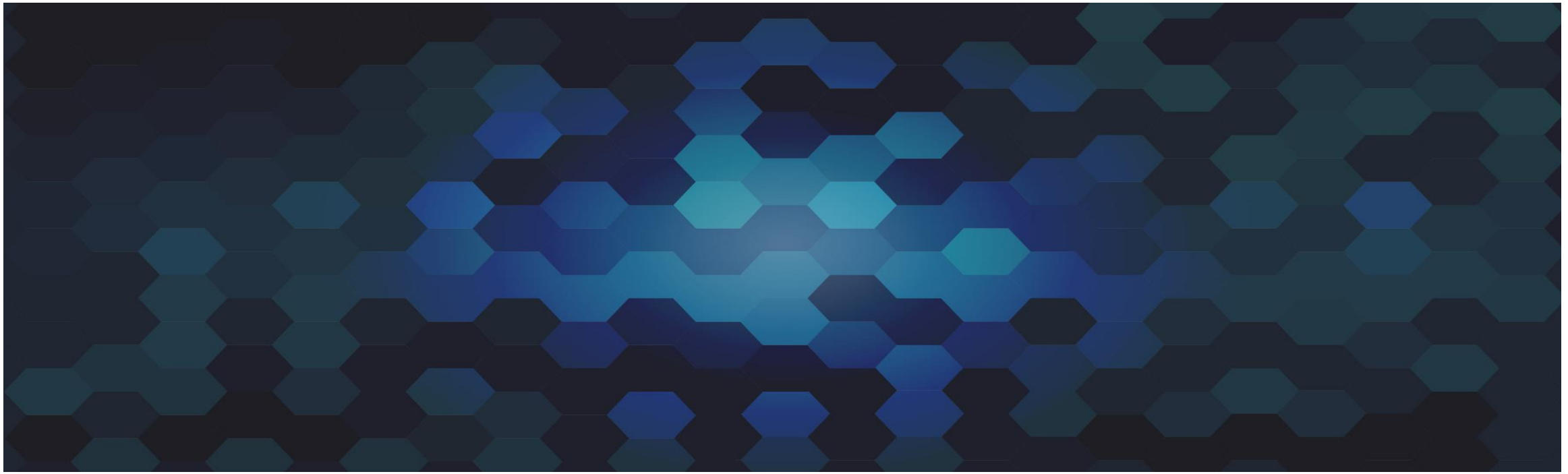
Développement mobile pour Android

Bases de Données SQLite

ENSEIGNANTE: MIRNA AWAD

RESPONSABLE DE COURS: A. Toudeft



PLAN

- Introduction aux bases de données (rappels)
- Introduction à SQLite
- Modes d’accès
- Création d’une base de données
- Modification de la structure d’une base de données
- Modification des données
- Consultation des données
- Curseurs
- Adaptateurs de curseurs
- Démonstration de l’exemple du cours
- Exercice.

INTRODUCTION AUX BASES DE DONNÉES

- Une base de données est un ensemble organisé de données structurées, généralement stockées électroniquement dans un système informatique. Elle permet de stocker, gérer et récupérer efficacement des informations.

- Types de Bases de Données
  1. **Bases de données relationnelles** : Utilisent des tables pour stocker des données interconnectées, avec des relations entre elles. Exemple : MySQL, PostgreSQL, SQLite.
  2. **Bases de données NoSQL** : Utilisent différents modèles de données que les bases de données relationnelles. Exemple : MongoDB, Cassandra, Redis.

REQUÊTES SQL

- SQL (Structured Query Language) est un langage de programmation utilisé pour communiquer avec les bases de données relationnelles.

- **Création de Table :**

```
CREATE TABLE table_name (
    column1 datatype PRIMARY KEY,
    column2 datatype,
    column3 datatype,
    ...
);
```

REQUÊTES SQL

- **Suppression de Table :**

```
DROP TABLE table_name;
```

- **Modification de table :**

```
ALTER TABLE table_name
ADD column_name datatype;
```

```
ALTER TABLE table_name
DROP COLUMN column_name;
```

```
ALTER TABLE table_name
MODIFY COLUMN column_name new_datatype;
```

REQUÊTES SQL

- **Insertion de Données :**

```sql
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

- **Mise à Jour de Données :**

```sql
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

REQUÊTES SQL

- **Suppression de Données :**

```
DELETE FROM table_name WHERE condition;
```

- **Sélection de Données**

SELECT : Définit les colonnes que vous souhaitez récupérer.

