

# Введение в состояние в Compose

---

## Прежде чем начать

Эта практическая работа расскажет вам о **состоянии** и о том, как его можно использовать и манипулировать им в Jetpack Compose.

По своей сути, состояние в приложении - это любое значение, которое может изменяться с течением времени. Это определение очень широкое и включает в себя все - от базы данных до переменной в вашем приложении. Подробнее о базах данных вы узнаете в одном из следующих разделов, а пока вам нужно знать, что база данных - это организованная коллекция структурированной информации, например, файлы на вашем компьютере.

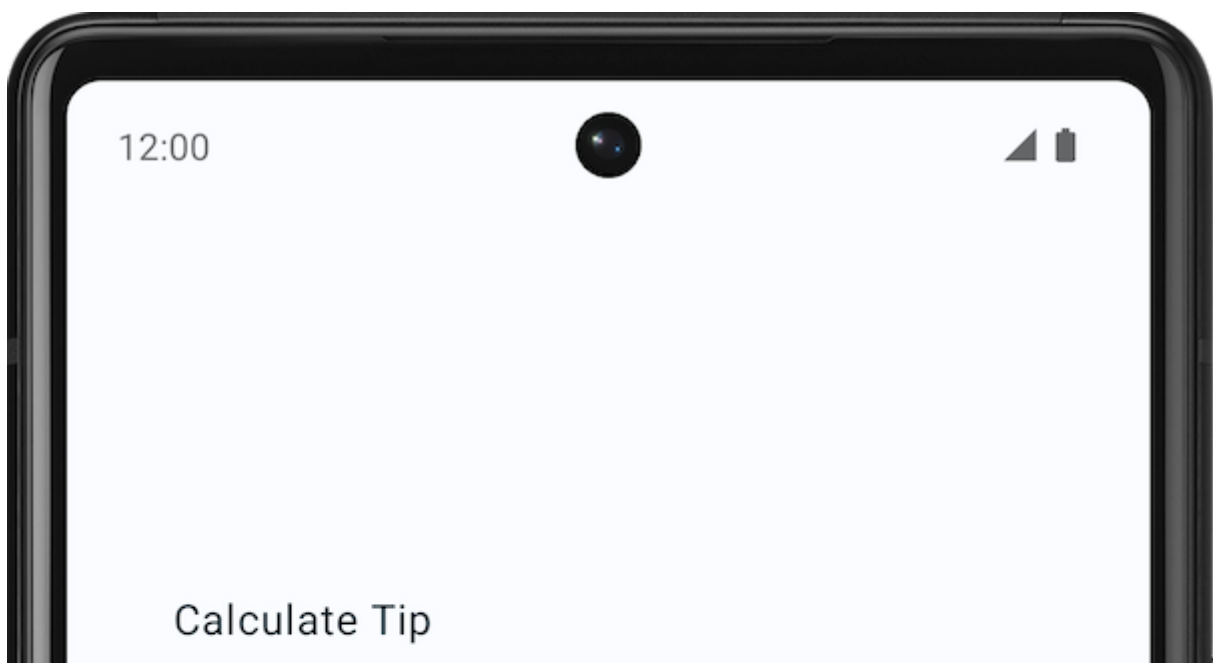
Все приложения для Android отображают состояние для пользователя. Несколько примеров состояния в приложениях Android включают:

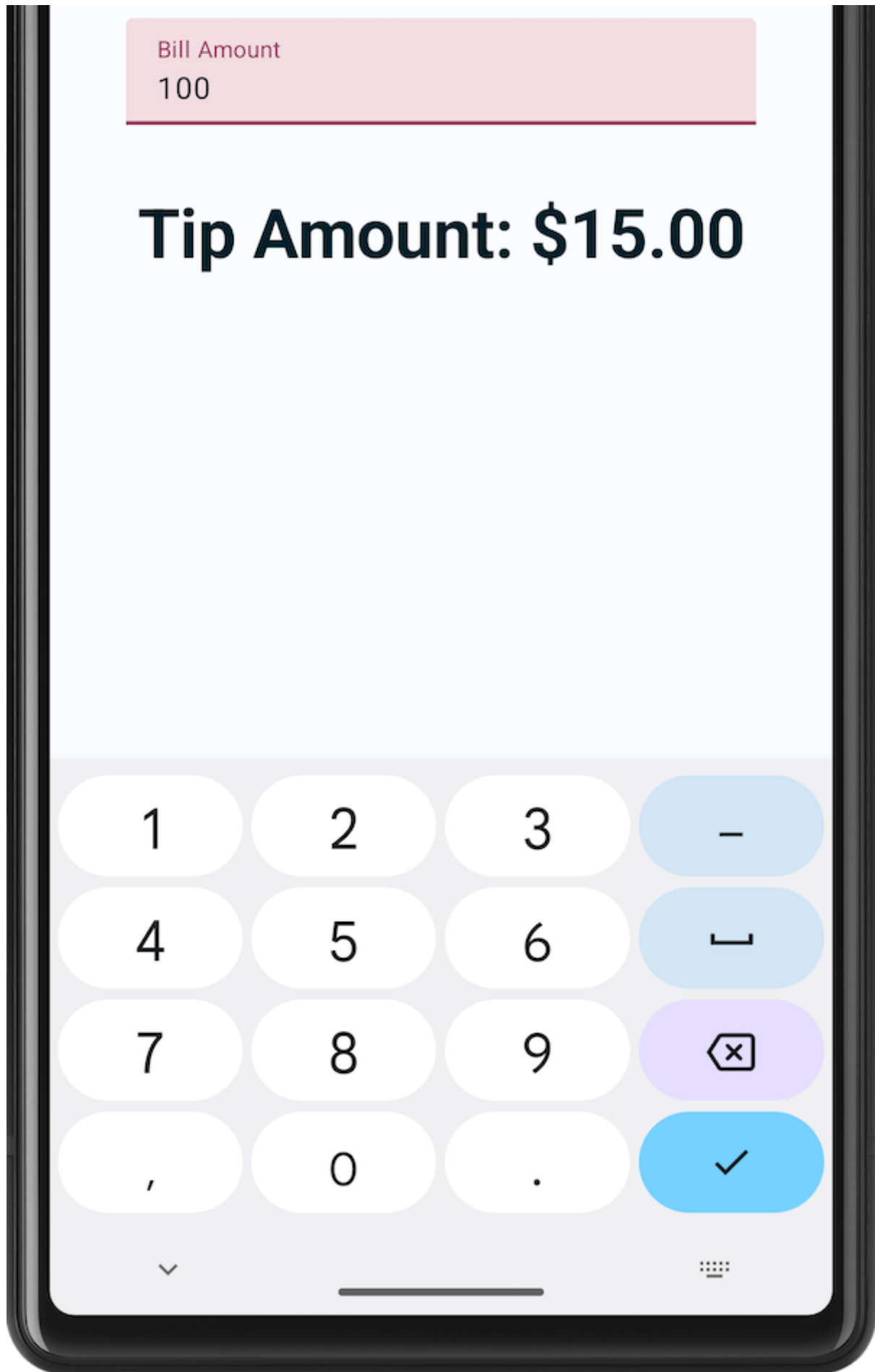
- Сообщение, которое появляется, когда не удастся установить сетевое соединение.
- Формы, например формы регистрации. Состояние может быть заполнено и отправлено.
- Настраиваемые элементы управления, например кнопки. Состояние может быть не нажато, нажато (анимация отображения) или нажато (действие onClick).

В этом уроке вы узнаете, как использовать состояние и думать о нем при работе с Compose. Для этого вы создадите приложение-калькулятор чаевых под названием **Tip Time** со встроенными в Compose элементами пользовательского интерфейса:

- Текстовое поле, предназначенное для ввода и редактирования текста.
- A Text - для отображения текста.
- Spacer - для отображения пустого пространства между элементами пользовательского интерфейса.

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





Необходимые условия

- Базовое понимание Compose, например аннотации @Composable.
- Базовое знакомство с макетами Compose, такими как композиты макетов Row и Column.
- Базовое знакомство с модификаторами, такими как функция Modifier.padding().
- Знакомство с композитом Text.

### Что вы узнаете

- Как думать о состоянии в пользовательском интерфейсе.
- Как Compose использует состояние для отображения данных.
- Как добавить текстовое поле в ваше приложение.
- Как поднять состояние.

### Что вы создадите

Приложение-калькулятор чаевых под названием **Tip Time**, которое рассчитывает сумму чаевых на основе суммы обслуживания.

### Начните

- Посмотрите онлайн-калькулятор чаевых от Google. Обратите внимание, что это всего лишь пример, и это не то приложение для Android, которое вы будете создавать в этом курсе.

Bill	<input type="text" value="100.00"/>	Tip	\$15.00
Tip %	<div><div>-</div><div><input type="text" value="15%"/></div><div>+</div></div>	Total	\$115.00
Number of people	<div><div>-</div><div><input type="text" value="1"/></div><div>+</div></div>		

Bill	<input type="text" value="0.00"/>	Tip	\$0.00
Tip %	<div><div>-</div><div><input type="text" value="15%"/></div><div>+</div></div>	Total	\$0.00
Number of people	<div><div>-</div><div><input type="text" value="1"/></div><div>+</div></div>		

- Введите разные значения в поля Счет и % чаевых. Значения чаевых и общей суммы изменятся.

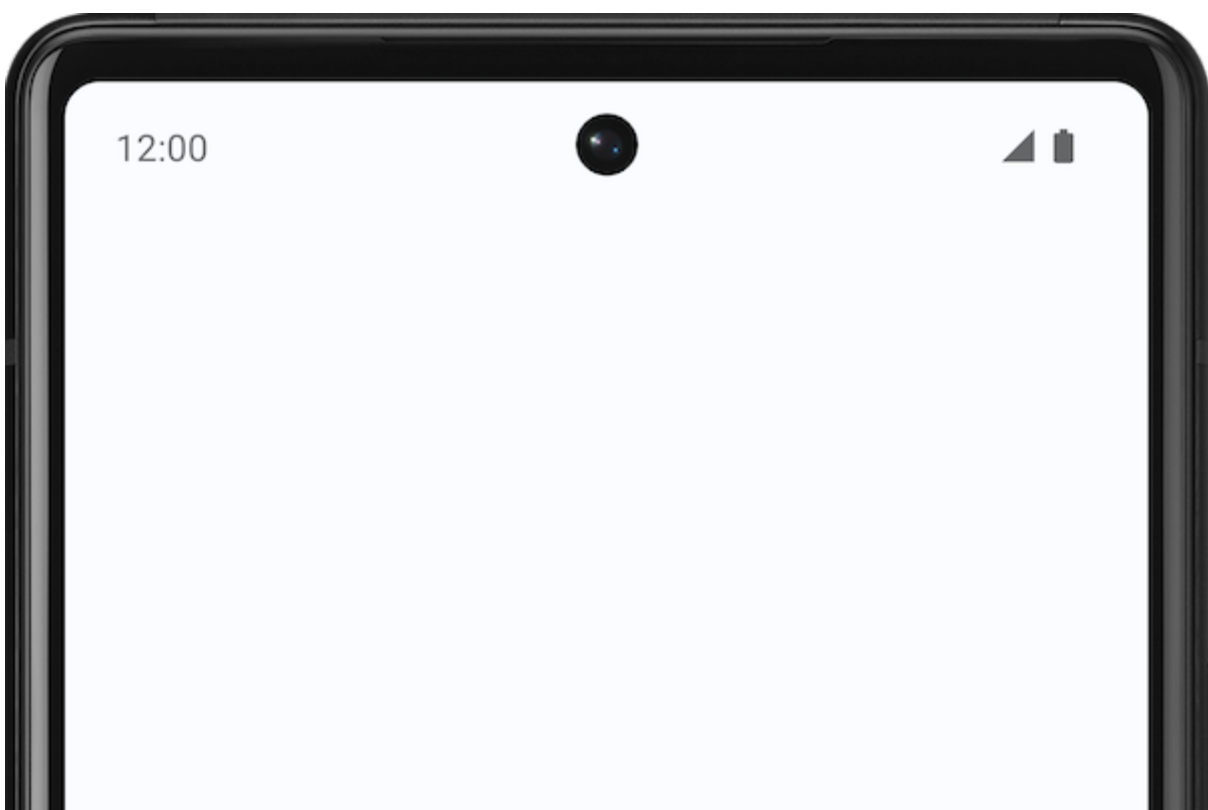
Bill	<input type="text" value="150.00"/>	
Tip %	<div><div>-</div><div>20%</div><div>+</div></div>	Tip \$30.00
Number of people	<div><div>-</div><div>1</div><div>+</div></div>	Total \$180.00

