# 5. JETBACK COMPOSE

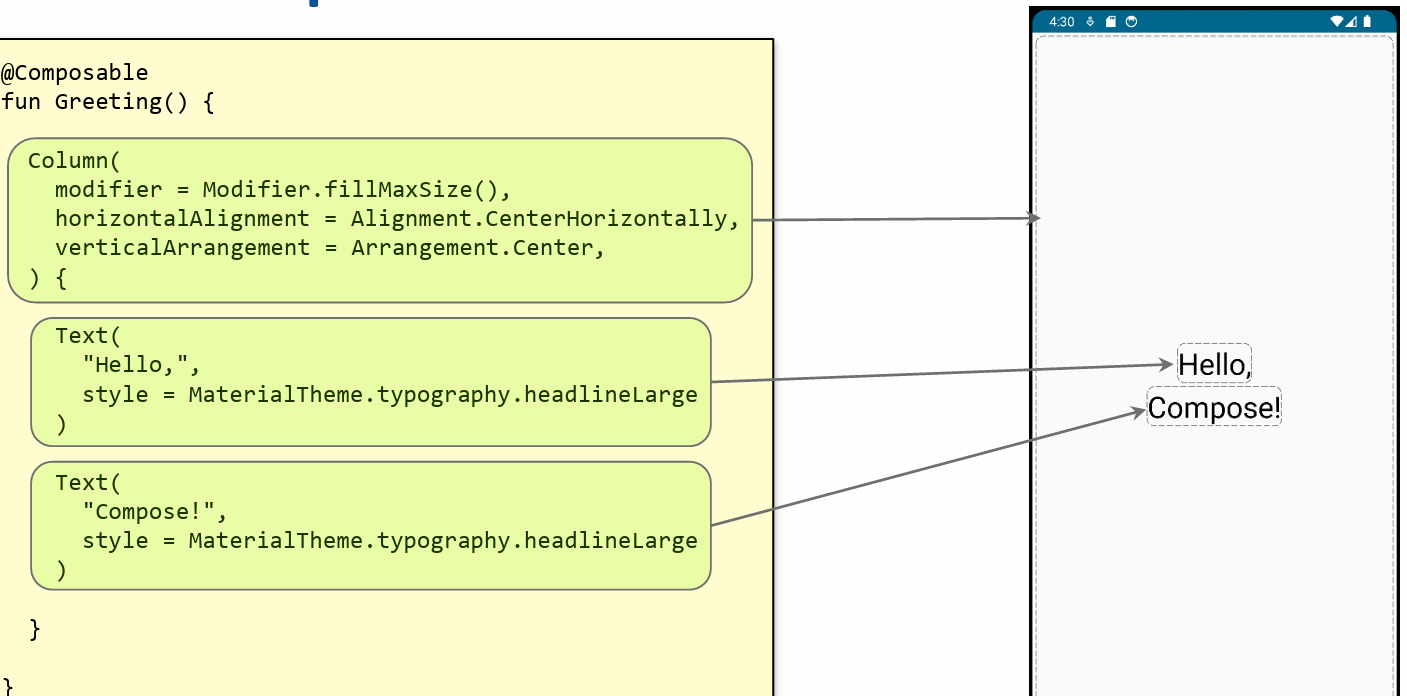
Approccio funzionale, funzioni etichettate con annotazione **@Composable**. Notazione è gestita in compilazione.

**NOTA**: una funzione composable può invocare una funzione plain mentre una funzione plain, non può invocare una funzione composable. C’è solo un’eccezione → setContent: accetta come parametro una lambda composable.

Ogni volta che una funzione composable è eseguita, Andorid traccia i parametri forniti alla funzione.

**Puramente dichiarativo**: funzione di basso livello producono codice che viene eseguito dalla cpu per produrre delle rappresentazioni. Se android si accorge che sono cambiati dei parametri → android riesegue la funzione per la GPU → se cambiano i dati, cambia cosa vedo su schermo 🡪 **data driven**.

Android esegue funzioni sofisticate, in modo parallelo → dunque per evitare incongruenze, le funzioni composable **non possono avere “side effect”**, come ad esempio modificare variabili globali, etc… loro possono solo generare del codice.

Le funzioni composable, solitamente, **non ritornano nulla** ma emettono dei blocchi internamente **chiamando altre funzioni composable** primitive fornite da Android o non primitive create → queste funzioni consistono in istruzioni per la GPU.

Scritte in Uppercase →**Column**

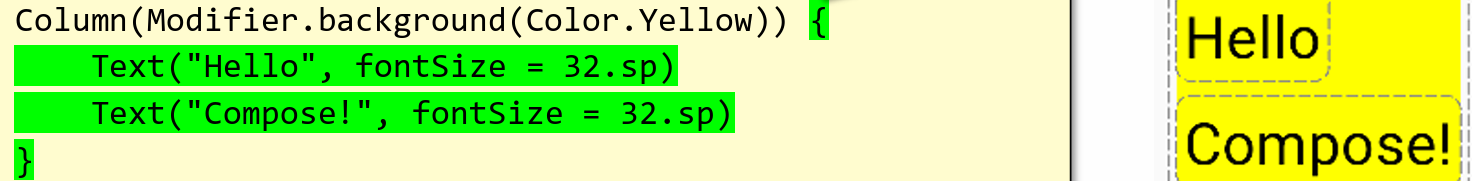
* Modifier
* horizontalAlignment
* verticalArrangment
* trailing lambda
  + contiene 2 buildComposable Text

**NOTA su column**: mette uno sotto l’altro i blocchi che contiene nella lambda, rispettando indicazioni date.  
**NOTA su row**: mette uno a fianco all’altro  
**NOTA su text**: trasforma una stringa in un insieme di comandi grafici

* stringa
* style → mette insieme le informazioni

Compose DSL

Quando è presente una trailing lambda, questa crea una composizione gerarchica: column è padre dei 2 Text

Non avendo specificato posizione e dimensione, il blocco di column sarà grande il minimo che serve per contenere i due text e i due text non avranno un allineamento particolare.

**NOTA**: Una lambda può avere un receiver → elemento this che fornisce alcune funzionalità aggiuntive.

### **TEXT** → mostrare uno o più linee di testo

* **Immagine che contiene testo, schermata, Carattere

  Descrizione generata automaticamentetext**: obbligatorio, indica la stringa che vogliamo mostrare (il resto sono opzionali)
* **fontsize**: dimensione
* **modifier**: azioni o decorazioni che vogliamo
  + **size**: area del riquadro
  + **background**: colore sfondo
  + **wapContectSize**: se non c’ abbastanza spazio per la stringa, questa viene wrappata e ci sarà una rappresentazione su più righe
  + **blur**: nebbia su elemento, sfoca

### **CANVAS** → modo per rappresentare una superficie, un’area rettangolare dipingendo qualcosa

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamenteMolti metodi.

