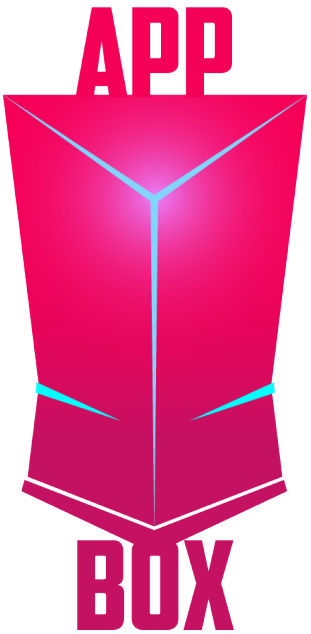
Guide d'utilisation du framework AppBox

Développer rapidement des applications éducatives Android

Auteurs :
LABORDE Victor
SERGUIER Nicklos

Tuteur :
FRANCES Fabrice



23 Juin 2015

Table des matières

Chapitre 1

Informations préalables

1.1 Qu'est ce qu'AppBox

AppBox est un framework, une bibliothèque de composants destiné à accélérer et simplifier le développement d'applications Android. Il a principalement été conçu pour le développement d'applications éducatives mais peut aussi convenir pour tout type d'application.
Ce framework se concentre sur l'aspect fonctionnel en faisant abstraction des composants intermédiaires à mettre en œuvre pour réaliser une fonctionnalité. Il factorise le code lourd et redondant que l'on peut avoir dans l'environnement Android pour vous simplifier le code.

1.2 Pré-requis

Pour comprendre ce guide, il vaut mieux déjà savoir coder en Java. L'environnement de programmation Android n'est pas un langage mais bien un environnement, il mélange du Java (avec de nouvelles classes propres à Android) et du XML pour la gestion de l'affichage. Ce guide vous apprendra les bases du développement en Android avant de détailler l'utilisation du framework en lui même.

Chapitre 2

Les bases d'Android

Voici un court tutoriel sur le langage java Android qui vous accompagnera lors de vos premiers pas en Android. Tout au long de ce tutoriel, nous mettrons en pratique ce qui a été dit au travers de divers exemples de code, ce qui pourra aussi vous donner de nombreuses clés sur l'utilisation des méthodes et des mécaniques d'Android.

2.1 Materiel et logiciel

Avant toute chose, pour pouvoir coder des applications, il vous faut un IDE (environnement de programmation : il contient un compilateur, un éditeur, un debugger,...) tel qu'Eclipse ou Android studio (le plus répandu maintenant, mais c'est la même chose). Il est très facile à télécharger et gratuit. Nous faisons ce tutoriel avec Eclipse, mais les deux fonctionnements sont rigoureusement les mêmes donc pas de panique si nos screenshots ne correspondent pas exactement aux vôtres !

Vous pouvez à présent développer du code, mais à quoi bon si vous n'avez pas de quoi le tester ? Soit vous avez à votre disposition une tablette ou un smartphone et donc ce cas c'est très bien. Nous allons vous expliquer comment la connecter. Si votre tablette ne répond pas quand vous la branchez, essayez ceci : passez en mode debug dans paramètres → Options pour les développeurs et connectez la tablette en USB à votre pc. Ensuite vous faites un clic droit sur poste de travail → gérer → gestionnaire de périphérique → Android device → mettre à jour les pilotes → rechercher un pilote sur mon ordinateur. Ce pilote ce trouve dans le sdk d'Android studio (ou Eclipse) dans extra → google → usb driver.

Si vous ne possédez pas de tablette ou smartphone, vous avez trois options :

- Allez en acheter, c'est la meilleure des solutions à long terme.
- Vous pouvez demander à Android Studio de créer un Android virtual device (AVD), une tablette virtuelle sur laquelle vous pouvez exécuter votre code, mais ce sera très long.
- Le meilleur compromis est de télécharger un émulateur de tablette Android comme BlueStacks. Vous devrez ensuite le synchroniser avec AndroidStudio : allez dans le répertoire d'installation d'AndroidStudio/Eclipse → Sdk → platformTool, créez un fichier texte où vous écrirez «adb connect localhost» et enregistrez le en **.bat**, ainsi vous n'aurez plus qu'à cliquer dessus pour que la synchronisation s'opère. Sinon faites Maj+clic droit dans ce même répertoire → ouvrir une console et tapez le code directement

Vous pouvez maintenant produire du code Android et créer vos propres applications, mais comment faire ?

2.2 Introduction à l'environnement Android

Dans une application Android, on distingue deux choses : la **vue** et le **modèle**.
Le modèle englobe lui même deux catégories d'objets : les classes Java standard et les activités (*Activity*). Les activités peuvent être vues comme les différentes fenêtres de l'application. Pour chaque fenêtre dans laquelle l'utilisateur navigue, il y a une activité derrière. Lors de la création d'un projet, une première activité sera systématiquement créée. Le code que l'on va insérer dedans va permettre de manipuler les différents composants de notre scène, gérer les événements que l'utilisateur va déclencher lors de son interaction avec ces composants, etc.

La vue quant à elle contient tout ce que l'on voit, les éléments graphiques, les boutons, etc. Elle est définie dans le fichier res > layout > xxx_activity.xml. Il y a autant de fichiers .xml que d'activités, à chaque fois que vous créez une nouvelle activité, le fichier .xml correspondant sera créé automatiquement dans le répertoire layout. Ce fichier n'est pas en Java mais en XML, un langage balisé comme le HTML, nous verrons plus tard comment l'utiliser.

2.3 Votre première application

Tout d'abord nous allons créer un projet Android qui sera une nouvelle application. Pour créer le projet : File → new → other → Android application project → name : maPremiereApplication → empty activity → finish. Décortiquons ensemble ce projet en regardant l'exploreur :

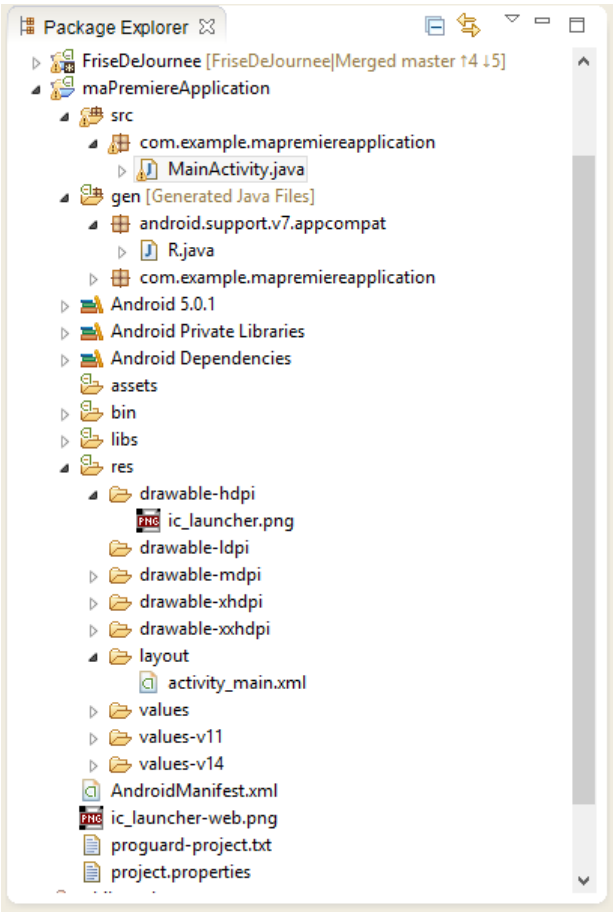


FIGURE 2.1 – Exploreur de l'Android application project

- **SRC :** la source, vous y êtes habitué en Java, c'est là que l'on met tous les packages, les classes et le code en général. C'est ici que se situe le modèle : activités et classes Java.
- **GEN :** les classes auto-générées comme la classe R.java. On ne doit JAMAIS modifier cette classe à la main. Sa mise à jour est automatique dès que l'on rajoute un élément dans la vue, que l'on crée des couleurs, des images ou même des objets, leur ID est stocké dans cette classe R.java.
- **ASSETS :** c'est dans ce fichier que l'on met les polices de caractère par exemple.
- **BIN, LIBS :** les fichiers binaires et les librairies, vous n'y toucherez quasiment jamais.
- **RES :** le deuxième gros morceau avec src, ce sont les ressources de l'application, tout ce qui permet d'alimenter la vue se trouve ici.
- **DRAWABLE :** c'est ici qu'on place nos images(par un simple cliqué-glissé depuis votre dossier). Il y a un fichier drawable pour chaque catégorie d'écran.
- **VALUES :** on définit ici nos couleurs, nos chaines de caractère et nos styles en .xml.
- **LAYOUT :** dans ce répertoire se trouvent tous les fichiers xxx_activity.xml dont nous avons déjà parlé.
- **ANDROIDMANIFEST.XML :** c'est un fichier XML qui sert à définir toutes les activités de l'application et l'activité qui est lancée en premier. Il permet de forcer une activité à se mettre en portrait ou paysage. Mais surtout, il définit quelle version du SDK (Versions d'Android) vous voulez utiliser.

