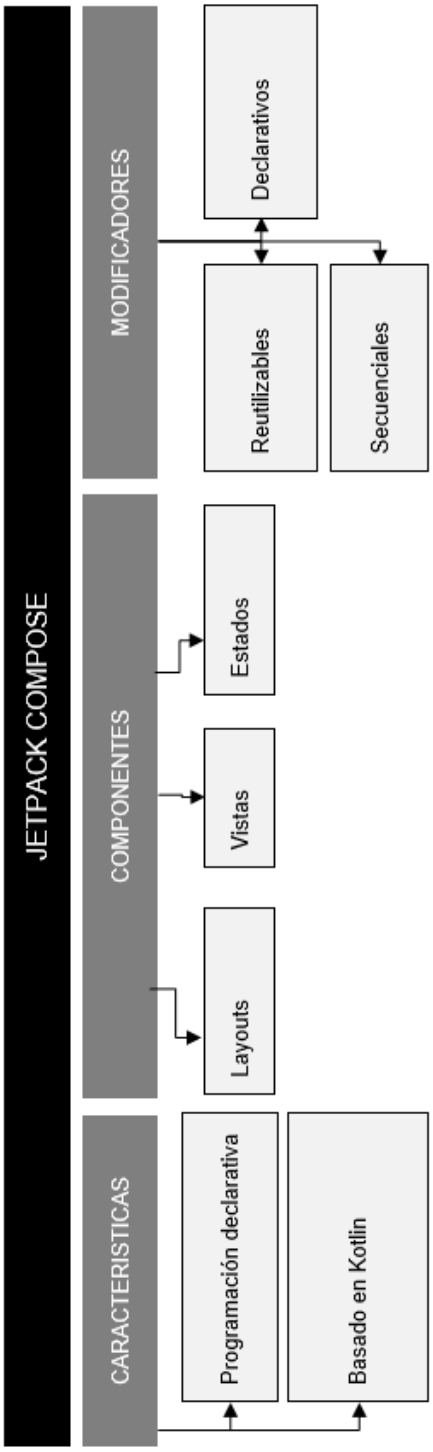
Programación multimedia y dispositivos móviles

Diseño de interfaces con JetPack Compose

Índice

Esquema


JETPACK COMPOSE
CARACTERISTICAS
Programación declarativa
Basado en Kotlin
COMPONENTES
Layouts
Vistas
Estados
MODIFICADORES
Declarativos
Reutilizables
Secuenciales

Material de estudio

4.1. Introducción y objetivos

En el transcurso de este tema se realizará una introducción al desarrollo de interfaces de usuario declarativas con Jetpack Compose, el nuevo kit de herramientas de Android para crear aplicaciones móviles de manera más eficiente y moderna. Se analizarán sus características y componentes principales, comparándolo con el sistema de vistas tradicional basado en XML, para entender cómo Jetpack Compose simplifica el desarrollo. Además, se explorará la estructura y fases de un proyecto Android utilizando Jetpack Compose, así como las herramientas más relevantes dentro de Android Studio para trabajar con esta tecnología, como el Live Preview, el soporte de Kotlin y las herramientas de depuración específicas.

Los objetivos que se persiguen en el siguiente tema son:

- Durante el desarrollo de este tema se deben conseguir los siguientes objetivos:
- Comprender los conceptos básicos de Jetpack Compose y su enfoque declarativo en comparación con el modelo basado en vistas tradicional.
- Conocer la estructura de un proyecto Android que utiliza Jetpack Compose y cómo se organizan las funciones composables.
- Identificar las fases de construcción de una interfaz gráfica con Jetpack Compose desde su diseño hasta su implementación.
- Comprender el uso de modificadores, layouts y el manejo de estado en el desarrollo de interfaces gráficas en Jetpack Compose.
- Familiarizarse con las herramientas de documentación y depuración en un proyecto Android con Jetpack Compose, incluidas las funciones de preview y el inspector de composición.

4.2. ¿Qué es JetPack Compose? Características principales

Jetpack Compose es el kit de herramientas moderno de UI de Google para crear interfaces de usuario nativas en Android. Está diseñado para reemplazar o complementar el sistema tradicional basado en vistas XML, ofreciendo una forma más eficiente y declarativa de construir interfaces. En lugar de definir la interfaz de usuario mediante archivos XML, en Jetpack Compose se utiliza código Kotlin directamente, lo que simplifica el desarrollo y mejora la flexibilidad.

Características principales

1. Declarativo: A diferencia del enfoque imperativo (como el tradicional XML + Views), Compose permite definir la UI declarativamente. Simplemente describes cómo debe verse la UI en un estado determinado, y Compose se encarga de actualizarla automáticamente cuando el estado cambia.

```
@Composable
fun Greeting(name: String) {
    Text(text = "Hello, $name!")
}
```

2. Basado en Kotlin: Jetpack Compose está completamente integrado con **Kotlin**, lo que permite aprovechar las características avanzadas del lenguaje, como funciones de orden superior, lambdas y extensiones. Esto da lugar a un código más conciso, expresivo y fácil de mantener.

