

# Списки данных

## Как отобразить много данных

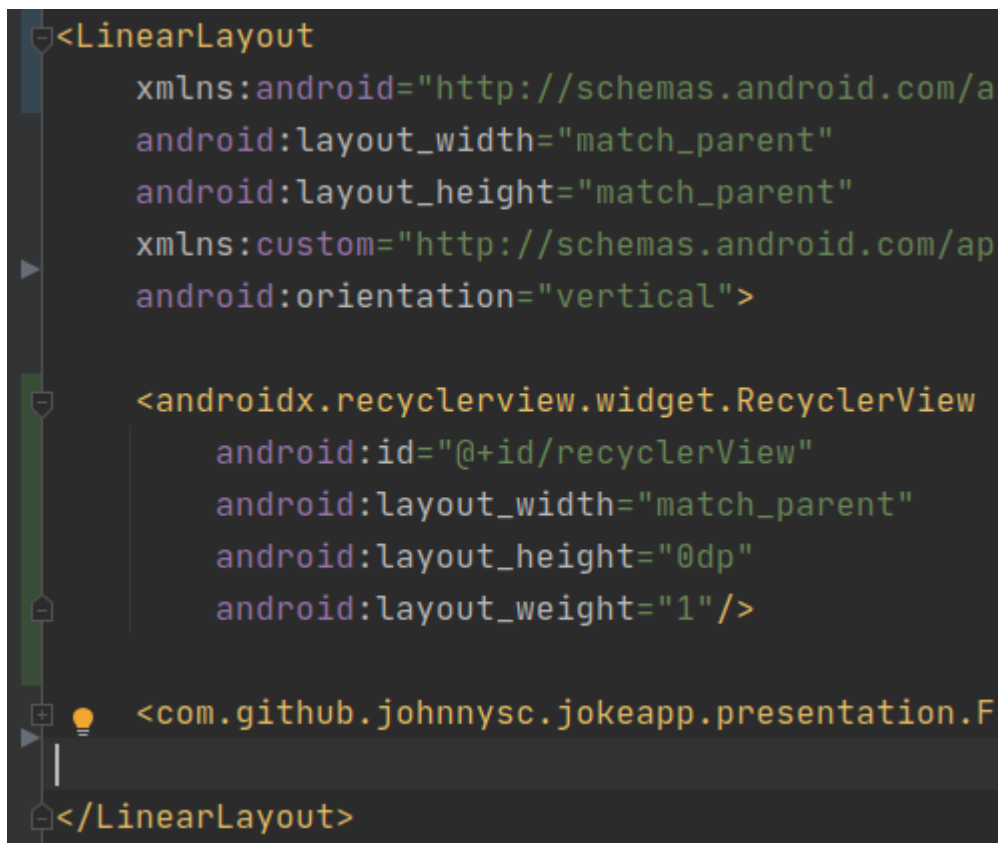
До сих пор наше приложение работало по одной логике: получаем данные из сети и можем сохранять в кеш и удалять из него. Но у нас одно но: мы показываем рандомный элемент из избранных. Хотелось бы иметь возможность просмотреть сразу весь список избранных.

Кстати говоря ровно так и звучит бизнес логика : Я как юзер хочу иметь возможность просмотреть весь список избранных шуток/цитат.

И давайте начнем с простого: сделаем отображение списка шуток и метод получения шуток на одном экране. После уже будем думать над тем как нам сделать отображение и шуток и цитат. Цель текущей лекции познакомить вас со списком, а не сразу со всем.

Итак, давайте для начала отредактируем xml. Наш экран будет выглядеть следующим образом : блок где можно получить шутки и список. Мы уже познакомились с простыми виджетами – TextView, ImageView, CheckBox, ProgressBar, ImageButton, Button. И теперь настал момент вам рассказать про RecyclerView. Некоторые могут сказать – но есть же ListView. Но для конкретной задачи подходит RecyclerView и я позже объясню почему. Но сразу скажу о том, что с ListView будет неудобно работать, потому что наш список избранных динамически будет меняться – мы как будем показывать данные, так и добавлять в них на лету и удалять из списка.