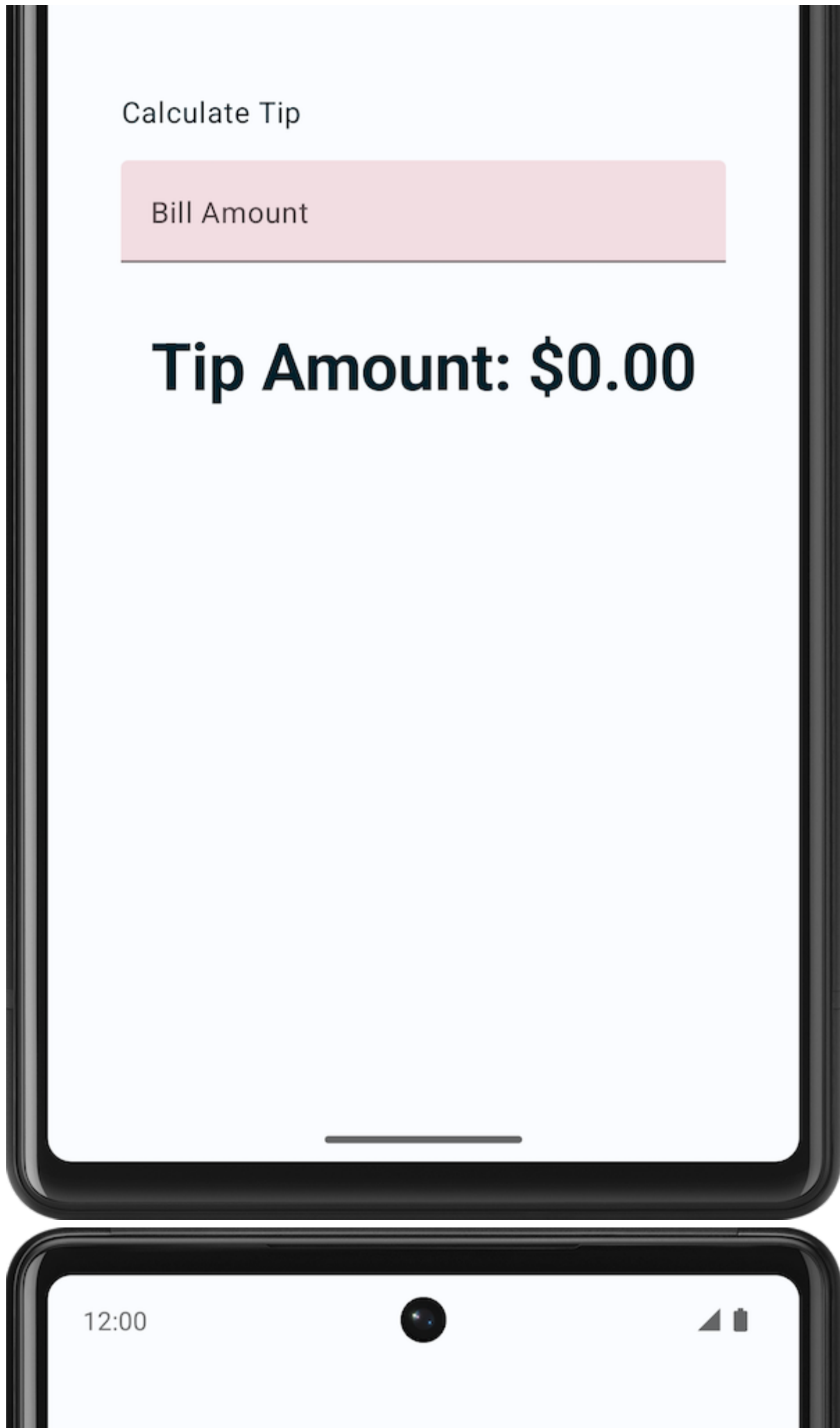
Обратите внимание, что при вводе значений чаевые и общая сумма обновляются. К концу следующего урока вы сможете разработать подобное приложение-калькулятор чаевых в Android.

В этом шаге вы создадите простое приложение для Android с калькулятором чаевых.

Разработчики часто работают именно так: создают простую версию приложения (даже если она выглядит не очень хорошо), а затем добавляют дополнительные функции и делают ее более привлекательной.

К концу этого урока ваше приложение для расчета чаевых будет выглядеть так, как показано на этих скриншотах. Когда пользователь вводит сумму счета, ваше приложение отображает предлагаемую сумму чаевых. Процент чаевых пока что жестко закодирован на 15%. В следующем уроке вы продолжите работу над своим приложением и добавите дополнительные функции, например, установку пользовательского процента чаевых.





Calculate Tip

Bill Amount

100

**Tip Amount: \$15.00**

1

2

3

–

4

5

6

⌋

7

8

9

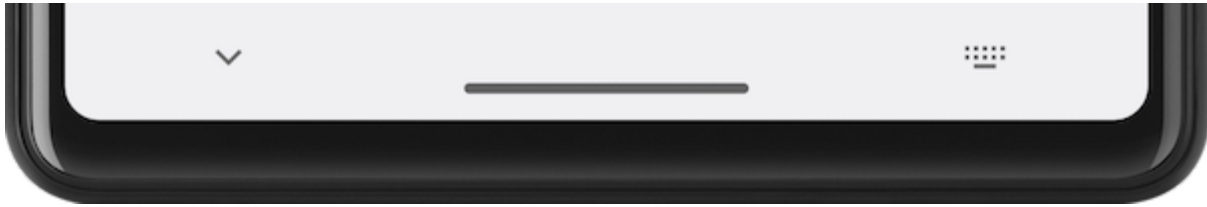
⌫

,

0

.

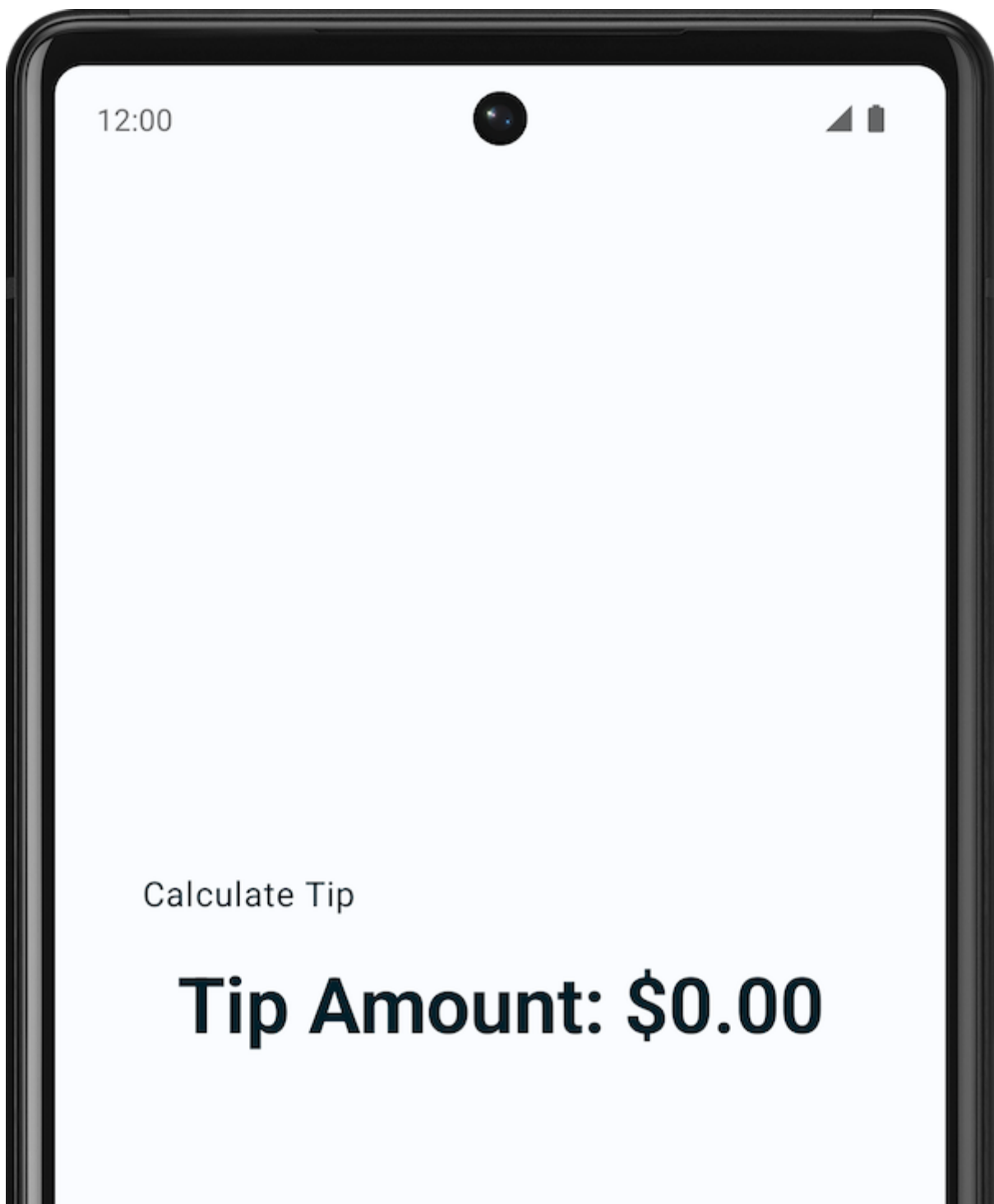
✓

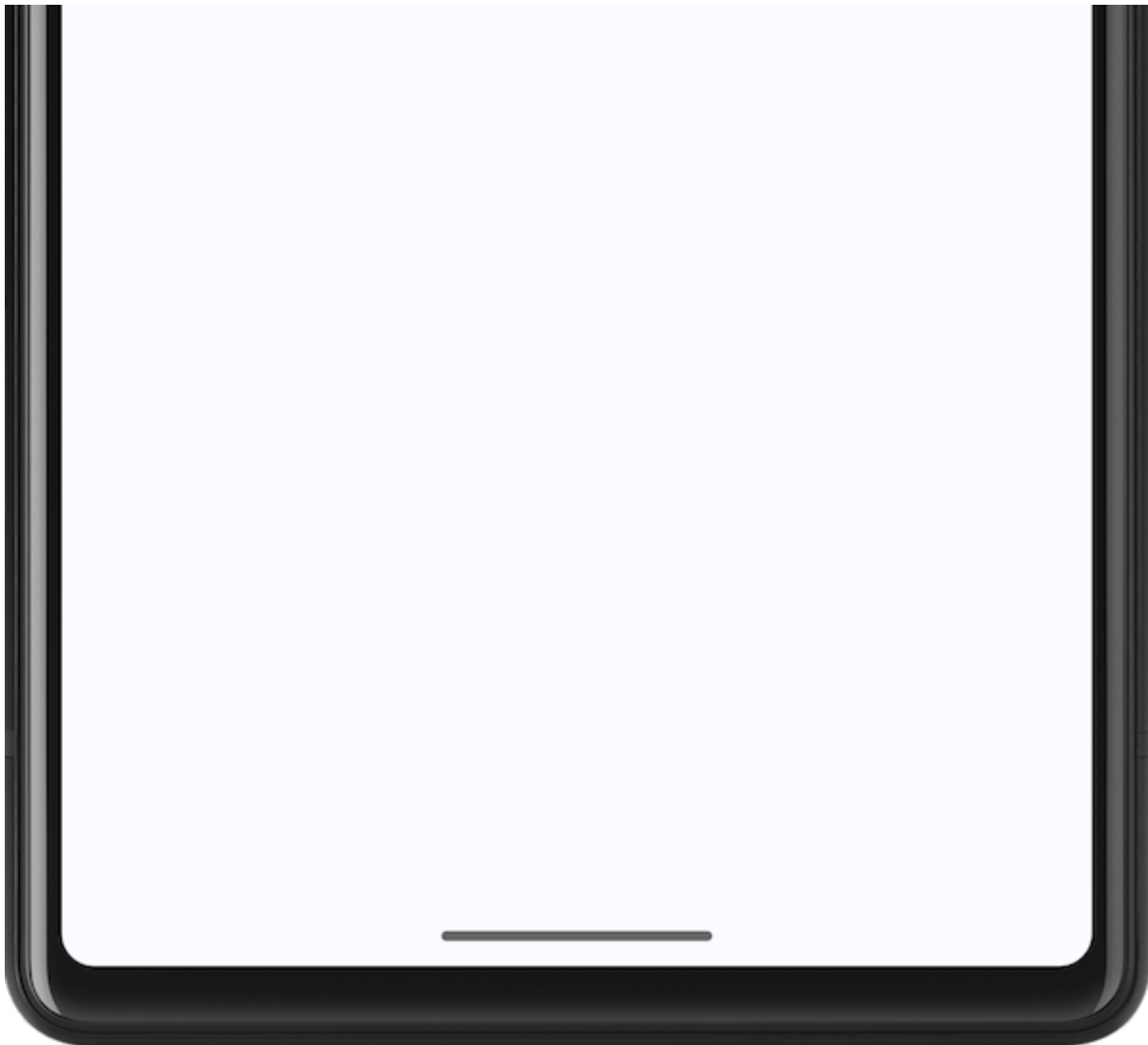


## Обзор стартового приложения

Чтобы ознакомиться со стартовым кодом, выполните следующие действия:

- Откройте проект со стартовым кодом в Android Studio.
- Запустите приложение на Android-устройстве или эмуляторе.
- Вы заметите два текстовых компонента: один - для надписи, другой - для отображения суммы чаевых.





## Просмотр стартового кода

Стартовый код содержит текстовые компоненты. На этом пути вы добавите текстовое поле для ввода пользователем. Вот краткое описание некоторых файлов для начала работы.

- res > values > 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 файл string.xml 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 текстовыми композитами, которые вы видите на скриншотах. Здесь также есть композит Spacer, добавляющий пространство для эстетики.



