

# PROGRAMMATION

# MOBILE

---

## 2 - Activity et Layout

---

NIDHAL JELASSI

[nidhal.jelassi@fsegt.utm.tn](mailto:nidhal.jelassi@fsegt.utm.tn)

# Cycle de vie d'une application

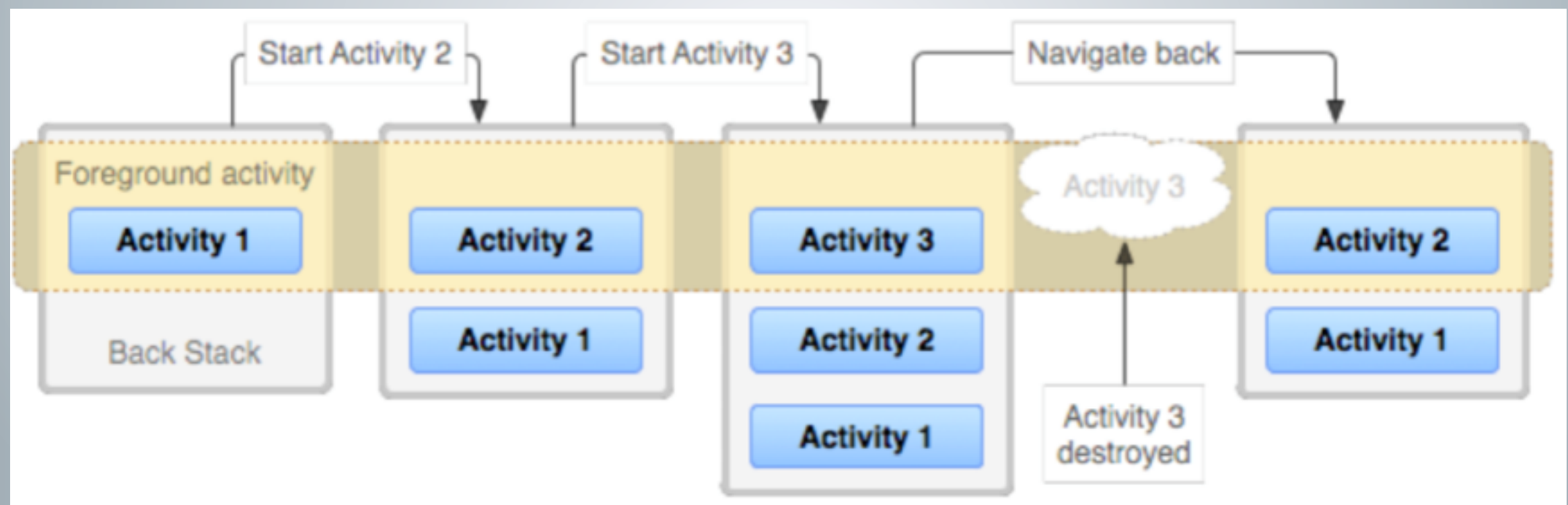
---

- Les composants (vue) d'une application ont un cycle de vie :
  - Un début : Lors de l'instanciation en réponse aux Intents.
  - Une fin : Lors de la destruction des instances.
- Entre le début et la fin, les composants passent par les états suivants :
  - Actifs ou inactifs
  - Visibles ou invisibles

# Cycle de vie d'une application

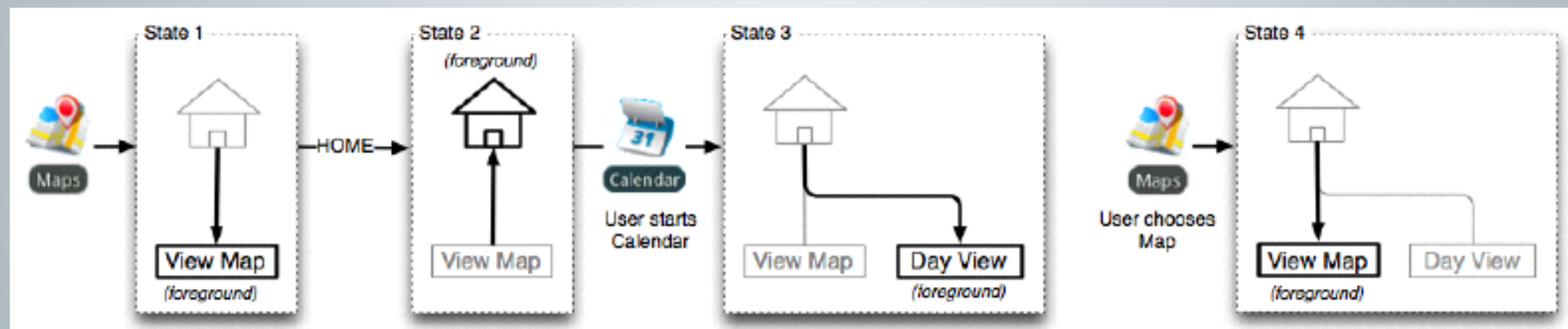
---

- Les activités d'une application sont gérées sous forme de Pile.



