МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное

образовательное учреждение высшего образования

«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Инженерно-технологическая академия

# Л. В. ПИРСКАЯ

РАЗРАБОТКА СОВРЕМЕННЫХ МОБИЛЬНЫХ ПРИЛОЖЕНИЙ ДЛЯ ОС ANDROID

*Учебное пособие*

Ростов-на-Дону – Таганрог Издательство Южного федерального университета

2020

1

УДК 004.451.9(075.8) ББК 32.973-018.2я73

*Содержание*

П337

*Печатается по решению кафедры математического обеспечения и применения ЭВМ Института компьютерных технологий*

*и информационной безопасности Южного федерального университета (протокол № 5 от 11 февраля 2020 г.)*

**Рецензенты:**

генеральный директор ООО «Оджетто веб» *С. А. Друппов*

генеральный директор ООО «Иностудио Солюшинс» *М. В. Болотов*

**Пирская, Л. В.**

П337 Разработка современных мобильных приложений для ОС Android : учебное пособие / Л. В. Пирская ; Южный федеральный универси- тет. – Ростов-на-Дону ; Таганрог : Издательство Южного федерального университета, 2020. – 116 с.

ISBN 978-5-9275-3700-6

Учебное пособие «Разработка современных мобильных приложений для ОС Android» представляет актуальный материал для продвинутого уров- ня разработки под Android, сопровождающийся примерами на двух языках программирования Kotlin и Java. Пособие охватывает базовые понятия о раз- работке на языке Kotlin, работу с сетью и базой данных с использованием современных библиотек, создание правильной архитектуры приложения, те- стирование приложения и API.

Пособие предназначено для студентов направлений подготовки бака- лавриата 09.03.04 «Программная инженерия», 02.03.03 «Математическое обеспечение и администрирование информационных систем и магистрату- ры» 09.04.04 «Программная инженерия» Института компьютерных техноло- гий и информационной безопасности. Также учебное пособие может быть полезно для студентов технических направлений подготовки, связанных с разработкой программного обеспечения для мобильных устройств.

УДК 004.451.9(075.8) ББК 32.973-018.2я73

ISBN 978-5-9275-3700-6

© Южный федеральный университет, 2020

© Пирская Л. В., 2020

© Оформление. Макет. Издательство Южного федерального университета, 2020

2

# СОДЕРЖАНИЕ

*Содержание*

[ВВЕДЕНИЕ …………………………………………………………… 7](#_TOC_250100)

1. РАЗРАБОТКА МОБИЛЬНЫХ ПРИЛОЖЕНИЙ НА ЯЗЫКЕ

KOTLIN В ANDROID ……………………………………………….. 9

* 1. [О Kotlin ………………………………………………………….. 9](#_TOC_250099)
	2. [Основы Kotlin 10](#_TOC_250098)
	3. [Kotlin в Android 13](#_TOC_250097)
		1. [*Первый проект на Kotlin* 14](#_TOC_250096)
		2. [*Лямбда-выражения* 15](#_TOC_250095)
		3. [*Kotlin-расширения (KTX)* 18](#_TOC_250094)
	4. [Безопасность в Kotlin 19](#_TOC_250093)
		1. [*Nullable-типы* 19](#_TOC_250092)
		2. [*Безопасные вызовы* 20](#_TOC_250091)
		3. [*Elvis-оператор* 20](#_TOC_250090)
		4. [*Оператор NPE* 21](#_TOC_250089)
	5. [Функции и лямбды 21](#_TOC_250088)
		1. [*Аргументы по умолчанию* 22](#_TOC_250087)
		2. [*Локальные функции и замыкания* 22](#_TOC_250086)
		3. [*Функции-расширения* 22](#_TOC_250085)
		4. [*Лямбды* 23](#_TOC_250084)
	6. [Операции над коллекциями 25](#_TOC_250083)
		1. [*Трансформация* 25](#_TOC_250082)
		2. [*Фильтрация* 26](#_TOC_250081)
		3. [*Другие операции* 26](#_TOC_250080)
	7. [Функции среды 26](#_TOC_250079)
		1. [*Функция let* 27](#_TOC_250078)
		2. [*Функция apply* 27](#_TOC_250077)
		3. [*Функция with* 28](#_TOC_250076)
		4. [*Функция also* 28](#_TOC_250075)
	8. [Идиомы 28](#_TOC_250074)
		1. [*Поиск в коллекции* 28](#_TOC_250073)
		2. [*Kotlin-синглтон* 29](#_TOC_250072)
		3. [*Проверка типа* 29](#_TOC_250071)
	9. [Многопоточность и сопрограммы (coroutines) 29](#_TOC_250070)
		1. [*Запуск корутин (launch и async)* 30](#_TOC_250069)
		2. [*Прерываемые функции* 32](#_TOC_250068)
		3. [*Контекст вызова* 34](#_TOC_250067)

[Контрольные вопросы 35](#_TOC_250066)

1. [РАБОТА С СЕТЬЮ (RETROFIT) 37](#_TOC_250065)
	1. [Инициализация зависимостей 37](#_TOC_250064)
	2. [Конфигурация Retrofit-интерфейса 37](#_TOC_250063)
	3. [Конфигурация Retrofit-объекта 38](#_TOC_250062)
	4. [Инстанцирование Retrofit-сервиса 39](#_TOC_250061)
	5. [Добавление конвертеров 40](#_TOC_250060)
	6. [Выполнение веб-запросов с помощью Retrofit 40](#_TOC_250059)

[Контрольные вопросы 41](#_TOC_250058)

1. [БАЗА ДАННЫХ (ROOM) 43](#_TOC_250057)
	1. [Начало работы с Room 43](#_TOC_250056)
		1. [*Настройка зависимостей для Kotlin* 43](#_TOC_250055)
		2. [*Настройка зависимостей для Java* 44](#_TOC_250054)
	2. [Создание базы данных 45](#_TOC_250053)
		1. [*Определение сущностей* 45](#_TOC_250052)
		2. [*Инициализация объекта доступа к данным* 47](#_TOC_250051)
		3. [*Инициализация базы данных* 49](#_TOC_250050)
	3. [Использование базы данных 51](#_TOC_250049)
		1. [*Использование паттерна «Репозиторий»* 51](#_TOC_250048)
		2. [*Использование паттерна «ViewModel»* 53](#_TOC_250047)
	4. [Кэширование с Room 54](#_TOC_250046)
		1. [*Реализация логики кэширования* 55](#_TOC_250045)

[Контрольные вопросы 58](#_TOC_250044)

1. [РАБОТА С POSTMAN 60](#_TOC_250043)
	1. [Установка Postman 60](#_TOC_250042)
	2. [Начало работы с Postman 61](#_TOC_250041)
	3. [Отправляем первый запрос через Postman 62](#_TOC_250040)
		1. [*Тело запроса в Postman* 63](#_TOC_250039)
	4. [Настройка среды Postman. Сохранение запросов в коллекции 64](#_TOC_250038)

[Контрольные вопросы 67](#_TOC_250037)

4

*Содержание*

1. [УВЕДОМЛЕНИЯ 68](#_TOC_250036)
	1. [Создание уведомления 68](#_TOC_250035)
	2. [Показ уведомления 69](#_TOC_250034)
	3. [Каналы уведомлений 70](#_TOC_250033)
	4. [Взаимодействие пользователя с уведомлением 71](#_TOC_250032)

[Контрольные вопросы 72](#_TOC_250031)

1. [АРХИТЕКТУРНЫЕ КОМПОНЕНТЫ ANDROID 74](#_TOC_250030)
	1. [ViewModel 74](#_TOC_250029)
		1. [*Использование ViewModel на практике* 76](#_TOC_250028)
	2. [LiveData 79](#_TOC_250027)
		1. [*Использование LiveData* 79](#_TOC_250026)
		2. [*Инициализация LiveData во ViewModel* 80](#_TOC_250025)
		3. [*Подписка на LiveData во View* 80](#_TOC_250024)
		4. [*Обновление данных LiveData* 82](#_TOC_250023)
	3. [View Binding 82](#_TOC_250022)
		1. [*Настройка установки* 83](#_TOC_250021)
		2. [*Использование View Binding* 84](#_TOC_250020)
		3. [*Использование View Binding в Activity* 84](#_TOC_250019)
		4. [*Использование View Binding в фрагментах* 85](#_TOC_250018)
		5. [*Сравнение View Binding с Data Binding* 87](#_TOC_250017)
	4. [Data Binding 88](#_TOC_250016)
		1. [*Макеты и выражения binding* 89](#_TOC_250015)
		2. [*Объекты данных* 90](#_TOC_250014)
		3. [*Связывание данных* 91](#_TOC_250013)
		4. [*Обработка событий* 92](#_TOC_250012)
			1. *Ссылки на метод* 92
			2. *Привязки слушателя* 94
		5. [*Генерирование binding классов* 96](#_TOC_250011)

[Контрольные вопросы 97](#_TOC_250010)

1. [ТЕСТИРОВАНИЕ В ANDROID 99](#_TOC_250009)
	1. [Unit-тестирование 99](#_TOC_250008)
		1. [*Разбор простейшего теста* 99](#_TOC_250007)
		2. [*Реализация первого Unit-теста* 101](#_TOC_250006)
		3. [*Mockito и «моки»* 103](#_TOC_250005)
	2. [UI-тестирование 106](#_TOC_250004)

6

* + 1. [*Реализация простейшего UI-теста* 107](#_TOC_250003)

[Контрольные вопросы 110](#_TOC_250002)

[ЗАКЛЮЧЕНИЕ 111](#_TOC_250001)

[СПИСОК ЛИТЕРАТУРЫ 112](#_TOC_250000)

# ВВЕДЕНИЕ

*Содержание*

В настоящее время область разработки мобильных приложений для ОС Android находится на пике актуальности и популярности. Сегодня мобильные разработчики востребованы в разных областях текущих реа- лий жизни: корпоративные приложения имеют мобильные версии, СМИ имеют мобильные приложения, бизнес активно переводит свои програм- мы лояльности в приложения и т.д. Поэтому мобильные разработчики нужны абсолютно в разных сферах: мобильные игры и развлекательные приложения, развлекательно-образовательные приложения, банковские приложения, приложения электронной коммерции и т.д.

Стек технологий мобильной разработки активно меняется. Основные мобильные платформы постоянно обновляют стек и развивают его. Поэто- му обновлять учебные материалы в данной области требуется постоянно.

Данное учебное пособие является продолжением и дополнением изданного ранее учебного пособия «Разработка мобильных приложений в среде Android Studio» [1], в котором были рассмотрены базовые темы для начала разработки на Android: работа в среде разработки AndroidStudio, разработка интерфейса мобильного приложения, работа с ресурсами при- ложения, организация данных в виде списка, сетевое взаимодействие, ра- бота с JSON-файлами, реализация базы данных в системе Android, осо- бенности организации современного интерфейса мобильного приложения. В текущем учебном пособии представлены абсолютно новые и не пересекающиеся темы с имеющимися в описанном выше пособии [1], а именно, создание мобильных приложений на языке Kotlin, работа с сетью и базой данных с использованием современных библиотек Retrofit и Room, создание правильной архитектуры приложения, тестирование при- ложения и API (начало работы с Postman), создание уведомлений (ноти- фикаций). Представленные материалы сопровождаются примерами на двух языках с описанием их реализации: сначала представлен пример на

Kotlin, следом за ним – пример на Java.

Данное учебное пособие разрабатывалось для студентов направле- ний подготовки бакалавриата 09.03.04 «Программная инженерия», изуча- ющих дисциплину «Разработка мобильных приложений», а также маги- стратуры 09.04.04 «Программная инженерия», изучающих дисциплину

«Программирование для мобильных платформ». Материалы, представ- ленные в учебном пособии, помогут облегчить разработку с помощью использования специализированных библиотек и создать надежные, те- стируемые и легкие в поддержке приложения за счет выстраивания гра- мотной архитектуры. Разработанное учебное пособие будет полезно для подготовки студентов к участию в чемпионате WorldSkills по компетен- ции «Разработка мобильных приложений».

8

# РАЗРАБОТКА МОБИЛЬНЫХ ПРИЛОЖЕНИЙ НА ЯЗЫКЕ KOTLIN В ANDROID

*1.2. Основы Kotlin*

## О Kotlin

Kotlin – это новый кроссплатформенный язык программирования, разработанный компанией JetBrains, преимущественно в санкт-петербург- ском офисе компании, и вышедший в релиз в 2016 г.

Kotlin [1] – лаконичный, безопасный и прагматичный язык, совме- стимый преимущественно с Java, а также имеющий совместимость с JavaScript и возможность нативной компиляции. Используется практически везде, где применяется Java: серверные приложения, Android-приложения, десктопные приложения на базе популярнейших Java-библиотек и многое другое. Работает с наследием Java в виде огромного количества фреймвор- ков и библиотек, не уступая в производительности.

Kotlin был представлен как основной язык для Android-разра- ботки на конференции «Google I/O» в 2019 г., так как обладает рядом преимуществ:

* **краткость.** В Kotlin значительно уменьшено количество шаб- лонного кода. Например, так будет выглядеть класс, который в Java мож- но было бы представить как POJO:

data class Student(val status: String, var name: String = "Android")

Так в рамках виртуальной машины Java (JVM) у представленного выше класса будут переопределены все геттеры и сеттеры, а также неко- торые шаблонные методы, такие как hashCode(), toString() и equals().

* **читаемость.** Благодаря своей лаконичности и идиоматично- сти, Kotlin легко читается даже без особых знаний о языке, образуя чело- векочитаемые языковые конструкции: «When this is string, do something with string» (в дословном переводе с английского языка – «когда это стро- ка, сделай что-нибудь со строкой»):

element.apply { when (this) {

is Int -> doSomethingWithInt()

is String -> doSomethigWithString() else -> doSomethingElse()

}

}

* + **поддержка Kotlin в современных библиотеках**, в том числе в Android Jetpack. Многие преимущества Kotlin, например, корутины функ- ции расширения, лямбда-выражения, поддерживаются в современных библиотеках для Android.
	+ **взаимная совместимость с Java.** Kotlin можно использовать вместе с Java в одном проекте без необходимости переводить весь код про- граммы на Kotlin. Например, если в проекте много старых POJO-классов, то имеется доступ к ним из Kotlin, причем с Kotlin-синтаксисом, т.е. можно использовать синтаксис доступа к полям вместо сеттеров и геттеров.

intent.action = "ACTION" intent.setAction("ACTION")

* + **мультиплатформенность.** Kotlin можно использовать не только для разработки под Android, но также для iOS, бэкенда и веб- приложений. Отделяя логику от представления, можно использовать оди- наковый код в разных проектах.
	+ **безопасность кода.** Kotlin позволяет избавиться от популяр- нейшей ошибки NullPointerException, известной как «Ошибка на милли- ард долларов».

var string: String

string = null // Ошибка компиляции var string: String?

string = null // Всё ок

## Основы Kotlin

Синтаксис Kotlin схож с языками программирования, ориентиро- ванными на ООП. По своей структуре он подобен Java, модифицирован- ный исходя из потребностей современного разработчика.

Разберем базовый синтаксис на примере определения простого класса:

class Learning {

val str: String = "Hello"

var i: Int = 0

fun hello() { print("$str World")

}

fun sum(x: Int, y: Int): Int { return x + y

}

fun maxOf(a: Float, b: Float) = if (a > b) a else b

}

Определение класса выглядит следующим образом:

class Learning

С определением атрибута уже немного сложнее:

val str: String = "Hello"

Здесь появляется ключевое слово «val» (value, то есть «значение»), что означает «неизменяемое» [2]. При попытке записать значение в пере- менную str, компилятор выдаст ошибку. За ним идёт имя переменной, а после двоеточия – тип. В данном примере писать «String» необязательно, поскольку компилятор сам определит тип переменной после того, как пе- редадите туда строку. Таким образом представленную выше строку мож- но сократить до

val str = "Hello"

В примере в строчке ниже «*var i: Int = 0*» записывается целочис- ленное значение и данную переменную уже можно перезаписать. Ключе- вое слово «var» определяется как «variable», т.е. изменяемое. Это выраже- ние также можно сократить. Передавая в переменную «0», вы сообщаете компилятору о том, что эта переменная целочисленная (Int).

Функции в Kotlin декларируются ключевым словом «fun».

fun hello() { print("$str World")

}

Функция *hello()* ничего не возвращает, в ней только вызывается си- стемная функция *print()* (ближайший аналог *System.out.print()* в Java), ко-

торая выводит в консоль фразу «Hello World» с помощью интерполяции строк [3]. Не обязательно указывать компилятору, что функция ничего не возвращает. Если для разработчика всё-таки имеется необходимость, то можно написать, что она возвращает тип *Unit* [4] (аналог void в Java и не- которых других языках программирования):

fun hello(): Unit { print("$str World")

}

Функция *sum*, возвращает простое сложение двух входных пара- метров. Параметры передаются в формате *имя: Тип*, ключевые слова *var* или *val* здесь не требуются, поскольку параметры всегда передаются в виде неизменяемых значений (обратите внимание, что если не можете изменить значение переменной, это не означает, что не можете модифи- цировать объект).

fun sum(x: Int, y: Int): Int { return x + y

}

Функция *maxOf* демонстрирует сразу две синтаксических специфи- ки языка – «модифицированные» тернарные операторы и простое возвра- щаемое значение. Не обязательно открывать блок функции, если её логика умещается в одну строку. Например, представленную в примере функцию sum можно было бы написать иначе:

fun sum(x: Int, y: Int) = x + y

Также и в функции *maxOf* не обязательно указывать тип возвраща- емого значения, поскольку компилятор сам понимает, какое значение он должен вернуть.

Отсутствие тернарного оператора считают большой проблемой синтаксиса Kotlin, но на это есть свои причины. Наверное, уже обратили внимание, что знак двоеточия используется в Kotlin только в случаях с указанием типа. Это относится не только к функциям и переменным, но и к классам. Если бы класс *Learning* требовалось унаследовать, например, от класса *Start*, то было бы написано следующее:

class Learning : Start()

Даже если выделяется память под объект, реализующий интер- фейс (что часто приходится делать в Java на Android, там таким образом часто реализуются callback-операции), фактически пишется, что требу- ется объект, реализующий определенный интерфейс. Ниже представлен пример того, как бы выглядела функция установки слушателя на нажа- тие кнопки:

val btn = findViewById<Button>(R.id.button) btn.setOnClickListener(object : View.OnClickListener {

override fun onClick(v: View?) { TODO("Not yet implemented")

}

})

На самом деле, это всего лишь пример и интерфейсу подобного ро- да есть замена в Kotlin, которую мы рассмотрим позже. Тем не менее, как вы можете увидеть, в Kotlin нет ключевого слова *new*. Оно не требуется для компилятора, чтобы он понял, что выделяется память под объект. Ес- ли бы требовалось выделить память под созданный ранее объект Learning, то это было бы записано так:

val learning = Learning()

Таким образом и под объект, который реализует интерфейс *View.OnClickListener*, выделяем память без ключевого слова *new.* Важно понять, что ключевое слово object не выделяет память под объект, а опре- деляет его [5]. Рекомендуется поэкспериментировать, чтобы не допускать ошибок в будущем, особенно при работе с наследием Java.

