LiveData

Упрощаем работу View+ViewModel

Содержание

1. Убираем колбек из вьюмодели

Помните в 11 лекции мы убирали колбеки и заменяли их корутинами? Мы избавились от 4 колбеков и оставили 1 который использовали для отображения данных в активити. Теперь же удалим и его. Сначала я покажу ливдаты, а после обьясню что они делают.

Итак, давайте отредактируем нашу ВМ и все же переименуем ее в MainViewModel

Сначала я покажу как делают многие и обьясню почему это неправильно, после покажу как стоит делать. Сначала уберем метод мап из юай модели

```
abstract class JokeUiModel(private val text: String, private val punchline: String) {
    private fun text() = "$text\n$punchline"

    @DrawableRes
    protected abstract fun getIconResId(): Int

    fun getData() = Pair(text(), getIconResId())
}
```

Теперь можем редактировать вьюмодель

```
class BaseViewModel(private val model: Model) : ViewModel() {

    val liveData = MutableLiveData<Pair<String, Int>>()

    fun changeJokeStatus() = viewModelScope.launch {  this: CoroutineScope
        model.changeJokeStatus()?.let {  it: JokeUiModel
            liveData.value = it.getData()
        }
    }

    fun getJoke() = viewModelScope.launch {  this: CoroutineScope
        liveData.value = model.getJoke().getData()
    }

    fun chooseFavorites(favorites: Boolean) = model.chooseDataSource(favorites)
}
```

Видите как чисто стало? Нет никакого метода инит и никакой зачистки колбека тоже нет.

У нас появилось новое поле liveData Мутабельная ливдата по типу пары строка и число просто потому что наша юаймодель отдает пару строка и число. Да, зачем нам 2 отдельных метода если у нас одновременно меняются текст и иконка. Ладно, что такое ливдата я обещал рассказать позже. Давайте теперь посмотрим на то, как будет использоваться ливдата.

```
viewModel.liveData.observe( owner: this, { (text, drawableResId) ->
    button.isEnabled = true
    progressBar.visibility = View.INVISIBLE
    textView.text = text
    changeButton.setImageResource(drawableResId)
})
```

Итак, мы берем ливдату в активити и обсервим ее. Что-то очень знакомое, неправда-ли?

Итак, обсервим в активити и в лямбде получаем пару и спасибо котлину можно назвать части нормальным образом вместо it.first, it.second у нас будут нормальные названия какие мы захотим дать. И вот так вот красиво у нас выглядит теперь активити. Запустим код? Все ОК.

И давайте просто посмотрим на то что мы сделали – у нас в вьюмодели паблик поле! И мало того оно инициализируется прямо там же! Сколько принципов мы нарушили? Можем ли мы написать юнит тест на вьюмодель? Конечно же нет. Мы не заменим ливдату никак. Именно поэтому я не рекомендую так делать никому никогда и ни при каких обстоятельствах.

Принципы ООП нельзя нарушать даже ради кажущейся красоты вашего кода

2. Как правильно использовать ливдату чтобы тестировать вьюмодель

Для начала я дам вам 5 минут на подумать. Еще раз – мы нарушили инкапсуляцию это раз. У нас поле класса доступно снаружи. Мы нарушили принципы SOLID L & D нельзя ни заменить ливдату ни получить снаружи подстановочный инстанс для тестов. Значит нам нужно во-первых иметь некое приватное поле которое получим в конструкторе и оно должно быть интерфейсом. Ладно, давайте попробуем сделать это. Для начала интерфейс

```
interface Communication {

    fun showData(data: Pair<String, Int>)

    fun observe(owner: LifecycleOwner, observer: Observer<Pair<String, Int>>)
}
```

Почему-то мне кажется что так называть нормально, у интерфейса 2 метода. В одном методе мы покажем данные, через второй будем наблюдать. У нас же патерн наблюдатель все же.

Да, как видите учить патерны полезно, они используется в гугловых ливдатах.

Итак, напишем базовую реализацию с ливдатой которая инкапсулирована внутри

В принципе можно в конструктор получить саму ливдату, а не инициализировать внутри

```
class BaseCommunication : Communication {
    private val liveData = MutableLiveData<Pair<String, Int>>()

    override fun showData(data: Pair<String, Int>) {
        liveData.value = data
    }

    override fun observe(owner: LifecycleOwner, observer: Observer<Pair<String, Int>>) {
        liveData.observe(owner, observer)
    }
}
```

И здесь просто 2 метода интерфейса вызывают аналогичные у ливдаты. Кстати, заметьте, ливдата мутабельная, это как мутабельный список и имутабельный. В мутабельную ливдату можно сетить значение, а в немутабельную ливдату нельзя, можно только обсервить.

