Views

Android Views è il paradigma standard per la creazione e manipolazione di tutti gli elementi grafici che compongano la *User Interface*. Una **View** definisce un concetto primitivo ma essenziale per lo sviluppo di applicazioni, contenitrice di tutti gli oggetti visualizzati a schermo. Pertanto una *View* è responsabile non solo della modellazione di tutti gli elementi grafici che caratterizzano l'applicazione di riferimento, ma anche della gestione degli eventi che li riguardano, come l'interazione con agenti esterni oppure l'acquisizione di dati.

Esempi più comuni di *View* sono distinti in:

- **Widgets**, descritte come delle etichette, da non confondere con il sistema software già in uso all'interno di dispositivi mobili
- **Layouts**, insieme di caratteristiche grafiche e comportamentali

Per determinare una *View* si adottano due metodi essenziali:

- **Declarative method**, metodo dichiarativo di una *View* tramite il file *XML*, simile a quanto definito a livello di *Activity*, in cui il *Manifest* assume un'importanza essenziale. Il loro accesso tramite linguaggi di programmazione, *Java* oppure *Kotlin*, avviene tramite il metodo **findViewById()** a cui è passato come parametro l'elemento grafico contenuto all'interno della medesima *View*

```
<TextView
  android:id = "@+id/myTextView"
  android:layout_width = "match_parent"
  android:layout_height = "wrap_content"
  android:text = "Hello World"
  android:textAlignment = "center"
  title=Dichiarazione all'interno del file XML
/>
```

```
    public TextView textView;
    textView = (TextView)findViewById(R.id.myTextView);
```

Dichiarazione all'interno del file XML

- **Programmatic View**, in questo ambito le *View* sono direttamente create in *Java* o *Kotlin*, immettendo il *contesto* di riferimento. Tuttavia si tratta di un approccio non raccomandato, a causa del fatto che nella sezione *code* occorre dettare tutte le proprietà visive

Handling Events

La fase successiva ritrae la gestione degli eventi associati agli oggetti visivi. *Java*/*Kotlin* manipolano i possibili eventi attraverso la keyword **OnClick**; mediante la stessa definizione è possibile addirittura forzare l'avvenimento di un evento senza alcuna interazione.

Esistono tre differenti metodi per gestire gli eventi, suddivisi in:

- **XML**, il tutto gestito mediante le **callbacks**, direttamente indicate nel file; tuttavia il metodo descritto riguarda solamente un numero ristretto di componenti.
  METTERE ESEMPIO SLIDE 10
- **Event Handlers**, in questa sezione ogni *View* contiene un ammontare di metodi, richiamati qualora dovesse verificarsi un evento. Le differenti funzioni citate sono frutto di innumerevoli *extend* adattati nella classe *View*, riferiti agli oggetti grafici, per cui tramite questo approccio occorre compiere molteplici *override*; di conseguenza più articolata risulterà la *View*, maggiore sarà l'*impraticabilità*
- **Event Listeners**, in quest'ultima casistica, ogni *View* delega l'implementazione del comportamento, successivo ad un certo evento, ad un oggetto. In tal senso, ogni *listener* gestisce una singola tipologia di evento e contiene un unico metodo *callback*. Si evince in questa breve descrizione la volontà di dividere nettamente ciò che è ritenuto dinamico, come l'interfaccia grafica, da elementi che costituiscono la logica dell'applicazione, ossia il comportamento successivo alla veridicità dell'evento, sovrapponendo un layer di astrazione in grado di isolare le due entità da modifiche e cambiamenti reciproci.

```
Button button;
class MainActivity extends AppCompatActivity implements OnClickListener{
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    ...
    button = findViewById(R.id.button1);
    button.setOnClickListener(this);
  }

  @Override
  void onClick(View v) { }
}
```

Gestione evento tramite Event Listener

Di seguito sono definiti brevemente gli aspetti salienti della sezione di codice riportata:

- In questo esempio la *MainActivity* implementa la classe astratta *OnClickListener*, affinchè possa gestire correttamente gli eventi scaturiti da interazioni con agenti esterni
- All'interno del metodo *onCreate()*, viene acquisito il bottone immesso all'interno della *View*. In questo modo è possibile, attraverso il *listener*, definire il comportamento del bottone una volta creato; si ricorda che la gestione e il coordinamento degli eventi avviene tramite la keyword *OnClick*
- Infine all'interno del metodo *onClick()*, da cui si osserva la keyword *Override*, è sviluppata la vera logica del bottone, rispettando in questo modo la suddivisione in business logic e classi di alto livello descritta precedentemente. Nel caso in cui dovessero essere presenti più bottoni occorrerà implementare uno *switch* che possa distinguere le possibili casistiche

Layout

Un **Layout** deve estendere una **ViewGroup**. Una *ViewGroup* è un contenitore di *View* adottato per definire la struttura del loro *Layout*.

