Android Partie 4

Persistance

Et

Communication

- Différentes manières existent pour stocker des données localement :
  - Les préférences

  Servent normalement à stocker les paramètres de l'utilisateur.
  - Les fichiers

  Possibilité de les stocker en interne ou en externe.
  - Les bases de données

  SQLite est un moteur de base de données, basé sur SQL, léger et rapide, directement intégré au *Runtime* d'Android.

La persistance

⇨ les préférences (1)

- Les préférences permettent de stocker des données sous la forme *clé / valeur.*
- Les données des préférences sont limitées aux types de base et aux chaînes de caractères.
- Elles sont effacées si l'application est désinstallée.
- Elles sont stockées dans un fichier.
- Pour récupérer un fichier de préférences, deux méthodes de l'objet ***SharedPreferences*** sont utilisables :
  - *getSharedPreferences()* :
  - *getPreferences()*

La persistance

⇨ les préférences (2)

- *getSharedPreferences()* :
  - ramène les fichiers de préférence de l'application. Il faut passer en paramètre le nom du fichier préférence que l'on veut récupérer et le mode (on utilisera Context.MODE_PRIVATE).

- *getPreferences()* :
  - ramène le fichier de préférence de l'activité (ce fichier a pour nom le nom de l'activité)

La persistance

⇨ les préférences (3)

- Une fois que l'on a récupéré le fichier de préférences, on peut stocker des données dedans en opérant comme suit :
  - On récupère un objet ***SharedPreferences.Editor*** en appelant la méthode ***edit()*** du SharedPreferences.
  - On ajoute des valeurs via l'objet Editor en utilisant des méthodes put (***putBoolean(), putInt(), putFloat(), putString()*** …). Toutes ces méthodes ***put()*** ont en paramètre la clé (en String) et la valeur.
  - On Sauvegarde ces nouvelles valeurs via la méthode ***commit()*** ou ***apply().***

La persistance

⇨ les préférences (4)

- Des valeurs peuvent être supprimées du fichier de préférence par les méthodes :
  - ***remove()*** (pour une clé donnée)
  - ***clear()*** (suppression de toutes les clés).
- Pour lire les données d'un SharedPreferences, utiliser les méthodes du type de la donnée :
  - ***getBoolean()***
  - ***getInt()***
  - ***getString()***

Toutes ces méthodes *getXX()* ont en paramètre la clé (en String) et une valeur par défaut qui sera renvoyée si la clé n'est pas trouvée.

La persistance

⇨ les préférences (5)

- Exemple de lecture de la clé nom dans le fichier CDIAfpa :

```
private static final String MON_PREF = "CDIAfpa"

SharedPreferences mesPrefs = getSharedPreferences(
                                    MON_PREF,
                                    Context.MODE_PRIVATE);
String monNom = mesPref.getString("nom", "inconnu");
```

- Le fichier des préférences est en fait un fichier xml rangé dans le dossier :
  - data/data/nomDuPackagAppli/
    shared_prefs/CDIAfpa.xml

La persistance

⇨ les préférences (5)

- Exemple d'écriture de la clé nom dans le fichier CDIAfpa :

```
private static final String MON_PREF = "CDIAfpa"
private String monNom = "afpa CDI"

SharedPreferences mesPrefs = getSharedPreferences(
                                    MON_PREF,
                                    Context.MODE_PRIVATE);

Editor monEdit = mesPrefs.edit();

monEdit.putString("nom", monNom);

monEdit.commit();
```

commit() est synchrone, apply() est asynchrone

La base de données SQLite

⇨ Introduction (1)

- Abdroid intègre la gestion de bases de données SQLite.
- Le fichier correspondant est stocké dans le smartphone sous :

  data/data/NomPackageAppli/NomBaseDeDonnees

  (répertoire accessible sur un AVD et protégé sur un smartphone)

- Les requêtes sur la base se font dans le même processus que l'application.
- La base n'est accessible que par son application.

