[**Введение в состояние в Compose**](https://developer.android.com/codelabs/basic-android-kotlin-compose-using-state?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-2-pathway-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-using-state)**. Приложение-калькулятор чаевых**

***Выполните лабораторную работу. Составьте краткий конспект по состояниям в Compose***

Эта лаборатория знакомит вас с состоянием, а также с тем, как его можно использовать и управлять им с помощью Jetpack Compose.

По своей сути **состояние в приложении** — это любое значение, которое может меняться со временем. Это определение очень широкое и включает в себя все: от базы данных до переменной в вашем приложении.

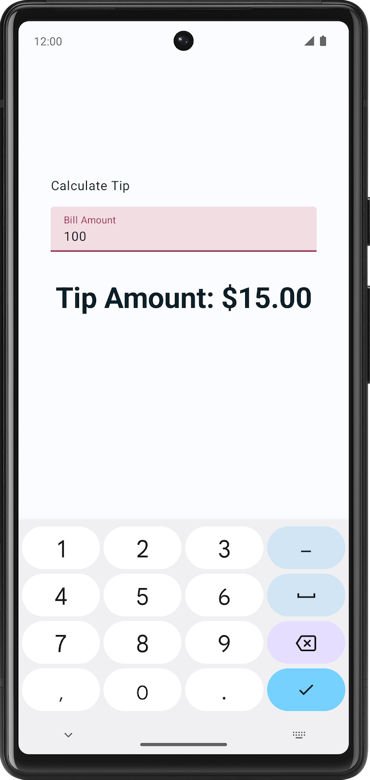
Все приложения Android отображают состояние пользователю. Вот несколько примеров состояния в приложениях Android:

* Сообщение, которое показывает, что сетевое соединение не может быть установлено.
* Формы, такие как регистрационные формы. Состояние можно заполнить и отправить.
* Нажимаемые элементы управления, например кнопки. Состояние может быть *не касанием*, *касанием* (анимация отображения) или *касанием* ( onClick действие).

В этой лаборатории кода вы узнаете, как использовать состояние и думать о нем при использовании Compose. Для этого вы создаете **приложение-калькулятор чаевых** под названием Tip Time со следующими встроенными элементами пользовательского интерфейса Compose:

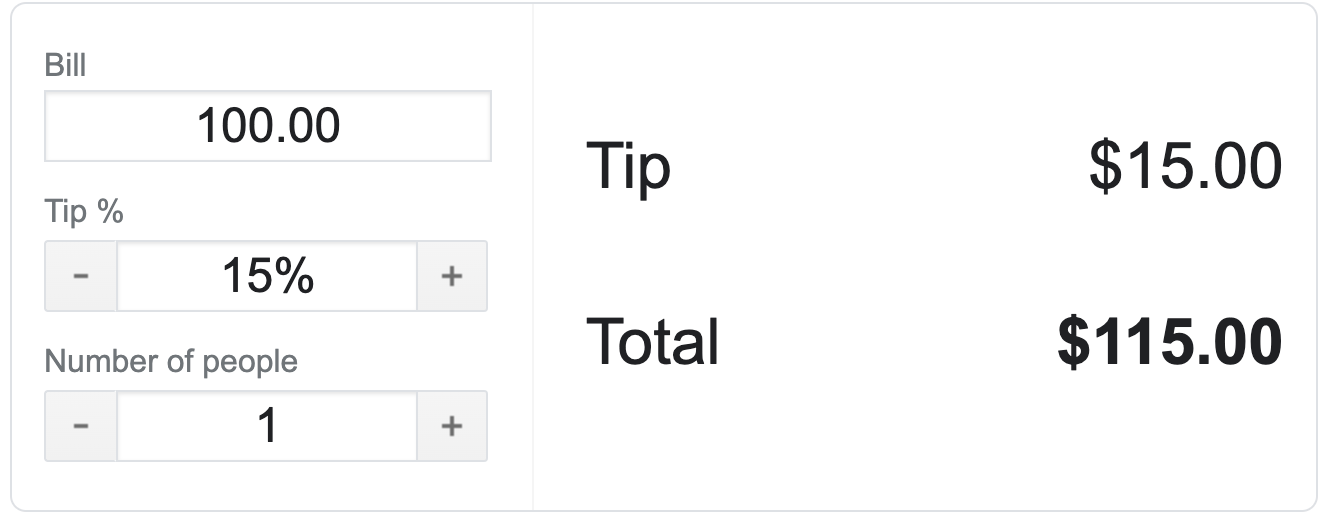
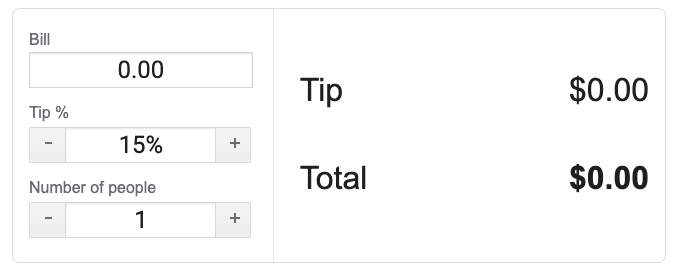
* Составной элемент TextFieldдля ввода и редактирования текста.
* Составной элемент Textдля отображения текста.
* Составной элемент Spacerдля отображения пустого пространства между элементами пользовательского интерфейса.

В конце этой лабораторной работы вы создадите интерактивный калькулятор чаевых, который автоматически рассчитывает сумму чаевых при вводе суммы услуги. На этом изображении показано, как выглядит окончательное приложение:

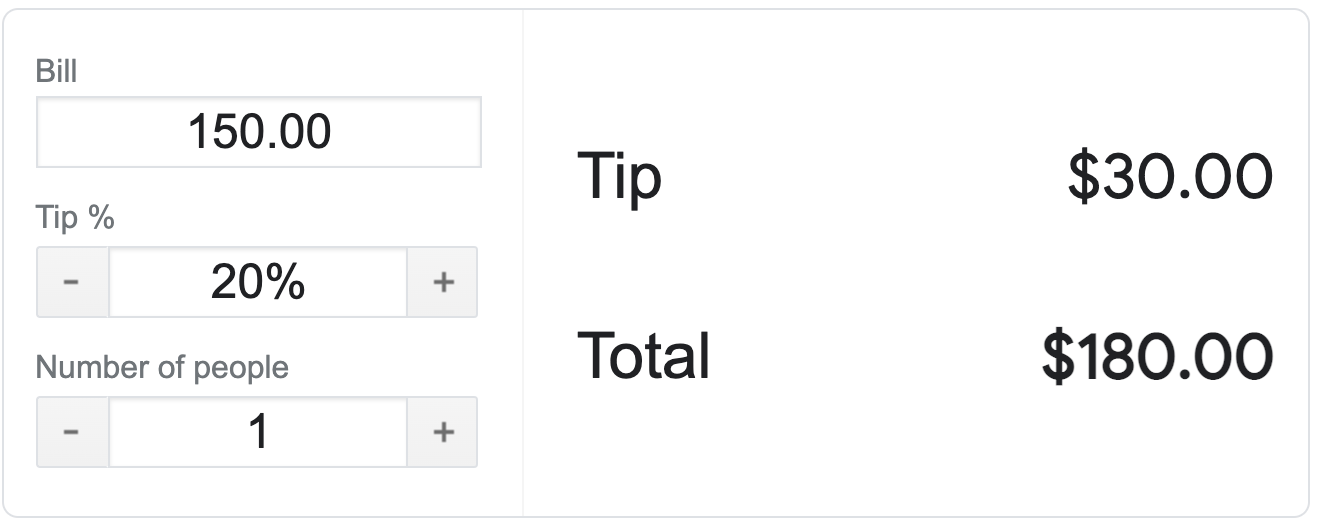


## [**2. Начните работу**](https://developer.android.com/codelabs/basic-android-kotlin-compose-using-state?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-2-pathway-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-using-state#1)

1. Проверьте [онлайн-калькулятор чаевых Google](https://www.google.com/search?q=tip+calculator) . Обратите внимание, что это всего лишь пример, а не приложение для Android, которое вы будете создавать в этом курсе.

