Développement d'applications mobiles
Partie 2

Accès et gestion des bases de données - SQLite

I) Introduction

Android inclut un SGBD basé sur SQL et nommé **SQLite**. Pour des informations détaillées, on peut consulter le site *www.sqlite.org*. Ce SGBD est utilisé par de nombreuses applications, y compris des applications très courantes telles que Skype ou Firefox.

SQLite n'utilise pas de serveur, et ne requiert aucune configuration. Il est embarqué dans chaque appareil Android, et est économe en terme d'occupation mémoire. Il s'agit d'un moteur transactionnel de bases de données SQL qui implémente une grande partie du standard SQL92. Il est écrit en C et C++. On pourra effectuer aussi bien des requêtes de manipulation de données (SELECT, INSERT par exemple) que de requêtes de définition de données (CREATE TABLE par exemple).

Une base de données est par défaut privée à l'application qui l'a créée, et n'est accessible directement que par cette application. Physiquement, elle est stockée à l'emplacement suivant :

```
data / data / package de l'application / databases
```

Cependant, le partage des données avec d'autres applications est possible grâce à la notion de fournisseur de contenu (voir plus loin, dans un autre chapitre).

Tout comme pour les fichiers, les accès à la base de données qu'ils soient en lecture ou en écriture, ne doivent pas être effectués dans le *thread* principal de l'application. Ils devront être placés dans un *thread* dédié (voir plus loin, les chapitres sur la réactivité des applications).

Il existe **5 types de données** que l'on peut stocker dans une table. Toutefois, il faut savoir que les types spécifiés, lors de la création de la table, ne le sont qu'à titre indicatif :

- ✓ **NULL** pour les données nulles
- ✓ **INTEGER**
- ✓ **REAL**
- ✓ **TEXT**
- ✓ **BLOB** pour les données brutes, comme une image par exemple. Il n'est pas conseillé d'utiliser cette possibilité pour conserver la propriété de légèreté d'une base de données SQLite. Il sera préférable de stocker dans la base le chemin du fichier contenant l'image, sous la forme d'une URI (Uniform Resource Identifier, chaîne de caractères qui identifie une ressource).

Les classes permettant permettant de gérer une base de données SQLite se trouvent dans le *package* ***android.database.sqlite***. Parmi celles-ci, les deux principales sont ***SQLiteOpenHelper***, et ***SQLiteDatabase***.

Le programmeur qui souhaite utiliser une base de données dans une application doit réaliser les deux étapes suivantes :

1) **Ecrire une classe pour gérer la connexion à la base de données**. Cette classe permettra notamment de créer la base de données et de gérer ses différentes versions. Elle héritera de la classe ***SQLiteOpenHelper.***

2) Dans les classes activités, il faudra **écrire les accès à la base de données**. Notons toutefois que les accès, surtout s'ils sont nombreux et sous des formes diverses ont tout intérêt à être codés dans une classe spécifique (voir la section 5 de ce chapitre), qui sera ensuite utilisée dans les classes activités elles-mêmes.

Les accès à une base de données SQLite sont réalisés grâce à la classe ***android.database.sqlite.SQLiteDatabase***. Celle-ci fournit des méthodes pour ouvrir, effectuer des requêtes, mettre à jour et fermer la base de données.

II) Connexion à une base de données avec la classe SQLiteOpenHelper

Avant toute utilisation d'une base de donnée, il est nécessaire de se connecter à celle-ci. Pour ce faire, le programmeur doit définir une classe de gestion de la connexion. Il s'agira d'une sous-classe de ***android.database.sqlite.SQLiteOpenHelper***.

Cette sous-classe contiendra au moins les 3 méthodes suivantes :

- ***void onCreate(SQLiteDatabase bd)***

Lors d'un accès à la base de données, cette méthode est automatiquement appelée si la base n'existe pas encore. On trouvera donc dans celle-ci les instructions permettant de créer la base argument, et éventuellement celles permettant de renseigner la base avec des données initiales.

- ***void onUpgrade(SQLiteDatabase bd, int ancienneVersion, int nouvelleVersion)***

Cette méthode sera invoquée chaque fois que l'utilisateur mettra à jour l'application avec un nouveau schéma de base de données. On trouvera en général dans la méthode une demande d'exécution d'une requête pour détruire la base actuelle, et des instructions permettant de créer un nouvelle base conforme à la nouvelle version.

- ***un constructeur avec argument SQLiteOpenHelper (Context context, String name, SQLiteDatabase.CursorFactory factory, int version)***

Il fera appel au constructeur analogue de la classe de base ***SQLiteOpenHelper***. Les arguments sont :

- le contexte,
- le nom de la base de données,
- une éventuelle fabrique de curseur que le programmeur a pu définir lui-même. Le plus souvent, on donnera à ce paramètre la valeur *null* qui indique que la fabrique par défaut doit être utilisée,
- un entier égal au numéro de la version du schéma de la base de données.

Exemple

Nous souhaitons créer une base de données pour stocker la description des pays de l'Union Européenne: nom du pays, capitale et année d'entrée dans l'union, sachant que celle-ci est au moins égale à 1952. Le code SQL sera le suivant :

```
CREATE TABLE unionEuropeenne ( _id INTEGER PRIMARY KEY AUTOINCREMENT,
                               intitule TEXT,
                               capitale TEXT,
                               annee INTEGER ( CHECK annee > 1951));
```

Les arguments servent à préciser le nom de chaque attribut et le type associé. Il est possible d'associer des contraintes aux attributs comme :

- ✓ PRIMARY KEY — pour préciser que cet attribut est la clé primaire de la table
  Il est préférable de nommer cet attribut *_id*.
  Il est fortement conseillé que cette clé soit auto-incrémentée.
- ✓ NOT NULL — pour interdire la valeur nulle pour cet attribut
- ✓ CHECK — pour préciser une contrainte spécifique
- ✓ DEFAULT — pour indiquer une valeur par défaut
- ✓ ...

Dans la classe *SQLiteDatabase*, on trouve la méthode : ***void execSQL(String requete)***
Elle permet d'exécuter une requête SQL sur la base. Cette méthode n'est utilisable qu'à la condition que la requête ne renvoie pas de résultat. Donc on ne peut pas l'utiliser pour effectuer une requête SELECT, par exemple.

Pour créer la base de l'Union Européenne, la requête SQL de création précédente pourra être placée en argument de la méthode ***execSQL***.

Code Java

```
/*
 * Classe permettant de gérer la base de données qui contient la description des pays
 * GestionBDPaysUE.java                                                    02/15
 */
package com.example.exbasedonnee;

import android.content.ContentValues;
import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteDatabase.CursorFactory;
import android.database.sqlite.SQLiteOpenHelper;
import android.util.Log;

/**
 * Cette classe permet de gérer la base de données qui contient la description des pays
 * de l'UE. Elle contient :
 *   - un constructeur
 *   - une méthode permettant de créer la base, et de l'initialiser avec quelques pays
 *   - une méthode qui sera invoquée si on change de version pour la base de données
 * @author LP
 * @version 1.0
 */
public class GestionBDPaysUE extends SQLiteOpenHelper {

    /** Nom du champ correspondant à l'identifiant du pays, la clé */
```

```
    public static final String PAYS_CLE = "_id";

    /** Nom du champ correspondant au nom du pays */
    public static final String PAYS_INTITULE = "intitule";

    /** Nom du champ correspondant à la capitale du pays */
    public static final String PAYS_CAPITALE = "capitale";

    /** Nom du champ correspondant à l'année d'adhésion du pays */
    public static final String PAYS_ANNEE = "annee";

    /** Nom de la table qui contiendra la description des pays de l'UE */
    public static final String NOM_TABLE_PAYS = "UnionEuropeenne";

    /** Requête pour sélectionner tous les enregistrements de la table */
    public static final String REQUETE_TOUT_SELECTIONNER =
              "select "
              + PAYS_CLE + ", "
              + PAYS_INTITULE + ", "
              + PAYS_CAPITALE + ", "
              + PAYS_ANNEE
              + " from " + NOM_TABLE_PAYS
              + " order by " + PAYS_ANNEE;

    /** Requête pour la création de la table */
    private static final String CREATION_TABLE_PAYS =
              "CREATE TABLE " + NOM_TABLE_PAYS  + " ( "
              + PAYS_CLE + " INTEGER PRIMARY KEY AUTOINCREMENT, "
              + PAYS_INTITULE + " TEXT, "
              + PAYS_CAPITALE + " TEXT, "
              + PAYS_ANNEE + " INTEGER CHECK (" + PAYS_ANNEE  + "  > 1951)); ";

    /** Requête pour supprimer la table */
    private static final String SUPPRIMER_TABLE_PAYS =
              "DROP TABLE IF EXISTS " + NOM_TABLE_PAYS + " ;" ;


    /**
     * Constructeur de la classe
     * @param contexte    contexte d'appel
     * @param nom         nom de la base de données
     * @param fabrique    une fabrique de curseur ou le plus souvent null
     * @param version     entier égal au numéro de version du schéma de la base de données
     */
    public GestionBDPaysUE(Context contexte, String nom, CursorFactory fabrique,
                           int version) {
        super(contexte, nom, fabrique, version);
    }


    @Override
    public void onCreate(SQLiteDatabase bd) {
        bd.execSQL(CREATION_TABLE_PAYS);   // exécution de la requête pour créer la base

        // objet pour préparer chacun des enregistrements à ajouter à la base
        ContentValues enregistrement = new ContentValues();

        /*
         *  on prépare un enregisrement, en donnant la valeur de chacune des colonnes.
         *  Cet enregistrement est ensuite inséré dans la base
         */
        enregistrement.put(PAYS_INTITULE, "Allemagne");
        enregistrement.put(PAYS_CAPITALE, "Berlin");
        enregistrement.put(PAYS_ANNEE, 1952);
        bd.insert(NOM_TABLE_PAYS, PAYS_INTITULE, enregistrement);

        // on fait de même pour les autres enregistrements
```

```
        enregistrement.put(PAYS_INTITULE, "Belgique");
        enregistrement.put(PAYS_CAPITALE, "Bruxelles");
        enregistrement.put(PAYS_ANNEE, 1952);
        bd.insert(NOM_TABLE_PAYS, PAYS_INTITULE, enregistrement);

        ......
    }

    @Override
    public void onUpgrade(SQLiteDatabase bd, int ancienneVersion, int nouvelleVersion) {
        bd.execSQL(SUPPRIMER_TABLE_PAYS);  // on détruit et on recrée la base des pays
        onCreate(bd);                      // méthode simplifiée
    }
}
```

Remarques

Une bonne habitude consiste à définir en tant que constantes chaînes de caractères :

- ✓ les intitulés des attributs de la table
- ✓ le nom de la table
- ✓ éventuellement les requêtes à réaliser sur la table, y compris celles qui ne sont pas utilisées dans la classe qui gère la connexion, mais qui le seront dans d'autres classes, comme les activités de l'application par exemple. Cependant nous verrons plus loin dans ce chapitre comment améliorer ce dernier point grâce à l'utilisation d'un DAO (*Data Access Object*)

Insertion de données dans la base

Pour insérer des données dans la base, il existe 2 manières de procéder :

- ✓ utiliser un appel à ***execSQL*** et donner en argument l'instruction SQL permettant d'ajouter un enregistrement : "INSERT ...."

- ✓ appeler la méthode ***insert*** (voir exemple précédent) sur la base :

long insert(String nomTable, String nomDUneColonne, ContentValues enregistrementAInserer)

si l'instance *ContentValues* est vide (c'est-à-dire si on n'a pas inséré de valeur dans celle-ci), la colonne indiquée recevra la valeur NULL. Si l'argument *nomDUneColonne* est égal à *null*, si l'enregistrement à insérer est vide, alors il ne sera pas inséré.

La méthode renvoie l'identifiant de l'enregistrement inséré, ou -1 en cas d'erreur.

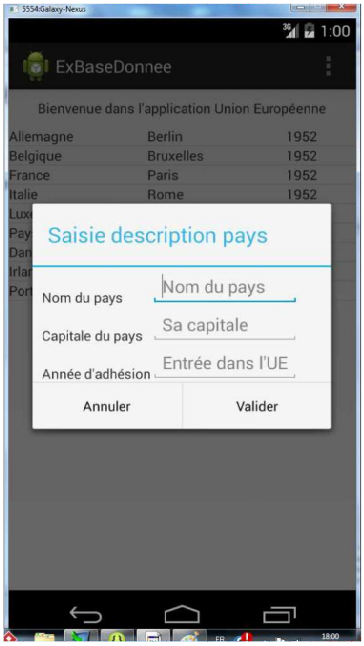
Avant d'effectuer l'appel à la méthode ***insert***, il faut préparer une instance de type ***ContentValues***. Pour insérer des données dans l'objet ***ContentValues***, on utilise une méthode ***put*** qui possède 2 arguments:

- le premier est une chaîne de caractères qui indique le nom de l'attribut, ou autrement dit le nom d'une colonne de la table
- et le deuxième est la valeur de l'attribut. En fait, il existe plusieurs méthodes ***put*** surchargées, avec des types différents pour le deuxième argument.

III) Exemple simplifié - Comment accéder à une base de données à partir d'une activité ?

Suite de l'exemple

Au lancement d'une activité, on souhaite afficher la liste des pays présents dans la base de données. Via un menu d'options, il sera possible d'ajouter un nouveau pays ou de supprimer un pays existant.



Dans cette partie, nous allons programmer toutes les opérations d'accès à la base de donnée dans la classe activité elle-même. Cette manière de procéder n'est pas la meilleure du point de vue de la structuration du code. Nous verrons dans la section suivante comment l'améliorer avec un DAO.

Pour accéder à la base de données et afficher son contenu, nous procédons par étapes :

1. il faut créer une instance de la classe de gestion de la base *GestionBDPaysUE* (classe qui hérite de ***SQLiteOpenHelper***)

2. invoquer sur cette instance la méthode ***getWritableDatabase()*** pour obtenir une instance de la base de données. Remarque : il existe aussi ***getReadableDatabase()*** si nous souhaitions seulement accéder en lecture à la base

3. effectuer une requête sur la base de manière à récupérer l'ensemble des enregistrements qu'elle contient. Le résultat de la requête est placé dans un curseur, instance de ***Cursor***. La requête s'effectue grâce à la méthode (de la classe *SQLiteDatabase*) :
   Cursor rawQuery(String requete, String[] argument)
   La chaîne qui correspond à la requête peut contenir le symbole '?'.

Les arguments présents dans le tableau argument viendront prendre la place de ce symbole, dans la requête.

4. comme d'habitude, pour afficher une liste, il faut utiliser un adaptateur. Celui qui convient bien pour afficher une instance de type *Cursor* est ***SimpleCursorAdapter***. Son constructeur attend en paramètre:
   - ✓ le contexte
   - ✓ une référence sur un fichier *layout* contenant la vue destinée à afficher chacune des lignes de la liste
   - ✓ un curseur
   - ✓ un tableau de chaînes de caractères contenant les intitulés des champs de la table à afficher (ils doivent correspondre à des intitulés présents dans le curseur)
   - ✓ un tableau d'entiers contenant les références des *widgets* présents dans la vue et qui recevront les valeurs des champs (récupérées dans le curseur)

   Attention : Pour que ce constructeur fonctionne correctement, le curseur doit contenir une colonne dont l'identifiant est **_id**
   Si la colonne de la clé ne se nomme pas _id, on peut appliquer au préalable la requête
   ***SELECT** id **as** _id, intitule, capitale, annee **from** UnionEuropeenne*

5. associer l'adaptateur à la liste

Exemple

→ Appel à la méthode *rawQuery* pour sélectionner tous les pays de la base :

```
curseurSurBase = base.rawQuery(GestionBDPaysUE.REQUETE_TOUT_SELECTIONNER, null );
```

→ Appel au constructeur de la classe SimpleCursorAdpater pour initialiser l'adaptateur :

```
adaptateur = new SimpleCursorAdapter(this,
                                     R.layout.ligne_liste,
                                     curseurSurBase,
                                     new String[] {GestionBDPaysUE.PAYS_INTITULE,
                                                   GestionBDPaysUE.PAYS_CAPITALE,
                                                   GestionBDPaysUE.PAYS_ANNEE},
                                     new int[] {R.id.nom_pays,
                                                R.id.nom_capitale,
                                                R.id.annee_adhesion}, 0);
```

Remarque

Si par la suite on est amené à modifier la base (pour ajouter ou supprimer des enregistrements), on souhaitera bien sûr modifier la liste des pays affichés. Il sera nécessaire de mettre à jour le curseur en relançant la requête qui sélectionne tous les enregistrements, et de faire en sorte que l'adaptateur prenne en compte le changement du curseur. Ceci se fera par un appel à la méthode ***swapCursor*** (voir exemple).

Opérations possibles sur le curseur

Les curseurs sont des objets qui représentent les résultats d'une recherche dans une base de données. Le curseur renvoyé en tant que résultat lors de l'appel à une méthode pointe généralement vers une ligne du résultat de la recherche. L'espace mémoire et le temps d'exécution sont ainsi optimisés, car toutes les données constituant le résultat ne sont pas chargées en mémoire.

La classe ***Cursor*** contient des méthodes pour parcourir les données que contient le curseur. Au départ, le curseur est positionné avant le premier enregistrement. Les principales méthodes sont :

Déplacements du curseur	
boolean moveToFirst()	pour amener le curseur sur la première ligne (premier enregistrement renvoyé par la requête)
boolean moveToLast()	ou à la dernière ligne
boolean moveToNext()	permet de déplacer le curseur sur l'enregistrement suivant. Elle renvoie un booléen, vrai si il y a bien une donnée suivante.
boolean moveToPrevious()	permet de déplacer le curseur sur l'enregistrement précédent.
boolean moveToPosition(int position)	pour amener le curseur à une position précise
Tests et accès divers	
int getCount()	renvoie le nombre de lignes référencées par le curseur
int getPosition()	renvoie la position actuelle du curseur
boolean isAfterLast()	pour tester si le pointeur est situé après la dernière ligne
getString(int numéroColonne)	permet d'extraire la donnée d'une colonne précise, ceci à partir de l'enregistrement pointé par le curseur. A utiliser si la donnée à extraire est de type *String*
getFloat(int numéroColonne)	Même principe. A utiliser si la donnée à extraire est de type *Float*
getDouble(int numéroColonne)	Même principe. A utiliser si la donnée à extraire est de type *Double*
close()	Pour fermer le curseur, et libérer les ressources

Exemple :

```
// exemple : pour afficher la valeur de la colonne 1
// de tous les enregistrements présents dans le curseur
while (curseur.moveToNext() ) {
    Log.i(TAG, cursor.getString(1));
}
curseur.close();
```

Remarque importante

Lorsque l'on a fini d'utiliser une base de données, il faut réaliser un appel à ***close()*** pour libérer la connexion. De même, après utilisation d'un curseur, il faut libérer les ressources qu'il occupe en invoquant la méthode ***close()*** sur celui-ci.

Classe qui représente l'activité

```
/*
 * Cette activité gère les pays de l'UE présents dans une base de données
```

```java
 * ActivitePaysUE.java
 */
package com.example.exbasedonnee;

import android.content.ContentValues;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.widget.SimpleCursorAdapter;
import ....

/**
 * Cette activité gère la liste des pays de l'UE (nom, capitale, année d'adhésion à
 * l'UE). Ces pays sont stockés dans une base de données gérée à travers la classe
 * GestionBDPaysUE.
 * Via un menu d'options, l'utilisateur pourra ajouter ou supprimer un pays.
 * @author INFO2
 * @version 1.0
 *
 */
public class ActivitePaysUE extends ListActivity {

    /** Numéro de version de la base de données */
    private static final int VERSION = 1;

    /** Nom de la base de données qui contiendra les pays gérés */
    private static final String NOM_BD = "lespaysue.db";

    /** Gestionnaire permettant de créer la base de données */
    private GestionBDPaysUE gestionnaireBase;

    /** Base de données contenant la description des pays */
    private SQLiteDatabase base;

    /** Curseur sur l'ensemble des pays de la base */
    private Cursor curseurSurBase;

    /** Adaptateur : il sert d'intermédiaire entre le curseur et la vue qui
     *  affiche les pays. C'est curseurSurBase qui lui sera associé
     */
    private SimpleCursorAdapter adaptateur;


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

        /*
         * Préparation des données (les pays de l'UE) en vue de leur affichage :
         *   - on crée un gestionnaire de la base de données (si la base n'existe pas
         *       elle sera créée et initialisée grâce à ce gestionnaire)
         *   - on récupère la base de donnée en précisant que l'on souhaitera écrire
         *       dans celle-ci
         *   - on initialise un curseur contenant l'ensemble de tous les enregistrements
         *       de la base
         */
        gestionnaireBase = new GestionBDPaysUE(this, NOM_BD, null, VERSION);
        base = gestionnaireBase.getWritableDatabase();
        curseurSurBase = base.rawQuery(GestionBDPaysUE.REQUETE_TOUT_SELECTIONNER, null );

        /*
         *  On crée un adpatateur pour rassembler les données à afficher.
         *  ligne_liste est une vue qui affiche sur une ligne les 3 informations :
         *  nom du pays, capitale et année d'adhésion.
         *  Il s'agit de 3 TextView ayant pour identifiants :
         *     R.id.nom_pays,   R.id.nom_capitale,    R.id.annee_adhesion.
         *  Ces 3 TextView seront renseignés avec les données de la base situées sur
```

```
         *  les colonnes nommées respectivement :
         *     PAYS_INTITULE,   PAYS_CAPITALE,   et PAYS_ANNEE
         */
        adaptateur = new SimpleCursorAdapter(this,
                                             R.layout.ligne_liste,
                                             curseurSurBase,
                                             new String[] {GestionBDPaysUE.PAYS_INTITULE,
                                                           GestionBDPaysUE.PAYS_CAPITALE,
                                                           GestionBDPaysUE.PAYS_ANNEE},
                                             new int[] {R.id.nom_pays,
                                                        R.id.nom_capitale,
                                                        R.id.annee_adhesion}, 0);
        setListAdapter(adaptateur);

        // on précise qu'un menu d'options est associé à l'activité
        registerForContextMenu(getListView());
    }


    /**
     * Méthode invoquée automatiquement lorsque l'utilisateur active le menu d'options,
     * pour la première fois
     */
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {

        /*
         *  on désérialise le fichier XML décriant le menu et on l'associe
         *  au menu argument (celui qui a été activé)
         */
        new MenuInflater(this).inflate(R.menu.menu_pays, menu);
        return super.onCreateOptionsMenu(menu);
    }


    /**
     * Méthode invoquée automatiquement lorsque l'utilisateur choisira une option
     * dans le  menu d'options
     */
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {

        // selon l'option sélectionnée dans le menu, on réalise le traitement adéquat
        switch(item.getItemId()) {
        case R.id.ajout_pays :            // ajout d'un nouveau pays
              saisirAjouterPays();
              break;
        case R.id.suppression_pays:       // suppression d'un pays
              saisirSupprimerPays();
              break;
        case R.id.annuler :               // retour à la liste principale
              break;

        }
        return (super.onOptionsItemSelected(item));
    }


    /**
     * Cette méthode affiche une boîte de dialogue permettant à l'utilisateur de
     * saisir un nouveau pays et toutes ses caractéristiques. Ce pays est ensuite
     * ajouté à la base de données et à la liste gérée par l'activité
     */
    private void saisirAjouterPays() {

        // on désérialise le layout qui est associé à la boîte de saisie d'un pays
        final View boiteSaisie = getLayoutInflater().inflate(R.layout.saisie_pays, null);
```

```
        // Création d'une boîte de dialogue pour saisir le pays
        new AlertDialog
                .Builder(this)
                .setTitle(getResources().getString(R.string.titre_boite_saisie_pays))
                .setView(boiteSaisie)
                .setPositiveButton(getResources().getString(R.string.bouton_positif),
                                new DialogInterface.OnClickListener() {

                        // méthode invoquée lorsque l'utilisateur validera la saisie
                        public void onClick(DialogInterface dialog, int leBouton) {

                                // on récupère un accès sur les zones de saisies de la boîte
                                EditText nomPays =
                                                (EditText) boiteSaisie.findViewById(R.id.saisie_nom_pays);
                                EditText nomCapitale =
                                                (EditText) boiteSaisie.findViewById(R.id.saisie_capitale);
                                EditText annee =
                                                (EditText) boiteSaisie.findViewById(R.id.saisie_annee);

                                // préparation d'un nouvel enregistrement pour la base de données
                                ContentValues enregistrement = new ContentValues();

                                enregistrement.put(GestionBDPaysUE.PAYS_INTITULE,
                                                        nomPays.getText().toString());
                                enregistrement.put(GestionBDPaysUE.PAYS_CAPITALE,
                                                        nomCapitale.getText().toString());
                                enregistrement.put(GestionBDPaysUE.PAYS_ANNEE,
                                                        Integer.parseInt(annee.getText().toString()));

                                // insertion de l'enregistrement dans la base
                                base.insert(GestionBDPaysUE.NOM_TABLE_PAYS,
                                            GestionBDPaysUE.PAYS_INTITULE, enregistrement);

                                // on met à jour le curseur en refaisant la requête de sélection
                                curseurSurBase =
                                                base.rawQuery(GestionBDPaysUE.REQUETE_TOUT_SELECTIONNER,
                                                                null);

                                // on met à jour l'adapteur et on informe la liste du changement
                                adaptateur.swapCursor(curseurSurBase);
                                onContentChanged();
                        }
                })
                .setNegativeButton(getResources().getString(R.string.bouton_negatif), null)
                .show();
    }


    /**
     * Cette méthode affiche une boîte de dialogue permettant à l'utilisateur de
     * saisir le nom d'un pays. Ce pays est ensuite supprimé de la base
     */
    private void saisirSupprimerPays() {

        // on désérialise le layout qui est associé à la boîte de saisie
        final View boiteSaisie = getLayoutInflater().inflate(R.layout.saisie_nom, null);

        // Création d'une boîte de dialogue pour saisir le nom du pays
        new AlertDialog.Builder(this)
                .setTitle(getResources().getString(R.string.titre_boite_saisie_nom))
                .setView(boiteSaisie)
                .setPositiveButton(getResources().getString(R.string.bouton_positif),
                                new DialogInterface.OnClickListener() {

                        // méthode invoquée lorsque l'utilisateur validera la saisie
```

```
                        public void onClick(DialogInterface dialog, int leBouton) {

                            // on récupère un accès sur la zone de saisie de la boîte de dialogue
                            EditText nomPays =
                                (EditText) boiteSaisie.findViewById( R.id.saisie_le_nom_a_supprimer);

                            // suppression de l'enregistrement dans la base
                            int resultat =
                                        base.delete(GestionBDPaysUE.NOM_TABLE_PAYS,
                                                    GestionBDPaysUE.PAYS_INTITULE + " = ?",
                                                    new String[] { nomPays.getText().toString() });

                            // résultat est égal au nombre de lignes supprimées
                            if (resultat != 0) {

                                /*
                                 *  une suppression a eu lieu : on refait la requête de sélection
                                 *  pour actualiser le curseur. L'adaptateur est modifié avec le
                                 *  nouveau curseur, et la liste des pays est également actualisée
                                 */
                                curseurSurBase =
                                        base.rawQuery(GestionBDPaysUE.REQUETE_TOUT_SELECTIONNER,
                                                      null);
                                adaptateur.swapCursor(curseurSurBase);
                                onContentChanged();
                            } else {

                                // aucune suppression effectuée : le pays n'existe pas
                                Toast.makeText(ActivitePaysUE.this,
                                               R.string.messageToast, Toast.LENGTH_LONG)
                                     .show();
                            }
                        }
                    })
                    .setNegativeButton(getResources().getString(R.string.bouton_negatif), null)
                    .show();
    }

}
```

Fichier *ActivitePaysUE.java*

```
<!--  vue qui décrit chacune des lignes de la liste des pays
      Plus précisément, le nom du pays est affiché, ensuite sa capitale, puis son année d'adhésion.
      La largeur des TextView est égale à 0, le poids correspond donc à un pourcentage d'occupation
      de l'espace en largeur (respectivement 40%, 40% et 20%) -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal" >

    <TextView
        android:id="@+id/nom_pays"
        android:layout_width="0dp"
        android:layout_height="fill_parent"
        android:layout_weight="40" />

    <TextView
        android:id="@+id/nom_capitale"
        android:layout_width="0dp"
        android:layout_height="fill_parent"
        android:layout_weight="40" />

    <TextView
        android:id="@+id/annee_adhesion"
        android:layout_width="0dp"
        android:layout_height="fill_parent"
```

```
        android:layout_weight="20" />

</LinearLayout>
```

Fichier *ligne_liste.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<!--  vue associée à la boîte de dialogue permettant de saisir un nom de pays.
      Il s'agit du nom du pays à supprimer -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal" >

     <TextView
               android:layout_height="wrap_content"
               android:layout_width="wrap_content"
               android:layout_marginLeft="10dip"
               android:layout_marginTop="10dip"
               android:layout_marginBottom="10dip"
               android:text="@string/message_saisie_nom" />

     <EditText
               android:id="@+id/saisie_le_nom_a_supprimer"
               android:layout_marginLeft="10dip"
               android:layout_height="wrap_content"
               android:layout_width="wrap_content"
               android:hint="@string/aide_saisie_nom" />

</LinearLayout>
```

Fichier *saisie_nom.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<!--  vue associée à la boîte de dialogue permettant de saisir les
      caractéristiques d'un nouveau pays -->
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <TableRow>       <!--  saisie du nom du pays -->
        <TextView
               android:layout_height="wrap_content"
               android:layout_marginLeft="10dip"
               android:layout_marginTop="10dip"
               android:layout_marginBottom="10dip"
               android:text="@string/message_saisie_nom" />

        <EditText
               android:id="@+id/saisie_nom_pays"
               android:layout_height="wrap_content"
               android:hint="@string/aide_saisie_nom" />
    </TableRow>

    <TableRow>      <!--  saisie du nom de la capitale du pays -->
        <TextView
               android:layout_height="wrap_content"
               android:layout_marginLeft="10dip"
               android:layout_marginTop="10dip"
               android:layout_marginBottom="10dip"
               android:text="@string/message_saisie_capitale" />

        <EditText
               android:id="@+id/saisie_capitale"
               android:layout_height="wrap_content"
```

```
                android:hint="@string/aide_saisie_capitale" />
    </TableRow>

    <TableRow>     <!--  saisie de l'année de l'adhésion du pays -->
        <TextView
                android:layout_height="wrap_content"
                android:layout_marginLeft="10dip"
                android:layout_marginTop="10dip"
                android:layout_marginBottom="10dip"
                android:text="@string/message_saisie_annee" />

        <EditText
                android:id="@+id/saisie_annee"
                android:layout_height="wrap_content"
                android:inputType="number"
                android:hint="@string/aide_saisie_annee" />
    </TableRow>

</TableLayout>
```

Fichier *saisie_pays.xml*

IV) Principales opérations

Le tableau ci-dessous résume les principales opérations possibles sur une table. Il s'agit de méthodes de la classe ***SQLiteDateBase***. Pour chaque opération, le programmeur a deux alternatives : utiliser une méthode spécifique à laquelle on donne en arguments les valeurs des différentes parties de la requête équivalente en SQL ou bien utiliser une méthode qui aura en paramètre directement la requête écrite dans le langage SQL.

	En utilisant une méthode utilitaire de la classe *SQLiteDateBase*	En écrivant la requête dans le langage SQL et en donnant cette requête en argument d'une méthode
Effectuer une requête de sélection	***Cursor query (boolean distinct, String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String orderBy, String limit)***	***Cursor rawQuery(String requete, String[] argument)***
Insérer une donnée	***long insert(String nomTable, String nomDUneColonne, ContentValues enregistrementAInserer)***	***void execSQL(String sql)***
Mettre à jour	***int update(String table, ContentValues enregistrement, String clauseWhere, String[] argumentDeWhere)***	***void execSQL(String sql)***
Supprimer	***int delete(String table, String clauseWhere, String[] argumentDeWhere)***	***void execSQL(String sql)***

Effectuer une requête de sélection

La méthode la plus générale pour effectuer une requête de sélection est la suivante :

Cursor query (boolean distinct, String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String orderBy, String limit)

Cette méthode permet, à partir de la valeur de ses arguments, de construire une requête SQL. Celle-ci sera ensuite compilée pour interroger la base de données.

Les paramètres de la méthode ***query*** ont la signification suivante.

distinct	Cet argument est optionnel. S'il a la valeur *vrai*, alors le résultat de la requête contiendra des éléments uniques.
table	Le nom de la table sur laquelle la requête sera réalisée.
columns	Indique quelles colonnes seront renvoyées par la requête, ou autrement dit la projection de la requête. On place dans le tableau les noms des colonnes souhaitées. Si on indique la valeur *null*, toutes les colonnes seront renvoyées. On a tout intérêt à limiter le nombre de colonnes quand c'est possible, ceci afin de réduire les ressources nécessaires.
selection	Indique la clause ***WHERE*** de la requête SQL, sans le mot-clé *WHERE*, et peut contenir le marqueur noté '?' pour paramétrer la clause. Si aucune clause *WHERE* n'est souhaitée, il faut indiquer la valeur *null*.
selectionArgs	Spécifie les valeurs de remplacement du marqueur ' ?' présent dans l'argument *selection*.
groupBy	Un filtre de regroupement de lignes, similaire à la clause *GROUP BY* d'une requête SQL. Si aucun regroupement n'est souhaité, on indique la valeur *null*.
having	Un filtre de condition d'apparition des lignes, similaire à la clause *HAVING* d'une requête SQL. Si on spécifie la valeur *null*, toutes les lignes seront incluses dans le résultat. La valeur *null* est requise, si la valeur du paramètre *groupBy* est égale à *null*.
orderBy	Spécifie l'ordre de tri des lignes selon le format de la clause *ORDER BY* d'une requête SQL. Si on indique la valeur *null*, l'ordre de tri par défaut sera appliqué.
limit	Cet argument est optionnel. S'il est présent, il sert à limiter le nombre de lignes retournées par la requête SQL. Si on spécifie la valeur *null*, toutes les lignes seront retournées.

Pour effectuer une requête de sélection, on peut aussi utiliser la méthode ***rawQuery*** (voir exemple dans la section 3 de ce chapitre). Elle permet de réaliser des requêtes simples respectant le schéma suivant :

SELECT * FROM *une_table* **WHERE** *une_condition*

Son profil est :

Cursor rawQuery(String requete, String[] argument)

La chaîne qui correspond à la requête doit être écrite en respectant la syntaxe SQL et peut contenir le marqueur noté '?'. Les arguments présents dans le tableau argument viendront prendre la place de ce marqueur, dans la requête.

Insérer une donnée avec la méthode *insert*

Pour insérer une donnée dans une table, on utilise la méthode *insert* comme indiqué dans la section 2 de ce chapitre.

Par exemple, pour ajouter un pays à la table des pays, on écrit :

```
base.insert(GestionBDPaysUE.NOM_TABLE_PAYS, GestionBDPaysUE.PAYS_INTITULE, enregistrement);
```

enregistrement est une instance de type *ContentValues* préparée au préalable et contenant la ligne à ajouter à la table. La méthode renvoie l'identifiant de l'enregistrement inséré ou -1 en cas d'erreur.

Supprimer une information avec la méthode *delete*

Pour supprimer un enregistrement d'une table, il faut fournir un critère pour identifier l'enregistrement (ou les enregistrements) à supprimer. Ce critère est fourni à la méthode ***delete*** via les arguments *clauseWhere* et *argumentDeWhere*.

int delete(String table, String clauseWhere, String[] argumentDeWhere)

La chaîne *clauseWhere* correspond à la clause ***WHERE*** de la requête SQL, sans le mot-clé *WHERE*, et peut contenir le marqueur noté '?'. Les chaînes présentes dans le tableau *argumentDeWhere* viendront prendre la place de ce marqueur, dans la clause *where*.

La valeur de retour de ***delete*** est le nombre d'enregistrements supprimés.

Par exemple pour supprimer le pays Grande-Bretagne, on écrira :

```
        base.delete(GestionBDPaysUE.NOM_TABLE_PAYS,
                    GestionBDPaysUE.PAYS_INTITULE + " = ?",
                    new String[] { "Grande-Bretagne" });
```

Mise à jour avec la méthode *update*

Pour modifier des enregistrements d'une table, on utilise la méthode ***update***.

int update(String table, ContentValues enregistrement, String clauseWhere, String[] argumentDeWhere)

La méthode ***update*** prend en paramètre :

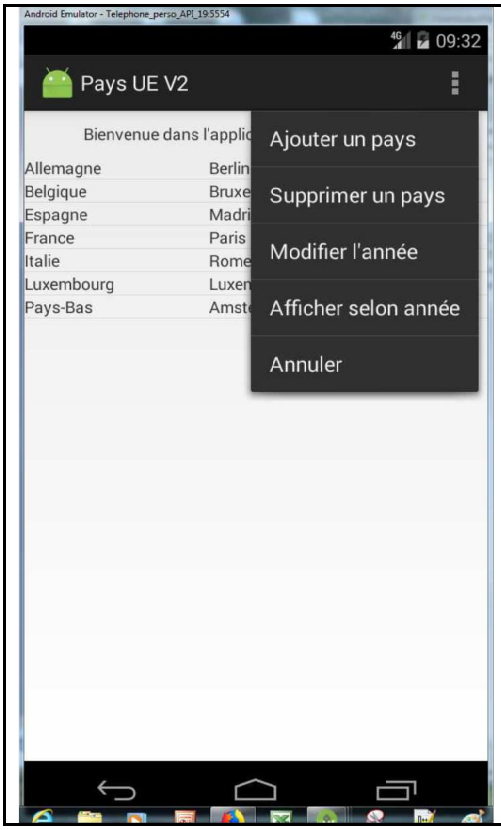
- ✓ le nom de la table
- ✓ un objet *ContentValues* contenant les colonnes et les nouvelles valeurs
- ✓ éventuellement une clause *where* de type *String* qui permet de filtrer les lignes à mettre à jour. Cet argument peut avoir la valeur null. Dans ce cas, toutes les lignes sont mises à jour.
- ✓ et une liste de paramètres, sous la forme d'un tableau de *String,* qui remplaceront les marqueurs notés ' **?**' et présents dans la clause *where*,

La valeur de retour de ***update*** est le nombre d'enregistrements modifiés.

V) Exemple en utilisant le pattern DAO

La manipulation d'une base de données nécessite d'écrire de nombreuses instructions. Pour aboutir à du code plus facile à écrire, à relire et à maintenir, il est fortement conseillé de fournir une couche d'abstraction supplémentaire. Cette couche permettra de séparer les accès à la base de données de la classe activité qui les réalise. Elle permettra aussi de prendre en compte plus facilement tout changement dans la structuration des données : passage à un fichier au lieu d'une base de données, utilisation d'un service Web au lieu d'une base de données locale …

Nous allons modifier l'exemple précédent pour ajouter des fonctionnalités à l'application et intégrer un DAO (*Data Access Object*).



