
24     var sum = wert1
25     for(zahl in weitere){
26         sum += zahl
27     }
28     return sum
29 }
```

Codeausschnitt 9.3: Methoden

9.4 Datenstrukturen

Wenn mehrere Daten in einer Liste abgespeichert werden sollen, so wird eine Datenstruktur benötigt. In dieser sind beliebig viele Daten eines Datentyps miteinander verkettet und können über einen Index abgerufen werden. Im folgenden sind drei wichtige Datenstrukturen erläutert.

- Array

- Set
- Map
- List

9.4.1 Array

Ein Array ist eine statische Datenstruktur, das heißt es ist in seiner Größe unveränderlich. Erzeugt werden kann ein Array über die Methode `arrayOf(vararg values:T)` wobei als Parameter die zu füllenden Werte übergeben werden. Soll das Array zu Beginn keine Werte enthalten, so lässt es sich mittels der Methode `arrayOfNulls<T>(size:Int)` erzeugen. Dabei muss mit `T` der Datentyp und `size` die Größe des Arrays festgelegt werden. Der resultierende Datentyp des Array ist dann allerdings unabhängig der Vorgabe Nullable (Array<T?>). Für einige Basisklassen wie

- Int
- Long
- Float
- Double
- Boolean

sind bereits Arrays vorhanden, diese können dann mit Methoden wie `intArrayOf(vararg elements:Int)` erzeugt werden. Um auf ein Element des Arrays zuzugreifen, wird entweder der `[]`-Operator oder die `.get(index:Int)`-Methode verwendet[5]. Sollen mehrere Elemente abgerufen werden, so kann über das Array mit einer For-Schleife iteriert werden. Diese wird in Abschnitt 9.16 genauer erläutert. Der Codeausschnitt 9.31 zeigt ein Beispiel für den Umgang mit Arrays.

```

1  val a = intArrayOf(1,2,3,4,5)
2  val b = arrayOf<String>("eins", "zwei", "drei")
3  val c: Array<Double?> = arrayOfNulls(3)
4  c[0] = 1.0
5  c[1] = 1.1
6  c[2] = 1.2
7  for(i in a){
8      print(i)
9  }
10 for(i in b){
11     print(i)
12 }
13 for(i in c){
14     print(i)
15 }
```

Codeausschnitt 9.4: ArrayList

9.4.2 ArrayList

Wenn ein veränderbares Array benötigt wird, kann eine *ArrayList* verwendet werden. Die Erzeugung und Abfrage einer *ArrayList* verläuft gleich wie ein normales Array, allerdings ist es möglich, mit Hilfe von Methoden wie

- add
- addAll
- remove

das Array zu manipulieren, also Elemente hinzufügen und löschen, ohne auf eine feste Größe angewiesen zu sein.

9.4.3 Set

Ein Set ist ebenfalls eine Struktur von Daten. Die Besonderheit ist allerdings, das in einem Set jedes Element höchstens einmal existiert. Auf Grund dessen sind mit Sets mathematische Mengenoperationen wie zum Beispiel

- Schnittmenge - intersect()
- Vereinigungsmenge - union()
- Differenzmenge - minus()

möglich[5]. Man unterscheidet bei Sets zwischen *mutable*-und *not-mutable* Sets. Ein *mutableSet* ist von der Größe veränderbar, ein *not-mutable*-Set nicht. Erzeugt werden Sets genau wie Arrays mit Methoden wie *setOf()* beziehungsweise *mutableSetOf()*.

9.4.4 Map

In einer map werden Schlüsselwertpaare abgespeichert. Das heißt zu einem Schlüssel eines Datentyps wird ein Wert eines beliebigen Datentyps zugeordnet. Der Schlüssel ist dabei eindeutig, das heißt er existiert nur einmal. Der zugehörige Wert kann auch mehrfach in der Map vorkommen. Die Erzeugung funktioniert mit der Methode *mapOf()* beziehungsweise *mutableMapOf()* wobei als Parameter Schlüssel-Wert-Paare der Form (*schlüssel to wert*) übergeben werden. Die Abfrage verläuft über den Schlüssel, welcher als Index verwendet wird. Ein Beispiel für die Erzeugung und Abfrage ist im Codeausschnitt 9.32 beispielhaft gezeigt.

```
1 val a:Map<String ,Int> = mapOf("1" to 1,"2" to 2,"3" to 3)
2 print(a["1"])
3 print(a["2"])
4 print(a["3"])
```

Codeausschnitt 9.5: Map

9.4.5 List

Die Liste wurde in der App am häufigsten verwendet. Es handelt sich dabei um eine simple Liste von Daten, ähnlich wie die *ArrayList*. Auch hier wird zwischen *MutableList* und *List* unterschieden.

9.5 Schleifen

Um mehrere Elemente einer Datenstruktur abzufragen, wird eine Schleife benötigt. In Kotlin gibt es viele Möglichkeiten, eine Schleife zu implementieren. Die wichtigsten sind in den folgenden Abschnitten aufgeführt.

9.5.1 While

Eine While-Schleife durchläuft einen Codeabschnitt so lange, bis eine bestimmte Bedingung nicht mehr erfüllt wird, oder manuell aus der Schleife ausgetreten wird. Ein Beispiel für eine While-Schleife ist im Codeausschnitt 9.33 zu sehen.

```
1 var zaehler = 0
2 while(zaehler < 10)
3 {
4     print(zaehler)
5     zaehler++
6 }
```

Codeausschnitt 9.6: While-Schleife

Mit dem Schlüsselwort *break* kann die Schleife schon vorzeitig beendet werden, mit dem Schlüsselwort *continue* wird sofort mit der nächsten Iteration fortgefahren. Dies gilt auch für die anderen Schleifen.

9.5.2 For

Mit einer For-Schleife wird über eine Sammlung von Daten iteriert. Dies kann zum Beispiel ein Zahlenbereich sein, es kann sich aber auch um eine Datenstruktur irgendeines Datentyps handeln. Der Codeausschnitt 9.34 zeigt eine For-Schleife über die Zahlen von 0 bis 9. Der Zahlenbereich wird mit dem Schlüsselwort *until* aufgebaut, wobei die 10 in diesem Fall nicht mit eingeschlossen ist. Die Variable *i* ist in dem Fall die Iterationsvariable, sie nimmt nacheinander jeden Wert des Zahlenbereichs an.

```
1 for(i in 0 until 10){
2     print(i)
3 }
```

Codeausschnitt 9.7: For-Schleife über einen Zahlenbereich

Der Codeausschnitt 9.35 zeigt eine Iteration über eine Datenstruktur. Die Variable *item* nimmt in diesem Fall alle Werte innerhalb der Datenstruktur nacheinander an.

```
1 val a = arrayOf("Eins", "Zwei", "Drei", "Vier", "Fuenf")
2 for(item in a){
3     print(item)
4 }
```

Codeausschnitt 9.8: For-Schleife über eine Datenstruktur

9.5.3 Foreach

Die *forEach*-Schleife ist eine Alternative zur For-Schleife. Sie wird von einer Datenstruktur als Methode aufgerufen und führt einen bestimmten Codeblock für jedes Element der Datenstruktur aus. Auf das Element aus der Datenstruktur kann wie im Beispiel 9.36 in Zeile 3 zu sehen über das Schlüsselwort *it* oder wie in Zeile 2 über eine selbst benannte Variable zugegriffen werden.

```
1 val a = arrayOf("Eins", "Zwei", "Drei", "Vier", "Fuenf")
2     a.forEach{i -> print(i)}
3     a.forEach{print(it)}
```

Codeausschnitt 9.9: ForEach-Schleife über eine Datenstruktur

9.6 Verzweigungen

Natürlich gibt es auch in Kotlin Verzweigungen, wo eine Bedingung geprüft wird, und je nach Ergebnis ein anderer Codepfad durchlaufen wird.

9.6.1 If-Verzweigung

Die einfachste Möglichkeit ist die If-Verzweigung. Diese überprüft eine Bedingung auf Wahr(true) oder Falsch(false) und wählt je nach Ergebnis einen von zwei Pfaden. Ein Beispiel ist im Codeausschnitt 9.37 gezeigt. Eine Besonderheit in Kotlin ist, dass wie in Zeile 8 zu sehen, eine If-Verzweigung in einen Ausdruck mit eingebaut werden kann. Die Variable *c* nimmt dann abhängig der Variablen *a* und *b* einen anderen Wert an.

