The Complete Android 14 and Kotlin Development Masterclass

Resumen de Contenido

<https://www.udemy.com/course/android-kotlin-developer>

# Kotlin Basics

Algunas de las cosas importantes a recordar sobre Kotlin son:

* Para variables cuyo valor no cambia (constante) se usa **val**.
* Para variables cuyo valor si cambia, usar **var**.
* Se manejan tipos como Int, Long, Short, Byte, Float, Double, Boolean y otros.
* En los if, se puede tener rangos que incluyan la palabra in, por ejemplo:

if (age in 18..39){

* Usar readln() para leer un valor de consola.
* Para declarar una función se usa la palabra fun.
* Una función con parámetros, define primero el nombre del parámetro y luego el tipo, por ejemplo:

fun makeCoffee(sugarCount: Int) {  
}

* Para usar variables al momento de desplegar un string, se usa el símbolo de dólar:

println("Coffee with $sugarCount spoons of sugar")

* Una function que retorna un valor, define el tipo de valor después de los paréntesis:

fun makeCoffee(sugarCount: Int): Double {

Algunos resumenes sobre estas lecciones:

<https://tutorials.eu/exploring-basic-kotlin-syntax-and-structure-day-2-android-14-masterclass/>

# Kotlin Classes

En Kotlin uno puede definir una clase con la palabra class, agregar funciones, y un inicializador (para ejecutar código al momento de instanciar la clase).

Las clases pueden recibir parámetros, si estos parámetros vienen precedidos de la palabra val, se vuelven propiedades de la clase, sin la necesidad de tener que declararlos como tal.

Las clases pueden tener más de un constructor.

En el constructor, se pueden enviar propiedades con valores por defecto, por ejemplo, age en en siguiente código.

**class** Dog (val name: String, var age: Int = 0) {  
 // Code that is executed the moment we instantiate the class  
 **init** {  
 bark(name)  
 }  
  
 **fun** bark(name: String){  
 *println*("$name says Woof!")  
 }  
}

## Data Class

En Kotlin, se pueden crear Data classes, cuyo propósito principal es soportar datos. Estas clases vienen automáticamente con funciones adicionales que le permiten imprimir una instancia a una salida legible, comparar instancias, copiar instancias y más y están marcadas por la palabra data.

data class ShoppingItem(val id:Int,  
 var name:String,  
 var quantity:Int,  
 var isEditing: Boolean = false)

<https://tutorials.eu/basic-kotlin-syntax-functions-objects-and-classes-in-kotlin-day-3-android-14-masterclass/>

# List and Objects

En Kotlin se pueden usar diferentes tipos de listas.

Se puede declarar una lista inmutable (sus valores no se pueden cambiar):

// Immutable List  
val shoppingList = *listOf*("CPU", "RAM", "SSD")

Se puede crear una lista mutable, donde se pueden agregar ítems después de definirla:

val shoppingList = mutableListOf("CPU", "RAM", "SSD")

Kotlin ofrece métodos para agregar o remover elementos de una lista, inclusive haciendo uso del índice.

El método set permite reemplazar un ítem en el índice especificado. Por ejemplo:

val shoppingList = *mutableListOf*("CPU", "RAM", "SSD")  
shoppingList.set(2, "Hard Drive")

Para verificar si una lista contiene un elemento en específico, se puede usar el método contains y pasar un elemento del tipo que contenga la lista.

Ejemplo de for para recorrer una lista:

for(item in shoppingList){  
 *println*(item)  
}

Otra forma de recorrer una lista pero haciendo uso del índice es esta:

for(item in 0 *until* shoppingList.size){  
 *println*(item)  
}

<https://tutorials.eu/kotlin-basics-for-loops-and-list-manipulations-day-4-android-14-masterclass/>

# Jetpack Compose Introduction

Jetpack Compose es el nuevo estándar adoptado por Google para la creación de aplicaciones de Android. Viene a reemplazar el uso de XML para generar la estructura que maneja el UI de la aplicación.

Una de las ventajas de JetPack Compose es que las vistas o los elementos de UI se escriben directamente en código, no se tiene la separación del archivo de XML y el código con el que se interactúa con dichos elementos por aparte. Todo se genere a través de “Composables”, que son funciones que generan elementos para el UI.

El siguiente es un ejemplo del MainActivity que se genera al crear un app nueva con JetPack Compose:

class MainActivity : ComponentActivity() {

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

*setContent* **{**

MyShoppingListTheme **{**

// A surface container using the 'background' color from the theme

Surface(  
 modifier = Modifier.*fillMaxSize*(),

color = MaterialTheme.colorScheme.background

) **{**

ShoppingListApp()

**}  
 }  
 }** }  
}

La clase hereda de ComponentActivity. El código inicial generalmente se pone en el onCreate.

Existe un setContent donde se usa el theme del app, uno que se puede customizar.

Dentro del código que invoca al Theme, se llama al Surface, que es básicamente la parte de background de la vista, ahí se define el color a utilizar y se pueden enviar modificadores, como el fillMaxSize para que tome todo el espacio disponible.

En el ejemplo, el ShoppingListApp es un composable donde se definen los elementos a presentar en la pantalla.

La apariencia del app se puede cambiar dependiendo del theme que uno quiera que tenga, como el darktheme o lighttheme.

## Composables

En JetPack, una función Composable es aquella que tiene el annotation ***@Composable***, y que no devuelve nada, pero si define un elemento a visualizar en pantalla. Estas funciones pueden recibir diversos parámetros, entre ellos está el modifier, que se puede utilizar el Modifier por defecto, pero en ocasiones se puede usar uno que ya tenga definidas ciertas propiedades que tienen que ver con diseño.

En Kotlin, los composables son funciones declarativas, facilitan tanto la composición como la recomposición. Composición se refiere a la representación inicial del UI, mientras que la recomposición se basa en la recalibración o a volver a mostrar los elementos del UI basados en cambios de datos o el estado.

Para poder ordenar los elementos en JetPack Compose, se utilizan elementos que sirven como contenedores y estos son: **Column** (ordena los elementos de forma vertical) y **Row** (ordena los elementos de forma horizontal).

## Text Fields

Para permitir al usuario ingresar Texto, hay 3 diferentes elementos:

* TextField
* BasicTextField
* OutlinedTextField

En el caso del último, se puede usar un valor String o un TextField para lo que se va a desplegar en pantalla y con el onValueChange, definimos la lógica de lo que pasa cuando el valor cambia. Este onValueChange es en sí una función anónima.

## Preview

En JetPack se puede tener un Composable con un annotation llamado @Preview, que permite hacer eso, llamar a otro composable, que generalmente es el composable que despliega todo el UI de la pantalla y mostrar un preview de como luce sin tener que correr el emulador. Cada cambio que se haga en el código se refleja automáticamente en el preview sin necesidad de tener que ejecutar el código.

Por convención, se usa el nombre del composable a mostrar más el sufijo Preview como nombre de la función preview. Por ejemplo:

@**Preview**(showBackground = true, showSystemUi = true)

@Composable

fun UnitConverterPreview() {

UnitConverterTheme { UnitConverter()

}

}

## Button

Como los demás elementos en JetPack, el button es otro composable, que trae un onClick para que uno defina la lógica al presionarlo.

