Корутины в андроид

Никаких больше колбеков

Содержание

1. Подключаем корутины

Корутины это сопрограммы в котлин, т.е. отдельные легковесные потоки которые работают паралельно текущему потоку. Это очень удобно потому что нам не нужно создавать руками огромные обьекты класса Thread и запускать их.

Детальную информация по корутинам можно найти здесь
https://developer.android.com/kotlin/coroutines

Подключаем корутины отдельной либой в build.gradle

```
implementation("org.jetbrains.kotlinx:kotlinx-coroutines-android:1.3.9")
```

Так же для того чтобы они работали исправно подключаем еще одну либу

```
implementation("androidx.lifecycle:lifecycle-viewmodel-ktx:2.3.1")
```

Суть в том, что наши вьюмодели простые классы которые опять же ни от кого не наследуются. И мы это исправим.

```
import androidx.lifecycle.ViewModel

/** @author Asatryan on 13.06.2021 ...*/
class ViewModel(private val model: Model) : ViewModel() {
```

Но нашу историю про корутины мы начнем с конца – а точнее с сервиса ретрофита.

```
interface JokeService {

    @GET( value: "https://official-joke-api.appspot.com/random_joke/")
    suspend fun getJoke() : JokeServerModel
}
```

Мы убираем первый колбек и наш метод становится как будто бы синхронным. Но погодите, что за новое ключевое слово suspend. Оно указывает нам на то, что метод как будто бы асинхронный, хотя на самом деле магия в том, что мы прерываем поток и продолжаем работу.

Ведь в чем проблема обычных Thread – если ты вызвал start() то все, поток начал исполняться и ты ничего не сможешь с этим поделать (ну кроме wait). Все что можно сделать это отписаться от результата. А с корутинами не так. Давайте посмотрим дальше

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

    override fun getJoke(callback: JokeCloudCallback) {
        val result =  service.getJoke()
```

Посмотрите внимательно как я вызываю метод в датасторе. Видно что я вызываю его как будто бы он синхронный. Но погодите-ка, что-то не так. Наведем мышкой на красное подчеркивание -

Suspend function 'getJoke' should be called only from a coroutine or another suspend function

Alt+Enter – make getJoke suspend. И вуаля, сам метод интерфейса становится suspend.

Но это не все, посмотрите на значок слева – видите? В этом и суть – корутины прерывают текущий поток в котором они вызваны и продолжают его тогда, когда приходит ответ от сервера. Т.е. еще раз: в обычном потоке исполнение идет сверху вниз и ничего не остановит его кроме как Thread.sleep() (или wait()), и все методы которые синхронные (обычные) идут сверху вниз и просто вызываются. Даже если у вас асинхронный метод он вызовется и пойдет дальше вниз. Именно поэтому ранее мы отдавали колбеки чтобы вернутья из другой точки (прочитайте предыдущую лекцию #10). Теперь же мы находимся в потоке и сверху вниз читаем код – когда компилятор видит suspend функцию то ставит исполнение текущего потока на паузу и дожидается ответа от корутины. И когда она закончит работу то текущий поток разблокируется и продолжит свою работу. Считайте это нечто Thread.sleep но пока текущий поток спит в другом потоке (корутине) идет работа. И только эта корутина может вернуть текущий поток к жизни и продолжить работу. Ничего страшного если непонятно, мы подебажим и вы увидете как это работает (спойлер: сверху вниз как обычный код в 1 потоке).

И так как мы получаем результат как будто бы сразу, то нам нужен силд класс для того, чтобы работать с успехом и ошибкой. Напишем дженерик силд класс для этого

```
sealed class Result<out R, out E> {
    data class Success<out T>(val data: T) : Result<T, Nothing>()
    data class Error<out S>(val exception: S) : Result<Nothing, S>()
}
```

И теперь посмотрите как мы поменяем наш метод в клаудДатаСторе

