Jetpack

Compose Lifecycle

E SUA RELAÇÃO COM O ANDROID PADRÃO

Centro de Informática
UFPE

Relembrando

Lifecycle

- São **diferentes estados** no **ciclo de vida** de um **aplicativo**

- **Activity** fornece vários **callbacks** que permitem que a atividade saiba quando um **estado muda** ou que o **sistema está criando**
  - Programar como a atividade deve se comportar quando o usuário sai e retorna dela

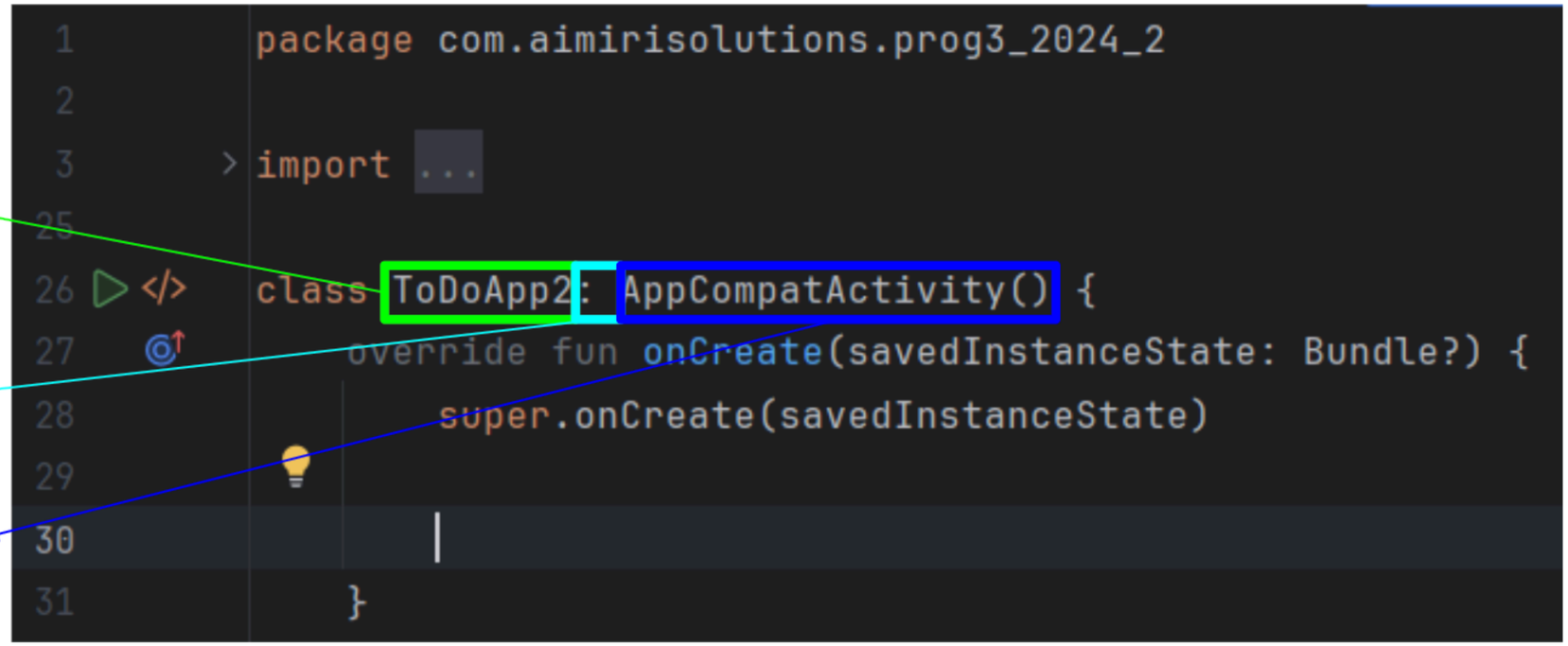
Relembrando

Activity

Nome da classe que se tornará uma activity

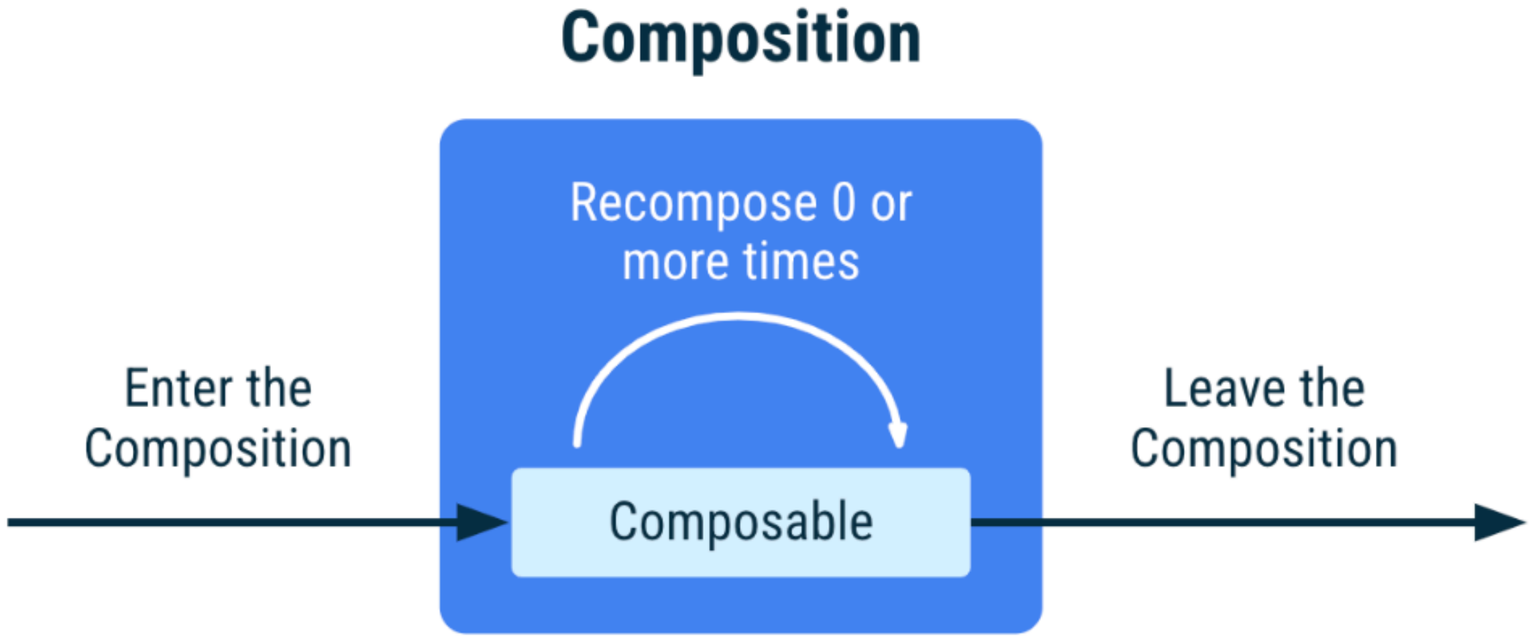
Herdando de ...