# Appels aux activités

---





# Cycle de vie d'une application

---

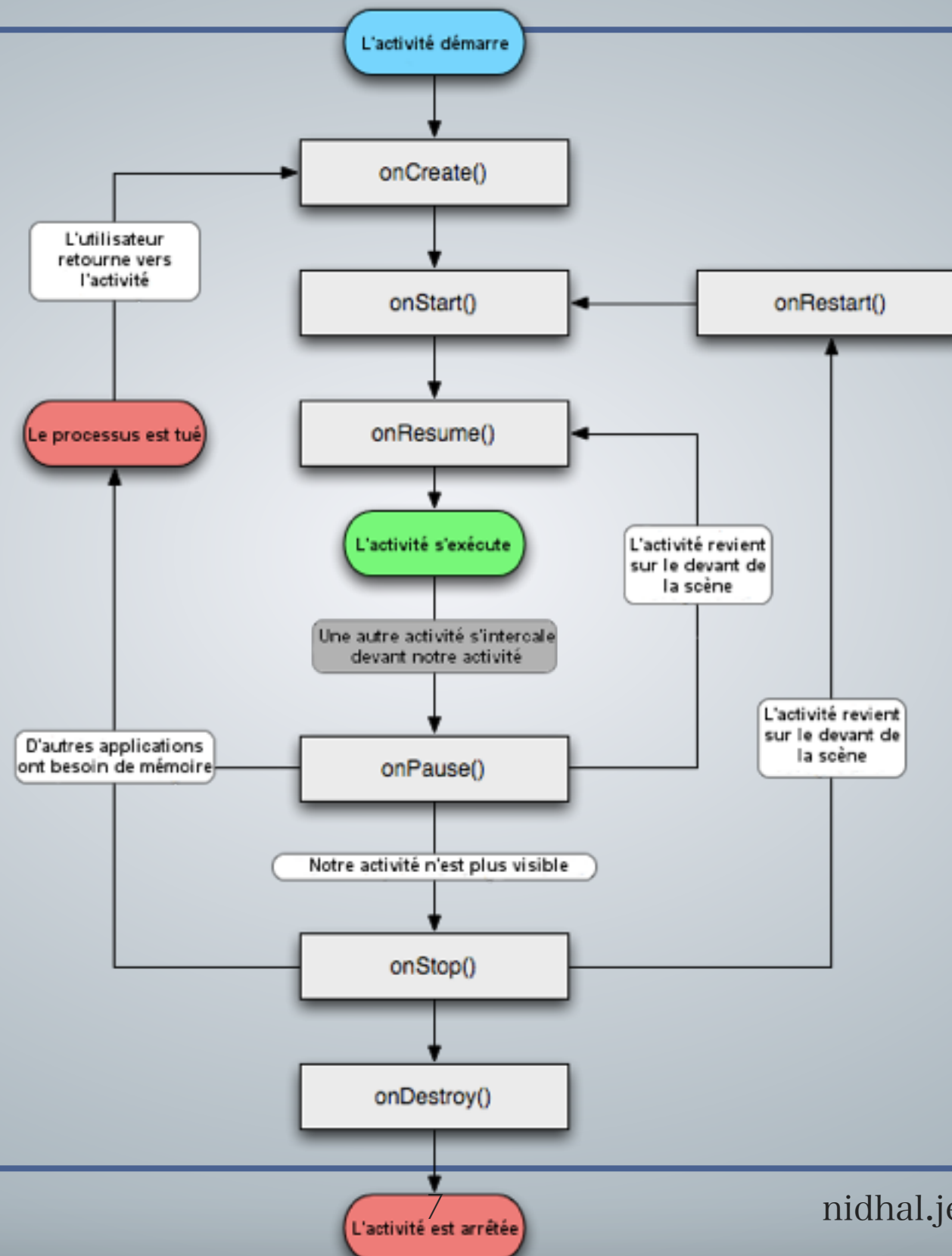
- Au lancement d'une nouvelle activité, elle est placée à la tête de la pile. Elle est alors l'activité en exécution.
- L'activité précédente ne revient en tête de la pile que si la nouvelle activité est fermée.
- Sur le terminal, en cliquant sur le bouton Retour, l'activité suivante dans la pile devient active.

# Cycle de vie d'une application

---

- Il n'existe pas de méthode **main** dans un programme Android
- Android exécute le code d'une activité en appelant des **callbacks** qui correspondent aux phases de la vie d'une activité
- Il n'est pas nécessaire d'implémenter toutes les callbacks.

# Etats d'une activité



# Evénements liés

- Au passage d'une activité d'un état à un autre, le framework Android appelle les méthodes de transition correspondantes :

```
public class Main extends Activity {  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.acceuil); }  
    protected void onDestroy() {  
        super.onDestroy(); }  
    protected void onPause() {  
        super.onPause(); }  
    protected void onResume() {  
        super.onResume(); }  
    protected void onStart() {  
        super.onStart(); }  
    protected void onStop() {  
        super.onStop(); } }  
}
```

Callbacks

Obligatoire

Recommandé



# Callbacks

---

**VOID ONCREATE(BUNDLE SAVEDINSTANCESTATE)**

- Invoquée à la création d'une activité
- Initialisation de tous les éléments
- Le Bundle `savedInstanceState` contient l'état précédent de l'activité.
- Est toujours suivie de la méthode `onStart`.

# Callbacks

---

## VOID ONSTART()

- Invoquée juste avant que l'activité ne devienne visible.
- Est toujours suivie de la méthode :
  - **onResume** si l'activité revient active.
  - **onStop** si l'activité est cachée.

## VOID ONRESTART()

- Invoquée quand l'activité est redémarré
- Est toujours suivie de la méthode onStart

# Callbacks

---

## VOID ONPAUSE()

- Invoquée quand le système va démarrer une autre activité.
- Arrête tout ce qui consomme de la mémoire (ex: animation, connexion DB).
- N'empêche pas le système de tuer l'activité.
- Est suivie des méthodes :
  - **onResume**
  - **onStop**

# Callbacks

---

## VOID ONSTOP()

- Invoquée quand l'activité n'est plus visible. Elle est alors totalement cachée et ne peut plus exécutée de code
- N'empêche pas le système de tuer l'activité
- Peut être suivie de l'une des méthodes :
  - **onRestart**
  - **onDestroy**



# Callbacks

---

## VOID ONDESTROY()

- Invoquée quand l'activité est détruite.
- Deux scénarios possibles :
  - l'activité est en cours de finition.
  - le système détruit temporairement cette instance de l'activité pour économiser de l'espace