3. Menos código: Compose reduce la cantidad de código necesario para crear interfaces de usuario, eliminando la necesidad de escribir múltiples capas de código como controladores de eventos de UI (como findViewById en XML).

4. UI reactiva: Jetpack Compose sigue un modelo de programación reactiva. La interfaz de usuario se "reconstruye" automáticamente cuando el estado cambia. Esto se gestiona a través del estado observable, que se puede declarar en el código Kotlin, y Compose se asegura de que la UI siempre esté sincronizada con el estado actual de los datos.

```
var count by remember { mutableStateOf(0) }

Column {
    Text(text = "Count: $count")
    Button(onClick = { count++ }) {
        Text(text = "Increase")
    }
}
```

5. Interoperabilidad con View (XML): El diseño basado en componentes facilita la reutilización y organización de código. Cada función marcada con @Composable es una unidad modular de la UI que se puede reutilizar en diferentes partes de la aplicación.

6. Modularidad: El diseño basado en componentes facilita la reutilización y organización de código. Cada función marcada con @Composable es una unidad modular de la UI que se puede reutilizar en diferentes partes de la aplicación.

7. Soporte para Material Design: Compose tiene soporte nativo para **Material Design**, lo que facilita la creación de aplicaciones con interfaces atractivas y consistentes utilizando componentes como botones, tarjetas y entradas de texto. Además, es fácil personalizar los componentes para adaptarlos a diferentes estilos.

```
@Composable
fun MyMaterialButton() {
    Button(onClick = { /* Acción */ }) {
        Text("Material Button")
    }
}
```

8. Animaciones simplificadas: Compose proporciona APIs fáciles de usar para animaciones que permiten crear transiciones, animaciones de propiedad, gestos, etc., de forma declarativa y sin código excesivamente complejo.

```
val alpha by animateFloatAsState(targetValue = if (visible) 1f else 0f)
Box(Modifier.alpha(alpha))
```

9. Soporte multiplataforma: Aunque Jetpack Compose está diseñado principalmente para Android, con el proyecto **Kotlin Multiplatform**, puedes usar Compose para crear interfaces de usuario en diferentes plataformas, como escritorio y web.

10. Fácil de probar: Compose facilita las pruebas de UI, permitiendo a los desarrolladores probar las funciones composables de forma aislada. Puedes escribir pruebas de unidad o de interfaz de usuario con herramientas de prueba como **Espresso** o **Jetpack Compose Testing**.

11. Live Previews: Android Studio permite previsualizar las composables en tiempo real sin necesidad de compilar y ejecutar la aplicación. Esto hace que el flujo de trabajo sea más rápido y eficiente, ya que puedes ver cómo se verá la UI mientras la desarrollas.

12. Integración fluida con Android Studio: Compose está completamente integrado con **Android Studio**, ofreciendo herramientas como Live Previews, compatibilidad con el inspector de composición y sugerencias de código específicas de Compose, lo que facilita mucho el desarrollo.

Ventajas de JetPack Compose:

- Desarrollo más rápido: La programación declarativa y el menor uso de boilerplate permiten a los desarrolladores construir pantallas de manera más rápida y concisa.
- Menos errores: Al trabajar directamente en Kotlin y con menos código repetitivo, el potencial de errores disminuye.

- Facilidad para realizar animaciones: Las animaciones se implementan fácilmente sin tener que manipular complicados listeners o transiciones manuales.
- Alineado con el ciclo de vida de Android: Las funciones @Composable se adaptan mejor a la gestión del ciclo de vida, lo que reduce la cantidad de errores relacionados con la UI y el ciclo de vida de las actividades o fragmentos.

En resumen, Jetpack Compose moderniza la creación de interfaces en Android, haciéndola más rápida, menos propensa a errores y mejor adaptada a las necesidades actuales del desarrollo de aplicaciones móviles.

4.3. Vistas y modificadores

En **Jetpack Compose**, las vistas (llamadas "composables") y los modificadores son los componentes esenciales para construir interfaces de usuario. Las composables definen la estructura y el contenido de la UI, mientras que los modificadores permiten ajustar su apariencia y comportamiento. La combinación de ambos proporciona un flujo de trabajo poderoso, declarativo y flexible para el desarrollo de UI en Android

Vistas

En Jetpack Compose, las vistas se crean usando funciones anotadas con @Composable. Estas funciones describen la UI de manera declarativa. Una composable es una unidad modular y reutilizable que puede representar una parte de la interfaz gráfica. Las composables pueden ser simples, como mostrar texto, o complejas, como crear un diseño con múltiples elementos visuales.

