

CustomView

Масштабируемость проекта

В предыдущей лекции мы поменяли сервер и ответ от него и сделали все за 5 минут. Теперь же мы хотим поменять юай слой. Но не просто поменять, а именно масштабировать. Что это значит? Сейчас у нас есть активити где можно получишь шутку, сохранить и так далее. Я бы хотел чтобы этот функционал был масштабируемым. Т.е. я бы мог работать по той же логике но с другими данными. Например пусть у меня будут еще в приложении цитаты. Я так же хочу их получать, хранить и т.д. Но посмотрите на класс активити. В нем сейчас так много кода! Если мы хотим добавить еще фичу с цитатами то наш класс активити удвоится в размере. Что бы мы могли придумать? Инкапсулировать все в класс кастомвью и переиспользовать целый блок.

```
17         val button = findViewById<CorrectButton>(R.id.actionButton)
18         val progressBar = findViewById<CorrectProgress>(R.id.progressBar)
19         val textView = findViewById<CorrectTextView>(R.id.textView)
20         val checkBox = findViewById<CheckBox>(R.id.checkBox)
21         val changeButton = findViewById<CorrectImageButton>(R.id.changeButton)
22         progressBar.visibility = View.INVISIBLE
23         checkBox.setOnCheckedChangeListener { _, isChecked ->
24             viewModel.chooseFavorites(isChecked)
25         }
26         changeButton.setOnClickListener { it: View!
27             viewModel.changeJokeStatus()
28         }
29         button.setOnClickListener { it: View!
30             viewModel.getJoke()
31         }
```

Мы уберем все эти 30 линий кода в класс и наш активити станет намного меньше даже с 2 блоками одинаковых фиц. Давайте напомним кастомвью класс для блока. Назовем его FavoriteDataView, если придумаем что лучше, то переименуем. Благо в АС это очень просто.

```
class FavoriteDataView : View {

    constructor(context: Context) : super(context)
    constructor(context: Context, attrs: AttributeSet) : super(context, attrs)
    constructor(context: Context, attrs: AttributeSet, defStyleAttr: Int) : super(
        context,
        attrs,
        defStyleAttr
    )

}
```

Конечно же кладем в новый файл и в пакет presentation. Хотя по сути его можно переместить позже в ядро core. Итак, копируем 3 конструктора чтобы АС не ругался и нам нужно выделить разметку в отдельный файл чтобы здесь можно было использовать.

Просто скопируйте все что было в разметке активности и вынесите в новый файл `favorite_data_view.xml`

Теперь же нам нужно получить все вью внутри кастомвью. Как это сделать?

В активности у нас первой линией было `setContentView` и вью и хмл преобразовывались в классы котлин и мы их находили через `findViewById`. Теперь же в кастомвью нам нужно тоже как-нибудь указать из какого хмл файла нужно распарсить данные. (я поменял наследование)

```
class FavoriteDataView : LinearLayout {  
  
    private val checkBox: CheckBox  
    private val textView: CorrectTextView  
    private val changeButton: CorrectImageButton  
    private val actionButton: CorrectButton  
    private val progress: CorrectProgress  
  
    |constructors  
    init {  
        orientation = VERTICAL  
        (context  
            .getSystemService(Context.LAYOUT_INFLATER_SERVICE) as LayoutInflater)  
            .inflate(R.layout.favorite_data_view, root: this, attachToRoot: true)  
        checkBox = getChildAt(index: 0) as CheckBox  
        val linear = getChildAt(index: 1) as LinearLayout  
        textView = linear.findViewById(R.id.textView)  
        changeButton = linear.findViewById(R.id.changeButton)  
        progress = getChildAt(index: 2) as CorrectProgress  
        actionButton = getChildAt(index: 3) as CorrectButton  
    }  
}
```

