Diseño gráfico con J. Compose

Indice general

El siguiente Indice es un resumen del Documentacion de Android Developer y debe servir como orientación para comprender a fondo Compose

- Descripción general
- Instructivo - ejemplo paso a paso-

  El contenido de estos documentos: (los enlaces no están actualizados y debéis usar las páginas anteriores para llegar al punto deseado)

  - Cómo pensar en Compose. Conceptos básicos
  - Cómo administrar estados
    - Descripción general
    - Dónde elevar el estado
    - Cómo guardar el estado de la IU
  - Ciclo de vida
  - Modificadores
  - Efectos secundarios
  - Fases
  - Capas de arquitectura
  - Rendimiento
    - Descripción general
    - Herramientas
      - Descripción general
      - Inspector de diseño
      - Rastreo
    - Prácticas recomendadas
  - Semántica
  - CompositionLocal
  - Entorno de desarrollo
    - Herramientas
      - Descripción general
      - Compatibilidad de Android Studio con Compose
      - Herramientas de desarrollador y diseñador de retransmisión
        - Descripción general
        - Instalar retransmisión
        - Configura tu proyecto de Android
        - Instructivo básico
          - Descripción general
          - Crea paquetes de IU en Figma
          - Convierte los diseños a código en Android Studio
          - Crea y propaga actualizaciones de diseño

Indice de los temas tratados en este documento:

https://developer.android.com/jetpack/compose/layouts?hl=es-419

Conceptos básicos

Las funciones que admiten composición son los componentes fundamentales de Compose. Una función de este tipo devuelve una **Unit**, dicho de otra forma no devuelve nada.
La función puede tomar algun argumento de entrada y dibuja el componente en pantalla.

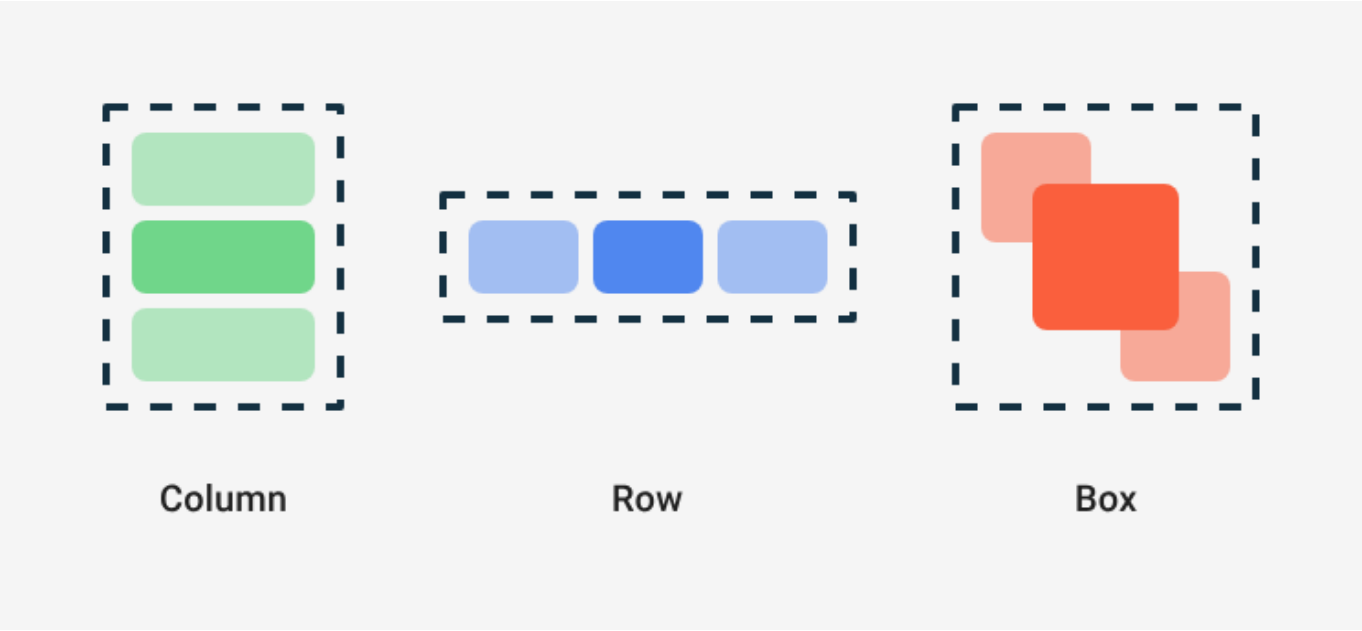
```
@Composable
fun ArtistCard(artist: Artist) {
    Row(verticalAlignment = Alignment.CenterVertically) {
        Image(/*...*/)
        Column {
            Text(artist.name)
            Text(artist.lastSeenOnline)
        }
    }
}
```