El siguiente es un ejemplo de un Button, al que se le define que despliegue un Toast al presionarlo y cuyo texto es “Click me!”.

Box {

val context = LocalContext.current

Button(onClick = { Toast

.makeText(context, "Thanks for clicking!",

Toast.LENGTH\_LONG).show()

}){

Text("Click me!")

}

}

El onClick también espera una función, la cual puede ser una función anónima que en el caso anterior va a ejecutar un Toast.

## Box and DropDownMenu

Box es otro composable como el Row o Column, que permite que uno le agregue contenido, y el cual se ajusta a lo que contenga. El Box ordena los elementos encima uno del otro. Se usa generalmente para crear layouts complejos, por ejemplo, para usar con el Drowdown menú.

Para el siguiente ejemplo, se crea un botón que al presionarse, muestra un dropdown menú con diferentes opciones de medidas. El Boton muestra el valor de una variable llamada inputUnit junto con un icono de una flecha hacia abajo. Las imágenes que son importantes para el UI deben tener un texto descriptivo en el contentDescription.

Junto al Boton, se agrega el DropdownMenu, el cual es un composable que tiene una propiedad llamada expanded que especifica si el menú debe estar desplegado o no y en este caso está usando el valor de inputExpanded, que se cambia al presionar el botón. Tambien tiene un método llamado onDismissRequest, que básicamente vuelve el valor del inputExpanded a false cuando se “cierra” el dropdown.

Dentro del DropdownMenu, tenemos los diferentes DropdownMenuItems, que tienen un texto y un método onClick que define que hace cada uno, para el ejemplo, se cambian diferentes valores de variables y se llama a una funcion convertUnits.

Box {

// Input Button

Button(onClick = { inputExpanded = true }) {

Text(inputUnit)

Icon(Icons.Default.ArrowDropDown, contentDescription = "Arrow Down")

}

DropdownMenu(expanded = inputExpanded, onDismissRequest = { inputExpanded = false }) {

// Where the dropdownmenu items will be added

DropdownMenuItem(

text = { Text("Milimeters") },

onClick = {

inputExpanded = false

inputUnit = "Milimeters"

conversionFactor.value = 0.001

convertUnits()

}

)

DropdownMenuItem(

text = { Text("Centimeters") },

onClick = {

inputExpanded = false

inputUnit = "Centimeters"

conversionFactor.value = 0.01

convertUnits()

}

)

DropdownMenuItem(

text = { Text("Meters") },

onClick = {

inputExpanded = false

inputUnit = "Meters"

conversionFactor.value = 1.0

convertUnits()

}

)

}

}

## Spacer vs Padding

En JetPack Compose se pueden usar dos formas para dar espacio a los elementos.

Padding: es un modificador que se puede usar para dar espacio alrededor de los elementos.

Spacer: es un elemento como tal, con capacidad de ser reutilizable, y que suele usarse para agregar espacio entre multiples componentes.

Ejemplo:

Text("Unit Converter", style = MaterialTheme.typography.headlineLarge, color = DarkSecondary)

**Spacer**(modifier = Modifier.height(16.dp))

OutlinedTextField(

value = inputValue,

onValueChange = {

// Here goes what should happen when the value of OutlinedTextField changes

inputValue = it

convertUnits()

},

label = { Text("Enter value") })

**Spacer**(modifier = Modifier.height(16.dp))

<https://tutorials.eu/the-power-of-jetpack-compose-and-ui-customization-day-5-android-14-masterclass/>

# Understanding State

## Remember and MutableState

En JetPack se trabaja con el State. El manejo del estado permite manejar la “dirección” del app cuando ocurre un evento o se actualiza la data. Los eventos permiten informar y ajustar el estado del app. Los eventos y el estado trabajan juntos para que la aplicación permita tener una interacción suave para el usuario.

Hay dos elementos importantes al trabajar con el State:

* Remember function: se usa para crear un estado persistente. Permite mantener el estado a través de recomposiciones incluso cuando los composables se recrean. La recomposición es el proceso de generar y actualizar el UI para reflejar cambios en el app.
* Mutable State property delegate: es usado para crear un estado mutable que se puede actualizar. Generalmente usado en conjunto con el remember para manejar los cambios de estado.

Ejemplos de propiedades que usan el remember y el mutable state para almacenar diferentes valores:

var inputValue = remember { mutableStateOf("") }

var outputValue = remember { mutableStateOf("") }

var inputUnit by remember { mutableStateOf("Meters")}

var outputUnit by remember { mutableStateOf("Meters") }

var inputExpanded by remember { mutableStateOf(false) }

var outputExpanded by remember { mutableStateOf(false) }

Cuando un evento modifica alguno de estos valores, al hacer el recomposition del UI, los valores se mantienen haciendo que el app se comporte según los valores que se han actualizado.

El keyword **by** que se observa en algunas de las propiedades es una forma más sencilla de declararlo, esta segunda forma ocupa importar el State.getValue, que permite obtener el valor directamente. Si se declara sin el by, al tratar de obtener el valor, por ejemplo, de la variable inputValue, se necesita el .value, por ejemplo:

val inputValueDouble = inputValue.**value**.*toDoubleOrNull*() ?: 0.0

Si se declara con el **by**, no hace falta usar el value ya que el valor se obtiene directamente.

## Implementar el State

Trabajar con el state es como trabajar con variables, en el siguiente ejemplo, se declara una variable llamada inputValue que usa remember y el mutableStateOf y más abajo en el código se tiene un OutlinedTextField donde se le da el valor que haya ingresado el usuario mediante el onValueChange:

@Composable

fun UnitConverter(){

var inputValue by remember { mutableStateOf("") }

….

Column(

modifier = Modifier.fillMaxSize(),

verticalArrangement = Arrangement.Center,

horizontalAlignment = Alignment.CenterHorizontally

) {

// Here all the UI elements will be stacked below each other

Text("Unit Converter", style = MaterialTheme.typography.headlineLarge, color = DarkSecondary)

Spacer(modifier = Modifier.height(16.dp))

OutlinedTextField(

**value = inputValue**,

onValueChange = {

// Here goes what should happen when the value of OutlinedTextField changes

**inputValue = it**

convertUnits()

},

label = { Text("Enter value") })

El codigo completo del Proyecto UnitConverter:

class MainActivity : ComponentActivity() {

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

setContent {

UnitConverterTheme {

// A surface container using the 'background' color from the theme

Surface(

modifier = Modifier.fillMaxSize(),

color = MaterialTheme.colorScheme.background

) {

UnitConverter()

}

}

}

}

}

@OptIn(ExperimentalMaterial3Api::class)

@Composable

fun UnitConverter(){

var inputValue by remember { mutableStateOf("") }

var outputValue by remember { mutableStateOf("") }

var inputUnit by remember { mutableStateOf("Meters")}

var outputUnit by remember { mutableStateOf("Meters") }

var inputExpanded by remember { mutableStateOf(false) }

var outputExpanded by remember { mutableStateOf(false) }

// The multiplier we are going to apply to whatever the user has entered to give us

