TECNOLOGICO NACIONAL DE MEXICO

INSTITUTO TECNOLOGICO DE CIUDAD MADERO

Carrera: Ingeniería en Sistemas Computacionales.

Materia: Programación Nativa Para Móviles

Alumnos:

Yañez Herrera Karina Aileen

Zavala Osorio Camilo Alexander

Números de control:

21070415

21070336

Profesor: Jorge Peralta Escobar

Hora: 2:00 pm – 3:00 pm

Semestre: Enero - Junio 2025.

Lección 1

Video 1:

Nos adentramos en conceptos más profundos de programación en Kotlin, comenzando con los condicionales, una herramienta fundamental para crear aplicaciones interactivas. Aprendimos que los condicionales permiten que una aplicación tome decisiones, dependiendo de ciertas condiciones, como si un teléfono está en modo "No molestar" o no. Usamos la estructura if para que la aplicación ejecute ciertas acciones solo si se cumple una condición.También exploramos cómo utilizar bloques else para definir lo que debe ocurrir cuando no se cumple una condición, y cómo manejar múltiples condiciones con estructuras if-else if-else. Además, descubrimos que en Kotlin existe una alternativa más limpia y legible: el when statement, que nos permite manejar varios casos diferentes de manera más compacta y ordenada.

Después, aprendimos sobre las expresiones condicionales, como if usado como expresión, lo cual nos permite asignar directamente valores a variables según las condiciones. Esto hace que el código sea más conciso y elegante, pero debemos tener cuidado de que todas las ramas devuelvan un valor, o el compilador nos marcará un error. En la segunda parte, vimos que en Kotlin, las funciones son de primera clase, lo que significa que se pueden asignar a variables, pasar como argumentos y retornar desde otras funciones, igual que si fueran datos. Aprendimos sobre las expresiones lambda, una forma corta y poderosa de declarar funciones sin necesidad de nombres. Esto nos será muy útil para manejar eventos como clics en botones y otras interacciones del usuario.

Todos estos conocimientos condicionales, expresiones y funciones como datos nos preparan para construir aplicaciones mucho más dinámicas e interactivas. En las próximas actividades, aplicaremos estos conceptos para cambiar el comportamiento de nuestras aplicaciones según lo que haga el usuario, como cuando presiona un botón o ingresa información. Esta unidad no solo mejora nuestra comprensión del lenguaje Kotlin, sino que también amplía nuestras habilidades para crear aplicaciones que respondan y se adapten a diferentes situaciones.

Actividades:

```
// Punto de entrada principal del programa
fun main() {

    // Variable que almacena el color actual del semáforo
    val trafficLightColor = "Black"
```

```
    // Condición si el color es rojo
    if (trafficLightColor == "Red") {
        println("Stop") // Imprime "Stop" si es rojo
    }
    // Condición si el color es amarillo
    else if (trafficLightColor == "Yellow") {
        println("Slow") // Imprime "Slow" si es amarillo
    }
    // Condición si el color es verde
    else if (trafficLightColor == "Green") {
        println("Go") // Imprime "Go" si es verde
    }
    // Si el color no coincide con ninguno de los anteriores
    else {
        println("Invalid traffic-light color") // Imprime mensaje de error
    }
}
```

Explicación: Este programa en Kotlin simula el comportamiento de un semáforo evaluando el color almacenado en la variable trafficLightColor. Según el valor de esta variable, imprime una instrucción específica: "Stop" si es rojo, "Slow" si es amarillo y "Go" si es verde. Si el valor no corresponde a ninguno de los colores esperados, como en este caso que es "Black", el programa muestra "Invalid traffic-light color". Este tipo de estructura condicional es útil para tomar decisiones basadas en diferentes casos posibles.

```
// Función principal del programa
fun main() {

    // Se define el color actual del semáforo
    val trafficLightColor = "Black"

    // Se evalúa el color del semáforo y se imprime el mensaje correspondiente
```

```
    // Pero en lugar de guardar el resultado como texto, se imprimen
directamente los mensajes.
    val message = if (trafficLightColor == "Red") {
        println("Stop")  // Se imprime "Stop" si el color es "Red"
    } else if (trafficLightColor == "Yellow") {
        println("Slow")  // Se imprime "Slow" si el color es "Yellow"
    } else if (trafficLightColor == "Green") {
        println("Go")    // Se imprime "Go" si el color es "Green"
    } else {
        println("Invalid traffic-light color")  // Se imprime si el color no es válido
    }

    // Aquí message contiene Unit (nada útil), ya que println() no devuelve un
valor útil
}
```