All’interno del blocco it.polito.mad, aggiungo **un file kotlin chiamato gui.kt** dicendo che è un file.

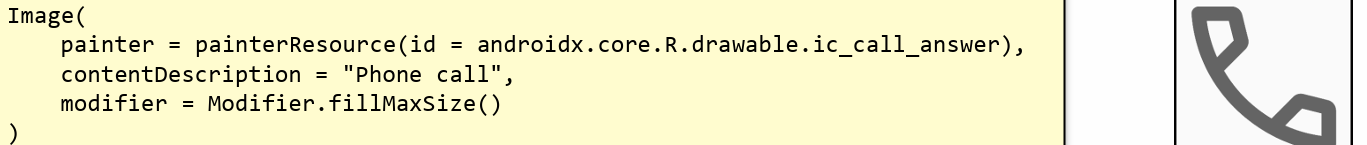
Creo funzione **@Composable fun MyGui() { }**

* **Text**
  + **style** = MaterialTheme.typography.\_\_\_\_\_ → contiene stili predefiniti
  + **textAlign** = TextAlign.\_\_\_\_\_\_\_\_ → allineamenti orrizzontali
    - allinea all’interno del suo riquadro, non in generale
  + **modifier** = Modifier
    - .background(Color.\_\_\_) → colorare sfondo blocco
    - .fillMaxWidth() → prende tutto lo spazio possibile orrizzontalmente e solo

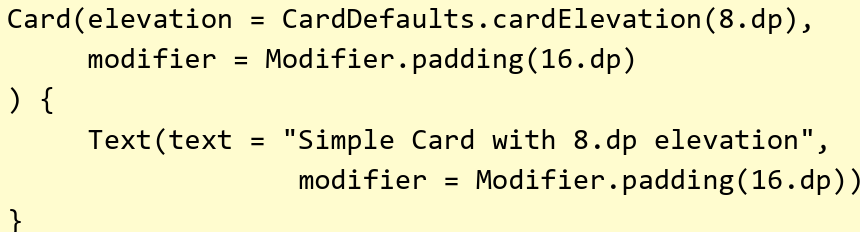
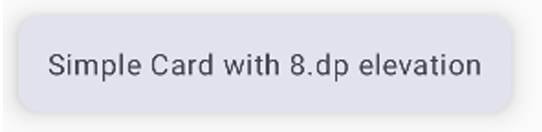
quello di cui ha bisogno verticalmente

* + - .fillMaxWeigth() → prende tutto lo spazio possibile verticalmente
    - .fillMaxSize(.8f) →prendi 80% veritcalmente e 80% orrizontalmente
* **Canvas**(modifier = Modifier.\_\_\_\_\_\_\_\_\_ ){
  + val r = minOf(size.width, size.height)\*0.4f →raggio
  + drawCircle(Color.\_\_\_\_\_\_\_, r, Offset(size.width\*0.5f, size.height\*0.5f) )

}

* **Image()** → mostrare immagini forniti da instanza Painter, bisogna aver già scaricato le immagini e devono essere disponibili

**NOTA**: esiste la libreria coil che mi permette di usare async image

* AsyncImage(
  + model=”url”
  + contentDescription =”stringa per non vedenti”
  + contentScale= ContentScale.Crop →
  + modifier = \_\_\_\_\_\_\_\_\_\_
  + placeholder = \_\_(qualcosa in res)\_\_ → usato per mostrare qualcosa mentre carica l’immagine
  + occhio che non funzionava
* **Card()** → blocco dove si possono inserire cose in colonna
  + Elevation: assottiglia angoli
  + Padding: spazio interno

**NOTA**: aggiungendo **@Preview** ho la possibilità di vedere cosa fa *(Aggiungere sopra a composable*)

**BASIC LAYOUTS:**

@Composable

fun ColumnLayout() {

**Column**(

Modifier.fillMaxSize(),

horizontalAlignment = Alignment.CenterHorizontally

verticalArrangment = Arrangment.SpaceEvenly,

){

YellowBlock()

RedBlock()

}

}

@ComposaBble

fun RowLayout(){

**Row**(

modifier = Modifier.fillMaxSize(),

horizontalArrangment = Arrangment.SpaceEvenly,

verticalAlignment = Alignment. CenterVertically,

){

YellowBlock()

RedBlock()

}

}

@Preview

@Composable

Fun MyGui() {

**BoxWithConstraints** {

if (this.maxHeight > this.maxWidth)

ColumnLayout()

else

RowLayout()

}

}

* COLUMN: gerarchico
  + Ogni figlio, uno sotto la’ltro
* RAW: gerarchio
  + Ogni figlio, alla dx del precedente
* BOX: gerarchico
  + Ogni figlio, sopra al precedente → insieme di layer

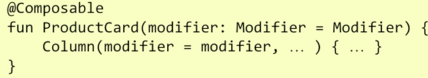
NOTA: Column e War possono avere un peso **weight**