# Une Activité

---

- Est un composant d'application.
  - Fournit un écran avec lequel les utilisateurs peuvent interagir avec l'application et ses différentes fonctionnalités.
- Chaque **Activity** est associée à une fenêtre qui représente l'interface utilisateur.
- Une application = Enchaînement des activités

# Une Activité

---

- Pour pouvoir être lancée, toute activité doit être préalablement déclaré dans le **AndroidManifest**
- Une activité est désigné comme activité initiale de l'application (dans **AndroidManifest**)
- Lancer une activité simple : Méthode **startActivity**
- Lancer une activité en vue d'obtenir un résultat en retour (on parle alors de "sous-activite")
  - Methode **startActivityForResult(...)**

# Une Activité

- Une classe qui hérite de la classe mère Activity

```
public class MainActivity extends AppCompatActivity {

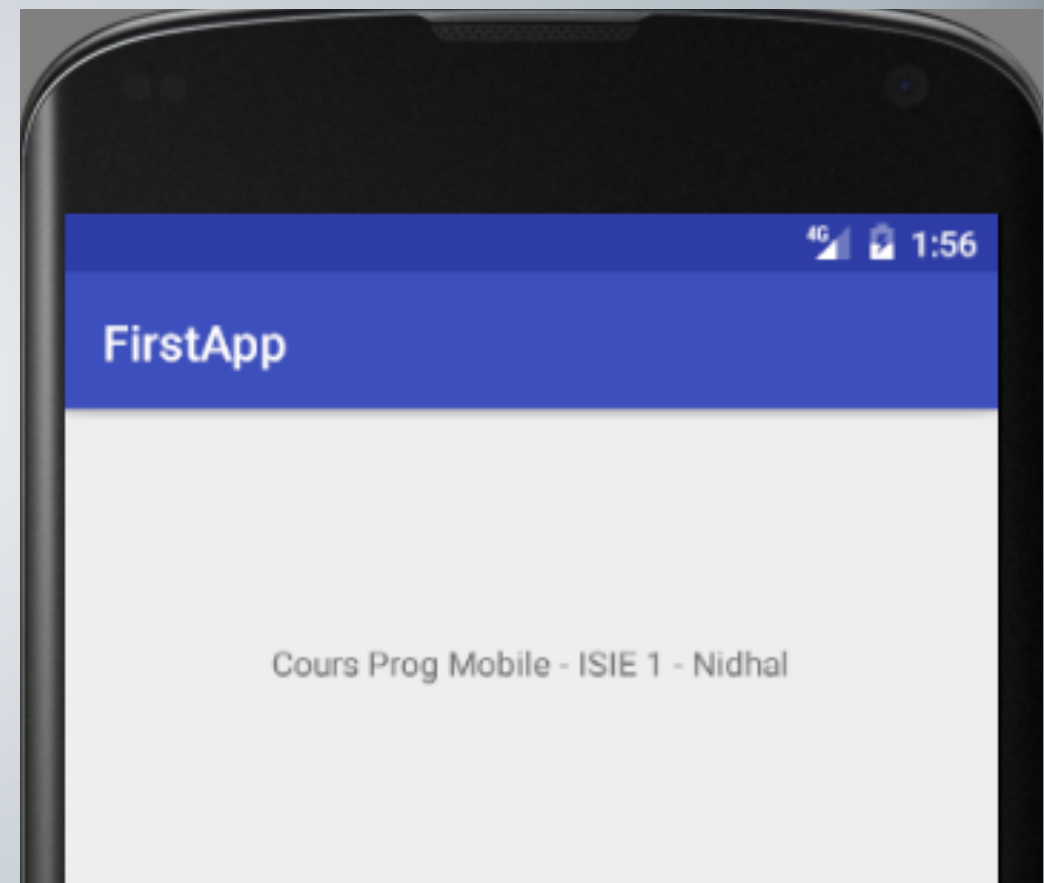
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        TextView tx = (TextView) this.findViewById(R.id.txtv);
