ViewModelFactory, IoC, FragmentManager

Подгружаем фичу когда нужно

В предыдущей лекции мы сделали так, что обе фичи были доступны условно сразу. Ты свайпал экран и сразу читал цитаты если был на “экране” шуток. Да, все вроде бы классно, но здесь возникает следующий вопрос : а что если юзер откроет приложение и будет читать только шутки? Да, он видит что есть цитаты, но не имеет желания переходить на этот экран и читать их и тем более сохранять. Или же наоборот, юзер предпочитает только цитаты и не нужно ему каждый раз на старте показывать шутки.

Давайте начнем с того, что будем подгружать нужный фрагмент при тапе на таб и очищать ненужный фрагмент. Если помните, то на данном этапе в апликейшн классе у нас сразу создаются вьюмодели для обеих фичей. Чуть позже я покажу что и как нужно сделать для этого. А пока давайте начнем с передела хмл. Удалим вьюпейджер который писали в предыдущей лекции и сразу в хмл добавим 2 таба в таблейаут и переместим их вниз экрана.

```
<?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="vertical">

   <FrameLayout
      android:id="@+id/container"
      android:layout_width="match_parent"
      android:layout_height="0dp"
      android:layout_weight="1" />

   <com.google.android.material.tabs.TabLayout
      android:id="@+id/tabLayout"
      android:layout_width="match_parent"
      android:layout_height="wrap_content">

      <com.google.android.material.tabs.TabItem
         android:layout_width="match_parent"
         android:layout_height="wrap_content"
         android:text="@string/jokes" />

      <com.google.android.material.tabs.TabItem
         android:layout_width="match_parent"
         android:layout_height="wrap_content"
         android:text="@string/quotes" />
   </com.google.android.material.tabs.TabLayout>

</LinearLayout>
```

Это наш хмл для мейн активити, теперь нам нужно переписать код
Для начала напишем лиснер для выбранного таба. Так как у нас таблейаут, а не просто кнопки, то нам нужно слушать изменения выбранного таба, потому что при выборе одного меняется также состояние и второго (выбранное→ невыбрано).

```
private class TabListener(private val tabChosen: (Boolean) -> Unit) :
    TabLayout.OnTabSelectedListener {
    override fun onTabSelected(tab: TabLayout.Tab?) = tabChosen.invoke(tab?.position == 0)
    override fun onTabUnselected(tab: TabLayout.Tab?) = Unit
    override fun onTabReselected(tab: TabLayout.Tab?) = Unit
}
```

Из трех методов нам нужен лишь первый и то, я бы не хотел использовать низкоуровневый код и потому отдаю в конструкто классу лямбду по которой уже пойму какой таб был выбран.
Теперь можем написать код в методе onCreate у мейн активити

```
val tabLayout = findViewById<TabLayout>(R.id.tabLayout)
val tabChosen: (Boolean) -> Unit = { jokesChosen ->
    if (jokesChosen)
        showJokes()
    else
        showQuotes()
}
tabLayout.addOnTabSelectedListener(TabListener(tabChosen))
```

И это в принципе все что нам нужно. Находим таблейаут и начинаем обрабатывать изменения выбора. Если выбраны шутки, то отобразим экран шуток, иначе – цитаты. Надеюсь пока что все понятно. Посмотрите сейчас еще раз на хмл мейн активити, там кроме самих табов есть еще framelayout – это простой контейнер для фрагментов, которые будем отображать. Отличие от вьюпейджера в том, что у него нет свайпов. Теперь мы сможем переключать экраны лишь по нажатию на табы. И потому собственно мы их и переместили на дно экрана.
Теперь нам нужно написать код, аналогичный PagerAdapter из предыущей лекции. В этой я удалил этот класс за ненадобностью. Он самостоятельно менял фрагменты, а мы теперь должны заняться этим самостоятельно. Для того чтобы менять фрагменты в андроид есть такой класс как FragmentManager. Давайте напишем 1 метод для смены фрагмента

```
private fun show(fragment:Fragment) {
    supportFragmentManager.beginTransaction()
        .replace(R.id.container, fragment)
        .commit()
}
```

