

# Содержание

<b>1</b>	<i>Первые шаги в Android-разработке</i>	<b>3</b>
1.1	Знакомство с курсом . . . . .	3
1.1.1	Первое приложение . . . . .	6
1.1.2	Интерфейс студии и структура проекта . . . . .	11
1.1.3	Создание и запуск Android эмулятора . . . . .	18
1.1.4	Создание стороннего эмулятора . . . . .	21
1.1.5	Краткое знакомство с часто встречающимися понятиями .	21
<b>2</b>	<i>Компоненты Android и элементы интерфейса</i>	<b>28</b>
2.1	Activity и View . . . . .	28
2.1.1	Основные компоненты Android. Context . . . . .	28
2.1.2	Activity. Жизненный цикл . . . . .	30
2.1.3	Интерфейс. View и ViewGroup . . . . .	33
2.1.4	Реализация ViewGroup . . . . .	34
2.1.5	ImageView, EditText . . . . .	37
2.2	Инструменты сборки и отладки . . . . .	40
2.2.1	Система сборки Gradle . . . . .	40
2.2.2	Toast . . . . .	43
2.2.3	Menu, ContextMenu . . . . .	45
2.3	Фрагменты и файлы Preferences . . . . .	48
2.3.1	Знакомство с Fragment . . . . .	48
2.3.2	Формат JSON. Библиотека GSON . . . . .	52
<b>3</b>	<i>Старт курсового проекта(КП)</i>	<b>56</b>
3.1	Activity авторизации . . . . .	56
3.1.1	КП. Вёрстка экрана логина . . . . .	56
3.1.2	КП. Добавление ссылок на View элементы . . . . .	58
3.1.3	КП. Валидация email и password. Ошибки в Toast . . . . .	60
3.2	Activity профиля . . . . .	62
3.2.1	КП. Вёрстка экрана профиля . . . . .	62
3.2.2	КП. Создание активити профиля . . . . .	65
3.2.3	КП. Создание класса User . . . . .	68
3.3	Добавление фрагментов . . . . .	70

3.3.1	КП. Создание хост активити для фрагментов . . . . .	70
3.3.2	КП. Миграция логики AuthActivity во фрагмент . . . . .	71
3.3.3	КП. Добавление фрагмента регистрации. Создание класса PreferenceHelper . . . . .	75
<b>4</b>	<b>Завершение курсового проекта</b>	<b>83</b>
4.1	Добавление логики авторизации . . . . .	83
4.1.1	КП. Логика авторизации. Работа с бэкстеком . . . . .	83
4.1.2	КП. Экран профиля. Логаут, меню . . . . .	87
4.1.3	КП. Обновлённая логика авторизации . . . . .	89
4.1.4	КП. Экран профиля. Извлечение изображения из галереи .	91
4.1.5	КП. Градиентный фон . . . . .	93

# Неделя 1

## *Первые шаги в Android-разработке*

### 1.1 Знакомство с курсом

Эта специализация спроектирована не только для того, чтобы вы узнали основы разработки под Android, но еще для того, чтобы вы легко могли влиться в любой коллектив. Иначе говоря, в нашей серии курсов будет не только ванильный Android, в нем будет разбор архитектур: MVP, MVVM, Clean, новоиспеченного Life Architecture. Будет изучение популярных библиотек, таких как Retrofit, Picasso, ButterKnife, Moxy и других, использованных на реальных проектах. Будет функционально-реактивное программирование посредством RxJava и нового Stream API. Помимо этого вы научитесь сетевому взаимодействию и хранению данных в базе, возьмете многопоточность под контроль, наведете лоск с помощью анимации и material-дизайна и узнаете многое другое.

В этом курсе мы разберём:

1. Установку и настройку Android Studio
2. Создание и работу с эмулятором
3. Основные компоненты Android приложения
4. Основные элементы интерфейса
5. View и ViewGroup реализации
6. Работу с Activity и Fragment
7. Работу с ресурсами
8. Работу с Preferences

Также будут приведены маленькие примеры кода для ознакомления с некоторыми аспектами различных механизмов. Совместно с этим, будут продемонстрированы приложения из Маркета, которые реализуют какие-либо комплексные механизмы. Для лучшего понимания реализации этих механизмов в коммерческом приложении.

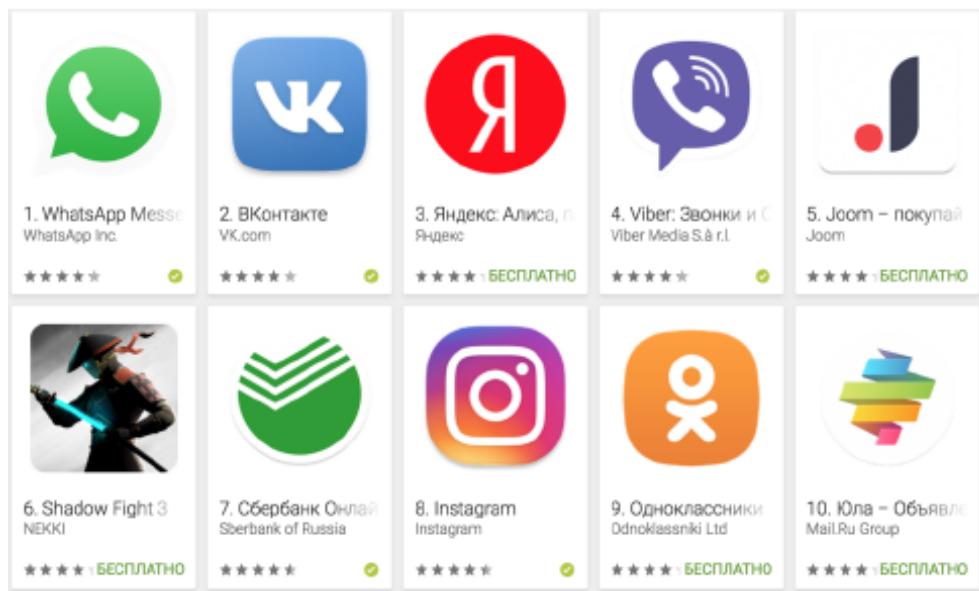


Рис. 1.1: Приложения, на которых будут разбираться разные механизмы

В 2003 году инженер-программист Энди Рубин основал Android Incorporated, которая разработала мобильную операционную систему Android, а в 2005 году эта компания была куплена Google. Первый телефон на Android (T-Mobile G1) поступил в продажу в 2008 году. В первой версии Android не было возможности воспроизведения видео, не было браузера и виртуальной клавиатуры. С каждой версией добавлялся новый функционал. Под Android программируют на Java потому что язык работает на виртуальной машине и его не нужно перекомпилировать для каждого нового устройства.

Version	Codename	API	Distribution
2.3.3 - 2.3.7	Gingerbread	10	0.5%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	0.5%
4.1.x	Jelly Bean	16	2.2%
4.2.x		17	3.1%
4.3		18	0.9%
4.4	KitKat	19	13.8%
5.0	Lollipop	21	6.4%
5.1		22	20.8%
6.0	Marshmallow	23	30.9%
7.0	Nougat	24	17.6%
7.1		25	3.0%
8.0	Oreo	26	0.3%

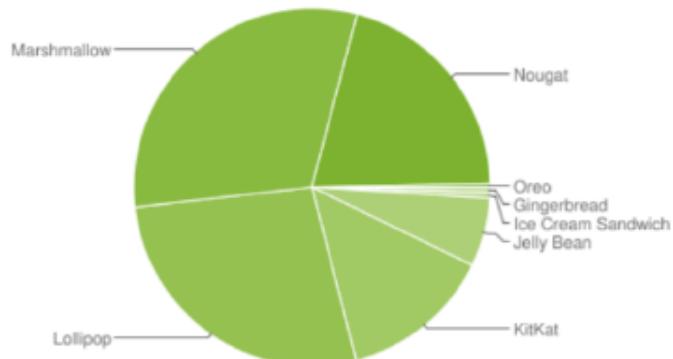


Рис. 1.2: Распределение устройств по различным версиям Android

### Данные по распределению процента устройств по их API.

Весной 2017 года Google объявил об официальной поддержке Kotlin в качестве основного языка разработки под Android. Kotlin — это очень мощный JVM-язык, в котором много новых интересных возможностей, недоступных на Java. Помимо Java и Kotlin на Android можно писать и на других JVM-языках (Java Virtual Machine): Scala, Groovy и прочих.

Библиотеки поддержки:

- com.android.support:appcompat-v7:27.0.1
- com.android.support:design:27.0.1
- com.android.support:support-v13:27.0.1

Основные источники информации и ответы на типичные вопросы:

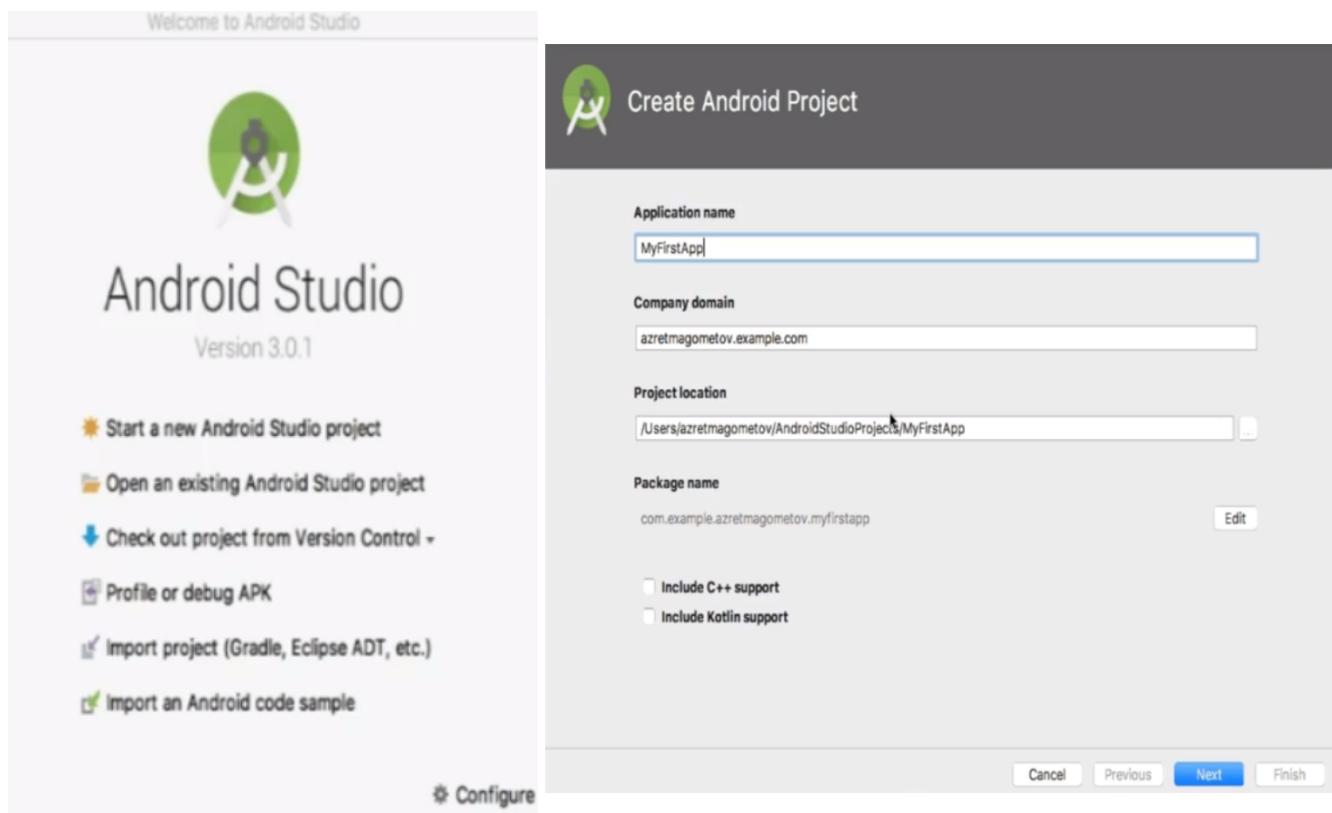
- [Официальная документация](#) - Гайды, обучение, дизайн, классы, best practices...
- [Stack Overflow](#) - 99,9% вопросов, которые у вас возникнут, уже были тут.
- [Блог разработчиков Android](#) - Новые инструменты, best practices...
- [Гугл-документ с основными полезными ссылками](#)

Далее выполняем всё согласно инструкции:

1. [Алгоритм установки Android Studio](#)
2. [Настройка Android SDK.](#)

### 1.1.1 Первое приложение

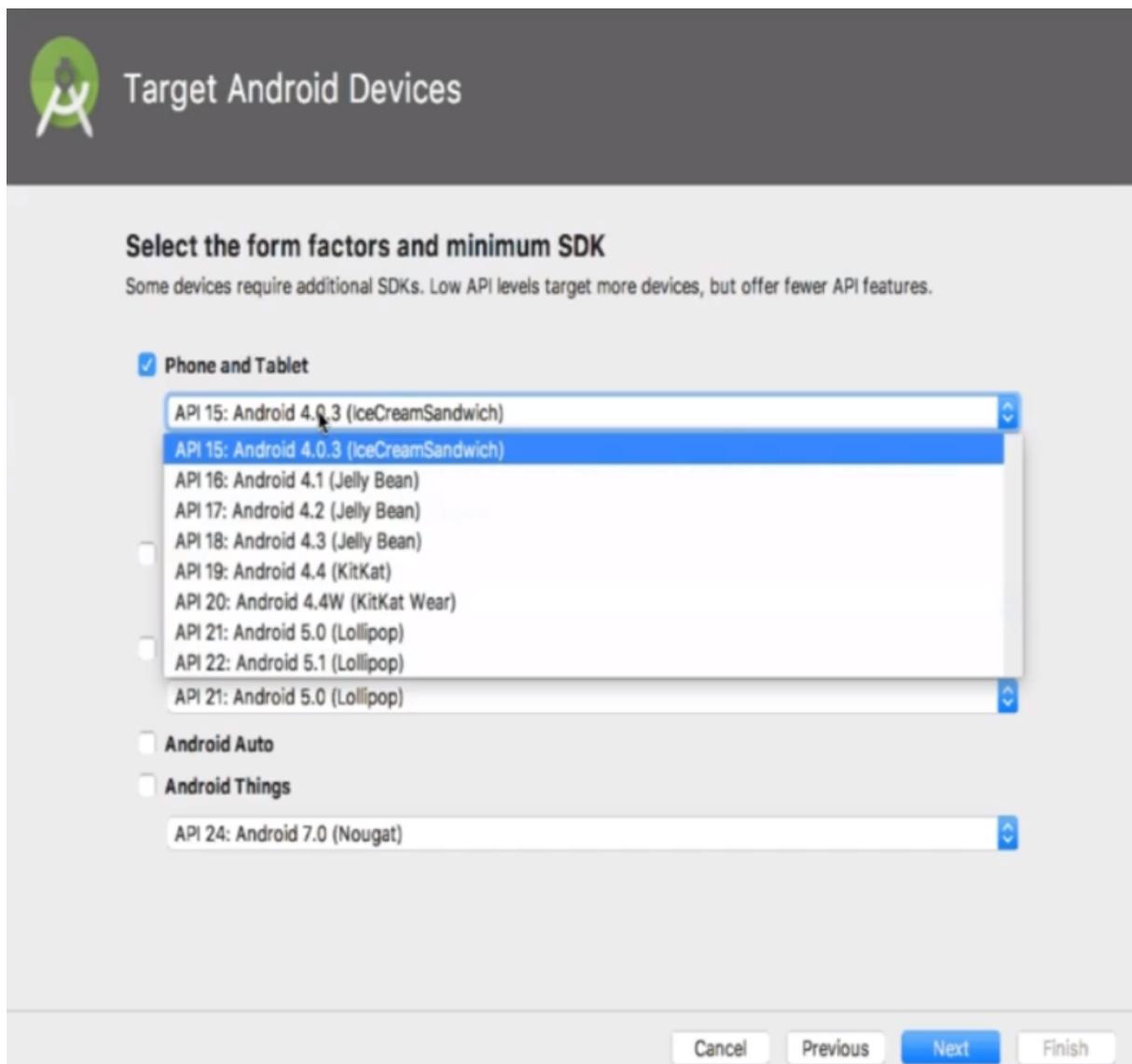
Создание нового проекта



Для уникальности названий проектов в Android создаётся уникальное название пакета состоящее из доменной компании (или, например, адреса сайта) и названия приложения. В Google Play приложения можно опознать именно по названию пакета.

Поддержка C++ и поддержка Kotlin сейчас нам не нужны, но вообще поддержку C++ можно включить, если планируется вызвать C++ код из Java (либо использование библиотеки, написанной на C++, либо обращение к более низкому слою Android-фреймворка). Kotlin — это новый JVM-язык, который недавно получил официальную поддержку от Google. Kotlin достаточно серьезно отличается от Java и своей парадигмой, и объектно-функциональным стилем, и непосредственно синтаксисом. Поэтому пока не будем его включать.

Далее мы должны определиться с устройствами и версиями Android, для которых мы пишем приложение:



Каждой версии Android соответствует свой уровень API (application programming interface). Важный момент: **чем ниже минимальный уровень API, тем на большем количестве устройств может работать ваше приложение**. Почему это происходит? Смартфоны с Android далеко не всегда обновляются до последней актуальной версии API. И они остаются у людей.

Чем ниже уровень API, то меньше новых возможностей вам доступны из коробки. Каждая версия Android преподносит что-то новое, и если минимальный уровень API в котором появилась та или иная возможность. Для использования этой фичи, вам нужно будет либо проверить текущую версию OS устройства на соответствие, либо использовать функционал этой фичи из библиотеки обратной совместимости. Например, (узнать, что когда добавилось можно кликнув на "help me choose") в 6-ой версии Android (API 23) появился функционал для работы с отпечатками пальцев. Однако если я выставлю минимальный API —

19, Kitkat, то я уже не могу просто так вызвать код, чтобы использовать сенсор «отпечаток пальцев», так как приложение может быть запущено на устройстве с Android 4.4, 5.0, 5.1, то есть на OS без поддержки сенсора. И если просто вызвать код, то приложение крашнется. В этом случае есть два варианта: либо перед вызовом проверить, что текущая версия операционной системы Android  $\geq 23$ , либо используем библиотеку обратной совместимости, которая просто использует заглушки для API ниже 23. Однако следует учитывать, что не для всех новых фич есть библиотеки обратной совместимости. Получаем, что **чем выше версия API, тем больше возможностей (фич) доступно из коробки.**

Мы будем использовать Android 4.4 KitKat (API 19), чтобы наше приложение поддерживалось 90,1% устройств.

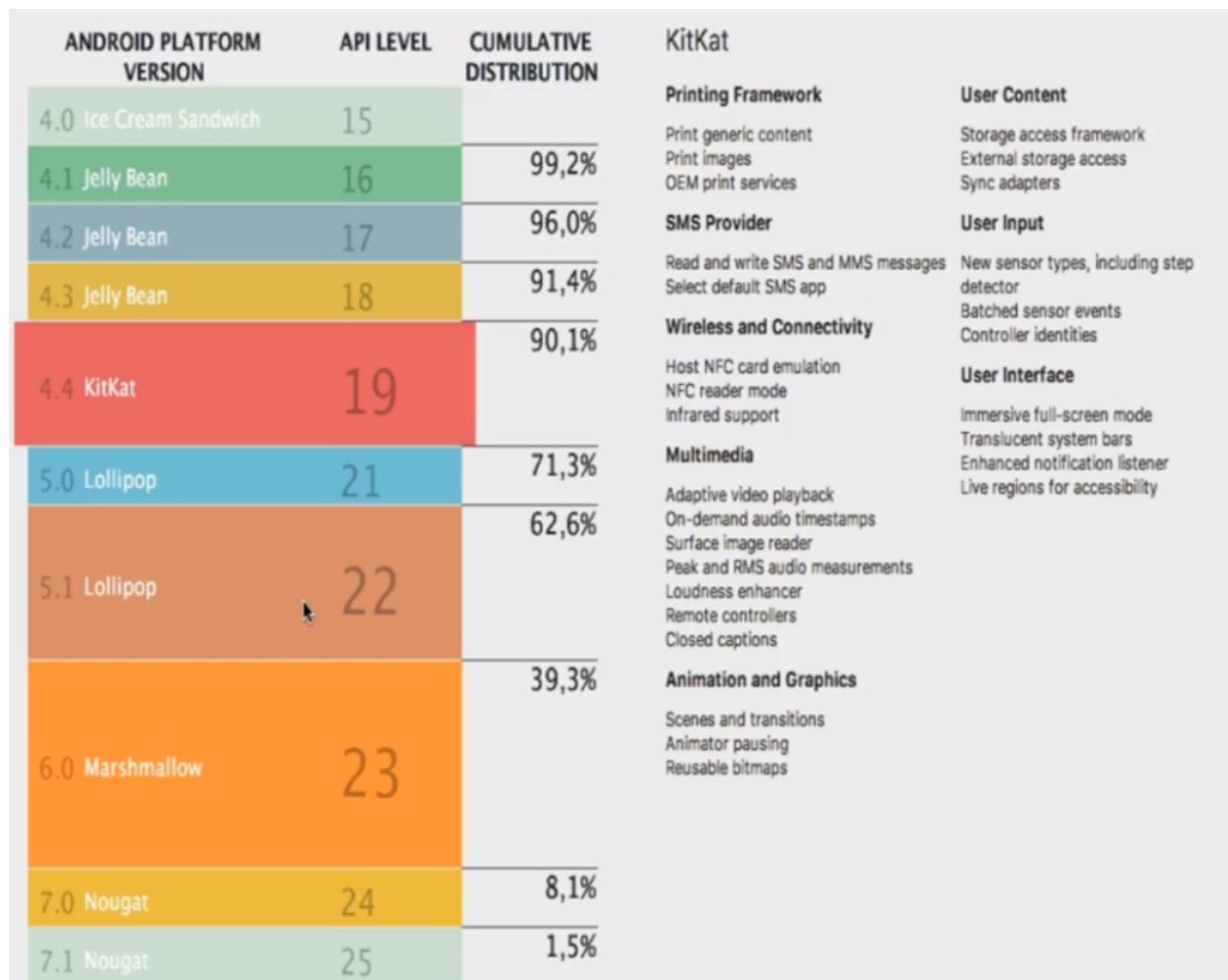


Рис. 1.3: Процент устройств, поддерживающих соответствующий API

Далее мы должны выбрать запускаемую Activity (по сути это какой-либо экран

приложения). Например, при запуске приложения у вас открывается экран авторизации — это первая Activity. Вы вбиваете логин, пароль, щелкаете вход и открывается вторая Activity — главный экран. Вы щелкаете на Настройки и открывается третья Activity — экран с настройками. Тут важно понимать одну вещь: все эти Activity — это все одна сущность, но с разной версткой интерфейса и соответствующей бизнес-логикой.

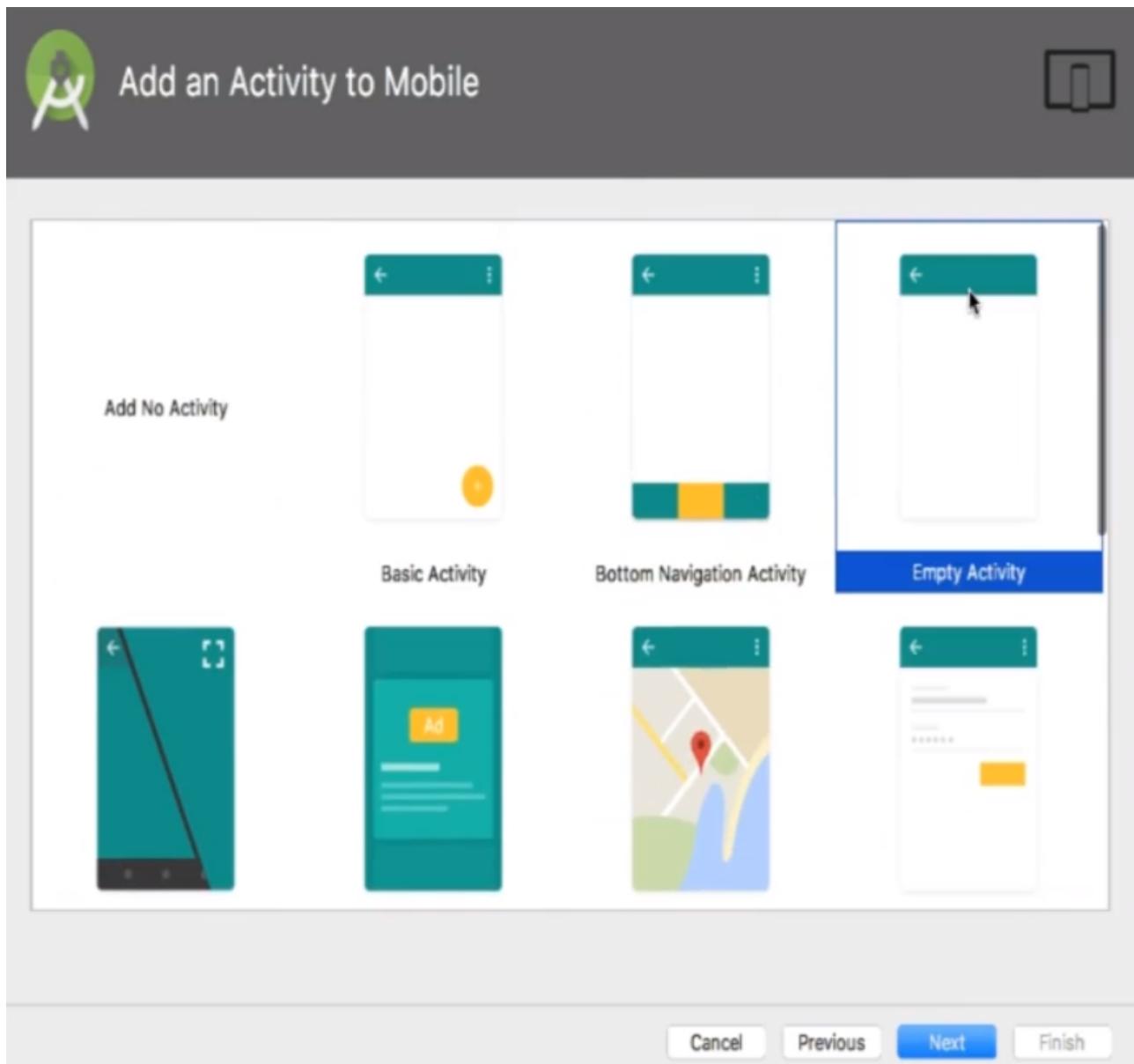


Рис. 1.4: Выбор типа Activity

Выбираем Empty Activity, щелкаем Next. Здесь выбираем название Activity, оно должно быть опознаваемым и обоснованным. Например, не стоит называть Activity своим именем. И еще: хорошим тоном считается использовать название класса, наследующегося от компонента Android, название этого самого компонента. В нашем случае это Activity, LaunchActivity. Галочку «Сгенерировать

разметку» оставляем, чтобы не создавать ее самим. Галочку обратной совместимости оставляем, что именно она делает, я расскажу чуть попозже. Нажимаем Next. Studio качает необходимые файлы. Щелкаем Finish и ждём запуска студии (это может занять час времени и требовать стабильного подключения к интернету).

