

**CURSOS  
TÉCNICOS**

**DESENVOLVIMENTO DE  
APLICATIVOS I**

Eixo Informática para Internet

**Unidade 5**

## **SUMÁRIO**

UNIDADE 5	
1. CONFIGURANDO A UI POTENCIALIZADA PELO COMPOSE	3
1.1. O que vamos trabalhar?	3
1.2. A Composição	3
1.3. Obtendo e entendendo o projeto	4
1.4. Trabalhando com TextField para receber dados do usuário	7
1.5. A função remember e o state	9
1.6. Tratando a acessibilidade do App	11
1.7. Criando a lógica da calculadora em uma função privada	13
1.8. Configurando a função calculateTip e aplicando toDoubleOrNull:	14
1.9. Elevando states	15
1.10. Review do conteúdo desta unidade	19
1.11. Leituras e estudos complementares	20
2. Referências	21

## **UNIDADE 5**

### **1. CONFIGURANDO A UI POTENCIALIZADA PELO COMPOSE**

Nas unidades anteriores, nós trabalhamos bastante com o Compose (JetPack Compose) e a sua importância na agilização de desenvolvimento de interfaces para mobile. Salientando, sempre, que a abordagem deste módulo está atendendo às especificações de desenvolvimento de mercado atuais, onde o Compose assumiu um papel de quase protagonista na construção de interfaces destas aplicações.

Estudamos, também, a introdução dos States e as interações dos usuários na alteração dos valores nos componentes renderizados em composição na tela do App.

#### **1.1. O que vamos trabalhar?**

Nesta unidade vamos, primeiro, criar uma calculadora com os seguintes tópicos de desenvolvimento:

- **Conceito de Composição**
- Usar, avaliar e gerenciar o **estado** de componentes ao utilizar o Compose
- **Elevação de estados** e sua relevância
- Um elemento combinável **TextField** para inserir e editar texto.
- Um elemento combinável **Text** para mostrar texto
- Um elemento combinável **Spacer** para mostrar espaços vazios entre os elementos da interface.
- Ainda com propriedades do **Modifier** → alinhamentos, espaçamentos, fonte e cores
- **Funções privadas**
- **Função toDoubleOrNull()**
- **Condicionais** → If

#### **1.2. A Composição**

De acordo com a documentação do [Android Developers](#) → **todos os elementos combináveis em um App, descrevem uma interface que pode incluir uma coluna contendo texto, um espaçador e uma caixa de texto, uma imagem, e até mesmo um botão forma o contexto da composição.**

A composição é uma descrição da interface gerada pelo Compose quando executa os elementos combináveis. Os Apps Compose utilizam funções combináveis para transformar dados em elementos da interface e interativos para o usuário. **Se ocorrer uma alteração de estado, o Compose executará novamente as funções combináveis afetadas com o novo estado, resultando em uma interface atualizada. Esse processo é chamado de recomposição, e o Compose o gerencia automaticamente.**

Quando o Compose executa os elementos combináveis durante a composição inicial, ele mantém um rastreamento dos elementos chamados para descrever a interface. A recomposição ocorre quando o Compose reexecuta os elementos de composição, que foram alterados devido a mudanças nos dados, atualizando, assim, a composição para refletir essas alterações.

**A composição só pode ser gerada, inicialmente, por uma composição inicial e atualizada por meio da recomposição. A recomposição é a única maneira de modificar a composição.**

No Compose, usamos os tipos "State" e "MutableState" para tornar o estado do aplicativo observável ou rastreável pelo Compose. **O tipo "State" é imutável, o que significa que seu valor é apenas lido, enquanto o tipo "MutableState" é mutável. Para criar um "MutableState" observável, pode-se empregar a função "mutableStateOf()". Ela aceita um valor inicial como parâmetro, encapsulado por um objeto "State", o que torna seu valor observável.**

### ***1.3. Obtendo e entendendo o projeto***

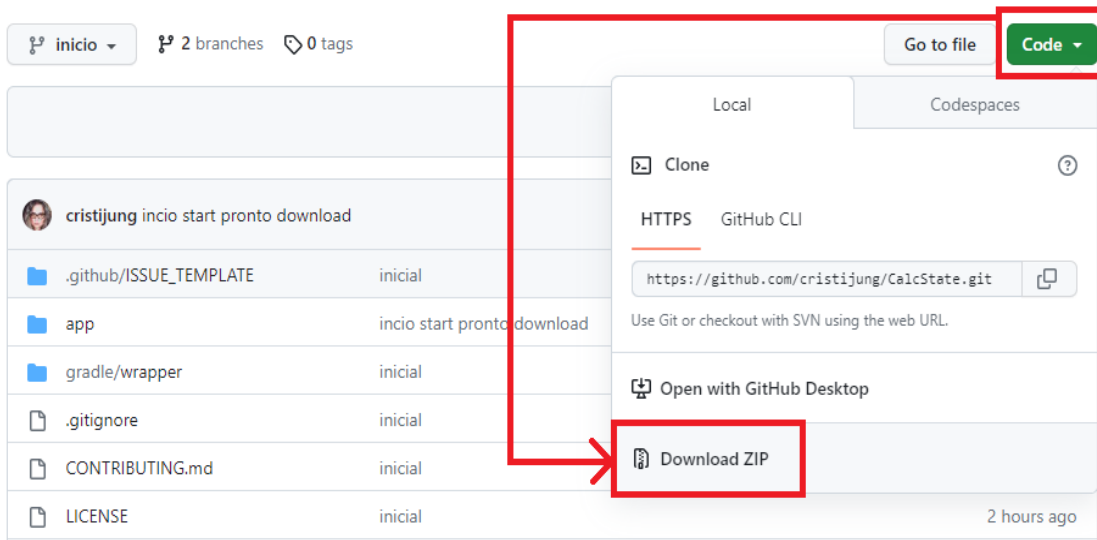
Neste tópico, vamos focar em vários conceitos, portanto está sendo disponibilizado um projeto inicial, com um pré código com suas funções.

