RecyclerView

как работает

В этой лекции мы подробно остановимся на ресайклере и на его методах и посмотрим как еще можно его использовать.

Для начала скажем что в предыдущей лекции мы добавили ресайклер в наш юай и списком отображали все избранные элементы. Но мы не учли один момент : мы получали список и показываем его, но у нас есть возможность менять содержимое списка извне. С помощью сердечка в блоке кастомвью. Т.е. сейчас мы можем поставить галку на “Показать избранные” и нажимать на сердечко и удалить все избранные, а список так и останется каким он был. Или же наоборот : добавить в избранное новую шутку, тогда как список обновляется лишь один раз на старте и второй раз при повороте. Я бы не хотел поворачивать устройство чтобы иметь актуальные данные. Я бы хотел в реальном времени менять содержимое списка.

Но перед тем как мы это сделаем давайте залогируем все методы в адаптере и добавим в список избранных побольше элементов. Например 30 штук. Откройте приложение и добавьте около 30 шуток. После чего залогируем методы в адаптере следующим образом

```
private var onCreateViewHolderCallsCount = 0

override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): CommonDataViewHolder {
    val view =
        LayoutInflater.from(parent.context).inflate(R.layout.common_data_item, parent,  attachToRoot: false)
    onCreateViewHolderCallsCount++
    Log.d(
        tag: "DataRecyclerAdapterTAG",
        msg: "onCreateViewHolderCallsCount $onCreateViewHolderCallsCount"
    )
    return CommonDataViewHolder(view)
}

private var onBindViewHolderCallsCount = 0

override fun onBindViewHolder(holder: CommonDataViewHolder, position: Int) {
    onBindViewHolderCallsCount++
    Log.d( tag: "DataRecyclerAdapterTAG",  msg: "onBindViewHolderCallsCount $onBindViewHolderCallsCount")
    holder.bind(list[position])
}
```

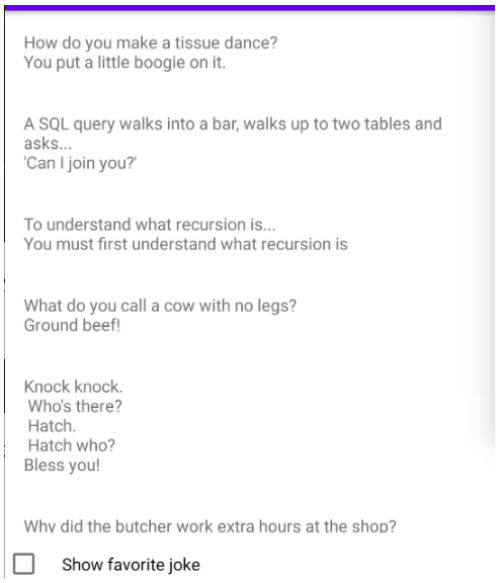
Просто создадим счетчик вызовов первого метода и второго. Кладем в избранное 30 штук и запускаем проект

```
2021-06-30 10:00:52.541 32123-32123/com.github.johnnysc.jokeapp D/DataRecyclerAdapterTAG: onCreateViewHolderCallsCount 6
2021-06-30 10:00:52.541 32123-32123/com.github.johnnysc.jokeapp D/DataRecyclerAdapterTAG: onBindViewHolderCallsCount 6
```

Итак, что у нас происходит : 6 раз вызвался и первый метод и второй. Почему именно 6?

Потому что на моем эмуляторе поместилось 6 вью. И поэтому ресайклер адаптер создал 6 вьюхолдеров и вызвал 6 раз метод байнда данных – т.е. в область ресайклервью поместилось 6 вью и для них всех вызвалось 6 раз метод onBind. Вроде как логично. И у нас 30 элементов

в списке и вот в этом вся суть ресайклера. Ресайклер не создает больше вьюх чем он может показать в той области в который его положили



И давайте просто проскролим список вниз до конца, что должно произойти?

```
2021-06-30 10:05:38.566 32123-32123/com.github.johnnysc.jokeapp D/DataRecyclerAdapterTAG: onBindViewHolderCallsCount 30
```

Мы видим что метод связывания данных вызвался 30 раз. Просто потому что у нас список из 30 элементов и нужно вызвать метод этот чтобы отобразить информацию. Ок, а что насчет OnCreateViewHolder? Он тоже вызвался 30 раз? Конечно же нет!

```
2021-06-30 10:05:38.449 32123-32123/com.github.johnnysc.jokeapp D/DataRecyclerAdapterTAG: onCreateViewHolderCallsCount 12
```

Всего 12! Почему 12? Загадка? Может 12 просто потому что 6*2?

Если вы и дальше будете скролить вверх и вниз то увидете как растет число вызовов onBindViewHolder, но количество вызовов onCreateViewHolder по-прежнему остается 12. Как же так? Еще раз – из названия можно понять что ресайклер переиспользует вью, которые создает, точнее вьюхолдеры. Если бы у нас был список не из 30 элементов, а например из 3000, то андроид система бы не смогла столько вью сгенерировать и под них вьюхолдеры и держать все в памяти. Ровно так работает ListView – сразу генерирует нужное количество вью под список данных и потому не рекомендуется к использованию если в списке много элементов. Можете проверить это сами: перепишите код на ListView и увидете в какой момент крашится приложение.