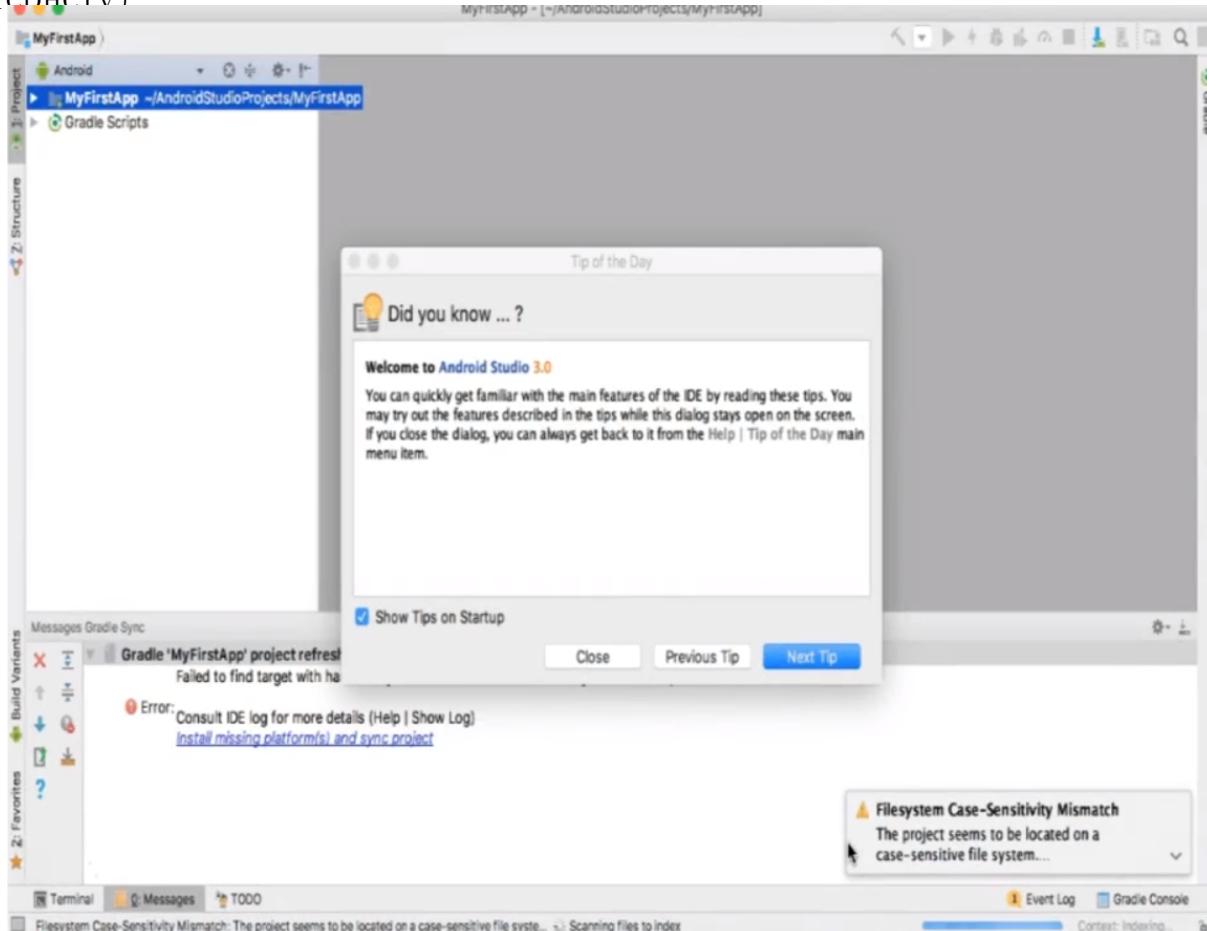
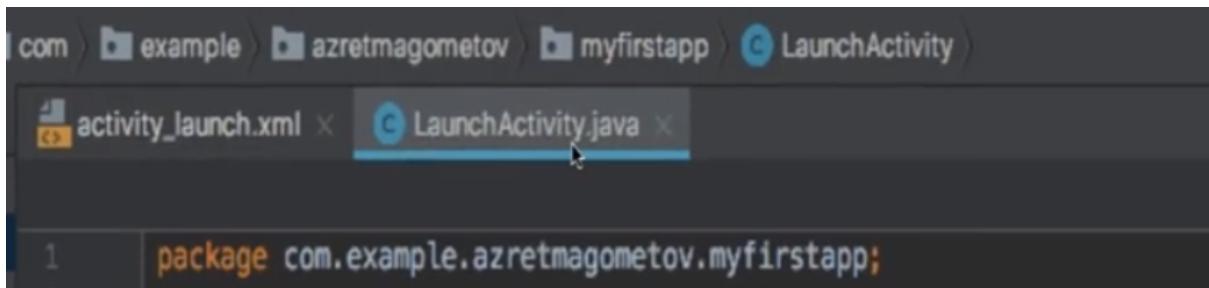


Рис. 1.5: Запуск Android Studio

Закрываем окно подсказок. И видим, что студия выдаёт ошибку "Failed to find target with hash...". Кликаем на "Install missing platform(s) and sync project" при-нимаем лицензионное соглашение. Next. Ждём загрузки. Finish. Студия сно-ва выдаёт ошибку, на этот раз нажимаем "Install Build Tools 26.0.2 and sync project". Всё устанавливается и происходит долгий первый запуск студии.

### 1.1.2 Интерфейс студии и структура проекта

Итак, прямо по центру — это окно, где мы непосредственно пишем наш код. Сейчас тут открыт файл LaunchActivity, и это, как вы можете заметить, java-файл. В java-файл классах мы пишем бизнес-логику нашего приложения.



В начале у нас в LaunchActivity.java появился код:

Листинг 1.1: LaunchActivity.java

```
package com.example.azretmagometov.myfirstapp;
import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
//стандартно для всех
public class LaunchActivity extends AppCompatActivity {
//наша LaunchActivity наследуется от AppCompatActivity.
//Из-за обратной совместимости работает везде одинаково
//При написании Android-приложений мы наследуемся компонентов
//android, которые вшиты в телефоны и добавляем свою логику
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        //вызов метода родителя
        setContentView(R.layout.activity_auth);
        //задаём layout экрана
    }
}
```

после завершения проекта, после сборки мы получим apk, в котором не будет оригинальных компонентов Android. В apk будут компоненты, которые мы написали сами, или добавили через библиотеки. Смотрите, наша LaunchActivity наследуется от AppCompatActivity. И само собой, AppCompatActivity в конечном итоге будет наследоваться от обычной activity. Давайте проследим этот момент. Нажимаем Ctrl (или Command если вы на Mac) и щелкаем по AppCompatActivity - мы перешли в класс AppCompatActivity и видим, что оно наследуется от FragmentActivity и т.д. имеем:

LaunchActivity ⇒ AppCompatActivity ⇒ FragmentActivity ⇒ BaseFragmentActivity16 ⇒ BaseFragmentActivity14 ⇒ SupportActivity ⇒ Activity.

Саму Activity мы не можем просто так посмотреть. Поэтому кликаем на неё. принимаем соглашение JetBrainsDecompiler и видим сам класс Activity без исходников, потому что код вшит в телефон. Android Studio предлагает скачать Android API 26 Platform. Щёлкаем Download и качаем код Android 8.0. Но надо понимать, что activity у разных API могут чем-то отличаться и вести себя по-разному на разных моделях Android. После скачивания вернёмся к LaunchActivity (1.1). Слова AppCompat, Compat, Support значат что работает поддержка обратной совместимости и на всех устройствах Activity должна работать максимально одинаково (т.к. класс взят из библиотеки поддержки). При создании Activity (т.е. нашего экрана), вызывается метод onCreate, который создает экран. super.onCreate означает, что вызывается метод родителя onCreate(). И в компонентах Android чаще всего необходимо вызывать методы родителей, потому что они, в конце концов, упираются в какую-то низкоуровневую логику, наподобие сохранения state'a, отправки сообщений, очистки и тому подобное. С помощью Ctrl (или Command) можно посмотреть код супер-метода onCreate(). Видим, что в super методе есть какой-то код, после которого опять же вызывается super метод. И это означает, что в вашем текущем коде тоже нужно вызывать super метод. Конкретно сейчас если попытаться убрать super метод, то при запуске Activity вылетит с ошибкой, потому что мы убиваем ее логику. Через Ctrl нажимаем на activity\_auth:

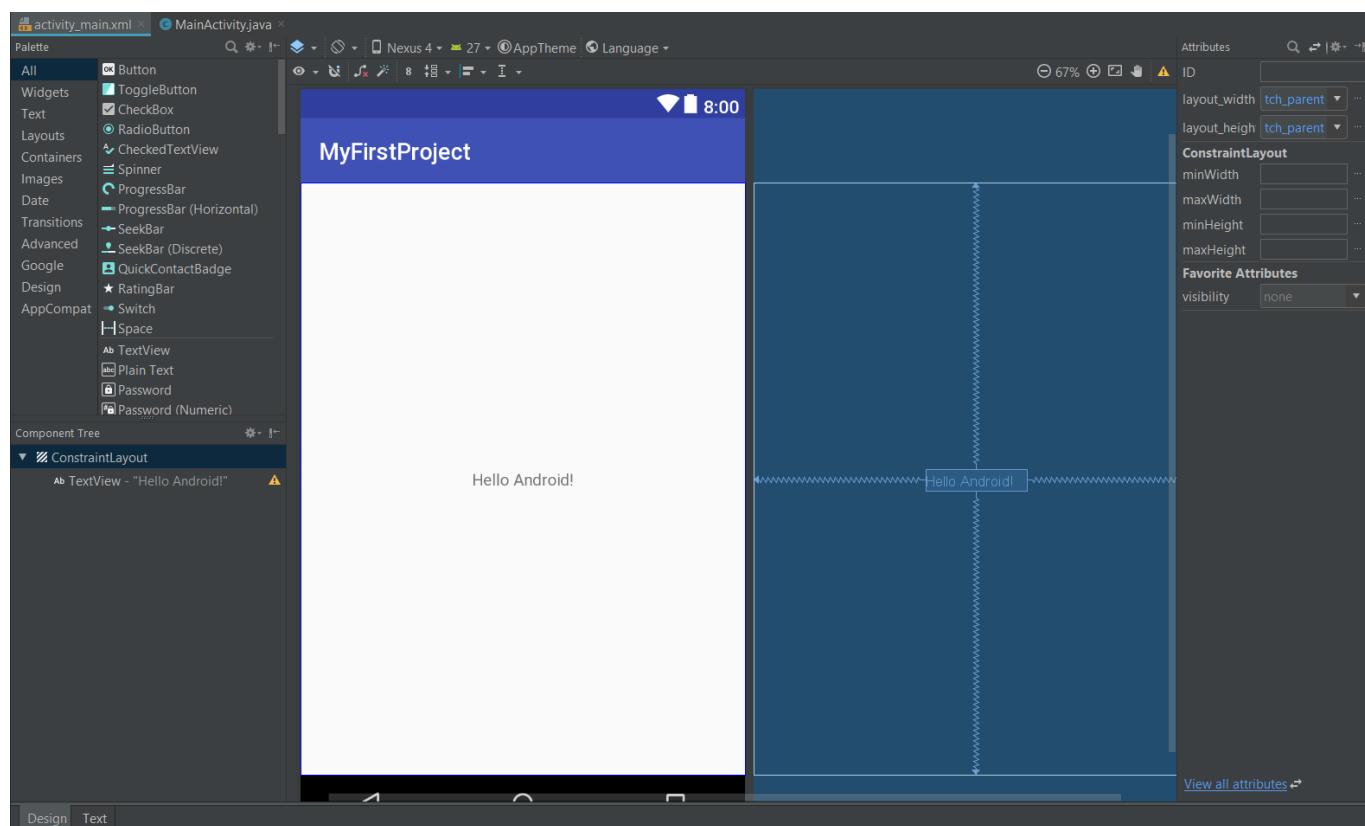


Рис. 1.6: вкладка Design у activity\_main.xml Файла

Сверху слева находится **Palette** (палитра) т.е. список элементов интерфейса, которые можно добавить на верстку. Она отфильтрована по типам: работа с текстом, виджеты, и т.д.

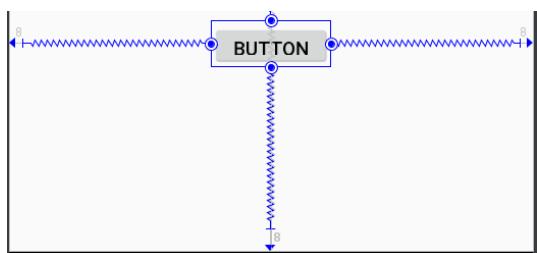
Под палитрой видим **Component Tree** (дерево элементов) - дерево компонентов, из которых состоит текущая верстка. Здесь мы видим, что корневым является ConstraintLayout, и в нем находится textView с текстом «Hello World!». Справа мы видим превью экрана, именно так и выглядит, будет выглядеть наш экран на устройстве.

Сверху **панель управления превью**, можно выбрать девайс, на котором будет запущено приложение, ну и проверить предпросмотр, повернуть телефон при желании. Ниже видим ещё инструменты. Они используются только для работы с ConstraintLayout, про это я расскажу чуть попозже.

Правее находится **Attributes(панель атрибутов)**. Этот список атрибутов на самом деле неполный. Внизу находится ссылка View all attributes. Щелкаем и видим уже весь список атрибутов. Что еще вам нужно знать? Атрибуты для каждого элемента свои. То есть если я сейчас щелкну на ConstraintLayout, то атрибуты поменяются.

Давайте сейчас попробуем поменять текст на кнопке. Видим атрибут Hello World, text, точнее, со значением «Hello World!». Пишем в text: «Hello Android!». Видим, что текст поменялся.

Важный момент - расположение элементов на верстке зависит от того, в каком контейнере они располагаются. В нашем случае контейнер — это ConstraintLayout, и в ConstraintLayout элементы мы можем располагать как угодно, важно их к чему-нибудь привязать с помощью Constraint (пружинки в четыре стороны от "Hello, Android!"). Важный момент - расположение элементов на верстке зависит от того, в каком контейнере они располагаются.



В нашем случае контейнер — это ConstraintLayout, и в ConstraintLayout элементы мы можем располагать как угодно, важно их к чему-нибудь привязать с помощью Constraint (пружинки в четыре стороны от "Hello, Android!"). Теперь мы нажимаем на белые точечки по краям синей рамки и привязываем кнопку к краям и надписи "Hello, Android".

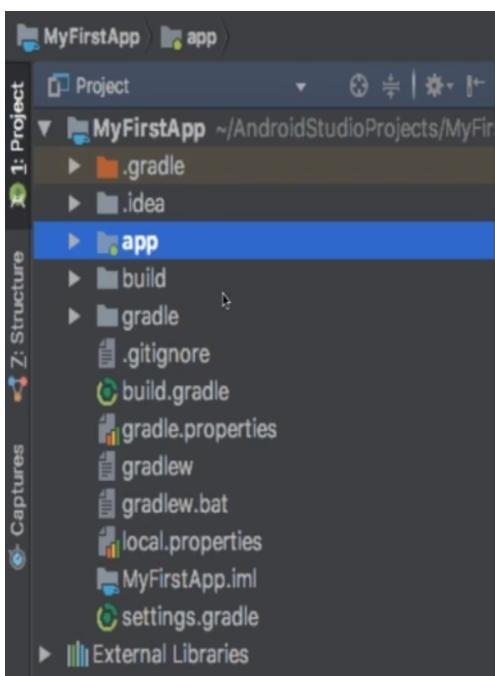
Таким образом привязываем с четырёх сторон нашу кнопочку. Заметим, что в поле её атрибутов есть TextView(унаследованный от textView). Поменяем Button на, например, Click. Надпись изменилась.

Всё это время мы находились во вкладке Design (снизу). Но есть ещё вкладка Text - это xml разметка нашего Activity. Получается, что Studio позволяет нам создавать разметку либо с помощью дизайна, вкладки Design, когда мы просто перетягиваем элементы и растягиваем мышкой, либо напрямую через XML.

Откроем получившуюся разметку:

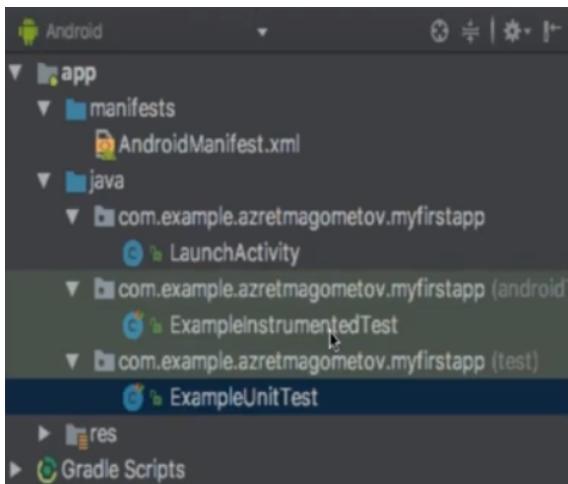
```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/
        android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="jzargo.myfirstproject.MainActivity">
    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginBottom="8dp"
        android:text="Hello Android!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="34dp"
        android:layout_marginBottom="8dp"
        android:layout_marginEnd="8dp"
        android:layout_marginStart="8dp"
        android:layout_marginTop="60dp"
        android:text="Click"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/textView"
        app:layout_constraintVertical_bias="0.0" />
</android.support.constraint.ConstraintLayout>
```

Закроем файл разметки (activity\_xml) и наша активити (LaunchActivity.java). Теперь вопрос: как найти наши файлы?



Слева, как вы могли заметить, находится панель со структурой нашего проекта. И эта панель, она поддерживает несколько различных представлений. Прямо сейчас используется представление Android. Это представление оставляет только те файлы и папки, которые важны прямо здесь и сейчас для работы с Android-кодом. Но мы можем переключиться на Project, и тогда структура файлов и папок будет такой же, какой она является на вашем жестком диске. Но опять же это плодит очень много ненужных файлов, слишком много папок, поэтому мы переключимся обратно на Android и разберем структуру здесь. Итак, изначально мы видим две папки: app и Gradle Scripts.

app — это модуль вашего приложения. **Модуль** — это отдельная папка со своими ресурсами и кодом, которые используются на конкретном типе устройств. Смотрите, при генерации, при создании проекта мы ставили галочку на «Планшеты и телефоны». Поэтому в нашем приложении сгенерировался только один модуль. Если бы мы поставили еще одну галочку, например, на часах, то у нас было бы в приложении два модуля: app для телефонов и планшетов, как сейчас, и wear для умных часов. Код и ресурсы на умных часах, отличаются от кода и ресурсов, которые использовались бы в приложении. Но если у них были бы какие-то схожие, смежные, смежный функционал, действительно, смежный функционал, то его, в свою очередь, можно было бы выделить в отдельный модуль, который использовали бы и модуль app, и модуль wear. Примерно так же работают подключаемые библиотеки: они просто добавляют еще один модуль к вашему приложению (т.е. библиотеки, исходный код которых вы встраиваете в проект и можете менять).



Вернёмся к представлению Android и увидим это.

AndroidManifest — это файл, в котором находится основная информация о вашем приложении. В первой папке наш LaunchActivity. Во второй - инструмент тесты. А в третьей - юнит тесты.

Листинг 1.2: AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.azretmagometov.myfirstapp">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".LaunchActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Видим: иконку и круглую иконку (в более поздних версиях) для запуска приложения с главного экрана, тема и активити, которые будут использоваться в приложении. В intent-filter написано, что она — запускаемая Activity, то есть при нажатии на иконку на экране телефона приложение запустится именно с этой Activity. [Подробная информация о файле AndroidManifest](#). В папке java:

- в первой подпапке видим нашу LaunchActivity

- во второй (android) - **инструментальные тесты** - тестирование взаимодействия компонентов андроид и поэтому они запускаются как обычное приложение - на эмуляторе или смартфоне.
- В третьей (test) - **юнит-тесты** - проверка java-классов и тестирование бизнес-логики приложения. Они запускаются прямо с компьютера.

В папке res находятся **ресурсы** приложения - это всё, что не является java кодом и не планируется качать с интернета(строки, цвета, константы, разметка экрана, картинки и т.д.).

- Папка drawable хранятся изображения. По умолчанию там лежат иконки (в виде векторной графики) запуска в виде нескольких слоёв.
- В layout находится наш файл разметки (вёрстки) activity\_launch.xml
- В mipmap хранится иконка с главного экрана. Инструкция по построению изображения
- В values значения строк, цветов и стилей (несколько атрибутов, объединённых в один, например, одинаковые кнопки)

Теперь рассмотрим Gradle Scripts. Там нас интересует build.gradle (app).

```
apply plugin: 'com.android.application'
android {
    compileSdkVersion 26 //уровень API сборки проекта
    defaultConfig {
        applicationId "com.example.azretmagometov.
            myfirstproject"
        minSdkVersion 19 //минимальная поддерживаемая версия SDK
        targetSdkVersion 26 //максимальная поддерживаемая версия
        versionCode 1 //версия самого приложения. Для обновлений
        versionName "1.0" //номер версии, видный пользователю
        testInstrumentationRunner "android.support.test.
            runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-
                android.txt'), 'proguard-rules.pro'
        }
    }
}
dependencies {
```

```

implementation fileTree(dir: 'libs', include: ['*.jar'])
implementation 'com.android.support:appcompat-v7:26.1.0'
implementation 'com.android.support.constraint:
    constraint-layout:1.0.2'
testImplementation 'junit:junit:4.12'
androidTestImplementation 'com.android.support.test:
    runner:1.0.1'
androidTestImplementation 'com.android.support.test.
    espresso.espresso-core:3.0.1'
}

```

В скобках пишется, к какому уровню (папок) принадлежит данный файл - в режиме просмотра Project он лежит соответственно в папке app, отвечающей за Android (для часов был бы wear и т.д.). В build.gradle (app) находятся инструкции для сборки нашего проекта.

В блоке dependencies хранятся библиотеки, которые мы добавляем в проект.

Всё со структурой. Итак, в файле Java у нас хранятся Java-файлы, в ресурсах хранятся ресурсы. Ресурсы делятся на подтипы. В gradle-скриптах хранятся build.gradle файлы. Самый важный из них — Module: app.

Полезные материалы: [о директории java](#) и [директориях res и assets](#)

### 1.1.3 Создание и запуск Android эмулятора

В Android Studio откроем меню ⇒ Tools ⇒ Android ⇒ AVD manager (Android Virtual Device Manager) ⇒ Create Virtual Device и тут выбираем нужную конфигурацию (или заранее собранный preset т.е. существующий телефон)

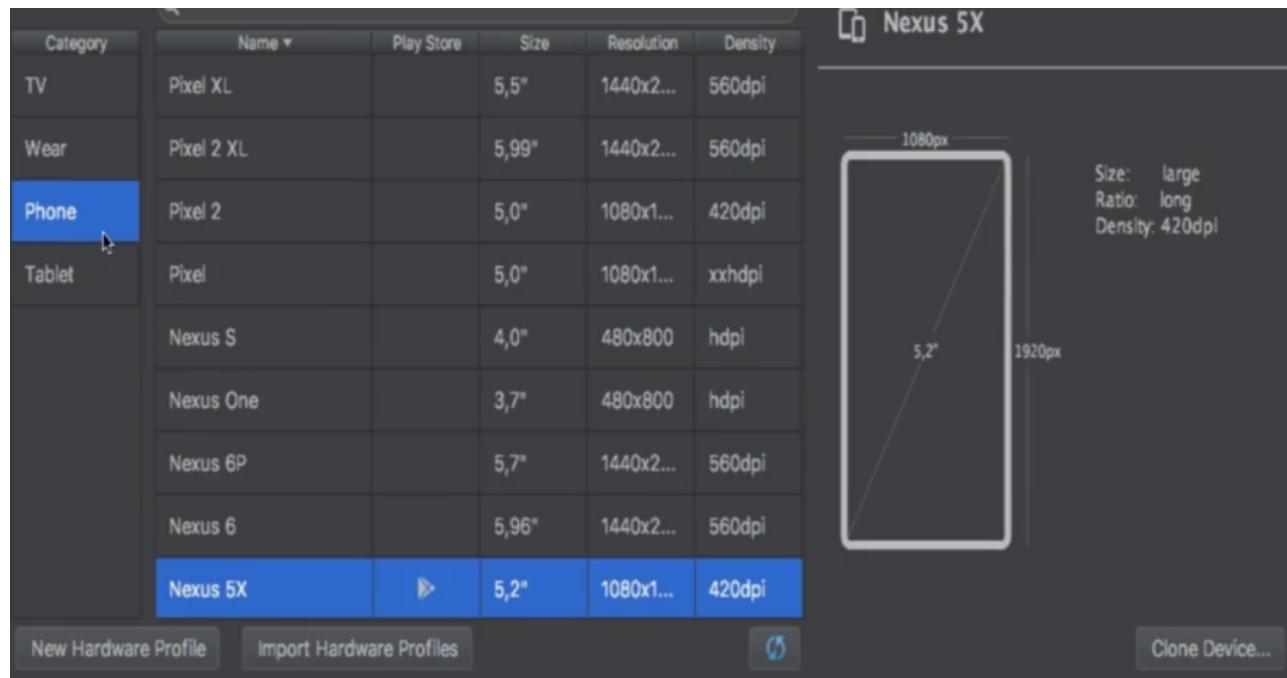


Рис. 1.7: Выбор эмулятора из готовых пресетов

Обращать внимание надо на Size (размер диагонали в дюймах), Resolution (разрешение экрана в пикселях) и Density (dpi=dots per inch т.е. точек на дюйм). [Подробнее про разрешение здесь](#). Если ни один пресет не устраивает, то можно создать свою комбинацию нажав на new hardware profile.

Мы же выберем Pixel 5,0 дюйма  $1080 \times 1920$ . Жмём Next. Выбираем операционную систему Орео т.е. API 26 (если она не установлена то закружаем через Download). Это может занять много времени). Можно и другую, главное помнить, что если мы для приложения выбирали какой-то уровень API, то наш эмулятор должен быть уровнем не ниже.

Насчёт архитектуры (ABI): Архитектура x86 либо ARM. Если у вас процессор от Intel и операционная система Windows, то эмулятор (и если процессор поддерживает технологию VTx), то эмулятор будет работать очень быстро засчет ускорения. Если у вас AMD процессор и вы работаете на Windows, то штатный эмулятор, который предлагает Android Studio, будет работать очень медленно. В этом случае вам нужно воспользоваться либо реальным устройством, либо сторонним эмулятором. Откуда брать сторонние эмуляторы, я расскажу попозже. Кроме того x86 рано или поздно упираются в 10-й уровень API Android 2.3.3. У ARM есть даже образы первого Android, уровень API 3, Android 1.5.

В последнем столбце Target выделено, что некоторые из образов имеют в скобках Google API. Это Google Play сервисы. Соответственно, даже на них можно будет запустить Play market Google Play и установить какие-то приложения и другие возможности Google Play Services. Оставляем Орео, щелкаем Next.

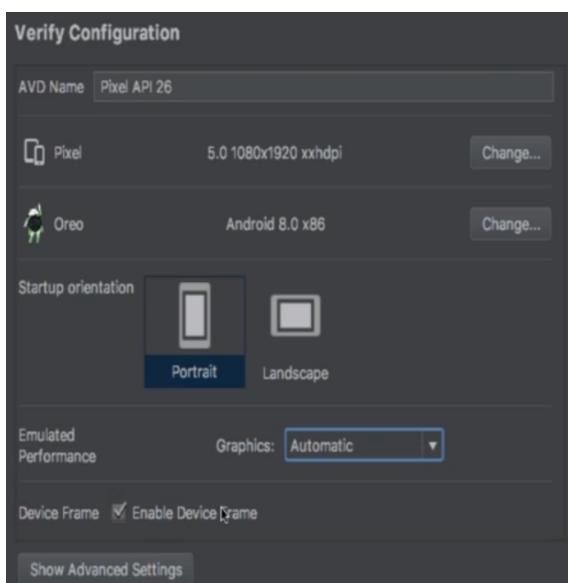
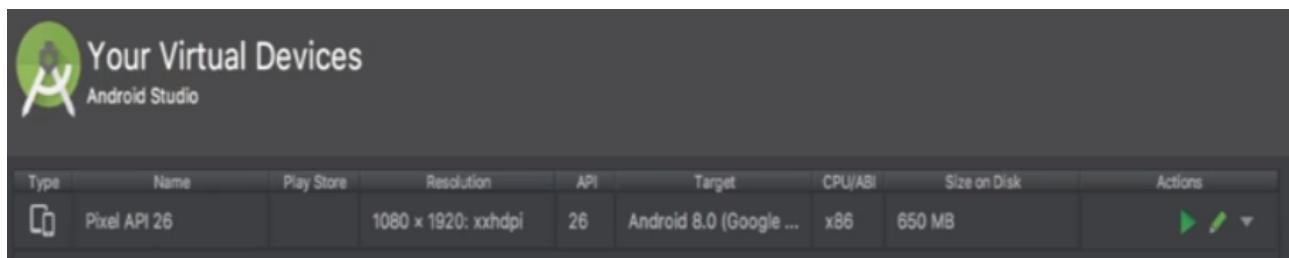


Рис. 1.8: Конфигурация эмулятора

Эмулятор создался. И мы видим такое окошко:

Открывается окно с итогами. Graphics Automatic можно не трогать. Эмулятор будет сам решать, на чем ему будет работать лучше, быстрее: либо на железном ускорении, либо на софтовом ускорении. Дальше галочка Enable Device Frame означает, что в эмуляторе будет такая декоративная окантовка в виде той оболочки, которую мы выбрали. Лучше отключим. Кликаем на Show Advanced Settings, и здесь настройки эмулятора, но самая важная настройка — это Enable keyboard input, то есть включение возможности вбивать текст в эмулятор через соответственно клавиатуру компьютера, а не набирать его в самом эмуляторе через виртуальную клавиатуру. Оставляем галочку, щелкаем Finish.