Explicación: Este programa intenta asignar a una variable message el resultado de una estructura if-else que imprime un mensaje basado en el color de un semáforo. Sin embargo, como se usan println() en lugar de retornar cadenas, lo que realmente se guarda en message es Unit (el equivalente a void en otros lenguajes), lo cual no tiene ningún uso en este contexto. A pesar de eso, el programa imprimirá "Invalid traffic-light color" porque el color "Black" no es válido.

```
// Función principal del programa
fun main() {

    // Se declara el color del semáforo
    val trafficLightColor = "Amber"

    // Se asigna un mensaje según el color, usando 'when'
    val message = when(trafficLightColor) {
        "Red" -> "Stop"                    // Si es "Red", asigna "Stop"
```

```
        "Yellow", "Amber" -> "Slow"        // Si es "Yellow" o "Amber", asigna "Slow"
        "Green" -> "Go"                    // Si es "Green", asigna "Go"
        else -> "Invalid traffic-light color" // Cualquier otro valor, mensaje inválido
    }

    // Se imprime el mensaje final
    println(message)
}
```

Explicación: Este programa evalúa el color de un semáforo utilizando la expresión when, que actúa como un switch avanzado. Según el valor de la variable trafficLightColor, se asigna un mensaje correspondiente a la variable message: si el color es "Red" se asigna "Stop", si es "Yellow" o "Amber" se asigna "Slow", y si es "Green" se asigna "Go". Si el color no coincide con ninguno de estos, se asigna "Invalid traffic-light color". Finalmente, se imprime el mensaje. En este caso, como el color es "Amber", se imprimirá "Slow".

Video 2:

En esta unidad, hemos aprendido cuatro conceptos fundamentales en Kotlin para desarrollar aplicaciones interactivas y funcionales. Comenzamos con los condicionales, que permiten tomar decisiones dentro de la app dependiendo de ciertas condiciones. Usamos IF, ELSE IF y ELSE para estructurar nuestras condiciones, y Kotlin también nos ofrece una manera más concisa de escribir condicionales con un solo when statement. Además, aprendimos a utilizar los condicionales como expresiones, lo que nos permite asignar valores a variables directamente desde la condición.

A continuación, aprendimos sobre programación orientada a objetos (OOP). Ahora sabemos cómo definir clases, agregarles propiedades y funciones miembros, extender clases existentes con métodos adicionales y sobrescribir funciones heredadas. También vimos cómo crear objetos a partir de estas clases y cómo controlar la instancia de estos, lo que nos permite organizar el código en componentes más modulares y reutilizables.

El siguiente concepto que exploramos fue la seguridad de nulos. Kotlin nos permite definir variables que no pueden ser nulas, lo que reduce el riesgo de errores al acceder a propiedades o métodos de estas variables. Sin embargo, también vimos que hay situaciones en las que necesitamos trabajar con variables nulas. En estos casos, Kotlin ofrece formas seguras de manejar variables nulas,

evitando que nuestra app se cierre inesperadamente por un intento de acceder a una propiedad de una variable nula.

Finalmente, aprendimos sobre funciones y cómo Kotlin trata las funciones como primeras clases. Esto significa que las funciones pueden ser asignadas a variables, pasadas como parámetros a otras funciones, y retornadas desde funciones. También vimos las expresiones lambda, que nos permiten definir funciones de manera más compacta y eficiente, sin necesidad de usar el keyword fun. Esto será especialmente útil cuando empecemos a trabajar con eventos y acciones de usuarios en aplicaciones Android.

Lección 2

Video 1:

En esta unidad, aprenderemos a crear una app que simula lanzar los dados. Sabemos que, generalmente, un dado tiene seis lados, y en cada uno de ellos hay un número. Cuando se lanza el dado, hay la misma probabilidad de que cualquiera de los lados quede hacia arriba, lo que nos da el número para esa ronda del juego. Lo que haremos es crear una aplicación donde se añada una imagen de dado y un botón. Al hacer clic en el botón, la app lanzará los dados (en realidad, generará un número aleatorio entre 1 y 6), y actualizará la imagen del dado para que coincida con el valor obtenido.