Теперь мы сделаем здесь один важный пойнт. На собеседованиях вас могут спросить об этом: сколько вьюхолдеров создается при использовании ресайклервью. И мейнстрим ответ – сколько влезает плюс 2. Как видите по факту это не (всегда) так. Правильный ответ: сколько ресайкладаптеру понадобится создать. В нашем случае это количество видимых умноженное на 2. Почему так? Нужно посмотреть исходный код ресайкла.

Called when RecyclerView needs a new **RecyclerView.ViewHolder** of the given type to represent an item.

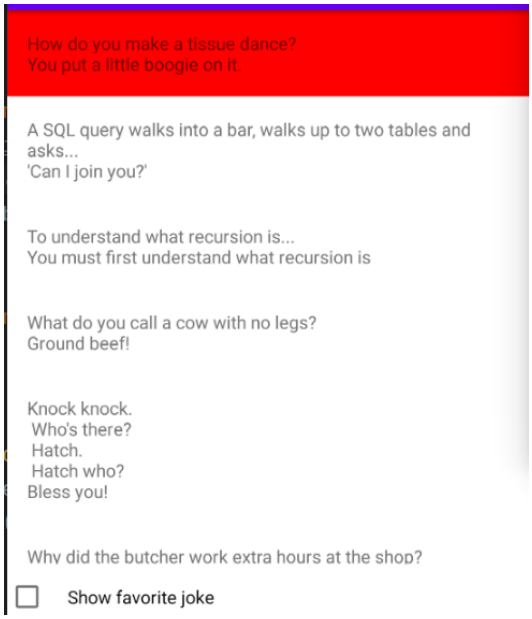
Вот вырезка из https://developer.android.com/reference/androidx/recyclerview/widget/RecyclerView.Adapter#onCreateViewHolder(android.view.ViewGroup,%20int)

Как видите здесь не сказано : количество видимых элементов плюс 2. Нет. Сколько понадобится. Итак, давайте подумаем, почему может понадобиться в 2 раза больше вьюх чем может показать сразу? Не утруждайте себя. Или читайте исходники или гуглите вопрос.

Итак, зачем нам это все сейчас? Я просто хочу вас предостеречь от плохого кода который порождает проблемы на ровном месте. Давайте я создам проблему и вы увидете в чем она заключается.

```
fun bind(model: CommonUiModel) {
    model.show(textView)
    if (onBindViewHolderCallsCount == 1) {
        textView.setBackgroundColor(Color.RED)
    }
```

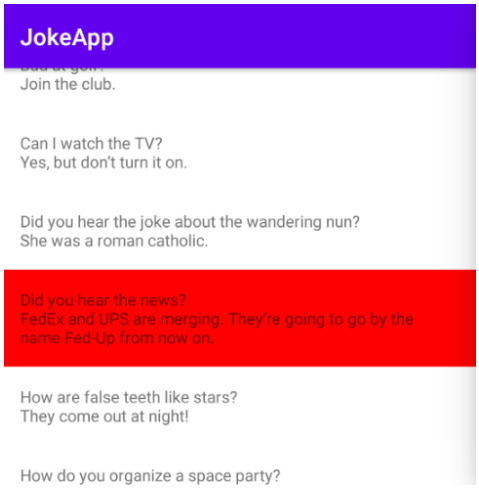
Если количество равно 1 то меняем цвет фона программно. И любой здравомыслящий человек задастся вопросом : а как же иначе? Правильно! Нужно написать else! Но давайте запустим код и посмотрим что произойдет. Ожидаемый результат – только 1 вью будет с красным фоном. А по факту? Запустим и посмотрим!



Как видите первый элемент с красным фоном как и планировалось. Но что будет если проскролить список вниз? Вуаля! Какой-то элемент тоже с красным фоном хотя и не должен. Почему так? А мы уже выяснили : метод onBindViewHolder вызывается каждый раз когда нужно поменять данные. Но секунду, тогда бы все элементы были бы с красным фоном. Нет, дело в том, что метод onCreateViewHolder вызывается не один раз и не 6 раз. Т.е. когда создается вьюхолдер то вьюшке (текстовой) ставится фон. И эту вью с вьюхолдером переиспользуем повторно, но фон не меняли после установки. И потому баг!

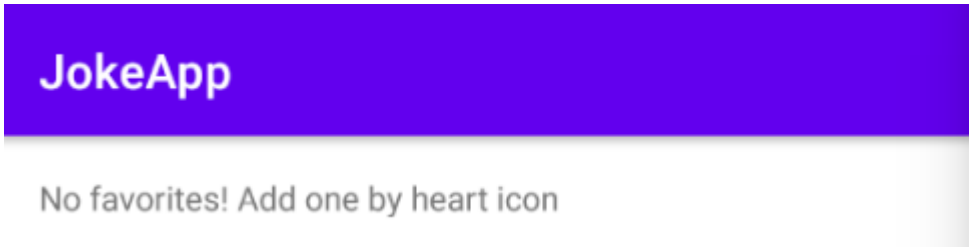
И самое интересное что из 30 элементов красный фон у трех. Ну математика здесь не сложная на самом деле – если у нас 12 раз вызывается метод onCreateViewHolder то получается из 30 элементов 18 остаются на старых вьюхолдерах. Еще раз. При создании вьюхолдера в методе байнд фон был изменен и этот инстанс был переиспользован далее и

потому мы видим более 1 вьюхолдера с красным фоном точнее один и тот же. т.е. в метод байнд приходил уже существующий вьюхолдер с красным фоном. И это все что вам нужно знать про код в методе onBindViewHolder – не пишите if без else. Вы можете поставить айди вьюхолдеру и залогировать и поймете что инстанс вьюхолдера тот же самый в 3 местах.



