# C.3 – Liste

Difficilmente troveremo un’app che non include qualche forma di lista: dai post di un social network ai video di youtube, le liste sono il metodo di esposizione preferito per contenuti numerosi che condividono un formato.

Il vantaggio principale del formato lista è che può mostrare un numero praticamente illimitato di informazioni, spacchettando per l’utente le fasi di ricerca e fruizione in due momenti distinti.

*Crea un nuovo progetto con una Empty Activity.*

## RecyclerView

Il widget che si usa per mostrare liste è la *RecyclerView*, e già dal nome, è possibile intuire che è una view *che* *ricicla.* Le liste infatti possono essere molto impegnative in merito all’impiego delle risorse hardware: per ovviare a questo problema, in questa view è stato sviluppato un sistema per il quale vengono messe in memoria solo le risorse necessarie per mostrare all’utente le righe che sta guardando in quel momento. Le righe fuori dallo schermo vengono messe in stanby ed eventualmente “riciclate” per caricarne altre visibili.

A picture containing graphical user interface

Description automatically generated

La *RecyclerView* è un componente complesso, che è composto da varie parti che vanno a interagire tra di loro:

* *Layout*
  + **La view *RecyclerView*** conterrà le righe
  + **Un layout xml** rappresenta una singola riga, e verrà ripetuto per ogni elemento della lista
  + **La classe LayoutManager** indica il formato della lista: una lista di elementi singoli o una griglia.
* *Data*
  + **Un dataset** in formato lista (List<Type>) conterrà gli elementi che andremo a mostrare
  + **La classe Adapter** gestisce il dataset, informando la RecyclerView su Type, grandezza, ed eventuali cambiamenti durante l’utilizzo.
  + **La classe ViewHolder** accoppia un elemento del dataset ad una riga, per popolare le view (wiring)

Nel layout, inserisci una *RecyclerView* impostando i constraint al parent:

Graphical user interface, text

Description automatically generated

È consigliato che, come ScrollView, RecyclerView abbia delle dimensioni predefinite così da poter gestire al meglio il suo contenuto – evitare *wrap\_content* quando possibile.

Imposta anche un colore di sfondo leggermente scuro, sul quale poi si andranno a mostrare le righe.

## Layout riga

Aggiungiamo il layout che andrà a definire ogni singola riga.

Crea un nuovo file XML nella directory *res/layout* e aggiungi al nome il prefisso **row**per differenziarlo dagli altri layout. Sarà un layout rettangolare, con dentro un campo di testo e un'immagine.

Graphical user interface, text

Description automatically generated

Nel ConstraintLayout della riga sono state utilizzate alcune accortezze:

* Questo layout sarà mostrato all'interno della RecyclerView, è quindi buona norma fornire un colore di sfondo in maniera tale da distinguerlo dal contenitore.
* ha un margin bottom impostato a un 1dp, questo farà in modo che le righe saranno leggermente separate tra di loro.
* La sua Height è impostata a wrap content, così la riga sarà alta tanto quanto il contenuto.

Tornando sul layout della pagina, impostiamo all'interno della RecyclerView una preview del layout che abbiamo appena creato, aggiungendo il campo *tools:listitem*

Text

Description automatically generated

Il colore grigio che avevamo impostato prima come sfondo alla RecyclerView ora funge da separatore, dovuto al fatto che il layout di riga ha un margin bottom.

## LayoutManager

Il LayoutManager si occupa del formato generale della lista, e di come funziona tecnicamente lo scroll con le gestures. Una lista puo’ essere mostrata in 2 formati principali:

* Un formato lineare dove ogni riga rappresenta un elemento
* Un formato a griglia, dove ogni riga contiene più elementi in una serie di colonne

Nel nostro caso scegliamo il formato lineare, e non avendo bisogno di particolari customizzazioni andiamo ad impostare il LayoutManager di default.

Text

Description automatically generated

1. Dataset

Entriamo ora nella parte riguardante i dati da mostrare.

La lista avrà bisogno di un modello, rappresentato da una classe, che ne identificherà il contenuto. Comunemente vengono utilizzate delle DataClasses per questo scopo.

Le data classes spesso sono complesse: pensando ad una classe User, ci immaginiamo i numerosi campi che compongono la sua identità.

Crea una nuova DataClass che rappresenterà il singolo elemento della lista.   
Ogni elemento avrà un *id* e un *color* che lo identificano. Come default di entrambi, inserisci dei valori randomici.

