

Chapitre 2

Création d'interfaces utilisateur

Ce chapitre explique la création d'interfaces utilisateur :

- Activités
- Relations entre un source Java et des ressources
- Layouts et vues

On ne s'intéresse qu'à la mise en page. L'activité des interfaces sera étudiée dans le chapitre prochain

NB: les textes [fuchsia](#) sont des liens cliquables vers des compléments d'information.

On va commencer par une présentation très rapide des concepts, puis revenir en détails.

2.1. Présentation rapide des concepts

2.1.1. Composition d'une application

L'interface utilisateur d'une application Android est composée d'écrans. Un «écran» correspond à une *activité*, ex :

- afficher des informations
- éditer des informations

Les dialogues et les *pop-up* ne sont pas des activités, ils se superposent temporairement à l'écran d'une activité.

Android permet de naviguer d'une activité à l'autre, ex :

- une action de l'utilisateur, bouton, menu ou l'application fait aller sur l'écran suivant
- le bouton back ramène sur l'écran précédent.

2.1.2. Structure d'une interface utilisateur

L'interface d'une activité est composée de *vues* :

- vues élémentaires : boutons, zones de texte, cases à cocher...

- vues de groupement qui permettent l'alignement des autres vues : lignes, tableaux, onglets, panneaux à défilement...

Chaque vue d'une interface est gérée par un objet Java, comme en Java classique, avec AWT, Swing ou JavaFX.

Il y a une hiérarchie de classes dont la racine est [View](#). Elle a une multitude de sous-classes, dont par exemple [TextView](#), elle-même ayant des sous-classes, par exemple [Button](#).

Les propriétés des objets sont généralement visibles à l'écran : titre, taille, position, etc.

2.1.3. Création d'une interface

Ces objets d'interface pourraient être créés manuellement, voir plus loin, mais :

- c'est très complexe, car il y a une multitude de propriétés à définir,
- ça ne permet pas de *localiser*, c'est à dire adapter une application à chaque pays (sens de lecture de droite à gauche)

Alors, on préfère définir l'interface par l'intermédiaire d'un fichier XML qui décrit les vues à créer. Il est lu automatiquement par le système Android lors du lancement de l'activité et transformé en autant d'objets Java qu'il faut.

Chaque objet Java est retrouvé grâce à un *identifiant* appelé «identifiant de ressource».

2.1.4. Création d'un écran

Chaque écran est géré par une instance d'une sous-classe de [Activity](#) que vous programmez. Il faut au moins surcharger la méthode onCreate selon ce qui doit être affiché sur l'écran :

```
public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) { super.onCreate(
        savedInstanceState); setContentView(R.layout.main) ;
    }}
```

C'est l'appel setContentView(...) qui met en place l'interface. Son paramètre **R.layout.main** est l'identifiant d'une disposition de vues d'interface. C'est ce qu'on va étudier maintenant.

2.2. Ressources

2.2.1. Définition

Les ressources sont tout ce qui n'est pas programme (classes, bibliothèques) dans une application. Dans Android, ce sont les textes, messages, icônes, images, sons, interfaces, styles, etc.

C'est une bonne séparation, car cela permet d'adapter une application facilement pour tous les pays, cultures et langues. On n'a pas à bidouiller dans le code source et recompiler chaque fois. C'est le même code compilé, mais avec des ressources spécifiques.

Le programmeur doit simplement prévoir des variantes linguistiques des ressources qu'il souhaite permettre de traduire. Ce sont des sous-dossier, ex: values-fr, values-en, values-jp, etc et il n'y a qu'à modifier des fichiers XML.

2.2.2. Identifiant de ressource

Le problème est alors de faire le lien entre les ressources et les programmes : par un identifiant. Par exemple, la méthode `setContentView` demande l'identifiant de l'interface à afficher dans l'écran : `R.layout.main`.

Cet identifiant est un entier qui est généré automatiquement par le SDK Android. Comme il va y avoir de très nombreux identifiants dans une application :

- chaque vue possède un identifiant (si on veut)
- chaque image, icône possède un identifiant
- chaque texte, message possède un identifiant
- chaque style, thème, etc. etc.

