



GOBIERNO DE  
MÉXICO

EDUCACIÓN  
SECRETARÍA DE EDUCACIÓN PÚBLICA



TECNOLÓGICO  
NACIONAL DE MÉXICO



# 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:**

En esta unidad, aprendimos conceptos esenciales para crear aplicaciones más completas y funcionales en Android. A diferencia de las apps básicas con las que empezamos, ahora nos enfocamos en cómo construir apps con más lógica, varias pantallas y una estructura sólida.

Primero, conocimos la arquitectura de una app, lo cual nos ayuda a organizar el código para que sea más fácil de mantener, probar y expandir, especialmente si trabajamos en equipo. Después, vimos cómo funciona la navegación entre pantallas, algo esencial cuando una app no se limita a una sola vista. Aprendimos a crear experiencias más fluidas para los usuarios al moverse dentro de la app.

También conocimos los diseños adaptativos, que permiten que nuestras apps funcionen bien en dispositivos con diferentes tamaños de pantalla, como celulares y tablets. Nos explicaron que, a diferencia de lo que asumimos al inicio del curso (como que el usuario siempre está en la app sin interrupciones), en la vida real el usuario puede rotar la pantalla, recibir llamadas o cambiar de app. Todo esto afecta cómo se comporta nuestra aplicación. Por eso, debemos tener en cuenta el ciclo de vida de una actividad, que define lo que pasa con una pantalla (Activity) desde que se crea hasta que se destruye.

Aprendimos que podemos usar métodos especiales, llamados callbacks del ciclo de vida, como `onCreate()`, para controlar qué sucede en nuestra app en cada etapa. También conocimos librerías que facilitan este manejo. Para practicar esto, trabajamos con una app llamada Dessert Clicker, donde vendemos postres haciendo clic en una imagen, lo cual nos ayudó a entender cómo mantener el estado de la app incluso si el usuario la deja y vuelve más tarde.

Luego, pasamos a un proyecto más avanzado: Unscramble, un juego de palabras desordenadas. Este juego nos enseñó cómo manejar estados más complejos, como qué palabras ya se usaron, cuántos puntos llevamos, y qué hacer si el usuario responde mal o quiere pasar la palabra.

Este tipo de apps requiere mejor organización, así que estudiamos cómo separar la app en capas: la capa de interfaz (UI) y la capa de datos, y cómo implementar un flujo unidireccional de datos para que todo sea más claro y mantenible. Finalmente, aprendimos la importancia de hacer pruebas al código, especialmente al de la lógica del juego, para asegurarnos de que todo funcione como se espera.

## **Actividades:**

### **Training dessert:**

#### **MainActivity.kt**

```
package com.example.dessertclicker

// Importaciones necesarias para Android, Jetpack Compose, y manejo de UI,
navegación, etc.

private const val TAG = "MainActivity" // Constante para usar en logs

// MainActivity que extiende ComponentActivity para usar Jetpack Compose
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        enableEdgeToEdge() // Habilita que la app use toda la pantalla, incluso
        detrás de barras del sistema
        super.onCreate(savedInstanceState)
        Log.d(TAG, "onCreate Called") // Log para seguimiento del ciclo de vida

        // Establece el contenido UI con Jetpack Compose
        setContent {
            DessertClickerTheme {
                // Surface es el contenedor que aplica el tema y fondo
                Surface(
                    modifier = Modifier
                        .fillMaxSize()
                        .statusBarsPadding(), // Padding para no tapar la barra de
                estado
                ) {
                    DessertClickerApp(desserts = Datasource.dessertList) // Llama la
                    función composable principal con la lista de postres
                }
            }
        }
    }

    // Logs para otros estados del ciclo de vida (inicio, pausa, destrucción, etc)
    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") }  
}  
  
/**  
 * Función que determina cuál postre mostrar basado en la cantidad vendida  
 * Recorre la lista de postres (ordenada por startProductionAmount) y  
 * devuelve el último postre cuyo startProductionAmount sea menor o igual que  
 dessertsSold  
 */  
fun determineDessertToShow(  
    desserts: List<Dessert>,  
    dessertsSold: Int  
) : Dessert {  
    var dessertToShow = desserts.first()  
    for (dessert in desserts) {  
        if (dessertsSold >= dessert.startProductionAmount) {  
            dessertToShow = dessert  
        } else {  
            break // Termina ciclo al encontrar el primer postre cuyo  
startProductionAmount sea mayor que dessertsSold  
        }  
    }  
    return dessertToShow  
}  
  
/**  
 * Función para compartir la información de postres vendidos y revenue usando  
un Intent ACTION_SEND  
 * Abre un chooser para seleccionar app de compartir, muestra toast si no hay  
app disponible  
 */  
private fun shareSoldDessertsInformation(intentContext: Context, dessertsSold:  
Int, revenue: Int) {  
    val sendIntent = Intent().apply {  
        action = Intent.ACTION_SEND  
        putExtra(  
            Intent.EXTRA_TEXT,  
            intentContext.getString(R.string.share_text, dessertsSold, revenue)  
        )  
        type = "text/plain"  
    }  
  
    val shareIntent = Intent.createChooser(sendIntent, null)
```

```
try {
    ContextCompat.startActivity(intentContext, shareIntent, null)
} catch (e: ActivityNotFoundException) {
    Toast.makeText(
        intentContext,
        intentContext.getString(R.string.sharing_not_available),
        Toast.LENGTH_LONG
    ).show()
}

/**
 * Composable principal que maneja el estado de la app (revenue,
dessertsSold, currentDessert)
 * Muestra UI con Scaffold, barra superior y contenido principal
 */
@Composable
private fun DessertClickerApp(
    desserts: List<Dessert>
) {
    // Estados guardados para mantener valores tras cambios de configuración
    var revenue by rememberSaveable { mutableStateOf(0) }
    var dessertsSold by rememberSaveable { mutableStateOf(0) }

    val currentDessertIndex by rememberSaveable { mutableStateOf(0) }
    var currentDessertPrice by rememberSaveable {
        mutableStateOf(desserts[currentDessertIndex].price)
    }
    var currentDessertImageId by rememberSaveable {
        mutableStateOf(desserts[currentDessertIndex].imageId)
    }

    Scaffold(
        topBar = {
            val intentContext = LocalContext.current
            val layoutDirection = LocalLayoutDirection.current
            DessertClickerAppBar(
                onShareButtonClicked = {
                    shareSoldDessertsInformation(
                        intentContext = intentContext,
                        dessertsSold = dessertsSold,
                        revenue = revenue
                    )
                }
            )
        }
    )
}
```

```
        },
        modifier = Modifier
            .fillMaxWidth()
            .padding(
                start = WindowInsets.safeDrawing.asPaddingValues()
                    .calculateStartPadding(layoutDirection),
                end = WindowInsets.safeDrawing.asPaddingValues()
                    .calculateEndPadding(layoutDirection),
            )
            .background(MaterialTheme.colorScheme.primary)
    )
}

) { contentPadding ->
    // Pantalla principal con imagen de postre y datos de ventas
    DessertClickerScreen(
        revenue = revenue,
        dessertsSold = dessertsSold,
        dessertImageId = currentDessertImageId,
        onDessertClicked = {
            // Al hacer click, aumenta ingresos y postres vendidos
            revenue += currentDessertPrice
            dessertsSold++

            // Cambia el postre mostrado según la cantidad vendida
            val dessertToShow = determineDessertToShow(desserts,
dessertsSold)
            currentDessertImageId = dessertToShow.imageId
            currentDessertPrice = dessertToShow.price
        },
        modifier = Modifier.padding(contentPadding)
    )
}
}

/***
 * Barra superior con título y botón para compartir
 */
@Composable
private fun DessertClickerAppBar(
    onShareButtonClicked: () -> Unit,
    modifier: Modifier = Modifier
) {
    Row(
        modifier = modifier,
```

```
    horizontalArrangement = Arrangement.SpaceBetween,
    verticalAlignment = Alignment.CenterVertically,
) {
    Text(
        text = stringResource(R.string.app_name),
        modifier = Modifier.padding(start =
dimensionResource(R.dimen.padding_medium)),
        color = MaterialTheme.colorScheme.onPrimary,
        style = MaterialTheme.typography.titleLarge,
    )
    IconButton(
        onClick = onShareButtonClicked,
        modifier = Modifier.padding(end =
dimensionResource(R.dimen.padding_medium)),
    ) {
        Icon(
            imageVector = Icons.Filled.Share,
            contentDescription = stringResource(R.string.share),
            tint = MaterialTheme.colorScheme.onPrimary
        )
    }
}
}

/**
 * Pantalla principal que muestra la imagen del postre y la información de
transacciones
 * Al hacer click en la imagen se ejecuta onDessertClicked
 */
@Composable
fun DessertClickerScreen(
    revenue: Int,
    dessertsSold: Int,
    @DrawableRes dessertImageId: Int,
    onDessertClicked: () -> Unit,
    modifier: Modifier = Modifier
) {
    Box(modifier = modifier) {
        // Imagen de fondo
        Image(
            painter = painterResource(R.drawable.bakery_back),
            contentDescription = null,
            contentScale = ContentScale.Crop
        )
    }
}
```

```
Column {
    Box(
        modifier = Modifier
            .weight(1f)
            .fillMaxWidth(),
    ) {
        // Imagen del postre clickable para aumentar ventas
        Image(
            painter = painterResource(dessertImageId),
            contentDescription = null,
            modifier = Modifier
                .width(dimensionResource(R.dimen.image_size))
                .height(dimensionResource(R.dimen.image_size))
                .align(Alignment.Center)
                .clickable { onDessertClicked() },
            contentScale = ContentScale.Crop,
        )
    }
    // Muestra datos de ventas e ingresos
    TransactionInfo(
        revenue = revenue,
        dessertsSold = dessertsSold,
        modifier =
        Modifier.background(MaterialTheme.colorScheme.secondaryContainer)
    )
}
}

/**
 * Muestra la info de transacciones: postres vendidos y revenue
 */
@Composable
private fun TransactionInfo(
    revenue: Int,
    dessertsSold: Int,
    modifier: Modifier = Modifier
) {
    Column(modifier = modifier) {
        DessertsSoldInfo(
            dessertsSold = dessertsSold,
            modifier = Modifier
                .fillMaxWidth()
                .padding(dimensionResource(R.dimen.padding_medium)))
    }
}
```

```
)  
RevenueInfo(  
    revenue = revenue,  
    modifier = Modifier  
        .fillMaxWidth()  
        .padding(dimensionResource(R.dimen.padding_medium))  
    )  
}  
}  
  
/**  
 * Composable que muestra el total de ingresos  
 */  
@Composable  
private fun RevenueInfo(revenue: Int, modifier: Modifier = Modifier) {  
    Row(  
        modifier = modifier,  
        horizontalArrangement = Arrangement.SpaceBetween,  
    ) {  
        Text(  
            text = stringResource(R.string.total_revenue),  
            style = MaterialTheme.typography.headlineMedium,  
            color = MaterialTheme.colorScheme.onSecondaryContainer  
        )  
        Text(  
            text = "$${revenue}",  
            textAlign = TextAlign.Right,  
            style = MaterialTheme.typography.headlineMedium,  
            color = MaterialTheme.colorScheme.onSecondaryContainer  
        )  
    }  
}  
  
/**  
 * Composable que muestra la cantidad total de postres vendidos  
 */  
@Composable  
private fun DessertsSoldInfo(dessertsSold: Int, modifier: Modifier = Modifier) {  
    Row(  
        modifier = modifier,  
        horizontalArrangement = Arrangement.SpaceBetween,  
    ) {  
        Text(  
            text = stringResource(R.string.dessert_sold),
```

```

        style = MaterialTheme.typography.titleLarge,
        color = MaterialTheme.colorScheme.onSecondaryContainer
    )
    Text(
        text = dessertsSold.toString(),
        style = MaterialTheme.typography.titleLarge,
        color = MaterialTheme.colorScheme.onSecondaryContainer
    )
}
}

/**
 * Preview para Android Studio que muestra la app con un postre de prueba
 */
@Preview
@Composable
fun MyDessertClickerAppPreview() {
    DessertClickerTheme {
        DessertClickerApp(listOf(Dessert(R.drawable.cupcake, 5, 0)))
    }
}