Classe Activity Pai



Compose

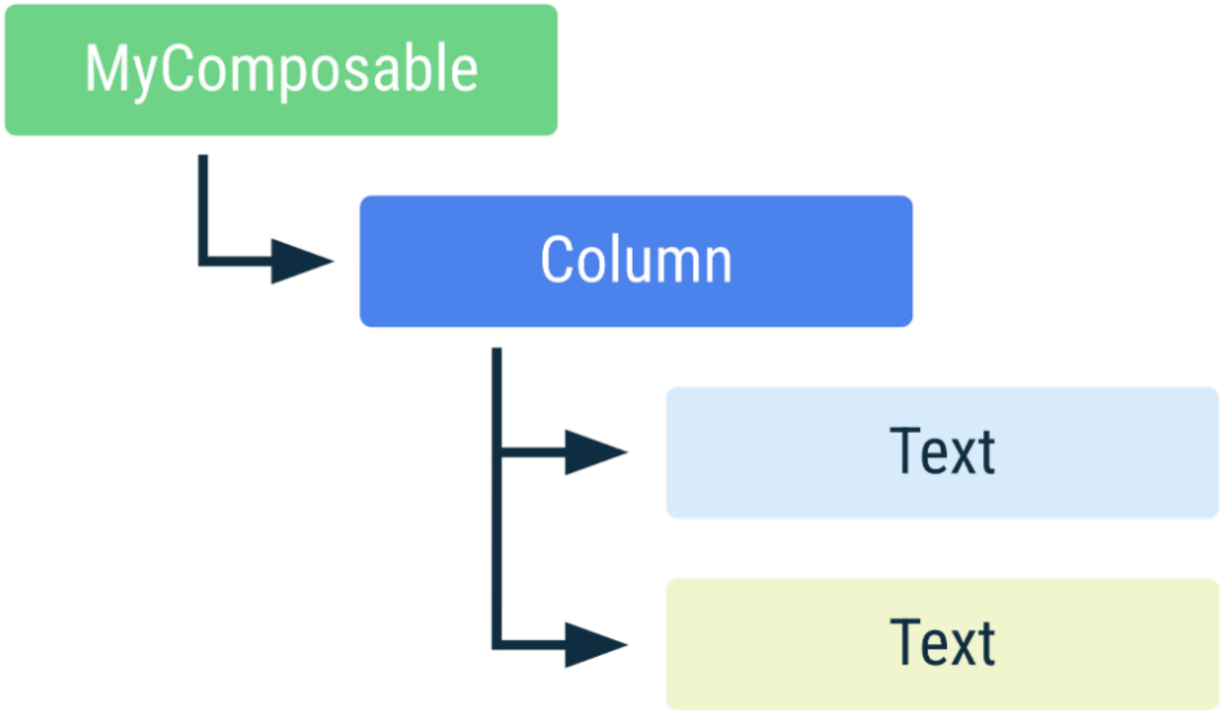
Composables não possuem os métodos de ciclo de vida como onCreate(), onStart(), onResume(), etc. Isso porque eles são funções que descrevem a UI, e não componentes com ciclo de vida próprio como as Activities.



Anatomia

Exemplo de anatomia de uma função Compose:

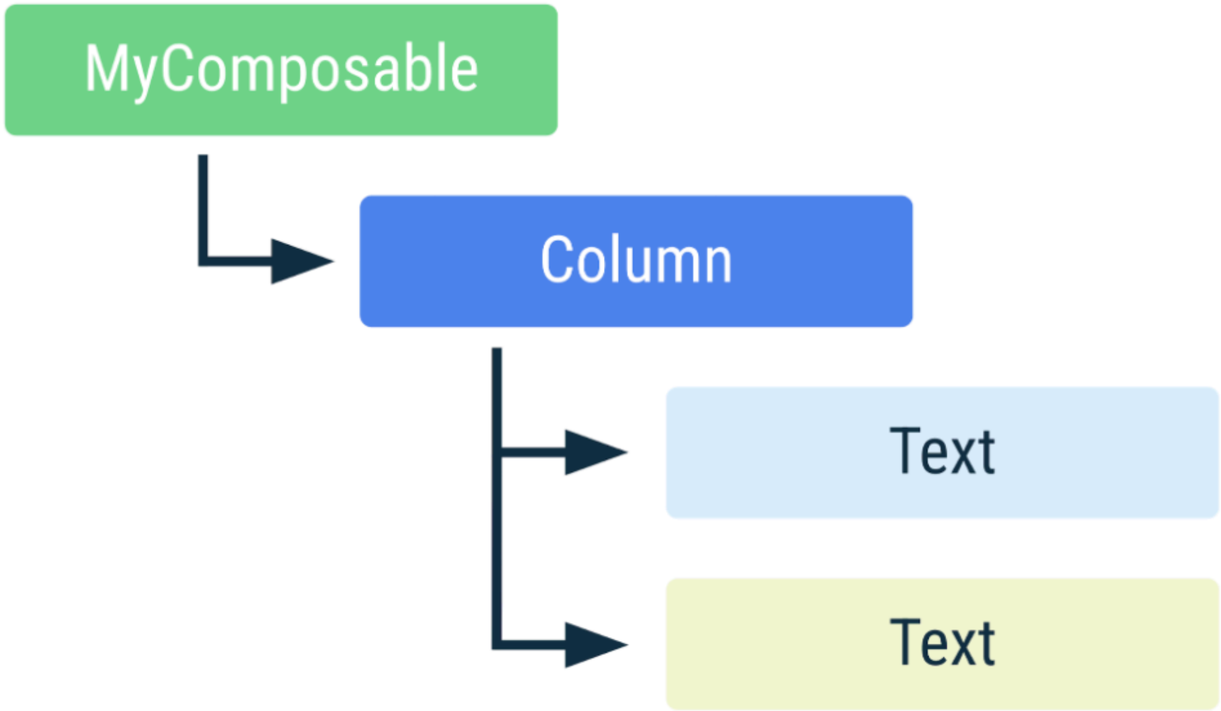
```
@Composable
fun MyComposable() {
    Column {
        Text("Hello")
        Text("World")
    }
}
```