Но на самом деле есть и другое решение. Вместо того чтобы писать низкоуровневый код в адаптере можно использовать более крутую штуку как разные типы вью. ViewType.

Смотрите. У нас на старте приложения нет совсем никаких избранных и все что мы делаем сейчас это показываем сообщение в том же виде что и шутку и большая часть экрана пустая



Но как это решить? У нас список из элементов одного вида. Ну и ответ сразу же выводится из вопроса : сделать 2 разных вида вью. Один для пустого случая, а второй обычный. Как это делается? У нас уже готовы наследники модели юай слоя и там уже есть Failed для ошибки. Значит мы можем смотреть на конкретный тип класса в адаптере и поставить другой хмл файл для разметки вью. Давайте напишем уже код!

```
override fun getItemViewType(position: Int) = when (list[position]) {
    is FailedCommonUiModel -> 0
    else -> 1
}
```

Есть метод, который определяет тип вью и здесь мы можем завязаться на конкретный класс. Да, проверка инстанса не совсем красиво, но мы позже сможем поменять таким образом чтобы у базового было поле “тип”. И кароче мы говорим что элемент имеет вью тип 0 если это ошибка (когда у нас пустой список, то мы кладем туда Fail, посмотрите BaseInteractor), а для всех других видов пусть будет 1. У нас список избранных и в него не попадет элемент не избранный. Но в этом случае можете отдавать например 2. Теперь нам нужно поменять метод onCreateViewHolder чтобы он создавал нужный для разных вьютипов.

```kotlin
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): CommonDataViewHolder {
    val emptyList = viewType == 0
    val view = LayoutInflater.from(parent.context).inflate(
        if (emptyList)
            R.layout.no_favorite_item
        else
            R.layout.common_data_item,
        parent, attachToRoot: false
    )
    return if (emptyList) EmptyFavoritesViewHolder(view) else CommonDataViewHolder(view)
}
```

Итак, у нас в аргумент метода приходит вьютайп который мы определили выше и мы можем указать нужную разметку и класс вьюхолдера. Просто написали новый хмл файлик и новый класс который наследуется от базового и ничего не меняли в нем. Да, просто фулскрин текст с фоном например и другим стилем текста. Айди тот же чтобы работало.

```xml
<?xml version="1.0" encoding="utf-8"?>
<com.github.johnnysc.jokeapp.presentation.CorrectTextView
    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="match_parent"
    android:background="@color/purple_200"
    android:gravity="center"
    android:textAppearance="@style/TextAppearance.AppCompat.Title"
    tools:text="No favorite items" />
```

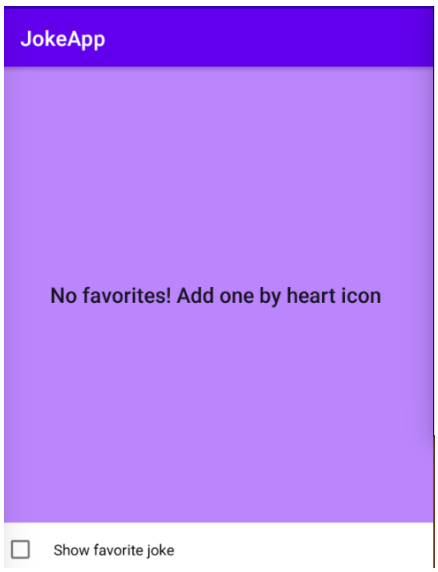
А класс простой наследник, да, это не совсем круто, надо сделать базовый и от него отнаследоваться, но пока у нас нет сильных различий я сделал быстрое решение. Можете сами переписать по красоте.

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

inner class EmptyFavoritesViewHolder(view: View) : CommonDataViewHolder(view)
```

Ладно, давайте уже запустим проект и посмотрим на результат. Удалите все что было чтобы увидеть элемент при отустствии избранных

И как видите у нас полномасштабная текстовка с фоном. Завтра вы захотите например добавить туда изображение как делается в многих компаниях. И вам просто нужно будет дополнить разметку и все. Остальное все уже готово и работает.

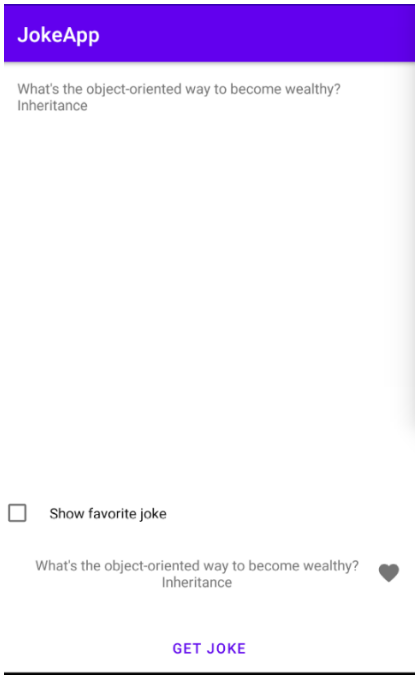


Но у нас остался один нерешенный вопрос. Список не обновляется в реальном времени при добавлении шутки. Давайте же пофиксим это! У нас уже есть метод обновления данных, верно? Давайте для начала заюзаем его. После я расскажу как можно сделать иначе.