```

**Resumen:** MainActivity es la actividad principal que usa Jetpack Compose para mostrar una app sencilla donde el usuario "vende postres" al hacer click en la imagen. Se actualizan ingresos y postres vendidos, y la imagen cambia según el progreso. También incluye función para compartir las ventas.

### Dessert.kt (Modelo de datos)

```

package com.example.dessertclicker.model

/**
 * Data class que representa un postre con:
 * - imagId: recurso drawable para la imagen
 * - price: precio de venta del postre
 * - startProductionAmount: número de postres vendidos necesarios para
empezar a mostrar este postre
*/
data class Dessert(val imagId: Int, val price: Int, val startProductionAmount: Int)

```

**Resumen:** Clase simple que define la estructura de un postre con imagen, precio y cantidad mínima vendida para mostrarlo.

### **Datasource.kt (Fuente de datos)**

```
package com.example.dessertclicker.data

import com.example.dessertclicker.R
import com.example.dessertclicker.model.Dessert

/**
 * Objeto singleton que provee una lista fija de postres para la app
 */
object Datasource {
    val dessertList = listOf(
        Dessert(R.drawable.cupcake, 5, 0),
        Dessert(R.drawable.donut, 10, 5),
        Dessert(R.drawable.eclair, 15, 20),
        Dessert(R.drawable.froyo, 30, 50),
        Dessert(R.drawable.gingerbread, 50, 100),
        Dessert(R.drawable.honeycomb, 100, 200),
        Dessert(R.drawable.icecreamsandwich, 500, 500),
        Dessert(R.drawable.jellybean, 1000, 1000),
        Dessert(R.drawable.kitkat, 2000, 2000),
        Dessert(R.drawable.lollipop, 3000, 4000),
        Dessert(R.drawable.marshmallow, 4000, 8000),
        Dessert(R.drawable.nougat, 5000, 16000),
        Dessert(R.drawable.oreo, 6000, 20000)
    )
}
```

**Resumen:** Define una lista estática de postres con su imagen, precio y la cantidad mínima vendida para empezar a producirlos. Sirve como fuente de datos para la app.

### **Unscramble:**

#### **MainActivity.kt**

```
package com.example.unscramble

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.unscramble.ui.GameScreen
import com.example.unscramble.ui.theme.UnscrambleTheme

// MainActivity es la actividad principal de la app
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        // Habilita que la UI ocupe toda la pantalla, incluso detrás de barras de
        // estado o navegación
        enableEdgeToEdge()
        super.onCreate(savedInstanceState)

        // Define el contenido composable de la app con Compose
        setContent {
            // Aplica el tema de la app
            UnscrambleTheme {
                // Surface es un contenedor con fondo que ocupa toda la pantalla
                Surface(
                    modifier = Modifier.fillMaxSize(),
                ) {
                    // Muestra la pantalla principal del juego (composable)
                    GameScreen()
                }
            }
        }
    }
}

```

**Breve explicación:** Esta clase inicia la aplicación configurando la UI con Jetpack Compose y muestra la pantalla principal del juego (GameScreen) usando un tema personalizado.

### GameViewModel.kt

```

package com.example.unscramble.ui

import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.setValue
import androidx.lifecycle.ViewModel
import com.example.unscramble.data.MAX_NO_OF_WORDS
import com.example.unscramble.data.SCORE_INCREASE

```

```
import com.example.unscramble.data.allWords
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.asStateFlow
import kotlinx.coroutines.flow.update

/**
 * ViewModel que contiene la lógica del juego y mantiene el estado de la UI
 */
class GameViewModel : ViewModel() {

    // Estado interno de la UI (usando StateFlow para reactividad)
    private val _uiState = MutableStateFlow(GameUiState())
    val uiState: StateFlow<GameUiState> = _uiState.asStateFlow()

    // Guarda la palabra que el usuario está escribiendo
    var userGuess by mutableStateOf("")
        private set

    // Conjunto de palabras ya usadas para evitar repetir
    private var usedWords: MutableSet<String> = mutableSetOf()

    // La palabra original sin mezclar actualmente en juego
    private lateinit var currentWord: String

    init {
        resetGame() // Inicializa el juego al crear el ViewModel
    }

    /*
     * Reinicia el juego limpiando las palabras usadas y asignando la primera
     * palabra mezclada
     */
    fun resetGame() {
        usedWords.clear()
        _uiState.value = GameUiState(currentScrambledWord =
            pickRandomWordAndShuffle())
    }

    /*
     * Actualiza la palabra que el usuario escribe
     */
    fun updateUserGuess(guessedWord: String){
        userGuess = guessedWord
    }
}
```

```

}

/*
 * Verifica si la palabra escrita por el usuario es correcta
 * Si es correcta, incrementa el puntaje y actualiza el estado del juego
 * Si es incorrecta, marca error para la UI
*/
fun checkUserGuess() {
    if (userGuess.equals(currentWord, ignoreCase = true)) {
        val updatedScore = _uiState.value.score.plus(SCORE_INCREASE)
        updateGameState(updatedScore)
    } else {
        _uiState.update { currentState ->
            currentState.copy(isGuessedWordWrong = true)
        }
    }
    updateUserGuess("") // Reinicia la entrada del usuario después de
verificar
}

/*
 * Permite saltar la palabra actual y pasar a la siguiente sin incrementar
puntaje
*/
fun skipWord() {
    updateGameState(_uiState.value.score)
    updateUserGuess("")
}

/*
 * Actualiza el estado del juego después de adivinar o saltar:
 * - Si se llegó al máximo de palabras, marca el juego como terminado
 * - Si no, selecciona una nueva palabra mezclada y actualiza el contador y
puntaje
*/
private fun updateGameState(updatedScore: Int) {
    if (usedWords.size == MAX_NO_OF_WORDS){
        _uiState.update { currentState ->
            currentState.copy(
                isGuessedWordWrong = false,
                score = updatedScore,
                isGameOver = true
            )
        }
    }
}

