Développement mobile

Patrick Albers

9 février 2021

Objectifs en matière de compétences : mettre en application les fonctionnalités et capacités de la plate-forme *Android* dédiée à la réalisation d'applications pour téléphones mobiles

Objectifs en matière de connaissances :

- comprendre l'architecture *Android*
- comprendre le principe des modèles de composants
- comprendre les interfaces utilisateurs *Android*

Programme : 13 séances de cours planifiées

- architecture de la plate-forme Android : environnement de développement, compilation et exécution, sécurité des applications
- composants logiciels : activités, intentions, fragments, services
- interface utilisateur : *layouts*, menus, listes, navigation, boîtes de dialogue
- gestion des données : configuration, utilisation de fichiers, base de données *SQLite*
- projet applicatif de mise en œuvre

Évaluation : QCM intermédiaire + projet applicatif

1 Présentation d'Android

Android permet le développement d'applications mobiles sur tablettes et *smartphones*, mais également sur des objets connectés, des téléviseurs ou sur des systèmes embarqués comme les voitures.

Android est devenu en quelques années le système d'exploitation le plus déployé pour la mobilité. Ce succès est en partie dû au fait que cette plate-forme est libre. Cela a eu comme conséquence que de nombreux développeurs ont pu s'en accaparer et développer une multitude d'applications disponibles pour le grand public. On compte aujourd'hui plus d'un milliard d'applications téléchargeables sur des plates-formes comme le *Play store* de *Google*. Ces applications y sont gratuites ou payantes. Tout développeur, possédant un compte sur le *play store*, peut publier une application.

1.1 Versions d'Android

Créé par une *startup*, Android était à la base un nouveau système d'exploitation pour appareil photo numérique. Racheté par Google en 2005, il évolue très vite avec aujourd'hui plus de 40 versions qui se sont succédées.

Jusqu'en 2019, les versions majeures possèdaient un nom de dessert et se suivaient par ordre alphabétique. Chaque version utilise une version d'API [1] différente. C'est par le numéro d'API que les développeurs précisent la version d'Android avec laquelle ont été développées leurs applications.

Version	Date	Nom	API
1.0	09/2008	Apple Pie	1
1.1	02/2009	Banana Split	2
1.5	04/2009	Cupcake	3
1.6	09/2009	Donut	4
2.0	10/2009	Eclair	5
2.2	05/2010	Froyo	8
2.3	12/2010	GingerBread	9
3.0	02/2011	Honeycomb	11
4.0.1	10/2011	Ice cream sandwish	14
4.1	07/2012	Jelly bean	16
4.4	10/2013	Kit kat	19
5.0	11/2014	Lollipop	21
6	10/2015	Marchmallow	23
7	08/2016	Nougat	24
8	08/2017	Oreo	26
9	03/2018	Pie	28
10	09/2019	Android 10	29
11	09/2020	Android 11	30

FIGURE 1 – Les différentes versions d'Android

Android garantit une compatibilité ascendante des applications : c'est à dire que les applications développées avec une version particulière sont garanties de fonctionner avec des versions ultérieures. Mais cela implique également qu'une application développée par exemple avec la version 3.0 ne pourra pas s'exécuter sur une version 2.

Le développement des applications Android doit non seulement prendre en compte la version, mais également la diversité des terminaux sur lesquels les applications vont s'exécuter.

1. *Application Programming Interface*

Lorsque le système d'exploitation Android est utilisé sur une plate-forme mobile, ce sont les constructeurs qui précisent la version d'Android choisie et la mettent à jour. Cela complique la tache des programmeurs qui doivent s'assurer en plus que l'application développée fonctionnera sur la majorité des plates-formes mobiles.

Afin d'aider les développeurs à faire ces choix, Google fournit deux fois par mois des statistiques sur le nombre de statistiques d'utilisation de chaque version à l'adresse suivante :

`https://developer.android.com/about/dashboards`

Ces statistiques se basent sur la publication des applications sur *Play Store*, le magasin d'applications de Google. Voici les chiffres pour mai 2019 :

Gingerbread	0,3%
Ice cream sandwich	0,3%
Jelly Bean	3,2%
Kit kat	6,9%
Lollipop	14,5%
Mashmallow	16,9%
Nougat	19,2.3%
Oreo	28,3%
Pie	10,4%

FIGURE 2 – Les statistiques d'utilisation des versions en mai 2019

Il est possible aujourd'hui d'utiliser deux langages pour développer les applications Android : *Java* et *Kotlin*. Ce dernier n'a été créé qu'en 2017. C'est en 2018 que *Google* a conseillé l'utilisation de *Kotlin* pour le développement des applications d'Android. On peut imaginer que ce langage prendra la place petit à petit de *Java*. Nous nous concentrerons uniquement dans le cours de développement mobile sur la programmation en *Java*.

1.2 Architecture d'une application Android

<table>
<tr><td colspan="2">Applications (Java)
(VM ART - Android Run Time)</td><td rowspan="2">Applications
(natives)</td></tr>
<tr><td>Bibliothèques (Java)
(affichage, BDD, web, audio, ...)</td><td>Bibliothèques (Java)
(standard)</td></tr>
<tr><td colspan="3">Bibliothèques (C)
(std, affichage 2D/3D, moteur BDD, moteur web, audio, ...)</td></tr>
<tr><td colspan="3">Noyau Linux</td></tr>
</table>

FIGURE 3 – Architecture simplifiée de la plate-forme Android avec *Java*

La plate-forme Android se base sur un noyau *linux*. Le langage utilisé est *Java*. Android propose des bibliothèques supplémentaires qui permettront de développer les applications mobiles. Les programmes s'exécutent sur une machine virtuelle *Java* spécifique : *Android Run Time*. Il est possible d'intégrer du code *C++* (développement natif) pour des aspects particuliers de programmation, notamment pour des applications nécessitant des temps d'exécution très rapides comme pour de l'affichage $2D$ ou $3D$.

1.3 Environnement de développement

Le développement des applications Android peut se faire de manières différentes. On peut utiliser des outils en ligne de commande qui sont fournis par le *SDK*. Historiquement, c'est l'environnement de développement intégré *eclipse ADT* (*Android development Tools*) qui était utilisé, mais il n'est plus maintenu par *Google*. L'environnement officiel de développement depuis 2015 est *Android Studio*.

Le *SDK* d'*android* (*Software Development Kit*) fournit :
- l'ensemble des bibliothèques pour le développent *Java* : code sources et documentation des *API*,
- des outils en ligne de commande,
- l'outil *AVD* (*Android Virtual device*) pour l'émulation de terminaux *android* sur un ordinateur,
- des outils d'administration comme *ADB* (*Android Debug Bridge*) pour la communication avec les périphériques *android* cible[2].

2 Une première application

Android s'appuie sur le noyau *linux* pour son fonctionnement. En particulier, chaque application s'exécute dans un processus de manière indépendante. *Linux* étant multitâches, les applications Android peuvent s'exécuter en parallèle de manière optimale.

Chaque application Android possède des droits d'accès au système de fichiers de l'appareil qui lui propres. Pour se faire, lorsqu'une nouvelle application est installée, un utilisateur (au sens *unix* du terme) lui est associé.

Pour garantir un maximum de sécurité, la liste des ressources de l'appareil Android auxquelles l'application doit accéder doit être spécifiée par l'application et doit être validée par l'utilisateur lors de son installation.

Enfin des mécanismes sont disponibles pour permettre aux applications de communiquer en cas de besoin.

2.1 création d'un nouveau projet

1. Lancez l'application *AndroidStudio*.

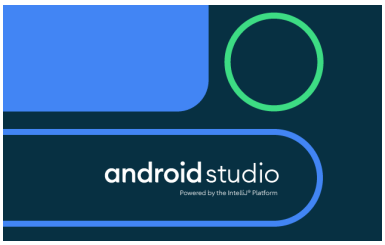

FIGURE 4 – Lancement de l'application *AndroidStudio*

2. Si un projet Android est ouvert, alors l'environnement de programmation s'affiche directement (figure 9). Sinon la fenêtre de démarrage s'affiche.

   Sur la partie gauche, apparaissent les projets en cours. Sur la partie droite, on peut choisir :

2. Typiquement, l'ordinateur où est développé le code et le terminal *android* cible, sont connectés par cable *USB* ou par *wifi*.

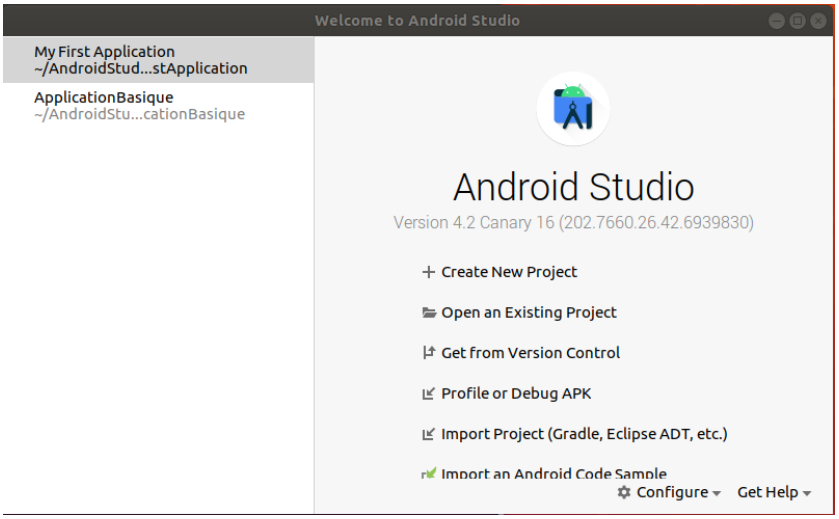

FIGURE 5 – Fenêtre de démarrage

— de créer un nouveau projet,
— d'ouvrir des projets existants,
— d'ouvrir des projets provenant d'un dépôt *git* ou autre,
— ou d'éditer les propriétés du projet.

Le menu du bas *configure* permet de configurer l'environnement de programmation. On peut notamment ajouter un *sdk* d'une version particulière (par défaut, seul le *sdk* de la dernière version est installée), configurer le terminal virtuel pour l'exécution ou encore spécifier l'endroit de stockage des fichiers. Tous ces paramètres sont modifiables après la création.

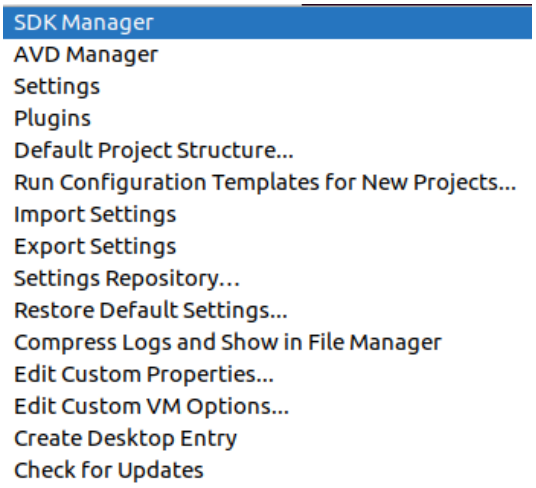

FIGURE 6 – Fenêtre de configuration

3. Choisissez la création d'un nouveau projet. L'assistant de création démarre.

   Sur la gauche, une liste d'appareils Android cible est présenté : smartphones et tablettes, objets connectés (*wear OS*), télévisions Android, TV et Android Auto, ou autres objets Android.

   Sur la droite, on peut choisir l'activité principale désirée. Les activités sont les éléments de base des applications d'Android, et correspondent chacune à un écran particulier. Les choix

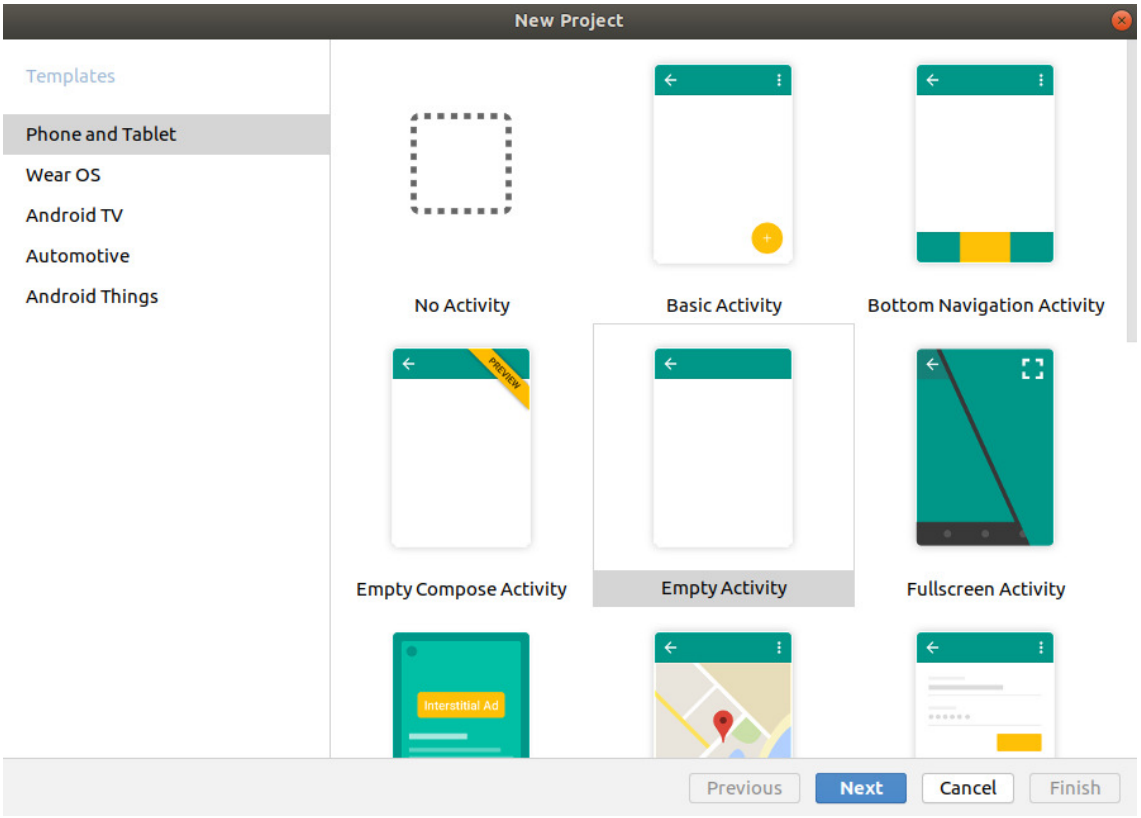

FIGURE 7 – Assistant de création de nouveau projet

possibles pour les applications pour smartphones et tablettes sont :
- *NoActivity* : pas d'activité par défaut,
- *BasicActivity* : une activité comportant un bouton cliquable en bas de l'écran,
- *BottomNavigationActivity* : une barre de navigation ajoutée à ma page en bas de l'écran,
- *EmptyActivity* : une simple activité où aucun élément n'est ajouté,
- *FullscreenActivity* : une activité en mode plein écran ; la barre de navigation est masquée par défaut,
- *GoogleAdmobAdsActivity* : une activité pour gérer les publicités de la régie publicitaire de *Google*
- *GoogleMapsActivity* : une activité pour intégrer les éléments de *Google maps*
- *LoginActivity* : une activité pour gérer les connexions des utilisateurs
- *Primary/Detail Flow* : une activité avec une page principale (*primary*) et une vue détaillée
- *NavigationDrawerActivity* : une activité avec plusieurs écrans,
- *SettingsActivity* : une activité affichant un écran de paramétrage
- *ScrollingActivity* : une activité permettant de faire défiler l'écran
- *TabbedActivity* : une activité permettant d'intégrer des onglets de navigation,
- *FragmentViewModel* : une activité pour afficher plusieurs écrans,
- *NaticeC++* : une activité appelant du code en *C++*.