```
1 val a = 4
2 val b = 2
3 if(a<b){
4     print(a)
5 } else{
6     print(b)
7 }
8 val c = if(a<b) a else b
9 print(c)
```

Codeausschnitt 9.10: If-Verzweigung

When

Eine Erweiterung der If-Verzweigung ist die When-Verzweigung. Diese ist die Kotlin alternative zu Javas Switch-Case und vergleicht einen Wert mit einer beliebigen Anzahl anderer Werte. Bei Gleichheit wird dann der entsprechende Code ausgeführt. Die Implementierung der When-Verzweigung ist im Codeausschnitt 9.38 an einem Beispiel gezeigt. Auf der linken Seite des Pfeiles -> steht jeweils der Wert, der mit dem Wert *a* verglichen werden soll, auf der rechten Seite steht der Code, der bei Gleichheit ausgeführt werden soll. Mit dem Schlüsselwort *else* wird beschrieben was passiert, wenn keiner der Fälle zu trifft. Dieser Zweig kann aber auch weggelassen werden. [5]

```

1 val a = 3
2   when(a) {
3     1 -> print("eins")
4     2 -> print("zwei")
5     3 -> print("drei")
6     4 -> print("vier")
7     5 -> print("fuenf")
8   else -> print("anderer Wert")
9 }
```

Codeausschnitt 9.11: When-Verzweigung

9.7 Klassen

Kotlin ist eine Objektorientierte Programmiersprache. Klassen, oder auch Objekte, spielen daher eine entscheidende Rolle bei der Programmierung.

9.7.1 Definition

Eine normale Kotlin-Klasse wird mit dem Schlüsselwort *class* erzeugt. Die Sichtbarkeit der Klasse ist durch das vorangestellte Schlüsselwort festgelegt.

- **public** - Auf die Klasse kann aus jeder anderen Klasse zugegriffen werden
- **protected** - Auf die Klasse kann nur innerhalb der Elternklasse, oder allen Klassen, die von der Elternklasse erben, zugegriffen werden
- **private** - Auf die Klasse kann nur von der Elternklasse zugegriffen werden

Innerhalb einer Klassen können verschiedene Parameter und Methoden definiert werden. Für beide kann die Sichtbarkeit ebenfalls eingeschränkt werden. Jeder Parameter und jede Methode ist per default *public*.

```

1 public class TestKlasse{
2
3   private var ersterParameter:String? = null
4
5   val zweiterParameter = 1
6
7   private fun ersteMethode() {
8   }
9
10 fun zweiteMethode() {
11   }
12 }
```

Codeausschnitt 9.12: Einfache Klasse

9.7.2 Konstruktoren

Um von einer Klasse ein Objekt zu erzeugen, muss ein Konstruktor der Klasse aufgerufen werden. Wurde kein eigener Konstruktor definiert, so wird automatisch ein leerer Konstruktor ohne Parameter und ohne Code erzeugt. Wenn ein speziellerer Konstruktor benötigt wird, gibt es verschiedene Möglichkeiten, diesen zu erzeugen.

Primärkonstruktor

Der Primärkonstruktor steht, wie in Beispiel 9.40 zu sehen, direkt hinter dem Klassennamen. Diese Parameter müssen dann allerdings direkt an globale Parameter der Klasse übergeben werden. Sie können selber nicht innerhalb von Klassenmethoden verwendet werden.[5]

```
1 class Test(a:String ,b:Int){  
2     private val a = a  
3     private val b = b  
4 }
```

Codeausschnitt 9.13: Primärkonstruktor

Alternativ kann den Variablen im Primärkonstruktor, wie in Beispiel 9.41 zu sehen, eine Deklaration vorangestellt werden, wodurch der Parameter als globale Variable der Klasse gesehen wird.

```
1 class Test(private val a:String ,var b:Int){  
2  
3 }
```

Codeausschnitt 9.14: Primärkonstruktor mit integrierten Variablen

Initialisierungsblock

Wenn für die Zuweisung der Primärkonstruktorparameter eine komplexere Logik als die direkte Zuweisung benötigt wird, ist es möglich, die Zuweisungen in einen Initialisierungsblock zu verschieben. Dieser wird mit dem Schlüsselwort *init*, gefolgt von einem Codeblock implementiert. Ausschnitt 9.42 zeigt ein Beispiel für eine mögliche Implementierung.[5]

```
1 class Test(aInput:String ,bInput:Int){  
2  
3     private var a:String  
4     private var b:Int  
5  
6     init{  
7         a = if(aInput=="") "unbekannt" else aInput  
8         b = if(bInput<0) 0 else bInput  
9     }  
10 }
```

Codeausschnitt 9.15: Initialisierungsblock

Sekundärkonstruktoren

Wenn es für eine Klasse mehrere verschiedene Konstruktoren geben sollen, zum Beispiel weil manche Parameter auf Grund von Defaultwerten keine Initialisierung benötigen, so können Sekundärkonstruktoren verwendet werden. Ein Sekundärkonstruktor wird mit dem Schlüsselwort *constructor*, gefolgt von einer Parameterliste und einem Methodenblock definiert. Sekundärkonstruktoren können sich auch gegenseitig aufrufen, dafür wird der Methodenblock weggelassen, und hinter die Parameterliste der Ausdruck *:this(param1,param2 ...)* angehängt. Dabei werden für die Parameter *(param1,param2 ...)* die passenden Werte für

den aufgerufenen Konstruktor eingesetzt. Dies können zum Beispiel Parameter des aufrufenden Konstruktors oder Defaultwerte sein. Um eindeutig zu bleiben, muss jeder Sekundärkonstruktor eine Parameterliste, mit unterschiedlichen Datentypen besitzen. Dies ist in Beispiel 9.43 zu sehen.

```

1  class Test{
2
3      private var a:String
4      private var b:Int
5
6      constructor(pa:String ,pb:Int){
7          a = if(pa=="") "unbekannt" else pa
8          b = if(pb<0)0 else pb
9      }
10
11     constructor(pa:String) :this(pa,0)
12     constructor(pb:Int) :this(" ",pb)
13     constructor() :this(" ",0)
14 }
```

Codeausschnitt 9.16: Sekundärkonstruktoren

Soll ein Primärkonstruktor mit weiteren Sekundärkonstruktoren kombiniert werden, so muss jeder Sekundärkonstruktor den Primärkonstruktor aufrufen, entweder direkt oder über einen anderen Sekundärkonstruktor.[5]

9.7.3 Vererbung

Genau wie in anderen objektorientierten Programmiersprachen ist es auch in Kotlin möglich, dass Objekte von anderen Objekten erben. Dabei übernimmt die erbende Klasse (Kindklasse) alle öffentlichen (public) oder geschützten (protected) Attribute und Methoden der Basisklasse (Elternklasse). Damit von einer Elternklasse geerbt werden kann, muss diese mit dem Schlüsselwort *open* definiert worden sein. Zusätzlich muss jede Methode und jedes Attribut der Elternklasse mit dem Schlüsselwort *open* versehen werden, wenn die Kindklasse diese überschreiben können soll. Die Vererbung findet mit dem *:*-Operator statt. In Beispiel 9.44 erbt die Klasse B von der Klasse A und überschreibt sowohl das Attribut *text* als auch die Methode *print()*.

```

1  open class A{
2      open protected val text:String = "Hello World!"
3      open fun print(){
4          println(text)
5      }
6  }
7
8  class B:A(){
9      override val text = "World Hello!"
10     override fun print(){
11         println(text.reversed())
12     }
13 }
```

Codeausschnitt 9.17: Vererbung

9.8 Erweiterungsmethoden

Da es allerdings schnell unüberschaubar werden kann, wenn jedes mal, wo zum Beispiel eine neue Methode für die Elternklasse benötigt wird, eine neue Kindklasse implementiert werden muss, gibt es in Kotlin eine elegante Lösung. Mit so genannten *Erweiterungsmethoden* kann einer bereits bestehenden Klasse eine neue Methode hinzugefügt werden, ohne eine Kindklasse erstellen zu müssen. Das ist besonders nützlich für Klassen aus fremden Paketen wie zum Beispiel der Kotlin Standardbibliothek, auf die nur lesend zugegriffen werden kann. Die Methoden müssen außerhalb einer Klasse stehen und werden wie im Beispiel 9.45 in Zeile 6 implementiert. Dabei ist zu beachten, dass innerhalb der Methode ausschließlich nur auf die öffentlichen Attribute und Methoden der Klasse zugegriffen werden kann. Private oder Schützte bleiben weiterhin verborgen.

```
1 fun main() {
2     val i:Int = 8
3     println(i.half())
4 }
5
6 fun Int.half():Int{
7     return this/2
8 }
```

Codeausschnitt 9.18: Erweiterungsmethoden

9.9 Object und Companion

In Java gibt es neben normalen Klassen auch noch statische Klassen. Diese sind im gesamten Projekt nur ein einziges mal vorhanden und können daher ohne vorherige Instanziierung verwendet werden. Dies ist nützlich um zum Beispiel Hilfsmethoden zur Verfügung zu stellen, für die keine Instanz eines Objektes benötigt wird. Die Alternative dafür in Kotlin sind Klassen, die an Stelle des Schlüsselwortes *class* mit dem Schlüsselwort *object* definiert werden. Beispiel 9.46 zeigt wie eine statische Klasse in Kotlin implementiert und verwendet wird.

```
1 fun main() {
2     Printer.print()
3     Printer.text = Printer.text.reversed()
4     Printer.print()
5 }
6
7 object Printer{
8     var text = "Hello World!"
9
10    fun print(){
11        println(text)
12    }
13 }
```

Codeausschnitt 9.19: Statische Klasse

Wenn allerdings eine nicht-statische Klasse mit statischen Methoden und Attributen ausgestattet werden soll, so kann innerhalb der Klasse ein Companion-Object definiert werden. Variablen und Methoden innerhalb des Companion-Objects werden als statisch erkannt und sind daher für alle Instanzen der Klasse

identisch. Das Companion-Object wird wie in Beispiel 9.47 mit den Schlüsselwörtern *companion object* definiert. Im Beispiel wurde ein Counter für die Klasse implementiert, der mitzählt wie oft ein Objekt der Klasse erzeugt wurde. Da das Attribut counter im Companionobjekt enthalten ist, ist sein Wert für alle Klassen der selbe.