2.4 La première activité

On remarque qu'à la création de notre application, nous avons déjà une activité qui a été créée et un fichier XML associé dans layout. Nous allons ensemble détailler le code de cette *MainActivity.java* :

```
1  package com.example.projettest;
2
3  import android.app.Activity;
7
8  public class MainActivity extends Activity {
9
10     @Override
11     protected void onCreate(Bundle savedInstanceState) {
12         super.onCreate(savedInstanceState);
13         setContentView(R.layout.activity_main);
14     }
15 }
16
```

FIGURE 2.2 – Code de MainActivity

La méthode *onCreate* va s'exécuter, comme son nom l'indique, à la création de l'activité. C'est ici que nous allons mettre tout ce qui permet de rendre interactif le layout(= la vue, l'affichage). Ce layout est ajouté avec la méthode *setContentView*, qui est obligatoire. Votre code doit se placer après cette méthode.
Cette activité bien que quasiment vide est déjà fonctionnelle, vous pouvez la lancer pour voir s'afficher un magnifique hello world ! Ce texte a été défini dans le fichier main_activity.xml. Ce qui est écrit dans le fichier xxx_activity.xml constitue la **première scène** de votre activité. Le code que vous rajouterez dans votre *onCreate* sert à **modifier dynamiquement** cette scène et la rendre interactive !

Pour créer une nouvelle activité, faites un clic droit sur le package dans src → new → other → Android → Android Activity. Choisissez ensuite Empty Activity puis changez son nom (il doit être unique dans le projet), le nom du layout est généré automatiquement, enfin cliquez sur Finish.

2.5 Les layouts

Nous allons maintenant parler des layouts avant de traiter un exemple complet de fichier XML. Les layouts sont des conteneurs, ils peuvent contenir des composants ou d'autres layouts pour les organiser dans l'espace. Un composant est tout ce qui va être visible à l'écran (ils héritent tous de la classe *View*) comme *TextView*, *ImageView*, *Button*, etc. On distingue ainsi une arborescence : un layout parent qui possède plusieurs enfants qui eux même peuvent posséder des enfants, etc.

Les layouts se différencient par leur façon de disposer leurs enfants dans l'espace : Un *LinearLayout* servira à placer les éléments en ligne (verticalement ou horizontalement), alors qu'un *RelativeLayout* permettra de placer les éléments les uns par rapport aux autres et aux bords de l'écran :

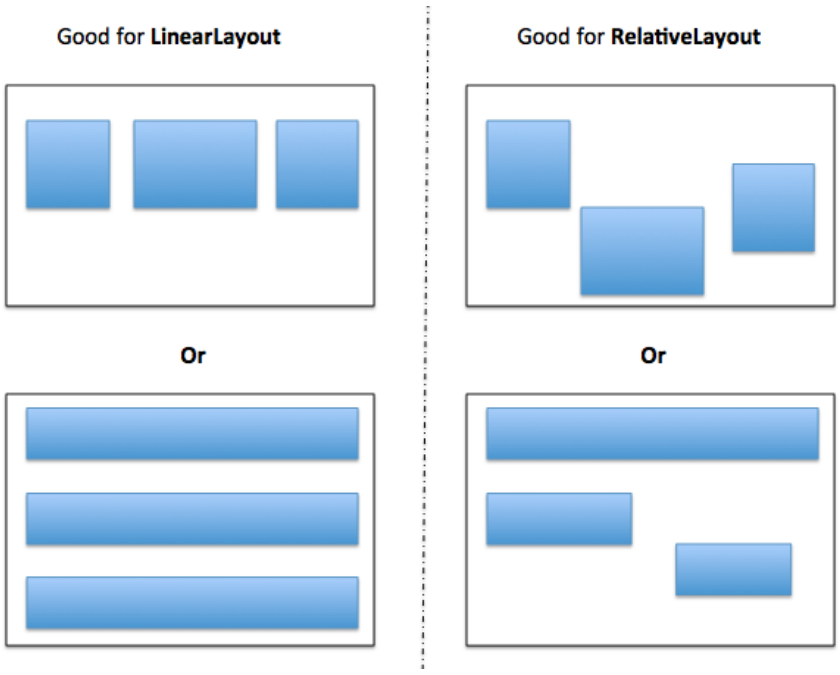


FIGURE 2.3 – Choix des layouts (1)

Il y en a d'autres de disponible, mais combiner ces deux layouts vous permettra déjà de réaliser des configurations complexes !
Pour finir, vous pouvez avoir un aperçu de la scène que votre XML va donner en cliquant sur l'onglet **Graphical Layout** en bas à gauche de la fenêtre lorsque vous êtes sur votre layout.xml :

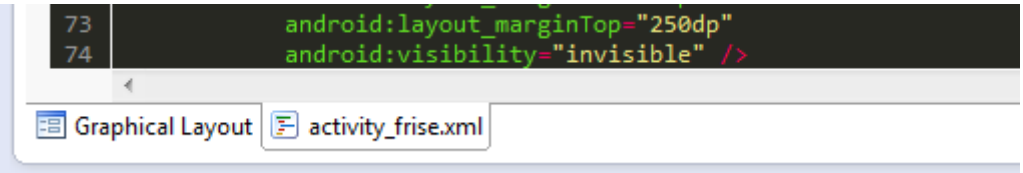


FIGURE 2.4 – Les onglets du XML

2.6 Les fichiers XML

Pour rappel, les fichiers XML vont coder la vue de votre activité. C'est un fichier balisé comme HTML par exemple, chaque balise ouverte doit être fermée. C'est ici que vont intervenir nos fameux conteneurs vus au dessus : une balise signale le début d'un conteneur et une autre la fin. Tous les composants situés entre ces deux balises vont être des enfants de ce conteneur et se positionner selon les règles que celui-ci impose. Ce sera plus clair avec l'exemple :

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/parent"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/purple4" >

    <ImageView
        android:id="@+id/image"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true"
        android:background="@drawable/gnar" />

    <RelativeLayout
        android:id="@+id/fils"
        android:layout_width="300dp"
        android:layout_height="200dp"
        android:layout_below="@id/image"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="50dp" >

        <Button
            android:id="@+id/bouton"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:background="@color/blue2"
            android:text="@string/hello_world"
            android:textAlignment="center"
            android:textColor="@color/red4"
            android:textSize="30sp" >
        </Button>
    </RelativeLayout>

