Chapitre 7

Persistance et partage des données

1 Introduction

Android offre plusieurs méthodes pour stocker les données d'une application.

La solution choisie va dépendre des besoins : données privées ou publiques, un petit ensemble de données à préserver ou un large ensemble à préserver localement ou via le réseau.

Ces méthodes utilisent soit des éléments propres à l'API Java ou ceux associés à l'API d'Android.

Ces méthodes sont :

Persistance dans l'état d'une application

On utilise pour cela la notion de « Bundle » et les différents cycles de l'activité pour sauvegarder l'information utile à l'aide du « bundle » et récupérer cette information dans un autre état de l'activité.

On ne peut utiliser qu'un seul bundle, par ailleurs la donnée n'est pas persistante et n'est disponible que tant que l'application est utilisée.

Préférences partagées

Un ensemble de paires : clé et valeur. Clé est un « String » et Valeur est un type primitif (« boolean », « String », etc.).

Ces préférences sont gérées à travers un code Java ou bien via une activité.

Les données ne sont pas cryptées.

Les préférences sont adaptées pour des paires simples, mais dès qu'il est question de données plus complexes, il est préférable d'utiliser des fichiers.

Fichiers (création et sauvegarde)

Android permet la création, la sauvegarde et la lecture de fichiers à travers un média persistant (mémorisation et disponibilité).

Les fichiers peuvent être de n'importe quel type (image, XML, etc.).

Les fichiers peuvent être considérés pour une utilisation interne, donc local à l'application, ou bien externe, donc partagée avec plusieurs applications.

Base de données relationnelle, SQLite

Android offre aussi la possibilité d'utiliser toutes les propriétés d'une base de données relationnelle.

Android utilise pour cela une base de données basée sur « SQLite » (www.sqlite.org).

Android stocke la base de données localement à l'application.

Si l'on veut partager cette structure de données avec d'autres applications, il faudra utiliser dans ce cas, un gestionnaire de contenu (content provider) configuré à cet effet.

Stockage réseau

Android permet de stocker des fichiers sur un serveur distant.

On peut utiliser pour cela les différentes techniques examinées dans le chapitre « Internet ».

2 Persistance dans l'état d'une application

Android peut arrêter l'activité et la redémarrer quand il y a :

- Rotation de l'écran.
- Changement de langue.
- L'application est en arrière-plan et le système a besoin de ressources.
- Et quand vous cliquez le bouton « retour » (« back »).

Ce redémarrage peut provoquer la perte des changements apportés à votre activité.

Une solution consiste à préserver les données dans un bundle à travers la méthode onSaveInstanceState puis récupérer cette information par la suite à l'aide de la méthode onRestoreInstanceState (ou bien dans la méthode onCreate).

Cette solution n'est pas appropriée dans le cas d'un clic sur le bouton « retour ». Dans ce cas, l'application démarre à partir de zéro et les données préservées sont détruites.

Généralement, l'orientation est gérée par la création d'une vue appropriée. Mais supposons que vous n'ayez pas eu le temps de le faire!

Nous pouvons fixer dans le fichier « AndroidManifest.xml » l'orientation supportée, comme suit :

```
<activity android:name=".YourActivity"
android:label="@string/app_name"
android:screenOrientation="portrait">
```

On déclare un widget « EditText », on insère une valeur quelconque, puis on change l'orientation de l'écran (CTRL-F11/CTRL-F12) :

```
<EditText android:layout_width="fill_parent"
  android:layout_height="wrap_content"/>
```

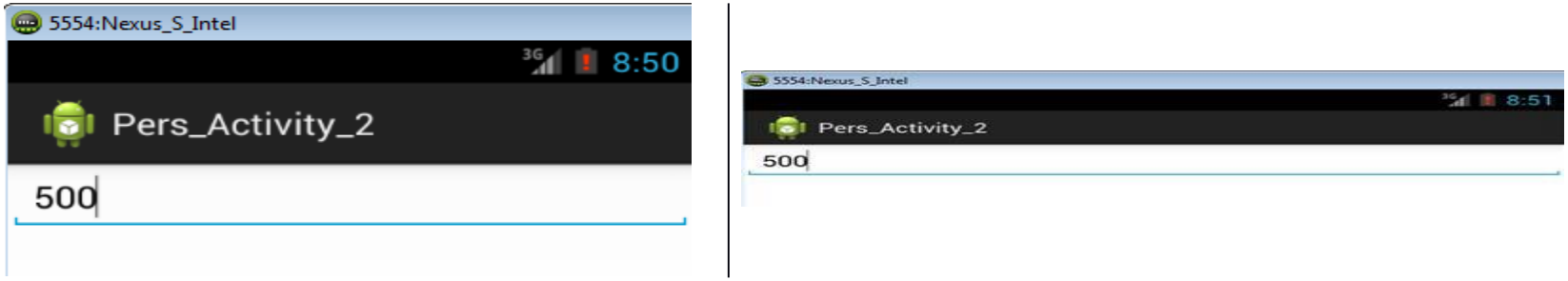
On constate que l'on a perdu le contenu du Widget en changeant l'orientation de l'écran.

Si on avait ajouté dans le fichier manifeste cette ligne « `android:configChanges="orientation|screenSize"` », le contenu du Widget va être préservé. Cependant si une vue paysage a été définie avec l'application, Android ne va pas la prendre en considération.

On reprend la même activité et on identifie maintenant le Widget, « EditText », avec un identificateur comme suit :

```
<EditText android:id="@+id/edittext1"
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"/>
```

On teste de nouveau l'activité et on remarque maintenant que le contenu du Widget a été préservé.

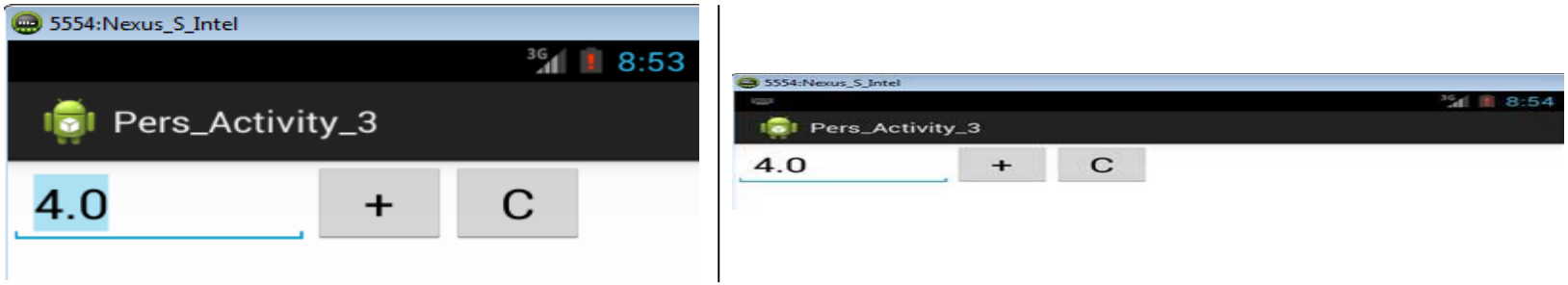


Le fait, de passer d'une orientation à une autre, a fait en sorte que l'activité a été arrêtée puis démarrée de nouveau. Sauf que cette fois-ci, le contenu du Widget n'a pas été remis à zéro.

Android préserve automatiquement l'état d'un Widget quand ce dernier est identifié. Ce qui est le cas dans le second exemple.

Prenons un autre exemple, une simple calculatrice.

Faites 2 + 2 +, le résultat sera 4. Faites tourner l'orientation de l'écran, le résultat va rester 4. À noter que le bouton « C » permet de remettre à zéro la calculatrice.



Faites maintenant le test suivant : appuyez sur les touches 2 + 2, puis changez l'orientation de l'écran.

Appuyez maintenant sur la touche + et examinez le résultat :

Le résultat final est égal à 2 et non pas 4! Un bogue!

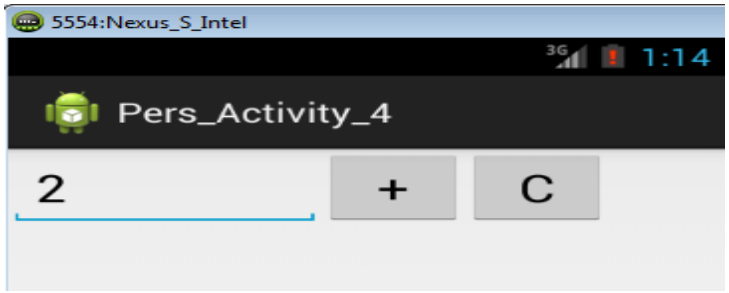
En changeant d'orientation, l'activité a été arrêtée puis démarrée de nouveau. Durant cette opération, nous avons perdu la valeur de la somme intermédiaire, ce qui explique pourquoi la somme finale affichée est erronée.

Dans le code Java associé à cette activité, la somme intermédiaire est représentée par la variable « total ». Il faudra donc préserver l'état de cette variable au moment de changer d'orientation.

Pour ce faire, nous allons utiliser la méthode « onSaveInstanceState » pour préserver la valeur associée à cette variable au moment de l'arrêt de l'application.

On peut récupérer cette valeur lors du démarrage de l'activité à l'aide de la méthode « onRestoreInstanceState ».

```
@Override
public void onSaveInstanceState(Bundle outState) {
      super.onSaveInstanceState(outState);
      outState.putFloat("TOTAL", total);
}
@Override
public void onRestoreInstanceState(Bundle savedInstanceState) {
      super.onRestoreInstanceState(savedInstanceState);
      total=savedInstanceState.getFloat("TOTAL");
}
```



On clique sur le signe + :

Le total est maintenant correct.

-

3 Préférences partagées

Cette méthode permet de sauvegarder les préférences dans un fichier.

Ces préférences seront accessibles aux différentes activités associées à l'application.

Les préférences seront utilisées pour sauvegarder l'état de la configuration de l'application ou bien les données de session.

Les préférences sont comme les « bundles » sauf qu'elles sont persistantes ce qui n'est pas le cas des bundles.

Les préférences ne sont pas cryptées. Il faudra faire attention si l'intention est de préserver des données critiques.

Les préférences peuvent être effacées par l'utilisateur de l'application.

Chaque préférence a la forme d'une paire dont la clé est un élément du type String et dont la valeur est un des types primitifs (int, long, float, boolean) ou bien une collection de String (Set<String>).

Cette méthode n'est appropriée que pour une petite collection de paires.

Nous allons utiliser les méthodes de la classe « SharedPreferences » pour sauvegarder puis lire une préférence.

Une instance de la classe « SharedPreferences » est créée dans un mode prédéfini. Le mode le plus couramment utilisé est « MODE_PRIVATE » pour signifier que les préférences ne seront accessibles que par l'application.

Il existe d'autres modes :

- MODE_WORLD_READABLE : les autres applications peuvent lire l'ensemble,
- MODE_WORLD_WRITEABLE : les autres applications peuvent modifier l'ensemble,
- MODE_MULTI_PROCESS : plusieurs processus peuvent accéder à l'ensemble.

Les modes « WORLD_READABLE » et « WORLD_WRITEABLE » sont dépréciés depuis l'API 17. Ils provoqueront l'exception « SecurityException » depuis l'API 24 (Nougat).

Création :

Nous pouvons utiliser l'une des deux méthodes :

getPreference(int mode) : elle est associée à l'activité courante. Une procédure transparente est définie par défaut pour préserver les préférences désirées et associées à l'activité. Nous devons juste signifier l'argument qui représente l'un des modes d'accès précédemment mentionnés.

```
private SharedPreferences settings =
                getPreference(context.MODE_PRIVATE);
```

getSharedPreferences(String nom,int mode) : elle est utilisée si les préférences sont préservées dans plusieurs fichiers. Le premier argument représente le nom du fichier à utiliser et le second argument, le mode d'accès.

```
private SharedPreferences settings =
    getSharedPreference("nom_preferences",context.MODE_PRIVATE);
```

On édite la préférence:

```
SharedPreferences.Editor editor = settings.edit();
```

On définit la paire « string,string » et on valide l'inscription dans le conteneur des préférences :

```
editor.putString(NOM, PrefValeur);
editor.apply();
```

Pour récupérer la paire :

```
PrefValeur = parametres.getString(NOM, "Introuvable");
```

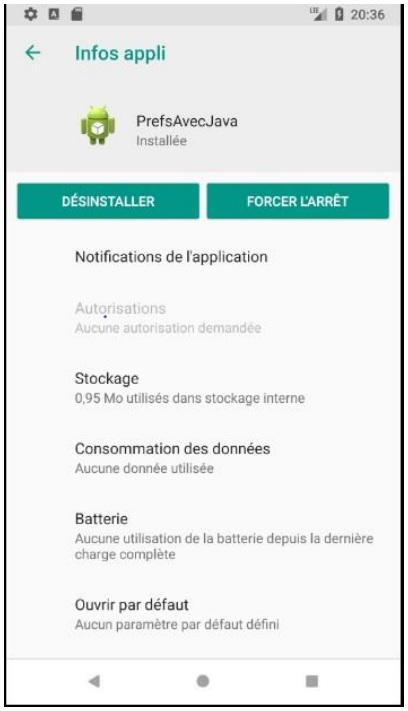
On peut détruire la préférence en détruisant la clé associée à l'aide de la méthode « removeString(String clé) ». On peut détruire toutes les préférences à l'aide de la méthode « clear » :

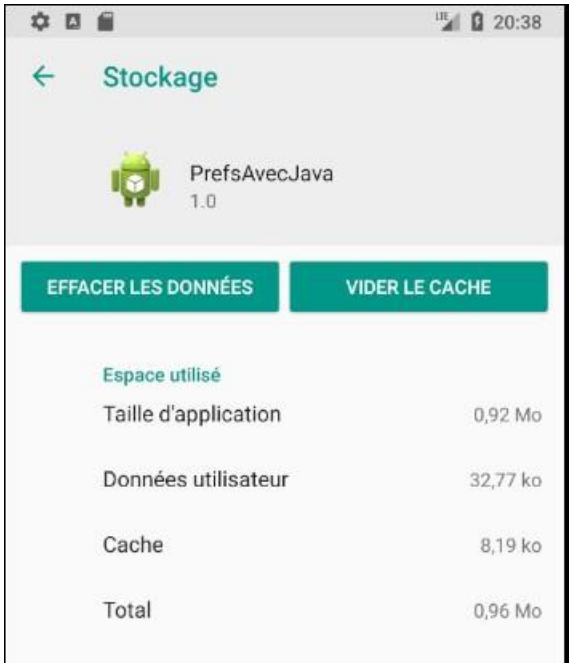
```
editor.removeString(Nom);

editor.clear();
```

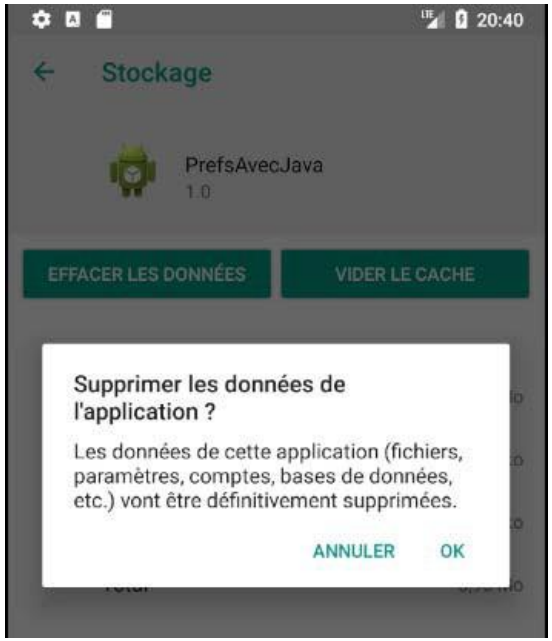
L'utilisateur peut aussi détruire les préférences sauvegardées par son application en procédant ainsi sous « Pie » :

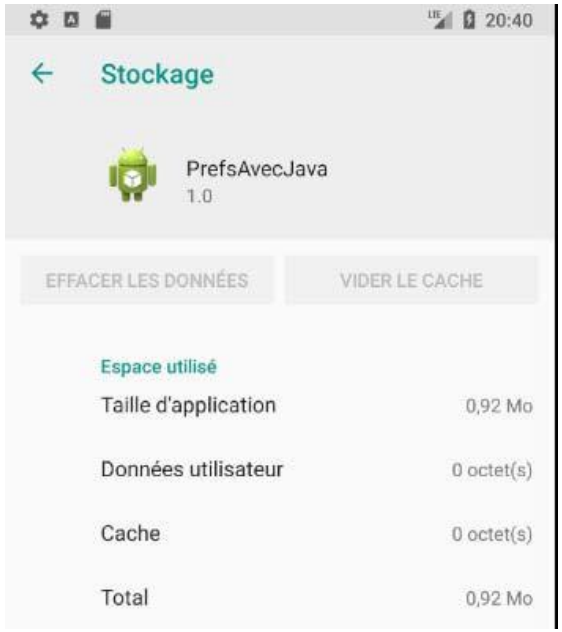
- On se positionne sur le menu des applications.
- On sélectionne notre application et on maintient le bouton gauche de la souris enfoncé. Une nouvelle fenêtre va apparaitre. Cliquer sur « App info ». Vous allez obtenir ce qui suit :





Vous cliquer par la suite sur « Stockage ». Vous pouvez effacer les préférences en cliquant sur le bouton « EFFACER LES DONNÉES ».





Une autre manière de procéder est de passer par « settings », puis « Apps », puis sélectionnez votre application « PrefsAvecJava ».

Les préférences sont sauvegardées dans un fichier. Ce fichier a par défaut le format « XML » et est localisé à :

/data/data/cis493.preferences/shared_prefs/MyPreferences_001.xml

Sur l'émulateur, vous pouvez voir l'existence du fichier grâce à la vue « Device File Explorer ». Vous pouvez aussi extraire le fichier à l'aide de « adb » et examiner son contenu. Il faut avoir les privilèges d'un utilisateur « root » pour pouvoir y avoir accès. Pour obtenir ces privilèges dans le cadre d'un émulateur, exécutez la commande :

```
adb root
restarting adbd as root
adb pull
 /data/data/cis493.preferences/shared_prefs/MyPreferences_001.xml .
```

Choisissez en premier « Pref Simple UI » et examinez par la suite le contenu du fichier « MyPreferences_001.xml » :

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
<int name="textSize" value="15" />
<int name="backColor" value="-1" />
</map>
```

Choisissez par la suite « Pref Fancy UI » et examinez le contenu du fichier « MyPreferences_001.xml » :

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
<int name="layoutColor" value="-16711936" />
<int name="textSize" value="20" />
<int name="backColor" value="-16776961" />
<string name="textStyle">bold</string>
</map>
```

Cliquer maintenait sur le bouton retour (« Back ») puis examinez le contenu du fichier « MyPreferences_001.xml » :

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <int name="layoutColor" value="-16711936" />
    <string name="DateLastExecution">3 févr. 2018</string>
    <int name="textSize" value="20" />
    <int name="backColor" value="-16776961" />
    <string name="textStyle">bold</string>
</map>
```

Sauvegarde des préférences à travers une activité

L'exemple crée une interface pour gérer les préférences de l'utilisateur.

En utilisant ce programme avec, par exemple, l'API 17, nous obtenons cet avertissement pour l'appel :

```
addPreferencesFromResource(R.xml.prefs);
```

[The method addPreferencesFromResource(int) from the type PreferenceActivity is deprecated].

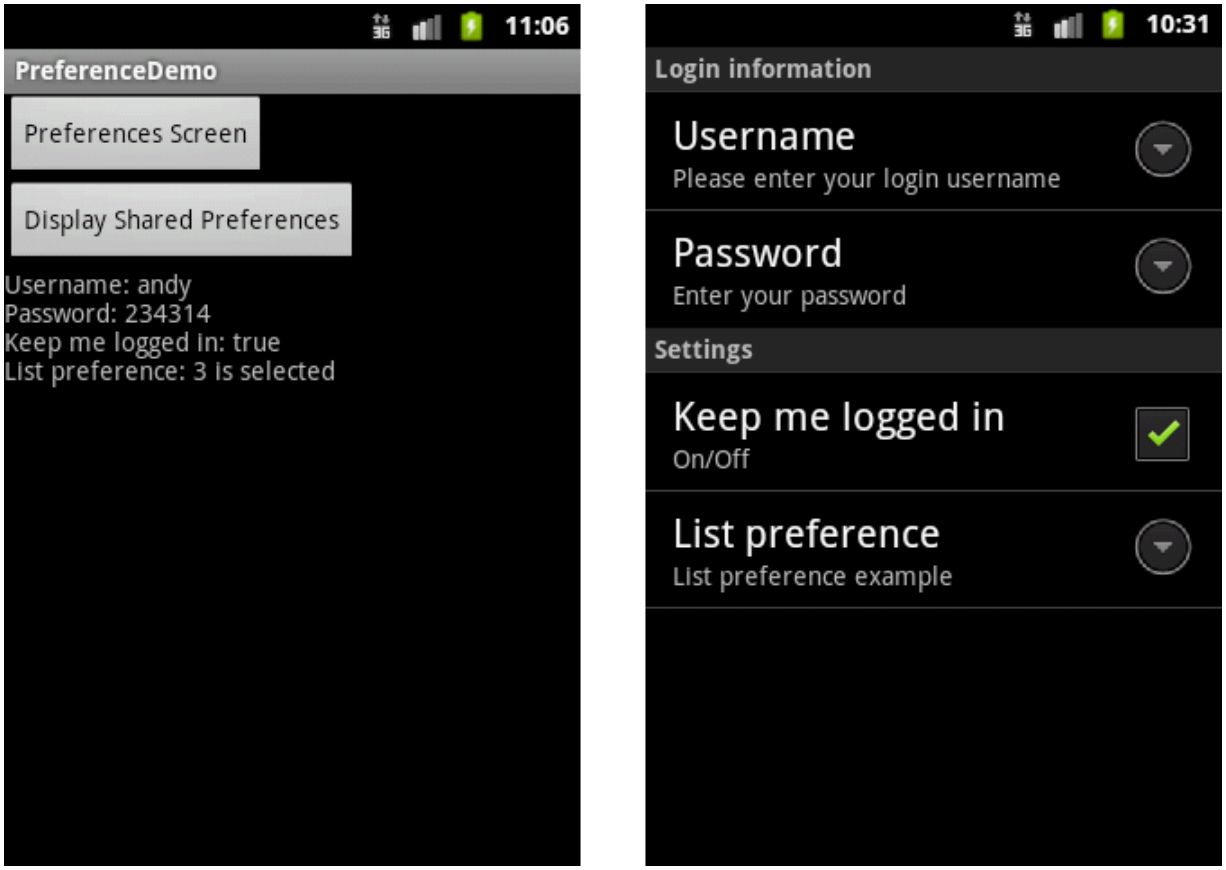
La méthode « `addPreferencesFromResource` » de la classe « PreferenceActivity » a été déclarée désuète à partir de l'API 11 (Honeycomb).

Nous devons utiliser à la place la méthode équivalente définie dans la classe « PreferenceFragment ».

Nous avons dû donc apporter des modifications à cet exemple pour qu'il soit compatible, peu importe la version de l'API utilisé.

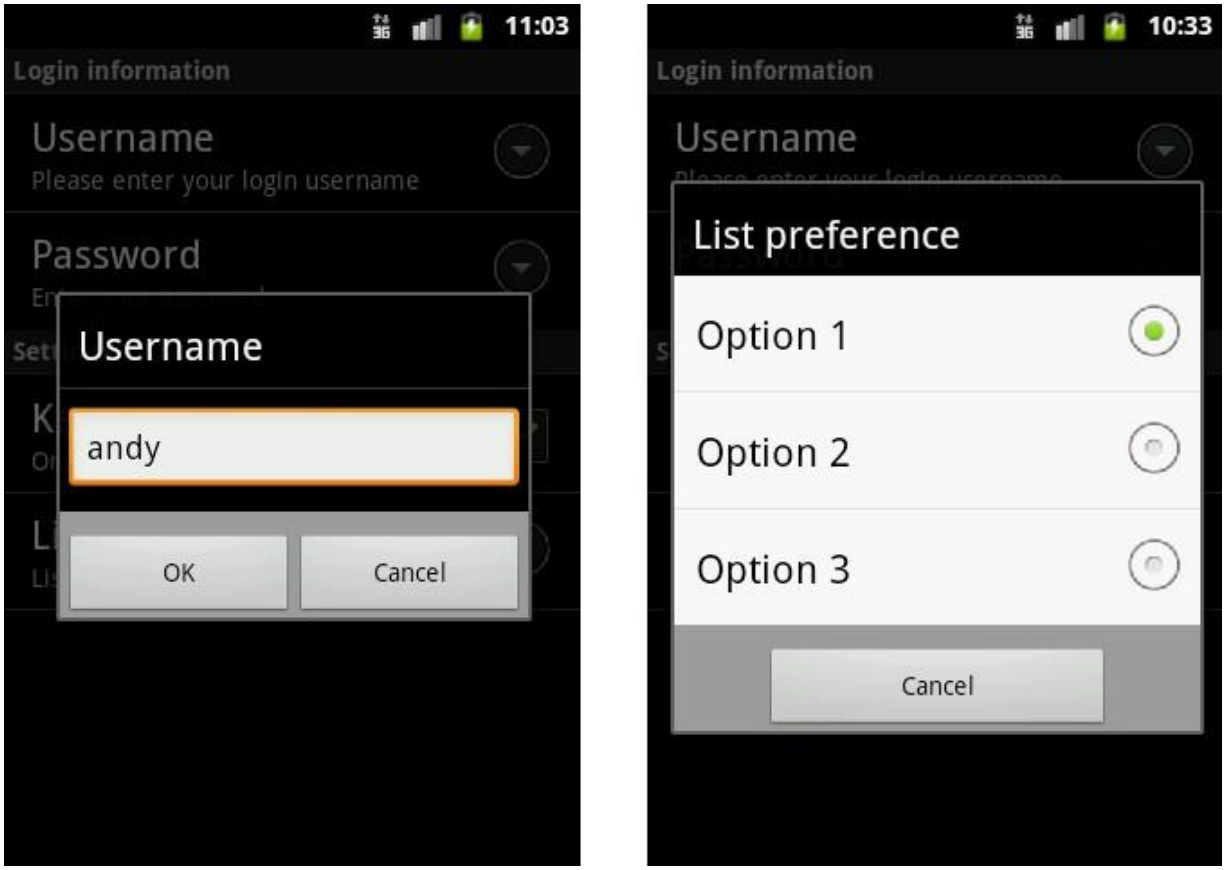
Un test d'API va permettre de choisir entre « PreferenceActivity » et « PreferenceFragment ».

Nous allons commencer par décrire brièvement l'exemple :



En cliquant sur le bouton « Preferences Screen », une nouvelle activité va être lancée (écran de droite).

Quand l'utilisateur choisit une des options …



Écran de gauche, nom d'utilisateur; écran de droite, la liste des préférences.

Le contenu de cette 2e activité se trouve dans le fichier « prefs.xml », dans le répertoire « res/XML ». Nous avons ajouté aussi le fichier « PrefsActivity.java », qui contient cet appel dans la méthode « onCreate » :

```
addPreferencesFromResource(R.xml.prefs);
```

Cette méthode permet de télécharger des préférences à partir d'une ressource, décrite dans un fichier « XML ». Ce fichier va définir la vue de l'activité.

Examiner le répertoire « xml » dans la hiérarchie du projet.

Dans le fichier « prefs.xml », le tag « PreferenceScreen » sera utilisé comme la racine du fichier. Quand une activité pointe ce fichier, le tag « PreferenceScreen » sera utilisé comme le point d'entrée.

Nous distinguons plusieurs tags :

<PreferenceCategory>	définit une catégorie de préférences. Pour l'exemple, nous avons défini deux catégories « Login Information » et « Settings ».
<EditTextPreference>	définit un champ pour stocker de l'information.
<CheckBoxPreference>	définit une boîte à cocher, « checkbox ».
<ListPreference>	définit une liste d'éléments. La liste apparait comme des boutons radio.

Par la suite, l'activité principale récupère automatiquement les préférences préservées :

```
SharedPreferences prefs =
            PreferenceManager.getDefaultSharedPreferences(
                                        PreferenceDemoActivity.this);
```

Pour l'API 17, nous avons développé une version avec fragments.

Nous avons défini une classe pour lire le fichier « XML » :

```
public class PrefsActivity02 extends PreferenceFragment {
@Override
public void onCreate(Bundle savedInstanceState) {
     super.onCreate(savedInstanceState);
     addPreferencesFromResource(R.xml.prefs);
}
}
```

Nous avons apporté aussi les modifications suivantes à la classe « PrefsActivity » :

```
if (Build.VERSION.SDK_INT < Build.VERSION_CODES.HONEYCOMB) {
     addPreferencesFromResource(R.xml.prefs);
}else{
     getFragmentManager().beginTransaction().replace(android.R.id.content,
               new PrefsActivity02()).commit();
}
```

Ainsi donc, la commande à exécuter va dépendre de la version de l'API.

4 Fichiers (création et sauvegarde)

Android s'appuie sur l'API Java pour réaliser la gestion de fichiers.

Préférences partagées

Une première utilisation des fichiers était en rapport avec les préférences partagées. Nous avons expliqué comment il était possible de paramétrer la méthode « `getSharedPreference` » pour préserver ces préférences dans un fichier.

Stockage interne

Comme nous l'avons fait depuis le début du cours, nous pouvons ajouter des fichiers supplémentaires à l'application. Ces fichiers peuvent être disposés dans divers endroits en fonction de leur utilisation : « assets », « res/raw », etc.

Ils feront partie du paquetage « apk » final.

Pour l'exemple, nous allons d'abord préserver le fichier « my_text_file.txt » dans le répertoire « res/raw » de l'application. Par la suite, l'application va se charger de lire le fichier en question et de l'afficher dans la vue principale.

Association d'un flux à la ressource :

```
int fileResourceId = R.raw.my_text_file;
InputStream is = this.getResources()
                        .openRawResource(fileResourceId);
```

Lecture du contenu de la ressource :

```
if (is!=null) {
    BufferedReader reader =
            new BufferedReader(new InputStreamReader(is));
    while ((str = reader.readLine()) != null) {
        buf.append(str).append("\n");
    }
}
```

Comme il s'agit d'une opération I/O, il faudra penser à capturer l'exception « IOException ».

Quand l'application démarre pour la première fois, nous allons d'abord écrire un texte dans le fichier « notes.txt » et cliquer sur le bouton de sauvegarde. Ce bouton va terminer l'application, mais avant cela, la méthode « onPause » sera exécutée. Cette méthode aura la tâche de sauvegarder le texte dans le fichier « notes.txt ».

Examiner le contenu du fichier « notes.txt » qui se trouve dans le répertoire :

```
/data/data/cis470.matos.filewriteread/files
```

Il faut être « root » pour y accéder au répertoire.

Quand l'activité est démarrée une seconde fois, la méthode « onStart » est exécutée. Cette méthode va se charger de lire puis d'afficher le contenu du fichier sur l'écran.

Stockage externe

Nous allons examiner comment effectuer des opérations de lecture et écriture à partir d'un média externe. Nous allons utiliser pour cela une carte mémoire SD pour effectuer ces opérations I/O.

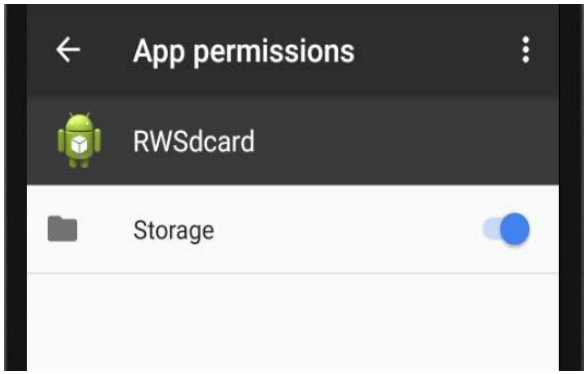
Toutes les applications ont par défaut la permission de lire à partir d'une unité externe. Ce comportement va changer dans les prochaines versions de l'API. Il est préférable donc d'ajouter la permission de lecture dès à présent, si votre application doit lire à partir d'une unité externe.

La permission d'écrire n'est pas accordée par défaut. Il faudra l'autoriser explicitement.

Ces permissions doivent être ajoutées dans le fichier « AndroidManifest.xml » :

```
<uses-permission
    android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

Il faudra manuellement activer la permission de stockage de votre application :



Comme Android est basé sur un noyau Linux, il faudra vérifier que l'unité externe est bien disponible (l'expression utilisée dans le jargon est « montée »).

```
public boolean isExternalStorageWritable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state)) {
        return true;
    }
    return false;
}
```

La méthode « `Environment.getExternalStorageState()` » retourne un « MEDIA_MOUNTED » pour nous informer si le média est présent, monté, et dans quel mode, lecture/écriture.

Nous testons par la suite le mode qui a été autorisé avec la méthode « `Environment.MEDIA_MOUNTED.equals` ». Si elle retourne « true » alors l'écriture a été autorisée. Dans le cas contraire, il faudra vérifier si la lecture a été autorisée en procédant ainsi :

```
/* Checks if external storage is available to at least read */
public boolean isExternalStorageReadable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state) ||
        Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
        return true;
    }
    return false;
}
```

Dans notre exemple, nous avons mis en cascade les différents appels comme suit :

```
if(!Environment.MEDIA_MOUNTED.equals
            (Environment.getExternalStorageState())){
    Toast.makeText(this, "External SD card not mounted",
                                    Toast.LENGTH_LONG).show();
}
```

Après avoir validé la disponibilité de l'unité externe, pour préciser le chemin à joindre au nom du fichier, nous allons faire appel à la méthode « `Environment.getExternalStorageDirectory()` » :

```
File root = Environment.getExternalStorageDirectory();
File directory = new File (root.getAbsolutePath() + "/Data");
boolean wasSuccessfulmkdirs = directory.mkdirs();
File myFile = new File (directory, "textfile.txt");
boolean wasSuccessfulnewfile = myFile.createNewFile();
```

Le fichier « textfile.txt » va être disponible dans le répertoire « /sdcard/Data ».

Les autres méthodes de lecture et d'écriture sont celles de l'API Java.

Nous pouvons vérifier par la suite avec la commande « adb » la présence du fichier et son contenu :

```
adb shell
cd sdcard
cd Data
cat textfile.txt
```

5 SQLite

Android intègre le système de gestion de bases de données, SQLite.

Pour plus de détails, consultez ce lien : http://www.sqlite.org/

C'est un système compact, très efficace pour les systèmes embarqués. En effet, il utilise très peu de mémoire.

SQLite ne nécessite pas de serveur pour fonctionner, ce qui n'est pas le cas de MySQL par exemple.

Les opérations sur la base de données se feront donc dans le même processus que l'application. Il faudra faire attention aux opérations « lourdes », votre application va ressentir les contres coups. Il est conseillé dans ce cas d'utiliser les tâches asynchrones (ou threads).

Chaque application peut avoir donc ses propres bases.

Ces bases sont stockées dans le répertoire « databases » associé à l'application (`/data/data/APP_NAME/databases/nom_base`). Nous pouvons les stocker aussi sur une unité externe (sdcard).

Chaque base créée, elle le sera en mode « MODE_PRIVATE ». Aucune autre application ne peut y accéder que l'application qui l'a créée.

Pour y avoir accès, il faut que la base ait été sauvegardée sur un support externe, sinon utiliser le mécanisme d'échange de données fourni par Android (il sera développé plus tard dans ce chapitre).

L’accès par « adb » nécessite les privilèges « root ».

SQLite supporte les types : TEXT (chaîne de caractères), INTEGER (entiers), REAL (réels). Tous les types doivent être convertis pour être utilisés. SQLite ne vérifie pas le typage des éléments. À vous de vous en assurer que vous n’avez pas écrit un entier à la place d’une chaîne de caractères par exemple.

Création et mise à jour de la base

L’exemple va créer une table de commentaires. Chaque commentaire est identifié par un identificateur unique.

Nom de la table : `comments`	
`_id`	`comments`

La base va porter le nom « `commments.db` ».

L’organisation des fichiers permet de faciliter l’organisation de la base de données et la compréhension de l’exemple.

- Le fichier « `Comment.java` » va contenir un enregistrement d’une table et les différentes méthodes qui gravitent autour.

La classe « Comment » décrite dans le fichier « Comment.java » contient deux attributs :

```
private long id;
private String comment;
```

La base de données doit utiliser un identifiant unique « _id » comme clé primaire de la table. Des méthodes d'Android se servent de ce standard.

- Le fichier « MySQLiteHelper.java » contient la classe qui dérive de « SQLiteOpenHelper ».

La classe « MySQLiteHelper » :

Créez une nouvelle classe qui va dériver de la classe « SQLiteOpenHelper » :

```
public class MySQLiteHelper extends SQLiteOpenHelper { …}
```

Dans le constructeur de la classe, faites appel à la méthode « super » de « SQLiteOpenHelper » et spécifiez le nom de la base et sa version.

```
public MySQLiteHelper(Context context) {
    super(context, "commments.db", null, 1);
}
```

Dans cette classe, vous devez redéfinir les méthodes « onCreate(SQLiteDatabase MaBase) » et « onUpgrade(SQLiteDatabase MaBase) ». L'argument représente votre base.

La méthode « onCreate » est appelée pour la création de la base si elle n'existe pas.

```
public void onCreate(SQLiteDatabase database) {
    database.execSQL(DATABASE_CREATE);
}
```

La variable « *DATABASE_CREATE* » va contenir la requête « SQL » qui permet de créer la base.

La méthode « onUpgrade » est appelée pour mettre à jour la version de votre base. Elle vous permet de mettre à jour le schéma de votre base.

```
public void onUpgrade(SQLiteDatabase db, int oldVersion,
                                                    int newVersion) {
    db.execSQL("DROP TABLE IF EXISTS " + "comments");
    onCreate(db);
}
```

Il est préférable de créer une classe par table. Cette classe va définir les méthodes « onCreate » et « onUpgrade ». Vous allégez ainsi le code de la classe qui dérive de « SQLiteOpenHelper ».

- Le fichier « CommentsDataSource.java » contient la classe contrôleur. Elle contient les différentes méthodes qui vont interagir avec la base de données. C'est le DAO (Data Access Object)

La classe « SQLiteOpenHelper » fournit les deux méthodes « getReadableDatabase() » et « getWriteableDatabase() » pour accéder à une instance « SQLiteDatabase » en mode de lecture ou écriture.

Ouverture de la base :

```
public void open() throws SQLException {
    database = dbHelper.getWritableDatabase();
}
```

Fermeture de la base :

```
public void close() {
    dbHelper.close();
}
```

Insérer un élément :

Pour insérer un élément dans la base, il faut d'abord former l'enregistrement. On utilise pour cela un objet du type « ContentValues » qui représente une collection de champs.

```
public long createComment(String comment) {
    ContentValues values = new ContentValues();
    values.put(MySQLiteHelper.COLUMN_COMMENT, comment);
    long insertId = database.insert(MySQLiteHelper.TABLE_COMMENTS,
                                                    null, values);
    return insertId;
}
```

Effacer un élément :

```
public void deleteComment(Comment comment) {
    long id = comment.getId();
    database.delete(MySQLiteHelper.TABLE_COMMENTS,
                MySQLiteHelper.COLUMN_ID     + " = " + id, null);
}
```

Faire une sélection :

```
    Cursor cursor = database.query(MySQLiteHelper.TABLE_COMMENTS,
            allColumns, MySQLiteHelper.COLUMN_ID + " = " + insertId,
                                            null, null, null, null);
```

La méthode « query » retourne une instance de « Cursor » qui représente un ensemble de résultats.

- Le fichier « `TestDatabaseActivity.java` » contient l'activité associée à notre application.

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_testdatabase);

    datasource = new CommentsDataSource(this);
    datasource.open();

}

protected void onResume() {
    super.onResume();
    datasource.open();
}

protected void onPause() {
    super.onPause();
    datasource.close();
}
```

Accès à la base de données SQLite

Le SDK d'Android inclut un programme permettant de lire une base de données SQLite.

Nous devons extraire le fichier de l'émulateur via la commande « adb/shell/pull » en tant que « root » ou bien en utilisant la vue « Device File Explorer », puis la commande « pull » :

```
adb pull
  /data/data/ca.umontreal.iro.ift1155.testdatabaseactivit
y/databases/comments.db
```

```
C:> sqlite3 comments.db
SQLite version 3.9.2 2015-11-02 18:31:45
Enter ".help" for usage hints.
sqlite> .tables
android_metadata  comments
sqlite> select * from comments;
1|Hate it
2|Cool
3|Very nice
4|Hate it
5|Very nice
6|Cool
sqlite> .exit
```

En tant que « root », il est possible aussi d'y avoir accès par la commande « adb » comme suit :

```
C:> adb shell
generic_x86_64:/ #
generic_x86_64:/                    #                    cd
/data/data/ca.umontreal.iro.ift1155.testdatabaseactivity
/databases/
generic_x86_64:/data/data/
ca.umontreal.iro.ift1155.testdatabaseactivity/databases
# sqlite3 comments.db
SQLite version 3.9.2 2015-11-02 18:31:45
Enter ".help" for usage hints.
Sqlite>.exit
```

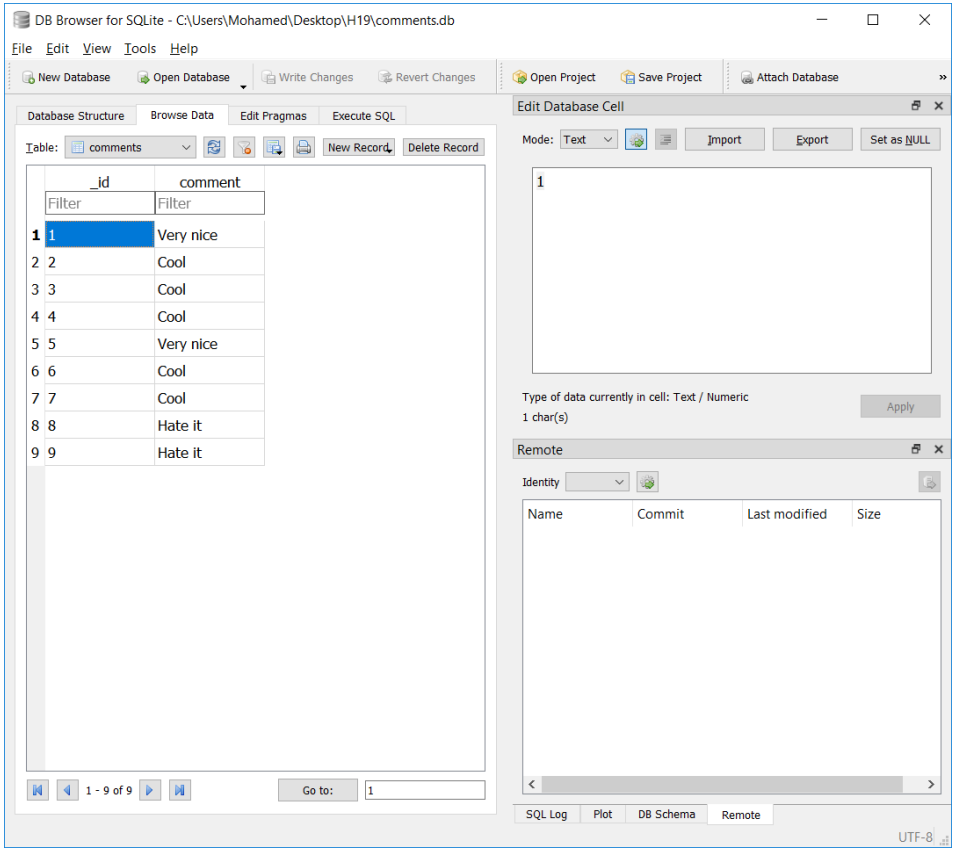
On peut utiliser un adds-on dans Firefox pour accéder à la base de données :

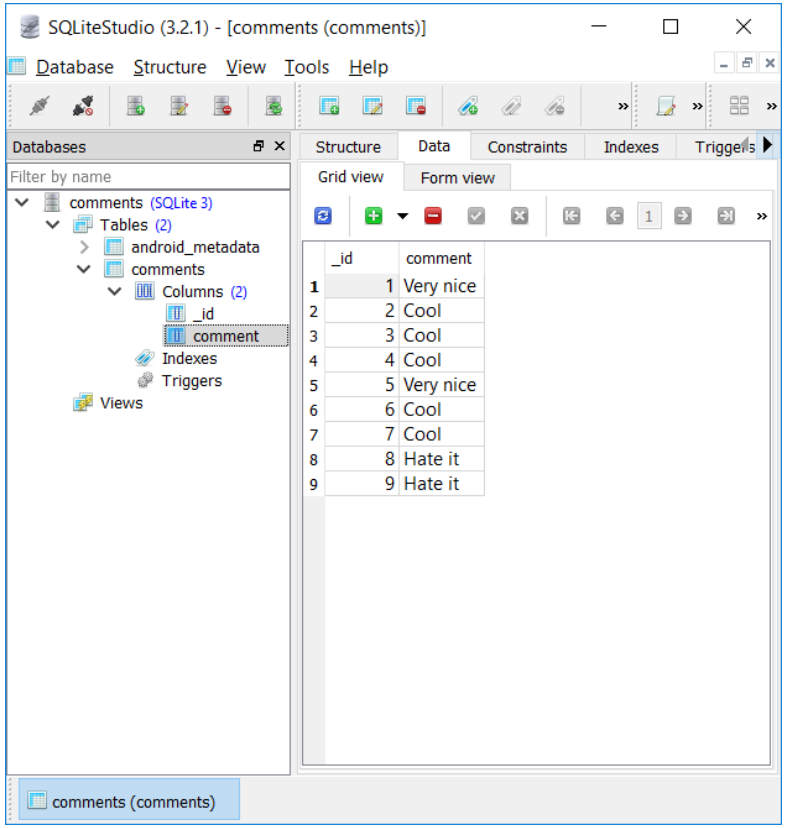
https://addons.mozilla.org/en-US/firefox/addon/sqlite-manager-webext/

Comme il est possible aussi d'utiliser un de ces deux logiciels :

https://sqlitebrowser.org/

https://sqlitestudio.pl/index.rvt


DB Browser for SQLite - C:\Users\Mohamed\Desktop\H19\comments.db
File Edit View Tools Help
New Database
Open Database
Write Changes
Revert Changes
Open Project
Save Project
Attach Database
Database Structure
Browse Data
Edit Pragmas
Execute SQL
Table: comments
New Record
Delete Record
_id
comment
Filter
1 1 Very nice
2 2 Cool
3 3 Cool
4 4 Cool
5 5 Very nice
6 6 Cool
7 7 Cool
8 8 Hate it
9 9 Hate it
1 - 9 of 9
Go to: 1
Edit Database Cell
Mode: Text
Import
Export
Set as NULL
1
Type of data currently in cell: Text / Numeric
1 char(s)
Apply
Remote
Identity
Name
Commit
Last modified
Size
SQL Log
Plot
DB Schema
Remote
UTF-8


SQLiteStudio (3.2.1) - [comments (comments)]
Database Structure View Tools Help
Databases
Filter by name
comments (SQLite 3)
Tables (2)
android_metadata
comments
Columns (2)
_id
comment
Indexes
Triggers
Views
Structure
Data
Constraints
Indexes
Triggers
Grid view
Form view
_id
comment
1 1 Very nice
2 2 Cool
3 3 Cool
4 4 Cool
5 5 Very nice
6 6 Cool
7 7 Cool
8 8 Hate it
9 9 Hate it
comments (comments)