La notion d'activité ainsi que les différents éléments graphiques seront détaillés plus loin.

4. Choisissez une application pour smartphone, et l'activité *Basic Activity*. Une fenêtre pour renseigner les informations nécessaires du projet apparaît.

   On y retrouve le nom de l'application, le nom du paquetage où se trouveront les classes de

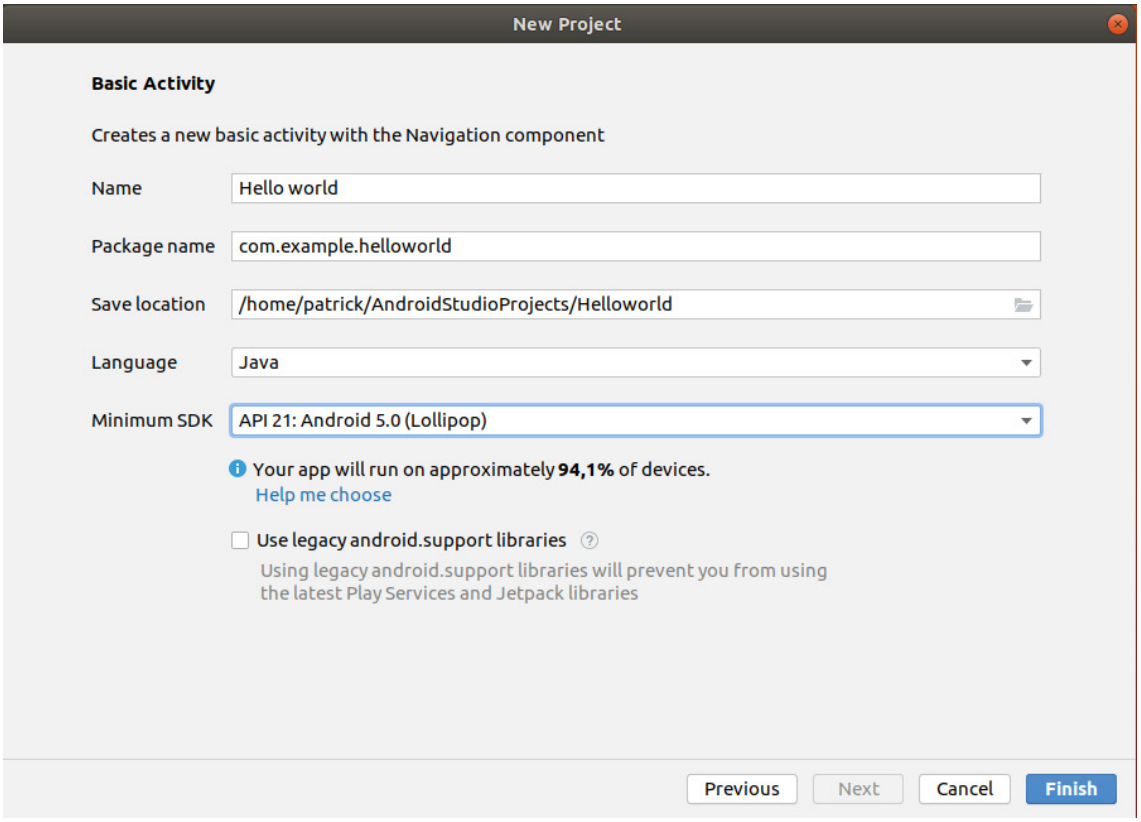

FIGURE 8 – Informations pour le nouveau projet

l'application, le langage désiré (*Java* ou *Kotlin*) et le *SDK* voulu. *Android studio* indique le pourcentage d'appareils Android où pourra s'exécuter l'application en fonction du *SDK* choisi.

Si l'on veut utiliser les anciennes bibliothèques *android.support*, il faut cocher la case *Use legacy android.support librairies*. Si la case n'est pas cochée, ce sont les nouvelles bibliothèques *AndroidX* qui seront utilisées. Toutes les nouvelles applications devraient utiliser ces nouvelles bibliothèques.

5. Choisissez le nom de votre application, le langage *Java*, ainsi que la version de l'API 21 (Lollipop). Terminez la configration en cliquant sur *Finish*.

   Le projet se crée et l'environnement de travail apparaît (figure 9). Être connecté sur internet est parfois nécessaire pour le chargement de bibiothèques.

2.2 Présentation de l'environnement

L'environnement de travail est présenté en figure 9. L'interface est classique et reste identique aux différents environnements de développement intégré que l'on peut rencontrer.

- dans la barre du haut, on y retrouve les menus :
  - *File* : ouvrir et fermer les projets, paramétrer l'application (*settings*), éditer la structure du projet, choisir l'encodage des caractères, synchroniser le projet avec le gestionnaire de compilation (*gradle*), importer ou exporter les données du projet
  - *Edit* : éditer le texte, choisir le mode de sélection
  - *Views* : afficher les différentes fenêtres d'*Android studio*, afficher les informations sur les éléments de programmation (classe, méthodes, paramètres, ...)

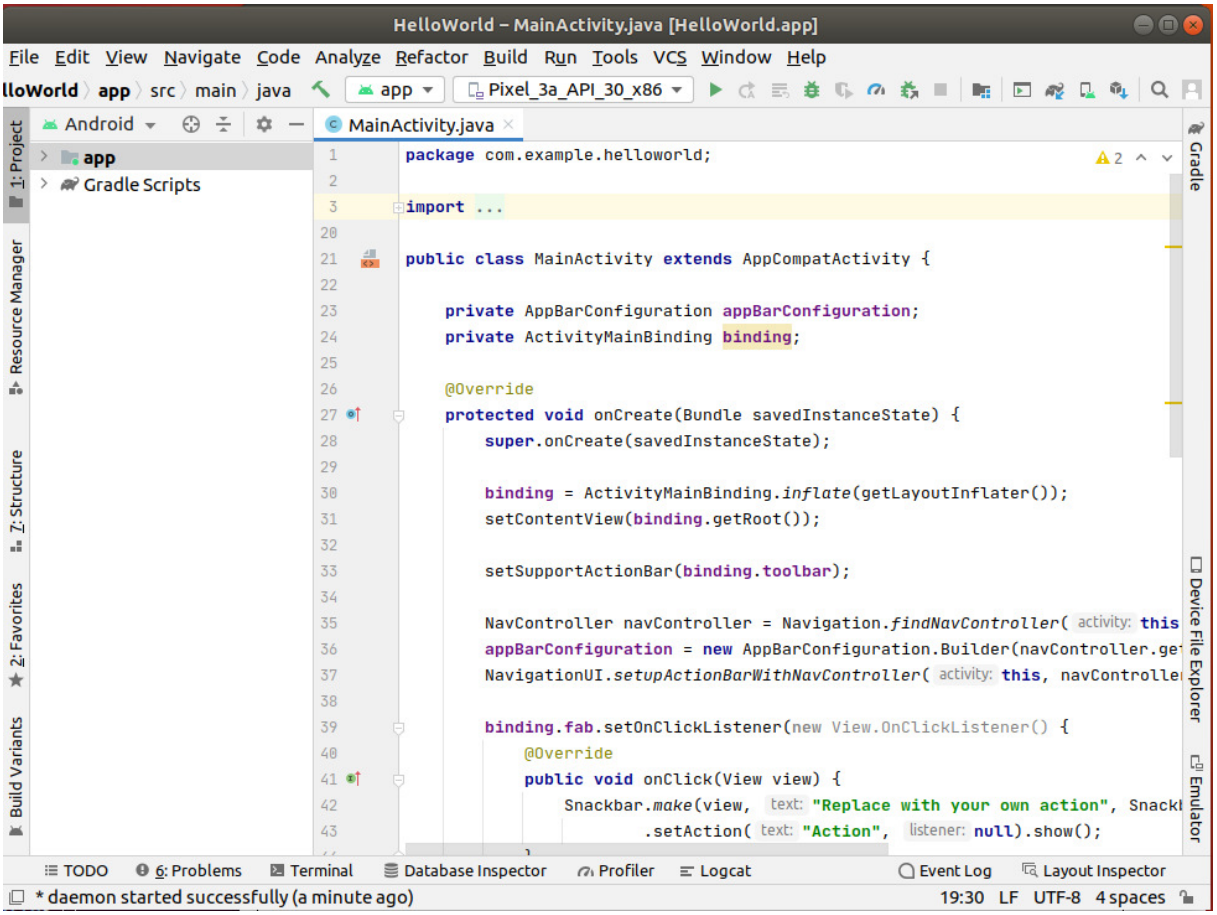

FIGURE 9 – Environnement de programmation *AndroidStudio*

- *Navigate* : naviguer dans les fichiers de programmation
- *Code* : ajouter des méthodes, faire de la complétion de code, indenter le code, ...
- *Analyse* : inspecter le code, gérer les dépendances
- *Refactor* : modifier, renommer, migrer le code
- *Build* : compiler le code
- *Run* : exécuter ou déboguer le code
- *Tools* : gérer les tâches et les contextes, gérer des terminaux virtuels, gestion des *SDK*, gérer les ressources, ...
- *VCS* (*version support system* : gérer les systèmes de gestion de versions comme git, GitHub, CVS, Subversion ; ...
- *Window* : gérer l'affichage des fenêtres
- *Help* : afficher l'aide en ligne

- Juste en dessous de la barre des menus, la barre d'outils présente des raccourcis pour des opérations qui reviennent régulièrement : lancer ou stopper l'application (run), mettre à jour l'application sur des matériels android, déboguer, voir la structure du projet, manager les terminaux virtuels (*AVD*), manager les *SDK*, ...
- sur la partie gauche, on retrouve l'ensemble des fichiers et des ressources du projet,
- sur la partie droite est affiché le code des fichiers,
- sur la partie basse, sont présents différents onglets : *todo* liste, affichage des problèmes de compilation, affichage d'un terminal, informations sur le débogage, profilage de l'application

(performances de la gestion des ressources comme la mémoire, le processeur, le réseau, ou encore la batterie), affichage des fichiers de logs

2.3 Compilation et exécution

Le gestionnaire *Gradle* est utilisé dans *Android Studio* pour la compilation. Son but est d'organiser la compilation du projet en prenant en compte non seulement l'ensemble des fichiers du projet mais également les bibliothèques externes utilisées, ainsi que le *SDK* qu'il faut utiliser. Ce gestionnaire est présenté plus en détail dans la partie 3.3.

La compilation est faite au fur et à mesure du codage. Il est possible cependant de faire une compilation de l'ensemble du projet en cliquant sur *Make Project* du menu *Build* (ou *Ctrl-F9*).

On peut voir juste en dessous de la barre d'outil (en haut à droite), les erreurs et les *warnings* de chaque fichier : le point d'interrogation indique le nombre d'erreurs, et le triangle indique le nombre de *warnings* détectés. Si aucune erreur n'est détectée, le symbole ✔ s'affiche.

En survolant cette zone, on peut choisir ce que l'on veut afficher : seulement les erreurs ou, les erreurs et les *warnings*.

Cliquer sur cette zone ouvre l'onglet *problems* où l'on peut voir le détail des erreurs.

En cliquant sur les flèches, on accède directement aux différentes lignes concernées. Une ampoule orange indique la ligne où se trouve l'erreur. En cliquant sur ce point, on peut voir les suggestions de correction d'erreur que nous propose *Android Studio*.

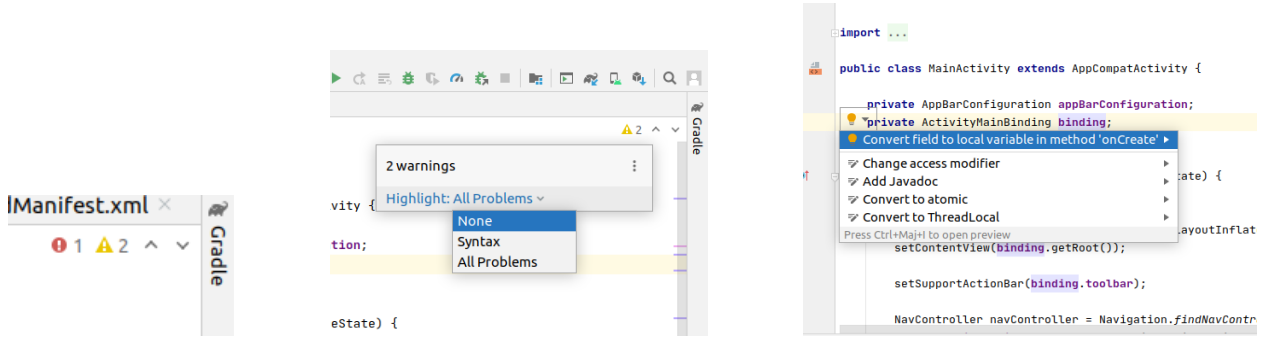

FIGURE 10 – compilation automatique

L'outil *AVD* (*Android Virtual device*) permet d'émuler des terminaux *android*[3]. Il est fourni avec le *SDK*, et permet de tester localement une application sur la machine de développement.

On accède à *AVD* directement dans la barre d'outils ($4^{\text{ème}}$ icone à partir de la droite). Si aucun terminal virtuel n'est dans la liste, il faut en ajouter un via le bouton *create virtual device*.

On peut également tester le terminal en cliquant sur le triangle de gauche de la colonne *Actions*.

Sur *Ubuntu*, il se peut qu'il y ait des problèmes de droit *kvm*. Voir la page :

`https://stackoverflow.com/questions/37300811/android-studio-dev-kvm-device-permission-denied/45749003`

L'outil de communication avec les périphériques *ADB* (*Android Debug Bridge*) est fourni avec le *SDK*, et permet de connecter tout type d'appareil Android et de leur transférer le code. Cette étape est nécessaire dans la mesure où le but final est de tester l'application dans des conditions réelles. Tout appareil Android peut être utilisé.

3. Pour plus d'informations, voir : https ://developer.android.com/studio/run/emulator

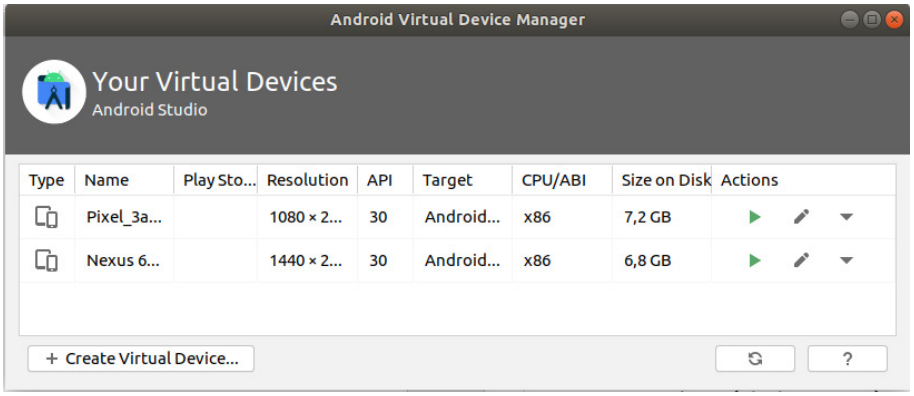

FIGURE 11 – Le manager de terminaux virtuels

Le test d'une application sur un appareil Android nécessite deux étapes préliminaires :

1. Il faut tout d'abord activer le débogage *USB* sur l'appareil Android dans ses paramètres de configuration. Il est conseillé de se reporter aux instructions d'utilisation de l'appareil.

   En général, il faut aller dans les paramètres de l'appareil, puis chercher le menu **options de développement** (menu *système* ou *applications*)[4]. Il faut ensuite cocher la case **débogage USB** et valider.