Ejemplos de vistas composables:

Text: Muestra un texto en pantalla.

```
@Composable
fun Greeting(name: String) {
    Text(text = "Hello, $name!")
}
```

Button: Muestra un botón que puede manejar eventos como clics.

```
@Composable
fun MyButton() {
    Button(onClick = { /* Acción cuando se hace clic */ }) {
        Text("Click me")
    }
}
```

Row: Alinea elementos en una fila horizontal.

```
@Composable
fun MyRow() {
    Row {
        Text("Elemento 1")
        Text("Elemento 2")
    }
}
```

Column: Alinea elementos en una columna vertical.

```
@Composable
fun MyColumn() {
    Column {
        Text("Elemento 1")
        Text("Elemento 2")
    }
}
```

Image: Muestra una imagen.

```
@Composable
fun MyImage() {
    Image(painter = painterResource(id = R.drawable.my_image),
contentDescription = null)
}
```

Las composables también pueden ser anidadas, y puedes crear funciones @Composable personalizadas para organizar mejor el código.

Modificadores

Los **modificadores** en Jetpack Compose son una herramienta clave que permiten alterar o decorar las vistas composables, añadiendo funcionalidades como el tamaño, padding, alineación, clics, colores, bordes, etc. Los modificadores se aplican mediante el patrón de encadenamiento, lo que permite componer varios ajustes en una sola línea de código.

Los modificadores se usan con el método Modifier y se pueden aplicar en cualquier composable.

Ejemplo básico de modificadores:

```
@Composable
fun MyText() {
    Text(
        text = "Hello, Jetpack Compose!",
        modifier = Modifier
            .padding(16.dp) // Aplica un padding de 16dp
            .background(Color.Blue) // Fondo de color azul
            .border(2.dp, Color.Red) // Borde rojo de 2dp
            .clickable { /* Acción cuando se hace clic */ }
    )
}
```

Los modificadores en Compose son declarativos y cada uno retorna un nuevo Modifier. Cuando encadenas varios modificadores, Compose los aplica en el orden en que aparecen. Esto permite una gran flexibilidad al definir el comportamiento y la apariencia de los componentes visuales.

- **Reutilizables**: Puedes componer modificadores y aplicarlos en diferentes componentes, lo que te permite tener un control más granular sobre la apariencia y el comportamiento de la UI.

- **Declarativos**: Al ser declarativos, puedes ver directamente cómo el modificador afecta a la UI sin necesidad de cambiar otros atributos en archivos separados (como sucedía en el XML).
- **Orden de aplicación**: El orden en el que aplicas los modificadores importa, ya que se procesan secuencialmente. Por ejemplo, aplicar primero padding() y luego background() tiene un efecto visual diferente que aplicarlos en orden inverso.

Modificadores comunes

1. padding(): Añade espacio alrededor de un composable.

```
Modifier.padding(8.dp)
```

2. background(): Establece un fondo para el composable.

```
Modifier.background(Color.Gray)
```

3. fillMaxSize(): Hace que el composable llene todo el espacio disponible tanto en ancho como en alto.

```
Modifier.fillMaxSize()
```

4. fillMaxWidth() y fillMaxHeight(): Hacen que el composable ocupe el ancho o alto máximo disponible.

```
Modifier.fillMaxWidth()
```

5. wrapContentSize(): Ajusta el tamaño del composable a su contenido.

```
Modifier.wrapContentSize()
```

6. clickable(): Permite que el composable responda a eventos de clic.

```
Modifier.clickable { /* Acción al hacer clic */ }
```

7. border(): Añade un borde al composable.

```
Modifier.border(2.dp, Color.Black)
```

8. size(): Establece un tamaño fijo para el composable.

```
Modifier.size(100.dp)
```

9. align(): Alinea el composable dentro de un contenedor (como Row, Column o Box).

```
Modifier.align(Alignment.Center)
```

10. clip(): Recorta el composable a una forma específica (por ejemplo, redondeada).

```
Modifier.clip(RoundedCornerShape(8.dp))
```

11. offset(): Desplaza el composable en los ejes X e Y.

```
Modifier.offset(x = 10.dp, y = 10.dp)
```

Ejemplo completo con varios modificadores:

```
@Composable
fun ModifiedBox() {
    Box(
        modifier = Modifier
            .size(150.dp)              // Tamaño fijo
            .background(Color.Green)    // Fondo verde
            .padding(16.dp)             // Padding de 16dp
            .border(2.dp, Color.Blue)   // Borde azul
            .clickable { /* Acción al hacer clic */ } // Hacer clic
    ) {
        Text("Contenido",  Modifier.align(Alignment.Center))  //  Alineado
centrado
    }
}
```

Ejemplo de Composable con varios modificadores y vistas

Este ejemplo muestra cómo combinar composables básicos como Row, Column y Text con modificadores (Modifier) para crear una interfaz gráfica simple, con estilos y organización flexible. El código es una función composable en **Jetpack Compose** que genera una interfaz de usuario simple para mostrar un perfil de usuario con un nombre y un correo electrónico.

```
@Composable
fun UserProfile(name: String, email: String) {
    Row(
```

```
        modifier = Modifier
            .padding(16.dp)
            .background(Color.LightGray)
            .border(1.dp, Color.Black)
            .fillMaxWidth()
    ) {
        Column(
            modifier = Modifier.padding(8.dp)
        ) {
            Text(text = name, fontSize = 20.sp, fontWeight =
FontWeight.Bold)
            Text(text = email, fontSize = 16.sp)
        }
    }
}
```

La función UserProfile toma dos parámetros:

- **name**: El nombre del usuario (tipo String).
- **email**: El correo electrónico del usuario (tipo String).