Как видите здесь все просто – берем фрагментменеджер у активити который доступен. Начинаем транзакцию : т.е. изменения. Заменяем то, что есть в контейнере по айди (наш фреймлейаут) и отдаем фрагмент который придет в аргументе метода. В конце комитим, точно так же как комитим изменения и в шердпреференсах и в гите. И давайте подробнее останвимся на методе replace. Он просто заменит то, что лежит в контейнере. Что значит заменить? Удалить то что есть (если там что-либо есть) и положить в пустое место новое.

В нашем случае в первый раз будет пусто и туда положим например шутки, после, нажав на цитаты мы удалим фрагмент шуток из контейнера и положим туда фрагмент цитат. Точно так же как и у Активити : у фрагмента тоже есть методы жизненого цикла. Мы их залогируем и проверим что все действительно работает. Но нам нужно вызвать наш новый метод. Давайте перепишем прямо там же где у нас были красные методы в onCreate

```
val tabChosen: (Boolean) -> Unit = { jokesChosen ->
    if (jokesChosen)
        show(JokesFragment())
    else
        show(QuotesFragment())
```

Да, каждый раз мы будем создавать новый инстанс фрагмента, потому что нам нечего хранить. Все данные сохранены в реалм бд. Но если нам и понадобится что-либо хранить то мы придумаем иной способ. А пока давайте залогируем методы жц фрагментов.

```
override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInst
    Log.d( tag: "BaseFragmentTag",  msg: "onCreateView() ${javaClass.simpleName}")
    return inflater.inflate(R.layout.data_fragment, container,  attachToRoot: false)
}

override fun onDestroyView() {
    super.onDestroyView()
    Log.d( tag: "BaseFragmentTag",  msg: "onDestroyView() ${javaClass.simpleName}")
}
```

Так как у нас есть BaseFragment то я логирую прямо там, но чтобы понимать какой именно фрагмент у меня в работе я использую simpleName у джавовского класса. Теперь, давайте запустим проект и проверим что при нажатии на цитаты вызывается метод уничтожения вью фрагмента шуток и создается вью фрагмента цитат. И после уже опять нажмем на шутки, тогда должен очиститься из контейнера фрагмент цитат и новый экран шуток будет заменен Единственное я забыл на старте выбрать фрагмент который я хочу показать

```
tabLayout.addOnTabSelectedListener(TabListener(tabChosen))
show(JokesFragment())
```

Не беспокойтесь об этом, мы сохраним выбранный экран и в дальнейшем на старте будем сразу отображать нужный, а не шутки по умолчанию.

```
D/BaseFragmentTag: onCreateView() JokesFragment
D/BaseFragmentTag: onCreateView() QuotesFragment
D/BaseFragmentTag: onDestroyView() JokesFragment
D/BaseFragmentTag: onCreateView() JokesFragment
D/BaseFragmentTag: onDestroyView() QuotesFragment
```

Сначала создается экран шуток, меняем на цитаты, обратите внимание что второй экран создается раньше чем очищается первый. После я опять нажимаю на экран шуток и он создается раньше чем очищается второй.

Хорошо, с фрагментами в принципе разобрались. Они создаются лишь тогда, когда нам нужно и зазря не забивают память. А что насчет апликейшн класса? Там у нас создаются вьюмодели обеих фич сразу. Время разобраться с этим
Чтобы ясно понимать в чем проблема давайте залогируем теперь методы вьюмодели. И если для создания вьюмодели у нас есть блок init то как залогировать очищение вьюмодели? Так как у нас вьюмодели наследуются от ViewModel от Google, то у них есть метод onCleared который вызывается в тот момент, когда инстанс должен подчиститься перед смертью.
Чтобы понимать к какой фиче принадлежит фича я прокину в конструктор строку с именем.

```
class BaseViewModel<T>(
    private val name:String,
    private val interactor: CommonInteractor<T>,
    private val communication: CommonCommunication<T>,
    private val dispatcher: CoroutineDispatcher = Dispatchers.Main
) : ViewModel(), CommonViewModel<T> {

    init {
        Log.d( tag: "BaseViewModelTAG",  msg: "init $name")
    }

    override fun onCleared() {
        super.onCleared()
        Log.d( tag: "BaseViewModelTAG",  msg: "onCleared $name")
    }
```