```
override fun changeItemStatus() {
    viewModelScope.launch(dispatcher) {  this: CoroutineScope
        if (communication.isState(State.INITIAL)) {
            interactor.changeFavorites().to().show(communication)
            communication.showDataList(interactor.getItemList().toUiList())
        }
```

Если помните у нас в вм был метод. Он дергал метод у интерактора и обновлял 1 элемент в статусе. Но я могу просто вызвать второй метод который обновит список и вуаля!

Запустите проект! Вы добавляете шутку в избранное и список сразу обновляется!



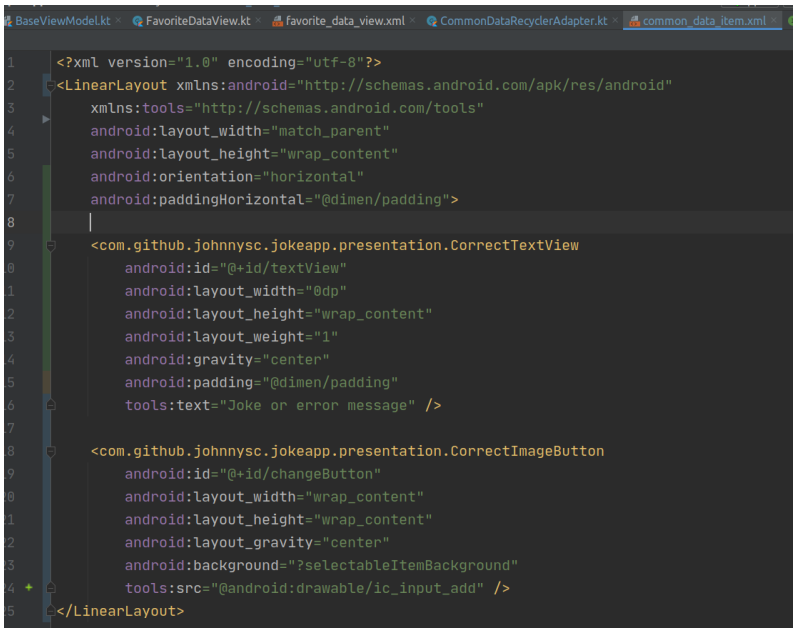
Можете проверить что и вторая добавится и удалится все и вы опять увидете фиолетовый фон и сообщение что нет избранных. Заметьте насколько просто мы добавили функционал. 1 линией. А теперь представьте что у вас не клин (чистая архитектура), а спагети код. Сколько часов вам понадобилось бы чтобы написать код?

Ладно, мы добавили в список новые шутки в реальном времени. Но что не так? А то, что мы после изменения кешдатасорса сразу же вызываем метод получения всех элементов и обновляем весь список. А что не так? Я уже рассказывал и не зря что метод notifyDataSetChanged вызывает все методы адаптера и вы понимаете что условно 6 элементов будут постоянно обновляться. Но это меньшее из зол на самом деле. Наши элементы лишь текстовки и ничего тяжелого. А есть ли другие способы? Да. Мы можем менять не весь список сразу, а уведомить адаптер что добавили или удалили один элемент. Делается это с помощью notifyItemChanged но там надо дать внутрь позицию. И это немного затрудняет положение. И самое простое решение конечно же просто постить весь список.

Но у нас остался незакрытый вопрос. Ок. Я могу добавить в список избранных в реальном времени и удалить из него если элемент отображен на дне экрана. А что если я хочу удалять элементы из избранного прямо в списке?

И здесь мы сталкиваемся с такой проблемой которая якобы решена в некоторых приложениях. Вот вы удалили из списка например свайпом влево: список обновить сразу? А что если вы случайно удалили? В некоторых приложениях появляется снекбар с кнопкой : отменить удаление. Но я за другой подход. Подтверждение действия, а не отмена действия с секундомером в 5 секунд. Т.е. я предлагаю сделать удаление элемента и показать снекбар где подтвердить действие. Давайте попробуем написать код. Нам нужно чтобы элементы свайпались например или удалять их при нажатии на весь? Как сделать? Самое простое по клику на элемент. Но это не очень явно. Давайте переиспользовать существующий подход и сделаем сердечко, на которое сможем нажать.

Кстати, заметьте, что мне сейчас нужно скопировать кусок из хмл для кастомвью. Но я бы не хотел этого делать. Как же поступить? <include ! Мы вынесем в отдельную разметку и будем использовать в списке и эту разметку переиспользуем в кастомвью. Смотрите внимательно.



У нас уже был файлик с названием common_data_item и мы туда просто положим линейный контейнер с текстом и кнопкой сердечком. И в разметке кастомкласса сделаем инклуд

Но увы, это невозможно. Поэтому у нас будет блок в хмл одинаковый в 2 файлах. Но ничего страшного. Позже мы придумаем как справиться с этим (будем писать вью кодом без хмл).

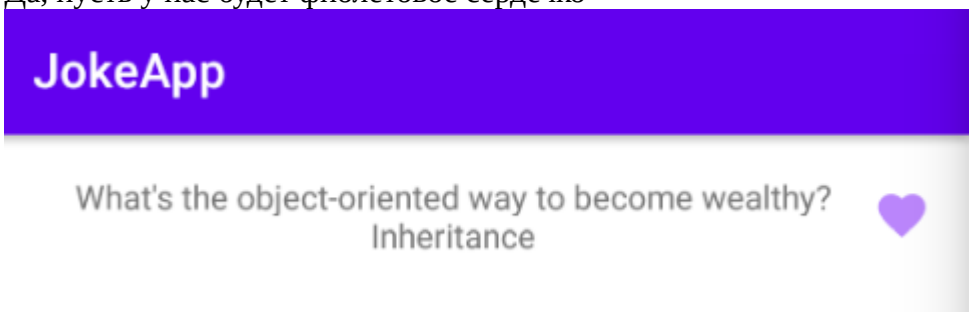
Ладно. Теперь нужно переписать код в адаптере немного. Поменяйте обратно айди

<com.github.johnnysc.jokeapp.presentation.CorrectTextView
android:id="@+id/commonDataTextView"

Чтобы не возникло проблем при наследовании классов.
И значит нам нужно сделать кликабельным иконку и поставить ей значение. Хотя мы можем сделать это прямо в разметке. Давайте не будем писать лишний код и напишем в хмл

android:tint="@color/purple_200"
android:src="@drawable/baseline_favorite_24" />

Да, пусть у нас будет фиолетовое сердечко



Все вроде бы хорошо, но мы не написали код чтобы что-то происходило при нажатии. Давайте значит сделаем это. Нам теперь нужно выделить базовый класс и 2 наследника. А не наследоваться от одного. Потому что когда пусто и у нас просто текст то кнопки нет. А я не хочу делать нулабл переменную. Итак, поехали

