1- Rappel :

Les bases de données

De nombreuses applications manipulent des données et il est souvent nécessaire de les stocker dans des bases de données. Ces bases de données peuvent être exploités par des requêtes SQL.

Notions de base

Une base de données (*database*) est un « conteneur » permettant de stocker et de retrouver l'intégralité des données brutes ou d'informations. Dans la très grande majorité des cas, ces informations sont très structurées, et la base est localisée dans un même lieu et sur un même support.

Le dispositif comporte un système de gestion de base de données (SGBD) : un logiciel moteur qui manipule la base de données et dirige l'accès à son contenu. De tels dispositifs comportent également des logiciels applicatifs, et un ensemble de règles relatives à l'accès et l'utilisation des informations.

Une base de données relationnelle est une base de données où l'information est organisée dans des tableaux à deux dimensions appelés des relations ou tables. Les lignes de ces relations sont appelées des *nuplets* (tuples) ou enregistrements. Les noms des colonnes (ou champs) sont appelés des attributs.

Les logiciels qui permettent de créer, utiliser et maintenir des bases de données relationnelles sont des systèmes de gestion de base de données relationnels (SGBDR).

Pratiquement tous les systèmes relationnels utilisent le langage SQL (*Structured Query Language*) pour interroger les bases de données. Ce langage permet de rechercher, d'ajouter, de modifier ou de supprimer des données dans les bases de données relationnelles.

Terminologie

Modèle de données

Le schéma ou modèle de données est la description de l'organisation des données. Il renseigne sur les caractéristiques de chaque type de donnée et les relations entre les différentes données qui se trouvent dans la base de données. Il existe plusieurs types de modèles de données (relationnel, entité-association, objet, hiérarchique et réseau).

Entité

Une entité est un objet, un sujet, une notion en rapport avec le domaine d'activité pour lequel la base de données est utilisée, et concernant celui pour lequel des données sont enregistrées (exemple : des personnes, des produits, des commandes, des réservations, …).

Attribut

Un attribut est une caractéristique d'une entité susceptible d'être enregistrée dans la base de données. Par exemple une personne (entité), son nom et son adresse (des attributs). Les attributs sont également appelés des champs ou des colonnes.

Enregistrement

Un enregistrement est une donnée qui comporte plusieurs champs dans chacun desquels est enregistrée une donnée.

Association

Les associations désignent les liens qui existent entre différentes entités, par exemple entre un vendeur, un client et un magasin.

Cardinalité

La cardinalité d'une association (d'un lien entre deux entités A et B par exemple) est le nombre de A pour lesquelles il existe un B et inversement. Celle-ci peut être un-a-un, un-a-plusieurs ou plusieurs-à-plusieurs. Par exemple un compte bancaire appartient à un seul client, et un client peut avoir plusieurs comptes bancaires (cardinalité un-a-plusieurs).

Modèle de données relationnel

C'est le type de modèle de données le plus couramment utilisé pour la réalisation d'une base de données. Selon ce type de modèle, la base de données est composée d'un ensemble de tables (les relations) dans lesquelles sont placées les données ainsi que les liens. Chaque ligne d'une table est un enregistrement.

Base de données relationnelle

C'est une base de données organisée selon un modèle de données de type relationnel, à l'aide d'un SGBD permettant ce type de modèle.

Clé primaire

Dans les modèles de données relationnels, la clé primaire est un attribut dont le contenu est différent pour chaque enregistrement de la table, ce qui permet de retrouver un et un seul enregistrement (unicité). Dans les modèles de données relationnels, une clé étrangère est un attribut qui contient une référence à une donnée connexe (la valeur de la clé primaire de la donnée connexe).

SQL (*Structured Query Language*)

C'est un langage de requête structurée et normalisé servant à exploiter des bases de données relationnelles. La partie langage de manipulation des données de SQL permet de rechercher, d'ajouter, de modifier ou de supprimer des données dans les bases de données relationnelles.

