Kotlin 1.7
Jetpack Compose 1.2

Лайв Тайпинг

Разработка Android приложений с Jetpack Compose

12+

Денис Попков

Разработка Android приложений с Jetpack Compose

«Автор»
2022

Попков Д. С.

Разработка Android приложений с Jetpack Compose /
Д. С. Попков — «Автор», 2022

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.

© Попков Д. С., 2022
© Автор, 2022

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 справится 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 использовать 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**.

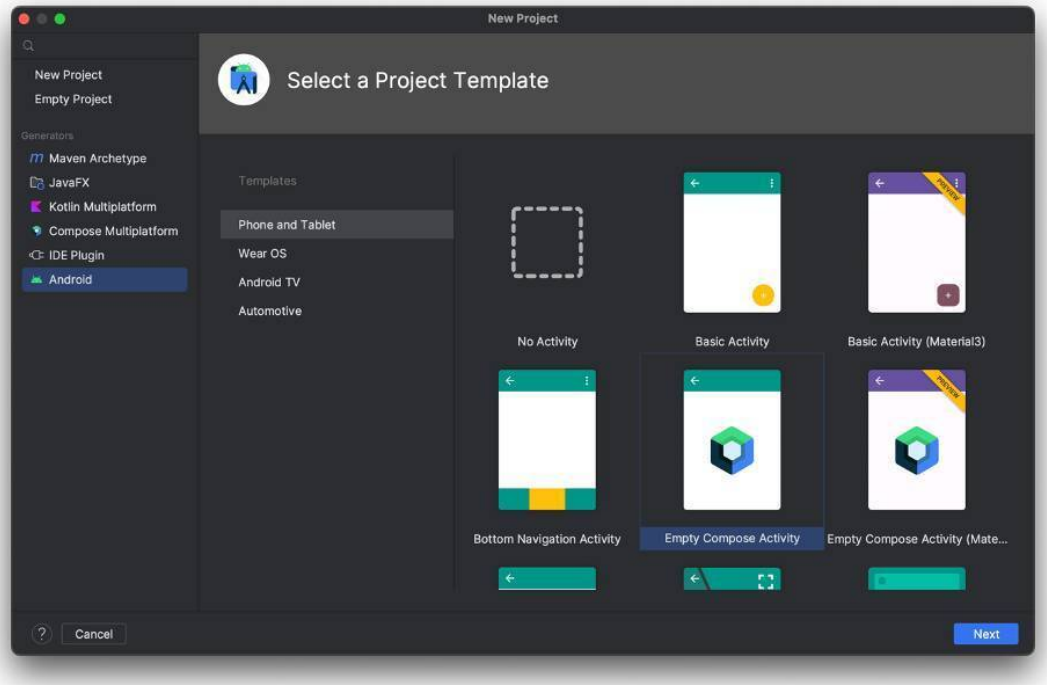
Для кого это руководство?

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 вызываться 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

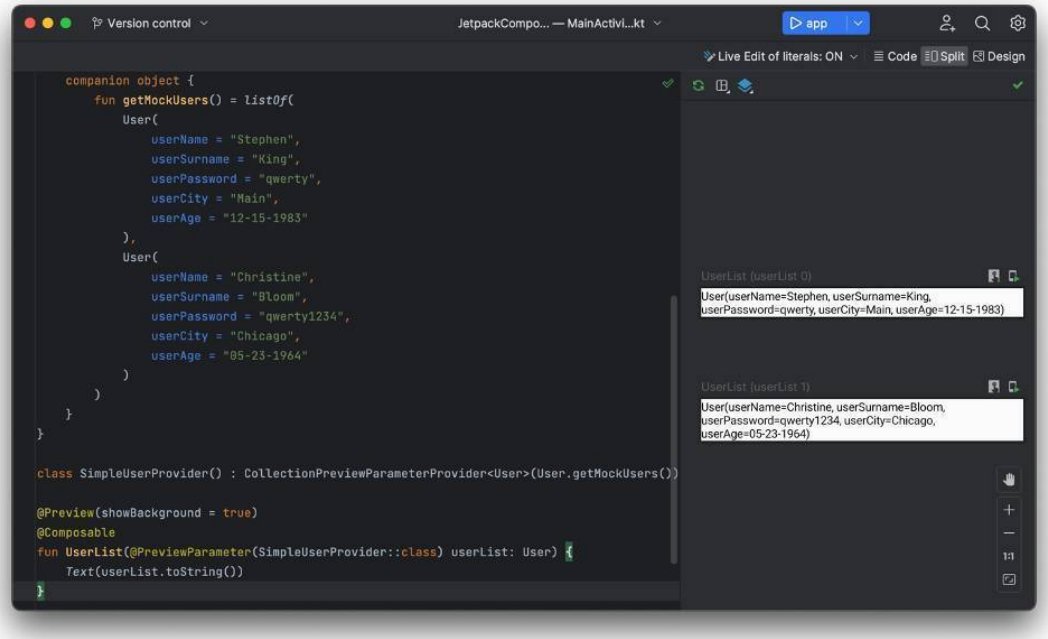
Аннотация Preview отображает в боковом меню сверстанные элементы. Если необходимо отобразить дополнительно фон View, укажите showBackground = true. После внесения изменений нужно нажать build&refresh для обновления Preview, если изменения небольшие, то Preview обновится автоматически.