El propósito de esta función es generar una fila (Row) que contenga dos líneas de texto (el nombre y el correo electrónico), con algunos estilos aplicados para mejorar la apariencia de los elementos. La función Row organiza los elementos hijos en una disposición horizontal. En este caso, dentro de la fila, agregaremos una Column que contendrá dos textos: el nombre y el correo electrónico del usuario. Las Column en Compose organizan los elementos hijos de manera **vertical** (uno debajo del otro). En este caso, la Column contiene dos elementos Text: uno para el nombre y otro para el correo electrónico.

El resultado visual vendrá dado por la siguiente disposición de los elementos:

- La función crea un **rectángulo** con un fondo gris claro, un borde negro, y 16dp de espacio interno (padding) entre el borde y el contenido.
- Dentro de este rectángulo, los elementos están organizados en una fila horizontal (Row), aunque en este caso hay solo una Column.
- Dentro de la Column, los elementos se organizan de forma **vertical**:

- El nombre del usuario en texto más grande y en negrita (20sp, Bold).
- El correo electrónico del usuario en texto un poco más pequeño (16sp).

4.4. Layouts

En **Jetpack Compose**, los **layouts** (disposiciones o diseños) son elementos fundamentales que organizan las vistas o composables dentro de la interfaz de usuario. A diferencia de la antigua estructura XML en Android, en Compose los layouts son funciones composables que describen cómo deben disponerse los elementos en la pantalla de manera declarativa.

Un **layout** es responsable de posicionar y medir los elementos dentro de un contenedor. En Jetpack Compose, hay layouts predefinidos, como Row, Column y Box, pero también puedes crear tus propios layouts personalizados si es necesario.
Los layouts en Compose son extremadamente flexibles y declarativos, lo que significa que puedes definir cómo quieres que se comporten los elementos con solo describirlo en el código. Además, los layouts funcionan en conjunto con **modificadores** para aplicar estilos, espacios y comportamiento a los elementos.

Tipos principales de layouts en Jetpack Compose

1. Row: El layout Row organiza los elementos **horizontalmente** en una fila. Los elementos hijos dentro de un Row se colocan uno al lado del otro de izquierda a derecha (o de derecha a izquierda dependiendo del contexto de idioma).

Modificadores aplicables a Row:

- fillMaxWidth(): Hace que la fila ocupe todo el ancho disponible.
- horizontalArrangement: Controla cómo se distribuyen los elementos horizontalmente dentro de la fila.
- verticalAlignment: Alinea los elementos verticalmente dentro de la fila.

```
@Composable
fun MyRowExample() {
    Row(
        modifier = Modifier.fillMaxWidth(),
        horizontalArrangement = Arrangement.SpaceBetween
    ) {
        Text("Elemento 1")
        Text("Elemento 2")
        Text("Elemento 3")
    }
}
```

2. Column: El layout Column organiza los elementos verticalmente en una columna. Los elementos hijos se apilan uno encima del otro.

Modificadores aplicables a Column:

- fillMaxHeight(): Hace que la columna ocupe todo el alto disponible.
- verticalArrangement: Controla la distribución vertical de los elementos.
- horizontalAlignment: Alinea los elementos horizontalmente dentro de la columna.

```
@Composable
fun MyColumnExample() {
    Column(
        modifier = Modifier.fillMaxHeight(),
        verticalArrangement = Arrangement.SpaceBetween
    ) {
        Text("Elemento 1")
        Text("Elemento 2")
        Text("Elemento 3")
    }
}
```

3. Box: El layout Box es similar al **FrameLayout** en el sistema de vistas tradicional de Android. Superpone los elementos uno encima de otro. Es útil cuando deseas apilar varios elementos o crear interfaces más complejas con solapamientos.

Modificadores aplicables a Box:

- align(): Permite alinear los elementos dentro del Box. Ejemplo: Alignment.TopCenter, Alignment.BottomEnd.
- contentAlignment: Alinea el contenido dentro del Box de manera global.

```
@Composable
fun MyBoxExample() {
    Box(
        modifier = Modifier.size(100.dp)
    ) {
        Text("Capa Inferior")
        Text(
            "Capa Superior",
            modifier = Modifier.align(Alignment.BottomEnd)
        )
    }
}
```

4. LazyColumn y LazyRow: Estos son layouts especiales para listas grandes de elementos que se cargan de manera eficiente. Son las alternativas modernas a RecyclerView en el sistema tradicional de vistas de Android. La ventaja es que solo renderizan los elementos que están en pantalla, lo que mejora el rendimiento.

