
UD4.3 – Componentes

Jetpack Compose

2º CFGS
Desarrollo de Aplicaciones Multiplataforma

2023-24

1.- Layout

El **layout** es el diseño gráfico o disposición de los elementos.

Es la manera en la que se colocan los elementos dentro de una interfaz.

En Jetpack Compose existen una serie de componentes que permiten agrupar a otros componentes y ayudan al diseño del layout de las diferentes pantallas de la aplicación.

Anteriormente ya se han usado algunos componentes de layout como Surface, Column y Row.

1.- Layout

En Jetpack Compose hay muchos componentes que ayudan a diseñar el layout de la aplicación:

- Source
- Card
- Box
- BoxWithConstraints
- Column
- Row
- LazyRow y LazyColumn
- LazyVerticalGrid y LazyHorizontalGrid
- HorizontalPager y VerticalPager
- LazyVerticalStaggeredGrid y LazyHorizontalStaggeredGrid
- ConstraintLayout
- Scaffold
- TopAppBar
- BottomBar
- NavigationDrawer
- NavigationRail
- Layout
- DropDownMenu
- BottomSheets y SideSheets
- ...

2.- Componente Surface

El componente **Surface** es el eje central en el que se basa la filosofía de **Material Design**.

Surface es un contenedor con estilo predefinido para seguir el tema por defecto de Material Design.

Se puede cambiar el estilo por defecto ya que Surface tiene parámetros para definir la forma, la elevación, el borde y colores para el contenido y el continente.

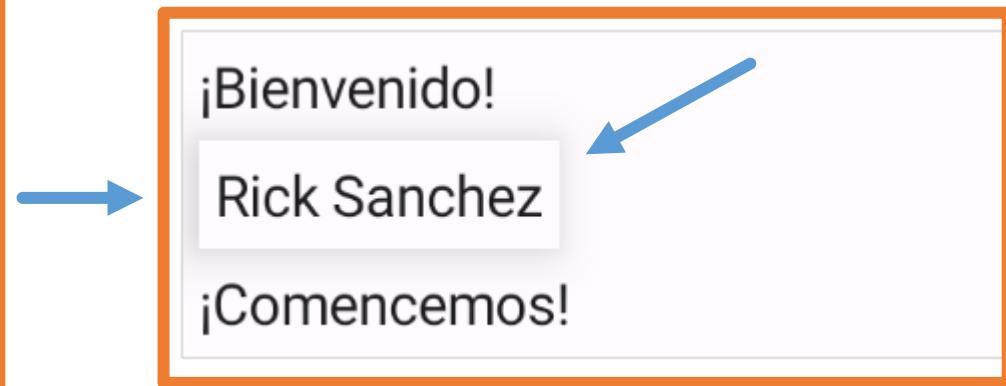
Se recomienda el uso de Surface como elemento padre de toda la interfaz.

```
setContent {  
    UD4PruebasTheme {  
        // A surface container using the 'background' color from the theme  
        Surface(←  
            modifier = Modifier.fillMaxWidth(),  
            color = MaterialTheme.colorScheme.background  
        ) {  
            Content()  
        }  
    }  
}
```

2.- Componente Surface

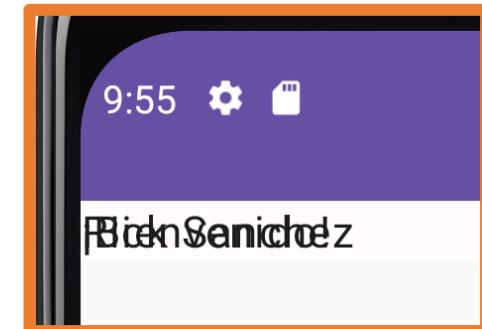
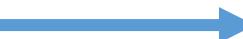
Se puede usar el componente **Surface** tantas veces como se desee cada vez que se requiera tener una superficie que se diferencie del resto de la interfaz.

```
Surface(  
    modifier = Modifier.fillMaxWidth()  
) {  
    Column(modifier = Modifier.padding(4.dp)) { this: ColumnScope  
        Text(text = "¡Bienvenido!")  
        Spacer(modifier = Modifier.height(4.dp))  
        Surface(shadowElevation = 10.dp) {  
            Text(text = "Rick Sanchez", modifier = Modifier.padding(4.dp))  
        }  
        Spacer(modifier = Modifier.height(4.dp))  
        Text(text = "¡Comencemos!")  
    }  
}
```



El componente Surface por si solo no sirve para organizar los elementos de la interfaz.

```
Surface(  
    modifier = Modifier.fillMaxWidth(),  
    color = MaterialTheme.colorScheme.background  
) {  
    Text( text: "¡Bienvenido!")  
    Text( text: "Rick Sanchez")  
}
```

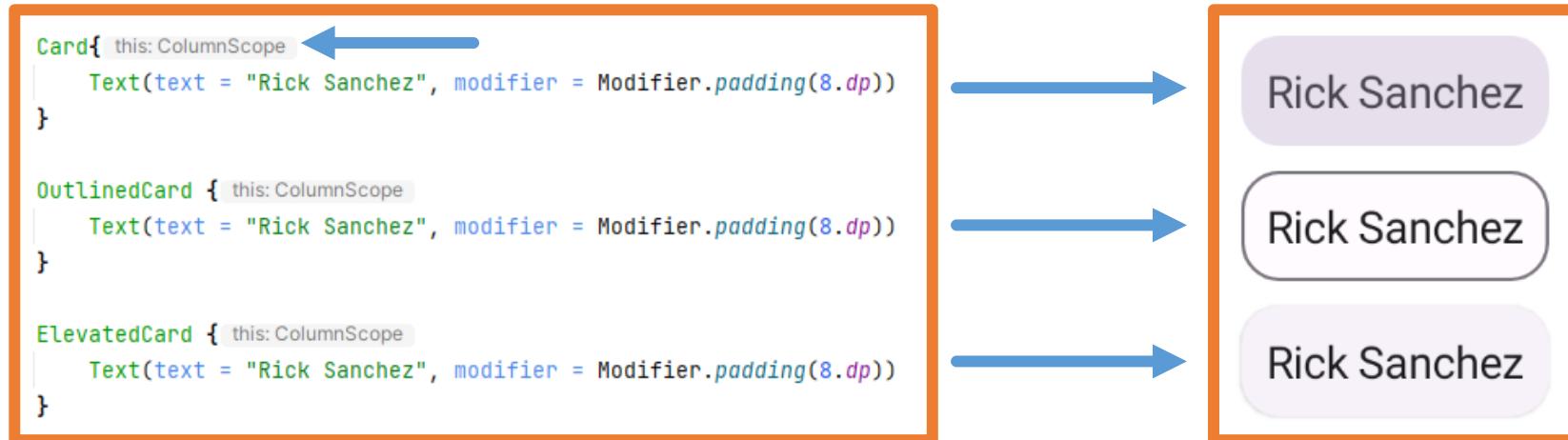


3.- Componente Card

Las **Card** son componentes Surface con un estilo predefinido (que se puede cambiar con los parámetros) para tener el aspecto de una tarjeta con borde, elevación y sombra.

Existen tres tipos: **Card**, **ElevatedCard** y **OutlinedCard**.

Dentro de una **Card** los elementos se organizan en **columna** (ColumnScope).



4.- Componente Box

El componente **Box** permite ubicar varios elementos en el mismo lugar uno encima de otro.

Un posible uso podría ser mostrar un contenido u otro según se pulse un botón.

```
var visible by rememberSaveable { mutableStateOf( value: true) }

Box { this: BoxScope
    if (visible) Text( text: "Rick Sanchez") else Text( text: "Morty Smith")
}

Button(onClick = { visible = !visible }) { this: RowScope
    Text( text: "Cambio")
}
```

4.- Componente BoxWithConstraints

El componente **BoxWithConstraints** es una versión especial de Box en la que se transmiten al contenido las dimensiones del componente para poder usarlas si fuera necesario.

```
BoxWithConstraints { this: BoxWithConstraintsScope  
    //var constraints = this.constraints  
    val constraints = constraints  
    Column { this: ColumnScope  
        Text( text: "Ancho mínimo: ${constraints.minWidth}")  
        Text( text: "Ancho máximo: ${constraints.maxWidth}")  
        Text( text: "Ancho mínimo: ${constraints.minHeight}")  
        Text( text: "Ancho máximo: ${constraints.maxHeight}")  
    }  
}
```



```
Ancho mínimo: 0  
Ancho máximo: 1080  
Ancho mínimo: 0  
Ancho máximo: 1977
```

5.- Componentes Column y Row

Los componentes **Column** y **Row** ya se han utilizado anteriormente.

Para alinear el contenido verticalmente y horizontalmente dentro de estos componentes se utilizan las propiedades:

- **Arrangement**: eje principal (columnas: vertical, filas: horizontal).
- **Alignment**: eje secundario (columnas: horizontal, filas: vertical).

```
Column(  
    verticalArrangement = Arrangement.Center,  
    horizontalAlignment = Alignment.CenterHorizontally  
) { this: ColumnScope  
  
}
```

```
Row(  
    horizontalArrangement = Arrangement.Center,  
    verticalAlignment = Alignment.CenterVertically  
) { this: RowScope  
  
}
```

5.- Componentes Column y Row

Arrangement

Center: alineación centrada.

SpaceEvenly: el mismo espacio entre los elementos y en la parte superior e inferior

SpaceBetween: el mismo espacio entre los elementos pero no deja en la parte superior ni inferior.

SpaceArround: el mismo espacio arriba y debajo de cada elemento (entre objetos habrá el doble de espacio que arriba y abajo).

spacedBy: se indica el espacio entre elementos en dp.

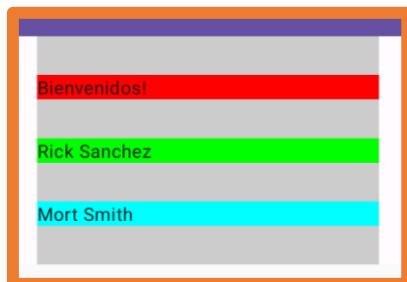
Top: alineación a la parte superior → solo Column.

Bottom : alineación a la parte inferior → solo Column.

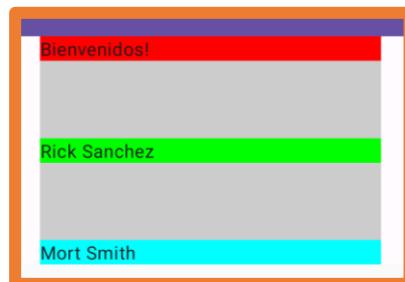
End: alineación al principio de la línea según sentido de lectura → solo Row.

Start: alineación al final de la línea según sentido de lectura → solo Row.

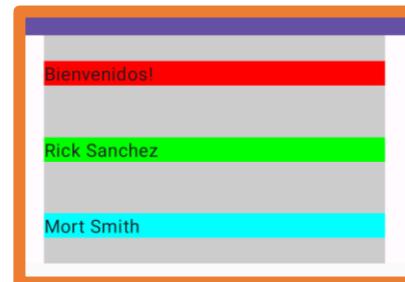
SpaceEvenly



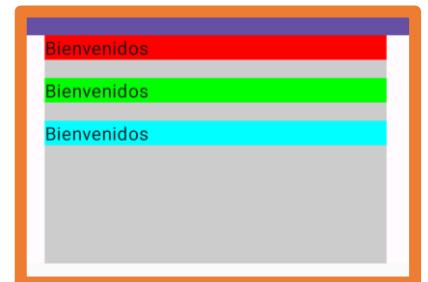
SpaceBetween



SpaceArround



spacedBy(16.dp)

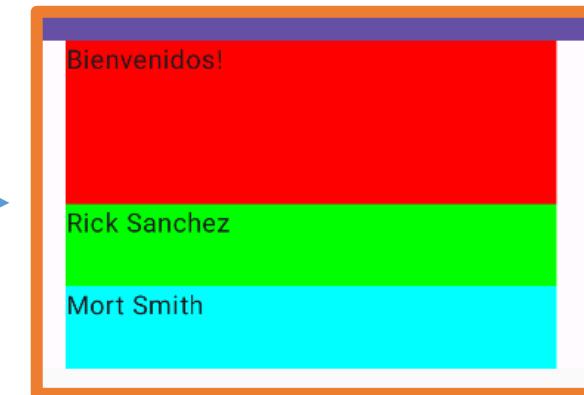


5.- Componentes Column y Row

Peso

Se puede indicar el peso que debe tener un componente dentro de otro con el modifier **weight()**.

```
Column(  
    modifier = Modifier.size(width = 300.dp, height = 200.dp)  
) { this: ColumnScope  
    Text(  
        text = "Bienvenidos!",  
        modifier = Modifier  
            .fillMaxWidth()  
            .weight(2f)  
            .background(Color.Red)  
    )  
    Text(  
        text = "Rick Sanchez",  
        modifier = Modifier  
            .fillMaxWidth()  
            .weight(1f)  
            .background(Color.Green)  
    )  
    Text(  
        text = "Mort Smith",  
        modifier = Modifier  
            .fillMaxWidth()  
            .weight(1f)  
            .background(Color.Cyan)  
    )  
}
```



5.- Componentes Column y Row

Alignment: eje secundario (columnas: horizontal, filas: vertical).

Start: alineación al principio de la línea según sentido de lectura → solo Column.

End: alineación al final de la línea según sentido de lectura → solo Column.

CenterHorizontally: alineación centrada → solo Column.

Top: alineación a la parte superior → solo Row.

