

Kotlin 1.7
Jetpack Compose 1.2

 Лайв Тайпинг

Разработка Android приложений с Jetpack Compose



12+

Денис Сергеевич Попков

Разработка Android

приложений с Jetpack Compose

http://www.litres.ru/pages/biblio_book/?art=67971732

SelfPub; 2022

Аннотация

Для кого это руководство?

Для тех кто хочет использовать фреймворк в новых проектах или познакомиться с современным способом верстки UI. В руководстве рассмотрены часто используемые в разработке инструменты, компоненты.

Главное отличие этого руководства от других книг – это обзор всех возможностей фреймворка и его компонентов.

Денис Попков

Разработка Android приложений с Jetpack Compose

Вступление

Верстка экранов – неотъемлемая часть создания мобильных приложений. Долгое время разработчики использовали для этого язык текстовой разметки XML. Данный подход себя хорошо зарекомендовал, но зачастую для решения тривиальных задач необходимо писать много boilerplate кода.

Задача отображения списка новостей требует в XML проектах: адаптер, делегат, верстку элемента списка + фрагмента, **Presenter/ViewModel**. Слишком много кода для повседневной задачи.

Jetpack Compose значительно упрощает процесс верстки, он позволит справиться с задачей выше, менее чем в 200 строк. Помимо лаконичности, фреймворк предоставляет возможности писать мультиплатформенный легко поддерживаемый **reusable** код.

Нововведения из последних версий **Android** быстрее приходят в фреймворк, нежели в XML. Google заинтересована в развитие нового видения и предлагает использо-

вать современный стэк технологий, который входит в состав **Jetpack**: MVVM, Coroutines, Jetpack Compose, Room, Hilt, Ktor, DataStore, WorkManager, Coil.

В стэке технологий также представлены **3rd party** библиотеки и архитектурное решение **MVVM – Model-View-Model**.

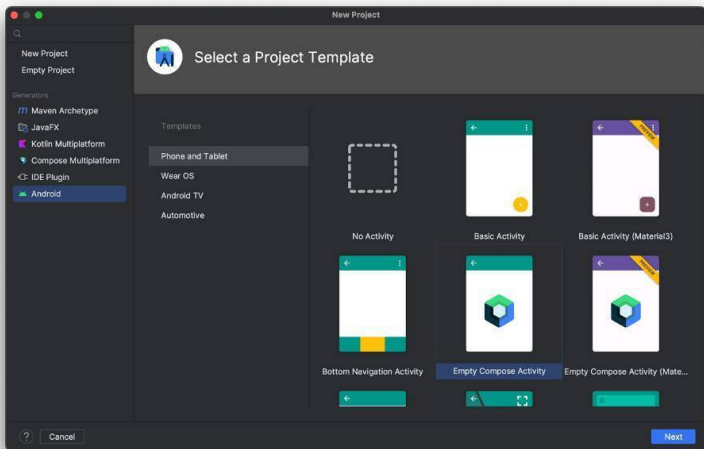
Для кого это руководство?

Для тех, кто хочет использовать фреймворк в новых проектах или познакомиться с современным способом вёрстки **UI**. В руководстве рассмотрены часто используемые в разработке инструменты и компоненты.

Главное отличие этого руководства от других книг – это обзор всех возможностей фреймворка и его компонентов.

Настройка проекта

Процесс создания проекта схож со стандартным **File – New – New project**. Выберите **Empty Compose Activity**, далее укажите название проекта, пакет, версию Android и нажмите **Finish**. Обновите версии библиотек до последних версий.



Код зависимостей вы можете найти по ссылке –
[<https://inky-belief-259.notion.site/23f6848a200346d3b1b49ad211226b27>].

Основные аспекты

Все построение UI в **Jetpack Compose** завязано на функциях, помеченных аннотацией **Composable**, она указывает компилятору на то, что эта функция должна быть сконвертирована в UI.