Итак, поехали. Открываем xml файл активности и добавляем туда список



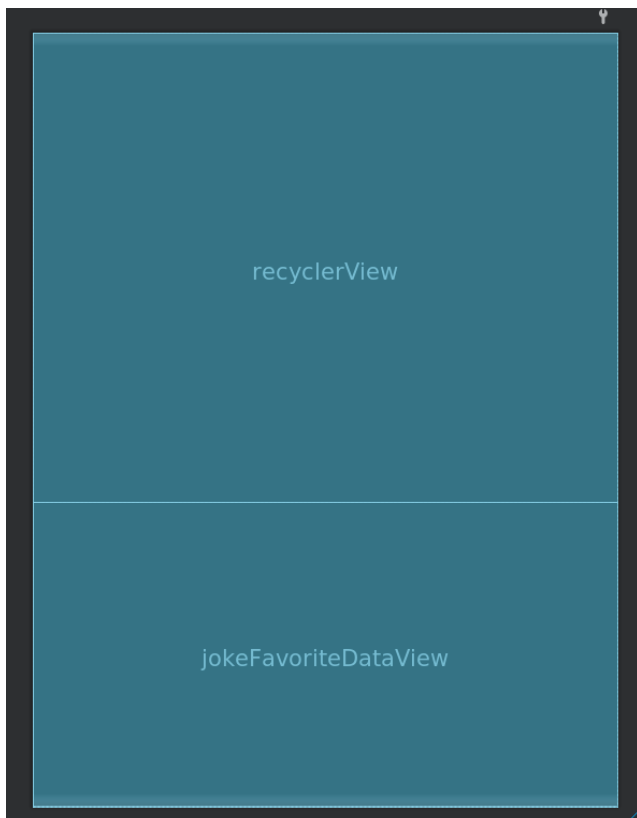
```
<LinearLayout
    xmlns:android="http://schemas.android.com/a
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    xmlns:custom="http://schemas.android.com/ap
    android:orientation="vertical">

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/recyclerView"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"/>

    <com.github.johnnysc.jokeapp.presentation.F

</LinearLayout>
```

Я оставил блок для шуток и схлопнул чтобы показать RecyclerView (далее ресайклер).



Я поставил вес 1 для ресайклера и высоту 0 чтобы во-первых было удобно кликать на кнопку на дне экрана и смотреть список сверху. Делайте всю коммуникацию внизу экрана чтобы юзеру было удобно тапать большим пальцем руки. О дизайне и их поговорим в последующих лекциях.

И я забыл сразу в разметке указать каким образом будет отображаться список (это самая частая ошибка всех, особенно новичков). Суть в том, что список может быть как горизонтальной ориентации, так и вертикальной. А еще можно показывать не в 1 столбец, а в несколько. Поэтому мы указываем какой конкретно layoutManager у ресайклера. Это можно сделать как в коде, так и в хмл. Давайте в хмл

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/recyclerView"
    app:layoutManager="androidx.recyclerview.widget.GridLayoutManager (androidx
    android:layout
    android:layout
    android:layout
```

И да, я убрал custom неймспейс и заменил на appns. Как видите можно сразу указать способ отображения списка – выбираем первый вариант и идем дальше

Теперь, так как у нас список данных то мы матчим не к одной вью, а к списку из вью. И под это дело нам нужен патерн адаптер. Благо Андроид уже написал свой абстрактный адаптер и мы просто отнаследуемся от него. Но пока что я объясню в чем суть патерна адаптер.

Итак, у вас есть список данных – например `List<String>` и вам нужно спаять его к ресайклу в котором текстовка `TextView` и что-то еще. Каким образом строка из котлин превратится в текстовку андроид? С помощью адаптера. Простой мост между одними данными и другими.

Но я бы даже сказал что не простой мост, а именно адаптер. Вспомните адаптер которым вы пользуетесь когда заряжаете смартфон – вы втыкаете его в розетку где 220вольт и он преобразовывает это все в подходящий смартфону 1.5А или 2А. Ведь если бы адаптера не было, то ваш смартфон бы сгорел сразу. Итак, нам нужен преобразователь данных для нашего случая это будет JokeUiModel к некоему юай элементу. Для начала давайте наш элемент списка будет состоять лишь из 1 текстовки. Мы покажем шутку но даже не всю, а лишь первую линию и конце просто многоточие поставим (если не знаете как посмотрите первую лекцию по андроид). Далее юзер если захочет прочитать всю шутки то кликнет на нее и вся она покажется там, где у нас отображается из сети.

Напишем код в активити