Bottom: alineación a la parte inferior → solo Row.

CenterVertically: alineación centrada → solo Row.

6.- Scroll

Como se ha podido observar hasta el momento, cuando hay demasiados componentes en la pantalla es posible que algunos queden fuera del alcance del usuario (fuera de la parte visible de la pantalla).

Cuando se explicó el parámetro **modifier** se indicó que si el contenido de un componente no cabe en la pantalla se pueden usar los modificadores **verticalScroll** y **horizontalScroll** para deslizar y poder alcanzar todo el contenido.

Este comportamiento no es el idóneo ya que supone que se debe cargar todo el contenido del componente en memoria RAM se vaya a visualizar o no.

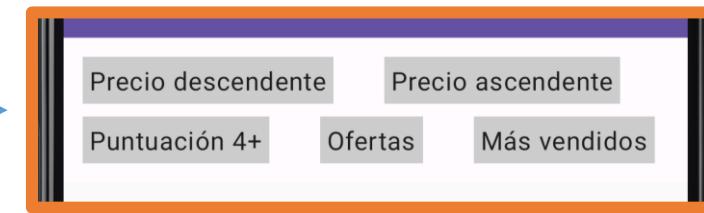
Para solucionar esto están los componentes **Lazy** que se estudiarán a continuación.

Los componentes **Lazy** por defecto tienen un parámetro que por defecto crea un estado con **rememberSaveable** para gestionar automáticamente el estado del scroll en esos componentes.

7.- Componentes Flow

Los componentes **FlowColumn** y **FlowRow** son similares a Column y Row pero cuando los elementos no caben en una única columna/fila entonces se pasarán a la siguiente columna/fila.

```
val textList = listOf(  
    "Precio descendente",  
    "Precio ascendente",  
    "Puntuación 4+",  
    "Ofertas",  
    "Más vendidos"  
)  
  
FlowRow(  
    horizontalArrangement = Arrangement.spacedBy(24.dp),  
    modifier = Modifier.padding(8.dp),  
    //maxItemsInEachRow = 2  
) { this: RowScope  
    textList.forEach { text ->  
        Text(  
            text = text,  
            modifier = Modifier  
                .padding(4.dp)  
                .background(Color.LightGray)  
                .padding(4.dp)  
        )  
    }  
}
```



8.- Componentes Lazy

Los componentes **Lazy** permiten tener una serie de componentes organizados en fila, en columna o en cuadrícula.

Los componentes **Lazy** tienen **dos ventajas** muy importantes:

- En el caso de que el contenido no quepa en el espacio disponible, **se podrá hacer scroll sin tener que añadir nada más.**
- **Solo se cargan los componentes que se ven en pantalla, el anterior y el siguiente** de manera que se optimiza el uso de memoria.

8.- Componentes Lazy

Los componentes **Lazy** disponibles son:

- **LazyColumn** y **LazyRow**.
- **LazyVerticalGrid** y **LazyhorizontalGrid**.
- **LazyVerticalStaggeredGrid** y **LazyHorizontalStaggeredGrid**.

Los componentes **Grid** organizan el contenido en forma de cuadrícula.

8.- Componentes Lazy

Todos los componentes Lazy tienen varios parámetros comunes:

- **reverseLayout**: booleano que indica si los elementos se deben mostrar en orden inverso, por defecto es false.
- **userScrollEnabled**: booleano que indica si el scroll está activo o no, por defecto es true.

Los componentes Lazy para organizar en cuadrícula tienen los parámetros:

- **columns**: en una cuadrícula vertical indica el número de columnas.
- **rows**: en una cuadrícula horizontal indica el número de filas.

Para añadir contenido a los elementos Lazy se usan los siguientes bloques:

- **item**: añade un elemento al componente Lazy.
- **items**: recorre una lista y añade cada elemento de la lista al componente Lazy.

9.- Componente LazyColumn

```
val dragonballCharacters = listOf("Son Goku", "Vegeta", "Piccolo", "Son Gohan",
    "Trunks", "Bulma", "Krillin", "Master Roshi",
    "Freezer", "Cell")

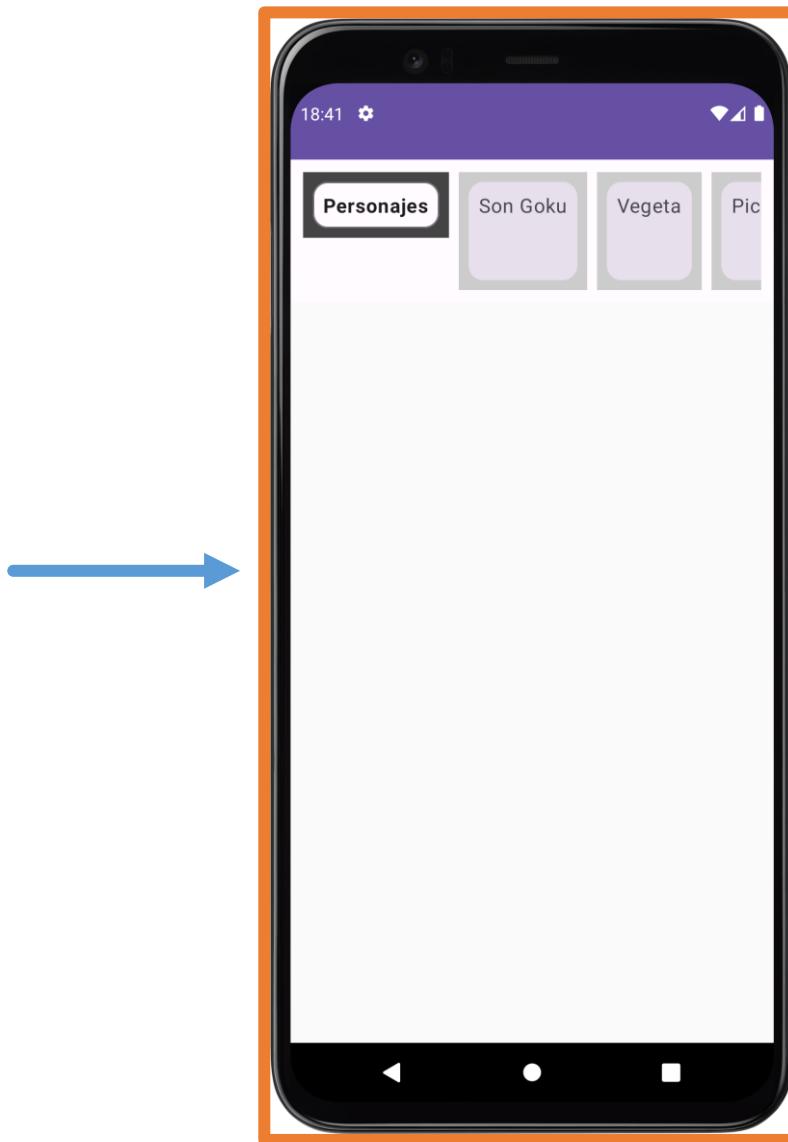
LazyColumn(verticalArrangement = Arrangement.spacedBy(8.dp)) { this: LazyListScope
    item { this: LazyItemScope
        OutlinedCard(
            modifier = Modifier
                .background(Color.DarkGray)
                .padding(8.dp)
                .fillMaxWidth()
        ) { this: ColumnScope
            Text(
                text = "Personajes",
                fontWeight = FontWeight.Bold,
                modifier = Modifier.padding(8.dp)
            )
        }
    }
    items(dragonballCharacters) { this: LazyItemScope character ->
        Card(
            modifier = Modifier
                .background(Color.LightGray)
                .padding(8.dp)
                .fillMaxWidth()
        ) { this: ColumnScope
            Text(
                text = character,
                modifier = Modifier.height(80.dp).padding(8.dp)
            )
        }
    }
}
```



9.- Componente LazyRow

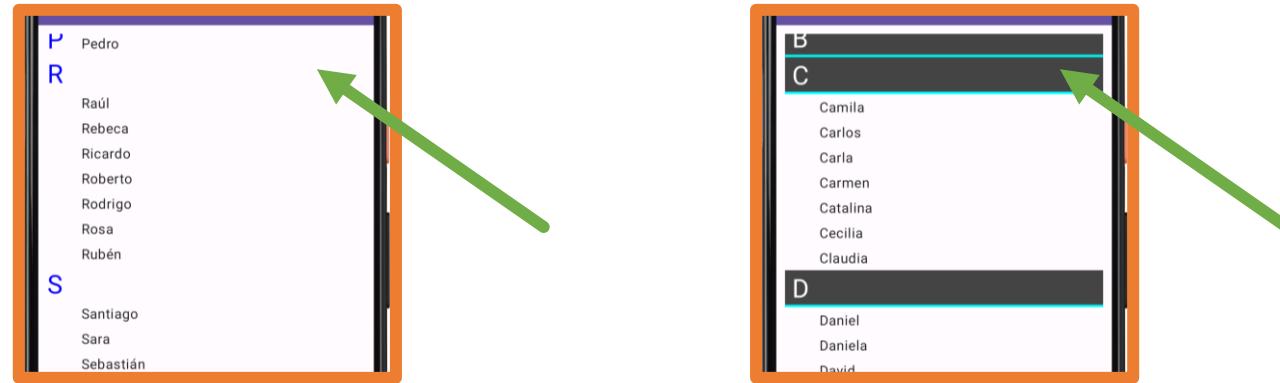
```
val dragonballCharacters = listOf("Son Goku", "Vegeta", "Piccolo", "Son Gohan",
    "Trunks", "Bulma", "Krillin", "Master Roshi",
    "Freezer", "Cell")

LazyRow(horizontalArrangement = Arrangement.spacedBy(8.dp)) { this: LazyListScope
    item { this: LazyItemScope
        OutlinedCard(
            modifier = Modifier
                .background(Color.DarkGray)
                .padding(8.dp)
                .fillMaxWidth()
        ) { this: ColumnScope
            Text(
                text = "Personajes",
                fontWeight = FontWeight.Bold,
                modifier = Modifier.padding(8.dp)
            )
        }
    }
    items(dragonballCharacters) { this: LazyItemScope character ->
        Card(
            modifier = Modifier
                .background(Color.LightGray)
                .padding(8.dp)
                .fillMaxWidth()
        ) { this: ColumnScope
            Text(
                text = character,
                modifier = Modifier.height(80.dp).padding(8.dp)
            )
        }
    }
}
```



9.- Componentes LazyColumn y LazyRow: StickyHeaders

Un comportamiento muy habitual con LazyColumn y LazyRow son las cabeceras "pegajosas" usadas en la aplicación contactos.



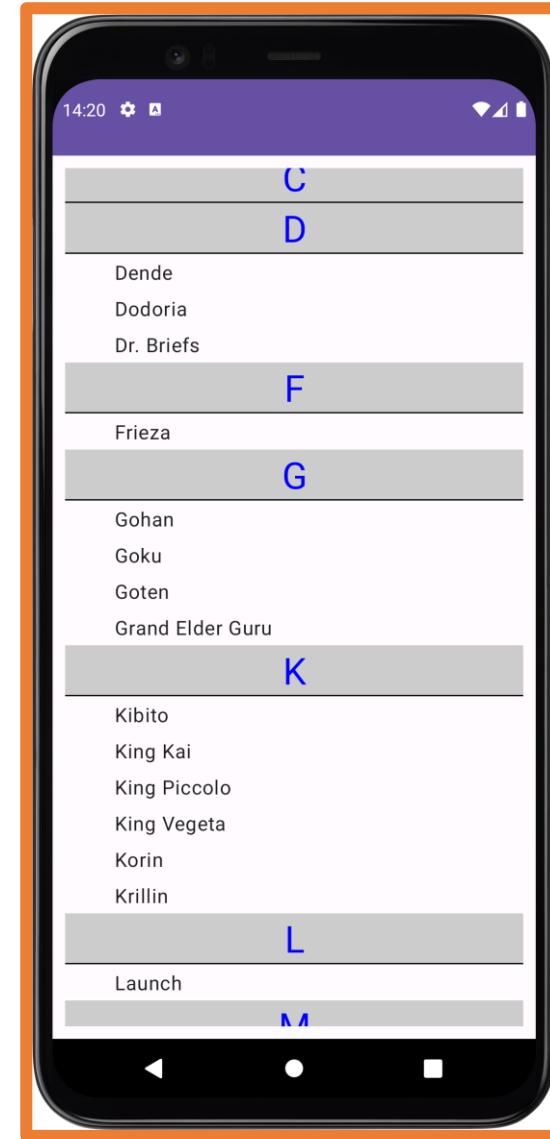
Para usar estas cabeceras se utiliza el elemento **stickyHeader** y se deben agrupar los elementos a mostrar por algún criterio como puede ser la primera letra.

```
// dragonBallCharacters es un List<String>
val groupedCharacters: Map<Char, List<String>> = dragonBallCharacters.sorted().groupBy { it[0] }
```

De esta manera se tiene un Map que se puede recorrer para llenar el contenido de la lista.