1. Введите разные значения в поля **«Bill»** и **«Tip»** . Чаевые и общие значения изменяются.

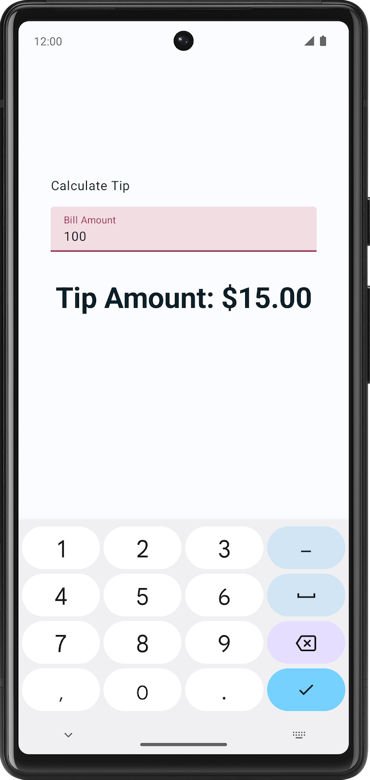
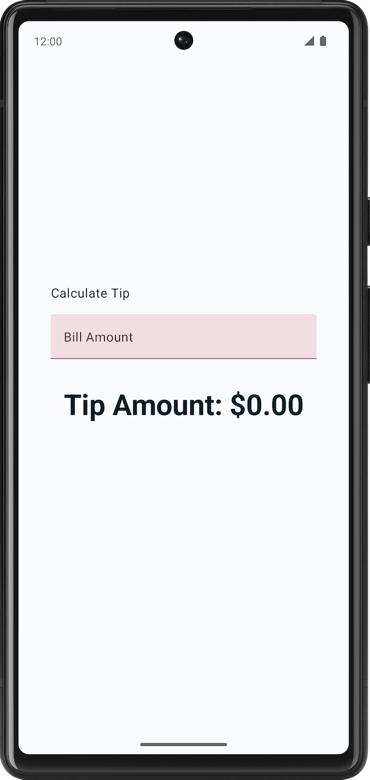


Обратите внимание, что в тот момент, когда вы вводите значения, **Tip** и **Total** обновляются. К концу следующей лабораторной работы вы разработаете аналогичное приложение для расчета чаевых на Android.

На этом этапе вы создадите простое приложение для Android с калькулятором чаевых.

Разработчики часто работают таким образом: получают готовую и работающую простую версию приложения (даже если она выглядит не очень хорошо), а затем добавляют дополнительные функции и делают ее более визуально привлекательной позже.

К концу этой лабораторной работы ваше приложение для расчета чаевых будет выглядеть так, как показано на этих скриншотах. Когда пользователь вводит **сумму счета** , ваше приложение отображает рекомендуемую сумму чаевых. На данный момент процент чаевых жестко запрограммирован на уровне **15%**. В следующей лаборатории вы продолжите работу над своим приложением и добавите дополнительные функции, такие как установка индивидуального процента чаевых.



## [**3. Получите стартовый код**](https://developer.android.com/codelabs/basic-android-kotlin-compose-using-state?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-2-pathway-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-using-state#2)

Стартовый код — это заранее написанный код, который можно использовать в качестве отправной точки для нового проекта. Это также может помочь вам сосредоточиться на новых концепциях, изучаемых в этой лаборатории.

Начните работу со стартовым кодом, загрузив его здесь:

[file\_downloadСкачать zip](https://github.com/google-developer-training/basic-android-kotlin-compose-training-tip-calculator/archive/refs/heads/starter.zip)

Альтернативно вы можете клонировать репозиторий GitHub для кода:

$ 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 starter

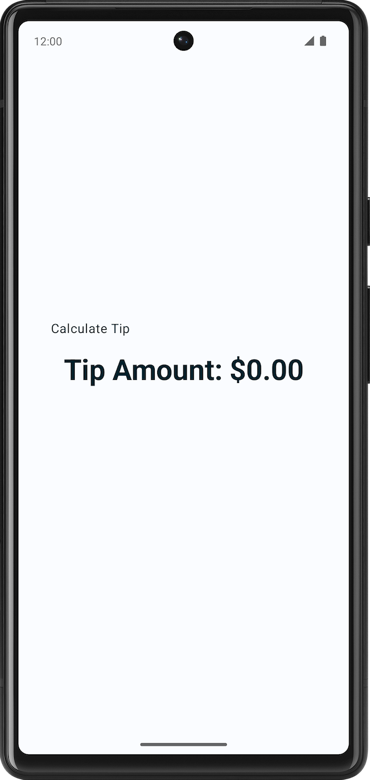
**Примечание.** Стартовый код находится в  ветке starter скачанного репозитория.

Вы можете просмотреть стартовый код в [TipTime](https://github.com/google-developer-training/basic-android-kotlin-compose-training-tip-calculator/tree/starter)репозитории GitHub.

## **Обзор стартового приложения**

Чтобы ознакомиться со стартовым кодом, выполните следующие действия:

1. Откройте проект со стартовым кодом в Android Studio.
2. Запустите приложение на устройстве Android или в эмуляторе.
3. Вы заметите два текстовых компонента; один предназначен для метки, а другой — для отображения суммы чаевых.



## **Прохождение стартового кода**

Стартовый код содержит компонуемые тексты. На этом пути вы добавите текстовое поле для ввода данных пользователем. Вот краткое описание некоторых файлов, которые помогут вам начать.

**res > значения > strings.xml**

<resources>  
   <string name="app\_name">Tip Time</string>  
   <string name="calculate\_tip">Calculate Tip</string>  
   <string name="bill\_amount">Bill Amount</string>  
   <string name="tip\_amount">Tip Amount: %s</string>  
</resources>

Это string.xml файл в ресурсах со всеми строками, которые вы будете использовать в этом приложении.

**MainActivity**

Этот файл содержит в основном код, сгенерированный шаблоном, и следующие функции.

* Функция TipTimeLayout() содержит элемент Column с двумя составными текстовыми объектами, которые вы видите на скриншотах. Он также имеет spacer, чтобы добавить пространство по эстетическим соображениям.
* Функция calculateTip(), которая принимает сумму счета и рассчитывает сумму чаевых в размере 15%. Параметру tipPercent присвоено значение аргумента 15.0 по умолчанию. На данный момент значение чаевых по умолчанию устанавливается равным 15%. В следующей кодовой лаборатории вы получаете сумму чаевых от пользователя.

