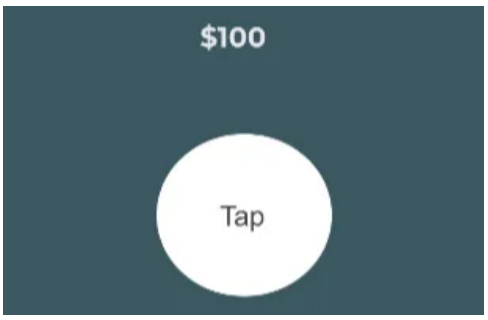
Aula 02.2 - Primeiros passos com o Jetpack Compose - Projeto Tap Count

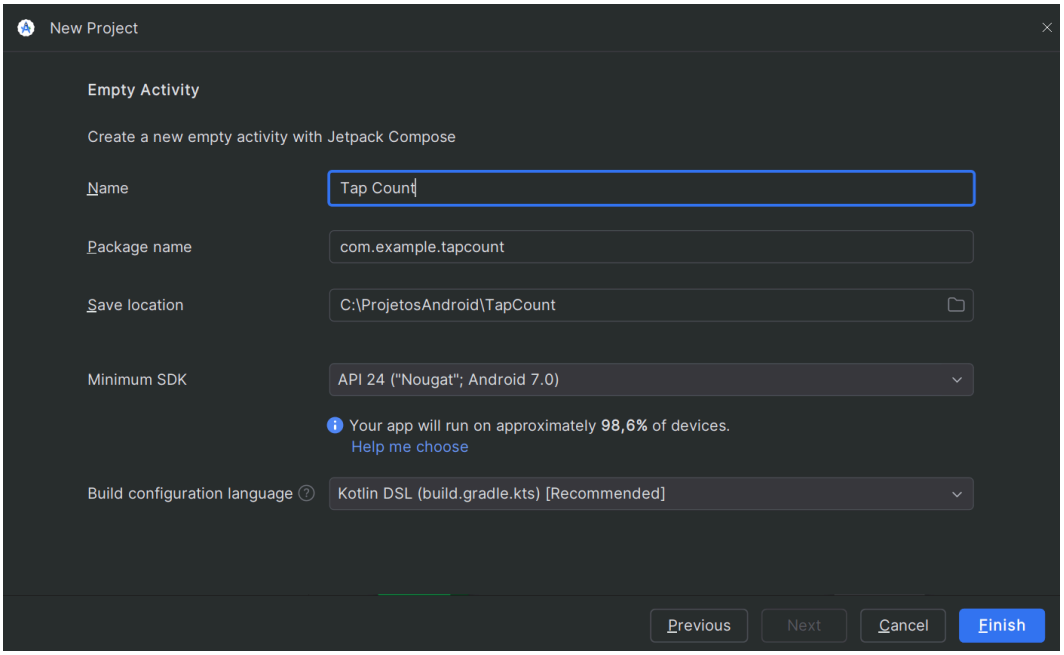
O próximo passo será criar uma interface simples

A ideia dessa interface simples é, ao tocar no “Tap” o valor acima dele será incrementado.



Portanto criaremos um novo projeto para criar essa interface:

Chamaremos de `Tap Count`



Após a criação do projeto, estruture ele dessa forma:

```
package com.example.tapcount

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.fillMaxHeight
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Surface
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.tooling.preview.Preview
import com.example.tapcount.ui.theme.TapCountTheme

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            TapCountTheme {
                Surface(
                    modifier = Modifier.fillMaxHeight().fillMaxWidth(),
                    color = MaterialTheme.colorScheme.primary
                ){
                                Text(text = "Hello World")
                }
            }
        }
    }
}

@Preview(showBackground = true)
@Composable
fun DefaultPreview() {
    TapCountTheme {
```

```
    }
}
```

Note que possuí uma estrutura inicial idêntica ao projeto da apostila anterior.

Criando a estrutura base da interface

Agora criamos nossa **primeira função Composable**, chamada `MyApp()`. Ela é responsável por estruturar a base da interface do aplicativo. Utilizamos o componente `Surface` , ou seja, é um recorte do que já estava disponível na nossa MainActivity.