Ils ont tous été regroupés dans une classe spéciale appelée **R**.

2.2.3. Génération de la classe R

Le SDK Android (aapt) construit automatiquement cette classe statique appelée **R**. Elle ne contient que des constantes entières groupées par catégories : id, layout, menu... :

```
public final class R { public static final class string { public static
    final int app_name=0x7f080000; public static final int
    message=0x7f080001;
  }
  public static final class layout { public static final int
    main=0x7f030000;
  }
  public static final class menu { public static final int
    main_menu=0x7f050000; public static final int
    context_menu=0x7f050001;
  } ...
}
```

2.2.4. La classe R

Cette classe R est générée automatiquement (dans le dossier generated) par ce que vous mettez dans le dossier res : interfaces, menus, images, chaînes... Certaines de ces ressources sont des fichiers XML, d'autres sont des images PNG.

Par exemple, le fichier `res/values/strings.xml` :

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">Exemple</string>
    <string name="message">Bonjour !</string>
</resources>
```

Cela rajoute automatiquement deux entiers dans `R.string` : `app_name` et `message`.

2.2.5. Rappel sur la structure d'un fichier XML

Un [fichier XML](#) : éléments (racine et sous-éléments), attributs, texte et namespaces.

```
<?xml version="1.0" encoding="utf-8"?>
<racine xmlns:exemple="http://...">
    <!-- commentaire -->
    <element attribut1="valeur1" attribut2="valeur2">
        <feuille1 exemple:attribut3="valeur3"/>
        <feuille2>texte</feuille2>
    </element> texte
    en vrac
</racine>
```

Rappel : dans la norme XML, le namespace par défaut n'est jamais appliqué aux attributs, donc il faut mettre le préfixe sur ceux qui sont concernés. Voir le cours [XML](#).

2.2.6. Espaces de nommage dans un fichier XML

Dans le cas d'Android, il y a un grand nombre d'éléments et d'attributs normalisés. Pour les distinguer, ils ont été regroupés dans le *namespace* android.

Vous pouvez lire [cette page](#) et [celle-ci](#) sur les *namespaces*.

```
<menu xmlns:android=
    "http://schemas.android.com/apk/res/android">
    <item android:id="@+id/action_settings"
        android:orderInCategory="100"
        android:showAsAction="never"
        android:title="Configuration"/>
</menu>
```

2.2.7. Ressources de type chaînes

Dans `res/values/strings.xml`, on place les chaînes de l'application, au lieu de les mettre en constantes dans le source :

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">HelloWorld</string>
    <string name="main_menu">Menu principal</string>
    <string name="action_settings">Configuration</string>
    <string name="bonjour">Demat !</string>
</resources>
```

Intérêt : pouvoir traduire une application sans la recompiler.

2.2.8. Traduction des chaînes (*localisation*)

Lorsque les textes sont définis dans `res/values/strings.xml`, il suffit de faire des copies du dossier `values`, en `values-us`, `values-fr`, `values-de`, etc. et de traduire les textes en gardant les attributs `name`. Voici par exemple `res/values-de/strings.xml` :

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">HelloWorld</string> <string
name="main_menu">Hauptmenü</string>
    <string name="action_settings">Einstellungen</string>
    <string name="bonjour">Guten Tag</string>
</resources>
```

Le système android ira chercher automatiquement le bon texte en fonction des paramètres linguistiques configurés par l'utilisateur.

2.2.9. Emploi des ressources texte dans un programme

Dans un programme Java, on peut très facilement placer un texte dans une vue de l'interface :

```
TextView tv = ... // ... voir plus loin pour les vues tv.setText(R.string.bonjour) ;
```

`R.string.bonjour` désigne le texte de `<string name="bonjour">...` dans le fichier `res/values*/strings.x`

Cela fonctionne car `TextView.setText()` a deux surcharges :

- void setText(String text) : on peut fournir une chaîne quelconque
- void setText(int idText) : on doit fournir un identifiant de ressource chaîne, donc forcément l'un des textes du fichier res/values/strings.xml

Par contre, si on veut récupérer l'une des chaînes des ressources pour l'utiliser dans le programme, c'est un peu plus compliqué :