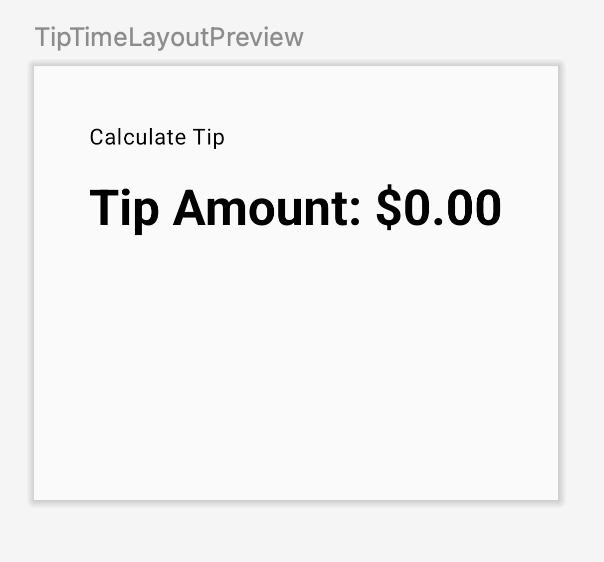
@Composable  
fun TipTimeLayout() {  
    Column(  
        modifier = Modifier  
            .statusBarsPadding()  
            .padding(horizontal = 40.dp)  
            .verticalScroll(rememberScrollState())  
            .safeDrawingPadding(),  
        horizontalAlignment = Alignment.CenterHorizontally,  
        verticalArrangement = Arrangement.Center  
    ) {  
        Text(  
            text = stringResource(R.string.calculate\_tip),  
            modifier = Modifier  
                .padding(bottom = 16.dp, top = 40.dp)  
                .align(alignment = Alignment.Start)  
        )  
        Text(  
            text = stringResource(R.string.tip\_amount, "$0.00"),  
            style = MaterialTheme.typography.displaySmall  
        )  
        Spacer(modifier = Modifier.height(150.dp))  
    }  
}

private fun calculateTip(amount: Double, tipPercent: Double = 15.0): String {  
   val tip = tipPercent / 100 \* amount  
   return NumberFormat.getCurrencyInstance().format(tip)  
}

override fun onCreate(savedInstanceState: Bundle?) {  
   //...  
   setContent {  
       TipTimeTheme {  
           Surface(  
           //...  
           ) {  
               TipTimeLayout()  
           }  
       }  
   }  
}

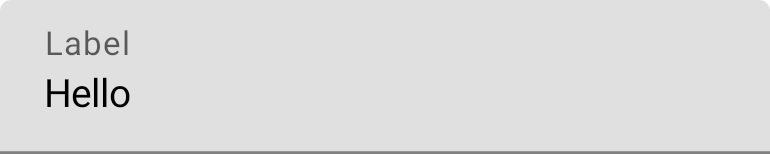
Блок функции TipTimeLayoutPreview(), функция  TipTimeLayout()вызывается. При этом макет приложения отобразится на панели **«Design»** и на панели **«Split»** .

@Preview(showBackground = true)  
@Composable  
fun TipTimeLayoutPreview() {  
   TipTimeTheme {  
       TipTimeLayout()  
   }  
}



## **Получите информацию от пользователя**

В этом разделе вы добавляете элемент пользовательского интерфейса, который позволяет пользователю вводить сумму счета в приложении. Вы можете увидеть, как это выглядит на этом изображении:



Ваше приложение использует собственный стиль и тему.

Стили и темы — это набор атрибутов, определяющий внешний вид одного элемента пользовательского интерфейса. В стиле можно указать такие атрибуты, как цвет шрифта, размер шрифта, цвет фона и многое другое, которые можно применить ко всему приложению. В последующих лабораторных работах будет рассказано, как реализовать их в вашем приложении. На данный момент это уже сделано для того, чтобы ваше приложение стало красивее.

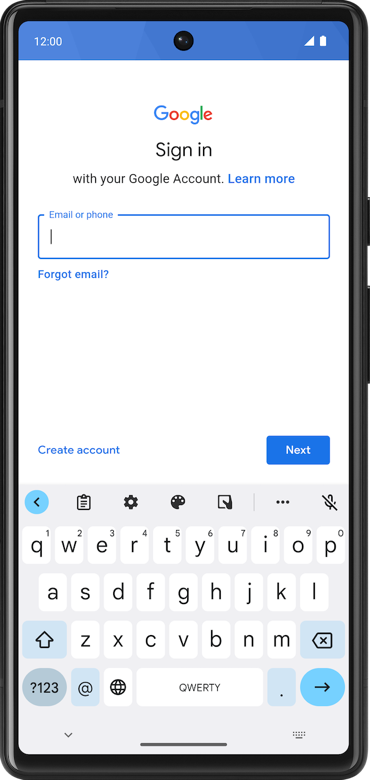
Чтобы лучше понять, ниже представлено параллельное сравнение версии решения приложения с настраиваемой темой и без нее.

|  |  |
| --- | --- |
| Без собственной темы. | С индивидуальной темой. |

**Примечание.** Когда вы добавляете компонуемое текстовое поле в свое приложение, оно будет иметь цветовую схему как часть пользовательской темы, примерно такую:



Компонуемая функция TextField позволяет пользователю вводить текст в приложение. Например, обратите внимание на текстовое поле на экране входа в приложение Gmail на этом изображении:



Добавьте компоновку TextField в приложение:

1. В файл MainActivity.kt добавьте  составную функцию EditNumberField(), принимающую параметр Modifier.
2. В теле приведенной  ниже функции EditNumberField()  добавьте объект  TextField, который принимает именованный параметр value, установленный в пустую строку, и именованный параметр onValueChange, установленный в пустое лямбда-выражение:

@Composable  
fun EditNumberField(modifier: Modifier = Modifier) {  
**TextField(  
      value = "",  
      onValueChange = {},  
      modifier = modifier  
   )**}

1. Обратите внимание на переданные вами параметры:

* Параметр value представляет собой текстовое поле, в котором отображается строковое значение, которое вы здесь передаете.
* Параметр onValueChange— это обратный вызов лямбда-выражения, который срабатывает, когда пользователь вводит текст в текстовое поле.

1. Импортируйте эту функцию:

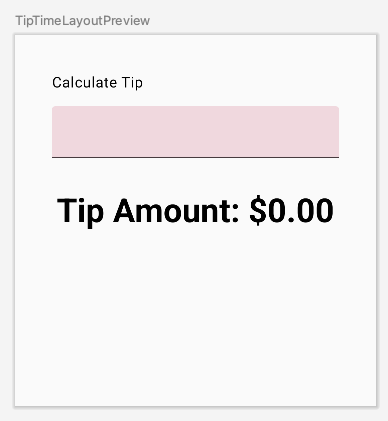
import androidx.compose.material3.TextField

1. В составном элементе TipTimeLayout() в строке после первой текстовой составной функции вызовите функцию EditNumberField(), передав следующий модификатор.

import androidx.compose.foundation.layout.fillMaxWidth  
  
@Composable  
fun TipTimeLayout() {  
   Column(  
        modifier = Modifier  
            .statusBarsPadding()  
            .padding(horizontal = 40.dp)  
            .verticalScroll(rememberScrollState())  
            .safeDrawingPadding(),  
        horizontalAlignment = Alignment.CenterHorizontally,  
        verticalArrangement = Arrangement.Center  
   ) {  
       Text(  
           ...  
       )  
       EditNumberField(modifier = Modifier.padding(bottom = 32.dp).fillMaxWidth())  
       Text(  
           ...  
       )  
       ...  
   }  
}

На экране появится текстовое поле.

1. На панели **«Design»** вы должны увидеть текст Calculate Tip, пустое текстовое поле и  компонуемый текст Tip Amount.



## [**4. Используйте состояние в Compose**](https://developer.android.com/codelabs/basic-android-kotlin-compose-using-state?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-2-pathway-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-using-state#3)

Состояние в приложении — это любое значение, которое может меняться со временем. В этом приложении состояние — это сумма счета.

Добавьте переменную для хранения состояния:

1. В начале функции EditNumberField() используйте  ключевое слово val, чтобы добавить переменную amountInput, присвоив ей значение "0":

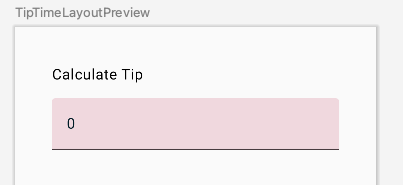
val amountInput = "0"

Это состояние приложения для суммы счета.

