Aula 02.1 - Primeiros passos com o Jetpack Compose

Após gerar o primeiro projeto no Android Studio utilizando Kotlin + Jetpack Compose, vamos entender esse projeto e começar a dar nossos primeiros passos na linguagem.

O que é `@Composable`?

No **Jetpack Compose**, toda interface de usuário é construída com **funções especiais** chamadas de **composable functions**. Essas funções são marcadas com a **anotação** `@Composable`, que é o coração do Jetpack Compose.

Por que usar `@Composable`?

O `@Composable` informa ao compilador que aquela função **descreve um pedaço da interface (UI)** do aplicativo e pode ser **recomposta automaticamente** sempre que o estado da aplicação mudar.

Exemplo:

```
@Composable
fun Greeting(name: String) {
    Text(text = "Olá, $name!")
}
```

Neste exemplo:

- Criamos uma função chamada `Greeting`.
- A anotação `@Composable` permite que o Compose entenda que essa função **desenha algo na tela**.
- Usamos a função `Text()`, que também é composable, para exibir uma mensagem.

O que é `@Preview`?

A anotação `@Preview` é usada no Jetpack Compose para **visualizar uma função composable diretamente no Android Studio**, **sem precisar rodar o app no**

emulador ou dispositivo físico.

Ela é muito útil para **testar rapidamente como a interface está ficando**, economizando tempo durante o desenvolvimento.

Exemplo básico:

```
@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    Greeting("Android")
}
```

O que está acontecendo aqui?

- `@Preview`: Diz ao Android Studio que queremos **ver a saída dessa função composable** na aba *Preview* da IDE.
- `showBackground = true`: Mostra um fundo branco (útil para ver melhor o layout).
- A função `GreetingPreview()` é uma função composable normal, **mas serve apenas para exibir o conteúdo de outra composable** (no caso, a `Greeting()`).

Dica adicional:

- Você pode **criar vários** `@Preview` para testar diferentes estados de uma mesma tela ou componente:

```
@Preview
@Composable
fun GreetingPreviewBruno() {
    Greeting("Bruno")
}

@Preview
@Composable
fun GreetingPreviewMaria() {
    Greeting("Maria")
}
```

Criando nossa primeira função `@Composable`

Agora que já entendemos o que é uma função composable, vamos criar nossa primeira.

Exemplo: Exibindo uma idade

```
@Preview
@Composable
fun ShowAge(age: Int = 12) {
    Text(text = age.toString())
}
```

Resultado:

Essa função vai mostrar o número **12** na tela ou qualquer valor passado como parâmetro, se quiser reutilizá-la em outro lugar.

Utilizando composables: combinando `Greeting` e `ShowAge`

No Jetpack Compose, você pode **utilizar suas funções composables** dentro de outras, compondo a interface como peças de LEGO.

Vamos ver como isso funciona na prática.

Exemplo: Usando `Greeting` e `ShowAge` no `GreetingPreview`

```
@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    PrimeiroProjetoTheme {
        Greeting("Android")
        ShowAge(age = 34)
    }
}
```

> Ao tentarmos exibir mais de um elemento na tela, como `Greeting()` e `ShowAge()`, podemos nos deparar com um

> problema: **os textos ficam sobrepostos**.
>
> Isso acontece porque os elementos estão sendo desenhados um **em cima do outro**. Para evitar isso, precisamos organizá-los em um layout vertical.

Solução: Evita sobreposição de elementos com o `Column {}`

```
@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    PrimeiroProjetoTheme {
            Column(){
            Greeting("Android")
            ShowAge(age = 34)
        }
    }
}
```

O que o `Column` faz?

- O `Column` é um **composable de layout** que organiza seus filhos **um abaixo do outro**.
- Ele pertence ao pacote `androidx.compose.foundation.layout`.
- Resolve problemas de **sobreposição** e permite um posicionamento mais claro dos elementos na interface.

Entendendo a `MainActivity`

Toda aplicação Android começa a partir de uma `Activity`. No Jetpack Compose, usamos uma classe chamada `ComponentActivity`, que serve como ponto de entrada da aplicação. Abaixo está o exemplo da nossa `MainActivity`:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge() // Deixa o app ocupar toda a área da tela
```