Запускаем эмулятор (зелёный треугольник) и видим экран:

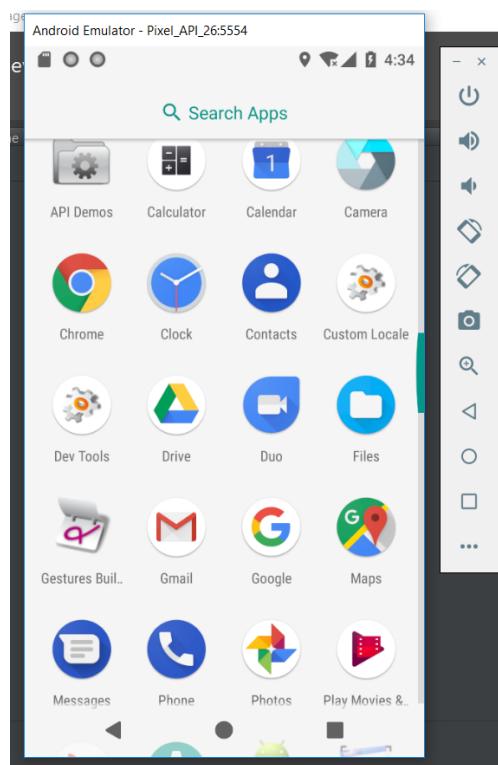


Рис. 1.9: Работающий эмулятор

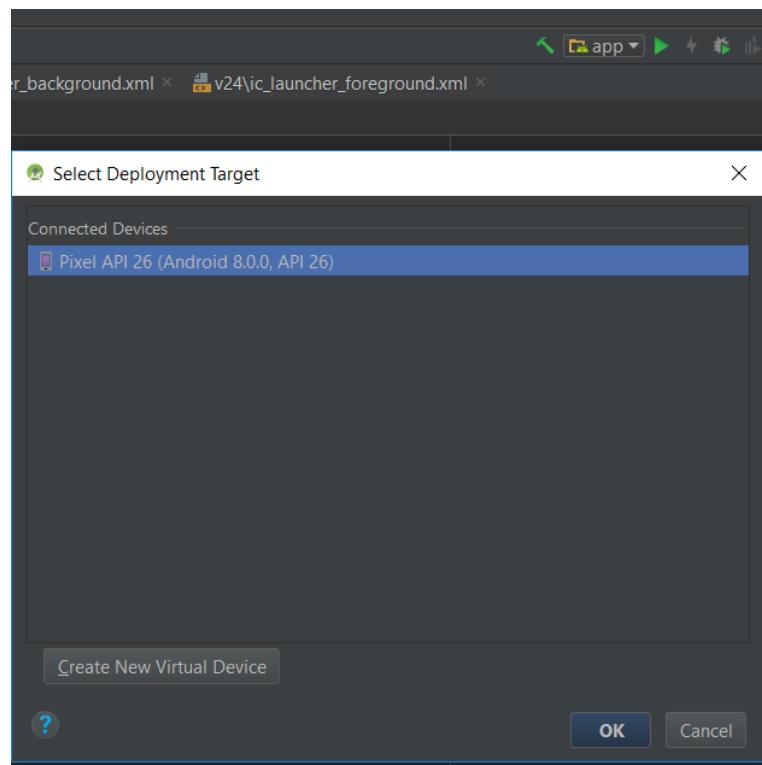


Рис. 1.10: Окно выбора эмулятора

Теперь давайте запустим на нём наш проект. Переключаемся обратно в Android Studio и щёлкаем на Run (зелёный треугольник в правом верхнем углу). И всё работает как на макете. Можно попробовать изменить размер текста дописав в activity\_launch.xml в разделе TextView

```
android:textSize="40sp"
```

Снова щёлкаем на Run (Shift+F10) и видим изменения. Android Studio 3 поддерживает instant run т.е. при изменении .xml не требуется повторная установка приложения. Проверим изменения дописав ещё

```
android:textStyle="bold"
android:textColor="@color/colorAccent"
```

И щёлкаем на Instant run (Apply Changes, на Ctrl+F10). И через пару секунд всё поменялось. Это куда быстрее чем сборка, установка и запуск с нуля. Но у этого

метода есть минусы, например, если у вас очень много классов и очень много зависимостей, то изменения в одном классе не могут применяться к изменениям в других классах. Поэтому если у вас тяжелый проект, то лучше не пользоваться Instant Run, а железно запускать заново.

#### 1.1.4 Создание стороннего эмулятора

Если ваш процессор от AMD либо от Intel, но не поддерживает VT-x, то, скорее всего, вы заметите, что штатный эмулятор, который предлагает нам Android Studio, работает очень медленно. Тогда качаете [эмулятор от Genymoution](#) (для этого надо просто зарегистрироваться на их сайте). Но в этом эмуляторе есть ограничения (на работу с камерой, геолокацией и т.д.). Этот эмулятор работает в среде VirtualBox, поэтому если его у вас нет, качаете вместе с VirtualBox. Установка такая же как и у любого приложения для Windows (через Wizard). После установки открываем сам Virtual device creation wizard, логинимся через аккаунт, который создали на сайте, кликаем Add и аналогично созданию эмулятора в AVD Manager выбираем себе нужный пресет нужной версии Android и нужной модели и качаете образ.

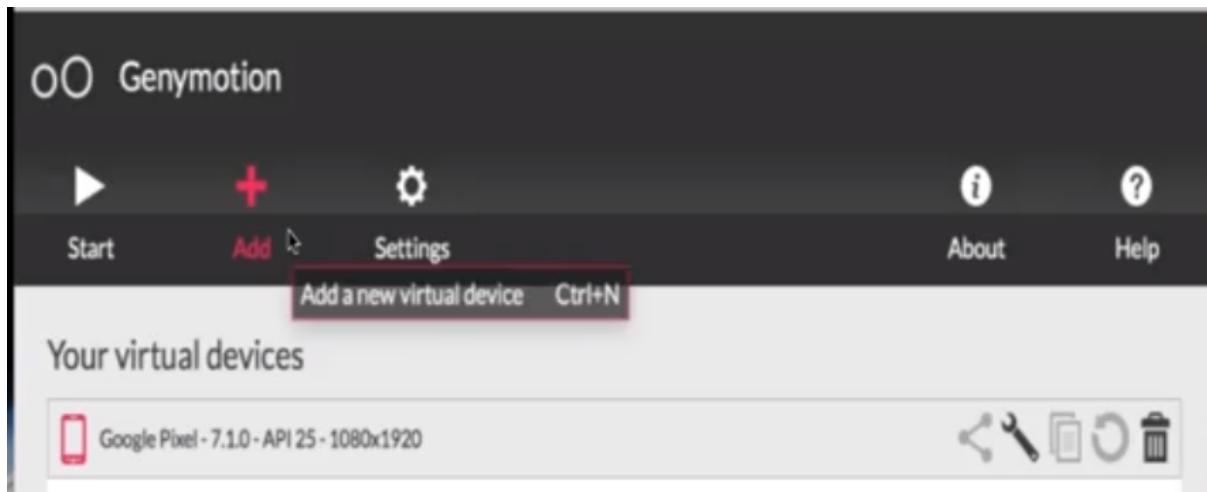


Рис. 1.11: Добавление эмулятора от Genymoution

После запуска эмулятора переходим в студию и так же в Run выбираем нужный эмулятор и включаем Instant Run. Всё запускается.

Но если по какой-то причине вам нужен другой, то можно [скачать эмулятор от Microsoft](#), в котором можно разрабатывать в среде Visual Studio на платформе Xamarin, используя язык C#, но этим мы в данном курсе заниматься не будем.

#### 1.1.5 Краткое знакомство с часто встречающимися понятиями

Для начала про работу с ресурсами. Каждый раз, когда вы добавляете новый ресурс в структуру ваших ресурсов, для него генерируется уникальный идентификатор. Для примера заходим в activity\_launch, видим код нашей Activity. И

Studio подсказывает нам, что текст у нас захардкожен (записан в самом xml коде), что неплохо было бы вместо текста Hello Android! использовать ссылку на строковый ресурс. Давайте воспользуемся этим советом, потому что это правильно.

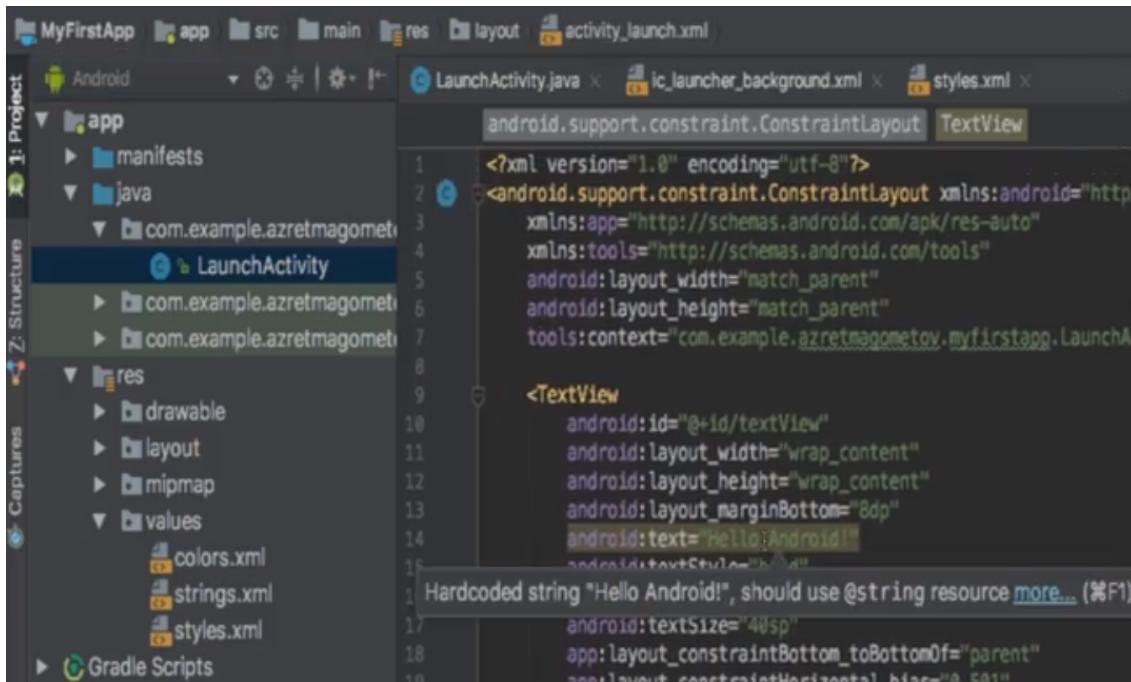


Рис. 1.12: Ошибка при использовании строки, а не строкового ресурса



Правильно хранить все строковые ресурсы в одном файле strings, вместо того чтобы хардкодить их, а потом при изменении искать по всем файлам, переименовывать, переделывать, переписывать. Плюс если вы вдруг решите перевести свое приложение с английского на французский, вам достаточно будет перевести только файл strings, так как все строки находятся в нем. Для того, чтобы это исправить наводим на значение атрибута текст(Hello Android) нажимаем Alt+Enter и выбираем в появившемся контекстном меню Extract string resource. Вбиваем название и значение ресурса. Ok.

Теперь вместо захардкоженного Hello Android! мы видим ссылку на этот ресурс. Причем ссылка вида: тип ресурса, @string и название ресурса Hello Android! Давайте перейдем в файл strings щелкнув на strings, либо зажав Ctrl (Command) и щелкнуть прямо на ресурс.

Рис. 1.13: Добавление строкового ресурса

Теперь мы можем в strings менять содержание строки и оно изменится везде, где использовалось. И все переиспользованные ресурсы крайне желательно именно хранить в ресурсах.

К ресурсам из java файлов обращаться можно строкой

```
getString(R.string.hello_android);
```

Каждый раз, когда вы добавляете новый ресурс (в папку resources) для него генерируется уникальный идентификатор, уникальный указатель. Этот идентификатор имеет числовой тип, и он хранится в классе R в своем собственном подмножестве. В нашем случае это string.

Создадим другой ресурс и посмотрим на поведение. Заходим в res⇒ values ⇒ colors.xml и пишем

```
<color name="red">#FF0000</color>
```

Возвращаемся к launch\_activity.java и обращаемся к этому цвету java-кодом:

```
getColor(R.color.red);
```

Процесс следующий: добавляем ресурс, студия для него сама генерирует указатель, который находится в классе R в соответствующем подмножестве.

Варианты R:

- color - цвета
- string - строки
- anim - анимации, attr - атрибуты, style - стили
- bool - булевые значения, integer - целочисленные значения
- drawable - картинки, mipmap - иконка запуска приложения
- layout - соответственно, layout-ы, о которых мы говорили
- styleable — это добавляемые атрибуты для своих собственных view-элементов

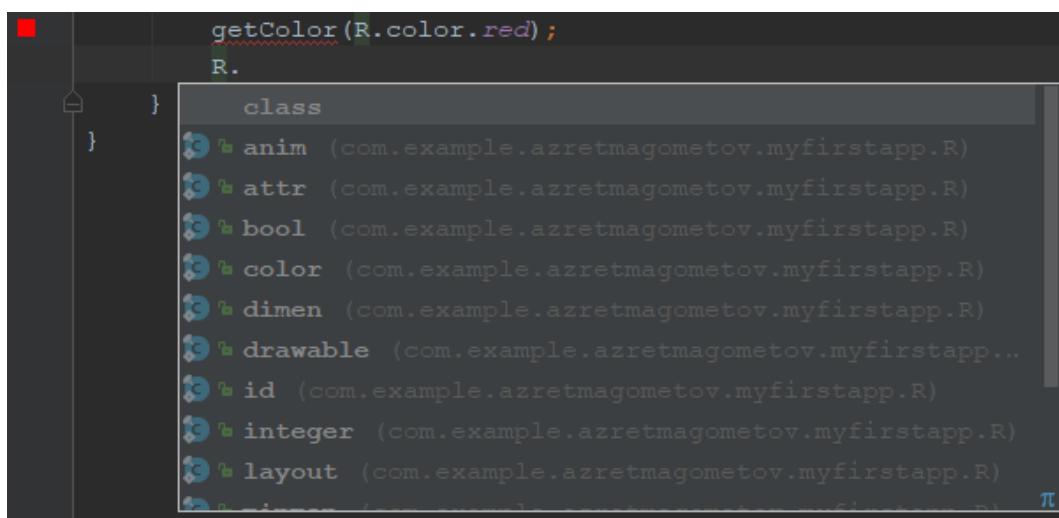


Рис. 1.14: контекстное меню возможных типов ресурсов

Заметим, что с getColor() у нас возникает ошибка:

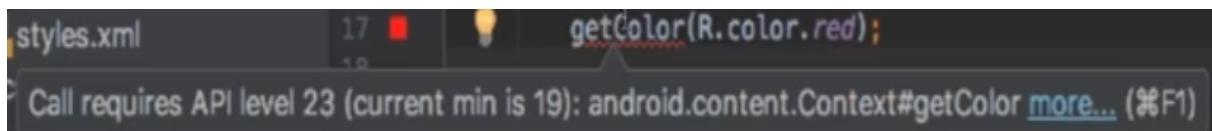


Рис. 1.15: Ошибка несоответствия API

Студия подчеркивает getColor(), потому что Call requires API level 23 (именно про это и говорилось в выборе минимального API). То есть мы не можем просто вызвать getColor(), потому что конкретно этот метод появился в 23-м API. А у нас минимальное API 19. Поэтому не исключено, что приложение будет работать на 19-22 API, и вызов этого метода вызовет ошибку. Можем нажать Alt + Enter, и Surround with — стандартный метод на проверку:

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {  
    getColor(R.color.red);  
}
```

Теперь этот метод запустится только если версия SDK не меньше, чем M - Marshmello(23 API). Либо, как я также говорил, мы можем воспользоваться методом из библиотеки поддержки. И выглядит он так: ContextCompat, (Compat указывает на то, что класс из библиотеки поддержки). getColor() this вместо Context и R.color.red. То есть этот метод не ругается на то, что у нас версия API неправильная. И мы плавно переходим к следующему понятию, следующему термину — это **Context**.

```
ContextCompat.getColor(this, R.color.red);
```

Заметим, что перед **this** появится серая надпись "context:"

Context — это глобальная точка ресурса, это такой очень мощный, много умеющий объект. Он может буквально все: доступ к ресурсам, создание view-элементов прямо в коде, доступ к возможностям телефона и т.д. Поэтому конкретно этот метод просит использовать Context. Но мы вместо Context передаем this потому что activity (а this здесь указывает именно именно на Activity) является одним из наследников контекста. То есть если мы будем щелкать на extends, в конце упремся в контекст. Поэтому во всех методах, которые принимают контекст, можно передавать Activity (либо this, либо activity.this). В нашем случае this можно заменить на LaunchActivity.this т.е. Activity может передавать себя вместо контекста — методы, которые требуют контекста.

Теперь у нас есть Java-код (вся логика в этом коде). И отдельно XML-разметка. Вопрос: как обратиться к View-элементам — к кнопке и к тексту, которые у меня определены в XML-разметке из Java-кода? Все достаточно просто: когда мы определяем View-элемент в разметке и если планируется обращаться к нему из Java-кода, мы обязательно указываем атрибут android:id :

```
android:id="@+id/textView"
```

Надпись "@+id" означает, что мы добавляем новое id. Мы можем у объектов писать любые id и потом по этим id к ним обращаться. Переименовываем в .xml файле кнопку и текст:

```
<TextView  
    android:id="@+id/tv_text"  
    ...  
<Button  
    android:id="@+id	btn_click"  
    ...  
    app:layout_constraintTop_toBottomOf="@+id/tv_text" />
```

Теперь из .java класса обратимся к элементам:

```
Button mBtnClick = findViewById(R.id.btn_click);
```

Во-первых, мы определяем тип View-элемента. У нас это кнопка. Пишем: тип класса — Button, название — mBtnClick, опять же любое, что нам нужно. И называем метод findViewById, findViewById — вот он, — в который мы передаем указатель на нашу кнопку, передаем id, которое определяли в XML-layout. И, как, вы наверное, уже догадались, указатель на кнопку тоже хранится в R-классе: R.id.btn\_click.

К слову Android Studio 3 поддерживает SmartCast, то есть раньше, когда мы обращались к View-элементам, нужно было еще кастовать найденный элемент к нужному типу. Сейчас это не нужно. Раньше findViewById возвращал просто View, а View — это класс-родитель для всех View-элементов, для всех элементов интерфейса. Сейчас он возвращает public <T extends View>View и этот метод определенного AppCompat Activity.

Стандартно в Java переменные, определённые в методе, видны только внутри этого метода. Для того, чтобы сделать mBtnClick глобальной (видной по всей activity), сделаем её переменной поля (а не метода): в начале (перед @Override) напишем

```
private Button mBtnClick;  
... // уже внутри метода onCreate  
mBtnClick = findViewById(R.id.btn_click);
```

Т.е. вне всех методов в теле класса мы её объявили. И уберём слово Button в месте, где изначально определяли нашу кнопку. Теперь мы хотим проверить, что эта кнопка та и что она работает. Добавим для неё функционал.

Для определения нажатия в Android используется стандартный Java-механизм с listener'ами, то есть со слушателями. И выглядит это так: mBtnClick.setOnClickListener(new) — Ctrl, пробел — View.OnClickListener. Что происходит? У класса View, который является родителем всех View-элементов элементов интерфейса имеется поле OnClickListener, и setOnClickListener(), метод setOnClickListener() записывает текущий OnClickListener, аргумент, в это поле. И при нажатии на кнопку

выполняется вызов метода onClick() этого listener. То есть если убрать всю мишурку, то при нажатии на эту кнопку выполнится вот этот метод. Давайте что-нибудь попробуем сделать. Мы можем изменить текст на TextView-элементе, то есть при нажатии на эту кнопку поменяется этот текст.

Сначала в res ⇒ strings.xml создадим строку, на которую будем менять старую:

```
<string name="new_text">this text is new</string>
```

Теперь изменяем наш LaunchActivity.java вот так:

Листинг 1.3: LaunchActivity.java

```
package com.example.azretmagometov.myfirstapp;
import android.os.Build;
import android.support.v4.content.ContextCompat;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

public class LaunchActivity extends AppCompatActivity {
    private Button mBtnClick; //объявили кнопку снаружи
    private TextView mText; //объявили текст снаружи

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_launch);

        final String newString = getString(R.string.new_text)
            ;
        mText = findViewById(R.id.tv_text);
        mBtnClick = findViewById(R.id.btn_click);
        mBtnClick.setOnClickListener(new
            View.OnClickListener(){
                @Override
                //при нажатии на кнопку выполняется метод onClick()
                public void onClick(View view){
                    mText.setText(newString);
                } //заранее задали в strings.xml
            });
    }
}
```

Один маленький лайфхак с шорткатами: получить строку

```
final String newString = getString(R.string.new_text);
```

можно было так: вводим "getString(R.string.new\_text);", выделяем это и нажимаем Ctrl+Alt+v и получаем "String string = " перед выделенным. Название меняем на newString и отмечаем флагок "final?" т.к. в TextView мы загоняем текст, который мы строку текста, которую мы забрали из ресурса: newString. И так как мы обращаемся к переменной из коллбэка — этот метод onClick() называется коллбэком, — то эта переменная обязательно должна быть final, то есть при компиляции кода JVM должна четко знать, что эта переменная не изменится, пока мы обращаемся к ней.

Нажимаем Run, переключаемся на эмулятор и (если наше приложение в нём обновилось), видим, что при клике текст поменяется на новый.

Теперь про **Bundle**, который мы видим в строчке

```
protected void onCreate(Bundle savedInstanceState) {
```

Bundle — это своего рода обертка над массивом данных, и он используется для передачи данных либо для сохранения и считывания каких-либо значений:

```
Bundle bundle = new Bundle(); //Создадим Bundle  
bundle.putString("KEY", "SAD");  
//вызвали метод по ключу и значению
```

значения в Bundle хранятся с помощью механизма KeyValue, то есть так же, как они хранятся в HashMap. Помимо строкового типа Bundle поддерживает byte, char, float, integer, integer array (Т.е. все примитивные типы, их массивы и строки). Еще он поддерживает **Serializable** и **Parcelable** — это специальный тип, который позволяет объекту класса сохранять свое состояние. То есть если наш класс помечен как Serializable либо Parcelable, то он его можно сериализовать и передавать в этом сериализованном виде в, допустим, в другой Activity. Bundle используются для передачи данных еще в другой Activity. И потом в другом Activity просто развернуть (десериализовать) этот файл и получить его в том же состоянии, в котором мы его оставили. Разница между Serializable и Parcelable в том, что первый — это стандартный Java-механизм, и он использует рефлексию для работы. Из-за этого он ощутимо медленнее. Parcelable — это механизм, который появился в Android. В нем мы уже сами прописываем, как что сериализовать и десериализовать. То есть мы можем сказать какие поля класса должны быть сериализованы. Или при десериализации, чтобы эти поля класса принимали вот это значение, а не то, которое они имеют на данный момент.

# Неделя 2

## *Компоненты Android и элементы интерфейса*

### 2.1 Activity и View

#### 2.1.1 Основные компоненты Android. Context

Android с точки зрения разработчика это framework, то есть глобальная библиотека, которая берет на себя всю грязную работу, например, связанную с управлением памятью, процессами, приложениями, датчиками и тому подобное. Framework предоставляет нам несколько компонентов, наследуя и изменяя которые, мы строим свое приложение. Всего таких компонентов пять:

- **Application** — это компонент, который первым загружается при начале работы с приложением и умирает последним. Он представляет собой singleton (единственный экземпляр своего класса). Реализация Application необязательна, но если вам нужна глобальная точка доступа к переменными либо методам, то Application — вполне подходящий вариант. Для того, чтобы система знала о вашем Application, необходимо указать его класс в манифесте, в node Application через tagname, иначе будет использоваться стандартная реализация.
- **Activity** — это основной компонент, с помощью которого пользователь взаимодействует с приложением. Упрощённо, Activity представляет собой какой-либо экран приложения, логику в Java-коде и интерфейс в XML-верстке. Все Activity должны наследоваться от базового класса Activity, а в методах должны обязательно вызываться методы родителя. Activity, как и другие компоненты из списка, должны обязательно указываться в манифесте, чтобы система знала о нем и могла запустить.
- **Service** — это компонент, главное предназначение которого выполнять долгие операции, которые не требуют взаимодействия с интерфейсом. Например, музыка играющая при сворачивании приложения.
- **BroadcastReceiver** — это приемник широковещательных сообщений, посыпаемых системой, другими приложениями либо вашим приложением, и

каким-либо образом реагирующий на них. Вы сами определяете, на что и как он будет реагировать. Например, приемник, подписанный на включение устройства, а реакция будет в виде запроса на сервер с обновлением данных. К слову, все мессенджеры работают подобным образом.

- **ContentProvider** — это мощный компонент, представляющий собой программный интерфейс, который позволяет нескольким приложениям пользоваться одним источником данных. Например, мессенджер имеющий доступ к телефонной книге, с которой он считывает номера или добавляет новые контакты. Или банковское приложение позволяющие переводить средства по номеру телефона, сравнивая номера из списка контактов и своей базе.

Все эти компоненты должны быть обязательно прописаны в манифесте (`AndroidManifest.xml`). Во всех этих компонентах так или иначе присутствует класс `Context`, за реализацию которого отвечает сам `Android`. Например, `Activity` — это одна из таких реализаций, так как она является потомком `Context`. Наиболее частый случай использования `Context` — это

- доступ к ресурсам приложения, будь то строки, цвета, меню, цифровые константы или `RAW`-файлы, не поддающиеся классификации.
- доступ к системным возможностям телефона, например, к сканеру отпечатков пальцев, будильнику, акселерометру или другим возможностям
- решение проблемы, когда интерфейс заранее неизвестен и зависит от данных с сервера, то для создания элементов интерфейса прямо в коде обязательно нужно передавать в конструктор этого элемента текущий контекст.
- это создание файлов, хранящихся на устройстве и используемых приложением. К примеру, база данных, скачанные файлы, сделанные фотоснимки или файлы `preferences`, в которых хранятся настройки приложения.

Класс очень важный, но напрямую с ним не работают - всё происходит через его реализации. К примеру, `Context`, который реализован в `Application`, как и сам `Application`, является singleton. При создании своих собственных singleton, которым необходим `Context` для работы, следует использовать именно `Context Application`. Напротив, `Context`, реализованные в `Activity` или `Service`, у каждого экземпляра свои. Логично, что такой `Context` живет столько, сколько живет `Activity` или `Service`. Поэтому хранить ссылки на них — не самая лучшая затея. В крайнем случае можно воспользоваться `WeakReference`, чтобы сборщик мусора мог утилизировать объект контекста и освободить ресурсы.

Дополнительно про контекст и про различие контекстов

## 2.1.2 Activity. Жизненный цикл

activity — это компонент, с помощью которого пользователь взаимодействует с вашим предложением через UI. Представляет собой один какой-либо экран приложения (логика работы) в общем случае. В любом приложении может быть несколько activity, каждая из которых представляет какой-либо экран. К примеру, экран авторизации с логикой обработки логина и пароля, главный экран приложения, чаще всего список каких либо элементов, экран настроек и другие варианты. При создании своего activity необходимо наследоваться от базового activity, одной из его реализаций или от AppCompatActivity( из библиотеки поддержки, которая упрощает работу с более ранними версиями Android). Для запуска приложения через иконку на главном экране устройства в манифесте обязательно должна быть activity с категорией Launcher. Это и будет главная точка входа в приложение. Activity — это самостоятельные единицы в том плане, что они не зависят от других activity. Являясь наследником контекста, activity имеют доступ ко всем возможностям контекста и даже больше. Activity имеют множество методов оберток, упрощающих доступ к тем или иным ресурсам. Также во все методы, которые принимают контекст, можно передавать activity, и если дело происходит в самом activity, то просто передается ключевое слово this или activity.this.

Рассмотрим такой важный механизм работы activity, как жизненный цикл. Жизненный цикл — это совокупность состояний, через которые проходит каждая activity по ходу своей работы.

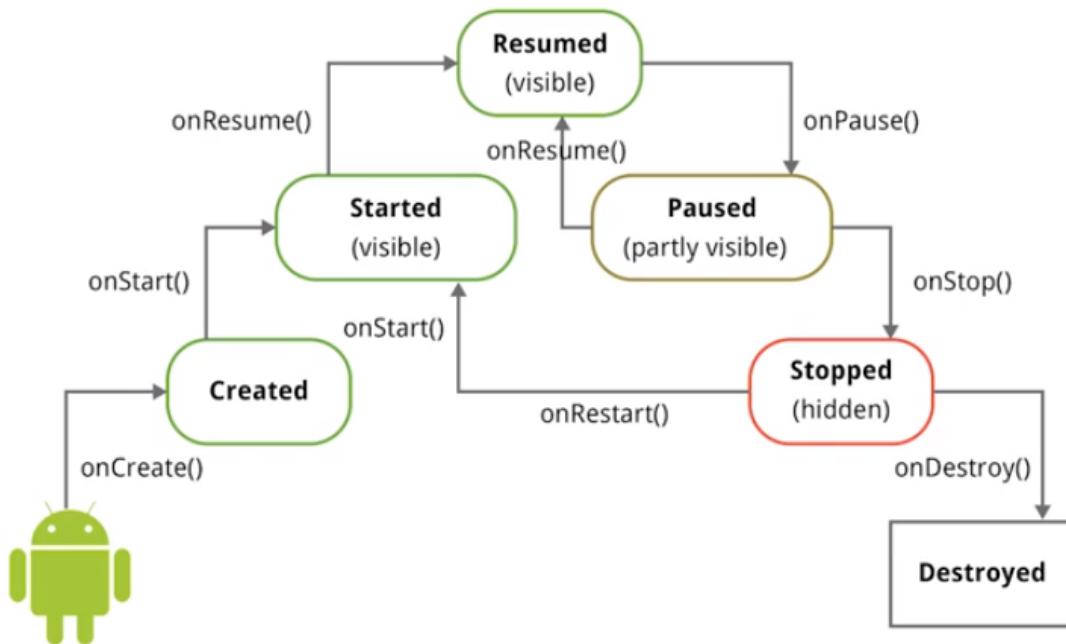


Рис. 2.1: Жизненный цикл Activity (упрощённая версия)

Переходы между состояниями происходят по требованию фреймворка. Различать их мы можем через callbacks (методы обратного вызова). Это упрощённая схема жизненного цикла Activity. Первая тройка callbacks вызывается, когда

activity готовится к взаимодействию с пользователем. Callbacks вызываются в следующем порядке:

1. **OnCreate()** - инициализируем наши поля и проводим первоначальную настройку activity, проверяя входные параметры или предыдущее состояние activity( в качестве аргумента в виде объекта bundle).  
Также задается xml-верстка интерфейса через метод SetContentView().  
После метода OnCreate() activity приходит состояние **Created** (еще не видно для пользователя).
2. **OnStart()** - мы проводим настройку, касающуюся видимой части activity.  
Например в этом методе мы можем зарегистрировать BroadcastReceiver, если он связан с изменением интерфейса.  
После onStart() activity приходит состояние **Started** (становится видимым для пользователя, но еще не готовым для взаимодействия).
3. **OnResume()** (вызывается сразу же после OnStart()) нам нужно настроить activity, чтобы оно было готово для взаимодействия с пользователем.  
В частности, мы навешиваем обработчики событий на кнопки, на прокручувающиеся списки, запускаем анимации.  
После вызова этого метода activity приходится в состояние **OnResume()** и остается в нем, пока пользователь работает с ним.
4. **OnPause()** - первый звоночек того, что пользователь собирается покинуть activity. В этом методе необходимо освободить ресурсы, связанные с взаимодействием, то есть все, то что мы добавляли в методе OnResume(). Этот метод скоротечен, и не годится для сохранения каких-либо значений.  
После него activity переходит в состояние **Paused**. Оно остается частично видимым для пользователя и должно делать все то, что ожидается от просто видимого экрана, в частности обновлять интерфейс. После этого метода может вызваться метод OnResume(), и тогда пользователь снова работает с нашим activity, либо OnStop(), и activity окончательно покинута.
5. **OnStop()** теперь activity больше не видимо для пользователя, скорее всего, потому что над ним находится другое activity, либо потому, что приложение закрыто или свернуто. Этот метод — самое лучшее место, чтобы выполнить запись в базу данных. Помимо этого необходимо освободить все ресурсы, которые не нужны, пока пользователь не пользуется activity. Опять же, это все то, что было проинициализировано в методе OnStart().  
После вызова этого метода, activity переходит в состояние **Stopped**.
6. **OnDestroy()** - activity окончательно уничтожено, вся память очищена

Рассмотрим пример запуска родительского activity, с которого мы переходим на дочернее activity и возвращаемся назад. Все activity-приложения собираются в

**backstack activity**: при запуске нового activity оно добавляется на вершину стека, при нажатии кнопки назад — удаляется.

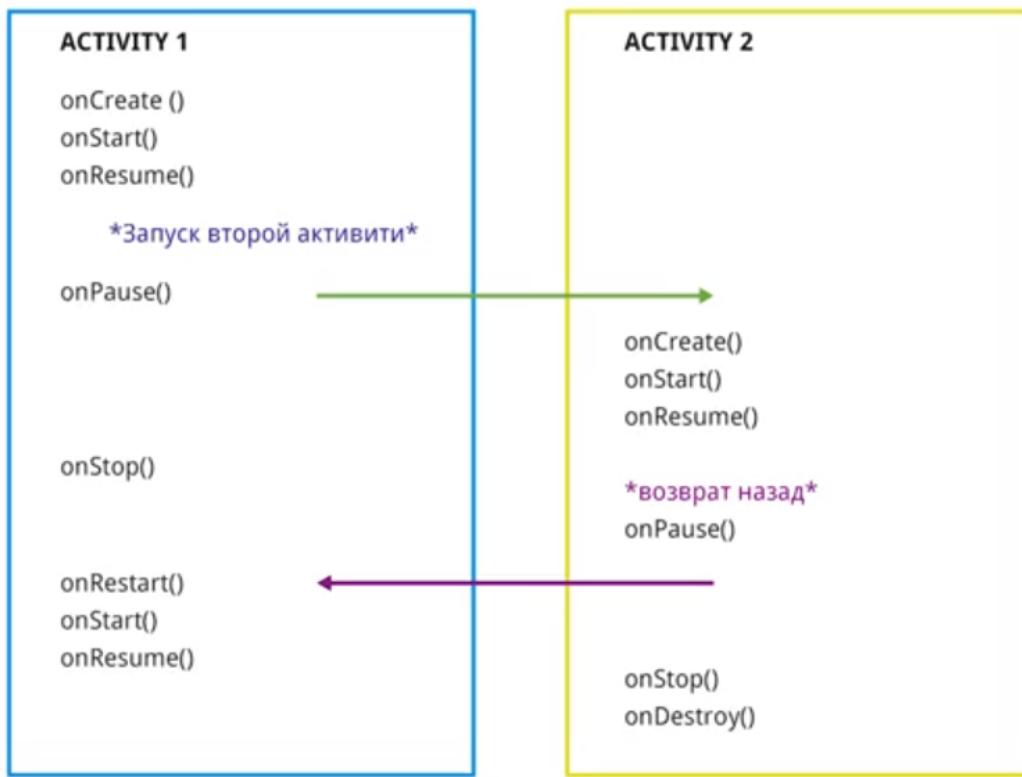


Рис. 2.2: Порядок вызова методов жизненного цикла при запуске дочерней Activity

Посмотрим, как это происходит:

1. Запускается первое activity, у него вызываются методы `OnCreate()`, `OnStart()`, `OnResume()`. Оно становится готовым для взаимодействия, находится на вершине своего стека.
2. Дальше мы запускаем дочернее activity. В родительском activity вызывается метод `OnPause()`, с ним мы больше не взаимодействуем.
3. У дочернего activity вызываются методы `OnCreate()`, `OnStart()` и `OnResume()`. И оно выходит на передний план. А у родительского activity вызывается метод `OnStop()`. Теперь родительское activity находится в состоянии `Stopped`, оно невидимо. Дочернее activity находится в состоянии `Resumed`, и оно готово к взаимодействию.
4. Если мы сейчас, будучи в дочернем activity, нажмем кнопку назад, то в дочернем activity вызовется метод `OnPause()`, оно отойдет, у родительского activity, соответственно, `OnRestart()`, `OnStart()` и `OnResume()`, оно станет готовым для взаимодействия. У дочернего Activity — `OnStop()` и `OnDestroy()`, оно будет окончательно уничтожено.

Возможны два основных случая уничтожения Activity:

- система считает, что уничтожение activity — это нормальное поведение приложения. Когда нажал кнопку «назад» либо при программном вызове метода `finish`
- ненормальным считается поведение, когда framework вынужден закрыть activity. Например, если activity, которое находится на вершине стека, испытывает какую-либо нехватку ресурсов, то activity, которые находятся под ним, могут быть уничтожены.

Для сохранения пользовательской работы при ненормальном уничтожении Activity, между `OnPause()` и `OnStop()` вызывается метод `OnSaveInstanceState()`, в котором будет сохранено текущее состояние activity. В частности, это касается view-элементов с указанными ID. Помимо этого мы можем переопределить метод `OnSaveInstanceState()` и дописать туда что-нибудь свое. Теперь, если пользователь вернется на это activity, уничтоженное activity, то вызов метода `OnCreate()` будет передан bundle с сохраненным стейтом. Если bundle не равен null, то извлекаем данные, которые мы записали.

Также существует метод `OnRestoreInstanceState()`, который вызывается, только если в bundle что-то записано. Относительно жизненного цикла этот метод вызывается после `OnStart()` и до `OnResume()`. Уничтожение activity системой — это не единственный случай сохранения стейта. Другим распространенным вариантом является смена конфигурации (в частности, поворот экрана). Пока что можете использовать вариант с `OnSaveInstanceState()`.

### 2.1.3 Интерфейс. View и ViewGroup

Для пользователя интерфейс - самая важная часть любого приложения. Android-приложениях логика и интерфейс несколько разделены. Логика находится в Java-классах, а интерфейс в XML-разметке. Подобное разделение очень удобно, так как структура интерфейса сразу бросается в глаза, и в ней легко разобраться. Для того чтобы манипулировать элементами интерфейса в Java-коде, их различают с помощью Id — идентификатора, уникального для данного XML-файла. Элементы интерфейса — обычные Java-классы, поэтому их можно создавать программно, в рантайме, если интерфейс требует такой динамики. Элемент интерфейса, который находится в XML-файле, делится на две главные категории: это `View` и его наследник `ViewGroup`. Как достаточно понятно из названия, `ViewGroup` представляет собой контейнер, в котором по тому или иному принципу располагается `View` — конкретный элемент интерфейса.

**ViewGroup** — базовый класс для всех элементов интерфейса, в том числе и для `ViewGroup`. **ViewGroup** — это базовый класс для всех классов-контейнеров, В принципе, любой интерфейс можно построить в любом `ViewGroup`-контейнере, но тот или иной `ViewGroup` лучше и быстрее справляется с поставленной задачей, и, соответственно, более предпочтителен. Например:

- **LinearLayout** располагает свои элементы друг за другом в одном направлении - вертикально или горизонтально
- **RelativeLayout** представляет возможность располагать свои элементы относительно друг друга
- **FrameLayout** удобен для того, чтобы располагать элементы друг на друге, то есть верхний элемент будет частично загораживать нижний

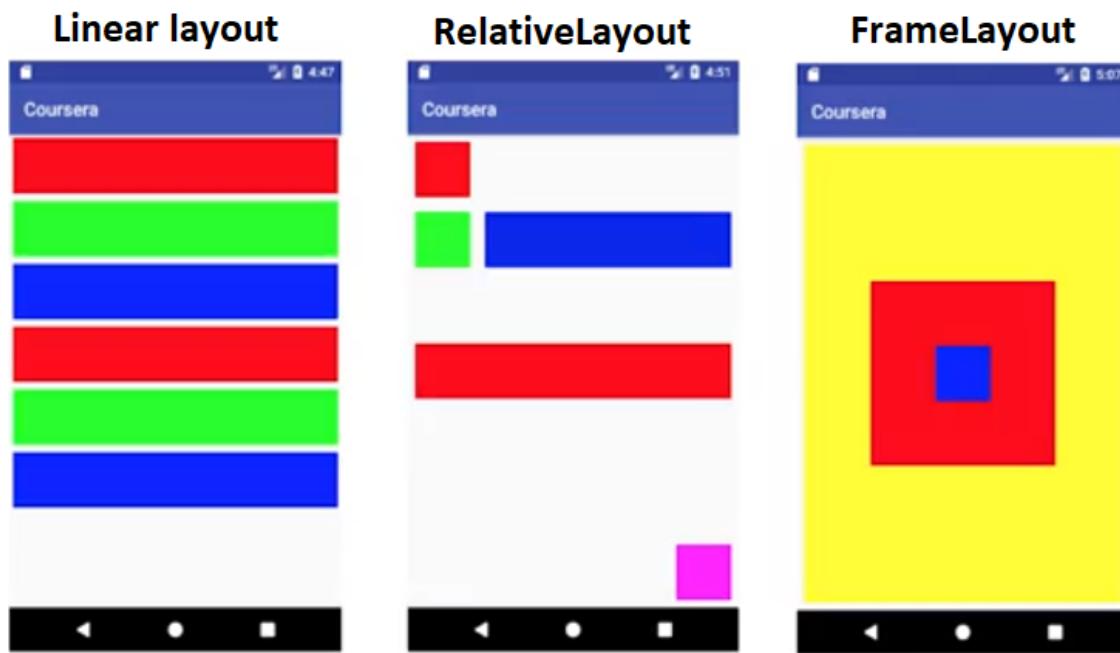


Рис. 2.3: Виды разных Layout-ов

Некоторые View встречаются повсеместно — текст в Android это TextView, поле ввода текста — EditText, кнопка — Button, картинка — ImageView, индикаторы прогресса — ProgressBar, CheckBox и RadioButton — соответственно выбор нескольких опций и одной. Все они входят в стандартный пакет Android SDK. Отдельно от вышеперечисленных элементов стоят наследники так называемого AdapterView, который используется для показа каких-либо данных в виде списка, по большей части однородного. Для заполнения View-элемента данными используется Adapter — конкретная реализация одноименного паттерна проектирования.

#### 2.1.4 Реализация ViewGroup

ViewGroup используется как контейнеры для обычных View-элементов, и реализации ViewGroup отличаются друг от друга расположениями этих элементов.

- **LinearLayout** располагает свои элементы друг за другом в одном направлении - вертикально или горизонтально. Направление задаётся атрибутом orientation

- распределяет свободное пространство по длине или ширине через веса: атрибут `layout_weight` - на дочерних View-элементах и `weightSum` - на `LinearLayout` контейнере. (Если  $weightSum \neq \sum layout\_weight$  то на разметке будет неиспользованное пространство)
  - `LinearLayout` хорошо справляется с одной задачей — располагать элементы друг за другом. При более разветвленном дизайне интерфейса верстка через `LinearLayout` может стать проблемой из-за большого количества вложенных контейнеров.
- **RelativeLayout** позволяет более гибко располагать свои дочерние элементы с помощью большого списка атрибутов
    - `layout_alignParentLeft`(Right, Top, Bottom, Start, End) — выравнивает элемент по указанному краю родителя
    - `layout_alignLeft`(Right, Top, Bottom, Start, End) — выравнивает сторону элемента по соответствующей стороне указанного элемента
    - `layout_below`(above) — элемент под/над указанным элементом
    - `layout_toLeftOf`(Right, Start, End) — элемент с указанной стороны указанного элемента
    - `layout_centerHorizontal`(Vertical) — по центру контейнера и другие
  - **FrameLayout** обычно используемый как контейнер для одной View, при добавление этой View происходит программно.
    - если есть необходимость расположить несколько View друг на друге, то `FrameLayout` — вполне подходящий вариант. Android рисует элементы в том порядке, в котором они описаны в Layout-файле.
  - **ConstraintLayout** где для расположения дочерних элементов используют так называемый constraint'ы, или правила. Если верстку можно сделать с помощью другого layout'a, то лучше воспользоваться им, иначе есть вероятность погрязнуть в пучине constraint'ов, так и не добившись желаемого результата
  - **GridLayout** и **TableLayout** выстраивают свои дочерние элементы в виде таблицы. Однако они не очень распространены из-за неудобства их использования. Тот же табличный вид можно получить с помощью других Layout

На скриншоте корневой `LinearLayout` с вертикальной ориентацией. В нем пара кнопок и вложенный горизонтальный `LinearLayout`, в котором, еще две кнопки. Веса этих кнопок указаны в числителях, в знаменателе значения `weightSum` горизонтального `LinearLayout`. Так как сумма числителя не равна знаменателю, `LinearLayout` оставляет пустое место, размером с остаток. Атрибут `layout_width`

в этом случае распространяется только на ширину кнопок, поэтому в целях производительности Studio подсказывает нам установить ширину кнопок 0dp.

Корневой LinearLayout  
(фиолетовый):  
**android:orientation="vertical"**

Вложенный LinearLayout  
(малиновый):  
**android:orientation="horizontal"**  
**android:weightSum="4"**

Кнопки горизонтального  
LinearLayout:

**android:layout\_weight="1" / "2",**  
**android:layout\_width="0dp"**



Рис. 2.4: Вложенные LinearLayout-ы

- Button 1 - выровнена по верхнему левому краю RelativeLayout
- Button 2 - находится под и справа от Button 1
- Button 3 - выровнена по центру RelativeLayout
- Button 4 - находится справа от Button 3 и нижняя граница соответствует нижней границе Button 3
- Button 5 - левый край соответствует левому краю Button 3, правый край соответствует правому краю Button 4, расположена под Button 3

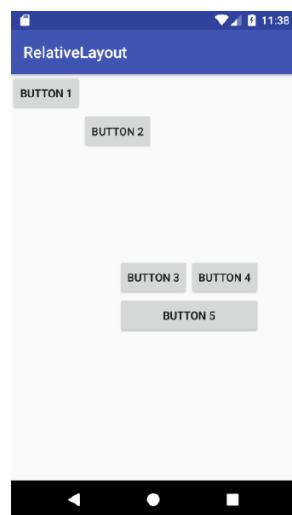


Рис. 2.5: пример RelativeLayout

```

<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="match_parent">

    <View
        android:layout_width="200dp"
        android:layout_height="200dp"
        android:layout_gravity="center"
        android:background="@color/red" />

    <View
        android:layout_width="180dp"
        android:layout_height="50dp"
        android:layout_gravity="center"
        android:layout_marginTop="60dp"
        android:background="@color/yellow" />

</FrameLayout>

```

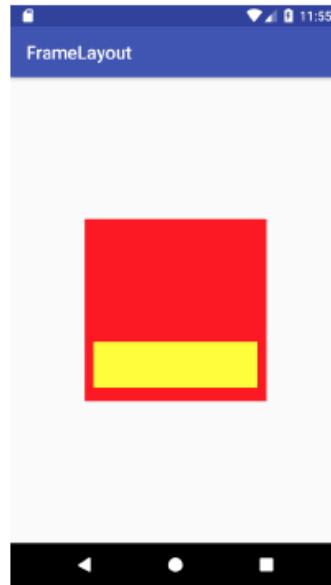


Рис. 2.6: пример FrameLayout

Чаще всего в разработке мы будем использовать Linear, Relative и FrameLayout. Остальные обязательно попробуйте сами. Кроме этого у View-элементов есть общие атрибуты. А здесь [подробнее про TextView, Button, CheckBox, RadioButton](#)

### 2.1.5 ImageView, EditText

ImageView — это элемент интерфейса, который заточен показывать какие-либо изображения. Само изображение в xml можно задать через атрибут src и в качестве значения указать ссылку на файл в папке drawable. В xml:

```
android:src="@drawable/my_image"
```

В Java:

```
image.setImageResource(R.drawable.my_image);
```

ImageView — это один из немногих элементов, размеры которого чаще всего задаются в конкретных значениях. Для масштабирования изображения используется атрибут ScaleType, имеющий возможные значения: fitXY, fitCenter, fitStart, fitEnd, center, centerCrop, centerInside, matrix:



Рис. 2.7: Различные значения атрибута `ScaleType`

На картинке создано `ImageView`, с фиолетовым бэкграундом и изображением маскота Андроида, чтобы можно было легко понять, где заканчивается граница картинки и самой `View`.

1. `fitXY` — растягивается по высоте и ширине до размеров `ImageView`
2. `FitCenter` — (по умолчанию) изображение выравнивается по центру и масштабируется, пока не упрется в одну из сторон. Соотношение сторон не нарушается, изображение будет видно полностью.
3. `FitStart` — масштабирование работает как у `FitCenter`, но выравнивается у верхнего края `View`.
4. `FitEnd` — то же самое, только изображение выравнивается у нижнего края `View`.
5. `Center` — изображение выравнивается по центру, но не масштабируется, то есть если изображение не помещается, то оно обрежется, и наоборот, если `View` больше, чем изображение, то останется пустое пространство.

6. CenterCrop — изображение масштабируется до тех пор, пока не заполнит всю View, а не поместившаяся часть изображения обрежется.
7. enterInside — изображение масштабируется до тех пор, пока стороны не будут равны или меньше соответствующих сторон View, но маленькое изображение не будет увеличиваться, чтобы заполнить большую View.
8. matrix — изображение масштабируется согласно матрице, которая задается в коде через метод SetImageMatrix. Матрица используется для поворота, масштабирования, наклона и перемещения изображения, причем необязательно привязываться к ImageView.

Теперь про **EditText**. Это поле ввода текста, причем формат введенного текста и соответствующая ей клавиатура могут задаваться с помощью атрибута **InputType** (текст, номер, электронная почта, пароль, дата и другие).

Также есть возможность кастомизировать кнопку ввода на клавиатуре и показать вместо нее какую-нибудь другую через атрибут **imeOptions** (варианты actionDone, actionSearch, actionNext и другие).

Другой полезный атрибут **hint** используется для того, чтобы показать подсказку, что именно следует ввести в поле ввода.

Также для того чтобы задать максимальное количество символов в элементе, используется атрибут **maxLength**.

## Всё о работе с ресурсами приложения О квалификаторах

## 2.2 Инструменты сборки и отладки

### 2.2.1 Система сборки Gradle

Gradle — это система автоматической сборки, которая была разработана для расширяемых многопроектных сборок. Gradle поддерживает инкрементальные сборки, самостоятельно определяя, какие компоненты дерева сборки не изменились и какие задачи, зависимые от этих компонентов, не требуют перезапуска. Сам по себе Gradle ничего не знает о том, как собирать Android-проект. В этом ему помогает плагин, который разрабатывается и развивается вместе с Android SDK. В основном сборочном скрипте Build Gradle студия самостоятельно добавляет зависимость от этого плагина. Мы будем работать с недавно вышедшим Gradle 4.1.

Самая популярная возможность — это добавление сторонних зависимостей т.е. с помощью команды `implementation` во время сборки в наше приложение загружаются все исходные коды зависимостей, которые мы указали в данном блоке.

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    androidTestImplementation('com.android.support.test.espresso:espresso-core:2.2.2') {
        exclude group: 'com.android.support',
                module: 'support-annotations'
    }
    implementation 'com.android.support:appcompat-v7:26.1.0'
    implementation 'com.android.support.constraint:constraint-layout:1.0.2'
    testImplementation 'junit:junit:4.12'
    implementation 'org.jetbrains:annotations:15.0'
}
```

В версиях Gradle до 4-й вместо `implementation` использовалась команда `compile`. Gradle также позволяет собирать сборки, которые частично будут отличаться друг от друга. Делается это с помощью так называемых `productFlavor` и `buildConfig`. `productFlavor` — это Gradle-механизм, который полезен при создании нескольких версий одного приложения, разница в которых имеет значение для конечного пользователя, например платная или бесплатная версия приложения. Для этого достаточно добавить несколько строк кода в раздел `Android` файла `build.gradle`.

```

flavorDimensions 'buildType'
productFlavors {
    free {//бесплатная версия приложения
        buildConfigField 'String', 'YOUR_PARAM_NAME',
            '"YOUR STRING VALUE"'
        versionName "0.0.1-free"
        buildConfigField 'String', 'API_BASE_URL',
            '"https://yourapi.com/free/"'
    }
    pro {//платная версия приложения
        versionName "0.0.1-pro"
        buildConfigField 'String', 'API_BASE_URL',
            '"https://yourapi.com/pro/"'
    }
}

```

Альтернативно можно расписать различные buildTypes. Мы используем build-тайпы, чтобы собирать приложения с существенными различиями. Эти отличия имеют большое значение для разработчиков. Например, сборка для отладки или сборка для релиза, сборка с обfuscацией кода или без него, каким сертификатом будет подписано приложение и тому подобное. Соответственно, перед сборкой приложения мы выбираем конкретный build-тайп и productFlavor. В совокупности эти две сущности составляют Build Variant.

```

buildTypes {
    debug {
        buildConfigField 'String', 'API_BASE_URL',
            '"https://yourapi.com/debug/"'
    }
    beta {
        buildConfigField 'String', 'API_BASE_URL',
            '"https://yourapi.com/beta/"'
    }
    release {
        buildConfigField 'String', 'API_BASE_URL',
            '"https://yourapi.com/release/"'
    }
}

```

Gradle также позволяет добавлять статические поля в собираемый проект с помощью указания buildConfigField. Это поле может использоваться для любых целей и может меняться в зависимости от build-тайпа и productFlavor. В каждом из build-тайпов или productFlavor может содержаться одно и то же поле, но с разными значениями. Учтите, что в случае с одним и тем же полем и в build type, и в product flavor значение будет браться из build type.

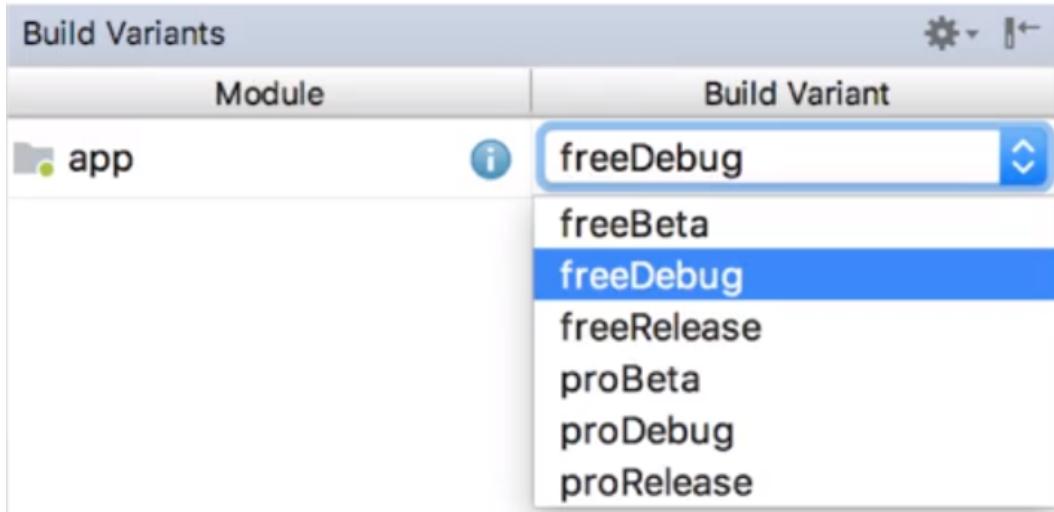


Рис. 2.8: Варианты сборки

Кстати, поле, записанное в defaultConfig будет присутствовать в каждом варианте сборки, но его можно переопределить в конкретном build-тайпе или product failer. Синтаксис buildConfigField достаточно прост: мы пишем buildConfigField и далее тип, имя поля и значение.

#### Листинг 2.1: BuildConfigField

```
defaultConfig {
    applicationId "com.elegion.coursera"
    minSdkVersion 16
    targetSdkVersion 26
    versionCode 1
    versionName "1.0"
    testInstrumentationRunner
        "android.support.test.runner.AndroidJUnitRunner"
    buildConfigField 'String', 'YOUR_NAME',
        '"DEFAULT STRING"'
}
```

Во время сборки проекта Gradle сгенерирует класс buildConfig, в котором будут храниться наши buildConfig-поля. Чтобы использовать эти поля внутри приложения, нужно обращаться к ним как к обычным public static полям.

## Листинг 2.2: Класс BuildConfig

```

public final class BuildConfig {
    public static final boolean DEBUG = false;
    public static final String APPLICATION_ID =
        "com.e_legion.coursera";
    public static final String BUILD_TYPE = "beta";
    public static final String FLAVOR = "free";
    public static final int VERSION_CODE = 1;
    public static final String VERSION_NAME = "0.0.1-free";
        // Fields from build type: beta
    public static final String API_BASE_URL =
"https://yourapi.com/beta/";
        // Fields from product flavor: free
    public static final String YOUR_PARAM_NAME = "YOUR
        STRING VALUE";
        // Fields from default config.
    public static final String YOUR_NAME = "DEFAULT STRING
        VALUE";
}

```

**Android Debug Bridge (ADB)** - консольное приложение, которое позволяет с помощью терминала управлять подключенными устройствами (эмуляторами и реальными девайсами).