1. Установите  для именованного параметра value значение amountInput:

TextField(  
**value = amountInput**,  
   onValueChange = {},  
)

1. Проверьте предварительный просмотр. В текстовом поле отображается значение, установленное для переменной состояния, как вы можете видеть на этом изображении:



1. Запустите приложение в эмуляторе, попробуйте ввести другое значение. Жестко закодированное состояние остается неизменным, поскольку компонуемый элемент TextField не обновляется сам. Он обновляется при  изменении его параметра value, установленного в  свойстве amountInput.

Переменная amountInput представляет состояние текстового поля. Наличие жестко закодированного состояния бесполезно, поскольку его нельзя изменить и оно не отражает вводимые пользователем данные. Вам необходимо обновить состояние приложения, когда пользователь обновляет сумму счета.

## [**5. Композиция**](https://developer.android.com/codelabs/basic-android-kotlin-compose-using-state?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-2-pathway-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-using-state#4)

Составные элементы в вашем приложении описывают пользовательский интерфейс, в котором отображается столбец с текстом, разделитель и текстовое поле. В тексте отображается текст Calculate Tip, а в текстовом поле отображается  значение 0 или любое другое значение по умолчанию.

Compose — это декларативный фреймворк пользовательского интерфейса. Это означает, что вы указываете, как пользовательский интерфейс должен выглядеть в вашем коде. Если вы хотите, чтобы в вашем текстовом поле изначально отображалось значение 100, вы должны установить начальное значение в коде для составных элементов равным значению 100.

Что произойдет, если вы хотите, чтобы ваш пользовательский интерфейс менялся во время работы приложения или при взаимодействии пользователя с ним? Например, что, если вы хотите обновить переменную amountInput значением, введенным пользователем, и отобразить его в текстовом поле? Именно тогда вы полагаетесь на процесс, называемый **рекомпозицией**, для обновления состава приложения.

Композиция — это описание пользовательского интерфейса, созданного Compose при выполнении компонуемых объектов. Приложения Compose вызывают составные функции для преобразования данных в пользовательский интерфейс. Если происходит изменение состояния, Compose повторно выполняет затронутые компонуемые функции с новым состоянием, что создает обновленный пользовательский интерфейс — это называется рекомпозицией. Составьте график рекомпозиции для вас.

Когда Compose запускает ваши составные объекты в первый раз во время первоначальной композиции, он отслеживает составные объекты, которые вы вызываете для описания вашего пользовательского интерфейса в композиции. **Рекомпозиция** — это когда Compose повторно выполняет композицию, которая могла измениться в ответ на изменения данных, а затем обновляет композицию, чтобы отразить любые изменения.

Композиция может быть создана только с помощью исходной композиции и обновлена путем рекомпозиции. Единственный способ изменить композицию — это рекомпозиция.  Для этого Compose необходимо знать, какое состояние отслеживать , чтобы можно было запланировать рекомпозицию при получении обновления. В вашем случае это переменная amountInput, поэтому при изменении ее значения Compose планирует перекомпоновку.

Вы используете [State](https://developer.android.com/reference/kotlin/androidx/compose/runtime/State) и [MutableState](https://developer.android.com/reference/kotlin/androidx/compose/runtime/MutableState) типы в Compose, чтобы сделать состояние вашего приложения наблюдаемым или отслеживаемым с помощью Compose. Тип State является неизменяемым, поэтому вы можете только прочитать его значение, а тип [MutableState](https://developer.android.com/reference/kotlin/androidx/compose/runtime/MutableState) является изменяемым. Вы можете использовать эту [mutableStateOf()](https://developer.android.com/reference/kotlin/androidx/compose/runtime/package-summary#mutableStateOf(kotlin.Any,androidx.compose.runtime.SnapshotMutationPolicy)) функцию для создания наблюдаемого объекта MutableState. Он получает начальное значение в качестве параметра, который обертывается в State объект, что затем делает его value наблюдаемым.

Значение, возвращаемое функцией mutableStateOf():

* Удерживает состояние, которое соответствует сумме счета.
* Изменяемый, поэтому значение можно изменить.
* Является наблюдаемым, поэтому Compose отслеживает любые изменения значения и запускает рекомпозицию для обновления пользовательского интерфейса.

Добавьте состояние стоимости обслуживания:

1. В  функции EditNumberField() измените ключевое слово val перед  переменной состояния amountInput на var ключевое слово:

**var** amountInput = "0"

Это делает amountInput изменчивым.

1. Используйте MutableState<String> тип вместо жестко запрограммированной String переменной, чтобы Compose мог отслеживать состояние amountInput, а затем передавать строку "0", которая является начальным значением по умолчанию для переменной состояния amountInput:

import androidx.compose.runtime.MutableState  
import androidx.compose.runtime.mutableStateOf  
  
var amountInput: MutableState<String> = mutableStateOf("0")

Инициализацию amountInput также можно записать следующим образом с выводом типа:

var amountInput = mutableStateOf("0")

Функция  mutableStateOf() получает начальное "0" значение в качестве аргумента, что затем делает amountInput наблюдаемым. Это приводит к появлению предупреждения о компиляции в Android Studio, но вы скоро это исправите:

*Creating a state object during composition without using remember.*

1. В  составной функции TextField используйте свойство amountInput.value:

TextField(  
   value = **amountInput.value**,  
   onValueChange = {},  
   modifier = modifier  
)

Compose отслеживает каждый компонуемый объект, который считывает свойства состояния value и запускает рекомпозицию при его изменении.

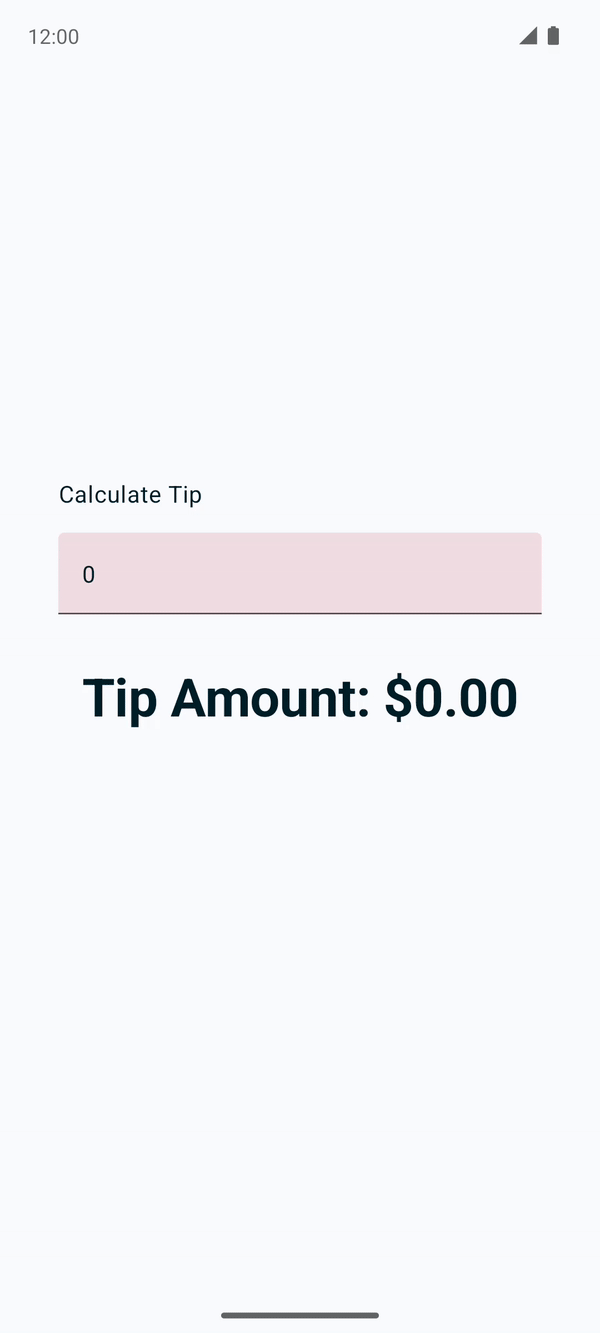
Обратный вызов onValueChange запускается при изменении ввода в текстовом поле. В лямбда-выражении переменная содержит новое значение it.