```
        setContent {
            PrimeiroProjetoTheme {
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                    Greeting(
                        name = "Bruno",
                        modifier = Modifier.padding(innerPadding)
                    )
                }
            }
        }
    }
}
```

Explicação por partes:

- `MainActivity : ComponentActivity()`

  É a **activity principal** da aplicação, herda de `ComponentActivity` (que dá suporte ao Jetpack Compose).

- `onCreate()`

  Método chamado **quando a activity é criada**. É onde iniciamos nossa interface.

- `enableEdgeToEdge()`

  Permite que o conteúdo da tela use **toda a área disponível**, incluindo atrás da status bar, por exemplo.

- `setContent { ... }`

  Essa função **substitui o uso de arquivos XML**. Aqui é onde dizemos o que será exibido na tela, com Composables.

- `PrimeiroProjetoTheme { ... }`

  Aplica o **tema visual da aplicação** (cores, tipografia, formas, etc.).

- `Scaffold(...) { innerPadding -> ... }`

  Estrutura base do Material Design, que permite incluir barra superior, FAB, e conteúdo principal.

  O `innerPadding` é usado para evitar que o conteúdo fique escondido atrás de elementos da interface.

- `Greeting(...)`

  É o **composable personalizado** que criamos para mostrar um texto.

Vamos entender um pouco sobre Interfaces

As interfaces do Jetpack Compose são criadas a partir de uma “superfície”, para que possamos desenhar elementos na tela, funciona como uma **base visual** que aplica **estilo, tema e comportamento** ao conteúdo que estiver dentro dele.

Temos alguns tipos de Superfícies, mas as mais comuns de serem utilizadas são a `Scaffold` e a `Surface`
Vamos entender um pouco mais sobre elas.

O que é `Scaffold` no Jetpack Compose?

O `Scaffold` é um **Composable** do Jetpack Compose que representa uma "superfície" onde podemos desenhar elementos da interface do usuário.

`Scaffold` é um **componente de layout avançado** que organiza a interface com **estrutura de app padrão**, como:

- TopBar (barra superior)
- BottomBar (barra inferior de navegação)
- FloatingActionButton (botão flutuante)
- Drawer (menu lateral)
- Área de conteúdo principal

É ideal para telas completas e já traz padrões visuais do Material Design.

Usos comuns do `Scaffold`:

- Criar a **estrutura completa de uma Activity**
- Garantir **responsividade e organização** da interface
- Trabalhar com **componentes padrões** do Android moderno

O que é `Surface` no Jetpack Compose?

O

`Surface` é um **Composable** do Jetpack Compose que representa uma "superfície" onde podemos desenhar elementos da interface do usuário. Ele funciona como uma **base visual** que aplica **estilo, tema e comportamento** ao conteúdo que estiver dentro dele.

É um
componente básico de layout usado para **exibir conteúdo com estilo visual**, como:

- **Cor de fundo**
- **Forma (bordas arredondadas)**
- **Elevação (sombra)**
- **Bordas/padding**

Ele é muito útil para criar blocos visuais isolados e organizar a interface em camadas visuais.

Usos comuns do `Surface`:

- Envolver áreas da tela com uma cor ou tema específico
- Criar caixas com sombra
- Aplicar responsividade ao tema (cores, modos escuro/claro, etc.)

O recomendado para quem esta iniciando no desenvolvimento com o Jetpack Compose é o uso do `Surface` por ser uma Superfície básica para criação de interfaces.

Alterando o MainActivity para utilizar o Surface

No início do entendimento do Jetpack Compose utilizaremos o `Surface` para desenvolver as primeiras interfaces.

Copie o código abaixo e substitua o MainActivity

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

```
                Greeting(name = "Bruno")
            }
        }
    }
}
```

Após a troca da Superfície para o Surface, temos que realizar a importação dos componentes que ele esta utilizando para que funcione, para isso utilizamos o `Alt + Enter` nos componentes que estão em erro e solicitando a importação dos mesmos.

Para entendermos como esse Composable esta preenchendo a tela podemos modificar a `Surface` com uma cor, adicionaremos esse código `MaterialTheme.colorScheme.primary` nos parâmetros do Surface, assim conseguimos **ver claramente o espaço que ela ocupa**.

```
color = MaterialTheme.colorScheme.background
```

O que é `MaterialTheme.colorScheme.background`?

No Jetpack Compose, `MaterialTheme` representa o sistema de **temas e estilos da interface**, baseado no Material Design 3. Ele fornece cores, formas e tipografias padronizadas.

Dentro dele, `colorScheme` é um conjunto de cores organizadas por função, como:

- `primary`, `secondary`, `tertiary`

Preenchendo a tela com `Surface`

Após definirmos uma **cor de fundo** para a `Surface`, podemos aplicar um modificador que fará com que ela **ocupe todo o espaço disponível da tela**. Isso nos ajuda a **visualizar claramente seu comportamento** como um contêiner de fundo.