```
interface CloudDataSource {
    suspend fun getJoke(): Result<JokeServerModel, ErrorType>
}
```

Красоте же ну – получаем данные или ошибку, но сразу же как синхронный метод.

Теперь давайте напишем реализацию в базовом классе.

Вы просто посмотрите на это. Если бы вы не знали про корутины и про то, что мы берем данные из сети, вы бы подумали что это синхронный метод. Мы начинаем сверху вниз и идем : линия 26, после идет или 27 или 29. У нас больше нет колбеков. Синхронно

написанный код. Красота. И заметьте как проще стало отдавать данные: 1 линией. В колбеке мы проверяли на успешность и так далее. А здесь 1 try catch. Мы обезопасили себя сразу.

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

    override suspend fun getJoke(): Result<JokeServerModel, ErrorType> {
        return try {
            val result = service.getJoke()
            Result.Success(result)
        } catch (e: Exception) {
            val errorType = if (e is UnknownHostException)
                ErrorType.NO_CONNECTION
            else
                ErrorType.SERVICE_UNAVAILABLE
            Result.Error(errorType)
        }
    }
}
```

Теперь давайте посмотрим что произойдет далее. Нам нужно поменять нашу модель.

```
interface Model {

    suspend fun getJoke() : JokeUiModel
```

Опять же, если вы вызываете саспенд функции из другой функции, то она должна быть так же саспенд. Переписали метод и перепишем реализацию.

```
override suspend fun getJoke(): JokeUiModel {
    if (getJokeFromCache) {...} else {
        return when (val result = cloudDataSource.getJoke()) {
            is Result.Success<JokeServerModel> -> {
                result.data.toJoke().let { it: Joke
                    cachedJoke = it
                    it.toBaseJoke()
                }
            }
            is Result.Error<ErrorType> -> {
                cachedJoke = null
                val failure = if (result.exception == ErrorType.NO_CONNECTION)
                    noConnection
                else serviceUnavailable
                FailedJokeUiModel(failure.getMessage())
            }
        }
    }
}
```

Итак, как видите все красиво и прекрасно – рассматриваем что пришло от интерфейса датастора – если успех то сохраняем в кеш и мапим к юай слою, если же ошибка, то чистим кеш и мапим к failed модели юай слоя. Но подождите-ка, последняя фигурная красная.

Потому что метод должен вернуть результат, а у нас кроме клаудДатаСтора еще и кешДатаСтор. Значит надо пофиксить и там.

```
interface CacheDataSource {
    suspend fun getJoke() : Result<Joke, Unit>
```

Мы опять же делаем саспенд функцию, которая вернет или успех или ничего. Пофиксим реализацию.

```
class BaseCachedDataSource(private val realm: Realm) : CacheDataSource {

    override suspend fun getJoke(): Result<Joke, Unit> {
        realm.let { it: Realm
            val jokes = it.where(JokeRealm::class.java).findAll()
            if (jokes.isEmpty())
                return Result.Error(Unit)
            else
                jokes.random().let { joke ->
                    return Result.Success(
                        Joke(
                            joke.id,
                            joke.type,
                            joke.text,
                            joke.punchLine
```

Как видите все просто. Теперь можно пофиксить и метод модели

```
override suspend fun getJoke(): JokeUiModel {
    if (getJokeFromCache) {
        return when (val result = cacheDataSource.getJoke()) {
            is Result.Success<Joke> -> result.data.let { it: Joke
                cachedJoke = it
                it.toFavoriteJoke()
            }
            is Result.Error -> {
                cachedJoke = null
                FailedJokeUiModel(noCachedJokes.getMessage())
            }
```

Выглядит классно, неправда ли? Никаких колбеков!

Пофиксили модель, значит нужно пофиксить и вьюмодель. Давайте посмотрим

И здесь мы сталкиваемся с тем, что не можем так же пометить фунцию в вьюмодели suspend.

Почему? Потому что это вьюмодель и здесь мы должны запустить корутину. Но как? Вот так

```
fun getJoke() = viewModelScope.launch {  this: CoroutineScope
    val uiModel = model.getJoke()
    dataCallback?.let {  it: DataCallback
        uiModel.map(it)
    }
}
```

Что? Колбек опять? Да, пока что здесь он нам нужен. Мы его заменим чуть позже, не беспокойтесь. А пока посмотрим на код – viewModelScope что это? Скоуп вьюмодели.