// the output with the data type that fits for the second value

val conversionFactor = remember { mutableStateOf(1.0) }

val outputConversionFactor = remember { mutableStateOf(1.0) }

fun convertUnits(){

// ?: = elvis operator

val inputValueDouble = inputValue.toDoubleOrNull() ?: 0.0

val result = (inputValueDouble \* conversionFactor.value \* 100.0 / outputConversionFactor.value).roundToInt() / 100.0

outputValue = result.toString()

}

Column(

modifier = Modifier.fillMaxSize(),

verticalArrangement = Arrangement.Center,

horizontalAlignment = Alignment.CenterHorizontally

) {

// Here all the UI elements will be stacked below each other

Text("Unit Converter", style = MaterialTheme.typography.headlineLarge, color = DarkSecondary)

Spacer(modifier = Modifier.height(16.dp))

OutlinedTextField(

value = inputValue,

onValueChange = {

// Here goes what should happen when the value of OutlinedTextField changes

inputValue = it

convertUnits()

},

label = { Text("Enter value") })

Spacer(modifier = Modifier.height(16.dp))

Row {

// Input Box

Box {

// Input Button

Button(onClick = { inputExpanded = true }) {

Text(inputUnit)

Icon(Icons.Default.ArrowDropDown, contentDescription = "Arrow Down")

}

DropdownMenu(expanded = inputExpanded, onDismissRequest = { inputExpanded = false }) {

// Where the dropdownmenu items will be added

DropdownMenuItem(

text = { Text("Milimeters") },

onClick = {

inputExpanded = false

inputUnit = "Milimeters"

conversionFactor.value = 0.001

convertUnits()

}

)

DropdownMenuItem(

text = { Text("Centimeters") },

onClick = {

inputExpanded = false

inputUnit = "Centimeters"

conversionFactor.value = 0.01

convertUnits()

}

)

DropdownMenuItem(

text = { Text("Meters") },

onClick = {

inputExpanded = false

inputUnit = "Meters"

conversionFactor.value = 1.0

convertUnits()

}

)

DropdownMenuItem(

text = { Text("Feet") },

onClick = {

inputExpanded = false

inputUnit = "Feet"

// A feet is 0.3048 meters

conversionFactor.value = 0.3048

convertUnits()

}

)

}

}

Spacer(modifier = Modifier.width(16.dp))

// Output box

Box {

// Output button

Button(onClick = { outputExpanded = true }) {

Text(outputUnit)

Icon(Icons.Default.ArrowDropDown, contentDescription = "Arrow Down")

}

DropdownMenu(expanded = outputExpanded, onDismissRequest = { outputExpanded = false }) {

// Where the dropdownmenu items will be added

DropdownMenuItem(

text = { Text("Milimeters") },

onClick = {

outputExpanded = false

outputUnit = "Milimeters"

outputConversionFactor.value = 0.001

convertUnits()

}

)

DropdownMenuItem(

text = { Text("Centimeters") },

onClick = {

outputExpanded = false

outputUnit = "Centimeters"

outputConversionFactor.value = 0.01

convertUnits()

}

)

DropdownMenuItem(

text = { Text("Meters") },

onClick = {

outputExpanded = false

outputUnit = "Meters"

outputConversionFactor.value = 1.0

convertUnits() }

)

DropdownMenuItem(

text = { Text("Feet") },

onClick = {

outputExpanded = false

outputUnit = "Feet"

// A feet is 0.3048 meters

outputConversionFactor.value = 0.3048

convertUnits()

}

)

}

}

}

Spacer(modifier = Modifier.height(16.dp))

// Result Text

Text("Result: $outputValue $outputUnit",

style = MaterialTheme.typography.headlineMedium,

color = DarkSecondary

)

}

}

@Preview(showBackground = true, showSystemUi = true)

@Composable

fun UnitConverterPreview() {

UnitConverterTheme {

UnitConverter()

}

}

<https://tutorials.eu/mastering-state-management-and-essential-kotlin-syntax-day-6-android-14-masterclass/>

# Loads of UI Features

## LazyColumns

Un LazyColumns es una columna que solo carga. O muestra cierta cantidad de ítems en pantalla y mientras el usuario hace scroll, nuevos ítems se cargan mientras que los que quedan fuera de pantalla se descartan, asegurando un mejor desempeño del app.

Por ejemplo, teniendo un composable llamado ShoppingListApp, donde definimos una lista mutable de elementos de un tipo custom, creamos un LazyColumn que tenga como ítems dicha lista:

data class **ShoppingItem**(val id:Int,

var name:String,

var quantity:Int,

var isEditing: Boolean = false)

@Composable

fun ShoppingListApp() {

// Create a var using remember to keep the list of shopping items

var **sItems** by remember { mutableStateOf(**listOf**<ShoppingItem>()) }

var showDialog by remember { mutableStateOf(false) }

Column(

modifier = Modifier.fillMaxSize(),

verticalArrangement = Arrangement.Center

)

{

Button(

onClick = { showDialog = true },

modifier = Modifier.align(Alignment.CenterHorizontally)

) {

Text("Add Item")

}

**LazyColumn**(

modifier = Modifier

.fillMaxSize()

.padding(16.dp)

) {

**items**(sItems) {

}

}

}

}

## AlertDialog

Un composable que sirve para desplegar un mensaje tipo pop-up que generalmente provee uno o varios botones para que el usuario interactúe con él.

Hay un par de implementaciones para el AlertDialog, en el ejemplo se usa uno que permite definir varias propiedades como el title, el text, el onDismissRequest y el confirmButton, estos dos últimos son requeridos. Aquí un ejemplo de la implementación:

if(showDialog){

AlertDialog(

onDismissRequest = { showDialog = false },

confirmButton = {

Row(modifier = Modifier.fillMaxWidth().padding(8.dp),

horizontalArrangement = SpaceBetween ){

// Add item button

Button(onClick = {

if(itemName.isNotBlank()){

// Set new item

val newItem = ShoppingItem(

id = sItems.size + 1,

name = itemName,

quantity = itemQuantity.toInt()

)

// Adding the new item to the list

sItems = sItems + newItem

// Close the dialog

showDialog = false

itemName = ""

}

}) {

Text("Add")

}

// Cancel button

Button(onClick = { showDialog = false }) {

Text("Cancel")

}

}

},

title = { Text("Add Shopping Item") },

text = {

Column {

OutlinedTextField(

value = itemName,

onValueChange = {itemName = it },

singleLine = true,

modifier = Modifier

.fillMaxWidth()

.padding(8.dp)

)

OutlinedTextField(

value = itemQuantity,

onValueChange = {itemQuantity = it },

singleLine = true,

modifier = Modifier

.fillMaxWidth()

.padding(8.dp)

)

}

}

)

}

Algunas cosas a notar:

* En el onDismisRequest simplemente se está pasando la variable showDialog a false para ocultar el dialog basado en un if donde se define el mismo.
* El confirmButton espera un @Composable, por lo que podemos definir una estructura con varios elementos, como en este caso que se usó un Row para definir otros dos botones y se le definió cierto estilo usando el modifier. El SpaceBetween permite acomodar los elementos dentro del row de forma que quede un espacio entre ellos pero no al inicio ni después del último elemento.
* La propiedad title define el texto principal o título del dialog.
* La propiedad text también espera un @Composable, por lo que aquí definimos el cuerpo del dialog, que en este caso es un Column que contiene un par de outlined text fields.