```
val recyclerView = findViewById<RecyclerView>(R.id.recyclerView)
recyclerView.
    [] (index: Int) for ViewGroup in androidx.core.view
    adapter (from getAdapter()/setAdapter()) RecyclerView.Adapter<raw> RecyclerView.ViewHolder!>?
    get(index: Int) for ViewGroup in androidx.core.view
    compatAccessibilityDelegate (from getCompatAccessibilityDel... RecyclerViewAccessibilityDelegate?
    edgeEffectFactory (from getEdgeEffectFactory()/setEdgeEffectFac... RecyclerView.EdgeEffectFactory
    isAnimating (from isAnimating()) Boolean
    isComputingLayout (from isComputingLayout()) Boolean
    itemAnimator (from getItemAnimator()/setItemAnimator()) RecyclerView.ItemAnimator?
    itemDecorationCount (from getItemDecorationCount()) Int
    layoutManager (from getLayoutManager()/setLayoutManager()) RecyclerView.LayoutManager?
    layout(l: Int, t: Int, r: Int, b: Int) Unit
    maxFlingVelocity (from getMaxFlingVelocity()) Int
    minFlingVelocity (from getMinFlingVelocity()) Int
    onFlingListener (from getOnFlingListener()/setOnFlingListener()) RecyclerView.OnFlingListener?
    preserveFocusAfterLayout (from getPreserveFocusAfterLayout()/setPreserveFocusAfterLayo... Boolean
    recycledViewPool (from getRecycledViewPool()) RecyclerView.RecycledViewPool
    Press Enter to insert. Tab to replace
```

Как видите с ресайклером можно сделать следующие вещи – и первым (почти) идет именно установка адаптера. И тип у нее RecyclerView.Adapter. Теперь, зная джава и котлин вы точно понимаете, что адаптер у ресайкла это абстрактный класс вложенный в класс ресайкла.

И нам нужно написать конкретную реализацию. И раз уж у нас дженерик общий тип для данных, то переиспользуем его.

```
class CommonDataRecyclerViewAdapter<E>() :
    RecyclerView.Adapter<CommonDataRecyclerViewAdapter<E>.CommonDataViewHolder<E>>() {
    private val list: List<CommonDataModel<E>> = ArrayList()

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): CommonDataViewHolder<E> {
        TODO( reason: "Not yet implemented")
    }

    override fun onBindViewHolder(holder: CommonDataViewHolder<E>, position: Int) {
        TODO( reason: "Not yet implemented")
    }

    override fun getItemCount(): Int {
        TODO( reason: "Not yet implemented")
    }

    inner class CommonDataViewHolder<E>(view: View) : RecyclerView.ViewHolder(view) {
        // TODO: 22.06.2021
    }
}
```

Не надо пугаться этого кода. Вы будете почти каждый божий день на работе писать такой же.

Итак, вы наследуетесь от абстрактного класса адаптера ресайкла и у вас 3 метода которые нужно определить. Итак, я говорил, что мы мапим данные из котлин классов в вью. Но у нас непонятно какая вью и специально для этого мы предоставляем класс вьюхолдера. Он в конструктор принимает некое вью. Нет, писать кастомвью не нужно, хотя и можно. Мы задействуем первый метод адаптера: создаем в нем вьюхолдер. Далее нам нужно собственно смэпить данные из списка (я создал приватное поле и инициализировал его сразу) к вьюхолдеру. И здесь нам поможет второй метод – холдер получаем в аргументе метода и позицию чтобы достать из списка нужные данные. Далее у нас просто метод для получения количества. У вас может быть список из конкретного числа элементов и зная это андроиду будет проще создать ресайкл. В коде кстати сразу напишете `hasFixedSize`, но у нас не этот случай. Ладно, давайте от слов к делу. Начнем заполнять тудушки.

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): CommonDataViewHolder<E> {  
    val view = LayoutInflater.from(parent.context).inflate(R.layout.common_data_item, parent, false)  
    return CommonDataViewHolder(view)  
}
```

Через инфлейтер распарсим хмл и создаем вьюхолдер. Кстати, заметьте, у нас в ресайкладаптере доступен контекст.

**Запомните! Никогда не прокидывайте контекст внутрь адаптера ресайклера! Он там и так уже есть, просто берите и используйте! Каждая вью имеет доступ к контексту.**

Второй аргумент инфлейта это родительская вью и мы указываем ее. Это нужно для понимания в какой вью инфлейтим новое вью. На самом деле вы можете посмотреть исходники этого метода и прочитайте джавадоки сами. Но давайте посмотрим на исходники метода `LayoutInflater.from`. Почему это не конструктор? А потому что порождать новый объект инфлейтера который парсит хмл было бы слишком затратно для системы и поэтому там публик статик метод который работает с закешированным объектом. Сами гляньте код и вы убедитесь в этом.

```
public static LayoutInflater from(Context context) {  
    LayoutInflater inflater =  
        (LayoutInflater) context.getSystemService(Context.LAYOUT_INFLATER_SERVICE);  
    if (inflater == null) {  
        throw new AssertionError("LayoutInflater not found.");  
    }  
    return inflater;  
}
```

Именно поэтому я настоятельно рекомендую не писать самому `context.getSystemService`.

Ладно, идем дальше. Нам нужно смэпить данные к вью из вьюхолдера, но мы не создали разметку. Давайте там будет пока что одна текстовка



Проверяйте в дизайне что высота элемента не заполняет все окно!