```

1 class Klasse{
2     constructor(){
3         counter++
4     }
5     companion object{
6         var counter:Int = 0
7
8         fun printCounter(){
9             println(counter.toString())
10        }
11    }
12 }
```

Codeausschnitt 9.20: Companion Objekt

9.10 Interfaces

Ein Problem, was die Vererbung in der objektorientierten Programmierung mit sich bringt, ist die Tatsache, dass eine Kindklasse immer nur von höchstens einer Elternklasse erben kann. Wenn eine Klasse von mehreren Eltern erben soll, so ist dies nur über Interfaces möglich. Interfaces kommen dann zum Einsatz, wenn eine spezielle Klasse auf eine bestimmte Funktionalität beschränkt werden soll. Ein Beispiel wäre das speichern verschiedener Klassen in einer einzigen Liste, wenn alle eine Gemeinsamkeit teilen. Ein Beispiel ist in 9.48 zu sehen. Da wurden zwei Klassen implementiert, die jeweils zwei Interfaces implementieren. In Zeile 2 und 3 werden dann zwei Arrays erstellt, die jeweils Elemente von sowohl *Klasse1* als auch *Klasse2* enthalten. Die Klassen werden dann jeweils auf ihre Implementierung des Interfaces zurückgestuft. Weiter Funktionen der Klassen sind dann nicht mehr vom Array aus aufrufbar, sie können allerdings in den Interface-Methoden verwendet werden.

```

1 fun main() {
2     val a:Array<IA> = arrayOf(Klasse1(),Klasse2())
3     val b:Array<IB> = arrayOf(Klasse1(),Klasse2())
4     for(e in a){
5         e.printA()
6     }
7     for(e in b){
8         e.printB()
9     }
10 }
11
12 class Klasse1:IA,IB{
13     override val text = "Klasse1:IB"
14     override fun printA(){
15         println("Klasse1:IA")
16     }
17     override fun printB(){
18         println(text)
19     }
}
```

```

20 }
21
22 class Klasse2 : IA, IB{
23     override val text = "Klasse2:IB"
24     override fun printA(){
25         println("Klasse2:IA")
26     }
27     override fun printB(){
28         println(text)
29     }
30 }
31
32 interface IA{
33     fun printA()
34 }
35
36 interface IB{
37     val text:String
38     fun printB()
39 }

```

Codeausschnitt 9.21: Interfaces

Die Erstellung eines Interfaces funktioniert in Kotlin mit dem Schlüsselwort *interface* anstelle von *class*. Jedes Attribut und jede Methode eines Interfaces muss *public* sein, Attribute dürfen keine Werte im Interface zugewiesen bekommen und bei Methoden wird der Körper weggelassen. Wenn ein Interface von einer Klasse implementiert wird, muss jede Methode und jedes Attribut des Interfaces von der Klasse überschrieben werden.

9.10.1 Anonymes Interface

Neben der Implementierung in einer Klasse, kann ein Interface auch anonym verwendet werden. Dadurch muss nicht unbedingt eine Klasse erzeugt werden die das Interface implementiert, sondern das Interface wird zur Laufzeit im Code erzeugt. Dies wird zum Beispiel bei den *OnClickListener* der Buttons verwendet. Diese benötigen nämlich nur eine Implementation des OnClick-Interfaces und keine Klasse. Die Erzeugung eines anonymen Interfaces wird mit den Schlüsselwörtern *object:* gefolgt von dem Namen des Interfaces umgesetzt. Im daraufliegenden Block müssen dann alle notwendigen Funktionen des Interfaces überschrieben werden. Ein Beispiel für den Umgang mit anonymen Interfaces ist in Abbildung 9.49 gezeigt.

```

1 fun interface Print{
2     fun Print(text:String)
3 }
4
5 fun Execute(p:Print){
6     p.Print("Hello World")
7 }
8 fun main() {
9     Execute(object:Print{
10         override fun Print(text:String){
11             println(text)
12         }
13     })
14     Execute(object:Print{
15         override fun Print(text:String){

```

```

16         println(text.reversed())
17     }
18 }
19 }
```

Codeausschnitt 9.22: Anonyme Interfaces

9.10.2 Vereinfachung

Die Implementation eines anonymen Interfaces lässt sich allerdings noch vereinfachen, um den Code kürzer zu gestalten. Diese Operation wird in Kotlin *SAM* genannt. Dies ist dann möglich, wenn es im Interface nur eine einzige Methode gibt. Im ersten Schritt wird das Schlüsselwort *object*: weggelassen und die Überschreibung der Methode durch einen Lambdaausdruck ersetzt. Dazu werden zuerst alle Eingabeparameter der Methode durch Komma getrennt deklariert und nach dem Lambda-Operator *->* wird dann der Programmcode implementiert. Wenn das Interface in einer Funktion als Eingabeparameter verwendet wird, die ansonsten keine weiteren Parameter benötigt, kann das Interface ohne Namen direkt im Anschluss an den Funktionsnamen geschrieben werden. Dabei werden auch die runden Klammern der Funktion weggelassen. Dies ist in Abbildung 9.50 zu sehen.

```

1 fun interface Print{
2     fun Print(text: String)
3 }
4
5 fun Execute(p:Print){
6     p.Print("Hello World")
7 }
8
9 fun main() {
10     Execute{text->
11         println(text)
12     }
13     Execute{text->
14         println(text.reversed())
15     }
16 }
```

Codeausschnitt 9.23: Anonyme Interfaces

Wenn die Methode ebenfalls nur höchstens einen einzigen Parameter besitzt, so kann dieser ebenfalls weggelassen werden, und auf den Parameter wird dann mit dem Schlüsselwort *it* zugegriffen. Dies ist in Abbildung 9.51 gezeigt.

```

1 fun interface Print{
2     fun Print(text: String)
3 }
4
5 fun Execute(p:Print){
6     p.Print("Hello World")
7 }
8
9 fun main() {
10     Execute{
11         println(it)
12     }
13 }
```

```

13     Execute{
14         println(it.reversed())
15     }
16 }
```

Codeausschnitt 9.24: Zweite Vereinfachung des Anonymen Interfaces

```

1 fun interface Print{
2     fun Print(text:String)
3 }
4
5 fun Execute(p:Print){
6     p.Print("Hello World")
7 }
8
9 fun main() {
10    //Ohne SAM
11    Execute(object:Print{
12        override fun Print(text:String){
13            println(text)
14        }
15    })
16    //Mit SAM
17    Execute{
18        println(it.reversed())
19    }
20 }
```

Codeausschnitt 9.25: Gegenüberstellung mit und ohne SAM

In Abbildung 9.52 sind die Implementierungen mit und ohne SAM gegenübergestellt. Dabei ist deutlich zu sehen, wie sehr die Verkürzung der Übersichtlichkeit hilft.

9.11 Coroutines

Damit eine App benutzerfreundlich ist, ist eine wichtige Voraussetzung, dass der Nutzer zu jeder Zeit mit der App interagieren kann ohne auf eine Reaktion der App warten zu müssen. Wenn in einer App allerdings aufwendige Aufgaben gelöst werden müssen, kann dies auch mal länger dauern. Ein Beispiel wäre die Abfrage einer Datenbank, die Aufgrund der Netzverbindung und der eventuell großen Datenmengen längere Zeit in Anspruch nehmen kann. Würde diese Abfrage auf dem selben Thread wie die UI ausgeführt werden, so müsste der Nutzer solange warten, bis die Abfrage fertig ist, bis er mit der Bedienung der App fortfahren könnte. Um dieses Problem zu lösen, bietet Kotlin die Coroutines an. Diese lassen Code Asynchron zur UI ablaufen, wodurch diese nicht blockiert werden würde. Da das Thema zu Coroutines sehr umfangreich ist, wird hier nur der Teil erklärt, der auch in dem Projekt umgesetzt wurde.

9.11.1 Suspend

Damit eine Funktion in Kotlin asynchron zum Mainthread arbeiten kann, muss sie mit dem Schlüsselwort *suspend* versehen werden. Dies sorgt dafür, dass die Funktion gestartet, gestoppt und fortgesetzt werden kann[2]. Eine solche Funktion muss immer entweder in einem Coroutinescope, welches in Kapitel 9.22.2

näher erläutert wird, oder in einer weiteren Suspendfunction ausgeführt werden. Dies sorgt für Sicherheit, dass die Funktion den Mainthread nicht blockiert.

9.11.2 Coroutine Scope

Um eine Suspendfunction in einer synchronen Funktion aufzurufen, muss zuerst ein *Coroutinescope* definiert werden. Dieses kann ein globales Attribut einer Klasse sein, kann allerdings auch innerhalb einer Methode erzeugt werden.

Definition

Als globales Attribut kann das Scope wie in Beispiel 9.53 erstellt werden.

