Tutoriel Android sous Eclipse - TP de prise en main

Dima Rodriguez

Tutoriel Android ™

TP de prise en main

Dima Rodriguez

Polytech' Paris Sud

Tutoriel Android^{TM}

Dima Rodriguez

Novembre 2014

TP de prise en main

Table des matières

Table des figures

Préambule

Le système d'exploitation ANDROID est actuellement l'OS le plus utilisé dans le monde faisant tourner des smartphones, tablettes, montres connectées, liseuses électroniques, télévisions interactives, et bien d'autres. C'est un système, *open source* qui utilise le noyau Linux. Il a été créée par ANDROID, Inc. qui fut rachetée par GOOGLE en 2005. Le développement d'applications pour ANDROID s'effectue en Java en utilisant des bibliothèques spécifiques.

Le but de ce tutoriel est de vous familiariser avec l'esprit de développement ANDROID et ses bibliothèques. Nous introduirons les concepts de bases de création d'application en mettant en œuvre quelques fonctionnalités simples. Ce tutoriel n'est en aucun cas exhaustive, le potentiel des applications ANDROID est beaucoup plus ample, les exemples cités dans ce document ne devront pas brider votre imagination ni votre curiosité.

Sur le site officiel pour les développeurs ANDROID vous trouverez la documentation des classes, des tutoriels ainsi que les lignes directrices pour préparer une distribution GOOGLE PLAY. Un lexique à la fin de ce document définit quelques mot du vocabulaire ANDROID utilisé dans ce tutoriel.

1 Installation de l'IDE

Dans cette section nous allons décrire la procédure d'installation d'un environnement de développement ANDROID.
Attention : Il faut exécuter les étapes dans l'ordre cité ci-dessous.

a. Téléchargez le JDK7 (*Java Development Kit*) que vous pouvez trouver sur le site d'ORACLE [1].

b. Désinstallez des éventuelles versions antérieures du JDK

c. Installez le nouveau JDK

d. Téléchargez le paquet ADT (*Android Developer Tools*). Il contient le SDK (*Software Development Kit*) ANDROID et une version d'ECLIPSE avec ADT intégré.

e. Pour installer l'IDE, il faut juste placer le dossier téléchargé dans le répertoire où vous avez l'habitude d'installer vos programmes (ou directement sur votre partition principale) et le dé-zipper. Vous pouvez également lui changer de nom si vous souhaitez, mais veillez à ne pas mettre des espaces ou des accents quand vous le renommez.

f. Dans le dossier dé-zippé vous trouverez un exécutable ECLIPSE que vous pouvez désormais lancer pour commencer la configuration de votre environnement.

Au moment de l'écriture de ce document, ECLIPSE est le seul IDE (*Integrated Development Environment*) officiellement supporté. Un nouvel environnement, ANDROID STUDIO, est en cours de développement mais est encore en version bêta pas très stable.
Si vous souhaitez utiliser une version d'ECLIPSE que vous avez déjà sur votre machine il faudrait prendre le SDK et un *plugin* ADT et configurer ECLIPSE pour son utilisation.

1. Ce tutoriel a été réalisé avec JDK7u60

2 Configuration de l'IDE

Installation des paquets supplémentaires et des mises à jours

a. Lancez ECLIPSE

b. On commencera par s'assurer que l'environnement installé est à jour. Dans le menu *Help* sélectionnez *Check for Updates* et installez les mises à jour le cas échéant.

c. Pour vérifier la version du SDK installé, allez dans le menu *Window* > *Android SDK Manager* et lancez le gestionnaire du SDK. Dans le gestionnaire (fig.2.1) vous verrez la version du SDK installé (avec les mises jour disponibles) et aussi la version de l'API (*Application Programming Interface*) installée et la version du OS pour laquelle elle vous permettra de développer. Installez les paquets proposés par défaut.

Si vous voulez développer pour des versions ANDROID plus anciennes il faut installer les versions API correspondantes.

Configuration d'un émulateur

Un émulateur permet de reproduire le comportement d'un appareil réel d'une façon virtuelle. L'utilisation d'un émulateur nous évite d'avoir à charger à chaque fois l'application dans un appareil pour la tester. On pourra ainsi lancer l'application dans l'IDE et elle s'exécutera sur un appareil virtuel appelé *Android Virtual Device* AVD qui émule le comportement d'un téléphone, une tablette ou autre.
ECLIPSE ne propose pas d'émulateur par défaut, avant de commencer à créer notre application il faut en configurer un.

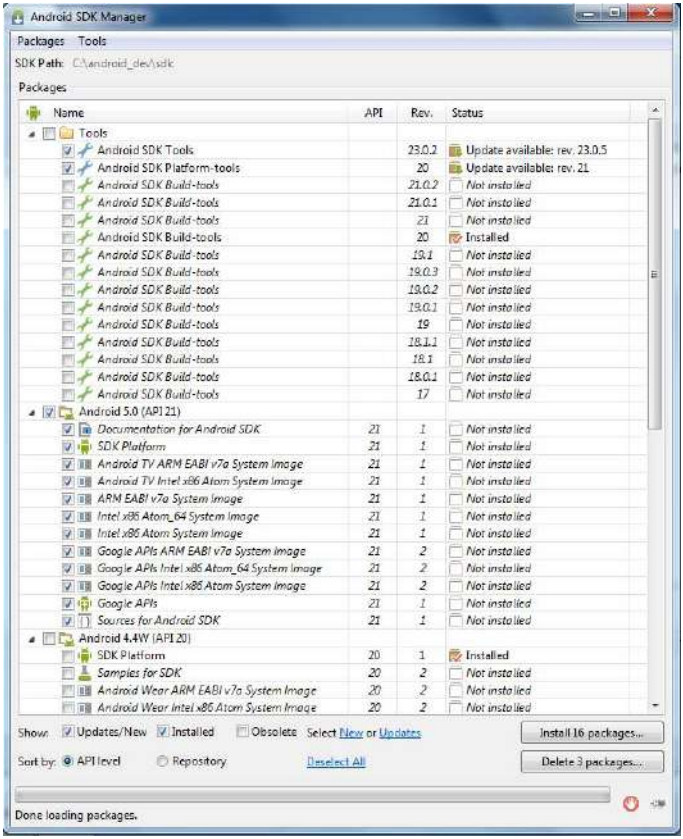


Figure 2.1 – SDK Manager

Dans cet exemple, il existe une mise à jour disponible pour le SDK. L'API installée est la version 20 qui permet un développement pour ANDROID 4.4, mais il existe une API plus récente pour ANDROID 5.0.

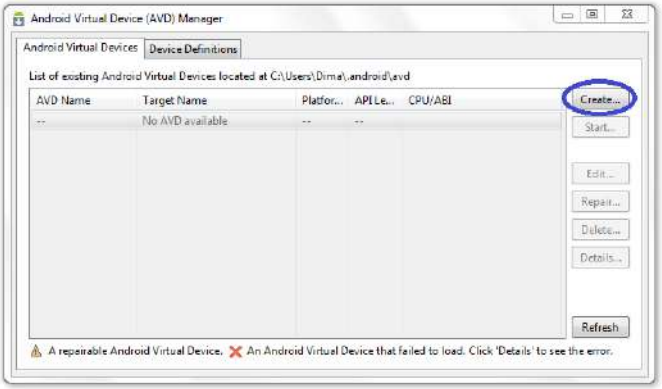
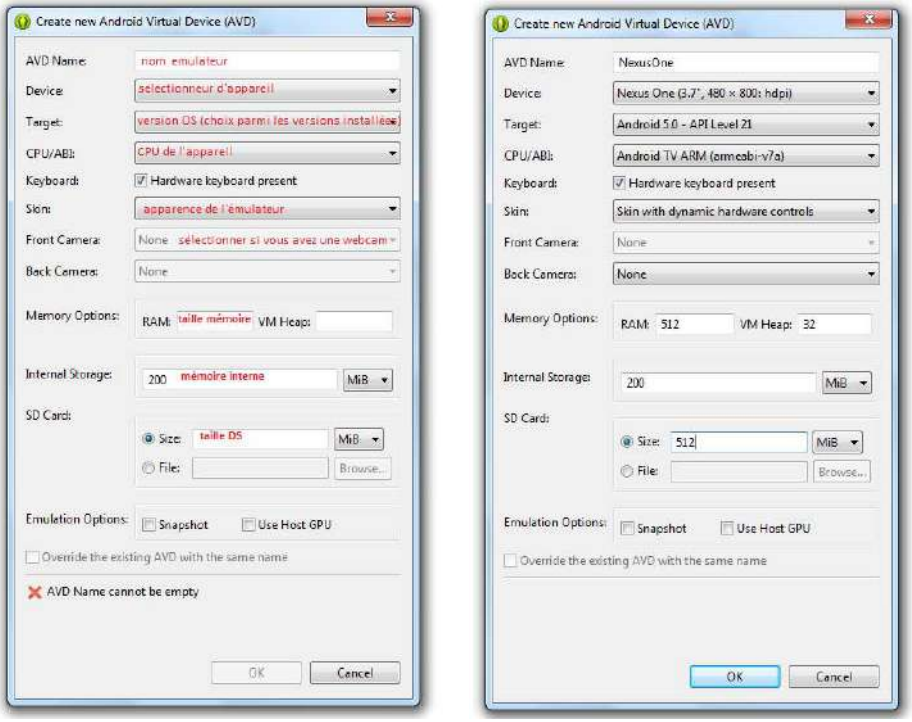


Figure 2.2 – Android Virtual Device Manager

Allez dans le menu *Window > Android Virtual Device Manager*, une fois le gestionnaire ouvert cliquez sur le bouton *Create* (fig. 2.2). Une fenêtre de configuration s'affiche (fig. 2.3a). On propose de configurer un émulateur *Nexus One* avec les paramètres indiqués (fig.2.3b).
Notez qu'à la création de l'appareil sa résolution vous est signalée. Dans cet exemple l'appareil a une résolution 480x800 qui correspond à hdpi (*high density dots per inch*). Ceci est important à noter pour l'intégration d'images dans l'application.