2. Il faut ensuite configurer l'ordinateur afin qu'il reconnaisse le matériel connecté par *USB*[5]. Cette opération dépend du système d'exploitation de l'ordinateur.

   Sous *Mac* et *Chrome OS*, la connexion est détectée automatiquement.

   Sous *Ubuntu*, il faut tout d'abord s'assurer que l'utilisateur fait partie du groupe *plugdev*, puis installer si ce n'est déjà fait le paquetage *udev rules*, en tapant les instructions suivantes :

   ```
   sudo usermod -aG plugdev $USERNAME
   sudo apt-get install android-sdk-platform-tools-common
   ```

   Sous *Windows*, il est nécessaire d'installer un pilote *USB* approprié[6]. Pour cela, il faut tout d'abord récupérer le driver sur le site du fabricant, puis l'associer au périphérique aprés l'avoir connecté (menu *"gestion des périphériques"*).

Une fois connecté, l'appareil Android doit apparaître dans la liste des appareils (*running devices*). Pour lancer l'application, il suffit d'appuyer sur ▶ (*Run app*) et pour la stopper sur ■.

2.4 Outils de débogage

2.4.1 Messages *Toast*

La classe *Toast*[7] permet d'afficher des messages dans une fenêtre pendant une courte durée. Ce n'est pas à proprement parlé du débogage, mais cela permet d'afficher des messages afin de savoir si l'on *passe* bien au bon endroit du code ou de connaître la valeur d'une variable à un moment donné. C'est l'équivalent du *System.out.println()* des applications *Java*. Voici un exemple de code :

4. Il se peut que le menu soit caché sur certains appareils (les tablettes notamment). Pour le faire appraître, allez dans le menu *"Systèmes>À propos>Numéro de build"*, et cliquez plusieurs fois sur ce même numéro jusqu'à ce que le message *"vous êtes maintenant développeur"* s'affiche. Revenez alors dans le menu *Système*, le menu *"Options pour le développeur"* est maintenant présent.
5. https ://developer.android.com/studio/run/device
6. https ://developer.android.com/studio/run/oem-usb
7. https ://developer.android.com/guide/topics/ui/notifiers/toasts

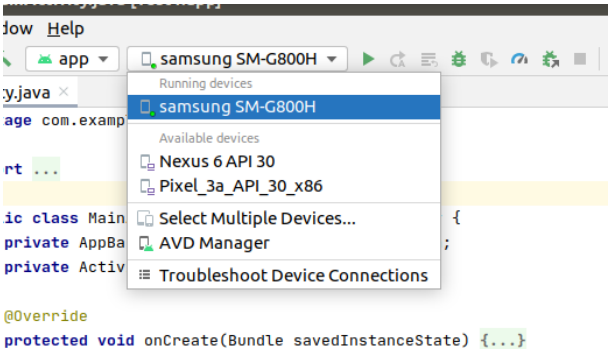

FIGURE 12 – Liste des terminaux disponibles

```
Toast.makeText(getApplicationContext(), "un petit message", 2).show();
```

Les deux méthodes utilisées ici sont la méthode statique *makeText*() qui construit le message, et la méthode *show*() qui permet d'afficher la fenêtre. Le premier argument de *makeText*() correspond au contexte de l'application, le second au message lui même, et le troisième à la durée d'apparition en secondes de la fenêtre. La notion de contexte sera abordée plus tard.

2.4.2 Le mécanisme *Logcat*

Le mécanisme *Logcat*[8] est intégré à la plate-forme *Android Studio*. C'est un fichier de log où l'on trouve les informations du terminal connecté. On accède à ces logs sur l'onglet *Logcat* en bas de la page d'*Android Studio*.

On peut filter selon le degré d'importance du message du plus simple au plus important : *verbose*, *debug*, *info*, *warning*, *error* et *assert*. Il est possible d'écrire dans ces logs à chaque niveau d'importance :

```
Log.v(String, String) (verbose)
Log.d(String, String) (debug)
Log.i(String, String) (information)
Log.w(String, String) (warning)
Log.e(String, String) (error)
```

Le premier argument de ces méthodes correspond à une étiquette (*TAG*) qui caractérise le message du log (*i.e.* le nom de la classe ou de la méthode où se trouve le message). Le second correspond au message lui même. Voici un exemple d'utilisation :

```
package com.example.test1;
...
import android.util.Log;

public class MainActivity extends AppCompatActivity {
    public static final String TAG = "MainActivity";

    protected void onCreate(Bundle savedInstanceState) {
        ...
        Log.v(TAG, "mon message dans les logs");
    }
```

8. https ://developer.android.com/studio/command-line/logcat#logClass

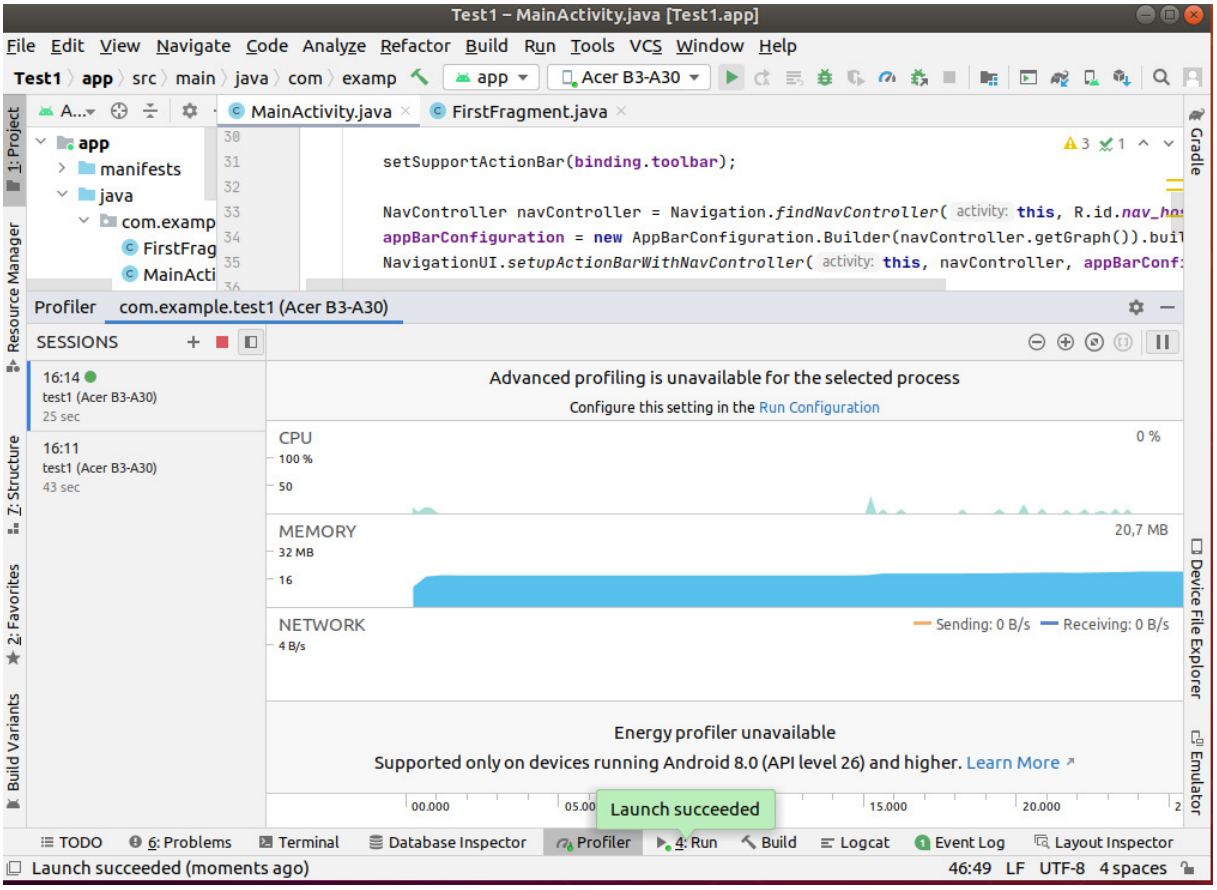

FIGURE 13 – Outil de profilage des performances

```
    ...
}
```

Le message s'inscrira comme suit dans les logs :

```
12-29 15:44:59.523 14513-14513/com.example.test1
                   V/MainActivity: mon message dans les logs
```

2.5 Outils d'analyse de performance

Les outils de performance permettent d'analyser l'utilisation des ressources (le processeur, la mémoire, les graphiques, le réseau ou encore la batterie) de l'application. Cet outil est disponible dans l'onglet *Profiler* de l'environnement *Android Studio*.

Une fois l'application lancée, l'outil permet de voir ses performances, comme il est possible de voir dans la figure 13. Pour plus d'informations sur son utilisation, voir :

```
https://developer.android.com/studio/profile/android-profiler?hl=en
```

3 Structure d'un projet avec *Android Studio*

Tous les projets Android possèdent la même structure. Il existe plusieurs vues possibles d'un projet qui permettent de voir différemment son architecture :

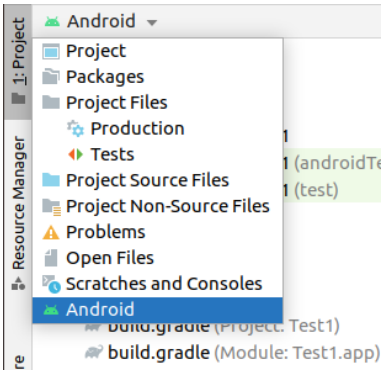

FIGURE 14 – Les différentes vues d'un projet

— La vue *Project* permet de voir la structure du projet conforme aux différents environnements de programmation comme *eclipse* par exemple. On y retrouve notamment les différentes bibliothèques du projet et leur code source.
— La vue *Package* permet voir l'ensemble des paquetages.
— La vue *Project Files* reprend l'architecture du point de vue du système de fichiers.
— La vue *Project Source Files* permet de voir tous les fichiers source du projet.
— La vue $Project\ Non-Source\ Files$ permet de voir tous les autres fichiers qui ne sotn pas des fichiers de code source, comme par exemple les fichiers *xml*.
— La vue *Problems* permet de voir les éventuels fichiers où il y a des erreurs de compilation.
— La vue *Open Files* permet de voir l'ensemble des fichiers ouverts du projet dans l'éditeur.
— La vue *Scratches and Consoles* permet de voir les consoles ouvertes.
— Enfin la vue *Android* permet de voir la structure du projet du point de vue la plate-forme Android :
  - $app/manifests$: la configuration de l'application
  - $app/java$: le code source et les tests de l'application, le code généré
  - app/res : les ressources que l'application utilise
    > *drawable* : les images
    > *layout* : la composition de vues
    > *menu* : la composition de menus
    > *mipmap* : les icônes de lancement de l'application
    > *nagivation* : la navigation entre les différentes vues
    > *values* : les valeurs de données (couleur, dimension, ...)
  - *Gradles* : les règles de compilation de l'application

3.1 Les fichiers *manifeste*

Les fichiers *manifeste* peuvent être vus comme des fichiers de configuration au format *xml*. Ils contiennent toutes les informations nécessaires au bon fonctionnement de l'application. On y retouve le point d'entrée de l'application, les permissions nécessaires pour son fonctionnement, la version minimale d'Android pour l'exécution, les bibliothèques nécessaires, ou encore les systèmes et matériels compatibles avec l'application.

Voici un exemple de fichier :

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.example.test1">
```

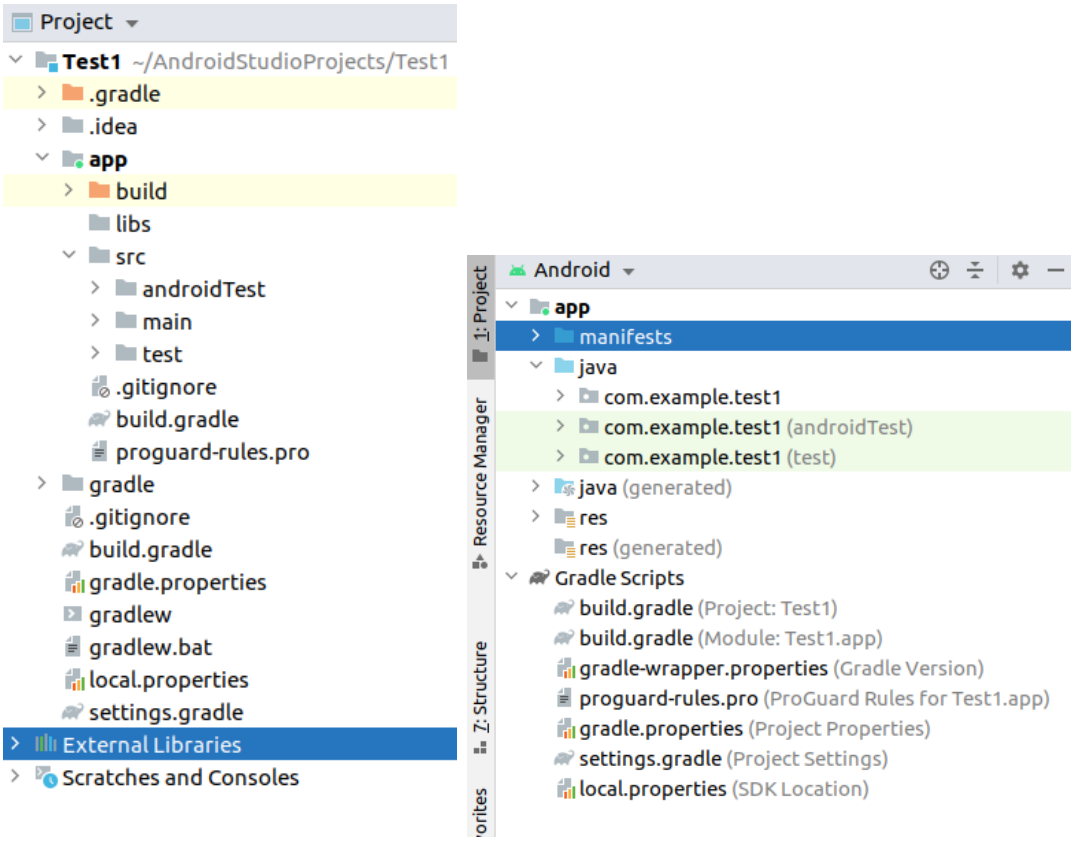

FIGURE 15 – Les vues *Project* et *Android*

```
 <application
   android:allowBackup="true"
   android:icon="@mipmap/ic_launcher"
   android:label="@string/app_name"
   android:roundIcon="@mipmap/ic_launcher_round"
   android:supportsRtl="true"
   android:theme="@style/Theme.Test1">
   <activity
     android:name=".MainActivity"
     android:label="@string/app_name"
     android:theme="@style/Theme.Test1.NoActionBar">
     <intent-filter>
       <action android:name="android.intent.action.MAIN"/>
       <category android:name="android.intent.category.LAUNCHER"/>
     </intent-filter>
   </activity>
 </application>
</manifest>
```

La balise <*manifest*> est la balise racine du fichier. L'attribut *xmlns* : *android* a pour valeur l'espace de nom Android ; sa valeur ne doit pas être modifiée. L'attribut *package* est le nom du paquetage de création.

La balise <*application*> décrit tous les éléments de l'application (images, icônes, thème). La

balise <*activity*> donne le point d'entrée de l'application.

3.2 Le fichier *APK*

Ce fichier de configuration *manifest.xml* ne contient pas pourtant toutes les informations. Il est en fait complété avec les informations recueillies lors de la compilation. On peut le trouver dans le répertoire :

```
app/build/outputs/apk/debug/app-debug.apk
```

Ce fichier *APK* (*Android Package Kit*) est unique pour chaque application Android, et contient tous les fichiers la composant : les fichiers sources, les fichiers ressources, ou encore les bibliothèques extérieures. Il est destiné en particulier pour le déploiement sur un terminal Android.

Le fichier *APK* contient également un fichier *manifest.xml* dont voici un exemple :

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:versionCode="1"
  android:versionName="1.0"
  android:compileSdkVersion="30"
  android:compileSdkVersionCodename="11"
  package="com.example.test1"
  platformBuildVersionCode="30"
  platformBuildVersionName="11">
  <uses-sdk
    android:minSdkVersion="21"
    android:targetSdkVersion="30" />
  <application
    android:theme="@ref/0x7f1101cd"
    android:label="@ref/0x7f10001c"
    android:icon="@ref/0x7f0d0000"
    android:debuggable="true"
    android:testOnly="true"
    android:allowBackup="true"
    android:supportsRtl="true"
    android:roundIcon="@ref/0x7f0d0001"
    android:appComponentFactory="androidx.core.app.CoreComponentFactory">
  <activity
    android:theme="@ref/0x7f1101cf"
    android:label="@ref/0x7f10001c"
    android:name="com.example.test1.MainActivity">
    <intent-filter>
      <action android:name="android.intent.action.MAIN" />
      <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
   </activity>
 </application>
</manifest>
```

On peut voir que les informations sur la version d'Android, le numéro du *SDK* utilisé, ou encore le numéro du *SDK* minimum, ont été ajoutées.