Composable функции могут быть вызваны только из других **Composable** функций или лямбд.

В классе **MainActivity** есть точка входа, **Composable setContent** лямбда-блок, внутри которой как раз и будет вы-

зываться весь интерфейс приложения. Функции вызываются друг под другом и отображаются в порядке вызова их в коде.

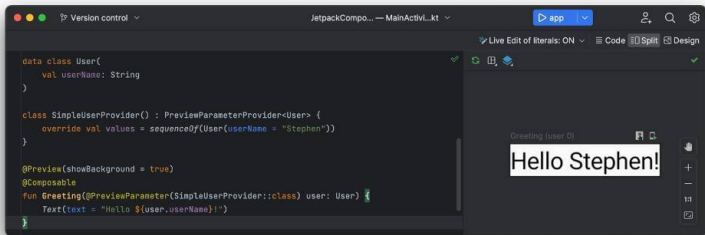
Compose предоставляет возможность переиспользовать элементы, так же как и в **custom XML View**. Для этого нужно создать **package** с базовыми компонентами, задать некоторые параметры **Composable** функции, а поведение определять через параметры.

Preview

Аннотация **Preview** отображает в боковом меню сверстаные элементы. Если необходимо отобразить дополнительно фон **View**, укажите `showBackground = true`. После внесения изменений нужно нажать `build&refresh` для обновления **Preview**, если изменения небольшие, то **Preview** обновится автоматически.

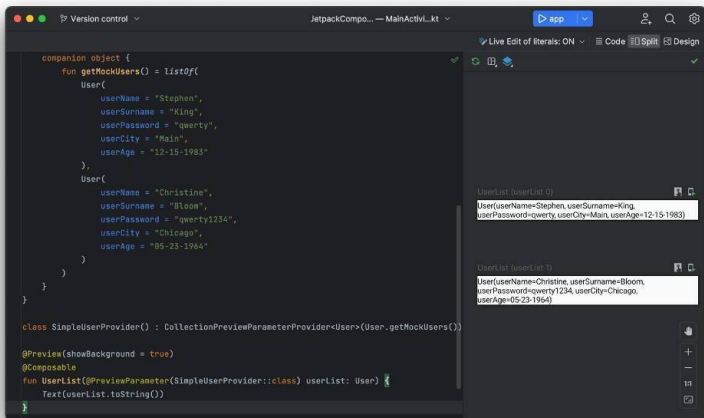
Учтите, что **Preview** не будет работать, если **Composable** функция принимает какие-либо параметры и они при этом не заданы по умолчанию.

Preview Parameter позволит это исправить, также упростит использование **mock** файлов в проекте для тестирования верстки.



Так для создания **provider**-класса необходимо наследоваться от **PreviewParameterProvider<T>** и переопределить значение по умолчанию. **Preview Parameter** принимает **provider** класс и параметр – максимальное количество элементов, которые он должен отобразить.

В случае, если **provider** класс возвращает объект, обернутый в список, то **Preview** отобразится единожды, что порой удобно для анализа верстки.

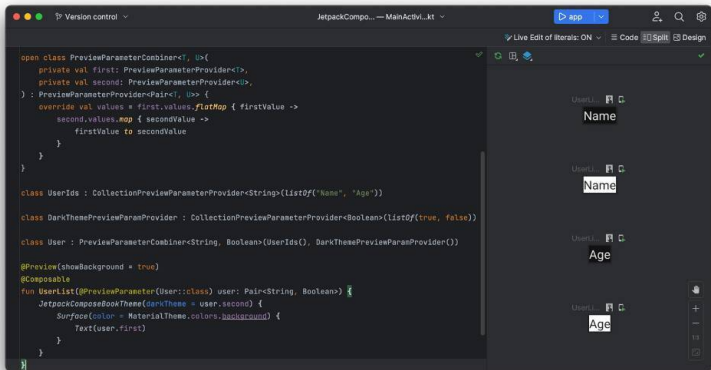


Класс **CollectionPreviewParameterProvider** принимает в конструктор коллекцию, в отличие от **PreviewParameterProvider**, которой требуется переопределять каждый раз переменную **value** типа **Sequence**.

