# B.2 – ViewBinding e Azioni

## 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 widgetche 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**

Graphical user interface, text, application

Description automatically generated

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:

Graphical user interface, text, application

Description automatically generated

## Activity e Context

L’Activity è la classe del framework Android che ci permette di interagire con il layout creato.   
Insieme e **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**.

## 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)*](https://developer.android.com/reference/android/app/Activity.html#findViewById(int))
* 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.

Graphical user interface, text, application

Description automatically generated

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

Text

Description automatically generated

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.

## 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 **mainActionBtn**

Graphical user interface, text, application, website

Description automatically generated

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

binding.mainActionBtn.*text* = "Activate"

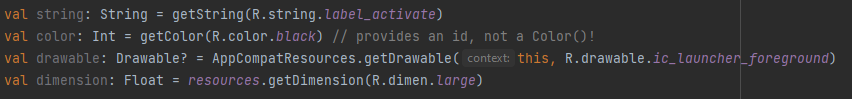
Lanciando l’app ora vedremo che il testo del pulsante è modificato.

## Acquisizione risorse

L’IDE ci informa che non dovremmo hardcordare 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:  


## 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

Graphical user interface, application

Description automatically generated

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:

Graphical user interface, text

Description automatically generated

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:

Text

Description automatically generated

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*

Graphical user interface, text

Description automatically generated

Questa forma abbreviata è lo standard.

## 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:

Graphical user interface, text

Description automatically generated

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*

*Text

Description automatically generated*

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:

Text

Description automatically generated

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

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

## 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:

Graphical user interface

Description automatically generated

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

A screenshot of a computer

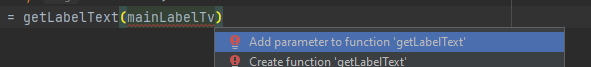
Description automatically generated

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



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

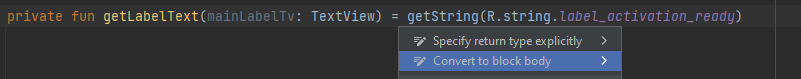
inseriamo mainLabelTv tra le parentesi della funzione e utilizziamo l’assistente dell’IDE per aggiungerlo come parametro ad essa:





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:

Text

Description automatically generated

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:

Graphical user interface, text

Description automatically generated Text

Description automatically generated

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

Text

Description automatically generated

## 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