En el proceso, aprenderás a agregar un botón que responderá a los toques del usuario, y a activar la recomposición de la interfaz de usuario cuando se haga clic. Además, entenderás cómo comportarse adecuadamente en esta app y crearás un resultado diferente cada vez que el usuario haga clic en el botón. La próxima vez que tengas una noche de juegos, tus amigos podrán sacar su teléfono, abrir tu app de dados, y disfrutar del juego.

Actividades:

Dice roller:

```
// Paquete base del proyecto

package com.example.diceroller


// Importaciones necesarias para crear la interfaz y funcionalidad

import android.os.Bundle

import androidx.activity.ComponentActivity

import androidx.activity.compose.setContent
```

```
import androidx.compose.foundation.Image
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.wrapContentSize
import androidx.compose.material3.Button
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Surface
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.res.painterResource
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import com.example.diceroller.ui.theme.DiceRollerTheme

// Actividad principal de la aplicación
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
```

```
        // Establece el contenido de la UI usando Jetpack Compose
        setContent {
            DiceRollerTheme { // Aplica el tema de la app
                Surface(
                    modifier = Modifier.fillMaxSize(), // Ocupa toda la pantalla
                    color = MaterialTheme.colorScheme.background // Usa el color de fondo del tema
                ) {
                    DiceRollerApp() // Llama a la función principal de la app
                }
            }
        }
    }
}

// Función composable que representa la app. Se puede previsualizar en Android Studio.
@Preview
@Composable
fun DiceRollerApp() {
    // Llama al dado con botón e imagen centrado
    DiceWithButtonAndImage(
        modifier = Modifier
            .fillMaxSize()
            .wrapContentSize(Alignment.Center) // Centra el contenido en pantalla
    )
}

// Composable que representa el dado con el botón
@Composable
```

```
fun DiceWithButtonAndImage(modifier: Modifier = Modifier) {
    // Variable de estado que guarda el resultado del dado (inicializa en 1)
    var result by remember { mutableStateOf(1) }

    // Selecciona la imagen según el valor del dado
    val imageResource = when (result) {
        1 -> R.drawable.dice_1
        2 -> R.drawable.dice_2
        3 -> R.drawable.dice_3
        4 -> R.drawable.dice_4
        5 -> R.drawable.dice_5
        else -> R.drawable.dice_6
    }

    // Columna que contiene la imagen del dado y el botón
    Column(modifier = modifier, horizontalAlignment =
Alignment.CenterHorizontally) {
        // Muestra la imagen del dado
        Image(
            painter = painterResource(imageResource),
            contentDescription = result.toString()
        )

        // Botón que al hacer clic cambia aleatoriamente el valor del dado
        Button(
            onClick = { result = (1..6).random() }, // Genera un número aleatorio del
1 al 6
        ) {
            // Texto del botón (obtenido de recursos)
            Text(
```

```
            text = stringResource(R.string.roll), // Texto tipo "Roll" o "Lanzar"
            fontSize = 24.sp
        )
    }
  }
}
```

Explicación: Esta aplicación para Android está desarrollada en Kotlin utilizando Jetpack Compose, el framework moderno de UI declarativa de Android. Al iniciar, la MainActivity configura la interfaz mediante el método setContent, aplicando un tema visual definido por DiceRollerTheme. Dentro de la interfaz, se renderiza el componente DiceRollerApp, que a su vez utiliza el composable DiceWithButtonAndImage para mostrar el contenido principal de la pantalla. Este contenido consiste en una imagen de un dado y un botón centrados en la pantalla. La imagen del dado se selecciona dinámicamente en base a una variable de estado (result), la cual almacena el número actual del dado (del 1 al 6). Al presionar el botón, el valor de result se actualiza de forma aleatoria usando (1..6).random(), y como consecuencia, la imagen del dado también cambia para reflejar el nuevo número. El botón muestra un texto que proviene de los recursos localizados (R.string.roll), lo que permite que la app sea traducible a otros idiomas. Toda la lógica está contenida en componentes composables, lo que hace que la interfaz sea reactiva: cualquier cambio en el estado se refleja inmediatamente en la pantalla. En resumen, esta app simula de manera visual el lanzamiento de un dado al presionar un botón, mostrando la imagen correspondiente según el número obtenido.