Que produce la siguiente salida:

Alfred Sisley

3 minutes ago

En el código anterior: Row, Column y Box son componentes contenedores y en el caso de **Box**, este permite colocar componentes apilados (diriamos en el eje Z), por ejemplo texto sobre imagenes.



Paradigma de la programación declarativa.

Seguid el enlace del título ...
Se centra en el **que hacer** en lugar de como hacerlo.
SQL y HTML son ejemplos de lengujes declarativos.

Contenido dinámico

Puesto que las funciones Componibles son kotlin, los componentes componibles se pueden crear *al vuelo*. En el siguinete ejemplo se crea un saludo para cada elemento de la lista.

```
@Composable
fun Greeting(names: List<String>) {
    for (name in names) {
        Text("Hello $name")
    }
}
```

Pueden contener sentencias condicionales (if) o bucles, etc.

Composición y recomposición

En un modelo de IU imperativo, para cambiar un widget, se debe llamar a un método set en el widget que cambie el estado interno.
En Compose, se vuelve a llamar a la función de componibilidad con datos nuevos. Eso provoca la **recomposición** de la función (los componentes que forman la función se vuelven a dibujarse, solamente si es necesario y con datos nuevos). El framework de Compose puede recomponer de forma inteligente solo los componentes que cambiaron.

Por ejemplo, veamos esta función de componibilidad que muestra un botón:

Continuar aqui

Estados

Descripción General

El estado de una app es cualquier valor que puede cambiar con el paso del tiempo. Esta es una definición muy amplia y abarca desde una base de datos de Room hasta una variable de una clase.

Compose es declarativo y, por lo tanto, la única manera de actualizarlo es llamar al mismo elemento que admite composición con argumentos nuevos. Estos argumentos son representaciones del estado de la IU. Cada vez que se actualiza un estado, se produce una recomposición.

En consecuencia, elementos, como TextField, no se actualizan automáticamente de la misma manera que en las vistas imperativas que se basan en XML.

```
@Composable
fun HelloContent() {
   Column(modifier = Modifier.padding(16.dp)) {
       Text(
           text = "Hello!",
           modifier = Modifier.padding(bottom = 8.dp),
           style = MaterialTheme.typography.h5
       )
       OutlinedTextField(
           value = "",
           onValueChange = { },
           label = { Text("Name") }
       )
   }
}
```

En este ejemplo no se observa cambio en la IU cuando se escribe el TextField.

```
Términos clave. **Composición**: descripción de la IU que compila Jetpack Compose
cuando ejecuta elementos que admiten composición.

**Composición inicial**: creación de una composición mediante la primera ejecución
de elementos que admiten composición.
```

```
**Recomposición**: nueva ejecución de los elementos que admiten composición para
actualizar la composición cuando los datos cambian.
```

Estado en elementos componibles

Las funciones de componibilidad pueden usar la API de **remember** para almacenar un objeto en la memoria. Un valor calculado por remember se almacena durante la *composición inicial*, y el valor almacenado se muestra durante la recomposición. Se puede usar remember para almacenar tanto objetos mutables como inmutables.