Итак, попробуем исправить теперь вьюмодель

```
class BaseViewModel(
    private val model: Model,
    private val communication: Communication
) : ViewModel() {

    fun changeJokeStatus() = viewModelScope.launch {  this: CoroutineScope
        model.changeJokeStatus()?.let {  it: JokeUiModel
            communication.showData(it.getData())
        }
    }

    fun getJoke() = viewModelScope.launch {  this: CoroutineScope
        communication.showData(model.getJoke().getData())
    }

    fun chooseFavorites(favorites: Boolean) = model.chooseDataSource(favorites)

    fun observe(owner: LifecycleOwner, observer: Observer<Pair<String, Int>>) =
        communication.observe(owner, observer)

}
```

Там где раньше просто вызывали метод у ливдаты теперь вызываем метод у интерфейса. Все просто. Теперь пофиксим наш апликейшн класс

```
viewModel = BaseViewModel(
    BaseModel(
        model dependencies
    BaseCommunication()
)
```

И пофиксим нашу активити в конце концов

```
viewModel.observe( owner: this, { (text, drawableResId) ->
    button.isEnabled = true
    progressBar.visibility = View.INVISIBLE
    textView.text = text
    changeButton.setImageResource(drawableResId)
})
```

И это самое простое. Заметьте, раньше было viewModel.liveData.observe, теперь же сразу метод у вмки. Так намного правильнее. Никто не должен знать о полях класса! Никогда!

Запустим код и проверим. Конечно же все работает. Теперь можем написать юнит тест на вьюмодель. Поверьте мне, во многих проектах нет никаких юнит тестов на вьюмодели. Потому что там нарушены все возможные и невозможные принципы. И да, еще один момент. У вьюмодели от гугла есть наследник, не используйте его никогда. Вы не сможете в конструктор дать другую реализацию апликейшн класса

```
public class AndroidViewModel extends ViewModel {
    @SuppressLint("StaticFieldLeak")
    private Application mApplication;

    public AndroidViewModel(@NonNull Application application) {
        mApplication = application;
    }
```

Просто потому что апликейшн класс не интерфейс!

```
public class Application extends ContextWrapper implements ComponentCallbacks2 {
```

Вернемся к юнит тестам. Создадим один Alt+Enter и пакет test

Но для того чтобы писать юнит тест с использованием корутин нам нужна либа

```
testImplementation 'org.jetbrains.kotlinx:kotlinx-coroutines-test:1.5.0'
```

Проблема в том, что нельзя тестировать код который был написан viewModelScope.launch и потому нам придется еще внести правки в сам класс вьюмодели

```
class BaseViewModel(
    private val model: Model,
    private val communication: Communication,
    private val dispatcher: CoroutineDispatcher = Dispatchers.Main
) : ViewModel() {

    fun changeJokeStatus() = viewModelScope.launch(dispatcher) {...}

    fun getJoke() = viewModelScope.launch(dispatcher) {...}
```

Мы передадим в конструктор диспетчер – на каком потоке нам нужно будет наблюдать за результатом. Для тестов мы заменим его на тестовый диспетчер чтобы тест заработал.

Теперь спокойно пишем код для теста

```
class BaseViewModelTest {

    @ExperimentalCoroutinesApi
    @Test
    fun test_get_joke_from_cloud_success(): Unit = runBlocking { this: CoroutineScope
        val model = TestModel()
        val communication = TestCommunication()
        val viewModel = BaseViewModel(model, communication, TestCoroutineDispatcher())

        model.success = true
        viewModel.chooseFavorites( favorites: false)
        viewModel.getJoke()

        val actualText = communication.text
        val actualId = communication.id
        val expectedText = "cloudJokeText\ncloudJokePunchline"
        assertEquals(expectedText, actualText)
        assertNotEquals( unexpected: 0, actualId)
    }
```