```
1 private val scope: CoroutineScope = CoroutineScope(CoroutineName
    ("Scope") + Dispatchers.IO)
```

Codeausschnitt 9.26: Erzeugung eines Coroutinescopes

Dabei ist der Name des Scopes frei wählbar. Innerhalb des Konstruktors der Klasse *CoroutineScope()* werden bestimmte Eigenschaften für das Scope festgelegt. In diesem Fall wird dem Scope ein Name gegeben und ein Dispatcher zugeordnet. Die Attribute müssen mit einem + voneinander getrennt werden.

Dispatcher

Der Dispatcher gibt an, in welcher Umgebung das Scope ausgeführt wird. Die drei Möglichkeiten sind

- Main
- IO
- Default

Wenn der Main-Dispatcher gewählt werden würde, würde der Code innerhalb des Scopes auf dem selben Thread wie die UI ausgeführt werden. Der IO und der Default-Dispatcher laufen beide Asynchron zur UI, dabei ist der IO-Dispatcher für Netzwerk-oder Laufwerklastige Aufgaben, der Default-Dispatcher für CPU-lastige Aufgaben ausgelegt.

Aufruf

Um eine Coroutine in einem Scope zu starten, muss dies mit der Methode *launch* des Coroutinescopes eingeleitet werden. Jeglicher Code, der im Block dieser Methode ausgeführt wird, läuft asynchron auf dem vordefinierten Dispatcher. Ein Beispiel ist in Abbildung 9.54 zu sehen.

```
1 fun main() {
2     scope.launch{
3         System.out.println("Coroutine 1 started")
4         System.out.println("Calculation 1 started\n")
5         HeavyCalculation(100000000)
6         System.out.println("Calculation 1 ended\n")
7     }.invokeOnCompletion(){
8         System.out.println("Coroutine 1 finished\n")
9     }
}
```

```

10     scope.launch{
11         System.out.println("Coroutine 2 started")
12         System.out.println("Calculation 2 started\n")
13         HeavyCalculation(1000000)
14         System.out.println("Calculation 2 ended\n")
15     }.invokeOnCompletion(){
16         System.out.println("Coroutine 2 finished\n")
17     }
18 }
19
20 suspend fun HeavyCalculation(n:Long){
21     var result = 0L
22     for(i in 0 until n){
23         result += i
24     }
25     System.out.println(result)
26 }
```

Codeausschnitt 9.27: Aufruf einer Coroutine

Die Reihenfolge der Ausgabe für Abbildung 9.54 ist:

- Coroutine 1 Started
- Calculation 1 Started
- Coroutine 2 Started
- Calculation 2 Started
- Result 2
- Calculation 2 Ended
- Coroutine 2 finished
- Result 1
- Calculation 1 Ended
- Coroutine 1 finished

Mit Hilfe der Funktion *invokeOnCompletion* kann Programmcode nach Beendigung der Corouine ausgeführt werden.

Kotlin ist eine Programmiersprache, die 2016 von dem Unternehmen Jet-brains in der Version 1.0 veröffentlicht wurde und 2017 von Google zur offiziellen Programmiersprache für Android erklärt wurde.[6] [3]

Genauso wie Java werden Kotlinprogramme in der JVM (Java-Virtual-Machine) ausgeführt. Dies sorgt für gute Kompatibilität dieser beiden Sprachen und macht so den Umstieg deutlich einfacher. Ein entscheidender Vorteil ist zum Beispiel, das Javaklassen in Kotlinklassen eingebunden werden können. Dies erspart viel Arbeit, da keine neuen Klassen programmiert werden müssen und viel mit dem alten Wissen weiter gemacht werden kann. Zusätzlich gibt es auch Tools, zum Beispiel von Android Studio, die es einem ermöglichen, bestehenden Javacode automatisch in Kotlincode umzuwandeln. Dies funktioniert zum einen für die Dateien im eigenen Projekt, aber auch mit einkopiertem Code aus externen Quellen.

9.12 Vorteile gegenüber Java

Kotlin bietet aber auch einige Vorteile gegenüber Java. Dazu gehören

- Nullpointer Sicherheit
- Weniger Codezeilen (dadurch allerdings erschwerte Lesbarkeit)

[1]. Die Nullpointer Sicherheit wird in Kapitel 9.13.2 näher erläutert. Die Codeverkürzung wird dadurch erreicht, dass vieles unnötige weggelassen werden können. Dazu gehört zum Beispiel das Semikolon und die Getter-und Setter Methoden. Zudem gibt es viele Möglichkeiten zur Vereinfachung, wie zum Beispiel SAM aus Kapitel 9.21.2 oder die Nullüberprüfungen aus Kapitel 9.13.2.

9.13 Variablen

Kotlin ist eine statisch-typisierte Programmiersprache. Das bedeutet, dass der Datentyp von Variablen im Gegensatz zu dynamisch-typisierten Sprachen nicht der Laufzeit, sondern bereits bei der Kompilierung festgelegt wird. Daher muss bei der Deklarierung von Variablen der Datentyp festgelegt werden wie es in Kapitel 9.13.1 erklärt ist, aber auch Verkürzungstechniken wie

9.13.1 Definition

Variablen in Kotlin werden über die Schlüsselwörter *var* und *val* definiert. Dabei steht *var* für eine Variable, die zur Laufzeit neue Werte zugewiesen werden können. Variablen die mit *val* definiert wurden, erhalten nur einmal bei der Erzeugung einen Wert und können danach nur noch gelesen werden. Der Codeausschnitt in Abbildung 9.28 zeigt verschiedene Deklarations- und Initialisierungsmöglichkeiten von Variablen in Kotlin. Datentypen werden, wie in Zeile 1,2 zu sehen, mit der Erweiterung *:Typ* festgelegt. Wenn der Variable bei der Deklaration sofort ein Wert zugewiesen wird, kann die Typisierung auch weggelassen werden, da der Compiler aus der Zuweisung den Datentyp automatisch bestimmt.

```
1  var a:Int = 3
2  val b = 2
3  a = 2
```

Codeausschnitt 9.28: Variablen

9.13.2 Null-Sicherheit

Eine Variable kann als Null-Sicher definiert werden. Das bedeutet, dass diese Variable den Wert *null* annehmen kann, ohne das beim Zugriff eine Nullpointer-Exception geworfen wird. Um dies zu ermöglichen, wird hinter den Datentyp ein Fragezeichen angehängt, wie es im Codeausschnitt 9.29 zu sehen ist. Der Zugriff auf eine Nullable-Variable muss allerdings immer mit einer Nullüberprüfung stattfinden. Dafür gibt es verschiedene Möglichkeiten. Die einfachste Möglichkeit ist in Zeile 4 zu sehen. Durch den Operator *?.* wird eine Nullüberprüfung durchgeführt und die Methode *Split()* wird nur ausgeführt, wenn *a* einen Wert

besitzt. Sollte mit dem Ergebnis der Methode weiter gearbeitet werden, so muss für jede zusammenhängende Methode ebenfalls eine Nullüberprüfung durchgeführt werden, wie es in Zeile 6 zu sehen ist. Mit dem Operator ?: wird in dem Fall eine Alternative angegeben, sollte die Variable *b* null enthalten. Mit dem !!-Operator in Zeile 5 wird die Nullable-Variable *a* für den Ausdruck in eine Not-Nullable-Variable umgewandelt. Wenn dies getan wird, muss allerdings vorher sichergestellt werden, dass die Variable zu keiner Zeit den Wert *null* annehmen kann oder es muss eine manuelle Nullüberprüfung durchgeführt werden. Für den Fall, dass eine Methode nur ausgeführt werden soll, wenn der Eingabeparameter nicht *null* ist, gibt es die *let*-Operation aus Zeile 7. Die Methode in den geschweiften Klammern wird nur ausgeführt wenn die aufrufende Variable einen Wert besitzt. Auf die Variable wird dann in den Klammern mit dem Schlüsselwort *it* zugegriffen.

```

1  var a:String? = null
2  print(a)
3  a = "Hello World!"
4  val b = a?.Split(" ")
5  val c = a!!.Split(" ")
6  print(b?:"Leeres Array")
7  b?.let{print(it)}

```

Codeausschnitt 9.29: Umgang mit Nullable-Variablen

9.14 Methoden

Methoden in Kotlin, werden mit dem Schlüsselwort *fun* generiert. Der folgende Methodename ist frei wählbar. In den Klammern werden Parameter festgelegt und nach den Klammern kann mit dem Ausdruck *:Datentyp* ein Rückgabedatentyp für die Methode festgelegt werden. Im Beispiel 9.30 sind ein paar Varianten aufgeführt. In Zeile 1 ist eine Methode ohne Rückgabewert und ohne Parameter zu sehen. In Zeile 10 ist eine Methode zu sehen, die zwei mögliche Eingabeparameter und einen Rückgabeparameter besitzt. Der zweite Eingabeparameter kann allerdings auch weggelassen werden da für ihn ein Default-Wert festgelegt wurde. In Zeile 15 ist eine Methode mit einer variablen Anzahl an Eingabeparameter definiert. Das heißt der Methode können beliebig viele Werte von einem Datentyp übergeben werden und die Methode kann dann über diese Werte iterieren. Feste und variable Parameter können auch kombiniert werden, wie es in Zeile 23 der Fall ist.

