Práctica 04.

¿Por qué necesitamos usar RecyclerView si ya tenemos otros componentes que hacen filas y columnas?

Cuando creamos filas y columnas lo que hace Android es crear TODAS las vistas directamente, pero no carga todo el resto del contenido. Por eso usamos RecyclerView, ya que es mucho más óptimo para el dispositivo móvil porque necesita menos memoria.

RecyclerView facilita que se muestren de manera eficiente grandes conjuntos de datos. Tú proporcionas los datos y defines el aspecto de cada elemento, y la biblioteca RecyclerView creará los elementos de forma dinámica cuando se los necesite.

Como su nombre lo indica, RecyclerView recicla esos elementos individuales. Cuando un elemento se desplaza fuera de la pantalla, RecyclerView no destruye su vista. En cambio, reutiliza la vista para los elementos nuevos que se desplazaron y ahora se muestran en pantalla. Esto mejora en gran medida el rendimiento y la capacidad de respuesta de tu app y reduce el consumo de energía.

Listas y cuadrículas en Jetpack Compose

Todos los tipos de RecyclerView solo reciben `items`… objetos de tipo `items`. Todos nuestros componentes deben ir dentro del elemento item.

- LazyRow()
- LazyColumn()
- LazyVerticalGrid()

Recomendaciones de Android para el buen uso de RV:

- No usar dentro componentes con tamaño 0 y que después se redimensione. Por ejemplo una imagen que al inicio tenga tamaño 0.
- Evitar introducir componentes deslizables en la misma dirección.

Ejemplo 1.

Este primer ejemplo es muy sencillo, simplemente utiliza el contenedor ***LazyColumn***, dónde dentro hemos comentado antes que solo podemos usar objetos de tipo ***item***. De manera individual o varios items del mismo tipo, repitiendo el mismo componente o utilizando una colección, por ejemplo una lista.

```
@Composable
fun SimpleRecyclerView() {
    val myList = listOf("Marta", "Pepe", "Manolo", "Jaime")
    LazyColumn {
        item { Text(text = "Header") }
        items(3) {
            Text(text = "Este es el item $it")
        }
        items(myList) {
            Text(text = "Hola me llamo $it")
        }
        item { Text(text = "Footer") }
    }
}
```

Ejemplo 2.

Vamos a realizar una función Composable un poco más elaborada con una lista de superhéroes.

1. Decarga las imágenes de los super héroes que están adjuntas en la tarea e insertarlas en los recursos.
2. Crea un ***data class*** con el nombre ***Superhero*** que tendrá las siguientes propiedades:

- *superheroName*: String
- *realName*: String
- *publisher*: String
- *photo*: Int *(utilizad la anotación del constructor DrawableRes para indicar que este entero retorna una referencia a un recurso de tipo drawable → @DrawableRes var photo: Int)*

3. Crear un fichero de Kotlin que se llame ***SuperHero.kt*** dónde insertaremos nuestras vistas.
4. Crear una función normal que retorne una lista de elementos de tipo Superhero *(utilizad las referencias a las imágenes en la propiedad photo)*. La función se podría

llamar getSuperheroes() y los datos de sus elementos serían los siguientes superhéroes:

- "Spiderman", "Petter Parker", "Marvel", R.drawable.spiderman
- "Wolverine", "James Howlett", "Marvel", R.drawable.logan
- "Batman", "Bruce Wayne", "DC", R.drawable.batman
- "Thor", "Thor Odinson", "Marvel", R.drawable.thor
- "Flash", "Jay Garrick", "DC", R.drawable.flash
- "Green Lantern", "Alan Scott", "DC", R.drawable.green_lantern
- "Wonder Woman", "Princess Diana", "DC", R.drawable.wonder_woman