Notez que pour certains émulateurs proposés le processeur n'est pas installé par défaut, pour pouvoir les créer il faut installer un processeur adapté dans le *SDK Manager*.



(a) Fenêtre de création AVD **(b)** Création d'un appareil Nexus One

Figure 2.3 – Création d'un appareil virtuel

3 Notre première application Android

Création d'un projet et d'une application "Hello World"

a. Dans le menu *File > New*, sélectionnez *Android Application Project* , et renseignez les informations comme dans la figure 3.1

Application name : c'est le nom qui va apparaitre dans la liste des applications sur l'appareil et dans le PLAY STORE.

Project name : c'est le nom utilisé par ECLIPSE (typiquement le même que celui de l'application).

Package name : il est utilisé comme identifiant de l'application, il permet de considérer différentes versions d'une application comme étant une même application.

Minimum required SDK : c'est la version ANDROID la plus ancienne sur laquelle l'application peut tourner. Il faut éviter de remonter trop en arrière ça réduirait les fonctionnalités que vous pourriez donner à votre application.

Target SDK : c'est la version pour laquelle l'application est développée et testée. Typiquement la dernière version API que vous avez installée. [1]

Compile with : c'est la version d'API à utiliser pour la compilation. Typiquement la dernière version du SDK installée.

Theme : c'est l'apparence par défaut qu'aura votre application.

1. Ce tutoriel a été réalisé avec la version 4.4.2

b. Cliquez sur *Next* et laissez les choix par défaut. Vous pouvez éventuellement modifier l'emplacement de votre projet en décochant *Create Project in Workspace* et parcourir le disque pour sélectionner un autre dossier.

c. Cliquez sur *Next*. La fenêtre suivante vous propose de définir une icône pour votre application. Nous laisserons l'icône proposée par défaut. Vous pourrez ultérieurement créer votre propre icône pour vos applications. Remarquez que l'image doit être proposée avec différentes résolutions pour s'adapter aux différents appareils.

d. Cliquez sur *Next*. Nous arrivons à la création d'une activité (un écran avec une interface graphique). Sélectionnez *Blank Activity* (fig. 3.2) et cliquez *Next*.

e. Selon la version de l'ADT que vous avez, vous verrez soit la fenêtre de la figure 3.3a ou celle de la figure 3.3b. La dernière version impose l'utilisation de fragments. Chaque activité dispose d'un *layout* qui définit la façon dont les composants seront disposés sur l'écran. Une activité peut être divisée en portions (ou fragments) chacune ayant son propre *layout*. La notion de fragment a été introduite pour favoriser la ré-utilisabilité de morceaux d'activité (un fragment peut être définit une fois et réutilisé dans plusieurs activités). Renseignez les champs comme indiqué dans la figure.

f. Cliquez sur *Finish*, le projet est crée.

⚠ **Si vous créez un fragment ce sera le fichier *fragment_principale.xml* que vous devriez modifier dans la suite du tutoriel sinon vous modifierez le fichier *activite_principale.xml*.**

Exécution de l'application

Sur l'émulateur

Appuyez sur le bouton d'exécution (fig.3.4) et sélectionnez *Android Application* dans la fenêtre qui s'affiche. L'émulateur se lance, ça peut prendre quelques minutes soyez patients. Rassurez-vous, vous n'aurez pas à le relancer à chaque fois que vous compilez votre projet, laissez-le ouvert et à chaque fois que vous compilez et relancez votre application, elle sera rechargée dans l'émulateur en cours.

Figure 3.1 – Création d'un projet

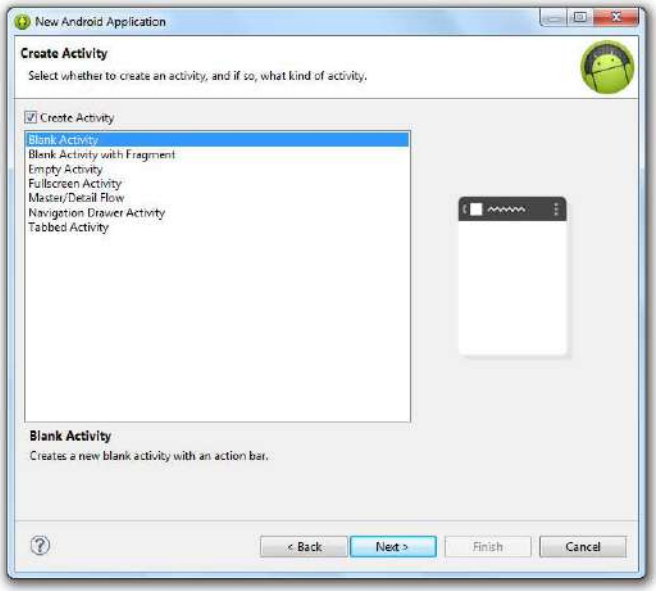
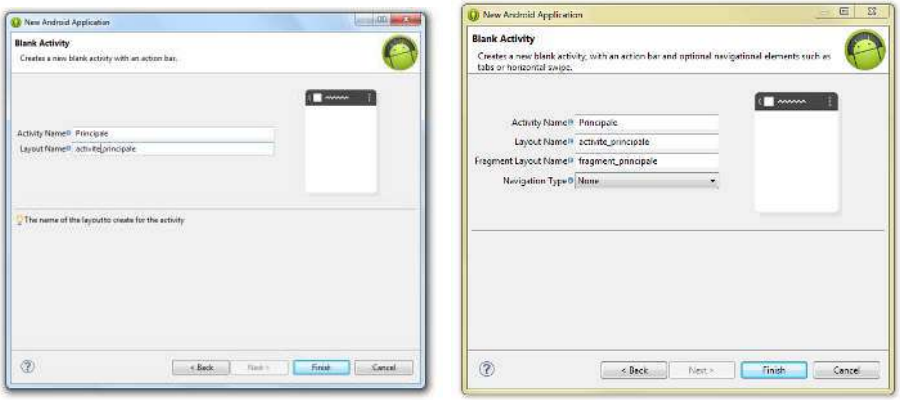


Figure 3.2 – Créer une activité



(a) Création d'activité sans fragment **(b)** Création d'activité avec fragment

Figure 3.3 – Nouvelle activité

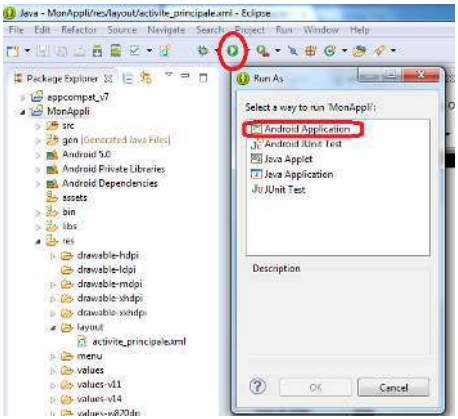


Figure 3.4 – Exécution de l'application

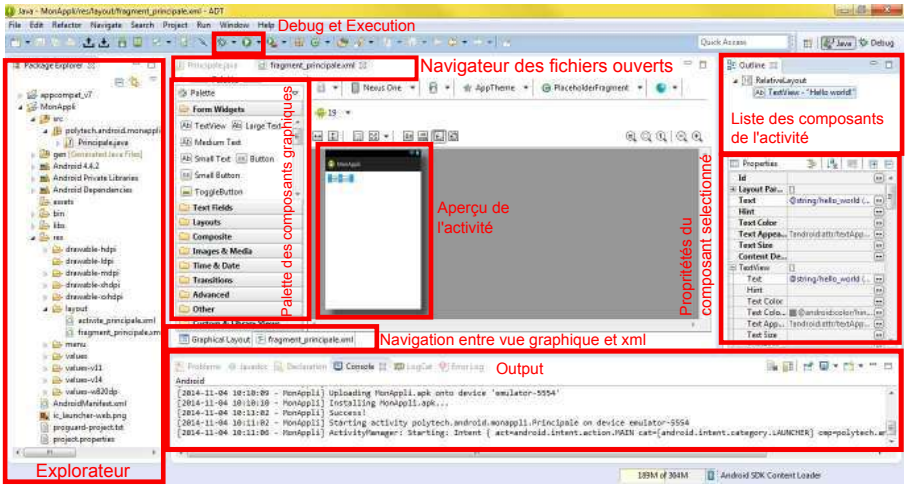


Figure 3.5 – Aperçu de l'interface ECLIPSE

Sur un appareil réel

Connectez l'appareil par câble USB à l'ordinateur et installez le pilote si nécessaire. Activez l'option de débogage USB sur votre appareil (en général sous *Settings > Applications > Development*). Lancez l'application depuis ECLIPSE comme précédemment. ECLIPSE charge l'application sur votre appareil et la lance.

Une fois que votre application est compilée, un fichier *MonAppli.apk* est créé dans le dossier *bin* de votre répertoire de travail. C'est l'exécutable de votre application. C'est ce fichier que vous devez déployer pour distribuer votre application. Le contenu de ce fichier peut être inspecté à l'aide de n'importe quel logiciel standard de compression/-décompression de fichiers.

Se repérer dans le projet

La figure 3.5 montre les principaux éléments de l'interface ECLIPSE.

Tout projet ANDROID doit respecter une hiérarchie bien précise qui permettra au compilateur de retrouver les différents éléments et ressources lors de la génération de l'application. Cette hiérarchie favorise la modularité des applications ANDROID. A la création du projet, ECLIPSE crée automatiquement des dossiers pour contenir

les fichiers de code Java, les fichiers XML, et les fichiers multimédias. L'explorateur de projet vous permettra de naviguer dans ces dossiers.
Les dossiers que nous utiliserons le plus sont *src* et *res*. Le premier contient le code Java qui définit le comportement de l'application et le second comporte des sous dossiers où sont stockés les ressources qui définissent l'interface de l'application (l'apparence).

La séparation entre fonctionnalité et apparence est un point essentiel de la philosophie ANDROID.

Le code de la classe principale de l'application (*Principale.java*) est situé dans le sous dossier *polytech.android.monappli* de *src*. Vous trouverez en annexe une brève explication du code qui y est généré par défaut. C'est dans le dossier *src* que seront enregistrées toutes les classes que nous allons créer dans ce projet.
Par ailleurs, tout ce qui touche à l'interface utilisateur sera intégré dans les sous dossiers de *res*, dont voici une brève description :

layout regroupe les fichiers XML qui définissent la disposition des composants sur l'écran. Il contient déjà, dès la création du projet, le *layout* de l'activité principale que nous avons créée.

drawable-**** contient tout élément qui peut être dessiné sur l'écran : images (en PNG de préférence), formes, animations, transitions, icône, etc.. Cinq dossiers *drawable* permettent aux développeurs de proposer des éléments graphiques pour tout genre d'appareil ANDROID en fonction de sa résolution. En populant correctement ces dossiers on peut ainsi créer des applications avec une interface qui s'adapte à chaque résolution d'écran avec un seul fichier .apk.

ldpi	low-resolution dots per inch. Pour des images destinées à des écrans de basse résolution (~120dpi)
mdpi	pour des écrans de moyenne resolution (~160dpi)
hdpi	pour des écrans de haute résolution (~240dpi)
xhdpi	pour des écrans ayant une extra haute résolution (~320dpi)
xxhdpi	pour des écrans ayant une extra extra haute résolution (~480dpi).

menu contient les fichiers XML définissant les menus

values contient les fichiers XML qui définissent des valeurs constantes (des chaines de caractères, des dimensions, des couleurs, des styles etc.)

Dans le dossier *gen* vous verrez du code java généré automatiquement par ECLIPSE. Nous nous intéresserons particulièrement au fichier *R.java* dans le package *polytech.android.monappli*. Ce fichier définit une classe **R** dans laquelle sont définis les identifiants des ressources de l'application. A chaque fois que vous rajouterez une ressource à votre application un identifiant sera généré automatiquement dans cette classe vous permettant par la suite de pouvoir le référencer pour l'utiliser dans votre code [2].

Vous trouverez également sur la racine du projet un fichier nommé *AndroidManifest.xml*. Ce fichier est obligatoire dans tout projet ANDROID, et doit toujours avoir ce même nom. Ce fichier permet au système de reconnaitre l'application.

Modification de l'interface utilisateur

Pour l'instant notre application ne fait qu'afficher un message sur l'écran, dans cette section nous allons modifier l'interface pour y mettre un champ de saisie et un bouton.

Une interface utilisateur est en général constituée de ce qu'on appelle des `ViewGroups` qui contiennent des objets de type `View` ainsi que d'autres `ViewGroups`. Un `View` est un composant, tel un bouton ou un champ de texte, et les `ViewGroups` sont des conteneurs qui définissent une disposition des composants (`Views`) qui y sont placés. `ViewGroup` définit la classe de base des différents *layouts*.

Comprendre le *layout*

La disposition de notre interface est définie dans le fichier *fragment_principale.xml* situé dans le dossier *layout* de *res*. (ou bien le fichier *activite_principale.xml* si vous n'avez pas définit de fragment à la création de votre projet). Ouvrez ce fichier.

2. A l'intérieur de classe `R` sont définies plusieurs classes, dites nichées, telles que `string`, `drawable`, `layout`, `menu`, `id`, etc. Une classe nichée est membre de la classe qui la contient. On a recours à ce genre de classe en général lorsqu'on veut définir une classe qui n'est utilisée qu'à l'intérieur d'une autre classe. Si on la déclare privée elle ne sera visible qu'a l'intérieur de la classe qui l'a définie. Par ailleurs cette dernière peut également accéder aux attributs privés de la classe nichée. C'est une façon d'améliorer la lisibilité du code en regroupant les fonctionnalités qui vont ensemble. Dans notre cas toutes les classes nichées dans `R` sont publiques, donc accessibles depuis l'extérieur, mais comme elles sont membres de la classe `R`, pour y accéder, il faut passer par `R`. On utilisera des notations telles que `R.string` puisque ces classes sont statiques.

La première balise que vous retrouverez est `<RelativeLayout>` qui définit le type du conteneur qui compose l'interface, il impose la façon dont les composants seront disposés. Plusieurs types de conteneurs existent, les plus communs sont **`RelativeLayout`**, **`LinearLayout`**, **`TableLayout`**, **`GridView`**, **`ListView`**. L'utilisation d'un **`RelativeLayout`**, par exemple, implique que les composants seront placés selon des positions relatives les uns par rapport aux autres. Un **`LinearLayout`** implique une disposition linéaire verticale ou horizontale, un **`GridView`** permet la disposition des éléments selon une grille qui peut défiler, etc.
A l'intérieur de la balise `<RelativeLayout>` vous verrez un ensemble d'attributs définis selon le format

`plateforme:caractéristique="valeur"`

Par exemple le premier attribut `xmlns:android` précise où sont définis les balises ANDROID utilisées dans ce fichier.
La balise `<TextView>`, fille de la balise `<RelativeLayout>`, définit un composant texte qui sera placé sur le *layout*. En effet, c'est sur ce composant là qu'on écrit le "Hello World" qu'affiche notre application. Cette chaine de caractère est définie par l'attribut `android:text`. La notation `"@string/hello_world"` fait référence à une chaine de caractère qui s'appelle `hello_world` et qui est définie dans le fichier *strings.xml* (dans le dossier *values*).

Modifier le type de *layout*

Nous allons maintenant modifier le type du *layout* pour le transformer en **`LinearLayout`**. La figure 3.6 trace la dérivation de la classe **`LinearLayout`**. Nous rajouterons ensuite nos composants sur ce *layout* dans une disposition linéaire.

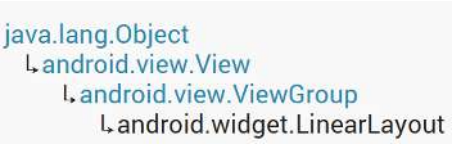


Figure 3.6 – Hiérarchie de LinearLayout
Les layouts sont des ViewGroup qui sont eux mêmes des View [1]

Dans le fichier *fragment_principale.xml*

- ▷ supprimez l'élément `<TextView>`
- ▷ remplacez l'élément `<RelativeLayout>` par `<LinearLayout>`
- ▷ rajoutez l'attribut `android:orientation` et mettre sa valeur à *"horizontal"*

Le code dans le fichier devient ainsi

```
<LinearLayout
        xmlns:android="http://schemas.android.com/apk/
            res/android"
        xmlns:tools="http://schemas.android.com/tools"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="horizontal" >
</LinearLayout>
```

Rajouter d'un champ de saisie

- ▷ Rajoutez un élément `<EditText>` dans le `<LinearLayout>` tel que

```
<EditText
        android:id="@+id/chp_saisie"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:hint="@string/str_chp_saisie" />
```

Nous avons ainsi placé un champ de saisie avec les attributs suivants :

android :id permet de donner un identifiant unique à ce **View** qu'on utilisera pour référencer cet objet à l'intérieur de notre code.
Le symbol **@** est nécessaire pour faire référence à un objet ressource à partir d'un fichier XML. *id* est le type de ressource et *chp_saisie* est le nom qu'on donne à notre ressource. Le symbole **+** est utilisé pour définir un ID pour la première fois. Il indique aux outils du SDK qu'il faudrait générer un ID dans le fichier *R.java* pour référencer cet objet. Un attribut `public static final chp_saisie` sera défini dans la classe **id**.Le symbole **+** ne doit être utilisé qu'une seule fois au moment où on déclare la ressource pour la première fois. Par la suite si on veut faire référence à cet élément, à partir d'un XML, il suffira d'écrire *@id/chp_saisie*.

android :layout_width permet de spécifier la largeur de élément.
"*wrap_content*" signifie que le **View** doit être aussi large que nécessaire pour s'adapter à la taille de son contenu. Si en revanche on précise "*match_parent*" comme on l'avait fait pour le **LinearLayout**, dans ce cas le **EditText** occuperait toute la largeur de l'écran puisque sa largeur sera celle de son parent c-à-d le **LinearLayout**

android :layout_height idem que pour le `layout_width` mais pour la hauteur

android :hint précise le texte par défaut à afficher dans le champ de saisie quand il est vide. Nous aurions pu préciser directement la chaine de caractère ici codée en dur, mais on préfère utiliser plutôt une ressource qu'on définira dans *strings.xml* . Noter que l'utilisation de + ici n'est pas nécessaire parce qu'on fait référence à une ressource concrète (qu'on définira dans le fichier xml) et non pas à un identifiant que le SDK doit créer dans la classe R.

Privilégiez toujours l'utilisation des ressources *strings* plutôt que des chaines de caractères codées en dur. Cela permet de regrouper tout le texte de votre interface dans un seul endroit pour simplifier la recherche et la mise à jour du texte, de plus ceci est indispensable pour que votre application puisse être multilingue. l'IDE vous affichera un avertissement en cas de non respect de cette recommandation.

Après la modification du code que nous venons de faire, quand vous sauvegarderez le fichier, un message d'erreur vous indiquera que l'identifiant *str_chp_saisie* n'est pas connu. Nous allons donc le définir.

- ▷ Ouvrez le fichier *strings.xml* qui se trouve dans *res>values*
- ▷ Rajoutez une nouvelle *string* nommée *str_chp_saisie* et dont la valeur est "Entrer un texte"
- ▷ Vous pouvez éventuellement supprimer la ligne qui définit "*hello_world*"

Votre fichier *strings.xml* ressemblera donc à ceci

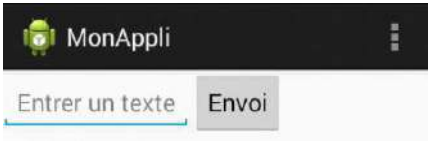
```
<?xml version="1.0" encoding="utf-8"?>
<resources>
        <string name="app_name">MonAppli</string>
        <string name="str_chp_saisie">Entrer un texte</
            string>
```

Figure 3.7 – Premier test de l'application modifiée

```
        <string name="action_settings">Settings</string>
</resources>
```

- ▷ Une fois que vos modifications sont sauvegardées vous remarquerez la création de deux attributs dans le fichier *R.java*.
  - Un attribut constant nommé `chp_saisie` dans la classe **`id`**. C'est un numéro unique qui identifie l'élément **`EditText`** que nous venons de rajouter. Cet identifiant nous permettra de manipuler l'élément à partir du code.
  - Un attribut constant nommé `str_chp_saisie` dans la classe **`string`**. Il fait référence à la chaine de caractère et nous permettra de l'utiliser dans le code.