**Lembrando que este código disponibilizado é adaptado da documentação [Desenvolvedores Android](#) e seu link está em referência no readme do repositório abaixo.**

**1. Clique no link abaixo:**

Trabalhando com código inicial → neste link:  
<https://github.com/cristijung/CalcState/tree/inicio>

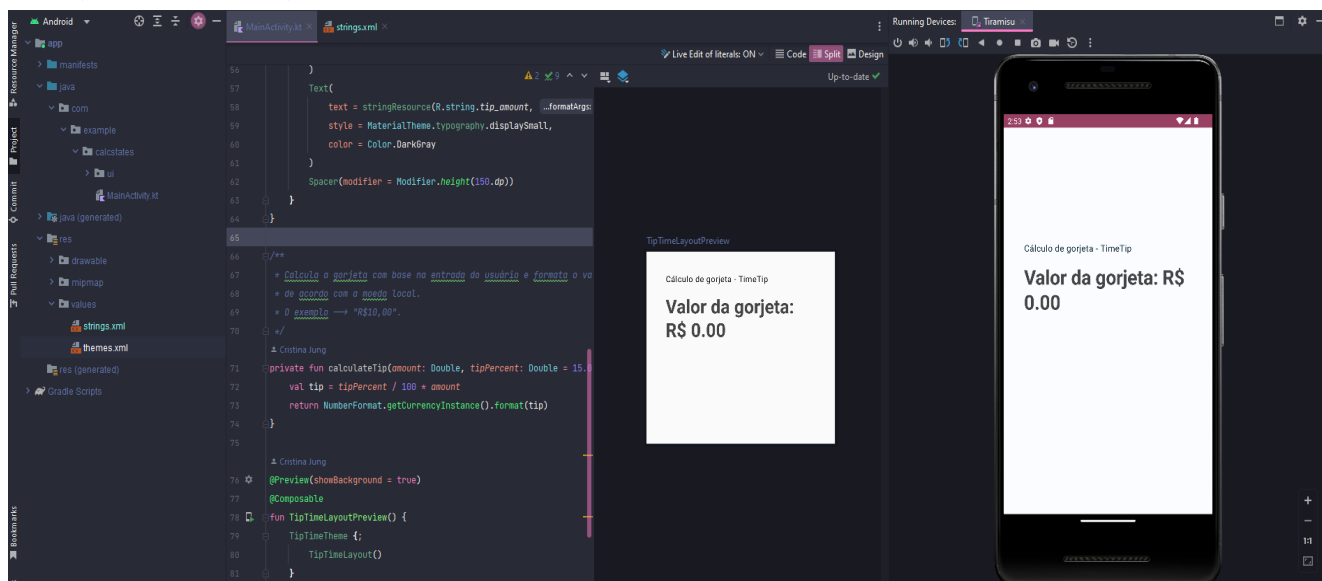
**2. No repositório faça download compactado dele, clicando no botão conforme a imagem:**



Fonte da imagem: autoria própria

**3. Descompacte a pasta e abra o projeto no Android Studio.**

**4. Execute o projeto, você encontrará a visualização do modo Split e o emulador (se executá-lo) da forma como está sendo ilustrada na próxima imagem:**



Fonte da imagem: de própria autoria

Vamos observar atentamente a imagem acima, os elementos combináveis no app descrevem uma interface que mostra uma coluna com texto. A composição do layout exhibe dois componentes de texto:

- ✓ Um deles para um marcador;
- ✓ Outro para mostrar o valor da gorjeta.

### Analizando o que já temos declarado no código:



A composição do layout

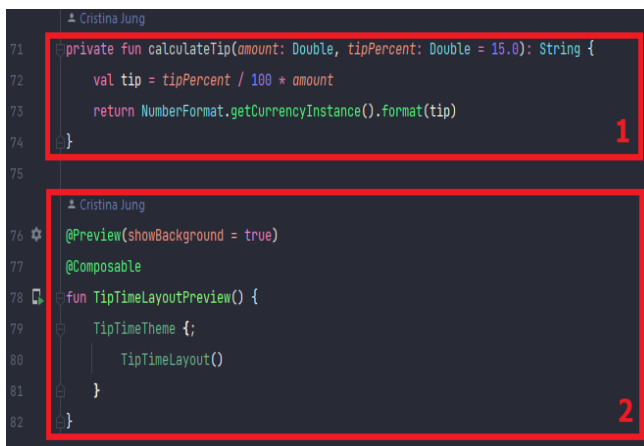
A composição do *layout* por meio da função: `TipTimeLayout` juntamente com uma coluna e dois componentes de `Text`.

Fonte imagem: autoria própria



Fonte imagem: autoria própria

O dado que está sendo exibido no Design é a definição que vamos adicionar no arquivo `strings.xml` dentro da pasta `res`. O texto será exibido pela classe `R` que já estudamos nas unidades anteriores.



1 - Função que calcula a gorjeta, com base na entrada do usuário e formata o valor da gorjeta, de acordo com a moeda local.

2 - É o Preview do nosso App, onde criamos uma função que invoca o tema da aplicação, adicionando a função principal do aplicativo.

#### 1.4. Trabalhando com TextField para receber dados do usuário

O TextField é um componente da Jetpack Compose. O TextField é usado para coletar entradas de texto dos usuários, permitindo que eles insiram dados de texto, como nomes, senhas, mensagens etc. **Ele substitui o widget EditText da abordagem tradicional de desenvolvimento de IU no Android.**

O **TextField no Compose** é parte da biblioteca **androidx.compose.material**, que fornece componentes de **UI de Material Design** prontos para uso. Com o TextField, podemos criar campos de entrada de texto interativos e personalizáveis, incluindo a adição de rótulos, dicas, tratamento de erros, estilos e muito mais. Ele precisa de importação → **import androidx.compose.material3.TextField**

Sintaxe base do TextField:

```

TextField(
    value = "",
    onChange = {},
    modifier = modifier
)
  
```

Fonte imagem: autoria própria

**Quanto aos parâmetros que estamos passando:**

O parâmetro **value** é uma caixa de texto que mostra o valor da string que você passa aqui.