```

```

        } else{
            _uiState.update { currentState ->
                currentState.copy(
                    isGuessedWordWrong = false,
                    currentScrambledWord = pickRandomWordAndShuffle(),
                    currentWordCount = currentState.currentWordCount.inc(),
                    score = updatedScore
                )
            }
        }
    }

/*
 * Mezcla aleatoriamente las letras de una palabra para crear el "scrambled
word"
*/
private fun shuffleCurrentWord(word: String): String {
    val tempWord = word.toCharArray()
    tempWord.shuffle()
    // Si la palabra mezclada es igual a la original, mezcla otra vez
    while (String(tempWord) == word) {
        tempWord.shuffle()
    }
    return String(tempWord)
}

/*
 * Selecciona una nueva palabra aleatoria que no se haya usado y la mezcla
*/
private fun pickRandomWordAndShuffle(): String {
    currentWord = allWords.random()
    return if (usedWords.contains(currentWord)) {
        pickRandomWordAndShuffle() // Recursión para evitar palabras
        repetidas
    } else {
        usedWords.add(currentWord)
        shuffleCurrentWord(currentWord)
    }
}
}

```

**Breve explicación:** ViewModel que maneja la lógica del juego de palabras mezcladas: controla el estado del juego, valida respuestas, maneja el puntaje, evita repetir palabras, y actualiza la UI mediante un StateFlow.

## GameUiState.kt

```
package com.example.unscramble.ui

/**
 * Clase de datos que representa el estado actual de la UI del juego.
 */
data class GameUiState(
    val currentScrambledWord: String = "", // Palabra actual mezclada para
    mostrar
    val currentWordCount: Int = 1,          // Número de palabra actual en el juego
    val score: Int = 0,                   // Puntaje acumulado
    val isGuessedWordWrong: Boolean = false, // Indica si la última respuesta
    fue incorrecta
    val isGameOver: Boolean = false        // Indica si el juego terminó
)
```

**Breve explicación:** Clase que contiene los datos que la UI observa para mostrar el estado actual del juego: palabra mostrada, puntaje, número de palabra, si la respuesta fue correcta o no, y si el juego terminó.

## Código GameScreen

```
package com.example.unscramble.ui

import android.app.Activity
import androidx.compose.foundation.background
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.safeDrawingPadding
import androidx.compose.foundation.layout.statusBarsPadding
import androidx.compose.foundation.layout.wrapContentHeight
import androidx.compose.foundation.rememberScrollState
import androidx.compose.foundation.text.KeyboardActions
import androidx.compose.foundation.text.KeyboardOptions
import androidx.compose.foundation.verticalScroll
import androidx.compose.material3.AlertDialog
import androidx.compose.material3.Button
import androidx.compose.material3.Card
import androidx.compose.material3.CardDefaults
import androidx.compose.material3.MaterialTheme.colorScheme
import androidx.compose.material3.MaterialTheme.shapes
```

```
import androidx.compose.material3.MaterialTheme.typography
import androidx.compose.material3.OutlinedButton
import androidx.compose.material3.OutlinedTextField
import androidx.compose.material3.Text
import androidx.compose.material3.TextButton
import androidx.compose.material3.TextFieldDefaults
import androidx.compose.runtime.Composable
import androidx.compose.runtime.collectAsState
import androidx.compose.runtime.getValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.res.dimensionResource
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.text.input.ImeAction
import androidx.compose.ui.text.style.TextAlign
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import androidx.lifecycle.viewmodel.compose.viewModel
import com.example.unscramble.R
import com.example.unscramble.ui.theme.UnscrambleTheme

// Composable principal que muestra la pantalla del juego
@Composable
fun GameScreen(gameViewModel: GameViewModel = viewModel()) {
    // Obtiene el estado actual del juego desde el ViewModel (Flow)
    val gameUiState by gameViewModel.uiState.collectAsState()

    // Obtiene una dimensión de padding definida en recursos
    val mediumPadding = dimensionResource(R.dimen.padding_medium)

    // Layout principal en columna con scroll vertical y padding para la barra de
    // estado y safe area
    Column(
        modifier = Modifier
            .statusBarsPadding() // Evita superposición con barra de estado
            .verticalScroll(rememberScrollState()) // Habilita scroll vertical
            .safeDrawingPadding() // Padding para safe area (ej. notch)
            .padding(mediumPadding), // Padding general
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
```

```
// Título de la app
Text(
    text = stringResource(R.string.app_name),
    style = typography.titleLarge,
)

// Composable que muestra el área principal del juego (palabra
desordenada, input, etc)
GameLayout(
    onUserGuessChanged = { gameViewModel.updateUserGuess(it) }, // Callback para actualizar la palabra escrita
    wordCount = gameUiState.currentWordCount, // Número de palabra actual
    userGuess = gameViewModel.userGuess, // Texto que escribe el usuario
    onKeyboardDone = { gameViewModel.checkUserGuess() }, // Acción cuando se envía el texto (tecla done)
    currentScrambledWord = gameUiState.currentScrambledWord, // Palabra desordenada a adivinar
    isGuessWrong = gameUiState.isGuessedWordWrong, // Si la respuesta anterior fue incorrecta
    modifier = Modifier
        .fillMaxWidth()
        .wrapContentHeight()
        .padding(mediumPadding)
)

// Contenedor para los botones (Enviar y Saltar)
Column(
    modifier = Modifier
        .fillMaxWidth()
        .padding(mediumPadding),
    verticalArrangement = Arrangement.spacedBy(mediumPadding), // Espacio entre botones
    horizontalAlignment = Alignment.CenterHorizontally
) {
    // Botón para enviar la palabra adivinada
    Button(
        modifier = Modifier.fillMaxWidth(),
        onClick = { gameViewModel.checkUserGuess() }
    ) {
        Text(
            text = stringResource(R.string.submit),
            fontSize = 16.sp
        )
    }
}
```

```
        )
    }

// Botón para saltar la palabra actual y pasar a la siguiente
OutlinedButton(
    onClick = { gameViewModel.skipWord() },
    modifier = Modifier.fillMaxWidth()
) {
    Text(
        text = stringResource(R.string.skip),
        fontSize = 16.sp
    )
}
}

// Muestra la puntuación actual dentro de una tarjeta con padding
GameStatus(score = gameUiState.score, modifier =
Modifier.padding(20.dp))

// Si el juego terminó, muestra un diálogo con la puntuación final y
opciones para salir o jugar de nuevo
if (gameUiState.isGameOver) {
    FinalScoreDialog(
        score = gameUiState.score,
        onPlayAgain = { gameViewModel.resetGame() }
    )
}
}

// Composable que muestra la puntuación actual en una Card
@Composable
fun GameStatus(score: Int, modifier: Modifier = Modifier) {
    Card(
        modifier = modifier
    ) {
        Text(
            text = stringResource(R.string.score, score),
            style = typography.headlineMedium,
            modifier = Modifier.padding(8.dp)
        )
    }
}
```

```
// Composable que contiene el diseño de la parte principal del juego:  
// Muestra el conteo de palabras, la palabra desordenada, instrucciones y  
campo de texto para adivinar  
@Composable  
fun GameLayout(  
    currentScrambledWord: String,  
    wordCount: Int,  
    isGuessWrong: Boolean,  
    userGuess: String,  
    onUserGuessChanged: (String) -> Unit,  
    onKeyboardDone: () -> Unit,  
    modifier: Modifier = Modifier  
) {  
    val mediumPadding = dimensionResource(R.dimen.padding_medium)  
  
    Card(  
        modifier = modifier,  
        elevation = CardDefaults.cardElevation(defaultElevation = 5.dp)  
    ) {  
        Column(  
            verticalArrangement = Arrangement.spacedBy(mediumPadding), //  
Espacio entre elementos  
            horizontalAlignment = Alignment.CenterHorizontally,  
            modifier = Modifier.padding(mediumPadding)  
        ) {  
            // Muestra el número actual de palabra dentro de un fondo con color  
            Text(  
                modifier = Modifier  
                    .clip(shapes.medium)  
                    .background(colorScheme.surfaceTint)  
                    .padding(horizontal = 10.dp, vertical = 4.dp)  
                    .align(alignment = Alignment.End),  
                text = stringResource(R.string.word_count, wordCount),  
                style = typography.titleMedium,  
                color = colorScheme.onPrimary  
            )  
  
            // Muestra la palabra desordenada grande  
            Text(  
                text = currentScrambledWord,  
                style = typography.displayMedium  
            )  
  
            // Muestra instrucciones para el usuario
```

```

Text(
    text = stringResource(R.string.instructions),
    textAlign = TextAlign.Center,
    style = typography.titleMedium
)

// Campo de texto donde el usuario escribe su intento de palabra
OutlinedTextField(
    value = userGuess,
    singleLine = true,
    shape = shapes.large,
    modifier = Modifier.fillMaxWidth(),
    colors = TextFieldDefaults.colors(
        focusedContainerColor = colorScheme.surface,
        unfocusedContainerColor = colorScheme.surface,
        disabledContainerColor = colorScheme.surface,
    ),
    onValueChange = onUserGuessChanged, // Actualiza texto escrito
    label = {
        if (isGuessWrong) {
            Text(stringResource(R.string.wrong_guess)) // Muestra error si
        la respuesta es incorrecta
        } else {
            Text(stringResource(R.string.enter_your_word)) // Etiqueta
        normal
        }
    },
    isError = isGuessWrong,
    keyboardOptions = KeyboardOptions.Default.copy(
        imeAction = ImeAction.Done // Acción "Done" en teclado
    ),
    keyboardActions = KeyboardActions(
        onDone = { onKeyboardDone() } // Al presionar Done ejecuta
    validación
    )
)
}
}

/*
 * Diálogo que muestra la puntuación final al terminar el juego,
 * con botones para salir o jugar otra vez.
 */

