# Часть I – Создание UI

В Jetpack Compose Modifier (*androidx.compose.ui.Modifier*) — это оболочка, позволяющая настраивать поведение и внешний вид компонуемого элемента пользовательского интерфейса. Он действует как набор инструкций, изменяющих способ отрисовки, расположения и взаимодействия компонуемого элемента с пользователем. По сути, он позволяет добавлять стили, обрабатывать события и управлять компоновкой компонентов пользовательского интерфейса, не изменяя их базовую реализацию.

Размер и расположение:

* size(width, height): Установка фиксированного размера Composable фунцкции.
* fillMaxSize(): Растягивание Composable функции по всей поверхности
* fillMaxWidth(): Растягивание Composable функции по всей ширине
* wrapContentSize(): Масштабирование размера до вмещения всего контента.
* padding(all), padding(horizontal, vertical): отступы вокруг Composable функции.
* align(alignment): Расположение Composable функции.
* requiredSize(width, height): Установка фиксированного размера, даже если он выходит за границы родителя.
* weight(weight): Установка веса Composable.

Стили:

* background(color): Установка цвета фона.
* border(width, color, shape): Установка рамки объекта
* clip(shape): Обрезание Composable по форме.
* shadow(elevation, shape, clip): добавление тени для composable.
* alpha(alpha): Устнаовка непрозрачности.

Взаимодействие:

* clickable(onClick): Делает элемент доступным для нажатия.
* combinedClickable(onClick, onLongClick): Добавляет возможность обрабатывать нажатие и удержание.
* pointerInput(key1, block): Обработка жестов.

Прочее:

* blur(radiusX, radiusY, edgeTreatment): Добавление размытия Composable.
* rotate(degrees): Поворот и натсройка угла Composable функции.
* scale(scaleX, scaleY): Масштабирование.

## Расположение элементов

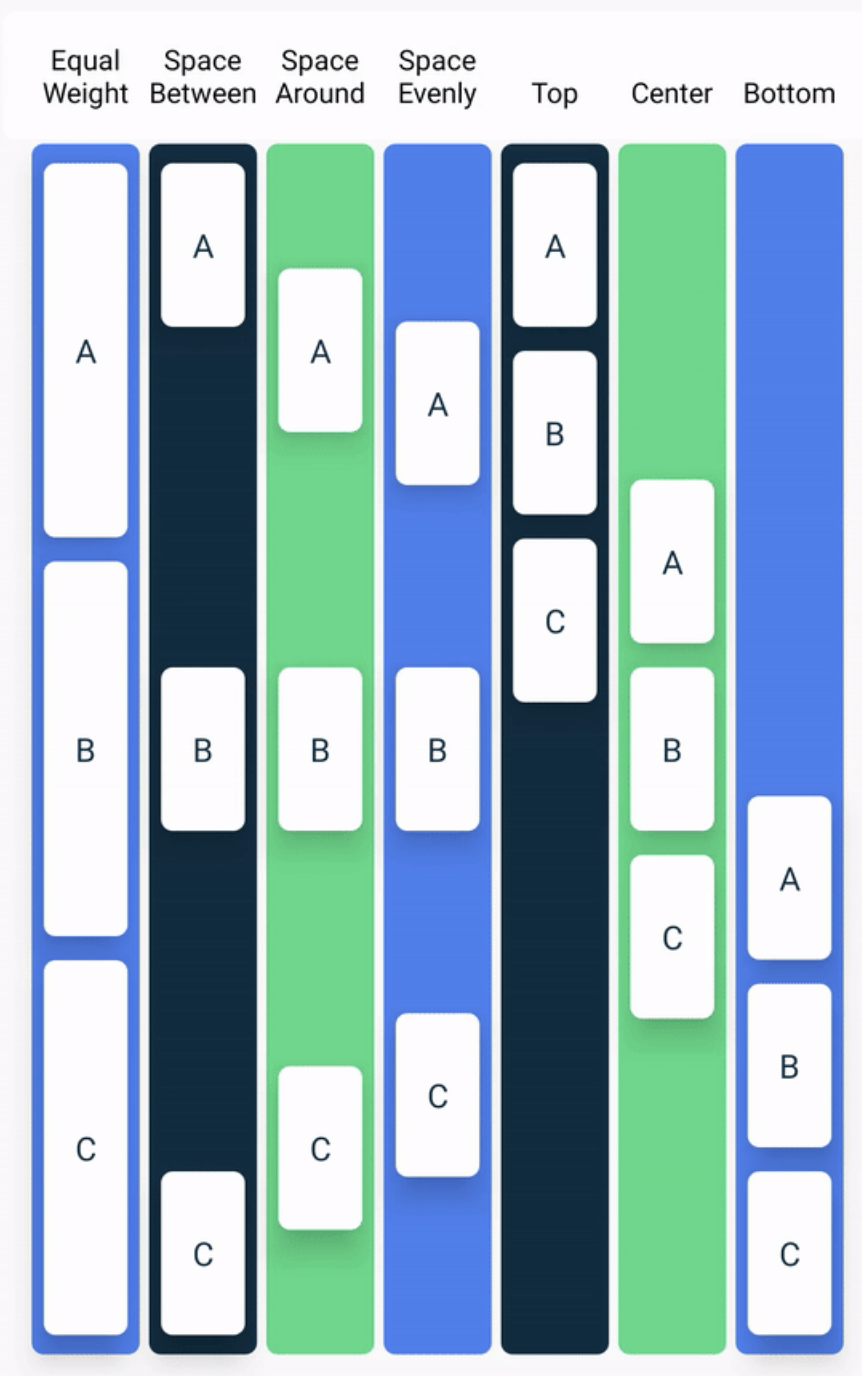
В Jetpack Compose *Column()* — это компонуемый макет, в котором дочерние элементы располагаются вертикально, один под другим. Это фундаментальный строительный блок для создания вертикальных интерфейсов.

modifier: объект Modifier, который позволяет настроить внешний вид и поведение компонента

verticalArrangement: объект Arrangement.Vertical, который устанавливает выравнивание компонента по вертикали. По умолчанию имеет значение Arrangement.Top (расположение в верху)

horizontalAlignment: объект Alignment.Horizontal, который устанавливает выравнивание компонента по горизонтали. По умолчанию имеет значение Alignment.Start (расположение в начале - слева для языков с левосторонним письмом и справа для языков с правосторонним письмом)

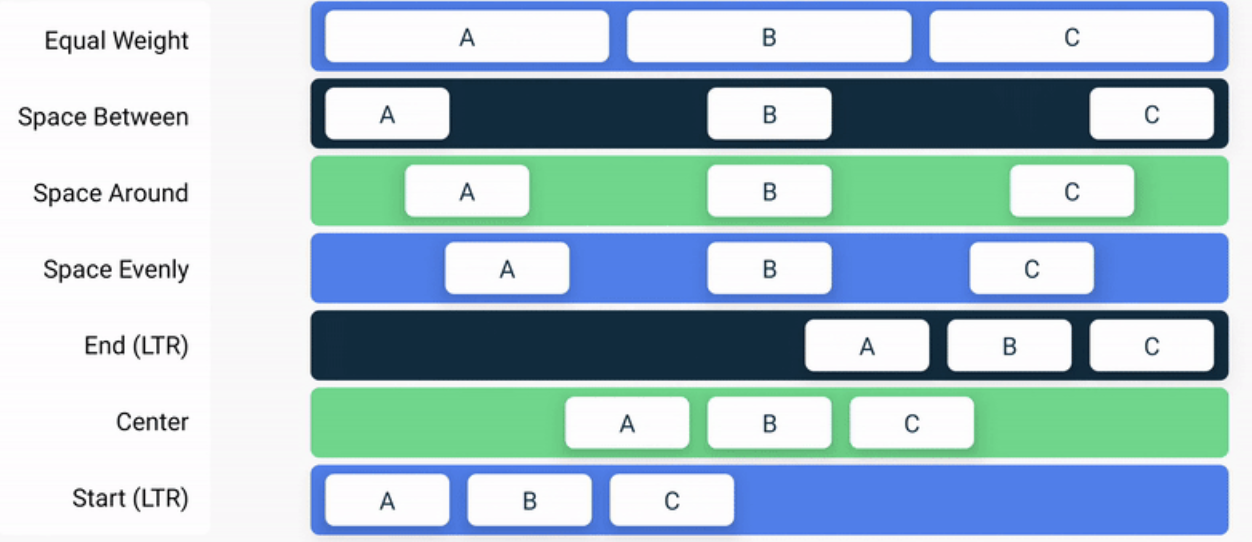
Модификаторы для расположения элементов:



В Jetpack Compose Row — это компонуемый макет, который размещает дочерние элементы горизонтально, аналогично LinearLayout с горизонтальной ориентацией в традиционных представлениях Android.

horizontalArrangement: объект Arrangement.Horizontal, который устанавливает выравнивание компонента по горизонтали. По умолчанию имеет значение Arrangement.Start (расположение в вначале: слева для левосторонних языков и справа для правосторонних языков)

verticalAlignment: объект Alignment.Vertical, который устанавливает выравнивание компонента по вертикали. По умолчанию имеет значение Alignment.Top (расположение вверху)



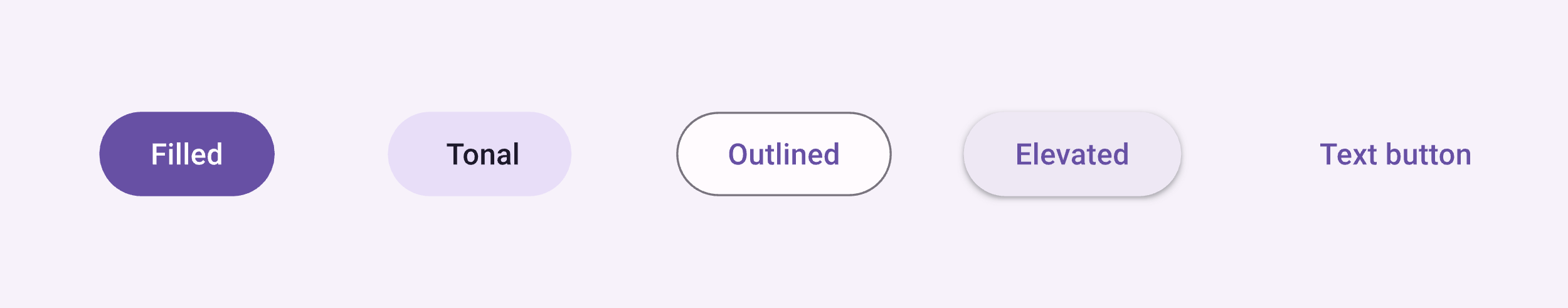
Пример:

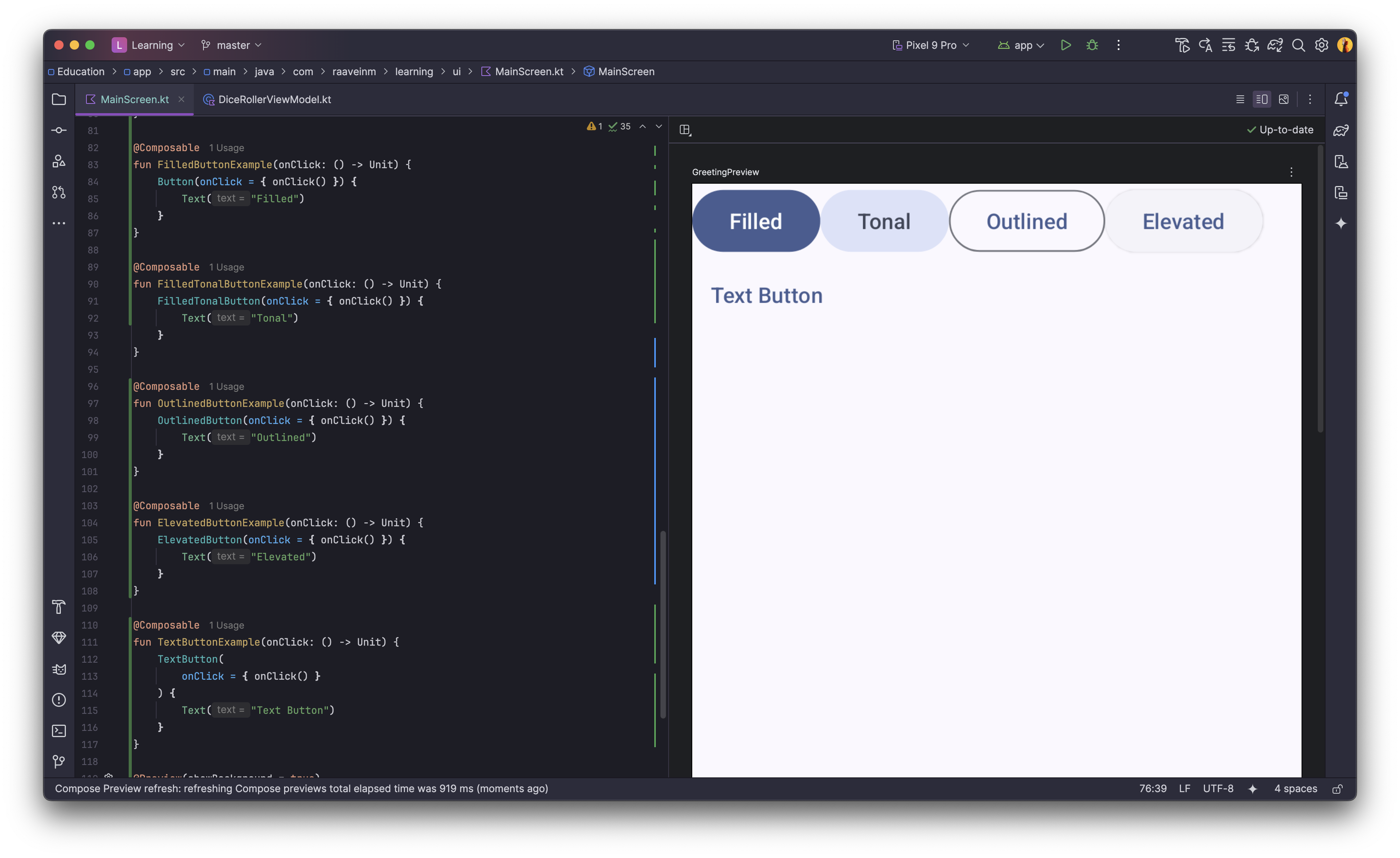
@Composable  
fun MainScreen(  
 modifier: Modifier = Modifier  
){  
 // Создание родительского контейнера  
 Scaffold (  
 // Установка Modifier с наслаиванием аттрибутов  
 // те что были переданы при вызове MainScreen и записаны в переменную modifier  
 modifier = modifier  
 .*fillMaxSize*()**,** // Заполнение контентом  
 content = **{** Column (  
 // Заполнение "чистого" Modifier  
 modifier = Modifier  
 .*padding*(**it**)  
 .*wrapContentSize*()**,** // Для Column базовой осью является вертикаль  
 // Центровка вертикали с левого края будет выглядеть так:  
 verticalArrangement = Arrangement.Center**,** horizontalAlignment = Alignment.Start  
 ) **{** Row (  
 modifier = Modifier.*fillMaxWidth*()**,** // Для Row базовой осью является горизонталь  
 // Центровка вертикали с левого края будет выглядеть так:  
 horizontalArrangement = Arrangement.Start**,** verticalAlignment = Alignment.CenterVertically  
 ) **{  
  
 }  
 }  
 }** )  
}

## Button

Кнопки — это основные компоненты, позволяющие пользователю выполнять определённое действие. Существует пять типов кнопок. В следующей таблице описан внешний вид каждого из пяти типов кнопок, а также области их применения.