```
Surface(
    modifier = Modifier.fillMaxSize(),
    color = MaterialTheme.colorScheme.primary
```

```
) {
    Greeting(name = "Bruno")
}
```

`modifier = Modifier`

O parâmetro `modifier` é usado para **configurar o comportamento visual e estrutural** de um componente no Jetpack Compose.

O que é `Modifier.fillMaxSize()`?

A função `fillMaxSize()` faz com que o componente **ocupe todo o espaço disponível** no layout pai, ou seja, ele vai tentar preencher toda a tela (ou o espaço da composição onde ele estiver inserido).

Preenchendo Altura e Largura com `Modifier`

Além de `fillMaxSize()`, o Jetpack Compose oferece formas mais **específicas** de controlar o tamanho de um componente usando:

```
modifier = Modifier
    .fillMaxHeight()
    .fillMaxWidth()
```

Quando usar cada um:

- `fillMaxSize()` é um atalho que **preenche altura e largura** ao mesmo tempo.
- `fillMaxHeight()` e `fillMaxWidth()` permitem controlar **independentemente** cada dimensão, ideal para situações em que você quer, por exemplo, que algo ocupe toda a altura mas não toda a largura.

Você pode utilizar esses modificadores em qualquer `Composable`, como `Surface`, `Column`, `Box`, entre outros, para **definir claramente a área que eles devem ocupar** na tela.

Agora aplicaremos um `padding()` na nossa Surface

O comando `padding` é um **modificador** usado para adicionar **espaçamento interno** ao redor de um `Composable`. Isso faz com que o conteúdo fique afastado das bordas do seu contêiner.

```
modifier = Modifier.fillMaxHeight().fillMaxWidth().padding(all = 50.dp),
```

- `all`: É um nome de parâmetro que indica que o padding será aplicado **igualmente em todos os lados**: top, bottom, start e end.

  Ou seja, o conteúdo ficará com 50 unidades de padding para cada direção.

- `dp`: É a sigla para **density-independent pixels** (pixels independentes da densidade).

O que é `dp` (density-independent pixels)

`dp` significa **density-independent pixels**, ou em português, **pixels independentes de densidade**.

Para que serve?

Em dispositivos Android, as telas possuem diferentes densidades (quantidade de pixels por polegada). Isso significa que 1 pixel em uma tela pode ser visualmente menor ou maior em outra.

O `dp` resolve esse problema! Ele permite que você defina tamanhos visuais **consistentes** entre diferentes telas.

Como funciona?

- O Android converte automaticamente os valores de `dp` em pixels reais, de acordo com a densidade da tela.
- Assim, se você definir um padding de `10.dp`, ele terá o **mesmo tamanho visual** em qualquer dispositivo, seja ele um celular simples ou um tablet de alta resolução.

Conclusão

O que aprendemos até aqui:

- Como criar uma **função composable** usando `@Composable`.
- Como usar a anotação `@Preview` para visualizar a interface diretamente no Android Studio.
- Como exibir um texto com o componente `Text`.

- Como passar **parâmetros** para uma função composable (ex: `name` e `age`).
- Como evitar a sobreposição de elementos utilizando o **layout** `Column`.
- O que é a classe **MainActivity** e qual seu papel no app Android.
- Como estruturar o conteúdo visual dentro do `setContent {}` usando o **Jetpack Compose**.
- O que é e como usar o componente `Surface` como container visual.
- Como aplicar **cor de fundo** com `color = MaterialTheme.colorScheme.primary`.
- Como fazer com que o componente ocupe todo o espaço disponível com `Modifier.fillMaxSize()`.
- Como controlar altura e largura com `Modifier.fillMaxHeight()` e `fillMaxWidth()`.
- Como aplicar espaçamento interno com `Modifier.padding(all = 50.dp)`.
- O que significa **dp (density-independent pixels)** e por que essa unidade é usada na interface Android.

Abaixo está o **código completo** do nosso primeiro aplicativo com Jetpack Compose:

```
package com.example.primeiroprojeto

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.fillMaxHeight
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.padding
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 androidx.compose.ui.unit.dp
import com.example.primeiroprojeto.ui.theme.PrimeiroProjetoTheme
```

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            PrimeiroProjetoTheme {
                Surface(
                    modifier = Modifier.fillMaxHeight().fillMaxWidth().padding(all = 50
                    color = MaterialTheme.colorScheme.primary
                ) {
                    Greeting(name = "Bruno")
                }
            }
        }
    }
}

@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Text(
        text = "Hello $name!",
        modifier = modifier
    )
}

@Preview
@Composable
fun ShowAge(age: Int = 12) {
    Text(text = age.toString())
}

@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    PrimeiroProjetoTheme {
        Column() {
            Greeting("Android")
            ShowAge(age = 34)
        }
```

```
    }
}
```

Na proxima apostila veremos como criar uma interface simples com elementos `Mutables`