Via le menu d'options, il sera maintenant possible :

- d'ajouter un pays
- supprimer un pays identifié à partir de son nom
- modifier l'année d'adhésion d'un pays identifié à partir de son nom
- afficher seulement les pays ayant une année d'adhésion précise. L'utilisateur indiquera la valeur 0 s'il souhaite afficher tous les pays.

Nous allons définir 4 classes Java :

- une classe *Pays* pour représenter un objet métier décrivant les caractéristiques d'un pays
- une classe de type *Helper* pour gérer la connexion à la base de données
- une classe facilitant l'accès aux pays de la base de donnés. Celle-ci joue le rôle du DAO
- la classe activité elle-même.

Dans la classe *Pays* ci-dessous, nous remarquons la présence d'un attribut pour chaque colonne de la table des pays, d'un accesseur et d'un modificateur pour chacun des attributs.

```
/*
 * Représentation d'un pays
 * Pays.java                                          11/17
 */
package com.cours.exemple.bd.application.paysbddao;

/**
 * Classe qui représente un pays de l'Union Européenne.
 * Les informations pertinentes sont : le nom du pays, sa capitale et
 * son année d'entrée dans l'union européenne.
 * @author LP MMS
 * @version 1.0
 */
public class Pays {

    /** Identifiant du pays dans la table qui le contient */
    private int identifiant;
```

```
    /** Nom du pays */
    private String intitule;

    /** Nom de la capitale du pays */
    private String capitale;

    /** Année d'entrée du pays dans l'Union Européenne */
    private int annee;


    /**
     * Constructeur avec initialisations par défaut
     */
    public Pays() {
        identifiant = 0;
        intitule = "";
        capitale = "";
        annee = 0;
    }
    /**
     * Constructeur avec données initiales en argument
     * @param intitule      nom du pays
     * @param capitale      nom de la capitale du pays
     * @param annee         année d'entrée dans l'Union Européenne
     */
    public Pays(String intitule, String capitale, int annee) {
        this.intitule = intitule;
        this.capitale = capitale;
        this.annee = annee;
        this.identifiant = 0;
    }

    /** Accesseur sur l'identifiant
     * @return  l'identifiant du pays
     */
    public int getIdentifiant() {
        return this.identifiant;
    }

    /**
     * Modificateur de l'identifiant
     * @param identifiant  nouvel identifiant
     */
    public void setIdentifiant(int identifiant) {
        this.identifiant = identifiant;
    }

    /**
     * Acesseur sur l'intitiulé
     * @return  l'intitulé du pays
     */
    public String getIntitule() {
        return this.intitule;
    }

    /**
     * Modificateur de l'intitulé
     * @param intitule  nouveau nom du pays
     */
    public void setIntitule(String intitule) {
        this.intitule = intitule;
    }

    /**
     * Accesseur sur le nom de la capitale
     * @return  le nom de la capitale
     */
```

```
    public String getCapitale() {
        return this.capitale;
    }

    /**
     * Modificateur du nom de la capitale; ,
     * @param capitale nouveau nom pour la capitale
     */
    public void setCapitale(String capitale) {
        this.capitale = capitale;
    }

    /**
     * Accesseur sur l'année d'entrée dans l'Union Européenne
     * @return l'année d'entrée dans l'Union Européenne
     */
    public int getAnnee() {
        return this.annee;
    }

    /**
     * Modificateur de l'année d'entrée dans l'Union Européenne
     * @param annee nouvelle valeur pour l'année d'entrée dans l'UE
     */
    public void setAnnee(int annee) {
        this.annee = annee;
    }
}
```

Classe *Pays.java*

La classe ci-dessous gère la connexion à la base de données et est très semblable à celle de la version précédente de l'application.

```
/*
 * Classe permettant de gérer la base de données qui contient la description des pays
 * HelperBDPaysUE.java                                          11/17
 */
package com.cours.exemple.bd.application.paysbddao;

. . .

/**
 * Cette classe permet de gérer la base de données qui contient la description des pays
 * de l'UE. Elle contient :
 *   - un constructeur
 *   - une méthode permettant de créer la base, et de l'initialiser avec quelques pays
 *   - une méthode qui sera invoquée si on change de version pour la base de données
 * @author LP MMS
 * @version 1.0
 */
public class HelperBDPaysUE extends SQLiteOpenHelper {

    /** Nom du champ correspondant à l'identifiant du pays, la clé */
    public static final String PAYS_CLE = "_id";

    /** Nom du champ correspondant au nom du pays */
    public static final String PAYS_INTITULE = "intitule";

    /** Nom du champ correspondant à la capitale du pays */
    public static final String PAYS_CAPITALE = "capitale";

    /** Nom du champ correspondant à l'année d'adhésion du pays */
    public static final String PAYS_ANNEE = "annee";

    /** Nom de la table qui contiendra la description des pays de l'UE */
```

```
    public static final String NOM_TABLE_PAYS = "UnionEuropeenne";


    /** Requete pour la création de la table */
    private static final String CREATION_TABLE_PAYS =
            "CREATE TABLE " + NOM_TABLE_PAYS  + " ( "
                    + PAYS_CLE + " INTEGER PRIMARY KEY AUTOINCREMENT, "
                    + PAYS_INTITULE + " TEXT, "
                    + PAYS_CAPITALE + " TEXT, "
                    + PAYS_ANNEE + " INTEGER CHECK (" + PAYS_ANNEE  + "  > 1951)); ";

    /** Requete pour supprimer la table */
    private static final String SUPPRIMER_TABLE_PAYS =
            "DROP TABLE IF EXISTS " + NOM_TABLE_PAYS + " ;" ;


    /**
     * Constructeur de la classe
     * @param contexte   contexte de l'appel
     * @param nom        nom de la base de données
     * @param fabrique   une fabrique de curseur ou le plus souvent null
     * @param version    entier égal au numéro de version du schéma de la base de données
     */
    public HelperBDPaysUE(Context contexte, String nom, SQLiteDatabase.CursorFactory fabrique, int version) {
        super(contexte, nom, fabrique, version);
    }


  @Override
    public void onCreate(SQLiteDatabase bd) {
        bd.execSQL(CREATION_TABLE_PAYS);      // exécution de la requête pour créer la base

        // objet pour préparer chacun des enregistrements à ajouter à la base
        ContentValues enregistrement = new ContentValues();

      /*
       *  on prépare un enregisrement, en donnant la valeur de chacune des colonnes.
       *  Cet enregistrement est ensuite inséré dans la base
       */
        enregistrement.put(PAYS_INTITULE, "Allemagne");
        enregistrement.put(PAYS_CAPITALE, "Berlin");
        enregistrement.put(PAYS_ANNEE, 1952);
        bd.insert(NOM_TABLE_PAYS, PAYS_INTITULE, enregistrement);

        // on fait de même pour les autres enregistrements
        . . .
    }

    @Override
    public void onUpgrade(SQLiteDatabase bd, int ancienneVersion, int nouvelleVersion) {
        bd.execSQL(SUPPRIMER_TABLE_PAYS);     // on détruit et on recrée la base des pays
        onCreate(bd);
    }
}
```

HelperBDPaysUE.java

La classe DAO possède les caractéristiques suivantes :

- ✓ Un constructeur pour créer l'objet gestionnaire de la base (le *helper*)
- ✓ Une méthode pour ouvrir et une autre pour refermer la base
- ✓ Des méthodes qui renvoient un curseur résultat d'une requête de sélection. Ici 2 méthodes pour renvoyer respectivement tous les pays de la table et seulement ceux ayant une année d'adhésion précise
- ✓ Une méthode qui renvoie tous les pays de la base sous la forme d'une *ArrayList*
- ✓ Une méthode qui renvoie une instance de type *Pays* obtenue à partir du nom du pays

- ✓ Des méthodes pour ajouter un pays, supprimer un pays ou mettre à jour un pays.

On remarque également la présence de deux méthodes privées destinées à faciliter l'écriture de certaines des autres méthodes de la classe :

- ✓ Renvoyer le pays référencé par un curseur sous la forme d'une instance de type *Pays*
- ✓ Renvoyer tous les pays référencés pas un curseur sous la forme d'une instance de type *ArrayList*. Cette liste contient des instances de type Pays.

```
/*
 * Classe d'aide d'accès à la table de la BD contenant les pays
 * PaysDAO.java                                                    11/17
 */
package com.cours.exemple.bd.application.paysbddao;

. . .

import java.util.ArrayList;

/**
 * Cette classe joue le rôle de facilitateur pour l'accès à la table
 * contenant les pays. La classe propose les services suivants :
 *    - une constante contenant la requête permettant de sélectionner
 *      tous les pays dans la table des pays
 *    - un constructeur qui crée un objet pour gérer la connexion à la BD
 *    - ouverture de la table en écriture
 *    - fermeture de la table et la connexion
 *    - renvoyer sous la forme d'un curseur toutes les lignes de la table
 *    - renvoyer tous les pays de la table sous la forme d'une ArrayList
 *    - renvoyer sous la forme d'un curseur toutes les lignes de la table qui correspondent
 *              à pays dont l'année d'adhésion a la valeur spécifiée
 *    - insérer un pays
 *    - supprimer un pays
 *    - mettre à jour un pays
 *    - modifier l'année d'adhésion d'un pays
 *    - renvoyer une instance de type Pays à partir du nom du pays
 * @author LP MMS
 * @version 1.0
 */
public class PaysDAO {

    /** Numéro de version de la base de donnée */
    private static final int VERSION = 1;

    /** Nom de la base de données qui contiendra les pays gérés */
    private static final String NOM_BD = "lespaysue.db";

    /** Numéro de la colonne contenant l'identifiant du pays, la clé */
    public static final int COLONNE_NUM_CLE = 0;

    /** Numéro de la colonne contenant le nom du pays */
    public static final int COLONNE_NUM_INTITULE = 1;

    /** Numéro de la colonne contenant la capitale du pays */
    public static final int COLONNE_NUM_CAPITALE = 2;

    /** Numéro de la colonne contenant l'année d'adhésion du pays */
    public static final int COLONNE_NUM_ANNEE = 3;

    /** Gestionnaire permettant de créer la base de donnée */
    private HelperBDPaysUE gestionnaireBase;

    /** Base de données contenant la description des pays */
    private SQLiteDatabase basePays;
```

```
    /** Requete pour sélectionner tous les enregistrements de la table */
    public static final String REQUETE_TOUT_SELECTIONNER =
            "select "
                    + HelperBDPaysUE.PAYS_CLE + ", "
                    + HelperBDPaysUE.PAYS_INTITULE + ", "
                    + HelperBDPaysUE.PAYS_CAPITALE + ", "
                    + HelperBDPaysUE.PAYS_ANNEE
                    + " from " + HelperBDPaysUE.NOM_TABLE_PAYS
                    + " order by " + HelperBDPaysUE.PAYS_INTITULE;
    /**
     * Constructeur permettant de créer l'objet gestionnaire de la BD
     * @param leContexte  contexte de l'activité créatrice
     */
    public PaysDAO(Context leContexte) {
        gestionnaireBase = new HelperBDPaysUE(leContexte, NOM_BD, null, VERSION);
    }

    /**
     * Ouverture de la table des pays
     */
    public void open() {
        basePays = gestionnaireBase.getWritableDatabase();
    }

    /**
     * Fermeture de la table des pays
     */
    public void close() {
        gestionnaireBase.close();
        basePays.close();
    }

    /**
     * Renvoie d'un curseur sur la totalité des pays
     * @return  un curseur référençant tous les pays de la base
     */
    public Cursor getCurseurPays() {
        return basePays.rawQuery(REQUETE_TOUT_SELECTIONNER, null );
    }

    /**
     * Renvoie tous les pays présents dans la table, sous la forme d'une liste
     * @return  une ArrayList contenant tous les pays de la table
     */
    public ArrayList<Pays> getAllPays() {
        Cursor curseurTous = getCurseurPays();
        return cursorToListePays(curseurTous);
    }

    /**
     * Sélectionne les pays dont l'année d'adhésion est donnée en argument
     * @param annee   année à sélectionner
     * @return  un curseur référençant les lignes de la table des pays pour
     *          lesquelles l'année d'adhésion est égale à l'argument
     */
    public Cursor getCurseurPaysSelonAnnee(int annee) {
        Cursor c = basePays.query(HelperBDPaysUE.NOM_TABLE_PAYS,
                new String[] {
                        HelperBDPaysUE.PAYS_CLE,
                        HelperBDPaysUE.PAYS_INTITULE,
                        HelperBDPaysUE.PAYS_CAPITALE,
                        HelperBDPaysUE.PAYS_ANNEE},
                HelperBDPaysUE.PAYS_ANNEE + " = ? ",
                new String[] {annee + ""}, null, null, null);
        return c;
    }
```

```
    /*
      * Insère le pays argument dans la table des pays
      * @param aInserer pays à insérer dans la table
      * @return
     */
    public long insertPays(Pays aInserer) {
        ContentValues enregistrement = new ContentValues();
        enregistrement.put(HelperBDPaysUE.PAYS_INTITULE, aInserer.getIntitule());
        enregistrement.put(HelperBDPaysUE.PAYS_CAPITALE, aInserer.getCapitale());
        enregistrement.put(HelperBDPaysUE.PAYS_ANNEE, aInserer.getAnnee());

        // insertion de l'enregistrement dans la base
        return basePays.insert(HelperBDPaysUE.NOM_TABLE_PAYS,
                        HelperBDPaysUE.PAYS_INTITULE, enregistrement);
    }

    /**
     * Supprime un pays de la table des pays
     * @param nomDuPays   nom du pays à supprimer
     * @return un entier égal au nombre de lignes supprimées
     */
    public int deletePays(String nomDuPays) {
        return basePays.delete(HelperBDPaysUE.NOM_TABLE_PAYS,
                        HelperBDPaysUE.PAYS_INTITULE + " = ?",
                        new String[] { nomDuPays });
    }

    /**
     * Modifie un pays avec l'instance de type Pays argument.
     * L'enregistrement à modifier est recherché selon le nom du pays argument
     * @param aModifier nouvelle valeur pour le pays à modifier
     * @return  un entier égal au nombre d'enregistrements modifiés
     */
    public int updatePays(Pays aModifier) {
        ContentValues nouveau  = new ContentValues();
        nouveau.put(HelperBDPaysUE.PAYS_INTITULE, aModifier.getIntitule());
        nouveau.put(HelperBDPaysUE.PAYS_CAPITALE, aModifier.getCapitale());
        nouveau.put(HelperBDPaysUE.PAYS_ANNEE, aModifier.getAnnee());
        return basePays.update(HelperBDPaysUE.NOM_TABLE_PAYS, nouveau,
                HelperBDPaysUE.PAYS_INTITULE + " = ?",
                new String[] {aModifier.getIntitule()} );
    }

    /**
     * Modifie l'année d'adhésion d'un pays
     * @param aModifier  nom du pays à modifier
     * @param nouvelleAnnee  nouvelle valeur pour l'année d'adhésion
     * @return  un entier égal au nombre d'enregistrements modifiés
     */
    public int updatePays(String aModifier, int nouvelleAnnee) {
        Pays lePays = getPays(aModifier);
        if (lePays == null) {
            return 0;
        }
        lePays.setAnnee(nouvelleAnnee);
        return updatePays(lePays);
    }

    /**
     * Recherche dans la table des pays, le pays dont le nom est donné en argument
     * @param nomPays   nom du pays à chercher dans la table des pays
     * @return  l'instance Pays dont le nom est donné en arugment (null si non trouvé)
     */
    public Pays getPays(String nomPays) {
        Cursor c = basePays.query(HelperBDPaysUE.NOM_TABLE_PAYS,
                new String[] {HelperBDPaysUE.PAYS_INTITULE,
                              HelperBDPaysUE.PAYS_CAPITALE,
                              HelperBDPaysUE.PAYS_ANNEE},
```

```
                HelperBDPaysUE.PAYS_INTITULE + " = ? ",
                new String[] {nomPays}, null, null, null);
        return cursorToPays(c);
    }



    /**
     * Transforme la ligne référencée par un curseur sur la table des pays
     * en instance de type Pays
     * @param c   curseur qui référence une ligne dans la table des pays
     * @return une instance de type Pays correspondant à celui référencé par
     *          le curseur (éventuellement null)
     */
    private Pays cursorToPays(Cursor c) {
        Pays aRenvoyer;
        if (c.getCount() == 0) {
            aRenvoyer = null;
        } else {
            c.moveToFirst();
            // on initialise l'instance Pays avec les valeurs des colonnes
            aRenvoyer = new Pays();
            aRenvoyer.setIntitule(c.getString(COLONNE_NUM_INTITULE - 1));
            aRenvoyer.setCapitale(c.getString(COLONNE_NUM_CAPITALE - 1));
            aRenvoyer.setAnnee(c.getInt(COLONNE_NUM_ANNEE -1));
        }
        c.close();
        return aRenvoyer;
    }


    /**
     * Transforme l'ensemble des pays référencés par un curseur en une liste de pays
     * @param c  un curseur sur un ensemble de pays
     * @return une liste contenant les pays référencés par le curseur
     */
    private ArrayList<Pays> cursorToListePays(Cursor c) {
        ArrayList<Pays> listePays = new ArrayList<>();
        if (c.getCount() != 0) {
            Pays aAjouter;
            c.moveToFirst();

            /* on parcourt toutes les lignes du curseur, et on ajoute
             * le pays référencé par le curseur à la liste
             */
            do {
                aAjouter = cursorToPays(c);
                listePays.add(aAjouter);
            } while (c.moveToNext());
        }
        c.close();
        return listePays;
    }
}
```

Classe *PaysDAO.java*

La classe ci-dessous est la nouvelle classe activité. On constate qu'elle n'accède plus directement à la base de données. Mais elle invoque les méthodes de la classe *PaysDAO* sur l'instance de ce type déclarée en tant qu'attribut.

```
/*
 * Cette activité gère les pays de l'UE présents dans une base de données     11/17
```

```
 * ActivitePrincipale.java
 */
package com.cours.exemple.bd.application.paysbddao;

. . .


/**
 * Cette activité gère la liste des pays de l'UE (nom, capitale, année d'adhésion à
 * l'UE). Ces pays sont stockées dans une base de données et affichés sous la forme
 * d'une liste (nom du pays, capitale, année d'adhésion)
 * Via un ménu d'options, il est possible de :
 *    - ajouter un pays
 *    - supprimer un pays
 *    - modifier l'année d'adhésion d'un pays
 *    - afficher les pays adhérents à partir d'une année précise
 *
 * Les accès à la base de données ne sont pas codés directement dans cette classe.
 * Ils le sont dans la classe PaysDAO. L'activité utilise les méthodes de la classe
 * PaysDAO pour accéder à la base de données.
 * @author  LP MMS
 * @version 1.0
 *
 */
public class ActivitePrincipale extends ListActivity {

    /**
     * Curseur sur l'ensemble ou un sous-ensemble des pays de la base
     * Ce sont les pays référencés par ce curseur qui sont actuellement
     * affichés par la liste associée à l'activité */
    private Cursor curseurSurBase;

    /**
     * Adaptateur : il sert d'intermédiaire entre le curseur et la vue qui
     *  affiche les pays. C'est curseurSurBase qui lui est associé
     */
    private SimpleCursorAdapter adaptateur;

    /** Objet destiné à faciliter l'accès à la table des pays */
    private PaysDAO accesPays;


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

      /*
       * On crée un objet pour faciliter l'accès à la table des pays.
       * On récupère un curseur référençant tous les pays de la base
       */
      accesPays = new PaysDAO(this);
      accesPays.open();
      curseurSurBase = accesPays.getCurseurPays();

      /*
       *  On crée un adpateur pour rassembler les données à afficher.
       *  ligne_liste est une vue qui affiche sur une ligne les 3 informations :
       *  nom du pays, capitale et année d'adhésion.
       *  Il s'agit de 3 TextView ayant pour identifiants : R.id.nom_pays,
       *  R.id.nom_capitale, R.id.annee_adhesion.
       *  Ces 3 TextView seront renseignés avec les données de la base situées sur
       *  les colonnes nommées respectivement : PAYS_INTITULE, PAYS_CAPITALE, et
       *  PAYS_ANNEE
       */
        adaptateur =
                new SimpleCursorAdapter(this,
                        R.layout.ligne_liste,
```

```
                    curseurSurBase,
                    new String[] {HelperBDPaysUE.PAYS_INTITULE,
                            HelperBDPaysUE.PAYS_CAPITALE,
                            HelperBDPaysUE.PAYS_ANNEE},
                    new int[] {R.id.nom_pays,
                            R.id.nom_capitale,
                            R.id.annee_adhesion}, 0);
        setListAdapter(adaptateur);
    }


    /**
     * Méthode invoquée automatiquement lorsque l'utiisateur active le menu d'options,
     * pour la première fois
     */
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {

       //on désérialise le fichier XML décriant le menu
        new MenuInflater(this).inflate(R.menu.menu_pays, menu);
        return super.onCreateOptionsMenu(menu);
    }


    /**
     * Méthode invoquée automatiquement quand l'utilisateur choisit une option
     */
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {

        // selon l'option sélectionnée dans le menu, on réalise le traitement adéquat
        switch(item.getItemId()) {
            case R.id.ajout_pays :          // ajout d'un nouveau pays
                saisirAjouterPays();
                break;
            case R.id.suppression_pays:     // suppression d'un pays
                saisirSupprimerPays();
                break;
            case R.id.modification_pays:        // modification d'un pays
                saisirModifierPays();
                break;
            case R.id.selection_annee:      // sélection selon une année
                saisirSelectionAnnee();
                break;
            case R.id.annuler :              // retour à la liste principale
                break;

        }
        return (super.onOptionsItemSelected(item));
    }



    /**
     * Cette méthode affiche une boîte de dialogue permettant à l'utilisateur de
     * saisir un nouveau pays et toutes ses caractéristiques. Ce pays est ensuite
     * ajouté à la base de donnée et à la liste gérée par l'activité
     */
    private void saisirAjouterPays() {

        // on désérialise le layout qui est associé à la boîte de saisie d'un pays
        final View boiteSaisie = getLayoutInflater().inflate(R.layout.saisie_pays, null);

      //  Création d'une boîte de dialogue :
        new AlertDialog.Builder(this)
                .setTitle(getResources().getString(R.string.titre_boite_saisie_pays))
                .setView(boiteSaisie)
                .setPositiveButton(getResources().getString(R.string.bouton_positif),
```

```
                        new DialogInterface.OnClickListener() {

                            // méthode invoquée lorsque l'utilisateur validera la saisie
                            public void onClick(DialogInterface dialog,
                                                int leBouton) {

                                // on récupère un accès sur les zones de saisies de la boîte
                                EditText nomPays =
                                        (EditText) boiteSaisie.findViewById(R.id.saisie_nom_pays);
                                EditText nomCapitale =
                                        (EditText) boiteSaisie.findViewById(R.id.saisie_capitale);
                                EditText annee =
                                        (EditText) boiteSaisie.findViewById(R.id.saisie_annee);

                                // préparation d'un nouvel enregistrement pour la base de données
                                accesPays.insertPays(
                                        new Pays(nomPays.getText().toString(),
                                                 nomCapitale.getText().toString(),
                                                 Integer.parseInt(annee.getText().toString())));


                                // on met à jour le curseur en refaisant la requete de sélection
                                curseurSurBase = accesPays.getCurseurPays();

                                // on met à jour l'adapteur et on informe la liste du changement
                                adaptateur.swapCursor(curseurSurBase);
                                onContentChanged();
                            }
                        })
                .setNegativeButton(getResources().getString(R.string.bouton_negatif), null)
                .show();
    }


    /**
     * Cette méthode affiche une boîte de dialogue permettant à l'utilisateur de
     * saisir le nom d'un pays. Ce pays est ensuite supprimé de la base
     */
    private void saisirSupprimerPays() {

        // on désérialise le layout qui est associé à la boîte de saisie
        final View boiteSaisie = getLayoutInflater().inflate(R.layout.saisie_nom, null);

        // Création d'une boîte de dialogue
        new AlertDialog.Builder(this)
                .setTitle(getResources().getString(R.string.titre_boite_saisie_nom))
                .setView(boiteSaisie)
                .setPositiveButton(getResources().getString(R.string.bouton_positif),
                        new DialogInterface.OnClickListener() {

                            // méthode invoquée lorsque l'utilisateur validera la saisie
                            public void onClick(DialogInterface dialog,
                                                int leBouton) {

                                // on récupère un accès sur la zone de saisie de la boîte de dialogue
                                EditText nomPays =
                                        (EditText) boiteSaisie.findViewById(
                                                R.id.saisie_le_nom_a_supprimer);

                                // suppression de l'enregistrement dans la base
                                int resultat = accesPays.deletePays( nomPays.getText().toString());

                                // résultat est égal au nombre de lignes supprimées
                                if (resultat != 0) {

                                    /*
                                     *  une suppression a eu lieu : on refait la requete
                                     *  de sélection pour actualiser le curseur. L'adaptateur
```

```
                                 *  est modifié avec le nouveau curseur, et la liste
                                 *  des pays est également actualisée
                                 */
                                curseurSurBase = accesPays.getCurseurPays();
                                adaptateur.swapCursor(curseurSurBase);
                                onContentChanged();
                            } else {

                                // aucun suppression effectuée : le pays n'existe pas
                                Toast.makeText(ActivitePrincipale.this,
                                        R.string.messageToast, Toast.LENGTH_LONG)
                                        .show();
                            }
                        }
                    })
            .setNegativeButton(getResources().getString(R.string.bouton_negatif), null)
            .show();
    }


    /**
     * Cette méthode affiche une boîte de dialogue permettant à l'utilisateur de
     * modifier l'année d'adhésion d'un pays.
     */
    private void saisirModifierPays() {

        // on désérialise le layout qui est associé à la boîte de saisie
        final View boiteSaisie = getLayoutInflater().inflate(R.layout.saisie_nom_annee, null);

        // Création d'une boîte de dialogue
        new AlertDialog.Builder(this)
                .setTitle(getResources().getString(R.string.titre_boite_modif_annee))
                .setView(boiteSaisie)
                .setPositiveButton(getResources().getString(R.string.bouton_positif),
                        new DialogInterface.OnClickListener() {

                            // méthode invoquée lorsque l'utilisateur validera la saisie
                            public void onClick(DialogInterface dialog,
                                                int leBouton) {

                                // on récupère un accès aux zones de saisie
                                EditText nomPays =
                                        (EditText) boiteSaisie.findViewById(
                                                R.id.saisie_pays_a_modifier);
                                EditText nouvelleAnnee =
                                        (EditText) boiteSaisie.findViewById(
                                                R.id.saisie_nouvelle_annee);

                                //  modification de l'enregistrement dans la base
                                int resultat = accesPays.updatePays(
                                        nomPays.getText().toString(),
                                        Integer.parseInt(nouvelleAnnee.getText().toString()));

                                // résultat est égal au nombre de lignes supprimées
                                if (resultat != 0) {
                                    curseurSurBase = accesPays.getCurseurPays();
                                    adaptateur.swapCursor(curseurSurBase);
                                    onContentChanged();
                                } else {

                                    // aucun modification effectuée : le pays n'existe pas
                                    Toast.makeText(ActivitePrincipale.this,
                                            R.string.messageToast, Toast.LENGTH_LONG)
                                            .show();
                                }
                            }
                        })
                .setNegativeButton(getResources().getString(R.string.bouton_negatif), null)
```

```
                .show();
    }


    /**
     * Cette méthode affiche une boîte de dialogue permettant à l'utilisateur de
     * modifier l'année d'adhésion d'un pays.
     */
    private void saisirSelectionAnnee() {

        // on désérialise le layout qui est associé à la boîte de saisie
        final View boiteSaisie = getLayoutInflater().inflate(R.layout.layout_selection, null);

        // Création d'une boîte de dialogue
        new AlertDialog.Builder(this)
                .setTitle(getResources().getString(R.string.titre_boite_modif_annee))
                .setView(boiteSaisie)
                .setPositiveButton(getResources().getString(R.string.bouton_positif),
                        new DialogInterface.OnClickListener() {

                            // méthode invoquée lorsque l'utilisateur validera la saisie
                            public void onClick(DialogInterface dialog,
                                                int leBouton) {

                                // on récupère un accès aux zones de saisie
                                EditText annee =
                                        (EditText) boiteSaisie.findViewById(
                                                R.id.saisie_choix_annee);
                                int anneeChoisie = Integer.parseInt(annee.getText().toString());
                                //  modification de l'enregistrement dans la base

                                if (anneeChoisie == 0) {   // afficher tous les pays
                                    curseurSurBase = accesPays.getCurseurPays();
                                } else {
                                     curseurSurBase =
                                             accesPays.getCurseurPaysSelonAnnee(anneeChoisie);


                                }
                                adaptateur.swapCursor(curseurSurBase);
                                onContentChanged();
                            }
                        })
                .setNegativeButton(getResources().getString(R.string.bouton_negatif), null)
                .show();
    }

    @Override
    public void onDestroy() {
        accesPays.close();
        super.onDestroy();
    }
}
```

Classe *ActivitePrincipale.java*

VI) Le pattern singleton

Le plus souvent, plusieurs activités d'une même l'application vont devoir accéder à la base de données associée à l'application. Pour garantir l'unicité de celle-ci au sein de l'application, il faut utiliser un *pattern* bien précis : le *pattern* ***singleton.***

Le *pattern singleton*

Le *pattern singleton* permet de garantir qu'une seule instance d'une classe précise sera créée et qu'un unique point d'accès à cette instance existera dans l'application. Ce *pattern* est tout indiqué pour gérer la connexion à une base de données.

Pour garantir l'unicité de l'instance de la classe, il faut contraindre la création de celle-ci. On interdit donc l'utilisation de l'opération *new* à l'extérieur de la classe. Pour ce faire, il faut que les **constructeurs** de la classe soient **privés**.

Pour obtenir une instance de la classe, le programme utilisateur devra invoquer une méthode outil (une méthode de classe donc statique) à définir d'une manière bien précise. Celle-ci vérifiera si une instance de la classe existe déjà ou pas. Si une telle instance n'existe pas encore, elle l'a créera. La méthode renverra dans tous les cas l'instance unique associée à la classe. Notons bien sûr que cette instance est statique.

```
/**
 * Implémentation simple d'un singleton.
 * L'instance est créée lors de l'appel à la méthode getInstance,
 * et pas par un appel au constructeur
 */
public class Singleton {
    /**
     * Constructeur privé
     * (dans cet exemple basique : ne fait rien)
     **/
    private Singleton() {
    }

    /** Instance unique de type Singleton associé à la classe */
    private static Singleton instance;

    /**
     * Point d'accès pour l'instance unique du singleton
     * @return une instance de type Singleton
     **/
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Exemple de mise en œuvre du *pattern singleton*

```
public class HelperBDPaysUE extends SQLiteOpenHelper {

    /** Nom de la base de données qui contiendra les pays */
    private static final String NOM_BD = "lespaysue.db";

    /** Version de la base de données */
    private static final int VERSION_BD = 1;

    . . .


    /** Instance de HelperBDPaysUE : elle sera unique au sein de l'application */
    private static HelperBDPaysUE instanceHelperBDPays;



    /**
     * Création si besoin d'une instance de type HelperBDPaysUE,
     * et renvoie de celle-ci
     * @param context  contexte de l'activité à l'origine de l'appel
     * @return  une instance de type HelperBDPaysUE pour accéder ensuite à la base de
     *          données
     */
    public static synchronized HelperBDPaysUE getInstance(Context context) {

        if (instanceHelperBDPays == null) {

            // l'instance n'existe pas encore : on la crée
            instanceHelperBDPays = new HelperBDPaysUE(context.getApplicationContext());
        }
        return instanceHelperBDPays;
    }


    /**
     * Le constructeur est privé pour empêcher qu'il soit appelé par les autres classes.
     * (celles-ci invoqueront getInstance() )
     * On est donc sûr qu'une unique instance de type HelperBDPaiement existera au sein de
     * l'application.
     */
    private HelperBDPaysUE(Context context) {
        super(context, NOM_BD, null, VERSION_BD);
    }

    . .  .
```

Dans la classe ci-dessus, le pattern singleton a été utilisé. Il faut donc appeler la méthode statique ***getInstance*** au lieu du constructeur pour obtenir une instance de type *HelperBDPaysUE*. On procède de même dans la classe ***PaysDAO*** pour garantir l'unicité de l'instance de type ***PaysDAO*** au sein de l'application.

```
public class PaysDAO {

    /** Numéro de version de la base de donnée */
    private static final int VERSION = 1;

    . . .

    /** Gestionnaire permettant de créer la base de donnée */
    private HelperBDPaysUE gestionnaireBase;

    /** Base de données contenant la description des pays */
    private SQLiteDatabase basePays;

    /** Instance de PaysDAO : elle sera unique au sein de l'application */
    private static PaysDAO instanceDAO;


    /**
     * Création si besoin d'une instance de type PaysDAO,
     * et renvoie de celle-ci
     * @param context  contexte de l'activité à l'origine de l'appel
     * @return  une instance de type PaysDAO
     */
    public static synchronized PaysDAO getInstance(Context context) {
        if (instanceDAO == null) {
            instanceDAO = new PaysDAO(context.getApplicationContext());
        }
        return instanceDAO;
    }


    /**
     * Constructeur de la base
     * Le constructeur est privé pour empêcher qu'il soit appelé par les autres classes.
     * (celles-ci invoqueront getInstance() )
     * On est donc sûr qu'une unique instance de type HelperBDPaiement existera au sein de
     * l'application.
     * @param context  contexte de création de la base (activité créatrice)
     */
    private PaysDAO(Context context) {
        gestionnaireBase = HelperBDPaysUE.getInstance(context);
    }


    /**
     * Ouverture de la table des pays
     */
    public void open() {
        if (basePays == null) {
            basePays = gestionnaireBase.getWritableDatabase();
        }
    }

    /**
     * Fermeture de la table des pays
     */
    public void close() {
        if (gestionnaireBase != null) {
            gestionnaireBase.close();
        }
        if (basePays != null) {
            basePays.close();
        }
    }

    . . .
```

VII) Quelques compléments

Temps d'accès à une base de données

La base de données est stockée dans la mémoire flash du terminal. Les accès en lecture sont relativement rapides, mais bien sûr tout dépend du volume des données lues. Par contre les temps d'accès en écriture sont variables selon l'état de la mémoire, et peuvent être très longs. Dans tous les cas, et comme dit précédemment il faut prendre en considération les temps d'accès à une base de données, et chercher à les réduire autant que possible. Une première amélioration consiste à utiliser la notion de transaction.

Notion de transaction sur une base de données

Lorsque plusieurs opérations doivent être exécutées sur la base de données, il est recommandé de réaliser des traitements par lot, ou transactions. Rappelons qu'une transaction est atomique : soit elle est réalisée avec succès, soit elle n'est pas réalisée. C'est un premier avantage d'une transaction. Le deuxième est le gain en rapidité d'exécution. Par exemple, si l'on souhaite ajouter 100 enregistrements à une table, si on place les requêtes d'insertion dans une transaction, on pourra observer un gain de rapidité de 50 % (voir le livre « *Android Database Best Practices* » de Adam Stroud).

La classe SQLiteDatabase propose 3 méthodes qui permettent de réaliser des transactions :

void beginTransaction()	initie la transaction
void setTransactionSuccessful()	demande la réalisation de la transaction, si elle peut être réalisée avec succès.
void endTransaction()	termine la transaction, après l'avoir réalisée dans le cas où elle a été évaluée comme réalisable avec succès

Généralement le schéma de code à respecter sera le suivant :

```
db.beginTransaction();
try {

  // on code ici les opérations à réaliser sur la base de données
  ...
  db.setTransactionSuccessful();
} finally {

  // on est sûr que le bloc finally est executé dans tous les cas
  db.endTransaction();
}
```

Optimisation du temps d'exécution pour l'ouverture et la fermeture de la base

Notons que le constructeur de la classe qui hérite de *SQLiteDatabase* est assimilé à un proxy qui s'exécute rapidement. En effet, la base sera effectivement créée lors de l'appel à ***getWritableDatabase*** ou ***getReadableDatabase.*** Ces deux méthodes par contre sont longues d'un point de vue temps d'exécution. Il est donc recommandé de placer leur appel dans un *thread* dédié (voir un autre chapitre pour la notion de *thread*, de tâche asynchrone ou de service).

Pour cette même raison, on évite de fermer une base de données si l'on sait qu'il sera nécessaire de l'ouvrir à nouveau ultérieurement. Il est donc conseillé de fermer la base de données dans la méthode *onDestroy()* de la classe activité. Nous pourrions donc compléter la classe exemple de la manière suivante :

```
@Override
public void onDestroy() {
    gestionnaireBase.close();
    super.onDestroy();
}
```

La classe *CursorLoader*

Comme évoqué précédemment, il est conseillé d'effectuer les opérations sur une base de données dans un *thread* dédié (voir plus loin pour cette notion). Pour faciliter la tâche du programmeur, il existe une classe nommée ***CursorLoader*** (chargeur de curseur) qui permet de constituer un curseur de manière asynchrone. Pour le programmeur, il sera plus simple d'utiliser cette classe plutôt que de gérer lui-même un *thread* dédié aux accès à la base de données. L'utilisation de la classe ***CursorLoader*** sera illustrée dans le chapitre consacré aux fournisseurs de contenu.

La classe *SQLiteQueryBuilder*

Il existe une classe ***SQLiteQueryBuilder*** pour faciliter la construction des requêtes complexes. L'une de ses méthodes contient des arguments pour spécifier les clauses **groupBy**, **having** et **sortOrder** d'une instruction **select.**

Consultation et manipulation directe de la base de données

Si l'application est installée sur l'émulateur, il est possible d'accéder à cette base à partir d'Android Studio.

On peut utiliser un utilitaire (à télécharger gratuitement) pour visualiser le contenu de la base et éventuellement la modifier : **SQLiteBrowser**.

Communication entre les activités

I) Introduction

Une application est le plus souvent constituée de plusieurs activités distinctes. Ceci implique qu'une activité puisse en lancer une autre. Par exemple, dans une application destinée à gérer un calendrier, si dans la vue qui présente tout le calendrier (donc une vue gérée par une première activité), l'utilisateur clique sur un événement précis, l'activité courante doit lancer une deuxième activité, celle qui gère la vue affichant tout le détail de l'événement. Pour ce faire, la première activité devra envoyer une **intention** à la deuxième.

Supposons qu'une activité A souhaite lancer une activité B. Plusieurs cas sont envisageables :
- A lance B et A n'a pas besoin de savoir quand l'activité B va se terminer. Eventuellement A communique des informations à B.
- A lance B et souhaite être informée quand la sous-activité B sera terminée. Eventuellement B devra communiquer à A des informations.

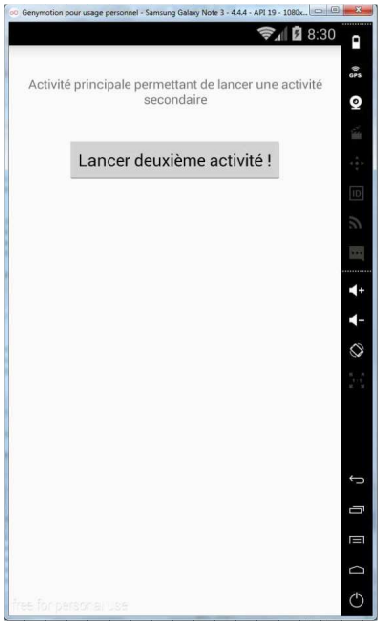
Exemple

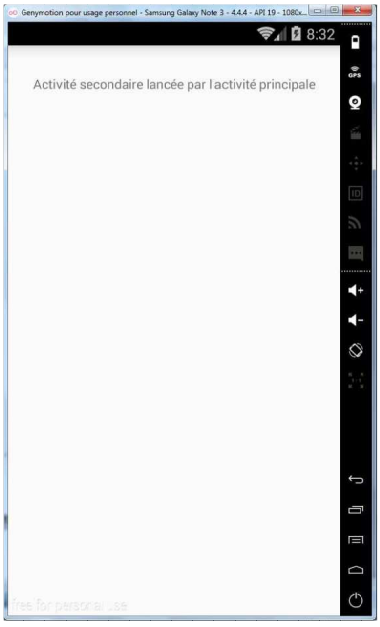
L'activité principale de l'application affiche un bouton permettant à l'utilisateur de lancer une deuxième activité (activité fille).

Les 2 activités n'échangent aucune information.

L'activité principale, lors du clic sur le bouton, doit créer une **intention** contenant les informations utiles au lancement de la deuxième activité.

Elle doit ensuite demander le démarrage de la deuxième activité via l'intention préalablement créée.





Nous devons coder 2 classes activités : *MainActivity* et *ActiviteSecondaire*. Dans la méthode de la classe *MainActivity* qui gère le clic sur le bouton , nous créons et lançons l'intention de la manière suivante :

```
   // création d'une intention pour demander lancement de l'activité secondaire
   Intent secondeActivite =
                        new Intent(MainActivity.this, ActiviteSecondaire.class);

    // lancement de l'activité secondaire via l'intention préalablement créée
    startActivity(secondeActivite);
```

Le code complet des deux activités est le suivant :

```
/*  Activité principale de l'exemple 1 illustrant la communication entre activités
    MainActivity.java                                                    07/16
*/
package com.example.cours.android.exmultiactivitesanscommunication;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;

/**
 * Activité principale de l'application permettant d'illustrer la communication
 * entre activités
 * Exemple 1 : une activité parente (activité principale) lance une activité
 *             secondaire
 *             Il n'y a pas d'échange d'informations entre les 2 activités
 * La vue de l'activité contient un bouton sur lequel l'utilisateur doit cliquer
 * pour lancer l'activité secondaire
 * @author C. Servières
 * @version 1.0
 */
public class MainActivity extends Activity {

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

    /* Méthode invoquée lors du clic sur le bouton de création de l'activité
     * secondaire. Celle-ci est créée et lancée.
     * @param bouton  bouton dont la méthode intercepte le clic
     */
    public void clicBouton(View bouton) {

        // création d'une intention pour demander lancement de l'activité secondaire
        Intent secondeActivite =
                new Intent(MainActivity.this, ActiviteSecondaire.class);

        // lancement de l'activité secondaire via l'intention préalablement créée
        startActivity(secondeActivite);
    }
}
```

Classe de l'activité principale

```
/*  Activité secondaire de l'exemple 1 illustrant la communication entre activités
    ActiviteSecondaire.java                                                 07/16
*/
package com.example.cours.android.exmultiactivitesanscommunication;

import android.app.Activity;
import android.os.Bundle;

/**
 * Activité secondaire de l'application permettant d'illustrer la communication
 * entre activités
 * Exemple 1 : une activité parente (activité principale) lance une activité
 *             secondaire
 *             Il n'y a pas d'échange d'informations entre les 2 activités
 * La vue de cette activité affiche seulement un texte
 * @author C. Servières
 * @version 1.0
 */
public class ActiviteSecondaire extends Activity {

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

Classe de l'activité secondaire

II) Lancement d'une activité fille sans attente d'un retour

Pour lancer l'activité B, l'activité A devra créer une intention, et ensuite la lancer.