И давайте я открою приложение и посмотрим логи. Я буду нажимать опять же на цитаты и вернусь на шутки. Посмотрим что в LogCat будет отображаться.

```
2021-07-22 11:45:41.049 6179-6179/com.github.johnnysc.jokeapp D/BaseViewModelTAG: init jokes
2021-07-22 11:45:41.051 6179-6179/com.github.johnnysc.jokeapp D/BaseViewModelTAG: init quotes
```

Как видите на старте приложения у меня сразу создаются обе вьюмодели и ни одна не очищается. Потому что вьюмодели нужно инициализировать не через апликейшн класс напрямую, а через фабрику вьюмоделей как рекомендует Гугл.
Для этого давайте напишем свою фабрику, почему свою? Потому что наши вьюмодели имеют не пустой конструктор. Если бы они имели пустой конструктор, то не нужно было бы писать фабрику вьюмоделей.
Итак, для начала чтобы различать вьюмодели давайте напишем конкретные классы для них

```
class JokesViewModel(
    interactor: CommonInteractor<Int>,
    communication: CommonCommunication<Int>
) : BaseViewModel<Int>( name: "jokes", interactor, communication)

class QuotesViewModel(
    interactor: CommonInteractor<String>,
    communication: CommonCommunication<String>
) : BaseViewModel<String>( name: "quotes", interactor, communication)
```

Чтобы это было возможно сделайте базовый абстрактным.

Теперь мы можем различать вьюмодели, потому что это нам нужно в нашей фабрике

```
class ViewModelsFactory : ViewModelProvider.Factory {
    override fun <T : ViewModel?> create(modelClass: Class<T>): T {
        val viewModel = when {
            modelClass.isAssignableFrom(JokesViewModel::class.java)-> makeJokesViewModel()
            modelClass.isAssignableFrom(QuotesViewModel::class.java)-> makeQuotesViewModel()
            else-> throw IllegalStateException("unknown type of viewModel")
        }
        return viewModel as T
    }
}
```

Вкратце обьясню суть фабрики вьюмоделей. Простой хешмап! Мы кладем вьюмодельку по ключу если в мапе еще нет такой и берем по ключу если таковая уже была положена в нее. Именно поэтому нам нужны разные вьюмодель классы. Итак, давайте вспомним какой конструктор у нас для вьюмоделей – ему нужны интерактор и комуникации (обертка над ливдатой). И здесь возникает вопрос : а как мне прокинуть в фабрику вьюмоделей нужные инстансы? Давайте напишем класс модуля, который умеет все это делать.

```
abstract class BaseModule<E, T : BaseViewModel<E>> {
    abstract fun getViewModel(): T
    abstract fun getCommunications() : BaseCommunication<E>
}
```

Просто абстрактный класс с 2 методами. Все наши модули фичей будут наследоваться от него и видимые методы будут : получить вьюмодель и коммуникации (они используются в адаптере ресайклера, возможно в дальнейшем мы придумаем иное решение чтобы использовать лишь вьюмодель). Итак, давайте напишем для начала модуль для шуток.

```
class JokesModule : BaseModule<Int, JokesViewModel>() {

    override fun getViewModel(): JokesViewModel {
        return JokesViewModel(getInteractor(), getCommunications())
    }

    override fun getCommunications(): BaseCommunication<Int> {

    }
}
```