```
String message = getResources().getString(R.string.bonjour) ;
```

getResources() est une méthode de la classe Activity (héritée de la classe abstraite Context) qui retourne une représentation de toutes les ressources du dossier res. Chacune de ces ressources, selon son type, peut être récupérée avec son identifiant.

2.2.10. Emploi des ressources texte dans une interface

Maintenant, dans un fichier de ressources décrivant une interface, on peut également employer des ressources texte :

```
< RelativeLayout >
    <TextView android:text="@string/bonjour" />
    <Button android:text="Commencer" />
< /RelativeLayout >
```

- Le titre du TextView sera pris dans le fichier de ressource des chaînes,
- par contre, le titre du Button sera une chaîne fixe *hard coded*, non traduisible, donc Android Studio mettra un avertissement.

@string/nom est une référence à la chaîne du fichier res/values*/strings.xml ayant ce nom.

2.2.11. Images : R.drawable.nom

De la même façon, les images PNG placées dans res/drawable et res/mipmaps-* sont référençables :

```
< ImageView android:src="@drawable/velo"
    android:contentDescription="@string/mon_velo" />
```

La notation @drawable/nom référence l'image portant ce nom dans l'un des dossiers.

NB: les dossiers res/mipmaps-* contiennent la même image à des définitions différentes, pour correspondre à différents téléphones et tablettes. Ex: mipmap-hdpi contient des icônes en 72 x 72 pixels.

2.2.12. Tableau de chaînes : R.array.nom

Voici un extrait du fichier res/values/arrays.xml :

```

< resources >
    <string-array name="planetes">
        <item>Mercure< /item >
        <item>Venus< /item >
        <item>Terre< /item >
        <item>Mars< /item > ...
    < /string-array >
< /resources >

```

Dans le programme Java, il est possible de faire :

```

Resources res = getResources() ;
String[] planetes = res.getStringArray(R.array.planetes) ;

```

2.2.13. Autres

D'autres notations existent :

- @style/nom pour des définitions de res/style
- @menu/nom pour des définitions de res/menu

Certaines notations, @package:type/nom font référence à des données prédéfinies, comme :

- @android:style/TextAppearance.Large
- @android:color/black Il y a aussi une notation en ?type/nom pour référencer la valeur de l'attribut nom, ex : ?android:attr/textColorSecondary.

2.3. Mise en page (*layouts*)

2.3.1. Structure d'une interface Android

Un écran Android de type formulaire est généralement composé de plusieurs vues. Entre autres :

- TextView, ImageView : titre, image
- EditText : texte à saisir
- Button, CheckBox : bouton à cliquer, case à cocher

Ces vues sont alignées à l'aide de groupes sous-classes de ViewGroup, éventuellement imbriqués :

- LinearLayout : positionne ses vues en ligne ou en colonne
- RelativeLayout, ConstraintLayout : positionnent leurs vues l'une par rapport à l'autre
- TableLayout : positionne ses vues sous forme d'un tableau

2.3.2. Arbre des vues

Les groupes et vues forment un [arbre](#) :

figure 16

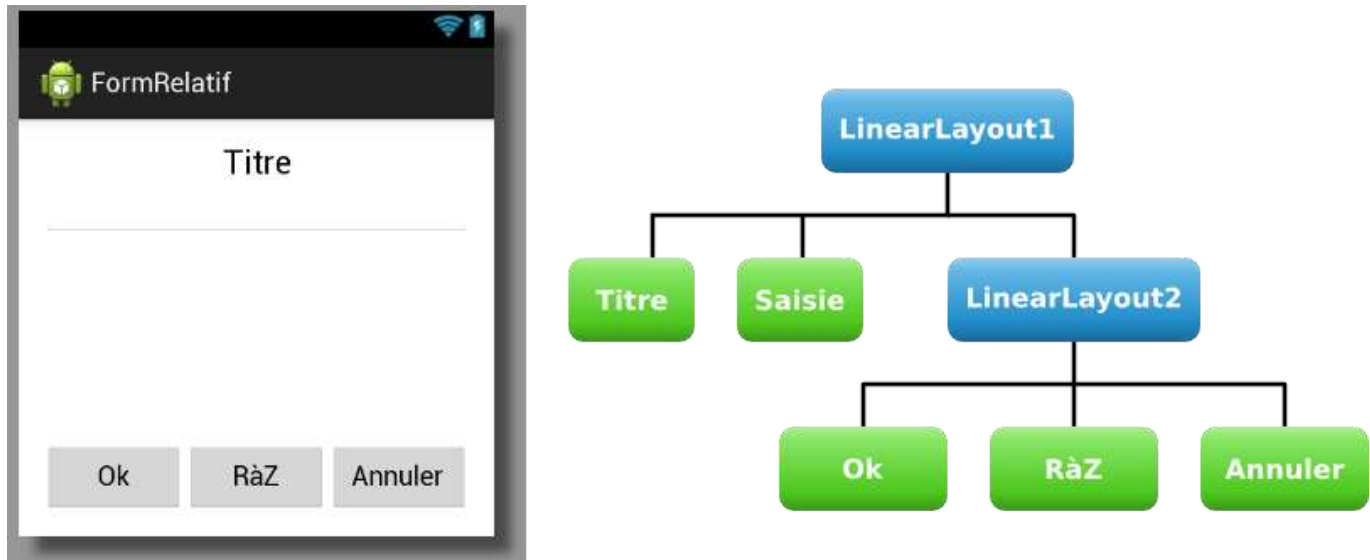


Figure 16: Arbre de vues

2.3.3. Création d'une interface par programme

Il est possible de créer une interface par programme, comme avec JavaFX et Swing, mais c'est assez compliqué :

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate( savedInstanceState); TextView tv = new
    TextView(this) ; tv.setText(R.string.bonjour) ;
    LinearLayout rl = new LinearLayout(this) ;

    LayoutParams lp = new LayoutParams() ;
    lp.width = LayoutParams.MATCH_PARENT;
```



```

        lp.height = LayoutParams.MATCH_PARENT;
        rl.addView(tv, lp); setContentView( rl);
    }