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/commonDataTextView"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:ellipsize="end"
    android:maxLines="1"
    android:padding="@dimen/padding"
    android:textAppearance="@style/TextAppearance.AppCompat.Body2"
    tools:text="some long text here that should break into dots at the end of line you know"
```

Toolsns напишите и сразу АС предложит автозаполнение. Очень удобно сразу в превью видеть как будет выглядеть текстовка.

**Запомните навсегда – как только создали разметку для айтема ресайкла – ставьте высоту wrap\_content.**

Типичная ошибка всех и особенно новичков – забыть поменять высоту с match\_parent. Здесь я сразу использовал стиль и отступы чтобы все не сливалось в списке.

Идем дальше и пишем второй метод в адаптере

```
override fun onBindViewHolder(holder: CommonDataViewHolder<E>, position: Int) {
    holder.bind(list[position])
}
```

**Запомните, этот метод всегда должен выглядеть именно так и никак не иначе! 1 линия!**

Метод получения количества максимально примитивен

```
override fun getItemCount() = list.size
```

И последнее что нам нужно сделать – дописать метод bind в ViewHolder классе

Так, стоп. У нас класс CommonDataModel лишь 1 метод map которому нужен мапер и так далее. Нам нужно заменить TextView в разметке на CorrectTextView

И давайте добавим метод в класс для работы с интерфейсом отображения текста напрямую. Мы будем передавать лишь первый текст и нам в элементах списка не нужны состояния.

```
class CommonDataModel<E>(
    private val id: E,
    private val firstText: String,
    private val secondText: String,
    private val cached: Boolean = false
) : ChangeCommonItem<E> {

    fun map(showText: ShowText) = showText.show(firstText)
```

И теперь можем написать метод у вьюхолдера

```
inner class CommonDataViewHolder<E>(view: View) : RecyclerView.ViewHolder(view) {
    private val textView = itemView.findViewById<CorrectTextView>(R.id.commonDataTextView)
    fun bind(model: CommonDataModel<E>) = model.map(textView)
}
```

Вьюхолдеру доступен itemView : это корневой элемент разметки – в нашем случае это и есть текстовка. Можем позже проверить кастанув. Но общая логика – через нее получать вью по айди. И метод мап вызываем у модели и передадим туда текстовку. Вуаля. Все очень просто. Но подождите, у нас же список данных приватный. Как мы передадим список в адаптер? Действительно, нам нужен метод для этого и мы его сейчас напишем. Просто если помните у нас список избранных получается из реалма и это может занять время – поэтому мы запустим корутины и она отдаст через ливдату данные. Напишем метод уже

```
private val list = ArrayList<CommonDataModel<E>>()