## Lambdas

Las lambdas son funciones anónimas cortas que se pueden recibir parámetros y devolver un tipo especifico si así se requiere, a cómo puede no tener parámetros y no devolver nada. Un ejemplo muy básico es:

() -> Unit

Esta función es un lambda que no recibe parámetros y no devuelve ningún tipo. Cuando se usa Unit significa que no necesita devolver nada.

Cuando se tiene un método con una firma como este:

@Composable

fun ShoppingListItem(

item: ShoppingItem,

onEditClick: () -> Unit,

onDeleteClick: () -> Unit

){

Al llamar al método, en los parámetros que esperan un lambda, se puede enviar una función como tal o bien, el código que se quiere correr como por ejemplo en otros métodos vistos anteriormente como el onClick o el onValueChange:

onValueChange = {itemName = it },

Otro ejemplo:

val doubleNumber:(Int) -> Int = { it \* 2 }

Text(doubleNumber(2).toString())

En el ejemplo anterior, definimos una variable que internamente es una función lambda, a esta se le especifica que recibe un parámetro Integer, después de la flecha se indica que devuelve un tipo Integer también y se define el código poniendo el igual y luego entre llaves lo que hace la función. El valor it es el valor que está recibiendo como parámetro, si la función tuviera más de un parámetro, sería necesario definir el nombre de los mismos.

## IconButtons

En JetPack existen también unos botones que permiten utilizar iconos, los IconButton, los cuales tienen un onClick para definir que van a hacer cuando se presionen y el icono se define mediante un composable llamado Icon. Ejemplo:

@Composable

fun ShoppingListItem(

item: ShoppingItem,

onEditClick: () -> Unit,

onDeleteClick: () -> Unit

){

Row(

modifier = Modifier

.padding(8.dp)

.fillMaxWidth()

.border(

border = BorderStroke(2.dp, Color(0xFF03A9F4)),

shape = RoundedCornerShape(20)

)

) {

//Name

Text(text = item.name, modifier = Modifier.padding(8.dp))

//Quantity

Text(text = "Qty: ${item.quantity}", modifier = Modifier.padding(8.dp))

Row(modifier = Modifier.padding(8.dp)){

**IconButton**(onClick = { onEditClick }) {

**Icon**(imageVector = Icons.Default.Edit, contentDescription = "Edit button for item")

}

**IconButton**(onClick = { onDeleteClick }) {

Icon(imageVector = Icons.Default.Delete, contentDescription = "Delete button for item")

}

}

}

}

En el ejemplo anterior, cada icon button utiliza una de las funciones que se pasó por parámetro al ShoppingListItem y define un Icon usando una librería propia de Android con diferentes iconos como el Edit o el Delete.

## Map Keyword

Map permite transformar una lista. Va iterando a través de los diferentes elementos de la lista para ir transformándolos. Ejemplo:

val numbers = listOf(1,2,3)

val doubled = numbers.map { it \* 2 } // Result [2,4,6]

## Copy Keyword

Copy permite hacer una copia de un objeto, lo cual hace que uno pueda hacer cambios a la copia sin que esto altere el objeto original. Por ejemplo:

data class Vase(val color: String, val design: String)

fun main(){

val blueRoseVase = Vase(color = "Blue", design = "Rose")

val redRoseVase = blueRoseVase.copy(color = "Red")

}

El objeto redRoseVase tendria el mismo valor de design que el blueRoseVase, pero su valor de color es diferente.

## Let and Nullable

Let permite transformer un objeto sin alterar su estado original. Por ejemplo:

// nullable string

val name: String? = "Ella"

name?.**let** {

println(it.toUpperCase())

}

Let se pude usar para invocar una o más funciones en el resultado de una cadena de llamadas.

## Project Code

Este es el codigo del proyecto ShoppingList, específicamente el ShoppingList.kt

/\*\*

\* Data class that represents every shopping item in the list app

\*\*/

data class ShoppingItem(val id:Int,

var name:String,

var quantity:Int,

var isEditing: Boolean = false)

@OptIn(ExperimentalMaterial3Api::class)

@Composable

fun ShoppingListApp() {

// Create a var using remember to keep the list of shopping items

var sItems by remember { mutableStateOf(listOf<ShoppingItem>()) }

// Define if the alert dialog should be displayed or not

var showDialog by remember { mutableStateOf(false) }

var itemName by remember { mutableStateOf("") }

var itemQuantity by remember { mutableStateOf("") }

Column(

modifier = Modifier.fillMaxSize(),

verticalArrangement = Arrangement.Center

)

{

Button(

onClick = { showDialog = true },

modifier = Modifier.align(Alignment.CenterHorizontally)

) {

Text("Add Item")

}

LazyColumn(

modifier = Modifier

.fillMaxSize() //use max space available

.padding(16.dp)

) {

items(sItems) {

item ->

if(item.isEditing){

ShoppingItemEditor(item = item,

onEditComplete = {

editedName, editedQuantity ->

// Set to false the isEditing

sItems = sItems.map { it.copy(isEditing = false) }

// Finds the item we are currently editing

val editedItem = sItems.find { it.id == item.id }

editedItem?.let {

it.name = editedName

it.quantity = editedQuantity

}

})

}else{

ShoppingListItem(item = item,

onEditClick = {

// Finding out which item we are editing and changing its "isEditing" to true

sItems = sItems.map{ it.copy(isEditing = it.id==item.id)}

},

onDeleteClick = {

sItems = sItems - item

})

}

}

}

}

if(showDialog){

AlertDialog(

onDismissRequest = { showDialog = false },

confirmButton = {

Row(modifier = Modifier

.fillMaxWidth()

.padding(8.dp),

horizontalArrangement = SpaceBetween ){

// Add item button

Button(onClick = {

if(itemName.isNotBlank()){

// Set new item

val newItem = ShoppingItem(

id = sItems.size + 1,

name = itemName,

quantity = itemQuantity.toInt()

)

// Adding the new item to the list

sItems = sItems + newItem

// Close the dialog

showDialog = false

itemName = ""

itemQuantity = ""

}

}) {

Text("Add")

}

// Cancel button

Button(onClick = { showDialog = false }) {

Text("Cancel")

}

}

},

title = { Text("Add Shopping Item") },

text = {

Column {

OutlinedTextField(

value = itemName,

onValueChange = {itemName = it },

singleLine = true,

modifier = Modifier

.fillMaxWidth()

.padding(8.dp)

)

OutlinedTextField(

value = itemQuantity,

onValueChange = {itemQuantity = it },

singleLine = true,

modifier = Modifier

.fillMaxWidth()

.padding(8.dp)

)

}

}

)

}

}

@Composable

fun ShoppingListItem(

item: ShoppingItem,

onEditClick: () -> Unit,

onDeleteClick: () -> Unit

){

Row(

modifier = Modifier

.padding(8.dp)

