

Активити и ЖЦ

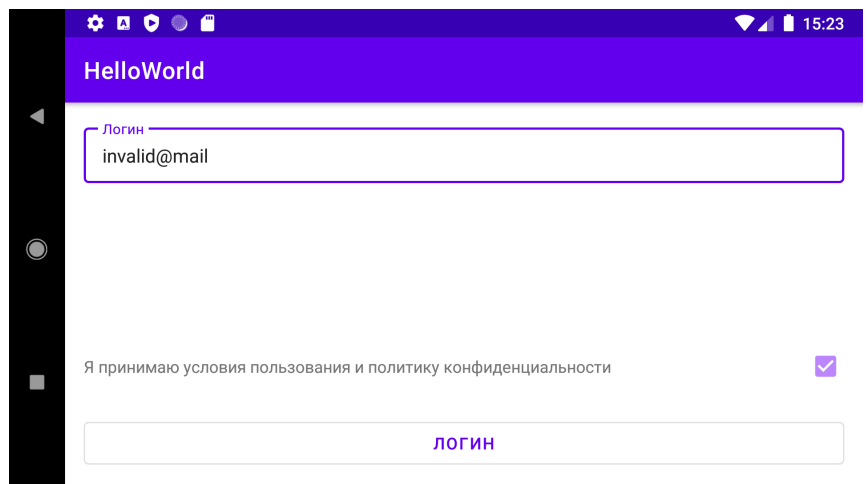
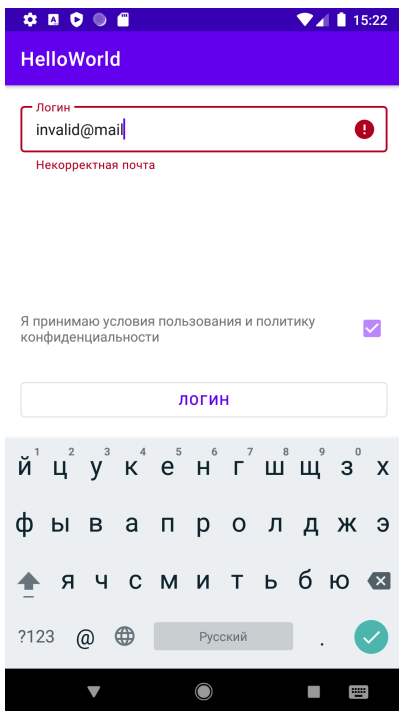
Следим за состоянием экрана

Содержание

1. Поворот экрана
2. Bundle и методы жизненного цикла активити
3. Введение в MVVM

1. Поворот экрана

Мы продолжаем рассматривать задачу логин экрана. Если помните у нас было поле ввода, кнопка и загрузчик. Давайте рассмотрим такой кейс – введем невалидные данные, поставим галку и нажмем на кнопку. Получаем ошибку и поворачиваем экран



Как видите состояние ошибки исчезло, хотя мы не делали ничего кроме как поворота экрана.

Если хотите сделайте еще вот как: введите валидный мейл, нажмите на кнопку и дождитесь ошибки сервера (имитация) : помните, мы сделали диалог? Так вот, после поворота оно исчезает. Да почему же так? И сегодня мы познакомимся с жизненным циклом Активити.

И мы уже знаем про один из методов жизненного цикла : onCreate. Давайте для начала залогим вызов и узнаем побольше об аргументе. Также мы убирали ошибку когда текст в поле ввода менялся. Давайте залогим также и там. Посмотрим в логкат что происходит

```

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    Log.d(TAG, msg: "onCreate ${savedInstanceState == null}")

    init views
    loginButton.setOnClickListener {...}
    textInputEditText.listenChanges { it: String
        Log.d(TAG, msg: "changed $it")
        textInputLayout.isErrorEnabled = false
    }
}

```

Итак, вводим мейл, жмем кнопку, поворот экрана и смотрим логи

```

ysc.helloworld D/TextWatcherTag: onCreate true
ysc.helloworld D/TextWatcherTag: changed i
ysc.helloworld D/TextWatcherTag: changed in
ysc.helloworld D/TextWatcherTag: changed inv
ysc.helloworld D/TextWatcherTag: changed inva
ysc.helloworld D/TextWatcherTag: changed inval
ysc.helloworld D/TextWatcherTag: changed invali
ysc.helloworld D/TextWatcherTag: changed invalid
ysc.helloworld D/TextWatcherTag: changed invalid@
ysc.helloworld D/TextWatcherTag: changed invalid@m
ysc.helloworld D/TextWatcherTag: changed invalid@ma
ysc.helloworld D/TextWatcherTag: changed invalid@mai
ysc.helloworld D/TextWatcherTag: changed invalid@mail
ysc.helloworld D/TextWatcherTag: onCreate false
ysc.helloworld D/TextWatcherTag: changed invalid@mail

```

Как видим сначала у нас залогирован метод onCreate и его аргумент null. Все хорошо. После ввода текста и нажатия кнопки мы повернули и наш метод onCreate вызвался еще раз но с непустым аргументом! И после этого еще раз вызвался слушатель текста.