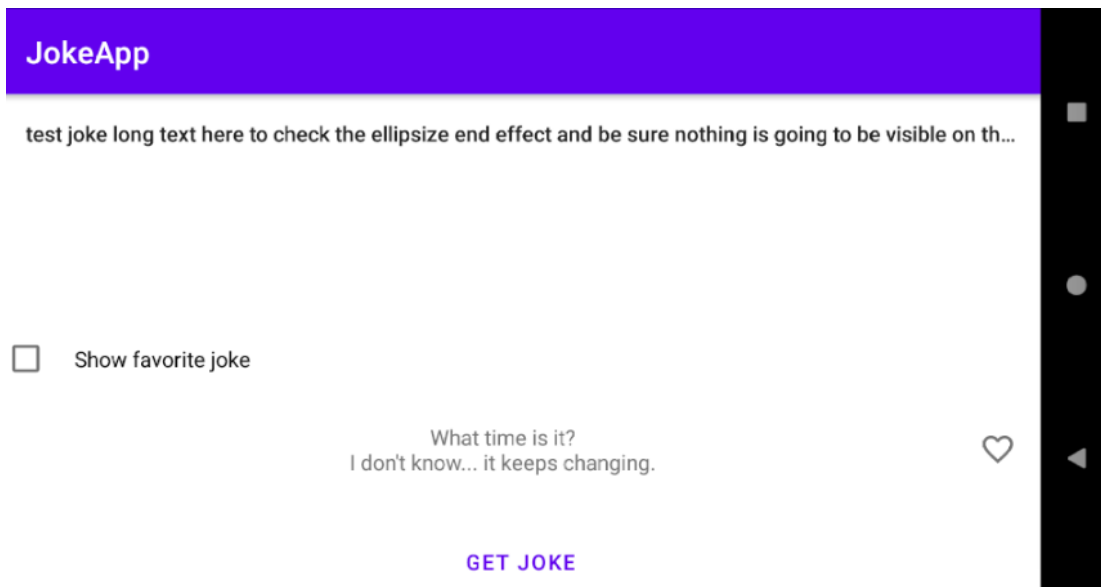
fun show(data: List<CommonDataModel<E>>) {
    list.clear()
    list.addAll(data)
    notifyDataSetChanged()
}
```

Во-первых тип должен быть сразу ArrayList чтобы мы могли очищать список. Но секунду. У нас же и так был пустой список. Ну да, на старте. А после? Как работает ливдата – при повороте она достает данные и постит наблюдателю еще раз (сейчас я показываю вам как делают все и после попробуем переписать красиво). После очистки списка я добавляю все элементы и вызываю метод notifyDataSetChanged() - чтобы адаптер заметил изменения и заново вызвал нужные методы для мапинга данных. Ведь как работает ресайклер – он при создании берет из списка данные – создает вьюхолдеры и мапит через onBindViewHolder. И если на старте список был пуст, то и ничего не будет видно на экране. А после когда данные пришли из корутины нужно обновить список и заново запустить все методы – для этого вызываем метод нотифай. Ладно, давайте теперь для теста попробуем закинуть тестовые данные в ресайклер и проверим что все работает. В активит напишем код

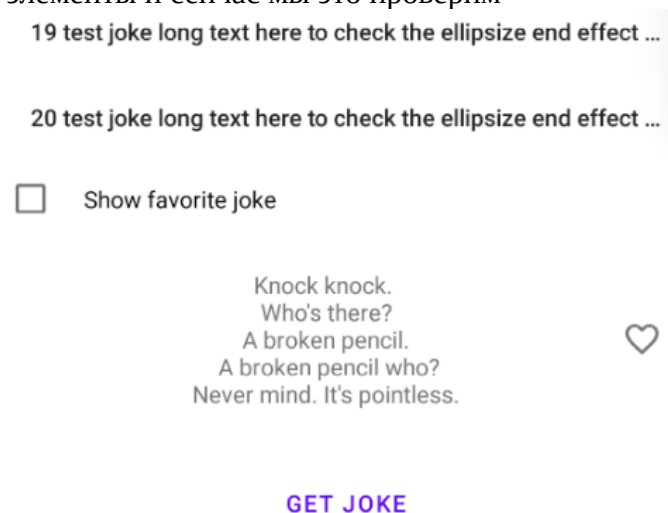
```
val recyclerView = findViewById<RecyclerView>(R.id.recyclerView)
val adapter = CommonDataRecyclerViewAdapter<Int>()
recyclerView.adapter = adapter
adapter.show(
    listOf(
        CommonDataModel(
            id: 0,
            firstText: "test joke long text here to check the ellipsi
            secondText: "not visible text",
            cached: true
        )
    )
)
```

Кстати, если вас раздражает каждый раз передавать <Int> напишите класс наследник class JokeModel : CommonDataModel<Int> и все.

Ладно, давайте запустим уже проект и проверим что все работает



Есть 2 причины почему скриншот в ландшафтной ориентации – во-первых так меньше места занимает в лекции и во-вторых : сразу проверил что даже так текст не вмещается. Но мы создавали список не для 1 элемента, а для многих. Давайте напишем цикл для тестовых данных и проверим что все работает. Суть ресайклера чтобы можно было скролить элементы и сейчас мы это проверим



Заметьте, что когда от сервера получаем новую шутку и блок меняет высоту то и ресайклер меняет свою высоту. Все работает просто замечательно. Но на тестовых данных. Давайте напишем метод получения данных из реалма. И нам придется добавить несколько методов во все слои. Начнем пожалуй с кешдатасорса

```
interface DataFetcher<E> {
    suspend fun getData(): CommonDataModel<E>
    suspend fun getDataList(): List<CommonDataModel<E>>
}
```

Кешдатасорс наследует DataFetcher, но кроме него наследуют и все клаудДатасорсы, так что так делать не нужно. Потому что из сети мы не получаем список, а лишь 1 элемент. Ну и мы нарушаем так же SOLID I, так что не надо добавлять второй метод в интерфейс. Давайте напишем его прямо в CacheDataSource. Далее если понадобится, то мы сделаем иначе



```
interface CacheDataSource<E> : DataFetcher<E>, ChangeStatus<E> {
    suspend fun getDataList(): List<CommonDataModel<E>>|
```

И теперь метод доступен лишь в кешдатасорсах – ок, давайте напишем общий метод в базовом классе

```
override suspend fun getDataList(): List<CommonDataModel<E>> {
    realmProvider.provide().use { it: Realm
        val list = it.where(dbClass).findAll()
        if (list.isEmpty())
            throw NoCachedDataException()
        else
            return list.map { item ->
                realmToCommonDataMapper.map(item)
            }
    }
}
```

Вам не кажется что мы уже писали такое? Да, я скопипастил код из метода getData. Там точно так же получали все элементы и выбирали случайный. А здесь мапим каждый элемент из реалма к общей модели. Но вся остальная логика же одинакова, можно не нарушать DRY?