Несколько **provider** классов можно объединить в **Pair** при помощи класса обертки. В качестве **provider** классов можно использовать как **CollectionPreviewParameterProvider**, так и **PreviewParameterProvider**. Данный подход полезен, когда необходимо отобразить один и тот же **mock** в разном окружении, например в темной и светлой теме – и проанализировать – как элемент будет себя вести в каждой из них.

Перейдите в **Notion**, чтобы подробнее рассмотреть код создания **mock**,

provider **классов**— [<https://inky-belief-259.notion.site/Preview-909c458b613f49eabbd7dc67235382e5>].



В **Preview** можно производить какие-то действия в боковом меню, без запуска в эмуляторе, для этого нужно нажать на иконку с указательным пальцем. Вторая кнопка запускает экран в эмуляторе.

Еще одной полезной фишкой в **Preview** является группирование. Внутри аннотации укажите **group** и присвойте ей любое название. Так вы сможете быстро отыскать определенную группу элементов в **Preview**.

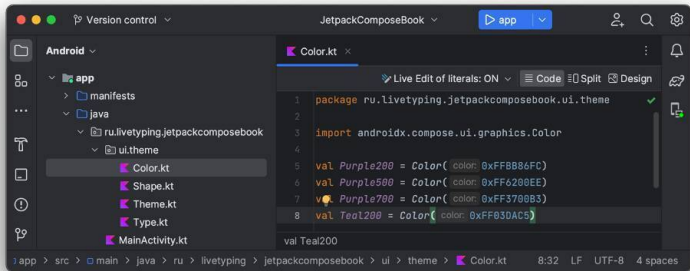
Стиль написания кода

Jetpack Compose обязывает писать название функций с заглавной буквы, далее все как в **camel case** стиле. У всех

Composable, функций есть набор параметров, именованные параметры должны идти в верном порядке во избежание ошибок. Прочие названия ресурсов, таких как цвет, шрифт и т.п. должны следовать той же логике.

Структура проекта

В отличие от проектов, где используется **XML**, вы не встретите пакета **layout**. Теперь весь интерфейс расположен внутри самого проекта. Помимо этого, **Jetpack Compose** предоставляет дополнительные файлы для стилей, шрифтов, тем. Первый файл в пакете **ui.theme** – **Color**. Внутри располагаются цвета, которые можно вызвать внутри интерфейса по их имени.

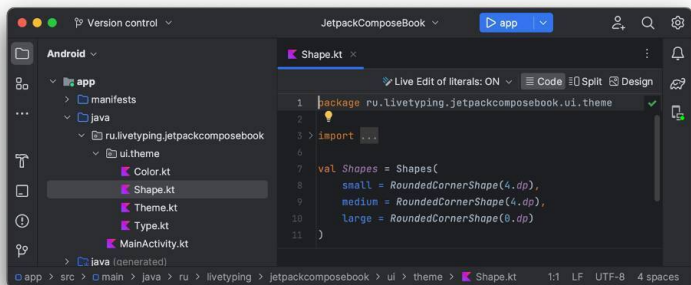


```
1 package ru.livotyping.jetpackcomposebook.ui.theme
2
3 import androidx.compose.ui.graphics.Color
4
5 val Purple200 = Color( color: 0xFFB886FC)
6 val Purple500 = Color( color: 0xFF628EE)
7 val Purple700 = Color( color: 0xFF3788B3)
8 val Teal200 = Color( color: 0xFF63D8C5)
9
10 val Teal200
```

Сперва указывается **0x**, после прозрачность **FF** и сам хэш код цвета.