1. В  лямбда-выражении onValueChange задайте для свойства именованного параметра amountInput.value переменную it:

@Composable  
fun EditNumberField(modifier: Modifier = Modifier) {  
   var amountInput = mutableStateOf("0")  
   TextField(  
       value = amountInput.value,  
       onValueChange = { **amountInput.value = it** },  
       modifier = modifier  
   )  
}

Вы обновляете состояние TextField (то есть переменной amountInput), когда TextField уведомляет вас об изменении текста с помощью функции onValueChange обратного вызова.

1. Запустите приложение и введите текст в текстовое поле. В текстовом поле по-прежнему отображается значение 0, как вы можете видеть на этом изображении:



Когда пользователь вводит текст в текстовое поле,  вызывается обратный вызов onValueChange, и  переменная amountInput обновляется новым значением. Состояние amountInput отслеживается Compose, поэтому в момент изменения его значения планируется перекомпоновка и компонуемая функция EditNumberField() выполняется снова. В этой составной функции переменная amountInput сбрасывается до исходного значения 0. Таким образом, в текстовом поле отображается  значение 0.

Благодаря добавленному вами коду изменения состояния приводят к планированию рекомпозиций.

Однако вам нужен способ сохранить значение переменной amountInput при перекомпозиции, чтобы она не сбрасывалась до  значения 0 каждый раз, когда функция EditNumberField()перекомпоновывает. Эту проблему вы решите в следующем разделе.

## [**6. Используйте функцию запоминания для сохранения состояния.**](https://developer.android.com/codelabs/basic-android-kotlin-compose-using-state?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-2-pathway-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-using-state#5)

Составные методы можно вызывать много раз из-за рекомпозиции. Составной объект сбрасывает свое состояние во время рекомпозиции, если он не сохранен.

Компонуемые функции могут хранить объект при рекомпозиции с помощью **метода**[**remember**](https://developer.android.com/reference/kotlin/androidx/compose/runtime/package-summary#remember(kotlin.Function0)). Значение, вычисленное функцией, remember сохраняется в композиции во время первоначальной композиции, и сохраненное значение возвращается во время повторной композиции. Обычно функции remember и mutableStateOf используются вместе в составных функциях, чтобы состояние и его обновления правильно отражались в пользовательском интерфейсе.

Используйте функцию remember в функции EditNumberField():

1. В функции EditNumberField() инициализируйте переменную amountInput с помощью by remember делегата свойства Kotlin, окружив вызов функции remember .[mutableStateOf](https://developer.android.com/reference/kotlin/androidx/compose/runtime/package-summary#mutableStateOf(kotlin.Any,androidx.compose.runtime.SnapshotMutationPolicy))()
2. В функцию передайте пустую строку вместо статической строки: [mutableStateOf](https://developer.android.com/reference/kotlin/androidx/compose/runtime/package-summary#mutableStateOf(kotlin.Any,androidx.compose.runtime.SnapshotMutationPolicy))()"0"

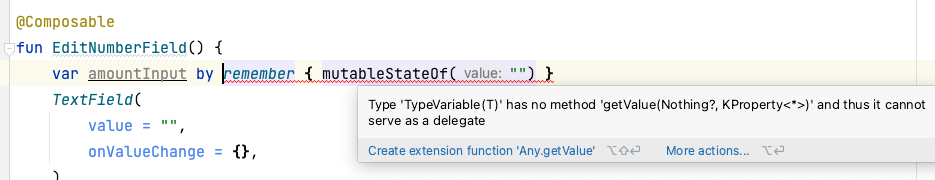
var amountInput by remember { mutableStateOf("") }

Теперь пустая строка является начальным значением по умолчанию для amountInput переменной. by— это [делегирование свойств Kotlin](https://kotlinlang.org/docs/delegated-properties.html) . Функции получения и установки по умолчанию для свойства amountInput делегируются remember функциям получения и установки класса соответственно.

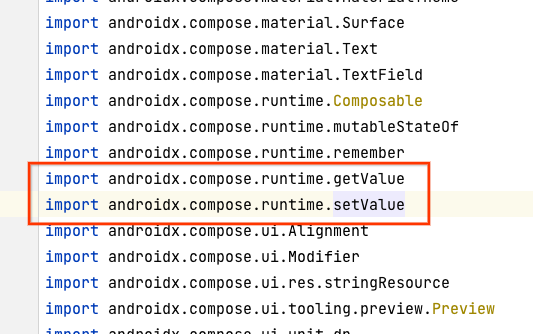
1. Импортируйте эти функции:

import androidx.compose.runtime.remember  
import androidx.compose.runtime.getValue  
import androidx.compose.runtime.setValue

**Предупреждение.** Этот код может вызвать ошибку, которую вы видите на этом изображении относительно getValue()функции расширения:



Если да, вручную добавьте getValue и setValue imports в блок импорта в начале файла, как вы можете видеть на этом изображении:



Добавление импорта методов получения и установки делегата позволяет читать и устанавливать amountInput без обращения к MutableState свойству value.

Обновленная функция EditNumberField() должна выглядеть так:

@Composable  
fun EditNumberField(modifier: Modifier = Modifier) {  
**var amountInput by remember { mutableStateOf("") }**   TextField(  
       value = amountInput,  
       onValueChange = { amountInput = it },  
       modifier = modifier  
   )  
}

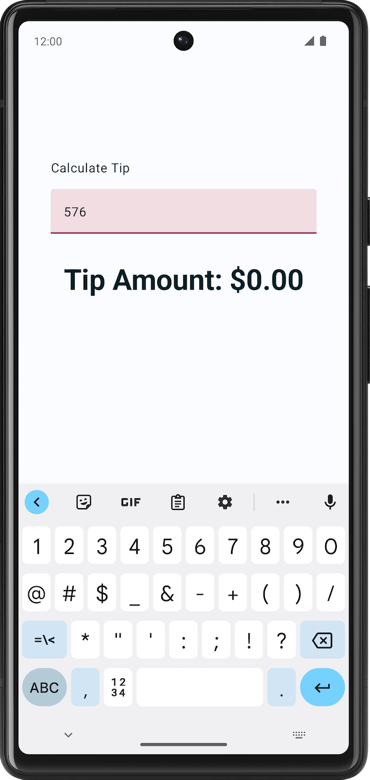
**Примечание.** Во время первоначальной композиции value в поле TextField устанавливается начальное значение, которое представляет собой пустую строку.

Когда пользователь вводит текст в текстовое поле, onValueChange вызывается обратный вызов лямбды, лямбда-выражение выполняется, и amountInput.value в качестве значения устанавливается обновленное значение, введенное в текстовое поле.

Это  изменяемое состояние amountInput, отслеживаемое Compose, запланирована рекомпозиция. Составная EditNumberField()функция перекомпоновывается. Поскольку вы используете remember { },изменение, оно сохраняется при рекомпозиции, и поэтому состояние не инициализируется повторно до "".

Для value текстового поля установлено значение amountInput. Текстовое поле перерисовывается (перерисовывается на экране с новым значением).

1. Запустите приложение и введите текст в текстовое поле. Вы должны увидеть текст, который вы набрали сейчас.



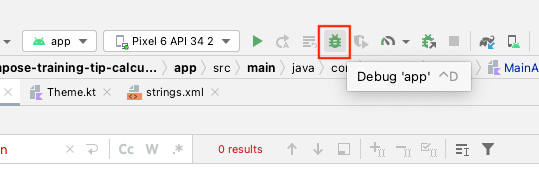
## [**7. Состояние и рекомпозиция в действии**](https://developer.android.com/codelabs/basic-android-kotlin-compose-using-state?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-2-pathway-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-using-state#6)

В этом разделе вы установите точку останова и отладите EditNumberField()составную функцию, чтобы увидеть, как работают начальная композиция и рекомпозиция.