```

```
@Composable
private fun FinalScoreDialog(
    score: Int,
    onPlayAgain: () -> Unit,
    modifier: Modifier = Modifier
) {
    val activity = (LocalContext.current as Activity) // Obtiene la actividad actual

    AlertDialog(
        onDismissRequest = {
            // Aquí podría manejarse la acción de dismiss, si se desea
        },
        title = { Text(text = stringResource(R.string.congratulations)) },
        text = { Text(text = stringResource(R.string.you_scored, score)) },
        modifier = modifier,
        dismissButton = {
            TextButton(
                onClick = {
                    activity.finish() // Cierra la app al salir
                }
            ) {
                Text(text = stringResource(R.string.exit))
            }
        },
        confirmButton = {
            TextButton(onClick = onPlayAgain) {
                Text(text = stringResource(R.string.play_again))
            }
        }
    )
}

// Vista previa para desarrollo en Android Studio, que muestra la pantalla del
// juego
@Preview(showBackground = true)
@Composable
fun GameScreenPreview() {
    UnscrambleTheme {
        GameScreen()
    }
}
```

**Resumen:** Este archivo GameScreen.kt contiene la interfaz gráfica del juego "Unscramble" hecho con Jetpack Compose. Es el UI que el usuario ve y con el que interactúa para jugar.

**GameScreen:** Función principal que arma la pantalla completa, con título, área de juego, botones para enviar o saltar palabra, estado de puntuación y un diálogo para mostrar resultado final.

**GameLayout:** Muestra el área central con la palabra desordenada, un campo de texto para ingresar el intento de palabra y muestra mensajes de error si la respuesta es incorrecta.

**GameStatus:** Muestra la puntuación actual del jugador.

**FinalScoreDialog:** Diálogo que aparece al terminar el juego, mostrando la puntuación final y ofreciendo opciones para salir o volver a jugar.

El flujo de datos ocurre desde un ViewModel llamado GameViewModel, que expone el estado actual del juego a la UI mediante StateFlow. La UI actualiza y valida las palabras ingresadas y maneja la navegación y la lógica básica del juego.

## **Video 2:**

En este video aprendimos por qué es tan importante entender el ciclo de vida de una actividad (Activity) cuando desarrollamos apps en Android. Hasta ahora, creímos que una vez que se abría la app, todo funcionaba sin interrupciones. Pero aprendimos que eso no es lo que pasa realmente en los dispositivos móviles.

Las apps pueden cerrarse o pausarse en cualquier momento: cuando giramos el celular, cambiamos de app, nos entra una llamada o simplemente bloqueamos el teléfono. Esto nos enseñó que el sistema Android puede detener o incluso destruir nuestra actividad en cualquier momento, y cuando eso pasa, nosotros como desarrolladores debemos saber cómo manejarlo.

El video nos explicó que una actividad es como la pantalla principal de nuestra app, donde se muestra la interfaz y donde el usuario interactúa. Pero esta actividad pasa por varias etapas: se crea, se vuelve visible, se ejecuta, se pausa, se detiene, y se destruye. Todo eso forma parte del ciclo de vida de una actividad. También aprendimos que hay métodos llamados callbacks, como `onCreate()`, `onStart()`, `onResume()`, `onPause()`, `onStop()` y `onDestroy()`, que podemos usar para controlar lo que pasa en cada una de esas etapas. Por ejemplo, en `onCreate()` se crea la interfaz, y en `onPause()` podemos guardar datos para no perderlos si el usuario se va de la app.

Una parte muy importante fue entender cómo guardar el estado de la app. Por ejemplo, si el usuario estaba jugando, ¿cómo hacemos para que, al volver a la

app, no tenga que empezar desde cero? Para eso, aprendimos a usar métodos como `onSaveInstanceState()` y `onRestoreInstanceState()`, que nos permiten guardar información temporal, como la puntuación de un juego o el contenido de un formulario.

El video también nos dio ejemplos concretos de situaciones reales, como rotar el celular, cerrar la app desde el fondo, o simplemente cambiar de pantalla. En todas esas situaciones, debemos asegurarnos de que la app mantenga el estado del usuario, para que no se pierda información y la experiencia sea fluida.

### **Video 3:**

En este video aprendimos por qué es tan importante guardar el estado de la interfaz de usuario (UI) en las aplicaciones Android. Como estudiantes, ya sabíamos que al girar el celular o cambiar de aplicación, Android puede destruir y volver a crear una actividad, pero ahora comprendimos cómo eso afecta directamente a lo que el usuario ve y hace dentro de la app. Por ejemplo, si alguien está llenando un formulario o jugando y gira el dispositivo, toda esa información puede perderse si no se ha guardado correctamente, lo que da como resultado una mala experiencia.

Para resolver esto, vimos tres formas principales de preservar el estado de la UI. Primero, aprendimos que usar variables normales dentro de la actividad no es suficiente, ya que si la actividad se destruye (por ejemplo, al rotar la pantalla), esas variables se reinician y se pierde la información. Luego, conocimos el método `onSaveInstanceState()` que permite guardar datos simples en un objeto `Bundle`, como texto o números.

Esta técnica es útil para datos temporales o estados básicos que deben sobrevivir durante una sesión activa del usuario. Finalmente, aprendimos a usar `ViewModel`, una herramienta más robusta para mantener el estado de la UI incluso si la actividad se destruye y se vuelve a crear. El `ViewModel` conserva los datos mientras la actividad o fragmento esté en uso y no se reinicia con cambios como la rotación, lo que lo convierte en la opción ideal cuando trabajamos con lógica más compleja. Además, el uso de `ViewModel` ayuda a separar la lógica de la interfaz de la lógica de los datos, lo cual hace que el código sea más limpio, organizado y fácil de mantener.

En resumen, entendimos que si queremos ofrecer una experiencia de usuario fluida y confiable, debemos aprender a guardar correctamente el estado de la UI, ya sea usando `onSaveInstanceState()` para casos simples o `ViewModel` para apps más completas y dinámicas.

#### **Video 4:**

En este video aprendimos qué es un ViewModel y por qué es tan importante al desarrollar aplicaciones Android. Como estudiantes, ya habíamos visto que al rotar el dispositivo o cambiar de actividad, Android puede destruir y volver a crear la UI, lo cual hace que los datos se pierdan si solo usamos variables normales. Aquí entendemos que el ViewModel es una clase que nos permite guardar y administrar datos relacionados con la interfaz de usuario de forma que sobrevivan a cambios de configuración como la rotación de pantalla. A diferencia de las variables normales, los datos almacenados en un ViewModel no se pierden cuando la actividad se vuelve a crear, lo que garantiza una experiencia más estable para el usuario.

Además, el ViewModel nos ayuda a separar la lógica de negocio de la lógica de UI, lo que hace que el código sea más limpio y fácil de mantener. En el ejemplo del video, vimos cómo se crea una clase que hereda de ViewModel y cómo se usa dentro de una actividad mediante el viewModel() delegado. También vimos cómo se pueden almacenar datos dentro del ViewModel, como contadores o información del juego, y cómo esa información sigue disponible incluso después de cambios de configuración. Este enfoque es especialmente útil en apps que requieren manejar estados complejos, como juegos, formularios o aplicaciones que muestran datos dinámicos.

En resumen, el ViewModel es una herramienta esencial para desarrollar apps robustas y bien estructuradas, y usarlo nos permite crear experiencias más consistentes para los usuarios y mantener un código más organizado.