- **LazyColumn**: Diseña elementos de forma vertical (como una lista tradicional).
- **LazyRow**: Diseña elementos de forma horizontal.

```
@Composable
fun LazyColumnExample() {
    LazyColumn(
        modifier = Modifier.fillMaxSize(),
        contentPadding = PaddingValues(16.dp)
    ) {
        items(100) { index ->
            Text("Elemento #$index")
```

```
        }
    }
}
```

5. ConstraintLayout: Compose también tiene soporte para ConstraintLayout, que proporciona una disposición similar al ConstraintLayout del sistema de vistas clásico, pero adaptada a la programación declarativa. Con ConstraintLayout puedes crear layouts complejos donde los elementos tienen relaciones espaciales entre sí, como restricciones de márgenes y alineaciones con respecto a otros elementos o al contenedor principal.

```
@Composable
fun MyConstraintLayout() {
    ConstraintLayout(
        modifier = Modifier.fillMaxSize()
    ) {
        val (text1, text2) = createRefs()

        Text(
            "Texto 1",
            Modifier.constrainAs(text1) {
                top.linkTo(parent.top, margin = 16.dp)
                start.linkTo(parent.start)
            }
        )

        Text(
            "Texto 2",
            Modifier.constrainAs(text2) {
                top.linkTo(text1.bottom, margin = 8.dp)
                start.linkTo(text1.end)
            }
        )
    }
}
```

6. Personalización de Layouts: Jetpack Compose te permite crear tus propios layouts personalizados cuando los layouts predefinidos no son suficientes para tu caso de uso. Para hacer esto, puedes usar la función Layout que te proporciona control total sobre cómo se deben medir y posicionar los elementos hijos. Con Layout, puedes medir y colocar los composables hijos en cualquier posición que desees, dándote flexibilidad total para diseñar interfaces complejas.

```
@Composable
fun CustomLayout(
    modifier: Modifier = Modifier,
    content: @Composable () -> Unit
) {
    Layout(
        content = content,
        modifier = modifier
    ) { measurables, constraints ->
        // Medir los hijos (composables internos)
        val placeables = measurables.map { measurable ->
            measurable.measure(constraints)
        }

        // Calcular el tamaño del layout
        val width = constraints.maxWidth
        val height = constraints.maxHeight

        // Disponer los elementos hijos
        layout(width, height) {
            placeables.forEach { placeable ->
                // Posicionar cada hijo
                placeable.placeRelative(0, 0) // Posiciona todos en la
esquina superior izquierda
            }
        }
    }
}
```

Gestión del Espacio en Layouts

1. Modifier.padding(): El modificador padding añade espacio alrededor de un composable. Puede aplicarse a cualquier composable, y define un margen interno que separa el contenido de los bordes del composable.

2. Modifier.fillMaxWidth() y Modifier.fillMaxHeight(): Estos modificadores se utilizan para hacer que un composable ocupe todo el ancho o alto disponible dentro de su contenedor.

3. Modifier.size(): El modificador size() define un tamaño fijo para un composable, tanto en ancho como en alto.
4. Modifier.wrapContentSize(): Este modificador ajusta el tamaño del composable al tamaño de su contenido.

Manejo de estado en los layouts

Una característica clave de Jetpack Compose es su enfoque reactivo y declarativo. Cuando el estado cambia, la UI se vuelve a componer automáticamente para reflejar el nuevo estado. Esto es importante cuando trabajas con layouts dinámicos, ya que los datos o configuraciones de los elementos visuales pueden cambiar y la UI debe reaccionar.

4.5. Estados

El estado en Jetpack Compose se refiere a cualquier dato que controla o influye en la representación de la UI. La UI es una función de este estado, y cuando el estado cambia, Compose vuelve a ejecutar la función @Composable que depende de ese estado, actualizando así la UI.

El estado puede provenir de varias fuentes, como los datos de entrada del usuario, respuestas de una API o cualquier otro tipo de información que esté sujeta a cambios durante el ciclo de vida de la aplicación.

1. Declaración del estado: El estado en Jetpack Compose generalmente se declara con la función remember y el tipo mutable MutableState. Esta función garantiza que el estado se mantenga mientras el composable esté en memoria y solo se restablezca si se recompone.

2. Reactividad: Cuando el valor de un State cambia, cualquier composable que dependa de ese valor se "recompondrá" automáticamente. Esto significa que no necesitas preocuparte por actualizar la UI manualmente; Compose se encarga de esto por ti.

3. Inmutabilidad:Aunque el estado mutable se puede cambiar, el enfoque declarativo de Compose implica que los composables deben ser puros, es decir, que su salida solo dependa de su entrada (el estado). No deberían modificar el estado directamente, sino que deberían reaccionar a los cambios en él.

Ejemplo básico de uso de estado:

```
@Composable
fun Counter() {
    var count by remember { mutableStateOf(0) }

    Column {
        Text("Contador: $count")
        Button(onClick = { count++ }) {
            Text("Incrementar")
        }
    }
}
```

Explicación del ejemplo:

1. Estado: Se define una variable count con remember { mutableStateOf(0) }. Esto establece que el valor inicial del estado es 0. La función remember se utiliza para que el estado sobreviva a recomposiciones, pero se restablecerá si el composable deja de existir en la jerarquía de UI.