Graphical user interface, application

Description automatically generated

In MainActivity, crea un dataset di 50 elementi PrettyItem

Text

Description automatically generated

## Adapter e ViewHolder

Definiamo ora le classi che andranno a popolare la lista.

Crea una classe MainListAdapter, che avrà come input il dataset appena creato

Text

Description automatically generated

Al suo interno, crea una inner class MainListViewHolder, che:

* accetta come input il ViewBinding del layout riga che hai creato
* estende RecyclerView.ViewHolder, a cui passa la root del binding
* ha una funzione *bind*con input un elemento del dataset

Text

Description automatically generated

Ora fa che MainListAdapter estenda RecyclerView.Adapter: quest’ultimo chiederà di essere tipizzato con un ViewHolder, che sarà MainListViewHolder

Text

Description automatically generated

L’IDE ci informa che è necessario implementare i membri di Recyclerview.Adapter, fallo:

Graphical user interface, text, application, chat or text message

Description automatically generated Graphical user interface, application

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

Sono 3 funzioni necessarie, i loro compiti sono:

* **onCreateViewHolder** viene lanciato quando la RecyclerView viene inizializzata, creando una serie di righe che permetteranno di essere ciclate affinchè l’utente veda sempre qualcosa nella lista
* **onBindViewHolder** viene lanciato quando una riga diventa visibile, popolandone i campi. Qui faremo il wiring.
* **getItemCount** indica il numero degli elementi che saranno disponibili nella lista

Comincia con getItemCount. Il numero di elementi da mostrare è solitamente il numero degli elementi presenti nel dataset, quindi:



onCreateViewHolder richiede come output un MainListViewHolder, crealo:

* Si utilizza la versione di *LayoutInflater.from* che richiede anche di specificare il parent.

Text

Description automatically generated

onBindViewHolder richiede di popolare le view. Per questo, deleghiamo alla nostra implementazione del viewHolder, a cui abbiamo aggiunto la fun *bind*:

Graphical user interface, text

Description automatically generated

Ora l’Adapter è completo.

Codice:



## Wiring

E’ ora necessario comunicare al ViewHolder come gli elementi del dataset andranno a popolare le sue view.

All’interno della funzione *bind,* utilizza i campi *id e color* di PrettyItem per popolare gli elementi della riga.

Text

Description automatically generated

Questa funzione verrà ripetuta per ogni elemento della lista.

Ora l'unica cosa che manca è comunicare alla RecyclerView quale sarà il suo adapter. Tornando su MainActivity, impostiamo il campo *adapter* creando un’istanza della nostra implementazione:

Text

Description automatically generated

Ora possiamo far partire l’app e vedere il risultato:

A screenshot of a phone

Description automatically generated with medium confidence

## Esercizio: Card collection 1

Crea un’app dove viene mostrata una collezione di carte

* Gli elementi del dataset devono avere un
  + ID numerico (1..999)
  + “value” valore numerico (1..100)
* Per il layout, utilizza come viewgroup root CardView, con al suo interno un constraint Layout
* Il layout mostrerà id e valore, dovranno essere distinguibili
* Il layout contiene anche una ImageView, contenente un’icona. Gli oggetti con un ID pari avranno un’icona, mentre quelli con ID dispari ne avranno un’altra.

## Aggiornare il dataset

È molto comune che il dataset subisca delle modifiche: elementi possono essere aggiunti o rimossi, o possono cambiare i loro valori interni. La recyclerView non si aggiorna automaticamente durante questi eventi, ma deve essere notificata. Sarà l’adapter a ricevere la notifica.

Aggiungiamo la possibilità di rigenerare il dataset

Nel layout activity\_main aggiungi un Floating Action Button (FAB)

Graphical user interface, text, application

Description automatically generated

In MainActivity, è ora necessario che il dataset diventi mutabile, così che possiamo aggiornarlo.



Estrai la creazione della lista in una funzione, così che sia possibile riutilizzarla.



Ora facciamo in modo che, alla pressione del FAB, il dataset venga sbiancato per essere rimpiazzato da uno nuovo.

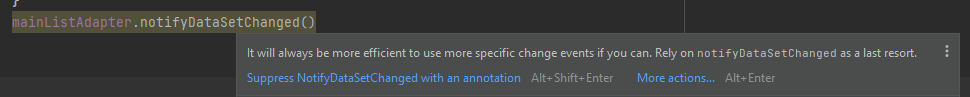
Text

Description automatically generated

L'oggetto dataset non è cambiato, ma il suo contenuto sì. L’adapter ha già un riferimento a dataset, quindi non è necessario modificarlo, però è necessario notificargli che il contenuto è cambiato, in quanto i binding tra elementi e ViewHolders sono stati già effettuati. La funzione *adapter.notifyDataSetChanged()* rimuove tutti i binding dalla recyclerView e li rigenera.