Lancez l'application, l'émulateur affichera un écran tel que dans la figure 3.7. Tapez un texte et remarquez comment la taille du champ de saisie s'adapte à la longueur du texte.



(a) Disposition par défaut **(b)** Le EditText a un poids de 1

Figure 3.8 – Champ de saisie et bouton

Rajouter un bouton

- ▷ Dans le fichier *strings.xml* rajoutez une chaine de caractère qui s'appelle *"btn_envoyer"* et qui vaut Envoi.
- ▷ Dans le fichier du *layout* rajoutez un élément `<Button>` tel que

```
<Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/btn_envoyer" />
```

Lancez l'application. Vous devez voir un bouton à coté du champ de saisie (fig.3.8a). Si vous souhaitez que votre champ de saisie occupe toute la largeur de l'écran qui reste après le positionnement du bouton il faut spécifier un poids de 1 au `EditText` et une largeur de 0.

```
<EditText       ...
        android:layout_weight="1"
        android:layout_width="0dp"
        ... />
```

Répondre aux évènements

Pour répondre à un appui sur le bouton il suffit de définir un attribut `android:onClick` pour le bouton en lui donnant comme valeur le nom de la méthode qui devrait être appelée quand le bouton est appuyé, et d'implémenter cette méthode de réponse dans la classe principale de l'activité.

- ▷ Dans le fichier xml du *layout*, rajoutez l'attribut `android:onClick` à l'élément bouton tel que :

```
<Button
        ...
        android:onClick="envoiMessage"
        ... />
```

▷ Dans la classe **Principale** rajoutez la méthode

```
/** Méthode appelée quand on appuie sur Envoi */
public void envoiMessage (View view){
        // le code de traitement ira ici
}
```

Il faut absolument respecter cette signature pour la méthode afin que le système puisse l'associer au nom donné par `android:onClick`. Le paramètre `view` est rempli par le système et correspond à l'élément qui a généré l'évènement (le bouton Envoi dans notre cas).

Avant d'aller plus loin dans le traitement, vous pouvez déjà tester si l'appel s'effectue correctement quand le bouton est appuyé. Pour cela, mettez un point d'arrêt à l'intérieur de la méthode `envoiMessage()` et lancez l'application en mode *Debug* (fig. 3.5).

▷ Dans l'émulateur appuyez sur le bouton *Envoi* et vérifiez que le programme entre bien dans la méthode `envoiMessage()`.

▷ Arrêtez le débogage et revenez en mode Java en cliquant sur le bouton correspondant en haut à droite de l'IDE.

Créer et lancer une autre activité

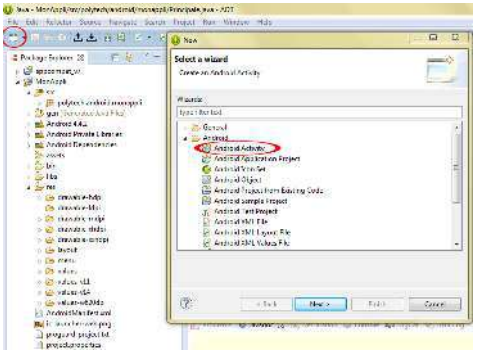
Dans la suite, nous allons répondre à l'appui du bouton en lançant une deuxième activité qui affichera le texte qu'on aurait tapé dans le champ de saisie de l'activité principale.

Création d'une activité

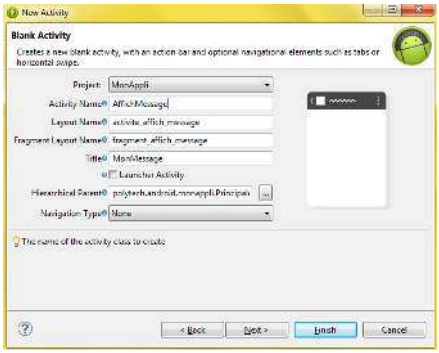
▷ Cliquez sur le bouton *new* de la barre d'outil ECLIPSE et sélectionner *Android Activity*, puis cliquez sur *Next* (fig. 3.9a)

▷ Sélectionnez *Blank Activity* et appuyez sur *Next*.

▷ Définissez les paramètres de l'activité comme dans la figure 3.9b



(a) *Wizard* de création



(b) Définition de l'activité

Figure 3.9 – Création d'une nouvelle activité

En plus des champs déjà vus au moment de la création de l'activité principale, vous remarquez que pour notre nouvelle activité il faut définir une activité parent. Ceci est utile pour implémenter le comportement par défaut du bouton retour.
Une fois l'activité créée ECLIPSE génère :

- un fichier *AffichMessage.java* contenant le code la classe
- les fichiers xml correspondant au *layout* de la nouvelle activité
- un élément `<activity>` dans le fichier *AndroidManifest.xml* et affecte ses attributs avec les valeurs que nous avons précisées lors de la création de l'activité

```
<activity
        android:name="polytech.android.monappli.
            AffichMessage"
        android:label="@string/
            title_activity_affich_message"
        android:parentActivityName="polytech.android.
            monappli.Principale" >
        <meta-data
                android:name="android.support.
                    PARENT_ACTIVITY"
                android:value="polytech.android.monappli
                    .Principale" />
</activity>
```

- une chaine de caractère dans le fichier *strings.xml* correspondant au titre de notre nouvelle activité.

```
<resources>
        ...
        <string name="title_activity_affich_message">
            MonMessage</string>
        ...
</resources>
```

Lancement de l'activité

Pour faire communiquer les deux activés (l'activité principale et celle que nous venons de créer) il faut passer par un *Intent*. Ce dernier représente l'intention de faire quelque chose, et permet à l'activité principale de lancer l'activité d'affichage. Dans la méthode `envoiMessage()` de la classe **`Principale`** :

- Créez une intention,

  ```
  Intent intent = new Intent(this, AffichMessage.class);
  ```

  sans oublier d'importer la classe

  ```
  import android.content.Intent;
  ```

  A la construction de l'objet `intent`, nous précisons deux arguments : le premier est un objet de type **`Context`** qui fait référence à l'application qui crée l'intention et le deuxième précise le nom (de la classe) de l'activité qui crée l'intention. Comme le **`Intent`** peut être utilisé pour faire communiquer deux applications, il ne suffit pas de préciser uniquement le nom de l'activité qui le crée mais il faut également définir l'application qui l'invoque.
- lancez l'activité

  ```
  startActivity(intent);
  ```

Exécutez l'application et appuyez sur le bouton Envoi. La nouvelle activité se lance et affiche "Hello World !". En effet c'est le comportement par défaut qui a été défini par le *layout* dans *fragment_affich_message.xml*. Notez que le bouton de retour est déjà fonctionnel et permet de remonter à l'activité principale. Ceci est dû au

fait que nous l'avons indiqué comme activité parent au moment de la création de notre activité d'affichage.

Communication entre les activités

Envoyer un message

Si nous souhaitons que le texte tapé dans l'activité principale soit affiché dans l'activité d'affichage, il faut faire communiquer les deux activités de sorte à ce que la première envoie le texte à la deuxième. Ceci s'effectue en utilisant le même *Intent* qui a servi pour le lancement de l'activité. En effet une intention peut aussi transporter un paquet de données.
Modifier la méthode `envoiMessage()` pour qu'elle contienne le code ci-dessous, sans oublier d'importer les classes nécessaires (ECLIPSE vous les proposera)

```
public void envoiMessage (View view){
        Intent intent = new Intent(this, AffichMessage.class);
        EditText editText = (EditText) findViewById(R.id.
            chp_saisie);
        String message = editText.getText().toString();
        intent.putExtra(MESSAGE_SUPP, message);
        startActivity(intent);
}
```

La méthode `findViewById()` permet de retrouver un objet de type **`View`** à partir de son identifiant. Ici elle renvoie l'objet correspondant à `chp_saisie` qu'on *cast* en **`EditText`**. La variable `editText` contient désormais l'objet champ de saisie que nous avions posé sur l'interface principale. Nous récupérons ensuite la chaine de caractère que contient ce champ en appelant `editText.getText().toString()`. Cette chaine est ensuite stockée dans la variable `message` qui est passée en paramètre à la méthode `putExtra()` de l'objet `intent` afin de charger l'intention avec ce message. Afin que l'activité d'affichage puisse identifier et récupérer les données supplémentaires transportées par l'intention il faut définir une clé pour ces données moyennant une constante publique. Nous définissons donc la constante `MESSAGE_SUPP` dans la classe **`Principale`**.

```
public class Principale extends ActionBarActivity {
        public final static String MESSAGE_SUPP = "polytech.
            android.monappli.MESSAGE";
```

```
        ...
```

En général on définit ce genre de clé en utilisant le nom de notre *package* comme préfixe. Ceci garantit l'unicité des clés dans le cas où notre application interagit avec d'autres.