**Примечание** . Точка останова — это место в вашем коде, где вы хотите приостановить нормальное выполнение вашего приложения, чтобы выполнить другие действия, такие как проверка переменных, оценка выражений или выполнение кода построчно. Вы можете установить точку останова на любой исполняемой строке кода.

Установите точку останова и отладьте приложение на эмуляторе или устройстве:

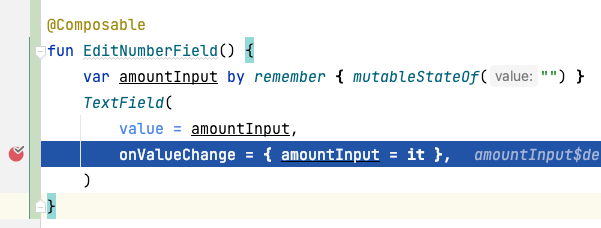
1. В EditNumberField() функции рядом с именованным параметром onValueChange установите точку разрыва строки.
2. В меню навигации нажмите **«Debug app»** . Приложение запускается на эмуляторе или устройстве. Выполнение вашего приложения приостанавливается в первый раз при создании элемента TextField.



1. На панели **«Debug»** нажмите  **«Resume app»** . Текстовое поле создано.
2. На эмуляторе или устройстве введите букву в текстовое поле. Выполнение вашего приложения снова приостанавливается, когда оно достигает установленной вами точки останова.

При вводе текста  вызывается обратный вызов onValueChange. Внутри лямбды  содержится новое значение it, которое вы ввели на клавиатуре.

Как только значение «it» присвоено amountInput,  Compose запускает рекомпозицию с новыми данными, поскольку наблюдаемое значение изменилось.



1. На панели **«Debug»** нажмите  **«Resume app»** . Текст, введенный в эмуляторе или на устройстве, отображается рядом со строкой с точкой останова, как показано на этом изображении:



Это состояние текстового поля.

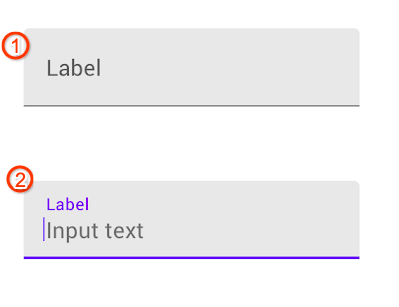
1. Нажмите  **«Resume app»**. Введенное значение отображается на эмуляторе или устройстве.

## [**8. Измените внешний вид**](https://developer.android.com/codelabs/basic-android-kotlin-compose-using-state?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-2-pathway-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-using-state#7)

В предыдущем разделе вы задействовали текстовое поле. В этом разделе вы улучшите пользовательский интерфейс.

## **Добавьте метку в текстовое поле**

Каждое текстовое поле должно иметь метку, позволяющую пользователям знать, какую информацию они могут ввести. В первой части следующего примера изображения текст метки находится в середине текстового поля и выровнен по строке ввода. Во второй части следующего примера изображения метка перемещается выше в текстовом поле, когда пользователь щелкает текстовое поле для ввода текста.



Измените функцию EditNumberField(), чтобы добавить метку к текстовому полю:

1. В составной функции функции  EditNumberField()добавьте TextField()набор именованных параметров label в пустое лямбда-выражение:

TextField(  
//...  
   label = { }  
)

1. В лямбда-выражении вызовите функцию Text(), которая принимает : *stringResource*(R.string.*bill\_amount*)

label = { Text(stringResource(R.string.bill\_amount)) },

1. В  составной функции TextField() добавьте  набор именованных параметров singleLine со значением true:

TextField(  
  // ...  
   **singleLine = true**,  
)

Это сжимает текстовое поле до одной строки с горизонтальной прокруткой из нескольких строк.

1. Добавьте  набор параметров keyboardOptions в KeyboardOptions():

import androidx.compose.foundation.text.KeyboardOptions  
  
TextField(  
  // ...  
   **keyboardOptions = KeyboardOptions(),**  
)

Android предоставляет возможность настроить отображаемую на экране клавиатуру для ввода цифр, адресов электронной почты, URL-адресов и паролей, и это лишь некоторые из них. Чтобы узнать больше о других типах клавиатуры, см. [KeyboardType](https://developer.android.com/reference/kotlin/androidx/compose/ui/text/input/KeyboardType) .

1. Установите тип клавиатуры на цифровую клавиатуру для ввода цифр. Передайте функции KeyboardOptions именованный параметр keyboardType = KeyboardType.Number:

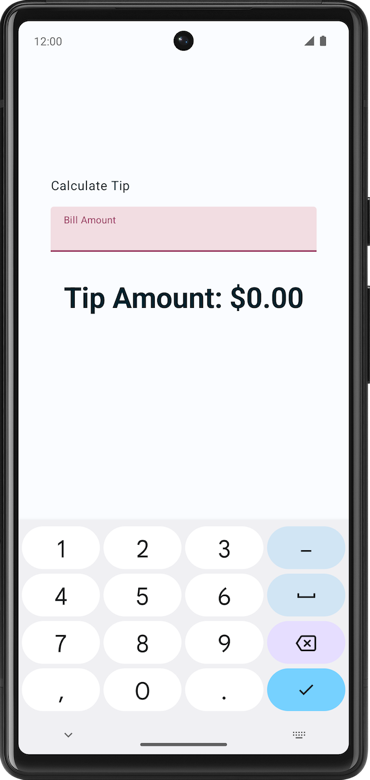
import androidx.compose.ui.text.input.KeyboardType  
  
TextField(  
  // ...  
**keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Number),**)

Завершенная функция EditNumberField() должна выглядеть следующим образом:

@Composable  
fun EditNumberField(modifier: Modifier = Modifier) {  
    var amountInput by remember { mutableStateOf("") }  
    TextField(  
        value = amountInput,  
        onValueChange = { amountInput = it },  
        singleLine = true,  
        label = { Text(stringResource(R.string.bill\_amount)) },  
        keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Number),  
        modifier = modifier  
    )  
}

1. Запустите приложение.

Изменения в клавиатуре вы можете увидеть на этом скриншоте:



## [**9. Отображение суммы чаевых**](https://developer.android.com/codelabs/basic-android-kotlin-compose-using-state?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-2-pathway-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-using-state#8)

В этом разделе вы реализуете основной функционал приложения – возможность расчета и отображения суммы чаевых.

В  файле   MainActivity.kt вам предоставляется функция private calculateTip() как часть стартового кода. Вы будете использовать эту функцию для расчета суммы чаевых:

private fun calculateTip(amount: Double, tipPercent: Double = 15.0): String {  
    val tip = tipPercent / 100 \* amount  
    return NumberFormat.getCurrencyInstance().format(tip)  
}

В приведенном выше методе вы используете формат чаевых в виде валюты [NumberFormat](https://developer.android.com/reference/java/text/NumberFormat).

Теперь ваше приложение может вычислить подсказку, но вам все равно нужно отформатировать и отобразить ее с помощью класса.

## **Используйте функцию calculateTip()**

Текст, введенный пользователем в составное текстовое поле, возвращается в функцию обратного вызова onValueChange как String даже если пользователь ввел число. Чтобы это исправить, необходимо преобразовать значение amountInput, содержащее сумму, введенную пользователем.