|  |  |  |
| --- | --- | --- |
| Тип | Внешний вид | Назначение |
| Filled | Сплошной фон с контрастным текстом. | Кнопки с высоким уровнем выделения. Предназначены для основных действий в приложении, таких как «Отправить» и «Сохранить». Эффект тени подчеркивает важность кнопки. |
| Filled tonal | Цвет фона варьируется в зависимости от поверхности. | Также для основных или важных действий. Заполненные кнопки визуально более выразительны и подходят для таких функций, как «Добавить в корзину» и «Войти». |
| Elevated | Выделяется тенью | Выполняет ту же функцию, что и тональные кнопки. Увеличьте высоту, чтобы кнопка выглядела еще заметнее. |
| Outlined | Имеет рамку без заливки | Кнопки со средним уровнем выделения, содержащие важные, но не основные действия. Они хорошо сочетаются с другими кнопками для обозначения альтернативных, второстепенных действий, таких как «Отмена» или «Назад». |
| Text | Отображает текст без фона и рамки | Кнопки с низким акцентом, идеально подходящие для менее важных действий, таких как навигационные ссылки или второстепенные функции, например «Узнать больше» или «Просмотреть подробности». |



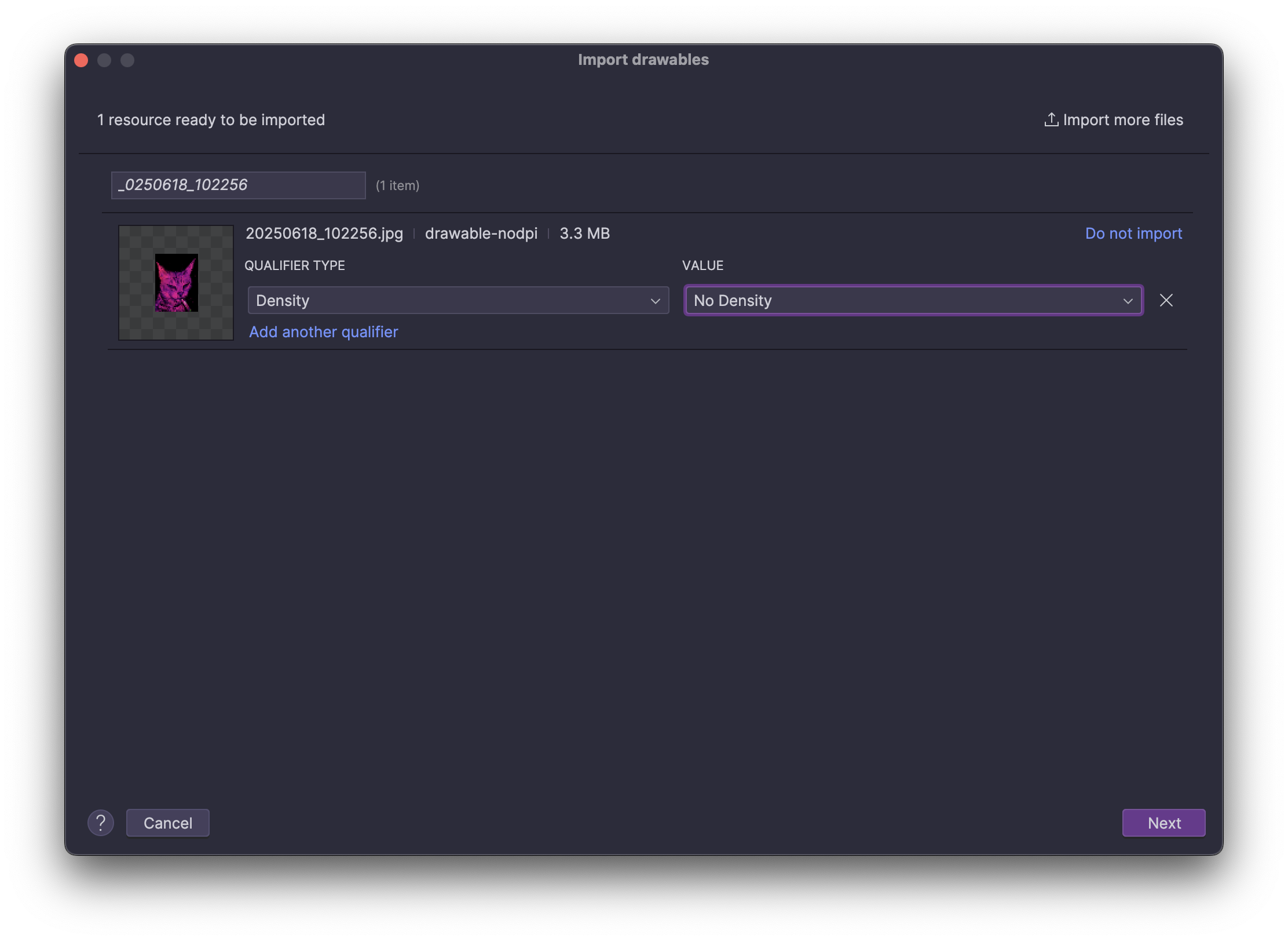


## Image

A screenshot of a computer

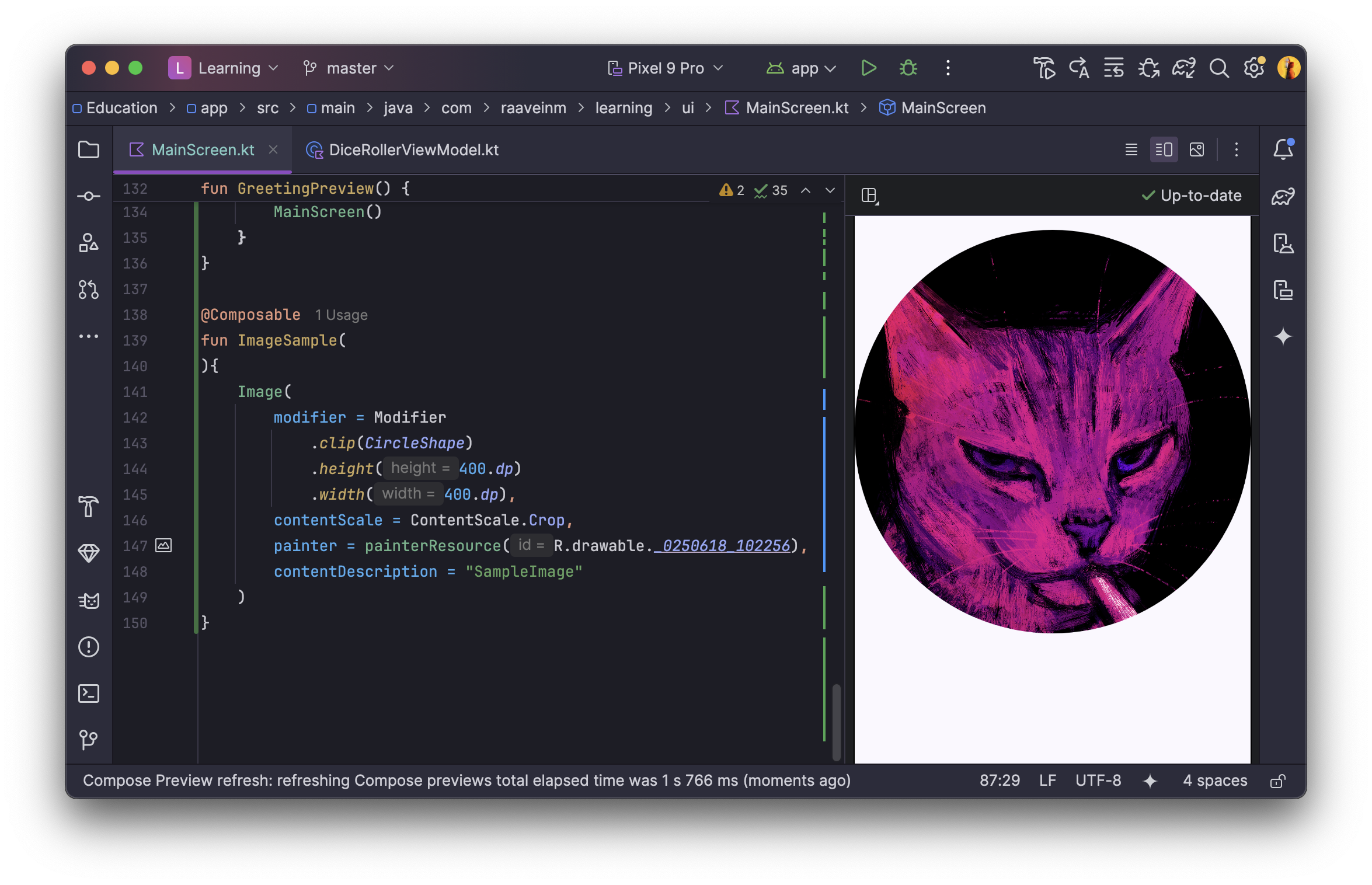
AI-generated content may be incorrect.