</RelativeLayout>
```

FIGURE 2.5 – Exemple de code XML

Observons les balises, on voit que le conteneur parent est de type *RelativeLayout*, sa balise ouvrante : <RelativeLayout>, sa balise fermante : </RelativeLayout>. Ce qui est ajouté à la balise ouvrante (en vert et jaune), ce sont les attributs du conteneur, dans l'ordre : son identifiant, sa largeur, sa hauteur et son fond. Ces attributs sont toujours de la forme : ***android :nom_attribut = "valeur"***. Les attributs disponibles pour chaque composant sont différents, par exemple ***android :textSize*** n'est disponible que pour les vues pouvant accueillir un texte.

Ensuite on crée un composant de type *ImageView*. On ne lui définit pas de taille précise en pixel mais la valeur *wrap_ content*, cela signifie qu'il va prendre la taille minimale pour afficher son contenu. Le background ajouté est une image placée dans le dossier drawable_xhdpi.

Puis on crée encore un *RelativeLayout* qui constitue donc le parent de tout ce qu'il y a en dessous. On utilise l'attribut *below* et l'identifiant de l'image pour positionner le layout relati-

vement à cette image (ici en dessous). Ce *RelativeLayout* contient un bouton dont on a défini le texte et la couleur. Vous pouvez encapsuler les layouts les uns dans les autres à volonté comme ceci.

Pensez bien à nommer vos composants XML avec l'attribut *id*, cela vous permettra de pouvoir placer d'autres composants relativement à eux mais aussi d'y avoir accès depuis votre activité !
Voici le rendu sur l'émulateur Bluestacks, obtenu uniquement avec cette page XML et une activité vide (hormis la méthode déjà présente) :

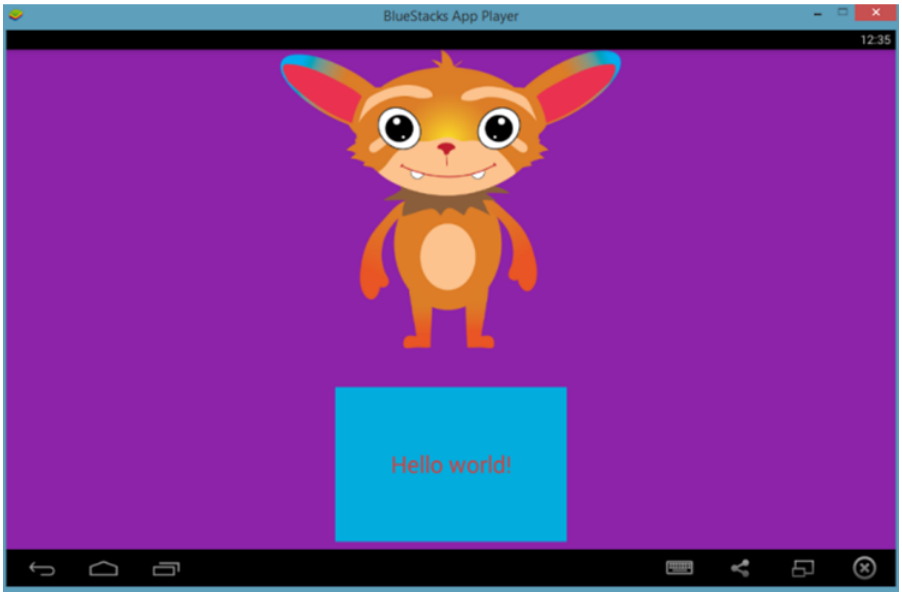


FIGURE 2.6 – Résultat graphique du XML

2.7 Les attributs XML

Voici les principaux attributs qu'il vous faut connaître pour définir correctement votre vue :

- ***android :layout_ width*** → largeur de la vue. Valeurs possibles :
  "wrap_content" taille minimale pour afficher le contenu, à privilégier.
  "match_parent" remplit la largeur du parent, à utiliser pour des barres de navigation par ex.
  "30dp" fixe une taille donnée.

- ***android :layout_ height*** → hauteur de la vue. Valeurs possibles : idem.

- ***android :id*** → l'identifiant unique de votre vue, indispensable si vous voulez accéder à votre composant depuis votre classe Activity. Valeurs possibles :
  "@+id/mon_bouton" après le "/" vous pouvez mettre ce que vous voulez pour nommer votre composant.

- ***android :elevation*** → composante sur l'axe Z, une élévation grande permet au composant de s'afficher par dessus et de créer une ombre sur ceux d'en dessous. Valeur possible :
  "10dp" la valeur que vous souhaitez en dp.
  Attention : attribut disponible uniquement à partir de l'API 21, ne marchera pas sur

Bluestacks.

- ***android :background*** → arrière-plan de votre vue. Valeurs possibles :
"@drawable/mon_image" pour une image.
"@color/red1" remplir avec une couleur unie.

- ***android :margin_top*** → distance à maintenir entre le bord haut de la vue et tout autre composant.Existe aussi en right, left et bottom. Valeur possible :
"10dp" distance donnée, de préférence en dp.

- ***android :padding_top*** → distance à maintenir entre le bord haut de la vue et le contenu de la vue. Existe aussi en right, left et bottom. Valeur possible :
"10dp" distance donnée, de préférence en dp.

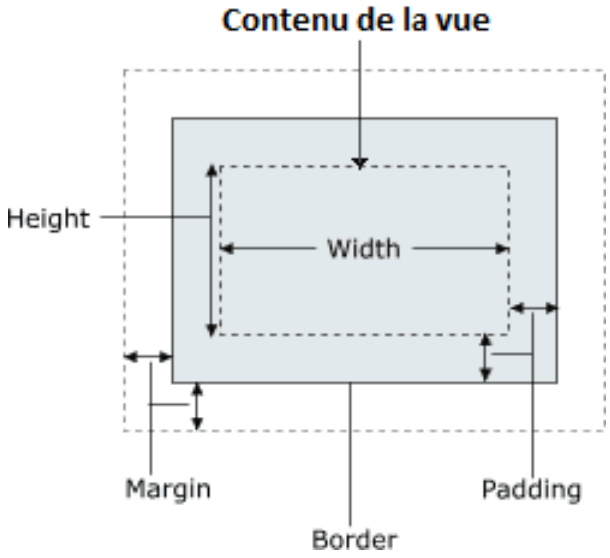


FIGURE 2.7 – Représentation des margins et paddings (2)

Pour les enfants d'un RelativeLayout :

- ***android :layout_below*** → pour placer le composant courant en dessous d'un autre. Existe aussi en above, toRightOf et toLeftOf. Valeur possible :
"@id/autre_composant" ici l'identifiant du composant de référence.

- ***android :align_bottom*** → pour aligner le bas du composant courant avec le bas d'un autre. Existe aussi top, end et start. Valeur possible :
"@id/autre_composant" ici l'identifiant du composant de référence.

- ***android :center_horizontal*** → pour placer le composant au centre horizontalement du RelativeLayout parent. Existe aussi center_vertical et center_inParent. Valeur possible :
"true" tout simplement.

- ***android :align_parentBottom*** → pour aligner un composant au bas du parent. Existe aussi en Top, Right et Left. Valeur possible :
"true"

2.8 Les drawables

Rappel : pour importer une image il suffit de la faire glisser dans un dossier drawable (situé dans **res**). Attention, le nom de votre image ne doit pas contenir d'espace et de lettres majuscules ! (utilisez des noms du type *my_ image*)

2.8.1 Les différents DPI

Nous avons vu que le répertoire drawable était en fait divisé en plusieurs sous répertoires : un pour chaque résolution d'écran. Le point par pouce ou dots per inch (dpi) sert à définir la résolution d'un écran, on retiendra uniquement xhdpi, hdpi et mdpi (dans l'ordre de qualité décroissante) qui correspondent à 3 dossiers drawable dans votre répertoire **res** (dossiers à créer si vous ne les avez pas tous).

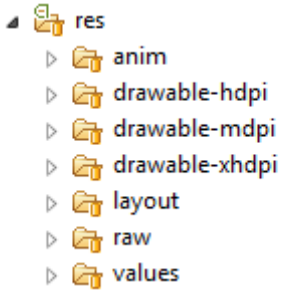


FIGURE 2.8 – Les différents fichiers drawables

Android va sélectionner les images dans le dossier correspondant à la résolution de l'appareil sur lequel votre appli est exécutée. S'il ne les trouve pas, ce n'est qu'ensuite qu'il ira chercher dans les autres dossiers drawable. Mais si vous voulez que votre image rende bien et ai la bonne taille sur toutes les résolutions, il va falloir placer dans chaque dossier drawable une version de votre image adaptée à la résolution. Pour vous aidez dans cette tâche :

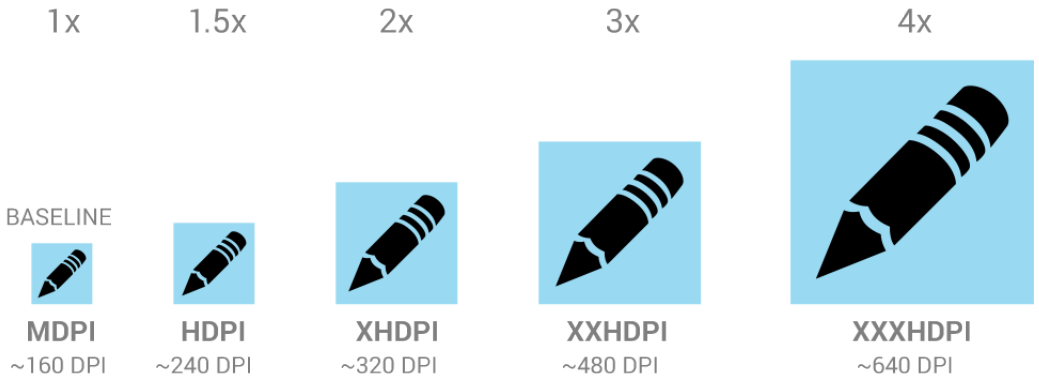


FIGURE 2.9 – Echelle des différentes résolution (3)

Vous avez là les ratios à respecter lorsque vous enregistrerez vos images dans les 3 résolutions citées plus haut. Ainsi si vous avez une image dans drawable-xdhpi qui fait 100*100 px, la version de l'image dans drawable-hdpi devra faire 75*75 px et 50*50 px dans drawable-mdpi. Pour cela utilisez un logiciel type gimp, partez de votre image en xhdpi et appliquez lui des redimensionnements à 75% et à 50%.

2.8.2 Les shapes

Outre les images que vous importez, vous pouvez aussi définir des drawables avec du code XML. Ce seront bien sur des formes relativement simples : des rectangles, des cercles,... mais aussi des empilements de formes. Nous allons voir très succinctement comment faire de simples shapes (on pourrait en faire de plus sophistiquées, mais ce serait trop long). Essayons de construire un bouton arrondi : Tout d'abord clic droit sur l'un des dossier drawable → New → Other → Android → Android XML file → Root element : shape → Finish. Puis nous précisons que la shape est un rectangle, on arrondit ses bords, lui donne un gradient de couleur et un contour :

```xml
<?xml version="1.0" encoding="UTF-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="rectangle" >

    <corners android:radius="50dp" />

    <gradient
        android:angle="90"
        android:endColor="#00828E"
        android:startColor="#4CCFE0" />