Да, друзья мои. Как помните я говорил ранее что мы можем написать разные ресурсы под разные конфигурации : en/strings.xml, ru/strings.xml так же можно иметь разные layout.xml

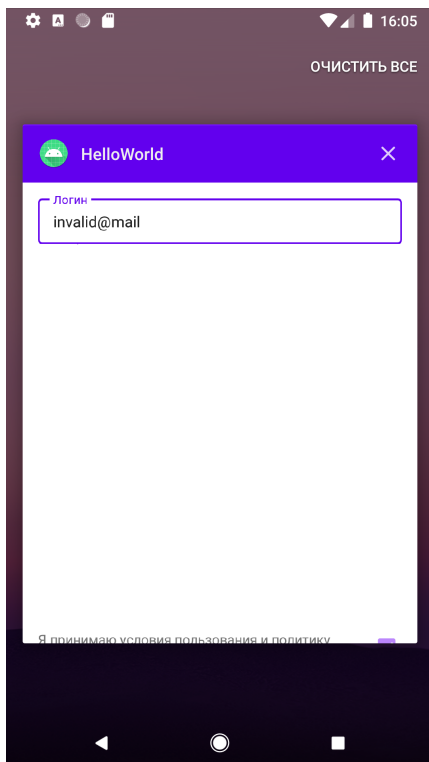
для портретной ориентации и landscape. И давайте пару слов скажем о них и о том, что делает `LayoutInflater` под капотом метода `setContentView`. В xml у нас текст в формате xml и он просто лежит в проекте. Когда мы открываем активити то в методе `setContentView(R.layout.activity_main)` берется `LayoutInflater` и читает наш xml файл преобразовывая вью в джава классы. Т.е. если у вас в xml написан `<TextView` то он превращается в `class TextView` который мы после уже находим с помощью метода `findViewById`. Теперь, если мы поворачиваем экран, то вне зависимости есть у нас разметка (layout файл) под другую ориентацию или нет, андроид пересоздает активити и вызывает метод `onCreate` заново. Но чтобы вы не потеряли свои данные предлагает вам аргумент `savedInstanceState`. А что по поводу поля ввода? Почему же там вызывается слушатель? Когда пересоздается активити то и разметка заново загружается, потому что естественно ширина и высота не равны друг другу (нет в андроид идеально квадратных девайсов), да и даже если равны друг другу то все равно заново высчитывается и вырисовывается вью (об этом как-нибудь в другой лекции), а поля ввода еще имеют свойство фокусирования. Это когда клавиатура поднята и вы можете вводить текст именно в этом поле ввода (условно, потому что можно клавиатуру опустить, а поле ввода все равно будет сфокусировано). Если быть точнее, то состояние фокуса это когда поле ввода выбрано. Так вот, после пересоздания активити выбирается вью для фокусировки (если у вас больше 1 поля ввода). Но это все равно не объясняет почему в поле ввода было изменение. Для этого нужно понять что такое `Bundle` : тип аргумента для метода `onCreate(savedInstanceState:Bundle?)`

2. Bundle и методы жизненного цикла активити

Итак, у нас пересоздается активити, но как тогда текст в поле ввода остался таким каким он был? Что за магия произошла? Перед смертью активити выделяется память под данные которые нужны будут для восстановления и под это дело и нужен объект типа `Bundle`.

По факту перед смертью активити в бандл по айди нашей вью система положила его значение и когда активити восстановилось из этого бандла значение взяли и поставили в поле ввода, именно поэтому и мы видим вызов метода в слушателе. И так как при изменении текста в поле ввода у нас очищается ошибка, то мы и не видим ее. Как это исправить? На самом деле легко! Нужно слушать изменения поля ввода лишь тогда, когда юзер может с ним взаимодействовать и прекращать обращать внимание на изменения когда юзер не видит поле ввода. Но секунду, как мы можем знать когда юзер видит поле ввода, а когда нет? Чисто логически, если у нас есть метод активити когда оно создается или пересоздается, то и должны быть другие методы. Можете погуглить методы жизненного цикла активити и увидеть всю картину. А мы для начала рассмотрим лишь 2 из них – `onResume`, `onPause`. Первый метод вызывается когда юзер уже видит и может взаимодействовать с активити. И здесь у кого-то возникнет вопрос : если юзер видит активити то и может с ней взаимодействовать, нет? Нет, бывают ситуации когда ваша активити видна частично, ибо ее перекрывает диалоговое окно например или когда вы в режиме просмотра задач: вы видите что у вас на активити, но взаимодействовать не можете пока не выберете ее. Хорошо, разобрались. Теперь, нам нужно перед тем как активити умрет и пересоздастся убрать и поставить заново наш лиснер поля ввода. Как это делается? Легко. Сначала выделяем `textWatcher` , в методе `onResume` добавляем к полю ввода, а в методе `onPause` убираем его.

И теперь при повороте у вас не будет проблем с состоянием поля ввода. Смотрите сами.



Вот это и есть состояние onPause. Активити видна, но юзер не может с ней взаимодействовать

```
private lateinit var textInputLayout:TextInputLayout
private lateinit var textInputEditText: TextInputEditText

private val textWatcher = object : SimpleTextWatcher() {
    override fun afterTextChanged(s: Editable?) {
        Log.d(TAG, msg: "changed ${s.toString()}")
        textInputLayout.isErrorEnabled = false
    }
}

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    Log.d(TAG, msg: "onCreate ${savedInstanceState == null}")

    //region init views
    textInputLayout = findViewById(R.id.textInputLayout)
    textInputEditText = textInputLayout.editText as TextInputEditText
    textInputLayout.findViewById(R.id.textInputLayout)
}
```

Да, у вас встает проблема с тем, что вам нужны вью вне метода onCreate и у вас 2 выбора – lateinit или же nullable. Я выбрал первое потому что не люблю нулабельные, но и это не самое правильное решение, а какое правильное? Узнаем позже когда пройдем вьюбиндинг.