[подробнее про ADB](#). Ещё немного про сам [Debug](#)

### 2.2.2 Toast

Toast (также тост) — это всплывающее сообщение. Легко реализуется. Может использоваться как в целях дебага, так и для уведомления пользователя.

```

//статический метод для создания тоста
Toast.makeText( //вызов статического метода для создания
    MainActivity.this, //1 параметр контекст
    "Connection lost", //2 параметр текст (или R.string.
        ресурс)
    Toast.LENGTH_LONG) //3 параметр - продолжительность
.show(); //вызов метода для показа тоста на экране

```

LENGTH\_SHORT — 2 секунды, LENGTH\_LONG — 3.5 секунды

Чтобы создать кастомный Toast, нужно:

1. Создать желаемую XML верстку
2. Объявить и инициализировать переменную класса Toast
3. Преобразовать верстку во View с помощью LayoutInflater

4. Задать желаемые значения для элементов View

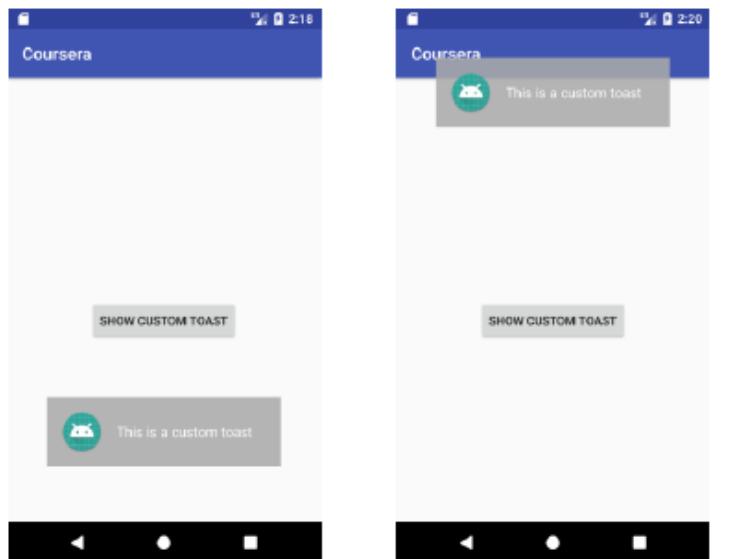
5. Применить метод toast.setView(View v).

доп можно задать длительность и gravity.

Пример кастомного toast:

```
public void showCustomToast() {  
    LayoutInflator inflater = getLayoutInflater();  
    View layout = inflater.inflate(R.layout.custom_toast,  
        null);  
    TextView text = layout.findViewById(R.id.tvTitle);  
    text.setText("This is a custom toast");  
    ImageView image = layout.findViewById(R.id.image);  
    image.setImageResource(R.mipmap.ic_launcher_round);  
    Toast toast = new Toast(this); // контекст  
    toast.setView(layout);  
    toast.show();  
}
```

Вот как это будет выглядеть с атрибутом gravity:



toast.setGravity(Gravity.TOP, 0, 100);

0, 100 - отступы  
от притягивающей стороны (TOP)  
по X и Y, в пикселях

Рис. 2.9: Toast с изменённым атрибутом gravity

### 2.2.3 Menu, ContextMenu

Меню можно создать как в коде, так и через xml ресурс. Во втором способе правой кнопкой кликаем по res ⇒ New ⇒ Android resource file. В появившемся окне выбираем resource type - Menu и указать название. Создастся заготовка.

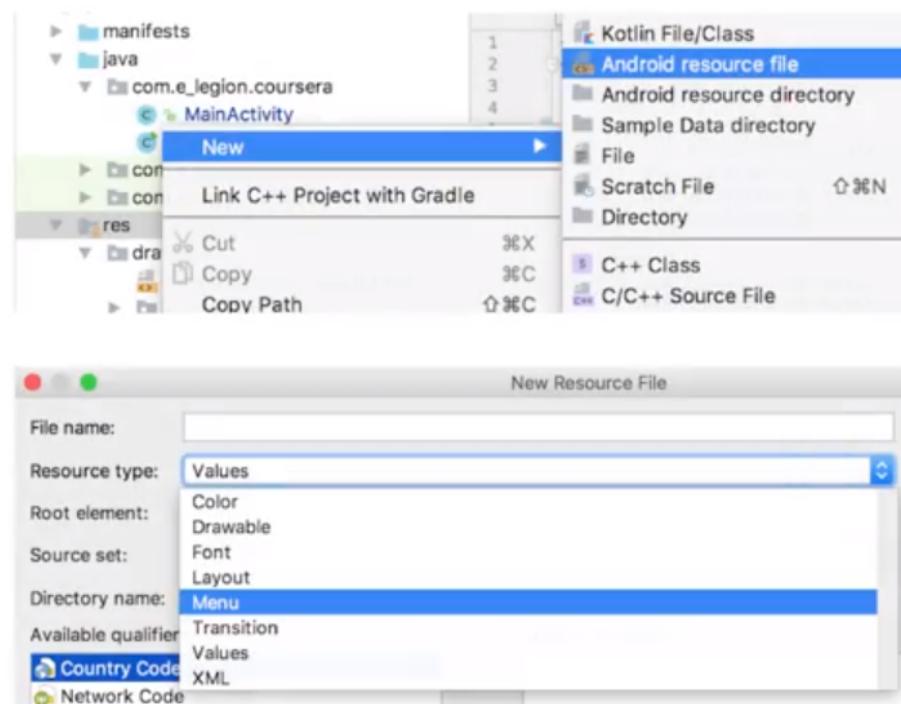


Рис. 2.10: Создание файла menu

Пример заполненного меню

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:app="http://schemas.android.com/apk/res-auto"
      xmlns:android="http://schemas.android.com/apk/res/
      android">
    <item android:id="@+id/settings"
          android:title="Settings"
          android:icon="@drawable/ic_settings"
          app:showAsAction="never" />
    <item android:id="@+id/search"
          android:title="Search"
          android:icon="@drawable/ic_search"
          app:showAsAction="never" />
    <item android:id="@+id/logout"
          android:title="Logout"
          app:showAsAction="never"/>
</menu>
```

id - идентификатор. title - название, видное пользователю, icon - иконка, видная пользователю. ShowAsAction - переместить в тулбар: never - никогда, always - всегда (не рекомендуется), ifRoom - если есть место, можно добавить |withText - вместе с title если есть место.

Добавим меню в activity:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.main_menu, menu);
    //наш объект меню мапится на созданный ресурс меню
    return true;
}
```

Возвращаемое значение булевского типа - будет ли меню показано, или нет. Т.е. можно добавить в метод условие и решать, будет меню показано или нет. Аргумент - объект типа Menu - наше меню, которое будет показываться. Теперь обрабатываем нажатия на пункты (логика работы с меню):

```
@Override
public boolean onOptionsItemSelected(MenuItem item){
    switch (item.getItemId()) {
        case R.id.settings:
            Toast.makeText(this, "Settings clicked",
                    Toast.LENGTH_SHORT).show();
            return true; //показываем, что мы сами вызвали метод
        case R.id.search:
            Toast.makeText(this, "Search clicked",
                    Toast.LENGTH_SHORT).show();
            return true;
        default: return super.onOptionsItemSelected(item);
    }
}
```

Если же предполагается менять меню в рантайме, то используем другой метод:

```
@Override
public boolean onPrepareOptionsMenu (Menu menu) {
    if (!isUserAuthorized()) {
        menu.removeItem(R.id.logout);
        return true;
    }
    return super.onPrepareOptionsMenu(menu);
}
```

Здесь у залогиненного пользователя будет возможность разлогиниться.

Теперь рассмотрим **КОНТЕКСТНОЕ МЕНЮ** - меню, вызываемое при долгом нажатии на элементе экрана, и наполненное специфичными для данного элемента

пунктами. Элемент и пункты контекстного меню мы определяем сами. Допустим, что у нас есть TextView, и мы хотим повесить на него контекстное меню. Рассмотрим его создание:

```
mHelloTv = findViewById(R.id.tv_hello_world);
registerForContextMenu(mHelloTv);
//включение контекстного меню (в onResume())
unregisterForContextMenu(mHelloTv);
//отключение контекстного меню (в onPause())
public static final int GROUP_ID = Menu.NONE;
public static final int MENU_ITEM_ID = 42;
public static final int ORDER = Menu.NONE;
@Override
public void onCreateContextMenu(ContextMenu menu, View v,
ContextMenu.ContextMenuInfo menuInfo) {
    if (v.getId() == R.id.tv_hello_world) {
        // динамическое заполнение методом add(), но можно как и в
        // обычном
        menu.add(GROUP_ID, MENU_ITEM_ID, ORDER, "Context menu"
            );
    } else {
        super.onCreateContextMenu(menu, v, menuInfo);
    }
}
```

ContextMenu.ContextMenuInfo - предоставляет какую-нибудь информацию об объекте. Для обработки нажатия используется метод:

```
@Override
public boolean onContextItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case MENU_ITEM_ID:
            Toast.makeText(this, "Context menu clicked",
                Toast.LENGTH_SHORT).show();
            return true;
        default: return super.onContextItemSelected(item);
    }
}
```

Кроме разобранного ещё есть: **ActionView** — это View, которое появляется в тулбаре при нажатии на пункт меню. И распространенный пример такого View — это строка поиска. **ActionMode** — режим для работы с элементами.



Рис. 2.11: Как выглядят ActionView и ActionMode

Например, долгое нажатие на фотографии в галерее приведет к активации ActionMode. В тулбаре интерфейс изменится. Вы сможете выбрать несколько фотографий и, например, удалить их. Либо нажать кнопку «Назад» и выйти из ActionMode. Кроме этого полезно будет разобраться с материалами:

[Подробнее о Intents \(неявные\), IntentFilters](#)

[BackStack Activity, launchMode, intentFlags, taskAffinity](#)

## 2.3 Фрагменты и файлы Preferences

### 2.3.1 Знакомство с Fragment

**Fragment** — это модульная часть activity, у которой свой жизненный цикл и свои обработчики различных событий.

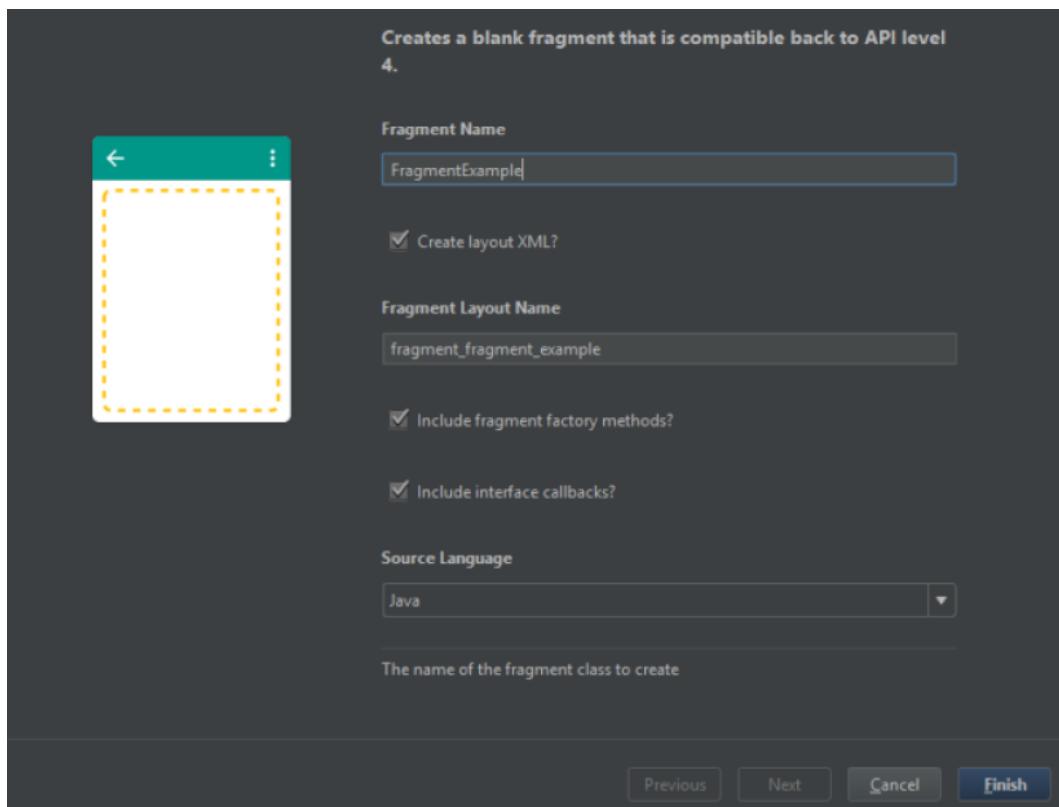


Рис. 2.12: Создание фрагмента

По сути, это подобие Activity — кусок интерфейса с логикой, но Fragment содержится внутри Activity. И, соответственно, одна Activity может отображать

одновременно несколько фрагментов. Фрагменты появились в Android с API11 (Android 3.0), для разработки более гибких пользовательских интерфейсов на больших экранах, таких как экраны планшетов. Но впоследствии они нашли применение не только для этого. Через некоторое время была написана библиотека Android Support Library, которая добавляла поддержку фрагментов и в более ранней версии Android. Существует два основных подхода в использовании фрагментов:

1. Первый основан на замещение родительского контейнера. Создается разметка, и в том месте, где нужно отобразить фрагменты, размещается контейнер, чаще всего это FrameLayout. В коде в контейнер помещается фрагмент.

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.
    com
        /apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/fr_container"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"/>
```

2. фрагмент добавляется в разметку через тег fragment, но учтите, что его потом нельзя будет поменять. То есть указанные фрагменты будут использоваться до конца работы Activity.

```
<fragment
    android:id="@+id/fragment_example"
    android:name="com.elegion.coursera.ExampleFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

При работе с фрагментами мы используем три основных класса:

- **Fragment**, от которого наследуются все фрагменты
- **FragmentManager** — с помощью экземпляра этого класса происходит все взаимодействие между фрагментами и Activity
- **FragmentTransaction** — сама транзакция.

Также существуют различные подклассы фрагментов: DialogFragment, WebViewFragment, MapFragment и тому подобное.

Рассмотрим пример реализации фрагмента:

```

public class ExampleFragment extends Fragment {
    private TextView mExampleText;
    public static ExampleFragment newInstance() {
        return new ExampleFragment();
    }
    @Override
    public View onCreateView(LayoutInflater inflater,
    ViewGroup container, Bundle savedInstanceState) {
        View view = inflater.inflate
            (R.layout.fr_example, container, false);
        mExampleText = view.findViewById(R.id.tv_example);
        mExampleText.setText("Example text");
        return view;
    }
}

```

**LayoutInflater** — это класс, который позволяет построить нужный макет, получая информацию из указанного XML-файла Layout. Обратите внимание на то, что метод `inflate()` получает три параметра. Во-первых, это разметка нашего фрагмента, второе — контейнер, и булево значение `attachToRoot`. Если установлено значение `true`, то после вызова функции `inflate()` полученный View будет добавлено в Layout-контейнер. Но при работе с фрагментами здесь всегда будет `false`, так как фрагмент подключается к разметке через `FragmentManager` (добавить фрагмент, заменить фрагмент, удалить фрагмент). Его методы:

- `FragmentManager transaction = getFragmentManager().beginTransaction();`
- `transaction.add(R.id.fr_container, fragment);`
- `transaction.replace(R.id.fr_container, fragment);`
- `transaction.remove(fragment);`
- `transaction.hide(fragment);`
- `transaction.show(fragment);`
- `transaction.detach(fragment);`
- `transaction.attach(fragment);`
- `transaction.commit();`

Также обратите внимание на метод `newInstance()` — этот метод мы будем использовать для создания экземпляра нашего фрагмента. Конструкторы фрагментов не рекомендуется переопределять, так как фрагмент может быть уничтожен и восстановлен системой, и если не будет подходящего конструктора,

то приложение просто крашнется. В отличие от Activity, фрагмент не нужно добавлять в AndroidManifest.

Для того чтобы добавлять наши фрагменты в стек так, как это работает по умолчанию с Activity, можно вызвать метод **addToBackStack()** у FragmentManager. Чтобы перейти на предыдущий экран с фрагментом, нужно вызвать метод **popBackStack()**, но для правильной работы нужно, чтобы при вызове **popBackStack()** в стеке был хотя бы один фрагмент, иначе мы получим просто пустоту в контейнере фрагмента. По идее нужно проверять этот момент и закрывать Activity, если в стеке остался последний фрагмент.

```
transaction.addToBackStack(fragment.getClass() .
    getSimpleName());
getFragmentManager().popBackStack();

@Override
public void onBackPressed() {
    if (getFragmentManager().getBackStack
        EntryCount() > 0) {
        getFragmentManager().popBackStack();
    } else {
        super.onBackPressed();
    }
}
```

С помощью метода **getActivity()** во фрагменте можно получить ссылку на HostActivity. Кстати, если наша HostActivity пересоздавалась, то FragmentManager сам должен восстановить все фрагменты. Проще всего уточнить этот момент можно спомощью проверки **savedInstanceState** на null.

```
getActivity();
if(savedInstanceState == null) {
    //то добавляем фрагмент
    ExampleFragment fragment = ExampleFragment.newInstance();
    FragmentManager fragmentManager = getFragmentManager();
    fragmentManager.beginTransaction()
        . add(R.id.fr_container, fragment)
        . addToBackStack(ExampleFragment.
            class.getSimpleName()).commit();
}
```

Жизненный цикл фрагмента отличается от жизненного цикла Activity, но состояние и Activity, и находящихся в ней фрагментов всегда коррелирует.  
Жизненный цикл фрагмента(ф.):

1. **onAttach()** — ф. присоединяется к Activity. Пока Fragment и Activity еще не полностью инициализированы.
2. **onCreate()** — в этом методе можно инициализировать переменную (так же, как в Activity).
3. **onCreateView()** — создает интерфейс ф-а метод
4. **onActivityCreated()** вызывается когда отработает метод Activity onCreate(начиная с этого момента, во ф-е можно обращаться к компонентам Activity).
5. **onStart()** — вызывается, когда ф. видим
6. **onResume()** — ф. готов к взаимодействию
7. **onPause()** — ф. остается видимым, но с ним нельзя взаимодействовать.
8. **onStop()** — ф. невидим
9. **onDestroyView()** — уничтожает интерфейс ф.
10. **onDestroy()** — полное уничтожение ф-а
11. **onDetach()** — отсоединение ф-а от Activity.

#### Дополнительно о фрагментах

Для хранения пар ключ-значение в Android можно использовать [SharedPreferences](#)

### 2.3.2 Формат JSON. Библиотека GSON

**JSON** — это текстовый формат данных, легко читаемый человеком и используемый для сериализации объектов и обмена данными.

**Сериализация** это сохранение состояния объекта для передачи и последующей десериализации — восстановления.

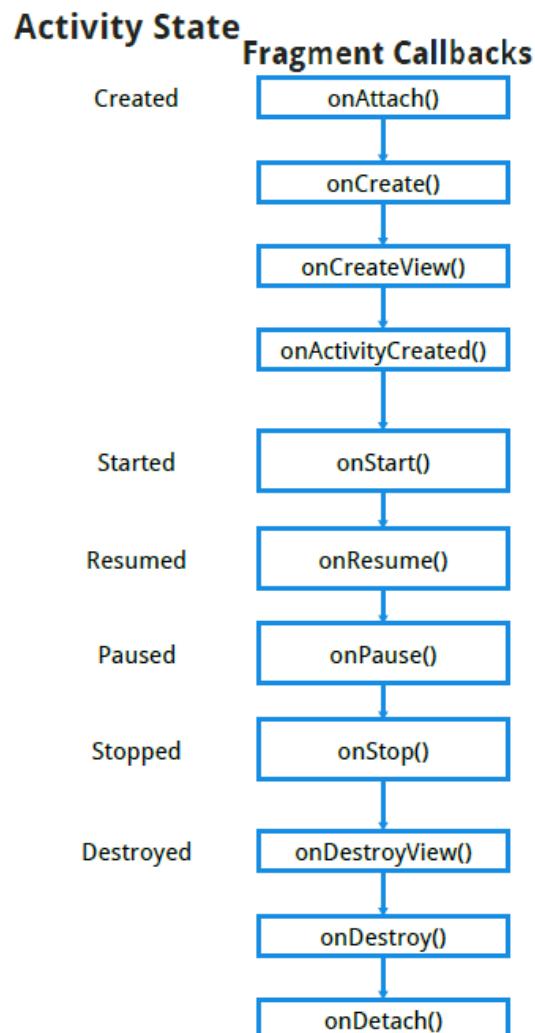


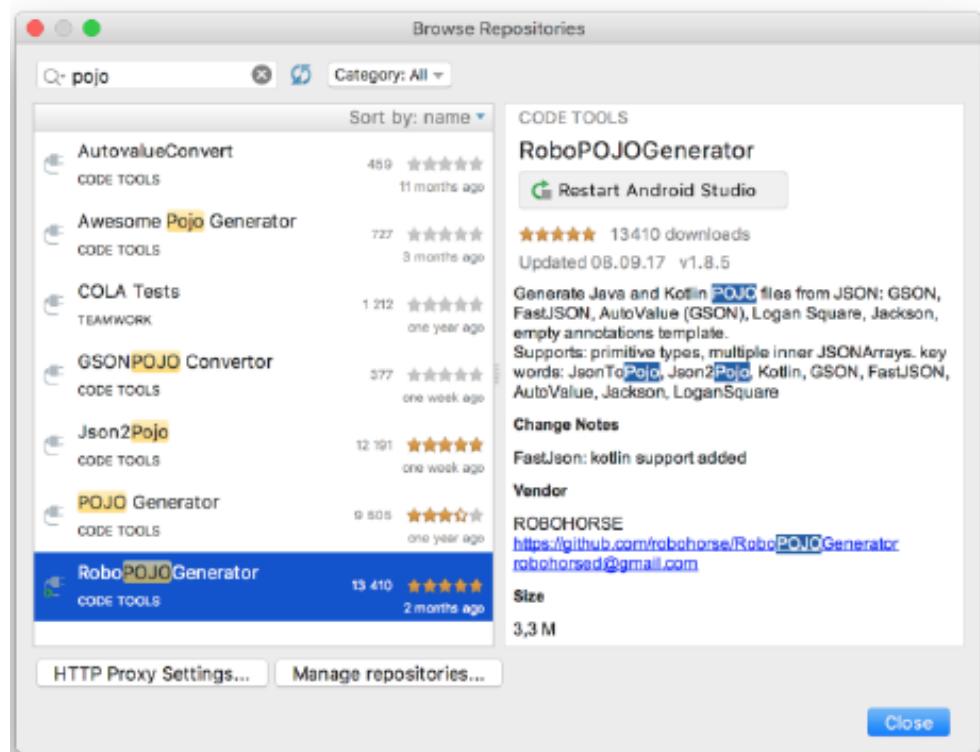
Рис. 2.13: Жизненный цикл фрагмента

В любом правильно сформированном json-файле можно выделить два основных вида структур: это поля типа ключ-значение и упорядоченный набор значений. Ключ всегда является строковым типом, а значения могут принимать вид строки, числа, литералов true, false, null, массивов либо объектов, вложенных в json. Объекты обрамляются фигурными скобками, внутри которых хранятся неупорядоченное множество пар ключ-значение. Ключ от значения отделяется двоеточием. Между парами ставится запятая. Массивы в json заключаются в квадратные скобки, между значениями также ставится запятая.

```
{
  "name": "Ivanov Ivan",
  "age": 42,
  "kids": [
    {
      "name": "Ivanov Artiom",
      "age": 8,
      "kids": null
    },
    {
      "name": "Ivanova Irina",
      "age": 4,
      "kids": null
    }
  ]
}
```

Вот пример - отображение объекта, в котором видна модель человека. Поля: имя, возраст и детей. Дети — это массив объекта такого же типа «Человек», поэтому у них есть имя, возраст и поле "kids" со значением null. Теперь попробуем собрать из этого сериализованного файла объект.

Прежде всего нам нужен класс «модель» или, как его еще называют, "pojo"— plain old java object, старый добрый java-объект. pojo-классы можно генерировать либо с помощью плагина для Studio, либо на [специальных сайтах](#).



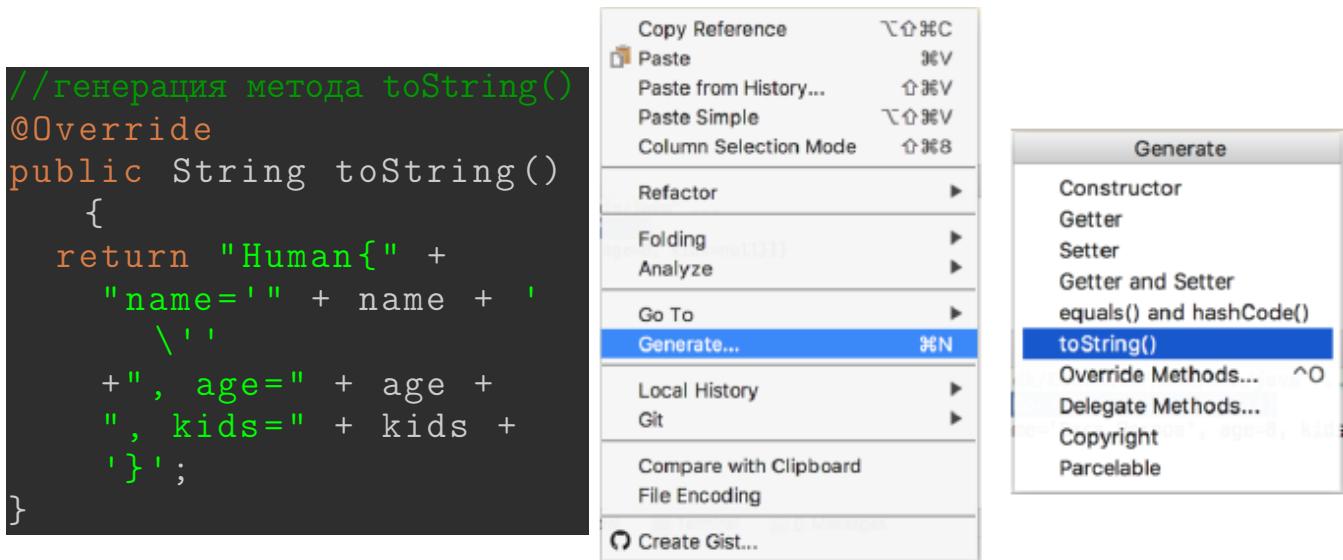
На основе json был создан роjo-класс. Теперь спокойно можно создать экземпляр этого класса и задать ему значение из json.

```
public class Human {
    @SerializedName("name")
    @Expose
    private String name;
    @SerializedName("age")
    @Expose
    private int age;
    @SerializedName("kids")
    @Expose
    private List<Human> kids = null;
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public int getAge() { return age; }
    public void setAge(int age) { this.age = age; }
    public List<Human> getKids() { return kids; }
    public void setKids(List<Human> kids){this.kids=kids;}
}
```

Это скучный процесс и были придуманы библиотеки, автоматизирующие конвертацию из json в объект и наоборот. Одна из таких библиотек — GSON. Подключим ее в файле gradle уровня app и рассмотрим простую сериализацию и десериализацию. **Сериализация** — это создание json из объекта. Сначала нам нужно создать такой объект. Создаём объект класс Human

```
Human father = new Human();
father.setAge(38);
father.setName("Petr Petrov");
Human kid = new Human();
kid.setName("Vasya Petrov");
kid.setAge(4);
kid.setKids(null);
List<Human> kids = new ArrayList<>();
kids.add(kid);
father.setKids(kids);
//Сериализация:
String json = new Gson().toJson(human);
System.out.println(json);
>{"name": "Petr Petrov", "age": 42,
"kids": [{"name": "Vasya Petrov", "age": 8}]} 
```

Теперь нам нужно у объекта класса Gson вызвать метод toJson(), который вернет нам json-строку.

Рис. 2.14: `toString()`

Для десериализации у Gson нужно вызвать метод `fromJson()`, который на вход принимает json-строку и название класса, объект которого нужно создать. Попробуем воссоздать объект, который мы только что сериализовали. И выведем его в лог. Для этого мы переопределяем метод `toString()` у класса `Human`. Проще всего это можно сделать с помощью встроенной кодогенерации в Studio. Правой кнопкой кликаем по коду класса, Generate, `toString()`. И теперь давайте сравним json и вывод `toString()`.

```
Human fromJson = new Gson().fromJson(json, Human.class);
System.out.println(fromJson.toString());
toString() - Human{name='Petr Petrov',
    age=42, kids=[Human{name='Vasya Petrov',
        age=8, kids=null}]}
json - {"name":"Petr Petrov", "age":42, "kids":
    [{"name":"Vasya Petrov", "age":8}]}
```

Сериализовать и десериализовать объекты в json с помощью библиотеки Gson это основа для клиент-серверного взаимодействия. Помимо этого сериализованные в виде строки объекты можно хранить в файле preferences либо просто на устройстве.

Свой код можно публиковать на GitHub. [про](#) [заливку](#) кода на GitHub

# Неделя 3

## Старт курсового проекта(КП)

Начнём разрабатывать android-приложение со стартовым функционалом, который может понадобиться в других приложениях. Оно будет иметь три экрана — это экран авторизации, экран регистрации и экран показа профиля.

