

Desenvolvimento Mobile

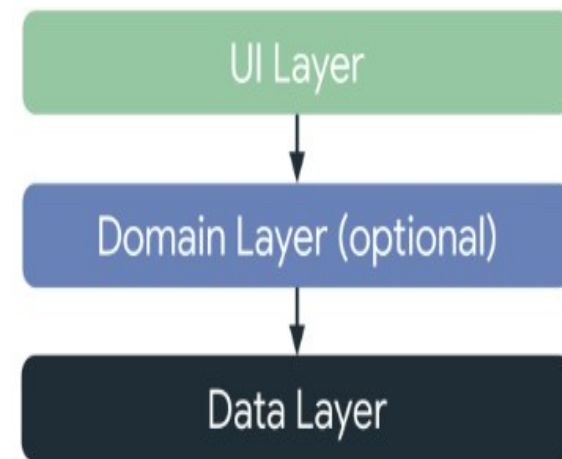
Professor Maurício Buess

mbuess@up.edu.br



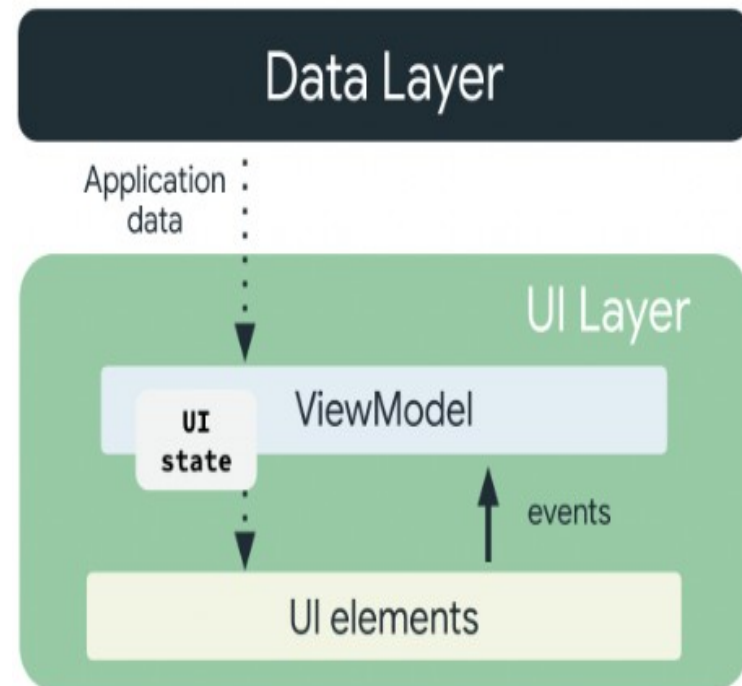
- Objetivos:
 - Compreensão da UDF (unidirectional data flow)
 - Navegação entre telas
 - `getSystemService`

- Cada App deve ter, pelo menos, duas camadas:
- Camada UI: uma camada que exibe os dados do aplicativo na tela, mas é independente dos dados.
- Camada de dados: uma camada que armazena, recupera e expõe os dados do aplicativo.
- Pode haver uma 3ª camada, chamada camada de domínio, para simplificar e reutilizar as interações entre a UI e as camadas de dados (não será utilizada nesse momento).



As setas representam as dependências.

- Camada da IU
- A função da camada UI, ou camada de apresentação, é exibir os dados do aplicativo na tela. Sempre que os dados forem alterados devido a uma interação do usuário, como pressionar um botão, a IU deverá ser atualizada para refletir as alterações.
- A camada UI é composta pelos seguintes componentes:
- **Elementos da UI:** componentes que renderizam os dados na tela. Você cria esses elementos usando o Jetpack Compose.
- **State Holders:** componentes que contêm os dados, os expõem à IU e lidam com a lógica do aplicativo. Um exemplo de detentor de estado é ViewModel.



- Estado da IU
- A IU é o que o usuário vê, e o estado da IU é o que o aplicativo diz que ele deveria ver;
- A IU é a representação visual do estado da UI. Qualquer alteração no estado da IU é refletida imediatamente na IU;
- A IU é o resultado da vinculação de elementos da IU na tela ao estado da IU.



- **O Fluxo de Dados Unidirecional (UDF) segue um ciclo previsível:**

- 1) Eventos da UI são disparados:

- O usuário interage com a interface (por exemplo, clicando em um botão).
- A IU emite eventos para o ViewModel ou outro controlador de estado.

- 2) Manipulação de eventos:

- O ViewModel recebe os eventos e executa a lógica de negócios necessária.
- O estado é atualizado no ViewModel em resposta aos eventos.

- 3) Atualização do estado:

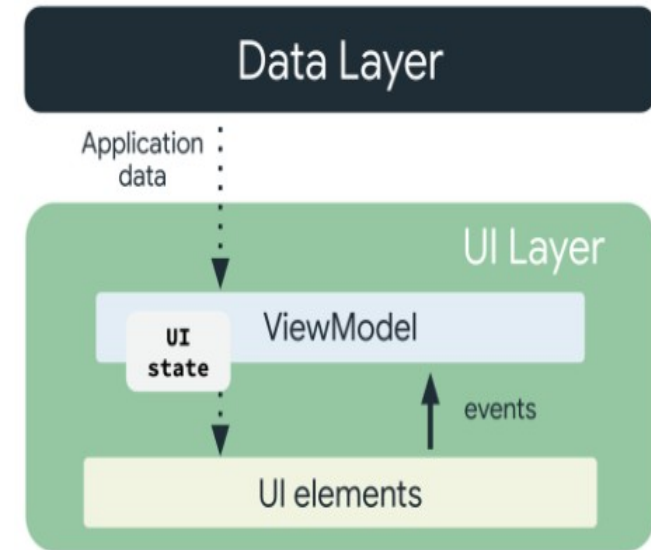
- A mudança de estado no ViewModel notifica os observadores (UI) sobre a mudança.