FROM : Spécifie la table à partir de laquelle vous voulez récupérer les données.

```
SELECT column1, column2 FROM table_name;
SELECT * FROM table_name;
```

REQUÊTES SQL

- **Sélection de Données**

WHERE : Filtre les lignes basées sur une condition spécifiée. Vous pouvez utiliser des opérateurs logiques tels que =, !=, <, >, AND, OR, etc., pour définir des conditions de filtrage.

```sql
-- Sélectionner des données avec une condition
SELECT * FROM table_name WHERE condition;
```

```sql
SELECT * FROM table_name WHERE column1 = 'value';
SELECT * FROM table_name WHERE column1 > 10 AND column2 = 'value';
```

REQUÊTES SQL

- **Sélection de Données**

GROUP BY : Regroupe les lignes ayant la même valeur dans une ou plusieurs colonnes. Elle est souvent utilisée avec des fonctions d'agrégation telles que COUNT, SUM, AVG, MIN, MAX, etc., pour effectuer des calculs sur les données regroupées.

```sql
SELECT column1, COUNT(*) FROM table_name GROUP BY column1;
SELECT column1, AVG(column2) FROM table_name GROUP BY column1;
```

REQUÊTES SQL

- **Sélection de Données**

ORDER BY : Trie les résultats selon une ou plusieurs colonnes.

```sql
-- Sélectionner des données triées
SELECT * FROM table_name ORDER BY column1;
```

INTRODUCTION À SQLITE

SQLite : Système de gestion de base de données relationnelle léger et rapide;

Caractéristiques importantes :

- Pas de serveur à administrer.
- Aucune configuration nécessaire.
- Toutes les données stockées dans un seul fichier.
- Open source, ce qui permet une personnalisation et une amélioration aisées.

Avantages dans le développement Android :

- Intégré à la plateforme Android, aucune installation supplémentaire requise.
- Stockage de données local pour un accès rapide, même hors ligne.

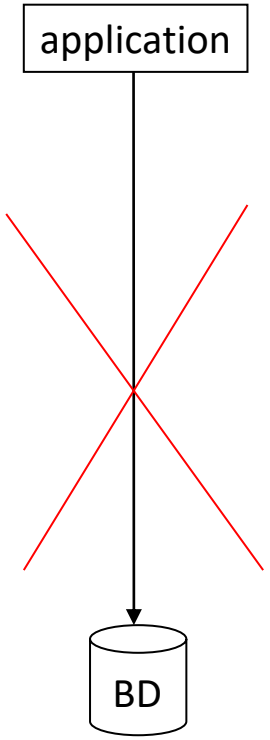
MODES D'ACCÈS

- Une application Androïd peut accéder directement à une base de données SQLite. Cependant, ce n'est pas une bonne pratique.

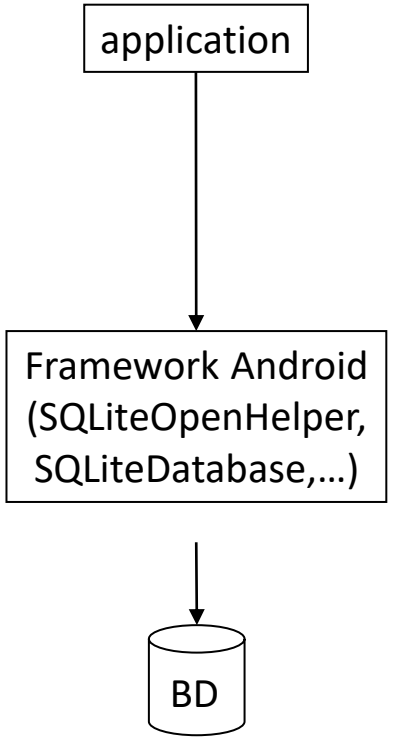
- L'application peut accéder à la base de données en utilisant les services de **classes utilitaires** du framework d'Androïd (**SQLiteOpenHelper**, **SQLiteDatabase**,…). Cette approche est acceptable lorsque la base de données est utilisée par une seule application (base de données interne à l'application).

- Finalement, l'application peut accéder à la base de données **en passant par un fournisseur de contenu** : le fournisseur de contenu fournit une interface standard et gère les accès à la base de données en passant par les classes utilitaires (SQLiteOpenHelper, SQLiteDatabase,…). Cette approche est à privilégier lorsque la base de données est appelée à être accédée par plusieurs applications.