Что значит скоуп? Думайте о пределах жизни вьюмодели. Ведь если вы ушли с экрана и закрыли приложение то ваша корутина вернет ответ… куда? А никуда. Вам уже не нужно будет. Т.е. мы смотрим на ответ от корутины (саспенд функции) пока наша вьюмодель жива. Она умирает (onCleared()) когда закрывается экран (условно Activity#onDestroy). И в рамках скоупа мы запускаем нашу саспенд функцию. Когда вернется ответ мы отправим его на юай. Запустим код?

Да, все работает как и предполагалось. Но давайте вернемся в активити. RunOnUiThread.

Давайте его уберем. Но как? У нас же данные приходят с другого потока. Как мы можем менять потоки с помощью корутин проще и не писать каждый раз это в активити?

Я убрал runOnUiThread из активити и запустил код и вуаля! Все и так работает. Почему?

```
viewModel.init(object : DataCallback {
    override fun provideText(text: String) {
        button.isEnabled = true
        progressBar.visibility = View.INVISIBLE
        textView.text = text
    }

    override fun provideIconRes(id: Int) {
        changeButton.setImageResource(id)
    }
})
```

Давайте я покажу вам как я немного переписал метод addOrRemove

```
suspend fun changeJokeStatus(): JokeUiModel?
```

Сначала я переписал на саспенд функцию изменения статуса, да, пока что нулабл.

```
override suspend fun changeJokeStatus(): JokeUiModel? = cachedJoke?.change(cacheDataSource)
```

Далее я вызываю у нулабл кеша метод изменения и его я тоже поменял

```
suspend fun change(cacheDataSource: CacheDataSource) = cacheDataSource.addOrRemove(id, joke: this)
```

И давайте посмотрим что я натворил в кешдатасорс

```
    suspend fun addOrRemove(id: Int, joke: Joke): JokeUiModel
}
```

А теперь реализация

```
override suspend fun addOrRemove(id: Int, joke: Joke): JokeUiModel =
    withContext(Dispatchers.IO) { this: CoroutineScope
        Realm.getDefaultInstance().use { it: Realm!
            val jokeRealm =
                it.where(JokeRealm::class.java).equalTo( fieldName: "id", id).findFirst()
            return@withContext if (jokeRealm == null) {
                it.executeTransaction { transaction ->
                    val newJoke = joke.toJokeRealm()
                    transaction.insert(newJoke)
                }
                joke.toFavoriteJoke()
            } else {
                it.executeTransaction { it: Realm
                    jokeRealm.deleteFromRealm()
                }
                joke.toBaseJoke()
```

Во-первых я написал withContext(Dispathers.IO) это означает – выполнить корутину на этом потоке – потоке ввода вывода. Вы можете посмотреть какие еще есть потоки, например Main который и есть MainThread который UI поток. Почему я это сделал? А потому что реалм требует писать в бд в другом потоке и нельзя полученный обьект (jokeRealm) писать в него же если он был получен из другого потока. Как видите я убрал Async и теперь мои методы executeTransaction на том же потоке что и Dispatchers.IO. Т.е. я читаю на потоке ввода вывода и пишу в нем же. (да, я написал сразу Realm.getDefaultInstance().use и позже улучшу код)

```
fun changeJokeStatus() = viewModelScope.launch { this: CoroutineScope
    val uiModel = model.changeJokeStatus()
    dataCallback?.let { it: DataCallback
        uiModel?.map(it)
    }
}
```

Теперь метод в вьюмодели выглядит так – я в скоупе вьюмодели вызываю метод у модели который работает на потоке ввода вывода. И в итоге все равно отдаю данные на мейн поток и ничего не крашится? Вроде бы да. Суть в том, что метод launch по умолчанию переключает поток на мейн. Чтобы в этом убедиться можете убрать из модели withContext(Dispatchers.IO) и увидете краш. Или поменять его на Dispatchers.Main и тоже будет краш.

Все что вам нужно знать на данный момент – viewModelScope.launch меняет поток на мейн и потому вам не нужно писать runOnMainThread каждый раз когда данные из вьюмодели идут на юай. Но вам нужно писать withContext(Dispathers.IO) если вы выполняете код в другом потоке, например получаете данные из сети. Можете вернуть код на HttpUrlConnection и увидете что это так.