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.

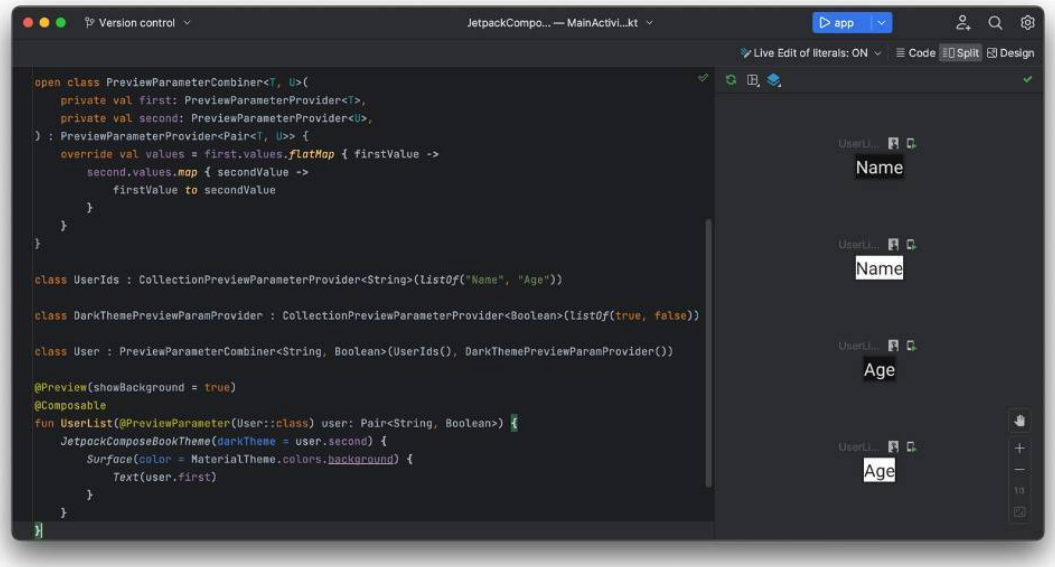
В случае, если **provider** класс возвращает объект, обернутый в список, то **Preview** отобразится единожды, что порой удобно для анализа верстки.



Класс **CollectionPreviewParameterProvider** принимает в конструктор коллекцию, в отличие от **PreviewParameterProvider**, которой требует 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.

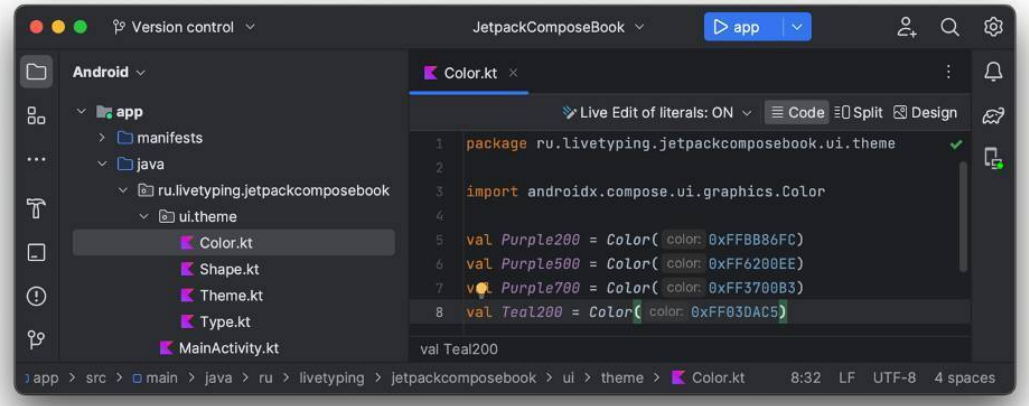
Еще одной полезной фичей в **Preview** является группирование. Внутри аннотации укажите **group** и присвойте ей любое название. Так вы сможете быстро отыскать определенную группу элементов в **Preview**.