Récupérer et afficher le message

Arrivés à ce point, nous avons fait en sorte à ce que l'activité principale envoie un message à l'activité d'affichage. Il nous reste maintenant à récupérer ce message dans **AffichMessage**. Pour cela il suffit de rajouter le code ci- dessous dans la méthode `onCreate()` de la classe **AffichMessage** . Cette méthode est appelée à la création de l'activité.

```
Intent intent = getIntent();
String message = intent.getStringExtra(Principale.MESSAGE_SUPP
    );
```

Ensuite, pour afficher le message nous allons créer un **TextView**, lui affecter le message puis le rajouter à un *layout* qu'on passera à `setContentView()`.
Dans les sections précédentes nous avons appris à créer et rajouter des composants à partir du fichier xml, ici nous le faisons dans le code.
Voici le code complet de la méthode `onCreate()`

```
protected void onCreate(Bundle savedInstanceState) {
        /*appeler onCreate de la classe mère*/
        super.onCreate(savedInstanceState);
        /*récupérer le message transporté par l'intention*/
        Intent intent = getIntent();
        String message = intent.getStringExtra(Principale.
            MESSAGE_SUPP);
        /*créer le textView*/
        TextView textView = new TextView(this);
        textView.setTextSize(40);
        textView.setText(message);
        /*créer un layout tabulaire*/
        TableLayout monLayout = new TableLayout(this);
        /*rajouter le textView au layout*/
        monLayout.addView(textView,0);
```

```
        /*définir monLayout comme étant le layout de l'
            activité*/
        setContentView(monLayout);
}
```

Remarquez qu'ayant supprimé la ligne

```
setContentView(R.layout.activite_affich_message);
```

qui s'y trouvait par défaut nous n'utilisons plus le fichier *activite_affich_message.xml* pour définir le *layout* de l'activité, mais nous définissons le contenu de l'écran dans le code.
On aurait pu simplement définir le `textView` comme *layout* sans passer par `monLayout` et ce en écrivant

```
setContentView(textView);
```

dans ce cas l'activité ne contiendrait que le message. Cependant nous avons préféré créer un *layout* tabulaire qui nous servira pour la suite de ce tutoriel.

Exécutez l'application, entrez un texte dans le champ de saisie et appuyez sur le bouton Envoi. Votre texte devrait apparaitre sur l'écran suivant.

Créer des animations

Il existe deux façons de faire des animations avec ANDROID. La première est basée sur l'affichage d'une série d'images qui constituent l'animation. C'est la méthode la plus simple. La deuxième méthode, quant à elle, repose sur le calcul de transformations mathématiques pour animer graphiquement un objet. Alors que la première, basée sur les images, est gourmande en terme d'espace de stockage (taille de l'application) la deuxième, dite vectorielle, nécessite plus de ressources de calculs. Dans le cadre de ce tutoriel nous nous limiterons à l'introduction de la méthode la plus simple, vous pouvez vous référez à [2] pour une explication des animations vectorielles.
Pour commencer il faut créer, pour chaque résolution, une série d'images qui composera l'animation. Joint à ce tutoriel nous proposons 12 images définissant une séquence de rotation de la terre pour la résolution hdpi.

- ▷ Copiez les images dans le dossier *drawable-hdpi*

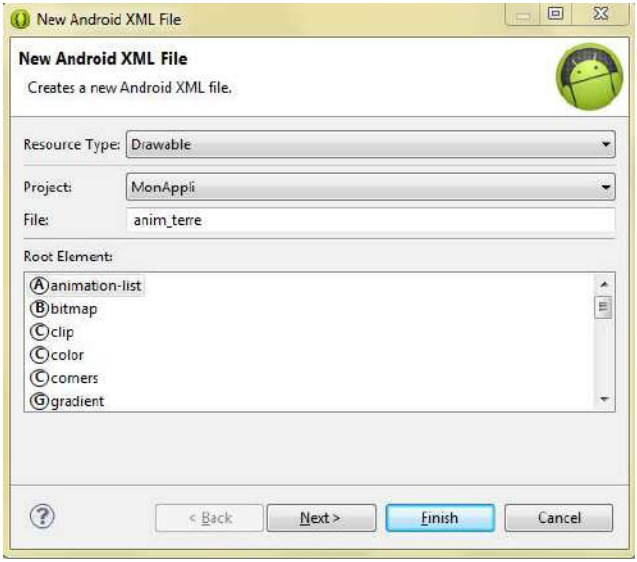


Figure 3.10 – Nouveau xml pour définir une animation

- ▷ A l'aide d'un clic droit sur ce dossier dans l'explorateur ECLIPSE sélectionnez *New>Android XML File*
- ▷ Sélectionnez *Drawable* comme ressource, dans le projet *MonAppli* et nommez-le *anim_terre* avec *animation-list* comme élément de base et cliquer *Finish* (fig. 3.10)
- ▷ ECLIPSE génère alors un fichier xml contenant un squelette d'animation avec la balise `<animation-list>`
- ▷ Modifiez le fichier de sorte à rajouter un paramètre `android:oneshot` en le mettant à *"false"* pour que l'animation boucle indéfiniment et remplissez l'animation par des items définissant les images.

```
<?xml version="1.0" encoding="utf-8"?>
<animation-list xmlns:android="http://schemas.android.
    com/apk/res/android"  android:oneshot="false">
        <item android:drawable="@drawable/image1"
            android:duration="250" />
        <item android:drawable="@drawable/image2"
            android:duration="250" />
        <item android:drawable="@drawable/image3"
```

```
            android:duration="250"/>
        <item android:drawable="@drawable/image4"
            android:duration="250" />
        <item android:drawable="@drawable/image5"
            android:duration="250" />
        <item android:drawable="@drawable/image6"
            android:duration="250" />
        <item android:drawable="@drawable/image7"
            android:duration="250" />
        <item android:drawable="@drawable/image8"
            android:duration="250" />
        <item android:drawable="@drawable/image9"
            android:duration="250" />
        <item android:drawable="@drawable/image10"
            android:duration="250" />
        <item android:drawable="@drawable/image11"
            android:duration="250" />
        <item android:drawable="@drawable/image12"
            android:duration="250" />
</animation-list>
```

Pour chacune des images il faut préciser `android:duration` qui définit la durée d'affichage de l'image en ms.
Notez que dès que les fichiers png sont rajoutés au dossier *drawable* des identifiants image1, image2, ... image12 sont automatiquement crées dans la classe **drawable** de *R.java*. Ce sont ces identifiants que nous utilisons ici.
Pour faire apparaitre l'animation il faut rajouter un élément `<ImageView>` dans le *layout* principal.

```
<ImageView
        android:id="@+id/animTerre"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:contentDescription="@string/
            anim_terre_descript"
        android:src="@drawable/anim_terre" />
```

Figure 3.11 – Animation en LinearLayout

L'attribut `android:contentDescription` permet de définir un label pour l'image. ECLIPSE donnera un avertissement si vous ne fournissez pas cette description pour assurer l'accessibilité de votre application à tous les utilisateurs. **N'oubliez pas de définir la chaine de caractère *`anim_terre_descript`* dans le fichier *string.xml***

Exécutez l'application, vous verrez l'animation s'afficher à droite de l'écran à coté du bouton (fig. 3.11). Un `LinearLayout` ne compte qu'une ligne, tous les éléments sont donc disposés sur la même ligne. Si on souhaite placer l'animation sur une ligne à part, il faudrait mettre deux `LinearLayout` ou bien changer le type de *layout*.

Pour changer de *layout* on pourrait bien entendu modifier directement le code xml pour remplacer `<LinearLayout>` par un autre type en adaptant les attributs du *layout*, ainsi que ceux des éléments qui y sont disposés, mais on pourrait également le faire graphiquement.

- ▷ Ouvrez le fichier *fragment_principale.xml*
- ▷ Sélectionnez l'onglet *Graphical layout* (fig.3.5). A l'aide d'un clic droit sur *LinearLayout* sélectionnez *Change Layout* et passez en *RelativeLayout*
- ▷ En cliquant sur *Preview* vous pouvez avoir un aperçu des modifications qu'effectuera l'environnement de développement pour s'adapter au changement de *layout*.
- ▷ Une fois que vous avez modifié le *layout*, vous pouvez maintenant, en mode graphique (glisser-déposer), disposer le `ImageView` en dessous des autres éléments.
- ▷ Réarrangez le champ de saisie et le bouton pour occuper complètement la première ligne. Vous devriez supprimer la ligne concernant l'attribut `android:layout_weight` dans le xml (il ne peut pas s'appliquer à un

Figure 3.12 – Animation en RelativeLayout

`<RelativeLayout>`)

Exécutez l'application, l'activité devrait ressembler à la figure 3.12

Création et contrôle des animations à partir du code

Nous venons de créer une animation à partir des fichiers xml, cependant dans certaines situations il est utile de créer et de contrôler des animations depuis le code Java. Nous allons, dans ce qui suit, refaire la même animation dans l'activité **`AffichMessage`** mais, cette fois, à partir du code de la classe. Nous en profiterons pour apprendre à créer des boutons dans le code Java et à gérer leurs évènements.

- Créez trois données membres de la classe **`AffichMessage`** tels que

```
protected Button playBtn;
protected Button stopBtn;
protected ImageView globe;
```

- Dans la méthode `onCreate()` créez le `globe` et associez-lui l'animation créée précédemment

```
globe = new ImageView(this);
globe.setImageResource(R.drawable.anim_terre);
```

- Toujours dans `onCreate()`, créez les deux boutons et gérez leurs clics

```
/*créer le bouton stop*/
stopBtn = new Button(this);
stopBtn.setText(R.string.btn_stop);
```

```
/*définir et implémenter le callback du Click*/
stopBtn.setOnClickListener(new View.OnClickListener() {
        public void onClick(View v) {
                AnimationDrawable monAnimation =(
                    AnimationDrawable)globe.getDrawable();
                monAnimation.stop();
        }
});

/*créer le bouton play*/
playBtn = new Button(this);
playBtn.setText(R.string.btn_play);
/*définir et implémenter le callback du Click*/
playBtn.setOnClickListener(new View.OnClickListener() {
        public void onClick(View v) {
                AnimationDrawable monAnimation =(
                    AnimationDrawable)globe.getDrawable();
                monAnimation.start();
        }
});
```

Pensez à définir les chaines de caractères *btn_play*, et *btn_stop* dans *string.xml*

▷ Ajoutez `globe`, `play_btn` et`stop_btn` au *layout*

```
monLayout.addView(globe,1);
monLayout.addView(playBtn,2);
monLayout.addView(stopBtn,3);
```

Exécuter et tester le comportement de l'application.

