Universidade POSITIVO

Desenvolvimento de Aplicativos Móveis

Professor Maurício Buess

mbuess@up.edu.br

https://github.com/mauriciobuess

Introdução ao Desenvolvimento Android:

- Complementação do State
- Ciclo de vida da Activity

State (estado):

- O state (estado) é a base da reatividade da UI.
- A UI é composta em função do estado atual, e qualquer mudança no estado faz com que a UI seja atualizada automaticamente.
- **Estado Imutável** refere-se a dados que não mudam uma vez definidos.
  - Normalmente, usa-se estado imutável para informações que não dependem da interação do usuário ou de processos dinâmicos.
  - **Quando usar:**
    - Dados fixos: Informações que são constantes durante o ciclo de vida da UI.
    - Constantes de configuração: Como textos de botões ou rótulos.

State (estado):

- Exemplo de **Estado Imutável:**

```
@Composable
fun EntradaTela() {
    val entradaMsg = "Bem-vindo ao Aplicativo!"

    // UI
    Text(text = entradaMsg)
}
```

* Aqui, entradaMsg é um estado imutável porque seu valor não muda.

State (estado):

- **Estado Mutável -** usado para dados que podem mudar ao longo do tempo, como entradas do usuário ou resultados de ações.

  - Quando usar:

    - Dados dinâmicos: Informações que mudam com interações do usuário, como contadores ou campos de entrada.
    - Interatividade: Qualquer dado que afete a interação do usuário com a aplicação.

Universidade POSITIVO

State (estado):

- Exemplo de **Estado Mutável**

```
@Composable
fun Counter() {
    var count by remember { mutableStateOf(0) }

    Column {
        Text(text = "Contagem: $count")
        Button(onClick = { count++ }) {
            Text("Incrementar")
        }
    }
}
```

Aqui **count** é um estado mutável, pois seu valor muda toda vez que o botão é clicado.

State Hoisting

- Padrão que move o estado para um nível superior na hierarquia de composables para permitir que diferentes funções composables compartilhem e atualizem o mesmo estado.

```
@Composable
fun CounterScreen() {
    var count by remember { mutableStateOf(0) }
     // Hoisted state
    Counter(count = count, onIncrement = { count++ })
}
```

```
@Composable
fun Counter(count: Int, onIncrement: () -> Unit) {
    Column {
        Text(text = "Contagem: $count")
        Button(onClick = onIncrement) {
            Text("Incrementar")
        }
    }
}
```

O estado count é mantido em CounterScreen, mas Counter recebe o estado como um parâmetro.

State Hoisting

- @Composable – Anotation que sinaliza a declaração de uma U.I.
- fun Counter(count: Int, onIncrement: () -> Unit)
  - count: Int: Um inteiro que representa o valor atual do contador.
  - onIncrement: () -> Unit: Uma função lambda que não recebe parâmetros e não retorna nada, usada para incrementar o contador. Esse lambda será chamado quando o botão for clicado.
- Button(onClick = onIncrement) { Text("Incrementar") }
  - Parâmetro onClick: onClick é uma função lambda que define o que acontece quando o botão é clicado. No caso, onIncrement é passado como o lambda a ser executado, que geralmente incrementará o valor do contador.
  - Corpo do Button: Dentro do Button, há um composable Text que exibe o texto "Incrementar". Este texto é o que aparece dentro do botão.