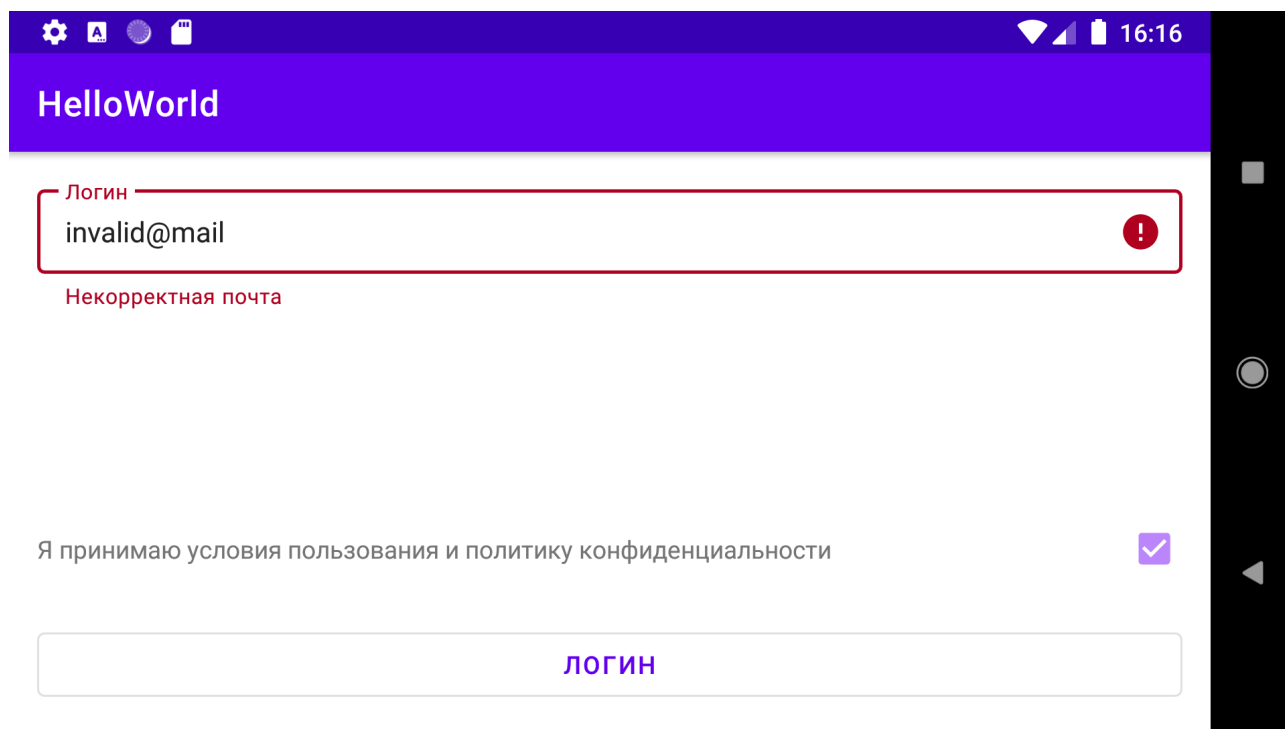
```

override fun onPause() {
    super.onPause()
    textInputEditText.removeTextChangedListener(textWatcher)
}

override fun onResume() {
    super.onResume()
    textInputEditText.addTextChangedListener(textWatcher)
}

```

Ну что ж, протестируем? Давайте!



HelloWorld

Логин

invalid@mail

Некорректная почта

Я принимаю условия пользования и политику конфиденциальности ☒

ЛОГИН

И чекнем логи (кстати, в логкате можно сделать скриншот, слева столбец иконок, фото).



logcat

2021-06-05 16:13:38.216 16889-16889/com.github.johnnysc.helloworld D/TextWatcherTag: onCreate true

2021-06-05 16:16:08.080 16889-16889/com.github.johnnysc.helloworld D/TextWatcherTag: changed i

2021-06-05 16:16:08.160 16889-16889/com.github.johnnysc.helloworld D/TextWatcherTag: changed in

2021-06-05 16:16:08.286 16889-16889/com.github.johnnysc.helloworld D/TextWatcherTag: changed inv

2021-06-05 16:16:08.345 16889-16889/com.github.johnnysc.helloworld D/TextWatcherTag: changed inval

2021-06-05 16:16:08.396 16889-16889/com.github.johnnysc.helloworld D/TextWatcherTag: changed invali

2021-06-05 16:16:08.418 16889-16889/com.github.johnnysc.helloworld D/TextWatcherTag: changed invalid

2021-06-05 16:16:08.539 16889-16889/com.github.johnnysc.helloworld D/TextWatcherTag: changed invalid@

2021-06-05 16:16:09.495 16889-16889/com.github.johnnysc.helloworld D/TextWatcherTag: changed invalid@ma

2021-06-05 16:16:09.626 16889-16889/com.github.johnnysc.helloworld D/TextWatcherTag: changed invalid@mai

2021-06-05 16:16:09.793 16889-16889/com.github.johnnysc.helloworld D/TextWatcherTag: changed invalid@mail

2021-06-05 16:16:09.854 16889-16889/com.github.johnnysc.helloworld D/TextWatcherTag: changed invalid@mail

2021-06-05 16:16:09.969 16889-16889/com.github.johnnysc.helloworld D/TextWatcherTag: changed invalid@mail

2021-06-05 16:16:18.582 16889-16889/com.github.johnnysc.helloworld D/TextWatcherTag: onCreate false