## **Lección 2**

#### **Video 1:**

En este video aprendimos cómo funciona el ciclo de vida de una actividad en Android, algo que al principio nos parecía un poco abstracto, pero que ahora entendemos que es fundamental para crear apps estables y eficientes. Descubrimos que una actividad representa una pantalla con la que el usuario interactúa, y que no siempre está activa o visible, ya que Android la va cambiando de estado según lo que el usuario haga o según las necesidades del sistema. Por ejemplo, si abrimos otra app, recibimos una llamada o simplemente rotamos el dispositivo, la actividad puede pausarse, detenerse o incluso destruirse.

Por eso, Android nos ofrece una serie de métodos del ciclo de vida, como onCreate(), onStart(), onResume(), onPause(), onStop() y onDestroy(), que se ejecutan automáticamente cuando la actividad entra en esos estados. Cada uno tiene un propósito específico, como inicializar la UI, liberar recursos o guardar datos antes de que algo cambie. Vimos también que es muy importante usar correctamente estos métodos para asegurarnos de que la app no consuma recursos innecesarios ni pierda datos importantes.

Algo que nos pareció muy útil es que Android nos permite personalizar el comportamiento de la app en cada etapa del ciclo de vida, por ejemplo, pausando un video cuando el usuario cambia de app o guardando información antes de que se destruya la actividad. En resumen, entender el ciclo de vida de una actividad nos ayuda a crear apps más seguras, eficientes y con una mejor experiencia para el usuario.

Actividades:

Cupcake:

MainActivity.kt

```
package com.example.cupcake
```

```
import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.activity.enableEdgeToEdge
import com.example.cupcake.ui.theme.CupcakeTheme

// Clase principal que lanza la aplicación
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        // Permite que el contenido ocupe todo el espacio (incluyendo zonas como
        // el status bar)
        enableEdgeToEdge()
        super.onCreate(savedInstanceState)
        // Establece el contenido de la pantalla
        setContent {
            // Aplica el tema de la app
            CupcakeTheme {
                // Llama a la función composable principal de la app
                CupcakeApp()
            }
        }
    }
}
```

Explicación:

MainActivity es el punto de entrada de la app. Usa setContent para definir la UI con Jetpack Compose, aplicando el tema visual y mostrando CupcakeApp().

## DataSource.kt

```
package com.example.cupcake.data

import com.example.cupcake.R

// Objeto que contiene los datos que se usan para construir la UI
object DataSource {

    // Lista de sabores disponibles, representados como IDs de recursos de
    strings
    val flavors = listOf(
        R.string.vanilla,
        R.string.chocolate,
        R.string.red_velvet,
        R.string.salted_caramel,
        R.string.coffee
    )

    // Lista de cantidades disponibles con su ID de string y valor numérico
    val quantityOptions = listOf(
        Pair(R.string.one_cupcake, 1),
        Pair(R.string.six_cupcakes, 6),
        Pair(R.string.twelve_cupcakes, 12)
    )
}
```

Explicación:

DataSource es un objeto singleton que contiene los datos estáticos: sabores de pastelillos y cantidades posibles para ordenar, usando recursos localizables (R.string).

## ◊ OrderUiState.kt

```
package com.example.cupcake.data

// Clase que representa el estado actual de la orden en la interfaz
data class OrderUiState(
    val quantity: Int = 0,           // Cantidad seleccionada
    val flavor: String = "",        // Sabor seleccionado
    val date: String = "",          // Fecha seleccionada
    val price: String = "",         // Precio total
    val pickupOptions: List<String> = listOf() // Fechas disponibles para recoger
```

)

Explicación:

OrderUiState es una clase de datos que guarda toda la información que el usuario selecciona durante el proceso de pedido.

CupcakeScreen.kt (principal lógica y navegación)

1. Enum para pantallas

```
enum class CupcakeScreen(@StringRes val title: Int) {
    Start(title = R.string.app_name),           // Pantalla inicial
    Flavor(title = R.string.choose_flavor),      // Selección de sabor
    Pickup(title = R.string.choose_pickup_date), // Selección de fecha
    Summary(title = R.string.order_summary)       // Resumen del pedido
}
```

2. Barra superior @Composable

```
fun CupcakeAppBar(
    currentScreen: CupcakeScreen,
    canNavigateBack: Boolean,
    navigateUp: () -> Unit,
    modifier: Modifier = Modifier
) {
    TopAppBar(
        title = { Text(stringResource(currentScreen.title)) }, // Título dinámico
        colors = TopAppBarDefaults.mediumTopAppBarColors(
            containerColor = MaterialTheme.colorScheme.primaryContainer
        ),
        modifier = modifier,
        navigationIcon = {
            // Botón de retroceso si es posible volver
            if (canNavigateBack) {
                IconButton(onClick = navigateUp) {
                    Icon(
                        imageVector = Icons.Filled.ArrowBack,
                        contentDescription = stringResource(R.string.back_button)
                    )
                }
            }
        }
)
```

```
}
```

### 3. Función principal CupcakeApp()

```
@Composable
fun CupcakeApp(
    viewModel: OrderViewModel = viewModel(), // ViewModel que contiene el
    estado
    navController: NavHostController = rememberNavController() // Controlador
    de navegación
) {
    val backStackEntry by navController.currentBackStackEntryAsState() // 
    Pantalla actual
    val currentScreen = CupcakeScreen.valueOf(
        backStackEntry?.destination?.route ?: CupcakeScreen.Start.name
    )

    Scaffold(
        topBar = {
            CupcakeAppBar(
                currentScreen = currentScreen,
                canNavigateBack = navController.previousBackStackEntry != null,
                navigateUp = { navController.navigateUp() }
            )
        }
    ) { innerPadding ->
        val uiState by viewModel.uiState.collectAsState() // Observar estado

        NavHost(
            navController = navController,
            startDestination = CupcakeScreen.Start.name,
            modifier = Modifier
                .fillMaxSize()
                .verticalScroll(rememberScrollState())
                .padding(innerPadding)
        ) {
            // Pantalla de inicio
            composable(route = CupcakeScreen.Start.name) {
                StartOrderScreen(
                    quantityOptions = DataSource.quantityOptions,
                    onNextButtonClicked = {
                        viewModel.setQuantity(it)
                        navController.navigate(CupcakeScreen.Flavor.name)
                    }
                )
            }
        }
    }
}
```

```
        },
        modifier = Modifier
            .fillMaxSize()
            .padding(dimensionResource(R.dimen.padding_medium))
    )
}

// Pantalla de selección de sabor
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()
    )
}

// Pantalla de selección de fecha
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()
    )
}

// Pantalla resumen
composable(route = CupcakeScreen.Summary.name) {
    val context = LocalContext.current
    OrderSummaryScreen(
        orderUiState = uiState,
```

```

        onCancelButtonClicked = {
            cancelOrderAndNavigateToStart(viewModel, navController)
        },
        onSendButtonClicked = { subject: String, summary: String ->
            shareOrder(context, subject = subject, summary = summary)
        },
        modifier = Modifier.fillMaxHeight()
    )
}
}
}
}

```

#### 4. Función para cancelar y volver al inicio

```

private fun cancelOrderAndNavigateToStart(
    viewModel: OrderViewModel,
    navController: NavHostController
) {
    viewModel.resetOrder()
    navController.popBackStack(CupcakeScreen.Start.name, inclusive = false)
}

```

#### 5. Función para compartir resumen del pedido

```

private fun shareOrder(context: Context, subject: String, summary: String) {
    val intent = Intent(Intent.ACTION_SEND).apply {
        type = "text/plain"
        putExtra(Intent.EXTRA_SUBJECT, subject)
        putExtra(Intent.EXTRA_TEXT, summary)
    }
    context.startActivity(
        Intent.createChooser(
            intent,
            context.getString(R.string.new_cupcake_order)
        )
    )
}

```

Training luch:

MainActivity.kt – Código comentado

package com.example.lunchtray

```
import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import com.example.lunchtray.ui.theme.LunchTrayTheme

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

        // Define el contenido de la actividad con Compose
        setContent {
            LunchTrayTheme {
                // Llama al Composable principal de la app
                LunchTrayApp()
            }
        }
    }
}
```

### ¿Qué hace este archivo?

MainActivity es el punto de entrada de la app. Usa setContent para iniciar la UI con Jetpack Compose y aplicar el tema. Dentro de ese tema, carga el componente principal LunchTrayApp().

