

Разные классы для разных слоев

Мапинг данных

Давайте все же отдельной лекцией пофиксим наши классы для шуток.

Итак, проблема на данный момент такова что у нас класс `JokeServerModel` делает слишком много вещей

```
fun change(cacheDataSource: CacheDataSource)
fun toBaseJoke()
fun toFavoriteJoke()
fun toJokeRealm()
```

Но по-хорошему здесь должен быть лишь 1 метод – мапинг к объекту бизнес логики. А она уже в свою очередь должна преобразовываться к реалм модели и к юай и что угодно. Почему? А потому что завтра вы решите не хранить в кеше данные и вам нужно будет модифицировать класс серверной модели. Вообще не логично. Удаление или изменение реалм модели не должно влиять на серверную модель. Ну и нейминг тоже нужно пофиксить.

Предлагаю так, те классы шуток которые отображаются на экране будут `JokeUiModel`, те которые от сервера все так же `JokeServerModel`, а те которые в реалм `JokeRealmModel`.

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

Так же я переименовал метод чтобы не было тафтологии. Теперь напишем уже просто класс `Joke` без суффиксов как модель бизнес логики.

```
class Joke {
    private val id: Int,
    private val type: String,
    private val text: String,
    private val punchline: String,
} {
    fun change(cacheDataSource: CacheDataSource) = cacheDataSource.addOrRemove(id, joke: this)
    fun toBaseJoke() = BaseJokeUiModel(text, punchline)
    fun toFavoriteJoke() = FavoriteJokeUiModel(text, punchline)
    fun toJokeRealm() = {...}
}
```

Просто переместили код из серверной модели. Далее нам нужно в серверной модели написать метод мапинга к этой модели.

```
fun toJoke() = Joke(id, type, text, punchline)
```

Теперь нам нужно найти все места и пофиксить код

Для начала начнем с кешдатасорса

```
interface CacheDataSource {  
    fun getJoke(jokeCachedCallback: JokeCachedCallback)  
    fun addOrRemove(id: Int, joke: Joke): JokeUiModel  
}  
  
interface JokeCachedCallback {  
    fun provide(joke: Joke)  
    fun fail()  
}
```

Теперь выглядит логичнее. Никаких серверных моделей при работе с кешем.

Так же нам нужно в колбеке клауда отдавать уже самую модель бизнес логики

```
interface JokeCloudCallback {  
    fun provide(joke: Joke)  
    fun fail(error: ErrorType)  
}
```

```
if (response.isSuccessful) {  
    callback.provide(response.body()!!.toJoke())  
} else {
```

Пофиксим же теперь Model

```
class BaseModel(  
    private val cacheDataSource: CacheDataSource,  
    private val cloudDataSource: CloudDataSource,  
    private val resourceManager: ResourceManager  
) : Model {  
    private val noConnection by lazy { NoConnection(resourceManager) }  
    private val serviceUnavailable by lazy { ServiceUnavailable(resourceManager) }  
    private val noCachedJokes by lazy { NoCachedJokes(resourceManager) }  
  
    private var jokeCallback: JokeCallback? = null  
    private var cachedJoke: Joke? = null  
    private var getJokeFromCache = false  
  
    override fun chooseDataSource(cached: Boolean) {  
        getJokeFromCache = cached  
    }  
}
```

```

override fun getJoke() {
    if (getJokeFromCache) {
        cacheDataSource.getJoke(object : JokeCachedCallback {
            override fun provide(joke: Joke) {
                cachedJoke = joke
                jokeCallback?.provide(joke.toFavoriteJoke())
            }
            override fun fail() {
                cachedJoke = null
                jokeCallback?.provide(FailedJokeUiModel(noCachedJokes.getMessage()))
            }
        })
    } else {
        cloudDataSource.getJoke(object : JokeCloudCallback {
            override fun provide(joke: Joke) {
                cachedJoke = joke
                jokeCallback?.provide(joke.toBaseJoke())
            }

            override fun fail(error: ErrorType) {
                cachedJoke = null
                val failure = if (error == ErrorType.NO_CONNECTION) noConnection else serviceUnavailable
                jokeCallback?.provide(FailedJokeUiModel(failure.getMessage()))
            }
        })
    }
}

override fun changeJokeStatus(jokeCallback: JokeCallback) {
    cachedJoke?.let { it: Joke
        jokeCallback.provide(it.change(cacheDataSource))
    }
}

```

Запускаем проект и видим что все правильно работает.

И напоследок давайте отредактируем один кусок там где возврат ошибки

```

override fun onFailure(call: Call<JokeServerModel>, t: Throwable) {
    if (t is UnknownHostException)
        callback.fail(ErrorType.NO_CONNECTION)
    else
        callback.fail(ErrorType.SERVICE_UNAVAILABLE)
}

```

Так как в обоих ветвях вызов одного и того же метода но с разными аргументами, то лучше

```

override fun onFailure(call: Call<JokeServerModel>, t: Throwable) {
    val errorType = if (t is UnknownHostException)
        ErrorType.NO_CONNECTION
    else
        ErrorType.SERVICE_UNAVAILABLE
    callback.fail(errorType)
}

```

Вот и все. Создаем переменную и отдаем в метод. Так намного лучше и не дублируем код.

