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 este video explica cómo almacenar datos localmente en una aplicación para que persistan, incluso si la aplicación se cierra. Se menciona la necesidad de gestionar gastos mediante una base de datos en lugar de una simple lista, que se perdería al cerrar la app. El contenido introduce SQLite, una base de datos utilizada en Android para almacenar información estructurada en tablas con filas y columnas. Se presenta un ejemplo de una tabla con parques nacionales, donde cada fila representa un parque con propiedades como ID, nombre y ciudad.

También se explican conceptos clave de SQL, el lenguaje utilizado para interactuar con bases de datos, incluyendo operaciones como SELECT (para consultar datos), INSERT (para agregar registros), UPDATE (para modificar información) y DELETE (para eliminar registros). Se detallan métodos para filtrar datos con la cláusula WHERE, ordenar resultados y limitar el número de registros devueltos. Finalmente, se destaca la importancia de comprender y manipular bases de datos dentro de una aplicación, lo que permitirá a los desarrolladores mejorar la funcionalidad de sus apps y optimizar el almacenamiento de datos.

Video 2:

En este video, se presenta una introducción al desarrollo en Android utilizando Compose, específicamente en relación con el manejo de bases de datos SQLite. El instructor destaca la importancia de comprender las bases de datos como colecciones de datos estructurados, donde las filas representan entradas individuales y las columnas sus atributos. Se mencionan conceptos clave de SQL, incluyendo las operaciones básicas como seleccionar, insertar, actualizar y eliminar datos, así como el uso de filtros y agrupamientos. Se anticipa que en el próximo curso, los estudiantes aprenderán a utilizar la API de Room, que facilitará la interacción con bases de datos SQLite en sus aplicaciones.

El instructor explica que una base de datos SQLite es una colección organizada de datos que permite almacenar y manipular información de manera eficiente. Se enfatiza que las filas son las entradas individuales y las columnas definen los atributos de esos datos. Este entendimiento es crucial para cualquier desarrollador que maneje datos en sus aplicaciones.

Se presenta el lenguaje de consulta estructurado (SQL), que permite a los desarrolladores leer y procesar datos en bases de datos relacionales. Se enseñan las sentencias básicas que incluyen Seleccionar, Insertar, Actualizar y Eliminar, proporcionando a los estudiantes una base sólida para interactuar con bases de datos.

El contenido aborda cómo combinar operadores lógicos como Not, And y Or para filtrar datos de manera efectiva mediante la cláusula WHERE. Esta habilidad es fundamental para realizar consultas complejas y recuperar la información

específica que se necesita de la base de datos. Se menciona la capacidad de ordenar y agrupar datos por columnas específicas, lo que permite a los desarrolladores organizar la información de manera más comprensible. También se discute cómo limitar el número de resultados en las consultas, facilitando la obtención de datos relevantes en un formato manejable.

El instructor anticipa que el siguiente curso se centrará en la API de Room, que proporciona una capa de abstracción sobre SQLite, simplificando su uso en aplicaciones Android. Este conocimiento permitirá a los estudiantes construir aplicaciones más eficientes y efectivas en el manejo de datos. Se invita a los estudiantes a aplicar lo aprendido construyendo una aplicación de catálogo de productos, donde podrán guardar, mostrar, actualizar y eliminar artículos. Esta aplicación práctica servirá como una excelente oportunidad para aplicar los conceptos de bases de datos en un contexto real de desarrollo.

Lección 2

Video 1:

En este video explica la programación reactiva en Android utilizando Kotlin Flow para modelar y gestionar el flujo de datos en una aplicación. Se compara este proceso con la infraestructura de tuberías que transportan agua, destacando la ventaja de observar datos en lugar de solicitarlos manualmente.

Se menciona que en una aplicación Android, los datos pueden enviarse desde diversas fuentes, como bases de datos locales o servidores, y se analiza cómo Flow permite transformar y exponer estos datos de manera eficiente. También se introduce el concepto de productores y consumidores, donde un productor emite datos y un consumidor los recoge para mostrarlos en la interfaz. El texto explica cómo crear Flows mediante el Flow Builder, que permite emitir y actualizar información en coroutines de Kotlin. Además, se describen operadores intermedios, como map, que transforman los datos en diferentes estructuras para ajustarse a la capa de la aplicación.

Finalmente, se enfatiza la importancia de mantener el flujo de datos en una sola dirección para minimizar errores y mejorar la administración del estado de la app. Se destaca que Flow es parte de Kotlin Coroutines y es ampliamente utilizado en bibliotecas populares como Room, Retrofit y WorkManager.