        tx.setText("Cours Prog Mobile - ISIE 1 - Nidhal");
    }
}

<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.nidhal.firstapp">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="FirstApp"
        android:supportRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

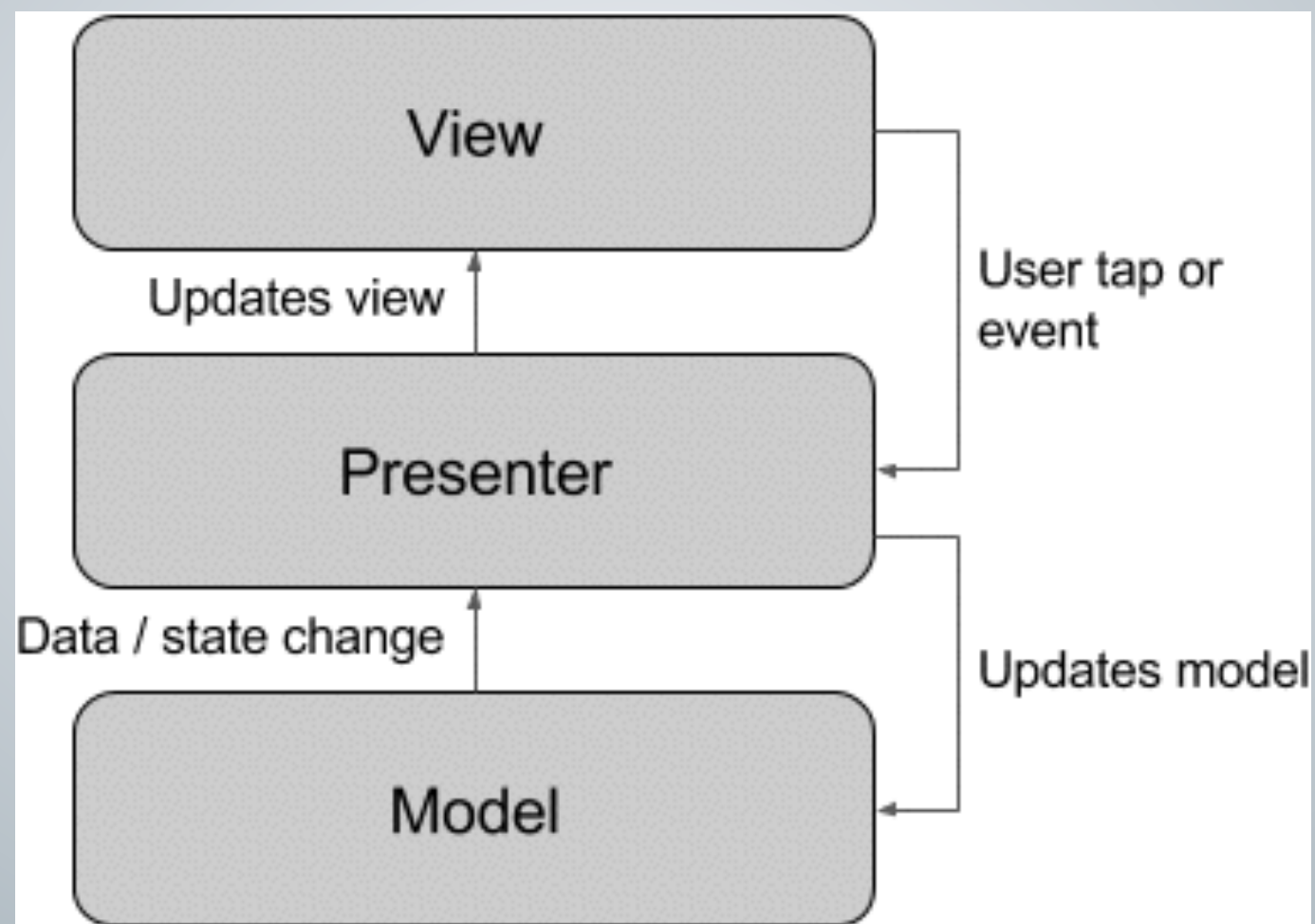




# Le modèle MVP

---

- La relation entre l'activité et son layout se fait selon le modèle MVP (Model-View-Presenter).



# Le modèle MVP

---

- Les **Views** sont des éléments d'interface utilisateur qui affichent des données et répondent aux actions de l'utilisateur. Chaque élément de l'écran est une vue.
- Les **Presenters** connectent les vues de l'application au modèle. Ils fournissent les vues avec les données spécifiées par le modèle, ainsi que le modèle avec les entrées utilisateur à partir de la vue.
- Le **Model** spécifie la structure des données de l'application et le code permettant d'accéder aux données et de les manipuler.

# View et ViewGroup

---

