CURSOS TÉCNICOS

DESENVOLVIMENTO DE APLICATIVOS I

Eixo Informática para Internet

UNIDADE 7

QI Faculdade & Escola Técnica

SUMÁRIO

UNIDADE 7

1. ACTIVITY E CICLO DE VIDA 3
1.1. O que vamos aprender 3
1.2. Obtendo o projeto inicial 3
2. O que é uma Activity? 4
2.1 Estágios do ciclo de vida de uma Activity 4
2.2 Como funcionam os estágios de vida das Activities 6
2.3 Classe Log 8
2.3.1 Console Logcat 9
2.4 Implementando método onStart() 9
3. NAVEGAR ENTRE TELAS DE UM APP USANDO O COMPOSE 11
3.1 Conceitos que definem a navegação 11
3.2 Obtendo e entendendo o código inicial 13
3.3 Configurando o projeto 14
3.4 Configurando o componente de navegação - NavHostController 15
3.4.1 O componente de navegação tem três partes principais: 15
3.4.2 Trabalhando com as rotas do App 15
3.4.3 NavHost 16
3.4.3.1 Configurando o NavHost 17
3.4.3.2 Configurando o “Navegar entre rotas” 19
3.5 Navegando por outras rotas usando método navigate() 22
3.6 Retornando à tela inicial 24
4. Configurando a AppBar 26
5. Referências 28

UNIDADE 7

1. ACTIVITY E CICLO DE VIDA

Nas unidades anteriores trabalhamos com vários tipos de abordagens na criação de aplicativos básicos. Aprendemos a trabalhar com o Jetpack Compose, e a fazer algumas interações importantes.

A partir desta unidade, vamos aprender como criar várias telas em um aplicativo, e como definirmos o fluxo de navegação do app, tudo isso desenvolvendo os aplicativos.

1.1. O que vamos aprender

Vamos aprender sobre os aspectos conceituais e práticos. Vejamos:

- Os conceitos básicos do **ciclo de vida da Activity e os callbacks invocados** quando a atividade se move entre estados.
- Como **modificar métodos de callback** do ciclo de vida para realizar operações em diferentes momentos do ciclo de vida da atividade.
- Como substituir os métodos de callback do ciclo de vida e registrar as mudanças no estado da atividade.
- Classe **Log** e console **Logcat.**
- Como **implementar rememberSaveable** para reter os dados do app que podem ser perdidos caso a configuração do dispositivo mude.
- Criar um elemento de **composição do NavHost** para definir rotas e telas no seu app.
- Navegar entre telas usando um **NavHostController**.
- **Manipular a backstack** para voltar às telas anteriores.
- **onSelectionChanged, onCancelButtonClicked e onNextButtonClicked.**
- Usar **intents** para compartilhar dados com outro app.
- **Personalizar a AppBar, incluindo o título e o botão "Voltar".**

1.2. Obtendo o projeto inicial

Neste link, você encontrará o repositório para fazer *download* do código inicial. As referências de pesquisa sempre estão no readme do repositório. Então:

1. Faça download do repositório.
2. Descompacte.
3. Abra no Android Studio.

4. Execute a visualização do app pela opção Split no IDE.

2. O que é uma Activity?

Activity é um componente fundamental que representa uma única tela com uma interface do usuário. Ela é responsável por interagir com o usuário e exibir informações ou receber entrada do usuário.

Uma Activity é parte integrante de um aplicativo Android e atua como uma unidade independente de interação. Ela pode conter elementos de interface do usuário, como botões, campos de texto, imagens e outros componentes visuais. **Quando um usuário interage com um aplicativo Android, ele está, na verdade, interagindo com diferentes Activities que são exibidas em sequência.**

Cada Activity possui um ciclo de vida, que consiste em uma série de estados e métodos que são chamados automaticamente pelo sistema Android.

Esses estados incluem:

- ✔ **Created:** a Activity foi criada, mas não está visível para o usuário.
- ✔ **Started:** a Activity está visível, mas não está em primeiro plano. Ela pode ser parcialmente obscurecida por outras Activities ou elementos da interface do usuário.
- ✔ **Resumed:** a Activity está em primeiro plano e ativamente interagindo com o usuário. É o estado em que o usuário pode realizar ações na tela.
- ✔ **Paused:** a Activity ainda está visível, mas não está em primeiro plano. Isso ocorre quando outra Activity é sobreposta a ela.
- ✔ **Stopped:** a Activity não está mais visível. Isso pode acontecer quando uma nova Activity é iniciada ou quando a Activity atual é finalizada.
- ✔ **Destroyed:** a Activity foi encerrada e liberou seus recursos. Ela não está mais disponível.

É possível manipular esses estados para controlar o comportamento da interface do usuário e a lógica do aplicativo. Isso permite que o aplicativo responda adequadamente às ações do usuário e ao ciclo de vida do dispositivo.

2.1 Estágios do ciclo de vida de uma Activity

O que é um ciclo de vida? **O ciclo de vida de uma Activity refere-se ao conjunto de estados e métodos pelos quais uma Activity passa desde o momento em que é criada até o momento em que é completamente destruída.** Esse ciclo de vida é gerenciado pelo

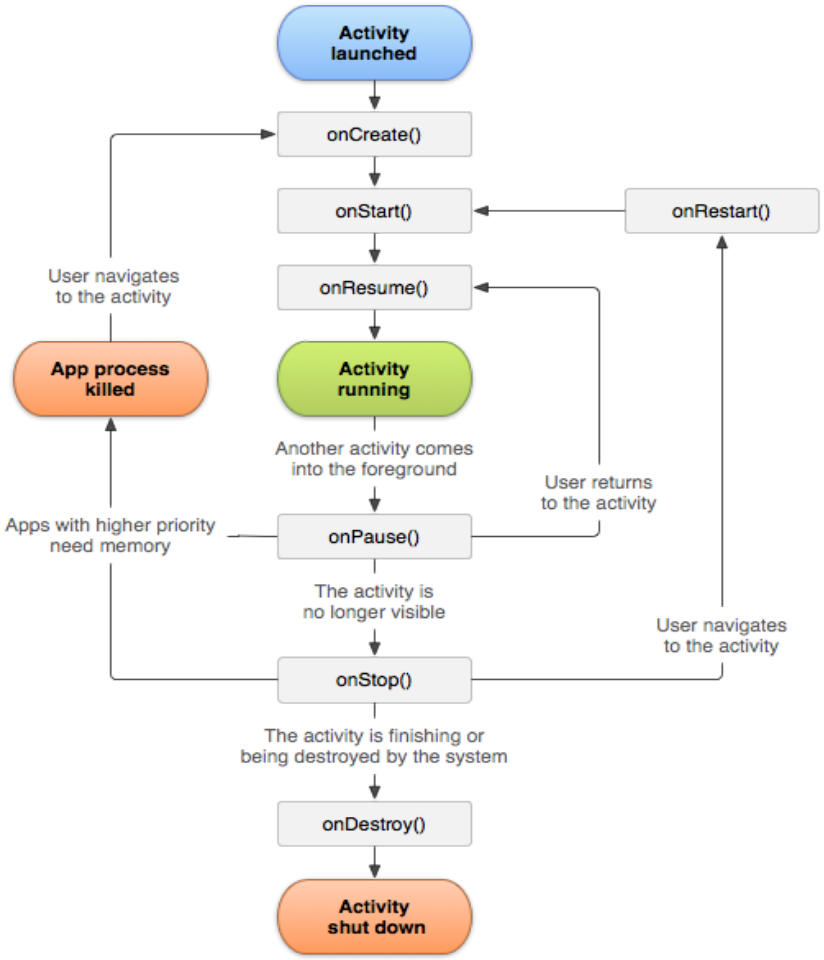
sistema Android e permite que os desenvolvedores controlem o comportamento da Activity em diferentes situações definidas pelos estágios do ciclo de vida.

Conforme vimos acima, uma Activity possui estados, e esses estados possuem métodos associados para a manipulação da Activity.