Funcionamento Geral

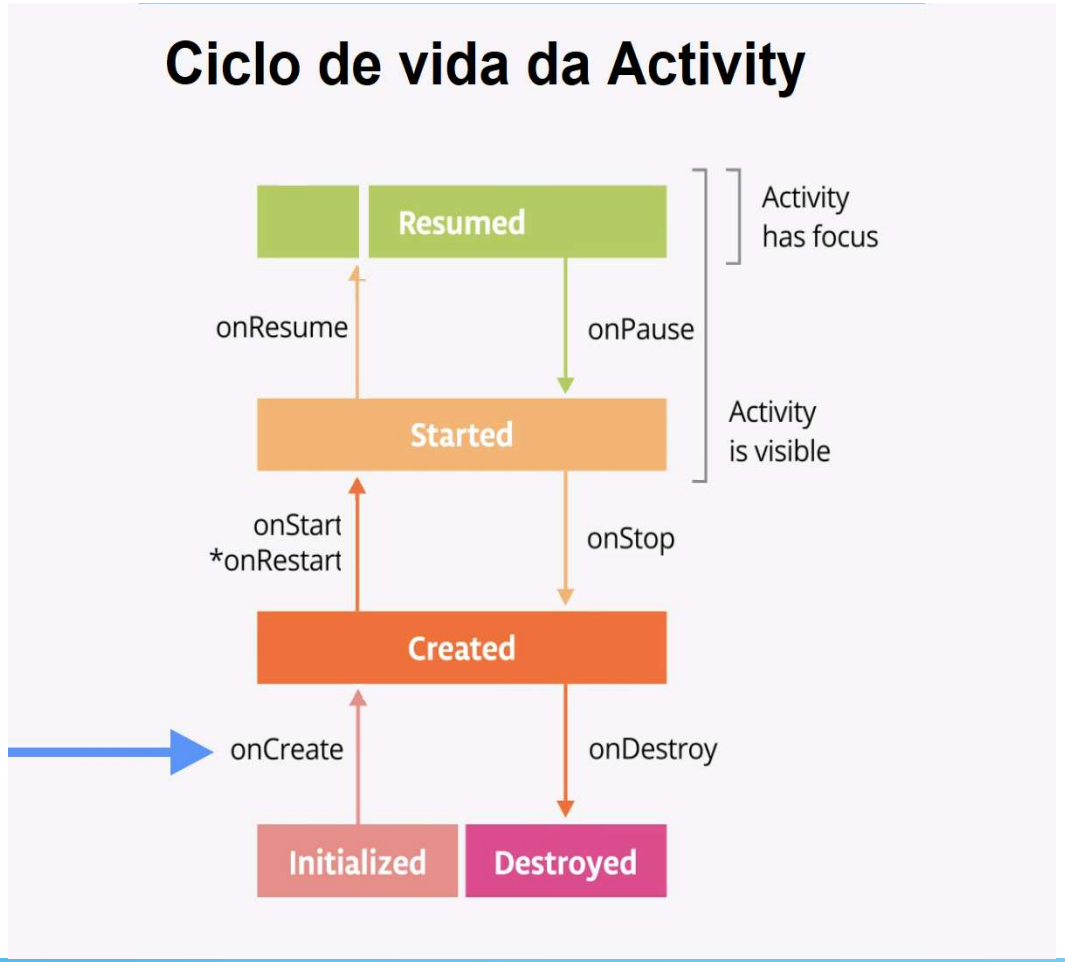
- A função Counter cria uma interface de usuário com um texto que mostra o valor atual do contador e um botão que, quando clicado, chama a função onIncrement para incrementar o contador.

- A função onIncrement é passada como um argumento e pode ser definida em outro lugar, normalmente na função onde o Counter é usado, para alterar o estado do contador e causar uma nova renderização da UI.

Universidade POSITIVO

Ciclo de Vida da Activity

- É essencial para o gerenciamento correto da UI e dos recursos;
- Diz respeito ao conjunto de métodos e estados que uma Activity pode ter ao longo de sua existência;
- Tais métodos permitem que o desenvolvedor controle o que acontece com a Activity em momentos específicos tais como quando ela é criada, iniciada, pausada, retomada ou destruída;
- São funções “callback”



Como Funciona o Ciclo de Vida da Activity

- A Activity passa por várias fases ao longo de sua vida;
- O sistema Android chama métodos específicos para gerenciar essas fases;
- A transição entre essas fases pode ocorrer devido a ações do usuário, mudanças de configuração ou eventos do sistema.