```
abstract class CommonDataViewHolder(view: View) : RecyclerView.ViewHolder(view) {
    private val textView = itemView.findViewById<CorrectTextView>(R.id.commonDataTextView)

    open fun bind(model: CommonUiModel) = model.show(textView)

    class Base(view: View) : CommonDataViewHolder(view) {
        private val iconView = itemView.findViewById<CorrectImageButton>(R.id.changeButton)
        override fun bind(model: CommonUiModel) {
            super.bind(model)
            iconView.setOnClickListener { it: View!
```

Я просто поменял класс на абстрактный и сделал базовую реализацию. Пустую не трогаем. Так же делаем метод переопределяемым для наследников но там оставляем отображение текста. Теперь у нас остался один вопрос. Как нам при нажатии на сердечко показать снекбар? По-хорошему нам нужно просто выйти из адаптера наружу в активити, потому что вьюмодель находится там. Следовательно нам нужен интерфейс для обработки нажатия. Давайте напишем его! Но у нас маленькая проблема. Мы не прокидывали айди шутки в юай слой и теперь не сможем менять ее статус. В кастомвью у нас была такая логика : перед

показом в юай мы кешировали шутку и работали с кешем. Как же нам теперь работать с элементом списка? Может тогда надо прокидывать айди из датаслоя до юай? Какие еще варианты у нас есть? Менять кеш и как потом возвращать значение? Сложно. Нам все же нужен айди как уникальный определитель элемента чтобы с ним работать. Давайте тогда уж прокидывать айди. Напишем все же интерфейс для работы с айди, но айди 2 видов и потому:

```
interface FavoriteItemClickListener<T> {
   fun change(id: T)
}
```

Теперь перепишем юай модели. Так как айди нам нужен лишь в избранном классе, то:

```
abstract class CommonUiModel<T>(private val first: String, private val second: String) {
    protected open fun text() = "$first\n$second"
    open fun change(listener: CommonDataRecyclerAdapter.FavoriteItemClickListener<T>) = Unit
    fun show(showText: ShowText) = showText.show(text())
```

Я не буду заставлять всех наследников писать реализацию метода, только там где нужно. И потому открытый метод с дефолтной пустой реализацией. Заметьте что теперь класс дженерик. Значит надо указать всем наследникам тип

class FailedCommonUiModel(private val text: String) : CommonUiModel<Unit>(text, "") {

Для ошибки нет обработчика нажатия и потому пусто будет

class BaseCommonUiModel<E>(text: String, punchline: String) : CommonUiModel<E>(text, punchline)

То же самое и для базовой модели. т.е. для неизбранного. Оно не будет жить в списке и потому нажимать на него и убирать флаг избранного не будем. Но обратите внимание что я все же передаю дженерик хоть он и не нужен. Без него не будет работать мапинг.
И наконец избранная модель:

```
class FavoriteCommonUiModel<E>(private val id: E, text: String, punchline: String) :
    CommonUiModel<E>(text, punchline) {
    override fun change(listener: CommonDataRecyclerAdapter.FavoriteItemClickListener<E>) =
        listener.change(id)
```

Так как айдишник у меня приватный, то я сделал метод с аргументом лиснера, и у него уже вызову метод изменения где передаю айди. Вот и все. Теперь осталось поменять код где создавались инстансы. Теперь я получаю айди и прокидываю его в избранную модель.

