P4B - Programmation Mobile Android **KOCAK Mikail**
M. Naegel Benoit 25 janvier 2019

Table des matières

1 Le comportement d'une application Android **2**

2 Structure d'un projet Android **2**

3 Le manifest Android **2**

4 Les layouts **2**

5 Persistance des données **2**
5.1 Préférences partagées . . . 2
5.1.1 Les différents modes . . . 2
5.1.2 Activité d'édition des préférences . . . 2

6 Les threads sur Android **3**
6.1 Gestion des threads sous Android . . . 3
6.2 Création de threads . . . 3
6.2.1 Modifier l'UI . . . 4
6.2.2 Tâche périodique . . . 4

7 La gestion de tâches asynchrones **5**

8 La gestion de tâches web asynchrones avec la librairie Volley **6**
8.1 Installation . . . 6
8.2 Exemple de requête JSON . . . 6
8.2.1 Ajout de l'autorisation INTERNET dans le manifest . . . 6
8.2.2 Définition de la queue associée à l'activité courante . . . 6
8.2.3 Lancement d'une requête HTTP . . . 6
8.2.4 Les différents types de requêtes . . . 7
8.2.5 Exemple complet . . . 7
8.3 Création de requêtes avec type personnalisé . . . 7

9 Les fragments **8**
9.1 Les différents types de fragments . . . 8
9.2 Implantation . . . 8
9.3 Exemple . . . 8
9.4 Insertion d'un fragment dans un layout . . . 9
9.4.1 De manière statique . . . 9
9.4.2 De manière dynamique . . . 9
9.5 La gestion de fragments . . . 10
9.5.1 Transactions sur les fragments . . . 10
9.6 S'adapter à la taille et l'orientation de l'appareil . . . 10

10 Les bases de données SQLite sur Android **10**
10.1 Les types . . . 10
10.2 Création d'une base de données . . . 10
10.3 Requêtes sur une base . . . 11
10.3.1 Insertions . . . 11
10.3.2 Requêtes . . . 11
10.3.3 Mises à jour et suppressions . . . 12

1 Le comportement d'une application Android

— Un utilisateur ne peut tuer une application ;
— Système *Viking Killer* qui garde en mémoire les applications non-utilisés et sont tués s'il y a plus assez de mémoire pour les garder ;
— C'est un système optimisé pour les systèmes à faible puissance et mémoire.

2 Structure d'un projet Android

— `build` fichiers compilés,
— `src/main/java` code source Java du projet,
— `src/main/res/drawable` ressource images.
— `src/main/res/values` chaînes de caractères et dimensions. Permet l'internationalisation.
— `src/main/AndroidManifest.xml` , le fichier qui décrit le contenu de l'application et liens entre les modules.

3 Le manifest Android

Les intents sont associés à une vue (points d'entrées).

4 Les layouts

— Les éléments graphiques héritent de la classe `View` ;
— Ces élèments peuvent être regroupés au sein d'un `ViewGroup` ;
— Des `ViewGroup` sont prédéfinis :
—

5 Persistance des données

— Persistance des données liées à une activité, tout contenu d'un widget possédant un identifiant persiste durant toute la durée de l'application ;
— On peut avoir un fichier associé à une application, c'est une **préférence partagée** utilisant un mécanisme de clé/ valeur (classe `Bundle`) ;
— Base de données native `SQLite` .

5.1 Préférences partagées

La classe `SharedPreferences` permet de gérer des paires clé/ valeurs associées à un activité ;

5.1.1 Les différents modes

— `MODE_PRIVATE` restreint l'accée au fichier créé à l'application ;
— `MODE_WORLD_READABLE` et `MODE_WORLD_WRITABLE` permettent d'autoriser toutes les applications à lire/ écrire le fichier.

5.1.2 Activité d'édition des préférences

L'activité `PreferenceActivity` permet la création d'une interface de modification de préférences.