Métodos do Ciclo de Vida da Activity

- **Criação (onCreate)**
- Método: onCreate(Bundle savedInstanceState)

- Descrição: Chamado quando a Activity é criada pela primeira vez.
  - Aqui o desenvolvedor deve configurar a Activity, definir o layout e inicializar variáveis.
  - É o lugar para realizar a configuração inicial, como vincular a interface do usuário com o código e restaurar o estado salvo se necessário.
- Baseado em: Activity.onCreate()

Métodos do Ciclo de Vida da Activity

- **Início (onStart)**
- Método: onStart()

- Descrição: Chamado quando a Activity está prestes a se tornar visível para o usuário.
  - Neste estágio, a Activity está começando a se tornar visível, mas ainda não é interativa.

- Baseado em: Activity.onStart()

Métodos do Ciclo de Vida da Activity

- **Retomada (onResume)**
- Método: onResume()

- Descrição: Chamado quando a Activity começa a interagir com o usuário.
  - A Activity está agora em primeiro plano e totalmente interativa.
  - Aqui que se inicia processos que devem ser executados enquanto a Activity está visível e ativa, como iniciar animações ou ouvir eventos do usuário.
- Baseado em: Activity.onResume()

Métodos do Ciclo de Vida da Activity

- **Pausa (onPause)**
- Método: onPause()
- Descrição: Chamado quando a Activity está prestes a perder o foco, mas ainda está visível.
  - Usa-se este método para parar operações que não devem continuar quando a Activity não está em primeiro plano, como pausar animações ou gravações.
  - Se a Activity não for visível, mas não for destruída, ela pode ser reexibida posteriormente sem passar novamente pelo processo de criação.
- Baseado em: Activity.onPause()

Métodos do Ciclo de Vida da Activity

- **Parada (onStop)**
- Método: onStop()

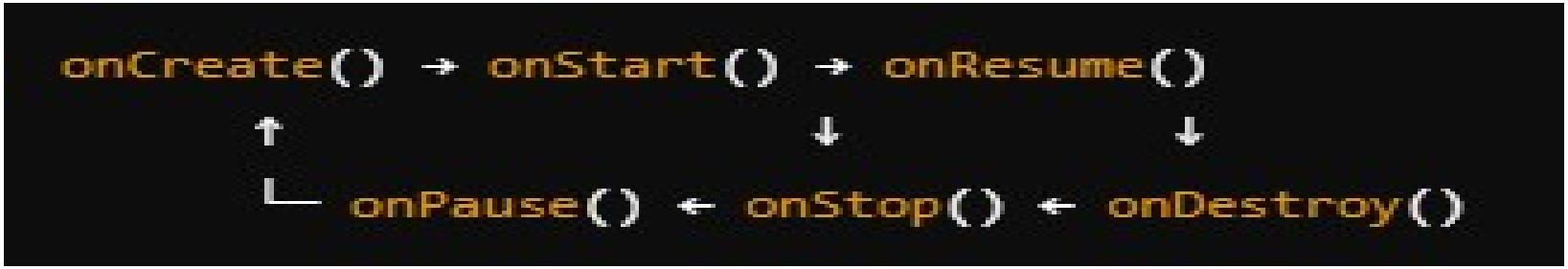
- Descrição: Chamado quando a Activity não está mais visível para o usuário.
  - É neste estágio, que se deve liberar recursos que não são necessários enquanto a Activity não está visível.
  - Pode ser usado para salvar o estado ou liberar recursos mais pesados.
- Baseado em: Activity.onStop()

Métodos do Ciclo de Vida da Activity

- **Destruição (onDestroy)**
- Método: onDestroy()

- Descrição: Chamado quando a Activity está prestes a ser destruída, seja porque o usuário a fechou ou o sistema precisa liberar recursos.
  - Aqui se deve realizar qualquer limpeza final antes que a Activity seja destruída, como liberar recursos pesados ou salvar o estado final.
- Baseado em: Activity.onDestroy()

Métodos do Ciclo de Vida da Activity



- **onCreate** é chamado quando a Activity é criada pela primeira vez.
- **onStart** é chamado quando a Activity está prestes a se tornar visível.
- **onResume** é chamado quando a Activity começa a interagir com o usuário.
- **onPause** é chamado quando a Activity está prestes a perder o foco.
- **onStop** é chamado quando a Activity não está mais visível.
- **onDestroy** é chamado quando a Activity está prestes a ser destruída.