9.- Componentes LazyColumn y LazyRow: StickyHeaders

```
LazyColumn { this: LazyListScope
    // dragonBallCharacters es un List<String>
    val groupedCharacters: Map<Char, List<String>> = dragonBallCharacters.sorted().groupBy { it[0] }
    groupedCharacters.forEach { header, names ->
        stickyHeader { this: LazyItemScope
            Text(
                text = header.toString(),
                fontSize = 30.sp,
                color = Color.Blue,
                textAlign = TextAlign.Center,
                modifier = Modifier
                    .background(Color.LightGray)
                    .fillMaxWidth()
            )
            Divider(
                color = Color.Black,
                thickness = 1.dp
            )
        }
        items(names) { this: LazyItemScope name ->
            Text(
                text = name,
                modifier = Modifier
                    .padding(start = 40.dp)
                    .padding(vertical = 4.dp)
            )
        }
    }
}
```



10.- Componentes LazyVerticalGrid y LazyHorizontalGrid

Con **LazyVerticalGrid** y **LazyHorizontalGrid** se pueden organizar los elementos en cuadrícula por columnas o por filas.

Todos **los elementos ocuparán el mismo espacio**, el espacio que ocupan es el espacio del elemento más grande.

Para indicar el número de columnas en **LazyVerticalGrid** se usa el parámetro **columns** y para indicar el número de filas en **LazyHorizontalGrid** se usa el parámetro **rows** con el valor:

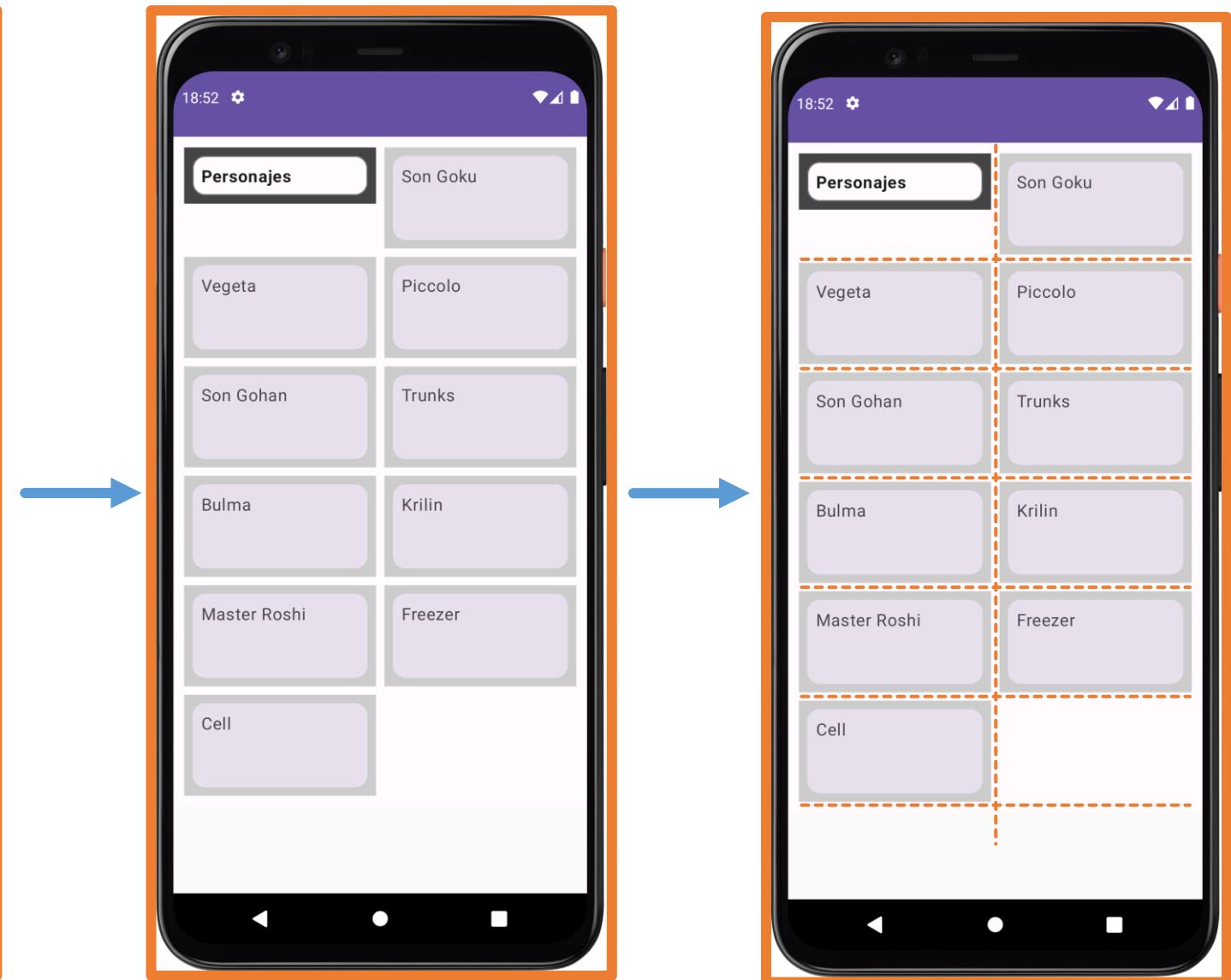
`GridCells.Fixed(número_columnas/filas)`

`GridCells.Adaptive(tamaño_mínimo_columna/fila.dp)`

10.- Componente LazyVerticalGrid

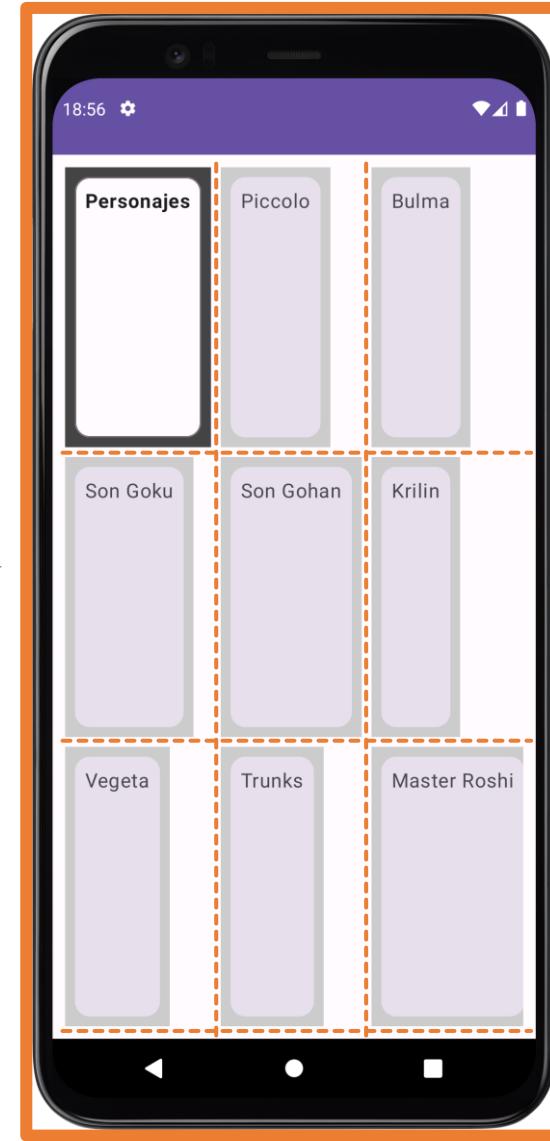
```
val dragonballCharacters = listOf("Son Goku", "Vegeta", "Piccolo", "Son Gohan",
    "Trunks", "Bulma", "Krillin", "Master Roshi",
    "Freezer", "Cell")

LazyVerticalGrid(
    columns = GridCells.Adaptive(150.dp),
    verticalArrangement = Arrangement.spacedBy(8.dp),
    horizontalArrangement = Arrangement.spacedBy(8.dp),
) { this: LazyGridScope
    item { this: LazyGridItemScope
        OutlinedCard(
            modifier = Modifier
                .background(Color.DarkGray)
                .padding(8.dp)
                .fillMaxWidth()
        ) { this: ColumnScope
            Text(
                text = "Personajes",
                fontWeight = FontWeight.Bold,
                modifier = Modifier.padding(8.dp)
            )
        }
    }
    items(dragonballCharacters) { this: LazyGridItemScope character ->
        Card(
            modifier = Modifier
                .background(Color.LightGray)
                .padding(8.dp)
                .fillMaxWidth()
        ) { this: ColumnScope
            Text(
                text = character,
                modifier = Modifier.height(80.dp).padding(8.dp)
            )
        }
    }
}
```



10.- Componente LazyHorizontalGrid

```
val dragonballCharacters = listOf("Son Goku", "Vegeta", "Piccolo", "Son Gohan",
    "Trunks", "Bulma", "Krillin", "Master Roshi",
    "Freezer", "Cell")  
  
LazyHorizontalGrid(  
    rows = GridCells.Fixed(count: 3),  
    verticalArrangement = Arrangement.spacedBy(8.dp),  
    horizontalArrangement = Arrangement.spacedBy(8.dp),  
) { this: LazyGridScope  
    item { this: LazyGridItemScope  
        OutlinedCard(  
            modifier = Modifier  
                .background(Color.DarkGray)  
                .padding(8.dp)  
                .fillMaxWidth()  
        ) { this: ColumnScope  
            Text(  
                text = "Personajes",  
                fontWeight = FontWeight.Bold,  
                modifier = Modifier.padding(8.dp)  
            )  
        }  
    }  
    items(dragonballCharacters) { this: LazyGridItemScope character ->  
        Card(  
            modifier = Modifier  
                .background(Color.LightGray)  
                .padding(8.dp)  
                .fillMaxWidth()  
        ) { this: ColumnScope  
            Text(  
                text = character,  
                modifier = Modifier.height(80.dp).padding(8.dp)  
            )  
        }  
    }  
}
```



11.- Componentes Staggered

Los componentes **LazyVerticalStaggeredGrid** y **LazyHorizontalStaggeredGrid** son similares a los anteriores pero cada elemento ocupa el espacio de su contenido.

Los parámetros más usados con **LazyVerticalStaggeredGrid** son:

columns: número de columnas

verticalSpacing: espacio vertical entre elementos.

horizontalArrangment: espacio horizontal entre elementos.

Los parámetros más usados con **LazyHorizontalStaggeredGrid** son:

rows: número de filas.

horizontalSpacing: espacio horizontal entre elementos.

verticalArrangment: espacio vertical entre elementos.

La cantidad de filas y columnas se indica con el valor:

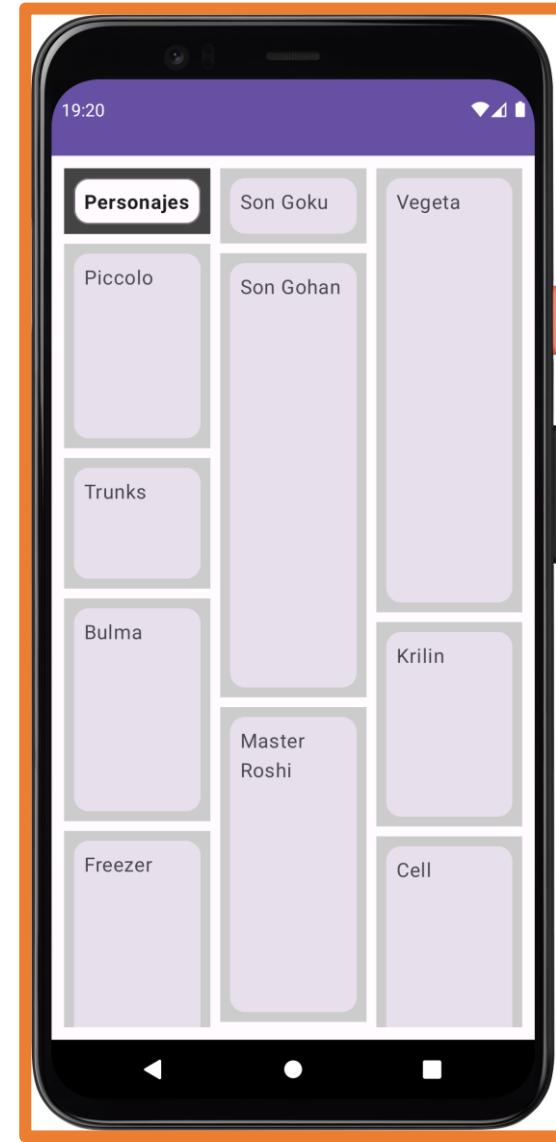
`StaggeredGridCells.Fixed(número_columnas/filas)`

