

Jetpack Compose

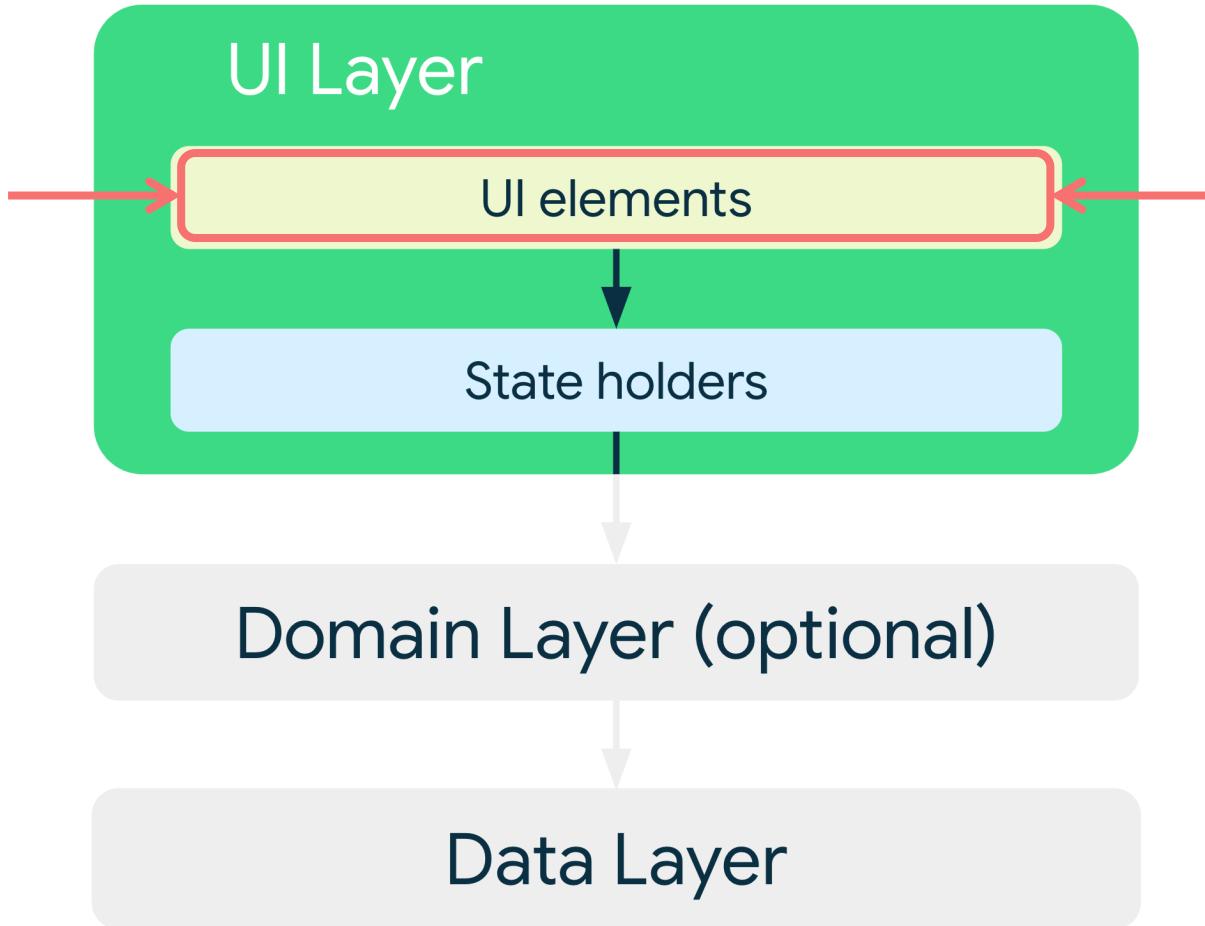
Elementi di base

Laboratorio 0

Esercitazioni di laboratorio

- Le esercitazioni riguarderanno gli argomenti trattati a lezione (più o meno)
- Ogni esercitazione è composta da:
 - esercizi svolti insieme
 - esercizi da svolgere individualmente
- Tranne quella introduttiva di oggi che, essendo la prima, sarà tutta interattiva

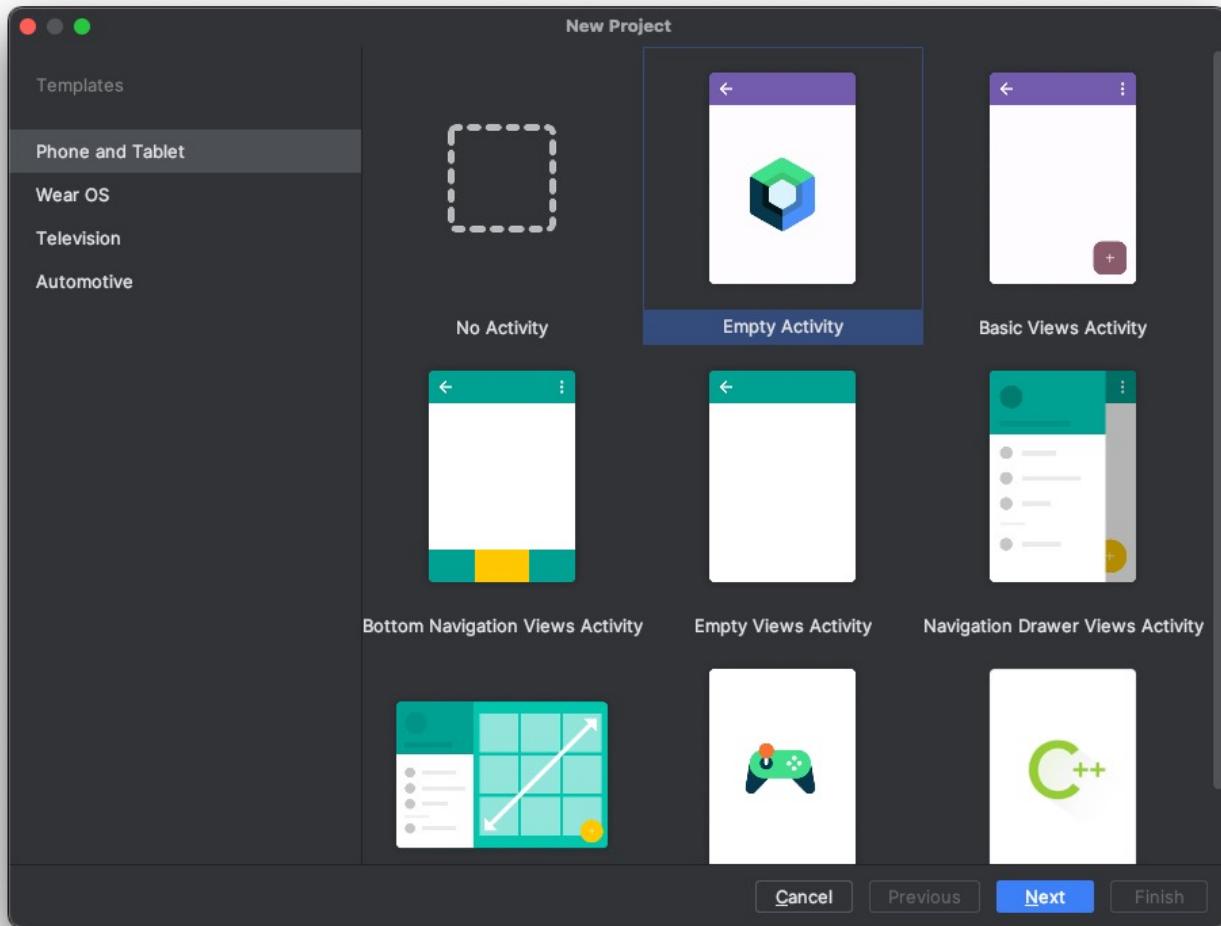
Focus di questa esercitazione



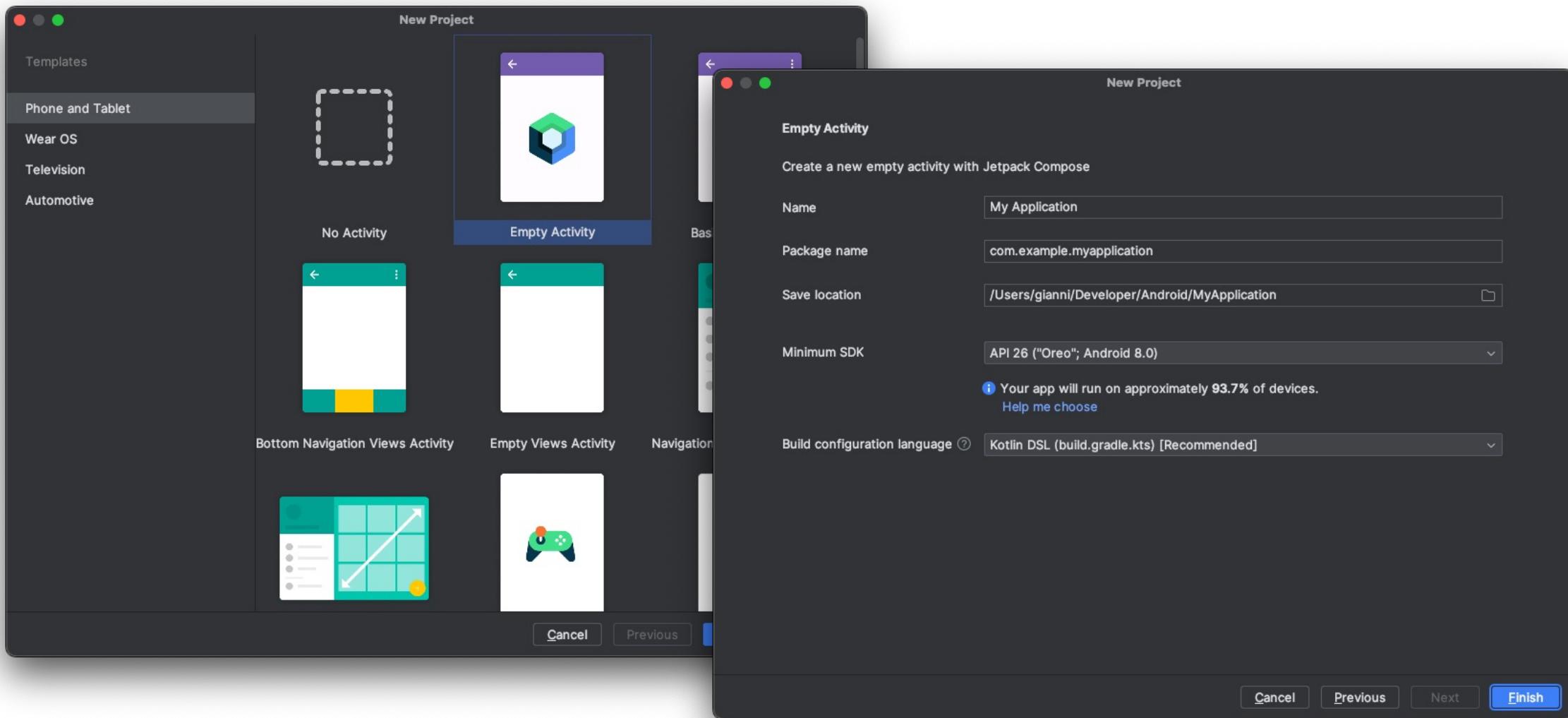
Esercizi

1. Creare un nuovo progetto su Android Studio
2. Cambiare il colore di sfondo
3. Cambiare le dimensioni della view
4. Rilevare l'orientamento del dispositivo
5. Cambiare lo stile del testo
6. Creare una riga di elementi testuali
7. Creare una colonna di elementi testuali
8. Rilevare le dimensioni del container
9. Gestire contenuti dinamici
10. Creare una lista con scroll