[Архив с кодом для экрана логина](#)

[Архив с кодом добавления ссылок в activity](#) [Архив с валидацией email и password](#)

### 3.1 Activity авторизации

#### 3.1.1 КП. Вёрстка экрана логина

Создадим новый проект, как на первой неделе. Start a new Android Studio project. Назовём MyFirstApplication (мы делалис company domain = elegion.com). Next. Phone and Tablet API 19. Next. Empty Activity. Next. Activity Name = MainActivity (generate layout file и Backwards Compatibility оставляем включенными). Finish. Проект создался. Открываем app ⇒ res ⇒ layout и переименуем (shift+F6 по нему) activity\_main.xml в ac\_auth.xml. Откроем его во вкладке xml. И подчистим его так, чтобы осталось только:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/
        android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
/>
```

И добавим внутрь два поля TextView: для логина и пароля:

```
<EditText
    android:id="@+id/etLogin"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginLeft="8dp"
    android:layout_marginRight="8dp"
    android:hint="@string/login_hint"/>
```

```
<EditText
    android:id="@+id/etPassword"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginLeft="8dp"
    android:layout_marginRight="8dp"
    android:hint="@string/login_password"/>
```

layout\_marginLeft и layout\_marginRight - отступы слева и справа. hint - подсказка, причём помним, что надо создать строковый ресурс и сослаться на него, поэтому в res ⇒ values ⇒ strings.xml допишем:

```
<resources>
    <string name="app_name">MyFirstApplication</string>
    <string name="login_hint">Login</string>
    <string name="login_password">Password</string>
    <string name="login_enter">Sign in</string>
    <string name="login_register">Sign up</string>
</resources>
```

Кроме того, это можно было сделать комбинацией Alt+Enter по строке, которую надо добавить в строковые ресурсы (с.м. 1 неделю). Заметим, что отображается только одно поле. Мы забыли добавить ориентацию нашему layout. Добавим её перед полями EditText:

```
android:orientation="vertical">
```

Теперь перейдём к созданию кнопок и регистрации в приложении. Для этого надо добавить LinearLayout:

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">
    <Button
        android:id="@+id/buttonEnter"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="@string/login_enter"/>
    <Button
        android:id="@+id/buttonRegister"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="@string/login_register"/>
</LinearLayout>
```

Не забываем добавить orientation, которая на этот раз горизонтальная - мы хотим две кнопочки рядом. Обеим кнопкам ставим вес =1 (android:layout\_weight="1"). Кроме того, не забудем задать кнопкам их уникальные id (в соответствии с их назначениями buttonEnter и buttonRegister). Значения кнопок соответственно войти - sign in и зарегистрироваться sign up (помним про строковые ресурсы).

### 3.1.2 КП. Добавление ссылок на View элементы

Рассмотрим, как присоединить наш макет к Activity, и в конце видео запустим это на эмуляторе. Зайдем в наш проект. Откроем package ⇒ app ⇒ java ⇒ внутренний package. Увидим MainActivity, который нам автоматически создал Android Studio. В сочетании клавиш Shift + F6 переименуем его в AuthActivity. Нажмем Refactor и откроем его.

```
package com.elegion.myfirstapplication;
import android.os.Bundle;
import android.support.annotation.Nullable;
import android.support.v7.app.AppCompatActivity;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;

public class AuthActivity extends AppCompatActivity {

    @Override
    protected void onCreate(@Nullable Bundle
        savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.ac_auth);
    }
}
```

Здесь, как мы видим, у нас есть метод onCreate(). И, соответственно, используется метод setContentView(). Мы хотим, чтобы наши view присоединялись к нашим объектам в коде. Давайте откроем наш layout. Как мы видим, у нас есть здесь два id-текста и два button. Соответственно, создадим эти же поля внутри нашей Activity. После этого мы сможем обращаться к view по ссылкам. Сразу добавим слушателей: кнопки mEnter и mRegistration. Эти слушатели будут вызываться при нажатии на кнопки.

```
public class AuthActivity extends AppCompatActivity {
    //создаём каждый объект
    private EditText mLogin;
    private EditText mPassword;
    private Button mEnter;
    private Button mRegister;
    //слушатель для кнопки входа
    private View.OnClickListener mOnEnterClickListener
        = new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            //todo Обработка нажатия по кнопке
        }
    };
    //слушатель для кнопки регистрации
    private View.OnClickListener mOnRegisterClickListener
        = new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            //todo Обработка нажатия по кнопке
        }
    };
    protected void onCreate(@Nullable Bundle
        savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.ac_auth);
        //Делаем так, чтобы можно было обращаться к view по ссылкам
        mLogin = findViewById(R.id.etLogin);
        mPassword = findViewById(R.id.etPassword);
        mEnter = findViewById(R.id.buttonEnter);
        mRegister = findViewById(R.id.buttonRegister);

        mEnter.setOnClickListener(mOnEnterClickListener);
        mRegister.setOnClickListener(mOnRegisterClickListener);
    }
}
```

Теперь запускаем наш эмулятор (кнопочка run):

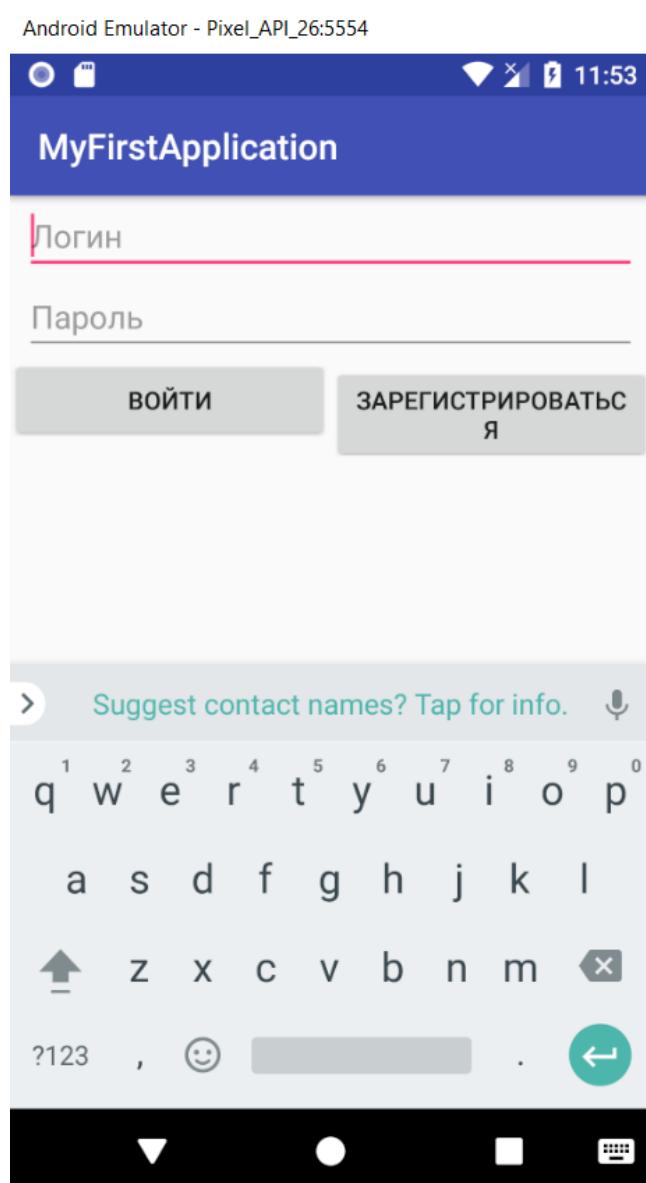


Рис. 3.1: Окно логина на эмуляторе

### 3.1.3 КП. Валидация email и password. Ошибки в Toast

Добавим логику обработки данных логина, который по факту будет email'ом, пароля и нажатия на кнопку «Войти». Откроем AuthActivity, перейдём в метод onClick в mOnEnterClickListener, вместо «todo» добавим логику нажатия по кнопке. Для начала проверим email и добавим метод:

```
private boolean isValidEmail() {  
    return !TextUtils.isEmpty(mLogin.getText())  
        && Patterns.EMAIL_ADDRESS.matcher(mLogin.getText())  
            .matches();  
} //проверка на непустоту и на правильность паттерна емейла
```

Перепишем немного OnClickListener:

```
private View.OnClickListener mOnEnterClickListener
    = new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        if (isValidEmail()) {
            //если email подходит, можем перейти в приложение
        }
    }
};
```

Аналогично создаём isPasswordValid():

```
private boolean isPasswordValid() {
    return !TextUtils.isEmpty(mPassword.getText());
} //берём текст из поля mPassword на совпадение с базой
```

Так же не забудем изменить соответствующего слушателя:

```
private View.OnClickListener mOnEnterClickListener
    = new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        if (isValidEmail() && isPasswordValid()) {
            // переход в приложение
        } else {
            showMessage(R.string.login_input_error);
        }
    }
};
```

Теперь мы добавили ошибку. Создадим тост, который будет вылетать с ошибкой при регистрации:

```
private void showMessage(@StringRes int string) {
    Toast.makeText(this, string, Toast.LENGTH_LONG).show();
} //показываем сообщение об ошибке с помощью тоста
```

Не забываем передавать контекст - this. LENGTH\_LONG потому что хотим показывать сообщение долго. Создадим строковый ресурс:

```
<string name="login_input_error">Error !!</string>
```

Теперь если мы запустим приложение и ничего не введём хотя бы одно из полей (или не валидный email), вылетит тост с ошибкой. Если же всё хорошо, то пока ничего не произойдёт.

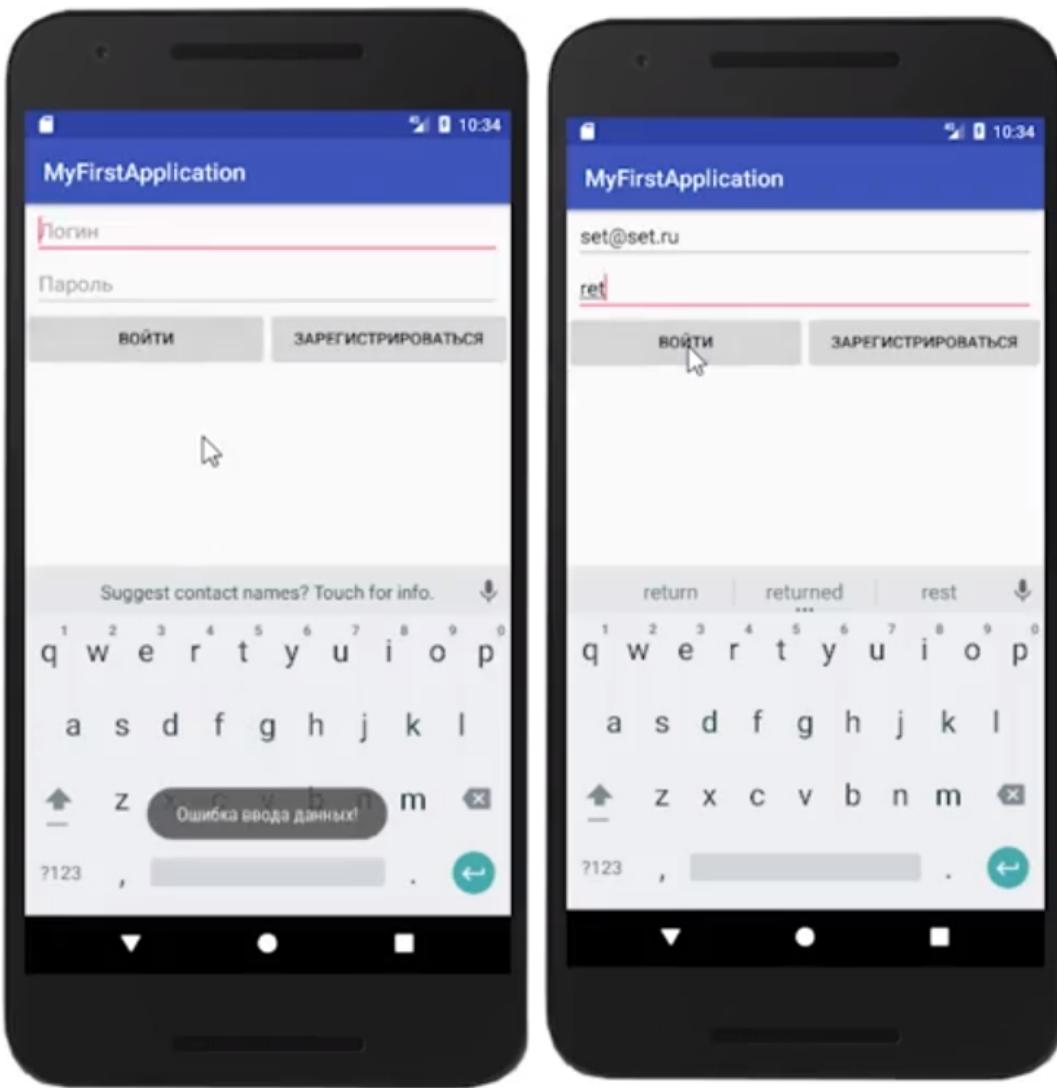


Рис. 3.2: Ввели неправильные данные Рис. 3.3: Ввели правильные данные

Но пароль отображает буквы! Для того, чтобы вместо букв были звёздочки, перейдём к ac\_auth.xml и введём в соответствующий вводу пароля EditText строчку:

```
android:inputType="textPassword"/>
```

Теперь сразу появляются точки и пароль не отображается.

## 3.2 Activity профиля

### 3.2.1 КП. Верстка экрана профиля

Создадим макет экрана профиля, в который можно попасть после успешной авторизации. Заходим в на проект ⇒ res ⇒ layout. Правой кнопкой по layout ⇒ new ⇒ Layout resource file. И назовём его ac\_profile. Enter. Переходим во вкладку text

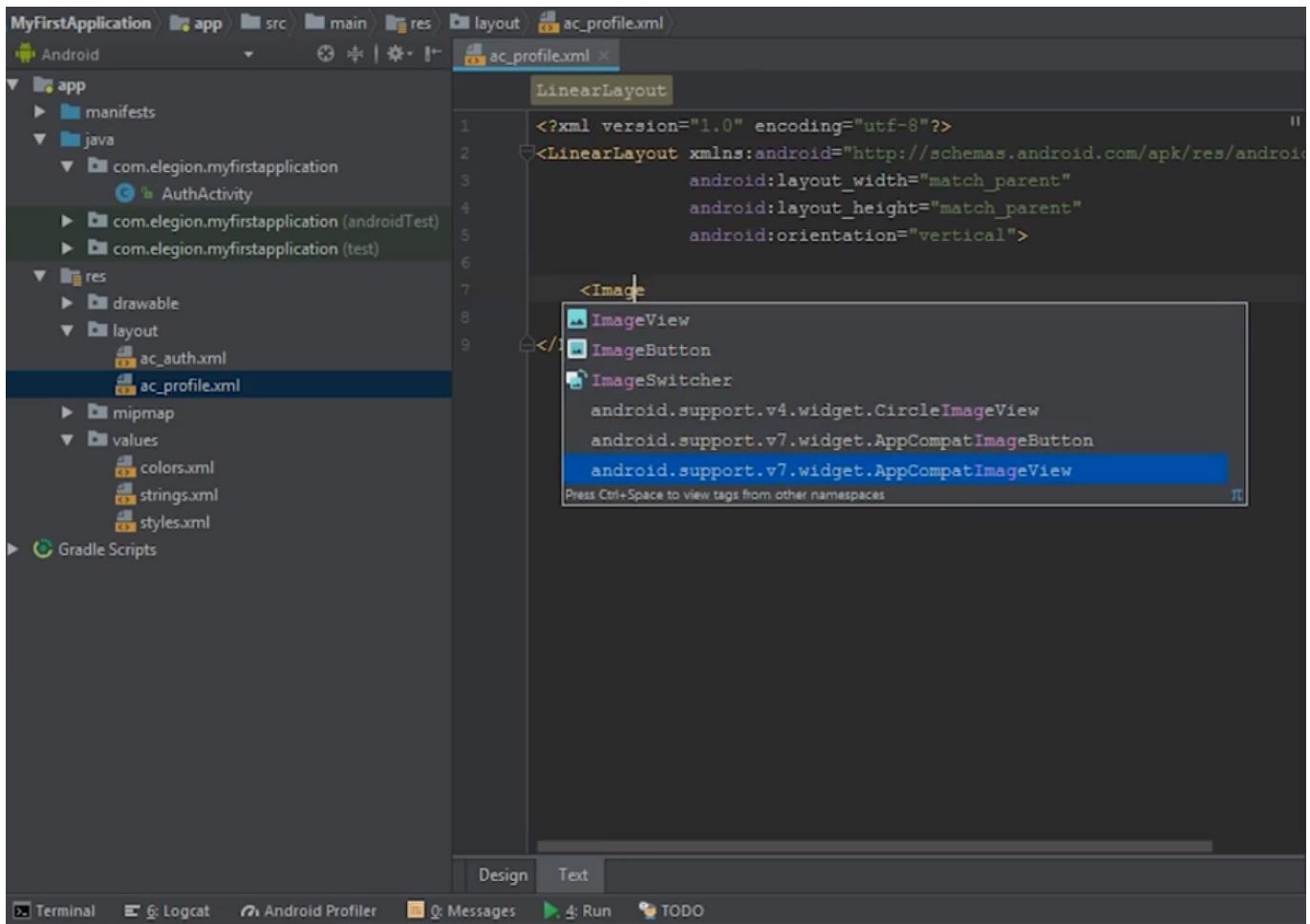


Рис. 3.4: Выбор AppCompatImageView

И допишем туда параметры layout\_width, layout\_height и отступы со всех сторон (layout\_margin). Ориентация будет горизонтальной.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/
        android"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="horizontal">
    <android.support.v7.widget.AppCompatImageView
        android:id="@+id/ivPhoto"
        android:layout_width="100dp"
        android:layout_height="100dp"
        android:layout_margin="16dp"/>
```

Далее допишем LinearLayout и в нём два TextView с полями - отображаемые email и пароль. И зададим первому текст. А второму будем передавать значение

самого емейла, для этого дадим ему id. Самому же LinearLayout зададим два отступа - от верха (layout\_marginTop) и от правого края (layout\_marginRight)

```
<LinearLayout android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginRight="16dp"
    android:layout_marginTop="16dp"
    android:orientation="vertical">
    <TextView android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/email"/>
    <TextView android:id="@+id/tvEmail"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"/>
    <TextView android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/password"/>
    <TextView android:id="@+id/tvPassword"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"/>
</LinearLayout>
</LinearLayout>
```

Не забываем, что каждая строка должна быть отдельным строковым ресурсом. Тогда наш res ⇒ strings.xml будет выглядеть так:

Листинг 3.1: strings.xml

```
<resources>
    <string name="app_name">MyFirstApplication</string>
    <string name="login_hint">Login</string>
    <string name="login_password">Password</string>
    <string name="login_enter">Sign in</string>
    <string name="login_register">Sign up</string>
</resources>
```

### 3.2.2 КП. Создание активити профиля

Настроим запуск экрана профиля через нажатие кнопки "Log in". Пока что наш ProfileActivity.java выглядит так:

Листинг 3.2: ProfileActivity.java

```
public class ProfileActivity extends AppCompatActivity {
    private AppCompatImageView mPhoto; // именно AppCompat
    private TextView mLogin;
    private TextView mPassword;
    private View.OnClickListener mOnPhotoClickListener
        = new View.OnClickListener() {
        @Override
        public void onClick(View view) {
        }
    };
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.ac_profile);
        // далее инициализируем все поля (находим их по id)
        mPhoto = findViewById(R.id.ivPhoto);
        mLogin = findViewById(R.id.tvEmail);
        mPassword = findViewById(R.id.tvPassword);
        mPhoto.setOnClickListener(mOnPhotoClickListener);
    }
}
```

Мы забыли добавить id для Photo и поэтому R.id.ivPhoto подсвечивается красным. Для этого в ac\_profile.xml перед android:layout\_width надо добавим:

```
android:id="@+id/ivPhoto"
```

Чтобы не было проблем с совместимостью надо писать AppCompatImageView. Наш UI проинициализирован.

Теперь давайте попробуем передать некоторые параметры и запустить наш ProfileActivity.

Для этого допишем в начало ProfileActivity.java строки:

```
public static final String EMAIL_KEY = "EMAIL_KEY";
public static final String PASSWORD_KEY = "PASSWORD_KEY";
```

Для этого перейдём в AuthActivity. По нажатию на кнопку входа при правильно введённом email и пароле мы будем его запускать; для этого мы будем использовать интенты. Для этого в View.OnClickListener допишем:

```

if (isValidEmail() && isValidPassword()) {
    Intent startProfileIntent = //класс, к которому переходим
        new Intent(AuthActivity.this, ProfileActivity.
            class);
    startProfileIntent.putExtra(ProfileActivity.EMAIL_KEY,
        mLogin.getText().toString());
    startProfileIntent.putExtra(ProfileActivity.
        PASSWORD_KEY,
        mPassword.getText().toString());
    startActivity(startProfileIntent);
} else {
    showMessage(R.string.login_input_error);
}

```

Получить переданные данные, можно методом getIntent() в ProfileActivity. И создадим бандл, в который передадим extra:

```

Bundle bundle = getIntent().getExtras();
mLogin.setText(bundle.getString(EMAIL_KEY));
mPassword.setText(bundle.getString(PASSWORD_KEY));
//устанавливаем значения логина и пароля из бандла

```

Важный момент, чтобы ProfileActivity работала, её обязательно надо добавить в манифест, который будет выглядеть как-то так.

Листинг 3.3: AndroidManifest.xml

```

...
<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
    <activity android:name=".AuthActivity">
        <intent-filter>
            <action android:name="android.intent.action.MAIN"/>
            <category android:name="android.intent.category.LAUNCHER"/>
        </intent-filter>
    </activity>
    <activity android:name=".ProfileActivity">
    </activity>
</application>
</manifest>

```

Запускаем приложение и пробуем ввести что-нибудь. После успешного ввода переходим на экран профиля.

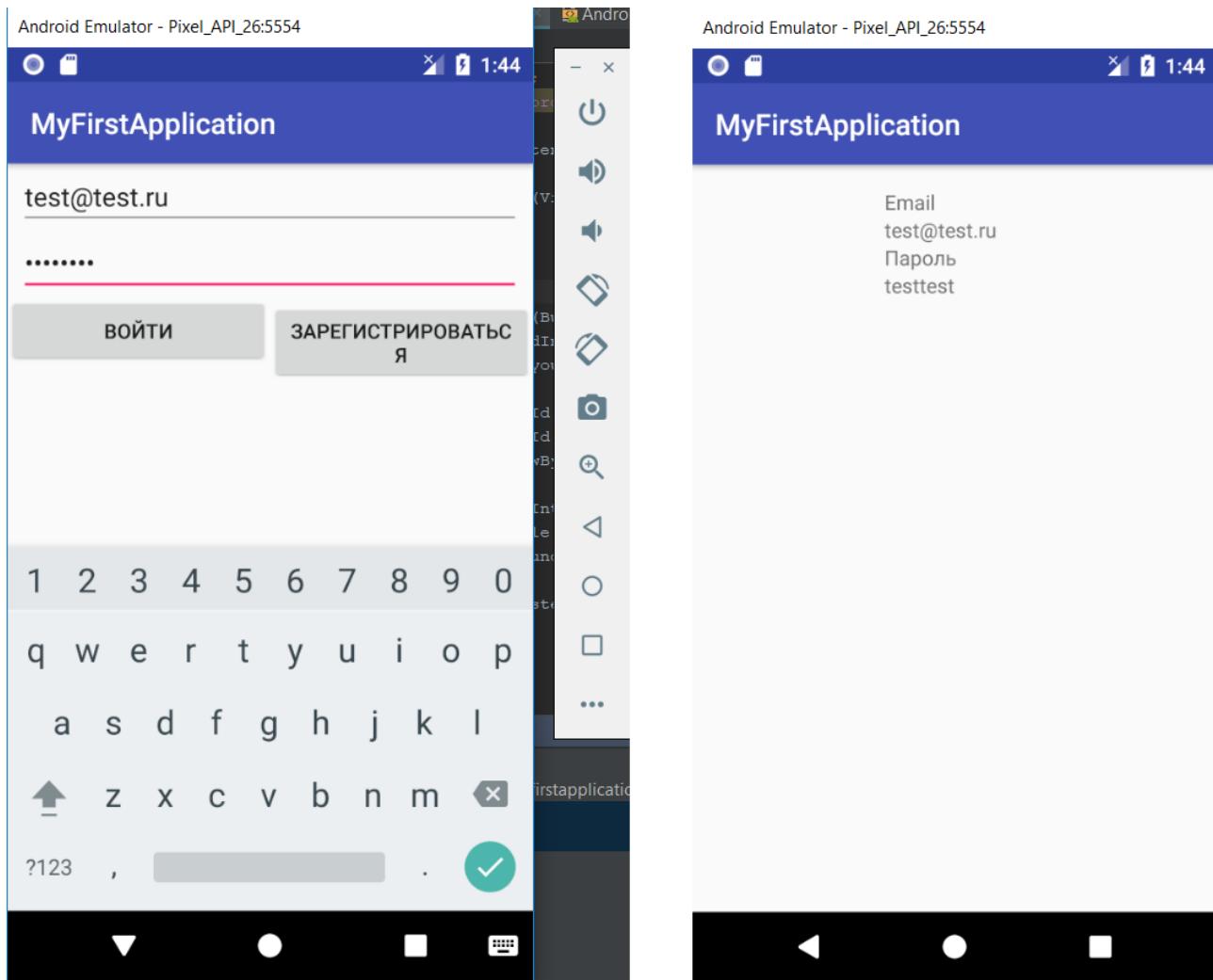


Рис. 3.5: Logging in

[Архив с проектом](#), в котором работают log in и ProfileActivity

### 3.2.3 КП. Создание класса User

Теперь давайте заменим эти параметры и добавим класс User в наш проект. Откроем Android Studio ⇒ package app ⇒ java и создадим New Java Class. Назовем его User. В данный класс User добавим два поля — private String email, пусть будет mLogin, и private String mPassword. Сочетанием клавиш Alt + Insert добавим Getter и Setter для логина и пароля. И добавим Constructor, который принимает в себя логин и пароль. В итоге файл класса User будет выглядеть так:

Листинг 3.4: User.java

```
package com.elegion.myfirstapplication;
public class User {
    private String mLogin;
    private String password;
    public User(String mLogin, String password) {
        this.mLogin = mLogin;
        this.password = password;
    } //конструктор, принимающий в себя логин и пароль

    public String getLogin() {
        return mLogin;
    } //Getter логина

    public String getPassword() {
        return password;
    } //Getter пароля

    public void setLogin(String mLogin) {
        this.mLogin = mLogin;
    } //Setter логина

    public void setPassword(String password) {
        this.password = password;
    } //Setter пароля
}
```

Далее, перейдем в AuthActivity и вместо того, чтобы передавать сюда email и пароль по отдельности, будем передавать User. Для этого перейдем в ProfileActivity, удалим PASSWORD\_KEY, а EMAIL\_KEY переделаем под USER\_KEY.

```
// в profileActivity.java
public class ProfileActivity extends AppCompatActivity {
    public static final String USER_KEY = "USER_KEY"; . . .
```

Do Refactor (заменяем везде, где встречалось). Перейдем в AuthActivity. Удалим строку с password. В putExtra добавим new User. И в качестве передаваемых параметров передадим в него логин, который мы возьмем из mLogin.getText().toString(), и mPassword.getText().toString(). [БЕЗ\_ЗВУКА] Android Studio ругается и не может понять, что мы хотим передать. Все правильно. Ведь мы забыли указать то, что класс User является Serializable. Давайте исправим это.

```
public class User implements Serializable
```

И вот Android Studio перестала ругаться. С учётом изменений, AuthActivity будет выглядеть так

Листинг 3.5: AuthActivity

```
private View.OnClickListener mOnEnterClickListener =
    new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            if (isValidEmail() && isValidPassword()) {
                Intent startProfileIntent =
                    new Intent(
                        AuthActivity.this, ProfileActivity.class);
                startProfileIntent.putExtra(ProfileActivity.
                    USER_KEY,
                    new User(mLogin.getText().toString(),
                        mPassword.getText().toString()));
                startActivity(startProfileIntent);
            } else {
                showMessage(R.string.login_input_error);
            }
        }
   };
```

Давайте посмотрим, как это работает, вернее, исправим это в ProfileActivity. Ведь мы получаем bundle, в котором мы должны получить не строку, а уже пользователя. Соответственно, создадим его и передаем ему наш ключ

```
// в методе onCreate() после строк с mPassword
Bundle bundle = getIntent().getExtras();
User user = (User) bundle.get(USER_KEY);
mLogin.setText(user.getLogin());
mPassword.setText(user.getPassword());
```

И с помощью приведения типов превращаем его в User. mLogin поменяем на mPassword, а внутреннее содержимое поменяем на user.getLogin, и user.getPassword. Готово. Всё работает так же, как и в предыдущем случае (рис. 3.5)

### 3.3 Добавление фрагментов

#### 3.3.1 КП. Создание хост активити для фрагментов

В последующих уроках мы с вами будем использовать фрагменты. Но для начала напишем базовый класс для того, чтобы в нашей Activity был всего один фрагмент. Откроем проект. app ⇒ java ⇒ New ⇒ Java Class. Назовем нашу активность SingleFragmentActivity. Ok.