- **onCreate():** chamado quando a Activity está sendo criada. Aqui, geralmente declaramos as inicializações, como configuração da interface do usuário e preparativos iniciais. É onde nós normalmente populamos o layout da Activity.
- **onStart():** chamado quando a Activity entra no estado "Started", tornando-se visível para o usuário. Aqui é um bom lugar para iniciar recursos que devem estar disponíveis quando a Activity estiver visível.
- **onResume():** chamado quando a Activity entra no estado "Resumed", ou seja, está em primeiro plano e ativamente interagindo com o usuário. Aqui, podemos iniciar animações, atualizar dados e preparar a interface para a interação do usuário.
- **onPause():** chamado quando a Activity perde o foco, mas ainda é parcialmente visível. Isso ocorre quando outra Activity é sobreposta a ela. Usamos esse método para liberar recursos que não são mais necessários enquanto a Activity não está em primeiro plano.
- **onStop():** chamado quando a Activity não está mais visível, entrando no estado "Stopped". Isso pode acontecer quando uma nova Activity é iniciada ou quando a Activity atual é finalizada. Podemos usar esse método para fazer operações de limpeza mais pesadas.
- **onDestroy():** chamado quando a Activity está prestes a ser destruída. Isso pode acontecer quando o usuário a finaliza ou quando o sistema precisar liberar recursos. Neste ponto, devemos liberar todos os recursos, encerrar conexões e fazer qualquer limpeza final.

Fonte da imagem: <Entenda o ciclo de vida da atividade | Desenvolvedores Android>

Um exemplo mais completo →



Fonte da imagem: <Entenda o ciclo de vida da atividade | Desenvolvedores Android>

Frequentemente, surge o desejo de alterar um comportamento ou executar um trecho de código quando ocorrem mudanças no estado de um ciclo de atividade. Por isso, a classe Activity em si, bem como todas as suas subclasses, como a Component Activity, incorporam um conjunto de métodos de retorno de chamada, para acompanhar o ciclo de vida. O sistema Android invoca esses retornos de chamada, sempre que a atividade transita de um estado para outro. Neste sentido, é possível sobrescrever esses métodos nas suas próprias atividades, permitindo a realização de tarefas em resposta a essas flutuações no ciclo de vida.

2.2 Como funcionam os estágios de vida das Activities

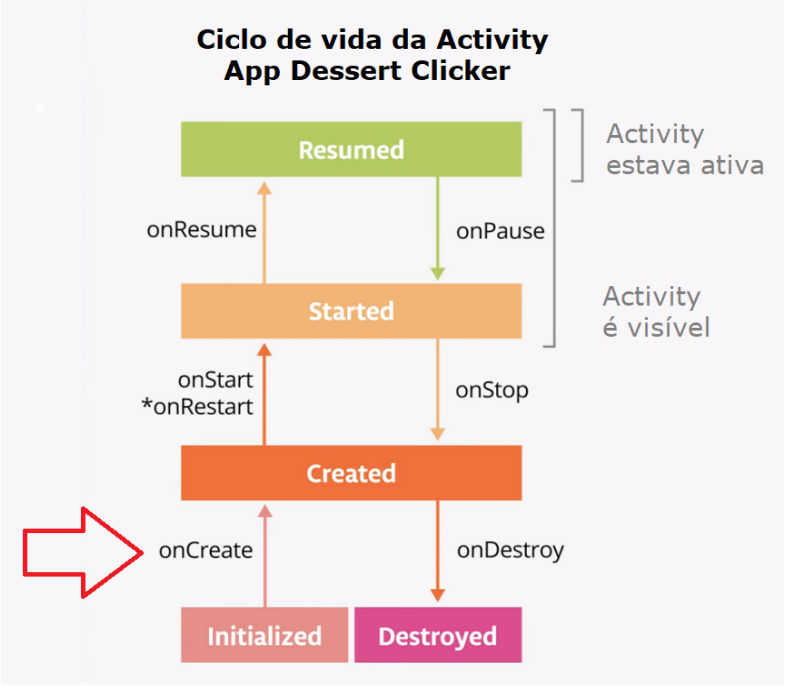
As atividades do Android começam sempre com o método onCreate(), ou seja, quando o usuário abre o app, navega entre as activities ou mesmo entra e sai do app, a activity sempre muda de estado.

É importante salientar aqui, que um app Android pode ter várias activities e que todas podem registrar um tipo de estado.

Para compreender a dinâmica do ciclo de vida do Android, e rastrear a invocação dos diversos métodos do ciclo de vida, torna-se relevante conhecer a sincronia em que essas funções são chamadas. Esta visão é instrumental e serve para identificar eventuais problemas no app modelo Dessert Clicker.

Uma abordagem simples para adquirir essa informação reside na exploração da capacidade de registro, integrada ao Android. **A funcionalidade de registro concede a habilidade de registrar mensagens concisas, em um console enquanto o aplicativo está em execução, proporcionando, assim, a oportunidade de verificar quando diferentes callbacks são ativados.**

Se executarmos o app no emulador, veremos o valor das sobremesas (dessert) vendidas e o valor em reais mudando.



Fonte da imagem: de própria autoria adaptada de <Desenvolvedores Android>

Na imagem acima, que exibe o diagrama do ciclo de vida da activity, está assinalado o método onCreate(). Ele é o único método que todas as activities precisam implementar. É no método onCreate() que precisamos fazer todas as inicializações únicas para a activity. Um bom exemplo é: em onCreate(), chamamos setContent(), que especifica o layout da interface da activity.

De acordo com a documentação do Android, **"... ao modificar o método onCreate(), você precisa chamar a implementação da superclasse para concluir a criação da**

atividade. Dessa forma, é necessário chamar super.onCreate() imediatamente. O mesmo acontece em outros métodos de callback do ciclo de vida." (Entenda o ciclo de vida da atividade | Desenvolvedores Android)

De forma elevada, vamos declarar uma variável para que possamos ver em log os ciclos de vida funcionando no app modelo.

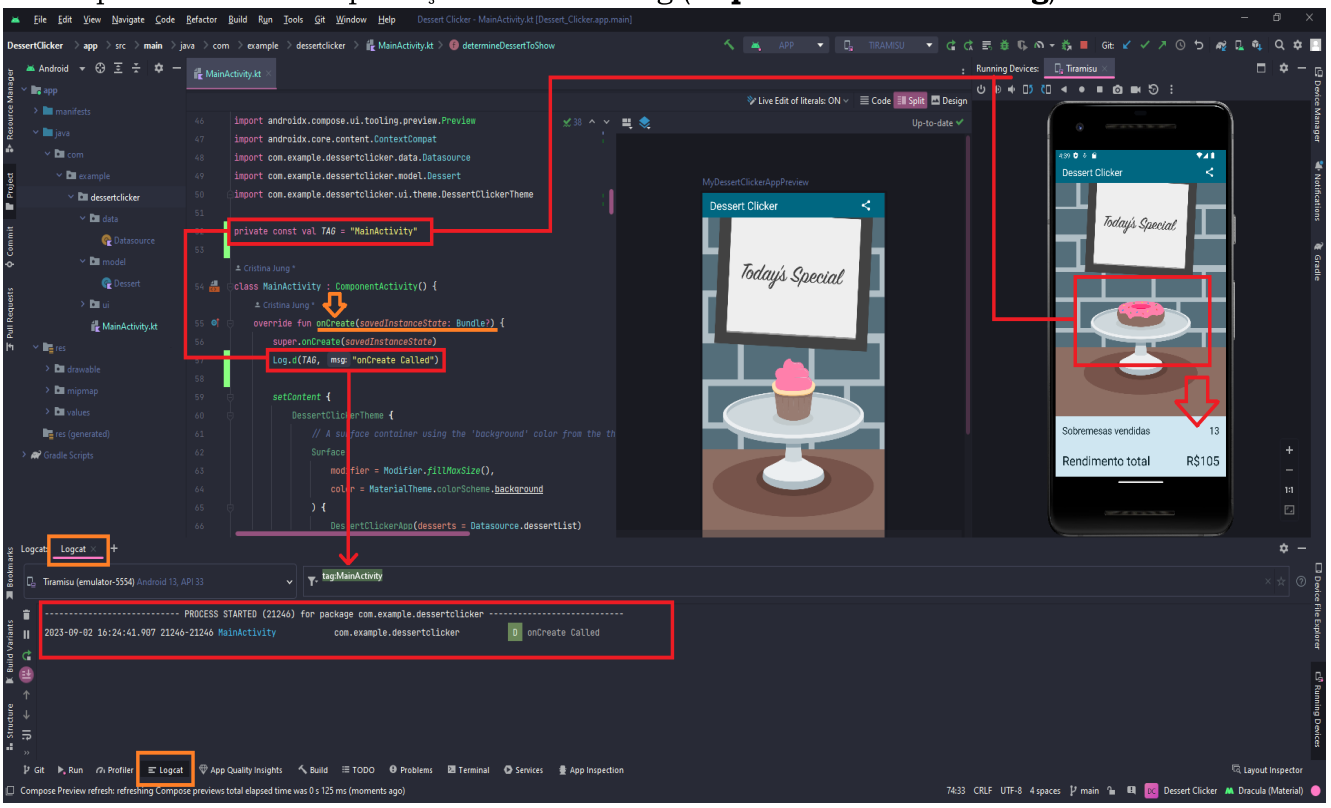
Vamos declarar em elevação, acima da classe MainActivity:

```
private const val TAG = "MainActivity"
```

E no método onCreate():

```
Log.d(TAG, "onCreate Called")
```

É preciso fazer a importação da classe Log (**import android.util.Log**).