2.1) Les intentions

Une **intention** (ou ***intent***) est en fait un objet de la classe ***Intent*** constitué de sous-informations :

- un composant
- une action (sous-information obligatoire)
- une catégorie
- un ensemble de données et un type pour chacune d'elles
- des extras
- des flags

Le champ **composant** peut être renseigné ou pas. S'il est renseigné, l'intention sera dite explicite. Il définit alors le destinataire de l'intention, celui qui devra la gérer. Il est constitué de 2 informations : le package où se situe le composant, et le nom du composant lui-même. Ainsi, Android pourra retrouver exactement le composant destinataire de l'intention.

A l'opposé, si l'intention est implicite, ce champ composant ne sera pas renseigné. En effet, dans ce cas, le destinataire précis n'est pas connu. D'autres champs de l'intention seront renseignés pour donner à

Android les informations nécessaires pour qu'il détermine quel composant est capable de traiter l'intention.

Il faut au moins préciser **l'action** que l'on désire que le destinataire fasse.

Les autres informations sont facultatives :

- La **catégorie** permet d'apporter des informations supplémentaires sur l'action à exécuter et le type de composant qui devra gérer l'intention (voir plus de détails dans la section sur les intentions implicites)

- Les **données** sont celles utiles au destinataire pour qu'il effectue l'action. Le type est normalement contenu dans les données, mais en précisant cette information il est possible de désactiver le type par défaut et d'imposer un type particulier (voir plus de détails dans la section sur les intentions implicites)

- Les **extras** permettent d'ajouter du contenu aux intentions et donc de transmettre celui-ci aux autres composants (voir ci-dessous)

- Les **flags** permettent de modifier le comportement de l'intention.

2.2) Dans le code Java, comment créer et lancer d'une intention ?

Pour représenter une intention, Java fournit la classe ***Intent***. Avant de lancer une intention, il faut d'abord la créer, en invoquant le constructeur de la classe ***Intent***. Par exemple :

```
Intent intention = new Intent (A.this, B.class);
```

Le profil de ce constructeur est :

```
Intent(Context packageContext, Class<?> activityToLaunch)
```

Le contexte est celui de l'activité courante. Le deuxième argument est une référence vers la classe cible, destinataire de l'intention.

Pour lancer l'intention, on fait appel à la méthode `startActivity(Intent intention)` de la classe ***Activity*** :

```
startActivity(intention);
```

La nouvelle activité (donc *B*) démarrera dans un cycle normal, en commençant par un appel automatique à la méthode ***onCreate***.

2.3) Remarque importante

Il ne faut pas oublier de déclarer toutes les activités de l'application dans le fichier *AndroidManifest.xml*. Notons que, en général, Android Studio, met à jour automatiquement ce fichier, lorsque que l'on créé la nouvelle activité. La nouvelle activité doit apparaître dans la balise `<application>`, avec au minimum les informations suivantes :

```
<activity android:name=".B"></activity>
```

2.4) Dans le code Java, comment transmettre des données à l'activité fille ?

Cette transmission se fait via les extras de l'objet de type ***Intent***. Un extra est en fait un couple clé/valeur qui s'appuie sur le même principe que l'insertion de données dans un objet de type *Bundle* (voir chapitre sur le Cycle de vie d'une activité). Une autre possibilité pour transmettre les informations consisterait à utiliser un fournisseur de contenu ou *Content Provider* (voir plus loin).

Pour insérer des extras dans l'intention, on utilise la méthode suivante (définie dans la classe ***Intent***) :

```
Intent putExtra(String key, XXX value)
```

avec *key* la clé de l'extra, et *value* la valeur associée. Le type de celle-ci (noté XXX) peut être un type de base du langage Java (int, String, float, double, byte) y compris un tableau d'éléments de ces types de base. Si le type de la valeur n'est pas de l'un de ces types, il faut faire appel à l'interface *Parcelable* (voir les détails dans une autre section).

Exemple :

Supposons que *login* et *password* soient 2 chaînes de caractères (donc de type *String*) que l'on souhaite transmettre avec l'intention, et que l'on ait défini 2 constantes pour les noms des clés :

```
public static final String EXTRA_LOGIN = "login";
public static final String EXTRA_PASSWORD = "password";
```

Une bonne habitude, pour éviter les problèmes d'homonymie entre les clés, consistera à préfixer le nom de la clé par le nom du package à l'origine de l'intention : "*com.exemple.intention*", ce qui donnerait :

```
private static final String EXTRA_LOGIN = "com.exemple.intention.login";
private static final String EXTRA_PASSWORD = " com.exemple.intention.password";
```

On pourra transmettre, à l'activité fille, les 2 chaînes de la manière suivante :

```
intention.putExtra(EXTRA_LOGIN, login);
intention.putExtra(EXTRA_PASSWORD, password);
startActivity(intention);
```

2.5) Comment, dans le code de l'activité fille, récupérer les informations transmises via l'intention ?

Si l'activité fille souhaite récupérer des données en provenance de l'activité parente, elle devra d'abord récupérer l'intention. Pour ce faire, il faut utiliser la méthode

`getIntent()` Cette méthode est définie de la classe ***Activity***.

Elle renvoie l'intention qui est à l'origine de la création de l'activité. L'appel à ***getIntent*** figurera, le plus souvent dans la méthode *onCreate* de l'activité fille.

Ensuite, on invoque, sur l'instance de type ***Intent*** renvoyée par ***getIntent***, les méthodes permettant d'accéder aux extras véhiculés avec l'intention. Ces méthodes sont définies dans la classe ***Intent*** :

`Bundle getExtras()`	on accède ainsi à tous les couples clé-valeurs contenus dans le *Bundle*
`X getXXXExtra(String key)`	où XXX est en fait le type de la donnée à récupérer, par exemple *int*, *String*, *FloatArray* ...
`X getXXXExtra(String key, X defaultValue)`	*defaultValue* est la valeur par défaut renvoyée si la clé n'existe pas.

Exemple pour récupérer le login et le mot de passe :

```
Intent intention = getIntent();
String texteLogin = intention.getStringExtra(EXTRA_LOGIN);
String textePassword = intention.getStringExtra(EXTRA_PASSWORD);
```

III) Lancement d'une activité fille avec attente d'un retour

Une activité peut en lancer une autre et attendre le résultat de celle-ci. Pour ce faire, nous devons écrire les instructions adéquates dans l'activité parente et dans l'activité fille.

3.1) Activité parente

Dans l'activité parente, l'activité fille doit être démarrée par un appel la méthode :

```
startActivityforResult(Intent intentionLancement, int codeRequete)
```

Cette méthode est définie dans la classe *Activity*. Le code requête servira à faire le lien entre la requête et la réponse.

L'activité parente doit aussi définir une méthode `onActivityResult` qui sera appelée automatiquement lors du retour de l'activité fille :

```
protected void onActivityResult( int codeRequete, int codeResultat,
                                 Intent intentionResultat)
```

- ✓ Le code requête doit à priori, être le même que celui qui a été communiqué lors du lancement de l'activité fille.
- ✓ Le code résultat est celui renvoyé par l'activité fille, et sert à indiquer si l'opération demandée s'est correctement déroulée ou pas.

- ✓ L'intention résultat est celle émise par l'activité fille pour effectuer le retour vers l'activité parente. Cette intention est en fait facultative.

Si la méthode ***onActivityResult*** est bien écrite, elle doit vérifier les valeurs de ces codes.

3.2) Activité fille

1) L'activité fille doit déclarer qu'elle envoie un résultat à l'activité parente. 2 méthodes peuvent être invoquées :

```
setResult(int codeResultat)

setResult(int codeResultat , Intent intentionResultat)
```

invoquée si l'activité fille souhaite communiquer des informations à l'activité parente

Le code résultat sera souvent : `Activity.RESULT_OK` ou `Activity.RESULT_CANCELED`. Le programmeur peut aussi définir ses propres codes.

2) Ensuite l'activité fille doit invoquer la méthode `finish()` pour se terminer.

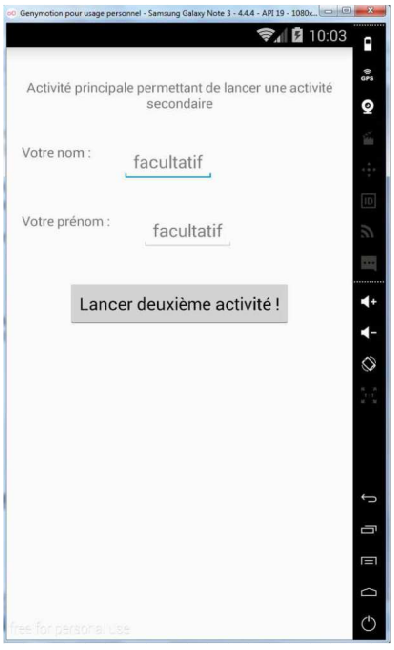
Remarque : si l'utilisateur utile la touche "*back*" du terminal pour revenir à l'activité parente, un appel à `setResult(Activity.RESULT_CANCELED)` sera effectué automatiquement.

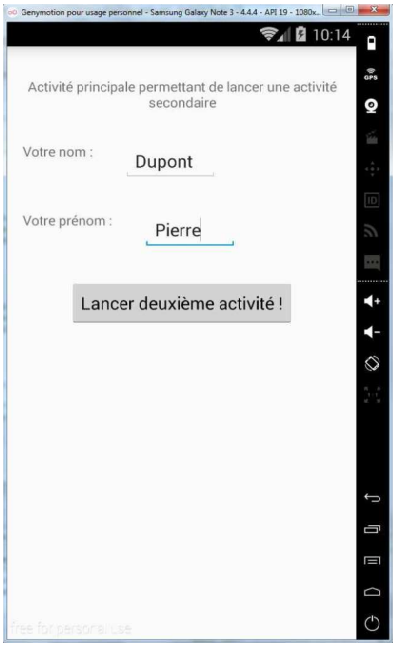
3.3) Exemple

Dans l'activité principale, l'utilisateur est invité à saisir son nom et son prénom (facultatifs), puis à cliquer sur le bouton de création de l'activité secondaire. Le nom et le prénom sont communiqués à l'activité secondaire. Une fois celle-ci lancée, l'activité principale attend un retour de la part de l'activité secondaire.

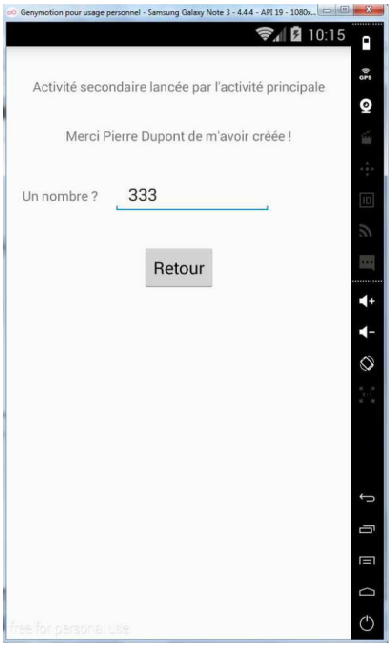
L'activité secondaire affiche le nom et le prénom de l'utilisateur, et l'invite à saisir un nombre entier. Quand ce dernier clique sur le bouton "retour", on revient à l'activité principale tout en communiquant à celle-ci le nombre saisi.

Le nombre saisi est alors affiché par l'activité principale.

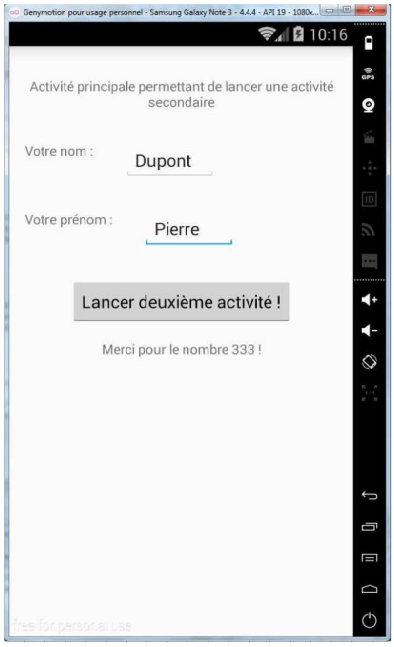




Activité principale



Activité secondaire



Retour activité principale

Le code Java de l'activité principale est le suivant :

```
/*  Activité principale de l'exemple 2 illustrant la communication entre activités
    MainActivity.java                                                    07/16
*/
package com.example.cours.android.exmultiactiviteaveccommunication;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;
import android.widget.TextView;

/**
 * Activité principale de l'application permettant d'illustrer la communication
 * entre activités
 * Exemple 2 : une activité parente (activité principale) lance une activité
 *             secondaire
 *             Les 2 activités échangent des informations
 * La vue de l'activité contient un bouton sur lequel l'utilisateur doit cliquer
 * pour lancer l'activité secondaire.
 * Au préalable, l'utilisateur est invité à saisir son nom et son prénom.
 * S'il donne ces informations, elles seront communiquées à l'activité secondaire.
 * Au retour de l'activité secondaire, l'entier que l'utilisateur aura saisi
 * dans celle-ci sera affiché.
 * @author C. Servières
 * @version 1.0
 */
public class MainActivity extends Activity {

    /** Clé pour le nom transmis à l'activité secondaire */
    public final static String CLE_NOM = "com.example.cours.android.NOM";

    /** Clé pour le prenom transmis à l'activité secondaire */
    public final static String CLE_PRENOM = "com.example.cours.android.PRENOM";
```

```
    /** Clé pour le nombre transmis par l'activité secondaire */
    public final static String CLE_NOMBRE = "com.example.cours.android.NOMBRE";


    /** Identifiant de l'intention pour demander la saisie d'un nombre*/
    public final static int CHOIX_NOMBRE = 0;


    /** Zone de saisie du nom (facultatif) */
    private EditText saisieNom;

    /** Zone de saisie du prénom (facultatif) */
    private EditText saisiePrenom;

    /** Texte résultat affiché au retour de l'activité secondaire */
    private TextView texteResultat;

    /** Texte à remplir qui correspond au résultat */
    private String texteARemplir;


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

        saisieNom = (EditText) findViewById(R.id.saisie_nom);
        saisiePrenom = (EditText) findViewById(R.id.saisie_prenom);
        texteResultat = (TextView) findViewById(R.id.zone_resultat);

        texteARemplir = texteResultat.getText().toString();
    }


    /* Méthode invoquée lors du clic sur le bouton de création de l'activité
     * secondaire. Celle-ci est créée et lancée.
     * @param bouton  bouton dont la méthode intercepte le clic
     */
    public void clicBoutonLancer(View bouton) {

        // on accède aux nom et prénom saisis par l'utilisateur (éventuellement)
        String leNom = saisieNom.getText().toString();
        String lePrenom = saisiePrenom.getText().toString();

        // création d'une intention pour demander lancement de l'activité secondaire
        Intent intention =
                new Intent(MainActivity.this, ActiviteSecondaire.class);

        // on ajoute le nom et le prénom en tant qu'extra
        intention.putExtra(CLE_NOM, leNom);
        intention.putExtra(CLE_PRENOM, lePrenom);

        /* lancement de l'activité secondaire via l'intention préalablement créée
           on précise que l'on attend un retour de cette activité, et que son
           identifiant est CHOIX_NOMBRE */
        startActivityForResult(intention, CHOIX_NOMBRE);
    }


    /**
     * Méthode invoquée automatiquement lors du retour de l'activité secondaire
     */
    @Override
```

```
    protected void onActivityResult(int codeRequete, int codeResultat,
                                    Intent intention) {

        /* on vérifie qui l'activité qui renvoie son résultat est bien celle qui a
           l'identifiant CHOIX_NOMBRE
         */
        if (codeRequete == CHOIX_NOMBRE) {

            // si le code retour correspond à un résultat correct
            if (codeResultat == Activity.RESULT_OK) {
                texteResultat.setText(
                        String.format(texteARemplir,
                                      intention.getIntExtra(CLE_NOMBRE, 0)));
                texteResultat.setVisibility(View.VISIBLE);
            }
        }
    }
}
```

Activité principale MainActivity.java

Le code Java de l'activité secondaire est le suivant :

```
/*  Activité secondaire de l'exemple 2 illustrant la communication entre activités
    ActiviteSecondaire.java                                              07/16
*/
package com.example.cours.android.exmultiactiviteaveccommunication;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;
import android.widget.TextView;

/**
 * Activité secondaire de l'application permettant d'illustrer la communication
 * entre activités
 * Exemple 2 : une activité parente (activité principale) lance une activité
 *             secondaire
 *             Les 2 activités échangent des informations
 * Cette activité reçoit le nom de l'utilisateur saisi dans l'activité principale.
 * Elle demande à l'utilisateur de saisir un nombre.
 * Celui-ci est communiqué à l'activité principale, lorsque l'utilisateur clique
 * sur le bouton "retour".
 * @author C. Servières
 * @version 1.0
 */
public class ActiviteSecondaire extends Activity {

    /** Texte affiché qui contient le nom et le prénom provenant
     * de l'activité principale */
    private TextView messageMerci;

    /** Zone de saisie du nombre entier */
    private EditText saisieNombre;


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

```
        messageMerci = (TextView) findViewById(R.id.texte_merci);
        saisieNombre = (EditText) findViewById(R.id.saisie_nombre);

        // on récupère l'intention qui a lancé cette activité
        Intent intentionRecu = getIntent();

        // on récupère les 2 chaînes de caractères contenues dans l'intention
        String leNomRecu = intentionRecu.getStringExtra(MainActivity.CLE_NOM);
        String lePrenomRecu = intentionRecu.getStringExtra(MainActivity.CLE_PRENOM);

        // on modifie le texte du message de remerciement pour inclure nom et prénom
        String aAfficher = (leNomRecu.length() == 0 && lePrenomRecu.length() == 0 ?
                                "inconnu" : lePrenomRecu + " " +  leNomRecu);
        messageMerci.setText(String.format(messageMerci.getText().toString(),
                                aAfficher));
    }

    /* Méthode invoquée lors du clic sur le bouton de retour vers l'activité
     * principale.
     * @param bouton  bouton dont la méthode intercepte le clic
     */
    public void clicBoutonRetour(View bouton) {

        // on recupère le nombre saisi par l'utilisateur
        int leNombre;
        try {
            leNombre = Integer.parseInt(saisieNombre.getText().toString());
        } catch(NumberFormatException err) {
            leNombre = 0;
        }


        // création d'une intention pour informer l'activté parente
        Intent intentionRetour = new Intent();
        intentionRetour.putExtra(MainActivity.CLE_NOMBRE, leNombre);
        setResult(Activity.RESULT_OK, intentionRetour);
        finish();
    }
}
```

Activité secondaire ActiviteSecondaire.java

IV) Utiliser l'interface Parcelable

Nous avons vu comment communiquer, à une activité fille, des informations d'un type de base Java. Nous allons étudier maintenant comment communiquer des informations qui ne sont pas d'un type de base. En fait, ces informations seront d'abord placées dans un objet de type ***Parcel***, et c'est cet objet qui sera communiqué à l'activité fille, via l'intention.

Le *Bundle* ne peut gérer que des objets sérialisables. Pour Android, on considère qu'un objet est sérialisable à partir du moment où il implémente correctement l'interface ***Parcelable***.

L'interface ***Parcelable*** contient 2 méthodes publiques :

```
int describeContents()
```

Cette méthode permet de spécifier si l'instance de type ***Parcelable*** contient des objets spéciaux, comme par exemple un fichier, donc un objet de type *FileDescriptor*. Si ce n'est pas le cas, la méthode doit renvoyer la valeur 0. Sinon, elle peut renvoyer la constante `Parcelable.CONTENTS_FILE_DESCRIPTOR`.

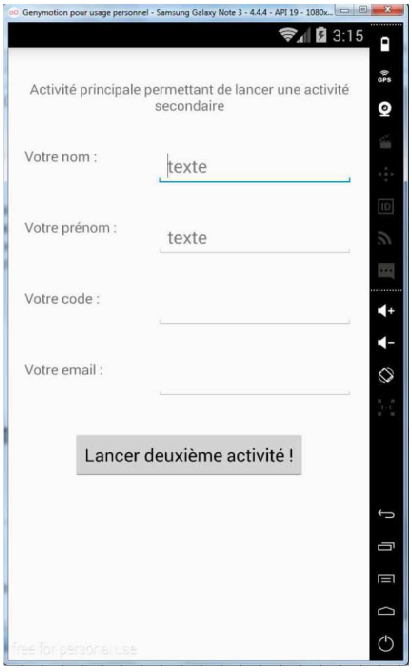
```
void writeToParcel (Parcel dest, int flags)
```

Le paramètre *dest* est l'instance de type ***Parcel*** dans lequelle il faut insérer les attributs de l'objet que l'on souhaite communiquer via l'intention. Le paramètre *flags* est un entier le plus souvent égal à 0. On peut aussi lui donner la valeur `Parcelable.PARCELABLE_WRITE_RETURN_VALUE`

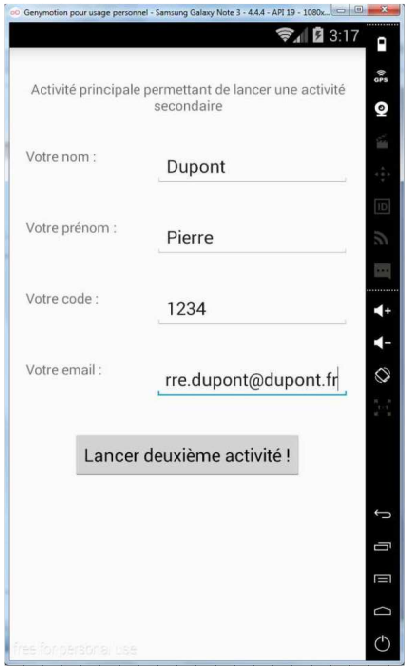
Attention : il faut insérer dans l'instance de type ***Parcel*** les attributs dans l'ordre dans lequel ils sont déclarés dans la classe.

Exemple

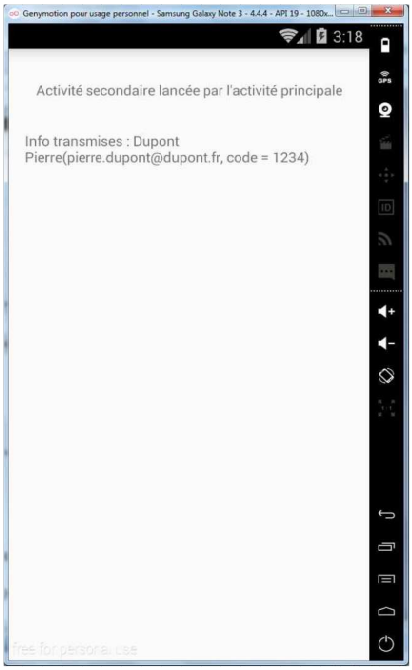
Supposons que l'on demande à l'utilisateur de saisir le nom, le prénom, un code (entier) et l'adresse mail d'une personne (pour s'authentifier, par exemple). Une activité spécifique sera chargée d'effectuer cette saisie. Les informations recueillies seront ensuite transmises à une activité fille.



Activité principale



Activité principale



Activité secondaire

Une possibilité consisterait à communiquer 3 chaînes de caractères, et 1 entier, en effectuant 4 fois un appel à la méthode *putExtra*, sur l'intention. Une variante consiste à communiquer les 4 informations ensemble dans un objet de type ***Parcelable*** spécifique.

Pour ce faire, il faut d'abord créer une classe nommée *InfoAuthentification* regroupant les 3 chaînes et l'entier, définis en tant qu'attributs, et implémentant l'interface ***Parcelable***. C'est une instance de la classe *InfoAuthentification* qui sera transmise à l'activité fille, via l'intention qui l'a créera. Nous notons que cette classe implémente l'interface ***Parcelable***, et redéfinit donc les méthodes : ***describeContents*** et ***writeToParcel*.**

Nous devons aussi prévoir dans cette classe un attribut statique de type `Parcelable.Creator` portant impérativement le nom **CREATOR**. Cet attribut permettra ensuite de reconstruire un objet de type *InfoAuthentification* à partir d'un objet de type ***Parcel***. Cette reconstruction se fera en invoquant la méthode ***createFromParcel.*** Noter aussi la présence de la méthode ***newArray*** permettant de créer un tableau d'objets de type *InfoAuthentification.*

Enfin dans la classe *InfoAuthentification,* nous définissons aussi un constructeur permettant de créer un objet de type *InfoAuthentification* à partir d'un argument de type ***Parcel***. Les valeurs véhiculées via l'objet de type ***Parcel*** sont recopiées dans chacun des attributs de l'objet courant.

```
/* Classe utilisé dans l'exemple 3 pour illustrer la communication entre activités */
package com.example.cours.android.exmultiactiviteavecparcealable;

import android.os.Parcel;
import android.os.Parcelable;

/**
 * Cette classe regroupe en tant qu'attribut les informations saisies par
 * l'utilisateur dans une première activité et qui seront transmises
 * à une activité secondaire sous la forme d'un objet de type Parcelable.
 * @author C. Servières
 * @version 1.0
 */
public class InfoAuthentification implements Parcelable {

    /** nom de l'utilisateur */
    private String nom;

    /** prénom de l'utilisateur */
    private String prenom;

    /** Adresse électronique de l'utilisateur */
    private String adresse;

     /** Code de l'utilisateur */
    private int code;


    /**
     * Constructeur avec valeurs des attributs en argument
     * @param leNom      nom de l'utilisateur
     * @param lePrenom   prénom de l'utilisateur
     * @param lAdresse   adresse électronique de l'utilisateur
     * @param leCode     code de l'utilisateur (un entier)
     */
    public InfoAuthentification(String leNom, String lePrenom, String lAdresse,
                                int leCode) {
        nom = leNom;
        prenom = lePrenom;
        adresse = lAdresse;
        code = leCode;
```

```
    }

    @Override
    public String toString() {
        return nom + " " + prenom + "(" + adresse + ", code = " + code + ")";
    }
    @Override
    public int describeContents() {

        // on renvoie 0, car la classe ne contient d'attribut pas fichier
        return 0;
    }

    @Override
    public void writeToParcel(Parcel dest, int flags) {

        // on ajoute les objets attributs dans l'ordre dans lequel ils sont déclarés,
        // à l'objet dest
        dest.writeString(nom);
        dest.writeString(prenom);
        dest.writeString(adresse);
        dest.writeInt(code);
    }

    /**
     * Création, en tant que constante, d'un objet de type Parcelable.Creator
     * Cet objet assure la conversion d'un objet de type Parcel en un objet
     * de type InfoAuthentification
     */
    public static final Parcelable.Creator<InfoAuthentification> CREATOR =
            new Parcelable.Creator<InfoAuthentification>() {

                @Override
                public InfoAuthentification createFromParcel(Parcel source) {
                    return new InfoAuthentification(source);
                }

                @Override
                public InfoAuthentification[] newArray(int size) {
                    return new InfoAuthentification[size];
                }
            };


    /**
     * Constructeur pour initialiser les attributs avec les informations de l'objet
     * de type Parcel argument
     * @param in  instance de type Parcel contenant les informations pour initialiser
     *            l'objet courante
     */
    public InfoAuthentification(Parcel in) {
        nom = in.readString();
        prenom = in.readString();
        adresse = in.readString();
        code = in.readInt();
    }
}
```

Classe *InfoAuthentification* (schéma à respecter pour obtenir un objet *Parcelable*)

Définissons maintenant l'activité principale.

```java
/*  Activité principale de l'exemple 3 illustrant la communication entre activités
    MainActivity.java                                                    07/16
*/

package com.example.cours.android.exmultiactiviteavecparcealable;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;

/**
 * Activité principale de l'application permettant d'illustrer la communication
 * entre activités
 * Exemple 3 : une activité parente (activité principale) lance une activité
 *             secondaire
 *             L'activité principale envoie des informations à l'activité
 *             secondaire en utilisant un objet de type Parcealable
 * @author C. Servières
 * @version 1.0
 */
public class MainActivity extends Activity {

    /** Clé pour les informations transmises à l'activité secondaire */
    public final static String CLE_INFO = "com.example.cours.android.INFO";

    /** Zone de saisie du nom  */
    private EditText saisieNom;

    /** Zone de saisie du prénom  */
    private EditText saisiePrenom;

    /** Zone de saisie de l'adresse  */
    private EditText saisieAdresse;

    /** Zone de saisie du code  */
    private EditText saisieCode;


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

        saisieNom = (EditText) findViewById(R.id.saisie_nom);
        saisiePrenom = (EditText) findViewById(R.id.saisie_prenom);
        saisieAdresse = (EditText) findViewById(R.id.saisie_adresse);
        saisieCode = (EditText) findViewById(R.id.saisie_code);
    }


    /* Méthode invoquée lors du clic sur le bouton de création de l'activité
     * secondaire. Celle-ci est créée et lancée.
     * @param bouton  bouton dont la méthode intercepte le clic
     */
    public void clicBoutonLancer(View bouton) {

        // on accède aux nom, prénom, adresse saisis par l'utilisateur
        String leNom = saisieNom.getText().toString();
```

```
        String lePrenom = saisiePrenom.getText().toString();
        String lAdresse = saisieAdresse.getText().toString();

        // on accède au code (si incorrect alors initialisé à 0)
        int leCode;
        try {
            leCode = Integer.parseInt(saisieCode.getText().toString());
        } catch(NumberFormatException err) {
            leCode = 0;
        }

        // création d'un objet contenant toutes les info. saisies
        InfoAuthentification info =
                new InfoAuthentification(leNom, lePrenom, lAdresse, leCode);

        // création d'une intention pour demander lancement de l'activité secondaire
        Intent intention =
                new Intent(MainActivity.this, ActiviteSecondaire.class);

        // on ajoute les informations d'authentification en tant qu'extra
        intention.putExtra(CLE_INFO, info);
        startActivity(intention);       // on démarre l'activité secondaire
    }

}
```

Classe *MainActivity*

Pour finir, définissons l'activité secondaire.