- 4) A UI se “re-renderiza”:

- A IU observa o estado e se re-renderiza automaticamente quando o estado muda, refletindo as mudanças para o usuário.

- Benefícios do Padrão UDF
 - Previsibilidade: Com o fluxo de dados unidirecional, é fácil seguir o caminho dos dados e entender como a UI é atualizada.
 - Depuração: Problemas de estado são mais fáceis de rastrear porque todas as mudanças de estado são centralizadas.
 - Reusabilidade: A separação entre a lógica de negócios (ViewModel) e a UI (Composables) facilita a reutilização e o teste.
- O padrão de fluxo de dados unidirecional no Jetpack Compose ajuda a criar aplicativos robustos e fáceis de manter, promovendo uma clara separação de responsabilidades e garantindo que o estado da aplicação seja gerenciado de forma previsível e consistente..

- O uso do padrão UDF para arquitetura de aplicativos tem as seguintes implicações:
- O ViewModel mantém e expõe o estado que a UI consome.
- O estado da UI são dados do aplicativo transformados pelo ViewModel.
- A UI notifica o ViewModel sobre eventos do usuário.
- O ViewModel trata das ações do usuário e atualiza o estado.
- O estado atualizado é retornado à interface do usuário para renderização.
- Este processo se repete para qualquer evento que cause uma mutação de estado.



- **Navegação com o Compose**

- O componente de navegação tem três partes principais:
 - NavController: responsável por navegar entre os destinos, ou seja, as telas do seu app.
 - NavGraph: mapeia os destinos de composição para navegar.
 - NavHost: elemento combinável que funciona como um contêiner para mostrar o destino atual do NavGraph.

- **Definir rotas para destinos no seu app**
- Rota:
 - A rota é um dos conceitos fundamentais de navegação em um app;
 - Uma rota é uma string correspondente a um destino.
 - Semelhante ao conceito de URL. Assim como um URL diferente mapeia para outra página em um site, uma rota é uma string que mapeia para um destino e serve como seu identificador exclusivo.
 - Um destino normalmente é um único elemento ou um grupo de elementos de composição correspondentes ao que o usuário vê.

• Adicionar um NavHost

- **NavHost** → elemento combinável que mostra outros destinos combinável, com base em uma determinada rota.
- navController: instância de classe NavController. É usado para navegar entre telas, por exemplo, chamando o método navigate() para navegar para outro destino. Pode-se buscar o NavController chamando rememberNavController() em uma função de composição.
- startDestination: uma rota de string que define o destino mostrado por padrão quando o app mostra o NavHost pela primeira vez.

NavHost (

navController ,

startDestination ,

modifier ,

) {

content

}

- **Adicionar um NavHost**

```
import androidx.compose.foundation.layout.padding
```

```
// variável navController que receberá rememberNavController  
val navController = rememberNavController()
```

```
// configuração do NavHost que vai trabalhar com a  
// variável navController e vai iniciar na Tela01  
NavHost(navController = navController, startDestination = "Tela01") {  
  
}
```

• Processar rotas no NavHost

- O NavHost usa um tipo de função para o próprio conteúdo, assim como outros elementos combináveis;
- Na função de conteúdo de um NavHost, chama-se a função `composable()`.
- A função `composable()` tem dois parâmetros obrigatórios.
 - `route`: uma string correspondente ao nome de uma rota. Ela pode ser qualquer string exclusiva.
 - `content`: aqui é possível chamar um elemento combinável que você queira mostrar no trajeto especificado.
- Chamar-se a função `composable()` uma vez para cada uma das rotas.

```
composable ( route ) {  
    content  
}
```

```
NavHost(navController = navController, startDestination = "Tela01") {  
    // configuração da função composable  
    // que vai montar a "pilha" de navegação  
  
    // 1a. rota (ou tela)  
    composable(route = "Tela01" ) {  
        Tela01(navController)  
    }  
  
    // 2a. rota (ou tela)  
    composable(route = "Tela02") {  
        Tela02(navController)  
    }  
}
```

- **Navegar entre rotas**

- Agora que as rotas foram definidas e mapeadas para elementos combináveis em um NavHost, é hora de navegar entre as telas.
- O `NavHostController`, (propriedade `navController` vinda de `rememberNavController()`), é o responsável pela navegação entre as rotas. No entanto, essa propriedade é definida no elemento combinável. É preciso ter uma maneira de acessá-lo nas diferentes telas do app.
- Basta transmitir `navController` como um parâmetro para cada elemento combinável.