Fonte da imagem: autoria própria, 2023

2.3 Classe Log

A classe Log é uma classe fundamental na plataforma Android, que é usada para registrar mensagens de depuração, informações ou erros durante a execução de um aplicativo. Ela escreve mensagens no **Logcat,** que é o console usado para registrar mensagens. As mensagens do Android sobre seu app são exibidas nesse local, incluindo as mensagens que você envia, explicitamente, para o registro com o método Log.d() ou outros métodos da classe Log.

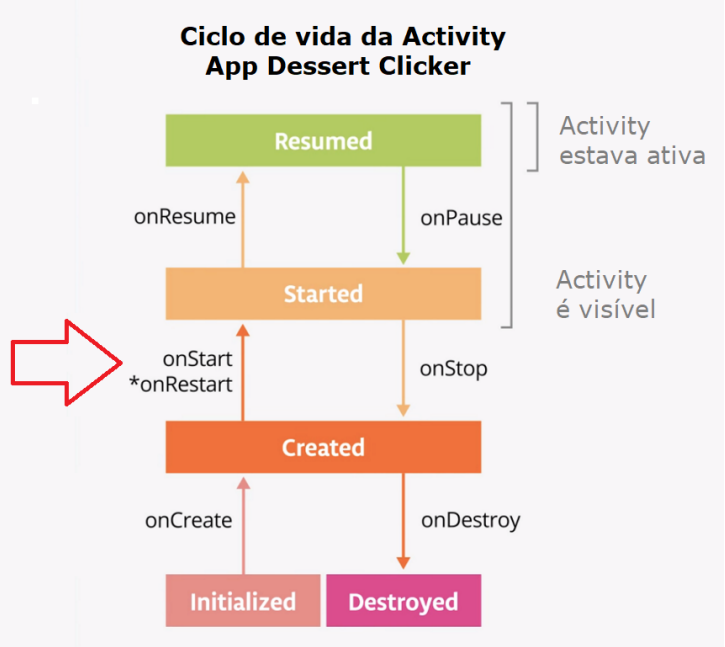
A classe Log inclui vários métodos estáticos para registrar mensagens em diferentes níveis de gravidade, como "**verbose,**" "**debug,**" "**info,**" "**warning**" **e** "**error.**" Esses níveis de gravidade permitem que você organize as mensagens de acordo com sua importância.

2.3.1 Console Logcat

O Logcat é uma ferramenta de linha de comando presente no Android, que permite visualizar e analisar registros (logs) gerados pelo sistema e pelos apps em execução no dispositivo Android. Lembrando que, para usar o Logcat, é necessário ter acesso ao emulador onde o app está em execução. É possível acessar o Logcat, por meio de um terminal ou na barra de status da IDE Android (visualizado na imagem acima).

2.4 Implementando método onStart()

A função **onStart()** é invocada imediatamente após a **onCreate()** dentro do ciclo de vida. Após a execução do onStart(), a sua atividade torna-se visível na tela. Diferentemente da onCreate(), que é acionada apenas uma vez para inicializar a atividade, o onStart() pode ser chamado múltiplas vezes pelo sistema ao longo do ciclo de vida da atividade.

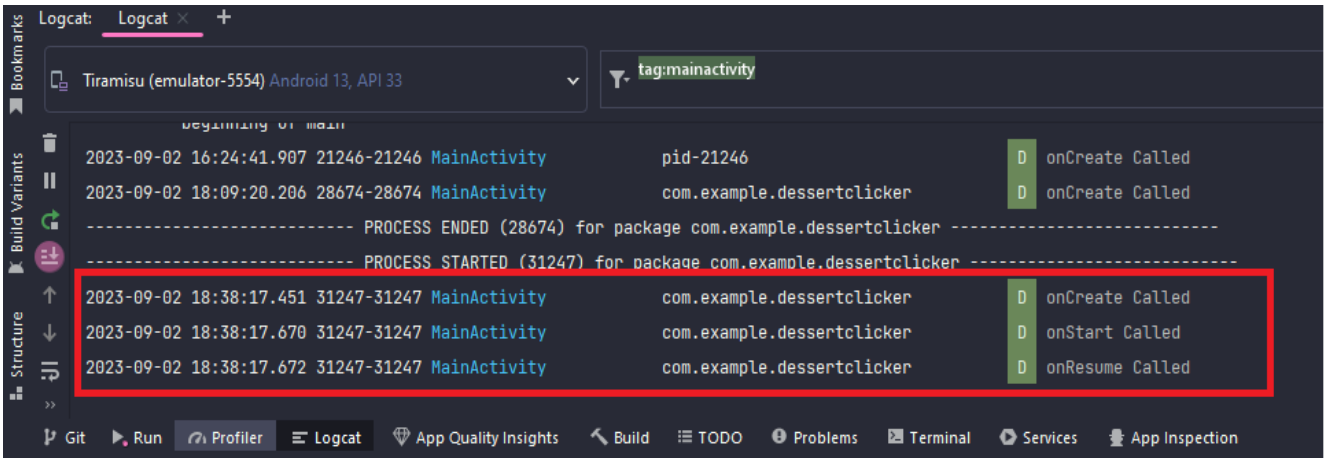


Fonte da imagem: de própria autoria adaptada de <Desenvolvedores Android>

Para que possamos ver o ciclo de vida em funcionamento, podemos declarar os métodos responsáveis por este processo dentro da MainActivity.

```
override fun onStart() {
    super.onStart()
    Log.d(TAG, "onStart Called")
}

override fun onResume() {
    super.onResume()
    Log.d(TAG, "onResume Called")
}

override fun onRestart() {
    super.onRestart()
    Log.d(TAG, "onRestart Called")
}

override fun onPause() {
    super.onPause()
    Log.d(TAG, "onPause Called")
}

override fun onStop() {
    super.onStop()
    Log.d(TAG, "onStop Called")
}

override fun onDestroy() {
    super.onDestroy()
    Log.d(TAG, "onDestroy Called")
}
```

Uma dica!
Para que possamos ver estes ciclos de vida no Logcat, será preciso reiniciar seu emulador, desta forma, veremos no Logcat as seguintes mensagens:



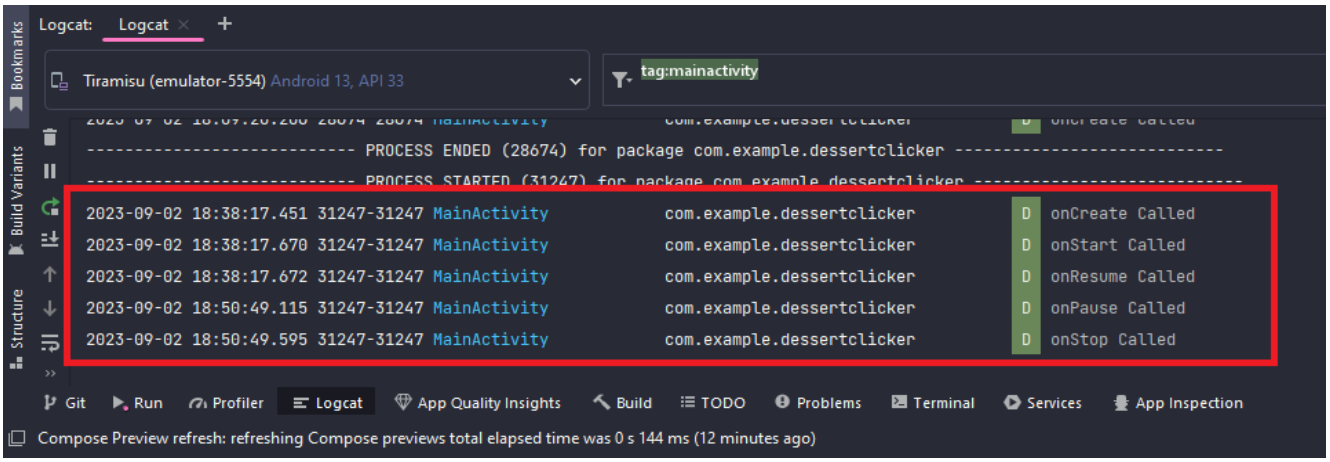
Fonte da imagem: autoria própria

Quando uma activity é iniciada desde o começo, veremos os três callbacks do ciclo de vida chamados nesta ordem:

- ✔ **onCreate()**, quando o sistema cria o app.
- ✔ **onStart()** deixa o app visível na tela, mas o usuário ainda não pode interagir com ele.