```
/*  Activité secondaire de l'exemple 3 illustrant la communication entre activités
    ActiviteSecondaire.java                                                 07/16
*/

package com.example.cours.android.exmultiactiviteavecparcealable;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;
import android.widget.TextView;

/**
 * Activité secondaire de l'application permettant d'illustrer la communication
 * entre activités
 * Exemple 3 : une activité parente (activité principale) lance une activité
 *             secondaire
 *             L'activité principale envoie des informations à l'activité
 *             secondaire en utilisant un objet de type Parcealable
 * L'activité secondaire affiche ces informations pour vérification
 * @author C. Servières
 * @version 1.0
 */
public class ActiviteSecondaire extends Activity {

    /** Texte affiché qui contient les informations provenant
     * de l'activité principale */
```

```
    private TextView messageRetour;

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

        messageRetour = (TextView) findViewById(R.id.texte_retour);

        // on récupère l'intention qui a lancé cette activité
        Intent intentionRecu = getIntent();

        // on récupère les informations transmises
        InfoAuthentification information =
                intentionRecu.getParcelableExtra(MainActivity.CLE_INFO);
        messageRetour.setText(String.format(
                                messageRetour.getText().toString(),
                                information.toString()));
    }

}
```

Classe *ActiviteSecondaire*

V) Les intentions implicites

Les intentions des parties précédentes étaient explicites puisqu'elles spécifiaient exactement la classe activité à démarrer. Au contraire, d'autres intentions sont dites implicites. Elles se conteront de demander la réalisation d'une tâche ou action sans préciser l'activité précise à démarrer. Par exemple, on pourra demander à prendre une photo, à ouvrir un fichier *pdf*, sans connaître exactement l'application installée sur le terminal, ni son nom, ni son package.

Plus précisément, l'objet de type ***Intent*** qui encapsulera la demande de lancement d'une activité contiendra un descriptif des capacités de traitement dont l'activité réceptrice devra être dotées. Ensuite, le *framework* Android tentera de satisfaire cette demande en recherchant dans tout le système, et pas uniquement dans l'application courante, les activités répondant aux exigences exprimées dans l'intention implicite. A la fin de cette recherche, soit l'activité trouvée s'affichera, soit une liste d'activités sera proposée à l'utilisateur si plusieurs activités répondent à la demande, soit une exception sera levée si aucune activité n'est trouvée.

5.1) Définition d'une intention implicite

L'action demandée par une intention implicite est définie par : une **action**, une ou plusieurs **catégories**, et une ou plusieurs **données**.

L'**action** est définie sous la forme d'une chaîne de caractères qui symbolise le traitement à déclencher. De nombreuses constantes sont prédéfinies pour représenter les actions nativement disponibles. Exemples :

`Intent.ACTION_EDIT`	pour demander l'ouverture d'un éditeur pour visualiser et modifier des données
`Intent.ACTION_WEB_SEARCH`	pour demander la réalisation d'une recherche sur Internet
`Intent.ACTION_CALL`	pour passer un appel téléphonique

L'intention peut contenir **une ou plusieurs catégories**, sous la forme de chaînes de caractères qui complètent l'action. Par exemple :

`Intent.CATEGORY_DEFAULT`	pour choisir l'activité prévue par défaut pour réaliser l'action demandée
`Intent.CATEGORY_LAUNCHER`	pour indiquer que l'activité est placée sur le lanceur du terminal

Les **données** associées à l'intention sont facultatives. Leur présence dépend de l'action à réaliser. Par exemple, si l'action à réaliser est `Intent.ACTION_CALL`, la donnée sera le numéro de téléphone à composer. En général, la donnée est couplée avec une autre information qui est le type MIME de celle-ci. Par exemple, pour l'appel téléphonique, la donnée sera « `tel:0600000000` ».

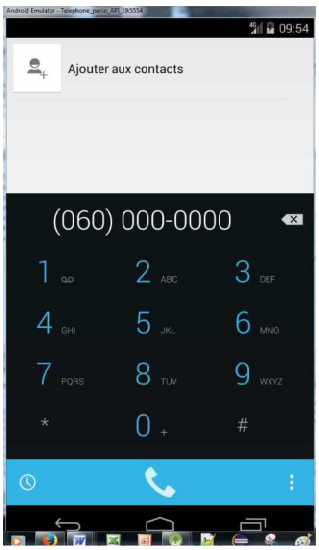
5.2) Exemple

Pour illustrer le principe des intentions implicites, supposons que l'on souhaite proposer à l'utilisateur d'une application de :

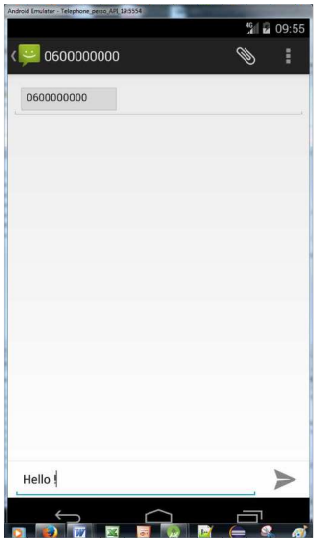
- ✓ appeler un numéro de téléphone précis
- ✓ envoyer un sms à un numéro précis
- ✓ ouvrir un navigateur pour effectuer une recherche avec le moteur de Google
- ✓ prendre une photo

L'activité principale de l'application affichera 4 boutons permettant de lancer chacune des 4 opérations possibles.

Selon le bouton sur lequel l'utilisateur cliquera, l'une ou l'autre des activités suivantes sera lancée :

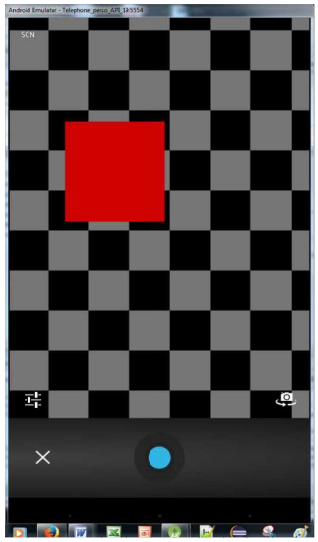


Appel téléphonique



Envoi SMS

Recherche Google


Appareil photo

Le code Java de l'activité principale est le suivant :

```java
/*  Activité principale de l'exemple 4 illustrant la communication entre activités
    MainActivity.java                                                    07/16
*/

package com.example.cours.android.exmultiactivitesimplicite;

import android.app.Activity;
import android.content.ActivityNotFoundException;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.provider.MediaStore;
import android.view.View;
import android.widget.Toast;

/**
 * Activité principale de l'application permettant d'illustrer la communication
 * entre activités
 * Exemple 4 : une activité parente (activité principale) lance des intentions
 *             implicites : appel téléphonique, envoi SMS, ouverture navigateur ...
 * @author C. Servières
 * @version 1.0
 */
public class MainActivity extends Activity {

    /** Numéro de téléphone utilisé comme exemple  */
    public final static String NUMERO = "0600000000";


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


    /**
```

```
     * Méthode invoquée lors du clic sur le bouton de demande d'ouverture
     * du composeur téléphonique
     * @param bouton  bouton dont la méthode intercepte le clic
     */
    public void clicAppel(View bouton) {

        // préparation de l'URI contenant un numéro à appeler
        Uri telephone = Uri.parse("tel:" + NUMERO);

        // création d'une intention
        Intent intention = new Intent(Intent.ACTION_DIAL, telephone);
        startActivity(intention);
    }

    /**
     * Méthode invoquée lors du clic sur le bouton de demande d'envoi d'un sms
     * @param bouton  bouton dont la méthode intercepte le clic
     */
    public void clicSms(View bouton) {

        try {

            // préparation de l'URI contenant un numéro auquel envoyer le sms
            Uri telephone = Uri.parse("smsto:" + NUMERO);
            Intent intention = new Intent(Intent.ACTION_SENDTO, telephone);

            // on ajoute un corps au sms
            intention.putExtra("sms_body", "Hello !");
            startActivity(intention);
        } catch (ActivityNotFoundException e) {

            // si aucune activité n'est trouvée pour envoyer le sms
            Toast.makeText(this, "Pb envoi SMS", Toast.LENGTH_LONG).show();
        }
    }

    /**
     *  Méthode invoquée lors du clic sur le bouton de demande d'ouverture
     * d'un navigateur. Le navigateur est ouvert et affiche la page de Google
     * @param bouton  bouton dont la méthode intercepte le clic
     */
    public void clicNavigateur(View bouton) {

        try {

            // création de l'intention qui demande à afficher la page Google
            Intent intentionNavigateur =
                    new Intent(Intent.ACTION_VIEW,
                            Uri.parse("http://www.google.com"));
            startActivity(intentionNavigateur);
        } catch (ActivityNotFoundException e) {

            // si aucun navigateur n'est trouvé
            Toast.makeText(this, "Impossible d'ouvrir le navigateur",
                    Toast.LENGTH_LONG).show();
        }
    }

    /**
     * Méthode invoquée lors du clic sur le bouton de demande de prise de photo
     * @param bouton  bouton dont la méthode intercepte le clic
     */
    public void clicPhoto(View bouton) {
```

```
        // préparation de l'intention
        Intent intention = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);

        /*
         * on demande au package manager si l'action décrite dans l'intention
         * est réalisable, ou autrement dit on vérifie si le terminal est
         * doté d'un appareil photo
         */
        if (intention.resolveActivity(getPackageManager()) != null) {
            startActivity(intention);
        } else {
            // si aucun appareil photo n'est trouvé
            Toast.makeText(this, "Le terminal ne dispose par d'appareil photo !",
                           Toast.LENGTH_LONG).show();
        }
    }
}
```

5.3) Les données : Notion d'URI - Uniform Resource Indentifier

Elles sont fournies via l'URI du fournisseur de contenu et en appelant la méthode ***setData.***

Les données fournies à l'intention sont formatées dans une syntaxe bien précise. Plus précisément, elles sont formatées sous la forme d'URI (Uniform Resource Identifier). Il y a une analogie entre la notion d'URI et celle d'URL.

La syntaxe d'une URI est la suivante (entre { et } figurent les parties facultatives) :

```
<schéma> : <information> { ? <requête> } { # <fragment> }
```

- Le **schéma** indique la nature de l'information. Par exemple, si l'information est un numéro de téléphone, le schéma sera `tel`, si c'est un site internet, alors le schéma sera `http` ...
- **L'information** est la donnée elle-même. Elle doit aussi respecter un certain format qui dépend du schéma. Par exemple, pour un numéro de téléphone, on indique seulement le numéro (suite de chiffres), pour des coordonnées GPS, on sépare la latitude de la longitude par une virgule. Pour un site internet, on indique le chemin.
- La **requête** permet de fournir une précision par rapport à l'information.
- Le **fragment** permet d'accéder à une sous-partie de l'information.

Au niveau du code Java, pour créer un objet de type URI, il faut appeler la méthode statique :

```
Uri  Uri.parse(String texteUri)
```

Par exemple, pour construire une URI destinée à l'envoi d'un SMS connaissant le numéro du destinataire :

```
Uri sms = Uri.parse("sms:0600000000");
```

On peut aussi indiquer plusieurs destinataires et le texte du message :

```
Uri sms = Uri.parse("sms:0600000000,0600000001,0600000002?body=Hello");
```

Attention : le contenu de la chaîne doit être encodé. Si on veut insérer un espace dans le message du sms, par exemple, il faudra écrire %20.

5.4) Précisions sur l'action

Une action est fournie via une constante de la classe ***Intent***. Les identificateurs des constantes possibles commencent par **ACTION_**. Le programmeur peut aussi créer ses propres actions.

Exemples d'actions possibles (donc de constantes de la classe ***Intent***) :

Intitulé	Action	Entrée attendue	Sortie attendue
ACTION_MAIN	Pour indiquer qu'il s'agit du point d'entrée de l'application		
ACTION_DIAL	Pour ouvrir le composeur de numéro de téléphones	Un numéro de téléphone	
ACTION_DELETE	Supprimer des données	Une URI vers les données à supprimer	
ACTION_EDIT	Ouvrir un éditeur pour visualiser et modifier des données	Une URI vers les données à éditer.	
ACTION_INSERT	Insérer des données	L'URI qui correspond à un répertoire dans lequel insérer les données.	L'URI des nouvelles données créées.
ACTION_PICK	Sélectionner un élément dans un ensemble de données	L'URI qui correspond à un répertoire de données à partir duquel l'élément sera sélectionné	L'URI de l'élément qui a été sélectionné.
ACTION_SEARCH	Effectuer une recherche	Le texte à rechercher	
ACTION_SENDTO	Envoyer un message	Le destinataire du message	
ACTION_VIEW	Pour visionner une donnée	Plusieurs possibilités : une adresse mail, un numéro de téléphone ...	
ACTION_WEB_SEARCH	Effectuer une recherche sur Internet	Si le texte commence par "http", le site s'affichera directement, sinon c'est une recherche Google qui sera faite.	

Exemple : on souhaite ouvrir le composeur téléphonique avec un numéro de téléphone précis.

```
Uri telephone = Uri.parse("tel:0600000000");
Intent intention = new Intent(Intent.ACTION_DIAL, telephone);
startActivity(intention);
```

5.5) Résolution des intentions - Notion de filtre

Quand on lance, par exemple, une intention avec l'action ACTION_VIEW et une donnée adresse internet c'est un navigateur qui s'ouvre. Si par contre la donnée est un numéro de téléphone, ce sera le composeur de numéros qui s'ouvrira. Comment Android détermine-t-il quelle application doit répondre à l'intention ?

En fait toute activité (et il en est de même pour les services et les récepteurs de diffusion), qu'elle soit prédéfinie ou développée par le programmeur, déclare des filtres (ou ***IntentFilter***) d'intention dans le fichier *AndroidManifest.xml*. Les filtres spécifient l'ensemble des actions, catégories et données que l'application est capable de gérer.

Le système Android les prend en charge lors de l'installation de l'application. Plus précisément, c'est le gestionnaire de packages (ou ***PackageManager***) qui gère une base de registres à partir de tous les fichiers *manifest*, et qui utilise celle-ci pour résoudre les intentions implicites. Pour ce faire, il compare les triplets {action, catégories, données} de l'intention émise avec les valeurs des filtres inscrits dans sa base. Plusieurs cas se présentent :

- ✓ une seule activité est trouvée : c'est elle qui est lancée
- ✓ aucune activité avec ce filtre n'est trouvée : une exception ***ActivityNotFoundException*** est levée Il faudra donc penser à gérer ce type d'erreur dans notre code.
- ✓ si plusieurs activités correspondent au filtre, le système va demander à l'utilisateur d'en choisir une, via une boîte de dialogue.

Dans le code Java, on peut accéder au ***PackageManager*** et tester si une intention pourra être traitée, avant de la lancer. Pour ce faire, on écrit :

```
Intent intention = new Intent(Intent.ACTION_VIEW,Uri.parse("tel:0600000000"));
PackageManager gestionnairePackage = getPackageManager();
ComponentName composant = intention.resolveActivity(gestionnairePackage);

if (composant != null) {
    // une activité sera capable de gérer l'intention : on peut la lancer
    .......
```

Déclaration d'un filtre avec la balise <intent-filter>

L'élément **<intent-filter>** déclare un filtre dans un fichier *manisfest*. Il est imbriqué dans les éléments <activity>, <service>, <receiver>.

Il se compose des sous-éléments suivants : *action*, *category* et *data*.

Par exemple, pour l'activité principale d'une application, on mentionne généralement le filtre suivant :

```
<intent-filter>
     <action android:name="android.intent.action.MAIN" />
     <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
```

L'activité est alors considérée comme l'activité principale de l'application, et le système fait en sorte qu'elle puisse être lancée à partir du lanceur du terminal.

a) l'action

Elle permet de filtrer en fonction du champ action de l'intention. Si le filtre ne contient aucune action, aucune intention ne passera au travers. Une intention sera acceptée si l'une des actions qui se trouve dans son champ *action* est identique à au moins une des actions du filtre.

Par exemple, si le filtre est :

```
<intent-filter>
      <action android:name="android.intent.action.VIEW" />
      <action android:name="android.intent.action.SENDTO" />
</intent-filter>
```

L'activité ayant ce filtre ne pourra intercepter que les intentions qui ont comme action ACTION_VIEW ou ACTION_SENDTO.

b) la catégorie

Une intention n'a pas nécessairement une catégorie. Si par contre, une intention a les catégories *C1, C2,* et *C3*, il faut que toutes ces catégories correspondent à celles du filtre, pour que l'intention passe au travers de celui-ci.

Attention, si on ne souhaite pas associer de catégorie à un filtre, il faut mentionner

```
<category android:name="android.intent.category.DEFAULT" />
```

sinon aucune intention ne passera au travers, même si l'intention n'a pas de catégorie.

Par exemple, on suppose que le programmeur a créé sa propre catégorie `MA_CATEGORIE`. Notons que pour veiller à l'unicité de celle-ci, il est préférable de préfixer son nom par celui de son package de définition. Si une activité possède le filtre suivant :

```
<activity>
    <intent-filter>
          <action android:name="android.intent.action.VIEW" />
          <action android:name="android.intent.action.SEARCH" />
          <category android:name="android.intent.category.DEFAULT" />
          <category android:name="com.exemple.cours.android.categorie.MA_CATEGORIE"/>
    </intent-filter>
</activity>
```

Ce filtre laissera passer les intentions ayant comme action `ACTION_VIEW` et/ou `ACTION_SEARCH`, et n'ayant soit pas de catégorie, soit la catégorie `CATEGORY_MA_CATEGORIE`.

Gérer la rotation de l'écran

I) Introduction

Lorsqu'une modification de la configuration du terminal risque d'affecter la sélection des ressources faite par le système, Android supprime toutes les activités en cours d'exécution ou en pause, et les recrée la prochaine fois où elles sont affichées.

Quelques exemples de situations qui affectent la sélection des ressources :

- ✓ la rotation de l'écran, c'est-à-dire la modification de l'orientation
- ✓ ouverture ou fermeture d'un clavier physique
- ✓ placement du terminal sur un support de bureau, de voiture ...
- ✓ changement au niveau des paramètres du terminal : pas exemple la langue préférée

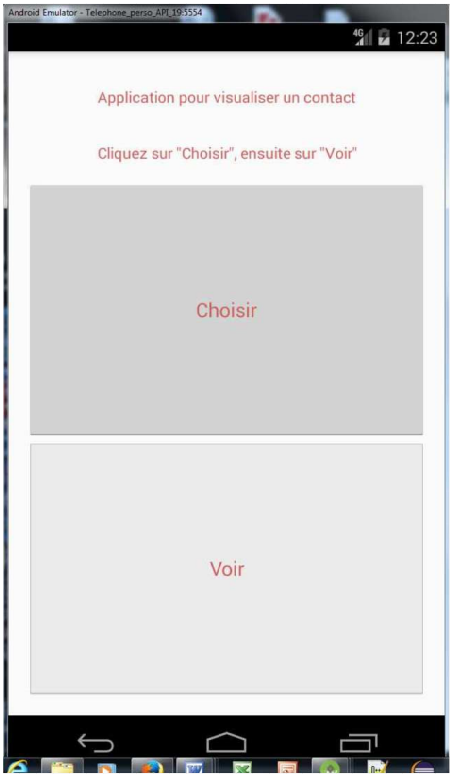
A propos de l'orientation de l'écran, il faut se souvenir que le développeur peut créer 2 fichiers *layout* pour une même activité :

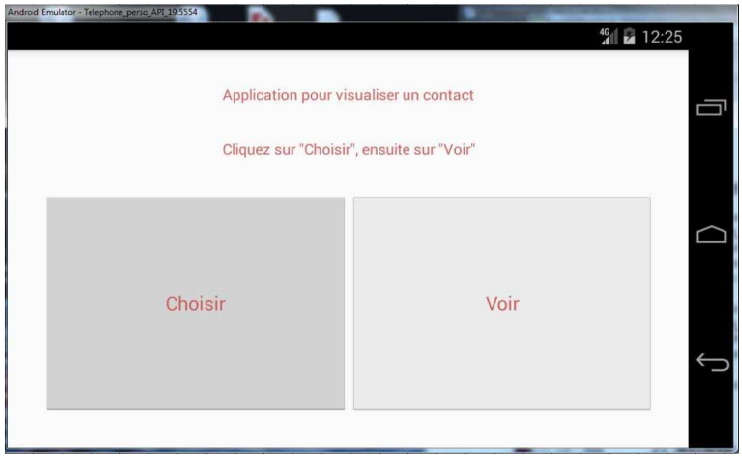
- l'un, nommé *layout_main.xml* par exemple, sera placé dans le dossier `res/layout` et décrira la vue pour une orientation portrait
- l'autre, portant le même nom, sera placé dans le dossier `res/layout-land` et décrira la vue pour une orientation paysage

Dans la suite, nous allons illustrer les principes à l'aide de l'exemple suivant. L'application exemple devra permettre à l'utilisateur de visualiser la liste des contacts enregistrés dans le terminal, et d'en sélectionner un parmi cette liste. En elle-même, cette l'application ne présente pas d'intérêt. Mais les fonctionnalités que nous allons développer pourraient être intégrées à une application plus vaste qui proposerait, par exemple, d'envoyer un SMS prédéfini à un contact choisi par l'utilisateur.

L'activité principale de l'application devra afficher 2 boutons : l'un pour choisir un contact, l'autre pour visualiser le contact choisi. Ce deuxième bouton ne sera cliquable que si un contact a été choisi au préalable. De plus, on souhaite qu'en mode portrait les boutons s'affichent l'un en dessous de l'autre, alors qu'en mode paysage, ils seront affichés côte à côte.

On souhaite, bien sûr, que lorsqu'un changement d'orientation se produit, l'état du bouton « Choisir » soit conservé, et si de plus l'utilisateur a déjà sélectionné un contact, on souhaite que celui-ci ne soit pas perdu. Si le programmeur ne gère pas lui-même le changement d'orientation, ces deux informations ne suivront pas en cas de rotation de l'écran.





Comment accéder aux contacts enregistrés dans le terminal ?

Pour afficher la liste des contacts disponibles, il faudra envoyer une intention de type `Intent.ACTION_PICK`. Il s'agit en fait du type de l'intention implicite envoyée. Elle indiquera à Android que nous souhaitons sélectionner un élément. Pour préciser qu'il s'agit de sélectionner précisément un contact, et pas un élément d'une autre nature, le deuxième argument du constructeur de l'intention devra être la constante `Contacts.CONTENT_URI`.

Si l'utilisateur sélectionne un contact, il sera communiqué à l'activité qui a envoyé l'intention de type `Intent.ACTION_PICK`, à condition bien sûr que celle-ci ait demandé à recevoir un résultat, via l'intention de retour. En appliquant la méthode ***getData()*** sur cette dernière, on obtient une instance de type ***Uri***, représentant le contact choisi.

Nous allons envisager plusieurs manières de procéder pour gérer les changements d'orientation.

2) Méthode 1 - Sauvegarde des informations dans la méthode onSaveInstanceState

La méthode ***onSaveInstanceState*** est appelée lorsque l'activité termine son cycle visible, donc avant qu'elle ne soit détruite. On peut donc, dans cette méthode, sauvegarder les informations qui ne le sont pas automatiquement par le système.

```
protected void onSaveInstanceState(Bundle etat)
```

Précisément, l'idée est d'enregistrer les informations dans l'objet de type *Bundle* paramètre de la méthode. Cet enregistrement se fait en invoquant les méthodes : `putXXX` où **XXX** est en fait un type de base du langage Java, ou bien *FloatArray*, *StringArrayList*, *Parcealable*, *Serializable* .... La méthode possède 2 arguments : une clé (chaîne de caractères), et la valeur à enregistrer dont le type doit correspondre à **XXX**.

Exemple : `etat.putString(CLE_TEXTE, leTexte);`

Les informations ainsi sauvegardées pourront être restaurées dans la méthode ***onCreate***, ***onStart***, ou bien ***onRestaureInstanceState***. Rappelons que cette dernière méthode n'est appelée que si un état a été au préalable enregistré avec la méthode ***onSaveInstanceState***.

Attention, lors du premier appel à la méthode ***onCreate***, par exemple, l'argument de type *Bundle* est égal à ***null***. Il ne faudra donc pas tenter de restaurer un état antérieur dans ce cas précis.

Pour accéder aux informations stockées dans l'objet de type *Bundle*, on utilise les méthodes ***getXXX*** symétriques à ***putXXX***. Les méthodes ***getXXX*** ont en argument une clé. Elle renvoie un résultat dont le type correspond à ***XXX***, ou éventuellement la valeur ***null*** si aucune information possédant cette clé n'a été trouvée.

3) Méthode 2 - Utilisation de la méthode onRetainNonConfigurationInstance

Si les informations à sauvegarder sont d'un type de base du langage, ou sont sérialisables, ou en résumé sont d'un type pris en charge par les méthodes ***putXXX***, la technique précédente convient tout à fait.

Cependant, on peut souhaiter parfois sauvegarder et rétablir des éléments d'une autre nature : une connexion à un serveur, par exemple.

Pour ce faire, on peut utiliser la méthode de rappel ***onRetainNonConfigurationInstance()*** au lieu de ***onSaveInstanceState()***. Cette méthode de la classe *Activity* peut renvoyer un résultat de type ***Object*** que l'on peut récupérer ultérieurement avec la méthode ***getLastNonConfigurationInstance()***. Généralement cet objet sera un objet de type ***Context*** contenant l'état de l'activité, donc y compris les *threads* en cours d'exécution ou les sockets ouvertes, par exemple.

4) Méthode 3 - Gestion personnalisée des rotations

Le programmeur peut gérer lui-même la rotation de l'écran. Plus précisément, il peut empêcher le comportement par défaut qui engendre la destruction de l'activité lors d'une rotation. Ceci peut être très utile dans un jeu d'action par exemple, pour que l'application ne soit pas ralentie lorsque l'utilisateur fait pivoter le terminal.

Pour ce faire, il faut :

- ✓ modifier une propriété de l'activité, dans le fichier ***AndroidManisfest.xml***
- ✓ et implémenter la méthode ***onConfigurationChanged()*** dans la classe activité.

Plus précisément, dans le fichier *AndroidManisfest.xml*, dans l'élément ***activity*** de l'activité concernée, on ajoute la propriété : **`android:configChanges`** et on lui donne pour valeur la liste des modifications de configuration que nous souhaitons gérer nous-mêmes.

Par exemple, si nous souhaitons gérer les ouvertures de clavier et les rotations :

```
<activity
    android:name=".MainActivity"
    android:configChanges="keyboardHidden|orientation">
```

Dans la classe qui représente l'activité, il faut implémenter la méthode :

```
void onConfigurationChanged(Configuration nouvelleConfiguration)
```

en commençant par un appel à la méthode de la classe parente :

```
super.onConfigurationChanged(nouvelleConfiguration)
```

Cette méthode sera appelée automatiquement chaque fois que l'un des changements énumérés dans la propriété **`android:configChanges`** se produira.

Remarques

Il n'est pas conseillé d'utiliser systématiquement cette troisième méthode. En effet, le risque est d'oublier de restaurer certaines ressources : les dispositions, les menus, les animations ...

Certaines activités ne sont pas conçues pour changer d'orientation : certains jeux, les lecteurs vidéo, la prévisualisation de l'appareil photo ... Dans ce cas, on peut demander à Android de ne pas faire pivoter la vue de l'activité. Pour ce faire, on ajoute la propriété **android:screenOrientation** dans l'élément **`activity`** du fichier *AndroidManisfest.xml.*

Par exemple, pour imposer une orientation portrait :

```
<activity
    android:name=".MainActivity"
    android:screenOrientation="portrait">
```

Pour une orientation paysage, la valeur de la propriété est **`"landscape"`**.

Cependant Android, supprimera et recréera malgré tout l'activité en cas de changement d'orientation. Pour éviter ceci, il faut utiliser conjointement la propriété **`android:configChanges`** comme expliqué précédemment.

Spécificité des tablettes : les fragments

I) Introduction

Les tablettes sont prises en charge depuis la version Honeycomb d'Android. Cependant, depuis Ice Cream Sandwich, les versions d'Android sont conçues à la fois pour les tablettes et les téléphones. Donc les applications actuelles doivent être capables de fonctionner à la fois sur les téléphones et les tablettes.

Lorsque l'on développe des applications pour les tablettes, il faut prendre en compte des éléments supplémentaires : les écrans sont plus grands, la résolution plus élevée, par exemple.

Parfois, l'interface d'une application fonctionnant sur tablette, pourra être spécifique et comporter, par exemple, plusieurs volets. Chaque volet correspondra à un **fragment**.

Volet 1	Volet 2

Un **fragment** peut être vu comme une couche optionnelle entre les activités et les *widgets*, une sorte de sous-activité. On ne les déclare pas dans le fichier *AndroidManifest.xml*.

2) Généralités sur les fragments d'interface

Eléments liste	
Item 1 *Item 2* ***Item 3*** *Item 4* *Item 5*	*Détails de l'item 3* *(celui sélectionné)*

Une même activité peut accueillir plusieurs fragments. Un exemple typique est le suivant : sur la moitié gauche de l'écran, on visualise une liste, et sur la moitié droite on visualise l'élément de la liste qui a été sélectionné. Chaque volet sera donc décrit par un fragment spécifique. L'activité gèrera les 2 fragments.

Un même fragment peut être utilisé par plusieurs activités. Le concept de fragment favorise donc la réutilisation du code déjà écrit.

Les fragments ont beaucoup de points communs avec les activités : ils ont un cycle de vie, et sont associés à une vue. Mais ils donnent plus de possibilités : plusieurs fragments peuvent être actifs simultanément, il est possible de les placer dans une pile, par exemple.

On dit qu'un fragment est "attaché" à une activité. Le cycle de vie du fragment est affecté par celui de l'activité à laquelle il est attaché. Par exemple, si une activité est détruite, tous les fragments qui la composent sont détruits aussi automatiquement.

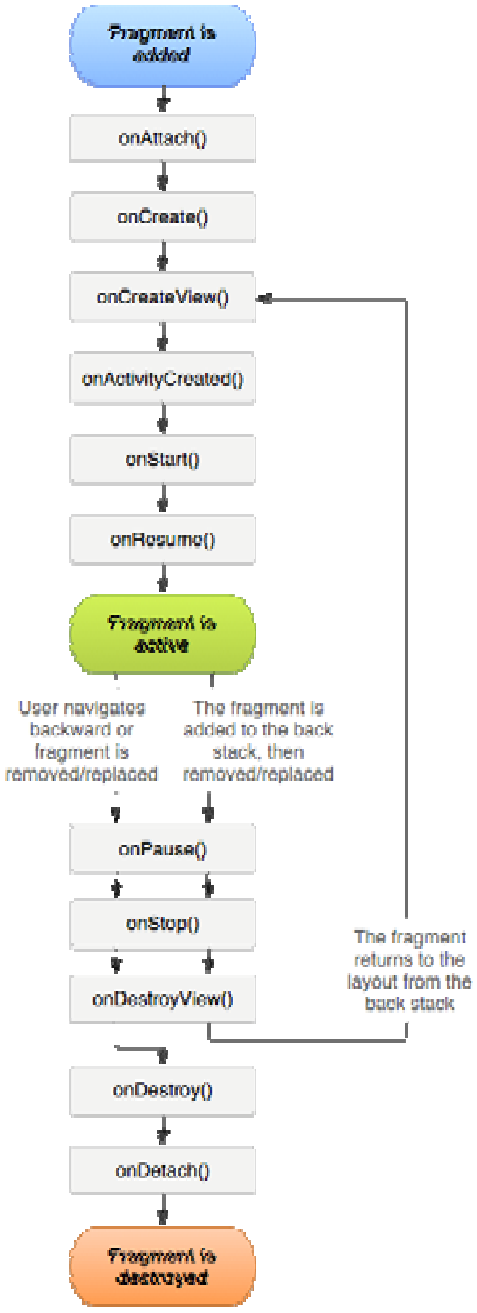
Cycle de vie d'un fragment

Pour créer un fragment, il faudra écrire une classe qui hérite de la classe prédéfinie ***Fragment***. Cette classe contient plusieurs méthodes décrivant le cycle de vie du fragment. Le programmeur devra redéfinir certaines d'entre elles.

Parmi les méthodes du cycle de vie d'un fragment, nous retrouvons celles du cycle de vie d'une activité. Parmi ces dernières, il faut obligatoirement implémenter ***onCreate()***, appelée à la création du fragment.

Le cycle de vie d'un fragment contient les méthodes spécifiques suivantes. Parmi celles-ci, l'implémentation de ***onCreateView*** est obligatoire.

`onAttach`	Appelée quand le fragment est attaché (associé) à son activité parente. Un fragment ne peut pas exister sans être associé à une activité.
`onCreateView`	Appelée quand le fragment doit créer sa vue (dessiner son contenu)
`onActivityCreated`	Appelée quand la création de l'activité mère est terminée
`onDestroyView`	Appelée quand la vue associée au fragment est en cours de destruction
`onDetach`	Appelée quand le fragment est sur le point d'être dissocié de son activité parente



Un fragment ne peut pas exister sans être associé à une activité. Son cycle de vie débute au moment où il est attaché à son activité parente. La méthode ***onAttach*** est alors exécutée.

L'implémentation de la méthode ***onCreate*** est obligatoire. La méthode est appelée automatiquement à la création du fragment. On initialise dans celle-ci les variables ou objets essentiels au fonctionnement du fragment.

L'implémentation de la méthode ***onCreateView*** est également obligatoire. Elle est appelée lorsque le fragment doit dessiner son contenu. On place donc dans cette méthode les instructions permettant de charger la vue. Cette dernière doit être renvoyée en tant que résultat de la méthode.

La méthode ***onActivityCreated*** est appelée quand le fragment a fini de se créer et de se dessiner. On peut coder dans cette méthode les interactions avec l'utilisateur, par exemple.

La méthode ***onStart*** est invoquée quand le fragment passe au premier plan (tout comme la méthode *onStart* de la classe *Activity*).

De même, la méthode ***onResume*** est appelée dès lors que l'utilisateur peut interagir avec le fragment (analogie avec *onResume* de la classe *Activity*).

Ensuite, lorsque le fragment devient inactif, les méthodes ***onPause*** et ***onStop*** sont appelées, sur le même principe que les mêmes méthodes de la classe *Activity* (*onPause* quand le fragment passe au second plan, *onStop* quand il n'est plus visible).

Puis sont appelées :
onDestroyView : quand la vue du fragment est détruite
onDestroy : quand le fragment est détruit
onDetach : quand le fragment est séparé de l'activité parente

Important : toutes ces méthodes doivent comporter un appel à la méthode équivalente de la classe de base ! (donc un appel à ***super***).

Deux méthodes importantes de la classe *Fragment*

A) `View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstance)`

Les paramètres ont la signification suivante :
- ✓ *inflater* est un objet outil permettant de transformer une vue décrite en XML en un objet de type *View* du langage Java (il permet donc de désérialiser une vue)
- ✓ *container* est la vue parente du fragment, celle qui le contient (donc éventuellement celle de l'activité qui contient le fragment)

- ✓ *savedInstance* est l'objet *Bundle* qui contient les données utiles à la reconstruction du fragment. Il est égal *null* s'il s'agit de la première construction du fragment.

Dans cette méthode, on associe la vue du fragment à l'activité mère, et le fragment doit créer sa propre vue. Pour ce faire, on n'utilise pas un appel à *setContentView* comme on le fait pour une activité, mais on fait appel à la méthode ***inflater.inflate*** en lui passant en paramètre l'identifiant de la ressource de la vue. La méthode doit renvoyer comme résultat la vue créée. Donc, la méthode aura souvent le squelette suivant :

```
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                         Bundle savedInstanceState) {

    // initialisation de la vue du fragment, et de la zone de résultat
    View view = inflater.inflate(R.layout.layout_du_fragment,
                                 container,
                                 false);
    // des instructions en lien avec la vue
    . . .
    return view;    // il faut renvoyer la vue associée au fragment
}
```

B) onActivityCreated()

Quand cette méthode s'exécute, la méthode *onCreate()* de l'activité mère est terminée. On initialise dans cette méthode les données (par exemple, les éléments d'une liste). On aura parfois besoin dans cette méthode d'accéder à l'activité mère, pour interagir avec elle ou pour utiliser des méthodes de la classe *Context* (car le contexte est celui de l'activité mère). Pour accéder à l'activité mère, il faut écrire :

```
Activity activiteParente = getActivity();
```

Remarques

Il existe une classe ***Fragment*** pour gérer les fragments "classiques", et aussi 3 classes pour gérer des cas particuliers de fragments :

- ✓ ***ListFragment*** pour afficher une liste d'items, selon le même principe que *ListActivity*
- ✓ ***DialogFragment*** pour afficher un fragment dans une boîte de dialogue qui se superpose à la vue courante
- ✓ ***PreferenceFragment*** pour gérer les préférences

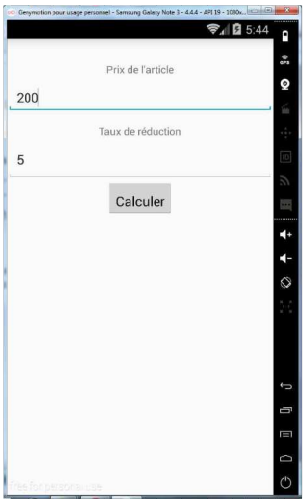
Un fragment peut être chargé dans la vue d'une activité soit statiquement grâce à la description présente dans les fichiers décrivant les *layouts*, soit dynamiquement à l'aide de la classe ***FragmentManager***. Examinons ces deux manières de procéder.

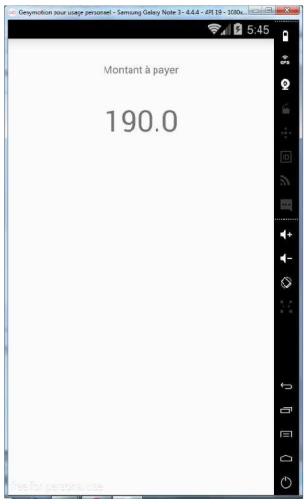
3) Les fragments statiques

Ils ont les particularités suivantes :

- ✓ ils sont déclarés dans le *layout* de l'activité à laquelle ils appartiennent, et ce avec la balise ***fragment***
- ✓ ils ne peuvent pas être ajoutés, remplacés ou supprimés dynamiquement.

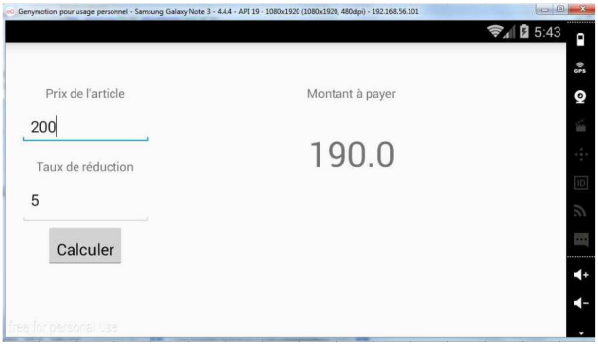
Dans l'exemple, nous allons envisager 2 interfaces pour l'application :





- ✓ l'une en mode portrait : il y aura 2 activités, l'une permettant à l'utilisateur de saisir les données, l'autre affichant le résultat

- ✓ l'autre en mode paysage : il y aura une seule activité dont la vue sera composée de 2 fragments. Quand l'utilisateur cliquera sur le bouton de calcul, le 2ème fragment apparaîtra à droite pour afficher le résultat.



Nous aurons besoin de 5 fichiers *layout* :

- <u>Pour le mode paysage :</u>
  - ✓ *layout_main.xml* : une vue contenant 2 fragments
  - ✓ *layout_fragment_saisie.xml* : le contenu du fragment des saisies (avec le bouton calculer)
  - ✓ *layout_fragment_resultat.xml* : le contenu du fragment affichant le résultat
- <u>Pour le mode portrait</u> (donc dans un dossier `layout-port`) :
  - ✓ *layout_main.xml* : une vue contenant un seul fragment (celui des saisies)
  - ✓ *layout_resultat_portrait.xml* : une vue contenant un seul fragment (celui affichant le résultat)

Nous définirons 4 classes Java pour gérer :
- ✓ l'activité principale
- ✓ l'activité secondaire qui affiche le résultat. Cette classe ne sera utilisée que dans le mode portrait
- ✓ le fragment qui permet de saisir les données
- ✓ le fragment qui permet d'afficher le résultat

Le code complet de cette application exemple est donné en annexe.

Explications à propos des fichiers *layout*

Le *layout* de l'activité principale, en mode paysage contient 2 éléments de type ***fragment***. Chaque élément de type ***fragment*** comporte une propriété pour l'identifier et une autre pour définir la classe Java décrivant le comportement ce fragment (donc une classe qui hérite de ***Fragment***). Quand le fragment est chargé dans la vue, le code de cette classe est exécuté **automatiquement**, selon le cycle de vie décrit précédemment.

```
<LinearLayout
    . . . >

    <!-- fragment de gauche : il permet de saisir les montants
         l'espace réservé sera occupé par layout_fragment_saisie -->
    <fragment
        android:id="@+id/fragmentSaisie"
        . . .
        class="com.exemple.cours.android.exfragmentstatique.FragmentSaisie">
    </fragment>

    <!-- fragment de gauche : il permet d'afficher le résultat
         l'espace réservé sera occupé par layout_fragment_resultat -->
    <fragment
        android:id="@+id/fragmentResultat"
        . . .
        class="com.exemple.cours.android.exfragmentstatique.FragmentResultat">
    </fragment>

</LinearLayout>
```

***layout_main.xml* (extrait, en mode paysage)**

Lorsque l'activité principale sera lancée en mode portrait, le fichier *layout* décrivant sa vue sera le suivant. Notons qu'il y a un seul élément de type ***fragment***.

```
<LinearLayout
    . . . >

    <!-- fragment de gauche : il permet de saisir les montants
         l'espace réservé sera occupé par layout_fragment_saisie -->
    <fragment
        android:id="@+id/fragmentSaisie"
        . . .
        class="com.exemple.cours.android.exfragmentstatique.FragmentSaisie">
    </fragment>

</LinearLayout>
```

***layout_main.xml* (extrait, en mode portrait)**

Les fichiers ***layout_fragment_resultat.xml*** et ***layout_fragment_saisie.xml*** décrivent la vue associée aux 2 fragments de la vue principale. Ils sont écrits de manière habituelle, c'est-à-dire ils contiennent un élément de type *layout*, et les différents *widgets* tels que *TextView*, *EditText*, *Button*.

Enfin, la vue correspondant à l'activité qui affiche le résultat en mode portrait est décrite par le fichier suivant. Notons que cette vue contient un seul fragment.

```
<LinearLayout
    . . . >

    <fragment
        android:id="@+id/fragmentResultat"
        . . .
        class="com.exemple.cours.android.exfragmentstatique.FragmentResultat">
    </fragment>

</LinearLayout>
```

layout_resultat_portrait.xml (extrait)

Explications à propos des classes Java

Dans chaque classe Java correspondant à un fragment, on a redéfini la méthode ***onCreateView*** dans laquelle on associe une vue au fragment, comme par exemple :

```
View view = inflater.inflate(R.layout.layout_fragment_saisie,
                             container,
                             false);
```

Lors du clic sur le bouton "calculer", il faut envisager 2 cas, selon que le terminal est en mode portrait ou pas. Si le deuxième fragment, celui qui affiche le résultat, existe, cela signifie que le terminal est en mode paysage. Dans ce cas, il suffit d'appeler une méthode de la classe qui gère ce deuxième fragment et qui a pour rôle de mettre à jour le *TextView* correspondant au résultat. Nous avons nous-mêmes écrit cette méthode.

Par contre, si ce deuxième fragment n'existe pas, cela veut dire que le terminal est en mode portrait, il faut lancer l'activité secondaire chargée d'afficher le résultat. On lui transmet, via l'intention, la valeur qu'elle devra afficher en tant que résultat.