```
sealed class CommonItem<E> : Mapper<CommonUiModel<E>> {
    class Success<E>(
        private val id: E,
        private val firstText: String,
        private val secondText: String,
        private val favorite: Boolean
    ) : CommonItem<E>() {
        override fun to() = if (favorite) {
            FavoriteCommonUiModel(id, firstText, secondText)
        } else {
            BaseCommonUiModel(firstText, secondText)
```

И не забудьте для ошибки передать юнит тип

class Failed(private val failure: Failure) : CommonItem<Unit>()

И поменяем наш мапер успеха

```
class CommonSuccessMapper<E> : CommonDataModelMapper<CommonItem.Success<E>, E> {
    override fun map(id: E, first: String, second: String, cached: Boolean) =
        CommonItem.Success(id, first, second, cached)
```

Если помните ранее мы не передавали айди никуда, хоть и получали его в методе мап.

И последнее что нужно сделать это пофиксить адаптер. В конструктор ему нужно передать лиснер и сделать дженериком.

class CommonDataRecyclerAdapter<T>(private val listener: FavoriteItemClickListener<T>) :
RecyclerView.Adapter<CommonDataRecyclerAdapter.CommonDataViewHolder<T>>() {

После чего у нас поменяются все методы

private val list = ArrayList<CommonUiModel<T>>()

fun show(data: List<CommonUiModel<T>>) {

И далее переопределенные методы адаптера

override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
CommonDataViewHolder<T> {

При создании вьюхолдера тоже нужно прокинуть лиснер
return if (emptyList) EmptyFavoritesViewHolder(view)
else CommonDataViewHolder.Base(view, listener)

И уже в классе вьюхолдера в методе setOnClickListener вызовем метод у модели

```
abstract class CommonDataViewHolder<T>(view: View,) : RecyclerView.ViewHolder(view) {
    private val textView = itemView.findViewById<CorrectTextView>(R.id.commonDataTextView)
    open fun bind(model: CommonUiModel<T>) = model.show(textView)
    class Base<T>(view: View, private val listener: FavoriteItemClickListener<T>) :
        CommonDataViewHolder<T>(view) {
        private val iconView = itemView.findViewById<CorrectImageButton>(R.id.changeButton)
        override fun bind(model: CommonUiModel<T>) {
            super.bind(model)
            iconView.setOnClickListener { it: View!
                model.change(listener)
```

Еще раз : мы не знаем какая будет модель, успеха или ошибки. Мы вызываем у нее метод и отдаем лиснер. Если это успешная модель, то она отдаст айди лиснеру. Вот и все

inner class EmptyFavoritesViewHolder<T>(view: View) : CommonDataViewHolder<T>(view)

Пустому тоже пробросим дженерик чтобы не было конфликта в коде адаптера.
И следующий шаг теперь поменять код в активити. Ведь адаптеру нужен лиснер в конструктор и мы создадим аноним класс где вызовем метод у вьюмодели.

Если вы забыли то айди у шуток был числовой, значит передаем Int адаптеру

```
val adapter = CommonDataRecyclerAdapter(object : CommonDataRecyclerAdapter.FavoriteItemClickListener<Int> {
    override fun change(id: Int) {
        viewModel.changeItemStatus(id)
    }
})
```

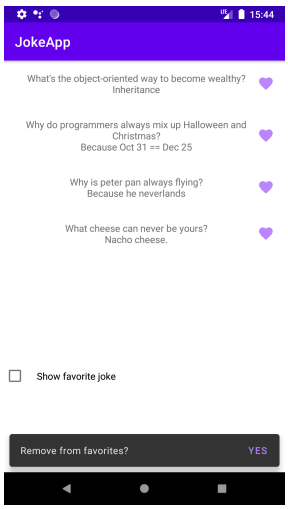
И если помните то метод changeItemStatus относился именно к айтему который закеширован в слое данных и это тот, который отображается в кастомвью. Теперь нам нужен новый метод, который примет аргументом айди и это будет дженерик. Давайте напишем его
Хотя стоп. Мы же хотели все же подтверждать действите. Поэтому надо написать вызов снекбара с действием. Сначала добавим текст вопроса и для кнопки подтверждения

<string name="remove_from_favorites">Remove from favorites?</string>
<string name="yes">yes</string>

Теперь уже можно написать код в активити

```
val adapter = CommonDataRecyclerAdapter(object :
    CommonDataRecyclerAdapter.FavoriteItemClickListener<Int> {
    override fun change(id: Int) {
        Snackbar.make(
            favoriteDataView,
            R.string.remove_from_favorites,
            Snackbar.LENGTH_SHORT
        ).setAction(R.string.yes) { it: View!
            viewModel.changeItemStatus(id)
        }.show()
```

Создаем снекбар, передаем вопрос и после добавим действите подтверждения. При нажатии будет вызываться метод у вьюмодели. Можете прямо сейчас уже проверить код. Закомментируйте код вызова метода который еще не написали и запустите проект



Как видите снекбар появляется на дне экрана и доступна кнопка для подтверждения удаления. Теперь мы можем написать сам код удаления и обновления списка.

Давайте поступим так : мы будем обновлять список который лежит в ливдате в комуникации сразу, а в бд следующей линией.

```
override fun changeItemStatus(id: T) {
    communication.removeItem(id)
    viewModelScope.launch(dispatcher) {  this: CoroutineScope
        interactor.removeItem(id)
```

Давайте сначала напишем код у комуникатора

```
interface Communication {
    fun showState(state: State)
    fun observe(owner: LifecycleOwner, observer: Observer<State>)
    fun isState(type: Int): Boolean
}

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

interface CommonCommunication<T> : Communication, ListCommunication<T>
```

Я разделил интерфейсы по принципу Inteface segregation для списка и для айтема в кастомвью. Просто потому что в одних методах нужен дженерик тип, а для других методов нет. В телеграме я укажу ссылку на сам комит где я пофиксил все слои и проставил дженерик всем классам и интерфейсам.

Для того чтобы из списка который в ливдате удалить элемент по айди который приватный, нужно написать метод в классе модели юай слоя.