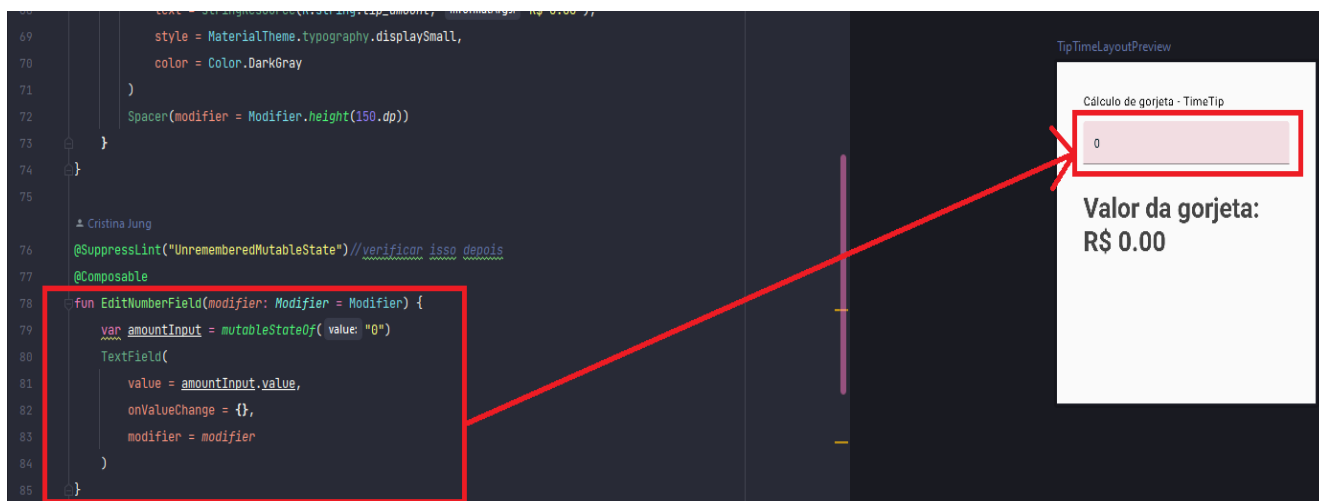
O parâmetro **onChange** é o callback lambda, acionado quando o usuário insere

texto na caixa.

Na continuidade, iremos declarar os valores para os parâmetros do Text Field

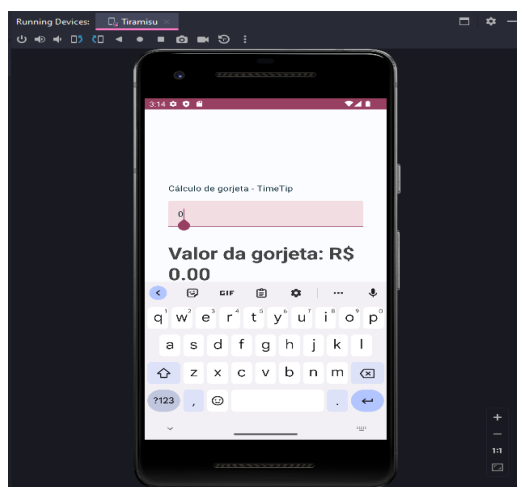
```
fun EditNumberField(modifier: Modifier = Modifier) {
    var amountInput = mutableStateOf("0")
    TextField(
        value = amountInput.value,
        onValueChange = {},
        modifier = modifier
    )
}
```

Observe a próxima imagem:



Fonte imagem: autoria própria

Executando o projeto, nós veremos a seguinte imagem é importante: se você tentar adicionar qualquer dígito ou caractere, verá que nada acontece, o zero continua ali, no TextField.



Fonte imagem: autoria própria



Quando o usuário digita na caixa de texto, **o callback `onValueChange` é chamado, e a variável `amountInput` é atualizada com o novo valor.** O estado `amountInput` é acompanhado pelo `Compose`. Assim, quando o valor muda, a recomposição é programada, e a função de composição `EditNumberField()` é executada novamente. Nesta função de composição, a variável `amountInput` é redefinida para o valor inicial de 0. Assim, a caixa de texto mostra um valor 0.

Com o código adicionado, as mudanças de estado fazem com que **as recomposições sejam programadas. A recomposição refere-se ao processo pelo qual o `Compose` reavalia e atualiza somente as partes da interface do usuário, que realmente mudaram.**

Quando algum estado interno utilizado na interface do usuário é atualizado, somente as partes afetadas são recompostas, evitando, assim, a necessidade de atualizar a UI inteira. Isso resulta em uma renderização mais eficiente, menor consumo de recursos e uma experiência de desenvolvimento mais fluida.

Voltando à variável `amountInput`, é necessário preservar o valor desta variável nas recomposições, para que ela não seja redefinida como 0 sempre que a função `EditNumberField()` é recomposta. É a próxima etapa para declararmos e corrigir esta funcionalidade.

### ***1.5. A função `remember` e o `state`***

Os métodos de construção têm a capacidade de serem invocados, repetidamente, graças à técnica de recomposição ou renderização. Se não for preservado, o elemento de construção irá redefinir seu estado, sempre que ocorrer uma recomposição.

As funções de composição têm a capacidade de preservar um objeto entre diferentes recomposições, usando a função **"remember"**. O valor calculado, por meio da função "remember", é mantido na composição durante a fase de construção inicial e é recuperado nas subsequentes recomposições.