Вернемся к нашему методу для получения данных из сети. Помните как было раньше? Ретрофит работал с колбеком и возвращал из другого потока данные и потому мы не могли их сразу показать на мейне, потому что если обьект создан в другом потоке, то он не может просто так взять и перекочевать на другой поток. М – многопоточность. Именно поэтому мы и использовали runOnUiThread. Но давайте посмотрим что там под капотом у этого метода

```
public final void runOnUiThread(Runnable action) {
    if (Thread.currentThread() != mUiThread) {
        mHandler.post(action);
    } else {
        action.run();
    }
}
```

Если текущий поток не юай поток, то постим ранебл экшн через хендлер, если же мы и так на юай потоке, то просто запускаем код. Ладно, что такое хендлер и что за пост. Время поговорить о многопоточности по-настоящему.

Итак, мы говорим, что поток начинает выполнение по методу start и беспрепятственно идет к завершению и в конце умирает. Ладно, давайте поговорим про андроид мейн поток в котором работают все приложения. Как же так – это ведь тоже поток и он запускается когда юзер нажимает на иконку на рабочем столе (очень грубо говоря). Почему же он не завершает работу и приложение не умирает спустя 3 миллисекунды? Все дело в бесконечном цикле. В мейн потоке работает некий бесконечный цикл (грубо говоря while(true)) и когда вы выходите из приложения, то в нем есть 1 иф (if (exiting) break;) и при выходе этот флаг ставится в true. Вот и все. Но на самом деле конечно же там есть такая штука как Looper, вы можете найти такой код

```
public static Looper getMainLooper() {
```

это именно тот цикл в котором и работает мейн. Далее вам нужно в этот цикл прокидывать некие экшны типа Runnable, т.е. код выглядит как-то так (вы всегда можете посмотреть исходники сами).

Но в общем и целом есть некая очередь (да, в реальности там стек а не список простой) в которую вы кладете некое действие и когда цикл доходит до вашего действия то выполняет

его и убирает из очереди. Но как нам добавить ранебл в очередь? Притом что он приходит из другого потока? Для этого есть специальная очередь которая умеет работать с этим кейсом.

```
class MyMainThread {

    private val list = mutableListOf<Runnable>()
    private var exiting = false
    fun addAction(runnable: Runnable) {
        list.add(runnable)
    }
    fun main() {
        while (true) {
            if (exiting)
                break
            list.forEach { it: Runnable
                it.run()
                list.remove(it)
            }
        }
    }
    fun exit() {
        exiting = true
    }
}
```

Если вам интересно вы можете сами погуглить и поискать как это все под капотом, но суть я думаю донес. Итого, вам нужен хендлер Handler который добавит ваш экшн ранебл в очередь и когда мейн поток освободится (закончит шаг и вернется в while(true)) то выполнит код.

Теперь предлагаю посмотреть на исходники ретрофита. Ранее был простой колбек асинхронный, как он стал работать с саспенд функцией? А вот так вот просто, смотрите:

Обычная экстеншн функция, которая работает с колбеком и когда выполняется асинхронный код то просто с помощью корутины стартует ее и возвращается с ответом или ошибкой.

Видите? Тот же самый код который был до того как мы внедрили корутины: Простой колбек и 2 метода: но корутина ожидает их и продолжает работу с помощью методов continuation.resume и continuation.resumeWithException. Вы точно так же можете любой старый или унаследованный код не переписывать на корутины, а написать экстеншн функцию и просто с помощью continuation отдавать дальше ваш код. Вот и вся суть. И кто-то может сказать – так может тогда и ретрофит код выполнять на потоке Dispatchers.IO ? Ну можно конечно, хотя оно и так асинхронное и имеет свои потоки, так что не парьтесь об этом (ниже рассмотрим детальнее). Ваш viewModelScope.launch переключит потоки в конце