Coloque ela entre a MainActivity e o @Preview

```
@Composable
fun MyApp(){
    Surface(
        modifier = Modifier.fillMaxHeight().fillMaxWidth(),
        color = MaterialTheme.colorScheme.primary
    ){
        Text(text = "Hello World")
    }
}
```

Atualizando o código para usar a função `MyApp()`

Com a função `MyApp()` criada, agora ajustamos o código principal para utilizá-la dentro do `setContent`. Assim, toda a interface passa a ser organizada a partir dessa função. Esse padrão é muito utilizado para manter o código mais **modular** e **fácil de manter**.

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            TapCountTheme {
                MyApp()
            }
        }
    }
```

```
}

@Composable
fun MyApp(){
    Surface(
        modifier = Modifier.fillMaxHeight().fillMaxWidth(),
        color = MaterialTheme.colorScheme.primary
    ){
        Text(text = "Hello")
    }
}

@Preview(showBackground = true)
@Composable
fun DefaultPreview() {
    TapCountTheme {
        MyApp()
    }
}
```

Aplicando uma cor personalizada à Surface

Até agora, usamos a cor do tema com:

```
color = MaterialTheme.colorScheme.primary
```

Mas agora vamos usar uma cor **específica** definida diretamente no código com valor hexadecimal:

```
color = Color(0xFF546E7A)
```

Explicação:

- `Color(...)`: é uma função do Jetpack Compose que recebe um valor hexadecimal (como no HTML/CSS) para definir cores.
- `0xFF546E7A`: esse valor representa uma cor no formato ARGB:
    - `0x`: O prefixo `0x` indica que o número está sendo representado **em hexadecimal (base 16)**.

- `FF546E7A` → é o valor da cor no formato **ARGB (Alpha, Red, Green, Blue)**
- `FF`: Alpha (opacidade máxima, `FF` = 100% valor máximo)
- `54`, `6E`, `7A`: são os valores de vermelho, verde e azul, respectivamente (em hexadecimal).

Criaremos o Circulo que receberá o toque do usuário

Vamos agora criar uma nova função `Composable` que renderizará um **círculo** utilizando o componente `Card` e a forma `CircleShape`. Essa estrutura será utilizada para o botão "Tap" que incrementará o contador.

```
@Preview
@Composable
fun CreateCircle() {
    Card(
        modifier = Modifier
            .padding(3.dp)     // Espaçamento externo
            .size(45.dp),      // Altura e largura iguais
        shape = CircleShape    // Formato circular
    ) {

    }
}
```

Explicando o código:

- `Card`: é um container visual com uma aparência elevada, ideal para destacar elementos.
- `Modifier.padding(3.dp)`: define um espaço externo de 3dp ao redor do círculo.
- `Modifier.size(45.dp)`: define a largura e altura do círculo como 45dp, formando um quadrado.
- `shape = CircleShape`: transforma o quadrado em um círculo ao arredondar completamente suas bordas **(Isso só funciona corretamente se a largura e a altura forem iguais)**.

> Neste momento o Card ainda está vazio, mas em breve vamos adicionar um texto no centro para representar o botão

> "Tap".

O
`shape` define **como os cantos de um componente são desenhados**, e você pode utilizar várias opções prontas ou criar suas próprias.

Exemplos de valores para `shape`:

`RoundedCornerShape(dp)`

Arredonda os cantos com o valor em dp:

```
shape = RoundedCornerShape(16.dp)
```

`RoundedCornerShape(topStart = ..., bottomEnd = ...)`

Permite arredondar **cantinhos específicos**:

```
shape = RoundedCornerShape(topStart = 16.dp, bottomEnd = 8.dp)
```

`RectangleShape`

Formato padrão, **sem cantos arredondados** (é um retângulo):

```
shape = RectangleShape
```

`CutCornerShape(dp)`

Corta os cantos ao invés de arredondar:

```
shape = CutCornerShape(12.dp)
```

Personalizado (Custom Shape)

Você pode criar formas próprias implementando a interface `Shape`.

📣 Manteremos o `shape = CircleShape`

O próximo passo é adicionar um texto ao nosso Circle dessa forma:

```
@Preview
@Composable
fun CreateCircle() {
    Card(modifier = Modifier.padding(3.dp)
        .size(45.dp),
        shape = CircleShape
    ){
        Text(text = "Tap")
    }
}
```

Notem que o texto ficou desalinhado com o centro do circulo.

Alinhando o texto ao centro do Circle

Há diversas formas de se realizar esse alinhamento, mas nesse momento utilizaremos um Composable chamado de `Box`.