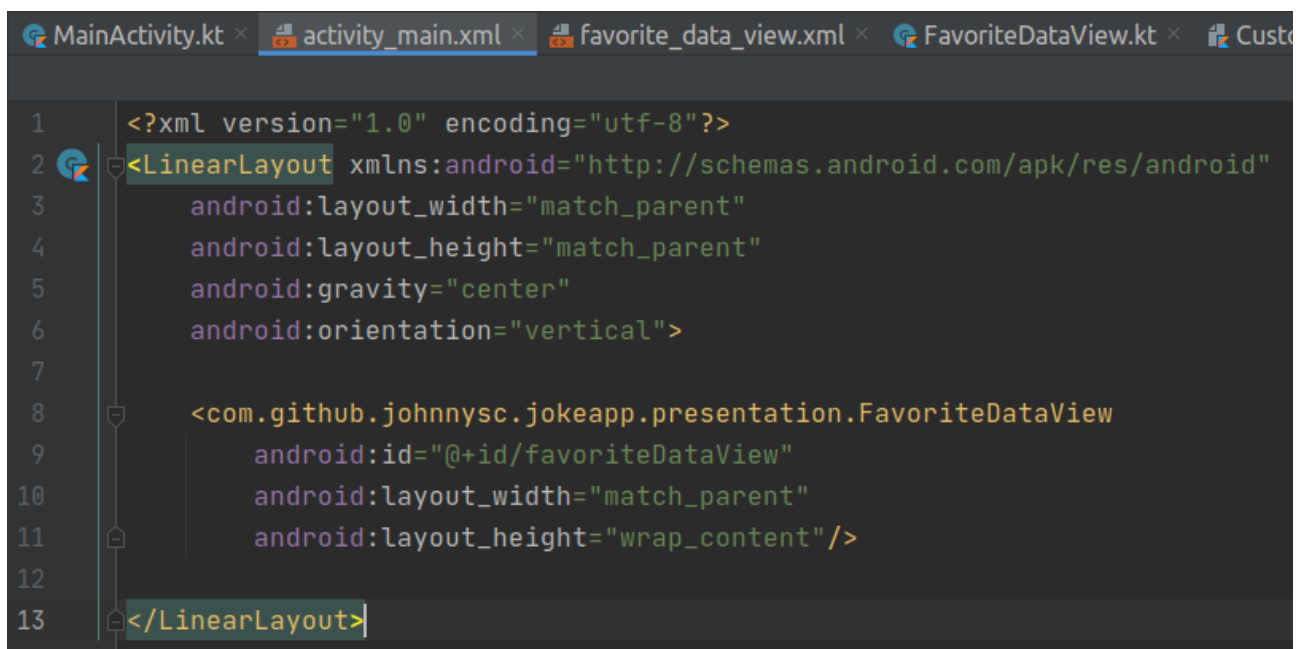
Так как у нас сама кастомвью наследуется от линейного контейнера мы можем даже из разметки убрать родительский контейнер и написать там `<merge>` и тогда у нас не будет лишнего контейнера в активности. Merge работает таким образом что вставляется в тот контейнер в котором пишем в хмл. Благодаря ориентации компилятор поймет как располагать элементы. Так же мы можем обращаться к вью через индекс, ведь у нас линейный контейнер и мы можем не писать `findViewById` хотя у вложенного линейного все же написали так. Выбирайте способ по вкусу. Суть одна и та же. В дальнейшем если поменять порядок у вас все сломается конечно же, так что лучше через айди. Хотя порядок быстрее должен работать.

```
<?xml version="1.0" encoding="utf-8"?>  
|<merge  
    xmlns:android="http://schemas.android.com/apk/res/android"  
    xmlns:tools="http://schemas.android.com/tools">  
  
    <CheckBox
```

Точно так же как в активности находим вью и далее можем работать с ними. Но для работы с ними нужно написать методы. Скопируем из активности

```
fun listenChanges(block: (checked: Boolean) -> Unit) =
    checkBox.setOnCheckedChangeListener { _, isChecked ->
        block.invoke(isChecked)
    }
fun handleChangeButton(block: () -> Unit) = changeButton.setOnClickListener { it: View!
    block.invoke()
}
fun handleActionButton(block: () -> Unit) = actionButton.setOnClickListener { it: View!
    block.invoke()
}
```

Мы передадим лямбды внутрь вместо андроид интерфейсов и все будет работать. Давайте уже сейчас заменим кастомвью в активности и проверим что все работает. В коде же найдем кастом вью и вызовем методы у нее, а не у отдельных вьюх.



```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center"
    android:orientation="vertical">

    <com.github.johnnysc.jokeapp.presentation.FavoriteDataView
        android:id="@+id/favoriteDataView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"/>

</LinearLayout>
```

