Développement Android

J.-F. Couchot

7 février 2012

Table des matières

1 Introduction à l'OS embarqué Android **3**
- I Composants principaux d'une application . . . 3
  - I.1 Une activité . . . 3
  - I.2 Un service . . . 3
  - I.3 Une Intention . . . 3
  - I.4 Un fournisseur de contenus . . . 3
- II Organisation d'un projet Android . . . 3
  - II.1 Arborescence . . . 3
  - II.2 Travaux Pratiques : Hello World . . . 4
  - II.3 Exercice . . . 4

2 Les interfaces utilisateur **5**
- I Rendu visuel . . . 5
- II Interface : programmation Java ou déclaration XML . . . 5
  - II.1 Programme Java . . . 5
  - II.2 Déclaration XML . . . 5
- III Les principaux composants graphiques . . . 6
  - III.1 Exercice . . . 6
- IV Les Layouts . . . 7
  - IV.1 FrameLayout . . . 7
  - IV.2 LinearLayout . . . 7
  - IV.3 TableLayout . . . 7
  - IV.4 RelativeLayout . . . 7
  - IV.5 Travaux Pratiques . . . 7
- V Les menus des activités . . . 7
  - V.1 Définition statique des menus . . . 7
  - V.2 Exercices . . . 8
  - V.3 Gonflage du menus défini en XML . . . 8
  - V.4 Gestion des événements liés aux menu . . . 8
  - V.5 Exercice . . . 8

3 Cycle de vie d'une activité **9**
- I Méthodes du cylce de vie . . . 9
- II Exercice . . . 9

4 Sauvegarder des données **11**
- I Les préférences . . . 11
  - I.1 Méthodes . . . 11
  - I.2 Exercices . . . 11
  - I.3 Travaux Pratiques . . . 11
- II Les fichiers . . . 12
- III Le cloud . . . 12
  - III.1 Hériter de la classe `BackupAgentHelper` . . . 12
  - III.2 Modifier le manifest pour toute l'application . . . 13
  - III.3 La demande de synchronisation . . . 13
  - III.4 Évaluer le développement . . . 13

III.5 Travaux pratiques 13

5 **Récupération périodique de données grâce à un service** **14**
I Organisation générale 14
II L’application web 14
II.1 La basse de données 14
II.2 Les pages PHP 14
III L’application Android 16
III.1 Vue organisée à l’aide d’onglets 16
III.2 L’envoi de messages 17
III.3 Le service de réception de messages 17
III.4 L’affichage des messages 18

Chapitre 1

Introduction à l'OS embarqué Android

I Composants principaux d'une application

Une application Android est construite à partir de quatre catégories de composants : les activités, les services, les intentions et les fournisseurs de contenu.

I.1 Une activité

C'est la brique de base. Par défaut une application est une activité. C'est un morceau de code qui s'exécute à la demande de l'utilisateur, de l'OS,... et qui peut être tué par l'utilisateur ou l'OS pour préserver la mémoire du téléphone. Une activité peut interagir

- avec l'utilisateur en lui demandant des données
- avec d'autres activités ou services en émettant des intentions ou fournissant du contenu (voir ci dessous)

I.2 Un service

C'est l'analogue des services ou démons qui s'exécutent sur un OS classique. C'est un morceau de code qui s'exécute habituellement en arrière-plan entre le moment où il est lancé jusqu'à l'arrêt du mobile.

I.3 Une Intention

Une intention (intent en anglais) est un message contenant des données émis par Android (l'OS, une autre application, un autre service) pour prévenir les applications s'exécutant de la survenue d'un événement : déplacement du GPS, réception d'un SMS...

I.4 Un fournisseur de contenus

Une application Android répond normalement à un besoin et fournit donc certaines fonctionnalités. On appelle cela un fournisseur de contenus. Une application doit normalement se déclarer comme fournisseur de tel ou tel contenu (lorsqu'ils sont identifiés). Une autre application qui nécessiterait à un moment ce contenu émet une requête auprès de l'OS pour l'obtenir. L'OS lance alors directement l'application déclarée comme fournisseur.

II Organisation d'un projet Android

II.1 Arborescence

Un projet Android développé via l'ADT possède peu ou prou l'arborescence de fichiers et dossiers suivante :

- Le dossier *assets* (un bien) qui contient les paquetages externes utilisés par l'application développée : les fontes, les fichiers jar, etc....
- Le dossier *bin* qui contient toutes les classes compilées (.class) ainsi qu'une archive exécutable du projet (.apk)
- Le dossier *gen* qui contient les fichiers générés par l'ADT, notamment R.java
- Le dossier *res* qui contient les ressources statiques du projet, notamment :
  - les dossiers *drawable* : on y met les images du projet ;
  - le dossier *layout* : contient les descriptions XML de l'interface graphique ;

  – le dossier *menu* : contient les descriptions XML des menus ;
  – le dossier *values* : qui spécifie les chaînes de caractères (strings.xml), les styles. . .

  Noter que chaque dossier ressource peut être particularisé en fonction du support d'exécution cible (voir http ://developer.android.com/guide/topics/resources/providing-resources.html).
– Le dossier *src* (raccourci de « source »). C'est là qu'est placé l'ensemble des fichiers source Java de l'application. Pratiquement tout le travail effectué lors de la création d'une application Android est faite dans ce dossier ou bien dans le dossier "res"
– Le dossier *doc*. Raccourci pour documentation. C'est là que seront mises les documentation relatives au projet.
– Le fichier XML *AndroidManifest.xml* Ce fichier est généré par l'ADT lorsqu'on créé un nouveau projet Android. Ce fichier définit les fondamentaux de l'application : quelle est l'activité principale, quelles sont les permissions requises sur l'OS pour que l'application puisse s'exécuter,. . .

II.2 Travaux Pratiques : Hello World

Suivre le TP `Hello World`.

II.3 Exercice

1. Faire en sorte que lorsqu'on passe en mode paysage (dans l'émulateur `Ctrl+F11`) l'application `Hello World` affiche en plus *Landscreen*.
2. Comment faire en sorte que l'icône de lancement de l'application `Hello World` soit le même quelle que soit la résolution du mobile ? Modifier en conséquence le dossier `res`.
3. Comment faire en sorte que l'icône de lancement de l'application `Hello World` soit le fichier icon2.jpg lorsque l'affichage est en mode paysage ?
4. Comment faire en sorte que l'application affiche les messages en français si la langue du téléphone est le français et des messages en anglais sinon ?
5. Dans le code suivant extrait du fichier `manifest.xml` expliquer
   (a) la ligne 1
   (b) les ligne 2 à 4

```
1 <activity android:name=".HelloWorldActivity" android:label="@string/app_name">
2   <intent-filter>
3     <action android:name="android.intent.action.MAIN" />
4     <category android:name="android.intent.category.LAUNCHER" />
5   </intent-filter>
6 </activity>
```

Chapitre 2

Les interfaces utilisateur

On commence par créer une interface utilisateur graphique. La gestion des événements liés à celle-ci sera entrevue en TP et approfondie plus tard.

I Rendu visuel

Dans une application Android, l'interface utilisateur est construite à l'aide d'objets de type `View` et `ViewGroup`. Les objets de type `View` sont les unités de base, i.e. des composants gérant les interactions avec l'utilisateur. Les objets de type `ViewGroup` organisent la manière dont sont présents les composants. Un `ViewGroup` est classiquement défini à l'aide d'un `Layout`.

II Interface : programmation Java ou déclaration XML

La construction d'interface peut se faire selon deux approches complémentaires : la première est programmatique (Java) et la seconde est une déclaration à l'aide d'un fichier XML.

II.1 Programme Java

Dans la version programmatique, un objet `ViewGroup` est instancié dans le code Java (à l'aide du mot-clé `new`) comme un `Layout` particulier (ligne 4). Puis chaque composant est instancié (lignes 4 et 6) et ajouté à ce `ViewGroup` via la méthode `addView` (ligne 9 et 10) et enfin le `ViewGroup` est attaché à l'activité via `setContentView`.

```
1  public class Program extends Activity {
2      public void onCreate(Bundle savedInstanceState) {
3          super.onCreate(savedInstanceState);
4           LinearLayout llayout =  new LinearLayout(this);
5           llayout.setOrientation(LinearLayout.VERTICAL);
6           TextView nlbl = new TextView(this);
7           nlbl.setText("Nom");
8           EditText nedit = new EditText(this);
9           llayout.addView(nlbl);
10          llayout.addView(nedit);
11          setContentView(llayout);
12     }
13 }
```

On se passe complètement du fichier XML de description de l'organisation (dossier `res/layout`) dans une telle approche programmatique.

II.2 Déclaration XML

La philosophie est autre ici : l'organisation visuelle de l'activité est statique et définie une fois pour toute dans un fichier XML du dossier `res/layout`. Celui donné ci dessous aura la même interprétation visuelle que le programme donné avant.

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

```
     android:orientation="vertical"
     android:layout_width="fill_parent"
     android:layout_height="fill_parent">
  <TextView
    android:id="@+id/txtvname"
     android:layout_width="fill_parent"
     android:layout_height="wrap_content"
     android:text="@string/name" />
  <EditText
    android:id="@+id/edtname"
     android:layout_width="fill_parent"
     android:layout_height="wrap_content"/>
</LinearLayout>
```

Il reste uniquement à attacher cette organisation à l'activité via `setContentView` (ligne 4).

```
1 public class Program extends Activity {
2     public void onCreate(Bundle savedInstanceState) {
3         super.onCreate(savedInstanceState);
4         setContentView(R.layout.main);
5     }
6 }
```

III Les principaux composants graphiques

III.1 Exercice

On considère une activité dont la vue est liée au layout défini par les fichiers `res/layout/main.xml` et `res/values/arrays.xml` ci-dessous. Dire quels éléments vont être affichés, comment …

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3    android:orientation="vertical"
4    android:layout_width="fill_parent"
5    android:layout_height="fill_parent" >
6   <CheckBox android:id="@+id/cbxYN"
7     android:layout_width="20dp"
8     android:layout_height="20dp"
9     android:checked="false" />
10  <RadioGroup android:id="@+id/rgGroup1"
11    android:layout_width="fill_parent"
12    android:layout_height="wrap_content"
13    android:orientation="vertical">
14    <RadioButton android:id="@+id/RB1" android:text="Button1" />
15    <RadioButton android:id="@+id/RB2" android:text="Button2" />
16    <RadioButton android:id="@+id/RB3" android:text="Button3" />
17  </RadioGroup>
18  <Spinner android:id="@+id/spnSaisons"
19    android:layout_width="wrap_content"
20    android:layout_height="wrap_content"
21    android:entries="@array/saisons"/>
22  <DatePicker android:id="@+id/dPdate"
23    android:layout_width="wrap_content"
24    android:layout_height="wrap_content" />
25  <ImageView android:id="@+id/imgIcon"
26    android:layout_width="wrap_content"
27    android:layout_height="wrap_content"
28    android:layout_gravity = "center"
29    android:src="@drawable/icon" />
30 </LinearLayout>
```

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <resources>
3     <array name="saisons">
```

```
4           <item>printemps</item>
5           <item>ete</item>
6           <item>automne</item>
7           <item>hiver</item>
8       </array>
9 </resources>
```

IV Les Layouts

Les Layouts organisent le positionnement des composants graphiques dans l'interface utilisateur. On énonce les principales ci-dessous. On pourra se référer à `http://developer.android.com/guide/topics/ui/layout-objects.html` et `http://developer.android.com/resources/tutorials/views/index.html`

IV.1 FrameLayout

On empile les composant les uns sur les autres. Chacun est positionné dans le coin en haut à gauche en masquant plus ou moins celui du dessous sauf en cas de composant transparent.

IV.2 LinearLayout

Cette organisation aligne les composants dans une seule direction : verticalement ou horizontalement (en fonction de la valeur l'attribut `android:orientation`). Gère l'alignement (`android:gravity`) du composant.

IV.3 TableLayout

Cette organisation agence les composants selon un quadrillage (comme avec l'élément `<table>` en HTML). Il utilise pour cela l'élément `<tableRow>` qui déclare une nouvelle ligne. Les cellules sont définies par les composants qu'on ajoute aux lignes.

IV.4 RelativeLayout

Permet de déclarer des postions relativement par rapport au parent ou par rapport à d'autres composants. Il n'est pas facile à utiliser et on essaiera de s'en passer.

IV.5 Travaux Pratiques

Réaliser le TP intitulé « Les menus de l'utilisateur : Lanceur, menus et sudoku ».

V Les menus des activités

On se restreint ici aux menus que l'on peut lancer à partir du bouton menu du téléphone. Le modèle de construction est à nouveau basé sur le patron MVC :

1. le menu est défini statiquement en XML et gonflé via un inflater,
2. son contrôle se fait en gérant un événement et
3. le modèle est modifié en accédant à l'unique instance de celui-ci de manière statique.

V.1 Définition statique des menus

Les menus sont enregistrés dans un fichier xml du dossier `/res/menu` du projet. Ils contiennent éventuellement des sous-menus. Ci dessous on donne le fichier `/res/menu/sudoku_menu.xml` qui définit deux menus et un sous menu.

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/menu_param"
          android:icon="@drawable/ic_menu_preferences"
          android:title="@string/libelle_param">
    <menu>
    <group and
      android:checkableBehavior="single">
      <item android:id="@+id/menu_n1"
      android:title="@string/libelle_niv_1" />
      <item android:id="@+id/menu_n2"
      android:title="@string/libelle_niv_2" />
      <item android:id="@+id/menu_n3"
      android:title="@string/libelle_niv_3" />
    </group>
    </menu>
</item>
    <item android:id="@+id/menu_quitter"
          android:icon="@drawable/ic_menu_close_clear_cancel"
          android:title="@string/libelle_quitter" />
</menu>
```

V.2 Exercices

1. Que sont `@drawable/ic_menu_close_clear_cancel` et `ic_menu_preferences` ? Où récupérer de tels objets ?
2. Comment compléter le projet pour qu'il devienne exécutable ?
3. A quoi servent les lignes `<menu>...</menu>` ?

V.3 Gonflage du menus défini en XML

Lors de l'appuie sur le bouton menu du telephone, la méthode `onCreateOptionsMenu(Menu menu)` est invoquée dans l'objectif d'associer un menu xml à l'objet `menu` passé en paramètre.

Comme le `LayoutInflater` gonflait les layouts défini en xml, le `MenuInflater` gonfle les menus définis en xml. On effectue l'association comme suit :

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

V.4 Gestion des événements liés aux menu

La méthode `onOptionsItemSelected(MenuItem item)` de l'activité où est crée le menu est invoqué lorsqu'un élément est sélectionné dans le menu apparu sur le téléphone. Il suffit de donner un code traitant tous les items du menu (et des éventuels sous-menus) comme suit :

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  // Handle item selection
  switch (item.getItemId()) {
    case R.id.menu_n1:
    ...
    return true;
    case R.id.menu_quitter:
    this.finish();
    return true;
  }
}
```

V.5 Exercice

Compléter le code pour qu'on puisse choisir entre trois niveaux et que cela charge trois grilles différentes correspondantes aux niveaux.

Chapitre 3

Cycle de vie d'une activité

I Méthodes du cylce de vie

La figure 3.1 résume le cycle de vie d'une activité en montrant quelles méthodes sont appelées lors des événement qu'elle peut rencontrer.

1. `onCreate` : Cette méthode est tout d'abord invoquée lorsque l'activité est créee : C'est là qu'on associe la vue, initialise ou récupère les données persistantes... La méthode `onCreate` reçoit toujours en paramètreun objet `Bundle` qui contient l'état dans lequel était l'activité avant l'invocation.
2. `onStart` : Cette méthode est invoquée avant que l'activité soit visible et Une fois que l'exécution de cette méthode est finie,
   - si l'activité apparaît en premier plan, c'est `onResume` qui est invoquée
   - si l'activité ne peut apparaître en premier plan, c'est `onStop` qui est invoquée
3. `onResume` : Cette méthode est appelée immédiatement avant que l'activité ne passe en Elle est appelée soit parce qu'elle vient d'être (re)lancée, par exemple lorsqu'une autre activité a pris le devant puis a été fermée, remettant votre activité en premier plan.

   C'est souvent l'endroit où l'on reconstruit les interfaces en fonction de ce qui s'est passé depuis que l'utilisateur l'a vue pour la dernière fois.
4. `onPause` : Lorsque l'utilisateur est détourné de votre acivité, en la passant en second plan, c'est cette méthode qui est invoquée avant de pouvoir afficher la nouvelle activité en premier plan. Une fois cette méthode exécutée, Android peut tuer à tout moment l'activité sans vous redonner le contrôle. C'est donc là qu'on arrête proprement les services, threads,... et que l'on sauvegarde les données utiles à l'activité lorsqu'elle redémarera.
5. `onStop` : Cette méthode est invoquée lorsque l'activité n'est plus visible soit par ce qu'une autre est passée en premier plan, soit parce qu'elle est en cours de destruction.

II Exercice

Réaliser le TP intitulé « Cycle de vie d'une activité Android : Vie, pause et mort ».

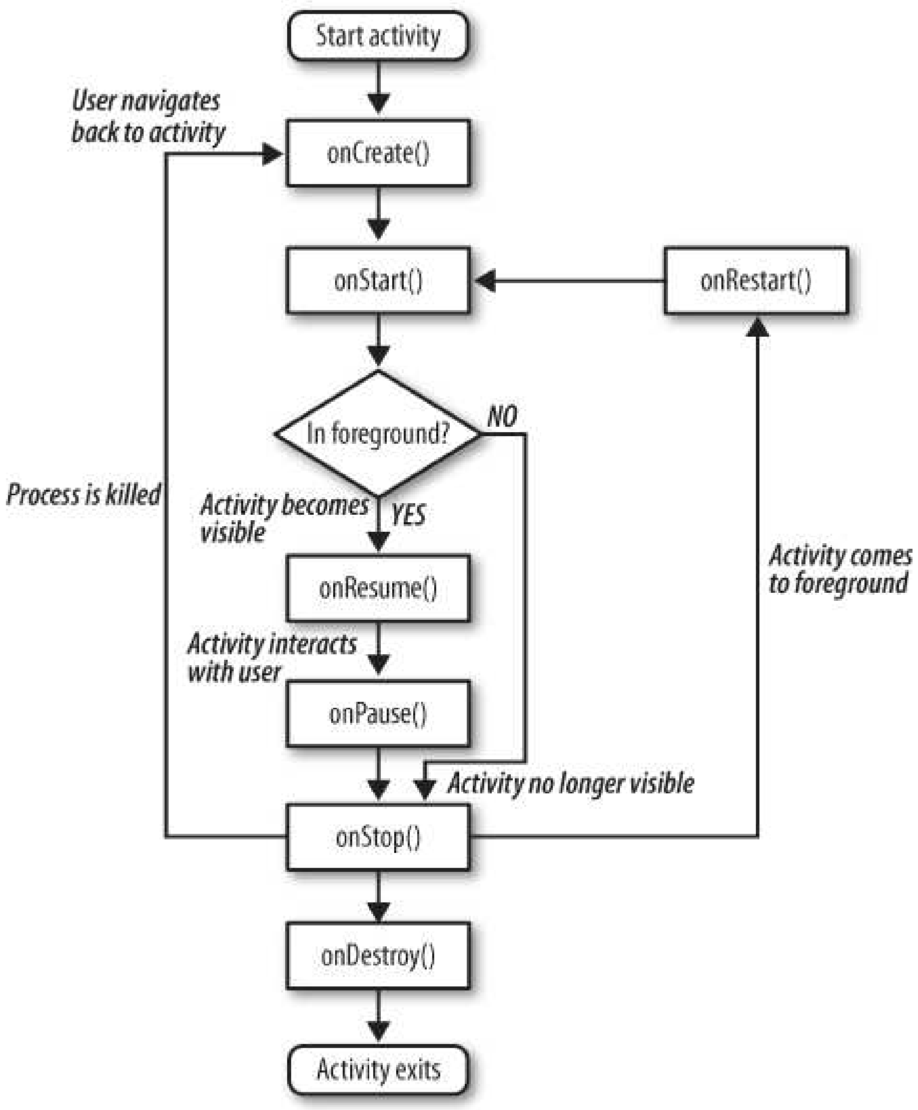

FIGURE 3.1 – Cycle de vie d'une activité android

Chapitre 4

Sauvegarder des données

Dans ce chapitre on voit comment sauvegarder des données d'une application android : les préférences (données courtes), les fichiers (données plus volumineuses) et la serialisation (en binaire).

I Les préférences

Android permet de mémoriser une information sous la forme d'une paire *(clef,valeur)* et d'y accéder lors de l'exécution de l'activité ou du service. La valeur est de type primaire (booléen,flottant, entier, long ou chaîne de caractères). Destinée essentiellement pour sauvegarder les préférences de l'application, la méthode permet en fait de stocker toute information pouvant se mettre sous la forme (clef,valeur).

I.1 Méthodes

On accède aux préférences partagées avec la méthode
`getSharedPreferences(String nomFichier, int typeAcces)`
où

- `typeAcces` est le type d'accès au fichier :
  - `MODE_PRIVATE` pour un accès interne à l'application,
  - `MODE_WORLD_READABLE` et
  - `MODE_WORLD_WRITEABLE` pour des accès universels en lecture et en écriture
- `nomFichier` est le nom du fichier à lire. S'il n'existe pas, il sera crée lorsqu'on écrira des données dedans

Notez que pour un nom de fichier donné, il n'y a qu'une seule instance d'objet correspondant à ce fichier. Cette instance est partagée par toutes les applications éventuellement.

Sur un objet de type `SharedPreferences`, on invoque la méthode `edit()` pour obtenir un flux en écriture. La méthode `putBoolean(String clef, boolean valeur)`, permet d'associer à une clef identifiée par une chaîne une valeur sous la forme d'un booléen. On a aussi `putString`, `putFloat`, `putInt` et `putLong` de même signature et au comportement évident. Les mises à jour ne sont réellement effectuées qu'après invocation de la méthode `commit()` sur le flux.

Pour récupérer les données enregistrées, sur un objet de type `SharedPreferences`, on invoque la méthode `getBoolean(String clef, boolean vd)` qui retourne la valeur booléenne associée à la clef si elle existe ou la valeur par défaut vd si elle n'existe pas. On a aussi, `getFloat(String clef, float vd)`, `getInt(String clef, int vd)`,`getLong(String clef, boolean vd)` pour les valeurs numériques et `getString(String clef, String vd)` pour récupérer une variable sous la forme d'une chaîne de caractères.

I.2 Exercices

1. Dans quelle méthode de l'activité principale devrait se faire la sauvegarde des préférences ?
2. Dans quelle méthode de l'activité principale devrait se faire une récupération des paramètres utilisateurs ?

I.3 Travaux Pratiques

1. Dans la classe `JeuxModel`, créer la méthode `grid2String()` qui retourne la grille sous la forme d'une chaîne de 81 caractères numériques.

2. Dans la classe `JeuxModel`, créer la méthode `string2grid(String r)` qui affecte à l'attribut `grid` la grille sous la forme d'une chaîne de 81 caractères numériques passée en paramètres.
3. Modifier le code de l'application pour que celle-ci sauvegarde l'état de la grille lorsqu'elle est est tuée.
4. Modifier le code de l'application pour que celle-ci recharge la dernière grille utilisée lorsque le bouton "continuer" est appuyé.

II Les fichiers

En plus de supporter les classes usuelles Java d'entrée/sortie la bibliothèque Android fournit les méthodes simplifiées `openFileInput` et `openFileOuput` permettant de lire et d'écrire aisément dans des fichiers. L'exemple suivant résume la situation :

```
try{
  FileOutputStream fos = openFileOutput("tempfile.tmp", MODE_PRIVATE);
  fos.write(new String("Hello World").getBytes());
  fos.close();

  String s = "";
  FileInputStream fis = openFileInput("tempfile.tmp");
  int b = fis.read();
  while (b != -1){
    s += new Character((char)b);
    b = fis.read();
  }
  fis.close();
  Toast.makeText (this,s,Toast.LENGTH_SHORT).show();
}catch(Exception e){
//
}
```

On note les contraintes suivantes :

- On ne peut créer un fichier qu'à l'intérieur du dossier de l'application. Préciser un séparateur de dossier ("\") dans la méthode `openFileOutput` engendre en effet une exception.
- Par contre d'autres applications peuvent éventuellement le lire et le modifier. Si l'on souhaite ceci, la permission à déclarer à la création sera `MODE_WORLD_READABLE` et `MODE_WORLD_WRITEABLE` respectivement.
- Si le fichier `tempfile.tmp` n'existe pas, l'appel à la méthode `openFileOutput` le crée.

III Le cloud

Cette partie est largement inspirée des pages 214 à 224 du livre "Android, guide de développements d'applications pour Smartphones et Tablettes" de Sébastien PÉROCHON aux éditions ENI, juillet 2011.

Depuis la version 2.2 (API 8) Android permet de sauvegarder les données de l'application sur un serveur de Google. Pour un utilisateur ayant plusieurs appareils, la même application a ses données cohérentes sur tous ces appareils grâce à un processus de synchronisation en arrière plan avec les serveurs de Google.

Pour une utilisation en dehors de l'émulateur les données sont associées au compte Google renseigné dans l'appareil. L'utilisateur doit avoir un compte Google identique sur tous ses appareils pour bénéficier de ce service.

Le principe général est le suivant : l'application communique avec le gestionnaire de sauvegarde des données persistantes sur le serveur. C'est ce dernier qui se charge d'envoyer les données dans le nuage et de les récupérer au lancement de l'application, par exemple.

III.1 Hériter de la classe `BackupAgentHelper`

La méthode la plus simple consiste à exploiter la classe `BackupAgentHelper` qui fournit une interface minimale pour synchroniser les données avec un serveur du cloud.

```
public class MonAgentDeSauvegarde extends BackupAgentHelper{
  public void onCreate(){
    SharedPreferencesBackupHelper assistantFichierPrefs = new SharedPreferencesBackupHelper(this,Sudoku.PREFS_NAME);
    addHelper("clefPourAgent",assistantFichierPrefs);
  }
}
```

Dans le code précédent, on a construit un agent de sauvegarde, qui hérite de la classe `BackupAgentHelper`. À la création, on comence par définir quel(s) fichier(s) doit(doivent) être synchronisés dans le cloud. Ici c'est un fichier de préférénce. La démarche est semblable pour un fichier quelconque. Ensuite, on renseigne que c'est cet assistant qui va être utilisé pour la synchronisation.

III.2 Modifier le manifest pour toute l'application

Pour bénéficier de ce service Google, le développeur doit tout d'abord réserver un espace dans le cloud. Cela se fait via la page suivante où doit être renseigné le nom du package.

```
http://code.google.com/intl/fr/android/backup/signup.html
```

Une fois enregistré, le développeur reçoit une clef, à insérer dans l'élément application du manifest comme suit :

```
<application>
  ...
  <meta-data android:name="com.google.android.backup.api_key"
           android:value="your_backup_service_key" />
</application>
```

Enfin, l'élément application doit définir l'attribut `android:backupAgent` pour préciser quelle est la classe qui assure la gestion de la persistance des données avec le serveur de Google.

```
<application android:icon="@drawable/icon"
             android:label="@string/app_name"
             android:backupAgent=".MonAgentDeSauvegarde">
```

III.3 La demande de synchronisation

Pour faire une demande de sauvegarde, l'application doit invoquer un objet de type `BackupManager` dans toute activité necéssitant la sauvegarde des préférences.

```
BackupManager backupManager = new BackupManager(this);
```

Pour lui dire que les données ont changé, on insère le code suivant chaque fois que les données doivent être modifiées.

```
this.backupManager.dataChanged();
```

III.4 Évaluer le développement

On peut simuler une sénario d'enregistrement dans le cloud dans une invite de commande comme suit :

```
adb shell bmgr enable true
adb shell bmgr backup your.package.name
adb shell bmgr run
adb uninstall your.package.name
```

Ici, successsivement
- on active le service de sauvegarde (ligne 1 et 2)
- puis on demande et on force l'enregistrement (ligne 3)
- enfin on supprime l'application

III.5 Travaux pratiques

Reprendre le code Sudoku et enregistrer le niveau de difficulté dans le cloud. Lorsqu'on réinstalle l'application, c'est la dernière valeur sauvegardée qui est affichée.

Chapitre 5

Récupération périodique de données grâce à un service

I Organisation générale

L'application android est composée d'une partie qui envoie à la demande des messages courts et d'une partie qui publie les derniers messages envoyés par tous les utilisateurs de l'application.

Pratiquement, aucune donnée n'est sauvegardée par l'application :

- lorsque l'utilisateur valide un message, celui-ci est envoyé à un serveur web qui mémorise le message dans une base de données.
- lors du lancement de l'application, un service est activé. Celui-ci interroge régulièrement le serveur web pour obtenir les derniers messages. Une fois ceux-ci récupérés, il envoie une intention à l'OS android. Cette intention est récupérée par l'application qui met à jour la liste des messages publiés.

II L'application web

Cette partie n'est pas détaillée car sort du contexte du cours.

II.1 La basse de données

Pour les besoins de l'exercice, on crée la base de données Mysql `messagerieAndroid` et la table `messages` selon le script suivant :

```
CREATE TABLE IF NOT EXISTS `messages` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `utilisateur` varchar(256) NOT NULL,
  `text` varchar(256) NOT NULL,
  `date` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`)
) ENGINE=MyISAM  DEFAULT CHARSET=latin1 AUTO_INCREMENT=2 ;
```

II.2 Les pages PHP

On construit 4 fichiers PHP sur le serveur :

- `conf.class.php` : c'est une classe dont les attributs statiques sont les paramètres de l'application web ;
- `message.class.php` : c'est une classe qui est la version objet de la table `messages` ; elle fournit une méthode d'ajout d'un message et une méthode statique de lecture des cinq derniers messages ;
- `ajoutemsg.php` : page PHP qui réceptionne les données envoyées par l'application android et qui demande l'insertion du message dans la base ;
- `recupemsg.php` : page PHP qui est interrogée régulièrement par le service android, qui demande la récupération des cinq derniers messages de la base et qui affiche ceci au format xml

Les Classes de l'application web On donne ci dessous le fichier `message.class.php`

```php
<?php
include_once "conf.class.php";
class Message{
  private $utilisateur;
  private $text;

  public function __construct($utilisateur,$text){
    $this->utilisateur = $utilisateur;
    $this->text = $text;
  }

  public final static function cnx(){
    $lien = mysql_connect(Conf::$server, Conf::$user,Conf::$passwd);
    $db = mysql_select_db(Conf::$base, $lien);
  }

  public function insere(){
    Message::cnx();
    $query = "INSERT INTO messages (id, utilisateur, text, date)
        VALUES (NULL, '$this->utilisateur', '$this->text', CURRENT_TIMESTAMP)";
    $result = mysql_query($query);
    mysql_free_result($result);
  }

  public final static function recupere_cinq_derniers(){
    Message::cnx();
    $query = "SELECT utilisateur, text, date FROM messages ORDER by date DESC
    LIMIT 5";
    $result = mysql_query($query);
    echo "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n";
    while ($row = mysql_fetch_assoc($result)) {
      echo "<msg>\n";
      echo "<utilisateur>".$row['utilisateur']."</utilisateur>\n";
      echo "<text>".$row['text']."</text>\n";
      echo "<date>".$row['date']."</date>\n";
      echo "</msg>\n";
    }
    mysql_free_result($result);
  }
}
?>
```

et le fichier `conf.class.php`

```php
<?php
class Conf{
  public static $server="_";
  public static $user="_";
  public static $passwd="_";
  public static $base="messagerieAndroid";
}
?>
```

Les pages pour l'Ajout et la suppression de messages Le fichier `ajoutemsg.php`

```php
<?php
include_once "message.class.php";
$msg = new Message($_GET['utilisateur'],$_GET['text']);
$msg->insere();
?>
```

et le fichier `recupemsg.php`

```php
<?php
include_once "message.class.php";
Message::recupere_cinq_derniers();
?>
```

III L'application Android

III.1 Vue organisée à l'aide d'onglets

L'organisation d'une interface utilisateur à l'aide d'onglets facilite souvent la navigation entre les principales parties d'une application. Pratiquement il suffit d'utiliser l'élément `<TabHost>` dans le layout principal de l'application comme suit :

```
<?xml version="1.0" encoding="utf-8"?>
<TabHost xmlns:android="http://schemas.android.com/apk/res/android"
   android:id="@android:id/tabhost"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent">
  <LinearLayout android:orientation="vertical"
     android:layout_width="fill_parent"
     android:layout_height="fill_parent">
    <TabWidget android:id="@android:id/tabs"
       android:layout_width="fill_parent"
       android:layout_height="wrap_content" />
    <FrameLayout android:id="@android:id/tabcontent"
       android:layout_width="fill_parent"
       android:layout_height="fill_parent">
      <LinearLayout android:id="@+id/tablinear"
         android:orientation="vertical"
         android:layout_width="fill_parent"
         android:layout_height="fill_parent">
        <TextView android:layout_width="wrap_content"
           android:layout_height="wrap_content"
           android:text="Utilisateur␣:␣" />
        <EditText android:id="@+id/edt_txt_u"
           android:layout_width="fill_parent"
           android:layout_height="wrap_content"/>
        <TextView android:layout_width="wrap_content"
           android:layout_height="wrap_content"
           android:text="Message␣:␣" />
        <EditText android:id="@+id/edt_txt_m"
           android:layout_width="fill_parent"
           android:layout_height="wrap_content"/>
        <Button android:id="@+id/btn"
           android:layout_width="wrap_content"
           android:layout_height="wrap_content"
           android:text="Valider"
           android:layout_gravity="center_horizontal"/>
      </LinearLayout>
      <ScrollView android:id="@+id/scrll"
         android:layout_width="fill_parent"
         android:layout_height="fill_parent">
        <TextView android:id="@+id/webtxt"
           android:layout_width="fill_parent"
           android:layout_height="fill_parent" />
      </ScrollView>
    </FrameLayout>
  </LinearLayout>
</TabHost>
```

Les identifiants des éléments `TabHost`, `TabWidget` et `FrameLayout` qui sont `@android:id/tabhost`, `@android:id/tabs` et `"@android:id/tabcontent` respectivement ne doivent pas être modifiés.

L'activité principale de l'application est représentée par des onglets, ceci se réalise en étendant la classe `TabActivity`. De plus, comme elle écoute le bouton de validation, elle implante l'interface `OnClickListener`.

Comme dans toute activité, la méthode `onCreate` vise à construire la vue liée à l'activité. La vue avec onglets est définie par un objet de type `TabHost` instancié directement par `getTabHost()` qui le lie à l'élément du même nom identifié par `@android:id/tabhost"`.

```
TabHost mTabHost = getTabHost();

TabSpec TabGSpec = mTabHost.newTabSpec("tab_message");
TabGSpec.setIndicator("N.␣Message", getResources().getDrawable(android.R.drawable.ic_menu_add));
```

```
TabGSpec.setContent(R.id.tablinear);

TabSpec TabDSpec = mTabHost.newTabSpec("tab_contribs");
TabDSpec.setIndicator("Contribs",getResources().getDrawable(android.R.drawable.ic_menu_view));
TabDSpec.setContent(R.id.webtxt);

mTabHost.addTab(TabGSpec);
mTabHost.addTab(TabDSpec);

Button bouton_envoyer = (Button)findViewById(R.id.btn);
bouton_envoyer.setOnClickListener(this);
```

III.2 L'envoi de messages

Il s'agit ici d'appeler une page php depuis l'application android. Quatre classes suffisent pour remplir cet objectif.

- `Uri.Builder` qui permet de construire méthodiquement l'url à utiliser. Cette classe fournit pour cela les méthodes
  - `scheme(String scheme)` : fixe le protocole (http, https...)
  - `authority(String authority)` : fixe l'adresse de l'url
  - `path(String path)` : fixe le chemin jusqu'à la page à atteindre
  - `appendQueryParameter(String key, String value)` : encode la clef et sa valeur.
  - `build` : construit l'objet correspondant à l'url `http://nom_serveur/ajoutemsg.php?utilisateur=couchot&text=blabla`
- `HttpGet` : l'url passée en paramètres lors de la construction d'un objet de ce type est amenée à être utilisée avec la méthode GET.
- `DefaultHttpClient` : c'est un client HTTP dont l'objectif est d'exécuter des requêtes via la méthode `execute(HttpUriRequest request)` qui exécute la requête (par exemple l'objet de la classe `HttpGet` donné plus haut). Cette méthode retourne une `HttpResponse` que l'on peut analyser pour y détecter des erreurs éventuelles.

Exercice

Dans la classe Messagerie :

1. construire la méthode `envoie_msg(String utilisateur, String msg)` qui envoie le message identifié par la paire (`utilisateur`, `msg`) à la page `ajoutemsg.php` hébergée sur votre serveur ;
2. construire la méthode `onClick` qui invoque la méthode `envoie_msg` avec les paramètres récupérés dans les objets de la classe `EditText` ;
3. vérifier la correction du code en constatant que les messages ont bien été mémorisés dans votre base de données ;
4. essayer de faire de même en s'adressant à la même page stockée sur le serveur de votre voisin.

III.3 Le service de réception de messages

Le travail de récupération des message se fait à l'aide d'un service indépendant qui périodiquement interroge la page `recupemsg.php` et affiche son contenu.

Deux classes et trois étapes suffisent pour réaliser ceci :

- la classe `RecupService` qui étend `Service`, qui contient l'objet chrono (de la classe `Timer`) et la tâche à réaliser `tmt` (de la classe `RecupTimerTask`). Ces deux objets sont initialisés comme suit :

  ```
  public void onStart(Intent intent, int startId) {
    chrono.scheduleAtFixedRate(tmt,0,(long)10000);}

  public void onCreate() {
    chrono = new Timer("chrono_maj_msg");
    tmt  = new RecupTimerTask(this);}
  ```

- la classe `RecupTimerTask` qui étend `TimerTask`. Sa méthode `run()` appelée toutes les 10s dans l'exemple invoque la méthode `msg_maj()` de la classe `RecupService`
- dans la classe `RecupService`, la méthode `msg_maj()`
  1. interroge la page `recupemsg.php` via un client `DefaultHttpClient` comme avant ;

2. récupère la réponse via un gestionnaire de réponse `BasicResponseHandler`
3. construit une intention et l'envoie à l'OS Android comme suit :

   ```
   Intent intent = new Intent("liste de message");
   intent.putExtra(``log'', chaine);
   this.sendBroadcast(intent);
   ```

Exercice

Compléter le développement des deux classes décrites ci-dessus et vérifier dans le débugueur que les intentions sont envoyées par le service.

III.4 L'affichage des messages

Pour capturer les intentions émises par le service, il suffit dans la classe Messagerie d'instancier un récepteur de la classe `MsgRecepteur` et de le déclarer comme receveur d'intentions de la sorte "liste de message" comme suit :

```
this.msgrcpt = new MsgRecepteur(this);
IntentFilter filter = new IntentFilter("liste de message");
this.registerReceiver(this.msgrcpt, filter);
```

Ceci se fait dans la méthode `onResume` de la classe `Messagerie`. Dire pourquoi. Enfin le la classe `MsgRecepteur` qui étend `BroadcastReceiver` doit définir la méthode `onReceive(Context arg0, Intent arg1)` : celle-ci invoque uniquement la méthode `msg_affichage_maj` (à définir) de la classe `Messagerie` qui remplit le texte de la textview.