✔ **onResume()** coloca o app em primeiro plano, e o usuário agora pode interagir com ele.

Se clicarmos no botão voltar do dispositivo, veremos os métodos **onPause() seguido de onStop()**:



Os métodos **onCreate()** e **onDestroy()**, são invocados apenas uma vez ao longo da existência de uma única instância de atividade. O **onCreate** entra em cena para a inicialização do aplicativo, enquanto o **onDestroy()** desempenha o papel de invalidar, encerrar ou destruir os objetos utilizados pela atividade, evitando que eles persistam e consumam recursos, como memória,, por exemplo.

Com esta prática, entendemos que o ciclo de vida das activities é um conjunto de estados que uma activity passa. Ela é criada, modificada, pausada, parada e por fim, destruída quando o app não está mais sendo usado, ou mesmo quando algum processo interno da aplicação é encerrado.

3. NAVEGAR ENTRE TELAS DE UM APP USANDO O COMPOSE

A navegação, entre telas, de um aplicativo usando o Jetpack Compose é uma parte essencial do desenvolvimento de apps. Até agora, trabalhamos apenas com uma tela, porém aprofundamos algumas características fundamentais e básicas para o desenvolvimento de um aplicativo.

Nesta etapa do aprendizado, após termos conhecido o ciclo de vida das atividades, nada mais natural do que evoluirmos para uma etapa importante: **a navegação**.

Navegar entre telas, com o Android Jetpack Compose é um processo que permite aos desenvolvedores criar transições suaves e eficientes, entre diferentes partes de um aplicativo Android. Ao contrário da abordagem tradicional com XML de layout e atividades, o Jetpack Compose permite que possamos definir as interfaces do usuário, por meio de

código Kotlin, tornando a navegação entre telas uma parte integrada e flexível do desenvolvimento de aplicativos.

3.1 Conceitos que definem a navegação

Vejamos agora o rol de conceitos que definem a navegação entre telas.

- **Composables de destino:** cada tela ou destino do seu aplicativo é representado por um composable, que é uma função Kotlin que desenha a interface do usuário daquela tela.
- **NavHost:** o NavHost é um composable especializado que atua como um contêiner para seus composables de destino. É responsável por gerenciar o estado de navegação e trocar os composables, conforme o usuário navega entre as telas.
- **NavController:** o NavController é uma classe que gerencia o estado de navegação e as transições entre as telas. Ele é usado para navegar entre composables de destino, e manter o histórico de navegação.
- **Gráfico de navegação:** um gráfico de navegação é uma representação visual da estrutura de navegação do seu aplicativo. Ele define os destinos (telas) e as conexões entre eles.
- **Argumentos de navegação:** podemos passar argumentos de um composable para outro, ao navegar entre telas. Isso é útil para transmitir dados ou informações entre as partes do seu aplicativo.
- **Animações de transição:** o Compose permite criar animações de transição personalizadas entre as telas, proporcionando uma experiência de usuário mais agradável.
- **Ações de navegação:** as ações de navegação são eventos acionados pelo usuário ou pelo código, que iniciam a navegação entre telas. Por exemplo, um clique em um botão, pode ser uma ação de navegação que leva o usuário a uma nova tela.
- **Back stack:** o NavController mantém um histórico de navegação chamado "back stack". Isso permite que o usuário volte às telas anteriores, pressionando o botão de retorno ou gesto de voltar.
- **Destino de início:** o destino de início é a primeira tela exibida quando o aplicativo é lançado. É a tela inicial do aplicativo.

Exemplo de app com várias telas

Fonte da imagem: <Como navegar com o Compose - Jetpack>

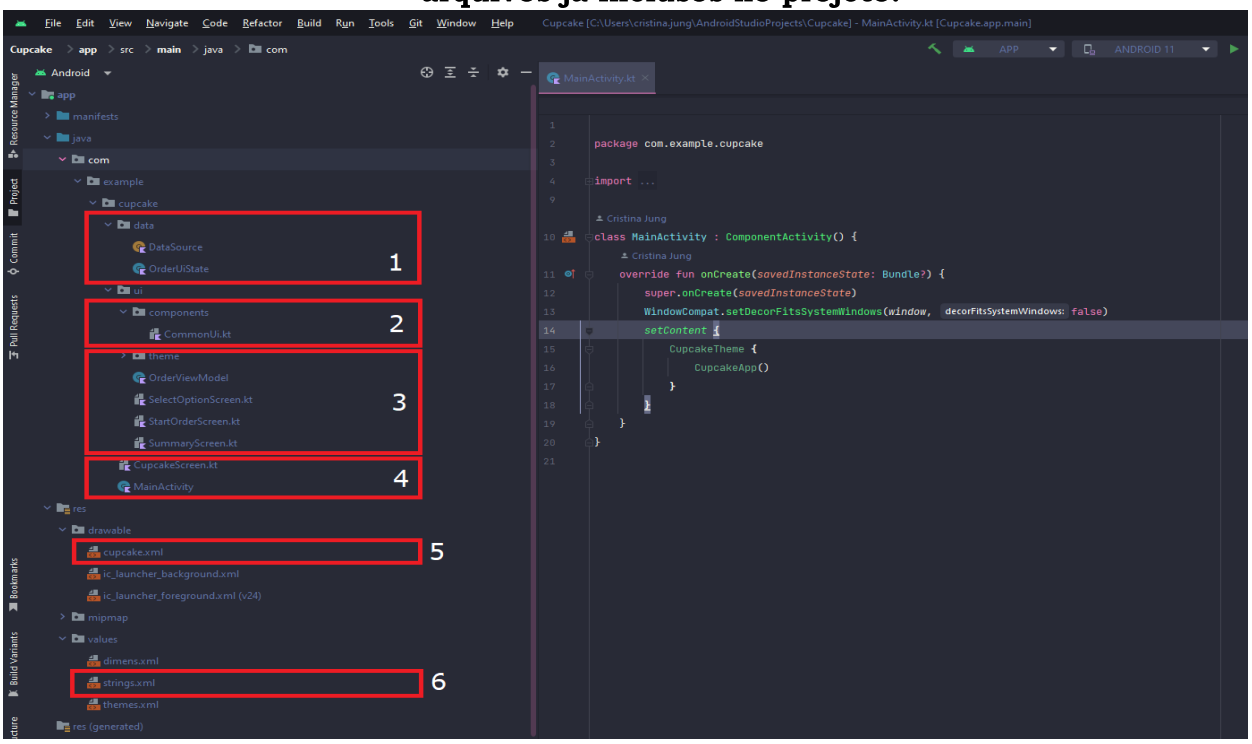
3.2 *Obtendo e entendendo o código inicial*

Neste link, você encontrará o repositório para fazer *download* do código inicial. As referências de pesquisa sempre estão no readme do repositório. Neste sentido:

1. Faça download do repositório.
2. Descompacte.
3. Abra no Android Studio.

Observe que o projeto ainda está com poucas definições na classe MainActivity, porém a estrutura para que possamos fazer as configurações da navegação usando o Compose já está pronta. Todos os códigos já declarados estão com comentários especificando a funcionalidade de cada um deles.

Agora, observe a próxima imagem (*cupcake*), onde encontramos a definição dos arquivos já inclusos no projeto.