Функция `calculateTip()`, которая принимает сумму счета и рассчитывает сумму чаевых в размере 15 %. Параметр `tipPercent` установлен в значение 15,0 по умолчанию. Таким образом, значение чаевых по умолчанию пока равно 15 %. В следующем кодовом примере вы получите 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)
}
```

В блоке `Surface()` функции `onCreate()` вызывается функция `TipTimeLayout()`. Она отображает макет приложения на устройстве или в эмуляторе.

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

```
    }  
  }  
}
```

В блоке `TipTimeTheme` функции `TipTimeLayoutPreview()` вызывается функция `TipTimeLayout()`. Она отображает макет приложения в Design и в панели Split.

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

## TipTimeLayoutPreview

Calculate Tip

**Tip Amount: \$0.00**

### Приём ввода от пользователя

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

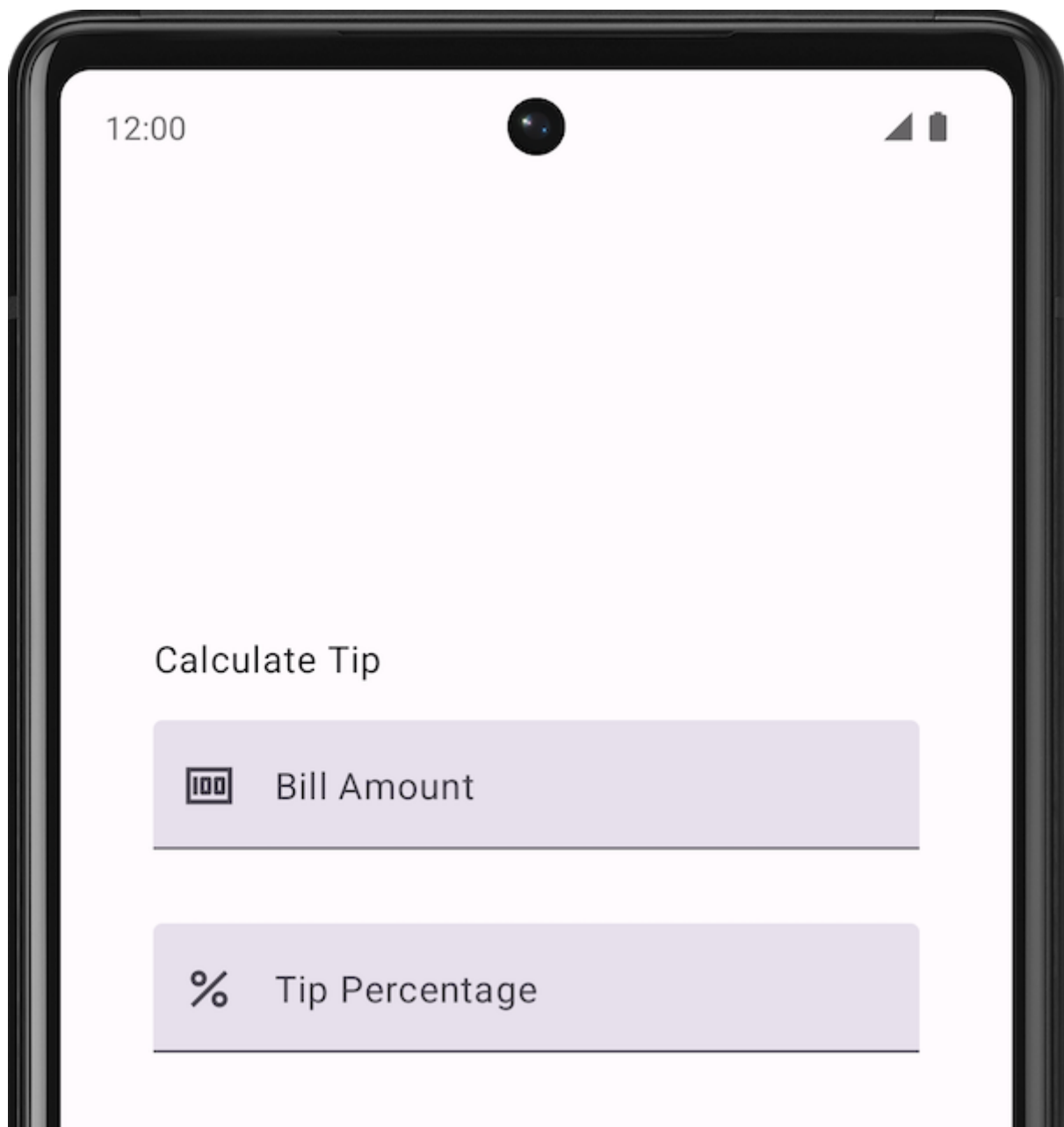
изображении:

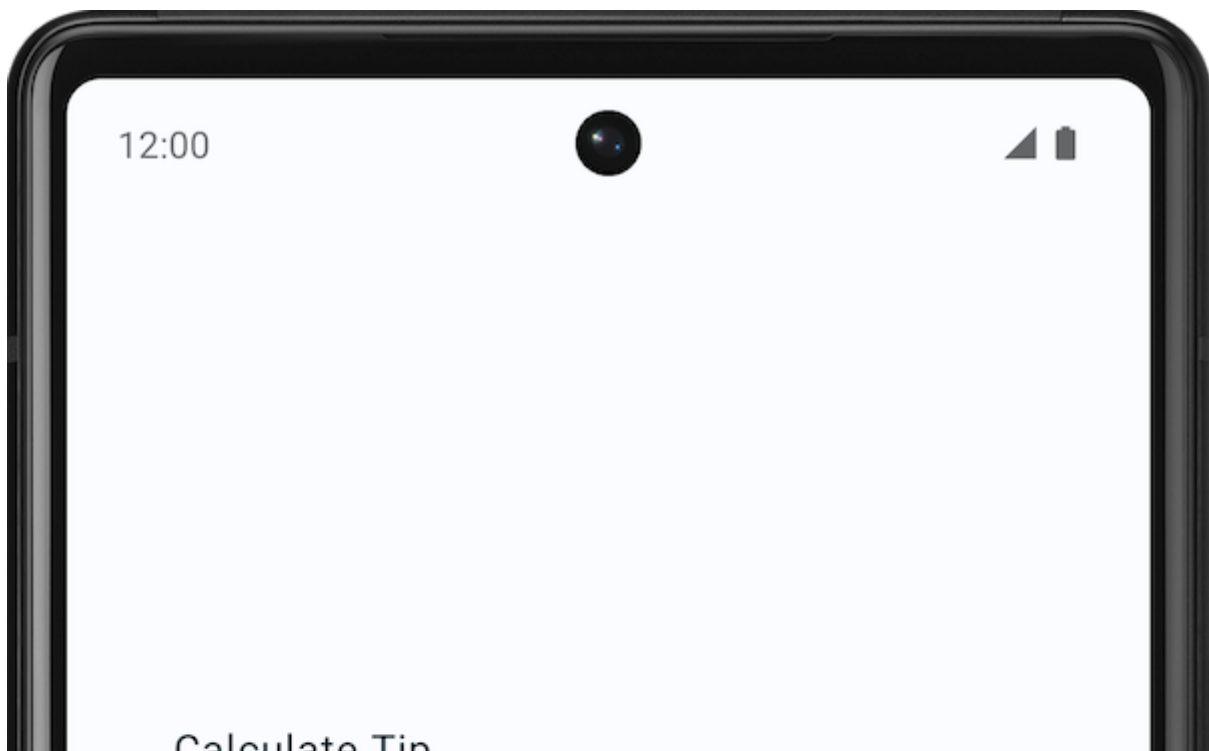
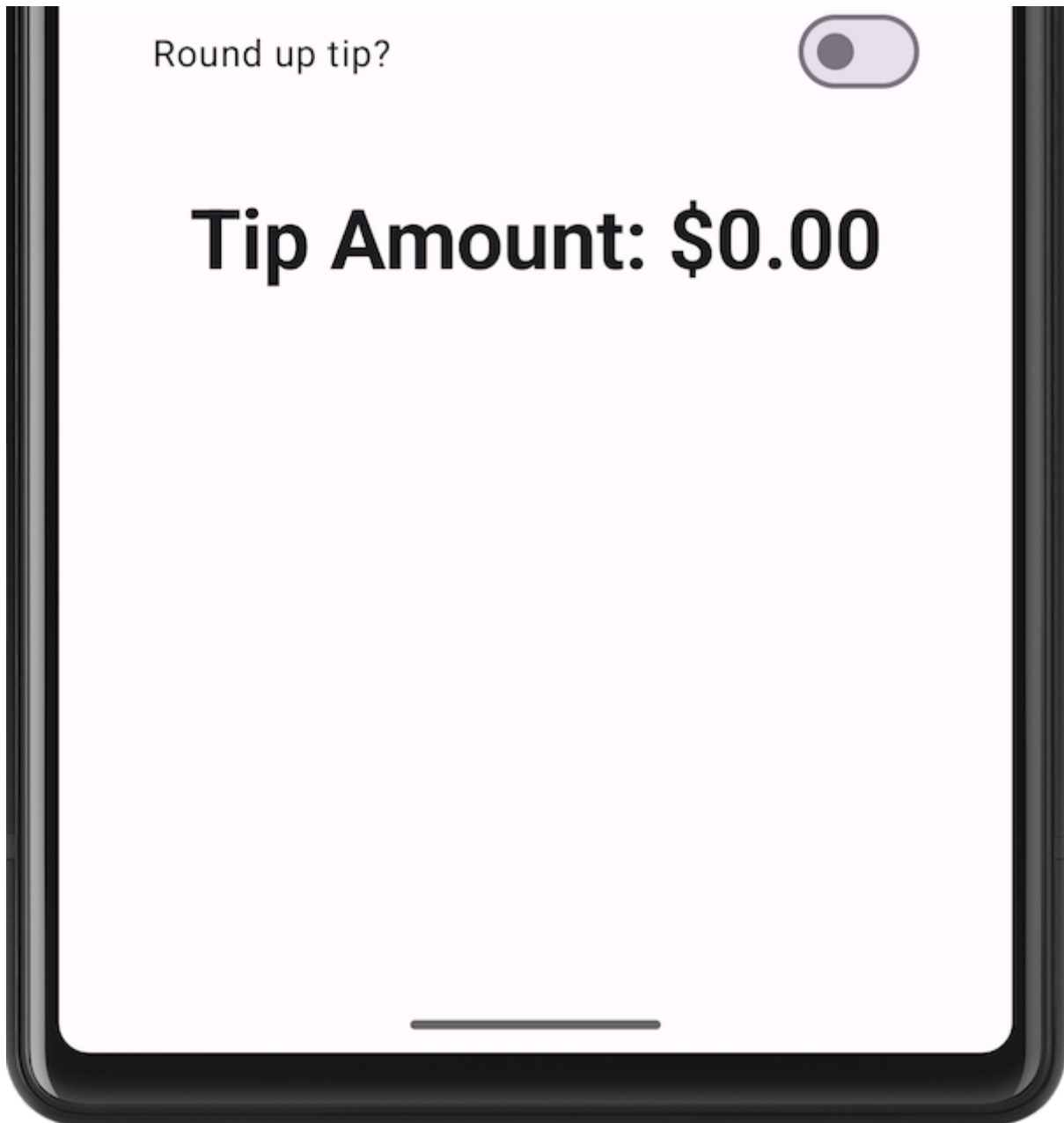
Label  
Hello

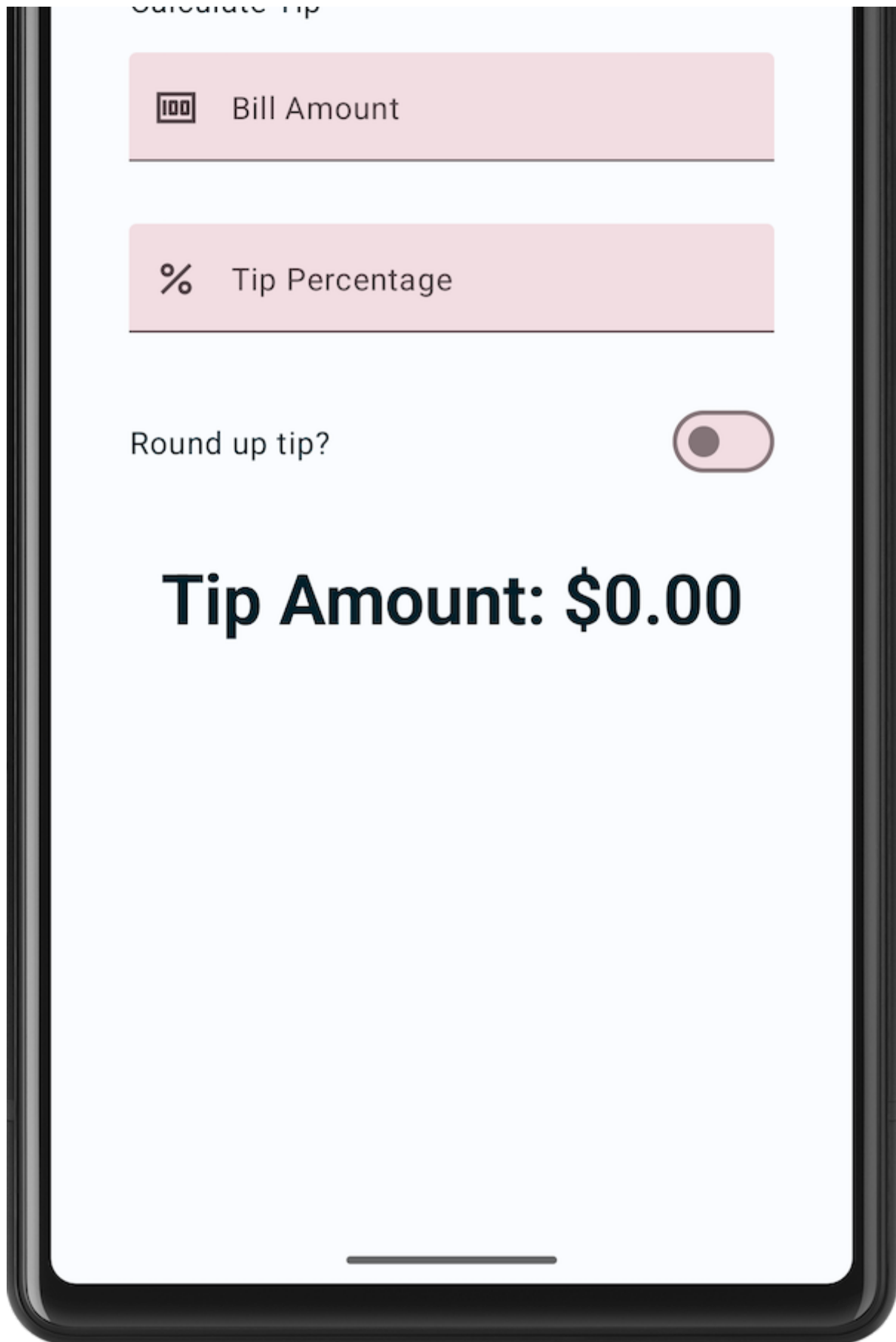
В вашем приложении используются пользовательские стили и темы.

**Стили и темы** - это набор атрибутов, определяющих внешний вид отдельного элемента пользовательского интерфейса. Стил может задавать такие атрибуты, как цвет шрифта, размер шрифта, цвет фона и многое другое, что может быть применено ко всему приложению. В последующих уроках будет рассказано, как реализовать их в вашем приложении. На данный момент это уже сделано для вас, чтобы сделать ваше приложение более красивым.

Чтобы лучше понять, вот боковое сравнение версии приложения с пользовательской темой и без нее.



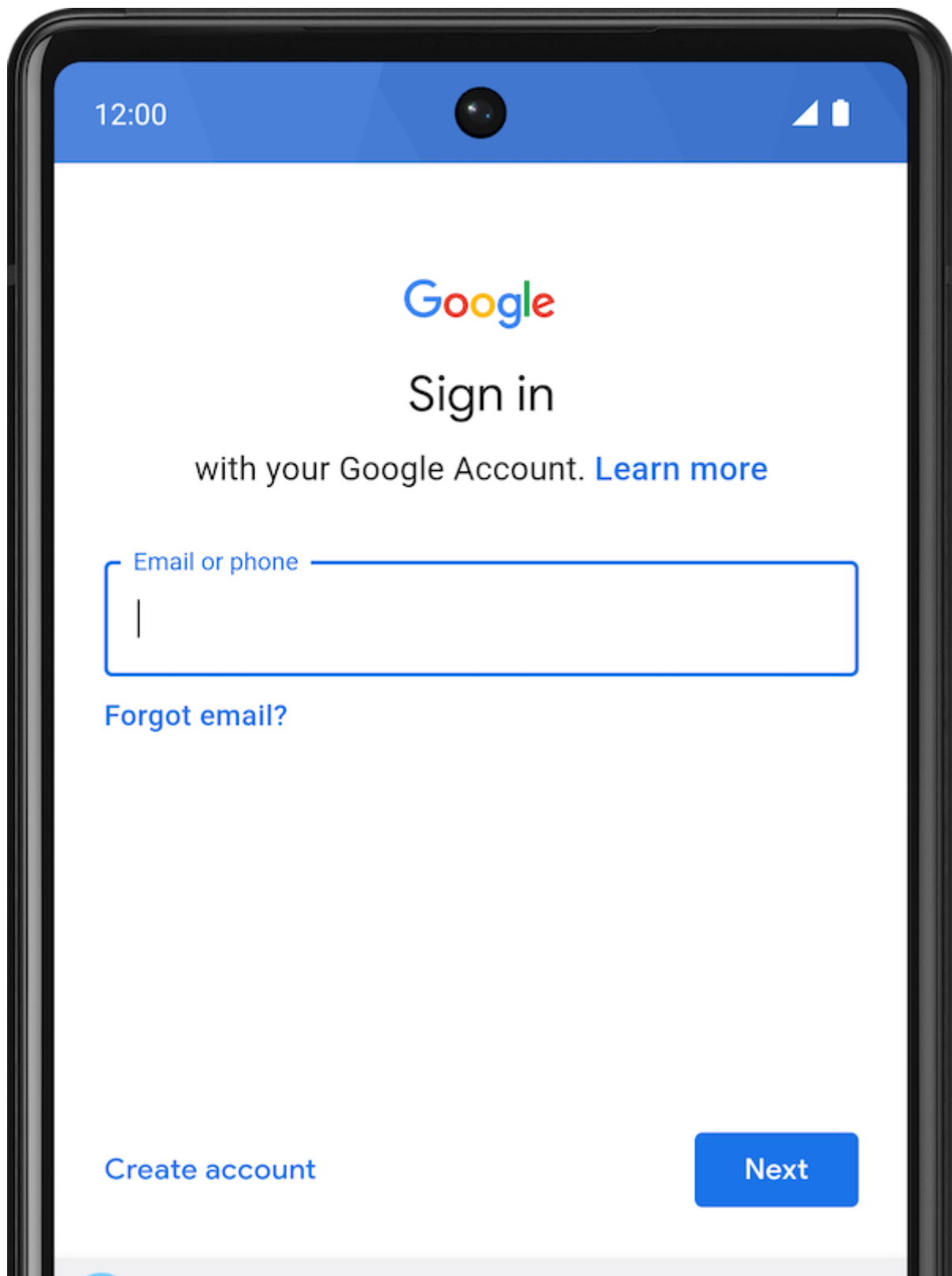


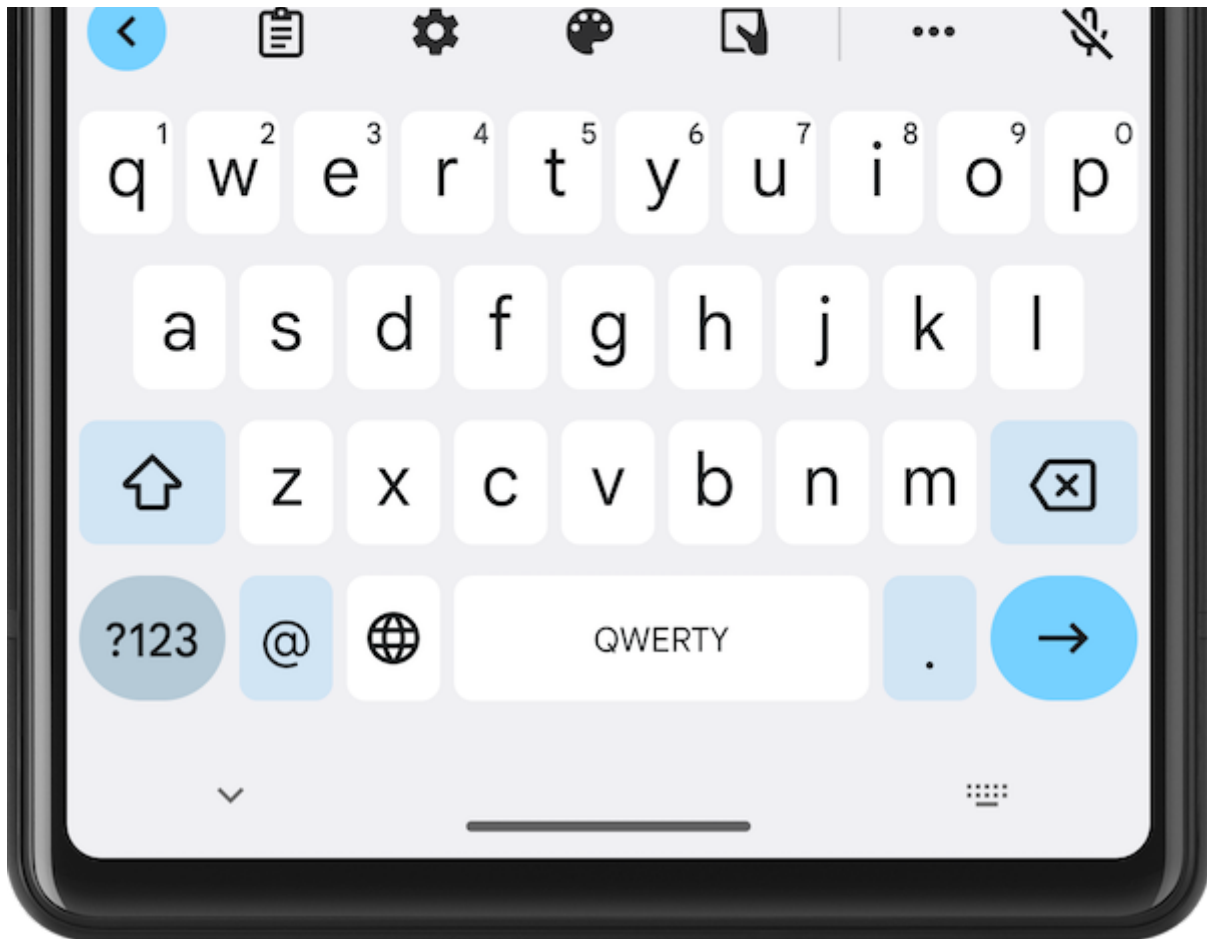


Примечание: Когда вы добавляете текстовое поле в ваше приложение, оно будет иметь цветовую схему как часть пользовательской темы, примерно так:

label  
Hello

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





- Добавьте в приложение составную функцию `TextField`:

В файл `MainActivity.kt` добавьте составную функцию `EditNumberField()`, которая принимает параметр `Modifier`. В теле функции `EditNumberField()` под `TipTimeLayout()` добавьте `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
    )
}
```