.fillMaxWidth()

.border(

border = BorderStroke(2.dp, Color(0xFF0370A2)),

shape = RoundedCornerShape(20)

),

horizontalArrangement = Arrangement.SpaceBetween,

verticalAlignment = Alignment.CenterVertically

) {

//Name

Text(text = item.name, modifier = Modifier.padding(8.dp))

//Quantity

Text(text = "Qty: ${item.quantity}", modifier = Modifier.padding(8.dp))

Row(modifier = Modifier.padding(8.dp)){

IconButton(onClick = onEditClick) {

Icon(imageVector = Icons.Default.Edit, contentDescription = "Edit button for item")

}

IconButton(onClick = onDeleteClick) {

Icon(imageVector = Icons.Default.Delete, contentDescription = "Delete button for item")

}

}

}

}

@Composable

fun ShoppingItemEditor(item: ShoppingItem, onEditComplete: (String, Int) -> Unit) {

var editedName by remember { mutableStateOf(item.name) }

var editedQuantity by remember { mutableStateOf(item.quantity.toString()) }

var isEditing by remember { mutableStateOf(item.isEditing ) }

Row(modifier = Modifier

.fillMaxWidth()

.background(Color.White)

.padding(8.dp),

horizontalArrangement = Arrangement.SpaceEvenly)

{

Column {

BasicTextField(

value = editedName,

onValueChange = { editedName = it},

singleLine = true,

modifier = Modifier

.wrapContentSize()

.padding(8.dp)

)

BasicTextField(

value = editedQuantity,

onValueChange = { editedQuantity = it},

singleLine = true,

modifier = Modifier

.wrapContentSize()

.padding(8.dp)

)

}

Button(onClick = {

isEditing = false

onEditComplete(editedName, editedQuantity.toIntOrNull() ?: 1)

//toIntOrNull means try to convert the value into int, if it cannot be converted, use 1

}) {

Text(text = "Save")

}

}

}

# MVVM – Model View ViewModel Architecture

La arquitectura MVVM es un patrón que busca la separación del UI (View) de la lógico del negocio o el back-end (modelo) de forma que la vista no dependa de un modelo especifico. El viewmodel convierte los data objects del modelo de forma que pueda ser utilizado por la vista.

Elementos del patron:

* **Model**: Representa la capa de datos y/o la lógica del negocio. El modelo contiene la información, pero nunca las acciones o servicios que lo manipulan. No tiene dependencia con la vista.
* **View**: Representa la información a través de los elementos visuales que la componen.
* **ViewModel**: Actor intermediario entre modelo y vista. Contiene la lógica de la presentación y se comporta como una abstracción de la interfaz.

## Clase ViewModel

Una clase ViewModel en Android actúa como un manager que maneja la comunicación entre la data y el UI del app. Es responsable de mantener y procesar todo la data que necesita el UI mientras se da el ciclo de vida de los activities y fragments del app.

## How Does ViewModel Work?

* Initialization: Un VM es usualmente inicializado en un activity o fragment. Sobrevive a cambios de configuración y es destruida cuando la actividad o el fragment es permanentemente removido.
* Data Handling: El VM obtiene data del model, la mantiene y la vista observa eta data para actualizar el UI.

Ejemplo:

import androidx.lifecycle.ViewModel

class MyViewModel : ViewModel() {

    var number: Int = 0

    fun incrementNumber() {

        number++

    }

}

En la clase anterior se crea un ViewModel llamada MyViewModel que tiene una función que incrementa el valor de una variable llamada number.

class MainActivity : AppCompatActivity() {

    private val myViewModel: MyViewModel by viewModels()

    override fun onCreate(savedInstanceState: Bundle?) {

        super.onCreate(savedInstanceState)

        setContentView(R.layout.activity\_main)

*// Use the ViewModel*

        myViewModel.incrementNumber()

    }

}

En el activity, se crea una instancia del VM y es inicializada usando el delegado **by viewModels**. Este delegado es usado para asociar el VM con el activity.

Cosas a recordar:

* VM no es un reemplazo de onSaveInstanceState, no maneja todos los tipos de cambios de configuración.

## Inheritance

Cuando se trabaja con herencia en Kotlin, la clase base o clase padre, necesita la palabra reservada open en la declaración de la clase.

Igualmente, si se quiere tener un método que se pueda sobreescribir, en la clase base la firma debe llevar el keyword open y en la clase que la hereda, se usa override antes del fun.

* La herencia ayuda a reutilizar y organizar el código eficientemente.
* La palabra open es esencial para permitir la herencia en Kotlin
* La clase derivada tiene acceso a los miembros públicos y protegidos (métodos y campos) de la clase base.

## Open Function

En Kotlin, el keyword open no se limita a clases, también se puede usar en funciones. Por defecto, las funciones en Kotlin son final, lo que significa que no se puede sobreescribir en una clase derivada.

Con el keyword open se permite que una función pueda ser sobre escrita.

Cuando se desea sobreescribir una función en la clase derivada, se usa el keyword override. Ejemplo:

Clase base:

open class Vehicle {

    open fun start() {

        println("The vehicle is starting.")

    }

}

Clase derivada:

class Car : Vehicle() {

    override fun start() {

        println("The car is starting with a roar!")

    }

}

## Super Keyword

El keyword super se usa dentro de las funciones sobre escritas para llamar a la función de la clase base. Permite que la clase derivada use la implementación original de la función.

## Interfaces

Las interfaces en Kotlin son una forma de definir contratos para las clases sin que implementen ningún comportamiento.

Pueden contener métodos abstractos (sin un cuerpo) y métodos con una implementación por default. Las clases que implementan una interfaz deben proveer implementación para todos sus métodos abstractos.

* Las interfaces permiten la múltiple herencia.
* Las interfaces proveen una forma de definir métodos que deben ser implementados por una clase, asegurando que ciertas funcionaliades esten presentes.

Ejemplo:

Definiendo la interfaz:

interface Drivable {

    fun drive()

}

Implementando la interfaz:

class Car : Drivable {

    override fun drive() {

        println("The car is driving.")

    }

}

Interfaz con una implementación default:

interface Drivable {

    fun drive() {

        println("Driving the vehicle.")

    }

}

class Bicycle : Drivable

val myBicycle = Bicycle()

myBicycle.drive()  *// Output: Driving the vehicle.*

## Repositories in Android

En Android, particularmente usando MVVM, un repositorio es una clase que actúa como un API para el acceso a datos para el resto de la aplicación.

Abstrae el origen de los datos, el cual puede venir de una fuente externa, el cache, o una base de datos.

Porque usarlo:

* Los repositorios permiten desacoplar las fuentes de datos del resto de la aplicación. El VM interactúa con un Repositorio y no necesita saber de dónde vienen los datos.
* Un repositorio puede manejar y coordinar dato de multiples fuentes, sirviendo como un API unificado.
* Pueden tener data en cache, permitiendo a las apps trabajar offline y brindando una mejor experiencia.