Exemples SQL

Recherche des lignes (aussi appelés tuples) dans une table existante :

```
SELECT nom, service
FROM   employe
WHERE  statut = 'stagiaire'
ORDER  BY nom;
```

Insère une ligne (aussi appelés tuple) dans une table existante :

```
INSERT INTO employe (nom, service, statut)
VALUES ('toto', 'rd', 'developpeur');
```

Modifie un tuple existant dans une table :

```
UPDATE employe
SET statut = 'chef'
WHERE nom = 'toto';
```

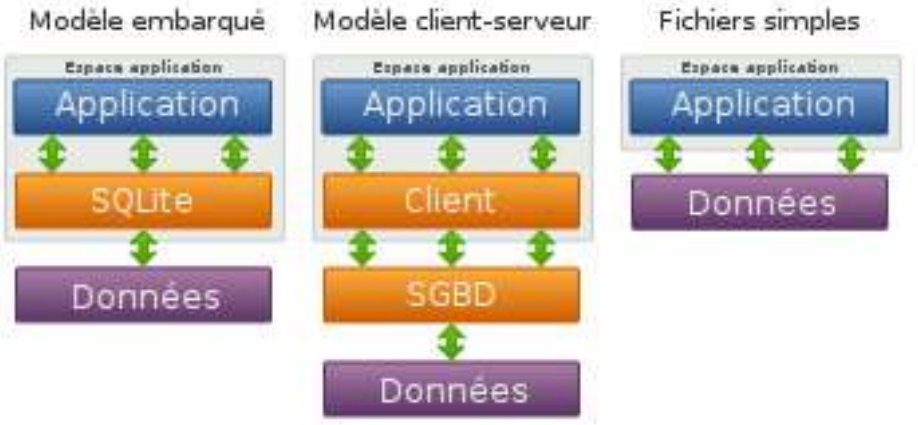
Supprime un tuple existant dans une table :

```
DELETE FROM employe
WHERE nom = 'toto';
```

SQLite

SQLite est une bibliothèque écrite en C qui propose un moteur de base de données relationnelle accessible par le langage SQL. Contrairement aux serveurs de bases de données traditionnels, comme MySQL ou PostgreSQL, sa particularité est de ne pas reproduire le schéma habituel client-serveur mais d'être directement intégrée aux programmes. L'intégralité de la base de données (déclarations, tables, index et données) est stockée dans un fichier indépendant de la plateforme. SQLite est le moteur de base de données le plus distribué au monde, grâce à son utilisation dans de nombreux logiciels grand public comme Firefox, Skype, Google Gears, dans certains produits d'Apple, d'Adobe et de McAfee et dans les bibliothèques standards de nombreux langages comme PHP ou Python. De par son extrême légèreté (moins

de 300 Kio), il est également très populaire sur les systèmes embarqués, notamment sur la plupart des smartphones modernes : l'iPhone ainsi que les systèmes d'exploitation mobiles Symbian et Android l'utilisent comme base de données embarquée.



SQLite est intégrée dans chaque appareil Android.

Remarque : L'accès à une base de données SQLite implique l'accès au système de fichiers. Cela peut être lent. Par conséquent, il est recommandé d'effectuer les opérations de base de données de manière asynchrone.

Si l'application crée une base de données, celle-ci est par défaut est enregistrée dans le répertoire :

`/data/APP_NAME/databases/DATABASE_NAME`.

Le *package* `android.database` contient toutes les classes nécessaires pour travailler avec des bases de données. Le *package* `android.database.sqlite` contient les classes spécifiques à SQLite.

Exercice 1 :

L'objectif de cet exercice est de découvrir l'utilisation de SQLite par le biais d'une petite application de prise de notes. **La prise de notes désigne la transcription écrite résumée du langage parlé. Elle est particulièrement utilisée en cours au niveau de l'enseignement secondaire et des études supérieures.**
L'application permettra d'ajouter les notes saisies dans un éditeur aux listes de notes affichée au-dessous, de la plus récente à la plus ancienne.

Mise en page de l'application

Dans un nouveau projet, créez une activité vide et modifiez la mise en page pour avoir un EditText avec en dessous un ListView. Le plus rapide est de changer le Layout de RelativeLayout en LinearLayout (en orientation verticale). Pensez à ajuster les dimensions horizontale (mettre MatchParent), et à donner tout le poids au ListView.

Il faut ensuite ajouter quelques paramètres (en mode graphique ou texte) sur l'EditText:

```
android:inputType="text"
android:imeOptions="actionDone"
```

Le premier paramètre sert à indiquer le type de contenu. Il est important pour qu'Android choisisse un clavier virtuel adapté à la saisie (on n'utilise pas le même pour un numéro de téléphone et du texte). Le deuxième paramètre indique ce qui sera affiché sur le bouton pour valider la saisie.

Création de la classe chargée de gérer la base de données

La classe SQLiteOpenHelper dispose de quelques fonctions utiles pour gérer une base de données. Nous allons faire hériter la classe chargée de notre base de cette classe. Pour cela, créez une nouvelle classe appelée DbHelper. Cette classe comportera la définition de notre table, ainsi que les méthodes permettant d'accéder à la base.

Définition de la base

Pour chaque table, on créera une classe interne à DbHelper (c'est à dire une classe à l'intérieur de notre classe). Le début de notre fichier va ressembler à ça:

```
package com.hexagonalgames.blocnote;

import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.provider.BaseColumns;

/**
 *
 */
```

```
public class DbHelper extends SQLiteOpenHelper {

    public static final int DATABASE_VERSION = 1;
    public static final String DATABASE_NAME = "List.db";

    // Définition des tables

    public static abstract class Note implements BaseColumns {
        public static final String TABLE_NAME = "note";
        public static final String COLUMN_NAME_TEXT = "text";
        public static final String COLUMN_NAME_DATE = "date";
    }
```

On trouve ici la définition de deux constantes (le numéro de version de notre base, et le nom du fichier à utiliser), puis la classe interne Note. Dans la classe note, on définit des constantes pour le nom de la table et pour chaque champ. Le fait de définir des constantes permet d'éviter les fautes de frappes : si on utilise un littéral (une valeur entre « »), il faut être certain de toujours écrire exactement les noms de la même manière. Si on définit une constante, Android studio complète automatiquement et nous prévient en cas d'erreur. C'est donc plus sûr.

Définition des requêtes

Juste après avoir définit les tables, il faut écrire les requêtes pour la création et le vidage de la base. Le code est le suivant:

```
    // Définition des requètes en SQLite

    private static final String TEXT_TYPE = " TEXT";
    private static final String DATE_TYPE = " INT";
    private static final String COMMA_SEP = ",";

    private static final String SQL_CREATE_ENTRIES =
            "CREATE TABLE " + Note.TABLE_NAME + " (" +
                    Note._ID + " INTEGER PRIMARY KEY" + COMMA_SEP +
                    Note.COLUMN_NAME_TEXT+ TEXT_TYPE + COMMA_SEP +
                    Note.COLUMN_NAME_DATE+ DATE_TYPE +
            " )";

    private static final String SQL_DELETE_ENTRIES =
            "DROP TABLE IF EXISTS " + Note.TABLE_NAME;
```

Quelques points importants à noter :

- SQLite dispose de peu de types (principalement int et text), il faut faire avec
- Encore une fois, on utilise des constantes pour les textes. Cela permet entre autres de propager automatiquement les changements de nom de champs/table sans risque d'oubli.

Fonctionnalités minimales

Notre classe héritant SQLiteOpenHelper, il faut définir un certain nombre de méthodes. A part le constructeur, ces méthodes sont normalement créées automatiquement, il faut juste les remplir. Elles servent à définir le comportement à tenir lors de l'évolution de la base (création, passage à une nouvelle version). L'approche choisie ici est

minimaliste : si on change de version, on détruit la base avant de la recréer. A noter que c'est dans ce code que les requêtes que l'on vient de définir vont être utilisées.

```
 public DbHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(SQL_CREATE_ENTRIES);
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int
newVersion) {
        db.execSQL(SQL_DELETE_ENTRIES);
        onCreate(db);
    }

    @Override
    public void onDowngrade(SQLiteDatabase db, int oldVersion, int
newVersion) {
        onUpgrade(db, oldVersion, newVersion);
    }
```

Accès à la base

Il faut enfin écrire le code permettant d'ajouter une ligne, ou de consulter les lignes disponibles.

```
    /***
     * Retourne la liste actuelle
     * @return la liste actuelle
     */
    public Cursor getList() {
        String[] projection = {
                Note._ID,
                Note.COLUMN_NAME_TEXT,
                Note.COLUMN_NAME_DATE};
        return this.getReadableDatabase().query(Note.TABLE_NAME,
projection, null, null, null, null, Note.COLUMN_NAME_DATE+" DESC");
    }

    /***
     * Ajoute une entrée dans la table
     * @param text
     * @return
     */
    public long insertRow(String text){
        SQLiteDatabase db = getWritableDatabase();

// Create a new map of values, where column names are the keys
        ContentValues values = new ContentValues();
        values.put(Note.COLUMN_NAME_TEXT, text);
        values.put(Note.COLUMN_NAME_DATE, System.currentTimeMillis());

// Insert the new row, returning the primary key value of the new row
        return db.insert(
                Note.TABLE_NAME,
```

```
                    null,
                    values);
        }
    }
```

Il est possible d'écrire les requêtes SELECT et INSERT directement sous forme de chaînes de caractère (comme cela a été fait pour le TABLE CREATE et le DROP TABLE), mais les méthodes query() et insert() sont plus efficaces. N'hésitez pas à consulter la documentation pour voir les paramètres possibles.

SQLiteDatabase (Javadoc)

Le fichier DbHelper.java étant maintenant complet, il ne doit plus générer d'erreurs.

Configuration des composants

Pour que notre application fonctionne, il faut maintenant compléter notre activité afin de réaliser les tâches suivantes :

- Gérer la saisie d'un nouveau texte. Le texte doit être ajouté dans notre base
- Configurer le ListView pour afficher le contenu de la base

Comme d'habitude, l'essentiel du code à ajouter se trouve dans le onCreate(), après le setContentView

Configuration du ListView

La méthode correspond à ce que l'on a déjà fait avec d'autres ListView (ou GridView): On récupère le composant, on crée un adaptateur, et on configure le composant pour qu'il l'utilise. Il faut, comme on l'a souvent fait, créer une mise en page pour les éléments. Créez donc un fichier de layout pour les éléments (par exemple layout/element.xml) contenant un TextVIew (on suppose que son id sera @id/textView). On ajoute le code java suivant:

```
        ListView listView  = (ListView) findViewById(R.id.listView);

        adapter = new SimpleCursorAdapter(this, R.layout.element,
                null,
                new String[]{
                        DbHelper.Note.COLUMN_NAME_TEXT},
                new int[]{
                        R.id.textView
                }, 0);

        listView.setAdapter(adapter);
```

Normalement, Android studio vous indique qu'adapter n'est pas déclaré. Corrigez l'erreur en demander à créer le champ correspondant.
il faut ensuite charger le contenu actuel de la base. Pour cela, il nous faut une instance de la classe que l'on vient de créer, DbHelper, et s'en servir pour faire notre SELECT. Voici le code :

```
        db = new DbHelper(this);
        adapter.changeCursor(db.getList());
```

Encore une fois, corrigez l'erreur en faisant créer le champ (ici db). Ces deux champs nous seront utiles dans une autre méthode, pour mettre à jour la base quand on aura du texte à ajouter.

Configuration de l'EditText

Pour écouter les évènements liés à la saisie du texte, on ajoute le code suivant :

```
EditText editText = (EditText) findViewById(R.id.editText);
editText.setOnEditorActionListener(this;
```

Cela provoque une erreur, notre activité (this) ne gérant pas encore ces évènements. Comme on l'a déjà fait dans plusieurs TP, on utilise la complétion automatique pour implémenter l'interface adaptée (alt-entrée suivi de implements…)

On complète ensuite la nouvelle méthode :

```
    @Override
    public boolean onEditorAction(TextView textView, int i, KeyEvent
keyEvent) {
        //On récupère le texte
        final String text = textView.getText().toString();

        //On vide l'editText
        textView.setText(null);

        //On ajoute le texte à la base avec un INSERT
        db.insertRow(text);

        //On réexécute le SELECT
        Cursor cursor = db.getList();
        cursor.moveToFirst();

        //On fait afficher ce nouveau résultat
        adapter.changeCursor(cursor);

        //On indique que l'évènement a bien été géré
        return true;
    }
```

Travail à réaliser

Rajouter les fonctionnalités suivantes à ce qui a été fait plus haut :

- Appels à la base de données en tache de fond.
- Suppression d'un élément. On peut ajouter un écouteur pour les clics longs sur les éléments de la liste, et demander à supprimer l'élément en question. La difficulté est de récupérer l'id de la ligne à effacer (il faut passer par l'adapter pour récupérer le cursor, se positionner sur ligne en question, afin d'obtenir l'id…).

L'ajout de la méthode de suppression d'une ligne par son id dans DbHelper est la partie la plus simple.

- L'affichage de la date. Il faut un composant texte en plus pour les éléments, mais surtout créer votre propre adapter

Exercice 2 :

On veut, dans cet exercice, construire une application Android qui permet de gérer des contacts (essentiellement un nom associé à un numéro de téléphone). Ces contacts seront mis dans une base de données Android gérée par SQLite.

Construction de la base de données

1°) Définir une classe Contact contenant les informations :

```
public class Contact {

	private int _id;

	private String nom;

	private String numTelephone;

	...
}
```

Compléter cette classe.

2°) Construire un "Database Helper" permettant de gérer une base de données. Le début de cette classe est :

```
public class LeDatabaseHandler extends SQLiteOpenHelper {
	// All Static variables
	// Database Version
	private static final int DATABASE_VERSION = 1;
	// Database Name
	private static final String DATABASE_NAME =
	"contactsManager";
	// Contacts table name
	private static final String TABLE_CONTACTS = "contacts";
	// Contacts Table Columns names
	private static final String KEY_ID = "id";
	private static final String KEY_NAME = "name";
	private static final String KEY_PH_NO = "phone_number";
	// A compléter ...
}
```

3°) Compléter cette classe de sorte à :

a) créer la base de données

b) pouvoir insérer des Contacts dans cette base

c) récupérer tous les contacts de la base à l'aide de la méthode :

```
public List<Contact> getAllContacts() { ... }
```

4°) Manipuler cette base à l'aide d'une activité principale possédant la méthode :

```
public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      LeDatabaseHandler db = new LeDatabaseHandler(this);
      /*
      * Operation CRUD
      */
      Log.d("Salam", "Insertion de Contact");
      db.addContact(new Contact("Ali", "0672852402"));
      db.addContact(new Contact("Idris", "06756117657"));
      db.addContact(new Contact("Wahid", "0669408375"));
      db.addContact(new Contact("Ahmed", "0661370401"));
      // Reading all contacts
      Log.d("Salam", "Lecture des Contacts");
      List<Contact> contacts = db.getAllContacts();
      for (Contact cn : contacts) {
      String log = "Id: "+cn.getID()+" ,Name: " +
      cn.getName() + " ,Phone: " + cn.getPhoneNumber();
      // Writing Contacts to log
      Log.d("Salam", log);
      }
}
```