Legendas da imagem:

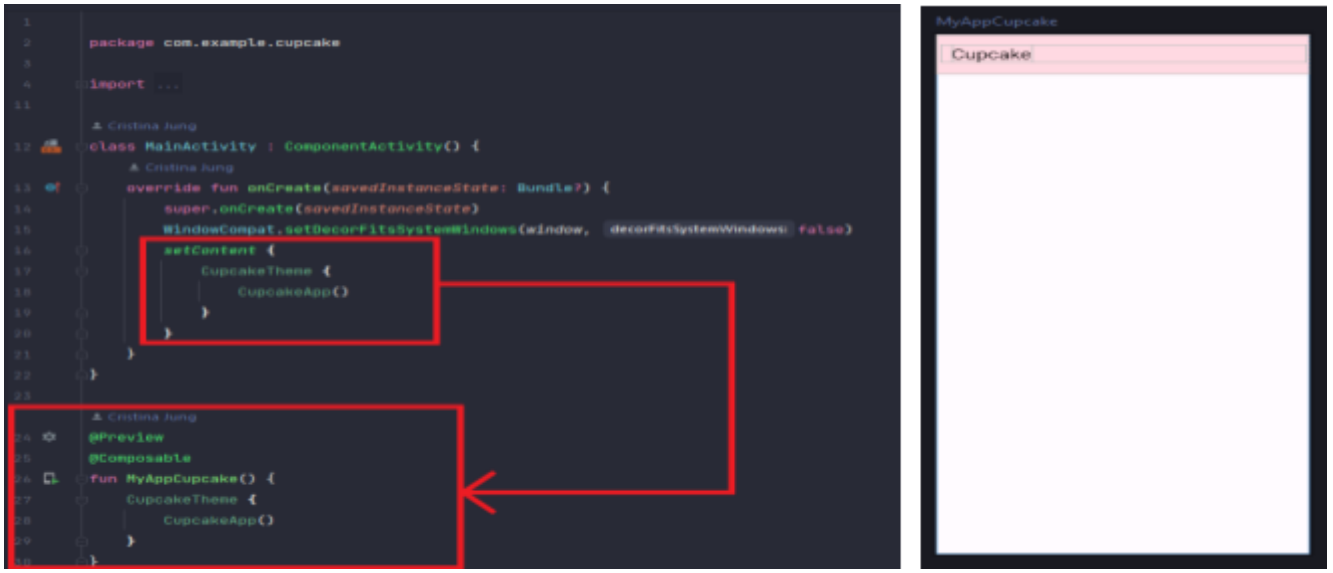
1. **Arquivos:**
   a. **Objeto DataSource:** contém as listas de sabores e a quantidade de opções dos cupcakes.
   b. **Data Class OrderUiState:** data classe que representa o estado inicial da UI.
2. **Arquivo e Package:**
   a. **Package ui/Component com a classe Kotlin CommomUi.kt:** Um componente Composable que exibe o preço formatado que será formatado e exibido na tela.

3. **Arquivos e Package:**
   a. **Package theme.**
   b. **Data Class OrderViewModel:** contém informações sobre um pedido de cupcake em termos de quantidade, sabor e data de retirada. Também tem o cálculo do preço total, com base nos detalhes do pedido.
   c. **Classe SelectOpitonsScreen.kt:** composable que exibe a lista de itens, como opções de cupcakes.
   d. **Classe StartOrderScreen.kt:** composable que permite ao usuário selecionar a quantidade desejada de cupcake, e espera 'onNex Button Clicked'--> lambda que espera a quantidade selecionada, e aciona a navegação para a próxima tela.
   e. **Classe SummaryScreen.kt:** composable que espera 'orderUiState', representando o estado do pedido, 'onCancelButtonClicked' → lambda que aciona o cancelamento do pedido, e passa o pedido final para 'onSendButtonClicked'.
4. **Arquivos e Package:**
   a. **cupcake/Classe CupcakeScreen.kt:** composable que exibe o topBar, e o botão Voltar se a navegação de volta for possível.
   b. **cupcake/Classe MainActivity.kt:** a classe principal que está à espera do código.
5. **Na pasta 'res':**
   a. **drawable/cupcake.xml:** contém as imagens, em SVG do aplicativo.
6. **Em values, no arquivo strings.xml encontramos os textos que serão exibidos no app.**

3.3 Configurando o projeto

A primeira coisa que precisamos fazer, será gerar uma visualização do início do nosso App de Cupcake. Para isso, após abrir o projeto que foi feito download do repositório, vamos acessar a classe MainActivity e declarar a visualização da nossa classe de interface.

Declaração @Preview	Renderização



Fonte da imagem: autoria própria, 2023.

A função **CupcakeApp()** se encontra na classe **CupcakeScreen** e é ela que define o início do layout que vamos usar.

3.4 Configurando o componente de navegação - NavHostController

NavHostController é uma classe que atua como um controlador de navegação, responsável por gerenciar o estado de navegação em um aplicativo Compose. O NavHostController permite que possamos realizar diversas ações de navegação, como navegar para frente (avançar para uma nova tela), voltar (retornar à tela anterior), passar argumentos entre telas e muito mais.

3.4.1 O componente de navegação tem três partes principais:

1. **NavController:** responsável por navegar entre os destinos, ou seja, as telas do nosso app.
2. **NavGraph:** mapeia os destinos de composição para navegar.
3. **NavHost:** elemento de composição que funciona como um contêiner para mostrar o destino atual do **NavGraph.**

3.4.2 Trabalhando com as rotas do App

Em um aplicativo Compose, um dos conceitos essenciais relacionados à navegação é a ideia de uma "rota". **Basicamente, uma rota é uma representação em forma de string associada a um destino específico no aplicativo.** Podemos pensar nisso de maneira semelhante ao conceito de URLs, em um site. Assim como URLs diferentes levam a páginas diferentes em um site, uma rota é uma string que leva a um destino e funciona como uma

espécie de identificador exclusivo para esse destino. Geralmente, um destino corresponde a um único componente ou a um grupo de componentes de Compose, que compõem o que o usuário vê na tela. Por exemplo, no aplicativo Cupcake, precisaríamos de destinos para as telas, que serão:

1. Início do pedido;
2. Seleção do sabor;
3. Escolha da data de retirada;
4. Visualização do resumo do pedido.

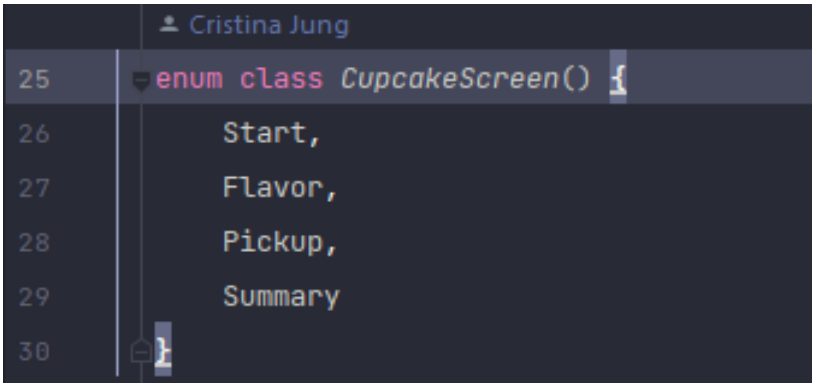
É importante destacar que, uma vez que um aplicativo tem um número limitado de telas, também há um número finito de rotas correspondentes a essas telas. Uma abordagem comum, para definir essas rotas em um aplicativo Compose é por meio do uso de uma classe de enumeração. **No Kotlin, as classes de enumeração possuem uma propriedade de nome, que retorna uma string com o nome da própria propriedade, tornando mais simples a associação das rotas aos destinos, dentro do aplicativo.**

Vamos, agora, definir as rotas baseadas nas telas que iremos declarar e com o resumo das funcionalidades.

1. **Start:** Selecione um dos três botões para selecionar a quantidade de cupcakes.
2. **Flavor:** Selecionar o sabor em uma lista de opções.
3. **Pickup:** Selecionar a data de retirada em uma lista de opções.
4. **Summary:** Revisar as seleções e enviar ou cancelar o pedido.

Vamos acessar o arquivo **CupcakeScreen.kt,** e declarar uma classe enum. Reforçando que, uma classe enum no Kotlin é uma estrutura de dados especializada, que permite definir um conjunto fixo de valores constantes nomeados.

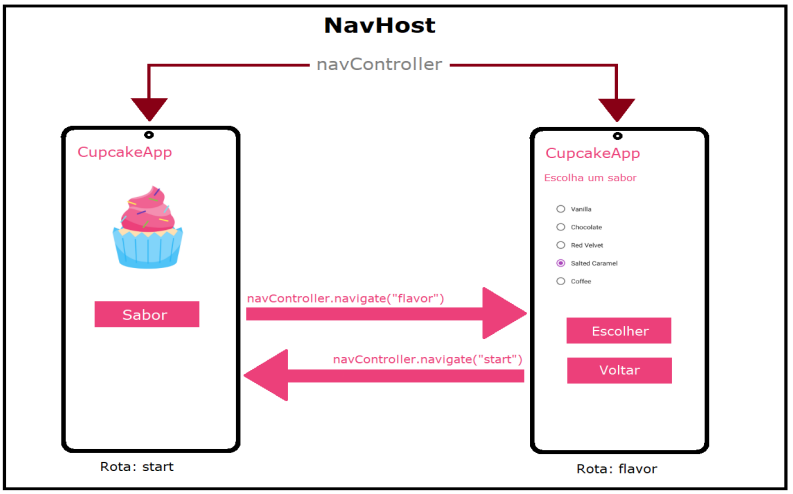
Declare a classe enum no início do código, antes do **@Composable CupcakeAppBar,** logo após o comentário de descrição da funcionalidade do arquivo.