**Em geral, as funções "remember" e "mutableStateOf" são, frequentemente, usadas em conjunto dentro de funções compostas, garantindo que o estado e suas atualizações sejam adequadamente refletidos na interface do usuário.**

Depois desta reflexão, nossa função `EditNumberField()` atualizada irá ficar assim:

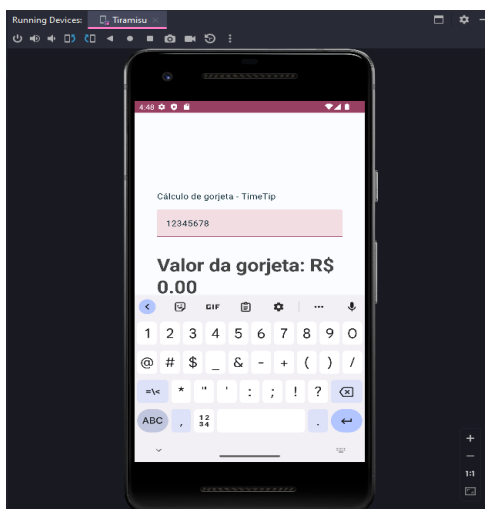
```

80  @Composable
81  fun EditNumberField(modifier: Modifier = Modifier) {
82      var amountInput by remember { mutableStateOf( value: "" ) }
83      TextField(
84          value = amountInput,
85          onChange = { amountInput = it },
86          modifier = modifier
87      )
88  }
    
```

Fonte imagem: autoria própria

**Importante salientarmos aqui que:**

- Durante o estágio inicial de construção, o valor no **TextField** é **inicializado como um valor vazio, representado por uma string vazia**.
- Quando o **usuário insere texto no campo**, o **callback lambda onChange** é **ativado**. Esse lambda é executado, e o atributo **amountInput.value** é **atualizado com o novo valor inserido** (recomposição da interface).
- O **amountInput** representa o **estado mutável que está sendo monitorado pelo mecanismo de composição do Compose**. A recomposição é agendada, o que significa que a função **EditNumberField()** é **reavaliada**. **Graças ao uso de remember { }, as alterações persistem através da recomposição**. **Consequentemente, o estado não é resetado para uma string vazia**.
- O valor exibido no campo de texto é configurado para refletir o valor lembrado de **amountInput**. Como resultado, o campo de texto passa por uma recomposição e é redesenhado na tela, exibindo o novo valor atualizado.



Fonte imagem: autoria própria

### 1.6. Tratando a acessibilidade do App

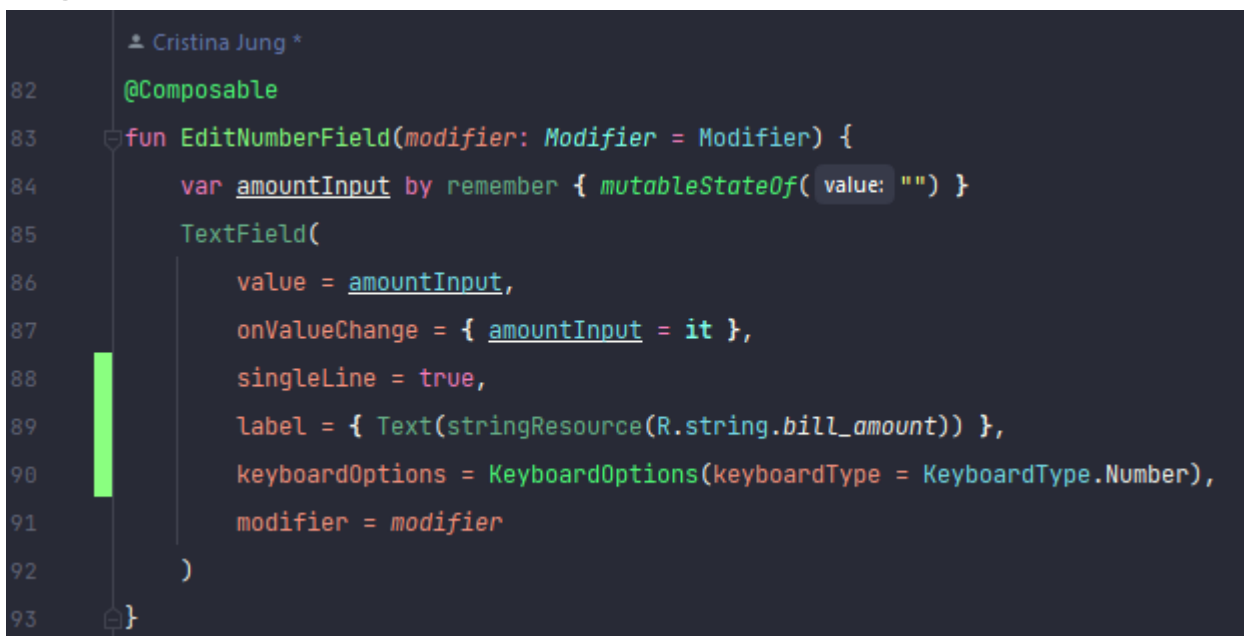
A acessibilidade é um conceito de extrema importância quando desenvolvemos alguma aplicação, seja ela usada no privado como uma intranet, uma aplicação web ou mesmo um app mobile. Primeiramente, é importante entender que acessibilidade não é apenas para pessoas com deficiência. Ela significa pensar na inclusão de todos, em qualquer situação, em qualquer momento em que precise acessar.

O desenvolvimento de uma aplicação, que leva em conta a importância da acessibilidade tem um resultado comercial efetivo, pois encontra resposta adequada no usuário e, uma das primeiras etapas quando pensamos em acessibilidade é inserir informações relevantes para o usuário.

Um dos recursos mais utilizados é o elemento Label, que exibe para o usuário o que um campo de formulário faz.