Elle utilise les attributs suivants :
— `android:title` , le nom de la préférence ;
— `android:summary` , la description de la préférence ;

— `android:key` , la clé pour stockage de la préférence.

6 Les threads sur Android

On ne bloque jamais l'interface graphique. Toute tâche longue doit s'exécuter dans un thread à part. Les entrées/sorties doivent être faits dans un thread afin d'améliorer la réactivité de l'application.

Il y a donc deux règles :

1. Le thread principal (UI thread) ne doit pas être bloqué ;
2. Il est interdit d'agir sur l'interface utilisateur **en dehors** du thread principal.

6.1 Gestion des threads sous Android

L'API Android fournit trois méthodes pour cela :

— `Activity.runOnUiThread(Runnable)` ;

— `View.post(Runnable)` ;

— `View.postDelayed(Runnable, long)` , permet de retarder une tâche.

De plus, elle fournit les classes pour des tâches asynchrones : `Handler` et `ASyncTask <U, V, W>` .

En plus des classes Java (`Thread` , `Executor` , `TimerTask` , ...).

Note : s'il est nécessaire d'agir sur l'interface graphique à partir d'un autre thread, il faut le déléguer au thread principal pour exécution ultérieure.

6.2 Création de threads

On peut créer une classe dérivée de `Thread` ou implanter l'interface `Runnable` .

```
public interface Runnable {
    void run();
}
```

On peut soit étendre `Thread` dans l'implantation de `Runnable` , ou alors, utiliser l'implantation en tant que classe anonyme en la passant à `Thread` .

```
class myThread extends Thread {
    public void run() {
        // Code qui sera execute par le thread
    }
}

myThread t = new myThread();

t.join();
```

```
class myTask implement Runnable {
    public void run() {
        // Code qui sera execute par le thread
    }
}

myTask task = new myTask();
Thread t = new Thread(task);

t.join();
```

```
Thread t = new Thread(new Runnable() {
    public void run() {
        // Code qui sera execute par le thread
    }
});

t.join();
```

6.2.1 Modifier l'UI

Pour modifier une vue, on va donc procéder de cette manière :

```
Handler h = new Handler();
h.postDelayed(new Runnable() { // postDelayed ou post
    public void run() {
        bt.setText("");
    }
},1000);
```

6.2.2 Tâche périodique

```
Timer t = new Timer();
final Handler h = new Handler();

TimerTask task = new TimerTask() {
    int i = 0;
    public void run() {
        h.post(new Runnable() {
            public void run() {
                bt.setText(i.toString());
            }
        });
        i++;
    }
};

t.schedule(task, 0, 1000);
```