```

2.3.4. Ressources de type *layout*

Il est donc préférable de stocker l'interface dans un fichier `res/layout/main.xml` :

```

<LinearLayout ...>
    <TextView android:text="@string/bonjour" ... />
</LinearLayout >

```

qui est référencé par son identifiant `R.layout.nom_du_fichier` (donc ici c'est `R.layout.main`) dans le programme Java :

```

protected void onCreate(Bundle bundle) { super.onCreate( bundle);
    setContentView(R.layout.main) ;
}

```

La méthode `setContentView` fait afficher le *layout* indiqué.

2.3.5. Identifiants et vues

Lorsque l'application veut manipuler l'une de ses vues, elle doit utiliser `R.id.symbole`,

```

ex : TextView tv = findViewById(R.id.message) ;

```

avec la définition suivante dans `res/layout/main.xml` :

```

<LinearLayout ...>
    < TextView android:id="@+id/message"
        android:text="@string/bonjour" />
</LinearLayout >

```

La notation `@+id/nom` définit un identifiant pour le `TextView`.

2.3.6. @id/nom ou @+id/nom ?

Dans les fichiers `layout.xml`, il y a deux notations à ne pas confondre :

@+id/nom pour définir (créer) un identifiant

@id/nom pour référencer un identifiant déjà défini ailleurs

Exemple, le Button btn se place sous le TextView titre :

```
<RelativeLayout xmlns:android="..." ... > <TextView ...  
    android:id="@+id/titre"  
    android:text="@string/titre" /> <Button ...  
    android:id="@+id/btn"  
    android:layout_below="@id/titre" android:text="@string/ok" />  
< /RelativeLayout >
```

2.3.7.Paramètres de positionnement

La plupart des groupes utilisent des *paramètres de taille et 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 obligatoires : android:layout_width et android:layout_height, • ceux qui sont demandés par le groupe englobant et qui en sont spécifiques, comme android:layout_weight, android:layout_alignParentBottom, android:layout_centerInParent..