Далее необходимо импортировать необходимые изображения:



Для вывода картинки необходимо использовать Composable функцию *Image(painter, contentDescription).* Для использования данной функции необходимо обязательно передать *painter –* картинку и *contentDescription* – Nullable переменная типа String необходимая для дальнейшего обнаружение элемента (в том числе для тестирования).

Подробнее про модификаторы изображений: <https://developer.android.com/develop/ui/compose/graphics/images/customize>



Пример использования:

@Composable  
fun ImageSample(  
){  
 Image(  
 modifier = Modifier  
 .*clip*(*CircleShape*) // Обрезания картинки в круглую форму  
 .*height*(400.*dp*) // Установка высоты  
 .*width*(400.*dp*)**,** // Установка ширины  
 // Обрезание картинки по размерам контейнера  
 contentScale = ContentScale.Crop**,** // Установка ресурса картинки  
 painter = painterResource(R.drawable.*\_0250618\_102256*)**,** contentDescription = "SampleImage" // Установка описания картинки  
 )  
}

## Состояния

Compose — это декларативный фреймворк пользовательского интерфейса, то есть вы определяете, как должен выглядеть пользовательский интерфейс, в коде. Если вы хотите, чтобы в текстовом поле изначально отображалось значение 100, вам нужно установить начальное значение 100 в коде для компонуемых элементов.

Composition — это описание пользовательского интерфейса, создаваемого Compose при выполнении composable элементов. Приложения Compose вызывают composable функции для преобразования данных в пользовательский интерфейс. При изменении состояния Compose повторно выполняет затронутые composable функции с новым состоянием, создавая обновлённый пользовательский интерфейс — это называется рекомпозицией. Compose планирует рекомпозицию автоматически.

Когда Compose впервые запускает ваши composable элементы во время первоначальной композиции, он отслеживает composable элементы, которые вы вызываете для описания пользовательского интерфейса в композиции. Рекмпозиция — это когда Compose повторно выполняет компонуемые элементы, которые могли измениться в ответ на изменение данных, а затем обновляет композицию для отражения всех изменений.

Композиция может быть создана только первоначальной композицией и обновлена при перекомпозиции. Единственный способ изменить композицию — это перекомпозиция. Для этого Compose необходимо знать, какое состояние отслеживать, чтобы он мог запланировать рекомпозицию при получении обновления.

Типы State и MutableState в Compose существуют для того, чтобы Compose мог наблюдать (или отслеживать) состояние приложения. Тип State неизменяемый, поэтому возможно только читать его значение, в то время как тип MutableState — изменяемый. Функция mutableStateOf() позволяет создать наблюдаемое состояние MutableState. Она получает начальное значение в качестве параметра, которое затем обёртывается в объект State, что делает его значение наблюдаемым.

Composabe функции могут сохранять объект между рекомпозициями с помощью функции *remember*. Значение, вычисляемое функцией *remember*, сохраняется в композиции во время начальной композиции, а сохранённое значение возвращается во время рекомпозиции. Обычно функции *remember* и *mutableStateOf* используются вместе в компонуемых функциях для корректного отображения состояния и его обновлений в пользовательском интерфейсе.

@Composable  
fun StateScreen(  
 modifier: Modifier = Modifier  
) {  
 var rememberState by remember **{** *mutableIntStateOf*(0) **}** OutlinedTextField(  
 value = rememberState.toString()**,** onValueChange = **{** rememberState = **it**.*toInt*() **},** modifier = Modifier  
 )  
}

## KeyboardOptions

Компонент KeyboardOptions в Jetpack Compose используется для определения поведения программной клавиатуры, которая появляется при взаимодействии пользователя с полем ввода текста. Он предоставляет ряд свойств, позволяющих настраивать клавиатуру в соответствии с потребностями приложения.

Компонент KeyboardOptions имеет несколько свойств, которые можно использовать для настройки поведения клавиатуры:

keyboardType: объект KeyboardType, задающий тип отображаемой клавиатуры. Это свойство необязательно, но рекомендуется задать ему значение, соответствующее ожидаемому вводу. Например, можно задать его как KeyboardType.Number для поля, принимающего только цифры.

imeAction: объект ImeAction, задающий действие, выполняемое при нажатии пользователем клавиши «Enter» на клавиатуре. Это свойство необязательно, но рекомендуется задать ему значение, соответствующее ожидаемому действию. Например, можно задать его как ImeAction.Done для поля, используемого для ввода пароля.

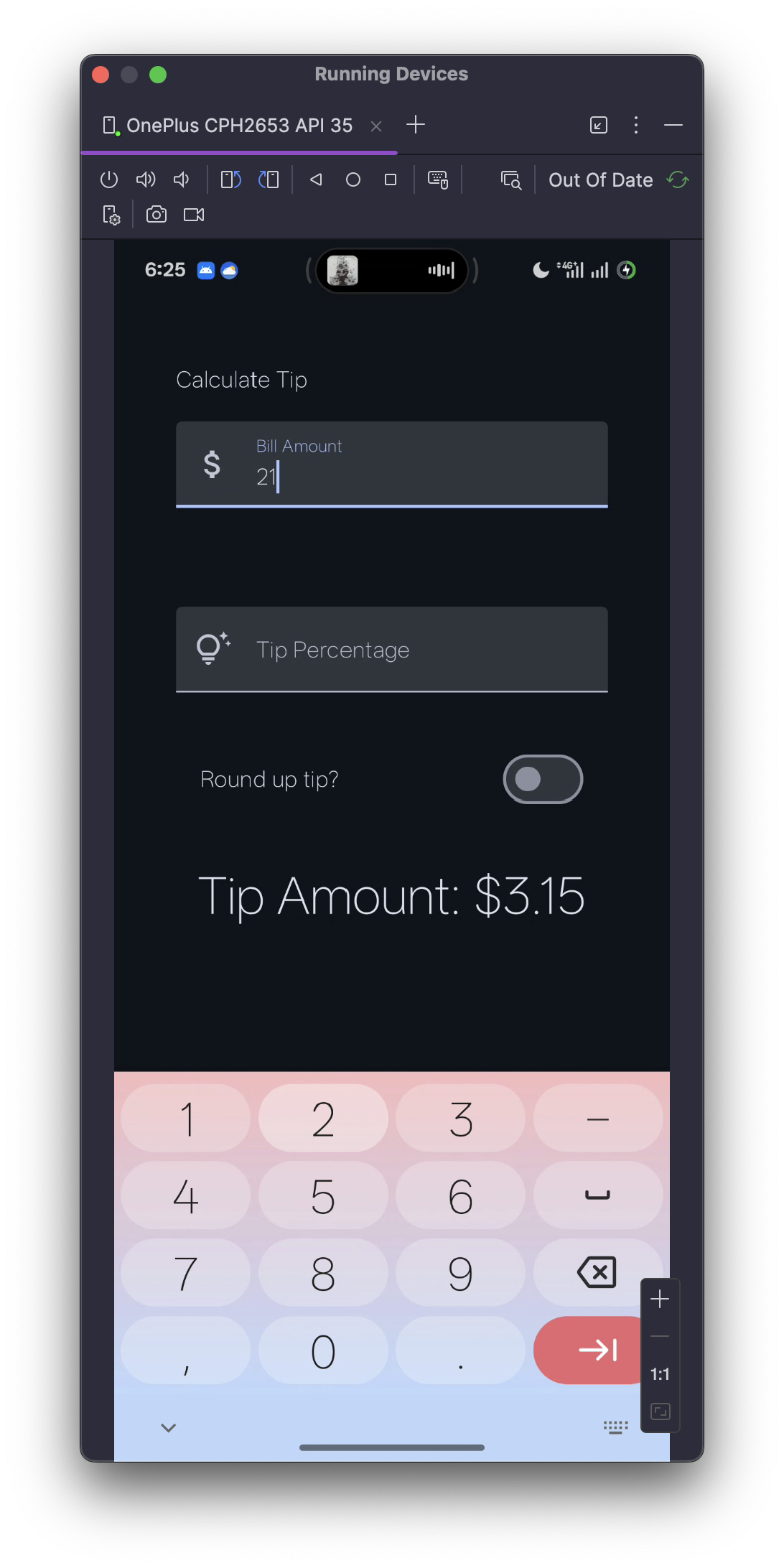
autoCorrect: логическое значение, указывающее, следует ли включать автокоррекцию на клавиатуре. Это свойство необязательно, но рекомендуется задать ему значение true для полей ввода текста, ожидающих ввода на естественном языке.