```
private fun getRealmData() : RealmResults<T> {
    realmProvider.provide().use { it: Realm
        val list = it.where(dbClass).findAll()
        if (list.isEmpty())
            throw NoCachedDataException()
        else
            return list
    }
}
```

Вынесли дублирующий код, теперь можем переписать методы в 1 линию

```
override suspend fun getDataList() = getRealmData().map { realmToCommonDataMapper.map(it) }
override suspend fun getData() = realmToCommonDataMapper.map(getRealmData().random())
```

Вуаля! Кстати, а нам ведь больше не нужен метод получения 1 случайного элемента. Так что мы можем удалить и вовсе этот код, но это потом. Когда будем переписывать репозиторий. Идем дальше. Нам нужен метод в репозитории специально для списка данных.

```
interface CommonRepository<E> {
    suspend fun getCommonItem(): CommonDataModel<E>
    suspend fun getCommonItemList(): List<CommonDataModel<E>>|
```



И сразу скажу про нейминг – есть плохое решение писать `getCommonItems`, но я же предпочитаю писать слово `List` потому что в таком случае меньше шансов спутать методы. Теперь можем написать реализацию в репозитории

```
override suspend fun getCommonItemList(): List<CommonDataModel<E>> = withContext(Dispatchers.IO) {
    cacheDataSource.getDataList()
}
```

Всего в 1 линию. Да! Переключаем поток и выбираем у кешдатасorsa метод. Далее в интеракторе обработаем ошибку, ведь если в кеше пусто, будет ошибка. Поехали

```
interface CommonInteractor {
    suspend fun getItem(): CommonItem
    suspend fun getItemList() : List<CommonItem>
}
```

Так, стоп. У нас же `CommonDataModel` это модель датаслая, а мы переиспользуем ее в адаптере ресайкла! Так не пойдет. Давайте будем просто идти от слоя к слою по аналогии с уже существующим кодом. Я добавил метод и мне не надо ни о чем думать, был 1 элемент, теперь список элементов. Я сюда не передам модель датаслая. Давайте посмотрим на реализацию интерактора, надо написать хороший метод для списка

```
override suspend fun getItemList(): List<CommonItem> {
    return try {
        repository.getCommonItemList().map { it: CommonDataModel<E>
            it.map(mapper)
        }
    } catch (e: Exception) {
        listOf(CommonItem.Failed(failureHandler.handle(e)))
    }
}
```

Спасибо котлину за удобные методы мапинга. У нас уже есть все нужные классы для работы с 1 айтемом. Для списка просто используем `map` и готово. Идем дальше. Интерактор работает в вьюмодели. Значит надо написать код и там

```
interface CommonViewModel {
    fun getItem()
    fun getItemList()
}
```

Видите как просто? Нужно просто было сразу начинать с конца – с датаслая, а не с юай слоя. Но ничего страшного, мы не так много кода написали чтобы переписывать его нужно было долго. Теперь пофиксим реализацию вьюмодели

```
override fun getItemList() {
    viewModelScope.launch(dispatcher){ this: Coroutine
        interactor.getItemList()
    }
}
```

И здесь нас ожидает очередная сложность. Раньше мы просто получали айтем у интерактора и мапили к юай слою и вуаля – через комуникатор показывали на юай. Но теперь же у нас список из айтемов бизнес-слоя. Как же нам смापить `List`? Экстеншн функции!

```
fun List<CommonItem>.toUiList() = map { it.to() }
```

Метод мапинга to() был описан у элемента. Мы мапим список к списку и у каждого айтема вызываем метод преобразования к юай слою. И далее у нас проблема с комуникатором. Но я предлагаю написать полноценный метод у самого коммуникатора который примет список аргументом

```
interface Communication {  
    fun showState(state: State)  
    fun showDataList(list: List<CommonUiModel>)
```

И теперь можем вызвать метод в вьюмодели

```
override fun getItemList() {  
    viewModelScope.launch(dispatcher){ this: CoroutineScope  
        communication.showDataList(interactor.getItemList().toUiList())  
    }  
}
```

Теперь нам нужна реализация для комуникатора. Пофиксим код там?

```
private val listLiveData = MutableLiveData<List<CommonUiModel>>()  
override fun showDataList(list: List<CommonUiModel>) {  
    listLiveData.value = list  
}
```

Да, нам нужна новая ливдата внутри реализации комуникатора. И еще нам нужен метод наблюдения за изменениями ливдаты.

```
fun showDataList(list: List<CommonUiModel>)  
fun observe(owner: LifecycleOwner, observer: Observer<State>)  
fun observeList(owner: LifecycleOwner, observer: Observer<List<CommonUiModel>>)
```

Итак, в интерфейсе уже 5 методов. Вам не кажется что это неправильно? По-хорошему нужно для каждой ливдаты иметь свой класс, но пока оставим так, если почувствуем непреодолимое желание переписать код это будет легко сделать. Хотя я уже чувствую.

```
interface ListCommunication {  
    fun showDataList(list: List<CommonUiModel>)  
    fun observeList(owner: LifecycleOwner, observer: Observer<List<CommonUiModel>>)  
}
```

SOLID ISP лучше 2 интерфейса по 2-3 метода чем 1 на 5 методов. А реализацию давайте оставим какая она есть, но напишем 1 интерфейс который имплементит оба

```
interface CommonCommunication : Communication, ListCommunication
```

И в вьюмодель напишем именно этот интерфейс чтобы работать со всеми методами

```
class BaseViewModel(  
    private val interactor: CommonInteractor,  
    private val communication: CommonCommunication,
```

Теперь можем написать реализацию

```
class BaseCommunication : CommonCommunication {
```

И метод в классе будет выглядеть так

```
    override fun observeList(owner: LifecycleOwner, observer: Observer<List<CommonUiModel>>) {  
        listLiveData.observe(owner, observer)  
    }
```

Добавим метод в интерфейс вьюмодели теперь CommonViewModel

```
    fun observe(owner: LifecycleOwner, observer: Observer<State>)  
    fun observeList(owner: LifecycleOwner, observer: Observer<List<CommonUiModel>>)
```

Фиксим реализацию вьюмодели в 1 линию

```
    override fun observeList(owner: LifecycleOwner, observer: Observer<List<CommonUiModel>>) =  
        communication.observeList(owner, observer)
```

И напишем уже код для активити в конце концов

```
recyclerView.adapter = adapter  
viewModel.observeList(owner: this, { list ->  
    adapter.show(list)  
})
```

И да, у нас не тот тип списка принимает адаптер, пофиксим адаптер

```
class CommonDataRecyclerAdapter :  
    RecyclerView.Adapter<CommonDataRecyclerAdapter.ViewHolder>()  
  
    private val list = ArrayList<CommonUiModel>()  
  
    fun show(data: List<CommonUiModel>) {  
        list.addAll(data)  
    }
```

И да, можно убрать дженерики. Видите, даже если вы не помните (как я) что датамодель она для датаслая, то спустя 10 минут сама структура проекта вам скажет что нужно делать. И если помните, мы написали метод мапинга у датамодели, нужно переместить в юаймодель

```
    open fun show(communication: Communication) = communication.showState(  
        State.Initial(text(), getIconResId())  
    )  
  
    fun show(showText: ShowText) = showText.show(first)
```

И спокойно вызываем в вьюхолдере внури адаптера

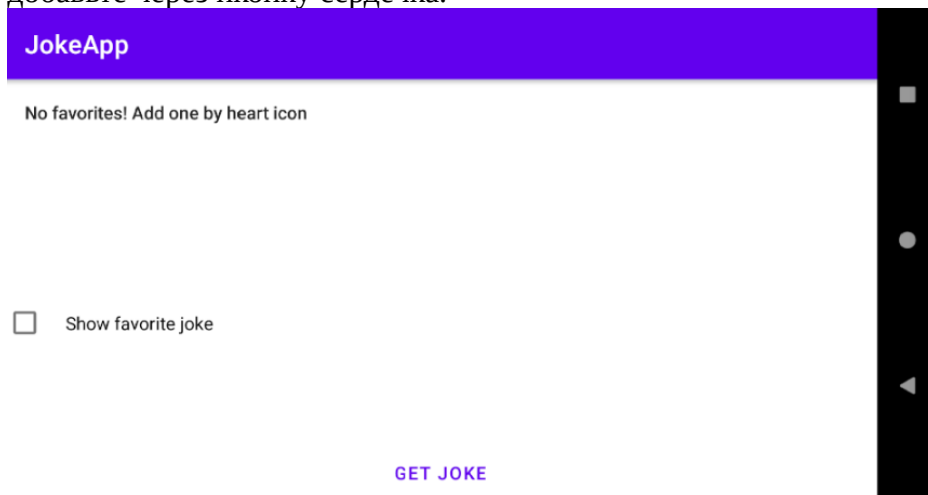
```
    inner class CommonDataViewHolder(view: View) : RecyclerView.ViewHolder(view) {  
        private val textView = itemView.findViewById<CorrectTextView>(R.id.commonDataTextView)  
        fun bind(model: CommonUiModel) = model.show(textView)  
    }
```

Запустим уже код! Все должно работать.

Стоп! Запустился проект. Добавляю в избранное элементы, но ничего не происходит! Почему же так? Ну во-первых мы забыли вызвать метод у вьюмодели. После инициализации ресайклера нужно вызвать метод у вьюмодели