2. Reactividad: Cada vez que el botón es presionado, count++ aumenta el valor de count, lo que desencadena una recomposición del composable Text que muestra el nuevo valor del contador.

3. Modificación del estado: La acción del botón modifica el estado, pero no se hace ninguna llamada directa a la UI para actualizarla. Compose detecta el cambio de estado y vuelve a componer el componente afectado automáticamente.

Tipos de Estado en Compose

1. Estado local: El estado que se maneja dentro de un composable y que se "recuerda" solo mientras ese composable sigue existiendo. Este tipo de estado es útil para manejar pequeñas interacciones, como el valor de un contador, texto de entrada, etc.

Ejemplo con estado local usando remember y mutableStateOf:

```
@Composable
fun SimpleSwitch() {
    var isChecked by remember { mutableStateOf(false) }

    Switch(
        checked = isChecked,
        onCheckedChange = { isChecked = it }
    )
}
```

2. Estado compartido: A veces, varios composables deben compartir el mismo estado. Para esto, puedes levantar el estado a un nivel más alto en la jerarquía de composables y pasarlo como argumento a los composables hijos. Esto sigue el principio de "propagar hacia abajo, levantar hacia arriba" (lifting state up).

Ejemplo de estado compartido entre varios composables:

```
@Composable
fun ParentComposable() {
```

```
    var text by remember { mutableStateOf("Hola") }

    Column {
        TextField(
            value = text,
            onValueChange = { newText -> text = newText }
        )
        Text("El texto es: $text")
    }
}
```

3. Estado persistente: A veces es necesario preservar el estado más allá del ciclo de vida de un composable (por ejemplo, cuando cambias de una pantalla a otra y luego vuelves). Para estos casos, Jetpack Compose proporciona rememberSaveable, que recuerda el estado incluso si el composable se destruye temporalmente (como en rotaciones de pantalla).

Ejemplo usando rememberSaveable:

```
@Composable
fun PersistentCounter() {
    var count by rememberSaveable { mutableStateOf(0) }

    Column {
        Text("Contador persistente: $count")
        Button(onClick = { count++ }) {
            Text("Incrementar")
        }
    }
}
```

Con rememberSaveable, el valor de count se conservará incluso si hay un cambio de configuración (como la rotación de la pantalla).

State Hoisting y Unidirectional Data Flow (Flujo de datos unidireccional)

Jetpack Compose sigue el principio de flujo de datos unidireccional. Esto significa que los datos fluyen en una sola dirección: desde los composables padres hacia los hijos.

Los composables hijos reciben datos y notifican los cambios de estado al padre, lo que asegura un control centralizado del estado.

Ejemplo del flujo de datos unidireccional:

```
@Composable
fun ParentComponent() {
    var count by remember { mutableStateOf(0) }

    Column {
        ChildDisplayCount(count)
        ChildIncrementButton { count++ }
    }
}

@Composable
fun ChildDisplayCount(count: Int) {
    Text("El contador es: $count")
}

@Composable
fun ChildIncrementButton(onIncrement: () -> Unit) {
    Button(onClick = onIncrement) {
        Text("Incrementar")
    }
}
```

En este ejemplo:

- El estado se almacena en el ParentComponent (count).
- El valor de count fluye hacia ChildDisplayCount.
- El evento de clic en ChildIncrementButton se notifica al padre para actualizar el estado.

Buenas prácticas para manejar el estado en Compose

1. Levanta el estado cuando sea necesario: El estado debería estar en el nivel más alto posible para ser compartido entre los composables que lo necesitan, pero no más arriba.

2. Usa remember solo cuando sea necesario: remember debería usarse para retener el estado mientras el composable sigue en el árbol de UI. Si el estado debe persistir a través de cambios de configuración, utiliza rememberSaveable.

3. Manten el estado local cuando sea posible: Si el estado no necesita ser compartido entre varios composables, manténlo local dentro del composable donde es utilizado.

4. Evita side-effects en los composables: Como los composables pueden recomponerse muchas veces, evita modificar el estado dentro de ellos de manera que se creen efectos secundarios no deseados.

4.6. Componentes

En Jetpack Compose, los términos vistas y componentes a menudo pueden confundirse debido a su uso en el sistema tradicional de Android (basado en vistas) y el nuevo enfoque declarativo de Compose. Sin embargo, en Jetpack Compose, no existe una distinción técnica entre "vistas" y "componentes" en el sentido en que se usaban en el antiguo sistema de vistas.

Vamos a ver algunos componentes importantes:

1. Text: El componente Text se utiliza para mostrar texto en la pantalla. Es una de las composables más comunes y sencillas en Jetpack Compose.

Atributos principales:

- text: El contenido de texto que se va a mostrar.
- fontSize, fontWeight, fontStyle: Se utilizan para estilizar el texto (tamaño de fuente, peso de fuente, estilo).
- color: Define el color del texto.
- modifier: Permite aplicar modificadores como padding, align, etc.