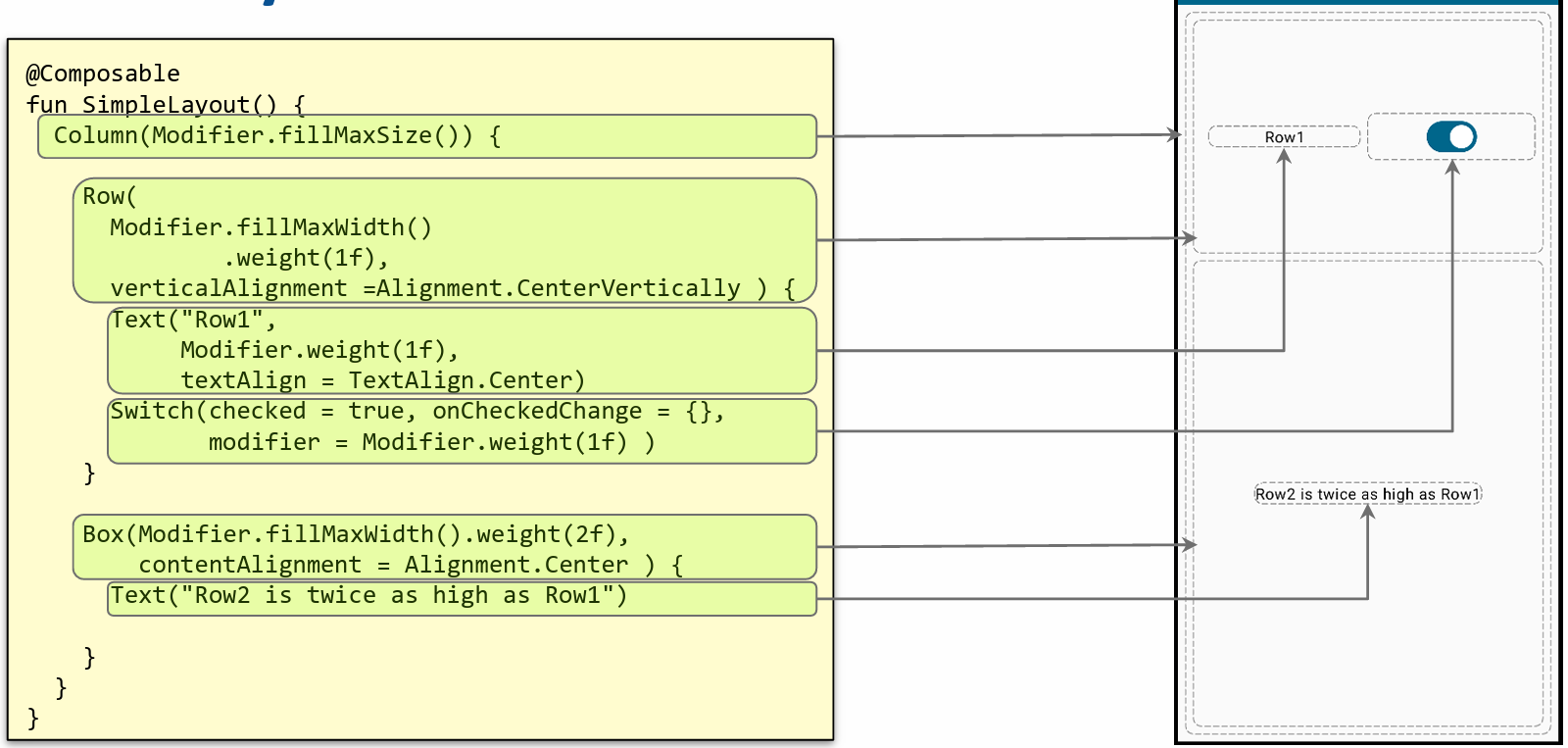
* Immagine che contiene testo, Carattere, schermata

  Descrizione generata automaticamenteImmagine che contiene testo, Carattere, schermata

  Descrizione generata automaticamentePrendono spazio proporzionale al peso

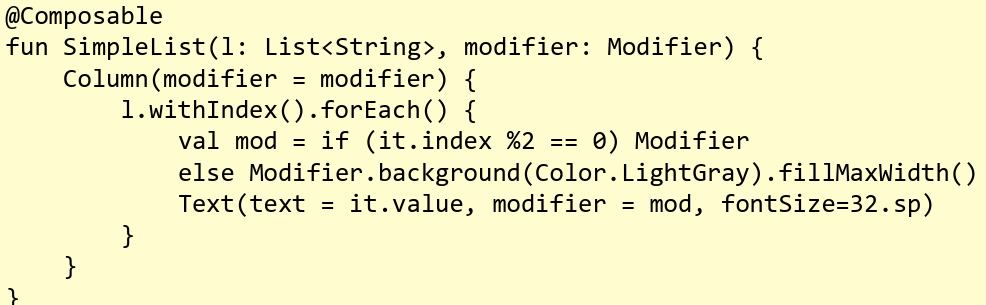
## CREATING COMPLEX LAYOUTS

Quando SI CREA UN Composable, conviene aggiungere come parametro un Modifier che è inizializzato di default con una lista Modifier vuota.



**Layout base**

**Funzioni composable**

* possono ricevere parametri, usare istruzioni condizionali, cicliche, etc
* possono chiamare altre funzioni

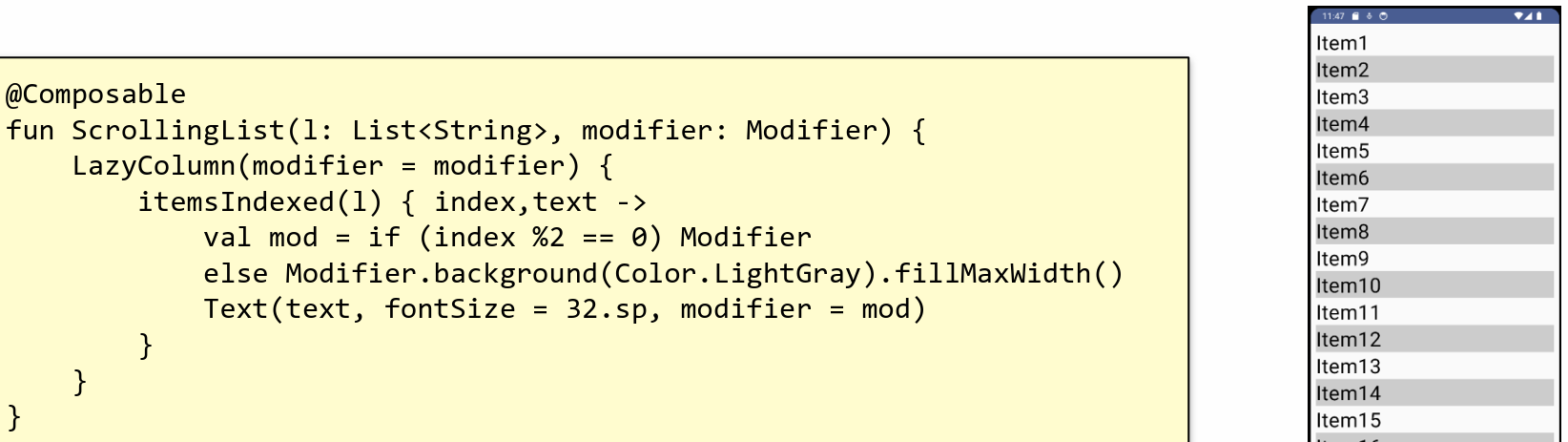
questo metodo per la lista, va bene se ho pochi elementi altrimenti no perché non ho scrolling.