Ogni *View* in un *Layout* deve specificare:

- Una *lunghezza*, espressa in **android:layout_height**
- Una *larghezza*, espressa in **android:layout_width**
- Una *dimensione*, espressa numericamente oppure staticamente tramite la sintassi *match_parent* oppure *wrap_content*, dove la prima definisce una grandezza pari a tutto lo spazio disponibile, contrariamente la seconda racchiude la sua dimensione rispetto al contenitore in cui è posto l'oggetto

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
<TextView
  android:id="@+id/textView"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:text="TextView"
  android:textAlignment="center" />
<Button
  android:id="@+id/button1"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:text="First Button" />
<Button
  android:id="@+id/button2"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:text="Second Button" />
</LinearLayout>
```

Definizione del layout di una view

Come da codice per ogni elemento grafico inserito all'interno della *ViewGroup* è specificata sia una lunghezza che una dimensione. Tuttavia, la sola creazione e modellazione non basta, necessita che il *Layout* ideato sia, dopo essere stato compilato e divenuto una risorsa di tipologia *View*, caricato dall'*Activity*, permettendone l'uso.
Per riuscire nell'intento occorre che all'interno del metodo *onCreate()* sia disposto il *Layout* realizzato, mediante il comando **setContentView()**.

```
Button button;
class MainActivity extends AppCompatActivity implements OnClickListener{
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    ...
    setContentView(R.layout.activity_main);
    button = findViewById(R.id.button1);
```

```
    button.setOnClickListener(this);
  }

  @Override
  void onClick(View v) { }
}
```

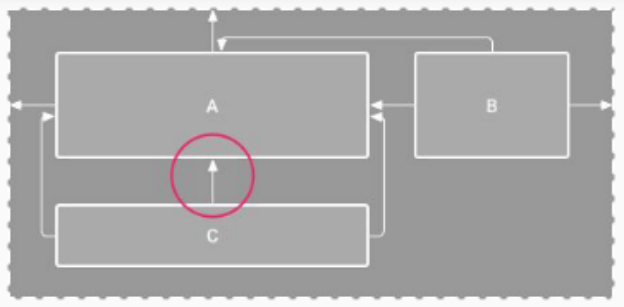
Caricamento del layout all'interno della activity

Gli attributi del *Layout* tipicamente promuovono una denominazione simile a **layout_something**, altrimenti in assenza della keyword **layout_** le proprietà fanno riferimento direttamente alla *View*. Di seguito è promosso un elenco dei tag più utilizzati con una breve descrizione, tra cui:

- **android:layout_margin**, definisce la distanza dell'elemento grafico rispetto ai potenziali componenti che possano circondarlo
- **android:layout_orientation**, dispone la *View* in una singola riga oppure in una sola colonna
- **android:padding**, definisce la distanza del contenuto dell'oggetto grafico dai propri margin
- **android:layout_weight**, parametro necessario per stabilire l'importanza dell'elemento grafico all'interno della *View*

ConstraintLayout

In relazione alle *View* spesso è nominato il concetto di **constraints**. Definisce un *vincolo* visivo, associando l'elemento grafico ad ulteriori oggetti, ad esempio ulteriori *View*.



Come da raffigurazione è evidenziata la presenza di un vincolo tra l'entita A e l'entità C; è denominato **punto di ancoraggio**, in questo caso rappresentativo di un *constraint* di tipologia *top*.

ConstraintLayout sono caratterizzati da una complessa e verbosa sintassi; proprio per questa ragione in Android Studio è predisposto un *editor* in grado di posizionare manualmente vincoli tra *Layout*, *View* oppure, più generalmenete, elementi grafici, e di visualizzare una anticipazione della modellazione in atto.

RecyclerView

Una **RecyclerView** è attuata per visualizzare in maniera efficiente grandi insiemi di dati. Pertanto fornendo i dati di riferimento e definendo la modalità in cui essi debbano apparire,

la *RecyclerView* dinamicamente crea gli elementi grafici e li dispone a video dove vi sia la necessità.

Pertanto, rende facile la visualizzazione di grandi moli di informazioni, da cui si evidenzia la presenza di molteplici layout dedicati ad ogni insieme di membri.

Tutto ciò che è stato descritto può essere attuato attraverso un esempio, in cui le *ListView* acquisiscono la predominanza; meccanismo tramite il quale è possibile aggiungere, rimuovere e aggiornare il contenuto posto al loro interno durante runtime, senza che ogni volta sia completamente ridisegnata la *ViewGroup*. A conoscenza di questo scopo sono implementati alcuni metodi specifici, come:

- **notifyDataSetChanged**(), ...
- **notifyItemInserted**(), ...
- **notifyItemUpdated**(), ...
- **notifyItemDeleted**(), ...

Adeguandosi alle caratteristiche osservate, il caso di studio proposto pone come obiettivo da conseguire la creazione di una *ListView* contenente elementi visivi *TODO*, composti a loro volta da una *TextView* e da una *CheckBox*. I passi necessari per riuscire nell'intento sono cosi suddivisi:

- Definire il layout dell'elemento grafico, in questo caso dell'oggetto *TODO*. In tale ambito una buona scelta implementativa prevede l'utilizzo di una *CardView*, rappresentante un contenitore che mostra sequenzialmente i dati immessi al suo interno