### LunchTrayScreen.kt

```
package com.example.lunchtray
import androidx.annotation.StringRes
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.rememberScrollState
import androidx.compose.foundation.verticalScroll
import androidx.compose.material.icons(Icons)
import androidx.compose.material.icons.filled.ArrowBack
import androidx.compose.material3.*
import androidx.compose.runtime.*
import androidx.compose.ui.Modifier
import androidx.compose.ui.res.dimensionResource
import androidx.compose.ui.res.stringResource
import androidx.lifecycle.viewmodel.compose.viewModel
import androidx.navigation.compose.*
import com.example.lunchtray.datasource.DataSource
```

```
import com.example.lunchtray.ui.*

enum class LunchTrayScreen(@StringRes val title: Int) { Start(title = R.string.app_name),
    Entree(title = R.string.choose_entree),
    SideDish(title = R.string.choose_side_dish),
    Accompaniment(title = R.string.choose_accompaniment),
    Checkout(title = R.string.order_checkout)
}

// Barra superior de la app con botón de regreso opcional
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun LunchTrayAppBar(
    @StringRes currentScreenTitle: Int,
    canNavigateBack: Boolean,
    navigateUp: () -> Unit,
    modifier: Modifier = Modifier
) {
    CenterAlignedTopAppBar(
        title = { Text(stringResource(currentScreenTitle)) },
        modifier = modifier,
        navigationIcon = {
            if (canNavigateBack) {
                IconButton(onClick = navigateUp) {
                    Icon(
                        imageVector = Icons.Filled.ArrowBack,
                        contentDescription = stringResource(R.string.back_button)
                    )
                }
            }
        }
    )
}

@Composable
fun LunchTrayApp() {
    val navController = rememberNavController()
    val backStackEntry by navController.currentBackStackEntryAsState()

    // Determina la pantalla actual
    val currentScreen = LunchTrayScreen.valueOf(
        backStackEntry?.destination?.route ?: LunchTrayScreen.Start.name
    )
}
```

```
// ViewModel que gestiona el estado de la orden
val viewModel: OrderViewModel = viewModel()

Scaffold(
    topBar = {
        LunchTrayAppBar(
            currentScreenTitle = currentScreen.title,
            canNavigateBack = navController.previousBackStackEntry != null,
            navigateUp = { navController.navigateUp() }
        )
    }
) { innerPadding ->

    val uiState by viewModel.uiState.collectAsState()

    NavHost(
        navController = navController,
        startDestination = LunchTrayScreen.Start.name,
        modifier = Modifier.padding(innerPadding)
    ) {
        composable(LunchTrayScreen.Start.name) {
            StartOrderScreen(
                onStartOrderButtonClicked = {
                    navController.navigate(LunchTrayScreen.Entree.name)
                },
                modifier = Modifier.fillMaxSize()
            )
        }

        composable(LunchTrayScreen.Entree.name) {
            EntreeMenuScreen(
                options = DataSource.entreeMenuItems,
                onCancelButtonClicked = {
                    viewModel.resetOrder()
                    navController.popBackStack(LunchTrayScreen.Start.name,
                        false)
                },
                onNextButtonClicked = {
                    navController.navigate(LunchTrayScreen.SideDish.name)
                },
                onSelectionChanged = { item -> viewModel.updateEntree(item) },
                modifier = Modifier.verticalScroll(rememberScrollState())
            )
        }
    }
}
```

```
composable(LunchTrayScreen.SideDish.name) {
    SideDishMenuScreen(
        options = DataSource.sideDishMenuItems,
        onCancelButtonClicked = {
            viewModel.resetOrder()
            navController.popBackStack(LunchTrayScreen.Start.name,
false)
        },
        onNextButtonClicked = {

navController.navigate(LunchTrayScreen.Accompaniment.name)
        },
        onSelectionChanged = { item -> viewModel.updateSideDish(item)
},
        modifier = Modifier.verticalScroll(rememberScrollState())
    )
}

composable(LunchTrayScreen.Accompaniment.name) {
    AccompanimentMenuScreen(
        options = DataSource.accompanimentMenuItems,
        onCancelButtonClicked = {
            viewModel.resetOrder()
            navController.popBackStack(LunchTrayScreen.Start.name,
false)
        },
        onNextButtonClicked = {
            navController.navigate(LunchTrayScreen.Checkout.name)
        },
        onSelectionChanged = { item ->
viewModel.updateAccompaniment(item) },
        modifier = Modifier.verticalScroll(rememberScrollState())
    )
}

composable(LunchTrayScreen.Checkout.name) {
    CheckoutScreen(
        orderUiState = uiState,
        onCancelButtonClicked = {
            viewModel.resetOrder()
            navController.popBackStack(LunchTrayScreen.Start.name,
false)
        },
    ),
```

```
        onNextButtonClicked = {
            viewModel.resetOrder()
            navController.popBackStack(LunchTrayScreen.Start.name,
false)
        },
        modifier = Modifier
            .verticalScroll(rememberScrollState())
            .padding(
                start = dimensionResource(R.dimen.padding_medium),
                end = dimensionResource(R.dimen.padding_medium),
            )
        )
    }
}
```

¿Qué hace este archivo?

Define el flujo de navegación entre pantallas (inicio, menú, acompañamiento y pago). Administra el ViewModel de la orden y construye la interfaz según la pantalla actual.

StartOrderScreen.kt – Código comentado

```

        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ) {
        Button(
            onClick = onStartOrderButtonClicked,
            Modifier.widthIn(min = 250.dp)
        ) {
            Text(stringResource(R.string.start_order))
        }
    }
}

@Preview
@Composable
fun StartOrderPreview(){
    StartOrderScreen(
        onStartOrderButtonClicked = {},
        modifier = Modifier
            .padding(dimensionResource(R.dimen.padding_medium))
            .fillMaxSize()
    )
}

```

¿Qué hace este archivo?

Muestra la pantalla de inicio con un botón que permite al usuario comenzar una orden. Al presionar, se navega a la siguiente pantalla (entradas).

SideDishMenuScreen.kt – Código comentado

```

package com.example.lunchtray.ui
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.rememberScrollState
import androidx.compose.foundation.verticalScroll
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.res.dimensionResource
import androidx.compose.ui.tooling.preview.Preview
import com.example.lunchtray.R
import com.example.lunchtray.datasource.DataSource
import com.example.lunchtray.model.MenuItem
import com.example.lunchtray.model.MenuItem.SideDishItem

// Pantalla de selección de guarnición
@Composable
fun SideDishMenuScreen(

```

```

        options: List<SideDishItem>,
        onCancelButtonClicked: () -> Unit,
        onNextButtonClicked: () -> Unit,
        onSelectionChanged: (SideDishItem) -> Unit,
        modifier: Modifier = Modifier
    ) {
    BaseMenuScreen(
        options = options,
        onCancelButtonClicked = onCancelButtonClicked,
        onNextButtonClicked = onNextButtonClicked,
        onSelectionChanged = onSelectionChanged as (MenuItem) -> Unit,
        modifier = modifier
    )
}

@Preview
@Composable
fun SideDishMenuPreview(){
    SideDishMenuScreen(
        options = DataSource.sideDishMenuItems,
        onNextButtonClicked = {},
        onCancelButtonClicked = {},
        onSelectionChanged = {},
        modifier = Modifier
            .padding(dimensionResource(R.dimen.padding_medium))
            .verticalScroll(rememberScrollState())
    )
}

```

¿Qué hace este archivo?

Muestra el menú de guarniciones. Permite seleccionar una opción, cancelar el pedido o avanzar a la pantalla de acompañamientos.

## Video 2:

En este video profundizamos en los métodos del ciclo de vida de una actividad en Android y entendimos cómo se usan en la práctica para manejar el comportamiento de una app en diferentes situaciones. Como estudiantes, ya sabíamos que Android puede pausar, detener o destruir una actividad dependiendo de lo que el usuario haga o de los recursos que el sistema necesite, pero ahora vimos con más claridad cuándo y por qué se llaman métodos como `onCreate()`, `onStart()`, `onResume()`, `onPause()`, `onStop()` y `onDestroy()`.

Aprendimos que `onCreate()` se llama una sola vez cuando se crea la actividad, y es el lugar ideal para inicializar componentes como la interfaz de usuario. Luego, `onStart()` y `onResume()` se ejecutan cuando la app se vuelve visible y empieza a interactuar con el usuario, mientras que `onPause()` y `onStop()` se llaman cuando la app deja de estar en primer plano o visible, y es ahí donde conviene liberar recursos o guardar información temporal. Finalmente, `onDestroy()` se llama justo antes de que la actividad sea destruida por completo.

Lo que más nos ayudó fue ver un ejemplo en tiempo real, donde se mostraban mensajes (logs) al pasar por cada método, lo que nos permitió visualizar mejor el orden y el flujo de estos eventos. En resumen, este video nos ayudó a entender que usar correctamente estos métodos no solo mejora el rendimiento de la app, sino que también evita errores como pérdida de datos o mal uso de los recursos del sistema. Ahora tenemos más claro cómo y cuándo reaccionar ante los cambios del sistema para que nuestra app sea más robusta y profesional.