Обратите внимание на переданные параметры:

Параметр `value` - это текстовое поле, в котором отображается строковое значение, которое вы передали сюда. Параметр `onValueChange` - это обратный вызов лямбды, который срабатывает, когда пользователь вводит текст в текстовое поле.

- Импортируйте эту функцию:

```
import androidx.compose.material3.TextField
```

В композите `TipTimeLayout()` в строке после первой текстовой композитной функции вызовите функцию `EditNumberField()`, передав ей следующий модификатор.

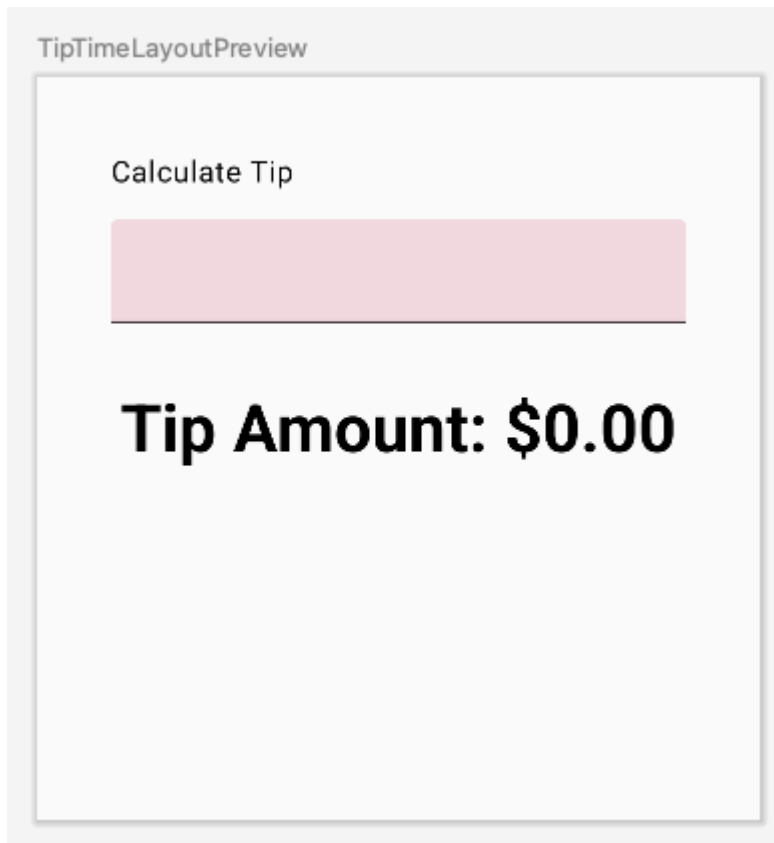
```
import androidx.compose.foundation.layout.fillMaxWidth

@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(
            ...
        )
        ...
    }
}
```

Это выведет текстовое поле на экран.

На панели Design вы должны увидеть текст Calculate Tip, пустое текстовое поле и составной текст Tip Amount.





## Используйте состояние в Compose

Состояние в приложении - это любая величина, которая может меняться с течением времени. В этом приложении состояние - это сумма счета.

- Добавьте переменную для хранения состояния:

В начале функции `EditNumberField()` с помощью ключевого слова `val` добавьте переменную `amountInput`, установив в ней значение «0»:

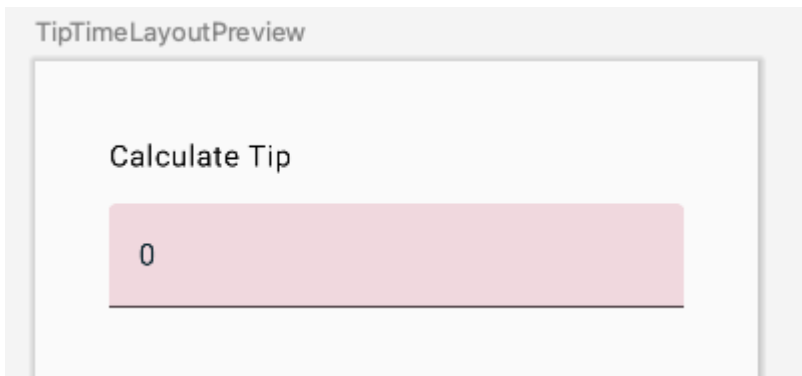
```
val amountInput = "0"
```

Это состояние приложения для суммы счета.

- Установите именованный параметр `value` в значение `amountInput`:

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

- Проверьте предварительный просмотр. В текстовом поле отображается значение, установленное для переменной `state`, как показано на этом изображении:



- Запустите приложение в эмуляторе, попробуйте ввести другое значение. Закодированное состояние останется неизменным, потому что составное поле `TextField` не обновляется само по себе. Оно обновляется при изменении параметра `value`, который устанавливается в свойство `amountInput`.

Переменная `amountInput` представляет собой состояние текстового поля. Иметь жестко закодированное состояние бесполезно, поскольку его нельзя изменить, и оно не отражает пользовательский ввод. Вам нужно обновлять состояние приложения, когда пользователь обновляет сумму счета.

## Композиция

Композиции в вашем приложении описывают пользовательский интерфейс, который показывает колонку с текстом, разделитель и текстовое поле. В тексте отображается надпись `Calculate Tip`, а в текстовом поле - значение `0` или любое значение по умолчанию.

**Compose** - это декларативный UI-фреймворк, то есть вы объявляете, как должен выглядеть пользовательский интерфейс в вашем коде. Если бы вы хотели, чтобы в текстовом поле изначально отображалось значение `100`, вы бы установили в коде композитных элементов начальное значение `100`.

Как быть, если вы хотите, чтобы ваш пользовательский интерфейс менялся во время работы приложения или при взаимодействии пользователя с приложением? Например, что если вы хотите обновить переменную `amountInput` значением, введенным пользователем, и отобразить его в текстовом поле? Тогда вы прибегнете к процессу, называемому **рекомпозицией**, чтобы обновить состав приложения.

**Композиция** - это описание пользовательского интерфейса, создаваемого Compose при выполнении композитных функций. Приложения Compose вызывают композитные функции для преобразования данных в пользовательский интерфейс.

Если происходит изменение состояния, Compose повторно выполняет затронутые композитные функции с новым состоянием, что создает обновленный пользовательский интерфейс - это называется **рекомпозицией**. Compose планирует рекомпозицию для вас.

Когда Compose запускает композиты в первый раз во время начальной композиции, он отслеживает композиты, которые вы вызываете для описания пользовательского интерфейса в композиции.

**Рекомпозиция** - это когда Compose повторно выполняет композиции, которые могли измениться в ответ на изменения данных, и затем обновляет композицию, чтобы отразить все изменения.

**Композиция** может быть создана только путем первоначальной композиции и обновлена путем **рекомпозиции**.

Единственный способ изменить композицию - это рекомпозиция.

Для этого Compose необходимо знать, какое состояние отслеживать, чтобы запланировать рекомпозицию при получении обновления. В вашем случае это переменная `amountInput`, поэтому при изменении ее значения Compose запланирует перекомпозицию.

Вы используете типы `State` и `MutableState` в Compose, чтобы сделать состояние вашего приложения **наблюдаемым**, или отслеживаемым, Compose. Тип `State` является неизменяемым, поэтому вы можете только прочитать его значение, в то время как тип `MutableState` является изменяемым. Вы можете использовать функцию `mutableStateOf()` для создания наблюдаемого `MutableState`. Она получает в качестве параметра начальное значение, которое оборачивается в объект `State`, что делает его значение наблюдаемым.

Значение, возвращаемое функцией `mutableStateOf()`:

- Удерживает состояние, которое представляет собой сумму счета.
- Является мутабельным, поэтому значение может быть изменено.
- Является наблюдаемым, поэтому Compose отслеживает любые изменения значения и запускает рекомпозицию для обновления пользовательского интерфейса.

Добавьте состояние стоимости услуг:

- В функции `EditNumberField()` замените ключевое слово `val` перед переменной состояния `amountInput` на ключевое слово `var`:

```
var amountInput = "0"
```

Таким образом, `amountInput` становится мутабельной.

- Используйте тип `MutableState<String>` вместо жестко заданной переменной `String`, чтобы Compose знал, что нужно отслеживать состояние `amountInput`, а затем передавать строку «0», которая является начальным значением по умолчанию для переменной `amountInput state`:

```
import androidx.compose.runtime.MutableState
import androidx.compose.runtime.mutableStateOf

var amountInput: MutableState<String> = mutableStateOf("0")
```

Инициализация `amountInput` также может быть записана таким образом с помощью вывода типов:

```
var amountInput = mutableStateOf("0")
```

Функция `mutableStateOf()` получает в качестве аргумента начальное значение «0», которое затем делает `amountInput` наблюдаемым. Это приводит к появлению предупреждения о компиляции в Android Studio, но вскоре вы это исправите:

Создание объекта состояния во время композиции без использования `remember`.

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

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

Compose отслеживает каждую композицию, которая считывает свойства значения состояния, и запускает перекомпозицию при изменении значения.

Обратный вызов `onValueChange` срабатывает при изменении входных данных текстового поля. В лямбда-выражении переменная `it` содержит новое значение.

В лямбда-выражении именованного параметра `onValueChange` установите свойство `amountInput.value` в 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 текстового поля (это переменная `amountInput`), когда текстовое поле уведомляет вас об изменении текста с помощью функции обратного вызова `onValueChange`.

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

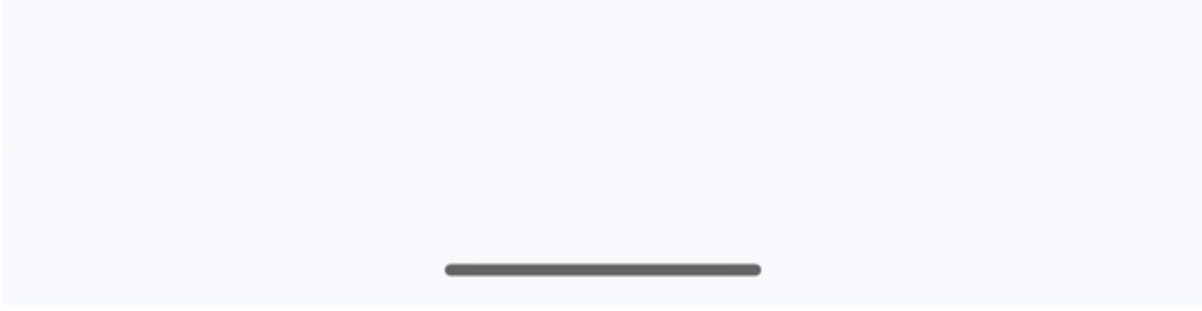
12:00



Calculate Tip

0

**Tip Amount: \$0.00**



Когда пользователь вводит текст в текстовое поле, вызывается обратный вызов `onValueChange`, и переменная `amountInput` обновляется новым значением. Состояние `amountInput` отслеживается `Compose`, поэтому в тот момент, когда ее значение меняется, планируется перекомпозиция и снова выполняется композитная функция `EditNumberField()`. В этой композитной функции переменная `amountInput` возвращается к своему первоначальному значению 0. Таким образом, в текстовом поле отображается значение 0.

С помощью добавленного вами кода изменения состояния приводят к планированию перекомпоновки.

Однако вам нужен способ сохранить значение переменной `amountInput` во время всех рекомпозиций, чтобы она не сбрасывалась в 0 при каждой рекомпозиции функции `EditNumberField()`. Вы решите эту проблему в следующем разделе.

## Используйте функцию `remember` для сохранения состояния

Методы `Composable` могут вызываться много раз из-за рекомпозиции. Если состояние не было сохранено, то при рекомпозиции композит сбрасывает его.

С помощью функции `remember` композитные функции могут сохранять объект во время рекомпозиций. Значение, вычисленное функцией `remember`, сохраняется в `Composition` при начальной композиции, а сохраненное значение возвращается при рекомпозиции. Обычно функции `remember` и `mutableStateOf` используются вместе в композитных функциях, чтобы состояние и его обновления правильно отражались в пользовательском интерфейсе.

Используйте функцию `remember` в функции `EditNumberField()`:

- В функции `EditNumberField()` инициализируйте переменную `amountInput` с помощью делегата свойства `by remember` Kotlin, окружив вызов функции `mutableStateOf()` символом `remember`.

В функции `mutableStateOf()` передайте пустую строку вместо статической строки «0»:

```
var amountInput by remember { mutableStateOf("") }
```

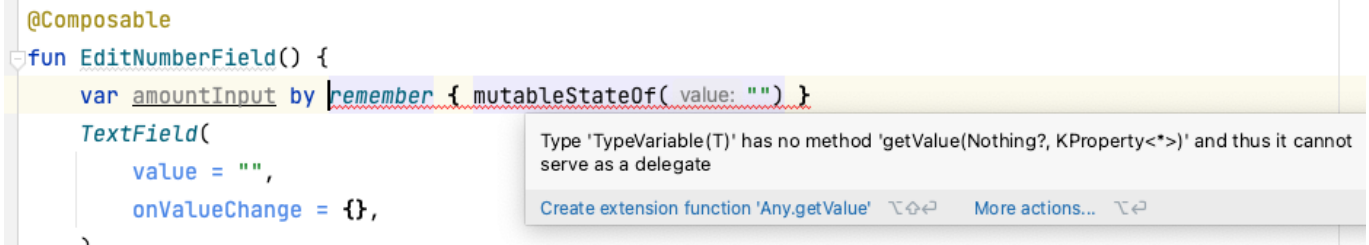
Теперь пустая строка является начальным значением по умолчанию для переменной `amountInput`.

`by` - это делегирование свойств в Kotlin. Функции геттера и сеттера по умолчанию для свойства `amountInput` делегируются функциям геттера и сеттера класса `remember` соответственно.

Импортируйте эти функции:

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

Предупреждение: Этот код может выдать ошибку, которую вы видите на этом изображении, относительно функции расширения `getValue()`:



Если это так, вручную добавьте импорт `getValue` и `setValue` в блок импорта в начале файла, как показано на этом изображении:

```
import androidx.compose.material.Surface
import androidx.compose.material.Text
import androidx.compose.material.TextField
import androidx.compose.runtime.Composable
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.getValue
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.ui.unit.dp
```

Добавление импорта геттера и сеттера делегата позволяет читать и устанавливать `amountInput`, не обращаясь к свойству `value` `MutableState`.

Обновленная функция `EditNumberField()` должна выглядеть следующим образом:

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

Примечание: Во время первоначального создания значение в `TextField` устанавливается на начальное значение, которое является пустой строкой.

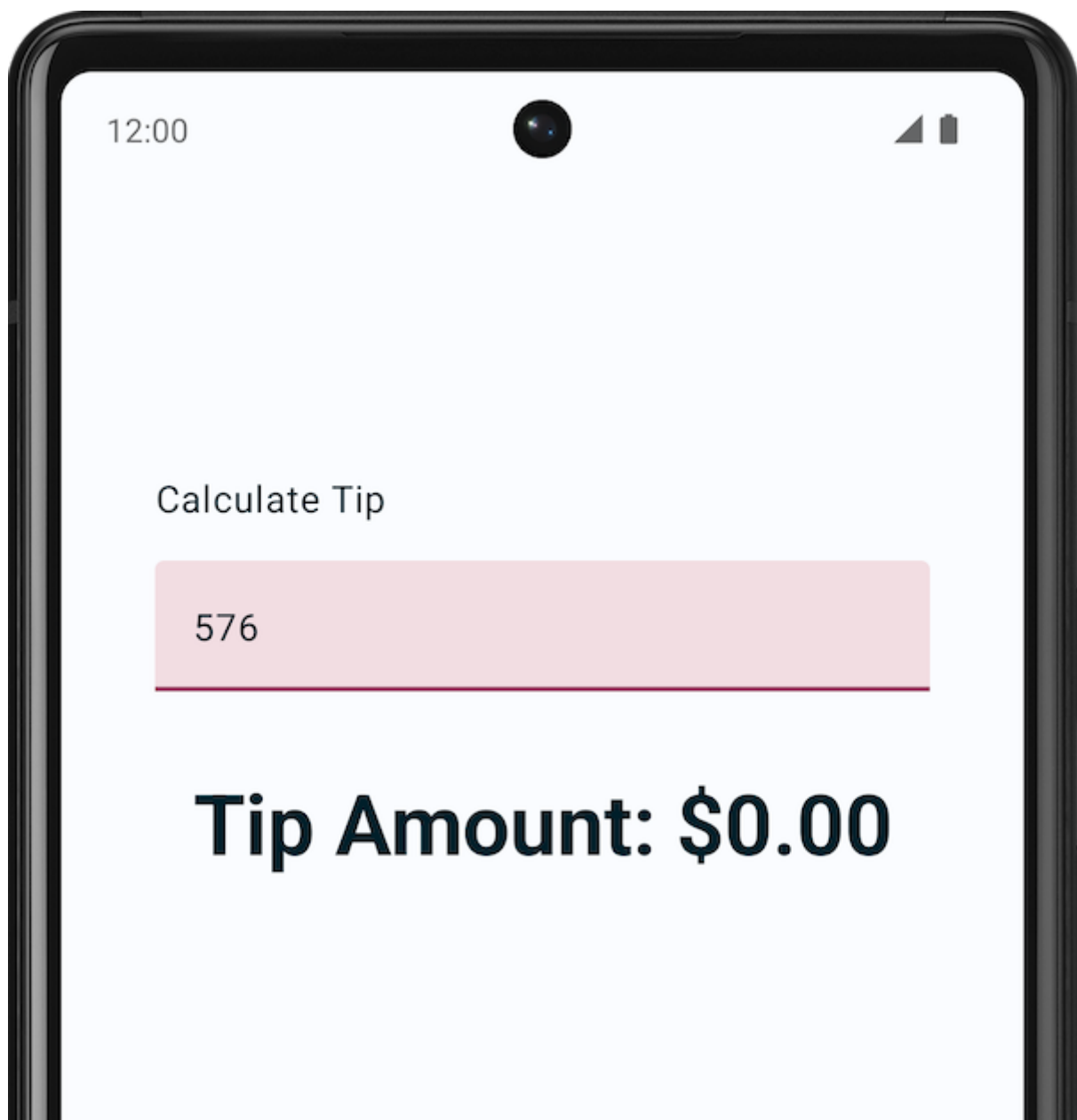
Когда пользователь вводит текст в текстовое поле, вызывается обратный вызов лямбды `onValueChange`, лямбда выполняется, и `amountInput.value` устанавливается в обновленное значение, введенное в текстовое поле.

`amountInput` - это мутабельное состояние, отслеживаемое `Compose`, перекомпозиция которого запланирована.