```

1 fun main() {
2     val a = add(1,2)
3     val b = add(1)
4     print(a.toString())
5     val c = add(1,2,3,4,5)
6     print(b.toString())
7     addAndPrint(1,2)
8 }
9
10 fun add(wert1:Int, wert2:Int=0):Int{
11     val sum = wert1+wert2
12     return sum
13 }
14

```

```

15 fun add(vararg werte:Int):Int{
16     var sum = 0
17     for(num in werte){
18         sum += num
19     }
20     return sum
21 }
22
23 fun add2(wert1:Int, vararg weitere:Int):Int{
24     var sum = wert1
25     for(zahl in weitere){
26         sum += zahl
27     }
28     return sum
29 }

```

Codeausschnitt 9.30: Methoden

9.15 Datenstrukturen

Wenn mehrere Daten in einer Liste abgespeichert werden sollen, so wird eine Datenstruktur benötigt. In dieser sind beliebig viele Daten eines Datentyps miteinander verkettet und können über einen Index abgerufen werden. Im folgenden sind drei wichtige Datenstrukturen erläutert.

- Array
- Set
- Map
- List

9.15.1 Array

Ein Array ist eine statische Datenstruktur, das heißt es ist in seiner Größe unveränderlich. Erzeugt werden kann ein Array über die Methode `arrayOf(vararg values:T)` wobei als Parameter die zu füllenden Werte übergeben werden. Soll das Array zu Beginn keine Werte enthalten, so lässt es sich mittels der Methode `arrayOfNulls<T>(size:Int)` erzeugen. Dabei muss mit `T` der Datentyp und `size` die Größe des Arrays festgelegt werden. Der resultierende Datentyp des Array ist dann allerdings unabhängig der Vorgabe Nullable (Array<T?>). Für einige Basisklassen wie

- Int
- Long
- Float
- Double
- Boolean

sind bereits Arrays vorhanden, diese können dann mit Methoden wie *intArrayOf(vararg elements:Int)* erzeugt werden. Um auf ein Element des Arrays zuzugreifen, wird entweder der `[]`-Operator oder die `.get(index:Int)`-Methode verwendet[5]. Sollen mehrere Elemente abgerufen werden, so kann über das Array mit einer For-Schleife iteriert werden. Diese wird in Abschnitt 9.16 genauer erläutert. Der Codeausschnitt 9.31 zeigt ein Beispiel für den Umgang mit Arrays.

```

1  val a = intArrayOf(1,2,3,4,5)
2  val b = arrayOf<String>("eins", "zwei", "drei")
3  val c: Array<Double?> = arrayOfNulls(3)
4  c[0] = 1.0
5  c[1] = 1.1
6  c[2] = 1.2
7  for(i in a){
8      print(i)
9  }
10 for(i in b){
11     print(i)
12 }
13 for(i in c){
14     print(i)
15 }
```

Codeausschnitt 9.31: ArrayList

9.15.2 ArrayList

Wenn ein veränderbares Array benötigt wird, kann eine *ArrayList* verwendet werden. Die Erzeugung und Abfrage einer *ArrayList* verläuft gleich wie ein normales Array, allerdings ist es möglich, mit Hilfe von Methoden wie

- add
- addAll
- remove

das Array zu manipulieren, also Elemente hinzufügen und löschen, ohne auf eine feste Größe angewiesen zu sein.

9.15.3 Set

Ein Set ist ebenfalls eine Struktur von Daten. Die Besonderheit ist allerdings, dass in einem Set jedes Element höchstens einmal existiert. Auf Grund dessen sind mit Sets mathematische Mengenoperationen wie zum Beispiel

- Schnittmenge - `intersect()`
- Vereinigungsmenge - `union()`
- Differenzmenge - `minus()`

möglich[5]. Man unterscheidet bei Sets zwischen *mutable*-und *not-mutable* Sets. Ein *mutableSet* ist von der Größe veränderbar, ein *not-mutable*-Set nicht. Erzeugt werden Sets genau wie Arrays mit Methoden wie `setOf()` beziehungsweise `mutableSetOf()`.

9.15.4 Map

In einer map werden Schlüsselwertpaare abgespeichert. Das heißt zu einem Schlüssel eines Datentyps wird ein Wert eines beliebigen Datentyps zugeordnet. Der Schlüssel ist dabei eindeutig, das heißt er existiert nur einmal. Der zugehörige Wert kann auch mehrfach in der Map vorkommen. Die Erzeugung funktioniert mit der Methode `mapOf()` beziehungsweise `mutableMapOf()` wobei als Parameter Schlüssel-Wert-Paare der Form (*schlüssel to wert*) übergeben werden. Die Abfrage verläuft über den Schlüssel, welcher als Index verwendet wird. Ein Beispiel für die Erzeugung und Abfrage ist im Codeausschnitt 9.32 beispielhaft gezeigt.

```
1 val a:Map<String , Int> = mapOf( "1" to 1, "2" to 2, "3" to 3)
2 print(a["1"])
3 print(a["2"])
4 print(a["3"])
```

Codeausschnitt 9.32: Map

9.15.5 List

Die Liste wurde in der App am häufigsten verwendet. Es handelt sich dabei um eine simple Liste von Daten, ähnlich wie die `ArrayList`. Auch hier wird zwischen `MutableList` und `List` unterschieden.

9.16 Schleifen

Um mehrere Elemente einer Datenstruktur abzufragen, wird eine Schleife benötigt. In Kotlin gibt es viele Möglichkeiten, eine Schleife zu implementieren. Die wichtigsten sind in den folgenden Abschnitten aufgeführt.

9.16.1 While

Eine While-Schleife durchläuft einen Codeabschnitt so lange, bis eine bestimmte Bedingung nicht mehr erfüllt wird, oder manuell aus der Schleife ausgetreten wird. Ein Beispiel für eine While-Schleife ist im Codeausschnitt 9.33 zu sehen.

```
1 var zaehler = 0
2 while(zaehler<10)
3 {
4     print(zaehler)
5     zaehler++
6 }
```

Codeausschnitt 9.33: While-Schleife

Mit dem Schlüsselwort `break` kann die Schleife schon vorzeitig beendet werden, mit dem Schlüsselwort `continue` wird sofort mit der nächsten Iteration fortgefahren. Dies gilt auch für die anderen Schleifen.

9.16.2 For

Mit einer For-Schleife wird über eine Sammlung von Daten iteriert. Dies kann zum Beispiel ein Zahlenbereich sein, es kann sich aber auch um eine Datenstruktur irgendeines Datentyps handeln. Der Codeausschnitt 9.34 zeigt eine For-Schleife über die Zahlen von 0 bis 9. Der Zahlenbereich wird mit dem Schlüsselwort *until* aufgebaut, wobei die 10 in diesem Fall nicht mit eingeschlossen ist. Die Variable *i* ist in dem Fall die Iterationsvariable, sie nimmt nacheinander jeden Wert des Zahlenbereichs an.

```
1 for(i in 0 until 10){  
2     print(i)  
3 }
```

Codeausschnitt 9.34: For-Schleife über einen Zahlenbereich

Der Codeausschnitt 9.35 zeigt eine Iteration über eine Datenstruktur. Die Variable *item* nimmt in diesem Fall alle Werte innerhalb der Datenstruktur nacheinander an.

```
1 val a = arrayOf("Eins", "Zwei", "Drei", "Vier", "Fuenf")  
2 for(item in a){  
3     print(item)  
4 }
```

Codeausschnitt 9.35: For-Schleife über eine Datenstruktur

9.16.3 Foreach

Die *forEach*-Schleife ist eine Alternative zur For-Schleife. Sie wird von einer Datenstruktur als Methode aufgerufen und führt einen bestimmten Codeblock für jedes Element der Datenstruktur aus. Auf das Element aus der Datenstruktur kann wie im Beispiel 9.36 in Zeile 3 zu sehen über das Schlüsselwort *it* oder wie in Zeile 2 über eine selbst benannte Variable zugegriffen werden.

```
1 val a = arrayOf("Eins", "Zwei", "Drei", "Vier", "Fuenf")  
2     a.forEach{i -> print(i)}  
3     a.forEach{print(it)}
```

Codeausschnitt 9.36: ForEach-Schleife über eine Datenstruktur

9.17 Verzweigungen

Natürlich gibt es auch in Kotlin Verzweigungen, wo eine Bedingung geprüft wird, und je nach Ergebnis ein anderer Codepfad durchlaufen wird.

9.17.1 If-Verzweigung

Die einfachste Möglichkeit ist die If-Verzweigung. Diese überprüft eine Bedingung auf Wahr(true) oder Falsch(false) und wählt je nach Ergebnis einen von zwei Pfaden. Ein Beispiel ist im Codeausschnitt 9.37 gezeigt. Eine Besonderheit in Kotlin ist, dass wie in Zeile 8 zu sehen, eine If-Verzweigung in einen Ausdruck mit eingebaut werden kann. Die Variable *c* nimmt dann abhängig der Variablen *a* und *b* einen anderen Wert an.