Ejemplo:

Definir el Repositorio:

class UserRepository(private val userDao: UserDao,

private val userService: UserService) {

    fun getUser(userId: String): LiveData<User> {

*// Logic to fetch data from network or local database*

    }

}

El repositorio es una clase que toma un UserDao (base de datos local) y un UserService(network service) como parámetros.

class UserViewModel(private val repository: UserRepository) : ViewModel() {

    fun getUser(userId: String): LiveData&lt;User&gt; {

        return repository.getUser(userId)

    }

}

UserViewModel interactúa con el repositorio para obtener data. No sabe de dónde viene dicha data.

Un API es un set de reglas y protocolos que permiten a una aplicación interactuar con otra. Definen los métodos y los formatos de datos que las aplicaciones pueden usar para comunicarse entre sí.

Las API se utilizan para integrar diferentes sistemas de software, permitiéndoles trabajar juntos, compartir datos y mejorar la funcionalidad.

En términos más simples, piense en una API como el menú de un restaurante. El menú ofrece una lista de platos que puedes pedir, junto con una descripción de cada plato. Cuando especificas qué plato quieres, la cocina (es decir, el sistema) prepara el plato y lo sirve.

En esta analogía, el menú es la API, el pedido es la solicitud y el plato que se sirve es la respuesta.

<https://tutorials.eu/the-foundations-of-mvvm-inheritance-and-interfaces-day-8-android-14-masterclass/>

# JSON, Retrofit, HTTP Requests and Restful APIs

## JSON

JavaScript Object Notation es un formato liviano para la transmisión uniforme de datos. Es fácil de leer y escribir para humanos y fácil de convertir y generar para las maquinas.

Algunas de las ventajas de usar JSON son:

* Simplicidad: al usar llaves para objectos y llaves cuadradas para representar listas.
* Universalidad: sin importar de donde vengan los datos, al usar JSON se asegura que cualquiera pueda entender el contenido sin usar un traductor.
* Compacidad: se eliminan detalles innecesarios.
* Jerarquía: permite representar objetos complejos al poder representar objetos anidados.

JSON se basa en una coleccion de pares de nombre/valor, que representan objetos.

## API (Application Programming Interface)

Un API es un set de reglas y definiciones que permiten a las aplicaciones comunicarse con otras. Definen metodos y formato de datos que las aplicaciones pueden usar para solicitar y cambiar informacion.

Hay varios tipos de APIs:

* **Web API**: Accesibles a traves de internet usando HTTP/HTTPS. Permiten a las aplicaciones interactuar con servicios y fuentes de datos externos, y comunmente usados en el desarrollo web para integrar servicios de terceros.
* **Library o Framework APIs**: Set de rutinas, protocolos y herramientas para construir software y aplicaciones.
* **OS APIs**: permiten interactuar con el sistema operativo, habilitando funcionalidades como leer archivos, interactuar con dispositivos de hardware y mas.

adfs

* **Endpoints**: Paths especificos or URLs a donde se pueden dirigir los requests.
* **Methods:** Metodos HTTP como GET, POST, PUT, DELETE, los cuales definen el tipo de operacion a llevar a cabo.
* **Requests:** La llamada al API, que consiste del endpoin, el metodo, headers y cualquier data enviada al API.
* **Responses:** Los datos devueltos por el API, usualmente en formato JSON o XML, junto con los codigos de status indicando si fue exitoso o si fallo.

### API Key

Un API key es un codigo que se utiliza para identificar al programa que llama al API, el developer o el usuario del website. Se usa para controlar el acceso.

El proposito de un API key es autenticar al usuario o la aplicacion que esta haciendo el request. Autorizar o controlar el acceso a ciertos endpoints. Monitorear el uso del API, limitando el numero de requests que se pueden hacer en un periodo especifico. Proveer seguimiento del uso del API, o informacion util para analisis.

## Gradle Scripts in Android Studio 14

Los scripts de Gradle manegan el proceso de construccion del app en Android Studio.

1. **Project-level build.gradle (top-level)**: Ubicado en la raíz del proyecto, define configuraciones comunes entre los modulos, como repositorios y classpath dependencies.
2. **Module-level build.gradle(app-level):** Ubicado en cada modulo. Especifica las configuraciones especificas del modulo, como el application ID, SDK version y dependencias.
3. **Gradle properties:** Configuraciones que se aplican globalmente entre tareas y proyectos, como settings de memoria.
4. **Gradle Tasks:** Acciones que Gradle ejecuta durante el proceso de build, como compilar y empaquetar el app.

## Dependencies in Android Studio

Las dependencias son librerias o modulos externos que el proyecto utiliza. Proveen funcionalidades varias y permten construir aplicaciones ricas en funcionalidad.

* Library Dependencies:
  + Librerias de Kotlin y Android: librerias pre construidas especificamente diseñadas para usarse con Kotlin y android como Kotlin Coroutines o Retrofit.
* SDK Dependencies:
  + Android Support Libraries: Como AndroidX o Jetpack components, ofrecen versioens compatibles de nuevas caracteristicas de Android y otras utilidades. Ejemplo: AndroidX RecyclerView
* Remote Dependencies:
  + Maven y Jcenter: Dependencias alojadas en repositorios remotos. Gradle se encarga de descargar e integrar estas librerias automaticamente.

### Agregar una Dependencia

Al crear un proyecto en Android, se crean un par de archivos llamados build.gradle.kts, uno es para el proyecto como tal, mientras el otro es para el modulo :app, es decir, por cada modulo que se cree en el proyecto, habrá uno de estos archivos.

En el build.gradle para el proyecto se pueden configurar los plugins que son comunes a todos los proyecto.

En el build.gradle para el modulo, se configuran las dependencias de cada modulo.

Ejemplo de otras dependencias que se pueden agregar:

//Compose ViewModel  
*implementation*("androidx.lifecycle:lifecycle-viewmodel-compose:2.7.0")  
  
//Network calls  
*implementation*("com.squareup.retrofit2:retrofit:2.9.0")  
  
//JSON to Kotlin objects  
*implementation*("com.squareup.retrofit2:converter-gson:2.9.0")  
  
//Image loading  
*implementation*("io.coil-kt:coil-compose:2.4.0")

Estas dependencias se usan en el proyecto MyRecipeApp, para hacer llamadas al API, trabajar con el ViewModel y facilitar la carga de imagenes.

## Retrofit

Retrofit es un cliente HTTP de tipo seguro para Android y Java, muy usado en aplicaciones Android.

* Retrofit simplifica el proceso de hacer requests HTTP a REST APIs, manejando la construccion de la URL, la session y el procesamiento de errores.
* Permite convertir automaticamente responses de JSON o XML en objetos Kotlin o Java.
* Usa Annotations para definir lod endpoints del API y lo HTTP methods.
* Tiene una buena integracion con Corutinas de Kotlin, permitiendo una facil implementacion de requests y responses asíncronos.