```
<androidx.cardview,.widget.CardView

</androidx.cardview.widget.CardView>
```

Definizione della CardView degli elementi TODO

- Definire la classe degli elementi *TODO*. I *TODOs* sono composti da un titolo testuale, incastonato all'interno di una *TextView*, e da un valore booleano, definito per la *CheckBox*

```
data class Todo {
  var todoTitle: String,
  var done: Boolean = false
}
```

Definizione della classe TODO

- Definire la *ViewHolder*. La *ViewHolder* consiste in un *contaneir* in grado di mantenere tutte le referenze alle proprie *children view*, in modo tale che le stesse possano essere modificate durante *runtime*

```
class TodoViewHolder(itemView: View): ViewHolder(itemView) {
  val tvTodoTitle: TextView = itemView.findViewById(R.id.todoTitle)
```

```
  val cbDone: CheckBox = itemView.findViewById(R.id.todoCheck)
}
```

Definizione della ViewHolder

- Definire un *Adapter*. L'*Adapter* è una parte fondamentale per lo sviluppo e il funzionamento di una *RecyclerView* poichè capace di acquisire in input un insieme di dati, in questo caso di studio si trattano di oggetti *TODO*, e di generare per ogni *Entry* elaborata una *ViewHolder*, compiendo un'essenziale attività di *override*

```
class TodoAdapter (private val todos: MutableList<Todo>):
    Adapter<TodoViewHolder>() {
  override fun onCreateViewHolder(parent: ViewGroup, viewType: int):
      TodoViewHolder (...)
  override fun onBindViewHolder(holder: TodoViewHolder, position: int): {...}
  override fun getItemCount(): int {...}
}
```

Definizione dell'Adapter

- In questa breve sezione si commenta il funzionamento di ogni metodo che contraddistingue l'esempio di codice precedente, in cui:

  - **onCreateViewHolder()**, funzione invocata qualora un nuovo elemento debba essere inserito. Formulata in maniera tale che la *ViewHolder* abbia al suo interno il layout corretto; tuttavia è ancora vuota, ossia devono essere associati i valori correnti all'oggetto TODO mantenuto al suo interno. La fase di *bind* dell'elemento grafico e delle sue caratteristiche avviene solamente qualora dettata una posizone all'interno della *RecyclerView*

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: int):
    TodoViewHolder {
  return TodoViewHolder(
    Layoutinflater.from(parent.context).inflate(R.layout.todo_card,
        parent, false)
  )
}
```

Creazione della ViewHolder

  - **onBindViewHolder()**, metodo attuato nel momento in cui sia data una posizione al nuovo elemento *TODO* all'interno della *RecyclerView*; proprio in questo istante sono popolati i field dell'oggetto

```
override fun onBindViewHolder(holder: TodoViewHolder, position: int) {
  holder.apply {
    tvTodoTitle.text = todos[position].todoTitle
    cbDone.apply{
      isChecked = todos[position].done
      setOncheckedChangeListener(_, b -> todos[position].done = b)
    }
  }
}
```

Popolamento dell'elemento grafico

- **getItemCount()**, *helper function* attuata per restituire il numero di elementi presenti all'interno della struttura dati

```
override fun getItemCoun(): Int {
      return todos.size;
}
```

Return degli elementi della lista di TODO

- Assegnare un *LayoutManager* alla *RecyclerView*. In questo modo è applicato uno "stile" visivo ai differenti elementi grafici

```
val recyclerTodo: RecyclerView = findViewById(R.id.recyclerTodo)
recyclerTodo.adapter = TodoAdapter(mutableListOf())
recyclerTodo.layoutManager = LinearLayoutManager(this)
```

Assegnamento del layout ai nuovi oggetti grafici contenuti nela RecyclerView