концов и на ваш активити пойдут данные безопасно. Т.е. launch грубо говоря под капотом (условно) делает то же самое что и runOnUiThread. Вот и все.

```
suspend fun <T : Any> Call<T>.awaitResponse(): Response<T> {
  return suspendCancellableCoroutine { continuation ->
    continuation.invokeOnCancellation {  it: Throwable?
      cancel()
    }
    enqueue(object : Callback<T> {
      override fun onResponse(call: Call<T>, response: Response<T>) {
        continuation.resume(response)
      }

      override fun onFailure(call: Call<T>, t: Throwable) {
        continuation.resumeWithException(t)
      }
    })
  }
}
```

Но давайте все же почистим наш код перед тем как мы будем писать юай тесты (ой, спойлер). Я не хочу чтобы мы в кеш дата сорсе просто брали реалмовый дефолтный инстанс и работали с ним. Давайте напишем реалмпровайдер какой-нибудь и просто отдавать будет нужный инстанс. Зачем? Чтобы при юай тестах легко и просто менять этот провайдер и добавлять нужный для тестов реалм, чтобы не писать в ту же бд что и для юзера какие-то данные и удалять их вместе с нужными. Покажу в следующей лекции что и как.

```
interface RealmProvider {

    fun provide(): Realm
}

class BaseRealmProvider : RealmProvider {

    override fun provide(): Realm = Realm.getDefaultInstance()
}
```

Да, один метод который даст нам реалм инстанс. Перепишем немного кешдатасорс теперь.

```
class BaseCachedDataSource(private val realmProvider: RealmProvider) : CacheDataSource {

    override suspend fun getJoke(): Result<Joke, Unit> {
        realmProvider.provide().use {  it: Realm
```

И пофиксим наконец наш апликейшн класс

```
viewModel = ViewModel(
    BaseModel(
        BaseCachedDataSource(BaseRealmProvider()),
        BaseCloudDataSource(retrofit.create(JokeService::class.java)),
        BaseResourceManager( context: this)
```

Запускаем проект и проверяем: да, все работает как и ранее.

Вернемся к вопросу о потоке на котором запускается ретрофит.

Посмотрим исходники еще раз:

провалимся в метод enqueue

```
@Override public void enqueue(final Callback<T> callback) {
  checkNotNull(callback, message: "callback == null");

  delegate.enqueue(new Callback<T>() {
    @Override public void onResponse(Call<T> call, final Response<T> response) {
      callbackExecutor.execute(new Runnable() {
```

как видите метод работает на другом потоке: callbackExecutor. Посмотрим на него

```
static final class ExecutorCallbackCall<T> implements Call<T> {
  final Executor callbackExecutor;
  final Call<T> delegate;
```

В джава есть пакет java.util.concurrency и писать многопоточный код можно не только создавая потоки с помощью Thread, Runnable но еще и так. Как видите у интерфейса есть много реализаций. Значит ретрофит все же сам создает себе поток и там обрабатывает ответ от сервера и после уже на мейн поток мы отправляем с помощью viewModelScope.launch.

```
public interface Executor {

    /**
     * Executes the given command at some time in the future.  The command
     * may execute in a new thread, in a pooled thread, or in the calling
     * thread, at the discretion of the {@code Executor} implementation.
     *
     * @param command the runnable task
     * @throws RejectedExecutionException if this task cannot be
     * accepted for execution
     * @throws NullPointerException if command is null
     */
    void execute(Runnable command);
}
```

Теперь у нас встал вопрос таким образом: мы же внедряем легковесные корутины вместо существующих конструкций. Проще говоря – что поменялось в использовании ретрофита?

Раньше у нас выполнялся код асинхронно с помощью executor и просто отдавался результат в колбек. Теперь мы не страдаем от колбеков, но все равно код выполняется внутри ретрофита на этом executor. Мы не используем котлин корутины до конца, а лишь 1 фишку – прерывание себя и продолжение кода из другой точки чтобы снаружи было красиво. Давайте все же напишем код иначе. Ведь мы же смогли переписать код для кешдатасорса таким образом чтобы на шедулере ввода вывода мы и читали и писали реалм данные. Ретрофит так же имеет возможность выполнять код синхронно на том потоке на котором он вызывался. Давайте посмотрим как

```
interface JokeService {

    @GET( value: "https://official-joke-api.appspot.com/random_joke/")
    fun getJoke() : Call<JokeServerModel>
}
```