Lanciando l’app, vediamo che alla pressione del FAB il nuovo dataset viene correttamente mostrato.

L’IDE ci notifica che *notifyDataSetChanged* è un’operazione pesante:



Nel nostro caso, il suo uso è giustificato, perché abbiamo modificato tutto il dataset in blocco.

Adapter fornisce una serie di metodi alternativi da utilizzare per modifiche meno invasive, ad esempio quando un singolo membro del dataset viene modificato, o una parte del dataset viene rimossa.

Text

Description automatically generated

## Kotlin: operazioni sulle liste

Non è una casualità che la recyclerview funzioni tramite le liste. Kotlin ha una gran capacità di ottimizzare l'utilizzo di liste, oltre a fornire un gran numero di funzioni di aiuto per lavorarci sopra.

Il nostro dataset è ora una mutableList, che al contrario della List (immutabile) ha la capacità di aggiungere o rimuovere i suoi membri. Abbiamo visto un esempio di questo quando abbiamo prima sbiancato la lista (clear) e poi abbiamo aggiunto ad essa tutti i membri di un’altra lista (addAll)

MutableList è un discendente di List, la cui unica capacità aggiuntiva è di avere quei pochi metodi e qualità necessarie per essere, appunto, mutabile. Per il terzo principio SOLID - Liskov substitution - può essere quindi castato a List ed utilizzato come essa.

Kotlin ci aiuta effettuando smart cast al posto nostro quando possibile. Lo abbiamo utilizzato quando abbiamo passato dataset (MutableList) ad adapter (che richiede una List).

Aggiungiamo il requirement di avere gli elementi ordinati per id crescente

Per farlo, dobbiamo modificare l'ordine degli elementi della lista quando viene creata in *generatePrettyItems*. Essendo un'operazione comune, Kotlin ci offre ho un metodo precostruito:

Text

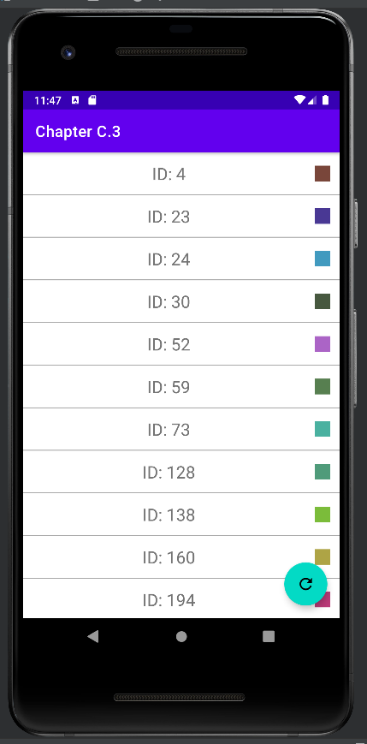
Description automatically generated

La funzione *sortBy* ordina gli elementi della lista in base ad un campo che gli specifichiamo , e modifica la lista su cui viene invocata (è una funzione che ritorna Unit). Utilizziamo quindi *apply* per applicarla alla nuova MutableList.  
sortBy ha come input la funzione *{ inputValue -> outputValue }*. Essendo *inputValue* l’unico parametro di questa funzione, può essere abbreviato in *it.*

Text

Description automatically generated

Questa è la forma di scrittura preferita.



Rimuoviamo dalla lista tutti i membri con ID dispari

Aggiungiamo questa condizione utilizzando la funzione *filter* che ci viene fornita da Kotlin

Text

Description automatically generated

*filter* ritorna una nuova *List,* quindi andremo ad accodarla alle altre modifiche.

*Filter* richiede come input la funzione *{ inputValute -> boolean }*, ovvero una condizione per la quale gli elementi saranno filtrati, scartando quelli che non la rispettano.

La nostra modifica ha avuto un effetto indesiderato: ha cambiato l’output della funzione *generatePrettyItems* da MutableList a List, ma la mutabilità ci è necessaria alla funzionalità di rigenerazione lista quando viene spinto il FAB.