- Une vue est une classe qui étend **View**.
- Un groupe de vue étend **ViewGroup** (et donc **View**) et peut contenir d'autres vues.
- Un groupe de vues organise l'affichage des vues qu'il contient
- La méthode **setContentView** (de **Activity**) sert à préciser la vue à afficher.

# Layouts

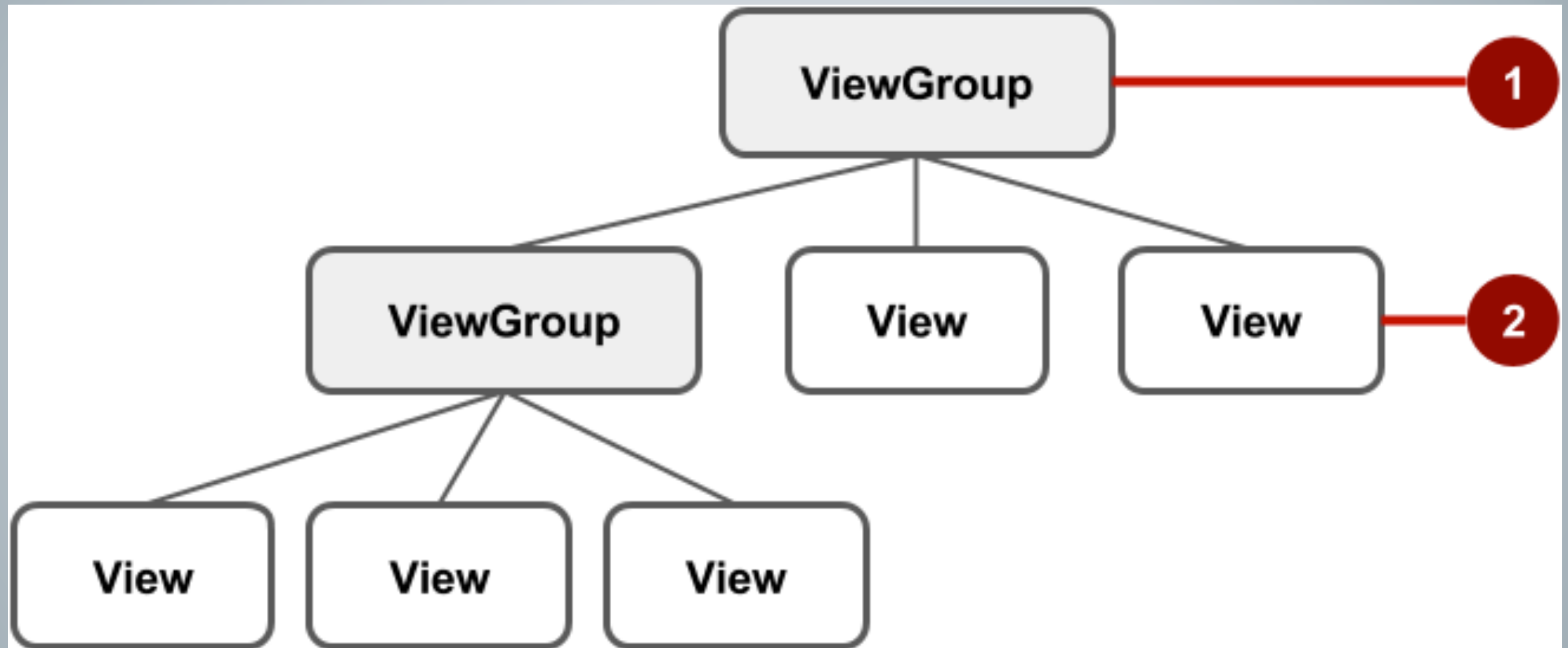
---

- Vues (LinearLayout, RelativeLayout, etc.) héritant de ViewGroup qui gère le placement de vues filles à l'intérieur du ViewGroup
- Chaque layout permet d'associer à une **View** (fille du Layout) un ensemble de contraintes de placement.
- Arbre de vues statique doit être définie en XML dans le répertoire ressource layout



# Layouts

---



# Types de Layouts

---

- **LinearLayout:** dispose les éléments de gauche à droite ou du haut vers le bas
- **RelativeLayout:** les éléments sont placés relativement les uns par rapport aux autres
- **TableLayout:** disposition matricielle