Video 2:

Actividad 1:

Código:

```
package com.example.inventory

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

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        enableEdgeToEdge()
        super.onCreate(savedInstanceState)
        setContent {
            InventoryTheme {
                Surface(
                    modifier = Modifier.fillMaxSize(),
                ) {
                    InventoryApp()
                }
            }
        }
    }
}
```

Explicación:

MainActivity: El código define la actividad principal de una aplicación Android llamada MainActivity, que utiliza Jetpack Compose para construir su interfaz de usuario. Al iniciarse la actividad, se llama a enableEdgeToEdge() para permitir que la interfaz ocupe toda el área de la pantalla, incluidos los bordes. Luego, mediante setContent, se establece el contenido visual de la app utilizando el tema personalizado InventoryTheme. Dentro de este tema, se utiliza un componente Surface que ocupa todo el tamaño de la pantalla gracias al modificador fillMaxSize(), y dentro de este contenedor se invoca la función InventoryApp(), que probablemente define la estructura principal de la interfaz de usuario de la aplicación.

Código:

```
package com.example.inventory
import android.app.Application
import com.example.inventory.data.AppContainer
import com.example.inventory.data.AppDataContainer

class InventoryApplication : Application() {
```

```
    /**
     * AppContainer instance used by the rest of classes to obtain dependencies
     */
    lateinit var container: AppContainer

    override fun onCreate() {
        super.onCreate()
        container = AppDataContainer(this)
    }
}
```

Explicación:

InventoryApplication: Este código define la clase InventoryApplication, que extiende de Application y se utiliza para inicializar dependencias globales en la app Inventory. Al sobrescribir el método onCreate(), se crea una instancia de AppDataContainer y se asigna a la propiedad container, que es de tipo AppContainer. Esta instancia actúa como un contenedor de dependencias que otras clases de la aplicación pueden usar para acceder a los objetos necesarios, promoviendo una arquitectura más limpia y modular. Así, InventoryApplication centraliza la gestión de dependencias y asegura que estén disponibles desde el inicio del ciclo de vida de la aplicación.

Código:

```
package com.example.inventory
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
```

```
fun InventoryApp(navController: NavHostController = rememberNavController())
{
    InventoryNavHost(navController = navController)
}


@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun InventoryTopAppBar(
    title: String,
    canNavigateBack: Boolean,
    modifier: Modifier = Modifier,
    scrollBehavior: TopAppBarScrollBehavior? = null,
    navigateUp: () -> Unit = {}
) {
    CenterAlignedTopAppBar(
        title = { Text(title) },
        modifier = modifier,
        scrollBehavior = scrollBehavior,
        navigationIcon = {
            if (canNavigateBack) {
                IconButton(onClick = navigateUp) {
                    Icon(
                        imageVector = Filled.ArrowBack,
                        contentDescription = stringResource(string.back_button)
                    )
                }
            }
        }
    )
}
```

Explicación:

InventoryApp: Este código define dos funciones composables clave para una aplicación Android construida con Jetpack Compose:

InventoryApp: Es el punto de entrada para la interfaz de usuario de la aplicación. Recibe un controlador de navegación (NavHostController), y por defecto crea uno si no se proporciona. Luego llama a InventoryNavHost, que se encarga de gestionar la navegación entre las diferentes pantallas de la app.

InventoryTopAppBar: Define una barra superior (TopAppBar) centrada, reutilizable y configurable. Acepta parámetros como el título a mostrar, si se puede navegar hacia atrás (canNavigateBack), una función navigateUp para

manejar la acción de retroceso, y un comportamiento de desplazamiento (scrollBehavior). Si se permite volver atrás, muestra un ícono de flecha que al presionarse ejecuta la función navigateUp.

Actividad 2:

Código:

```
package com.example.inventory

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

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        enableEdgeToEdge()
        super.onCreate(savedInstanceState)
        setContent {
            InventoryTheme {
                Surface(
                    modifier = Modifier.fillMaxSize(),
                ) {
                    InventoryApp()
                }
            }
        }
    }
}
```

Explicación:

Este código define la actividad principal de la aplicación Android Inventory, utilizando Jetpack Compose para construir la interfaz de usuario. La clase MainActivity hereda de ComponentActivity y sobrescribe el método onCreate. Dentro de este método, primero se habilita el uso de toda el área de la pantalla con enableEdgeToEdge(). Luego, se establece el contenido de la pantalla con setContent, aplicando el tema personalizado InventoryTheme. Dentro del tema, se utiliza un contenedor Surface que ocupa todo el espacio disponible en pantalla (fillMaxSize()), y dentro de él se invoca la función composable InventoryApp(),

que lanza la estructura principal de la interfaz de usuario de la aplicación. En conjunto, este archivo configura y muestra la pantalla inicial de la app.

Código:

```
package com.example.inventory

import android.app.Application
import com.example.inventory.data.AppContainer
import com.example.inventory.data.AppDataContainer

class InventoryApplication : Application() {

    /**
     * AppContainer instance used by the rest of classes to obtain dependencies
     */
    lateinit var container: AppContainer

    override fun onCreate() {
        super.onCreate()
        container = AppDataContainer(this)
    }
}
```

Explicación:

Este código define la clase InventoryApplication, que extiende de Application y se utiliza para inicializar componentes globales cuando la aplicación se lanza. Dentro del método onCreate(), se crea una instancia de AppDataContainer, la cual se asigna a la propiedad container de tipo AppContainer. Este contenedor actúa como un gestor centralizado de dependencias, permitiendo que otras partes de la aplicación accedan fácilmente a objetos o servicios necesarios sin tener que instanciarlos directamente. Así, se promueve una arquitectura más organizada, desacoplada y fácil de mantener mediante la inyección de dependencias.

Código:

```
package com.example.inventory

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

/**
 * Top level composable that represents screens for the application.
 */
@Composable
fun InventoryApp(navController: NavHostController = rememberNavController(

{
    InventoryNavHost(navController = navController)
}

/**
 * App bar to display title and conditionally display the back navigation.
 */
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun InventoryTopAppBar(
    title: String,
    canNavigateBack: Boolean,
    modifier: Modifier = Modifier,
    scrollBehavior: TopAppBarScrollBehavior? = null,
    navigateUp: () -> Unit = {}
) {
    CenterAlignedTopAppBar(
        title = { Text(title) },
        modifier = modifier,
        scrollBehavior = scrollBehavior,
        navigationIcon = {
            if (canNavigateBack) {
                IconButton(onClick = navigateUp) {
                    Icon(
                        imageVector = Filled.ArrowBack,
```

```
                    contentDescription = stringResource(string.back_button)
                )
            }
        }
    }
)
}
```

Explicación:

Este código define dos componentes principales de la interfaz de usuario para una aplicación Android utilizando Jetpack Compose:

- InventoryApp es una función composable que representa la estructura principal de navegación de la aplicación. Utiliza un NavHostController, que por defecto se crea con rememberNavController(), y lo pasa a InventoryNavHost, que se encarga de definir las rutas y pantallas de la app. Este componente es el punto de entrada visual a las distintas pantallas de la aplicación.
- InventoryTopAppBar es una barra superior reutilizable que muestra un título centrado y, si se permite, un botón de navegación para regresar. Este botón muestra un ícono de flecha hacia atrás (ArrowBack) y ejecuta la función navigateUp cuando se presiona. La barra también puede aceptar un comportamiento de desplazamiento (scrollBehavior) y un modificador visual. El uso de @OptIn(ExperimentalMaterial3Api::class) indica que se está utilizando una API experimental de Material 3.

Actividad 3:

Código:

```
package com.example.busschedule

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import com.example.busschedule.ui.BusScheduleApp
import com.example.busschedule.ui.theme.BusScheduleTheme

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            BusScheduleTheme {
                BusScheduleApp()
```

```
        }
      }
    }
}
```

Explicación:

Este código define la clase MainActivity, que es la actividad principal de la aplicación BusSchedule, desarrollada con Jetpack Compose. La clase hereda de ComponentActivity y sobrescribe el método onCreate, donde se establece el contenido de la interfaz de usuario mediante la función setContent. Dentro de este bloque, se aplica el tema personalizado BusScheduleTheme, y se muestra la función composable BusScheduleApp(), que probablemente organiza la estructura principal de la interfaz y navegación de la aplicación. En resumen, este archivo configura y lanza la pantalla inicial de la app utilizando una arquitectura basada en Compose.

Código:

```
package com.example.busschedule

import android.app.Application
import com.example.busschedule.data.AppDatabase

class BusScheduleApplication: Application() {
    val database: AppDatabase by lazy { AppDatabase.getDatabase(this) }
}
```

Explicación:

Este código define la clase BusScheduleApplication, que extiende de Application y se encarga de inicializar la base de datos de la aplicación BusSchedule. Dentro de ella, se declara una propiedad database de tipo AppDatabase, la cual se inicializa de forma perezosa (lazy) usando el método getDatabase(this), asegurando que la base de datos se cree solo cuando sea necesaria. Esta clase permite acceder a una instancia única de la base de datos desde cualquier parte de la aplicación, facilitando una arquitectura centralizada y eficiente para el manejo de datos persistentes.

Video 3:

En esta conferencia se da la bienvenida al final de la segunda etapa del curso, en la que se aprendió sobre la "Líbría de la Cuerza", una capa de abstracción que facilita la interacción con la base de datos, permitiendo escribir código más seguro al verificar consultas en tiempo de compilación. Se recomienda usar esta librería en lugar de acceder directamente a la base de datos. Durante esta etapa, se creó una aplicación de inventario usando Room para guardar datos en una base de datos SQLite. Se aprendió a crear la base de datos, implementar los tres componentes principales de Room (entidad, DAO y clase de base de datos), y realizar operaciones básicas como leer, insertar, actualizar y eliminar datos. En la siguiente etapa, se aprenderá a almacenar datos simples como pares clave-valor usando DataStore, y se aplicará este conocimiento para actualizar la app de postres permitiendo que guarde y muestre la preferencia del usuario entre un diseño de lista o cuadrícula.

Lección 3

Video 1:

Jetpack Datastore es una biblioteca moderna de Android diseñada para reemplazar las SharedPreferences, ofreciendo una forma segura, eficiente y asincrónica de almacenar pequeños volúmenes de datos como configuraciones de usuario o estados de la aplicación. Está basada en corrutinas y Flows de

Kotlin, lo que permite operaciones sin bloqueo del hilo principal, a diferencia de SharedPreferences, que pueden causar errores de interfaz (UI jank) o ANRs al realizar operaciones de E/S en el hilo principal.

Datastore cuenta con dos implementaciones:

Preferences Datastore: similar a SharedPreferences, utiliza pares clave-valor y permite una migración rápida.

Proto Datastore: usa protocol buffers, define un esquema para los datos y proporciona seguridad de tipo completo, ideal para estructuras de datos más complejas como listas o enums.

A diferencia de SharedPreferences, Datastore es más robusto frente a errores de tipo, ofrece operaciones atómicas con consistencia de datos garantizada y facilita la migración desde preferencias antiguas con mecanismos integrados. Además, permite capturar excepciones de forma estructurada usando los mecanismos de Kotlin Flow.

Por otro lado, si el proyecto requiere manejar datos complejos, grandes, con integridad referencial o actualizaciones parciales, se recomienda usar Room. En cambio, para datos simples como preferencias, Datastore es más apropiado.

Video 2:

En esta presentación de la serie MatSkill se introduce Jetpack DataStore, una alternativa moderna y más segura a SharedPreferences para almacenar datos simples en aplicaciones Android. Se explican sus dos implementaciones: Preferences DataStore, que usa pares clave-valor sin esquema, y Proto DataStore, que permite definir esquemas con Protocol Buffers y brinda seguridad de tipos. Se destacan los problemas comunes de SharedPreferences, como errores de tipo, bloqueos del hilo principal, falta de consistencia y riesgos de excepciones en tiempo de ejecución, y cómo DataStore los resuelve con una API completamente asíncrona basada en Kotlin coroutines y Flow, operaciones en segundo plano y mejor manejo de errores. También se aborda la facilidad de migración desde SharedPreferences a DataStore. Finalmente, se hace una comparación con Room, indicando que Room es ideal para conjuntos de datos grandes y complejos con relaciones entre tablas, mientras que DataStore es más adecuado para datos pequeños y simples como estados de la app o preferencias del usuario. En los próximos episodios, se profundizará en cómo usar Proto y Preferences DataStore, incluyendo lectura, escritura, manejo de errores y migración.

Actividad 1:

Código:

```
package com.example.dessertrelease

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import com.example.dessertrelease.ui.DessertReleaseApp
import com.example.dessertrelease.ui.theme.DessertReleaseTheme

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            DessertReleaseTheme {
                DessertReleaseApp()
            }
        }
    }
}
```

Explicación:

Este código define la clase MainActivity, que actúa como la actividad principal de la aplicación DessertRelease, construida con Jetpack Compose. La clase hereda de ComponentActivity y sobrescribe el método onCreate, donde se utiliza setContent para establecer la interfaz de usuario. Dentro del bloque setContent, se aplica el tema visual personalizado DessertReleaseTheme, y se llama a la función composable DessertReleaseApp(), que probablemente organiza la navegación y el contenido principal de la app. En conjunto, este archivo inicia y configura la pantalla principal de la aplicación con una arquitectura moderna basada en Compose.