2.3.8.Paramètres obligatoires

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 prend la place minimale
- "match_parent" : la vue occupe tout l'espace restant
- "valeurdp" : une taille fixe, ex : "100dp" mais c'est peu recommandé, sauf 0dp pour un cas particulier, voir plus loin

Les dp sont indépendants de l'écran ([explications](#)). 100dp font 100 pixels sur un écran de 160 dpi

(160 *dots per inch*) tandis qu'ils font 200 pixels sur un écran 320 dpi. Ça fait la même taille apparente quelque soit la finesse des pixels.

Par exemple, trois boutons dans un LinearLayout horizontal :

Bouton	layout_width	layout_height
OK1		
	wrap_content	

```

wrap_content OK2
    wrap_content
    match_parent
OK3    match_parent wrap_content

```

Voir la figure 17,

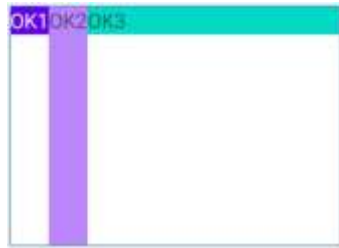


Figure 17: Arbre de vues

2.3.9. Autres paramètres géométriques

Il est possible de modifier l'espacement des vues :

Padding espace entre le texte et les bords,
 géré par chaque vue **Margin** espace autour des
 bords, géré par les groupes figure 18

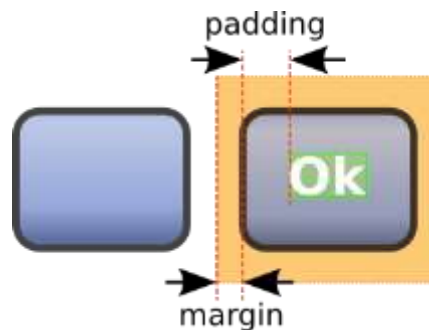


Figure 18: Bords et marges

2.3.10. Marges et remplissage

On peut définir les marges et les remplissages séparément sur chaque bord (Top, Bottom, Left, Right) , ou identiquement sur tous :

```

< Button
    android:layout_m
    argin="10dp"

```

```
android:layout_m  
arginTop="15dp"  
android:padding="  
10dp"  
android:paddingL  
eft="20dp" />
```

C'est très similaire à CSS.

2.3.11. Groupe de vues 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_width="wrap_content"  
        android:layout_height="wrap_content"/>  
    <Button android:text="Annuler" android:layout_width="wrap_content"  
        android:layout_height="wrap_content"/>  
< /LinearLayout >
```

Il faut seulement définir l'attribut `android:orientation` à "horizontal" ou "vertical". Lire la [doc Android](#).

2.3.12. Pondération des tailles

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

Pour cela, il faut aussi fixer la taille de ces vues (ex : `android:layout_width`) soit à "wrap_content", soit à "0dp".

- Si la taille vaut "wrap_content", alors le poids agit seulement sur l'espace supplémentaire alloué aux vues.
- Mettre "0dp" pour que ça agisse sur la taille entière.

2.3.13. Exemple de poids différents

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

En 3^e ligne, les boutons ont une largeur de 0 dp figure 19



Figure 19: Influence des poids sur la largeur

2.3.14. Groupe de vues TableLayout

C'est une variante du LinearLayout : les vues sont rangées en lignes de colonnes bien alignées. Il faut construire une structure XML comme celle-ci. Voir sa [doc Android](#).

```
<TableLayout ...> < TableRow >
    <vue 1.1 .../>
    <vue 1.2 .../>
</ TableRow >
< TableRow >
    <vue 2.1 .../>
    <vue 2.2 .../>
</ TableRow >
< TableLayout >
```

NB: les <TableRow> n'ont aucun attribut.

2.3.15. Largeur des colonnes d'un TableLayout

Ne pas spécifier android:layout_width dans les vues d'un TableLayout, car c'est obligatoirement toute la largeur du tableau. Seul la balise <TableLayout> exige cet attribut.

Deux propriétés intéressantes permettent de rendre certaines colonnes étirables. Fournir les numéros (première = 0).

- android:stretchColumns : numéros des colonnes étirables
- android:shrinkColumns : numéros des colonnes reductibles