    <stroke
        android:width="3dp"
        android:color="#4CB5AB" />

    <size
        android:height="100dp"
        android:width="250dp" />

</shape>
```

FIGURE 2.10 – Code XML de notre shape

Le résultat est le suivant :

FIGURE 2.11 – Rendu du notre shape

Un conseil : utilisez CTRL+ESPACE pour voir les attributs disponibles et la documentation associée !

2.8.3 Les 9-patch

Parfois vous allez devoir étirer vos drawables pour qu'ils s'adaptent aux dimensions d'une vue. Le problème c'est que vous voudriez peut-être que les bords ou d'autres parties de votre image ne soient pas étirés car cela dénaturerait votre image ! Et bien le 9patch est la solution : Vous pouvez choisir les parties de votre image que vous voulez étirer, les autres resterons intactes lors d'un redimensionnement. Voici un exemple sur une bulle pour mieux comprendre :

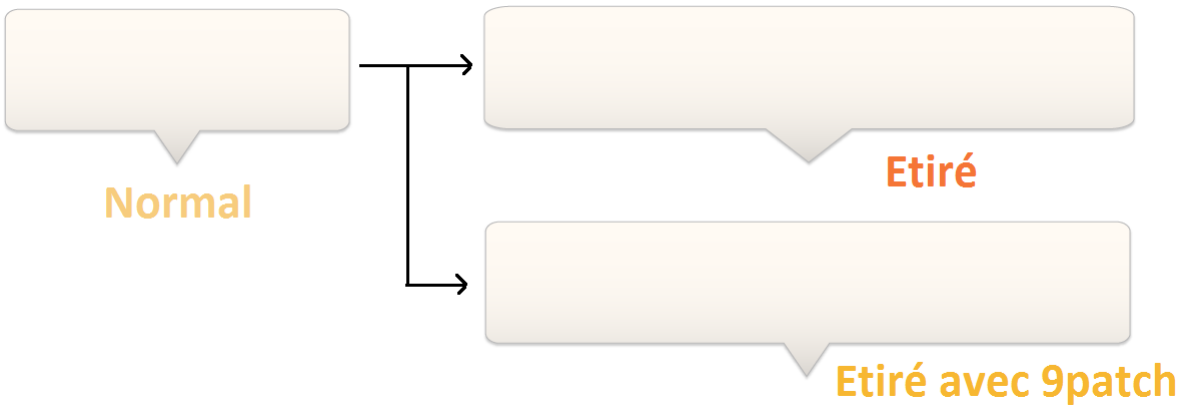


FIGURE 2.12 – Illustration de l'étirement d'une image

Lors de l'étirement normal (celui du haut) les bords arrondis sont étirés sur la longueur, ils ne sont plus symétriques et la pointe de la bulle est elle aussi étirée. En revanche avec l'utilisation du 9patch on voit bien que les bords et la pointe sont conservés !

Maintenant voyons comment transformer une image en une image 9patch (format qui sera interprété par Android) ? Rendez vous dans le répertoire d'installation de votre Eclipse Android/ Android Studio → sdk → tools → draw9patch. Faites glisser votre image dans l'application qui s'ouvre alors. Définissez les zones qui vont s'étirer en faisant glisser la souris en haut de l'image (pour l'étirement horizontal) et à gauche de l'image (pour l'étirement vertical). Les zones marqués en noir sont celles qui vont s'étirer, les autres ne bougeront pas. Voici le résultat pour la bulle par exemple :

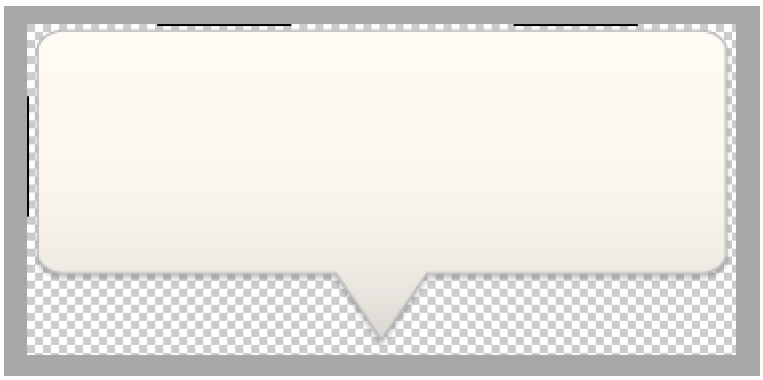


FIGURE 2.13 – Rendu dans le créateur de 9patch

Regardez bien si les aperçus affichés sur la droite vous conviennent, puis enregistrez votre image au format 9patch. Importez là dans votre projet comme une image ordinaire et c'est tout !

2.9 Faire le lien activité-layout

Maintenant que nous avons vu comment ces deux éléments fonctionnaient séparément, il faut maintenant apprendre à câbler tout cela !
Pour accéder à un composant du fichier XML depuis l'activité, c'est très simple : il suffit comme on l'a vu de donner un identifiant à ce composant dans le XML puis dans l'activité d'écrire le code suivant :

```
Button bouton = (Button) findViewById(R.id.bouton1);
bouton.setText("Joli bouton");
```

On récupère le bouton avant de lui appliquer une méthode. L'effet est immédiat sur la vue : le texte du bouton change. Chaque composant XML correspond à une classe Java du même nom (classe Button par ex), on peut donc récupérer n'importe quel composant du XML depuis l'activité. Réciproquement, on peut créer dans l'activité un composant et le placer dans la vue pour la modifier dynamiquement. Souvenez-vous, les composants se placent dans des conteneurs il faut donc récupérer un conteneur si on veut placer un composant dans la vue :

```
LinearLayout boite = (LinearLayout) findViewById(R.id.layout1);
TextView texte = new TextView("Texte dynamique");
boite.addView(texte);
```

On remarque que chaque appel à la méthode *findViewById* donne lieu à un cast, c'est à dire à une conversion de l'objet retourné (de type *View* à la base) au type souhaité. Ce cast est matérialisé par les parenthèses encadrant le type : (Button) et (LinearLayout) ici.
En résumé, placer les composants dans le XML si vous voulez qu'ils soient présents de base dans votre scène. En revanche ajoutez les dynamiquement depuis votre activité si vous voulez qu'ils apparaissent à certaines conditions ou moments.

2.10 Evénements et Listeners

Un *Listenner* est un objet que l'on applique sur une *View* (layout, bouton et plein d'autres choses encore) et qui prévient Java qu'un événement extérieur peut se produire sur cette *View* (un clic par exemple). Le *Listenner* a pour charge de prévenir Java que le clic à été fait. Il va alors exécuter du code que vous aurez défini lorsque l'événement se déclenche.
Nous allons reprendre l'exemple du XML vu précédemment et rendre le bouton cliquable ! Dans l'activité correspondante à ce XML nous allons écrire dans la méthode *onCreate*, à la suite de ce qu'il y a déjà :

```
//Recuperation du bouton
final Button bouton = (Button) findViewById(R.id.bouton);
//Definition du listener
bouton.setOnClickListener(new View.OnClickListener() {

                @Override
                public void onClick(View v) { // Code a executer lors du clic
                        bouton.setBackgroundColor(R.color.light_green4);
                        bouton.setText("Je suis devenu un bouton vert !");
                }
});
```

Voilà comment on utilise un listener, pensez à l'auto complétion lorsque vous voulez écrire un listener appuyer sur CTRL+ESPACE et Android Studio ou Eclipse vous complétera directement le corps de la méthode *onClick* ! Vous remarquerez que le bouton est désigné **final** ce qui veut dire qu'on ne peut plus modifier la référence de l'objet bouton (mais on peut toujours

appliquer des méthodes dessus). Ce **final** est indispensable ici car il rend l'objet bouton visible depuis le *onClick*, ne l'oubliez pas.
Le rendu après le clic sur le bouton est le suivant :

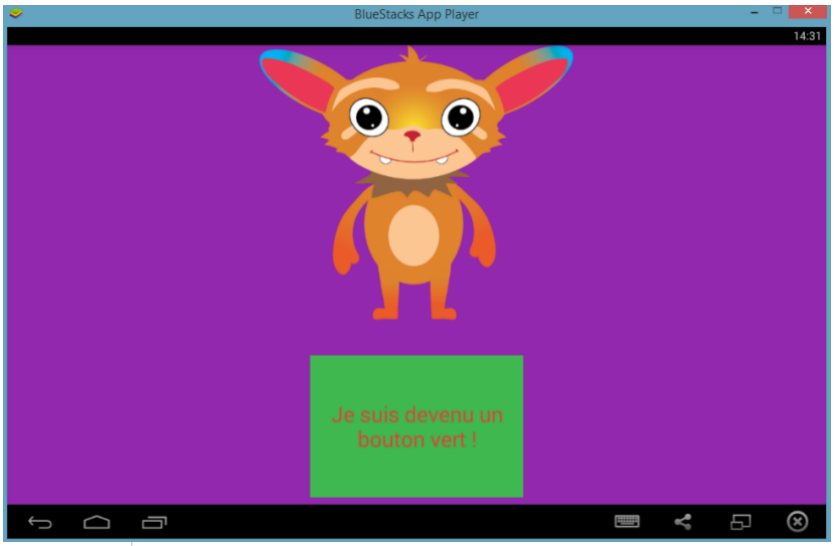


FIGURE 2.14 – Résultat d'une action utilisateur

2.10.1 Les différents listeners

Le listener le plus souvent utilisé est le *OnClickListener* qui écoute les événements relatifs aux clics mais il y en a d'autres, en voici quelques uns qui pourrait vous êtres utile :

- ONTOUCHLISTENER : Gère les événements liés au touché
- ONDRAGLISTENER : Gère les événements liés au glissé
- ANIMATIONLISTENER : Gère les événements liés aux animations. Plus de détails dans la partie animation du chapitre suivant.

2.11 Contexte et Activity

En simplifié, le contexte d'une application est tout ce qui gère les ressources de l'application. Si vous voulez obtenir une ressource comme un drawable (une image) ou bien une couleur, vous allez écrire la ligne suivante :