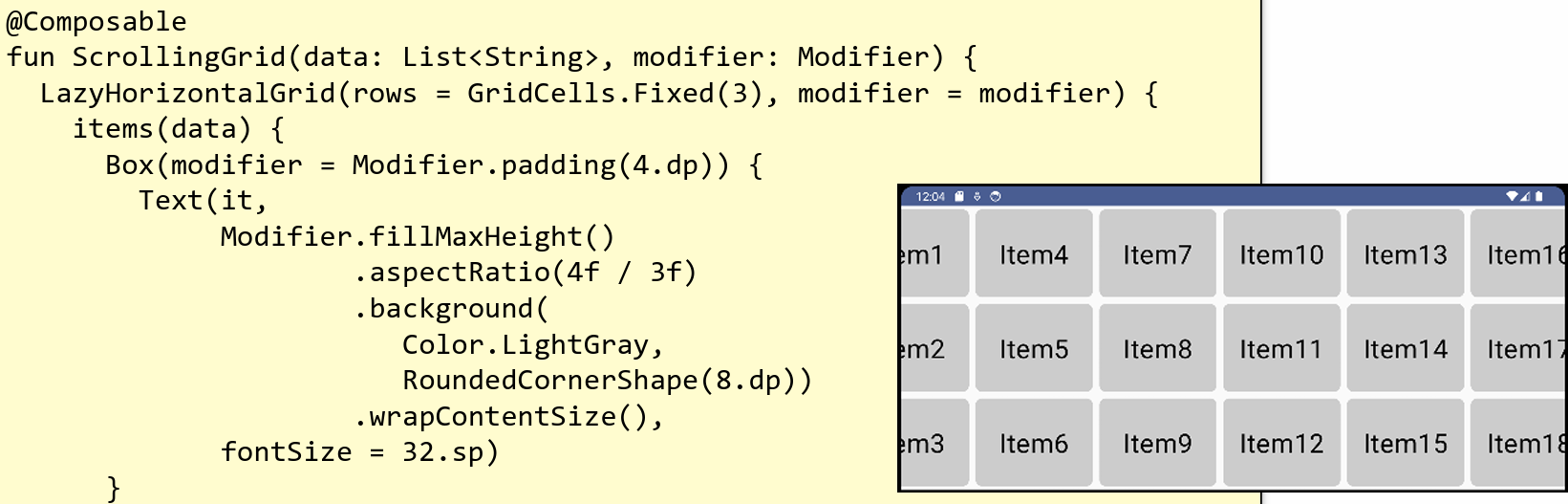
#### Lists & Grids

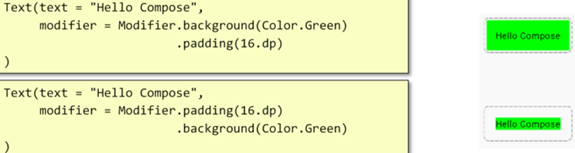
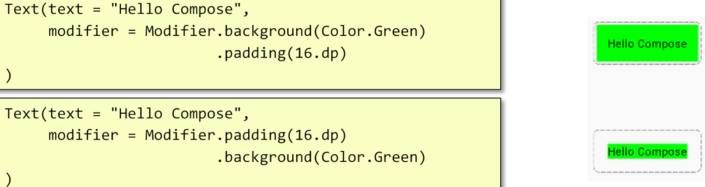
Implementa lo **scrolling** nel caso in cui il nr di elementi sia maggior dello spazio disponibile. Inoltre, la maggior parte dei blocchi non vengono eseguiti dalla GPU quando esegue in quanto è **LAZY** → quando inizio a scrollare, istanzio gli elementi che diventano visibili.

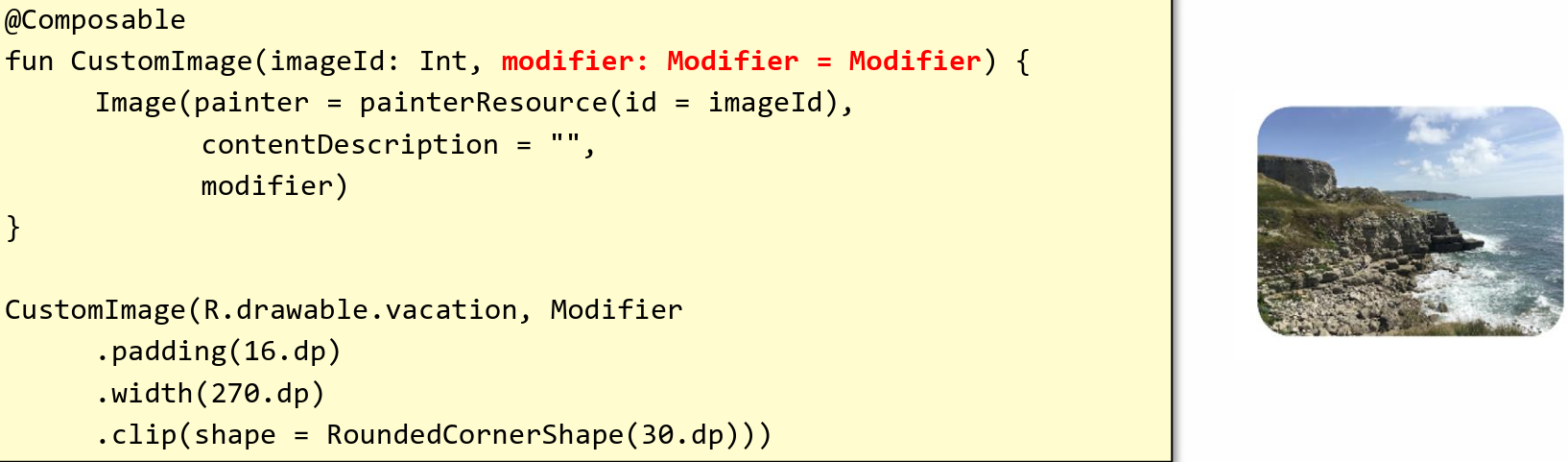
Quando si crea un Lazy column/grid, ci sono un insieme di parametri e composizione gerarchica. Accetta un receiver che ha LazyListScope → fornisce una serie di metodi:

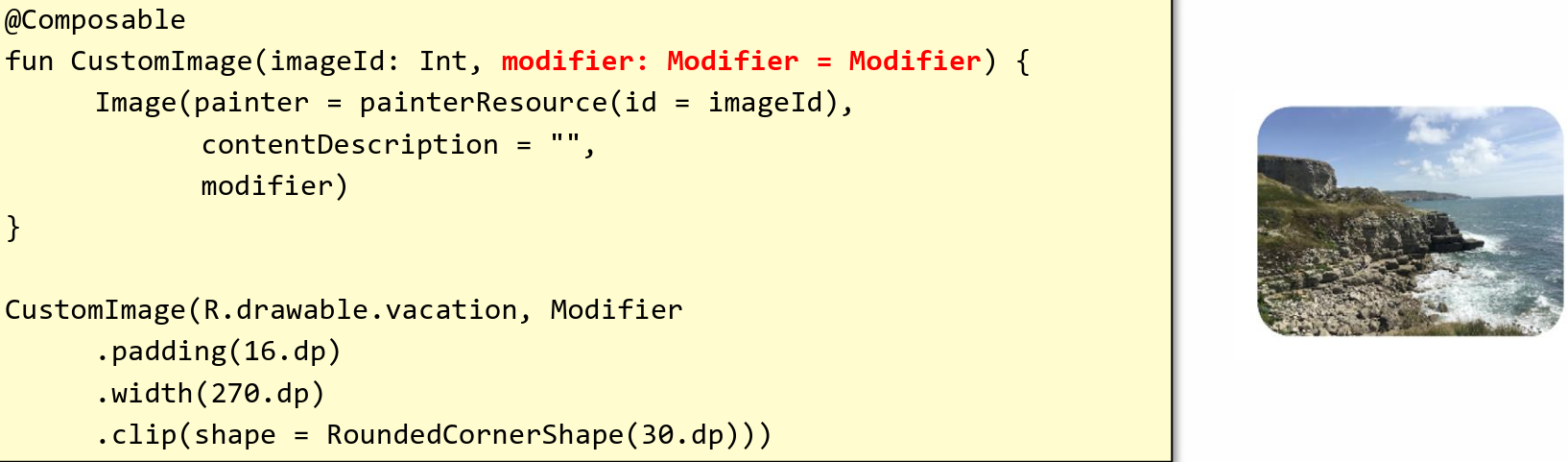
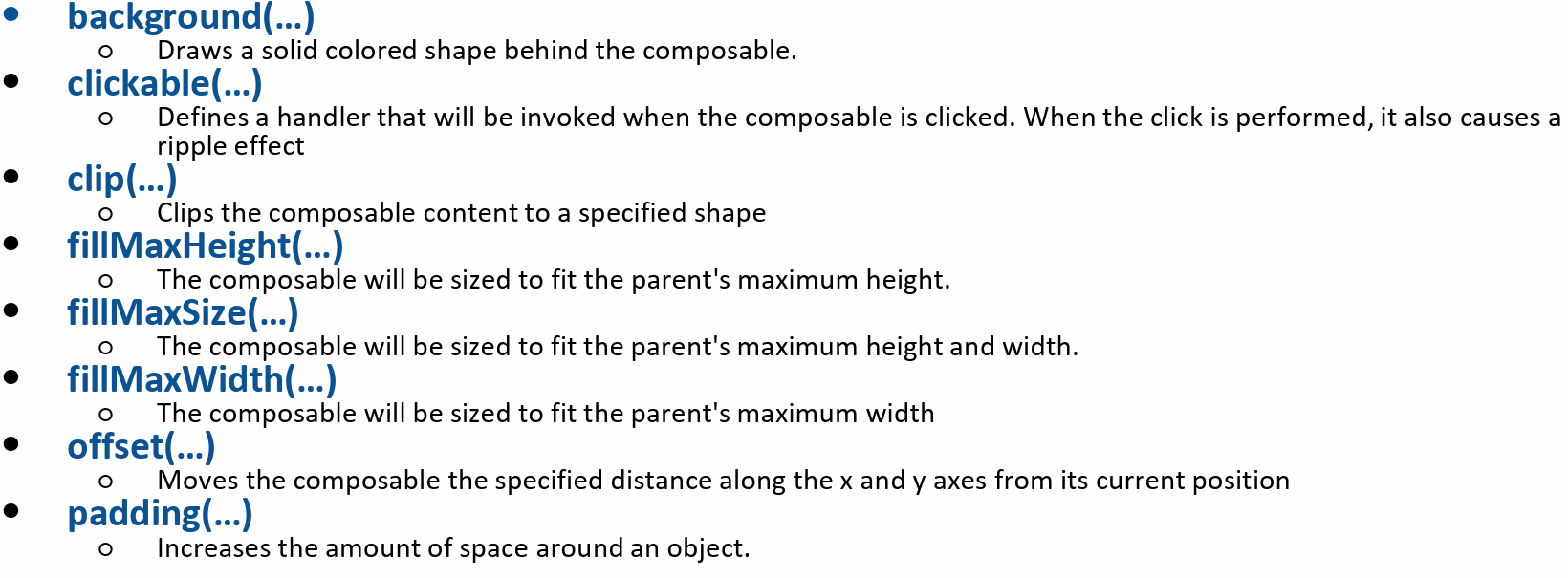
* items: accetta una lista o un array di items e una lambda responsabile di emettere il contenuto per ogni elemento visibile nel container
* item() : come sopra ma per un singolo elmemento
* indexedItems(): provvede una lambda function con un parametro extra che riporta l’indice corrente
* stickyHeadeR() emette item header che rimane pinnato anche se si scrolla.