autoComplete: логическое значение, указывающее, следует ли включать автодополнение на клавиатуре. Это свойство необязательно, но рекомендуется установить его в значение true для полей ввода текста, ожидающих часто используемый ввод.

capitalization: Объект KeyboardCapitalization, определяющий режим использования заглавных букв на клавиатуре. Это свойство необязательно, но рекомендуется установить его значение, соответствующее ожидаемому вводу. Например, для поля, ожидающего ввода на естественном языке, можно установить его равным KeyboardCapitalization.Sentences.

label: Строка, определяющая метку для клавиатуры. Это свойство необязательно, но рекомендуется установить его описательным значением, соответствующим ожидаемому вводу.

TextField(  
 modifier = modifier**,** value = text**,** onValueChange = **{** onValueChange(**it**) **},** label = **{** Text(text = label) **},** keyboardOptions = KeyboardOptions.Default.copy(  
 keyboardType = KeyboardType.Number**,** imeAction = ImeAction.Next  
 )**,** singleLine = true**,** leadingIcon = **{** icon?.*let* **{** Icon(imageVector = **it,** contentDescription = null) **} },** placeholder = **{** Text(text = hint) **},**)



## Switch

Switch включает или выключает состояние одного элемента.

Переключатель имеет два состояния, позволяющих пользователю выбирать между двумя вариантами. Переключатель состоит из трека, ползунка и опционального значка, как показано ниже.

A screenshot of a computer

AI-generated content may be incorrect.

checked — отмечен ли этот переключатель;

onCheckedChange — вызывается при щелчке по этому переключателю. Если null, то этот переключатель не будет интерактивным, если только другой элемент не обработает его входные события и не обновит его состояние.

modifier — модификатор, применяемый к этому переключателю;

thumbContent — содержимое, которое будет отображаться внутри ползунка, ожидаемый размер — SwitchDefaults.IconSize;

enabled — управляет включенным состоянием этого переключателя. Если false, этот компонент не будет реагировать на пользовательский ввод и будет отображаться как визуально отключенный и недоступный для служб специальных возможностей.

colors — SwitchColors, которые будут использоваться для определения цветов, используемых для этого переключателя в различных состояниях.

@Composable  
fun SwitchLayout (  
 modifier: Modifier = Modifier**,** @StringRes text: Int**,** isChecked: Boolean**,** onCheckupChanged: (Boolean) -> Unit  
){  
 Row (  
 modifier = modifier  
 .*fillMaxWidth*()  
 .*size*(48.*dp*)  
 .*padding*(horizontal = 16.*dp*)**,** verticalAlignment = Alignment.CenterVertically**,** horizontalArrangement = Arrangement.SpaceBetween  
 )**{** Text(  
 text = stringResource(id = text)**,** style = MaterialTheme.typography.bodyLarge**,** )  
  
 Switch(  
 checked = isChecked**,** onCheckedChange = **{** onCheckupChanged(**it**) **}** )  
 **}**}

# Часть II – Тестирование

Тестирование программного обеспечения — это структурированный метод проверки программного обеспечения для обеспечения его корректной работы. Автоматизированное тестирование — это проверка кода, которая гарантирует корректность работы другого написанного кода.

Тестирование также позволяет постоянно проверять существующий код по мере внесения изменений.

Хотя ручное тестирование практически всегда имеет место, тестирование в Android часто можно автоматизировать.

Автоматизированное тестирование — неотъемлемая часть любой разработки программного обеспечения, и разработка для Android — не исключение.

Почему автоматизированные тесты важны

На первый взгляд может показаться, что тесты в вашем приложении не нужны, но тестирование необходимо для приложений любого размера и сложности.

Чтобы расширять кодовую базу, необходимо тестировать существующую функциональность по мере добавления новых компонентов, что возможно только при наличии готовых тестов. По мере роста приложения ручное тестирование требует гораздо больше усилий, чем автоматизированное.

Автоматизированные тесты — это тесты, выполняемые программным обеспечением, в отличие от ручных тестов, которые выполняются человеком, непосредственно взаимодействующим с устройством. Автоматизированное и ручное тестирование играют решающую роль в обеспечении комфортного использования продукта пользователями. Однако автоматизированные тесты могут быть более точными и оптимизируют производительность команды, поскольку их запуск не требуется, и они выполняются гораздо быстрее ручного тестирования.

## Тип автоматизированных тестов

### Локальные тесты

Локальные тесты — это тип автоматизированных тестов, которые непосредственно тестируют небольшой фрагмент кода, чтобы убедиться в его корректной работе. С помощью локальных тестов можно тестировать функции, классы и свойства. Локальные тесты выполняются на локальной рабочей станции, то есть в среде разработки без необходимости использования устройства или эмулятора. Кроме того, они требуют очень мало ресурсов компьютера, поэтому могут выполняться быстро даже при ограниченных ресурсах. Android Studio готова к автоматическому запуску локальных тестов.

### Инструментальные тесты

В разработке для Android инструментальные тесты — это тесты пользовательского интерфейса (UI). Они позволяют тестировать части приложения, зависящие от API Android, а также API и сервисов платформы.

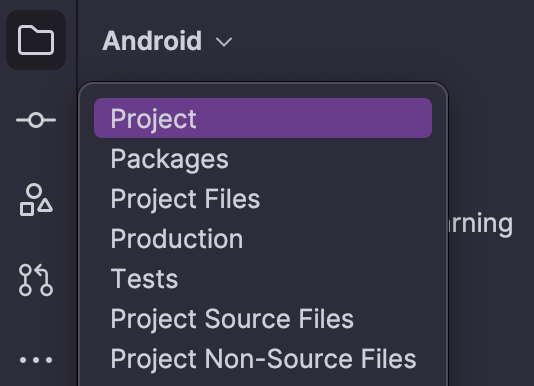
В отличие от локальных тестов, UI-тесты запускают приложение или его часть, имитируют взаимодействие с пользователем и проверяют, отреагировало ли приложение должным образом. На протяжении всего курса UI-тесты запускаются на физическом устройстве или эмуляторе.

При запуске инструментального теста на Android тестовый код фактически встроен в собственный пакет приложения Android (APK), как и в обычном приложении для Android. APK — это сжатый файл, содержащий весь код и необходимые файлы для запуска приложения на устройстве или эмуляторе. Тестовый APK устанавливается на устройство или эмулятор вместе с обычным APK-файлом приложения. Затем тестовый APK-файл запускает свои тесты на APK-файле приложения.