Второй файл – **Shape**. Для **XML** приложений часто при-

ходилось создавать множество файлов с различными формами для элементов. В **Jetpack Compose** это располагается в одном месте и вам при этом необходимо передать внутрь только параметр, на сколько нужно закруглить **View**.

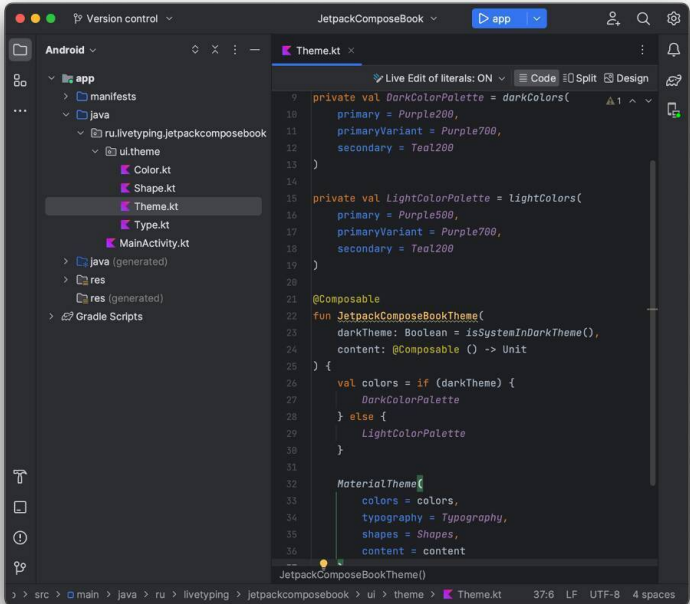


Для указания размера шрифта или другого элемента, используют следующий синтаксис: **число.dp**, **число.sp**.

Третий файл отвечает за цветовую схему в приложение, **background**, цвет текста и так далее. **DarkColorPalette** содержит в себе цвета для темной темы, можно дополнительно переопределить другие цвета, если это необходимо. **LightColorPalette** содержит цвета для светлой темы.

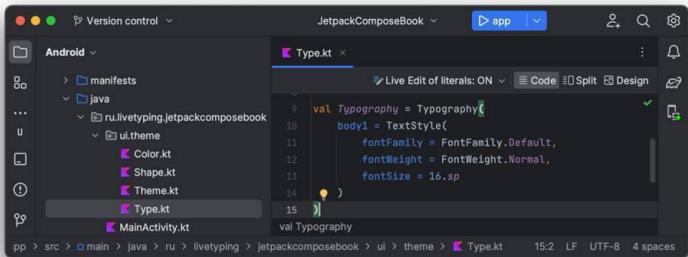
После определения светлой и темной темы, расположена **Composable** функция, которая отвечает за цвета в проекте. Таких функций может быть бесчисленное множество. Они

вызываются внутри лямбда-блока `setContent` как это было реализовано в начальном проекте, который сгенерировала Android-студия.



```
9 private val DarkColorPalette = darkColors(
10     primary = Purple200,
11     primaryVariant = Purple700,
12     secondary = Teal200
13 )
14
15 private val LightColorPalette = lightColors(
16     primary = Purple500,
17     primaryVariant = Purple700,
18     secondary = Teal200
19 )
20
21 @Composable
22 fun JetpackComposeBookTheme(
23     darkTheme: Boolean = isSystemInDarkTheme(),
24     content: @Composable () -> Unit
25 ) {
26     val colors = if (darkTheme) {
27         DarkColorPalette
28     } else {
29         LightColorPalette
30     }
31
32     MaterialTheme(
33         colors = colors,
34         typography = Typography,
35         shapes = Shapes,
36         content = content
37     )
38 }
39
40 JetpackComposeBookTheme()
```

Файл `Type` содержит настройки для шрифтов, внутри функции `TextStyle` можно задать название шрифта, вес, размер и т.д.