И здесь нужно обьяснить что такое IoC (Inversion of Control). Я создаю интерактор не заблаговременно, а только лишь тогда, когда он понадобится моему вьюмодель. Точно так же я не создам заранее комуникации, а лишь тогда, когда они понадобятся. И здесь нужно сказать что не нужно создавать 2 инстанса, мы можем закешировать их. Покажу ниже как. Сейчас просто берем код из апликейшн класса и перемещаем его в модуль.
Итак, так как у нас в 2 местах используются коммуникации то я создаю локальную переменную и если в ней ничего не хранится то инициализирую ее и отдаю. Во второй раз на проверке мы не создадим еще одного инстанса, а будем использовать его же. Типа синглтон получается. Но конечно же я позже переделаю код таким образом чтобы не хранить коммуникации. А пока пусть будет так. Теперь напишем инит интерактора.
Для него нам нужны общие инстансы которые я наверно оставлю в апликейшн классе на время. Позже перенесу их в какой-нибудь модуль ядро, если понадобится.

```
private var communication: BaseCommunication<Int>? = null

override fun getCommunications(): BaseCommunication<Int> {
    if (communication == null)
        communication = BaseCommunication()
    return communication!!
}
```

Итак, для инициализации интерактора нам нужны пару обьектов и их я прокину в конструктор модуля.

```
class JokesModule(
    private val failureHandler: FailureHandler,
    private val realmProvider: RealmProvider,
    private val retrofit: Retrofit
) : BaseModule<Int, JokesViewModel>() {

    private var communication: BaseCommunication<Int>? = null

    override fun getCommunications(): BaseCommunication<Int> {
        if (communication == null)
            communication = BaseCommunication()
        return communication!!
    }

    override fun getViewModel() = JokesViewModel(getInteractor(), getCommunications())

    private fun getInteractor() =
        BaseInteractor(getRepository(), failureHandler, CommonSuccessMapper())
    private fun getRepository() =
        BaseRepository(getCacheDataSource(), getCloudDataSource(), BaseCachedData())
    private fun getCacheDataSource() =
        JokeCachedDataSource(realmProvider, JokeRealmMapper(), JokeRealmToCommonMapper())
    private fun getCloudDataSource() =
        JokeCloudDataSource(retrofit.create(BaseJokeService::class.java))
```

Да, нам нужно 3 общих обьекта и я их прокинул в конструктор. Там где создание инстанса просто вызов конструктора без аргументов я не стал писать методы. Надеюсь теперь вам понятна идея IoC – мы не создаем заранее то, что может и не нужно, начиная с вьюмодели шуток, если юзер решил не читать их, а сразу перейти к цитатам. Ладно, давайте напишем аналогичный модуль для цитат.

И в принципе ровно те же самые поля нужны и этому модулю. Мы бы могли их обозначить в базовом модуле, но это не так критично, если хотите сделайте так. Теперь нам нужно вернуться к фабрике вьюмоделей и прокинуть внутрь в конструктор модули.

```
class QuotesModule(
    private val failureHandler: FailureHandler,
    private val realmProvider: RealmProvider,
    private val retrofit: Retrofit
) : BaseModule<String, QuotesViewModel>() {

    private var communication: BaseCommunication<String>? = null

    override fun getCommunications(): BaseCommunication<String> {
        if (communication == null)
            communication = BaseCommunication()
        return communication!!
    }

    override fun getViewModel() = QuotesViewModel(getInteractor(), getCommunications())

    private fun getInteractor() =
        BaseInteractor(getRepository(), failureHandler, CommonSuccessMapper())
    private fun getRepository() = BaseRepository(
        getCacheDataSource(),
        getCloudDataSource(),
        BaseCachedData()
    )
    private fun getCacheDataSource() =
        QuoteCachedDataSource(realmProvider, QuoteRealmMapper(), QuoteRealmToCommonMapper())
    private fun getCloudDataSource() =
        QuoteCloudDataSource(retrofit.create(QuoteService::class.java))
}
```