1. Creare un nuovo progetto su Android Studio

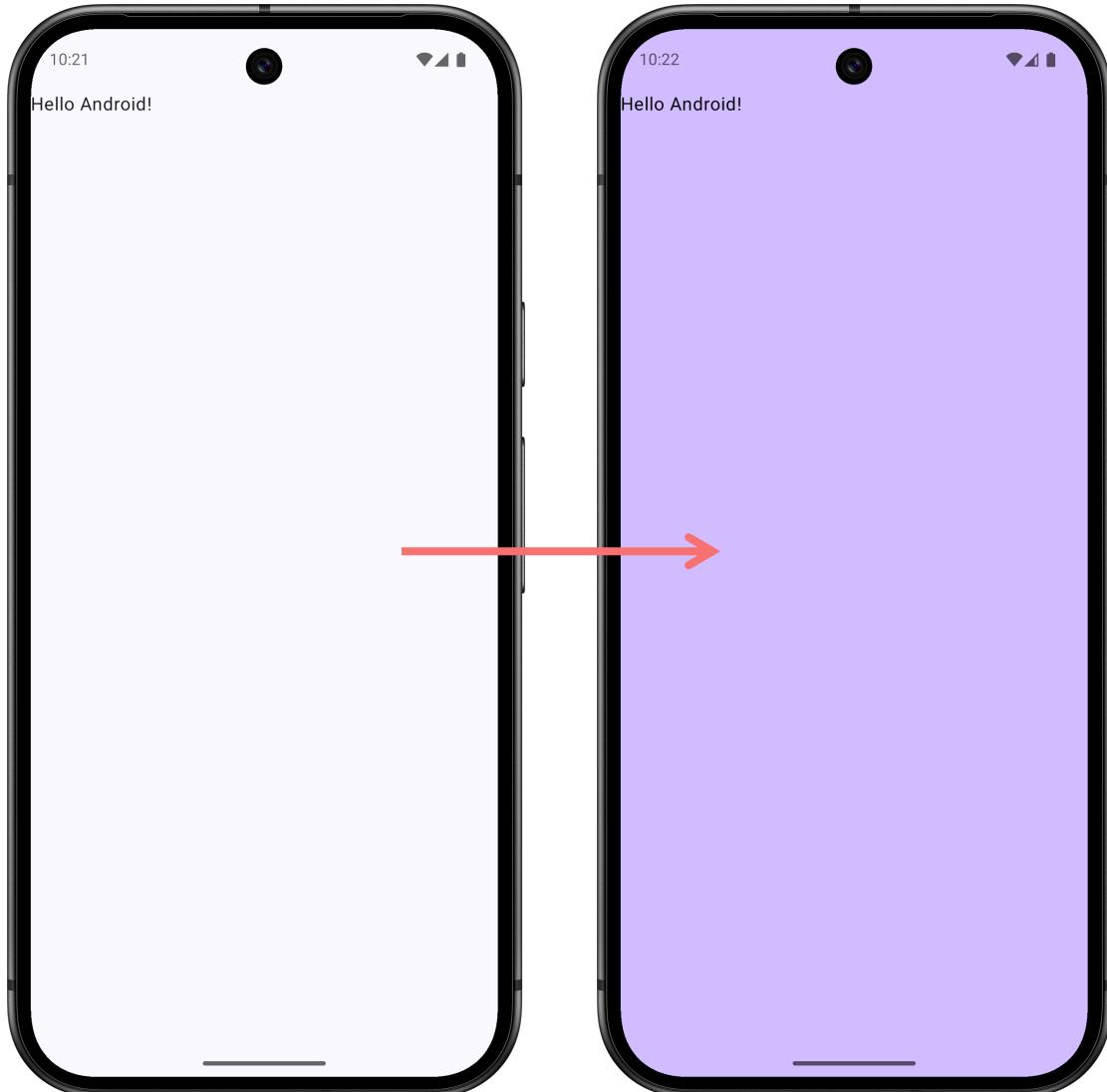


1. Creare un nuovo progetto su Android Studio



2. Cambiare il colore di sfondo

- Modificare il colore di sfondo dell'app in violetto 
- **Hints**
 - **Scaffold** accetta un parametro **containerColor**
 - Controllare se c'è già una variabile con il colore desiderato



ui.theme

- Il package **ui.theme** contiene il tema della nostra applicazione
 - **Color.kt** contiene l'elenco dei colori creati manualmente dallo sviluppatore
 - **Type.kt** contiene gli stili tipografici
 - **Theme.kt** contiene un composable che applica il tema all'intera app

2. Prima soluzione

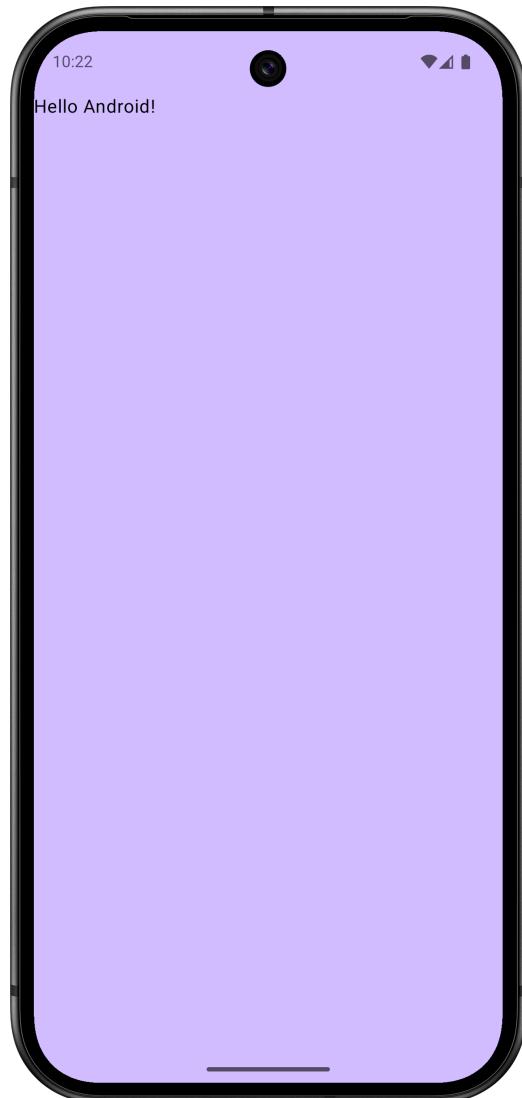
Color.kt

```
val Purple80 = Color(0xFFD0BCFF)
val PurpleGrey80 = Color(0xFFCCC2DC)
val Pink80 = Color(0xFFEB8C8)

val Purple40 = Color(0xFF6650a4)
val PurpleGrey40 = Color(0xFF625b71)
val Pink40 = Color(0xFF7D5260)
```

MainActivity.kt

```
Scaffold(
    containerColor = Purple80,
    modifier = Modifier.fillMaxSize()
)
```



Scaffold ed edge-to-edge

- Lo **Scaffold** occupa l'intero schermo, incluse status bar e navigation bar
 - Questo anche grazie alla modalità edge-to-edge abilitata nella **MainActivity** tramite `enableEdgeToEdge()`
- E se volessimo modificare solo il colore del contenuto, escludendo top bar e navigation bar?

2. Soluzione con Surface

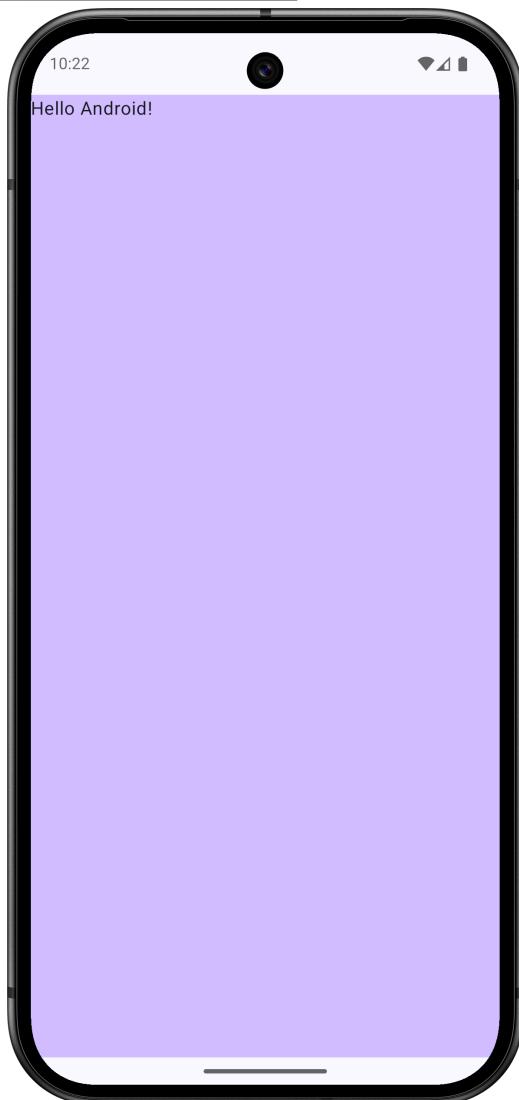
Color.kt

```
val Purple80 = Color(0xFFD0BCFF)
val PurpleGrey80 = Color(0xFFCC2DC)
val Pink80 = Color(0xFFEB8C8)

val Purple40 = Color(0xFF6650a4)
val PurpleGrey40 = Color(0xFF625b71)
val Pink40 = Color(0xFF7D5260)
```