Ejemplo:

import retrofit2.Retrofit

import retrofit2.converter.gson.GsonConverterFactory

import retrofit2.http.GET

*// Define the API endpoints*

interface ApiService {

    @GET("users/1")

    suspend fun getUser(): User

}

*// Create a Retrofit instance*

val retrofit = Retrofit.Builder()

    .baseUrl("https://jsonplaceholder.typicode.com/")

    .addConverterFactory(GsonConverterFactory.create())

    .build()

*// Create an API service instance*

val apiService = retrofit.create(ApiService::class.java)

*// Make an API request*

val user = apiService.getUser()

En el previo ejemplo, se crea un ApiService que basicamente define los endpoints del API, es decir, las llamadas que se pueden hacer al API a traves de funciones **suspend**. Una funcion suspend es un tipo de funcion que se debe ejecutar en segundo plano, no en el hilo principal del app.

El annotation de la funcion suspend fun getUser indica que se debe hacer un request de tipo HTTP GET al endpoint “users/1”.

Se define tambien una instancia de retrofit a la que se le pasa la URL base y se usa el addConverterFactory para especificar que haga la conversion entre JSON y objectos Kotlin al recibir responses.

Luego se crea una instancia del ApiService, el cual se usa al final para obtener la informacion del usuario.

## Coroutines in Android Development

Las corutinas son una caracteristica de Kotlin que hacen las programacion asíncrona mas manejable y concisa.

Facilitan escribir codigo asíncrono de manera secuencial, haciendolo mas facil de entender. Algunos componentes y parte de la sintaxis de las corutinas son:

* **Suspend Functions**: Funciones marcadas con el keyword suspend pueden ser pausadas y reanudadas, permitiendo la ejecucion asíncrona que no bloquea la aplicación. Estas pueden ser invocadas desde otras funciones Suspend o dentro del scope de una corutinea.
* **Coroutine builder** – **launch:** inicia una nueva corutina sin bloquear el hilo actual y retorna una referencia de la corutina como un Job.
* **Coroutine builder - async**: Inicia una nueva corutina y retorna un Deferred, que es un futuro cancelable sin bloqueo que representa el resultado.
* **Scope**: el Scope controla la vida de la corutina.

Ejemplo:

suspend fun fetchData() {

*// Simulate a network call or any asynchronous operation*

    delay(1000)

    println("Data fetched successfully")

}

*// Calling the suspend function within a coroutine scope*

GlobalScope.launch {

    fetchData()

}

La funcion suspend fun fetchData() es una funcion suspend, puede ejecutar operaciones largas sin bloquear el hilo principal. El keyword suspend indica que puede ser pausada o reanudada.

La liena GlobalScope.launch {… } es donde se ejecuta la nueva corutina.

Otro ejemplo es:

suspend fun fetchData(): String {

    delay(1000)

    return "Data fetched successfully"

}

GlobalScope.launch {

    val result = fetchData()

    println(result)

}

Aqui la funcion suspend es similar a la anterior pero devuelve un String.

Ejemplo 3:

// In an Android ViewModel or any lifecycle-aware component

viewModelScope.launch {

    val result = fetchData()

    // Update UI with the fetched data

}

En este ejemplo se usa el scope de un ViewModel. Eso significa que la corutina puede ser cancelada cuando el viewmodel se ha limpiado, previniendo fugas de memoria.

Una vez que se llamao al fetchData, que es una funcion suspend, se actualiza el UI con dicha informacion.

## CircularProgressIndicator

Un componente que permite desplegar un circulo giratorio para mostrar que el app esta ocupada procesando algo. Ejemplo:

import androidx.compose.runtime.Composable

import androidx.compose.material.CircularProgressIndicator

import androidx.compose.material.Text

@Composable

fun LoadingScreen(isLoading: Boolean) {

    if (isLoading) {

**CircularProgressIndicator**()

        Text("Loading, please wait...")

    } else {

        Text("Content loaded!")

    }

}

## LazyVerticalGrid or LazyGridColumn

LazyVerticalGrid es un composable que permite desplegar multiples items en un grid de forma vertical y solo despliega los items que caben en pantalla. Ejemplo:

import androidx.compose.foundation.lazy.LazyVerticalGrid

import androidx.compose.foundation.lazy.items

import androidx.compose.runtime.Composable

import androidx.compose.ui.unit.dp

import androidx.compose.foundation.layout.GridCells

@Composable

fun SimpleGrid() {

    val items = List(100) { "Item $it" } *// Creating a list of 100 items*

    LazyVerticalGrid(

        cells = GridCells.Fixed(3), *// Setting the number of columns*

    ) {

        items(items) { item ->

            Text(item) *// Displaying each item in the grid*

        }

    }

}

Otro ejemplo utilizando imagenes:

@Composable

fun ImageGrid(images: List<Image>) {

    LazyVerticalGrid(

        cells = GridCells.Fixed(4), *// Setting the number of columns*

    ) {

        items(images) { image -&gt;

*// Displaying each image in the grid*

            Image(painter = image.painter, contentDescription = null)

        }

    }

}

Los LazyVerticalGrid son especialmente utiles para desplegar imagenes, productos o cualquier collecion similar.

## Displaying Remote Images

Para poder cargar imagenes desde internet usando Kotlin y Jetpack Compose se puede utilizar el compose Image junto con utilidades como el rememberAsyncImagePainter para manejar la carga y el despliegue de las imagenes.

Ejemplo 1:

import androidx.compose.runtime.Composable

import androidx.compose.ui.res.painterResource

import androidx.compose.ui.tooling.preview.Preview

import com.google.accompanist.imageloading.rememberAsyncImagePainter

@Composable

fun DisplayRemoteImage(imageUrl: String) {

**Image**(

        painter = **rememberAsyncImagePainter**(imageUrl),

        contentDescription = null *// Description for accessibility*

    )

}

El composable Image lleva varios parametros como el painter, que dibuja la imagen. La instrucción rememberAsynImagePainter se encarga de cargar la imagen desde el URL provisto. Maneja diferentes estados tambien, como loading, success o error.

En el siguiente ejemplo se usa una imagen de placeholder la cual se va a mostrar mientra la imagen remota se carga:

@Composable

fun DisplayRemoteImageWithPlaceholder(imageUrl: String) {

    val painter = rememberAsyncImagePainter(

        imageUrl,

        builder = {

            placeholder(painterResource(id = R.drawable.placeholder))

        }

    )

    Image(

        painter = painter,

        contentDescription = null *// Description for accessibility*

    )

}

## Android Manifest and Permissions

El Android Manifest en un archivo que todo proyecto de android tiene para definir informacion escencial sobre el app, como el nombre y los permisos que necesita.

Para agregar permisos, como el del acceso a internet, se usa un tag llamado uses-permission al mismo nivel que el de application, por ejemplo:

<?xml version="1.0" encoding="utf-8"?>

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:tools="http://schemas.android.com/tools">

<uses-permission android:name="android.permission.INTERNET" />

<application

android:allowBackup="true"

android:dataExtractionRules="@xml/data\_extraction\_rules"

android:fullBackupContent="@xml/backup\_rules"

android:icon="@mipmap/ic\_launcher"

android:label="@string/app\_name"

android:roundIcon="@mipmap/ic\_launcher\_round"

android:supportsRtl="true"

android:theme="@style/Theme.MyRecipeApp"

tools:targetApi="31">

<activity

android:name=".MainActivity"

android:exported="true"

android:label="@string/app\_name"

android:theme="@style/Theme.MyRecipeApp">

<intent-filter>

<action android:name="android.intent.action.MAIN" />

<category android:name="android.intent.category.LAUNCHER" />

</intent-filter>

</activity>

</application>

</manifest>

Hay muchos mas permisos que se pueden solicitar para el buen funcionamiento de un app dependiendo de lo que necesite hacer el app.