`StaggeredGridCells.Adaptative(tamaño_mínimo_columna/fila.dp)`

11.- Componente LazyVerticalStaggeredGrid

```
val dragonballCharacters = listOf("Son Goku", "Vegeta", "Piccolo", "Son Gohan",
    "Trunks", "Bulma", "Krillin", "Master Roshi",
    "Freezer", "Cell")

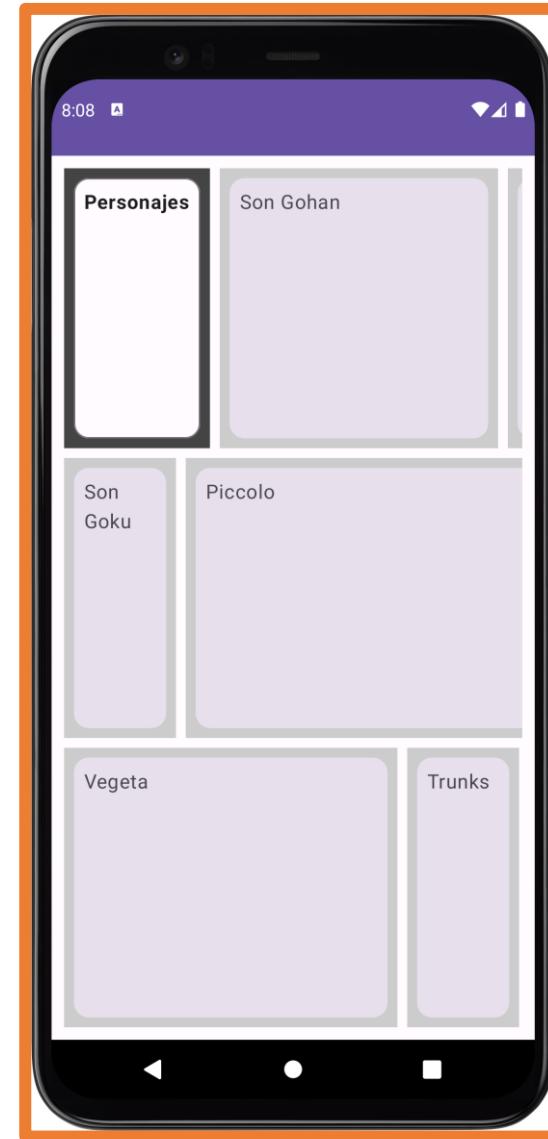
LazyVerticalStaggeredGrid(
    columns = StaggeredGridCells.Fixed( count: 3),
    verticalItemSpacing = 8.dp,
    horizontalArrangement = Arrangement.spacedBy(8.dp),
) { this: LazyStaggeredGridScope
    item { this: LazyStaggeredGridItemScope
        OutlinedCard(
            modifier = Modifier
                .background(Color.DarkGray)
                .padding(8.dp)
                .fillMaxWidth()
        ) { this: ColumnScope
            Text(
                text = "Personajes",
                fontWeight = FontWeight.Bold,
                modifier = Modifier.padding(8.dp)
            )
        }
    }
    items(dragonballCharacters) { this: LazyStaggeredGridItemScope, character ->
        val randomHeight = ((1 .. 30).random()*15).dp
        Card(
            modifier = Modifier
                .background(Color.LightGray)
                .padding(8.dp)
                .fillMaxWidth()
                .height(randomHeight)
        ) { this: ColumnScope
            Text(
                text = character,
                modifier = Modifier.height(80.dp).padding(8.dp)
            )
        }
    }
}
```



11.- Componente LazyHorizontalStaggeredGrid

```
val dragonballCharacters = listOf("Son Goku", "Vegeta", "Piccolo", "Son Gohan",
    "Trunks", "Bulma", "Krillin", "Master Roshi",
    "Freezer", "Cell")

LazyHorizontalStaggeredGrid(
    rows = StaggeredGridCells.Fixed( count: 3),
    horizontalItemSpacing = 8.dp,
    verticalArrangement = Arrangement.spacedBy(8.dp),
) { this:LazyStaggeredGridScope
    item { this:LazyStaggeredGridItemScope
        OutlinedCard(
            modifier = Modifier
                .background(Color.DarkGray)
                .padding(8.dp)
                .fillMaxWidth()
        ) { this:ColumnScope
            Text(
                text = "Personajes",
                fontWeight = FontWeight.Bold,
                modifier = Modifier.padding(8.dp)
            )
        }
    }
    items(dragonballCharacters) { this:LazyStaggeredGridItemScope character ->
        val randomHeight = ((2 .. 30).random()*15).dp
        Card(
            modifier = Modifier
                .background(Color.LightGray)
                .padding(8.dp)
                .fillMaxHeight()
                .width(randomHeight)
        ) { this:ColumnScope
            Text(
                text = character,
                modifier = Modifier.height(80.dp).padding(8.dp)
            )
        }
    }
}
```



12.- Gestionar el estado en componentes Lazy

Como se puede ver en la definición de los componentes Lazy, **Android gestiona automáticamente el estado del scroll:**

```
@Composable
fun LazyVerticalStaggeredGrid(
    columns: StaggeredGridCells,
    modifier: Modifier = Modifier,
    state: LazyStaggeredGridState = rememberLazyStaggeredGridState(),
    contentPadding: PaddingValues = PaddingValues(0.dp),
    reverseLayout: Boolean = false,
    verticalItemSpacing: Dp = 0.dp,
```



Es interesante crear una variable propia para gestionar el estado y saber:

- Dirección en la que el usuario está haciendo scroll.
- Cantidad de scroll realizada.
- Cuál es el índice del primer elemento visible.
- ...

12.- Gestionar el estado en componentes Lazy

En el ejemplo se crea un estado propio para guardar el scroll del componente LazyVerticalStaggeredGrid y se asigna a dicho componente.

También se crea un estado para mostrar/ocultar un botón. Este estado depende del estado anterior así que se debe utilizar **derivedStateOf**.

El estado para el botón cambia cuando el primer ítem de la lista ya no está visible completamente.

Por último, el botón utiliza una corutina (se verán más adelante) para mover el scroll al primer elemento de la lista.

```
Column(
    horizontalAlignment = Alignment.CenterHorizontally,
    modifier = Modifier.fillMaxSize().padding(8.dp)
) { this: ColumnScope
    val dragonballCharacters = listOf("Son Goku", "Vegeta", "Piccolo", "Son Gohan", "Trunks", "Bulma", "Krillin",
        "master Roshi", "Freezer", "Cell")

    val scrollState = rememberLazyStaggeredGridState()
    val showEndTextGrid by remember {
        derivedStateOf {
            scrollState.firstVisibleItemIndex > 0
        }
    }
    val coroutineGridScope = rememberCoroutineScope()

    LazyVerticalStaggeredGrid(
        columns = StaggeredGridCells.Fixed(count: 3),
        verticalItemSpacing = 8.dp,
        horizontalArrangement = Arrangement.spacedBy(8.dp),
        state = scrollState,
        modifier = Modifier.weight(.5f)
    ) { this: LazyStaggeredGridScope
        item { this: LazyStaggeredGridItemScope
            OutlinedCard(
                modifier = Modifier
                    .background(Color.DarkGray)
                    .padding(8.dp)
                    .fillMaxWidth()
            ) { this: ColumnScope
                Text(text = "Personajes", fontWeight = FontWeight.Bold, modifier = Modifier.padding(8.dp))
            }
        }
        items(dragonballCharacters) { this: LazyStaggeredGridItemScope, character ->
            val randomHeight = ((2 .. 30).random() * 15).dp
            Card(
                modifier = Modifier.background(Color.LightGray).padding(8.dp).fillMaxWidth().height(randomHeight)
            ) { this: ColumnScope
                Text(text = character, modifier = Modifier.height(80.dp).padding(8.dp))
            }
        }
        if (showEndTextGrid) {
            Button(
                onClick = {
                    coroutineGridScope.launch { this: CoroutineScope
                        scrollState.animateScrollToItem(index: 0, scrollOffset: 0)
                    }
                }
            ) { this: RowScope
                Text(text = "Volver arriba")
            }
        }
    }
}
```

13.- Componente ListItem

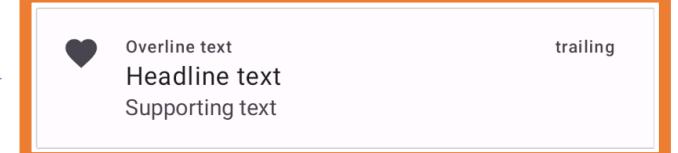
Aunque en una columna, sea del tipo que sea, se puede añadir cualquier tipo de componente, Jetpack Compose ofrece el componente **ListItem** para dar un estilo uniforme a los elementos de una lista.

Un **ItemList** tiene cinco zonas, generalmente en la zona **leadingText** se suele situar un icono o una imagen. En la zona **trailingContent** también se suele situar icono o texto.

Si hay contenido en la zona **overlineText** el contenido de **leadingContent** y de **trailingContent** alinearán a la parte superior, si no hay **overlineText** se alinearán al centro verticalmente.

```
@Composable  
@ExperimentalMaterial3Api  
fun ListItem(  
    headlineText: @Composable () -> Unit,  
    modifier: Modifier = Modifier,  
    overlineText: @Composable (() -> Unit)? = null,  
    supportingText: @Composable (() -> Unit)? = null,  
    leadingContent: @Composable (() -> Unit)? = null,  
    trailingContent: @Composable (() -> Unit)? = null,  
    colors: ListItemColors = ListItemDefaults.colors(),  
    tonalElevation: Dp = ListItemDefaults.Elevation,  
    shadowElevation: Dp = ListItemDefaults.Elevation,  
) {
```

```
ListItem(  
    headlineText = { Text(text = "Headline text") },  
    overlineText = { Text(text = "Overline text") },  
    supportingText = { Text(text = "Supporting text") },  
    leadingContent = {  
        Icon(  
            imageVector = Icons.Filled.Favorite,  
            contentDescription = "Favorito"  
        )  
    },  
    trailingContent = { Text(text = "trailing") }
```



14.- Componente Scaffold

El componente **Scaffold** es una estructura por defecto para crear interfaces complejas que ofrece Jetpack Compose siguiendo los principios de Material.

Incorpora varios componentes habituales como:

- **topBar**: barra de navegación superior.
- **bottomBar**: barra de navegación inferior.
- **snackbarHost**: permite mostrar mensajes que sustituyen a los Toast antiguos.
- **floatingActionButton**: icono flotante generalmente en la parte inferior de la pantalla.

```
@Composable
fun Scaffold(
    modifier: Modifier = Modifier,
    topBar: @Composable () -> Unit = {},
    bottomBar: @Composable () -> Unit = {},
    snackbarHost: @Composable () -> Unit = {},
    floatingActionButton: @Composable () -> Unit = {},
    floatingActionButtonPosition: FabPosition = FabPosition.End,
    containerColor: Color = MaterialTheme.colorScheme.background,
    contentColor: Color = contentColorFor(containerColor),
    contentWindowInsets: WindowInsets = ScaffoldDefaults.contentWindowInsets,
    content: @Composable (PaddingValues) -> Unit
) {
```

14.- Componente Scaffold

El único parámetro obligatorio es **content** que al ser el último y una función lambda se puede extraer fuera de los paréntesis.

```
@Composable
fun Scaffold(
    modifier: Modifier = Modifier,
    topBar: @Composable () -> Unit = {},
    bottomBar: @Composable () -> Unit = {},
    snackbarHost: @Composable () -> Unit = {},
    floatingActionButton: @Composable () -> Unit = {},
    floatingActionButtonPosition: FabPosition = FabPosition.End,
    containerColor: Color = MaterialTheme.colorScheme.background,
    contentColor: Color = contentColorFor(containerColor),
    contentWindowInsets: WindowInsets = ScaffoldDefaults.contentWindowInsets,
    content: @Composable (PaddingValues) -> Unit
) {
```

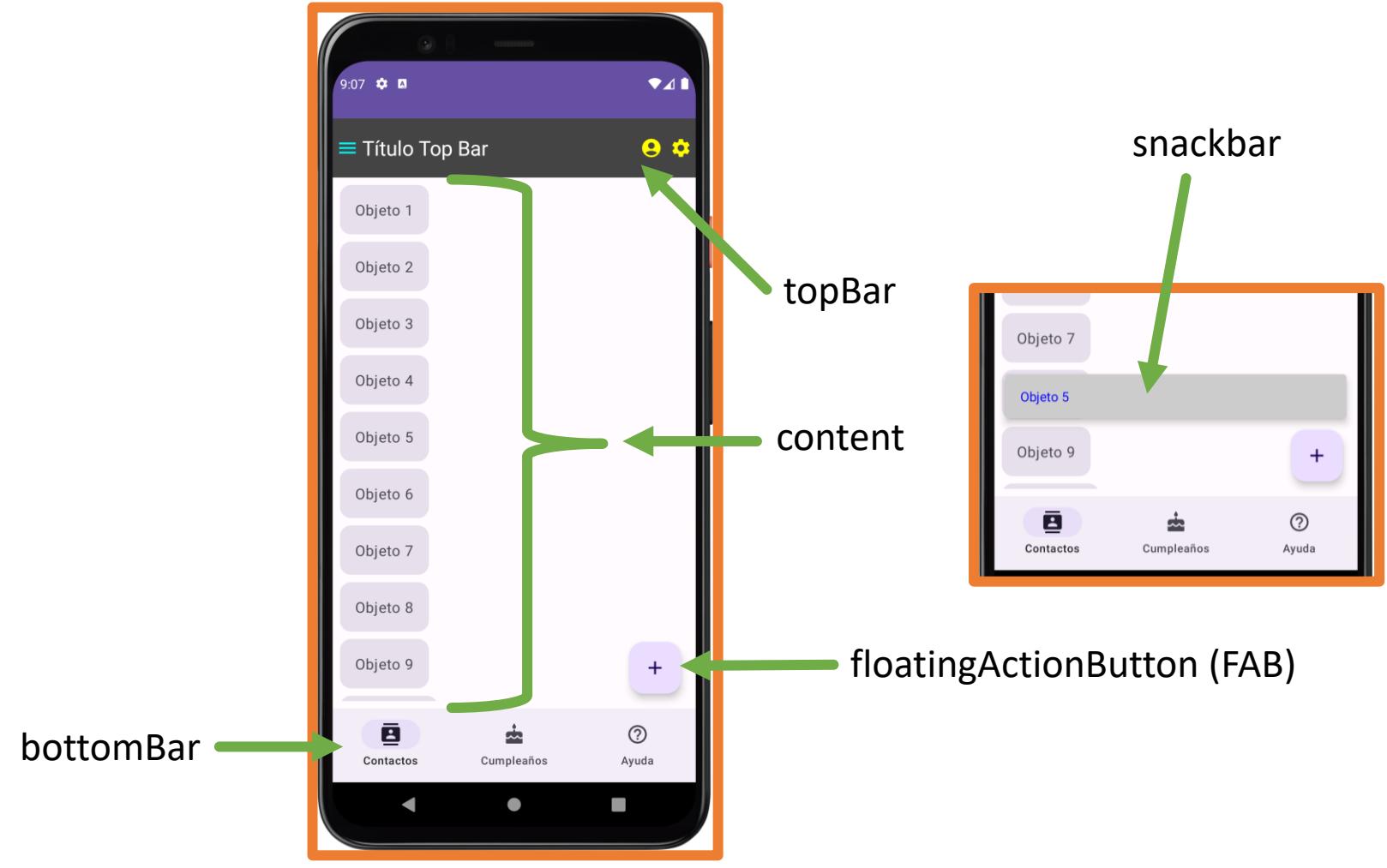
Si se decide utilizar **Scaffold** se debe elegir entre las siguientes opciones:

- Que el **Scaffold** sea el único hijo del componente **Surface** principal del proyecto y añadir el resto de componentes a dicho **Scaffold**.
- Sustituir el **Surface** principal por un **Scaffold** y en él añadir el resto de componentes.