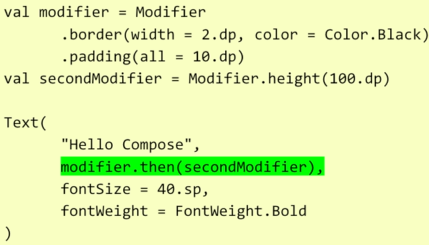
**LIST**

**GRID**

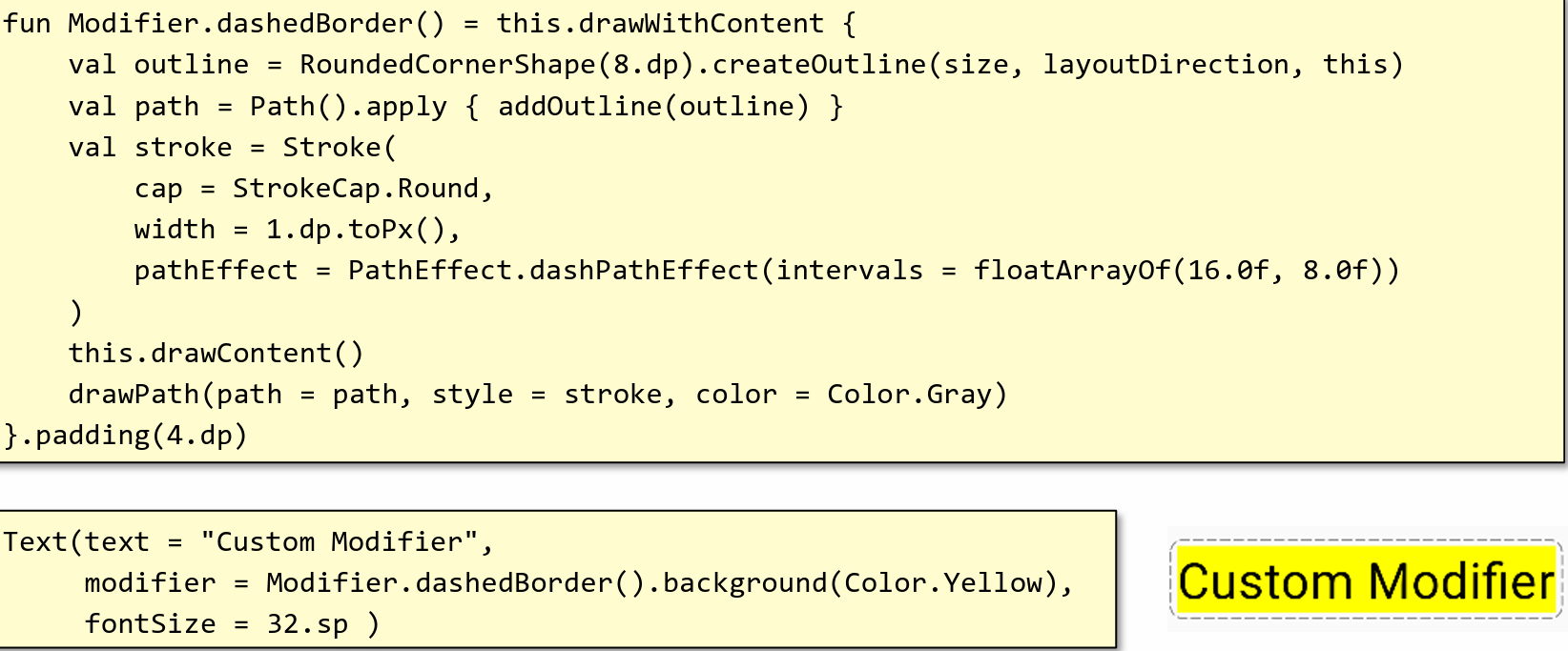
**MODIFIERS**: forniscono decoratori e comportamenti per elemento a cui applichiamo.  
 **NOTA**: l’ordine conta

Possono essere forniti ad ogni composable, di default è settato come empty.



**Then** → permette di appendere una ModiferList ad un’altra ModiferList

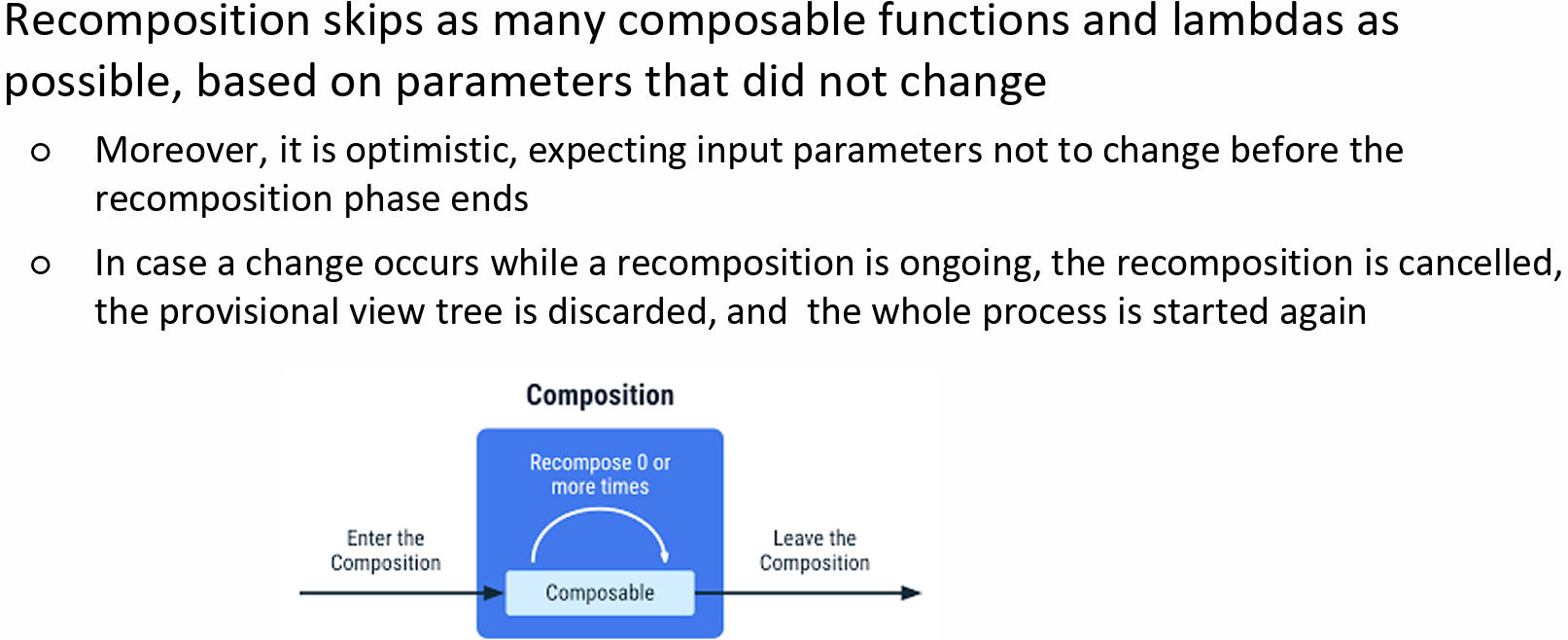
### Custom modifier

Si possono creare dei modifier. Per fare ciò si può partire da drawWithContent().

* Direzione viene presa dalla direzione che arriva nella lambda → crea outline
* Outline trasformata in path → per disegnarlo
* Stroke

#### Composable possono essere **statefull** → dipendono dai dati → cambiano

Quando i dati/parametri/stati\_interni cambiano, riceve una notifica e deve ridisegnarsi → **Recomposition.** → Android reinvoca la funzione composable con i dati aggiornati.   
***NOTA****: non posso usare le variabili locali per storare lo stato in quanto scompaiono e le variabili globali sono inadeguate in quando ci sarebbe facilmente incongruenza a causa del parallelismo.*

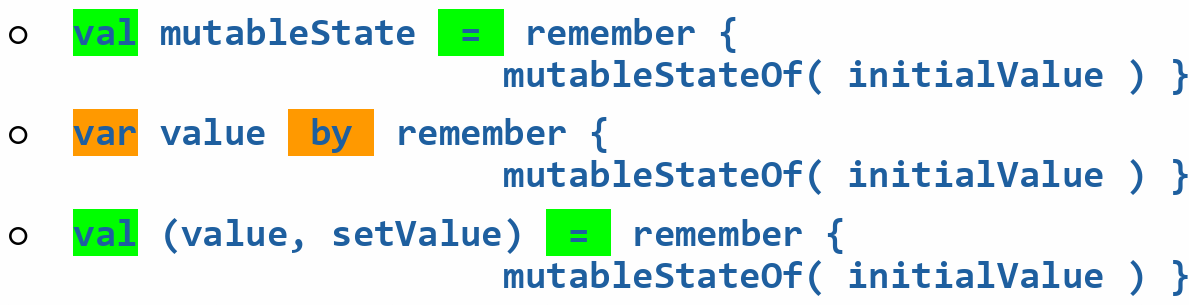
Android prova ad essere conservativo → prova ad evitare ricomposizione per elementi che non cambiano.