Créer un View personnalisé pour gérer un jeu

La plupart du temps quand il s'agit de concevoir un jeu, il est indispensable de pouvoir dessiner sur l'écran. Dans cette section nous allons créer une classe dérivée de **`View`** que nous personnaliserons pour faire déplacer une image sur l'écran et interagir avec l'utilisateur.

Figure 3.13 – Création de la classe MonViewPerso

Création de la classe MonViewPerso

- ▷ Dans l'explorateur d'ECLIPSE faites un clic droit sur le *package* contenant vos classes (*polytech.android.monappli*) et sélectionnez *New>Class*
- ▷ Nommez-la **`MonViewPerso`** et faites-la hériter de **`View`**(fig.3.13). Vous pouvez naviguer pour retrouver le nom de la classe mère.
- ▷ Cliquez sur *Finish*, ECLIPSE génère un fichier *MonViewPerso.java*

Une erreur s'affiche déjà, il faut définir explicitement un constructeur qui fait appel au constructeur de la classe de base. En effet si nous ne le définissons pas, un constructeur par défaut, sans paramètre, est rajouté implicitement et fait appel au constructeur `View()` sans paramètre, or ce dernier n'existe pas. La classe **`View`** ne définit que des constructeurs avec arguments.
Nous définirons donc le constructeur ci dessous

```
public MonViewPerso(Context context,AttributeSet attrs) {
        super(context,attrs);
        }
```

Afin de définir comment notre vue se dessine sur l'écran il faut implémenter la méthode `onDraw()` qui sera invoquée automatiquement par le système à chaque fois qu'il a besoin d'afficher ou de rafraichir le **`View`**.
Nous définirons d'abord un objet de dessin (une sorte de pinceau) comme attribut de la classe **`MonViewPerso`**

```
Paint p = new Paint();
```

que nous utiliserons dans la méthode `onDraw()` comme suit

```
public void onDraw (Canvas canvas)      {

        /*définir la couleur de l'objet de dessin */
        p.setColor(Color.BLACK);
        /*définir son style en remplissage*/
        p.setStyle(Paint.Style.FILL);
        /*dessiner un rectangle qui occupe la totalité du View*/
        canvas.drawRect(0,0,getWidth(),getHeight(), p);

        /*définir une autre couleur pour dessiner un texte*/
        p.setColor(Color.WHITE);
        /*définir la taille du texte*/
        p.setTextSize(20);
        /*définir le centre du texte comme étant son origine*/
        p.setTextAlign(Paint.Align.CENTER);
        /*dessiner le texte en positionnant son origine au centre du
            View */
        String texte = getResources().getString(R.string.hello_world);
        canvas.drawText(texte, getWidth()/2, getHeight()/2, p);
}
```

Avec le code ci-dessus notre vue consistera en un écran noir au centre duquel on affiche Hello World ! en blanc. L'argument `canvas`, que le système passe à la méthode `onDraw()`, représente la zone de dessin de l'écran.
La méthode

```
public void drawText (String text, float x, float y, Paint paint);
```

dessine l'origine du texte à la position donnée par x et y. L'origine du texte est définie avec

```
public void setTextAlign (Paint.Align align);
```

Notez qu'ici nous avons juste affiché un texte défini dans *strings.h*, mais il est également possible de créer des chaines de caractères formatées pour, par exemple, récupérer des valeurs de variables. Modifiez le code pour définir le `texte` tel que

```
String texte = String.format("%s %d x %d",texte,canvas.
    getWidth(),canvas.getHeight());
```

pour afficher, en plus du Hello World, la résolution de l'écran.
Notez qu'ici nous récupérons les dimensions du `canvas` qui ne sont pas égales à celle du **`View`** que nous avions utilisées dans `drawText()`. En effet la taille de notre vue est inférieure à celle de l'écran.

Création d'une activité qui contiendra la vue personnalisée

Afin d'afficher notre vue il faut l'associer à une activité. Nous commencerons par la création d'une activité que nous nommerons **`MonJeu`**, comme vu précédemment. Nous définirons l'activité **`Prinicpale`** comme activité parent de **`MonJeu`**.
Une fois l'activité créée avec le *Wizard*, ECLISPE la rajoute au *AndroidManifest.xml* et crée le .java associé et les .xml de son *layout*.
Pour lancer cette activité nous rajouterons un bouton sur l'interface principale. Nous avons déjà vu comment rajouter un élément sur un *layout* dans le fichier xml, mais nous pouvons également le faire dans le *Graphical layout*.

- ▷ Ouvrez le fichier *fragment_principale.xml*
- ▷ Sélectionnez l'onglet *Graphical layout*.
- ▷ Dans la palette de composants, sélectionnez un bouton, glissez-le et déposez-le sur l'interface de l'activité.
- ▷ Dans la fenêtre de propriétés définir :
  - Id : @+id/btn_jeu
  - Text : @string/btn_jeu (sans oublier de définir btn_jeu dans strings.xml)
  - On Click : jouer
- ▷ Le fichier *fragment_principale.xml* est mis à jour avec les propriétés que nous venons de définir
- ▷ Dans le code de la classe **`Principale`** implémenter la méthode `jouer()` qui répondra aux appuis sur le bouton

```
public void jouer (View view){
        Intent intent = new Intent(this, MonJeu.class);
        startActivity(intent);
```

Figure 3.14 – Ajout d'un bouton pour lancer le jeu

```
}
```

Si vous testez l'application (fig.3.14), l'appui sur le bouton que nous venons de rajouter lancera l'activité **MonJeu**. Cependant celle ci n'est pas encore associée à notre vue et ne fait qu'afficher un message par défaut. Il nous reste donc à rajouter un élément `<MonViewPerso>` sur le *layout* de l'activité.

- ▷ Modifier le fichier *fragment_mon_jeu.xml* en supprimant l'élément `<TextView>` et le remplaçant par un élément `<MonViewPerso>` tel que

```
<polytech.android.monappli.MonViewPerso
        android:id="@+id/maVue"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:layout_alignParentLeft="true"
        android:layout_alignParentRight="true"
        android:layout_alignParentTop="true" />
```

Vous pouvez également faire cela de façon graphique dans le *Graphical layout*. En effet, notre vue personnalisée apparait maintenant dans la palette et nous pouvons la rajouter sur l'activité avec un glisser-déposer.

Interaction avec l'utilisateur

Dans la suite nous souhaitons afficher une image sur la vue personnalisée et permettre à l'utilisateur de la déplacer en la touchant.

- ▷ Rajoutez l'image au projet. Joint à ce tutoriel nous proposons des images de planètes pour différentes résolutions qui sont utilisées dans [2]. Placez les images dans les dossiers *drawable* correspondants.
- ▷ Si besoin, rafraichissez les dossiers *drawable* dans l'explorateur ECLIPSE pour faire apparaitre ces nouvelles images dans l'arborescence. Vérifiez la création automatique d'identifiant pour chacune de ces images dans la classe **drawable** de *R.java*.
- ▷ Dans la classe **MonViewPerso** créez des attributs pour contenir l'image, sa taille et sa position

```
Bitmap planet =null;
float xOri=0,yOri=0;
int largImage,hautImage;
```

- ▷ Dans le constructeur de **MonViewPerso**, récupérez l'image et ses dimensions

```
BitmapDrawable d = (BitmapDrawable) getResources().
    getDrawable(R.drawable.earth);
planet = d.getBitmap();
largImage=planet.getWidth();
hautImage=planet.getHeight();
```

- ▷ Dans la méthode `onDraw()` dessinez l'image (après `drawRect()`)

```
canvas.drawBitmap(planet, xOri, yOri, p);
```

Exécutez l'application, vous devriez voir l'image s'afficher en haut à gauche de votre vue (fig.3.15). Afin de réagir au toucher de l'utilisateur il faut implémenter la méthode `onTouchEvent()` qui est appelée quand l'utilisateur touche l'écran.

```
public boolean onTouchEvent (MotionEvent event){
    int action = event.getAction();
    switch (action) {
        case MotionEvent.ACTION_DOWN: /*on a touché l'ecran*/
            /*calculer la distance de la touche à l'origine de l'image
                */
            deltaX = event.getX() - xOri;
```

```
            deltaY = event.getY() - yOri;
            /*tester si on a touché la planète*/
            if (deltaX>= 0 && deltaX <= largImage
                && deltaY>= 0 && deltaY <= hautImage)
            /*on a touché l'image, permettre le movement*/
                move = true;
            break;

        case MotionEvent.ACTION_MOVE: /* le doigt bouge sur l'écran*/
            if(move){
                /*si le movement est permis, mettre à jour les
                    coordonnées de l'image*/
                xOri = event.getX() - deltaX;
                yOri = event.getY()- deltaY;
            }
            break;

        case MotionEvent.ACTION_UP: /*le doigt a quitté l'écran*/
            move = false;
            break;
    }
    /*forcer un repaint pour rafraichir l'affichage*/
    invalidate ();

    return true;
}
```

Les attributs `deltaX`, `deltaY` et `move` sont définis tels que

```
float deltaX=0, deltaY=0;
boolean move=false;
```

Exécutez et testez l'application. On pourrait rajouter des conditions afin d'interdire à la planète de sortir de la vue.

Temporisation

Dans certaines applications on est parfois amené à effectuer une tâche après un certain délai, ou bien périodiquement toutes les x ms. Nous allons, dans la suite, introduire la notion de temporisation que nous utiliserons pour faire déplacer notre planète le long de l'écran.

Figure 3.15 – Activité avec vue personnalisée

Pour ce faire nous passerons par un objet **Handler** qui permet de programmer l'appel d'une méthode après un délai déterminé.