Итак, самый первый тестовый метод и самый простой – проверить что при выборе источника сеть и при успешном ответе у нас будет вызываться метод у интерфейса communication с данными где текст это конкатенация текста и панчлайна, а айди не ноль. Если помните, то для успешного ответа мы не передавали ноль для айди изображения. Сам тест довольно простой: создаем тестмодель, тесткоммуникатор и вьюмодель инициализируем тестовым диспетчером. Далее заставляем модель вернуть нам успешный ответ и берем данные из сети. Вот и все. Берем шутку и проверяем текст и айди. Все просто, а теперь посмотрим как выглядят тестовые классы наших зависимостей.

```
private inner class TestModel : Model {

    private val cacheJokeUiModel = BaseJokeUiModel( text: "cachedJokeText",  punchline: "cachedJokePunchline")
    private val cacheJokeFailure = FailedJokeUiModel( text: "cacheFailed")
    private val cloudJokeUiModel = BaseJokeUiModel( text: "cloudJokeText",  punchline: "cloudJokePunchline")
    private val cloudJokeFailure = FailedJokeUiModel( text: "no connection")
    var success: Boolean = false
    private var getFromCache = false
    private var cachedJoke: JokeUiModel? = null

    override suspend fun getJoke(): JokeUiModel {...}

    override suspend fun changeJokeStatus(): JokeUiModel? {...}

    override fun chooseDataSource(cached: Boolean) {
        getFromCache = cached
    }
}
```

Я создал 4 ответа для всех случаев жизни – успех или провал, сеть или кеш. Также я даю легко получить успех или ошибку через переменную, все остальное такое же как в базовой модели. И я покажу вам метод получения шутки в тестовой модели

```
override suspend fun getJoke(): JokeUiModel {
    return if (success) {
        if (getFromCache) {
            cacheJokeUiModel.also { it: BaseJokeUiModel
                cachedJoke = it
            }
        } else {
            cloudJokeUiModel.also { it: BaseJokeUiModel
                cachedJoke = it
            }
        }
    } else {
        cachedJoke = null
        if (getFromCache) {
            cacheJokeFailure
        } else {
            cloudJokeFailure
        }
    }
}
```

Я все так же кеширую данные если был успех и зануляю если моя тестовая модель ответила ошибкой. Теперь посмотрим на тестовый коммуникатор

```
private inner class TestCommunication : Communication {
    var text = ""
    var id = -1
    var observedCount = 0
    override fun showData(data: Pair<String, Int>) {
        text = data.first
        id = data.second
    }
    override fun observe(owner: LifecycleOwner, observer: Observer<Pair<String, Int>>) {
        observedCount++
    }
}
```

Здесь все просто – так как я получаю айди сердечек то я не знаю их числовой эквивалент и буду просто сравнивать с нулем поле в классе. Ноль для ошибок и не ноль для успеха. Вот так вот. Теперь давайте напишем кейс, когда от сети пришел неуспешный ответ

```
@ExperimentalCoroutinesApi
@Test
fun test_get_joke_from_cloud_fail(): Unit = runBlocking { this: CoroutineScope
    val model = TestModel()
    val communication = TestCommunication()
    val viewModel = BaseViewModel(model, communication, TestCoroutineDispatcher())

    model.success = false
    viewModel.chooseFavorites( favorites: false)
    viewModel.getJoke()

    val actualText = communication.text
    val actualId = communication.id
    val expectedText = "no connection\n"
    val expectedId = 0
    assertEquals(expectedText, actualText)
    assertEquals(expectedId, actualId)
}
```

Все то же самое. Я в тестовом классе возвращаю ошибку что нет интернета и потому проверка текста в конце перенос строки, ведь мы конкатенировали текст и панчлайн через перенос. Ну и айди у ошибочных данных ноль, потому проверка на ноль прошла.

Думаю остальные тесты вы сможете написать сами. Если нет, то дождитесь лекции про юай тесты, там подробно остановимся на том, как писать тесткейсы чтобы проверить все что можно и нельзя.