```
// Recuperation des ressources
Drawable image = getResources.getDrawable(R.drawable.mon_image);
int couleur = getResources.getColor(R.color.red2);
// Utilisation de ces ressources
maVue.setBackground(image);
autreVue.setBackgroundColor(couleur);
```

Comme vous le voyez, nous récupérons les ressources par l'intermédiaire de leur identifiant qui est stocké dans le fichier R.java. Les identifiants sont classés selon le type d'objet stocké (R.drawable, R.color, R.string, R.id etc.). R.id stocke les identifiants des vues, attribués dans le XML par *android :id* ou dans l'activité par *maVue.setId*.

La méthode *getResources* s'applique à un objet *Context*, ici il est implicite car on se situe dans une activité. Mais lors de l'utilisation du framework il faudra souvent passer le contexte en paramètre pour permettre l'utilisation des ressources. Cela se fera simplement avec la variable **this**, lorsque vous êtes dans une activité elle en représente le *Context*.
Ensuite il y a l'*Activity*, c'est sur cet objet que la méthode *findViewById* s'applique, toujours de façon implicite dans une activité. Si besoin, sachez que l'on peut caster le *Context* en *Activity*.
Attention : **this** désigne la classe courante (ou son contexte pour une activité). Donc si vous appelez **this** à l'intérieur du code d'un Listener inclut dans votre activité, **this** ne désignera plus le contexte de l'activité mais le Listener. Vous pouvez néanmoins accéder au contexte avec *getApplicationContext()*.

2.12 Versions d'Android

Google sort régulièrement de nouvelles versions de l'API Android, nous en sommes déjà à l'API 22 à l'heure actuelle. Des versions plus récentes de l'API vous donneront accès à de nouvelles méthodes, mais votre appli ne pourra pas tourner sur les appareils ayant une version d'Android antèrieure, faites donc attention à cette valeur.

Lors de la création de votre projet, il vous est demandé de choisir une **minSdkVersion** et une **targetSdkVersion**. La première détermine la version minimale de l'API qu'il faut avoir pour installer votre appli et les méthodes que vous pourrez utiliser. La seconde sert seulement à dire que vous avez testé votre appli avec cette API mais que vous ne garantissez pas qu'elle fonctionne de façon optimale avec une API supérieure (cette valeur n'est pas très importante). Pour modifier ces valeurs, il faut aller dans le fichier AndroidManifest.xml (en bas dans le navigateur de votre projet) :

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3      package="com.example.projet_test"
4      android:versionCode="1"
5      android:versionName="1.0" >
6
7      <uses-sdk
8          android:minSdkVersion="18"
9          android:targetSdkVersion="22" />
10
11     <application
12         android:allowBackup="true"
13         android:icon="@drawable/ic_launcher"
14         android:label="@string/app_name"
15         android:theme="@style/AppTheme" >
16         <activity
17             android:name=".MainActivity"
18             android:label="@string/app_name" >
19             <intent-filter>
20                 <action android:name="android.intent.action.MAIN" />
21
22                 <category android:name="android.intent.category.LAUNCHER" />
23             </intent-filter>
24         </activity>
25     </application>
26
27 </manifest>
```

FIGURE 2.15 – Code du AndroidManifest.xml

Changez directement les valeurs entre guillemets et enregistrez.

2.13 Ajouter AppBox à votre projet

Avant de passer à la description du framework AppBox, nous allons voir comment l'ajouter à votre projet.
Il faut d'abord récupérer le framework sur GitHub (site de gestion de versions, vous devez vous y inscrire avant toute chose), copier ce lien vers le répertoire Git du framework :
https://github.com/ShuBuster/Framework.

Puis dans Eclipse/Android Studio : File → Import → Git/Projects from Git → URL → Coller l'URL dans le champ, et rentrez vos identifiants et mot de passe GitHub → Next, Next ,Finish. Vous devriez avoir le projet framework qui est apparu dans votre workspace, s'il affiche des erreurs essayez Project → Clean → framework → OK (cela va générer le dossier **gen** et le R.java s'il n'était pas là).

Maintenant pour l'ajouter à votre projet faites un clic droit sur le répertoire de votre projet → Properties → Android → Library → Add → Framework → OK (*2).
Si jamais vous avez des erreurs dans votre projet lorsque vous essayez d'utiliser les fonctions du framework, cliquez sur l'ampoule rouge où s'affiche l'erreur, puis Fix Project Setup (cela résout parfois les problèmes).

Enfin, n'oubliez pas de faire les importations qui vont bien à chaque fois que vous utilisez une classe d'AppBox (raccourci CTRL+MAJ+O).

2.14 Se documenter et débugger

Ce tutoriel est quasiment terminé, vous en savais déjà plus sur l'environnement de développement Android mais bien évidemment vous ne saurez pas tout faire directement. Lorsque vous êtes bloqué, ne perdez pas de temps et allez chercher sur internet la solution à votre problème ! Bien souvent vous allez trouvez facilement, des fois il faut chercher un peu plus profondément... Nous vous conseillons tout simplement de taper "Android" suivi de ce que vous souhaitez faire ou de l'erreur que vous obtenez (en anglais toujours pour avoir plus de résultats). Les réponses sont souvent sur Stack Overflow, un très bon site que nous vous conseillons. Sinon pensez aussi à l'API de Google qui est bien fournie.

Parfois vous ne trouverez pas la solution sur internet, si par exemple une variable prend une valeur qu'elle ne devrait pas et que cela vous affiche des choses incohérentes ou vous génère des erreurs. Vous pouvez alors debugger vous même en mettant des points d'arrêt dans le code (clic dans la marge à gauche sur la bonne ligne) et en lançant votre appli en mode debuggage (icône d'insecte). Cela va vous permettre d'avancer ligne par ligne dans votre code et d'avoir accès aux valeurs des différentes variables. C'est très efficace pour détecter un problème !

Pour ceux qui préfèrent le débuggage au *System.out.println()*, il y a un équivalent en Android c'est le *Log.d("tag","mon message")*. Ce message ne s'affiche pas dans la console mais dans le LogCat (Windows → show view → Other → Android, LogCat). C'est aussi dans cette fenêtre que s'affiche les messages d'erreurs en rouge lorsque votre appli plante, ils peuvent vous apporter beaucoup d'informations donc lisez les !

Chapitre 3

Fonctionnalités du framework AppBox

Maintenant que vous connaissez les bases de l'environnement Android, nous allons apprendre à utiliser le framework AppBox. Pour cela nous allons voir comment utiliser chaque fonctionnalité.

3.1 Animer

Le composant Animer contient une collection d'animations qui permettent à une vue de s'animer de différentes manières. Ce composant sera très utile pour rendre votre activité plus vivante. Attention à ne pas en abuser, une surcharge d'animations peut nuire à la compréhension (effet sapin de Noël).

3.1.1 Les animations disponibles

- Rotation
- Translation
- Échelle
- Fade in → fait apparaître en fondu
- Fade out → fait disparaître en fondu
- Pop in → fait apparaître en grandissant
- Pop out → fait disparaître en rapetissant

3.1.2 Exemple d'utilisation

- Faire tourner une vue :

```
ImageView img = (ImageView) findViewById(R.id.image);
Animer.rotate(img, 1000, 120, true);
```

Ici, l'image va tourner de 120° en 1 seconde puis recommencer indéfiniment. Consultez la documentation Java pour avoir le détails des paramètres et des autres types d'animations simples.

- Faire apparaître une vue :

```
TextView txt = (TextView) findViewById(R.id.texte);
txt.setVisibility(View.INVISIBLE);
```

```
Animer.pop_in(txt, 1000);
```

La vue doit être rendue invisible avant de lancer cette animation pour créer l'effet d'apparition. Ceci est effectué grâce à la méthode *View.setVisibility* ou l'attribut *android :visibility = invisible* dans le XML. La méthode *fade_ in* s'utilise de la même façon.

- Faire disparaître une vue :

```
Button bouton = (Button) findViewById(R.id.bouton);
Animer.pop_out(bouton, 1000,true);
```

La vue disparaît avec le même effet, mais il y a un paramètre booléen en plus. S'il vaut *true* la vue sera retirée de la scène à la fin de l'animation, s'il vaut *false* la vue sera simplement rendue invisible.

3.1.3 Les animations listener

Parfois on aimerait **exécuter une action relativement au déroulement d'une animation**, par exemple au début, à la fin ou même à chaque fois qu'elle se répète. AppBox ne vous permet pas de faire cela car les composants ne peuvent pas prendre en paramètre une méthode. Vous devez alors directement coder votre animation (s'aider du code du composant Animer), et lui appliquer un *AnimationListener*, qui va s'occuper de lancer vos actions au moments clés de l'animation.

```
ImageView img = new ImageView(this);
TranslateAnimation translation = new TranslateAnimation(0, 10, 0,
    20);
translation.setAnimationListener(new AnimationListener() {

        @Override
        public void onAnimationStart(Animation animation) {
                // Code a executer au demarrage de cette animation
        }

        @Override
        public void onAnimationRepeat(Animation animation) {
                // Code a executer a chaque repetition
        }

        @Override
        public void onAnimationEnd(Animation animation) {
                // Code a executer a la fin de cette animation
        }
});
img.startAnimation(translation);
```

Pour aller plus vite vous pouvez écrire uniquement *translation.setAnimationListener(new Ani...* et l'auto-complétion lors du choix de l'AnimationListener vous remplira le corps des 3 méthodes

directement. Si vous ne voulez utiliser que le *onAnimationEnd* par exemple, il faut tout de même laisser les autres méthodes sinon Java ne compilera pas.

3.1.4 Position réelle d'une vue animée

Les animations font en effet bouger le visuel d'une vue mais elle ne font pas bouger l'objet lui même ! C'est-à-dire que si un objet est cliquable et se déplace à un autre endroit avec la méthode *translate* du framework, le clic s'effectuera toujours sur sa position initiale.

Par défaut, une vue qui est animée retournera à sa position initiale à la fin de l'animation. La méthode *Animation.setFillAfter(true)* permet de faire en sorte que la vue reste à sa position finale. C'est dans ce dernier cas que l'image de la vue et la vue réelle ne sont pas au même endroit... Pour remédier à cela il suffit souvent de déplacer "à la main" la vue en fin d'animation (en utilisant un *AnimationListener* !) en modifiant ses LayoutParams (que l'on verra plus tard).