```
// on récupère le fragment d'affichage du résultat, s'il existe
FragmentResultat fragment =
     (FragmentResultat) getFragmentManager().findFragmentById(R.id.fragmentResultat);

/* Si le fragment de résultat existe, on met à jour le montant résultat
 * affiché sur ce fragment (l'application est donc en mode paysage)
 */
if (fragment != null && fragment.isInLayout()) {
    fragment.majMontant(montantAAfficher);
} else {

    /*
     * le fragment résultat n'existe pas : l'application est donc en mode
     * portrait. Il faut créer l'activité chargée d'afficher le résultat et
     * lui communiquer le montant à afficher.
     */
    Intent intention = new Intent(getActivity(), ActiviteResultat.class);
```

```
    intention.putExtra(CLE_RESULTAT, montantAAfficher);
    startActivity(intention);
}
```

L'activité présentant le résultat doit normalement être lancée seulement lorsque l'application est en mode portrait. Cependant, le scénario suivant peut se produire : l'utilisateur consulte le terminal en mode portrait, puis il change d'orientation. A ce moment-là, automatiquement l'activité courante est détruire et aussitôt recréée. Or le terminal étant maintenant en mode paysage, la vue *layou_resultat_portrait* (qui est spécifique au mode portrait) n'est plus accessible. Si on tente de la charger une exception se produit. Il faut donc intercepter cette exception, et demander à l'activité courante de se terminer. Le code ci-dessous est un extrait de la méthode *onCreate* de cette activité.

```
try {
    /*
     * si le terminal est en mode portrait, l'appel suivant
     * se déroule sans erreur
     */
   setContentView(R.layout.layout_resultat_portrait);

   // on récupère l'intention et le texte résultat
   Intent intention = getIntent();
   Bundle extra = getIntent().getExtras();
   String resultatBis = extra.getString(FragmentSaisie.CLE_RESULTAT);

   String resultat = intention.getStringExtra(FragmentSaisie.CLE_RESULTAT);

   // on affiche le résultat
   TextView texte = (TextView) findViewById(R.id.zone_resultat);
   if (texte == null) {
       texte.setText("");
   } else {
       texte.setText(resultat);
   }

} catch(Resources.NotFoundException erreur) {
      /*
       * se produit si le terminal est en mode paysage.
       * Il faut détruire l'activité résultat courante.
       * L'activité principale sera rechargée
       */
      finish();
}
```

4) Les fragments dynamiques

Principe

Les fragments dynamiques sont ajoutés, supprimés, ou remplacés de manière dynamique à une activité, ceci selon les besoins de l'application.

Chaque ajout, suppression ou remplacement d'un fragment s'obtient en effectuant une transaction auprès du ***FragmentManager***. Il faut suivre les étapes suivantes :

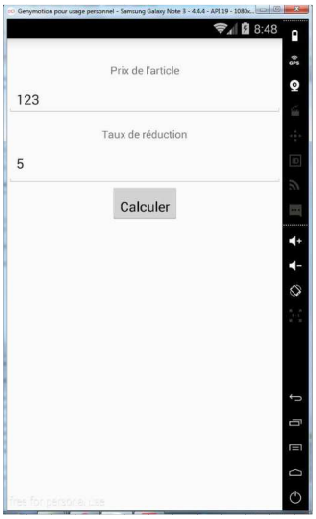
- ✓ obtenir une référence sur le gestionnaire de fragment ***FragmentManager*** en invoquant la méthode ***getFragmentManager***

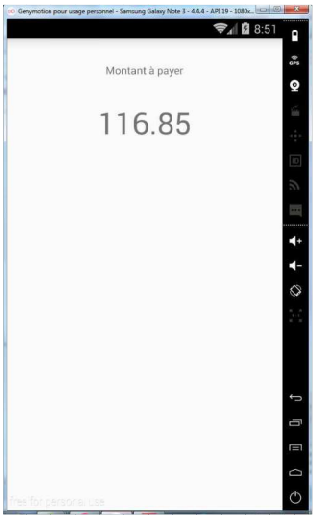
- ✓ débuter la transaction en invoquant ***beginTransaction***
- ✓ effectuer l'opération concernant le fragment. S'il s'agit d'un ajout, par exemple, on donne en argument l'identifiant de la zone où le sera ajouté le fragment et l'instance du fragment à ajouter
- ✓ valider la transaction en invoquant la méthode ***commit***.

Exemple

Prenons un exemple, en nous appuyant sur des fragments dynamiques. L'application sera constituée d'une seule activité et de deux fragments. Bien sûr, elle aurait aussi pu être programmée avec deux activités. Notez également que l'application présentée ici est différente de celle illustrant les fragments statiques. En effet, elle ne gère pas deux modes d'affichage portrait et paysage. Il serait possible de la compléter pour lui permettre de gérer les deux modes d'affichage.

Au lancement, l'application demandera à l'utilisateur de saisir un montant et un taux de réduction. Un clic sur le bouton calculer doit afficher une deuxième vue qui affichera le montant à payer. Un fragment va gérer chacune des 2 vues. Cet exemple montre bien qu'un fragment peut être considéré comme une sorte de sous-activité.





L'application comportera 3 fichiers *layout* :

- ✓ *activity_main* contient seulement un *FrameLayout,* doté d'un identifiant accessible ensuite en écrivant `R.id.container`
- ✓ *layout_saisie* un *LinearLayout* avec les *widgets* de la vue correspondant à la saisie
- ✓ *layout_resultat* un *LinearLayout* avec deux *TextView* pour l'affichage du résultat

Selon la situation (saisie ou affichage du résultat), le *layout* décrit par *layout_saisie*, ou bien celui décrit par *layout_resultat* ira prendre la place du *FrameLayout* de la vue de l'activité principale.

L'application comporte 4 fichiers Java :

- ✓ l'activité principale *MainActivity.java*
- ✓ *FragmentSaisie.java*
- ✓ *FragmentResultat.java*
- ✓ *InterfaceCalcul.java*

Interface pour la méthode *calcul*

La méthode de calcul doit bien sûr calculer le montant à payer, mais elle doit aussi afficher le fragment de résultat. L'affichage de ce fragment, via un ajout au gestionnaire de fragment, ne peut se faire que dans l'activité principale. Or la méthode de calcul doit être invoquée dans le fragment de saisie, puisque c'est dans ce fragment que le clic sur le bouton « calculer » est intercepté.

Pour prendre en compte ces particularités, il est nécessaire de définir une interface contenant la méthode de calcul.

```
public interface InterfaceCalcul {

      /**
       * Méthode permettant de calculer un montant à payer et de l'afficher
       * dans la vue appropriée
       * @param montant   montant initial
       * @param taux      taux de réduction en pourcentage
       */
       void calculer(double montant, double taux);
}
```

Activité principale

Dans l'activité principale, au premier appel à *onCreate*, il faut placer la vue *layout_saisie* à la place du conteneur défini dans le fichier *activity_main.xml* :

```
if (savedInstanceState == null) {
    getFragmentManager().beginTransaction()
                        .add(R.id.container, new FragmentSaisie())
                        .commit();
}
```

Notez l'appel à la méthode ***add*** : `add(R.id.container, new FragmentSaisie())`
Il s'agit ici d'ajouter un fragment. Son comportement est décrit par la classe ***FragmentSaisie***, d'où l'appel au constructeur de cette classe. Ce fragment ira prendre la place du *layout* identifié par `R.id.container` dans la vue courante.

On définit également une méthode ***calculer*** que le fragment de saisie invoquera lors du clic sur le bouton « calculer ». Dans cette méthode, il s'agit de calculer le prix à payer, et de remplacer le fragment de saisie par le fragment de résultat :

```
getFragmentManager().beginTransaction()
                    .replace(R.id.container,
                             FragmentResultat.getInstance(aPayer))
                    .addToBackStack("result")
                    .commit();
```

Le remplacement du fragment s'effectue grâce à l'appel à la méthode ***replace*** :

```
replace(R.id.container, FragmentResultat.getInstance(aPayer))
```

Le fragment de résultat est obtenu par un appel à la méthode ***getInstance*** de la classe ***FragmentResultat***. Celle-ci renvoie le fragment de résultat, initialisé de manière à ce que l'argument *aPayer* soit affiché dans la zone de résultat.

Fragment de saisie

Lorsque ce fragment est attaché à l'activité principale, il est nécessaire de conserver une référence sur celle-ci. En effet, on aura par la suite besoin d'invoquer la méthode *calcul* définie dans l'activité parente. Pour conserver cette référence, on déclare un attribut de type ***InterfaceCalcul*** :

```
/** Ecouteur pour gérer le clic sur le bouton calculer
 *  Il s'agit de l'écouteur de l'activité parente
 */
private InterfaceCalcul ecouteurActiParente;
```

Et on l'initialise dans la méthode ***onAttach*** :

```
ecouteurActiParente = (InterfaceCalcul) activite;
```

Lors du clic sur le bouton de calcul, il est nécessaire d'accéder aux valeurs saisies par l'utilisateur et d'invoquer la méthode calculer de l'activité parente. L'écouteur du bouton de calcul est donc codé de la manière suivante :

```
/**
 * Classe qui contient la méthode permettant de gérer le clic sur le bouton
 * calculer. Les valeurs saisies sont transmises à la méthode calculer de
 * l'écouteur de la classe parente
 */
public class EcouteurBoutonCalculer implements View.OnClickListener {

    @Override
    public void onClick(View v) {
        double prix = Double.parseDouble(saisiePrix.getText().toString());
        double taux = Double.parseDouble(saisieTaux.getText().toString());
        ecouteurActiParente.calculer(prix, taux);
    }
}
```

Fragment de résultat

Dans le fragment de résultat, la méthode ***getInstance*** est codée de la manière suivante :

```
public static Fragment getInstance(double valeur) {
    FragmentResultat fragment = new FragmentResultat();

    /*
    * On crée un bundle dans le lequel on place la valeur résultat.
    * Ce bundle est ajouté en tant qu'arguemnt du fragment
    */
    Bundle bundle = new Bundle();
    bundle.putDouble(CLE_RESULTAT, valeur);
    fragment.setArguments(bundle);
    return fragment;
}
```

La méthode renvoie une instance de type ***FramentResultat***, ayant pour argument un *Bundle* dans lequel la valeur du résultat à afficher a été insérée.

Dans la classe du fragment de résultat, on a défini une méthode valeurResultat qui renvoie la valeur du résultat présente en tant qu'argument du fragment :

```
public double valeurResultat() {
    return getArguments().getDouble(CLE_RESULTAT);
}
```

Un appel à cette méthode permettra d'initialiser la zone de résultat :

```
zoneResultat.setText(Double.toString(valeurResultat()));
```

Remarques

Un fragment peut communiquer avec son activité parente. Il doit d'abord accéder à celle-ci en invoquant ***getActivity()***

Une activité peut communiquer avec un fragment qu'elle contient. Pour ce faire, elle y accède d'abord avec un appel à ***findFragmentById***

Par défaut, une transaction effectuée sur un fragment n'est pas ajouté à la *back stack (pile d'appels)*. Cela implique que lorsque l'utilisateur cliquera sur le bouton "back", il ne pourra pas revenir à l'état antérieur. Si l'on ne souhaite pas adopter ce comportement par défaut, il faut demander explicitement à ce que le fragment soit ajouté à la pile. Pour ce faire, on utilise la méthode ***addToBackStack(String nom).***

5) Annexe 1 : Exemple fragments statiques – Code complet

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Vue de l'activité principale, en mode paysage.
     2 fragments sont situés côte à côte :
            celui de gauche permet de saisir des valeurs
            celui de droite affiche le résultat
     layout_main.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:orientation="horizontal"
    tools:context="com.exemple.cours.android.exfragmentstatique.MainActivity">


    <!-- fragment de gauche : il permet de saisir les montants
         l'espace réservé sera occupé par layout_fragment_saisie -->
    <fragment
        android:id="@+id/fragmentSaisie"
        android:layout_width="150dip"
        android:layout_height="match_parent"
        class="com.exemple.cours.android.exfragmentstatique.FragmentSaisie">
    </fragment>

    <!-- fragment de gauche : il permet d'afficher le résultat
         l'espace réservé sera occupé par layout_fragment_resultat -->
    <fragment
        android:id="@+id/fragmentResultat"
        android:layout_width="match_parent"
```

```
        android:layout_height="match_parent"
        class="com.exemple.cours.android.exfragmentstatique.FragmentResultat">
    </fragment>

</LinearLayout>
```

***layout_main.xml* (en mode paysage)**

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Vue de l'activité principale, en mode portrait.
     1 seul fragment est affiché : celui qui permet de saisir des valeurs
     layout_main.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal" android:layout_width="match_parent"
    android:layout_height="match_parent">

    <!-- fragment de gauche : il permet de saisir les montants
         l'espace réservé sera occupé par layout_fragment_saisie -->
    <fragment
        android:id="@+id/fragmentSaisie"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        class="com.exemple.cours.android.exfragmentstatique.FragmentSaisie">
    </fragment>

</LinearLayout>
```

***layout_main.xml* (en mode portrait)**

```


<!-- Vue décrivant le fragment permettant d'effectuer la saisie des montants.
     Si mode portrait : ce fragment est le seul de la vue
     Si mode paysage : ce fragment occupe la moitié gauche de l'écran
     layout_fragment_saisie.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin">

    <!-- invite pour la saisie du prix de base -->
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:paddingTop="@dimen/activity_vertical_margin"
        android:layout_gravity="center"
        android:text="@string/texte_prix"/>

    <!-- zone de saisie du prix de base -->
    <EditText
        android:id="@+id/saisie_prix"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:paddingTop="@dimen/activity_vertical_margin"
        android:hint="@string/indication_prix"
        android:inputType="number|numberDecimal"
        android:layout_gravity="center" />

    <!-- invite pour la saisie du taux de réduction -->
    <TextView
```

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:paddingTop="@dimen/activity_vertical_margin"
        android:layout_gravity="center"
        android:text="@string/texte_taux"/>

    <!-- zone de saisie du taux de réduction -->
    <EditText
        android:id="@+id/saisie_taux"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:paddingTop="@dimen/activity_vertical_margin"
        android:paddingBottom="@dimen/activity_vertical_margin"
        android:hint="@string/indication_taux"
        android:inputType="number|numberDecimal"
        android:layout_gravity="center" />

    <!-- bouton pour lancer le calcul de la réduction -->
    <Button
        android:id="@+id/btn_calculer"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:paddingTop="@dimen/activity_vertical_margin"
        android:layout_gravity="center"
        android:text="@string/btn_calculer"/>

</LinearLayout>
```

layout_fragment_saisie.xml

```
<!-- Vue décrivant le fragment permettant d'afficher le résultat
     layout_fragment_resultat.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin">

    <!-- texte de présentation -->
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:paddingTop="@dimen/activity_vertical_margin"
        android:paddingBottom="@dimen/activity_vertical_margin"
        android:layout_gravity="center"
        android:text="@string/texte_a_payer"/>

    <!-- texte qui affiche le résultat : le montant à payer -->
    <TextView
        android:id="@+id/zone_resultat"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:paddingTop="@dimen/activity_vertical_margin"
        android:paddingBottom="@dimen/activity_vertical_margin"
        android:textSize="@dimen/taille_resultat"
        android:layout_gravity="center" />

</LinearLayout>
```

layout_fragment_resultat.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<!-- Vue permettant d'afficher le résultat, en mode portrait
     layout_resultat_portrait.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout_width="match_parent"
    android:layout_height="match_parent">

    <fragment
        android:id="@+id/fragmentResultat"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        class="com.exemple.cours.android.exfragmentstatique.FragmentResultat">
    </fragment>

</LinearLayout>
```

layout_resultat_portrait.xml

```java
/*
 * Exemple illustrant la notion de fragment statique
 * MainActivity.java
 */
package com.exemple.cours.android.exfragmentstatique;

import android.app.Activity;
import android.os.Bundle;

/**
 * Activité principale de l'exemple illustrant la notion de fragment statique.
 * L'application est divisée en 2 volets :
 *   un volet permet à l'utilisateur de saisir un montant et un taux de réduction
 *   l'autre volet affiche le montant à payer
 * Si le terminal est en mode paysage : les 2 volets sont affichées côte à côte, une
seule
 * activité gère l'ensemble
 * Si le terminal est en mode portrait :
 *    une première activité, la principale, gère la saisie des valeurs
 *    une deuxième activité gère l'affichage du résultat
 * @author Servières
 * @version 1.0
 */
public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        /*
          * le fichier layout existe en 2 versions :
          *    une version en mode paysage : 2 fragments sont placés côte à côte
          *       l'un gère les saisies, l'autre l'affichage du résultat
          *    une version mode portrait : un fragment qui gère seulement la saisie
          */
        setContentView(R.layout.layout_main);
    }
}
```

MainActivity.java

```
/*
 *  Gestion du fragment permettant de saisir les valeurs : montant et taux
 *  FragmentSaisie.java
 */
package com.exemple.cours.android.exfragmentstatique;

import android.content.Intent;
import android.os.Bundle;
import android.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.Button;
import android.widget.EditText;


/**
 * Classe décrivant le fragment qui permet de saisir le montant initial et le taux
 * de réduction.
 * Si l'application est en mode paysage : le résultat sera affiché sur le fragment de
 * droite.
 * Si elle est en mode paysage : une deuxième activité est créée (ActiviteResultat).
C'est
 * elle qui gèrera l'affichage du résultat. Le résultat lui sera transmis via
l'intention
 * de lancement.
 * @author Servières
 * @version 1.0
 */
public class FragmentSaisie extends Fragment {

    /** clé pour identifier le résultat transmis via l'intention */
    public static final String CLE_RESULTAT = "cle_resultat";

    /** Zone de saisie du montant initial */
    private EditText saisiePrix;

    /** Zone de saisie du taux de réduction */
    private EditText saisieTaux;

    /** Bouton pour effectuer le calcul */
    private Button boutonCalculer;


    /**
     * Classe décrivant le comportement de l'écouteur du bouton
     */
    public class EcouteurBoutonCalculer implements View.OnClickListener {
        @Override
        public void onClick(View v) {
            double prix = Double.parseDouble(saisiePrix.getText().toString());
            double taux = Double.parseDouble(saisieTaux.getText().toString());

            // calcul du résultat
            double aPayer = prix - ((prix * taux) / 100);
            String montantAAfficher = Double.toString(aPayer);

            // on récupère le fragment d'affichage du résultat, s'il existe
            FragmentResultat fragment =
                    (FragmentResultat)
```

```
                        getFragmentManager().findFragmentById(R.id.fragmentResultat);

            /* Si le fragment de résultat existe, on met à jour le montant résultat
             * affiché sur ce fragment (l'application est donc en mode paysage)
             */
            if (fragment != null && fragment.isInLayout()) {
                fragment.majMontant(montantAAfficher);
            } else {

                /*
                 * le fragment résultat n'existe pas : l'application est donc en mode
                 * paysage. Il faut créer l'activité chargée d'afficher le résultat
                 * et lui communiquer le montant à afficher.
                 */
                Intent intention = new Intent(getActivity(), ActiviteResultat.class);
                intention.putExtra(CLE_RESULTAT, montantAAfficher);
                startActivity(intention);
            }
        }
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }


    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
    }

    @Override
    public View onCreateView(LayoutInflater inflater,
                             ViewGroup container,
                             Bundle savedInstanceState) {

        // on associe la vue du fragment
        View view = inflater.inflate(R.layout.layout_fragment_saisie,
                container,
                false);

        // on récupère un accès aux différents widgets de la vue
        saisiePrix = (EditText)view.findViewById(R.id.saisie_prix);
        saisieTaux = (EditText)view.findViewById(R.id.saisie_taux);
        boutonCalculer = (Button)view.findViewById(R.id.btn_calculer);

        // on associe un écouteur au bouton
        boutonCalculer.setOnClickListener(new EcouteurBoutonCalculer());
        return view;
    }
}
```

FragmentSaisie.java

```
/*
 *  Gestion du fragment permettant d'afficher le résulat
 *  (utile si le terminal est en mode paysage)
 *  FragmentResultat.java
 */
package com.exemple.cours.android.exfragmentstatique;
```

```
import android.os.Bundle;
import android.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.TextView;

/**
 * Classe décrivant le fragment qui permet d'afficher le résultat : le montant
 * à payer tenant compte de la réduction.
 * La valeur à afficher est transmise via un appel à la méthode majMontant
 * @author Servières
 * @version 1.0
 */
public class FragmentResultat extends Fragment {

    /** Zone permettant d'afficher le résultat */
    private TextView zoneResultat;


    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {

        // initialisation de la vue du fragment, et de la zone de résultat
        View view = inflater.inflate(R.layout.layout_fragment_resultat,
                container,
                false);
        zoneResultat = (TextView) view.findViewById(R.id.zone_resultat);
        return view;
    }

    /**
     * Méthode invoquée pour mettre à jour la valeur du résultat, le montant à payer
     * @param montant   montant sous la forme d'une chaîne de caractères
     */
    public void majMontant(String montant) {
        zoneResultat.setText(montant);
    }
}
```

FragmentResultat.java

```
/*
 *  Activité secondaire de l'exemple illustrant la notion de fragment statique
 *  ActiviteResultat.java
 */
package com.exemple.cours.android.exfragmentstatique;

import android.app.Activity;
import android.content.Intent;
import android.content.res.Configuration;
import android.content.res.Resources;
import android.os.Bundle;
import android.util.Log;
import android.widget.TextView;
```

```
/**
 * Classe décrivant l'activité qui permet d'afficher le résultat : le montant
 * à payer tenant compte de la réduction.
 * La valeur à afficher est transmise via l'intention à l'origine du lancement
 * de l'activité
 * @author Servières
 * @version A1.0
 */
public class ActiviteResultat extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        try {
            /*
             * si le terminal est en mode portrait, l'appel suivant
             * se déroule sans erreur
             */
            setContentView(R.layout.layout_resultat_portrait);

            // on récupère l'intention et le texte résultat
            Intent intention = getIntent();
            Bundle extra = getIntent().getExtras();
            String resultatBis = extra.getString(FragmentSaisie.CLE_RESULTAT);

            String resultat = intention.getStringExtra(FragmentSaisie.CLE_RESULTAT);

            // on affiche le résultat
            TextView texte = (TextView) findViewById(R.id.zone_resultat);
            if (texte == null) {
                 texte.setText("");
            } else {
                 texte.setText(resultat);
            }

        } catch(Resources.NotFoundException erreur) {
            /*
             * se produit si le terminal est en mode paysage.
             * Il faut détruire l'activité résultat courante.
             * L'activité principale sera rechargée
             */
            finish();

        }
    }
}
```

ActiviteResultat.java

6) Annexe 2 : Exemple fragments dynamiques – Code complet

```
<?xml version="1.0" encoding="utf-8"?>
<!-- vue de l'activité principale de l'application permettant d'illustrer
     la notion de fragment dynamique
     activity_main.xml -->
```

```
<!-- ce FrameLayout sert de réceptacle pour un futur fragment
     noter qu'il possède pour identifiant "container"-->
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/container"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.exemple.cours.android.exfragment.MainActivity">
</FrameLayout>
```

activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Vue qui sera associée au fragment permettant d'effectuer la saisie des montants.
     layout_fragment_saisie.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin">

    <!-- invite pour saisir le montant -->
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:paddingTop="@dimen/activity_vertical_margin"
        android:layout_gravity="center"
        android:text="@string/texte_prix"/>

    <!-- zone de saisie du montant -->
    <EditText
        android:id="@+id/saisie_prix"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:paddingTop="@dimen/activity_vertical_margin"
        android:inputType="number"
        android:layout_gravity="center" />

    <!-- invite pour saisir le taux de réduction -->
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:paddingTop="@dimen/activity_vertical_margin"
        android:layout_gravity="center"
        android:text="@string/texte_taux"/>

    <!-- zone de saisie du taux de réduction -->
    <EditText
        android:id="@+id/saisie_taux"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:paddingTop="@dimen/activity_vertical_margin"
        android:paddingBottom="@dimen/activity_vertical_margin"
        android:inputType="number"
        android:layout_gravity="center" />

    <!-- bouton pour lancer le calcul du prix à payer -->
    <Button
        android:id="@+id/btn_calculer"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
```

```
        android:paddingTop="@dimen/activity_vertical_margin"
         android:layout_gravity="center"
        android:text="@string/btn_calculer"/>

</LinearLayout>
```

Layout_saisie.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Vue qui sera associé au fragment permettant d'afficher le résultat
     layout_resultat.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin">


    <!-- texte de présentation -->
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:paddingTop="@dimen/activity_vertical_margin"
        android:paddingBottom="@dimen/activity_vertical_margin"
        android:layout_gravity="center"
        android:text="@string/texte_a_payer"/>

    <!-- affichage du résultat : le montant à payer -->
    <TextView
        android:id="@+id/zone_resultat"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:paddingTop="@dimen/activity_vertical_margin"
        android:paddingBottom="@dimen/activity_vertical_margin"
        android:textSize="@dimen/taille_resultat"
        android:layout_gravity="center" />

</LinearLayout>
```

Layout_resultat.xml

```
/*
 *  Activité principale de l'application permettant d'illustrer la notion de
 *  fragment dynamique.
 *  MainActivity.java
 */
package com.exemple.cours.android.exfragment;

import android.app.Activity;
import android.os.Bundle;

/**
 *  Activité principale d'une application permettant de calculer le montant
 *  à payer à partir d'un montant initial et d'un taux de réduction entrés par
 *  l'utilisateur. L'application comporte une seule activité et 2 fragments.
 *  L'un gère la vue permettant d'effectuer les saisies, l'autre celle permettant
 *  d'afficher le résultat.
 *  La classe implémente l'interface InterfaceCalcul et contient donc la méthode
 *  calculer qui sera invoquée pour calculer le montant à payer et l'afficher
 *  sur le fragment gérant l'affichage du résultat
```

```
 *  @author Servières
 *  @version 1.0
 */
public class MainActivity extends Activity implements InterfaceCalcul{

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

        /*
          * s'il s'agit du 1er appel à onCreate : on crée le fragment de saisie
          * et on l'ajoute dans le FrameLayout de la vue courante
          */
        if (savedInstanceState == null) {
            getFragmentManager().beginTransaction()
                    .add(R.id.container, new FragmentSaisie())
                    .commit();
        }
    }

    /**
     * Calcule le montant à payer et prenant en compte la réduction
     * Le fragment destiné à afficher le résultat va remplacer le fragment courant
     * @param montant   montant saisi par l'utlisateur
     * @param taux      taux de réduction saisi par l'utilisateur
     */
    public void calculer(double montant, double taux) {
        double aPayer = montant - (montant * taux / 100);

        /*
         * à noter : le fragment destiné à gérer l'affichage du résultat
         * est créé et initialisé avec le montant à payer
         */
        getFragmentManager().beginTransaction()
                .replace(R.id.container,
                        FragmentResultat.getInstance(aPayer))
                .addToBackStack("result")
                .commit();
    }
}
```

MainActivity.java

```
package com.exemple.cours.android.exfragment;

/**
 * Inteface contenant la méthode permettant d'effectuer le calcul d'un
 * montant à payer et de l'afficher dans la vue appropriée
 * @author Servières
 * @version 1.0
 */
public interface InterfaceCalcul {

    /**
     * Méthode permettant de calculer un montant à payer et de l'afficher
     * dans la vue appropriée
     * @param montant   montant initial
     * @param taux      taux de réduction en pourcentage
     */
    void calculer(double montant, double taux);

}
```

InterfaceCalcul.java

```
/*
 * Exemple illustrant la notion de fragment dynamique : fragment qui gère les saisies
 * FragmentSaisie.java
 */
package com.exemple.cours.android.exfragment;
. . .

/**
 * Ce fragment gère la vue permettant d'effectuer les saisies : montant initial et
 * taux de réduction. Un clic sur le bouton de calcul doit provoquer un appel
 * à la méthode calculer de l'activité parente.
 * @author Servières
 * @version 1.0
 */
public class FragmentSaisie extends Fragment {

    /** Zone de saisie du montant initial */
    private EditText saisiePrix;

    /** Zone de saisie du taux de réduction */
    private EditText saisieTaux;

    /** Bouton pour effectuer le calcul */
    private Button boutonCalculer;

    /** Ecouteur pour gérer le clic sur le bouton calculer
     *  Il s'agit de l'écouteur de l'activité parente */
    private InterfaceCalcul ecouteurActiParente;


    @Override
    public void onCreate(Bundle leBundle){
        super.onCreate(leBundle);
    }

    /**
     * Classe qui contient la méthode permettant de gérer le clic sur le bouton
     * calculer. Les valeurs saisies sont transmises à la méthode calculer de
     * l'écouteur de la classe parente
     */
    public class EcouteurBoutonCalculer implements View.OnClickListener {

        @Override
        public void onClick(View v) {
            double prix = Double.parseDouble(saisiePrix.getText().toString());
            double taux = Double.parseDouble(saisieTaux.getText().toString());
            ecouteurActiParente.calculer(prix, taux);
        }
    }


    @Override
    public View onCreateView(LayoutInflater inflater,
                             ViewGroup container,
                             Bundle savedInstanceState) {

        // la vue décrite par layout_saisie est associée au fragment
        View view = inflater.inflate(R.layout.layout_saisie,
                container,
```

```
                false);

        // on récupère un accès aux widgets
        saisiePrix = (EditText)view.findViewById(R.id.saisie_prix);
        saisieTaux = (EditText)view.findViewById(R.id.saisie_taux);
        boutonCalculer = (Button)view.findViewById(R.id.btn_calculer);

        // on associe un écouteur au bouton de calcul
        boutonCalculer.setOnClickListener(new EcouteurBoutonCalculer());
        return view;
    }


    @Override
    public void onAttach(Activity activite) {
        super.onAttach(activite);

        /*
          * on récupère un accès à l'écouteur de l'activité parente
          * C'est via cet écouteur que le résultat pourra être affiché
          */
        try {
            ecouteurActiParente = (InterfaceCalcul) activite;
        } catch (ClassCastException e) {
            throw new ClassCastException(activite.toString()
                    + " must implement calculer");
        }
    }
}
```

FragmentSaisie.java

```
/*
 * Exemple illustrant la notion de fragment dynamique : fragment qui gère l'affichage
 * du résultat
 * FragmentResultat.java
 */
package com.exemple.cours.android.exfragment;

import android.app.Activity;
import android.os.Bundle;
import android.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.TextView;


/**
 * Ce fragment gère la vue permettant d'effectuer d'afficher le résultat :
 * le montant à payer
 * @author Servières
 * @version 1.0
 */
public class FragmentResultat extends Fragment {

    /** clé pour la valeur constituant le résultat à afficher */
    private static final String CLE_RESULTAT = "valeur_resultat";


    /** Zone pour l'affichage du résultat */
    private TextView zoneResultat;
```

```
    @Override
    public void onCreate(Bundle leBundle) {
        super.onCreate(leBundle);
    }


    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {

        // pour associer la vue layout_resultat au fragment courant
        View view = inflater.inflate(R.layout.layout_resultat,
                container,
                false);

        /*
         * on place le résultat dans la zone prévue à cet effet
         * Le résultat est présent en tant qu'argument du fragment
         */
        zoneResultat = (TextView)view.findViewById(R.id.zone_resultat);
        zoneResultat.setText(Double.toString(valeurResultat()));
        return view;
    }


    /**
     * Cette méthode crée un fragment de type FragmentResultat et lui associe un
     * argument, la valeur du résultat.
     * @param valeur  valeur transmise en tant que paramère du fragment
     * @return le fragment crée
     */
    public static Fragment getInstance(double valeur) {
        FragmentResultat fragment = new FragmentResultat();

        /*
         * On crée un bundle dans le lequel on place la valeur résultat.
         * Ce bundle est ajouté en tant qu'arguemnt du fragment
         */
        Bundle bundle = new Bundle();
        bundle.putDouble(CLE_RESULTAT, valeur);
        fragment.setArguments(bundle);
        return fragment;
    }

    @Override
    public void onAttach(Activity activite) {
        super.onAttach(activite);
    }

    /**
     * Renvoie la valeur du résultat présente en tant qu'argument du fragment
     * courant
     * @return  la valeur du résultat
     */
    public double valeurResultat() {
        return getArguments().getDouble(CLE_RESULTAT);
    }

}
```

FragmentResultat.java

Améliorer la réactivité des applications Les threads et les tâches asynchrones

I) Introduction

Remarque : un utilisateur est capable de détecter un ralentissement d'une application en cours d'utilisation dès qu'il dure au moins 100 ms.

Android surveille la réactivité des applications : les applications qui ne sont pas suffisamment réactives sont détruites. Pour Android, une application n'est pas assez réactive si le délai de réponse, à une action de l'utilisateur, est supérieur à 5 secondes (ou bien si une instance de *BroadcastReceiver* ne se termine pas en moins de 10 secondes). Dans ce cas, l'erreur qui est provoquée par le système se nomme **ANR** pour *Application Not Responding*. Android affiche alors une boîte de dialogue invitant l'utilisateur a quitter l'application.

Pour éviter l'erreur ANR, il ne faut pas réaliser d'opérations trop longues dans le *thread* principal. Au contraire, il faut les réaliser en parallèle du *thread* principal.

1.1) Notions de processus et de thread

Tous les programmes Android s'exécutent dans un processus. Un processus est une instance de programme en cours d'exécution. Il contient les instructions du programme, mais aussi les données (variables) manipulées par celui-ci, et qui représentent l'état courant.

Plus précisément, ce n'est pas le processus qui exécute le code, mais l'un de ses constituants, appelés *threads* (ou fils d'exécution en français). Un processus Android peut être constitué d'un ou de plusieurs *threads*, auquel cas ils vont s'exécuter en parallèle. Les *threads* peuvent communiquer entre eux, ainsi que les processus.

Dès qu'un composant d'une application est lancé (par exemple, une activité), si cette application n'a pas encore de processus actif, un processus est créé. Au départ, il ne possède qu'un seul *thread* : le *thread* principal.

Par contre, si un composant démarre alors qu'il y a déjà un processus pour cette application, le composant se lancera dans le processus en utilisant le même *thread*. En effet, par défaut, tous les composants d'une même application se lancent dans le même processus. Ce comportement est presque toujours celui que l'on souhaite. Si l'on souhaite qu'un composant s'exécute dans un autre processus, il

faut en faire la déclaration dans le fichier *AndroidManifest.xml* (par exemple, continuer à faire un traitement même si l'utilisateur ferme l'application).

Quelques explications sur le *thread* UI :

Quand la première activité d'une application est lancée, le système crée un *thread* principal dans lequel s'exécutera l'application, on dit qu'il s'agit du *thread* UI (user interface) puisque c'est lui qui va gérer les interactions avec la vue.

En particulier, ce *thread* principal qui doit être capable de gérer les évènements rapidement (événements système, interaction avec l'utilisateur …). S'il est occupé avec diverses opérations longues ou attend une connexion réseau par exemple, il ne répondra pas assez rapidement et une erreur ANR se produira.

Quand on effectue une modification sur une vue, elle ne se réalise pas instantanément. Dans un premier temps, un événement est créé, sous la forme d'un message envoyé dans une pile de messages. Le *thread* UI accède régulièrement à la pile des messages et les dépile pour les traiter. Ce *thread* exécute aussi toutes les méthodes dites de *callback* du système (comme *onCreate*, *onClick* … donc toutes les méthodes appelées automatiquement par le système). Si le *thread* est trop occupé, il ne pourra pas traiter rapidement les interactions avec l'utilisateur, ni les méthodes de *callback* et c'est alors qu'il déclenchera une erreur ANR.

Il faut donc respecter 2 règles très importantes : ne jamais bloquer le *thread* UI en y réalisant des opérations longues, et ne pas manipuler l'interface utilisateur en dehors du *thread* UI.

Concrètement, on évitera de réaliser les opérations suivantes dans le *thread* UI :
- ✓ Accès réseau, même si l'opération est courte en théorie
- ✓ Certaines opérations sur les bases de données, surtout les sélections multiples
- ✓ Les accès fichiers
- ✓ Les accès matériels, car des temps de chargements peuvent être très longs

1.2) Techniques pour améliorer la réactivité des applications

Pour améliorer la réactivité des applications, plusieurs techniques sont à la disposition du programmeur.

1) On peut utiliser des *threads* : il s'agit donc de créer un fil d'exécution parallèle pour traiter les tâches longues. Android utilise l'API standard de gestion des *threads* du langage Java standard (voir section suivante)

2) On peut utiliser **la classe *AsyncTask*** pour réaliser des traitements en tâche de fond, ces traitements permettent des interactions avec l'utilisateur (voir l'une des sections suivantes dans ce chapitre)

3) On peut utiliser la notion de **services**. Il s'agit de tâches de fond qui exécutent en arrière plan les traitements longs. Les services n'ont pas d'interface utilisateur (ils peuvent toutefois envoyer une notification). Ils continuent à s'exécuter même si les activités de l'application sont en pause ou arrêtées. Attention toutefois, une mauvaise conception d'un service peut

provoquer une erreur ANR. Par exemple, un service doit créer des *threads*, sinon on obtiendra l'erreur ANR (voir un autre chapitre).

2) Utilisation des threads

Les *threads* séparent un processus en plusieurs files d'exécution. En apparence, ils sont exécutés en parallèle. Mais en réalité, c'est seulement le temps d'exécution du processeur qui est réparti entre les *threads*.

Leur utilisation engendre une amélioration des performances, puisque les *threads* en attente cèdent leur droit à s'exécuter aux autres *threads* (par exemple : attente que l'utilisateur saisissent des données). Cependant, la réactivité des applications est privilégiée et donc le *thread* principal de l'application (le *thread* UI) est celui qui a toujours la plus haute priorité.

Pour créer un *thread*, on peut hériter de la classe ***Thread*** qui implémente l'interface ***Runnable***, ou bien implémenter directement l'interface ***Runnable***. Celle-ci possède une seule méthode : `void run()`
Dans les 2 cas, il faudra implémenter cette méthode ***run***.

Schéma de construction d'un *thread* en utilisant l'interface *Runnable*

a) créer une classe qui implémente l'interface *Runnable*
b) dans la méthode `void run()` de cette classe, placer les opérations longues
c) créer une instance de la classe *Thread*. On donne en paramètre du constructeur la classe créée précédemment (celle-ci peut être anonyme, comme montré dans le schéma ci-dessous)
d) appliquer la méthode *start* sur l'instance créée : le code de la méthode *run* sera ainsi exécuté dans un nouveau *thread*