Жизненный цикл и рекомпозиция

Отрисовка элементов в **Jetpack Compose** разделяется на три этапа. **Composition, Layout, Drawing**.

1. **Composition**: какой **UI** показывать. **Compose** запускает **Composable** функции и создает описание вашего **UI**.

2. **Layout**: где размещать **UI**. Этот шаг состоит из двух: измерение и размещение. Элементы верстки измеряют и помещают самих себя и все дочерние элементы.

3. **Drawing**: как рендерить. **UI**-элементы отрисовываются в **Canvas**, обычно на экране устройства.

Jetpack Compose не всегда выполняет все эти три этапа при обновлении данных или отображении элементов. Будут перерисованы только те **Composable** функции, в которых произошли изменения, что хорошо отражается на производительности.

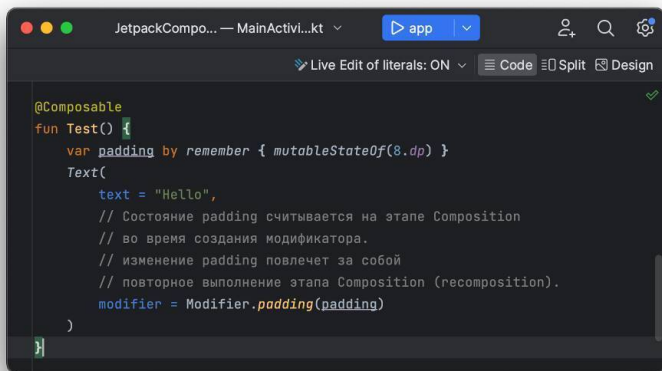
Внутри каждого из состояний происходит считывание со-

стояния, опишем, что происходит внутри каждого из них.

Этап 1: Composition.

Чтение состояния в **Composable** функции или лямбда-блоке влияет на **Composition** и потенциально на следующие этапы. Когда значение состояния меняется, **recomposer** планирует перезапуск всех **Composable** функций, которые его считывали. Обратите внимание, что среда выполнения может решить пропустить некоторые или все **Composable** функции, если входные данные не изменились.

В зависимости от результата **Composition**, Compose UI запускает этапы **Layout** и **Drawing**. Эти этапы могут быть пропущены, если контент не изменился, и, следовательно, общий размер элементов не изменится.

A screenshot of an IDE window titled "JetpackCompo... — MainActivity...kt". The window shows a Kotlin code snippet for a Composable function. The code defines a function "Test" with a parameter "padding" that is remembered. Inside the function, a "Text" widget is created with the text "Hello" and a modifier that applies the "padding" state. Comments in Russian explain that the padding state is read during the Composition phase and that changes to padding will trigger a recomposition. The IDE interface includes a top bar with a play button labeled "app", and a bottom bar with options for "Live Edit of literals: ON", "Code", "Split", and "Design".

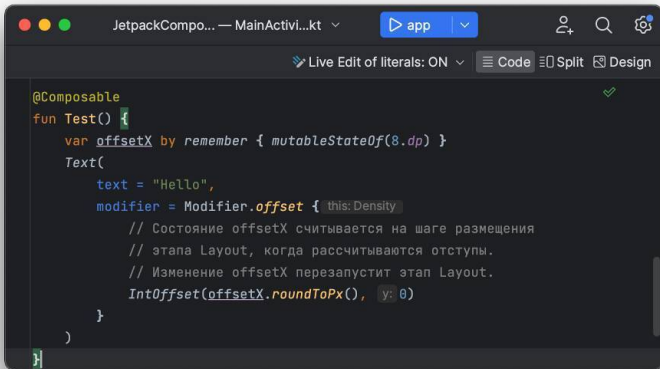
```
@Composable
fun Test() {
    var padding by remember { mutableStateOf(8.dp) }
    Text(
        text = "Hello",
        // Состояние padding считывается на этапе Composition
        // во время создания модификатора.
        // изменение padding повлечет за собой
        // повторное выполнение этапа Composition (recomposition).
        modifier = Modifier.padding(padding)
    )
}
```

Этап 2: Layout.