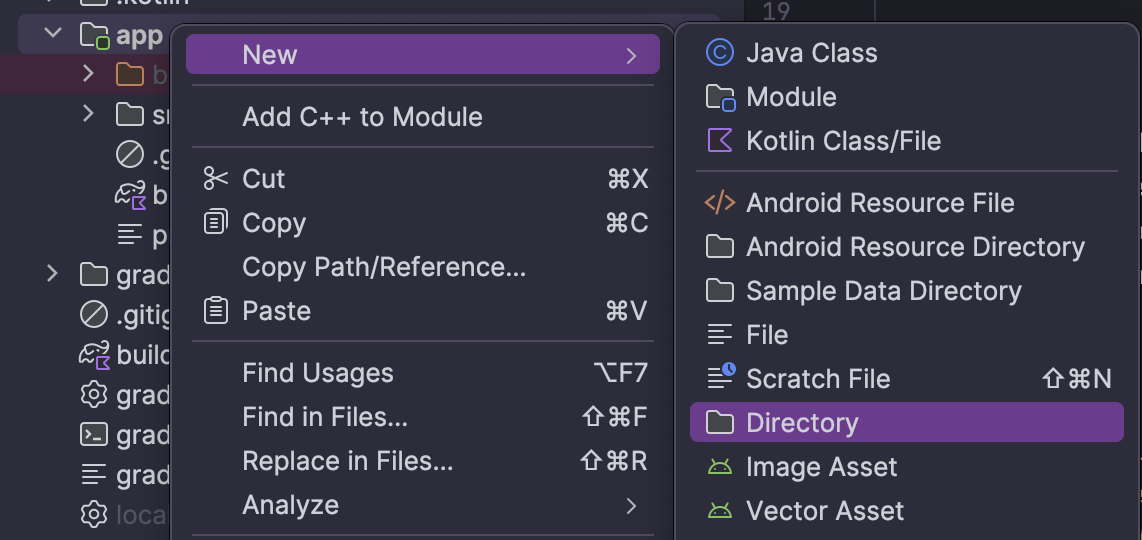
## Написание тестов

### Локальное тестирование

При отсутствии соответствующей директории для тестирования необходимо ее создать. Для этого для начала необходимо сменить отображение структуры проекта с “Android” на “Project”



Далее на уровне app необходимо добавить новую директорию



Далее необходимо создать директорию test/Java

A screenshot of a computer

AI-generated content may be incorrect.

Каталог test должен иметь структуру пакетов, идентичную структуре основного каталога, где находится код вашего приложения. Другими словами, точно так же, как код приложения находится в пакете main > java > com > example > app, ваши локальные тесты будут находиться в test > java > com > example > app.

Создание утверждения обычно является конечной целью автоматизированного теста и нечасто используется в коде приложения. Для этого можно использовать метод assertEquals() из библиотеки JUnit.

Метод assertEquals() принимает два параметра: ожидаемое и фактическое значения. Если эти значения равны, утверждение и тест пройдены. Если они не равны, утверждение и тест не пройдены.

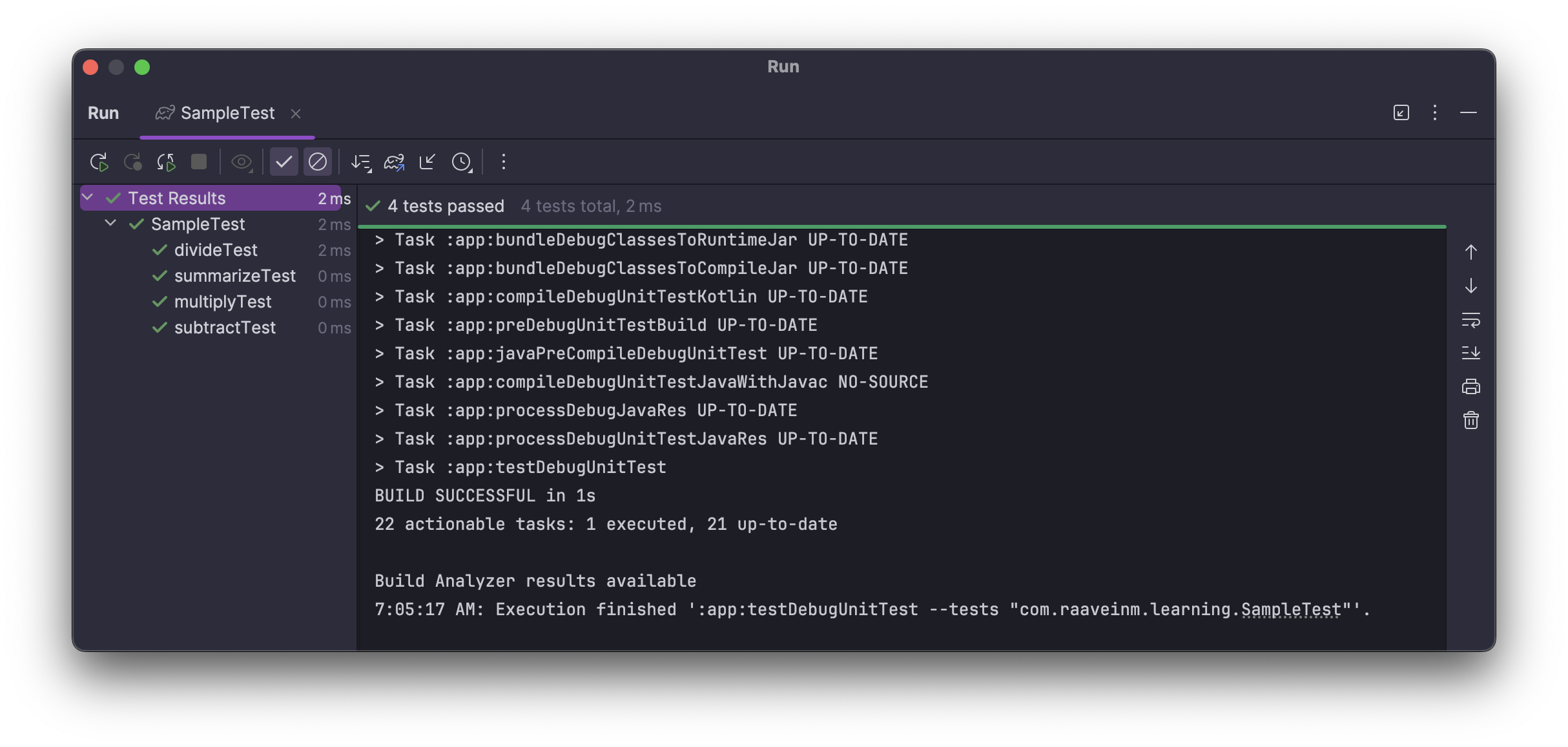
Пример класса для тестирования

package com.raaveinm.learning  
  
class TestSample {  
 private val firstDigit: Int = 24  
  
 fun summarize(secondDigit: Int): Int = firstDigit + secondDigit  
 fun subtract(secondDigit: Int): Int = firstDigit - secondDigit  
 fun multiply(secondDigit: Int): Int = firstDigit \* secondDigit  
 fun divide(secondDigit: Int): Int = firstDigit / secondDigit  
}