Cette partie est grandement inspiré de l'article de Ravi Tamada à
http://www.androidhive.info/2011/11/android-sqlite-database-tutorial/
Voici une partie additionnelle permettant de mieux manipuler la base de données.
Une IHM pour manipuler une base de données

5°) Définir une activity qui fait afficher :

il s'agit d'une TextView ("Voulez-vous ?") suivi d'une ListView à 4 items.

6°) Ajouter le code de sorte que lorsque l'utilisateur clique sur l'item "Initialisation de la base !", la base (table) est recréée avec 4 Contacts. On rappelle que le code de gestion d'une ListView est :

```
lv.setOnItemClickListener(new OnItemClickListener() {

      public void onItemClick(AdapterView<?> parent, View view,

            int position, long id) {

          if (position == 0) {

          // traitement si l'utilisateur a choisi le premier
          item de la ListView

          } else { ...

          }

      }

});
```

où lv est la ListView.

7°) De même, écrivez le code qui affiche tous les contacts lorsque l'utilisateur a choisi le premier item. On pourra utiliser un TableLayout.

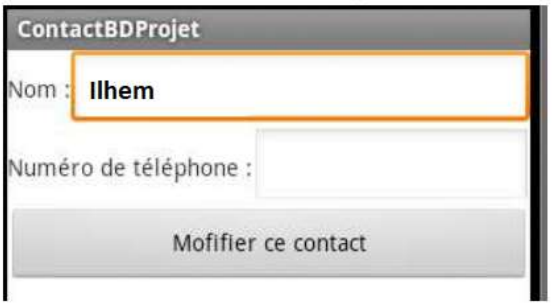
8°) Ecrire une activity qui affiche :

Cette activité est lancée lorsque l'utilisateur sélectionne l'item "Ajouter un contact".

Ecrire le code permettant d'ajouter un Contact dans la base de données grâce à cette interface.

9°) Ecrire l'activité qui présente tous les contacts dans une ListView pour pouvoir en modifier. L'enchaînement des activités doit être :

Par la suite, lorsque l'utilisateur choisit un des contacts, une nouvelle activité est affichée, initialisée par le nom du Contact à modifier. L'enchaînement des activités peut être :



Il faudra donc passer le Contact d'une activité à une autre. Pour cela on utilisera le code (peut-être à adapter) :

```
Contact ctAModifier = arContacts.get(position) ;

Intent i = new Intent(getApplicationContext(),
ContactAModifierActivity.class);

Bundle b = new Bundle();

b.putSerializable(Constants.CONTACT, ctAModifier) ;

i.putExtras(b) ;

startActivity(i);
```

Pour récupérer ce Contact dans l'activité destinataire on peut écrire le code :

```
Bundle b = this.getIntent().getExtras();

if (b != null) {

ct = (Contact)(b.getSerializable(Constants.CONTACT));

}
```

10°) Enrichir l'activité qui présente tous les contacts dans une ListView pour pouvoir en détruire le contact choisi par l'utilisateur. On pourra construire de nouvelles activités ou enrichir celles déjà développées.

Documentation

- http://tvaira.free.fr/dev/android/android-3.html
- Guide de référence d'Android SDK
- Cours et tutoriels pour Android
- FAQ Android
- Utilisation de base de données SQLite sous Android
- https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=2ahUKEwjXnbDWw6npAhVJBWMBHR07A0wQFjAAegQIAhAB&url=http%3A%2F%2Fcedric.cnam.fr%2F~farinone%2FNFA025%2FenonceTPSQLite.pdf&usg=AOvVaw2B9zEqoabnsT84XS3RAqeC