1. В  составной функции EditNumberField() создайте новую переменную amount, вызываемую после определения amountInput. Вызовите функцию [toDoubleOrNull](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/to-double-or-null.html) для переменной amountInput, чтобы преобразовать ее String в Double:

val amount = amountInput.toDoubleOrNull()

Эта функция toDoubleOrNull()представляет собой предопределенную функцию Kotlin, которая преобразует строку в число Double и возвращает результат или возвращает null, если строка не является допустимым представлением числа.

1. В конце инструкции добавьте оператор Элвиса ?:, который возвращает значение 0.0, если amountInput равно нулю:

val amount = amountInput.toDoubleOrNull() ?: 0.0

**Примечание** .  Оператор Элвиса ?: возвращает выражение, которое предшествует ему, если значение не равно null и выражение, которое следует за ним, если значение равно null. Это позволяет вам писать этот код более идиоматично.

1. После  переменной amount создайте еще одну  переменную val с именем tip. Инициализируйте его с помощью  calculateTip(), передав  параметр amount.

val tip = calculateTip(amount)

Функция EditNumberField() должна выглядеть следующим образом:

@Composable  
fun EditNumberField(modifier: Modifier = Modifier) {  
   var amountInput by remember { mutableStateOf("") }  
  
   val amount = amountInput.toDoubleOrNull() ?: 0.0  
   val tip = calculateTip(amount)  
  
   TextField(  
       value = amountInput,  
       onValueChange = { amountInput = it },  
       label = { Text(stringResource(R.string.bill\_amount)) },  
       modifier = Modifier.fillMaxWidth(),  
       singleLine = true,  
       keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Number)  
   )  
}

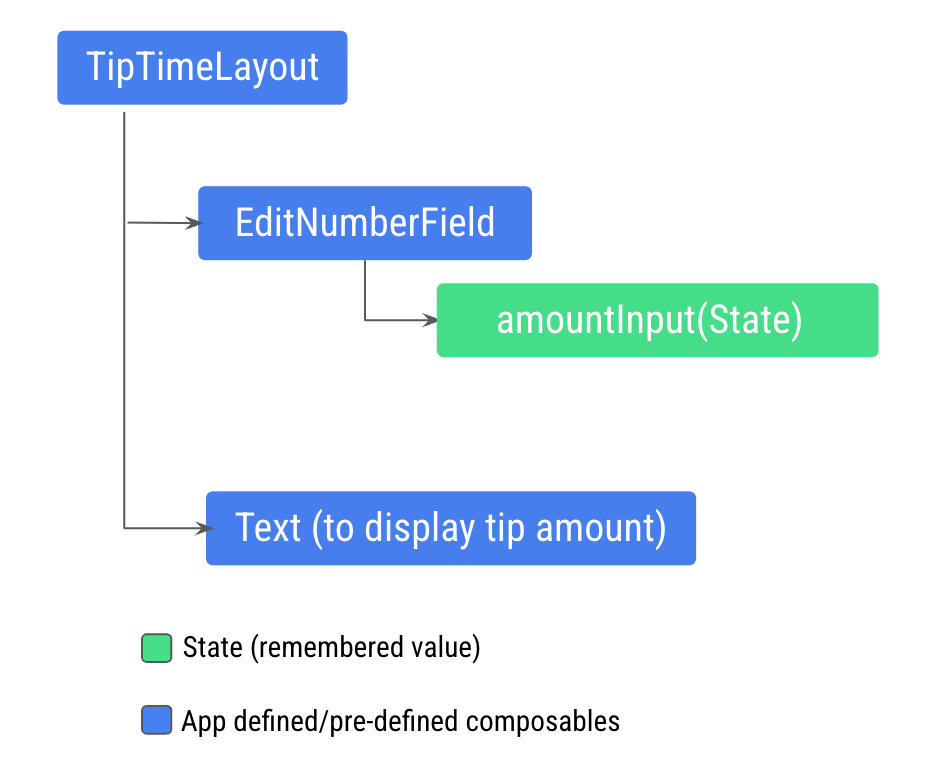
## **Показать рассчитанную сумму чаевых**

Вы написали функцию для расчета суммы чаевых, следующим шагом будет отображение рассчитанной суммы чаевых:

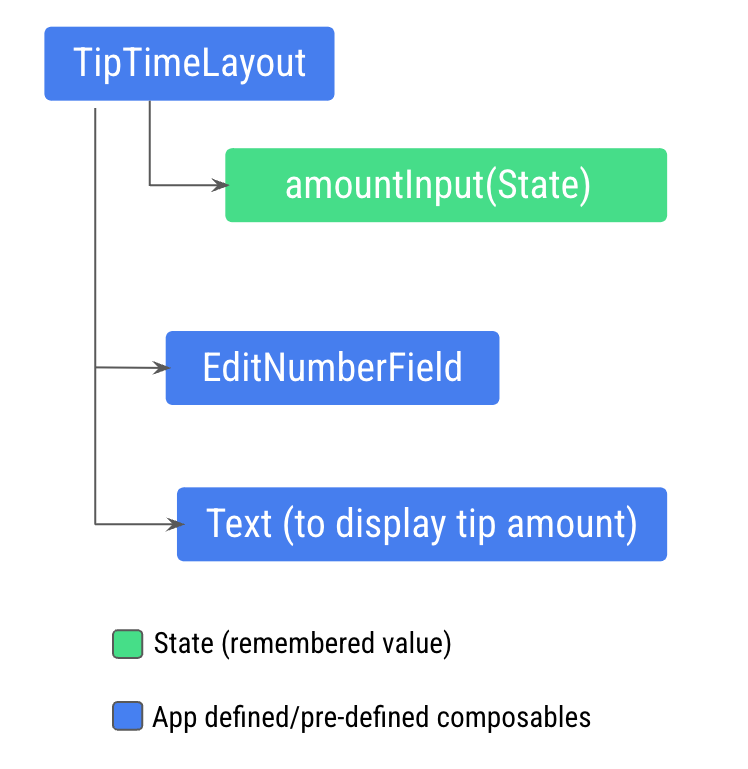
1. В  функции TipTimeLayout() в конце блока Column() обратите внимание на составной текст, который отображает  $0.00. Вы обновите это значение до рассчитанной суммы чаевых.

@Composable  
fun TipTimeLayout() {  
    Column(  
        modifier = Modifier  
            .statusBarsPadding()  
            .padding(horizontal = 40.dp)  
            .verticalScroll(rememberScrollState())  
            .safeDrawingPadding(),  
        horizontalAlignment = Alignment.CenterHorizontally,  
        verticalArrangement = Arrangement.Center  
    ) {  
        // ...  
        Text(  
            text = stringResource(R.string.tip\_amount, "$0.00"),  
            style = MaterialTheme.typography.displaySmall  
        )  
        // ...  
    }  
}

Вам необходимо получить доступ к  переменной amountInput в функции TipTimeLayout(), чтобы вычислить и отобразить сумму чаевых, но эта переменная amountInput представляет собой состояние текстового поля, определенного в составной функции EditNumberField(), поэтому вы пока не можете вызывать ее из  функции TipTimeLayout(). Это изображение иллюстрирует структуру кода:



Эта структура не позволит вам отображать сумму чаевых в новом  составном объекте Text, поскольку составному объекту Text необходим доступ к переменной amount, рассчитанной на основе переменной amountInput. Вам нужно предоставить переменную amount функции TipTimeLayout(). Это изображение иллюстрирует желаемую структуру кода, которая делает  компонуемый объект EditNumberField() без состояния:



Этот шаблон называется **подъемом состояния** . В следующем разделе вы поднимете состояние составного объекта, чтобы сделать его **апатридом**.

**Примечание.** Составной объект без сохранения состояния — это составной объект, который не хранит свое собственное состояние. Он отображает любое состояние, заданное в качестве входных аргументов.