```
recyclerView.adapter = adapter
viewModel.observeList( owner: this, { list ->
    adapter.show(list)
})
viewModel.getItemList()
```

Теперь запустим еще раз и вуаля! Там где список видим текст ошибки : нет избранных – добавьте через иконку сердечка.



Давайте добавим в избранные несколько штук и посмотрим что будет. Эм.. ничего? Да. Почему же так? А потому что метод получения всех шуток вызывается только на старте. Переверните девайс и увидите... эм. Что-то пошло не так. Дебажим: ставим брейкпойнт в интеракторе и видим ошибку : this realm instance is already closed. Почему же так? Я неправильно написал метод в кешдатасорсе. Реалм закрывается раньше чем мы возвращаем список. Т.е. метод возвращает список и потом он мапится к другим моделям. Давайте пофиксим это с помощью дженерика и лямбды

```
override suspend fun getDataList() = getRealmData { results->
    results.map { realmToCommonDataMapper.map(it) }
}
override suspend fun getData() = getRealmData { it: RealmResults<T>
    realmToCommonDataMapper.map(it.random())
}

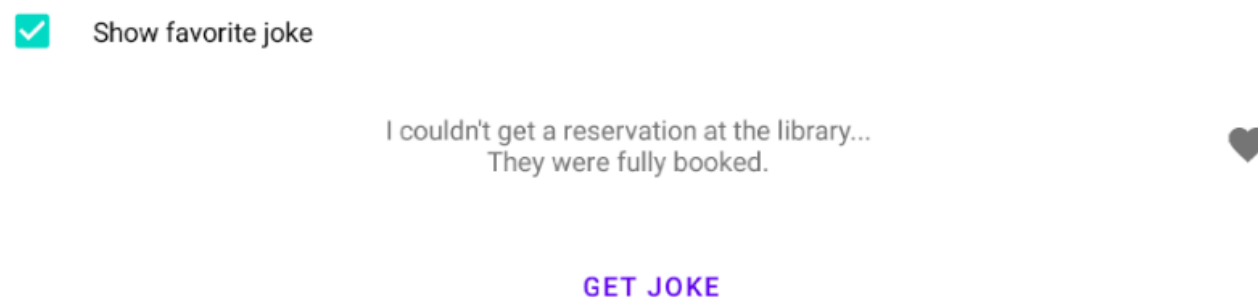
private fun <R> getRealmData(block: (list: RealmResults<T>) -> R) : R {
    realmProvider.provide().use { it: Realm
        val list = it.where(dbClass).findAll()
        if (list.isEmpty())
            throw NoCachedDataException()
        else
            return block.invoke(list)
    }
}
```

Итак, мы не можем просто вернуть список реалм объектов потому что сразу закрываем инстанс, значит нам нужно выполнить код в этом месте. Но какой? Даем лямбду и вызываем метод у нее. В одном случае мапим рандом, в другом список. Я написал просто дженерик.

Если вам сложно читать такой код, то удалите getItem и просто напишите сразу. И теперь запустим код и увидим результат



Не удивляйтесь что у первой шутки троеточие. Оно в оригинале такое. У нас все еще есть возможность посмотреть рандомную шутку из избранных и я проверил



Да, давайте не будем лишать возможности (пока что) читать рандом шутку из избранных. Но толку от нашего списка не так много, не считаете?

Список отображает лишь первый элемент из 2. С ним ничего невозможно сделать, с ним никак нельзя взаимодействовать. Самое простое что мы можем сделать это поменять метод мапинга чтобы показывать не 1 линию, а сразу обе. Перепишем код

```
abstract class CommonUiModel(private val first: String, private val second: String) {  
    protected open fun text() = "$first\n$second"  
    fun show(showText: ShowText) = showText.show(text())  
}
```

У нас же есть метод конкатенации. Но мы же написали в разметке элемента что у нас 1 линия. Ну давайте уберем это и еще мне не нравится стиль. Поменяем его

android:textAppearance="@style/TextAppearance.AppCompat.Small"

И теперь все выглядит более менее. Хотя теперь у нас другая проблема – список получаем один раз на старте приложения и он не меняется за все время работы. Т.е. если мы добавим в избранное новую шутку или удалим из избранных шутку то в списке все равно будут старые данные пока юзер не перевернет экран. Можете сами проверить это.

Было бы классно в реальном времени добавлять шутки и удалять их.

Как это сделать? Давайте рассмотрим в следующей лекции. А пока можете самостоятельно переписать код таким образом, чтобы с цитатами работало точно так же.

## JokeApp

What happens to a frog's car when it breaks down?  
It gets toad away

Why did the butcher work extra hours at the shop?  
To make ends meat.

I couldn't get a reservation at the library...  
They were fully booked.

Bad at golf?  
Join the club.

Can I watch the TV?  
Yes, but don't turn it on.

How many bones are in the human hand?  
A handful of them.



Show favorite joke

What happens to a frog's car when it breaks down?  
It gets toad away



GET JOKE