3.2 AnimatedGnar

Pour rendre les applications plus vivantes et ludiques, nous avons créé une mascotte appelée Gnar qui suit l'enfant tout au long de sa progression dans l'application. Au début il était statique, nous avons ensuite décider de l'animer en le découpant en différentes parties mobiles entre elles.

FIGURE 3.1 – Gnar, la mascotte

L'animation finale le fait bouger légèrement pour que l'on ait l'impression qu'il respire, il cligne des yeux également. De plus lorsqu'on lui clique dessus, son visage change ! Tout ceci est disponible en une seule ligne grâce à AppBox :

```
RelativeLayout RL = (RelativeLayout) findViewById(R.id.rl);
AnimatedGnar.addGnar(this,RL);
```

RL désigne le *RelativeLayout* dans lequel le personnage sera placé, créez en simplement un dans votre xxx_activity.xml avec un ID pour pouvoir le récupérer dans l'activité et des contraintes de placement que le personnage aura aussi.
Vous pouvez aussi ajouter un bébé Gnar pour les plus petits avec *addMiniGnar* !

3.3 AnimatedText

Encore un composant qui donnera rapidement de la vie à un menu ou tout autre activité. Il s'agit d'un texte animé, pour l'instant la seule animation disponible est une animation de "vague" mais vous pouvez en créer facilement d'autres en vous aidant du code. L'animation fait monter puis redescendre les lettres les unes après les autres pour créer cette impression de vague, elle se répète indéfiniment.

FIGURE 3.2 – Animation de vague sur titre

Pour ajouter un texte animé :

```
LinearLayout layout_titre = (LinearLayout) findViewById(R.id.titre);
int[] colors = {R.color.light_green3,R.color.light_green4,R.color.
    light_green5};
AnimatedText.add(this, layout_titre,"Hey",colors, 80);
```

Le titre doit être placé dans un *LinearLayout* vide. Les couleurs stockées dans le tableau colorent chacune une lettre du texte, dans l'ordre. Il faut autant de couleurs que de lettres. Le dernier argument : 80 donne la taille de la police souhaitée.
Même si nous l'avons imposé, on peut aussi modifier l'espacement entre les lettres, pour cela il faut directement le changer dans le code du framework.

3.4 Bulle

Le système des bulles d'aide peut être très utile pour guider les enfants dans votre application, en leur disant sur quel bouton cliquer ou donner des informations sur un composant de la scène. Les bulles se placent relativement par rapport à un objet, en dessus, en dessous, à gauche ou à droite :

FIGURE 3.3 – Les bulles d'aide en action

Le texte peut être plus long, les bulles s'adapteront à la bonne taille en gardant le texte bien au centre. Pour ajouter des bulles d'aide :

```
Button bouton = (Button) findViewById(R.id.bouton_bulle);
TextView bulle1 = Bulle.create(bouton, "Je suis une bulle a droite","right",true,
    this);
TextView bulle2 = Bulle.create(bouton, "Je suis une bulle a gauche","left",true,
    this);
```

Ici c'est un bouton mais les bulles d'aides peuvent se placer sur n'importe quelle vue. Les lieux des bulles sont "right","left","above" et "below". Le paramètre booléen est à mettre à *true* si vous voulez que la bulle apparaisse directement à sa création, *false* si vous voulez qu'elle soit invisible. En stockant la bulle dans une variable comme dans le code ci-dessus, vous pourrez la faire apparaître au moment que vous souhaitez comme après une action de l'enfant, en utilisant par exemple *pop_in* ou *fade_in* pour un effet réussi ! De même pour la faire disparaître.

3.5 Horloge

Si vous voulez incorporer dans votre application une horloge colorée et sympathique, c'est très simple en utilisant AppBox ! Vous pouvez créer une horloge sur une heure fixe ou bien une horloge qui est à l'heure en temps réel. Voici la marche à suivre :

```
RelativeLayout horloge_conteneur = (RelativeLayout) findViewById(R.id.
    horloge);
Horloge.create(horloge_conteneur,this,13,22,30);
```

La commande ci-dessus créera une horloge fixe indiquant 13h 22min 30s. Le RelativeLayout donné en paramètre doit être vide et placé dans votre scène à l'endroit où vous voulez que votre horloge se place.

FIGURE 3.4 – Rendu de l'horloge

Pour une horloge en temps réel, retirez simplement les 3 paramètres liés à l'heure pour ne laisser que les deux premiers arguments.

3.6 MyLayoutParams

3.6.1 Qu'est ce que'un LayoutParams

Nous avions vu dans le chapitre sur les bases d'Android que nous pouvions ajouter des composants à la scène depuis l'activité avec notamment la méthode *layout_conteneur.addView(maVue)*.

Mais où la vue se place-t-elle dans le conteneur ? Ici nous n'avons rien spécifié, elle ira donc par défaut tout en haut à gauche du layout conteneur/parent. Dans le XML nous pouvions choisir l'emplacement grâce à des attributs liés au *RelativeLayout* comme *toRightOf*, ou bien grâce aux margins.

Et bien pour spécifier ces attributs de placement depuis l'activité il vous faut utiliser un *LayoutParams*. Un objet *LayoutParams* contient toutes les informations de placement que l'on retrouve dans le XML : les marges, les règles de placement relatifs etc. Cet objet peut ensuite être attribué à une vue pour que celle-ci soit placée selon ces règles.

3.6.2 Utiliser MyLayoutParams

Utiliser la classe *LayoutParams* nous a semblé fastidieux et lourd en syntaxe, nous avons donc créé le composant *MyLayoutParams* héritant de *LayoutParams* qui la rend plus simple d'utilisation.

Par exemple, voici comment placer une vue nommée maVue en bas du RelativeLayout nommé parent, centrée horizontalement avec une marge par rapport au bas de 10 :

```
MyLayoutParams params = new MyLayoutParams().alignParentBottom().
    centerHorizontal().marginBottom(10);
parent.addView(maVue,params);
```

C'est aussi simple que cela : les instructions de règle de placement sont mises à la chaîne comme ceci, dans n'importe quelle ordre. Et les paramètres ainsi créés sont passés en paramètre de *addView* pour spécifier que maVue doit être positionnée de telle façon. Regardez la documentation des méthodes de *MyLayoutParams* pour savoir tous les placements disponibles !

Avec le constructeur par défaut, *myLayoutParams* affecte aux dimensions de la vue les valeurs *wrap_content* mais vous pouvez utiliser le deuxième constructeur pour spécifier d'autres dimensions.

3.7 Bouton

Le composant Bouton permet de créer des boutons à l'allure et aux couleurs variées et permet également de lui donner quelques fonctionnalités de base très utiles pour des applications éducatives comme le text-to-speech.

3.7.1 Personnaliser ses boutons

La classe Bouton est celle qui permet de créer des boutons personnalisés : on peut définir la couleur et la forme (arrondi ou carré) du bouton. Cette classe permet aussi bien de créer un *Button* ou un *ImageButton*, c'est-à-dire une image cliquable. Voici deux exemples de code qui permettent de créer un bouton arrondi de couleur ambre :

```
Button bouton = (Button) findViewById(R.id.bouton);
Drawable d = Bouton.roundedDrawable(this,R.color.amber7, 0.5f);
bouton.setBackground(d);
```

Le résultat est le suivant :

FIGURE 3.5 – Rendu du "rounded" bouton

Une autre façon de le faire est de créer directement un bouton avec la méthode *createRoundedButton* et de le positionner au centre avec des *LayoutParams*.

La classe propose aussi un builder de bouton qui vous permet de créer rapidement un bouton en chaînant toutes les instructions, voici un exemple d'utilisation :

```
ViewGroup parent = (ViewGroup) findViewById(R.id.parent);
Drawable draw = Bouton.roundedDrawable(this,R.color.amber5, 1f);
Button bouton = Bouton.create(this)
                .setBack(draw)
                .setText("bouton")
                .setTextColor(R.color.blanc_casse)
                .setTextSize(30)
                .build();
```

Le résultat est le suivant :

FIGURE 3.6 – Résultat du builder de bouton

Le dernier attribut de *roundedDrawable* est un float qui représente un facteur multiplicateur de la largeur du bouton par rapport à une largeur de référence.

3.7.2 Fonctionnalités utiles sur les boutons

Dans cette section nous allons voir quelques fonctions utiles toutes faites pour les boutons.

- TEXT-TO-SPEECH
- LISTENER DE CHANGEMENT D'ACTIVITÉ
- GLOW SUR LES BOUTONS

3.7.2.1 Faire parler un bouton

Cette fonctionnalité permet d'ajouter à un bouton une lecture de texte lors du clic, avec la voix d'Android par défaut. par exemple on peut faire parler notre bouton dernièrement créé en rajoutant cette ligne de code :

```
TTSBouton.parle(bouton, "je parle !", this);
```

Quand on clique sur le bouton la voix dit "je parle" ! On peut aussi le faire taire avec la méthode *TTSBouton.fermer(bouton,Context)*.