7 La gestion de tâches asynchrones

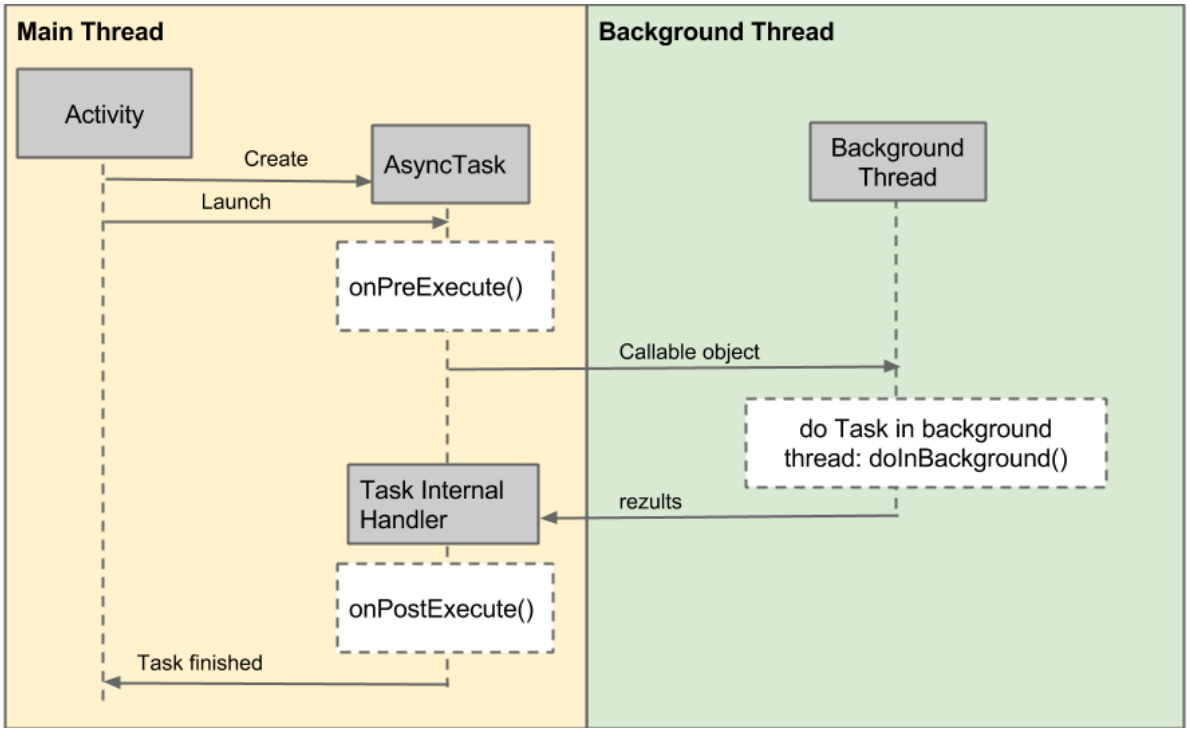

FIGURE 1 – La classe `AsyncTask`

```
1  private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
2      protected Long doInBackground(URL... urls) {
3          int count = urls.length;
4          long totalSize = 0;
5
6          for (int i = 0; i < count; i++) {
7              totalSize += Downloader.downloadFile(urls[i]);
8              publishProgress((int) ((i / (float) count) * 100));
9
10             // Escape early if cancel() is called
11             if (isCancelled()) break;
12         }
13
14         return totalSize;
15     }
16
17     protected void onProgressUpdate(Integer... progress) {
18         setProgressPercent(progress[0]);
19     }
20
21     protected void onPostExecute(Long result) {
22         // Note: pas bien, il faudrait associer un écouteur
23         // et passer directement le message.
24         showDialog("Downloaded " + result + " bytes");
25     }
26 }
27
28 new DownloadFilesTask().execute(url1, url2, url3);
```

8 La gestion de tâches web asynchrones avec la librairie Volley

La librairie Android *Volley* permet de faire des requêtes HTTP asynchrones de manière très largement simplifiée et bien plus performante.

8.1 Installation

Pour installer la librairie Volley, il suffit d'ajouter la dépendance `com.android.volley:volley:1.1.1` dans les dépendances du fichier `build.gradle` de votre projet ou application (`app/build.gradle` par exemple).

```
dependencies {
    ...
    implementation 'com.android.volley:volley:1.1.1'
}
```

8.2 Exemple de requête JSON

8.2.1 Ajout de l'autorisation INTERNET dans le manifest

Afin de connecter votre application à internet, il faut définir la permission dans le manifest. Pour cela, dans `AndroidManifest.xml` ajoutez :