```
< TableLayout
    android:stretchColumns="
1,2"
    android:shrinkColumns="0,
3"
    android:layout_width="ma
tch_parent"
    android:layout_height="wr
ap_content" >
```

2.3.16. Groupe de vues RelativeLayout

C'est le plus complexe à utiliser mais il donne de bons résultats. Il permet de spécifier la position relative de chaque vue à l'aide de *paramètres* complexes : ([LayoutParams](#))

- Tel bord aligné sur le bord du parent ou centré dans son parent :
 - android:layout_alignParentTop, android:layout_centerVertical...
- Tel bord aligné sur le bord opposé d'une autre vue :
 - android:layout_toRightOf, android:layout_above, android:layout_below...
- Tel bord aligné sur le même bord d'une autre vue :
 - android:layout_alignLeft, android:layout_alignTop...

2.3.17. Utilisation d'un RelativeLayout

Pour bien utiliser un [RelativeLayout](#), il faut commencer par définir les vues qui ne dépendent que des bords du Layout : celles qui sont collées aux bords ou centrées.

```
<TextView android:id="@+id/titre"
    android:layout_alignParentTop="true"
    android:layout_alignParentRight="true"
    android:layout_alignParentLeft="true" .../>
```

Puis créer les vues qui dépendent des vues précédentes.

```
<EditText android:layout_below="@id/titre"
    android:layout_alignParentRight="true"
    android:layout_alignParentLeft="true" .../>
```

Et ainsi de suite.

2.3.18. Autres groupements

Ce sont les sous-classes de [ViewGroup](#) également présentées dans [cette page](#). Impossible de faire l'inventaire dans ce cours. C'est à vous d'aller explorer en fonction de vos besoins.

En TP, nous étudierons le [ConstraintLayout](#), présenté sur [cette page](#).

2.4. Composants d'interface

2.4.1. Vues

Android propose un grand nombre de vues, à découvrir en TP :

- Textes : titres, chaînes à saisir
- Boutons, cases à cocher...
- Curseurs : pourcentages, barres de défilement...

Beaucoup ont des variantes. Ex: saisie de texte = n° de téléphone, ou adresse, ou texte avec suggestion, ou ...


Consulter la doc en ligne de toutes ces vues. On les trouve dans le package [android.widget](#).

À noter que les vues évoluent avec les versions d'Android, certaines changent, d'autres disparaissent.

2.4.2. TextView

Le plus simple, il affiche un texte statique, comme un titre. Son libellé est dans l'attribut android:text.

```
< TextView android:id="@+id/tvTitre"
    android:text="@string/titre"
... />
```

On peut le changer dynamiquement : 

```
TextView tvTitre = findViewById(R.id.tvTitre) ; tvTitre.setText("blablabla")
;
```

2.4.3. Button

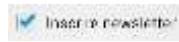
L'une des vues les plus utiles est le [Button](#) : 

```
< Button android:id="@+id/btnOk"
    android:text="@string/ok"
... />
```


- En général, on définit un identifiant pour chaque vue active, ici : android:id="@+id/btnOk"
- Son titre est dans l'attribut android:text.
- Voir la semaine prochaine pour son activité : réaction à un clic.

2.4.4. Bascules

Les **CheckBox** sont des cases à cocher :

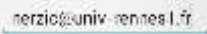


```
< CheckBox android:id="@+id/cbxAbonnementNL"
    android:text="@string/abonnement_newsletter"
    ... />
```

Les **ToggleButton** sont une variante : . On peut définir le texte actif et le texte inactif avec `android:textOn` et `android:textOff`.

NB: l'esthétique de toutes ces vues change avec les versions d'Android.

2.4.5. EditText

Un **EditText** permet de saisir un texte  :

```
< EditText android:id="@+id/email_address"
    android:inputType="textEmailAddress"
    ... />
```

L'attribut `android:inputType` spécifie le type de texte : adresse, téléphone, etc. Ça définit le clavier qui est proposé pour la saisie.

Lire [la référence Android](#) pour connaître toutes les possibilités.

2.4.6. Autres vues

On reviendra sur certaines de ces vues les prochaines semaines, pour préciser les attributs utiles pour une application. D'autres vues pourront aussi être employées à l'occasion.

2.4.7. C'est tout

C'est fini pour ce chapitre , rendez-vous prochainement pour un cours sur les écouteurs et les activités.