И посмотрим как теперь выглядит активности класс

```
val favoriteDataView = findViewById<FavoriteDataView>(R.id.favoriteDataView)
favoriteDataView.listenChanges { isChecked ->
    viewModel.chooseFavorites(isChecked)
}
favoriteDataView.handleChangeButton {
    viewModel.changeJokeStatus()
}
favoriteDataView.handleActionButton {
    viewModel.getJoke()
}

viewModel.observe( owner: this, { state ->
    state.show(progressBar, button, textView, changeButton)
})
```

Теперь выглядит намного короче, но у нас все вью использовались в комуникаторе! Значит нам нужно переписать метод show. Суть в том, что теперь класс State относится непосредственно к классу FavoriteDataView и мы можем написать уже метод show у самого класса FavoriteDataView например вот так

```
fun show(state: State) = state.show(progress, actionButton, textView, changeButton)
```

И в активити будет вот так

Как видите намного проще стало и кода в разы меньше. Не нужно инициализировать 5 вью, можно сразу инициализировать 1 кастомвью в которой все они есть.

```
viewModel.observe( owner: this, { state ->
    favoriteDataView.show(state)
})
```

Теперь пойдем дальше и запустим код для проверки

☐ Show favorite joke

Whats the Grinchs least favorite band?
The Who.

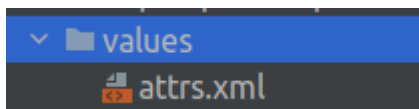


GET JOKE

Все работает как и по-прежнему. Еще один плюс чистой архитектуры в том, что мы поменяли юай слой, не трогая остальные и все равно все работает как и раньше.

Но у нас маленькая проблема – тексты в разметке указаны напрямую. Это можно решить 2 способами: или из кода для каждого кастомвью сетить 2 значения или же из самой хмл. Но как? Ранее мы просто писали у чекбокса и у кнопки тексты и все работало. Теперь же у нас в активити доступен лишь кастомвью класс. Было бы классно прямо в разметке активити иметь способ написать текста чекбоксу и кнопке.

И у нас есть такая возможность. Мы можем создать свои атрибуты для нашего класса.



Нужно создать хмл файл в пакете values в дереве вида Project и туда нужно написать наши кастомные атрибуты.

У нас 2 текста и потому тип пишем стринг у обоих атрибутов. Даем простые имена и теперь можем в кастомвью получить их. Но каким образом? Давайте для начала удалим из кастомвью те тексты которые есть и напишем их в активити разметке

```
ewModel.kt × activity_main.xml × favorite_data_view.xml × attrs.xml ×
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <declare-styleable name="FavoriteDataView">
        <attr name="chekBoxText" format="string" />
        <attr name="actionButtonText" format="string" />
    </declare-styleable>
</resources>
```

Нам нужен новый неймспейс – пространство имен для кастом вью

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    xmlns:custom="http://schemas.android.com/apk/res-auto"
    android:gravity="center"
    android:orientation="vertical">

    <com.github.johnnysc.jokeapp.presentation.FavoriteDataView
        android:id="@+id/favoriteDataView"
        custom:actionButtonText="@string/get_joke"
        custom:chekBoxText="@string/show_favorite_joke"
        android:layout_width="match_parent"
```

Я назвал custom, но вы можете дать любое имя. И теперь в кастомвью в разметке доступны новые атрибуты. Вуаля! Но это еще не все. Мы в разметке дали новые атрибуты, но их никто не прочитает в коде кастомвью. Давайте исправим это.

Проблема в том, что атрибуты доступны из 2 конструкторов и поэтому нужно переписать немного поля класса с val на lateinit var и передать атрибуты в метод init.