Possiamo ovviare a questo ricastando la lista a Mutable:

Text

Description automatically generated

A screenshot of a phone

Description automatically generated with medium confidence

Queste due funzioni utilizzano dei pattern molto comuni non solo per le liste ma anche in altre applicazioni in Kotlin. Lo standard per le funzioni in input è:

* Una funzione **selector o transform** prende in input un valore e ritorna un valore - *{ inputValue -> outputValue }*   
  Identifica un **modello**.
* Una funzione **predicate** prende in input un valore e ritorna un boolean – *{ inputValue -> boolean }*Identifica una **condizione**.

## Kotlin: map()

Una delle operazioni sulle liste più vantaggiose è **.map()**

Questa funzione permette di trasformare una lista in un’altra*.*

Poniamo ad esempio di voler creare gli ID delle nostre PrettyItems a partire da una lista che contiene i primi 10 numeri della serie di Fibonacci:

Text

Description automatically generated with medium confidence

La soluzione classica è creare una nuova lista vuota, per poi inserire membro per membro, tramite un ciclo for su *fibs*, tutti i PrettyItem.

Text

Description automatically generated

.map permette di sintetizzare questa comune architettura:

Graphical user interface, text

Description automatically generated

Possiamo cosi’ procedere a rendere *inline* la nostra funzione



## Immagini con Glide

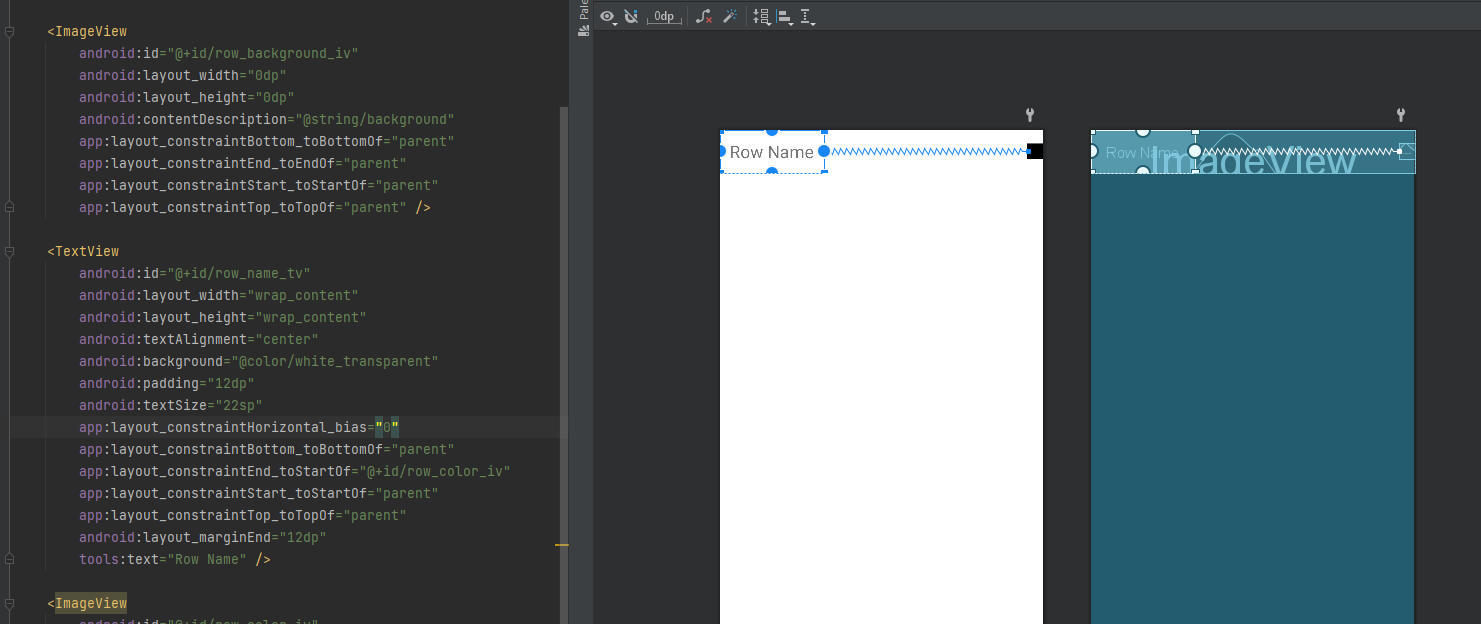
La gestione delle immagini è un tema particolarmente complicato in merito alle risorse di sistema. Per questo sono state sviluppate delle librerie apposite, in questo corso useremo Glide.