Anatomia

Exemplo de anatomia de uma função Compose:

```
@Composable
fun MyComposable() {
    Column {
        Text("Hello")
        Text("World")
    }
}
```



Lifecycle

Pontos importantes:

1. Caso uma função Composable seja chamada várias vezes, diversas instâncias serão colocadas na Composição. **Cada chamada tem um ciclo de vida próprio**.
2. A instância de uma função Composable na composição é identificada pelo local de chamada. O compilador Compose considera que cada local de chamada é distinto. Chamar funções de vários locais de chamadas criará diversas instâncias da mesma função na composição.

Lifecycle

Pontos importantes:

Se durante a recomposição, a função composable chama funções diferentes das chamadas na composição anterior, o **Compose identificará quais composables foram chamados ou não**, e para os que foram chamadas, o Compose evitará a recomposição caso os inputs não tenham mudado. Veja o exemplo a seguir.

Lifecycle

Exemplo:

```
@Composable
fun LoginScreen(showError: Boolean) {
    if (showError) {
        LoginError()
    }
    LoginInput() // This call site affects where LoginInput is placed in Composition
}

@Composable
fun LoginInput() { /* ... */ }

@Composable
fun LoginError() { /* ... */ }
```

```kotlin
@Composable
fun LoginScreen(showError: Boolean) {
    if (showError) {
        LoginError()
    }
    LoginInput()
}

@Composable
fun LoginInput() { /* ... */ }

@Composable
fun LoginError() { /* ... */ }
```

Quando o parâmetro showError muda, acontece a recomposição. Pois a função Composable chama outra função Composable que não havia sido chamada inicialmente na sua construção

Lifecycle

Efeitos Colaterais

Um efeito colateral é uma mudança no estado do app que acontece fora do escopo de uma função de composição.

LaunchedEffect

É executado quando entra na composição e é automaticamente cancelado quando ele sai da composição. É ideal para efeitos colaterais que precisam ser disparados uma vez ou sempre que certas dependências mudam.

Lifecycle

LaunchedEffect

Pode ser comparado à lógica que tradicionalmente seria colocada em onStart() ou onResume() para recursos que precisam ser iniciados com a visibilidade da UI.

A principal diferença é que LaunchedEffect opera no escopo de um Composable específico, enquanto onStart() e onResume() operam no escopo da Activity inteira.

Lifecycle

LaunchedEffect

```
@Composable
fun UserProfile(viewModel: UserViewModel) {
    val userName by viewModel.userName.collectAsState()

    LaunchedEffect(Unit) {
        viewModel.loadUserProfile()
    }

    Text(text = "Bem-vindo, $userName!")
}
```

Lançado apenas uma vez

Carrega dados do usuário através do viewModel

Lifecycle

LaunchedEffect

```
@Composable
fun LoginScreen(viewModel: LoginViewModel, navController: NavController) {
    val isLoggedIn by viewModel.isLoggedIn.collectAsState()

    LaunchedEffect(isLoggedIn) {
        if (isLoggedIn) {
            navController.navigate("home_route") {
                popUpTo("login_route") { inclusive = true }
            }
        }
    }

    // ... UI de login
```

Verifica se o usuário está logado e redireciona para tela home

Lifecycle

DisposableEffect

É usado para efeitos colaterais que precisam de limpeza quando o Composable sai da composição. Ele permite que você registre um bloco de código que será executado quando o efeito for "descartado" (dispose). Isso é crucial para evitar vazamentos de memória ou comportamentos indesejados.