3.3 *Gradle* : le gestionnaire de compilation

Le gestionnaires de compilation est le chef d'orchestre du projet. Il comporte des règles de compilation mais surtout des règles pour obtenir le fichier *APK* final.

Gradle est un ensemble de fichiers de script écrits dans le langage *Groovy*[9]. L'ensemble des fichiers de script est visible dans la vue Android du projet. Ces fichiers ont été générés automatiquement par *Android Studio*.

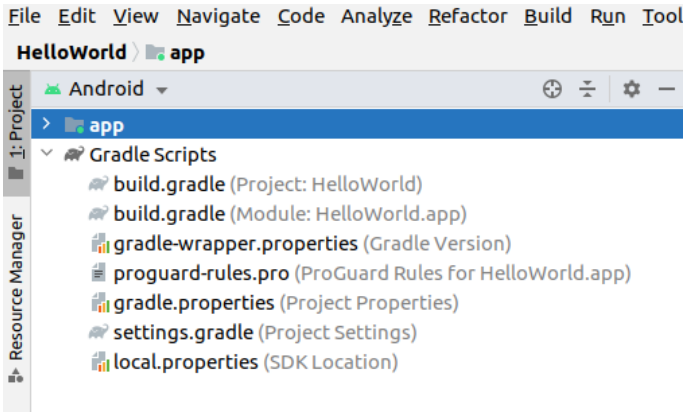

FIGURE 16 – *Gradle* - le module de gestion de la compilation

La figure 16 présente la vue Android des fichiers *Gradle*. Ces fichiers de script sont définis soit au niveau de l'application, soit au niveau du projet. En effet, un même projet Android peut contenir plusieurs applications, où chacune d'elles possède des règles de compilation différentes. Les scripts relatifs au projet sont valables pour toutes les applications qui le composent.

Les scripts définis au niveau du projet sont : *build.gradle*, *settings.gradle*, *gradle.properties*, *gradle-wrapper.properties* et *local.properties*.

Pour tous modules déclarés dans le fichier *settings.gradle*, il y a deux fichiers de script (niveau application) : *build.gradle* et *proguard – rules.pro*.

3.3.1 *settings.gradle* (niveau projet)

contient la liste des modules à traiter. Voici un exemple de script :

```
ootProject.name = "Test1"
include ':app'
include ':module2'
```

3.3.2 *local.properties*

contient l'emplacement du *SDK*. Voici un exemple de script :

```
sdk.dir=/home/user/Android/Sdk
```

9. https ://docs.gradle.org/current/userguide/userguide.html

3.3.3 *build.gradle*

contient la configuration générale des règles de compilation. Voici un exemple de contenu du fichier :

```
buildscript {
    repositories {
        google()
        jcenter()
    }
    dependencies {
        classpath "com.android.tools.build:gradle:4.2.0-alpha16"
    }
}
allprojects {
    repositories {
        google()
        jcenter()
    }
}
task clean(type: Delete) {
    delete rootProject.buildDir
}
```

Dans le premier bloc *buildscript*, le *classpath* indique quel plugin appelé pour la compilation. Ce plugin se trouve dans les dépôts (*repository*) *google* et *jcenter*.

Le bloc *allprojects* précise la configuration de l'ensemble du projet. Ici *google* et *jcenter* seront les dépôts du projet.

Le bloc *task* contient des tâches à effectuer.

3.3.4 *build.gradle* (niveau application)

Voici un exemple de fichier :

```
plugins {
  id 'com.android.application'
}
android {
  compileSdkVersion 30
    defaultConfig {
      applicationId "com.example.test1"
      minSdkVersion 21
      targetSdkVersion 30
      versionCode 1
      versionName "1.0"
      testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
  }
  buildTypes {
    release {
      minifyEnabled false
      proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'),
                                           'proguard-rules.pro'
```

```
    }
  }
  compileOptions {
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
  }
  buildFeatures {
    viewBinding true
  }
}
dependencies {
  implementation 'androidx.appcompat:appcompat:1.2.0'
  ...
}
```

Le bloc d'instructions *plugins* précise le *plugin* qui doit être utilisé.

Le bloc *android* contient quant à lui les informations de compilation. Le bloc $defaultConfig$ contient les informations nécessaires à la publication éventuelle du module. Le bloc *buildTypes* spécifie les informations relatives à la construction de l'application. Il y a deux types de construction possibles : *release* (version finale) ou *debug* (version en mode *debug*). Le bloc *compileOptions* contient la version de *Java* utilisée. Le bloc *buildFeatures* permet d'activer la *liaison de vue* du module [10].

La dernier bloc *dependencies* liste l'ensemble des dépendances de l'application. Le programmeur sera amené à le compléter si nécessaire.

3.4 Spécification des ressources

Les ressources que peut utiliser une application Android sont par exemple des images, des sons ou des pages XML destinées à l'IHM[11]. Ces ressources doivent être intégrées au code binaire qui sera téléchargé ; elles vont partie intégrante de l'application et sont stockées dans le répertoire *res* du projet. Chaque projet aura ses propres ressources réparties dans les répertoires suivants :

- $res/anim$: animations de transitions au format XML
- $res/color$: définitions des couleurs au format XML.
- $res/drawable$: images (format *png*, *jpg* ou gif), dessins au format XML
- $res/layout$: interfaces graphiques au format XML
- $res/menu$: menus de l'application au format XML
- $res/mipmap$: icônes de l'application
- res/raw : données brutes
- $res/values$: valeurs de types simples contenus dans des fichiers XML

Comme il n'est pas possible d'ajouter des sous répertoires dans les répertoires existants, il est conseillé d'adopter une convention de nommage des ressources afin de s'y retrouver.

On accède à ces ressources à l'aide de la classe R (pour *Resource*). Chaque ressource est accessible via son type et son nom explicites : pour un fichier, le type correspond au nom du fichier sans l'extension. Depuis les fichiers XML, on accède aux ressources à l'aide de la syntaxe : @[*paquetage.*]*type/nom* . Depuis les classes *Java*, on utilisera la syntaxe : [*paquetage.*] *R.type.nom* . Dans les deux cas, le nom du paquetage est optionnel. Par défaut, ce sera le paquetage courant.

Voici quelques exemples d'accès aux ressources :

10. Fonctionnalité qui permet de générer une classe de liaison pour chaque fichier de mise en page du module.
11. https ://developer.android.com/guide/topics/resources/providing-resources

depuis le code *Java*	depuis un fichier *XML*
R.drawable.monIcone	@drawable/monIcone
R.layout.fichierPrincipal	@layout/fichierPrincipal
R.string.maChaine	@string/maChaine
R.int.monEntier	@int/monEntier

4 Composants logiciels

4.1 Les activités

Une activité représente un écran unique de l'application [12]. Il y aura donc autant d'activités dans une application que d'écrans, mais une seule activité sera active à la fois.

Une activité sera lancée soir directement par l'utilisateur, soit par une autre activité par le mécanisme d'intention qui sera vue en partie 4.2. Dans tous les cas, toutes les activités devront être spécifiées dans le fichier *manifeste*.

4.1.1 Déclaration d'une activité

Une activité est une sous classe de *Activity* [13]. Une activité correspond à la fenêtre. L'activité principale (c'est à dire l'activité de démarrage) doit nécessairement être déclarée comme telle dans le fichier manifeste (voir partie 3.1). La balise $<activity>$ qui déclare l'activité doit pour cela inclure la balise $<intent-filter>$ de la manière suivante :

```
<activity
  android:name=".MainActivity"
  android:icon="@mipmap/ic_launcher"
  android:label="@string/app_name">
    <intent-filter>
      <action android:name="android.intent.action.MAIN"/>
      <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
</activity>
```

Les trois attributs principaux de la balise $<activity>$ sont *name*, *icon* et *label*. L'attribut *name* précise le nom de l'activité concernée. On peut voir ici que le nom du paquetage n'est pas spécifié en entier (juste un $'.'$) car il est déjà précisé dans l'attibut *package* de la balise $<manifeste>$. L'attribut *icon* précise le nom de la ressource qui sera utilisée comme image. L'attribut *label* précise quant à lui le titre de l'application.

Les attributs *icon* et *label* peuvent ne pas être précisés ; ce sera dans ce cas les valeurs données dans la balise $<application>$ qui seront prises.

Dans la balise $<intent-filter>$, la balise *action* indique que l'activité est le point d'entrée de l'application, et la balise *category* indique que cette activité peut être lancée par l'utilisateur. Pour plus de détails sur les actions et les catégories, voir la partie 4.2.1.

4.1.2 Le cycle de vie d'une activité

On parle de cycle de vie pour indiquer les différents états successifs dans lesquels une activité se trouvera. La figure 4.1.2 présente ces différentes étapes.

12. https ://developer.android.com/guide/components/activities/intro-activities
13. https ://developer.android.com/reference/android/app/Activity.html

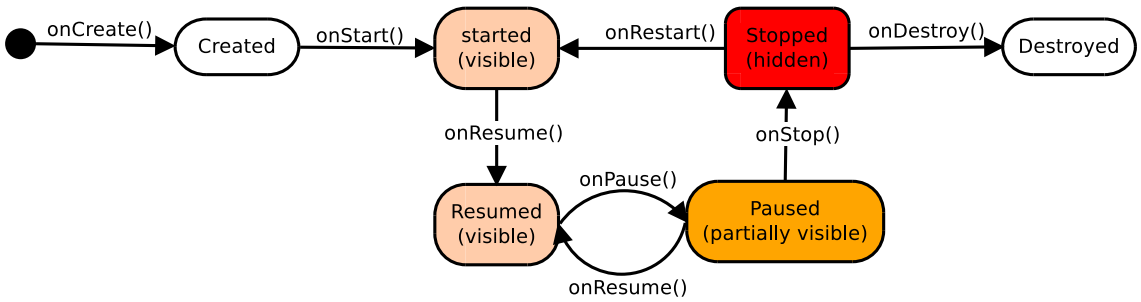

FIGURE 17 – Cycle de vie d'une activité

Une activité peut être active mais peut aussi donner la main à une autre activité. Dans ce cas, elle peut être présente mais non visible : elle peut être soit en pause, soit même être arrêtée.

Chaque transition du cycle de vie correspond à une méthode de la classe *Activity*. La méthode *onCreate*() devra être nécessairement redéfinie, les autres méthodes le seront ou non en fonction des besoins. Il n'est pas nécessaire de faire de contructeur car celui-ci ne sera pas appelé.

La méthode *onCreate*(*Bundle b*) ne sera appelée qu'une seule fois lors du lancement de l'application. Son rôle est d'initialiser le composant comme par exemple créer les vues.

```
1  protected void onCreate(Bundle savedInstanceState) {
2        super.onCreate(savedInstanceState);
3        binding = ActivityMainBinding.inflate(getLayoutInflater());
4        setContentView(binding.getRoot());
5        ...
6        Log.v("Mon activité ", "mon message dans les logs");
7  }
```

Le paramètre de type *Bundle* [14] contient toutes les données qui ont été sauvegardées lors de la dernière exécution de l'activité. La méthode *setContentView*() (ligne 4) crée la vue de l'activité.

On peut dans cette méthode *onCreate*() terminer tout de suite l'activité en appelant la méthode *finish*() sans que les autres méthodes (*onStart*(), ...) soient appelées, et donc sans que l'activité soit affichée.

La méthode *onStart*() est appelée au démarrage de l'application et à chaque fois que l'activité a été stoppée (*i.e.* après un *onStop*()). Cette méthode précède l'affichage de la vue.

La méthode *onResume*() est appelée ensuite pour afficher l'activité qui sera donc au premier plan.

La méthode *onPause*() permet de mettre en pause l'activité qui est partiellement visible. Elle est appelée lorsqu'une autre activité prend la main.

La méthode *onStop*() est appelée lorsque l'activité n'est plus visible par l'utilisateur. Le système Android peut décider d'arrêter l'activité après l'appel à cette méthode, pour des raisons de place mémoire notamment. C'est dans cette méthode qu'il faudra donc gérer les sauvegardes des données persistantes avant l'arrêt de l'activité.

La méthode *onRestart*() est appelée si l'activité revient au premier plan. On pourra redemander les ressources qui auront pu être sauvegardées dans *onStop*().

14. https ://developer.android.com/reference/android/os/Bundle

La méthode *onDestroy*() est appelée après l'appel de $finish()$ ou directement par le système. L'activité après cette méthode sera définitivement supprimée. Il faudra dans cette méthode supprimer les ressources qui auront pu être allouées.

4.1.3 Sauvegarde et restauration des données

La sauvegarde et la restauration des données dans une activité est réalisée par l'objet *Bundle* que l'on voit apparaître dans la méthode *onCreate*().

Comme vu précédemment, la sauvegarde des données persistantes n'est nécessaire que dans l'arrêt de l'activité, le fait de mettre en pause l'activité ne nécessitant pas de sauvegarde.

Il existe deux méthodes qui permettent respectivement de sauvegarder et restaurer cet objet *Bundle* :

```
protected void onSaveInstanceState (Bundle outState)
protected void onRestoreInstanceState (Bundle savedInstanceState)
```

La méthode *onSaveInstanceState*() est appelée automatiquement avant la méthode *onStop*(). La méthode *onRestoreInstanceState*() est appelée après *onStart*() et avant *onResume*().

Dans l'exemple suivant, deux valeurs persistantes *monEntier* et *maChaine* sont sauvegardées et restaurées :

```
public class MainActivity extends Activity{
  private static final String CLEF1 = "com.example.test1.clef1";
  private static final String CLEF2 = "com.example.test1.clef2";
  private int monEntier;
  private String maChaine;
  ...
  protected void onSaveInstanceState(Bundle b){
    super.onSaveInstanceState(b);
    b.putString(CLEF1,maChaine);
    b.putInt(CLEF2,monEntier);
  }
  protected void onRestoreInstanceState(Bundle b){
    b.getString(CLEF1,maChaine);
    b.getInt(CLEF2,monEntier);
    super.onRestoreInstanceState(b);
  }
}
```

4.1.4 Pile d'activités

Pour gérer les appels des différentes activités au sein d'une même application, le système Android utilise une pile $FIFO$ d'activités. Une pile est créée pour chaque lancement d'applications.

Lorsque l'utilisateur revient à la page d'accueil de l'appareil Android, la pile de l'activité en cours d'exécution est sauvegardée par le système, puis restaurée lorsque l'utilisateur relance l'application, l'activité courante étant celle qui se trouve en haut de la pile. L'utilisateur retrouve ainsi l'application dans l'état dans lequel il l'avait quittée.

Cette pile est gérée directement par le système. Le programmeur n'a donc rien à faire. Cependant, dans certains cas, il est nécessaire de vider cette pile pour retrouver l'application dans l'état initial de lancement. Ceci peut être précisé dans l'intention qui permet de lancer une activité en y ajoutant le drapeau $Intent.FLAG\_ACTIVITY\_CLEAR$.

4.2 Les intentions

Les intentions permettent de faire le liens entre les composants applicatifs du système ; ce sont un des éléments clef de la programmation Android. Une intention est une description abstraite d'une action à effectuer via un composant du système. Ce composant peut être explicite (cible désignée explicitement) ou implicite (cible non désignée), la résolution étant dynamique lors de l'exécution dans le second cas. Une intention peut être mise à disposition pour l'application elle même ou pour toute autre application.

Une intention est un couple *action/données* où une action est associée aux données. Les actions sont des constantes ($ACTION\_MAIN$, $ACTION\_EDIT$, ou encore $ACTION\_PICK$), et les données sont des URI. D'autres informations optionnelles peuvent être précisées en fonction des cas : la catégorie de l'action, le type des données, le nom explicite d'une classe ciblée, et des informations supplémentaires (extra).

Lors du lancement de l'intention, le système recherche tous les composants pouvant répondre à la demande. Si plusieurs solutions sont possibles, le système demande à l'utilisateur de choisir.

D'autre part, afin que le système puisse savoir quel composant peut être lancé, il faut indiquer explicitement à quelle intention chaque composant peut répondre. Cela se fait par un filtre d'intention (classe $intent - filter$ [15]) que l'on doit déclarer dans le fichier *manifeste*.

4.2.1 Création d'une intention

Chaque intention sera une instance de la classe *Intent*. Pour avoir l'ensemble des informations de cette classe, reportez-vous à l'adresse :

https ://developer.android.com/reference/android/content/Intent.html

Voici les informations que peut contenir une intention :

Le nom du composant de type $ComponentName$ est un champ optionnel. Si le nom est indiqué, il s'agit d'une intention explicite, sinon elle est implicite. La valeur de ce champ correspond à un nom de classe complet avec le nom du paquetage (comme *com.example.test1*).

L'action de type *String* est un champ obligatoire. Il est possible de spécifier n'importe quel nom de l'action ; mais il est préférable d'utiliser les noms définis dans la classe *Intent* dont voici les plus courants :

- $ACTION\_VIEW$ pour montrer quelque chose à l'utilisateur (texte, image, ...) ; se lance avec *startActivity*()
- $ACTION\_SEND$ pour partager des données avec une autre application ; se lance avec *startActivity*()
- $ACTION\_MAIN$ pour spécifier le point d'entrée principale : aucune donnée n'est attendue.
- $ACTION\_EDIT$ pour fournir en accès en écriture à des données
- $ACTION\_PICK$ pour choisir un élément parmi un répertoire de données ; en sortie, l'élément sélectionné
- $ACTION\_DIAL$ pour composer un numéro de téléphone dont le numéro est donné ou non
- $ACTION\_ANSWER$ pour gérer un appel téléphonique entrant

Les données de type Uri [16] est un champ obligatoire [17]. La syntaxe d'une URI est la suivante :

```
<schéma>:<information> {? <requête>} { # <fragment>}
```

15. https ://developer.android.com/reference/android/content/IntentFilter
16. https ://developer.android.com/reference/android/net/Uri
17. *Uniform Ressource Identifier* est un identifiant d'une ressource qui permet de l'identifier de manière unique

La catégorie de type *String* est un champ optionnel. Voici une liste des catégories principalement utilisées :
- *CATEGORY_HOME* pour lancer l'écran principal (avec *ACTION_MAIN*)
- *CATEGORY_OPENABLE* pour afficher un ensemble de données pour que l'utilisateur puisse en sélectionner une.
- *CATEGORY_BROWSABLE* pour afficher un navigateur Web.
- *CATEGORY_LAUNCHER* pour indiquer au système que le composant est le composant principal.
- *CATEGORY_ALTERNATIVE* pour inclure le composant dans une liste d'actions que l'utilisateur peut effectuer.

Le type de *String* est un champ optionnel. Il correspond à un type *MIME* et est utilisé lorsque l'*URI* n'est pas suffisante pour caractériser les données.

Voici quelques exemples de type : *image/∗*, *image/png*, *text/html*, *text/plain*

Le composant de type *String* est un champ optionnel. Il permet de spécifier explicitement le composant que l'on veut appeler. Lorsque ce champ est rempli, aucune évaluation n'est effectuée par le système, et les autres champs deviennent facultatifs.

Les extras correspondent à un ensemble d'informations supplémentaires. Par exemple, si l'action est d'envoyer un courriel, une des informations supplémentaires sera de fournir un sujet, un corps de message, ...

Les ajouts d'informations se font avec la méthode *putExtra*(). Un *extra* correspond à un couple $<clef, valeur>$: la clef est une chaîne de caractère, et la valeur doit être sérialisable puisque transmise à un autre composant via le réseau (interface *Parcelable*).

Pour récupérer les données, on peut utiliser les méthodes *getExtras*(), *getStringExtra*(), *getIntExtra*() ... [18]

On peut également créer un objet *Bundle* [19].

Les *flags* sont de type *int* et spécifient comment l'intention doit être gérée.

4.2.2 Intention explicite

Une intention est explicite lorsque le programmeur veut appeler un composant défini. C'est utilisé lorsque l'on appelle un composant de manière interne à l'application, ou lorsque l'on veut appeler un service externe mais clairement identifié.

Dans l'exemple suivant, on appelle depuis *PremierComposant* le composant qui est défini dans la classe *SecondComposant* :

```
Intent intent = new Intent((Context) PremierComposant.this, SecondComposant.class);
startActivity(intent);
```

La méthode *startActivity*() lance l'exécution du composant. Dans le cas présent, le second composant ne renvoie rien au premier.

Il est possible de récupérer des données en faisant appel à la méthode *startActivityForResult*(). Le second composant renverra dans ce cas une intention au premier.

```
Intent intent = new Intent((Context) PremierComposant.this, SecondComposant.class);
startActivityForResult(intent, 0);
```

Le second paramètre de *startActivityForResult*() est de type entier et permet d'identifier l'appel. On récupère le résultat via la méthode *onActivityResult*(*int*, *int*, *Intent*) du premier composant.

18. *voir* https ://developer.android.com/reference/android/content/Intent
19. https ://developer.android.com/reference/android/os/Bundle

4.2.3 Intention implicite

Une intention implicite désigne non pas un composant mais une action à réaliser. C'est le système qui se charge de trouver les composants pouvant traiter cette action. S'il en existe plusieurs, le système demandera à l'utilisateur de choisir le composant.

Voici quelques exemples de déclaration d'intentions implicites :

```
Intent t1 = new Intent(Intent.ACTION_DIAL);
Intent t2 = new Intent(Intent.ACTION_DIAL, Uri.parse("tel: 0241966510"));
Intent t3 = new Intent(Intent.ACTION_VIEW, Uri.parse("https://www.esaip.org/"));
Intent t4 = new Intent(Intent.ACTION_SEND_TO, Uri.parse("mailto:palbers@esaip.org");
Intent t5 = new Intent(Intent.ACTION_VIEW, Uri.parse("file://home/user/esaip.mp3"));

Intent t6 = new Intent(Intent.ACTION_SEND);
String [] adresses = {"durand@esaip.org", "dupont@esaip.org", "martin@esaip.org"};
t6.setExtra(android.content.Intent.EXTRA_EMAIL, adresses);
```

De la même manière que pour les intentions explicites, l'appel au composant se fera par les deux méthodes : $startActivity(Intent)$ et $startActivityForResult(Intent, int)$.

4.2.4 Filtres d'intention

Les filtres d'intention sont destinés au système afin que le système puisse savoir à quelle action un composant peut répondre lors d'une demande d'intention implicite ou explicite.

C'est donc au programmeur de le spécifier explicitement dès qu'il a implémenté un composant. Cette déclaration de traitement d'intentions se fait dans le manifeste via la balise $<intent\text{-}filter>$.

Voici un exemple de déclaration d'un composant *activity* pour qu'il se lance au chargement de l'application :

```
<activity
  android:name=".MainActivity"
  ...
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
  </intent-filter>
</activity>
```

La grande différence entre intention explicite et intention implicite est que le système Android intervient dans la recherche de composants. La figure 18 présente le cycle de vie du traitement d'une intention implicite.

Première étape la demande d'indention est envoyée par le premier composant au système Android

Deuxième étape le système cherche tous les composants pouvant répondre à la demande. Si plusieurs composant répondent à la demande, le système Android demande à l'utilisateur de choisir.

Troisième étape le système démarre le composant choisi en appelant sa méthode $onCreate()$, et lui transmet la demande d'intention.

Il est possible de savoir s'il y a un composant qui répond à la demande en utilisant la méthode $resolveActivity()$ comme suit :

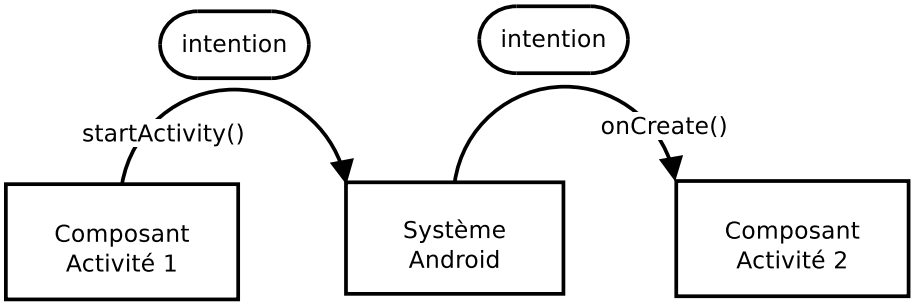

FIGURE 18 – Cycle de vie du traitement d'une intention implicite

```
Intent intent = new Intent(Intent.ACTION_SEND);
intent.putExtra(Intent.EXTRA_TEXT, "mon message");
intent.setType(HTTP.PLAIN_TEXT_TYPE);

// on vérifie que l'intention peut être traitéé par un composant:
if(intent.resolveActivity(getPackageManager()) != null){
    startActivity(intent);
}
```

La méthode $getPackageManager()$ permet de récupérer le $PackageManager$ où se trouvent toutes les informations relatives aux applications disponibles sur le système Android.

4.3 Les éléments de l'interface utilisateur

4.3.1 La mise en page : vue, *layout* et *widget*

L'interface utilisateur d'une application correspond à une structure d'éléments hiérarchique sous forme d'arbre [20]. L'élément de base est la vue (classe $View$). Les différents éléments de la vue sont regroupés dans un conteneur de vues (classe $ViewGroup$).

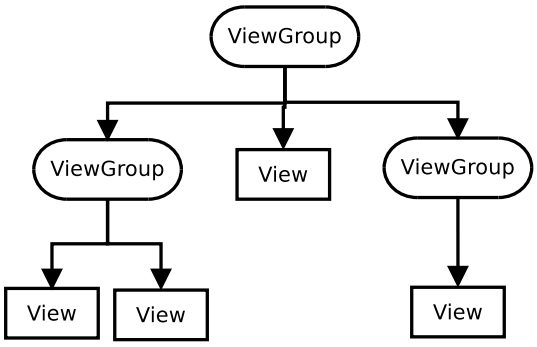

FIGURE 19 – Hiérachies des vues de l'interface utilisateur

Une vue possède les propriétés principales suivantes :

android.background couleur de fond de la vue (couleur ou image)

android.clickable la vue réagit au clic

android.id identifiant de la vue ; cet identifiant doit être unique pour tous les éléments de la vue

20. https ://developer.android.com/guide/topics/ui/declaring-layout

android.minHeight hauteur minimale

android.minWidth largeur minimale

android.onClick nom de la méthode à appelerlors d'un clic

android.tag objet associé à la vue

android.visibility visibilité de la vue

Les éléments de l'interface sont tous des sous classes de la classe $View$. On les appelle des *widgets* sous Android : des champs texte éditables ou non, des boutons, des menus, *etc*

Pour définir les conteneurs de vues, on utilise la classe *Layout* et ses sous-classes qui possèdent chacune une structure de mise en page spécifique. Les principales sous classes sont :

$FrameLayout$: les éléments sont dessinés les uns sur les autres à partir du coin supérieur gauche

$LinearLayout$: les éléments sont disposés les uns à la suite des autres horizontalement ou verticalement

$RelativeLayout$: les éléments sont positionnés les uns par rapport aux autres

$TableLayout$: les éléments sont disposés sont forme de tableau (à une dimension)

Pour définir une interface utilisateur plus simplement, la plate-forme *Android Studio* nous fournit un outil visuel assez pratique : le manager de mise en page (*Layout Manager*). Il permet de créer des fichiers XML dans lesquels on peut créer et placer directement des éléments graphique dans la page. Chaque fichier XML correspondra à un écran (donc à une activité).

On peut également créer l'interface directement via le code, apportant ainsi une plus grande flexibilité. Il y a donc deux manières de programmer une interface : le mode déclaratif qui utilise l'assistant de mise en page, et le mode programmatif.

4.3.2 Déclaration en mode déclaratif

Dans la vue *Android*, les fichiers XML de mise en page se trouve dans le répertoire *res/layout* du projet. Chaque activité ayant une interface doit avoir son fichier XML correspondant.

Créer un fichier XML se fait en cliquant droit sur *layout* et en choisissant $New/Layout\ ResourceFile$. Comme le montre la figure 20, il faut saisir un nom de fichier (les majuscules ne sont pas prises en compte), et préciser le nom du conteneur de vue racine, c'est à dire celui qui contiendra toutes les autres vues. Il est également possible préciser des quantificateurs pour ce dernier.

Le manager *layout* permet de modifier les paramètres du conteneur de vue racine, mais également d'ajouter des éléments graphiques. Il suffit de choisir dans la palette sa vue. En cliquant droit sur le nom de la vue, on peut ajouter la vue en cliquant sur *Add to design*.

En choisissant l'onglet *code* (en haut et à droite du manager), on peut voir directement le code XML. L'onglet *split* permet de voir à la fois le contenu du fichier et le rendu visuel. En fonction du layout racine choisi, on peut déplacer les éléments de la vue à la main sur l'interface. Les paramètres peuvent être changés soit directement dans le code, soit dans l'interface du manager.

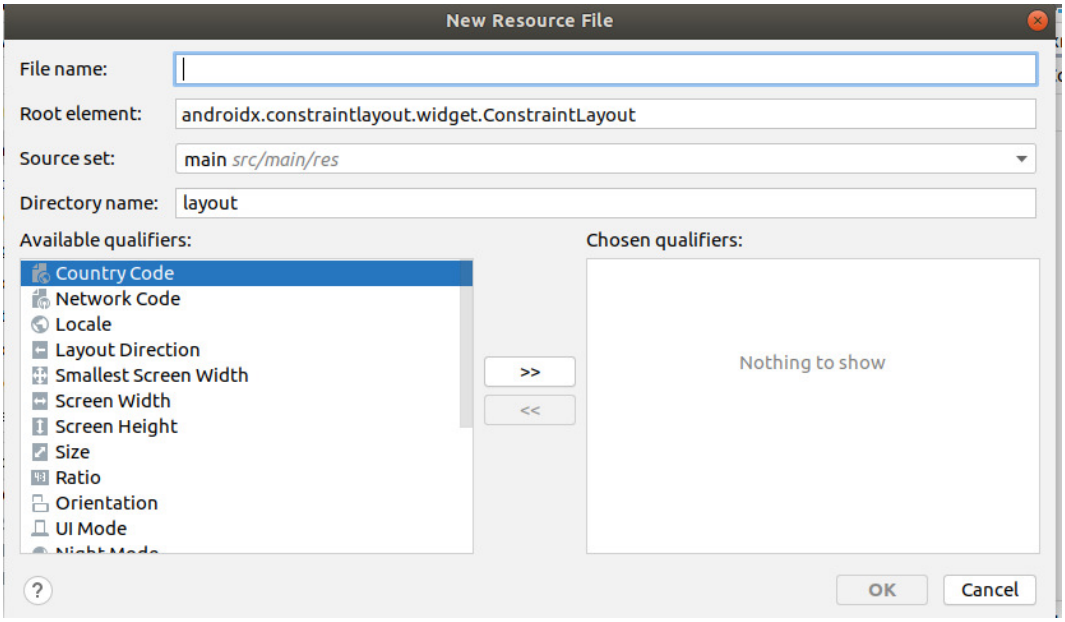

FIGURE 20 – Création de fichier *layout*

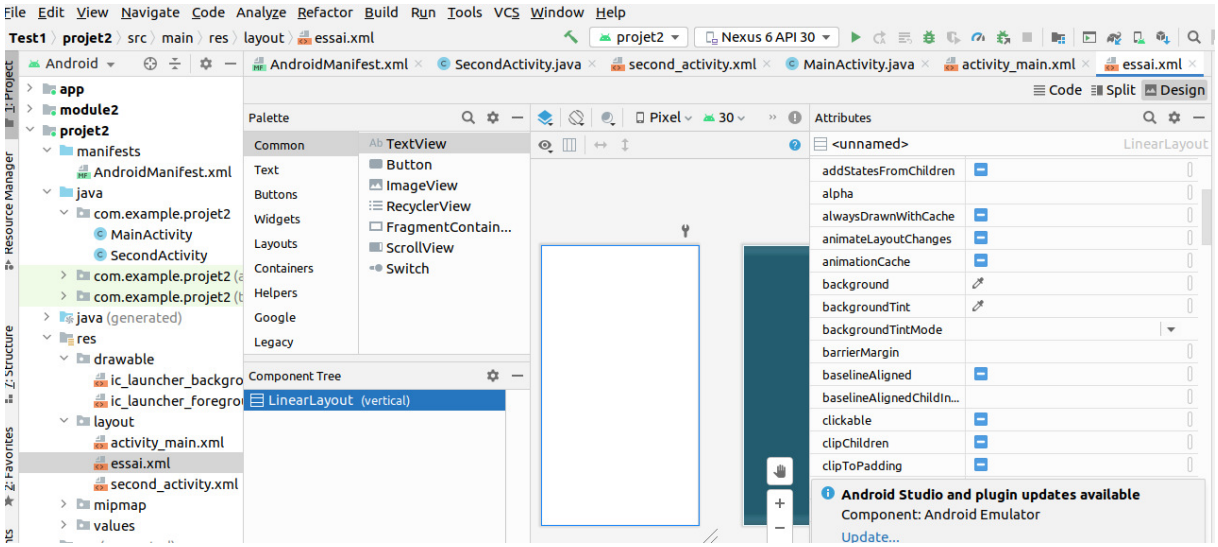

FIGURE 21 – Le manager *layout*

Le fichier XML doit avoir la syntaxe suivante :

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

Toutes les vues doivent avoir au minimum les propriétés *xmlns*, *android* : *layout_width* et *android* : *layout_height*. La largeur et la hauteur peuvent avoir comme valeur *match_parent* (dimension de la fenêtre parente), *wrap_content* (dimension de son contenu) ou un nombre à virgule flottante suivi d'une unité (100*dp* par exemple). Les vues sont ensuite ajoutées de manière imbriquée. On peut voir dans l'exemple précédent un champ texte de type $TextView$.

Une fois que la vue est définie, il faut créer l'activité correspondante si ce n'est pas déjà fait, puis lier cette activité avec le fichier XML que l'on vient de créer. On utilise pour cela la méthode $setContentView(int)$ qui prend en paramètre l'identifiant de la vue. On utilise le nom de ressource associée au fichier (voir partie 3.4). On pourra par exemple avoir le code suivant dans l'activité correspondante :

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

4.3.3 Déclaration en mode programmatique

La déclaration peut également se faire directement dans le code de l'activité. Il suffit pour cela de créer le conteneur racine et les différentes vues que le constitue. Il ne fait pas oublier d'associer ce contenuer racine à l'activité avec $setContentView()$.

Voici un exemple de code :

```
public class MainActivity extends AppCompatActivity {
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    LinearLayout layout = new LinearLayout(this);
    ViewGroup.LayoutParams v = new ViewGroup.LayoutParams(
                ViewGroup.LayoutParams.MATCH_PARENT,
                ViewGroup.LayoutParams.MATCH_PARENT);
    layout.setLayoutParams(v);
    setContentView(layout);
  }
}
```

Les paramètres du conteneur sont spécifiés à l'aide la méthode $setLayoutParams()$.

4.4 Les fragments

On peut considérer les fragments [21] comme des sous activités, où chaque fragment possède sa propre interface. Cela apporte de la souplesse dans le *design* des applications en partitionnant les éléments de l'interface. On peut aussi utiliser le même fragment dans plusieurs activités différentes.

Chaque fragment doit correspondre à une sous classe de $Fragment$ ou à l'une de ses sous-classes ($DialogFragment$, $ListFragment$ ou encore $PreferenceFragment$), et doit surcharger les méthodes si nécessaire.

Il est également possible d'ajouter et de supprimer des fragments *à la volée* dans une activité à l'aide du gestionnaire de fragments : $FragmentManager$ [22].

4.4.1 Cycle de vie

Le cycle de vie d'un fragment est similaire à celui d'une activité (partie 4.1.2). On retrouve les mêmes méthodes que dans une activité, ainsi que les méthodes spécifiques suivantes : $onActivityCreated()$, $onAttach()$, $onCreateView()$, $onDestroyView()$ et $onDetach()$.

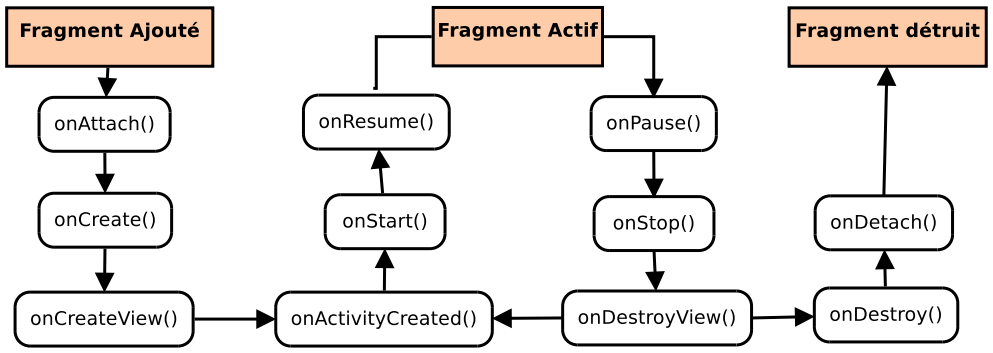

FIGURE 22 – Le cycle de vie des fragments

Phase de création

$onAttach(Context)$ est appelée lorsque le fragment a été associée à l'activité. Elle a en paramètre le contexte de l'application.

$onCreate(Bundle)$ est appelée lorsque le fragment est créé

$onCreateView(LayoutInflater, ViewGroup, Bundle)$ est appelée pour configurer l'interface. On peut considérer cette méthode comme l'équivalent de la méthode $onCreate()$ d'une activité.

Le paramètre de type $LayoutInflater$ permet d'ajouter des vues au fragment. Le paramètre de type $ViewGroup$ correspond au conteneur qui contiendra la vue du fragment. Et le paramètre de type $Bundle$ contient les sauvegardes effectuées auparavant dans la vie du fragment.

Cette méthode doit renvoyer la vue racine du fragment.

Phase de visualisation

$onActivityCreated(Bundle)$ permet de récupérer les données des vues et de les restaurer après l'arrêt du fragment. Ces données sont passées dans le paramètre de type $Bundle$.

$onStart()$ est appelée lorsque le fragment devient visible

21. https ://developer.android.com/guide/fragments
22. https ://developer.android.com/reference/android/app/FragmentManager.html

Phase d'interaction

onResume() est appelée lorsque commence l'interaction avec l'utilisateur

Phase de mise en pause

onPause() est appelée lorsque l'utilisateur n'interagit plus avec l'utilisateur

Phase d'arrêt

onStop() est appelée lorsque le fragment est arrêté

Phase de destruction

onDestroyView() est appelée pour détruire les ressources associées au fragment

onDestroy()() est appelé pour détruire le fragment

onDetach() est appelé pour détacher la fragment de l'activité

Remarque : l'instance de l'activité peut être récupérée à l'aide de la méthode *getActivity*() de la classe *Fragment*.

4.4.2 Création des fragments

Comme pour les activités, on peut créer un fragment en utilisant le *layout manager* (mode déclaratif) ou via le code (mode programmatique). Si aucune vue n'est associée au fragment, sa création ne peut se faire que de manière programmatique.

D'autre part, le mode déclaratif ne peut que créer les fragments de manière statique. La gestion dynamique des fragments (via *FragmentManager*) ne peut quant à elle être réalisée que de manière programmatique.

Comme pour les activités, on peut associer un fichier *layout XML* à chaque classe représentant un fragment. Contrairement aux activités, cette association ne peut pas se faire dans le code à l'aide de la méthode *setContentView*(). Il est nécessaire d'utiliser la méthode *inflate*() de la classe *LayoutInflater*. Une instance de cette classe est passée en paramètre de la méthode *onCreateView*() ; c'est cette instance qu'il faut utiliser pour associer la vue.

La méthode *inflate*(*XmlPullParser*, *ViewGroup*, *boolean*) possède trois paramètres. Le premier correspond à la vue, et le second à la vue racine. Le troisième paramètre indique si la vue doit être attachée ou non à la vue racine.

Voici un exemple de code du fichier *layout_fragment.xml* correspondant à la vue du fragment, fragment qui contient un champ *TextView* :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <TextView
        android:id="@+id/textView2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="Mon texte de fragment" />
</LinearLayout>
```

Voici le code de la classe :

```
package com.example.tp2;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.ViewGroup;
import android.app.Fragment;

public class MonFragment extends Fragment {
    public View onCreateView(LayoutInflater inflater,
                             ViewGroup container,
                             Bundle savedInstanceState) {
        return inflater.inflate(R.layout.layout_fragment, container, false);
    }
}
```

Il reste maintenant à ajouter le fragment à l'activité. Deux possibilités s'offrent à nous : soit en mode déclaratif, soit en mode programmatique.

Le mode déclaratif permet de décrire la vue directement en XML. Pour inclure un fragment dans le *layout* de l'activité, il suffit d'utiliser la balise $<fragment>$. Voici le code du fichier $layout.xml$:

```
<?xml version="1.0" encoding="utf-8"?>
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/fragment_fixe"
    android:name="com.example.tp2.MonFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

En plus des attributs habituels, il est nécessaire de spécifier l'attribut *name* qui doit contenir le nom complet de la classe (avec le nom du paquetage) du fragment.

Il reste à implémenter la classe de l'activité dont voici le code :

```
package com.example.tp2;
import android.app.Activity;
import android.os.Bundle;
public class ClassePrincipale extends Activity {
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.layout);
    }
}
```

Le mode programmatique permet quant à lui de rendre dynamique l'ajout et la suppression de fragments, contrairement au mode déclaratif qui rend l'activité figée.

Nous utiliserons dans cet exemple les mêmes fichiers $MonFragment.java$ et $layout\_fragment.xml$. Il n'est pas utile en effet d'utiliser un autre fragment. Pour rappel, un fragment peut être utilisé par deux activités différentes.

Il nous faut redéfinir par contre le *layout* de l'activité principale. Pour cela, il faut un conteneur de vue qui permet de définir l'ensemble des fragments qui vont constituer la vue de l'activité. Voici le code du fichier $layout2.xml$:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="match_parent"
android:layout_height="match_parent">
    <FrameLayout
        android:id="@+id/fragment_dynamique"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
</FrameLayout>
```

Le conteneur principal est ici un *FrameLayout* contenant un second *FrameLayout*. C'est ce dernier qui sera associé à la vue du fragment. Voici maintenant le code l'activité :

```
package com.example.tp2;
import android.app.FragmentManager;
import android.app.FragmentTransaction;
import android.os.Bundle;
import android.app.Activity;

public class ClassePrincipale2 extends Activity {
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.layout2);

        MonFragment fragment = new MonFragment();
        FragmentManager monManager = getFragmentManager();
        FragmentTransaction transaction = monManager.beginTransaction();
        transaction.add(R.id.fragment_dynamique, fragment);
        transaction.commit();
    }
}
```

Dans le code ci-dessus, la méthode *onCreate*() associe à *ClassePrincipale*2 le fichier *layout*2.*xml*, comme dans toute activité.

La gestion dynamique des fragments est réalisée dans un *gestionnnaire de fragment* que nous allons détailler dans la section suivante.

4.4.3 Le gestionnaire de fragment

Il existe un gestionnaire spécifique pour chaque activité [23]. La méthode *getFragmentManager* de la classe *Acitivity* permet de le récupérer à tout moment. Cette dernière méthode renvoie une instance de la classe *FragmentManager*.

Un gestionnaire de fragments contiendra plusieurs fragments sous la forme d'une pile de fragments. La prise en charge de cette pile est abordée dans la section suivante.

Les opérations concernant les fragments passent par des transactions, chaque transaction pouvant contenir une ou plusieurs opérations. Il faut donc ouvrir une transaction puis l'exécuter, ce que réalisent respectivement méthodes *beginTransaction*() et *commit*() de la classe *FragmentManager*. Ce qui donne la suite d'instructions suivantes :

23. https ://developer.android.com/reference/android/app/FragmentManager.html

```
FragmentManager monManager = getFragmentManager();
FragmentTransaction transaction = monManager.beginTransaction();
// opérations de la transaction...
transaction.commit();
```

Les opérations possibles dans une transaction sont l'ajout, le remplacement ou le retrait d'un fragment. La méthode *add* de la classe $FragmentTransaction$ [24] permet d'ajouter des fragments à une transaction. Elle prend en paramètre le fragment, ainsi que l'identifiant ou l'étiquette (*tag*) du conteneur qui contiendra le fragment. Trois méthodes *add* sont disponibles :

```
public abstract FragmentTransaction add (Fragment fragment,  String tag);
public abstract FragmentTransaction add (int containerViewId,  Fragment fragment);
public abstract FragmentTransaction add (int containerViewId, Fragment fragment, String tag);
```

Voici un exemple d'utilisation :

```
MonFragment fragment = new MonFragment();
transaction.add(R.id.fragment_dynamique, fragment);
```

Les méthodes *remove* et *replace* permettent respectivement de supprimer ou remplacer un fragment.

```
public abstract FragmentTransaction remove (Fragment fragment);
public abstract FragmentTransaction replace (int containerViewId, Fragment fragment, String ta
public abstract FragmentTransaction replace (int containerViewId, Fragment fragment)
```

Il est possible de retrouver un fragment dans le gestionnaire à l'aide des méthodes $findFragmentById()$ et $findFragmentByTag()$ qui sont définis dans la classe $FragmentManager$:

```
public abstract Fragment findFragmentById (int id);
public abstract Fragment findFragmentByTag (String tag);
```

Voici un exemple d'utilisation :

```
FragmentManager monManager = getFragmentManager();
MonFragment fragment =
    (MonFragment) monManager.findFragmentById("R.id.fragment_dynamique");
```

4.4.4 La pile de fragments

Le gestionnaire de fragments permet de stocker les fragments dans une pile. Il est donc possible de revenir facilement en arrière dans une interface en cliquant sur un bouton *retour*. Contrairement auw activités, c'est au développeur de gérer cette pile.

L'ajout d'un fragment dans cette pile est réalisée à l'aide de la méthode $addToBackStack()$ de la classe $FragmentManager$. Cette méthode peut prendre en paramètre une chaîne de caractère identifiant la transaction. Lorsque la méthode $commit()$ est appelée lors d'une transaction, les fragments remplacés ou supprimés sont ajoutés à la pile et les méthodes des fragments $onPause()$, $onStop()$ et $onDestroyView$ sont appelées.

```
public abstract FragmentTransaction addToBackStack (String name);
```

Les méthodes $popBackStack()$ et $popBackStackImmediate()$ de $FragmentManager$ permettent de dépiler les fragments. La première le fait de manière asynchrone alors que la seconde le fait immédiatement :

```
public abstract boolean popBackStackImmediate ();
public abstract boolean popBackStackImmediate (int id, int flags);
public abstract boolean popBackStackImmediate (String name, int flags);
```

24. https ://developer.android.com/reference/android/app/FragmentTransaction

4.4.5 La mise en page dynamique

On peut être amené à gérer différemment les fragments lorsque l'application s'exécute sur un *smartphone* ou sur une tablette. On peut en effet afficher deux fragments sur le même écran sur une tablette alors que ce n'est pas possible sur un *smartphone*.

Il est possible de différencier les *layouts* par rapport au terminal Android utilisé. C'est l'appareil lui même qui détermine quels *layouts* prendre. Typiquement, les *layouts* destinés aux *smartphones* sont dans le répertoire par défaut *./layout*, et les *layouts* des tablettes sont dans le répertoire *./layout−large*.

On pourra par exemple définir le code du *layout* de l'activité principale pour *smartphone* comme suit dans répertoire *./layout* :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:orientation="vertical">
    <FrameLayout
        android:id="@+id/fragment_dynamique"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
</LinearLayout>
```

Et le code du *layout* de l'activité principale pour tablette comme suit dans répertoire *./layout−large* :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:orientation="horizontal">
    <FrameLayout
        android:id="@+id/fragment1"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
    <FrameLayout
        android:id="@+id/fragment2"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
</LinearLayout>
```

Dans la version pour *smartphones*, il n'y a qu'un seul conteneur qui est utilisé, alors que dans le version tablette, deux conteneurs sont déclarés.

Dans le code, pour connaître le type de matériel sur lequel tourne l'application, il suffit de vérifier par exemple avec la méthode $findViewById$ de la classe $FragmentManager$ si tel ou tel conteneur est présent.

À noter que l'attribut *orientation* permet de forcer l'orientation de l'affichage en mode paysage ou portrait.

4.4.6 La bibliothèque *android.support.v4*

Les fragments ont fait leur apparition à partir de la version 3 d'Android. Pour éviter tout problème de compatibilité avec les anciennes versions, les concepteur d'Android ont introduit une bibliothèque supplémentaire : *android.support.v4*.

Cette bibliothèque est encore assez utilisée dans diverses APIs. Il est souvent recommandé de l'utiliser pour la manipulation des fragments. Les différences sont peu nombreuses et l'on peut voir une différence dans l'utilisation du mot *support* dans le nom des classes et des méthodes. On peut noter que la classe *Activity* est remplacée par *FragmentActivity*.

4.5 Les listes

Android fournit de nombreux composants graphiques dont le plus utilisé est la liste verticale d'éléments possédant une barre de défilement. La vue représentant une liste est le composant *ListView*. Au niveau du code, il est possible d'utiliser les classes fournies par Android comme *ListActivity* ou d'implémenter une liste en spécifiant un *layout* particulier.

Quelque soit l'implémentation choisie, la mise en place d'une liste nécessite un *layout* pour l'affichage des éléments, et un adaptateur pour gérer les éléments de la liste.

4.5.1 Le composant *ListView*

Le *layout manager* permet d'ajouter le composant *ListView* au même titre que n'importe quel autre composant. Il se trouve dans la palette *legacy*.

Voici un exemple de code du *layout* principal :

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <ListView
        android:id="@+id/maListe"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
</FrameLayout>
```

Il reste à implémenter le code des classes qui seront associées à cette vue.

4.5.2 Implémentation avec les classes standard

L'implémentation standard des listes se base sur l'utilisation des classes *ListActivity* [25] sous classe d'*Activity* ou *ListFragment* [26] sous classe de *Fragment*.

Voici le code de la classe *MonMenu* dont le but est d'afficher une liste de desserts :

```
package com.example.projet3;
import androidx.appcompat.app.AppCompatActivity;
import android.app.ListActivity;
import android.os.Bundle;
import android.widget.ArrayAdapter;
import android.widget.ListView;
public class MonMenu extends ListActivity {
    private final String[] desserts = new String[] {
            "crème brulée", "crumble aux framboises",
            "panna cotta", "tarte aux pommes"
    };
    protected void onCreate(Bundle savedInstanceState) {
```

25. https ://developer.android.com/reference/android/app/ListActivity
26. https ://developer.android.com/reference/android/app/ListFragment

```
        super.onCreate(savedInstanceState);
        super.setListAdapter(new ArrayAdapter<String>(
                this, android.R.layout.simple_list_item_1, desserts));
    }
}
```

On peut remarquer qu'il n'est pas nécessaire d'utiliser la méthode *setContentView*(). En effet, la classe *ListActivity* utilise un *layout* par défaut qui affiche une liste unique centrée au milieu de l'écran.

Pour afficher les données de la liste, il faut maintenant faire le lien entre les éléments de la liste et la vue qui les représente. C'est la méthode *setListAdapter*() de la classe *ListActivity* qui permet de faire ce lien.

Android fournit deux *layouts* prédéfinis :

— *android.R.layout.simple_list_item*1 qui correspond à la vue d'une liste dont chaque élément est un composant *TextView*,

— *android.R.layout.simple_list_item*2 qui correspond quant à lui à la vue de deux éléments de type *TextView* dont le deuxième est en dessous du premier avec une police plus petite.

Android fournit plusieurs classes qui permettent de faire cette adaptation. Ces classes implèmentent toutes l'interface *ListAdapter*.

— *ArrayAdapter*<*T*> pour des données d'un type générique ; les données sont fournies dans un tableau de type *T* dans le constructeur,

— *BaseAdapter* pour n'importe quelles données basiques,

— *CursorAdapter* pour des données provenant d'une base de données ; les méthodes *bindview* et *newView* doivent être implémentées. La première méthode est appelée pour lier les données à la vue, et la seconde méthode doit retourner la vue qui sera utilisée comme *layout*,

— et *SimpleCursorAdapter* pour des données de type *String* ou de type image.

4.5.3 Implémentation spécifique

L'implémentation standard reste limitée dans l'affichage des éléments. Il est possible d'associer à un composant *ListView* un *layout* spécifique en ne respectant qu'une seule contrainte : le composant *ListView* doit avoir un identifiant de type : @*android*:*id*/*list*.

Dans l'exemple ci-dessous, on définit deux composants dans le fichier *layout*, le premier étant un *TextView* et le second une *ListView* :

```
?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <TextView
        android:id="@+id/textView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:textSize="20"
        android:text="Liste des desserts" />
    <ListView
        android:id="@android:id/list"
        android:layout_width="match_parent"
```

```
        android:layout_height="match_parent"
        android:layout_weight="1" />
</LinearLayout>
```

Dans le code de l'activité, il faut associer cette fois-ci la classe avec le *layout* :

```
package com.example.projet3;
import androidx.appcompat.app.AppCompatActivity;
import android.app.ListActivity;
import android.os.Bundle;
import android.widget.ArrayAdapter;
import android.widget.ListView;
public class MainActivity extends ListActivity {
    private final String[] desserts = new String[] {
            "crème brulée", "crumble aux framboises",
            "panna cotta", "tarte aux pommes"
    };
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        super.setListAdapter(new ArrayAdapter<String>(
                this, android.R.layout.simple_list_item_1, desserts));
    }
}
```

L'intérêt de cette implémentation est qu'il devient possible de créer son propre *layout* pour afficher les éléments de la liste. Comme tout layout, il peut se définir avec le *layout manager*.

On peut également définir un adaptateur spécifique. En fonction de la super classe choisie, le programmeur sera amené à redéfinir les méthodes suivantes définies dans la classe abstraite *BaseAdapter* : $getView()$, $getItemId()$, $getItem()$ et $getCount()$.

4.5.4 Gestion des listes

Sélection d'un élément

La méthode $onListItemClick(ListView, View, int, long)$ de la classe *ListActivity* est appelée lorsqu'un élément de la liste est sélectionné. Il suffit alors de surcharger cette méthode pour prendre en charge le clic d'un élément dans la liste.

Cette méthode prend en paramètre l'objet $ListView$ ayant généré l'événement, la vue qui a été cliquée, la position de l'élément cliqué dans la liste et l'identifiant de cet élément.

On peut également gérer le clic long sur un élément. Mais comme il n'existe pas de méthode dans la classe *ListActivity*. Il faut alors aller chercher la méthode $setOnItemLongClickListener()$ de la classe $ListView$.

Cette méthode prend en paramètre une variable de type $AdapterView.OnItemLongClickListener$ pour laquelle il faut redéfinir la méthode :

$onItemLongClick(AdapterView <?> \ parent, \ View \ view, \ int \ position, \ long \ id)$

Cette méthode renvoie un booléen indiquant si oui ou non le clic long a été entièrement géré. La récupération de $ListView$ de l'activité est réalisée à l'aide de la méthode $getListView()$ de *ListActivity*.

Voici un exemple de code permettant d'afficher le numéro de l'élément sélectionné dans une liste :

```
public class MainActivity extends ListActivity {
    private final String[] desserts = new String[] {
            "crème brulée", "crumble aux framboises",
            "panna cotta", "tarte aux pommes"
    };
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        super.setListAdapter(new ArrayAdapter<String>(
                this, android.R.layout.simple_list_item_1, desserts));

        // gestion du clic long:
        getListView().setOnItemLongClickListener(
                new AdapterView.OnItemLongClickListener() {
                    @Override
                    public boolean onItemLongClick(AdapterView<?> parent,
                                                   View view, int position, long id) {
                        String s = "click LONG de l'item numéro " + position;
                        Log.i("onItemLongClick", s);
                        return true;
                    }
                }
        );
    }

    // gestion du simple clic:
    public void onListItemClick(ListView liste, View vue, int position, long id){
        String s = "click de l'item numéro " + position;
        Log.i("onListItemClick", s);
    }
}
```

Mise à jour dynamique

Il est possible d'ajouter ou de supprimer des éléments de manière dynamique dans une liste. Il existe des méthodes dans la classe *ArrayAdapter* qui permettent de modifier directement la liste :

add(*T object*) ajoute un élément en fin de liste

addAll(*T objects*) ajoutent plusieurs éléments en fin de liste

clear() retire tous les éléments de la liste

remove(*T object*) retire un élément de la liste

Ces méthodes ne pourront être appelées que si et seulement si les éléments sont immuables (c'est à dire des objets qui ne peuvent être modifiés, comme les objets de type *String* par exemple). Dans l'exemple suivant, afin d'avoir une liste immuable, on utilise une variable de type $Array < List >$ pour initialiser l'adaptateur ; en effet, l'initialisation de ce dernier par un tableau ne conviendrait pas, car les tableaux ne sont pas des objets immuables.

L'idée de l'exemple ci-dessous est d'ajouter un nouvel élément en cliquant sur *monBouton* :

```
public class MainActivity extends ListActivity {
    private final String[] desserts = new String[] {
            "crème brulée", "crumble aux framboises",
            "panna cotta", "tarte aux pommes"
    };
    protected ArrayAdapter<String> monAdaptateur;
    private ArrayList<String> listeDesserts;

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

        listeDesserts = new ArrayList<String>();
        for(int i=0; i<4; i++)
            listeDesserts.add(desserts[i]);
        this.monAdaptateur = new ArrayAdapter<String>(
                this, android.R.layout.simple_list_item_1, listeDesserts);
        super.setListAdapter(this.monAdaptateur);

        Button monBouton = findViewById(R.id.monBouton);
        monBouton.setOnClickListener(new View.OnClickListener() {
             public void onClick(View v) {
                // ajout d'un nouvel item à la liste:
                monAdaptateur.add(new String("nouvel item"));
            }
        });
     }
}
```

De manière générale, il est possible de modifier la liste directement et de mettre à jour l'adaptateur en appelant la méthode $notifyDataSetChanged()$ de la sur classe de tous les adaptateurs : *BaseAdapter*. Il est cependant déconseiller de l'utiliser trop souvent, car elle est gourmande en ressource. Le code de la méthode pour ajouter un nouvel item serait :

```
Button monBouton = findViewById(R.id.monBouton);
monBouton.setOnClickListener(new View.OnClickListener() {
        public void onClick(View v) {
             listeDesserts.add("nouveau");
             monAdaptateur.notifyDataSetChanged();
        }
});
```

4.5.5 Listes déroulantes

Android met à disposition la classe *Spinner* ainsi que le layout *<Spinner>* pour implémenter une liste déroulante. Leur utilisation est similaire à $ListView$; en particulier, il faut utiliser les mêmes adaptateurs qui implémentent tous l'interface *SpinnerAdapter*.

La spécification du *layout* pour afficher les éléments de la liste déroulante s'effectue à l'aide de la méthode $setDropDownViewResource(int)$, qui prend en paramètre l'identifiant du *layout*. Android fournit le *layout* par défaut : *simple_spinner_dropdown_item*.

Voici un exemple d'utilisation :

```
<?xml version="1.0" encoding="utf-8"?>
<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"
    tools:context=".MainActivityListeDeroulante">
    <Spinner android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/liste_deroulante" />
</LinearLayout>

public class MainActivityListeDeroulante extends Activity {
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main_liste_deroulante);

        Spinner listeDeroulante = (Spinner)findViewById(R.id.liste_deroulante);
        ArrayAdapter adapter = new ArrayAdapter(this,
                android.R.layout.simple_list_item_1,
                new String[]{"Lundi","Mardi","Mercredi","Jeudi","Vendredi",
                                "Samedi","Dimanche"});
        adapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
        listeDeroulante.setAdapter(adapter);
    }
}
```

4.6 Menus et barres d'actions

Il existe trois types de menus dans les applications Android :

— les menus d'options qui sont disponibles au niveau de l'application et que l'on retrouve dans les activités,

— les menus contextuels qui sont attachés à un élément comme une image, un texte, ou encore un élément d'une liste,

— et les menus *pop-up* qui sont attachés à un composant mais qui ne proposent pas d'actions concernant pas directement le composant lui même.

Les menus d'option apparaissent nécessairement dans une barre d'action située généralement en haut de l'écran. Ils peuvent être directement visibles dans la barre d'action, ou apparaître après un clic sur trois petits points verticaux apparaissant dans la barre lorsque l'espace pour l'afficher n'est pas suffisant.

