

Développement Android

Jean-Francois Lalande - November 2017 - Version 2.6

Le but de ce cours est de découvrir la programmation sous Android, sa plate-forme de développement et les spécificités du développement embarqué sur *smartphone*.



INSA | INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
CENTRE VAL DE LOIRE

1 Plan du module

Plan du module

1 Plan du module	2
1.1 Objectifs et ressources	3
2 Introduction aux concepts d'Android	5
2.1 Introduction	5
2.2 Android	6
2.3 Les ressources	9
2.4 Les activités	11
3 Interfaces graphiques	13
3.1 Vues et gabarits	14
3.2 Inclusions de gabarits	17
3.3 Positionnement avancé	18
3.4 Les listes	19
3.5 Les Fragments	22
3.6 Action bar	28
3.7 Animations et helpers	30
4 Les Intents	34
4.1 Principe des Intents	34
4.2 Intents pour une nouvelle activité	34
4.3 Ajouter des informations	36
4.4 Types d'Intent: actions	36
4.5 Broadcaster des informations	37
4.6 Recevoir et filtrer les Intents	37
5 Persistance des données	41
5.1 Différentes persistances	41
5.2 Préférences partagées	41
5.3 Les fichiers	44
5.4 BDD SQLite	44
5.5 XML	45
6 Programmation concurrente	47
6.1 Composants d'une application	47
6.2 Processus	48
6.3 Threads	48
6.4 Services	49
6.5 Tâches concurrentes	51
6.6 Bilan: processus et threads	54
6.7 Coopération service/activité	55
6.8 Etude de cas	57
7 Connectivité	60

7.1	Téléphonie	60
7.2	Réseau	61
7.3	Bluetooth	62
7.4	Localisation	63
7.5	Capteurs	65
7.6	Caméra	67
7.7	Secure element et NFC	67
8	Développement client serveur	69
8.1	Architectures	69
8.2	Applications Natives	70
8.3	Applications Hybrides	70
8.4	Architectures REST	71
9	Android Wear	74
9.1	Philosophie	74
9.2	UI Design	75
9.3	Faire communiquer Handheld et Wearable	81
9.4	Watchfaces	84
10	Google Cloud Messaging	86
10.1	GCM: Google Cloud Messaging	86
10.2	Etape 1: Developer Console	86
10.3	Etape 2: créer un serveur de démo	88
10.4	Etape 3: développer l'application	88
11	Divers	94
11.1	Librairies natives: JNI	94
12	Annexes: outils	97
12.1	Outils à télécharger	97
12.2	L'émulateur Android	97
12.3	ADB: Android Debug Bridge	98
12.4	Simuler des sensors	99
12.5	HierarchyViewer	100

1.1 Objectifs et ressources

Les grandes notions abordées dans ce cours sont:

- Bâtir l'interface d'une application
- Naviguer et faire communiquer des applications
- Manipuler des données (préférences, fichiers, ...)
- Services, threads et programmation concurrente
- Les capteurs, le réseau
- La sécurité dans Android
- Android Wear

Les ressources de ce cours sont disponibles en ligne à l'adresse:



<http://www.univ-orleans.fr/lifo/Members/Jean-Francois.Lalande/teaching.html>.

On y trouve deux versions du même contenu et des codes sources:

- [Slides du cours](#)
- [Support de cours](#)
- [Codes source sur GitHub](#)

2 Introduction aux concepts d'Android

2.1 Introduction	5
Historique des versions	5
Dernières versions	6
2.2 Android	6
L'Operating System	6
Dalvik et ART	7
Le plugin de développement d'Eclipse: ADT	7
L'environnement Android Studio	7
L'architecture d'un projet Android Studio	8
Attention aux vues dans Android Studio	8
Les éléments d'une application	8
Le Manifest de l'application	9
2.3 Les ressources	9
Les chaines	10
Internationalisation	10
Autres valeurs simples	10
Autres ressources	11
2.4 Les activités	11
Cycle de vie d'une activité	12
Sauvegarde des interfaces d'activité	12
Démonstration	12

2.1 Introduction

Il est important de prendre la mesure des choses. A l'heure actuelle (November 2017):

- juillet 2011: **550 000** activations **par jour**
- décembre 2011: **700 000** activations **par jour**
- sept. 2012: **1.3 millions** d'activations **par jour** ([Wikipedia](#))
- avril 2013: **1.5 millions** d'activations **par jour** ([Wikipedia](#))

Il y aurait donc un parc de **400 millions** d'appareils Android.

Vous pouvez visionner de la propagande [ici](#) et [là](#).

Historique des versions

Le nombre de *release* est impressionnant ([Version](#)):

Nom	Version	Date
Android	1.0	09/2008
Petit Four	1.1	02/2009
Cupcake	1.5	04/2009
Donut	1.6	09/2009
Gingerbread	2.3	12/2010
Honeycomb	3.0	02/2011

Ice Cream Sandwich	4.0.1	10/2011
Jelly Bean	4.1	07/2012
KitKat	4.4	10/2013
Lollipop	5.0	10/2014
Marshmallow	6.0	05/2015
Nougat	7.0	09/2016
Oreo	8.0	08/2017

Dernières versions



CC BY - Google Inc.



CC BY - Google Inc.

2.2 Android

L'écosystème d'Android s'appuie sur deux piliers:

- le langage Java
- le SDK qui permet d'avoir un environnement de développement facilitant la tâche du développeur

Le kit de développement donne accès à des exemples, de la documentation mais surtout à l'API de programmation du système et à un émulateur pour tester ses applications.

Stratégiquement, Google utilise la licence Apache pour Android ce qui permet la redistribution du code sous forme libre ou non et d'en faire un usage commercial.

Le SDK était:

- **anciennement** manipulé par un plugin d'Eclipse (obsolète)
- **maintenant** intégré à Android Studio (IntelliJ)

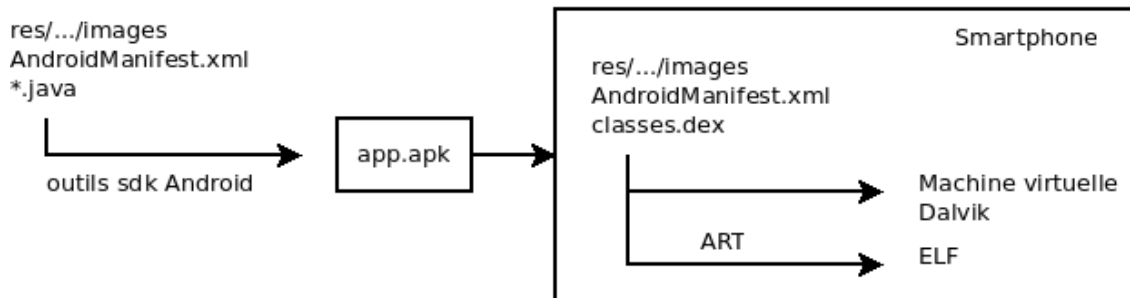
L'Operating System

Android est en fait un système de la famille des Linux, pour une fois sans les outils GNU. L'OS s'appuie sur:



- un noyau Linux (et ses drivers)
- un couche d'abstraction pour l'accès aux capteurs (HAL)
- une machine virtuelle: *Dalvik Virtual Machine* (avant Lollipop)
- un compilateur de bytecode vers le natif *Android Runtime* (pour Lollipop)
- des applications (navigateur, gestion des contacts, application de téléphonie...)
- des bibliothèques (SSL, SQLite, OpenGL ES, etc...)
- des API d'accès aux services Google

Anatomie d'un déploiement:



Dalvik et ART

[Dalvik] est le nom de la machine virtuelle open-source utilisée sur les systèmes Android. Cette machine virtuelle exécute des fichiers .dex, plus ramassés que les .class classiques. Ce format évite par exemple la duplication des **String** constantes. La machine virtuelle utilise elle-même moins d'espace mémoire et l'adressage des constantes se fait par un pointeur de 32 bits.

[Dalvik] n'est pas compatible avec une JVM du type Java SE ou même Java ME. La librairie d'accès est donc redéfinie entièrement par Google.

A partir de Lollipop, Android dispose d'**ART** qui compile l'application au moment du déploiement (Ahead-of-time compilation).

Le plugin de développement d'Eclipse: ADT

[Obsolète] Un projet basé sur le plugin ADT **était** décomposé de la manière suivante:

- **src/**: les sources Java du projet
- **libs/**: bibliothèques tierces
- **res/**:
 - **res/drawable**: ressources images
 - **res/layout**: description des IHMs en XML
 - **res/values**: chaînes de caractères et dimensions
- **gen/**: les ressources auto générées par ADT
- **assets/**: ressources brutes (*raw bytes*)
- **bin/**:
 - **bin/classes**: les classes compilées en .class
 - **bin/classes.dex**: exécutable pour la JVM Dalvik
 - **bin/myapp.zip**: les ressources de l'application
 - **bin/myapp.apk**: application empaquetée avec ses ressource et prête pour le déploiement

L'environnement Android Studio

Depuis mi-2015, il faut utiliser Android Studio (IDE IntelliJ ayant subi l'intégration de fonctionnalités de développement Android).

Avantages:



- meilleur intégration du SDK dans Android Studio
- puissance de l'IDE IntelliJ

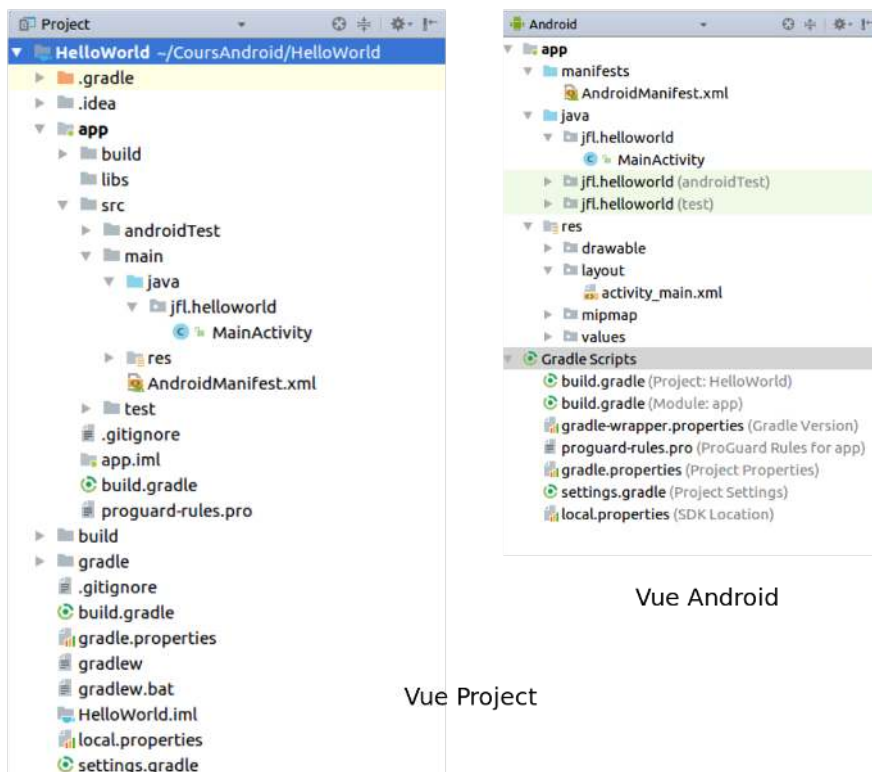
Désavantages:

- lourdeur de l'IDE IntelliJ
- moins d'outils standalone (gestion des émulateurs, du SDK)
- obligation de s'habituer à un autre IDE
- nouvelle architecture des répertoires
- meilleur gestion des dépendances avec gradle

L'architecture d'un projet Android Studio

- **app**: le code de votre application
 - **build**: le code compilé
 - **lib**: les bibliothèques natives
 - **src**: les sources de votre applications
 - **main/java**: vos classes
 - **main/res**: vos ressources (XML, images, ...)
 - **test**: les tests unitaires
- Des fichiers de configuration:
 - **build.gradle** (2 instances): règles de dépendance et de compilation
 - **settings.gradle**: liste de toutes les applications à compiler (si plusieurs)

Attention aux vues dans Android Studio



Les éléments d'une application



Une application Android peut être composée des éléments suivants:

- des activités (**android.app.Activity**): il s'agit d'une partie de l'application présentant une vue à l'utilisateur
- des services (**android.app.Service**): il s'agit d'une activité tâche de fond sans vue associée
- des fournisseurs de contenus (**android.content.ContentProvider**): permet le partage d'informations au sein ou entre applications
- des widgets (**android.appwidget.**): une vue accrochée au Bureau d'Android
- des *Intents* (**android.content.Intent**): permet d'envoyer un message pour un composant externe sans le nommer explicitement
- des récepteurs d'*Intents* (**android.content.BroadcastReceiver**): permet de déclarer être capable de répondre à des *Intents*
- des notifications (**android.app.Notifications**): permet de notifier l'utilisateur de la survenue d'événements

Le Manifest de l'application

Le fichier **AndroidManifest.xml** déclare l'ensemble des éléments de l'application.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="andro.jf"
    android:versionCode="1"
    android:versionName="1.0">
    <application android:icon="@drawable/icon"
        android:label="@string/app_name">

        <activity android:name=".Main"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <service>...</service>
        <receiver>...</receiver>
        <provider>...</provider>

    </application>
</manifest>
```

2.3 Les ressources

Les ressources de l'application sont utilisées dans le code au travers de la classe statique **R**. ADT re-génère automatiquement la classe statique **R** à chaque changement dans le projet. Toutes les ressources sont accessibles au travers de **R**, dès qu'elles sont déclarées dans le fichier XML ou que le fichier associé est déposé dans le répertoire adéquat. Les ressources sont utilisées de la manière suivante:

```
android.R.type_ressource.nom_ressource
```

qui est de type int. Il s'agit en fait de l'identifiant de la ressource. On peut alors utiliser cet identifiant ou récupérer l'instance de la ressource en utilisant la classe **Resources**:

```
Resources res = getResources();
String hw = res.getString(R.string.hello);
XXX o = res.getXXX(id);
```

Une méthode spécifique pour les objets graphiques permet de les récupérer à partir de leur id, ce qui permet d'agir sur ces instances même si elles ont été créées via leur définition XML:

```
TextView texte = (TextView) findViewById(R.id.le_texte);
texte.setText("Here we go !");
```

Les chaînes

Les chaînes constantes de l'application sont situées dans **res/values/strings.xml**. L'externalisation des chaînes permettra de réaliser l'internationalisation de l'application. Voici un exemple:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="hello">Hello Hello JFL !</string>
  <string name="app_name">AndroJF</string>
</resources>
```

La récupération de la chaîne se fait via le code:

```
Resources res = getResources();
String hw = res.getString(R.string.hello);
```



Internationalisation

Le système de ressources permet de gérer très facilement l'internationalisation d'une application. Il suffit de créer des répertoires **values-XX** où **XX** est le code de la langue que l'on souhaite implanter. On place alors dans ce sous répertoire le fichier xml **strings.xml** contenant les chaînes traduites associées aux mêmes clefs que dans **values/strings.xml**. On obtient par exemple pour les langues es et fr l'arborescence:

```
MyProject/
  res/
    values/
      strings.xml
    values-es/
      strings.xml
    values-fr/
      strings.xml
```

Android chargera le fichier de ressources approprié en fonction de la langue du système.

Autres valeurs simples

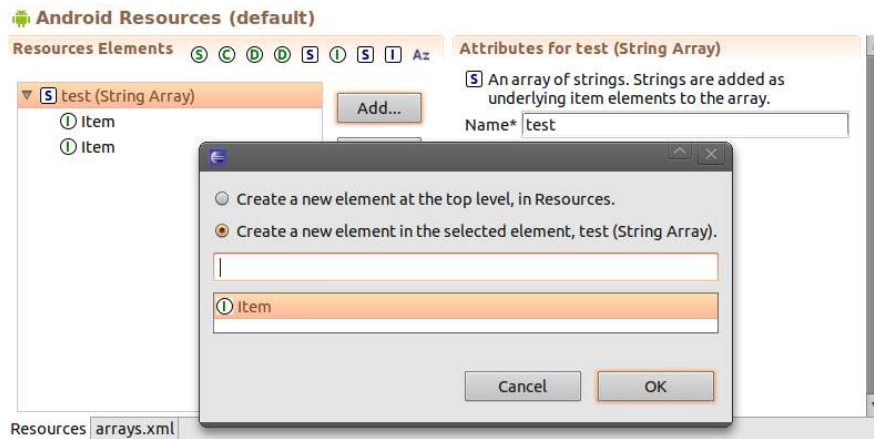
Plusieurs fichiers xml peuvent être placés dans **res/values**. Cela permet de définir des chaînes, des couleurs, des tableaux. L'assistant de création permet de créer de nouveaux fichiers de ressources contenant des valeurs simples, comme par exemple un tableau de chaînes:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string-array name="test">
    <item>itl</item>
```

```

<item>it2</item>
</string-array>
</resources>

```



Autres ressources

D'autres ressources sont spécifiées dans **res**:

- les menus (**R.menu**)
- les images (**R.drawable**)
- des couleurs (**R.color**):

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <resources>
3      <color name="colorPrimary">#3F51B5</color>
4      <color name="colorPrimaryDark">#303F9F</color>
5      <color name="colorAccent">#FF4081</color>
6  </resources>
7

```

2.4 Les activités

Une application Android étant hébergée sur un système embarqué, le cycle de vie d'une application ressemble à celle d'une application Java ME. L'activité peut passer des états:

- démarrage -> actif: détient le focus et est démarré
- actif -> suspendue: ne détient plus le focus
- suspendue -> actif:
- suspendue -> détruit:

Le nombre de méthodes à surcharger et même plus important que ces états:

```

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

```

```
protected void onStart() {
    super.onStart(); }
protected void onStop() {
    super.onStop(); }
```

Cycle de vie d'une activité

onCreate() / **onDestroy()**: permet de gérer les opérations à faire avant l'affichage de l'activité, et lorsqu'on détruit complètement l'activité de la mémoire. On met en général peu de code dans **onCreate()** afin d'afficher l'activité le plus rapidement possible.

onStart() / **onStop()**: ces méthodes sont appelées quand l'activité devient visible/invisible pour l'utilisateur.

onPause() / **onResume()**: une activité peut rester visible mais être mise en pause par le fait qu'une autre activité est en train de démarrer, par exemple B. **onPause()** ne doit pas être trop long, car B ne sera pas créé tant que **onPause()** n'a pas fini son exécution.

onRestart(): cette méthode supplémentaire est appelée quand on relance une activité qui est passée par **onStop()**. Puis **onStart()** est aussi appelée. Cela permet de différencier le premier lancement d'un relancement.

Le cycle de vie des applications est très bien décrit sur la page qui concerne les [Activity](#).

Sauvegarde des interfaces d'activité

L'objet **Bundle** passé en paramètre de la méthode **onCreate** permet de restaurer les valeurs des interfaces d'une activité qui a été déchargée de la mémoire. En effet, lorsque l'on appuie par exemple sur la touche *Home*, en revenant sur le bureau, Android peut-être amené à décharger les éléments graphiques de la mémoire pour gagner des ressources. Si l'on rebascule sur l'application (appui long sur *Home*), l'application peut avoir perdu les valeurs saisies dans les zones de texte.

Pour forcer Android à décharger les valeurs, il est possible d'aller dans "Development tools > Development Settings" et de cocher "Immediately destroy activities".

Si une zone de texte n'a pas d'identifiant, Android ne pourra pas la sauver et elle ne pourra pas être restaurée à partir de l'objet **Bundle**.

Si l'application est complètement détruite (tuée), rien n'est restauré.

Le code suivant permet de visualiser le déclenchement des sauvegardes:

```
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    Toast.makeText(this, "Sauvegarde !", Toast.LENGTH_LONG).show();
}
```

Démonstration

[Video](#)

3 Interfaces graphiques

3.1 Vues et gabarits	14
Attributs des gabarits	14
Exemples de ViewGroup	14
L'interface comme ressource	15
Les labels de texte	15
Les zones de texte	15
Les images	16
Les boutons	16
Interface résultat	17
Démonstration	17
3.2 Inclusions de gabarits	17
Merge de gabarit	18
3.3 Positionnement avancé	18
Preview du positionnement	18
3.4 Les listes	19
Démonstration	19
Liste de layouts plus complexes	19
Interface résultat	20
Liste avec plusieurs données	20
UserAdapter héritant de ArrayAdapter	21
RecyclerView	21
L'Adapter	22
3.5 Les Fragments	22
Fragments dynamiques	23
Fragment Manager	24
Gerer la diversité des appareils	24
Exemple: Master Detail/Flow	25
Cycle de vie d'un fragment	25
Callback d'un fragment vers l'activité	26
Démonstration	27
4 fragments spéciaux	27
DialogFragment	28
Les retain Fragments	28
3.6 Action bar	28
Gérer les actions sur l'action bar	29
Menu déroulant vs actions avec icône	29
3.7 Animations et helpers	30
L'animation des gabarits	30
Les onglets	30
Démonstration	31

Défilement d'écrans	32
Démonstration	33

3.1 Vues et gabarits

Les éléments graphiques héritent de la classe **View**. On peut regrouper des éléments graphiques dans une **ViewGroup**. Des **ViewGroup** particuliers sont prédéfinis: ce sont des gabarits (*layout*) qui proposent une prédispositions des objets graphiques:

- **LinearLayout**: dispose les éléments de gauche à droite ou du haut vers le bas
- **RelativeLayout**: les éléments enfants sont placés les uns par rapport aux autres
- **TableLayout**: disposition matricielle
- **FrameLayout**: disposition en haut à gauche en empilant les éléments
- **GridLayout**: disposition matricielle avec N colonnes et un nombre infini de lignes

Les déclarations se font principalement en XML, ce qui évite de passer par les instanciations Java.

Attributs des gabarits

Les attributs des gabarits permettent de spécifier des attributs supplémentaires. Les plus importants sont:

- **android:layout_width** et **android:layout_height**:
 - **"match_parent"**: l'élément remplit tout l'élément parent
 - **"wrap_content"**: prend la place minimum nécessaire à l'affichage
 - **"fill_parent"**: comme **match_parent** (deprecated, API<8)
- **android:orientation**: définit l'orientation d'empilement
- **android:gravity**: définit l'alignement des éléments

Voici un exemple de **LinearLayout**:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center"
    android:id="@+id/accueilid"
    >
</LinearLayout>
```

Exemples de ViewGroup

Un **ViewGroup** contient des éléments graphiques. Pour construire un gabarit linéaire on fera:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center"
    android:id="@+id/accueilid"
    >
    <TextView android:id="@+id/le_texte" android:text="@string/hello" />
    <TextView android:id="@+id/le_texte2" android:text="@string/hello2" />
</LinearLayout>
```

Alors que pour empiler une image et un texte:

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
```

```

android:id="@+id/accueilid"
>
  <ImageView
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:src="@drawable/mon_image" />

  <TextView android:id="@+id/le_texte" android:text="@string/hello" />
</FrameLayout

```

L'interface comme ressource

Une interface graphique définie en XML sera aussi générée comme une ressource dans la classe statique **R**. Le nom du fichier xml, par exemple *accueil.xml* permet de retrouver le layout dans le code java au travers de **R.layout.accueil**.

Ainsi, pour associer la première vue graphique à l'activité principale de l'application, il faut faire:

```

public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.accueil);
}

```

Le layout reste modifiable au travers du code, comme tous les autres objets graphiques. Pour cela, il est important de spécifier un id dans la définition XML du gabarit (**android:id="@+id/accueilid"**). Le "+" signifie que cet id est nouveau et doit être généré dans la classe **R**. Un id sans "+" signifie que l'on fait référence à un objet déjà existant.

En ayant généré un *id*, on peut accéder à cet élément et agir dessus au travers du code Java:

```

LinearLayout l = (LinearLayout) findViewById(R.id.accueilid);
l.setBackgroundColor(Color.BLACK);

```

Les labels de texte

En XML:

```

<TextView
  android:id="@+id/le_texte"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="@string/hello"
  android:layout_gravity="center"
/>

```

Par la programmation:

```

public class Activity2 extends Activity {
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        LinearLayout gabarit = new LinearLayout(this);
        gabarit.setGravity(Gravity.CENTER); // centrer les éléments graphiques
        gabarit.setOrientation(LinearLayout.VERTICAL); // empiler vers le bas !

        TextView texte = new TextView(this);
        texte.setText("Programming creation of interface !");
        gabarit.addView(texte);
        setContentView(gabarit);
    }
}

```

Les zones de texte

En XML:



```
<EditText android:text=""
    android:id="@+id/EditText01"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
</EditText>
```

Par la programmation:

```
EditText edit = new EditText(this);
edit.setText("Edit me");
gabarit.addView(edit);
```

Interception d'événements:

```
edit.addTextChangedListener(new TextWatcher() {
    @Override
    public void onTextChanged(CharSequence s, int start,
        int before, int count) {
        // do something here
    }
});
```

Les images

En XML:

```
<ImageView
    android:id="@+id/logoEnsi"
    android:src="@drawable/ensi"
    android:layout_width="100px"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
></ImageView>
```

Par la programmation:

```
ImageView image = new ImageView(this);
image.setImageResource(R.drawable.ensi);
gabarit.addView(image);
```

Les boutons

En XML:

```
<Button android:text="Go !"
    android:id="@+id/Button01"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content">
</Button>
```

La gestion des événements de *click* se font par l'intermédiaire d'un listener:

```
Button b = (Button) findViewById(R.id.Button01);
b.setOnClickListener(new OnClickListener() {

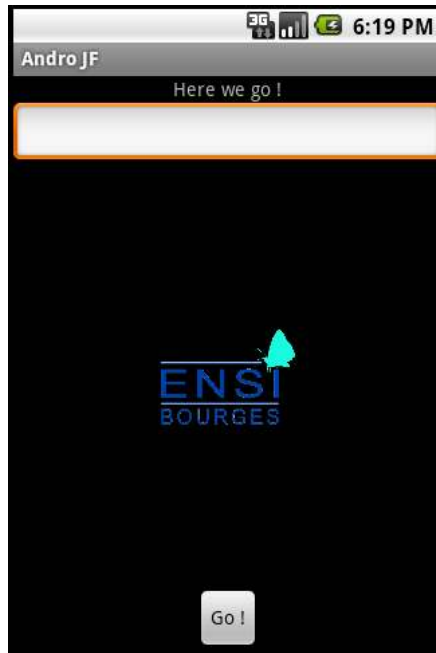
    @Override
    public void onClick(View v) {
        Toast.makeText(v.getContext(), "Stop !", Toast.LENGTH_LONG).show();
    }
});
```



```
});
}
```

Interface résultat

Ce *screenshot* montre une interface contenant des **TextView**, **EditText**, **ImageView**, et un bouton (cf. [Code-IHM](#)).



Démonstration

[Video](#)

3.2 Inclusions de gabarits

Les interfaces peuvent aussi inclure d'autres interfaces, permettant de factoriser des morceaux d'interface. On utilise dans ce cas le mot clef **include**:

```
<?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"
    >
    <include android:id="@+id/include01"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        layout="@layout/acceuil"
    >>/include>
</LinearLayout>
```

Si le gabarit correspondant à `accueil.xml` contient lui aussi un **LinearLayout**, on risque d'avoir deux imbrications de **LinearLayout** inutiles car redondant:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android">
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android">
    <TextView ... />
    <TextView ... />
</LinearLayout>
```

```

</LinearLayout>
</LinearLayout>

```

Merge de gabarit

Le problème précédent est dû au fait qu'un layout doit contenir un unique *root element*, du type **View** ou **ViewGroup**. On peut pas mettre une série de **TextView** sans *root element*. Pour résoudre ce problème, on peut utiliser le tag ****merge****. Si l'on réécrit le layout accueil.xml comme cela:

```

<merge xmlns:android="http://schemas.android.com/apk/res/android">
  <TextView ... />
  <TextView ... />
</merge>

```

L'inclusion de celui-ci dans un layout linéaire transformera:

```

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android">
  <include android:id="@+id/include01" layout="@layout/accueil"></include>
</LinearLayout>

```

en:

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android">
  <TextView ... />
  <TextView ... />
</LinearLayout>

```

3.3 Positionnement avancé

Pour obtenir une interface agréable, il est souvent nécessaire de réaliser correctement le positionnement des éléments graphiques. La difficulté est d'arriver à programmer un placement qui n'est pas dépendant de l'orientation ou de la taille de l'écran.

Dans [VL], on trouve une explication pour réaliser un placement simple: un texte à gauche et une image à droite de l'écran, alignée avec le texte. Cela peut être particulièrement utile si par exemple on réalise une liste d'item qui contient à chaque fois un texte et une icône.

Le principe réside dans l'imbrication de **LinearLayout**:

- Un premier *layout* contiendra l'ensemble des éléments. Son orientation doit être *horizontal* et sa gravité (*gravity*) *center*. On inclut ensuite le texte.
- Puis, l'image étant censée être à droite, il faut créer un **LinearLayout** consécutif au texte et préciser que la gravité (*gravity*) est *right*. Pour aligner les éléments, il faut préciser que la gravité du layout (*layout_gravity*) est *center*.

Preview du positionnement

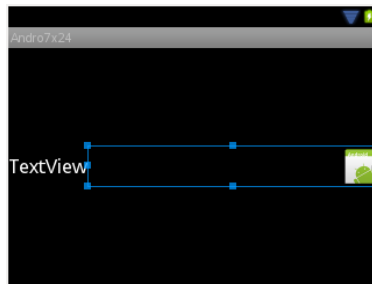
Le *layout* décrit ci-avant ressemble à:

```

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_height="match_parent" android:orientation="horizontal"
  android:layout_width="match_parent"
  android:gravity="center">
  <TextView ...></TextView>
  <LinearLayout android:layout_height="wrap_content"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:gravity="right"
    android:layout_gravity="center">
    <Image .../>
  </LinearLayout></LinearLayout>

```

Ce qui produit dans l'interface de preview, en orientation portrait:



3.4 Les listes

Au sein d'un gabarit, on peut implanter une liste que l'on pourra dérouler si le nombre d'éléments est important. Si l'on souhaite faire une liste plein écran, il suffit juste de poser un layout linéaire et d'y implanter une **ListView**. Le XML du gabarit est donc:

```
<LinearLayout ...>
<ListView android:id="@+id/listView1" ...>
</ListView></LinearLayout>
```

Etant donné qu'une liste peut contenir des éléments graphiques divers et variés, les éléments de la liste doivent être insérés dans un **ListAdapter** et il faut aussi définir le gabarit qui sera utilisé pour afficher chaque élément du **ListAdapter**. Prenons un exemple simple: une liste de chaîne de caractères. Dans ce cas, on crée un nouveau gabarit **montexte** et on ajoute dynamiquement un **ArrayAdapter** à la liste **listView1**. Le gabarit suivant doit être placé dans *montexte.xml*:

```
<TextView ...> </TextView>
```

Le code de l'application qui crée la liste peut être:

```
ListView list = (ListView) findViewById(R.id.listView1);
ArrayAdapter<String> tableau = new ArrayAdapter<String>(list.getContext(),
                                                    R.layout.montexte);

for (int i=0; i<40; i++) {
    tableau.add("coucou " + i);
}
list.setAdapter(tableau);
```

Démonstration

[Video](#)

(cf [Code-List](#))

Liste de layouts plus complexes

Lorsque les listes contiennent un layout plus complexe qu'un texte, il faut utiliser un autre constructeur de **ArrayAdapter** (ci-dessous) où **resource** est l'id du layout à appliquer à chaque ligne et **textViewResourceId** est l'id de la zone de texte inclu dans ce layout complexe. A chaque entrée de la liste, la vue générée utilisera le layout complexe et la zone de texte contiendra la *string* passée en argument à la méthode **add**.

```
ArrayAdapter (Context context, int resource, int textViewResourceId)
```

Le code de l'exemple précédent doit être adapté comme ceci:

```
ListView list = (ListView) findViewById(R.id.maliste);
ArrayAdapter<String> tableau = new ArrayAdapter<String>(
    list.getContext(), R.layout.ligne, R.id.monTexte);
for (int i=0; i<40; i++) {
```

```

    tableau.add("coucou " + i); }
list.setAdapter(tableau);

```

Avec le layout de liste suivant (ligne.xml):

```

<LinearLayout ...>
    <TextView ... android:id="@+id/monTexte"/>
    <LinearLayout> <ImageView /> </LinearLayout>
</LinearLayout>

```

Une autre solution consiste à hériter de la classe **BaseAdapter**, ce qui oblige à coder la méthode **getView()** mais évite la complexité du constructeur de **ArrayAdapter**.

Interface résultat

(cf [Code-ListesComplexes](#))



Liste avec plusieurs données

Lorsque chaque item de la liste contient plusieurs données dynamiques, il faut recoder la classe **ArrayAdapter**, comme expliqué dans [LVAA](#). Si l'on suppose par exemple que la donnée est:

```

public class User {
    public String name;
    public String hometown; }

```

Et que le layout **item_user.xml** d'un item est:

```

<LinearLayout ...>
    <TextView android:id="@+id/tvName" />
    <TextView android:id="@+id/tvHome" /> </>

```

On doit alors écrire une méthode **getView** d'une classe **UsersAdapter** héritant de **ArrayAdapter** qui va renvoyer l'élément graphique correspondant à une item (donné ci-après). Pour utiliser cette **ArrayAdapter** on fait comme précédemment:

```
// Construct the data source
ArrayList<User> arrayOfUsers = new ArrayList<User>();
// Create the adapter to convert the array to views
UsersAdapter adapter = new UsersAdapter(this, arrayOfUsers);
ListView listView = (ListView) findViewById(R.id.lvItems);
listView.setAdapter(adapter)
adapater.add(new User(...));
```

UserAdapter héritant de ArrayAdapter

```
public class UsersAdapter extends ArrayAdapter<User> {
    public UsersAdapter(Context context, ArrayList<User> users) {
        super(context, 0, users); }
    int getCount()
    { // TODO !!! IMPORTANT !!! REQUIRED !!! DO NOT FORGET !!! }
    public View getView(int position, View convertView, ViewGroup parent) {
        // Get the data item for this position
        User user = getItem(position);
        // Check if an existing view is being reused, otherwise inflate the view
        if (convertView == null) {
            convertView = LayoutInflater.from(getContext())
                .inflate(R.layout.item_user, parent, false);
        }
        // Lookup view for data population
        TextView tvName = (TextView) convertView.findViewById(R.id.tvName);
        TextView tvHome = (TextView) convertView.findViewById(R.id.tvHome);
        // Populate the data into the template view using the data object
        tvName.setText(user.name);
        tvHome.setText(user.hometown);
        // Return the completed view to render on screen
        return convertView;
    }
}
```

On notera l'appel à `LayoutInflater.from(getContext()).inflate` qui permet de gonfler l'élément graphique pour un item de liste à partir du XML décrivant le *layout* d'un item. Sur cet élément gonflé, on peut chercher des éléments graphiques à l'aide de `findViewById`. Enfin, ne pas oublier `getCount()` qui sera appelé pour savoir combien d'éléments sont à afficher et donc déclenchera les appels à `getView`.

RecyclerView

Dans les dernières version d'Android, un nouveau mécanisme permet de gérer des listes volumineuses pour lesquelles on recycle les éléments graphiques dynamiquement lorsque la liste est parcourue. Il s'agit de la classe `RecyclerView`.

Comme pour une `ListView` il faut un *Adapter*. Il faut en plus un *layout manager* qui va gérer le recyclage des éléments graphiques. Après avoir déclaré un `RecyclerView` dans votre gabarit:

```
<android.support.v7.widget.RecyclerView
    android:id="@+id/rv"
    android:layout_below="@+id/button"
    android:layout_height="match_parent"
    android:layout_width="match_parent">
</android.support.v7.widget.RecyclerView>
```

Il faut accrocher l'*Adapter* et le *LayoutManager* à ce composant graphique:

```
RecyclerView rv = (RecyclerView) findViewById(R.id.rv);
rv.setLayoutManager(new LinearLayoutManager(this, LinearLayoutManager.VERTICAL, false));
rv.setAdapter(new RVAdapter());
```

Chaque item aura pour *layout* (id: `R.id.itemlayout`):

```
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content" android:layout_height="wrap_content"
    android:id="@+id/itemid" />
```

L'Adapter

Le code de l'**Adapter** est plus complexe: il s'agit d'une classe utilisant un générique qui doit créer un objet encapsulant l'objet graphique à afficher pour chaque item (dans l'exemple **StringHolder** héritant de **RecyclerView.ViewHolder**). L'*Adapter* a deux buts: créer les objets graphiques ou changer le contenu d'un objet graphique (car la liste est déroulée).

```
public class RVAdapter extends RecyclerView.Adapter<StringHolder> {
    Vector<String> elements;
    public RVAdapter() {
        elements = new Vector<String>();
        for (int i=0; i<490; i++)
            elements.add("" + i);
    }

    public StringHolder onCreateViewHolder(ViewGroup parent, int viewType) {
        LayoutInflater lf = LayoutInflater.from(parent.getContext());
        TextView v = (TextView)lf.inflate(R.layout.itemlayout, parent, false);
        StringHolder sh = new StringHolder(v);
        return sh;
    }

    public void onBindViewHolder(StringHolder holder, int position) {
        String aAfficher = elements.elementAt(position);
        holder.setText(aAfficher);
    }

    public int getItemCount() {
        return elements.size();
    }
}
```

onCreateViewHolder(ViewGroup parent, int viewType): doit générer un objet graphique, par exemple à partir de l'id d'un gabarit. Dans mon exemple, on génère un **TextView**, accorché au **ViewGroup** parent. Il doit aussi générer un objet **Holder** qui pourra être mis à jour plus tard pour chaque donnée. On stocke donc l'objet graphique dans ce **Holder** (cf. le constructeur).

onBindViewHolder(StringHolder holder, int position) doit mettre à jour l'objet graphique stocké dans ce **Holder** en fonction de la position de la liste.

getItemCount(): doit dire combien d'éléments sont dans la liste.

Le code du **Holder** est assez simple:

```
public class StringHolder extends RecyclerView.ViewHolder {
    private TextView v_;

    public StringHolder(TextView v) {
        super(v);
        v_ = v;
    }

    public void setText(String text) {
        v_.setText(text);
    }
}
```

3.5 Les Fragments

A partir d'Android 3.0, on dispose d'un composant graphique similaire aux **Activity**: les **Fragments**. Un fragment est une sorte de sous activité qui possède son propre cycle de vie. On peut intégrer plusieurs fragments dans une activités, et changer un fragment par un autre dynamiquement, sans changer l'activité principale. Cela rend la programmation graphique dynamique plus aisée.

Un fragment est dépendant d'une activité:

- si l'activité passe en pause, les fragments aussi
- si l'activité est détruite, les fragments aussi

Mais le but d'un fragment est d'être réutilisable dans plusieurs activités !

Les changements de fragments au sein d'une activité peuvent être enregistré dans la *back stack* (notion de transaction). Ainsi, si l'on presse le bouton retour, la transaction est annulée et on retrouve le fragment précédent.

On peut intégrer un **Fragment** dans le layout de l'activité à l'aide du tag **fragment**. Il faut placer ce fragment dans un **ViewGroup** (i.e. un **LinearLayout**, **RelativeLayout**, etc.). L'*id* permet de cibler le fragment à l'exécution, et le tag *name* permet de spécifier la classe à utiliser à l'instanciation, si on le sait à l'avance à la conception:

```
<LinearLayout ...>
  <fragment
    android:id="@+id/fragmentstatic"
    android:name="andro.jf.MonFragmentStatic" />
</LinearLayout>
```

Et la classe **MonFragmentStatique** contient le code permettant de générer la **View**:

```
public class MonFragmentStatique extends Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        View v = inflater.inflate(R.layout.fragment_main, container, false);
        return v;
    }
}
```

Fragments dynamiques

Le générateur d'exemples du SDK propose de créer un template de projet contenant un fragment dynamique, dont la classe est **PlaceholderFragment**, ajouté à la création de l'activité (cf. [Code-Fragments](#)):

```
public class MainActivity extends Activity {

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

        if (savedInstanceState == null) {
            getFragmentManager().beginTransaction()
                .add(R.id.container, new MyDynamicFragment()).commit();
        }
    }

    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        View rootView = inflater.inflate(R.layout.fragment_main_dyn, container, false);
        return rootView;
    }
}
```

Avec le gabarit de l'activité comportant un **ViewGroup** d'id container (pas besoin de tag **Fragment** !):

```
<LinearLayout android:id="@+id/container">
</LinearLayout>
```

La classe associée est toujours à implémenter, comme le cas statique:

```
/**
 * A placeholder fragment containing a simple view.
 */
```

```
public class MyDynamicFragment extends Fragment {  
  
    public MyDynamicFragment() {  
    }  
  
    @Override  
    public View onCreateView(LayoutInflater inflater,  
                             ViewGroup container,  
                             Bundle savedInstanceState) {  
        View rootView = inflater.inflate(R.layout.fragment_main_dyn, container, false);  
        return rootView;  
    }  
}
```

Fragment Manager

Le **Fragment Manager** permet de gérer les fragments d'une activité. Les principales utilisations sont:

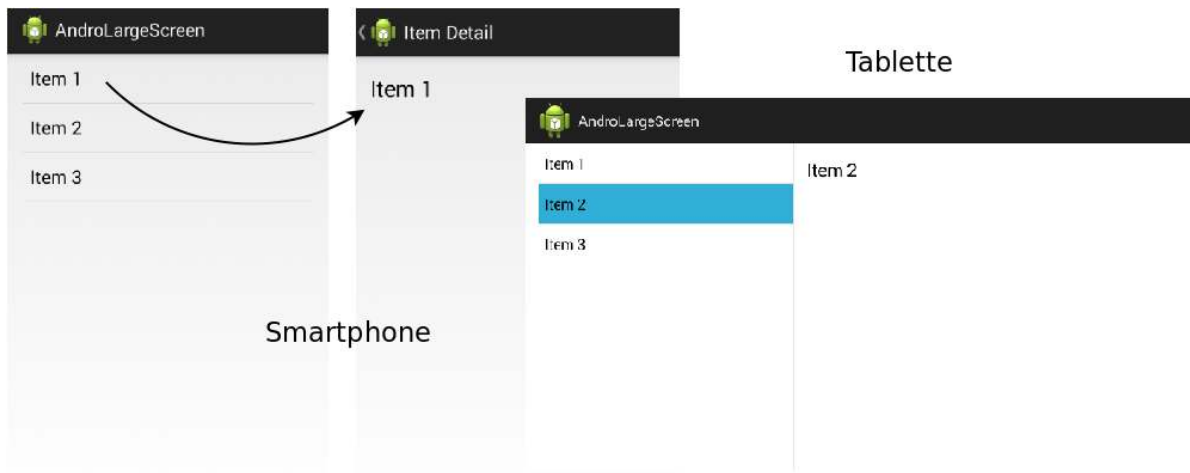
- ajouter, enlever, remplacer un fragment dans une activité
- retrouver un fragment par son *id* dans l'activité (**findFragmentById()**)
- sauvegarder un état d'un ou plusieurs fragments (**addToBackStack()**)
- restaurer un état sauvegardé précédemment d'un ou plusieurs fragments (**popBackStack()**)
- ajouter un écouteur de changement d'état (**addOnBackStackChangeListener()**)

Pour que la restauration fonctionne, en préparant une transaction, et avant d'avoir appelé *commit* il faut explicitement enregistrer l'état courant dans la pile à l'aide d'**addToBackStack()**. Dans ce cas, il est sauvegardé, et si l'utilisateur appuie sur la touche retour, il retrouvera l'interface précédente. Cela peut donner par exemple:

```
// Create new fragment and transaction  
Fragment newFragment = new ExampleFragment();  
FragmentTransaction transaction = getFragmentManager().beginTransaction();  
// Replace whatever is in the fragment_container view with this fragment,  
// and add the transaction to the back stack  
transaction.replace(R.id.fragment_container, newFragment);  
transaction.addToBackStack(null);  
// Commit the transaction  
transaction.commit();
```

Gerer la diversité des appareils

Les fragments permettent aussi de gérer la diversité des appareils qui ne disposent pas des mêmes tailles d'écran. Par exemple, on peut créer un fragment de navigation, à gauche d'une tablette, qui contrôle le contenu d'un autre fragment à droite. Il sera affiché en deux écrans successifs sur un téléphone. Cet exemple est disponible dans l'assistant de création d'applications (Master Detail/Flow).



Exemple: Master Detail/Flow

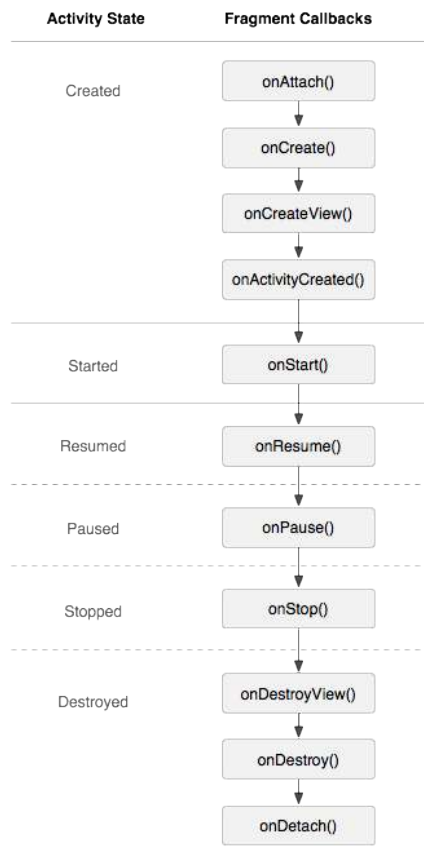
Pour contrôler le fragment de droite, on code alors deux comportements dans une *callback*, au niveau de l'activité (l'évènement remonte depuis le fragment vers l'activité). En fonction du type d'appareil, on change le fragment de droite ou on démarre une activité (qui utilise le second fragment):

```
public void onItemSelected(String id) {
    if (mTwoPane) {
        // In two-pane mode, show the detail view in this activity by
        // adding or replacing the detail fragment using a fragment transaction.
        Bundle arguments = new Bundle();
        arguments.putString(ItemDetailFragment.ARG_ITEM_ID, id);
        ItemDetailFragment fragment = new ItemDetailFragment();
        fragment.setArguments(arguments);
        getFragmentManager().beginTransaction()
            .replace(R.id.item_detail_container, fragment).commit();
    } else {
        // In single-pane mode, simply start the detail activity
        // for the selected item ID.
        Intent detailIntent = new Intent(this, ItemDetailActivity.class);
        detailIntent.putExtra(ItemDetailFragment.ARG_ITEM_ID, id);
        startActivity(detailIntent);
    }
}
```

La détection de la taille de l'écran se fait grâce à une valeur de *values-large/refs.xml* qui n'est chargée que si l'écran est large.

(cf. [Code-IHM](#))

Cycle de vie d'un fragment



Différentes *callbacks* sont appelées en fonction de l'état de l'activité principale. Un fragment est d'abord attaché à l'activité, puis créé, puis sa vue est créée. On peut même intercaler du code quand l'activité est créée.

Les différents états (Start ou Resume / Pause / Stop) correspondent aux états liés à l'activité:

- Resume/Start: le fragment est visible et l'activité s'exécute
- Pause: une autre activité est au premier plan mais l'activité qui contient le fragment est encore visible, par transparence par exemple
- Stop: le fragment n'est plus visible: il peut avoir été enlevé de l'activité par exemple, mais ajouté dans la *back stack*

Comme pour les activités, si le fragment est tué, son état peut être sauvé avec un **Bundle** en surchargeant **onSaveInstanceState()**, puis en rechargeant l'état depuis le **Bundle** dans **onCreate()**.

Représentation du cycle de vie d'un fragment, par rapport à celui d'une activité:

Callback d'un fragment vers l'activité

Pour faire communiquer fragments et activités, plusieurs possibilités sont offertes. La première est de réaliser une *callback* vers l'activité dans le fragment, afin que de pouvoir notifier l'activité depuis le fragment. Le fragment devra juste appeler la *callback* dans son code, ce qui provoquera l'exécution d'une méthode de l'activité avec d'éventuelles paramètres.

Une interface de communication doit être définie, par exemple:

```
public interface OnMyFragmentEvent {
    public void onMyEvent(String articleUri); // callback avec une URI en paramètre
}
```

Puis, lorsque le fragment est associée à une activité, la méthode **onAttach** permet d'enregistrer l'activité:

```
public class MyDynamicFragment extends Fragment {
```

```

private OnMyFragmentEvent mListener;

public void onAttach(Activity activity) {
    super.onAttach(activity);
    try {
        mListener = (OnMyFragmentEvent) activity;
    } catch (ClassCastException e) {
        throw new ClassCastException(activity.toString() + " must implement OnArticleSelectedListener");
    }
}

```

A ce stade, l'application n'est plus fonctionnelle, car l'activité n'implémente pas l'interface **OnMyFragmentEvent**. Du coup, le fragment fait planter l'application s'il tente de notifier l'activité en faisant:

```
mListener.onMyEvent("My URI");
```

L'activité doit donc implémenter l'interface **OnMyFragmentEvent**:

```

public class MainActivity extends Activity implements OnMyFragmentEvent {
    @Override
    public void onMyEvent(String s) {
        Toast.makeText(getApplicationContext(), "Callback: " + s, Toast.LENGTH_SHORT).show();
    }
}

```

Mais...

Mais... à partir de l'API 23, la méthode **onAttach(Activity a)** est dépréciée. Il faut désormais utiliser **onAttach(Context context)**, où le contexte reçu en paramètre est l'activité. Dans le cas d'un téléphone sous l'API 23, les deux méthodes sont appelées.

```

@TargetApi(23)
@Override
public void onAttach(Context context) {
    super.onAttach(context);
    Toast.makeText(context, "via onAttach(Context)", Toast.LENGTH_SHORT).show();
    try {
        mListener = (OnMyFragmentEvent) context;
    } catch (ClassCastException e) {
        throw new ClassCastException(context.toString() + " must implement OnArticleSelectedListener");
    }
}

@SuppressWarnings("deprecation")
@Override
public void onAttach(Activity activity) {
    super.onAttach(activity);
    Toast.makeText(activity, "via onAttach(Activity)", Toast.LENGTH_SHORT).show();

    if (Build.VERSION.SDK_INT < Build.VERSION_CODES.M) {
        try {
            mListener = (OnMyFragmentEvent) activity;
        } catch (ClassCastException e) {
            throw new ClassCastException(activity.toString() + " must implement OnArticleSelectedListener");
        }
    }
}

```

Démonstration

[Video](#)

4 fragments spéciaux

DialogFragment:

Affiche un boîtier de dialogue flottant au-dessus de l'activité. A contrario d'un fragment incorporé dans une activité, ce fragment a sa propre fenêtre qui capture les événements utilisateurs.

ListFragment

Affiche une liste d'éléments qui sont gérés par un adapter (par exemple **SimpleCursorAdapter**), comme **ListActivity**. **ListFragment** fournit des callbacks comme **onListItemClick()**.

PreferenceFragment

Fournit une hiérarchie de préférence sous forme de liste, comme **PreferenceActivity**. Cela permet d'incorporer des préférences dans une activité plus complexe.

WebViewFragment

Un fragment qui encapsule une **WebView**

Et d'autres classes héritant de fragments

DialogFragment

DialogFragment se gère d'une manière particulière. On utilise la notion de *tag* pour récupérer le fragment. Ainsi, si un fragment existe déjà avec ce tag, on le récupère. Puis, au lieu de remplacer ce fragment par un nouveau, on utilise la méthode **show()** qui va faire apparaître ce fragment à l'écran, et faire le *commit()*. Quand le fragment est annulé, il est enlevé de l'activité.

Un extrait de l'exemple de la documentation est donné ci-dessous:

```
// DialogFragment.show() will take care of adding the fragment
// in a transaction. We also want to remove any currently showing
// dialog, so make our own transaction and take care of that here.
FragmentTransaction ft = getFragmentManager().beginTransaction();
Fragment prev = getFragmentManager().findFragmentByTag("dialog");
if (prev != null) {
    ft.remove(prev);
}
ft.addToBackStack(null);

// Create and show the dialog.
DialogFragment newFragment = MyDialogFragment.newInstance(mStackLevel);
newFragment.show(ft, "dialog");
```

Les retain Fragments

Un fragment peut être utilisé pour conserver des données ou des tâches persistantes et indépendantes de l'interface graphique. Par exemple, lorsque l'on tourne le téléphone, l'activité et ses fragments sont recréés. Cependant, on peut faire perdurer une tâche asynchrone ou des données de configuration dans un fragment indépendant et non inclus dans la *back stack*. On dénomme un tel fragment un *retain fragment*. Pour ce faire, au démarrage, on indique qu'il s'agit d'un *retain fragment*. Dans ce cas, ce fragment est conservé lorsqu'Android fait une opération de recréation d'activité.

```
public static class RetainedFragment extends Fragment {
    final Thread mThread = new Thread() { ... };
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setRetainInstance(true);
        mThread.start(); ... }
}
```

A nouveau, l'utilisation d'un *tag* prend son sens ici, pour retrouver ce fragment depuis l'interface graphique (cf. [cette API demo](#)):

```
FragmentManager fm = getFragmentManager();
// Check to see if we have retained the worker fragment.
mWorkFragment = (RetainedFragment) fm.findFragmentByTag("work");
// If not retained (or first time running), we need to create it.
if (mWorkFragment == null) {
    mWorkFragment = new RetainedFragment();
    mWorkFragment.setTargetFragment(this, 0);
    fm.beginTransaction().add(mWorkFragment, "work").commit(); }
}
```

3.6 Action bar



Pour ajouter une *action bar*, il suffit d'utiliser un thème l'ayant prévu, par exemple **Theme.AppCompat.Light** et de faire hériter votre activité de **AppCompatActivity**. Dans votre manifest, il faut modifier l'attribut **android:theme** du tag **application**:

```
<application android:theme="@style/Theme.AppCompat.Light" ... >
```

Le menu de l'action bar est contrôlé par la méthode **onCreateOptionsMenu(Menu menu)** qui charge un fichier *menu* depuis les ressources. L'attribut **app:showAsAction="ifRoom"** permet de demander d'ajouter cette action comme un bouton dans la barre d'action, si la place est disponible.

```
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu; this adds items to the action bar if it is present.
    getMenuInflater().inflate(R.menu.menu_main, menu);
    return true;
}
```

```
<menu ... tools:context=".MainActivity">
    <item android:id="@+id/action_settings" android:title="@string/action_settings" android:orderInCategory="100"
        app:showAsAction="ifRoom" />
    <item android:id="@+id/uneentree" android:title="Une entrée" />
    <item android:id="@+id/marecherche" android:title="Ma recherche" app:showAsAction="ifRoom"/>
</menu>
```



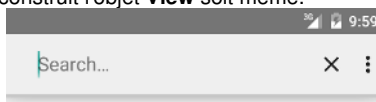
Gérer les actions sur l'action bar

Pour filtrer les clicks de l'utilisateur sur un des boutons de l'action bar, il faut utiliser la méthode *onOptionsItemSelected* de votre activité:

```
public boolean onOptionsItemSelected(MenuItem item) {
    int id = item.getItemId();
    if (id == R.id.marecherche) {
        Toast.makeText(getApplicationContext(), "Recherche depuis l'action bar !", Toast.LENGTH_LONG).show();
    }
    return super.onOptionsItemSelected(item);
}
```

Il est possible d'élaborer des *action bar* avec des actions plus complexes:

- un champ de recherche: avec l'attribut **app:actionViewClass="android.support.v7.widget.SearchView"**
- une action personnalisée:
 - **ShareActionProvider**: pour partager un élément dans l'écosystème Android
 - une action personnalisée: on doit construire l'objet **View** soit même.



Pour aller plus, se référer à la documentation de l'[Action bar](#).

Menu déroulant vs actions avec icône

Par défaut, les entrées du menu se retrouvent dans le menu déroulant, en haut à droite. Il est possible de mettre certaines entrées dans la barre de l'action bar, sous forme d'icône. Pour se faire, il faut préciser que l'attribut **app:showAsAction="always"**.

Attention, cette option n'est pas dans le même *namespace* que les autres attributs d'un *item*. En effet, il y a deux namespace à utiliser:

- <http://schemas.android.com/apk/res/android>: le namespace Android
- <http://schemas.android.com/apk/res-auto>: un namespace utilisé par appcompat-v7 pour des attributs spéciaux qui n'existe que dans des versions ultérieures d'Android (c'est le cas ici) cf [explications](#)

Ainsi, le menu sera au final:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto" >

  <item
    android:id="@+id/toast_me"
    android:icon="@drawable/icône"
    app:showAsAction="always"
    android:title="@string/toast_me"/>

</menu>
```

3.7 Animations et helpers

Certaines classes helpers permettent d'organiser l'espace de l'écran ou de réaliser des animations pour changer l'état de l'écran.

- L'animation des gabarits
- Les onglets: une combinaison de l'*action bar* et de fragments
- Le défilement à gauche ou à droite: en utilisant **ViewPager**, **PagerAdapter** et des fragments

Et d'autres animations que je ne détaille pas par la suite:

- fondu enchainé de deux activités
- animation de retournement de carte
- effet de zoom sur un élément

L'animation des gabarits

L'animation d'un gabarit est simple à mettre en oeuvre: il suffit par exemple d'ajouter l'attribut **android:animateLayoutChanges="true"** à un **LinearLayout** pour que l'animation soit jouée quand on ajoute un élément à ce gabarit:

```
mContainerView.addView(newView, 0);
```

Video

(Vidéo d'exemple de la documentation Google)

Les onglets

La réalisation d'onglets permet de mieux utiliser l'espace réduit de l'écran. Pour réaliser les onglets, les choses se sont considérablement simplifiées depuis l'API 11. Dans l'activité principale, il faut ajouter à l'*action bar* existante des onglet des objets de type **Tab** ayant obligatoirement un listener de type **TabListener**, comme présenté dans la documentation sur les [onglets](#):

```
final ActionBar actionBar = getActionBar();

// Specify that tabs should be displayed in the action bar.
actionBar.setNavigationMode(ActionBar.NAVIGATION_MODE_TABS);
// Create a tab listener that is called when the user changes tabs.
ActionBar.TabListener tabListener = new ActionBar.TabListener() {
    public void onTabSelected(ActionBar.Tab tab, FragmentTransaction ft) {
        // show the given tab
    }
    public void onTabUnselected(ActionBar.Tab tab, FragmentTransaction ft) {
        // hide the given tab
    }
}
```

```

    }
    public void onTabReselected(ActionBar.Tab tab, FragmentTransaction ft) {
        // probably ignore this event
    }
};
// Add 3 tabs, specifying the tab's text and TabListener
for (int i = 0; i < 3; i++) {
    Tab t = actionBar.newTab().setText("Tab " + (i + 1));
    t.setTabListener(tabListener);
    actionBar.addTab(t);
}

```

Il faut ensuite coder le changement d'écran lorsque l'événement *onTabSelected* survient. On utilise pour cela l'objet **FragmentTransaction** pour lequel on passe l'*id* du composant graphique à remplacer par un objet de type **Tab**:

```

public void onTabSelected(ActionBar.Tab tab, FragmentTransaction ft) {
    FragmentTab newft = new FragmentTab();
    if (tab.getText().equals("Tab 2")) {
        newft.setChecked(true);
    }
    ft.replace(R.id.fragmentContainer, newft);
}

```

La classe **FragmentTab** hérite de **Fragment**: c'est une sous partie d'activité qui a son propre cycle de vie. Dans cet exemple, au lieu de faire 2 fragments différents, on surcharge la méthode *onCreateView* pour cocher la case à cocher:

```

public class FragmentTab extends Fragment {
    ...
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment1, container, false);
        CheckBox cb = (CheckBox) view.findViewById(R.id.checkBox1);
        cb.setChecked(this.checked);
        return view;
    }
}

```

Le gabarit de l'activité, est donc:

```

<RelativeLayout>
<TextView
    android:id="@+id/textView1"
    android:text="@string/hello_world" />
<FrameLayout
    android:id="@+id/fragmentContainer" />
</FrameLayout>
</RelativeLayout>

```

Et celui du fragment *fragment1.xml*:

```

<LinearLayout
    <CheckBox android:id="@+id/checkBox1" />
</LinearLayout>

```

Il faut bien remarquer que:

- *R.id.fragment1* est l'*id* d'un composant de type **ViewGroup** de l'activité.
- la méthode *inflate* crée la vue en se basant sur le layout *R.layout.fragment1*.

Démonstration

[Video](#)



Défilement d'écrans

Pour faire défiler des écrans en les poussant, on utilise **ViewPager** et **PagerAdapter** combiné à des fragments (cf. [documentation](#)).

Le gabarit de l'activité principale est un **ViewPager**:

```
<!-- activity_screen_slide.xml -->
<android.support.v4.view.ViewPager
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

Le gabarit de chaque élément qui sera poussé est un **ScrollView**:

```
<!-- fragment_screen_slide_page.xml -->
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/content"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <TextView style="?android:textAppearanceMedium"
        android:padding="16dp"
        android:lineSpacingMultiplier="1.2"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/lorem_ipsum" />
</ScrollView>
```

Chaque élément qui défile est un fragment:

```
public class ScreenSlidePageFragment extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        ViewGroup rootView = (ViewGroup) inflater.inflate(
            R.layout.fragment_screen_slide_page, container, false);

        return rootView;
    }
}
```

L'activité instancie un **ScreenSlidePagerAdapter** (l'adaptateur):

```
protected void onCreate(Bundle savedInstanceState) {
    ...
    // Instantiate a ViewPager and a PagerAdapter.
    mPager = (ViewPager) findViewById(R.id.pager);
    mPagerAdapter = new ScreenSlidePagerAdapter(getSupportFragmentManager());
    mPager.setAdapter(mPagerAdapter);
}
```

L'adaptateur est, comme pour les listes, le fournisseur de fragments:

```
private class ScreenSlidePagerAdapter extends FragmentStatePagerAdapter {
    public ScreenSlidePagerAdapter(FragmentManager fm) {
        super(fm);
    }

    public Fragment getItem(int position) {
        return new ScreenSlidePageFragment();
    }

    public int getCount() { return NUM_PAGES;
}
```



```
    }  
  }  
}
```

Une particularité de cette activité est que l'appui sur le bouton *back* doit être surchargé. En effet, il faut faire reculer le **Pager**, ou bien quitter s'il n'y a plus d'items:

```
public void onBackPressed() {  
    if (mPager.getCurrentItem() == 0) {  
        // If the user is currently looking at the first step, allow the system to handle the  
        // Back button. This calls finish() on this activity and pops the back stack.  
        super.onBackPressed();  
    } else {  
        // Otherwise, select the previous step.  
        mPager.setCurrentItem(mPager.getCurrentItem() - 1);  
    }  
}
```

Demonstration

[Video](#)

(Vidéo d'exemple de la documentation Google)

4 Les Intents

4.1 Principe des Intents	34
4.2 Intents pour une nouvelle activité	34
Retour d'une activité	35
Résultat d'une activité	35
Principe de la Démonstration	36
Démonstration	36
4.3 Ajouter des informations	36
4.4 Types d'Intent: actions	36
Types d'Intent: catégories	37
4.5 Broadcaster des informations	37
4.6 Recevoir et filtrer les Intents	37
Filtrage d'un Intent par l'activité	37
Utilisation de catégories pour le filtrage	38
Réception par un BroadcastReceiver	38
Récepteur d'Intent dynamique	39
Les messages natifs	39
Principe de la Démonstration	39
Démonstration	40

4.1 Principe des Intents

Les *Intents* permettent de gérer l'envoi et la réception de messages afin de faire coopérer les applications. Le but des *Intents* est de déléguer une action à un autre composant, une autre application ou une autre activité de l'application courante.

Un objet **Intent** contient les informations suivantes:

- le nom du composant ciblé (facultatif)
- l'action à réaliser, sous forme de chaîne de caractères
- les données: contenu MIME et URI
- des données supplémentaires sous forme de paires clef/valeur
- une catégorie pour cibler un type d'application
- des drapeaux (informations supplémentaires)

On peut envoyer des *Intents* informatifs pour faire passer des messages. Mais on peut aussi envoyer des *Intents* servant à lancer une nouvelle activité.

4.2 Intents pour une nouvelle activité

Il y a plusieurs façons de créer l'objet de type *Intent* qui permettra de lancer une nouvelle activité. Si l'on passe la main à une activité interne à l'application, on peut créer l'Intent et passer la classe de l'activité ciblée par l'Intent:

```
Intent login = new Intent(getApplicationContext(), GiveLogin.class);
startActivity(login);
```

Le premier paramètre de construction de l'Intent est en fait le contexte de l'application. Dans certains cas, il ne faut pas mettre **this** mais faire appel à **getApplicationContext()** si l'objet manipulant l'Intent n'hérite pas de **Context**.

S'il s'agit de passer la main à une autre application, on donne au constructeur de l'Intent les données et l'URI cible: l'OS est chargé de trouver une application pouvant répondre à l'Intent.

```

Button b = (Button)findViewById(R.id.Button01);
b.setOnClickListener(new OnClickListener() {
public void onClick(View v) {
    Uri telnumber = Uri.parse("tel:0248484000");
    Intent call = new Intent(Intent.ACTION_DIAL, telnumber);
    startActivity(call);
}
});

```

Sans oublier de déclarer la nouvelle activité dans le Manifest.

Retour d'une activité

Lorsque le bouton *retour* est pressé, l'activité courante prend fin et revient à l'activité précédente. Cela permet par exemple de terminer son appel téléphonique et de revenir à l'activité ayant initié l'appel.

Au sein d'une application, une activité peut vouloir récupérer un code de retour de l'activité "enfant". On utilise pour cela la méthode **startActivityForResult** qui envoie un code de retour à l'activité enfant. Lorsque l'activité parent reprend la main, il devient possible de filtrer le code de retour dans la méthode **onActivityResult** pour savoir si l'on revient ou pas de l'activité enfant.

L'appel d'un *Intent* devient donc:

```

public void onCreate(Bundle savedInstanceState) {
    Intent login = new Intent(getApplicationContext(), GivePhoneNumber.class);
    startActivityForResult(login, 48);
    ... }

```

Le filtrage dans la classe parente permet de savoir qui avait appelé cette activité enfant:

```

protected void onActivityResult(int requestCode, int resultCode, Intent data)
{
    if (requestCode == 48)
        Toast.makeText(this, "Code de requête récupéré (je sais d'ou je viens)",
            Toast.LENGTH_LONG).show();
}

```

Résultat d'une activité

Il est aussi possible de définir un résultat d'activité, avant d'appeler explicitement la fin d'une activité avec la méthode **finish()**. Dans ce cas, la méthode **setResult** permet d'enregistrer un code de retour qu'il sera aussi possible de filtrer dans l'activité parente.

Dans l'activité enfant, on met donc:

```

Button finish = (Button)findViewById(R.id.finish);
finish.setOnClickListener(new OnClickListener() {

@Override
public void onClick(View v) {
    setResult(50);
    finish();
}
});

```

Et la classe parente peut filtrer ainsi:

```

protected void onActivityResult(int requestCode, int resultCode, Intent data)
{
    if (requestCode == 48)
        Toast.makeText(this, "Code de requête récupéré (je sais d'ou je viens)",
            Toast.LENGTH_LONG).show();
    if (resultCode == 50)
        Toast.makeText(this, "Code de retour ok (on m'a renvoyé le bon code)",

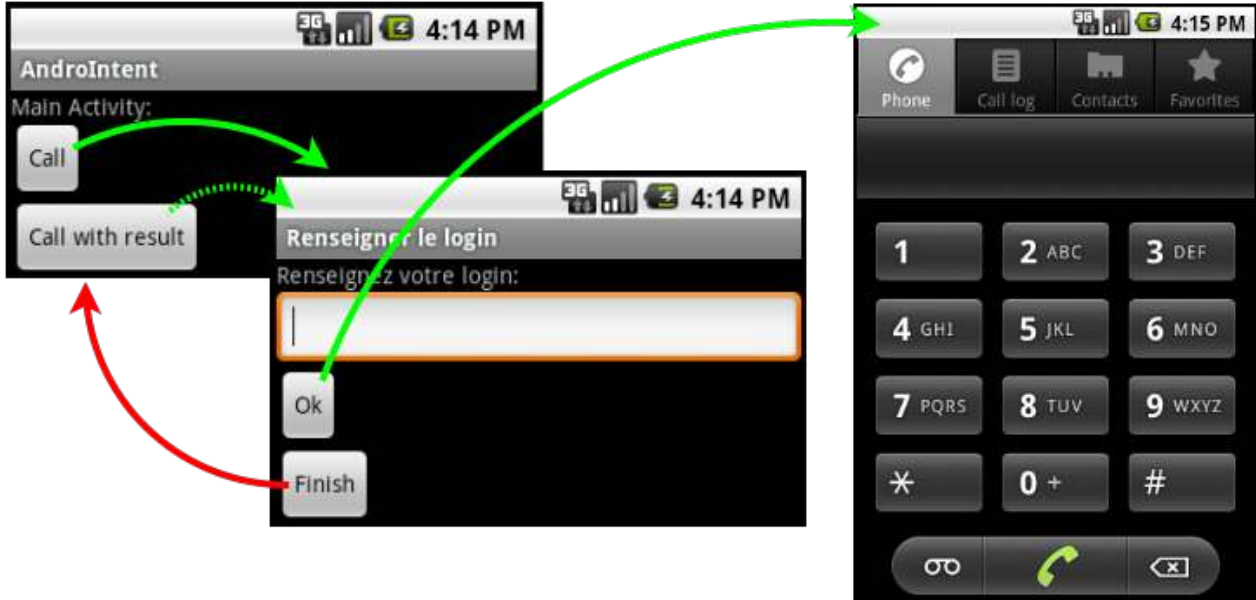
```

```

    Toast.LENGTH_LONG).show();
}

```

Principe de la Démonstration



(cf Code-Intents)

Démonstration

[Video](#)

4.3 Ajouter des informations

Les *Intent* permettent de transporter des informations à destination de l'activité cible. On appelle ces informations des *Extra*: les méthodes permettant de les manipuler sont **getExtra** et **putExtra**. Lorsqu'on prépare un *Intent* et que l'on souhaite ajouter une information de type "clef -> valeur" on procède ainsi:

```

Intent callactivity2 = new Intent (getApplicationContext (), Activity2.class);
callactivity2.putExtra ("login", "jfl");
startActivity (callactivity2);

```

Du côté de l'activité recevant l'Intent, on récupère l'information de la manière suivante:

```

Bundle extras = getIntent ().getExtras ();
String s = new String (extras.getString ("login"));

```

4.4 Types d'Intent: actions

Le premier paramètre de construction de l'*Intent* est le type de l'action véhiculé par cet Intent. Ces types d'actions peuvent être les actions natives du système ou des actions définies par le développeur.

Plusieurs actions natives existent par défaut sur Android. La plus courante est l'action **Intent.ACTION_VIEW** qui permet d'appeler une application pour visualiser un contenu dont on donne l'URI.

Voici un exemple d'envoi d'email (issu de [E-mail]). Dans ce cas ou l'on utilise l'action native, il faut ajouter des informations supplémentaires à l'aide de **putExtra**:

```
Intent emailIntent = new Intent(android.content.Intent.ACTION_SEND);
String[] recipients = new String[]{"my@email.com", ""};
emailIntent.putExtra(android.content.Intent.EXTRA_EMAIL, recipients);
emailIntent.putExtra(android.content.Intent.EXTRA_SUBJECT, "Test");
emailIntent.putExtra(android.content.Intent.EXTRA_TEXT, "Message");
emailIntent.setType("text/plain");
startActivity(Intent.createChooser(emailIntent, "Send mail..."));
finish();
```

Pour définir une action personnelle, il suffit de créer une chaîne unique:

```
Intent monIntent = new Intent("andro.jf.nom_du_message");
```

Types d'Intent: catégories

Les catégories d'Intent permettent de grouper les applications par grands types de fonctionnalités (clients emails, navigateurs, players de musique, etc...). Par exemple, on trouve les catégories suivantes qui permettent de lancer:

- **DEFAULT**: catégorie par défaut
- **BROWSABLE**: une activité qui peut être invoquée depuis un clic sur un navigateur web, ce qui permet d'implémenter des nouveaux types de lien, e.g. foo://truc
- **APP_MARKET**: une activité qui permet de parcourir le market de télécharger des applications
- **APP_MUSIC**: une activité qui permet de parcourir et jouer de la musique
- ...

4.5 Broadcaster des informations

Il est aussi possible d'utiliser un objet **Intent** pour broadcaster un message à but informatif. Ainsi, toutes les applications pourront capturer ce message et récupérer l'information.

```
Intent broadcast = new Intent("andro.jf.broadcast");
broadcast.putExtra("extra", "test");
sendBroadcast(broadcast);
```

La méthode **putExtra** permet d'enregistrer une paire clef/valeur dans l'Intent. On peut récupérer les données à l'aide de la méthode **getExtras** dans l'objet **Bundle** qui est dans l'Intent:

```
Bundle extra = intent.getExtras();
String val = extra.getString("extra");
```

4.6 Recevoir et filtrer les Intents

Etant donné la multitude de messages véhiculés par des *Intent*, chaque application doit pouvoir facilement "écouter" les *Intents* dont elle a besoin. Android dispose d'un système de filtres déclaratifs permettant de définir dans le Manifest des filtres.

Un filtre peut utiliser plusieurs niveaux de filtrage:

- **action**: identifie le nom de l'Intent. Pour éviter les collisions, il faut utiliser la convention de nommage de Java
- **category**: permet de filtrer une catégorie d'action (DEFAULT, BROWSABLE, ...)
- **data**: filtre sur les données du message par exemple en utilisant android:host pour filtrer un nom de domaine particulier

Filtrage d'un Intent par l'activité

En déclarant un filtre au niveau du tag **activity**, l'application déclare les types de message qu'elle sait gérer et qui l'invoquent.

```
<activity android:name=".Main" android:label="@string/app_name">
  <intent-filter>
```

```

    <action android:name="andro.jf.nom_du_message" />
    <category android:name="android.intent.category.DEFAULT" />
  </intent-filter>
</activity>

```

Ainsi, l'application répond à la sollicitation des *Intents* envoyés par :

```

Button autoinvoc = (Button)findViewById(R.id.autoinvoc);
autoinvoc.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        Intent intent = new Intent("andro.jf.nom_du_message");
        startActivity(intent);
    }
});

```

Utilisation de catégories pour le filtrage

Les catégories d'*Intent* évoquées précédemment permettent de répondre à des événements dans un contexte particulier. L'exemple le plus utilisé est l'interception d'un clic lors d'une navigation sur le web ou un protocole particulier est prévu. Par exemple, les liens vers le market sont de la forme **market://**.

Pour par exemple répondre à un clic sur un lien **foo://ensi-bourges.fr/mapage**, il faut construire un filtre d'*Intent* utilisant la catégorie définissant ce qui est "navigable" tout en combinant cette catégorie avec un type d'action signifiant que l'on souhaite "voir" la ressource. Il faut en plus utiliser le tag *data* dans la construction du filtre :

```

<intent-filter>
  <action android:name="android.intent.action.VIEW" />
  <category android:name="android.intent.category.DEFAULT" />
  <category android:name="android.intent.category.BROWSABLE" />
  <data android:scheme="foo" android:host="ensi-bourges.fr" />
</intent-filter>

```

L'attribut *host* définit l'URI de l'autorité (on évite ainsi le phishing). L'attribut *scheme* définit la partie de gauche de l'URI. On peut ainsi lancer une activité lors d'un clic sur :

```

<a href="foo://ensi-bourges.fr/mapage">lien</a>

```

Réception par un *BroadcastReceiver*

Les messages broadcastés par les applications sont réceptionnés par une classe héritant de **BroadcastReceiver**. Cette classe doit surcharger la méthode **onReceive**. Dans le Manifest, un filtre doit être inséré dans un tag **receiver** qui pointe vers la classe se chargeant des messages.

Dans le Manifest :

```

<application android:icon="@drawable/icon" android:label="@string/app_name">
  <receiver android:name="MyBroadcastReceiver">
    <intent-filter>
      <action android:name="andro.jf.broadcast" />
      <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
  </receiver>
</application>

```

La classe héritant de **BroadcastReceiver** :

```

public final class MyBroadcastReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        Bundle extra = intent.getExtras();
        if (extra != null)

```

```

    {
        String val = extra.getString("extra");
        Toast.makeText(context, "Broadcast message received: " + val,
            Toast.LENGTH_SHORT).show();
    }
}

```

Récepteur d'Intent dynamique

Il est possible de créer un récepteur et d'installer le filtre correspondant dynamiquement.

```

private MyBroadcastReceiverDyn myreceiver;

public void onCreate(Bundle savedInstanceState) {
    // Broadcast receiver dynamique
    myreceiver = new MyBroadcastReceiverDyn();
    IntentFilter filtre = new IntentFilter("andro.jf.broadcast");
    registerReceiver(myreceiver, filtre);
}

```

Lorsque ce receiver n'est plus utile, il faut le désenregistrer pour ainsi libérer les ressources:

```
unregisterReceiver(myreceiver);
```

C'est particulièrement important puisque un filtre de réception reste actif même si l'application est éteinte (elle peut être relancée si un message la concernant survient).

Les messages natifs

Un certain nombre de messages sont diffusés par l'OS:

- **ACTION_BOOT_COMPLETED**: diffusé lorsque le système a fini son boot
- **ACTION_SHUTDOWN**: diffusé lorsque le système est en cours d'extinction
- **ACTION_SCREEN_ON / OFF**: allumage / extinction de l'écran
- **ACTION_POWER_CONNECTED / DISCONNECTED**: connexion / perte de l'alimentation
- **ACTION_TIME_TICK**: une notification envoyée toutes les minutes
- **ACTION_USER_PRESENT**: notification recue lorsque l'utilisateur délock son téléphone
- ...

Tous les [messages de broadcast](#) se trouvent dans la documentation des *Intents*.

D'autres actions permettent de lancer des applications tierces pour déléguer un traitement:

- **ACTION_CALL (ANSWER, DIAL)**: passer/réceptionner/afficher un appel
- **ACTION_SEND**: envoyer des données par SMS ou E-mail
- **ACTION_WEB_SEARCH**: rechercher sur internet

Principe de la Démonstration



Démonstration

Video

(cf Code-Intents)

5 Persistance des données

5.1 Différentes persistances	41
5.2 Préférences partagées	41
Représentation XML d'un menu de préférences	42
Activité d'édition de préférences	42
Attributs des préférences	42
Préférences: toggle et texte	43
Préférences: listes	43
Ecriture de préférences	44
Démonstration	44
5.3 Les fichiers	44
5.4 BDD SQLite	44
Lecture / Ecriture dans la BDD	45
5.5 XML	45
SAX parser	46
DOM parser	46

5.1 Différentes persistances

Android fournit plusieurs méthodes pour faire persister les données applicatives:

- la persistance des données de l'activité (cf [`Le SDK Android`_](#))
- un mécanisme de sauvegarde clé/valeur, utilisé pour les fichiers de préférences (appelé préférences partagées)
- des entrées sorties de type fichier
- une base de donnée basé sur SQLite

La persistance des données des activités est géré par l'objet **Bundle** qui permet de restaurer les **View** qui possède un *id*. S'il est nécessaire de réaliser une sauvegarde plus personnalisée, il suffit de recoder les méthodes **onSaveInstanceState** et **onCreate** et d'utiliser les méthodes qui permettent de lire/écrire des données sur l'objet **Bundle**.

Android fournit aussi automatiquement, la persistance du chemin de navigation de l'utilisateur, ce qui le renvoie à la bonne activité lorsqu'il appuie sur la touche Retour. La navigation vers le parent (bouton "up") n'est pas automatique car c'est le concepteur qui doit décider vers quelle activité l'application doit retourner quand on appuie sur "up". Elle peut être programmée dans le Manifest avec l'attribut `android:parentActivityName`.

5.2 Préférences partagées

La classe **SharedPreferences** permet de gérer des paires de clé/valeurs associées à une activité. On récupère un tel objet par l'appel à **getPreferences**:

```
SharedPreferences prefs = getPreferences(Context.MODE_PRIVATE);
String nom = prefs.getString("login", null);
Long taille = prefs.getLong("taille", null);
```

La méthode **getPreferences(int)** appelle en fait **getPreferences(String, int)** à partir du nom de la classe de l'activité courante. Le mode **MODE_PRIVATE** restreint l'accès au fichier créé à l'application. Les modes d'accès **MODE_WORLD_READABLE** et **MODE_WORLD_WRITABLE** permettent aux autres applications de lire/écrire ce fichier.

L'intérêt d'utiliser le système de préférences prévu par Android réside dans le fait que l'interface graphique associé à la modification des préférences est déjà programmé: pas besoin de créer l'interface de l'activité pour cela. L'interface sera générée automatiquement par Android et peut ressembler par exemple à cela:



Représentation XML d'un menu de préférences

Une activité spécifique a été programmée pour réaliser un écran d'édition de préférences. Il s'agit de **PreferenceActivity**. A partir d'une description XML des préférences, la classe permet d'afficher un écran composé de modificateurs pour chaque type de préférences déclarées.

Voici un exemple de déclarations de préférences XML, à stocker dans `res/xml/preferences.xml`:

```
<PreferenceScreen
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:key="first_preferencescreen">
    <CheckBoxPreference
        android:key="wifi enabled"
        android:title="WiFi" />
    <PreferenceScreen
        android:key="second_preferencescreen"
        android:title="WiFi settings">
        <CheckBoxPreference
            android:key="prefer wifi"
            android:title="Prefer WiFi" />
        ... other preferences here ...
    </PreferenceScreen>
</PreferenceScreen>
```

Activité d'édition de préférences

Pour afficher l'écran d'édition des préférences correspondant à sa description XML, il faut créer une nouvelle activité qui hérite de **PreferenceActivity** et simplement appeler la méthode **addPreferencesFromResource** en donnant l'id de la description XML:

```
public class MyPrefs extends PreferenceActivity {
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        addPreferencesFromResource(R.xml.preferences);
    }
}
```

Pour lancer cette activité, on crée un bouton et un **Intent** correspondant.

Attributs des préférences

Les attributs suivants sont utiles:

- **android:title**: La string apparaissant comme nom de la préférence
- **android:summary**: Une phrase permettant d'expliquer la préférence
- **android:key**: La clef pour l'enregistrement de la préférence

Pour accéder aux valeurs des préférences, on utilise la méthode **getDefaultSharedPreferences** sur la classe **PreferenceManager**. C'est la clef spécifiée par l'attribut **android:key** qui est utilisée pour récupérer la valeur choisie par l'utilisateur.

```
SharedPreferences prefs = PreferenceManager.getDefaultSharedPreferences(
    getApplicationContext());
String login = prefs.getString("login", "");
```

Des attributs spécifiques à certains types de préférences peuvent être utilisés, par exemple **android:summaryOn** pour les cases à cocher qui donne la chaîne à afficher lorsque la préférence est cochée. On peut faire dépendre une préférence d'une autre, à l'aide de l'attribut **android:dependency**. Par exemple, on peut spécifier dans cet attribut le nom de la clef d'une préférence de type case à cocher:

```
<CheckBoxPreference android:key="wifi" ... />
<EditTextPreference android:dependency="wifi" ... />
```

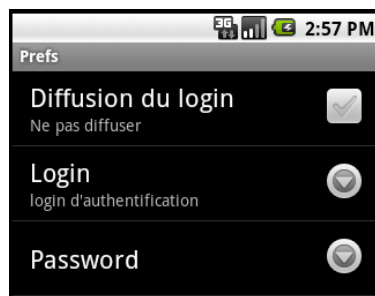
Préférences: toggle et texte

Une case à cocher se fait à l'aide de **CheckBoxPreference**:

```
<CheckBoxPreference android:key="wifi"
  android:title="Utiliser le wifi"
  android:summary="Synchronise l'application via le wifi."
  android:summaryOn="L'application se synchronise via le wifi."
  android:summaryOff="L'application ne se synchronise pas."
/>
```

Un champs texte est saisi via **EditTextPreference**:

```
<EditTextPreference android:key="login&"
  android:title="Login utilisateur"
  android:summary="Renseigner son login d'authentification."
  android:dialogTitle="Veuillez saisir votre login"
/>
```



Préférences: listes

Une entrée de préférence peut être liée à une liste de paires de clef-valeur dans les ressources:

```
<resources>
<array name="key"> <!-- Petite=1, Moyenne=5, Grande=20 -->
  <item>"Petite"</item>
  <item>"Moyenne"</item>
  <item>"Grande"</item>
</array>
<array name="value">
  <item>"1"</item>
  <item>"5"</item>
  <item>"20"</item>
</array>
</resources>
```

qui se déclare dans le menu de préférences:

```
<ListPreference android:title="Vitesse"
  android:key="vitesse"
  android:entries="@array/key"
```

```

    android:entryValues="@array/value"
    android:dialogTitle="Choisir la vitesse:"
    android:persistent="true">
</ListPreference>

```

Lorsque l'on choisit la valeur "Petite", la préférence *vitesse* est associée à "1".

Écriture de préférences

Il est aussi possible d'écraser des préférences par le code, par exemple si une action fait changer le paramétrage de l'application ou si l'on reçoit le paramétrage par le réseau.

L'écriture de préférences est plus complexe: elle passe au travers d'un éditeur qui doit réaliser un *commit* des modifications (*commit* atomique, pour éviter un mix entre plusieurs écritures simultanées):

```

SharedPreferences prefs = getPreferences(Context.MODE_PRIVATE);
Editor editor = prefs.edit();
editor.putString("login", "jf");
editor.commit();

```

Il est même possible de réagir à un changement de préférences en installant un écouteur sur celles-ci:

```

prefs.registerOnSharedPreferenceChangeListener(
    new OnSharedPreferenceChangeListener() {
        ...
    });

```

Démonstration

[Video](#)

(cf [Code-Preferences](#))

5.3 Les fichiers

Android fournit aussi un accès classique au système de fichier pour tous les cas qui ne sont pas couverts par les préférences ou la persistance des activités, c'est à dire de nombreux cas. Le choix de Google est de s'appuyer sur les classes classiques de Java EE tout en simplifiant la gestion des permissions et des fichiers embarqués dans l'application.

Pour la gestion des permissions, on retrouve comme pour les préférences les constantes **MODE_PRIVATE** et **MODE_WORLD_READABLE/WRITABLE** à passer en paramètre de l'ouverture du fichier. En utilisant ces constantes comme un masque, on peut ajouter | **MODE_APPEND** pour ajouter des données.

```

try {
    FileOutputStream out = openFileOutputStream("fichier", MODE_PRIVATE);
    ...
} catch (FileNotFoundException e) { ... }

```

Les ressources permettent aussi de récupérer un fichier embarqué dans l'application:

```

Resources res = getResources();
InputStream is = res.openRawResource(R.raw.fichier);

```

A noter: les fichier sont à éviter. Mieux vaut alléger l'application au maximum et prévoir le téléchargement de ressources nécessaires et l'utilisation de fournisseurs de contenu.

5.4 BDD SQLite

Android dispose d'une base de donnée relationnelle basée sur SQLite. Même si la base doit être utilisée avec parcimonie, cela fournit un moyen efficace de gérer une petite quantité de données.

[[DBSQL](#)] présente un exemple de création de base de données, ce qui se fait en héritant de **SQLiteOpenHelper**:



```

public class DictionaryOpenHelper extends SQLiteOpenHelper {

    private static final int DATABASE_VERSION = 2;
    private static final String DICTIONARY_TABLE_NAME = "dictionary";
    private static final String DICTIONARY_TABLE_CREATE =
        "CREATE TABLE " + DICTIONARY_TABLE_NAME + " (" +
        KEY_WORD + " TEXT, " +
        KEY_DEFINITION + " TEXT);";

    DictionaryOpenHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(DICTIONARY_TABLE_CREATE);
    }
}

```

Lecture / Ecriture dans la BDD

Pour réaliser des écritures ou lectures, on utilise les méthodes `getWritableDatabase()` et `getReadableDatabase()` qui renvoient une instance de `SQLiteDatabase`. Sur cet objet, une requête peut être exécutée au travers de la méthode `query()`:

```

public Cursor query (boolean distinct, String table, String[] columns,
    String selection, String[] selectionArgs, String groupBy,
    String having, String orderBy, String limit)

```

L'objet de type `Cursor` permet de traiter la réponse (en lecture ou écriture), par exemple:

- `getCount()`: nombre de lignes de la réponse
- `moveToFirst()`: déplace le curseur de réponse à la première ligne
- `getInt(int columnIndex)`: retourne la valeur (int) de la colonne passée en paramètre
- `getString(int columnIndex)`: retourne la valeur (String) de la colonne passée en paramètre
- `moveToNext()`: avance à la ligne suivante
- `getColumnName(int)`: donne le nom de la colonne désignée par l'index
- ...

5.5 XML

[XML] Sans surprise, Android fournit plusieurs parsers XML (*Pull parser*, *Document parser*, *Push parser*). SAX et DOM sont disponibles. L'API de *streaming* (StAX) n'est pas disponible (mais [pourrait le devenir](#)). Cependant, une librairie équivalente est disponible: `XmlPullParser`. Voici un exemple simple:

```

String s = new String("<plop><blup attr=\"45\">Coucou !</blup></plop>");
InputStream f = new ByteArrayInputStream(s.getBytes());
XmlPullParser parser = Xml.newPullParser();
try {
    // auto-detect the encoding from the stream
    parser.setInput(f, null);

    parser.next();
    Toast.makeText(this, parser.getName(), Toast.LENGTH_LONG).show();
    parser.next();
    Toast.makeText(this, parser.getName(), Toast.LENGTH_LONG).show();
    Toast.makeText(this, parser.getAttributeValue(null, "attr"),
        Toast.LENGTH_LONG).show();

    parser.nextText();
}

```

```
Toast.makeText(this, parser.getText(), Toast.LENGTH_LONG).show();
} ...
```

Video

SAX parser

SAX s'utilise très classiquement, comme en *Java SE*. Voici un exemple attaquant du XML au travers d'une connexion http:

```
URI u = new URI("https://www.site.com/api/xml/list?apikey=845ef");
DefaultHttpClient httpClient = new DefaultHttpClient();
HttpGet httpget = new HttpGet(u);
HttpResponse response = httpClient.execute(httpget);
HttpEntity entity = response.getEntity();
InputStream stream = entity.getContent();
InputSource source = new InputSource(stream);

SAXParserFactory spf = SAXParserFactory.newInstance();
SAXParser sp = spf.newSAXParser();
XMLReader xr = sp.getXMLReader();
DefaultHandler handler = new DefaultHandler() {
    @Override
    public void startElement(String uri, String localName,
        String qName, Attributes attributes) throws SAXException {

        Vector monitors = new Vector();

        if (localName.equals("monitor")) {
            monitors.add(attributes.getValue("displayname"));
        }
    }
};
xr.setContentHandler(handler);
xr.parse(source);
```

DOM parser

Enfin, le parser DOM permet de naviguer assez facilement dans la représentation arborescente du document XML. Ce n'est évidemment pas préconisé si le XML en question est volumineux. L'exemple suivant permet de chercher tous les tags "monitor" dans les sous-tags de la racine.

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
try {
    DocumentBuilder builder = factory.newDocumentBuilder();
    Document dom = builder.parse(source);
    Element root = dom.getDocumentElement();
    // Récupère tous les tags du descendant de la racine s'appelant monitor
    NodeList items = root.getElementsByTagName("monitor");
    for (int i=0; i<items.getLength(); i++){
        Node item = items.item(i);
        // Traitement:
        // item.getNodeName()
        // item.getTextContent()
        // item.getAttributes()
        ...
    }
} catch (...)
```

6 Programmation concurrente

6.1 Composants d'une application	47
6.2 Processus	48
Vie des processus	48
6.3 Threads	48
Threads et interface graphique	49
6.4 Services	49
Démarrer / Arrêter un service	50
Auto démarrage d'un service	50
onStartCommand()	50
Service dans le thread principal	51
Service dans un processus indépendant	51
Démonstration	51
6.5 Tâches concurrentes	51
Tâche régulière	52
Démonstration	52
Tâches asynchrones	52
La classe BitmapDownloaderTask	52
Méthodes de BitmapDownloaderTask	53
Méthodes de BitmapDownloaderTask	53
Démonstration	54
IntentService	54
Loaders	54
6.6 Bilan: processus et threads	54
6.7 Coopération service/activité	55
Lier activité et service: IBinder	55
Lier activité et service: ServiceConnexion	55
Démonstration	56
Service au travers d'IPC AIDL	56
Service exposant l'interface	56
6.8 Etude de cas	57
Service opérant dans onStart()	57
Service opérant dans onStart() + processus	57
Service opérant dans une tâche asynchrone	58
Service opérant dans onStart() + processus	58
Service opérant dans onStart() + processus	58

6.1 Composants d'une application

Une application Android n'est pas qu'une simple activité. Dans de nombreux cas, une application Android est amenée à tirer partie de la programmation concurrente afin de réaliser des tâches parallèles.

Dans ce chapitre, on s'intéresse à la notion de processus:



- processus "lourd" appelé par la suite "processus"
- processus "léger" appelé par la suite *thread*

Une fois ces deux notions présentées, nous verrons comment réaliser des tâches particulières qui sont souvent concurrente à l'activité principale. Ces tâches s'appuient principalement sur les *threads*, mais aussi sur un nouveau composant d'une application appelé "service".

6.2 Processus

[PT] Par défaut, une application android s'exécute dans un processus unique, le processus dit "principal". L'interface graphique s'exécute elle aussi au travers de ce processus principal. Ainsi, plus une application réalisera de traitement, plus la gestion de la programmation devient délicate car on peut arriver très rapidement à des problèmes de blocage de l'interface graphique par exemple.

Il est possible de séparer les composants d'une application en plusieurs processus. Ces composants sont les codes attachés aux tags `<activity>`, `<service>`, `<receiver>`, et `<provider>` de l'application. Par défaut, l'application (tag `<application>`) s'exécute dans le processus principal portant le même nom que le *package* de l'application. Bien que cela soit inutile, on peut le préciser dans le Manifest à l'aide de l'attribut **android:process**, par exemple:

```
<application android:process="andro.jf">
```

Les sous-tags du manifest, c'est-à-dire les composants de l'application héritent de cet attribut et s'exécutent donc dans le même processus. Si l'on souhaite créer un nouveau processus indépendant, on peut utiliser l'attribut **android:process** en préfixant le nom du processus par ":", par exemple:

```
<service android:name=".AndroService" android:process=":p2">
```

Pour cet exemple, le service et l'application sont indépendants. L'interface graphique pourra s'afficher indépendamment.

Vie des processus

Android peut devoir arrêter un processus à cause d'autres processus qui requièrent plus d'importance. Par exemple, un service peut être détruit car une application gourmande en ressources (navigateur web) occupe de plus en plus de mémoire. Plus classiquement, le processus affecté à une activité qui n'est plus visible a de grandes chances d'être détruit.

Ainsi, une *hiérarchie* permet de classer le niveau d'importance des processus:

1. Processus en avant plan (activité en interaction utilisateur, service attaché à cette activité, **BroadCastReceiver** exécutant **onReceive()**)
2. Processus visible: il n'interagit pas avec l'utilisateur mais peut influencer sur ce que l'on voit à l'écran (activité ayant affiché une boîte de dialogue (**onPause()** a été appelé), service lié à ces activité "visibles").
3. Processus de service
4. Processus tâche de fond (activité non visible (**onStop()** a été appelé))
5. Processus vide (ne comporte plus de composants actifs, gardé pour des raisons de *cache*)

Ainsi, il faut préférer l'utilisation d'un service à la création d'un thread pour accomplir une tâche longue, par exemple l'*upload* d'une image. On garantit ainsi d'avoir le niveau 3 pour cette opération, même si l'utilisateur quitte l'application ayant initié l'*upload*.

6.3 Threads

Dans le processus principal, le système crée un *thread* d'exécution pour l'application: le *thread* principal. Il est, entre autre, responsable de l'interface graphique et des messages/notifications/événements entre composants graphiques. Par exemple, l'événement générant l'exécution de **onKeyDown()** s'exécute dans ce *thread*.

Ainsi, si l'application doit effectuer des traitements longs, elle doit éviter de les faire dans ce thread. Cependant, il est interdit d'effectuer des opérations sur l'interface graphique en dehors du thread principal (aussi appelé *UI thread*), ce qui se résume dans la [documentation sur les threads](#) par deux règles:

- *Do not block the UI thread*
- *Do not access the Android UI toolkit from outside the UI thread*

L'exemple à ne pas faire est donné dans la documentation et recopié ci-dessous. Le comportement est imprévisible car le toolkit graphique n'est pas *thread-safe*.


```
public void onClick(View v) { // DO NOT DO THIS !
    new Thread(new Runnable() {
        public void run() {
            Bitmap b = loadImageFromNetwork("http://example.com/image.png");
            mImageView.setImageBitmap(b); }
    }).start(); }
```

Une exception peut parfois (pas toujours) être levée quand cela est détecté: **CalledFromWrongThreadException**.

Threads et interface graphique

Android fournit des méthodes pour résoudre le problème précédemment évoqué. Il s'agit de créer des objets exécutables dont la partie affectant l'interface graphique n'est pas exécutée mais déléguée à l'*UI thread* pour exécution ultérieure.

Par exemple, un appel à la méthode **View.post(Runnable)** permet de réaliser cela et donne, pour l'exemple précédent:

```
public void onClick(View v) {
    new Thread(new Runnable() {
        public void run() {
            final Bitmap bitmap =
                loadImageFromNetwork("http://example.com/image.png");
            mImageView.post(new Runnable() {
                public void run() {
                    mImageView.setImageBitmap(bitmap);
                }
            });
        }
    }).start(); }
```

La documentation donne les quelques méthodes utiles pour ces cas délicats:

- **Activity.runOnUiThread(Runnable)**
- **View.post(Runnable)**
- **View.postDelayed(Runnable, long)**

6.4 Services

La structure d'une classe de service ressemble à une activité. Pour réaliser un service, on hérite de la classe **Service** et on implémente les méthodes de création/démarrage/arrêt du service. La nouvelle méthode qu'il faut implémenter ici est **onBind** qui permet aux IPC de faire des appels à des méthodes distantes.

```
public class MyService extends Service {
    public void onCreate() {
        // Création du service
    }
    public void onDestroy() {
        // Destruction du service
    }
    @deprecated // utile pour les versions antérieures d'Android 2.0
    public void onStart(Intent intent, int startId) {
        // Démarrage du service
    }
    public int onStartCommand(Intent intent, int flags, int startId) {
        // Démarrage du service
        return START_STICKY;
    }
    public IBinder onBind(Intent arg0) {
        return null;
    }
}
```

Le service se déclare dans le Manifest dans le tag *application*:

```
<service android:name=".MyService"/>
```

Démarrer / Arrêter un service

Les *Intents* fournissent un moyen de démarrer/arrêter un service en utilisant le nom de la classe du service ou un identifiant d'action:

```
startService(new Intent(this, AndroService.class));
stopService(new Intent(this, AndroService.class));
startService(new Intent("andro.jf.manageServiceAction"));
```

Ces méthodes sont à invoquer dans l'activité de l'application développée. On dit alors que le service est *local* à l'application: il va même s'exécuter dans le *thread* de l'application. Un bouton de démarrage d'un service local est simple à coder:

```
Button b = (Button) findViewById(R.id.button1);
b.setOnClickListener(new OnClickListener() {

    public void onClick(View v) {
        Intent startService = new Intent("andro.jf.manageServiceAction");
        startService(startService);
    }
});
```

Auto démarrage d'un service

Il est aussi possible de faire démarrer un service **au démarrage du système**. Pour cela, il faut créer un **BroadcastReceiver** qui va réagir à l'action **BOOT_COMPLETED** et lancer l'**Intent** au travers de la méthode **startService**.

Le Manifest contient donc:

```
<application android:icon="@drawable/icon" android:label="@string/app_name">
<service android:name=".AndroService"></service>
<receiver android:name=".AutoStart">
<intent-filter>
    <action android:name="android.intent.action.BOOT_COMPLETED" />
</intent-filter>
</receiver>
</application>
```

Et la classe **AutoStart** doit être:

```
public class AutoStart extends BroadcastReceiver {
    public void onReceive(Context context, Intent intent) {
        Intent startServiceIntent = new Intent(context, AndroService.class);
        context.startService(startServiceIntent);
    }
}
```

onStartCommand()

Un problème subsiste après que l'application ait démarré le service. Typiquement, le service démarré exécute **onCreate()** puis **onStart()** qui va par exemple lancer un **Thread** pour réaliser la tâche de fond. Si le service doit être détruit par la plate-forme (pour récupérer de la mémoire), puis est récréé, seule la méthode **onCreate()** est appelée [API-service].

Afin de résoudre le problème précédent, une nouvelle méthode a fait son apparition dans les versions ultérieures à la version 5 de l'API. La méthode **onStartCommand()** est très similaire à **onStart()** mais cette méthode renvoie un entier qui permet d'indiquer au système ce qu'il faut faire au moment de la ré-instanciation du service, si celui-ci a dû être arrêté. La méthode peut renvoyer (cf [API-service]):

- **START_STICKY**: le comportement est similaire aux API<5. Si le service est tué, il est ensuite redémarré. Cependant, le système prend soin d'appeler à nouveau la méthode **onStartCommand(Intent intent)** avec un *Intent* null, ce qui permet au service qu'il vient d'être démarré après un *kill* du système.

- **START_NOT_STICKY**: le service n'est pas redémarré en cas de *kill* du système. C'est utile lorsque le service réagit à l'envoi d'un *Intent* unique, par exemple une alarme qui envoie un *Intent* toutes les 15 minutes.
- **START_REDELIVER_INTENT**: similaire à **START_NOT_STICKY** avec en plus le rejoue d'un *Intent* si le service n'a pas pu finir de le traiter et d'appeler **stopSelf()**.

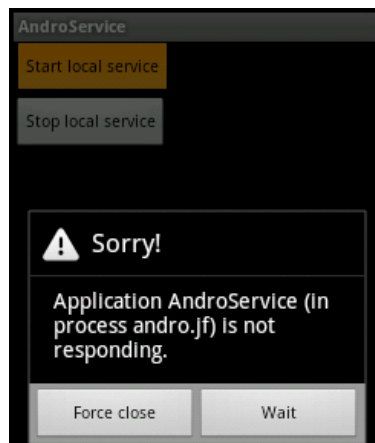
Dans les versions ultérieures à l'API 5, la commande **onStart()** se comporte comme **onStartCommand()** avec un ré-appel de la méthode avec un *Intent* null. Pour conserver le comportement obsolète de l'API (pas d'appel à **onStart()**), on peut utiliser le flag **START_STICKY_COMPATIBILITY**.

Service dans le thread principal

Le service invoqué par l'*Intent* de l'application s'exécute dans le thread de l'application. Cela peut être particulièrement ennuyeux si les traitements à effectuer sont coûteux, par exemple si l'on souhaite faire:

```
boolean blop = true;
while (blop == true)
;
```

On obtient:



Service dans un processus indépendant

Une manière de résoudre le problème précédent est de demander à Android d'exécuter le service dans un processus indépendant en le déclarant dans le Manifest:

```
<service android:name=".AndroService" android:process=":p2">
```

Cependant, cela n'autorise pas le service à réaliser un traitement long dans la méthode **onStart()** (comme une boucle infinie). La méthode **onStart()** sert à créer des threads, des tâches asynchrones, des tâches programmées, qui vont s'exécuter en tâche de fond.

En fait, isoler le service dans un processus à part rend la programmation plus difficile, car la programmation de la communication entre l'activité principale et le service devient plus difficile (il faut utiliser AIDL). Il est donc plus simple de laisser le service dans le processus principal de l'application et de lui faire lancer des threads ou tâches asynchrones.

Démonstration

Dans cette démonstration, le service, qui comporte une boucle infinie, s'exécute dans un processus indépendant de l'interface principale. Android va tuer le service qui occupe trop de temps processeur, ce qui n'affectera pas l'application et son interface (cf [Code-Process](#)).

[Video](#)

6.5 Tâches concurrentes

En plus des classes de programmation concurrentes de Java (**Thread**, **Executor**, **ThreadPoolExecutor**, **FutureTask**, **TimerTask**, etc...), Android fournit quelques classes supplémentaires pour programmer des tâches concurrentes (**AsyncTask**, **IntentService**, etc...).

Si la tâche concurrente est lancée:

- par l'activité principale: sa vie durera le temps de l'activité
- par un service: la tâche survivra à l'activité principale

Tâche régulière

S'il s'agit de faire une tâche répétitive qui ne consomme que très peu de temps CPU à intervalles réguliers, une solution ne nécessitant pas de processus à part consiste à programmer une tâche répétitive à l'aide d'un **TimerTask**.

```
final Handler handler = new Handler();
task = new TimerTask() {
    public void run() {
        handler.post(new Runnable() {
            public void run() {
                Toast.makeText(AndroService.this, "plop !", Toast.LENGTH_SHORT).show();
            }
        });
    }
};
timer.schedule(task, 0, 5000);
```

La création du **TimerTask** est particulièrement tarabiscotée mais il s'agissait de résoudre le [problème du lancement du Toast](#). Pour des cas sans *toast*, un simple appel à `new TimerTask() { public void run() { code; } }` suffit.

Attention à bien détruire la tâche programmée lorsqu'on détruit le service ! (sinon, le *thread* associé à la tâche programmée reste actif).

```
public void onDestroy() { // Destruction du service
    timer.cancel(); }
```

Démonstration

Dans cette démonstration, le service local à l'application est lancé: il affiche toutes les 5 secondes un *toast* à l'aide d'un **TimerTask**. Il est visible dans la section *running services*. En cliquant une seconde fois sur le bouton start, deux services s'exécutent en même temps. Enfin, les deux services sont stoppés. (cf [Code-Service](#))

[Video](#)

Tâches asynchrones

Dans [\[AT\]](#), Gilles Debunne présente comment gérer le chargement d'image de façon asynchrone afin d'améliorer la fluidité d'une application. Dans le cas d'une liste d'image chargée depuis le web, c'est même obligatoire, sous peine de voir planter l'application.

Pour réaliser cela, il faut se baser sur la classe **AsyncTask<U,V,W>** basée sur 3 types génériques, cf [\[ATK\]](#):

- U: le type du paramètre envoyé à l'exécution
- V: le type de l'objet permettant de notifier de la progression de l'exécution
- W: le type du résultat de l'exécution

A partir de l'implémentation d'[\[AT\]](#), nous proposons une implémentation utilisant 3 paramètres: l'url de l'image à charger, un entier représentant les étapes d'avancement de notre tâche et la classe **Bitmap** renvoyant l'image quand celle-ci est chargée, soit la classe **AsyncTask<String, Integer, Bitmap>**.

La classe **BitmapDownloaderTask**

Notre classe héritant de **AsyncTask** est donc dédiée à la gestion d'une tâche qui va impacter l'interface graphique. Dans notre exemple, la classe doit charger une image, et donc modifier un **ImageView**. Elle doit aussi gérer l'affichage de la progression de la tâche, ce que nous proposons de faire dans un **TextView**. Le constructeur de la classe doit donc permettre d'accéder à ces deux éléments graphiques:

```

public class BitmapDownloaderTask extends AsyncTask<String, Void, Bitmap> {

    private final WeakReference<ImageView> imageViewReference;
    private final WeakReference<TextView> textViewReference;

    public BitmapDownloaderTask(ImageView imageView, TextView textView) {
        imageViewReference = new WeakReference<ImageView>(imageView);
        textViewReference = new WeakReference<TextView>(textView);
    }
}

```

On notera l'emploi de *weak references* qui permet au *garbage collector* de détruire les objets graphiques même si le téléchargement de l'image est encore en cours.

Méthodes de *BitmapDownloaderTask*

void onPreExecute(): invoquée juste après que la tâche soit démarrée. Elle permet de modifier l'interface graphique juste après le démarrage de la tâche. Cela permet de créer une barre de progression ou de charger un élément par défaut. Cette méthode sera exécutée par l'*UI thread*.

```

protected void onPreExecute() {
    if (imageViewReference != null) {
        ImageView imageView = imageViewReference.get();
        if (imageView != null) {
            imageView.setImageResource(R.drawable.interro);
        }
    }
}

```

W doInBackground(U...): invoquée dans le thread qui s'exécute en tâche de fond, une fois que **onPreExecute()** est terminée. C'est cette méthode qui prendra un certain temps à s'exécuter. L'objet reçu en paramètre est de type **U** et permet de gérer la tâche à accomplir. Dans notre exemple, l'objet est l'url où télécharger l'image. A la fin, la tâche doit renvoyer un objet de type **W**.

```

protected Bitmap doInBackground(String... params) {
    String url = params[0];
    publishProgress(new Integer(0));
    AndroidHttpClient client = AndroidHttpClient.newInstance("Android");
    HttpGet getRequest = new HttpGet(url);
    HttpResponse response = client.execute(getRequest);
    publishProgress(new Integer(1));
    ... }
}

```

Méthodes de *BitmapDownloaderTask*

void onProgressUpdate(V...): invoquée dans le thread qui gère l'UI, juste après que la méthode **doInBackground(U...)** ait appelé **publishProgress(V...)**. On reçoit donc l'objet **V** qui contient les informations permettant de mettre à jour l'UI pour notifier de la progression de la tâche.

```

protected void onProgressUpdate(Integer... values) {
    Integer step = values[0];
    if (textViewReference != null) {
        textViewReference.get().setText("Step: " + step.toString());
    }
}

```

onPostExecute(W): invoquée quand la tâche est terminée et qu'il faut mettre à jour l'UI avec le résultat calculé dans l'objet de la classe **W**.

```

protected void onPostExecute(Bitmap bitmap) {
    if (imageViewReference != null) {
        ImageView imageView = imageViewReference.get();
        if (imageView != null) {
            imageView.setImageBitmap(bitmap);
        }
    }
}

```

Démonstration

(cf [Code-AsyncTask](#)) La démonstration suivante montre la notification au travers de l'entier passant de 0 à 3. Une image par défaut est chargée avant que l'image soit complètement téléchargée depuis le web et affichée à son tour.

Video

IntentService

[II] Un autre *design pattern* a été simplifié dans Android: l'exécution dans un processus isolé de *jobs* provenant d'une autre partie de l'application. Les *jobs* sont provisionnés au travers d'un **Intent** et l'**IntentService** exécute la tâche dans la méthode **onHandleIntent()**, indépendamment de l'*UI thread*.

```
public class MonServiceLourd extends IntentService {
{
    public MonServiceLourd() {
        super("MonServiceLourd");
    }
    @Override
    protected void onHandleIntent(Intent intent) {
        ... // Tâche longue, par exemple un téléchargement
    }
}
```

Le bénéfice, qui n'est pas visible ici, est que les **Intent** sont mis dans une file d'attente et traités les uns après les autres, évitant une surcharge du système. Chaque **Intent** est traité par **onHandleIntent()** et le service s'arrête tout seul quand il n'y a plus rien à traiter.

Loaders

Encore un autre *design pattern* est simplifié: le chargement au travers de **Loaders** et du **LoaderManager**.

La classe **Loader** permet de gérer le chargement en tâche de fond, si l'on utilise un **AsyncTaskLoader<D>**. Combinée à un **Cursor** cela permet de gérer indépendamment une interrogation de base de donnée au travers d'un **AsyncTaskLoader<Cursor>**. Une classe existe déjà pour ce cas: **CursorLoader**. Un **Loaders** peut-être créé *from scratch* mais il faut implémenter de nombreuses méthodes, et notamment celles qui gèrent le cas où les données surveillées changent.

La classe **LoaderManager** permet d'instancier un **Loader** (il faut surcharger des *callbacks* pour cela), qui va gérer les chargement. L'intérêt du **LoaderManager** est de conserver les données même si l'activité est rechargée par exemple à cause d'un changement de configuration. Cela économise le rechargement de ces données.

Le bénéfice attendu est le déchargement du thread principal et d'éviter tout ralentissement lorsque les données changent beaucoup ou sont volumineuses à traiter.

L'implémentation d'un *Loader* est assez technique. On peut se référer à [LLB] pour une explication assez complète.

6.6 Bilan: processus et threads

Par défaut, un service s'exécute donc dans le même processus que l'application, dans l'*UI thread*. Dans ce cas, le service doit programmer des tâches concurrentes qui s'exécuteront alors dans des threads indépendants. Ces threads survivront à l'activité principale car ils sont lancés depuis la classe héritant de **Service**.

On peut toutefois vouloir absolument caser un service dans un processus indépendant. Dans ce cas, le service et l'application sont séparés. La difficulté de la séparation du service et de l'application réside dans la difficulté de l'interaction de ces deux composants. Il faut alors utiliser les mécanismes prévus dans Android pour faire coopérer ces deux entités à l'aide de l'objet **IBinder** ou d'IPC AIDL.

Je ne résiste pas au plaisir de vous citer une discussion du googlegroup android-developer [LSAS], à propos des services persistants, ressemblant à des démons systèmes:

R. Ravichandran> *I have a need to create a background service that starts up during the system boot up, and keeps running until the device is powered down. There is no UI or Activity associated with this.*

Dianne Hackborn> *Mark answered how to do this, but please: think again about whether you really need to do this. Then think another time. And think once more. And if you are really really absolutely positively sure, this what you want to do, fine, but realize --*

On current Android devices, we can keep only a small handful of applications running at the same time. Having your application do this is going to take resources from other things that at any particular point in time would be better used elsewhere. And in fact, you can be guaranteed that your service will -not- stay running all of the time, because there is so much other stuff that wants to run (other background services that are only running when needed will be prioritized over yours), or needs more memory for what

the user is actually doing (running the web browser on complicated web pages is a great way to kick background stuff out of memory).

We have lots of facilities for implementing applications so they don't need to do this, such as alarms, and various broadcasts from events going on in the system. Please please please use them if at all possible. Having a service run forever is pretty close to the side of evil.

6.7 Coopération service/activité

Un service est souvent paramétré par une activité. L'activité peut être en charge de (re)démarrer le service, d'afficher les résultats du service, etc. Trois possibilités sont offertes pour faire coopérer service et activité *frontend*:

- utiliser des *Intents* pour envoyer des informations
- lier service et activité d'un même processus
- définir une interface de service en utilisant le langage AIDL

Lier le service et l'activité peut se faire assez simplement en définissant une interface (le contrat de communication entre le service et l'activité) et en utilisant la classe **Binder**. Cependant, cela n'est réalisable que si le service et l'activité font partie du même processus.

Pour deux processus séparés, il faut passer par la définition de l'interface en utilisant le langage AIDL. La définition de l'interface permettra à l'outil *aidl* de générer les stubs de communication inter processus permettant de lier une activité et un service.

Lier activité et service: **IBinder**

[BAS] Pour associer un service à une activité, le service doit implémenter la méthode **onBind()** qui renverra à l'application un objet de type **IBinder**. Au travers de l'objet **IBinder**, l'application aura accès au service. On implémente donc dans le service:

```
private final IBinder ib = new MonServiceBinder();
public IBinder onBind(Intent intent) {
    return ib;
}
```

et on crée une nouvelle classe **MonServiceBinder**, de préférence classe interne du service, qui implémente le *binder* c'est-à-dire fournit les méthodes que l'on pourra interroger depuis l'activité:

```
private int infoOfService = 0; // La donnée à transmettre à l'activité
private class MonServiceBinder extends Binder implements AndroServiceInterface {
    // Cette classe qui hérite de Binder implémente une méthode
    // définie dans l'interface AndroServiceInterface
    public int getInfo() {
        return infoOfService;
    }
}}
```

L'interface, à définir, déclare les méthodes qui seront accessibles à l'application:

```
public interface AndroServiceInterface {
    public int getInfo();
}
```

Lier activité et service: **ServiceConnexion**

Pour lier l'activité au service, un objet de la classe **ServiceConnexion** doit être instancié à partir de l'application. C'est cet objet qui fera le lien entre l'activité et le service. Deux méthodes de cet objet doivent être surchargées:

- **onServiceConnected** qui sera appelé lorsque le service sera connecté à l'application et qui permettra de récupérer un pointeur sur le binder.
- **onServiceDisconnected** qui permet de nettoyer les choses créées à la connexion, par exemple un pointeur d'attribut de classe sur l'objet **Binder**.

```
private int infoFromService = 0;
private ServiceConnexion maConnexion = new ServiceConnexion() {
```

```

public void onServiceConnected(ComponentName name, IBinder service) {
    AndroServiceInterface myBinder = (AndroServiceInterface)service;
    infoFromService = myBinder.getInfo();
}
public void onServiceDisconnected(ComponentName name) { }
};

```

Puis, dans le code de l'activité, on initie la connexion:

```

Intent intentAssociation =
    new Intent(AndroServiceBindActivity.this, AndroService.class);
bindService(intentAssociation, maConnexion, Context.BIND_AUTO_CREATE);
Toast.makeText(getApplicationContext(), "Info lue dans le service: "
    + infoFromService, Toast.LENGTH_SHORT).show();
unbindService(maConnexion);

```

Démonstration

Dans cette démonstration, un service lance un thread qui attendra 4 secondes avant d'enregistrer la valeur 12 comme résultat du service. A chaque click du bouton, l'activité se connecte au service pour *toaster* le résultat. (cf [Code-ServiceBind](#))

Video

Service au travers d'IPC AIDL

Le langage AIDL est très proche de la définition d'interfaces Java EE (mais pas tout à fait identique). Il s'agit de définir ce que le service est capable de faire et donc, quelles méthodes peuvent être appelées. Un peu comme un web service, la définition de l'interface permettra à l'outil AIDL de générer des stubs de communication pour les IPC. Ainsi, une autre application android pourra appeler le service au travers d'IPC.

Une interface en AIDL peut par exemple ressembler à:

```

package andro.jf;

interface AndroServiceInterface {
    int getInfo();
}

```

L'outil aidl va générer les *stubs* correspondants à l'interface dans un fichier .java portant le même nom que le fichier aidl, dans le répertoire *gen*.

```

// This file is auto-generated. DO NOT MODIFY.
package andro.jf;
public interface AndroServiceInterface extends android.os.IInterface {
    public static abstract class Stub extends android.os.Binder
        implements andro.jf.AndroServiceInterface {
    private static final java.lang.String DESCRIPTOR =
        "andro.jf.AndroServiceInterface";
    ...
}

```

Service exposant l'interface

Dans le service, on utilise alors l'implémentation générée par aidl pour implémenter ce que fait le service dans la méthode déclarée par l'interface:

```

private int infoOfService = 0; // La donnée à transmettre à l'activité
private AndroServiceInterface.Stub ib = new AndroServiceInterface.Stub() {
    @Override
    public int getInfo() throws RemoteException {
        return infoOfService;
    }
}

```



```
};
public IBinder onBind(Intent arg0) {
    return ib; }
}
```

Du côté de l'application, il faut aussi utiliser le code de la *stub* pour récupérer l'objet implémentant l'interface (cf [Code-ServiceBind2](#)):

```
private ServiceConnection maConnexion = new ServiceConnection() {
    public void onServiceConnected(ComponentName name, IBinder service) {
        AndroServiceInterface myBinder =
            AndroServiceInterface.Stub.asInterface(service);
        try { // stockage de l'information provenant du service
            infoFromService = myBinder.getInfo();
        }
    }
}
```

Il devient alors possible de déclarer le service comme appartenant à un autre processus que celui de l'activité, i.e. `android:process=":p2"` (cf. [Processus](#)).

6.8 Etude de cas

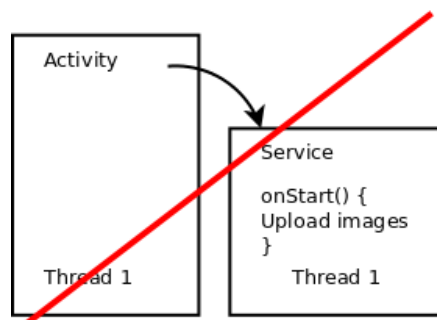
Cette section présente une étude de cas sur les solutions possible au problème suivant: on souhaite réaliser l'*upload* d'une image volumineuse en tâche de fond d'une activité qui peut éventuellement être fermée. Ce type de mécanisme est couramment employé par les applications qui chargent des images dans le *cloud*. Le but n'est pas d'étudier comment réaliser l'upload mais plutôt *où* le réaliser dans le code.

Les caractéristiques visées sont les suivantes:

- depuis l'application, on souhaite choisir une image et lancer l'upload
- l'upload doit se poursuivre si on ferme l'application
- si on lance plusieurs upload, l'idéal serait qu'ils se fassent les uns derrière les autres
- une notification sur l'avancement du chargement dans l'application principale serait un plus

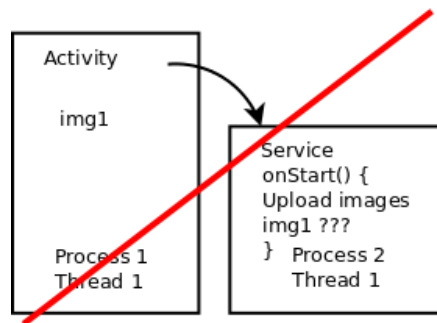
L'utilisation d'un service et d'une tâche programmée semble tout indiqué...

Service opérant dans onStart()



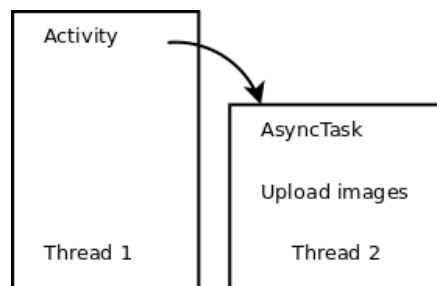
Cette solution n'est pas bonne: elle bloque l'activité principale, c'est-à-dire l'*UI thread* et donc l'interface utilisateur !

Service opérant dans onStart() + processus



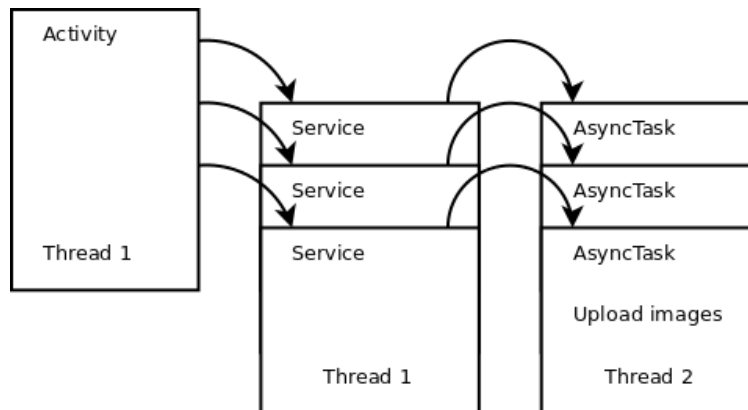
Cette solution est un peu mieux: l'interface de l'activité n'est plus bloquée. Cependant, en isolant le processus du service de celui de l'activité, il sera difficile de récupérer l'image.

Service opérant dans une tâche asynchrone



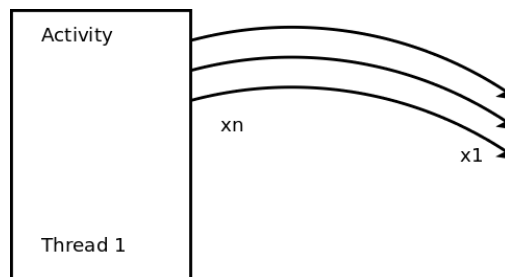
Dans cette solution, la tâche asynchrone aura tout loisir de charger l'image et de notifier l'activité principale. Cependant, si l'utilisateur quitte l'activité, l'*upload* s'arrête.

Service opérant dans onStart() + processus



En introduisant un service entre l'activité et la tâche synchronisée on résout le problème précédemment évoqué: la tâche survivra à l'activité. Dans une telle configuration, à chaque *upload* sera généré un nouveau service qui vivra le temps de la tâche.

Service opérant dans onStart() + processus



La manière la plus élégante d'opérer est d'utiliser un **IntentService**. L'**IntentService** va gérer toutes les requêtes d'*upload* dans une *working queue* qui sera dans un *thread* à part inclu dans un service.

7 Connectivité

7.1 Téléphonie	60
Passer ou déclarer savoir passer un appel	60
Envoyer et recevoir des SMS	61
7.2 Réseau	61
Gérer le réseau Wifi/Mobile	62
7.3 Bluetooth	62
S'associer en bluetooth	62
Utiliser le réseau	63
7.4 Localisation	63
Coordonnées	64
Alerte de proximité	64
Carte google map	64
Reverse Geocoding	65
7.5 Capteurs	65
Hardware	66
Lecture des données	66
7.6 Caméra	67
7.7 Secure element et NFC	67
Smart card	67
NFC: mode Host Card Emulation	68

7.1 Téléphonie

Les fonctions de téléphonie sont relativement simples à utiliser. Elles permettent de récupérer l'état de la fonction de téléphonie (appel en cours, appel entrant, ...), d'être notifié lors d'un changement d'état, de passer des appels et de gérer l'envoi et réception de SMS.

L'état de la téléphonie est géré par la classe **TelephonyManager** qui permet de récupérer le nom de l'opérateur, du téléphone, et l'état du téléphone. Pour lire ces informations, il est nécessaire de disposer de la permission **android.permission.CALL_PHONE**.

```
TelephonyManager tel =
    (TelephonyManager) getSystemService (Context.TELEPHONY_SERVICE);
int etat = tel.getCallState();
if (etat == TelephonyManager.CALL_STATE_IDLE)
    // RAS
if (etat == TelephonyManager.CALL_STATE_RINGING)
    // Le téléphone sonne
String SIMnb = tel.getSimSerialNumber();
```

Il est aussi possible d'être notifié d'un changement d'état en utilisant un écouteur:

```
public class Ecouteur extends PhoneStateListener {
    public void onCallStateChanged(int etat, String numero) {
        super.onCallStateChanged(etat, numero)
        if (etat == TelephonyManager.CALL_STATE_OFFHOOK)
            // Le téléphone est utilisé
    }
}
```

Passer ou déclarer savoir passer un appel



Il est bien sûr possible de passer un appel ou de déléguer l'appel, ces deux actions étant réalisées avec un *Intent* (attention aux permissions):

```
Uri telnumber = Uri.parse("tel:0248484000");
Intent call = new Intent(Intent.ACTION_CALL, telnumber);
startActivity(call);
```

```
Uri telnumber = Uri.parse("tel:0248484000");
Intent call = new Intent(Intent.ACTION_DIAL, telnumber);
startActivity(call);
```

Les applications qui peuvent passer des appels doivent filtrer ce type d'*Intent* pour pouvoir être invoquée lorsque l'*Intent* est lancé:

```
<receiver android:name=".ClasseGerantLAppel">
  <intent-filter>
    <action android:name="Intent.ACTION_CALL"/>
  </intent-filter>
</receiver>
```

Envoyer et recevoir des SMS

Si la permission `android.permission.SEND_SMS` est disponible, il est possible d'envoyer des SMS au travers de `SmsManager`:

```
SmsManager manager = SmsManager.getDefault();
manager.sendTextMessage("02484840000", null, "Coucou !", null, null);
```

Inversement, il est possible de créer un filtre d'*Intent* pour recevoir un SMS qui sera géré par un *broadcast receiver*. L'action à préciser dans le filtre d'*Intent* du receveur est `android.provider.Telephony.SMS_RECEIVED`:

```
<receiver android:name=".SMSBroadcastReceiver">
  <intent-filter>
    <action android:name="android.provider.Telephony.SMS_RECEIVED"/>
  </intent-filter>
</receiver>
```

Puis, le début du code du *broadcast receiver* est par exemple:

```
public final class MyBroadcastReceiver extends BroadcastReceiver {
  public void onReceive(Context context, Intent intent) {
    if (intent.getAction().equals(android.provider.Telephony.SMS_RECEIVED)) {
      String val = extra.getString("extra");
      Object[] pdus = (Object[]) intent.getExtras().get("pdus");
      SmsMessage[] messages = new SmsMessage[pdus.length];
      for (int i=0; i < pdus.length; i++)
        messages[i] = SmsMessage.createFromPdu((byte[]) pdus[i]);
    }
  }
}
```

7.2 Réseau

Le réseau (cf [Code-Network](#)) peut être disponible ou indisponible, suivant que le téléphone utilise une connexion Wifi, 3G, bluetooth, etc. Si la permission `android.permission.ACCESS_NETWORK_STATE` est déclarée, la classe `NetworkInfo` (depuis `ConnectivityManager`) permet de lire l'état de la connexion réseau parmi les constantes de la classe `State`: `CONNECTING`, `CONNECTED`, `DISCONNECTING`, `DISCONNECTED`, `SUSPENDED`, `UNKNOWN`.

```
ConnectivityManager manager =
  (ConnectivityManager) getSystemService(CONNECTIVITY_SERVICE);
NetworkInfo net = manager.getActiveNetworkInfo();
```

```
if (net.getState().compareTo(State.CONNECTED)
    // Connecté
```

Il est possible de connaître le type de la connexion:

```
int type = net.getType();
```

Le type est un entier correspondant, pour l'instant, au wifi ou à une connexion de type mobile (GPRS, 3G, ...).

- **ConnectivityManager.TYPE_MOBILE**: connexion mobile
- **ConnectivityManager.TYPE_WIFI**: wifi

Gérer le réseau Wifi/Mobile

Le basculement entre les types de connexion est possible si la permission **WRITE_SECURE_SETTINGS** est disponible. On utilise alors la méthode **setNetworkPreference** sur l'objet **ConnectivityManager** pour lui donner l'entier correspondant au type de connexion voulu. Par exemple:

```
manager.setNetworkPreference(ConnectivityManager.TYPE_WIFI);
```

L'accès au réseau wifi est gérable depuis une application, ce qui permet d'allumer ou de couper le wifi. L'objet **WifiManager** permet de réaliser cela.

```
WifiManager wifi = (WifiManager) getSystemService(Context.WIFI_SERVICE);
if (!wifi.isWifiEnabled())
    wifi.setWifiEnabled(true);
```

Les caractéristiques de la connexion Wifi sont accessibles par des appels statiques à des méthodes de **WifiManager**:

- force du signal projeté sur une échelle [0,levels]: **WifiManager.calculateSignalLevel(RSSI ?, levels)**
- vitesse du lien réseau: **info.getLinkSpeed()**
- les points d'accès disponibles: **List<ScanResult> pa = manager.getScanResults()**

7.3 Bluetooth

Le bluetooth se gère au travers de principalement 3 classes:

- **BluetoothAdapter**: similaire au **WifiManager**, cette classe permet de gérer les autres appareils bluetooth et d'initier les communications avec ceux-ci.
- **BluetoothDevice**: objet représentant l'appareil distant.
- **BluetoothSocket** et **BluetoothServerSocket**: gère une connexion établie.

Pour pouvoir utiliser les fonctionnalités bluetooth, il faut activer les permissions **android.permission.BLUETOOTH** et **android.permission.BLUETOOTH_ADMIN** pour pouvoir chercher des appareils ou changer la configuration bluetooth du téléphone.

```
BluetoothAdapter bluetooth = BluetoothAdapter.getDefaultAdapter();
if (!bluetooth.isEnabled())
{
    Intent launchBluetooth = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
    startActivity(launchBluetooth);
}
```

S'associer en bluetooth

Pour pouvoir associer deux appareils en bluetooth, il faut que l'un d'eux soit accessible (s'annonce) aux autres appareils. Pour cela, l'utilisateur doit autoriser le mode "découverte". L'application doit donc le demander explicitement via un **Intent**:

```
Intent discoveryMode = new Intent (BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE);
discoveryMode.putExtra (BluetoothAdapter.EXTRA_DISCOVERABLE_DURATION, 60);
startActivity (discoveryMode);
```

A l'inverse, si un appareil externe diffuse une annonce de découverte, il faut capturer les intents recus en broadcast dans le mobile:

```
public final class BluetoothBroadcastReceiver extends BroadcastReceiver {
    public void onReceive (Context context, Intent intent) {
        String action = intent.getAction();
        BluetoothDevice appareil = null;
        if (action.equals (BluetoothDevice.ACTION_FOUND))
            appareil = (BluetoothDevice)intent.getParcelableExtra (
                BluetoothDevice.EXTRA_DEVICE);
    }
}
```

Enfin les appareils associés se parcourent à l'aide d'un **Set<BluetoothDevice>**:

```
Set<BluetoothDevice> s = bluetooth.getBondedDevices();
for (BluetoothDevice ap : s)
    Toast.makeText (getApplicationContext(), "" + ap.getName(),
        Toast.LENGTH_LONG).show();
```

Utiliser le réseau

De nombreuses méthodes de développement permettent d'exploiter le réseau. Elles ne sont pas rappelées en détail ici (ce n'est pas l'objet du cours) et sont de toutes façons déjà connues:

- HTTP: **HttpClient**, **HttpResponse**
- SOAP: **SoapObjet**, **SoapSerializationEnvelope**
- REST:
 - JSON: **JSONObject**
 - XML: **DocumentBuilder**
- Sockets: **Socket**, **ServerSocket**
- Bluetooth: **BluetoothSocket**, **BluetoothServerSocket**

7.4 Localisation

(cf [Code-Location](#)) Comme pour le réseau, Android permet d'utiliser plusieurs moyens de localisation. Cela permet de rendre transparent l'utilisation du GPS, des antennes GSM ou des accès au Wifi. La classe **LocationManger** permet de gérer ces différents fournisseurs de position.

- **LocationManager.GPS_PROVIDER**: fournisseur GPS
- **LocationManager.NETWORK_PROVIDER**: fournisseur basé réseau

La liste de tous les fournisseurs s'obtient au travers de la méthode **getAllProviders()** ou **getAllProviders(true)** pour les fournisseurs activés:

```
LocationManager manager =
    (LocationManager) getSystemService (Context.LOCATION_SERVICE);
List<String> fournisseurs = manager.getAllProviders();
for (String f : fournisseurs) {
    Toast.makeText (getApplicationContext(), "" + f, Toast.LENGTH_SHORT).show();
    if (f.equals (LocationManager.GPS_PROVIDER))
        ...
}
```

Les permissions associées pour la localisation sont:



- `android.permission.ACCESS_FINE_LOCATION` via le GPS
- `android.permission.ACCESS_COARSE_LOCATION` via le réseau

Coordonnées

A partir du nom d'un fournisseur de position actif, il est possible d'interroger la dernière localisation en utilisant l'objet **Location**.

```
Location localisation = manager.getLastKnownLocation("gps");
Toast.makeText(getApplicationContext(), "Latitude" +
    localisation.getLatitude(), Toast.LENGTH_SHORT).show();
Toast.makeText(getApplicationContext(), "Longitude" +
    localisation.getLongitude(), Toast.LENGTH_SHORT).show();
```

Il est possible de réagir à un changement de position en créant un écouteur qui sera appelé à intervalles réguliers et pour une distance minimum donnée:

```
manager.requestLocationUpdates("gps", 6000, 100, new LocationListener() {
    public void onStatusChanged(String provider, int status, Bundle extras) {
    }
    public void onProviderEnabled(String provider) {
    }
    public void onProviderDisabled(String provider) {
    }
    public void onLocationChanged(Location location) {
        // TODO Auto-generated method stub
    }
});
```

Alerte de proximité

Il est possible de préparer un événement en vue de réagir à la proximité du téléphone à une zone. Pour cela, il faut utiliser la méthode **addProximityAlert** de **LocationManager** qui permet d'enregistrer un *Intent* qui sera envoyé lorsque des conditions de localisation sont réunies. Cette alerte possède une durée d'expiration qui la désactive automatiquement. La signature de cette méthode est:

addProximityAlert(double latitude, double longitude, float radius, long expiration, PendingIntent intent)

Il faut ensuite filtrer l'intent préparé:

```
IntentFilter filtre = new IntentFilter(PROXIMITY_ALERT);
registerReceiver(new MyProximityAlertReceiver(), filtre);
```

La classe gérant l'alerte est alors:

```
public class MyProximityAlertReceiver extends BroadcastReceiver {
    public void onReceive(Context context, Intent intent) {
        String key = LocationManager.KEY_PROXIMITY_ENTERING;
        Boolean entering = intent.getBooleanExtra(key, false);
    }
}
```

Carte google map

Google fournit [une librairie](#) qui permet d'inclure une carte google map dans son application. Voici quelques éléments pour utiliser la version 2 de cette API.

Le manifest doit tout d'abord déclarer la librairie et plusieurs permissions, dont notamment celle de l'API. Il faut donc ajouter au tag application un tag signifiant l'utilisation des google play services (à intégrer comme un projet Eclipse et à déclarer comme une dépendance) ainsi que votre clef d'accès à l'API:

```
<meta-data android:name="com.google.android.gms.version"
    android:value="@integer/google_play_services_version" />
```



```
<meta-data android:name="com.google.android.geo.API_KEY"
  android:value="AIzaS....."/>
```

La clef d'API est une clef à générer sur le serveur de google gérant le service des cartes à partir de votre clef de signature d'application. Votre clef de signature de travail générée par Eclipse suffit pour la phase de développement.

Ensuite, il faut un certain nombre de permissions, et déclarer une permission dépendante du nom de package et que l'on utilise soi même:

```
<permission
  android:name="jf.andro.mapv2.permission.MAPS_RECEIVE"
  android:protectionLevel="signature" />
<uses-permission android:name="jf.andro.mapv2.permission.MAPS_RECEIVE"/>

<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permission android:name="com.google.android.providers.gsf.permission.
  READ_GSERVICES"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission android:name="android.permission.INTERNET"/>
```

On peut alors utiliser un fragment, déclarant une carte:

```
<fragment
xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:map="http://schemas.android.com/apk/res-auto"
  android:name="com.google.android.gms.maps.MapFragment"
  android:id="@+id/map"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  map:mapType="normal" />
```

Reverse Geocoding

Une autre classe intéressante fournie par Android permet de retrouver une adresse à partir d'une localisation. Il s'agit de la classe **Geocoder** qui permet d'interroger un service Google à partir de coordonnées. Le code est assez simple à mettre en oeuvre:

```
Geocoder geocoder = new Geocoder(context, Locale.getDefault());
List<Address> addresses = geocoder.getFromLocation(loc.getLatitude(),
  loc.getLongitude(), 1);

String addressText = String.format("%s, %s, %s",
  address.getMaxAddressLineIndex() > 0 ? address.getAddressLine(0) : "",
  address.getLocality(),
  address.getCountryName());
```

Attention à utiliser l'émulateur contenant les "Google APIs" pour pouvoir utiliser ce service. Pour savoir si l'OS dispose du *backend* permettant d'utiliser la méthode **getFromLocation**, on peut appeler la méthode **isPresent()** qui doit renvoyer **true** dans ce cas. Un exemple de code utilisant une tâche asynchrone est donné dans [Code-Geocoding](#).

[Video](#)

7.5 Capteurs

(cf [Code-Capteurs](#)) Android introduit la gestion de multiples capteurs. Il peut s'agir de l'accéléromètre, du gyroscope (position angulaire), de la luminosité ambiante, des champs magnétiques, de la pression ou température, etc. En fonction, des capteurs présents, la classe **SensorManager** permet d'accéder aux capteurs disponibles. Par exemple, pour l'accéléromètre:

```
SensorManager manager = (SensorManager) getSystemService(SENSOR_SERVICE);
manager.registerListener( new SensorEventListener() {
  public void onSensorChanged(SensorEvent event) { // TODO method stub
  }
  public void onAccuracyChanged(Sensor sensor, int accuracy) { // TODO
```

```

    }}
    , manager.getDefaultSensor (Sensor.TYPE_ACCELEROMETER)
    , SensorManager.SENSOR_DELAY_UI );
}

```

Quand l'accéléromètre n'est plus utilisé, il doit être désenregistré à l'aide de:

```

manager.unregisterListener( pointeur ecouteur ,
    manager.getDefaultSensor (Sensor.TYPE_ACCELEROMETER) );

```

La fréquence d'interrogation influe directement sur l'énergie consommée. Cela va du plus rapide possible pour le système à une fréquence faible: `SENSOR_DELAY_FASTEST < SENSOR_DELAY_GAME < SENSOR_DELAY_NORMAL < SENSOR_DELAY_UI`.

Hardware

Pour déclarer l'utilisateur d'un capteur et avoir le droit d'accéder aux valeurs lues par le *hardware*, il ne faut pas oublier d'ajouter dans le Manifest la déclaration adéquat à l'aide du tag *uses-feature* [UF]. Cela permet à Google Play de filtrer les applications compatibles avec l'appareil de l'utilisateur. Par exemple, pour l'accéléromètre:

```

<uses-feature android:name="android.hardware.sensor.accelerometer"></uses-feature>

```

La directive n'est cependant pas utilisée lors de l'installation, ce qui signifie qu'il est possible d'installer une application utilisant le bluetooth sans posséder de *hardware* bluetooth. Evidemment, il risque d'y avoir une exception ou des dysfonctionnements. Un booléen supplémentaire permet alors d'indiquer si ce *hardware* est indispensable au fonctionnement de l'application:

```

<uses-feature android:name="xxx" android:required="true"></uses-feature>

```

L'outil `aapt` permet d'apprécier la façon dont Google Play va filtrer l'application, en donnant un dump des informations collectées:

```

./aapt dump badging ~/workspace/AndroSensors2/bin/AndroSensors2.apk

package: name='jf.andro.as' versionCode='1' versionName='1.0'
sdkVersion:'8'
targetSdkVersion:'14'
uses-feature:'android.hardware.sensor.accelerometer'

```

Lecture des données

Les valeurs sont récupérées au travers de la classe `SensorEvent`. Toutes les données sont dans un tableau, dont la taille dépend du type de capteur utilisé. Par exemple pour l'accéléromètre:

```

if (event.sensor.getType() == Sensor.TYPE_ACCELEROMETER)
{
    float x, y, z;
    x = event.values[0];
    y = event.values[1];
    z = event.values[2];
}

```

Le code est similaire pour le capteur d'orientation:

```

float azimuth, pitch, roll;
azimuth = event.values[0];
pitch = event.values[1];
roll = event.values[2];
}

```

Pour le capteur de lumière, on se limite à une seule valeur:

```
float lumiere;
lumiere = event.values[0];
```

7.6 Caméra

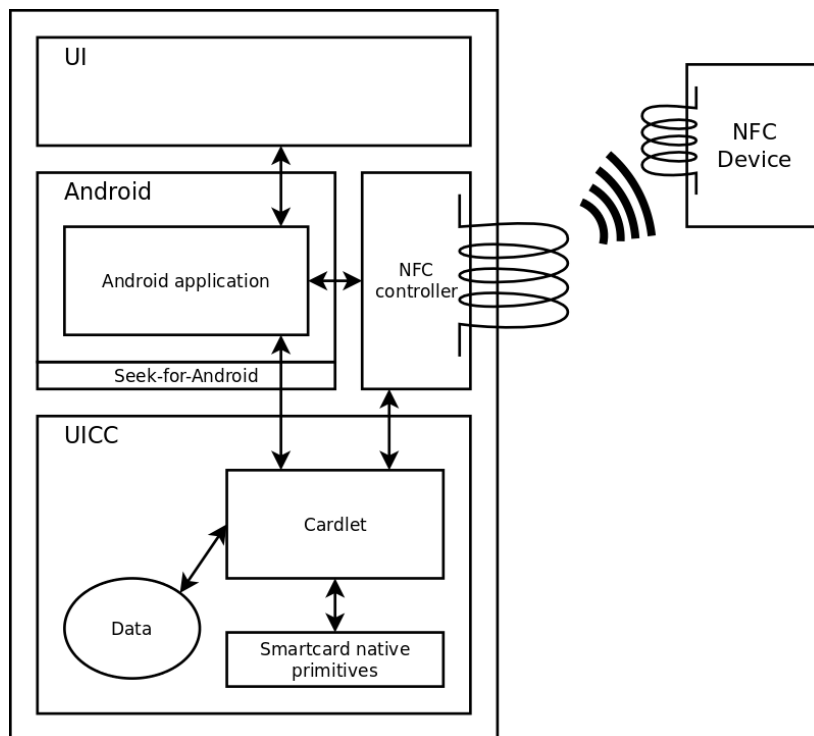
La documentation proposée par Google dans [CAM] permet de bien comprendre comment fonctionne la caméra. Il est souvent nécessaire de construire une zone de *preview* de ce que la caméra voit, avant par exemple de prendre une photo. Avant toute chose, il faut prévoir dans le *manifest* la permission nécessaire i.e. **android.permission.CAMERA**. Ensuite les grandes étapes sont les suivantes:

Dans l'activité principale on ajoute une **ViewGroup** spéciale et on ouvre la caméra (arrière, d'*id* 0, frontale, d'*id* 1):

```
Camera mCamera;
Preview mPreview = new Preview(this);
setContentView(mPreview);
Camera.open(id);
mPreview.setCamera(mCamera);
```

Avec une classe **Preview** héritant de **ViewGroup** et implémentant **SurfaceHolder.Callback**. Cela oblige à recoder les méthodes **surfaceChanged**, **surfaceCreated**, **surfaceDestroyed** et **onLayout** qui seront appelées quand la surface de rendu est effectivement créée. Notamment, il faut accrocher un objet **SurfaceHolder** à l'objet **Camera**, puis dire à l'objet **SurfaceHolder** que le **callback** est l'objet **Preview**. Plus de détails sont observables en annexes [Code-Camera](#).

7.7 Secure element et NFC



Smart card

```
import org.simalliance.openmobileapi.Channel;
import org.simalliance.openmobileapi.Reader;
import org.simalliance.openmobileapi.SEService;
import org.simalliance.openmobileapi.Session;

public static Reader[] getReaders(SEService seService) {
```

```

Reader[] readers = seService.getReaders();
return readers; }

public static void initChannel() throws IOException {
    Reader[] readers = getReaders(seService);
    Reader reader = readers[0];
    Session session = reader.openSession();
    card = session.openLogicalChannel(appletAID); }

public static byte[] transmit(byte[] apdu) throws IOException{
    return card.transmit(apdu); }

// Example:
ArrayList<String> apduList = new ArrayList<String>();
apduList.add(0, "00007000");
apduList.add(1, "00008000");
...
SendCard.initChannel();
for (int i = 0; i < apduList.size(); i++){
    cmd = SendCard.transmit(SendCard.stringToBytes(apduList.get(i))); }
SendCard.closeChannel();

```

NFC: mode Host Card Emulation

Il est possible d'émuler une carte à puce au niveau d'un service Android.

Déclaration de la classe:

```

<uses-permission android:name="android.permission.NFC" />
<uses-feature android:name="FEATURE_NFC_HOST_CARD_EMULATION"/>
<service android:name=".MyOffHostApuService"
    android:exported="true"
    android:permission="android.permission.BIND_NFC_SERVICE" >
    <intent-filter>
        <action android:name="android.nfc.cardemulation.action.OFF_HOST_APDU_SERVICE" />
    </intent-filter>
    <meta-data
        android:name="android.nfc.cardemulation.off_host_apdu_ervice"
        android:resource="@xml/apduservice" />
</service>

```

Filtrage des AID:

```

<offhost-apdu-service xmlns:android="http://schemas.android.com/apk/res/android"
    android:description="@string/servicedesc">
    <aid-group android:description="@string/subscription" android:category="other">
        <aid-filter android:name="F0010203040506"/>
        <aid-filter android:name="F0394148148100"/>
    </aid-group>
</offhost-apdu-service>

```

8 Développement client serveur

8.1 Architectures	69
Types d'applications	69
8.2 Applications Natives	70
8.3 Applications Hybrides	70
Surcharge du WebClient	70
Démonstration	70
JQueryMobile	70
JQueryMobile: exemple de liste	71
JQueryMobile: intégration hybride	71
8.4 Architectures REST	71
Exemple de client JSON	72
Exemple de serveur Tomcat	72

8.1 Architectures

Les applications clientes interagissant avec un serveur distant, qu'il soit dans un *cloud* ou non, peuvent permettre de déporter des traitements sur le serveur. La plupart du temps, on déporte les traitements pour:

- garder et protéger le savoir métier
- rendre l'utilisateur captif
- améliorer les performances lors de traitements lourds
- économiser les ressources du client

Le prix à payer est le temps de latence induit par le réseau, voire l'incapacité de l'application à fonctionner correctement si l'utilisateur n'a plus de réseau ou un réseau dégradé (e.g. 2G). L'expérience utilisateur peut aussi être largement impacté.

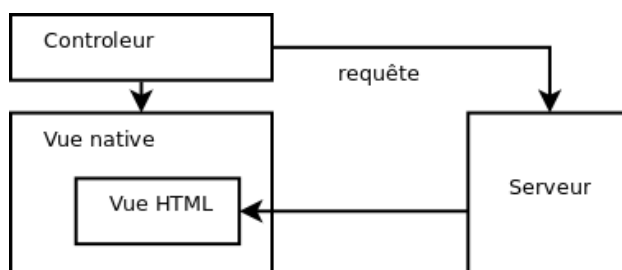
Lorsqu'un développeur réalise une application cliente, il utilise presque de manière inconsciente le modèle MVC. S'il ajoute la dimension réseau, il peut alors choisir de déporter des éléments du modèle MVC sur le serveur. En effet, il faut décider quels éléments du modèle MVC sont du côté client ou du côté serveur.

Types d'applications

Lorsqu'on réalise une application client-serveur, on peut considérer que trois choix d'architectures sont possibles pour l'emplacement du modèle MVC:

- application native: MVC sont codés en java, dans une machine virtuelle Dalvik
- application hybride: certains éléments du Contrôle sont codés en java, mais Modèles et Vues sont codés par le serveur
- application web: tout est réalisé sur le serveur

Dans ce cours, jusqu'à maintenant, il a toujours été présenté des applications natives. Une application hybride utilise le langage HTML5 et les capacités d'Android à afficher des pages web pour intégrer des composants de visualisation dans une application native. Ainsi, une partie de l'application est classiquement native, tandis que certaines parties de la vue sont des affichages de documents HTML calculés coté serveur.



8.2 Applications Natives

Même si elle native, une application peut tout à fait dépendre d'un serveur distant. L'interface graphique utilise des composants Android, mais la localisation et le traitement des données est partagé entre le téléphone et le serveur.

On peut décrire plusieurs cas différents:

- Native offline: tout est sur le client: interface graphique, données, traitements.
- Native + Cloud/API online ou partiellement offline: interface graphique sur le client mais données dans le cloud ou via un accès à une API avec des traitements possibles des deux côtés. Le mode partiellement offline nécessite de faire persister les données sur le téléphone, au moins partiellement.
- Native + Cloud/API online: interface graphique sur le client mais données à l'extérieur avec des traitements possibles des deux côtés.

8.3 Applications Hybrides

Pour réaliser une application hybride, il suffit d'utiliser un objet de type **WebView** qui, comme son nom l'indique, est un objet graphique fait pour visualiser une page web. Après l'avoir intégré dans le gabarit, il faut lui faire charger une page:

```
WebView wv = (WebView) findViewById(R.id.webView1);
wv.getSettings().setJavaScriptEnabled(true);
wv.loadUrl("http://www.univ-orleans.fr/lifo/Members/Jean-Francois.Lalande/");
```

Cependant, les liens web appellent tout de même le navigateur, ce qui est assez gênant:

[Video](#)

Surcharge du WebClient

Afin de ne pas lancer le navigateur par défaut, il faut surcharger le client web afin de recoder la méthode agissant lorsqu'un lien est cliqué. La classe est très simple:

```
private class MyWebViewClient extends WebViewClient {
    @Override
    public boolean shouldOverrideUrlLoading(WebView view, String url) {
        view.loadUrl(url);
        Toast.makeText(getApplicationContext(), "A link has been clicked! ",
            Toast.LENGTH_SHORT).show();
        return true;
    }
}
```

Du côté de l'activité, il faut **remplacer le comportement du client web par défaut**. On peut aussi prévoir un bouton afin de pouvoir revenir en arrière car le bouton *back* du téléphone va tuer l'activité.

```
wv.setWebViewClient(new MyWebViewClient());
Button back = (Button) findViewById(R.id.back);
back.setOnClickListener(new OnClickListener() {
    public void onClick(View v) {
        wv.goBack();
    }
});
```

Démonstration

Dans cet exemple (cf [Code-Hybride](#)), on navigue sur une page web incluse dans une activité. Un bouton transparent "Back" permet de contrôler la **WebView**:

[Video](#)

JQueryMobile



Evidemment, une page web classique n'est pas très adaptée à un téléphone ou une tablette. Il existe des technologies pour calculer le rendu javascript d'une page web en fonction du navigateur, ce qui est particulièrement pratique. On peut par exemple utiliser **jQueryMobile** pour cela, comme expliqué dans [JQM].

```

<!DOCTYPE html>
<html>
<head>
<title>My Page</title>
<meta name="viewport" content="width=device-width, initial-scale=1">
<link rel="stylesheet" href="http://code.jquery.com/mobile/1.2.0/
  jquery.mobile-1.2.0.min.css" />
<script src="http://code.jquery.com/jquery-1.8.2.min.js"></script>
<script src="http://code.jquery.com/mobile/1.2.0/jquery.mobile-1.2.0.min.js"/>
</head>
<body>

<div data-role="page">

<div data-role="header">
<h1>jQuery example</h1>
</div>

<div data-role="content">
<p>Hello world !</p>
</div><!-- /content -->

</div> </body> </html>

```

jQueryMobile: exemple de liste

Par exemple, une liste d'items se transforme facilement en éléments facilement cliquable dans un environnement tactile:

```

<ul data-role="listview" data-inset="true" data-filter="true">
<li><a href="http://www.univ-orleans.fr/lifo/Members/Jean-Francois.Lalande">
  Home</a></li>
<li><a href="publications.html">Publications</a></li>
<li><a href="teaching.html">Enseignement</a></li>
<li><a data-role="button" data-icon="star" data-theme="b"
  href="jquerymobiletest2.html">page 2</a></li>
</ul>

```

Video

jQueryMobile: intégration hybride

La combinaison de jquerymobile avec une webview donne alors tout son potentiel:

Video

8.4 Architectures REST

Les applications clientes Android peuvent tirer partie d'une architecture de type REST car de nombreuses technologies côté serveur sont disponibles. Les principes de REST sont bien résumés sur [la page wikipedia](#) leurs étant dédiée:

- les données sont stockées dans le serveur permettant au client de ne s'occuper que de l'affichage
- chaque requête est *stateless* c'est à dire qu'elle est exécutable du côté serveur sans que celui-ci ait besoin de retenir l'état du client (son passé i.e. ses requêtes passées)
- les réponses peuvent parfois être mises en cache facilitant la montée en charge
- les ressources (services) du serveur sont clairement définies; l'état du client est envoyé par "morceaux" augmentant le nombre de requêtes.

Pour bâtir le service du côté serveur, un serveur PHP ou Tomcat peut faire l'affaire. Il pourra fournir un web service, des données au format XML ou JSON. Du côté client, il faut alors implémenter un parseur SAX ou d'objet JSON, comme montré dans [JSONREST].



Exemple de client JSON

```

HttpClient httpClient = new DefaultHttpClient();
HttpGet httpget = new HttpGet(url);
HttpResponse response;
try {
    response = httpClient.execute(httpget);
    // Examine the response status
    Log.i("Praeda", response.getStatusLine().toString());

    // Get hold of the response entity
    HttpEntity entity = response.getEntity();
    if (entity != null) {

        // A Simple JSON Response Read
        InputStream instream = entity.getContent();
        String result= convertStreamToString(instream);
        Log.i("Praeda", result);

        // A Simple JSONObject Creation
        JSONObject json=new JSONObject(result);
        Log.i("Praeda", "<jsonobject>\n"+json.toString()+"\n</jsonobject>");

        // A Simple JSONObject Parsing
        JSONArray nameArray=json.names();
        JSONArray valArray=json.toJSONArray(nameArray);
        for(int i=0;i<valArray.length();i++)
        {
            Log.i("Praeda", "<jsonname"+i+">\n"+nameArray.getString(i)+
                "\n</jsonname"+i+">\n"
                +"<jsonvalue"+i+">\n"+valArray.getString(i)+"\n</jsonvalue"+i+">");
        }
    }
}

```

Exemple de serveur Tomcat

Du côté serveur, on peut par exemple utiliser une servlet tomcat pour capturer la requête.

```

public class SimpleServlet extends HttpServlet {
    public void init(ServletConfig c) throws ServletException {
        // init
    }
    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        Vector myVector = new Vector();
        String param = req.getParameter("param");
        PrintWriter out = response.getWriter();
        // traitements
    }
    public void destroy() { }
    public String getServletInfo() { }
}

```

La servlet peut même redéleguer la partie "Vue" à une JSP, en fin de traitement:

```

<title>
    <!-- get a value that has been set by the servlet -->
    <%= request.getAttribute("title") %>
</title>
<jsp:foreach collection="<%= myVector %>">
    <jsp:item ref="myItem" type="java.lang.Integer">
        <element> <%= myItem %> </element>
    </jsp:item>
</jsp:foreach>

```




9 Android Wear

9.1 Philosophie	74
Wearable and Handheld apps	74
Particularités d'un Wearable	75
9.2 UI Design	75
Layouts	76
Layouts alternatifs	77
Layout spécial	77
CardFragment	78
ListView	79
Applications à écrans multiples	80
Confirmations	80
Ambient Mode	81
9.3 Faire communiquer Handheld et Wearable	81
Notifications	82
Action callback	82
Boutons d'action	83
Envoyer un message au Wearable	83
9.4 Watchfaces	84

9.1 Philosophie

Android Wear est apparu courant 2014. Il répond à la problématique des objets connectés, et en premier lieu, des montres connectées. Pour l'instant, Android Wear suppose que votre montre soit appairée à votre téléphone en bluetooth, notamment pour le déploiement d'application. En effet, les appareils autonomes ayant accès à internet (et donc ayant du wifi ou une carte SIM) sont encore rares.

L'interface graphique est propre à Android Wear, bien que le système sous-jacent soit très similaire à Android. L'interface est adaptée à des écrans carrés ou ronds et une faible diagonale. Il a aussi fallu repenser la gestion de l'énergie pour économiser encore plus de batterie.

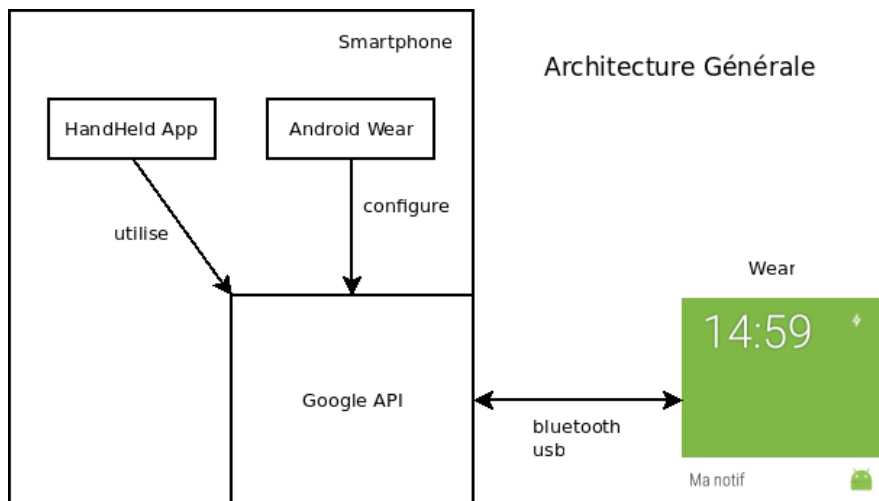
Le cas d'usage le plus basique est la gestion des notifications que l'on peut recevoir, annuler et plus généralement interagir avec elles.



Karl-Leo Spettmann - Eigenes Werk (CC-BY-SA 4.0)

Wearable and Handheld apps

Une application **Wearable** ne fonctionne en général pas seule. Elle est accompagnée d'une **Handheld app** c'est-à-dire d'une application compagnon qui va gérer l'application **Wearable**. D'un point de vue de l'installation, c'est l'installation de l'application **Handheld** qui va pousser la partie **Wearable**. D'un point de vue développement, on peut cependant installer avec *adb* l'application **Wearable** directement sur l'émulateur **Wear**. L'architecture générale peut être schématisée de la façon suivante:



Particularités d'un Wearable

Un Wearable possède deux modes:

- *Interactive mode*: en général en couleur, pour les interactions utilisateurs.
- *Ambient mode*: un mode économe en énergie où l'écran est en noir et blanc, voire en niveaux de gris. Il est prévu pour être toujours allumé.

Toutes les applications peuvent changer de mode, comme le montre Google sur sa page d'explication des modes:



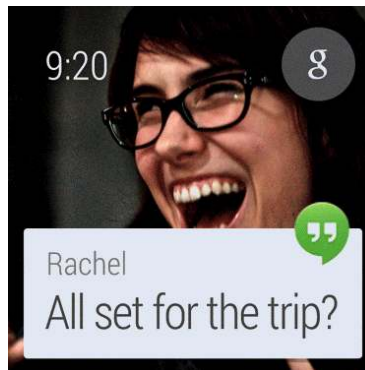
Google CC BY

En mode ambiant, l'application ne doit pas montrer de boutons (éléments interactifs) car il peut faire croire à l'utilisateur que la montre est en mode interactif. Elle doit aussi éviter d'afficher des informations trop privées (respect de la *privacy*). Les mises à jour doivent être rares, par exemple chaque minute.

9.2 UI Design

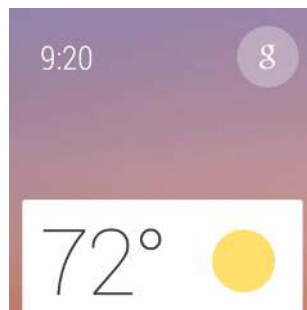
L'interface étant réduite, l'interface d'un wearable est basée sur deux fonctions principales: **Suggest** et **Demand**.

Suggest présente l'interface comme une liste de cartes verticales. Une carte contient du texte et on peut mettre en fond une image. Une carte représente une notification, dont le contenu peut être étendu en poussant la carte horizontalement. Cela permet de mettre des infos supplémentaires ou des boutons d'action.



Google - CC BY

Demand est une carte spéciale (Cue Card) qui permet de réaliser des actions vocales après avoir dit "Ok Google". Un développeur peut réaliser des actions lorsque les intents générés par la **Cue Card** sont envoyés.



Google - CC BY

Enfin, comme un téléphone, l'écran par défaut est le **Home Screen**, qui peut contenir une première carte, un écran personnalisé, par exemple l'affichage de l'heure, des indicateurs de batterie ou de connectivité, etc.

Il est tout de même possible de lancer une application en plein écran sans passer par les concepts **Suggest** ou **Demand**. Il faut pour cela construire une structure d'application compatible.

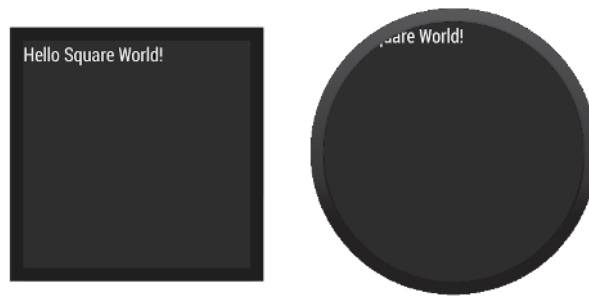
Layouts

Les *layouts* classiques Android peuvent être utilisés, par exemple:

```
<LinearLayout xmlns...
  android:orientation="vertical">

  <TextView
    android:id="@+id/text"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/hello_square" />
</LinearLayout>
```

Cependant, ils peuvent produire des problèmes d'affichage pour les montres au format ronds:



Google - CC BY

Il y a deux façons de résoudre ce problème: prévoir des layouts alternatifs ou utiliser un layout spécial gérant les formats d'écrans spéciaux.

Layouts alternatifs

Si l'on utilise Android Studio combiné à gradle pour le build, il faut ajouter le support des **Wearable UI** (ce qui est automatique avec le wizard de Studio):

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.google.android.support:wearable:+'
    compile 'com.google.android.gms:play-services-wearable:+'
}
```

Puis, on utilise comme *layout* **WatchViewStub** qui fera référence aux deux layouts alternatifs:

```
<android.support.wearable.view.WatchViewStub
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/watch_view_stub"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:rectLayout="@layout/rect_activity_wear"
    app:roundLayout="@layout/round_activity_wear">
</android.support.wearable.view.WatchViewStub>
```

L'intérêt de ce layout est que le code de la classe **WatchViewStub** détecte le type d'écran à l'exécution et choisira le bon layout en fonction.

Layout spécial

Il est aussi possible d'utiliser un *layout* spécial appelé **BoxInsetLayout** qui s'adapte à la forme de l'écran (principalement les cas des écrans carrés et ronds). Ce gabarit calcule le placement des objets graphiques en fonction d'un carré inscrit dans la forme de l'écran. C'est l'attribut **layout_box** qui donne les instructions de placement par rapport à ce carré. Par exemple, **layout_box="left|top"** place l'élément en haut à gauche du carré, ce qui n'aura pas d'effet si la montre est carrée (le carré inscrit dans un carré est un carré). Le *padding* permet d'éloigner l'élément des bords virtuels.

```
<android.support.wearable.view.BoxInsetLayout ... >
<FrameLayout
    android:padding="0dp"
    app:layout_box="left|top">

    <TextView ... />
```



Google - CC BY

CardFragment

Pour instancier une interface de type **Card** (philosophie **Suggest**), on programme dans l'activité principale de l'application **Wearable** l'instanciation d'un **CardFragment** que l'on place dans le layout de votre activité. Si le layout de votre activité est très simple et constituée d'un unique **FrameLayout** comme:



```
<android.support.wearable.view.BoxInsetLayout ...

<FrameLayout
    android:id="@+id/frame_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_box="bottom">

</FrameLayout>
</android.support.wearable.view.BoxInsetLayout>
```

On peut pousser une carte à l'aide du **FragmentManager**:

```
FragmentManager fragmentManager = getFragmentManager();
FragmentManager.beginTransaction().beginTransaction();
CardFragment cardFragment = CardFragment.create("Titre", "Mon message", R.drawable.dandelion);
fragmentTransaction.add(R.id.frame_layout, cardFragment);
fragmentTransaction.commit();
```

Il est aussi possible de définir complètement statiquement une carte dans le gabarit de l'activité:

```
<android.support.wearable.view.BoxInsetLayout ...

<android.support.wearable.view.CardScrollView
    android:id="@+id/card_scroll_view"
    android:layout_height="match_parent"
    android:layout_width="match_parent"
    app:layout_box="bottom">

    <android.support.wearable.view.CardFrame
        android:layout_height="wrap_content"
        android:layout_width="fill_parent">
```

```

<LinearLayout
    ...
</LinearLayout>
</android.support.wearable.view.CardFrame>
</android.support.wearable.view.CardScrollView>
    <LinearLayout
</android.support.wearable.view.BoxInsetLayout>

```

ListView

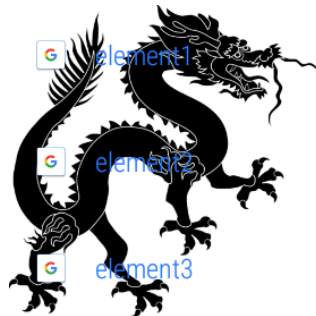
Comme pour un téléphone, l'activité d'un **Wearable** peut contenir une liste d'éléments graphiques. Les principes sont identiques:

- Définir dans le gabarit un **android.support.wearable.view.WearableListView**
- Créer un **Adapter** aux éléments à afficher
- Récupérer l'objet **WearableListView** et lui assigner l'adaptateur

```

setContentView(R.layout.maliste);
// Get the list component from the layout of the activity
WearableListView listView = (WearableListView) findViewById(R.id.maliste);
// Assign an adapter to the list
String[] array = new String[] {"element1", "element2", "element3"};
listView.setAdapter(new MonAdapteur(this, array));
// Set a click listener
listView.setOnClickListener(this);

```



Ne pas oublier la méthode **onCreateViewHolder** qui *inflate* chaque élément de la liste en utilisant un gabarit spécifique pour chaque item (layout/list_item.xml qui utilise une classe spécifique e.g. `jf.andro.weartest2.MyListView` héritant de `LinearLayout`).

```

// Create new views for list items
// (invoked by the WearableListView's layout manager)
@Override
public WearableListView.ViewHolder onCreateViewHolder(ViewGroup parent,
                                                    int viewType) {
    // Inflate our custom layout for list items
    return new ItemViewHolder(mInflater.inflate(R.layout.list_item, null));
}

```

qui *inflate* list_item.xml:

```

<jf.andro.weartest2.MyListView ...>
  <ImageView...
  <TextView...
</jf.andro.weartest2.MyListView>

```

qui correspond à **`jf.andro.weartest2.MyListView` extends `LinearLayout`**.

Le code de l'exemple complet se trouve à: <http://developer.android.com/training/wearables/ui/lists.html>

Applications à écrans multiples

On peut utiliser un layout particulier permettant de faire un damier d'écrans (cf. [Code-WearGridApp](#)). Il faut, un peu comme pour une liste, implémenter la méthode qui génère le fragment pour chaque case [i,j]. Voici une implémentation simple réalisant un damier 5x5:

```
public class MyFragmentGridPagerAdapter extends FragmentPagerAdapter {

    private final Context mContext;
    private List mRows;

    public MyFragmentGridPagerAdapter(Context ctx, FragmentManager fm) {
        super(fm);
        mContext = ctx;
    }

    @Override
    public Fragment getFragment(int i, int j) {
        CardFragment fragment = CardFragment.create("Page " + i + " " + j, "Page", R.drawable.common_google_signin);
        return fragment;
    }

    @Override
    public int getRowCount() {
        return 5; }
    @Override
    public int getColumnCount(int i) { return 5;
    }}
```

Il faut ensuite affecter cet *adapter* à votre layout graphique:

```
<android.support.wearable.view.BoxInsetLayout ...>
    <android.support.wearable.view.GridViewPager
        android:id="@+id/pager" ... />
</android.support.wearable.view.BoxInsetLayout>
```

```
GridViewPager pager = (GridViewPager) findViewById(R.id.pager);
pager.setAdapter(new MyFragmentGridPagerAdapter(this, getFragmentManager()));
```



Confirmations

Les **DelayedConfirmationView** permettent de faire une animation et d'avoir une callback en fonction de la réaction de l'utilisateur (cf. [Code-WearConfirmationDialog](#)). En pratique, il faut implémenter **DelayedConfirmationListener** en fonction du choix ou non de l'utilisateur:

```
public class MainActivity extends WearableActivity
    implements DelayedConfirmationView.DelayedConfirmationListener {

    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.confirmation_layout);
        DelayedConfirmationView mDelayedView =
            (DelayedConfirmationView) findViewById(R.id.delayed_confirm);
        mDelayedView.setListener(this);
        mDelayedView.setTotalTimeMs(5000);
        mDelayedView.start();

        public void onTimerFinished(View view) {
```



```
//finish(); // Ou préparer un fragment à afficher:
... fragmentTransaction.commit(); }
```



Le layout contient un **DelayedConfirmationView**:

```
<android.support.wearable.view.DelayedConfirmationView
    android:id="@+id/delayed_confirm"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:src="@drawable/tourne"
    app:circle_color="@color/blue"
    app:circle_radius="30dip"
    app:circle_radius_pressed="55dip"
    app:circle_border_width="4dip"
    app:circle_border_color="@color/blue"/>
</LinearLayout>
```

Autres éléments graphiques intéressants:

- **ConfirmationActivity**: An activity that displays confirmation animations after the user completes an action.
- **DismissOverlayView**: A view for implementing long-press-to-dismiss.

Ambient Mode

La dualité mode *ambient* et *interactif* impose de simplifier l'interface de votre application si elle est prévue pour rester toujours visible à l'écran. Dans ce cas, certains éléments du gabarit doivent disparaître. Pour ce faire, dans le **onCreate()** de l'application on appelle la méthode **setAmbientEnabled()**; afin d'être notifié en cas de changement de mode:

```
public void onEnterAmbient(Bundle ambientDetails) {
    updateDisplay(); }
public void onUpdateAmbient() {
    updateDisplay(); }
public void onExitAmbient() {
    updateDisplay(); }
private void updateDisplay() {
    if (isAmbient()) {
        mContainerView.setBackgroundColor(getResources().getColor(android.R.color.black));
        mTextView.setTextColor(getResources().getColor(android.R.color.white));
        mClockView.setTextColor(getResources().getColor(android.R.color.white));
        mClockView.setVisibility(View.VISIBLE);
        mClockView.setText(AMBIENT_DATE_FORMAT.format(new Date()));
        findViewById(R.id.button).setVisibility(View.GONE);
        findViewById(R.id.thetext).setVisibility(View.GONE); } else {
```



9.3 Faire communiquer Handheld et Wearable

La procédure est décrite ici: <http://stackoverflow.com/a/25506889> Il faut tout d'abord créer 1 émulateur de téléphone (Handheld) ayant les google API ou utiliser un vrai téléphone; y installer le **Android Wear Companion**:

```
adb install com.google.android.wearable.app-2.apk
```

Créer 1 émulateur de montre (Wear) et lancer les deux émulateurs et se rappeler de leur nom:

```
adb devices
List of devices attached
emulator-5554    device
emulator-5556    device
```

Il faut maintenant faire une redirection de ports sur le téléphone:

Entre 2 émulateurs:

```
telnet localhost 5556
redir add tcp:5601:5601
```

Avec 1 smartphone réel:

```
adb -s smartphone forward tcp:5601 tcp:5601
```

Android Wear doit pouvoir détecter la présence de l'émulateur et s'appairer.

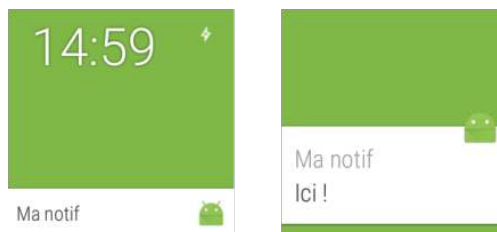
Notifications

Une fois appairés et si l'on demande au système de pousser les notifications vers Android Wear, celles-ci généreront des cartes avec le titre et le texte de la notification. Celles-ci sont construites à l'aide du **NotificationCompat.Builder** et le **NotificationManagerCompat**:

```
int notificationId = 001;

NotificationCompat.Builder notificationBuilder =
    new NotificationCompat.Builder(this)
        .setSmallIcon(R.drawable.common_google_signin_btn_icon_dark)
        .setContentTitle("Ma notif")
        .setContentText("Ici !");

NotificationManagerCompat notificationManager =
    NotificationManagerCompat.from(this);
notificationManager.notify(notificationId, notificationBuilder.build());
```



Action callback

Si l'on veut pouvoir faire une action retour vers le téléphone, par exemple ouvrir une application comme si on avait fait un click sur la notification du téléphone, il faut préparer un **PendingIntent**.

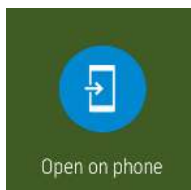
```
int notificationId = 001;
// Build intent for notification content
Intent viewIntent = new Intent("jff.andro.maCallBack");
PendingIntent viewPendingIntent =
    PendingIntent.getActivity(this, 0, viewIntent, 0);
```

```

NotificationCompat.Builder notificationBuilder =
    new NotificationCompat.Builder(this)
        .setSmallIcon(R.drawable.common_google_signin_btn_icon_dark)
        .setContentTitle("Ma notif")
        .setContentText("Ici !")
        .setContentIntent(viewPendingIntent);

NotificationManagerCompat notificationManager =
    NotificationManagerCompat.from(this);
notificationManager.notify(notificationId, notificationBuilder.build());

```



Boutons d'action

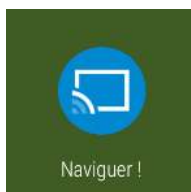
On peut ajouter un bouton d'action supplémentaire à l'action de click de la notification elle-même. Il suffit d'appeler la méthode `addAction(Drawable, String, PendingIntent)` sur l'objet `NotificationCompat.Builder`:

```

// Build an intent for an action to view a map
Intent mapIntent = new Intent(Intent.ACTION_VIEW);
Uri geoUri = Uri.parse("geo:0,0?q=" + Uri.encode("Bourges, France"));
mapIntent.setData(geoUri);
PendingIntent mapPendingIntent =
    PendingIntent.getActivity(this, 0, mapIntent, 0);

notificationBuilder.addAction(R.drawable.cast_ic_notification_2,
    "Naviguer !", mapPendingIntent);

```



Pour une action visible uniquement par le Wearable, utiliser la méthode `extend()` et la classe `WearableExtender`:

```
notificationBuilder.extend(new WearableExtender().addAction(action))
```

Envoyer un message au Wearable

Pour faire démarrer une application sur la montre à partir d'un événement du téléphone, il ne faut pas utiliser une notification. Il faut se baser sur un système de messages. On code côté montre un receveur de message et on envoie ce message côté téléphone. Il faut, avant d'envoyer le message se connecter à l'API Google. Les étapes de l'envoi sont les suivantes:

```

GoogleApiClient mApiClient = new GoogleApiClient.Builder(getApplicationContext())
    .addApi(Wearable.API)
    .build();

Log.i("JFL", "Connection...");
mApiClient.blockingConnect();
Log.i("JFL", "Connected !");

NodeApi.GetConnectedNodesResult nodes =

```

```

        Wearable.NodeApi.getConnectedNodes(mApiClient).await();
    for(Node node : nodes.getNodes()) {
        Log.i("JFL", "Node Found !");
        String msg = new String("Message");
        MessageApi.SendMessageResult result =
            Wearable.MessageApi.sendMessage(
                mApiClient, node.getId(), "path", msg.getBytes()).await();
    }
    Log.i("JFL", "Message sent !");

    mApiClient.disconnect();

```

Du côté de l'application, il faut filtrer la réception du message et démarrer l'activité:

```

public class MyWearableListenerService extends WearableListenerService {
    @Override
    public void onMessageReceived(MessageEvent messageEvent) {
        Log.i("JFL", "MESSAGE RECEIVED !");
        // Filtering message...
        Intent i = new Intent(getApplicationContext(), MainWearableActivity.class);
        i.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
        startActivity(i);
    }
}

```

```

<service android:name=".MyWearableListenerService" >
<intent-filter>
<action android:name="com.google.android.gms.wearable.BIND_LISTENER" />
</intent-filter>
</service>

```

Penser à ajouter au build.gradle la dépendance:

```

compile 'com.google.android.gms:play-services-wearable:+'

```

9.4 Watchfaces

Et on oublie qu'il est aussi indispensable de programmer l'écran qui donne l'heure...
<http://developer.android.com/training/wearables/watch-faces/index.html>



Remix of Moto 360 by David Pascual - CC BY <https://flic.kr/p/nSZyW6>

10 Google Cloud Messaging

- 10.1 GCM: Google Cloud Messaging** **86**
- 10.2 Etape 1: Developer Console** **86**
- 10.3 Etape 2: créer un serveur de démo** **88**
- 10.4 Etape 3: développer l'application** **88**
 - Architecture de l'application 88
 - Gestion du token 89
 - Rafraichissement du token 89
 - Réception des messages 90
 - Plugin google-services 90
 - Enregistrement auprès du service 92
 - Résultat 92
 - Demo 93

Ressources pour ce cours:

- Tutoriel de Dan Andrei: <https://www.digitalocean.com/community/tutorials/how-to-create-a-server-to-send-push-notifications-with-gcm-to-android-devices-using-python>
 - Demo de ce cours: <https://github.com/jflalande/GCMReceiver>
 - Faire un client Android au service GCM: <https://developers.google.com/cloud-messaging/android/client>
- Classe de la démo de référence: <https://github.com/googlesamples/google-services/tree/master/android/gcm/app/src/main/java/gcm/play/android/samples/com/gcmquickstart>
 - Manifest de la démo de référence: <https://github.com/googlesamples/google-services/blob/master/android/gcm/app/src/main/AndroidManifest.xml>

10.1 GCM: Google Cloud Messaging

Google Cloud Messaging est un des nombreux services fournis par les Google API. Ce service, très simple à comprendre mais pas facile à mettre en oeuvre, permet d'envoyer des messages depuis un serveur, vers un ensemble de téléphones, au travers du Cloud de Google. La solution fournit la scalabilité, au détriment de la simplicité. Cependant, la documentation est abondante sur le sujet.

Le cas traité ici est le plus simple possible: on veut envoyer un message depuis un serveur web vers des téléphones (cas *Downstream message*). D'autres services sont possibles (*Upstream messages*, groupes de messages, etc.).

Le protocole sous-jacent utilisé est HTTP ou XMPP. Les requêtes sont simples:

```

https://gcm-http.googleapis.com/gcm/send
Content-Type:application/json
Authorization:key=AiZaSyAU3.....ij90vT50Tg4KrI

{
  "data": {
    "score": "5x1",
    "time": "15:10"
  },
  "to" : "bk3RNwTe3H0:CI2k_HHwgIpoDKCIZvvDMExUdFQ3P1..."
}

```

10.2 Etape 1: Developer Console

Il faut créer un projet dans la console développeur <https://console.developers.google.com>:





puis activer le service "Google Cloud Messaging":



Google Cloud Messaging

Google Cloud Messaging allows for push messaging to Android, iOS and Chrome users. [En savoir plus](#)

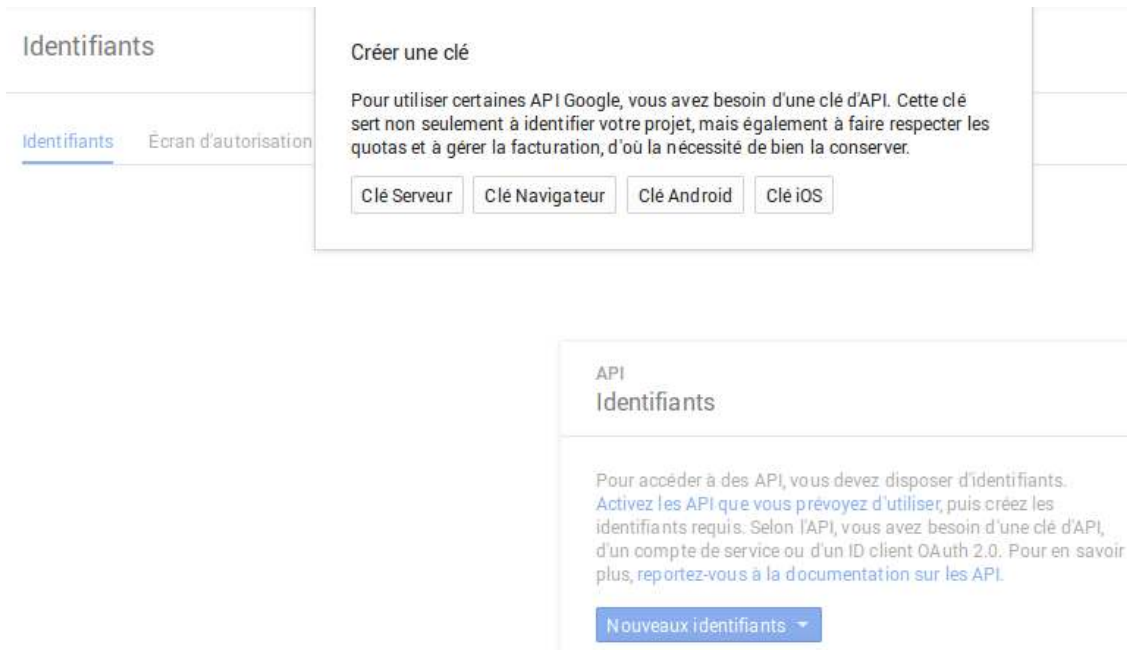
Utilisation d'identifiants avec cette API

Utilisation d'une clé d'API

Pour utiliser cette API, vous avez besoin d'une clé d'API. Celle-ci identifie votre projet pour vérifier les quotas et les accès. Pour obtenir une clé d'API, accédez à la page 'Identifiants'. Vous avez besoin d'une clé pour chaque plate-forme (Web, Android, iOS, etc.). [En savoir plus](#)

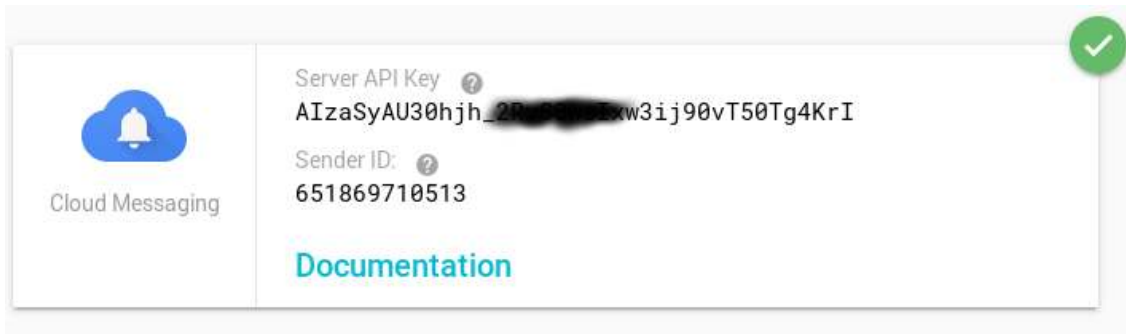


Dans l'onglet "Identifiants", on peut générer un identifiant qui autorisera l'accès à l'API depuis l'application du serveur web: cette clef est une "Clé Serveur":



On obtient alors 2 choses:

- la clef d'accès à l'API qui va permettre l'application Server d'accéder au service Google (API key).
- un identifiant unique qui sera l'identité de notre service d'envoi de message, d'un point de vue des utilisateurs du service (Sender ID).



10.3 Etape 2: créer un serveur de démo

Notre serveur de démo sera ici très rudimentaire, écrit en python, il fait quelques lignes:

```
from gcm import *

gcm = GCM("AIzaSyAU.....0vT50Tg4KrI")
data = {'the_message': 'Super, ca marche !', 'param2': 'value2'}
reg_id = 'TOKEN TEMPORAIRE: à renseigner plus tard'
gcm.plaintext_request(registration_id=reg_id, data=data)
```

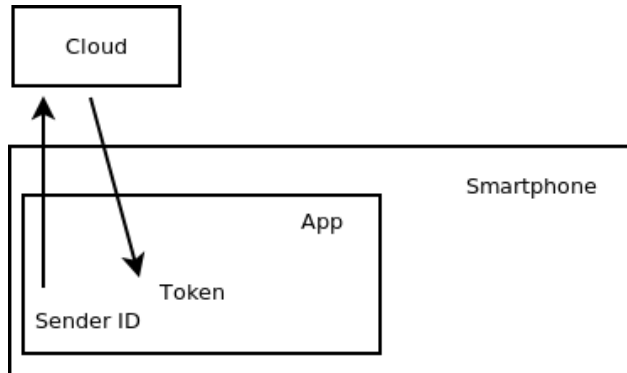
pour qu'il marche, il faut installer un module python:

```
sudo easy_install install python-gcm
# ou bien:
sudo pip install python-gcm
```

10.4 Etape 3: développer l'application

Pour cette étape, on pourra s'inspirer le projet de référence située à: <https://github.com/ptitmain/GCMReceiver> qui lui même s'inspire de <https://github.com/googlesamples/google-services/tree/master/android/gcm>.

Pour fonctionner, l'application cliente va se connecter avec l'identifiant du fournisseur de service (Sender ID) et un dérivé de l'API key. A partir de cela, le Cloud va lui renvoyer un token temporaire d'accès au service. Ce token temporaire sera communiqué au serveur applicatif pour signifier qu'un client vient de s'enregistrer.

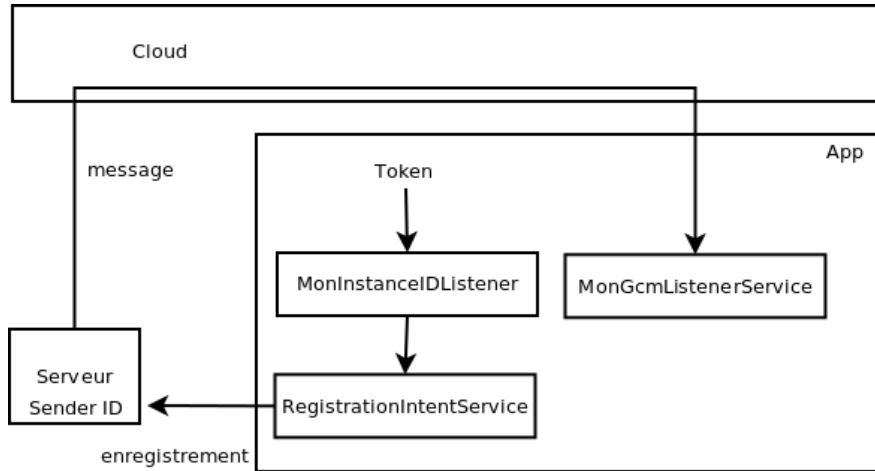


Architecture de l'application

L'application minimum est constituée de:

- **MonGcmListenerService** extends **GcmListenerService**: le service qui va être notifié du message
- **MyInstanceIdListenerService** extends **InstanceIdListenerService**: le service qui va être notifié d'un nouveau token utilisateur de service

- **RegistrationIntentService** extends **IntentService**: le service qui va gérer ce token d'utilisateur de service et l'envoyer au serveur de l'application.
- **MainActivity** qui va gérer l'inscription aux Google APIs.



Gestion du token

Le Manifest contient la déclaration de deux services qui vont s'occuper de la gestion du token.

```

<service
    android:name="jf.andro.gcmreceiver.MyInstanceIDListenerService"
    android:exported="false">
    <intent-filter>
        <action android:name="com.google.android.gms.iid.InstanceID"/>
    </intent-filter>
</service>

<service
    android:name="jf.andro.gcmreceiver.RegistrationIntentService"
    android:exported="false">
</service>
    
```

Rafraichissement du token

La classe **MyInstanceIDListenerService** filtre les intents **com.google.android.gms.iid.InstanceID**. Si l'on est notifié d'un rafraichissement de token temporaire, on envoie un intent informatif à **RegistrationIntentService**:

```

public class MyInstanceIDListenerService extends InstanceIDListenerService {

    private static final String TAG = "MyInstanceIDLS";

    /**
     * Called if InstanceID token is updated. This may occur if the security of
     * the previous token had been compromised. This call is initiated by the
     * InstanceID provider.
     */
    // [START refresh_token]
    @Override
    public void onTokenRefresh() {
        // Fetch updated Instance ID token and notify our app's server of any changes (if applicable).
        Intent intent = new Intent(this, RegistrationIntentService.class);
        startService(intent);
    }
    // [END refresh_token]
}
    
```

C'est alors dans cette classe que l'on gère l'envoi du token temporaire vers nos serveurs applicatifs, pour connaître ce nouveau client:

```
public class RegistrationIntentService extends IntentService {

    protected void onHandleIntent(Intent intent) {
        // [START register_for_gcm]
        InstanceID instanceID = InstanceID.getInstance(this);
        String token = instanceID.getToken(getString(R.string.gcm_defaultSenderId),
            GoogleCloudMessaging.INSTANCE_ID_SCOPE, null);
        // [END get_token]
        Log.i(TAG, "GCM Registration Token: " + token);

        // TODO: Implement this method to send any registration to your app's servers.
        sendRegistrationToServer(token);
    }
}
```

Réception des messages

Le Manifest contient donc les déclaration d'un receiver **GcmReceiver** qui sera contacté par les Google APIs lorsqu'un message arrive. Il est en charge de délivrer le message à **MonGcmListenerService**, sous-classe de **GcmListenerService**:

```
<receiver
    android:name="com.google.android.gms.gcm.GcmReceiver"
    android:permission="com.google.android.c2dm.permission.SEND" >
    <intent-filter>
        <action android:name="com.google.android.c2dm.intent.RECEIVE" />
        <!-- for Gingerbread GSF backward compat -->
        <action android:name="com.google.android.c2dm.intent.REGISTRATION" />
        <category android:name="jf.andro.gcmreceiver" />
    </intent-filter>
</receiver>

<service
    android:name="jf.andro.gcmreceiver.MonGcmListenerService"
    android:exported="false" >
    <intent-filter>
        <action android:name="com.google.android.c2dm.intent.RECEIVE" />
    </intent-filter>
</service>
```

Rien de bien sorcier dans la classe **MonGcmListenerService**:

```
public class MonGcmListenerService extends GcmListenerService {
    @Override
    public void onMessageReceived(String from, Bundle data) {

        Log.i("JFL", "Message received from " + from);
        Log.i("JFL", "Message: " + data.getString("the_message"));

        // Todo: traitement du message
    }
}
```

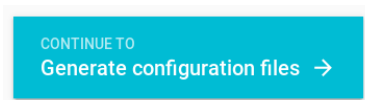
Plugin google-services

Il reste à résoudre encore quelques petits choses concernant la configuration du client aux Google Services. En effet, l'appel:

```
String token = instanceID.getToken(getString(R.string.gcm_defaultSenderId),
    GoogleCloudMessaging.INSTANCE_ID_SCOPE, null);
```

ne fonctionne pas encore, car Android Studio ne reconnaît pas `R.string.gcm_defaultSenderId`. Ce paramètre de la classe `R` est à générer à l'aide d'un plugin gradle et d'un fichier de configuration JSON généré par Google :

Puis après :



on obtient le fichier de configuration :

Qui ressemble à :

```
{
  "project_info": {
    "project_id": "cours-android-jf1",
    "project_number": "651869710513",
    "name": "Cours Android"
  },
  "client": [
    {
      "client_info": {
        "mobilesdk_app_id": "1:651869710513:android:be293f8ca6084314",
        "client_id": "android:jf.andro.gcmreceiver",
        "client_type": 1,
        "android_client_info": {
          "package_name": "jf.andro.gcmreceiver"
        }
      }
    }
  ],
  "oauth_client": [],
  "api_key": [],
}
```

```

    ....
    ,
    "client_info": [],
    "ARTIFACT_VERSION": "1"
  }

```

Ce fichier de configuration, combiné à un plugin gradle, va permettre de modifier la compilation du projet, et notamment la classe **R**. Pour cela il faut placer le fichier JSON envoyé par Google dans le sous répertoire **app** de votre projet Android Studio, puis modifier les deux **build.gradle**:

Pour le **build.gradle** du projet, ajouter:

```

dependencies {
    classpath 'com.android.tools.build:gradle:2.0.0-alpha9'
    classpath 'com.google.gms:google-services:2.0.0-alpha9'
}

```

Pour le **build.gradle** de l'application, ajouter:

```

dependencies {
    compile 'com.android.support:appcompat-v7:23.1.1'
    compile "com.google.android.gms:play-services:8.4.0"
}

apply plugin: 'com.google.gms.google-services'

```

Toutes ces étapes fastidieuses sont bien sûr documentées à: <https://developers.google.com/cloud-messaging/android/client>

Enregistrement auprès du service

Dernière étape: l'application doit, après avoir vérifié que les Google API sont disponibles, démarrer l'enregistrement auprès du service en lançant le service **RegistrationIntentService**:

```

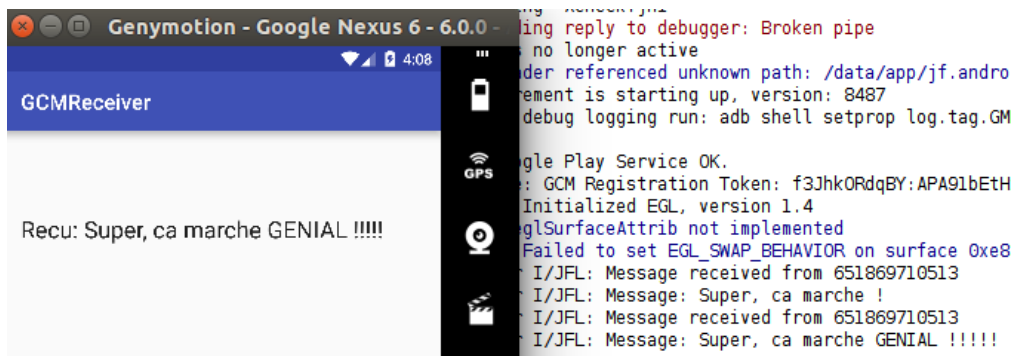
// Dans le onCreate() de votre application:
if (checkPlayServices()) {
    // Start IntentService to register this application with GCM.
    Intent intent = new Intent(this, RegistrationIntentService.class);
    startService(intent);
}

// Ailleurs dans votre classe:
private boolean checkPlayServices() {
    GoogleApiAvailability apiAvailability = GoogleApiAvailability.getInstance();
    int resultCode = apiAvailability.isGooglePlayServicesAvailable(this);
    if (resultCode != ConnectionResult.SUCCESS) {
        if (apiAvailability.isUserResolvableError(resultCode)) {
            apiAvailability.getErrorDialog(this, resultCode, PLAY_SERVICES_RESOLUTION_REQUEST).show();
            Log.i("JFL", "Connected !");
        } else {
            Log.i("JFL", "This device is not supported.");
            finish();
        }
        return false;
    } else { Log.i("JFL", "Succes: Google Play Service OK."); }
    return true;
}

```

Résultat





Bien entendu, je n'explique pas la partie qui consiste à rafraichir l'interface graphique de l'application avec le contenu du message. De plus, il faut remarquer que le token temporaire est "en dur" dans le script python puisque la partie qui consiste à envoyer le token vers le serveur applicatif n'a pas été codée.

Demo

[Video](#)

11 Divers

11.1 Bibliothèques natives: JNI	94
Classe d'interface	94
Génération du .h	94
Écriture du .c et compilation	95
Démonstration	95

11.1 Bibliothèques natives: JNI

Dans [JNI], un excellent tutoriel présente les grandes étapes pour programmer des appels natifs vers du code C, ce qui, en java s'appelle **JNI** pour *Java Native Interface*. Le but recherché est multiple:

- les appels natifs permettent de s'appuyer sur du code C déjà développé et évite de tout recoder en java
- le code ainsi déporté est davantage protégé contre la décompilation des .dex
- le code C peut éventuellement être utilisé ailleurs, par exemple sous iOS

Evidemment, ce type de développement est tout à fait orthogonal au but d'Android, c'est à dire à la programmation Java s'appuyant sur les fonctionnalités offertes par l'API Android. Dans votre code C, vous n'aurez accès à quasiment rien, ce qui restreint grandement l'intérêt de programmer des méthodes natives. Cependant, cette possibilité est très appréciée pour le développement des jeux car cela permet un accès facile aux primitives graphiques OpenGL.

Pour travailler avec des appels **JNI**, il faudra utiliser un outil complémentaire de Google permettant de compiler votre programme C vers une bibliothèque partagée .so: le [Android NDK](#).

Classe d'interface

La première étape consiste à réaliser le code Java permettant de faire le pont entre votre activité et les méthodes natives. Dans l'exemple suivant, on déclare une méthode statique (en effet, votre programme C ne sera pas un objet), appelant une méthode native dont l'implémentation réalise l'addition de deux nombres:

```
package andro.jf.jni;

public class NativeCodeInterface {

    public static native int calcul1(int x, int y);

    public static int add(int x, int y)
    {
        int somme;
        somme = calcul1(x,y);
        return somme;
    }

    static {
        System.loadLibrary("testmodule");
    }
}
```

Un appel natif depuis une activité ressemblera à:

```
TextView text = (TextView) findViewById(R.id.texte);
text.setText("5+7 = " + NativeCodeInterface.add(5, 7));
```

Génération du .h

A partir de la classe précédemment écrite, il faut utiliser l'outil **jvarkit** pour générer le fichier .h correspondant aux méthodes natives déclarées dans le .java. En dehors du répertoire **src/** de votre projet, vous pouvez créer un répertoire **jni/** afin d'y placer les fichiers de travail de la partie C. On réalise donc la compilation, puis l'extraction du .h:

```
javac javac -d ./jni ./src/andro/jf/jni/NativeCodeInterface.java
cd ./jni
javah -jni andro.jf.jni.NativeCodeInterface
```

On obtient alors le fichier `andro_jf_jni_NativeCodeInterface.h` suivant:

```
// Header for class andro_jf_jni_NativeCodeInterface

#ifndef __Included_andro_jf_jni_NativeCodeInterface
#define __Included_andro_jf_jni_NativeCodeInterface
#ifdef __cplusplus
extern "C" {
#endif
JNIEXPORT jint JNICALL Java_andro_jf_jni_NativeCodeInterface_calcul1
(JNIEnv *, jclass, jint, jint);

#ifdef __cplusplus
}
#endif
#endif
```

Écriture du `.c` et compilation

A partir du fichier `.h`, il faut maintenant écrire le code de l'appel natif:

```
#include "andro_jf_jni_NativeCodeInterface.h"
JNIEXPORT jint JNICALL Java_andro_jf_jni_NativeCodeInterface_calcul1
(JNIEnv * je, jclass jc, jint a, jint b) {
    return a+b;
}
```

Il faut ensuite créer le Makefile approprié pour la compilation du fichier `.c` à l'aide du script `ndk-build`. Le fichier de Makefile doit s'appeler `Android.mk`:

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)

LOCAL_MODULE     := testmodule
LOCAL_CFLAGS     := -Werror
LOCAL_SRC_FILES  := test.c
LOCAL_LDLIBS     := -llog

include $(BUILD_SHARED_LIBRARY)
```

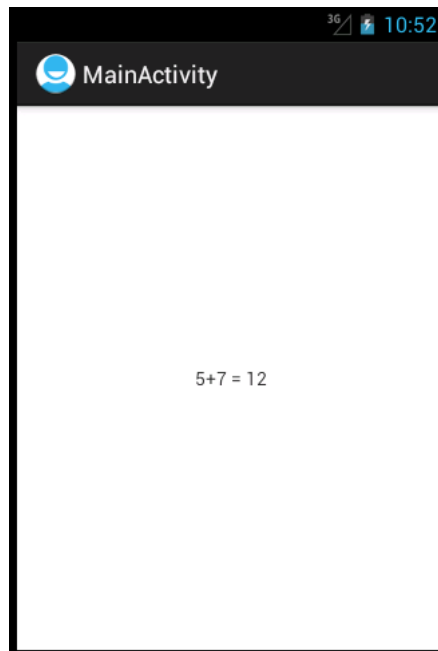
Et la phase de compilation doit ressembler à:

```
/usr/local/android-ndk-r8b/ndk-build
Compile thumb  : testmodule <= test.c
SharedLibrary  : libtestmodule.so
Install        : libtestmodule.so => libs/armeabi/libtestmodule.so
```

Démonstration

L'annexe [Code-JNI](#) présente le code complet d'un appel natif à une librairie C réalisant l'addition de deux nombres. Le résultat obtenu est le suivant:





12 Annexes: outils

12.1 Outils à télécharger	97
12.2 L'émulateur Android	97
AVD: le gestionnaire d'appareils	97
Accélération matérielle pour l'émulateur	98
Lancer l'émulateur	98
12.3 ADB: Android Debug Bridge	98
Debugging	99
Tester sur son téléphone	99
12.4 Simuler des sensors	99
Adaptation de votre application au simulateur	100
12.5 HierarchyViewer	100

12.1 Outils à télécharger

Les outils minimum:

- [JDK 6 ou 7](#)
- [Eclipse](#)
- ADT plugin, via l'*update manager* avec l'url <https://dl-ssl.google.com/android/eclipse/>
- [Android SDK](#)
- ou bien directement l'[ADT bundle](#) qui regroupe les 3 précédents.

Les outils complémentaires:

- [Android apktool](#)
- [Dex2Jar](#)
- [JD-GUI](#)
- [Smali](#)

12.2 L'émulateur Android

L'émulateur Android résulte d'une compilation des sources d'Android permettant d'exécuter le système sur un PC classique. En fonction de ce qui a été installé par le SDK, on peut lancer un émulateur compatible avec telle ou telle version de l'API.

L'émulateur reste limité sur certains aspects. Par exemple, il est difficile de tester une application utilisant l'accéléromètre ou le wifi. Il devient alors nécessaire de réaliser des tests avec un téléphone réel.

L'émulateur peut maintenant s'exécuter directement de manière native, car il est possible de le compiler pour une architecture x86. Dans les anciennes versions de l'émulateur, seuls les processeurs ARM étaient supportés car ce sont eux qui équipent la plupart de nos smartphones. Dans ce cas, l'émulateur utilise la virtualisation pour s'exécuter sur du x86.

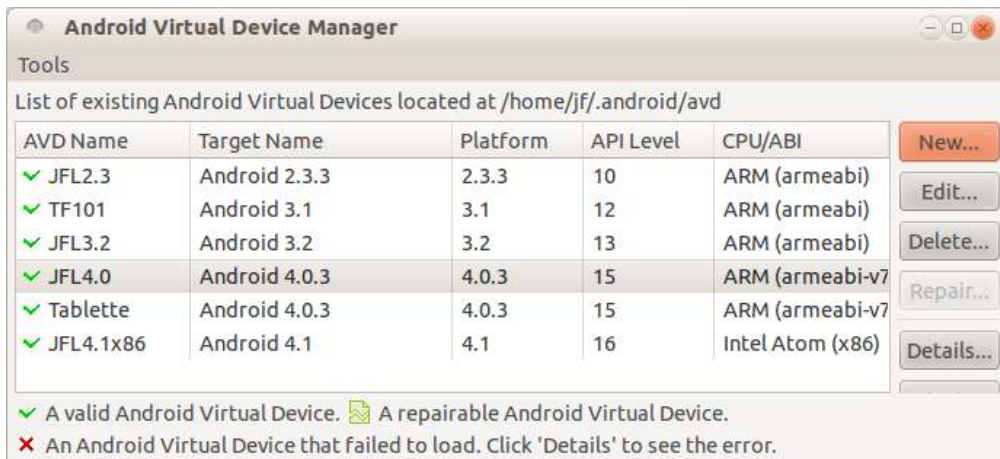
Une version particulière de l'émulateur, nommée "Google APIs", permet de tester une application qui interroge l'un des nombreux services Google (non open-source), par exemple le service Google Map. L'émulateur "Google APIs" n'est disponible qu'en version ARM.

AVD: le gestionnaire d'appareils

[[Emulator](#)] Pour tester une application, Google fournit un émulateur Android, abrégé par AVD pour *Android Virtual Device*. Il est possible, en fonction de ce qui est présent dans le SDK installé, de créer plusieurs configurations différentes de téléphones:

- l'outil `tools/android` permet de mettre à jour et installer différentes version du SDK
- l'outil `tools/android avd` (ajouter "avd" en ligne de commande) liste les différentes configurations d'appareils de l'utilisateur





Accélération matérielle pour l'émulateur

Depuis Android 4.0 (ICS), Google fournit une image **kvm** de l'émulateur, compatible avec les plate-forme x86. Cette image permet de se passer de qemu qui permettait d'émuler un processeur ARM sur une architecture x86: on gagne l'*overhead* d'émulation !

Pour se faire, il faut bien entendu disposer de **kvm** et installer depuis le SDK manager l'outil *Intel x86 Atom System Image*, disponible dans ICS. Pour utiliser l'accélération, il faut créer un nouvel appareil utilisant un CPU de type *Intel Atom (x86)*, comme montré ici:



Lancer l'émulateur

[Emulator] Dans le répertoire **tools**, l'exécutable **emulator** permet de lancer l'émulateur Android en précisant la configuration d'appareil à l'aide de l'option **-avd**. La syntaxe de la ligne de commande est donc:

```
emulator -avd <avd_name> [-<option> [<value>]]
```

Par défaut, la commande sans option démarre l'émulateur.

L'émulateur peut aussi être paramétré en ligne de commande. Les options possibles permettent par exemple de:

- spécifier un serveur DNS (-dns-server <IP>)
- ralentir le CPU (-cpu-delay <value>)
- charger une partition de SDcard (-ramdisk <filepath>)
- effacer des données utilisateur (-wipe-data, -initdata)
- ...

12.3 ADB: Android Debug Bridge

L'outil **adb** (Android Debug Bridge) permet d'interagir avec l'émulateur pour effectuer des opérations de configuration, d'installation et de débogging. Par défaut, **adb** se connecte à la seule instance de l'émulateur qui est lancée (comportement de l'option **-e**) pour exécuter les commandes demandées.

Voici quelques commandes utiles:

Installation/Désinstallation d'une application

```
./adb install newDialerOne.apk
./adb uninstall kz.mek.DialerOne
```

Pour connaître le nom de *package* d'une application dont on est pas le développeur, on peut utiliser la commande `./aapt dump badging app.apk`.

Envoi/Récupération d'un fichier

```
adb push file /sdcard/
adb pull /sdcard/file ./
```

Exécution d'un shell dans le téléphone

```
adb shell
```

Debugging

Les commandes classiques **System.out.xxx** sont redirigées vers **/dev/null**. Il faut donc déboguer autrement. Pour ce faire, le SDK fournit un moyen de visualiser les logs du système et de les filtrer au travers de l'outil **adb** situé dans le répertoire **platform-tools** du SDK. L'outil permet de se connecter à l'émulateur et d'obtenir les messages de logs. On obtient par exemple pour une exception:

```
./adb logcat
I/System.out( 483): debugger has settled (1441)
W/System.err( 483): java.io.FileNotFoundException: /test.xml
                    (No such file or directory)
...
W/System.err( 483): at andro.jfl.AndroJFLActivity.onCreate(
                    Andro7x24Activity.java:38)
```

Cependant, il est beaucoup plus propre d'utiliser la classe **Log** qui permet de classer les logs par niveaux et par tags. Le niveau dépend des méthodes statiques utilisées, à savoir: **.v** (verbose), **.d** (debug), **.e** (error), **.w** (warning), etc. Par exemple:

```
Log.w("monTAG", "Mon message à logger");
```

On peut alors filtrer ce log, en posant un filtre avec **adb**:

```
adb logcat TODO
```

Tester sur son téléphone

Il est extrêmement simple d'utiliser un appareil réel pour tester une application. Il faut tout d'abord activer le débogage Android afin qu'*adb* puisse se connecter à l'appareil lorsque le câble USB est branché. On voit alors apparaître l'appareil, par exemple:

```
./adb devices
List of devices attached
363234B----- device
```

Dans Eclipse, il suffit de changer la configuration du "Run" pour "Active Devices", ce qui lancera automatiquement l'application sur le téléphone si aucun émulateur n'est lancé.

Les tests sur un appareil réel sont particulièrement utiles lorsque l'on travaille avec les capteurs.

12.4 Simuler des sensors

[SS] (cf [Code-Capteurs](#)) Afin de simuler les sensors, vous pouvez utiliser l'outil [Sensor Simulator](#) qui permet de simuler le changement de valeurs des senseurs à partir d'un client Java en dehors de l'émulateur. Pour réaliser cela il faut:

- Téléchargez [Sensor Simulator](#)
- Lancez l'outil `bin/sensorsimulator-x.x.x.jar` (`java -jar bin/sensorsimulator-x.x.x.jar`)

- Ajoutez `bin/sensorsimulator-lib-x.x.x.jar` à votre projet Eclipse
- Ajoutez `bin/SensorSimulatorSettings-x.x.x.apk` à votre device virtuel (utiliser la commande `adb install SensorSimulatorSettings-x.x.x.apk`)

Vous pouvez, pour commencer, tester la connexion entre l'outil externe et l'outil de configuration interne de Sensor Simulator. Dans l'émulateur, lancez Sensor Simulator et après avoir vérifié la concordance entre les adresses IP, vous pouvez connecter les deux outils depuis l'onglet Testing. Vous pouvez ensuite faire bouger le device virtuel dans le client java (téléphone en haut à gauche).

[Video](#)

Adaptation de votre application au simulateur

Le principe du simulateur est de remplacer la déclaration de votre `SensorManager` par celui du simulateur dans le code de votre application. Ainsi, les appels à la lecture des senseurs appellera la librairie du simulateur qui, à travers une connexion réseau, ira lire les données dans le client externe Java. Remplacez votre déclaration ainsi:

```
// SensorManager manager = (SensorManager) getSystemService(SENSOR_SERVICE);
SensorManagerSimulator manager =
    SensorManagerSimulator.getSystemService(this, SENSOR_SERVICE);
manager.connectSimulator();
```

Attention aux directives import qui, par défaut, importent les classes de `android.hardware` et doivent désormais utiliser `org.openintents.sensorsimulator.hardware`. Ne pas oublier non plus d'ajouter la permission permettant à votre application d'accéder à internet. Quelques adaptations du code peuvent être nécessaires...

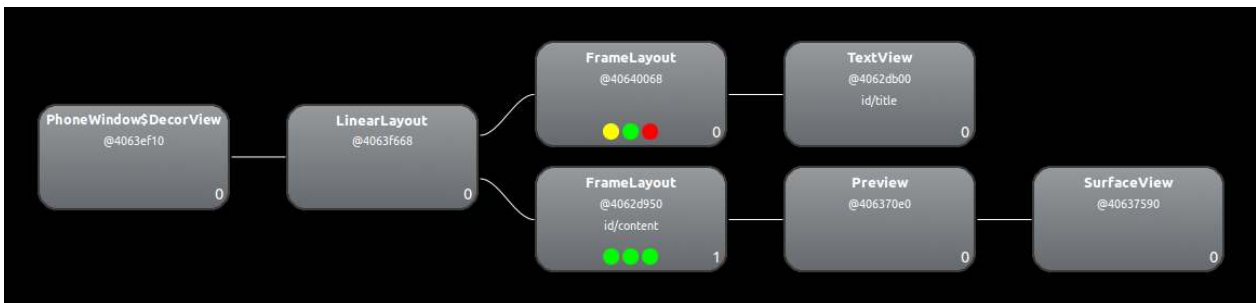
[Video](#)

12.5 HierarchyViewer

Android fournit un outil d'analyse *live* de la hiérarchie des objets graphiques. Ainsi, il est possible de mieux comprendre sa propre application lors de la phase de debugging. Pour l'utiliser, il faut lancer: `./hierarchyviewer`.

Dans l'exemple suivant, on voit deux `FrameLayout` qui sont enfants de l'éléments de plus haut niveau `LinearLayout`. Le premier `FrameLayout` a été décroché de l'élément `root` et remplacé par un nouveau `FrameLayout` contenant l'élément `Preview` permettant de visualiser ce que voit la caméra arrière (cf [Code-Camera](#)).

Les trois points de couleur représentent la vitesse de rendu. De gauche à droite on a le temps utilisé pour mesurer (calculer les dimensions), positionner (calculer la position des enfants) et dessiner. L'interprétation est: vert: 50% plus rapide que les autres éléments graphiques, jaune: moins de 50%, rouge: le plus lent de la hiérarchie.



PA	Programmation Android, de la conception au déploiement avec le SDK Google Android , Damien Guignard, Julien Chable, Emmanuel Robles, Eyrolles, 2009.
DA	Decompiling Android , Godfrey Nolan, Apress, 2012.
Cookbook	Android Cookbook , Ian F. Darwin, O'Reilly Media, decembre 2011.
AAS	Android Apps Security , Sheran Gunasekera, Apress, septembre 2012.
ASI	Android Security Internals: An In-Depth Guide to Android's Security Architecture < http://www.nostarch.com/androidsecurity >, Nikolay Elenkov, 2014.

Dalvik	Article Wikipedia , from Wikipedia, the free encyclopedia.
E-mail	Email sending in Android , Oleg Mazurashu, The Developer's Info, October 22, 2009.
SO	Android permissions: Phone Calls: read phone state and identity , stackoverflow.
SD	Utiliser les services sous Android , Nicolas Druet, Developpez.com, Février 2010.
API-Service	Service API changes starting with Android 2.0 , Dianne Hackborn, 11 Février 2010.
XML	Working with XML on Android , Michael Galpin, IBM, 23 Jun 2009.
VL	How to Position Views Properly in Layouts , jwei512, Think Android, Janvier 2010.
BAS	Basics of Android : Part III – Android Services , Android Competency Center, Janvier 2009.
MARA	Implementing Remote Interface Using AIDL , Marko Gargenta, novembre 2009.
AT	Multithreading For Performance , Gilles Debunne, 19 juillet 2010.
HWM	MapView Tutorial , 20 aout 2012.
JSONREST	How-to: Android as a RESTful Client , Cansin, 11 janvier 2009.
JNI	Tutorial: Android JNI , Edwards Research Group, CC-BY-SA, Avril 2012.
BAESystems	The Butterfly Effect of a Boundary Check , Sergei Shevchenko, 22 janvier 2012.
SELinux	Security Enhanced (SE) Android: Bringing Flexible MAC to Android , S. Smalley and R. Craig, in 20th Annual Network & Distributed System Security Symposium, San Diego, California, USA, 2013.
SECUSTOM	Android* Security Customization with SEAndroid , Liang Zhang, Intel, March 2014.
II	Les IntentService , Florian FournierProfil Pro, Developpez.com, Décembre 2014.
LLB	Life Before Loaders , Alex Lockwood, Juillet 2012.
ON	Tab Layout , Android developers.
PT	Processes and Threads , Android developers.
DBSQL	Data storage: Using Databases , Android developers, Novembre 2010.
SS	Sensor Simulator .
Emulator	Android Emulator .
ATK	AsyncTask .
JQM	Getting Started with jQuery Mobile .
UF	<uses-feature> .
CAM	Controlling the Camera .
SIGN	Signing Your Applications .
Storage	Storage Options .
Signing	Signing your applications .
LSAS	[android-developers] Launching a Service at the startup , Archives googlegroups, R Ravichandran and Dianne Hackborn, Juillet 2009.
KEK	Comment on fait un jeu pour smartphone ? , Kek, 2011.
ZZJN12	Detecting repackaged smartphone applications in third-party android market- places , W. Zhou, Y. Zhou, X. Jiang, and P. Ning, in Second ACM conference on Data and Application Security and Privacy, E. Bertino and R. S. Sandhu, Eds. San Antonio, TX, USA: ACM Press, Feb. 2012, pp. 317–326.
QC13	Mobile Security: A Look Ahead , Q. Li and G. Clark, Security & Privacy, IEEE, vol. 11, no. 1, pp. 78–81, 2013.
Faruki15	Faruki, P., Bharmal, A., Laxmi, V., Ganmoor, V., Gaur, M. S., Conti, M., & Rajarajan, M. (2015). Android Security: A Survey of Issues, Malware Penetration and Defenses. IEEE Communications Surveys & Tutorials, PP(99), 1-27.