Video 2:

En esta ruta, pudimos poner en práctica lo que aprendimos sobre Kotlin para desarrollar nuestra primera aplicación interactiva. Un aspecto importante fue cómo agregamos un botón que responde al clic del usuario usando una lambda en la propiedad onClick del botón. Cada vez que se hace clic, la función lambda se ejecuta, y eso hace que la aplicación sea más interactiva.

También aprendimos a modificar algunos atributos del botón, como el tamaño de la fuente, lo que hizo que nuestra aplicación se viera más atractiva y visualmente agradable. Además, usamos el componente remember para guardar el último lanzamiento de los dados y mantener ese valor incluso después de que la interfaz de usuario se recomponía. Esto es súper útil para mantener la coherencia en la aplicación y hacerla más dinámica. Otro punto clave fue que aprendimos a usar el depurador. Esto nos permitió detener la ejecución de la app en puntos específicos, inspeccionar las variables y entender mejor qué estaba sucediendo en nuestro código. Así, pudimos solucionar cualquier problema de manera más efectiva.

Lección 3

Video 1:

En este Pathway, aprendemos sobre el estado en Jetpack Compose y cómo gestionarlo en nuestras aplicaciones. Un estado es cualquier valor que puede cambiar con el tiempo, como el texto de un correo electrónico o una lista de destinatarios en una aplicación de correo. Los objetos componibles no recuerdan su estado de forma predeterminada, por lo que debemos hacer que recuerden ese estado o moverlo fuera del objeto componible.

Cuando Compose ejecuta un componible por primera vez, hace un seguimiento de los elementos que describen la interfaz de usuario. Si el estado de la aplicación cambia, Compose programa una recomposición, es decir, vuelve a ejecutar los elementos componibles para reflejar los cambios. Sin embargo, cualquier valor dentro de un componible se restablece a su valor inicial durante la recomposición, a menos que utilicemos el comando remember, que permite que Compose recuerde el valor durante las recomposiciones. Cuando el estado debe compartirse con otros componibles, se utiliza un estado mutable de función, que permite observar los cambios y activar la recomposición. Si el estado debe ser accesible en diferentes partes de la aplicación, se eleva el estado, moviéndolo fuera del componible y pasándolo como parámetro al componente padre, permitiendo que otros componibles compartan y utilicen ese estado.

Video 2:

En esta ruta, aprendemos a crear una aplicación que calcula el monto de la propina en función del costo del servicio. La propina es una cantidad adicional que un cliente da por un servicio, y su cálculo no siempre es sencillo. La aplicación se desarrollará en dos laboratorios de código. En el primer laboratorio, crearás una aplicación simple con un campo para ingresar el monto del servicio. La aplicación calculará automáticamente la propina basada en el porcentaje promedio de propina en los Estados Unidos.

En el segundo laboratorio, continuarás trabajando en la misma aplicación, pero agregarás funciones adicionales, como permitir que el usuario ingrese el porcentaje deseado para la propina o redondear el monto total. A lo largo de este proceso, aprenderás a aceptar la entrada del usuario mediante un campo de texto y un interruptor componible. Además, descubrirás cómo manejar el estado de la aplicación y cómo compartir datos entre los componibles, lo que te permitirá crear una experiencia interactiva para el usuario.

Actividades:

Tip

ActivityMain:

// Paquete de la aplicación package com.example.tiptime

// Importación de clases necesarias para Compose, diseño y control de estado import android.os.Bundle import androidx.activity.ComponentActivity import androidx.activity.compose.setContent import androidx.activity.enableEdgeToEdge import androidx.annotation.DrawableRes import androidx.annotation.StringRes import androidx.compose.foundation.layout.* // Importa todos los componentes de layout import androidx.compose.foundation.rememberScrollState import androidx.compose.foundation.text.KeyboardOptions import androidx.compose.foundation.verticalScroll import androidx.compose.material3.* // Material Design 3 import androidx.compose.runtime.* // Manejo del estado en Compose import androidx.compose.ui.* // Utilidades de la UI import androidx.compose.ui.res.painterResource import androidx.compose.ui.res.stringResource import androidx.compose.ui.text.input.ImeAction import androidx.compose.ui.text.input.KeyboardType import androidx.compose.ui.tooling.preview.Preview import androidx.compose.ui.unit.dp import com.example.tiptime.ui.theme.TipTimeTheme import java.text.NumberFormat // Para formato de moneda