Да, как видите у нас не вызвался метод, потому что мы и не слушали изменения когда поворачивался экран. Чтобы быть уверенным в этом, давайте залогируем методы.

```

2021-06-05 16:20:20.163 17177-17177/com.github.johnnysc.helloworld D/TextWatcherTag: onCreate true
2021-06-05 16:20:20.170 17177-17177/com.github.johnnysc.helloworld D/TextWatcherTag: onResume
2021-06-05 16:20:22.162 17177-17177/com.github.johnnysc.helloworld D/TextWatcherTag: changed i
2021-06-05 16:20:22.190 17177-17177/com.github.johnnysc.helloworld D/TextWatcherTag: changed in
2021-06-05 16:20:22.294 17177-17177/com.github.johnnysc.helloworld D/TextWatcherTag: changed inv
2021-06-05 16:20:22.362 17177-17177/com.github.johnnysc.helloworld D/TextWatcherTag: changed inva
2021-06-05 16:20:22.413 17177-17177/com.github.johnnysc.helloworld D/TextWatcherTag: changed inval
2021-06-05 16:20:22.462 17177-17177/com.github.johnnysc.helloworld D/TextWatcherTag: changed invali
2021-06-05 16:20:22.549 17177-17177/com.github.johnnysc.helloworld D/TextWatcherTag: changed invalid
2021-06-05 16:20:23.190 17177-17177/com.github.johnnysc.helloworld D/TextWatcherTag: changed invalid@
2021-06-05 16:20:23.353 17177-17177/com.github.johnnysc.helloworld D/TextWatcherTag: changed invalid@m
2021-06-05 16:20:23.451 17177-17177/com.github.johnnysc.helloworld D/TextWatcherTag: changed invalid@ma
2021-06-05 16:20:23.525 17177-17177/com.github.johnnysc.helloworld D/TextWatcherTag: changed invalid@mai
2021-06-05 16:20:23.651 17177-17177/com.github.johnnysc.helloworld D/TextWatcherTag: changed invalid@mail
2021-06-05 16:20:30.137 17177-17177/com.github.johnnysc.helloworld D/TextWatcherTag: onPause
2021-06-05 16:20:30.242 17177-17177/com.github.johnnysc.helloworld D/TextWatcherTag: onCreate false
2021-06-05 16:20:30.255 17177-17177/com.github.johnnysc.helloworld D/TextWatcherTag: onResume

```

И я вам сразу скажу, что методы onPause и onResume обязательно будут вызваны. Так что вы всегда можете полагаться на них. Что значит обязательно? А какие необязательно тогда?

Ок, смотрите. У нас есть метод onCreate и если этот метод отвечает за создание, то по аналогии с onPause onResume пауза и воспроизведение, должен быть метод уничтожения активности, верно? Да, есть такой – onDestroy. Но он не обязательно вызывается, так что не полагайтесь на него. Почему же так? Я уже говорил что onPause вызывается когда мы уходим с экрана в диспетчер задач и если вы знаете, то мы можем смахнуть активности и тем самым убить ее. Метод onDestroy не успеет вызваться. Чтобы в этом убедиться, давайте залогируем метод onDestroy, сначала снимем логи, когда мы повернули экран и потом когда смахнули активности в диспетчере задач.

```

override fun onPause() {
    super.onPause()
    Log.d(TAG, msg: "onPause")
    textInputEditText.removeTextChangedListener(textWatcher)
}

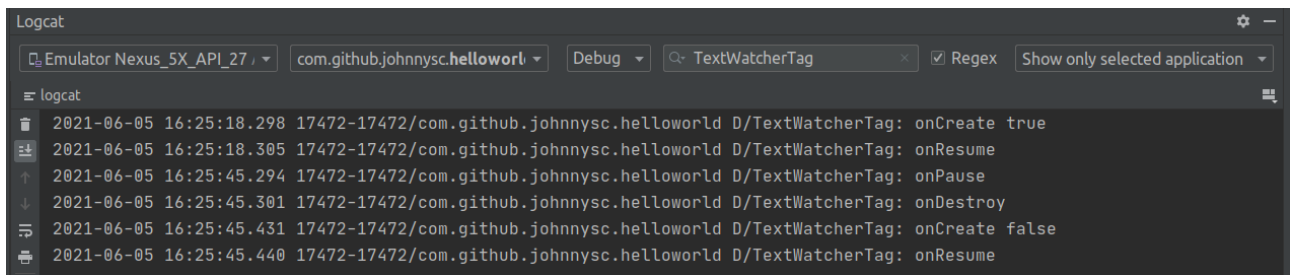
override fun onResume() {
    super.onResume()
    Log.d(TAG, msg: "onResume")
    textInputEditText.addTextChangedListener(textWatcher)
}

override fun onDestroy() {
    Log.d(TAG, msg: "onDestroy")
    super.onDestroy()
}

```

Сначала поворот! Я уберу пока вызов лога в слушателе поля ввода чтобы было меньше логов

Как видите, действительно, сначала пауза, потом смерть, потом создание



А теперь уберем активити смахиванием!

Проблема в том, что это не всегда так и onDestroy не вызывается только если недостаточно ресурсов т.е. памяти ОЗУ. Вы наверно видели много раз, когда вы сидите в одном приложении, ушли в другое, вернулись, а там все сбросилось. Это потому что пересоздалось активити, а программисты не написали внятный код чтобы сохранить состояние.

Давайте не будем как они и напишем такой код. Но чтобы понимать что мы сохраняем, нам нужно добиться ситуации когда состояние не сохраняется. Хотя мы же уже добились этого! Когда получали ошибку и выделили диалоговое окно, но после поворота оно исчезало.

Как же нам сохранять состояние? Мы говорили что у нас есть Bundle и в него можно положить данные и прочитать когда вызывается onCreate с непустым аргументом. Ок.

Давайте тогда напишем класс состояния нашего экрана. Состояний у нас конечно же 4. Начальное, прогрес, успех и ошибка. В чем проблема, еще раз. Если у нас долгий процесс получения данных от сервера и например он кончается успехом или ошибкой в то время когда юзер не видит приложение, то по возвращению он может не обнаружить ваше диалоговое окно или снейкбар, потому что при пересоздании диалоговое окно уйдет, а у снейкбара и вовсе своя продолжительность жизни (поэтому не делайте снейкбар, а юзайте статичные экраны успеха или уведомления) (в режиме разработчика DNKA).

Ок, давайте немного покодим! Напишем 4 константы под это дело числового типа

```
class MainActivity : AppCompatActivity() {  
  
    private companion object {  
        const val INITIAL = 0  
        const val PROGRESS = 1  
        const val SUCCESS = 2  
        const val FAILED = 3  
    }  
  
    private var state = INITIAL  
}
```

И нам нужна переменная которая хранит состояние, инициализируем ее сразу.

Немного поменяем наш код там где обрабатываем кнопку логина


```
loginButton.setOnClickListener { it: View!
    if (EMAIL_ADDRESS.matcher(textInputEditText.text.toString()).matches()) {
        hideKeyboard(textInputEditText)
        contentLayout.visibility = View.GONE
        progressBar.visibility = View.VISIBLE
        state = PROGRESS
        Handler(Looper.myLooper()!!).postDelayed({
            state = FAILED
            contentLayout.visibility = View.VISIBLE
            progressBar.visibility = View.GONE
            val dialog = Dialog(context: this)
            val view =
                LayoutInflater.from(context: this).inflate(R.layout.dialog, contentLayout,
                dialog.setCancelable(false)
                view.findViewById<View>(R.id.closeButton).setOnClickListener { it: View!
                    state = INITIAL
                    dialog.dismiss()
                }
                dialog.setContentView(view)
                dialog.show()
            }, delayMillis: 3000)
    },
```

Если все ок то мы переходим в состояние прогресса, при возвращении будет состояние ошибки, но когда мы уберем диалог, то будет состояние начальное. Надеюсь здесь все понятно. Мы сделали состояния чтобы например по истечению времени и пересоздании активити мы смогли увидеть все же наше диалоговое окно. Но мы не написали самого главного! Состояние же нужно сохранять, ведь если пересоздается активити, то и поля будут повторно инициализированы. Можете поставить стейт иной после создания и после поворота посмотреть что он поменялся обратно на начальный.

```
2021-06-05 16:45:22.195 18460-18460/com.github.johnnysc.helloworld D/TextWatcherTag: onCreate true
2021-06-05 16:45:22.195 18460-18460/com.github.johnnysc.helloworld D/TextWatcherTag: state is 2
2021-06-05 16:45:22.200 18460-18460/com.github.johnnysc.helloworld D/TextWatcherTag: onResume
2021-06-05 16:45:33.039 18460-18460/com.github.johnnysc.helloworld D/TextWatcherTag: onPause
2021-06-05 16:45:33.044 18460-18460/com.github.johnnysc.helloworld D/TextWatcherTag: onDestroy
2021-06-05 16:45:33.159 18460-18460/com.github.johnnysc.helloworld D/TextWatcherTag: onCreate false
2021-06-05 16:45:33.159 18460-18460/com.github.johnnysc.helloworld D/TextWatcherTag: state is 0
2021-06-05 16:45:33.164 18460-18460/com.github.johnnysc.helloworld D/TextWatcherTag: onResume
```

Кстати, 2 слова про коменты типа `//region myregion` `//endregion` если у вас много чего и вы хотите схлопнуть, то напишите эти 2 комента над и после блока кода и слева АС схлопнет их

```
Log.d(TAG, msg: "onCreate ${savedIn
if (savedInstanceState == null)
    state = SUCCESS
Log.d(TAG, msg: "state is $state")
//region init views
```

Как видите переменная не сохраняется сама по себе и для этого нужно ее положить в бандл.

Итак, давайте уже сделаем это! Если у нас бандл приходит во время создания активности, то где-то же он должен сохраняться перед смертью. И это метод `onSaveInstanceState`.

```
override fun onSaveInstanceState(outState: Bundle) {  
    super.onSaveInstanceState(outState)  
    outState.putInt("screenState", state)  
}
```

Будьте внимательны, у активности есть 2 метода с таким именем но разными аргументами.

Перепишем теперь `onCreate` и проверим!

```
super.onCreate(savedInstanceState)  
setContentView(R.layout.activity_main)  
Log.d(TAG, msg: "onCreate ${savedInstanceState == null}")  
savedInstanceState?.let { it: Bundle  
    state = it.getInt(key: "screenState")  
}  
Log.d(TAG, msg: "state is $state")  
//region init views
```

Если бандл не пуст, то я инициализирую переменную и логирую ее. Давайте теперь дойдем до конца сценария и повернем экран.

```
2021-06-05 16:50:56.987 18666-18666/com.github.johnnysc.helloworld D/TextWatcherTag: onCreate true  
2021-06-05 16:50:56.987 18666-18666/com.github.johnnysc.helloworld D/TextWatcherTag: state is 0  
2021-06-05 16:50:56.992 18666-18666/com.github.johnnysc.helloworld D/TextWatcherTag: onResume  
2021-06-05 16:51:15.506 18666-18666/com.github.johnnysc.helloworld D/TextWatcherTag: onPause  
2021-06-05 16:51:15.512 18666-18666/com.github.johnnysc.helloworld D/TextWatcherTag: onDestroy  
2021-06-05 16:51:15.650 18666-18666/com.github.johnnysc.helloworld D/TextWatcherTag: onCreate false  
2021-06-05 16:51:15.650 18666-18666/com.github.johnnysc.helloworld D/TextWatcherTag: state is 3  
2021-06-05 16:51:15.662 18666-18666/com.github.johnnysc.helloworld D/TextWatcherTag: onResume
```

Кстати, я забыл рассказать – в бандл легко класть переменные простого типа : числа или строки. Для этого у него есть простые методы типа `putInt`, `getInt`, `putString`, `getString`. Но вам нужно класть по ключу как в мапу и забирать из нее по тому же ключу. Поэтому я рекомендую не делать как на скриншоте, а выделить константу типа `const val KEY="screenState"`. Хорошо. Мы сохраняем наш стейт при повороте, но у нас до сих пор не отображается диалог когда мы повернули. Ну потому что мы не написали код который это делает. Давайте сделаем это. Но это не так просто, придется немного порефакторить код. Ведь у нас вызов одних и тех же блоков кода, значит надо выделить методы.

```

        when (state) {
            FAILED -> showDialog(contentLayout)
            SUCCESS -> {
                Snackbar.make(contentLayout, text: "Success", Snackbar.LENGTH_LONG).show()
                state = INITIAL
            }
        }
    }

    private fun showDialog(viewGroup: ViewGroup) {
        val dialog = Dialog(context: this)
        val view =
            LayoutInflater.from(context: this).inflate(R.layout.dialog, viewGroup, attachToRoot: false)
        dialog.setCancelable(false)
        view.findViewById<View>(R.id.closeButton).setOnClickListener { it: View!
            state = INITIAL
            dialog.dismiss()
        }
        dialog.setContentView(view)
        dialog.show()
    }
}

```

Теперь все должно быть верно.

```

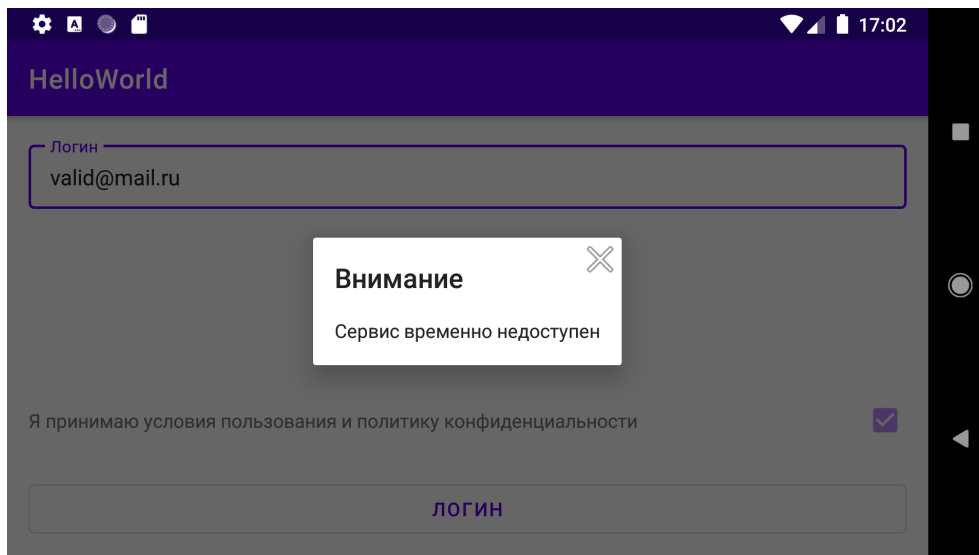
loginButton.setOnClickListener { it: View!
    if (EMAIL_ADDRESS.matcher(textInputEditText.text.toString()).matches()) {
        hideKeyboard(textInputEditText)
        contentLayout.visibility = View.GONE
        progressBar.visibility = View.VISIBLE
        state = PROGRESS
        Handler(Looper.myLooper()!!).postDelayed({
            state = FAILED
            contentLayout.visibility = View.VISIBLE
            progressBar.visibility = View.GONE
            showDialog(contentLayout)
        }, delayMillis: 3000)
    }
}

```

Мы в конце метода onCreate исходя из состояния экрана будем показывать или диалог или снейкбар. Давайте проверим же! Дойдем до состояния ошибки и повернем экран.

Как видите все правильно отображается. Мы сохранили стейт перед смертью активности и когда пересоздался мы показали диалог. Теперь же можно проверить еще раз поворот. Закройте диалоговое окно и еще раз повернуть. Никогда не забывайте возвращать состояние экрана. Ведь если вы не вернете стейт после закрытия диалога, то он так и останется фейлд, значит после закрытия диалога если повернете экран то снова увидите диалог об ошибке.

Ок, давайте же посмотрим на то, во что превратился наш мейнактивити класс. И учтите, мы еще не написали код для прогресса, ведь у нас в реальной жизни может запрос идти не 2 секунды, а дольше



Не знаю как у вас, а у меня код разросся до более 100 линий кода. И сейчас самое время поговорить про SOLID.

3. Введение в MVVM

До сих пор мы рассмотрели лишь принципы ООП: инкапсуляция, наследование и полиморфизм. В реальной разработке мы основываемся на куда более большем числе принципов. Группой принципов основанных на принципах ООП является SOLID: Single Responsibility, Open-closed, Liskov substitution, interface segregation, dependency inversion.

И сейчас мы поговорим про первый принцип: SRP – Single Responsibility Principle. Он нам говорит о том, что каждый класс должен иметь 1 ответственность. Посмотрите на наш класс активити. Мы в нем обрабатываем ввод от юзера (проверки валидности), также запускаем прогресс и вся логика сохранения и восстановления тоже в нем. Именно поэтому у нас более 100 линий кода. Исходя из принципа SRP наша активити должна иметь лишь 1 ответственность: отвечать за отображение, потому что все вью лежат именно в ней. Принимать события от юзера (ввел текст, нажал кнопку) она может, но не обрабатывать. Отображать данные да. Должна и может. Но никак не обработка логики. Что же делать? Нужно разделять ответственности. И в этом нам поможет патерн MVVM. Что это такое?

Model-View-ViewModel Согласен, название не самое удачное. Но суть довольно проста – мы разделяем наш код на 3 условные группы/слоя. Model – модель данных, где храним данные (типа база данных или как получить их из сети и т.д.). Там чисто классы с методами получения и отправки данных. Ничего лишнего. Далее у нас View – вью, в случае андроид это активити (и иже с ними, фрагменты и т.д.), класс где одна ответственность – показать что-либо и получить от юзера события в виде взаимодействия с вью. Ну и самый главный слой – ViewModel – это тот мост если хотите, который связывает одно с другим – данные с отображением.