14.- Componente Scaffold

```
@Composable  
fun Scaffold(  
    modifier: Modifier = Modifier,  
    topBar: @Composable () -> Unit = {},  
    bottomBar: @Composable () -> Unit = {},  
    snackbarHost: @Composable () -> Unit = {},  
    floatingActionButton: @Composable () -> Unit = {},  
    floatingActionButtonPosition: FabPosition = FabPosition.End,  
    containerColor: Color = MaterialTheme.colorScheme.background,  
    contentColor: Color = contentColorFor(containerColor),  
    contentWindowInsets: WindowInsets = ScaffoldDefaults.contentWindowInsets,  
    content: @Composable (PaddingValues) -> Unit  
) {
```

```
Scaffold (  
    topBar = {  
  
    },  
    bottomBar = {  
  
    },  
    snackbarHost = {  
  
    },  
    floatingActionButton = {  
  
    },  
    floatingActionButtonPosition = FabPosition.End  
) { it: PaddingValues  
    Column(  
        modifier = Modifier  
            .padding(it)  
    ) { this: ColumnScope  
  
    }  
}
```



14.- Componente Scaffold

Se puede observar que los parámetros **topBar**, **bottomBar**, **snackbarHost** y **floatingActionButton** son de tipo **@Composable** admitiendo así cualquier componente JetpackCompose.

```
@Composable  
fun Scaffold(  
    modifier: Modifier = Modifier,  
    topBar: @Composable () -> Unit = {},  
    bottomBar: @Composable () -> Unit = {},  
    snackbarHost: @Composable () -> Unit = {},  
    floatingActionButton: @Composable () -> Uni  
    floatingActionButtonPosition: FabPosition =
```

Gracias a esto se podrá personalizar el Scaffold como se quiera.

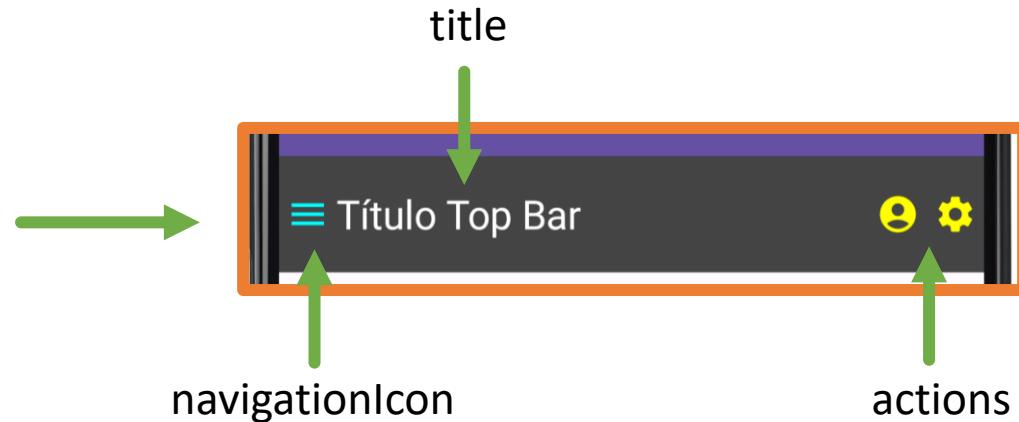
Jetpack Compose ofrece una serie de **componentes específicos** para esos parámetros y se recomienda su uso. Además, estos **parámetros se pueden configurar** igualmente para personalizar la interfaz. Esos componentes también se pueden usar sin un Scaffold.

15.- Componente TopAppBar

El componente **TopAppBar** permite crear una barra superior con un título e iconos para realizar acciones.

El único parámetro obligatorio es el **título**.

```
@Composable  
fun TopAppBar(  
    title: @Composable () -> Unit,  
    modifier: Modifier = Modifier,  
    navigationIcon: @Composable () -> Unit = {},  
    actions: @Composable RowScope.() -> Unit = {},  
    windowInsets: WindowInsets = TopAppBarDefaults.windowInsets,  
    colors: TopAppBarColors = TopAppBarDefaults.topAppBarColors(),  
    scrollBehavior: TopAppBarScrollBehavior? = null  
) {
```



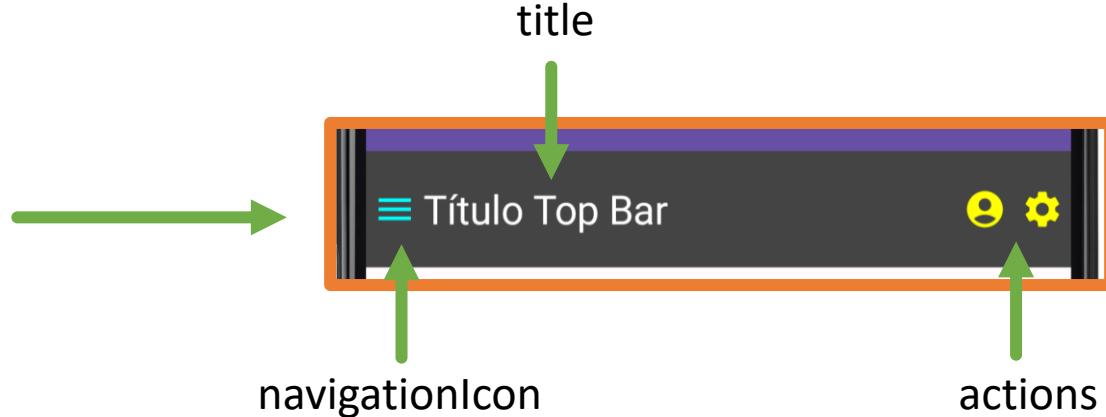
Existen tres versiones más para la topBar:

- **CenterAlignedTopAppBar**: igual que `TopAppBar` pero con el título centrado.
- **MediumTopAppBar**: el título se sitúa una línea más abajo.
- **LargeTopAppBar**: igual que `MediumTopAppBar` pero con el título más grande.

15.- Componente TopAppBar

```
@Composable  
fun TopAppBar(  
    title: @Composable () -> Unit,  
    modifier: Modifier = Modifier,  
    navigationIcon: @Composable () -> Unit = {},  
    actions: @Composable RowScope.() -> Unit = {},  
    windowInsets: WindowInsets = TopAppBarDefaults.windowInsets,  
    colors: TopAppBarColors = TopAppBarDefaults.topAppBarColors(),  
    scrollBehavior: TopAppBarScrollBehavior? = null  
) {
```

```
topBar = {  
    TopAppBar(  
        title = { Text(text = "Título Top Bar") },  
        navigationIcon = {  
            Icon(  
                imageVector = Icons.Default.Menu,  
                contentDescription = "Menú"  
            )  
        },  
        actions = { this: RowScope  
            Icon(  
                imageVector = Icons.Default.AccountCircle,  
                contentDescription = "Cuenta"  
            )  
            Spacer(Modifier.width(8.dp))  
            Icon(  
                imageVector = Icons.Default.Settings,  
                contentDescription = "Configuración"  
            )  
        },  
        colors = TopAppBarDefaults.topAppBarColors(  
            navigationIconContentColor = Color.Cyan,  
            containerColor = Color.DarkGray,  
            titleContentColor = Color.White,  
            actionIconContentColor = Color.Yellow  
        )  
    )  
},
```

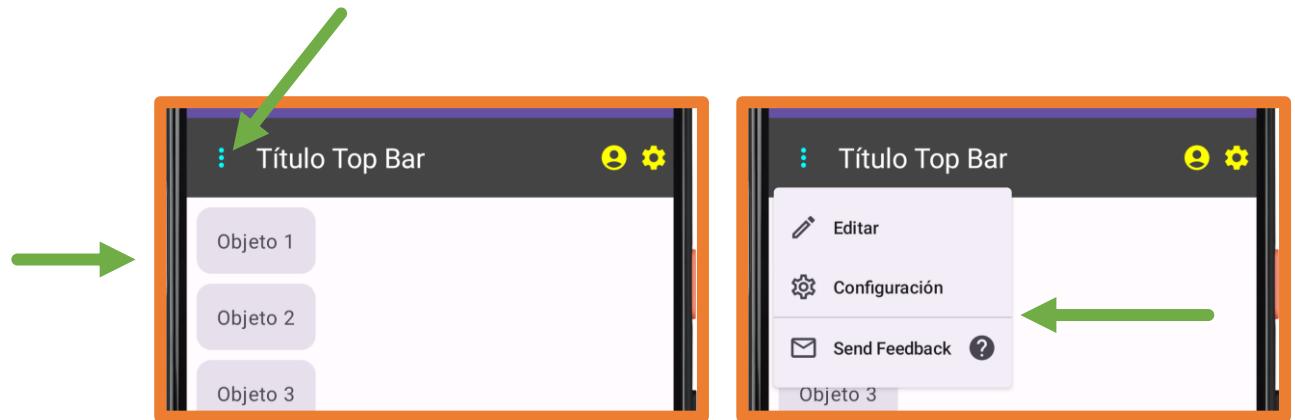


16.- Componente DropdownMenu

Es habitual que en la **TopAppBar** aparezca un ícono que abra un menú desplegable, tanto como **navigationIcon** o como algún ícono de **actions**.

Este menú se crea con los componentes **DropdownMenu** y **DropdownMenuItem**.

```
navigationIcon = {
    var expanded by rememberSaveable { mutableStateOf( value: false ) }
    IconButton(onClick = { expanded = true }) {
        Icon(imageVector = Icons.Default.MoreVert, contentDescription = "Menú")
    }
}
DropdownMenu(
    expanded = expanded,
    onDismissRequest = { /*TODO*/ }
) { this: ColumnScope
    DropdownMenuItem(
        text = { Text( text: "Editar") },
        onClick = { /*TODO*/ },
        leadingIcon = { Icon(imageVector = Icons.Outlined.Edit, contentDescription = null) }
    )
    DropdownMenuItem(
        text = { Text( text: "Configuración") },
        onClick = { /*TODO*/ },
        leadingIcon = { Icon(imageVector = Icons.Outlined.Settings, contentDescription = null) }
    )
    Divider()
    DropdownMenuItem(
        text = { Text( text: "Envía comentarios") },
        onClick = { /*TODO*/ },
        leadingIcon = { Icon(imageVector = Icons.Outlined.Email, contentDescription = null) },
        trailingIcon = { Icon(imageVector = Icons.Outlined.Help, contentDescription = null) }
    )
}
```



17.- Componente FloatingActionButton

Los **FloatingActionButton** (FAB) son botones que representan la acción más importante en una pantalla, tienen un estilo predeterminado y deberían de flotar sobre el resto de elementos de la pantalla.

Su uso principal es con el layout Scaffold pero se puede usar independientemente.

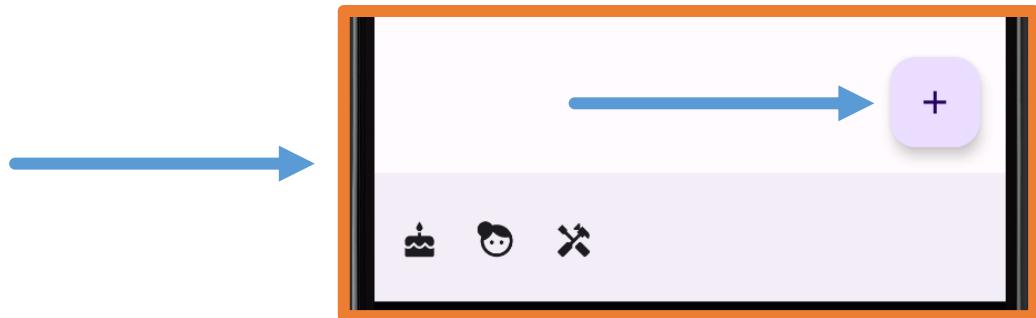
Existen cuatro componentes diferentes:

- **FloatingActionButton**: solo contiene un ícono.
- **SmallFloatingActionButton**: contiene un ícono y se usa en pantallas pequeñas.
- **LargeFloatingActionButton**: contiene un ícono y se usa en pantallas grandes.
- **ExtendedFloatingActionButton**: contiene un ícono y un texto y se usa cuando un FAB necesita algo más de información.