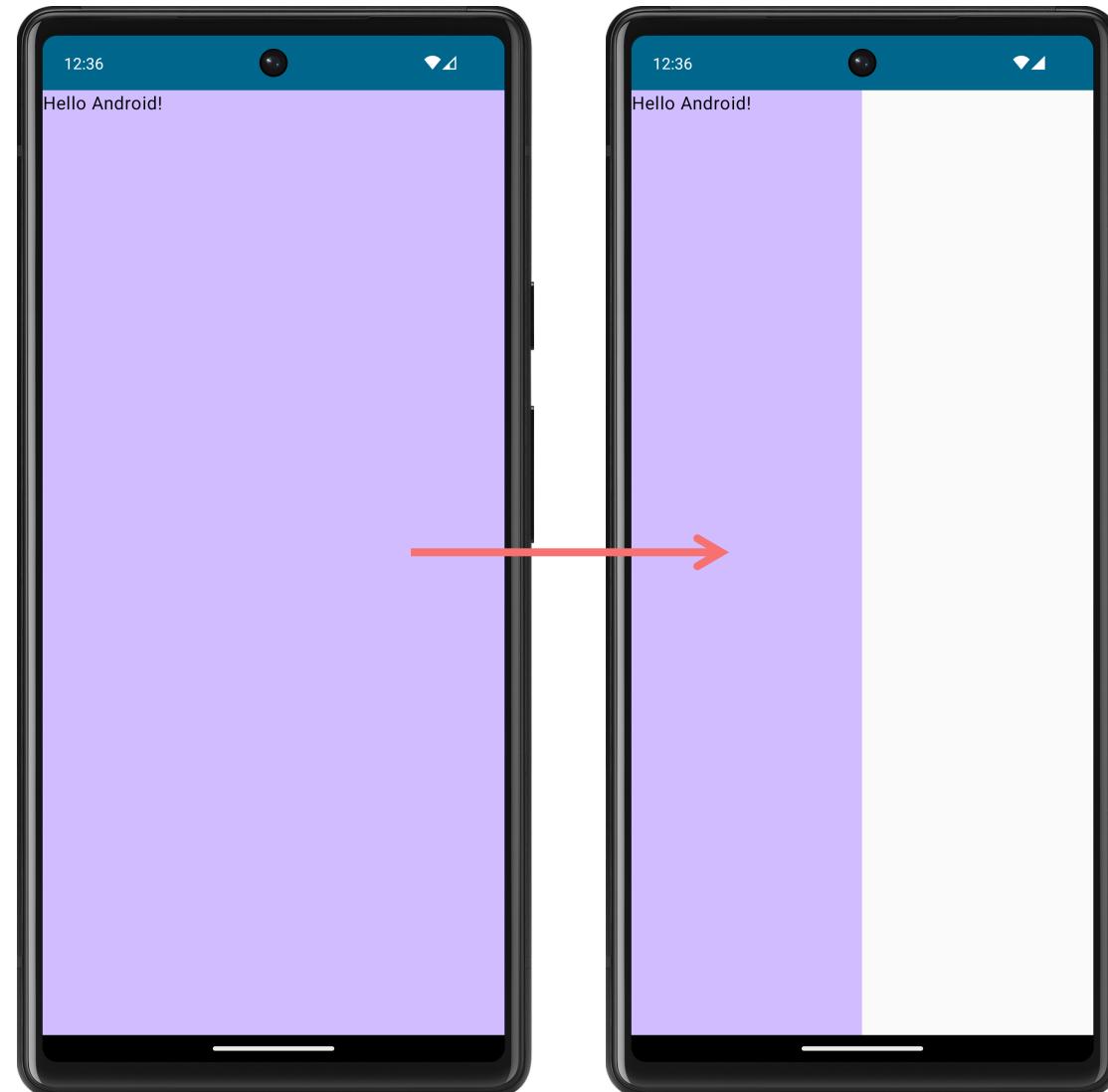
MainActivity.kt

```
Scaffold(
    modifier = Modifier.fillMaxSize()
) { innerPadding ->
    Surface(
        color = Purple80,
        modifier = Modifier.padding(innerPadding).fillMaxSize()
    ) {
        // ...
    }
}
```



3. Cambiare le dimensioni della surface

- Modificare la dimensione della surface in modo che occupi metà della larghezza del dispositivo e tutta l'altezza
- **Hints**
 - Utilizzare **LocalConfiguration.current** per recuperare informazioni sul dispositivo
 - Utilizzare un **modifier** per impostare le dimensioni

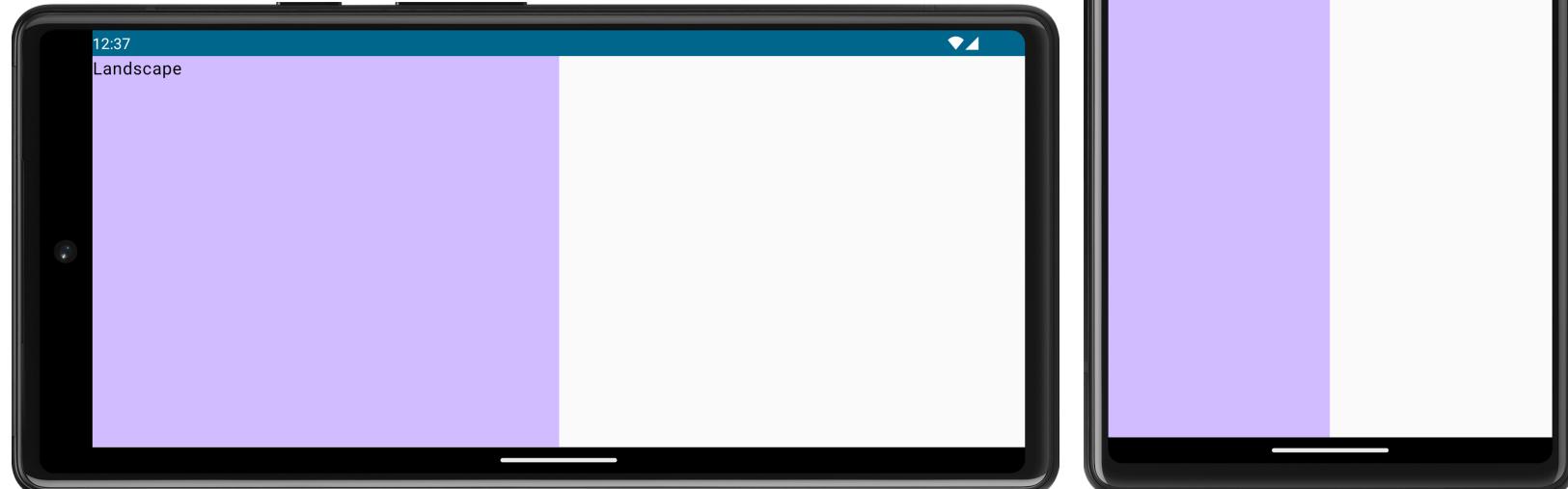


3. Soluzione

```
Surface(  
    color = Purple80,  
    modifier = Modifier.size(  
        LocalConfiguration.current.screenWidthDp.dp / 2,  
        LocalConfiguration.current.screenHeightDp.dp  
    )  
) {
```

4. Rilevare l'orientamento del dispositivo

- Creare un composable che mostri il testo “Portrait” quando il dispositivo è in verticale e “Landscape” quando è in orizzontale
- **Hint**
 - Anche l’orientamento del dispositivo può essere reperito tramite **LocalConfiguration.current**



4. Soluzione

```
@Composable
fun Orientation(config: Configuration) {
    Text(when(config.orientation) {
        Configuration.ORIENTATION_PORTRAIT -> "Portrait"
        Configuration.ORIENTATION_LANDSCAPE -> "Landscape"
        else -> "Unknown"
    })
}

val config = LocalConfiguration.current
// A surface container using the 'background' color from the theme
Surface(
    color = Purple80,
    modifier = Modifier.size(
        config.screenWidthDp.dp / 2,
        config.screenHeightDp.dp
    )
) {
    // Greeting("Android")
    Orientation(config)
}
```

4. Soluzione

```
@Composable
fun Orientation(config: Configuration) {
    Text(when(config.orientation) {
        Configuration.ORIENTATION_PORTRAIT -> "Portrait"
        Configuration.ORIENTATION_LANDSCAPE -> "Landscape"
        else -> "Unknown"
    })
}
```

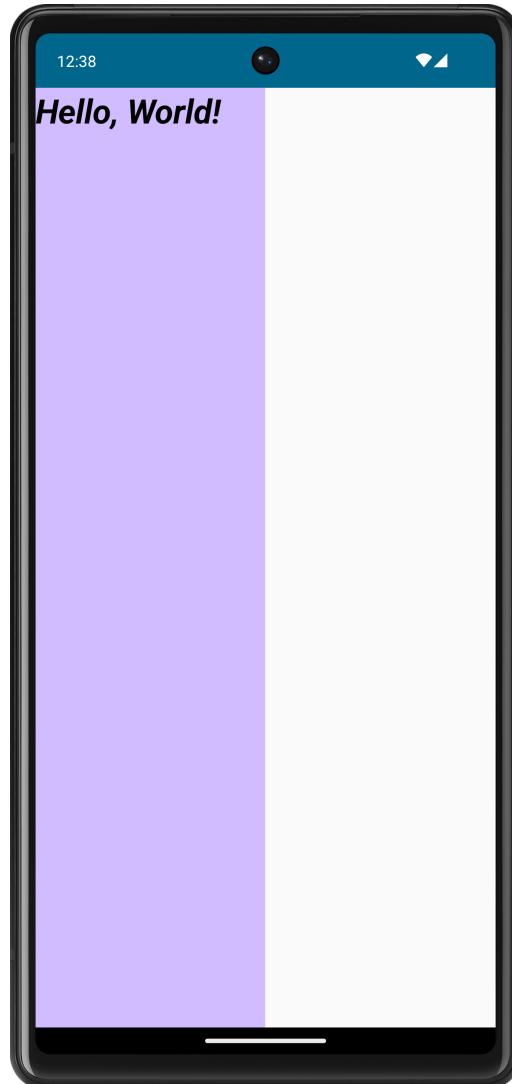
Possiamo salvarlo
in una variabile e
passarlo come
parametro al
composable

```
val config = LocalConfiguration.current
// A surface container using the 'background' color from the theme
Surface(
    color = Purple80,
    modifier = Modifier.size(
        config.screenWidthDp.dp / 2,
        config.screenHeightDp.dp
    ) {
        // Greeting("Android")
        Orientation(config)
    }
)
```

LocalConfiguration.current è
richiamabile solo all'interno di
una funzione @Composable

5. Cambiare lo stile del testo

- Creare un composable che mostri un testo con le seguenti caratteristiche:
 - Dimensione del font: 30 sp
<https://developer.android.com/guide/topics/resources/more-resources.html#Dimension>
 - Grassetto
 - Italico

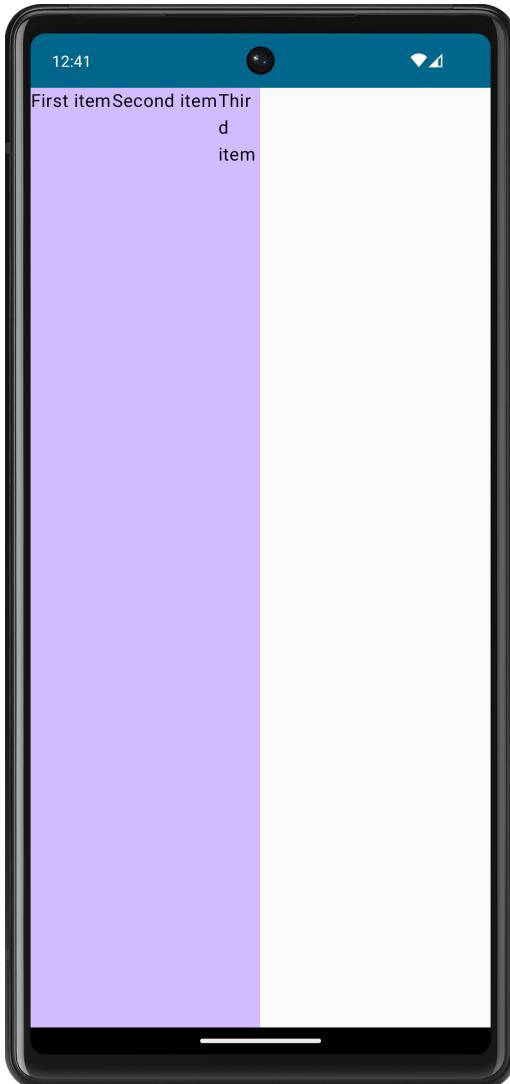


5. Soluzione

```
@Composable
fun TextStyle() {
    Text(
        "Hello, World!",
        fontSize = 30.sp,
        fontWeight = FontWeight.Bold,
        fontStyle = FontStyle.Italic
    )
}
```

6. Creare una riga di elementi testuali

- Creare un composable che disponga tre elementi testuali uno accanto all'altro



6. Primo tentativo

- Inseriamo tre elementi **Text** senza alcun container

```
@Composable
fun HorizontalLayout() {
    Text("First item")
    Text("Second item")
    Text("Third item")
}
```

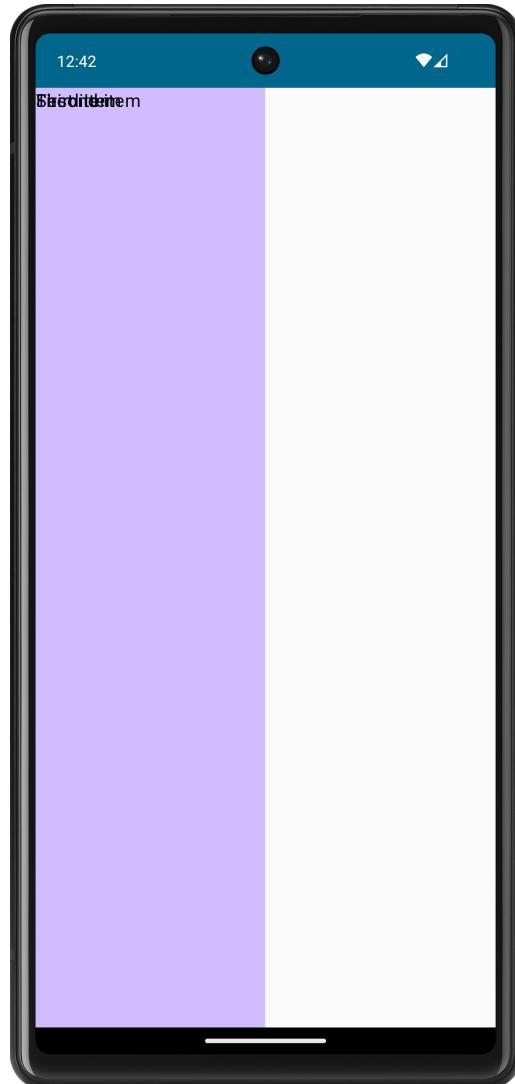
- Cosa succede?

6. Primo tentativo

- Inseriamo tre elementi **Text** senza alcun container

```
@Composable  
fun HorizontalLayout() {  
    Text("First item")  
    Text("Second item")  
    Text("Third item")  
}
```

- Cosa succede?
 - Gli elementi si sovrappongono



Box layout

- Il composable **Surface**, all'interno del quale abbiamo inserito i **Text**, ha un layout di tipo Box
- Possiamo replicare lo stesso risultato inserendo i **Text** all'interno di una **Box**

```
@Composable
fun HorizontalLayout() {
    Box {
        Text("First item")
        Text("Second item")
        Text("Third item")
    }
}
```

Row layout

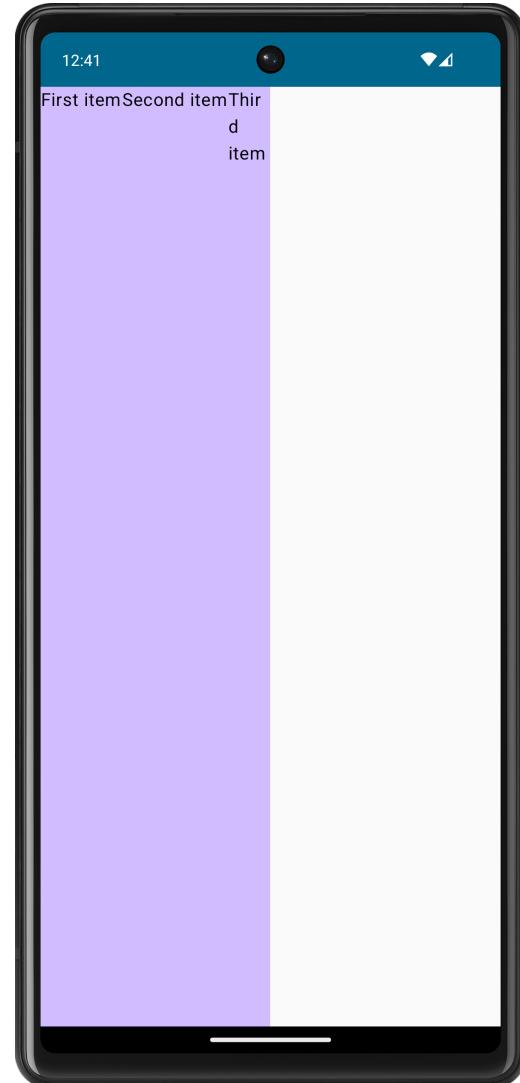
- Per disporre gli elementi uno a fianco all'altro dobbiamo utilizzare un layout di tipo **Row**

6. Soluzione

```
fun HorizontalLayout() {
    Row {
        Text("First item")
        Text("Second item")
        Text("Third item")
    }
}
```

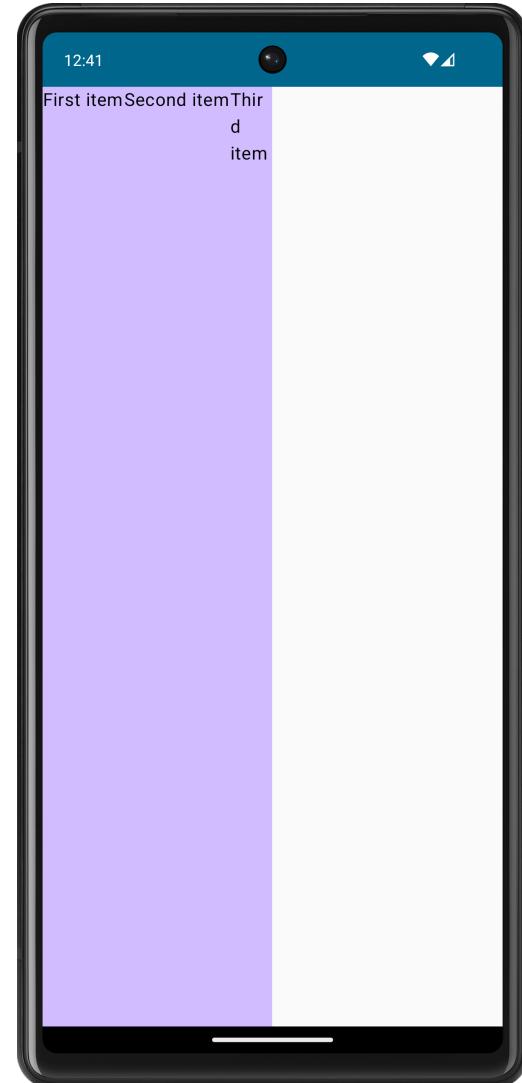
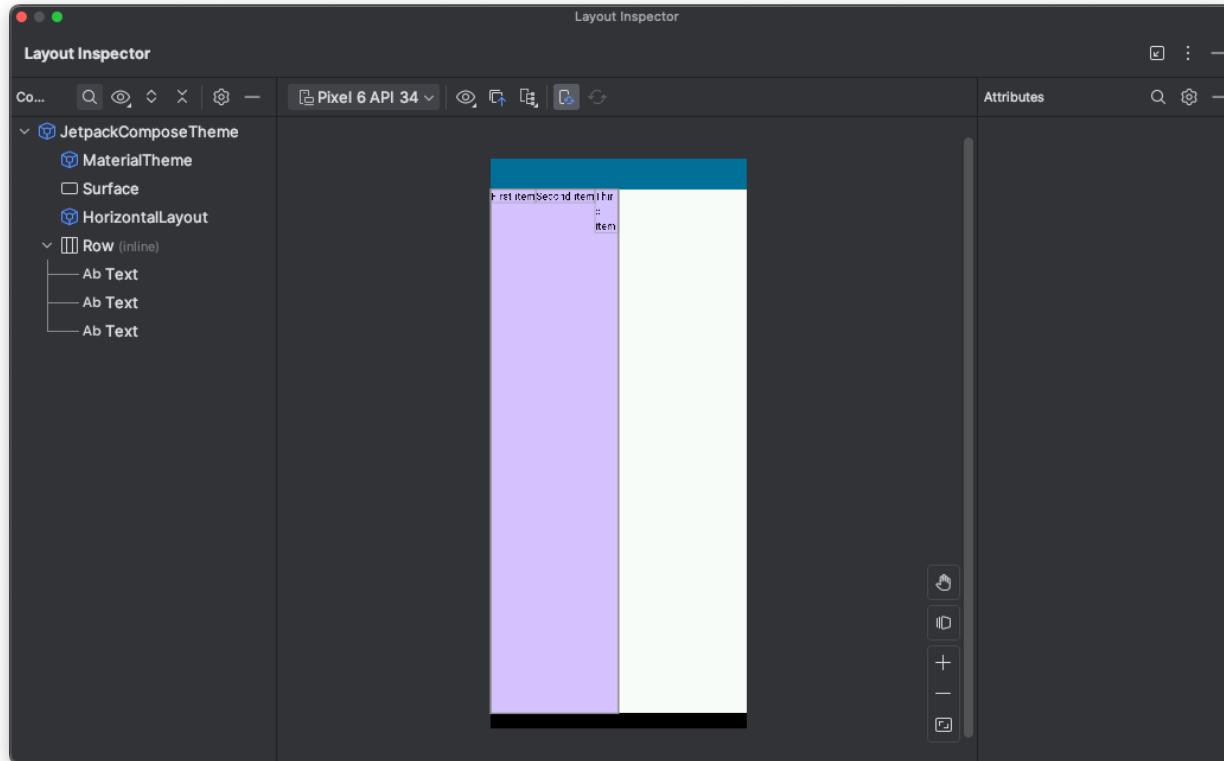
Layout inspector

- Perché gli elementi vengono visualizzati in questo modo?



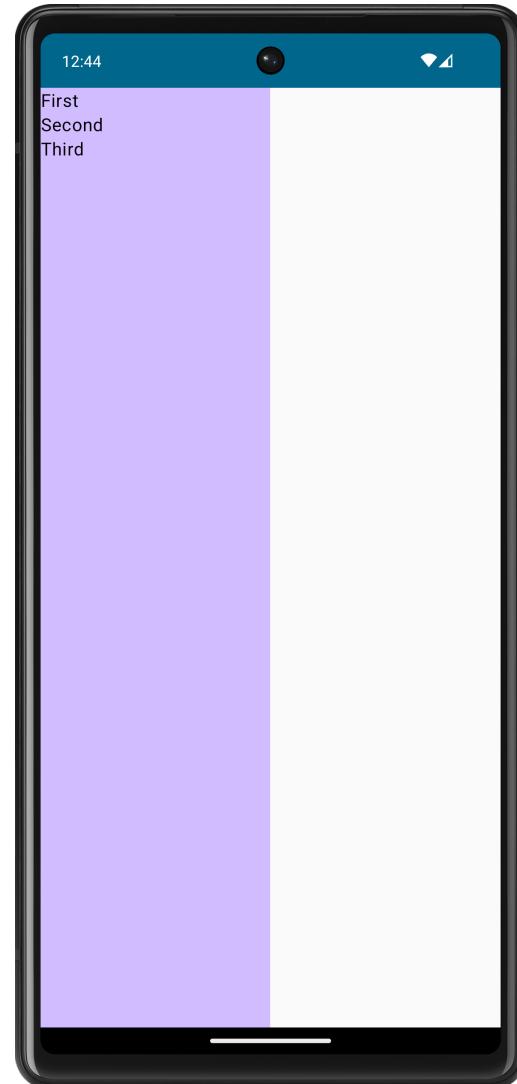
Layout inspector

- Perché gli elementi vengono visualizzati in questo modo?
- Possiamo scoprirlo esaminandoli con il **layout inspector**



7. Creare una colonna di elementi testuali

- Creare un composable che disponga tre elementi testuali uno sotto all'altro
- **Hint**
 - Basta partire dall'esercizio precedente e cambiare layout

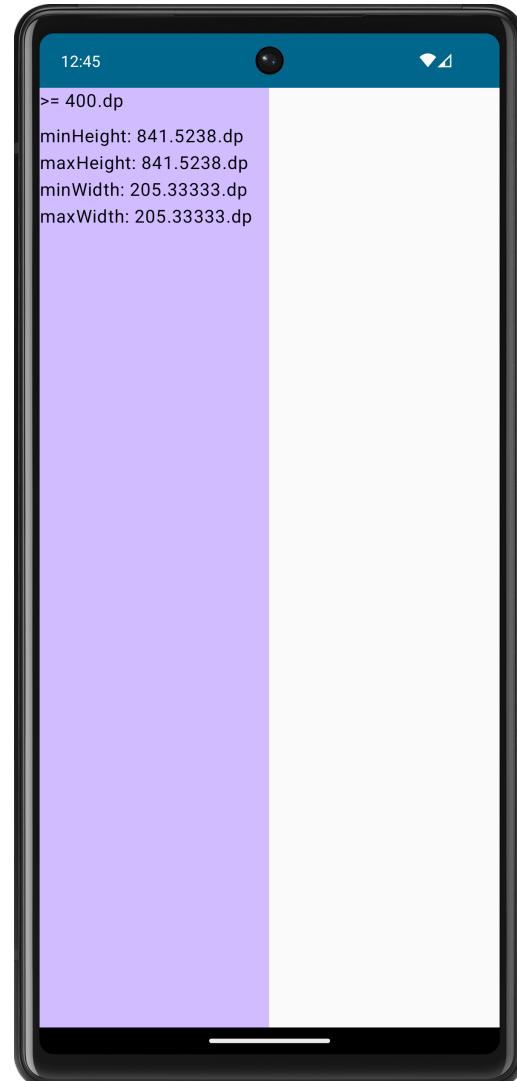
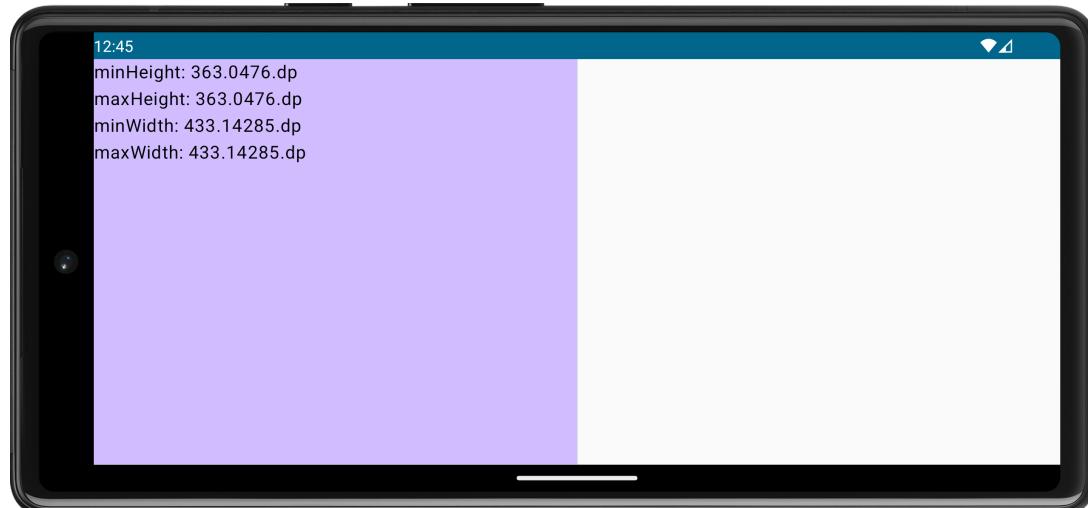


7. Soluzione

```
fun VerticalLayout() {
    Column {
        Text("First")
        Text("Second")
        Text("Third")
    }
}
```

8. Rilevare le dimensioni del container

- Creare un composable che:
 - Mostri le dimensioni minime e massime del suo container
 - Mostri il testo “ ≥ 400 dp” solo se la dimensione massima del container è ≥ 400 dp



BoxWithConstraints

- Per creare layout dinamici che cambiano in base alle dimensioni del container, Jetpack Compose offre il layout **BoxWithConstraints**

8. Soluzione

```
@Composable
fun BoxWithConstraintsLayout() {
    BoxWithConstraints {
        // Salva lo scope del BoxWithConstraints per utilizzarlo negli altri composable
        val box = this

        Column {
            if (box.maxHeight >= 400.dp) {
                Text(">= 400.dp")
                Spacer(modifier = Modifier.size(10.dp))
            }
            Text("""
                minHeight: ${box.minHeight}
                maxHeight: ${box.maxHeight}
                minWidth: ${box.minWidth}
                maxWidth: ${box.maxWidth}
            """.trimIndent())
        }
    }
}
```