DRY – Don't Repeat Yourself даже в таких мелочах (а это поверьте мне не мелочи вовсе).

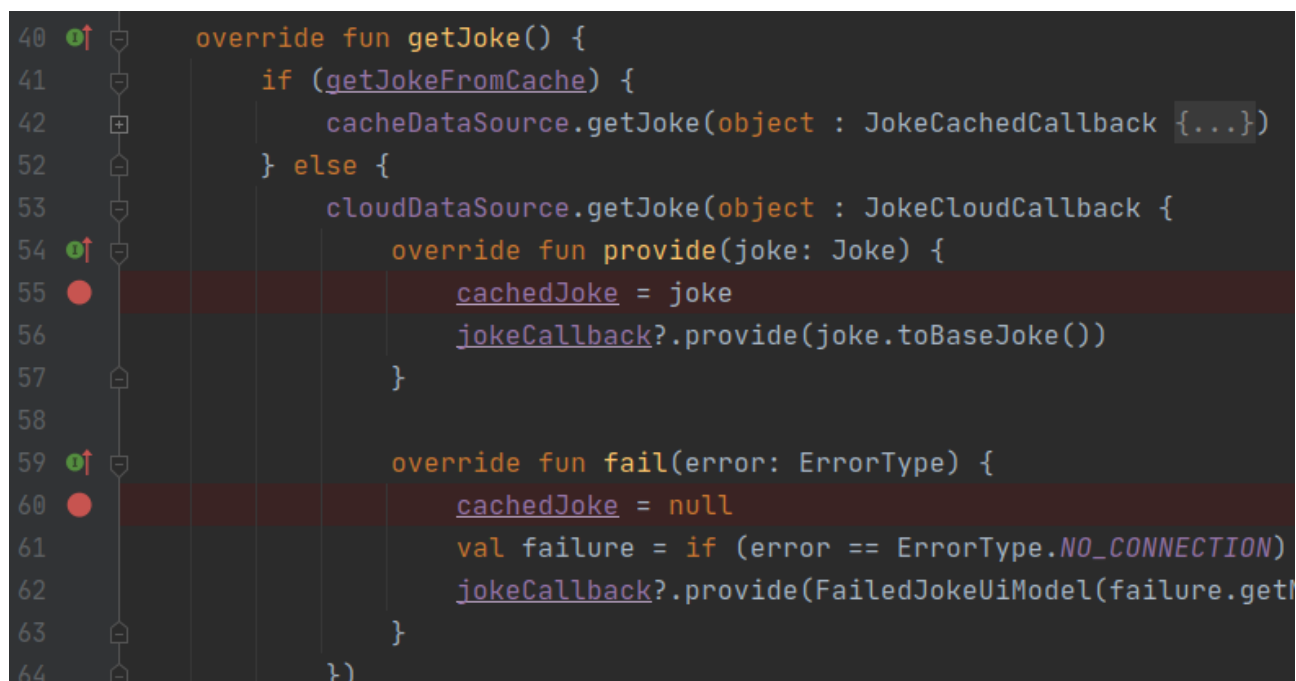
Ну и анонс следующей темы:

Вам не кажется что в нашем маленьком проекте так много колбеков?

```
interface JokeCloudCallback
interface JokeCachedCallback
object : retrofit2.Callback<JokeServerModel>
interface DataCallback
interface JokeCallback
```

Самое ужасное что наш код идет прямо снизу вверх и после прерывается и возвращается то в одно место, то в другое.

Возьмем модель. Мы идем сверху вниз. Линия 41 if(getJokeFromCache) после идем например вниз на 53 линию и после.... На 55 или 60



```
40  override fun getJoke() {
41      if (getJokeFromCache) {
42          cacheDataSource.getJoke(object : JokeCachedCallback {...})
52      } else {
53          cloudDataSource.getJoke(object : JokeCloudCallback {
54              override fun provide(joke: Joke) {
55                  cachedJoke = joke
56                  jokeCallback?.provide(joke.toBaseJoke())
57              }
58
59              override fun fail(error: ErrorType) {
60                  cachedJoke = null
61                  val failure = if (error == ErrorType.NO_CONNECTION)
62                      jokeCallback?.provide(FailedJokeUiModel(failure.get))
63              }
64          })
}
```

Конечно же этому есть объяснение, у нас асинхронный код. Мы не знаем когда вернется ответ от сервера и потому делаем колбеки. Но их надо хранить, зачищать и так далее.

Было бы замечательно писать асинхронный код синхронно... т.е. последовательно.

И это возможно! В следующей лекции я расскажу про корутины!