2. Button: El componente Button se usa para crear un botón que puede recibir clics o toques del usuario.

Atributos principales:

- onClick: Acción que se ejecuta cuando el botón es presionado.
- enabled: Habilita o deshabilita el botón.
- content: Los composables dentro del botón, que normalmente incluyen un Text para mostrar el contenido del botón.

3. Image: Image se utiliza para mostrar imágenes en pantalla. Puedes cargar imágenes desde recursos locales o URLs.

Atributos principales:

- painter: Define el origen de la imagen.
- contentDescription: Describe el contenido de la imagen, útil para accesibilidad.
- modifier: Permite aplicar modificaciones como tamaño, alineación, etc.

4. TextField es un componente de entrada de texto que permite al usuario escribir datos.

Atributos principales:

- value: El valor actual del campo de texto.
- onValueChange: Lambda que se ejecuta cuando el valor cambia.
- label: Texto de etiqueta que aparece cuando el campo está vacío.
- modifier: Permite aplicar modificaciones como el tamaño y el padding.

5. Card: El componente Card permite mostrar contenido con un fondo y sombra, dándole el estilo de una tarjeta.

Atributos principales:

- elevation: Define la sombra alrededor de la tarjeta.
- modifier: Permite modificar el estilo, como añadir padding o size.
- content: Define lo que contiene la tarjeta, puede ser texto, imágenes o incluso layouts.

6. Scaffold: es un componente que permite construir la estructura básica de una pantalla, como una barra superior, el cuerpo de la pantalla y una barra inferior.

Atributos principales:

- topBar: Permite incluir una barra superior o de acción.
- bottomBar: Permite incluir una barra inferior.
- floatingActionButton: Permite agregar un botón de acción flotante.
- content: Define el contenido principal de la pantalla.

7. Slider: El componente Slider es un componente que permite al usuario seleccionar un valor dentro de un rango mediante un control deslizante.

Atributos principales:

- value: El valor actual del control deslizante.
- onValueChange: Función que se ejecuta cuando el usuario cambia el valor.
- valueRange: Define el rango permitido del valor del slider.

A fondo

Framework Compose

https://developer.android.com/jetpack/androidx/releases/compose?hl=es-419
Documentación en línea. (2024)

Documentación oficial sobre Compose, conjunto de 7 Maven Group.

Material 3

https://m3.material.io/ Documentación en línea. (2024)

Documentación oficial sobre Material3, librería de estilos y diseño de Google para todas sus herramientas de desarrollo.

Compose Multiplataform

https://www.jetbrains.com/compose-multiplatform/ Documentación en línea. (2024)

Framework declarativo para compartir interfaces de usuario en múltiples plataformas. Basado en Kotlin Multiplatform y Jetpack Compose.

OpenCompose pro

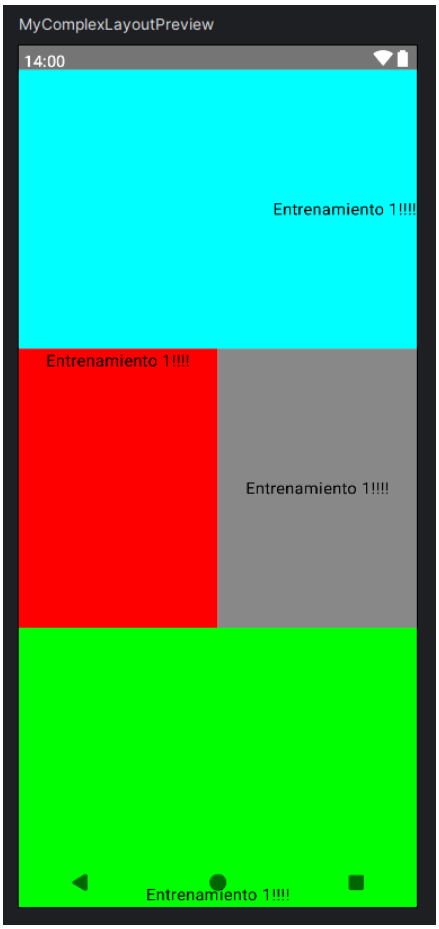
https://www.jetpackcompose.pro/ Documentación en línea. (2024)

La mayor base de datos de @Composables de habla hispana por y para la comunidad.