4. Суть ливдаты

Итак, казалось бы, ливдата простой патерн наблюдатель. В одном месте наблюдаем (в активити) и в другом месте сетим значения (в комуникаторе, в вьюмодели), но гугл бы не создал целую либу просто переименовав патерн наблюдатель. Кроме того что мы доставляем данные патерном наблюдатель ливдата еще умеет хранить значение. Да-да, речь про поворот экрана. Помните мы клали в bundle и забирали оттуда данные? Все это автоматом умеет наша ливдата. Как это проверить? Если наша ливдата имеет метод наблюдения и она хранит последнее значение которые мы засетили в нее, то по логике должно повторно сработать там где обсервим. Давайте поставим лог и посмотрим

```
viewModel.observe( owner: this, { (text, drawableResId) ->
    Log.d( tag: "mainActivityUniqueTag",  msg: "observe method call $text")
```

Теперь запустим код, получим шутку и повернем экран. Посмотрим логи

```
2021-06-18 19:37:16.212 10838-10838/com.github.johnnysc.jokeapp D/mainActivityUniqueTag: observe method call What do you call a factory that sells passable products?
    A satisfactory
2021-06-18 19:37:31.116 10838-10838/com.github.johnnysc.jokeapp D/mainActivityUniqueTag: observe method call What do you call a factory that sells passable products?
    A satisfactory
```

Почему же так? Сейчас обьясню. Вся суть ливдаты в том, что она во-первых хранит данные в себе и отображает в активити лишь тогда, когда у нее вызвался метод onResume. Если ваша модель отдает в ливдату данные, но ваш юзер не смотрит в приложение, то ливдата хранит данные до момента когда юзер вернется на экран. Как это проверить? Легко. Давайте

добавим задержку в 5 секунд перед тем как метод отдает шутку с сервера и скроем приложение после нажатия на кнопку и посмотрим логи. Там должно быть пусто.

```
class BaseCloudDataSource(private val service: JokeService) : CloudDataSource {

    override suspend fun getJoke(): Result<JokeServerModel, ErrorType> {
        return try {
            Thread.sleep( millis: 5000)
            val result: JokeServerModel = service.getJoke().execute().body()!!
```

Также залогируем момент нажатия на кнопку, onResume и onPause.

Запустим проект теперь, нажмем кнопку и свернем. После подождем 10 секунд и вернемся.

```
2021-06-18 19:43:15.931 11157-11157/com.github.johnnysc.jokeapp D/mainActivityUniqueTag: button clicked
2021-06-18 19:43:17.096 11157-11157/com.github.johnnysc.jokeapp D/mainActivityUniqueTag: onPause
2021-06-18 19:43:40.376 11157-11157/com.github.johnnysc.jokeapp D/mainActivityUniqueTag: observe method call I g
    It was a soft drink.
2021-06-18 19:43:40.404 11157-11157/com.github.johnnysc.jokeapp D/mainActivityUniqueTag: onResume
```

Нажали на кнопку в 19.43.15. Свернули через 2 секунды. Вернулись и метод сработал после лишь когда юзер увидел экран. Но перед onResume. Дело в том, что есть еще один метод onStart и сначала срабатывает он, потом уже onResume. На методе onStart юзер уже видит экран, но на методе onResume уже может взаимодействовать с экраном. Этот метод нужен если у вас диалоговое окно перекрыло вашу активити и вы не можете взаимодесйтвовать с экраном пока не уберете диалоговое окно.

```
2021-06-18 19:46:44.708 11460-11460/com.github.johnnysc.jokeapp D/mainActivityUniqueTag: button clicked
2021-06-18 19:46:45.904 11460-11460/com.github.johnnysc.jokeapp D/mainActivityUniqueTag: onPause
2021-06-18 19:46:58.292 11460-11460/com.github.johnnysc.jokeapp D/mainActivityUniqueTag: onStart
2021-06-18 19:46:58.292 11460-11460/com.github.johnnysc.jokeapp D/mainActivityUniqueTag: observe method call Why
    Fur protection.
```

И здесь мы сделаем важную паузу. Все другие библиотеки для работы не поддерживают этот момент. Они асинхронно получают данные и отдают на юай даже тогда, когда он не готов обработать данные. Именно поэтому я вам рассказываю про ливдату и наш колбек старый не лучшее решение. Кстати, если посмотрите исходники ливдаты то увидете настоящую работу с данными пришедшими из другого потока, ведь кроме метода setValue есть и postValue где переключается поток на мейн. Но нам это не нужно, ведь мы запускаем корутины в вьюмоделскоуп, а всю асинхронную работу делаем на другом потоке, например ввода вывода.

Но с использованием ливдаты могут быть проблемы, когда вы например засетили в ливдату данные и при повороте они снова сработали, а вы не ожидали такого. Предположим вы через ливдату показываете диалог об ошибке. Значит когда юзер повернет экран то увидит диалоговое окно еще раз. Что же делать спросите меня вы. Использовать специальный класс SingleLiveEvent (погуглите сами). Но в общем случае у вас немного неправильная логика.

Добавьте после логирования код для отображения Toast и поверните экран (не забудьте убрать Thread.sleep(5000) из класса датасорса). И вы увидете что тоаст срабатывает каждый раз при повороте. Если для вашего случая это нормально, то ок. Но если вы считаете что юзер должен увидеть его лишь 1 раз и не надо при каждом повороте показывать, то используйте класс события единоразового отображения.

И на последок я бы хотел поменять код для комуникатора, он получает от шутки данные напрямую. Давайте исправим это и не будем показывать всем и каждому. А лишь комуникатору.

```
abstract class JokeUiModel(private val text: String, private val punchline: String) {
    private fun text() = "$text\n$punchline"

    @DrawableRes
    protected abstract fun getIconResId(): Int

    fun show(communication: Communication) = communication.showData(Pair(text(), getIconResId()))
}
```

И теперь наш код в вьюмодели станет еще прекраснее

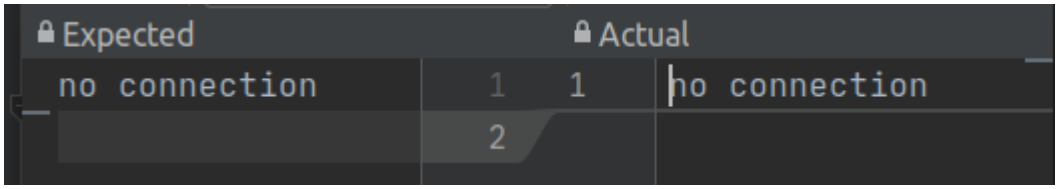
```
fun getJoke() = viewModelScope.launch(dispatcher) {  this: CoroutineScope
    model.getJoke().show(communication)
}
```

И если вас смущает класс Pair то передавайте сразу нужные обьекты. Кстати, если вас смущает то, что при ошибке все равно конкатенируется пустая строка, то можно пофиксить это сделав метод text protected open.

```
class FailedJokeUiModel(private val text: String) : JokeUiModel(text,  punchline: "") {
    override fun text() = text
    override fun getIconResId() = 0
}

abstract class JokeUiModel(private val text: String, private val punchline: String) {
    protected open fun text() = "$text\n$punchline"
```

Теперь запустите юнит тест вьюмодели и увидете что он не прошел



Ожидался перенос строки, а по факту не было. Можете пофиксить юнит тест теперь

```
val expectedText = "no connection"
val expectedId = 0
assertEquals(expectedText, actualText)
assertEquals(expectedId, actualId)
```

Запомните простую вещь – если вас не устраивает до конца то, что вы написали то найдите способ сделать лучше, не миритесь с плохим кодом даже в таких мелочах.

И еще мне не нравится наш код в активити. В нем слишком много логики, вам не кажется?

```
button.setOnClickListener {  it: View!
    button.isEnabled = false
    progressBar.visibility = View.VISIBLE
    viewModel.getJoke()
}
viewModel.observe( owner: this, { (text, drawableResId) ->
    button.isEnabled = true
    progressBar.visibility = View.INVISIBLE
    textView.text = text
    changeButton.setImageResource(drawableResId)
})
```

Все же понимают что так не должно быть. Мы меняем доступность кнопки в 2 местах и меняем видимость прогреса в 2 местах. Здесь есть 2 решения. Первое сделать 2 метода у комуникатора просто чтобы обсервить ливдаты для кнопки и для прогресса, но мне это не нравится. Я за второе решение: сделать класс состояния экрана с 2 подклассами. Давайте покажу как

```
sealed class State {
    object Progress : State()
    data class Initial(val text: String, @DrawableRes val id: Int)
}
```

И далее используем перепишем наш комуникатор

```
interface Communication {

    fun showState(state: BaseViewModel.State)

    fun observe(owner: LifecycleOwner, observer: Observer<BaseViewModel.State>)
}

class BaseCommunication : Communication {
    private val liveData = MutableLiveData<BaseViewModel.State>()

    override fun showState(state: BaseViewModel.State) {
        liveData.value = state
    }

    override fun observe(owner: LifecycleOwner, observer: Observer<BaseViewModel.State>) {
        liveData.observe(owner, observer)
    }
}
```

Да, я положил стейт внутрь вьюмодели, поэтому выглядит это вот так. Теперь поменяем стейт в вьюмодели перед тем как шутку получить

```
fun getJoke() = viewModelScope.launch(dispatcher) {
    communication.showState(State.Progress)
    model.getJoke().show(communication)
}
```

Кстати, если вы ранее замечали, то когда мы меняем статус у шутки на мгновение отображается прогрес и скрывается, просто потому что код был написан в активити. Теперь же прогресс будет отображаться по логике из вьюмодели только тогда, когда нужно. В одном месте. И давайте уже посмотрим как переписать код в активити.

```
button.setOnClickListener { it: View!
    viewModel.getJoke()
}
viewModel.observe( owner: this, { state ->
    when (state) {
        BaseViewModel.State.Progress -> {
            button.isEnabled = false
            progressBar.visibility = View.INVISIBLE
        }
        is BaseViewModel.State.Initial -> {
            button.isEnabled = true
            progressBar.visibility = View.INVISIBLE
            textView.text = state.text
            changeButton.setImageResource(state.id)
        }
    }
})
```

Выглядит не лучше, правда? Мы можем поступить иначе, сделать все же 3 ливдаты. Но у меня есть совсем другое решение. Давайте напишем код таким образом, чтобы в активити не было логики when if else и вот это вот все.

Кстати, надо пофиксить еще метод в JokeUiModel

```
fun show(communication: Communication) = communication.showState(
    BaseViewModel.State.Initial(text(), getIconResId())
)
```

Теперь вернемся к силд классу стейта и добавим туда абстрактный метод, посмотрим что у нас выйдет

```
sealed class State {
    abstract fun show(
        progress: View,
        button: Button,
        textView: TextView,
        imageButton: ImageButton
    )
```

Проблема в том, что наш класс стейта знает о вью классах нашей активити. Но пока что это не самое большое зло

```
object Progress : State() {
    override fun show(
        progress: View,
        button: Button,
        textView: TextView,
        imageButton: ImageButton
    ) {
        progress.visibility = View.VISIBLE
        button.isEnabled = false
    }
}
```

Мы в классе прогресса имеем доступ к другим вью, но не используем их, создали проблему на ровном месте, но давайте посмотрим на второй класс

```
class Initial(private val text: String, @DrawableRes private val id: Int) : State() {
    override fun show(
        progress: View,
        button: Button,
        textView: TextView,
        imageButton: ImageButton
    ) {
        progress.visibility = View.INVISIBLE
        button.isEnabled = true
        textView.text = text
        imageButton.setImageResource(id)
    }
```

Здесь у нас вернулась инкапсуляция и здесь мы используем все аргументы метода. И кто-то спросит, а зачем это все? А теперь посмотрите на эту красоту в активити.

Мы убрали when для силд класса из активити и переписали наш стейт класс чтобы он сам все умел делать внутри себя. Давайте запустим проект и проверим что все работает исправно.

```
viewModel.observe( owner: this, { state ->
    state.show(progressBar, button, textView, changeButton)
})
```

Да, все работает. Итак, в чем же проблемы: в том, что наш класс стейта знает о наших вьюхах слишком много. Но по сути это не так плохо как кажется, неправда-ли? Мы можем решить эту проблему 2 способами : не использовать метод в силд классе и вернуться к 3 ливдатам под каждый обьект : кнопка, прогрес и текст с иконкой. Или же второй способ использовать интерфейсы. Ведь наши классы кнопки или текствью не являются интерфейсами.

Но мы можем исправить это вот таким вот способом! Смотрите сами

```
interface ShowText {
    fun show(text: String)
}

interface ShowImage {
    fun show(@DrawableRes id: Int)
}

interface ShowView {
    fun show(show: Boolean)
}

interface EnableView {
    fun enable(enable: Boolean)
}
```

Выглядит круто, но наши классы вьюх все равно не имплементируют их. Но давайте покажу как использовать

```
sealed class State {
    abstract fun show(
        progress: ShowView,
        button: EnableView,
        textView: ShowText,
        imageButton: ShowImage
    )
```

Сначала поменяем типы аргументов в главном методе стейта и после уже в реализациях

Посмотрите как круто выглядит, как будто мы и не андроид приложение пишем

```
object Progress : State() {
    override fun show(
        progress: ShowView,
        button: EnableView,
        textView: ShowText,
        imageButton: ShowImage
    ) {
        progress.show( show: true)
        button.enable( enable: false)
    }
}
```

Метод получает прогресс и отображает его, получает кнопку которую можно лишь дизейблить и вуаля. Проблема лишь в том, что у нас 2 лишних аргумента. Давайте дальше

```
class Initial(private val text: String, @DrawableRes private val id: Int) : State() {
    override fun show(
        progress: ShowView,
        button: EnableView,
        textView: ShowText,
        imageButton: ShowImage
    ) {
        progress.show( show: false)
        button.enable( enable: true)
        textView.show(text)
        imageButton.show(id)
    }
}
```

Ладно, как же теперь будет выглядеть наш активити код? Давайте попробуем пофиксить.