Ciclo de Vida da Activity com Jetpack Compose

- Com Jetpack Compose, a abordagem ao ciclo de vida é um pouco diferente, pois o Compose é baseado em funções composables que descrevem a UI.
- No entanto, os conceitos fundamentais do ciclo de vida da Activity ainda se aplicam.
- O desenvolvedor deve usar a Activity e seus métodos do ciclo de vida para gerenciar recursos, estado e interações com o sistema, enquanto Jetpack Compose cuida da construção da UI e da reatividade a mudanças de estado.
- Pode-se usar a função **LaunchedEffect** para responder a mudanças no ciclo de vida em um composable:

```
@Composable
fun Initial_UI() {
    val context = LocalContext.current
    val lifecycleOwner = LocalLifecycleOwner.current

    LaunchedEffect(lifecycleOwner) {
        val lifecycle = lifecycleOwner.lifecycle
        val observer = LifecycleEventObserver { _, event ->
            when (event) {
                Lifecycle.Event.ON_START -> {
                    // A Activity está começando a ser visível
                }
                Lifecycle.Event.ON_STOP -> {
                    // A Activity não está mais visível
                }
                // Outros eventos do ciclo de vida
            }
        }
        lifecycle.addObserver(observer)
        // Remova o observer quando o
        //  composable for removido
        onDispose { lifecycle.removeObserver(observer) }
    }
}
```

Initial_UI()

- O código configura um observer para eventos do ciclo de vida da Activity dentro de um composable. Usando LaunchedEffect para configurar e limpar o observer de forma segura e adequada.
- O observer reage a mudanças no ciclo de vida da Activity, como quando a Activity se torna visível (ON_START) ou não está mais visível (ON_STOP), garantindo que o observer seja limpo corretamente quando o composable é removido.
- **LaunchedEffect** é usado para observar eventos do ciclo de vida da Activity dentro de um composable.

LifecycleEventObserver permite ao App reagir a eventos específicos do ciclo de vida.

- **Obtendo o Contexto e o LifecycleOwner**

```
val context = LocalContext.current
val lifecycleOwner = LocalLifecycleOwner.current
```

- **LocalContext.current**: Fornece o contexto atual da aplicação, que pode ser necessário para acessar recursos, iniciar atividades, etc. No código fornecido, context não é usado diretamente, mas pode ser útil para outras operações que precisam do contexto da aplicação.
- **LocalLifecycleOwner.current**: Fornece o LifecycleOwner atual, que é a entidade que possui o ciclo de vida e é responsável por notificar sobre mudanças de estado, como onStart, onStop, etc. Normalmente, isso é a Activity ou Fragment que contém o composable.

- **Usando LaunchedEffect**

LaunchedEffect(lifecycleOwner) {

- **LaunchedEffect** é um composable que executa um bloco de código em um coroutine scope quando o composable é lançado. Ele é executado no início e sempre que a chave fornecida (lifecycleOwner aqui) muda.

- O que faz? A função dentro do LaunchedEffect será executada quando o composable for exibido pela primeira vez e sempre que a chave (lifecycleOwner) mudar. Ele garante que o código dentro do bloco seja executado em uma coroutine segura.

- **Configurando o LifecycleEventObserver**

```
val lifecycle = lifecycleOwner.lifecycle
val observer = LifecycleEventObserver { _, event ->
    when (event) {
        Lifecycle.Event.ON_START -> {
            // A Activity está começando a ser visível
        }
        Lifecycle.Event.ON_STOP -> {
            // A Activity não está mais visível
        }
        // Outros eventos do ciclo de vida
    }
}
```

- **lifecycleOwner.lifecycle**: Obtém o ciclo de vida associado ao LifecycleOwner atual, fornecemdo eventos e estados vinculados à vida da Activity .
- **LifecycleEventObserver**: Cria um observer que reage a eventos do ciclo de vida. O lambda passado para LifecycleEventObserver é chamado sempre que um evento de ciclo de vida ocorre.
- **Eventos**: when reage aos eventos:
- **Lifecycle.Event.ON_START**: pode iniciar tarefas ou processos que devem ocorrer enquanto a Activity está visível
- **Lifecycle.Event.ON_STOP**: A Activity não está mais visível - parar tarefas ou liberar recursos que não são necessários.