- Navegar para outra rota

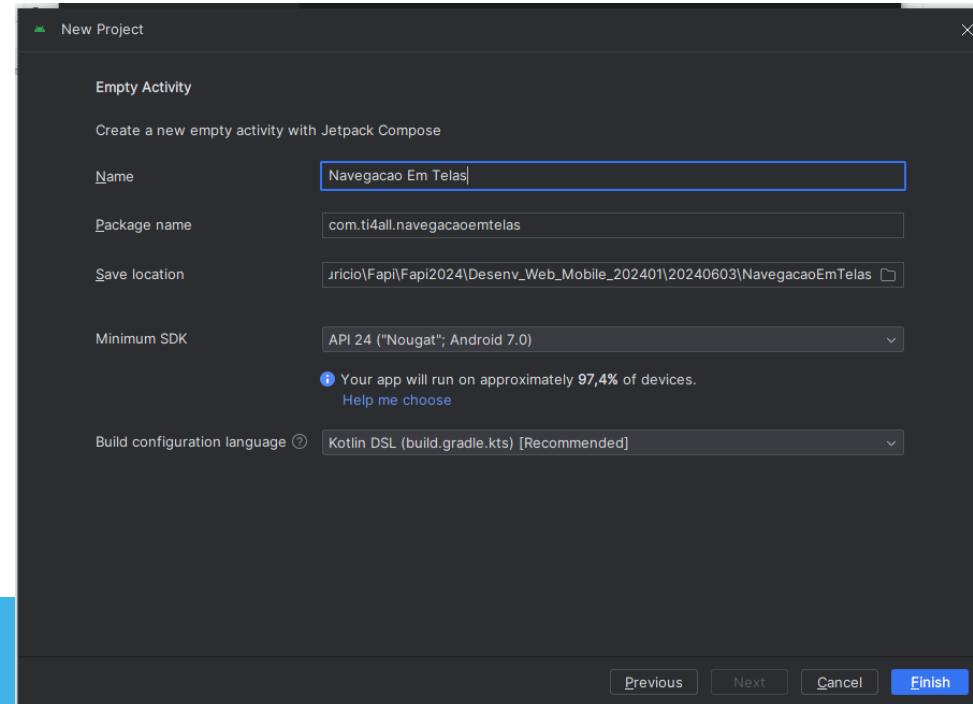
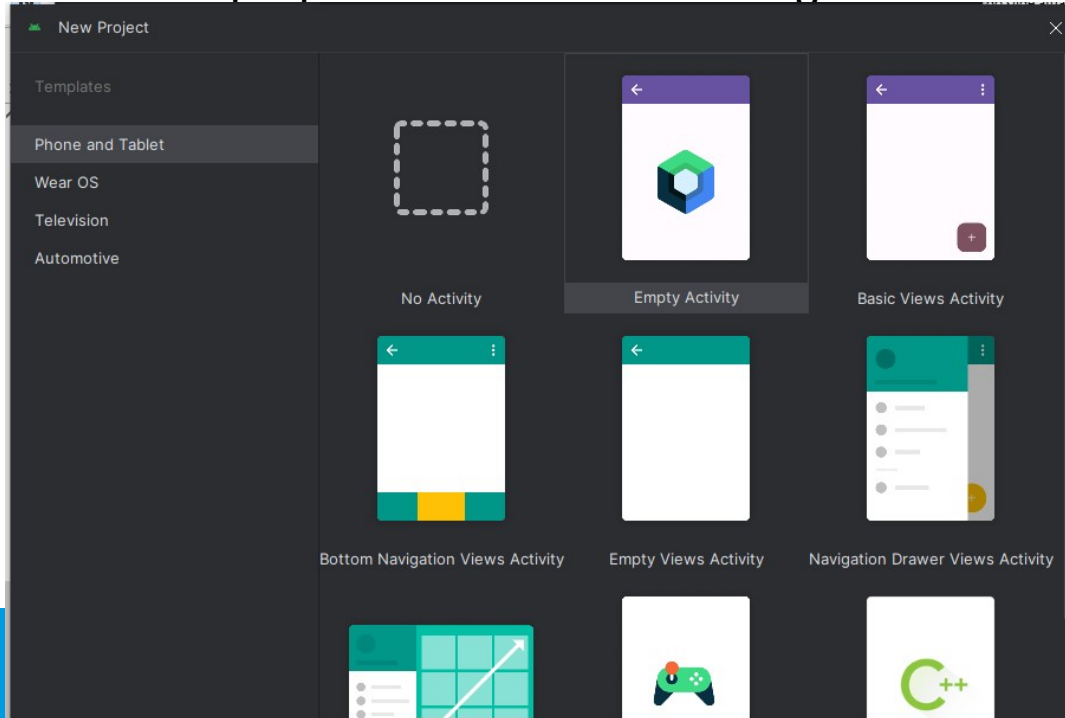
- Para navegar para outra rota, basta chamar o método `navigate()` na instância de `NavController`.

```
navController.navigate(route)
```

- O método de navegação usa um único parâmetro: uma `String` correspondente a uma rota definida no `NavHost`. Se a rota corresponder a uma das chamadas para `composable()` no `NavHost`, o app vai navegar para essa tela.

```
@Composable
fun Tela01(navController: NavController) {
    Box(modifier = Modifier.fillMaxSize(),
        contentAlignment = Alignment.TopCenter
    ){
        Column(horizontalAlignment = Alignment.CenterHorizontally
            ,verticalArrangement = Arrangement.Center
            ,modifier = Modifier.padding(100.dp)
        ){
            Text(text = "Tela 01"
                ,color = Color.Blue
                ,fontSize = 30.sp)
        }
    }
    Box(modifier = Modifier.fillMaxSize()
        ,contentAlignment = Alignment.Center
    ){
        Column {
            Button(onClick = { navController.navigate("Tela02") }) {
                Text("Vá para Tela 02")
            }
        }
    }
}
```