- **FrameLayout:** disposition en haut à gauche en empilant les éléments (un seul visible à la fois). . Les vues sont stockées dans une pile, La taille de FrameLayout est la taille de sa plus grande vue enfant.

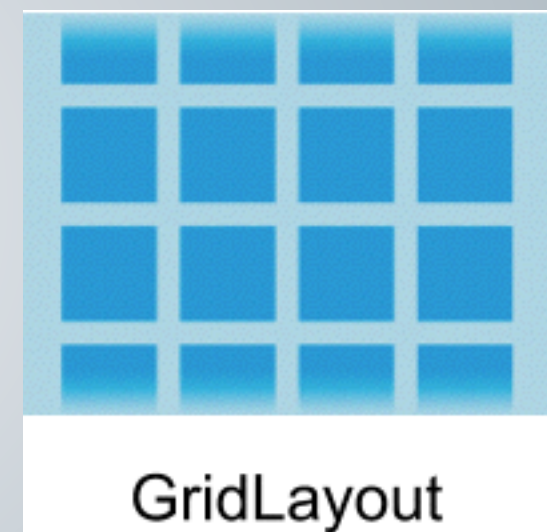
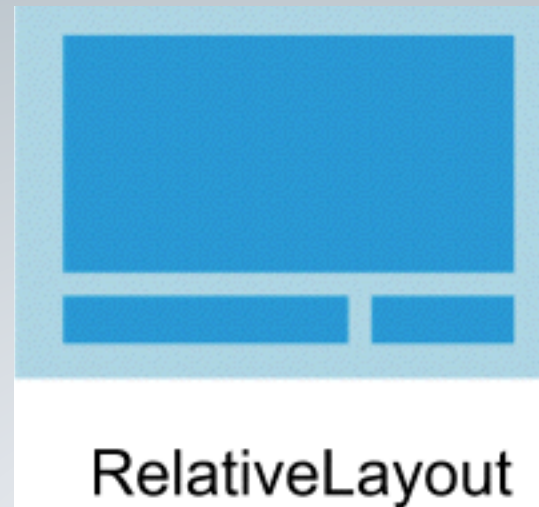
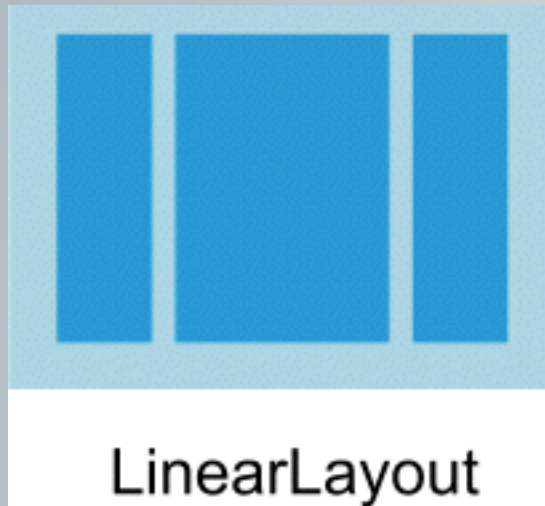
# Types de Layouts

---

- **GridLayout:** disposition des composants sur une grille.
- **ConstraintLayout:** groupe de vues utilisant des points d'ancrage, des arêtes et des instructions pour contrôler le positionnement des vues par rapport aux autres éléments de la présentation.
- **AbsoluteLayout:** un groupe qui vous permet de spécifier les emplacements exacts (coordonnées x / y) de ses vues enfants. Le moins souple.
- Les déclarations se font principalement en XML, ce qui évite de passer par les instanciations Java.

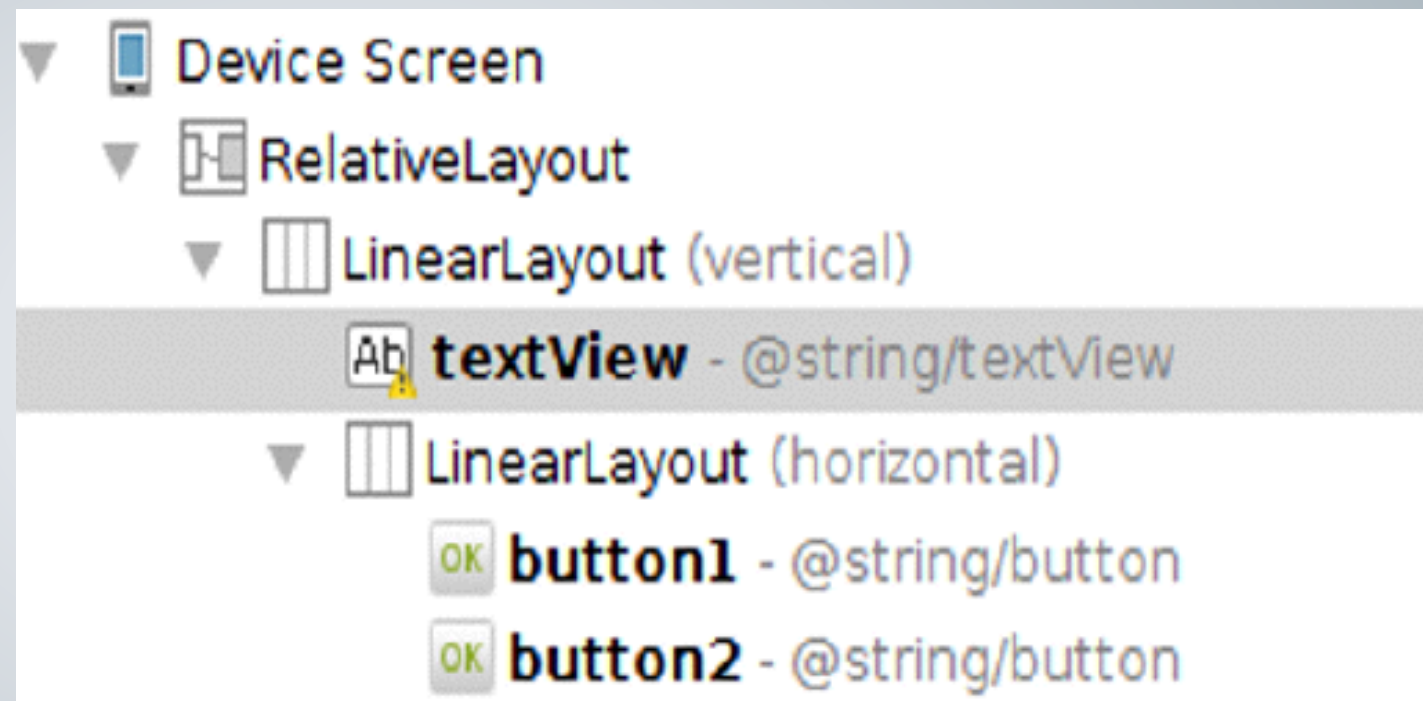
# Examples

---





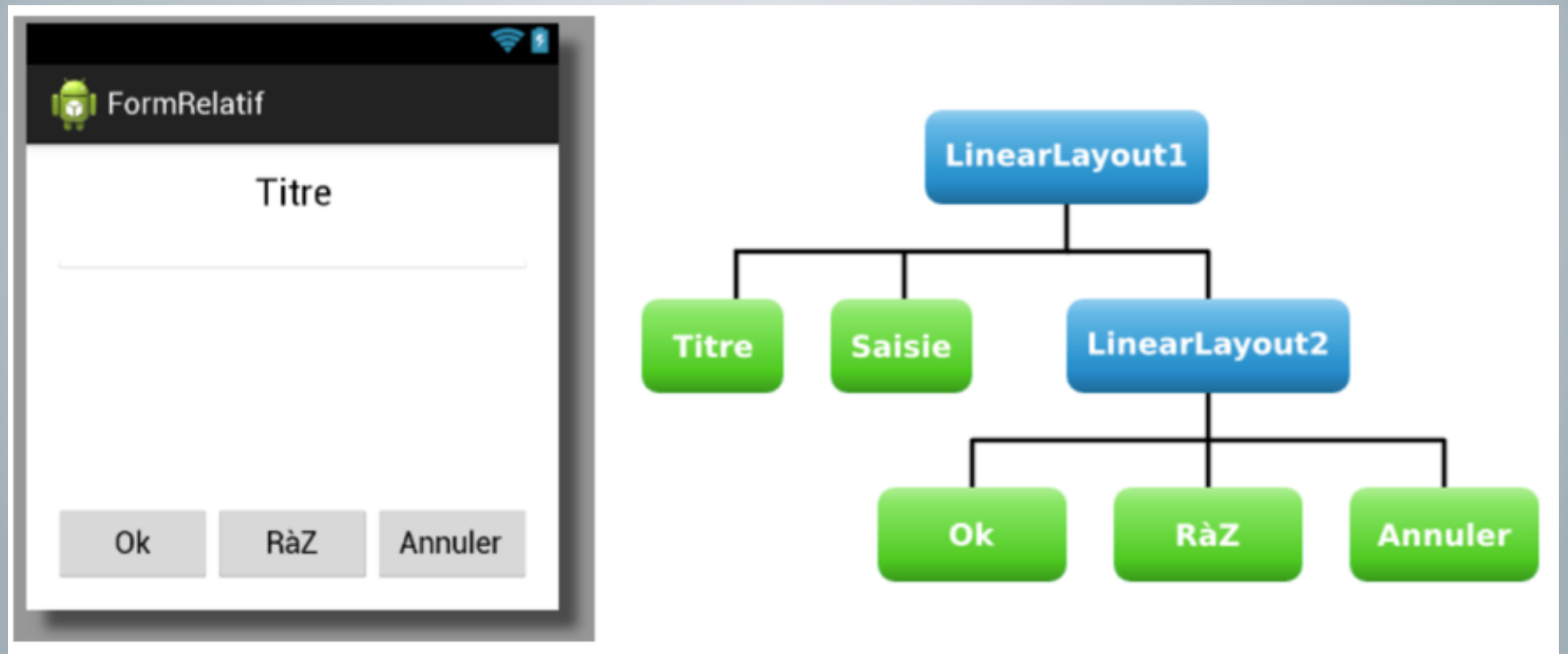
# Hierarchie des vues



# Hierarchie des vues

---

Les groupes et vues forment un arbre



# Layout

---

- En XML, cet arbre donne ça :