```
Box (modifier = Modifier.fillMaxSize(), contentAlignment = Alignment.Center){
    Text(text = "Tap")
}
```

Explicando o código:

- `modifier`: O modifier definirá a altura e largura do Box, para que o contentAlignment possa realizar o calculo que centraliza o seu conteúdo no centro.
- `contentAlignment`: Realiza o alinhamento do conteúdo interno.

Adicionando Interação com `clickable`

Atualizamos nossa função `CreateCircle()` para permitir que o círculo responda ao toque do usuário. Para isso, usamos o modificador:

```
.clickable {
    Log.d("Tap", "CreateCircle: Tap")
}
```

O que foi adicionado:

- `clickable { ... }`: permite que o componente detecte cliques. No caso, usamos o `Log.d()` para registrar a ação no logcat. Esse tipo de log é útil para depuração (debug) e testes iniciais.
- `Log.d("Tap", "CreateCircle: Tap")`:
  A função
  `Log.d()` pertence à classe `Log` do Android e é usada para registrar mensagens no **logcat**, que é a ferramenta de visualização de logs do Android Studio. O `d` indica que essa mensagem é de nível **debug**, ou seja, destinada ao desenvolvedor para verificar o comportamento da aplicação durante o desenvolvimento.
  - `"Tap"` é a tag usada para identificar o log.
  - `"CreateCircle: Tap"` é a mensagem que será exibida no log quando o usuário tocar no círculo.

Outros métodos da classe `Log` incluem:

- `Log.i()` → info (informação)
- `Log.w()` → warning (aviso)
- `Log.e()` → error (erro)
- `Log.v()` → verbose (muito detalhado)

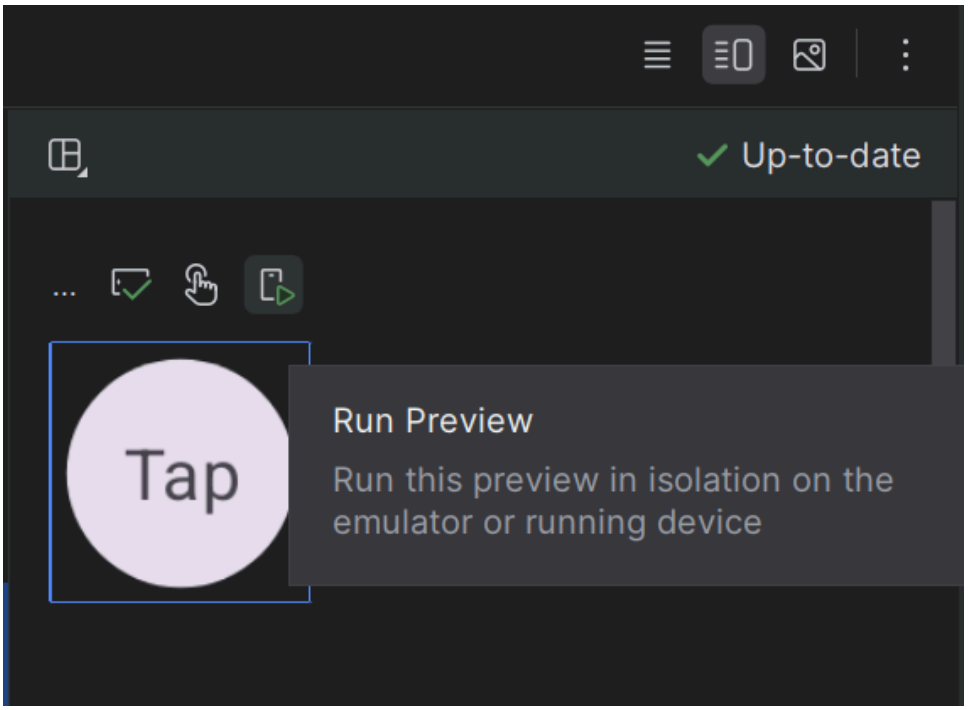
O Composable CreateCircle deve ficar dessa forma

```
@Preview
@Composable
fun CreateCircle() {
    Card(modifier = Modifier
        .padding(all = 30.dp)
        .size(45.dp)
        .clickable {
            Log.d("Tap", "CreateCircle: Tap")
        },
        shape = CircleShape
    ){
```

```
        Box (modifier = Modifier.fillMaxSize(), contentAlignment = Alignment.Cen
            Text(text = "Tap")
        }
    }
}
```

Executando separadamente um Preview para testar o seu funcionamento

No espaço onde o Preview é apresentado há a disponibilidade de algumas funções, uma delas é a função `Run Preview` que realiza o Build separadamente do Preview para que possamos realizar o teste do mesmo.



Chamando a função `CreateCircle` dentro do `MyApp`

Com o nosso botão criado, agora vamos chamá-lo dentro da função `MyApp()` para vermos seu funcionamento na interface principal. Basta inserir a chamada da função composable `CreateCircle()` dentro do conteúdo da `Surface`:

```
@Composable
fun MyApp(){
    Surface(
        modifier = Modifier.fillMaxHeight().fillMaxWidth(),
```

```
        color = Color(0xFF546E7A)
    ){
        CreateCircle()
    }
}
```

Ajustando o Layout com Column

Ao chamarmos a função `CreateCircle()` diretamente dentro da `Surface`, o círculo passou a ocupar todo o espaço da tela. Isso acontece porque o `Card`, por padrão, tenta preencher todo o espaço disponível, especialmente quando usado diretamente dentro de um container como `Surface`.

A solução é envolver o conteúdo com um **layout de coluna**, utilizando o Composable `Column()`. Isso organiza os elementos verticalmente e permite um melhor controle sobre o espaço ocupado por cada componente.

```
@Composable
fun MyApp(){
    Surface(
        modifier = Modifier.fillMaxHeight().fillMaxWidth(),
        color = Color(0xFF546E7A)
    ){
        Column(){
            CreateCircle()
        }
    }
}
```

Dessa forma, o círculo é exibido com o tamanho correto, e podemos adicionar outros elementos abaixo ou acima dele dentro da `Column`, organizando a interface de forma mais controlada.

Agora que estamos usando
`Column` para organizar os elementos na tela, podemos adicionar dois parâmetros importantes para controlar o alinhamento:

`verticalArrangement = Arrangement.Center`

- Esse parâmetro centraliza os elementos **verticalmente** dentro da `Column`.
- Como temos apenas um elemento (`CreateCircle`), ele será posicionado no meio da altura da tela.

`horizontalAlignment = Alignment.CenterHorizontally`

- Esse parâmetro centraliza os elementos **horizontalmente** dentro da `Column`.
- Ou seja, o círculo (nosso botão) vai aparecer bem no **centro da tela**, tanto na horizontal quanto na vertical.

A `Column` ficará dessa forma:

```
Column(
    verticalArrangement = Arrangement.Center,
    horizontalAlignment = Alignment.CenterHorizontally
){
    CreateCircle()
}
```

Adicionando o Texto e Espaçamento

Agora vamos adicionar um texto acima do botão para exibir um valor no nosso exemplo, o valor inicial de `R$100`. Também incluiremos um espaçamento entre o texto e o botão para garantir um melhor posicionamento visual.

Atualize a `column` da seguinte forma

```
Column(
    verticalArrangement = Arrangement.Center,
    horizontalAlignment = Alignment.CenterHorizontally
){
    Text(text = "R$100")
    Spacer(modifier = Modifier.height(100.dp))
    CreateCircle()
}
```

Estilizando o Texto

Vamos agora aplicar um estilo ao texto exibido acima do botão para deixá-lo mais visível e esteticamente agradável. Usaremos a classe `TextStyle` para definir a cor da fonte e seu tamanho.

Atualize o `Text` com o seguinte código:

```
Text(text = "R$100",
      style = TextStyle(
       color = Color.White,
       fontSize = 39.sp)
)
```

O que é `sp` no Android?

A sigla `sp` significa **scale-independent pixels** (*pixels independentes de escala*). Esse é um tipo de unidade usada especialmente para textos no Android.

Por que usar `sp`?

- Ele leva em consideração as configurações de acessibilidade do usuário, como o aumento de tamanho da fonte no sistema.
- Isso garante que o texto fique legível em diferentes dispositivos e tamanhos de tela.
- É uma boa prática recomendada pelo Android para manter a acessibilidade e responsividade da interface.

Finalizamos a tela, agora focaremos na lógica do clique do botão

Lógica de Clique no Botão

Agora adicionamos a parte lógica ao nosso botão para que ele reaja ao clique. Para isso, seguimos os seguintes passos:

1. Criamos uma variável `moneyCounter` dento da função `CreateCircle`

   ```
   var moneyCounter = 0
   ```

2. Dentro do `clickable`, incrementamos a variável e mostramos o valor no log:

   ```
   moneyCounter += 1
   ```

```
Log.d("Contador", "CreateCircle: $moneyCounter")
```

Agora testamos e verificamos o log para ver se esta incrementando

Vamos tentar exibir o valor dos cliques diretamente no botão

Para deixar a aplicação mais interativa, atualizamos o `Text()` do botão para exibir o valor atual da variável `moneyCounter`, que armazena o número de toques no botão.

```
Text(text = "Tap $moneyCounter")
```

📣 Ao testar, notamos que o valor não é atualizado no botão

Entendendo a forma com que os elementos da UI são redesenhados (Recomposição - Recomposition)

`Recomposição ou Recomposition` é o processo de redesenho das suas funções Composables toda vez que os dados que compõe a interface são alterados.

Na etapa anterior, criamos o botão e adicionamos uma variável chamada `moneyCounter` para contar os cliques. Dentro do `Modifier.clickable` fizemos o incremento.

Problema

Mesmo clicando no botão e vendo os valores sendo atualizados no `Logcat`, **o valor exibido na tela não muda**. Isso acontece porque o Compose só redesenha a tela automaticamente quando usamos variáveis reativas, ou seja, observáveis.

A solução: `remember` + `mutableStateOf`

Para que o Compose saiba que essa variável precisa provocar um redesenho da interface quando alterada, usamos o `remember` em conjunto com `mutableStateOf`. Veja a forma correta:

```
var moneyCounter by remember { mutableStateOf(0) }
```

Esse código faz com que o Compose "lembre" o valor atual de `moneyCounter` mesmo após recomposições da tela, e redesenhe a interface toda vez que esse valor mudar.

O código da nossa função CreateCircle ficará dessa forma:

```
@Preview
@Composable
fun CreateCircle() {
    var moneyCounter by remember{ mutableStateOf(0) }
    Card(
        modifier = Modifier
            .padding(3.dp)
            .size(105.dp)
            .clickable {
                moneyCounter += 1
                Log.d("Counter", "CreateCircle: $moneyCounter")
            },
        shape = CircleShape
    ) {
        Box (modifier = Modifier.fillMaxSize(), contentAlignment = Alignment.Cen
            Text(text = "Tap $moneyCounter")
        }
    }
}
```

Mover o estado para a função `MyApp`

Para exibir o valor do `moneyCounter` no lugar correto precisaremos mover a variável `moneyCounter` para dentro do `MyApp()` e passar ela e a função de incremento como parâmetros para o `CreateCircle`. Ficando dessa forma:

```
@Composable
fun MyApp(){
    var moneyCounter by remember { mutableStateOf(0) }
    Surface(
        modifier = Modifier.fillMaxHeight().fillMaxWidth(),
        color = Color(0xFF546E7A)
    ){
```

```
        Column(
            verticalArrangement = Arrangement.Center,
            horizontalAlignment = Alignment.CenterHorizontally
        ){
            Text(text = "R$$moneyCounter",
                style = TextStyle(
                    color = Color.White,
                    fontSize = 39.sp)
            )
            Spacer(modifier = Modifier.height(100.dp))
            CreateCircle(moneyCounter) {moneyCounter+=1}
        }
    }
}
```

E atualizar a função `CreateCircle` para receber esses parâmetros:

```
//@Preview
@Composable
fun CreateCircle(moneyCounter: Int, onTap: () -> Unit) {
    Card(
        modifier = Modifier
            .padding(3.dp)
            .size(150.dp)
            .clickable {
                onTap()

                Log.d("Contador", "CreateCircle: $moneyCounter")
            },
        shape = CircleShape    // Formato circular
    ) {
        Box (modifier = Modifier.fillMaxSize(), contentAlignment = Alignment.Cen
            Text(text = "Tap $moneyCounter")
        }
    }
}
```

Código completo da aplicação desenvolvida

```
package com.example.tapcount

import android.os.Bundle
import android.util.Log
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.clickable
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.fillMaxHeight
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.shape.CircleShape
import androidx.compose.material3.Card
import androidx.compose.material3.Surface
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.text.TextStyle
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import com.example.tapcount.ui.theme.TapCountTheme

class MainActivity : ComponentActivity() {
```

```
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        setContent {
            TapCountTheme {
                MyApp()
            }
        }
    }
}

@Composable
fun MyApp(){
    var moneyCounter by remember { mutableStateOf(0) }
    Surface(
        modifier = Modifier.fillMaxHeight().fillMaxWidth(),
        color = Color(0xFF546E7A)
    ){
        Column(
            verticalArrangement = Arrangement.Center,
            horizontalAlignment = Alignment.CenterHorizontally
        ){
            Text(text = "R$$moneyCounter",
                style = TextStyle(
                    color = Color.White,
                    fontSize = 39.sp)
            )
            Spacer(modifier = Modifier.height(100.dp))
            CreateCircle(moneyCounter) {moneyCounter+=1}
        }
    }
}

//@Preview
@Composable
fun CreateCircle(moneyCounter: Int, onTap: () -> Unit) {
    Card(
        modifier = Modifier
```

```
            .padding(3.dp)     // Espaçamento externo
            .size(150.dp)      // Altura e largura iguais
            .clickable {
                onTap()

                Log.d("Contador", "CreateCircle: $moneyCounter")
            },
        shape = CircleShape    // Formato circular
    ) {
        Box (modifier = Modifier.fillMaxSize(),
                contentAlignment = Alignment.Center)
                {
            Text(text = "Tap $moneyCounter")
        }
    }
}

@Preview(showBackground = true)
@Composable
fun DefaultPreview() {
    TapCountTheme {
        MyApp()
    }
}
```

Outra forma de construir essa interface

Outra forma de construir essa interface seria unindo todo o código dentro da função `MyApp`, dessa forma não teríamos que passar a variável `moneyCounter` e a função de incremento como parâmetro para a função `CreateCircle` ficando mais simples o código

O código ficaria dessa forma:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            TapCountTheme {
```

```
            MyApp()
        }
    }
  }
}

@Composable
fun MyApp(){
  var moneyCounter by remember { mutableStateOf(0) }
  Surface(
    modifier = Modifier.fillMaxHeight().fillMaxWidth(),
    color = Color(0xFF546E7A)
  ){
    Column(
      verticalArrangement = Arrangement.Center,
      horizontalAlignment = Alignment.CenterHorizontally
    ){
      Text(text = "R$$moneyCounter",
        style = TextStyle(
          color = Color.White,
          fontSize = 39.sp)
      )
      Spacer(modifier = Modifier.height(100.dp))
      Card(
        modifier = Modifier
          .padding(3.dp)     // Espaçamento externo
          .size(150.dp)      // Altura e largura iguais
          .clickable {
            moneyCounter += 1
            Log.d("Contador", "CreateCircle: $moneyCounter")
          },
        shape = CircleShape    // Formato circular
      ) {
        Box (modifier = Modifier.fillMaxSize(),
        contentAlignment = Alignment.Center)
        {
          Text(text = "Tap")
        }
```

```
        }
      }
    }
}

@Preview(showBackground = true)
@Composable
fun DefaultPreview() {
    TapCountTheme {
        MyApp()
    }
}
```