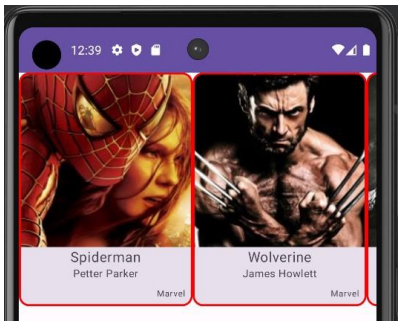
5. Insertar dos funciones Composable, una llamada ***SuperHeroView()*** montaremos nuestro RecyclerView y otra ***ItemHero()***, que tendrá un parámetro de entrada de tipo Superhero y dónde diseñaremos el layout de cada item.

```
@Composable
fun SuperHeroView() {
    LazyRow() {
        items(getSuperheroes()) {
            ItemHero(superhero = it)
        }
    }
}
```

O también podemos darle un nombre diferente al argumento que recibe nuestra función ItemHero():

```
@Composable
fun SuperHeroView() {
    LazyRow() {
        items(getSuperheroes()) { superhero ->
            ItemHero(superhero = superhero)
        }
    }
}
```

6. Desarrollar vosotros la función ItemHero() para que tenga un aspecto cómo el siguiente:



Para ello insertad la función

```
@Composable
fun ItemHero(superhero: Superhero) {
}
```

Utilizad un componente de tipo Card con un borde de 2dp color rojo y un tamaño de 200dp.

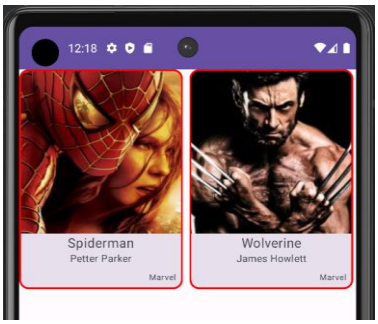
Dentro, incluye en una columna los componentes necesarios para mostrar la info del superhéroe que recibe por parámetro.

Para la imagen utiliza las descripción del contenido “SuperHero Avatar”, la función painterResource() para recuperar el recurso, que coja el ancho máximo y la opción de escalado ***contentScale = ContentScale.Crop***

Introduce también un padding al último texto para que no esté pegado a los bordes del Card.

Para separar los items del LazyRow puedes usar la opción

```
horizontalArrangement = Arrangement.spacedBy(8.dp)
```

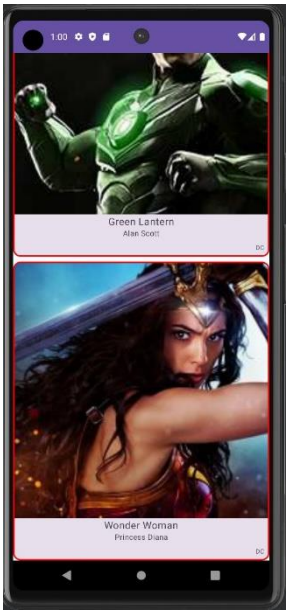


7. Cambiad ahora LazyRow por LazyColumn.

Haced ahora que el componente Card reaccione al evento click y la app muestre un mensaje con Toast con el nombre real del superhéroe. Para esto debemos incluir en el Modifier la opción clickable { } dónde llamaremos a una función lambda que recibirá como argumento un objeto de tipo Superhero.

```
fun ItemHero(superhero: Superhero, onItemSelected: (Superhero) ->
Unit) {
    Card(
        border = BorderStroke(2.dp, Color.Red),
        modifier = Modifier
            .width(200.dp)
            .clickable { onItemSelected(superhero) }
    ) {
…
```

Ahora solo tenéis que usar Toast para mostrar el nombre real del superhéroe en la función SuperHeroView()… Modificad también el tamaño del Card para que ocupe todo el ancho de la pantalla:



8. Duplicar la función Composable SuperHeroView() e intentad ahora cambiar LazyColumn por LazyVerticalGrid… investigad y jugad con los parámetros para intentar conseguir lo siguiente:

Usa en el parámetro columns GridCells.Fixed() y GridCells.Adaptive() para ajustar las celdas.

Para dar un padding alrededor de todos los items se utiliza el parámetro contentPadding = PaddingValues(horizontal = 4.dp, vertical = 4.dp). Si queremos que el interior de las elementos también tengan una separación usaremos el modificador padding del componente Card. Si además usamos en esta ocasión un GridCells.Adaptive(130.dp) la app quedaría como la siguiente imagen: