
App Architecture e Jetpack Compose

Introduzione e sintassi

Lezione 6

Premessa

- Tenete presente che i dispositivi mobile sono anche ***resource-constrained***, quindi in qualsiasi momento il sistema operativo potrebbe eliminare alcuni processi dell'app per fare spazio a quelli nuovi
- È possibile che i **componenti** dell'app vengano lanciati *singolarmente* e in *ordine sparso* e che il sistema operativo o l'utente possano distruggerli in qualsiasi momento
- Poiché questi eventi non sono controllabili, non si dovrebbe memorizzare o tenere in memoria alcun dato o stato dell'applicazione nei componenti dell'app, i quali non dovrebbero dipendere l'uno dall'altro

Principi architetturali

- L'**architettura** di un'app definisce i confini tra le varie parti e le **responsabilità** di ciascuna
- È necessario progettare l'architettura dell'app seguendo alcuni **principi** specifici:
 - Separation of concerns
 - Drive UI from data models
 - Single source of truth
 - Unidirectional Data Flow

Separation of concerns

- È un errore comune scrivere tutto il codice in un'Activity. Queste classi basate sull'UI dovrebbero contenere solo la **logica che gestisce le interazioni con l'interfaccia e il sistema operativo**
- Mantenendo queste classi il più possibile **snelle**, si possono evitare molti problemi legati al ciclo di vita dei componenti e migliorare la testabilità di queste classi
- Il sistema operativo può distruggerle in qualsiasi momento in base alle interazioni dell'utente o a causa di condizioni di sistema come la mancanza di memoria
- Per offrire un'esperienza utente soddisfacente e una manutenzione dell'applicazione più gestibile, è **meglio ridurre al minimo la dipendenza da queste classi**

Drive UI from data models

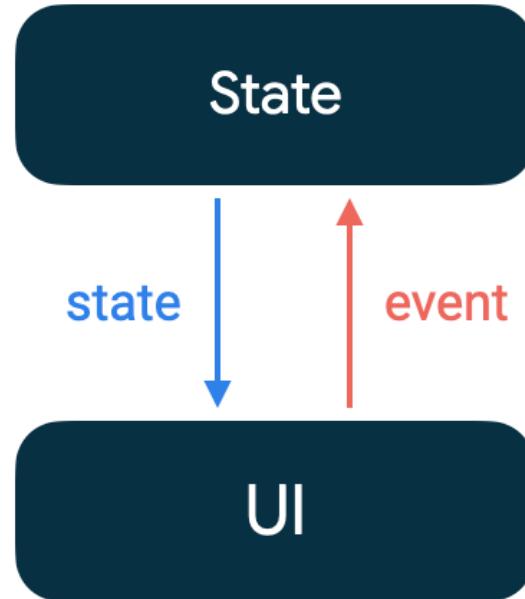
- Un altro principio importante è che **l'UI deve essere guidata da modelli di dati**, preferibilmente **persistenti**
- I modelli di dati sono **indipendenti dagli elementi dell'interfaccia utente** e dagli altri componenti dell'applicazione. Ciò significa che non sono legati al ciclo di vita dell'interfaccia utente e dei componenti dell'applicazione, ma saranno comunque distrutti quando il sistema operativo deciderà di rimuovere il processo dell'applicazione dalla memoria.
- I **modelli persistenti** sono ideali per i seguenti motivi:
 - Gli utenti non perdono dati se il sistema operativo Android distrugge l'app per liberare risorse.
 - L'app continua a funzionare anche quando la connessione di rete è debole o non disponibile.
- Se si basa l'architettura dell'applicazione su classi di modelli di dati, si rende l'applicazione più testabile e robusta.

Single source of truth

- Quando si definisce un nuovo tipo di dati nella propria applicazione, è necessario assegnargli una **Single Source of Truth** (SSOT). L'SSOT è il **proprietario** dei dati e solo lui può modificarli. A tale scopo, l'SSOT espone i dati utilizzando un tipo immutabile e, per modificarli, espone funzioni o riceve eventi che altri tipi possono chiamare.
- Questo modello offre molteplici **vantaggi**:
 - Centralizza tutte le modifiche a un particolare tipo di dati in un unico luogo.
 - Protegge i dati in modo che altri tipi non possano manometterli.
 - Rende le modifiche ai dati più tracciabili. In questo modo, i bug sono più facili da individuare.
- In un'applicazione **offline-first**, la SSOT è tipicamente un **database**. In altri casi, può essere un **ViewModel** o addirittura l'UI.

Unidirectional Data Flow

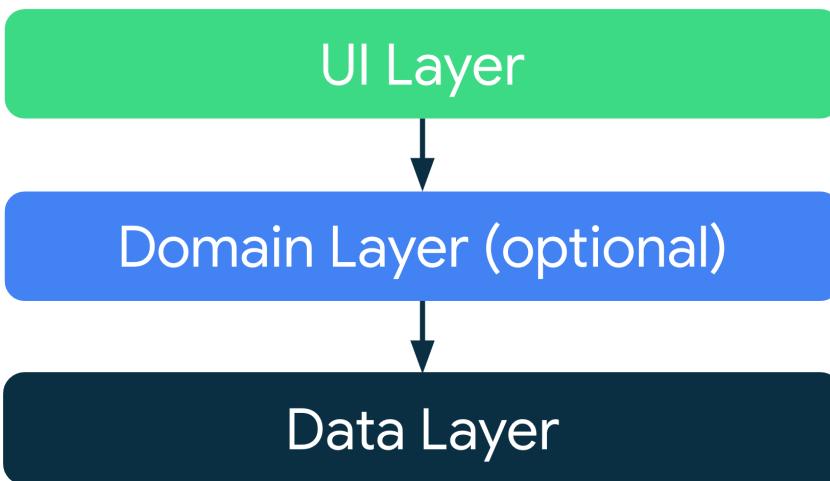
- Il principio della Single source of truth viene spesso usato con il modello UDF (Unidirectional Data Flow). In UDF, lo **stato scorre in una sola direzione**. Gli **eventi** che modificano i dati scorrono nella **direzione opposta**.
 - Ad esempio, i dati dell'applicazione di solito fluiscono dalle fonti di dati all'interfaccia utente. Gli eventi dell'utente, come la pressione di un pulsante, passano dall'interfaccia utente all'SSOT, dove i dati dell'applicazione vengono modificati ed esposti in un tipo immutabile.
- Questo modello garantisce meglio la **coerenza** dei dati, è meno soggetto a errori, è più facile da debuggare e offre tutti i vantaggi del modello SSOT.



Architettura

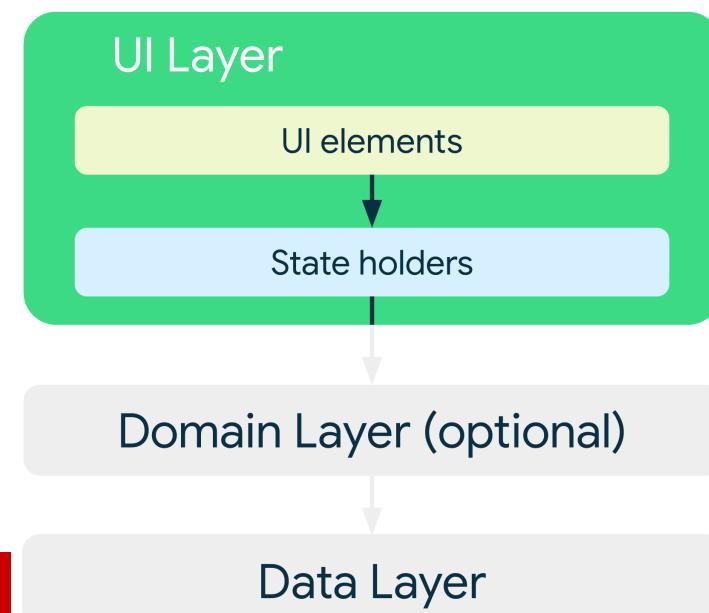
- Ogni applicazione dovrebbe avere almeno due layer:
 - Il **layer UI** che visualizza i dati dell'applicazione sullo schermo.
 - Il **layer dati**, che contiene la logica dell'applicazione ed espone i dati.
- È possibile aggiungere un ulteriore layer, chiamato **domain layer**, per semplificare e riutilizzare le interazioni tra i livelli UI e dati.

Le → rappresentano le dipendenze tra le classi. Ad esempio, il domain layer dipende dalle classi del layer dati.



UI layer

- Il ruolo del **layer UI** (o layer di presentazione) è quello di visualizzare i dati.
- Ogni volta che i dati cambiano, a causa dell'interazione dell'utente (come la pressione di un pulsante) o di un input esterno (come una risposta di rete), l'interfaccia utente deve aggiornarsi per riflettere le modifiche.
- Il livello UI è composto da **due elementi**:
 - **Elementi dell'interfaccia utente** che visualizzano i dati sullo schermo e si costruiscono utilizzando le funzioni di *Views* (prima) oppure **Jetpack Compose** (ora).
 - State holders (come le classi *ViewModel*) che contengono i dati, li espongono all'interfaccia utente e gestiscono la logica.



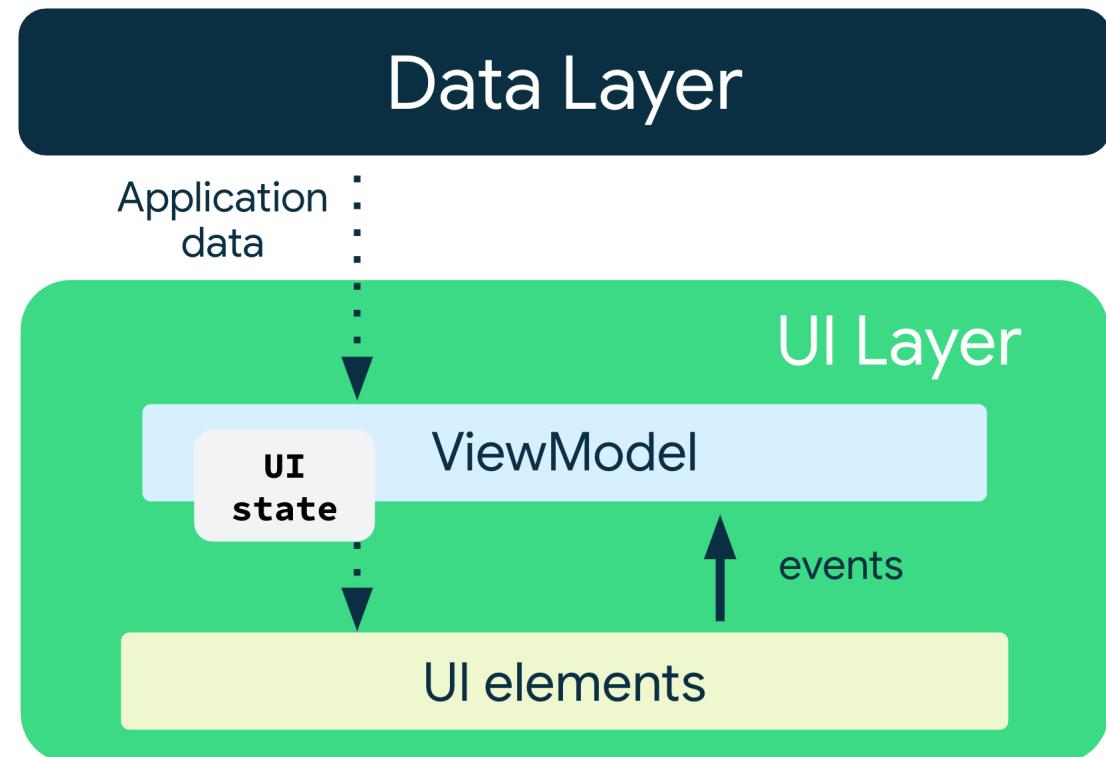
Approfondimento: UI state

- Se l'interfaccia utente (UI) è ciò che vede l'utente, lo stato dell'interfaccia utente è ciò che l'app dice che dovrebbe vedere
- Come due facce della stessa medaglia, l'interfaccia utente è la rappresentazione visiva dello stato dell'interfaccia utente. Qualsiasi modifica allo stato dell'interfaccia utente si riflette immediatamente nell'interfaccia utente.



Approfondimento: State holders

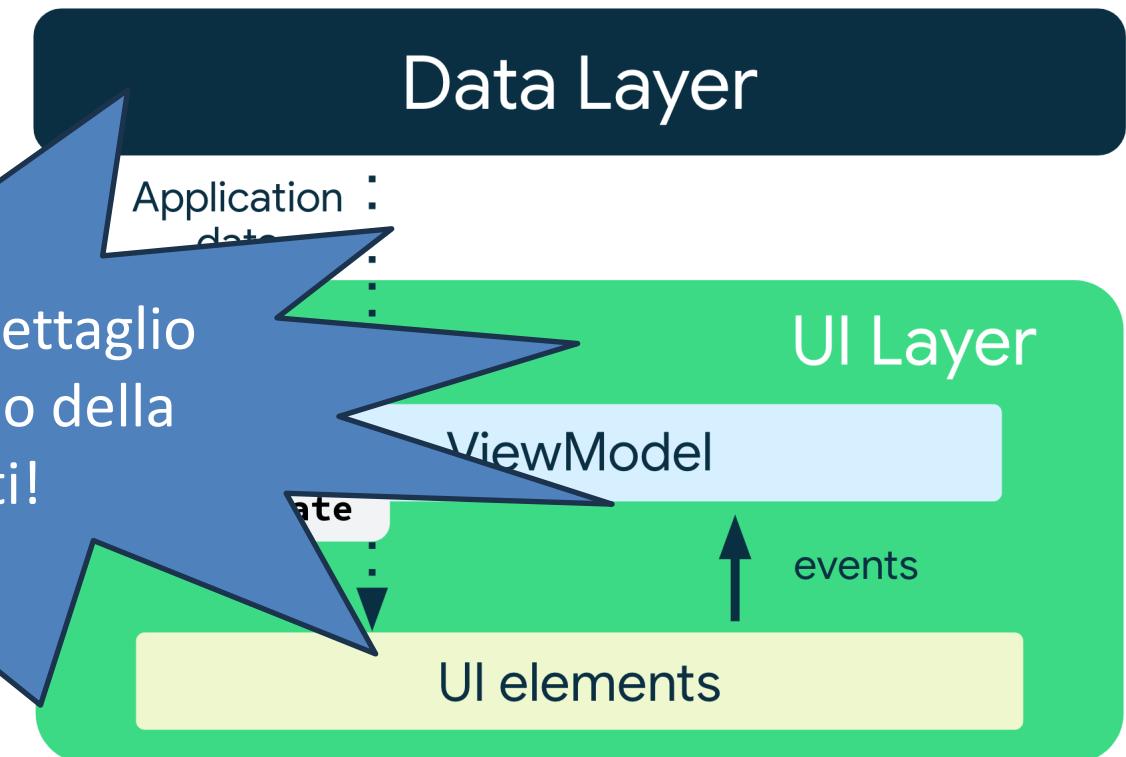
- Le classi responsabili della produzione dello stato dell'interfaccia utente e che contengono la logica necessaria per tale attività sono chiamate **State holders**
- Gli state holders sono disponibili in diverse *dimensioni* a seconda dell'ambito degli elementi dell'interfaccia utente corrispondenti che gestiscono, che vanno da un singolo widget come la barra dell'app inferiore a un intero schermo o una destinazione di navigazione ([ViewModel](#))
- Il ViewMode è l'implementazione consigliata per la gestione dello stato dell'interfaccia utente a livello di schermo con accesso al livello dati. Inoltre, sopravvive automaticamente alle modifiche della configurazione. Le classi ViewModel definiscono la logica da applicare agli eventi nell'app e producono di conseguenza uno stato aggiornato



Approfondimento: State holders

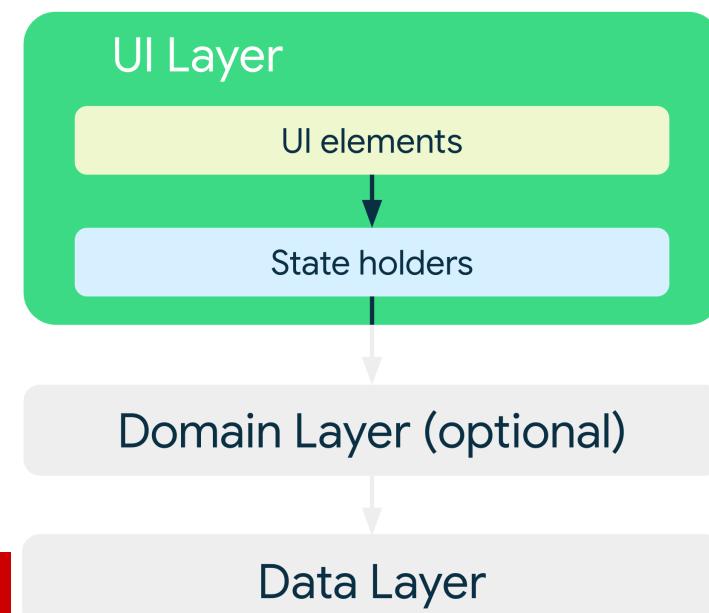
- Le classi responsabili della produzione dello stato dell'interfaccia utente e che contengono la logica necessaria per tale attività sono chiamate **State holders**
- Gli state holders sono disponibili in diverse *dimensioni* a seconda dell'ambito degli elementi dell'interfaccia utente corrispondenti che si trovano su un singolo widget come la barra intero schermo o (ViewModel)
- Il ViewModel è l'implementazione della gestione dello stato dell'interfaccia utente con accesso al livello di dati. Inoltre il ViewModel vive automaticamente alle modifiche della configurazione. Le classi ViewModel definiscono la logica da applicare agli eventi nell'app e producono di conseguenza uno stato aggiornato

Ne parleremo in dettaglio
quando parleremo della
gestione dati!



UI layer (..dove eramo rimasti..)

- Il ruolo del **layer UI** (o layer di presentazione) è quello di visualizzare i dati.
- Ogni volta che i dati cambiano, a causa dell'interazione dell'utente (come la pressione di un pulsante) o di un input esterno (come una risposta di rete), l'interfaccia utente deve aggiornarsi per riflettere le modifiche.
- Il livello UI è composto da **due elementi**:
 - Elementi dell'interfaccia utente che visualizzano i dati sullo schermo e si costruiscono utilizzando le funzioni di *Views* (prima) oppure **Jetpack Compose** (ora).
 - State holders (come le classi **ViewModel**) che contengono i dati, li espongono all'interfaccia utente e gestiscono la logica.



Jetpack compose

**Build better apps faster with
Jetpack Compose**

Jetpack compose

- Jetpack Compose è il **toolkit** moderno consigliato da Android per la creazione di **interfacce utente native**.
- Semplifica e accelera lo sviluppo dell'interfaccia utente su Android.
- Dà rapidamente vita alla vostra applicazione con **meno codice**, strumenti potenti e API Kotlin intuitive.

Less code

Do more with less code and avoid entire classes of bugs, so code is simple and easy to maintain.

Intuitive

Just describe your UI, and Compose takes care of the rest. As app state changes, your UI automatically updates.

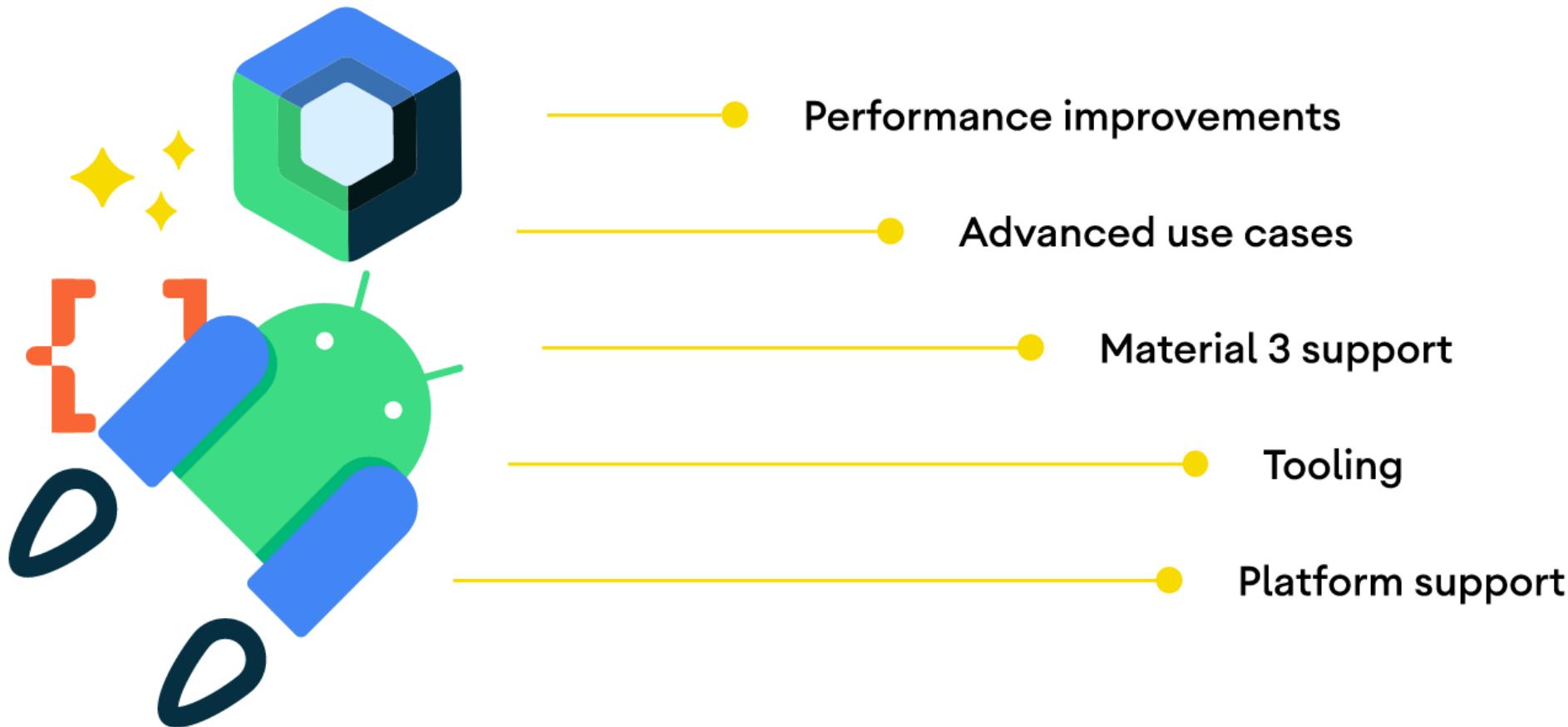
Accelerate development

Compatible with all your existing code so you can adopt when and where you want. Iterate fast with live previews and full Android Studio support.

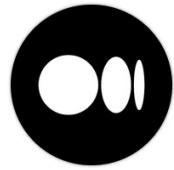
Powerful

Create beautiful apps with direct access to the Android platform APIs and built-in support for Material Design, Dark theme, animations, and more.

What's next for Jetpack Compose

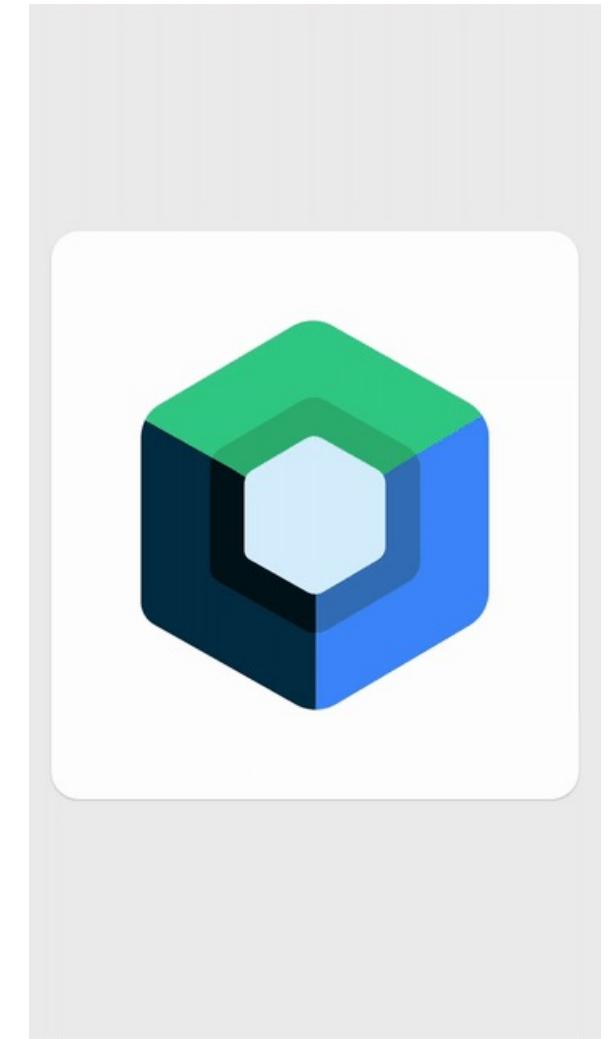


App che usano Compose



Jetpack compose

```
@Composable
fun JetpackCompose() {
    Card {
        var expanded by remember { mutableStateOf(false) }
        Column(Modifier.clickable { expanded = !expanded }) {
            Image(painterResource(R.drawable.jetpack_compose))
            AnimatedVisibility(expanded) {
                Text(
                    text = "Jetpack Compose",
                    style = MaterialTheme.typography.bodyLarge,
                )
            }
        }
    }
}
```



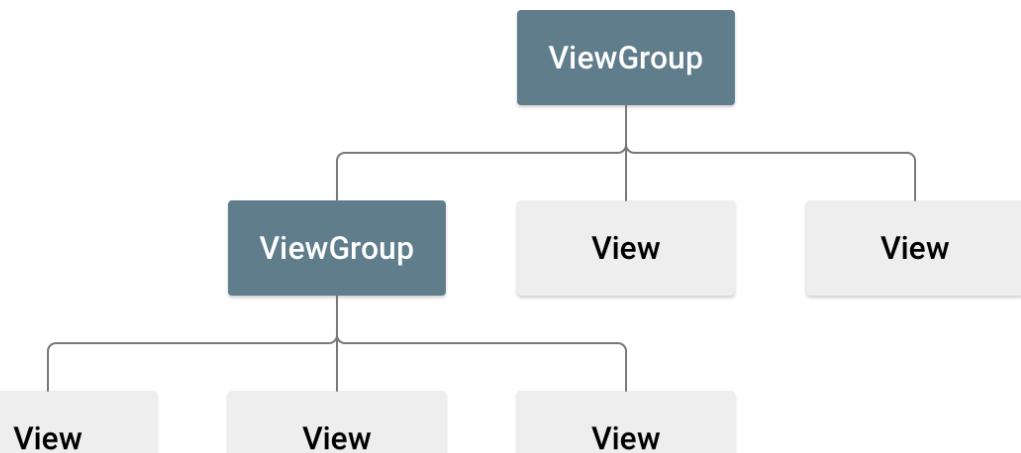
Come usare compose?

- build.gradle (module:app)
 - Dipendenze
 - <https://developer.android.com/jetpack/compose/setup>

```
dependencies {  
  
    val composeBom = platform("androidx.compose:compose-bom:2023.01.00")  
    implementation(composeBom)  
    androidTestImplementation(composeBom)  
  
    // Choose one of the following:  
    // Material Design 3  
    implementation("androidx.compose.material3:material3")  
    // or Material Design 2  
    implementation("androidx.compose.material:material")  
    // or skip Material Design and build directly on top of foundational components  
    implementation("androidx.compose.foundation:foundation")  
    // or only import the main APIs for the underlying toolkit systems,  
    // such as input and measurement/layout  
    implementation("androidx.compose.ui:ui")  
  
    // Android Studio Preview support  
    implementation("androidx.compose.ui:ui-tooling-preview")  
    debugImplementation("androidx.compose.ui:ui-tooling")  
  
    // UI Tests  
    androidTestImplementation("androidx.compose.ui:ui-test-junit4")  
    debugImplementation("androidx.compose.ui:ui-test-manifest")  
  
    // Optional - Included automatically by material, only add when you need  
    // the icons but not the material library (e.g. when using Material3 or a  
    // custom design system based on Foundation)  
    implementation("androidx.compose.material:material-icons-core")  
    // Optional - Add full set of material icons  
    implementation("androidx.compose.material:material-icons-extended")  
    // Optional - Add window size utils  
    implementation("androidx.compose.material3:material3-window-size-class")  
  
    // Optional - Integration with activities  
    implementation("androidx.activity:activity-compose:1.6.1")  
    // Optional - Integration with ViewModels  
    implementation("androidx.lifecycle:lifecycle-viewmodel-compose:2.5.1")  
    // Optional - Integration with LiveData  
    implementation("androidx.compose.runtime:runtime-livedata")  
    // Optional - Integration with RxJava  
    implementation("androidx.compose.runtime:runtime-rxjava2")  
}
```

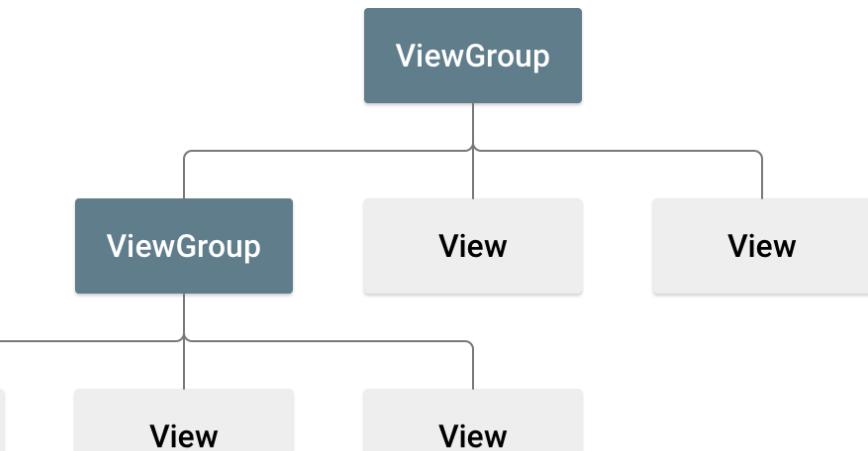
Prima...

- Storicamente, ogni interfaccia di app è stata rappresentata come una gerarchia di widget (View) dell'interfaccia utente.
- Quando lo stato dell'applicazione cambia a causa di interazioni dell'utente, la gerarchia deve essere aggiornata per visualizzare i dati correnti.
- Il modo più comune di aggiornare manualmente l'interfaccia utente è quello di percorrere la gerarchia e di modificare i nodi chiamando i metodi



Prima...

- **Problema:** la manipolazione manuale delle View aumenta la probabilità di errori.
- Se un *dato viene preso in più punti*, è facile dimenticare di aggiornare una delle View che lo mostrano.
- È anche facile avere due *aggiornamenti in conflitto* in modo imprevisto.
 - Ad esempio, un aggiornamento potrebbe tentare di impostare il valore di un nodo che è stato appena rimosso dall'interfaccia utente.
- In generale, la complessità della manutenzione del software cresce con il numero di View da aggiornare.



Ora...

- Negli ultimi anni, si è iniziato ad utilizzare un modello **dichiarativo** dell'interfaccia utente, che semplifica la progettazione associata alla costruzione e all'aggiornamento delle interfacce utente.
- La tecnica funziona **rigenerando** l'intera schermata da zero e applicando solo le modifiche necessarie.
- Questo approccio evita la complessità di aggiornare manualmente una gerarchia di View statiche.
- **Compose è un framework dichiarativo per l'interfaccia utente.**

Ora...

- La **rigenerazione** dell'intera schermata è potenzialmente *costosa* in termini di tempo, potenza di calcolo e utilizzo della batteria.
- Per ridurre questo costo, Compose sceglie in *modo intelligente* quali parti dell'interfaccia utente devono essere ridisegnate in qualsiasi momento.

Paradigma imperativo (prima)

- Con molti UI-kit **imperativi** orientati agli oggetti, si inizializza l'interfaccia utente istanziando un albero di View (o Widget più in generale). Spesso lo si fa tramite inflate di un file di layout XML.
- Ogni **widget** mantiene il proprio stato interno ed espone metodi **getter** e **setter** che consentono alla logica dell'applicazione di interagire con il widget.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"
    tools:context=".MainActivity">

    <EditText
        android:id="@+id/cost_of_service"
        android:hint="Cost of Service"
        android:layout_height="wrap_content"
        android:layout_width="160dp"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        android:inputType="numberDecimal"/>

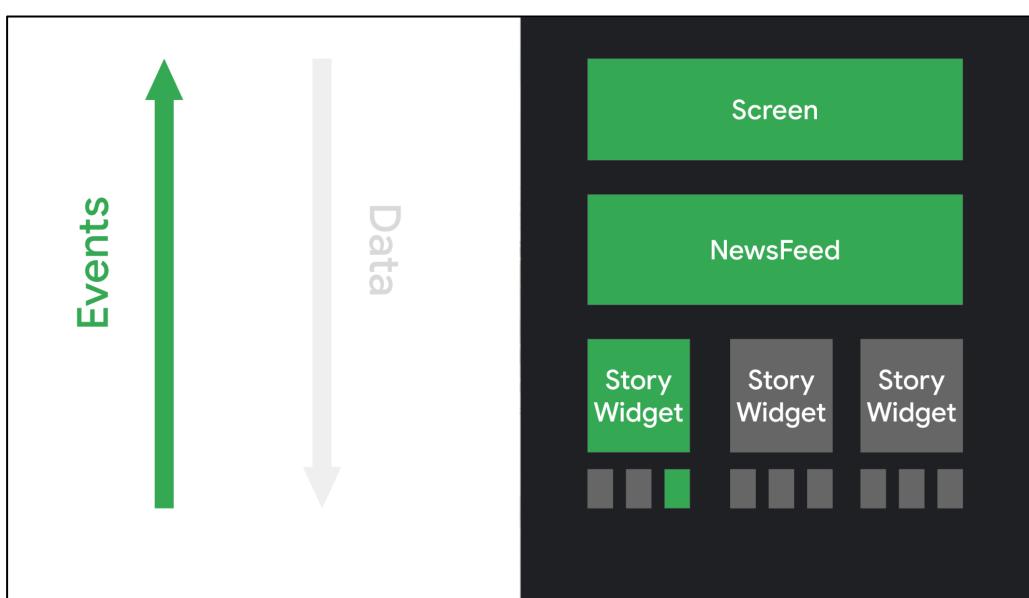
    <TextView
        android:id="@+id/service_question"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="How was the service?"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/cost_of_service"/>

</androidx.constraintlayout.widget.ConstraintLayout>
```

Paradigma dichiarativo (ora)

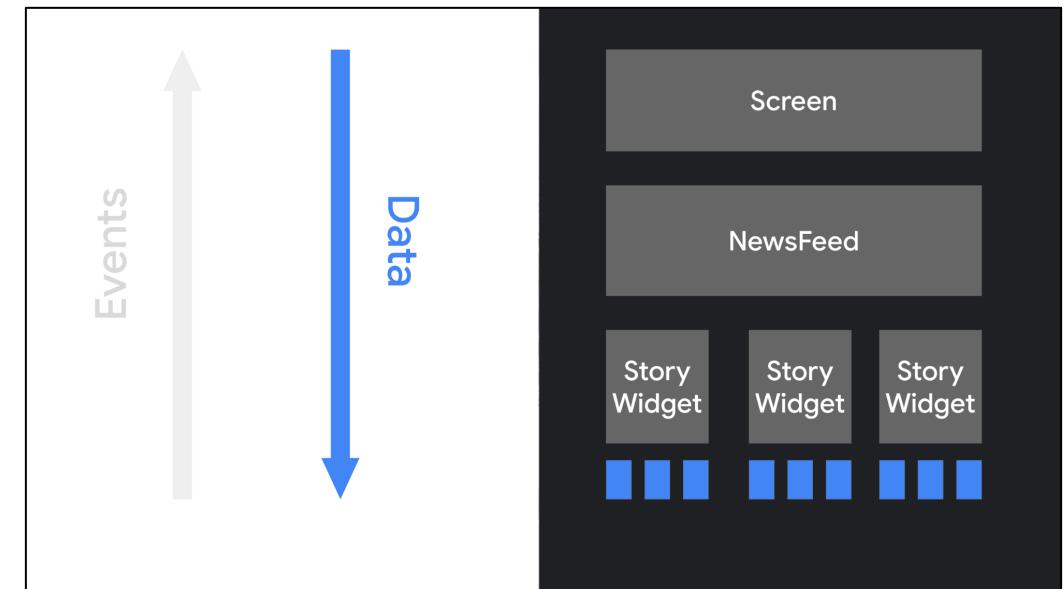
- Nell'approccio **dichiarativo** di Compose, i widget sono privi di stato e non espongono funzioni setter o getter => L'aggiornamento dell'interfaccia utente avviene richiamando la stessa funzione di Compose con argomenti diversi.
- I **composable** sono responsabili della trasformazione dello stato corrente dell'applicazione in un'interfaccia aggiornata ogni volta che i dati vengono aggiornati: processo di *recomposition* (-> a breve lo vedremo meglio)

Paradigma dichiarativo



L'utente ha interagito con un elemento dell'interfaccia (**evento**).

La logica dell'applicazione risponde all'evento, quindi le funzioni composable vengono automaticamente richiamate con nuovi parametri, se necessario.



La logica dell'applicazione fornisce i dati alla **funzione composable** di livello superiore. Tale funzione utilizza i dati per descrivere l'interfaccia chiamando altri composable e passando i dati appropriati a tali composable e così via lungo la gerarchia.

Funzioni composable

- Utilizzando Compose, è possibile costruire l'interfaccia definendo un insieme di **funzioni composable** che accettano dati ed emettono elementi dell'interfaccia.



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

La funzione deve essere annotata con l'annotazione **@Composable**, che informa il compilatore che questa funzione è destinata a convertire i dati in UI.

Funzioni composable

- Utilizzando Compose, è possibile costruire l'interfaccia definendo un insieme di **funzioni composable** che accettano dati ed emettono elementi dell'interfaccia.



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

La funzione accetta dati. Le funzioni composable possono accettare parametri, che consentono alla logica dell'applicazione di descrivere l'interfaccia.
In questo caso, il nostro widget accetta una stringa, in modo da poter salutare l'utente per nome.

Funzioni composable

- Utilizzando Compose, è possibile costruire l'interfaccia definendo un insieme di **funzioni composable** che accettano dati ed emettono elementi dell'interfaccia.

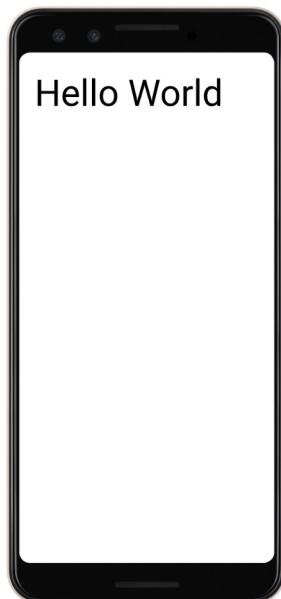


```
@Composable
fun Greeting(name: String) {
    Text("Hello $name")
```

La funzione visualizza il testo nell'interfaccia utente tramite *funzione composable Text()*, che crea effettivamente l'elemento testo. Le funzioni composable creano la gerarchia dell'interfaccia chiamando altre funzioni composable.

Funzioni composable

- Utilizzando Compose, è possibile costruire l'interfaccia definendo un insieme di **funzioni composable** che accettano dati ed emettono elementi dell'interfaccia.



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

A dark gray rectangular area contains the Kotlin code for a composable function named "Greeting". The code uses the `@Composable` annotation, defines a parameter `name` of type `String`, and returns a `Text` widget with the string "Hello \$name". A red arrow points from the explanatory text on the right towards the closing brace of the function definition.

La funzione non restituisce nulla.

Le funzioni composable non hanno bisogno di restituire nulla, perché **descrivono** lo stato dell'interfaccia invece di costruire i widget dell'interfaccia utente.

La funzione descrive l'interfaccia utente senza effetti collaterali, come la modifica di proprietà o variabili globali.

Contenuto dinamico

- Poiché le funzioni composable sono in Kotlin anziché in XML, possono essere **dinamiche** come qualsiasi altro codice Kotlin.

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

Dentro le composable si possono usare costrutti tipo:

- For
- If
- Loops
- ...

Ricomposizione

- In un **modello imperativo** (prima), per modificare un widget, si chiama un setter per cambiarne il suo stato interno.
- In **Compose**, si richiama la funzione composable con i nuovi dati. In questo modo, la funzione viene «ricomposta»: i widget emessi dalla funzione vengono ridisegnati, se necessario, con i nuovi dati.
- Il framework Compose può ricomporre in modo intelligente solo i componenti che sono cambiati.
 - In un composable è consigliato non eseguire:
 - Scrittura di una proprietà di un oggetto condiviso
 - Aggiornamento di un *observable* nel ViewModel [vedremo in seguito il significato]
 - Aggiornamento delle *sharedPreferences*

Composable functions

- Funzione composable per il design di 3 schermate in un tab layout
- **In che ordine vengono create??**

```
@Composable
fun ButtonRow() {
    MyFancyNavigation {
        StartScreen()
        MiddleScreen()
        EndScreen()
    }
}
```

Composable functions

```
@Composable
fun ButtonRow() {
    MyFancyNavigation {
        StartScreen()
        MiddleScreen()
        EndScreen()
    }
}
```

- Funzione composable per il design di 3 schermate in un tab layout
- **In che ordine vengono create?**
 - Le funzioni composable possono essere eseguite in qualsiasi ordine
 - Ciò significa che non è possibile, ad esempio, che StartScreen() definisca una variabile globale e che MiddleScreen() sfrutti tale cambiamento. Invece, ciascuna di queste funzioni deve essere autonoma.

Composable functions

```
@Composable
fun ListComposable(myList: List<String>) {
    Row(horizontalArrangement = Arrangement.SpaceBetween) {
        Column {
            for (item in myList) {
                Text("Item: $item")
            }
        }
        Text("Count: ${myList.size}")
    }
}
```

- Compose può ottimizzare la ricomposizione eseguendo le funzioni composable **in parallelo**.
- Ciò consente a Compose di sfruttare più core e di eseguire le funzioni composable non presenti sullo schermo con una priorità inferiore.
- Quando viene invocata una funzione composable, l'invocazione potrebbe avvenire su un thread diverso da quello del chiamante.
- Ciò significa che il codice che modifica le variabili in un composable deve essere evitato, sia perché tale codice non è thread-safe, sia perché è un side-effect.

State e Compose

- Lo stato in un'applicazione è qualsiasi valore che può cambiare nel tempo.
- Tutte le applicazioni Android mostrano lo «stato» all'utente. Alcuni esempi:
 - Una Snackbar che mostra quando non è possibile stabilire una connessione di rete.
 - Un post di un blog e i relativi commenti.
 - Animazioni sui pulsanti che vengono riprodotte quando l'utente fa clic su di essi.
 - Adesivi che l'utente può disegnare sopra un'immagine.
- Jetpack Compose aiuta a definire in modo esplicito dove e come memorizzare e utilizzare lo stato

State e Compose

- Compose è **dichiarativo** e come tale l'unico modo per aggiornarlo è richiamare lo stesso composable con nuovi **argomenti**.
- Questi **argomenti** rappresentano lo **stato** dell'interfaccia utente. Ogni volta che uno stato viene aggiornato, avviene una ricomposizione. Di conseguenza, è necessario comunicare esplicitamente a un composable il nuovo stato, affinché si aggiorni di conseguenza.

Si aggiorna quando il valore cambia

```
@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") }
        )
    }
}
```

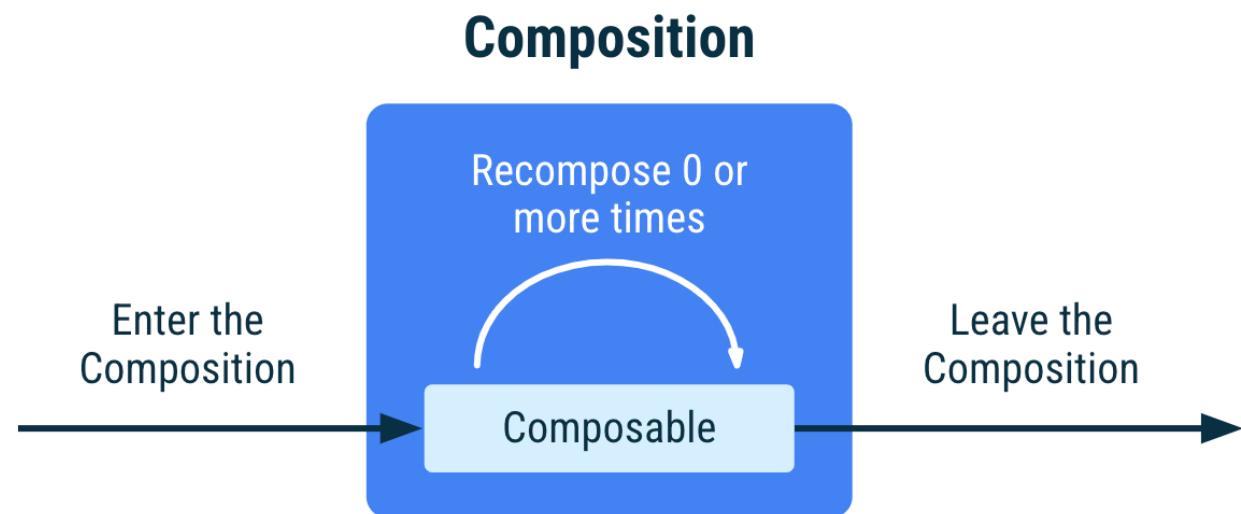
State e Compose

- Componiamo la funzione che ci permette di calcolare lo stesso valore.
 - Questo è un esempio di funzione che non ha senso per tutti i valori del dominio. Per esempio, se non abbiamo una radice quadrata di un numero negativo, non possiamo calcolarne la funzione.

**Vedremo State più in dettaglio
nelle prossime lezioni**

Ciclo di vita dei composable

- Una **composizione** può essere prodotta solo da una composizione iniziale e aggiornata tramite ricomposizione.
- L'unico modo per modificare una composizione è la **ricomposizione**.



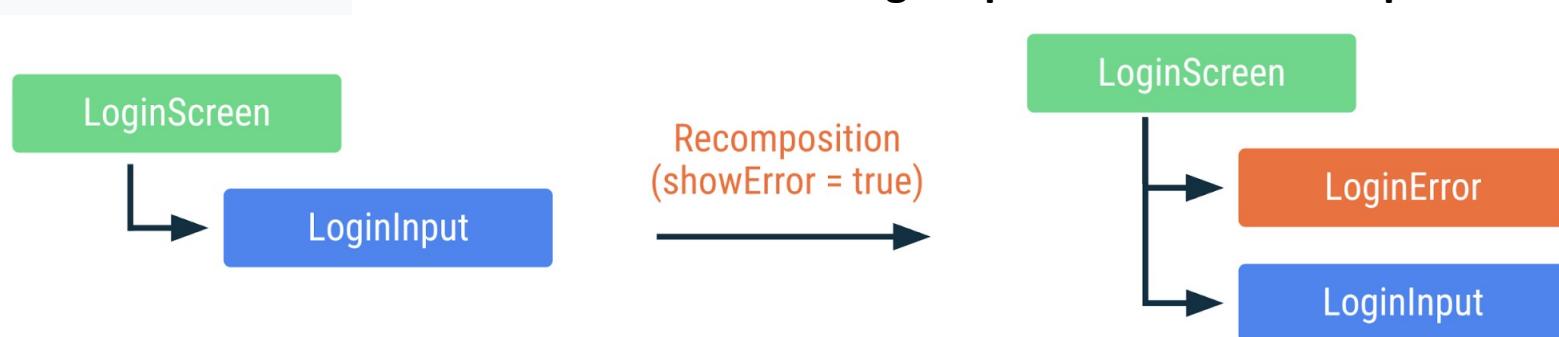
Ciclo di vita dei composable

- Se durante una ricomposizione un composable chiama composable diversi da quelli che ha chiamato durante la composizione precedente, Compose identifierà quali composable sono stati chiamati o meno e per i composable che sono stati chiamati in entrambe le composizioni, Compose eviterà di ricomporli se i loro input non sono cambiati → è importante che ogni composable sia **identificato univocamente**

```
@Composable
fun LoginScreen(showError: Boolean) {
    if (showError) {
        LoginError()
    }
    LoginInput() // This call site affects where LoginInput is placed in Composition
}

@Composable
fun LoginInput() { /* ... */ }
```

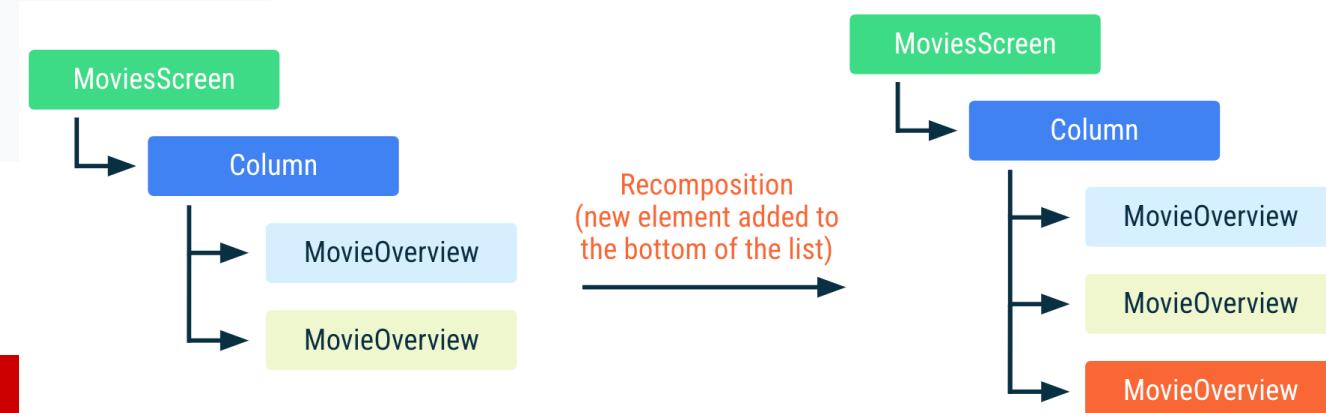
LoginInput non viene ricomposto!



Ciclo di vita dei composable

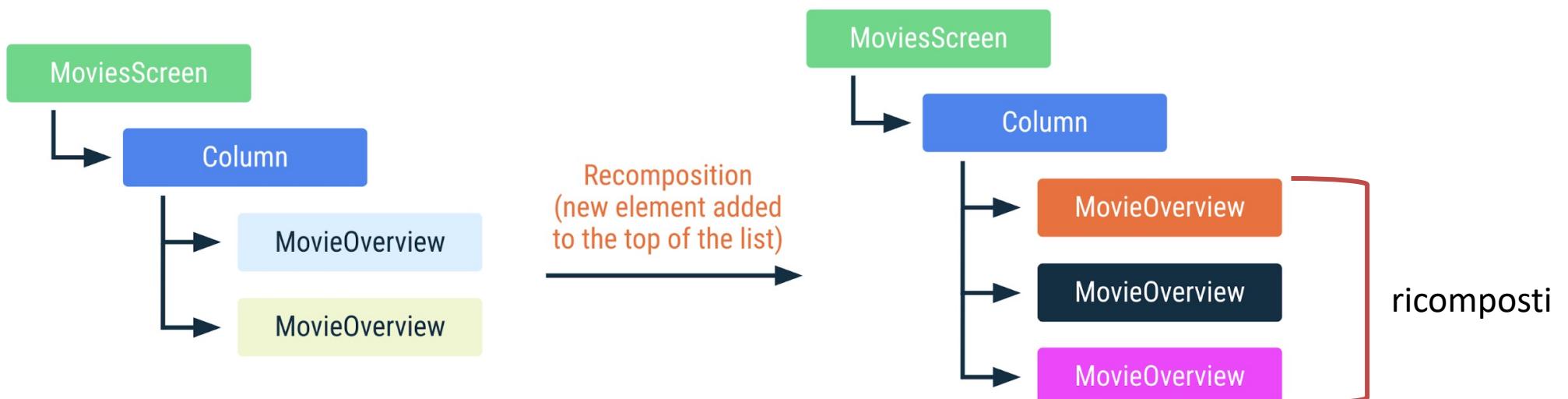
- Quando si richiama un composable più volte dallo stesso punto, Compose non ha informazioni per identificare in modo univoco ogni chiamata a quel composable, quindi **l'ordine di esecuzione** viene usato in aggiunta al **punto di chiamata** per mantenere le istanze distinte.

```
@Composable
fun MoviesScreen(movies: List<Movie>) {
    Column {
        for (movie in movies) {
            // MovieOverview composables are placed in Composition given its
            // index position in the for loop
            MovieOverview(movie)
        }
    }
}
```



Ciclo di vita dei composable

- La modifica della lista dei film, tramite **aggiunta in testa o al centro, rimozione o riordino**, causerà una ricomposizione in tutte le chiamate a MovieOverview il cui parametro di ingresso ha cambiato posizione nell'elenco.



Ciclo di vita dei composable

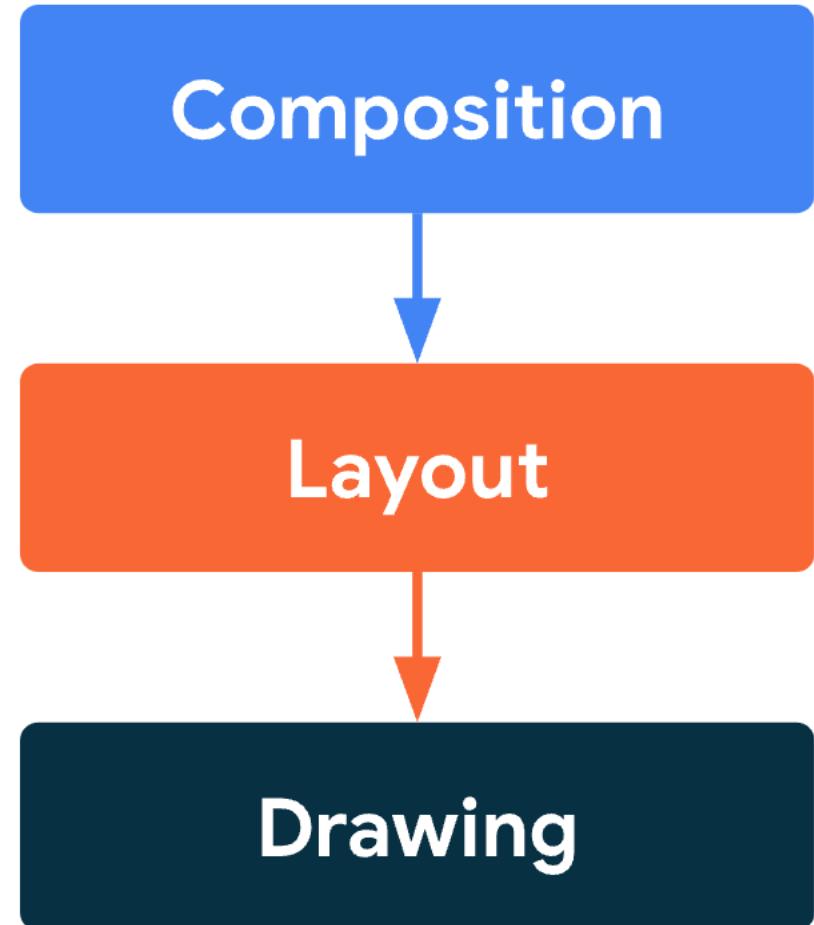
- Se però gli elementi sono caratterizzati da un **id univoco**, se non sono stati modificati non verranno ricomposti

```
@Composable
fun MoviesScreen(movies: List<Movie>) {
    Column {
        for (movie in movies) {
            key(movie.id) { // Unique ID for this movie
                MovieOverview(movie)
            }
        }
    }
}
```

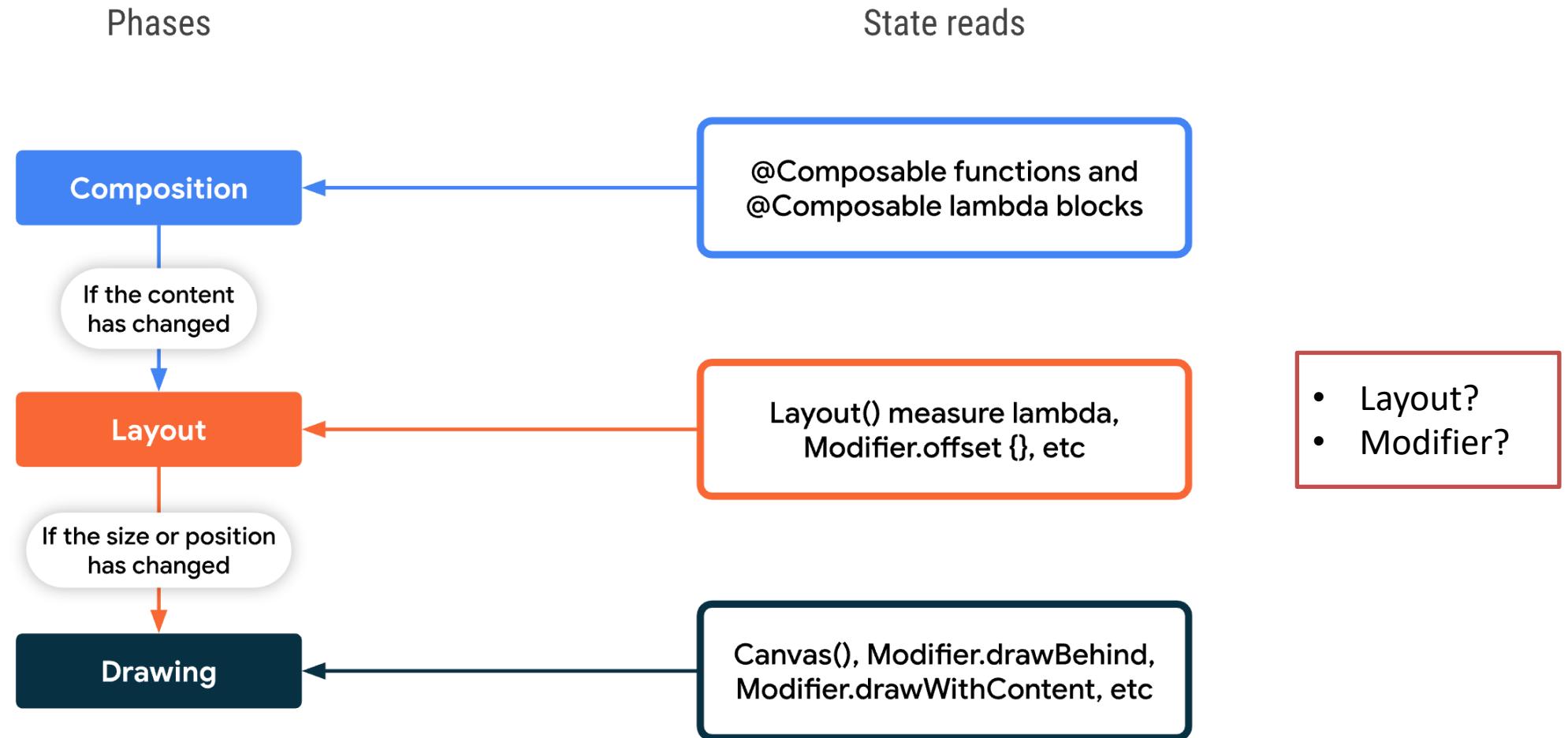


Fasi di Compose

1. **Composizione:** quale interfaccia utente mostrare. Compose esegue funzioni composable e crea una descrizione dell'interfaccia utente.
2. **Layout:** dove posizionare l'interfaccia utente. Questa fase consiste in due passaggi: *misurazione* e *posizionamento*. Gli elementi di layout misurano e posizionano se stessi ed eventuali elementi figli in coordinate 2D, per ogni nodo dell'albero di layout.
3. **Disegno:** come viene eseguito il rendering. Gli elementi dell'interfaccia utente vengono disegnati su una tela, di solito lo schermo di un dispositivo.



Fasi di Compose



Composable functions

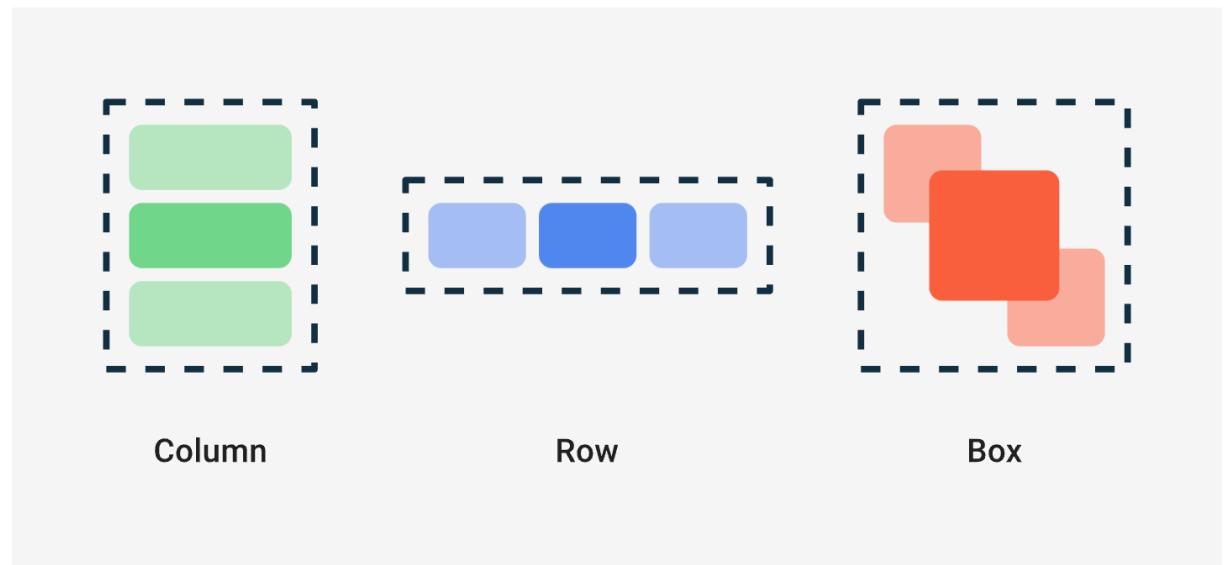
- Le **funzioni composable** sono i **basic building block** di Compose
- E' una funzione che emette *Unit* che descrive una parte dell'interfaccia. La funzione prende alcuni input e genera ciò che viene mostrato sullo schermo.
- Una funzione può emettere diversi elementi dell'UI. Tuttavia, se non si forniscono indicazioni su come devono essere disposti, Compose potrebbe disporre gli elementi in modo errato.

```
@Composable
fun ArtistCard() {
    Text("Alfred Sisley")
    Text("3 minutes ago")
}
```

3 minutes ago
Alfred Sisley

Layout

- Column
- Row
- Box
- BoxWithConstraints



Column

```
@Composable
fun ArtistCard() {
    Column {
        Text("Alfred Sisley")
        Text("3 minutes ago")
    }
}
```

Come verranno posizionati gli elementi?

Column

- Posiziona gli elementi verticalmente nell'interfaccia

```
@Composable
fun ArtistCard() {
    Column {
        Text("Alfred Sisley")
        Text("3 minutes ago")
    }
}
```

Alfred Sisley
3 minutes ago

Row

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

Come verranno posizionati gli elementi?

Row

- Posiziona gli elementi orizzontalmente

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



Alfred Sisley
3 minutes ago

Box

```
@Composable
fun ArtistAvatar(artist: Artist) {
    Box {
        Image(/*...*/)
        Icon(/*...*/)
    }
}
```

Come verranno posizionati gli elementi?

Box

- Posiziona gli elementi uno sopra l'altro

```
@Composable
fun ArtistAvatar(artist: Artist) {
    Box {
        Image(/*...*/)
        Icon(/*...*/)
    }
}
```



Alignment

- È possibile dare un **allineamento** verticale o orizzontale all'interno di questi layout

```
@Composable
fun ArtistCard(artist: Artist) {
    Row(
        verticalAlignment = Alignment.CenterVertically,
        horizontalArrangement = Arrangement.End
    ) {
        Image(/*...*/)
        Column { /*...*/ }
    }
}
```



Modificatori

- Per modificare un composable è possibile utilizzare i modificatori (modifier) che consentono di:
 - Cambiarne le dimensioni, il layout, il comportamento e l'aspetto
 - Aggiungere informazioni, come le label di accessibilità
 - Elaborare l'input dell'utente
 - Aggiungere interazioni, come rendere un elemento cliccabile, scrollabile, trascinabile o zoomabile.

```
import androidx.compose.ui.Modifier

@Composable
private fun Greeting(name: String) {
    Column(modifier = Modifier.padding(24.dp)) {  
        Text(text = "Hello,")  
        Text(text = name)  
    }
}
```



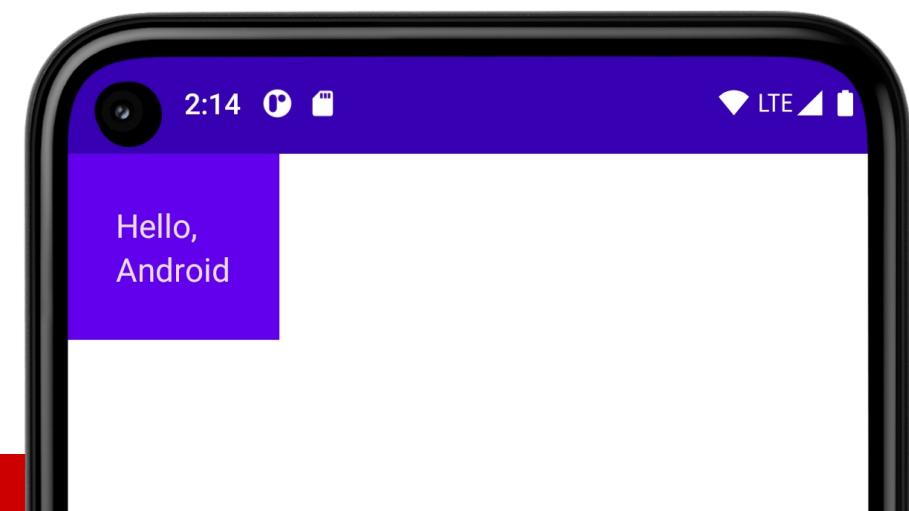
Padding???

Modificatori

- Per modificare un composable è possibile utilizzare i modificatori che consentono di:
 - Cambiarne le dimensioni, il layout, il comportamento e l'aspetto
 - Aggiungere informazioni, come le label di accessibilità
 - Elaborare l'input dell'utente
 - Aggiungere interazioni, come rendere un elemento cliccabile, scrollabile, trascinabile o zoomabile.

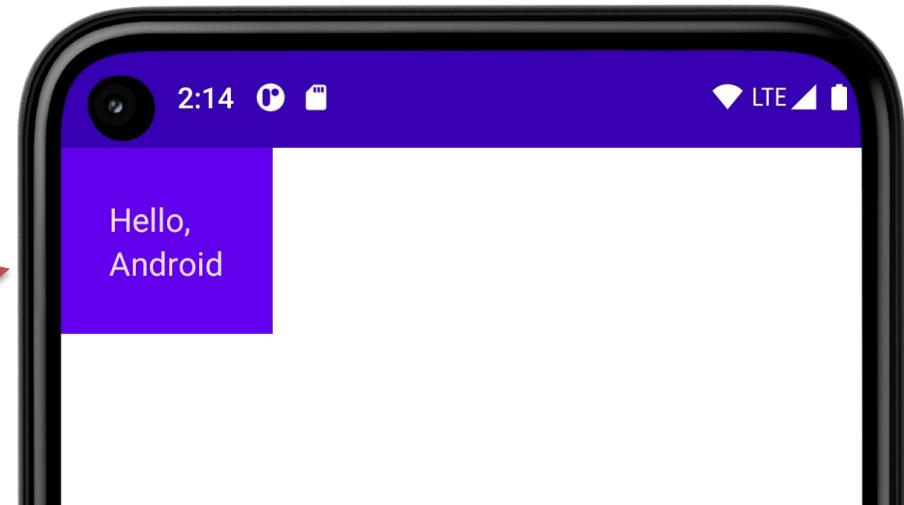
```
import androidx.compose.ui.Modifier

@Composable
private fun Greeting(name: String) {
    Column(modifier = Modifier.padding(24.dp)) {
        Text(text = "Hello, ")
        Text(text = name)
    }
}
```



Modificatori

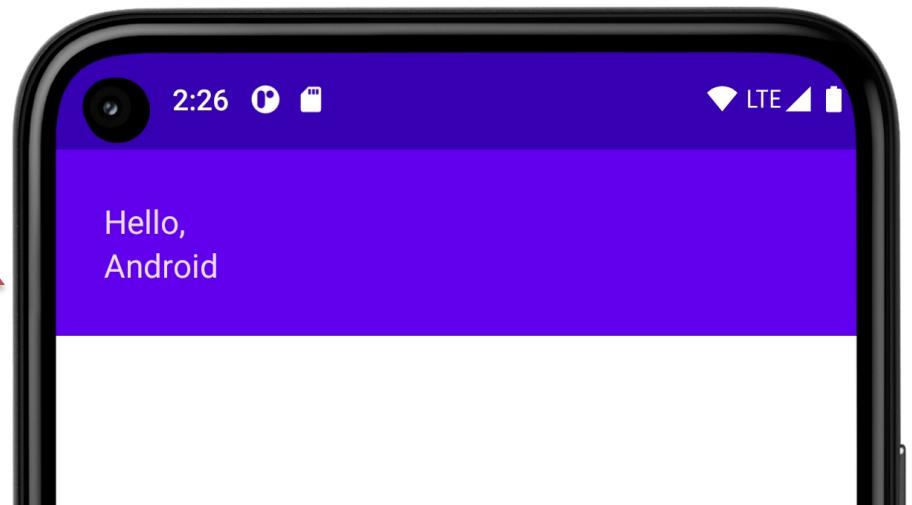
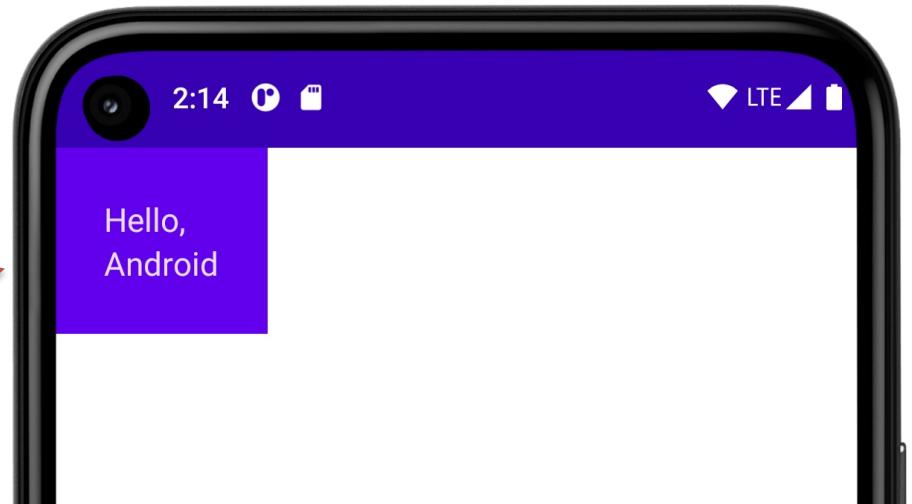
```
@Composable  
private fun Greeting(name: String) {  
    Column(modifier = Modifier  
        .padding(24.dp)  
        .fillMaxWidth()) {  
        Text(text = "Hello, ")  
        Text(text = name)  
    }  
}
```



?????

Modificatori

```
@Composable  
private fun Greeting(name: String) {  
    Column(modifier = Modifier  
        .padding(24.dp)  
        .fillMaxWidth()) {  
        Text(text = "Hello,")  
        Text(text = name)  
    }  
}
```



Modificatori

- L'ordine delle funzioni modificatrici è significativo. Poiché ogni funzione apporta modifiche al modificatore restituito dalla funzione precedente, la sequenza influenza sul risultato finale.

```
@Composable
fun ArtistCard(/*...*/) {
    val padding = 16.dp
    Column(
        Modifier
            .clickable(onClick = onClick)
            .padding(padding)
            .fillMaxWidth()
    ) {
        // rest of the implementation
    }
}
```

```
@Composable
fun ArtistCard(/*...*/) {
    val padding = 16.dp
    Column(
        Modifier
            .padding(padding)
            .clickable(onClick = onClick)
            .fillMaxWidth()
    ) {
        // rest of the implementation
    }
}
```

Differenze????

Modificatori

- L'ordine delle funzioni modificatrici è significativo. Poiché ogni funzione apporta modifiche al modificatore restituito dalla funzione precedente, la sequenza influenza sul risultato finale.

```
@Composable
fun ArtistCard(/*...*/) {
    val padding = 16.dp
    Column(
        Modifier
            .clickable(onClick = onClick)
            .padding(padding)
            .fillMaxWidth()
    ) {
        // rest of the implementation
    }
}
```

```
@Composable
fun ArtistCard(/*...*/) {
    val padding = 16.dp
    Column(
        Modifier
            .padding(padding)
            .clickable(onClick = onClick)
            .fillMaxWidth()
    ) {
        // rest of the implementation
    }
}
```

Nel primo caso tutta l'area è cliccabile (anche il padding), nel secondo caso il padding è escluso

Tipi di modificatori

- Size e requiredSize

```
@Composable
fun ArtistCard(/*...*/) {
    Row(
        modifier = Modifier.size(width = 400.dp, height = 100.dp)
    ) {
        Image(
            /*...*/
            modifier = Modifier.requiredSize(150.dp)
        )
        Column { /*...*/ }
    }
}
```

anche con l'altezza del genitore impostata a 100.dp, l'altezza dell'immagine sarà 150.dp, poiché il modificatore requiredSize ha la precedenza.

Tipi di modificatori

- **Fill**

```
@Composable
fun ArtistCard(/*...*/) {
    Row(
        modifier = Modifier.size(width = 400.dp, height = 100.dp)
    ) {
        Image(
            /*...*/
            modifier = Modifier.fillMaxHeight()
        )
        Column { /*...*/ }
    }
}
```



Alfred Sisley
3 minutes ago

- `fillMaxHeight`
- `fillMaxSize`
- `fillMaxWidth`

Tipi di modificatori

- padding

```
@Composable
fun ArtistCard(artist: Artist) {
    Row(/*...*/) {
        Column {
            Text(
                text = artist.name,
                modifier = Modifier.paddingFromBaseline(top = 50.dp)
            )
            Text(artist.lastSeenOnline)
        }
    }
}
```

Alfred Sisley
3 minutes ago

Se si vuole aggiungere un padding sopra la baseline del testo, in modo da ottenere una distanza specifica dalla parte superiore del layout alla baseline, utilizzare **paddingFromBaseline**

Tipi di modificatori

- **Offset**: per posizionare un layout rispetto alla sua posizione originale sugli assi x e y.
- Gli offset possono essere sia positivi che negativi.
- La differenza tra padding e offset è che l'aggiunta di un offset a un composable non ne modifica le misure.

```
@Composable
fun ArtistCard(artist: Artist) {
    Row(/*...*/) {
        Column {
            Text(artist.name)
            Text(
                text = artist.lastSeenOnline,
                modifier = Modifier.offset(x = 4.dp)
            )
        }
    }
}
```



Alfred Sisley
3 minutes ago

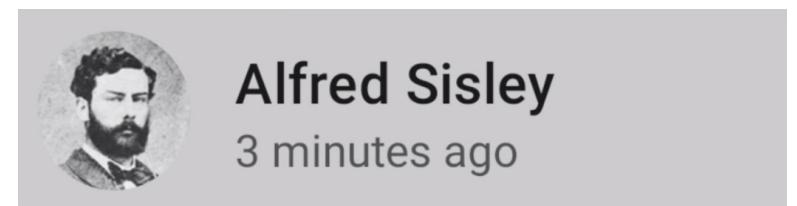
Il modificatore di offset viene applicato **orizzontalmente** in base alla **direzione del layout**.

In un contesto da sinistra a destra, un offset positivo sposta l'elemento a destra, mentre in un contesto da destra a sinistra lo sposta a sinistra.

Tipi di modificatori

- **matchParentSize**: se si desidera che un layout figlio abbia le stesse dimensioni di un riquadro genitore
- È disponibile solo all'interno di un ambito **Box**, cioè si applica solo ai figli diretti dei compositori Box.

```
@Composable
fun MatchParentSizeComposable() {
    Box {
        Spacer(Modifier.matchParentSize()).background(Color.LightGray))
        ArtistCard()
    }
}
```



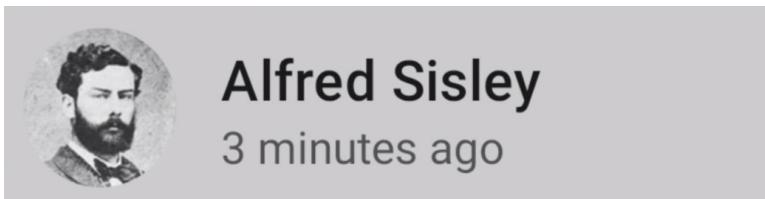
Spacer prende le sue dimensioni dal suo genitore **Box**, che a sua volta prende le sue dimensioni dal figlio più grande, in questo caso **ArtistCard**.

Tipi di modificatori

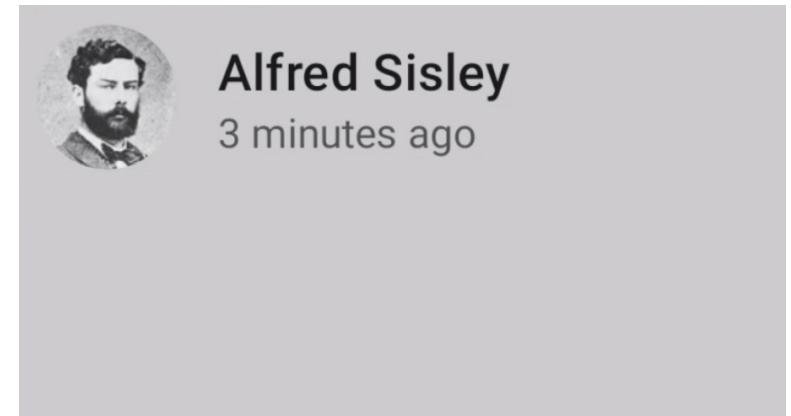
- Se si usasse **fillMaxSize** invece di `matchParentSize`, `Spacer` prenderebbe tutto lo spazio disponibile per il genitore, causando a sua volta l'espansione del genitore e il riempimento di tutto lo spazio disponibile.

```
@Composable
fun MatchParentSizeComposable() {
    Box {
        Spacer(Modifier.matchParentSize()).background(Color.LightGray)
        ArtistCard()
    }
}
```

matchParentSize



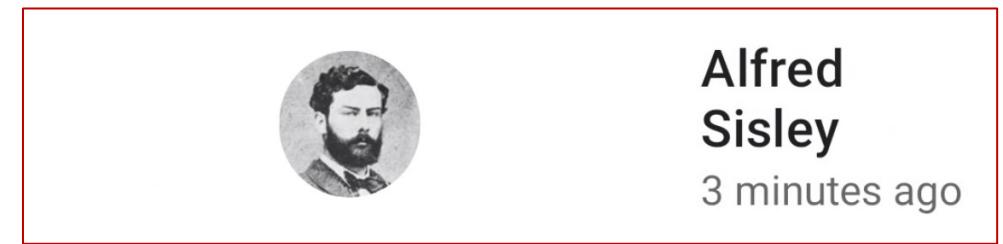
fillMaxSize



Tipi di modificatori

- Per impostazione predefinita, la dimensione di un composable è definita dal contenuto.
- È possibile impostare una dimensione in modo che sia flessibile all'interno del suo genitore, utilizzando il modificatore **weight**, disponibile solo in **Row** e **Column**.

```
@Composable
fun ArtistCard(/*...*/) {
    Row(
        modifier = Modifier.fillMaxWidth()
    ) {
        Image(
            /*...*/
            modifier = Modifier.weight(2f)
        )
        Column(
            modifier = Modifier.weight(1f)
        ) {
            /*...*/
        }
    }
}
```



Prendiamo una Row che contiene due composable. Al primo viene dato il doppio del peso del secondo, quindi la sua larghezza è doppia. Se la Row è larga 210.dp, la prima è larga 140.dp e la seconda 70.dp

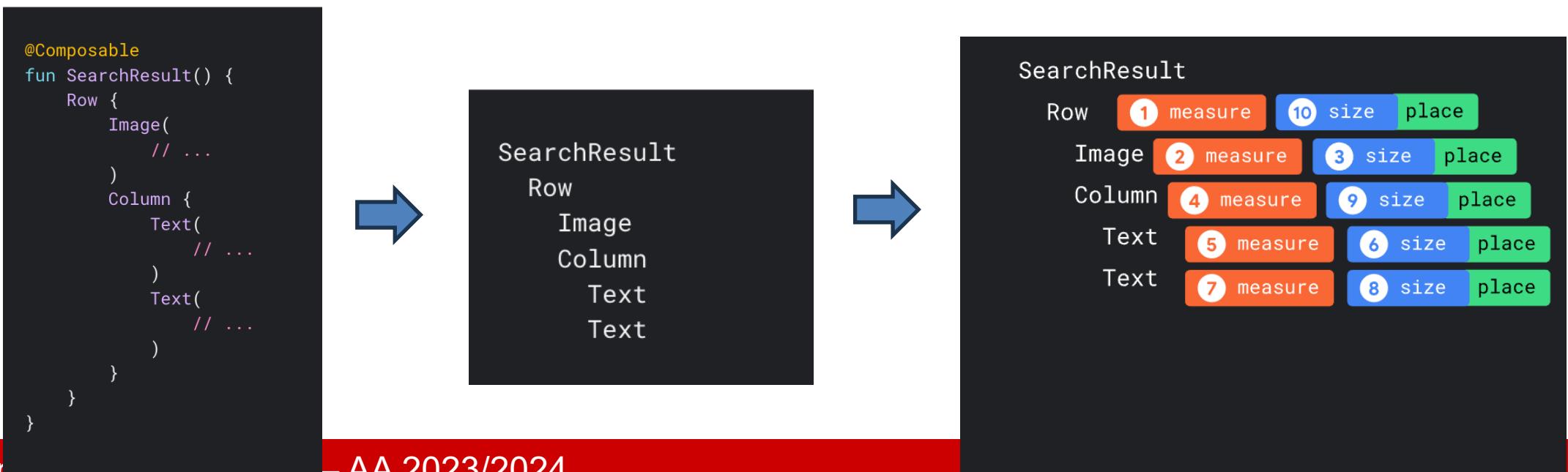
Modificatori riutilizzabili

```
val reusableModifier = Modifier  
    .fillMaxWidth()  
    .background(Color.Red)  
    .padding(12.dp)
```

```
@Composable  
fun AuthorField() {  
    HeaderText(  
        // ...  
        modifier = reusableModifier  
    )  
    SubtitleText(  
        // ...  
        modifier = reusableModifier  
    )  
}
```

Modello di layout

- Il **modello di layout** è rappresentato da un **albero**, dove a ogni nodo viene prima chiesto di misurare se stesso, poi di misurare gli eventuali figli in modo ricorsivo, passando le dimensione ai figli lungo l'albero. I nodi foglia vengono dimensionati e posizionati, e le dimensioni e le istruzioni di posizionamento vengono trasmesse nuovamente all'albero.
- In breve, i genitori si misurano prima dei figli, ma vengono dimensionati e posizionati dopo i figli.



SearchResult



In the `SearchResult` example, the UI tree layout follows this order:

1. The root node `Row` is asked to measure.
2. The root node `Row` asks its first child, `Image`, to measure.
3. `Image` is a leaf node (that is, it has no children), so it reports a size and returns placement instructions.
4. The root node `Row` asks its second child, `Column`, to measure.
5. The `Column` node asks its first `Text` child to measure.
6. The first `Text` node is a leaf node, so it reports a size and returns placement instructions.
7. The `Column` node asks its second `Text` child to measure.
8. The second `Text` node is a leaf node, so it reports a size and returns placement instructions.
9. Now that the `Column` node has measured, sized, and placed its children, it can determine its own size and placement.
10. Now that the root node `Row` has measured, sized, and placed its children, it can determine its own size and placement.

Responsive layouts

- Per conoscere i vincoli provenienti dal genitore e progettare il layout di conseguenza, è possibile utilizzare un **BoxWithConstraints**.
- I vincoli di misura possono essere usati per comporre diversi layout a seconda delle configurazioni dello schermo.

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

Liste

- Se il caso d'uso non richiede scrolling, si può usare una Column o una Row, iterando sull'elemento

```
@Composable
fun MessageList(messages: List<Message>) {
    Column {
        messages.forEach { message ->
            MessageRow(message)
        }
    }
}
```

Liste

- Se il caso d'uso non richiede scrolling, si può usare una Column o una Row, iterando sull'elemento
- È anche possibile rendere la colonna scrollabile utilizzando il modificatore **verticalScroll()**.

```
@Composable
fun MessageList(messages: List<Message>) {
    Column {
        messages.forEach { message ->
            MessageRow(message)
        }
    }
}
```

```
@Composable
private fun ScrollBoxesSmooth() {

    // Smoothly scroll 100px on first composition
    val state = rememberScrollState()
    LaunchedEffect(Unit) { state.animateScrollTo(100) }

    Column(
        modifier = Modifier
            .background(Color.LightGray)
            .size(100.dp)
            .padding(horizontal = 8.dp)
            .verticalScroll(state)
    ) {
        repeat(10) {
            Text("Item $it", modifier = Modifier.padding(2.dp))
        }
    }
}
```

Liste

- Se è necessario visualizzare un gran numero di elementi (o un elenco di lunghezza sconosciuta), l'uso di un layout come Column può causare **problemi di prestazioni**, poiché tutti gli elementi saranno composti e disposti indipendentemente dal fatto che siano visibili o meno.
- Compose fornisce un insieme di componenti che compongono e dispongono solo gli elementi visibili:
 - LazyColumn per una lista scrollabile verticalmente;
 - LazyRow per una lista scrollabile orizzontalmente.

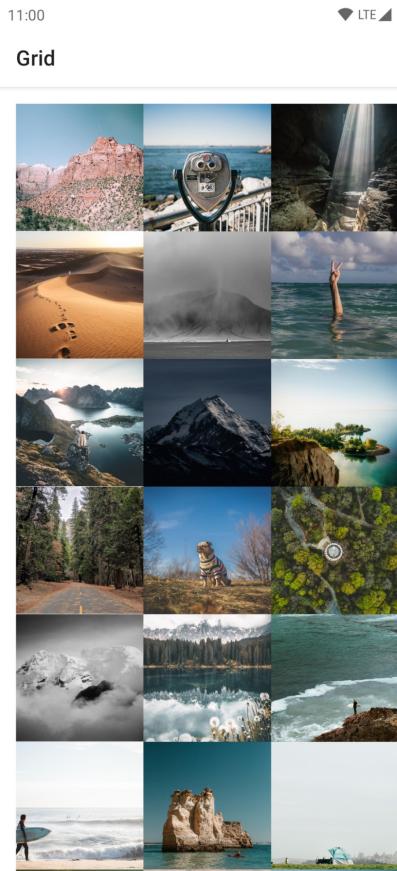
Liste

```
LazyColumn {  
    // Add a single item  
    item {  
        Text(text = "First item")  
    }  
  
    // Add 5 items  
    items(5) { index ->  
        Text(text = "Item: $index")  
    }  
  
    // Add another single item  
    item {  
        Text(text = "Last item")  
    }  
}
```

```
import androidx.compose.foundation.lazy.items  
  
@Composable  
fun MessageList(messages: List<Message>) {  
    LazyColumn {  
        items(messages) { message ->  
            MessageRow(message)  
        }  
    }  
}
```

Griglie

- I composable **LazyVerticalGrid** e **LazyHorizontalGrid** supportano la visualizzazione degli elementi in una griglia scrollabile (verticale o orizzontale).



```
@Composable
fun PhotoGrid(photos: List<Photo>) {
    LazyVerticalGrid(
        columns = GridCells.Adaptive(minSize = 128.dp)
    ) {
        items(photos) { photo ->
            PhotoItem(photo)
        }
    }
}
```

Permette di impostare ogni colonna con una larghezza minima di 128.dp

Griglie

- Si può specificare una **larghezza** per gli elementi e poi la griglia si adatta al maggior numero possibile di colonne (della stessa larghezza). Questo modo adattivo di dimensionamento è utile per visualizzare insiemi di elementi su schermi di dimensioni diverse.
- Se si conosce il **numero esatto di colonne** da utilizzare, è possibile fornire un'istanza di GridCells.Fixed contenente il numero di colonne richiesto.

Liste e griglie

```
LazyColumn(  
    contentPadding = PaddingValues(horizontal = 16.dp, vertical = 8.dp),  
) {  
    // ...  
}
```

```
LazyColumn(  
    verticalArrangement = Arrangement.spacedBy(4.dp),  
) {  
    // ...  
}
```

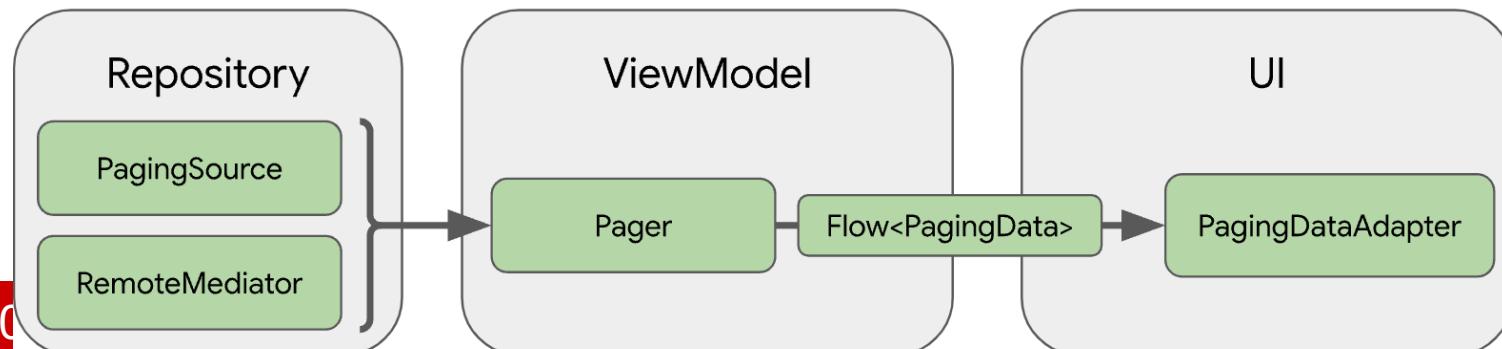
```
    LazyVerticalGrid(  
        columns = GridCells.Fixed(2),  
        verticalArrangement = Arrangement.spacedBy(16.dp),  
        horizontalArrangement = Arrangement.spacedBy(16.dp)  
    ) {  
        items(data) { item ->  
            Item(item)  
        }  
    }
```



Part of [Android Jetpack](#).

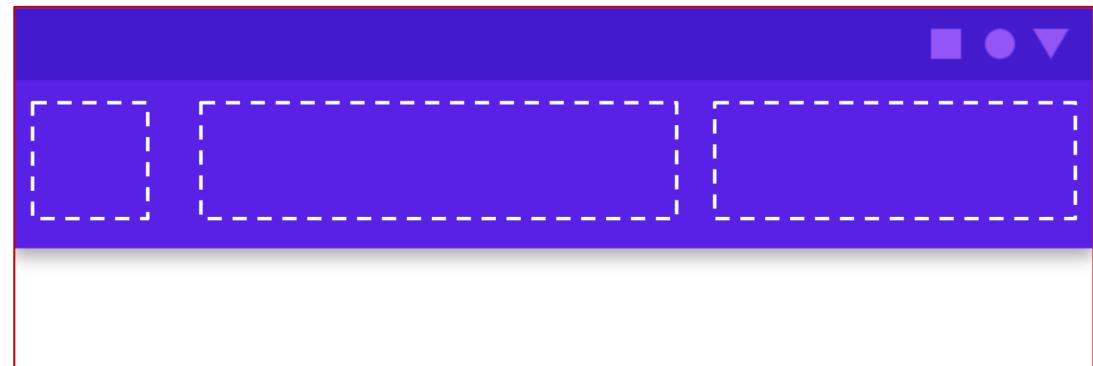
Paging

- La libreria **Paging** consente di supportare grandi liste di elementi, caricando e visualizzando piccole porzioni quando necessario.
- Include:
 - **Caching** in memoria per i dati paginati. Questo assicura che l'applicazione utilizzi le risorse di sistema in modo efficiente mentre lavora con i dati paginati.
 - **Deduplicazione** delle richieste incorporata, per garantire che l'applicazione utilizzi in modo efficiente la larghezza di banda della rete e le risorse del sistema.
 - Supporto per le **coroutine** Kotlin e Flow, nonché per LiveData e RxJava [lo vedremo poi!].
 - Supporto integrato per la **gestione degli errori**, comprese le funzionalità di refresh e retry.



Layout e Material Design

- Compose ha elementi basati sul Material Design con la dipendenza **androidx.compose.material:material** (inclusa quando si crea un progetto Compose in Android Studio) per semplificare la creazione dell'interfaccia utente.
- Vengono forniti elementi come **Drawer**, **FloatingActionButton** e **TopAppBar**.
- I componenti Material usano **layout slot-based**, un modello introdotto da Compose per personalizzare i componenti. Gli slot lasciano uno spazio vuoto nell'interfaccia utente che lo sviluppatore può riempire a suo piacimento. Ad esempio, questi sono gli slot che si possono personalizzare in una TopAppBar:



Esempio

- **Scaffold** consente di implementare un'interfaccia utente con la struttura di base del layout Material.
- Fornisce slot per i più comuni componenti, come TopAppBar, BottomAppBar, FloatingActionButton e Drawer.
- Utilizzando Scaffold, è facile assicurarsi che questi componenti siano posizionati correttamente e che lavorino insieme in modo corretto.

```
@Composable
fun HomeScreen(/*...*/) {
    Scaffold(
        drawerContent = { /*...*/ },
        topBar = { /*...*/ },
        content = { /*...*/ }
    )
}
```

Material Design

- Compose utilizza **temi** ed **elementi** di material.
- Lista di componenti Material:

<https://developer.android.com/reference/kotlin/androidx/compose/material/package-summary#components>

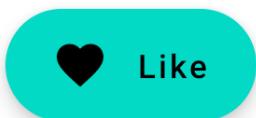
```
@Composable
fun MyApp() {
    MaterialTheme {
        // Material Components like Button, Card, Switch, etc.
    }
}
```

```
Button(
    onClick = { /* ... */ },
    // Uses ButtonDefaults.ContentPadding by default
    contentPadding = PaddingValues(
        start = 20.dp,
        top = 12.dp,
        end = 20.dp,
        bottom = 12.dp
    )
) {
    // Inner content including an icon and a text label
    Icon(
        Icons.Filled.Favorite,
        contentDescription = "Favorite",
        modifier = Modifier.size(ButtonDefaults.IconSize)
    )
    Spacer(Modifier.size(ButtonDefaults.IconSpacing))
    Text("Like")
}
```

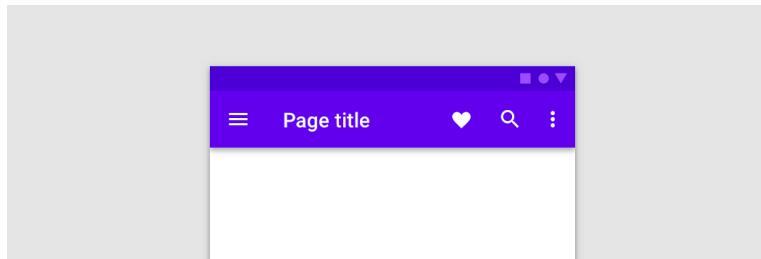


Material Design - Esempi

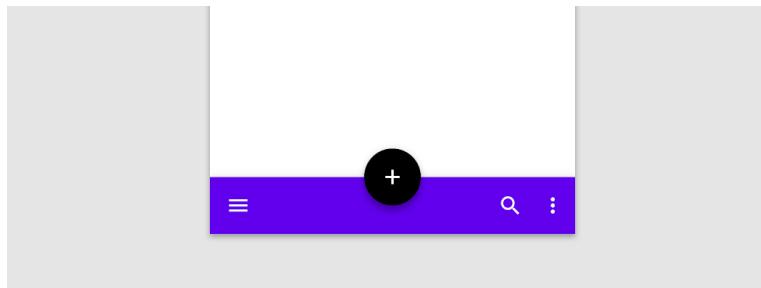
```
ExtendedFloatingActionButton(  
    onClick = { /* ... */ },  
    icon = {  
        Icon(  
            Icons.Filled.Favorite,  
            contentDescription = "Favorite"  
        )  
    },  
    text = { Text("Like") }  
)
```



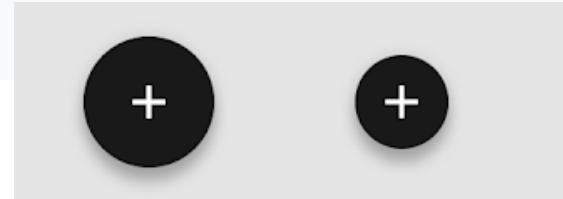
```
Scaffold(  
    topBar = {  
        TopAppBar { /* Top app bar content */ }  
    }  
) {  
    // Screen content  
}
```



```
Scaffold(  
    bottomBar = {  
        BottomAppBar { /* Bottom app bar content */ }  
    }  
) {  
    // Screen content  
}
```



```
Scaffold(  
    floatingActionButton = {  
        FloatingActionButton(onClick = { /* FAB content */ })  
    },  
    // Defaults to FabPosition.End  
    floatingActionButtonPosition = FabPosition.Center  
) {  
    // Screen content  
}
```



Material Design - Floating action buttons

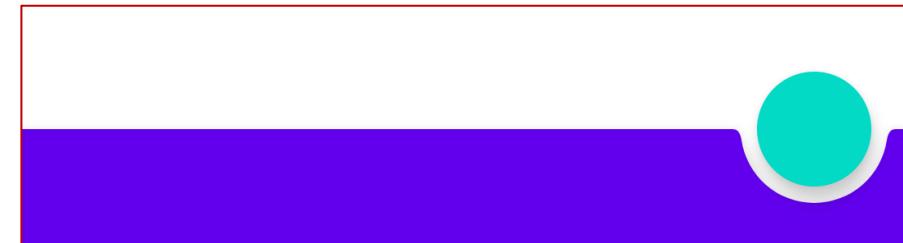
```
Scaffold(  
    floatingActionButton = {  
        FloatingActionButton(onClick = { /* ... */ }) {  
            /* FAB content */  
        }  
    },  
    // Defaults to false  
    isFloatingActionButtonDocked = true,  
    bottomBar = {  
        BottomAppBar { /* Bottom app bar content */ }  
    }  
) {  
    // Screen content  
}
```

false
true



Material Design - Floating action buttons

```
Scaffold(  
    floatingActionButton = {  
        FloatingActionButton(onClick = { /* ... */ }) {  
            /* FAB content */  
        }  
    },  
    isFloatingActionButtonDocked = true,  
    bottomBar = {  
        BottomAppBar(  
            // Defaults to null, that is, No cutout  
            cutoutShape = MaterialTheme.shapes.small.copy(  
                CornerSize(percent = 50)  
            )  
        ) {  
            /* Bottom app bar content */  
        }  
    }  
) {  
    // Screen content  
}
```



Compose e Material 3

- Approfondimenti:
 - <https://developer.android.com/jetpack/compose/designsystems/material3>
 - <https://developer.android.com/jetpack/compose/layouts/material>
- Vedremo qualcosa nei prossimi laboratori!

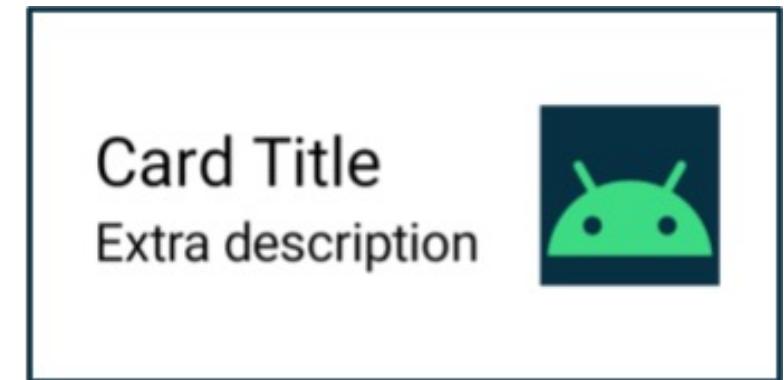
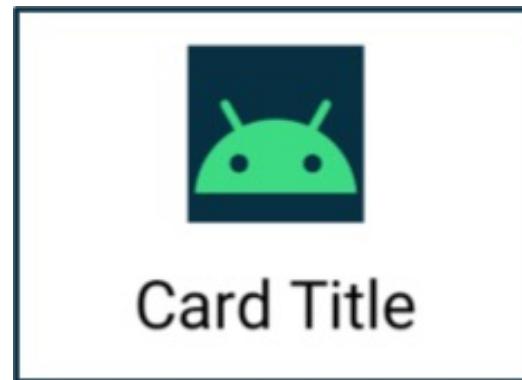
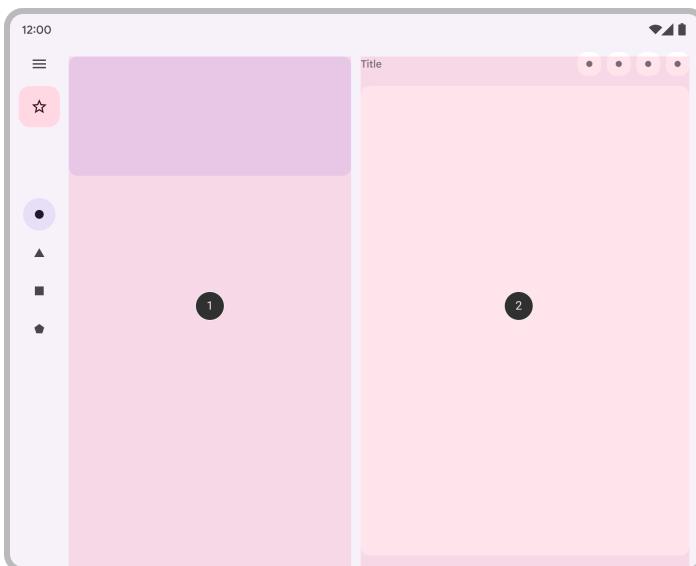
Layout adattivi

- L'interfaccia utente deve essere reattiva per tenere conto delle diverse dimensioni dello schermo, degli orientamenti etc...
- Un **layout adattivo** cambia in base allo spazio disponibile sullo schermo. Queste modifiche vanno da semplici aggiustamenti del layout per riempire lo spazio, fino alla modifica completa del layout per sfruttare lo spazio aggiuntivo.
- Come toolkit dichiarativo, Jetpack Compose è adatto alla progettazione e all'implementazione di layout che si adattano a rendere i contenuti in modo diverso a seconda delle dimensioni.



Layout adattivi

- Soprattutto con l'arrivo dei foldables, è aumentato l'interesse per lo sviluppo di interfacce per schermi adattivi e/o di grandi dimensioni
- <https://developer.android.com/jetpack/compose/layouts/adaptive>
- <https://developer.android.com/large-screens>

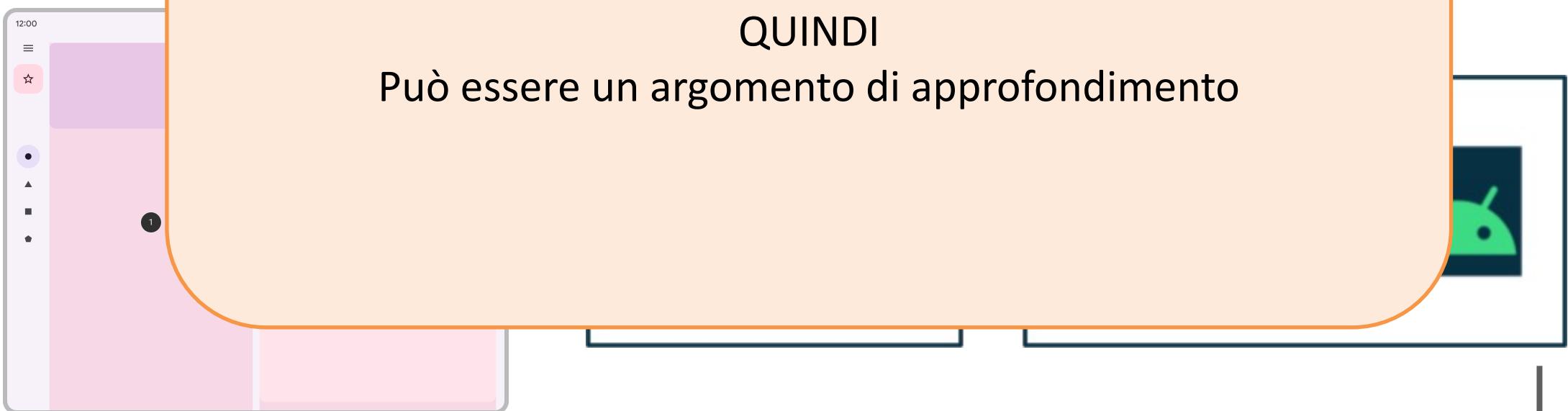


Layout adattivi

- Soprattutto con l'arrivo dei foldables, è aumentato l'interesse per lo sviluppo di layout adattivi
- <https://developer.android.com/guide/topics/ui/affordances>
- <https://developer.android.com/guide/topics/ui/affordances>

Non riusciremo a vederli in dettaglio
QUINDI

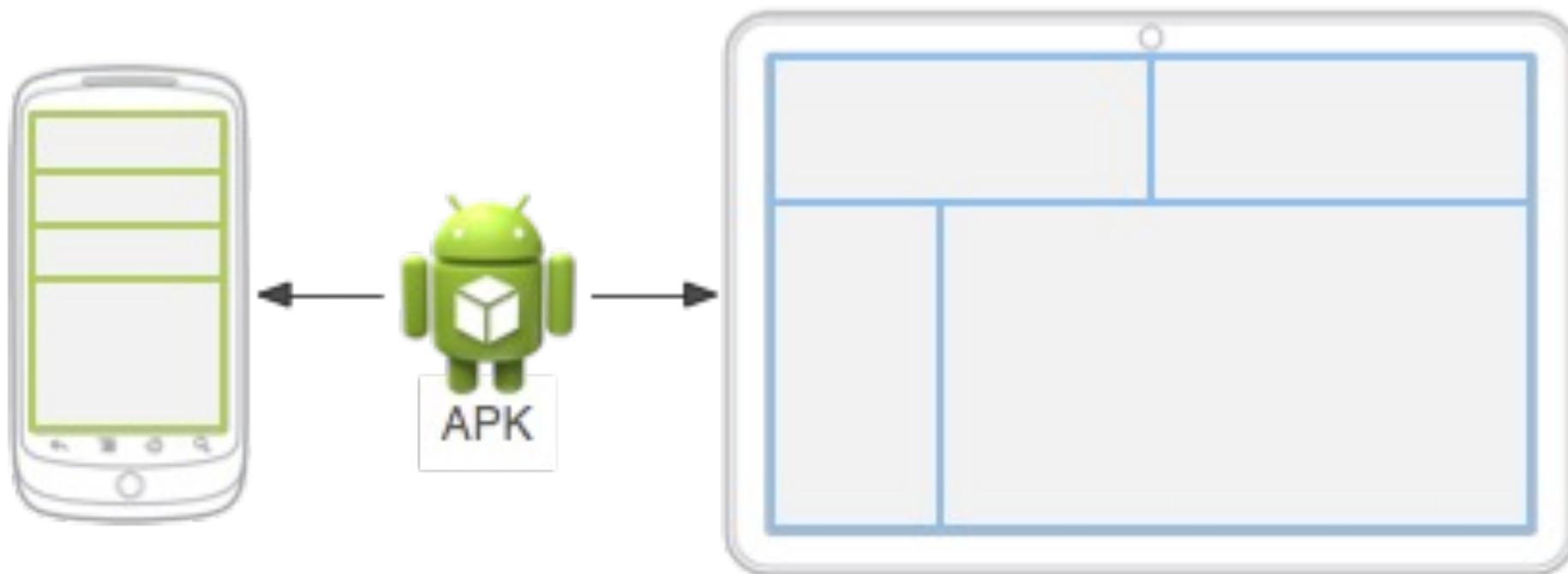
Può essere un argomento di approfondimento



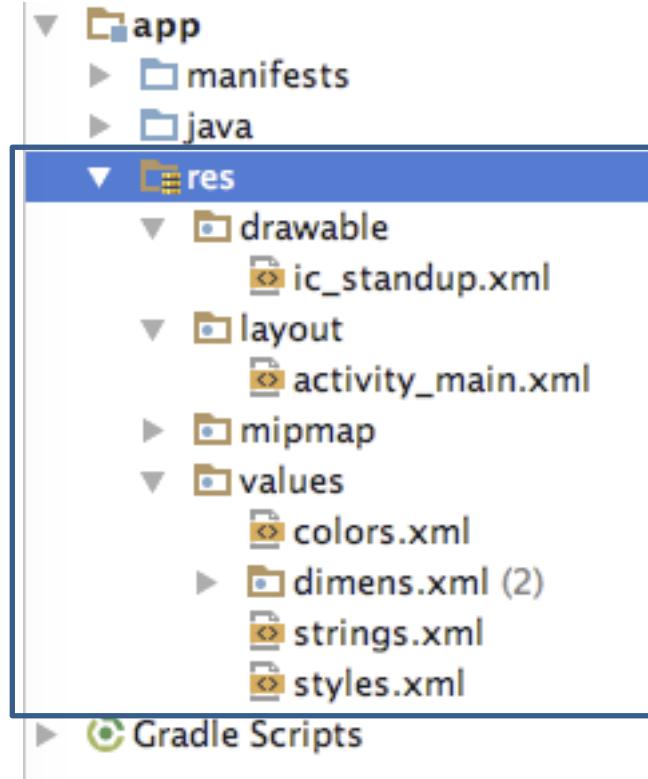
«Risorse» e Composable

- Jetpack Compose può accedere alle risorse definite nel tuo progetto Android.
- Le **risorse** sono file aggiuntivi e contenuto statico utilizzati dal codice, ad esempio bitmap, definizioni di layout, stringhe dell'interfaccia utente, istruzioni di animazione e altro ancora.
- In altre parole, le **risorse** sono tutto ciò che non è codice, come contenuti statici e file addizionali
 - File di layout XML, stringhe, immagini, audio/video, animazioni, etc ...
- Importante il loro uso per:
 - Separare la presentazione dei dati dalla loro gestione
 - Gestire specifiche configurazioni dei devices (lingua, dimensione schermo etc...)
- In fase di esecuzione, Android utilizza la risorsa appropriata in base alla configurazione corrente!! - molto importante per orientamento, lingua, ecc.
- Dopo aver esternalizzato le risorse dell'app, è possibile accedervi utilizzando gli ID risorsa generati **nella classe R** del progetto

Perché sono importanti?



Nel progetto Android



- Si trovano dentro la cartella «res»

Struttura

```
MyProject/  
    src/  
        MyActivity.java  
    res/  
        drawable/  
            graphic.png  
        layout/  
            main.xml  
            info.xml  
        mipmap/  
            icon.png  
        values/  
            strings.xml
```

→ Contiene tutte le stringhe, come nomi dei bottoni, label, testi di default, etc

Raggruppare per tipologia nelle varie sotto cartelle

I nomi delle cartelle non sono casuali, anzi, sono ben definiti da Android!

Struttura

```
MyProject/  
    src/  
        MyActivity.java  
    res/  
        drawable/  
            graphic.png  
        layout/  
            main.xml  
            info.xml  
        mipmap/  
            icon.png  
        values/  
            strings.xml
```

→ Contiene tutte le stringhe, come nomi dei bottoni, label, testi di default, etc

Mai salvare i file direttamente nella cartella res (invece che in una delle sottocartelle definite)! Comporta un errore in fase di compilazione!

Resource directories

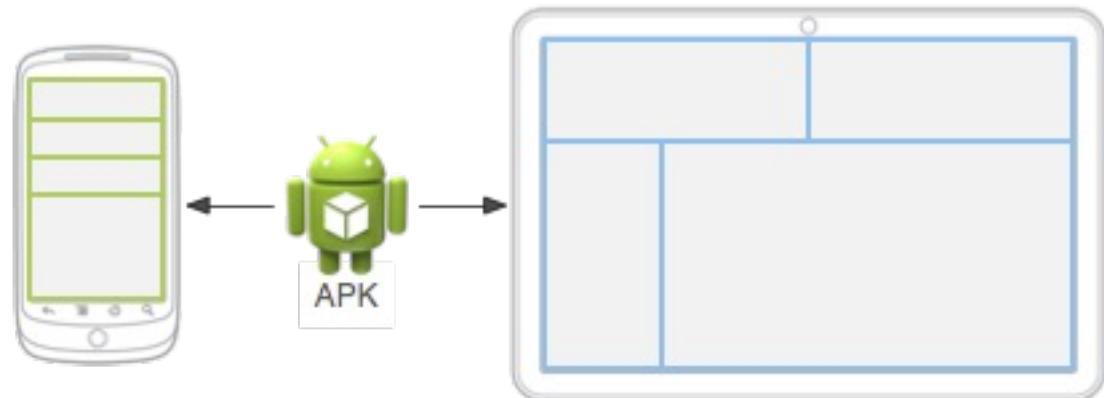
- **animator/** - property animations (durata, velocità, ecc.) (info: <https://developer.android.com/guide/topics/graphics/prop-animation>)
- **anim/** - tween animations (animazioni interpolate) (<https://developer.android.com/guide/topics/graphics/view-animation#tween-animation>)
- **color/** - a state list of colors (ovvero colori diversi in base allo stato della View)
- **drawable/** - Bitmap files (.png, .jpg, .gif)
- **mipmap/** - launcher icons
- **layout/** - user interface layout
- **menu/** - app menus
- **raw/** - File da salvare nella loro forma grezza (row)
- **values/** - file xml che contengono semplici valori come strings, integers, e colors
- **xml/** - file XML da leggere a runtime (con [Resources.getXML\(\)](#))
- **font/** - Font files con estensione .ttf, .otf, or .ttc, o file XML che includono un elemento <font-family>

Resource directories

- I nomi delle cartelle che vi ho appena mostrato sono quelli di default!
- Quindi se non c'è definito altro, Android utilizzerà quelli per configurare la vostra app
- Domanda:
 - Come faccio a definire quindi configurazioni diverse? Esempio gestire stringhe per lingue diverse?

Supporto per diversi device

- Per specificare alternative di configuration-specific per un set di risorse creare una sottocartella in res/ con il nome del tipo
<resources_name>-<qualifier>
- Dove
 - *<resources_name>* sono i nomi delle sotto cartelle che abbiamo appena visto
 - *<qualifier>* specifica una configurazione (definite),
se ne può usare più di uno , tutti separati da **-**, seguendo un **ordine specifico!**
ES: **drawable-en-rUS-land**
 - Dettagli, in “Table 2”,
<https://developer.android.com/guide/topics/resources/providing-resources>



Esempio: Configurazione lingua

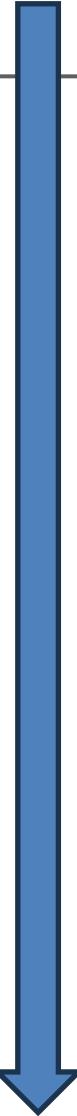
Language and region	Examples: en fr en-rUS fr-rFR fr-rCA b+en b+en+US b+es+419	<p>The language is defined by a two-letter ISO 639-1 language code, optionally followed by a two letter ISO 3166-1-alpha-2 region code (preceded by lowercase r).</p> <p>The codes are <i>not</i> case-sensitive; the r prefix is used to distinguish the region portion. You cannot specify a region alone.</p> <p>Android 7.0 (API level 24) introduced support for BCP 47 language tags, which you can use to qualify language- and region-specific resources. A language tag is composed from a sequence of one or more subtags, each of which refines or narrows the range of language identified by the overall tag. For more information about language tags, see Tags for Identifying Languages.</p> <p>To use a BCP 47 language tag, concatenate b+ and a two-letter ISO 639-1 language code, optionally followed by additional subtags separated by +.</p> <p>The language tag can change during the life of your app if the users change their language in the system settings. See Handling Runtime Changes for information about how this can affect your app during runtime.</p> <p>See Localization for a complete guide to localizing your app for other languages.</p> <p>Also see the getLocale() method, which provides the defined list of locales. This list includes the primary locale.</p>
---------------------	--	--

Tabella 2, <https://developer.android.com/guide/topics/resources/providing-resources>

Esempio: orintamento schermo

Screen orientation	<code>port</code> <code>land</code>	<ul style="list-style-type: none">• port: Device is in portrait orientation (vertical)• land: Device is in landscape orientation (horizontal) <p>This can change during the life of your app if the user rotates the screen. See Handling Runtime Changes for information about how this affects your app during runtime.</p> <p>Also see the orientation configuration field, which indicates the current device orientation.</p>
--------------------	--	---

Tabella 2, <https://developer.android.com/guide/topics/resources/providing-resources>



Configuration	Values Example	Description
MCC and MNC	mcc310, mcc208, etc	mobile country code (MCC)
Language and region	en, fr, en-rUS, etc	ISO 639-1 language code
smallestWidth	sw320dp, etc	shortest dimension of screen
Available width	w720dp, w320dp, etc	minimum available screen width
Available height	h720dp, etc	minimum available screen height
Screen size	small, normal, large	screen size expressed in dp
Screen aspect	long, notlong	aspect ratio of the screen
Screen orientation	port, land	screen orientation (can change!)
Screen pixel density (dpi)	ldpi, mdpi, hdpi	screen pixel density
Keyboard availability	keysexposed, etc	type of keyword
Primary text input method	nokeys, qwerty	availability of qwerty keyboard
Navigation key availability	navexposed, etc	navigation keys of the application
Platform Version (API level)	v3, v4, v7, etc	API supported by the device

Esempio

res/

drawable/

icon.png

background.png

drawable-hdpi/

icon.png

background.png

high-density screen

NB: I nomi delle risorse (dei file) sono le stesse!!!

Così l'ID della resource è lo stesso e Android seleziona la versione che meglio soddisfa il device in uso

Regole

- Si possono usare diversi quantificatori insieme, separati da -
- **Occorre seguire l'ordine definito in Tabella 2**
(<https://developer.android.com/guide/topics/resources/providing-resources>)
 - Quindi: *drawable-hdpi-port/* OK/No
 - Mentre: *drawable-port-hdpi/* OK/No??
- Non si possono creare cartelle indentate del tipo
res/drawable/drawable-en/ (NO! sbagliato)
- I valori sono case-insensitive (ma meglio seguire lo stile usato di default da Android)
- Si può usare un solo valore per ogni qualificatore
 - Es. Non si può fare: *drawable-es-fr/* ma occorre creare *drawable-es/* e *drawable-fr/*
- Android utilizza in automatico le giuste risorse in base al device!

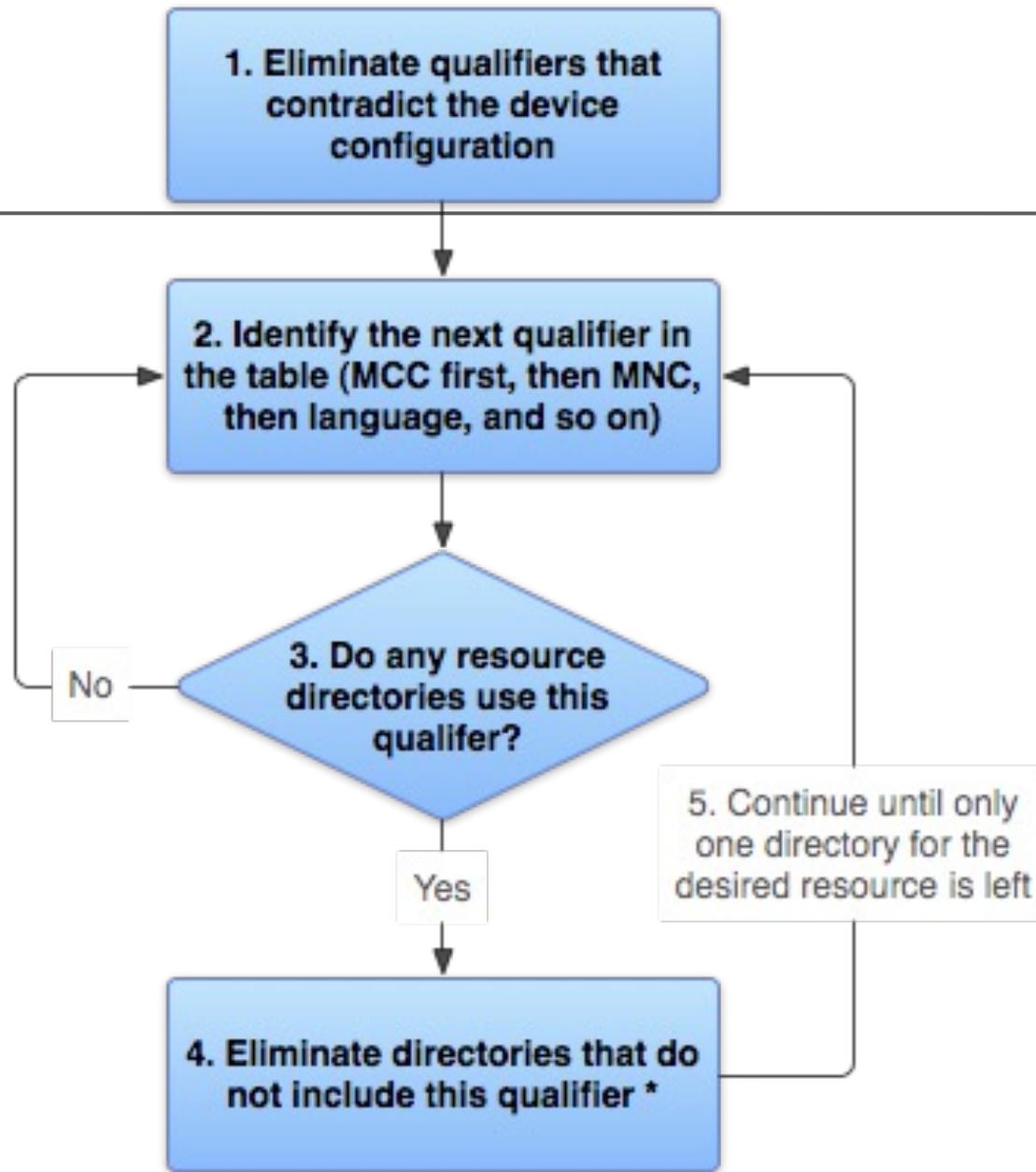
Regole

- Si possono usare diversi quantificatori insieme, separati da -
- **Occorre seguire l'ordine definito in Tabella 2**
(<https://developer.android.com/guide/topics/resources/providing-resources>)
 - Quindi: *drawable-hdpi-port/ NO!!!* sbagliato
 - Mentre: *drawable-port-hdpi/ OK*
- Non si possono creare cartelle indentate del tipo
res/drawable/drawable-en/ (NO! sbagliato)
- I valori sono case-insensitive (ma meglio seguire lo stile usato di default da Android)
- Si può usare un solo valore per ogni qualificatore
 - Es. Non si può fare: *drawable-es-fr/* ma occorre creare *drawable-es/* e *drawable-fr/*
- Android utilizza in automatico le giuste risorse in base al device!

Importante: scelta configurazione

- Per far sì che la tua app supporti diversi device, è importante fornire sempre le risorse di **default**
- Questo perché se non lo fate, e Android non riesce a trovare il match esatto per il device specifico, l'app fa crash
- Per esempio, se volete fornire più lingue, fornite sempre la directory `values/` in cui salvare le stringhe
- Altro esempio, se vuoi fornire risorse di layout diverse in base all'orientamento dello schermo, dovresti scegliere un orientamento come default. Ovvero, invece di fornire risorse di layout in `layout-land/` for landscape e `layout-port/` per portrait, lasciane uno come predefinito, cioè `layout/` for landscape e crea `layout-port/` per portrait

Scelta configurazione



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

88

89

90

91

92

93

94

95

96

97

98

99

100

101

102

103

104

105

106

107

108

109

110

111

112

113

114

115

116

117

118

119

120

121

122

123

124

125

126

127

128

129

130

131

132

133

134

135

136

137

138

139

140

141

142

143

144

145

146

147

148

149

150

151

152

153

154

155

156

157

158

159

160

161

162

163

164

165

166

167

168

169

170

171

172

173

174

175

176

177

178

179

180

181

182

183

184

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

205

206

207

208

209

210

211

212

213

214

215

216

217

218

219

220

221

222

223

224

225

226

227

228

229

230

231

232

233

234

235

236

237

238

239

240

241

242

243

244

245

246

247

248

249

250

251

252

253

254

255

256

257

258

259

260

261

262

263

264

265

266

267

268

269

270

271

272

273

274

275

276

277

278

279

280

281

282

283

284

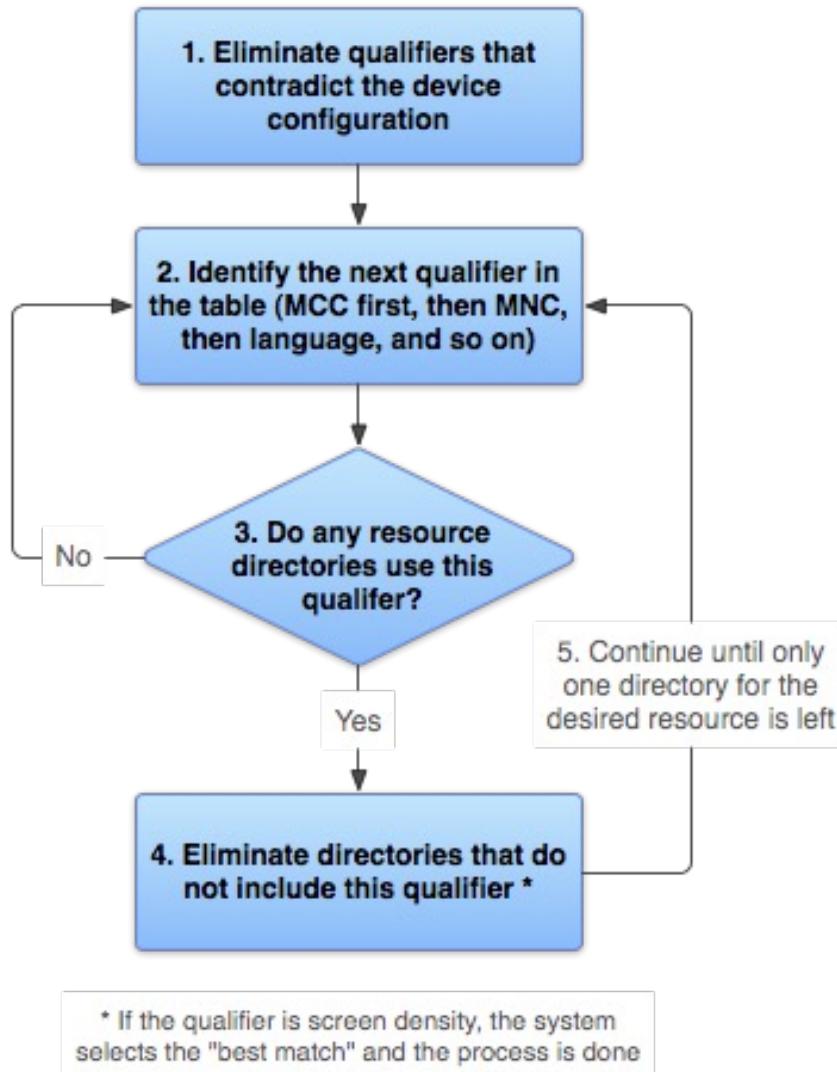
285

286

287

288

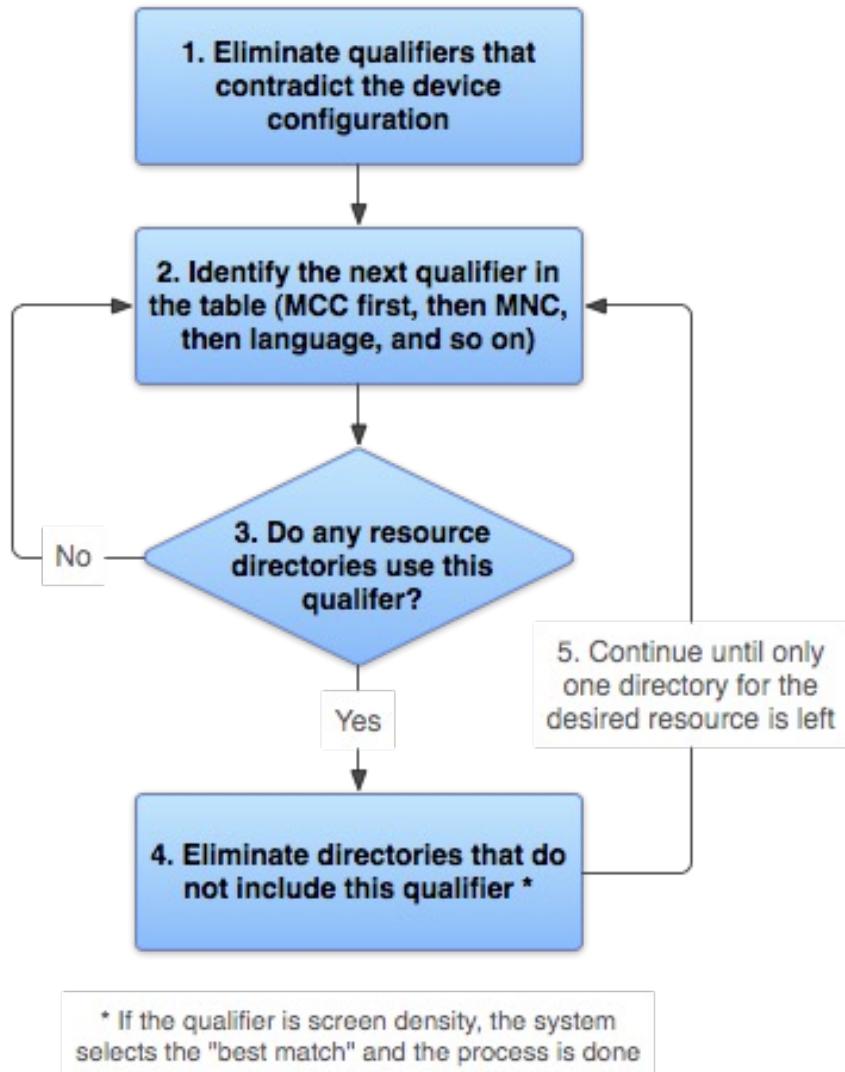
Scelta configurazione



Locale = en-GB
Screen orientation = port
Screen pixel density = hdpi
Touchscreen type = notouch
Primary text input method = 12key

drawable/
drawable-en/
~~drawable-fr-rCA/~~
drawable-en-port/
drawable-en-notouch-12key/
drawable-port-hdpi/
drawable-port-notouch-12key/

Scelta configurazione



Locale = en-GB

Screen orientation = port

Screen pixel density = hdpi

Touchscreen type = notouch

Primary text input method = 12key

drawable/

drawable-en/

drawable-fr rCA/

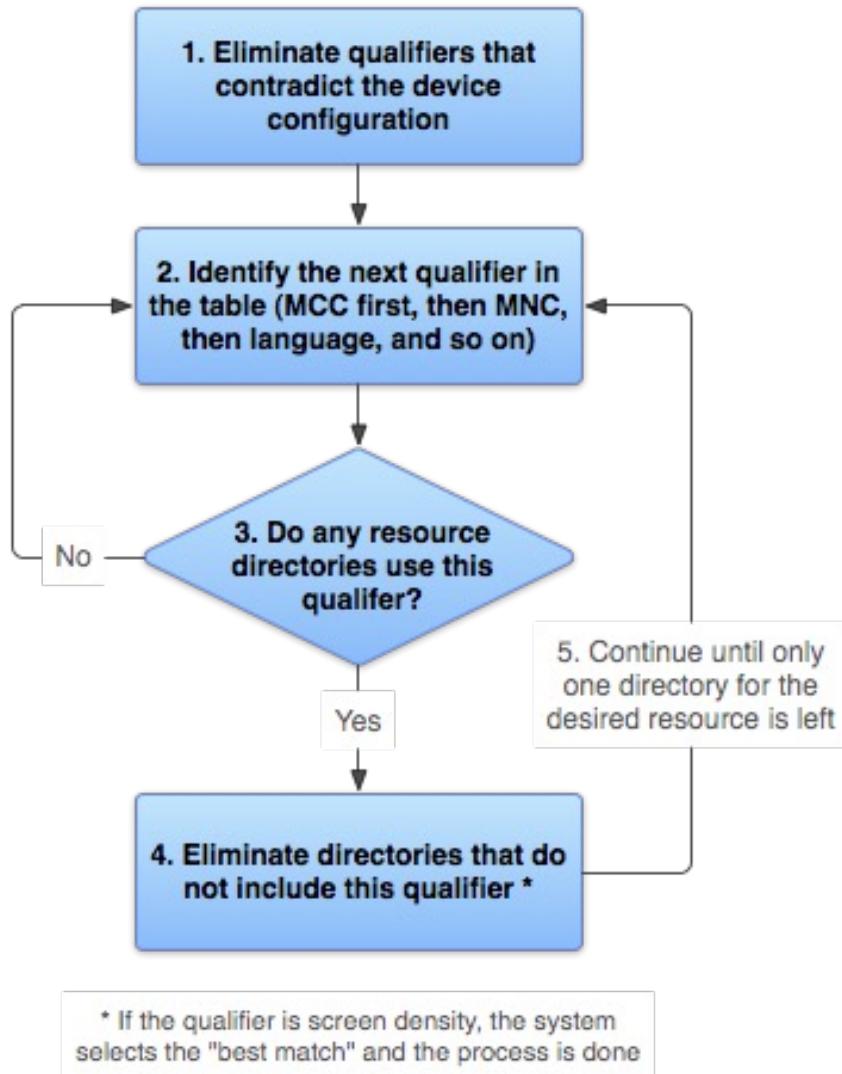
drawable-en-port/

drawable-en-notouch-12key/

drawable-port ldpi/

drawable-port notouch 12key/

Scelta configurazione



Locale = en-GB

Screen orientation = port

Screen pixel density = hdpi

Touchscreen type = notouch

Primary text input method = 12key

~~drawable/~~

~~drawable-en/~~

~~drawable-fr rCA/~~

drawable-en-port/

~~drawable-en notouch 12key/~~

~~drawable-port ldpi/~~

~~drawable-port notouch 12key/~~

Altra nota

- Le configurazioni possono cambiare anche a run-time
 - Pensate all'orientamento del device, o al Night mode
- Per questo Android fa il restart dell'Activity che sta eseguendo (`onDestroy()` viene chiamato, seguito da `onCreate()`) e carica la nuova configuazione
- Bisogna quindi far attenzione a riattivare lo stato precedente dell'app
 - Più info qui:
<https://developer.android.com/guide/topics/resources/runtime-changes>

Altra nota

- Le configurazioni possono cambiare anche a run-time
 - Pensate all'orientamento del device, o al Night mode
- Per questo Android fa il reset delle risorse:
`(onDestroy()) viene chiamato prima della nuova configuazione`
- Bisogna quindi ricaricare le risorse dell'app
 - Più info qui:



Lo vedremo nel
dettaglio!

<https://developer.android.com/guide/topics/resources/runtime-changes>

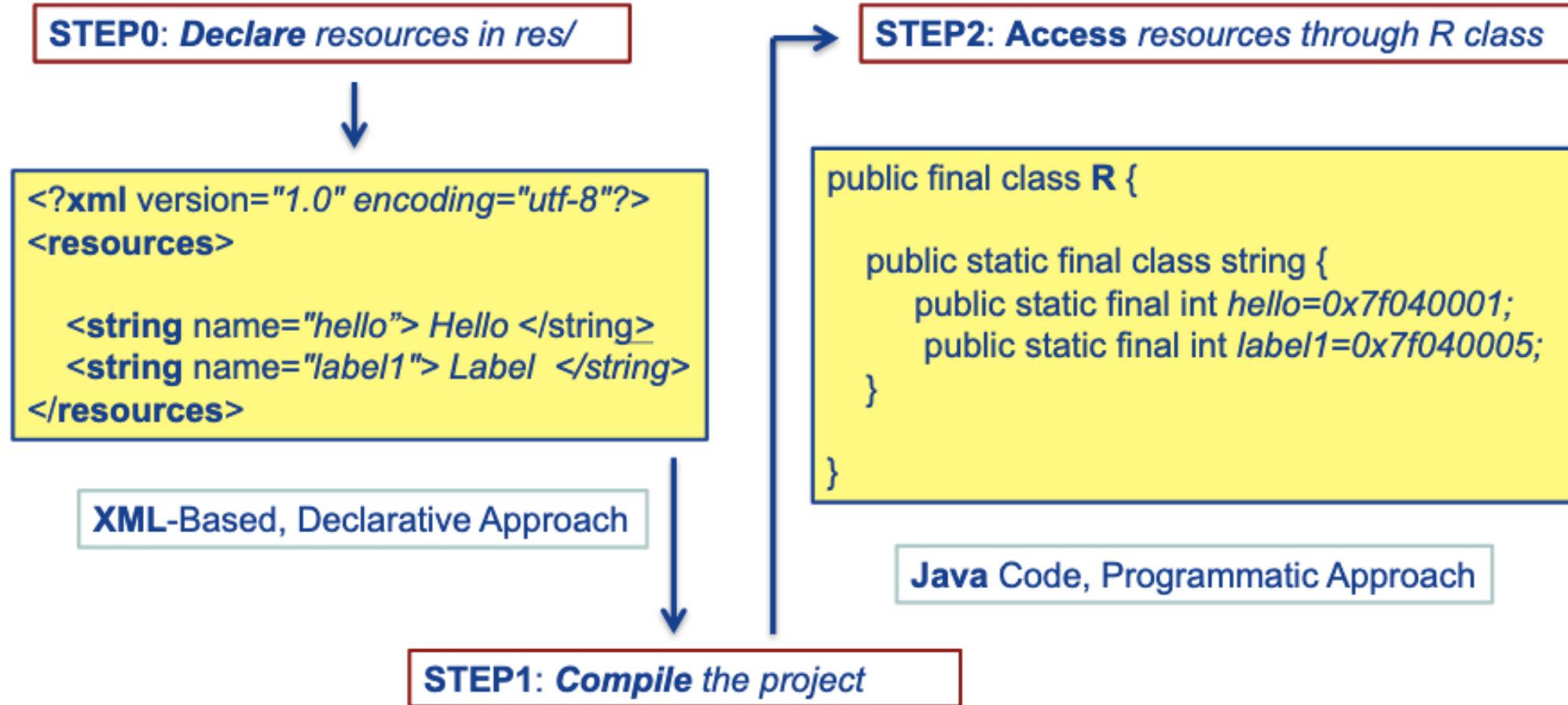
Accedere alle risorse

- Si possono accedere tramite la **classe R** che viene automaticamente generata durante la compilazione e contiene i resourceId per tutte le risorse nella cartella res/
- Per ogni tipo di risorsa c'è una sottoclasse di **R**
 - Es. R.string
- E per ogni risorsa di quel tipo, c'è un static integer
 - ES. R.string.hello
- Quel integer è il resourceId che si può usare per ottenere la risorsa

ID risorsa

- L'ID della risorsa deve essere univoco ed è composto da:
 - Tipo della risorsa: come string, drawable, etc...
 - <string name="hello">Jetpack Compose</string>
 - Nome della risorsa
 - <string name ="**hello**"> Jetpack Compose</string>

Per visualizzarlo meglio



@Credit: Luca Bedogni

Nel codice

[<package_name>.]R.<resource_type>.<resource_name>

- Dove
 - <package_name> è il nome del package in cui la risorsa è localizzata (non è richiesto se ci si riferisce a risorse nello stesso package)
 - <resource_type> è il nome del tipo di risorsa
 - <resource_name> è il nome del file (senza estensione) oppure il valore che si trova nell’attributo android:name dell’elemento XML

Risorse e Compose

- Vediamo alcuni esempi con:
 - Strings
 - Dimensions
 - Colors
 - Drawable
 - Fonts
 - Text
 - Ecc..

Strings

- Il tipo di risorsa più comune sono le stringe

```
// In the res/values/strings.xml file  
// <string name="compose">Jetpack Compose</string>  
  
// In your Compose code  
Text(  
    text = stringResource(R.string.compose)  
)
```

[ResourcesSnippets.kt](#) 

`stringResource` also works with positional formatting.

```
// In the res/values/strings.xml file  
// <string name="congratulate">Happy %1$s %2$d</string>  
  
// In your Compose code  
Text(  
    text = stringResource(R.string.congratulate, "New Year", 2021)  
)
```

[ResourcesSnippets.kt](#) 

Dimensions

```
// In the res/values/dimens.xml file
// <dimen name="padding_small">8dp</dimen>

// In your Compose code
val smallPadding = dimensionResource(R.dimen.padding_small)
Text(
    text = "...",
    modifier = Modifier.padding(smallPadding)
)
```

[ResourcesSnippets.kt](#) 

Colors

```
// In the res/colors.xml file  
// <color name="purple_200">#FFBB86FC</color>  
  
// In your Compose code  
Divider(color = colorResource(R.color.purple_200))
```

ResourcesSnippets.kt 

Attenzione!! E' meglio preferire i colori del tema piuttosto che i colori codificati. Anche se è possibile accedere ai colori utilizzando la funzione colorResource, è consigliabile che i colori della tua app siano definiti in un MaterialTheme a cui è possibile accedere dai tuoi componenti composable come MaterialTheme.colors.primary. Scopri di più su questo argomento nella documentazione relativa alla progettazione di sistemi nella sezione Compose (<https://developer.android.com/jetpack/compose/designsystems>).

Drawable

```
// Files in res/drawable folders. For example:  
// - res/drawable-nodpi/ic_logo.xml  
// - res/drawable-xxhdpi/ic_logo.png  
  
// In your Compose code  
Icon(  
    painter = painterResource(id = R.drawable.ic_logo),  
    contentDescription = null // decorative element  
)
```

[ResourcesSnippets.kt](#) 

Fonts



```
// Define and load the fonts of the app
private val light = Font(R.font.raleway_light, FontWeight.W300)
private val regular = Font(R.font.raleway_regular, FontWeight.W400)
private val medium = Font(R.font.raleway_medium, FontWeight.W500)
private val semibold = Font(R.font.raleway_semiBold, FontWeight.W600)

// Create a font family to use in TextStyles
private val craneFontFamily = FontFamily(light, regular, medium, semibold)

// Use the font family to define a custom typography
val craneTypography = Typography(
    titleLarge = TextStyle(
        fontFamily = craneFontFamily
    ) /* ... */
)

// Pass the typography to a MaterialTheme that will create a theme using
// that typography in the part of the UI hierarchy where this theme is used
@Composable
fun CraneTheme(content: @Composable () -> Unit) {
    MaterialTheme(typography = craneTypography) {
        content()
    }
}
```

ResourcesSnippets.kt

Text

- Il testo è un elemento centrale di qualsiasi UI e Jetpack Compose rende più semplice la visualizzazione e la scrittura del testo.
- Compose fornisce **Text** e **TextField**, ovvero composable che seguono le linee guida di Material Design.
- Si consiglia di utilizzarli, poiché hanno il look and feel giusti per gli utenti di Android e includono altre opzioni per semplificare la loro personalizzazione senza dover scrivere molto codice.

Esempi

```
@Composable  
fun StringResourceText() {  
    Text(stringResource(R.string.hello_world))  
}
```

Hello World

```
@Composable  
fun BlueText() {  
    Text("Hello World", color = Color.Blue)  
}
```

Hello World

```
@Composable  
fun BigText() {  
    Text("Hello World", fontSize = 30.sp)  
}
```

Hello World

Esempi

```
@Composable
fun CenterText() {
    Text(
        "Hello World", textAlign = TextAlign.Center, modifier = Modifier.width(150.dp)
    )
}
```



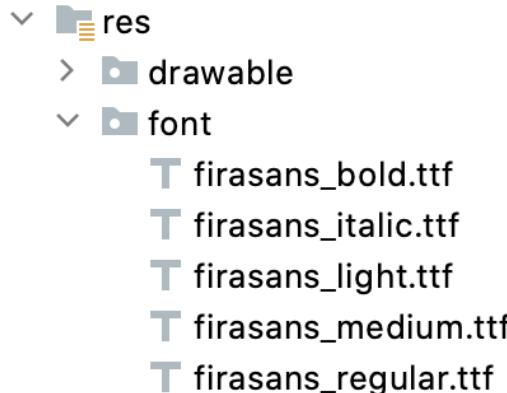
Hello World

```
@Composable
fun TextShadow() {
    val offset = Offset(5.0f, 10.0f)
    Text(
        text = "Hello world!",
        style = TextStyle(
            fontSize = 24.sp,
            shadow = Shadow(
                color = Color.Blue, offset = offset, blurRadius = 3f
            )
        )
    )
}
```



Hello world!

Esempi



```
val firaSansFamily = FontFamily(  
    Font(R.font.firasans_light, FontWeight.Light),  
    Font(R.font.firasans_regular, FontWeight.Normal),  
    Font(R.font.firasans_italic, FontWeight.Normal, FontStyle.Italic),  
    Font(R.font.firasans_medium, FontWeight.Medium),  
    Font(R.font.firasans_bold, FontWeight.Bold)  
)
```

```
Column {  
    Text(text = "text", fontFamily = firaSansFamily, fontWeight = FontWeight.Light)  
    Text(text = "text", fontFamily = firaSansFamily, fontWeight = FontWeight.Normal)  
    Text(  
        text = "text",  
        fontFamily = firaSansFamily,  
        fontWeight = FontWeight.Normal,  
        fontStyle = FontStyle.Italic  
    )  
    Text(text = "text", fontFamily = firaSansFamily, fontWeight = FontWeight.Medium)  
    Text(text = "text", fontFamily = firaSansFamily, fontWeight = FontWeight.Bold)  
}
```

Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World

Esempi

```
val gradientColors = listOf(Cyan, LightBlue, Purple /*...*/)
```

```
Text(  
    text = text,  
    style = TextStyle(  
        brush = Brush.linearGradient(  
            colors = gradientColors  
    )  
)  
)
```

RISULTATO???

Esempi

```
val gradientColors = listOf(Cyan, LightBlue, Purple /*...*/)

Text(
    text = text,
    style = TextStyle(
        brush = Brush.linearGradient(
            colors = gradientColors
        )
    )
)
```

RISULTATO???

Do not allow people to
dim your shine
because they are
blinded. Tell them to
put some sunglasses
on.

Esempi

```
var text by remember { mutableStateOf("") }
val brush = remember {
    Brush.linearGradient(
        colors = rainbowColors
    )
}
TextField(
    value = text, onValueChange = { text = it }, textStyle = TextStyle(brush = brush)
)
```

Nota: la funzione remember permette di mantenere il brush per tutte le ricomposizioni, ovvero quando lo stato della casella di testo cambia per ogni nuovo carattere digitato.



Esempi

```
val brush = Brush.linearGradient(colors = rainbowColors)

buildAnnotatedString {
    withStyle(
        SpanStyle(
            brush = brush, alpha = .5f
        )
    ) {
        append("Text in ")
    }
    withStyle(
        SpanStyle(
            brush = brush, alpha = 1f
        )
    ) {
        append("Compose ❤")
    }
}
```

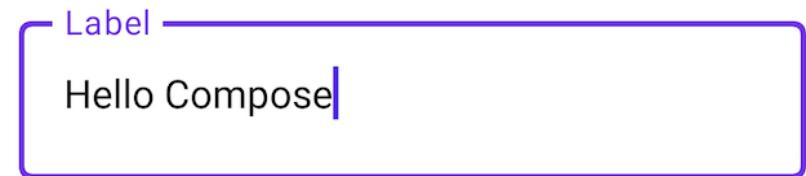


Text in Compose ❤

Esempi - textField

```
@Composable
fun SimpleOutlinedTextFieldSample() {
    var text by remember { mutableStateOf("") }

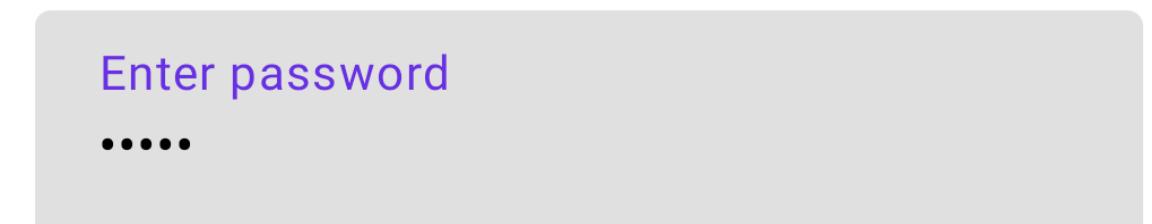
    OutlinedTextField(
        value = text,
        onValueChange = { text = it },
        label = { Text("Label") }
    )
}
```



Esempi - textField

```
@Composable
fun PasswordTextField() {
    var password by rememberSaveable { mutableStateOf("") }

    TextField(
        value = password,
        onValueChange = { password = it },
        label = { Text("Enter password") },
        visualTransformation = PasswordVisualTransformation(),
        keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Password)
    )
}
```



Immagini

- Esiste il composable **Image**.
- Per caricare un'immagine (ad esempio PNG, JPEG, WEBP) o una risorsa vettoriale dal disco, utilizzare l'API **painterResource** con il riferimento all'immagine.

```
Image(  
    painter = painterResource(id = R.drawable.dog),  
    contentDescription = stringResource(id = R.string.dog_content_description)  
)
```

```
AsyncImage(  
    model = "https://example.com/image.jpg",  
    contentDescription = "Translated description of what the image contains"  
)
```



Icone Material

- Icon è un modo comodo per disegnare sullo schermo un'icona di un solo colore che segue le linee guida del Material Design.

```
Icon(  
    painter = painterResource(R.drawable.baseline_directions_bus_24),  
    contentDescription = stringResource(id = R.string.bus_content_description)  
)
```

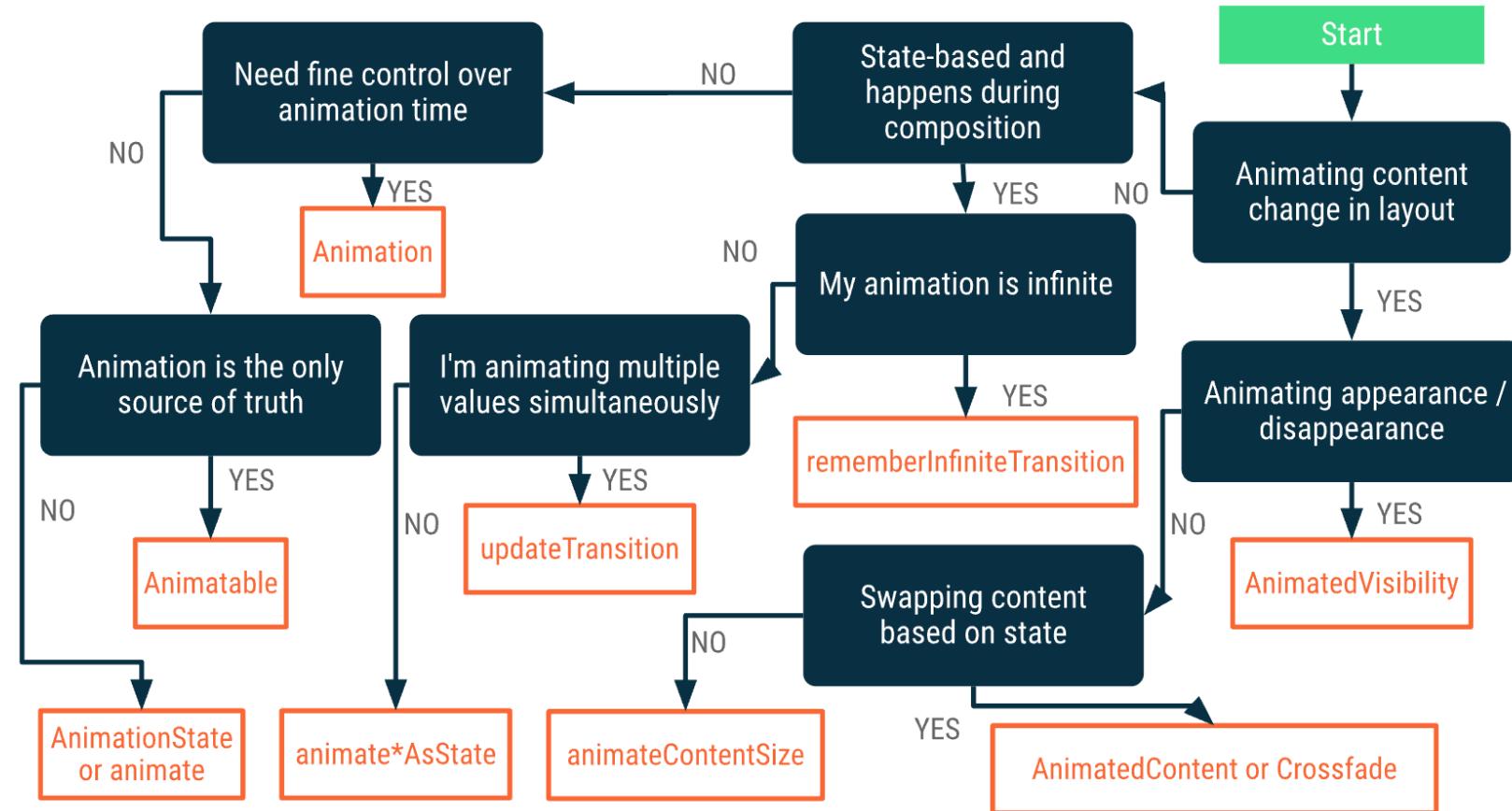
```
Icon(  
    Icons.Rounded.ShoppingCart,  
    contentDescription = stringResource(id = R.string.shopping_cart_content_desc)  
)
```



La libreria Material Icon include anche una serie di Icons predefinite che possono essere utilizzate in Compose senza dover importare manualmente un SVG

Animations

- Jetpack Compose fornisce API che rendono semplice l'implementazione di varie animazioni.
- Molte API di Jetpack Compose Animation sono disponibili come funzioni composable.



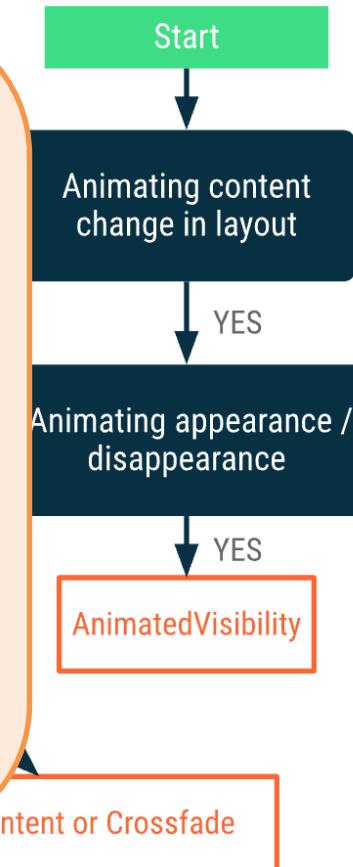
Animations

- Jetpack Compose fornisce API che rendono l'implementazione di animazioni più semplice
- Molte API di Jetpack Compose sono disponibili per la composizione di animazioni

Non riusciremo a vederle in dettaglio

QUINDI

Può essere un argomento di approfondimento



Gestures

- Tapping and pressing
- Scrolling
- Dragging
- Swiping
- Multitouch

Tapping and pressing

- Il modificatore **clickable** consente alle applicazioni di rilevare i clic sull'elemento a cui è applicato.

```
@Composable
fun ClickableSample() {
    val count = remember { mutableStateOf(0) }
    // content that you want to make clickable
    Text(
        text = count.value.toString(),
        modifier = Modifier.clickable { count.value += 1 }
    )
}
```

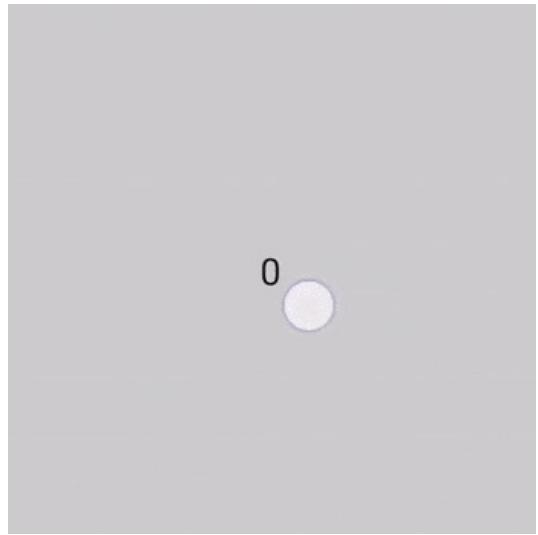
Cosa permette di fare???

Tapping and pressing

- Il modificatore **clickable** consente alle applicazioni di rilevare i clic sull'elemento a cui è applicato.

```
@Composable
fun ClickableSample() {
    val count = remember { mutableStateOf(0) }
    // content that you want to make clickable
    Text(
        text = count.value.toString(),
        modifier = Modifier.clickable { count.value += 1 }
    )
}
```

Cosa permette di fare???



Tapping and pressing

- È possibile definire un'azione anche a seconda del tipo di pressione

```
Modifier.pointerInput(Unit) {
    detectTapGestures(
        onPress = { /* Called when the gesture starts */ },
        onDoubleTap = { /* Called on Double Tap */ },
        onLongPress = { /* Called on Long Press */ },
        onTap = { /* Called on Tap */ }
    )
}
```

Scrolling

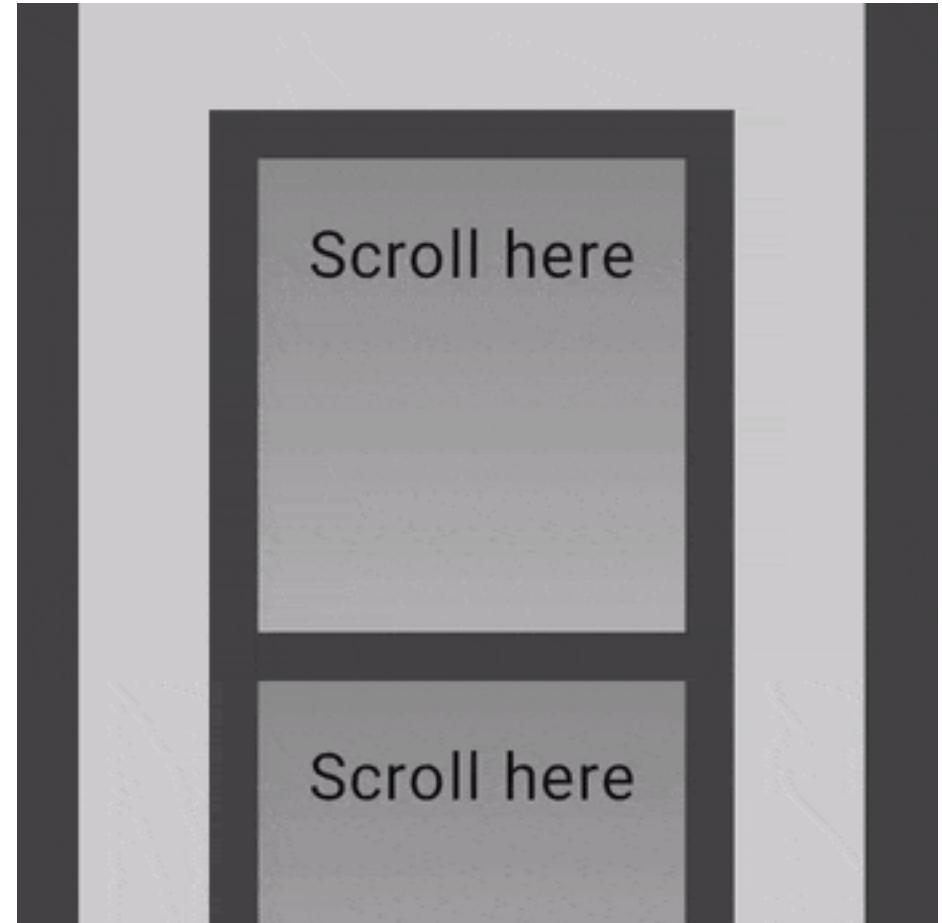
- Come detto prima, i modificatori **verticalScroll** e **horizontalScroll** forniscono il modo più semplice per consentire all'utente di scorrere un elemento.

```
@Composable
fun ScrollBoxes() {
    Column(
        modifier = Modifier
            .background(Color.LightGray)
            .size(100.dp)
            .verticalScroll(rememberScrollState())
    ) {
        repeat(10) {
            Text("Item $it", modifier = Modifier.padding(2.dp))
        }
    }
}
```



Nested scrolling

```
val gradient = Brush.verticalGradient(0f to Color.Gray, 1000f to Color.White)
Box(
    modifier = Modifier
        .background(Color.LightGray)
        .verticalScroll(rememberScrollState())
        .padding(32.dp)
) {
    Column {
        repeat(6) {
            Box(
                modifier = Modifier
                    .height(128.dp)
                    .verticalScroll(rememberScrollState())
            ) {
                Text(
                    "Scroll here",
                    modifier = Modifier
                        .border(12.dp, Color.DarkGray)
                        .background(brush = gradient)
                        .padding(24.dp)
                        .height(150.dp)
                )
            }
        }
    }
}
```



Dragging

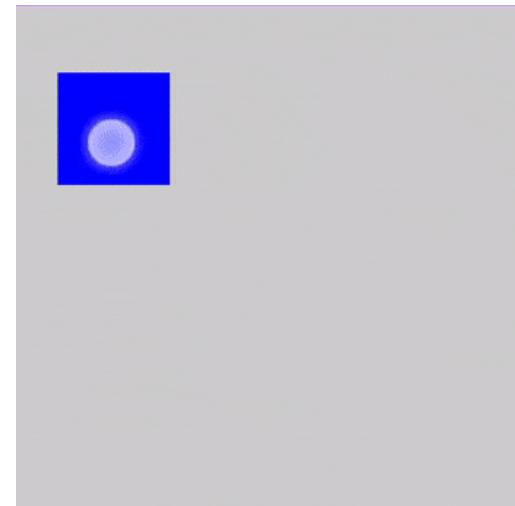
- Il modificatore **draggable** è il punto di ingresso per i gesti di trascinamento in un singolo orientamento e riporta la distanza di trascinamento in pixel.
- Questo modificatore è simile a scrollable, in quanto rileva solo il gesto. È necessario mantenere lo stato e rappresentarlo sullo schermo, ad esempio spostando l'elemento tramite il modificatore offset

```
var offsetX by remember { mutableStateOf(0f) }
Text(
    modifier = Modifier
        .offset { IntOffset(offsetX.toInt(), 0) }
        .draggable(
            orientation = Orientation.Horizontal,
            state = rememberDraggableState { delta ->
                offsetX += delta
            }
        ),
    text = "Drag me!"
)
```

Dragging

- Se è necessario controllare l'intero gesto di trascinamento, si può utilizzare il rilevatore del gesto di trascinamento, tramite il modificatore **pointerInput**.

```
Box(modifier = Modifier.fillMaxSize()) {  
    var offsetX by remember { mutableStateOf(0f) }  
    var offsetY by remember { mutableStateOf(0f) }  
  
    Box(  
        Modifier  
            .offset { IntOffset(offsetX.toInt(), offsetY.toInt()) }  
            .background(Color.Blue)  
            .size(50.dp)  
            .pointerInput(Unit) {  
                detectDragGestures { change, dragAmount ->  
                    change.consumeAllChanges()  
                    offsetX += dragAmount.x  
                    offsetY += dragAmount.y  
                }  
            }  
    )  
}
```



Swiping

- Il modificatore **swipeable** consente di trascinare elementi che, una volta rilasciati, si animano verso due o più punti di ancoraggio definiti in un orientamento.
- Un uso comune di questo modificatore è quello di implementare un modello di "**swipe-to-dismiss**".
- Questo modificatore non sposta l'elemento, ma rileva solo il gesto.

```
@Composable
fun SwipeableSample() {
    val width = 96.dp
    val squareSize = 48.dp

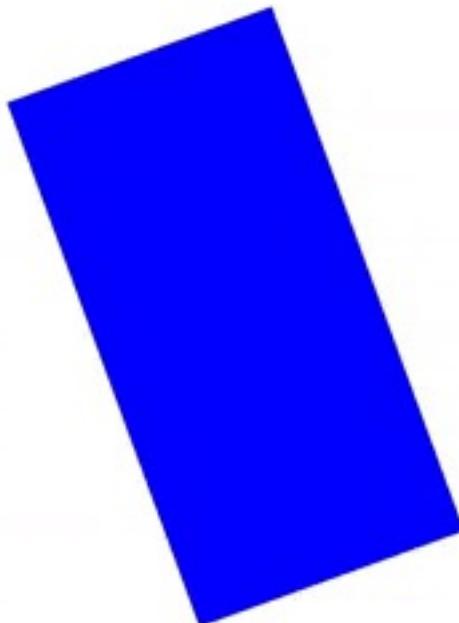
    val swipeableState = rememberSwipeableState(0)
    val sizePx = with(LocalDensity.current) { squareSize.toPx() }
    val anchors = mapOf(0f to 0, sizePx to 1) // Maps anchor points (in px) to states

    Box(
        modifier = Modifier
            .width(width)
            .swipeable(
                state = swipeableState,
                anchors = anchors,
                thresholds = { _, _ -> FractionalThreshold(0.3f) },
                orientation = Orientation.Horizontal
            )
            .background(Color.LightGray)
    ) {
        Box(
            Modifier
                .offset { IntOffset(swipeableState.offset.value.toInt(), 0) }
                .size(squareSize)
                .background(Color.DarkGray)
        )
    }
}
```

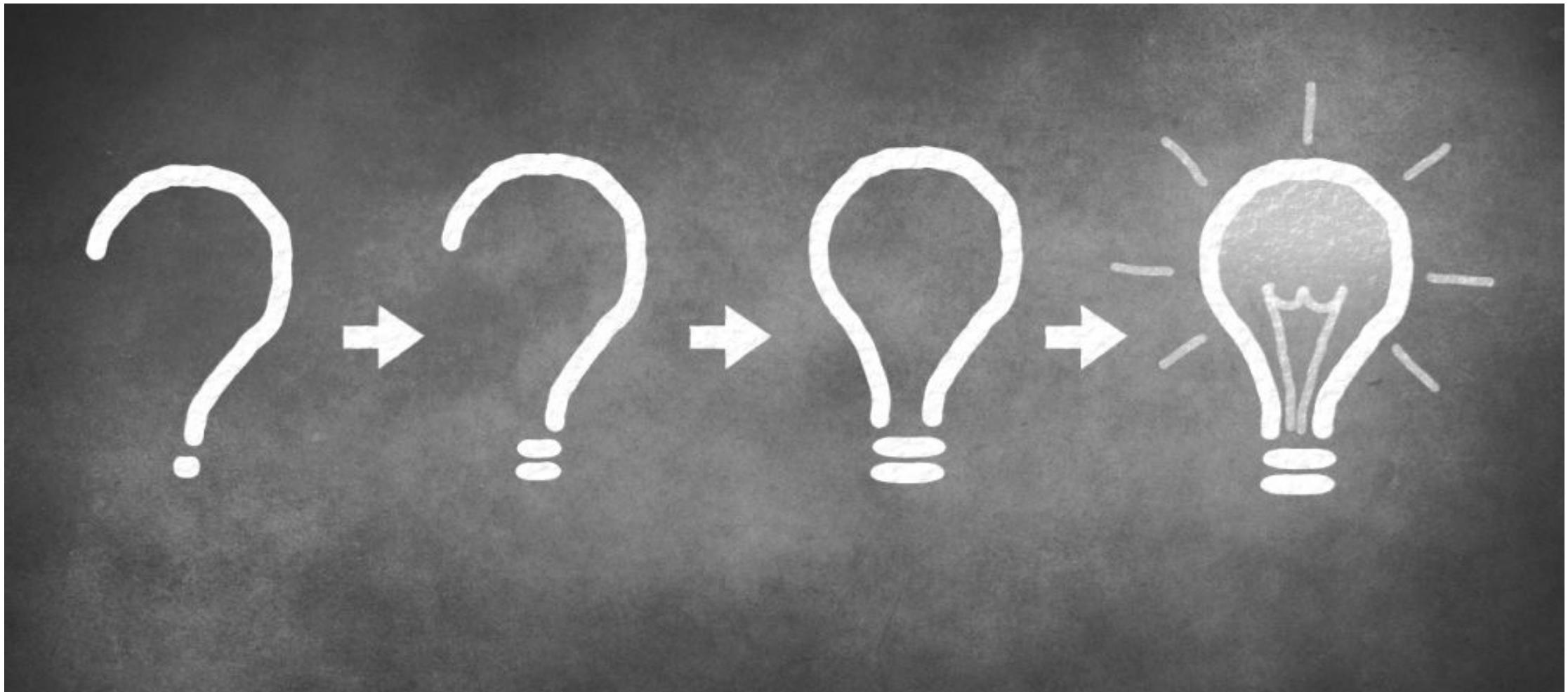


Multitouch: Panning, zooming, rotating

```
@Composable
fun TransformableSample() {
    // set up all transformation states
    var scale by remember { mutableStateOf(1f) }
    var rotation by remember { mutableStateOf(0f) }
    var offset by remember { mutableStateOf(Offset.Zero) }
    val state = rememberTransformableState { zoomChange, offsetChange, rotationChange ->
        scale *= zoomChange
        rotation += rotationChange
        offset += offsetChange
    }
    Box(
        Modifier
            // apply other transformations like rotation and zoom
            // on the pizza slice emoji
            .graphicsLayer(
                scaleX = scale,
                scaleY = scale,
                rotationZ = rotation,
                translationX = offset.x,
                translationY = offset.y
            )
            // add transformable to listen to multitouch transformation events
            // after offset
            .transformable(state = state)
            .background(Color.Blue)
            .fillMaxSize()
    )
}
```



Domande?



Riferimenti

- Documentazione Android Developers
 - <https://developer.android.com/topic/architecture>
 - <https://developer.android.com/topic/architecture/ui-layer>
 - <https://developer.android.com/courses/jetpack-compose/course>
 - <https://developer.android.com/jetpack/compose/documentation>
 - <https://developer.android.com/jetpack/compose/tutorial>
 - <https://developer.android.com/jetpack/compose/layouts/basics>
 - <https://developer.android.com/guide/topics/resources/providing-resources>