Итого, для вью данные непонятно откуда и как приходят, а для модели неважно куда они уйдут. Их связывает вьюмодель (далее вм), она знает откуда брать данные и куда отправлять.

Что это дает? Во-первых ваша активити не будет на 100500 линий кода. Во-вторых код проще читать и поддерживать когда вы явно знаете что он делает. Рекомендую к прочтению “Чистый код” Роберта Мартина. Один из популярных принципов KISS – keep it short and

simple. Сложно назвать класс простым и коротким, если в нем более 250 линий кода и у него более 5 методов. Ок, давайте тогда напишем нашу вм-ку. Но мы для простоты возьмем другой кейс – пусть у нас мейнактивити с 1 текстовкой на ней и мы на запуске просто каждую секунду будем увеличивать число.

Для начала напишем нашу вьюмодель

```
class ViewModel(private val textObservable: TextObservable) {  
    private val model = Model(object : TextCallback {  
        override fun updateText(str: String) {  
            textObservable.postValue(str)  
        }  
    })  
  
    fun init() {  
        model.start()  
    }  
}  
  
class TextObservable {  
    private lateinit var callback: TextCallback  
  
    fun observe(callback: TextCallback) {  
        this.callback = callback  
    }  
  
    fun postValue(text: String) {  
        callback.updateText(text)  
    }  
}  
  
interface TextCallback {  
    fun updateText(str: String)  
}
```

У нас есть модель внутри вм, когда будем дергать метод init то стартанем модель, которая отдает каждую секунду число на 1 больше чем было до. Модель работает по схеме колбека, Вьюмодель тоже по схеме колбека, поэтому итнерфейс колбека мы переиспользуем в 2

местах. Я написал класс обсервабл (если помните патерн наблюдатель). У него 2 метода – начать наблюдать и постить значения. Наблюдать будем в активности, а постить значения пришедшие из модели внутри вм. Посмотрим на модель.

```
class Model(private val textCallback: TextCallback) {  
  
    private var timer: Timer? = null  
  
    private var count = 0  
  
    fun start() {  
        timer?.cancel()  
        timer = Timer()  
        timer?.scheduleAtFixedRate(object : TimerTask() {  
            override fun run() {  
                count++  
                textCallback.updateText(count.toString())  
            }  
        }, delay: 1000, period: 1000)  
    }  
}
```

Здесь простой таймер который стартет по вызову метода и каждую секунду отдает в колбек число как строку. Мне на самом деле было лень писать колбек для числа, потому что модель должна отдавать число и лишь внутри вм должно происходить преобразование к строке. Но об этом позже.

```
super.onCreate(savedInstanceState)  
setContentView(R.layout.activity_main)  
val textView = findViewById<TextView>(R.id.textView)  
val observable = TextObservable()  
observable.observe(object : TextCallback {  
    override fun updateText(str: String) = runOnUiThread {  
        textView.text = str  
    }  
})  
val viewModel = ViewModel(observable)  
viewModel.init()  
}
```

Итак, модель каждую секунду увеличивает число и отдает дальше кому нужно. Дальше у нас `viewModel` получает число и постит в наблюдателя. А наблюдатель у нас в активности. Давайте запустим код и посмотрим что будет.

34

Да, идет счетчик, все ок. Но! Давайте перевернем устройство!

5

Счет пошел сначала! Почему же? А почему нет? Мы создаем нашу `viewModel` в методе `onCreate` и когда поворачиваем наш экран то и `viewModel` создается заново. Значит переменная в `viewModel` модель тоже создается заново и переменная счета тоже обнуляется. Как это пофиксить? Здесь или нужно сохранять значение из активности, передавая его в `viewModel` и потом в модель или же что проще – юзать синглтон! Помните я рассказывал про паттерн синглтон? Давайте заменим `class Model` на `object Model`

```
object Model {  
  
    private lateinit var textCallback: TextCallback  
    private var timer: Timer? = null  
  
    private var count = 0  
  
    fun init(callback: TextCallback) {  
        textCallback = callback  
    }  
  
    fun start() {
```

И придется немного поменять еще и `viewModel`

```

class ViewModel(private val textObservable: TextObservable) {

    init {
        Model.init(object : TextCallback {
            override fun updateText(str: String) {
                textObservable.postValue(str)
            }
        })
    }

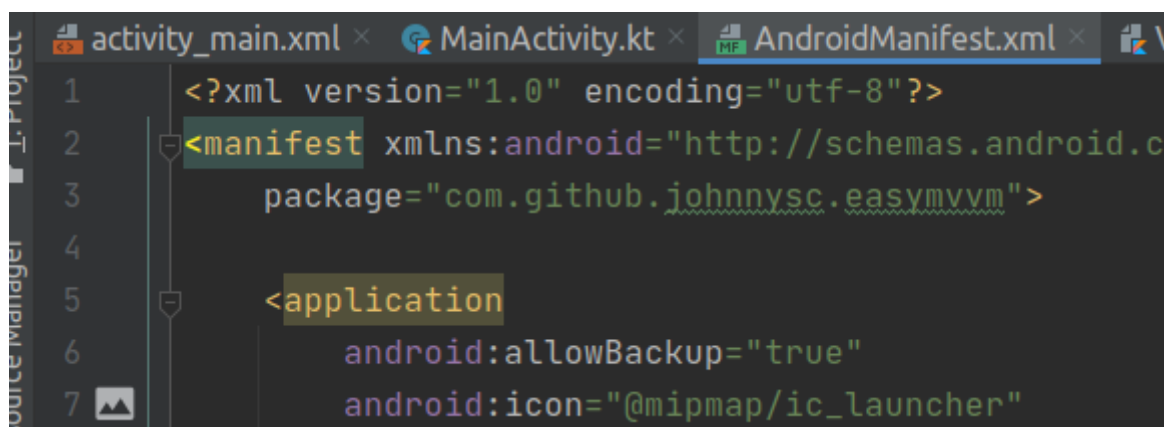
    fun init() {
        Model.start()
    }
}

```

Теперь запустим код! И перевернем спустя пару секунд.