17.- Componente FloatingActionButton

En un Scaffold se puede añadir un **FloatingActionButton** que se situará en la parte inferior del contenido. Se puede indicar si aparecerá en el centro o en el final (según el sentido de lectura del idioma configurado).

```
floatingActionButton = {  
    FloatingActionButton(onClick = { /*TODO*/ }) {  
        Icon(  
            imageVector = Icons.Default.Add,  
            contentDescription = "Añadir"  
        )  
    }  
},  
floatingActionButtonPosition = FabPosition.End
```



18.- BottomBar

Una **BottomBar** es una barra inferior con iconos que generalmente se utiliza para cambiar el contenido del Scaffold.

Existen tres posibilidades a la hora de crear la BottomBar:

- **BottomAppBar**: con iconos y/o texto.
- **BottomAppBar**: con iconos y FloatingActionButton.
- **NavigationBar**: barra con tres, cuatro o cinco iconos.

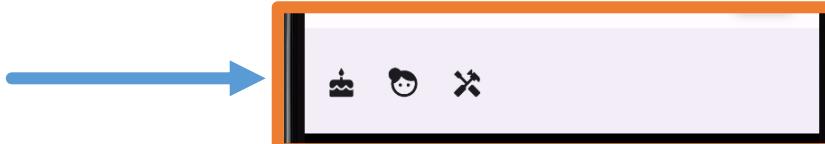
En una **NavigationBar** se usan **NavigationBarItem** para crear los elementos de la barra de navegación.

Si se quiere se pueden usar **NavigationBarItem** en una **BottomAppBar**.

18.- BottomBar: BottomAppBar

Si se usa **BottomAppBar** indicando **iconos** o **texto** estos se situarán alineados a la izquierda, si se usan **NavigationBarItem** estos se centrarán en el espacio disponible.

```
bottomBar = {  
    BottomAppBar { this: RowScope  
        IconButton(onClick = { /*TODO*/ }) {  
            Icon(  
                imageVector = Icons.Default.Cake,  
                contentDescription = null  
            )  
        }  
        IconButton(onClick = { /*TODO*/ }) {  
            Icon(  
                imageVector = Icons.Default.Face4,  
                contentDescription = null  
            )  
        }  
        IconButton(onClick = { /*TODO*/ }) {  
            Icon(  
                imageVector = Icons.Default.Handyman,  
                contentDescription = null  
            )  
        }  
    },  
}
```



18.- BottomBar: BottomAppBar

BottomAppBar tiene un constructor con los parámetros **actions** y **floatingActionButton**.

Si se ha configurado el Scaffold con el parámetro **floatingActionButton** no se debe utilizar una **BottomBar** con **floatingActionButton**.

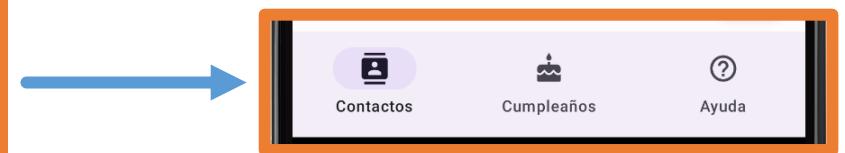
```
bottomBar = {
    BottomAppBar(
        actions = { this: RowScope ->
            IconButton(onClick = { /*TODO*/ }) {
                Icon(
                    imageVector = Icons.Default.Cake,
                    contentDescription = null
                )
            }
            IconButton(onClick = { /*TODO*/ }) {
                Icon(
                    imageVector = Icons.Default.Face4,
                    contentDescription = null
                )
            }
            IconButton(onClick = { /*TODO*/ }) {
                Icon(
                    imageVector = Icons.Default.Handyman,
                    contentDescription = null
                )
            }
        },
        floatingActionButton = {
            FloatingActionButton(onClick = { /*TODO*/ }) {
                Icon(
                    imageVector = Icons.Default.Add,
                    contentDescription = null
                )
            }
        }
    )
},
```



18.- BottomBar: BottomAppBar

NavigationBar con elementos NavigationBarItem.

```
bottomBar = {
    NavigationBar { this: RowScope
        NavigationBarItem(
            icon = { Icon(imageVector = Icons.Default.Contacts, contentDescription = null) },
            label = { Text( text: "Contactos") },
            selected = true,
            onClick = { /* TODO */ }
        )
        NavigationBarItem(
            icon = { Icon(imageVector = Icons.Default.Cake, contentDescription = null) },
            label = { Text( text: "Cumpleaños") },
            selected = false,
            onClick = { /* TODO */ }
        )
        NavigationBarItem(
            icon = { Icon(imageVector = Icons.Default.HelpOutline, contentDescription = null) },
            label = { Text( text: "Ayuda") },
            selected = false,
            onClick = { /* TODO */ }
        )
    }
},
```



19.- Componente Snackbar

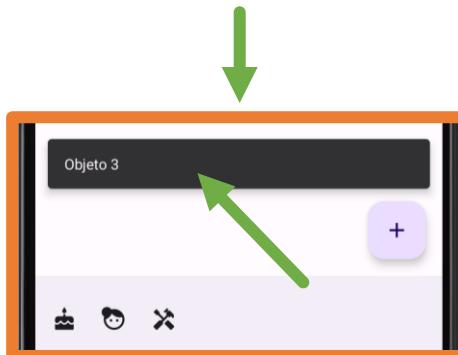
Los **Snackbar** son mensajes temporales que aparecen en la parte inferior de la pantalla.

Para crear un **Snackbar** se necesita un estado:

```
val snackbarHostState = remember { SnackbarHostState() }
```

En el Scaffold se configura en el parámetro **snackbarHost**, pudiendo configurar el estilo del Snackbar:

```
snackbarHost = {  
    SnackbarHost(hostState = snackbarHostState)  
},
```



```
snackbarHost = {  
    SnackbarHost(  
        hostState = snackbarHostState,  
        snackbar = { it: SnackbarData ->  
            Snackbar(  
                snackbarData = it,  
                containerColor = Color.LightGray,  
                contentColor = Color.Blue  
            )  
        }  
    )  
},
```



19.- Componente Snackbar

Para mostrar el **Snackbar** se necesita una corutina (se verán más adelante).

```
val coroutineScope = rememberCoroutineScope()
```

```
Column(  
    modifier = Modifier.padding(it),  
    verticalArrangement = Arrangement.spacedBy(4.dp)  
) { this: ColumnScope  
    for (i in 1 .. 5) {  
        Card(  
            onClick = {  
                coroutineScope.launch { this: CoroutineScope  
                    snackbarHostState.showSnackbar(message: "Objeto $i",  
                        duration = SnackbarDuration.Long)  
                }  
            },  
            modifier = Modifier.padding(start = 16.dp)  
        ) { this: ColumnScope  
            Text(  
                text = "Objeto $i",  
                modifier = Modifier.padding(8.dp)  
            )  
        }  
    }  
}
```

20.- Componente ConstraintLayout

Los layouts estudiados hasta este punto gestionan la posición de los elementos automáticamente.

El componente **ConstraintLayout** permite organizar los elementos en la pantalla teniendo que **indicar de manera expresa cómo se unen unos elementos con otros**.

De esta manera esas restricciones se mantendrán aunque se cambie la orientación de la pantalla o aunque el tamaño de pantalla sea diferente.

Este layout es uno de los más usados con las vistas antiguas en XML por ello es importante conocer su funcionamiento para poder realizar migraciones de XML a Jetpack Compose.

También es importante conocerlo porque da mucha libertad a la hora de crear las interfaces de usuario en Android.

20.- Componente ConstraintLayout

Para poder utilizar este componente primero se debe añadir una dependencia al archivo **build.gradle.kts (Module)** y sincronizar.

```
implementation("androidx.constraintlayout:constraintlayout-compose:1.0.1")
```

Una vez añadida la dependencia ya se puede utilizar el componente **ConstraintLayout** y en él añadir todos los componentes y restricciones.

Para poder indicar a los componentes cómo se unen entre sí, es necesario que los componentes tengan algún tipo de **referencia** que se pueda utilizar.

Se pueden crear referencias de una en una o bien varias de una sola vez.

```
ConstraintLayout(  
    modifier = Modifier.fillMaxSize()  
) { this: ConstraintLayoutScope  
    val titleRef = createRef()  
    val (iconRef, greetingRef) = createRefs()  
}
```

Una vez creadas las referencias ya se pueden asignar a los componentes del **ConstraintLayout**.

20.- Componente ConstraintLayout

Para asignar una referencia a un componente se utiliza el modificador **constrainAs**.

```
ConstraintLayout(modifier = Modifier.fillMaxSize()) { this: ConstraintLayoutScope
    val titleRef = createRef()
    val (iconRef, greetingRef, friendsTitleRef, friendsListRef) = createRefs()

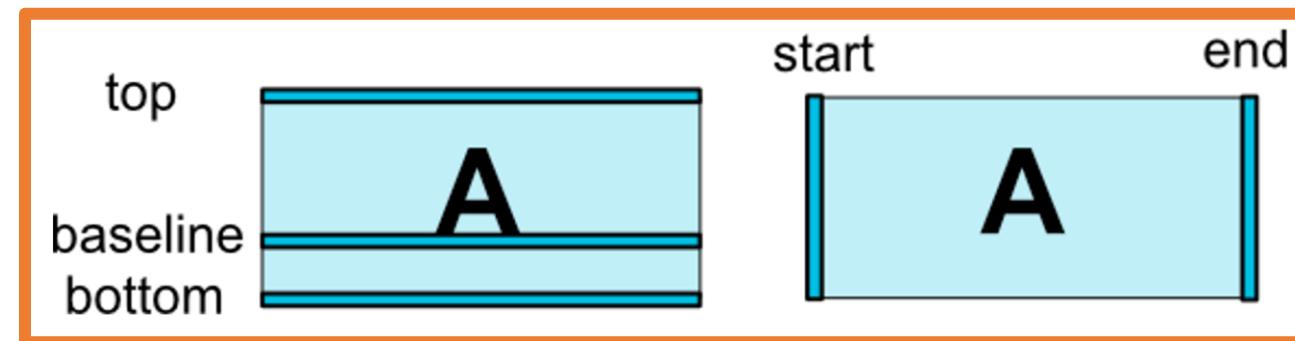
    Text(
        text = "Constraint Layout",
        fontSize = 20.sp,
        modifier = Modifier.constrainAs(titleRef) { this: ConstrainScope
            // Añadir restricciones
        }
    )
    Icon(
        imageVector = Icons.TwoTone.Android,
        contentDescription = "Android",
        modifier = Modifier.size(120.dp).constrainAs(iconRef) { this: ConstrainScope
            // Añadir restricciones
        }
    )
    Text(
        text = "Hola Rick!",
        modifier = Modifier.constrainAs(greetingRef) { this: ConstrainScope
            // Añadir restricciones
        }
    )
    Text(
        text = "Amigos:",
        modifier = Modifier.constrainAs(friendsTitleRef) { this: ConstrainScope
            // Añadir restricciones
        }
    )
    Column(modifier = Modifier.constrainAs(friendsListRef) { this: ConstrainScope
        // Añadir restricciones
    }) { this: ColumnScope
        listof("Hombre pájaro", "Gearhead", "Squanchy", "Unidad", "Dr. Xenon Bloom", "Presidente Obama", "Jaguar").forEach { it: String
            ListItem(
                headlineContent = { Text(it) },
                leadingContent = {
                    Icon(
                        imageVector = Icons.Default.AccountCircle,
                        contentDescription = "Destino: $it",
                        modifier = Modifier.size(50.dp)
                    )
                }
            )
        }
    }
}
```

20.- Componente ConstraintLayout

Una vez asignadas las referencias a los componentes ya se puede indicar cómo se unen estos en el layout.

Los elementos se pueden unir entre sí y también al propio contenedor que es el ConstraintLayout, la referencia del propio contenedor es **parent**.

Para poder realizar las uniones es necesario conocer en qué puntos se pueden unir.

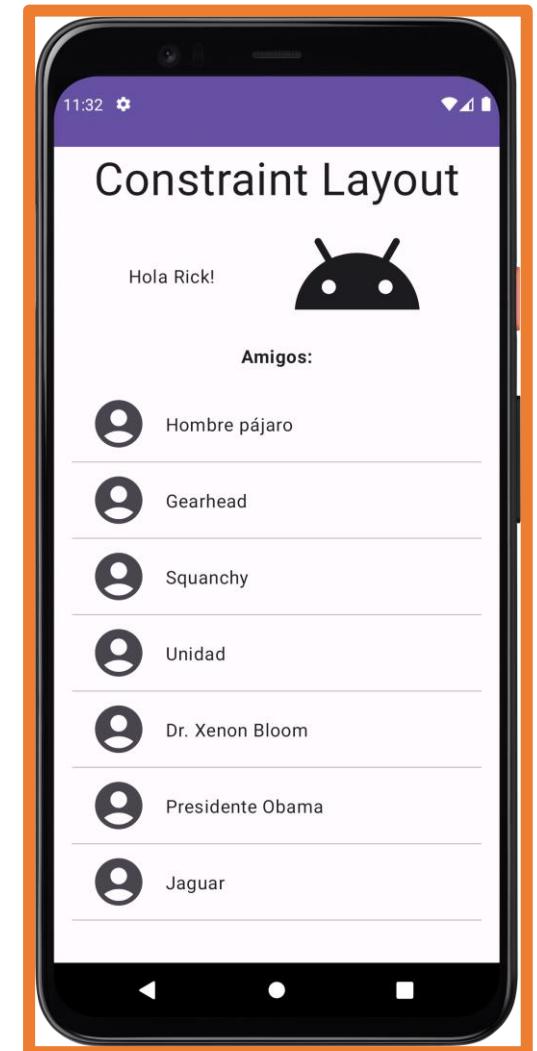


Los puntos **start** y **end** dependerán del sentido de lectura del idioma configurado en el dispositivo.