#### Inserindo um Label

Vamos usar o rótulo, para exibir a especificação do campo de texto. Observe a próxima imagem, e leia atentamente qual a responsabilidade de cada propriedade:



```

82  @Composable
83  fun EditNumberField(modifier: Modifier = Modifier) {
84      var amountInput by remember { mutableStateOf( value: "") }
85      TextField(
86          value = amountInput,
87          onChange = { amountInput = it },
88          singleLine = true,
89          label = { Text(stringResource(R.string.bill_amount)) },
90          keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Number),
91          modifier = modifier
92      )
93  }
  
```

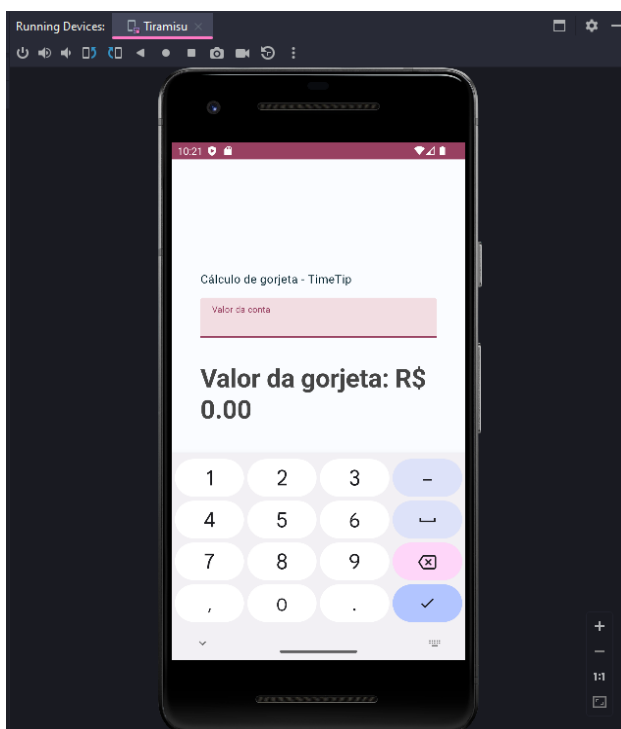
Fonte imagem: autoria própria

- value:** o valor do campo de entrada, que está vinculado à **variável amountInput**. Isso significa que qualquer alteração em amountInput será refletida, automaticamente, no campo de entrada.

- **onValueChanged:** uma função de callback, que é acionada sempre que o valor do campo de entrada é alterado. **Ela atualiza o valor de amountInput para refletir as alterações feitas no campo de entrada.**
- **singleLine:** define se o campo de entrada deve ser limitado a uma única linha.
- **label:** um rótulo exibido acima do campo de entrada, para indicar o que é esperado no campo.
- **keyboardOptions:** opções de teclado para o campo de entrada. Neste caso, está configurado para um teclado numérico.
- **modifier:** o modificador que permite personalizar a aparência e o comportamento do componente.

**Execute o projeto e observe que:**

- ✓ O label já identifica o que aquele campo faz
- ✓ O teclado numérico é ativado



Fonte imagem: autoria própria

### 1.7. Criando a lógica da calculadora em uma função privada

Nesta próxima etapa, iremos criar a funcionalidade principal desta calculadora de gorjetas. No nosso código, temos a seguinte função já disponibilizado no início desta unidade de ensino:

```

103 private fun calculateTip(amount: Double, tipPercent: Double = 15.0): String {
104     val tip = tipPercent / 100 * amount
105     return NumberFormat.getCurrencyInstance().format(tip)
106 }
  
```

Fonte imagem: autoria própria

#### Porém, o que significa aquele 'private' antes da função?

Uma função privada no Kotlin é uma função que só pode ser acessada dentro do mesmo arquivo onde foi declarada. Ela não pode ser acessada por outras classes ou arquivos. **A visibilidade "privada" é o nível mais restrito de visibilidade em Kotlin, e é usado para encapsular funcionalidades que não precisam ser expostas, fora de um escopo específico.**

**Quando declaramos uma função como privada, estamos indicando que essa função deve ser usada apenas internamente naquele arquivo.** Isso ajuda a modularizar e organizar o código, evitando que funcionalidades internas sejam acessíveis a partir de outros componentes do código, que não precisam delas e facilitando a manutenção deste.

Voltando à imagem acima, estamos definindo uma função privada chamada **calculateTip**. Ela aceita dois parâmetros: **amount (o valor total da conta) e tipPercent (a porcentagem de gorjeta)**. O parâmetro **tipPercent** tem um valor padrão de 15.0, o que significa que, se nenhum valor for fornecido, a função usará uma taxa de gorjeta de 15%.

- **val tip = tipPercent / 100 \* amount:** nesta linha, estamos calculando o valor da gorjeta. Dividimos **tipPercent** por 100, para obter a proporção da porcentagem e, em seguida, multiplicamos pelo **amount**, para obter o valor real da gorjeta, com base no valor total da conta.
- **return NumberFormat.getCurrencyInstance().format(tip):** aqui estamos formatando o valor da gorjeta, usando **NumberFormat.getCurrencyInstance()**. Isso formata o valor da gorjeta, de acordo com as configurações de **formatação regional do dispositivo, garantindo que ele seja exibido como um valor monetário adequado.**

**O tipo de retorno da função é String, pois o resultado é uma string formatada do valor da gorjeta.**

### 1.8. Configurando a função `calculateTip` e aplicando `toDoubleOrNull`:

O conteúdo que o usuário insere no campo de texto, dentro do elemento de composição é passado de volta para a função de **callback `onValueChange`**, **como uma string, mesmo que o usuário tenha inserido um número**. Para resolver essa situação, é **necessário realizar uma conversão do valor presente, na variável `amountInput`, que armazena a quantia inserida pelo usuário**. Isso garantirá que o valor seja tratado adequadamente como um número, permitindo operações matemáticas e cálculos apropriados.