Выбираем модуль и у него вызываем метод получения вьюмодели, все просто

```
class ViewModelsFactory(
    private val jokesModule: JokesModule,
    private val quotesModule: QuotesModule
) : ViewModelProvider.Factory {
    override fun <T : ViewModel?> create(modelClass: Class<T>): T {
        val module = when {
            modelClass.isAssignableFrom(JokesViewModel::class.java) -> jokesModule
            modelClass.isAssignableFrom(QuotesViewModel::class.java) -> quotesModule
            else -> throw IllegalStateException("unknown type of viewModel")
        }
        return module.getViewModel() as T
    }
}
```

Теперь нам нужно создать фабрику в апликейшн классе
Все нужные обьекты будут lateinit var так как сначала нам нужно их проинициализировать в методе onCreate и лишь после мы сможем инициализировать вьюмодельфабрику, потому пишем by lazy. Можете посмотреть лекции по котлин если не знакомы с инициализацией по требованию. Т.е. фабрика вьюмоделей создается только тогда, когда нужно. На самом деле мы бы могли точно так же создавать и все нужные инстансы, но мы точно знаем что они будут использоваться, потому не создаем заранее. Ровно так же можно было сделать и фабрику lateinit var factory и инициализировать после создания зависимых обьектов.
Но в таком случае я могу сделать переменную val и не дам ей в дальнейшем измениться, а что касается остальных полей, то они private. Хотя и фабрику можно сделать приватной и вынести метод создания вьюмодели. Давайте посмотрим как использовать ее

```
class JokeApp : Application() {

    val viewModelsFactory by lazy {
        ViewModelsFactory(
            JokesModule(failureHandler, realmProvider, retrofit),
            QuotesModule(failureHandler, realmProvider, retrofit)
        )
    }

    private lateinit var retrofit: Retrofit
    private lateinit var realmProvider: RealmProvider
    private lateinit var failureHandler: FailureHandler
```

Для этого вернемся в фрагмент и там проинициализируем вьюмодель.

```
abstract class BaseFragment<V : BaseViewModel<T>, T> : Fragment() {

    private lateinit var viewModel: BaseViewModel<T>

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        viewModel = ViewModelProvider(
            owner: this,
            (requireActivity().application as JokeApp).viewModelsFactory
        ).get(getViewModelClass())
    }

    protected abstract fun getViewModelClass(): Class<V>
```

Инициализируем в onCreate отдельно от остальных методов. Здесь нам понадобится провайдер от гугла, которому мы укажем owner фрагмент. Что это такое? Если указать активити через requireActivity то вьюмодель будет жить пока жива активити, т.е. владельцем вьюмодели будет активити. Когда она умрет тогда и вьюмодель. Но мы хотим чтобы вьюмодель очищалась когда фрагмент уже не отображается и потому передаем this.
Если вам нужно именно чтобы вьюмодель жила все время активити и не умирала с фрагментами (например 1 вьюмодель shared для 2 фрагментов : если они взаимодействуют между собой или делят одни данные), то просто укажите ViewModelProvider(requireActivity().
И так как у нас дженерик класс, то я создал протектед метод который даст имя класса.
И да, я пока что сделал комуникации открытыми но это на время пока не придумаю хорошее решение.

```
class JokesFragment : BaseFragment<JokesViewModel, Int>() {
    override fun checkBoxText() = "Show favorite joke"
    override fun actionButtonText() = "Get joke"
    override fun getViewModelClass() = JokesViewModel::class.java
```

Теперь конкретный фрагмент выглядит подобным образом. Перепишем так же и второй.
И можно запускать проект и смотреть на логи

```
2021-07-22 13:29:28.690 6641-6641/com.github.johnnysc.jokeapp D/BaseViewModelTAG: init jokes
2021-07-22 13:29:45.203 6641-6641/com.github.johnnysc.jokeapp D/BaseViewModelTAG: init quotes
2021-07-22 13:29:45.227 6641-6641/com.github.johnnysc.jokeapp D/BaseViewModelTAG: onCleared jokes
2021-07-22 13:29:56.020 6641-6641/com.github.johnnysc.jokeapp D/BaseViewModelTAG: init jokes
2021-07-22 13:29:56.049 6641-6641/com.github.johnnysc.jokeapp D/BaseViewModelTAG: onCleared quotes
```