Código:

```
package com.example.dessertrelease

import android.app.Application
import android.content.Context
import androidx.datastore.core.DataStore
import androidx.datastore.preferences.core.Preferences
import androidx.datastore.preferences.preferencesDataStore
import com.example.dessertrelease.data.UserPreferencesRepository
```

```
private const val LAYOUT_PREFERENCE_NAME = "layout_preferences"
private val Context.dataStore: DataStore<Preferences> by
preferencesDataStore(
    name = LAYOUT_PREFERENCE_NAME
)

/*
 * Custom app entry point for manual dependency injection
 */
class DessertReleaseApplication: Application() {
    lateinit var userPreferencesRepository: UserPreferencesRepository

    override fun onCreate() {
        super.onCreate()
        userPreferencesRepository = UserPreferencesRepository(dataStore)
    }
}
```

Explicación:

Este código define la clase DessertReleaseApplication, que actúa como el punto de entrada personalizado de la aplicación DessertRelease. Hereda de Application y se utiliza para configurar dependencias globales, en este caso una instancia de UserPreferencesRepository, la cual gestiona las preferencias del usuario usando Jetpack DataStore. Para ello, se declara una extensión dataStore sobre Context utilizando preferencesDataStore con el nombre "layout_preferences", lo que permite acceder a un almacén persistente de preferencias. En el método onCreate, se inicializa userPreferencesRepository con esta instancia de dataStore, habilitando así una forma centralizada y desacoplada de manejar configuraciones del usuario en toda la aplicación.

Código:

```
package com.example.dessertrelease.data

import android.util.Log
import androidx.datastore.core.DataStore
import androidx.datastore.preferences.core.Preferences
import androidx.datastore.preferences.core.booleanPreferencesKey
import androidx.datastore.preferences.core.edit
import androidx.datastore.preferences.core.emptyPreferences
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.catch
```

```
import kotlinx.coroutines.flow.map
import java.io.IOException

/*
 * Concrete class implementation to access data store
 */
class UserPreferencesRepository(
    private val dataStore: DataStore<Preferences>
) {
    private companion object {
        val IS_LINEAR_LAYOUT = booleanPreferencesKey("is_linear_layout")
        const val TAG = "UserPreferencesRepo"
    }

    val isLinearLayout: Flow<Boolean> = dataStore.data
        .catch {
            if (it is IOException) {
                Log.e(TAG, "Error reading preferences.", it)
                emit(emptyPreferences())
            } else {
                throw it
            }
        }
        .map { preferences ->
            preferences[IS_LINEAR_LAYOUT] ?: true
        }

    suspend fun saveLayoutPreference(isLinearLayout: Boolean) {
        dataStore.edit { preferences ->
            preferences[IS_LINEAR_LAYOUT] = isLinearLayout
        }
    }
}
```

Explicación:

Este código define la clase UserPreferencesRepository, que se encarga de gestionar las preferencias del usuario relacionadas con el diseño de la interfaz (lineal o en cuadrícula) utilizando Jetpack DataStore. La clase recibe una instancia de DataStore<Preferences> y proporciona una propiedad isLinearLayout, que es un Flow<Boolean> que emite el valor actual de la preferencia is_linear_layout, usando como valor predeterminado true si no se encuentra configurada.

El flujo maneja posibles errores de lectura (por ejemplo, IOException) mediante un bloque catch, registrando el error con Log.e y emitiendo preferencias vacías en caso de fallo. Además, la clase incluye la función suspend saveLayoutPreference, que permite guardar un nuevo valor para isLinearLayout editando las preferencias almacenadas.

Video 3:

En la unidad 6, nosotros aprendimos a usar bases de datos SQLite, la librería Room y DataStore para hacer apps más avanzadas. Al principio entendimos qué es una base de datos y cómo funciona con SQL, como insertar, actualizar o borrar datos. Luego, con Room, pudimos organizar mejor el código usando entidades, DAOs y clases de base de datos, y hasta hicimos una app de inventario. Después aprendimos a usar DataStore para guardar preferencias del usuario, como cambiar entre vista de lista y cuadrícula. Todo esto nos ayudó a crear apps más útiles y personalizadas. Ahora estamos listos para la siguiente unidad donde veremos cómo ejecutar tareas en segundo plano con WorkManager.