#### A função `toDoubleOrNull`

É uma função de extensão em Kotlin, que está disponível para objetos do **tipo `String`**. Ela é usada para tentar **converter uma string em um valor do tipo `Double` (número de ponto flutuante de precisão dupla)**, e se a conversão for bem-sucedida, ela retorna o **valor do tipo `Double`**. Se a conversão falhar, ou seja, se a string não puder ser interpretada como um número de ponto flutuante, a função retorna `null`.

Isso é particularmente útil, quando você está lidando com entrada de usuário, como no caso de campos de texto, onde o usuário pode inserir uma variedade de valores. **A função `toDoubleOrNull` permite que você tente converter essa entrada de string em um número, ao mesmo tempo em que lida com possíveis casos em que a conversão possa falhar.**



```

82  @Composable
83  fun EditNumberField(modifier: Modifier = Modifier) {
84      var amountInput by remember { mutableStateOf( value: "" ) }
85      val amount = amountInput.toDoubleOrNull() ?: 0.0
86      val tip = calculateTip(amount)
87
88      TextField(
89          value = amountInput,
90          onValueChange = { amountInput = it },
91          singleLine = true,
92          label = { Text(stringResource(R.string.bill_amount)) },
93          keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Number),
94          modifier = modifier
95      )
96  }
    
```

2 variáveis adicionadas

Fonte imagem: autoria própria

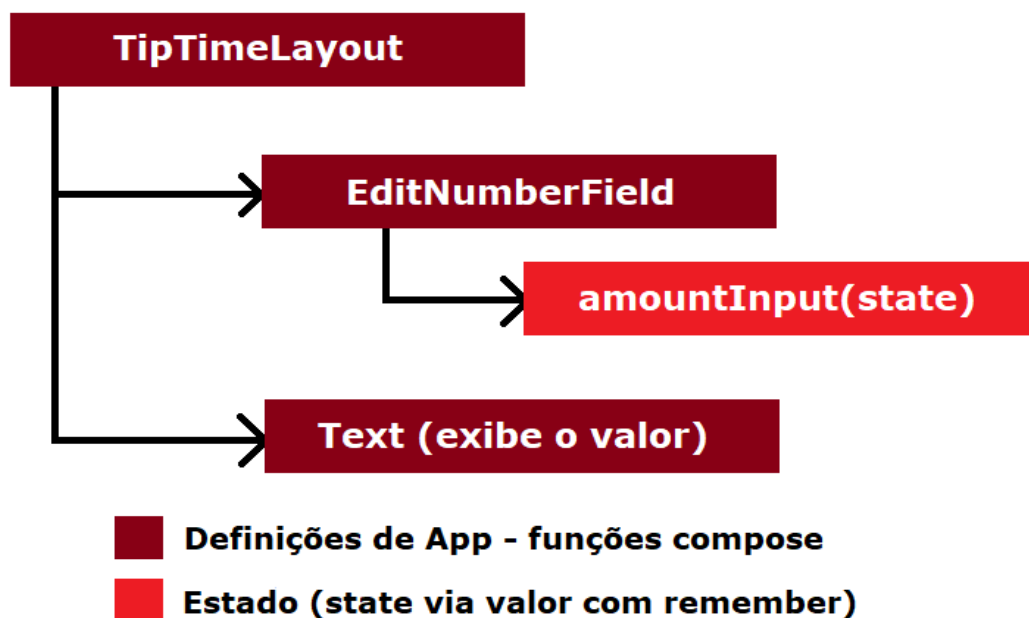
- **`val amount = amountInput.toDoubleOrNull() ?: 0.0`**: Aqui, a string `amountInput` é convertida em um valor do tipo `Double` usando a função `toDoubleOrNull()`. Se a conversão falhar, um valor padrão de `0.0` é atribuído a `amount`.

## DESENVOLVIMENTO DE APLICATIVOS I

- **O operador Elvis ?:** vai retornar a expressão precedente, se o valor não for null e à expressão seguinte, quando o valor for null. Ele permite programar esse código de maneira mais semântica. Para saber mais: ([Null safety | Kotlin Documentation](#)) .
- **val tip = calculateTip(amount):** aqui, o valor amount é passado para a função calculateTip, para calcular a gorjeta.

### 1.9. Elevando states

Para que possamos exibir de forma adequada o valor calculado da gorjeta, precisamos acessar a variável amountInput, que está declarada na função TipTimeLayout(), porém, esta variável (amountInput) é o estado de texto definido no escopo da função EditNumberField(). Observe a hierarquia da aplicação na próxima imagem.