Как видите все работает! Но у нас одна проблема: синглтоны. Мы говорили что в андроид по крайней мере они не нужны. Почему? А вот почему.

В андроид активности пересоздаются в рамках приложения. Ок, поняли. Но если перед смертью активности мы можем что-то сохранить и потом восстановить оттуда, значит это что-то где-то лежит? Скорей всего в синглтоне. Так? Скорей всего. Помните андроид манифест? Там у нас главный тег был application



```

1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3      package="com.github.johnnysc.easymvvm">
4
5      <application
6          android:allowBackup="true"
7          android:icon="@mipmap/ic_launcher"

```

И если активности может быть несколько (или вовсе не быть, да, андроид приложение без активности вовсе), то application у нас один. Это естественно. И если активности умирает раньше времени, то приложение само не умирает пока нет необходимости, пока юзер не вышел из него нажав назад на главной активности или не смахнул в диспетчере задач или система в фоне не убила процесс. Хорошо, поняли: есть синглтон апликейшна. Как мы теперь можем не делать синглтон свой для модели? Легко и просто: Нам даже вьюмодель не нужно будет создавать второй раз. Для этого нам надо написать свой класс и наследоваться от Application.

Но нам нужно немного переписать класс модели чтобы он не принимал аргумент в конструктор. И поправить `view` чтобы он в конструктор принимал именно модель (об этом поговорим когда пройдем другие принципы SOLID).

```
class Model {  
  
    private var timer: Timer? = null  
    private val timerTask = object : TimerTask() {  
        override fun run() {  
            count++  
            callback?.updateText(count.toString())  
        }  
    }  
    private var callback: TextCallback? = null  
    private var count = 0  
  
    fun start(textCallback: TextCallback) {  
        callback = textCallback  
        if (timer == null) {  
            timer = Timer()  
            timer?.scheduleAtFixedRate(timerTask, delay: 1000, period: 1000)  
        }  
    }  
}
```

Мы не будем создавать таймер если он уже есть и не будем отменять его и заново запускать. Так что на второй вызов метода `start` у нас уже будет таймер и значение счета не будет стерто потому как модель будет находиться в `view` которая будет храниться в приложении.

Итак, посмотрим на изменения в `view`. Так как мы работаем с анонимными классами, то получается что если из активности мы сетим аноним класс в `view`, то по сути это равно тому, что мы отдаем активности в `view`: и это называется утечкой, потому как система андроид должна убить активности, а она не может, потому что у нас `view` хранит ссылку на него и `view` лежит в `application`. Чтобы такого не было мы на старте инициализируем в `view` колбек и убираем его в `onDestroy` чтобы после него и до создания новой активности у нас не было утечки памяти.

```

class ViewModel(private val model: Model) {

    private var textObservable: TextObservable? = null

    private val textCallback = object : TextCallback {
        override fun updateText(str: String) {
            textObservable?.postValue(str)
        }
    }

    fun init(textObservable: TextObservable) {
        this.textObservable = textObservable
        model.start(textCallback)
    }

    fun clear() {
        textObservable = null
    }
}

```

Итак, наш класс для доступа к синглтону

```

class MyApplication : Application() {

    lateinit var viewModel: ViewModel

    override fun onCreate() {
        super.onCreate()
        viewModel = ViewModel(Model())
    }
}

```

Чтобы он заработал нужно его прописать в манифесте

```

<application
    android:allowBackup="true"
    android:name=".MyApplication"
    android:icon="@mipmap/ic_launcher"

```

И теперь мы можем получить нашу вм из активности через аппликейшн

```
class MainActivity : AppCompatActivity() {  
  
    private lateinit var viewModel: ViewModel  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
        viewModel = (application as MyApplication).viewModel  
        val textView = findViewById<TextView>(R.id.textView)  
        val observable = TextObservable()  
        observable.observe(object : TextCallback {  
            override fun updateText(str: String) = runOnUiThread {  
                textView.text = str  
            }  
        })  
        viewModel.init(observable)  
    }  
  
    override fun onDestroy() {  
        viewModel.clear()  
        super.onDestroy()  
    }  
}
```

Итак, при создании активности мы берем нашу вм из аппликейшн класса и просто юзаем его

Вы можете залогировать конструктор вм или onCreate в классе MyApplication и увидите что он вызывается лишь единожды. Но проще всего проверить что все правильно просто повернув экран.

Давайте еще раз проговорим: активности умирает и рождается когда поворачивается экран, а Application класс нет. Он живет пока мы не убьем процесс. Значит в нем можно хранить вм. Но чтобы не было утечек нам не стоит хранить в вм ссылку на активности посредством обсервабла в котором оно есть. Поэтому перед смертью активности мы очищаем вм и при создании опять сетим обсервабл. Чтобы проверить утечку сначала создайте ее – не очищайте вм и подключите к проекту LeakCanary. Она вам покажет где вы создали утечку. Учитывая что активности это тяжелый класс хранение инстанса в аппликейшне это огромная проблема в вашем хеловорлд. Поэтому будьте осторожны когда пишете код и в идеале не создавайте анонимных классов, но если без них никак : просто очищайте их когда уходите с активности.

Для большей уверенности можете очищать и ссылку в модели.