```

1 val a = 4
2 val b = 2
3 if(a<b){
4     print(a)
5 } else{
6     print(b)
7 }
8 val c = if(a<b) a else b
9 print(c)

```

Codeausschnitt 9.37: If-Verzweigung

When

Eine Erweiterung der If-Verzweigung ist die When-Verzweigung. Diese ist die Kotlin alternative zu Javas Switch-Case und vergleicht einen Wert mit einer beliebigen Anzahl anderer Werte. Bei Gleichheit wird dann der entsprechende Code ausgeführt. Die Implementierung der When-Verzweigung ist im Codeausschnitt 9.38 an einem Beispiel gezeigt. Auf der linken Seite des Pfeiles -> steht jeweils der Wert, der mit dem Wert *a* verglichen werden soll, auf der rechten Seite steht der Code, der bei Gleichheit ausgeführt werden soll. Mit dem Schlüsselwort *else* wird beschrieben was passiert, wenn keiner der Fälle zu trifft. Dieser Zweig kann aber auch weggelassen werden. [5]

```

1 val a = 3
2 when(a) {
3     1 -> print("eins")
4     2 -> print("zwei")
5     3 -> print("drei")
6     4 -> print("vier")
7     5 -> print("fuenf")
8 } else -> print("anderer Wert")
9

```

Codeausschnitt 9.38: When-Verzweigung

9.18 Klassen

Kotlin ist eine Objektorientierte Programmiersprache. Klassen, oder auch Objekte, spielen daher eine entscheidende Rolle bei der Programmierung.

9.18.1 Definition

Eine normale Kotlin-Klasse wird mit dem Schlüsselwort *class* erzeugt. Die Sichtbarkeit der Klasse ist durch das vorangestellte Schlüsselwort festgelegt.

- public - Auf die Klasse kann aus jeder anderen Klasse zugegriffen werden
- protected - Auf die Klasse kann nur innerhalb der Elternklasse, oder allen Klassen, die von der Elternklasse erben, zugegriffen werden
- private - Auf die Klasse kann nur von der Elternklasse zugegriffen werden

Innerhalb einer Klassen können verschiedene Parameter und Methoden definiert werden. Für beide kann die Sichtbarkeit ebenfalls eingeschränkt werden. Jeder Parameter und jede Methode ist per default *public*.

```

1 public class TestKlasse{
2
3   private var ersterParameter:String? = null
4
5   val zweiterParameter = 1
6
7   private fun ersteMethode(){
8   }
9
10  fun zweiteMethode(){
11  }
12 }
```

Codeausschnitt 9.39: Einfache Klasse

9.18.2 Konstruktoren

Um von einer Klasse ein Objekt zu erzeugen, muss ein Konstruktor der Klasse aufgerufen werden. Wurde kein eigener Konstruktor definiert, so wird automatisch ein leerer Konstruktor ohne Parameter und ohne Code erzeugt. Wenn ein speziellerer Konstruktor benötigt wird, gibt es verschiedene Möglichkeiten, diesen zu erzeugen.

Primärkontruktor

Der Primärkonstruktor steht, wie in Beispiel 9.40 zu sehen, direkt hinter dem Klassennamen. Diese Parameter müssen dann allerdings direkt an globale Parameter der Klasse übergeben werden. Sie können selber nicht innerhalb von Klassenmethoden verwendet werden.[5]

```

1 class Test(a:String,b:Int){
2   private val a = a
3   private val b = b
4 }
```

Codeausschnitt 9.40: Primärkonstruktor

Alternativ kann den Variablen im Primärkonstruktor, wie in Beispiel 9.41 zu sehen, eine Deklaration vorangestellt werden, wodurch der Parameter als globale Variable der Klasse gesehen wird.

```

1 class Test(private val a:String,var b:Int){
2
3 }
```

Codeausschnitt 9.41: Primärkonstruktor mit integrierten Variablen

Initialisierungsblock

Wenn für die Zuweisung der Primärkonstruktorparameter eine komplexere Logik als die direkte Zuweisung benötigt wird, ist es möglich, die Zuweisungen in einen

Initialisierungsblock zu verschieben. Dieser wird mit dem Schlüsselwort `init`, gefolgt von einem Codeblock implementiert. Ausschnitt 9.42 zeigt ein Beispiel für eine mögliche Implementierung.[5]

```

1 class Test(aInput:String, bInput:Int) {
2
3     private var a:String
4     private var b:Int
5
6     init {
7         a = if(aInput=="") "unbekannt" else aInput
8         b = if(bInput<0) 0 else bInput
9     }
10 }
```

Codeausschnitt 9.42: Initialisierungsblock

Sekundärkonstruktoren

Wenn es für eine Klasse mehrere verschiedene Konstruktoren geben sollen, zum Beispiel weil manche Parameter auf Grund von Defaultwerten keine Initialisierung benötigen, so können Sekundärkonstruktoren verwendet werden. Ein Sekundärkonstruktor wird mit dem Schlüsselwort `constructor`, gefolgt von einer Parameterliste und einem Methodenblock definiert. Sekundärkonstruktoren können sich auch gegenseitig aufrufen, dafür wird der Methodenblock weggelassen, und hinter die Parameterliste der Ausdruck `:this(param1,param2 ...)` angehängt. Dabei werden für die Parameter `(param1,param2 ...)` die passenden Werte für den aufgerufenen Konstruktor eingesetzt. Dies können zum Beispiel Parameter des aufrufenden Konstruktors oder Defaultwerte sein. Um eindeutig zu bleiben, muss jeder Sekundärkonstruktor eine Parameterliste, mit unterschiedlichen Datentypen besitzen. Dies ist in Beispiel 9.43 zu sehen.

```

1 class Test{
2
3     private var a:String
4     private var b:Int
5
6     constructor(pa:String, pb:Int){
7         a = if(pa=="") "unbekannt" else pa
8         b = if(pb<0) 0 else pb
9     }
10
11    constructor(pa:String) : this(pa,0)
12    constructor(pb:Int) : this("",pb)
13    constructor() : this("",0)
14 }
```

Codeausschnitt 9.43: Sekundärkonstruktoren

Soll ein Primärkonstruktor mit weiteren Sekundärkonstruktoren kombiniert werden, so muss jeder Sekundärkonstruktor den Primärkonstruktor aufrufen, entweder direkt oder über einen anderen Sekundärkonstruktor.[5]

9.18.3 Vererbung

Genau wie in anderen objektorientierten Programmiersprachen ist es auch in Kotlin möglich, dass Objekte von anderen Objekten erben. Dabei übernimmt die

erbende Klasse (Kindklasse) alle öffentlichen (public) oder beschützten (protected) Attribute und Methoden der Basisklasse (Elternklasse). Damit von einer Elternklasse geerbt werden kann, muss diese mit dem Schlüsselwort `open` definiert worden sein. Zusätzlich muss jede Methode und jedes Attribut der Elternklasse mit dem Schlüsselwort `open` versehen werden, wenn die Kindklasse diese überschreiben können soll. Die Vererbung findet mit dem `:`-Operator statt. In Beispiel 9.44 erbt die Klasse B von der Klasse A und überschreibt sowohl das Attribut `text` als auch die Methode `print()`.

```

1  open class A{
2      open protected val text:String = "Hello World!"
3      open fun print(){
4          println(text)
5      }
6  }
7
8  class B:A(){
9      override val text = "World Hello!"
10     override fun print(){
11         println(text.reversed())
12     }
13 }
```

Codeausschnitt 9.44: Vererbung

9.19 Erweiterungsmethoden

Da es allerdings schnell unüberschaubar werden kann, wenn jedes mal, wo zum Beispiel eine neue Methode für die Elternklasse benötigt wird, eine neue Kindklasse implementiert werden muss, gibt es in Kotlin eine elegante Lösung. Mit so genannten *Erweiterungsmethoden* kann einer bereits bestehenden Klasse eine neue Methode hinzugefügt werden, ohne eine Kindklasse erstellen zu müssen. Das ist besonders nützlich für Klassen aus fremden Paketen wie zum Beispiel der Kotlin Standardbibliothek, auf die nur lesend zugegriffen werden kann. Die Methoden müssen außerhalb einer Klasse stehen und werden wie in Beispiel 9.45 in Zeile 6 implementiert. Dabei ist zu beachten, dass innerhalb der Methode ausschließlich nur auf die öffentlichen Attribute und Methoden der Klasse zugegriffen werden kann. Private oder Beschützte bleiben weiterhin verborgen.

```

1  fun main() {
2      val i:Int = 8
3      println(i.half())
4  }
5
6  fun Int.half():Int{
7      return this/2
8 }
```

Codeausschnitt 9.45: Erweiterungsmethoden

9.20 Object und Companion

In Java gibt es neben normalen Klassen auch noch statische Klassen. Diese sind im gesamten Projekt nur ein einziges mal vorhanden und können daher ohne

vorherige Instanziierung verwendet werden. Dies ist nützlich um zum Beispiel Hilfsmethoden zur Verfügung zu stellen, für die keine Instanz eines Objektes benötigt wird. Die Alternative dafür in Kotlin sind Klassen, die an Stelle des Schlüsselwortes *class* mit dem Schlüsselwort *object* definiert werden. Beispiel 9.46 zeigt wie eine statische Klasse in Kotlin implementiert und verwendet wird.

```

1 fun main() {
2     Printer.print()
3     Printer.text = Printer.text.reversed()
4     Printer.print()
5 }
6
7 object Printer{
8     var text = "Hello World!"
9
10    fun print(){
11        println(text)
12    }
13 }
```

Codeausschnitt 9.46: Statische Klasse

Wenn allerdings eine nicht-statische Klasse mit statischen Methoden und Attributen ausgestattet werden soll, so kann innerhalb der Klasse ein Companion-Object definiert werden. Variablen und Methoden innerhalb des Companion-Objects werden als statisch erkannt und sind daher für alle Instanzen der Klasse identisch. Das Companion-Object wird wie in Beispiel 9.47 mit den Schlüsselwörtern *companion object* definiert. Im Beispiel wurde ein Counter für die Klasse implementiert, der mitzählt wie oft ein Objekt der Klasse erzeugt wurde. Da das Attribut counter im Companionobjekt enthalten ist, ist sein Wert für alle Klassen der selbe.