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.

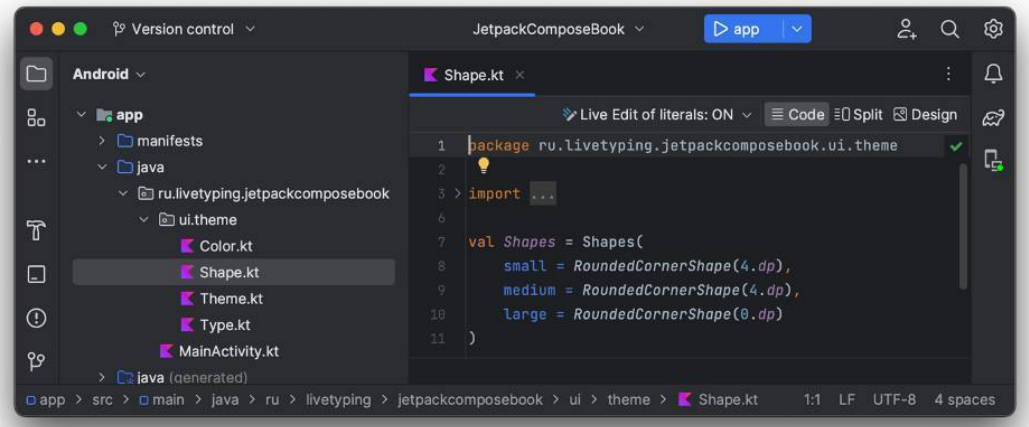
Структура проекта

В отличии 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.



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

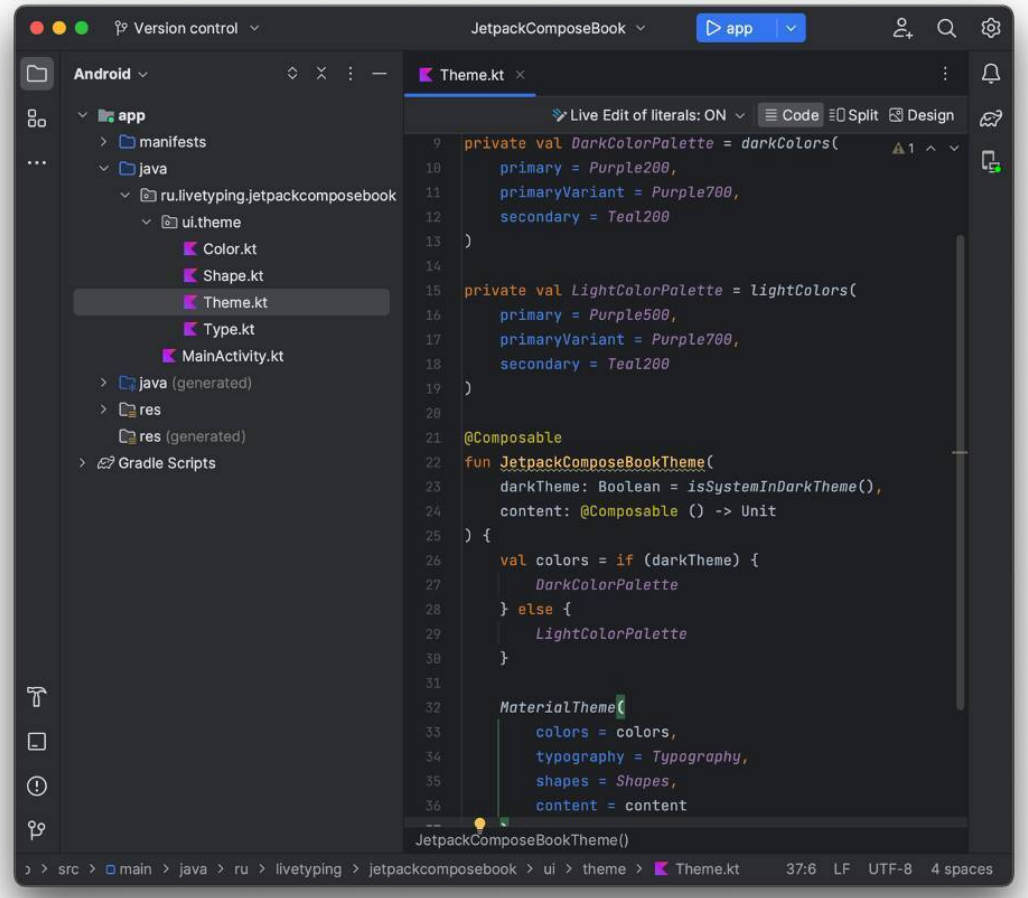
Второй файл – **Shape**. Для **XML** приложений часто приходилось создавать множество файлов с различными формами для элементов. В **Jetpack Compose** это располагается в одном месте и вам при этом необходимо передать внутрь только параметр, на сколько нужно закруглить **View**.