```
<LinearLayout android:id="@+id/groupe1" ...>
  <TextView android:id="@+id/titre" .../>
  <EditText android:id="@+id/saisie" .../>
  <LinearLayout android:id="@+id/groupe2" ...>
    <Button android:id="@+id/ok" .../>
    <Button android:id="@+id/raz" .../>
    <Button android:id="@+id/annuler" .../>
  </LinearLayout>
</LinearLayout>
```



# Layout

---

- La plupart des groupes utilisent des paramètres de placement sous forme d'attributs XML. Par exemple, telle vue à droite de telle autre, telle vue la plus grande possible, telle autre la plus petite.
- Ces paramètres sont de deux sortes :
  - ceux qui sont demandés pour toutes les vues, comme `android:layout_width`, `android:layout_height` et `android:layout_weight`
  - ceux qui sont demandés par le groupe englobant et qui en sont spécifiques, comme `android:layout_alignParentBottom`, `android:layout_centerInParent`. . .



# Layout

---

- Toutes les vues doivent spécifier ces deux attributs :
  - `android:layout_width` largeur de la vue
  - `android:layout_height` hauteur de la vue
- Ils peuvent valoir :
  - `"wrap_content"` : la vue est la plus petite possible
  - `« match_parent` (anciennement `fill_parent`) : la vue est la plus grande possible
  - `"valeurdp"` : une taille fixe, ex : `"100dp"`

# LinearLayout

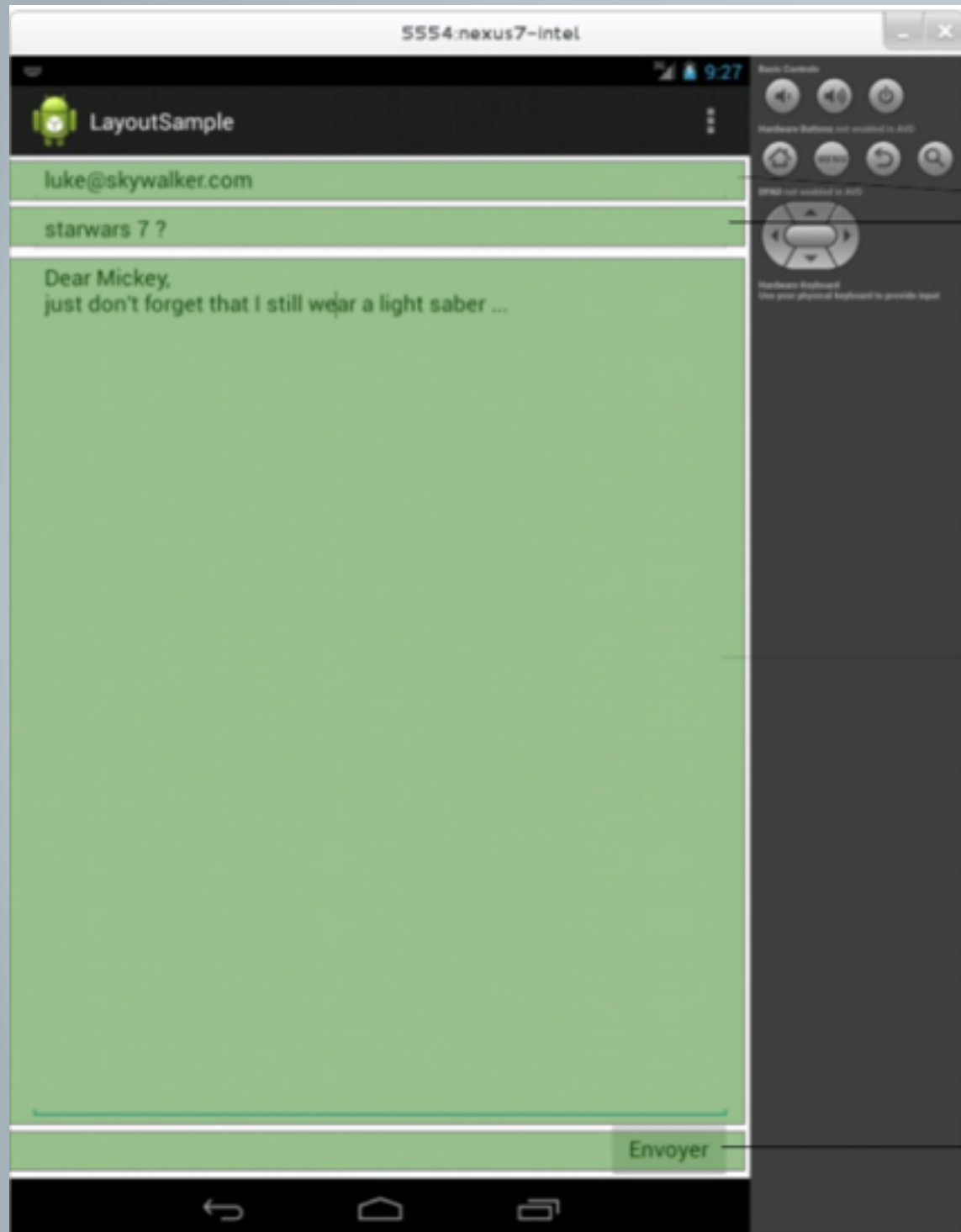
---

- Il range ses vues soit horizontalement, soit verticalement