4.6.1 Déclarations des menus

Les menus sont déclarés dans un fichier *layout* qui doit être placé dans le répertoire *res/menu*, à l'aide de la balise *<menu>*. Ses principaux attributs sont :

android : *id* l'identifiant du menu

android : *title* le titre du menu à afficher

android : *icon* l'image qui servira à représenter le menu, et qui se trouve dans *res/drawable*

android : *onClick* le nom de la méthode qui sera appelée lorsque l'utilisateur cliquera sur le menu

android : *checkable* ce champ indique si oui ou non une case à cocher apparaît dans le menu

android : *showAsAction* ce champ indique de quelle manière le menu figurera dans la barre d'action (voir plus loin) ; il peut prendre les valeurs : $[never|ifroom|allways]$ | $[withtext]$
- *never* : le menu n'est pas présent
- $ifroom$: le menu est présent s'il y a assez de places à l'écran
- *always* : le mnu est toujours présent
- *withtext* : indique que le titre du menu sera affiché ; cette option est combinable avec les trois actions précédentes en utilisant le caractère $'|'$.

Les éléments du menu seront déclarés avec la balise *<item>*. Il est possible de déclarer un sous menu dans un *item*.

La balise *<group>* permet de regrouper des éléments d'un menu qui possèdent les mêmes caractéristiques : on peut ainsi rendre visible ou invisible un groupe d'éléments du menu, les activer ou les désactiver. Les éléments d'un groupe sont au même niveau que les autres éléments du menu (*i.e.* il ne s'agit pas d'un sous menu).

Dans l'exemple ci-dessous, le deuxième item est constitué d'un second menu contenant deux items :

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/monItem1"
        android:title="menu 1"
        android:showAsAction="always" />
    <item android:id="@+id/monItem2"
        android:title="menu 2"
        android:showAsAction="always">
        <menu>
            <item android:id="@+id/monItem3"
            android:title="Item 3"
            android:showAsAction="always" />
            <item android:id="@+id/monItem4"
                android:title="Item 4 "
                android:showAsAction="always" />
        </menu>
    </item>
</menu>
```

4.6.2 Les barres d'actions

Les barres d'action sont depuis la version 7 d'Android (*lollipop*) des composants *widgets* à part entière, et doivent donc être déclarées dans le *layout* d'une activité. C'est l'objet *ToolBar* qui doit être utilisé.

Toute barre d'action contient par défaut l'icone de l'activité (ou son titre) ainsi que le menu de l'activité. Il est possible d'ajouter d'autres *widgets* comme une barre de recherche, des images ou encore des listes de navigation.

On utilise la balise *<ToolBar>* pour déclarer une barre d'action comme suit [27] :

27. Attention, il existe une autre version dans la bibloithèque de support *v.7*.

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <android.widget.Toolbar
        android:id="@+id/maBarreOutils"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        android:background="?attr/colorPrimary" />
</LinearLayout>
```

Dans l'exemple ci-dessus, la hauteur de la barre d'action correspond à *actionBarSize* qui est une constante définie dans Android. De même, la couleur de fond de la barre d'action correspond à la couleur par défaut *colorPrimary*.

Il faut dans le code de l'acitivité déclarer un objet de type $ToolBar$. L'association de la barre d'action à l'activité est effecutée comme tous les autres *widgets* à l'aide de la méthode $findViewById()$. La prise en compte de la barre dans l'activité est réalisée à l'aide de la méthode $setActionBar()$. Voici un exemple de code pour l'utilisation de la barre d'action dans une activité :

```
import android.app.Activity;
import android.os.Bundle;
import android.view.Menu;
import android.widget.Toolbar;
public class MainActivityMenu extends Activity {
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main_menu);

        Toolbar toolbar = findViewById(R.id.maBarreOutils);
        super.setActionBar(toolbar);
    }
}
```

4.6.3 Les menus d'option

Les menus d'options (ou menus d'activités) doivent être définis à l'aide de la surcharge de la méthode $onCreateOptionsMenu(Menu)$ de l'activité. Le paramètre correspond au menu à construire. Cette méthode doit renvoyer *true* afin que le menu apparaisse dans la barre d'action.

L'association avec le fichier *layout* correspondant au menu est réalisée à l'aide de la méthode $inflate(int, Menu)$ sur l'objet $MenuInflater$ de l'activité, que l'on récupère à l'aide de la méthode $getMenuInflater()$. Voici un exemple de code : ...

```
public boolean onCreateOptionsMenu(Menu menu) {
        MenuInflater menuDeMonActivite = super.getMenuInflater();
        menuDeMonActivite.inflate(R.menu.layout_menu, menu);
        return true;
}
```

Pour indiquer quelle action effectuée lorsqu'un item du menu est choisi par l'utilisateur, il existe de deux manières différentes : soit directement dans le fichier *layout* en donnant comme valeur à l'attribut *onClick* le nom de la méthode à appeler, soit en surchargeant la méthode *onOptionsItemSelected*() de l'activité.

Il faut dans tous cas définir tout d'abord la méthode à appeler dans l'activité. Cette méthode doit être publique, ne rien renvoyer et avoir en paramètre l'item du menu qui a été cliqué.

```
public void methodeAppelerPourLItem1(MenuItem item){
   // actions à effectuer lors du choix de l'item
   Log.i("methodeAppelerPourLItem1", "L'item " + item.toString() +
           " a bien été cliqué");
}
```

1. Il faut soit spécifier l'attribut *onClick* dans le *layout* de menu :

   ```
   <menu xmlns:android="http://schemas.android.com/apk/res/android">
       <item
           android:id="@+id/monItem1"
           android:title="menu 1"
           android:showAsAction="always"
           android:onClick="methodeAppelerPourLItem1"/>
   </menu>
   ```

2. ou alors surcharger la méthode *onOptionsItemSelected*(*Item*) dans l'activité. Cette méthode doit renvoyer *true* si le clic de l'item a bien été traité. Par défaut, on peut appeler la méthode de la sur-classe qui pourra traiter tous les cas par défaut possibles.

   Cette méthode prend en paramètre l'item qui a été choisi par l'utilisateur. Comme elle est appelée pour tous les items du menu d'options, il faut regarder la valeur de ce paramètre pour déterminer quel item a été choisi.

   ```
   public boolean onOptionsItemSelected(MenuItem item){
     if(item.getItemId() == R.id.monItem2){
       Log.i("onOptionsItemSelected","L'item " + item.toString() +
               " a bien été cliqué");
       return true;
     }
     return super.onOptionsItemSelected(item);
   }
   ```

Les menus contextuels

Ils apparaissent lors d'un clic long sur la vue à laquelle ils sont associés. La première étape est d'indiquer à l'activité que la vue doit gérer les clics longs ; il faut pour cela appeler la méthode $registerForContextMenu(View)$ de l'activité où le paramètre correspond à la vue qui doit gérer les clics longs. Cette opération pourra être faire dans la méthode d'initialisation $onCreate()$. Tous les menus contextuels utilisés dans l'acitivté devront être associés de cette manière.

C'est la méthode $onCreateContextMenu(ContextMenu, View, ContextMenuInfo)$ de l'activité qui sera appelée, lorsqu'un clic long est effecuté. Il suffit donc de surcharger cette méthode pour indiquer au système quelle action faire, en l'occurrence ici l'affichage d'un menu. Cette méthode prend en paramètre le menu contextuel en cours de création, la vue qui a été cliquée, et des informations supplémentraires sur l'élément cliqué. Ainsi, pour savoir sur quelle vue le clic long a été effectué, il suffit donc de regarder l'identifiant du deuxième paramètre.

De la même manière que pour les menus d'options, le menu sera créé en utilisant la méthode $inflate()$ du menu de l'activité. Voici un exemple de code :

```
import android.app.Activity;
import android.os.Bundle;
import android.view.ContextMenu;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.View;
import android.widget.TextView;
public class MainActivityMenu extends Activity {
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main_menu);
        TextView texte = findViewById(R.id.texteInfo);
        super.registerForContextMenu(texte);
    }
    public void onCreateContextMenu(ContextMenu monMenu, View maVue,
                                    ContextMenu.ContextMenuInfo monMenuInfo){
        super.onCreateContextMenu(monMenu, maVue, monMenuInfo);
        if( maVue.getId() == R.id.texteInfo){
            MenuInflater menuDeMonActivite = super.getMenuInflater();
            menuDeMonActivite.inflate(R.menu.layout_menu_contextuel, monMenu);
        }
    }
}
```

De la même manière que pour les menus d'option, pour déterminer quel item a été choisi dans un menu contextuel, il est nécessaire de surcharger la méthode $onContextItemSelected(MenuItem)$ de l'acitivté. Cette méthode prend en paramètre l'item sélectionné et renvoie *true* dans le cas où l'item a bien été traité.

```
public boolean onContextItemSelected(MenuItem item){
   if(item.getItemId() == R.id.monItemContextuel1){
      Log.i("onContextItemSelected","L'item " + item.toString() +
              " a bien été cliqué");
      return true;
   }
   return super.onContextItemSelected(item);
}
```

4.6.4 Les menus *pop-up*

Ces menus ne sont pas quant à eux associés à l'appel d'une méthode pour leur construction. C'est au programmeur d'instancier un objet *menu* au moment voulu, ce qui se fait à l'aide de la méthode $inflate()$ de l'objet $MenuInflater$ comme pour les autres types de menus.

Pour obtenir un menu *pop-up*, il faut déclarer une instance de la classe $PopupMenu$. Le constructeur de cette classe prend deux paramètres : le contexte de l'action (i.e. l'activité), et la vue à laquelle est attaché le menu *pop-up*.

Une fois que le menu est déclaré et associé à une vue, le programmeur va pouvoir l'afficher en appelant la méthode $show()$ de la classe $PopupMenu$. Dans l'exemple ci-dessous, le menu est affiché dès que le bouton auquel on a attaché ce même menu est cliqué :

```
public class MainActivityMenu extends Activity {
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main_menu);

        Button monBouton = findViewById(R.id.monBouton);
        PopupMenu monMenuPopup = new PopupMenu(this, monBouton);
        MenuInflater convertisseur = monMenuPopup.getMenuInflater();
        convertisseur.inflate(R.menu.mon_popup_menu, monMenuPopup.getMenu());

        monBouton.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                monMenuPopup.show();
            }
        });
    }
}
```

5 Persistance des données

Il est très souvent nécessaire de sauvegarder des données d'un lancement à l'autre d'une application : on parle de persistance des données. Les solutions pour sauvegarder des données sont soit d'utiliser des fichiers, soit d'utiliser une base de données lorsque ces données sont plus importantes.

5.1 Fichiers de préférences

Android utilise des fichiers de préférences qui permettent de sauvegarder des données de type primitif *Java* sous la forme de couples *clef/valeur* au format Xml.

Un fichier de préférences est associé par défaut à chaque activité, mais il est possible d'avoir plusieurs fichiers associés à une activité. La gestion des préférences est faite par l'intermédiaire de la classe $SharedPreferences$ [28], que l'on peut invoquer via la méthode $getSharedPreferences(String, int)$.

Le premier paramètre est optionnel et correspond au nom du fichier de préférences. S'il n'est pas renseigné, c'est le fichie de préférences par défaut qui est choisi.

Le second paramètre correspond au mode d'accès du fichier. Il existe trois valeurs possibles :

$Context.MODE\_PRIVATE$: le fichier ne peut être accessible que par l'activité courante,

$Context.MODE\_WORLD\_READABLE$: toutes les applications peuvent lire le fichier,

$Context.MODE\_WORLD\_WRITEABLE$: toutes les applications peuvent modifier le fichier.

Les données sont sauvegardées sous forme d'association *clef/valeur*. Seuls les types primitifs suivants sont possibles : *boolean*, $float$, *int*, *long* ou encore *String*.

Les méthodes suivantes sont à utiliser sur un objet $SharedPreferences$:

```
public abstract Boolean getBoolean(String key, boolean defaultValue)
public abstract Float getFloat(String key, float defaultValue)
public abstract Int getInt(String key, int defaultValue)
public abstract Long getLong(String key, long defaultValue)
public abstract String getString(String key, String defaultValue)
```

28. https ://developer.android.com/reference/kotlin/android/content/SharedPreferences

Le deuxième paramètre de ces fonctions correspond à la valeur par défaut qui sera renvoyée si la clef n'existe pas dans le fichier de préférences.

L'écriture des données est réalisée à l'aide d'un objet de type $SharedPreferences.Editor$ qui peut être récupéré par la méthode $edit()$ de l'objet $SharedPreferences$.

Les méthodes suivantes permettent d'écrire les données associées à leur clef :

```
public abstract SharedPreferences setBoolean(String key, boolean value)
public abstract SharedPreferences setFloat(String key, float value)
public abstract SharedPreferences setInt(String key, int value)
public abstract SharedPreferences setLong(String key, long value)
public abstract SharedPreferences setString(String key, String value)
```

L'écriture des données ne sera effective qu'après l'appel à la méthode $commit()$ de $SharedPreferences$. Si cette méthode n'est pas appelée, les données se seront pas sauvegardées.

Voici un exemple de code de lecture et d'écriture de données :

```
SharedPreferences mesPreferences = super.getPreferences(Context.MODE_PRIVATE);
SharedPreferences.Editor editeur = mesPreferences.edit();
// écriture de l'association nom/valeur
editeur.putString("nom","valeur");
editeur.commit();
// lecture de la valeur de la clef "nom":
String nom = mesPreferences.getString("nom","");
```

Table des matières

1 Présentation d'Android **2**
1.1 Versions d'Android . . . 2
1.2 Architecture d'une application Android . . . 3
1.3 Environnement de développement . . . 4

2 Une première application **4**
2.1 création d'un nouveau projet . . . 4
2.2 Présentation de l'environnement . . . 7
2.3 Compilation et exécution . . . 9
2.4 Outils de débogage . . . 10
2.4.1 Messages *Toast* . . . 10
2.4.2 Le mécanisme *Logcat* . . . 11
2.5 Outils d'analyse de performance . . . 12

3 Structure d'un projet avec *Android Studio* **12**
3.1 Les fichiers *manifeste* . . . 13
3.2 Le fichier *APK* . . . 15
3.3 *Gradle* : le gestionnaire de compilation . . . 16
3.3.1 *settings.gradle* (niveau projet) . . . 16
3.3.2 *local.properties* . . . 16
3.3.3 *build.gradle* . . . 17
3.3.4 *build.gradle* (niveau application) . . . 17
3.4 Spécification des ressources . . . 18

4 Composants logiciels **19**
4.1 Les activités . . . 19
4.1.1 Déclaration d'une activité . . . 19
4.1.2 Le cycle de vie d'une activité . . . 19
4.1.3 Sauvegarde et restauration des données . . . 21
4.1.4 Pile d'activités . . . 21
4.2 Les intentions . . . 22
4.2.1 Création d'une intention . . . 22
4.2.2 Intention explicite . . . 23
4.2.3 Intention implicite . . . 24
4.2.4 Filtres d'intention . . . 24
4.3 Les éléments de l'interface utilisateur . . . 25
4.3.1 La mise en page : vue, *layout* et *widget* . . . 25
4.3.2 Déclaration en mode déclaratif . . . 26
4.3.3 Déclaration en mode programmatique . . . 28
4.4 Les fragments . . . 29
4.4.1 Cycle de vie . . . 29
4.4.2 Création des fragments . . . 30
4.4.3 Le gestionnaire de fragment . . . 32
4.4.4 La pile de fragments . . . 33
4.4.5 La mise en page dynamique . . . 34
4.4.6 La bibliothèque *android.support.v4* . . . 34
4.5 Les listes . . . 35
4.5.1 Le composant *ListView* . . . 35
4.5.2 Implémentation avec les classes standard . . . 35

4.5.3 Implémentation spécifique 36
4.5.4 Gestion des listes 37
4.5.5 Listes déroulantes 39
4.6 Menus et barres d'actions 40
4.6.1 Déclarations des menus 40
4.6.2 Les barres d'actions 41
4.6.3 Les menus d'option 42
4.6.4 Les menus *pop-up* 44

5 Persistance des données **45**
5.1 Fichiers de préférences 45