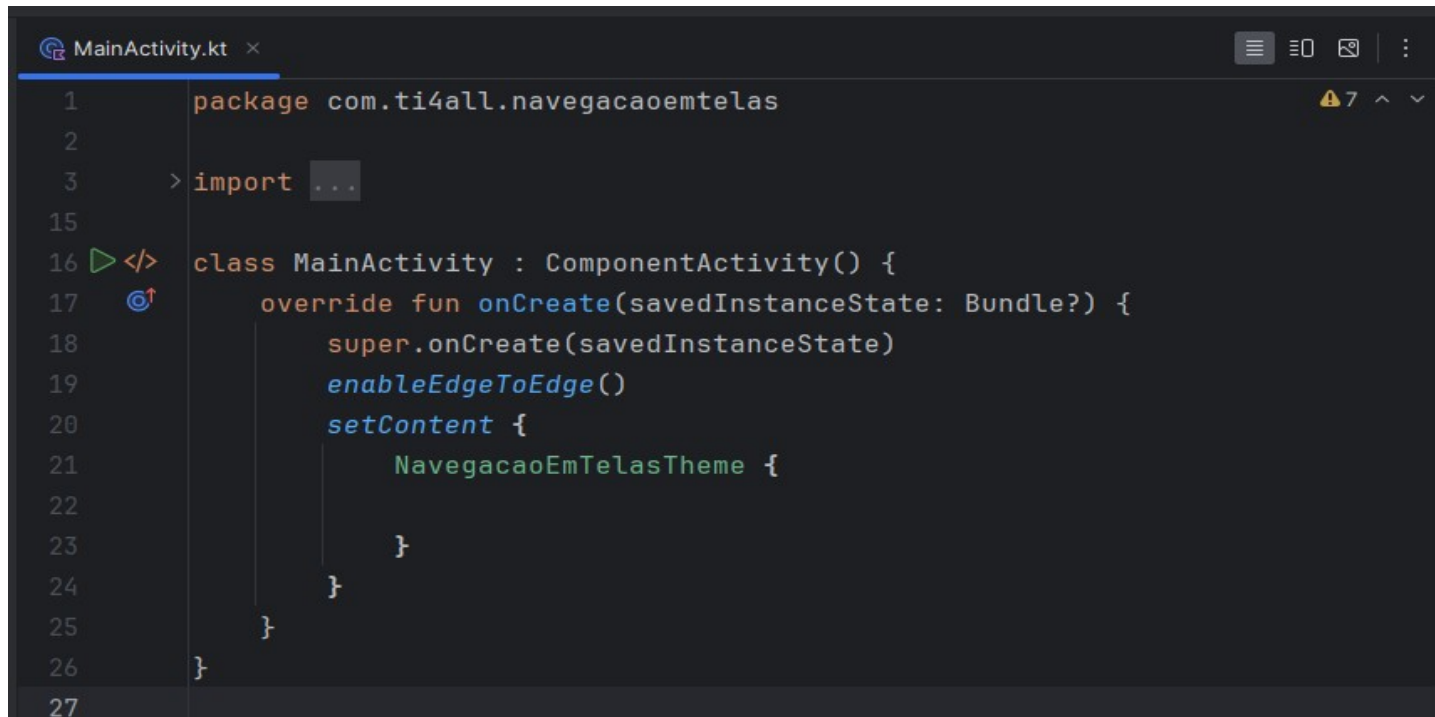

- **Sinopse Atividade Navegação em Telas**
- Selecione o tipo de template Phone and tablet;
- Selecione o template Empty Activity;
- Crie um projeto com o nome “Navegacao em Telas”;



- Na MainActivity remova as funções @Composable (Greeting e GreetingPreview)
- Remover todo o objeto Scaffold ou Surface da função onCreate.