### **Lección 3**

#### **Video 1:**

En este video aprendimos cómo manejar el estado en una app de Android usando `ViewModel`, lo cual es esencial cuando queremos que los datos de la app se conserven incluso después de eventos como rotar la pantalla. Como estudiantes, ya sabíamos que al rotar el dispositivo, la actividad se reinicia, lo que puede causar que las variables normales pierdan su valor. Aquí es donde entra el `ViewModel`, que actúa como una capa intermedia entre la UI y los datos de la app.

Nos explicaron que el `ViewModel` guarda el estado de la UI de manera segura y persistente, y que es independiente del ciclo de vida de la actividad, lo cual nos pareció súper útil. Vimos cómo usar `mutableStateOf` para crear variables observables, y cómo estas se pueden actualizar y reflejar automáticamente en la UI usando Jetpack Compose. También aprendimos que este enfoque ayuda a mantener una arquitectura más limpia y organizada, separando claramente la lógica de negocio del diseño visual.

Al ver un ejemplo práctico con un contador, entendimos mejor cómo se conecta la lógica dentro del `ViewModel` con los elementos visibles en la pantalla. En resumen, este video reforzó la idea de que manejar el estado correctamente con `ViewModel` y Compose es clave para crear apps más estables, reactivas y fáciles de mantener.

## Video 2:

En este video aprendimos cómo agregar un ViewModel a nuestra app de Android para manejar datos de manera más eficiente y resistente a cambios de configuración como la rotación de pantalla. Como estudiantes, nos quedó claro que si guardamos información directamente en la actividad, esa información se pierde cuando la actividad se reinicia. Por eso, el ViewModel se vuelve tan importante: nos permite conservar el estado de la app incluso cuando la actividad se destruye y se vuelve a crear.

En el video, vimos paso a paso cómo crear una clase que extiende de ViewModel, cómo declarar variables usando mutableStateOf y cómo exponer esas variables como State para que la UI pueda observarlas sin modificarlas directamente. También nos enseñaron cómo instanciar el ViewModel dentro de una función composable usando viewModel(), lo cual facilita mucho su uso en Jetpack Compose. Nos gustó ver un ejemplo práctico donde se actualizaba un contador y cómo los cambios se reflejaban en la pantalla automáticamente sin tener que escribir código complicado para manejar el ciclo de vida.

En resumen, entendimos que usar un ViewModel no solo mejora la organización del código, sino que garantiza que los datos se mantengan estables y la experiencia del usuario sea más fluida, incluso cuando ocurren interrupciones en la actividad.

Actividades:

MainActivity.kt – Código comentado

```
package com.example.lunch
import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import com.example.lunchtray.ui.theme.LunchTrayTheme

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

        // Define el contenido de la actividad con Compose
        setContent {
            LunchTrayTheme {
                // Llama al Composable principal de la app
                LunchTrayApp()
            }
        }
    }
}
```

```
}
```

¿Qué hace este archivo?

MainActivity es el punto de entrada de la app. Usa setContent para iniciar la UI con Jetpack Compose y aplicar el tema. Dentro de ese tema, carga el componente principal LunchTrayApp().

LunchTrayScreen.kt – Código comentado

```
package com.example.lunchtray
import androidx.annotation.StringRes
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.rememberScrollState
import androidx.compose.foundation.verticalScroll
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.filled.ArrowBack
import androidx.compose.material3.*
import androidx.compose.runtime.*
import androidx.compose.ui.Modifier
import androidx.compose.ui.res.dimensionResource
import androidx.compose.ui.res.stringResource
import androidx.lifecycle.viewmodel.compose.viewModel
import androidx.navigation.compose.*
import com.example.lunchtray.datasource.DataSource
import com.example.lunchtray.ui.*
```

```
enum class LunchTrayScreen(@StringRes val title: Int) {
    Start(title = R.string.app_name),
    Entree(title = R.string.choose_entree),
    SideDish(title = R.string.choose_side_dish),
    Accompaniment(title = R.string.choose_accompaniment),
    Checkout(title = R.string.order_checkout)
}
```

```
// Barra superior de la app con botón de regreso opcional
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun LunchTrayAppBar(
    @StringRes currentScreenTitle: Int,
    canNavigateBack: Boolean,
    navigateUp: () -> Unit,
    modifier: Modifier = Modifier
) {
    CenterAlignedTopAppBar(
```

```
title = { Text(stringResource(currentScreenTitle)) },
modifier = modifier,
navigationIcon = {
    if (canNavigateBack) {
        IconButton(onClick = navigateUp) {
            Icon(
                imageVector = Icons.Filled.ArrowBack,
                contentDescription = stringResource(R.string.back_button)
            )
        }
    }
}
}

@Composable
fun LunchTrayApp() {
    val navController = rememberNavController()
    val backStackEntry by navController.currentBackStackEntryAsState()

    // Determina la pantalla actual
    val currentScreen = LunchTrayScreen.valueOf(
        backStackEntry?.destination?.route ?: LunchTrayScreen.Start.name
    )

    // ViewModel que gestiona el estado de la orden
    val viewModel: OrderViewModel = viewModel()

    Scaffold(
        topBar = {
            LunchTrayAppBar(
                currentScreenTitle = currentScreen.title,
                canNavigateBack = navController.previousBackStackEntry != null,
                navigateUp = { navController.navigateUp() }
            )
        }
    ) { innerPadding ->

        val uiState by viewModel.uiState.collectAsState()

        NavHost(
            navController = navController,
            startDestination = LunchTrayScreen.Start.name,
            modifier = Modifier.padding(innerPadding)
        )
    }
}
```

```
) {  
    composable(LunchTrayScreen.Start.name) {  
        StartOrderScreen(  
            onStartOrderButtonClicked = {  
                navController.navigate(LunchTrayScreen.Entree.name)  
            },  
            modifier = Modifier.fillMaxSize()  
        )  
    }  
  
    composable(LunchTrayScreen.Entree.name) {  
        EntreeMenuScreen(  
            options = DataSource.entreeMenuItems,  
            onCancelButtonClicked = {  
                viewModel.resetOrder()  
                navController.popBackStack(LunchTrayScreen.Start.name,  
false)  
            },  
            onNextButtonClicked = {  
                navController.navigate(LunchTrayScreen.SideDish.name)  
            },  
            onSelectionChanged = { item -> viewModel.updateEntree(item) },  
            modifier = Modifier.verticalScroll(rememberScrollState())  
        )  
    }  
  
    composable(LunchTrayScreen.SideDish.name) {  
        SideDishMenuScreen(  
            options = DataSource.sideDishMenuItems,  
            onCancelButtonClicked = {  
                viewModel.resetOrder()  
                navController.popBackStack(LunchTrayScreen.Start.name,  
false)  
            },  
            onNextButtonClicked = {  
  
                navController.navigate(LunchTrayScreen.Accompaniment.name)  
            },  
            onSelectionChanged = { item -> viewModel.updateSideDish(item)  
},  
            modifier = Modifier.verticalScroll(rememberScrollState())  
        )  
    }  
}
```

```

composable(LunchTrayScreen.Accompaniment.name) {
    AccompanimentMenuScreen(
        options = DataSource.accompanimentMenuItems,
        onCancelButtonClicked = {
            viewModel.resetOrder()
            navController.popBackStack(LunchTrayScreen.Start.name,
false)
        },
        onNextButtonClicked = {
            navController.navigate(LunchTrayScreen.Checkout.name)
        },
        onSelectionChanged = { item ->
            viewModel.updateAccompaniment(item)
        },
        modifier = Modifier.verticalScroll(rememberScrollState())
    )
}

composable(LunchTrayScreen.Checkout.name) {
    CheckoutScreen(
        orderUiState = uiState,
        onCancelButtonClicked = {
            viewModel.resetOrder()
            navController.popBackStack(LunchTrayScreen.Start.name,
false)
        },
        onNextButtonClicked = {
            viewModel.resetOrder()
            navController.popBackStack(LunchTrayScreen.Start.name,
false)
        },
        modifier = Modifier
            .verticalScroll(rememberScrollState())
            .padding(
                start = dimensionResource(R.dimen.padding_medium),
                end = dimensionResource(R.dimen.padding_medium),
            )
    )
}
}
}
}

```

¿Qué hace este archivo?

Define el flujo de navegación entre pantallas (inicio, menú, acompañamiento y

pago). Administra el ViewModel de la orden y construye la interfaz según la pantalla actual.

#### StartOrderScreen.kt – Código comentado

```
package com.example.lunchtray.ui
import androidx.compose.foundation.layout.*
import androidx.compose.material3.Button
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.res.dimensionResource
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.ui.unit.dp
import com.example.lunchtray.R

// Pantalla inicial con botón para comenzar la orden
@Composable
fun StartOrderScreen(
    onStartOrderButtonClicked: () -> Unit,
    modifier: Modifier = Modifier
) {
    Column(
        modifier = modifier,
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ) {
        Button(
            onClick = onStartOrderButtonClicked,
            Modifier.widthIn(min = 250.dp)
        ) {
            Text(stringResource(R.string.start_order))
        }
    }
}

@Preview
@Composable
fun StartOrderPreview(){
    StartOrderScreen(
        onStartOrderButtonClicked = {},
        modifier = Modifier
```

```
    .padding(dimensionResource(R.dimen.padding_medium))
    .fillMaxSize()
)
}
```

¿Qué hace este archivo?