Fonte da imagem: autoria própria, 2023

3.4.3 *NavHost*

Um NavHost é um Composable que atua como um hospedeiro (como o próprio nome se refere) para outros composables, dependendo da rota especificada. Por exemplo, quando a rota é definida como "Sabor", o NavHost exibe a tela de seleção de sabores de cupcakes. No entanto, se a rota for "Resumo", o aplicativo mostrará a tela de resumo do pedido. **O NavHost desempenha um papel importante ao direcionar e apresentar dinamicamente o conteúdo apropriado com base nas rotas definidas.**



Fonte da imagem: autoria própria, 2023

Como podemos ver na imagem anterior, foi utilizado um parâmetro para setar a rota desejada no app, mas existe um segundo parâmetro também:

- ✔ **navController:** é uma instância da classe NavHostController. É possível usar esse objeto para navegar entre telas, por exemplo, chamando o método **navigate()** para navegar para outro destino.
- ✔ **startDestination:** é uma rota de string, que define o destino mostrado por padrão quando o app mostra o NavHost pela primeira vez. No caso do app Cupcake, é a rota Start.

3.4.3.1 *Configurando o NavHost*

A navegação com Jetpack Compose tem a função composable() que é usada para definir o conteúdo que será exibido, em uma determinada tela ou destino dentro do nosso app. Vamos analisar os dois parâmetros que são obrigatórios:

✔ **route:** este é um parâmetro do tipo String, que corresponde ao nome de uma rota no nosso aplicativo. Uma rota é basicamente um identificador exclusivo para uma tela específica. Podemos pensar nisso, como um caminho que o usuário pode seguir para chegar a uma tela específica. Geralmente, as rotas são definidas como strings exclusivas, que facilitam a identificação das telas. Por exemplo, podemos definir uma rota chamada "home" para a tela inicial do app e outra chamada "profile" para a tela de perfil do usuário.

✔ **content:** este é o parâmetro onde especificamos, qual elemento de composição (composable) deve ser exibido na tela correspondente à rota definida. Um elemento de composição é uma unidade de interface de usuário, em Jetpack Compose. Pode ser um botão, um texto, uma lista, ou qualquer outra coisa que queremos exibir. **A função composable()** permite a vinculação de uma rota a um elemento de composição específico. Isso significa que quando o usuário navegar para essa rota, o conteúdo definido no **content** será exibido na tela.

No nosso projeto de Cupcake, iremos chamar a **função composable() uma vez para cada uma das rotas**.

Observe na próxima imagem que estamos usando a função composable() para definir a tela (destino), que corresponde à rota “Start”. Logo após, estamos declarando qual o conteúdo é uma chamada para **StartOrderScreen com um parâmetro quantityOptions que está sendo passado**. Isso implica que a tela de início (ou "**StartOrderScreen**") está sendo exibida quando o usuário acessa essa rota e está recebendo **quantityOptions** como dados. **O parâmetro quantityOptions se refere ao collectAsState() na linha antes da NavHost.**

Faça a conferência de seu código da função Cupcake:

```
@Composable
fun CupcakeApp(
    //val navController = rememberNavController(),
    viewModel: OrderViewModel = viewModel(),
    navController: NavHostController = rememberNavController()
) {

    Scaffold(
        topBar = {
            CupcakeAppBar(
                canNavigateBack = false,
                navigateUp = { /* TODO: implement back navigation */ }
            )
        }
    ) { innerPadding ->
        val uiState by viewModel.uiState.collectAsState()
        NavHost(
            navController = navController,
            startDestination = CupcakeScreen.Start.name,
            modifier = Modifier.padding(innerPadding)
        ) {

            composable(route = CupcakeScreen.Start.name) {
                StartOrderScreen(
                    quantityOptions = quantityOptions
                )
            }
            composable(route = CupcakeScreen.Flavor.name) {
                val context = LocalContext.current
                SelectOptionScreen(
                    subtotal = uiState.price,
                    options = flavors.map { id -> context.resources.getString(id) },
                    onSelectionChanged = { viewModel.setFlavor(it) }
                )
            }

            composable(route = CupcakeScreen.Pickup.name) {
                SelectOptionScreen(
                    subtotal = uiState.price,
                    options = uiState.pickupOptions,
                    onSelectionChanged = { viewModel.setDate(it) }
                )
            }

            composable(route = CupcakeScreen.Summary.name) {
                OrderSummaryScreen(
                    orderUiState = uiState
                )
            }
            //final dos composable do NavHost
        }
    }
}
```

3.4.3.2 *Configurando o "Navegar entre rotas"*

Agora que nós definimos nossas rotas e mapeamos para elementos de composição em um NavHost, é hora de navegarmos entre as telas. **O NavHostController, a propriedade navController proveniente de chamar rememberNavController(), é responsável pela navegação entre as rotas.** No entanto, essa propriedade está definida no elemento de

composição CupcakeApp. Nós precisamos de uma maneira de acessá-la nas diferentes telas do app. Para isso, basta transmitir o navController como um parâmetro para cada elemento de composição.

A lógica de navegação é mantida em um só lugar, o que pode facilitar a manutenção do código e impedir bugs, porque as telas individuais não perdem acidentalmente a navegação no app. Em apps que precisam funcionar em diferentes formatos, como smartphones no modo retrato, smartphones dobráveis ou tablets de tela grande, um botão pode ou não acionar a navegação, dependendo do layout do app. As telas individuais precisam ser autônomas, e não precisam estar cientes de outras telas no app. Todo este comportamento lógico será passado pelo NavHost.

As configurações que faremos nesta undiade, se referem mais aos direcionamentos de eventos e principalmente às configurações dos botões.

Configurando gerenciadores de botões no arquivo/classe StartOrderScreen

Passo 1 - Abrir o arquivo StartOrderScreen:

No **@Composable StartOrderScreen()**, vamos adicionar os seguintes parâmetros:

```
Cristina Jung *
35  @Composable
36  fun StartOrderScreen(
37      quantityOptions: List<Pair<Int, Int>>,
38      onNextButtonClicked: (Int) → Unit,
39      modifier: Modifier = Modifier
40  ){
```

Cada botão corresponde a uma quantidade diferente de cupcakes. Precisamos dessas informações, para que a função passada para **onNextButtonClicked** possa atualizar o modelo de visualização de acordo com elas.

Vamos observar que → para que o **Int** seja passado ao chamar **onNextButtonClicked()**, iremos analisar o tipo do parâmetro quantityOptions:

- ✔ O tipo é **List<Pair<Int, Int>>** ou uma lista de **Pair<Int, Int>**. Isto significa que é um par de valores do tipo Int. Cada item em um par é acessado pela primeira ou pela segunda propriedade.
- ✔ No caso do **parâmetro quantityOptions do elemento de composição StartOrderScreen**, o **primeiro int é um ID** de recurso para a string ser mostrada em cada botão. O **segundo int é a quantidade real de cupcakes.**

Passo 2 - Ainda no @Composable StartOrderScreen, vamos passar uma expressão lambda para o parâmetro onClick:

```
70          ) { this: ColumnScope
71              quantityOptions.forEach { item →
72                  SelectQuantityButton(
73                      labelResourceId = item.first,
74                      onClick = { onNextButtonClicked(item.second) }
75                  )
76              }
```

Quando o botão é clicado, essa função é ativada e chama outra função chamada "**onNextButtonClicked**," passando o valor inteiro associado ao botão da lista "quantityOptions" como argumento. Isso permite que o valor correto seja processado quando o botão é clicado.