## **10.** [**State hoisting**](https://developer.android.com/codelabs/basic-android-kotlin-compose-using-state?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-2-pathway-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-using-state#9)

В этом разделе вы узнаете, как решить, где определить свое состояние, чтобы вы могли повторно использовать и делиться своими составными объектами.

В составной функции вы можете определить переменные, которые сохраняют состояние для отображения в пользовательском интерфейсе. Например, вы определили переменную amountInput как состояние в  составном объекте EditNumberField().

Когда ваше приложение становится более сложным и другим составным объектам требуется доступ к состоянию внутри составного объекта EditNumberField(), вам необходимо рассмотреть возможность поднятия или извлечения состояния из  составной функции EditNumberField().

## **Понимание составных элементов с сохранением состояния и без сохранения состояния**

Вы должны поднимать состояние, когда вам нужно:

* Поделитесь состоянием с помощью нескольких компонуемых функций.
* Создайте компонуемый объект без сохранения состояния, который можно повторно использовать в вашем приложении.

Когда вы извлекаете состояние из составной функции, результирующая составная функция называется без состояния. То есть составные функции можно сделать без состояния, извлекая из них состояние.

Составной объект без сохранения состояния — это составной объект, который не имеет состояния, то есть он не содержит, не определяет и не изменяет новое состояние.

С другой стороны, составной объект с сохранением состояния — это составной объект, который владеет частью состояния, которая может меняться с течением времени.

**Примечание.** В реальных приложениях может быть сложно добиться 100% компонуемого объекта без сохранения состояния в зависимости от обязанностей компонуемого объекта. Вы должны проектировать свои составные объекты таким образом, чтобы они владели как можно меньшим количеством состояния и позволяли поднимать состояние, когда это имеет смысл, путем предоставления его в API составного объекта.

**Поднятие состояния** — это шаблон перемещения состояния к вызывающему объекту, чтобы сделать компонент без состояния.

Применительно к составным объектам это часто означает введение в составной объект двух параметров:

* Параметр  value: T, представляет собой текущее значение для отображения.
* onValueChange: (T) -> Unit— лямбда обратного вызова, которая срабатывает при изменении значения, чтобы состояние можно было обновить в другом месте, например, когда пользователь вводит текст в текстовое поле.

Поднимите состояние в  функции EditNumberField():

1. Обновите  определение функции EditNumberField(), чтобы поднять состояние, добавив параметры value и onValueChange:

@Composable  
fun EditNumberField(  
   **value: String,  
   onValueChange: (String) -> Unit,**   modifier: Modifier = Modifier  
) {  
//...

Параметр value имеет  тип String, а параметр onValueChange имеет тип (String) -> Unit, поэтому это функция, которая принимает значение String в качестве входных данных и не имеет возвращаемого значения. Параметр onValueChange используется в качестве обратного вызова onValueChange, передаваемого в компонуемый объект TextField.

**Примечание.** Лучше всего предоставлять параметр [Modifier](https://developer.android.com/reference/kotlin/androidx/compose/ui/Modifier) по умолчанию для всех составных функций, что повышает возможность повторного использования. Его следует добавить в качестве первого необязательного параметра после всех обязательных параметров.

1. В функции EditNumberField()обновите составную функцию TextField(), чтобы использовать переданные параметры:

TextField(  
   value = value,  
   onValueChange = onValueChange,  
   // Rest of the code  
)

1. Поднимите состояние, переместите запомненное состояние из функции EditNumberField() в функцию TipTimeLayout():

@Composable  
fun TipTimeLayout() {  
   var amountInput by remember { mutableStateOf("") }  
  
   val amount = amountInput.toDoubleOrNull() ?: 0.0  
   val tip = calculateTip(amount)  
    
   Column(  
       //...  
   ) {  
       //...  
   }  
}

1. Вы подняли состояние на TipTimeLayout(), теперь передайте его EditNumberField(). В функции TipTimeLayout() обновите вызов функции, чтобы использовать поднятое состояние:*EditNumberField*()

EditNumberField(  
   value = amountInput,  
   onValueChange = { amountInput = it },  
   modifier = Modifier  
       .padding(bottom = 32.dp)  
       .fillMaxWidth()  
)

Это делает *EditNumberField* лицом без гражданства. Вы передали состояние пользовательского интерфейса его предку, TipTimeLayout(). Теперь он TipTimeLayout() является владельцем состояния (amountInput).

## **Позиционное форматирование**

Позиционное форматирование используется для отображения динамического содержимого в виде строк. Например, представьте, что вы хотите, чтобы в текстовом поле **«Сумма чаевых»** отображалось значение xx.xx, которое может представлять собой любую сумму, рассчитанную и отформатированную в вашей функции. Чтобы сделать это в файле strings.xml, вам необходимо определить строковый ресурс с аргументом-заполнителем, как в этом фрагменте кода:

// No need to copy.  
  
// In the res/values/strings.xml file  
<string name="tip\_amount">Tip Amount: %s</string>

В коде создания вы можете иметь несколько аргументов-заполнителей любого типа. Заполнитель string— это %s.

Обратите внимание на текст TipTimeLayout(), который можно компоновать . Вы передаете отформатированную подсказку в качестве аргумента функции stringResource().

// No need to copy  
Text(  
   text = stringResource(R.string.tip\_amount, "$0.00"),  
   style = MaterialTheme.typography.displaySmall  
)

1. В функции TipTimeLayout() используйте свойство tip для отображения суммы чаевых. Обновите параметр text составного объекта Text, чтобы использовать переменную tip в качестве параметра.

Text(  
     text = stringResource(R.string.tip\_amount, tip),  
     // ...

Завершенные функции TipTimeLayout() и EditNumberField() должны выглядеть следующим образом:

@Composable  
fun TipTimeLayout() {  
   var amountInput by remember { mutableStateOf("") }  
   val amount = amountInput.toDoubleOrNull() ?: 0.0  
   val tip = calculateTip(amount)  
  
   Column(  
       modifier = Modifier  
            .statusBarsPadding()  
            .padding(horizontal = 40.dp)  
            .verticalScroll(rememberScrollState())  
            .safeDrawingPadding(),  
       horizontalAlignment = Alignment.CenterHorizontally,  
       verticalArrangement = Arrangement.Center  
   ) {  
       Text(  
           text = stringResource(R.string.calculate\_tip),  
           modifier = Modifier  
               .padding(bottom = 16.dp, top = 40.dp)  
               .align(alignment = Alignment.Start)  
       )  
       EditNumberField(  
           value = amountInput,  
           onValueChange = { amountInput = it },  
           modifier = Modifier  
               .padding(bottom = 32.dp)  
               .fillMaxWidth()  
       )  
       Text(  
           text = stringResource(R.string.tip\_amount, tip),  
           style = MaterialTheme.typography.displaySmall  
       )  
       Spacer(modifier = Modifier.height(150.dp))  
   }  
}  
  
@Composable  
fun EditNumberField(  
   value: String,  
   onValueChange: (String) -> Unit,  
   modifier: Modifier = Modifier  
) {  
   TextField(  
       value = value,  
       onValueChange = onValueChange,  
       singleLine = true,  
       label = { Text(stringResource(R.string.bill\_amount)) },  
       keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Number),  
       modifier = modifier  
   )  
}

Подводя итог, вы перенесли состояние amountInput из EditNumberField() в TipTimeLayout() составное. Чтобы текстовое поле работало как раньше, вам необходимо передать компонуемой функции EditNumberField()два аргумента: значение amountInput и обратный вызов лямбда, который обновляет  значение amountInput из ввода пользователя. Эти изменения позволяют рассчитывать чаевые на основе свойства amountInput, чтобы отобразить их пользователю.

1. Запустите приложение на эмуляторе или устройстве, а затем введите значение в текстовое поле **суммы счета** . Сумма чаевых в размере 15 процентов от суммы счета отображается, как вы можете видеть на этом изображении:

