

Corso Android

Contents

A.1 – Kotlin	4
1. Variabili.....	4
2. Null-safety	4
3. Classi, Data Classes	5
4. Funzioni, Costruttori.....	7
5. Singleton.....	8
6. run, apply, let, also	8
7. map()	9
8. Esercizio: Persona virtuale.....	10
A.2 – Android studio	11
1. Creazione e struttura progetto.....	11
2. Funzionalità base.....	12
3. Emulatore	12
B.1- Nozioni di base di layout	15
Creare un file di layout	15
Constraint Layout	15
DP, margin e padding	16
Editor Grafico e Testuale	16
Aggiungere elementi: Design	16
Aggiungere elementi: Code	18
Best Practice: no hardcoding	19
Larghezza e Altezza widget.....	20
Immagini.....	20
Best Practice: resource naming	24
Chains	26
Esercizio: Postcard o Meme	29
B.2 – ViewBinding e Azioni	30
1. Button.....	30
2. Activity e Context	30
3. View Binding.....	31
4. Update interfaccia	31
5. Acquisizione risorse	32
6. Listeners	32
7. Best Practice: Design for readability	33
8. Condizionali	34
9. Esercizio: Lancio Dadi	36
B.3 – Debugger Intro	38
1. A cosa serve il Debugger	38
2. Breakpoints.....	38
3. Evaluate	41
B.4 – Testing: Unit Test	43
1. Introduzione a Unit Test su Android	43
2. Setup Progetto.....	43

3.	Creazione di un test.....	44
4.	Naming Convention.....	45
5.	Assertions	46
6.	Successo e Fallimento	46
7.	Strategie di test	47
8.	@Before, @After.....	48
9.	Esercizio: test Lancio dadi.....	50
C.1 – Input Utente		51
1.	EditText, TextInputLayout	51
2.	RadioButtons	55
3.	Compatibilità dei widget	56
4.	ScrollView	57
5.	Esercizio: Hero Generator	59
C.2 - Material Design		60
1.	Introduzione a design UI e UX	60
2.	Colori	61
3.	Theme.....	62
4.	Style	64
5.	Icône	66
6.	Launcher Icon	68
7.	Esercizio: Stilizziamo Hero Generator	70
C.3 – Liste.....		71
1.	RecyclerView	71
2.	Layout riga	72
3.	LayoutManager	73
5.	Adapter e ViewHolder	74
6.	Wiring	76
7.	Esercizio: Card collection 1.....	78
8.	Aggiornare il dataset	78
9.	Kotlin: operazioni sulle liste.....	80
10.	Kotlin: map().....	82
11.	Immagini con Glide	83
12.	Esercizio: Card collection 2.....	88
C.4 - Integration test.....		89
1.	Environment setup	89
2.	Introduzione a IT su Android	90
3.	Espresso.....	91
4.	Recyclerview Utilities	93
5.	Esercizio: Test Card Collection	94
D.1 – Activity e Intents		95
1.	Activity.....	95
2.	Controllo delle risorse da parte del sistema.....	97
3.	Intents.....	97
4.	Esercizio: Condividi Card Collection Power	101
Fragment e Navigation		102
1.	Fragment	102
2.	Navigation Graph.....	110
3.	Navigazione tra frammenti.....	112

4.	Inviare dati tra frammenti.....	118
5.	Setup Barra di Navigazione	120
6.	Esercizio: Card Collection detail	121
D.3 – Mockito e End-To-End test.....		122
1.	Testare i Fragment.....	122
2.	Mock e Mockito.....	124
3.	E2E Testing	126
4.	Esercizio: Test Card collection	129
E.1 – JSON		130
1.	Serializzazione\Deserializzazione	131
2.	JsonToKotlin	132
3.	Gson.....	135
E.1 - MVVM pattern.....		137
1.	Model	139
2.	ViewModel	140
3.	View.....	141
4.	Concetti di ownership	142
5.	Esercizio: shared viewmodel	143
F.1 – Async Programming.....		144
1.	Concurrency e Threads.....	144
2.	Coroutines	146
3.	Suspend Functions.....	147
4.	viewModelScope, launch()	148
5.	Futures.....	149
6.	Esercizio	151
F.2 – Servizi Remoti e Retrofit		152
1.	REST API service.....	152
2.	Retrofit.....	153
3.	Interface	154
4.	Retrofit Client	156
5.	HttpClient	158
6.	Esercizio	161
F.3 – MVVM Testing		162
1.	Model	162
2.	ViewModel	164
3.	Esercizio.....	169
Livedata e Observer Pattern.....		170
1.	Observer Pattern	170
2.	LiveData	171
3.	setValue,.postValue.....	173
4.	Testare LiveData	174
5.	Esercizio	176
H.1 - Dependency Injection e Hilt.....		177
1.	DI in Android.....	177
2.	Hilt Project Setup.....	178
3.	Entry Point	179
4.	Inject: constructor	180

5.	Inject: Module	180
6.	Instrumented Test Setup.....	184
7.	Instrumented Test	188
I.1 -	SharedPreferences	193
1.	Scopo delle SharedPreferences.....	193
2.	PreferenceManager.....	193
3.	Salvare e Recuperare valori.....	194
4.	Unit Test	195
5.	Esercizio.....	196
I.2 -	Room (DB)	197
1.	Scopo del DB.....	197
2.	Room setup	197
3.	Esercizio.....	201

A.1 – Kotlin

E' possibile provare il codice nel sito [Kotlin Playground](#)

<https://play.kotlinlang.org>

1. Variabili

La keyword **val** identifica variabili *final*, ovvero read-only

```
val number: Int = 1
```

var indica invece variabili che possono essere riassegnate

```
var string: String = "hi!"  
string = "hello!"
```

Kotlin capisce da solo il *Tipo* che si sta assegnando: viene quindi solitamente omesso nella scrittura a meno che non sia necessario per chiarificare il codice

```
val number = 1  
var string = "hi!"
```

Si può esplicitare il tipo di una variabile aggiungendolo dopo il nome:

```
val number: Int = 1  
var string: String = "hi!"
```

2. Null-safety

Kotlin è un linguaggio *null-safe*, ovvero integra nella definizione dei tipi se possono essere null o no, utilizzando il punto interrogativo:

```
val string: String = "Safe!"  
val nullableString: String? = null
```

Di base si lavora con tipi safe, ovvero che non possono produrre null.

Nel caso di tipi nullabili, è possibile evitare di scrivere null check con IF ELSE, accedendo alla variabile tramite la nomenclatura con il punto interrogativo:

```
val lowercaseString: String? = nullableString?.lowercase()
```

In questo esempio *lowercaseString* sarà anch'essa una String?, in quanto la funzione .lowercase() viene lanciata solo nel caso nullableString non sia null. Questo equivale a un check IF (null)

Per replicare IF(null) ELSE, è possibile utilizzare l'**Elvis Operator - ?:**

```
val lowercaseString: String = nullableString?.lowercase() ?: "default"
```

In questo caso lowercaseString non sarà mai null, perché in caso nullableString sia null, verrà valorizzata a "default"

L'elvis operator si chiama così perché ricorda i capelli di elvis ?:)

3. Classi, Data Classes

Le classi vengono dichiarate utilizzando la keyword *class*

```
class Person constructor(firstName: String) { /* class content here */ }
```

Ma la keyword *constructor* del costruttore primario è omessa

```
class Person (firstName: String) { /* class content here */ }
```

I parametri del costruttore possono essere utilizzati solo nelle assegnazioni delle *properties*. Assegnazioni e blocchi *init()* vengono eseguiti nell'ordine in cui sono scritti

```
class InitOrderDemo(name: String) {
    val firstProperty = "First property: $name"

    init {
        println("init of $name")
    }

    val secondProperty = "Second property: ${name.length}"

    init {
        println("init of ${name.length}")
    }
}
```

Invece di assegnare manualmente i parametri alle *properties*, è possibile dichiararli come tali

```
class Person(
    val firstName: String,
    private val secretName: String,
    var age: Int,
) { /*...*/ }
```

E' molto comune definire classi il cui unico scopo è mantenere dei dati. Kotlin ci fornisce le *data classes* che hanno delle facilitazioni, come funzioni di egualanza o di trasformazione in stringa leggibile. Affichè sia

valida, una data class deve avere le proprietà dichiarate nel costruttore. Utilizzare una data class per definire i modelli è lo standard.

```
data class Person(  
    val firstName: String,  
    private val secretName: String,  
    var age: Int,  
) { /*...*/ }
```

4. Funzioni, Costruttori

Le funzioni (metodi) hanno una struttura simile, utilizzando la keyword **fun**

Di base sono *public*, si può cambiare la visibilità con la keyword *private fun*.

```
fun performSum(first: Int, second: Int): Int {  
    return first + second  
}  
  
val resultIsTen = performSum(3, 7)
```

Possono essere dichiarate *inline*, ovvero il body può essere un'espressione. In questo caso il type è omesso.

```
fun performSum(first: Int, second: Int) = first + second
```

Quando sono membri di una classe, possono essere invocate con la struttura *dot notation*

```
MathHelper().performSum(3, 7)
```

Le funzioni possono essere dichiarate con argomenti di default, che possono essere omessi quando invocate

```
fun performSum(first: Int, second: Int = 7) = first + second  
val resultIsTen = performSum(3)
```

nel caso in cui sia necessario specificare quale dei due argomenti vogliamo passare, si utilizza l'invocazione con *named arguments*

```
fun performSum(first: Int = 3, second: Int = 7) = first + second  
val resultIsFive = performSum(second = 2)
```

se l'ultimo argomento è un *lambda*, può essere passato normalmente, o fuori in parentesi graffe.
Quest'ultimo è lo standard.

```
fun performSum(first: Int = 3, second: Int = 7, logme: () -> Unit) =  
    first + second  
  
val resultIsFive = performSum(second = 2) {  
    // this is logme  
}
```

5. Singleton

La keyword `object` permette di definire facilmente una classe con pattern *Singleton*:

```
object Car {  
    fun getCostByMaker(maker: String): Int {  
        // ...  
    }  
}  
  
Car.getCostByMaker("BMW")
```

Questa viene definita *object declaration*. Ci sono altri utilizzi chiamati *object expression* che servono ad altri scopi.

<https://kotlinlang.org/docs/object-declarations.html>

6. run, apply, let, also

Una particolarità di Kotlin è la sua capacità di definire dei *contesti temporanei* grazie alle Scope Functions.

Estremamente utile per creare costrutti logici e raggruppare ripetizioni.

Nell'esempio successivo, utilizziamo una variabile che viene poi invocata varie volte:

```
val person = Person(firstName: "Gino", lastName: "Pilotino", age: 12, hobby: "Transformers")  
println(person)  
person.incrementAge()  
person.changeHobby(new: "Lego")  
println(person)
```

Ecco una versione più Kotlinosa

```
Person(firstName: "Gino", lastName: "Pilotino", age: 12, hobby: "Transformers")  
.let { it: Person  
    println(it)  
    it.incrementAge()  
    it.changeHobby(new: "Lego")  
    println(it)  
}
```

Non solo abbiamo risparmiato l'assegnazione di una variabile, ma il codice è ora schematizzato all'interno di un solo costrutto.

run e *let* ritornano entrambi il **risultato dell'ultima riga scritta nel lambda**.

run identifica l'oggetto del contesto con *this*, mentre *let* con il parametro *it*

```
val person = Person(  
    firstName: "Gino",  
    lastName: "Pilotino",  
    age: 12,  
    hobby: "Lego")  
  
val myStory = person.let { it: Person  
    "At age ${it.age} i was in love with ${it.hobby}"  
}  
  
val sameStory = person.run { this: Person  
    "At age $age i was in love with $hobby"  
}
```

apply e **also** invece ritornano l'oggetto su cui si sta lavorando.

apply utilizza *this*, mentre **also** utilizza *it*

In questo caso *mutableListOf<Int>()* crea una lista vuota di Integer:

```
val numbers = mutableListOf<Int>()  
    .apply { this: MutableList<Int>  
        add(5)  
        add(2)  
        add(7)  
        sort()  
    }  
    .also { println(it) }
```

La variabile *numbers* viene modificata aggiungendo 3 numeri, poi ordinati in ordine crescente. Verrà inoltre stampata.

7. map()

Una interessante funzione riguardante gli *iterables* – oggetti come liste, array e mappe – è la funzione **.map()**

È una funzione che applica a tutti i membri dell'iterable la trasformazione dichiarata all'interno del lambda.

Questo permette di rimuovere strutture *for* e comprimere notevolmente il codice.

```
val brother = Person(firstName: "Mark")
val sister = Person(firstName: "Selma")

val persons = listOf(brother, sister)

val names = persons.map { it.firstName.lowercase() }
```

Nel nostro esempio a tutti gli elementi della lista *persons* viene applicato il trasform all'interno di *map()*: il risultato è una lista contenente i *firstname* portati a lowercase di tutti gli elementi di *persons*.

8. Esercizio: Persona virtuale

Crea il seguente programma (puoi usare Kotlin Playground):

- Dichiara una classe Persona che include: id, nome, cognome, età, cittadinanza e hobbies.
- Nome e cognome e hobbies sono campi read-only, mentre età, cittadinanza sono campi variabili.
- cittadinanza è impostata di default a “Italiana”
- hobbies è una lista di stringhe privata e mutable
- id è un double inizializzato con Random().nextDouble() del pacchetto java.util
- La classe ha una funzione che permette di aggiungere stringhe a hobby

Dopo aver instanziato la classe con i valori di default, fa in modo che questa venga:

1. stampata in un log come è stata creata nel costruttore
2. modificata cambiando età e cittadinanza
3. Aggiungi almeno 2 hobbies e ordinali per lunghezza della parola (basso prima).
4. stampata nuovamente dopo le modifiche

Una volta che l'oggetto è stato modificato e stampato correttamente:

1. Forma una lista popolata dall'oggetto e da 5 sue copie. Ogni copia deve avere un id diverso.
2. Estrai in una lista i 6 id dagli oggetti, e stampa la lista.

A.2 – Android studio

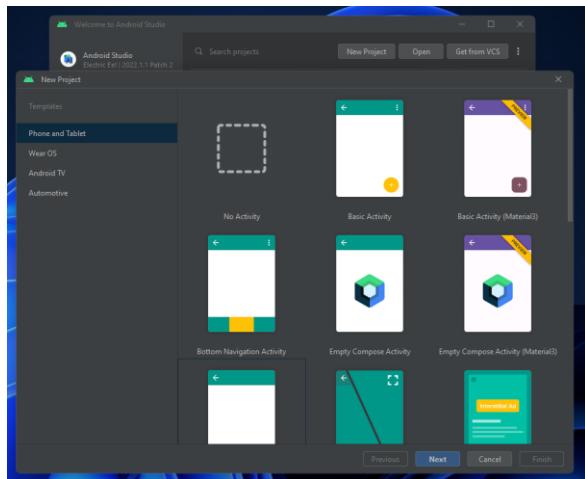
Android Studio è una versione specializzata di Jetbrains IntelliJ IDEA, una IDE creata per lavorare in Java e Kotlin. È un pacchetto di sviluppo Android completo di tutti gli strumenti, inclusi quelli in linea di codice.

Download link:

<https://developer.android.com/studio>

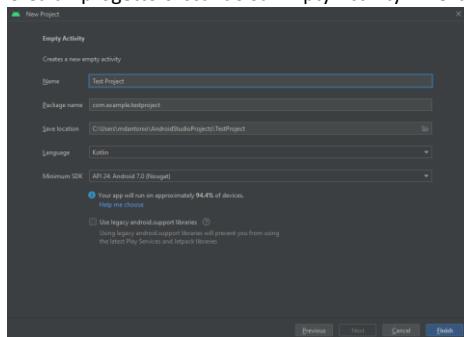
1. Creazione e struttura progetto

E' possibile creare un nuovo progetto dal pannello welcome o dal menu File.



L'ide ci propone alcuni template, molti dei quali utili sebbene non sempre aggiornati all'ultimo standard.

Crea un progetto cliccando su Empty Activity > Next e inserisci il nome del progetto:



- Package name: nome dei riferimenti interni del nostro progetto
- Save location: directory dove sono salvati i file
- Language: sviluppiamo in Kotlin
- Minimum SDK: minima versione di Android supportata. Cambiadola, si potrà vedere una stima di quanti, tra i device Android attualmente utilizzati, supporteranno la nostra app.

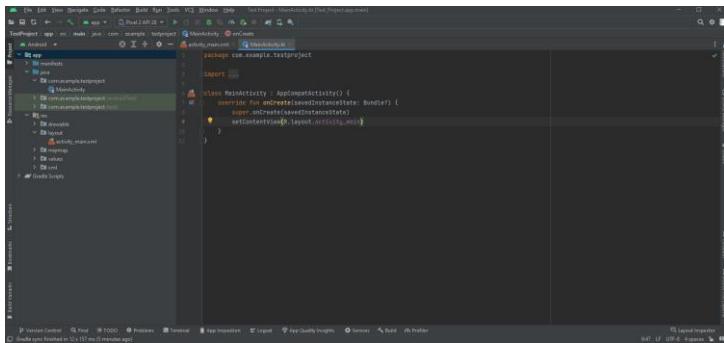
Cliccando *Finish* faremo partire la creazione dei file, l'importazione delle librerie standard e la prima build del progetto.

2. Funzionalità base

L'IDE offre alcune semplificazioni e strumenti fondamentali per il nostro lavoro:

- Salva automaticamente i nostri progressi, ha una *history* locale del progetto e di ogni file, oltre al supporto per VCS come GIT.
- Ci permette di importare librerie standard con un sistema di plugin per l'sdk
- Supporta lo sviluppo di interfacce sia grafico che testuale, fornendo una *preview* istantanea
- Ci fornisce warning su best practices e potenziali problemi anche senza buildare
- Ha un emulatore di telefoni, tablet ed altri devices android

Dopo aver completato la build del nostro progetto, avremo la schermata principale:



Sulla sinistra, la struttura del progetto è mostrata in un formato ad albero. E' un formato semplificato per lo sviluppo, che ignora file di build e autogenerati, o di librerie. Cliccando su Android e selezionando la modalità Project, si potranno vedere tutti i file del progetto.

E' possibile copiare-incollare da windows direttamente in una cartella di questo albero, come se fosse un'estensione di Esplora Risorse (Windows Explorer).

Ecco come è diviso il nostro progetto:

- **app** contiene i file del codice del nostro progetto. Il suo contenuto sarà il contenuto del pacchetto che pubblicheremo.
 - **manifests** contiene i file di configurazione necessari al sistema operativo Android
 - **java** include tutti i file e le classi dove è scritto il nostro codice
 - **res** contiene le risorse come immagini, testi e files.
- **Gradle Scripts** è dove vengono contenuti i file responsabili della configurazione del progetto.

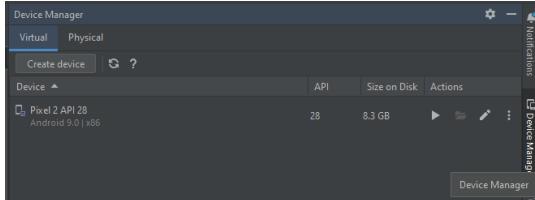
In particolare in Gradle Scripts ci sono 2 file build.gradle: *Project* e *Module*

- **Project** sono le impostazioni globali del progetto
- **Module** imposta un singolo modulo e ogni modulo ha uno di questi file

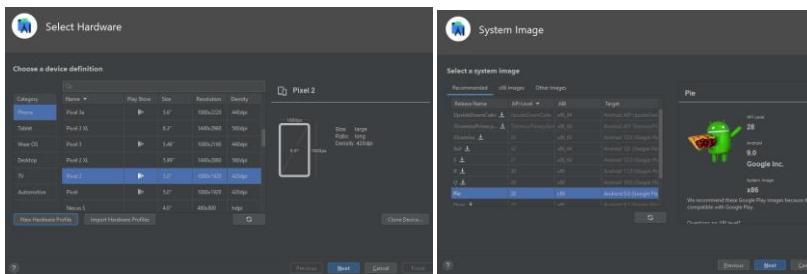
I file *Module* sono di particolare interesse, sarà qui che inseriremo le dipendenze e le compatibilità.

3. Emulatore

Per creare un device emulato, aprire la tab laterale *Device Manager*



Qui avremo i nostri device virtuali. *Create Device* comincerà il wizard per crearne uno nuovo:



Le due schermate ci permetteranno di definire un Hardware e un software sul quale testare le nostre app.

E' buona norma avere almeno due o tre device che includono versioni android differenti:

- per la minima versione di Android supportata dalla nostra app
- per l'ultima versione
- uno per una versione intermedia molto diffusa.

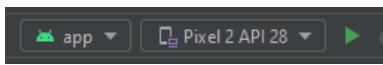
Per l'hardware è possibile creare il device ex-novo, o cercare nella lista uno dei device precostruiti.

Per ora creiamo un device **Pixel 2 – Android 9**, attualmente una buona via di mezzo tra compatibilità e prestazioni.

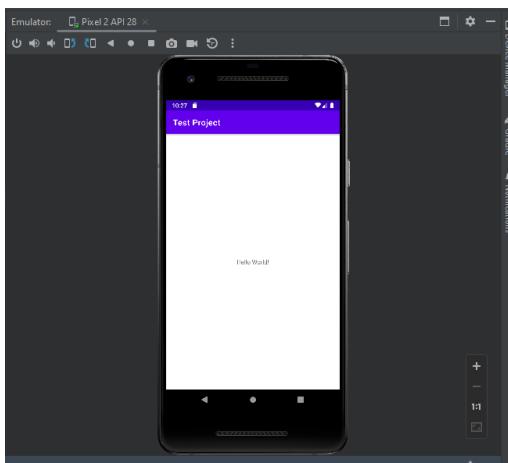
Saremo ora in grado di utilizzare il device emulato direttamente nell'ide tramite la tab laterale *Emulator*



Ora possiamo lanciare il nostro progetto sull'emulatore e vederlo funzionare dal vivo.



Nella barra orizzontale degli strumenti, assicurarsi che siano selezionati il modulo *app* e l'emulatore che abbiamo creato: spingi il pulsante play (freccia verde) per cominciare la build e installare l'app sull'emulatore.



B.1- Nozioni di base di layout

Android utilizza dei widget chiamati **VIEWS** per visualizzare l'interfaccia. Tutti i widget sono sottoclassi di una classe antenata comune, chiamata appunto **View**.

Tutte le view devono avere width e height definite affinchè siano visibili, e possiedono un campo id che permette di identificarle nell'app.

Una importante sottoclasse di View è **ViewGroup**, che è l'antenato di tutte le classi Layout: classi invisibili il cui ruolo è contenere e ordinare delle View.

Nello sviluppo di un'app è possibile creare layouts e views in 3 maniere:

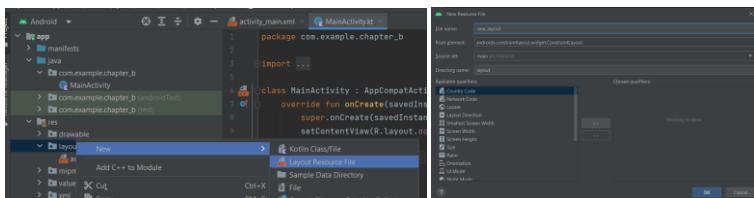
- Programmaticamente, utilizzando i loro costruttori
- Con dei file risorse XML
- Con una libreria dichiarativa chiamata Jetpack Compose

In questa guida utilizzeremo il **formato XML** per definire i layout, in quanto attualmente il più utilizzato.

Creare un file di layout

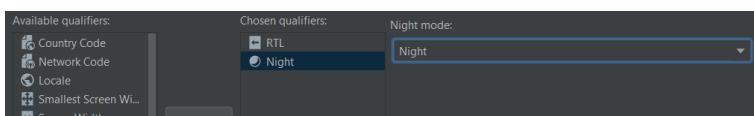
I file di layout sono contenuti all'interno della directory **res/layout**. Usando il menu contestuale (tasto destro) è possibile crearne di nuovi. Il popup risultante ci chiederà il nome e la classe **layout** di base, che di default è **ConstraintLayout**

Il nome del file layout deve essere scritto in **snake case** e segue la convenzione **tipo_nome**



Nella parte **qualifiers** è possibile inserire dei parametri che restringeranno l'utilizzo di quel layout per specifici dispositivi.

Esempio: solo dispositivi in modalità Night e che hanno il testo impostato a RightToLeft



Constraint Layout

Il Constraint Layout è un ViewGroup che lavora sul concetto di ordinare le view che contiene assegnando loro dei valori **constraint** per allineare ogni elemento rispetto a un altro. Le view contenute saranno tutte sullo stesso livello di parentela (**flat layout**), riducendo notevolmente la complessità rispetto a ordinamenti innestati.

Ad esempio, una pagina può contenere un titolo allineato al bordo in alto a sinistra del layout che lo contiene. Nella stessa pagina, un paragrafo può essere allineato al bordo di sinistra del layout, e verticalmente al di sotto del titolo.

DP, margin e padding

DP è l'unità di misura delle dimensioni delle View. Significa Density Indipendent Pixels, perché si ridimensiona automaticamente in base alla densità dello schermo su cui viene visualizzata la view, permettendo di poter costruire solo 1 layout per diverse grandezze di device.

Padding è la distanza che la view interpone tra i suoi bordi e il suo contenuto.

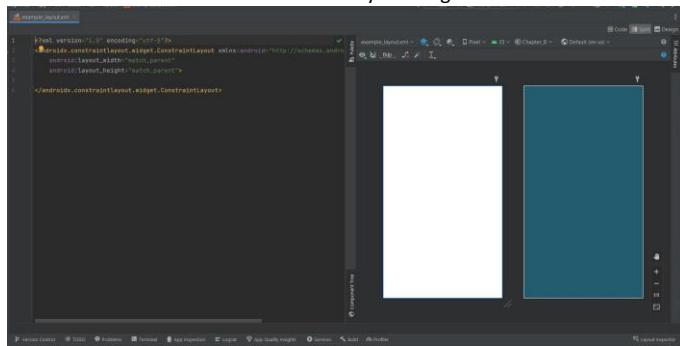
Margin è la distanza che la view interpone tra i suoi bordi e quelli delle view vicine. Interagisce solo con le view che hanno il suo stesso grado di parentela o maggiore. Nel constraint layout, questa distanza è applicata ai constraint che identificano dove è posizionata la view.

Ad esempio, utilizzando il titolo di prima, un Margin di 24dp creerà una distanza tra il titolo ed il bordo alto-sinistra a cui è allineato.

Editor Grafico e Testuale

Doc: <https://developer.android.com/studio/write/layout-editor>

Android Studio ci fornisce un editor di Layout sia grafico che testuale.



Utilizzando i pulsanti *Code*, *Split*, *Design* è possibile cambiare impostazione dell'editor lavorando sempre sullo stesso file.

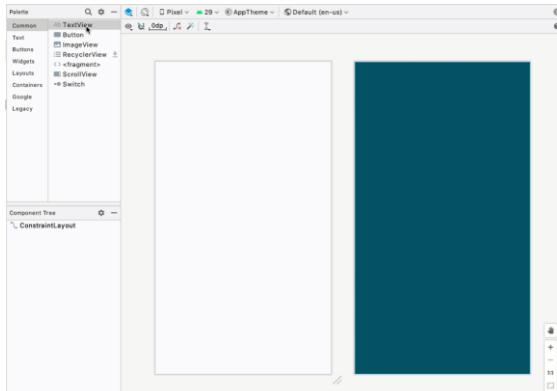
Design è utilizzato più spesso per dare un'impostazione generale, mentre la parte *Text* permette il fine-tuning e il lavoro su layout complessi.

Aggiungere elementi: Design

Selezionare l'impostazione *Design*.

Trascinare un oggetto *TextView* dalla tavolozza in alto a sinistra dell'area di progettazione nell'editor di layout e rilasciarlo.

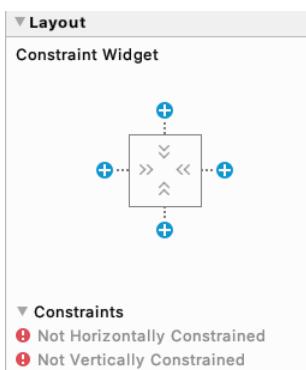
TextView è un widget il cui scopo è mostrare testo.



Una TextView sarà aggiunta, notare un punto esclamativo rosso nel Component tree.

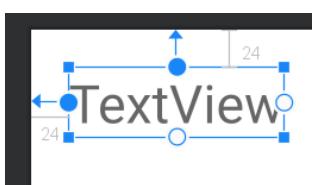
Posizionando il puntatore sul punto esclamativo si visualizzerà un messaggio di avviso dicendo che la view non ha dei *constraint*, quindi il layout non sa dove ordinarla.

Per creare dei constraint, nel pannello a destra Layout > Constraint Widget

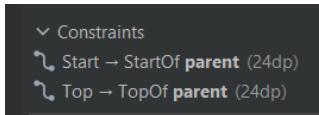


Cliccare sul + in alto, e selezionare 24 nel menu a tendina che comparirà. Ripetere per il + sinistro

Questo creerà due constraint che allineeranno il testo in alto a sinistra, con un *margin* di 24dp dall'elemento a cui ci si sta allineando.

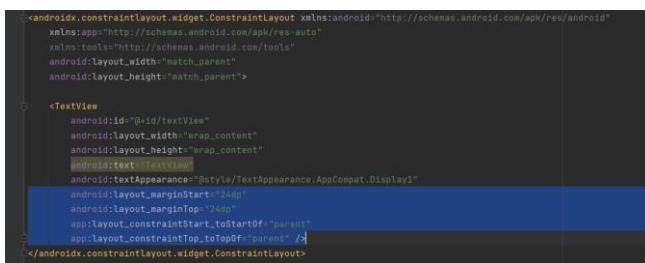


Questi constraint verranno creati allineandosi al padre della view, il ConstraintLayout



Ogni view ha 4 lati: Start, End, Top, Bottom. Ognuno di questi lati puo' essere allineato al lato di un'altra view.

Selezionando *Code* nell'editor potremo vedere come questi dati vengono convertiti a codice, creando margini e constraint.

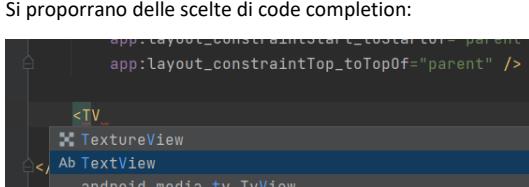


E' anche possibile creare constraint selezionando una view e trascinando i punti vuoti verso il bordo di altre view.

Aggiungere elementi: Code

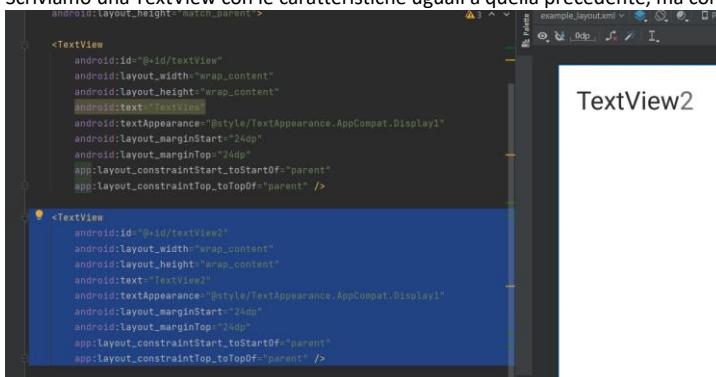
Utilizzando l'editor *Split*, andiamo ad aggiungere un altro campo di testo.

Sotto la textView che già abbiamo, aggiungere delle righe e digitare <TV>



Cliccando il suggerimento il nostro widget verrà parzialmente precompilato, con i campi width e height.

Scriviamo una TextView con le caratteristiche uguali a quella precedente, ma con id e testo diverso:



La nuova TV, textView2, è solo parzialmente visibile (il numero 2 grigio nella preview): è posizionata esattamente come quella creata in precedenza. Ha infatti gli stessi constraint e gli stessi margin.

Aggiungiamo un background verde e modifichiamo il testo in Avocado:

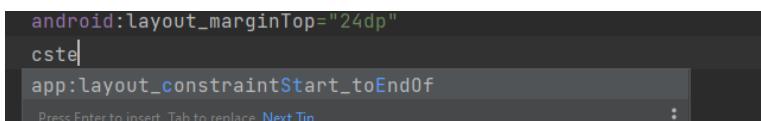


Ora vediamo la seconda textView ma non la prima. Questo perché quando più elementi si sovrappongono, vengono ordinati in base a come sono dichiarati nel file XML: textView è scritta prima, quindi andrà sotto textView2.

Ora spostiamo textView2 sulla destra di textView in modo da vederle entrambe.

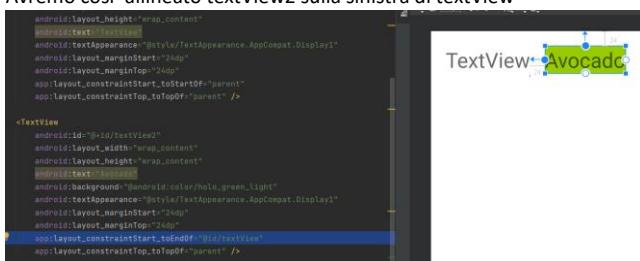
Sostituiamo il constraint *layout_constraintStart_toStartOf* con un constraint che allinea lo *start* di textView2 all'*end* di textView. Il constraint da utilizzare sarà *layout_constraintStart_toEndOf*.

Puoi utilizzare le lettere iniziali della definizione *Constraint Start To End* e premere invio per scriverlo rapidamente.



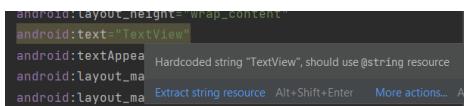
Infine, inseriamo l'*id* *textView* per specificare all'*end* di quale view allinearsi.

Avremo così allineato textView2 sulla sinistra di textView



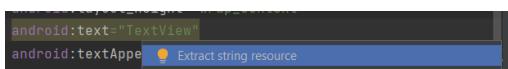
Best Practice: no hardcoded

L'IDE ci sta sgridando con un warning, dicendo che le nostre stringhe sono hardcodate:

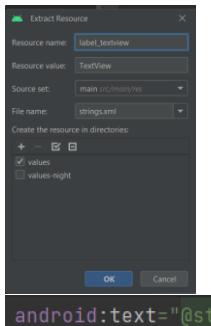


È buona norma non hardcodare risorse in generale, ma nel caso specifico delle stringhe è necessario: avere un file con tutte le stringhe serve per poter tradurre la nostra app in altre lingue.

Selezioniamo la linea di testo con il warning e attiviamo l'assistente (*alt-invio*) > extract string resource



Diamo un nome alla nostra nuova risorsa e salviamo



`android:text="@string/label_textview"`

La stringa sarà salvata e il nostro layout automaticamente aggiornato con il riferimento.

Controlliamo che tutto sia a posto andando a leggere il file di stringhe: nel codice, ctrl-click (o middle mouse btn) sul nome della stringa.

Questo ci porterà a `res/values/strings.xml` dove vengono salvate tutte le stringhe:

```
example_layout.xml x strings.xml x
Edit translations for all locales in the translations editor.

<resources>
    <string name="app_name">Chapter_B</string>
    <string name="label_textview">TextView</string>
</resources>
```

E' buona norma non hardcodare valori che possano essere contenuti in file di risorse – come stringhe, dimensioni e colori.

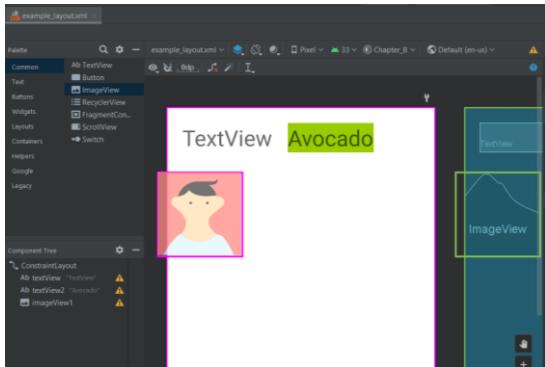
Larghezza e Altezza widget

`layout_width` e `layout_height` possono essere definiti in più maniere:

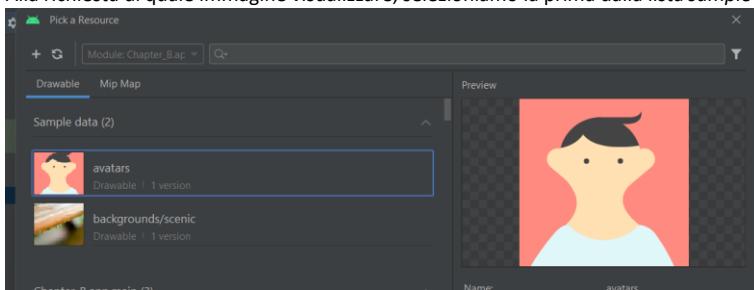
- *Valore fisso espresso in dp*: ad esempio `100dp`
- *wrap_content*: la view si ridimensionerà al minimo indispensabile per visualizzare il contenuto
- *match_parent*: la dimensione sarà pari alla stessa dimensione del layout contenitore
- *0dp*: questa opzione, disponibile solo per view all'interno di un Constraint Layout, sostituisce *match_parent*. La view si ridimensionerà per raggiungere la massima ampiezza consentita dai suoi constraint, che devono entrambi essere settati per quella direzione – verticale o orizzontale.

Immagini

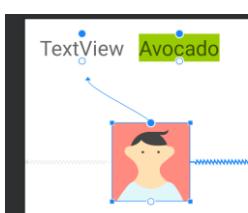
Tramite l'editor *Design*, inseriamo una `ImageView`:



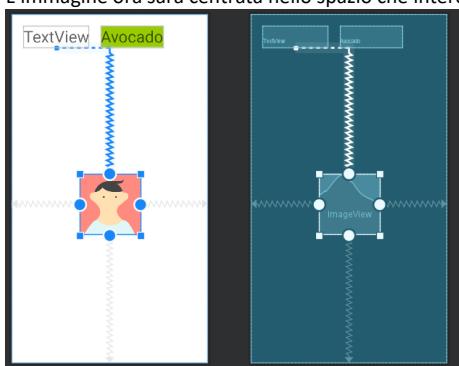
Alla richiesta di quale immagine visualizzare, selezioniamo la prima dalla lista *sample data*



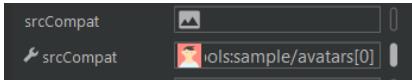
Definiamo i constraint dell'immagine legando start, end e bottom al layout. Infine trasciniamo top al bottom di textView



L'immagine ora sarà centrata nello spazio che intercorre tra il layout e il bottom di textView.



La nostra immagine funziona, ma in realtà avendo selezionato un'immagine dalla galleria "sample data" l'editor ha inserito un'immagine con una modalità che la rende visibile solo in fase di progettazione. Se lanciamo l'app con la freccia verde, l'immagine non comparirà nel nostro device.



La chiave inglese mostra i campi le cui modifiche sono visibili solo in progettazione.

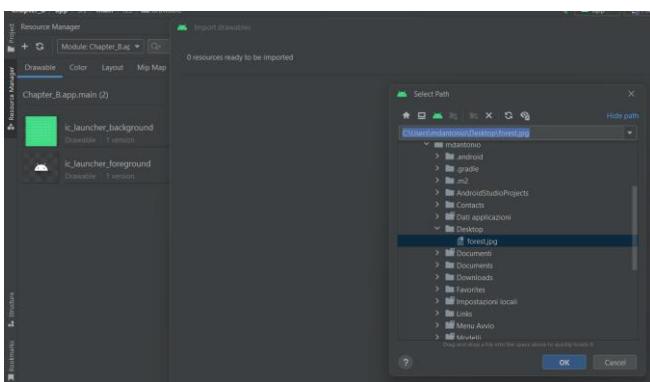
Aggiungiamo un nuovo obiettivo:

questa immagine deve essere un banner che mostra un paesaggio, e deve essere visibile sotto il testo dell'esempio precedente.

Passiamo alla modalità *Split*.

Aggiungiamo l'immagine da visualizzare: importiamo un file .jpg nel progetto.

Apri Resource Manager > pulsante + > Import Drawables. Seleziona il file dal tuo pc e accetta le opzioni di default nei menu successivi.



Chiudi e riapri resource manager per aggiornarlo (bug del'IDE). Ora la nostra immagine sarà visibile nelle risorse. È stata inserita nella dir **res/drawable** dove vengono mantenute la maggior parte delle immagini dell'app.

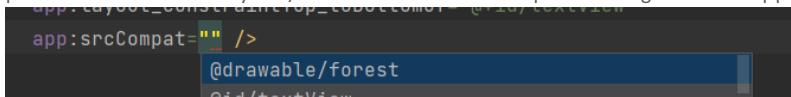
E' anche possibile inserire un file da windows con copia-incolla.

ctrl-c sul file in windows. Su android studio, selezionare la directory di destinazione dall'albero Project, poi ctrl-v.

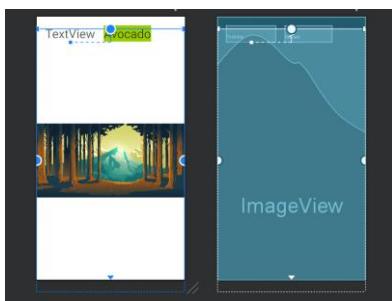
Ora andiamo a inserire l'immagine effettiva nell'ImageView

Sostituiamo la riga tools:srcCompat con app:srcCompat, nel valore selezionando il nome del file che abbiamo importato.

Il file immagine viene riconosciuto come *drawable* nelle risorse del progetto quando presente nella directory *res/drawable*. Non tutti i tipi di immagini sono supportati.



Ora verrà visualizzata la nostra immagine, ma c'è un problema. Essendo *imageView* impostata con *height* e *width wrap_content*, si andrà a ridimensionare in base alle dimensioni dell'immagine che gli abbiamo fornito. In questo caso, l'immagine è più grande della risoluzione dello schermo, e *imageView* diventa troppo grande:

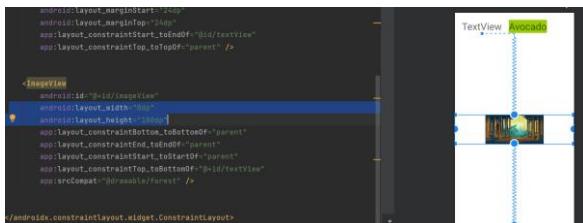


Sebbene di base *imageView* mostri l'immagine scalandola per le dimensioni dello schermo, la dimensione del widget è pari alle dimensioni dell'immagine originale.

Modifichiamo quindi la sua altezza in 100dp e larghezza in 0dp: questo metterà un limite a quanto può espandersi.

ImageView cerca sempre di mantenere le corrette proporzioni dell'immagine

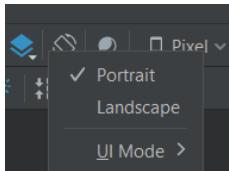
il risultato sarà poco elegante:



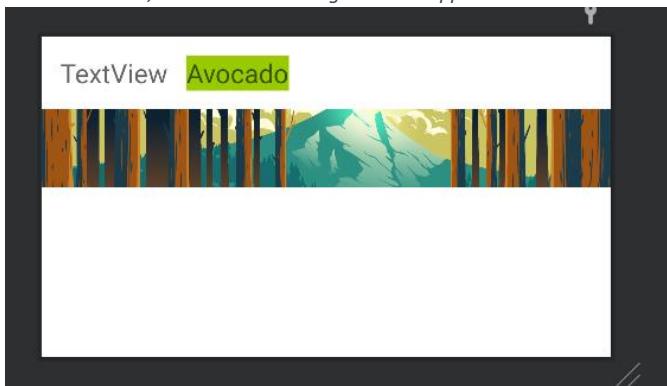
Per ottenere un banner, aggiungiamo il parametro *scaleType*="centerCrop". Inoltre rimuoviamo *constraint_bottom* così da rimuovere il centraggio e legarla solo a *textView* verso top. Infine, aggiungiamo *marginTop* 24dp.



Abbiamo raggiunto il nostro obiettivo, ma come ultimo controllo decidiamo di dare un'occhiata a cosa succede quando l'utente ruota il device in posizione landscape:



Sebbene diverso, il nostro banner è ugualmente apprezzabile.

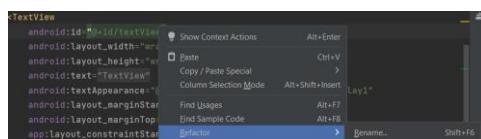


Best Practice: resource naming

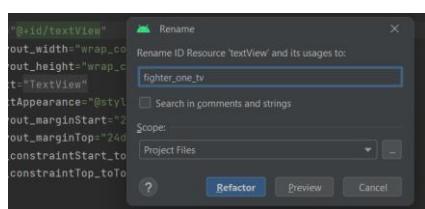
Aggiungiamo un obiettivo:

Il titolo deve rappresentare lo scontro mortale tra due frutti – ed esempio: **Mela vs Banana**

Modifichiamo l'id della prima textView per riflettere la sua nuova funzionalità. L'id è un campo che è utilizzato come riferimento da altre view – nel nostro esempio, nei constraint di textView2 e imageView. Potremmo modificare tutto a mano, ma Android Studio ci viene in soccorso con il *refactor > rename*



Selezionando l'opzione – ma è più pratico *shift-f6* – è possibile rinominare il campo e aggiornare tutti i punti del codice che ne fanno riferimento.



L'ID è una risorsa che identifica la view nell'app, e deve essere specifico e parlante. Una convenzione è: cosa contiene _ una identificazione tra elementi simili _ tipo di widget. Android non richiede ID univoci.

Così anche solo leggendo l'id possiamo avere un'idea di che widget sia.

Rinominiamo anche textView2 in fighter_two_tv, e imageView in fight_bg_iv.

Facciamo delle modifiche ai nostri titoli:

- Rinomina anche textView2 in fighter_two_tv, e imageView in fight_bg_iv.
- fighter_one_tv: cambia il testo in Apple e il bg in Holo red light



- Aggiungi tra fighter_one e fighter_two una nuova TV *fight_versus_tv*, che conterrà il testo "VS".
Imposta il constraint orizzontale all'end di fighter_one_tv
Imposta il constraint verticale *baseline* al *baseline* di fighter_one_tv.

```
<TextView
    android:id="@+id/fight_versus_tv"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/label_versus"
    android:textColor="@color/black"
    android:layout_marginStart="24dp"
    android:layout_marginTop="24dp"
    app:layout_constraintStart_toEndOf="@+id/fighter_one_tv"
    app:layout_constraintBaseline_toBaselineOf="@+id/fighter_one_tv"
    android:textAppearance="@style/TextAppearance.AppCompat.Large" />
```

layout_constraintBaseline_toBaselineOf allineerà verticalmente la base del testo di due widget.

- Modifica *fighter_two_tv* per legare il suo start all'end di *fight_versus_tv*

```
<TextView
    android:id="@+id/fighter_two_tv"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/label_avocado"
    android:textColor="@color/black"
    android:background="@color/holo_green_light"
    android:textAppearance="@style/TextAppearance.AppCompat.Display1"
    android:layout_marginStart="24dp"
    android:layout_marginTop="24dp"
    app:layout_constraintStart_toEndOf="@+id/fight_versus_tv"
    app:layout_constraintTop_toTopOf="parent" />
```

Risultato:



Chains

Finora i campi di testo sono allineati sulla sinistra dello schermo: se volessimo centrare *entrambi* i campi, mantenendo comunque l'ordine di uno prima dell'altro, incomberemmo in alcuni limiti: uno o entrambi, sarebbero sempre decentrati.

Le *Chains* risolvono questo problema!

Una Chain è una serie di widget legata strettamente uno all'altra, con un ordinamento in comune.

Decidiamo di centrare i tre campi di testo nello spazio orizzontale: costruiamo una Chain.

Nella parte grafica dell'editor *Split*, trascina e seleziona i tre campi di testo. Poi tasto destro > chains > create horizontal



Dando un'occhiata al codice, questo comando modifica *tutti i widget della catena* impostando *entrambi i constraint start e end* al widget precedente e successivo.

```
<TextView
    android:id="@+id/fight_versus_tv"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="24dp"
    android:layout_marginEnd="24dp"
    android:text="vs"
    android:textSize="24sp"
    android:fontFamily="sans-serif-normal"
    android:textColor="#000000"/>
<TextView
    android:id="@+id/fighter_one_tv"
    android:layout_constraintStart_toEndOf="@+id/fighter_vs_tv"
    android:layout_constraintEnd_toStartOf="@+id/fighter_two_tv"
    android:layout_constraintBaseline_toBaselineOf="@+id/fighter_one_tv"
    android:layout_constraintHorizontal_bias="0.5"/>
/>
```

Il risultato è promettente, ma troppo separato nello spazio:

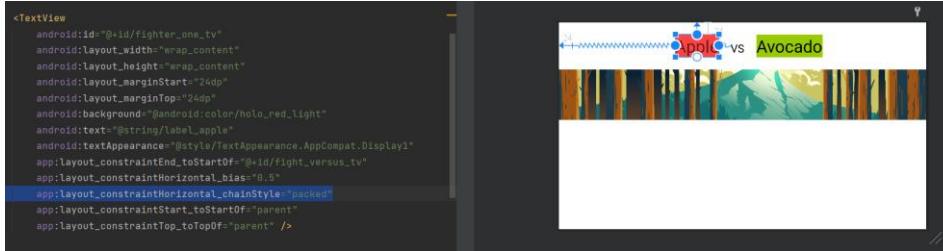


Modificando lo stile della chain possiamo scegliere tra alcune opzioni per come vengono organizzati gli spazi. E' possibile utilizzare sempre lo stesso menu contestuale per farlo, ma noi andremo nell'editor di testo.

Il widget responsabile del setup di una chain è il primo della chain nell'xml.

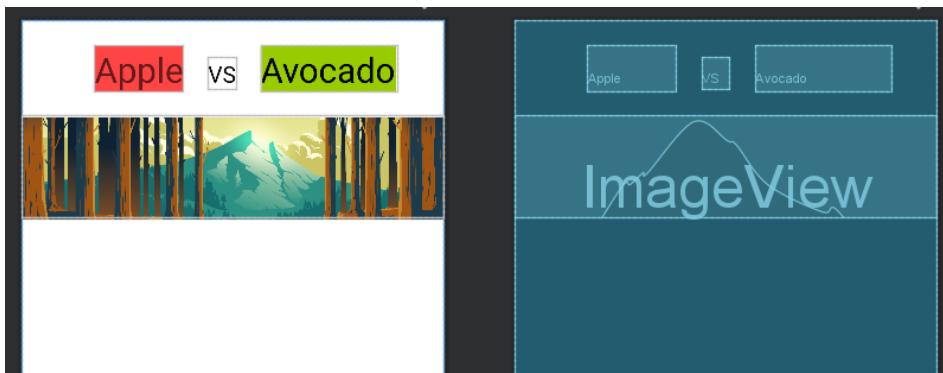
Nel nostro caso, `fighter_one_tv` – è il primo, i cui constraint start sono attaccati a *parent*. Andiamo ad aggiungere un `chainStyle`:

```
app:layout_constraintHorizontal_chainStyle="packed"
```



Le chains possono essere ordinate in : packed, spread, spread_inside

Ruotando il layout in modalità *portrait*, vedremo che le view si riorganizzano in base allo spazio disponibile.



Per ora tutto è un po' spostato a sinistra: questo perché Apple e Avocado sono parole con larghezza diversa, e le nostre TV sono impostate con larghezza `wrap_content`.

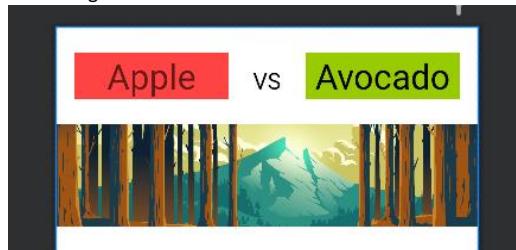
Facciamo queste modifiche a `fighter_one_tv` e `fighter_two_tv`:

- Rimuovi `marginStart`
- Aggiungi:
`android:minWidth="150dp"`
`android:textAlignment="center"`

E questa modifica a `fight_versus_tv`, per separarla dalle altre 2:

- Aggiungi
`android:layout_marginEnd="24dp"`

Ora i widget sono centrati



Esercizio: Postcard o Meme

Sviluppa un'app con le seguenti caratteristiche:

- Una singola pagina, con un'immagine di sfondo a schermo intero
 - Crea un nuovo progetto con una *Empty Activity* e scrivi l'interfaccia nel file layout *activity_main.xml*.
- Deve contenere almeno due campi di testo, il cui contenuto deve essere facilmente leggibile sullo sfondo
 - *TextView* ha un parametro *textColor*. Android fornisce alcuni colori di sistema con riferimento *@color/...* e *@android:color/...*
 - Uno dei due campi deve essere scritto in *Italic*
- Deve contenere almeno una chain
- No hardcoding: né di stringhe, né di dimensioni (dp)
- ID parlanti

B.2 – ViewBinding e Azioni

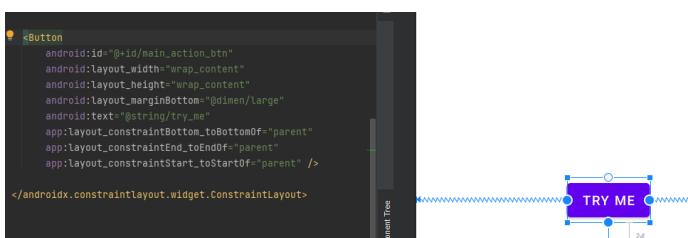
1. Button

Un'app normalmente richiede qualche tipo di interazione con l'utente: click, testi e gestures sono i più comuni. Android lavora con un sistema ad eventi che permette di interagire con queste attività.

Button è un widget che crea un pulsante già pronto con colori, animazioni e un fondamento di interazione con l'utente.

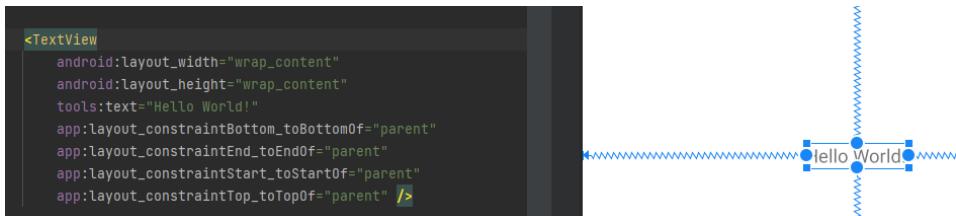
In un'app con un empty layout, creiamo un *Button* in `activity_main`.

Allineiamolo al centro in basso e cambiamo il testo in `Try me`. Cambiamo infine l'id in `main_action_btn`



Lanciando l'app e provandolo, vediamo che ha già alcune animazioni, ma non produce alcun effetto.

Infine modifichiamo il campo di testo in maniera tale che il contenuto "Hello world" sia visibile sono in progettazione:



2. Activity e Context

L'Activity è la classe del framework Android che ci permette di interagire con il layout creato.

Insieme a **Fragment** è una delle classi fondamentali per la realizzazione di un'app, ed è una delle poche classi a implementare l'interfaccia di sistema *LifecycleOwner*.

Una Activity è il punto di ingresso per l'interazione con l'utente. Rappresenta un flusso di operazioni che l'utente può compiere.

Aprendo la classe `MainActivity.kt` si nota che ha già un metodo `onCreate` prepopolato, che verrà chiamato a runtime dal sistema nel momento in cui il nostro layout è stato costruito ed è pronto per essere utilizzato.

Activity è un discendente della classe **Context** anch'essa fondamentale: fornisce o sarà richiesta nella maggior parte delle interazioni con i metodi provvisti dal framework.

Tutti questi metodi sono disponibili nelle classi discendenti della classe Context: **Activity, Fragment e View**.

3. View Binding

Ora dobbiamo scrivere il *wiring*, ovvero come il programma interagisce con il widget. Per farlo dobbiamo avere un riferimento all'istanza di quel widget che verrà creata per noi tramite l'xml.

Già di base MainActivity implementa il layout *activity_main* tramite il metodo *setContentView*. Ora dobbiamo prendere un riferimento all'istanza del widget sul quale intendiamo lavorare.

Ci sono due modi per farlo:

- Utilizzando il metodo [*findViewById\(R.id.widget_id\)*](#)
- Utilizzando il View Binding

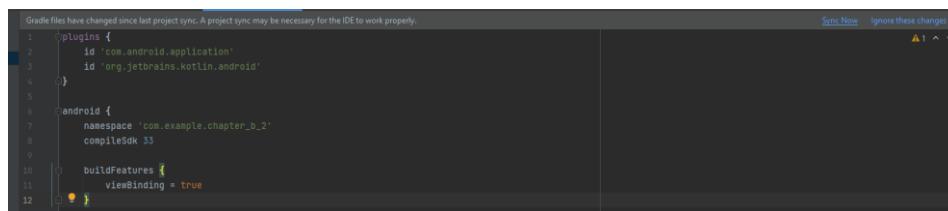
In questa guida utilizzeremo il View Binding.

View Binding è il processo di ottenere un riferimento a una *binding class* che rappresenta uno specifico layout. Questa classe conterrà riferimenti a tutte le view che hanno un id all'interno del layout.

Andiamo a modificare il file *build.gradle (Module)* aggiungendo nella sezione *android*:

```
buildFeatures {  
    viewBinding = true  
}
```

Seguiamo il suggerimento dell'IDE, che ci sta avvisando di effettuare un *gradle sync* dopo aver modificato il file.



Dobbiamo inoltre modificare la nostra activity per lavorare con il ViewBinding:

```
class MainActivity : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
  
        val binding = ActivityMainBinding.inflate(layoutInflater)  
        setContentView(binding.root)  
    }  
}
```

La classe *ActivityMainBinding* è stata generata dalla libreria ViewBinding che abbiamo appena impostato. Il nome della classe è ottenuto trasformando il nome del file layout xml in Camel Case e aggiungendo la parola *Binding*.

4. Update interfaccia

Vogliamo interagire con il pulsante, quindi andremo a cercarlo nella binding class.

Lo troveremo come membro, con nome il suo id (trasformato in Pascal case): nel nostro caso

mainActionButton

```
val binding = ActivityMainBinding.inflate(layoutInflater)
setContentView(binding.root)

binding.mainActionBtn
```

Modifichiamo ora *programmaticamente* il testo del pulsante in **Activate**

```
binding.mainActionBtn.text = "Activate"
```

Lanciando l'app ora vedremo che il testo del pulsante è modificato.

5. Acquisizione risorse

L'IDE ci informa che non dovremmo hardcodare risorse. Ancora una volta utilizziamo la funzione “extract string resource” disponibile nel menu contestuale e vediamo il risultato:

```
binding.mainActionBtn.text = getString(R.string.label_activate)
```

Context.getString(int) è il metodo fornito dal framework per ottenere risorse stringa. Questo metodo ci è disponibile perché stiamo lavorando all'interno di una classe Activity, discendente di Context.

Inizialmente tutte le risorse erano disponibili grazie a metodi simili, ma nel corso delle evoluzioni di android si è reso necessario trovare una soluzione per le esigenze di ogni risorsa, rendendo non-standard l'acquisizione. Ecco alcuni esempi:

```
val string: String = getString(R.string.label_activate)
val color: Int = getColor(R.color.black) // provides an id, not a Color()
val drawable: Drawable? = AppCompatResources.getDrawable(context, R.drawable.ic_launcher_foreground)
val dimension: Float = resources.getDimension(R.dimen.large)
```

6. Listeners

Button ci mette a disposizione l'interfaccia *OnClickListener*, che sarà attivata dal sistema al momento dell'azione dell'utente. All'interno del metodo onClick dell'interfaccia potremo scrivere come dovrà comportarsi l'app.

Aggiungiamo quindi un onClickListener al pulsante:

```
binding.mainActionBtn.setOnClickListener(
    object : View.OnClickListener {
        override fun onClick(p0: View?) {
            Log.d("onClick", "Clicked!")
        }
    }
)
```

Cliccando sul pulsante dopo aver lanciato l'app, ora stamperemo un Log

```
W/ple.chapter_b_: Accessing hi
D/onClick: Clicked!
```

Andiamo ora a modificare l'interfaccia tramite il click, modificando il testo del nostro messaggio.

Prima di tutto aggiungi un ID alla textview presente nell'xml, in maniera che compaia nel binding:



Poi impostala con un testo iniziale:

```
binding.mainLabelTv.text = getString(R.string.label_activation_ready)
```

All'interno del `onClickListener`, aggiungi questa riga dopo `Log.d`:

```
binding.mainLabelTv.text = "Activation complete!"
```

Ora cliccando il pulsante, il testo del messaggio verrà impostato in tempo reale.

Certo, il nostro codice è piuttosto complesso per una funzionalità così semplice:

```
binding.mainActionBtn.setOnClickListener(
    object : View.OnClickListener {
        override fun onClick(p0: View?) {
            Log.d("onClick", "Clicked!")
            binding.mainLabelTv.text = "Activation complete!"
        }
    }
)
```

L'IDE ci viene ancora in aiuto, sfruttando le funzionalità *lambda* di Kotlin per accorciare notevolmente la definizione e rimuovere gran parte del boiler-code. Seguiamo il suggerimento del warning e utilizziamo la funzionalità *convert to lambda*

```
binding.mainActionBtn.setOnClickListener { it: View!
    Log.d("onClick", "Clicked!")
    binding.mainLabelTv.text = "Activation complete!"
}
```

Questa forma abbreviata è lo standard.

7. Best Practice: Design for readability

Quando scriviamo codice, quasi mai lo scriviamo solo per noi stessi. Anche solo rileggendo cose che abbiamo scritto a distanza di tempo, spesso ci si trova di fronte a sensazioni estranianti del tipo "ma questa cosa l'ho scritta io??". E' necessario fermarsi un attimo e ragionare sullo "stile" di quello che scriviamo, in maniera da rendere il nostro codice chiaro e future-proof.

E' buona norma scrivere codice parlante, comprensibile dagli altri e facilmente manutenibile

Kotlin ci aiuta molto in questo campo, come abbiamo già visto nell'esempio precedente.

E' un linguaggio sviluppato di base su questi concetti.

Andiamo a fare delle modifiche al nostro esempio, che per ora vede alcune ripetizioni:

```
binding.mainActionBtn.text = getString(R.string.label_activate)
binding.mainActionBtn.setOnClickListener { it: View!
    Log.d("onClick", "Clicked!")
    binding.mainLabelTv.text = "Activation complete!"
}
```

Per prima cosa estraiamo le stringhe hardcodate.

Poi andiamo a lavorare su *binding*, usando il metodo *run* di kotlin per creare un *this* che lo rappresenti. Così potremo rimuovere la notevole ripetizione di *binding*

```
binding.run { this: ActivityMainBinding
    mainLabelTv.text = getString(R.string.label_activation_ready)
    mainActionBtn.text = getString(R.string.label_activate)
    mainActionBtn.setOnClickListener { it: View!
        Log.d("tag: onClick", "msg: Clicked!")
        mainLabelTv.text = getString(R.string.label_activation_complete)
    }
}
```

Il codice è già molto leggibile, ma la ripetizione di *mainActionBtn* può essere fonte di errore nel caso si dovesse rilavorare il codice. Andiamo quindi a utilizzare *apply* di Kotlin per raggruppare gli interventi che facciamo a questo widget:

```
binding.run { this: ActivityMainBinding
    mainLabelTv.text = getString(R.string.label_activation_ready)
    mainActionBtn.apply { this: Button
        text = getString(R.string.label_activate)
        setOnClickListener { it: View!
            Log.d("tag: onClick", "msg: Clicked!")
            mainLabelTv.text = getString(R.string.label_activation_complete)
        }
    }
}
```

Il risultato non è solo più compatto, ma anche logicamente più schematico.

Cerca sempre di mantenere un buon equilibrio tra compattezza e leggibilità.

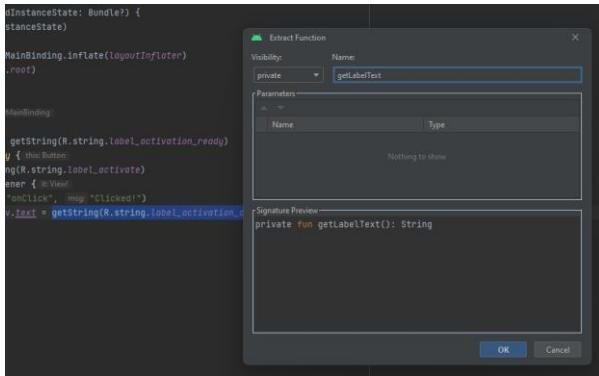
8. Condizionali

Aggiungiamo la possibilità di resettare la nostra app, in modo che premere il pulsante alterni i due testi: per fare questo dobbiamo lavorare sullo stato attuale dell'interfaccia.

Cominciamo estraendo il codice che modifica il testo della *TextView* in una funzione. Dopo aver selezionato questo getter:

```
Log.d("tag: onClick", "msg: Clicked!")
mainLabelTv.text = getString(R.string.label_activation_complete)
```

Spingi *control-alt-m*, attivando la funzionalità *extract function*

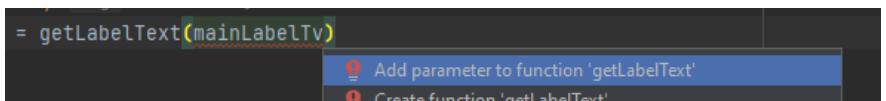


Modifica il nome della funzione in `getLabelText` e premi ok.

```
private fun getLabelText() = getString(R.string.label_activation_complete)
```

Ora abbiamo bisogno di un altro parametro, che ci permetterà di capire lo stato dell'interfaccia: scegiamo di passare il campo di testo alla funzione, così ne potremo valutare il contenuto.

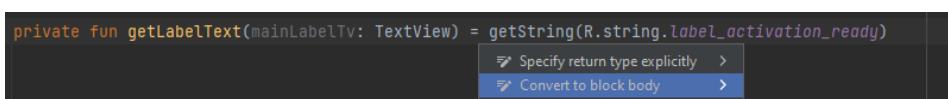
Inseriamo `mainLabelTv` tra le parentesi della funzione e utilizziamo l'assistente dell'IDE per aggiungerlo come parametro ad essa:



```
private fun getLabelText(mainLabelTv: TextView) = getString(R.string.label_activation_complete)
```

Ora la nostra funzione dovrà emettere un risultato in base a quello che contiene attualmente il campo di testo.

Modifichiamola posizionando il cursore sul carattere `=` e attivando l'assistente



Dopo averlo fatto, aggiungiamo i due possibili risultati

```
private fun getLabelText(mainLabelTv: TextView): String {
    val activatedString = getString(R.string.label_activation_complete)
    val readyString = getString(R.string.label_activation_ready)
}
```

Ora analizziamo `mainLabelTv` per capire quale sia il suo testo attuale. Per prima cosa otteniamo il testo:

```
val currentText = mainLabelTv.text
```

Poi analizziamolo con lo scopo di invertirlo: se è "activated", rimetterlo in "ready", e viceversa.

```

if (currentText == activatedString) return readyString
else if (currentText == readyString) return activatedString
else return "Error! I don't know this text."

```

Abbiamo anche aggiunto un caso di errore nel quale non riusciamo a capire lo stato attuale.

In conclusione, ecco la nostra funzione:

```

    setOnClickListener { v: View ->
        Log.d("MyApp", "onClick", mainLabelTv.text = getLabelText(mainLabelTv))
    }
}

private fun getLabelText(mainLabelTv: TextView): String {
    val activatedString = getString(R.string.label_activation_complete)
    val readyString = getString(R.string.label_activation_ready)
    val currentText = mainLabelTv.text

    if (currentText == activatedString) return readyString
    else if (currentText == readyString) return activatedString
    else return "Error! I don't know this text."
}

```

Ora il testo si alternerà tra le due label.

Aggiustiamo ora il codice per renderlo più Kotlinoso, utilizzando il costrutto `when`. Possiamo utilizzare il suggerimento che ci dà l'IDE, attivando l'assistente:

```

if (currentText == activatedString) {
    readyString -> return readyString
    activatedString -> return activatedString
    else -> return "Error! I don't know this text."
}

```

The screenshot shows an IDE interface with a tooltip suggesting to replace the 'if' statement with a 'when' expression. The tooltip contains three options: 'Lift return out of 'if'', 'Replace 'if' with 'when'', and 'Add brace'. The 'Replace 'if' with 'when'' option is highlighted.

Continuando a seguire i warning dell'IDE, e sostituendo le variabili con dichiarazioni inline, abbiamo:

```

private fun getLabelText(mainLabelTv: TextView): String {
    val activatedString = getString(R.string.label_activation_complete)
    val readyString = getString(R.string.label_activation_ready)

    return when (mainLabelTv.text) {
        activatedString -> readyString
        readyString -> activatedString
        else -> "Error! I don't know this text."
    }
}

```

9. Esercizio: Lancio Dadi

Sviluppa un'app con le seguenti caratteristiche:

- L'app mostra un pulsante e un campo di testo
- Il campo di testo è impostato di default a un suggerimento all'utente, es: "Ti senti fortunato?"
- Crea una funzione che produca un numero casuale da 1 a 6
 - Ci sono vari modi per farlo, cerca e scegli quello che reputi più Kotlinoso

- Alla pressione del pulsante il campo di testo viene aggiornato col risultato della funzione
 - Gli Int possono essere convertiti a stringa con `.toString()`
- Se il risultato è 1, il numero deve essere di colore rosso. Se è 6, di colore verde. Altrimenti, nero.
 - È possibile aggiungere colori in `res/values/colors.xml`

B.3 – Debugger Intro

1. A cosa serve il Debugger

Il Debugger è uno strumento che ci permette di controllare la nostra app a *runtime*, vedendo in tempo reale come si aggiornano valori e oggetti, e mettendo in pausa l'esecuzione di classi e funzioni.

2. Breakpoints

Scriviamo il seguente codice dentro *onCreate*

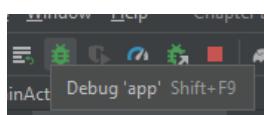
```
data class ColorHolder(val colors: List<String>) {  
    fun hasThreeColors() = colors.size == 3  
}  
  
var holder: ColorHolder  
  
val myColors = mutableListOf("Green", "Red", "Cyan")  
.also { holder = ColorHolder(it) }  
  
myColors.add("Yellow")  
  
val message =  
    if (holder.hasThreeColors())  
        "Holder has 3 colors!"  
    else  
        "Holder is bad ="  
  
println(message)
```

Lanciando l'app, il nostro holder non è quello che ci aspettiamo, avendo piu' di 3 colori.

Cliccando nella barra laterale dove sono numerate le linee di codice è possibile aggiungere un *Breakpoint*. Aggiungilo alla riga *myColors.add("Yellow")*:



Lanciamo ora l'app sull'emulatore con la funzionalità *Debug*



Giunto all'esecuzione della riga con il breakpoint, il programma si metterà in pausa **prima di eseguire la riga**. L'IDE ci mostrerà una diversa interfaccia:

```
data class ColorHolder(val colors: List<String>) {
    fun hasThreeColors() = colors.size == 3
}

var holder: ColorHolder? holder: ColorHolder(colors=[Green, Red, Cyan])

val myColors = mutableListOf("Green", "Red", "Cyan") myColors: size = 3
    .also { holder = ColorHolder(it) } holder: ColorHolder(colors=[Green, Red, Cyan])

myColors.add("Yellow") myColors: size = 3

val message =
    if (holder?.hasThreeColors())
        "Holder is good!"
    else
        "Holder is bad =("

println(message)

debug: app x
Debugger Console
✓ 'main' @ 10.638 in group 'main': RUNNING
onCreate(21, MainActivity (com.example.chapter3))
performLaunch(7136, Activity (android.app))
performCreate(7127, Activity (android.app))
callActivityOnCreate(127, Instrumentation (android.app))
performLaunchActivity(3933, ActivityThread (android.app))
handleLaunchActivity(3048, ActivityThread (android.app))
execute(78, LaunchActivityItem (android.app.servertransaction))
executeCallbacks(106, TransactionExecutor (android.app.servertransaction))
execute(68, TransactionExecutor (android.app.servertransaction))
handleMessage(108, ActivityThread$Handler (android.app))
dispatchMessage(106, Handler (android.os))
loop(193, Looper (android.os))
main(659, ActivityThread (android.app))
invoke(69, Method (java.lang.reflect))
run(493, RunTimeUnit$MethodRunnable$Caller (com.android.internal.os))
Switch frame
Launch succeeded IDE with Ctrl+Alt+Up and C... X
```

- In basso a sinistra, vediamo una lista di processi. In alto è indicato il thread dove siamo fermi; Il processo selezionato ci dice il punto del codice dove siamo fermi; mentre sotto sono elencate in sequenza le invocazioni, spesso interne del sistema Android, che alla fine hanno portato alla classe o la funzione dove ci siamo fermati.

✓ "main"@10,638 in group "main": RUNNING

onCreate:21, MainActivity (*com.example.chapterb3*)

performCreate:7136, Activity (*android.app*)

performCreate:7127, Activity (*android.app*)

callActivityOnCreate:1271, Instrumentation (*android.app*)

performLaunchActivity:2893, ActivityThread (*android.app*)

handleLaunchActivity:3048, ActivityThread (*android.app*)

execute:78, LaunchActivityItem (*android.app.servertransaction*)

executeCallbacks:108, TransactionExecutor (*android.app.servertransaction*)

execute:68, TransactionExecutor (*android.app.servertransaction*)

handleMessage:1808, ActivityThread\$H (*android.app*)

dispatchMessage:106, Handler (*android.os*)

loop:193, Looper (*android.os*)

main:6669, ActivityThread (*android.app*)

invoke:-1, Method (*java.lang.reflect*)

run:493, RunTimeInit\$MethodAndArgsCaller (*com.android.internal*)

- In basso a destra c'è una lista di variabili. E' possibile controllare come sono valorizzate al momento del nostro breakpoint:

```

    myColors = {ArrayList@11470} size = 3
    > 0 = "Green"
    > 1 = "Red"
    > 2 = "Cyan"
  
```

- Nel codice stesso vengono aggiunti dei commenti per mostrarci l'attuale valorizzazione delle variabili, oltre ad avere la riga selezionata per evidenziare il punto di stop.

```

5
6     var holder: ColorHolder  holder: ColorHolder(colors=[Green, Red, Cyan])
7
8     val myColors = mutableListOf("Green", "Red", "Cyan")  myColors: size = 3
9     .also { holder = ColorHolder(it) }  holder: ColorHolder(colors=[Green, Red, Cyan])
0
1  myColors.add("Yellow")  myColors: size = 3
0
  
```

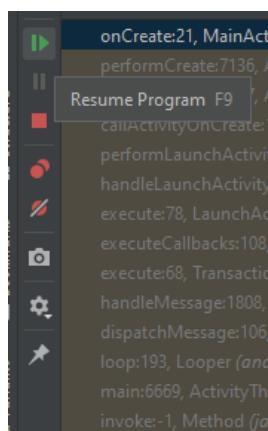
Per ora il nostro ColorHolder è corretto: ha ancora 3 colori. Aggiungiamo altri 2 breakpoint:

```

1 myColors.add("Yellow")  myColors: size = 3
2
3     val message =
4         if (holder.hasThreeColors())
5             "Holder is good!"
6         else
7             "Holder is bad =("
8
  
```

I breakpoints seguono l'esecuzione del programma: quest'ultimo procederà nell' IF se la condizione è vera, e nell'ELSE se è falsa.

Cliccando sul comando *Resume Program (F9)* possiamo farlo ripartire:



Ora ecco dove si fermerà:



```
5     val message =
6         if (holder.hasThreeColors()) "Holder is good!"
7         else "Holder is bad =("
```

La condizione è falsa, e leggendo il suggerimento è facilmente intuibile il perché.

3. Evaluate

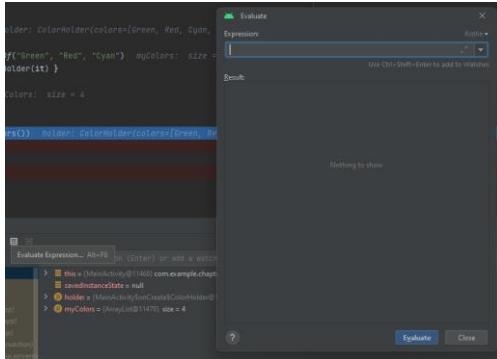
Il Debugger ci fornisce anche la possibilità di sfruttare il programma per scrivere codice mentre sta venendo eseguito.

Partendo da questi breakpoints, lanciamo l'app in debug:



```
10
11     myColors.add("Yellow")
12
13     | ⚡ val message =
14     |     if (holder.hasThreeColors())
15     |         "Holder is good!"
16     |     else
17     |         "Holder is bad =("
```

Giunti al primo breakpoint, clicca su *Evaluate Expression*



Dentro a questo tool possiamo scrivere del codice per comprendere meglio lo stato dell'app in questo istante. Ad esempio, richiediamo il size della lista *holder.colors* prima che venga lanciato IF ELSE



```
1
2
3     myColors.add("Yellow") myColors:
4
5     val message =
6         if (holder.hasThreeColors())
7             "Holder is good!"
8         else
9             "Holder is bad =("
10
11     println(message)
12
13     | ⚡ holder.colors.size|
```

Prevediamo quindi che sarà lanciata la condizione ELSE.

Abbiamo però la possibilità di modificare l'assegnazione delle variabili in tempo reale, modificando così il flusso dell'app.

Rimuoviamo il primo colore da myColors. L'IDE si aggiorna per mostrarcici i valori che sono stati modificati in corsa.

The screenshot shows the Android Studio interface. On the left, the code editor displays a Kotlin file with the following content:

```
15 var holder: ColorHolder holder: ColorHolder(colors=[Red, Cyan, Yellow])
16
17 val myColors = mutableListOf("Green", "Red", "Cyan") mycolors: size = 3
18     .also { holder = ColorHolder(it) }
19
20 myColors.add("Yellow") myColors: size = 3
21
22 val message =
23     if (holder.hasThreeColors()) "Holder is good"
24     else "Holder is bad"
25
26 println(message)
27
28
29
30
```

Line 24 is highlighted with a red breakpoint. The code editor has a 'Debug' tab selected. To the right, a 'Evaluate' dialog is open, showing the expression `myColors.removeAt(0)`. The result pane shows the updated state of `myColors`:

Result
0 result = "Green" 1 count = 10 2 hash = 69066467 3 > shadow\$_klass_ = (Class@8603)*class java.lang.String... Navigate 4 > shadow\$_monitor_ = 0

Procedendo al breakpoint successivo con *Resume Program* vediamo che ora la condizione è vera ed il programma passa dal flusso IF invece che da ELSE.

The screenshot shows the code editor again. The line `if (holder.hasThreeColors())` is now highlighted in blue, indicating it is the next line to be executed. The code editor has a 'Debugger' tab selected.

B.4 – Testing: Unit Test

1. Introduzione a Unit Test su Android

L'argomento del testing su Android è particolarmente complesso. Durante la creazione di un'app ci ritroviamo ad integrarci pesantemente con il framework, utilizzando classi al di fuori del nostro controllo, il che rende il testing difficile – a tratti impossibile – se non gestiamo correttamente il codice.

Android ci fornisce numerose librerie costruite solo allo scopo di permetterci di effettuare test.

In questa sezione del corso ci occuperemo di **Unit Test**: test automatici che permettono di valutare l'esito di una specifica funzione, che sono eseguibili sulla macchina di sviluppo in tempi molto rapidi.

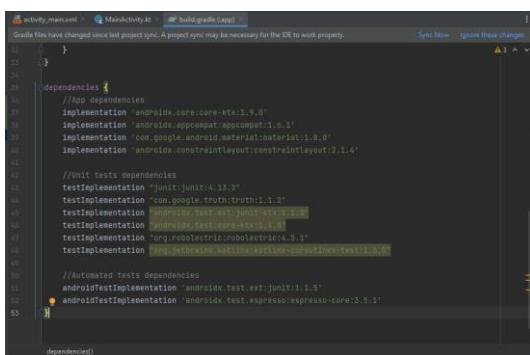
2. Setup Progetto

Per prima cosa, importiamo le librerie di test. Nel file build.gradle (app), sostituire gli import con la dicitura `testImplementation` con le seguenti righe:

```
//Unit tests dependencies
testImplementation "junit:junit:4.13.2"
testImplementation "com.google.truth:truth:1.1.2"
testImplementation "androidx.test.ext:junit-ktx:1.1.3"
testImplementation "androidx.test:core-ktx:1.4.0"
testImplementation "org.robolectric:robolectric:4.5.1"
testImplementation "org.jetbrains.kotlinx:kotlinx-coroutines-test:1.6.0"
```

Questo è un setup generico che coprirà gli Unit test per tutto il corso.

Il progetto richiederà ora un *gradle sync*



```
activity_main.xml  Mandevity_M  build.gradle (app)  Sync Now  Ignore These Changes
Gradle files have changed since last project sync. A project sync may be necessary for the IDE to work properly.

dependencies {
    //App dependencies
    implementation 'androidx.core:core-ktx:1.9.0'
    implementation 'androidx.appcompat:appcompat:1.0.1'
    implementation 'com.google.android.material:material:1.8.0'
    implementation 'androidx.constraintlayout:constraintlayout:2.3.4'

    //Unit tests dependencies
    testImplementation "junit:junit:4.13.2"
    testImplementation "com.google.truth:truth:1.1.2"
    testImplementation "androidx.test.ext:junit-ktx:1.1.3"
    testImplementation "androidx.test:core-ktx:1.4.0"
    testImplementation "org.robolectric:robolectric:4.5.1"
    testImplementation "org.jetbrains.kotlinx:kotlinx-coroutines-test:1.6.0"

    //Automated tests dependencies
    androidTestImplementation "androidx.test.ext:junit:1.1.3"
    androidTestImplementation "androidx.test.espresso:espresso-core:3.5.1"
}
```

3. Creazione di un test

Aggiungiamo il seguente codice a MainActivity:

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val result = performSum(5, 7)
    }

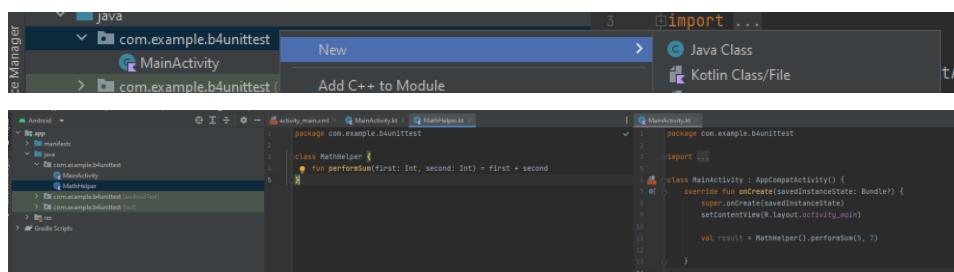
    private fun performSum(first: Int, second: Int) = first + second
}
```

Ora vogliamo testare se `performSum` funziona correttamente.

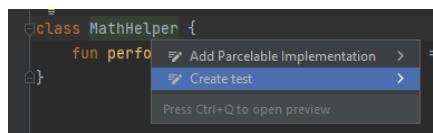
La nostra funzione però è contenuta in `MainActivity`, sarebbe quindi necessario instanziare tutta la classe – con annessi tutti i processi Android sottostanti - per potervi accedere. Il che equivale a testare tutto Android!

Per evitare di instanziare `MainActivity` durante il test, andiamo ad estrarre la funzione in una classe a sé stante.

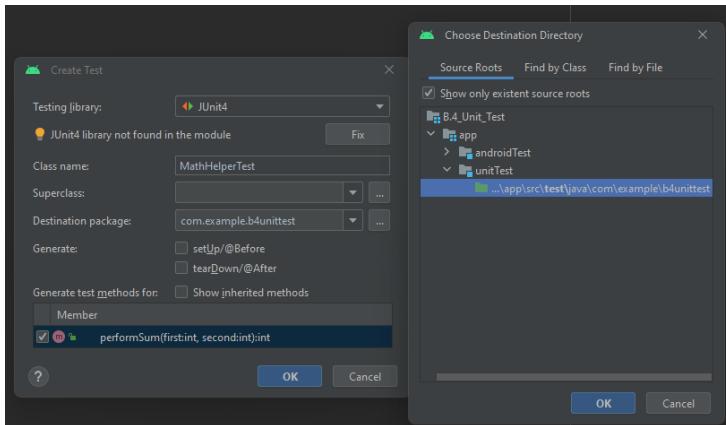
Crea una nuova *Kotlin Class* chiamata `MathHelper` nel tuo package, dove sposterai la funzione `performSum` rendendola pubblica:



Ora creiamo il file di test. Tasto destro sul nome della classe > Content actions (o *alt-invio*) > Create Test



Seleziona JUnit4 come framework di test, e segna la spunta a tutte le funzioni da testare > OK.
Infine, assicurati che il file venga creato sotto la cartella Unit Test > OK.



Ora il nostro file di test è stato generato. Dobbiamo però fare alcune modifiche:

Negli import, sostituisci `org.junit.Assert.*` con:

- `com.google.common.truth.Truth.*`

Nella riga prima di `class`, aggiungi:

- `@RunWith(AndroidJUnit4::class)`

```
1 package com.example.b4unittest
2
3 import com.google.common.truth.Truth.*
4
5 import androidx.test.ext.junit.runners.AndroidJUnit4
6 import org.junit.Test
7 import org.junit.runner.RunWith
8
9 @RunWith(AndroidJUnit4::class)
10 > class MathHelperTest {
11
12     @Test
13     fun performSum() {
14         ...
15     }
16 }
```

4. Naming Convention

La convenzione dei nomi dei test è la seguente:

`functionName_initialStatus_expectedResult`

Modifichiamo quindi il nome del nostro test in:

```
1 @Test
2     fun performSum_twoPositives_success() {
3 }
```

Lo scopo del nostro test sarà quindi di vedere se la funzione *performSum* produce il risultato corretto quando gli forniamo in input due numeri positivi.

5. Assertions

Il concetto di un test è comparare un valore o un oggetto ad un risultato che ci si aspetta. Se questa *affermazione(assertion)* produrrà un esito positivo *true*, il test sarà un successo (*passed*). Ad esempio:

"asserisco che una macchina ha 4 ruote."

Se la macchina uscirà dalla fabbrica con 3 ruote, non passerà il nostro test.

Per questa procedura si usano degli strumenti di comparazione. Noi useremo la libreria *Truth* di google, con le sue assertions *assertThat*, che permette di scrivere comparazioni in maniera tale che siano facilmente comprensibili.

6. Successo e Fallimento

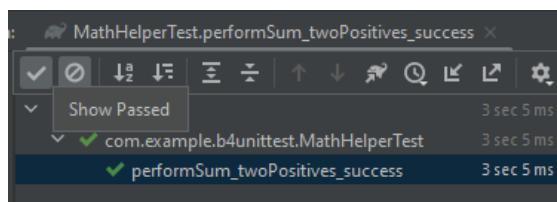
Scriviamo il test:

```
2
3     @Test
4     fun performSum_twoPositives_success() {
5         val result = MathHelper().performSum(5,5)
6         assertThat(result).isEqualTo( expected: 10)
7     }
```

Sarà la linea *assertThat* a definire la condizione per cui il soggetto passerà o no il test.

Lanciamo il test con la freccia verde a sinistra del nome del test

Nella tab in basso, selezionare le spunte *show passed* e *ignored* per mostrare tutti i test eseguiti



Il nostro primo test è passato. Vediamo cosa succede se il test fallisce:

```
3
4     @Test
5     fun performSum_twoPositives_success() {
6         val result = MathHelper().performSum(5,1)
7         assertThat(result).isEqualTo( expected: 10)
```

Lanciando il test, potremo vedere quale è il test che non è passato, e leggere nel log la condizione che ha fallito:



7. Strategie di test

E' buona norma valutare tutte le possibilità a cui la nostra funzione può essere sottoposta, con una serie di casistiche:

```

9
10    @RunWith(AndroidJUnit4::class)
11    class MathHelperTest {
12
13        @Test
14        fun performSum_twoPositives() {
15            val result = MathHelper().performSum(5, 5)
16            assertThat(result).isEqualTo(expected: 10)
17        }
18
19        @Test
20        fun performSum_twoNegatives() {
21            val result = MathHelper().performSum(-5, -5)
22            assertThat(result).isEqualTo(expected: -10)
23        }
24
25        @Test
26        fun performSum_oneNegative() {
27            val result = MathHelper().performSum(-5, 5)
28            assertThat(result).isEqualTo(expected: 0)
29        }
30

```

```

Run: MathHelperTest ×
  ✓ Tests passed: 3 of 3
  ✓ Test Results
  ✓ com.example.b4unittest.MathHelperTest
    ✓ performSum_twoPositives_success
    ✓ performSum_oneNegative_success
    ✓ performSum_twoNegatives_success

```

Se necessario, per ogni test è possibile inserire più condizioni, aggiungendo altre righe `assertThat`. La prima condizione non rispettata farà fallire il test.

È importante però mantenere queste condizioni strettamente all'interno dello scopo del test, per non falsarne i risultati:

```

@Test
fun performSum_twoPositives_success() {
    val testNumber = 5
    val result = MathHelper().performSum(testNumber, testNumber)
    assertThat(testNumber).isGreaterThan(other: 0)
    assertThat(result).isEqualTo(expected: 10)
}

```

Esempio corretto: stiamo valutando se lo stato iniziale del test è rispettato (numero positivo)

```

@Test
fun performSum_twoPositives_success() {
    val testNumber = 5
    val result = MathHelper().performSum(testNumber, testNumber)
    assertThat(testNumber).isAnyOf(first: 2, second: 3, ...rest: 5, 7, 11)
    assertThat(result).isEqualTo(expected: 10)
}

```

Esempio scorretto: sebbene il test passi, non è pertinente mettere come condizione che `testNumber` debba essere numero primo.

8. @Before, @After

Il nostro test vira intorno al fatto di testare un singolo *test subject*, in questo caso la classe *MathHelper* e le sue funzioni. Sebbene la nostra classe sia ora molto leggera, potrebbe nel tempo diventare molto complessa, anche nel costruttore. Andiamo quindi a inizializzarla una singola volta per questa serie di test:

```
//test subject
val mathHelper: MathHelper = MathHelper()

@Test
fun performSum_twoPositives_success() {
    val testNumber = 5
    val result = mathHelper.performSum(testNumber, testNumber)
    assertThat(result).isGreaterThan( other: 0)
    assertThat(result).isEqualTo( expected: 10)
}

@Test
fun performSum_twoNegatives_success() {
    val result = mathHelper.performSum(-5, -5)
    assertThat(result).isEqualTo( expected: -10)
}

@Test
fun performSum_oneNegative_success() {
    val result = mathHelper.performSum(-5, 5)
    assertThat(result).isEqualTo( expected: 0)
}
```

Tutto funziona, ma c'è un problema. *MathHelper* viene creato una sola volta, e per i nostri test questa cosa non è ideale: se il suo stato interno, ad esempio delle variabili, dovessero cambiare, i nostri test potrebbero fallire. Ad esempio il primo test potrebbe modificare una variabile che al secondo test serve sia quella di default. Scrivendo i test, ci rapportiamo ad uno specifico stato del test subject, che di solito è lo stato iniziale.

Modifichiamo *MathHelper* per fornigli uno stato:

```
class MathHelper {
    val recentResults: MutableList<Int> = mutableListOf()
}

fun performSum(first: Int, second: Int) = (first + second)
    .also { recentResults.add(it) }
```

Ora cambiamo l'inizializzazione di *MathHelper* nel test

```
//test subject
lateinit var mathHelper: MathHelper

@Before
fun before() {
    mathHelper = MathHelper()
```

La funzione taggata con `@Before` sarà lanciata prima di ogni test.

Aggiungiamo una coppia di test per la nuova funzionalità:

```
@Test
fun recentResults_noResults_oneResult() {
    mathHelper.performSum(5, 5)
    assertThat(mathHelper.recentResults.size).isEqualTo( expected: 1)

}

@Test
fun recentResults_oneResults_twoResults() {
    mathHelper.recentResults.add(99)
    val result = mathHelper.performSum(5, 5)
    assertThat(mathHelper.recentResults[1]).isEqualTo(result)
```

Se avessimo mantenuto `mathHelper` inizializzato come `val` una volta sola, saremmo potuti incombere in problemi con dei test che analizzano lo stato `recentResults`.

Sebbene il framework di test cerchi automaticamente di ricreare lo stato iniziale di tutti i soggetti ogni volta che effettua un `@Test`, è buona norma impostare sempre le inizializzazioni che vengono ripetute dentro la funzione `@Before`

Una funzione taggata `@After` ha lo stesso concetto: verrà lanciata dopo ogni test.

```
@After
fun after() {
    mathHelper.recentResults.clear()
```

9. Esercizio: test Lancio dadi

Nell'app Lancio dadi:

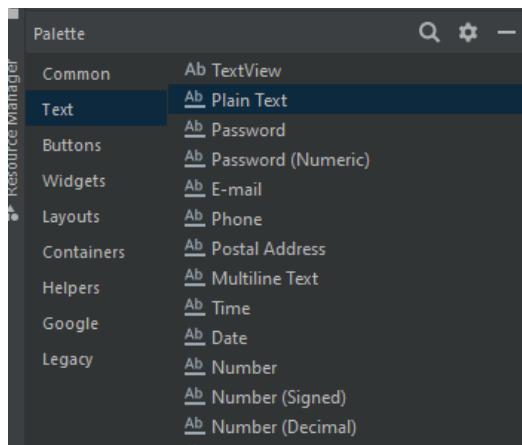
- Estrai le funzioni di lancio dalla main activity in una classe specifica *DiceRoller*
- Aggiungi a *DiceRoller* la possibilità di avere uno stato che rappresenta quante facce ha il dado da lanciare. Modifica le funzioni di conseguenza, per integrare questo stato.
- Crea la classe di test con tutte le funzioni
- Instanzia il test subject correttamente, utilizzando @Before o @After se necessario
- Testa tutti i casi che ritieni validi, in particolare gli edge-cases

C.1 – Input Utente

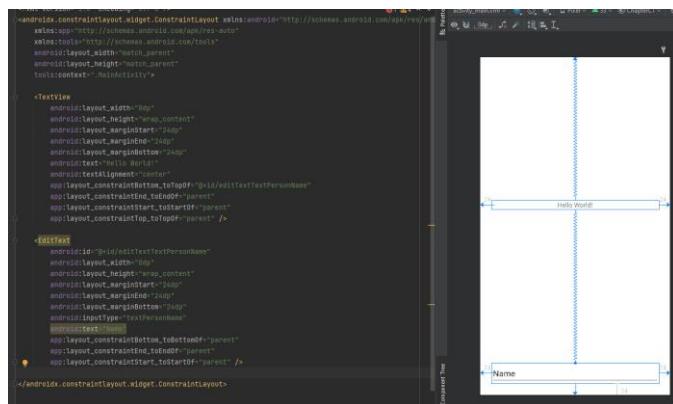
1. EditText, TextInputLayout

Un'operazione molto comune in un'app è l'inserimento di testo o numeri in un campo di testo. Lo strumento classico è il widget `EditText`, mentre uno più moderno è il `TextInputLayout`. Entrambi sono validi e possono essere applicati in base alle casistiche.

Nell'editor *Design*, sotto la categoria `Text` è possibile utilizzare dei campi precompilati. Questi campi limitano l'input dell'utente a certe categorie – ad esempio, solo numeri.



Trascina `Plain Text` nella parte bassa del layout, regolando i constraint delle due view per fare in modo che siano una sopra l'altra:



Lanciando l'app vediamo che è possibile scrivere all'interno del campo di testo.

`EditText` è un widget che permette di inserire testo

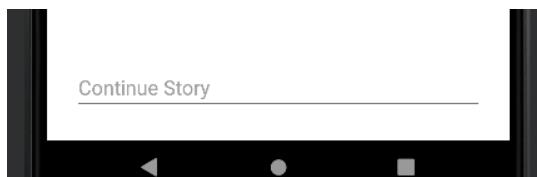
Decidiamo che l'utente potrà spingere un pulsante per aggiungere il testo ad Hello world, per formare un racconto.

Ora il campo di testo è già prepopolato, e bisogna cancellare a mano il testo "Name". Però questo testo è necessario, altrimenti l'utente non capirebbe cosa bisogna scrivere nel campo di testo. Inoltre, quando apriamo la tastiera, l'impostazione `inputType=textPersonName` fa sì che la tastiera venga modificata per non includere un "a capo", cosa che non è adatta a scrivere una storia.

Andiamo a cambiare il `text` in `hint`, inoltre modifichiamo anche l'`inputType` in `text`:

```
    android:inputType="text"  
    android:hint="@string/continue_story"
```

Ora l'`hint` sarà visibile in grigio e potremo scriverci sopra. La tastiera non avrà impostazioni particolari.



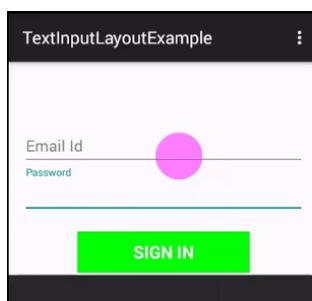
Fin qui tutto bene, ma c'è una limitazione: se dovessimo inserire più campi di testo, come ad esempio in un form dove si richiedono le informazioni dell'utente, le hint dei vari `EditText` diventerebbero meno intuitive, perché una volta scritto nel campo di testo non sono più visibili. Ecco un esempio:



Una volta scritto all'interno dei campi non abbiamo più riferimenti per capire chi sia l'eroe e chi il villain.

Potremmo cominciare ad inserire delle label sopra i campi di testo, ma essendo questa una casistica molto comune, Android ci fornisce un widget che fa proprio questo: il `TextInputLayout`

`TextInputLayout` è un widget che permette di inserire testo, facilitando lo stile



Aggiungiamolo e vediamo che è in realtà una coppia di widget:

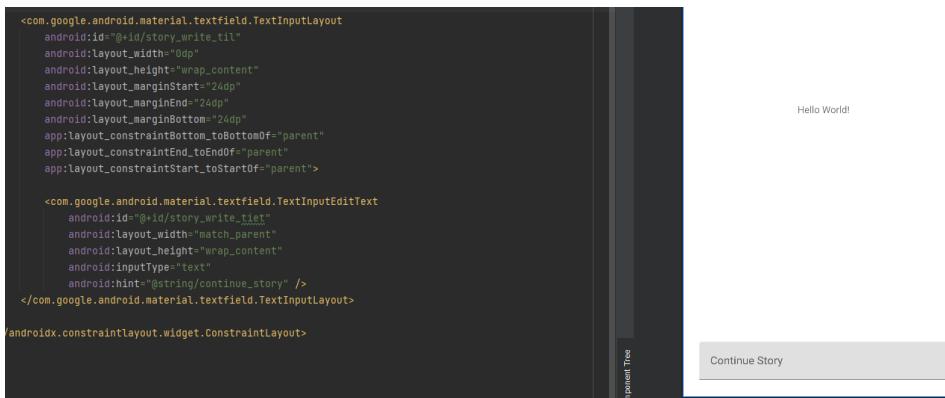


- TextInputLayout (TIL) si occupa di gestire constraint, dimensioni e animazioni
- TextInputEditText (TIET) gestisce testo, stile e input

Applica, secondo questi paradigmi, le impostazioni della edittext precedente a questo widget;

Diamo a entrambi i componenti del widget un id;

Infine, commenta l'edittext:



Per il nostro obiettivo, aggiungiamo anche un pulsante che l'utente andrà a spingere una volta che ha scritto il testo:



Layout:

```
<?xml version="1.0" encoding="utf-8"?>
<andoidx.ConstraintLayout>
<andoidx.ConstraintLayout>
<andoid:layout_width="match_parent"
    andoid:layout_height="match_parent"
    andoid:layout_weight="1"
    andoid:text="@string/add"
    andoid:layout_marginStart="0dp"
    andoid:layout_marginEnd="0dp"
    app:layout_constraintBottom_toBottomOf="@+id/story_write_til"
    app:layout_constraintTop_toTopOf="@+id/story_write_til"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toEndOf="@+id/story_write_til"/>
</andoidx.ConstraintLayout>
</andoidx.ConstraintLayout>
```

Ora andiamo ad ottenere programmaticamente il testo scritto.

Secondo il nostro caso d'uso, quando l'utente spinge il pulsante dobbiamo aggiungere il testo al messaggio stampato dall'app.

All'interno del *clickListener* di storyAddBtn:

```
storyAddBtn.setOnClickListener { it: View ->
    val newStoryText = "\n" + storyWriteTiet.text.toString()
```

- È necessario convertire il testo di storyWriteTiet in stringa, perché è un *Editable*, a mostrare il fatto che può essere modificabile.
- Il carattere speciale "\n" in una stringa crea una nuova riga.

Una volta ottenuto il testo, concatenalo al testo di storyTextTv:

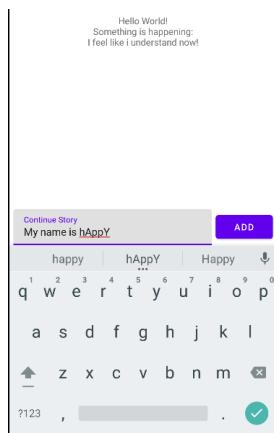
```
storyTextTv.apply { this: TextView
    text = text.toString() + newStoryText
}
```

- Kotlin permette di compattare le funzioni *setText* e *getText* semplicemente in *text*, che verrà considerato getter o setter in base a come viene utilizzato.
- Il Tipo di *TextView.text* è l'interfaccia [CharSequence](#), perché il contenuto di una *TextView* puo' essere qualsiasi tipo di testo (ad esempio, un *Editable*)

Infine, *clean* il campo di testo in maniera tale che l'utente possa continuare a scrivere comodamente:

```
storyWriteTiet.setText("") //sets the Editable content to a string
```

Ora lanciando l'app, possiamo vedere che alla pressione di Add il testo viene aggiunto ed il campo di testo resettato.

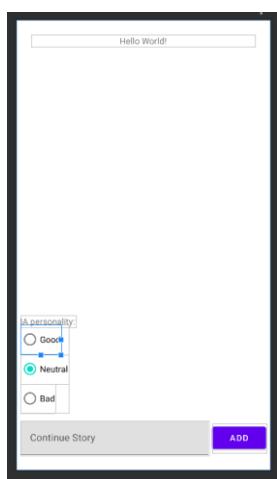


2. RadioButtons

I RadioButtons vengono utilizzati quando l'utente deve effettuare una scelta tra più opzioni.

RadioGroup è un layout che include più RadioButton, un widget che permette all'utente di fare una scelta univoca.

Aggiungiamo un RadioGroup con 3 opzioni, inoltre aggiungiamo un testo come label, perché RadioGroup non fornisce una label.



Layout:

```
<?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"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:id="@+id/story_text_tv"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        ...>
```

In questo esempio abbiamo impostato il secondo pulsante del radio group a *checked*, in maniera tale che sarà subito selezionato.

Decidiamo che la scelta del radio group influenzera il background del testo

Similmente a Button, RadioGroup fornisce la possibilità di aggiungere un listener per alcuni eventi, ad esempio quando cambia il membro selezionato tra i suoi child.

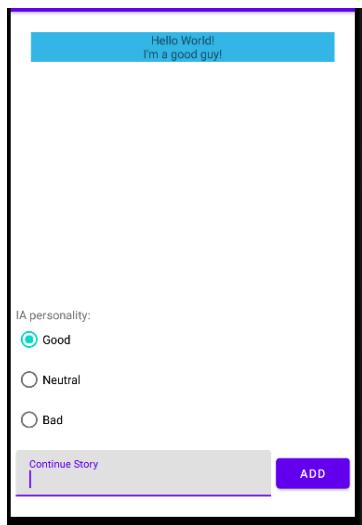
```
storyOptionsRg.setOnCheckedChangeListener { radioGroup, checkedId ->
```

andando a fare un razionale su *checkedId* possiamo impostare come modificare l'interfaccia in base al pulsante che è stato *checked*

```

storyOptionsRg.setOnCheckedChangeListener { radioGroup, checkedId ->
    val color = when(checkedId) {
        R.id.story_options_1_rb -> android.R.color.holo_blue_light
        R.id.story_options_3_rb -> android.R.color.holo_red_light
        else -> R.color.white
    }
    storyTextTv.setBackgroundColor(getColor(color))
}

```



3. Compatibilità dei widget

Alcune volte è necessario porre particolare attenzione alla versione del widget che si sta usando. Poniamo ad esempio di voler inserire uno Switch.

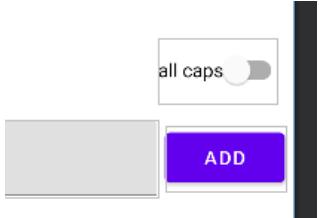
Switch è un interruttore, permette all'utente di attivare o disattivare una funzione

Aggiungiamo uno Switch, sopra il pulsante di aggiunta:

```

<Switch
    android:id="@+id/story_caps_sw"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/all_caps"
    android:layout_margin="12dp"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintBottom_toTopOf="@id/story_write_til"/>

```



L'IDE ci menziona che non è ideale usare questa versione del widget e ci suggerisce delle alternative:



Questo perché il widget Switch non include delle retrocompatibilità con vecchie versioni di android, implementate invece da SwitchCompat e dal suo successore SwitchMaterial.

È spesso più importante utilizzare classi compatibili con le versioni di android che si decide di supportare, rispetto a classi più aggiornate e funzionalità più avanzate

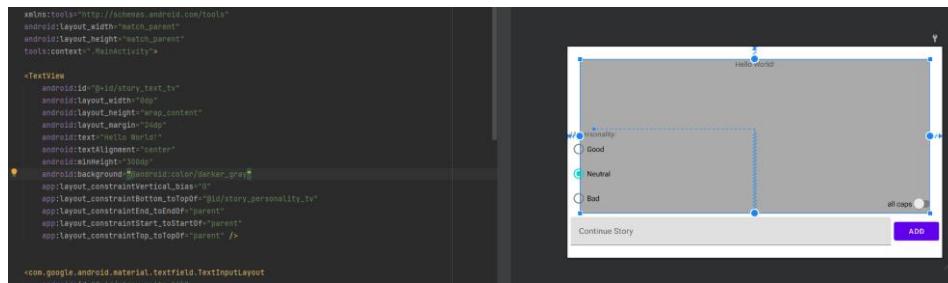
4. ScrollView

Il nostro layout si riordina in base alle dimensioni dello schermo del device, ma nel caso le view contenute nel layout richiedano più spazio di quello disponibile su schermo, potrebbero cominciare a sovrapporsi o ad estendersi al di fuori dello spazio visibile all'utente.

Aggiungi questi campi alla textview:

```
android:layout_height="300dp"
android:background="@android:color/darker_gray"
```

Questo imposterà il campo di testo ad un'altezza statica. Vedendo la preview in modalità landscape:



Ora il campo di testo richiede troppo spazio, e si sovrappone ai comandi.

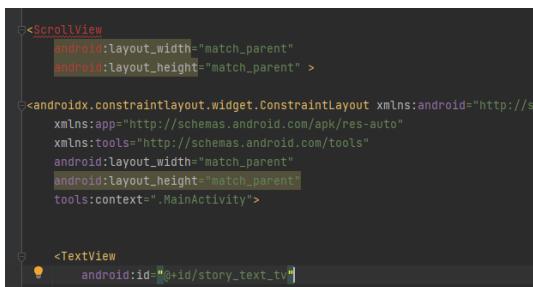
Rendiamo il nostro layout scrollabile

Per farlo, dobbiamo racchiudere il Constraint Layout dentro una scrollview.

ScrollView è in grado di far scrollare verticalmente o orizzontalmente il suo contenuto. Può avere solo un singolo discendente diretto.

Una implementazione comune è impostare la scrollview a match parent, per fare in modo che occupi tutto lo spazio dedicato al layout. Il suo contenuto invece viene impostato a wrap content, cosi' da potersi allungare quanto necessario al di fuori dello schermo visibile all'utente.

- Aggiungi il viewgroup *ScrollView* prima del constraint layout



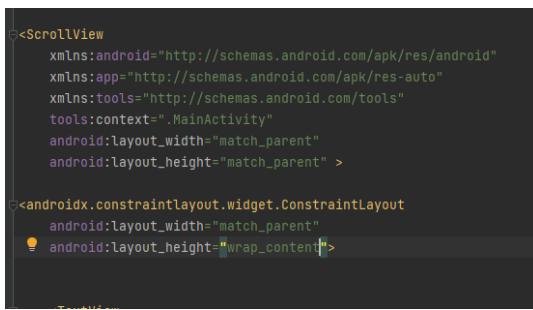
```
<ScrollView
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

<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"
    tools:context=".MainActivity" >

    <TextView
        android:id="@+id/story_text_tv" />

```

- Sposta i campi *xmlns*, che effettuano gli import di configurazione, da constraint layout a scrollview.
Sposta anche *tools:context*
- Rendi l'altezza di constraint layout *wrap_content*



```
<ScrollView
    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"
    tools:context=".MainActivity"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

<androidx.constraintlayout.widget.ConstraintLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content" >
```

- Alla fine del file, chiudi il tag ScrollView



```
110
111     </androidx.constraintlayout.widget.ConstraintLayout>
112 </ScrollView>
```

Ora il layout scorre verticalmente.

5. Esercizio: Hero Generator

Sviluppa un'app in una singola pagina che permette di generare la storia di un eroe.

- Un campo di testo deve definire il nome
- Un radioGroup permette di scegliere un superpotere o qualità
- Uno switch indica se ha un acerrimo nemico, oppure no
- Cliccando un pulsante, è possibile generare un testo discorsivo che include tutte le qualità selezionate.
 - Es: "Batman ha il superpotere di essere ricco. Il suo acerrimo nemico è Mario Rossi."
- Il testo deve essere posto in fondo al layout. Il layout deve essere scrollabile.
- Estrarre la o le funzioni di generazione, ed effettuare unit test.

C.2 - Material Design

Il Material Design è un insieme di guidelines per definire non solo come progettare l'interfaccia (UI), ma anche come le view debbano interagire visivamente, con l'utente e tra di loro, in maniera tale da creare una user experience (UX) fluida e istintivamente comprensibile.

Il tema del MD è molto ampio, per ulteriori approfondimenti e strumenti:

m2.material.io

Per cominciare, lavora sulla versione 2 del material design, in quanto la più consolidata.

Comincia questo capitolo creando un nuovo progetto con una Navigation Drawer Activity

1. Introduzione a design UI e UX

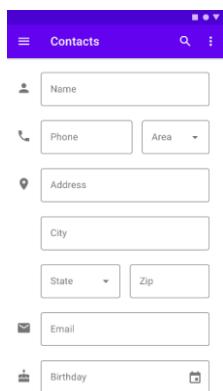
L'obbiettivo del design di un'app è rendere la sua interfaccia naturalmente fruibile da un utente, mantenendo al tempo stesso una sua unicità. Si può dividere il design in due concetti fondamentali:

- User Interface (UI): è come viene rappresentata graficamente l'interfaccia
- User Experience (UX): è come l'utente interagisce con l'app e naviga tra le funzioni

Molti concetti e azioni sono parte del nostro quotidiano utilizzo di telefoni e pc: è necessario allinearsi a queste abitudini per evitare che l'utente debba "studiarsi" l'interfaccia per utilizzare il nostro prodotto.

Da un punto di vista UX, l'ambiente mobile ha inoltre delle particolarità specifiche che rinforzano il concetto di rapida usabilità: ad esempio, è inusuale su un app trovare il pulsante "salva", in quanto ci si aspetta che delle opzioni modificate siano subito attive, che un messaggio scritto ma non inviato venga salvato come bozza, o che una foto sia subito disponibile nella galleria.

Analizziamo la UI della seguente schermata:



Tutto quello che vediamo è facilmente identificabile come un'azione o un aiuto visivo.

- La Lente di ingrandimento in alto a destra identifica il campo di ricerca, senza che ci sia un testo a spiegarlo.
- Tappando i tre puntini in alto a destra ci aspettiamo che esca un menu contestuale alla schermata, mentre le tre linee in alto a sinistra rappresentano un menu generale dell'app
- Le icone vicino ai campi di testo non sarebbero necessarie, ma istintivamente ci dividono il form in delle sezioni piuttosto chiare. L'icona "location" ci fa capire che quei campi di testo rappresentano una posizione geografica

- Vedendo l'Icona in basso a destra, ci si aspetta che tappando quel campo o l'Icona stessa si aprirà un picker di calendario

Questa iconografia è spesso necessaria per questioni di spazio. Le app vengono utilizzate su uno schermo relativamente piccolo per la quantità di informazioni che contengono: quindi dobbiamo veicolare queste informazioni in modo chiaro, ma conciso, ed anche pratico da utilizzare quando si è per strada.

Sono disponibili guidelines per ogni view Material, così da avere un riferimento durante la progettazione e lo sviluppo, a questo indirizzo

<https://m2.material.io/components>

2. Colori

Comincia questo capitolo creando un nuovo progetto con una Navigation Drawer Activity

<https://m2.material.io/design/color/the-color-system.html>

Nell'interfaccia di un'app, i colori non sono solo un abbellimento o una brandizzazione, ma rappresentano funzionalità e concetti.

Immagina di dover progettare l'app per IKEA.



Conosciamo questo brand per il giallo ed il blu, ma notiamo che nonostante il logo sia scritte blu su giallo, nei loro negozi invertono i colori. Questo perché un capannone giallo sarebbe un pugno in un occhio!

Un'app lavora su concetti simili. Avremo due "serie" di colori che identificano come vengono mostrate le nostre view:

- colorPrimary identifica il colore principale del brand, e sarà quello mostrato più frequentemente nelle schermate dell'app
- colorSecondary fa parte o no del brand, e può essere una variante del colorPrimary o un altro colore. Identificherà parti nell'app che vogliamo mettere in risalto, come i pulsanti o delle selezioni.

Il Material Design propone delle palette di colori da utilizzare:

Red 50	#FFEBEE	Pink 50	#FCE4EC	Purple 50	#E9E6FA
100	#FFCDD2	100	#F8BBD0	100	#E1BEE7
200	#F79A9A	200	#F4E8B1	200	#E9E9D8
300	#E57373	300	#F0D9D2	300	#E9A9C9
400	#E7336C	400	#ECAD71	400	#E8A8BC
500	#F44336	500	#E91E65	500	#F0C78D
600	#E53935	600	#D9B36D	600	#F8E2AA
700	#D02F2F	700	#C2B88B	700	#F8F1A2
800	#C62828	800	#AD1457	800	#F5A99A
900	#B71C1C	900	#B88BEA	900	#F4A1B8
A100	#FFB700	A100	#FFB3AB	A100	#FAB9FC
A200	#FF5722	A200	#FF7043	A200	#FDD0D8
A400	#FF7176	A400	#FF5050	A400	#FDD0D8
A700	#FF0000	A700	#C51162	A700	#FADBD8
Deep Purple 50	#EDE7E6	Indigo 50	#E8EAF6	Blue 50	#E1F2FD
100	#D0C4E9	100	#C5CAE9	100	#B8D7FD
200	#B399D8	200	#9EAFAD	200	#90CAF9

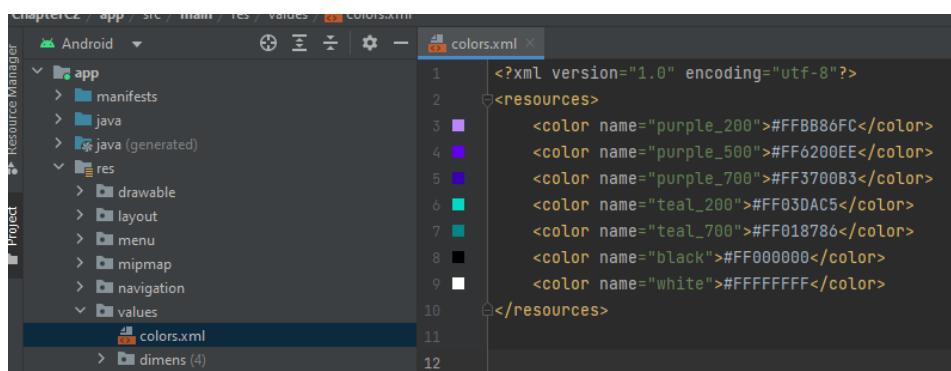
<https://m2.material.io/design/color/the-color-system.html#tools-for-picking-colors>

Queste palette forniscono delle tonalità che sono studiate per avere un buon contrasto sugli schermi dei telefoni, differenziandosi tra le varie sfumature, oltre ad essere funzionali alla vista della maggior parte delle persone.

È quindi possibile differenziarsi da questi colori, ma è consigliato prima cercare di utilizzarli.

Queste palette includono delle varianti con la lettera **A (accent)** che è la dicitura precedentemente usata per i **colorSecondary**

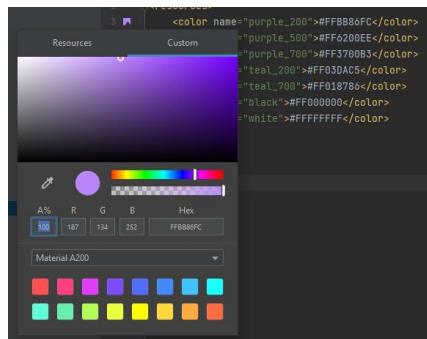
Il file **/res/values/colors.xml** contiene le risorse colore.



```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="purple_200">#FFBBB86FC</color>
    <color name="purple_500">#FF6200EE</color>
    <color name="purple_700">#FF3700B3</color>
    <color name="teal_200">#FF03DAC5</color>
    <color name="teal_700">#FF018786</color>
    <color name="black">#FF000000</color>
    <color name="white">#FFFFFF</color>
</resources>
```

I colori vengono definiti con la **hex notation** estesa, dove le coppie di lettere indicano in sequenza:
trasparenza, rosso, giallo, blu.

Android studio ci offre un'anteprima del colore nella colonna di sinistra, che se andiamo a cliccare attiverà un color picker



Nella parte bassa, potremo selezionare i colori della palette Material.

3. Theme

Oltre alle opzioni che permettono di stilizzare singolarmente ogni singola view, Android ci fornisce degli strumenti per effettuare una stilizzazione generica della nostra app.

Un Theme è una raccolta di attributi applicata a un'intera app, flusso o gerarchia di views

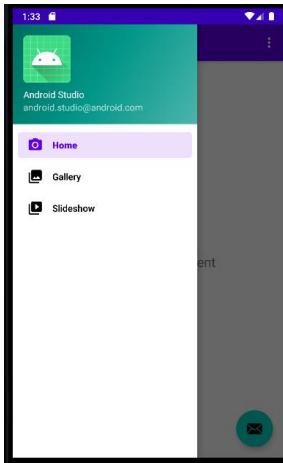
Apri il file themes.xml

Vediamo che una serie di attributi mappa altrettanti colori tramite l'xml tag `<style>`. Questi attributi sono predefiniti, e vengono applicati ad ogni view Material tramite il tema Theme.MaterialComponents. Le view Material includono tutti i widget ed i layout che abbiamo utilizzato finora, e che utilizzeremo.

I nomi degli attributi rappresentano i concetti di primary e secondary espressi nelle guidelines del Material Design. Alla seguente pagina sono presenti delle tabelle che indicano la funzionalità di ogni attributo:

<https://m2.material.io/develop/android/theming/color>

Ecco come appare l'app ora:



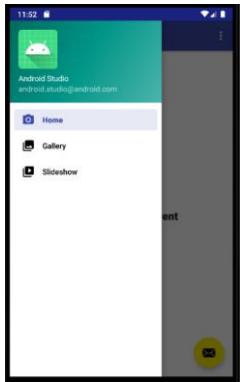
Vediamo come è possibile modificare il tema generale dell'app, per stilizzarla come se fosse l'app IKEA.

Aggiungi questi colori a colors.xml

```
<color name="indigo_600">#3949AB</color>
<color name="indigo_900">#1A237E</color>
<color name="yellow_400">#FFEAA0</color>
<color name="yellow_700">#FFD600</color>
```

Modifica ora i riferimenti ai colori nel tema, cambiando i primary in indigo e i secondary in yellow

Rilancia l'app:



Vediamo come i colori primari influenzano alcune view principali, come ad esempio la toolbar dell'app (primary) e di sistema (primaryVariant).

OnPrimary definisce il colore delle scritte o icone che si trovano sopra i colori primari, in maniera che risaltino.



I colori secondari definiscono invece view come pulsanti o dialoghi di notifiche, che veicolano all'utente la possibilità di effettuare azioni, o messaggi.



4. Style

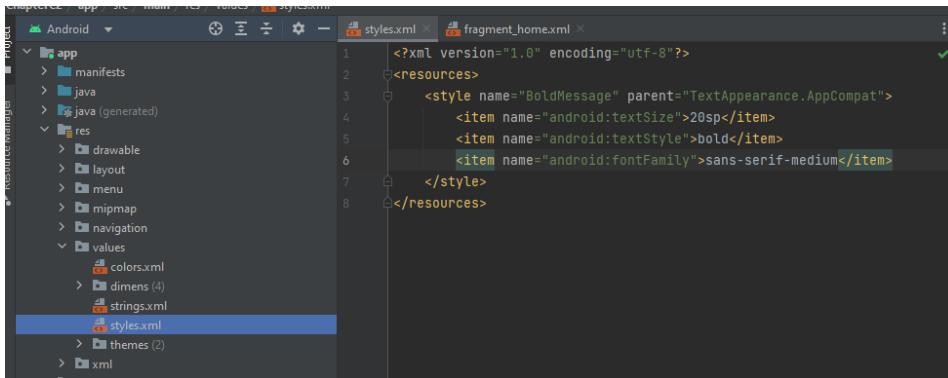
Spesso ci ritroviamo a dover standardizzare alcune view: un caso d'uso tipico è che tutte le TextView che fungono da titolo abbiano la stessa grandezza del carattere e siano in bold.

Uno Style è una raccolta di attributi applicata a molteplici istanze di una view.

Nel nostro progetto abbiamo tre pagine con una descrizione:

This is home Fragment

Ansiamo a cambiare lo *stile* di questo testo, in maniera tale da raggruppare gli attributi che poi applicheremo a 3 textView in 3 layout diversi.



In un nuovo file di risorse `styles`, creiamo uno `style` dandogli un nome, estendendo un parent di sistema: quest'ultimo punto è necessario per mantenere la retrocompatibilità dei widget.

Nel nostro caso, lo stile sarà uguale a `TextAppearance.AppCompat`, tranne per gli attributi che specifichiamo.

Ora definiamo degli attributi come se li stessimo applicando alla `TextView`:

```

<item name="android:textSize">20sp</item>
<item name="android:textStyle">bold</item>
<item name="android:fontFamily">sans-serif-medium</item>

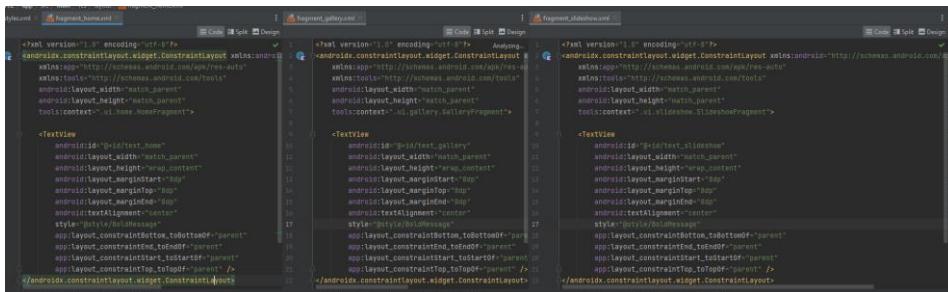
```

Possiamo scrivere qualsiasi attributo nello stile, ma la view lo considererà solo se è uno dei suoi attributi.

Infine, modifichiamo le 3 `textview` sostituendo

`android:textSize="20sp"`

che non è più necessario perché già incluso nel nostro nuovo stile, con `style="@style/BoldMessage"`



Lanciando l'app, vediamo che tutte le `textview` sono ora uguali.

This is home Fragment

Possiamo anche creare dei sotto-stili del nostro stile:

```

<style name="BoldMessage" parent="TextAppearance.AppCompat">
    <item name="android:textSize">20sp</item>
    <item name="android:textStyle">bold</item>
    <item name="android:fontFamily">sans-serif-medium</item>
</style>

❷ <style name="BoldMessage.large">
    <item name="android:textSize">28sp</item>
</style>

```

Impostando `BoldMessage.large` avremo lo stesso risultato che con `BoldMessage`, tranne che il `textSize` sarà più grande.

5. Icone

Le icone nelle app sono un metodo di comunicazione. Mettere un'icona può aiutare l'utente a chiarificare funzionalità, mentre icone superflue o fuori contesto possono confondere l'utente.

Android supporta vari tipi di immagine, ma i più comuni sono: png e jpg (raster), e vettoriali tramite xml.

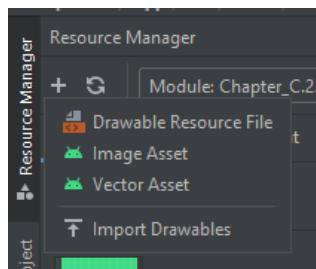
È possibile scaricare numerose icone dal sito Material dedicato, ed è buona norma cercare di utilizzarlo, in particolare per le icone che identificano un'azione molto comune come "settings" o "share". Le icone in questo sito sono infatti lo standard per Android, rimuovendo così una serie di problemi di ottimizzazione e mantenendo uno stile coerente non solo nell'app, ma anche con il sistema operativo.

<https://fonts.google.com/icons?icon.platform=android>

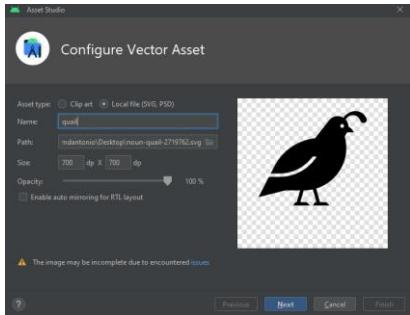
Accade spesso però che non ci sia l'icona che stiamo cercando.

Aggiungiamo un'icona vettoriale non-Material presa dal sito Noun Project.

Nel resource manager, importa un Vector Asset:

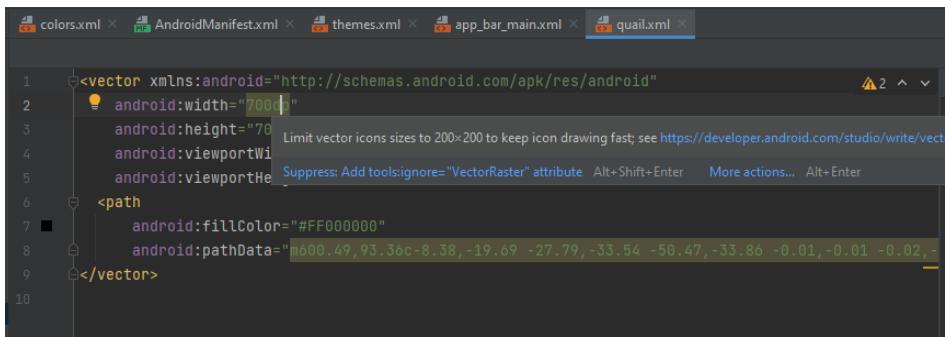


Scegli un file svg dal tuo computer



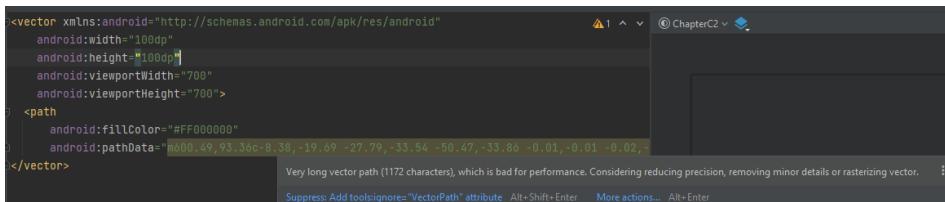
L'import convertirà il file da svg a xml, e nel caso di errori cercherà di completare il lavoro con il miglior risultato possibile. Sarà importata nella cartella `res/drawable`

Aprendo il file, vediamo che Android Studio ci notifica dei warning. Il Primo:



Le immagini vettoriali vendono ridimensionate in base alla risoluzione – sono una serie di punti tra i quali vengono tracciate delle righe (vettori). Non è necessaria quindi una grandezza specifica, ma solo mantenere le proporzioni corrette. Abbassiamo quindi i valori a 100 e 100.

Andiamo a vedere il secondo errore:

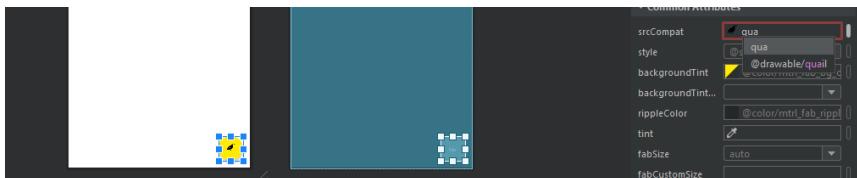


Il nostro disegno è troppo complesso con 1172 caratteri: la lunghezza consigliata è <800 caratteri.

Valutiamo subito se è possibile cambiare l'icona con un'altra, in quanto i passaggi di ottimizzazione sono spesso lunghi e complessi.

Se l'icona è necessaria, sono disponibili numerosi tool per ridurre la complessità dei file vettoriali. Nel caso anche i tool non siano sufficienti, è possibile modificarli a mano con editor vettoriali come Adobe Illustrator.

Infine, andiamo ad utilizzare l'icona come risorsa *drawable*.



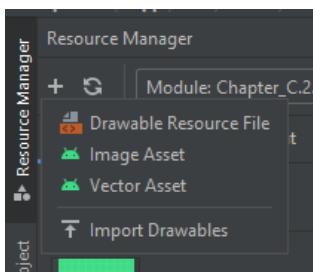
6. Launcher Icon

L'icona launcher rappresenta la nostra app nei menu di sistema, nello store e in altri contesti generici.

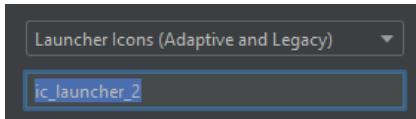
Viene implementata tramite la risorsa *mipmap*: una risorsa speciale esclusiva per le icone *launcher*, che devono essere particolarmente chiare non solo nelle varie risoluzioni, ma anche nelle customizzazioni del sistema operativo effettuate dai vari produttori di device.

Modifichiamo l'icona della nostra app (launcher icon).

Nel resource manager, importa un Image Asset:



ic_launcher è il nome di default dell'icona dell'app. Per evitare di sovrascrivere, chiamiamola ic_launcher_2.

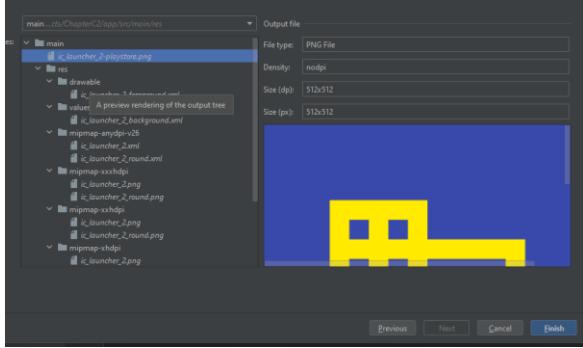


Andiamo a impostare i seguenti valori:

- **Foreground layer:** selezioniamo una clip art (nell'esempio, *business*) e un colore, dove utilizzeremo uno dei colori del nostro tema, primary o secondary (FFEA00)
- **Background layer:** utilizzeremo un colore del nostro tema (3949AB)

Assicurandosi che l'immagine sia all'interno del cerchio che rappresenta la *safe zone* dove l'immagine non potrà essere tagliata.

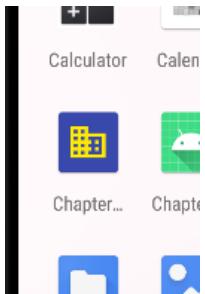
Confermando, Android Studio ci propone un'anteprima delle immagini che verranno create, incluso il launcher, varie risoluzioni, e per il play store.



Dopo aver creato le immagini, modifichiamo il *manifest* dove è specificata l'icona dell'app nel campo *icon*

```
4
5     <application
6         android:allowBackup="true"
7         android:dataExtractionRules="@xml/data_extraction_rules"
8         android:fullBackupContent="@xml/backup_rules"
9         android:icon="@mipmap/ic_launcher_2"■
10        android:label="Chapter 0.2"
11        android:supportsRtl="true"
12        android:theme="@style/Theme_ChapterC2"
13        tools:targetApi="31">
14             <activity>
```

Lanciando la nostra app vediamo la nuova icona nell'elenco delle app:



7. Esercizio: Stilizziamo Hero Generator

Seguendo i principi del Material Design fai queste modifiche a Hero Generator:

- Cambia il tema dell'app con dei colori personalizzati
- Crea ed applica uno Style per la\le textview che hanno lo stesso scopo
- Riposiziona le view e cambiane margin e padding, per trovare un bilanciamento tra UI e UX
- Aggiungi delle icone dove pensi sia utile o necessario
- Modifica campi di testo in maniera tale che siano Outlined

C.3 – Liste

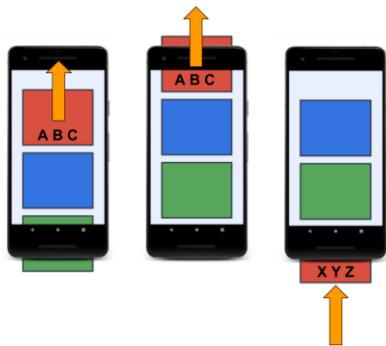
Difficilmente troveremo un'app che non include qualche forma di lista: dai post di un social network ai video di youtube, le liste sono il metodo di esposizione preferito per contenuti numerosi che condividono un formato.

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.

Creare un nuovo progetto con una Empty Activity.

1. RecyclerView

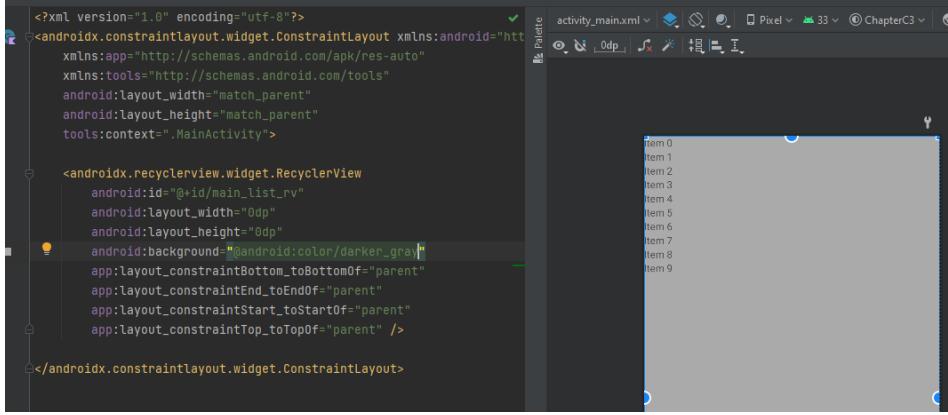
Il widget che si usa per mostrare liste è la *RecyclerView*, e già dal nome, è possibile intuire che è una view che ricicla. Le liste infatti possono essere molto impegnative in merito all'impiego delle risorse hardware: per ovviare a questo problema, in questa view è stato sviluppato un sistema per 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 standby ed eventualmente "riciclate" per caricarne altre visibili.



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:



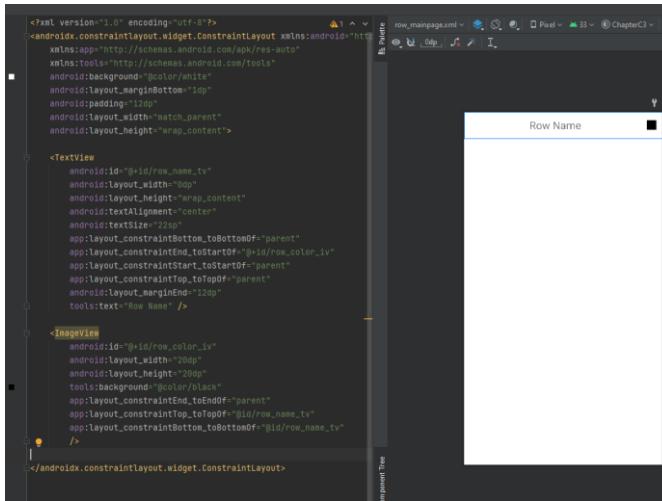
È consigliato che, come ScrollView, RecyclerView abbia delle dimensioni predefinite così da poter gestire al meglio il suo contenuto – evitare `wrap_content` quando possibile.

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

2. Layout riga

Aggiungiamo il layout che andrà a definire ogni singola riga.

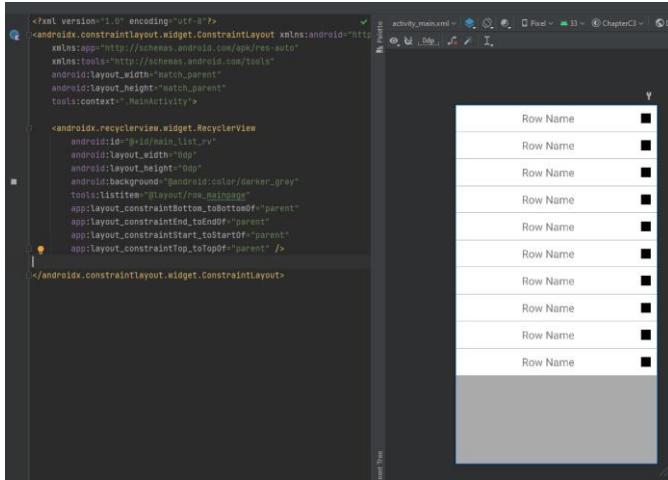
Crea un nuovo file XML nella directory `res/layout` e aggiungi al nome il prefisso **row** per differenziarlo dagli altri layout. Sarà un layout rettangolare, con dentro un campo di testo e un'immagine.



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`



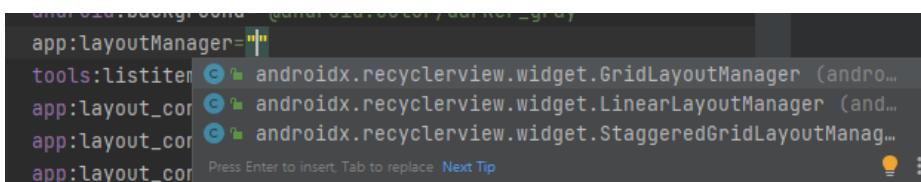
Il colore grigio che avevamo impostato prima come sfondo alla RecyclerView ora funge da separatore, dovuto al fatto che il layout di riga ha un margin bottom.

3. 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 sceglieremo il formato lineare, e non avendo bisogno di particolari customizzazioni andiamo ad impostare il LayoutManager di default.



4. 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.

```
import android.graphics.Color  
  
fun randomColor() = (50 .. 200).random()  
  
data class PrettyItem(  
    val id: Int = (1 .. 999).random(),  
    val color: Int = Color.rgb(randomColor(), randomColor(), randomColor())  
)
```

In MainActivity, crea un dataset di 50 elementi PrettyItem

```
class MainActivity : AppCompatActivity() {  
  
    private val dataset = List(50) { PrettyItem() }  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        val binding = ActivityMainBinding.inflate(layoutInflater)  
  
        binding.run { this@MainActivity@binding  
            setContentView(root)  
        }  
    }  
}
```

5. 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

```
class MainListAdapter(private val elements: List<PrettyItem>) {  
}
```

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

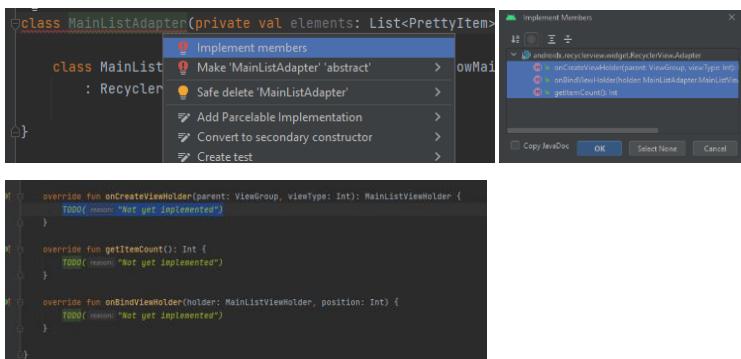
```
class MainListAdapter(private val elements: List<PrettyItem>)  
  
    class MainListViewHolder(private val itemBinding: RowMainpageBinding)  
        : RecyclerView.ViewHolder(itemBinding.root) {  
            fun bind(element: PrettyItem) {  
                TODO(reason: "Not yet implemented")  
            }  
        }  
    }
```

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

```
class MainListAdapter(private val elements: List<PrettyItem>) : RecyclerView.Adapter<MainListAdapter.MainListViewHolder>() {

    class MainListViewHolder(private val itemBinding: RowMainpageBinding)
        : RecyclerView.ViewHolder(itemBinding.root) {
        fun bind(element: PrettyItem) {
            TODO(reason: "Not yet implemented")
        }
    }
}
```

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



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 affinché l'utente veda sempre qualcosa nella lista
- **onBindViewHolder** viene lanciato quando una riga diventa visibile, popolandone i campi. Qui faremo il wiring.
- **getCount** indica il numero degli elementi che saranno disponibili nella lista

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

```
override fun getItemCount(): Int = elements.size
```

onCreateViewHolder richiede come output un MainListViewHolder, crealo:

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

```
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): MainListViewHolder {
        val itemBinding =
            RowMainpageBinding.inflate(LayoutInflater.from(parent.context), parent, false)
        return MainListViewHolder(itemBinding)
    }
```

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

```
    override fun onBindViewHolder(holder: MainListViewHolder, position: Int) {
        holder.bind(elements[position])
    }
```

Ora l'Adapter è completo.

Codice:

```
import android.view.LayoutInflater
import android.view.ViewGroup
import androidx.recyclerview.widget.RecyclerView
import com.example.chapterc3.databinding.RowMainpageBinding

class MainListAdapter(private val elements: List<PrettyItem>) :
    RecyclerView.Adapter<MainListAdapter.MainListViewHolder>() {

    class MainListViewHolder(private val itemBinding: RowMainpageBinding)
        : RecyclerView.ViewHolder(itemBinding.root) {
        fun bind(element: PrettyItem) {
            TODO("Not yet implemented")
        }
    }

    override fun getItemCount(): Int = elements.size

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
        MainListViewHolder {
        val itemBinding =
            RowMainpageBinding
                .inflate(LayoutInflater.from(parent.context), parent, false)
        return MainListViewHolder(itemBinding)
    }

    override fun onBindViewHolder(holder: MainListViewHolder, position: Int) {
        holder.bind(elements[position])
    }
}
```

6. 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.

```
fun bind(element: PrettyItem) {
    itemBinding.run { this: RowMainpageBinding
        rowNameTv.text = "ID: ${element.id}"
        rowColorIv.setBackgroundColor(element.color)
    }
}
```

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:

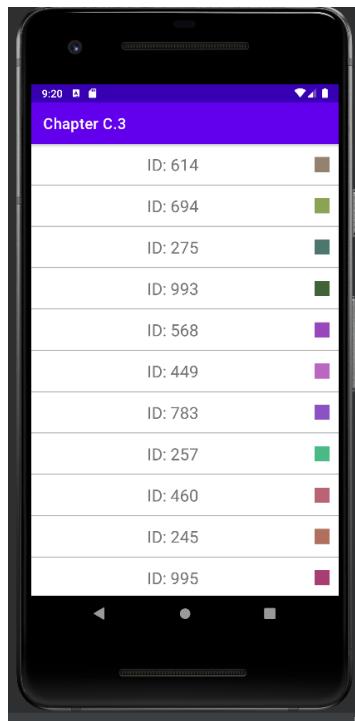
```
private val dataset = List( size=50) { PrettyItem() }
private val mainListAdapter = MainListAdapter(dataset)

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    val binding = ActivityMainBinding.inflate(layoutInflater)

    binding.run { this:ActivityMainBinding
        setContentView(root)

        mainListRV.adapter = mainListAdapter
    }
}
```

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



7. 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.

8. 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)



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

```
private val dataset = mutableListOf( size: 50 ) { PrettyItem() }
```

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

```
private val dataset = generatePrettyItems()
private fun generatePrettyItems() = mutableListOf( size: 50 ) { PrettyItem() }
```

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

```
binding.run { this: ActivityMainBinding  
    setContentView(root)  
  
    mainListRv.adapter = mainListAdapter  
  
    mainRefreshFab.setOnClickListener { it: View ->  
        dataset.apply { this: MutableList<PrettyItem>  
            clear()  
            addAll(generatePrettyItems())  
        }  
        mainListAdapter.notifyDataSetChanged()  
    }  
}
```

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.

```
notifyItemChanged(int),
notifyItemInserted(int),
notifyItemRemoved(int),
notifyItemRangeChanged(int, int),
notifyItemRangeInserted(int, int),
notifyItemRangeRemoved(int, int)
```

9. 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, `mutable`. 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:

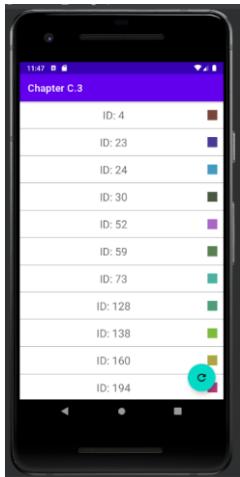
```
private fun generatePrettyItems() =
    MutableList(size = 50) { PrettyItem() }.apply { this: MutableList<PrettyItem>
        sortBy { prettyItem -> prettyItem.id }
    }
```

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`.

```
private fun generatePrettyItems() =
    MutableList(size = 50) { PrettyItem() }
        .apply { sortBy { it.id } }
```

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

```
private fun generatePrettyItems() =  
    mutableListOf(size: 50) { PrettyItem() }  
        .apply { sortBy { it.id } }  
        .filter { it.id % 2 == 0 }
```

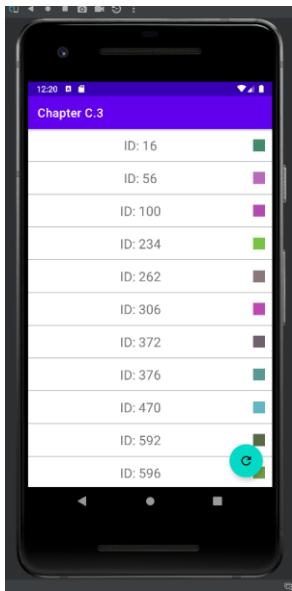
filter ritorna una nuova *List*, quindi andremo ad accodarla alle altre modifiche.

Filter richiede come input la funzione *{ inputValue -> 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*:

```
private fun generatePrettyItems() =  
    mutableListOf(size: 50) { PrettyItem() }  
        .apply { sortBy { it.id } }  
        .filter { it.id % 2 == 0 }  
        .toMutableList()
```



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**.

10. 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:

```
private fun generateByFibonacci() : List<PrettyItem> {  
    val fibs = listOf(0, 1, 1, 2, 3, 5, 8, 13, 21, 34)
```

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

```
private fun generateByFibonacci(): List<PrettyItem> {
    val fibs = listOf(0, 1, 1, 2, 3, 5, 8, 13, 21, 34)

    val result = mutableListOf<PrettyItem>()
    fibs.forEach { it:Int }
        result.add(PrettyItem(it))
    }

    return result
}
```

.map permette di sintetizzare questa comune architettura:

```
private fun generateByFibonacci(): List<PrettyItem> {
    val fibs = listOf(0, 1, 1, 2, 3, 5, 8, 13, 21, 34)

    return fibs.map { PrettyItem(it) }
}
```

Possiamo così procedere a rendere *inline* la nostra funzione

```
private fun generateByFibonacci() =
    listOf(0, 1, 1, 2, 3, 5, 8, 13, 21, 34).map { PrettyItem(it) }
```

11. Immagini con Glide

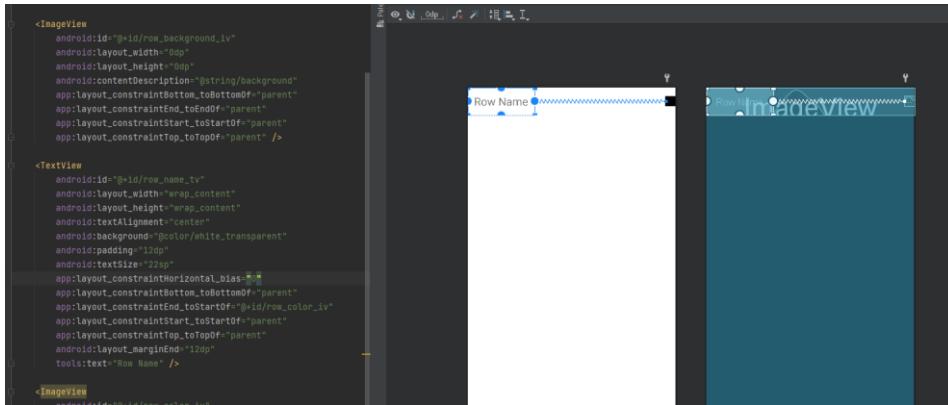
La gestione delle immagini è un tema particolarmente complicato in merito alle risorse di sistema. Per questo sono state sviluppate delle librerie apposite, in questo corso useremo Glide.

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:

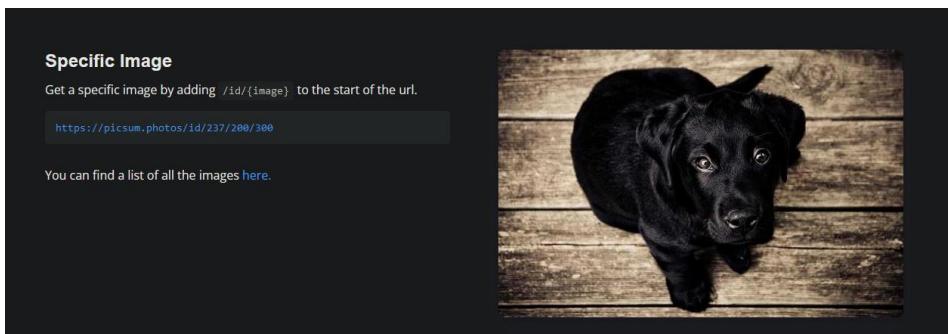
- Aggiungi una ImageView che fungerà da sfondo
- Cambia il background della textView per farla risaltare sul nuovo sfondo, spostala dal centro alla sinistra del layout, e rendila wrap_content



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 casuali!

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

```
xmlns:tools="http://schemas.android.com/tools">  
    <uses-permission android:name="android.permission.INTERNET"/>  
  
    <application>
```

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

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

```
private fun randomColorComponent() = (50 .. 200).random()  
  
data class PrettyItem(  
    val id: Int = (1 .. 999).random(),  
    val color: Int = Color.rgb(randomColorComponent(), randomColorComponent(), randomColorComponent()),  
    val imageUrl: String = "https://picsum.photos/id/$id/500/100"  
)
```

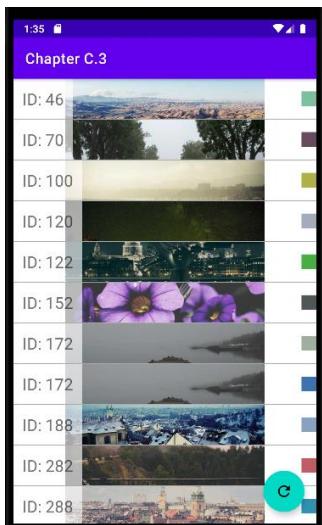
Nell'adapter, aggiungi l'invocazione a Glide:

```
class MainListViewHolder(private val itemBinding: RowMainpageCardBinding) :  
    RecyclerView.ViewHolder(itemBinding.root) {  
  
    fun bind(element: PrettyItem) {  
        itemBinding.run { this: RowMainpageCardBinding  
  
            rowNameTv.text = "ID: ${element.id}"  
            rowColorIv.setBackgroundColor(element.color)  
  
            Glide.with(root).RequestManager  
                .load(element.imageUrl).RequestBuilder<Drawable>  
                .into(rowBackgroundIV).^run  
        }  
    }  
}
```

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.

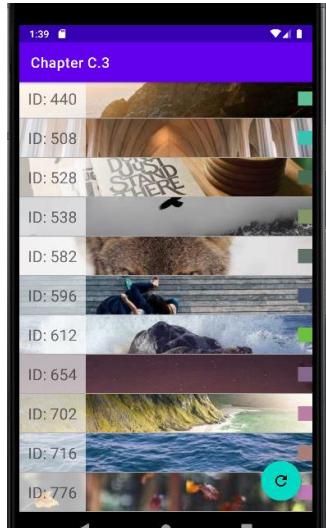
Lancia l'app:



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()*:

```
Glide.with(root) RequestManager
    .load(element.imageUrl) RequestBuilder<Drawable>
        .centerCrop()
        .into(rowBackgroundIv) ^run
```

Questo ritaglierà l'immagine al minimo per farla entrare nella nostra imageview, centrata.



12. Esercizio: Card collection 2

Aggiungi a card collection:

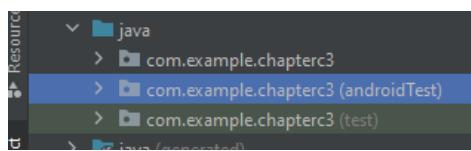
- Per ogni riga, un'immagine dinamica che cambia in base a ID
- L'utente puo' ordinare la lista tramite due pulsanti, uno per ID e l'altro per Valore
- Spingendo un terzo pulsante, è possibile sostituire un elemento randomico del dataset con uno nuovo.

C.4 - Integration test

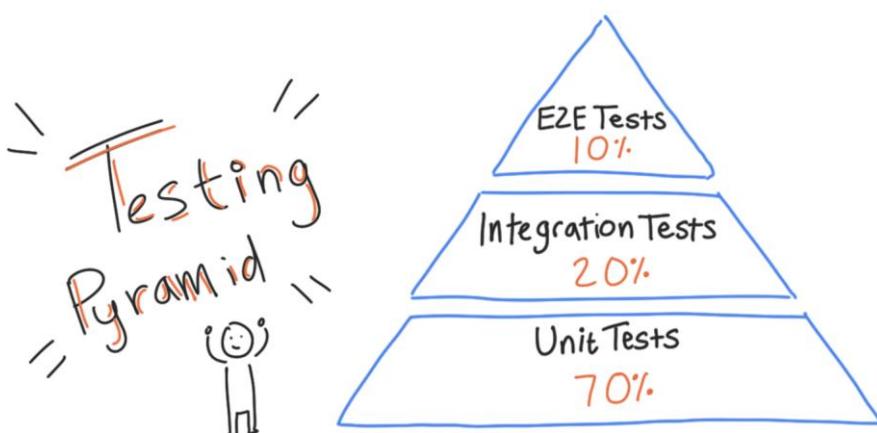
Finora abbiamo testato il codice in Unit test che, sebbene abbiano molte qualità, non consentono però di testare funzionalità che possono fallire anche se ogni componente, formalmente, si comporta in maniera corretta.

Gli Integration Tests che vedremo in questo capitolo, insieme ai End-to-End tests, sono considerati **Instrumented Tests**, perché gireranno non sulla nostra macchina dev, ma su di un device o emulatore Android.

Questi test sono inoltre salvati all'interno del progetto nel package `androidTest` per differenziarli dagli Unit Test.



Gli Instrumented test solitamente scambiano la velocità di esecuzione degli Unit test per una fedeltà maggiore all'esperienza dell'utente. In una immagine:



Mentre nella parte bassa della piramide avremo tanti test a fedeltà bassa, più saliamo meno sarà il numero di test, che avranno anche maggiore fedeltà e tempo di esecuzione.

1. Environment setup

Come gli Unit test, importiamo delle librerie con le quali lavoreremo.

In `build.gradle :app` sostituisci gli import `androidTestImplementation` con:

```

    androidTestImplementation "androidx.test.ext:junit:1.1.5"
    androidTestImplementation "org.mockito:mockito-core:2.28.2"
    androidTestImplementation "com.linkedin.dexmaker:dexmaker-mockito:2.28.1"
    androidTestImplementation "com.google.truth:truth:1.1.2"
    androidTestImplementation "androidx.navigation:navigation-testing:2.5.3"
    androidTestImplementation "androidx.test.espresso:espresso-core:3.5.1"
    androidTestImplementation("androidx.test.espresso:espresso-contrib:3.5.1") {
        exclude group: 'org.checkerframework', module: 'checker'
    }
    /// Fragment testing code should not be included in the main code.
    // Check https://issuetracker.google.com/128612536
    debugImplementation "androidx.fragment:fragment-testing:1.5.6"
    implementation "androidx.test:core:1.5.0"
    //

```

Questi import copriranno la maggior parte degli Instrumented Test del corso.

Bisogna anche modificare alcune impostazioni del nostro device\emulatore.

Andando su **Settings > Developer options** disattiva le seguenti animazioni:

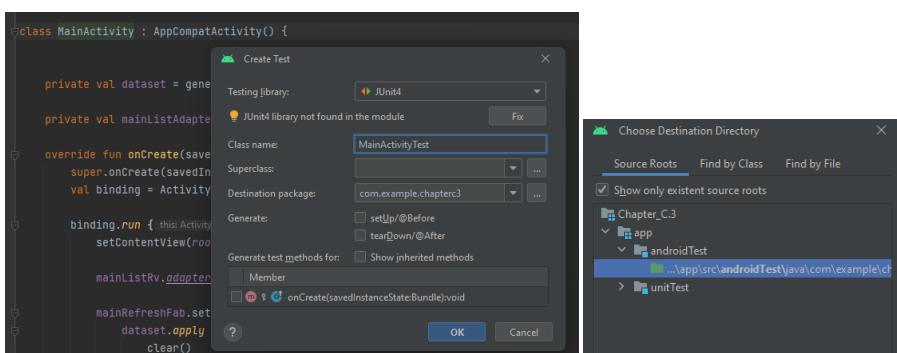
- Window animation scale
- Transition animation scale
- Animator duration scale

2. Introduzione a IT su Android

Gli Integration Tests verificano l'interazione tra diverse classi, per assicurarsi che si comportino come previsto quando vengono utilizzate insieme.

Un modo di sviluppare Integration Tests è dunque quello di testare funzionalità, come ad esempio **l'abilità di rigenerare la lista**.

Creiamo un test per MainActivity, assicurandoci di salvarlo nella directory *androidTest*



Aggiungiamo le seguenti annotazioni prima della dichiarazione della classe test:

```

    @MediumTest
    @RunWith(AndroidJUnit4::class)
    class MainActivityTest {

```

MediumTest identifica lo scopo del test, e sarà utile al sistema per assegnargli risorse, e a noi per differenziare tra i vari test.

Il nostro test ora dovrà testare le funzionalità di Main Activity, ma ancora una volta non ci è possibile farlo direttamente, altrimenti ci ritroveremmo a testare tutto il framework.

La libreria di test ci offre degli *scenari* che possiamo creare per simulare delle situazioni.

```
@MediumTest
@RunWith(AndroidJUnit4::class)
class MainActivityTest {

    lateinit var scenario: ActivityScenario<MainActivity>

    @Before
    fun before() {
        scenario = ActivityScenario.launch(MainActivity::class.java)
    }

    @After
    fun after() {
        scenario.close()
    }
}
```

Abbiamo creato uno scenario che simula la Main Activity.

Ora siamo pronti ad effettuare i nostri test!

3. Espresso

<https://developer.android.com/training/testing/espresso/basics>

Useremo la libreria di test **Espresso** che similmente a Truth ha una struttura dichiarativa, ma oltre alle assertions ci permette di fare anche delle azioni, come cliccare su un pulsante o inserire del testo in un campo editabile.

Espresso lavora su 3 categorie:

ViewMatchers – targettano la view

ViewActions – effettuano azioni

ViewAssertions – condizionali per la riuscita del test

```
@Test
fun fabIsClickable() {
    onView(withId(R.id.main_refresh_fab)) //withId() is a ViewMatcher
        .perform(click()) //click() is a ViewAction
        .check(matches(isDisplayed())) //matches() is a ViewAssertion
}
```

Lanciando il test, vedremo che la nostra app sarà eseguirà velocemente le interazioni che abbiamo scritto – cerca la view, click, controlla – il cui risultato sarà poi disponibile nell'IDE come per gli Unit Test.

Controlliamo che la pressione del FAB effettivamente cambi il dataset.

- In Main Activity, rendi *dataset* pubblico.
- In Main Activity test, salva un riferimento a *dataset*

```
lateinit var scenario: ActivityScenario<MainActivity>
lateinit var dataset: List<PrettyItem>

@Before
fun before() {
    scenario = ActivityScenario.launch(MainActivity::class.java)
    scenario.onActivity { it: MainActivity! ->
        dataset = it.dataset
    }
}
```

Scenario.onActivity ci dà un riferimento all'activity appena è pronta, permettendoci di interagire con i suoi componenti.

Poi in un nuovo test:

```
@Test
fun fabClick_listRefreshed() {
    val oldDataset = mutableListOf<PrettyItem>().apply { addAll(dataset) }
    onView(withId(R.id.main_refresh_fab)).perform(click())
    assertThat(oldDataset).isNotEqualTo(dataset)
}
```

- Crea una Deep Copy di dataset in *oldDataset*. Questo è necessario perché altrimenti otterremmo solo un riferimento all'oggetto *dataset*, e alla fine ci ritroveremmo a compararlo con se stesso!
 - Approfondimento: [Deep Copy vs Shallow Copy](#) (baeldung.com)
- Click sul FAB
- Check che il nuovo dataset sia diverso dal vecchio

4. Recyclerview Utilities

Vogliamo anche assicurarci che il nuovo dataset sia correttamente mostrato nella lista. Potremmo controllare l'id del primo elemento, ma facciamo un passo in più decidendo di controllare l'id dell'ultimo elemento.

Per fare questo però, bisogna scrollare la lista fino alla fine.

Le **RecyclerView utilities** permettono di effettuare azioni sulle liste, come scroll e azioni targettate a una specifica riga

- `scrollTo()` - Scrolls to the matched View, if it exists.
- `scrollToHolder()` - Scrolls to the matched View Holder, if it exists.
- `scrollToPosition()` - Scrolls to a specific position.
- `actionOnHolderItem()` - Performs a View Action on a matched View Holder.
- `actionOnItem()` - Performs a View Action on a matched View.
- `actionOnItemAtPosition()` - Performs a ViewAction on a view at a specific position.

Sono tutte actions, quindi devono essere inserite come ViewActions.

Continuando il test di prima, scriviamo un'azione che scorre la lista fino a un membro che abbia come id l'ultimo del dataset:

```
val textToMatch = "ID: ${dataset.last().id}"  
  
onView(withId(R.id.main_list_rv))  
    .perform(  
        RecyclerViewActions.scrollTo<MainListAdapter.MainListViewHolder>(  
            hasDescendant(withText(textToMatch))  
        )  
    )
```

- `hasDescendant` permette di andare a fondo nella gerarchia del layout:
RecyclerView > ViewHolder > ConstraintLayout > TextView

Infine, controlliamo con una assertion Espresso se è visibile. Ecco il test completo:

```
@Test
fun fabClick_listRefreshed() {
    val oldDataset = mutableListOf<PrettyItem>().apply { addAll(dataset) }
    onView(withId(R.id.main_refresh_fab)).perform(click())
    assertThat(dataset).isNotEqualTo(oldDataset)

    val textToMatch = "ID: ${dataset.last().id}"

    onView(withId(R.id.main_list_rv))
        .perform(
            RecyclerViewActions.scrollTo<MainListAdapter.MainListViewHolder>(
                hasDescendant(withText(textToMatch))
            )
        )

    onView(withText(textToMatch))
        .check(matches(isDisplayed()))
}
```

5. Esercizio: Test Card Collection

Tramite Integration Test, testa le funzionalità della Card Collection. Non dimenticarti degli Unit Test!

D.1 – Activity e Intents

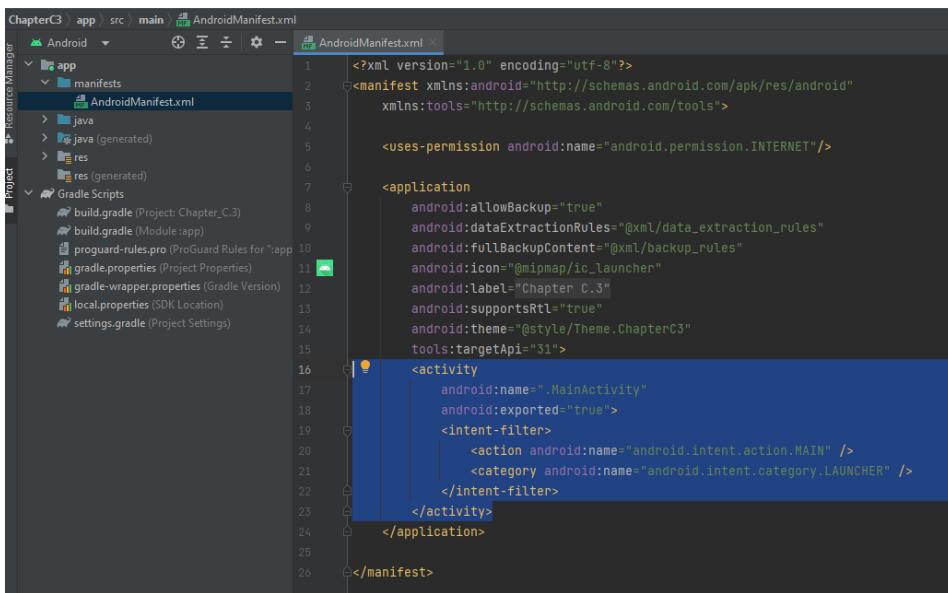
1. Activity

Finora abbiamo visto come modificare una singola classe, la Main Activity.

L'activity rappresenta una funzionalità dell'app, e spesso un Entry point per l'utente.

Quando la nostra app viene lanciata dal sistema, Android inizializza l'activity che si occupa di disegnare l'interfaccia. Durante il suo ciclo di vita, l'activity ci offre delle callback come *onCreated* con le quali la programmiamo.

Tutte le activity devono essere dichiarate all'interno del manifest, tramite l'unico parametro obbligatorio *name*. L'activity che verrà lanciata quando l'utente clicca sull'icona dal sistema, sarà impostata tramite i parametri *action-Main* e *category-Launcher* dentro *intent-filter*.



The screenshot shows the Android Studio interface with the project 'ChapterC3' open. The left sidebar shows the project structure with 'app' selected. Inside 'app', there's a 'manifests' folder containing 'AndroidManifest.xml'. The main editor window displays the XML code for 'AndroidManifest.xml'. The code defines a manifest with various permissions and an application section. Within the application section, there is one activity declaration. This activity is named '.MainActivity' and has its 'exported' attribute set to 'true'. It also contains an 'intent-filter' block with two entries: '*action android:name="android.intent.action.MAIN"*' and '*category android:name="android.intent.category.LAUNCHER"*'. The code is color-coded for readability, with tags in blue and attributes in green.

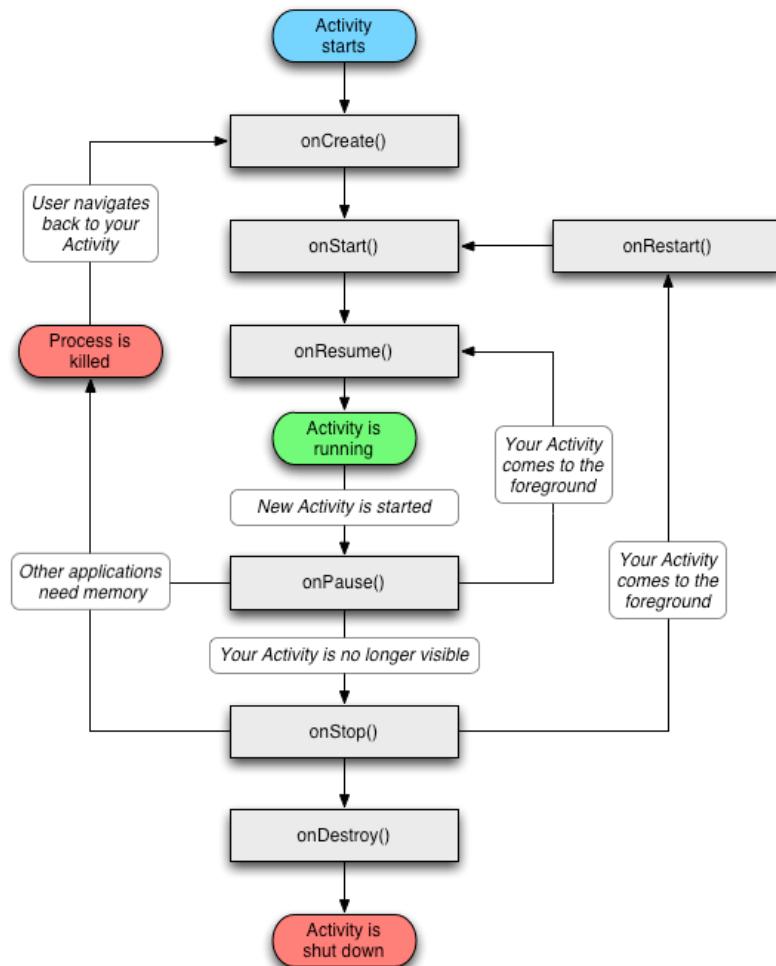
```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <uses-permission android:name="android.permission.INTERNET"/>

    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="Chapter C.3"
        android:supportsRtl="true"
        android:theme="@style/Theme.ChapterCJ"
        tools:targetApi="31">

        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

L'activity possiede un *lifecycle* – implementa l'interfaccia *lifecycleOwner* – che rappresenta i vari stati in cui si puo' trovare durante la sua esistenza. È infatti sottoposta a varie interazioni con l'utente, che puo' decidere da un momento all'altro di farla partire, tornare indietro, metterla in background o chiuderla definitivamente.



Lifecycle dell'activity

Possiamo sfruttare le varie callback per gestire le features che intendiamo sviluppare in base allo stato dell'activity. `onCreate()` richiede di bindare il layout, e il setup delle view viene impostato. `onResume()` avviene quando l'activity diventa visibile dopo il binding, o ritorna visibile dopo essere stata messa in background.

La documentazione che spiega a fondo questi stati è disponibile qui:

<https://developer.android.com/guide/components/activities/intro-activities#mtal>

2. Controllo delle risorse da parte del sistema

È importante chiarificare il concetto che Android si occupa molto aggressivamente della gestione delle risorse. Ci sono delle limitazioni nelle attività che possono essere effettuate, e molte richiedono che l'utente stia attivamente visualizzando l'app.

Questo implica che se la nostra app ha delle schermate che non sono visibili all'utente in quel momento, magari perché in un punto precedente dell'app, queste possono essere soggette alla distruzione da parte del sistema nel caso ci sia una necessità di risorse. Ad esempio, se l'utente riceve una telefonata o decide di scattare una foto, la nostra app potrebbe essere distrutta.

3. Intents

Le activity sono in grado di comunicare tra di loro, non solo all'interno della stessa app, ma anche tra le app installate nel sistema operativo. Pensiamo alla funzionalità di share, che ci permette di mandare un'immagine o un testo da un'app all'altra.

Eppure, le activity non sono legate strettamente tra di loro, non avendo modo diretto di comunicare.

Per questo si utilizzano gli **Intents**

Un Intent è un oggetto messaggero, che puoi utilizzare per richiedere un'azione da un altro componente.

Ci sono 3 casi fondamentali per l'utilizzo di Intenti:

- **Starting an activity**
 - Far partire e inviare dati ad altre activity, tramite `startActivity()`
- **Starting a service**
 - Un Service è un componente che effettua operazioni in background senza UI. ([Daemon](#))
- **Delivering a broadcast**
 - Un Broadcast è un messaggio intercettabile da tutte le app

Vediamo come è possibile utilizzarli:

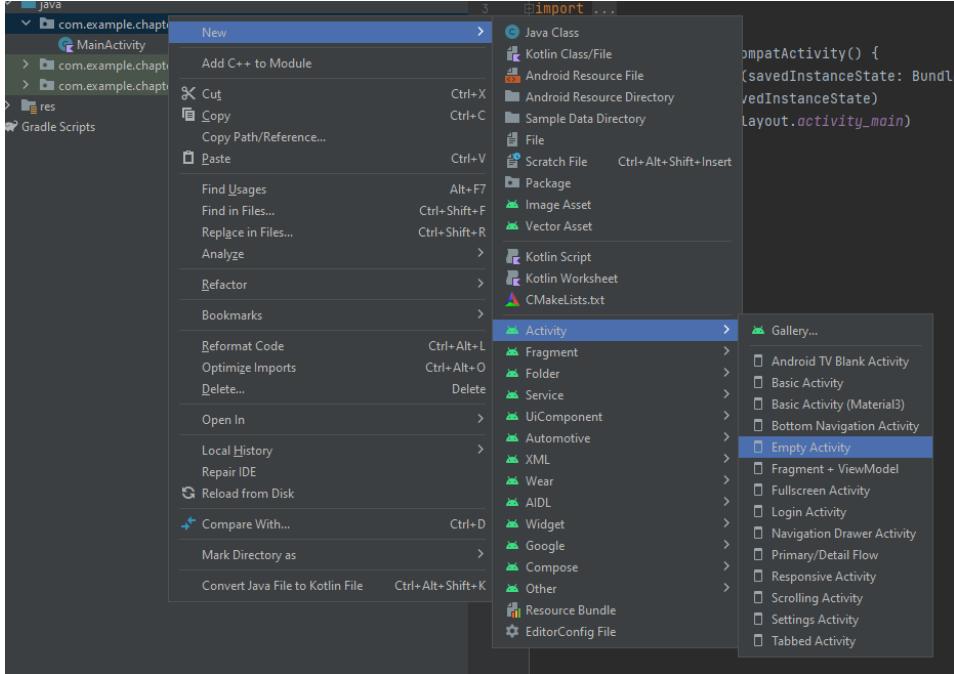
[Naviga verso un'altra activity](#)

Crea una nuova app con una Empty Activity

- Aggiungi un FAB un basso a dx

```
<com.google.android.material.floatingactionbutton.FloatingActionButton
    android:id="@+id/main_fab"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_margin="20dp"
    android:src="@android:drawable/ic_media_next"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintBottom_toBottomOf="parent" />
```

Crea una nuova Empty activity con il wizard:



Questo aggiungerà anche i necessari campi nel manifest.

Nel layout della nuova activity, aggiungi un testo che specifica che siamo in un'altra activity:

The screenshot shows the XML layout file for the second activity. It includes a TextView with the text "This is the second activity". The preview window on the right shows the text displayed in the activity's UI.

```

<androidx.constraintlayout.widget.ConstraintLayout>
    <TextView
        android:id="@+id/second_tv"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="This is the second activity"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>

```

Vai in Main Activity, effettua il binding. Aggiungi un clicklistener al fab

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        ActivityMainBinding.inflate(layoutInflater).run { this: ActivityMainBinding
            setContentView(root)

            mainFab.setOnClickListener { it: View! }
```

Ora vediamo come è possibile, alla pressione del fab, cambiare activity.

- Crea un Intent che prende come input un contesto (partenza) e una classe (destinazione)
- Lancialo tramite la funzione `startActivity`, disponibile qui perché siamo all'interno di una Activity:

```
mainFab.setOnClickListener { it: View!
    val navIntent = Intent(packageContext, this@MainActivity, MainActivity2::class.java)
    startActivity(navIntent)
}
```

Ora cliccando il FAB navigheremo verso la nuova activity.

Decidiamo di impostare il testo della seconda activity inviando dati dalla prima activity

Aggiungi all'Intent il testo che vuoi inviare alla seconda activity:

```
mainFab.setOnClickListener { it: View!  
    val navIntent = Intent(packageContext, this@MainActivity, MainActivity2::class.java)  
        .apply { this.intent  
            putExtra(name: "custom_text", value: "Sotto la pancia la capra è stanca")  
        }  
    startActivity(navIntent)  
}
```

Nell'activity di destinazione ora è possibile recuperare questo extra tramite il *name* da noi inserito – gli extra sono una mappa chiave-valore

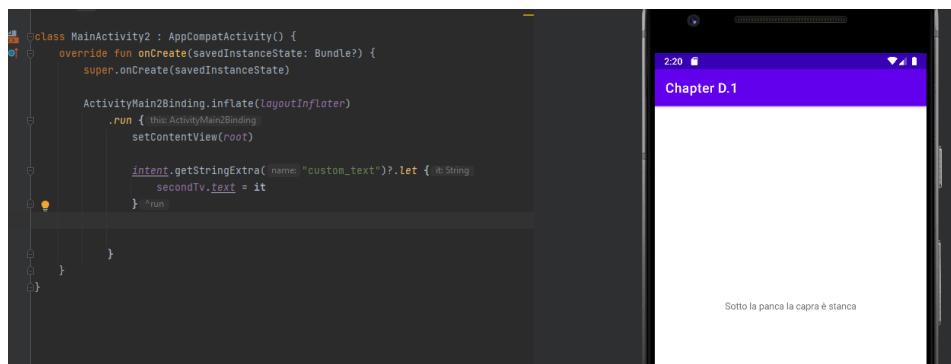
- Effettua il binding nella seconda activity

Activity ha un campo *intent* dove c'è il riferimento all'intent che ha lanciato l'activity.

Il dato che abbiamo messo negli extra è una stringa: utilizziamo la funzione `intent.getStringExtra(name)` per recuperarlo.

```
intent.getStringExtra(name: "custom_text")?.let { it:  
    secondTv.text = it  
}
```

La mappa degli extra non assicura che abbia la chiave *custom_text* che gli stiamo fornendo, quindi il risultato sarà nullable. Per prendere il risultato in caso di non-null, utilizziamo `?.let` e sfruttiamo subito la nostra stringa per popolare il testo della view *secondTv*.



Condividi un contenuto

Vediamo ora come è possibile inviare dati ad un'altra app tramite la funzionalità di *share* di Android.

Nella seconda activity

- Aggiungi un FAB con l'icona @android:drawable/ic_menu_share

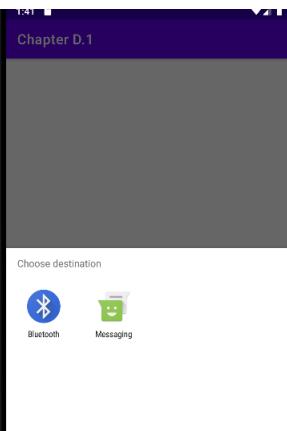
Ora condivideremo il testo del messaggio ad altre app, come Whatsapp o Gmail.

- Recupera il testo del messaggio tramite *textview.text.toString()*

L'intent da creare avrà una *action* che definisce l'azione da effettuare, e *type* che definirà le app che sono in grado di ricevere la condivisione

La funzione *Intent.createChooser* specifica al sistema che dovrà esporre il contenuto tramite picker.

Conterrà l'intent con i dati a le azioni, e un titolo da dare al picker.



The screenshot shows an Android application interface. At the top, there's a purple header bar with the text "Chapter D.1". Below it is a dark grey content area. In the center, a white modal dialog titled "Choose destination" is displayed. The dialog contains two items: "Bluetooth" with a blue circular icon and "Messaging" with a green circular icon. On the left side of the screen, there is a vertical stack of code snippets. The top snippet is in Java:

```
super.onCreate(savedInstanceState)
ActivityMain2Binding.inflate(layoutInflater).run { this: ActivityMain2Binding
    setContentView(root)

    secondFab.setOnClickListener { v: View ->
        val message = secondTv.text.toString() //get text from view

        val sendIntent = Intent().apply { this: Intent
            action = Intent.ACTION_SEND
            type = "text/plain"
            putExtra(Intent.EXTRA_TEXT, message) // add data to Intent
        }

        val shareIntent = Intent.createChooser( //Create an android share picker
            sendIntent,
            title = "Choose destination"
        )

        startActivity(shareIntent) // fire!
    }
}
```

4. Esercizio: Condividi Card Collection Power

Aggiungi queste features a Card collection

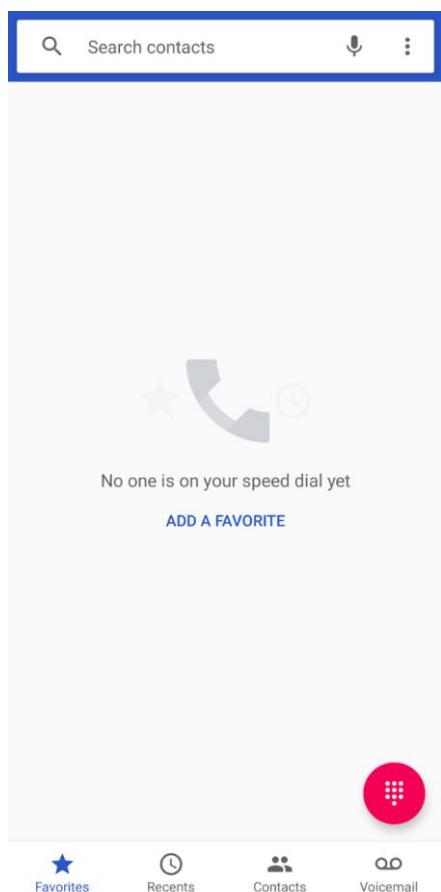
- La main activity contiene anche un campo che somma tutti i valori della lista per mostrare il totale.
- Condividi il valore di potenza del tuo mazzo con gli altri!

Fragment e Navigation

1. Fragment

Abbiamo visto come navigare tra activities, ma solitamente un'app non ha bisogno di diverse activity per ogni schermata proposta all'utente. A questo servono i **Fragments**

Un fragment è un pezzo di UI utilizzabile da una o più activities.



In questa schermata, cliccare una delle sezioni del menu orizzontale non cambia activity, bensì' ne modifica il contenuto alternando tra Fragments, mantenendo il menu, il FAB e il campo di ricerca nell'activity.

Il lifecycle del Fragment è simile a quello dell'Activity, ma leggermente più complesso. I fragment hanno lifecycle diversi per il fragment stesso e la view (layout) la lui contenuta, in maniera tale da ottimizzare le risorse occupate quando ci si trova ad avere numerosi Fragment in memoria.

Lifecycle State	Callback
CREATED	onCreate()
	onCreateView()
	onViewCreated()
STARTED	onStart()
RESUMED	onResume()
STARTED	onPause()
CREATED	onStop()
	onDestroyView()
DESTROYED	onDestroy()

Una delle differenze fondamentali è la fase di creazione: mentre nell'activity utilizzerebbero onCreate per effettuare linflate ed il binding del layout, nel Fragment è necessario effettuare linflate in onCreateView, ed il binding in onViewCreated.

Setup del progetto

Aggiungiamo il Navigation Component e il plugin Safe Args che ci aiuterà nello sviluppo.

Nel *build.gradle:project*, aggiungi prima di *plugins*:

```
buildscript {  
    ext {  
        nav_version = "2.5.3"  
    }  
    dependencies {  
        classpath("androidx.navigation:navigation-safe-args-gradle-plugin:$nav_version")  
    }  
}
```

Aggiungi queste dipendenze in *build.gradle:app*:

```
implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"  
implementation "androidx.navigation:navigation-ui-ktx:$nav_version"
```

Sempre nello stesso file, vai all'inizio. Nel tag *plugins*, aggiungi:

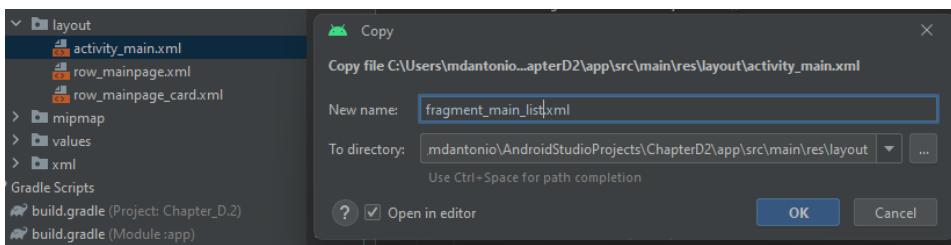
```
id 'androidx.navigation.safeargs.kotlin'
```

Da Activity a Fragment

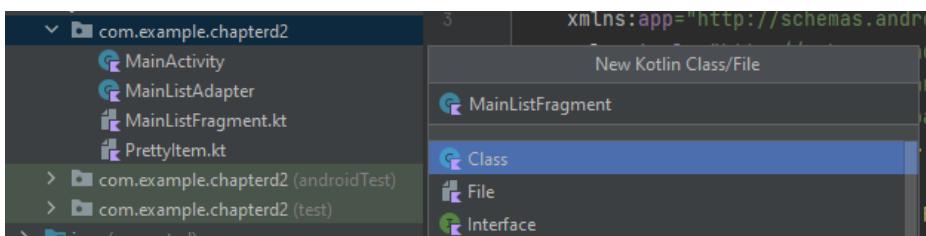
Trasforma l'esercizio lista da Activity a Fragment

Per farlo dovremo portare la UI in un fragment, e trasformare la Main Activity in un contenitore di frammenti.

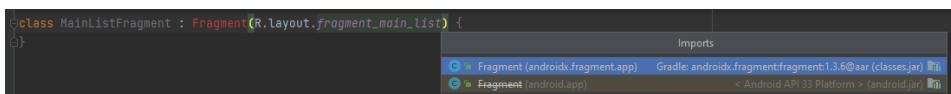
Comincia copiando il layout della main activity in un nuovo layout il cui nome comincia per **fragment_**
Ctrl-C → Ctrl-V nell'albero files:



Ora crea un nuovo Fragment. Dovrai farlo a mano in quanto, alla data odierna, i modelli di Android Studio non sono aggiornati con le ultime versioni di Fragment.



Questa classe estenderà **Fragment(R.layout)**, alla quale passeremo il layout appena creato



Questa versione di Fragment effettua automaticamente l'`inflate`, quindi non sarà per ora necessario implementare `onCreateView`.

Passiamo al binding facendo l'override di `onViewCreated`. Non dovendo più effettuare l'`inflate`, utilizzeremo un metodo alternativo del viewBinding che permette di applicare un binding a un `viewGroup`:

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    FragmentMainListBinding.bind(view).run { this: FragmentMainListBinding
        }
    }
}
```

Questo è possibile perché, leggendo la doc, `view` è la view root ritornata dall'inflate onCreateView.

Qui copieremo il codice responsabile di popolare le view che avevamo scritto nella main activity.

```
FragmentMainListBinding.bind(view).run { this: FragmentMainListBinding
    mainListRv.adapter = mainListAdapter

    mainRefreshFab.setOnClickListener { it: View!
        dataset.apply {
            clear()
            addAll(generatePrettyItems())
        }
        mainListAdapter.notifyDataSetChanged()
    }
}
```

Copiamo anche il dataset, l'adapter e la funzione di generazione, e abbiamo completato la migrazione.

```
class MainListFragment : Fragment(R.layout.fragment_main_list) {

    val dataset = generatePrettyItems()
    private val mainListAdapter = MainListAdapter(dataset)

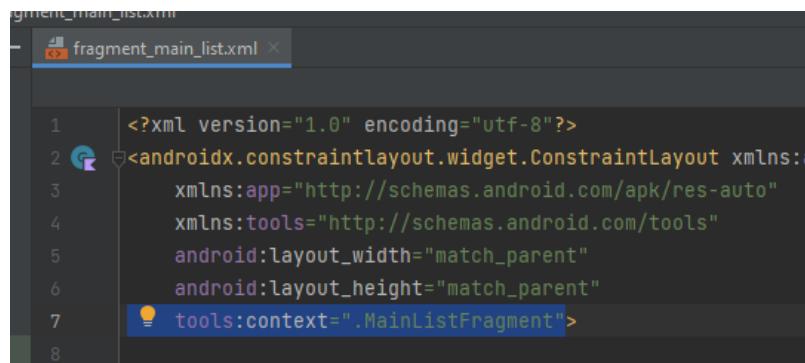
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        FragmentMainListBinding.bind(view).run { this: FragmentMainListBinding
            mainListRv.adapter = mainListAdapter

            mainRefreshFab.setOnClickListener { it: View ->
                dataset.apply { this: MutableList<PrettyItem>
                    clear()
                    addAll(generatePrettyItems())
                }
                mainListAdapter.notifyDataSetChanged()
            }
        }
    }

    private fun generatePrettyItems() =
        mutableListOf( size 50 ) { PrettyItem() }
            .apply { sortBy { it.id } } MutableList<PrettyItem>
            .filter { it.id % 2 == 0 } List<PrettyItem>
            .toMutableList()
    }
}
```

Per semplificare il lavoro e ottenere delle preview del fragment è necessario cambiare nel layout il parametro `tools:context` puntandolo al Fragment che abbiamo creato:



```
fragment_main_list.xml
1 <?xml version="1.0" encoding="utf-8"?>
2 <androidx.constraintlayout.widget.ConstraintLayout xmlns:a
3     xmlns:app="http://schemas.android.com/apk/res-auto"
4     xmlns:tools="http://schemas.android.com/tools"
5     android:layout_width="match_parent"
6     android:layout_height="match_parent"
7     tools:context=".MainListFragment">
8 
```

Main Activity come contenitore di Fragments

Prima modifica MainActivity rimuovendo tutti i wiring e tenendo solo il binding:

```
class MainActivity : AppCompatActivity() {  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        val binding = ActivityMainBinding.inflate(layoutInflater)  
  
        binding.run { this:ActivityMainBinding  
            setContentView(root)  
        }  
    }  
}
```

Poi modifica il layout di MainActivity :

- Rimuovi il FAB
- Cambia ConstraintLayout in FrameLayout
 - FrameLayout è un ViewGroup molto leggero utilizzato spesso come parent di View singole.
- Sostituisci la RecyclerView con una **FragmentContainerView**
 - <androidx.fragment.app.FragmentContainerView
 android:id="@+id/nav_host_fragment"
 android:name="androidx.navigation.fragment.NavHostFragment"
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 app:defaultNavHost="true"
 app:navGraph="@navigation/nav_graph"/>

Nel paragrafo successivo capiremo cos'è e come si crea il nav_graph necessario a questa view.

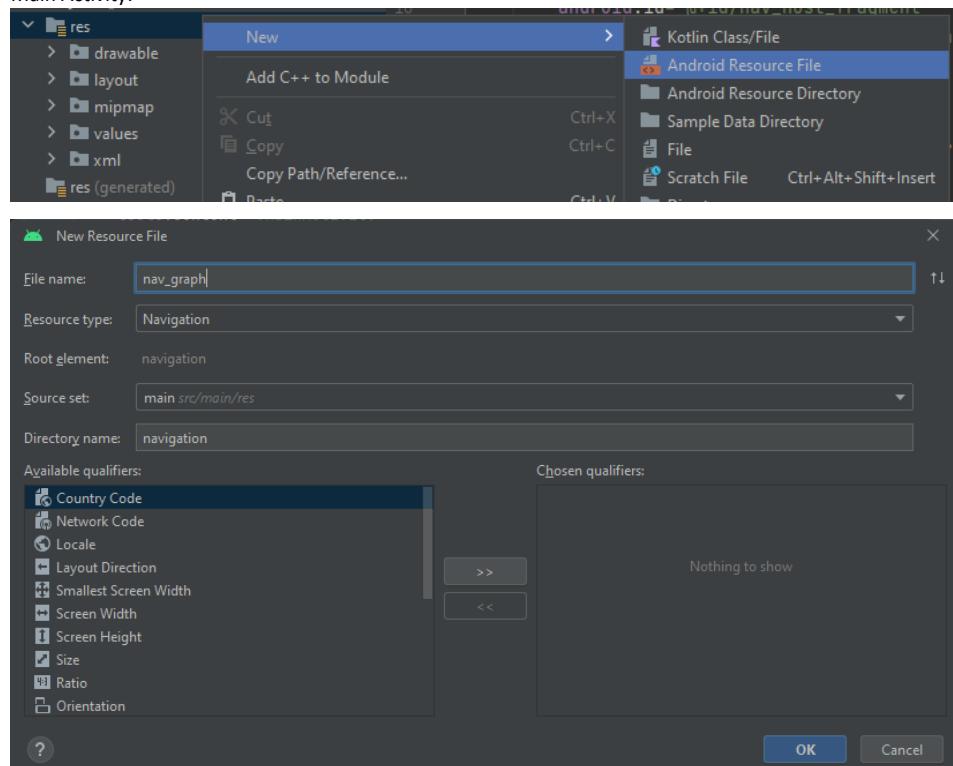
```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    xmlns:tools="http://schemas.android.com/tools"  
    xmlns:app="http://schemas.android.com/apk/res-auto"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"  
    tools:context=".MainActivity">  
  
    <androidx.fragment.app.FragmentContainerView  
        android:id="@+id/nav_host_fragment"  
        android:name="androidx.navigation.fragment.NavHostFragment"  
        android:layout_width="match_parent"  
        android:layout_height="match_parent"  
        app:defaultNavHost="true"  
        app:navGraph="@navigation/nav_graph"/>  
  
</FrameLayout>
```

2. Navigation Graph

I Fragment possono essere numerosi, e vengono utilizzati per avere una migliore gestione della navigazione tra le varie schermate dell'app. Ci sono alcuni modi per navigare tra i fragment, noi utilizzeremo il **Navigation Graph (per gli amici Nav Graph)**.

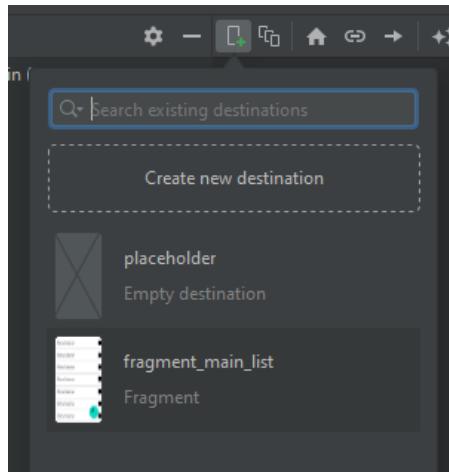
Il Navigation Graph è una rappresentazione schematica della navigazione tra Fragments.

Crea un nuovo file risorse di tipo Navigation, con nome *nav_graph* – lo stesso impostato nel layout della Main Activity:

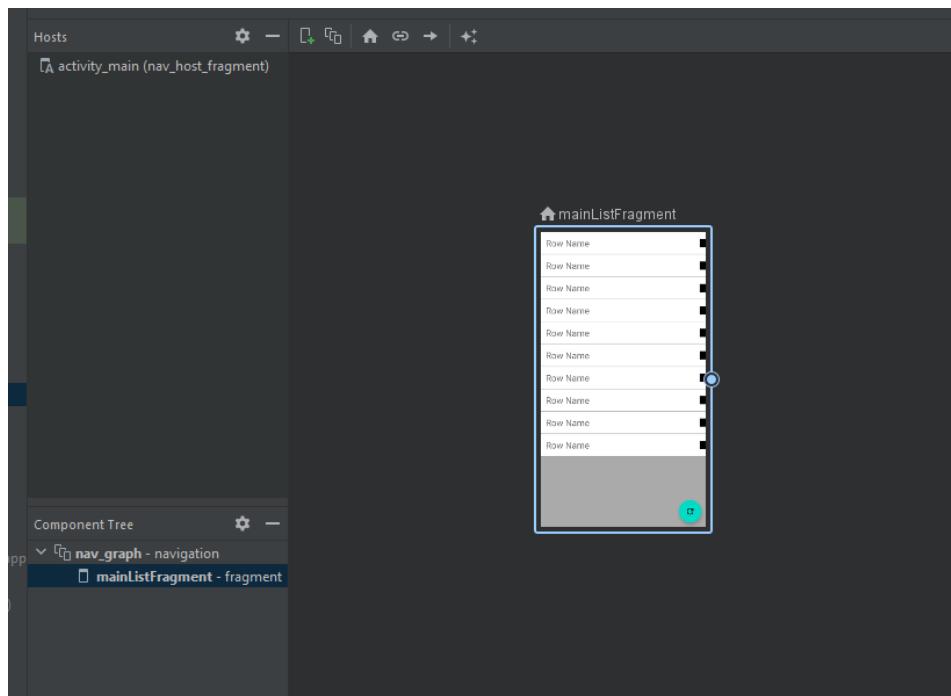


Si aprirà un nuovo editor grafico, che vedrà a sinistra l'host *activity_main* dove abbiamo impostato un riferimento a questo navgraph.

Aggiungi una nuova destinazione con il pulsante verde +, selezionando il Fragment che abbiamo creato



Ora il fragment verrà aggiunto al grafico, ed essendo il primo otterrà anche il flag *home* (l'icona della casa) che lo identifica come il primo del flusso.



3. Navigazione tra frammenti

Vediamo ora come è possibile passare da un Fragment a un altro, rimanendo sempre all'interno della Main Activity.

Layout e NavGraph

Crea una nuova classe Kotlin MainDetailFragment che useremo per mostrare il dettaglio di un oggetto della lista, poi utilizza l'assistente dell'IDE per creare un nuovo layout con il nome fragment_main_detail



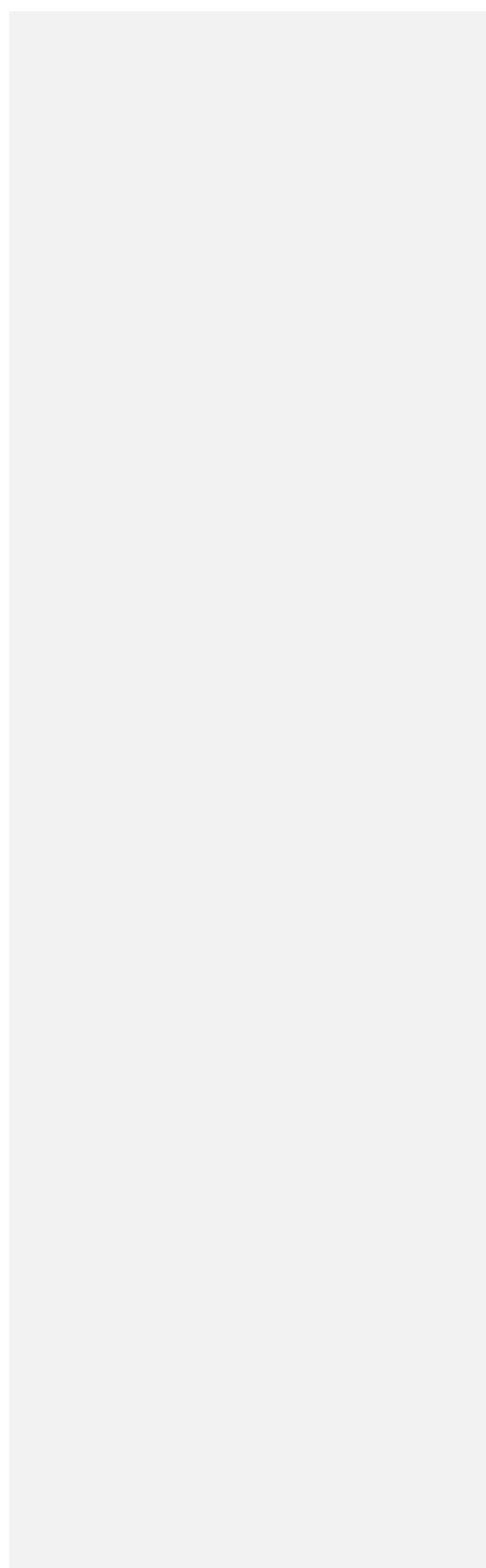
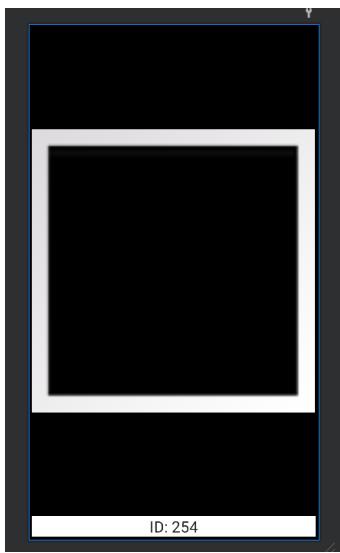
Copia il seguente layout:

```
<?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="4dp"
    tools:background="@color/black"
    tools:context=".MainDetailFragment">

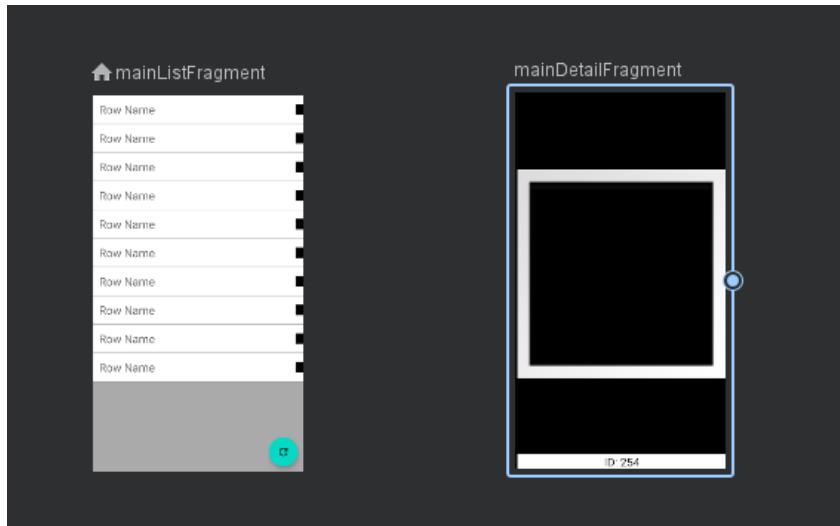
    <ImageView
        android:id="@+id/detail_image_iv"
        android:layout_width="0dp"
        android:layout_height="0dp"
        android:contentDescription="@string/image"
        app:layout_constraintBottom_toTopOf="@+id/detail_id_tv"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        tools:src="@android:drawable/gallery_thumb" />

    <TextView
        android:id="@+id/detail_id_tv"
        style="@style/TextAppearance.AppCompat.Large"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:background="@color/white"
        android:textAlignment="center"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        tools:text="ID: 254" />

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

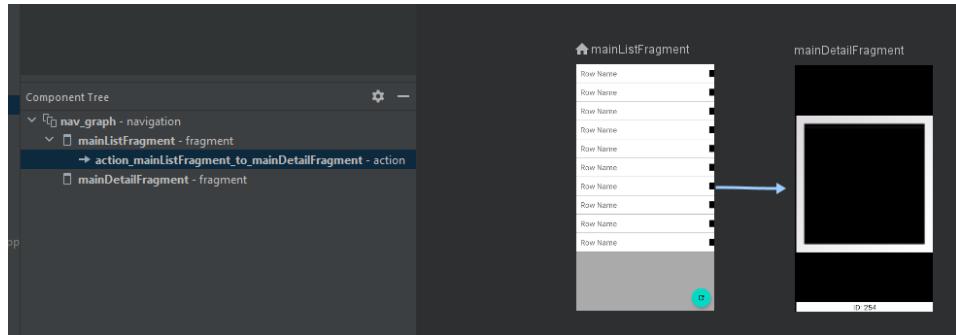


Aggiungi questo Fragment nel nav_graph, alla destra del fragment precedente



Ora trascina il lato destro di `mainListFragment` dentro `mainDetailFragment`: questo creerà una fraccia, chiamata **Action**





Useremo questa action come destinazione per la nostra navigazione.

Adapter

Ora introduciamo nell'adapter della lista la possibilità di cliccare un singolo elemento

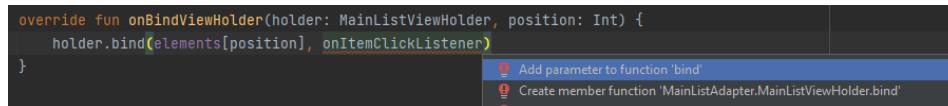
Andiamo a modificare il costruttore di MainListAdapter per includere una funzione Listener che comunicherà al fragment quando l'utente clicca sulla riga:

```
class MainListAdapter(  
    private val elements: List<PrettyItem>,  
    private val onItemClickListener: (element: PrettyItem) -> Unit  
) : RecyclerView.Adapter<MainListAdapter.MainListViewHolder>() {
```

onItemClickListener è una funzione anonima, che prende in input un elemento PrettyItem e non ha output (Unit).

Aggiungiamo questo parametro alla funzione *bind* del ViewHolder:

```
override fun onBindViewHolder(holder: MainListViewHolder, position: Int) {  
    holder.bind(elements[position], onItemClickListener)  
}
```



Ora all'interno del binding, possiamo applicare un clickListener alla view root (l'intera riga) e inviare l'elemento che è stato cliccato a onItemClickListener

```
fun bind(element: PrettyItem, onItemClickListener: (element: PrettyItem) -> Unit) {  
    itemBinding.run { this: RowMainpageCardBinding  
  
        root.setOnClickListener { it: View  
            onItemClickListener(element)  
        }  
    }  
}
```

Ora quando una riga viene cliccata, *onItemClickListener* ci invierà l'oggetto cliccato.

Avendo modificato il costruttore di Adapter, dobbiamo anche cambiare come è invocato in MainListFragment, aggiungendo il nuovo clickListener con delle parentesi graffe:

```
class MainListFragment : Fragment(R.layout.fragment_main_list) {  
  
    val dataset = generatePrettyItems()  
    private val mainListAdapter = MainListAdapter(dataset) { it: PrettyItem  
        //Row has been clicked  
    }  
}
```

Implementazione in Fragment

Aggiungiamo la funzione di navigazione, che sarà avviata al click della riga.

In MainListFragment, nel listener dell'adapter, invoca questa classe:

```
private val mainListAdapter = MainListAdapter(dataset) { it: PrettyItem
    val action = MainListDirections.actionMainListFragmentToMainDetailFragment()
```

Il plugin SafeArgs ha generato per noi i riferimenti all'action che abbiamo creato nel nav_graph:

- *MainListDirections* : il nome del fragment di partenza + Directions
- *actionMyName* : il nome dell'action in formato *actionPartenzaToDestinazione*

Ora utilizziamo l'action come parametro per il NavController, una classe che si occupa della navigazione

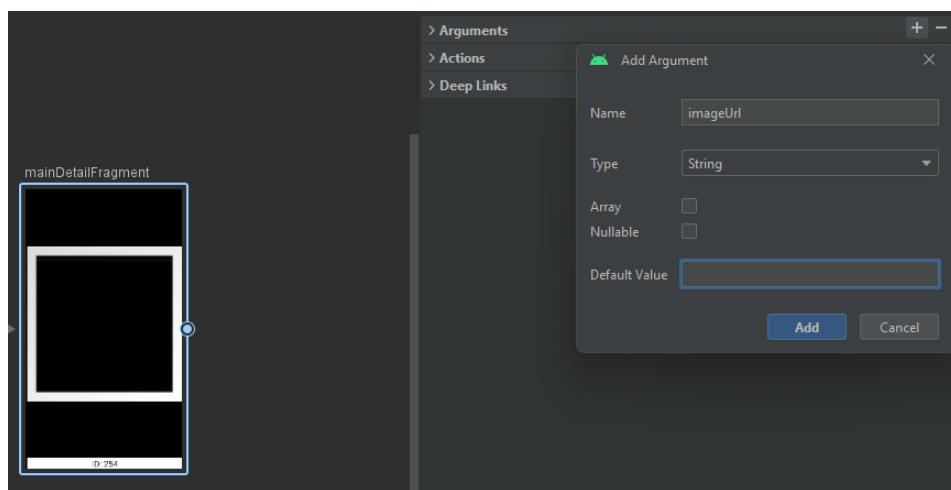
```
private val mainListAdapter = MainListAdapter(dataset) { it: PrettyItem
    val action = MainListDirections.actionMainListFragmentToMainDetailFragment()
    findNavController().navigate(action)
```

Cliccando una riga ora verrà visualizzato *MainDetailFragment*, ma sarà completamente bianco, in quanto non abbiamo legato ai campi nessun dato.

4. Inviare dati tra frammenti

Ora inviamo a DetailFragment l'indirizzo dell'immagine, così da poter essere caricata nella ImageView.

Nel NavGraph, seleziona Detail Fragment cliccandoci. Nella colonna di destra, alla riga **arguments** creane di nuovi con il pulsante +



Crea due arguments:

- 1 – id, String
- 2 - imageUrl, String

Poi fai una nuova build con Build -> Rebuild Project

Ora la funzione *actionStartToDestination* generata da SafeArgs richiederà questi due parametri.

```
private val mainListAdapter = MainListAdapter(dataset) { it: PrettyItem
    val action = MainListFragmentDirections.actionMainListFragmentToMainDetailFragment()
    findNavController().navigate(action)
}
```

Dobbiamo ora passare questi parametri alla funzione. In precedenza, abbiamo creato il listener di MainListAdapter in modo che ritorni l'oggetto cliccato, quindi possiamo utilizzarlo per inviare i parametri al fragment che sarà mostrato al click.

```
private val mainListAdapter = MainListAdapter(dataset) { it: PrettyItem
    val action = MainListFragmentDirections.actionMainListFragmentToMainDetailFragment(
        it.id.toString(), it.imageUrl
    )
    findNavController().navigate(action)
}
```

Ora bisogna recuperare i dati nel fragment di destinazione, generati per noi in una data class `FragmentNameArgs`. Useremo la funzione `by navArgs()`

```
val args: MainDetailFragmentArgs by navArgs()
```

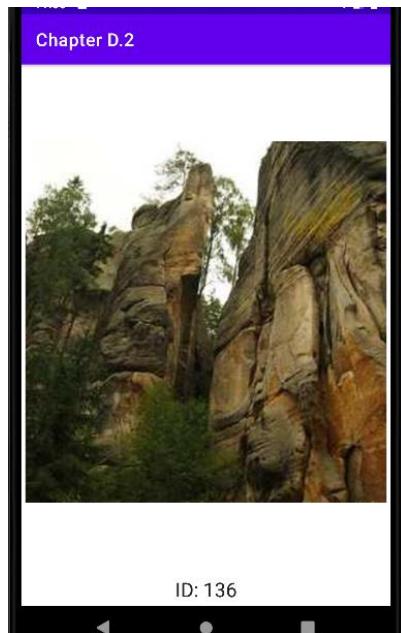
- Questa è una *delegated property*: proprietà che vengono inizializzate solo una volta.
In particolare stiamo utilizzando l'inizializzazione *by lazy*, dove la variabile sarà inizializzata solo la prima volta che viene acceduta, utilizzando la funzione a destra di *by*.
<https://kotlinlang.org/docs/delegated-properties.html>

Questo oggetto contiene i parametri che abbiamo inviato dall'altro Fragment, e possiamo utilizzarlo per popolare il nostro layout:

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    val args: MainDetailFragmentArgs by navArgs()

    FragmentMainDetailBinding.bind(view).run { this: FragmentMainDetailBinding
        detailIdTv.text = "ID: ${args.id}"
        Glide.with(fragment: this@MainDetailFragment) RequestManager
            .load(args.imageUrl) RequestBuilder<Drawable>
            .into(detailImageIv) ^run
    }
}
```



5. Setup Barra di Navigazione

È importante impostare la barra di navigazione presente in alto nella nostra app, perché non tutti i device hanno la possibilità di tornare indietro con il pulsante di sistema.

Questa barra è chiamata **Action Bar** e viene utilizzata per pulsanti che permettono azioni di navigazione o di setup.

La funzione per impostare la navigazione è attualmente buggata, utilizzeremo un workaround.

All'inizio di MainActivity aggiungi una variabile navController:

```
private lateinit var navController: NavController  
  
override fun onCreate(savedInstanceState: Bundle?) {
```

Nel binding:

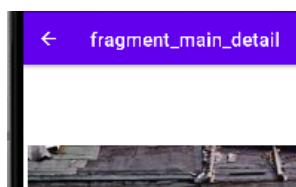
```
binding.run { this: ActivityMainBinding  
    setContentView(root)  
    navHostFragment.post {  
        navController = navHostFragment.findNavController()  
        setupActionBarWithNavController(navController)  
    } ^run  
}
```

- .post attende che navHostFragment abbia completato linflate
- Recupera e salva il navController tramite findNavController
- setupActionBarWithNavController informerà l'Action Bar di utilizzare come riferimento il navController

Nell'activity, effettua l'override della funzione onSupportNavigateUp, comunicando al sistema Android di utilizzare il nostro navController come riferimento.

```
override fun onSupportNavigateUp(): Boolean {  
    return navController.navigateUp() || super.onSupportNavigateUp()  
}
```

Ora nella pagina di dettaglio sarà presente la freccia di navigazione.



6. Esercizio: Card Collection detail

- In Card Collection, dovrà essere possibile cliccare su una carta nella lista e navigare a una pagina di dettaglio dove vengono mostrati ID, valore e immagine.
- Aggiungi ad ogni elemento del dataset un campo “color” con un colore randomico
- Nel dettaglio, lo sfondo o la cornice dell’immagine deve essere del colore “color”
- Imposta correttamente la navigazione dell’Action Bar

D.3 – Mockito e End-To-End test

Abbiamo convertito l'activity in un contenitore di fragment, e abbiamo spostato le funzionalità in MainFragment. Ora i nostri test non funzioneranno più, quindi aggiorniamoli

1. Testare i Fragment

Similmente all'activity, andiamo a creare uno scenario di test per il Fragment.

Dopo aver creato la classe di test, utilizza la funzione `launchFragmentInContainer` per creare un'activity temporanea che conterrà il nostro Fragment

```
@RunWith(AndroidJUnit4::class)
@MediumTest
class MainListFragmentTest {

    lateinit var scenario: FragmentScenario<MainListFragment>
    lateinit var dataset: List<PrettyItem>

    @Before
    fun setUp() {
        scenario = launchFragmentInContainer<MainListFragment>(Bundle(), R.style.Theme_ChapterC3)
            .apply { this: FragmentScenario<MainListFragment>
                onFragment { it: MainListFragment -
                    dataset = it.dataset
                }
            }
    }

}
```

Poi portiamo da MainActivityTest a MainFragmentTest le funzioni di test che avevamo scritto

```

}

    @Test
    fun fabIsClickable() {
        Espresso.onView(ViewMatchers.withId(R.id.mainFab))
            .perform(ViewActions.click())
            .check(ViewAssertions.matches(ViewMatcher))
    }

    @Test
    fun fabClick_listRefreshed() {
        val oldDataset = mutableListOf<PrettyItem>()
        Espresso.onView(ViewMatchers.withId(R.id.mainFab))
            .perform(ViewActions.click())
        Truth.assertThat(dataset).isNotEqualTo(oldDataset)
    }
}
```

Rimuoviamo `dataset` da `MainActivityTest`, non è più presente e quindi genera errore

```
@MediumTest
@RunWith(AndroidJUnit4::class)
class MainActivityTest {

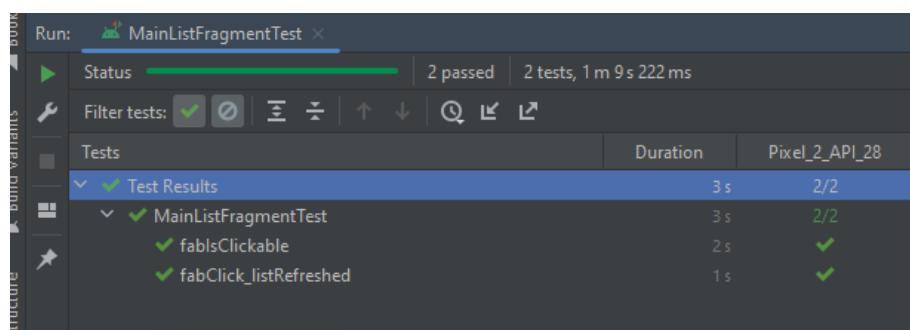
    lateinit var scenario: ActivityScenario<MainActivity>

    @Before
    fun before() {
        scenario = ActivityScenario.launch(MainActivity::class.java)
    }

    @After
    fun after() {
        scenario.close()
    }

}
```

Ora i nostri test funzionano sul fragment



2. Mock e Mockito

Come abbiamo visto i test si occupano di verificare che il codice che scriviamo abbia il comportamento corretto. Gli integration test però spesso richiedono di verificare che il codice che scriviamo produca una interazione con Android corretta. Per questo utilizzeremo un Mock, tramite la libreria **Mockito**

Un test double sono oggetti che sembrano e agiscono come componenti della tua app, ma vengono creati nel tuo test per fornire un comportamento o dati specifici.

Un mock è un tipo di test double che ha aspettative sulle sue interazioni. È un oggetto utilizzato per verificare se vengono applicate le tecniche e i percorsi corretti, ma senza una reale implementazione di metodi o valori.

Testiamo la funzione di navigazione

Testeremo la capacità del nostro codice di inviare al Navigation Component la destinazione corretta con i dati corretti, quando la riga viene cliccata.

Crea un nuovo test dove imposterai un navController *mock* al fragment

```
@Test
fun clickListItem_navigateToDetail() {
    //Test subject (mock)
    val navController = Mockito.mock(NavController::class.java)
    scenario.onFragment { fragment ->
        // Set the graph on the TestNavController
        navController.setGraph(R.navigation.nav_graph)

        // Make the NavController available via the findNavController() APIs
        Navigation.setViewNavController(fragment.requireView(), navController)
    }
}
```

Ora utilizza Espresso con le RecyclerViewActions per cliccare sull'oggetto della lista che abbia come testo "ID: itemId"

```
val firstItem = dataset.first()

onView(withId(R.id.main_list_rv))
    .perform(
        RecyclerViewActions.actionOnItem<MainListAdapter.MainListViewHolder>(
            hasDescendant(withText(text: "ID: ${firstItem.id}")),
            click()
        )
    )
```

Infine, utilizza la funzione `verify` di Mockito per verificare che il navcontroller abbia ricevuto il corretto input di navigazione che ci aspettiamo da questa interazione

```
Mockito.verify(navController)
    .navigate(
        MainListFragmentDirections.actionMainListFragmentToMainDetailFragment(
            firstItem.id.toString(),
            firstItem.imageUrl
        )
    )
```

Verify ritorna l'oggetto mock che stiamo analizzando (`navController`), del quale possiamo utilizzare tutte le funzioni pubbliche, come `navigate`.

Con questa definizione stiamo chiedendo a Mockito di **verificare** che `navController` abbia ricevuto l'input di `navigate` a una specifica `direction`.

Questo ci basta come condizione di test, perché il nostro test non deve validare Android, ma l'interazione della nostra app con i componenti Android.

Il test completo:

```
@Test
fun clickListItem_navigateToDetail() {
    // Test subject (mock)
    val navController = Mockito.mock(NavController::class.java)
    scenario.onFragment { fragment ->
        // Set the graph on the TestNavController
        navController.setGraph(R.navigation.nav_graph)

        // Make the NavController available via the findNavController() APIs
        Navigation.setViewNavController(fragment.requireView(), navController)
    }

    val firstItem = dataset.first()
    onView(withId(R.id.main_list_rv))
        .perform(
            RecyclerActions.actionOnItem<MainListAdapter.MainListViewHolder>(
                hasDescendant(withText("ID: ${firstItem.id}")),
                click()
            )
        )

    Mockito.verify(navController)
        .navigate(
            MainListFragmentDirections.actionMainListFragmentToMainDetailFragment(
                firstItem.id.toString(),
                firstItem.imageUrl
            )
        )
}
```

3. E2E Testing

Mentre gli Integration Test validano una funzionalità, gli E2E test validano un flusso composto da una serie di funzionalità.

Gli End-To-End test validano una serie di funzionalità che lavorano insieme.

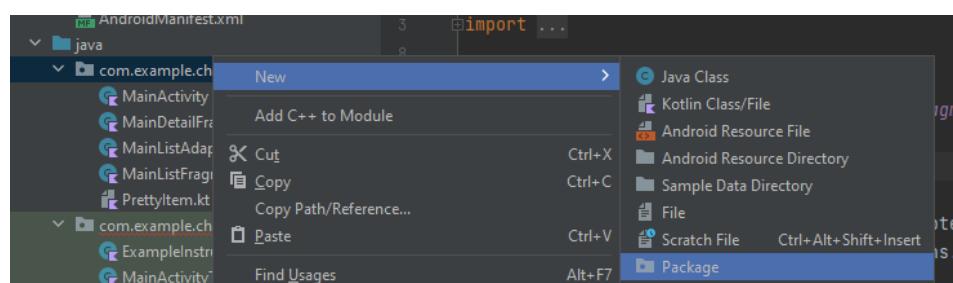
Testano grandi porzioni dell'app, simulano da vicino l'utilizzo reale e quindi sono generalmente lenti. Hanno la massima fedeltà e validano che l'applicazione funzioni effettivamente nel suo insieme.

Creiamo un nuovo test dove verifichiamo che i dati mostrati nella pagina di dettaglio siano quelli della riga cliccata

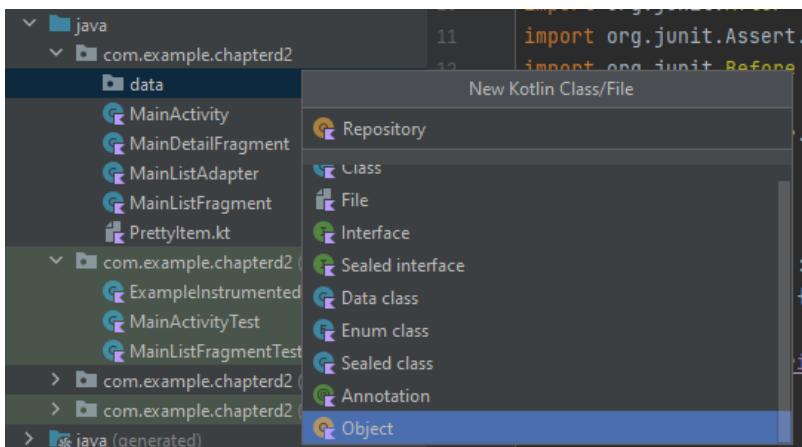
Effettueremo questo test nella MainActivity, per farlo dovremo indicare a Espresso cosa cliccare. Ora la main activity non ha più un riferimento a dataset, quindi non ci è possibile identificare un elemento della lista da cliccare per il test.

Per rimediare, facciamo in modo che sia MainActivity che MainListFragment possano accedere allo stesso dataset

Creare un nuovo package chiamato *data*



Al suo interno, crea un nuovo *object* chiamato **Repository**



Al suo interno:

- Sposta da MainListFragment il valore *dataset* e la funzione *generatePrettyItems*
- Aggiungi una funzione per rigenerare il dataset

```
object Repository {
    val dataset = generatePrettyItems()

    private fun generatePrettyItems() =
        mutableListOf(size: 50) { PrettyItem() }
            .apply { sortBy { it.id } } MutableList<PrettyItem>
            .filter { it.id % 2 == 0 } List<PrettyItem>
            .toMutableList()

    fun regenerateDataset() = dataset.apply { this: MutableList<PrettyItem>
        clear()
        addAll(generatePrettyItems())
    }
}
```

The code block shows the implementation of the 'Repository' object. It contains a 'dataset' property initialized to the result of the 'generatePrettyItems()' function. This function creates a mutable list of size 50, adds a 'PrettyItem' to each slot, sorts the list by 'id', filters items where 'id' is even, and then converts the filtered list back to a mutable list. Additionally, there is a 'regenerateDataset()' function that clears the current 'dataset' and adds all items from the 'generatePrettyItems()' function back into it.

Ora *dataset* e le funzioni relative sono disponibili a chiunque implementerà *Repository*

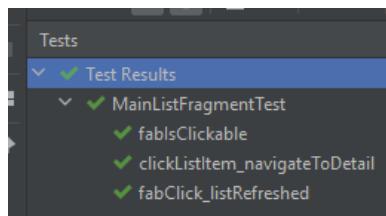
Modifica MainListFragment in modo che possa ottenere il dataset e la rigenerazione della lista da Repository

```
class MainListFragment : Fragment(R.layout.fragment_main_list) {

    private val repository = Repository
    val dataset = repository.dataset

    mainRefreshFab.setOnClickListener { it: View! ->
        repository.regenerateDataset()
        mainListAdapter.notifyDataSetChanged()
    }
}
```

Lancia l'app e i test, per controllare che funzioni tutto correttamente.



Passiamo ora al test E2E

Modifica MainActivityTest con @LargeTest per notificare che queste operazioni possono prendere molto tempo. Questa annotazione è di solito usata solo per gli E2E tests.

```
@LargeTest
@RunWith(AndroidJUnit4::class)
class MainActivityTest {
```

Crea un nuovo test e prendi il primo oggetto della lista dal Repository

```
@Test
fun listClick_detailShow_correctData() {
    val firstItem = Repository.dataset.first()
```

Effettua il click, poi controlla che la pagina di dettaglio venga visualizzata, utilizzando la textView dell'id come check

```
onView(withId(R.id.main_list_rv))
    .perform(
        RecyclerViewActions.actionOnItem<MainListAdapter.MainListViewHolder>(
            hasDescendant(withText("ID: ${firstItem.id}")),
            ViewActions.click()
        )
    )

onView(withId(R.id.detail_id_tv)).check(matches(isDisplayed()))
```

Infine controlla che le view abbiano i corretti dati inseriti

```
onView(withId(R.id.detail_id_tv)).check(matches(withText("ID: ${firstItem.id}")))
```

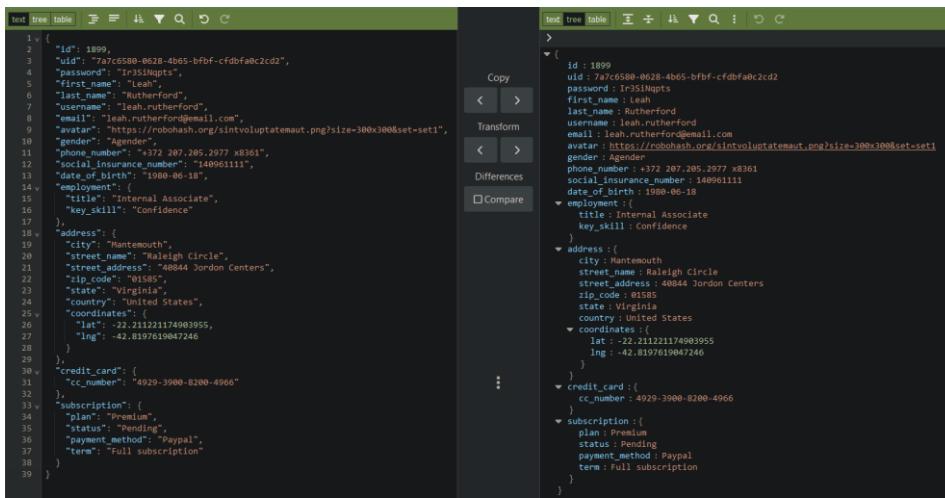
4. Esercizio: Test Card collection

Testa Card Collection ora che è stata convertita a Fragments, assicurati di effettuare tutti i seguenti test:

- Unit Tests
- Integration Tests
- Navigation Test
- E2E tests

E.1 – JSON

Il **JSON** (JavaScript Object Notation) è un *formato dati* tra i piu' diffusi, molto popolare perché **language-agnostic** e leggibile ad occhio umano.



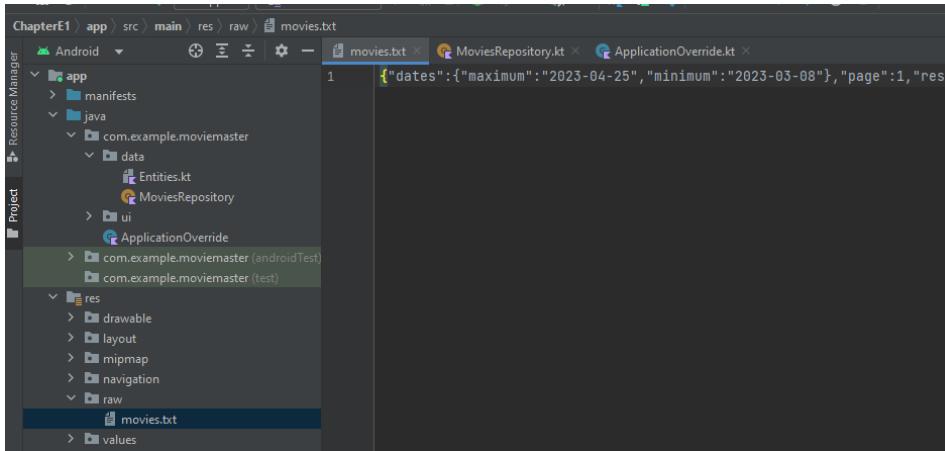
```
1 v [
2   "id": 1899,
3   "uid": "7a7c6580-0628-4b65-bfbf-cfdbfa0c2cd2",
4   "password": "Ir351Nqpts",
5   "first_name": "Leah",
6   "last_name": "Rutherford",
7   "username": "leah.rutherford",
8   "email": "leah.rutherford@gmail.com",
9   "avatar": "https://robohash.org/sintvoluptatemaut.png?size=300x300&set=set1",
10  "gender": "Agender",
11  "phone_number": "+372 207.205.2977 x8361",
12  "social_insurance_number": "140961111",
13  "date_of_birth": "1980-06-18",
14  "employment": [
15    {
16      "title": "Internal Associate",
17      "key_skill": "Confidence"
18    }
19  ],
20  "address": [
21    {
22      "city": "Mantemouth",
23      "street_name": "Raleigh Circle",
24      "street_address": "40844 Jordon Centers",
25      "zip_code": "01585",
26      "state": "Virginia",
27      "country": "United States",
28      "coordinates": [
29        {
30          "lat": -22.211221174903955,
31          "lng": -42.8197619047246
32        }
33      ]
34    }
35  ],
36  "credit_card": [
37    {
38      "cc_number": "4929-3900-8200-4966"
39    }
40  ],
41  "subscription": [
42    {
43      "plan": "Premium",
44      "status": "Pending",
45      "payment_method": "Paypal",
46      "term": "Full subscription"
47    }
48  ]
49 ]
```

```
> [
  id: 1899,
  uid: "7a7c6580-0628-4b65-bfbf-cfdbfa0c2cd2",
  password: Ir351Nqpts,
  first_name: Leah,
  last_name: Rutherford,
  username: leah.rutherford,
  email: leah.rutherford@gmail.com,
  avatar: https://robohash.org/sintvoluptatemaut.png?size=300x300&set=set1,
  gender: Agender,
  phone_number: +372 207.205.2977 x8361,
  social_insurance_number: 140961111,
  date_of_birth: 1980-06-18,
  employment: [
    {
      title: Internal Associate,
      key_skill: Confidence
    }
  ],
  address: [
    {
      city: Mantemouth,
      street_name: Raleigh Circle,
      street_address: 40844 Jordon Centers,
      zip_code: 01585,
      state: Virginia,
      country: United States,
      coordinates: [
        {
          lat: -22.211221174903955,
          lng: -42.8197619047246
        }
      ]
    }
  ],
  credit_card: [
    {
      cc_number: 4929-3900-8200-4966
    }
  ],
  subscription: [
    {
      plan: Premium,
      status: Pending,
      payment_method: Paypal,
      term: Full subscription
    }
  ]
]
```

Un esempio di json generato randomicamente all'indirizzo <https://random-data-api.com/api/v2/users>

La sorgente dati che utilizzeremo è un **file di testo in formato JSON**.

Apri il file *res/raw/movies.txt*:



Puoi copiare il testo in un editor JSON per leggerlo meglio, come <https://jsoneditoronline.org>

I campi del json possono contenere:

- Valori:

```
"id": 502356,  
"original_language": "en",
```

- Array:

```
"genre_ids": [  
    28,  
    53,  
    80  
,
```

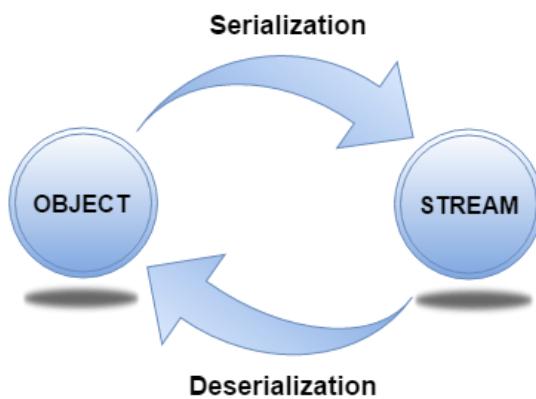
- Oggetti:

```
"dates": {  
    "maximum": "2023-04-25",  
    "minimum": "2023-03-08"  
},
```

1. Serializzazione\Deserializzazione

I Json possono essere convertiti in Istanze di Oggetti con un processo chiamato
Serializzazione/Deserializzazione

Serialization è il processo che converte l'istanza di un oggetto in testo formato json
Deserialization è il processo inverso

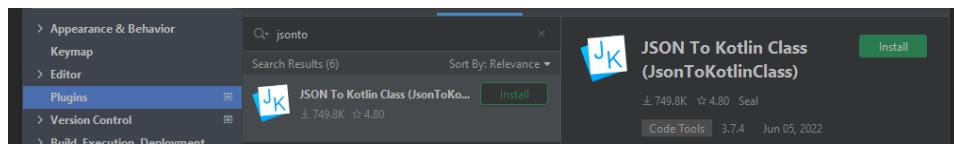


Per convertire una stringa Json a un oggetto strutturato e popolato con tutti i campi e valori presenti nel JSON è necessario creare una data class che funga da modello, che verrà poi utilizzata da una classe di lavoro che effettuerà il pairing tra i dati del JSON e i campi del modello.

2. JsonToKotlin

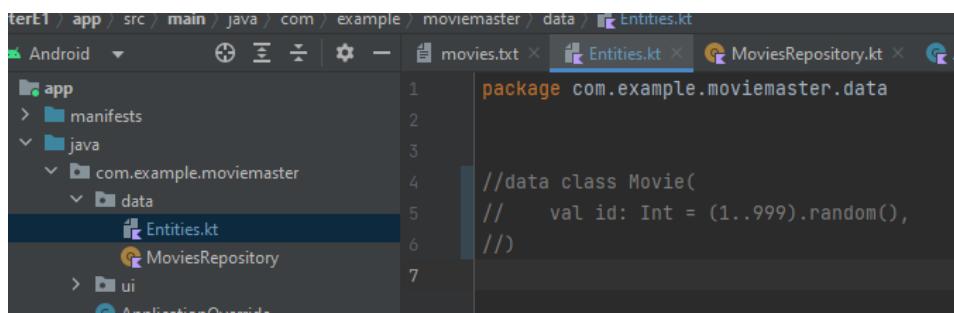
È possibile creare a mano la data class con tutti i relativi membri. Noi useremo un plugin di Android Studio chiamato **JsonToKotlinClass** che ci permetterà di automatizzare il processo evitando errori umani.

File > Settings > Plugins:

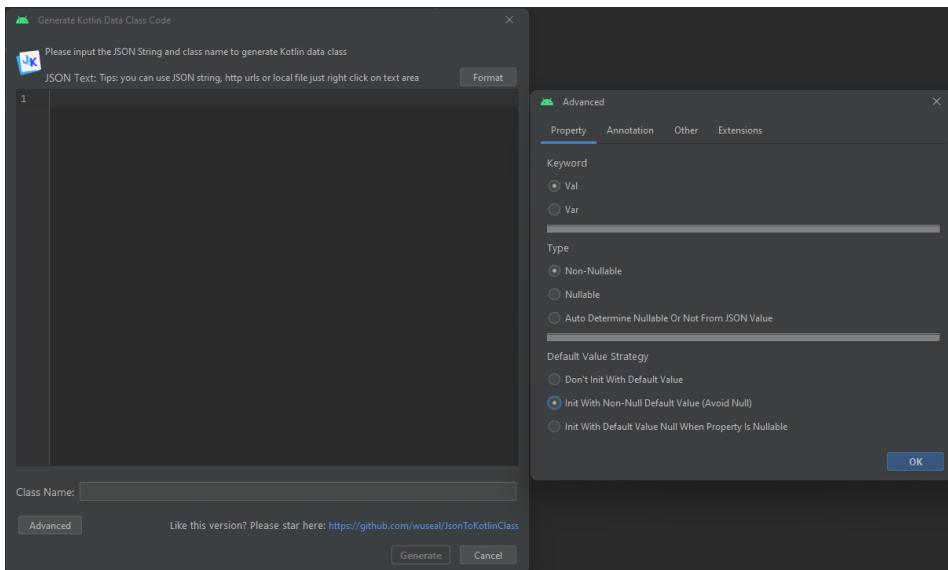


Installa e premi *Apply*.

Una volta installato il plugin, apri *data/Entities* il file che contiene le data classes del nostro progetto, e commenta *Movie*: lo ricreeremo grazie a questo plugin.



Ora clicca con il tasto destro > Generate (o ctrl-insert, o alt-k) e seleziona *Kotlin from Json*



Clicca su Advanced e seleziona Default value > Init with non-null

Ora che hai impostato il plugin, procediamo con la creazione della classe modello.

- Copia il contenuto di movies.txt del campo di testo
- In Class name, inserisci MoviesResponse

Spingendo *Generate* avremo questo risultato

```
▲ Manolo D'Antonio
└ data class MoviesResponse(
    val dates: Dates = Dates(),
    val page: Int = 0,
    val results: List<Result> = listOf(),
    val total_pages: Int = 0,
    val total_results: Int = 0
)

▲ Manolo D'Antonio
└ data class Dates(
    val maximum: String = "",
    val minimum: String = ""
)

▲ Manolo D'Antonio
└ data class Result{
    val adult: Boolean = false,
    val backdrop_path: String = "",
    val genre_ids: List<Int> = listOf(),
    val id: Int = 0,
    val original_language: String = "",
    val original_title: String = "",
    val overview: String = "",
    val popularity: Double = 0.0,
    val poster_path: String = "",
    val release_date: String = "",
    val title: String = "",
    val video: Boolean = false,
    val vote_average: Double = 0.0,
    val vote_count: Int = 0
}

)
```

Il plugin non ha solo generato tutti i membri della classe, ma anche altre classi in base agli oggetti presenti nella struttura Json.

I nomi dei membri della classe sono uguali ai nomi dei campi nel json.

3. Gson

GSON è una libreria che automatizza il processo di serializzazione\deserializzazione

Gson utilizzerà ogni membro della classe MoviesResponse per trovare nel Json un campo con lo stesso nome, effettuando la Deserializzazione.

- La libreria è importata nel progetto nel build.gradle:app

```
implementation 'com.squareup.retrofit2:converter-gson:2.9.0'
```

Annotations

Per il nostro caso dobbiamo fare una modifica: la classe *Result* rappresenta l'oggetto *Movie*, quindi rinominiamola, facendo attenzione a rinominare anche il nome del membro *results* in *movies*.

The screenshot shows the Android Studio code editor with the following code:

```
val page: Int = 0,  
    val movies: List<Movie> = listOf(),  
    val total_pages: Int = 0,  
    val total_results: Int = 0  
)  
  
▲ Manolo D'Antonio  
data class Dates(  
    val maximum: String = "",  
    val minimum: String = ""  
)  
  
▲ Manolo D'Antonio *  
data class Movie(  
    val adult: Boolean = false,  
    val backdrop_path: String = "",  
    val genre_ids: List<Int> = listOf()
```

Ora il membro *movies* non ha un corrispettivo nel testo del json, che rimane *results*.

Modifichiamo il comportamento della deserializzazione per venire incontro alla nostra modifica, aggiungendo a *movies* una annotazione:

```
@SerializedName("results") val movies: List<Movie> = listOf(),
```

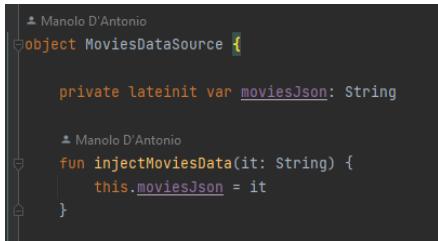
La Annotation *SerializedName* annuncia al deserializzatore di GSON che dovrà cercare nel json un campo con il nome "results" e inserire il suo valore nel membro *movies*.

SerializedName è una delle annotation fornite da GSON.

Builder

Ora vediamo come convertire il Json in Oggetto.

Apri *MoviesDataSource* dove è stato iniettato il json come stringa



```
object MoviesDataSource {  
  
    private lateinit var moviesJson: String  
  
    fun injectMoviesData(it: String) {  
        this.moviesJson = it  
    }  
}
```

Crea una nuova funzione chiamata *requestMovies* con il seguente contenuto:

```
fun requestMovies(){  
    val movies: MoviesResponse =  
        Gson().fromJson(moviesJson, MoviesResponse::class.java)  
  
    return movies  
}
```

- `Gson()` crea un'istanza del builder della libreria
- `fromJson` Deserializza il json `moviesJson` in una istanza della classe `MoviesResponse`.
Richiede come parametri:
 - una stringa con il testo in Json
 - una classe che prenderà come modello, indicata tramite [reflection](#).

`requestMovies` produrrà in output un `MoviesResponse` correttamente popolato con i dati del nostro file `movies.txt`

Il processo di serializzazione è simile, e si utilizza la funzione `toJson(objectInstance)`.

Qui non è necessario fornire un modello di classe – GSON capisce da solo quale è la data class che stiamo usando:

```
val jsonString: String = Gson().toJson(movies)
```

E.1 - MVVM pattern

IN QUESTO CAPITOLO AFFRONTEREMO UNA SERIE DI BEST PRACTICES CHE SI POSSONO APPLICARE AD UN AMPIO SPETTRO DI APP. È NECESSARIO INTENDERLE COME DELLE RACCOMANDAZIONI STANDARD DA SEGUIRE SEMPRE. SOLO IN CASI PARTICOLARI E DOPO ATTENTA ANALISI, POSSONO ESSERE ADATTATE O IGNORATE.

Docs: <https://developer.android.com/topic/architecture>

In questo e nei prossimi capitoli lavoreremo su MovieMaster, un'app sul mondo del cinema e delle serie tv:

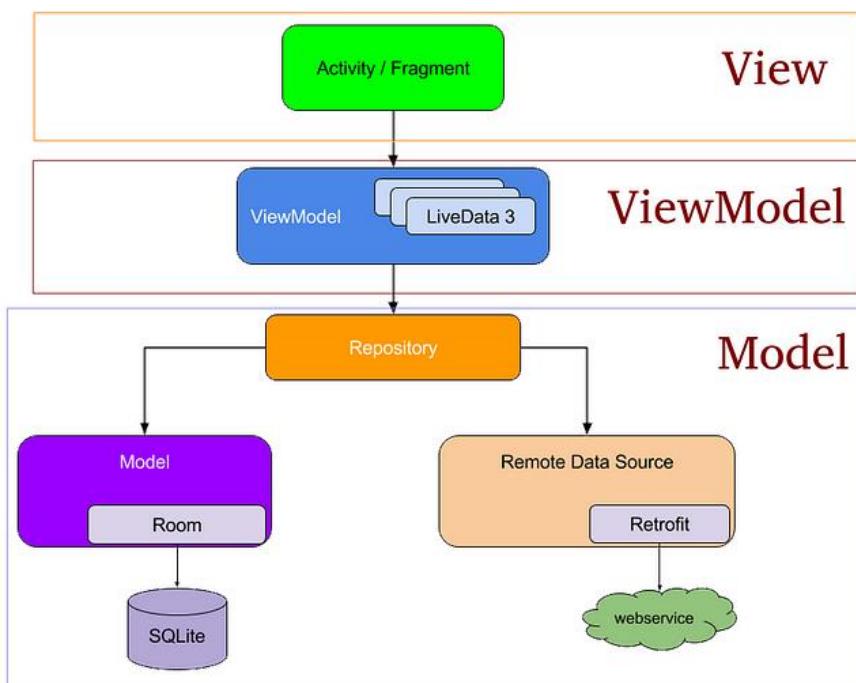
- Link progetto, utilizza il branch **start**

Finora abbiamo osservato metodi di composizione delle app, per creare funzionalità e testarle. In questo capitolo e nei prossimi, vedremo come è possibile strutturare un progetto in maniera tale che risulti sostenibile nel tempo, organizzato, e comprensibile dai colleghi che andranno a leggere il nostro codice.

Studieremo il **pattern architettonico MVVM**, che è l'abbreviazione di **MODEL/VIEW/VIEWMODEL**, l'attuale pattern consigliato per Android.

Lo scopo principale di questa architettura è di evitare *god classes* rispettando i principi di *Separation of Concerns* e *Single Responsibility*, e facilitare il testing.

Nel seguente schema generale dell'architettura, le frecce indicano le dipendenze dei vari Layers.



In questa architettura:

>>>>>>>>>>

VIEW invierà eventi al VIEWMODEL, che in base ad essi richiederà dati a MODEL.

<<<<<<<<<<

MODEL invierà dati a VIEWMODEL, che se necessario li elaborerà prima di inviarli a VIEW.

Questo pattern è chiamato **Unidirectional Data Flow (UDF)**

In UDF, lo **stato (dati)** scorre in una sola direzione. Gli **eventi** che modificano i dati scorrono nella direzione opposta.

L'idea di base dietro questo pattern è di rendere l'app guidata dallo stato dei dati. Sarà lo stato che verrà osservato dalla View, che reagirà in base a come esso cambia.

1. Model

Definizione

Il layer Model è **responsabile dei dati, e l'unico che può modificarli alla sorgente**. Riceve eventi ed invia dati a ViewModel.

Il Model utilizza classi chiamate **Repository** per interfacciarsi a ViewModel. Repository astrae la sorgente dati al resto dell'app, la quale non conosce niente della sorgente dati.

Solitamente all'interno del Model è contenuta la **Single Source of Truth**: una sorgente dati, che è proprietaria dei dati e che può modificarli. Spesso si tratta di un servizio remoto o di un database.

Esempio

Apri MoviesRepository, attualmente vuoto:

```
class MoviesRepository(val moviesDataSource: MoviesDataSource = MoviesDataSource)
```

Vedi che ha come dipendenza moviesDataSource, da cui otterremo i dati da inviare ai viewModels.

Aggiungi una nuova funzione:

```
fun getMoviesResponse() = moviesDataSource.requestMovies()
```

Ora MoviesRepository sta astraendo la sorgente dati dal resto dell'app. Nel caso ci fossero piu' sorgenti dati, si occuperebbe di organizzarle per fornire una risposta univoca alle richieste dei ViewModels.

- Puoi vedere queste modifiche nel branch *solution1_repository*

2. ViewModel

Definizione

Il Layer ViewModel è **responsabile di esporre dati a View**. Riceve eventi e invia dati a View. Invia eventi e riceve dati da Model.

Il ViewModel mantiene i dati relativi alla View, che non vengono distrutti quando l'Activity o il Fragment viene distrutto e ricreato dal framework Android. Può elaborare i dati ricevuti da Model per allinearli alle specifiche necessità di View, ad esempio cambiando l'ordine degli elementi di una lista, o rendendo un testo maiuscolo.

Il ViewModel non conosce come vengono utilizzati i dati che invia a View, e non ha alcun riferimento alla UI. Non conosce come Model ottiene i dati.

Esempio

Nel package UI, crea una nuova classe chiamata MovieListViewModel, che ha come parametro del costruttore il repository MoviesRepository, e che estende ViewModel

```
class MovieListViewModel(val repository: MoviesRepository = MoviesRepository()) : ViewModel()
```

Crea ora una funzione che permette alla view di ottenere la lista di Film in ordine di data di uscita recente:

```
fun getMovies() = repository.getMoviesResponse().movies
    .sortedByDescending { it.release_date }
```

Così facendo ViewModel fornirà i dati a View, senza che essa sia a conoscenza di come vengono recuperati, o di quale sia la loro struttura originale.

3. View

Definizione

Il Layer View (Activity, Fragment) è **responsabile della presentazione e dell'interazione con l'utente**. Informa il ViewModel sulle azioni dell'utente e osserva lo stato, esposto dal ViewModel. In base allo stato, mostra la corretta UI.

View osserva i dati senza modificarli, e non conosce come vengono elaborati né come vengono ottenuti.

Il sistema Android può distruggere e ricreare Activities e Fragments in qualsiasi momento in base a determinate interazioni dell'utente, ad esempio la rotazione del device, o a causa di condizioni del sistema, come memoria insufficiente. Poiché questi eventi non sono sotto il tuo controllo, la logica decisionale sui dati è di solito contenuta nel ViewModel, che ha la capacità di mantenere il suo stato.