```
new Thread(new Runnable() {

    @Override
    public void run() {

        // placer ici les opérations longues
        ……
    }

}).start() ;
```

Schéma de construction d'un *thread*

Il faut toutefois prêter attention au problème des accès concurrents entre les *threads*. En effet, que se passe-t-il si deux *threads* tentent de modifier la même ressource ? Par exemple, deux *threads* qui mettent à jour, en même temps, le même enregistrement d'une base de données. Le résultat sera aléatoire.

Donc, autant que possible, il faut éviter les accès concurrents d'une même ressource par différents *threads*. Si on ne peut éviter ces accès concurrents, il faut limiter l'exécution d'un bloc d'instructions à

un seul *thread* à la fois (mot-clé *synchronized*). La manière de coder est la même que dans le langage Java standard.

On obtient alors un code dit « protégé » sans situation de compétition (ou « *thread* safe »). Attention : le code de l'interface utilisateur n'est pas « *thread* safe ». Il ne faut donc, surtout pas, modifier l'interface à partir d'un *thread* qui ne serait pas le *thread* UI : on obtiendrait une erreur au niveau du système.

Comment dans un *thread* créé par l'utilisateur accéder à l'interface graphique ?

Pourtant très souvent, on souhaitera, par exemple, afficher, sur la vue de l'activité courante, le résultat d'un traitement effectué dans un *thread* dédié. Pour ce faire, il faudra « poster » un message en direction du *thread* UI pour lui demander de faire la mise à jour de l'interface.

Concrètement, si l'on souhaite modifier le texte d'un widget *phraseResultat* de type *TextView*, on écrira, à l'intérieur de la méthode *run* du *thread* dédié :

```
phraseResultat.post(new Runnable() {
    public void run() {
        phraseResultat.setText(.... le message à afficher .....);
    }
});
```

La méthode *post* de la classe *View* a le profil suivant :

```
boolean post(Runnable action)
```

Son rôle est d'ajouter à la pile des messages du *thread* UI le traitement décrit par l'action argument. Le booléen renvoyé indique que l'ajout à la pile des messages s'est correctement passé.

Pour déléguer une action au *thread* UI, il existe d'autres possibilities :
- appeler la méthode de la classe *Activity* :
  `void runOnUiThread(Runnable action)`
- invoquer
  `boolean postDelayed(Runnable action, long delayMillis)`
  l'action sera ajoutée à la pile des messages seulement au bout d'un certain délai

Remarque

Il existe une classe nommée ***Handler*** qui gère les interactions avec un *thread*, sous la forme de messages que l'on peut lui envoyer et qu'il va pouvoir traiter.

3) Utilisation de la classe AsyncTask

Les tâches asynchrones correspondent à un fil d'exécution qui peut être en interaction avec l'interface utilisateur. Il s'agit en fait d'un *thread* d'arrière plan qui envoie des messages à l'interface.

Pour utiliser cette notion, il faut définir une classe fille de la classe prédéfinie ***AsyncTask***. La classe ***AsyncTask*** est générique et paramétrée par 3 types :

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

- *Params* est le type des paramètres passés à la tâche,
- *Progress* est le type de l'indicateur de progression (pour indiquer l'avancement de la tâche)
- *Result* est le type du résultat de l'exécution de la tâche

Schéma d'instanciation de la classe *AsyncTask*

Une classe qui hérite de ***AsyncTask*** respectera le schéma suivant, par exemple :

```
class TacheAsynchrone extends AsyncTask<String, Integer, Double> {

    protected void onPreExecute() {
        // on met à jour l'interface juste avant le début de l'exécution
        // de la tâche d'arrière plan. Les instructions de la méthode sont
        // executées dans le thread UI
    }

    protected Double doInBackground(String... params) {
        // opérations qui seront exécutées dans le thread d'arrière plan
        ……
    }

    protected void onPostExecute(Double resultat) {
        // opérations exécutées dans le thread UI
        // cette méthode est appelée automatiquement
        // le paramètre resultat a la valeur renvoyée par doInBackground
        ……
    }

    protected void onProgressUpdate(Integer...  progress) {
        // méthode exécutée automatiquement dans le thread UI
        // en réponse à l'appel de publishProgress()
        // le paramètre progress pourrait indiquer le pourcentage du traitement
        // réalisé
        ……
    }
}
```

Remarques

- dans les listes de paramètres, les 3 points ... indiquent un nombre illimité de paramètres de ce type. Par exemple :

```
int somme(int ... nombres)
```

signifie que la méthode somme admet 1, 2, 3 ou plus paramètres de type *int*.

- si l'un des paramètres de la classe générique n'est pas utile, on indique à l'instanciation : *void*
- la seule méthode que nous sommes obligés de définir est *doInBackground*

Comment démarrer une tâche de fond ?

Pour démarrer une tâche de fond de ce type, on invoque la méthode ***execute*** sur la tâche :

```
new TacheAsynchrone().execute("chaine1", "chaine2", "chaine3");
```

Les paramètres de la méthode *execute* sont ceux passés à la méthode *doInBackground(),* méthode qui sera exécutée dans le *thread* d'arrière plan.

Précisément, le prototype de cette méthode est :

```
final   AsyncTask<Params, Progress, Result>   execute(Params ... params)
```

Que se passe-t-il lors de l'appel à *execute* ?

Les quatres méthodes de *callback* sont appelées automatiquement : ***onPreExecute***, ***doInBackground***, ***publishProgress***, et ***onPostExecute***.

a) **`void onPreExecute()`** est exécutée avant que la tâche de fond proprement dite ne soit lancée. Elle sert à effectuer des initialisations, ou modifications de l'interface utilisateur, par exemple. Son code s'exécute dans le *thread* UI.

b) **`Result doInBackground(Params ... params)`** : c'est dans cette méthode que le traitement d'arrière plan doit être codé. La méthode renvoie un résultat qui sera transmis à *onPostExecute*.
On peut appeler dans cette méthode, la méthode ***publishProgress*** qui a le profil suivant :
`final void publishProgress(Progress ... values)`. Un tel appel permet de mettre à jour la progression du traitement. Il se traduit en fait par un appel à la méthode :
`void publishProgressUpdate(Progress ... values)`. C'est dans cette méthode où l'on pourra, par exemple, modifier l'interface utilisateur, puisque cette méthode s'exécute dans le *thread* UI.

c) **`void onPostExecute(Result result)`** s'exécute dans le *thread* UI lorsque la méthode *doInBackground* est terminée.

En résumé, seule la méthode *doInBackground* ne s'exécute pas dans le *thread* UI.

Remarque : la notion de tâche asynchrone ajoute un niveau d'abstraction par rapport à celle de *thread*. C'est le *framework* qui réalise une partie du travail, à la place du programmeur. Il prépare la tâche, crée lui-même le *thread,* par exemple.

Interruption d'une tâche de fond

Il est possible d'interrompre une tâche préalablement créée en lui appliquant la méthode

```
final boolean cancel(boolean mayInterruptIfRunning)
```

Le booléen doit être égal à vrai pour interrompre la tâche. Cette méthode invoque :

```
protected void onCancelled(Result resultat)
```

que le programmeur peut avoir redéfinie ou pas dans son instanciation de la classe *AsyncTask*

Attention : une instance de *AsyncTask* ne peut être lancée qu'une seule fois. Si une nouvelle exécution est nécessaire, il faut créer une nouvelle instance.

Si l'activité est tuée, les tâches asynchrones qu'elle a lancées sont elles aussi détruites.

A quel endroit du projet définir la classe qui implémente *AsyncTask* ?

- Peut-on la définir à l'intérieur de la classe qui devra créer la tâche asynchrone ? donc en tant que classe interne ? C'est la solution la plus simple à coder (puisque chaque instance a une visibilité sur l'autre), mais pas la meilleure du point de vue efficacité mémoire. Le problème est que chaque fois que l'on crée une instance de la classe interne, celle-ci possède une référence à l'instance de la classe externe qui l'a créée. Et donc tant que la tâche asynchrone ne sera pas terminée, l'activité ne pourra pas être supprimée de la mémoire du terminal.

- On peut aussi définir la classe tâche asynchrone :
  - soit dans la classe C qui créera la tâche, mais pas comme une classe statique
  - soit dans un fichier séparé

Dans les deux cas, il sera en général nécessaire de stocker en tant qu'attribut (dans la classe C) l'instance de la classe asynchrone créée.

***Thread* et cycle de vie de l'activité**

Il faut se souvenir qu'une simple rotation de l'écran du terminal effectuée par l'utilisateur engendrera la destruction de l'activité et la création d'une nouvelle. Par défaut, les *threads* en arrière plan ne sont pas informés de ce changement au niveau de l'activité.

Si le programmeur ne fait rien, la tâche asynchrone située en arrière plan enverra ses résultats à l'ancienne activité, et la nouvelle ne sera pas informée, ce qui va bien sûr poser problème. On pourrait imaginer de coder dans la nouvelle activité, un nouveau lancement de la tâche de fond, mais ceci serait un gaspillage de ressources (par exemple, le téléchargement d'un fichier serait recommencé).

Une autre méthode pour résoudre ce problème est expliqué dans le dernier exercice du TP.

4) Conclusion

Tout bon programmeur doit veiller à éviter le blocage des applications qu'il développe. Ceci le conduira parfois à montrer à l'utilisateur une barre de progression, si l'application effectue en arrière plan, un traitement long, ou bien si l'application prend du temps au démarrage pour se charger, et faire des initialisations. Dans un autre contexte, pour programmer des jeux, le programmeur utilisera le plus souvent possible les *threads* dans lesquels les calculs sont faits.

Les notifications

I) Introduction

La barre de notification présente en haut de l'écran des terminaux Android a pour rôle d'informer l'utilisateur des événements importants : nouveaux messages, appel en absence, batterie en charge …

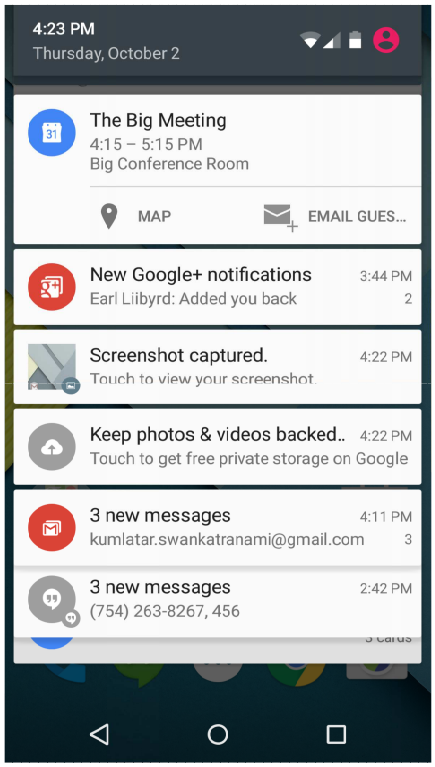
Souvent lorsque l'utilisateur clique sur une notification, une activité de l'application concernée doit se lancer, afin de lui permettre d'interagir. Par exemple, une notification indiquant la réception d'un message devrait rediriger vers l'activité permettant de lire et/ou de répondre au message.

Depuis la version *Lollipop*, le design des notifications respecte un schéma bien précis. Chaque rectangle affichant une notification comporte :

- ✓ Une icône : celle de l'application qui est l'origine de la notification ou une autre
- ✓ Le titre de la notification et sa description (par exemple : l'expéditeur du message et le début du texte de celui-ci)
- ✓ Le moment de la réception de la notification

Les notifications peuvent comporter également des boutons d'action pour lancer une activité précise. Par exemple, dans le cas de la réception d'un message, les boutons pourraient être « Répondre » et « Archiver ».

La copie d'écran ci-contre provient du site *Android Developer*.



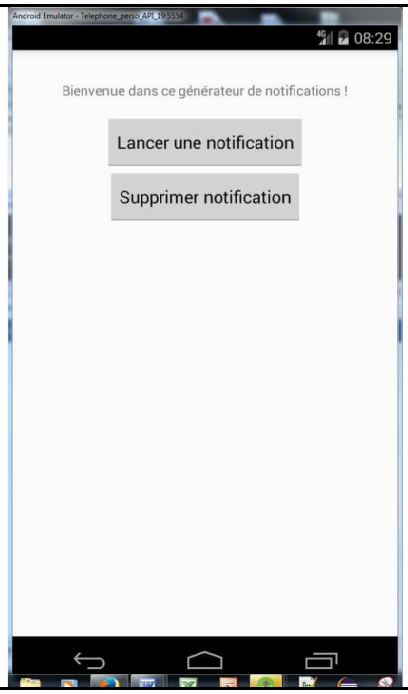
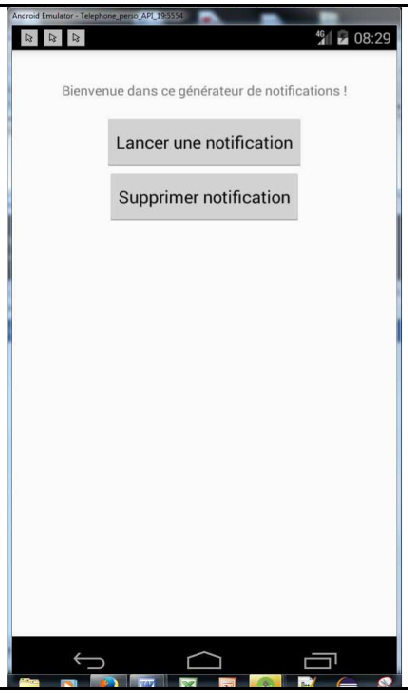
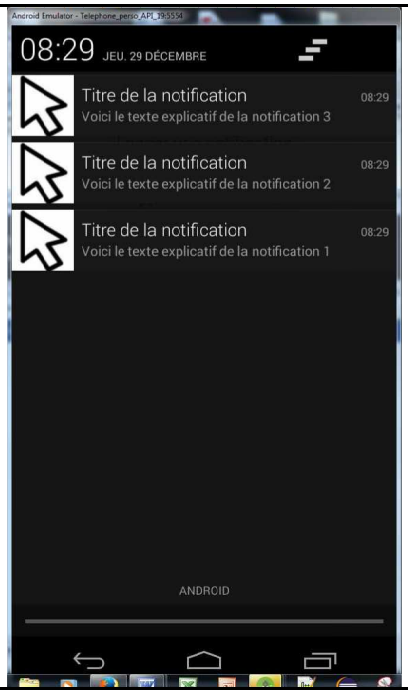
2) Principe et exemple

Exemple

On souhaite développer une application dotée de 2 boutons :

- ✓ L'un permettra d'envoyer une notification identifiée par un numéro unique
- ✓ L'autre permettra de supprimer toutes les notifications émises par l'application.

De plus, lorsque l'utilisateur cliquera sur une notification, on souhaite que l'activité principale de l'application soit lancée, même si l'utilisateur était en train d'utiliser une autre application.

		
Application à son lancement	Application après 3 clics sur « Lancer ». On voit apparaître les 3 icônes de notification (en haut à gauche du terminal)	Ici l'utilisateur a déroulé la ligne des notifications. Il peut alors cliquer sur l'une d'elles pour relancer l'activité

Gestionnaire de notifications

Pour faciliter la création des notifications, Android dispose d'une classe nommée ***NotificationManager***. Elle permet d'obtenir une instance d'un gestionnaire de notifications, en invoquant la méthode ***getSystemService***. En effet, la méthode ***getSystemService*** permet d'obtenir les différents gestionnaires proposés par Android, en précisant en argument le nom du service souhaité.

Plus précisément, pour obtenir un gestionnaire de notifications, on déclarera un attribut de ce type :

```
private NotificationManager gestionnaireNotification;
```

et, pour obtenir ensuite le gestionnaire, on écrira :

```
gestionnaireNotification =
        (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
```

Préparation d'une notification

Précisons d'abord que le type d'une notification est ***Notification***. Tout comme pour les boîtes de dialogue, par exemple, Android fournit une classe pour faciliter la construction des notifications. Il s'agit de la classe ***Notification.Builder***.

Par exemple, pour préparer une notification, on pourra écrire :

```
Notification note = new Notification.Builder(this)
        .setContentTitle("Titre de la notification")
        .setContentText("Texte de la notification")
        .setContentIntent(intention)
        .setVibrate(PATTERN_VIBRATION)
        .setSmallIcon(R.drawable.iconefleche)
        .build();
```

- ✓ Le constructeur de la classe ***Builder*** attend en paramètre le contexte d'exécution, donc par exemple celui de l'activité courante, c'est-à-dire *this*.
- ✓ Le titre et le texte de la notification sont renseignés en invoquant respectivement ***setContentTitle*** et ***setContentText.***
- ✓ La notification doit posséder une icône. Celle-ci est donnée en paramètre de la méthode ***setSmallIcon***. Il s'agit d'un fichier ressource de l'application.

Les informations décrites ci-dessus sont les seules indispensables pour préparer la notification. Toutefois, en général, on fournit des informations supplémentaires. Ici on a demandé à ce qu'une vibration soit émise à la réception de la notification, avec un appel à ***setVibrate***, et on a précisé aussi quelle intention doit être émise lorsque l'utilisateur cliquera sur la notification. Cette précision s'effectue par un appel à ***setContentIntent***.

Envoi d'une notification

L'envoi de la notification s'effectue par un appel à la méthode ***notify*** de la classe ***NotificationManager***. Par exemple :

```
gestionnaireNotification.notify(IDENT_NOTIFICATION + compteur, note);
```

- ✓ Le premier paramètre est l'identifiant de la notification de type entier. Si celui-ci est identique à celui d'une notification de la même application, et si cette notification est actuellement affichée, cette dernière sera remplacée par la nouvelle notification.
- ✓ Le deuxième argument est la notification à afficher.

Gestion du clic sur la notification

On souhaite que lorsque l'utilisateur cliquera sur la notification, l'activité qui a émise la notification soit relancée, et ce même si entre temps l'utilisateur l'a quittée pour utiliser une autre application. Pour ce faire, il faut d'abord préparer une intention de type ***PendingIntent***.

Une intention de type ***PendingIntent*** est une intention particulière dans le sens où elle décrit une action à exécuter, mais pas nécessairement immédiatement. En fait, c'est l'un des gestionnaires du système Android qui prendra l'initiative de traiter l'intention au moment adéquat. Le gestionnaire peut être celui des notifications, ou bien celui des alarmes, par exemple.

On peut construire une intention de type ***PendingIntent*** en effectuant un appel à la méthode ***getActivity*** de la classe ***PendingIntent***. Dans ce cas, l'intention pointera sur une activité à lancer lors du déclenchement de l'intention.

La méthode ***getActivity*** a le profil suivant :

PendingIntent getActivity (Context context, int requestCode, Intent intent, int flags)

- ✓ Le contexte est le contexte dans lequel l'intention décrite doit lancer l'activité
- ✓ Le code de requête est un code pour identifier la requête
- ✓ L'intention est celle qui permettra de lancer l'activité
- ✓ Le paramètre *flags* est destiné à apporter des informations complémentaires sur l'intention. En général, sa valeur est 0.

Dans l'exemple, la méthode ***getActivity*** a été invoquée de la manière suivante :

```
PendingIntent intention =
               PendingIntent.getActivity(this,
                                         0,
                                         new Intent(this, MainActivity.class),
                                         0);
```

Ensuite cette intention doit être fournie à la notification avec un appel à ***setContentIntent***. D'où dans l'exemple, la ligne :

```
            .setContentIntent(intention)
```

Supprimer une notification

Pour supprimer une notification, il faut invoquer la méthode ***cancel*** sur l'instance gestionnaire de notifications. On donne en paramètre de la méthode *cancel* l'identifiant de la notification à supprimer (celui qui a été donné lors de l'appel à ***notify***). Par exemple :

```
gestionnaireNotification.cancel(IDENT_NOTIFICATION + i);
```

Code complet de l'exemple

```
/*
 * Exemple pour illustrer la notion de notification
 * MainActivity.java                                          12/16
 */
package com.reactivite.exercice.android.exemplenotification;

import android.app.Notification;
import android.app.NotificationManager;
import android.app.PendingIntent;
import android.content.Intent;
import android.app.Activity;
import android.os.Bundle;
import android.view.View;

/**
 * Activité principale de l'application permettant d'illustrer le concept
 * de notification.
 * L'activité affiche 2 boutons : l'un permet d'envoyer une notification,
 * (chaque notification envoyée est identifiée par un numéro)
 * l'autre permet de supprimer toutes les notifications.
 * @author Servières
 * @version 1.0
 */
public class MainActivity extends Activity {


    /** Identifiant pour la notification */
    private static final int IDENT_NOTIFICATION = 1000;

    /** Pattern pour décrire la vibration lors de l'émission de la notification */
    private static final long[] PATTERN_VIBRATION =
                                    new long[] {500L, 200L, 200L, 500L};

    /** Gestionnaire de notifications */
    private NotificationManager gestionnaireNotification;

    /** Compteur permettant d'identifier chacune des notifications émises */
    private int compteur;


    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        compteur = 0;

        // on récupère une référence sur le gestionnaire de notifications
        gestionnaireNotification =
                (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
    }


    /**
     * Méthode exécutée lors du clic sur le bouton "lancer notification"
     * @param bouton  bouton sur lequel le clic a été effectué
     */
    public void clicLancer(View bouton) {

        compteur++;      // on compte une notification de plus

        /*
```

```
         * préparation d'une intention de type PendingIntent.
         * C'est l'intention qui sera lancée si l'utilisateur clique sur
         * la notification
         */
        PendingIntent intention = PendingIntent.getActivity(
                this,
                0,
                // lors du clic : c'est cette activité qui doit être lancée
                new Intent(this, MainActivity.class),
                0);

        // préparation de la notification
        Notification note = new Notification.Builder(this)
                .setContentTitle(getResources().getString(R.string.titre_notification))
                .setContentText(getResources().getString(R.string.texte_notification)
                                + " " + compteur)
                .setContentIntent(intention)
                .setVibrate(PATTERN_VIBRATION)
                .setSmallIcon(R.drawable.iconefleche)
                .build();

        /*
         * pour préciser que la notification doit être annulée
         * lors du clic de l'utilisateur
         */
        note.flags |= Notification.FLAG_AUTO_CANCEL;

        // on envoie la notification au gestionnaire de notifications
        gestionnaireNotification.notify(IDENT_NOTIFICATION + compteur, note);
    }

    /**
     * Méthode exécutée lors du clic sur le bouton "Supprimer"
     * @param bouton  bouton sur lequel le clic a été effectué
     */
    public void clicSupprimer(View bouton) {

        // toutes les notifications sont supprimées
        for (int i = 1; i <= compteur; i++) {
            gestionnaireNotification.cancel(IDENT_NOTIFICATION + i);
        }
    }
}
```

Fournisseurs de contenu – Content Provider

I) Introduction

Une base de données permet de stocker des données auxquelles seule l'application créatrice pourra accéder. Si on souhaite exposer les données d'une application à une autre application, il faut utiliser la notion de fournisseur de contenu d'Android (ou *content provider*). Par exemple, on peut souhaiter que la base de données des clients créés et gérés par une première application, soit accessible à une deuxième application.

D'une manière générale, un fournisseur de contenu procure un moyen homogène pour accéder aux données et les partager entre applications Android. On pourra via un fournisseur de contenu accéder à des données gérées nativement telles que les contacts, les signets, les photos, les vidéos, les musiques ….

Un fournisseur de contenu masque la façon dont les données sont effectivement stockées. Seul le concepteur du fournisseur de contenu connaît la structuration interne des données exposées (une base de données, un ou plusieurs fichiers …).

Pour accéder à un fournisseur de contenu, on doit utiliser la classe ***ContentResolver***. Une instance de celle-ci est obtenue en invoquant la méthode ***getContentResolver*** de la classe *Activity* :

```
ContentResolver accesFournisseur = getContentResolver();
```

Ensuite pour accéder à un fournisseur de contenu particulier, il faut utiliser son URI. L'URI identifie de manière unique un fournisseur de contenu.

II) Notion d'URI d'un fournisseur de contenu

Chaque fournisseur de contenu est identifié par une URI unique. Bien sûr, les fournisseurs natifs tels que les signets, les contacts ont des URI prédéfinies au niveau système. Par contre, les autres fournisseurs de contenu auront une URI définie par le programmeur de l'application qui les a créés.

Exemple d'URI : `content://com.licencemms.application.provider/photos`

Une URI se compose de 3 parties :

- Le **protocole**, ici *content*
- L'**autorité** ou autrement dit le nom du fournisseur de contenu, ici :
  `com.licencemms.application.provider`
  Notons que ce nom devra être déclaré dans le fichier *manifeste* et que par convention il s'agit d'un dérivé du nom du package contenant l'application créatrice.
  L'autorité est donc l'identifiant du fournisseur de contenu. Il doit être unique.
- Le **chemin** est la dernière partie de l'URI. Il permet d'identifier le sous-ensemble des données auxquelles on souhaite accéder au sein du fournisseur de contenu, ici *photos*. *Photos* pourrait donc être l'une des tables présentes dans le fournisseur de contenu.
  Le chemin peut se terminer par l'identifiant de l'élément auquel on souhaite accéder. Par exemple :
  `content://com.licencemms.application.provider/photos/20`
  pour accéder à la photo ayant pour identifiant 20.

Notons qu'un unique fournisseur de contenu peut gérer plusieurs types de données : des photos, des étudiants, des modules …

Exemples d'URI pour accéder aux fournisseurs de contenu du système :

`content://com.android.contacts/`	la totalité du fournisseur des contacts
`content://com.android.contacts/contact`	pour accéder uniquement à la table contact
`content://com.android.contacts/contact/20`	pour accéder uniquement au contact ayant l'identifiant 20

`ContactsContract.Contacts.CONTENT_URI`	URI de base pour les contacts
`ContactsContract.Contacts.CONTENT_LOOKUP_URI`	URI pour trouver un contact connaissant son identifiant
`Browser.BOOKMARKS_URI`	URI pour accéder aux signets du Navigateur internet

En fait, un fournisseur de contenu expose un ensemble d'URI (et pas une seule, en général) capable de donner accès à la totalité ou à une partie des données qu'il contient.

III) Accéder à un fournisseur de contenu

Le modèle de données

Un fournisseur de contenu présente les données qu'il contient sous la forme d'un tableau constitué de plusieurs colonnes (analogie avec une table d'une base de données). La première colonne est celle de l'identifiant numérique de la ligne de données (ou enregistrement). Elle se nomme _ID. Cet identifiant peut être utilisé pour accéder à un enregistrement précis via une URI. Les colonnes du tableau peuvent être de tout type.

On accède aux données du fournisseur via un curseur de type ***Cursor***. Il s'agit de la même classe que celle utilisée pour l'accès à une base de données.

Effectuer une requête

Pour accéder aux données d'un fournisseur de contenu, il faut d'abord se procurer une instance de type ***ContentResolver*** :

```
ContentResolver accesFournisseur = getContentResolver() ;
```

La classe ***ContentResolver*** contient les méthodes de requête suivantes :

query	`Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder, CancellationSignal cancellationSignal)`
insert	`Uri insert(Uri url, ContentValues values)`
update	`int update(Uri uri, ContentValues values, String where, String[] selectionArgs)`
delete	`int delete(Uri url, String where, String[] selectionArgs)`

Les paramètres de ces méthodes sont analogues à ceux des méthodes équivalentes de la classe *SQLiteDatabase*. Seul le premier paramètre diffère : on indiquera l'URI du fournisseur de contenu au lieu de l'instance de type base de données.

La méthode ***query*** retourne une instance de type *Cursor*. Tout comme avec les curseurs des bases de données, il est conseillé de prendre en compte le cycle de vie de l'activité, et donc de :

- ✓ désactiver le curseur lorsque l'activité qui l'utilise est stoppée
- ✓ détruire le curseur en même temps que l'activité
- ✓ réinterroger le fournisseur de contenu lorsque l'activité revient au premier plan, et ce pour intégrer les changements survenus sur les données

La classe *Activity* contient une méthode permettant d'automatiser toutes ces opérations :

```
Cursor managedQuery(Uri uri, String[] projection, String selection,
                    String[] selectionArgs, String sortOrder).
```

Les fournisseurs de contenu natifs

Les fournisseurs de contenu natifs sont décrits dans *android.provider*. Chacune des classes possède une constante CONTENT_URI qui est en fait l'URI pour accéder au fournisseur qu'elle représente

Fournisseur de contenu	**Nom de la classe**	**Description**
Contacts	`ContactsContract`	Gestion des contacts

Magasin multimédia	`MediaStore`
	`MediaStore.Audio`
	`MediaStore.Audio.Artists`
	`MediaStore.Video`
	`MediaStore.Image`
		Ensemble des fichiers multimédia du terminal

Navigateur	`Browser.SearchColumns` `Browser.BookmarkColumns`	Pour accéder à l'historique, aux marque-pages ou aux recherches
Appels	`CallLog.Calls`	Pour accéder à l'historique des appels entrants, sortants et manqués
Paramètres		Pour accéder aux paramètres système : sonnerie, … ou aux préférences
Dictionnaire	`UserDictionary.Words`	Les mots que connaît le dictionnaire utilisateur
Calendrier	`Calendar`	Gestion de l'agenda

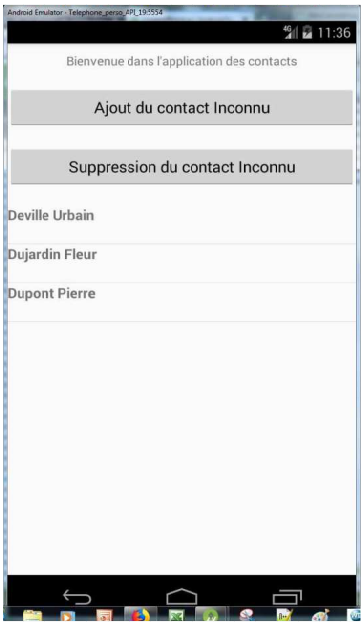
Le fournisseur de contenu *ContactsContract*

Ce fournisseur offre une base de données extensible contenant des informations relatives aux contacts. Le programmeur peut s'il le souhaite accéder aux contacts, mais aussi étendre l'ensemble des données stockées pour chaque contact ou même créer un fournisseur de contenu alternatif à celui qui est utilisé nativement.

Le fournisseur ContactsContract n'utilise pas une seule table pour stocker le détail des contacts. Il s'appuie sur un modèle 3-tiers comportant les sous-classes suivantes :

- ✓ ***Data***. Dans la table sous-jacente, chaque ligne définit un ensemble de données personnelles (numéros de téléphoe, adresses e-mail …)
- ✓ ***RawContacts***. Chaque ligne de la table sous-jacente définit un compte auquel un ensemble de valeurs de type *Data* est associé. Les comptes correspondent à différents comptes sources de contacts comme gmail, Facebook par exemple.
- ✓ ***Contracts***. Cette table agrège les lignes de *RawContacts* qui décrivent la même personne.

Exemple : accès aux contacts



A son lancement, l'activité affiche les contacts stockés sur le terminal.

Si l'utilisateur clique sur le bouton « Ajout », un contact nommé « inconnu » est ajouté au fournisseur de contenu qui gère les contacts. La vue de l'activité est actualisée en conséquence. On peut vérifier aussi que ce contact est bien ajouté aux contacts stockés dans le terminal en lançant l'application native qui gère les contacts.

Si l'utilisateur clique sur le bouton « Suppression », tous les contacts nommés « inconnu » sont supprimés du fournisseur de contenu. La vue de l'activité est actualisée. Là aussi on vérifie que la suppression est bien effective en lançant l'application native de gestion des contacts.

Nous devons ajouter deux permissions dans le fichier *AndroidManifest.xml* :

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.cours.exemple.foournisseur.application.accescontact">

    <uses-permission android:name="android.permission.READ_CONTACTS"/>
     <uses-permission android:name="android.permission.WRITE_CONTACTS"/>

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
```

```
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

     </application>
</manifest>
```

Le code de l'activité est le suivant :

```
/*
 * Exemple d'utilisation d'un fournisseur de contenu natif : gestionnaire des contacts   11/17
 * MainActivity.java
 */
package com.cours.exemple.foournisseur.application.accescontact;

. . .
/**
 * Cette activité affiche la liste des contacts présents dans le terminal.
 * Les contacts sont obtenus via le fournisseur de contenu natif qui
 * expose l'ensemble des contacts présents dans le terminal.
 * A chaque clic sur le bouton "Ajout", un contact nommé "inconnu"
 * est ajouté au fournisseur des contacts.
 * Au contraire, un clic sur le bouon "Suppression" supprime tous les
 * contacts ayant pour nom "Inconnu".
 * @author LP MMS
 * @version 1.0
 */
public class MainActivity extends ListActivity {

    /**
     * Constante définissant l'ordre d'affichage des contacts :
     * ici par ordre alphabétique. De plus, les lettres accentuées
     * (ou autres lettres particulières) seront placées à la suite de la
     * lettre correspondante non accentuée
     */
    private static final String ORDRE_DES_CONTACTS =
            ContactsContract.Contacts.DISPLAY_NAME + " COLLATE LOCALIZED ASC";

    /** Curseur sur l'ensemble des contacts */
    private Cursor curseurContact;

    /**
     * Adaptateur : il sert d'intermédiaire entre le curseur et la vue qui
     *  affiche les contacts. C'est curseurContact qui lui est associé
     */
    private CursorAdapter adaptateur;


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

        /*
         * on place dans le curseur tous les contacts
         * Le 1er argumant est l'Uri des contacts gérés nativement
         */
        curseurContact =
                getContentResolver().query(ContactsContract.Contacts.CONTENT_URI,
                                           null, null, null,
                                           ORDRE_DES_CONTACTS);

        /*
         * on définit un adaptateur pour le curseur,
```

```
         * et on l'associe à la liste affichée
         */
        adaptateur =
                new SimpleCursorAdapter(this,
                                        android.R.layout.two_line_list_item,
                                        curseurContact,
                                        new String[] {ContactsContract.Data.DISPLAY_NAME},
                                         new int[] {android.R.id.text1}, 0);
        setListAdapter(adaptateur);
    }


    /**
     * Invoquée lors du clic sur le bouton ajoutContact
     * Ajoute le contact "Inconnu" à la liste des contacts
     * @param bouton  bouton à l'origine du clic
     */
    public void ajouterContact(View bouton) {
        ContentValues nouveau = new ContentValues();

        // ajout à la table RawContacts
        nouveau.put(ContactsContract.RawContacts.ACCOUNT_NAME, "Inconnu");
        nouveau.put(ContactsContract.RawContacts.ACCOUNT_TYPE, "essai.com");
        Uri rawContactUri = getContentResolver().
                insert(ContactsContract.RawContacts.CONTENT_URI, nouveau);
        long rawContactId = ContentUris.parseId(rawContactUri);

        // ajout à la table Data
        nouveau.clear();
        nouveau.put(ContactsContract.Contacts.Data.RAW_CONTACT_ID, rawContactId);
        nouveau.put(ContactsContract.Contacts.Data.MIMETYPE,
                ContactsContract.CommonDataKinds.StructuredName.CONTENT_ITEM_TYPE);
        nouveau.put(ContactsContract.CommonDataKinds.StructuredName.DISPLAY_NAME, "Inconnu");
        getContentResolver().insert(ContactsContract.Data.CONTENT_URI, nouveau);

        // on actualise la vue
        curseurContact =
                getContentResolver().query(ContactsContract.Contacts.CONTENT_URI,
                        null, null, null,
                        ORDRE_DES_CONTACTS);
        adaptateur.changeCursor(curseurContact);
    }

    /**
     * Invoquée lors du clic sur le bouton supprimerContact
     * Supprime tous les contacts ayant pour nom "Inconnu"
     * @param bouton  bouton à l'origine du clic
     */
    public void supprimerContact(View bouton) {

        // suppression de la table Data
        getContentResolver().delete(ContactsContract.Data.CONTENT_URI,
                ContactsContract.CommonDataKinds.StructuredName.DISPLAY_NAME + " = ?",
                new String[]{"Inconnu"});

        // suppression de la table RawContacts
        getContentResolver().delete(ContactsContract.RawContacts.CONTENT_URI,
                ContactsContract.RawContacts.ACCOUNT_NAME + " = ? ",
                new String[]{"Inconnu"});

        // on actualise la vue
        curseurContact =
                getContentResolver().query(ContactsContract.Contacts.CONTENT_URI,
                        null, null, null,
                        ORDRE_DES_CONTACTS);
        adaptateur.changeCursor(curseurContact);
    }

}
```

Optimisation temps d'exécution

Les requêtes sur les bases de données et donc sur les fournisseurs de contenu sont coûteuses en temps d'exécution. Or pas défaut l'instance ***ContentResolver*** exécutera les requêtes dans le *thread* principal de l'application. Pour garantir la réactivité de l'application, il faut que ces requêtes soient exécutées de façon asynchrone.

Pour faciliter l'écriture du code, le programmeur peut utiliser la notion de ***chargeur de curseur*** (ajoutée à l'API 11). Cette notion permet aux curseurs de se synchroniser correctement avec le *thread* de l'interface utilisateur tout en assurant que les requêtes soient exécutées en arrière-plan.

Les chargeurs sont disponibles via la classe ***LoaderManager***. Ils sont conçus pour charger les données de façon asynchrone et pour surveiller les modifications de la source de données sous-jacente. D'une manière générale, ils sont implémentés pour charger n'importe quelle sorte de données à partir de n'importe quelle source.

Toutefois la classe ***CursorLoader*** est une spécialisation de chargeur pour les données de type ***Cursor***. Elle permet d'effectuer des requêtes asynchrones sur les fournisseurs de contenu.

Un chargeur de curseur gère toutes les tâches nécessaires à l'utilisation d'un curseur dans une activité. Celles-ci incluent notamment la gestion du cycle de vie des curseurs pour garantir qu'ils seront fermés lorsque l'activité sera terminée. Ils surveillent également les modifications du contenu de la source de données qui leur ait associé.

La tâche du programmeur consiste à implémenter 3 méthodes de *callback* gérant un chargeur de curseur. Ces méthodes sont présentes dans l'interface ***LoaderManager.LoadersCallbacks***. Cette interface est générique et paramétrée par le type des données à charger. Notre classe activité devra donc implémenter l'interface ***LoaderManager.LoadersCallbacks<Cursor>.***

L'interface comporte 3 méthodes :

1) onCreateLoader

Appelée lorsque le chargeur est initialisé. Un chargeur est créé et renvoyé. Il est de type ***CursorLoader***. Le constructeur de cette classe a les mêmes paramètres que la méthode *query* de la classe *ContentResolver* puisque c'est maintenant le chargeur qui se chargera de réaliser la requête (de manière asynchrone)

2) onLoadFinished

Appelée lorsque le *loaderManager* a terminé la requête asynchrone. En paramètre de la méthode, on trouve le curseur résultat de la requête. Il faudra donc dans cette méthode mettre à jour un adaptateur, par exemple.