8. Soluzione

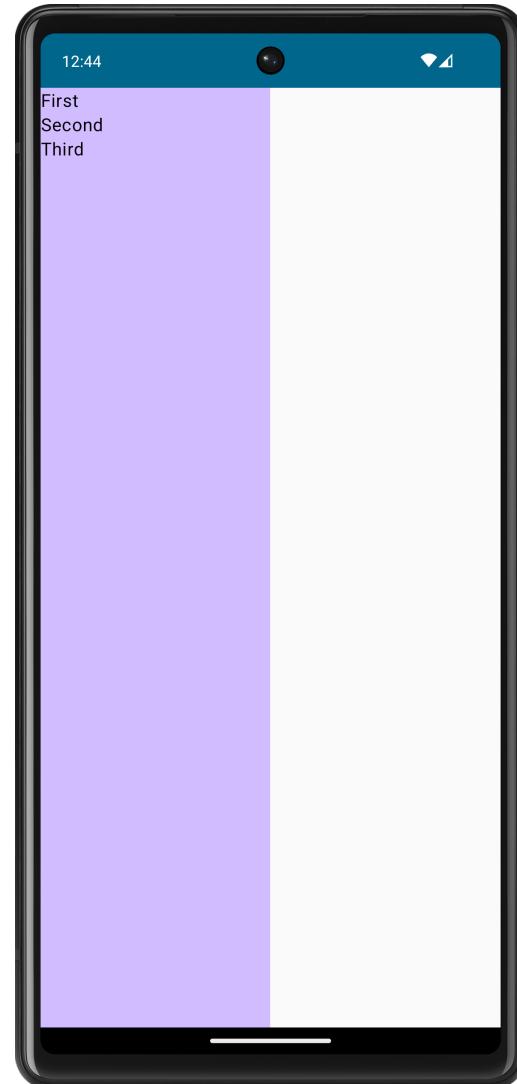
```
@Composable
fun BoxWithConstraintsLayout() {
    BoxWithConstraints {
        // Salva lo scope del BoxWithConstraints per utilizzarlo negli altri composable
        val box = this

        Column {
            if (box.maxHeight >= 400.dp) {
                Text(">= 400.dp")
                Spacer(modifier = Modifier.size(10.dp))
            }
            Text("""
                minHeight: ${box.minHeight}
                maxHeight: ${box.maxHeight}
                minWidth: ${box.minWidth}
                maxWidth: ${box.maxWidth}
            """.trimIndent())
        }
    }
}
```

BoxWithConstraints è un layout di tipo Box, quindi serve inserire una Column al suo interno per evitare sovrapposizioni

9. Gestire contenuti dinamici

- Creare un composable che mostri un numero variabile di elementi testuali, ricevuti come parametro, uno sotto all'altro



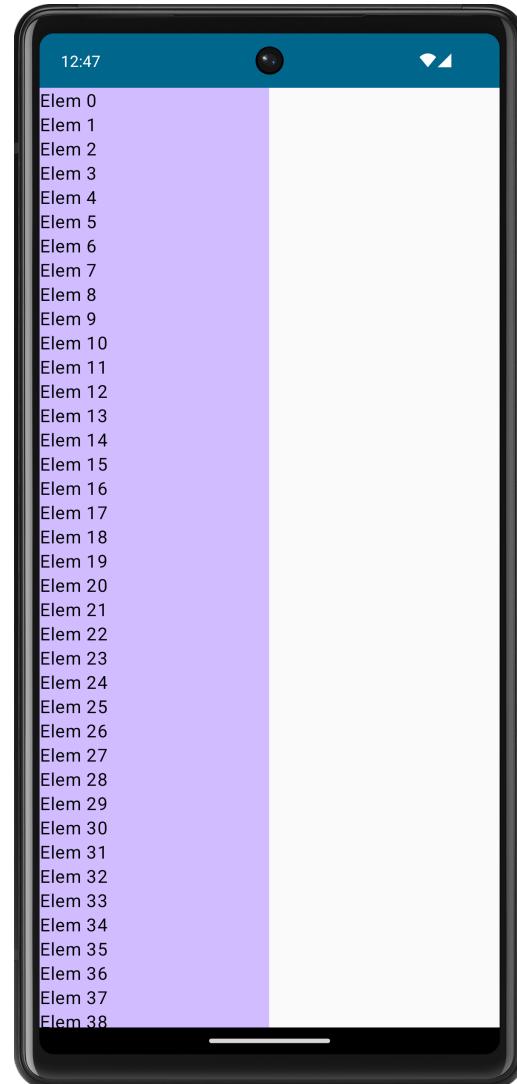
9. Soluzione

```
fun DynamicContent(items: List<String>) {  
    Column {  
        for (item in items) {  
            Text(item)  
        }  
    }  
}
```

```
DynamicContent(listOf("First", "Second", "Third"))
```

10. Creare una lista con scroll

- Creare un composable con una lista scrollabile di 100 elementi testuali



10. Primo tentativo

```
fun ScrollableList() {  
    val elems = (0..100).map { "Elem $it" }  
    val scrollState = rememberScrollState()  
  
    Column(modifier = Modifier.verticalScroll(scrollState)) {  
        for (elem in elems) Text(elem)  
    }  
}
```

10. Primo tentativo

rememberScrollState consente di mantenere lo stato della scrollbar in seguito ad eventuali recomposition

```
fun ScrollableList() {  
    val elems = (0..100).map { "Elem $it" }  
    val scrollState = rememberScrollState()  
  
    Column(modifier = Modifier.verticalScroll(scrollState)) {  
        for (elem in elems) Text(elem)  
    }  
}
```

10. Problemi di prestazioni della soluzione proposta

- Cosa succede aumentando il numero di elementi, ad esempio, da 100 a 10.000?

10. Problemi di prestazioni della soluzione proposta

- Cosa succede aumentando il numero di elementi, ad esempio, da 100 a 10.000?
- Le prestazioni dell'app lasciano... un po' a desiderare 😅

10. Problemi di prestazioni della soluzione proposta

- Cosa succede aumentando il numero di elementi, ad esempio, da 100 a 10.000?
- Le prestazioni dell'app lasciano... un po' a desiderare 😅
- Perché?
 - Anche se solo una piccola parte degli elementi è visibile nel display, Jetpack compose fa il rendering di tutti e 10.000

10. Problemi di prestazioni della soluzione proposta

- Cosa succede aumentando il numero di elementi, ad esempio, da 100 a 10.000?
- Le prestazioni dell'app lasciano... un po' a desiderare 😅
- Perché?
 - Anche se solo una piccola parte degli elementi è visibile nel display, Jetpack compose fa il rendering di tutti e 10.000
- Come possiamo evitarlo?
 - Con una **LazyColumn**

10. Soluzione

```
@Composable
fun ScrollableList() {
    val elems = (0..100).map { "Elem $it" }

    LazyColumn {
        items(elems) {
            Text(it)
        }
    }
}
```

10. Soluzione

```
@Composable
fun ScrollableList() {
    val elems = (0..100).map { "Elem $it" }

    LazyColumn {
        items(elems) {
            Text(it)
        }
    }
}
```

Combinando `LazyColumn` e l'helper `items`,
l'app ora fa il render solo degli elementi
attualmente visibili

Riferimenti

- Compose layout basics

<https://developer.android.com/jetpack/compose/layouts/basics>