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="com.example.tp3weather">
    <!-- ... -->
    <uses-permission android:name="android.permission.INTERNET"></uses-permission>
    <!-- ... -->
```

8.2.2 Définition de la queue associée à l'activité courante

Tout d'abord il faut définir un objet qui contiendra la queue des requêtes HTTP d'une activité ou d'un contexte donné. Pour cela, on ajoutera une propriété de type `RequestQueue` en important `com.android.volley.RequestQueue` dans notre activité. Puis, on la peuplera en faisant cet appel : `Volley.newRequestQueue(this);` lors de la création de l'activité.

8.2.3 Lancement d'une requête HTTP

Maintenant qu'on a défini une queue, on peut demander à *Volley* de faire une requête qui attend en réponse un objet JSON (`JSONObject` en Java). Pour cela, on fera l'appel suivant : `notreQueue.add(new JsonObjectRequest(url, null, listener, errorListener));` (qui est de `com.android.volley.toolbox`).

`listener` est une fonction qui prend en paramètre un objet de type `JSONObject` .

`errorListener` est une fonction **facultative** qui prend un objet `VolleyError` . Cet objet contient une propriété `networkResponse` qui est `null` si la requête a échouée, autrement, elle contient la réponse HTTP avec les propriétés suivantes :

`statusCode: int` Le code HTTP.

`notModified: bool` Vrai si la réponse est une HTTP 304 (Content Not Modified).

`headers: Map<String,String>` Les en-têtes de la réponses HTTP.

`data: byte[]` La réponse brute du serveur.

8.2.4 Les différents types de requêtes

La *toolbox* de la librarie Volley définie plusieurs méthodes en plus que celle de l'exemple précédent (JSON). Elles sont les suivantes :

`StringRequest` Une simple requête HTTP qui retourne le corps en une chaîne (`String`).

`JsonRequest` Attend et renvoie un `JSONObject` .

`ImageRequest` Attend et renvoie une image sous forme un objet `android.graphics.Bitmap` .

8.2.5 Exemple complet

```
String url = "http://my-json-feed";

JsonObjectRequest jsonObjectRequest = new JsonObjectRequest
       (Request.Method.GET, url, null, new Response.Listener<JSONObject>() {

    @Override
    public void onResponse(JSONObject response) {
        mTextView.setText("Response: " + response.toString());
    }
}, new Response.ErrorListener() {

    @Override
    public void onErrorResponse(VolleyError error) {
        // TODO: Handle error

    }
});

// Access the RequestQueue through your singleton class.
MySingleton.getInstance(this).addToRequestQueue(jsonObjectRequest);
```

8.3 Création de requêtes avec type personnalisé

On peut facilement définir des requêtes personnalisées, comme par exemple pour un Bitmap :

```
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;

import com.android.volley.NetworkResponse;
import com.android.volley.Request;
import com.android.volley.Response;
import com.android.volley.toolbox.HttpHeaderParser;

public class BitmapRequest extends Request<Bitmap> {
    private final Response.Listener<Bitmap> listener;

    public BitmapRequest(
           String url,
           Response.Listener<Bitmap> listener,
           Response.ErrorListener errorListener) {