(картинка на следующей странице)

Но что-то очень ужасно стало, неправда-ли? Вся красота пропала. Может поменять на лямбды? Было бы короче конечно, но все равно не то. Эх, как же так? Наши классы из андроид сдк не имплементируют интерфейсы. Ладно, но ведь они могут наследоваться!

Посмотрите на исходники например TextView

```
public class TextView extends View implements ViewTreeObserver.OnPreDrawListener {
```

Если помните если класс не помечен ключевым словом final значит его можно наследовать.

Наследоваться от андроид вью класса? Звучит страшно, но давайте попробуем это сделать

4. Кастомвью, введение

```
viewModel.observe(
    owner: this, { state ->
        state.show(
            object : ShowView {
                override fun show(show: Boolean) {
                    progressBar.visibility = if (show) View.VISIBLE else View.INVISIBLE
                }
            },
            object : EnableView {
                override fun enable(enable: Boolean) {
                    button.isEnabled = enable
                }
            },
            object : ShowText {
                override fun show(text: String) {
                    textView.text = text
                }
            },
            object : ShowImage {
                override fun show(id: Int) {
                    changeButton.setImageResource(id)
                }
            }
        )
    })
```

Для начала создадим класс наследник текствью и назовем его CorrectTextView

```
class CorrectTextView : androidx.appcompat.widget.AppCompatTextView, ShowText {
    constructor(context: Context) : super(context)
    constructor(context: Context, attrs: AttributeSet) : super(context, attrs)
    constructor(context: Context, attrs: AttributeSet, defStyleAttr: Int) : super(
        context,
        attrs,
        defStyleAttr
    )
    override fun show(text: String) {
        setText(text)
    }
}
```

При наследовании просто от TextView АС указало что так неправильно и надо наследоваться от другого класса. Ок. А что за 3 конструктора? Суть в том, что мы можем создать текствью не только в хмл, но еще и в коде. Да-да. Писать хмл совсем не нужно. Но намного проще.

Про атрибуты я расскажу позднее – мы получаем в коде доступ к атрибутам (ну, такие как textColor, textSize и другие). Но не об этом сейчас речь. Посмотрите на наш класс : он наследует интерфейс показа текста. Теперь давайте попробуем заменить классом в активити. Для этого нам надо пофиксить разметку хмл. Посмотрите

И теперь нам нужно пофиксить код в активити указав при поиске по айди тип класса

```
<com.github.johnnysc.jokeapp.CorrectTextView
    android:padding="16dp"
    android:id="@+id/textView"
```

Просто меняем 1 класс на наш кастомный

```
val textView = findViewById<CorrectTextView>(R.id.textView)
```

И теперь мы можем передать его прямо в методе show

```
},
textView,
object : ShowImage {
```

Проделайте сами с остальными интерфейсами, я покажу что у меня получилось

```
class CorrectButton : androidx.appcompat.widget.AppCompatButton, EnableView {
    constructors
    override fun enable(enable: Boolean) {
        isEnabled = enable
    }
}
```

Продолжаем с ImageButton

```
class CorrectImageButton : androidx.appcompat.widget.AppCompatImageButton , ShowImage {
    constructors
    override fun show(id: Int) {
        setImageResource(id)
    }
}
```

И для прогресса еще нужно написать класс и в принципе все

```
class CorrectProgress : ProgressBar, ShowView {
    constructors
    override fun show(show: Boolean) {
        visibility = if (show) View.VISIBLE else View.INVISIBLE
    }
}
```

И наконец-то мы сможем вернуть наш метод у вьюмодели к той красоте с чего начали

```
viewModel.observe( owner: this, { state ->
    state.show(progressBar, button, textView, changeButton)
})
```

Но все же мы остались недовольны тем, что конкретные классы стейта принимали слишком много аргументов. Давайте пофиксим это!