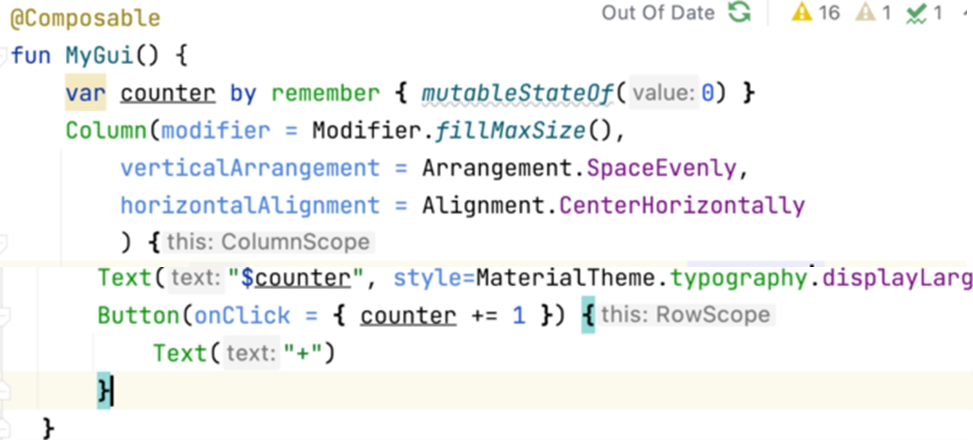
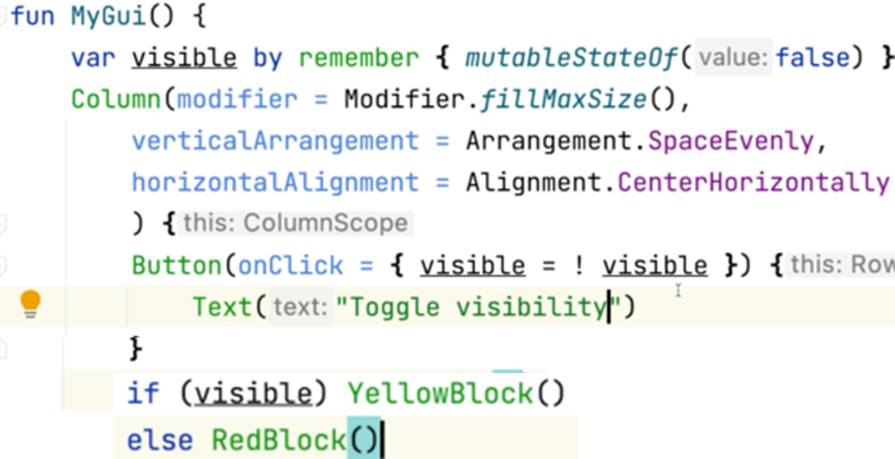
🡪 No side effect

*Nonostante ciò si possono avere dei side effect nelle callBack → es: OnClick*

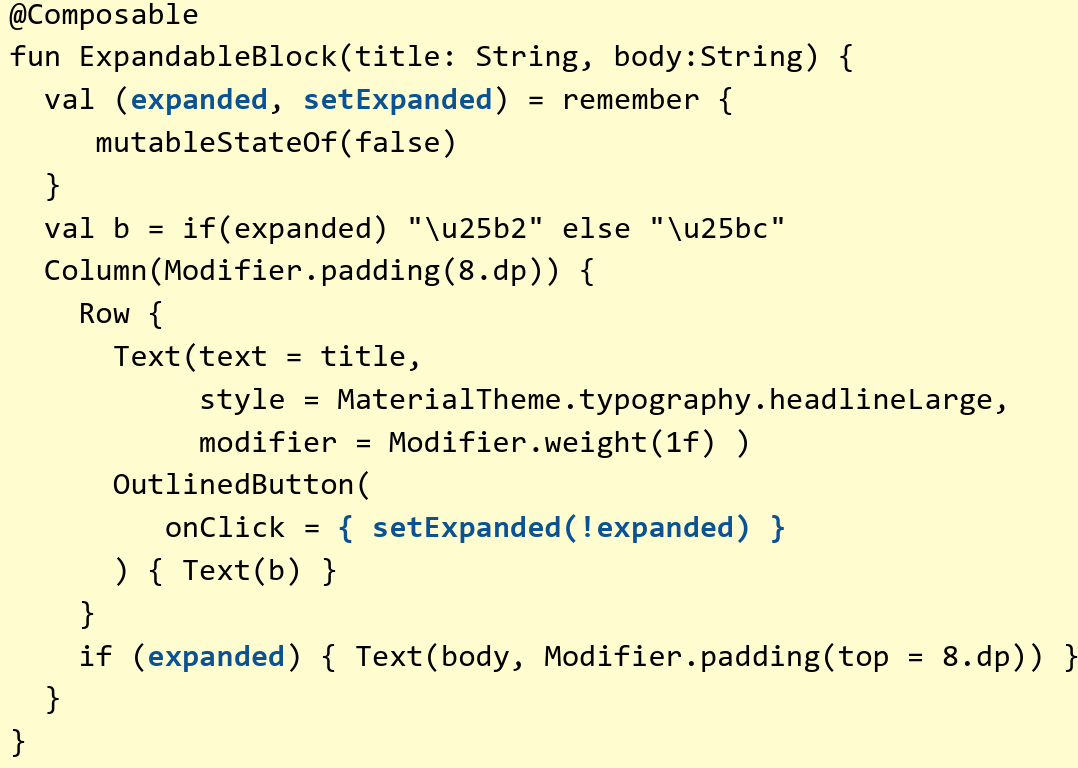
Android fornisce un container MutableState<T> → monade che wrappa ogni tipo di dato:

* **State**<T> :incapsula un pezzo di dato
* **remember**(..) : permette di tracciare ogni modifica e chiamare il processo di ricomposizione