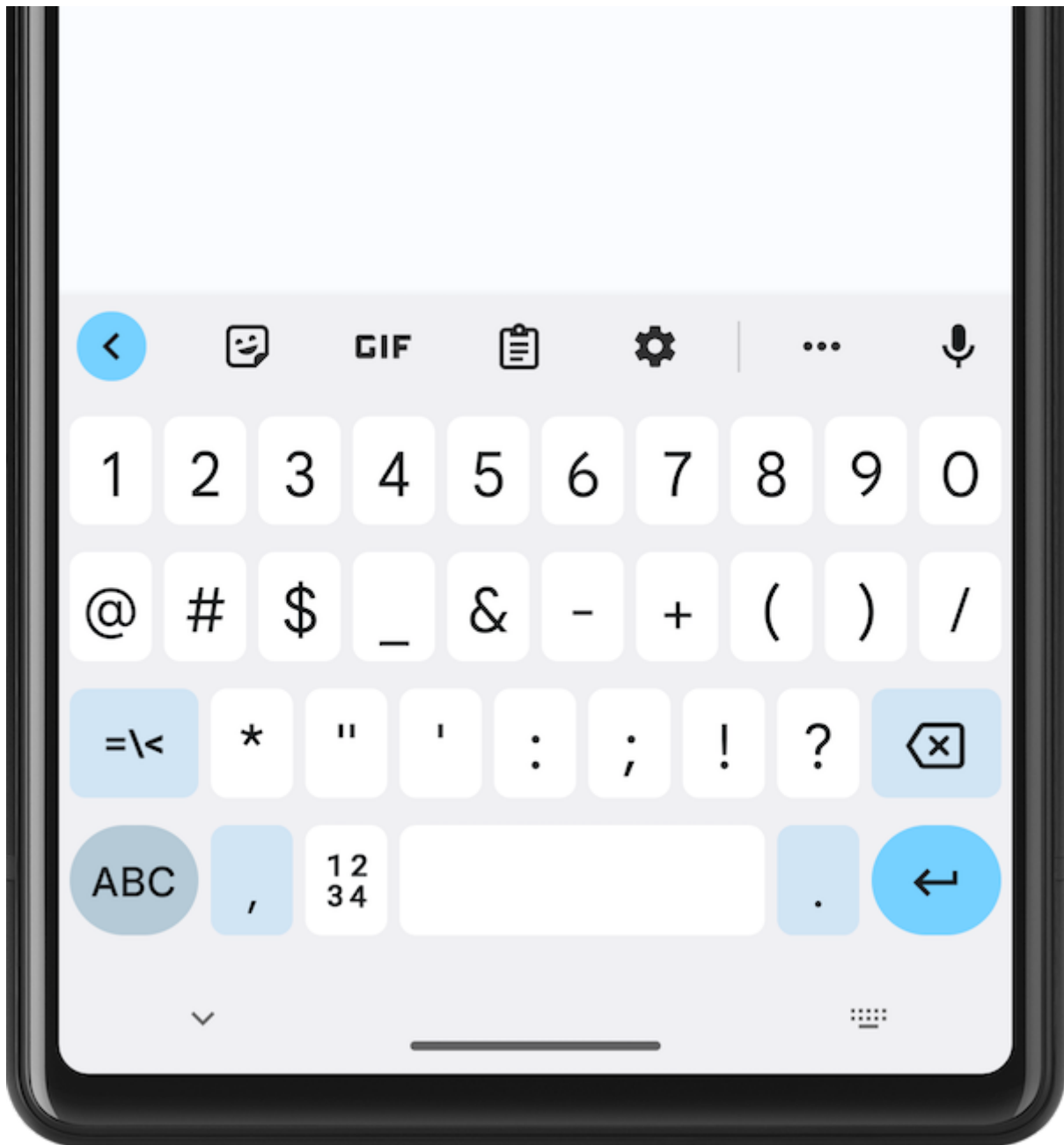
Композитная функция `EditNumberField()` перекомпонуется. Поскольку вы используете запоминание `{ }`, изменение переживет рекомпозицию, и поэтому состояние не будет повторно инициализировано в «».

Значение текстового поля устанавливается в запомненное значение `amountInput`. Текстовое поле перекомпонуется (перерисовывается на экране с новым значением).

- Запустите приложение и введите текст в текстовое поле. Теперь вы должны увидеть набранный текст.







## Композиция и рекомпозиция в действии

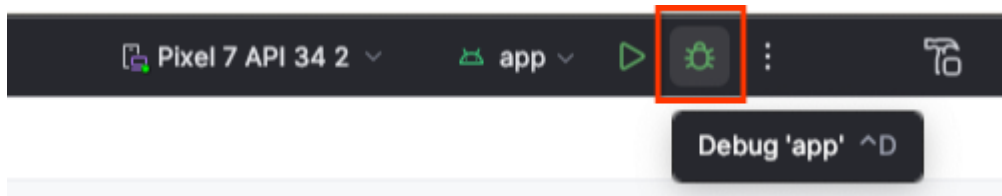
В этом разделе вы установите точку останова и отладите композитную функцию `EditNumberField()`, чтобы увидеть, как работают начальная композиция и рекомпозиция.

Примечание: **Точка останова** - это место в коде, где вы хотите приостановить нормальное выполнение приложения, чтобы выполнить другие действия, например, исследовать переменные, оценить выражения или выполнить код построчно. Вы можете установить точку останова в любой исполняемой строке кода.

- Установите точку останова и отладьте приложение на эмуляторе или устройстве:

В функции `EditNumberField()` рядом с именованным параметром `onValueChange` установите точку останова строки.

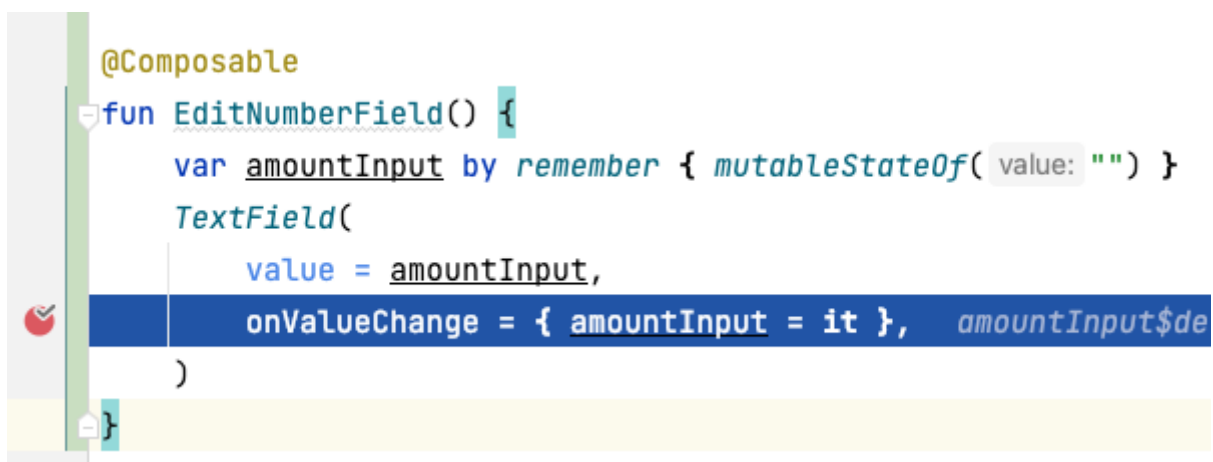
В навигационном меню выберите `Debug 'app'`. Приложение запускается на эмуляторе или устройстве. Выполнение приложения приостанавливается в первый раз, когда создается элемент `TextField`.



- На панели Debug нажмите Resume Program. Создается текстовое поле.

На эмуляторе или устройстве введите букву в текстовое поле. Выполнение вашего приложения снова приостановится, когда оно достигнет установленной вами точки останова. Когда вы вводите текст, вызывается обратный вызов `onValueChanged`. Внутри лямбды `it` находится новое значение, которое вы ввели с клавиатуры.

Как только значение «`it`» присваивается `amountInput`, Compose запускает перекомпоновку с новыми данными, поскольку наблюдаемое значение изменилось.



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



Это состояние текстового поля.

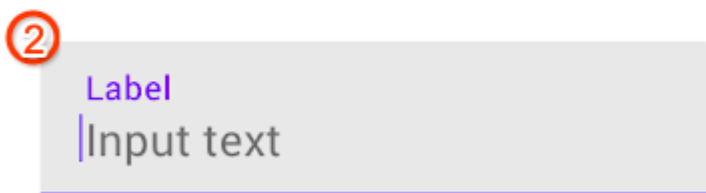
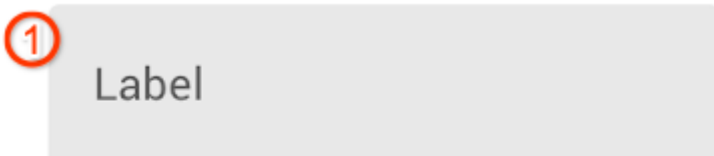
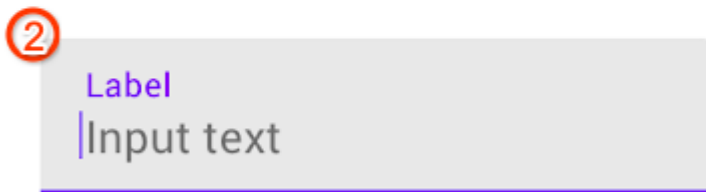
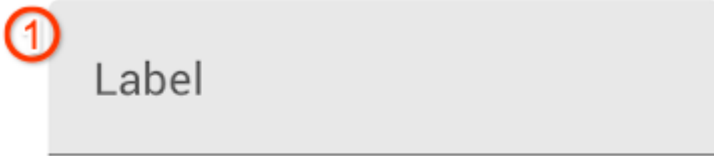
- Нажмите кнопку Возобновить программу. Введенное значение отображается на эмуляторе или устройстве.

## Изменение внешнего вида

В предыдущем разделе вы добились того, что текстовое поле стало работать. В этом разделе вы улучшите пользовательский интерфейс.

- Добавьте Label в текстовое поле

Каждое текстовое поле должно иметь метку, которая дает пользователям понять, какую информацию они могут вводить. В первой части приведенного ниже примера текст метки находится в середине текстового поля и выровнен по линии ввода. Во второй части следующего примера метка перемещается выше в текстовом поле, когда пользователь нажимает на текстовое поле для ввода текста.



Измените функцию `EditNumberField()`, чтобы добавить метку в текстовое поле:

В составной функции `TextField()` функции `EditNumberField()` добавьте метку с именем параметра, установленного в пустое лямбда-выражение:

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

В лямбда-выражении вызовите функцию `Text()`, которая принимает `stringResource(R.string.bill_amount)`:

```
label = { Text(stringResource(R.string.bill_amount)) },
```

В составной функции `TextField()` добавьте именованный параметр `singleLine`, установленный на значение `true`:

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

Это позволяет сократить текстовое поле до одной горизонтально прокручиваемой строки из нескольких строк.

Добавьте набор параметров `keyboardOptions` в `KeyboardOptions()`:

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

В Android есть возможность настроить клавиатуру, отображаемую на экране, для ввода цифр, адресов электронной почты, URL-адресов, паролей и т. д.

Установите тип клавиатуры на цифровую, чтобы вводить цифры. Передайте функции `KeyboardOptions` именованный параметр `keyboardType`, установленный в `KeyboardType.Number`:

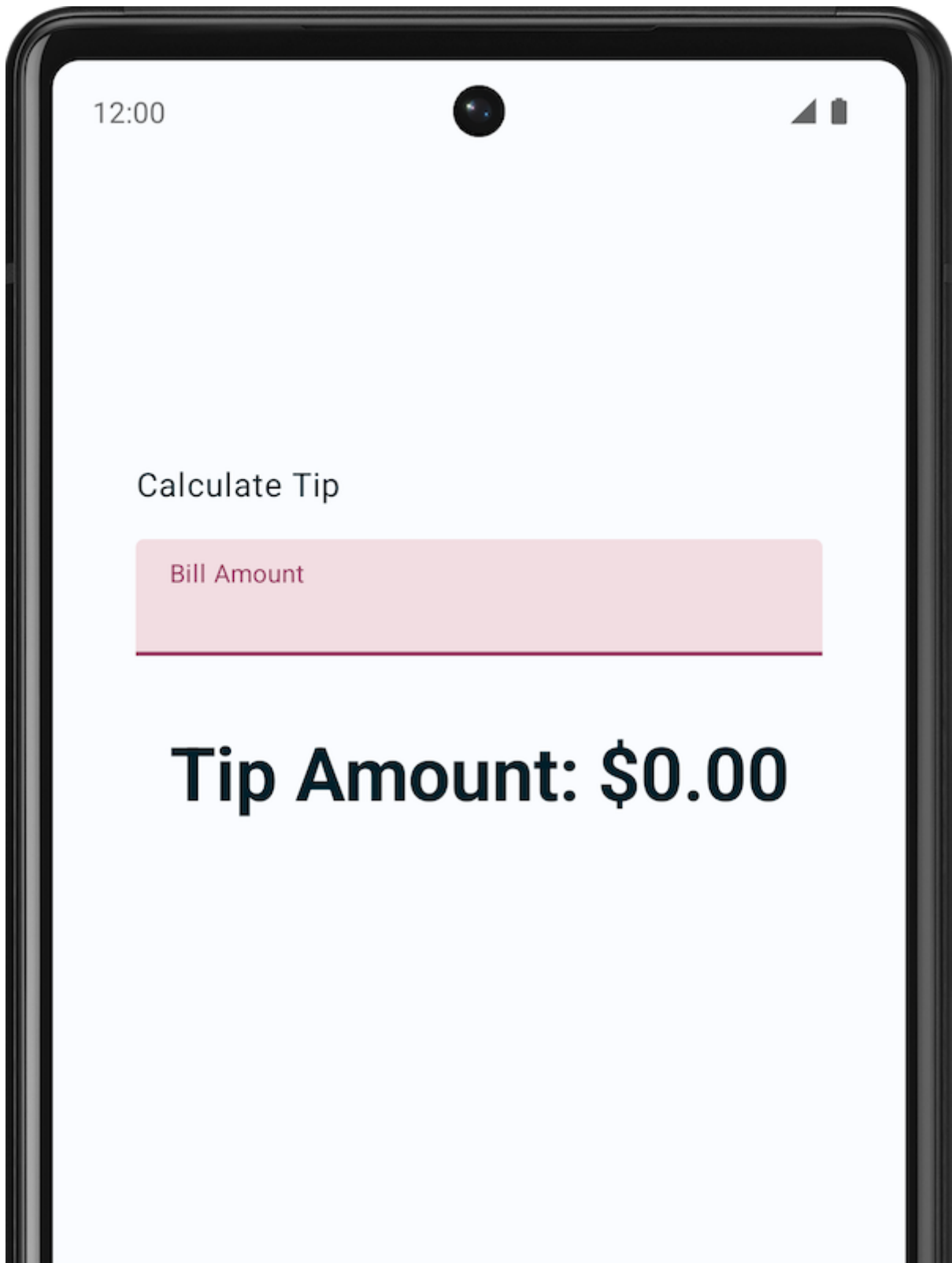
```
import androidx.compose.ui.text.input.KeyboardType  
  
TextField(  
    // ...  
    keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Number),  
)
```