20.- Componente ConstraintLayout

```
Text(  
    text = "Constraint Layout",  
    fontSize = 40.sp,  
    modifier = Modifier.constrainAs(titleRef) { this: ConstrainScope  
        top.linkTo(parent.top)  
        start.linkTo(parent.start)  
        end.linkTo(parent.end)  
    }  
)  
Icon(  
    imageVector = Icons.TwoTone.Android,  
    contentDescription = "Android",  
    modifier = Modifier.size(120.dp).constrainAs(iconRef) { this: ConstrainScope  
        top.linkTo(titleRef.bottom)  
        start.linkTo(greetingRef.end)  
        end.linkTo(parent.end)  
    }  
)  
Text(  
    text = "Hola Rick!",  
    modifier = Modifier.constrainAs(greetingRef) { this: ConstrainScope  
        top.linkTo(iconRef.top)  
        bottom.linkTo(iconRef.bottom)  
        start.linkTo(parent.start)  
        end.linkTo(iconRef.start)  
    }  
)
```

```
Text(  
    text = "Amigos:",  
    fontWeight = FontWeight.Bold,  
    modifier = Modifier.constrainAs(friendsTitleRef) { this: ConstrainScope  
        top.linkTo(iconRef.bottom)  
        start.linkTo(parent.start)  
        end.linkTo(parent.end)  
    }  
)  
Column(modifier = Modifier.padding(16.dp).constrainAs(friendsListRef) { this: ConstrainScope  
    top.linkTo(friendsTitleRef.bottom)  
}) { this: ColumnScope  
    listOf("Hombre pájaro", "Gearhead", "Squanchy", "Unidad", "Dr. Xenon Bloom", "Presidente Obama", "Jaguar").forEach { it: String  
        ListItem(  
            headlineContent = { Text(it) },  
            leadingContent = {  
                Icon(  
                    imageVector = Icons.Default.AccountCircle,  
                    contentDescription = "Destino: $it",  
                    modifier = Modifier.size(50.dp)  
                )  
            }  
            Divider()  
        }  
    }  
}
```



20.- Componente ConstraintLayout

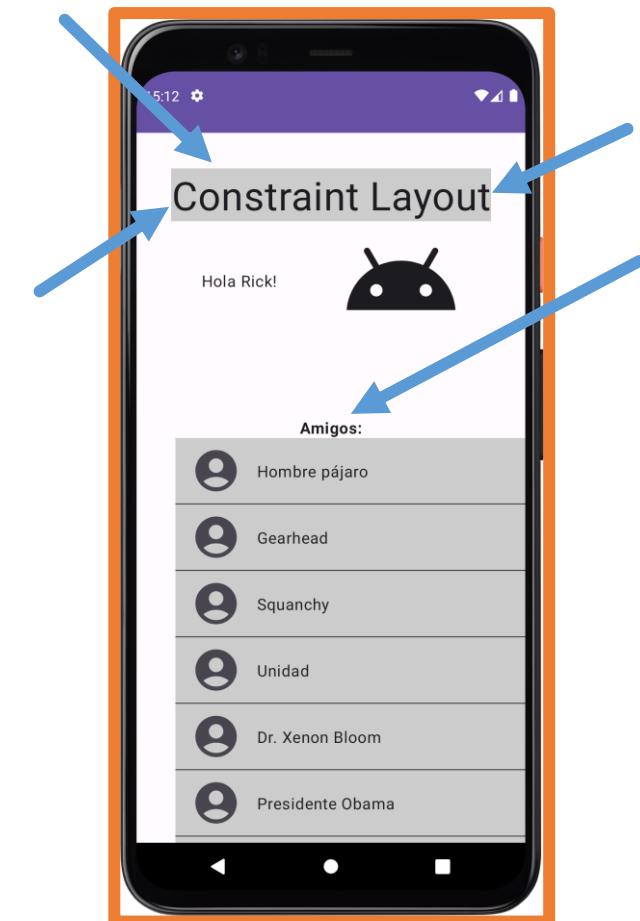
Dentro de **ConstraintLayout** se pueden crear **guías** para poder unir los componentes a dichas guías.

Las guías se pueden indicar o con **porcentaje** (float de 0 a 1) con directamente con **dp**.

```
ConstraintLayout(modifier = Modifier.fillMaxSize()) { this: ConstraintLayoutScope
    val titleRef = createRef()
    val (iconRef, greetingRef, friendsTitleRef, friendsListRef) = createRefs()
    val topGuide = createGuidelineFromTop(fraction: 0.05f)
    val startGuide = createGuidelineFromStart(fraction: 0.1f)
    val endGuide = createGuidelineFromEnd(fraction: 0.1f)
    val friendsGuide = createGuidelineFromTop(fraction: 0.4f)

    Text(
        text = "Constraint Layout",
        fontSize = 40.sp,
        textAlign = TextAlign.Center,
        modifier = Modifier
            .background(Color.LightGray)
            .constrainAs(titleRef) { this: ConstrainScope
                top.linkTo(topGuide)
                start.linkTo(startGuide)
                end.linkTo(endGuide)
            }
    )
    Icon(
        imageVector = Icons.TwoTone.Android,
        contentDescription = "Android",
        modifier = Modifier.size(120.dp).constrainAs(iconRef) { this: ConstrainScope
            top.linkTo(titleRef.bottom)
            start.linkTo(greetingRef.end)
            end.linkTo(parent.end)
        }
    )
    Text(
        text = "Hola Rick!",
        modifier = Modifier.constrainAs(greetingRef) { this: ConstrainScope
            top.linkTo(iconRef.top)
            bottom.linkTo(iconRef.bottom)
            start.linkTo(parent.start)
            end.linkTo(iconRef.start)
        }
    )
}
```

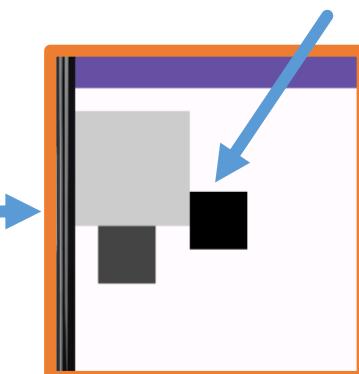
```
Text(
    text = "Amigos:",
    fontWeight = FontWeight.Bold,
    modifier = Modifier.constrainAs(friendsTitleRef) { this: ConstrainScope
        top.linkTo(friendsGuide)
        start.linkTo(parent.start)
        end.linkTo(parent.end)
    }
)
Column(
    modifier = Modifier
        .constrainAs(friendsListRef) { this: ConstrainScope
            top.linkTo(friendsTitleRef.bottom)
            start.linkTo(startGuide)
        }
) { this: ColumnScope
    listof("Hombre pájaro", "Gearhead", "Squanchy", "Unidad",
        "Dr. Xenon Bloom", "Presidente Obama", "Jaguar").forEach { it: String
        ListItem(
            headlineContent = { Text(it) },
            leadingContent = {
                Icon(
                    imageVector = Icons.Default.AccountCircle,
                    contentDescription = "Destino: $it",
                    modifier = Modifier.size(50.dp)
                )
            },
            colors = ListItemDefaults.colors(
                containerColor = Color.LightGray
            )
        )
        Divider(color = Color.DarkGray)
    }
}
```



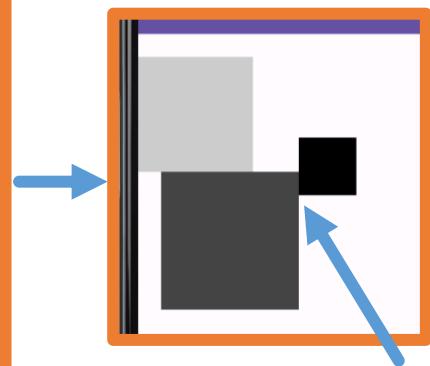
20.- Componente ConstraintLayout

También se pueden crear **barreras** con dos o más componentes que se usarán como guía para que otros componentes se unan a ellas.

```
ConstraintLayout(modifier = Modifier.fillMaxSize()) { this: ConstraintLayoutScope
    val (lightBox, darkBox, blackBox) = createRefs()
    val barrier = createEndBarrier(lightBox, darkBox)
    Box(
        modifier = Modifier.background(Color.LightGray)
            .size(100.dp)
            .constrainAs(lightBox) { this: ConstrainScope
                top.linkTo(parent.top, 20.dp)
            }
    )
    Box(
        modifier = Modifier.background(Color.DarkGray)
            .size(50.dp)
            .constrainAs(darkBox) { this: ConstrainScope
                top.linkTo(lightBox.bottom)
                start.linkTo(lightBox.start, 20.dp)
            }
    )
    Box(
        modifier = Modifier.background(Color.Black)
            .size(50.dp)
            .constrainAs(blackBox) { this: ConstrainScope
                top.linkTo(parent.top, 90.dp)
                start.linkTo(barrier)
            }
    )
}
```



```
ConstraintLayout(modifier = Modifier.fillMaxSize()) { this: ConstraintLayoutScope
    val (lightBox, darkBox, blackBox) = createRefs()
    val barrier = createEndBarrier(lightBox, darkBox)
    Box(
        modifier = Modifier.background(Color.LightGray)
            .size(100.dp)
            .constrainAs(lightBox) { this: ConstrainScope
                top.linkTo(parent.top, 20.dp)
            }
    )
    Box(
        modifier = Modifier.background(Color.DarkGray)
            .size(120.dp)
            .constrainAs(darkBox) { this: ConstrainScope
                top.linkTo(lightBox.bottom)
                start.linkTo(lightBox.start, 20.dp)
            }
    )
    Box(
        modifier = Modifier.background(Color.Black)
            .size(50.dp)
            .constrainAs(blackBox) { this: ConstrainScope
                top.linkTo(parent.top, 90.dp)
                start.linkTo(barrier)
            }
    )
}
```



21.- Componentes Modal

Los componentes **Modal** son componentes que aparecen sobre la pantalla, existen varios: `ModalNavigationDrawer`, `ModalBottomSheet`, `ModalSideSheet`...

El más conocido es **ModalNavigationDrawer** también conocido como **menú hamburguesa** que al pulsarlo se abre un menú lateral.



En dispositivos con pantalla grande se debe sustituir este menú por un otros que estén siempre visible como un **NavigationDrawer** o un **NavigationRail**.

A continuación, se muestra un ejemplo de su uso para entender su funcionamiento, pero algunos conceptos que se utilizan se estudiarán más adelante como las corrutinas y la navegación.

21.- Componentes Modal

```
// Clase con los objetos que servirán para crear el menú
sealed class Options(val title: String, val icon: ImageVector) {
    object Option1: Options( title: "Inicio", Icons.Default.Home)
    object Option2: Options( title: "Editar", Icons.Default.Edit)
    object Option3: Options( title: "Configuración", Icons.Default.Settings)
    object Option4: Options( title: "Ayuda", Icons.Default.Help)
}

@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun Content() {
    val options = listOf(Options.Option1, Options.Option2, Options.Option3, Options.Option4)
    val drawerState = rememberDrawerState(DrawerValue.Closed)
    var selectedOption by rememberSaveable { mutableStateOf(options[0].title) }
    val scope = rememberCoroutineScope()
    Scaffold(topBar = {
        TopAppBar(
            title = { Text(selectedOption) },
            navigationIcon = {
                IconButton(onClick = {
                    if (drawerState.isOpen) scope.launch { drawerState.close() }
                    else scope.launch { drawerState.open() }
                }) {
                    Icon(
                        imageVector = Icons.Default.Menu,
                        contentDescription = "Menú"
                    )
                }
            },
            colors = TopAppBarDefaults.topAppBarColors(containerColor = Color.LightGray)
        )
    }) { it: PaddingValues -->