- **Adicionando e Removendo o Observer**

```
lifecycle.addObserver(observer)
onDispose { lifecycle.removeObserver(observer) }
```

- **lifecycle.addObserver(observer)**: Adiciona o LifecycleEventObserver ao ciclo de vida. Isso garante que o observer receba notificações sobre os eventos do ciclo de vida.
- **onDispose { lifecycle.removeObserver(observer) }**: Registra uma função de limpeza que será chamada quando o composable for removido da composição. Isso garante que o observer seja removido e não receba mais atualizações quando o composable não estiver mais em uso, evitando vazamentos de memória e comportamento indesejado.

Exemplo: App Lista Tarefas

```
import android.os.Bundle
import android.util.Log
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.activity.viewModels
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.material3.Button
import androidx.compose.material3.Text
import androidx.compose.runtime.*
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Surface
import androidx.lifecycle.ViewModel
```

```
import com.up_des_mobile.listatarefas.ui.theme.ListaTarefasTheme

class MainActivity : ComponentActivity() {
    private val viewModel: TaskListViewModel by viewModels()
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        Log.d("Ciclo de Vida", "onCreate chamada")
        logTasks("onCreate")
        setContent {
            ListaTarefasTheme {
                Surface(color = MaterialTheme.colorScheme.background) {
                    TaskListScreen(viewModel)
                }
            }
        }
    }
}
```

Exemplo: App Lista Tarefas (cont)

```
override fun onStart() {
    super.onStart()
    Log.d("Ciclo de Vida", "onStart chamada")
    logTasks("onStart")
}
override fun onResume() {
    super.onResume()
    Log.d("Ciclo de Vida", "onResume chamada")
    logTasks("onResume")
}
override fun onPause() {
    super.onPause()
    Log.d("Ciclo de Vida", "onPause chamada")
    logTasks("onPause")
}
override fun onStop() {
    super.onStop()
    Log.d("Ciclo de Vida", "onStop chamada")
    logTasks("onStop")
}
override fun onDestroy() {
    super.onDestroy()
    Log.d("Ciclo de Vida", "onDestroy chamada")
    logTasks("onDestroy")
}
private fun logTasks(stage: String) {
    Log.d("Lista de Tarefas", "Lista de Tarefas $stage:
${viewModel.tasks.joinToString()}")
}
}
```

Exemplo: App Lista Tarefas (cont)

```
class TaskListViewModel : ViewModel() {
    private val _tasks = mutableStateListOf<String>()
    val tasks: List<String> get() = _tasks

    fun addTask() {
        val taskDescription = "Tarefa ${_tasks.size + 1}"
        _tasks.add(taskDescription)
    }
}
```

```
@Composable
fun TaskListScreen(viewModel: TaskListViewModel) {
    val tasks by remember { derivedStateOf { viewModel.tasks } }
    Column {
        Button(onClick = {
            viewModel.addTask()
        }) {
            Text("Inclui Tarefa")
        }
        Text(text = "Lista de Tarefas:")
        LazyColumn {
            items(tasks) { task ->
                Text(text = task)
            }
        }
    }
```

Exemplo: App Lista Tarefas (cont)

```
    LaunchedEffect(tasks) {
        Log.d("Lista de Tarefas", "Tarefas atuais: ${tasks.joinToString()}")
    }
}
```

Fonte – passo a passo

```
class MainActivity : ComponentActivity() {
    private val viewModel: TaskListViewModel by viewModels()
```

- **ComponentActivity**: MainActivity estende ComponentActivity, classe base para atividades em Jetpack Compose.
- **viewModel**: Atributo viewModel é uma instância do TaskListViewModel, obtida através do delegador by viewModels(). Isso permite que o ViewModel seja gerenciado de forma adequada pela arquitetura do Android.