        super(Method.GET, url, errorListener);
        this.listener = listener;
    }

    @Override
```

```
22      protected Response<Bitmap> parseNetworkResponse(NetworkResponse response) {
23          Bitmap bitmap = BitmapFactory.decodeByteArray(response.data, 0,
                response.data.length);
24          return Response.success(
25              bitmap, HttpHeaderParser.parseCacheHeaders(response));
26      }
27
28      @Override
29      protected void deliverResponse(Bitmap response) {
30          listener.onResponse(response);
31      }
32  }
```

9 Les fragments

Permet d'afficher plusieurs vues sur un écran. Ces fragments permettent une modularité, et donc, une réutilisabilité. On peut donc éviter de créer une activité par vue, l'utilisateur peut mieux naviguer ainsi.

Seul problème, ce n'est supporté qu'à partie de Honeycomp (Android 3.0, API 11). Il faut donc utiliser la librairie `android.support.v4.app.FragmentActivity` , pour être compatible avec les versions antérieures.

9.1 Les différents types de fragments

Il est possible d'utiliser les sous-classes suivantes de `Fragment` :

— `DialogFragment` , une fenêtre de dialogue flottante ;

— `ListFragment` , une liste d'éléments ;

— `PreferenceFragment` , une liste de préférences.

9.2 Implantation

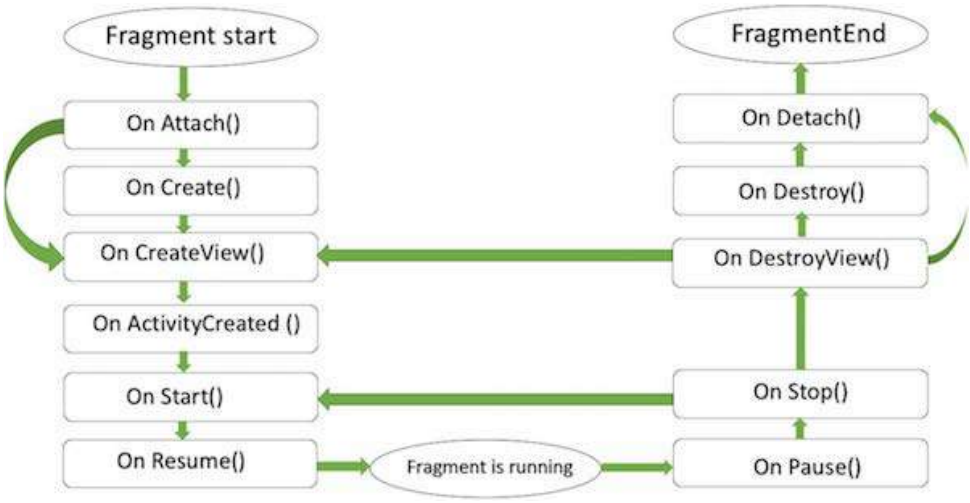

FIGURE 2 – Le cycle de vie de la classe `Fragment` .

9.3 Exemple

```
1   class TestFragment extends Fragment {
2       @Override
3       public void onCreate(Bundle savedInstanceState) {
```

```
        // TODO Auto-generated method stub
        super.onCreate(savedInstanceState);
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle
        savedInstanceState) {
        // TODO Auto-generated method stub
        return super.onCreateView(inflater, container, savedInstanceState);
    }

    @Override
    public void onPause() {
        // TODO Auto-generated method stub
        super.onPause();
    }
}
```

9.4 Insertion d'un fragment dans un layout

9.4.1 De manière statique

```
<LinearLayout
    android:id="@+id/linearLayout1"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_alignLeft="@+id/button2"
    android:layout_below="@+id/button2"
    android:layout_marginTop="54dp"
    android:orientation="vertical">
        <fragment android:id="@+id/fragment1"
            android:name="com.example.androidtp3.TestFragment"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_centerHorizontal="true"
            android:layout_centerVertical="true" />
</LinearLayout>
```

9.4.2 De manière dynamique

```
FragmentManager manager = getSupportFragmentManager();
FragmentTransaction transaction = manager.beginTransaction();
TestFragment fragment = new TestFragment();

transaction.add(R.id.linearLayout1, fragment);
transaction.commit();
```

Dans le cas d'un fragment sans UI, on lui donnera manuellement un identifiant nous permettant de le récupérer plus tard.

```
TestFragment fragment = new TestFragment();

transaction.add(fragment, "fragmentId");
transaction.commit();
```

9.5 La gestion de fragments

On utilise la classe `FragmentManager` pour gérer les fragments d'une activité. Pour récupérer le gestionnaire, on fera :

```
FragmentManager manager = getSupportFragmentManager()
```

On va ensuite pouvoir lister les fragments à l'aide des méthodes suivantes :

— `manager.findFragmentById()` , récupère les fragments de l'activité qui ont une UI (l'ID donnée est l'identifiant du container dans lequel est attaché le fragment) ;

— `manager.findFragmentByTag()` , récupère les fragments de l'activité qui n'ont pas d'IU (recherche par chaîne de caractères).

9.5.1 Transactions sur les fragments

Grâce au gestionnaire `FragmentManager` on va pouvoir commencer des transactions sur les fragments à l'aide de `FragmentTransaction` . Pour cela, on fera :