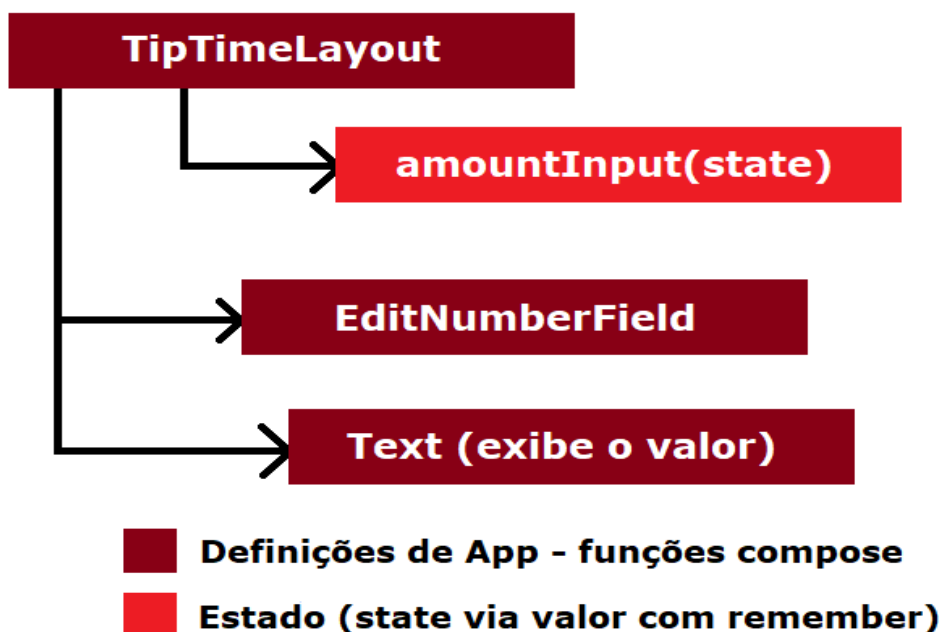
Fonte imagem: autoria própria

Essa configuração não viabiliza a exibição do valor da gorjeta no novo elemento **compose Text**, uma vez que o **Text** necessita de acesso à variável **amount** calculada com base na variável **amountInput**. É necessário disponibilizar a variável **amount**, para a função **TipTimeLayout()**, a fim de superar essa limitação de visibilidade.

As declarações de variáveis e funções no Kotlin, seguem uma ordem de execução sequencial padrão e precisam ser definidas antes de serem usadas. **Variáveis devem ser declaradas antes de serem referenciadas, e funções devem ser definidas antes de serem chamadas, a isso chamamos de Elevação de Estado.**

Precisamos deixar a nossa aplicação com a hierarquia da próxima imagem:





Fonte imagem: autoria própria

Dentro de uma função de composição combinável, é viável estabelecer variáveis destinadas a conter o status, visando a exibição na interface. Como exemplo, configuramos a variável "amountInput", como um estado presente no componente EditNumberField().

**À medida que a aplicação adquire complexidade, e outros componentes de composição demandam acesso ao estado interior do componente EditNumberField(), torna-se imperativo elevar o estado da função de composição EditNumberField(), isto é, realizar sua extração.**

#### **Em que momentos precisamos declarar a técnica de elevar estados?**

- ✓ Compartilhar o estado com várias funções de composição (compose);
- ✓ Criar um elemento de composição (compose) sem estado para ser reutilizado no app.

Ao efetuar a extração do estado de uma função de composição, o resultado obtido é referido como uma função "sem estado". Em outras palavras, as funções combináveis podem assumir um estado ausente, quando seu estado é extraído. Isso significa que elas não armazenam, definem ou modificam nenhum estado.

Um componente combinável destituído de estado é aquele que não engloba qualquer estado, ou seja, não retém, estabelece ou altera estado algum. Por contrapartida, um



componente combinável com estado detém um estado que é passível de alterações ao longo do tempo.

Na prática, em um App real, pode ser muito difícil ter um elemento de composição sem nenhum estado, é sempre aconselhável que estes elementos tenham no mínimo um estado e o projeto de aplicação precisa prever a elevação de estados como um tópico importante.

O primeiro passo que iremos fazer, será a elevação das variáveis que contém os estados, para isso, subiremos com a declaração dos dados. Faremos uma elevação, movendo o estado lembrado da função **EditNumberField()** para a **TipTimeLayout()**:



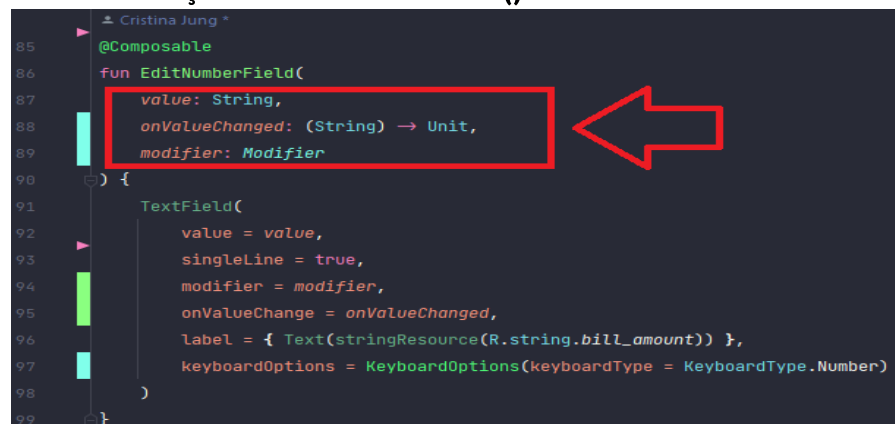
```

54  @Composable
55  fun TipTimeLayout() {
56      var amountInput by remember { mutableStateOf( value: "" ) }
57
58      val amount = amountInput.toDoubleOrNull() ?: 0.0
59      val tip = calculateTip(amount)
60
61      Column(
62          modifier = Modifier.padding(40.dp),
63          horizontalAlignment = Alignment.CenterHorizontally,
64          verticalArrangement = Arrangement.Center
65      ) { this: ColumnScope
66          Text(
67              text = stringResource(R.string.calculate_tip),
68              modifier = Modifier
69                  .padding(bottom = 16.dp)
70                  .align(alignment = Alignment.Start)
71          )
72          EditNumberField(
73              value = amountInput,
74              onValueChanged = { amountInput = it },
75              modifier = Modifier.padding(bottom = 32.dp).fillMaxWidth()
76          )
77          Text(
78              text = stringResource(R.string.tip_amount, tip),
79              style = MaterialTheme.typography.displaySmall
80          )
81          Spacer(modifier = Modifier.height(150.dp))
82      }
83  }
    
```

**Elevação dos estados**

Fonte imagem: autoria própria

#### Na função EditNumberField():



```

85  @Composable
86  fun EditNumberField(
87      value: String,
88      onValueChanged: (String) -> Unit,
89      modifier: Modifier
90  ) {
91      TextField(
92          value = value,
93          singleLine = true,
94          modifier = modifier,
95          onValueChange = onValueChanged,
96          label = { Text(stringResource(R.string.bill_amount)) },
97          keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Number)
98      )
99  }
    
```