La base de données SQLite

⇨ introduction (2)

- La base est modélisée par la classe :
- android.database.sqlite.SQLiteDatabase
- Elle fournit les méthodes pour la gestion des données :
  - insert() insertion de données
  - update() mise à jour
  - delete() suppression
  - query() requête SQL SELECT
  - execSQL() requête SQL ne retournant pas de données.

La base de données SQLite

⇨ la classe SQLiteOpenHelper (1)

- Pour créer ou mettre à jour une base, on utilise une classe « SQLiteOpenHelper ».

  android.database.sqlite.SQLiteOpenHelper

- On crée une classe qui étende cette dernière

  public classe MonOpenHelper extends SQLiteOpenHelper { … }

avec un constructeur dont les paramètres sont :

public MonOpenHelper(Context cont, String nomBase, SQLiteDatabase.CursorFactory fact, int version)

La base de données SQLite

⇨ la classe SQLiteOpenHelper (2)

- cont : le context de l'application
- nomBase : le nom de la base
- fact : un curseur pour la base (généralement à null)
- version : un entier correspondant à la version

(on se contentera d'appeler le constructeur de la classe mère)

- Dans notre activité, on instancie notre classe

```
SQLiteOpenHelper monHelper = new
                                MonOpenHelper( … );
```

- A partir de cet objet, on appelle la méthode d'ouverture de la base :

```
monHelper.getXXXableDatabase();
```

La base de données SQLite

⇨ la classe SQLiteOpenHelper (3)

avec XXX qui vaut :

- write pour ouverture en lecture / écriture
- read pour ouverture en lecture seule

```
public class MonOpenHelper extends SQLiteOpenHelper {


   public MonOpenHelper(Context context,
                        String name,
                        SQLiteDatabase.CursorFactory factory,
                        int version) {
      super(context, name, factory, version);
   }
```

La base de données SQLite

⇨ la classe SQLiteOpenHelper (4)

- Des méthodes de notre classe MonOpenHelper seront automatiquement appelées :
  - public void onCreate(SQLiteDatabase db)

  Uniquement appelée si la base de données n'existe pas.

  C'est dans cette méthode que l'on met le cade de création de la base.
  - public void onUpgrade(SQLiteDatabase db,

    int ancienneVersion, int nouvelleVersion)

  Appelée si le numéro de version de la base de données à changé.

La base de données SQLite

⇨ la classe SQLiteOpenHelper (5)

- La classe MonOpenHelper :

```
public static final String CREER_TABLE = "CREATE TABLE " + NOM_TABLE +
        " (" + PK_PEL + " INTEGER PRIMARY KEY AUTOINCREMENT, " +
        NOM + " TEXT, " + DATE + " DATE, " + REF + " TEXT, " +
        NB_POSE + " TEXT, " + REG_ISO + " TEXT);";

   public static final String DROP_TABLE = "DROP TABLE IF EXISTS
           + NOM_TABLE + ";";

@Override
   public void onCreate(SQLiteDatabase db) {

      db.execSQL(PelliculeDAO.CREER_TABLE);
   }

   @Override
   public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
      db.execSQL(PelliculeDAO.DROP_TABLE);
      this.onCreate(db);
   }
```

La base de données SQLite

⇨ la classe SQLiteOpenHelper (6)

- La classe Activity :

```
public class CreerPelActivity extends ActionBarActivity {
    static final int VERSION = 1;
    static final String NOM_BASE = "memoPel.db";
    private SQLiteDatabase memoPelDB = null;
    private SQLiteOpenHelper memoPelOpenHelper = null;

  memoPelOpenHelper = new MonOpenHelper(this, NOM_BASE, null, VERSION);

  open();
       // traitement …
  close();

  private void open(){
      memoPelDB = memoPelOpenHelper.getWritableDatabase();
  }
  private void close(){
     memoPelDB.close();
  }
}
```

La base de données SQLite

⇨ les ContentValues(1)