```
context.theme.obtainStyledAttributes(attrs, R.styleable.FavoriteDataView, defStyleAttr, defStyleRes).apply {
    try {
        val actionButtonText = getString(R.styleable.FavoriteDataView_actionButtonText)
        val checkBoxText = getString(R.styleable.FavoriteDataView_chekBoxText)
        checkBox.text = checkBoxText
        actionButton.text = actionButtonText
    } finally {
        recycle()
    }
}
```

В конце метода init добавим такой код – получаем атрибуты по ключу – не увидяйтесь – R.styleable.FavoriteDataView сгенерированная константа. Когда вы нажимаете на билд из хмл все имена превращаются в константы. Если у вас АС не предложило имена – билдите проект просто и все будет. Не забываем в конце подчистить за собой используемые данные.

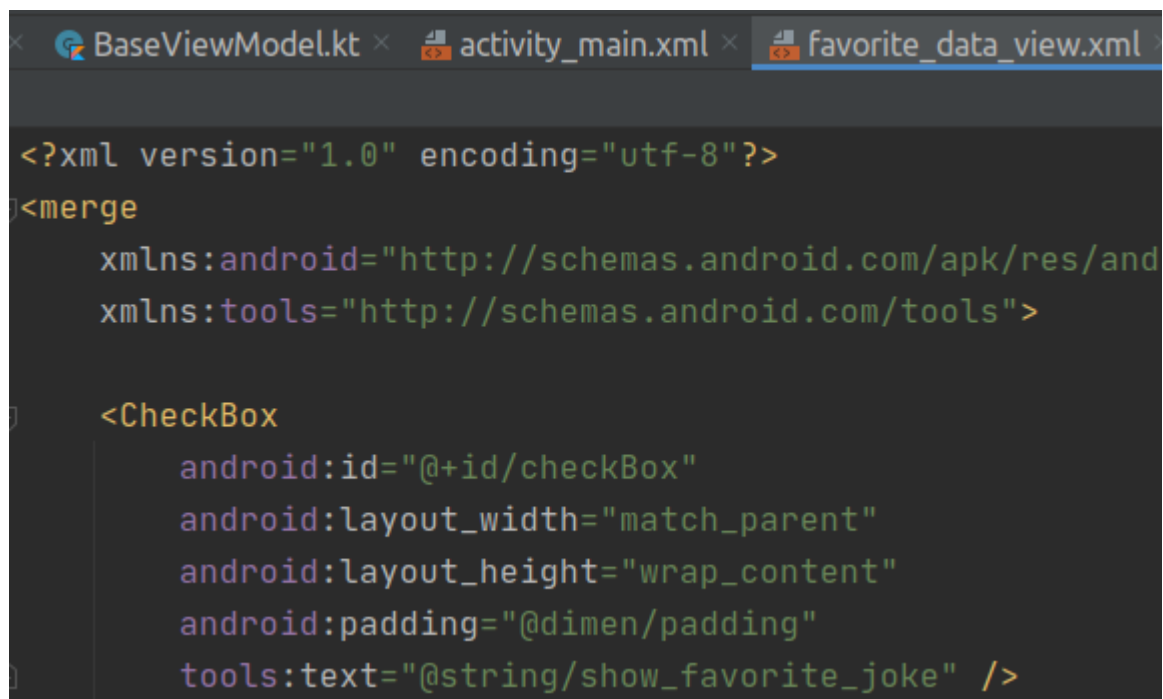
```
private lateinit var progress: CorrectProgress

//region constructors
constructor(context: Context) : super(context)
constructor(context: Context, attrs: AttributeSet) : super(context, attrs) {
    init(attrs)
}

constructor(context: Context, attrs: AttributeSet, defStyleAttr: Int) : super(
    context,
    attrs,
    defStyleAttr
) {
    init(attrs)
}

//endregion
private fun init(attrs:AttributeSet) {
```

И наконец можно запустить код и проверить что все работает (надеюсь вы не забыли удалить текстовки из разметки кастомвью или заменить на tools:text).