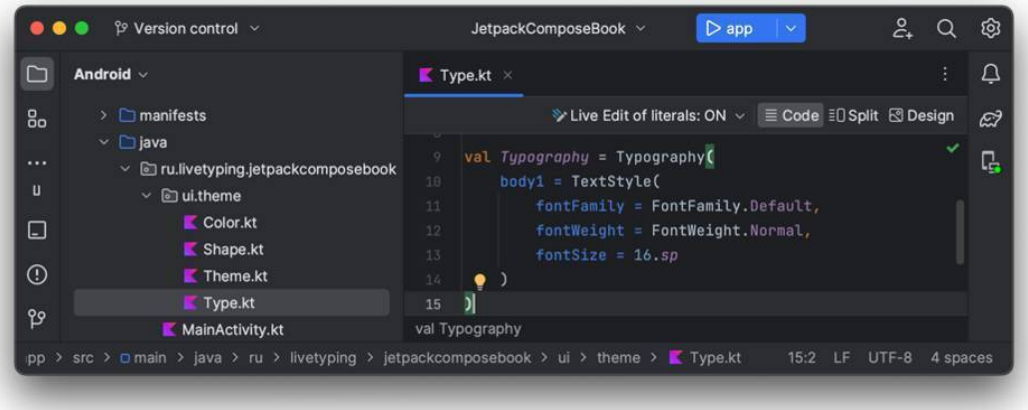
Для указания размера шрифта или другого элемента, используют следующий синтаксис: **число.dp**, **число.sp**.

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.



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.

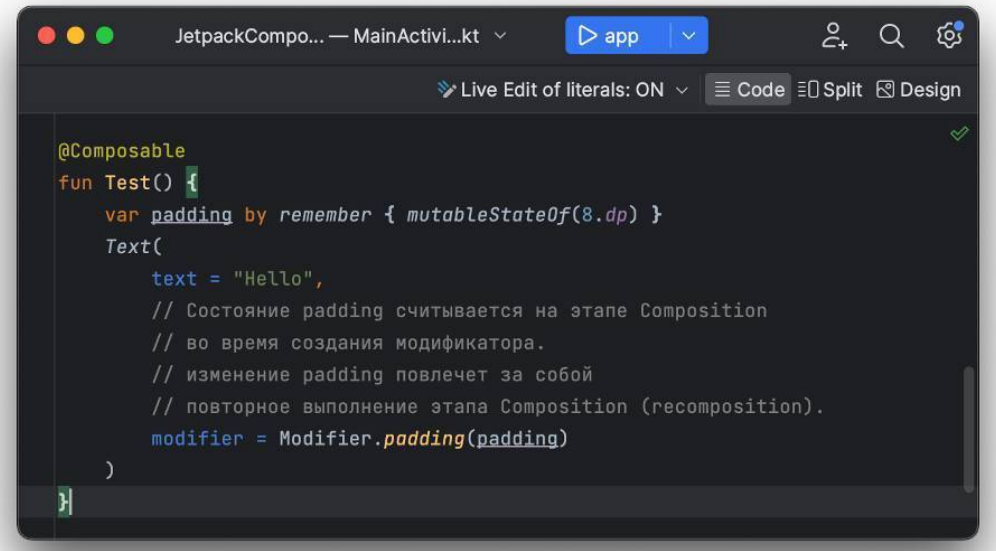
Jetpack Compose не всегда выполняет все эти три этапа при обновлении данных или отображении элементов. Будут перерисованы только те **Composable** функции, в которых произошли изменения, что хорошо отражается на производительности.