```

1 class Klasse{
2     constructor(){
3         counter++
4     }
5     companion object{
6         var counter:Int = 0
7
8         fun printCounter(){
9             println(counter.toString())
10    }
11 }
12 }
```

Codeausschnitt 9.47: Companion Objekt

9.21 Interfaces

Ein Problem, was die Vererbung in der objektorientierten Programmierung mit sich bringt, ist die Tatsache, dass eine Kindklasse immer nur von höchstens einer Elternklasse erben kann. Wenn eine Klasse von mehreren Eltern erben soll, so ist dies nur über Interfaces möglich. Interfaces kommen dann zum Einsatz, wenn eine spezielle Klasse auf eine bestimmte Funktionalität beschränkt werden soll. Ein Beispiel wäre das speichern verschiedener Klassen in einer einzigen Liste,

wenn alle eine Gemeinsamkeit teilen. Ein Beispiel ist in 9.48 zu sehen. Da wurden zwei Klassen implementiert, die jeweils zwei Interfaces implementieren. In Zeile 2 und 3 werden dann zwei Arrays erstellt, die jeweils Elemente von sowohl *Klasse1* als auch *Klasse2* enthalten. Die Klassen werden dann jeweils auf ihre Implementierung des Interfaces zurückgestuft. Weiter Funktionen der Klassen sind dann nicht mehr vom Array aus aufrufbar, sie können allerdings in den Interface-Methoden verwendet werden.

```

1 fun main() {
2     val a:Array<IA> = arrayOf(Klasse1(), Klasse2())
3     val b:Array<IB> = arrayOf(Klasse1(), Klasse2())
4     for(e in a){
5         e.printA()
6     }
7     for(e in b){
8         e.printB()
9     }
10 }
11
12 class Klasse1:IA,IB{
13     override val text = "Klasse1:IB"
14     override fun printA(){
15         println("Klasse1:IA")
16     }
17     override fun printB(){
18         println(text)
19     }
20 }
21
22 class Klasse2:IA,IB{
23     override val text = "Klasse2:IB"
24     override fun printA(){
25         println("Klasse2:IA")
26     }
27     override fun printB(){
28         println(text)
29     }
30 }
31
32 interface IA{
33     fun printA()
34 }
35
36 interface IB{
37     val text:String
38     fun printB()
39 }
```

Codeausschnitt 9.48: Interfaces

Die Erstellung eines Interfaces funktioniert in Kotlin mit dem Schlüsselwort *interface* anstelle von *class*. Jedes Attribut und jede Methode eines Interfaces muss *public* sein, Attribute dürfen keine Werte im Interface zugewiesen bekommen und bei Methoden wird der Körper weggelassen. Wenn ein Interface von einer Klasse implementiert wird, muss jede Methode und jedes Attribut des Interfaces von der Klasse überschrieben werden.

9.21.1 Anonymes Interface

Neben der Implementierung in einer Klasse, kann ein Interface auch anonym verwendet werden. Dadurch muss nicht unbedingt eine Klasse erzeugt werden die das Interface implementiert, sondern das Interface wird zur Laufzeit im Code erzeugt. Dies wird zum Beispiel bei den *OnItemClickListener* der Buttons verwendet. Diese benötigen nämlich nur eine Implementation des OnClick-Interfaces und keine Klasse. Die Erzeugung eines anonymen Interfaces wird mit den Schlüsselwörtern *object:* gefolgt von dem Namen des Interfaces umgesetzt. Im darunterliegenden Block müssen dann alle notwendigen Funktionen des Interfaces überschrieben werden. Ein Beispiel für den Umgang mit anonymen Interfaces ist in Abbildung 9.49 gezeigt.

```
1 fun interface Print{
2     fun Print(text: String)
3 }
4
5 fun Execute(p: Print){
6     p.Print("Hello World")
7 }
8 fun main() {
9     Execute(object: Print{
10         override fun Print(text: String){
11             println(text)
12         }
13     })
14     Execute(object: Print{
15         override fun Print(text: String){
16             println(text.reversed())
17         }
18     })
19 }
```

Codeausschnitt 9.49: Anonyme Interfaces

9.21.2 Vereinfachung

Die Implementation eines anonymen Interfaces lässt sich allerdings noch vereinfachen, um den Code kürzer zu gestalten. Diese Operation wird in Kotlin *SAM* genannt. Dies ist dann möglich, wenn es im Interface nur eine einzige Methode gibt. Im ersten Schritt wird das Schlüsselwort *object:* weggelassen und die Beschreibung der Methode durch einen Lambdaausdruck ersetzt. Dazu werden zuerst alle Eingabeparameter der Methode durch Komma getrennt deklariert und nach dem Lambda-Operator *->* wird dann der Programmcode implementiert. Wenn das Interface in einer Funktion als Eingabeparameter verwendet wird, die ansonsten keine weiteren Parameter benötigt, kann das Interface ohne Namen direkt im Anschluss an den Funktionsnamen geschrieben werden. Dabei werden auch die runden Klammern der Funktion weggelassen. Dies ist in Abbildung 9.50 zu sehen.

```
1 fun interface Print{
2     fun Print(text: String)
3 }
4
5 fun Execute(p: Print){
6     p.Print("Hello World")
```

```

7 }
8
9 fun main() {
10     Execute{text->
11         println(text)
12     }
13     Execute{text->
14         println(text.reversed())
15     }
16 }
```

Codeausschnitt 9.50: Anonyme Interfaces

Wenn die Methode ebenfalls nur höchstens einen einzigen Parameter besitzt, so kann dieser ebenfalls weggelassen werden, und auf den Parameter wird dann mit dem Schlüsselwort *it* zugegriffen. Dies ist in Abbildung 9.51 gezeigt.

```

1 fun interface Print{
2     fun Print(text:String)
3 }
4
5 fun Execute(p:Print){
6     p.Print("Hello World")
7 }
8
9 fun main() {
10     Execute{
11         println(it)
12     }
13     Execute{
14         println(it.reversed())
15     }
16 }
```

Codeausschnitt 9.51: Zweite Vereinfachung des Anonymen Interfaces

```

1 fun interface Print{
2     fun Print(text:String)
3 }
4
5 fun Execute(p:Print){
6     p.Print("Hello World")
7 }
8
9 fun main() {
10     //Ohne SAM
11     Execute(object:Print{
12         override fun Print(text:String){
13             println(text)
14         }
15     })
16     //Mit SAM
17     Execute{
18         println(it.reversed())
19     }
20 }
```

Codeausschnitt 9.52: Gegenüberstellung mit und ohne SAM

In Abbildung 9.52 sind die Implementierungen mit und ohne SAM gegenübergestellt. Dabei ist deutlich zu sehen, wie sehr die Verkürzung der Übersichtlichkeit hilft.

9.22 Coroutines

Damit eine App benutzerfreundlich ist, ist eine wichtige Voraussetzung, dass der Nutzer zu jeder Zeit mit der App interagieren kann ohne auf eine Reaktion der App warten zu müssen. Wenn in einer App allerdings aufwendige Aufgaben gelöst werden müssen, kann dies auch mal länger dauern. Ein Beispiel wäre die Abfrage einer Datenbank, die Aufgrund der Netzverbindung und der eventuell großen Datenmengen längere Zeit in Anspruch nehmen kann. Würde diese Abfrage auf dem selben Thread wie die UI ausgeführt werden, so müsste der Nutzer solange warten, bis die Abfrage fertig ist, bis er mit der Bedienung der App fortfahren könnte. Um dieses Problem zu lösen, bietet Kotlin die Coroutines an. Diese lassen Code Asynchron zur UI ablaufen, wodurch diese nicht blockiert werden würde. Da das Thema zu Coroutines sehr umfangreich ist, wird hier nur der Teil erklärt, der auch in dem Projekt umgesetzt wurde.

9.22.1 Suspend

Damit eine Funktion in Kotlin asynchron zum Mainthread arbeiten kann, muss sie mit dem Schlüsselwort *suspend* versehen werden. Dies sorgt dafür, dass die Funktion gestartet, gestoppt und fortgesetzt werden kann[2]. Eine solche Funktion muss immer entweder in einem Coroutinescope, welches in Kapitel 9.22.2 näher erläutert wird, oder in einer weiteren Suspendfunction ausgeführt werden. Dies sorgt für Sicherheit, dass die Funktion den Mainthread nicht blockiert.

9.22.2 Coroutine Scope

Um eine Suspendfunction in einer synchronen Funktion aufzurufen, muss zuerst ein *Coroutinescope* definiert werden. Dieses kann ein globales Attribut einer Klasse sein, kann allerdings auch innerhalb einer Methode erzeugt werden.

Definition

Als globales Attribut kann das Scope wie in Beispiel 9.53 erstellt werden.

```
1 private val scope: CoroutineScope = CoroutineScope(CoroutineName
("Scope") + Dispatchers.IO)
```

Codeausschnitt 9.53: Erzeugung eines Coroutinescopes

Dabei ist der Name des Scopes frei wählbar. Innerhalb des Konstruktors der Klasse *CoroutineScope()* werden bestimmte Eigenschaften für das Scope festgelegt. In diesem Fall wird dem Scope ein Name gegeben und ein Dispatcher zugeordnet. Die Attribute müssen mit einem + voneinander getrennt werden.

Dispatcher

Der Dispatcher gibt an, in welcher Umgebung das Scope ausgeführt wird. Die drei Möglichkeiten sind

- Main
- IO

- Default

Wenn der Main-Dispatcher gewählt werden würde, würde der Code innerhalb des Scopes auf dem selben Thread wie die UI ausgeführt werden. Der IO und der Default-Dispatcher laufen beide Asynchron zur UI, dabei ist der IO-Dispatcher für Netzwerk- oder Laufwerklastige Aufgaben, der Default-Dispatcher für CPU-lastige Aufgaben ausgelegt.

Aufruf

Um eine Coroutine in einem Scope zu starten, muss dies mit der Methode *launch* des Coroutinescopes eingeleitet werden. Jeglicher Code, der im Block dieser Methode ausgeführt wird, läuft asynchron auf dem vordefinierten Dispatcher. Ein Beispiel ist in Abbildung 9.54 zu sehen.