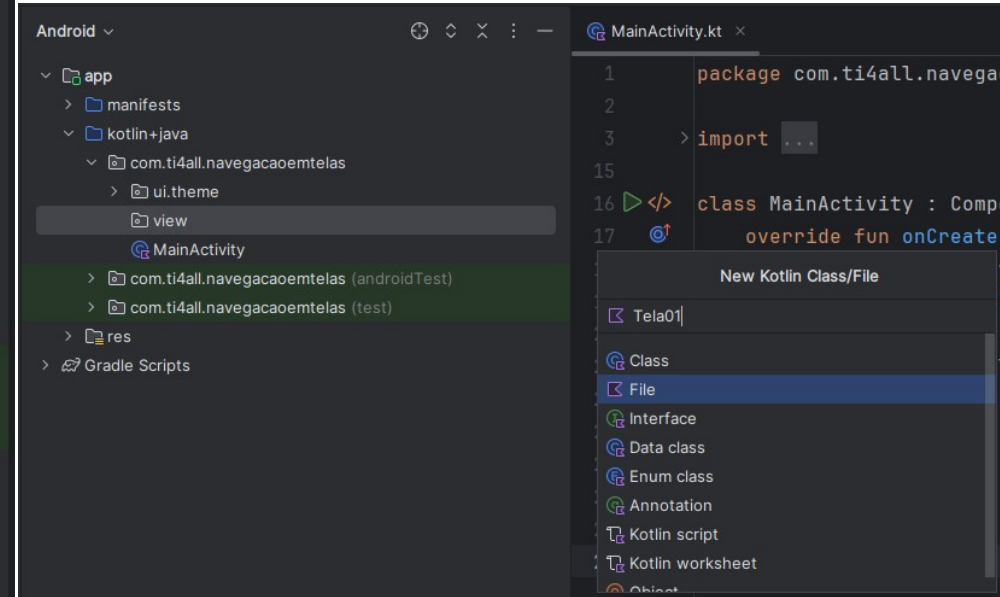
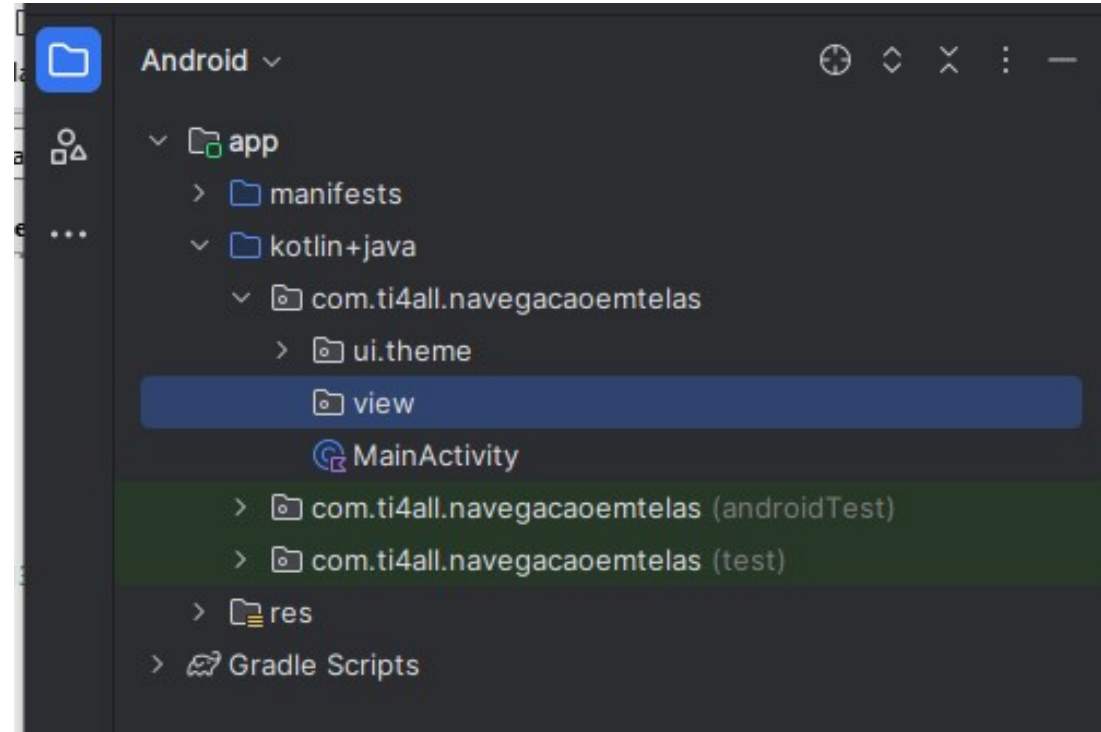
MainActivity → Será a activity que conterá todas as rotas (route) mas não será a tela principal (1ª. Tela) do projeto.

A aplicação terá duas telas (tela 01 e tela 02) e a navegação se dará nessas duas telas.

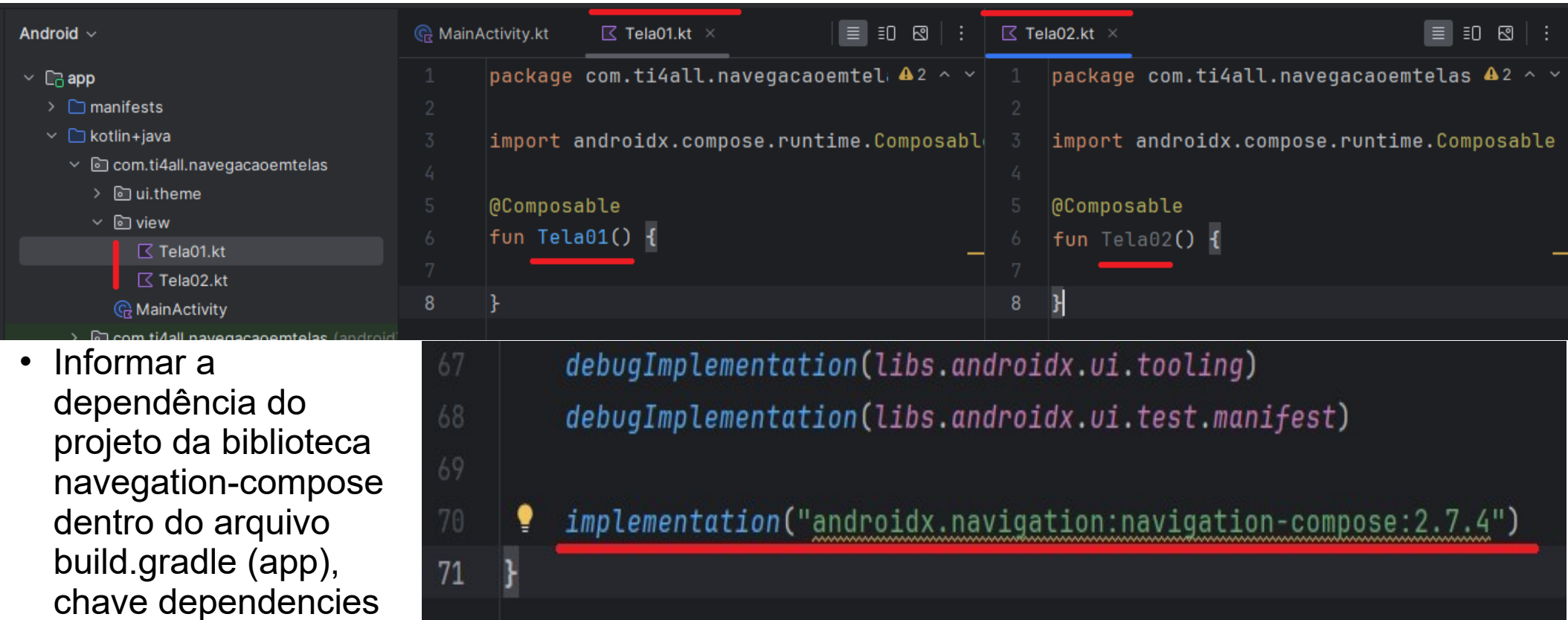


```
1 package com.ti4all.navegacaoemtelas
2
3 > import ...
4
5
6
7
8
9
10
11
12
13
14
15
16 class MainActivity : ComponentActivity() {
17     override fun onCreate(savedInstanceState: Bundle?) {
18         super.onCreate(savedInstanceState)
19         enableEdgeToEdge()
20         setContent {
21             NavegacaoEmTelasTheme {
22
23             }
24         }
25     }
26 }
27
```