Passo 3. Verifique se o @Preview da classe está conforme a próxima imagem:

```
Cristina Jung *
100 @Preview
101 @Composable
102 fun StartOrderPreview(){
103     StartOrderScreen(
104         quantityOptions = DataSource.quantityOptions,
105         onNextButtonClicked = {},
106         modifier = Modifier.fillMaxSize().padding(dimensionResource(R.dimen.padding_medium))
107     )
108 }
```

Configurando gerenciadores de botões no arquivo/classe SelectOptionsScreen:

Passo 1 - Configurar os parâmetros da @Composable SelectOptionsScreen():

```
41      onNextButtonClicked: () → Unit = {},
42      modifier: Modifier = Modifier
43  ){
```

Vamos observar aqui que adicionamos o parâmetro **onCancelButtonClicked e o tipo () -> Unit.**

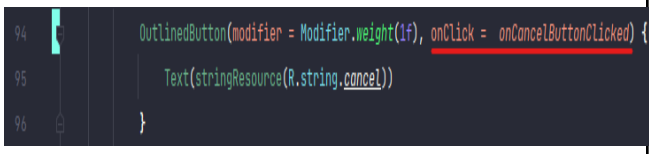
Estes parâmetros são usados para permitir que o usuário da função ou classe personalize o comportamento de eventos específicos, como uma seleção de item, o clique em um botão de cancelamento ou o clique em um botão de próxima etapa.

Estes parâmetros do Kotlin, representam funções que podem ser passadas como argumentos para outras funções ou classes.

- **onSelectionChanged** é uma função de retorno de chamada que aceita uma única string como argumento e não retorna nenhum valor (representado por Unit). O = {} no final significa que, por padrão, essa função é inicializada como uma função vazia que não faz nada.

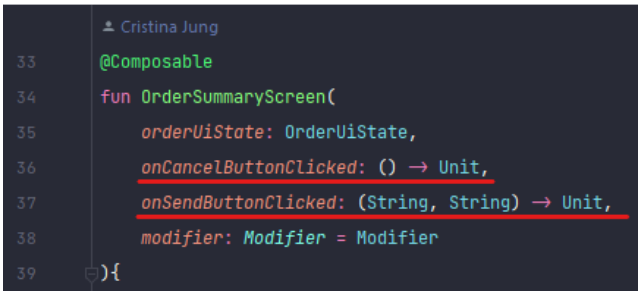
- **onCancelButtonClicked: () -> Unit = {}:** é uma função de retorno de chamada que não aceita nenhum argumento (representado por ()) e também não retorna nenhum valor (Unit). Da mesma forma, é inicializada como uma função vazia por padrão.
- **onNextButtonClicked: () -> Unit = {}:** O parâmetro onNextButtonClicked é semelhante aos outros dois. É uma função de retorno de chamada que não aceita argumentos e não retorna nenhum valor. Novamente, por padrão, é uma função vazia.

Passo 2 - Configurar OutLinedButton e Button:

OutlinedButton	Button
	

<u>Configurando gerenciadores de botões no arquivo/classe SummaryScreen</u>:

Passo 1- Adicionando parâmetros de onCancelButtonClicked que é uma função de callback que não recebe nenhum argumento e não retorna nada (() -> Unit). E **onSendButtonClicked** é uma função de callback que recebe dois argumentos de tipo String.



Ambas as funções serão chamadas quando o botão de envio ou de cancelamento for clicado na tela da OrderSummaryScreen. Os dois argumentos representam informações relacionadas ao envio (por exemplo, endereço de entrega, método de envio, etc.).

Passo 2. Configurando botões da OrderSummaryScreen: neste passo iremos passar **onSendButtonClicked ao parâmetro onClick do botão Send.** Passar newOrder e orderSummary, as duas variáveis definidas anteriormente em OrderSummaryScreen. Essas strings consistem nos dados reais que o usuário pode compartilhar com outro app.

```
Button(
    modifier = Modifier.fillMaxWidth(),
    onClick = { onSendButtonClicked(newOrder, orderSummary) }
) { this: RowScope
    Text(stringResource(R.string.send))
}
OutlinedButton(
    modifier = Modifier.fillMaxWidth(),
    onClick = onCancelButtonClicked
) { this: RowScope
    Text(stringResource(R.string.cancel))
}
```

Estes botões estão recebendo a implementação das funções onSendButtonClicked e onCancelButton Clicked, e da lógica que está por trás delas (que declaramos no passo anterior). O primeiro botão envia uma solicitação, enquanto o segundo botão, lida com o cancelamento.

3.5 Navegando por outras rotas usando método navigate()

A função **navigate()** em uma instância de **NavHostController,** é usada para navegar entre destinos (telas) em um aplicativo Android que utiliza o **componente de navegação da Jetpack Navigation**. Fornecemos o ID do destino que deseja navegar como argumento para navigate(), e o sistema de navegação cuida da transição e da exibição da tela de destino correspondente. Isso permite que possamos controlar a navegação de forma programática no app Android. 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.

A próxima etapa que faremos será passar funções que chamam o navigate() quando o usuário pressionar as telas Start, Flavor e Pickup.

Vamos observar as alterações no código da classe CupcakeScreen.kt:

```
//passo 1
composable(route = CupcakeScreen.Start.name) { it: NavBackStackEntry
    StartOrderScreen(
        quantityOptions = DataSource.quantityOptions,
        onNextButtonClicked = { it: Int
            viewModel.setQuantity(it)
            navController.navigate(CupcakeScreen.Flavor.name)
        },
        modifier = Modifier
            .fillMaxSize()
            .padding(dimensionResource(R.dimen.padding_medium))
    )
}
```

A função "onNextButtonClicked", é um callback que é chamado quando um botão "Next" é clicado na tela de início de pedido. Dentro deste callback, a quantidade selecionada é definida no ViewModel usando "viewModel.setQuantity(it)". Além disso, a navegação para a próxima tela é realizada usando: "navController.navigate (Cupcake Screen.Flavor.name)",onde "Cupcake Screen .Flavor.name" é a rota da tela de seleção do sabor de cupcake.

```
106         //passo 2
107         composable(route = CupcakeScreen.Flavor.name) { it: NavBackStackEntry
108             val context = LocalContext.current
109             SelectOptionScreen(  //passo 3
110                 subtotal = uiState.price,
111                 onNextButtonClicked = {
112                     navController.navigate(CupcakeScreen.Pickup.name) },
113                 onCancelButtonClicked = {},
114                 options = flavors.map { id → context.resources.getString(id) },
115                 onSelectionChanged = { viewModel.setFlavor(it) }
116             )
117         }
```

- ✔ **onNextButtonClicked = { navController.navigate(CupcakeScreen.Pickup.name) }:** Quando o botão "Next" é clicado na tela de seleção de sabores, a função navController.navigate é chamada para navegar para a próxima tela, que tem a rota "CupcakeScreen.Pickup.name". Isso faz avançar para a tela de seleção do método de retirada ou entrega.
- ✔ **onCancelButtonClicked = {}:** É fornecida uma função vazia para o evento de clique no botão de cancelamento. Portanto, neste caso, o botão de cancelamento não tem nenhuma ação associada.
- ✔ **options = flavors.map { id -> context.resources.getString(id) }:** Aqui, a lista de sabores de cupcakes, representada por "flavors", é mapeada para suas representações de string usando o contexto atual do app.
- ✔ **onSelectionChanged = { viewModel.setFlavor(it) }:** Quando o usuário seleciona um sabor na tela, a função viewModel.setFlavor(it) é chamada para definir o sabor escolhido no ViewModel do aplicativo. Isso permite que o aplicativo mantenha o controle do sabor selecionado pelo usuário.

```
118         //passo 4
119         composable(route = CupcakeScreen.Pickup.name) { it: NavBackStackEntry
120             SelectOptionScreen( //passo 5
121                 subtotal = uiState.price,
122                 onNextButtonClicked = {
123                     navController.navigate(CupcakeScreen.Summary.name) },
124                 onCancelButtonClicked = {},
125                 options = uiState.pickupOptions,
126                 onSelectionChanged = { viewModel.setDate(it) }
127             )
128         }
```

- **onNextButtonClicked = {navController.navigate(CupcakeScreen.Summary.name) }:** Quando o botão "Next" é clicado na tela de seleção de opções, a função navController.navigate é chamada para navegar para a próxima tela, que tem a rota "CupcakeScreen.Summary.name". Isso faz avançar para a tela de resumo do pedido.