```

1 fun main() {
2     scope.launch{
3         System.out.println("Coroutine 1 started")
4         System.out.println("Calculation 1 started\n")
5         HeavyCalculation(100000000)
6         System.out.println("Calculation 1 ended\n")
7     }.invokeOnCompletion(){
8         System.out.println("Coroutine 1 finished\n")
9     }
10    scope.launch{
11        System.out.println("Coroutine 2 started")
12        System.out.println("Calculation 2 started\n")
13        HeavyCalculation(1000000)
14        System.out.println("Calculation 2 ended\n")
15    }.invokeOnCompletion(){
16        System.out.println("Coroutine 2 finished\n")
17    }
18 }
19
20 suspend fun HeavyCalculation(n:Long){
21     var result = 0L
22     for(i in 0 until n){
23         result += i
24     }
25     System.out.println(result)
26 }
```

Codeausschnitt 9.54: Aufruf einer Coroutine

Die Reihenfolge der Ausgabe für Abbildung 9.54 ist:

- Coroutine 1 Started
- Calculation 1 Started
- Coroutine 2 Started
- Calculation 2 Started
- Result 2
- Calculation 2 Ended
- Coroutine 2 finished

- Result 1
- Calculation 1 Ended
- Coroutine 1 finished

Mit Hilfe der Funktion *invokeOnCompletion* kann Programmcode nach Beendigung der Corouine ausgeführt werden.

Kapitel 10

Weiterarbeit

Damit die App auch weiterhin gut strukturiert bleibt, werden in diesem Kapitel ein paar Regeln erläutert, an die sich gehalten werden sollte um weiterhin eine gute Weiterarbeit zu ermöglichen.

10.1 Farbthemen

Um neue Themen zu erstellen, müssen zuerst die notwendigen Farben definiert werden. Ein Thema besteht aus 11 Attributen, denen jeweils eine Farbe zugeordnet werden muss. Diese Attribute sind in folgender Aufzählung dargestellt.

- colorPrimary
- colorOnPrimary
- colorPrimaryDark
- colorOnPrimaryDark
- colorPrimaryLight
- colorOnPrimaryLight
- colorAccent
- colorOnAccent
- colorBackground
- colorOnBackground
- actionMenuTextColor

Zusätzlich zu diesen 11 Attributen gibt es auch noch weiter, welche allerdings für alle Themen identisch sind. Diese sind im *BaseTheme* festgelegt, ein neues Thema muss daher das *BaseTheme* als parent implementieren.

Um neue Farben zu definieren müssen diese in die beiden **colors.xml**-Dateien geschrieben werden. Dabei ist die eine für den Lightmode und die andere (night) für den Darkmode. Bei der Benennung kann sich gerne an den bestehenden orientiert werden. Falls dringend neue Attribute benötigt werden, können diese in

der **attr.xml**-Datei definiert werden. Diese müssten dann auch für jedes schon bestehende Thema initialisiert werden.

Innerhalb der UI darf ausschließlich auf die oben genannten Attribute zurückgegriffen werden. Dies geschieht über den Ausdruck `?attr/farbe`.

10.2 Strings

Wenn in der UI Text angezeigt werden soll, so ist dieser unbedingt in die **strings.xml**-Datei zu extrahieren. Dies ist notwendig um eventuell später mehrere Sprachen zur Verfügung zu stellen.

10.3 Bezeichnungen

Bei den Bezeichnungen für Klassen oder XML-Elemente kann gerne an dem bestehenden Schema festgehalten werden. Die Bezeichnungen sollten auf jeden Fall im Projekt eindeutig sein und grob beschreiben, was dahinter steckt.

10.4 MVVM

Um an dem Model-View-ViewModel-Pattern fast zu halten ist hier nochmal eine kurze Erklärung, was es zu beachten gilt.

10.4.1 Model

Bei dem Model handelt es sich um jegliche Zugriffe auf Daten. Dies können zum Beispiel die Datenbanken oder die Shared Preferences sein. In den sogenannten repositories werden die Datenzugriffe gebündelt. Es wird allerdings keine intelligente Logik implementiert sondern einfach nur ein simpler Zugriff. Die komplexere Logik, zum Beispiel das Hinzufügen von Kalendereinträgen bei Fvorisierung eines Eintrages, ist Teil des ViewModels.

10.4.2 ViewModel

In dem ViewModel ist die Logik hinter der App implementiert. Zum einen werden hier die Model-Zugriffe neu definiert, aber auch andere Logik die nicht direkt etwas mit der UI zu tun hat gehört ins ViewModel. Die Zugriffe auf das Model können in diesem Fall auch komplexer aussehen, es darf allerdings ausschließlich über die Repositories auf das Model zugegriffen werden.

10.4.3 View

in die View gehört ausschließlich die Logik zur Initialisierung und Verwaltung der UI-Elemente. Verboten in der View sind Zugriffe auf das Model, sowie Co-routines. Für beides soll das ViewModel verwendet werden.

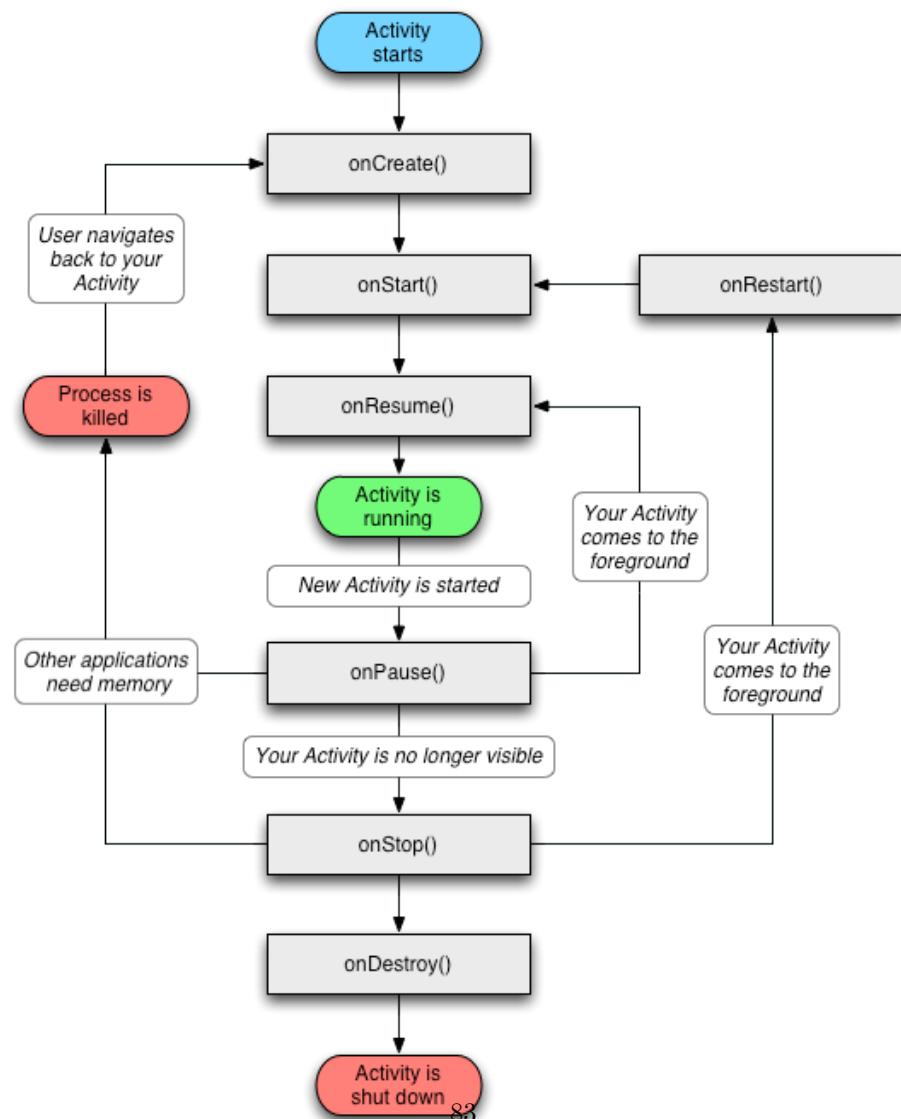
10.5 Dokumentation

Jede Methode, Klasse oder Parameter soll mit einem kurzen Kommentar versehen werden. Bei Methoden und Klassen sollte ebenfalls eine kurze Erklärung zu den Ein- und Ausgabeparametern, sowie die Appversion, in der die Methode oder Klasse erstellt wurde. Zusätzlich kann auch der Autor mit angegeben werden und es können auch weitere Verlinkungen hinzugefügt werden. Dabei kann sich an den bestehenden orientiert werden.

Kapitel 11

Lifecycles

11.1 Activity-Lifecycle



11.2 Fragment-Lifecycle