Внутри каждого из состояний происходит считывание состояния, 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.

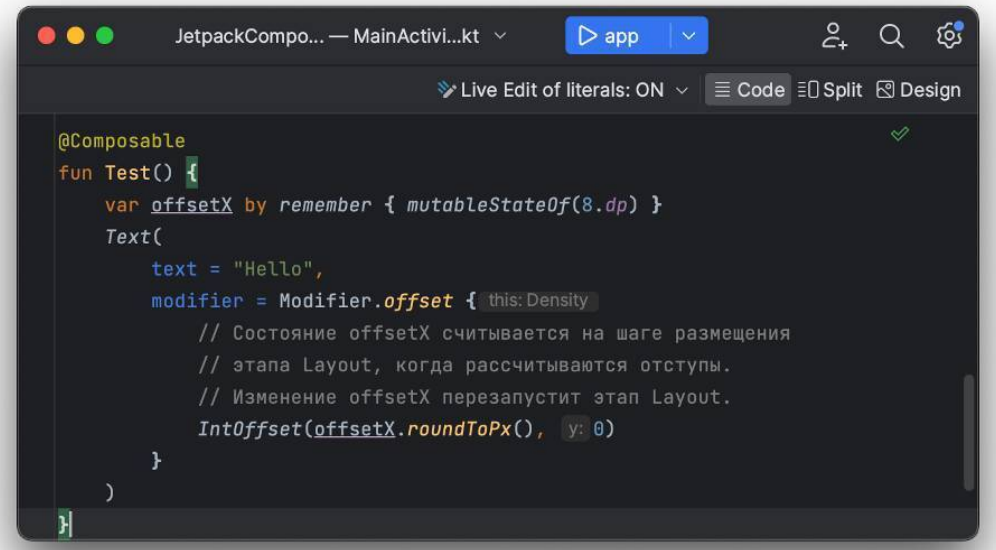


Этап 2: Layout.

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.

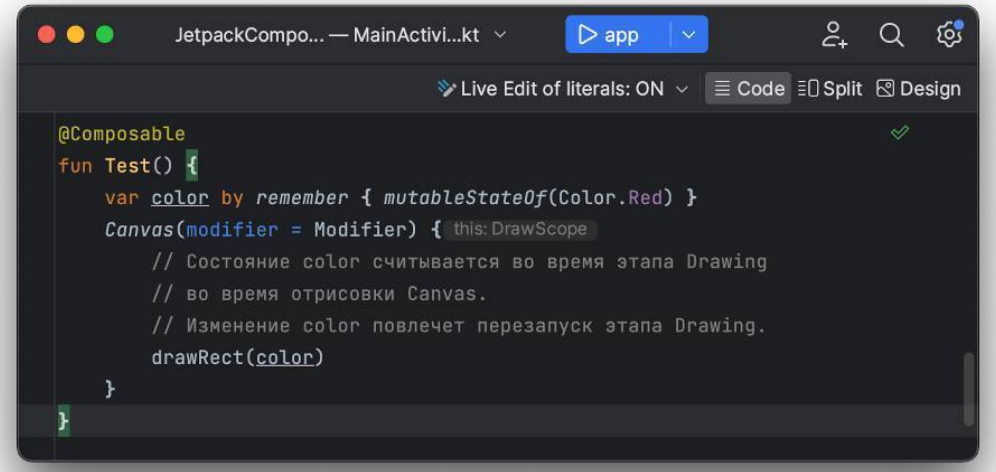
Считывание состояний во время каждого шага затрагивает этапы **Layout** и, потенциально, **Drawing**. Когда значение состояния меняется, **Compose UI** планирует выполнение этапа **Layout**. Это также запускает этап **Drawing**, если размер или расположение изменились.

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.

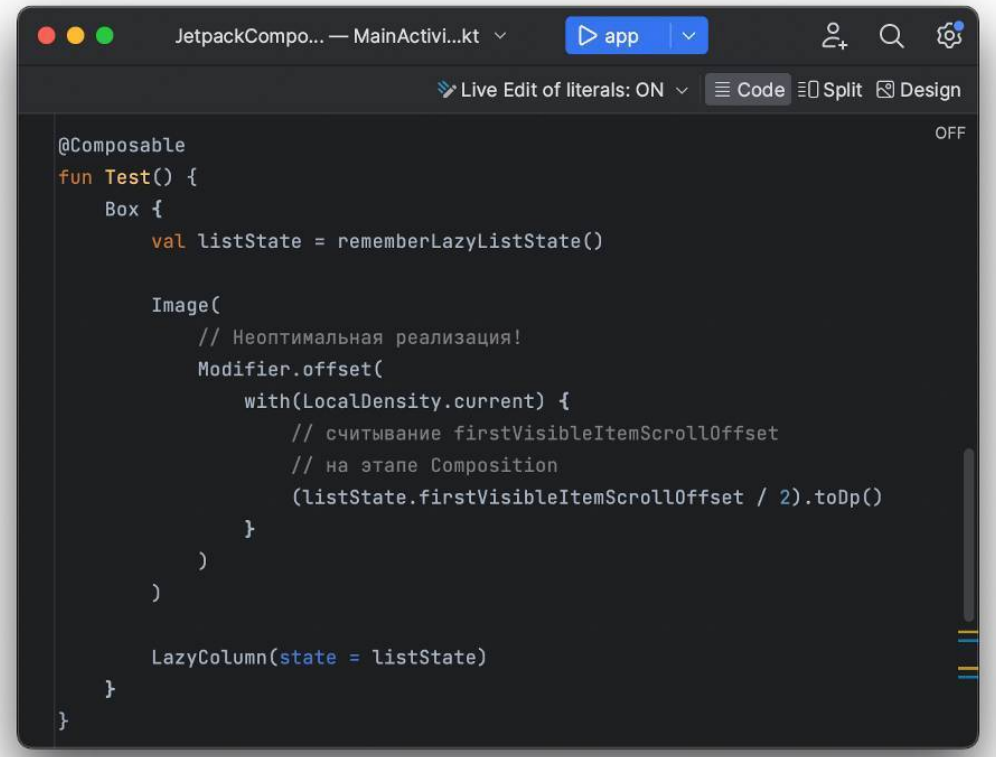


Этап 3: Drawing.

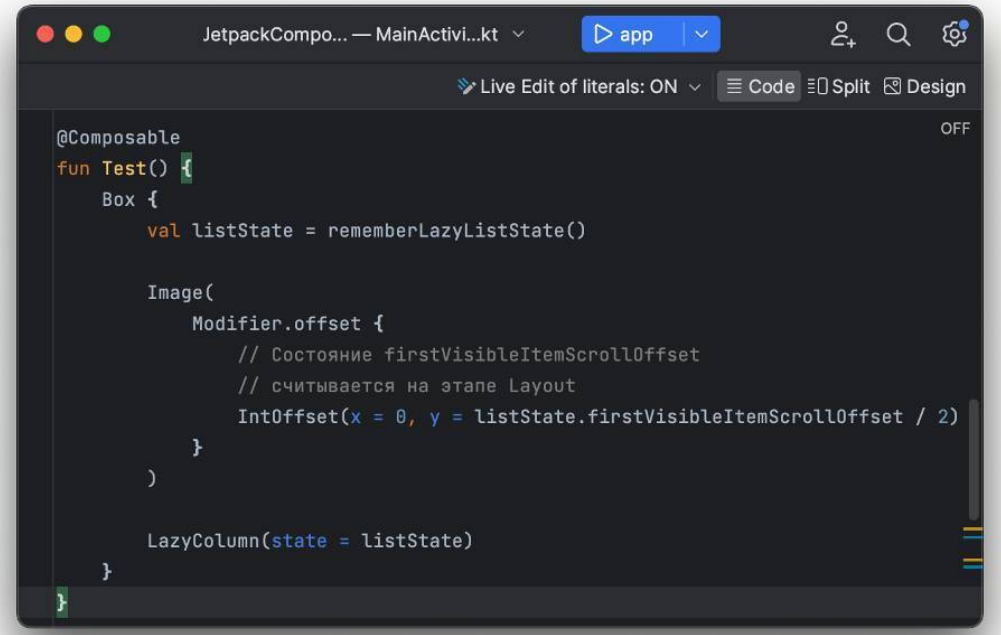
Чтение состояния внутри кода отрисовки влияет на этап **Drawing**. Распространенные примеры включают: **Canvas**, **Modifier.drawBehind** и **Modifier.drawWithContent**. Когда значение считанного на этом этапе состояния меняется, **Compose UI** запускает только этап **Drawing**.