```
<LinearLayout android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
    <Button android:text="Ok" android:layout_weight="1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <Button android:text="Annuler" android:layout_weight="1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
</LinearLayout>
```

# LinearLayout



5554.nexus7-intel

LayoutSample

luke@skywalker.com

starwars 7 ?

Dear Mickey,  
just don't forget that I still wear a light saber ...

Envoyer

layout\_width = fill\_parent  
layout\_height = wrap\_content

layout\_width = fill\_parent  
layout\_height = 0dp  
layout\_weight = 1  
layout\_gravity = top

layout\_width = 100dp  
layout\_height = wrap\_content  
layout\_gravity = right



# LinearLayout

---

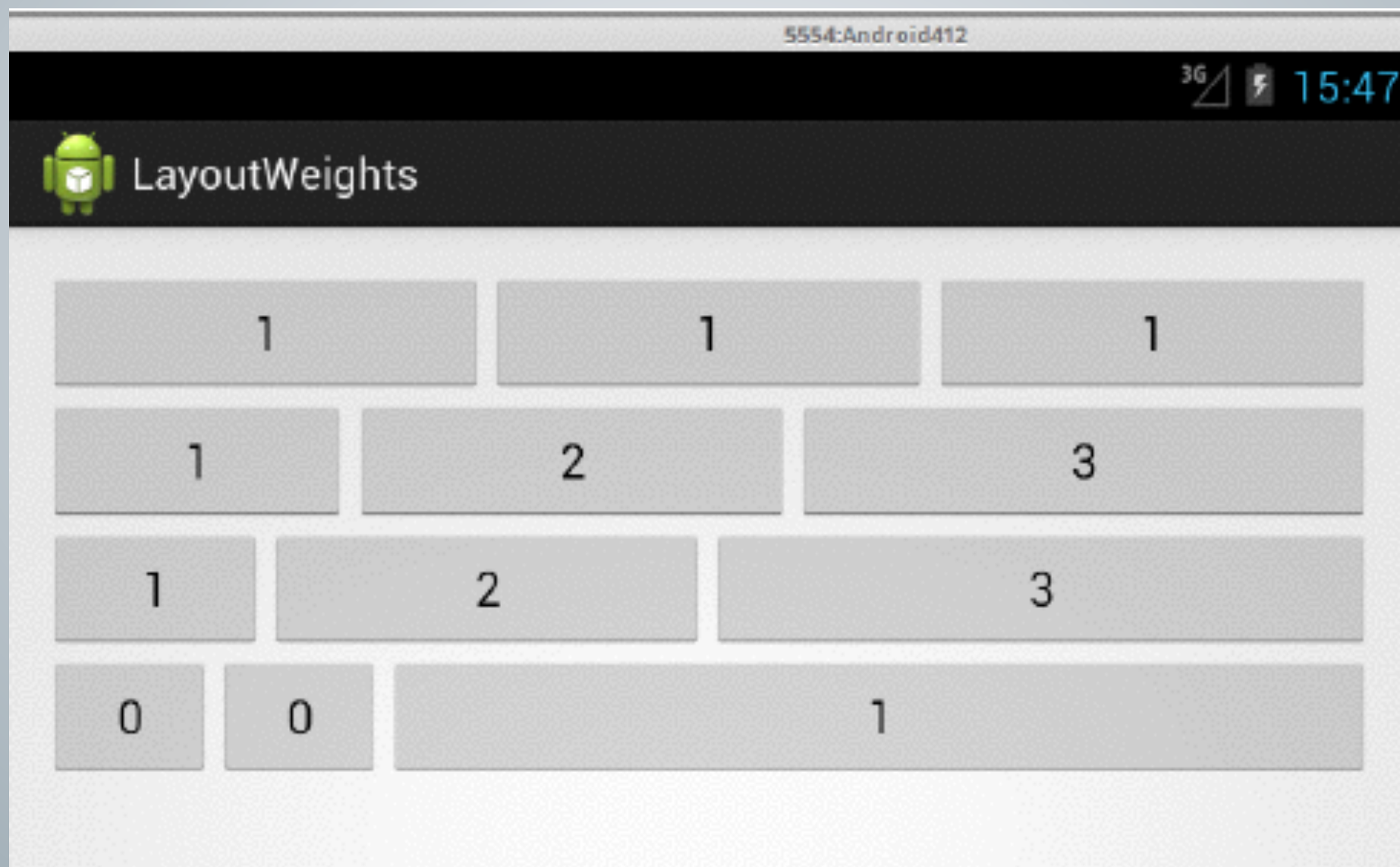
- Une façon intéressante de spécifier les tailles des vues dans un LinearLayout consiste à leur affecter un poids avec l'attribut `android:layout_weight`.
- Un `layout_weight` égal à 0 rend la vue la plus petite possible.
- Un `layout_weight` non nul donne une taille correspondant au rapport entre ce poids et la somme des poids des autres vues.



# LinearLayout

---

Voici 4 LinearLayout horizontaux de 3 boutons ayant des poids égaux à leurs titres.



# FrameLayout

---

- Affichage d'une pile de vues avec gestion basique du positionnement
  - Paramètre de positionnement  
`FrameLayout.LayoutParams` (width, height, gravity)
  - Gravity définie l'emplacement de la vue enfant (top, bottom, left, right, fill...)

# RelativeLayout

---

- **width/height:** contrôle l'occupation de la case
- **below, above:** placement relatif à un autre composant (en dessous/au dessus)
- **alignLeft, alignRight:** alignement relatif entre composants
- **marginTop, marginLeft, marginBottom, marginRight:** marges autour du composant

# TableLayout

---

- Positionnement des vues en ligne de TableRow
  - (similaire au `<table>` `<tr>` `<td>` de HTML)
- TableRow hérite de LinearLayout avec alignement automatique des colonnes sur chaque ligne
- Propriétés de **TableRow.LayoutParams**
  - `layout_column`: indice de départ de la colonne
  - `layout_span`: nombre de colonnes occupées



# GridLayout

---

- Positionnement sur une grille rectangulaire de N colonnes
- Contrairement au TableLayout les vues sont ajoutés directement avec leur paramètres de positionnement
- Propriétés de GridLayout.LayoutParams
  - `layout_column/layout_columnSpan`: colonne de départ, nombre de colonnes occupées
  - `layout_gravity`: emplacement de la vue enfant dans sa zone