Esempio

Ci affidiamo al framework android per recuperare il viewmodel nella view:

```
private val model: MovieListViewModel by viewModels()
```

A questo punto potremo utilizzare la funzione che abbiamo creato per ottenere i dati:

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    FragmentMovieListBinding.bind(view).run { this: FragmentMovieListBinding

        movieListRv.adapter = movieListAdapter.apply { this: MovieListAdapter
            dataset = model.getMovies()
        }
    }
}
```

E' ora possibile lanciare l'app per vedere una lista di cards con ID.

- Puoi vedere la soluzione nel branch *solution2_viewmodel*

4. Concetti di ownership

Doc: <https://developer.android.com/topic/libraries/architecture/viewmodel/viewmodel-apis>

Nella view (fragment) abbiamo ottenuto il viewmodel tramite `by viewModels`, una funzione che permette di definire lo **scope** del viewmodel.

Lo scope è un riferimento che viene utilizzato dal sistema Android per definire quando l'istanza di un viewModel deve essere distrutta.

Lo scope è un collegamento ad un `ViewModelStoreOwner`, un'interfaccia implementata dai `LifecycleOwner`.

Un.viewmodel può essere **scoped to**:

- Activity
- Fragment
- NavigationGraph

Ognuno di questi elementi ha dei componenti interni che il sistema Android prende a riferimento quando deve effettuare operazioni di distruzione\ricostruzione, e che rimangono in memoria durante queste operazioni. Finché anche questi riferimenti non verranno rimossi dal sistema, l'istanza del ViewModel resterà disponibile.

Queste sono le funzioni più comuni disponibili per lo scoping:

- **by viewModels()** : scoping al candidato più vicino. Se lanciato in un fragment, sarà scoping al fragment. Se lanciato in una activity, sarà scoping all'activity.
- **by activityViewModels()** : scoping all'activity *parent* di dove viene lanciato. Se lanciato in un fragment, sarà scoping all'activity che lo contiene.
- **by navGraphViewModels(R.id.nav_graph)** : scoping al navigation graph fornito come parametro

Queste funzioni cercano in memoria un'istanza di quel viewmodel con quello scope.

Se la trovano, forniscono un riferimento a quell'istanza.

Se non lo trovano, creano in memoria l'istanza del viewModel e gli assegnano lo scope, e poi forniscono il riferimento.

Ad esempio, è possibile usare `by activityViewModels()` per condividere la stessa istanza di viewmodel tra piu' fragments.

5. Esercizio: shared viewmodel

Utilizza i dati disponibili nel moviesDataSource per popolare l'app con i seguenti presupposti:

- Il fragment lista deve avere almeno immagine e titolo del film. Il layout riga deve essere semplice e contenere pochi dati utili.
 - La documentazione del MovieDB, il servizio da cui sono stati presi questi dati, è disponibile qui: <https://developer.themoviedb.org/docs>
- Il fragment dettaglio può contenere la maggior parte dei dati di Movie, in particolare il titolo, la locandina, e la descrizione. Mostra questi e altri dati in modo chiaro, e non mostrare dati senza che l'utente possa comprenderli (ad esempio, popularity stampato senza contesto)
- I fragment lista e dettaglio devono condividere la stessa istanza di viewmodel
- Nella navigazione, passa al fragment dettaglio solo l'id del film. Recupera il Movie corretto nel fragment di dettaglio, tramite il viewmodel.
 - Per rigenerare i file creati dai plugin di navigazione, come le directions o i navArgs, effettua un clean del progetto

• Feature: film preferiti.

Aggiungi a Movie una *var favourite: Boolean*

Aggiungi una stella nell'interfaccia di dettaglio, che puo' essere cliccata per contrassegnare un film come preferito.

Anche la lista deve mostrare se un film è preferito oppure no, ma non deve essere modificabile dalla lista.

Non è importante (per ora) che i film preferiti permangano quando si chiude e riapre l'app

- Per la stella, puoi importare una drawable nuova o utilizzare alcune disponibili nelle risorse android.R.drawable

F.1 – Async Programming

1. Concurrency e Threads

Finora abbiamo considerato l'app come se fosse una serie di eventi consecutivi (**sequential**).

Con l'aumentare della complessità della nostra app però, sarà necessario cominciare ad integrare il concetto di concorrenza (**concurrency**), ovvero operazioni che si svolgono in "parallello".

Un'operazione si dice Concurrent quanto è in grado di essere eseguita fuori ordine o apparentemente in parallelo, consentendo un uso più efficiente delle risorse.

Il multitasking è l'esecuzione simultanea di più attività concorrenti durante un certo periodo di tempo.

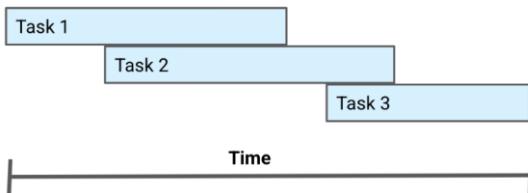
Commented [DM1]: Parallello è un termine che useremo spesso per praticità. In realtà, in un processore singolo core non è possibile effettuare più operazioni contemporaneamente, ma è possibile schedulare i thread in maniera che siano elaborati in maniera più efficiente.

Single Path of Execution



Time

Concurrency



Time

Utilizzare operazioni che avvengono in parallelo ci permetterà di evitare che l'app si blochi nel caso dovessimo effettuare delle operazioni particolarmente pesanti, come l'elaborazione di immagini o la richiesta di dati da un servizio remoto. Queste saranno eseguite in background, mentre l'utente potrà continuare ad utilizzare l'interfaccia.

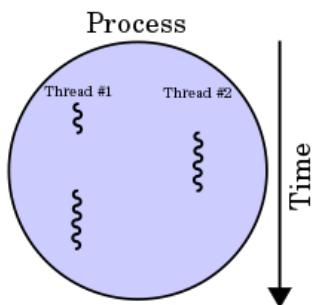
Definizione

I **Thread** sono delle sequenze che spezzettano il nostro programma e sono eseguite in maniera concorrente.

Il sistema Android utilizza vari thread. Il Main thread è quello che viene utilizzato di base per tutto, inclusa l'interfaccia. Appesantire troppo questo thread rende l'interfaccia inutilizzabile, quindi a volte è necessario spostare le operazioni su altri thread.

Inoltre, Android pone dei limiti a cosa può essere fatto sul Main thread. Ad esempio, lanciare un'operazione di rete (*Network*) sul Main thread creerà un'eccezione.

Vediamo un esempio di concorrenza utilizzando il sito
<https://developer.android.com/training/kotlinplayground>



Utilizza questo codice, che crea tre Thread concorrenti:

```
fun main() {
    val states = arrayOf("Starting", "Doing Task 1", "Doing Task 2", "Ending")
    repeat(3) {
        Thread {
            println("${Thread.currentThread()} has started")
            for (i in states) {
                println("${Thread.currentThread()} - $i")
                Thread.sleep(50)
            }
        }.start()
    }
}
```

I thread saranno stampati in questa maniera:

Thread[Thread-0,5,main] – Message

All'interno delle parentesi c'è il nome del thread (Thread-n), la sua priorità (5), ed il suo gruppo (main)

Lanciandolo varie volte, vedrai che il risultato non sarà sempre lo stesso: in base alle risorse disponibili in quel momento, i thread verranno eseguiti in un'ordine non-specifico e non sequenziale.

I thread sono un modo di effettuare operazioni in parallelo, ma hanno varie problematiche da tenere in conto:

- Richiedono molte risorse, spesso nullificando lo scopo del multitasking.
- La mancanza di sicurezza dell'ordine con cui vengono eseguiti può produrre risultati imprevedibili.
- Più thread che cercano di accedere allo stesso momento alla stessa risorsa creano dei bug o crash (**race condition**)

Quindi è complicato lavorare direttamente con i thread. Per questo solitamente si utilizzano le Kotlin **Coroutines**

2. Coroutines

<https://kotlinlang.org/docs/coroutines-overview.html>

Definizione

Le Coroutines sono delle astrazioni costruite sulla base di Threads.

Questo permette loro di essere **molto leggere**, di integrare features come la **possibilità di essere messe in pausa o interrotte**, e di lavorare in coro in un **multitasking cooperativo** tra i vari elementi concorrenti.

Una coroutine è composta da tre elementi principali:

Job	Un'unità di lavoro cancellabile che racchiude delle operazioni, come quella creata con la funzione <code>launch()</code> .
CouroutineScope	È un <i>context</i> che impone regole ai suoi discendenti, come i criteri per la cancellazione. Le funzioni utilizzate per creare nuove coroutine come <code>launch()</code> e <code>async()</code> estendono <code>CouroutineScope</code> .
Dispatcher	Determina il <i>thread</i> che verrà utilizzato dalla coroutine. Il dispatcher <i>Main</i> eseguirà sempre le coroutine sul <i>Main thread</i> , mentre i dispatcher come <i>Default</i> , <i>IO</i> o <i>Unconfined</i> utilizzeranno altri <i>thread</i> .

3. Suspend Functions

Definizione

suspend è una keyword che indica che la funzione contiene un'altra suspend function. Una suspend function può essere invocata solo all'interno di una coroutine, o all'interno di un'altra suspend function.

Apri MoviesRepository e fai questa modifica:

```
suspend fun getMoviesResponse(): MoviesResponse {  
    delay( timeMillis: 1000)  
    return moviesDataSource.requestMovies()  
}
```

delay() è una funzione che mette in pausa la funzione dove è invocata, in maniera asincrona *non-blocking*.

Puoi vedere sulla sinistra che la **freccetta con l'intersezione verde** ti notifica che *su quella riga c'è una suspend function*.

Ora la funzione richiederà circa 1 secondo per essere eseguita, e non potrà più essere eseguita in maniera sincrona, imponendo la concorrenza.

4. viewModelScope, launch()

Definizione viewModelScope

`viewModelScope` è un `CoroutineScope` molto importante fornito da `ViewModel`, che si occupa in maniera automatica di alcune attività. Se dovessimo lanciare delle funzioni che possono essere eseguite solo su alcuni specifici thread (ad esempio una richiesta network su `Dispatchers.IO`, o l'invio di dati all'interfaccia su `Dispatchers.Main`), `viewModelScope` lavorerebbe per noi effettuando questi salti da un thread all'altro.

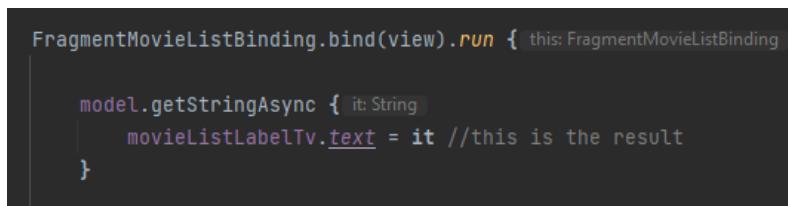
Il lifecycle di `viewModelScope` è legato a quello del suo `ViewModel`: se questo cessa di esistere, anche lo scope si chiude, interrompendo tutte le coroutines in corso.

In un `ViewModel`:



```
    fun getStringAsync(
        onCompleteListener: (result: String) -> Unit
    ) {
        viewModelScope.launch { this: CoroutineScope
            val result = getString()
            onCompleteListener(result)
        }
    }
}
```

`onCompleteListener` è una funzione anonima che passiamo come parametro, che ci permetterà di ottenere il risultato da una operazione asincrona, quando la invocheremo dalla view:



```
FragmentMovieListBinding.bind(view).run { this: FragmentMovieListBinding
    model.getStringAsync { it: String
        movieListLabelTv.text = it //this is the result
    }
}
```

Definizione launch()

`.launch` è una funzione che crea una coroutine.

Il contenuto viene eseguito in maniera asincrona, senza bloccare il Main thread: un particolare che ci viene specificato dalla documentazione.

`.launch` racchiude il suo contenuto in un `Job`, e produce questo `Job` come risultato (e non il suo contenuto). Quindi dobbiamo avere un altro modo per comunicare il risultato delle operazioni al suo interno. Nel nostro caso, è il listener.

5. Futures

Finora abbiamo utilizzato una coroutine che permette di lanciare un pezzo di codice pesante in parallelo, permettendo al codice principale di continuare l'esecuzione.

Ci sono casi però in cui, all'interno di una coroutine, ci troviamo a dover lanciare piu' suspend fun in parallelo.

Tipicamente questo avviene quando abbiamo bisogno non di una, ma di piu' chiamate network. Inoltre, queste chiamate devono dare il risultato tutte insieme, perché aggregano i dati che singolarmente non avrebbero senso.

Un esempio: per mostrare il costo di un oggetto in vendita, dobbiamo ottenere il costo dell'oggetto da un servizio, e al contempo dobbiamo ottenere da un altro servizio il tasso di cambio tra la valuta originale dell'oggetto e quella dell'utente che lo sta comprando. Non avrebbe senso mostrare il prezzo in rupie a un utente europeo, e tantomeno avrebbe senso mostrare solo il tasso di cambio senza il prezzo dell'oggetto.

Una soluzione è lanciare all'interno della coroutine altre coroutines: è un metodo lecito e spesso utilizzato. Il problema però è che le coroutines non si coordineranno tra di loro; quindi, dovremmo inventarci un metodo per controllare ciclicamente i due risultati dell'esempio finchè entrambi non sono disponibili.

Per risolvere questo scomodo problema Kotlin ci viene in soccorso implementando il concetto di **Futures**

Un Future è un oggetto che funge da placeholder per un risultato inizialmente sconosciuto, di solito perché il calcolo del suo valore non è ancora completo.

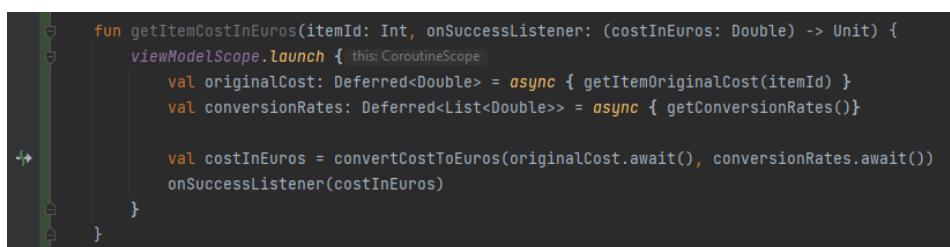
https://en.wikipedia.org/wiki/Futures_and_promises

Async / Await

Kotlin utilizza le funzioni `.async` per lanciare una coroutine che ritorna subito un future, e `.await` per utilizzare il future appena sarà disponibile.

Kotlin wrappa i futures in un oggetto chiamato **Deferred**, che offre alcune facilitazioni e funziona similmente a *Job*: ad esempio, se necessario, può interrompere la coroutine lanciata da `async` prima che sia terminata.

Deferred è una **Promise**. Il Future propriamente detto è il valore “wrappato” da Deferred.



```
fun getItemCostInEuros(itemId: Int, onSuccessListener: (costInEuros: Double) -> Unit) {
    viewModelScope.launch { thisCoroutineScope {
        val originalCost: Deferred<Double> = async { getItemOriginalCost(itemId) }
        val conversionRates: Deferred<List<Double>> = async { getConversionRates() }

        val costInEuros = convertCostToEuros(originalCost.await(), conversionRates.await())
        onSuccessListener(costInEuros)
    }
}
```

- In questo esempio `getItemOriginalCost` o `getConversionRates` sono due suspend fun, e non sappiamo quale verrà risolta prima. Decidiamo quindi di lanciarle come Futures tramite `async`.
- `Async` genera delle coroutines e ritorna subito (sincronicamente) un Deferred.
- `originalCost` e `conversionRates` sono due Deferred. Entrambi wrappano i risultati Future delle funzioni nelle coroutines, che saranno dei Double.
- `convertCostToEuros` viene invocato dandogli come parametri i valori Double prodotti da `await`, e verrà effettivamente lanciato solo quando entrambi i valori saranno disponibili.
- Gli `await` fanno in modo di coordinare le due coroutines: quando arriva la prima, il processo aspetterà che arrivi anche l'altra prima di proseguire.
- Puoi vedere infatti che l'unica riga che blocca il processo è proprio questa, visualizzata con una freccetta con intersezione verde, sulla sinistra. Sono gli `await` a bloccare.

6. Esercizio

Aggiungi a Movie Master:

- In MovieListViewModel:
 - ora `getMoviesResponse` nel repository non puo' essere piu' invocato in maniera sincrona.
Modifica `getMovies` in maniera che non crei errore, e rendila **privata**.
 - crea una nuova funzione **pubblica** `getMoviesAsync` dove
 - Lancia una coroutine dove ottieni la lista di film
 - Rendi questo risultato ottenibile dalla view
 - Salva questo risultato in una variabile del viewModel
 - Fa in modo che `getMoviesById` resti una funzione sincrona, che prende i dati da una variabile nel quale è stato salvato il risultato della richiesta dei film al repository.
- In MovieListFragment:
 - Assegna l'adapter alla recyclerview con un dataset vuoto
 - Quando i dati dei film sono disponibili, aggiorna il dataset dell'adapter.
 - Aggiungi una *Progressbar* al layout. Deve mostrare un'animazione circolare di caricamento.
 - Quando i dati dei film non sono disponibili, Progressbar deve essere visibile, mentre Recyclerview deve essere invisibile
 - Quando i dati sono disponibili, RV è visibile e Progressbar invisibile.
- Esercizio Async/Await
- Nella pagina dove è presente la lista di attori\attrici
 - Prendi l'endpoint Images e mostra almeno 3 poster
 - Prendi l'endpoint Reviews e mostra almeno 3 reviews
- Lancia queste richieste in parallelo

F.2 – Servizi Remoti e Retrofit

Finora abbiamo utilizzato una base dati presente all'interno della nostra app. Comunemente però, i dati di un'app vengono forniti da un servizio cloud remoto tramite connessione di rete.

Ci sono molti modi per interfacciarsi con un servizio remoto, il piu' comune è sicuramente l'architettura REST API

Un'API (*Application Programming Interface*) è un insieme di regole predefinite che consentono a diverse applicazioni di comunicare tra loro.

REST (*representational state transfer*) è una serie di principi architettonici che permettono di standardizzare le operazioni di trasferimento dati network, e che descrivono il WEB.

1. REST API service

<https://www.ibm.com/topics/rest-apis>

Le API REST sono formate da un server e un client. Ad esempio, il server è un servizio dati remoto, mentre il client è all'interno della nostra app.

Il server espone degli **endpoint**: degli [URI](#) dove riceve le richieste e invia le risposte. In un server remoto, questi sono una serie di indirizzi URL.

Il client invia richieste a questi endpoint per ottenere risposte specifiche per ogni endpoint.

Server e Client comunicano tramite richieste e risposte HTTP per eseguire funzioni di database standard come la creazione, la lettura, l'aggiornamento e l'eliminazione di record (noti anche come CRUD).

Ad esempio, il client di un'API REST può inviare al server una richiesta:

- POST per creare un record (C)reate
 - GET per recuperare un record (R)etrieve
 - PUT per aggiornare un record (U)pdate
 - DELETE per eliminare un record (D)elete

Tutti i metodi HTTP possono essere utilizzati nelle chiamate API

Le risposte sono comunemente in formato JSON.

2. Retrofit

Retrofit è una libreria Client che consente di semplificare enormemente il processo di richiesta e risposta, integrando anche il processo di serializzazione e deserializzazione json tramite GSON in un unico flusso.



Un'implementazione di Retrofit è composta principalmente da due componenti:

- Un'interfaccia, dove vengono elencate le richieste specificando l'endpoint, il modello della data class che ci aspettiamo come risposta, e il tipo di richiesta HTTP (GET, POST...)
- Un client HTTP, che se necessario può essere configurato a nostra discrezione

In una tipica implementazione, il **Repository** invierà una richiesta a una Interfaccia Retrofit (che rappresenta una **data source**) invocando una delle sue funzioni. Queste saranno **suspend functions** e forniranno come risposta direttamente l'istanza di un oggetto.

Sarà Retrofit a occuparsi della gestione della chiamata HTTP e della deserializzazione della risposta.

Setup Progetto
In build.gradle:app:

```
implementation 'com.squareup.retrofit2:retrofit:2.9.0'  
implementation 'com.squareup.retrofit2:converter-gson:2.9.0'
```

Il *converter* è la libreria che si occuperà della Ser\Deser – in questo caso, GSON.

3. Interface

La prima parte dell'implementazione è dedicata all'interfaccia, che astrarrà gli endpoints del servizio remoto.

Andiamo sul sito del servizio che usiamo: The Movie DB, dove possiamo trovare la **reference (interface agreement)** dell'API

<https://developer.themoviedb.org/reference/intro/getting-started>

Vediamo che sulla destra vengono indicati gli endpoint piu' popolari

The screenshot shows a list of popular endpoints under the heading "POPULAR ENDPOINTS". Each endpoint is represented by a green button with the method (GET) and the URL path. The listed endpoints are: /3/discover/movie, /3/search/movie, /3/authentication, /3/movie/popular, and /3/movie/now_playing.

Endpoint
/3/discover/movie
/3/search/movie
/3/authentication
/3/movie/popular
/3/movie/now_playing

A noi interessa l'ultimo, /3/movie/now_playing. Finora abbiamo utilizzato una risposta json di questo endpoint salvata nel file movies.txt. Adesso faremo in modo che ogni volta che si apre l'app, la lista venga aggiornata.

Trovando la chiamata nella lista, vedremo il suo indirizzo completo, e un esempio per essere invocata.

The screenshot shows the "Now Playing" endpoint details. It includes a note about it being a discover call, a curl command example, and a "Try It" button to test the API directly.

Now Playing

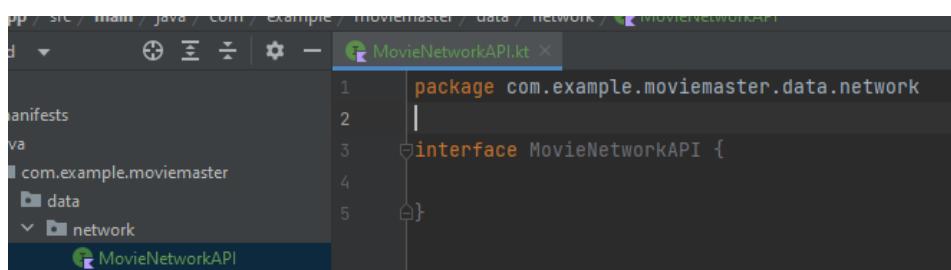
This call is really just a discover call behind the scenes. If you would like to tweak any of the default filters head over and read about [discover](#).

CURL

```
curl --request GET \
--url "https://api.themoviedb.org/3/movie/now_playing" \
--header "Accept: application/json"
```

Try It

Nel nostro progetto android, crea un package *network* dentro il package *data*. Crea dentro di esso un'interfaccia chiamata *MovieNetworkAPI*



```
pp / src / main / java / com / example / moviemaster / data / network / MovieNetworkAPI.kt
```

```
1 package com.example.moviemaster.data.network
2
3 interface MovieNetworkAPI {
4
5 }
```

All'interno dell'interfaccia, aggiungi questa funzione per mappare l'endpoint:

```
@GET("/3/movie/now_playing")
suspend fun getMoviesNowPlaying(): MoviesResponse
```

- L'annotazione **GET** identifica il tipo di chiamata HTTP da compiere, e l'endpoint dove farla
- La funzione è una **suspend fun**
- Il risultato è un **MoviesResponse**.

Per ottenere una chiamata funzionale, dobbiamo fare un altro step.

TMDB è una API esposta tramite HTTPS, quindi richiede una autorizzazione.

Aggiungi questa annotazione dopo GET per fornirgli un identificativo:

```
@Headers("Authorization: Bearer
eyJhbGciOiJIUzI1NiJ9.eyJhdWQiOiJkOTThkJyUyOWUwYmVjYmRkMmlzNDRhYTVmN2Q4M2VkYSIsInN1YiI6IjU
4OTFhNjczOTI1MTQzMmRjZDAwNGYwMCIsInNjb3BlcyI6WyJhcGlfcmVhZCJdLCJ2ZXJzaW9uljoxfQ.jl8QHdNF
S3srkDw5_n8ScNA5S608kFC2jC4m7UZmSiQ")
```

Questo è un **Access Token**, che di solito identifica un'app.



```
interface MovieNetworkAPI {

    @Manolo D'Antonio *
    @GET("/3/movie/now_playing")
    @Headers("Authorization: Bearer eyJhbGciOiJIUzI1NiJ9.eyJhdWQiOiJkOTThkJyUyOWUwYmVjYmRkMmlzNDRhYTVmN2Q4M2VkYSIsInN1YiI6IjU
4OTFhNjczOTI1MTQzMmRjZDAwNGYwMCIsInNjb3BlcyI6WyJhcGlfcmVhZCJdLCJ2ZXJzaW9uljoxfQ.jl8QHdNF
S3srkDw5_n8ScNA5S608kFC2jC4m7UZmSiQ")
    suspend fun getMoviesNowPlaying(): MoviesResponse

}
```

Un singolo token puo' fare solo un certo numero di richieste al minuto\ora\giorno.

Per ottenere il tuo token, regista un account su TMDB: loggando, ti comparirà direttamente nella pagina della reference, sotto "Authentication"

4. Retrofit Client

Ora creiamo l'implementazione di Retrofit che farà da ponte tra l'interfaccia e il client http.

Dentro *network*, crea una nuova classe *RetrofitClient*

Aggiungi una val *movieBaseAddress* che sarà una Stringa dove è indicato l'URL comune a tutti gli endpoint dell'API. Puoi trovare questo indirizzo controllando la reference di TMDB.

```
val movieBaseAddress = "https://api.themoviedb.org"
```

Quando si effettuerà una chiamata, l'endpoint specificato nell'interfaccia verrà concatenato a questo URL

Ora crea un'istanza di Retrofit, utilizzando il Builder:

```
val movieRemoteDataSource =  
    Retrofit.Builder()  
        .addConverterFactory(GsonConverterFactory.create(Gson()))  
        .baseUrl(movieBaseAddress)  
        .build()  
        .create(MovieNetworkAPI::class.java)
```

Questo builder indica:

- GSON come deserializzatore
- il base url che sarà utilizzato per effettuare le chiamate
- l'interfaccia che definisce gli endpoint
- se non si specifica il client http, retrofit ne costruirà uno standard

Implementazione

Ora modifica MoviesRepository per includere l'API come sorgente dati, e invoca il builder da RetrofitClient.movieRemoteDataSource

```
class MoviesRepository{
    private val moviesDataSource: MoviesLocalDataSource = MoviesLocalDataSource,
    private val movieRemoteDataSource: MovieNetworkAPI = RetrofitClient().movieRemoteDataSource
)
{
    suspend fun getMoviesResponse() = movieRemoteDataSource.getMoviesNowPlaying()
    //    fun getMoviesResponse() = moviesDataSource.requestMovies()
}
```

Adesso apprendo l'app, quando la View richiederà i movies al ViewModel, getMoviesResponse invocherà una richiesta network e i movie verranno scaricati da TMDB.

L'app funziona, ma abbiamo invocato getMoviesResponse da viewModelScope, che è abbastanza "intelligente" da saltare da un thread a un altro quando necessario.

getMoviesResponse non mette limiti da dove può essere invocata, quindi potremmo richiamarla anche da un Main Thread. Questo manderebbe l'app in crash, perché getMoviewNowPlaying effettua una chiamata network, cosa proibita sul main thread dal sistema Android.

Facciamo dunque un'ultima modifica per rendere questa funzione *safe*.

```
suspend fun getMoviesResponse() = withContext(Dispatchers.IO) { this: CoroutineScope
    movieRemoteDataSource.getMoviesNowPlaying()
}
```

withContext è una suspend function utilizzata per definire un Dispatcher, identificandone l'uso. In questo caso, utilizza Dispatchers.IO per una chiamata network.

Diversamente da *launch* che ritorna un job, *withContext* ritorna il contenuto delle operazioni al suo interno.

5. HttpClient

Quando abbiamo dichiarato getMoviesNowPlaying nell'interfaccia di retrofit, abbiamo dovuto aggiungere un header che ci ha permesso di autorizzarci al server.

Per evitare di dover specificare ad ogni chiamata andremo adesso a modificare il client retrofit creando un client HTTP personalizzato.

Copia i parametri che avevamo inserito dentro l'annotazione @Header in MovieNetworkApi dentro due val in RetrofitClient. I due punti sono il carattere di separazione.

```
private val authLabel = "Authorization"  
private val authValue = "Bearer eyJhbGciOiJIUzI1NiJ9eyJhdWQiOiJkOThkNjUyOWI
```

Ora rimuovi completamente l'annotazione @Header dalla chiamata getMoviesNowPlaying.

Torna in RetrofitClient e aggiungi un nuovo **Interceptor**, una classe del pacchetto okhttp3 che permetterà di modificare la chiamata in vari punti dell'esecuzione

```
private val authorizationInterceptor = object: Interceptor {  
    override fun intercept(chain: Interceptor.Chain) : Response {  
        val request = chain.request().newBuilder()  
            .header(authLabel, authValue)  
            .build()  
  
        return chain.proceed(request)  
    }  
}
```

Dopo aver ottenuto la request, l'interceptor creerà una nuova richiesta uguale, ma aggiungendo un header di autorizzazione. Infine, manderà la nuova richiesta modificata all'esecuzione della chiamata (chain)

OTTIMIZZAZIONE DI AUTHORIZATION INTERCEPTOR:

```
private val authorizationInterceptor = Interceptor { chain ->  
    chain.request().newBuilder() Request.Builder  
        .header(authLabel, authValue)  
        .build() Request  
        .let(chain::proceed)  
}
```

Crea un nuovo HttpClient con il builder OkHttpClient.Builder()

```
private val httpClient = OkHttpClient.Builder()
    .addInterceptor(authorizationInterceptor) OkHttpClient.Builder
    .build()

val movieRemoteDataSource =
    Retrofit.Builder() Retrofit.Builder
        .client(httpClient) Retrofit.Builder
        .addConverterFactory(GsonConverterFactory.create(Gson()))
        .baseUrl(movieBaseAddress)
        .build() Retrofit
        .create(MovieNetworkAPI::class.java)
```

- Aggiungi l'interceptor al client http
- Aggiungi il client http a retrofit

Lanciando l'app ora sarà il client http ad occuparsi dell'autorizzazione.

Log Network calls

Un caso molto comune è quello di analizzare i log delle chiamate network per esaminare potenziali problemi. Aggiungeremo adesso un interceptor che ci permetterà di fare questo lavoro

In build.gradle:app aggiungi la seguente riga e fai un sync:

```
implementation 'com.squareup.okhttp3:logging-interceptor:4.9.3'
```

Ora aggiungi un `HttpLoggingInterceptor` al client http:

```
private val loggingInterceptor = HttpLoggingInterceptor().apply { this.level = HttpLoggingInterceptor.Level.BODY }
    .addInterceptor(authorizationInterceptor)
    .addInterceptor(loggingInterceptor)
    .build()
```

- Il level identifica cosa deve loggare, con BODY loggherà headers e body

Ora lanciando l'app e aprendo il Logcat:

6. Esercizio

Cerca in TMDB un endpoint che fornisce i dati del cast, e integralo nell'app.

Nella pagina di dettaglio:

- Mostra l'immagine del regista - e fai capire che è il regista
- Mostra le immagini dei 5 attori più famosi, ed il loro ruolo nel film.
- Ordina gli attori secondo l'importanza del ruolo che hanno nel film: per primo il protagonista, poi il coprotagonista, ecc...

F.3 – MVVM Testing

1. Model

L'obiettivo del test sarà di mostrare che le operazioni di `MoviesRepository` funzionano correttamente.

`MoviesRepository` dipende però da alcune data sources:

```
class MoviesRepository(
    private val moviesDataSource: MoviesLocalDataSource = MoviesLocalDataSource,
    private val movieRemoteDataSource: MovieNetworkAPI = RetrofitClient().movieRemoteDataSource
)
```

Rimuovi `moviesDataSource`, dato che ad ora è inutilizzata.

Per `movieRemoteDataSource` creeremo un **fake**, un tipo di test double che riproduce risultati stabili senza interrogare il server di TMDB.

Fake

Crealo dentro un nuovo package *test* chiamato *data*

```
com.example.movieMaster (test)
  data
    FakeMovieDataSource
```



```
class FakeMovieDataSource : MovieNetworkAPI {

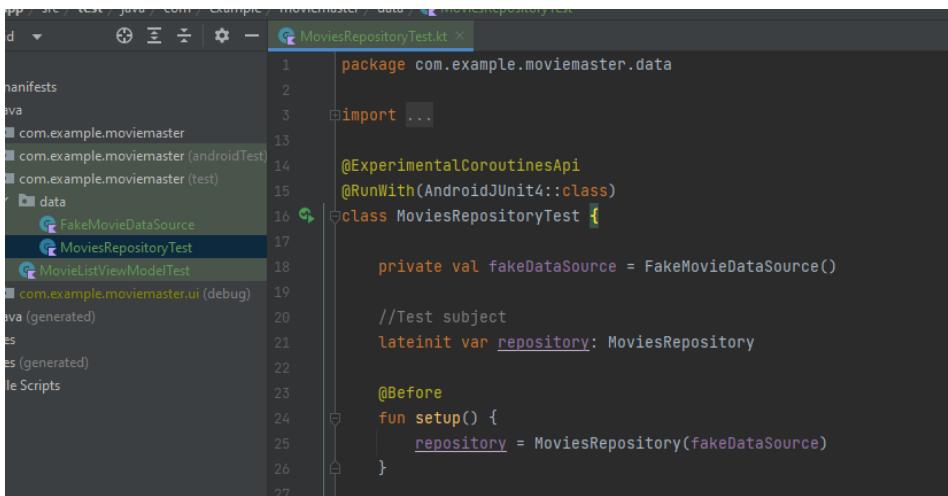
    val fakeData = MoviesResponse(
        movies = listOf(Movie(id = 999, title = "test"))
    )

    override suspend fun getMoviesNowPlaying(): MoviesResponse {
        return fakeData
    }
}
```

- *fakeData* farà le veci della risposta del servizio

Test

Ora crea il setup per il test di MoviesRepository



```
pp / src / test / java / com / example / moviemaster / data / MoviesRepositoryTest.kt
```

```
1 package com.example.moviemaster.data
2
3 import ...
4
5 @ExperimentalCoroutinesApi
6 @RunWith(AndroidJUnit4::class)
7 class MoviesRepositoryTest {
8
9     private val fakeDataSource = FakeMovieDataSource()
10
11     //Test subject
12     lateinit var repository: MoviesRepository
13
14     @Before
15     fun setup() {
16         repository = MoviesRepository(fakeDataSource)
17     }
18 }
```

- All'inizio la annotation `@ExperimentalCoroutinesApi`, che ci permetterà di utilizzare strumenti per le coroutines.
- Creiamo il repository immettendo nel costruttore `fakeDataSource`. Avremo così una risposta controllata del “servizio”, permettendo di focalizzare i test sulle funzionalità di Repository.

```
@Test
fun getMoviesResponse() {
    val response = repository.getMoviesResponse()
    assertThat(response).isEqualTo(fakeDataSource.fakeData)
}
```

`getMoviesResponse` è però una suspend function, che può essere lanciata solo in una coroutine o altre sus fun. Utilizziamo quindi il tool delle coroutines `runTest`, che ci permette di eseguire suspend function in maniera sincrona

```
@Test
fun getMoviesResponse() = runTest { this: TestScope
    val response = repository.getMoviesResponse()
    assertThat(response).isEqualTo(fakeDataSource.fakeData)
}
```

2. ViewModel

MovieListViewModel dipende da MoviesRepository

```
class MovieListViewModel(val repository: MoviesRepository) : ViewModel() {
```

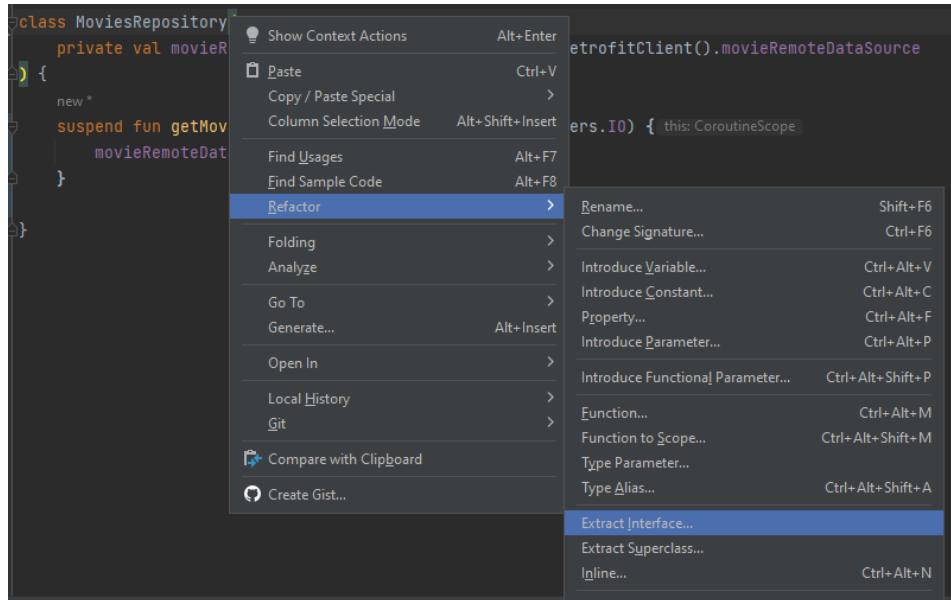
Andiamo a creare un Fake di MoviesRepository.

Interfaccia

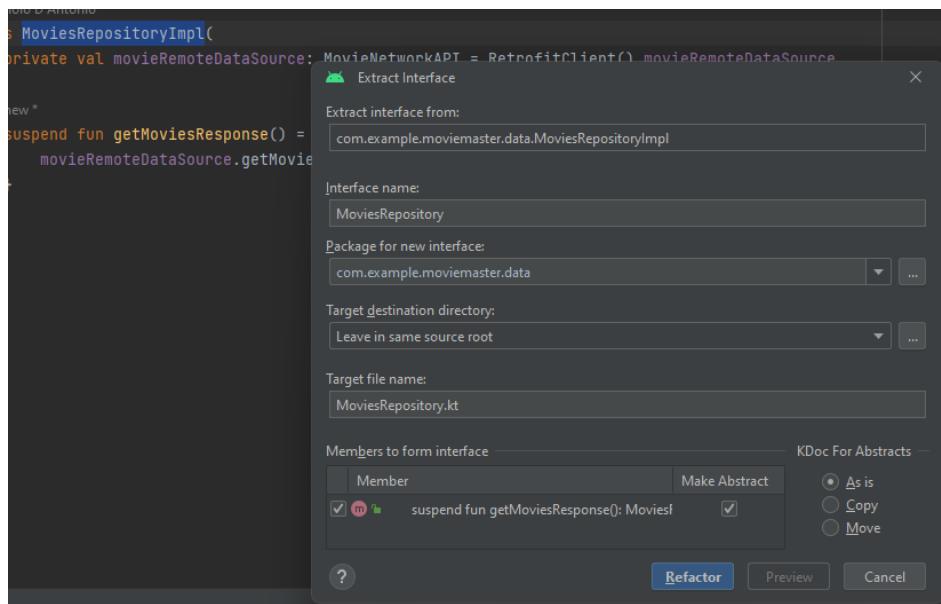
Quando abbiamo creato il fake di movieDataSource ci siamo appoggiati ad una interfaccia, faremo la stessa cosa per il repo.

Prima, rinomina il repo in **MoviesRepositoryImpl**

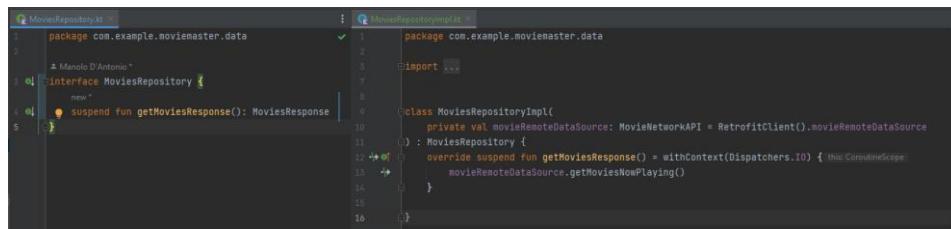
Poi estrai un'interfaccia dal repository:



Estraila in un nuovo file, dentro il package *data*, chiamato *MoviesRepository*. Fai attenzione a selezionare tutti i membri del repo.

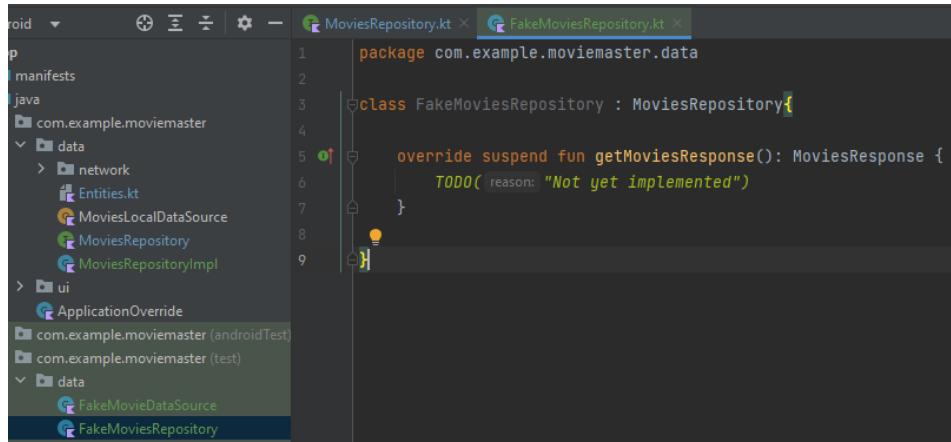


Ora c'è una nuova interfaccia, che il nostro repo implementa:



Fake

Crea `FakeMoviesRepository` dentro `test/data`. Questo repo deve implementare la nuova interfaccia e tutti i relativi membri.



The screenshot shows the Android Studio interface with the project navigation bar at the top. Below it is a tree view of the project structure:

- src
- manifests
- java
 - com.example.movieMaster
 - data
 - network
 - Entities.kt
 - MoviesLocalDataSource
 - MoviesRepository
 - MovieRepositoryImpl
 - ui
 - ApplicationOverride
 - com.example.movieMaster (androidTest)
 - com.example.movieMaster (test)- data
 - FakeMovieDataSource
 - FakeMoviesRepository

The right pane displays the code for `FakeMoviesRepository.kt`:1 package com.example.movieMaster.data
2
3 class FakeMoviesRepository : MoviesRepository {
4
5 override suspend fun getMoviesResponse(): MoviesResponse {
6 TODO(reason: "Not yet implemented")
7 }
8
9 }

Copia e aggiungi fakeData da `FakeMovieDataSource` e usalo per il risultato di `getMoviesResponse`

```
val fakeData = MoviesResponse(
    movies = listOf(Movie(id = 999, title = "test"))
)

override suspend fun getMoviesResponse(): MoviesResponse {
    return fakeData
}
```

Abbiamo copiato `fakeData` dentro il nostro `fakeRepo` perché così questo non sarà dipendente da `FakeMovieDataSource`.

Test

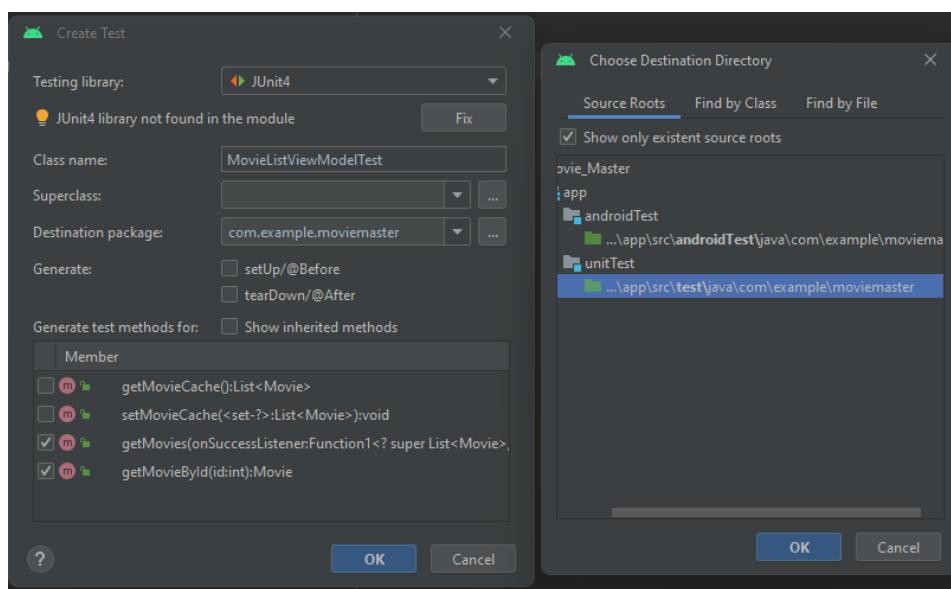
Per prima cosa modifichiamo il costruttore del MovieListViewModel per integrare la nuova interfaccia:

```
class MovieListViewModel(private val repository: MoviesRepository = MoviesRepositoryImpl()) : ViewModel() {
```

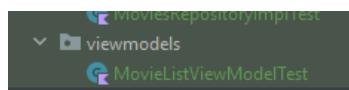
Rendi movieCache pubblico, ci servirà per i test. Aggiungi l'annotazione @VisibleForTesting per lanciare warning se viene utilizzato da pubblico al di fuori dell'ambito dei test.