- Pour insérer des données dans une table, il faut utiliser la méthode :

```
public long insert(String nomTable,
                   String nullColonne,
                   ContentValues valeurs)
```

- L'objet *valeurs* de la classe *ContentValues*

est un ensemble de « clés / valeurs »

- Pour ajouter un couple on utilise sur cet objet la méthode :

put(String nomColonne, valeur) où

nomColonne est le nom du champ de la table

La base de données SQLite

⇨ les ContentValues(2)

- Exemple d'insertion :

```
ContentValues pelDB = new ContentValues();
      pelDB.put(NOM,pel.getNom());
      pelDB.put(DATE, pel.getDate().getTime());
      pelDB.put(REF,pel.getReference());

open()
memoPelDB.insert(PelliculeDAO.NOM_TABLE, null, pelDB);
close();
```

- La méthode insert() retourne le numéro de la ligne insérée ou -1 en cas d'erreur.

La base de données SQLite

⇨ les Cursors(1)

- Récupération de données
- 2 possiblités :
  - public Cursor query(String table,String[] colonnes, String clauseWhere, String[] ArgsSelect, String GroupBy, String having, String orderBy)
  - public Cursor rawQuery(String sql, String[] argsSeleect)

La base de données SQLite

⇨ les Cursors(2)

Un Cursor représente un ensemble d'enregistrements contenant le résultat de la requête sql.

- Connaître le nombre d'enregistrements :
  - public int getCount()
- Positionner le cursor au début
  - public boolean moveToFirst()

  (retourne false si le cursor est vide)
- Tester si il y a un autre enregistrement :
  - public boolean moveToNext()

  (retourne false si plus d'enreg)

La base de données SQLite

⇨ les Cursors(3)

- Fermer le cursor
  - public abstract void close ()
- Récupérer l'index de la colonne :
  - public abstract int getColumnIndex(String nomColonne)
- Récupérer la valeur d'une colonne :
  - public XXX getXXX(int indexColonne)
  - XXX est le type de la donnée :
    - String, Short, Int, Long, Float, Double

La base de données SQLite

⇨ Exemple

```
public ArrayList<Pellicule> getAllPels(){
        ArrayList<Pellicule> pels = new ArrayList<Pellicule>();
        // création de la requête SQL
        String sql = "SELECT * FROM " + PelliculeDAO.NOM_TABLE;
        open();     /// ouverture de la base de données
        // exécution de la requete
        Cursor curseur = memoPelDB.rawQuery(sql, null);
        // remplissage de l'arraylist
        Pellicule pel = null;
        if (curseur.moveToFirst()) {
            do {
                pel = new Pellicule();
                pel.setIdPel(Long.parseLong(curseur.getString(0)));
                pel.setNom(curseur.getString(1));
                pels.add(pel);
            }while(curseur.moveToNext());
        }
        curseur.close();
        Log.d("getAllPels()", pels.toString());
        close();    // fermeture de la base
        return pels;
    }
```

La tâche asynchrone

⇨ introduction

- Android gère l'interface graphique dans une thread unique : la UI_Thread.
- Pour gérer un traitement long, il faut le faire dans une autre thread.
- Android propose une classe générique et abstraite qui prend tout en charge :

android.os.AsyncTask<Params, Progress, Result>

Params : type des paramètres pour la tâche

Progress : type de paramètres reçus par la méthode de mise à jour

Result : type du résultat reçus en fin de tâche

La tâche asynchrone

⇨ les méthodes (1)

- Cette classe définit 4 méthodes qui font le lien entre la ui_thread et ce nouveau thread.
- 3 d'entre elles s'éxécutent dans l'ui_thread :
  - protected void onPreExecute()
  - protected void onProgressUpdate(Progress… values)
  - protected void onPostExecute(Result result)
- 1 dans l'autre thread :
  - protected abstract Result doInBackground(Params… params)

La tâche asynchrone

⇨ les méthodes (2)

- onPreExecute() :
  - Est appelée avant le lancement de la tâche d'arrière plan.
- doInBackround() :
  - Est lancée après onPreExecute(), les paramètres de type Params sont passés à cette méthode.
  - C'est notre traitement.
  - Elle peut lancer la méthode publishProgress(Progress…) qui déclenche onProgressUpdate de l'ui_thread en lui passant les paramètres de type Progress.