Entrenamientos

Entrenamiento 1

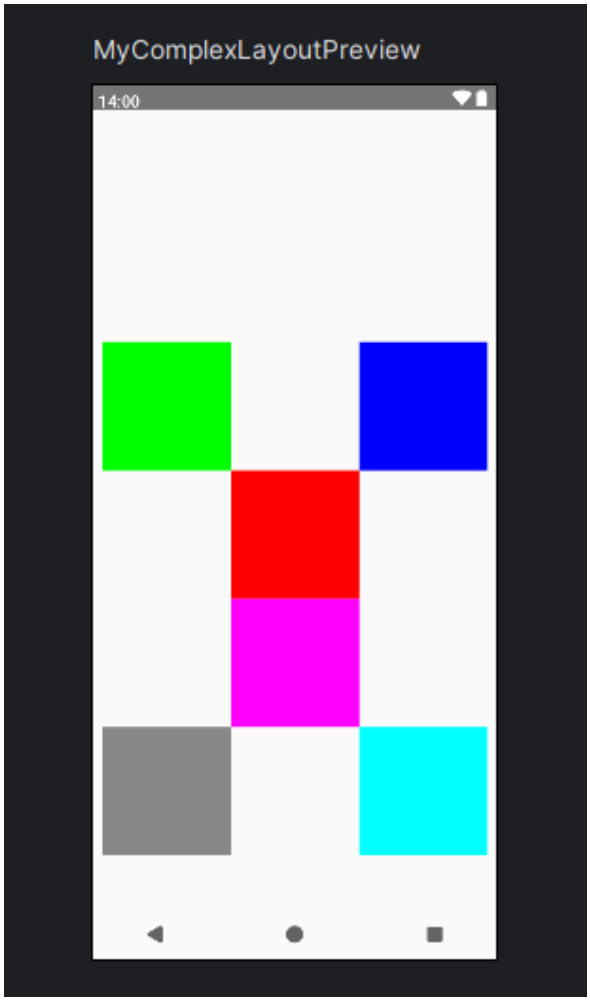
- Desarrollar una app Android Studio con Kotlin y JepPack Compose en el que se practique con los layoust: Box, columns y Rows. La Aplicación debe de tener este aspecto.



- Desarrollo paso a paso:
    - Crear una función Composable llamada MyComplexLayout.
    - Crear los elementos Column, Row, Box y Text necesarios.
    - Configurar los alineamientos.
- Solución: https://github.com/Anuar-UNIR/PMDM_2024-2025/tree/main/Tema%204/Entrenamiento_1

Entrenamiento 2

- Desarrollar una app Android Studio con Kotlin y JetPack Compose utilizando los ConstrainsLayouts. La Aplicación debe de tener este aspecto.



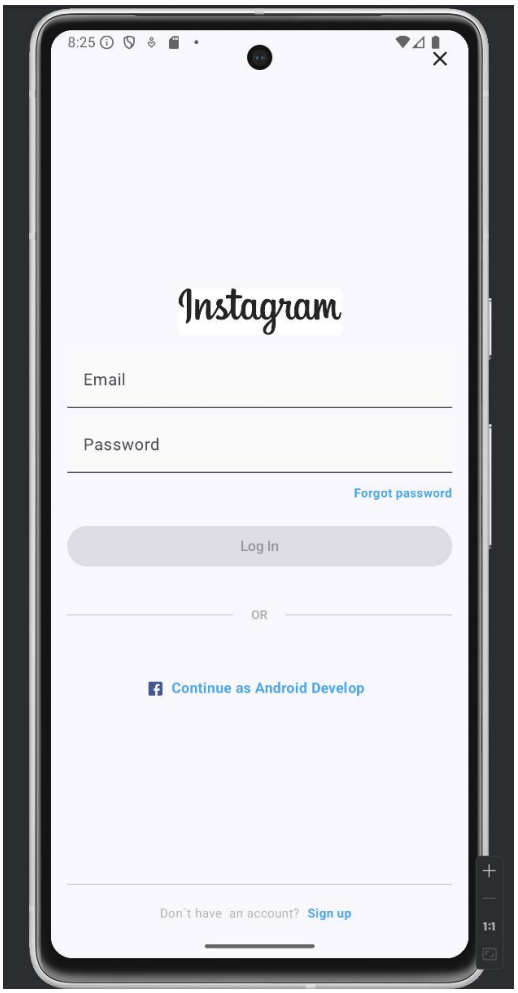
- Desarrollo paso a paso:
    - Crear una función Composable llamada MyConstraintLayout.
    - Crear los elementos Box.
    - Crear las referencias con createRefs()
    - Configurar las restricciones.
- Solución: https://github.com/Anuar-UNIR/PMDM_2024-2025/tree/main/Tema%204/Entrenamiento_2

Entrenamiento 3

- Desarrollar una App en Android usando Jet Pack Compose, que dispongo de una carrusel usando `HorizontalMultiBrowseCarousel`.
- Desarrollo paso a paso:
    - Crear proyecto Android usando Jet Pack Compose
    - Definir las funciones @Composables
    - Utilizar `HorizontalMultiBrowseCarousel` para la implementación del carrusel
- Solución: https://www.jetpackcompose.pro/carousel/horizontalmultibrowsecarousel/

Entrenamiento 4

- Desarrollar una app Android y JetPack Compose que tenga esta apariencia (solo la parte visual)



- Desarrollo paso a paso:
- Solución: https://github.com/Anuar-UNIR/PMDM_2024-2025/tree/main/Tema%204/Entrenamiento_4

Entrenamiento 5

- A partir del entrenamiento anterior, añadir la lógica para que el botón de log in se active cuando se cumpla que el email tiene el formato correcto y la contraseña tiene 8 o más caracteres.
- Desarrollo paso a paso: Se trata de un proyecto libre pero la app debería tener:
- Solución: https://github.com/Anuar-UNIR/PMDM_2024-2025/tree/main/Tema%204/Entrenamiento_5