Muestra la pantalla de inicio con un botón que permite al usuario comenzar una orden. Al presionar, se navega a la siguiente pantalla (entradas).

SideDishMenuScreen.kt – Código comentado

```
package com.example.lunchtray.ui
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.rememberScrollState
import androidx.compose.foundation.verticalScroll
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.res.dimensionResource
import androidx.compose.ui.tooling.preview.Preview
import com.example.lunchtray.R
import com.example.lunchtray.datasource.DataSource
import com.example.lunchtray.model.MenuItem
import com.example.lunchtray.model.MenuItem.SideDishItem

// Pantalla de selección de guarnición
@Composable
fun SideDishMenuScreen(
    options: List<SideDishItem>,
    onCancelButtonClicked: () -> Unit,
    onNextButtonClicked: () -> Unit,
    onSelectionChanged: (SideDishItem) -> Unit,
    modifier: Modifier = Modifier
) {
    BaseMenuScreen(
        options = options,
        onCancelButtonClicked = onCancelButtonClicked,
        onNextButtonClicked = onNextButtonClicked,
        onSelectionChanged = onSelectionChanged as (MenuItem) -> Unit,
        modifier = modifier
    )
}

@Preview
@Composable
fun SideDishMenuPreview(){
```

```
SideDishMenuScreen(  
    options = DataSource.sideDishMenuItems,  
    onNextButtonClicked = {},  
    onCancelButtonClicked = {},  
    onSelectionChanged = {},  
    modifier = Modifier  
        .padding(dimensionResource(R.dimen.padding_medium))  
        .verticalScroll(rememberScrollState())  
)  
}
```

¿Qué hace este archivo?

Muestra el menú de guarniciones. Permite seleccionar una opción, cancelar el pedido o avanzar a la pantalla de acompañamientos.

Trainin nav:

MainActivity.kt – Código comentado

```
package com.example.reply
```

```
import android.os.Bundle  
import androidx.activity.ComponentActivity  
import androidx.activity.compose.setContent  
import androidx.activity.enableEdgeToEdge  
import androidx.compose.foundation.layout.WindowInsets  
import androidx.compose.foundation.layout.asPaddingValues  
import androidx.compose.foundation.layout.calculateEndPadding  
import androidx.compose.foundation.layout.calculateStartPadding  
import androidx.compose.foundation.layout.padding  
import androidx.compose.foundation.layout.safeDrawing  
import androidx.compose.material3.Surface  
import  
    androidx.compose.material3.windowsizeclass.ExperimentalMaterial3WindowSizeClassApi  
import androidx.compose.material3.windowsizeclass.WindowWidthSizeClass  
import  
    androidx.compose.material3.windowsizeclass.calculateWindowSizeClass  
import androidx.compose.runtime.Composable  
import androidx.compose.ui.Modifier  
import androidx.compose.ui.platform.LocalLayoutDirection  
import androidx.compose.ui.tooling.preview.Preview  
import com.example.reply.ui.ReplyApp  
import com.example.reply.ui.theme.ReplyTheme
```

```
// Actividad principal de la app Reply
class MainActivity : ComponentActivity() {

    @OptIn(ExperimentalMaterial3WindowSizeClassApi::class)
    override fun onCreate(savedInstanceState: Bundle?) {
        // Habilita el diseño hasta los bordes de pantalla
        enableEdgeToEdge()
        super.onCreate(savedInstanceState)

        // Establece el contenido de la UI con Jetpack Compose
        setContent {
            // Aplica el tema de la app
            ReplyTheme {
                val layoutDirection = LocalLayoutDirection.current

                // Surface es el contenedor base que respeta los márgenes seguros
                // (Safe Drawing Insets)
                Surface (
                    modifier = Modifier
                        .padding(
                            start = WindowInsets.safeDrawing.asPaddingValues()
                                .calculateStartPadding(layoutDirection),
                            end = WindowInsets.safeDrawing.asPaddingValues()
                                .calculateEndPadding(layoutDirection)
                        )
                ){
                    // Calcula la clase de tamaño de ventana (compacto, medio o
                    // expandido)
                    val windowSize = calculateWindowSizeClass(this)

                    // Carga la app Reply pasando el tamaño de ventana
                    ReplyApp(
                        windowSize = windowSize.widthSizeClass,
                    )
                }
            }
        }
    }

    // Vista previa en modo compacto (útil para tablets o móviles)
    @Preview(showBackground = true)
    @Composable
    fun ReplyAppCompactPreview() {
```

```
ReplyTheme {  
    Surface {  
        ReplyApp(  
            windowSize = WindowWidthSizeClass.Compact,  
        )  
    }  
}  
}
```

¿Qué hace este archivo?

MainActivity es el punto de entrada de la app. Usa enableEdgeToEdge() para ocupar toda la pantalla, luego aplica el tema y carga ReplyApp, respetando los márgenes seguros y adaptándose al tamaño de ventana del dispositivo.

MailboxType.kt – Código comentado

```
package com.example.reply.data  
  
/**  
 * Clase enum que define los tipos de bandejas de correo electrónico.  
 */  
enum class MailboxType { // Bandeja de entrada  
    Inbox,  
  
    // Borradores  
    Drafts,  
  
    // Correos enviados  
    Sent,  
  
    // Correos marcados como spam  
    Spam  
}
```

¿Qué hace este archivo?

Define un enum con los tipos de bandeja de correo electrónico que puede tener un email: Inbox, Drafts, Sent, y Spam.

Email.kt – Código comentado

```
package com.example.reply.data  
import androidx.annotation.StringRes
```

```

/**
 * Clase de datos para representar un correo electrónico.
 */
data class Email(
    /** ID único del correo **/
    val id: Long,

    /** Emisor del correo **/
    val sender: Account,

    /** Destinatarios del correo **/
    val recipients: List<Account> = emptyList(),

    /** Asunto del correo (referencia a recurso String) **/
    @StringRes val subject: Int = -1,

    /** Cuerpo del correo (referencia a recurso String) **/
    @StringRes val body: Int = -1,

    /** Bandeja donde se encuentra el correo **/
    var mailbox: MailboxType = MailboxType.Inbox,

    /** Tiempo relativo desde que fue creado (como "hace 2 horas") **/
    @StringRes var createdAt: Int = -1
)

```

### ¿Qué hace este archivo?

Define un modelo de datos Email que representa un correo con su ID, emisor (sender), destinatarios, asunto, cuerpo, la bandeja (Inbox, etc.) y una marca de tiempo relativa (createdAt).

### Account.kt – Código comentado

```

package com.example.reply.data

import androidx.annotation.DrawableRes
import androidx.annotation.StringRes

/**
 * Clase que representa una cuenta de usuario.
 */
data class Account(

```

```
/** ID único del usuario */
val id: Long,

/** Nombre del usuario */
@StringRes val firstName: Int,

/** Apellido del usuario */
@StringRes val lastName: Int,

/** Dirección de correo del usuario */
@StringRes val email: Int,

/** ID del recurso de imagen para el avatar */
@DrawableRes val avatar: Int
)
```

¿Qué hace este archivo?

Define la clase Account, que modela un usuario con nombre, apellido, correo, avatar y un ID único. Se usa para representar tanto remitentes como destinatarios en los correos (Email).

### Video 3:

En este video aprendimos cómo integrar un ViewModel en una app que ya existe, lo cual es muy útil cuando queremos mejorar la arquitectura de nuestra app sin empezar desde cero. Como estudiantes, ya habíamos trabajado con apps que guardaban datos directamente en la actividad, pero ahora entendemos que eso no es lo más adecuado, sobre todo cuando hay cambios de configuración como la rotación de pantalla.

Lo interesante del video fue ver cómo se toma una app funcional, como el juego de Unscramble, y se modifica para que toda la lógica del estado, como las palabras usadas, la puntuación y la palabra actual, se maneje desde un ViewModel. Aprendimos a mover las variables que antes estaban en la actividad hacia el ViewModel, a usar mutableStateOf para que los cambios sean reactivos, y a acceder al ViewModel desde los composables con viewModel().

También vimos cómo encapsular los datos para que no puedan modificarse directamente desde la UI, y cómo actualizar los valores de forma segura desde el ViewModel. En resumen, este video nos enseñó que agregar un ViewModel a una app ya existente es una excelente manera de mejorar su organización, hacerla más robusta y prepararla para escalar o mantenerse más fácilmente a largo plazo.