Этап **Layout** включает два шага: измерение и размещение. Шаг измерения запускает лямбда-функции измерения, переданные в **Layout-composable**, метод **MeasureScope.measure** интерфейса **LayoutModifier**. Размещение запускает блок функции **layout**, лямбду из **Modifier.offset {...}** и т.д.

Считывание состояний во время каждого шага затрагивает этапы **Layout** и, потенциально, **Drawing**. Когда значение состояния меняется, **Compose UI** планирует выполнение этапа **Layout**. Это также запускает этап **Drawing**, если размер или расположение изменились.

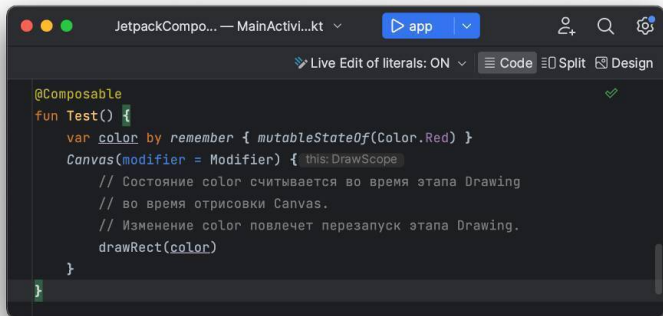
Если быть более точным, шаги измерения и размещения имеют различные области перезапуска. То есть изменение прочитанного состояния на шаге размещения не вызывает повторно шаг измерения, который шел раньше. Однако эти два шага часто взаимосвязаны, так что чтение состояния на шаге размещения может повлиять на области перезапуска, которые относятся к шагу измерения.



```
@Composable
fun Test() {
    var offsetX by remember { mutableStateOf(8.dp) }
    Text(
        text = "Hello",
        modifier = Modifier.offset { this: Density
            // Состояние offsetX считается на шаге размещения
            // этапа Layout, когда рассчитываются отступы.
            // Изменение offsetX перезапустит этап Layout.
            IntOffset(offsetX.roundToPx(), y: 0)
        }
    )
}
```

Этап 3: Drawing.

Чтение состояния внутри кода отрисовки влияет на этап **Drawing**. Распространенные примеры включают: **Canvas**, **Modifier.drawBehind** и **Modifier.drawWithContent**. Когда значение считанного на этом этапе состояния меняется, **Compose UI** запускает только этап **Drawing**.



```
JetpackCompo... — MainActivi...kt
Live Edit of literals: ON
Code Split Design

@Composable
fun Test() {
    var color by remember { mutableStateOf(Color.Red) }
    Canvas(modifier = Modifier) { this: DrawScope
        // Состояние color считывается во время этапа Drawing
        // во время отрисовки Canvas.
        // Изменение color повлечет перезапуск этапа Drawing.
        drawRect(color)
    }
}
```

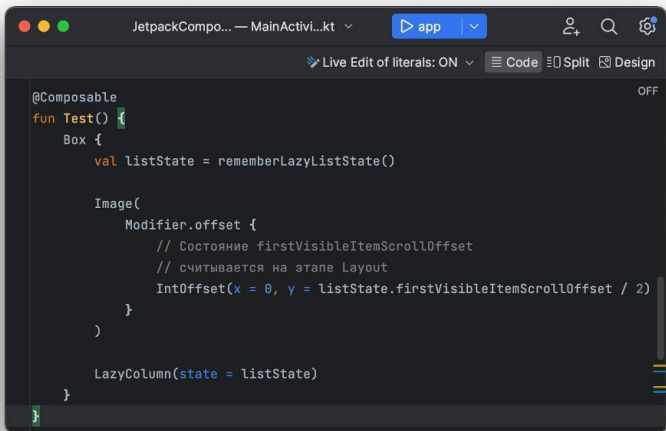
Оптимизация считывания состояния. Поскольку **Compose** выполняет отслеживание считывания состояний внутри этапов, мы можем минимизировать количество работы, выполняемой считыванием каждого состояния на этапах.

```
@Composable
fun Test() {
    Box {
        val listState = rememberLazyListState()

        Image(
            // Неоптимальная реализация!
            Modifier.offset(
                with(LocalDensity.current) {
                    // считывание firstVisibleItemScrollOffset
                    // на этапе Composition
                    (listState.firstVisibleItemScrollOffset / 2).toDp()
                }
            )
        )
    }

    LazyColumn(state = listState)
}
```