```
abstract class CommonUiModel<T>(private val first: String, private val second: String) {
    protected open fun text() = "$first\n$second"
    open fun change(listener: CommonDataRecyclerAdapter.FavoriteItemClickListener<T>) = Unit
    open fun matches(id: T) : Boolean = false
```

И в наследнике избранной модели пишем код

```
class FavoriteCommonUiModel<E>(private val id: E, text: String, punchline: String) :
    CommonUiModel<E>(text, punchline) {
    override fun change(listener: CommonDataRecyclerAdapter.FavoriteItemClickListener<E>) =
        listener.change(id)

    override fun matches(id: E): Boolean  = this.id == id
```

И теперь напишем код для обновления ливдаты

```
private val listLiveData = MutableLiveData<List<CommonUiModel<T>>>()
override fun removeItem(id: T) {
    val source = listLiveData.value?: emptyList()
    val list = source.filter { it: CommonUiModel<T>
        !it.matches(id)
    }
    showDataList(list)
}
```

Но стоп. Мы опять же написали код который обновит весь список. Мы же не хотели этого делать. Нам нужна новая ливдата которая бы убирала 1 элемент из списка, а не передавала весь список. Значит нам нужно удалить элемент из списка ливдаты, но тем временем не отправлять событие. Как же это сделать? Помните мы устанавливали текст в поле ввода программно? Мы удаляли текстводчер и устанавливали его позже. Точно это же нужно сделать и сейчас. Только у нас обсервер наблюдатель и его нужно убрать и поставить заново.

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

Сначала добавим аргуемнты методу комуникации списка.

```
override fun removeItem(
    id: T,
    owner: LifecycleOwner,
    observer: Observer<List<CommonUiModel<T>>>
) {
    val source = listLiveData.value ?: emptyList()
    val list = source.filter { it: CommonUiModel<T>
        !it.matches(id)
    }
    listLiveData.removeObserver(observer)
    showDataList(list)
    observeList(owner, observer)
}
```

Теперь после фильтрации списка можем убрать наблюдателя и засетить список и опять начать наблюдать за обновлениями.

```
interface CommonViewModel<T> : CommonItemViewModel {
    fun changeItemStatus(id: T, owner: LifecycleOwner,  observer: Observer<List<CommonUiModel<T>>>)
```

И нам нужно передать эти же аргументы из активити в вюмодель.

```
override fun changeItemStatus(
    id: T,
    owner: LifecycleOwner,
    observer: Observer<List<CommonUiModel<T>>>
) {
    communication.removeItem(id, owner, observer)
    viewModelScope.launch(dispatcher) {  this: CoroutineSco
        interactor.removeItem(id)
```

Код интерактора напишем чуть позже. Я хочу сейчас проверить что удаляется 1 элемент из списка а не обновляется весь список в адаптере. И давайте перепишем код в активити

```
val recyclerView = findViewById<RecyclerView>(R.id.recyclerView)
val observer: (t: List<CommonUiModel<Int>>) -> Unit = { list ->
    adapter.show(list)
}
adapter = CommonDataRecyclerAdapter(object :
    CommonDataRecyclerAdapter.FavoriteItemClickListener<Int> {
    override fun change(id: Int) {
        Snackbar.make(
            favoriteDataView,
            "Remove from favorites?",
            Snackbar.LENGTH_SHORT
        ).setAction("yes") {  it: View!
            viewModel.changeItemStatus(id,  owner: this@MainActivity, observer)
        }.show()
    }
})
recyclerView.adapter = adapter

viewModel.observeList( owner: this, observer)
```

Я вынес адаптер над методом onCreate активити чтобы можно было использовать в 2 местах. Хотя на самом деле мы бы могли не создавать ливдату, а сразу в акивити дернуть метод адаптера. Да, давайте упростим нам жизнь. Я добавлю метод удаления элемента прямо после того как вызывается код у вьюмодели.

Я не хочу писать новую ливдату для удаления одного элемента потому что удаление происходит в юай из-за действия юзера. А список обновляется в комуникации отдельно и бд обновится отдельно тоже.

```
viewModel.changeItemStatus(id, owner: this@MainActivity, observer)
adapter.removeItem(id)
```

Теперь посмотрим на код удаления элемента

```
fun removeItem(id: T) {
    val element = list.find { it.matches(id) }
    val position = list.indexOf(element)
    list.remove(element)
    notifyItemChanged(position)
}
```

Сначала я нахожу элемент по айди, для этого мы и писали метод matches. После я нахожу текующую позицию элемента в списке через indexOf и после удаляю элемент из списка и говорю адаптеру что я удалил элемент из списка. Чтобы проверить это все дело, давайте поставим брейкпойнты на методе адаптера show чтобы мы понимали что реально мы удалили обьект и обновили 1 вьюхолдер, а не все 6.

И у нас проблема. Потому что в момент когда мы начинаем опять наблюдать за изменениями значение подтягивается. И вам не кажется что мы дважды написали один и тот же код?

Вообще я вам обещал переписать код : ведь у нас список хранится в 2 местах: и в ливдате которая в комуникации и в адаптере самом. Давайте удалим из адаптера список и будем получать его из ливдаты. Напишем простой метод получения списка и будем использовать его вместо переменной.

```
interface ListCommunication<T> {
    fun getList() : List<CommonUiModel<T>>
```

Напишем реализацию в классе таким образом

```
override fun getList(): List<CommonUiModel<T>> {
    return listLiveData.value ?: emptyList()
}
```

И теперь перепишем адаптер

Передаем в конструктор коммуникации и там где вызывался список теперь будет вызов метода у интерфейса.

```
class CommonDataRecyclerAdapter<T>(
    private val listener: FavoriteItemClickListener<T>,
    private val communication: CommonCommunication<T>
) : RecyclerView.Adapter<CommonDataRecyclerAdapter.CommonDataViewHolder<T>>() {
```

Посмотрите теперь как мы удалим 2 метода remove и show внутри адаптера

```
override fun getItemViewType(position: Int) = when (communication.getList()[position]) {
    is FailedCommonUiModel -> 0
```

Ведь зачем нам хранить список из сотен предположим элементов в самом адаптере если он есть в ливдате.

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

И последний метод перепишем (метод создания вьюхолдеров остался прежним).

```
override fun getItemCount() = communication.getList().size
```

Но нам ведь нужно обновлять список на старте и удалять элементы! Как же нам теперь быть?

Мы заменим обсерв методы у ливдаты и будем передавать не список, а например пару из булеан и позиции. По позиции у нас будет удаление (или добавление элемента, ведь зачем опять же обновлять весь список, когда всего 1 элемент добавился). Итак, давайте перепишем код. Для начала напишем 2 метода в адаптере. Один просто добавит все значения на старте.

```
fun update() {
    notifyDataSetChanged()
}

fun update(pair: Pair<Boolean, Int>) {
    if (pair.first) {
        notifyItemInserted(pair.second)
    } else {
        notifyItemRemoved(pair.second)
    }
}
```

А второй метод или добавит новый элемент или удалит, точнее отреагирует на изменения.

Теперь исправим коммуникации

```
override fun removeItem(
    id: T,
    owner: LifecycleOwner,
    observer: Observer<List<CommonUiModel<T>>>
) {
    listLiveData.value?.find { it: CommonUiModel<T>
        it.matches(id)
    }?.let { it: CommonUiModel<T>
        listLiveData.value?.remove(it)
    }
}
```

Я нахожу в списке которое значение ливдаты айтем по айди и после удаляю из списка. В активити будет вызван метод обновления

```
).setAction("yes") { it: View!
    viewModel.changeItemStatus(id, owner: this@MainActivity, observer)
    adapter.update(Pair(false, id))
```

Теперь нужно прокинуь комуникации в адаптер. Для этого в апликейшн классе создадим поле и перед инициализацией вьюмодели проинициализируем ее

val jokeCommunication = (application as JokeApp).jokeCommunication

прямо после того как инициализируем вьюмодель в активити

Но стоп. Мы же передаем в метод update пару где вторым аргументом должна идти не айди а позиция элемента. А позиция элемента получается из вьюмодели. Значит нужно ее оттуда получать. Давайте пофиксим это

```
val position = viewModel.changeItemStatus(id, owner: this@MainActivity, observer)
adapter.update(Pair(false, position))
```

Перепишем метод у вьюмодели и исправим везде где нужно: в комуникациях и так далее
Не беспокойтесь если у вас не получается все пофиксить, в конце я запушу код в гитхаб и дам линк на комит в телеграм канале.

```
override fun removeItem(
    id: T,
    owner: LifecycleOwner,
    observer: Observer<List<CommonUiModel<T>>>
): Int {
    val found = listLiveData.value?.find { it: CommonUiModel<T>
        it.matches(id)
    }
    val position = listLiveData.value?.indexOf(found) ?: -1
    found?.let { it: CommonUiModel<T>
        listLiveData.value?.remove(it)
    }
    return position
}
```

И теперь должно корректно работать. Давайте запустим и проверим!
Заметьте какая красивая анимация при удалении. Но мы не удалили из бд! Давайте допишем код в интерактоpе и в репозитории и в кешдатасорсе.

```
override suspend fun remove(id: E) = withContext(Dispatchers.IO) { th
    realmProvider.provide().use { realm ->
        realm.executeTransaction { it: Realm
            findRealmObject(realm, id)?.deleteFromRealm()
```

Да, у нас одинаковый код и в методе удаления и в методе добавления удаления. Если хотите можете вынести общую часть отдельно. Я займусь этим позже. Давайте теперь проверим что все работает верно. И да, добавляется и удаляется все правильно. Хотя мы могли бы написать код добавления точно так же и с анимацией оно бы добавилось.
Предлагаю заняться этим вам самостоятельно.
Главное что вы поняли за эту лекцию как работать с ресайклервью. В чем суть методов которые нужно определить и в чем отличие notifyDataSetChanged и notifyItemRemoved.
В последующих лекциях мы возможно вернемся к этому проекту.

п.с. кстати, теперь мы создали багу : когда удаляешь по одной все айтемы то в конце не появляется айтем ошибки что нет избранных. Потому что нужно проверять что мы удалили последний элемент в списке и отправлять в адаптер элемент с ошибкой.
Вы можете написать это сами. Просто дерните метод получения списка элементов если после удаления список у ливдаты стал пустой.