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



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.



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-содержимого**, 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**.

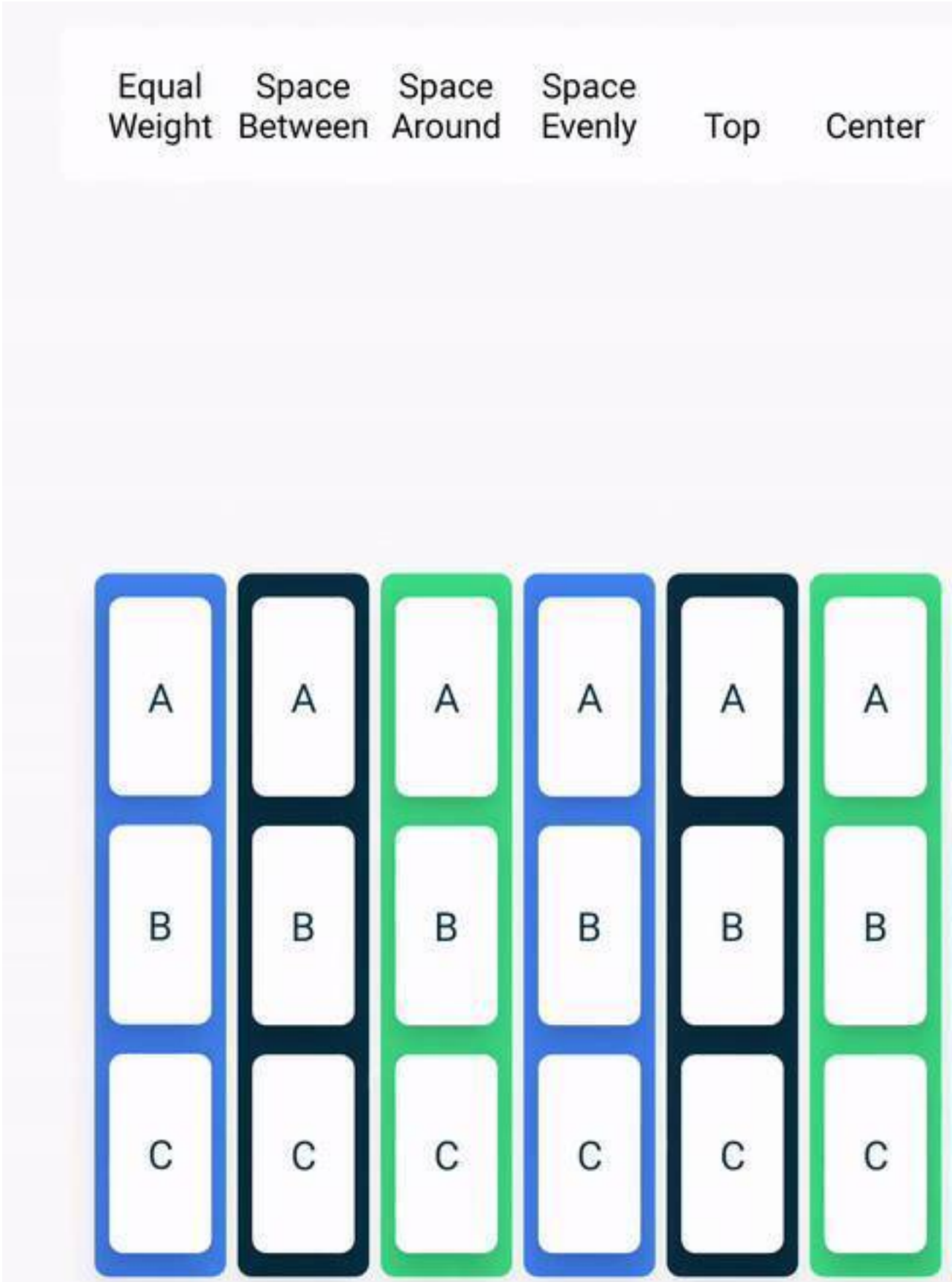
Почему этот способ более производительный? Лямбда, которую мы предоставляем модификатору, вызывается во время этапа **Layout** – если быть точнее, во время шага размещения – что означает, что наше состояние **firstVisibleItemScrollOffset** больше не считывается во время этапа **Composition**. Compose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