Exemplo: Gerenciamento de Conexões de Rede ou Banco de Dados

Lifecycle

DisposableEffect

```
@Composable
fun DatabaseConnectionWatcher(databaseClient: DatabaseClient) {
    DisposableEffect(databaseClient) {
        databaseClient.connect() // Conecta ao banco de dados

        onDispose {
            databaseClient.disconnect() // Desconecta para liberar recursos
            Log.d("DBWatcher", "Conexão de banco de dados fechada.")
        }
    }

    Text("Conectado ao Banco de Dados")
}
```

Lifecycle

rememberSaveable

É uma função Composable que armazena o valor do seu estado para que ele possa ser restaurado após uma mudança de configuração (como rotação de tela) ou um processo ser encerrado e reiniciado pelo sistema. É similar ao Bundle passado para onCreate()

Lifecycle

rememberSaveable

Exemplo de uso: Texto digitado sobrevive a rotação de tela em TextField

```
@Composable
fun UserInputScreen() {
    var text by rememberSaveable { mutableStateOf("") } // O estado é salvo

    TextField(
        value = text,
        onValueChange = { newText -> text = newText },
        label = { Text("Digite seu nome") }
    )
}
```

Lifecycle

remember

É uma função Composable que armazena um valor na memória para que ele possa ser lembrado através de recomposições do Composable.

Pontos importantes:

1. O valor é "esquecido" se o Composable for removido da composição.
2. Ele não sobrevive a mudanças de configuração (a menos que combinado com rememberSaveable)

Lifecycle

remember

Exemplo de uso:
Controlar a visibilidade de um diálogo

```
@Composable
fun ActionButton() {
    var showDialog by remember { mutableStateOf(false) } // Temp

    Button(onClick = { showDialog = true }) {
        Text("Mostrar Diálogo")
    }

    if (showDialog) {
        AlertDialog(
            onDismissRequest = { showDialog = false },
            title = { Text("Alerta") },
            text = { Text("Esta é uma mensagem de alerta.") },
            confirmButton = {
                Button(onClick = { showDialog = false }) {
                    Text("OK")
                }
            }
        )
    }
}
```

Lifecycle

Agora que não temos mais uma Activity para salvar os dados diretamente, o que utilizar?

Lifecycle

ViewModel

- Sobrevive a Mudanças de Configuração: Armazena e gerencia dados da UI, persistindo através de rotações de tela ou recriação da Activity.

- Ciclo de Vida Independente: Seu ciclo de vida é isolado da Activity, sendo preservado e fornecido à próxima instância da Activity.

- Separação de Preocupações: Ajuda a separar a lógica de negócios e os dados da UI, tornando o código mais limpo e testável.

- Integração Simplificada: Obtido em Composables via viewModel()

Lifecycle

ViewModel + Tela Composable

- **Separação de Preocupações**: Composables focam em UI, ViewModel gerencia estado e lógica de negócio.
- **Persistência de Dados**: ViewModel sobrevive a mudanças de configuração (ex: rotação), evitando perda de dados e recarregamentos.
- **Testabilidade Aprimorada**: ViewModel pode ser testado isoladamente, sem UI.
- **Reusabilidade de Lógica**: Lógica do ViewModel pode ser reutilizada em várias telas.
- **Melhor Desempenho**: Evita recargas desnecessárias de dados, resultando em UI mais rápida e fluida.

Lifecycle ViewModel

```
class MyDataViewModel : ViewModel() {
    // _count é um MutableState que pode ser modificado internamente no ViewModel.
    private val _count = mutableStateOf(0)

    // A UI irá observar este State e será recomposta quando seu valor mudar.
    val count: State<Int> = _count

    // Função para incrementar o contador. Esta lógica reside no ViewModel.
    fun incrementCount() {
        _count.value++
        Log.d("MyDataViewModel", "Contador incrementado para: ${_count.value}")
    }
}
```

```kotlin
@Composable
fun CounterScreen(viewModel: MyDataViewModel = viewModel()) {
    // Sempre que 'count' no ViewModel muda, 'currentCount' é atualizado
    val currentCount = viewModel.count.value

    Column(
        modifier = Modifier.fillMaxSize(),
    ) {
        Text(
            text = "Contador: $currentCount",
        )

        Button(onClick = {
            // Quando o botão é clicado, chama a função incrementCount() no ViewModel.
            // O ViewModel então atualiza seu estado (_count), o que aciona a recomposição.
            viewModel.incrementCount()
        }) {
            Text("Incrementar")
        }
    }
}
```

Tela Compose

Obrigado!

Centro de
Informática
UFPE