Посмотрим на пример ниже. У нас есть **Image**, который использует **offset-модификатор** для смещения своего положения. В результате во время скроллинга пользователь наблюдает эффект параллакса за счет добавления **offset**. Этот код работает, но дает неоптимальную производительность.



```
@Composable
fun Test() {
    Box {
        val listState = rememberLazyListState()

        Image(
            Modifier.offset {
                // Состояние firstVisibleItemScrollOffset
                // считывается на этапе Layout
                IntOffset(x = 0, y = listState.firstVisibleItemScrollOffset / 2)
            }
        )

        LazyColumn(state = listState)
    }
}
```

По мере прокрутки пользователем значение **firstVisibleItemScrollOffset** будет меняться. Как мы знаем, **Compose** отслеживает любое чтение состояния, чтобы можно было повторно вызвать считывающий этот состояние код, в нашем случае – содержимое **Box**.

В этом примере состояние читается внутри этапа **Composition**. Это необязательно плохо. Фактически – это основа рекомпозиции, позволяющая при изменении данных создавать новый UI. Причина не оптимальности кода в примере выше в том, что каждое событие скролла приводит к переоценке всего существующего **composable-содержимо-**

го, и затем новому измерению, расположению и финальной отрисовке.

Мы запускаем этап **Composition** на каждую прокрутку, даже если то, что мы показываем, не изменилось, а изменилось только где показываем. Мы можем оптимизировать считывание нашего состояния, чтобы повторно запускать этапы, начиная с **Layout**.

Существует другая версия **offset-модификатора**. Эта версия функции принимает лямбду, которая возвращает итоговый **offset**.

Почему этот способ более производительный? Лямбда, которую мы предоставляем модификатору, вызывается во время этапа **Layout** – если быть точнее, во время шага размещения – что означает, что наше состояние **firstVisibleItemScrollOffset** больше не считывается во время этапа **Composition**. **Compose** отслеживает, когда состояние считано. Поэтому, если значение **firstVisibleItemScrollOffset** меняется, **Compose** должен только перезапустить этапы **Layout** и **Drawing**.

Вы можете спросить: не может ли использование лямбды привести к дополнительным затратам по сравнению с использованием простого значения? Так и есть. Однако выигрыш от чтения состояния на этапе **Layout** перевешивает эти затраты. Значение **firstVisibleItemScrollOffset** меняет каждый кадр в течение прокрутки, и, отложив чтение состояния до этапа **Layout**, мы совсем избегаем повторных этапов

Composition.

Layouts

Основными **layout** в **Jetpack Compose** являются **Box**, **Row**, **Column**. Также **Compose** позволяет использовать аналоги **Constraint Layout**. Все эти компоненты **inline Composable** – функции. Это значит, что другие **Composable** функции могут быть вы-