- Criação das activities Tela 01 e Tela 02:
- Criar o package view. Dentro de view ficarão todas as telas e seus componentes;
- Criar os arquivos Tela01.kt e Tela02.kt (ambas como arquivo kotlin)



- Criar as respectivas funções @Composable:



```
Android ▾
├── app
│   ├── manifests
│   └── kotlin+java
│       └── com.ti4all.navegacaoemtelas
│           ├── ui.theme
│           └── view
│               ├── Tela01.kt
│               ├── Tela02.kt
│               └── MainActivity
└── com.ti4all.navegacaoemtelas (android)

MainActivity.kt
Tela01.kt
Tela02.kt

1 package com.ti4all.navegacaoemtelas
2
3 import androidx.compose.runtime.Composable
4
5 @Composable
6 fun Tela01() {
7
8 }

1 package com.ti4all.navegacaoemtelas
2
3 import androidx.compose.runtime.Composable
4
5 @Composable
6 fun Tela02() {
7
8 }

67 debugImplementation(libs.androidx.ui.tooling)
68 debugImplementation(libs.androidx.ui.test.manifest)
69
70 implementation("androidx.navigation:navigation-compose:2.7.4")
71 }
```

- Sincronize o projeto (menu File/Sync Project with Gradle Files);
- Configuração do objeto navegação na MainActivity

```
NavegacaoEmTelasTheme {  
    // variável navController que receberá rememberNavController  
    val navController = rememberNavController()  
    // configuração do NavHost que vai trabalhar com a  
    // variável navController e vai iniciar na Tela01  
    NavHost(navController = navController, startDestination = "Tela01") { this  
        // configuração da função composable  
        // que vai montar a "pilha" de navegação  
        composable(  
            route = "Tela01"  
        ) { this: AnimatedContentScope it: NavBackStackEntry  
            Tela01(navController)  
        }  
    }  
}
```

**Chamada da activity
Tela01, recebendo
navController - tem que
ajustar a @composable
Tela01 p/receber
navController**

- Ajustando a função @Composable Tela01 – arquivo Tela01.kt;

```
Tela02.kt  MainActivity.kt  build.gradle.kts (:app)  Tela01.kt x
1  package com.ti4all.navegacaoemtelas.view
2
3  import androidx.compose.runtime.Composable
4  import androidx.navigation.NavController
5
6  @Composable
7  fun Tela01(navController: NavController) {
8
9  }
```

- Continuaremos a configuração do NavHost, agora tratando a 2ª. Rota (activity ou tela):

```
NavHost(navController = navController, startDestination = "Tela01") { th  
    // configuração da função composabile  
    // que vai montar a "pilha" de navegação  
    // 1a. rota (ou tela)  
    composabile(  
        route = "Tela01"  
    ) { this: AnimatedContentScope it: NavBackStackEntry  
        Tela01(navController)  
    }  
    // 2a. rota (ou tela)  
    composabile(  
        route = "Tela02"  
    ) { this: AnimatedContentScope it: NavBackStackEntry  
        Tela02(navController)  
    }  
}
```

Também será preciso ajustar a função composabile Tela02 para receber navController

- Ajustando a função @Composable Tela02 – arquivo Tela02.kt;

```
Tela02.kt x MainActivity.kt Tela01.kt
1 package com.ti4all.navegacaoemtelas.view
2
3 import androidx.compose.runtime.Composable
4 import androidx.navigation.NavController
5
6 @Composable
7 fun Tela02(navController: NavController) {
8
9 }
```