```

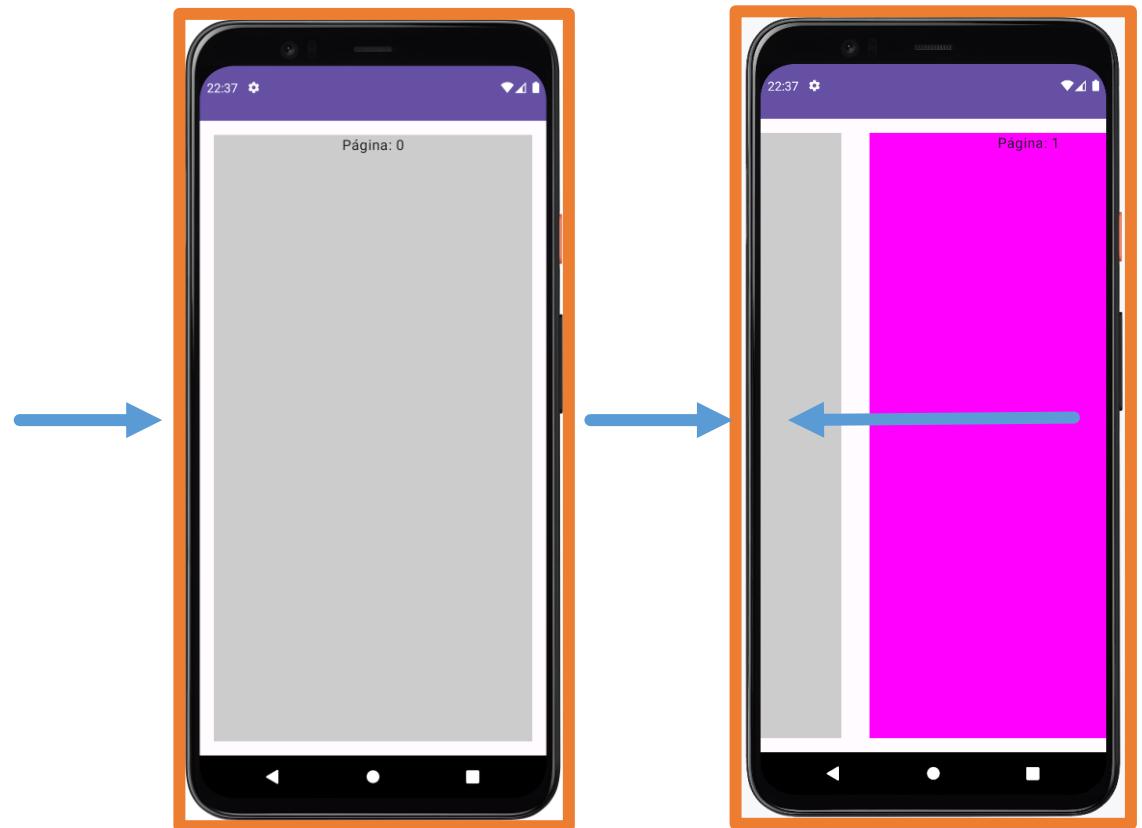
```
        } { it: PaddingValues -->
            ModalNavigationDrawer(drawerState = drawerState,
                gesturesEnabled = false,
                drawerContent = {
                    ModalDrawerSheet { this: ColumnScope -->
                        Spacer(Modifier.height(12.dp))
                        options.forEach { option ->
                            NavigationDrawerItem(
                                icon = {
                                    Icon(option.icon, contentDescription = option.title)
                                },
                                label = { Text(option.title) },
                                selected = option.title == selectedOption,
                                onClick = {
                                    scope.launch { drawerState.close() }
                                    selectedOption = option.title
                                },
                                modifier = Modifier.padding(NavigationDrawerItemDefaults.ItemPadding)
                            )
                        }
                    },
                    modifier = Modifier.padding(it), // Importante para que se visualice correctamente
                    content = {
                        // Cargar contenido según opción elegida: selectedOption
                    }
                }
            )
        }
    }
}
```



22.- Componentes HorizontalPager y VerticalPager

Mediante los componentes **HorizontalPager** y **VerticalPager** se pueden crear diferentes pantallas y navegar por ellas deslizando el dedo por la pantalla.

```
val colors = listOf(Color.LightGray, Color.Magenta, Color.Cyan, Color.Yellow, Color.White)
val pagerState = rememberPagerState( initialPage: 0 )
HorizontalPager(
    pageCount = 5,
    state = pagerState
) { page ->
    Text(
        text = "Página: $page",
        textAlign = TextAlign.Center,
        modifier = Modifier
            .padding(16.dp)
            .background(colors[page])
            .fillMaxSize()
    )
}
```



22.- Componentes HorizontalPager y VerticalPager

Modificando el estado del Pager se puede navegar entre las páginas con botones.

```
val colors = listOf(Color.LightGray, Color.Magenta, Color.Cyan, Color.Gray)
val characters = listOf("Rick Sanchez", "Morty Smith", "Summer Smith", "HombrePájaro")
val pagerState = rememberPagerState( initialPage: 0 )
val coroutineScope = rememberCoroutineScope()
Column { this: ColumnScope
    HorizontalPager(
        pageCount = colors.size,
        state = pagerState,
        modifier = Modifier.weight(9.5f)
    ) { page ->
        Text(
            text = characters[page],
            textAlign = TextAlign.Center,
            modifier = Modifier
                .background(colors[page])
                .padding(20.dp)
                .fillMaxSize()
        )
    }
    Row(
        horizontalArrangement = Arrangement.Center,
        modifier = Modifier
            .fillMaxWidth()
            .weight(.5f)
    ) { this: RowScope
        Icon(
            imageVector = Icons.Default.FirstPage,
            contentDescription = "Primero",
            modifier = Modifier.clickable {
                if (pagerState.currentPage != 0) {
                    coroutineScope.launch { this: CoroutineScope
                        pagerState.scrollToPage( page: 0 )
                    }
                }
            }
        )
    }
}
```

```
Icon(
    imageVector = Icons.Default.NavigateBefore,
    contentDescription = "Anterior",
    modifier = Modifier.clickable {
        if (pagerState.currentPage != 0) {
            coroutineScope.launch { this: CoroutineScope
                pagerState.scrollToPage( page: pagerState.currentPage-1 )
            }
        }
    }
)
Icon(
    imageVector = Icons.Default.NavigateNext,
    contentDescription = "Siguiente",
    modifier = Modifier.clickable {
        if (pagerState.currentPage != characters.size-1) {
            coroutineScope.launch { this: CoroutineScope
                pagerState.scrollToPage( page: pagerState.currentPage+1 )
            }
        }
    }
)
Icon(
    imageVector = Icons.Default.LastPage,
    contentDescription = "Último",
    modifier = Modifier.clickable {
        if (pagerState.currentPage != characters.size-1) {
            coroutineScope.launch { this: CoroutineScope
                pagerState.scrollToPage( page: characters.size-1 )
            }
        }
    }
)
```

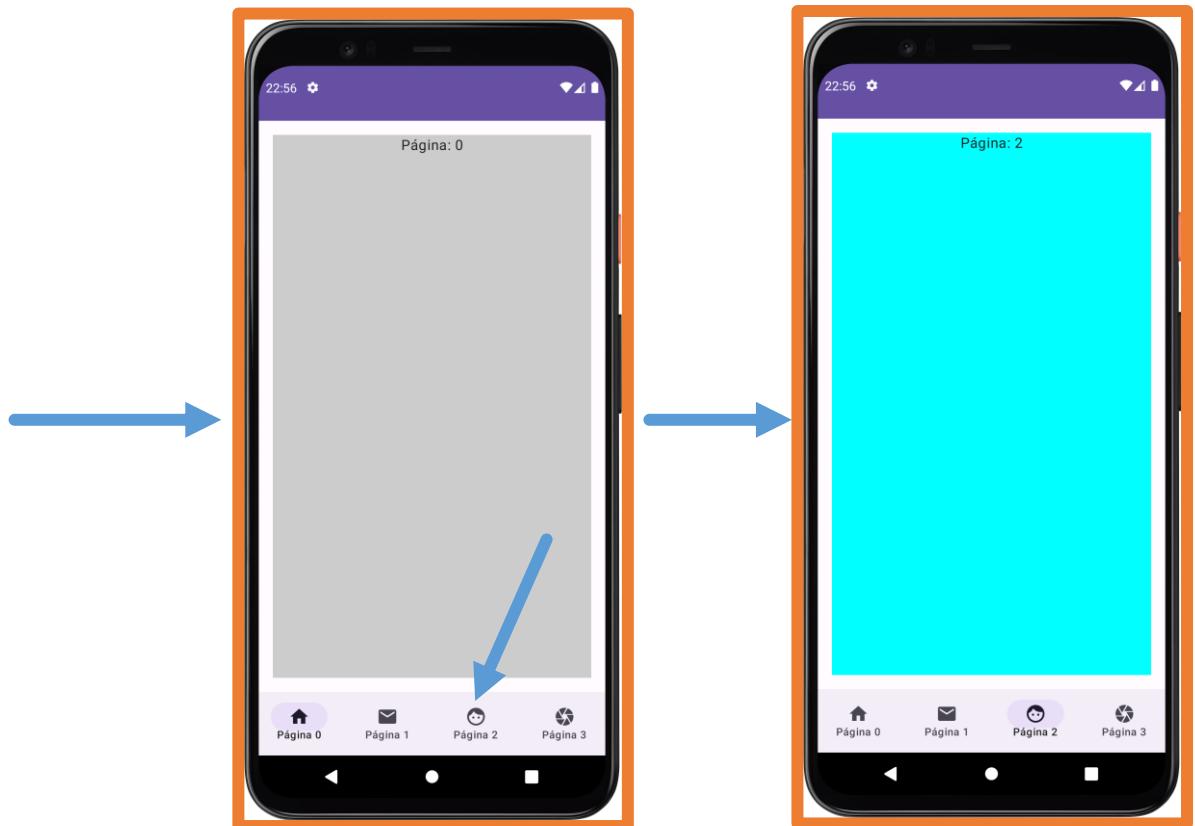


Se puede usar animateScrollToPage para que el cambio de página esté animado.

22.- Componentes HorizontalPager y VerticalPager

HorizontalPager y VerticalPager se pueden combinar con botones e incluso desactivar los gestos y controlarlos con botones.

```
Column { this: ColumnScope
    val colors = listOf(Color.LightGray, Color.Magenta, Color.Cyan, Color.Yellow)
    val icons = listOf(Icons.Default.Home, Icons.Default.Mail, Icons.Default.Face, Icons.Default.Camera)
    val pagerState = rememberPagerState( initialPage: 0 )
    val scope = rememberCoroutineScope()
    HorizontalPager(
        pageCount = 4,
        state = pagerState,
        modifier = Modifier.weight(0.9f)
    ) { page ->
        Text(
            text = "Página: $page",
            textAlign = TextAlign.Center,
            modifier = Modifier
                .padding(16.dp)
                .background(colors[page])
                .fillMaxSize()
        )
    }
    NavigationBar(
        modifier = Modifier.weight(0.1f)
    ) { this: RowScope
        colors.forEachIndexed { index, color ->
            NavigationBarItem(
                selected = index == pagerState.currentPage,
                onClick = {
                    scope.launch { this: CoroutineScope
                        pagerState.animateScrollToPage(index)
                    }
                },
                label = { Text( text: "Página $index" ) },
                icon = { Icon(
                    imageVector = icons[index],
                    contentDescription = icons[index].name
                ) })
        }
    }
}
```



23.- Componentes PlainTooltipBox y RichTooltipBox

Los **Tooltips** son pequeños mensajes que aparecen cuando se realiza una pulsación larga sobre un elemento.

Cualquier componente puede tener un Tooltip.

Para poder usar estos componentes se deben actualizar algunas versiones:

build.gradle.kts (Project):

Kotlin → 1.8.21

build.gradle.kts (Module):

kotlinCompilerExtensionVersion → 1.4.7

compose-bom → 2023.05.01

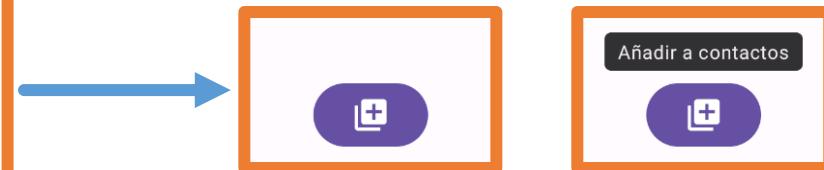
core-ktx → 1.10.1

Recuerda sincronizar tras realizar cambios en los archivos **gradle**.

23.- Componentes PlainTooltipBox y RichTooltipBox

Para añadir un Tooltip a un componente se debe envolver dicho componente en un **PlainTooltipBox** o un **RichTooltipBox** y añadir al componente envuelto el Modifier `tooltipAnchor`.

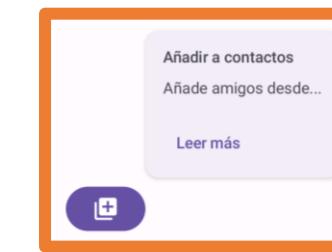
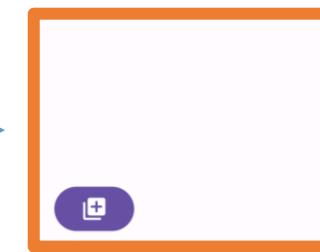
```
PlainTooltipBox(  
    tooltip = {  
        Text( text: "Añadir a contactos")  
    }  
) { this: TooltipBoxScope  
    Button(  
        onClick = { /*TODO*/ },  
        modifier = Modifier.tooltipAnchor()  
    ) { this: RowScope  
        Icon(  
            imageVector = Icons.Default.AddToPhotos,  
            contentDescription = "Añadir a contactos"  
        )  
    }  
}
```



23.- Componentes PlainTooltipBox y RichTooltipBox

Los **RichTooltipBox** permiten mostrar más información e incluso mostrar acciones. Necesitan un estado para mantener abiertos y una corutina para cerrarlo.

```
val tooltipState by remember { mutableStateOf(RichTooltipState()) }
val scope = rememberCoroutineScope()
RichTooltipBox(
    title = { Text(text: "Añadir a contactos") },
    action = {
        TextButton(
            onClick = {
                scope.launch { tooltipState.dismiss() }
            }
        ) { this: RowScope
            Text(text: "Leer más")
        }
    },
    text = { Text(text: "Añade amigos desde...") },
    tooltipState = tooltipState
) { this: TooltipBoxScope
    Button(
        onClick = { /*TODO*/ },
        modifier = Modifier.tooltipAnchor()
    ) { this: RowScope
        Icon(
            imageVector = Icons.Default.AddToPhotos,
            contentDescription = "Añadir a contactos"
        )
    }
}
```



Teniendo el estado se puede lanzar el RichTooltipBox siempre que se quiera, por ejemplo al pulsar un botón para mostrar la información siempre la primera vez que se pulsa el botón.

```
scope.launch { tooltipState.show() }
```

Práctica

Actividad 4

Dragon Ball