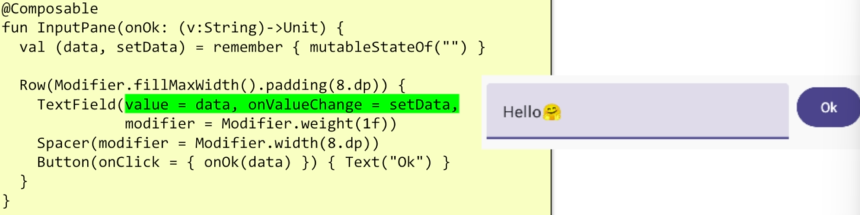




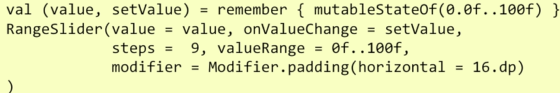
****

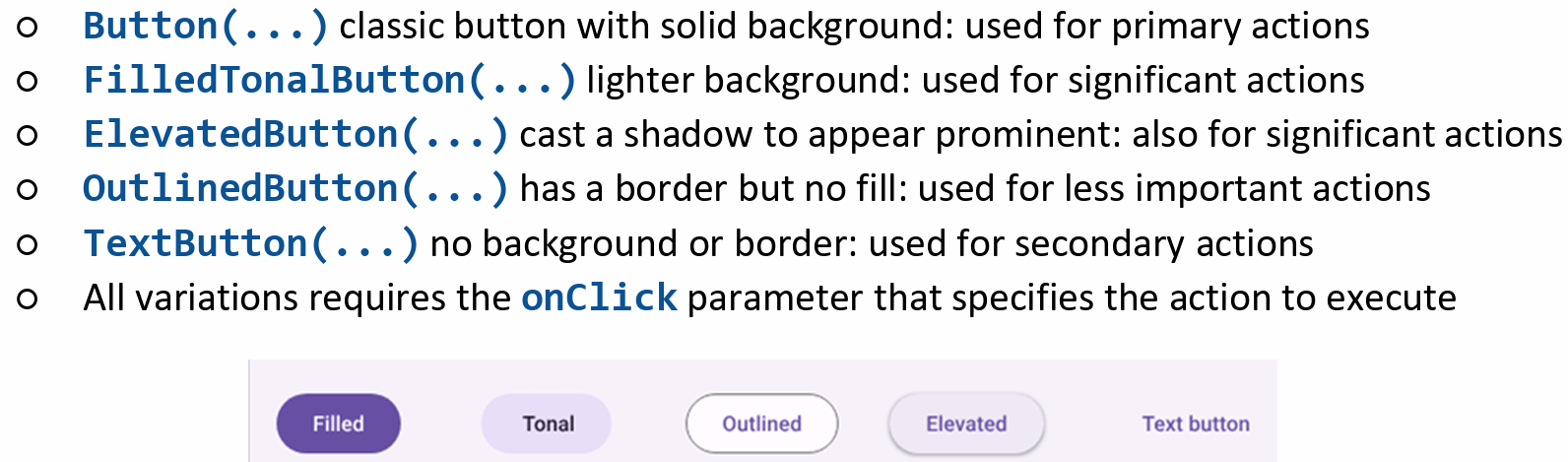
**espansione**

**Input widgets**

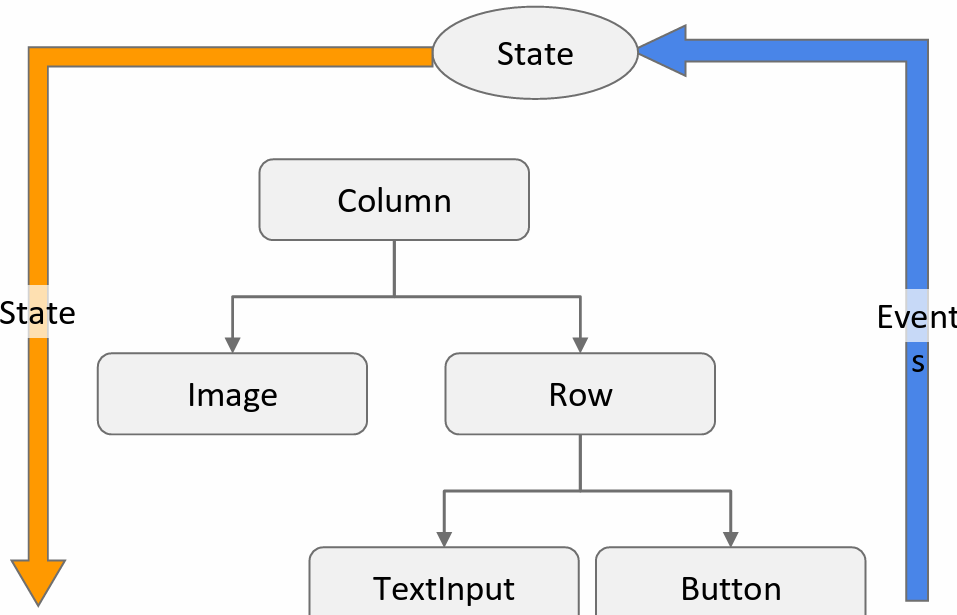
* **TextField** → blocco dove utente può scrivere qualcosa e nel caso in cui non scriva niente, ci sia un placeholder

È stateful e deve mantenere lo stato in memoria, se l’utente ha scritto qualcosa, deve mostare quello che ha scritto l’utente con lo stile definito per utente

* **Switch** → permette all’utente di scegliere tra due stati
  + Richiede cjecked e onCheckedCange
* **Slider** → permette all’utente di selezionare un valore in un range
  + Richiede value e onValueChange
* **RangeSlider** → permette all’utente di selezionare 2 valori in un range
  + Valori ClosedFloatingPointRange<Float>

**Button widgets**: permettono iterazione con utente:

Quando si ha uno stato, bisogna capire dove salvarlo (abbiamo capito che non vanno bene variabili locali o globali).

🡪**UNIDIRECTIONAL DATA FLOW**: lo stato è mantenuto al top dell’albero della logica ed è passato come parametro ad ogni elemento figlio che necessita accedere e ogni funzione. Ogni cambiamento è una conseguenza dell’iterazione con utente, nel caso di cambiamento il top component sarà recomposed e la modifica sarà propagata ai figli.

Quando si crea un’attività, l’attività è fornita con dei **viewModel** che possono essere acceduti da ogni composable. ViewModel contengono lo stato e metodi che modificano lo stato.

*Quando un’attività inizia, il viewModel viene creato, se un’attività viene distrutta → viewModel distrutto, Nel caso in cui un’attività viene ditrutta per essere successivamente ricreata → recompose (rotazione schermo), ViewModel non viene distrutto*.

* + - * Storiamo mutable state

**Material library** fornisce guidelines per stile e definire cose.

* Designer sceglie max 2,3 colori.
  + Il primary color deve essere usato per button, headlines, top barr.
  + Secondary Color: molto distante dal primary color → per azioni che devono essere totalmente diverse dal normale.
* UI contiene un insieme di file modificabili