Equal
Weight

Space
Between

Space
Around

Space
Even

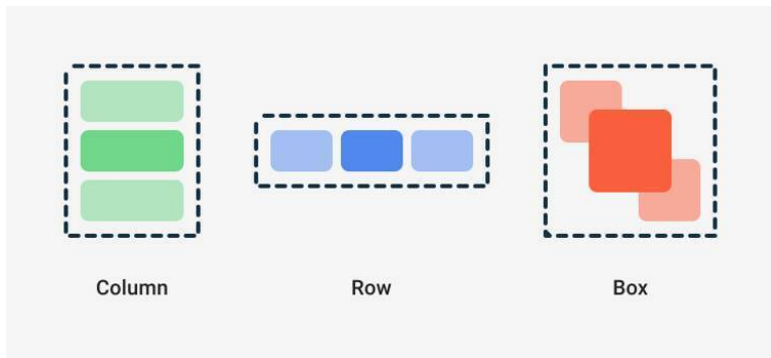
A

A

A

A

ны внутри них.



Layouts могут быть представлены в коде как лямбда-блок, принимающий дополнительные параметры для изменения внешнего вида или поведения компонентов – позиционирование элементов, которые находятся внутри.

Box

Box – аналог **FrameLayout** в **XML**. Нижний элемент будет отображаться поверх остальных, первый выполняет функцию подложки/фона.



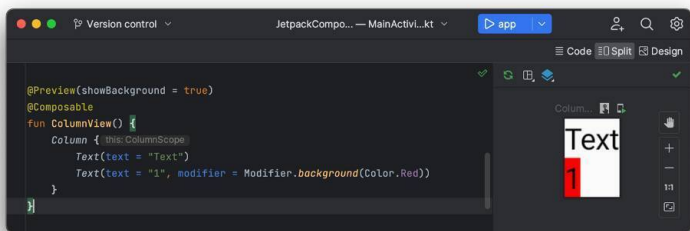
View без обернутого в **Box** имеет схожее поведение. За единственным исключением: вы не сможете контролировать расположение элементов на экране.

Box принимает 4 параметра:

- **modifier** – позволяет настроить внешний вид и его поведение;
- **contentAlignment** – устанавливает расположение элемента на экране, по умолчанию имеет значение **Alignment.TopStart** (расположение вначале контейнера в верхнем углу);
- **propagateMinConstraints** – указывает, надо ли применять к содержимому ограничения минимального размера во время отрисовки, по умолчанию **false**;
- **content** – объект интерфейса **BoxScope**, который подставляет вложенное содержимое, может быть заменен на лямбда-блок.

Column

Column layout – вертикальный список, **LinearLayout** в XML



Column принимает 4 параметра:

- **modifier** – позволяет настроить внешний вид и его поведение;
- **verticalArrangement** – выравнивание элементов по вертикали, по умолчанию имеет значение **Arrangement.Top**;
- **horizontalAlignment** – выравнивание по горизонтали, по умолчанию имеет значение **Alignment.Start**;
- **content** – объект интерфейса **ColumnScope**, который подставляет вложенное содержимое, может быть заменен на лямбда-блок;

VerticalArrangement позволяет изменить позиционирование элементов по вертикали не только стандартными модификаторами, такими как: **Arrangement.Center**, **Arrangement.Bottom**, **Arrangement.Top**, но и более гибкими,

которые позволяют располагать элементы равномерно внутри **layout**.

- **Arrangement.SpaceAround** – компоненты равномерно распределяются по всей высоте с отступами между элементами, при этом отступы между первым и последним элементами равны половине отступов между элементами;

- **Arrangement.SpaceBetween** – компоненты распределяются по всей высоте с равномерными отступами между элементами, при этом первый и последний элементы прижимаются к границам контейнера;

Конец ознакомительного фрагмента.

Текст предоставлен ООО «ЛитРес».

Прочитайте эту книгу целиком, [купив полную легальную версию](#) на ЛитРес.

Безопасно оплатить книгу можно банковской картой Visa, MasterCard, Maestro, со счета мобильного телефона, с платежного терминала, в салоне МТС или Связной, через PayPal, WebMoney, Яндекс.Деньги, QIWI Кошелек, бонусными картами или другим удобным Вам способом.