И функции для его тестирования

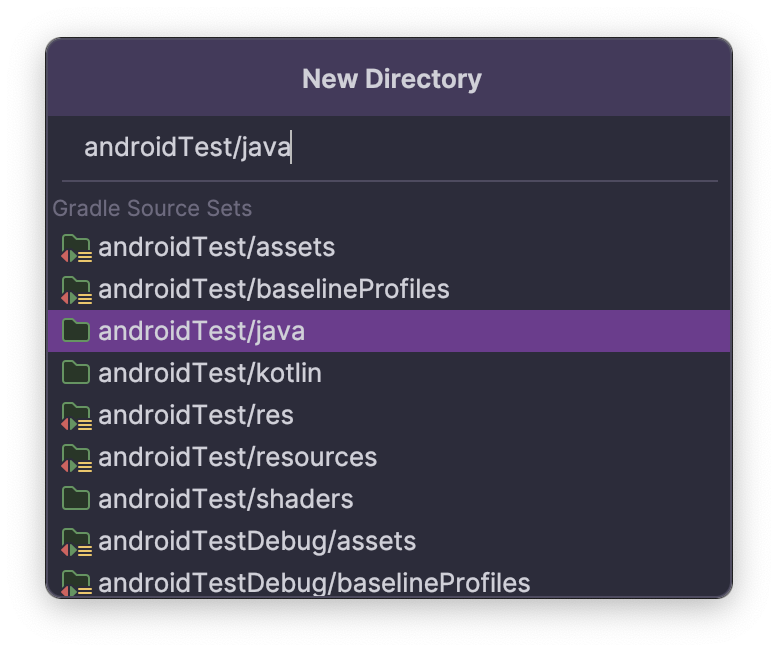
package com.raaveinm.learning  
  
import org.junit.Assert.\*  
import org.junit.Test  
  
class SampleTest {  
 @Test  
 fun summarizeTest() {  
 val testSample = TestSample()  
 val result = testSample.summarize(2)  
 assertEquals(result**,** 26)  
 }  
  
 @Test  
 fun subtractTest() {  
 val testSample = TestSample()  
 val result = testSample.subtract(2)  
 assertEquals(result**,** 22)  
 }  
  
 @Test  
 fun multiplyTest() {  
 val testSample = TestSample()  
 val result = testSample.multiply(2)  
 assertEquals(result**,** 48)  
 }  
  
 @Test  
 fun divideTest() {  
 val testSample = TestSample()  
 val result = testSample.divide(2)  
 assertEquals(result**,** 12)  
 }  
  
}

После запуска тестов необходимо удостоверить, что все функции прошли проверку:



### Инструментальное тестирование

Для написания тестов необходимо создать директорию androidTest/Java



Тесты инструментирования тестируют реальный экземпляр приложения и его пользовательский интерфейс, поэтому содержимое пользовательского интерфейса должно быть задано аналогично тому, как оно задаётся в методе onCreate() файла MainActivity.kt при написании кода для приложения. Это необходимо сделать до написания всех тестов инструментирования для приложений, созданных с помощью Compose.

composeTestRule.setContent **{** LearningTheme **{** SampleScreen()  
 **}  
}**

Для получения информации о состоянии Activity необходимо задать правило:

@get:Rule  
val composeTestRule = *createAndroidComposeRule*<ComponentActivity>()

Пример написания UI теста:

class TipUiTest {  
  
 @get:Rule  
 val composeTestRule = *createAndroidComposeRule*<ComponentActivity>()  
  
 @Test  
 fun calculateTwentyPercent() {  
 composeTestRule.setContent **{** LearningTheme **{** TipCalculatorScreen()  
 **}  
 }** composeTestRule.*onNodeWithText*("Bill Amount").*performTextInput*("100")  
 composeTestRule.*onNodeWithText*("Tip Percentage").*performTextInput*("20")  
 val expectedTip = NumberFormat.getCurrencyInstance().format(20)  
 val expectedText = composeTestRule.activity.getString(R.string.*tip\_amount***,** expectedTip)  
 composeTestRule.*onNodeWithText*(expectedText).assertExists()  
 }  
}

*\* $ git clone https://github.com/google-developer-training/basic-android-kotlin-compose-training-tip-calculator.git  
\* $ cd basic-android-kotlin-compose-training-tip-calculator  
\* $ git checkout test\_solution*

# Часть III – Практическое задание

**Задание 6**

Создайте в ресурсах файл circle.xml c векторным изображением закрашенного круга:

<vector xmlns:android="http://schemas.android.com/apk/res/android"

android:width="48dp"

android:height="48dp"

android:viewportWidth="24"

android:viewportHeight="24">

<path

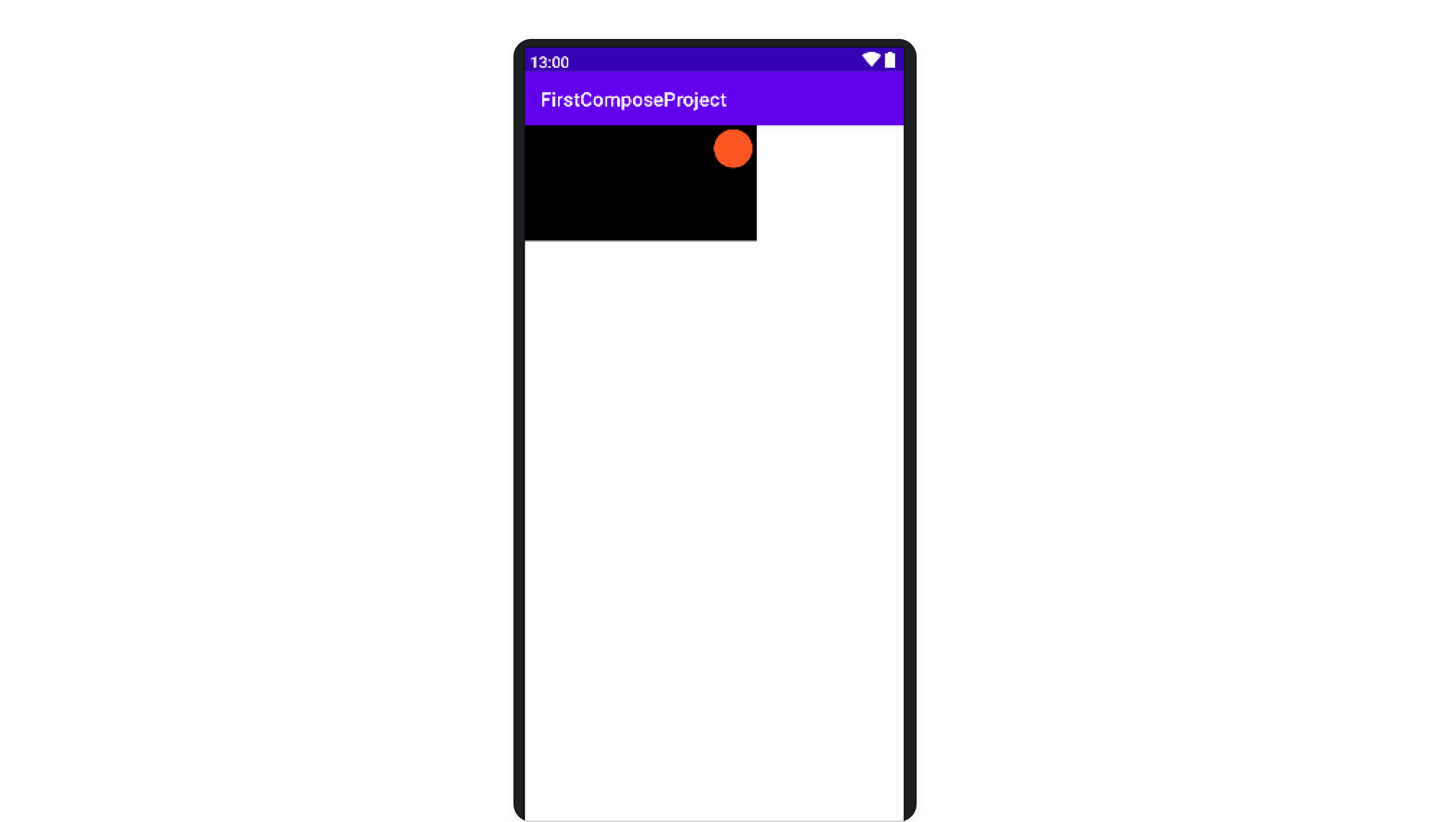
android:fillColor="#FF5722"