// Actividad principal class MainActivity : ComponentActivity() { override fun onCreate(savedInstanceState: Bundle?) { enableEdgeToEdge() // Hace que el contenido se dibuje detrás de la barra de estado super.onCreate(savedInstanceState) setContent { // Se aplica el tema visual TipTimeTheme { // Superficie principal que ocupa toda la pantalla Surface(modifier = Modifier.fillMaxSize(),) { // Llama al diseño principal TipTimeLayout() } } } } }

// Función composable principal que contiene la interfaz de usuario @Composable fun TipTimeLayout() { // Variables de estado para los valores ingresados y la opción de redondeo var amountInput by remember { mutableStateOf("") } var tipInput by remember { mutableStateOf("") } var roundUp by remember { mutableStateOf(false) }

// Conversión de cadenas a números
val amount = amountInput.toDoubleOrNull() ?: 0.0
val tipPercent = tipInput.toDoubleOrNull() ?: 0.0
val tip = calculateTip(amount, tipPercent, roundUp) // Cálculo de la propina

// Estructura vertical de la interfaz

Column(
    modifier = Modifier
        .statusBarsPadding() // Deja espacio para la barra de estado

```
            .padding(horizontal = 40.dp) // Margen horizontal
            .verticalScroll(rememberScrollState()) // Habilita el scroll
            .safeDrawingPadding(), // Respeta zonas seguras de la pantalla
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ) {
        // Título de la pantalla
        Text(
            text = stringResource(R.string.calculate_tip),
            modifier = Modifier
                .padding(bottom = 16.dp, top = 40.dp)
                .align(alignment = Alignment.Start)
        )
        // Campo para ingresar el monto de la cuenta
        EditNumberField(
            label = R.string.bill_amount,
            leadingIcon = R.drawable.money,
            keyboardOptions = KeyboardOptions.Default.copy(
                keyboardType = KeyboardType.Number,
                imeAction = ImeAction.Next
            ),
            value = amountInput,
            onValueChanged = { amountInput = it },
            modifier = Modifier
                .padding(bottom = 32.dp)
                .fillMaxWidth(),
        )
        // Campo para ingresar el porcentaje de propina
        EditNumberField(
            label = R.string.how_was_the_service,
            leadingIcon = R.drawable.percent,
            keyboardOptions = KeyboardOptions.Default.copy(
                keyboardType = KeyboardType.Number,
                imeAction = ImeAction.Done
            ),
            value = tipInput,
            onValueChanged = { tipInput = it },
            modifier = Modifier
                .padding(bottom = 32.dp)
                .fillMaxWidth(),
        )
        // Switch para redondear la propina
        RoundTheTipRow(
            roundUp = roundUp,
            onRoundUpChanged = { roundUp = it },
```

```
        modifier = Modifier.padding(bottom = 32.dp)
    )
    // Texto que muestra la propina calculada
    Text(
        text = stringResource(R.string.tip_amount, tip),
        style = MaterialTheme.typography.displaySmall
    )
    // Espaciador final
    Spacer(modifier = Modifier.height(150.dp))
}


}
```

// Función que crea un campo de texto con ícono e interacción @Composable fun EditNumberField(@StringRes label: Int, @DrawableRes leadingIcon: Int, keyboardOptions: KeyboardOptions, value: String, onValueChanged: (String) -> Unit, modifier: Modifier = Modifier) { TextField(value = value, singleLine = true, leadingIcon = { Icon(painter = painterResource(id = leadingIcon), null) }, modifier = modifier, onValueChange = onValueChanged, label = { Text(stringResource(label)) }, keyboardOptions = keyboardOptions) }

// Fila que contiene el texto y el switch para redondear la propina @Composable fun RoundTheTipRow(roundUp: Boolean, onRoundUpChanged: (Boolean) -> Unit, modifier: Modifier = Modifier) { Row(modifier = modifier.fillMaxWidth(), verticalAlignment = Alignment.CenterVertically) { Text(text = stringResource(R.string.round_up_tip)) Switch(modifier = Modifier .fillMaxWidth() .wrapContentWidth(Alignment.End), // Alinea a la derecha checked = roundUp, onCheckedChange = onRoundUpChanged) } }