3.7.2.2 Changer d'activité

Cette fonctionnalité est un *ClickListener* pré-câblé qui est ajouté à un bouton est qui permet de changer d'activité instantanément en cliquant sur le bouton. On peut rajouter des paramètres comme l'image du bouton enfoncé ou bien faire passer un paramètre de type *Serializable* d'une activité à l'autre (transfert d'informations entre activités). Voici un exemple d'utilisation pour passer de *TestActivity* à *MainActivity* :

```
bouton.setOnClickListener(new NextActivityListener(bouton,
d_pressed,TestActivity.this, MainActivity.class));
```

Et en voici un autre qui inclut le passage d'un paramètre, un entier par exemple mais ça peut-être n'importe quel objet qui implémente l'interface *Serializable* :

- code dans l'activité actuelle *TestActivity* :

```
bouton.setOnClickListener(new NextActivityListener(bouton,
d_pressed,TestActivity.this, MainActivity.class,2,"parametre"));
```

- code dans l'activité suivante *MainActivity* :

```
// recuperation du parametre
Intent i = getIntent();
int entier = (Integer) i.getSerializableExtra("parametre");
// entier = 2
```

C'est en utilisant les *Intent* que l'on peut passer des informations d'une activité à l'autre, c'est très pratique car cela permet de vraiment relier vos activités entre elles, alors n'hésitez surtout pas à en mettre !

3.7.2.3 Surbrillance de boutons

Pour le moment, cette fonctionnalité ne fonctionne qu'avec deux types de boutons : nos boutons ronds help et home (et leurs variantes enfoncées help_e et home_e) :

FIGURE 3.7 – Aspect des boutons help et home

Pour obtenir ces images de boutons, copier les depuis le dossier drawable-xhpdi du framework. Si vos boutons n'ont pas ce background, cela marche quand même mais le cercle du glow ne sera pas adapté est l'effet sera plutôt moche. Il vous faudra donc le modifier pour qu'il marche pour vos boutons (mais rien de très compliqué rassurez-vous !). Voici un exemple de code qui utilise l'effet "glow" sur un bouton de retour au menu principal :

```
Button bouton = (Button) findViewById(R.id.glow_bouton);
bouton.setBackground(getResources().getDrawable(R.drawable.home));
Bouton.makeGlow(bouton, this,1);
```

Ce qui donne le résultat suivant :

FIGURE 3.8 – Bouton home en surbrillance

3.8 La fabrique de menus

Toute application Android commence par un menu, c'est l'élément indispensable et récurrent à toute appli. C'est aussi fastidieux et un peu toujours la même chose. C'est pourquoi AppBox vous propose de les créer pour vous !

Avant de détailler la forme que l'on a choisi pour eux, revenons sur le principe de la fabrique. Nous avons rangé nos différents menus (océan, jungle,...) derrière le même type *Menu* via une interface. La classe *FabriqueMenu* permet ensuite de spécifier quel menu nous voulons via un paramètre de type *TypeMenu*. C'est une énumération (on peut la compléter quand on rajoute de nouveaux menus) qui permet d'accéder aux paramètres des menus comme leur background par exemple.

La fabrique permet de rajouter facilement et à l'infini de nouveaux menus juste en rajoutant une classe et un nouvel attribut dans l'énumération *TypeMenu* . cela permet de rendre le code

très modulaire.

3.8.1 Les différents menus

Pour le moment il existe 4 menus qui diffèrent selon leur orientation, couleur, background et animation. Néanmoins, il sont tous codés (excepté les animations) sur la même structure ce qui rend très facile l'ajout d'un nouveau menu. Voici l'organisation générale des éléments dans les menus déjà créés :

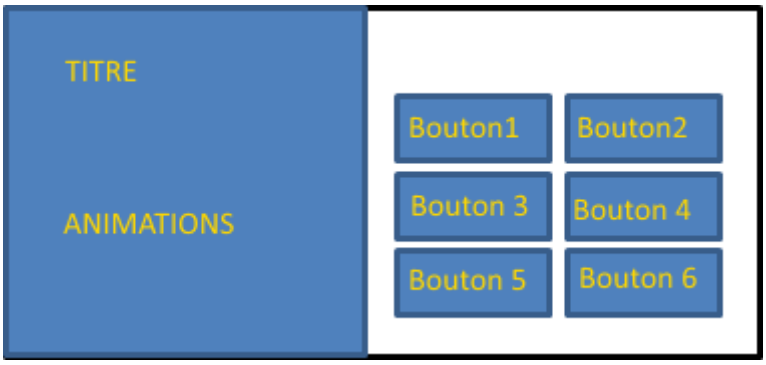


FIGURE 3.9 – Organisation des éléments dans le menu

Une fois créée cette disposition, vous avez des méthodes qui permettent d'ajouter un titre, des boutons dans les emplacements réservés et même de fusionner deux emplacements horizontaux en un si vous le souhaitez. Toutes les animations utilisées sont celles du composant Animer d'AppBox donc vous pouvez vous les approprier très facilement !

3.8.2 Exemple de code

Je souhaite créer un jeu qui se déroule dans une jungle et j'aurais donc besoin d'un menu principal ayant rapport à ce thème, voilà comment je dois procéder :

```
ViewGroup parent = (ViewGroup) findViewById(R.id.parent);
try {
        m = FabriqueMenu.create(TypeMenu.JungleHorizontal,this);
} catch (IllegalArgumentException | InstantiationException
                | IllegalAccessException e) {
        e.printStackTrace();
}

//on cree le menu et on l incorpore a l appli
m.createMenu(parent);
// on rassemble les premiers boutons
m.rassembler(1, 2);
m.rassembler(3, 4);
m.addTitre("MEMORY GNAR !! ");
//on cree les boutons, on pourra plus tard leur donner une tache
jouer = (Button) m.addButton("Jouer", 1);
decor = (Button) m.addButton("Autre decor !",3);
m.addButton("Aide de GNAR ?", 5);
m.addButton("tes scores !!!", 6);
```

Voici le résultat pour le menu jungle horizontal :

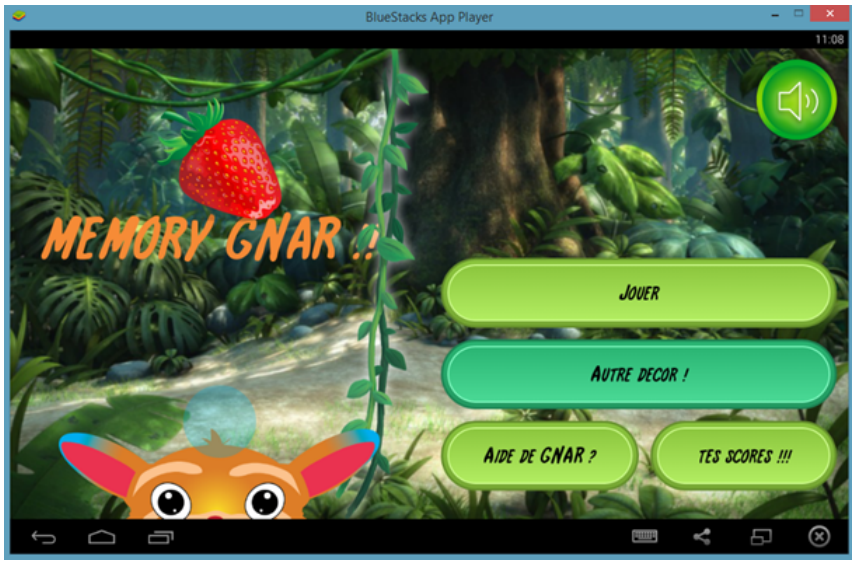


FIGURE 3.10 – Résultat final du menu jungle

Un autre intérêt de ces menus est qu'ils sont indépendant du code XML, ils sont entièrement codés en java et donc applicables dans n'importe quelle activité sans conflit. Il faut simplement fournir au menu un layout dans lequel s'inscrire et le tour est joué ! Il ne reste plus qu'à donner aux boutons un peu de fonctionnalités.

3.9 Utile

Dans le composant *Utile* nous avons rassemblé toutes les méthodes isolées mais néanmoins utiles que nous avons créés pour AppBox.

3.9.1 De belles couleurs

Ceci n'est pas une méthode du composant *Utile* mais peut quand même vous servir ! Vous pouvez définir vos propres couleurs dans votre projet à l'emplacement Android res > values > color.xml. Par défaut, ce fichier n'est pas créé dans votre projet, mais vous avez juste à copier/coller celui présent au même endroit dans AppBox. De nombreuses couleurs ont déjà été rentrées pour vous faire gagner du temps, mais vous pouvez rajouter les vôtres. Voici la palette de couleur qui vous permettra de vous y retrouver :