Мы убрали ключевое слово suspend и сделали наш метод простым Call<JokeServerModel>. Теперь сможем вызвать результат синхронно. Посмотрите на это

```
override suspend fun getJoke(): Result<JokeServerModel, ErrorType> {
    return try {
        val result: JokeServerModel = service.getJoke().execute().body()!!
        Log.d( tag: "threadLogTag",  msg: "currentThread ${Thread.currentThread().name}")
        Result.Success(result)
```

Так же уберите withContext(Dispatchers.IO) из метода в модели который над методами датасорсов. Запустите код и посмотрите что будет, а лучше подебажьте.

```
        return try {
            val result: JokeServerModel = service.getJoke().execute().body()!!
            Log.d( tag: "threadLogTag",  msg: "currentThread ${Thread.currentThread().name}")
            Result.Success(result)
        } catch (e: Exception) {  e: "android.os.NetworkOnMainThreadException"
            val errorType = if (e is UnknownHostException)  e: "android.os.NetworkOnMainThreadException"
                ErrorType.NO_CONNECTION
```

Мы на мейне запустили запрос в сеть. Теперь можем спокойно возвращать withContext(Dispatchers.IO) в модель, а здесь проигнорируйте подчеркивание execute(). Просто АС не видит что мы запускаем этот код не на мейн потоке снаружи. Итак, если мы поменяли потоки то теперь код будет выполняться на указанном.

Итого – withContext(Dispatchers.IO) указали в 1 месте – в классе Модели в методе getJoke()

```
    override suspend fun getJoke(): JokeUiModel = withContext(Dispatchers.IO) {  this: CoroutineScope
        if (getJokeFromCache) {
            return@withContext when (val result = cacheDataSource.getJoke()) {...}
        } else {
            return@withContext when (val result = cloudDataSource.getJoke()) {...}
        }
    }
```

Надеюсь все понятно теперь стало. Мы используем корутины чтобы не работать с колбеками и писать последовательный код сверху вниз и чтобы переключать потоки было легко и просто. Поэтому вот вам простое правило которое нужно запомнить:

Запомните одно простое правило: если вы получаете данные из сети или из бд или кладете в бд и никак не нужен вам юай (активити), то делайте все эти операции в другом потоке, например ввода-вывода. Мейн поток нужно использовать лишь для отображения данных и точка. Переключением займется viewModelScope.launch

Итак, давайте проверим что мы сделали. В предыдущей лекции мы составили список

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

Давайте теперь проверим все ли колбеки мы сможем сейчас убрать

JokeCloudCallback – убираем, у нас корутины

JokeCachedCallback – тоже можно удалить – корутины сделали свое дело

retrofit2.Callback убрали тоже, теперь мы получаем данные напрямую

DataCallback – все же оставили, ибо он помогает отправлять данные на активити

JokeCallback – давайте удалим и так же почистим все классы где он использовался

Стало намного чище

```
interface Model {
    suspend fun getJoke(): JokeUiModel
    suspend fun changeJokeStatus(): JokeUiModel?
    fun chooseDataSource(cached: Boolean)
}
```

Никаких непонятных методов типа init clear больше нет в модели. У нас нет колбеков и ничего инициализировать и зачищать не нужно. Посмотрите, 3 метода, 1 обычный, но и те 2 выглядят таким же образом. Будто они синхронные.

Итого: 4 из 5 колбеков убрали. Но это еще не все новости: писать юнит тесты с корутинами еще проще! Ведь если все методы выглядят как синхронные, то и писать тесты можно проще.

Напишем юнит тест для модели

Я написал тестовые реализации для того чтобы легко можно было получать данные от датасорсов и проверять что в базе есть айди или нет. Итак я выбираю датасорс не из кеша, заставляю мой тестклауддатасорс отдавать мне успех в следующий раз когда надо будет ответить и получаю шутку от модели. Проверяю что она успешная. Далее меняю ее статус, то есть нажимаю будто бы в юай на сердечко и провереряю что в тесткешдатасорс добавилось.