// Función que calcula la propina y la devuelve como string formateado en moneda local private fun calculateTip(amount: Double, tipPercent: Double = 15.0, roundUp: Boolean): String { var tip = tipPercent / 100 * amount if (roundUp) { tip = kotlin.math.ceil(tip) // Redondea hacia arriba si se seleccionó la opción } return NumberFormat.getCurrencyInstance().format(tip) // Formatea como "$xx.xx" }

// Vista previa para ver el diseño en Android Studio @Preview(showBackground = true) @Composable fun TipTimeLayoutPreview() { TipTimeTheme { TipTimeLayout() } }

Explicación: Este código implementa una aplicación Android en Kotlin usando Jetpack Compose que permite calcular la propina (tip) a partir del monto total de una cuenta (bill amount) y el porcentaje de propina deseado (tip percent). El usuario puede ingresar estos datos mediante campos de texto y activar una

opción para redondear la propina al número entero más cercano. La interfaz muestra el valor calculado de la propina en formato de moneda local. La estructura está compuesta por una actividad principal (MainActivity) que define el tema visual y llama al diseño principal (TipTimeLayout), el cual gestiona el estado de los inputs, renderiza los campos y muestra el resultado. Además, se incluye una función calculateTip que realiza el cálculo del monto de la propina y lo formatea correctamente.

Training Tip

MainActivity:

```
// Paquete principal de la app
package com.example.inventory


// Importaciones necesarias
import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.activity.enableEdgeToEdge
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.material3.Surface
import androidx.compose.ui.Modifier
import com.example.inventory.ui.theme.InventoryTheme // Tema personalizado de la app


// Actividad principal que representa la pantalla inicial
class MainActivity : ComponentActivity() {


    // Método que se ejecuta al crear la actividad
    override fun onCreate(savedInstanceState: Bundle?) {
        enableEdgeToEdge() // Habilita que la UI se dibuje detrás de la barra de estado/botones
        super.onCreate(savedInstanceState)
```

```
        // Establece el contenido visual de la app usando Jetpack Compose
        setContent {
            // Aplica el tema visual definido en InventoryTheme
            InventoryTheme {
                // Crea una superficie que ocupa toda la pantalla
                Surface(
                    modifier = Modifier.fillMaxSize(),
                ) {
                    // Llama al componente principal de la app (a definir por el
desarrollador)
                    InventoryApp()
                }
            }
        }
    }
}
```

Explicación: Este código define la actividad principal (MainActivity) de una aplicación Android llamada "Inventory", escrita en Kotlin usando Jetpack Compose. Su función es iniciar la interfaz de usuario cuando la app se ejecuta. Dentro del método onCreate, primero se habilita el dibujo en pantalla completa (enableEdgeToEdge()), luego se aplica el tema visual personalizado (InventoryTheme). Dentro de una superficie (Surface) que ocupa todo el tamaño de la pantalla, se llama a una función composable llamada InventoryApp(), que contendrá toda la lógica e interfaz de la aplicación (no incluida aquí). En resumen, este archivo configura la base visual sobre la que se construirá toda la app de inventario.

InventoryApp:

```
// Paquete principal de la aplicación package com.example.inventory

// Importa íconos y componentes de Material3 import
androidx.compose.material.icons.Icons.Filled import
```

androidx.compose.material.icons.filled.ArrowBack import
androidx.compose.material3.CenterAlignedTopAppBar import
androidx.compose.material3.ExperimentalMaterial3Api import
androidx.compose.material3.Icon import
androidx.compose.material3.IconButton import
androidx.compose.material3.Text import
androidx.compose.material3.TopAppBarScrollBehavior import
androidx.compose.runtime.Composable import androidx.compose.ui.Modifier
import androidx.compose.ui.res.stringResource import
androidx.navigation.NavHostController import
androidx.navigation.compose.rememberNavController import
com.example.inventory.R.string import
com.example.inventory.ui.navigation.InventoryNavHost

/**

- Composable de nivel superior que representa toda la estructura de pantallas
- de la aplicación usando un controlador de navegación.

*/ @Composable fun InventoryApp(navController: NavHostController = rememberNavController()) { // Llama al host de navegación que maneja las rutas/pantallas de la app InventoryNavHost(navController = navController) }

/**

- Barra superior de la app que muestra el título de la pantalla actual
- y, opcionalmente, un botón para regresar si se permite la navegación hacia atrás.

