P5 : Gestion des données

1. Les préférences
2. Stockage de fichiers
3. La base de données locale au terminal
4. Utiliser les threads d'arrière-plan
5. La consommation d'un web service (librairie Gson)
6. TP: Développement d'une application qui affiche les données d'un web service

1 – Les préférences

Les préférences utilisateur sont enregistrées dans un espace spécifique pour chaque application.

C'est une méthode de stockage simplifiée qui permet uniquement de mémoriser des types simples et il n'existe aucune relation entre les données enregistrées.

Il existe des préférences privées au sein d'une activité et des préférences partagées au sein d'une application.

1 – Les préférences

Les préférences privées au sein d'une activité :

Lecture :

```
SharedPreferences settings = getPreferences(Context.MODE_PRIVATE);
String prefCle = settings.getString("cle", "valeur");
```

Enregistrement :

```
SharedPreferences settings = getPreferences(Context.MODE_PRIVATE);

SharedPreferences.Editor editor = settings.edit();
editor.putString("cle", "la nouvelle valeur");

editor.commit();
```

1 – Les préférences

Les préférences partagées au sein d'une application :

A noter qu'il est impossible de partager des préférences entre plusieurs activités.

Lecture :

```
SharedPreferences settings = getSharedPreferences("Preference", 0);
String prefCle = settings.getString("cle", "valeur");
```

Enregistrement :

```
SharedPreferences settings = getSharedPreferences("Preference", 0);

SharedPreferences.Editor editor = settings.edit();
editor.putString("cle", "la nouvelle valeur");

editor.commit();
```

1 – Les préférences

Le « PreferenceManager » permet également d'utiliser les préférences partagées.

```
public class Preference {

    private static final String PREF_CITY = "PREF_CITY";

    private static SharedPreferences get(Context context) {
        return PreferenceManager.getDefaultSharedPreferences(context);
    }

    public static String getCity(Context context) { // pour récupérer la Ville

        return get(context).getString(PREF_CITY, "");
    }

    public static void setCity(Context context, String city) { // enregistrement
        get(context).edit().putString(PREF_CITY, city).commit();
    }
}
```

2 – Le stockage de fichiers

Il existe deux types de support physique pour stocker des fichiers.

La mémoire interne (système) :

/data/data/package/files/

Accessible dans une certaine limite. Il est en effet possible d'écrire dans un espace restreint un contenu dans le dossier d'installation de l'application. Les données écrites seront accessibles uniquement par l'application.

La mémoire externe (sdcard) :

/sdcard/Android/data/package/files/

Il faut vérifier si elle est accessible (sdcard non présente ou utilisée en mode stockage de masse).

2 – Le stockage de fichiers

Enregistrement dans la mémoire interne (système) :

Pour écrire utilisez la méthode « openFileOutput ».

```
static public void writeStringToFile(Context context, String content, String file) throws IOException {
    try {
        FileOutputStream fos = context.openFileOutput(file, Context.MODE_PRIVATE);
        fos.write(content.toString().getBytes());
        fos.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

2 – Le stockage de fichiers

Lecture depuis la mémoire interne (système) :

Pour lire utilisez la méthode « openFileInput ».

```
static public String readStringFromFile(Context context, String file) throws IOException {
    String data = null;

    try {
        BufferedReader inputReader = new BufferedReader(new InputStreamReader(context.openFileInput(file)));
        String inputString;
        StringBuffer stringBuffer = new StringBuffer();

        while ((inputString = inputReader.readLine()) != null) {
            stringBuffer.append(inputString);
        }
        data = stringBuffer.toString();
    } catch (Exception e) {
        e.printStackTrace();
    }

    return data;
}
```

2 – Le stockage de fichiers

Vérifier l'accessibilité de la mémoire externe (sdcard) :

```
boolean mExternalStorageAvailable = false;
boolean mExternalStorageWriteable = false;
String state = Environment.getExternalStorageState();