3) onLoaderReset

Appelée lorsque le *loaderManager* réinitialise le chargeur de curseur. Dans cette méthode, on doit libérer toutes les références aux données renvoyées par la requête et réinitialiser l'adaptateur et/ou l'interface. Notons que le curseur est fermé automatiquement par le *loaderManager*. Le programmeur ne doit donc pas invoquer la méthode *close* sur le curseur.

Attention : les méthodes ***onLoadFinish*** et ***onLoaderReset*** ne s'exécutent pas dans le *thread* UI. Il sera donc impossible d'accéder directement à un *widget* dans le corps de ces méthodes. Pour ce faire, il faut utiliser la méthode *post*.

La section suivante illustre l'utilisation d'un chargeur de curseur.

IV) Créer un fournisseur de contenu

Pour créer un nouveau fournisseur de contenu, il faut définir une nouvelle classe qui hérite de la classe abstraite ***ContentProvider***.

Les méthodes de cette classe ont des signatures proches de celles de la classe *SQLiteDatabase*. La principale différence réside en la présence de l'URI du fournisseur parmi les paramètres. Ces méthodes sont les suivantes :

```
public boolean onCreate ()
public String getType(Uri uri)
public Cursor query(Uri uri, String[] projection, String selection,
                    String[] selectionArgs, String sortOrder)
public Uri insert(Uri uri, ContentValues values)
public int update(Uri uri, ContentValues values, String where, String[] whereArgs)
public int delete(Uri uri, String where, String[] whereArgs)
```

La méthode ***onCreate*** permet d'initialiser le fournisseur de contenu. La méthode ***getType*** fournit le type MIME (Multi-purpose Internet Mail Extension) des données retournées. Cette information est utilisée par Android pour déterminer quelle application lancer en réponse à une intention.

Les 4 méthodes ***query***, ***update***, ***delete*** et ***insert***, correspondent aux 4 demandes auxquelles le fournisseur de contenu peut répondre. Certaines d'entre elles peuvent ne pas être implémentées si on estime que le fournisseur ne doit pas répondre à cette demande. Dans le cas où ces méthodes sont implémentées, elles obéissent au schéma suivant :

- ✓ Déterminer l'action à réaliser à partir du paramètre de type URI
- ✓ Si l'Uri contient un identifiant, il faut l'extraire de l'Uri
- ✓ Exécuter l'action demandée

Exemple

On souhaite définir une base de données formée d'une table contenant la description des planètes du système solaire. Chaque planète sera décrite pas son nom, la nature de son sol (rocheux ou gazeux), son rayon et la distance qui la sépare du Soleil. Cette base de données doit être accessible à plusieurs applications via un fournisseur de contenu.

Nous allons répartir le code dans 4 fichiers :

- ✓ Une classe Java décrivant les instances de type ***Planete***
- ✓ Une classe ***HelperBDPlanete*** pour gérer la connexion à la base de données
- ✓ Une classe ***PlaneteDAO*** pour faciliter l'accès à la base de données
- ✓ La classe fournisseur de contenu nommée ***FournisseurPlanete***

```java
/*
 * Représentation d'une planète                    12/17
 * Planete.java
 */
package com.applicationmms.persistance.fournisseur.exemple.fournisseurplanete;

/**
 * Cette classe représente une planète du système solaire
 * Les informations associées à une planète sont : le nom, la nature (rocheuse ou gazeuse),
 * le rayon, et la distance au soleil
 * @author LP MMS
 * @version 1.0
 */

public class Planete {

    /** Identifiant de la planète dans la table qui la contient */
    private int identifiant;

    /** Nom de la planète */
    private String nom;

    /** Vrai ssi la planète est rocheuse. Sinon elle est gazeuse */
    private boolean rocheuse;

    /** Rayon de la planète en kilomètres */
    private int rayon;

    /** Distance au Soleil en millions de km */
    private int distanceSoleil;

    /**
     * Constructeur avec initialisations par défaut
     */
    public Planete() {
        identifiant = 0;
        nom = "";
        rocheuse = false;
        rayon = 0;
        distanceSoleil = 0;
    }

    /**
     * Constructeur avec données initiales en argument
     * @param leNom          nom de la planète
     * @param avecRoche      nature du sol
     * @param leRayon        rayon de la planète en km
     * @param laDistance     distance au soleil en km
     */
    public Planete(String leNom, boolean avecRoche, int leRayon, int laDistance) {
        this.nom = leNom;
        this.rocheuse = avecRoche;
        this.rayon = leRayon;
        this.distanceSoleil = laDistance;
        this.identifiant = 0;
    }

    /** Accesseur sur l'identifiant
     * @return  l'identifiant de la planète
     */
    public int getIdentifiant() {
        return this.identifiant;
    }

    /**
     * Modificateur de l'identifiant
     * @param identifiant  nouvel identifiant
     */
```

```
    public void setIdentifiant(int identifiant) {
        this.identifiant = identifiant;
    }

    /**
     * Acesseur sur le nom
     * @return  le nom de la planète
     */
    public String getNom() {
        return this.nom;
    }

    /**
     * Modificateur du nom
     * @param  nouveauNom  nouveau nom de la planète
     */
    public void setNom(String nouveauNom) {
        this.nom = nouveauNom;
    }


    /**
     * Acesseur sur la nature du sol
     * @return  un booléen égal à vrai ssi la planète est rocheuse
     */
    public boolean getRocheuse() {
        return this.rocheuse;
    }

    /**
     * Modificateur de la nature du sol
     * @param  estRocheuse vrai ssi la planète est rocheuse
     */
    public void setRocheuse(boolean estRocheuse) {
        this.rocheuse = estRocheuse;
    }

    /**
     * Acesseur sur le rayon
     * @return  un entier égal au rayon de la planète
     */
    public int getRayon() {
        return this.rayon;
    }

    /**
     * Modificateur du rayon
     * @param  nouveauRayon nouvelle valeur du rayon
     */
    public void setRayon(int nouveauRayon) {
        this.rayon = nouveauRayon;
    }

    /**
     * Acesseur sur la distance au soleil
     * @return  un entier égal la distance au soleil
     */
    public int getDistanceSoleil() {
        return this.distanceSoleil;
    }

    /**
     * Modificateur de la distance au soleil
     * @param  nouvelleDistance nouvelle valeur pour la distance
     */
    public void setDistanceSoleil(int nouvelleDistance) {
        this.distanceSoleil = nouvelleDistance;
    }
}
```

Planete.java

```java
/*
 * Classe permettant de gérer la base de données qui contient les planètes
 * HelperBDPlanete.java                                                 12/17
 */
package com.applicationmms.persistance.fournisseur.exemple.fournisseurplanete;

import android.content.ContentValues;
import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;

/**
 * Cette classe permet de gérer la base de données qui contient la description
 * des planètes. Elle contient :
 *   - un constructeur
 *   - une méthode permettant de créer la base, et de l'initialiser avec quelques planètes
 *   - une méthode qui sera invoquée si on change de version pour la base de données
 * @author LP MMS
 * @version 1.0
 */
public class HelperBDPlanete extends SQLiteOpenHelper {

    /** Nom du champ correspondant à l'identifiant de la planète, la clé */
    public static final String CLE_PLANETE = "_id";

    /** Nom du champ correspondant au nom de la planète */
    public static final String NOM_PLANETE= "nom";

    /** Nom du champ correspondant à la nature du sol */
    public static final String SOL_PLANETE = "sol";

    /** Nom du champ correspondant au rayon en km*/
    public static final String RAYON_PLANETE = "rayon";

    /** Nom du champ correspondant à la distance au soleil */
    public static final String DISTANCE_PLANETE = "distance";


    /** Nom de la table qui contiendra la description des planètes du système solaire */
    public static final String NOM_TABLE_PLANETE = "PlaneteSystemeSolaire";



    /** Requete pour la création de la table */
    private static final String CREATION_TABLE_PLANETE =
            "CREATE TABLE " + NOM_TABLE_PLANETE  + " ( "
                    + CLE_PLANETE + " INTEGER PRIMARY KEY AUTOINCREMENT, "
                    + NOM_PLANETE + " TEXT, "
                    + SOL_PLANETE + " INTEGER, "
                    + RAYON_PLANETE + " INTEGER CHECK (" + RAYON_PLANETE  + "  > 0),"
                    + DISTANCE_PLANETE + " INTEGER CHECK (" + RAYON_PLANETE  + "  > 0)); ";

    /** Requete pour supprimer la table */
    private static final String SUPPRIMER_TABLE_PLANETE =
            "DROP TABLE IF EXISTS " + NOM_TABLE_PLANETE + " ;" ;


    /**
     * Constructeur de la classe
     * @param contexte
     * @param nom
     * @param fabrique
     * @param version
     */
```

```
    public HelperBDPlanete(Context contexte, String nom, SQLiteDatabase.CursorFactory fabrique,
                           int version) {
        super(contexte, nom, fabrique, version);
    }


    @Override
    public void onCreate(SQLiteDatabase bd) {
        bd.execSQL(CREATION_TABLE_PLANETE);       // exécution de la requête pour créer la base

        // objet pour préparer chacun des enregistrements à ajouter à la base
        ContentValues enregistrement = new ContentValues();

      /*
       *  on prépare un enregistrement, en donnant la valeur de chacune des colonnes.
       *  Cet enregistrement est ensuite inséré dans la base
       */
        enregistrement.put(NOM_PLANETE, "Mercure");
        enregistrement.put(SOL_PLANETE, 1);
        enregistrement.put(RAYON_PLANETE, 2439);
        enregistrement.put(DISTANCE_PLANETE, 58);
        bd.insert(NOM_TABLE_PLANETE, NOM_PLANETE, enregistrement);

        enregistrement.put(NOM_PLANETE, "Venus");
        enregistrement.put(SOL_PLANETE, 1);
        enregistrement.put(RAYON_PLANETE, 6051);
        enregistrement.put(DISTANCE_PLANETE, 108);
        bd.insert(NOM_TABLE_PLANETE, NOM_PLANETE, enregistrement);

        enregistrement.put(NOM_PLANETE, "Terre");
        enregistrement.put(SOL_PLANETE, 1);
        enregistrement.put(RAYON_PLANETE, 6378);
        enregistrement.put(DISTANCE_PLANETE, 150);
        bd.insert(NOM_TABLE_PLANETE, NOM_PLANETE, enregistrement);

        enregistrement.put(NOM_PLANETE, "Mars");
        enregistrement.put(SOL_PLANETE, 1);
        enregistrement.put(RAYON_PLANETE, 3393);
        enregistrement.put(DISTANCE_PLANETE, 228);
        bd.insert(NOM_TABLE_PLANETE, NOM_PLANETE, enregistrement);
    }

    @Override
    public void onUpgrade(SQLiteDatabase bd, int ancienneVersion, int nouvelleVersion) {
        bd.execSQL(SUPPRIMER_TABLE_PLANETE);      // on détruit et on recrée la base des pays
        onCreate(bd);
    }
}
```

HelperBDPlanete.java

```
/*
 * Classe d'aide d'accès à la table de la BD contenant les pays
 * PaysDAO.java                                                    12/17
 */
package com.applicationmms.persistance.fournisseur.exemple.fournisseurplanete;

import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;

import java.util.ArrayList;

/**
 * Classe d'aide d'accès à la table de la BD contenant les pays
```

```
 * @author LP MMS
 * @version 1.0
 */

public class PlaneteDAO {

    /** Numéro de version de la base de donnée */
    private static final int VERSION = 1;

    /** Nom de la base de données qui contiendra les planètes gérées */
    private static final String NOM_BD = "lesplanetes.db";

    /** Numéro de la colonne contenant l'identifiant d'une planète, la clé */
    public static final int COLONNE_NUM_CLE = 0;

    /** Numéro de la colonne contenant le nom de la planète */
    public static final int COLONNE_NUM_NOM = 1;

    /** Numéro de la colonne contenant la nature du sol de la planète */
    public static final int COLONNE_NUM_SOL = 2;

    /** Numéro de la colonne contenant le rayon de la planète */
    public static final int COLONNE_NUM_RAYON = 3;

    /** Numéro de la colonne contenant la distance de la planète */
    public static final int COLONNE_NUM_DISTANCE = 4;

    /** Gestionnaire permettant de créer la base de donnée */
    private HelperBDPlanete gestionnaireBase;

    /** Base de données contenant la description des pays */
    private SQLiteDatabase basePlanete;

    /** Requete pour sélectionner tous les enregistrements de la table */
    public static final String REQUETE_TOUT_SELECTIONNER =
            "select "
                    + HelperBDPlanete.CLE_PLANETE + ", "
                    + HelperBDPlanete.NOM_PLANETE + ", "
                    + HelperBDPlanete.SOL_PLANETE + ", "
                    + HelperBDPlanete.RAYON_PLANETE + " , "
                    + HelperBDPlanete.DISTANCE_PLANETE
                    + " from " + HelperBDPlanete.NOM_TABLE_PLANETE
                    + " order by " + HelperBDPlanete.NOM_PLANETE;

    /**
     * Constructeur permettant de créer l'objet gestionnaire de la BD
     * @param leContexte  contexte de l'activité créatrice
     */
    public PlaneteDAO(Context leContexte) {
        gestionnaireBase = new HelperBDPlanete(leContexte, NOM_BD, null, VERSION);
    }

    /**
     * Ouverture de la table des planetes
     */
    public void open() {
        basePlanete = gestionnaireBase.getWritableDatabase();
    }

    /**
     * Fermeture de la table des planetes
     */
    public void close() {
        gestionnaireBase.close();
        basePlanete.close();
    }

    /**
     * Renvoie d'un curseur sur la totalité des planetes
     * @return  un curseur référençant toutes les planètes de la base
```

```
     */
    public Cursor getCurseurToutesLesPlanetes() {
        return basePlanete.rawQuery(REQUETE_TOUT_SELECTIONNER, null );
    }


     /**
      *  Renvoie d'un curseur sur la totalité des planetes
      * @param ident identifiant de la planète à laquelle accéder
      * @return  un curseur référençant la planète identifiée
      */
    public Cursor getCurseurPlanete(int ident) {
        Cursor c = basePlanete.query(HelperBDPlanete.NOM_TABLE_PLANETE,
                new String[] {HelperBDPlanete.NOM_PLANETE,
                        HelperBDPlanete.SOL_PLANETE,
                        HelperBDPlanete.RAYON_PLANETE,
                        HelperBDPlanete.DISTANCE_PLANETE},
                HelperBDPlanete.CLE_PLANETE + " = ? ",
                new String[] {ident + ""}, null, null, null);
        return c;
    }

    /**
     * Renvoie toutes les planètes présentes dans la table, sous la forme d'une liste
     * @return  une ArrayList contenant toutes les planètes de la table
     */
    public ArrayList<Planete> getAllPays() {
        Cursor curseurTous = getCurseurToutesLesPlanetes();
        return cursorToListePlanete(curseurTous);
    }

    /*
      * Insère la planète argument dans la table des planètes
      * @param aInserer planète à insérer dans la table
      * @return l'identifiant de la planète ajoutée
     */
    public long insertPlanete(Planete aInserer) {
        ContentValues enregistrement = new ContentValues();
        enregistrement.put(HelperBDPlanete.NOM_PLANETE, aInserer.getNom());
        enregistrement.put(HelperBDPlanete.RAYON_PLANETE, aInserer.getRayon());
        enregistrement.put(HelperBDPlanete.DISTANCE_PLANETE, aInserer.getDistanceSoleil());
        enregistrement.put(HelperBDPlanete.SOL_PLANETE, aInserer.getRocheuse() ? 1 : 0);
        // insertion de l'enregistrement dans la base
        return basePlanete.insert(HelperBDPlanete.NOM_TABLE_PLANETE,
                HelperBDPlanete.NOM_PLANETE, enregistrement);
    }

    /*
      * Insère la planète argument dans la table des planètes
      * @param aInserer ContentValues contenant la planète à insérer
      * @return  l'identifiant de la planète ajoutée
     */
    public long insertPlanete(ContentValues aInserer) {

        // insertion de l'enregistrement dans la base
        return basePlanete.insert(HelperBDPlanete.NOM_TABLE_PLANETE,
                HelperBDPlanete.NOM_PLANETE, aInserer);
    }

    /**
     * Supprime une planète de la table des planètes
     * @param nomDeLaPlanete  nom de la planète à supprimer
     * @return un entier égal au nombre de lignes supprimées
     */
    public int deletePlanete(String nomDeLaPlanete) {
        return basePlanete.delete(HelperBDPlanete.NOM_TABLE_PLANETE,
                HelperBDPlanete.NOM_PLANETE + " = ?",
                new String[] { nomDeLaPlanete });
    }
```

```
    /**
     * Supprime une planète de la table des planètes
     * @param idDeLaPlanete  identifiant de la planète à supprimer
     * @return un entier égal au nombre de lignes supprimées
     */
    public int deletePlanete(int idDeLaPlanete) {
        return basePlanete.delete(HelperBDPlanete.NOM_TABLE_PLANETE,
                HelperBDPlanete.CLE_PLANETE + " = ?",
                new String[] { idDeLaPlanete+ "" });
    }

    /**
     * Supprime une planète de la table des planètes
     * @param selection   argument de sélection de la clause where
     * @param selectionArgs   valeurs des symboles ? présents dans selection
     * @return un entier égal au nombre de lignes supprimées
     */
    public int deletePlanete(String selection, String[] selectionArgs) {
        return basePlanete.delete(HelperBDPlanete.NOM_TABLE_PLANETE,
                selection, selectionArgs);

    }

    /**
     * Modifie une planète avec l'instance de type Planète argument.
     * L'enregistrement à modifier est recherché selon le nom de la planète argument
     * @param aModifier nouvelle valeur pour la planète à modifier
     * @return  un entier égal au nombre d'enregistrements modifiés
     */
    public int updatePlanete(Planete aModifier) {
        ContentValues nouveau  = new ContentValues();
        nouveau.put(HelperBDPlanete.NOM_PLANETE, aModifier.getNom());
        nouveau.put(HelperBDPlanete.RAYON_PLANETE, aModifier.getRayon());
        nouveau.put(HelperBDPlanete.DISTANCE_PLANETE, aModifier.getDistanceSoleil());
        nouveau.put(HelperBDPlanete.SOL_PLANETE, aModifier.getRocheuse() ? 1 : 0);

        return basePlanete.update(HelperBDPlanete.NOM_TABLE_PLANETE, nouveau,
                HelperBDPlanete.NOM_PLANETE + " = ?",
                new String[] {aModifier.getNom()} );
    }

    /**
     * Modifie une planète avec l'instance argument
     * L'enregistrement à modifier est recherché selon la sélection arguemnt
     * @param selection   argument de sélection de la clause where
     * @param selectionArgs   valeurs des symboles ? présents dans selection
     * @return  un entier égal au nombre d'enregistrements modifiés
     */
    public int updatePlanete(ContentValues nouvelle, String selection,
                             String[] selectionArgs) {
        return basePlanete.update(HelperBDPlanete.NOM_TABLE_PLANETE, nouvelle,
                selection, selectionArgs);
    }

    /**
     * Modifie une planète avec l'instance argument.
     * L'enregistrement à modifier est recherché à partir de son identifiant
     * @param identPlanete identifiant de la planète à modifier
     * @param nouvelle nouvelle valeur pour la planète à modifier
     * @return  un entier égal au nombre d'enregistrements modifiés
     */
    public int updatePlanete(int identPlanete, ContentValues nouvelle) {
        return basePlanete.update(HelperBDPlanete.NOM_TABLE_PLANETE, nouvelle,
                HelperBDPlanete.CLE_PLANETE + " = ?",
                new String[] {identPlanete + ""} );
    }


    /**
     * Recherche dans la table des planetes, la planète dont le nom est donné en argument
```

```
     * @param nomPlanete   nom de la planète à chercher dans la table des planètes
     * @return  l'instance Planete dont le nom est donné en argument (null si non trouvé)
     */
    public Planete getPlanete(String nomPlanete) {
        Cursor c = basePlanete.query(HelperBDPlanete.NOM_TABLE_PLANETE,
                new String[] {HelperBDPlanete.NOM_PLANETE,
                        HelperBDPlanete.SOL_PLANETE,
                        HelperBDPlanete.RAYON_PLANETE,
                        HelperBDPlanete.DISTANCE_PLANETE},
                HelperBDPlanete.NOM_PLANETE + " = ? ",
                new String[] {nomPlanete}, null, null, null);
        return cursorToPlanete(c);
    }

    /**
     * Transforme la ligne référencée par un curseur sur la table des planètes
     * en instance de type Planete
     * @param c   curseur qui référence une ligne dans la table des planètes
     * @return une instance de type Planete correspondant à celle référencé par le curseur
     *         (éventuellement null)
     */
    private Planete cursorToPlanete(Cursor c) {
        Planete aRenvoyer;
        if (c.getCount() == 0) {
            aRenvoyer = null;
        } else {
            c.moveToFirst();
            // on initialise l'instance Planète avec les valeurs des colonnes
            aRenvoyer = new Planete();
            aRenvoyer.setNom(c.getString(COLONNE_NUM_NOM - 1));
            aRenvoyer.setRayon(c.getInt(COLONNE_NUM_RAYON - 1));
            aRenvoyer.setDistanceSoleil(c.getInt(COLONNE_NUM_DISTANCE -1));
            aRenvoyer.setRocheuse(c.getInt(COLONNE_NUM_SOL -1) == 0 ? false : true);
        }
        c.close();
        return aRenvoyer;
    }

    /**
     * Transforme l'ensemble des planètes référencées par un curseur en une liste de planètes
     * @param c  un curseur sur un ensemble de planètes
     * @return une liste contenant les planètes référencées par le curseur
     */
    private ArrayList<Planete> cursorToListePlanete(Cursor c) {
        ArrayList<Planete> listePays = new ArrayList<>();
        if (c.getCount() != 0) {
            Planete aAjouter;
            c.moveToFirst();

            /* on parcourt toutes les lignes du curseur, et on ajoute
             * le pays référencé par le curseur à la liste
             */
            do {
                aAjouter = cursorToPlanete(c);
                listePays.add(aAjouter);
            } while (c.moveToNext());
        }
        c.close();
        return listePays;
    }

}
```

PlaneteDAO.java

```
/*
 *  Fournisseur de contenu pur accéder à la base de données des planètes      12/17
 *  FournisseurPlanete.java
 */
package com.applicationmms.persistance.fournisseur.exemple.fournisseurplanete;
```

```
import android.content.ContentProvider;
import android.content.ContentUris;
import android.content.ContentValues;
import android.content.UriMatcher;
import android.database.Cursor;
import android.net.Uri;


/**
 * Fournisseur de contenu donnant accès à la base de données des planètes
 * @author LP MMS
 * version 1.0
 */
public class FournisseurPlanete extends ContentProvider {

    /**
     * Autorité du fournisseur de contenu. Il s'agit d'une partie de l'URI qui
     * permet d'identifier de manière unique le fournisseur de contenu.
     * La valeur de l'autorité est liée au nom du package de l'application
     */
    private static final String AUTORITE =
            "com.applicationmms.persistance.fournisseur.exemple.fournisseurplanete";

    /** URI du fournisseur de contenu */
    private static final String URI_PLANETE = "content://" + AUTORITE + "/planetes";

    /** L'URI de type Uri doit être accessible à l'extérieur de la classe
     *  et définie dans une constante de nom CONTENT_URI
     */
    public static final Uri CONTENT_URI = Uri.parse(URI_PLANETE);

    /**
     * Constante pour identifier l'Uri qui permet d'accéder à toutes les planètes
     */
    private static final int TOUTES_PLANETES = 0;

    /**
     * Constante pour identifier l'Uri qui permet d'accéder à une planète
     * à partir de son identifiant
     */
    private  static final int PLANETE_UNIQUE = 1;

    /**
     * Instance qui permettra de vérifier qu'une Uri est syntaxiquement correcte
     */
    private static final UriMatcher uriMathcher;

    /**
     * Blocs d'instructions pour initialiser l'uriMatcher. 2 uris seront reconnues
     * par le fournisseur de contenu : celle pour accéder à toutes les planètes et
     * celle pour accéder à une planète précise connaissant son identifiant
     */
    static {
        uriMathcher = new UriMatcher(UriMatcher.NO_MATCH);
        uriMathcher.addURI(AUTORITE, "planetes", TOUTES_PLANETES);
        uriMathcher.addURI(AUTORITE, "planetes/#", PLANETE_UNIQUE);
    }

    /** DAO permettant de faciliter l'accès à la base de données des planètes */
    private PlaneteDAO bdPlanete;


    /** Constructeur pour initialisation par défaut */
    public FournisseurPlanete() {

    }
```

```
    @Override
    public boolean onCreate() {

        // ouverture de la base de données
        bdPlanete = new PlaneteDAO(getContext());
        bdPlanete.open();
        return true;
    }

    @Override
    public Uri insert(Uri uri, ContentValues valeur) {
        long id = bdPlanete.insertPlanete(valeur);   // insertion dans la bd
        if (id > 0) {

            // on ajoute à l'Uri renvoyée l'identifiant de la planète
            Uri aRenvoyer = ContentUris.withAppendedId(CONTENT_URI, id);

            //  informer les écouteurs qu'un changement a eu lieu
            getContext().getContentResolver().notifyChange(aRenvoyer, null);;
            return aRenvoyer;
        } else {
            return null;
        }
    }

    @Override
    public int update(Uri uri, ContentValues values, String selection,
                      String[] selectionArgs) {
        int retour = 0;    // nombre d'enregistrements modifiés
        switch (uriMathcher.match(uri)) {
            case PLANETE_UNIQUE :
                int identPlanete = 0;
                try {

                    // on récupère l'identifiant placé à la fin de l'uri
                    identPlanete = Integer.parseInt(uri.getPathSegments().get(1));
                    retour = bdPlanete.updatePlanete(identPlanete, values);
                } catch(Exception erreur) {
                    throw new IllegalArgumentException("URI non supportée : " + uri);
                }
                break;
            case TOUTES_PLANETES:
                retour = bdPlanete.updatePlanete(values, selection, selectionArgs);
                break;
            default :
                throw new IllegalArgumentException("URI non supportée : " + uri);
        }

        //  informer les écouteurs qu'un changement a eu lieu
        getContext().getContentResolver().notifyChange(uri, null);
        return retour;
    }

    @Override
    public int delete(Uri uri, String selection, String[] selectionArgs) {
        int retour = 0;     // nombre d'enregistrements modifiés
        switch (uriMathcher.match(uri)) {
            case PLANETE_UNIQUE :
                int identPlanete = 0;
                try {

                    // on récupère l'identifiant placé à la fin de l'uri
                    identPlanete = Integer.parseInt(uri.getPathSegments().get(1));
                    retour = bdPlanete.deletePlanete(identPlanete);
                } catch(Exception erreur) {
                    throw new IllegalArgumentException("URI non supportée : " + uri);
                }
                break;
```

```
            case TOUTES_PLANETES:
                retour = bdPlanete.deletePlanete(selection, selectionArgs);
                break;
            default :
                throw new IllegalArgumentException("URI non supportée : " + uri);
        }

        //  informer les écouteurs qu'un changement a eu lieu
        getContext().getContentResolver().notifyChange(uri, null);
        return retour;
    }


    @Override
    public Cursor query(Uri uri, String[] projection, String selection,
                        String[] selectionArgs, String sortOrder) {
        Cursor aRenvoyer = null;
        switch (uriMathcher.match(uri)) {
            case PLANETE_UNIQUE :
                int identPlanete = 0;
                try {

                    // on récupère l'identifiant placé à la fin de l'uri
                    identPlanete = Integer.parseInt(uri.getPathSegments().get(1));
                } catch(Exception erreur) {
                    throw new IllegalArgumentException("URI non supportée : " + uri);
                }

                // on récupère un curseur sur la planète ayant l'identifiant demandé
                aRenvoyer = bdPlanete.getCurseurPlanete(identPlanete);

                //  informer les écouteurs qu'un changement a eu lieu
                aRenvoyer.setNotificationUri(getContext().getContentResolver(), uri);
                break;
            case TOUTES_PLANETES:
                aRenvoyer = bdPlanete.getCurseurToutesLesPlanetes();

                //  informer les écouteurs qu'un changement a eu lieu
                aRenvoyer.setNotificationUri(getContext().getContentResolver(), uri);
                break;
            default :
                throw new IllegalArgumentException("URI non supportée : " + uri);
        }
        return aRenvoyer;
    }


    @Override
    public String getType(Uri uri) {

        /*
         * pour qu'Android puisse identifier la nature dse données retournées,
         * il faut renvoyer les types de contenus MIME
         */
        switch (uriMathcher.match(uri)) {
            case PLANETE_UNIQUE :
                return "vnd.android.cursor.collection/planete";
            case TOUTES_PLANETES:
                return "vnd.android.cursor.item/planete";
            default :
                throw new IllegalArgumentException("URI non supportée : " + uri);
        }
    }
}
```

FournisseurPlanete.java

Il est conseillé de définir dans cette classe des constantes statiques pour les noms de colonnes de la table sous-jacente par exemple, ou bien pour définir l'autorité du fournisseur de contenu. Ces valeurs seront par la suite utiles pour effectuer des opérations via le fournisseur de contenu.

Il est également conseillé pour un fournisseur de proposer 2 formes d'accès : une requête demandant le renvoie de toutes les lignes, et une autre demandant le renvoi d'une unique ligne précise repérée par son identifiant. Dans ce cas, la requête se termine par `/numéro_de_ligne`.

Un moyen simple de gérer ces 2 formes est d'utiliser la classe ***UriMatcher***. Elle permettra d'analyser des Uri et de déterminer leur forme. En fait, elle convertit les Uri en codes entiers. L'idée est d'associer à chaque forme d'Uri reconnue, une constante entière. Il sera facile ensuite de tester la valeur de cette constante.

Concrètement dans la classe du fournisseur de contenu, on définira un attribut statique et constant de type ***UriMatcher***. Il sera créé et initialisé dans un bloc d'instructions lui-même statique.

```
/**
 * Constante pour identifier l'Uri qui permet d'accéder à toutes les planètes
 */
private static final int TOUTES_PLANETES = 0;

/**
 * Constante pour identifier l'Uri qui permet d'accéder à une planète
 * à partir de son identifiant
 */
private  static final int PLANETE_UNIQUE = 1;

/**
 * Instance qui permettra de vérifier qu'une Uri est syntaxiquement correcte
 */
private static final UriMatcher uriMathcher;

/**
 * Blocs d'instructions pour initialiser l'uriMatcher. 2 uris seront reconnues
 * par le fournisseur de contenu : celle pour accéder à toutes les planètes et
 * celle pour accéder à une planète précise connaissant son identifiant
 */
static {
    uriMathcher = new UriMatcher(UriMatcher.NO_MATCH);
    uriMathcher.addURI(AUTORITE, "planetes", TOUTES_PLANETES);
    uriMathcher.addURI(AUTORITE, "planetes/#", PLANETE_UNIQUE);
}
```

Le fichier ***AndroidManisfest*** de l'application dans laquelle le fournisseur de contenu est défini doit contenir :

- ✓ La déclaration de différentes permissions. Généralement, il y a aura 2 permissions : l'une pour accéder au fournisseur en lecture, l'autre en écriture
- ✓ Un nœud ***provider*** donnant toutes les informations relatives au fournisseur : son nom, son autorité, une indication pour spécifier s'il est utilisable par des applications tierces, le nom des permissions permettant un accès en lecture ou en écriture.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.applicationmms.persistance.fournisseur.exemple.fournisseurplanete">
```

```
    <permission android:name="com.applicationmms.persistance.fournisseur.exemple.fournisseurplanete.READ"
                android:protectionLevel="normal"/>
    <permission android:name="com.applicationmms.persistance.fournisseur.exemple.fournisseurplanete.WRITE"
                android:protectionLevel="normal" />

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <provider
            android:name=".FournisseurPlanete"
            android:authorities="com.applicationmms.persistance.fournisseur.exemple.fournisseurplanete"
            android:enabled="true"
            android:exported="true"
    android:readPermission="com.applicationmms.persistance.fournisseur.exemple.fournisseurplanete.READ"
    android:writePermission="com.applicationmms.persistance.fournisseur.exemple.fournisseurplanete.WRITE">
        </provider>
    </application>

</manifest>
```

Supposons que nous souhaitions dans une autre application accéder aux planètes gérées par le fournisseur de contenu. Nous devons d'abord demander la permission de l'utiliser dans le fichier *AndroidManifest.xml* en insérant les lignes suivantes :

```
<uses-permission
  android:name="com.applicationmms.persistance.fournisseur.exemple.fournisseurplanete.READ" />
<uses-permission
  android:name="com.applicationmms.persistance.fournisseur.exemple.fournisseurplanete.WRITE"
/>
```

L'activité basée sur un ***CursorLoader*** sera codée de la manière suivante :

```
/*
 * Accès au fournisseur des planètes en utilisant un CursorLoader              12/17
 * MainActivity.java
 */
package com.applicationmms.persistance.fournisseur.exemple.accesplaneteloader;

. . .

/**
 * Cette activité permet d'afficher les planètes présentes dans le fournisseur
 * de contenu des planètes.
 * Les planètes sont chargées via un CursorLoader de manière à améliorer la
 * réactivité
 * @author LP MMS
 * @version 1.0
 */
public class MainActivity extends ListActivity
        implements LoaderManager.LoaderCallbacks<Cursor> {

    /** Nom de la colonne contenant le nom de la planète */
    private static final String NOM_COL_NOM = "nom";

    /** Nom de la colonne contenant la nature du sol de la planète */
    private static final String NOM_COL_SOL = "sol";
```

```
    /** Nom de la colonne contenant le rayon de la planète */
    private static final String NOM_COL_RAYON = "rayon";

    /** Nom de la colonne contenant la distance au soleil */
    private static final String NOM_COL_DISTANCE = "distance";


    /**
     * Curseur sur l'ensemble ou un sous-ensemble des pays de la base
     * Ce sont les pays référencés par ce curseur qui sont actuellement
     * affichés par la liste associée à l'activité */
    private Cursor curseurSurBase;

    /**
     * Adaptateur : il sert d'intermédiaire entre le curseur et la vue qui
     *  affiche les planètes. C'est curseurSurBase qui lui est associé
     */
    private SimpleCursorAdapter adaptateur;


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

        /*
         * Création de l'adaptateur : pour l'instant la source des données
         * est initialisée à null, puisque le cuseur n'a pas encore été chargé
         */
        adaptateur =
                new SimpleCursorAdapter(this,
                        R.layout.ligne_liste,
                        null,
                        new String[] {NOM_COL_NOM,
                                NOM_COL_SOL,
                                NOM_COL_RAYON,
                                NOM_COL_DISTANCE},
                        new int[] {R.id.nom_planete,
                                R.id.sol_planete,
                                R.id.rayon_planete,
                                R.id.distance_planete}, 0);
        setListAdapter(adaptateur);

        // on demander l'initialisation d'un chargeur de curseur
        getLoaderManager().initLoader(0, null, this);
    }


    /**
     * Méthode invoquée lors de la création du chargeur de curseur.
     */
    @Override
    public Loader<Cursor> onCreateLoader(int i, Bundle bundle) {
        return new CursorLoader(this,

Uri.parse("content://com.applicationmms.persistance.fournisseur.exemple.fournisseurplanete/pla
netes"),
                null, null, null, null);
    }

    /**
     * Méthode invoquée lorsque le curseur a fini d'être chargé
     */
    @Override
    public void onLoadFinished(Loader<Cursor> loader, Cursor curseurMisAJour) {
        adaptateur.swapCursor(curseurMisAJour);
    }

    /**
```

```
     * Méthode appelée lors de la réinitialisation du chargeur de curseur
     */
    @Override
    public void onLoaderReset(Loader<Cursor> loader) {
        adaptateur.swapCursor(null);
    }
}
```

V) Compléments

Remarques

La classe ***ContentUris*** aide à réaliser la création de ces URI.

Pour ajouter un identifiant à une URI, on peut écrire :

```
Uri uriDeBase = Uri.parse("content://com.licencemms.application.provider/photos") ;
Uri monUri = ContentUris.withAppendedId(uriDeBase, 20) ;
```

ou

```
Uri monUri = ContentUris.withAppendedId(uriDeBase, "20") ;
```

Et ensuite pour accéder à l'enregistrement souhaité via un curseur :

```
Cursor c = managedQuery(monUri, null, null, null, null) ;
```

Pour au contraire extraire l'identifiant d'une URI, on écrira :

```
long monId = ContentUris.parseId(uri) ;
```

Remarque sécurité

Nous avons dit que pour accéder à un fournisseur de contenu, il fallait passer une instance de type ***ContentResolver***. En fait les objets fournisseurs de contenu eux-mêmes sont de type *ContentProvider*. Ainsi si nous invoquons une méthode *m* sur l'instance *ContentResolver*, cet appel va engendrer un appel à la même méthode *m* sur l'instance de type *ContentProvider* cible.

Pourquoi ne peut-on pas appeler directement la méthode *m* sur l'instance de type *ContentProvider* ? pour des raisons de sécurité. Android est ainsi certain que l'application courante a bien reçu les autorisations nécessaires à l'exécution de cette opération. L'objet *ContentResolver* gére aussi la communication entre les différents processus impliqués dans la gestion du fournisseur de contenu.

Les services Web

I) Introduction

Les applications mobiles sont fréquemment associées à un serveur. Ce dernier peut fournir de nombreux services : interrogation d'une base de données, collaboration entre applications, sauvegarde, flux de données, traitements intensifs …

Android fournit une base solide pour les applications en réseau. On peut utiliser les packages :
- ✓ *java.net.** API réseau de base du langage Java Standard
- ✓ *android.net.** fonctionnalités spécifiques à la plate-forme Android

Il y a **trois principales options de communication** entre une application et un serveur :

1) **Communication basée sur les sockets**
   C'est le mode de communication bas niveau que nous ne développerons pas ici.

2) **Protocole HTTP**
   Il s'agit du protocole Web utilisé entre les navigateurs et les sites. Il est implémenté entièrement par le biais de l'API Apache HTTP client. Il permet à une application de se comporter comme un navigateur. Il permet également de créer le client d'un site Web existant.

3) **Services Web**
   Les services Web sont une solution largement adoptée pour la communication entre applications. Ils s'appuient sur les formats XML et JSON (format pour l'échange de données) et des communications HTTP. Nous sommes tous de gros utilisateurs de services Web : consultation de la météo sur un smartphone, par exemple. Android fournit les outils nécessaires pour implémenter les clients de services Web.

II) Préambule basé sur un exemple : Comment télécharger une image ?

2.1) Se connecter à Internet

Une application peut utiliser la connexion Internet du terminal afin d'accéder à des informations stockées en ligne. Pour ce faire, il faut préciser une ou deux permissions dans le fichier *AndroidManisfest.xml* :

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

La première autorise à utiliser la connexion Internet du terminal. Ainsi, l'utilisateur sera prévenu que ce type d'accès est réalisé par l'application, quand celle-ci sera installée.

La deuxième autorise à vérifier l'état du réseau. Dans le fichier *AndroidManifest.xml*, ces deux permissions se placent immédiatement à l'intérieur du nœud *manifest*, et à l'extérieur du nœud *application.*

2.2) Vérifier si une connexion est disponible

Avant d'utiliser la connexion Internet du terminal, donc avant de télécharger des données, ou d'accéder à un service Web, par exemple, il faut vérifier la disponibilité de la connexion. Pour ce faire, on passe par l'intermédiaire d'une instance de type *ConnectivityManager* (gestionnaire de connexion). On peut, par exemple, écrire le code suivant :

```
// on vérifie si la connexion à Internet est possible
ConnectivityManager gestionnaireConnexion =
        (ConnectivityManager) getSystemService(Context.CONNECTIVITY_SERVICE);

NetworkInfo informationReseau = gestionnaireConnexion.getActiveNetworkInfo();

if (informationReseau == null || ! informationReseau.isConnected()) {

    // problème de connexion réseau
    boutonTelecharger.setEnabled(false);
    Toast.makeText(this,
            getResources().getString(R.string.message_erreur_connexion),
            Toast.LENGTH_LONG).show();
}
```

On a aussi la possibilité de savoir si la connexion active est une connexion Wifi :

```
if (informationReseau.getType() == ConnectivityManager.TYPE_WIFI) {
```

2.3) Gestion du changement de connectivité

Il est bien sûr possible que l'état de la connectivité évolue au cours du cycle de vie de l'application ou de l'activité. Pour faciliter la gestion de ces changements, l'application peut s'abonner au changement de connectivité du terminal (hormis le Wifi). Pour ce faire, il faut utiliser les classes *PhoneStateListener* et *TelephonyManager*.

On créé d'abord un écouteur destiné à écouter les changements d'état de la connectivité, via la classe *PhoneStateListener*, puis on associe cet écouteur au gestionnaire de téléphonie. Le code à écrire aura la structure suivante :

```
// on crée un écouteur pour les changements de connectivité
PhoneStateListener ecouteurConnectivite = new PhoneStateListener() {

    @Override
    public void onDataConnectionStateChanged(int etat) {
        switch(etat) {
            case TelephonyManager.DATA_CONNECTED :
                // appareil connecté et réseau disponible. TODO : Compléter
                break;
            case TelephonyManager.DATA_CONNECTING :
                // appareil en cours de connexion. TODO : Compléter
                break;
            case TelephonyManager.DATA_DISCONNECTED :
                // appareil déconnecté. TODO : Compléter
                break;
            case TelephonyManager.DATA_SUSPENDED :
                // apparail connecté mais transfert de données impossible.
                // TODO : Compléter
                break;
        }
        super.onDataConnectionStateChanged(etat);
    }
};

// on associe l'écouteur au gestionnaire de téléphonie
TelephonyManager gestionnaireTelephonie =
        (TelephonyManager) getSystemService(TELEPHONY_SERVICE);
gestionnaireTelephonie.listen(ecouteurConnectivite,
                              PhoneStateListener.LISTEN_DATA_CONNECTION_STATE);
```

2.4) Téléchargement d'une image

Dans cet exemple, on suppose que l'on souhaite télécharger une image connaissant son URL. Cette image sera ensuite affichée sur la vue de l'activité principale.

Le téléchargement de l'image étant un traitement long, il sera délégué à une tâche asynchrone. L'activité principale créera celle-ci, et lui communiquera un écouteur afin de réaliser le traitement adéquat que l'on souhaite réaliser à l'issue du téléchargement. Deux cas sont envisagés : le téléchargement s'est déroulé sans erreur, ou bien une erreur s'est produite.

Pour décrire l'écouteur, nous définissons une interface nommée *EcouteurFinTelechargement*. Notons que cette manière de procéder n'est pas la seule possible. Une variante consiste pour l'activité principale à communiquer son contexte à la tâche asynchrone, lors de la création de cette dernière (voir le TP sur les tâches asynchrones).

Nous aboutissons donc à la définition :

- ✓ d'une interface *EcouteurFinTelechargement*,
- ✓ d'une classe activité principale
- ✓ et d'une tâche asynchrone.