*/ @OptIn(ExperimentalMaterial3Api::class) // Marca que se usa una API experimental de Material3 @Composable

fun InventoryTopAppBar(

title: String, // Título a mostrar canNavigateBack: Boolean, // Indica si se debe mostrar la flecha de retroceso

modifier: Modifier = Modifier, // Modificador opcional

scrollBehavior: TopAppBarScrollBehavior? = null, // Comportamiento del scroll (opcional)

navigateUp: () -> Unit = {} // Acción al hacer clic en el botón de retroceso) { CenterAlignedTopAppBar(title = { Text(title) }, // Título centrado

modifier = modifier, scrollBehavior = scrollBehavior, navigationIcon = { // Si se permite navegar hacia atrás, muestra el ícono de flecha

if (canNavigateBack) {

IconButton(onClick = navigateUp) {

Icon(imageVector = Filled.ArrowBack, // Flecha hacia atrás

contentDescription = stringResource(string.back_button) // Descripción accesible) } } }) }

Explicación: Este archivo define la interfaz de navegación principal de una aplicación de inventario utilizando Jetpack Compose. Contiene dos funciones composables: InventoryApp, que establece el gráfico de navegación de la app a través de InventoryNavHost, y InventoryTopAppBar, que representa una barra superior centrada con un título. Esta barra superior también puede mostrar un botón de navegación hacia atrás (flecha) si es necesario, lo que mejora la experiencia del usuario al navegar entre pantallas. Se utiliza Material 3, lo que permite aplicar un estilo moderno con comportamiento adaptable al scroll. El código está diseñado para ser reutilizable y flexible en cualquier pantalla de la app.

InventoryAplication:

// Paquete principal de la app package com.example.inventory

// Importa la clase base Application de Android import android.app.Application

// Importa las interfaces y clases del contenedor de dependencias import com.example.inventory.data.AppContainer import com.example.inventory.data.AppDataContainer

// Clase personalizada de Application para la app de inventario class InventoryApplication : Application() {

```
/**
 * Instancia de AppContainer utilizada por el resto de clases
 * para obtener las dependencias necesarias (como repositorios, DAOs, etc.)
 */
lateinit var container: AppContainer

// Se llama cuando la app se inicia por primera vez
```

```
override fun onCreate() {
    super.onCreate()
    // Inicializa el contenedor con la implementación concreta
    container = AppDataContainer(this)
}


}
```

Explicación: Este archivo define la clase InventoryApplication, que extiende Application en Android y se utiliza para inicializar dependencias globales cuando se lanza la app. En este caso, se utiliza un patrón llamado Inyección de Dependencias Manual mediante un contenedor llamado AppContainer. Al iniciar la aplicación (onCreate()), se crea una instancia de AppDataContainer, que contiene las dependencias necesarias (por ejemplo, repositorios o bases de datos). Esto permite que otras partes de la app accedan fácilmente a esos recursos compartidos sin tener que crearlos varias veces, promoviendo un código más limpio, desacoplado y fácil de mantener.

Video 3:

En esta unidad, aprendimos conceptos clave de Kotlin y cómo crear aplicaciones interactivas que responden a clics de botones y aceptan entradas de campos de texto. A lo largo de la unidad, pudimos explorar cómo funciona el estado al crear una aplicación para calcular propinas. Aprendimos a responder a la entrada del usuario, observar cambios en variables utilizando mutableStateOf, y actualizar el estado de la aplicación para activar la recomposición.

También descubrimos cómo hacer que un objeto componible recuerde su estado durante la recomposición. Al continuar con el desarrollo de la aplicación de calculadora de propinas, aprendimos a compartir el estado entre diferentes elementos componibles mediante el proceso de elevar el estado. Esto nos permitió mover el valor y la función lambda a los parámetros de la función, facilitando el paso del estado entre los componibles.

Finalmente, aprendimos sobre la importancia de las pruebas automatizadas. Estas nos permiten verificar que el código funciona correctamente, detectar errores a tiempo y garantizar que los cambios no afecten el funcionamiento de la aplicación. La práctica de las pruebas es esencial, especialmente cuando las aplicaciones son pequeñas, y nos prepara para crear aplicaciones más complejas con menos errores.