```
FragmentTransaction transaction = manager.beginTransaction();
// add(), remove(), replace(), ...
transaction.commit();
```

9.6 S'adapter à la taille et l'orientation de l'appareil

— `res/layout/main.xml` (défaut, recommandé pour les smartphones) ;

— `res/layout-normal/main.xml` ;

— `res/layout-large/main.xml` (tablettes) ;

— `res/layout-xlarge/main.xml` .

10 Les bases de données SQLite sur Android

10.1 Les types

— `NULL` ;

— `INTEGER` (1, 2, 4, 6 ou 8 octets) ;

— `REAL` (virgule flottante, 8 octets) ;

— `TEXT` ;

— `BLOB` (Binary Large OBject).

Attention ! Les clés étrangères ne sont pas supportées par ce SGBD.

10.2 Création d'une base de données

Pour créer une base de données, on utilise `SQLiteOpenHelper` et on surcharge la méthode `onCreate()` .

```
public class MyDataBase extends SQLiteOpenHelper {
    // En-têtes obligatoires
    private static final String DATABASE_NAME = "myDatabase.db";
    private static final int DATABASE_VERSION = 2; // => onUpgrade() si la base n'est à
        jour.

    // Table Names
    private static final String TABLE1_NAME = "myTable1";
    private static final String TABLE1_NAME = "myTable2";
```

```
9
10      // Column names
11      // ...
12
13      private static final String TABLE1_CREATE =
14          "CREATE TABLE " + TABLE1_NAME + " (" +
15          KEY_ID + "INTEGER PRIMARY KEY,"+ ...);
16
17      // ...
18
19      MyDataBase(Context context) {
20          super(context, DATABASE_NAME, null, DATABASE_VERSION);
21      }
22
23      // Appelé si la base n'existe pas
24      @Override
25      public void onCreate(SQLiteDatabase db) {
26          db.execSQL(MYBASE_TABLE_CREATE);
27      }
28  }
29
30  MyDataBase mDbHelper = new MyDataBase(getContext());
```

10.3 Requêtes sur une base

10.3.1 Insertions

```
1  // Gets the data repository in write mode
2  SQLiteDatabase db = mDbHelper.getWritableDatabase();
3
4  // Create a new map of values, where column names are the keys
5  ContentValues values = new ContentValues();
6  values.put(FeedEntry.COLUMN_NAME_ENTRY_ID, id);
7  values.put(FeedEntry.COLUMN_NAME_TITLE, title);
8  values.put(FeedEntry.COLUMN_NAME_CONTENT, content);
9
10 // Insert the new row, returning the primary key value of the new row
11 long newRowId;
12 newRowId = db.insert(FeedEntry.TABLE_NAME, FeedEntry.COLUMN_NAME_NULLABLE, values);
```

10.3.2 Requêtes

```
1  SQLiteDatabase db = mDbHelper.getReadableDatabase();
2
3  // Define a projection that specifies which columns from the database
4  // you will actually use after this query.
5  String[] projection = {
6      FeedEntry._ID,
7      FeedEntry.COLUMN_NAME_TITLE,
8      FeedEntry.COLUMN_NAME_UPDATED,
9      // ...
10 };
11
12 // How you want the results sorted in the resulting Cursor
13 String sortOrder = FeedEntry.COLUMN_NAME_UPDATED + " DESC";
14 Cursor cursor = db.query(
15     // The table to query
```

```
    FeedEntry.TABLE_NAME,

    // The columns to return
    projection,

    // The columns for the WHERE clause
    selection,

    // The values for the WHERE clause
    selectionArgs,

    // don't group the rows
    null,

    // don't filter by row groups
    null,

    // The sort order
    sortOrder
);

// Manipulate the cursor
cursor.moveToFirst();

long itemId = cursor.getLong(
    cursor.getColumnIndexOrThrow(FeedEntry._ID));
```

Note : on peut aussi utiliser `db.rawQuery(s)` pour faire une requête en pure SQL (attention aux injections).

10.3.3 Mises à jour et suppressions

Même chose que `db.insert(...)` mais avec les méthodes `db.update(...)` et `db.delete(...)` .