```
/*
 * Interface gérant la fin du téléchargement d'une image
 * EcouteurFinTelechargement.java                                    02/17
 */
package com.exemple.internet.communication.extelechargerimage;

import android.graphics.Bitmap;

/**
 * Interface contenant 2 méthodes permettant de gérer la fin du téléchargement
 * d'une image.
 *   - la méthode apresTelechargementImage est invoquée si le téléchargement s'est
 *     déroulé sans erreur. L'argument est l'image téléchargée.
 *   - la méthode apresErreurTelechargement est invoquée après une erreur lors du
 *     téléchargement
 */
public interface EcouteurFinTelechargement {

     void apresTelechargementImage(Bitmap bitmap);
     void apresErreurTelechargement();
}
```

Interface *EcouteurFinTelechargement*

```
/*
 * Activité permettant de télécharger une image
 * MainActivity.java                                    02/17
 */
package com.exemple.internet.communication.extelechargerimage;

import android.content.Context;
import android.graphics.Bitmap;
import android.net.ConnectivityManager;
import android.net.NetworkInfo;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.telephony.PhoneStateListener;
import android.telephony.TelephonyManager;
import android.view.View;
import android.widget.Button;
import android.widget.ImageView;
import android.widget.Toast;

/**
 * Activité permettant de télécharger une image
 * L'image est située à une URL fixe
 * @author Servières
 * @version 1.0
 */
public class MainActivity extends AppCompatActivity
                                        implements  EcouteurFinTelechargement{

    /** URL de l'image à télécharger */
    private static final String URL_IMAGE =
            "https://cdn.pixabay.com/photo/2016/12/27/21/03/lone-tree-1934897_960_720.jpg";

    /** Zone d'affichage de l'image */
    private ImageView zoneImage;

    /** Bouton pour lancer le téléchargement */
    private Button boutonTelecharger;
```

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        zoneImage = (ImageView) findViewById(R.id.id_image);
        boutonTelecharger = (Button) findViewById(R.id.bouton_telecharger);

        // vérifier et gérer les changements relatifs à la connexion
        gererConnexion();
    }

    /**
     * Invoquée lors du clic sur le bouton de téléchargement
     * @param bouton  bouton à l'origine du clic
     */
    public void clicTelecharger(View bouton) {

        // création d'une tâche asynchrone pour réaliser le téléchargement
        new TacheTelechargement(this).execute(URL_IMAGE);
    }


    /**
     * Invoquée lorsque le téléchargement se termine sans erreur
     * (c'est la tâche asynchrone qui déclenche l'appel)
     * @param bitmap  Image téléchargée
     */
    @Override
    public void apresTelechargementImage(Bitmap bitmap) {
        zoneImage.setImageBitmap(bitmap);
    }

    /**
     * Invoquée si une erreur se produit lors du téléchargement
     * (c'est la tâche asynchrone qui déclenche l'appel)
     */
    @Override
    public void apresErreurTelechargement() {
        Toast.makeText(this,
              getResources().getString(R.string.message_erreur), Toast.LENGTH_SHORT).show();
    }


    /**
     * Effectuer les opérations pour vérifier si la connexion est disponible,
     * et gérer les changements de disponibilité de celle-ci
     */
    private void gererConnexion() {

        // on vérifie si la connexion à Internet est possible
        ConnectivityManager gestionnaireConnexion =
                (ConnectivityManager) getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo informationReseau = gestionnaireConnexion.getActiveNetworkInfo();

        if (informationReseau == null || ! informationReseau.isConnected()) {

            // problème de connexion réseau
            boutonTelecharger.setEnabled(false);
            Toast.makeText(this,
                    getResources().getString(R.string.message_erreur_connexion),
                    Toast.LENGTH_LONG).show();
        }
```

```
        // on crée un écouteur pour les changements de connectivité
        PhoneStateListener ecouteurConnectivite = new PhoneStateListener() {

            @Override
            public void onDataConnectionStateChanged(int etat) {
                switch(etat) {
                    case TelephonyManager.DATA_CONNECTED :
                        // appareil connecté et réseau disponible. TODO : Compléter
                        break;
                    case TelephonyManager.DATA_CONNECTING :
                        // appareil en cours de connexion. TODO : Compléter
                        break;
                    case TelephonyManager.DATA_DISCONNECTED :
                        // appareil déconnecté. TODO : Compléter
                        break;
                    case TelephonyManager.DATA_SUSPENDED :
                        // apparail connecté mais transfert de données impossible
                        // TODO : Compléter
                        break;
                }
                super.onDataConnectionStateChanged(etat);
            }
        };

        // on associe l'écouteur au gestionnaire de téléphonie
        TelephonyManager gestionnaireTelephonie =
                (TelephonyManager) getSystemService(TELEPHONY_SERVICE);
        gestionnaireTelephonie.listen(ecouteurConnectivite,
                PhoneStateListener.LISTEN_DATA_CONNECTION_STATE);
    }
}
```

Activité principale

```
/*
 * Tâche asynchrone effectuant un téléchargement
 * TacheAsynchrone.java                                    02/17
 */

package com.exemple.internet.communication.extelechargerimage;

import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.os.AsyncTask;
import java.io.IOException;
import java.io.InputStream;
import java.net.URL;

/**
 * Tâche asynchrone réalisant le téléchargement d'une image située à une URL précise
 * @author  Servières
 * @version 1.0
 */
public  class TacheTelechargement extends AsyncTask<String, Void, Bitmap> {

    /** Ecouteur de la fin du téléchargement */
    private EcouteurFinTelechargement mListener;

    /**
     * Constructeur permettant d'initialiser l'écouteur de fin de téléchargement
     * @param listener  écouteur de fin du téléchargement
```

```
     */
    public TacheTelechargement(EcouteurFinTelechargement listener) {
        mListener = listener;
    }


    /**
     * Traitement réaliser en arrière plan
     * @param args  un argument de type String contenant l'URL de l'image à télécharger
     * @return  l'image téléchargée sous la forme d'une instance de type Bitmap
     */
    @Override
    protected Bitmap doInBackground(String... args) {

        // téléchargement de l'image dont l'URL est donnée en tant que premier argument
        try {

            // l'image est renvoyée sous la forme d'un objet de type Bitmap
            return BitmapFactory.decodeStream((InputStream)new URL(args[0]).getContent());
        } catch (java.net.MalformedURLException erreur) {
            // la valeur null sera renvoyée
        } catch (IOException erreur) {
            // la valeur null sera renvoyée
        }
        return null;
    }

    @Override
    protected void onPostExecute(Bitmap bitmap) {
        if (bitmap != null) {

            // le téléchargement s'est déroulé sans erreur
            mListener.apresTelechargementImage(bitmap);
        } else {
            mListener.apresErreurTelechargement();
        }
    }
}
```

Tâche asynchrone réalisant le téléchargement

L'image est téléchargée grâce à l'instruction suivante :

```
BitmapFactory.decodeStream(
            (InputStream) new URL(args[0]).getContent());
```

Détaillons les différentes étapes de l'instruction :

- ✓ Une instance de type URL est créée à partir de la chaîne de caractères contenant l'URL :
  `new URL(args[0])`

- ✓ L'appel à *getContent()* sur cette instance renvoie le contenu situé à cette URL. Notons que cet appel établi aussi la connexion à Internet.

- ✓ Le résultat renvoyé par *getContent()* est de type *Object*. Il faut donc le convertir en *InputStream* afin de le donner en argument à la méthode *decodeStream*. Cette dernière va transformer celui-ci en une instance de type *Bitmap*, donc une image.

III) Communication avec HTTP

3.1) Rappel : Principe de la communication avec HTTP

HTTP est le principal protocole de communication du Web. Il s'agit d'un protocole de requêtes/réponses basé sur TCP/IP (TCP = Transmission Control Protocol, IP =Internet Protocol).

Les principales étapes du protocole sont les suivantes :

1. Le client se connecte via le protocole IP sur le serveur
2. Le client envoie une requête HTTP
3. Le serveur renvoie une réponse
4. La connexion IP est fermée (cependant, bon nombre des implémentations laissent ouverte la connexion IP en vue d'une réutilisation par les requêtes suivantes)

Les requêtes sont le plus souvent de deux types :

- ✓ Méthode **GET** pour la demande d'un document au serveur
- ✓ Méthode **POST** pour l'envoi de données vers le serveur, et une réponse est attendue

Dans tous les cas, un code de retour est renvoyé.

Rappel : Structure d'un message HTTP

Une requête HTTP se compose de trois parties :

1) Une ligne de commande : la commande HTTP
Elle est composée de 3 parties : *Méthode* *identifiant (URI)* *HTTP_VERSION*
La *méthode* indique l'action à réaliser
L'*identifiant* indique le nom du document demandé ou de l'action à traiter. Il peut aussi indiquer le nom d'un programme ou d'un script à exécuter
La *version* est par exemple : http/1.1

2) Une variable d'en-tête : les métadonnées

3) Le corps du message, c'est-à-dire le contenu du message. Il contient les données à communiquer au serveur si la requête est de type POST.

Le message de réponse a une structure similaire.

3.2) Remarque – Serveur local

Lors du développement, il est fréquent d'utiliser un serveur HTTP local, nommé habituellement *localhost*. Si nous utilisons l'émulateur, et que nous essayons d'accéder à un serveur Web sur le poste de développement, *localhost* représente l'adresse interne de l'appareil Android. Au sein de l'émulateur, l'adresse **10.0.2.2** fournit l'accès au poste de développement qui exécute l'émulateur.

3.3) Implémentation d'un client http avec la classe DefaultHttpClient

Android est livré avec une implémentation de l'API Apache HTTP Client. Le package qui la contient est *org.apache.http.client*. Ce package contient donc toutes les classes utiles pour gérer la connexion avec le serveur.

Il contient aussi un ensemble d'interfaces Java pour gérer la communication HTTP. Parmi celles-ci, on trouve :

`HttpMessage`	pour modéliser une requête ou une réponse
`Header`	pour modéliser une encapsulation des en-têtes HTTP
`HttpEntity`	pour modéliser le corps du message lorsqu'il existe
`HttpConnection`	pour modéliser la connexion réseau sous-jacente

L'utilisation de l'API est simplifiée par la classe **`DefaultHttpClient`**. Elle représente un client HTTP opérationnel qui fournit l'implémentation de toutes les interfaces nécessaires.

Comment demander l'exécution d'une requête HTTP ?

Pour communiquer avec un serveur à l'aide du protocole HTTP, il faut :

1) Construire une requête HTTP
2) Créer une classe pour traiter la réponse.
   La classe **`ResponseHandler<T>`** permet ce traitement. D'une part, elle convertit les codes erreur en exception Java de type **`ClientProtocoleException`**, et d'autre part elle convertit les données de la réponse en un objet de type **`T.`**
3) Créer une instance de **`DefaultHttpClient`** pour disposer d'un client HTTP
4) Exécuter la requête en invoquant la méthode ***execute*** sur le client HTTP, sachant que
   - la valeur de retour de la méthode ***execute*** est la réponse du serveur
   - une exception **`ClientProtocolException`** est déclenchée en cas d'erreur
5) Fermer la connexion

A noter : depuis l'API 11, les connexions réseau ne doivent pas être réalisées dans le *thread* principal. Une exception `android.os.NetworkOnMainThreadException` est déclenchée dans ce cas. De toute façon, pour ne pas nuire à la réactivité de l'application, il est fortement déconseillé de gérer la connexion dans le *thread* UI. Une tâche asynchrone est une bonne option pour déléguer le travail de connexion à un *thread* dédié.

Plus concrètement, le code Java à écrire suit le schéma suivant :

```
// construction de l'URL de demande contenant l'adresse IP
String url = « placer ici l'URL de connexion »;

// résultat fourni par le web service
String resultat = "";

// déclaration d'un client HTTP
HttpClient clientHttp = null;

try {
    // initialisation d'un client Http et de la requête
    clientHttp = new DefaultHttpClient();
    HttpGet requete = new HttpGet(url);

    // déclaration d'un gestionnaire pour recevoir la réponse
    ResponseHandler<String> handler = new BasicResponseHandler();

    // on demande au client d'exécuter la requête
    resultat = clientHttp.execute(requete, handler);

} catch (ClientProtocolException erreur) {
```

```
        Log.i(ETIQUETTE_LOG, "Exception ClientProtocolException");
    } catch (IOException erreur) {
        Log.i(ETIQUETTE_LOG, "Exception IOException");
    } finally {
        if (clientHttp != null) {
            // on ferme la connexion avec le client
            clientHttp.getConnectionManager().shutdown();
        }
    }
```

Remarque

Le module Apache tel que utilisé ci-dessus a été déprécié par les concepteurs d'Android depuis l'API de niveau 22. Cependant, d'après les informations disponibles sur Internet, il semblerait que cette dépréciation soit due uniquement à des désaccords entre Google et Apache.

3.4) Implémentation d'un client http avec la classe AndroidHttpClient

La plateforme Android fournit la classe ***AndroidHttpClient*** qui implémente l'interface ***HttpClient*** et qui est configurée spécialement pour Android. Une particularité de celle-ci est d'interdire l'envoi de requêtes http depuis le *thread* principal, si tel est le cas une exception est levée.

Pour obtenir une instance de type ***AndroidHttpClient***, il ne faut pas invoquer le constructeur de la classe, mais invoquer la méthode statique ***newInstance***. Dans l'exemple ci-dessous, son argument est égal à *null*. Il pourrait aussi être une chaîne de caractères contenant des informations relatives au client.

Plus concrètement, le code Java à écrire pour une requête *get* suit le schéma suivant :

```
// construction de l'URL de demande contenant l'adresse IP
String url = « placer ici l'URL de connexion »;

// résultat fourni par le web service
String resultat = "";

// déclaration d'un client HTTP
AndroidHttpClient clientHttp = null;

try {
    // initialisation d'un client Http et de la requête
    clientHttp = AndroidHttpClient.newInstance(null);
    HttpGet requete = new HttpGet(url);

    // déclaration d'un gestionnaire pour recevoir la réponse
    ResponseHandler<String> handler = new BasicResponseHandler();

    // on demande au client d'exécuter la requête
    resultat = clientHttp.execute(requete, handler);

} catch (ClientProtocolException erreur) {
    Log.i(ETIQUETTE_LOG, "Exception ClientProtocolException");
} catch (IOException erreur) {
    Log.i(ETIQUETTE_LOG, "Exception IOException");
} finally {
    if (clientHttp != null) {
        // on ferme la connexion avec le client
        clientHttp.getConnectionManager().shutdown();
    }
}
```

3.5) Remarque : La bibliothèque *Volley*

La bibliothèque ***Volley*** est destinée à faciliter les développements réseaux sur Android. Son code source doit au préalable être téléchargé, et ensuite intégré au projet Android Studio.

IV) Les services Web – Généralités indépendantes d'Android

Nous sommes tous des utilisateurs de services web au quotidien : par exemple, lorsque nous consultons la météo sur un smartphone. Le développement des terminaux mobiles augmente les besoins en service Web. Les systèmes d'information des entreprises doivent les intégrer pour de nouveaux besoins. Par exemple, un commercial doit pouvoir saisir les commandes depuis l'entreprise de son client, vérifier des disponibilités, consulter un agenda... L'avantage est une prise en compte en temps réel des informations.

Les services Web interconnectent les applications distribuées. Un service Web est une application Web sans vue ; c'est le client du service qui fournit la vue.

De manière informelle, on peut dire qu'un service web est un ensemble de procédures qui sont déployées sur un serveur et qu'il est possible d'utiliser sur un poste client, à distance, afin de sous-traiter la réalisation d'une opération. Il peut être utilisé dans un contexte client-serveur.

Définition officielle d'un service Web (par le W3C, World Wide Web Consortium)

Un **service Web** est un système logiciel conçu pour permettre l'interopérabilité entre les machines sur un réseau. Il possède une interface qui décrit, dans un format normalisé, le moyen de communiquer avec la machine (par exemple : WSDL). D'autres systèmes interagissent avec les services Web, conformément à l'interface, en utilisant les messages SOAP envoyés par le protocole HTTP et écrits en XML, en liaison avec d'autres normes standards du Web.

Les auteurs du livre « Les web services » paru aux éditions Ellipses (notons qu'il s'agit d'une bonne introduction aux services Web), proposent la définition simplifiée suivante. Un **service Web** est une application informatique possédant une URI (Uniform Resource Identifier), hébergée par un serveur d'applications, qui est composée de procédures dont l'exécution représente un service proposé à un autre programme informatique (nommé client) et qui est accessible sur internet par l'utilisation de protocoles standards (HTML, XML ...).

Un service Web utilise **3 protocoles**, à des niveaux différents :
- ✓ Un protocole pour décrire le service : il s'agit de lister les méthodes disponibles et leurs arguments (WSDL par exemple)
- ✓ Un protocole pour décrire la composition des messages (SOAP par exemple)
- ✓ Un protocole de transport pour faire circuler les informations sur Internet (HTTP, SMTP …)

Les différentes architectures d'un service Web : XML-RPC, SOAP et REST.

Généralement, les services Web existent sous trois architectures différentes : XML-RPC, SOAP et REST.

L'architecture **XML-RPC** est la plus ancienne (depuis 1998). Elle est composée de XML et RPC (Remote Procedure Call), un protocole réseau. La couche transport est assurée par le langage HTTP. Le langage de description des messages est XML.

Plus précisément, un client envoie au serveur un document XML par la méthode POST du protocole HTTP. Le serveur le reçoit, le traite et envoie la réponse sous la forme d'un document XML, en utilisant le protocole HTTP.

Le protocole XML-RPC ne fournit aucune description des services proposés. Plus précisément, les services ne sont pas exposés. Cette particularité ne pose aucun problème pour le développement interne à une entreprise. Par contre c'est un handicap pour un service Web proposé sur internet, qui pourra être plus difficilement intégré au sein des applications.

Une autre possibilité est d'utiliser **SOAP**. En réalité **SOAP** (Services Simple Object Access Protocol) n'est pas réellement une architecture. Il s'agit en fait d'un protocole de description de messages. On peut dire aussi qu'il s'agit d'une architecture client-serveur utilisant les technologies SOAP/HTTP/WSDL. Les messages (requêtes ou réponses) sont structurés dans le format XML, et sont transmis via HTTP. L'interface pour communiquer avec le service Web est définie et publiée grâce au format WSDL (une norme). Ces services sont adaptés aux applications d'entreprise.

Les services de type **REST** (REpresentational State Transfer) sont légers et informels. Les données utilisent en général les formats XML et JSON (le plus commun des 2). Ils sont facilement exploitables par les navigateurs Web et les clients mobiles. Ils ont du succès en ce moment, car ils sont moins lourds que SOAP.

Notons qu'une autre forme de services Web émerge et occupe de plus en plus de place : les API JavaScript (facilite la connexion aux services pour les utilisateurs des technologies web).

Implémentation du client d'un service Web

Pour communiquer avec un service Web, il faut :

1) Se connecter au client via le protocole HTTP (comme présenté précédemment, par exemple)
2) Créer une requête avec les données de l'application
3) Envoyer la requête
4) Analyser la réponse

Comme dit précédemment, les messages échangés s'appuient souvent sur les formats XML et JSON. Au niveau du code Java, les données des messages structurées dans l'un ou l'autre de ces formats peuvent être fabriquées à partir des types Java, ou au contraire, elles peuvent être converties vers des types Java.

Ces deux formats sont donc embarqués dans Android. On trouve, dans Android,

- ✓ les processeurs XML standards de Java : SAX (Simple API for XML), DOM (Document Object Model), **XMLPullParser**. Android recommande d'utiliser ce dernier, car il est efficace et simple d'utilisation.
- ✓ et aussi un **analyseur JSON**, sous la forme de classes du package *org.json*.

V) REST

REST est une architecture définie en 2000, ce n'est surtout pas un protocole, comme par exemple SOAP. Elle n'a pas de spécification W3C, c'est-à-dire elle n'est pas normalisée. Les services Web d' eBay, Amazon.com, Google, Yahoo, par exemple utilisent cette architecture.

La particularité d'une architecture REST est que le web service ne conserve pas l'état courant : il est sans état (*stateless*). Le client doit donc toujours envoyer toutes les informations utiles au service web (l'avantage est une économie de mémoire pour le serveur). Il utilise le protocole HTTP pour communiquer entre le client et le serveur (méthodes GET, POST, PUT, DELETE).

Avec cette architecture, toutes les ressources sont identifiées par une URI (Uniform Resource Identifiier). Une URI permet donc d'identifier une ressource sur un réseau de manière unique. Une URL est une URI.

Les informations échangées entre le client et le serveur peuvent être de formes diverses : XML, JSON, texte, ou autre.

Comment communiquer avec un service REST ?

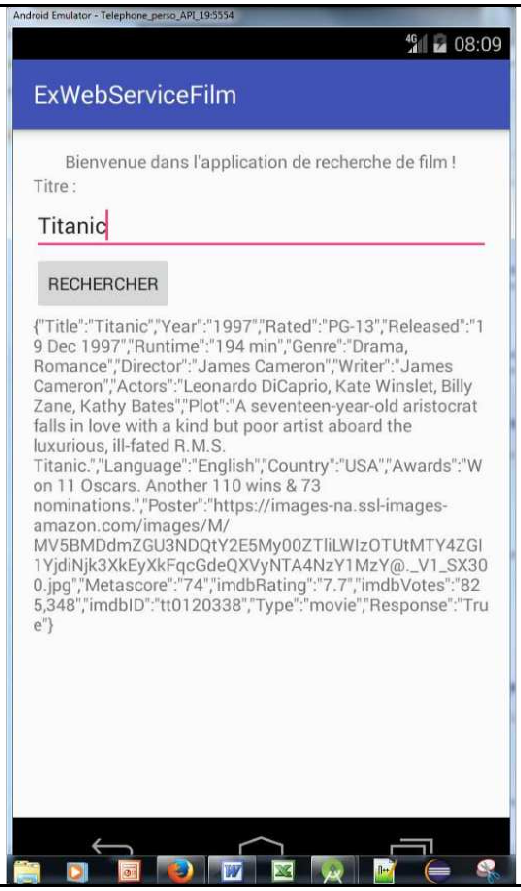
1) Il faut d'abord créer un message de requête (ce message peut être au format JSON, par exemple, mais pas nécessairement)
2) Ensuite, il faut envoyer le message de requête, et recevoir la réponse
3) Une fois la réponse reçue, il faut l'analyser (analyser du texte au format JSON, par exemple).

Exemple

Nous allons utiliser un service Web disponible gratuitement, via l'url www.omdbapi.com. Ce service permet d'interroger une base de données afin d'obtenir la fiche descriptive d'un film, à partir de son titre, par exemple. La réponse est renvoyée dans le format JSON

Dans un premier temps, l'application se contentera d'afficher le code JSON renvoyé par la requête. Un exemple d'exécution est montré ci-contre.

Pour aboutir à ce comportement, nous définissions une classe Java correspondant à l'activité, et aussi une tâche asynchrone qui sera chargée de se connecter au serveur, d'envoyer la requête et de récupérer la réponse.



```java
/*
 * Exemple d'utilisation d'un service Web : recherche d'informations sur un film
 * MainActivity.java                                    02/17
 */
package com.exemple.internet.communication.exwebservicefilm;

import . . .

/**
 * Classe principale de l'application permettant d'interroger un service Web.
 * Celui-ci permet de consulter la fiche descriptive d'un film.
 * Il se situe à l'URL : http://www.omdbapi.com
 * L'utilisateur est invité à saisir le titre d'un film.
 * Si le service Web comporte sa fiche descriptive, un résumé de celle-ci est affiché,
 * directement dans le format JSON
 * @author Servières
 * @version 1.0
 */
public class MainActivity extends AppCompatActivity {

    /** URL du sevice Web, paramétrée par le titre du film recherché */
    private static final String URL_FILM = "http://www.omdbapi.com/?t=%s&y=&plot=short&r=json";

    /** Zone de saisie du titre du film recherché */
    private EditText zoneTitre;

    /** Zone pour afficher le résultat de la recherche */
    private TextView zoneResultat;

    /** Bouton pour lancer la recherche du film */
    private Button boutonRechercher;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        zoneTitre = (EditText) findViewById(R.id.zone_titre);
        zoneResultat = (TextView) findViewById(R.id.zone_resultat);
        boutonRechercher = (Button) findViewById(R.id.bouton_rechercher);

        // on vérifie si une connexion est disponible
        ConnectivityManager gestionnaireConnexion =
                (ConnectivityManager) getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo informationReseau = gestionnaireConnexion.getActiveNetworkInfo();
        if (informationReseau == null ||  ! informationReseau.isConnected()) {

            // problème de connexion réseau : le bouton de recherche est désactivé
            boutonRechercher.setEnabled(false);
            Toast.makeText(this,
                           getResources().getString(R.string.message_erreur_connexion),
                           Toast.LENGTH_LONG).show();
        }
    }


    public void clicRechercher(View bouton) throws UnsupportedEncodingException {
        String titre = URLEncoder.encode(zoneTitre.getText().toString(), "UTF-8");
        String url = String.format(URL_FILM, titre);

        // création d'une tâche asynchrone pour réaliser le téléchargement
        new TacheRecherche(this).execute(url);

    }

    public void setZoneResultat(String texte){
        zoneResultat.setText(texte);
    }
}
```

Fichier *MainActivity.java*

```
/*
 * Tâche asynchrone qui réalise la connexion à un site distant
 * TacheRecherche.java                                                    02/17
 */
package com.exemple.internet.communication.exwebservicefilm;

import . . .

/**
 * @author Servières
 * @version 1.0
 */
public class TacheRecherche extends AsyncTask<String, Void, StringBuilder> {


    /** Etiquette pour les messages de log */
    private static final String TAG_LOG = "ACCES WEB";

    /** Activité parente de la tâche, celle qui a créé la tâche */
    private MainActivity activiteParente;

    /**
     * Constructeur de la tâche
     * @param parent  activité qui a créé la tâche
     */
    public TacheRecherche(MainActivity parent) {
        activiteParente = parent;
    }


    @Override
    protected StringBuilder doInBackground(String... args) {

        /*
         * le résultat renvoyé sera le texte dans le format JSON décrivant le
         * film recherché, ou bien une chaîne vide, si problème de connexion,
         * ou si le film n'a pas été trouvé
         */
        StringBuilder resultat = new StringBuilder();

        HttpURLConnection connexion = null;  // connexion au site distant
        BufferedReader reader = null;

        try {

            URL url = new URL(args[0]);
            connexion = (HttpURLConnection) url.openConnection();
            connexion.connect();

            /*
             * On récupère les données en provenance de la connexion.
             * Elles sont de type InputStream. Pour pouvoir les lire,
             * on les transforme en BufferedReader
             */
            reader = new BufferedReader(
                    new InputStreamReader(connexion.getInputStream(), "UTF-8"));
            String chaineLue= "";

            // on lit une par une chacune des lignes de l'objet BufferedReader
            while ((chaineLue = reader.readLine()) != null){
                resultat.append(chaineLue).append("\n");
            }

        } catch (java.net.MalformedURLException erreur) {
            // une chaîne vide sera renvoyée
            Log.i(TAG_LOG, "url mal formatée");
```

```
        } catch (IOException erreur) {
            Log.i(TAG_LOG, "problème lecture réponse");
            // une chaîne vide sera renvoyée
        } finally {

            // on referme la connexion et le flux de lecture
            if (connexion != null) {
                connexion.disconnect();
            }
            try {
                if (reader != null) {
                    reader.close();
                }
            } catch(IOException erreur) {
                Log.i(TAG_LOG, "problème fermeture fux de lecture");
            }

        }
        return resultat;
    }

    @Override
    protected void onPostExecute(StringBuilder resultat) {
        if (resultat.length() == 0) {

            // on n'a pas pu récupérer la réponse
            Toast.makeText(activiteParente,
                    activiteParente.getResources().getString(R.string.message_erreur),
                    Toast.LENGTH_LONG).show();
        } else {

            // le code JSON est écrit directement dans la zone de résultat
            activiteParente.setZoneResultat(resultat.toString());
        }
    }
}
```

Tâche asynchrone

VI) Le format JSON et son analyseur

Ce format est concis et facile à traiter. Syntaxe de base :

- Les **objets** sont déclarés entre { et }
- Les **propriétés** des objets sont représentés par des couples nom/valeur (syntaxe `nom :valeur`). Les couples sont séparés par des ‘,’.
- Les **valeurs** peuvent être : des entiers , des réels , des booléens (true ou false), des objets, des chaînes de caractères (elles contiennent des caractères Unicode ou des échappements avec \), des tableaux, ou null.
- Les **tableaux** commencent et se terminent pas des crochets [et] et contiennent des valeurs, elles-mêmes séparées par des ‘,’.

Exemple 1

Le texte ci-dessous représente un objet dont la valeur est un tableau contenant lui-même 2 objets. Chacun des 2 possède 3 propriétés : nom du produit, quantité, et prix.

```
{"commande" : [
      {"produit":"ramette papier",
       "quantite":10,
       "prix":4.5},
      {"produit":"classeur ",
       "quantite":20;
       "prix":2.3}]}
```

Exemple 2

Le texte ci-dessous correspond au code JSON par la requête de l'application utilisée comme exemple, dans la section précédente.

```
{"Title":"Titanic",
 "Year":"1997",
 "Rated":"PG-13",
 "Released":"19 Dec 1997",
 "Runtime":"194 min",
 "Genre":"Drama, Romance",
 "Director":"James Cameron",
 "Writer":"James Cameron",
 "Actors":"Leonardo DiCaprio, Kate Winslet, Billy Zane, Kathy Bates",
 "Plot":"A seventeen-year-old aristocrat falls in love with a kind but poor
artist aboard the luxurious, ill-fated R.M.S. Titanic.",
 "Language":"English",
 "Country":"USA",
 "Awards":"Won 11 Oscars. Another 110 wins & 73 nominations.",
 "Poster":"https://images-na.ssl-images-
amazon.com/images/M/MV5BMDdmZGU3NDQtY2E5My00ZTliLWIzOTUtMTY4ZGI1YjdiNjk3XkEyXkF
qcGdeQXVyNTA4NzY1MzY@._V1_SX300.jpg",
 "Metascore":"74",
 "imdbRating":"7.7",
 "imdbVotes":"825,348",
 "imdbID":"tt0120338",
 "Type":"movie",
 "Response":"True"}
```

Comment traiter les données JSON ?

Les classes du package `json.org` permettent la création et l'analyse de données au format JSON.

Une donnée JSON est modélisée par la classe `JSONObject`. Il s'agit en fait d'un tableau associatif de paires (nom, valeur).

Un tableau JSON est modélisé par la classe `JSONArray`. Il s'agit en fait d'un tableau d'instances de *JSONObject*.

Un objet JSON est analysé grâce à la classe `JSONTokener`.

Une donnée JSON est créée en appelant la méthode *toString()* sur une instance de type `JSONObject`.

ETAPE 1 : Créer un message de requête au format JSON

L'API JSON permet de créer une donnée au format JSON. Elle garantit une syntaxe correcte.

Pour convertir des données au format JSON, on crée d'abord une instance de JSONObject, on ajoute ensuite des données à cette instance (en fait à son tableau qui correspond à une *map*), et on appelle pour finir la méthode *toString()* sur l'instance.

Exemple – Obtenir le texte JSON correspond au premier objet du tableau de l'exemple 1

```
JSONObject ligneCommande1 = new JSONObject();
ligneCommande1.put(" produit ", "ramette papier");
ligneCommande1.put("quantite", 10);
ligneCommande1.put("prix", 4.5);

// pour convertir en chaîne de caractères
String json = ligneCommande.toString() ;
```

Exemple – Obtenir le texte JSON correspond au tableau de l'exemple 1

```
try {
    JSONArray tableauCommande = new JSONArray();
    JSONObject ligneCommande1 = new JSONObject();
    ligneCommande1.put(" produit ", "ramette papier");
    ligneCommande1.put("quantite", 10);
    ligneCommande1.put("prix", 4.5);

    JSONObject ligneCommande2 = new JSONObject();
    ligneCommande2.put(" produit ", "classeur");
    ligneCommande2.put("quantite", 20);
    ligneCommande2.put("prix", 2.3);

    tableauCommande.put(ligneCommande1);
    tableauCommande.put(ligneCommande2);
    String texteFormatJSON = tableauCommande.toString();
    . . .
} catch(JSONException erreur) {
    . . .
}
```

ETAPE 2 : Envoyer le message au service et obtenir la réponse

Il faut indiquer au client que le message envoyé est de type POST.

ETAPE 3 : Analyser la réponse au format JSON

Pour ce faire, nous procédons par étape :

1) Encapsuler les données de la réponse dans une instance de type *JSONTokener*
2) Récupérer les instances d'objets (de type *JSONObject*) ou de tableau d'objets (de type *JSONArray*) en appelant la méthode *nextValue()*. Cette méthode retourne *null* si il n'y a plus de données à analyser
3) Si la réponse obtenue est un tableau de type *JSONArray*, il faut parcourir le tableau pour accéder à chacun de ses éléments qui est de type *JSONObject*
4) Extraire les données de l'objet de type *JSONObject* en invoquant les méthodes :
   XXX getXXX(String cle) où *XXX* est l'un des types dde base du langage Java
   et *clé* est la clé de l'information à récupérer
5) Il faut penser aussi à gérer l'exception *JSONException* levée dans le cas où une donnée serait mal formatée

Exemple

Dans l'application qui consulte la base de données des films, on souhaite afficher comme résulat : le titre du film, l'année de sortie et la liste des acteurs.

On écrit donc la méthode *onPostExecute* de la manière ci-dessous.


```
@Override
  protected void onPostExecute(StringBuilder resultat) {
      if (resultat.length() == 0) {

        // on n'a pas pu récupérer la réponse
        Toast.makeText(activiteParente,

             activiteParente.getResources().getString(R.string.message_erreur),
             Toast.LENGTH_LONG).show();
      } else {

        // On extrait du résultat les informations pertinentes, avant de les afficher

        JSONTokener tokenJSON = new JSONTokener(resultat.toString());
        try {
            JSONObject objetJSON = (JSONObject) tokenJSON.nextValue();


            StringBuilder resultatFormate = new StringBuilder();
            resultatFormate.append("Titre : ");
            resultatFormate.append(objetJSON.getString("Title"));


            resultatFormate.append("\nAnnée de sortie : ");
            resultatFormate.append(objetJSON.getString("Year"));
```

```
            resultatFormate.append("\nActeurs : ");
            resultatFormate.append(objetJSON.getString("Actors"));


            activiteParente.setZoneResultat(resultatFormate.toString());
        } catch(JSONException erreur) {
            Log.i(TAG_LOG, "Problème lors de l'analyse JSON");
        }

}
```