if (Environment.MEDIA_MOUNTED.equals(state)) {
    // We can read and write the media
    mExternalStorageAvailable = mExternalStorageWriteable = true;
} else if (Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
    // We can only read the media
    mExternalStorageAvailable = true;
    mExternalStorageWriteable = false;
} else {
    // Something else is wrong. It may be one of many other states, but all we need
    //  to know is we can neither read nor write
    mExternalStorageAvailable = mExternalStorageWriteable = false;
}
```

2 – Le stockage de fichiers

TP « Créer un fichier » :

- vérifiez si le dossier « Android/data/nom.package » existe
- créez le dossier « mkdirs ».
- créez le fichier

A noter :

- Il ne faut pas oublier les permissions
- Pour récupérer le nom du package utilisez « getPackageName() ».

3 – La base de données locale au terminal

La base de données relationnelle SQLite est disponible en natif pour Android.

Toutes les applications peuvent utiliser SQLite.

SQLite est souvent utilisé dans les systèmes embarqués car il allie simplicité et mémoire légère.

Elle permet de lire et stocker des données en locale au travers du langage SQL (www.sqlite.org).

3 – La base de données locale au terminal

Créer une base de données :

Il faut utiliser une redéfinition de la classe SQLiteOpenHelper.

Le constructeur a besoin en paramètre du context(Activity), d'un nom et d'un numéro de version.

La méthode onCreate() nous donnera un objet SQLiteDatabase à peupler.

La méthode onUpgrade() permettra de définir le comportement à adopter si la version de la base change.

3 – La base de données locale au terminal

Créer une base de données (constructeur) :

```
private Context c;
public static final String DB_NAME="cnav";

private static SQLiteDatabase db;

private Database(Context c) {
    super(c, DB_NAME, null, 1);
    this.c = c;
}
```

3 – La base de données locale au terminal

Créer une table (onCreate) :

```
public synchronized void onCreate() {
    db.execSQL("CREATE TABLE IF NOT EXISTS "+TBL_SALARIE_STATS+" (" +
            STATS_IDENTIFIANT+" VARCHAR," +
            STATS_DATE_DEBUT+" INTEGER," +
            STATS_DATE_FIN+" INTEGER ," +
            STATS_REPONSE_CORRECTE+" INTEGER ," +
            STATS_REPONSE_INCORRECTE+" INTEGER," +
            STATS_QUIZZ_DEMARRE+" INTEGER ," +
            STATS_QUIZZ_ABANDONNE+" INTEGER ," +
            STATS_CATEGORIE+" VARCHAR," +
            STATS_QUESTION+" VARCHAR," +
            STATS_KEY+" VARCHAR" +
            ");" );
}
```

3 – La base de données locale au terminal

Ouvrir une connexion :

Deux méthodes sont possibles :

- getReadableDatabase() pour ouvrir la base en lecture seule
- getWritableDatabase() pour ouvrir la base en lecture et écriture

Ces deux méthodes nous fournissent un objet SQLiteDatabase qui nous permettra de faire des requêtes.

3 – La base de données locale au terminal

Ouvrir une connexion :

```
private static void open() {
    if (db == null) {
        db = mInstance.getWritableDatabase();
    }
}

public synchronized void closeConnecion() {
    if (mInstance != null) {
        mInstance.close();
        db.close();
        mInstance = null;
        db = null;
    }
}
```

3 – La base de données locale au terminal

Exécuter une requête (select) :

```
synchronized public Cursor selectData() {
    Cursor c = db.rawQuery("SELECT * FROM table WHERE _id = 1", null);
    return c;
}
```

Utilisation d'un Curseur :

```
Cursor cPrincipal = Database.getInstance(this).selectData();

if(cPrincipal.getCount() > 0) {

    if(cPrincipal.moveToFirst()) {

        while(!cPrincipal.isAfterLast()) {

            Log.e("select", cPrincipal.getString(cPrincipal.getColumnIndex("colonne1")));

            cPrincipal.moveToNext();
        }
    }
}
```

3 – La base de données locale au terminal

Il est également possible d'exécuter une requête (insert, update, delete) avec les méthodes fournies par SQLiteDatabase.

```
public synchronized long insert(String table, ContentValues values) {
    return db.insert(table, null, values);
}
```

On utilisera l'objet ContentValues en le passant en argument.

```
ContentValues values = new ContentValues();
values.put("colonne1", "valeur 1");
values.put("colonne2", "valeur 2");

DataBase.getInstance(ApplicationService.this).insert("la_table", values);
```

3 – La base de données locale au terminal

Update :

```
synchronized public long update(String table) {
    ContentValues values = new ContentValues();
    values.put("colonne1", "valeur 1");
    values.put("colonne2", "valeur 2");

    long i = db.update(table, values, "_id = 1", null);

    return i;
}
```

Delete :