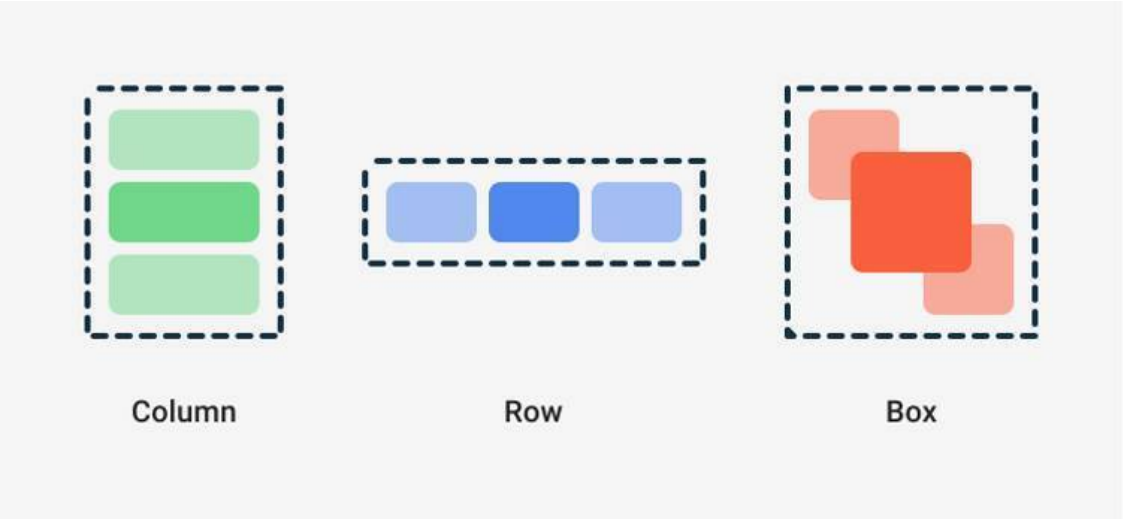
Вы можете спросить: не может ли использование лямбды привести к дополнительным затратам по сравнению с использованием простого значения? Так и есть. Однако выигрыш от чтения состояния на этапе **Layout** перевешивает эти затраты. Значение **firstVisibleItemScrollOffset** меняет каждый кадр в течение 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

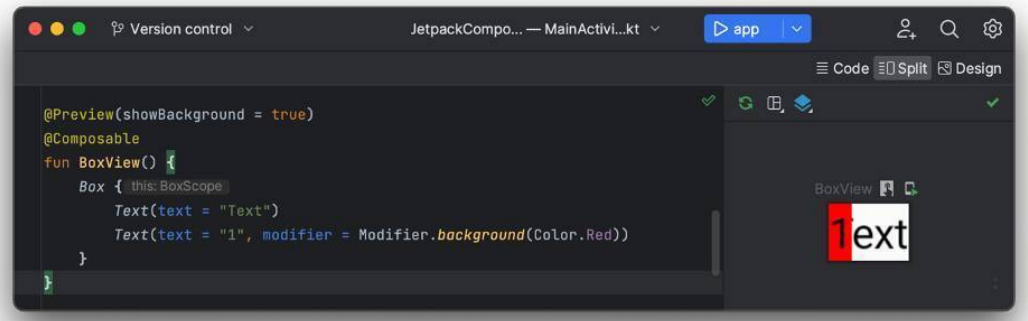

Equal Weight
Space Between
Space Around
Space Evenly
Top
Center
A
B
C



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;

• **verticalArrangment** – выравнивание элементов по вертикали, по умолчанию имеет значение **Arrangment.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;

VerticalArrangment позволяет изменить позиционирование элементов по вертикали не только стандартными модификаторами, такими как: **Arrangment**.**Center**, **Arrangment**.**Bottom**, **Arrangment**.**Top**, но и более гибкими, которые позволяют располагать элементы равномерно внутри **layout**.

• **Arrangment.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.