```
@VisibleForTesting(otherwise = VisibleForTesting.PRIVATE)
var movieCache = listOf<Movie>()
```

Poi crea un test, selezionando i metodi da testare:



Spostalo dentro un nuovo package `viewModels`



Effettua il setup, inserendo le annotations e i membri di test

```
@RunWith(AndroidJUnit4::class)
class MovieListViewModelTest {

    val fakeRepo = FakeMoviesRepository()

    //test subject
    lateinit var viewModel: MovieListViewModel

    @Before
    fun setup() {
        viewModel = MovieListViewModel(fakeRepo)
    }

    @Test
    fun getMovies() {
    }

    @Test
    fun getMovieById() {
    }
}
```

Ora puoi testare le funzioni come al solito. Se dovessi testare *suspend fun* utilizza *runTest*, ma ricordati di aggiungere l'annotazione *@ExperimentalCoroutinesApi*

3. Esercizio

Completa i test di MovieListViewModelTest.

Per getMovies i test devono controllare:

- L'ordinazione della lista prima della pubblicazione. Aggiungi dati a fakeData.
 - Per controllare la lista film, assicurati di prendere come riferimento fakeRepo.fakeData, non movieCache.
- la corretta interazione con movieCache. Devono inoltre venir effettuati test sia in caso che la cache parta da una lista vuota, che da una lista piena.

Per getMovieById:

- Ci sono dei casi in cui questa funzione produce eccezione. Poniti come obiettivo di fare in modo che non produca mai eccezione.
- **Lavora facendo prima i test** per i risultati che dovrebbe produrre questa funzione e **dopo modifica la funzione** per fare in modo che abbia le funzionalità desiderate e passi i test.
 - Così starai lavorando con la filosofia **Test Driven Development (TDD)**
- Per testare correttamente questa funzione serviranno almeno 3 test.

Livedata e Observer Pattern

1. Observer Pattern

Finora abbiamo interagito con il risultato di operazioni asincrone tramite i listener:

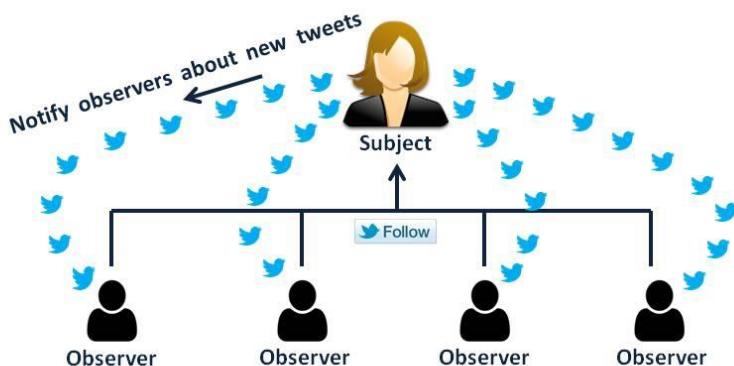
```
model.getMovies { it: List<Movie>
    movieListAdapter.apply { this: MovieListAdapter
        dataset = it
        notifyDataSetChanged()
    }
}
```

Ora vedremo come è possibile fare in modo di osservare direttamente i dati senza passare per una funzione intermedia, e fare in modo che la UI reagisca al cambio del valore di questi dati.

Ci si riferisce a questo concetto come **Reactive Programming**. Ci sono molti modi per implementarlo, noi vedremo l'**observer pattern**.

Observer Pattern definisce una dipendenza uno-a-molti tra gli oggetti in modo che quando un oggetto (Subject) cambia stato, tutti gli oggetti dipendenti da lui (Observers) vengono notificati e aggiornati automaticamente.

Observer Design Pattern



www.codepumpkin.com

In questo pattern, ci sono molti **Observers** che osservano un particolare **Subject**. Gli osservatori vogliono essere informati con delle **notifiche** quando viene apportata una modifica al valore di Subject: effettuano dunque una registrazione a Subject. Quando perdono interesse, annullano la registrazione.

2. LiveData

Android implementa l'observer pattern tramite **LiveData**

LiveData wrappa dei dati e li rende osservabili. Inoltre è Lifecycle-aware, si occupa quindi automaticamente di gestire le sottoscrizioni in base ai lifecycle.

Simile alle liste **LiveData è immutabile, MutableLiveData è mutable**.

Subject

Aggiungi livedata a MovieListViewModel:

```
@VisibleForTesting(otherwise = VisibleForTesting.PRIVATE)
private val _movies = MutableLiveData<List<Movie>>()
val movies: LiveData<List<Movie>> = _movies
```

- `_movies` è un MutableLiveData, creato con il costruttore che ne identifica il tipo. Inizialmente, non ha valore.
- `movies` è invece un Livedata immutabile, esposto pubblico alla View. View osserverà questa val per ottenere i dati dei film, e non chiamerà più `getMovies`.
- `movies` non è nient'altro che un *cast* di `_movies` a LiveData
- Rimuovi `movieCache`: useremo `_movies` come cache ora.

Modifica `getMovies`:

```
fun getMovies() = viewModelScope.launch { this: CoroutineScope
    repository.getMoviesResponse().movies
        .sortedByDescending { it.release_date }
        .let { _movies.value = it }
}
```

- rimuovi il listener
- il risultato viene impostato come valore di `movies`

Ora il ruolo di `getMovies` è impostare il valore di `_movies`.

`getMovies` non verrà più invocato dalla view, quindi facciamolo partire appena viene creato il viewmodel:

```
init {
    getMovies()
}
```

Infine, modifica `getMovieById` per far modo che si appoggi a `_movies` come cache:

```
fun getMovieById(id: Int) =
    _movies.value!!.first { it.id == id }
```

Observer

In MovieListFragment cambia `getMovies` con l'osservazione di `movies`, tramite la funzione `.observe`

```
model.movies.observe(viewLifecycleOwner) { it: List<Movie>!
    movieListAdapter.apply { this: MovieListAdapter
        dataset = it
        notifyDataSetChanged()
    }
}
```

`observe` verrà lanciato **ogni volta che il valore di movies viene modificato**.

`observe` richiede un `lifecycleOwner`. Mentre in una Activity è possibile utilizzare l'activity stessa, in un fragment è necessario utilizzare `viewLifecycleOwner`.

Ricapitolando ora il flusso della nostra app:

- Quando viene creato il view model, viene lanciata la chiamata network di `getMovies`
- il fragment (`lifecycleOwner`) si registra come osservatore ai cambiamenti del Livedata `movies`
- la risposta arriva dal servizio e `getMovies` aggiorna il valore di `movies`
- appena il valore di `movies` viene modificato, il fragment riceve una notifica e aggiorna l'interfaccia

Questo farà sì che, indipendentemente da chi modificherà il valore di `movies`, la nostra interfaccia sarà sempre aggiornata.

View osserva il valore di `movies`, e non sa chi o cosa lo modifica.

3. setValue, postValue

Prima abbiamo impostato il valore di `_movies` con un semplice `set`

```
repository.getMoviesResponse().movies
    .sortedByDescending { it.release_date }
    .let { _movies.value = it }
```

Questa è la abbreviazione di Kotlin del setter `setValue`. Livedata però ha 2 setters:

- `setValue` imposta il valore in maniera **sincrona**
- `postValue` imposta il valore in maniera **asincrona**

Solitamente si usa `postValue`. `SetValue` è utilizzato solo nei casi in cui sia necessario mantenere una specifica sincronicità.

4. Testare LiveData

Testare un oggetto osservabile può essere sfidante, perché il suo valore puo' cambiare all'interno del test stesso.

Testeremo gli eventi in maniera **sincrona**, cosi' da poter sempre ripetere il test e non [renderlo flaky](#).

Crea LiveDataTestUtil dentro la cartella di test e copia questa classe:

```
import androidx.annotation.VisibleForTesting
import androidx.lifecycle.LiveData
import androidx.lifecycle.Observer
import java.util.concurrent.CountDownLatch
import java.util.concurrent.TimeUnit
import java.util.concurrent.TimeoutException

@VisibleForTesting(otherwise = VisibleForTesting.NONE)
fun <T> LiveData<T>.getOrAwaitValue(
    time: Long = 2,
    timeUnit: TimeUnit = TimeUnit.SECONDS,
    afterObserve: () -> Unit = {}
) : T {
    var data: T? = null
    val latch = CountDownLatch(1)
    val observer = object : Observer<T> {
        override fun onChanged(o: T?) {
            data = o
            latch.countDown()
            this@getOrAwaitValue.removeObserver(this)
        }
    }
    this.observeForever(observer)
    try {
        afterObserve.invoke()
        // Don't wait indefinitely if the LiveData is not set.
        if (!latch.await(time, timeUnit)) {
            throw TimeoutException("LiveData value was never set.")
        }
    } finally {
        this.removeObserver(observer)
    }
}

@SuppressWarnings("UNCHECKED_CAST")
return data as T
}
```

Questo ci permetterà di bloccare l'esecuzione del test finchè non riceviamo il risultato di livedata.

```
@Test
fun getMovies() = runTest{ this: TestScope
    fakeRepo.fakeData = MoviesResponse(movies = listOf(Movie(id = 777)))

    viewModel.getMovies()

    val result = viewModel.movies.getOrAwaitValue()
    assertThat(result.first().id).isEqualTo( expected: 777)
}
```

Utilizzeremo getOrAwaitValue come se fosse un normale getter.

5. Esercizio

- Nella pagina di dettaglio, mostra i primi 6 attori presenti nel film.
- Dovrai integrare una nuova chiamata a TMDB:
<https://developer.themoviedb.org/reference/movie-credits>
- Utilizza coroutines e livedata.
- Effettua i test.

H.1 - Dependency Injection e Hilt

1. DI in Android

Diamo un'occhiata al costruttore di MovieListViewModel

```
class MovieListViewModel(
    private val repository: MoviesRepository = MoviesRepositoryImpl(),
) : ViewModel() {
```

Abbiamo fornito al viewModel tutte le sue dipendenze qui, non creando al suo interno istanze di altre classi. Questo pattern è definito **Dependency Injection**

Dependency Injection è un pattern di progettazione in cui un oggetto o una funzione riceve gli altri oggetti o funzioni da cui dipende

È una pratica fondamentale per ottenere un'app manutenibile e testabile.

Man mano che l'app cresce però, questo richiede un gran quantità di *boilerplate code* per instanziare una classe che, magari, fornisce solo poche funzionalità.

Vediamo le dipendenze del viewModel:

MovieListViewModel > MoviesRepository > MovieNetworkApi > Retrofit

Se dovessimo creare un'altra istanza del viewModel per riutilizzarlo in un'altra parte dell'app, dovremmo ricrearene anche tutte le dipendenze, magari cambiando l'implementazione di MoviesRepository, generando altro boilerplate. Questo codice cresce esponenzialmente con la crescita della nostra app.

Scrivere molto codice di setup non solo non è efficiente, ma è anche prone a errori umani.

Vediamo ora come utilizzare **HILT**, un framework di Dependency Injection che ci permetterà di automatizzare alcune di queste procedure.

Hilt richiede un importante setup, non solo delle dipendenze, ma anche nel dichiarare le injections. Teniamo sempre a mente però che nel lungo termine, questo ci permetterà di risparmiare tempo ed effettuare test altrimenti impossibili.

2. Hilt Project Setup

<https://developer.android.com/training/dependency-injection/hilt-android#setup>

E' consigliabile seguire sempre il setup presente nella documentazione quando si effettua in un nuovo progetto, per evitare problemi derivanti dalle ultime modifiche al framework.

build.gradle:project

```
plugins {
...
    id 'com.google.dagger.hilt.android' version "2.44" apply false
}

build.gradle:app

...
plugins {
...
    id 'kotlin-kapt'
    id 'com.google.dagger.hilt.android'
}

android {
...
    compileOptions {
        sourceCompatibility JavaVersion.VERSION_17
        targetCompatibility JavaVersion.VERSION_17
    }
    kotlinOptions {
        jvmTarget = '17'
    }
}

dependencies {
...
    // Hilt
    implementation "com.google.dagger:hilt-android:2.44"
    kapt "com.google.dagger:hilt-compiler:2.44"
    // // Inject of viewModel scoped to navGraph with: by hiltNavGraphViewModels
    implementation 'androidx.hilt:hilt-navigation-fragment:1.0.0'
}
// Allow references to generated code
kapt {
    correctErrorTypes true
}
```

È consigliato estrarre la versione (2.44 nell'esempio) in una variabile, così da coordinare gli import.

Application override

```
@HiltAndroidApp
class ExampleApplication : Application() { ... }
```

3. Entry Point

Dopo aver impostato l'Application, dobbiamo dire a Hilt dove intervenire nel framework Android.

In questo caso è il Fragment, che userà il ViewModel, annotiamolo con @AndroidEntryPoint

```
@AndroidEntryPoint  
class MovieDetailFragment : Fragment(R.layout.fragment_movie_detail) {
```

Facendo così stiamo creando un **Container** con scopo Fragment, e preparando il fragment a ricevere le Injection.

Avendo definito MovieDetailFragment come entry point, dobbiamo annotare anche *tutte le classi che dipendono da lui*, in questo caso, l'activity:

```
@AndroidEntryPoint  
class MainActivity : AppCompatActivity() {
```

Hilt currently supports the following Android classes:

- Application (by using @HiltAndroidApp)
- ViewModel (by using @HiltViewModel)
- Activity
- Fragment
- View
- Service
- BroadcastReceiver

4. Inject: constructor

Cominciamo dal basso a risolvere le dipendenze: RetrofitClient

```
class RetrofitClient @Inject constructor(){  
    private val movieBaseAddress = "https://api.themoviedb.org/3/  
    private val authLabel = "Authorization"  
    private val authValue = "Bearer " + BuildConfig.TMDB_API_KEY  
}
```

Aggiungendo @Inject al constructor() (ora dichiarato verbosamente) questa classe:

- Potrà ricevere Injection per le sue dipendenze
- Potrà essere iniettata come dipendenza

RetrofitClient non ha dipendenze, quindi possiamo lasciarlo vuoto

(In realtà è dipendente da Gson(), una classe al di fuori del nostro controllo, ma per ora ignoralo)

5. Inject: Module

Il nostro Repo ha bisogno di uno specifico valore esposto da RetrofitClient:

```
class MoviesRepositoryImpl(  
    private val movieRemoteDataSource: MovieNetworkAPI =
```

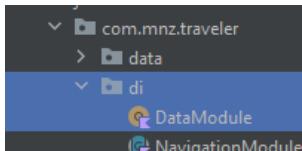
Rimuovi il default dal costruttore - forniremo questa dipendenza tramite iniezione:

```
class MoviesRepositoryImpl (  
    private val movieRemoteDataSource: MovieNetworkAPI  
) : MoviesRepository {
```

Sorge quindi un problema: MovieNetworkAPI è una Interfaccia, che non puo' essere instanziata ma solo implementata, non possiamo dunque usare @Inject constructor()

Andiamo quindi a vedere come dichiarare un **Modulo** dove definiremo quale implementazione di MovieNetworkAPI verrà ricevuta da MoviesRepositoryImpl.

Crea un nuovo Object chiamato DataModule dentro un nuovo package DI



```
@Module
@InstallIn(ViewModelComponent::class)
object DataModule {
```

- `@Module` lo definisce come Hilt Module
- `@InstallIn` definisce lo scope, ovvero il lifecycle di questo modulo. In questo caso, scegliamo il `viewModel`, che è dove sarà consumato il Repository – il nostro obiettivo.

Definiamo l'inject di MovieNetworkApi necessario al nostro repository:

```
@Provides
fun provideMoviesDataSource(retrofitClient: RetrofitClient): MovieNetworkAPI =
    retrofitClient.movieRemoteDataSource
```

- `@Provides` definisce a Hilt una funzione che fornisce un'implementazione
- indica che questo costruttore ottiene la sua dipendenza tramite iniezione
- indica che questa funzione inietta il suo risultato a qualcuno che ne è dipendente

Crea un'altra funzione che fa la stessa cosa per il repository, dato che anche lui estende una interfaccia:

```
@Provides
fun provideMovieRepository(movieRemoteDataSource: MovieNetworkAPI): MoviesRepository =
    MoviesRepositoryImpl(movieRemoteDataSource)
```

Infine, modifica MovieListViewModel per ricevere l'iniezione

```
@HiltViewModel
class MovieListViewModel @Inject constructor(
    private val repository: MoviesRepository
) : ViewModel()
```

- `@HiltViewModel` crea un **hilt container** che è **scoped** al `viewModel`

Ricapitolando:

```
class RetrofitClient @Inject constructor() {
```

Fornisce a..

```
@Provides  
fun provideMoviesDataSource(retrofitClient: RetrofitClient): MovieNetworkAPI =  
    retrofitClient.movieRemoteDataSource
```

Fornisce a..

```
@Provides  
fun provideMovieRepository(movieRemoteDataSource: MovieNetworkAPI): MoviesRepository =  
    MoviesRepositoryImpl(movieRemoteDataSource)
```

Fornisce a...

```
@HiltViewModel  
class MovieListViewModel @Inject constructor(  
    private val repository: MoviesRepository  
) : ViewModel()
```

Cosa abbiamo ottenuto con questa procedura:

Ogni elemento di questa catena puo' essere riutilizzato e iniettato in altre implementazioni.

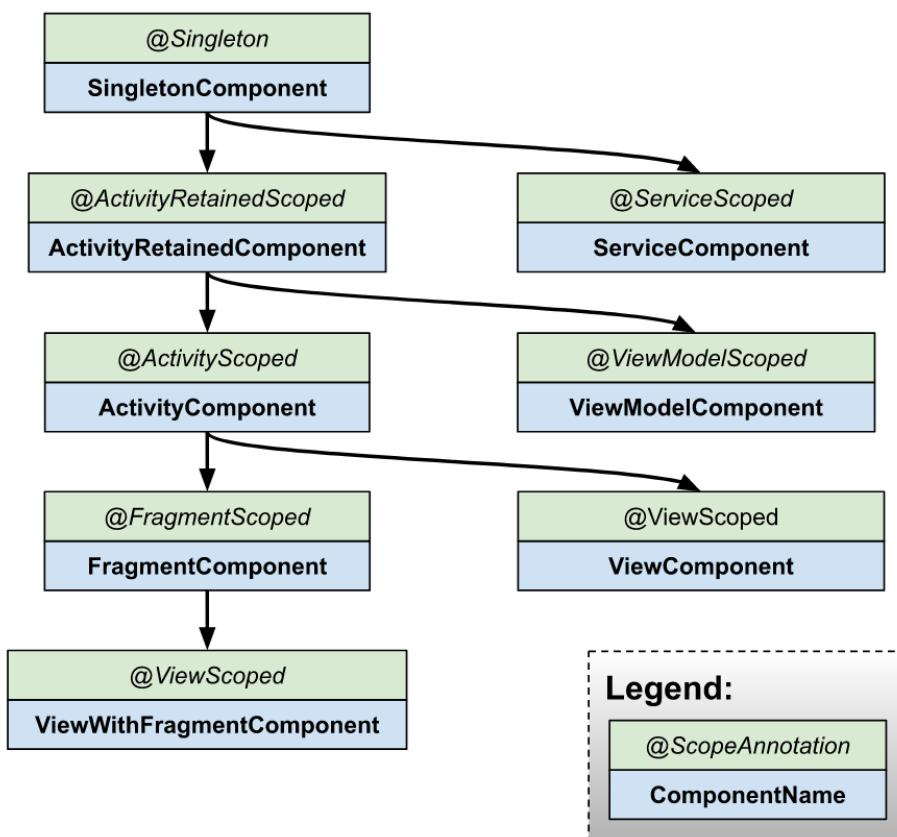
Ogni elemento puo' essere testato iniettando facilmente *test doubles*

Ora ogni volta che richiameremo MovieListViewModel tramite *byViewModels* o simili, le sue dipendenze verranno create e legate al suo lifecycle, quindi distrutte se esso viene distrutto.

Questo avviene perché il modulo ha lo scope `@InstallIn(ViewModelComponent::class)`, che è la classe dei container creati dall'annotazione `@HiltViewModel`. Il modulo sarà quindi installato in ogni container `@HiltViewModel`, e disponibile ad ogni viewModel che ha questa annotazione.

Gerarchia dei Componenti

Quando un modulo o altri elementi vengono installati in un componente, questi saranno disponibili non solo ad esso, ma anche a tutti i suoi discendenti:



6. Instrumented Test Setup

Per effettuare Unit Test con Hilt di solito non è necessaria alcuna modifica.

Per gli instrumented test dobbiamo invece collegare hilt al test runner e creare una activity di test.

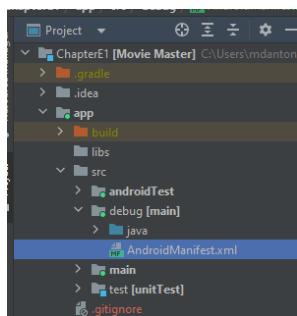
```
build.gradle:app
// Hilt For instrumented tests.
androidTestImplementation "com.google.dagger:hilt-android-
testing:$hilt_version"
// ...with Kotlin.
kaptAndroidTest "com.google.dagger:hilt-android-compiler:$hilt_version"
```

TestManifest e TestActivity

Usando *New > Directory*, crea una cartella **debug** dentro /app/src

Dentro questa cartella:

- Crea una sottocartella /java
- crea un nuovo AndroidManifest.xml



Contenuto Android Manifest.xml :

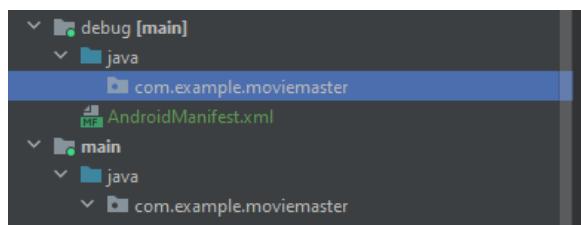
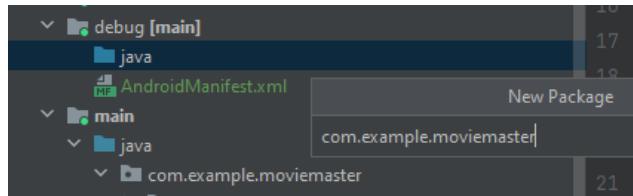
```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.movieMaster">

    <application>
        <activity
            android:name=".HiltTestActivity"
            android:exported="false" />
    </application>

</manifest>
```

Fai attenzione che il campo *package* sia compilato correttamente per il tuo progetto.

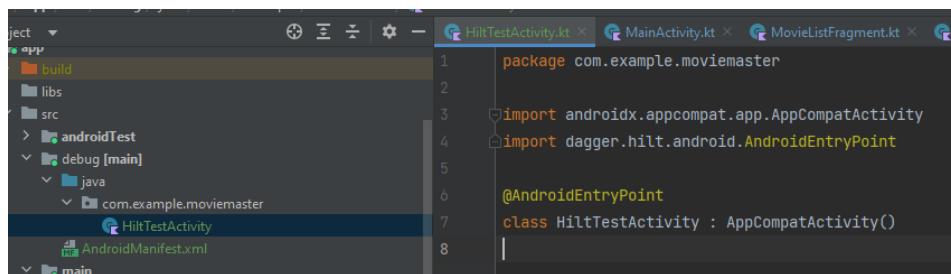
Dentro la cartella `java`, crea un nuovo package con `new > package`. Dagli lo stesso nome del tuo package dentro `main`



Dentro `movieplayer [debug]`, crea questa classe:

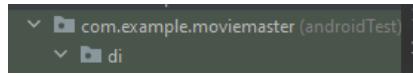
```
import androidx.appcompat.app.AppCompatActivity
import dagger.hilt.android.AndroidEntryPoint

@AndroidEntryPoint
class HiltTestActivity : AppCompatActivity()
```



TestRunner e Extensions

Aggiungi un package *di* dentro androidTest



Crea questa classe dentro il package *di*

```
import android.app.Application
import android.content.Context
import androidx.test.runner.AndroidJUnitRunner
import dagger.hilt.android.testing.HiltTestApplication

class HiltTestRunner : AndroidJUnitRunner() {
    override fun newApplication(cl: ClassLoader?, name: String?, context: Context?): Application {
        return super.newApplication(cl, HiltTestApplication::class.java.name,
        context)
    }
}
```

build.gradle:app, dentro *defaultConfig*, sostituisci il test runner con quello che hai appena creato

```
testInstrumentationRunner "com.example.movieMaster.di.HiltTestRunner"
```

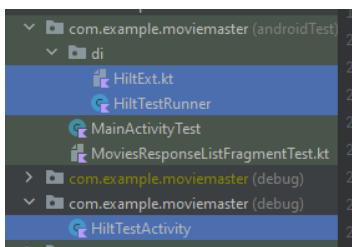
Crea questa classe *HiltExt* dentro il package di

```
import android.content.ComponentName
import android.content.Intent
import android.os.Bundle
import androidx.annotation.StyleRes
import androidx.core.util.Preconditions
import androidx.fragment.app.Fragment
import androidx.navigation.NavHostController
import androidx.navigation.Navigation
import androidx.test.core.app.ActivityScenario
import androidx.test.core.app.ApplicationProvider
import com.example.moviemaster.HiltTestActivity
import androidx.fragment.testing.manifest.R

/**
 * launchFragmentInContainer from the androidx.fragment:fragment-testing library
 * is NOT possible to use right now as it uses a hardcoded Activity under the hood
 * (i.e. [EmptyFragmentActivity]) which is not annotated with @AndroidEntryPoint.
 *
 * As a workaround, use this function that is equivalent. It requires you to add
 * [HiltTestActivity] in the debug folder and include it in the debug AndroidManifest.xml file
 * as can be found in this project.
 */
inline fun <reified T : Fragment> launchFragmentInHiltContainer(
    fragmentArgs: Bundle? = null,
    @StyleRes themeResId: Int = R.style.FragmentScenarioEmptyFragmentActivityTheme,
    navHostController: NavHostController? = null,
    crossinline action: Fragment.() -> Unit = {}
): ActivityScenario<HiltTestActivity> {
    val startActivityIntent = Intent.makeMainActivity(
        ComponentName(
            ApplicationProvider.getApplicationContext(),
            HiltTestActivity::class.java
        )
    ).putExtra(
        "androidx.fragment.app.testing.FragmentScenario.EmptyFragmentActivity.THEME_EXTRAS_BUNDLE_KEY",
        themeResId
    )

    return ActivityScenario.launch<HiltTestActivity>(startActivityIntent).apply {
        onActivity { activity ->
            val fragment: Fragment = activity.supportFragmentManager.fragmentFactory.instantiate(
                Preconditions.checkNotNull(T::class.java.classLoader),
                T::class.java.name
            )
            fragment.arguments = fragmentArgs
            fragment.viewLifecycleOwnerLiveData.observeForever { viewLifecycleOwner ->
                if (viewLifecycleOwner != null) {
                    navHostController?.let {
                        Navigation.setViewNavController(fragment.requireView(), it)
                    }
                }
            }
            activity.supportFragmentManager
                .beginTransaction()
                .add(android.R.id.content, fragment, "")
                .commitNow()
        }
        fragment.action()
    }
}
```

Ecco come appare l'albero ora



7. Instrumented Test

Testiamo MovieDetailFragment:

Test Setup

Creiamo la classe di test:

```
@HiltAndroidTest
@MediumTest
@RunWith(AndroidJUnit4::class)
class MovieDetailFragmentTest {

    @get:Rule
    var hiltRule = HiltAndroidRule(testInstance: this)

    @Before
    fun before() {
        hiltRule.inject()
    }
}
```

The code block shows a Java class named 'MovieDetailFragmentTest' annotated with '@HiltAndroidTest', '@MediumTest', and '@RunWith(AndroidJUnit4::class)'. It contains a field 'hiltRule' of type 'HiltAndroidRule' and a method 'before()' which calls 'inject()' on 'hiltRule'.

- @HiltAndroidTest permette a Hilt di Iniettare dipendenze
- hiltRule fa sì che tutto sia iniettato prima di cominciare i test
- Entrambi sono necessari per testare con Hilt.

Creiamo lo scenario:

```
lateinit var scenario: ActivityScenario<HiltTestActivity>

@Before
fun before() {
    hiltRule.inject()

    scenario =
        launchFragmentInHiltContainer<MovieDetailFragment>()
}
```

- Utilizziamo la extension per creare uno scenario che abbia come base l'activity che abbiamo creato dentro *debug*. E' necessario perché, come abbiamo visto, Hilt necessita per ogni fragment annotato con `@AndroidEntryPoint`, che ci sia tale annotazione anche nell'activity che lo contiene.

Ricreare Fragment dependencies

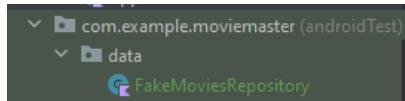
MovieDetail fragment ha le seguenti dipendenze

- E' dipendente da MovieListViewModel
- Richiede un **id** al momento della creazione
-

MovieListViewModel

Ora cominciamo a sfuggire Hilt per diminuire il carico di boilerplate.

Copia FakeMoviesRepository che abbiamo creato per gli UnitTest, dentro androidTest\data



Dentro la cartella DI, crea un nuovo modulo Hilt, *FakeDataModule*



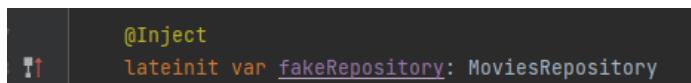
FakeDataModule *replaces* DataModule nei tests.

Ora MovieViewModel, che dipende da MoviesRepository, sarà iniettato in tutti i test senza bisogno di dichiararlo.

Movie ID

MovieDetailFragment richiede un ID al momento della creazione, che nell'app è passato da MovieList tramite gli *args*. Dobbiamo fornire questo *bundle*, e lo faremo in maniera molto simile a come è nell'app.

Ottieni un riferimento al Repo:



- Ottieniamo questo repo tramite provideRepository che abbiamo scritto nel fakeDataModule

Bisogna castare MoviesRepository a FakeMoviesRepository se vogliamo ottenere i suoi membri interni, come fakeData. Scriviamo una val che ci permette di evitare il cast ad ogni invocazione:

```
 @Inject  
lateinit var fakeRepository: MoviesRepository  
val fakeData by lazy { (fakeRepository as FakeMoviesRepository).fakeData }
```

Ora usa fakeData :

```
@Inject
lateinit var fakeRepository: MoviesRepository
val fakeData by lazy { (fakeRepository as FakeMoviesRepository).fakeData }

lateinit var scenario: ActivityScenario<HiltTestActivity>

@Before
fun before() {
    hiltRule.inject()

    val bundle = MovieDetailFragmentArgs(fakeData.movies[0].id).toBundle()
    scenario =
        launchFragmentInHiltContainer<MovieDetailFragment>(bundle, R.style.Theme_MovieMaster)
}
```

- MovieDetailFragmentArgs è lo stesso utilizzato nella navigazione dell'app. toBundle genera il bundle che possiamo passare al fragment.
- Aggiungiamo anche lo stile dell'app, cosi' il fragment sara renderizzato come nell'app.

Aggiungi un semplice test

```
@Test
fun dataShown_ok() {
    onView(withId(R.id.detail_title_tv)).check(matches(withText(fakeData.movies[0].title)))
}
```

I.1 - SharedPreferences

1. Scopo delle SharedPreferences

Le SharedPreferences sono un modo semplice per memorizzare dei dati su disco che non richiedano particolari strutture o coordinazione, fornendo al contempo dei metodi semplici di lettura e scrittura. Sono una lista di elementi chiave-valore, i cui valori possono essere solo delle primitive o delle stringhe.

Sono solitamente utilizzate per memorizzare dati di setup dell'app che devono permanere tra una sessione e l'altra, o semplici dati che non richiedono un database.

2. PreferenceManager

Per recuperare una istanza di SharedPreferences, utilizzeremo PreferenceManager.

Tramite questa classe è possibile creare, recuperare e cancellare istanze di SharedPreferences.

Tra queste, è presente un'istanza di default, che andiamo ad usare

Aggiungi a DataModule:

```
    @Provides
    fun provideSharedPreferences(@ApplicationContext applicationContext: Context): SharedPreferences =
        PreferenceManager.getDefaultSharedPreferences(applicationContext)
```

- PreferenceManager richiede sempre un context per creare o recuperare istanze.
- Hilt ci fornisce due context, Application e Activity, tramite annotations.

MovieListViewModel:

```
@HiltViewModel
class MovieListViewModel @Inject constructor(
    private val repository: MoviesRepository,
    private val preferences: SharedPreferences
) : ViewModel()
```

3. Salvare e Recuperare valori

Puoi utilizzare il builder di SharedPreferences per salvare dati su disco, in modo che i preferiti permangano tra le sessioni.

```
private fun saveFavouriteStatus(id: Int) {
    val movie = _movies.value!!.first { it.id == id }
    preferences.edit()
        .putBoolean(id.toString(), movie.isFavourite)
        .apply()
}
```

- .edit() comincia la modifica, .apply() la applica
- È necessario utilizzare i metodi in base al tipo di dato che si vuole salvare (Boolean, stringa..)
- I metodi richiedono una stringa (key) e un valore (value)
- Creano un record, o lo aggiornano se già presente.

Recuperare i valori è simile:

```
private fun retrieveFavouriteStatus(id:Int): Boolean {
    return preferences.getBoolean(id.toString(), false)
}
```

- C'è un metodo per ogni primitiva (getBoolean, getInteger...)
- I metodi richiedono la chiave e un default in caso il valore non venisse trovato.

4. Unit Test

Nei test le preferences non vanno testate, ma ci sono molte interazioni con le nostre classi.

Negli Instrumented test non c'è problema, dato che abbiamo a disposizione il contesto reale dell'app, e Hilt effettua l'iniezione dove necessario.

Negli UnitTest invece, creiamo le preferences in questo modo:

```
lateinit var preferences: SharedPreferences  
└ Manolo D'Antonio *  
@Before  
fun setup() {  
    val context: Context = ApplicationProvider.getApplicationContext()  
    preferences = PreferenceManager.getDefaultSharedPreferences(context)  
    viewModel = MovieListViewModel(fakeRepo, preferences)  
}
```

- ApplicationProvider.getApplicationContext() fornisce il contesto reale del nostro test. Questa funzione è **utilizzabile solo dentro @Before**

Nei test è importante ricordare di non testare la funzionalità di SharedPreferences, ma quella della nostra funzione.

```
@Test  
fun saveFavouriteStatus_empty_saved (){  
    assertThat(preferences.getBoolean("999", false)).isFalse()  
    viewModel.changeMovieFavourite( id: 999)  
    assertThat(preferences.getBoolean("999", false)).isTrue()  
}
```

Sebbene indirettamente, abbiamo testato il corretto comportamento di changeMovieFavourite.

5. Esercizio

L'app deve memorizzare i preferiti, che devono permanere tra sessioni.

Salva i film preferiti dell'utente, al momento del click della stella

Quando carichi getMovies, cerca se nelle preferences il film è un preferito, e aggiorna il dataset prima di pubblicarlo con livedata.

- Modificare un elemento di una lista in livedata NON genera una notifica da livedata. In questo caso, si modifica la lista, e poi si re-imposta la lista come contenuto del livedata, con setValue o.postValue.

I.2 - Room (DB)

1. Scopo del DB

Lo scopo principale del database in un'app Android è di limitare le richieste network e consentire all'utente di utilizzare l'app anche quando la rete non è disponibile – questo concetto è definito **Offline First**

Un database si utilizza quando è necessario memorizzare dati strutturati e sincronizzati.

Android utilizza una libreria ufficiale per implementare il database, chiamata **Room**

Una implementazione di Room è composta da 3 elementi: **Data Entity**, **DAO** e **Database**

2. Room setup

```
Build.gradle:app
plugins {
    ...
    id 'kotlin-kapt'
}

//Room
def room_version = "2.5.1"
implementation "androidx.room:room-runtime:$room_version"
kapt "androidx.room:room-compiler:$room_version"
```

Alla fine del file (dopo `dependencies{}`) aggiungere a livello root:

```
// Allow references to generated code
kapt {
    correctErrorTypes true
}
```

Data Entity

Room utilizza le data class per definire tabelle e record

```
└─ Manolo D'Antonio *
  @Entity
  data class Movie(
      @PrimaryKey val id: Int = 0,
      val adult: Boolean = false,
      val backdrop_path: String = "",
      val genre_ids: List<Int> = listOf()
```

Questo codice definisce un'entità dati Movie. Ogni istanza di Movie rappresenta una riga in una tabella `movie` nel database dell'app.

`@Entity` definisce una data entity

Il nome della classe Movie sarà il nome della tabella.

I campi di Movie saranno i nomi delle colonne della tabella.

`@PrimaryKey` è un campo *mandatory* di ogni entity, definisce la primary key nel db.

Type Converters

Room non converte da solo gli oggetti: è lasciato a noi definire come si vogliono salvare oggetti complessi nel db.

Nel nostro Movie c'è una List<Int> che non puo' essere salvata automaticamente.

Per adesso, usiamo una soluzione semplice: convertiamo un oggetto in stringa con GSON.

Crea db/TypeConverters.kt con questo contenuto

```
class MovieListTypeConverter {
    private val gson = Gson()
    private val movieListType = object : TypeToken<List<Int?>?>() {}.type

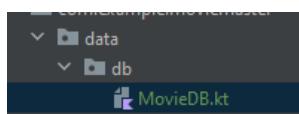
    @TypeConverter
    fun toIntList(data: String?): List<Int> =
        data?.let { gson.fromJson(it, movieListType) } ?: emptyList()

    @TypeConverter
    fun intListToString(data: List<Int?>?): String =
        gson.toJson(data)
}
```

Data access object (DAO)

Il DAO è un'interfaccia che definirà le funzioni con le quali interagirà il resto dell'app.

Crea un nuovo file `data/db/MovieDB`



Il suo contenuto:

```
1  @Dao
2  interface MovieDao {
3      @Insert(onConflict = OnConflictStrategy.REPLACE)
4      fun insertMovies(vararg movies: Movie)
5
6      @Update
7      fun updateMovies(vararg movies: Movie)
8
9      @Delete
10     fun deleteMovies(vararg movies: Movie)
11
12     @Query("SELECT * FROM movie")
13     fun getAll(): List<Movie>
14
15 }
```

- Le annotazioni Insert, Update e Delete, sono delle abbreviazioni per i relativi metodi di query
- In query è necessario scrivere il codice SQL

Database

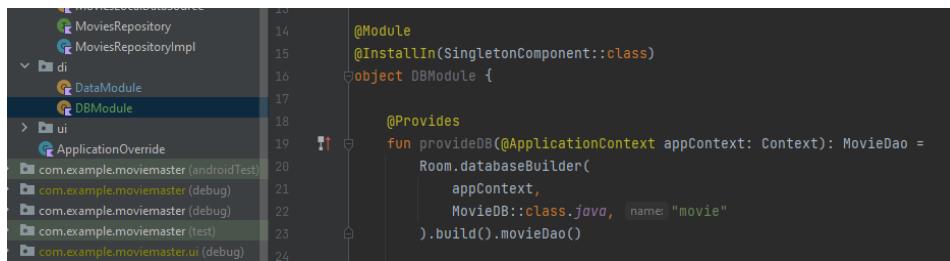
Dentro `MovieDB` aggiungi:

```
4
5  @Database(entities = [Movie::class], version = 1)
6  @TypeConverters(MovieListTypeConverter::class)
7  abstract class MovieDB : RoomDatabase() {
8      abstract fun movieDao(): MovieDao
9  }
```

- La classe deve essere annotata con un'annotazione `@Database` che include un array di entità che elenca tutte le entità di dati associate al database.
- `@TypeConverters` aggiunge uno o piu' convertitori per oggetti complessi.
- La classe deve essere una classe astratta che estende `RoomDatabase`.
- Per ogni classe DAO associata al database, la classe database deve definire un metodo astratto con zero argomenti e restituire un'istanza della classe DAO.

Implementazione, Modulo Hilt Singleton

Per l'occasione, crea un nuovo modulo Hilt *di/DBModule*



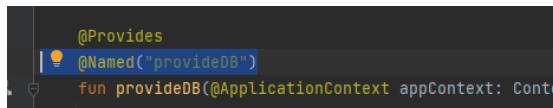
```
14  @Module
15  @InstallIn(SingletonComponent::class)
16  object DBModule {
17
18      @Provides
19      fun provideDB(@ApplicationContext appContext: Context): MovieDao =
20          Room.databaseBuilder(
21              appContext,
22              MovieDB::class.java, "movie"
23          ).build().movieDao()
24 }
```

- Installiamo questo modulo nel container Application, rendendolo un singleton. L'istanza di un DB è piuttosto pesante, e avere più copie del DB potrebbe creare problemi di sincronizzazione. Utilizziamo quindi un singolo DB.

Ora sorge un problema: abbiamo fornito all'app il repository MoviesRepositoryImpl tramite provideMovieRepository nel modulo DataModule, nel quale però non è presente provideDB, che appartiene al nuovo DBModule.

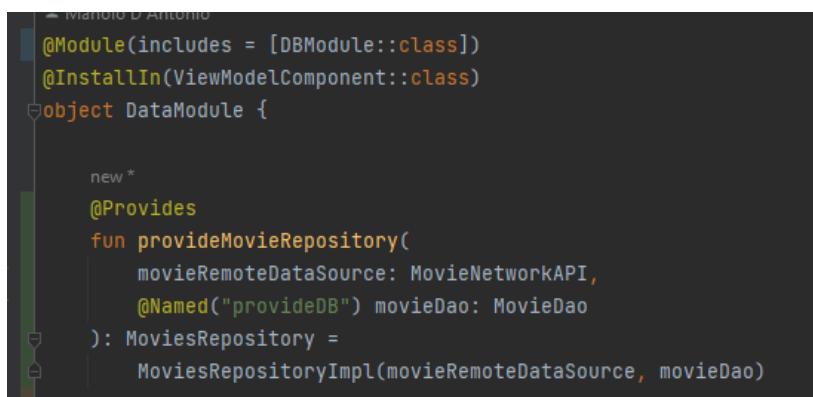
DataModule è scoped al viewModel: quindi se vogliamo mantenere il suo scope, non possiamo cambiarlo in singleton. Decidiamo quindi che abbia un riferimento al DBModule.

A DBModule.provideDB aggiungi questa annotation:



```
@Provides
@Named("provideDB")
fun provideDB(@ApplicationContext appContext: Conte
Room
```

Questo ci permette di richiamare direttamente la iniezione. In DataModule fai questa modifica:



```
@Module(includes = [DBModule::class])
@InstallIn(ViewModelComponent::class)
object DataModule {

    new *
    @Provides
    fun provideMovieRepository(
        movieRemoteDataSource: MovieNetworkAPI,
        @Named("provideDB") movieDao: MovieDao
    ): MoviesRepository =
        MoviesRepositoryImpl(movieRemoteDataSource, movieDao)
```

- **@Module(includes = [DBModule::class]):** DataModule ora ha visibilità sulle iniezioni di DBModule, mantenendo comunque il suo ciclo legato al ViewModel
- **@Named("provideDB"):** richiama direttamente l'iniezione



```

Repository
├── Manolo D'Antonio
    └── MoviesRepositoryImpl.kt
        class MoviesRepositoryImpl(
            private val movieRemoteDataSource: MovieNetworkAPI,
            private val movieDB: MovieDAO
        ) : MoviesRepository {
            override suspend fun getMoviesResponse() = withContext(Dispatchers.IO) { this: CoroutineScope ->
                val result = movieRemoteDataSource.getMoviesNowPlaying()
                movieDB.insertMovies(*result.movies.toTypedArray())
                result ^withContext
            }
        }

```

- `insertMovie` ha come input un `vararg`: uno o più elementi separati da una virgola. Per passare gli elementi della lista, la convertiamo in array. Come parametro di `insertMovies` non inviamo l'array, ma i singoli elementi contenuti nell'array, grazie al simbolo `*`. [Documentazione](#)

Dopo aver recuperato la lista di movies dal servizio, facciamo in modo di salvarla nel db prima di ritornarla al `viewModel`.

Questo apre la strada ad una serie di strategie per far sì che l'utente possa utilizzare l'app anche senza connessione a internet.

- `getMoviesResponse` potrebbe leggere i dati di una nuova `val moviesCache: List<Movie>`, che contiene i dati dell'ultima lista inviata al `viewModel` durante questa sessione. Se `moviesCache` è vuota, prova prima a scaricare i movies dal servizio, ma se non ce la fa, li carica dal DB.
- Un'evoluzione potrebbe essere di scaricare i movies dal servizio solo una volta al giorno, a determinati orari: è infatti improbabile che la lista di "nuove uscite" cambi più di una o due volte a settimana.

3. Esercizio

Aggiungi `moviesCache` a `MoviesRepositoryImpl` e fa sì che l'app si comporti come nell'esempio a)

- Basta disattivare il wifi e dati mobili (o modalità aereo) per testare una situazione senza internet
- Intercetta gli errori tramite `try\catch`
- Opzionale: controlla che la rete sia disponibile prima di effettuare la chiamata alla data source.