```
synchronized public int delete(String table) {
    return db.delete(table, "_id = 1", null);
}
```

3 – La base de données locale au terminal

TP :

- Créer une base de donnée
- Table « liste_musique » (_id, categorie, nom)
- Créer une formulaire pour ajouter et modifier les musiques
- Affiche une liste avec les informations de la table « liste_musique »

4 – Utiliser les threads d'arrière-plan

Pour garantir la réactivité de vos applications, il est conseillé de déplacer toutes les opérations longues hors du thread principal vers un thread enfant.

Android propose 2 possibilités :

- La classe AsyncTask : permet de définir une opération exécutée en arrière plan et fournit des gestionnaires d'événements.

- La classe Handler : permet d'implémenter vos propre threads pour la synchronisation avec le thread principal de l'interface.

4 – Utiliser les threads d'arrière-plan

La classe AsyncTask :

- Permet d'effectuer des opérations de base et d'envoyer le résultat sur l'interface graphique.
- Doit être utilisé pour des opérations de courte durée.
- Possibilité de définir du code qui sera effectué « avant », « pendant » et « après »

4 – Utiliser les threads d'arrière-plan

La classe AsyncTask :

```
private class LoginAsyncTask extends AsyncTask<Void, Void, Void> {

    @Override
    protected void onPreExecute() {

        super.onPreExecute();
    }

    @Override
    protected Void doInBackground(Void... params) {

        return null;
    }

    @Override
    protected void onPostExecute(Void result) {

        super.onPostExecute(result);
    }

}
```

Exécution : `new LoginAsyncTask().execute();`

4 – Utiliser les threads d'arrière-plan

La classe Handler :

- Associé au thread principal de l'application.
- Peut être utilisé pour des opérations longues.
- Il permet d'envoyer des messages vers le thread principal.

4 – Utiliser les threads d'arrière-plan

Classe Handler :

```
BackgroundThread backgroundThread;
TextView myText;
boolean myTextOn = true;

public class BackgroundThread extends Thread {
    boolean running = false;

    void setRunning(boolean b){
        running = b;
    }

    @Override
    public void run() {
        while(running){
            // TODO

            handler.sendMessage(handler.obtainMessage());
        }
    }
}

Handler handler = new Handler(){
    @Override
    public void handleMessage(Message msg) {
        // TODO
    }
};
```

Exécution :

```
backgroundThread = new BackgroundThread();
backgroundThread.setRunning(true);
backgroundThread.start();
```

5 – La consommation d'un web service …

Nous allons utiliser une librairie externe.

La librairie Gson (Gson-2.6.jar) est disponible sur http://code.google.com/p/google-gson/

Il faut copier le fichier dans le dossier « libs ».

5 – La consommation d'un web service ...

1) Vérifier l'état du réseau.

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

```
public static boolean isNetworkAvailable(Context context) {

    ConnectivityManager connectivity = (ConnectivityManager) context
            .getSystemService(Context.CONNECTIVITY_SERVICE);

    if (connectivity != null) {
        NetworkInfo[] info = connectivity.getAllNetworkInfo();
        if (info != null) {
            for (int i = 0; i < info.length; i++) {
                if (info[i].getState() == NetworkInfo.State.CONNECTED) {
                    return true;
                }
            }
        }
    }
    return false;
}
```

5 – La consommation d'un web service ...

2) Réception de données :

Requête « get » :

```
// requete HTTP
HttpParams httpParameters = new BasicHttpParams();

HttpConnectionParams.setConnectionTimeout(httpParameters, 10000);
HttpConnectionParams.setSoTimeout(httpParameters, 10000);

HttpClient httpclient = new DefaultHttpClient(httpParameters);
URI uri = new URI("http://www.monsite.com/webservice");
HttpGet httpGet = new HttpGet();
httpGet.setURI(uri);

HttpResponse response;

response = httpclient.execute(httpGet);
```

5 – La consommation d'un web service ...

3) Récupérer un objet :

La librairie GSON permet de récupérer un objet.

```
// GSON
GsonBuilder builder = new GsonBuilder();

Gson gson = builder.create();

HttpResponse streamData = response;
Reader reader = new InputStreamReader(streamData.getEntity().getContent());

MonObjet monObj = gson.fromJson(reader, MonObjet.class);

monObj.getPrenom();
```

5 – La consommation d'un web service ...

4) Envoi de données :

Requête « post » :

```
// nameValuePairs
List<NameValuePair> nameValuePairs = new ArrayList<NameValuePair>(1);
nameValuePairs.add(new BasicNameValuePair("prenom", "mathieu"));

// requete HTTP
HttpParams httpParameters = new BasicHttpParams();

httpParameters.setLongParameter(ConnManagerPNames.TIMEOUT, 20000);
HttpConnectionParams.setConnectionTimeout(httpParameters, 20000);

HttpConnectionParams.setSoTimeout(httpParameters, 30000);

HttpClient httpclient = new DefaultHttpClient(httpParameters);
HttpPost httppost = new HttpPost("http://www.monsite.com/webservice");
httppost.setEntity(new UrlEncodedFormEntity(nameValuePairs));

HttpResponse response = httpclient.execute(httppost);
```