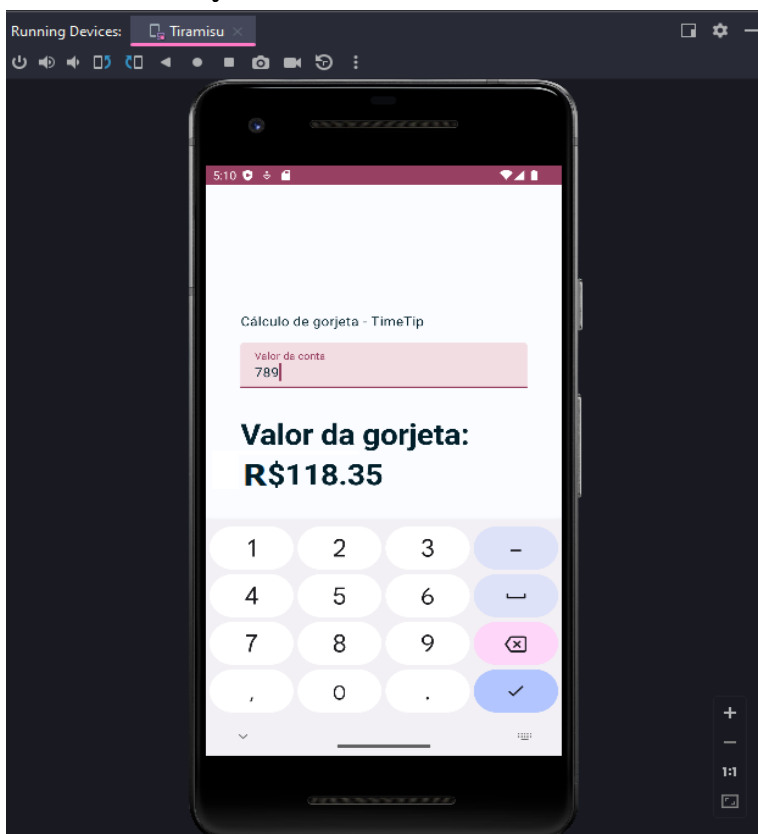
Ainda na mesma função `EditNumberField()`, vamos atualizar a função `compose TextField` para que ela possa usar os parâmetros que estão sendo passados.

```

85  @Composable
86  fun EditNumberField(
87      value: String,
88      onValueChanged: (String) → Unit,
89      modifier: Modifier
90  ) {
91      TextField(
92          value = value,
93          singleline = true,
94          modifier = modifier,
95          onValueChange = onValueChanged,
96          label = { Text(stringResource(R.string.bill_amount)) },
97          keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Number)
98      )
99  }
    
```

Fonte imagem: autoria própria

**A aplicação está pronta, agora podemos executar o emulador do nosso projeto e fazer a execução dele.**



Fonte imagem: autoria própria

Neste próximo código, você poderá conferir a compose `TipTimeLayout`. Observe que foi inserida mais uma função para `Text`, em função da formatação do valor da gorjeta. Faça a conferência de seu código.

```
@Composable
fun TipTimeLayout() {
    var amountInput by remember { mutableStateOf("") }

    val amount = amountInput.toDoubleOrNull() ?: 0.0
    val tip = calculateTip(amount)

    Column(
        modifier = Modifier.padding(40.dp),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ) {
        Text(
            text = stringResource(R.string.calculate_tip),
            modifier = Modifier
                .padding(bottom = 16.dp)
                .align(alignment = Alignment.Start)
        )
        EditNumberField(
            value = amountInput,
            onValueChanged = { 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))
    }
}
```

Fonte imagem: autoria própria

Também poderá acessar [neste link](#) o código final da `MainActivity.ktl`.

### **1.10. Review do conteúdo desta unidade**

- O estado em um app se refere a qualquer valor suscetível de alteração ao longo do decorrer do tempo.
- A composição representa a elaboração da interface feita pelo Compose, enquanto ele processa os elementos combináveis.
- Aplicações Compose invocam funções combináveis para converter informações em elementos de interface.
- A composição inicial corresponde à criação da interface pelo Compose, que ocorre quando as funções combináveis são executadas pela primeira vez.

- A recomposição consiste em reexecutar os mesmos elementos combináveis para atualizar a estrutura quando seus dados sofrem mudanças.
- Elevação de estado constitui um padrão utilizado para deslocar o estado para um nível superior, resultando na transformação de um componente em um componente desprovido de estado.

### ***1.11. Leituras e estudos complementares***

Importante sempre nos fundamentarmos na documentação do oficial para que possamos entender e assimilar melhor. Segue abaixo, links importantes da documentação do Android Developers que podem ajudá-lo no estudo de desenvolvimento de aplicações mobile Android nativas.

[Estado e Jetpack Compose | Android Developers](#)

[Trabalhando com o Compose | Jetpack Compose | Android Developers](#)

[Onde elevar o estado | Jetpack Compose | Android Developers](#)

## 2. Referências

---

- Múltiplos autores: **KOTLIN DOCS**. 2023. Disponível em: <<https://kotlinlang.org/docs/home.html>>. Acesso em: 17 de julho de 2023
- Múltiplos autores: **JETBRAINS**. 2022. Disponível em: <<https://www.jetbrains.com/pt-br/>>. Acesso em: 18 de julho de 2023
- Múltiplos autores: **DOCUMENTAÇÃO ANDROID STUDIO**. 2023. Disponível em: <<https://developer.android.com/studio>>. Acesso em: 18 de julho de 2023
- Múltiplos autores: ESTADO E JETPACK COMPOSE. 2023. Disponível em: <[Estado e Jetpack Compose | Android Developers](#)>. Acesso em: 13 de agosto de 2023
- Múltiplos autores: **KOTLIN DOCS. ELVIS OPERATOR**. 2023. Disponível em: <<https://kotlinlang.org/docs/home.html>>. Acesso em: 13 de julho de 2023