- Nesse momento, a navegação das telas já está configurado e o App iniciar-se-á na Tela 01;
- Porém, como não há nenhum objeto composável dentro da classe Tela01 e Tela02, nada será apresentado.
- Definição do layout usado no App:
 - Tela 01:
 - Título da tela na parte superior e no centro horizontal da mesma com o seguinte texto: “Tela 01”
 - Botão com a mensagem “Vá para Tela 02”
 - Tela 02:
 - Título da tela na parte superior e no centro horizontal da mesma com o seguinte texto: “Tela 02”
 - Botão com a mensagem “Vá para Tela 01”

- Configuração do layout da função @Composable Tela01 – arquivo Tela01.kt;

```
Tela02.kt  MainActivity.kt  Tela01.kt x
22  @Composable
23  fun Tela01(navController: NavController) {
24      Box(modifier = Modifier.fillMaxSize(),
25          contentAlignment = Alignment.TopCenter
26      ){ this: BoxScope
27          Column(horizontalAlignment = Alignment.CenterHorizontally
28              ,verticalArrangement = Arrangement.Center
29              ,modifier = Modifier.padding(100.dp)
30          ) { this: ColumnScope
31              Text(text = "Tela 01"
32                  ,color = Color.Blue
33                  ,fontSize = 30.sp)
34          }
35      }
36      Box(modifier = Modifier.fillMaxSize()
37          ,contentAlignment = Alignment.Center
38      ) { this: BoxScope
39          Column() { this: ColumnScope
40              Button(onClick = { navController.navigate(route: "Tela02") }) { th
41                  Text(text: "Vá para Tela 02")
42              }
43          }
44      }
45  }
```

- Configuração do layout da função @Composable Tela02 – arquivo Tela02.kt;

```
Tela02.kt x MainActivity.kt Tela01.kt
20
21 @Composable
22 fun Tela02(navController: NavController) {
23     Box(modifier = Modifier.fillMaxSize()
24         ,contentAlignment = Alignment.TopCenter
25     ) { this: BoxScope
26         Column(horizontalAlignment = Alignment.CenterHorizontally
27             ,verticalArrangement = Arrangement.Center
28             ,modifier = Modifier.padding(100.dp)) { this: ColumnScope
29             Text(text = "Tela 02"
30                 ,color = Color.Blue
31                 ,fontSize = 30.sp)
32         }
33     }
34     Box(modifier = Modifier.fillMaxSize()
35         ,contentAlignment = Alignment.Center
36     ) { this: BoxScope
37         Column(modifier = Modifier.padding(16.dp)) { this: ColumnScope
38             Button(onClick = { navController.navigate(route: "Tela01") }) { this:
39                 Text(text: "Vá para Tela 01")
40             }
41         }
42     }
43 }
```

- **getSystemService**

- Método da classe Context no Android que permite acessar serviços do sistema Android, como o LayoutInflater, ConnectivityManager, LocationManager, entre outros.
- Esses serviços são essenciais para realizar tarefas específicas, como gerenciar conexões de rede, acessar informações de localização, ou manipular o layout da interface do usuário.
- No Jetpack Compose, não há uma referência direta ao Context como em atividades ou fragmentos. No entanto, é possível obter o Context e usar o getSystemService através do LocalContext.current

- **getSystemService**

- Exemplo: Verificar a conectividade de rede em um composável:
- Lembrando que sempre que houver necessidade de acessar serviços disponíveis do equipamento é necessário adicionar a(s) devida(s) permissão(ões) no arquivo AndroidManifest.xml.
- Adicione as permissões necessárias no AndroidManifest.xml:

(`<uses-permission android:name = "android.permission.ACCESS_NETWORK_STATE" />`)

@Composable

```
fun NetworkStatus() {  
    // Obtém o contexto atual  
    val context = LocalContext.current  
    // Obtém o ConnectivityManager usando getSystemService  
    val connectivityManager = remember {context.getSystemService(Context.CONNECTIVITY_SERVICE) as ConnectivityManager }  
    // Verifica a conectividade de rede  
    val isConnected = remember {  
  
connectivityManager.getNetworkCapabilities(connectivityManager.activeNetwork) ? .hasCapability(NetworkCapabilities.NET_CAPABILITY_INTERNET) == true  
    }  
    // Exibe o status da rede  
    Column(modifier = Modifier.padding(16.dp)) { Text(text = "Network Status: ${if (isConnected) "Connected" else "Not Connected"}")  
    }  
}
```

```
val connectivityManager = remember { context.getSystemService(Context.CONNECTIVITY_SERVICE) as ConnectivityManager }
```

- Obtém uma instância de ConnectivityManager usando getSystemService.
- O remember é usado para preservar o valor entre recomposições.
- getSystemService(Context.CONNECTIVITY_SERVICE) retorna um serviço do sistema que gerencia a conectividade de rede.
- A asserção as ConnectivityManager garante que o serviço retornado é do tipo ConnectivityManager.

```
val isConnected = remember {
```

- Define uma variável isConnected que armazena o estado de conexão de rede.
- O remember é usado aqui para preservar o resultado entre recomposições..

```
connectivityManager.getNetworkCapabilities(connectivityManager.activeNetwork)?.hasCapability(NetworkCapabilities.NET_CAPABILITY_INTERNET) == true:
```

- Obtém as capacidades da rede ativa usando getNetworkCapabilities e verifica se a rede tem a capacidade de fornecer acesso à Internet.
- getNetworkCapabilities pode retornar null se não houver rede ativa, então o operador seguro ?. é usado para evitar exceções. O hasCapability(NetworkCapabilities.NET_CAPABILITY_INTERNET) retorna true se a rede ativa tem capacidade para acesso à Internet.