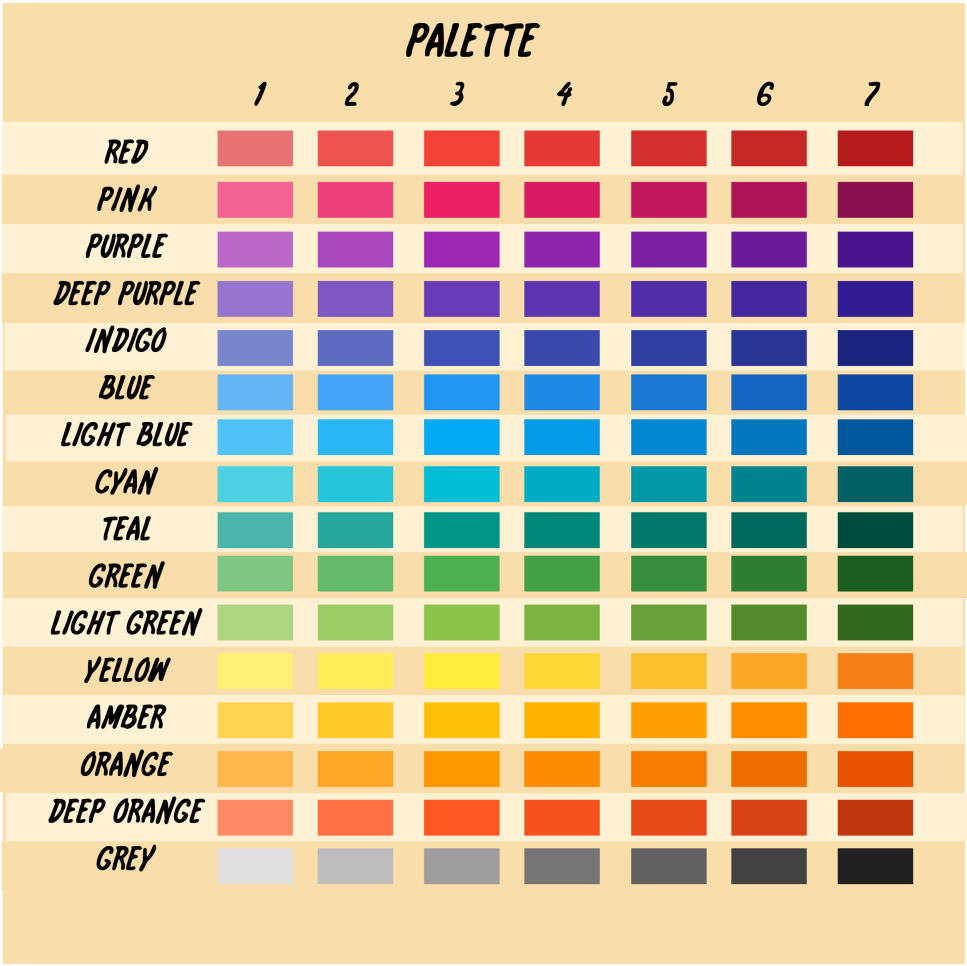


FIGURE 3.11 – Palette des couleurs disponibles

3.9.2 Opérations sur les couleurs

Vous pouvez réaliser des opérations sur les couleurs : les assombrir ou les éclaircir. Voici comment faire :

```
int couleur = getResources().getColor(R.color.blue4);
int couleur_sombre = Utile.darkenColor(couleur);
int couleur_claire = Utile.lightenColor(couleur);
```

Attention à ne pas confondre *R.color.bleu4* avec la couleur elle-même, ce n'est que l'identifiant de la couleur (un entier). Il faut passer par *getResources.getColor()* pour obtenir la couleur (qui est un autre entier).

3.9.3 Passer en plein écran

Par défaut, si vous lancer votre application sur tablette ou émulateur, vous aurez les barres de navigation visibles en haut et en bas de l'écran. Cela peut être gênant pour l'immersion dans l'application. Il est souvent préférable de passer en mode plein écran, pour cela dans chaque activité vous devez introduire le code suivant :
→ Dans *onCreate* :

```
Utile.fullScreen(this);
setContentView(R.layout.my_activity);
```

Il est important de faire l'appel à la méthode *fullScreen* avant le *setContentView* (déjà présent dans votre *onCreate*), sinon il y aura une erreur.
→ Dans *onResume* :

```
@Override
public void onResume() {
        super.onResume();
        Utile.fullScreenResume(this);
}
```

Le code ci dessus doit être dans votre activité, en dessous de *onCreate*. Il sera toujours possible d'activer les barres une fois dans l'application, en glissant le doigt tout en bas de l'écran vers le haut. Ainsi vous pouvez toujours quitter l'application, même si elle plante.

3.9.4 S'adapter à la taille de l'écran

Les supports Android sont multiples et il faut que votre application s'adapte à la taille de l'écran si vous voulez conservez votre rendu sur tous les supports. Une bonne habitude est de ne pas utiliser des valeurs en pixels (px) car cela va donner des résultats différents sur les différentes tailles d'écran. Une alternative est d'utiliser les dp (=dip=Density-independent pixels) notamment dans l'utilisation des marges dans le XML.

Pour ce qui est de la taille des vues dans votre activité, nous vous conseillons de mettre des fractions de la largeur et de la hauteur de l'écran. Par exemple en utilisant *setSize* qui redimensionne une vue :

```
Utile.setSize(maVue,height_screen/2,3*width_screen/4);
```

Nous modifions la taille de *maVue* (en pixels !), mais comme on le fait en fonction de la taille de l'écran le rendu sera le même pour n'importe quel écran. Pour obtenir la taille de l'écran, il vous suffit d'utiliser la méthode *getScreenSize* :

```
int[] size = Utile.getScreenSize(this);
width_screen = size[0]; // largeur de l ecran en px
height_screen = size[1]; // hauteur de l ecran en px
```

3.10 Drag and Drop

Nous avons créé un mécanisme de Drag and Drop (glissé-déposé) dans le framework. Il vous suffit de définir les vues qui sont déplaçables et les vues qui serviront de zone de dépôt pour les premières. Voici comment faire :

```
// L objet drag and drop
MyDragAndDrop dnd = new MyDragAndDrop(this);

// Objets que l on veut deplacer
dnd.addDrag(R.id.cercle1);
dnd.addDrag(R.id.cercle2);

// Zones de depot
dnd.addDrop(R.id.zone1, R.drawable.shape, R.drawable.shape_drop);
dnd.addDrop(R.id.zone2, R.drawable.shape, R.drawable.shape_drop);
```

Les objets sont repérés par leur identifiant définit dans le XML. Les zones de dépôt doivent être des *RelativeLayout*, les vues lâchées dedans deviendront leurs filles et se mettront en leur centre (effet "magnétique"). Toujours pour les zones de dépôt, il faut définir deux images : la première est celle que la zone prend en temps normal, la seconde est celle qu'elle prend lorsqu'un objet est en train d'être glissé au dessus d'elle (généralement ajouter de la surbrillance). Vous pouvez garder toujours la première image en entrant 0 à la place de la deuxième. Voici le rendu en utilisant des cercles comme images :

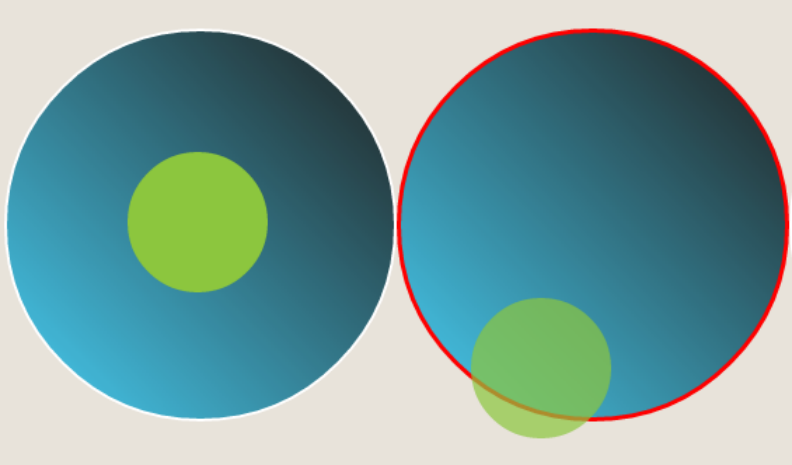


FIGURE 3.12 – Résultat du Drag and Drop

Toutefois, le drag and drop en l'état ne peut pas gérer les événements lorsqu'un objet est lâché dans une zone de drop. Pour exécuter vos propres actions, vous pouvez simplement copier le package dragAndDrop dans votre projet et ajouter le code à exécuter à l'endroit marqué d'un TODO dans la classe *MyDragListener* :

```
61          case DragEvent.ACTION_DROP: /* Drop de l'objet */
62          if(vide){
63              vide= !vide;
64              View view = (View) event.getLocalState(); // la vue qui est drag au dessus
65              ViewGroup owner = (ViewGroup) view.getParent();
66              owner.removeView(view);
67              RelativeLayout dropTarget = (RelativeLayout) v; // la vue qui sert de drop zone
68              dropTarget.addView(view);
69              view.setVisibility(View.VISIBLE);
70              //TODO ICI PLACER LE CODE A EXECUTER LORSQUE L'OBJET EST LACHE DANS LA ZONE
```

FIGURE 3.13 – L'un des "TODO" dans MyDragListener

Vous pouvez aussi rajouter des actions lorsqu'un objet déplaçable est touché ou passé au dessus de la zone, etc. Pour cela cherchez juste le bon TODO dans *MyTouchListener* et *MyDragListener*.

Bibliographie

Sites :

- API Android Developer, Google [consulté janvier-juin 2015] Disponible sur https://developer.android.com/index.html.
- GitHub [consulté janvier-juin 2015] Disponible sur https://github.com.
- OpenClassRoom [consulté décembre 2014-janvier 2015] Disponible sur http://openclassrooms.com.
- Stack Overflow [consulté décembre 2014- juin 2015] Disponible sur http://stackoverflow.com.

Images :

- (1) Codelearn.org, *Android Layout* [consulté le 21 juin 2015] Disponible sur http://www.codelearn.org/android-tutorial/android-layout.
- (2) OffRoad Urugay, *Margin CSS* [consulté le 21 juin 2015] Disponible sur http://offroaduruguay.org/img/margin-css.html.
- (3) Joan Zapata, *Android et les icônes, mdpi, hdpi, xhpdi, xxhdpi...*, Excilys Lab [consulté le 21 juin 2015] Disponible sur http://labs.excilys.com/2013/07/18/android-et-les-icones-mdpi-hdpi-xhdpi-xxhdpi.