- **onSelectionChanged = {viewModel.setDate(it) }:** Quando o usuário seleciona uma opção na tela, a função viewModel.setDate(it) é chamada para definir a data selecionada no ViewModel do aplicativo. Isso permite que o aplicativo mantenha o controle da data selecionada pelo usuário.

```
129         //passo 6
130         composable(route = CupcakeScreen.Summary.name) { it: NavBackStackEntry
131             val context = LocalContext.current
132             OrderSummaryScreen(
133                 orderUiState = uiState,
134                 onCancelButtonClicked = {},
135                 onSendButtonClicked = { subject: String, summary: String →
136                 }
137             )
```

3.6 *Retornando à tela inicial*

Estamos finalizando nosso app e para isso, precisamos agora configurar o retorno à tela inicial do aplicativo. Vamos usar o **método popBackStack()** que se utiliza de dois parâmetros obrigatórios:

- ✔ **route:** string que representa a rota do destino para onde queremos navegar de volta.
- ✔ **inclusive:** um valor booleano que, se for verdadeiro, também vai destacar (remover) a rota especificada. Se for falso, popBackStack() vai remover todos os destinos acima do destino inicial, mas não sem o incluir, deixando-o como a tela na camada superior visível para o usuário.

Após a fun CupcakeApp, vamos declarar o código:

```
144     //Redefine o [OrderUiState] e aparece em [CupcakeScreen.Start]
        Cristina Jung
145     private fun cancelOrderAndNavigateToStart(
146         viewModel: OrderViewModel,
147         navController: NavHostController
148     ) {
149         viewModel.resetOrder()
150         navController.popBackStack(CupcakeScreen.Start.name, inclusive = false)
151     }
```

Vamos observar que, navController.popBackStack(CupcakeScreen.Start.name, inclusive = false) → Usa o navController para fazer a transição de volta para a tela inicial (CupcakeScreen.Start.name) e, opcionalmente, exclui todas as telas empilhadas na parte superior da pilha de navegação. Isso é útil para cancelar o pedido e retornar à tela de início, descartando todas as etapas intermediárias da navegação.

Agora precisamos chamar este parâmetro nos elemento de **@Composable CupcakeApp()**, passando **cancelOrderAndNavigateToStart** para os parâmetros **onCancelButtonClicked dos dois elementos SelectOptionScreen e o OrderSummaryScreen.**

```
106        //passo 2
107        composable(route = CupcakeScreen.Flavor.name) { it: NavBackStackEntry
108            val context = LocalContext.current
109            SelectOptionScreen(  //passo 3
110                subtotal = uiState.price,
111                onNextButtonClicked = {
112                    navController.navigate(CupcakeScreen.Pickup.name) },
113                onCancelButtonClicked = { //aqui
114                    cancelOrderAndNavigateToStart(viewModel, navController)
115                },
116                options = flavors.map { id → context.resources.getString(id) },
117                onSelectionChanged = { viewModel.setFlavor(it) }
118            )
119        }
```

Fonte da imagem: autoria própria, 2023

Algumas alterações de códigos que foram feitas:

No arquivo:
CommonUi.kt, para exibir os valores em moeda local:

```
package com.example.cupcake.ui.components

import ...

// Composable que exibe [preco] formatado que será formatado e exibido na tela
Cristina Jung
@Composable
fun FormattedPriceLabel(subtotal: String, modifier: Modifier = Modifier) {
    val priceWithSymbol = "R$ $subtotal"
    Text(
        text = stringResource(R.string.subtotal_price, subtotal),
        modifier = modifier,
        style = MaterialTheme.typography.headlineSmall
    )
}
```

Fonte da imagem: autoria própria, 2023

No arquivo:
CupcakeScreen.kt para exibir os textos no botões

```
41      //alteração do enum ----
        Cristina Jung
42      enum class CupcakeScreen(@StringRes val title: Int) {
43          Start(title = R.string.app_name),
44          Flavor(title = R.string.choose_flavor),
45          Pickup(title = R.string.choose_pickup_date),
46          Summary(title = R.string.order_summary)
47      }
```

Fonte da imagem: autoria própria, 2023

4. Configurando a AppBar

Para terminar o nosso App e deixá-lo padrão, vamos finalizar com a configuração da AppBar. A AppBar no Android é um componente de interface do usuário que é muito usado para exibir informações de navegação, ações disponíveis e outros elementos relevantes em aplicativos Android.

Vamos às alterações que faremos na classe CupcakeScreen e suas respectivas explicações:

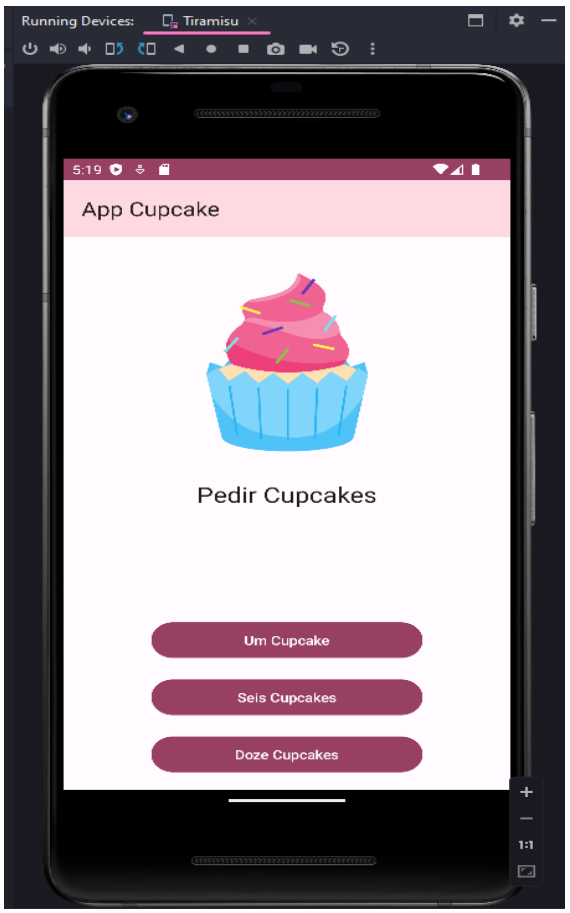
Passo 1:

```
@Composable
fun CupcakeApp(
    //val navControlle = rememberNavController(),
    viewModel: OrderViewModel = viewModel(),
    navController: NavHostController = rememberNavController()
) {
    // Obtendo a entrada atual da atual da backstack --- aqui
    val backStackEntry by navController.currentBackStackEntryAsState()
    // Pegando o nome da tela atual
    var currentScreen = CupcakeScreen.valueOf(
        value: backStackEntry?.destination?.route ?: CupcakeScreen.Start.name
    )
```

Passo 2:

```
87      Scaffold(
88          topBar = {
89              CupcakeAppBar( //cri
90                  currentScreen = currentScreen,
91                  canNavigateBack = navController.previousBackStackEntry ≠ null,
92                  navigateUp = { navController.navigateUp() }
93              )
94          }
```

Visão final do app:



Para obter o código final para correção, acesse:

- ✔ App de sobremesas
- ✔ App de cupcake

5. Referências

Múltiplos autores: **KOTLIN DOCS**. 2023. Disponível em: <https://kotlinlang.org/docs/home.html>. Acesso em: 17 de julho de 2023

Múltiplos autores: **JETBRAINS**. 2022. Disponível em: <https://www.jetbrains.com/pt-br/>. Acesso em: 18 de julho de 2023

Múltiplos autores: **DOCUMENTAÇÃO ANDROID STUDIO**. 2023. Disponível em: <https://developer.android.com/studio>. Acesso em: 18 de julho de 2023

Múltiplos autores: **ESTADO E JETPACK COMPOSE - ANDROID DEVELOPERS.** 2023. Disponível em: <Estado e Jetpack Compose | Android Developers>. Acesso em: 13 de agosto de 2023

Múltiplos autores: **ENTENDA O CICLO DE VIDA DA ACTIVITY - ANDROID DEVELOPERS.** 2023. Disponível em: <Entenda o ciclo de vida da atividade | Desenvolvedores Android>. Acesso em: 30 de agosto de 2023.

Múltiplos autores: **COMEÇAR A USAR O COMPONENTE DE NAVEGAÇÃO - ANDROID DEVELOPERS.** 2023. Disponível em: <Começar a usar o componente de navegação | Desenvolvedores Android>. Acesso em: 03 de setembro de 2023.

Múltiplos autores: **COMO NAVEGAR COM O COMPOSE - ANDROID DEVELOPERS.** 2023.Disponível em: <Como navegar com o Compose - Jetpack>. Acesso em: 03 de setembro de 2023.

Jalan, Nishant Aanjaney. **UNDERSTAND JETPACKCOMPOSE NAVIGATION**. Medium. 2022. Disponível em: <Understand Jetpack Compose Navigation in 3 Minutes | by Nishant Aanjaney Jalan | Medium>. Acesso em: 03 de setembro de 2023.