- **App Lanterna**
- Selecione o tipo de template Phone and tablet;
- Selecione o template Empty Activity;
- Crie um projeto com o nome “Lanterna”;
 - Nesse App, usaremos um switch button
 - Usaremos um recurso físico do dispositivo
 - Não esquecer de configurar a permissão em AndroidManifest.xml
 - Insira as seguintes linhas dentro da tag <manifest>, mas fora da tag <application>.
- `<uses-permission android:name="android.permission.CAMERA"/>`
- `<uses-feature android:name="android.hardware.camera" android:required="false" />`
- <https://www.youtube.com/watch?v=y9SdkPbKq24>

- Na MainActivity, apagar os códigos referentes às funções Composable Greeting e GreetingPreview;
- Ainda, na MainActivity, dentro da função onCreate, troque o Scaffold pela chamada da função Lanterna()

```
29 </> class MainActivity : ComponentActivity() {  
30     override fun onCreate(savedInstanceState: Bundle?) {  
31         super.onCreate(savedInstanceState)  
32         enableEdgeToEdge()  
33         setContent {  
34             Lanterna()  
35         }  
36     }  
37 }  
38
```

- Abaixo da definição da classe MainActivity, codifique a função Composable Lanterna():

```
40 @Composable
41 fun Lanterna() {
42     var isChecked by remember { mutableStateOf( value: false) }
43     Column(modifier = Modifier.fillMaxSize()
44             .padding(16.dp),
45             verticalArrangement = Arrangement.Center,
46             horizontalAlignment = Alignment.CenterHorizontally
47     ) { this: ColumnScope
48         Switch(checked = isChecked,
49               onCheckedChange = { isChecked = it},
50               modifier = Modifier.fillMaxWidth()
51                       .wrapContentWidth(Alignment.CenterHorizontally)
52         )
53         if (isChecked) {
54             LanternaOnOff(state = true)
55         } else {
56             LanternaOnOff(state = false)
57         }
58     }
59 }
```

- Abaixo da definição da função `Lanterna()`, insira o código da função `LanternaOnOff()`;

```
61 @RequiresApi(Build.VERSION_CODES.M)
62 @Composable
63 private fun LanternaOnOff(state: Boolean) {
64     val context = LocalContext.current
65     // Retorna o serviço de câmera do sistema
66     val cameraManager = context.getSystemService(Context.CAMERA_SERVICE) as CameraManager
67
68     try {
69         // Retorna a id da câmera a ser utilizada - [0 - câmera traseira] [1 - câmera frontal]
70         val idCamera: String = cameraManager.cameraIdList[0]
71         // Ativa e desativa a câmera
72         cameraManager.setTorchMode(idCamera, state)
73     } catch (e: Exception) {
74         Log.e(tag: "erroCamera", msg: "Erro ao ligar/desligar a lanterna: ${e.message}", e)
75         e.printStackTrace()
76     }
77 }
```

- Em um momento a IDE vai sugerir a inclusão da anotação **@RequiresApi(Build.VERSION_CODES.M)** que indica que a função requer a API mínima de nível 23 (Android Marshmallow) para ser executada.
- **val context = LocalContext.current** → é uma função do Jetpack Compose que permite obter o contexto atual. O contexto é necessário para acessar o serviço de câmera do sistema e fica “armazenado” na variável denominada **context**;
- **val cameraManager = context.getSystemService(Context.CAMERA_SERVICE)** as **CameraManager** → obtém o serviço de câmera do sistema usando o contexto. Usa-se o Context.CAMERA_SERVICE para acessar o serviço de câmera que é necessário para interagir com a câmera do dispositivo.
- **cameraManager.cameraIdList[0]**: Aqui obtém-se a lista de IDs das câmeras disponíveis no dispositivo. Como a lista pode conter mais de uma câmera, o código está se referenciando ao ID da primeira câmera na lista ([0]). Normalmente, a primeira câmera é a câmera traseira.

- **cameraManager.setTorchMode(idCamera, state)** → essa linha está ativando ou desativando a lanterna da câmera através do **setTorchMode()** para controlar o estado da lanterna. O primeiro parâmetro é o ID da câmera que se quer controlar e o segundo parâmetro é o estado (ligado ou desligado) da lanterna.
- **Log.e("erroCamera", "Erro ao ligar/desligar a lanterna: \${e.message}", e)** → esse bloco trata de possíveis erros que possam vir a ocorrer ao tentar ligar ou desligar a lanterna. Se o erro acontecer ele será capturado e registrado no Logcat do Android, indicando que houve um problema ao ligar ou desligar a lanterna.

- Atividade em sala:
 - No projeto Navegação em Telas, adicionar:
 - Criar a activity Tela03 no mesmo padrão de Tela01 e Tela02;
 - Na activity Tela02 inserir um botão que leve a navegação para a activity Tela03;
 - Na activity Tela03 inserir um botão que leve a navegação para a activity Tela02;
 - Nas activitys Tela02 e Tela03 inserir um botão “Home” cuja ação levará a navegação diretamente para a activity Tela01.
 - No projeto Lanterna:
 - Espaço sobre o switch button para receber uma imagem qualquer que transmita a mensagem de uma lanterna apagada ou acesa;
 - Alternar a imagem de acordo com o estado do switch button