Если помните, то у нас логируются методы создания вьюмодели и очищения. Я открыл приложение и нажал на цитаты. Шутки очистились. После я нажал на шутки и цитаты очистились. Все работает!
Но фабрика вьюмоделей работает не только в этом случае. Что насчет поворота экрана? При повороте экрана активити умирает и создается заново. А что насчет фрагмента? Фрагмнет менеджер хранит последнюю транзакцию и опять ее вызывает, а фабрика вьюмоделей не должна создать тот же инстанс вьюмодели, а взять из своей мапы старый. Давайте просто откроем приложение и повернем экран

```
logcat
2021-07-22 13:33:24.463 6743-6743/com.github.johnnysc.jokeapp D/BaseViewModelTAG: init jokes
2021-07-22 13:33:28.546 6743-6743/com.github.johnnysc.jokeapp D/BaseViewModelTAG: init jokes
2021-07-22 13:33:28.565 6743-6743/com.github.johnnysc.jokeapp D/BaseViewModelTAG: onCleared jokes
```

Стоп, что-то пошло не так? Да, все дело в том, что в активити опять же вызывается код, который ставит фрагмент. Нужно это предотвратить. И я покажу как можно сделать это

```
private fun show(fragment: Fragment, tag: String) {
    if (supportFragmentManager.fragments.isEmpty() ||
        supportFragmentManager.fragments.last().tag != tag
    ) {
        supportFragmentManager.beginTransaction()
            .replace(R.id.container, fragment, tag)
            .commit()
    }
}
```

Это не самое хорошее решение, но с его помощью я расскажу о тегах фрагмента.
У фрагмент менеджера мы можем получить доступ к фрагментам которые хранятся в нем (точнее в стеке). Если бы мы клали фрагменты поверх друг друга через add вместо replace то у нас бы в контейнере было бы больше 1 фрагмента. А через замену всегда 1. Итак, если список фрагментов пуст или же если тег последнего фрагмента не равен переданному, то мы заменяем фрагмент. Этот код позволяет не класть одинаковый фрагмент вместо себя.
Заметьте что у метода replace есть третий аргумент. Так же кроме тега есть еще айди и по нему можно найти в списке фрагментов нужный. Хотя у вас всегда есть список фрагментов и вы можете написать find и искать по нужному ключу. Но у фрагментменеджера уже есть методы типа findFragmentByTag, findFragmentById и т.д. Запустим код и проверим!

```
logcat
2021-07-22 13:39:29.495 6918-6918/com.github.johnnysc.jokeapp D/BaseViewModelTAG: init jokes
```

Как видите теперь при повороте экрана фрагмент не заменяется на себя же только другого инстанса и потому не срабатывает метод получения новой вьюмодели. Мы работаем с тем же фрагментом (типом) и с тем же инстансом вьюмодели. Потому что фабрика от гугла кладет в мапу по ключу (имени вьюмодели) и забирает ее когда фрагмент (тот кто владел вьюмоделью) умер и переродился и требует еще раз вьюмодель. Т.е. на первый onCreate фрагмента фабрика ищет в мапе вьюмодель и не находя ее создает через метод create и после

уже при повороте опять вызывается метод onCreate у фрагмента и мы опять пытаемся инициализировать вьюмодель, но фабрика находит в своей мапе инстанс и отдает его.
И давайте я сам добавлю сохранение выбранной позиции таба и будем загружать на старте нужный экран.

Я предлагаю вам самостоятельно написать вьюмодель для активити где будет храниться sharedpreferences выбора таба.
И в любом случае весь код будет доступен на гитхаб по ссылке
https://github.com/johnnysc/cleanarchexample

И вообще смотрите стримы на ютуб на канале
https://www.youtube.com/c/easyCodeRu
Там то же самое что и в лекциях и намного больше и лучше

п.с. я добавил метод тега в базовый фрагмент, можно использовать его
п.п.с. в следующих лекциях будет речь о сервис локаторе и юай тестах, так что мы не закончили с этим проектом