Nel *build.gradle****:app*** aggiungi la dipendenza ed effettua un sync:

implementation 'com.github.bumptech.glide:glide:4.14.2'

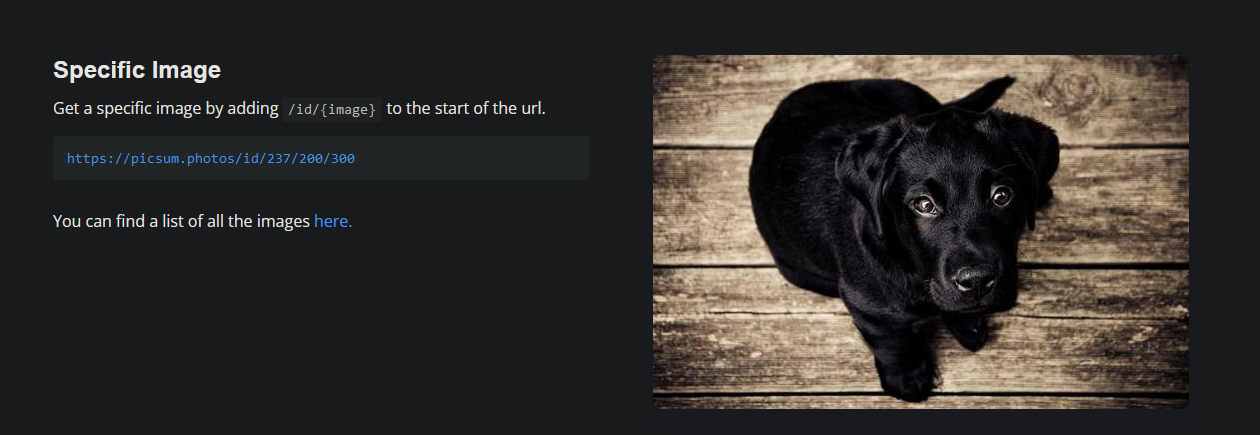
Nel layout di riga della recyclerview, fai le seguenti modifiche:

* Aggiungi una ImageView che fungerà da sfondo
* Cambia il background della textview per farla risaltare sul nuovo sfondo, spostala dal centro alla sinistra del layout, e rendila wrap\_content



Per trovare una serie di immagini randomiche, in questo esempio useremo il servizio <https://picsum.photos/>

Ci offre la possibilità , tramite uno dei suoi endpoint, di ottenere un’immagine specificando l’ID e richiedendone le dimensioni (200x300 in questo screenshot):



Useremo questo endpoint del servizio per emulare un database di immagini della nostra app.

Aspettiamoci che non tutte le immagini vengano trovate nel servizio, perché stiamo usando id causali!

Dobbiamo dichiarare nel manifest che la nostra app intende accedere ad internet, altrimenti avremo una eccezione.

A screen shot of a computer

Description automatically generated with medium confidence

In PrettyItem, aggiungi una proprietà *imageUrl*  con il seguente valore:

val imageUrl = "https://picsum.photos/id/$id/500/500"

A picture containing text, font, screenshot

Description automatically generated

Nell’adapter, aggiungi l’invocazione a Glide:

Text

Description automatically generated

Glide viene inizializzato con un’invocazione a builder, che richiede:

* **Glide.with(view)** specifica in che view lavorerà glide. Utilizziamo il root del binding per identificare il nostro layout.
* **.load(source)** può caricare non solo drawables e formati immagine, ma anche risorse remote come in questo caso. Sfruttiamo l’id del nostro elemento per richiedere un’immagine al servizio, con dimensioni 500x500.
* **.into(destination)** specifica l’imageView che riceverà l’immagine.

Lancia l’app:

A screenshot of a phone

Description automatically generated with low confidence

Le immagini vengono caricate, ma non hanno la dimensione corretta per la nostra riga. In un caso reale, potremmo richiedere le immagini con le dimensioni corrette, ma per ora stiamo utilizzando immagini casuali. Usiamo quindi una funzionalità di adattamento immagine di Glide *centerCrop()*:

Graphical user interface, application

Description automatically generated

Questo ritaglierà l’immagine il minimo per farla entrare nella nostra imageview, centrata.

A picture containing text, display, electronics, screenshot

Description automatically generated

## Esercizio: Card collection 2

Aggiungi a card collection:

* Per ogni riga, un’immagine dinamica che cambia in base a ID
* L’utente puo’ ordinare la lista tramite due pulsanti, uno per ID e l’altro per Valore
* Spingendo un terzo pulsante, è possibile sostituire un elemento randomico del dataset con uno nuovo.