android:pathData="M12,2a10,10 0,1,1 0,20a10,10 0,1,1 0,-20"/>

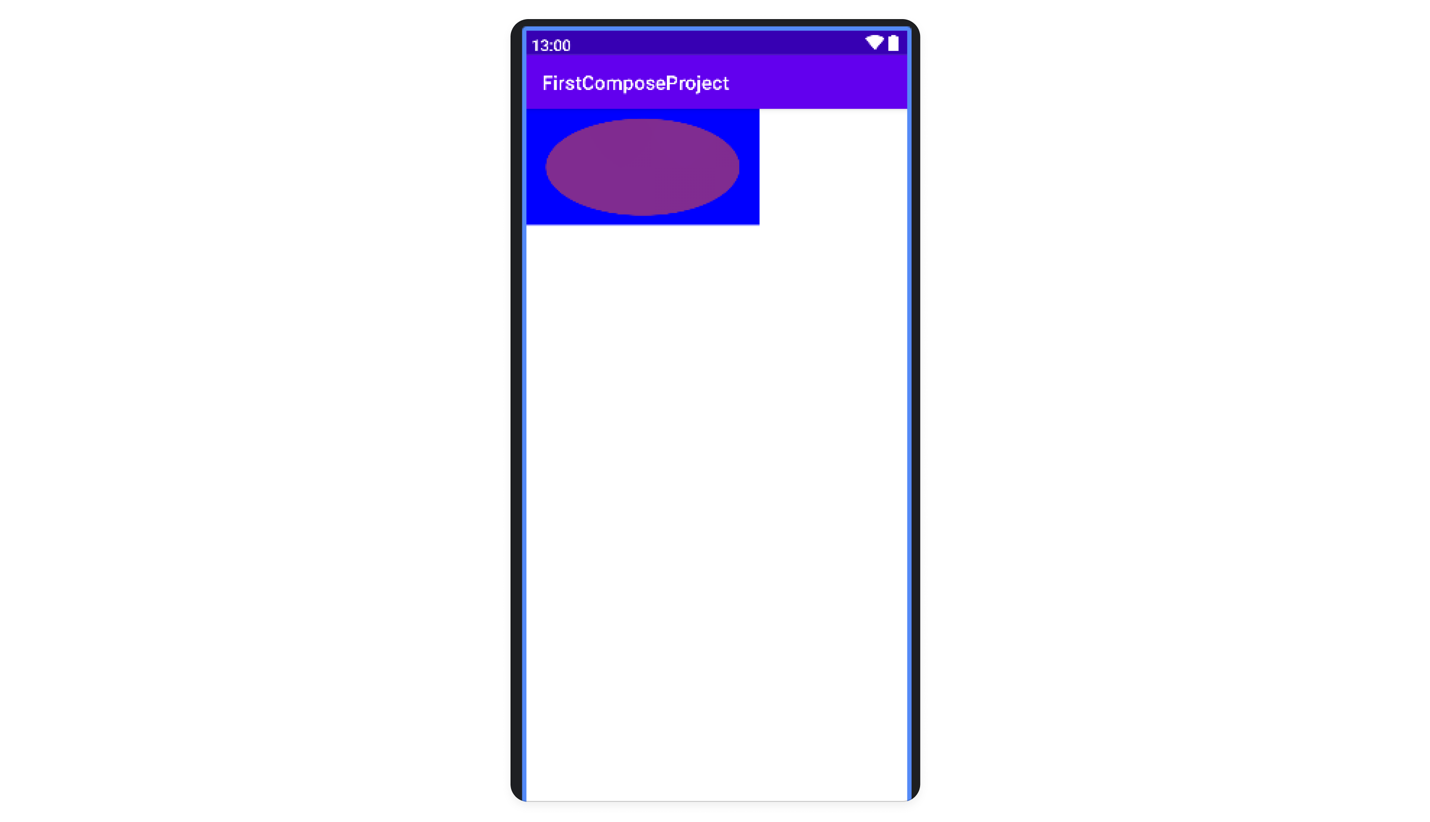
</vector>

Напишите две функции:

1. Расположите круг исходного размера в правом верхнем углу в компоненте c чёрным фоном, размер — 240.dp x 120.dp:



1. Задайте синий цвет фона компонента. Расположите круг по центру и растяните по размерам. Используйте один из параметров компонента, чтобы круг был фиолетового цвета:



**Задание 7**

Реализуйте Composable функции, используя контейнер Column, для изображений, представленных ниже.

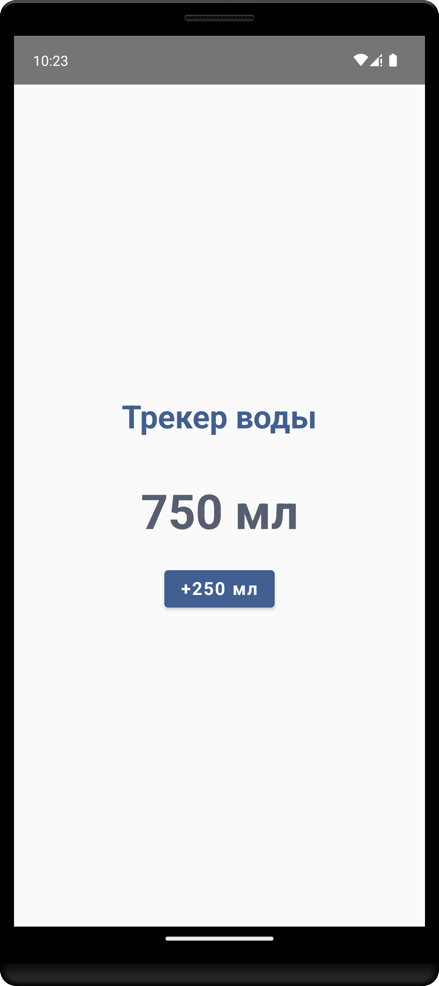
 

А также Composable функцию, применяя контейнер Row (изображение справа можно взять произвольное):



**Задание 8**

Имеется следующее приложение, которое представляет собой счетчик выпитой жидкости, см. скриншот:



А также представлен код Composable функции WaterTracker:

**@Composable**

**fun WaterTracker() {**

**var waterCount by { mutableStateOf(100) }**

**Column(**

**modifier = Modifier**

**.fillMaxSize()**

**.padding(16.dp),**

**) {**

**Text(**

**text = "Трекер воды",**

**fontSize = 32.sp,**

**fontWeight = FontWeight.Bold,**

**color = primaryColor**

**)**

**Spacer(modifier = Modifier.height(48.dp))**

**Text(**

**fontSize = 48.sp,**

**fontWeight = FontWeight.Bold,**

**color = secondaryColor**

**)**

**Spacer(modifier = Modifier.height())**

**Button(**

**colors = ButtonDefaults(backgroundColor = primaryColor),**

**onClick = { waterCount = 250 }**

**) {**

**Text(**

**text = "+250 мл",**

**fontSize = 18.sp,**

**fontWeight = FontWeight.Bold,**

**color = buttonTextColor**

**)**

**}**

**}**

**}**

Необходимо исправить код в Composable функции WaterTracker, чтобы при нажатии на кнопку происходило увеличение количества выпитой жидкости на 250 мл.   
 А также мы хотим дать пользователям возможность сбрасывать счётчик стаканов по завершении дня, а также выводить количество дней подряд, когда пользователь выпил 1500 мл и больше. Если в один из дней пользователь выпивает меньшее количество воды, счётчик должен обнулиться.

Для этого нужно:

* Добавить новый стейт с сохранением количества дней, в течение которых пользователь пил 1500 мл воды и больше.
* Добавить новую кнопку с текстом «Завершить день».
* При нажатии на кнопку сбрасывать счётчик стаканов воды до нуля, а при условии достижения 1500 повышать счётчик дней. В противном случае счётчик дней сбрасывается до нуля.
* Добавить вывод текста со значением количества дней, в рамках которых было выпито 1500 мл и больше, подряд.

**Задание 9**

Создать UI тесты для приложения, которое было создано в задании 8, а именно:  
 1) Проверить наличие и корректную работу кнопки добавления жидкости;

2) Проверить наличие и корректную работу кнопки “Завершить день”;

3) Проверить, что выводится необходимый текст, со значением количества дней, в рамках которых было выпито 1500 мл и больше, подряд.