- Ciclo de Vida da Activity
- Cada método do ciclo de vida da Activity chama logTasks, que registra o conteúdo da lista de tarefas no Logcat durante diferentes estágios do ciclo de vida:

Fonte – passo a passo

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    Log.d("Ciclo de Vida", "onCreate chamada")
    logTasks("onCreate")

    setContent {
        ListaTarefasTheme {
            Surface(color = MaterialTheme.colorScheme.background) {
                TaskListScreen(viewModel)
            }
        }
    }
}
```

- **onCreate**: Chamado quando a Activity é criada. Aqui, além de registrar o estado inicial das tarefas, o método setContent define o layout da Activity usando o Jetpack Compose. ListaTarefasTheme aplica o tema e Surface fornece um fundo para a tela.

Fonte – passo a passo

```
override fun onStart() {
    super.onStart()
    Log.d("Ciclo de Vida", "onStart chamada")
    logTasks("onStart")
}
override fun onResume() {
    super.onResume()
    Log.d("Ciclo de Vida", "onResume chamada")
    logTasks("onResume")
}
override fun onPause() {
    super.onPause()
    Log.d("Ciclo de Vida", "onPause chamada")
    logTasks("onPause")
}...
```

- **onStart, onResume, onPause, onStop, onDestroy**: Métodos do ciclo de vida da Activity que registram o estado da lista de tarefas em diferentes momentos da vida da Activity. Isso ajuda a ver como o estado muda em resposta a diferentes eventos.

Fonte – passo a passo

```
private fun logTasks(stage: String) {
    Log.d("Lista de Tarefas", "Lista de Tarefas $stage:
 ${viewModel.tasks.joinToString()}")
}
```

- **logTasks**: Método auxiliar que usa o Log.d para imprimir o conteúdo atual da lista de tarefas no log. O stage indica em qual ponto do ciclo de vida da Activity o log foi gerado.

```
class TaskListViewModel : ViewModel() {
    private val _tasks = mutableStateListOf<String>()
    val tasks: List<String> get() = _tasks
    fun addTask() {
        val taskDescription = "Tarefa ${_tasks.size + 1}"
        _tasks.add(taskDescription)
    }
}
```

- **_tasks**: Lista de tarefas gerenciada como um estado mutável (mutableStateListOf). Isso permite que a lista seja observada e atualizada reativamente.
- **tasks**: Propriedade de somente leitura que expõe a lista de tarefas sem permitir modificações externas.
- **addTask**: Método para adicionar uma nova tarefa à lista, com uma descrição baseada no tamanho atual da lista.

Fonte – passo a passo

```
@Composable
fun TaskListScreen(viewModel: TaskListViewModel) {
    val tasks by remember { derivedStateOf { viewModel.tasks } }
    Column {
        Button(onClick = {
            viewModel.addTask()
        }) {
            Text("Inclui Tarefa")
        }
        Text(text = "Lista de Tarefas:")
        LazyColumn {
            items(tasks) { task ->
                Text(text = task)
            }
        }
    }
    LaunchedEffect(tasks) {
        Log.d("Lista de Tarefas", "Tarefas atuais: ${tasks.joinToString()}")
    }
}
```

- **tasks by remember** { derivedStateOf { viewModel.tasks } }: Observa o estado da lista de tarefas. remember evita recomposições desnecessárias, e derivedStateOf cria um estado que é recomputado sempre que a lista de tarefas muda.
- **LazyColumn**: Exibe a lista de tarefas de forma eficiente, com rolagem se necessário.
- **LaunchedEffect(tasks)**: Executa um efeito colateral sempre que a lista de tarefas muda, registrando o conteúdo atual das tarefas no log.

Finalmente...

- **MainActivity**: Gerencia o ciclo de vida da Activity e configura a tela com TaskListScreen. Registra o estado da lista de tarefas em diferentes momentos do ciclo de vida.

- **TaskListViewModel**: Mantém e gerencia a lista de tarefas como um estado reativo.

- **TaskListScreen**: Define a UI e exibe a lista de tarefas, além de registrar o estado das tarefas sempre que a lista muda.