```
<?xml version="1.0" encoding="utf-8"?>
<merge
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <CheckBox
        android:id="@+id/checkBox"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:padding="@dimen/padding"
        tools:text="@string/show_favorite_joke" />
```

Ну и конечно же все работает! Кто бы сомневался

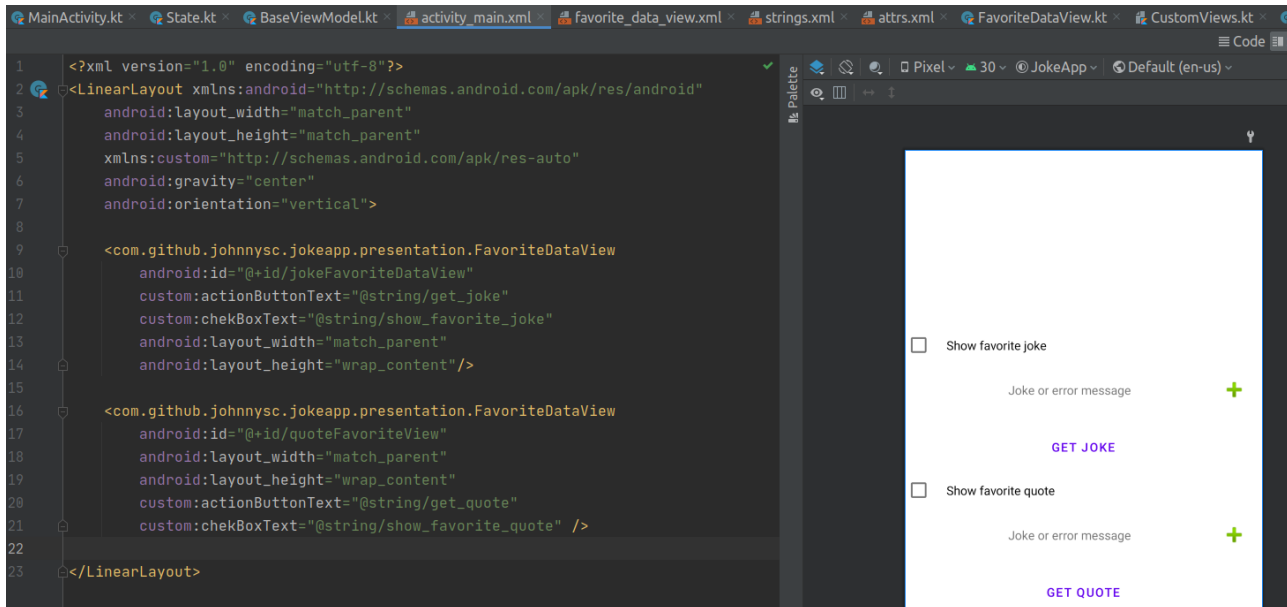
Но давайте вспомним зачем мы это делали – для того чтобы мы могли добавить в юай новый блок легко и просто. Давайте уже добавим второй блок для цитат. Да, у нас будет приложение в котором можно смотреть не только шутки, но и цитаты. Для этого создадим в строковых ресурсах еще 2 значения для юай слоя

```

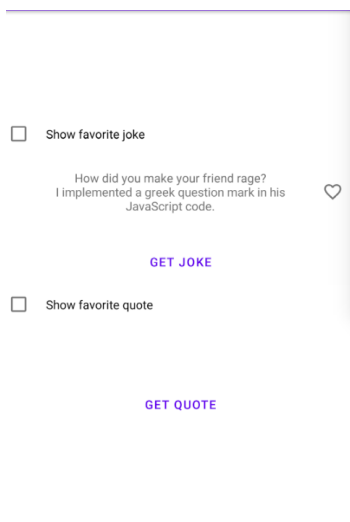
<string name="get_joke">Get joke</string>
<string name="get_quote">Get quote</string>
<string name="show_favorite_joke">Show favorite joke</string>
<string name="show_favorite_quote">Show favorite quote</string>

```

И теперь легко и просто добавим в активити новый блок



Вуаля! Вместо 12 вьюх всего 2. Красота же ну! Давайте запустим код и увидим что все выглядит прилично (кстати, я переименовал айди кастомвью в разметке активити).



Как видите все прекрасно уместилось. Шутки по прежнему работают. Но для цитат придется написать немного кода. И мы это сделаем в следующей лекции.

Точно так же как с юай слоем – мы вынесли общий класс кастомвью и переиспользовали его для 2 разных фиц мы поступим и со всем остальным кодом – у нас будет много общих классов с дженерик типами в пакете ядра. А для конкретных фиц уже конкретные реализации в пару линий и все будет работать точно так же как и раньше. Вот где сила чистой архитектуры. В следующей лекции будет много кодинга. А пока – оставайтесь на связи!