- Dans la classe **MonViewPerso**, définir un attribut de type **Handler** tel que

```
Handler timerHandler = new Handler();
```

- Dans le constructeur poser une tâche à exécuter après 0ms

```
timerHandler.postDelayed(updateTimerThread, 0);
```

`updateTimerThread` est un objet de type **Runnable** que nous définirons dans un instant. Le fait de poster cet objet avec un délai, met l'appel à sa méthode `Run()` dans la queue d'exécution du processus principal pour être exécutée une fois le délai écoulé. Autrement dit, le `postDelayed()` programme l'appel de la méthode `Run()` de `updateTimerThread` dans 0ms.

- Définir un attribut de type **Runnable** tel que

```
private Runnable updateTimerThread = new Runnable() {
        public void run() {
            /*mettre à jour les coordonnées de la planète*/
            xOri++;
            yOri++;
            /*forcer le rafraichissement de l'écran*/
```

```
            invalidate ();
            /*reprogrammer l'objet pour une exécution dans 50ms
                */
            timerHandler.postDelayed(this, 50);
        }
};
```

Avec ce code nous avons défini ce qu'on appelle une classe anonyme. Les classes anonymes en Java permettent de déclarer et instancier une classe en même temps. Elles ressemblent aux classes locales nichées mais, contrairement à ces dernières, elles ne possèdent pas de nom. On les utilise quand on veut utiliser une classe locale une seule fois. On n'a pas besoin de la nommer puisqu'on ne fera plus référence à elle ailleurs.
La définition de la classe s'effectue avec une expression au moment de l'appel du constructeur. Après l'invocation du constructeur on écrit un bloc contenant la définition de la classe. Il suffit de mettre

- ▷ `new` avec le nom de l'interface que la classe doit implémenter, ici **`Runnable`**, ou bien le nom de la classe mère si c'était le cas.
- ▷ suivi des parenthèses avec les paramètres du constructeur de la classe de base. Ici, comme il s'agit d'une interface, il n'y a aucun paramètre. (les interfaces n'ont pas de constructeurs)
- ▷ puis, entre accolades, le corps de la classe en terminant par un ;

Lancez l'application, vous verrez la planète se déplacer en diagonale sur l'écran. Notez qu'il est toujours possible de la déplacer avec le toucher.

La classe **`Handler`** permet aussi la communication entre deux processus différents, vous pouvez vous référez à [3] pour en savoir plus.

Rajouter un bouton sur la barre d'action

Lorsque nous avons crée nos activités, à chaque fois nous sommes partis d'une activité avec barre d'action. En effet nos deux activités héritent de **`ActionBarActivity`**. Un menu est rajouté par défaut contenant l'action *Settings*. Vous pouvez le faire apparaitre en cliquant sur le symbole en haut à droite de l'écran (fig. 3.16). Si vous cliquez sur *Settings* rien ne se passe, le code de réponse par défaut ne fait rien.

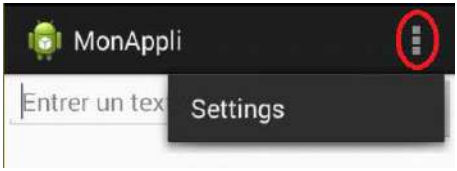


Figure 3.16 – Barre d'action

Nous allons, dans la suite, apprendre à rajouter notre propre item du menu et lui définir une action. C'est très simple !
Les boutons de la barre d'action sont définis dans un fichier xml du dossier *res/-menu*. Pour ajouter un bouton d'action à l'activité principale nous allons modifier le fichier *principale.xml* pour y insérer un item

```
<item
        android:id="@+id/action_mon_action"
        android:title="@string/action_mon_action"
        app:showAsAction="never"/>
```

L'attribut `app:showAsAction` définit quand et comment l'item doit apparaitre en tant que bouton sur la barre d'action. En précisant *"never"* notre item n'apparaitra pas sur la barre mais uniquement quand on déroule le menu. Si on précise *"ifRoom|withText"* le texte de l'item apparaitra s'il y a de la place.
Lancez l'application, vous devriez voir le nouvel item dans le menu. N'oubliez pas de définir la chaine *action_mon_action* dans *strings.xml*.
Il nous reste maintenant à implémenter l'action à exécuter quand notre item est sélectionné par l'utilisateur. Ceci s'effectue dans la méthode `onOptionsItem-Selected()` de la classe **Principale**. Repérez cette méthode et rajoutez les lignes suivantes dans son corps

```
if (id == R.id.action_mon_action) {
    //le traitement se fera ici
            return true;
}
```

Lancement d'une autre application

Nous allons maintenant utiliser notre bouton d'action pour lancer une autre application du système : un navigateur par exemple. Comme nous l'avons fait pour

lancer une deuxième activité de notre application, nous allons également utiliser un **Intent** pour lancer une deuxième application.

- Il faut d'abord créer l'intention

```
Uri webpage = Uri.parse("http://www.polytech.u-psud.fr");
Intent webIntent = new Intent(Intent.ACTION_VIEW, webpage);
```

La classe **Uri** fait référence à un URI (*uniform resource identifier*)[3]. Ici on forme un objet **Uri** à partir d'une chaine de caractère définissant une adresse web. Une localisation géographique, par exemple, peut aussi constituer un URI.

- Avant de la lancer, il faut tester s'il existe une application capable de répondre à cette intention dans notre cas ça revient à tester si un navigateur est présent sur l'appareil

```
PackageManager packageManager = getPackageManager();
List<ResolveInfo> activities = packageManager.
    queryIntentActivities(webIntent, 0);
boolean isIntentSafe = activities.size() > 0;
```

`queryIntentActivities()` retourne une liste d'activités capables de gérer l'objet **Intent** qu'on lui passe en paramètre.

- Si la liste n'est pas vide, on peut lancer l'intention en toute sécurité

```
if (isIntentSafe) {
            startActivity(webIntent);
}
```

Un **Intent** peut aussi porter un message à destination de l'autre application. On pourrait par exemple lancer l'application Calendrier en lui passant les détails de l'évènement qu'on souhaite rajouter sur notre agenda.
On peut aussi demander à ce que l'application qu'on lance nous renvoie un résultat comme pour par exemple récupérer les coordonnées d'un contact en lançant l'application Répertoire.
Des applications tiers peuvent aussi lancer notre application. Afin de gérer ces accès on peut définir des filtres d'intention et définir comment répondre à ces intentions.
Pour plus d'information sur les interactions entre les applications consultez [5].

3. l'URI est une chaine de caractère qui identifie le nom d'une ressource. L'URL est une forme d'URI

Changement de langue

Rendre votre application multilingue est très simple si vous avez défini tous vos textes dans *strings.xml*. Il suffit de définir un *strings.xml* pour chaque langue que vous voulez supporter et les placer dans des dossiers *values* nommés avec le code de la langue en préfixe. Par exemple *values-fr* pour le français, *values-en* pour l'anglais, *values-es* pour l'espagnol, etc.
Android sélectionnera les ressources appropriées en fonction de la langue que l'utilisateur a défini pour son appareil.

- ▷ Créez un dossier *values-en* dans le dossier *res* : Cilc droit sur *res* dans l'explorateur ECLIPSE puis *New>Folder*.
- ▷ Dans ce dossier créez un fichier *strings.xml* : Cilc droit sur le dossier dans l'explorateur ECLIPSE puis *New>Android XML File*
- ▷ Copiez ce que vous aviez déjà dans le *strings.xml* que nous avions utilisé jusqu'à présent et traduisez le texte.
- ▷ Chargez l'application. Changez la Locale de l'appareil et lancer l'application : L'application passe sur la langue que vous venez de sélectionner.

Si jamais vous choisissez une langue que votre application ne supporte pas, ce seront les chaines de caractères définies par défaut dans *values/strings.xml* qui seront utilisées.

Conclusion

Nous avons introduit quelques concepts de base de la programmation ANDROID. Avec les exemples relativement simples détaillés dans ce document nous avons présenté les premières étapes de la creation d'applcations.
Nous venons de vous accompagner dans vos premiers pas avec ANDROID, maintenant c'est à vous de jouer !

Lexique

Activity — Une activité représente un écran contenant une interface utilisateur. Une application est composée d'un ensemble d'activités. Les activités peuvent interagir entre elles, intra-application ou inter-applications.

apk file — C'est un fichier en format application package produit à l'issu de la compilation d'une application Android. C'est le fichier destiné aux utilisateurs. Ce fichier inclue le code de l'application sous forme d'un executable DVM (.dex), les ressources multimédias, et le AndroidManifest.xml.

DVM — Dalvik Virtual Machine. C'est une machine virtuelle disponible sur tout appareil Android. Elle execute des fichiers en format .dex (format de pseudo-code optimisé)

Fragment — Un fragment représente un comportement ou une portion de l'interface utilisateur dans une activité. On peut combiner plusieurs fragments dans une même activité pour créer une interface à volets multiples et réutiliser un fragment dans plusieurs activités.

Intent — C'est un message qui permet d'activer un composant (une activité par exemple). C'est un messager qui demande une action à un autre composant (de la même application ou d'une autre)

Layout — Le layout définit la structure visuelle d'une interface utilisateur. Les éléments d'un layout peuvent être définis soit dans un fichier XML ou bien à l'exécution à partir d'instructions dans le code. L'utilisation XML permet de mieux séparer l'apparence de l'application d'une part et le code qui définit son comportement d'autre part. Ceci permet de s'adapter plus facilement aux différentes cibles, il suffit de définir plusieurs fichiers XML pour un même code.

Manifest Toute application Android doit posséder un fichier XML nommé AndroidManifest.xml. Ce fichier contient des informations essentielles sur l'application que le système doit connaitre pour pouvoir la lancer. Ce fichier définit le nom du package de l'application, décrit les composants de l'application, détermine le processus qui accueillera ces composants, déclare les permissions dont l'application a besoin et celles que les autres doivent avoir pour interagir avec l'application, déclare la version Android minimale pour l'application, et liste les bibliothèques dont l'application a besoin.