Все остальные тесты можете написать сами. В чем суть юнит тестов еще раз – вы можете проверить все возможные сценарии не запуская проект и не используя свои пальцы и глаза. А лишь один раз написав тесты и запустив их. В последующих лекциях мы обсудим какие тест кейсы могут быть и попробуем написать юай тесты на все.

```kotlin
class BaseModelTest {

    @Test
    fun test_change_data_source(): Unit = runBlocking { this: CoroutineScope
        val cacheDataSource = TestCacheDataSource()
        val cloudDataSource = TestCloudDataSource()
        val model = BaseModel(cacheDataSource, cloudDataSource, TestResourceManager())
        model.chooseDataSource( cached: false)
        cloudDataSource.getJokeWithResult( success: true)
        val joke = model.getJoke()
        assertEquals(joke is BaseJokeUiModel,  actual: true)
        model.changeJokeStatus()
        assertEquals(cacheDataSource.checkContainsId( id: 0),  actual: true)
    }

    private inner class TestCacheDataSource : CacheDataSource {...}

    private inner class TestCloudDataSource : CloudDataSource {...}

    private inner class TestResourceManager : ResourceManager {...}
}
```

```kotlin
private inner class TestCacheDataSource : CacheDataSource {

    private val map = HashMap<Int, Joke>()
    private var success: Boolean = true
    private var nextJokeIdToGet = -1

    fun getNextJokeWithResult(success: Boolean, id:Int) {
        this.success = success
        nextJokeIdToGet = id
    }

    override suspend fun getJoke(): Result<Joke, Unit> {
        return if (success) {
            Result.Success(map[nextJokeIdToGet]!!)
        } else {
            Result.Error(Unit)
        }
    }

    override suspend fun addOrRemove(id: Int, joke: Joke): JokeUiModel {
        return if (map.containsKey(id)) {
            val result = map[id]!!.toBaseJoke()
            map.remove(id)
            result
        } else {
            map[id] = joke
            joke.toFavoriteJoke()
        }
    }

    fun checkContainsId(id: Int) = map.containsKey(id)
}
```

```
private inner class TestCloudDataSource : CloudDataSource {

    private var success = true
    private var count = 0

    fun getJokeWithResult(success: Boolean) {
        this.success = success
    }

    override suspend fun getJoke(): Result<JokeServerModel, ErrorType> {
        return if (success) {
            Result.Success(JokeServerModel(count++, type: "type", text: "text$count", punchline: "punchline$count"))
        } else {
            Result.Error(ErrorType.NO_CONNECTION)
        }
    }
}
```

```
}

private inner class TestResourceManager : ResourceManager {
    val message: String = ""
    override fun getString(stringResId: Int) = message
}
```

А я напомню суть тестовых классов – они должны быть созданы таким образом чтобы мы могли легко получать те данные от него, какие мы хотим заранее указав их.

Т.е. как мы можем использовать этот последний класс – перед тем как будет вызван метод getString мы засетим значение message = “a” и после того как у модели будет вызван метод который работает с методом ресурсменеджера мы проверим значение сравнив с этим.

Предлагаю вам написать тесткейсы для всего остального.

И да, забыл сказать – чтобы запустить юнит тест на корутинах вам нужно добавить runBlocking

```
@Test
fun test_change_data_source(): Unit = runBlocking {
```

Ведь мы не можем запускать корутины не в саспенд функциях и не в скоупах, значит нам нужно что-то одно. Тестовый метод не может быть саспенд. Значит второе.

```
@Throws(InterruptedException::class)
public fun <T> runBlocking(context: CoroutineContext = EmptyCoroutineContext, block: suspend CoroutineScope.() -> T): T {
    contract { this: ContractBuilder
```

Всегда проваливайтесь внутрь метода и смотрите что оно делает. Изучайте исходники, это лучше всяких документаций, ведь над каждым методом и классом есть джавадок.

И пишите юнит тесты осмысленные, соответствующие реальному сценарию использования.

Далее мы поговорим о том, как писать тесткейсы и юай тесты чтобы быть уверенней в нашем коде и не тратить дорогие человекочасы на такие простые вещи как проверки. Удачи!