```
public abstract class SingleFragmentActivity
    extends AppCompatActivity { //наследовали
    @Override
    protected void onCreate(@Nullable Bundle
        savedInstanceState){ //переопределили Create
        super.onCreate(savedInstanceState);
        if (savedInstanceState != null) { //добавленная логика
            FragmentManager fragmentManager = //обязательно support
                getSupportFragmentManager();
            //начнём транзацию по запуску фрагмента
            fragmentManager.beginTransaction()
                .replace(R.id.fragmentContainer, getFragment())
                .commit();
        }
    }
}
```

В этом случае нам нужен контейнер ViewId. Давайте его добавим. Идем res ⇒ layout ⇒ New ⇒ Layout resource file. и назовём его ac\_single\_fragment. Ok. Переходим во вкладочку «Текст» и сюда вместо LinearLayout добавляем самый простой FrameLayout и добавим ему id. Назовем его fragmentContainer.

Листинг 3.6: ac\_single\_fragment

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com
    /apk/res/android"
    android:id="@+id/fragmentContainer"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"/>
```

Перейдем в SingleFragmentActivity. После super.onCreate() добавим setContentView(R.layout.ac\_single\_fragment). Наш Android Studio опять-таки не видит вновь добавленный ресурс, поэтому синхронизируется. Ресурсы успешно добавились.

Теперь попробуем добавить наш фрагмент. FragmentManager.beginTransaction().replace (R.id. fragmentContainer) — это то, куда мы будем добавлять наш фрагмент.

мент. Название фрагмента. Так как это базовый класс, у нас нет конкретного фрагмента, поэтому создадим абстрактный метод.

#### Листинг 3.7: singleFragmentActivity

```
public class SingleFragmentActivity
    extends AppCompatActivity { //наследовали
    @Override
    protected void onCreate(@Nullable Bundle
        savedInstanceState){ //переопределили Create
        super.onCreate(savedInstanceState); //добавим
        setContentView(R.layout.ac_single_fragment);
        if (savedInstanceState != null) { //добавленная логика
            FragmentManager fragmentManager = //обязательно support
            getSupportFragmentManager();
            //начнём транзацию по запуску фрагмента
            fragmentManager.beginTransaction()
                .replace(R.id.fragmentContainer, getFragment())
                .commit(); //после реплейса вызвали commit
        }
    } //абстрактный метод получения фрагмента
    protected abstract Fragment getFragment();
}
```

Наш Android Studio ругается, что мы не назвали наш класс абстрактным. Исправим это:

```
public abstract class SingleFragmentActivity ...
```

Соответственно, в метод replace передадим наш getFragment. И по желанию мы можем передать тег. Этого мы делать не будем. Вот и все. Базовый класс для одного фрагмента готов.

### 3.3.2 КП. Миграция логики AuthActivity во фрагмент

На прошлых занятиях мы с вами написали Activity для одного фрагмента. Давайте теперь изменим AuthActivity на AuthFragment.

Зайдем в SingleFragmentActivity в условие if savedInstanceState не равно null. Это неправильно. Наша транзакция должна выполняться, когда savedInstanceState равен null.

После всех изменений наш SingleFragmentActivity должен будет выглядеть так:

## Листинг 3.8: SingleFragmentActivity.java

```

public abstract class SingleFragmentActivity extends
    AppCompatActivity {
    @Override
    protected void onCreate(@Nullable Bundle
        savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.ac_single_fragment);
        if (savedInstanceState == null) { //изменили на равенство
            FragmentManager fragmentManager =
                getSupportFragmentManager();
            fragmentManager.beginTransaction()
                .replace(R.id.fragmentContainer, getFragment())
                .commit();
        }
    }
    protected abstract Fragment getFragment();
}

```

Теперь перейдем в AuthActivity. Нажимаем на него и с помощью сочетаний Shift + F6 меняем на AuthFragment и экстендишем его от Fragment (с комментарием android.support.v4.app).

```
public class AuthFragment extends Fragment {
```

Android Studio подсвечивает некоторые блоки красным. Давайте посмотрим, почему, и попытаемся это исправить.

Заходим в java ⇒ com.elegion ⇒ AuthFragment.java и видим, что AuthFragment.this не является контекстом, а Activity является. Соответственно, заменим его на метод getActivity(). Теперь все работает. Скопируем его, он нам в будущем пригодится. То же самое с showMessage.

```

//было:
new Intent(AuthFragment.this, ProfileActivity.class);
//заменили на
new Intent(getActivity(), ProfileActivity.class);
//аналогично чуть дальше в showMessage было:
Toast.makeText(this, string, Toast.LENGTH_LONG).show();
//стало
Toast.makeText(getActivity(), string, Toast.LENGTH_LONG).
    show();

```

В том же AuthActivity, видим что метод onCreate() отличается от того метода, который есть в Activity. Вместо этого в фрагменте используется метод onCreateView(). Давайте заменим метод onCreate() на onCreateView() (когда на-

чиаем вводить в контекстном меню выбираем нужный нам onCreateView(). Копируем все, что у нас было в методе onCreate, и удаляем его. setContentView() - метод нам не пригодится. Вместо этого нам пригодится такая строка "View v = inflater.inflate (R.layout.ac\_auth)". Сразу же ac\_auth переименуем в fr\_auth. Далее, передаем container и в качестве Boolean parameter передаем false. Далее, у нас нет метода findViewById() внутри Activity. Но зато он есть у нашей View, которую мы заинфлейтили. Соответственно, делаем везде v.findViewById(). После всех преобразований получим:

```
//вместо onCreate() получили onCreateView() после ShowMessage
@NoArgsConstructor
@Override
    public View onCreateView(LayoutInflater inflater,
        @Nullable ViewGroup container,
        @Nullable Bundle savedInstanceState) {
    View v = inflater.inflate(R.layout.fr_auth,
        container, false);
    //везде поменяли на v.findViewById
    mLogin = v.findViewById(R.id.etLogin);
    mPassword = v.findViewById(R.id.etPassword);
    mEnter = v.findViewById(R.id.buttonEnter);
    mRegister = v.findViewById(R.id.buttonRegister);
    mEnter.setOnClickListener(mOnEnterClickListener);
    mRegister.setOnClickListener(
        mOnRegisterClickListener);
    return v;
}
}
```

Все вроде бы должно работать, за исключением одного но. Когда мы переименовывали наш AuthFragment в AuthActivity, он переименовался также в манифесте. Давайте зайдем туда и исправим это.

```
<activity android:name=".AuthActivity">
```

AuthFragment становится AuthActivity, которой у нас нет. Знаете, почему? Потому что её нужно создать. Заходим в наш package, создаем Java Class, называем его AuthActivity.

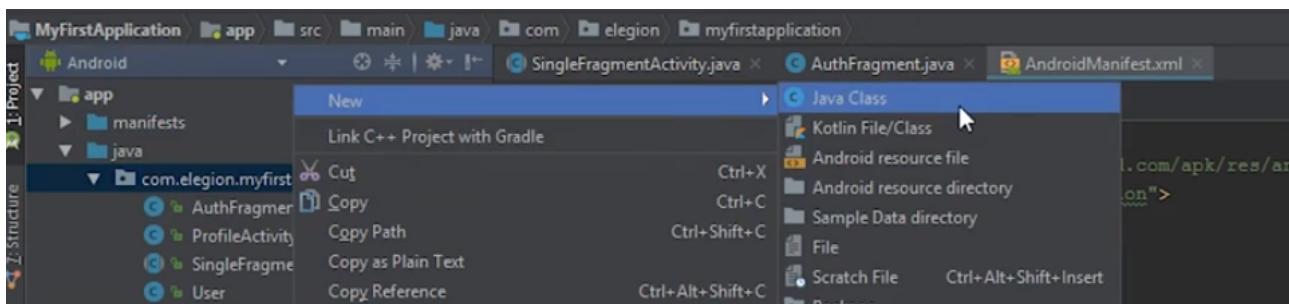


Рис. 3.6: Создание нового java-класса  
extends SingleFragmentActivity, который мы написали до этого.

```
public class AuthActivity extends SingleFragmentActivity{
    @Override
    protected Fragment getFragment() {
        return null; //поменяли
    }
}
```

И, соответственно, он просит нас реализовать один метод. Давайте сделаем это. В качестве возвращаемого параметра в GetFragment() мы должны передать instance фрагмента, который должен показываться на экране. Перед этим нужно создать метод, который будет создавать этот instance. Соответственно, нужно перейти в AuthFragment и создать метод newInstance() (сам сгенерируется когда начнём вводить newInstance), его мы поменяем потом.

```
public static AuthFragment newInstance() {
    Bundle args = new Bundle();
    AuthFragment fragment = new AuthFragment();
    fragment.setArguments(args);
    return fragment;
}
```

Вернемся в AuthActivity и передадим ему AuthFragment.newInstance.

```
protected Fragment getFragment() {
    return AuthFragment.newInstance(); //поменяли
}
```

Вот и все. Но в эмуляторе пустой экран. А все потому, что в AuthFragment, в onCreateView в качестве возвращаемого параметра мы забыли вернуть нашу View, в которую мы успешно заинфлейтили.

Листинг 3.9: конец AuthFragment

```
return super.onCreateView(inflater, container,
                           savedInstanceState); //было
return v; //стало
```

Запускаем и всё работает.

Можем даже попробовать ввести e-mail и перейти на следующий экран, чтобы убедиться, что мы ничего не сломали. Нажмем кнопочку «войти». Все прекрасно работает как и в [3.5](#).

Мы успешно использовали `SingleFragmentActivity` class, который обычно пригождается, когда нам нужно использовать один фрагмент в качестве основного.

### 3.3.3 КП. Добавление фрагмента регистрации.

#### Создание класса `PreferenceHelper`

В предыдущих занятиях мы с вами создали экран авторизации и экран показа профиля. Но откуда же возьмутся пользователи? Для этого нам нужно создать экран регистрации.

[Мы предварительно подготовили для вас весь UI и его инициализацию.](#)

Нам осталось только добавить логику. Но перед этим давайте посмотрим, что у нас уже есть. Откроем проект, перейдём в `RegistrationFragment`, увидим четыре поля: `mLogin`, `mPassword`, `PasswordAgain` и `Registration`. Первые три — `EditText` и кнопка `Registration`. Посмотрим, как он выглядит в XML

```
//начало RegistrationFragment.java
public class RegistrationFragment extends Fragment {
    private EditText mLogin; //email
    private EditText mPassword;//введите пароль
    private EditText mPasswordAgain;//введите пароль ещё раз
    private Button mRegistration;//зарегистрироваться

    public static RegistrationFragment newInstance() {
        return new RegistrationFragment();
    }
    private View.OnClickListener
        mOnRegistrationClickListener
        = new View.OnClickListener() {
        @Override//здесь будет вся логика регистрации
        public void onClick(View view) {
        }
    };
}
```

Ниже представлена xml разметка:

```
<LinearLayout  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:orientation="vertical">  
  
<EditText  
    android:id="@+id/etLogin"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:layout_marginLeft="16dp"  
    android:layout_marginRight="16dp"  
    android:hint="@string/email"  
    android:inputType="textEmailAddress"  
    android:textSize="16sp"/>  
<EditText  
    android:id="@+id/etPassword"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:layout_marginLeft="16dp"  
    android:layout_marginRight="16dp"  
    android:hint="@string/password_enter"  
    android:inputType="textPassword"  
    android:textSize="16sp"/>  
<EditText  
    android:id="@+id/tvPasswordAgain"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:layout_marginLeft="16dp"  
    android:layout_marginRight="16dp"  
    android:hint="@string/  
        password_enter_again"  
    android:inputType="textPassword"  
    android:textSize="16sp"/>  
  
<Button  
    android:id="@+id/btnRegistration"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:layout_margin="16dp"  
    android:text="@string/register"/>  
</LinearLayout>
```

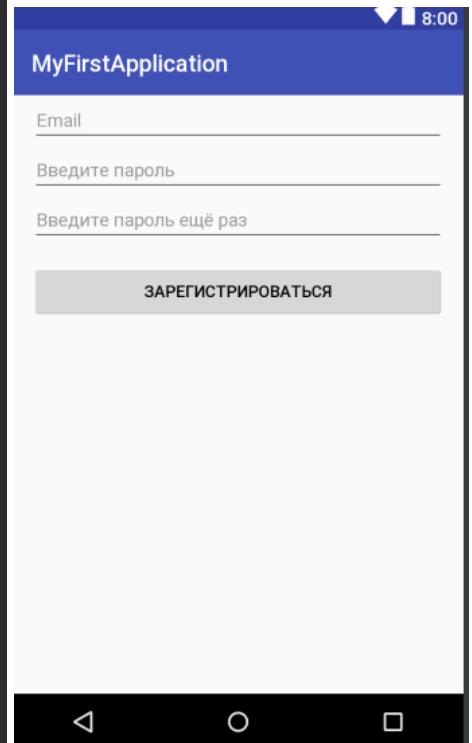


Рис. 3.7: Окно регистрации

Смотрим RegistrationFragment.java дальше:

```
//продолжение RegistrationFragment.java
@NoArgsConstructor
@Override //обычная инициализация View
public View onCreateView(LayoutInflater inflater,
    @Nullable ViewGroup container,
    @Nullable Bundle savedInstanceState) {
    View view = inflater.inflate(
        R.layout.fr_registration, container, false);
    mSharedPreferencesHelper = new
        SharedPreferencesHelper(getActivity());
    //обычная инициализация View
    mLogin = view.findViewById(R.id.etLogin);
    mPassword = view.findViewById(R.id.etPassword);
    mPasswordAgain = view.findViewById(R.id.
        tvPasswordAgain);
    mRegistration = view.findViewById(R.id.
        btnRegistration);
    mRegistration.setOnClickListener
        (mOnRegistrationClickListener);
    return view;
}
```

Далее идёт логика для проверки ввода данных (как в авторизации)

```
private boolean isValid() { //проверка мейла и пароля
    String email = mLogin.getText().toString();
    if (isValidEmail(email) && isPasswordsValid()) {
        return true;
    }
    return false;
}
private boolean isValidEmail(String email) { //емейл
    return !TextUtils.isEmpty(email) && Patterns.
        EMAIL_ADDRESS.matcher(email).matches();
}
private boolean isPasswordsValid() { //правильность пароля
    String password = mPassword.getText().toString();
    String passwordAgain = mPasswordAgain.getText().
        toString();
    return password.equals(passwordAgain)
        && TextUtils.isEmpty(password)
        && TextUtils.isEmpty(passwordAgain);
}
```

Метод showMessage() нужен для показа сообщений; пока что он у нас не используется.

```
private void showMessage(@StringRes int string) {  
    //todo показ сообщений  
    Toast.makeText(getActivity(), string, Toast.  
        LENGTH_LONG).show();  
}  
}//конец RegistrationFragment.java
```

Вся логика должна происходить при нажатии на кнопку Registration; для этого у нас есть mOnRegistrationClickListener. Всех пользователей, с которыми мы будем работать внутри приложения, мы будем хранить в SharedPreferences. Для этого создадим класс SharedPreferencesHelper.

```
// в том же RegistrationFragment.java переписываем OnClickListener  
private View.OnClickListener  
    mOnRegistrationClickListener  
        = new View.OnClickListener() {  
    @Override  
    public void onClick(View view) {  
        if (isValid()) {  
            boolean isAdded = mSharedPreferencesHelper.  
                addUser(new User(  
                    mLogin.getText().toString(),  
                    mPassword.getText().toString()  
                ));  
  
            if (isAdded) {  
                showMessage(R.string.login_register_success);  
            } else {  
                showMessage(R.string.login_register_error);  
            }  
        } else {  
            showMessage(R.string.input_error);  
        }  
    }  
};
```

Перейдём в Package ⇒ New ⇒ Java Class ⇒ SharedPreferencesHelper. OK.

```
public class SharedPreferencesHelper {
    public static final String SHARED_PREF_NAME = "SHARED_PREF_NAME";
    private SharedPreferences mSharedPreferences; //создали
    //инициализируем его в конструкторе, передим туда контекст
    public SharedPreferencesHelper(Context context) {
        //и получим его из контекста. 1 аргумент - название файла.
        //2 аргумент - режим работы самого контекста. Делаем приватным
        mSharedPreferences = context.getSharedPreferences(
            SHARED_PREF_NAME, Context.MODE_PRIVATE);
    }
}
```

Нам нужно описать саму логику сохранения пользователей в SharedPreferences. Для этого нам нужно создать два метода — getUsers и addUser. Для того чтобы хранить нам список пользователей внутри SharedPreferences, необходимо использовать формат данных JSON, потому что сам SharedPreferences не позволяет сохранять ни Serializable, ни Parcelable объекты, но позволяет сохранять строки, поэтому мы наш список пользователей будем форматировать в JSON и сохранять как строку. Далее пишем в SharedPreferencesHelper.java

```
public static final String USERS_KEY = "USERS_KEY";
//ключ, который будем передавать в mSharedPreferences.getString
//далее начинаем вводить и выбираем Type (Reflect).
//в TypeToken в качестве генерикового параметра передаём лист юзеров
public static final Type USERS_TYPE =
    new TypeToken<List<User>>() {}.getType();
//объявим Gson-овский объект
private Gson mGson = new Gson();
//чтобы получить список пользователей, из JSON делаем список
public List<User> getUsers() {
    List<User> users = mGson.fromJson(mSharedPreferences
        .getString(USERS_KEY, ""), USERS_TYPE);
    //в mSharedPreferences.getString создаём лист юзеров
    //2ой аргумент (USERS_TYPE) это тип (объявили в TypeToken)
    return users == null ? new ArrayList<User>() : users;
    //если пустой - выводим null. Иначе выводим список юзеров
}
```

Метод getUsers() готов. Теперь добавим метод addUser(). boolean нам нужен для того, чтобы определить, был добавлен пользователь до этого или нет. Для того чтобы добавить пользователя в SharedPreferences, сначала нужно получить всех пользователей оттуда.

```

public boolean addUser(User user) {
    List<User> users = getUsers();
    //пробегаемся в цикле и проверяем пользователя, которого мы создаём
    for (User u : users) {
        if (u.getLogin().equalsIgnoreCase(user.getLogin()))
            {
                return false; //если существует – просто не добавляем
            }
    } //если юзера нет, сохраняем его в mSharedPreferences
    users.add(user);
    mSharedPreferences.edit().putString(USER_TYPE_KEY,
        mGson.toJson(users, USER_TYPE)).apply();
    //параметром 2 передаём USER_TYPE, чтобы он перевёл нам правильно
    return true;
}
}

```

Метод addUser() и основная логика сохранения пользователей в SharedPreferences готовы. Давайте добавим её внутрь RegistrationFragment.

Переходим в RegistrationFragment, в mOnRegistrationClickListener. Допишем:

Листинг 3.10: RegistrationFragment.java

```

public class RegistrationFragment extends Fragment { //...
    private SharedPreferenceHelper
        mSharedPreferenceHelper;
    private View.OnClickListener
        mOnRegistrationClickListener
        = new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        if (isValid()) { //если всё верно добавим юзера
            boolean isAdded = mSharedPreferenceHelper.
                addUser(new User(
                    mLogin.getText().toString(),
                    mPassword.getText().toString()));
            if (isAdded) { //если добавился, покажем "Успех"
                showMessage(R.string.login_register_success);
            } else { //если не добавился, покажем сообщение "Ошибка"
                showMessage(R.string.login_register_error);
            }
        } else {
            showMessage(R.string.input_error);
        }
    }
}

```

Не забудем изменить метод onCreate()

```
public View onCreateView(LayoutInflater inflater,
    @Nullable ViewGroup container, @Nullable Bundle
    savedInstanceState) {
    View view = inflater.inflate(R.layout.fr_registration
        ,
        container, false);
    //принициализировали и передаём контекст
    mSharedPreferencesHelper = new
        SharedPreferencesHelper(getActivity());
    //...
}
```

Вроде готово. Давайте посмотрим: всё запускается, нажимаем на кнопку регистрации, вводим все данные, но ничего не происходит. Давайте дебажить (кнопка Attach debugger to Android во время работы нашего приложения на эмуляторе):

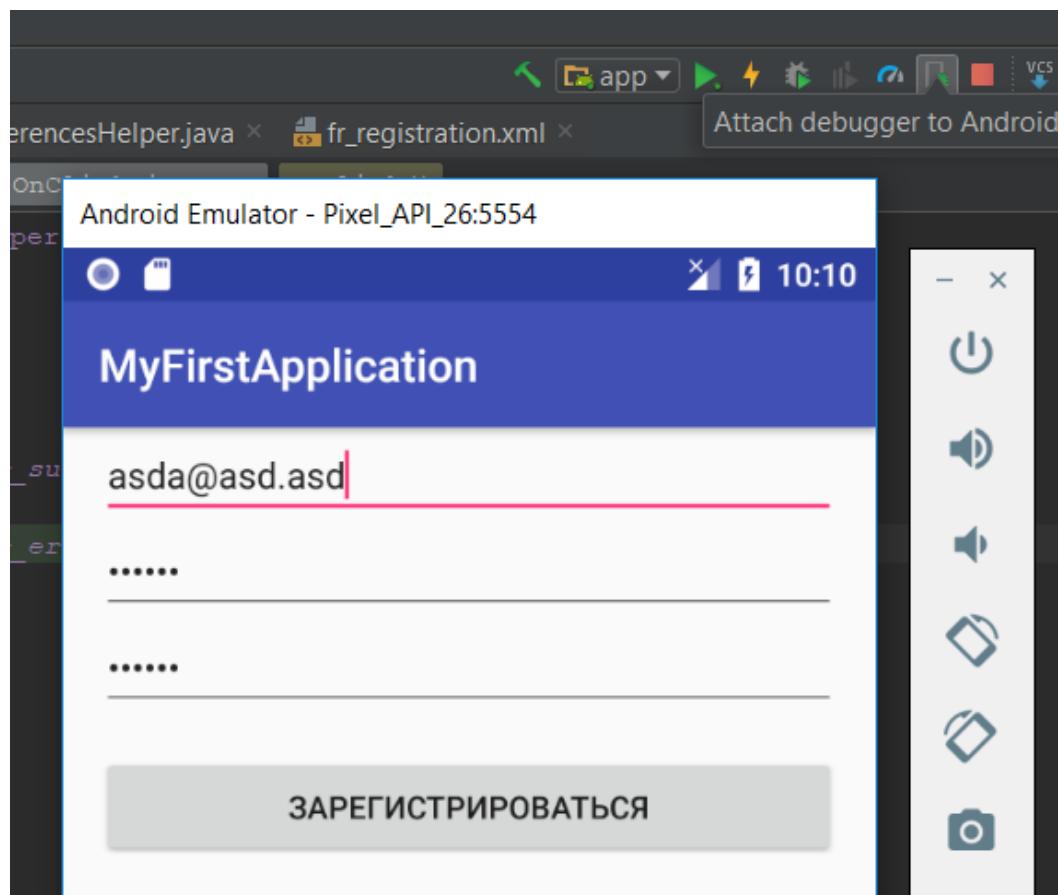


Рис. 3.8: Пробуем зарегистрироваться  
Посмотрим на поведение метода IsInputValid().

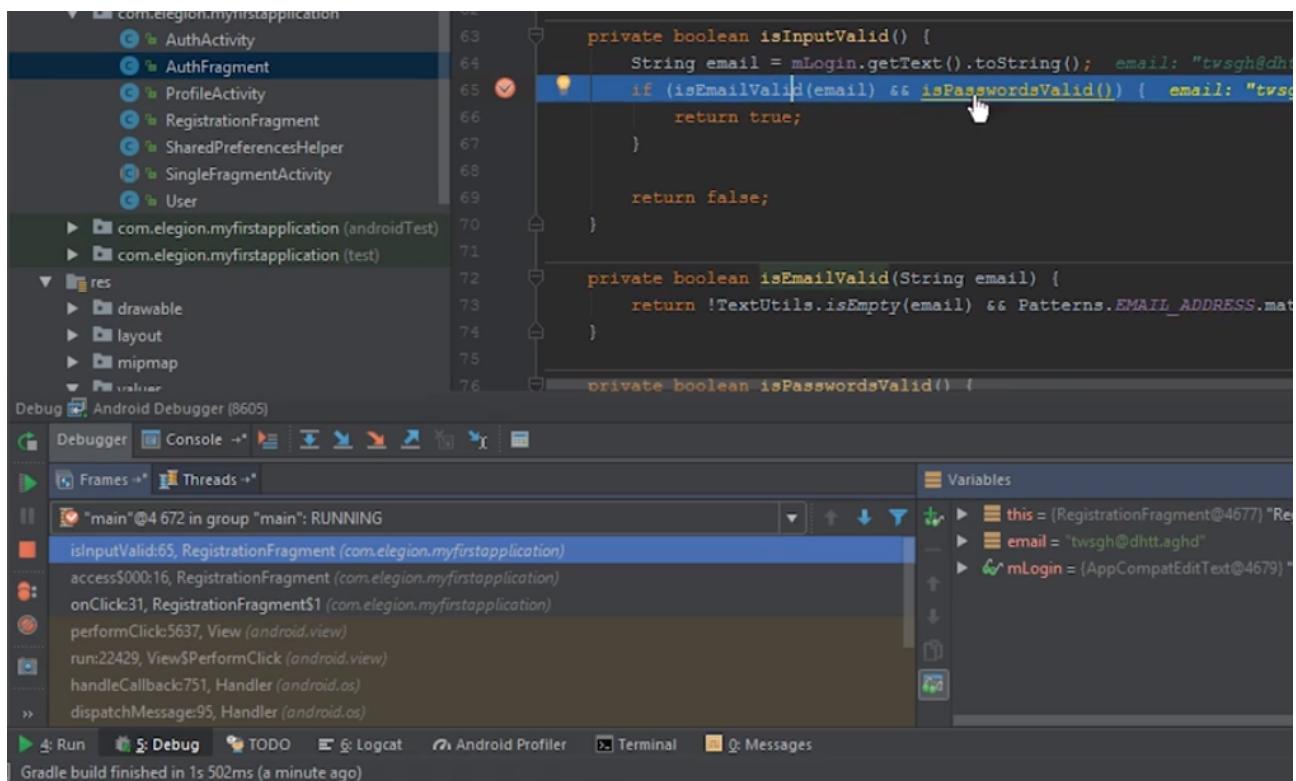


Рис. 3.9: Процесс дебага

`isValidEmail()` — true, `isValidPassword()` — false. Почему? Потому что перейдём в метод `isValidPassword()`; `password.equals()` — true, `isEmpty()` — false, `isEmpty()` — false. Всё верно, мы допустили ошибку в логике сравнения паролей. Добавим «не» перед ними; то есть если они не пустые, то они нам подходят

```
// RegistrationFragment.java :
private boolean isValidPassword() { //правильность пароля
    String password = mPassword.getText().toString();
    String passwordAgain = mPasswordAgain.getText().
        toString();
    return password.equals(passwordAgain)
        && TextUtils.isEmpty(password)
        && TextUtils.isEmpty(passwordAgain);
}
```

Запустим ещё раз и видим, что всё хорошо работает - при правильном вводе появляется тост с надписью "Успешно". Нажмём ещё раз - "Уже занято". Мы научились использовать SharedPreferences и научились писать какую-то бизнес-логику для приложения; эта бизнес-логика может подойти, когда вы не используете сервер, а пишете просто тестовые приложения

# Неделя 4

## Завершение курсового проекта

### 4.1 Добавление логики авторизации

Код по каждому обновлению в курсовом проекте (совпадает с пунктами недели):

1. КП. Логика авторизации. Работа с бэкстеком
2. КП. Экран профиля. Логаут, меню
3. КП. Обновлённая логика авторизации
4. КП. Экран профиля. Извлечение изображения из галереи
5. КП. Градиентный фон

#### 4.1.1 КП. Логика авторизации. Работа с бэкстеком

Рассмотрим, как работать с бэкстеком и как можно добавить логику авторизации в приложение. Откроем наш проект, зайдем в AuthFragment.java,

```
public class AuthFragment extends Fragment {  
    private EditText mLogin;  
    private EditText mPassword;  
    private Button mEnter;  
    private Button mRegister;  
    // добавим mSharedPreferencesHelper  
    private SharedPreferencesHelper  
    mSharedPreferencesHelper;  
    // и ниже в onCreateView проинициализируем его и дадим ему Activity  
    public View onCreateView(LayoutInflater inflater,  
        @Nullable ViewGroup container,  
        @Nullable Bundle savedInstanceState) {  
        View v = inflater.inflate(R.layout.fr_auth,  
            container, false);  
        mSharedPreferencesHelper = new  
        SharedPreferencesHelper(getActivity());  
    }  
}
```

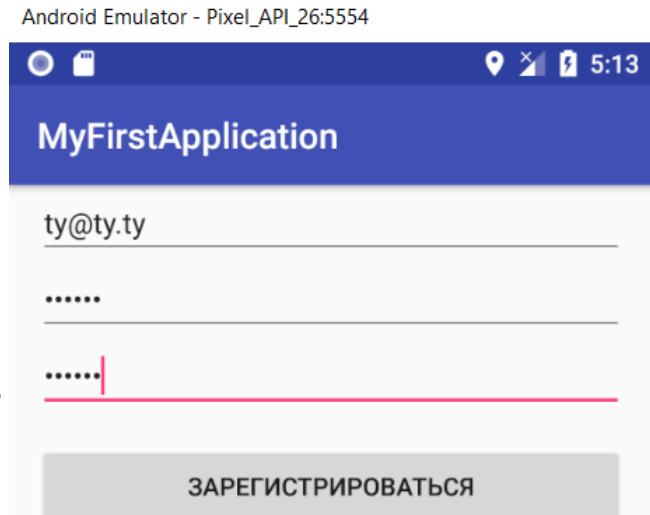
Дальше в mOnEnterClickListener(), то есть что происходит при нажатии на вход, добавляем логику проверки логина на наличие в SharedPrefereces и проверку пароля. После изменений OnClickListener() выглядит так:

```
@Override
public void onClick(View view) {
    boolean isLoginSuccess = false; //проверка удачности входа
    for (User user : mSharedPreferencesHelper.getUsers()) {
        //дописываем проверку логина и пароля
        if (user.getLogin().equalsIgnoreCase(
            mLogin.getText().toString()) && user.getPassword() .
            equals(mPassword.getText().toString())) {
            isLoginSuccess = true; //успешно вошли
            if (isValidEmail() && isPasswordValid()) {
                Intent startProfileIntent =
                    new Intent(getActivity(), ProfileActivity.class);
                startProfileIntent.putExtra(ProfileActivity.
                    USER_KEY
                    , new User(mLogin.getText().toString(), mPassword .
                        getText().toString()));
                startActivity(startProfileIntent);
                getActivity().finish();
            } else { //если не удалось войти пишем ошибку входа
                showMessage(R.string.input_error);
            }
            break; //чтобы цикл прекратил свою работу
        }
    }
}
```

где строку R.string.input\_error предварительно добавили в строковые ресурсы:

```
<string name="login_error">Error. Wrong login or password
</string>
```

Запустим. Для начала зарегистрируем какого-нибудь пользователя. Введем ему Email: ty@ty.ty. Введем пароль: qwerty. Пароль еще раз: qwerty. Enter, если точнее, зарегистрироваться. Зарегистрировались успешно, но почему-то после успешной регистрации не закрывается данный экран. Давайте попробуем перейти назад. Приложение закрылось. Проблема в том, что у нас неверно настроена обработка по клавише «Назад». Давайте исправим это.