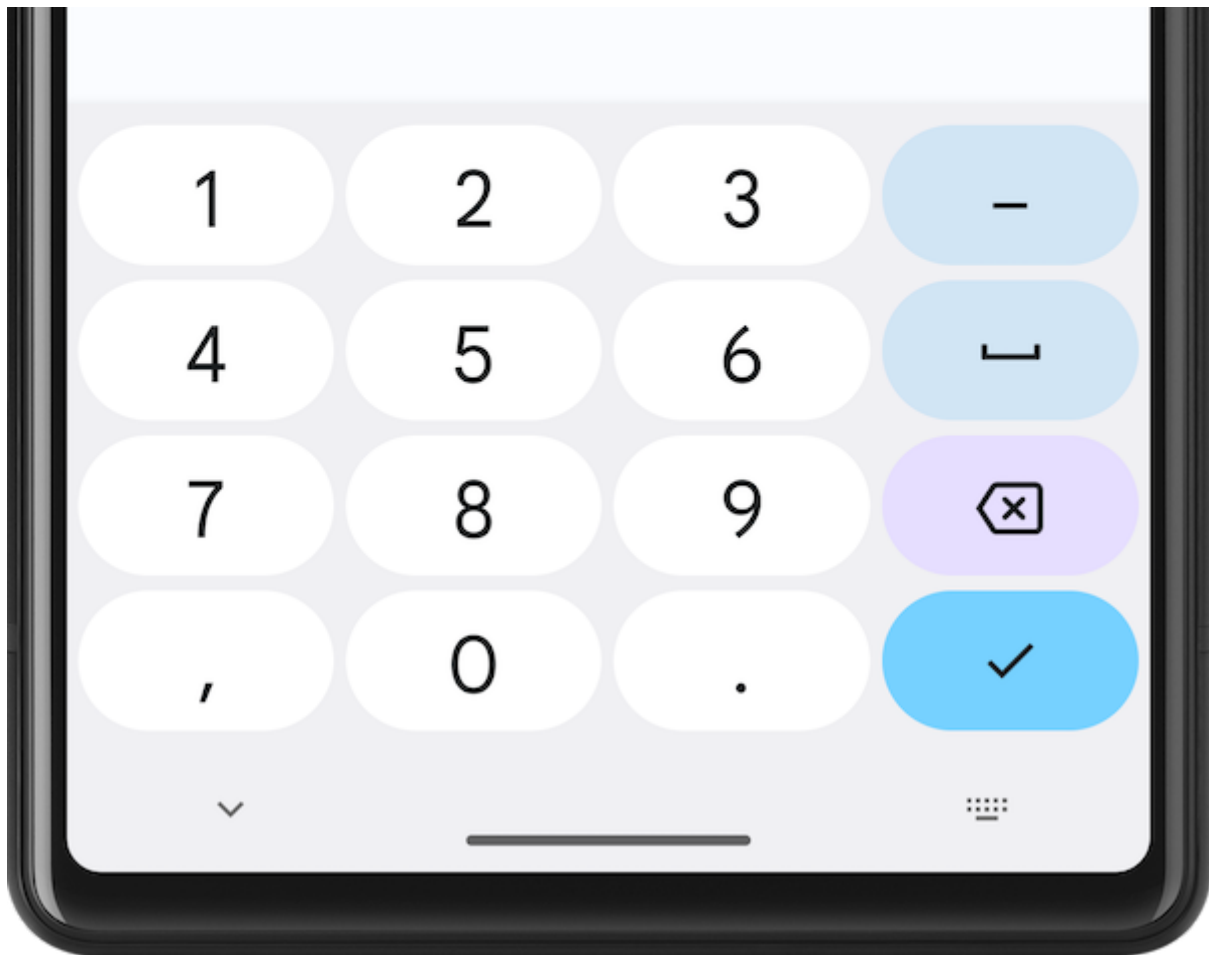
Завершенная функция `EditNumberField()` должна выглядеть так, как показано в этом фрагменте кода:

```
@Composable  
fun EditNumberField(modifier: Modifier = Modifier) {  
    var amountInput by remember { mutableStateOf("") }  
    TextField(  
        text = amountInput,  
        onTextChange = { amountInput = it },  
        keyboardType = KeyboardType.Number,  
        singleLine = true,  
        modifier = modifier
```

```
        value = amountInput,
        onChange = { amountInput = it },
        singleLine = true,
        label = { Text(stringResource(R.string.bill_amount)) },
        keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Number),
        modifier = modifier
    )
}
```

- Запустите приложение. Изменения в клавиатуре можно увидеть на этом снимке экрана:





## Отображение суммы чаевых

В этом разделе вы реализуете основную функциональность приложения - возможность подсчета и отображения суммы чаевых.

В файле `MainActivity.kt` в качестве части стартового кода вам предоставлена приватная функция `calculateTip()`. 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 `NumberFormat` для отображения формата чаевых в виде валюты.

Теперь ваше приложение может рассчитать размер чаевых, но вам все еще нужно отформатировать и отобразить его с помощью класса.

- Используйте функцию `calculateTip()`

Текст, введенный пользователем в текстовое поле `composable`, возвращается в функцию обратного вызова `onValueChange` как строка, хотя пользователь ввел число. Чтобы исправить это, необходимо преобразовать значение `amountInput`, которое содержит сумму, введенную пользователем.

В составной функции `EditNumberField()` создайте новую переменную `amount` после определения `amountInput`. Вызовите функцию `toDoubleOrNull` для переменной `amountInput`, чтобы преобразовать строку в `Double`:

```
val amount = amountInput.toDoubleOrNull()
```

Функция `toDoubleOrNull()` - это предопределенная функция Kotlin, которая разбирает строку как число `Double` и возвращает результат или `null`, если строка не является корректным представлением числа.

В конце оператора добавьте оператор `?:` Elvis, который возвращает значение `0.0`, если `amountInput` равно `null`:

```
val amount = amountInput.toDoubleOrNull() ?: 0.0
```

Примечание: Оператор `?:` Elvis возвращает выражение, которое предшествует ему, если значение не равно `null`, и выражение, которое следует за ним, если значение равно `null`. Он позволяет писать этот код более идиоматично.

После переменной `amount` создайте еще одну переменную `val` под названием `tip`. Инициализируйте ее с помощью `calculateTip()`, передав параметр `amount`.

```
val tip = calculateTip(amount)
```

Функция `EditNumberField()` должна выглядеть так, как показано в этом фрагменте кода:

```
@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)
    )
}
```

## Отображение рассчитанной суммы чаевых

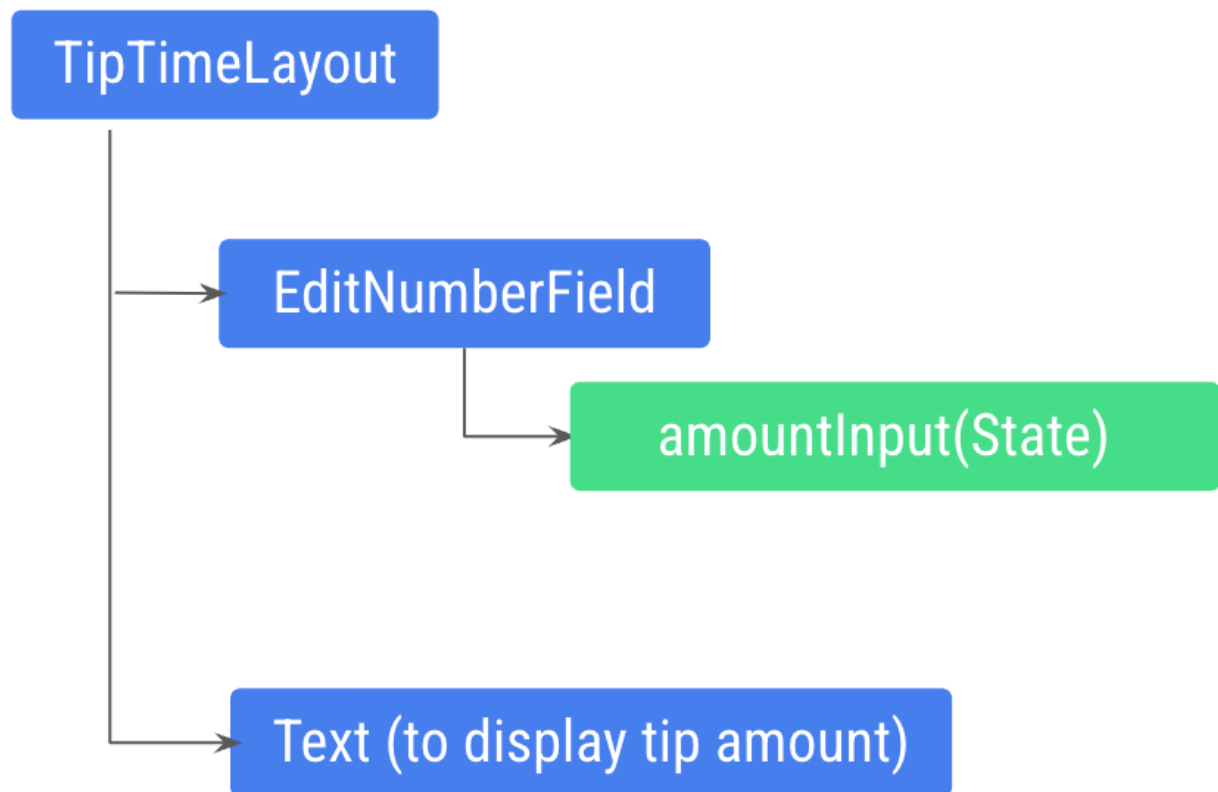
Вы написали функцию для расчета суммы чаевых, следующий шаг - отобразить рассчитанную сумму чаевых:

В функции `TipTimeLayout()` в конце блока `Column()` обратите внимание на текстовый компонент, 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
        )
        // ...
    }
}
```

Чтобы рассчитать и отобразить сумму чаевых, нужно обратиться к переменной `amountInput` в функции `TipTimeLayout()`, но переменная `amountInput` - это состояние текстового поля, определенного в составной функции `EditNumberField()`, поэтому ее пока нельзя вызвать из функции `TipTimeLayout()`. Это изображение иллюстрирует структуру кода:

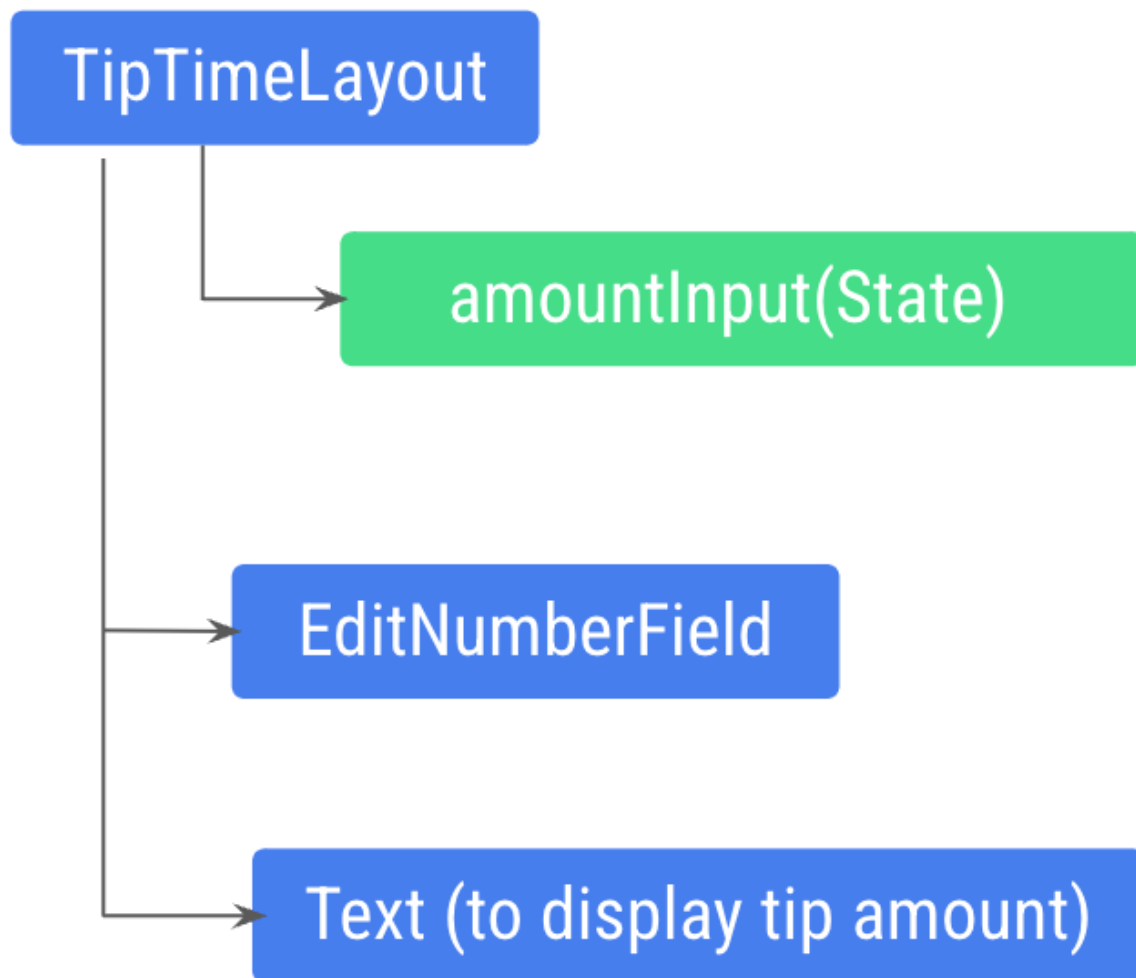





■ State (remembered value)

■ App defined/pre-defined composables

Эта структура не позволит вам отобразить сумму чаевых в новом составном элементе Text, потому что составному элементу Text необходимо получить доступ к переменной amount, вычисленной из переменной amountInput. Вам нужно раскрыть переменную amount для функции TipTimeLayout(). Это изображение иллюстрирует желаемую структуру кода, которая делает композит EditNumberField() нестационарным:



 State (remembered value)

 App defined/pre-defined composables

Этот паттерн называется подъемом состояния. В следующем разделе вы поднимете, или снимите, состояние с композита, чтобы сделать его нестационарным.

Примечание: композит без состояния(stateless) - это композит, который не хранит свое собственное состояние. Он отображает любое состояние, переданное ему в качестве входных аргументов.

## Подъем состояния

В этом разделе вы узнаете, как решить, где определить состояние так, чтобы можно было повторно использовать и совместно использовать композитные функции.

В композитной функции вы можете определить переменные, которые содержат состояние для отображения в пользовательском интерфейсе. Например, вы определили переменную `amountInput` как состояние в композитной функции `EditNumberField()`.

Когда ваше приложение станет более сложным и другим составным функциям понадобится доступ к состоянию в составной функции `EditNumberField()`, вам нужно будет рассмотреть возможность подъема, или извлечения, состояния из составной функции `EditNumberField()`.

### Понимание различий между композитными функциями с состоянием (statefull) и композитными функциями без состояния (stateless)

Вы должны поднимать состояние, когда это необходимо:

Разделение состояния с несколькими композитными функциями.

Создать композитную функцию без состояния, которая может быть использована повторно в вашем приложении.

Когда вы извлекаете состояние из композитной функции, результирующая композитная функция называется **stateless**. То есть композитные функции можно сделать *stateless*, извлекая из них состояние.

Композитная функция без состояния - это композитная функция, у которой нет состояния, то есть она не хранит, не определяет и не модифицирует новое состояние. С другой стороны, композит с состоянием - это композит, который владеет частью состояния, которая может меняться со временем.

Примечание: В реальных приложениях может быть сложно добиться 100% отсутствия состояния у композита в зависимости от его обязанностей. Вы должны проектировать свои композиты таким образом, чтобы они владели как можно меньшим количеством состояния и позволяли поднимать состояние, когда это имеет смысл, раскрывая его в API композита.

**Поднятие состояния** - это паттерн передачи состояния вызывающему компоненту, чтобы сделать компонент без состояния.

В применении к компокуемым компонентам это часто означает введение двух параметров в компокуемый компонент:

A value: T, который является текущим значением для отображения.

Лямбда обратного вызова `onValueChange: (T) -> Unit`, которая срабатывает при изменении значения, чтобы состояние могло быть обновлено в другом месте, например, когда пользователь вводит текст в текстовое поле.

Поднимите состояние в функции `EditNumberField()`:

- 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` - тип `(String) -> Unit`, то есть это функция, которая принимает на вход значение `String` и не имеет возвращаемого значения. Параметр `onValueChange` используется в качестве обратного вызова `onValueChange`, передаваемого в составное поле `TextField`.

Примечание: Лучшей практикой является предоставление параметра `Modifier` по умолчанию всем композитным функциям, что повышает возможность повторного использования. Вы должны добавить его в качестве первого необязательного параметра после всех обязательных параметров.

В функции `EditNumberField()` обновите составную функцию `TextField()`, чтобы она использовала переданные параметры:

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

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(
        //...
    ) {
        //...
    }
}
```

- Вы передали состояние в `TipTimeLayout()`, теперь передайте его в `EditNumberField()`. В функции `TipTimeLayout()` обновите вызов функции `EditNumberField()`, чтобы использовать поднятое состояние:

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

Это делает `EditNumberField` не имеющим состояния. Вы передали состояние пользовательского интерфейса его предку, `TipTimeLayout()`. Теперь `TipTimeLayout()` является владельцем `state(amountInput)`.

## Позиционное форматирование (интерполяция)

Позиционное форматирование используется для отображения динамического содержимого в строках. Например, представьте, что вы хотите, чтобы в текстовом поле «Сумма чаевых» отображалось значение `xx.xx`, которое может быть любой суммой, рассчитанной и отформатированной в вашей функции. Для этого в файле `strings.xml` нужно определить строковый ресурс с аргументом `placeholder`, 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 композиции можно использовать несколько аргументов-заместителей любого типа. Строковым заполнителем является `%s`.

Обратите внимание на композицию текста в `TipTimeLayout()`, 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  
)
```

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

```
Text(  
    text = stringResource(R.string.tip_amount, tip),  
    // ...
```

Готовые функции `TipTimeLayout()` и `EditNumberField()` должны выглядеть так, как показано в этом фрагменте кода:

```
@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))
}
}

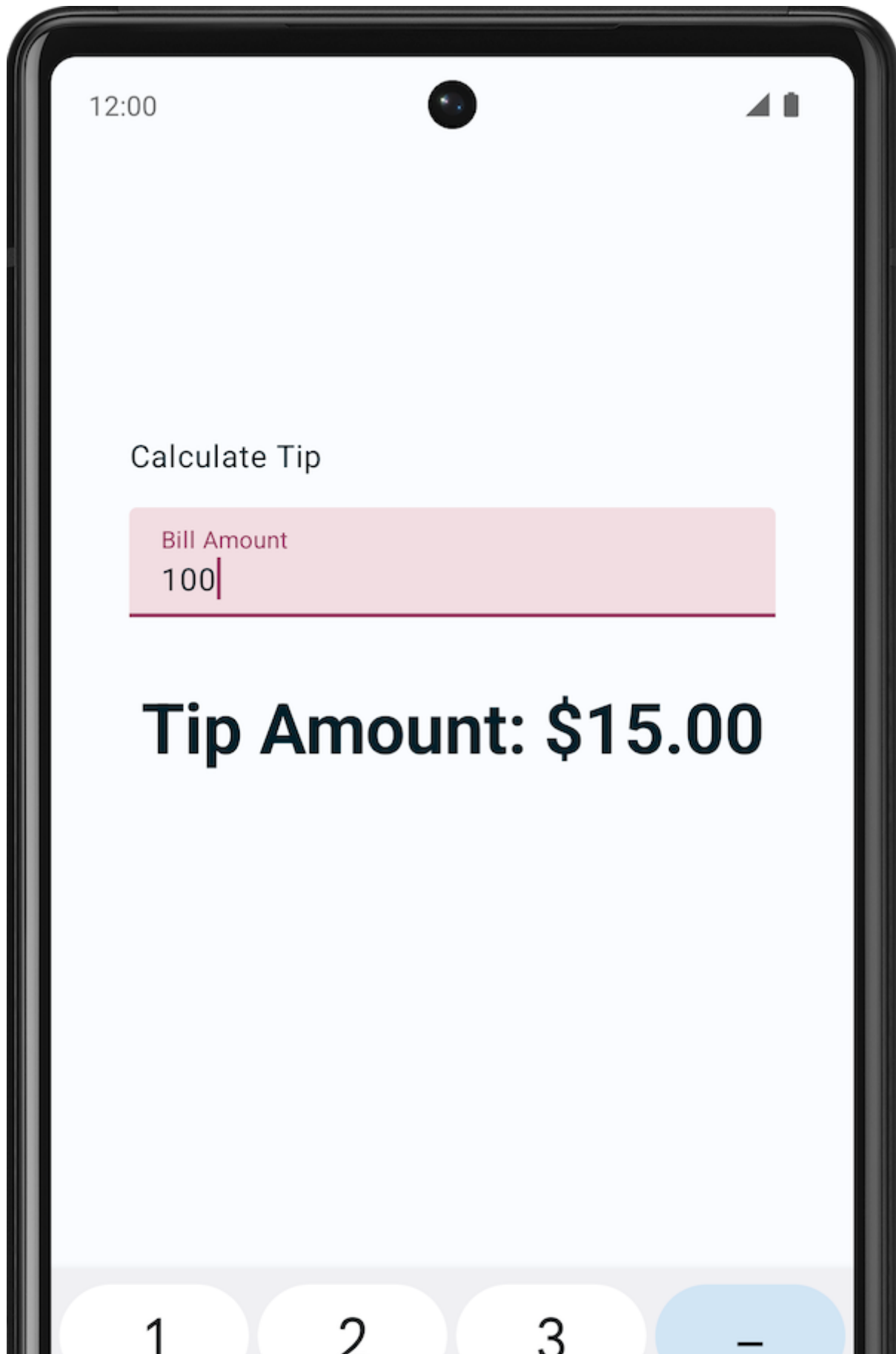
@Composable
fun EditNumberField(
    value: String,
    onValueChange: (String) -> Unit,
    modifier: Modifier = Modifier
) {
    TextField(
        value = value,
        onValueChange = onValueChange,
        singleLine = true,
        label = { Text(stringResource(R.string.bill_amount)) },
        keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Number),
        modifier = modifier
    )
}

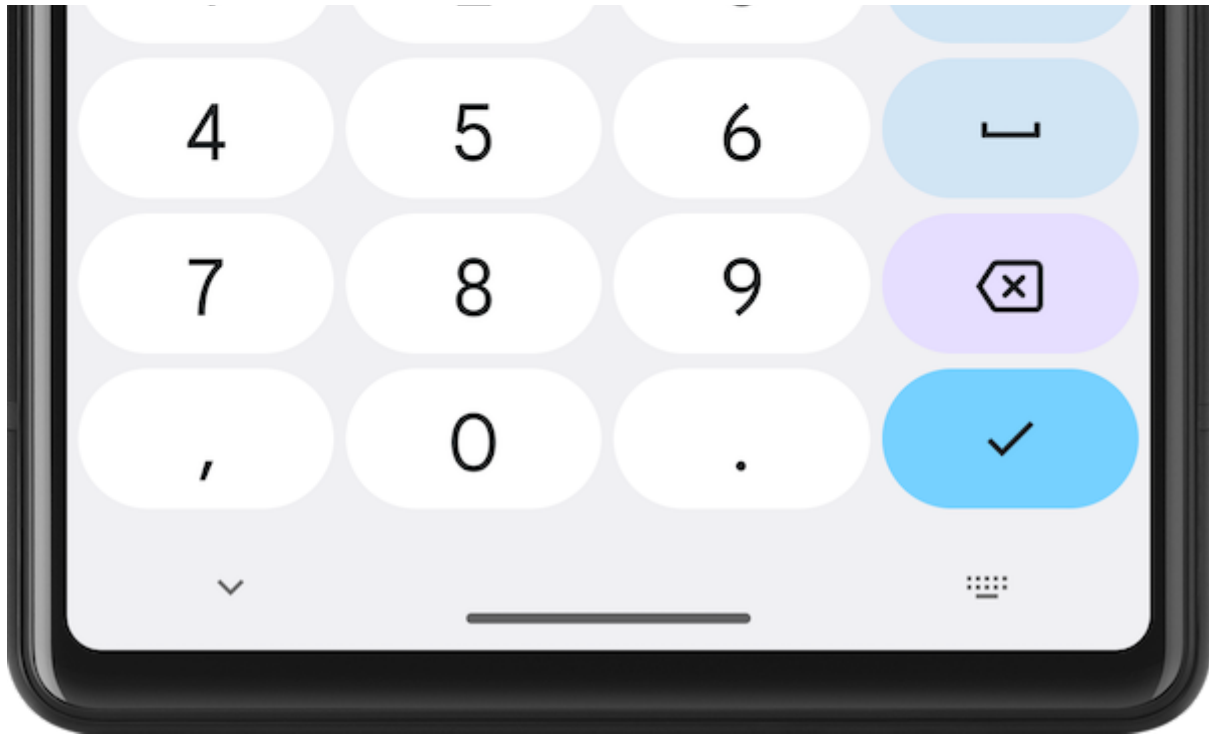
```

Подводя итог, можно сказать, что вы перенесли состояние `amountInput` из функции `EditNumberField()` в составную функцию `TipTimeLayout()`. Чтобы текстовое поле работало как прежде, нужно передать два аргумента в составную функцию `EditNumberField()`: значение `amountInput` и обратный вызов лямбда-функции, которая обновляет значение `amountInput` на основе пользовательского ввода. Эти изменения

позволяют вычислить чаевые из свойства `amountInput` в `TipTimeLayout()`, чтобы отобразить их пользователю.

Запустите приложение на эмуляторе или устройстве, а затем введите значение в текстовое поле «Сумма счета». Как показано на этом изображении, сумма чаевых составляет 15 процентов от суммы счета:





## Заключение

Вы завершили этот урок и узнали, как использовать состояние в приложении Compose!

## Резюме

- **Состояние в приложении** - это любое значение, которое может изменяться с течением времени.
- **Композиция** - это описание пользовательского интерфейса, создаваемого Compose при выполнении композиций. Приложения Compose вызывают композиционные функции для преобразования данных в пользовательский интерфейс.
- **Начальная композиция** - это создание пользовательского интерфейса при первом выполнении Compose композитных функций.
- **Рекомпозиция** - это процесс повторного выполнения тех же композитных функций для обновления дерева при изменении данных.
- **Передача состояния** (state hoisting) - это паттерн передачи состояния вызывающей стороне, чтобы сделать компонент нестационарным.