Подробнее про синтаксис языка и его устройство можно прочитать в официальной документации [1], книгах [6] или официальной докумен- тации от Android-разработчиков [7].

Далее будут рассмотрены более сложные примеры на практике и их сравнение с Java.

## Kotlin в Android

В данном подразделе будут рассмотрены некоторые примеры зна- комых конструкций в Android на Kotlin.

### *Первый проект на Kotlin*

Создание базового проекта на Kotlin ничем не отличается от созда- ния проекта на Java кроме того, что в поле “Language” выбираем Kotlin (рис. 1). Ниже представлено создание проекта с шаблона “Empty Activity”.



**Рис. 1.** Создание проекта в Android Studio

Среда разработки (IDE) создает привычную структуру папок, за ис- ключением того, что сгенерированный файл входной activity называется не *MainActivity.java*, а *MainActivity.kt*.

Рассмотрим код, сгенерированный в файле MainActivity.kt:

class MainActivity : AppCompatActivity() {

override fun onCreate(savedInstanceState: Bundle?) { super.onCreate(savedInstanceState) setContentView(R.layout.activity\_main)

}

}

Определение класса, как и в Java, должно совпадать с именем файла:

class MainActivity : AppCompatActivity()

Знак двоеточия обозначает, что класс наследуется от другого или ре- ализует какой-либо интерфейс. Как определить, что является классом, а что интерфейсом без ключевых слов *extends* и *implements*? В данном примере происходит наследование от класса *AppCompatActivity*, это известный вам Java-класс (в первой же строчке пример абсолютной совместимости Java и Kotlin). Если бы в нашем классе требовалось реализовать интерфейс слуша- теля кнопки View.OnClickListener, то код бы выглядел следующим образом:

class MainActivity : AppCompatActivity(), View.OnClickListener

Отличие от Java в наличии круглых скобок после имени класса. Ес- ли класс имеет обязательный конструктор с параметрами, то вместо вызо- ва super в конструкторе класса, помещаете значение в эти скобки (это не значит, что нельзя пользоваться ключевым словом *super*).

Представленный класс содержит единственный метод onCreate, те- ло которого абсолютно не отличается от эквивалента на Java (за исключе- нием наличия точки с запятой в последнем):

override fun onCreate(savedInstanceState: Bundle?)

Две отличительных особенности, которые можно здесь увидеть: ключевое слово *override* и вопросительный знак после объявления типа *Bundle*. Первое – это аналог аннотации *@Override* в Java и работает точно так же, за исключением того, что в отличие от Java, ключевое слово *over- ride* в Kotlin обязательно.

С вопросительным знаком немного сложнее. Если вы знакомы только с Java, то для вас это, возможно, абсолютно новая концепция. В подразд. 1.1 говорилось, что безопасность – одно из ключевых преиму- ществ языка. На данный момент стоит сказать, что декларация типа «Bun- dle?» говорит о том, что этот входной параметр может принимать значе- ние null. Дополнительную информацию по данному вопросу можно по- смотреть в официальной документации [8].

### *Лямбда-выражения*

Лямбда-выражения часто используются, когда вы пишете на Kotlin. Настолько часто, что познакомимся с ними на примере обработки нажа- тия на кнопку.

В проекте в *activity\_main.xml* удалим всё внутри сгенерированного

*ConstraintLayout* и добавим обычную кнопку:

<Button

android:id="@+id/buttonLearn" android:layout\_width="wrap\_content" android:layout\_height="wrap\_content" android:text="Выучить Kotlin" app:layout\_constraintBottom\_toBottomOf="parent" app:layout\_constraintLeft\_toLeftOf="parent" app:layout\_constraintRight\_toRightOf="parent" app:layout\_constraintTop\_toTopOf="parent" />

Обратите внимание, что было добавлено имя id в стиле Camel Case. Вспомним, как бы выглядел этот код на Java [9]:

@Override

protected void onCreate(@Nullable Bundle savedInstanceState) { super.onCreate(savedInstanceState); setContentView(R.layout.activity\_main);

Button button = findViewById(R.id.buttonLearn); button.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) { Toast.makeText(MainJavaActivity.this, "Выучили",

Toast.LENGTH\_SHORT).show();

}

});

}

Из примера для Java видно, что обычно кнопку можно найти по id с помощью метода findViewById и «повесить» на неё слушателя с помощью setOnClickListener, передавая в качестве параметра объект, реализующий интерфейс View.OnClickListener. Перегруженный метод onClick в данном случае работает как функция обратного вызова (callback). В этот callback вставляем показ Toast.

Что же здесь не так? Слишком много boilerplate-кода для простого нажатия на кнопку. На Kotlin получается гораздо меньше кода, используя буквально тот же самый метод:

override fun onCreate(savedInstanceState: Bundle?) { super.onCreate(savedInstanceState)

setContentView(R.layout.activity\_main)

buttonLearn.setOnClickListener { Toast.makeText(this@MainActivity, "Выучили",

Toast.LENGTH\_SHORT).show()

}

}

Данный код делает то же самое, что и Java-аналог.

Куда же делся громоздкий интерфейс? Создатели Kotlin увидели, что Java-программисты часто используют такую конструкцию для обыч- ного callback. Помимо того, что этот способ содержит много кода, он так- же включает своеобразную тавтологию. Программисту, который читает код, достаточно посмотреть на название метода setOnClickListener, чтобы понять, что делает код. И интерфейс OnClickListener, и onClick – это абсо- лютно бессмысленные конструкции с точки зрения читаемости кода.

Важно заметить, что такое преобразование относится только к сов- местимости с Java. Если встретитесь с подобной конструкцией, написан- ной на Kotlin, конвертировать это в лямба-выражение уже не получится.

Обратите внимание, что функция вызывается без скобок:

buttonLearn.setOnClickListener { Toast.makeText(this@MainActivity, "Выучили",

Toast.LENGTH\_SHORT).show()

}

Этот синтаксис относится только к **функциям высшего порядка**, которые принимают только одно лямбда-выражение. Функции высшего порядка – функции, которые принимают в качестве параметра или воз- вращают функцию.

Вызовем данную функцию следующим образом:

buttonLearn.setOnClickListener({ Toast.makeText(this@MainActivity, "Выучили",

Toast.LENGTH\_SHORT).show()

})

То есть функция *setOnClickListener* принимает единственный пара- метр типа *(View) -> Unit.*

Подробнее про лямбда-выражения и функции высшего порядка можно ознакомиться в официальной документации [10].

### *Kotlin-расширения (KTX)*

В подразд. 1.3.2 в примере с кнопкой отсутсвовал *findViewById* бла- годаря библиотеки Android KTX.

Android Kotlin Extensions (Kotlin-расширения для Android) [11] – библиотека (часть пакета Android Jetpack), созданная командой разработ- чиков Android, которая включает в себя огромное количество расширений для Kotlin, упрощающих написание кода.

В случае с кодом, который был написан ранее, используется рас- ширение *Synthetic Layout*, которое позволяет обращаться к id напрямую, минуя *findViewById*:

buttonLearn.setOnClickListener { Toast.makeText(this@MainActivity, "Выучили",

Toast.LENGTH\_SHORT).show()

}

Если в созданном своем проекте вы поднимитесь чуть выше, то в списке импортов увидите, что добавилось строка:

import kotlinx.android.synthetic.main.activity\_main.\*

Существует и работает это благодаря тому, что для проектов на Ko- tlin Android Studio добавляет в Gradle нужные библиотеки. Для того, что- бы внедрить KTX в проект, в котором его по какой-либо причине нет, добавьте следующий плагин в build-gradle модуля app:

apply plugin: 'kotlin-android-extensions'

А также представленную ниже библиотеку в список зависимостей:

implementation 'androidx.core:core-ktx:1.2.0'

Подробнее про использование и содержимое KTX можно прочи- тать в официальной документации Android [11]. Стоит отметить, что это не единственный способ избавиться от *findViewById* и вечной проблемы с обращением к компонентам *View*. Существует также библиотека *Data Binding* [12], которая относится к тому же сборнику, что и KTX, а также

«свежий» *View Binding* [13], который уже сейчас называют практически революцией в Android-разработке.

## Безопасность в Kotlin

Встречая в коде программы знаки «?» или «!!», вы имеете дело с кон- цепцией *Null Safety* (можно перевести это как «защита от null-значений»).

Зачем нужна эта безопасность? С вероятностью в 100 % процентов, разрабатывая приложения на Java, вы сталкиваетесь с исключением *Null- PointerException*. Поэтому короткий ответ на этот вопрос – избежать *NPE* и, как следствие, «краша» программы.

Благодаря Null Safety не требуется делать постоянные проверки в коде и надеяться на то, что автор какой-либо библиотеки не допустил ошибку и не вернется значение null. На Kotlin исключение *NullPointerEx- ception* может появиться лишь в нескольких случаях:

* преднамеренный вызов исключения с помощью конструкции throw NullPointerException();
* использование оператора «!!», о котором говорится ниже;
* утечки, связанные с инициализацией (часто относится к мно- гопоточным приложениям);
* совместимость с Java (не любой код можно «безболезненно» вытащить из Java).

Важно отметить, что представленные выше 4 случая программист может контролировать.

### *Nullable-типы*

Рассмотрим пример, представленный в подразд. 1.2.

var a: String = "abc" a = null

Kotlin не позволит скомпилировать представленный выше код. По умолчанию, типы в Kotlin не могут принимать null значение. Если воз- вращаемое значение используется без обозначений «?» или «!», то обра- щаться к этим данным возможно без лишних проверок.

Чтобы записать значение null в переменную, используется тип с обозначением «?», т.е. в нашем случае «String?»:

var a: String? = "abc" a = null

Знак вопроса всего лишь показывает программисту, что данное значение может быть null. Дальше он сам решает, что с ним делать.

У Kotlin есть ещё одно свойство, которое помогает не думать о том, что делать с null-значениями – **безопасные вызовы**.

### *Безопасные вызовы*

Оператор безопасного вызова выглядит как «?.», а используется следующим образом:

val a = "Kotlin"

val b: String? = null println(b?.length) println(a?.length)

В приведенном выше коде переменная *b* принимает «опасное» зна- чение. Оператор «?.» может читаться как «выполнить, если не null», т.е. команда в этой строке просто не выполнится:

println(b?.length)

Переменная *a* принимает безопасное значение «String». Если ис- пользуется оператор безопасного вызова, то код выполнится, но компиля- тор нам сообщит о том, что он здесь лишний.

Есть ли другие способы? Конечно, можно выполнить проверку, например:

if (b != null) println(b.length)

Таким образом, безопасный вызов не требуется. Kotlin понимает, что переменная *b* не может принять значение *null* после этой проверки и понимает её тип в рамках блока *if* как обычный безопасный «String».

Обратите внимание, что это не относится к изменяемым перемен- ным, поскольку, очевидно, их значение может измениться в любой мо- мент времени.

### *Elvis-оператор*

На практике elvis-оператор валидно использовать для отображения в UI. Если не пришли какие-то данные, то нельзя просто написать пользо- вателю «null».

Предположим, сделан какой-то запрос к серверу (а лучше всегда помечать модели, которые работают с внешними данными как *Nullable,*

для более гибкой обработки в приложении), а вместо имени пользователя пришел null. В таком случае, можно передать переменную с именем поль- зователя в следующем виде:

val username = response?.username ?: “Неизвестный пользователь”

Elvis-оператор обозначается как «?:» и заменяет конструкцию:

val username = if (response?.username) response.username else “Неизвестный пользователь”

Таким образом, вы можете гарантировать, что во View вернутся ка- кие-то данные, даже если эти данные обозначаются как ошибка (а это го- раздо лучше потенциального NPE).

### *Оператор NPE*

Обычно оператор «!!» используется для совместимости с Java. Дело в том, что Java не знает о существовании null-безопасности, поэтому по умолчанию Kotlin видит Java-типы как «тип!».

Ни в коем случае не используйте этот оператор, если не уверены, что его можно использовать.

Пример валидного использования в Android:

arguments?.getString("SOME\_STRING", "Дефолтная строка")!!

В примере представлено получение аргумента из Bundle. Возможно использовать здесь «!!», в случае, если будет null, то вернется значение из второго параметра (о чем не знает Kotlin). В этом же примере можно ис- пользовать elvis:

arguments?.getString("SOME\_STRING", "Дефолтная строка") ?: “”

Правая часть выражения никогда не сработает, потому что метод getString не может вернуть null.

## Функции и лямбды

Kotlin предоставляет достаточно гибкую работу с функциями. В данном подразделе не планируется углубления в специфику работы функций, а будут рассмотрены некоторые интересные особенности, кото- рые помогут сэкономить время разработчику.

### *Аргументы по умолчанию*

Предположим, имеется функция, которая в качестве аргументов принимает информацию о студенте. Известно, что у большей части сту- дентов в России – российское гражданство, поэтому функция может вы- глядеть следующим образом:

fun setStudentParams(name: String, surname: String, citizenship: String = “Rus- sian”) {

…

}

Таким образом, в функции для большинства студентов возможно передать только два первых аргумента:

setStudentParams(“Mark”, “Abramenko”)

Если захотим добавить параметр, то просто передадим третий ар- гумент:

setStudentParams(“Vahtang”, “Darbinyan”, “Armenian”)

мер,

### *Локальные функции и замыкания*

Функции в Kotlin могут быть вложены в другие функции. Напри-

fun calcSumWithY(x: Int): Int { val y = 5

fun sum(): Int { return x + y

}

return sum()

}

Функция *calcSumWithY* считает обычную сумму, но использует в каче- стве второго слагаемого константное значение *Y*, которое задекларировано внутри этой функции. Возвращает значение она при помощи вложенной функции *sum*, которая не принимает ни одного значения, но при этом считает *x* и *y*. В таком случае переменная *y* находится в **замыкании** (closure).

### *Функции-расширения*

Функции-расширения [14] используются для расширения функцио- нала существующего класса.

Например, класс String. Предположим, заказчик потребовал, чтобы мы срочно реализовали добавление слова «тащемта» к каждой входящей строке. Возможно это сделать следующим образом:

fun appendUselessWordToString(s: String): String { return "$s, тащемта"

}

И использовать потом эту функцию так:

appendUselessWordToString(“Паук”) // Вывод: «Паук, тащемта»

Kotlin позволяет расширить функционал класса String, поэтому можно записать по иному:

fun String.toTashemta(): String = "$this, тащемта"

…

“Паук”.toTashemta() // Так выглядит вызов этой функции

То есть можно вызвать эту функцию так, будто она принадлежит классу String. Сам параметр расширяемого типа определяется внутри функции как «this».

### *Лямбды*

Kotlin-функции – объекты первого класса. Программисты, которые разрабатывают на Kotlin, активно пользуются этим, поэтому достаточно тяжело встретить библиотеку, которая не использовала бы лямбды и функции высшего порядка. Полный функционал лямбд представлен в официальной документации [10].

Лямбды в Kotlin имеют специальный тип, который может быть представлен как *«(arg: Type) -> ReturnType»:*

(Int) -> String // функция принимает один параметр типа Int и возвращает String

(Int, Int, Int) -> String // принимает три Int и возвращает String () -> Int // не принимает ни одного параметра и возвращает Int () -> Unit // ничего не принимает и ничего не возвращает

В виде классических функций это можно написать в виде

fun (x: Int): String

fun (x: Int, y: Int, z: Int): String fun (): Int

fun ()

В подразд. 1.3.2 рассматривался пример со слушателем на кнопке:

buttonLearn.setOnClickListener { Toast.makeText(this@MainActivity, "Выучили",

Toast.LENGTH\_SHORT).show()

}

В представленном выше примере примере функция *setOnClick- Listener* является функцией высшего порядка, поскольку она принимает другую функцию в качестве аргумента.

Самый простой пример лямбда-выражения:

val lambda = {

println(“Наша лямбда”)

}

Из примера видно, что можно использовать переменную для хра- нения лямбды. Здесь не указан явно её тип, но машина воспримет его как *() -> Unit*, т.е. функцию без входных параметров и возвращаемого значе- ния. Вызывается она так же, как обычная функция:

… lambda()

…

Рассмотрим пример сложнее:

val hof: (() -> Unit) -> String = { it()

"Функция завершила работу"

}

Данная функция принимает в качестве аргумента еще одну функ- цию, а возвращает строку. По умолчанию имя единственного аргумента в функции обозначается как «it», но можно и явно задать имя входного па- раметра:

val hof: (() -> Unit) -> String = { callback ->

Таким образом, «callback» – имя единственного аргумента. Последняя строчка в теле лямбды – возвращаемое значение. Функция будет выполняться так: сначала вызовется функция, которую передали в качестве аргумента, затем функция hof вернет значение «Функция завершила работу».

Для наглядности попробуем вызвать функцию *hoc* и передать туда функцию lambda:

println(hof(lambda))

Выполнение этой строчки даст следующий результат:

Наша лямбда

Функция завершила работу

Не имеет смысла держать такую функцию, как lambda в отдель- ной переменной, достаточно передать безымянную функцию в качестве аргумента:

println(hof { println(“Выполнилась наша безымянная функция”) })

## Операции над коллекциями

Внедрение функций высшего порядка привнесло в Kotlin много хо- роших вещей из функционального программирования. В стандартной библиотеке Kotlin есть огромное количество перебирающих функций.

### *Трансформация*

Функция map, доступная у всех базовых коллекций в Kotlin, помо- жет быстро модифицировать список, «мапу» или «сет».

Рассмотрим пример на коллекции List. В качестве входного пара- метра она принимает лямбду с элементов списка в качестве аргумента и возвращает модифицированное значение.

Предположим, имеется список, который хранит числа и требуется получить новый список с удвоенными значениями (например, из [1, 2, 3, 4],

сделать [2, 4, 6, 8]). Обычным решением было бы инициализировать пу- стой список, сделать цикл и проходиться по каждому элементу попутно умножая его на 2. Решение с map выглядит следующим образом:

val list = listOf(1, 2, 3, 4)

val newList = list.map { it \* 2 } // [2, 4, 6, 8]

Результатом выполнения функции map является новый список. Рассмотрим более реалистичный пример. Предположим, имеется спи-

сок студентов типа *Student*, который принимает параметры *name*, *surname* и

*specialization*. Необходимо получить список фамилий из списка студентов:

data class Student(val name: String, val surname: String, val specialization: String)

val studentts = listOf(Student(“Даниил”, “Крюк”, “Python”), Student(“Елизавета”, “Зиненко”, “Swift”), Student(“Анатолий”, “Антоненко ”, “PHP”))

val surnames = list.map { it.surname } // [“Крюк”, “Зиненко”, “Антоненко”]

У map также есть много специфичных вариаций вроде flatMap, mapIndexed, mapNotNull, о которых можно почитать в официальной до- кументации [15].

### *Фильтрация*

Функция filter также принимает в качестве аргумента лямбду с аргу- ментом элемента списка, но возвращает значения true или false. При возвра- те true текущий элемент добавляется в модифицированную коллекцию, при false – игнорируется. Данная функция требуется, когда необходимо полу- чить новую коллекцию, отфильтрованную по какому-либо параметру.

Например, если требуются только числа больше 10:

val list = listOf(1, 7, 20, 8, 1, 15, 177)

val newList = list.filter { it > 10 } // [20, 15,177]

Функция filter возвращает массив из тех же элементов, но отфиль- трованных по указанному признаку.

Функции filer и map удобно комбинировать. Попробуем скомбини- ровать пример из предыдущего раздела с этим:

val list = listOf(2, 4, 6, 8)

val newList = list.map { it \* 2 }.filter { it > 10 } // [12, 16]

### *Другие операции*

Функции map и filter – самые распространенные. Также в библио- теке Kotlin присутствуют операции группирования (groupBy, reduce), раз- деления (slice), порядка (sortedBy). Об этих и других операциях над кол- лекциями можете почитать в официальной документации [15].

## Функции среды

Функции среды (или Scope Functions) – это обычные функции Ko- tlin, которые не относятся к синтаксису языка. Эти функции помогают работать с объектом в рамках его среды (контекста, если хотите). Всего таких функций 5, и они очень похожи: apply, let, also, with и run.

### *Функция let*

Чаще всего let используют в комбинации с оператором безопасного вызова. Например, если есть какая-то nullable-строка и мы хотим, чтобы определенный код выполнился только при условии, что эта строка суще- ствует:

val str: String? = null str?.let {

println(“$it”) // Этот код не выполнится благодаря Safe Call оператору

}

Функция let открывает контекст строки так, чтобы к ней можно бы- ло обращаться через определенную заданную переменную (чаще всего – стандартный «it»). По факту let – всего лишь функция-расширение, кото- рая в качестве параметра использует лямбду с единственным аргумен- том – объектом, над которым производится действие.

Функцию let также можно использовать как псевдоним, если назва- ние переменной слишком длинное, а нужно её использовать:

val veryVeryLongFreakingStringName: String? = “Костя Цзю” veryVeryLongFreakingStringName?.let {

println(“1 $it”) println(“2 $it”) println(“3 $it”)

}

### *Функция apply*

Функция apply используется почти так же, как и let, за исключени- ем того, что среда внутри блока открывается не как лямбда, а как контекст объекта. То есть внутри блока можно использовать ключевое слово this при обращении к элементу, над которым происходит операция. Возвра- щаемое значение – сам объект.

Ниже представлен пример из официальной документации Kotlin:

val adam = Person("Adam").apply {

age = 20 // то же, что this.age = 20 или adam.age = 20 city = "London"

}

println(adam)

Функция apply именно в таком виде используется чаще всего. Можно проинициализировать ваш объект и сразу применить к нему неко- торые настройки. Это отличный аналог паттерну «method chaining», по- скольку не требует реализации для каждого класса отдельно. Его можно использовать абсолютно с любым объектом.

### *Функция with*

Функция with не является расширением класса. Она запускается отдельно. В качестве аргумента принимает объект, над которым требуется проводить операции, а возвращает лямбду.

val numbers = mutableListOf("one", "two", "three") val firstAndLast = with(numbers) {

"The first element is ${first()}," + " the last element is ${last()}"

}

println(firstAndLast)

### *Функция also*

Функция also использует в среде лямбду, а возвращает объект. Бла- годаря этой особенности можно, например, поменять значения перемен- ных, используя только эту функцию:

var a = 1 var b = 2

a = b.also { b = a }

## Идиомы

В данном подразделе собраны некоторые идиомы, которые значи- тельно могут упростить написание кода.

### *Поиск в коллекции*

Например, можно проверить наличие какого-либо объекта в кол- лекции следующим образом:

if ("Liza" in names) { ... } // содержит

if ("Mark" !in names) { ... } // не содержит

### *Kotlin-синглтон*

В мире Kotlin синглтон не особо считается паттерном проектирова- ния, поскольку реализуется буквально в пару строчек при помощи кон- струкций языка:

object Singletone { val name = "Name"

}

### *Проверка типа*

С помощью выражений when или if можно проверить тип входного значения.

when (x) {

is Int -> doSomethingWithInt()

is String -> doSomethigWithString() else -> doSomethingElse()

}

В теле каждого из случаев можно работать с методами, относящи- мися к этому типу.

## Многопоточность и сопрограммы (coroutines)

Многопоточное1 программирование – одна из самых больших го- ловных болей в Java и которую крайне тяжело использовать на практике.

Во многих современных языках программирования есть свои пара- дигмы многопоточного (конкурентного, асинхронного, сейчас это не име- ет значения) программирования: async/await в C#, JavaScript и Python, callback в JS (хотя сейчас такое редко встретишь), «обещания» (promise) в том же JS. Последние также представлены в самой Java как future.

Kotlin также предоставляет простой высокоуровневый интерфейс для многопоточного программирования – сопрограммы (coroutines, далее – ко- рутины). Корутины [16] – это парадигма, которая не пытается навязывать

1 Многопоточность в данном контексте используется как обобщенное понятие, ко- торое объединяет в себе такие термины, как асинхронность, параллелизм, конкурентность. Важно запомнить, что сопрограммы в Kotlin не про многопоточность, а про асинхронное выполнение.

определенную парадигму. Корутины в Kotlin представлены как отдельная библиотека, набор функций, который предоставляет такой API, с помощью которого можно создать и promise-стиль, и async/await [17]. Они позволяют писать асинхронный код так, как вы бы писали последовательный.

В целом корутины:

* не определяют парадигму многопоточного программирования. Можно использовать их для ожидания, можно получить promise, а можно выполнять параллельно;
* легковесны для программистов, по факту это не имеет значе- ния, но для системы и её быстродействия – очень даже;
* не встроенная функция языка, это отдельная библиотека, по- чти не опирающаяся на ключевые слова языка (за исключением ключево- го слово suspend).

### *Запуск корутин (launch и async)*

При работе с корутинами чаще всего будут использованы следую- щие элементы:

* функции *launch* и *async* для запуска корутин;
* функции *runBlocking* и *coroutineScope* для создания среды ко- рутин (coroutine scope, далее – «скоуп»);
* ключевое слово *suspend*.

Для запуска корутины используются функции *launch* и *async*. Для тестирования можно выполнять код в обычной activity или создать от- дельный проект с входной функцией main. Пример запуска launch:

fun coroutineTest() { println("Start function") GlobalScope.launch { delay(2000) println("Inside function")

}

println("End function")

}

// Вывод в консоль:

// Start function

// Inside function

// End function

В примере для запуска корутины использован не просто *launch*, а *GlobalScope.launch*. Это говорит о том, что корутина запустится в гло- бальном скоупе. Что это значит? Всего лишь то, что жизненный цикл ко- рутины ограничен жизненным циклом всего приложения.

Очень важно понимать, для чего используется скоуп и в каких си- туациях его использовать. Предположим, корутина используется для того, чтобы подключиться к какому-либо каналу, например веб-сокету или мес- седж-брокеру. Нужно помнить о том, что после уничтожения activity ко- рутина будет жить. Это может привести к двум проблемам. Во-первых, теоретическая попытка вернуть данные в несуществующий объект (утечка памяти). Во-вторых, повторное подключение к серверу после повторного запуска этой activity, что может привести к конфликтам. Многие библио- теки, использующие корутины в Android, создают свой определенный скоуп. Например, библиотека ViewModel (о которой мы поговорим чуть позже) создает свой скоуп, привязанный к жизненному циклу activity, что решает много проблем, связанных с утечкой памяти.

Рассмотрим пример на запуск функции async:

private fun coroutineTest() { println(“Выполнение coroutineTest”)

val first = GlobalScope.async { delay(1000)

"First"

}

val second = GlobalScope.async { delay(3000)

"Second"

}

GlobalScope.launch { println(“Выполнение launch”)

val message = "${first.await()} plus ${second.await()}" println(message)

}

}

// Выполнение coroutineTest

// Выполнение launch

//// ждет 3 секунды

// First plus Second

Программа на основе представленного выше кода подождёт 3 с и напечатает «First plus Second». В данном случае не имеет значения, что одна функция выполняется одну секунду, а вторая – три. Значение в mes- sage присваивается только тогда, когда выполнилась более длительная операция.

Отметим, что async и launch – это не просто функции, а лямбды.

Поэтому последняя строка внутри тела async – возвращаемое значение.

Таким образом, подходим к ключевой разнице между async и launch: первая возвращает объект Deffered<T> (в нашем случае это был Deffered<String>), а вторая – ничего. Async не блокирует поток, а возвра- щает «обещание», что выполнится, а блокировка происходит только на вызове у «обещания» функции *await.* Launch же блокирует поток сразу после вызова.

Можете поэкспериментировать и вызвать функцию coroutineTest таким образом:

println(“ ”)

coroutineTest() println(“ ”)

Прошу прощение за спойлер, но результат будет таким:

//

// Выполнение coroutineTest

//

// Выполнение launch

//// ждет 3 секунды

// First plus Second

### *Прерываемые функции*

В ходе экспериментов можно заметить, что написать, например, функцию *delay()* в обычной функции не получится. Дело в том, что delay – прерываемая функция. Прерываемые функции могут быть запущены только внутри корутины или другой прерываемой функции (так называе- мой suspend-функции). Ключевое слово suspend показывает, что функция может быть прервана.

Перепишем представленный выше пример с использованием sus- pend-функций:

suspend fun waitOneSecond() {

delay(1000)

}

suspend fun waitThreeSeconds() { delay(3000)

}

fun coroutineTest() { println(“Выполнение coroutineTest”)

val first = GlobalScope.async { waitOneSecond()

"First"

}

val second = GlobalScope.async { waitThreeSeconds()

"Second"

}

GlobalScope.launch { println(“Выполнение launch”)

val message = "${first.await()} plus ${second.await()}" println(message)

}

}

Результат будет аналогичным.

Suspend-функции также могут возвращать значение. Можем напи- сать такие функции и убрать возвращаемое значение из вызовов async:

suspend fun waitOneSecond(): String { delay(1000)

return “First”

}

suspend fun waitThreeSeconds(): String { delay(3000)

return “Second”

}

Получается, что suspend-функции работают почти как функции об- ратного вызова (на самом деле, весь принцип корутин «под капотом» ос- нован на callback).

### *Контекст вызова*

В представленных примерах не был использован контекст вызова. Контекст – это специальный объект, который передается в качестве аргу- мента launch и async. Контекст используется для «настройки» потока, в котором будут выполняться корутины. Это может быть поток, настроен- ный фреймворком или даже созданный вручную. Еще раз уточним, кору- тины – не потоки и выполняются внутри определенного потока.

Рассмотрим контексты на примере Android. Android предоставляет три диспетчера для запуска корутин:

* Dispatchers.Main – мейн-тред для взаимодействия с UI;
* Dispatchers.IO – поток ввода-вывода для взаимодействия с данными (манипуляции с дисковым пространством, обращение к базе данных, обращение к серверу);
* Dispatchers.Default – поток для выполнения сложных вычисли- тельных операций (парсинг, сортировка очень больших данных, сложные математические расчеты).

Как вам должно быть известно, если мейн-тред Android «простаи- вает» 5 с, операционная система выбрасывает ошибку «Приложение не отвечает».

В качестве эксперимента с контекстом корутины запустим такой код из onCreate в activity:

fun blocking() = runBlocking(Dispatchers.Main) { delay(5000)

}

// а затем

fun blocking() = runBlocking(Dispatchers.IO) { delay(5000)

}

Первый код выдаст ошибку «Не отвечает», а второй – нет, но всё равно простоит 6 с. Ответ на эту загадку вы можете найти в документации к корутинам, прочитав блок про среду выполнения корутин (coroutine scope) [16].

Функция runBlocking блокирует поток, поэтому, чтобы приложение не упало, внутри runBlocking нужно запустить корутину:

fun blocking() = runBlocking(Dispatchers.IO) { GlobalScope.launch {

delay(5000)

}

}

Функцию выше приложение просто не заметит и будет работать в штатном режиме.

Представленного в подразд. 1.9 материала достаточно, чтобы по- нимать всё асинхронное взаимодействие, которое будет использоваться в последующих материалах. В данном блоке не были упомянуты такие ин- тересные аспекты как flow, channels и отлавливание ошибок. Подробнее с ними можно ознакомиться в документации [17].

## Контрольные вопросы

1. Особенности языка Kotlin.
2. Перечислите преимущества языка Kotlin.
3. Определение атрибута в Kotlin.
4. Описание функций в Kotlin.
5. Отличие в реализации функции установки слушателя в Kotlin от Java.
6. Создание проекта на Kotlin в Android Studio.
7. Что такое лямбда-выражение?
8. Реализация лямбда-выражений на Kotlin.
9. Для чего используется KTX?
10. Работа с KTX.
11. Реализация безопасности в Kotlin.
12. Реализация Nullable-типов на Kotlin.
13. Что такое безопасные вызовы?
14. Что такое elvis-оператор и как его использовать?
15. Что такое Оператор NPE и как его использовать?
16. Что такое замыкание?
17. Для чего и как используются функции-расширения?
18. Для чего и как используется функция filter?
19. Для чего и как используется функция let?
20. Для чего и как используется функция apply?

36

1. Для чего и как используется функция with?
2. Для чего и как используется функция also?
3. Что такое корутины в Kotlin?
4. Каким образом запустить корутину?
5. Что такое прерываемые функции и как с ними работать?
6. Что такое контекст?

# РАБОТА С СЕТЬЮ (RETROFIT)

*Контрольные вопросы*

Retrofit – это открытая библиотека, разработанная и поддерживае- мая разработчиками из компании Square [18]. «Под капотом» использует библиотеку OkHttp как HTTP-клиент.

Retrofit может помочь построить удобный класс для инициализации HTTP-клиента. Вы пишете интерфейс со специальными аннотациями, а Retrofit создает экземпляр. Этот экземпляр, в свою очередь, управляет созда- нием HTTP-запроса (который описан в интерфейсе) и парсит HTTP-ответ в формат, который требуется (строковый, собственный объект и т.д.).

## Инициализация зависимостей

Для работы с Retrofit необходимо добавить зависимость в

*build.gradle* модуля *app* и далее синхронизировать Gradle-файл.

dependencies {

implementation fileTree(dir: 'libs', include: ['\*.jar']) implementation"org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin\_version"

…

implementation 'com.squareup.retrofit2:retrofit:2.6.2'

}

Перед тем как попробовать Retrofit в деле, сначала необходимо его сконфигурировать.

## Конфигурация Retrofit-интерфейса

Изначально необходимо создать новую директорию (package) для Retrofit-интерфейсов. Можно это сделать в корневой директории (там, где обычно Android Studio создает MainActivity). Хорошим тоном разработки считается назвать директорию service или api. На официальном сайте биб- лиотеки такие интерфейсы называются именно “services”.

Пример простого сервиса с одним GET-запросом:

interface UserService {

@GET("api")

fun fetchUsers(): Call<Response>

}

Если IDE предлагает несколько вариантов импорта Call, то следует выбрать тот, что относится к пакету *retrofit2*.

Каждая функция в интерфейсе должна определять HTTP-запрос, т.е. иметь аннотацию HTTP-запроса. Данная аннотация сообщает Retrofit тип HTTP-запроса (также можно встретить определения «HTTP-метод» или «HTTP-verb»). Помимо @GET, показанного в примере, используются аннотации @POST, @PUT, @DELETE. Полный список можно посмотреть на сайте библиотеки [19].

Аннотация @GET в приведенном выше коде конфигурирует *вы- зов (тип Call),* возвращаемый функцией fetchUsers() для выполнения GET-запроса.

Аннотация @GET в скобках принимает некий строковый параметр. Это относительный путь базового адреса энд-поинта. В примере выше сервис будет обращаться к некому адресу *{URL}* по адресу *“api”* (напри- мер, https://mandarinshow.ru/api).

По умолчанию все веб-запросы, генерируемые Retrofit, возвращают объект типа Call. Объект Call представляет единственный веб-запрос (*re- quest*), который можно выполнить. При выполнении Call создается один соответствующий веб-ответ (*response*).

Retrofit – гибкая библиотека, в будущем можно заменить стандарт- ный Call, например, объектом Observable RxJava. Или вовсе избавиться от Call и управлять запросами более низкоуровневым способом с помощью Kotlin-корутин.

В generic-типе Call содержится тип, который представляет данные. Если требуется вернуть строку, то пишется String в generic (это будет вы- глядеть как *Call<String>*). Если требуется вернуть собственный POJO или Kotlin data class, можно обернуть его в generic.

## Конфигурация Retrofit-объекта

Экземпляр Retrofit отвечает за реализацию и создание экземпляров вашего сервиса (интерфейса).

Рекомендуется держать инициализацию Retrofit-объекта в отдель- ном файле. Например, можно создать пакет с названием *network* и опреде- лить там файл RestApi.kt (или RestApi.java, если используется Java):

class RestApi { companion object {

private const val BASE\_URL = "https://randomuser.me/"

}

var instance: Retrofit = Retrofit.Builder()

.baseUrl(BASE\_URL)

.build()

}

В данном случае реализовывать паттерн singleton вовсе необяза- тельно. Retrofit лишь инициализирует локальный HTTP-клиент. Исполь- зовать singleton стоило бы в случае, например, использования библиотеки, которая постоянно «слушает» HTTP-сервер, и разрыв соединения.

Retrofit.Builder() – это текучий интерфейс, который упрощает кон- фигурацию и создание экземпляра Retrofit. Методы класса Builder воз- вращают this, что обеспечивает функциональный вид инициализации Ret- rofit-объекта.

Метод *baseUrl* принимает строку URL. Она должна начинаться с определения протокола (например, *https://*) и заканчиваться литералом ‘/’.

Вызов метода *build* возвращает Retrofit-экземпляр, который был сконфигурирован ранее описанными методами.

## Инстанцирование Retrofit-сервиса

Конечным шагом в конфигурации Retrofit будет создание экзем- пляра сервиса (интерфейса). Логичным шагом было бы имплементировать этот интерфейс в каком-нибудь классе, но всё обстоит не так. Этот интер- фейс (а точнее ссылку на него) необходимо передать Retrofit, который создает его экземпляр сам в runtime. Такой способ работы с объектами называется *рефлексией*.

private val retrofit = RestApi().instance

private val userService = retrofit.create(UserService::class.java)

Этот код можно расположить в Activity (или Fragment). Но хоро- шим тоном считается, вывести этот код в контроллер (presenter, control- ler, viewmodel, в зависимости от архитектурного паттерна, который ис- пользуется).

## Добавление конвертеров

По умолчанию Retrofit десериализует ответы от сервера в объект ResponseBody, который является частью библиотеки okhttp3. Такой фор- мат не совсем подходит для удобной работы с парсингом различных отве- тов от сервера.

Как уже упоминалось, Retrofit – гибкая библиотека и позволяет ис- пользовать «кастомный» (взятый из публичного репозитория или свой собственный) способ парсинга.

Например, чтобы Retrofit спарсил строку, можно взять конвертер скалярных величин из открытого репозитория Square:

dependencies {

implementation fileTree(dir: 'libs', include: ['\*.jar'])

…

implementation 'com.squareup.retrofit2:retrofit:2.7.2' implementation 'com.squareup.retrofit2:converter-scalars:2.7.2' implementation 'com.squareup.retrofit2:converter-gson:2.7.2'

}

А затем добавить его к builder’у:

var instance: Retrofit = Retrofit.Builder()

.baseUrl(BASE\_URL)

.addConverterFactory(ScalarsConverterFactory.create())

.build()

После добавления этого конвертера можно добавить String в generic-параметр вызова Call в интерфейсе:

@GET("api")

fun fetchUsers(): Call<String>

Этот пример вернет в Call сырую строку из ответа body.

Обратите внимание, что в примере включение зависимости в Gradle- файл присутствует не только скалярный конвертер, но и Gson. С помощью этого конвертера можно парсить JSON-данные в POJO или Kotlin data class. Для этого нужно в generic добавить тип объекта с Gson-аннотациями.

## Выполнение веб-запросов с помощью Retrofit

После завершения конфигурации Retrofit можно выполнять HTTP-запросы. Особенность в том, что нет необходимости выполнять эти рутинные действия при написании каждого запроса, просто добавля-

ете нужные методы в интерфейс (конечно же, если вам не нужно рабо- тать с разными типами ответов и энд-поинтов. В таком случае, будет логично конфигурировать Retrofit заново).

После инициализации сервиса можно попробовать вызвать его ме- тод. Например, следующим образом:

val fetchUsersRequest: Call<String> = userService. fetchUsers()

После данной строчки кода ничего не произойдет, потому что дан- ная функция всего лишь конфигурирует запрос.

Для выполнения асинхронного запроса у типа Call есть метод

*enqueue().*

fetchUsersRequest.enqueue(object : Callback<String> { override fun onFailure(call: Call<String>, t: Throwable) {

TODO("Not yet implemented")

}

override fun onResponse(call: Call<String>, response: retro- fit2.Response<String>) {

TODO("Not yet implemented")

}

})

Этот метод принимает callback типа Callback<T>, где T – generic, который повторяет такой же в вашем сервисе. Этот callback переопреде- ляет две функции: onFailture (дословно «при неудаче»), onResponse («при ответе»).

Представленный функционал делает две важные вещи в контексте Android-разработки:

1. Выполняет «тяжелую» операцию в фоновом потоке (back- ground thread).
2. Обновляет UI в главном потоке (main thread), используя callback.

## Контрольные вопросы

1. Что такое Retrofit?
2. Для чего используется Retrofit?
3. Какие зависимости необходимо добавить при работе с Retrofit?
4. Как сконфигурировать Retrofit- интерфейс?

42

1. Что такое аннотация HTTP-запроса?
2. Какие существуют аннотации?
3. Что такое объект Call?
4. Как сконфигурировать Retrofit-объект?
5. Для чего используется Retrofit.Builder()?
6. Как создается экземпляр сервиса?
7. Парсинг ответов в Retrofit.
8. Как выполнить HTTP-запрос с использованием Retrofit?

# БАЗА ДАННЫХ (ROOM)

*3.1. Начало работы с Room*

Почти каждое приложение нуждается в месте, которое хранит дан- ные. В данном разделе будет рассмотрен пример реализации базы заметок.

Room [20] – библиотека для управления базой данных SQLite, со- зданная и поддерживаемая разработчиками из Google. Она помогает опи- сывать структуру базы данных в коде, используя аннотации.

Room состоит из API, аннотаций и компилятора. API включает классы, которые должны быть унаследованы для того, чтобы описать базу данных и создать её экземпляр. Аннотации применяются для того, чтобы компилятор понимал, что используется конкретный класс для описания модели БД, а также для описания таблиц и отношений между ними.

В первой части данного раздела будет рассмотрено создание про- стейшего приложения хранения заметок (без создания интерфейса). Во второй части – обратимся к приложению, на примере которого была рас- смотрена библиотека Retrofit в разд. 2.

## Начало работы с Room

Для того чтобы начать пользоваться Room, требуется настроить Gradle-файл и добавить зависимости из AndroidX.

В данном случае конфигурация для Kotlin [21] и Java [22] будут не- сколько отличаться, поскольку Room писалась именно для работы с Kotlin.

### *Настройка зависимостей для Kotlin*

В начало файла *build.gradle* (модуль: app) необходимо вставить инициализацию kapt-плагина:

apply plugin: 'kotlin-kapt'

Расшифровывается kotlin-kapt как «Kotlin annotation processor tool». Данный плагин позволяет Android Studio видеть файлы, сгенери- рованные библиотекой, что позволяет использовать их и импортировать в свои классы.

В блок dependencies требуется добавить следующие зависимости:

// Room-компоненты

implementation "androidx.room:room-runtime:$rootProject.roomVersion" kapt "androidx.room:room-compiler:$rootProject.roomVersion" androidTestImplementation "androidx.room:room- testing:$rootProject.roomVersion"

// Lifecycle-компоненты

\*implementation "androidx.lifecycle:lifecycle-

\*extensions:$rootProject.archLifecycleVersion"

\*kapt "androidx.lifecycle:lifecycle-compiler:$rootProject.archLifecycleVersion"

\*implementation "androidx.lifecycle:lifecycle-viewmodel-

\*ktx:$rootProject.archLifecycleVersion"

Зависимости, выделенные звездочкой, добавлять необязательно (ниже будет объяснена причина).

Переменные с версиями библиотек можно вынести в def в этом же модуле или определить в файле build.gradle в модуле проекта.

ext {

roomVersion = '2.2.5'

archLifecycleVersion = '2.2.0'

}

### *Настройка зависимостей для Java*

В файле *build.gradle* (модуль: app) в блок android необходимо вста- вить блок compileOptions следующего содержания (в случае, если предпо- лагаете использовать лямбда-выражения из Java 8):

compileOptions { sourceCompatibility = 1.8

targetCompatibility = 1.8

}

В блок dependencies требуется добавить следующие зависимости:

// Room components

implementation "androidx.room:room-runtime:$rootProject.roomVersion" annotationProcessor "androidx.room:room-compiler:$rootProject.roomVersion" androidTestImplementation "androidx.room:room- testing:$rootProject.roomVersion"

// Lifecycle components

implementation "androidx.lifecycle:lifecycle- extensions:$rootProject.archLifecycleVersion"

annotationProcessor "androidx.lifecycle:lifecycle- compiler:$rootProject.archLifecycleVersion"

Зависимости, выделенные звездочкой, добавлять необязательно (ниже будет объяснена причина).

Переменные с версиями библиотек можно вынести в def в этом же модуле или определить в файле build.gradle в модуле проекта.

ext {

roomVersion = '2.2.5'

archLifecycleVersion = '2.2.0'

}

## Создание базы данных

Рассмотрим основные пункты создания базы данных с Room [23]:

* добавляем аннотацию к классу модели, чтобы сделать её сущ- ностью;
* создаем класс, который конфигурирует базу данных (а также представляет сущность самой БД);
* создаем прослойку, чтобы «вытаскивать» данные из БД и представлять их в виде понятных коду объектов.

### *Определение сущностей*

Класс, который будет сейчас сделан представляет простую таблицу БД и называется *«сущность»* или *“entity”*[24]. Этот POJO-класс, каждое поле которого будет представлять поле таблицы базы данных SQLite.

Для примера создадим класс Note, с полями title (заголовок замет- ки), content (содержание заметки) и id (идентификатор).

Ниже представлен пример на Kotlin:

@Entity(tableName = "note\_table") data class Note(

override var title: String, override var content: String,

@PrimaryKey(autoGenerate = true) override val id: Long? = null

)

Далее представлен пример на Java:

@Entity(tableName = "note\_table") public class Note {

@PrimaryKey(autoGenerate = true) @NonNull

@ColumnInfo(name = "id") private Long mId;

@NonNull @ColumnInfo(name = "title") private String mTitle;

@NonNull

@ColumnInfo(name = "content") private String mContent;

public Note(String title, String content) { this.mTitle = title;

this.mContent = content;

}

public String getTitle(){return this.mTitle;}

// Остальные геттеры/сеттеры убраны из соображений экономии места

}

Аннотация @Entity применяется к POJO-классу и определяет, что данный класс является сущностью, параметр tableName задает имя таблицы.

По умолчанию имя поля БД задается автоматически по имени поля в нашем классе. Чтобы задать специфическое название поля, используйте аннотацию @ColumnInfo и параметр name.

Аннотация @NonNull сообщает, что в поле таблицы не может хра- ниться значение null. В Kotlin можно просто использовать встроенный Nullability.

Для того чтобы задать первичный ключ, можно использовать анно- тацию @PrimaryKey или параметр primaryKeys в аннотации @Entity. Ан- нотация @PrimaryKey задается к полю и имеет параметр autoGenerate, выставляя который, делегируете генерацию id базе данных (т.е. не создае- те его самостоятельно). Настройка первичного ключа через @Entity вы- глядит следующим образом для Kotlin:

@Entity(primaryKeys = arrayOf("firstName", "lastName"))

Для Java:

@Entity(primaryKeys = {"firstName", "lastName"})

### *Инициализация объекта доступа к данным*

В объекте доступа к данным [25] (или DAO, data access object) определяете SQL-запросы и привязываете их к вызову методов. Компиля- тор генерирует запрос по аннотации, которая указывается и данным, ко- торые передаются через функцию.

DAO обязательно должен быть интерфейсом или абстрактным классом.

Посмотрим на пример DAO для базы с заметками:

@Dao

interface NoteDao {

@Insert

suspend fun insert(note: Note)

@Update

suspend fun update(note: Note)

@Delete

suspend fun delete (note: Note)

@Query("DELETE FROM note\_table") suspend fun deleteAll()

@Query("SELECT \* FROM note\_table ORDER BY id DESC") fun getAll(): LiveData<List<Note>>

@Query("SELECT \* FROM note\_table WHERE id=:noteId") fun getNoteById(noteId: Long): LiveData<Note>

}

Для Kotlin здесь есть специфика. Представленный пример исполь- зует suspend-функции. Это значит, что библиотека умеет работать с кору- тинами (подробнее про корутины рассмотрено в подраз. 1.9).

Java-специфичный код:

@Dao

public interface NoteDao {

@Insert

void insert(Note note);

@Update

void update(Note note);

@Delete

void delete (Note note);

@Query("DELETE FROM note\_table") void deleteAll();

@Query("SELECT \* FROM note\_table ORDER BY id DESC") LiveData<List<Note>> getAll();

@Query("SELECT \* FROM note\_table WHERE id=:noteId") LiveData<Note> getNoteById(Long noteId);

}

Аннотация @Dao применяется к интерфейсу или абстрактному классу, который объявляется как объект доступа к данным.

Аннотации @Insert, @Update, @Delete – это манипуляторы управ- ления данными, аналогичные соответствующим SQL-запросам (как уже говорилось ранее, Room – всего лишь интерфейс для SQLite, а не какой-то новый способ хранения данных). Очевидно, что методы с такими аннота- циями не должны возвращать значения.

С помощью аннотации @Query можно написать обычный SQL-запрос (в том числе delete, insert, update, но это не имеет смысла при наличии от- дельных аннотаций). Например, метод deleteAll использует ключевое слово

«delete» в SQL-запросе, только запрос удаляет все элементы из базы данных. Для того чтобы упростить асинхронную работу с получением дан-

ных, используется библиотека LiveData. В методе getAll заворачиваем возвращаемое значение в LiveData-контейнер. Этот метод не требуется вызывать асинхронно, значение обновится в контейнере, а контейнер со- общит об этом наблюдателю. Из примера видно, что для метода getAll пишется обычный SQL-запрос, который возвращает список элементов, отсортированный в обратном порядке.

В методах deleteAll, getAll и getNoteById в запросах используется имя таблицы «note\_table», которое ранее было задано в Entity. Android Studio замечательно работает с Room и подскажет правильное имя табли- цы, а также выделит его в подсветке кода.

Метод getNoteById представляет запрос, который вытаскивает одну заметку из таблицы по идентификатору. Видно, что для указания нужного id используется параметр функции, название которого требуется также написать в самом запросе. Повторяя пример выше, синтаксис выглядит как «:param»:

@Query("SELECT \* FROM note\_table WHERE id=:noteId") LiveData<Note> getNoteById(Long noteId);

В примере также используется LiveData-контейнер. LiveData хоро- шо работает в комбинации с Room. Разработчики также рекомендуют ис- пользовать именно её, а не манипуляции с асинхронностью.

### *Инициализация базы данных*

Ранее была создана сущность таблицы, которая представлена в ви- де классического объекта. Затем инициализирована модель для управле- ния данными. Далее необходимо инициализировать саму базу данных.

@Database(entities = [Note::class], version = 1) abstract class NoteDatabase : RoomDatabase() {

abstract fun noteDao(): NoteDao

companion object {

private const val DATABASE\_NAME = "note-db"

@Volatile

private var instance: NoteDatabase? = null

fun getInstance(context: Context): NoteDatabase { return instance ?: synchronized(this) {

instance ?: buildDatabase(context).also { instance = it }

}

}

private fun buildDatabase(context: Context): NoteDatabase { return Room.databaseBuilder(context.applicationContext,

NoteDatabase::class.java, DATABASE\_NAME)

.build()

}

}

}

Что же видно из примера? По большому счету – классический син- глтон, который использует некоторые компоненты для инициализации и настройки базы данных.

Перед тем, как разобрать код, представим код для Java:

@Database(entities = {Note.class}, version = 1)

public abstract class NoteDatabase extends RoomDatabase { public abstract NoteDao noteDao();

private static volatile NoteDatabase INSTANCE;

static NoteDatabase getDatabase(final Context context) { if (INSTANCE == null) {

synchronized (NoteDatabase.class) { if (INSTANCE == null) {

INSTANCE =

Room.databaseBuilder(context.getApplicationContext(),

NoteDatabase.class, "note-db ")

.build();

}

}

}

return INSTANCE;

}

}

В примере NoteDatabase – абстрактный класс, который наследуется от класса RoomDatabase из библиотеки Room. Абстрактное состояние класса требуется для рефлективной инициализации уже готового класса, к полям и методам которого можно будет свободно обратиться.

Рефлективная инициализация осуществима благодаря аннотации @Database, которая применяется к классу. В нашем случае, к ней приме- няется два параметра: entities и version. Первый принимает в себя массив сущностей, которые будет виден в БД (это означает, что Room-сущности можно переиспользовать для нескольких баз данных, что снова отсылает нас к Retrofit и переиспользованию сервисов).

Единственный не статичный атрибут, который содержится в реали- зации базы – noteDao. Это абстрактная функция, которая возвращает ра- нее созданный объект доступа к данным, и единственный способ привя- зать dao-интерфейс к базе.

Таким образом, для создания экземпляра базы требуется унаследо- ваться от класса RoomDatabase, добавить аннотацию @Database и доба- вить абстрактные функции, которые будут служить DAO-объектами.

Далее перейдем к созданию синглтона. Здесь инициализируем обычный конкурентный синглтон, используя ключевые слова volatile и synchronized. Они служат для синхронизации при теоретическом обраще- нии к методам из нескольких разных потоков.

Чтобы не получилась ситуация, при которой один поток пытается получить инстанс, пока другой поток инициализирует базу, добавим клю- чевое слово volatile (или аннотацию @Volatile для Kotlin):

private static volatile NoteDatabase INSTANCE;

@Volatile private var instance: NoteDatabase? = null

Следующий опасный для одновременного обращения метод get- Database. В этом случае синхронизируем его при помощи ключевого сло- ва synchronized.

Про конкурентность и ключевые слова volatile и synchronized мо- жете прочитать в дополнительных материалах [26].

Подробности про использование @Database находятся в официаль- ных уроках Google [21, 22] и в официальной документации [20].

## Использование базы данных

### *Использование паттерна «Репозиторий»*

Использование паттерна «Репозиторий» является «хорошей прак- тикой», но не является обязательным. Репозиторий – абстракция для до- ступа к различным данным. Главной его целью является абстрагироваться от технологии, используемой для доступа к данным (Room, SQLite, Retro- fit, OkHttp), и сосредоточиться на том, какие данные получаем.

Простая реализация для приведенного в данном разделе примера будет выглядеть вот так:

class NoteRepository(private val noteDao: NoteDao) {

var allNotes: LiveData<List<Note>> = noteDao.getAll() suspend fun insert(note: Note) {

noteDao.insert(note)

}

suspend fun delete(note: Note) { noteDao.delete(note)

}

}

Или аналог на Java:

class NoteRepository {

private NoteDao mNoteDao;

private LiveData<List<Word>> mAllNotes;

WordRepository(NoteDao noteDao) { mNoteDao = noteDao;

mAllNotes = mNoteDao.getAll();

}

LiveData<List<Note>> getAllNotes() { return mAllNotes;

}

void insert(Note note) { NoteDatabase.databaseWriteExecutor.execute(() -> {

mNoteDao.insert(note);

});

}

}

Обратите внимание, что внутри класса ничего не инициализирует- ся, он получает данные извне и манипулирует ими. Таким образом, класс легко протестировать (подробнее про тестирование в разд. 7).

В конструкторе получаем уже инициализированный DAO noteDao и манипулируем им.

Метод getAllNotes возвращает LiveData-контейнер со списком за-

меток.

Функции insert и delete манипулируют noteDao и асинхронно запи-

сывают/удаляют данные. Для Kotlin гарантировано, что метод вызовется асинхронно, для Java используем *databaseWriteExecutor.* Метод database- WriteExecutor гарантирует, что код внутри колбека будет выполнен не в UI-потоке. На Kotlin он не используется благодаря корутинам.

### *Использование паттерна «ViewModel»*

Также, как и в случае с использованием «репозитория», использовать паттерн «ViewModel» не является обязательным. Более подробно про паттерн можно прочитать в главе про архитектурные компоненты в подразд. 6.1.

Для инициализации *ViewModel* в данном случае будет использован не одноименный класс, а *AndroidViewModel.* При инициализации он при- нимает параметр *application.* По большому счету AndroidViewModel явля- ется более низкой (близкой по отношению к слою View) прослойкой, чем ViewModel, и используется для специфичных случаев. Наличие в этом классе объекта с типом, который принадлежит Android SDK – нарушение принципа паттерна MVVM.

В этом случае используется AndroidViewModel для упрощения. «Хо- рошей практикой» было бы передать в конструктор уже инициализирован- ный репозиторий, а «отличной практикой» – интерфейс этого репозитория.

Рассмотрим пример на Kotlin:

class NotesViewModel(application: Application) : AndroidViewMod- el(application) {

private val repository: NoteRepository val allNotes = repository.allNotes

init {

val noteDao = NoteDatabase.getInstance(application).noteDao() repository = NoteRepository(noteDao)

}

fun insert(note: Note) = viewModelScope.launch(Dispatchers.IO) { repository.insert(note)

}

fun delete(note: Note) = viewModelScope.launch(Dispatchers.IO) { repository.delete(note)

}

}

Пример на Java:

public class NoteViewModel extends AndroidViewModel { private NoteRepository mRepository;

private LiveData<List<Note>> mAllNotes;

public NoteViewModel (Application application) {

super(application);

NoteDao dao = NoteDatabase.getInstance(application).noteDao() mRepository = new NoteRepository(dao);

mAllNotes = mRepository.getAllNotes();

}

LiveData<List<Note>> getAllNotes() { return mAllNotes; }

public void insert(Note note) { mRepository.insert(note); }

}

В конструкторе получаем экземпляр application, который требуется для создания базы данных. В конструкторе используем синглтон для об- ращения к базе данных и вытаскиваем оттуда noteDao. Далее инициализи- руем репозиторий и передаем туда noteDao в качестве параметра.

Для Kotlin есть одно отличие – сразу используем параллельный по- ток для обращения к данным. Делается это при помощи встроенной в An- droidViewModel переменной viewModelScope, передавая туда параметр Dispatchers.IO. Про запуск корутин подробнее говорилось в подраз. 1.9., про viewModelScope – в подраз. 6.1.

## Кэширование с Room

Популярные Java-библиотеки вроде Retrofit и OkHttp поддержива- ют кэширование по умолчанию, но манипулировать данными, которые получаются в результате такого подхода, крайне тяжело.

Связка обращений к оффлайн и онлайн данным с использованием Room и Retrofit очень популярна. Часто подходы кэширования с исполь- зованием базы данных называют «online first» или «offline first», подразу- мевая приоритет «вытягивания» из прослойки данных. Поскольку в дан- ном подразделе речь идет о кэшировании, очевидно, будет применяться метод «online first».

В таком подходе нет ничего сложного. Кэширование данных будет производиться каждый раз, когда данные приходят с сервера. В целом логику можно свести к следующей схеме (рис. 2).

Конкретно такая схема не только позволяет кэшировать данные, но и ограничить показ сообщений об ошибке пользователю. «Исключением»

в данной ситуации может являться и отсутствие интернета, и ошибка со стороны сервера, и клиентская ошибка.



**Рис. 2.** Логика работы кэширования

### *Реализация логики кэширования*

Напишем простейшую реализацию логики кэширования. Предста- вим, что имеется API, которая возвращает список пользователей.

Retrofit-сервис для этой API будет возвращать список пользовате- лей в обертке Response, которая хранит в себе мета-данные о запросе (в том числе ответа):

interface UserService {

@GET("api")

suspend fun getUsers(): Response<List<UserInfo>>

}

interface UserService { @GET("api")

Response<List<UserInfo>> getUsers();

}

Для примера, модель для сервера и локальной БД можно переис- пользовать путем добавления и аннотаций Room, и аннотаций Gson для сериализации:

@Entity(tableName = "users\_table") data class UserInfo(

@SerializedName(“user\_name”) override var name: String, @SerializedName(“user\_surname”) override var surname: String, @PrimaryKey(autoGenerate = true) override val id: Long? = null

)

На Java:

@Entity(tableName = "users\_table") public class User {

@PrimaryKey(autoGenerate = true) @NonNull

@ColumnInfo(name = "id") private Long mId;

@NonNull @ColumnInfo(name = "name")

@SerializedName(“user\_name”) private String mName;

@NonNull

@ColumnInfo(name = "surname") @SerializedName(“user\_surname”) private String mSurname;

public User(String name, String surname) { this.mName = name;

this.mSurname = surname;

}

public String getName(){return this.mName;}

// Остальные геттеры/сеттеры убраны из соображений экономии места

}

DAO-класс базы данных в таком случае будет хранить ту же самую модель, что получает c сервера. В нашем случае будем использовать толь- ко операции вставки списка (команда insert) и вытягивание списка (ко- манда select).

Предположим, что есть некий класс-репозиторий, в котором суще- ствуют Retrofit-сервис и DAO.

В целях экономии времени, представим код только на Kotlin (он будет объяснен построчно для самостоятельной реализации на Java):

class UserRepository {

…

suspend fun getAllUsers(): List<UserInfo> { try {

val response = service.getUsers()

when (response.code()) { 200 -> {

usersDao.insertUsers(response.body()) return response.body()

}

else -> {

return usersDao.getLastUsers()

}

}

} catch (t: Throwable) {

return usersDao.getLastUsers()

}

}

В примере имеется прерываемая функция, которая точно возвраща- ет список пользователей.

suspend fun getAllUsers(): List<UserInfo> {

Вся обработка завернута в try, чтобы отловить возможные исклю- чения (именно такая реализация делается в целях демонстрации, можно отлавливать исключения любым удобным способом):

Здесь получаем список пользователей с сервера:

val response = service.getUsers()

Пример представлен на Kotlin, поэтому код остановится на этой строчке, пока не придут данные или не будет выброшено исключение. Исключение может быть выброшено, например, в случае если нет под- ключения к интернету или что-то произошло с доменным именем.

Далее обращаемся к методу code объекта response. Обратите вни- мание, что в данной реализации он завернут в обёртку Response из биб- лиотеки Retrofit. Для Java-ориентированных: when – аналог switch. В дан- ном фрагменте просто рассматриваем все возможные варианты.

200 -> {

usersDao.insertUsers(response.body()) return response.body()

}

При получении кода «200» записываем результат в базу данных:

usersDao.insertUsers(response.body())

Затем возвращаем актуальный результат для последующего отоб- ражения в UI:

return response.body()

При ином результате (равно неуспешном), возвращаем последний список пользователей, которых удалось вытянуть (важно перезаписывать этот результат в базу, а не добавлять):

return usersDao.getLastUsers()

Если попадаем в обработчик исключений, то также возвращаем по- следних юзеров. В идеале к ним нужно приложить сообщение об ошибке, но для упрощения этого делать не будем.

Основываясь на подобной логике, можно строить «offline first» приложения или просто кэшировать последние данные.

## Контрольные вопросы

1. Что такое Room?
2. Из чего состоит Room?
3. Какие зависимости необходимо добавить при работе с Room?
4. Основные пункты создания базы данных с Room.
5. Что такое «сущность»?
6. Как создать таблицу в Room?
7. Что такое DAO?
8. Что такое аннотации?

59

1. Какие аннотации запросов бывают?
2. Каким образом инициализируется БД?
3. Что такое паттерн «Репозиторий»?
4. Использование паттерна «Репозиторий».
5. Использование паттерна «ViewModel».
6. Логика кэширования с использованием Room.
7. Реализация логики кэширования с использованием Room.

# РАБОТА С POSTMAN

*4. Работа с Postman*

Postman – это инструмент для тестирования API (Application Pro- gramming Interface). В Android-разработке Postman часто используют для проверки и валидации данных, которые приходят с сервера, а также для последующего анализа этих данных с целью проектирования структуры приложения [27].

## Установка Postman

Postman – открытое ПО. Можно зайти на сайт продукта (https:// [www.postman.com/downloads/)](http://www.postman.com/downloads/%29) и скачать её для любой из популярнейших сегодня платформ (macOS, Windows, Linux).

После установки увидите окно с созданием аккаунта (рис. 3). Обра- тите внимание, что продолжить работу с программой можно без автори- зации. Единственное на что это влияет – синхронизация данных, что часто бывает очень полезным.



**Рис. 3.** Окно авторизации

## Начало работы с Postman

После запуска приложения увидите окно приветствия (рис. 4). Можно заметить, что структура интерфейса похожа на браузер. По нажа- тию на «+» откроется вкладка с набором полей.



**Рис. 4.** Окно Postman с открытой вкладкой

Слева расположена панель с историей и структурой папок сохра- ненных запросов, пока что не будем делать на ней акцент.

Сверху расположено меню инструментов. Кнопка «New» открыва- ет контекстное меню с настройкой запроса.

В комбо-боксе с выбранным GET доступен список из методов HTTP-запроса: GET, POST, PUT, DELETE и т.д. Подробнее с HTTP- запросами можно ознакомиться на сайте Mozilla [28].

Следующее поле – адресная строка.

Кнопка «Send» отправляет запрос, кнопка «Save» сохраняет конфи- гурацию запроса.

Ниже располагается несколько «табов»:

* Params – параметры запроса типа ключ-значение.
	+ Authorization – вкладка для точечной настройки заголовка ав- торизации, может пригодиться для выполнения авторизации с помощью Bearer или OAuth.
	+ В Header можно увидеть заголовки (некоторые генерирует сам Postman) и указать свои (рис. 5).
	+ Body – работа с телом запроса.
	+ Pre-request script – скрипт, который выполняется перед запро-

сом.

 Test – скрипт для тестирования запроса.



**Рис. 5.** Заголовки, сгенерированные Postman

## Отправляем первый запрос через Postman

Для тестирования можно воспользоваться бесплатным API Random User – https://randomuser.me/.

Отправим обычный GET-запрос по эндпоинту «api» (рис. 6). URL за- просы будут выглядеть так: «https://randomuser.me/api/». Подробная инфор- мация о бесплатном API находится на сайте в разделе документации [29].



**Рис. 6.** Отправка GET-запроса

После отправки запроса в панель «Request» вернется тело запроса. В данном случае это JSON-объект, но если отправите запрос, например, к стартовой странице Google, то вернется HTML-разметка.

В панели также есть несколько параметров состояния:

* Status – статус запроса с кодом (например, 404, если страница не найдена).
* Time – время выполнения запроса, очень полезно знать, сколь- ко выполняется запрос к серверу, чтобы контролировать быстродействие приложения.
* Size – размер возвращаемых данных.

### *Тело запроса в Postman*

При переключении на вкладку «Body» доступны несколько чек- боксов с указанием различных данных: form-data и urlencoded с ключ- значением, binary с указанием бинарного файла, GraphQL (рис. 7).

Если хотите отправить JSON-объект, выберите чек-бокс «raw» и в комбо-боксе справа выберите JSON.



**Рис. 7.** Параметры для тела запроса

## Настройка среды Postman. Сохранение запросов в коллекции

При работе над проектом, в котором много запросов или много до- менов, такие запросы можно объединять в коллекции. Преимущество коллекций заключается в том, что с их помощью можно вести локальную документацию: можно задавать свои имена запросам, писать описание, создавать переменные (например, с одинаковыми параметрами для всех запросов, вроде базового URL), добавлять общую логику схемы авториза- ции и даже писать скрипты.

Ниже представлена стандартная коллекция запросов в Postman (рис. 8).

Из рис. 8 видно, что коллекция объединена общим названием (в данном случае названием проекта) и содержит в себе перечень запросов с названием и описанием, которые задает пользователь.

Чтобы создать коллекцию, нажмите кнопку «New Collections» во вкладке «Collections». Откроется меню с формой заполнения информации о коллекции (рис. 9).



**Рис. 8.** Вид «коллекций» в интерфейсе



**Рис. 9.** Форма заполнения информации о коллекции

В форме можно ввести имя, задать описание и общую схему авто- ризации и создать переменные среды, видимые только в этой коллекции.

Назовем новую коллекцию, например, «Random User», зададим ба- зовое описание и первую переменную. Перейдем во вкладку «Variables» (рис. 10) и назовем переменную BASE\_URL, которая будет хранить URL нашей API. В случае, когда будет много запросов, URL можно будет по- менять в одном месте, если сервер внезапно переедет на другой домен.



**Рис. 10.** Вкладка «Variables»

Теперь перейдем к запросу со случайными пользователями, нажмем кнопку «Save» и выберем созданную коллекцию.

В открывшемся окне для запроса можно выбрать имя и описание.

Теперь URL в запросе можно заменить на вариант с переменной (рис. 11).

{{BASE\_URL}}api/



**Рис. 11.** Запрос с переменной

Переменные также можно создавать и редактировать в окне «Environ- ment», нажав на глаз в правом верхнем углу окна (рис. 12). Дополнительная информация о переменных находится в официальной документации [30].



**Рис. 12.** Окно «Environment»

## Контрольные вопросы

67

1. Что такое Postman?
2. Для чего используется Postman?
3. Установка Postman.
4. Какие возможности имеются в интерфейсе Postman?
5. Из каких вкладок состоит интерфейс Postman?
6. Как отправляются запросы в Postman?
7. Статусы запроса.
8. Что можно отправлять в теле запроса?
9. Преимущества коллекций запросов?
10. Для чего запросы объединяются в коллекции?
11. Как создать коллекцию запросов в Postman?

# УВЕДОМЛЕНИЯ

*5. Уведомления*

Уведомления – единственный способ взаимодействия с пользова- телем без привязки к интерфейсу вашего приложения. Уведомления пред- ставляют из себя небольшие блоки информации (текст, картинка, состоя- ние), которые поступают от определенного приложения или системы для оповещения об изменении состояния системы. В последних версиях An- droid уведомления помимо оповещений позволяют пользователю выпол- нить короткие действия: ответить на сообщение, управлять музыкой или видео, подтвердить какое-либо действие [31].

В данном разделе рассмотрим наиболее популярные и универсаль- ные варианты использования уведомлений [32].

## Создание уведомления

Первый шаг к созданию уведомления – реализация объекта Notifi- cation. Уведомления «строятся» при помощи fluent-интерфейса Notifica- tion.Builder [32].

Единственное требование к показу уведомления – наличие иконки (small icon). Можно взять иконку, которая генерируется для вашего про- екта – ic\_launcher\_foreground, так пользователь поймет, что уведомление сгенерировало именно ваше приложение. В целом, можно использовать любую SVG-иконку, главное, чтобы она была монохромной.

Рассмотрим немного продвинутую структуру показанного «пу- зырька» уведомления (рис. 13).



**Рис. 13.** Структура уведомления

Рассмотрим содержимое уведомления:

* 1. иконка, которая задается методом setSmallIcon(res);
	2. название приложения, которое генерируется автоматически, скрыть или изменить его невозможно (естественно, в целях безопасности);
	3. время отправки, по умолчанию оно тоже генерируется автома- тически, но можно указать время при помощи setWhen();
	4. большая иконка, которая задается параметром setLargeIcon();
	5. заголовок уведомления, задающийся через setContentTitle();
	6. содержание уведомления, задающееся через setContentText(). Код такого уведомления будет выглядеть достаточно просто (ис-

ключим большую картинку для наглядности):

val notification = NotificationCompat.Builder(context, CHANNEL\_ID)

.setSmallIcon(R.drawable.ic\_launcher\_foreground)

.setContentTitle(title)

.setContentText(body)

.build()

Для Java код будет выглядеть абсолютно идентичным с использо- ванием того же самого паттерна method chaining.

Константа CHANNEL\_ID – любая строка, необходимая для иден- тификации канала уведомлений.

Метод setSmallIcon принимает ссылку на ресурс с иконкой. Если не будет задана иконка или будет выбран неподходящий формат, то уведом- ление не покажется, в худшем случае – приложение «вылетит».

Методы setContentTitle принимают строку или ссылку на ресурс. Если приходит слишком большое сообщение, также можно определить для него текст в «развернутом» виде (по умолчанию покажется первая строка с многоточием в конце):

.setStyle(NotificationCompat.BigTextStyle()

.bigText(message))

Подробнее про метод setStyle и доступные стили для кастомизации уведомлений доступно в официальной документации [33].

Метод build возвращает объект типа Notification или Notifica- tionCompat, т.е. «билдит» объект, который можно использовать для показа уведомлений.

## Показ уведомления

После создания уведомления, его нужно показать. Библиотека An- droid предоставляет класс NotificationManagerCompat для показа уведом- лений [34]. На Kotlin код можно представить следующим образом:

with(NotificationManagerCompat.from(this)) { notify(notificationId, builder.build())

}

Аналогичный код на Java:

NotificationManagerCompat notificationManager = NotificationManagerCom- pat.from(this);

notificationManager.notify(notificationId, builder.build());

Первый параметр, который принимает NotificationManager – иден- тификатор уведомления, необходимый, чтобы отделить одно уведомление от другого. Это может быть ID, который отправляется с сервера или лич- ный локальный идентификатор. Два уведомления с одним идентификато- ром показать нельзя, одно заменится другим. Часто происходит так, что каждое приходящее уведомление уникально. Для этого можно воспользо- ваться достаточно популярным в программировании уникальным иденти- фикатором – текущим временем:

System.currentTimeMillis()

## Каналы уведомлений

Каналы уведомлений были представлены в 8 версии Android Oreo [31], они позволяют пользователю настроить каждый тип уведомлений в приложении отдельно (например, выбрать показ только сообщения о до- ставке от AliExpress, игнорируя промо - акции). Разработчики должны четко разделять уведомления на темы, иначе пользователь сможет только заблокировать все.

Для реализации каналов уведомлений не требуется изменять ста- рый код, нужно добавить новый спецификатор для версии:

private fun createNotificationChannel() {

if (Build.VERSION.SDK\_INT >= Build.VERSION\_CODES.O) {

val name = getString(R.string.channel\_name)

val descriptionText = getString(R.string.channel\_description)

val importance = NotificationManager.IMPORTANCE\_DEFAULT val channel = NotificationChannel(CHANNEL\_ID, name, im-

portance).apply {

description = descriptionText

}

val notificationManager: NotificationManager = getSystemService(Context.NOTIFICATION\_SERVICE) as Notifica-

tionManager

notificationManager.createNotificationChannel(channel)

}

}

Или для Java:

private void createNotificationChannel() {

if (Build.VERSION.SDK\_INT >= Build.VERSION\_CODES.O) {

CharSequence name = getString(R.string.channel\_name); String description = getString(R.string.channel\_description);

int importance = NotificationManager.IMPORTANCE\_DEFAULT; NotificationChannel channel = new NotificationChannel(CHANNEL\_ID,

name, importance); channel.setDescription(description);

NotificationManager notificationManager = getSystem- Service(NotificationManager.class);

notificationManager.createNotificationChannel(channel);

}

}

Здесь стандартно определяется код для версии после Android O.

В данном примере определяется для канала ID (тот же самый, ко- торый был использован ранее), имя канала (буквально название, напри- мер, «Промо», «Сообщения»), важность канала (подробнее смотрите в официальной документации [34]) и описание (не требуемый параметр).

Далее этот канал передаем через createNotificationChannel объекта типа NotificationManager.

## Взаимодействие пользователя с уведомлением

Есть несколько способов, как регистрировать на нажатия пользо- вателем на уведомления. Самый популярный – использование Pend- ingIntent [35].

Инициализируется PendingIntent через метод getActivity, который принимает контекст, идентификатор запроса (для возможности опреде- лить источник intent в broadcast receiver, если он используется), интент и флаги (в нашем случае флаги будут задаваться в самом интенте).

Реализация PendingIntent для уведомления:

val intent = Intent(this, AlertDetails::class.java).apply { flags = Intent.FLAG\_ACTIVITY\_NEW\_TASK or In-

tent.FLAG\_ACTIVITY\_CLEAR\_TASK

}

val pendingIntent: PendingIntent = PendingIntent.getActivity(this, 0, intent, 0)

Или для Java:

Intent intent = new Intent(this, AlertDetails.class); intent.setFlags(Intent.FLAG\_ACTIVITY\_NEW\_TASK | In- tent.FLAG\_ACTIVITY\_CLEAR\_TASK);

PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, intent, 0);

Передаем ноль для requestId, так как не будем использовать Broad- cast receiver. А также ноль для флагов, так как будем создавать новую ac- tivity вне текущего контекста.

Добавляем PendingIntent в уведомление:

val notification = NotificationCompat.Builder(context, CHANNEL\_ID)

.setSmallIcon(R.drawable.***ic\_launcher\_foreground***)

.setContentTitle(title)

.setContentText(body)

.setContentIntent(pendingIntent)

.build()

NotificationCompat.Builder builder = new NotificationCompat.Builder (context, CHANNEL\_ID)

.setSmallIcon(R.drawable.***ic\_launcher\_foreground***)

.setContentTitle(title)

.setContentText(body)

.setContentIntent(pendingIntent)

.build();

## Контрольные вопросы

1. Для чего используются уведомления в Android приложениях?
2. Из чего состоит уведомление?
3. Как создать уведомление?
4. Методы показа уведомления пользователю.
5. Какие методы имеются у объекта Notification?
6. Что такое каналы уведомлений?
7. Какие параметры можно настроить при создании канала уве- домлений?

73

1. Как реализовать PendingIntent?
2. Что такое PendingIntent?

# АРХИТЕКТУРНЫЕ КОМПОНЕНТЫ ANDROID

*6. Архитектурные компоненты Android*

Архитектурные компоненты Android (Android Architecture Compo- nents) – набор библиотек, рекомендуемый разработчиками из Google и помогающий создавать надежные, тестируемые и легкие в поддержке приложения.

Архитектурные компоненты включатся в Android Jetpack – набор ре- комендаций для разработки современных приложений (преимущественно на Kotlin). Список компонентов Android Jetpack достаточно большой, при- ведем лишь некоторые наиболее интересные из них:

* Android KTX (набор расширений для Kotlin);
* Data Binding [12];
* View Binding [13];
* Lifecycles;
* LiveData [36];
* Navigation (современная замена классической Android- навигации);
* Room;
* ViewModel [37].

Некоторых из компонентов были рассмотрены в предыдущих раз- делах, ниже будет рассмотрены ViewModel, LiveData, View Binding, Data Binding.

## ViewModel

По данным из официальной документации Android Developers, ViewModel – это класс, разработанный для хранения данных связанных с UI и управления ими с ориентацией на жизненный цикл Android. View- Model позволяет данным «выживать» при различных изменениях (напри- мер, изменения ориентации экрана) [37].

Если перефразировать, ViewModel – класс, который гарантирован- но не будет уничтожен, пока не уничтожится привязанная к нему Activity (или Fragment).

Архитектурный компонент состоит из следующих классов: ViewModel, AndroidViewModel, ViewModelProvider, ViewModelProviders,

ViewModelStore, ViewModelStores. Разработчики могут столкнуться с ра- ботой только ViewModel, AndroidViewModel и ViewModelProvier.

Компонент ViewModel повторяет название средней прослойки из архитектурного паттерна MVVM (Model-View-ViewModel). Сообщество разработчиков Android рекомендует использовать именно этот паттерн для ваших приложений.



**Рис. 14.** Схема паттерна MVVM

На рис. 14 приведена схема архитектурного паттерна MVVM с привязкой к компонентам, которые рекомендуют разработчики Android (LiveData, Room, Retrofit). Каждый из этих компонентов представлен в данном учебном пособии.

Связка ViewModel + LiveData дает классическое представление паттерна MVVM, но с привязкой к специфике разработки под Android, например, привязка к жизненному циклу приложения и переключение контекста из IO-потока в UI-поток.

Можно реализовать MVVM и без использования данных компо- нентов. ViewModel можно заменить обычным классом с логикой, который инициализируется в Activity (т.е. в прослойке View), но ничего про эту Activity не знает (то туда не передаются контекст, ссылка на класс или какой-либо интерфейс для взаимодействия с View). А взаимодействия с View, использовать что-нибудь вроде классического паттерна Observer, чем по большому счету и является LiveData.

На рис. 15 представлена работа компонента VIewModel в жизнен- ном цикле Activity.



**Рис. 15.** Работа компонента VIewModel в жизненном цикле Activity

### *Использование ViewModel на практике*

Для начала, требуется указать его как зависимость в Gradle: implementation "androidx.lifecycle:lifecycle-extensions:2.2.0" Дополнительно для Kotlin (не забудьте подключить плагин kapt):

kapt "androidx.lifecycle:lifecycle-compiler:2.2.0"

implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:2.2.0"

Или для Java:

annotationProcessor "android.arch.lifecycle:compiler:2.2.0"

Как уже упоминалось, архитектурные компоненты предоставляют класс ViewModel, который можно унаследовать для того, чтобы обозна- чить класс как ViewModel.

Пример на Kotlin:

class MyViewModel : ViewModel() {

private val users: MutableLiveData<List<User>> by lazy { MutableLiveData().also {

loadUsers()

}

}

fun getUsers(): LiveData<List<User>> { return users

}

private fun loadUsers() {

// Выполнение асинхронной операции

}

}

Пример на Java:

public class MyViewModel extends ViewModel { private MutableLiveData<List<User>> users; public LiveData<List<User>> getUsers() {

if (users == null) {

users = new MutableLiveData<List<User>>(); loadUsers();

}

return users;

}

private void loadUsers() {

// Выполнение асинхронной операции

}

}

Данные примеры отражают надежную архитектурную прослойку, которая может хранить данные и управлять ими независимо от того, что происходит с жизненным циклом приложения [38].

Для того чтобы инициализировать ViewModel в Activity не полу- чится просто выделить память под класс MyViewModel. Для этого нужно воспользоваться классом ViewModelProvider, который выделит память под класс и привяжет его к Activity.

На Kotlin допустимо не использовать ViewModelProvider, а исполь- зовать делегат (про делегаты подробнее можете узнать в официальной документации Kotlin [1]):

class MyActivity : AppCompatActivity() {

override fun onCreate(savedInstanceState: Bundle?) { val model: MyViewModel by viewModels()

model.getUsers().observe(this, Observer<List<User>>{ users ->

// обновляем UI

})

}

}

Пример на Kotlin с использованием ViewModelProvider:

class MyActivity : AppCompatActivity() {

override fun onCreate(savedInstanceState: Bundle?) {

val model = ViewModelProvider(this).get(MyViewModel::class.java) model.getUsers().observe(this, Observer<List<User>>{ users ->

// обновляем UI

})

}

}

Аналогичный пример на Java:

public class MyActivity extends AppCompatActivity { public void onCreate(Bundle savedInstanceState) {

// Create a ViewModel the first time the system calls an activity's onCre- ate() method.

// Re-created activities receive the same MyViewModel instance created by the first activity.

MyViewModel model = new ViewModelProvid- er(this).get(MyViewModel.class);

model.getUsers().observe(this, users -> {

// update UI

});

}

}

Обратите внимание, что инициализировать ViewModel требуется, когда Activity вызывает метод onCreate или onCreateView в случае с Fragment.

## LiveData

Согласно официальной документации Android, LiveData предна- значен для хранения объекта и позволяет подписаться на его изменения. Как и в случае с ViewModel, данный архитектурный компонент осведом- лен о жизненном цикле приложения, что позволяет разработчику не забо- титься о потенциальных утечках памяти в связи с этой спецификой рабо- ты Android [36].

Архитектурный компонент состоит из классов LiveData, Mutable- LiveData, MediatorLiveData, LiveDataReactiveStreams, Transformations, а также интерфейса Observer.

Почему не использовать обычный паттерн Observer? LiveData дает много преимуществ в контексте Android-разработки. Документация An- droid Developers приводит следующие достоинства по использованию LiveData [36]:

* гарантирует, что отображение в UI соответствует текущему состоянию данных;
* не допускает утечек памяти, т.е. разрушается тогда же, когда разрушается activity;
* разработчик может описать свой собственный LiveData- объект, что позволяет его переиспользовать для похожих случаев в при- ложении;
* обновляет UI после изменений состояния activity или fragment.

### *Использование LiveData*

Для начала, требуется указать его как зависимость в Gradle: implementation "androidx.lifecycle:lifecycle-extensions:2.2.0" Дополнительно для Kotlin (не забудьте подключить плагин kapt): kapt "androidx.lifecycle:lifecycle-compiler:2.2.0"

implementation "androidx.lifecycle:lifecycle-livedata-ktx:2.2.0"

Или для Java:

annotationProcessor "android.arch.lifecycle:compiler:2.2.0"

### *Инициализация LiveData во ViewModel*

Рассмотрим базовый пример использования LiveData на практике. LiveData – это обёртка, которая может использоваться поверх абсолютно любых данных. В данном примере приведен тип String.

class NameViewModel : ViewModel() {

val currentName: MutableLiveData<String> by lazy { MutableLiveData<String>()

}

}

На Java это будет выглядеть следующим образом:

public class NameViewModel extends ViewModel { private MutableLiveData<String> currentName;

public MutableLiveData<String> getCurrentName() { if (currentName == null) {

currentName = new MutableLiveData<String>();

}

return currentName;

}

}

И примера видно, что в контейнере LiveData хранится строка. Изме- нения контейнера можно привязать к различным событиям: когда данные возвращаются с сервера, когда данные возвращаются из БД, после вычис- ления трудоёмких операций, после обычной задержки (delay или sleep).

### *Подписка на LiveData во View*

Далее на события созданной LiveData необходимо подписаться, и сделать это обязательно в onCreate (или onCreateView/onViewCreated в случае с фрагментом). Представим код стандартной Activity с LiveData- подписчиком.

class NameActivity : AppCompatActivity() {

private val model: NameViewModel by viewModels() override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

// …

val nameObserver = Observer<String> { newName ->

// Обновляем UI, например, TextView nameTextView.text = newName

}

model.currentName.observe(this, nameObserver)

}

}

Или на Java:

public class NameActivity extends AppCompatActivity { private NameViewModel model;

@Override

protected void onCreate(Bundle savedInstanceState) { super.onCreate(savedInstanceState);

// …

model = new ViewModelProvider(this).get(NameViewModel.class); final Observer<String> nameObserver = new Observer<String>() {

@Override

public void onChanged(@Nullable final String newName) { nameTextView.setText(newName);

}

};

model.getCurrentName().observe(this, nameObserver);

}

}

Обработка данных происходит в теле лямбды в случае с Kotlin или в *onChanged* в случае с Java:

val nameObserver = Observer<String> { newName ->

// Обновляем UI, например, TextView nameTextView.text = newName

}

final Observer<String> nameObserver = new Observer<String>() { @Override

public void onChanged(@Nullable final String newName) { nameTextView.setText(newName);

}

};

Код, описанный в представленных выше блоках, будет выполнять- ся каждый раз, когда данные в LiveData-контейнере будут изменены.

### *Обновление данных LiveData*

Как уже упоминалось выше, можно изменять данные по любым событиям. Чтобы не задавать обновление этих событий вручную, ис- пользуйте метод setValue(T), доступный у класса MutableLiveData из примера выше.

Например, можно изменить данные по событию нажатия кнопки:

button.setOnClickListener {

val anotherName = "John Doe" model.currentName.setValue(anotherName)

}

Данный пример на Java:

button.setOnClickListener(new OnClickListener() { @Override

public void onClick(View v) {

String anotherName = "John Doe"; model.getCurrentName().setValue(anotherName);

}

});

Использование LiveData в комбинации с Retrofit и Room были рас- смотрены в разд. 2 и 3 соответственно.

## View Binding

View Binding (представление привязки) является функцией, облег- чающей написание программного кода, который взаимодействует с вью. Как только View Binding включена в модуле, она генерирует binding класс

для каждого файла макета XML, присутствующего в этом модуле. Экзем- пляр binding класса содержит прямые ссылки на все вью, которые имеют идентификатор в соответствующем макете.

В большинстве случаев привязка представления заменяет функцию findViewById.

View Binding имеет существенные преимущества перед использо- ванием findViewById.

* null безопасность. Поскольку привязка вью создает прямые ссылки на вью, нет риска исключения указателя null из-за недействитель- ного идентификатора вью. Кроме того, когда вью присутствует только в некоторых конфигурациях макета, поле, содержащее его ссылку в binding классе, отмечено (@Nullable);
* безопасность типов. Поля в каждом binding классе имеют ти- пы, соответствующие вью, на которые они ссылаются в файле XML. Это означает, что нет риска исключения из класса.

Представленные выше различия говорят о том, что несовмести- мость между макетом и кодом приведут к тому, что сборка проекта не сработает во время компиляции, а не во время выполнения.

### *Настройка установки*

Для того чтобы включить View Binding в модуле, установите оп- цию сборки в файле уровня модуля, как показано в следующем примере: viewBinding true build.gradle.

android {

...

buildFeatures { viewBinding true

}

}

Если требуется, чтобы файл макета был проигнорирован при гене- рации binding классов, добавьте атрибут в корень этого файла макета: tools: viewBindingIgnore="true".

<LinearLayout

...

tools:viewBindingIgnore="true" >

...

</LinearLayout>

### *Использование View Binding*

Если View Binding включен для модуля, то для каждого файла ма- кета XML, который содержит модуль, генерируется binding класс. Каж- дый binding класс содержит ссылки на корневую вью и все вью, которые имеют идентификатор. Название класса связывания генерируется путем преобразования имени файла XML в Паскаль и добавления слова "Binding" в конце.

Например, возьмем файл макета под названием: result\_profile.xml.

<LinearLayout ... >

<TextView android:id="@+id/name" />

<ImageView android:cropToPadding="true" />

<Button android:id="@+id/button" android:background="@drawable/rounded\_button" />

</LinearLayout>

В макете нет идентификатора, поэтому в binding классе нет ссылки на него (ResultProfileBinding TextView name Button button ImageView).

Каждый binding класс также включает в себя метод, обеспечиваю- щий прямую ссылку на корневое вью соответствующего файла макета. В этом примере метод в классе возвращает корневое вью (getRoot() getRoot() ResultProfileBindingLinearLayout).

В следующих подразделах будет показано использование binding классов в Activity и фрагментах.

### *Использование View Binding в Activity*

Чтобы настроить экземпляр binding класса для использования в Ac- tivity, необходимо выполнить следующие шаги в методе onCreate ():

* вызвать статический метод, включенный в сгенерированный binding класс, для того, чтобы создать экземпляр binding класса для ис- пользования в Activity (inflate());
* получить ссылку на корневой вью, вызвав метод getRoot();
* перейти в корневой вью с использованием setContentView(), чтобы сделать его активным вью на экране.

Пример на Kotlin:

private lateinit var binding: ResultProfileBinding override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

binding = ResultProfileBinding.inflate(layoutInflater) val view = binding.root

setContentView(view)

}

Пример на Java:

private ResultProfileBinding binding; @Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

binding = ResultProfileBinding.inflate(getLayoutInflater()); View view = binding.getRoot();

setContentView(view);

}

Теперь можно использовать экземпляр binding класса для ссылки на любой из вью:

Пример на Kotlin:

binding.name.text = viewModel.name binding.button.setOnClickListener { viewModel.userClicked() }

Пример на Java:

binding.getName().setText(viewModel.getName()); binding.button.setOnClickListener(new View.OnClickListener() {

viewModel.userClicked()

});

### *Использование View Binding в фрагментах*

Чтобы настроить экземпляр binding класса для использования во фраг- ментах, необходимо выполнить следующие шаги в методе onCreateView():

* вызвать статический метод, включенный в сгенерированный binding класс, для того, чтобы создать экземпляр binding класса для ис- пользования фрагмента (inflate());
	+ получить ссылку на корневой вью, используя метод getRoot();
	+ получить корневой вью из метода onCreateView(), чтобы сде- лать его активным вью на экране.

Обратите внимание, что метод inflate() требует, чтобы был доступ через «лояут инфлейтер». Если макет уже «заинфлейчен», то можно вме- сто этого вызвать статический метод binding класса bind().

Пример на Kotlin:

private var \_binding: ResultProfileBinding? = null

// This property is only valid between onCreateView and

// onDestroyView.

private val binding get() = \_binding!!

override fun onCreateView( inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?

): View? {

\_binding = ResultProfileBinding.inflate(inflater, container, false) val view = binding.root

return view

}

override fun onDestroyView() { super.onDestroyView()

\_binding = null

}

Пример на Java:

private ResultProfileBinding binding;

@Override

public View onCreateView (LayoutInflater inflater,

ViewGroup container, Bundle savedInstanceState) {

binding = ResultProfileBinding.inflate(inflater, container, false); View view = binding.getRoot();

return view;

}

@Override

public void onDestroyView() {

super.onDestroyView(); binding = null;

}

Теперь можно использовать экземпляр binding класса для ссылки на любой из вью:

Пример на Kotlin:

binding.name.text = viewModel.name binding.button.setOnClickListener { viewModel.userClicked() }

Пример на Java:

binding.getName().setText(viewModel.getName()); binding.button.setOnClickListener(new View.OnClickListener() {

viewModel.userClicked()

});

### *Сравнение View Binding с Data Binding*

View Binding и Data Binding генерирует binding классы, которые можно использовать для прямых ссылок на вью. Тем не менее, View Bind- ing предназначена для обработки более простых случаев использования и обеспечивает следующие преимущества по отношению к Data Binding:

* более быстрая компиляция. View Binding не требует обработки аннотации, поэтому время компиляции происходит быстрее;
* простота в использовании. View Binding не требует специаль- но помеченных файлов макета XML, поэтому их быстрее использовать в приложениях. Как только будет включена View Binding в модуле, то из- менения применятся ко всем макетам этого модуля автоматически.

И наоборот, View Binding имеет следующие ограничения по срав- нению с Data Binding:

* View Binding не поддерживает переменные макета или выра- жения макета, поэтому он не может быть использован для декларирования динамического содержимого пользовательского интерфейса прямо из файлов макета XML;
* View Binding не поддерживает two-way data binding [38].

Учитывая перечисленные выше особенности, в некоторых случаях лучше всего использовать view binding и data binding в проекте. Вы може-

те использовать data binding в макетах, требующих расширенных функ- ций, и view binding в макетах, которые этого не делают.

## Data Binding

Библиотека Data Binding является библиотекой поддержки, которая позволяет связывать компоненты пользовательского интерфейса в маке- тах с источниками данных в приложении, используя декларативный фор- мат, а не программно.

Макеты часто определяются в Activity с кодом, который называется методами фреймворка. Например, код ниже gзволяет найти виджет и при- вязать его к свойству переменной:

findViewById() TextView userName viewModel

Пример на Kotlin:

findViewById<TextView>(R.id.sample\_text).apply { text = viewModel.userName

}

Пример на Java:

TextView textView = findViewById(R.id.sample\_text); textView.setText(viewModel.getUserName());

В следующем примере показано, как использовать библиотеку Data Binding для назначения текста виджету непосредственно в файле макета. Это устраняет необходимость вызова любого из представленного выше кода. Обратите внимание на использование синтаксиса в выражении назначения @{}:

<TextView android:text="@{viewmodel.userName}" />

Связывание компонентов в файле макета позволяет удалить многие вызовы фреймворка в Activity, делая их проще и легче в поддержке. Кро- ме того, это может повысить производительность приложения и предот- вратить утечки памяти и исключения из указателей.

Для начала работы с Data Binding необходимо включить опцию сборки в файле в модуле приложения, как показано в следующем приме- ре: dataBinding build.gradle

android {

...

buildFeatures { dataBinding true

}

}

### *Макеты и выражения binding*

Язык выражений позволяет писать выражения, которые обрабаты- вают события, отправленные вью. Библиотека Data Binding автоматически генерирует классы, необходимые для связывания вью в макете с объекта- ми данных [39].

Файлы компоновки связывания данных немного отличаются и начинаются с корневой метки последующего элемента и корневого эле- мента. В следующем коде показан образец файла макета layout data view:

<?xml version="1.0" encoding="utf-8"?>

<layout xmlns:android="<http://schemas.android.com/apk/res/android>">

<data>

<variable name="user" type="com.example.User"/>

</data>

<LinearLayout android:orientation="vertical" android:layout\_width="match\_parent" android:layout\_height="match\_parent">

<TextView android:layout\_width="wrap\_content" android:layout\_height="wrap\_content" android:text="@{user.firstName}"/>

<TextView android:layout\_width="wrap\_content" android:layout\_height="wrap\_content" android:text="@{user.lastName}"/>

</LinearLayout>

</layout>

Переменная внутри описывает свойство, которое может быть ис- пользовано в этом макете user data:

<variable name="user" type="com.example.User" />

Выражения в макете написаны в свойствах атрибута с помощью синтаксиса "". Здесь текст устанавливается на свойство переменной: @{}TextViewfirstNameuser

<TextView android:layout\_width="wrap\_content" android:layout\_height="wrap\_content" android:text="@{user.firstName}" />

### *Объекты данных*

Предположим, что имеется объект для описания сущности User. На Kotlin:

data class User(val firstName: String, val lastName: String)

На Java:

public class User {

public final String firstName; public final String lastName;

public User(String firstName, String lastName) { this.firstName = firstName;

this.lastName = lastName;

}

}

Этот тип объекта имеет данные, которые никогда не меняются. Обычно в приложениях есть данные, которые читаются один раз и нико- гда не меняются после этого. Также можно использовать объект, который следует набору конвенций, таких как использование методов доступа в Java, как показано в следующем примере.

На Kotlin:

data class User(val firstName: String, val lastName: String

На Java:

public class User {

private final String firstName; private final String lastName;

public User(String firstName, String lastName) { this.firstName = firstName;

this.lastName = lastName;

}

public String getFirstName() { return this.firstName;

}

public String getLastName() { return this.lastName;

}

}

С точки зрения связывания данных эти два класса эквивалентны. Выражение @{user.firstName}, используемое для атрибута android:text, получает доступ к полю firstName и методу getFirstName() последнего класса. Кроме того, допускается также firstName(), если этот метод существует.

### *Связывание данных*

Для каждого файла макета генерируется binding класс. По умолча- нию, имя класса основано на названии файла макета, преобразовании его в Паскаль и *добавлении к нему суффикс Binding.* Представленное выше имя макета – activity\_main.xml – соответствует генерируемому классу ActivityMainBinding. Этот класс содержит все привязки от свойств макета (например, переменную user) до вью макета и знает, как назначить значе- ния для связывающих выражений. Рекомендуемый метод создания при- вязки заключается в том, чтобы сделать это при «инфлейте» макета, как показано в следующем примере.

На Kotlin:

override fun onCreate(savedInstanceState: Bundle?) { super.onCreate(savedInstanceState)

val binding: ActivityMainBinding = DataBindingUtil.setContentView( this, R.layout.activity\_main)

binding.user = User("Test", "User")

}

На Java:

@Override

protected void onCreate(Bundle savedInstanceState) { super.onCreate(savedInstanceState);

ActivityMainBinding binding = DataBindingUtil.setContentView(this, R.layout.activity\_main);

User user = new User("Test", "User"); binding.setUser(user);

}

Во время выполнения приложение отображает пользователя Test в UI. Кроме того, можно получить представление с помощью LayoutInflater, как показано в следующем примере для Kotlin:

val binding: ActivityMainBinding = ActivityMainBind- ing.inflate(getLayoutInflater())

Пример на Java:

ActivityMainBinding binding = ActivityMainBind- ing.inflate(getLayoutInflater());

Если используется Data Binding внутри фрагмента, ListView или , адаптера RecyclerView, то предпочтительно использовать методы inflate() binding классов или класса DataBindingUtil, как показано в сле- дующем примере.

На Kotlin:

val listItemBinding = ListItemBinding.inflate(layoutInflater, viewGroup, false)

// or

val listItemBinding = DataBindingUtil.inflate(layoutInflater, R.layout.list\_item, viewGroup, false)

На Java:

ListItemBinding binding = ListItemBinding.inflate(layoutInflater, viewGroup, false);

// or

ListItemBinding binding = DataBindingUtil.inflate(layoutInflater, R.layout.list\_item, viewGroup, false);

### *Обработка событий*

Data Binding позволяет писать выражения обработки событий, ко- торые отправляются из вью (например, метод onClick()). Имена атрибутов событий определяются именем метода слушателя за некоторыми исклю- чениями. Например, View.OnClickListener использует метод onClick(), поэтому атрибутом для этого события является android:onClick.

Для обработки события можно использовать следующие механизмы:

* ссылки на метод;
* привязки слушателя.
	+ - 1. *Ссылки на метод*

События могут быть связаны с методами обработчика напрямую, подобно тому, как android:onClick может быть назначен методу в Activity. Одним из основных преимуществ по сравнению с атрибутом View onClick является то, что выражение обрабатывается во время компиляции, так что, если метод не существует или его подпись неверна, выдается ошибка времени компиляции.

Основное различие между ссылками на метод и привязками слуша- теля заключается в том, что фактическая реализация слушателя создается, когда данные связаны, а не когда событие срабатывает. Если требуется оценивать выражение, когда происходит событие, следует использовать привязку слушателя.

Чтобы назначить событие своему обработчику, используйте обыч- ное связывающее выражение, значение которого является именем метода для вызова. Например, рассмотрим следующий пример объекта данных макета на Kotlin:

class MyHandlers {

fun onClickFriend(view: View) { ... }

}

На Java:

public class MyHandlers {

public void onClickFriend(View view) { ... }

}

Binding выражение может назначить слушателя клика для просмот- ра вью методом onClickFriend() следующим образом:

<?xml version="1.0" encoding="utf-8"?>

<layout xmlns:android="<http://schemas.android.com/apk/res/android>">

<data>

<variable name="handlers" type="com.example.MyHandlers"/>

<variable name="user" type="com.example.User"/>

</data>

<LinearLayout android:orientation="vertical" android:layout\_width="match\_parent" android:layout\_height="match\_parent">

<TextView android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content" android:text="@{user.firstName}" android:onClick="@{handlers::onClickFriend}"/>

</LinearLayout>

</layout>

* + - 1. *Привязки слушателя*

Привязки слушателя являются binding выражениями, которые за- пускаются при событии. Они похожи на ссылки на метод, но позволяют запускать произвольные выражения data binding. Эта функция доступна с Android Gradle Plugin version Gradle 2.0 и позже.

В ссылках на метод параметры метода должны соответствовать па- раметрам слушателя события. В привязках слушателя только значение возврата должно соответствовать ожидаемому значению возврата слуша- теля (если только он не ожидает пустоты). Например, рассмотрим следу- ющий класс Presenter, который имеет метод onSaveClick().

Пример на Kotlin:

class Presenter {

fun onSaveClick(task: Task){}

}

На Java:

public class Presenter {

public void onSaveClick(Task task){}

}

Затем можно привязать событие клика к методу onSaveClick(), сле- дующим образом:

<?xml version="1.0" encoding="utf-8"?>

<layout xmlns:android="<http://schemas.android.com/apk/res/android>">

<data>

<variable name="task" type="com.android.example.Task" />

<variable name="presenter" type="com.android.example.Presenter" />

</data>

<LinearLayout android:layout\_width="match\_parent" an- droid:layout\_height="match\_parent">

<Button android:layout\_width="wrap\_content" an- droid:layout\_height="wrap\_content"

android:onClick="@{() -> presenter.onSaveClick(task)}" />

</LinearLayout>

</layout>

Когда в выражении используется обратный вызов, data binding ав- томатически создает необходимого слушателя и регистрирует его для со- бытия. Когда вью запускает событие, data binding оценивает данное вы- ражение. Как и в обычных data binding, по-прежнему получается null и потоковая безопасность data binding во время оценки этих выражений слушателя.

В приведённом выше примере не определен параметр вью, который передается onClick(View). Привязки слушателя предоставляют два вари- анта параметров слушателя: можно либо игнорировать все параметры метода, либо определить все из них. Если предпочтительнее определить параметры, то можно использовать их в своем выражении. Например, приведенное выше выражение может быть написано следующим образом:

android:onClick="@{(view) -> presenter.onSaveClick(task)}"

Если требуется использовать параметр в выражении, то можно за- писать следующим образом на Kotlin:

class Presenter {

fun onSaveClick(view: View, task: Task){}

}

android:onClick="@{(theView) -> presenter.onSaveClick(theView, task)}"

На Java:

public class Presenter {

public void onSaveClick(View view, Task task){}

}

android:onClick="@{(theView) -> presenter.onSaveClick(theView, task)}"

Также можно использовать лямбда выражения с более чем одним параметром.

Пример на Kotlin:

class Presenter {

fun onCompletedChanged(task: Task, completed: Boolean){}

}

<CheckBox android:layout\_width="wrap\_content" an- droid:layout\_height="wrap\_content"

android:onCheckedChanged="@{(cb, isChecked) -> present- er.completeChanged(task, isChecked)}" />

На Java:

public class Presenter {

public void onCompletedChanged(Task task, boolean completed){}

}

<CheckBox android:layout\_width="wrap\_content" an- droid:layout\_height="wrap\_content"

android:onCheckedChanged="@{(cb, isChecked) -> present- er.completeChang

Если событие, которое слушаете, возвращает значение, тип которо- го не является void, то выражения должны вернуть тот же тип значений. Например, если требуется прослушать событие с длинным щелчком мы- ши, выражение должно вернуть boolean.

На Kotlin:

class Presenter {

fun onLongClick(view: View, task: Task): Boolean { }

}

android:onLongClick="@{(theView) -> presenter.onLongClick(theView, task)}"

На Java:

public class Presenter {

public boolean onLongClick(View view, Task task) { }

}

android:onLongClick="@{(theView) -> presenter.onLongClick(theView, task)}"

Если выражение не может быть оценено из-за null объектов, data binding возвращает значение по умолчанию для этого типа. Например, data bindingдля эталонных типов, 0 для int, false для Boolean и т.д.

Если необходимо использовать выражение с предикатом (напри- мер, кратерный), вы можете использовать void в качестве символа:

android:onClick="@{(v) -> v.isVisible() ? doSomething() : void}"

### *Генерирование binding классов*

Сгенерированный binding класс связывания связывает переменные ма- кета с вью в макете. Имя и пакет binding класса могут быть настроены. Все генерируемые классы связывания наследуются от класса ViewDataBinding.

Для каждого файла макета генерируется binding класс. По умолча- нию имя класса основано на названии файла макета, преобразовании его в Паскал и добавлении к нему суффикс Binding. Вышеупомянутое кодовое имя файла так соответствует генерируемому классу. Этот класс содержит все привязки от свойств макета (например, переменной) до представлений макета и знает, как назначить значения для связывающих выражений activity\_main.xml ActivityMainBinding user.

Подробнее об использовании DataBinding можно ознакомиться в официальной документации [12].

## Контрольные вопросы

1. Что такое архитектурные компоненты Android?
2. Приведите основные компоненты Android Jetpack.
3. Что такое ViewModel?
4. Какие основные компоненты ViewModel?
5. Для чего используется связка ViewModel + LiveData?
6. Каким образом можно реализовать MVVM без использования архитектурных компонентов?
7. Добавление зависимостей для ViewModel.
8. Каким образом можно реализовать ViewModel?
9. Особенности реализации ViewModel на Kotlin.
10. Что такое LiveData?
11. Какие основные классы LiveData?
12. Приведите достоинства использования LiveData.
13. Добавление зависимостей для LiveData.
14. Инициализация LiveData во ViewModel.

98

1. Подписка на LiveData во View.
2. Как выполнить обновление данных LiveData?
3. Что такое View Binding?
4. Настройка View Binding.
5. Использование View Binding в Activity.
6. Использование View Binding в фрагментах.
7. В чем отличия View Binding с Data Binding?
8. Для чего используется Data Binding?
9. Макеты и выражения binding.
10. Объекты данных.
11. Связывание данных в Data Binding.
12. Какие методы обработки событий имеются в Data Binding?
13. Ссылки на метод в Data Binding.
14. Привязки слушателя в Data Binding.

# ТЕСТИРОВАНИЕ В ANDROID

*7.1. Unit-тестирование*

Тестирование – важная часть разработки программного обеспечения. Многие полагают, что тестированием чаще всего занимаются «тестировщи- ки», а само тестирование не имеет отношения к написанию кода програм- мы. Это вовсе не так, тестирование помогает разработчикам поддерживать код, что помогает найти ошибку или несостыковку в ситуации, когда код будет часто меняться. Часто в коммерческой разработке на тестирование тратят едва ли не столько же времени, сколько на бизнес-логику.

В данном разделе будут рассмотрены как обычные Unit-тесты, так и специфичные для мобильной разработки UI-тесты [41].

## Unit-тестирование

Перед началом тестирования необходимо установить небольшое количество различных инструментов. Стандартом в мире Java-разработки является JUnit. Данный инструмент будет встречаться как базовый почти в любом Java-бойлерплейте, в том числе в бойлерплейте, который генери- рует Android Studio.

Зависимость JUnit уже лежит в файле Gradle, но для наглядности укажем, что записывается она в файл build.gradle в dependencies:

testImplementation 'junit:junit:4.12' androidTestImplementation 'androidx.test.ext:junit:1.1.1'

androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'

Помимо неё подключим еще две зависимости непосредственно для Android-тестирования.

### *Разбор простейшего теста*

В Android-проекте тесты по умолчанию генерируются в папке тест. Если вы следовали инструкции по запуску базового приложения, то у вас ничего не изменится. Посмотрим на файл, который сгенерировала Android Studio в папке test:

class ExampleUnitTest { @Test

fun addition\_isCorrect() { assertEquals(4, 2 + 2)

}

}

public class ExampleUnitTest { @Test

public void addition\_isCorrect() { assertEquals(4, 2 + 2);

}

}

Из примера видно, что представлен обычный класс, но с необычной аннотацией. Аннотация @Test, очевидно, помечает функцию как тесто- вую. Метод assertEquals принадлежит библиотеке JUnit. Именно этот ме- тод, скорее всего, чаще будет использоваться на практике [42].

Метод assertEquals получает первым параметром ожидаемое значе- ние, а вторым параметром – реальную функцию.

Android Studio предоставляет удобный интерфейс для тестирова- ния. Запускать различные наборы тестов и конкретный тест можно с по- мощью боковой панели в IDE (рис. 16).



**Рис. 16.** Интерфейс для тестирования в Android Studio

После запуска данного базового теста откроется интерфейс с пе- речнем выполненных или проваленных тестов, которые соответственно помечаются галочкой или крестиком (рис. 17).

Если в assertEquals первый ожидаемый параметр заменим на 5, то тест, очевидно, провалится.

На самом деле, на этом базовом понятии «ожидание-реальность» строится вся основа тестирования проектов. При написании даже ком- плексных тестов нельзя забывать о том, что это всего лишь ожидаемое и реальное значение.



**Рис. 17.** Интерфейс выполненных и невыполненных тестов

### *Реализация первого Unit-теста*

Часто в проектах тестирование ограничивается тестированием раз- личных утилит, и, если в проекте используется «минимальное» тестиро- вание, то чаще всего речь идет именно об этом. В этом есть смысл только для первого тестирования, утилиты редко изменяются, только, если в них нашли «баг», который не был покрыт тестами.

Ниже рассмотрим пример так называемого «минимального по- крытия».

Предположим, что с сервера приходит строка вида «Александр

/Дугин», а дизайн требует, чтобы в интерфейсе это отображалось как

«Александр ДУГИН». При этом сервер не ответственен за количество пробелов в строке. Всё, что он может дать – гарантию того, что имя и фа- милия будут написаны с большой буквы.

По требованиям специальная утилита для данных действий будет иметь вид (чтобы не вносить путаницу, будет предоставлен пример толь- ко на языке Kotlin, так как код не требует специфичных знаний):

fun String.toValidName(): String {

val names = split("/").map { it.trimEnd().trimStart() } return "${names.first()} ${names.last().toUpperCase()}"

}

Первая строка делит строку по разделителю и «тримит» каждый элемент в начале и конце. Вторая строка с помощью интерполяции воз- вращает имя в формате «Имя ФАМИЛИЯ».

Проведем базовые тесты для этой утилиты. Прежде всего, нужно создать класс StringValidUtilTest в папке test (там же, где лежат сгенери- рованные тесты):

class StringUtilTest {

@Test

fun toValidName\_isCorrect() {

Assert.assertEquals("Виктор ПЕЛЕВИН", "Виктор / Пеле- вин".toValidName())

}

}

Функции для проверки рекомендуется именовать в формате «ис- ходнаяФункция\_названиеПроверки». В данном случае будет использо- ваться абстрактное имя isCorrect.

После «прогона» данного теста получим положительный результат. Добавим еще несколько тест-кейсов, а также вынесем ожидаемое значение в отдельную переменную. В таком случае значение выносят в

константу, но, в случае тестов, есть другие инструменты:

class StringUtilTest {

private lateinit var targetString: String

@Before

fun setUp() {

targetString = "Виктор ПЕЛЕВИН"

}

@Test

fun toValidName\_isCorrect() {

Assert.assertEquals(targetString, "Виктор / Пелевин".toValidName()) Assert.assertEquals(targetString, "Виктор / Пелевин ".toValidName())

}

}

В представленном выше коде внедрили новую функцию setUp. Пе- ред ней поставили аннотацию Before. Это значит, что функция будет вы- полняться перед каждым тест-кейсом. Если требуется, чтобы функция выполнилась один раз перед началом тестирования в классе, то нужна

аннотация @BeforeClass. Если требуется выполнить какие-либо действия после выполнения тестов, используйте аннотацию @After.

В нашем случае можно обойтись и константой, но в тестах так бы- вает, что тестовые данные деформируются или изменяются после каждой тестовой итерации, поэтому перед каждым тестом их нужно приводить к единому виду.

Также стоит отметить, что две assert-функции в одном тест-кейсе – не достаточно верно. Разделение поможет улучшить читаемость и найти ошибку быстрее.

В разработанной утилите isCorrect предусмотрены случаи положи- тельного результата. Но что будет, если будет передана не валидная стро- ка? Для начала нужно предусмотреть это в утилите. Предположим, что при невалидной строке будет возвращаться не строка, а null

Обновим утилиту isCorrect:

fun String.toValidName(): String? {

val names = split("/").map { it.trimEnd().trimStart() }

if (names.size > 2 || names.isEmpty() || names.any { it == "" }) return null return "${names.first()} ${names.last().toUpperCase()}"

}

Из примера видно, что исключительные ситуации – размер массива больше двух, если имя пустое или сам массив с именами пуст. Протести- руем также исключительные ситуации.

@Test

fun toValidName\_isInvalid() { Assert.assertEquals(null, " / ".toValidName()) Assert.assertEquals(null, " / Test".toValidName())

Assert.assertEquals(null, "Дугин / Александр / ".toValidName())

}

Во всех эти случаях будет получен null и тесты будут пройдены.

### *Mockito и «моки»*

Представленный в подразд. 7.1.2 тест не использовал никаких сложных данных: передавали обычную строку и передавали строку или null. А что необходимо делать, если нужно симулировать не строку, а це- лый класс сложности Retofit-сервиса? Для этого нужно «мокать» данные, т.е. заворачивать класс в такую обертку, чтобы не пришлось буквально

создавать его экземпляр, а сразу получать нужные данные. Ведь, в дей- ствительности, для тестирования не нужна настолько детальная проработ- ка. Как уже упоминалось ранее, все тесты сводятся к модели «ожидание- реальность», поэтому это просто бессмысленно.

Для создания «mock»-данных в мире Android чаще всего использу- ется библиотека Mockito. Изначально необходимо инициализировать её в build.gradle:

testImplementation 'org.mockito:mockito-core:2.25.0' testImplementation 'org.mockito:mockito-inline:2.25.0'

Далее представим для этих данных чуть более реальную модель для тестирования. Ранее, проходя темы Retrofit и Room в подразд. 2 и 3 соответ- ственно, встречались с паттерном «репозиторий». Реализуем его простей- шее подобие, основанное на абстрактном хранилище пользователя.

Реализация на Kotlin:

interface UserStorage { fun getUser(): String

}

class UsersRepository(private val storage: UserStorage) { @get:Throws(IOException::class)

val name: String?

get() {

val user = Gson().fromJson(storage.getUser(), User::class.java) return user.name

}

private data class User( val name: String

)

}

Реализация на Java:

interface UserStorage { String getUser();

}

public class UserRepository { private final UserStorage storage;

public UserRepository (UserStorage storage) { this.storage = storage;

}

public String getName() throws IOException { Gson gson = new Gson();

User user = gson.fromJson(storage.getUser(), User.class); return user.name;

}

private static final class User { String name;

}

}

На примере представленного выше кода видно, насколько важно делать обертку в виде интерфейсов над классами и правильно называть эти интерфейсы. Интерфейс UserStorage может представлять любое дей- ствие в реализации: HTTP-запрос, запрос к любой базе данных, SharedPreferences, обычный файл.

Прдеставленный код просто считает, что getUser интерфейса Us- erStorage возвращает JSON-строку.

Далее создадим в папке test отдельный тест UserRepositoryTest, код которого на Kotlin будет выглядеть следующим образом:

class UserRepositoryTest {

private val userStorage: UserStorage = mock(UserStorage::class.java) private val repository = UsersRepository(userStorage)

@Before

fun setUp() {

`when`(userStorage.getUser()).thenReturn("{ \"name\": \"Mark\"}")

}

@Test

fun getName\_isNameValid() {

val name: String? = repository.name assertEquals(name, "Mark")

}

}

Или на Java:

public class UserRepositoryTest { @Mock UserStorage storage;

UserRepository repository = new UserRepository(storage);

@Before

public void setUp() {

when(storage.getUser()).thenReturn("{ \"name\": \"Mark\"}");

}

@Test

public void getName\_isNameValid() { String name = repository.getName(); assertEquals(name, "Mark");

}

}

В первой строчке нет реализации для интерфейса UserStorage, а есть пометка его как mock. Это значит, что теперь можно эмулировать его реализацию.

В нашем Before-методе происходить инициализация поведения этого «мока». С помощью функции thenReturn задаем поведение для функции, которая задается в when.

Далее на вызов функции getUser будет возвращена JSON-строка. При изменении представленного класса репозитория или добавлении но- вой реализации для UserStorage, можно «прогнать» тесты и убедиться, что заложенная логика не «сломалась».

## UI-тестирование

Перед началом рассмотрения особенностей UI-тестирования, необ- ходимо понять, чем отличаются сгенерированные папки androidTest и test. В первой папке лежит тест с аннотацией RunWith и параметром An- droidJUnit. В Android есть два вида тестов: те, которые запускаются на базо- вой JVM, и те, которые запускаются на Android JVM. UI-тестирование, оче- видно, относится ко второму пункту.

@RunWith(AndroidJUnit4::class) class ExampleInstrumentedTest {

@Test

fun useAppContext() {

val appContext = InstrumentationRegis- try.getInstrumentation().targetContext

assertEquals("ru.mandarinshow.learnkotlin", appContext.packageName)

}

}

Рассмотрим сгенерированный IDE-тест. Данный вид тестов назы- вается инструментальный и нацелен на тестирование компонентов An- droid. В этих типах тестов можно получить контекст, а получая контекст, можно сделать буквально всё, что угодно. На примере выше показан не- большой тест, который проверяет правильность названия пакета прило- жения. Как видите, нет никаких отличий от обычного JUnit-теста, кроме аннотации @RunWith(AndroidJUnit4::class).

Для UI-тестирования в Android используется библиотека Espresso. В Espresso тесты работают в бэкграунд потоке, а взаимодействие с UI-элементами в потоке UI. Espresso имеет несколько основных классов для тестирования:

* Espresso – основной класс. Содержит в себе статические мето- ды, такие как нажатия на системные кнопки (Back, Home), вызвать/спрятать клавиатуру, открыть меню, обратится к компоненту;
* ViewMatchers – позволяет найти компонент на экране в теку- щей иерархии;
* ViewActions – позволяет взаимодействовать с компонентом (click, longClick, doubleClick, swipe, scroll и т.д.);
* ViewAssertions – позволяет проверить состояние компонента. Ниже представлено подключение Espresso через Gradle:

testImplementation 'junit:junit:4.12' androidTestImplementation 'androidx.test.ext:junit:1.1.1'

androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'

А также требуются еще несколько расширений для Android- тестирования:

androidTestImplementation 'androidx.test:rules:1.2.0' androidTestImplementation 'androidx.test:runner:1.2.0'

### *7.2.1. Реализация простейшего UI-теста*

Для первого UI-теста специально создадим простую Activity с единственным элементом на экране – текстовым полем (рис. 18), код ко- торой представлен ниже.



**Рис. 18.** Интерфейс макета Activity class UsefulActivity : AppCompatActivity() {

override fun onCreate(savedInstanceState: Bundle?) { super.onCreate(savedInstanceState) setContentView(R.layout.acitivity\_useful)

}

}

<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="<http://schemas.android.com/apk/res/android>" xmlns:app="<http://schemas.android.com/apk/res-auto>"

xmlns:tools="[http://schemas.android.com/tools"](http://schemas.android.com/tools) android:layout\_width="match\_parent" android:layout\_height="match\_parent">

<TextView android:id="@+id/tvText"

android:layout\_width="wrap\_content" android:layout\_height="wrap\_content" android:text="TextView" app:layout\_constraintBottom\_toBottomOf="parent" app:layout\_constraintEnd\_toEndOf="parent" app:layout\_constraintStart\_toStartOf="parent" app:layout\_constraintTop\_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>

Для представленной Activity создадим тест в каталоге */src/andro- idTest/java/…*.

На Kotlin:

@RunWith(AndroidJUnit4::class) class UsefulActivityTest {

@get:Rule

val activityRule = ActivityTestRule(UsefulActivity::class.java)

@Test

fun checkTextViewIsDisplayed() { onView(withId(R.id.tvText))

.check(matches(isDisplayed()))

}

}

На Java:

@RunWith(AndroidJUnit4.class) public class UsefulActivityTest {

@Rule

public ActivityTestRule<UsefulActivity> activityTestRule = new ActivityTestRule<>(UsefulActivity.class);

@Test

public void checkTextViewIsDisplayed() { onView(ViewMatchers.withId(R.id.tvText))

.check(matches(isDisplayed()));

}

}

Здесь появляется новое понятие – тестовое правило. Тестовое пра- вило (или test rule) – интерфейс для написания правил, которые будут вы- полняться перед запуском каждого теста. Такие правила можно писать самостоятельно или брать готовые, как в нашем случае. Наше правило создает тестовую Activity. Правила помечаются аннотацией @Rule.

Функция checkTextViewIsDisplayed – непосредственно сам тест, где уже используется Espresso.

Функция onView возвращает view, которую хотим протестировать, в данном случае это текст tvText, который был найден по id (по аналогии с findViewById).

Функция check выполняет одну из проверок, которые предоставле- ны интерфейсом Espresso. В данном случае идёт проверка, отображается ли UI-элемент на экране.

Если все действия были выполнены верно, то откроется эмулятор на нужной Actviity и IDE сообщит об успешном завершении теста.

## Контрольные вопросы

1. Для чего используется тестирование?
2. Тестирование в Android Studio.
3. В чем различия между Unit- и UI-тестами?
4. Реализация Unit-теста.
5. Реализация UI-теста.
6. Аннотации в тестировании.
7. Что такое «mock»-данные?
8. Для чего используется библиотека Mockito?
9. Для чего используется библиотека Espresso?
10. Какие основные классы библиотеки Espresso?

110

111

# ЗАКЛЮЧЕНИЕ

*7.2. UI-тестирование*

Учебное пособие содержит теоретический материал, практический материал с примерами выполнения и вопросы для самоконтроля. Пред- ставленные материалы сопровождаются примерами на двух языках с опи- санием их реализации: сначала представлен пример на Kotlin, следом за ним – пример на Java. Темы, охваченные в учебном пособии: создание мобильных приложений на языке Kotlin, работа с сетью на основе биб- лиотеки Retrofit, разработка и поддержка базы данных с использованием библиотеки Room, создание мобильных приложений на основе архитек- турных компонент Android, Unit- и UI-тестирование, тестирование API с использованием Postman, создание уведомлений (нотификаций).

Учебное пособие может быть использовано для подготовки к лек- циям, лабораторным занятиям, а также к экзамену по дисциплине «Раз- работка мобильных приложений» для студентов Института компьютер- ных технологий 09.03.04 – Программная инженерия и по дисциплине

«Программирование для мобильных платформ» для магистрантов Ин- ститута компьютерных технологий 09.04.04 – Программная инженерия. Также учебное пособие будет полезно для подготовки студентов к уча- стию в чемпионате WorldSkills по компетенции «разработка мобильных приложений».

Автор данного учебного пособия выражает благодарность студенту кафедры математического обеспечения и применения ЭВМ Института компьютерных технологий информационной безопасности ЮФУ, фина- листу межвузовского чемпионата по стандартам WorldSkills Russia 2019 г. по компетенции «разработка мобильных приложений» Абраменко Марку Андреевичу за помощь в подготовке материалов для учебного пособия.

# СПИСОК ЛИТЕРАТУРЫ

*Список литературы*

1. Learn Kotlin [Электронный ресурс]. – Режим доступа: https:// kotlin- lang.org/docs/reference/ (дата обращения: 1.12.2020).
2. Control Flow: if, when, for, while [Электронный ресурс]. – Режим до- ступа: https://kotlinlang.org/docs/reference/control-flow.html#when- expression (дата обращения: 1.12.2020).
3. Basic Syntax [Электронный ресурс]. – Режим доступа: https:// kotlin- lang.org/docs/reference/basic-syntax.html#defining-variables (дата об- ращения: 1.12.2020).
4. Basic Types [Электронный ресурс]. – Режим доступа: https://kotlinlang.org/docs/reference/basic-types.html#string-templates (дата обращения: 1.12.2020).
5. Object Expressions and Declarations [Электронный ресурс]. – Режим до- ступа: https://kotlinlang.org/docs/reference/object-declarations.html (дата обращения: 1.12.2020).
6. *Jemerov D., Isakova S.* Kotlin in Action. Manning, 2018.
7. Get Started with Kotlin on Android [Электронный ресурс]. – Режим до- ступа: https://developer.android.com/kotlin/get-started (дата обращения: 1.12.2020).
8. Null Safety [Электронный ресурс]. – Режим доступа: https:// kotlin- lang.org/docs/reference/null-safety.html (дата обращения: 1.12.2020).
9. *Пирская, Л. В.* Разработка мобильных приложений в среде Android Studio [Текст]: учебное пособие/ Л. В. Пирская. Южный федераль- ный университет. – Ростов-на-Дону; Таганрог: Издательство Южно- го федерального университета, 2019. – 128 с.
10. Higher-Order Functions and Lambdas [Электронный ресурс]. – Режим доступа: https://kotlinlang.org/docs/reference/lambdas.html (дата об- ращения: 1.12.2020).
11. Android KTX [Электронный ресурс]. – Режим доступа: https://developer.android.com/kotlin/ktx (дата обращения: 1.12.2020).
12. Data Binding Library [Электронный ресурс]. – Режим доступа: https:// developer.android.com/topic/libraries/data-binding (дата обращения: 1.12.2020).
13. View Binding [Электронный ресурс]. – Режим доступа: https:// de- veloper.android.com/topic/libraries/view-binding (дата обращения: 1.12.2020).
14. Extensions [Электронный ресурс]. – Режим доступа: https:// kotlin- lang.org/docs/reference/extensions.html (дата обращения: 1.12.2020).
15. Collection Operations [Электронный ресурс]. – Режим доступа: https://kotlinlang.org/docs/reference/collection-operations.html (дата обращения: 1.12.2020).
16. Coroutine Basics [Электронный ресурс]. – Режим доступа: https:// kotlinlang.org/docs/reference/coroutines/basics.html (дата обращения: 1.12.2020).
17. Coroutines Guide [Электронный ресурс]. – Режим доступа: https:// kotlinlang.org/docs/reference/coroutines/coroutines-guide.html (дата обращения: 1.12.2020).
18. Square / retrofit [Электронный ресурс]. – Режим доступа: https:// github.com/square/retrofit (дата обращения: 1.12.2020).
19. Retrofit [Электронный ресурс]. – Режим доступа: https:// square.github.io/retrofit/ (дата обращения: 1.12.2020).
20. Room Persistence Library [Электронный ресурс]. – Режим доступа: https://developer.android.com/topic/libraries/architecture/room (дата обращения: 1.12.2020).
21. Android Room with a View - Kotlin [Электронный ресурс]. – Режим доступа:https://codelabs.developers.google.com/codelabs/android- room-with-a-view-kotlin/ (дата обращения: 1.12.2020).
22. Android Room with a View - Java [Электронный ресурс]. – Режим доступа: https://codelabs.developers.google.com/codelabs/android- room-with-a-view (дата обращения: 1.12.2020).
23. Save data in a local database using Room [Электронный ресурс]. – Режим доступа: https://developer.android.com/training/data-storage/ room (дата обращения: 1.12.2020).
24. Defining data using Room entities [Электронный ресурс]. – Режим доступа: https://developer.android.com/training/data-storage/room/ de- fining-data (дата обращения: 1.12.2020).
25. Accessing data using Room DAOs [Электронный ресурс]. – Режим доступа: https://developer.android.com/training/data-storage/room/ ac- cessing-data (дата обращения: 1.12.2020).

113

*Список литературы*

1. Java 8 Concurrency Tutorial: Threads and Executors [Электронный ресурс]. – Режим доступа: https://winterbe.com/posts/2015/04/07/ ja- va8-concurrency-tutorial-thread-executor-examples/ (дата обращения: 1.12.2020).
2. Postman Learning Center // Documentation [Электронный ресурс]. – Режим доступа: https://learning.postman.com/docs/ (дата обращения: 1.12.2020).
3. HTTP request methods [Электронный ресурс]. – Режим доступа: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods (дата об- ращения: 1.12.2020).
4. Random User API // Documentation [Электронный ресурс]. – Режим доступа: https://randomuser.me/documentation (дата обращения: 1.12.2020).
5. Postman Learning Center // Using variables [Электронный ресурс]. – Режим доступа: https://learning.postman.com/docs/sending-requests/ variables/ (дата обращения: 1.12.2020).
6. Notifications Overview [Электронный ресурс]. – Режим доступа: https://developer.android.com/guide/topics/ui/notifiers/notifications (да- та обращения: 1.12.2020).
7. Github // Android Notifications Sample [Электронный ресурс]. – Ре- жим доступа: https://github.com/googlearchive/android-Notifications (дата обращения: 1.12.2020).
8. NotificationCompat.Builder [Электронный ресурс]. – Режим доступа: https://developer.android.com/reference/androidx/core/app/Notification Compat.Builder#setStyle(android.support.v4.app.NotificationCompat.Style) (дата обращения: 1.12.2020).
9. NotificationManager [Электронный ресурс]. – Режим доступа: https://developer.android.com/reference/android/app/NotificationManag er (дата обращения: 1.12.2020).
10. PendingIntent [Электронный ресурс]. – Режим доступа: https:// de- veloper.android.com/reference/android/app/PendingIntent (дата обра- щения: 1.12.2020).
11. LiveData Overview [Электронный ресурс]. – Режим доступа: https://developer.android.com/topic/libraries/architecture/livedata (дата обращения: 1.12.2020).
12. ViewModel Overview [Электронный ресурс]. – Режим доступа: https://developer.android.com/topic/libraries/architecture/viewmodel (дата обращения: 1.12.2020).

3

1. Two-way data binding [Электронный ресурс]. – Режим доступа: https://developer.android.com/topic/libraries/data-binding/two-way (да- та обращения: 1.12.2020).
2. Layouts and binding expressions [Электронный ресурс]. – Режим https://developer.android.com/topic/libraries/data-binding/expressions (дата обращения: 1.12.2020).
3. ViewModels: A Simple Example [Электронный ресурс]. – Режим доступа: https://medium.com/androiddevelopers/viewmodels-a-simple- example-ed5ac416317e (дата обращения: 1.12.2020).
4. Тестирование Android приложений [Электронный ресурс]. – Режим доступа: https://habr.com/ru/post/352334/ (дата обращения: 1.12.2020).
5. JUnit API // Assert [Электронный ресурс]. – Режим доступа: https:// junit.org/junit4/javadoc/4.12/org/junit/Assert.html (дата обращения: 1.12.2020).

*Учебное издание*

*Список литературы*

**ПИРСКАЯ Любовь Владимировна**

# РАЗРАБОТКА СОВРЕМЕННЫХ МОБИЛЬНЫХ ПРИЛОЖЕНИЙ ДЛЯ ОС ANDROID

*Учебное пособие*

Редакторы: *Н. И. Селезнева, З. И. Надточий*

Корректор *Л. В.Чиканенко*

Компьютерная верстка *И. А. Клочко*

Подписано к печати 08.02.2021 г.

Бумага офсетная. Формат 60х841/16. Усл. печ. лист. 6,74.

Уч.-изд. л. 3,81. Тираж 30 экз. Заказ № 7959.

Издательство Южного федерального университета

Отпечатано в отделе полиграфической, корпоративной и сувенирной продукции Издательско-полиграфического комплекса КИБИ МЕДИА ЦЕНТРА ЮФУ 344090, г. Ростов-на-Дону, пр. Стачки, 200/1. Тел. (863) 243-41-66

4