Referencia:

<https://tutorials.eu/navigating-libraries-apis-and-remote-content-day-9-android-14-masterclass/>

# Navigation in Android

## NavHost, NavController and Navigating

Para poder moverse entre “pantallas” en el app, Android ofrece el Navigation component, que provee soporte a Jetpack Compose. Para poder sacar provecho de este componente, lo primero que hay que hacer es agregar la dependencia al proyecto, por ejemplo:

dependencies {

val nav\_version = "2.7.7"

implementation("androidx.navigation:navigation-compose:$nav\_version")

}

Para implementar la navegacion, se necesita de tres cosas:

1. NavController: responsible de la navegación entre destinos (pantallas en el app). Expone métodos que permiten la navegación.
2. NavGraph: Mapea los composables para cada destino al que se puede navegar.
3. NavHost: un Composable que despliega otros composables destinos basados en una ruta. Es como un contendor para desplegar los destinos del NavGraph. Un beneficio de usar NavHost es que la logica de la navegación se mantiene separada del UI y todo en un mismo lugar.

### Implementing NavController and NavHost

El Navigation Controller es una parte fundamental, ya que este mantiene el navigation graph y expone metodos que permiten al app moverse entre los distintos destinos en el grafico.

El Navigation Controller hace uso de la clase NavController. Cuando se usa Jetpack Compose, se llama mediante el rememberNavController.

La documentacion para el Navigation Controller: <https://developer.android.com/guide/navigation/navcontroller>

Al definir el NavHost, se debe pasar la referencia del NavController y definir un destino de inicio. Dentro del NavHost, se definen las diferentes rutas para el app. Una ruta es un String que corresponde a un destino y funciona como su identificador único.

Se usa una funcion composable para definir la ruta pero también se mapea esta ruta a un composable específico o pantalla que se va a desplegar cuando se use la ruta.

Para navegar entre pantallas, se necesitan los métodos del navController. Lo que se hace es pasar una funcion como parametro a cada composable para especificar que es lo que va a pasar cuando el usuario le de click al botón para poder moverse a la siguiente pantalla. Por lo que cada composable va a recibir por parametro una función que trabaja como un button handler para cada botón que permita moverse entre pantallas.

Al llamar al navigate(), no solo se cambia de pantalla, tambien se posiciona dicha pantalla en el tope del back stack, por lo que cuando el usuario presiona el botón de back, se puede nevegar a la pantalla previa.

El siguiente es un ejemplo de un proyecto que usa varias pantallas para un app de pedidos de cupcakes.

/\*\*

\* Enum class to define the routes

\*/

enum class CupcakeScreen(){

Start,

Flavor,

Pickup,

Summary

}

/\*\*

\* Composable that displays the topBar and displays back button if back navigation is possible.

\*/

@Composable

fun CupcakeAppBar(

canNavigateBack: Boolean,

navigateUp: () -> Unit,

modifier: Modifier = Modifier

) {

TopAppBar(

title = { Text(stringResource(id = R.string.app\_name)) },

colors = TopAppBarDefaults.mediumTopAppBarColors(

containerColor = MaterialTheme.colorScheme.primaryContainer

),

modifier = modifier,

navigationIcon = {

if (canNavigateBack) {

IconButton(onClick = navigateUp) {

Icon(

imageVector = Icons.Filled.ArrowBack,

contentDescription = stringResource(R.string.back\_button)

)

}

}

}

)

}

@Composable

fun CupcakeApp(

viewModel: OrderViewModel = viewModel(),

navController: NavHostController = rememberNavController()

) {

Scaffold(

topBar = {

CupcakeAppBar(

canNavigateBack = false,

navigateUp = { /\* TODO: implement back navigation \*/ }

)

}

) { innerPadding ->

val uiState by viewModel.uiState.collectAsState()

// Define the NavHost and pass the navController and the start destination

NavHost(

navController = navController,

startDestination = CupcakeScreen.Start.name,

modifier = Modifier.padding(innerPadding)

){

// We call different composables to define the routes

// Define route for the StartOrderScreen

composable(route = CupcakeScreen.Start.name){

StartOrderScreen(quantityOptions = DataSource.quantityOptions,

onNextButtonClicked = {

// Update the view model so that the app displays the correct subtotal

viewModel.setQuantity(it)

// Navigate to the next screen

navController.navigate(CupcakeScreen.Flavor.name)

},

modifier = Modifier

.fillMaxSize()

.padding(dimensionResource(id = R.dimen.padding\_medium)))

}

// Define route for the Flavor Screen

composable(route = CupcakeScreen.Flavor.name){

val context = LocalContext.current

SelectOptionScreen(

subtotal = uiState.price,

onNextButtonClicked = { navController.navigate(CupcakeScreen.Pickup.name) },

onCancelButtonClicked = {

cancelOrderAndNavigateToStart(viewModel, navController)

},

options = DataSource.flavors.map { id -> context.resources.getString(id) },

onSelectionChanged = { viewModel.setFlavor(it) },

modifier = Modifier.fillMaxHeight()

)

}

composable(route = CupcakeScreen.Pickup.name){

SelectOptionScreen(

subtotal = uiState.price,

onNextButtonClicked = { navController.navigate(CupcakeScreen.Summary.name) },

onCancelButtonClicked = {

cancelOrderAndNavigateToStart(viewModel, navController)

},

options = uiState.pickupOptions,

onSelectionChanged = { viewModel.setDate(it) },

modifier = Modifier.fillMaxHeight()

)

}

composable(route = CupcakeScreen.Summary.name){

OrderSummaryScreen(

orderUiState = uiState,

onCancelButtonClicked = {

cancelOrderAndNavigateToStart(viewModel, navController)

},

onSendButtonClicked = { subject: String, summary: String ->

},

modifier = Modifier.fillMaxHeight())

}

}

}

}

/\*\*

\* Reset the app state in the view model and calls popBackStack to restart the flow again.

\*/

private fun cancelOrderAndNavigateToStart(

viewModel: OrderViewModel,

navController: NavHostController

) {

viewModel.resetOrder()

// Pop the controller's back stack back to the start screen

navController.popBackStack(CupcakeScreen.Start.name, inclusive = false)

}

En el proyecto se tiene tres composables que definen una pantalla cada uno (SelectOptionScreen, StartOrderScreen y SummaryScreen). Se utiliza un enum para definir los nombres de las pantallas, siendo Start la primera en visualizarse.

En el NavHost se definen los 4 destinos, cada uno llama a su respectivo composable.

En el caso del segundo y tercer destino se utiliza el mismo composable pero se le envía diferntes valores para las opciones a mostrar, asi como las funciones para un boton de Next y Cancel.

El botón Cancel llama a un metodo que basicamente reinicia el flujo y devuelve al usuario a la primer pantalla.