`mutableStateOf` se crea del interfaz `MutableState<T>`

```
interface MutableState<T> : State<T> {
    override var value: T
}
```

Cualquier cambio en `value` programa la recomposición de las funciones que admiten composición que lean value

MutableState puede declararse de estas tres formas equivalentes:

```
val mutableState = remember { mutableStateOf(default) }
var value by remember { mutableStateOf(default) }
val (value, setValue) = remember { mutableStateOf(default) }
```

La sintáxis de la delegacion `by` necesita los import:

```
import androidx.compose.runtime.getValue
import androidx.compose.runtime.setValue
```

Puedes usar el valor recordado como un parámetro para otros elementos que admiten composición o incluso como lógica en declaraciones para cambiar los elementos que se muestran. Por ejemplo, si no quieres mostrar el saludo cuando el nombre está vacío, usa el estado en una declaración if

```
Aunque remember te ayuda a retener el estado entre recomposiciones, **el estado no
se retiene entre cambios de configuración**. Para ello, debes usar
**rememberSaveable**. rememberSaveable almacena automáticamente cada valor que se
puede guardar en un Bundle. Para otros valores, puedes pasar un objeto Saver
personalizado.
```

Elevación de estado

TODO

Ciclo de vida de los elementos componibles

Modificadores de Compose

Lista completa de modificadores

Modelo de diseño. Árbol de componentes.

Los compoenntes se organizan jerarquicamente. Cuando se va a dibujar el componenente raíz realiza su medición y pide a los nodos hijos que se midan, de forma recursiva. Y en el nodo hoja se establece el tamaño que se pasa hacia arriba en el árbol: Ejemplo.

```
@Composable
fun SearchResult(...) {
  Row(...) {
    Image(...)
    Column(...) {
      Text(...)
      Text(..)
    }
  }
}
```

En el ejemplo anterior de SearchResult, el diseño del árbol de IU sigue este orden:

1. Se solicita que el nodo raíz Row realice la medición.
2. El nodo raíz Row le solicita a su primer elemento secundario, Image, que realice la medición.
3. Image es un nodo de hoja (es decir, no tiene elementos secundarios), por lo que informa un tamaño y devuelve instrucciones sobre la posición.
4. El nodo raíz Row le solicita al segundo elemento secundario, Column, que realice la medición.
5. El nodo Column le solicita al primer elemento secundario Text que realice la medición.
6. El primer nodo Text es un nodo de hoja, por lo que informa un tamaño y devuelve instrucciones sobre la posición.
7. El nodo Column le solicita a su segundo elemento secundario Text que realice la medición.
8. El segundo nodo Text es un nodo de hoja, por lo que informa un tamaño y devuelve instrucciones sobre la posición.
9. Ahora que el nodo Column realizó la medición de sus elementos secundarios, estableció sus tamaños y sus posiciones, puede determinar su propio tamaño y posición. 10 Ahora que el nodo raíz Row realizó la medición de sus elementos secundarios, estableció sus tamaños y sus posiciones, puede determinar su propio tamaño y posición.

Uso de modificadores en el diseño

Los `Modifier` que aparecen como argumentos en funciones Composable permiten personalizar el diseño en aspectos como:

- size: tamaño del componente

- border: bordes
- onClick: callback para pulsaciones
- padding: espacio entre el componente y el borde
- etc

Restricciones en el layout

Se puede utilizar BoxWithConstraints para un layout personalizado

```
@Composable
fun WithConstraintsComposable() {
    BoxWithConstraints {
        Text("My minHeight is $minHeight while my maxWidth is $maxWidth")
    }
}
```

Diseños basados en ranuras.

Con Compose se incluye `Material Design` con nuevos componenetes como Drawerm , TopAppBar, FloatingActionButton y otros. Compose con `Material Design` utiliza el patrón de diseño **basado en ranuras**

Las ranuras dejan un espacio vacío en la IU para que el desarrollador lo complete como quiera, una especie de armazón donde se pueden colocar componentes.

Por ejemplo, estas son las ranuras que puedes personalizar en una TopAppBar:

Los elementos componibles suelen adoptar una expresión lambda **content** componible `(content: @Composable () -> Unit)``. Las APIs con ranuras exponen varios parámetros de content para usos específicos. Por ejemplo, TopAppBar te permite proporcionar el contenido para **title**, **navigationIcon** y **actions**.

Por ejemplo, **Scaffold** te permite implementar una IU con la estructura básica de diseño de Material Design. Scaffold proporciona ranuras para los componentes de Material de nivel superior más comunes, como **TopAppBar**, **BottomAppBar**, **FloatingActionButton** y **Drawer**.

```
@Composable
fun HomeScreen(/*...*/) {
```

```
    Scaffold(
        drawerContent = { /*...*/ },
        topBar = { /*...*/ },
        content = { /*...*/ }
    )
}
```