Итак, я поменял метод у силд класса таким образом, чтобы наследники определяли лишь то, что им нужно

```
sealed class State {
    fun show(progress: ShowView, button: EnableView, textView: ShowText, imageButton: ShowImage) {
        show(progress, button)
        show(textView, imageButton)
    }

    protected open fun show(progress: ShowView, button: EnableView) {}
    protected open fun show(textView: ShowText, imageButton: ShowImage) {}
```

И теперь посмотрите на наследников, как красиво они выглядят

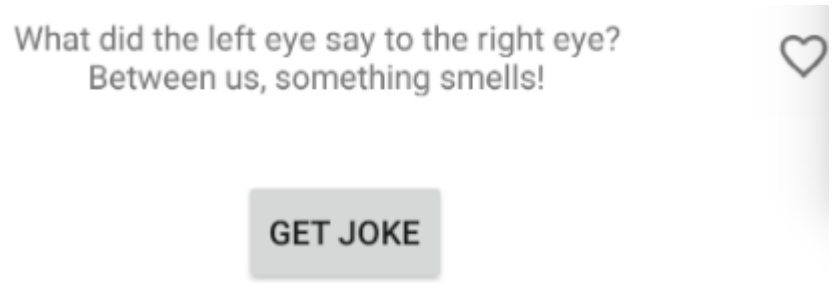
```
object Progress : State() {
    override fun show(progress: ShowView, button: EnableView) {
        progress.show( show: true)
        button.enable( enable: false)
    }
}
```

В классе прогресса только те аргументы которые нужны. А в классе инишл уже оба метода

```
class Initial(private val text: String, @DrawableRes private val id: Int) : State() {
    override fun show(progress: ShowView, button: EnableView) {
        progress.show( show: false)
        button.enable( enable: true)
    }
    override fun show(textView: ShowText, imageButton: ShowImage) {
        textView.show(text)
        imageButton.show(id)
    }
}
```

И давайте все же запустим код. Неужели все работает как и раньше?

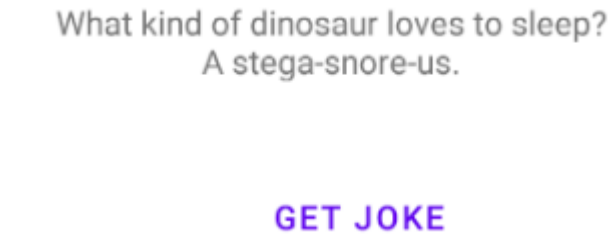
Да, за исключением того, что наша кнопка перестала быть красивой



Но это легко пофиксить на самом деле. Просто поставить стиль какой-нибудь

```
style="@style/Widget.MaterialComponents.Button.OutlinedButton"
```

Вот и все.



Мне жаль что многие классы в андроид сдк и в джава не являются интерфейсами и мы не можем написать наследников сразу, но благо классы вью можно наследовать и писать кастомвью. В последующих лекциях мы подробнее остановимся на теме кастомвью и напишем свой прогресс.
Поверьте мне, наш текущий код далеко не самый правильный и красивый, но мы с каждой лекцией делаем его лучше и лучше. Если вы не хотите иметь кастомные классы поменяйте интерфейсы в методе стейта на лямбды. Вы сэкономите много времени.
И да, напоследок. Может все же сделаем дженериком интерфейс с методом show ?
Например вот так

```
interface Show<T> {
    fun show(arg: T)
}

interface ShowText {
    fun show(text: String)
}
```

И наши интерфейсы преобразятся сразу же. Interface Segregation Principle

```
interface ShowText : Show<String>
interface ShowImage : Show<Int>
interface ShowView: Show<Boolean>
```

Нет ничего лучше чем удалить лишние линии кода.
И на последок скажу пару слов о том, зачем нам использовать кастом классы для текста и кнопок : вы можете сразу в них написать логику, отображение и поведение и использовать во всем проекте одни и те же вью. Например можно всем текстам сразу дать кастом фонт и в хмл использовать нужный класс вместо дефолтного андроид класса и не писать каждый раз лишнюю линию с фонтом. А на счет кнопок: у вас может быть один дизайн экшн кнопки например зеленая с круглыми углами и текстом с уже нужным фонтом и размером. В хмл просто одной линией пишете в теге <ActionButton и вуаля. Готово.

Как задание можете погуглить сами тему кастом вью или ливдаты и попробовать сделать что-нибудь самостоятельно.