Перейдем в SingleFragmentActivity.java, переопределим метод onBackPressed().

```
public void onBackPressed() {
    FragmentManager fragmentManager =
        getSupportFragmentManager();
    if (fragmentManager.getBackStackEntryCount() == 1) {
        finish(); // закончим работу, если это последний экран
    } else { // иначе вернёмся на предыдущий экран
        fragmentManager.popBackStack();
    }
}
```

AuthFragment.java в onRegisterClickListener() добавим логику нажатия клавиши «Зарегистрироваться».

```
private View.OnClickListener mOnRegisterClickListener
    = new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        getFragmentManager().beginTransaction()
            .replace(R.id.fragmentContainer,
                RegistrationFragment.newInstance())
            .addToBackStack(RegistrationFragment
                .class.getName()).commit();
    }
};
```

У нас формируется бэкстэк и по нажатию на клавишу назад мы переходим на предыдущий фрагмент, а не закрываем приложение.

Допишем в RegistrationFragment.java логику в mOnRegistrationClickListener()

```

private View.OnClickListener mOnRegistrationClickListener
    = new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        if (isValidInput()) {
            boolean isAdded = mSharedPreferencesHelper.
                addUser(new User(mLogin.getText().toString(),
                    mPassword.getText().toString()));
            if (isAdded) {
                showMessage(R.string.login_register_success);
                getFragmentManager().popBackStack(); // добавили
            } else {showMessage(R.string.login_register_error);}
        }
    }
};

```

Запустим эмулятор. Введём емейл, пароли и Зарегистрируемся. Регистрация прошла успешно, экран закрылся, все работает правильно. Введем логин с ошибкой при правильном пароле. Нажмем «Войти» — ошибка логина, неверный логин или пароль. Все работает правильно. Введём верные данные нас пустит в приложение. Нажмем «Войти» — нас впустило в приложения. Все логика написана правильно.



Рис. 4.1: Неверный ввод

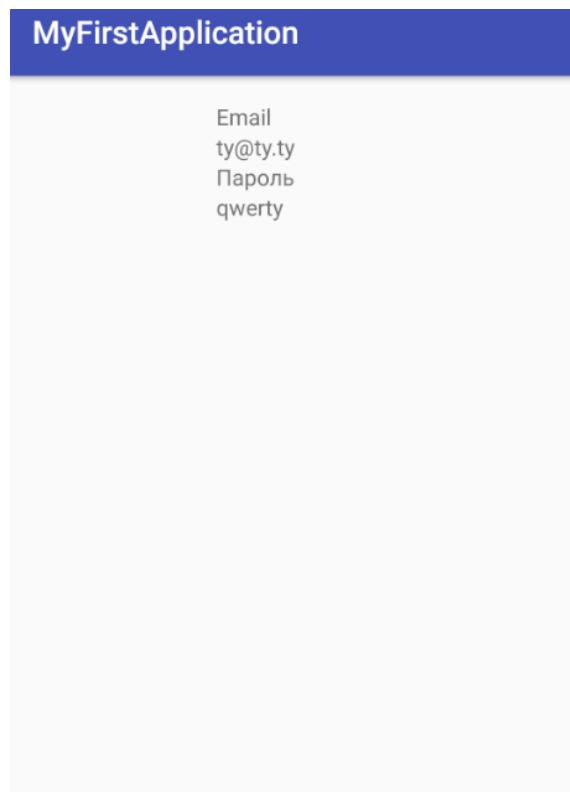


Рис. 4.2: Верный ввод

#### 4.1.2 КП. Экран профиля. Логаут, меню

Добавим меню внутри Activity. Проект ⇒ res ⇒ New ⇒ Android resource directory.

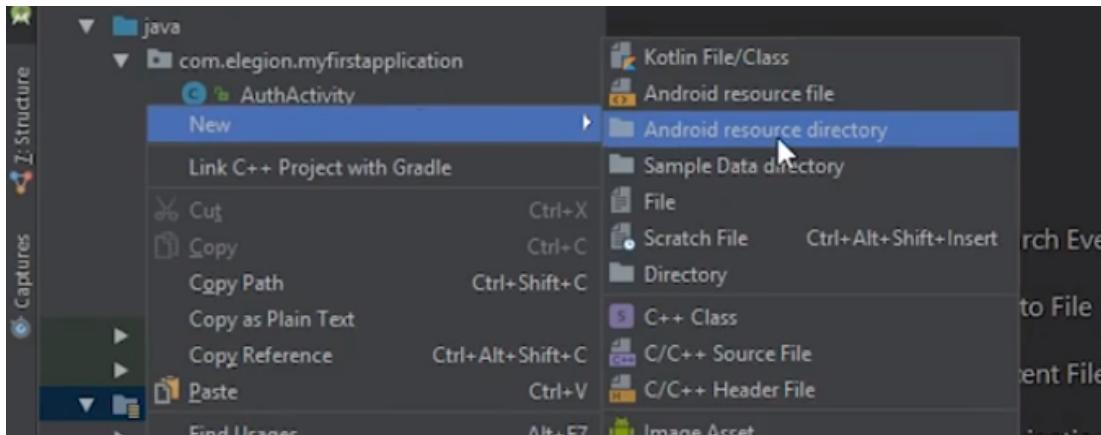


Рис. 4.3: Добавим Android resource directory

В Resource type выбираем меню, нажимаем OK. У нас добавилась папочка.

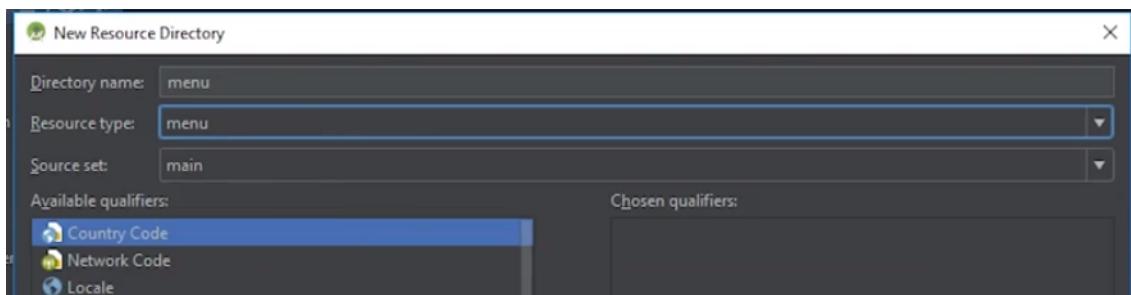


Рис. 4.4: Выбираем тип ресурса - меню

Далее, правый клик menu⇒ New⇒ Menu resource file. Назовем меню profile\_menu.

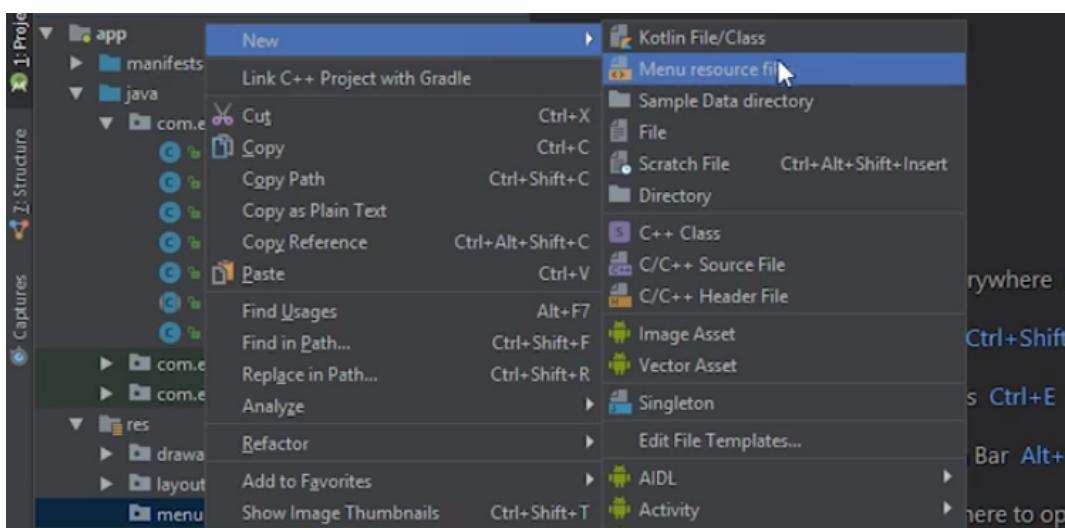


Рис. 4.5: В папке menu создаём ресурный файл разметки для меню

Мы будем использовать это меню внутри активности профиля. Мы добавим функциональность логаута из приложения.

## Листинг 4.1: profile\_menu.xml

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/
    android"
    xmlns:app="http://schemas.android.com/apk/res-auto">
    <item
        android:id="@+id/actionLogout"
        android:title="@string/logout"
        app:showAsAction="never"/>
</menu>
```

Добавим строковый ресурс string/logout="logout". Теперь добавим это меню в активность. Переходим в ProfileActivity.java.

Переопределяем методы OnCreateOptionsMenu() и OnOptionsItemSelected():

```
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.profile_menu, menu);
    return super.onCreateOptionsMenu(menu);
}
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) { //стандартный оператор java
        case R.id.actionLogout:
            startActivity(new Intent(this, AuthActivity.class));
            finish(); //запускаем активити авторизации и закрываем эту
            break;
        default:break;
    }
    return super.onOptionsItemSelected(item);
}
```

Посмотрим, как это работает в эмуляторе. Вводим существующие данные и Нажимаем Войти. У нас добавилось меню. Нажимаем Логаут. У нас открылся экран без логина и пароля. Мы научились использовать меню внутри Activity.

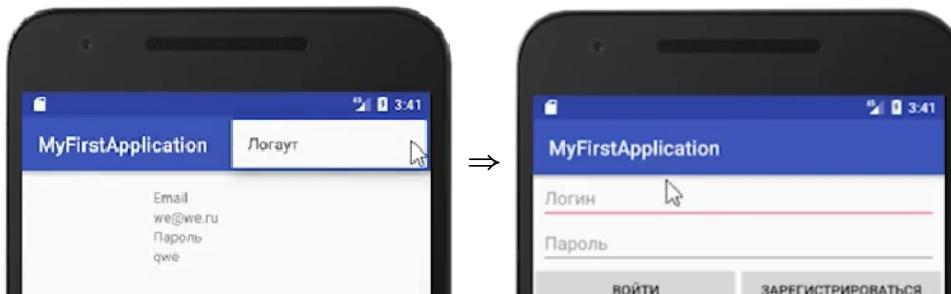


Рис. 4.6: Логаут

Рис. 4.7: Мы вышли

### 4.1.3 КП. Обновлённая логика авторизации

Добавим в поле логина список успешно авторизованных пользователей. То есть мы будем нажимать на логин, и у нас под ним будет выпадать список пользователей, которые уже авторизованы. Как вы помните, в предыдущих занятиях мы добавили логику авторизации и мы эту логику перенесли в SharedPreferencesHelper. Давайте посмотрим, как это выглядит теперь. Откроем проект, перейдем в AuthFragment. И посмотрим, как теперь выглядит mOnEnterClickListener:

```
private View.OnClickListener mOnEnterClickListener = new
    View.OnClickListener() {
    @Override
    public void onClick(View view) {
        if (isValidEmail() && isValidPassword()) {
            if (mSharedPreferencesHelper.login(new User(
                mLogin.getText().toString(),
                mPassword.getText().toString()))) {
                Intent startProfileIntent =
                    new Intent(getActivity(), ProfileActivity.class);
                startProfileIntent.putExtra(ProfileActivity.
                    USER_KEY, new User(mLogin.getText().toString(),
                        mPassword.getText().toString()));
                startActivity(startProfileIntent);
                getActivity().finish();
            } else {
                showMessage(R.string.login_error);
            }
        } else {
            showMessage(R.string.input_error);
        }
        for (User user:mSharedPreferencesHelper.getUsers()) {
            if (user.getLogin().equalsIgnoreCase(mLogin.
                getText().toString())&& user.getPassword().
                    equals(mPassword.getText().toString()))
            {
                break;
            }
        }
    }
};
```

Здесь у нас появился метод login(). Если вас будет интересовать какая-то бизнес-логика, которую мы не разбирали при вас, вы просто сможете посмотреть ее в проекте. mSharedPreferencesHelper.login(). Вся та же логика, что была до этого,

просто она перенеслась в SharedPreferencesHelper.

Чтобы показать выпадающий список, нам нужно: перейти в layout, fr\_auth.xml, открыть preview(Text), заменить первый <EditText на <AutoCompleteTextView,

```
...before
<EditText
...after
<AutoCompleteTextView
```

перейти в AuthFragment, заменить EditText логина на AutoCompleteTextView.

```
private EditText mLogin;
//стало
private AutoCompleteTextView mLogin;
...
private ArrayAdapter<String> mLoginedUsersAdapter; // дописали
... // в OnCreateView проинициализируем:
mLoginedUsersAdapter = new ArrayAdapter<>(getActivity(),
    android.R.layout.simple_dropdown_item_1line,
    mSharedPreferencesHelper.getSuccessLogins());
// добавим адаптер в сам AutoCompleteTextView:
mLogin.setAdapter(mLoginedUsersAdapter);
```

Сначала передаем контекст (можно передать getActivity()), далее передаем ему ссылку на наш layout-файл. Мы будем использовать стандартные системные ресурсы Андроида. И передать ему сами данные для отображения. Чтобы это сделать, мы берем mSharedPreferencesHelper и вызываем у него метод get SuccessLogins(). Но пока мы не увидим выпадающего списка, если кого-то зарегистрируем, потому что мы с вами не добавили показ нашего выпадающего списка, когда у нас изменяется фокус. Создадим FocusChangeListener()

```
private View.OnFocusChangeListener
mOnLoginFocusChangeListener = new View.
OnFocusChangeListener() {
@Override
public void onFocusChange(View view, boolean hasFocus)
{
    if (hasFocus) { // если фокус есть, покажем autocomplete
        mLogin.showDropDown();
    }
}
};
```

boolean в переименуем в HasFocus, чтобы было понятнее. И если у нас есть фокус, то в нашем AutoCompleteTextView вызовем метод ShowDropDown(). Также

мы забыли добавить OnFocusChangeListener() в наш AutoCompleteTextView.

```
//... исправляем
mEnter.setOnClickListener(mOnEnterClickListener);
mRegister.setOnClickListener(mOnRegisterClickListener);
//дописываем
mLogin.setOnFocusChangeListener(
    mOnLoginFocusChangeListener);
```

Запустим эмулятор. Выпадающий список показался, значит всё правильно.

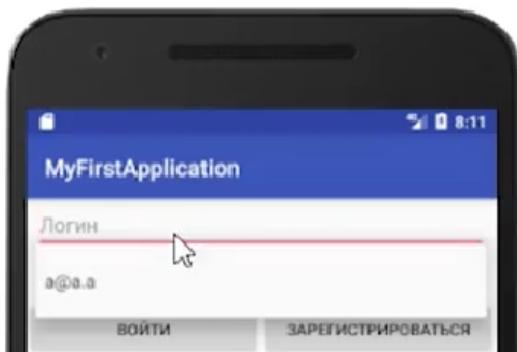


Рис. 4.8: Меню выбора логина

#### 4.1.4 КП. Экран профиля. Извлечение изображения из галереи

Научимся получать изображение из галереи. И изменим логику логина в приложение так, чтобы вместо boolean нам возвращался пользователь. Перейдем в SharedPreferencesHelper, сотрем входной параметр user, вместо этого добавим login и password. Входной параметр будет user. User get login заменяем на login и user get password заменяя на password. В качестве return возвращаем ему пользователя, которого он нашел в нашем shared preference хранилище.

```
public User login(String login, String password) {
    List<User> users = getUsers();
    for (User u : users) {
        if (login.equalsIgnoreCase(u.getLogin()))
            && password.equals(u.getPassword())) {
            u.setHasSuccessLogin(true);
            mSharedPreferences.edit().putString(USER_KEY,
                mGson.toJson(users, USERS_TYPE)).apply();
            return u;
        }
    }
    return null; //если никого не нашёл
}
```

Теперь изменим использование этого метода. Стираем здесь new user и перед этим создадим его. наш user будет равняться результату логина.

```
private View.OnClickListener mOnEnterClickListener = new
    View.OnClickListener() {
    @Override
    public void onClick(View view) {
        if (isValidEmail() && isValidPassword()) {
            User user = mSharedPreferencesHelper.login(
                mLogin.getText().toString(),
                mPassword.getText().toString());
            if (user != null) {
                Intent startProfileIntent =
                    new Intent(getActivity(), ProfileActivity.
                        class);
                startProfileIntent.putExtra(ProfileActivity.
                    USER_KEY, user);
                startActivity(startProfileIntent);
                getActivity().finish();
            } else {
                showMessage(R.string.login_error);
            }
        } else {
            showMessage(R.string.input_error);
        }
    }
}
```

Соответственно, если юзер != null, то можно переходить на следующий экран. И в качестве параметра следующего экрана мы будем передавать user, которого получили из нашего хранилища с помощью метода login(). Иначе у нас будет отображаться ошибка логина. Чтобы получить изображение из галереи, перейдем в profile activity. В mOnPhotoClickListener() мы добавим метод:

```
private View.OnClickListener mOnPhotoClickListener = new
    View.OnClickListener() {
    @Override
    public void onClick(View view) {
        openGallery(); //при клике на фото открываем галерею.
    }
};
private void openGallery() {
    Intent intent = new Intent(); //создаём интент типа картинка
    intent.setType("image/*");
    intent.setAction(Intent.ACTION_GET_CONTENT); //взяли фото
    startActivityForResult(intent, REQUEST_CODE_GET_PHOTO);
}
```

И дописываем в начало ProfileActivity свой request\_code

```
public static final int REQUEST_CODE_GET_PHOTO = 101;
```

нужно переопределить метод onActivityResult()

```
@Override
protected void onActivityResult(int requestCode,
                                int resultCode, Intent data) {
//если подаётся наш код запроса, результат OK и Data не null
    if (requestCode == REQUEST_CODE_GET_PHOTO
        && resultCode == Activity.RESULT_OK
        && data != null) {
        Uri photoUri = data.getData(); //ссылка на файл
        mPhoto.setImageURI(photoUri); //установим в изображение
    } else {
        super.onActivityResult(requestCode, resultCode, data);
    }
}
```

Запустим эмулятор. Выберем login, введем пароль, «войти», нажмем на наш image view, выберем какую-нибудь картинку, пусть это будет собачка. Как вы видите, картинка успешно установилась, то есть мы открыли галерею и выбрали картинку.

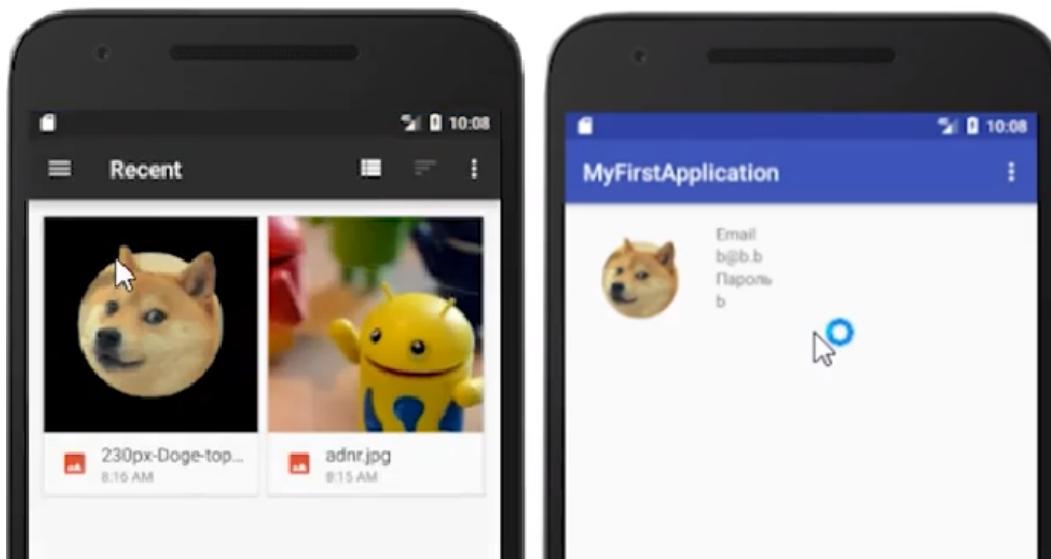


Рис. 4.9: Добавление картинки профиля

В данном уроке мы научились получать thumbnail картинки из галереи.

#### 4.1.5 КП. Градиентный фон

Рассмотрим, как можно создать градиент для фона, чтобы наше приложение выглядело лучше. Для этого перейдём в папочку res ⇒ drawable ⇒ New ⇒

Drawable resource file ⇒ File name: gradient background.

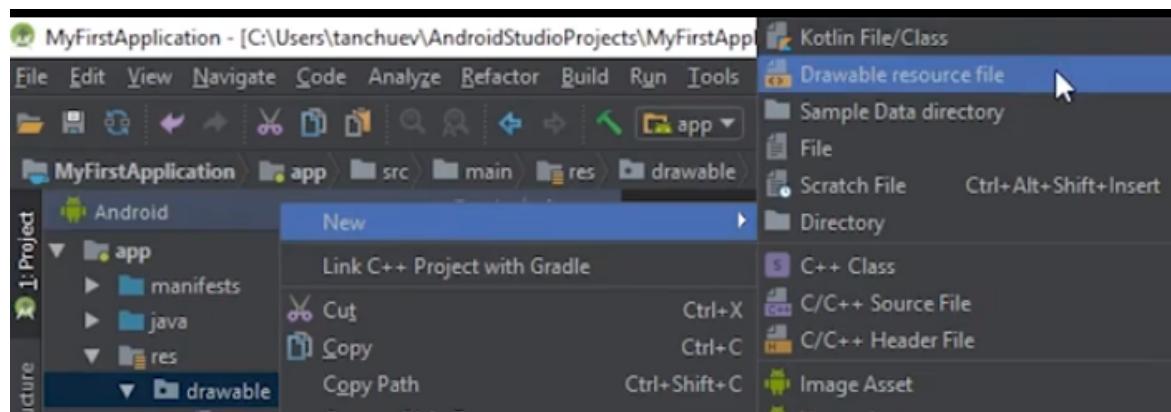


Рис. 4.10: Создание Drawable resource file

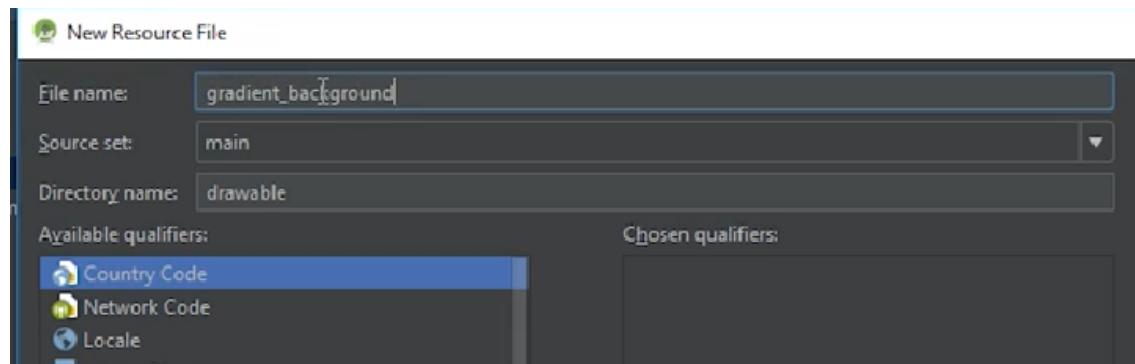


Рис. 4.11: Добавление градиентного фона

Внутри добавляем item, внутри item — shape, внутри shape добавляем gradient. Зададим угол 90, startColor = PrimaryDark. и endColor=colorAccent. И затем ему type=linear, то есть он по умолчанию. И закрывающийся тег.

Листинг 4.2: gradient\_background.xml

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item>
        <shape>
            <gradient android:angle="90"
                      android:endColor="@color/colorAccent"
                      android:startColor="@color/colorPrimaryDark"
                      android:type="linear"/>
        </shape>
    </item>
</selector>
```

Градиент создан. Перейдём в fr\_auth.xml, в корневой LinearLayout добавим

```
android:background="@drawable/gradient_background"
```

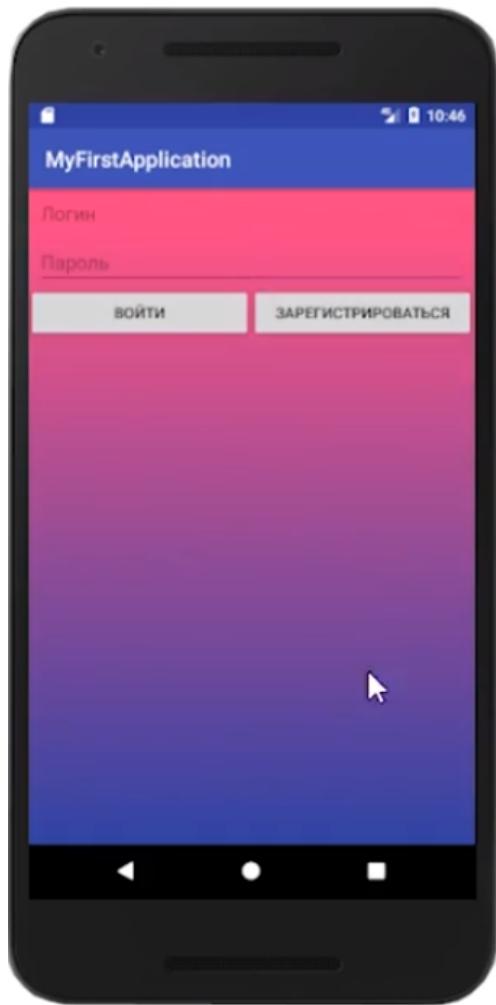


Рис. 4.12: Градиентный фон

Давайте запустим эмулятор и посмотрим, как это выглядит. Наш эмулятор запустился. И мы видим фон, который мы задали. В данном занятии мы научились создавать gradient background и устанавливать его как фон для экрана или какого-то layout.

Итак, мы с вами создали ваше первое Android-приложение. Оно содержит три экрана: экран авторизации, экран регистрации и экран показа профиля. Это приложение пригодится нам и в будущем; мы будем его изменять, улучшать и дорабатывать.