La tâche asynchrone

⇨ les méthodes (3)

- onPostExecute() :
  - Est lancée après la fin de la tâche d'arrière plan.
  - Elle récupère le paramètre de type Result donné par la tâche d'arrière plan.
- Pour utiliser cette classe, il faut créer une sous classe qui étend AsyncTask et appeler la méthode :

execute(Params)

sur une instance de cette sous classe.

La tâche asynchrone

⇨ Exemple

- La sous classe est une classe interne à la classe d'activité :

```
public class ListingDesStationsActivity extends Activity {

    private ListView listing;
    private StationsParser sp;
    private ProgressDialog progress;
    private StationsAdapter leStationsAdapter;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_listing_des_stations);

        listing = (ListView) findViewById(R.id.listViewFiltreStations);
```

- La sous classe :

```
class ChargementDesStationsTache extends AsyncTask<Void, Void, Void> {

    protected void onPreExecute() {
        progress = ProgressDialog.show(ListingDesStationsActivity.this,
        getResources().getString(R.string.app_name),
        getResources().getString(R.string.chargement_message),
        true);
    }

    @Override
    protected void doInBackground(void... params) {
        // charger les stations
        try {
            sp = new StationsParser();
             } catch (Exception e) { e.printStackTrace();}
        leStationsAdapter = new StationsAdapter(getBaseContext(), sp.getArrList());
        return null;
    }
```

La tâche asynchrone

⇨ Exemple

- La sous classe :

```
    protected void onProgessUpdate(Void... aAfficher) {  }

    protected void onPostExecute(Void result) {
        // arreter le progressDialog
        progress.dismiss();

        // mettre a jour la ListView des stations
        listing.setAdapter(leStationsAdapter);
    }
}    // fin de la sous classe

new ChargementDesStationsTache().execute();
}    // fin onCreate()
}
```

La communication

⇨ accès à un web service REST

- REST : REpresentational State Transfert
- Architecture qui propose une alternative à SOAP et qui est orientée ressources.
- Elles sont identifiées par des URI (Uniform Resource Identifier)
- Les ressources sur le web sont le plus souvent au format html, xml, json
- Accès par le protocole HTTP (GET ou POST)

La communication

⇨ accès à un web service REST

- L'appel du service REST s'effectue par :

```
DefaultHttpClient client = new
                                    DefaultHttpClient();
```

ou

```
AndroidHttpClient client = new
    AndroidHttpClient().newInstance("Android");
```

- On définit une requête à envoyer :
  - en GET ou en POST

- En GET

HttpGet requete = new
HttpGet("uri?param1=val1&param2=val2");

On exécute la requête :

```
HttpResponse response = client.execute(requete);

if(response.getStatusLine().getStatusCode()>=200
        &&
response.getStatusLine().getStatusCode()<300) {
…… }
```

- En POST

  On définit les paramètres dans une liste de paires nom/valeur :

```
List<NameValuePair> params = new
                        ArrayList<NameValuePair>()

params.add(new BasicNameValuePair("nom",
                                          valeur);
```

  On définit une requête :

```
URI uri = URIUtils.createURI("http", host, -1, path,
URLEncodedUtils.format(params, "UTF-8"), null);

HttpPost requete = new HttpPost(uri);
```

On récupère la réponse :

```
ResponseHandler<String> response = new
                                BasicResponseHandler();

String retourJson = client.execute(requete,
response);

// traitement du retour Json

JSONObject objJson = new JSONObject(retourJson);
```

- Attention : il faut déclarer la permission d'accès à internet dans le manifest.xml

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