Resources Ce sont les ressources en relation avec la présentation de l'application (images, fichiers audio etc.). Pour chaque fichier de ressource qu'on inclue dans le projet, un identifiant unique est créé pour référencer cette ressource. Le fait d'avoir des ressources séparées du code nous donne la possibilité de proposer des versions différentes en fonction de la configuration de la cible. En utilisant par exemple des images différentes selon la résolution de l'appareil, une mise en forme adaptée en fonction de l'orientation de l'affichage (portrait ou paysage), des fichiers de chaines de caractères en plusieurs langues pour que l'interface s'affiche dans la langue de l'utilisateur, etc.

Service Un service est un composant qui tourne en tâche de fond, pour effectuer de longues opérations ou exécuter des tâches pour un autre processus. Un service ne possède pas une interface utilisateur.

XML Extensible Markup Language (langage de balisage extensible), est un langage informatique conçu pour faciliter les échanges de données entre les systèmes d'informations. Il définit un ensemble de règles pour encoder les informations en format texte d'une façon indépendante de la machine. Il est basé sur des balises, cependant celles ci ne sont pas définies par le standardard XML, c'est l'utilisateur/programmeur qui définit ses propres balises. Le langage ne fait que définir les règles d'ecriture.

Annexes

Explication du code généré par défaut pour la classe Principale

```
/*nom du package que nous avons défini*/
package polytech.android.monappli;

/*****************************************************************/
/*importation des classes utilisées dans le code*/
/*****************************************************************/
import android.support.v7.app.ActionBarActivity;
import android.support.v7.app.ActionBar;
import android.support.v4.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.Menu;
import android.view.MenuItem;
import android.view.View;
import android.view.ViewGroup;
import android.os.Build;


/*****************************************************************/
/*Définition de la classe Principale qui hérite de ActionBarActivity*/
/*****************************************************************/
public class Principale extends ActionBarActivity {
/*redéfinition de la méthode onCreate() héritée de ActionBarActivity.
    Elle prend en paramètre un objet de type Bundle. La classe Bundle
    définit un type d'objet pouvant contenir un ensemble de données,
    et qui est destiné à échanger des données entre les activités. La
    méthode onCreate() est appelée une fois par le système au premier
    lancement de l'activité*/

    protected void onCreate(Bundle savedInstanceState) {
```

```
/* l'argument savedInstanceState permet au système de passer à l'
    activité l'etat dans lequel elle était à la fin de sa dernière
    exectution. En effet quand on arrête une activité le système
    appelle une certaine méthode ( onSaveInstanceState() ) dans
    laquelle on peut sauvegarder certaines informations concernant l'
    etat de notre activité pour les récupérer au moment où on la
    relance. On pourrait par exemple sauvegarder le texte qui a été
    tapé à l'exécution précédente, ou bien la position d'un scroll, ou
     autre, pour les remettre quand on redémarre l'activité.*/

/*appel de onCreate() de la classe mère*/
        super.onCreate(savedInstanceState);

/*avec setContentView() on définit le View que contiendra (affichera)
    notre activité. Ici c'est le layout activite_principale (identifié
     par R.layout.activite_principale)*/
        setContentView(R.layout.activite_principale);

/* une valeur nulle de savedInstanceState implique qu'on lance l'
    application pour la toute première fois*/
        if (savedInstanceState == null) {
/*d'abord, avec getSupportFragmentManager() on récupère le
    gestionnaire de fragments pour pouvoir interargir avec les
    fragments associés à cette activité. Pour ce gestionnaire on
    appelle beginTransaction() qui renvoie un objet de type
    FragmentTransaction qui représente une série de transactions qu'on
     pourra effectuer sur les fragments. Avec l'appel à add() on
    rajoute un fragment à l'activité. On lui donne l'identifiant du
    layout de notre activité principale (défini par
    activite_principale.xml) ainsi qu' un objet de type
    PlaceholderFragment qu'on définira plus loin. L'appel à add()
    renvoie l'objet FragmentTransaction auquel le fragment vient d'
    être rajouté. Enfin avec commit() on engage la transaction.*/
            getSupportFragmentManager().beginTransaction().add(R.id.
                container,new PlaceholderFragment()).commit();
/*POUR RESUMER: on vien de rajouter le fragment à l'activité*/
        }
    }

/*La méthode onCreateOptionsMenu() est appelée au moment de la
    création de la barre d'action. Le système lui passe un objet Menu
    en paramètre, qu'on populera dans la méthode: C'est sur cet objet
    qu'on rajoutera les éléments du menu */
    public boolean onCreateOptionsMenu(Menu menu) {
```

```
/*getMenuInflater() renvoie un objet de type MenuInflater. C'est une
    classe qui est capable de créer un objet menu à partir d'un
    fichier xml, et ce grace à la méthode inflate().  Il suffit de lui
     donner l'identifiant du menu xml (principale.xml) et l'objet de
    type Menu dans lequel on veut créer les items du menu*/
        getMenuInflater().inflate(R.menu.principale, menu);
        /*il faut renvoyer true pour que le menu s'affiche*/
        return true;
}

/*méthode appelée lorsqu'un item du menu est sélectionné*/
    public boolean onOptionsItemSelected(MenuItem item) {
/*l'appui sur le bouton de retour est automatiquement traité si on
    défini une activité parent dans AndroidManifest.xml, on n'a pas à
    le traiter ici*/

/*on récupère l'identifiant de l'item sur lequel l'action a été faite
    */
    int id = item.getItemId();
    if (id == R.id.action_settings) {
    /*l'action settings a été sélectionnée. Ici on ne fait rien*/
            return true;
    /*la méthode renvoie true pour indiquer qu'elle a traité l'action
        */
    }
    /*si on ne traite pas l'action, on demande à la classe mère de le
        faire en invoquant sa méthode onOptionsItemSelected(). Si la
        classe mère ne traite pas l'action elle renverra false (c'est
        le comportement par défaut de ActionBarActivity)*/
    return super.onOptionsItemSelected(item);
}

/*****************************************************************/
/*Définition d'une classe interne                                */
/*****************************************************************/

/* classe PlaceholderFragment qui hérite de Fragment*/
    public static class PlaceholderFragment extends Fragment {

/*constructeur qui ne fait rien*/
        public PlaceholderFragment() {
        }

/*redéfinition de onCreateView() de la classe Fragment. Elle est
```

```
    appelée pour créer l'interface utilisateur du fragment*/
        public View onCreateView(LayoutInflater inflater, ViewGroup
            container,Bundle savedInstanceState) {
/*inflater: objet qui permet la création d'un layout à partir d'un xml
container: le parent qui va contenir le fragment
savedInstanceState: même signification que le paramètre du onCreate de
     la classe Principale*/

/*appel à la méthode inflate() pour créer l'interface correspondante
    au layout. On lui passe l'identifiant de la ressource xml du
    layout et le parent auquel il faut se rattacher (le ViewGroup qui
    contiendra le fragment) si le troisième paramètre est true. S'il
    est false le container sera juste utilisé  pour récupérer les
    paramètres du layout. Lorsqu'on précise false c'est l'activité qui
     sera considérée comme le parent du view*/
            View rootView = inflater.inflate(R.layout.
                fragment_principale,container,false);
/*renvoi de l'objet View crée*/
            return rootView;
        }
    }
}
```

Cycle de vie d'une activité

Toute activité passe par plusieurs états durant son cycle de vie, il est important de connaitre ce cycle ainsi que les méthodes qui sont appelées à chaque fois que l'application bascule d'un état vers l'autre. Le diagramme de la figure 3.17 résume le cycle de vie d'une activité. Vous trouverez une explication détaillées des différents états et méthodes ici et là.

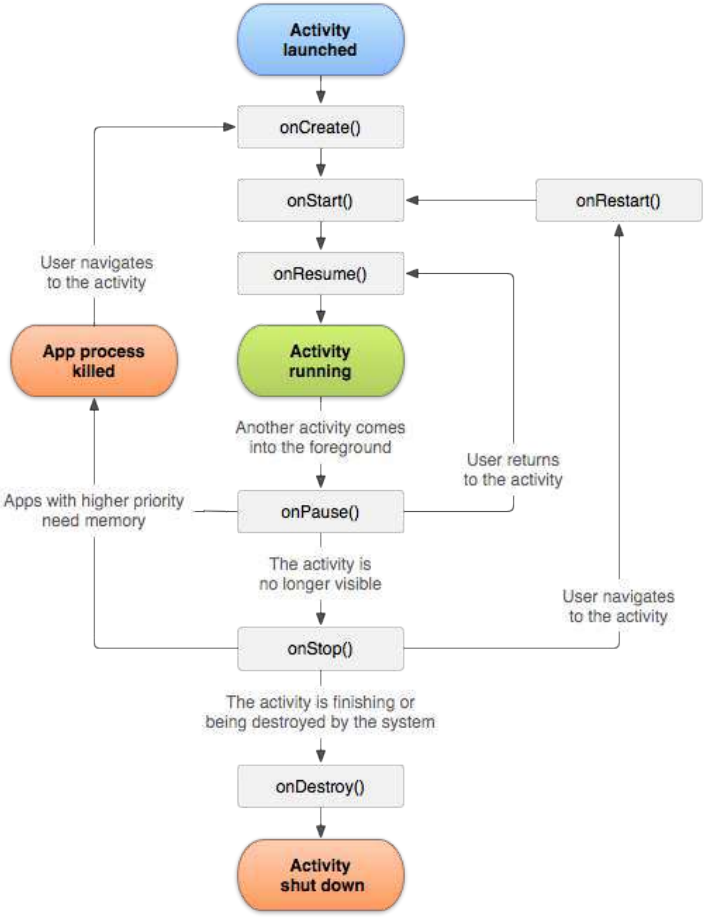


Figure 3.17 – Cycle de vie d'une activité

[1]

Bibliographie

[1] http ://developer.android.com/reference

[2] Wallace Jackson, Learn Android App Development, Apress (Springer Verlag), May 2013

[3] https ://developer.android.com/training/multiple-threads/ communicate-ui.html

[4] Chris Haseman, Android Essentials, Apress, July 2008

[5] https ://developer.android.com/training/basics/intents/index.html