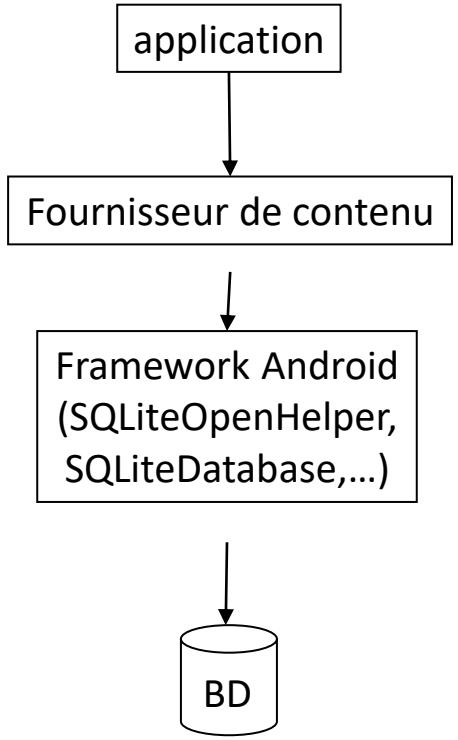
MODES D'ACCÈS



Accès direct
(déconseillé)



Accès par classes utilitaires
(BD propre à une
application)



Accès par fournisseur de contenu
(BD partagée par plusieurs
applications)

CRÉATION DE BASES DE DONNÉES

- Habituellement, on définit la structure de la base de données dans un « contrat ».
- Un contrat est généralement une classe Java qui contient des constantes statiques pour définir le nom de la table, le nom des colonnes et d'autres éléments liés à la base de données.
- Exemple :

```
public class TauxContract {
    public static final String DB_NAME=“MONNAIES.DB";
    public static final int DB_VERSION=1;
    public static final String TABLE_NAME="taux";

    public class Colonnes {
        public static final String ID= BaseColumns._ID;
        public static final String MONNAIE= "MONNAIE";
        public static final String VALEUR= "VALEUR";
    }
}
```

- Le numéro de version (**DB_VERSION**) est important pour la mise à jour de la structure de la base de données.

CRÉATION DE BASES DE DONNÉES

BaseColumns est une interface fournie par Android qui définit automatiquement les constantes _ID et _COUNT

- La constante **_ID** est généralement utilisée pour représenter la clé primaire de la table.
- La constante **_COUNT** peut être utilisée pour obtenir le nombre total de lignes dans un curseur.

CRÉATION DE BASES DE DONNÉES

- On utilise la classe utilitaire SQLiteOpenHelper.
- SQLiteOpenHelper est abstraite. On l'étend pour fournir un constructeur public et implémenter les 2 méthodes abstraites onCreate() et onUpgrade() :
  - **onCreate()** permet de créer un nouvelle base de données;
  - **onUpgrade()** permet de modifier la structure de la base de données (ajout de tables, ajout/modification de colonnes,...)

```
public class DbUtil extends SQLiteOpenHelper {
    public DbUtil(Context context) {
          //...
    }
    @Override
    public void onCreate(SQLiteDatabase db) {
          //...
    }
    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        //...
    }
}
```

CRÉATION DE BASES DE DONNÉES

- Le constructeur fait appel au constructeur de la classe mère :

```
public class DbUtil extends SQLiteOpenHelper {
    public DbUtil(Context context) {
        super(context, TauxContract.DB_NAME, null, TauxContract.DB_VERSION);
    }
```

- Le constructeur s'occupe de créer la BD, si elle n'existe pas. À la création, il appelle la méthode **onCreate()** pour créer les tables. Si la BD existe déjà et que le **numéro de version a changé**, il appelle la méthode **onUpgrade()**.

CRÉATION DE BASES DE DONNÉES

- La méthode **onCreate()** exécute la requête de la création des tables :
- Exemple :

```
@Override
public void onCreate(SQLiteDatabase db) {
    String requeteCreation = String.format("create table %s (%s
            INTEGER PRIMARY KEY AUTOINCREMENT, %s text, %s double)",
            TauxContract.TABLE_NAME,
            TauxContract.Colonnes.ID,
            TauxContract.Colonnes.MONNAIE,
            TauxContract.Colonnes.VALEUR);
    db.execSQL(requeteCreation);
}
```

MODIFICATION DE LA STRUCTURE D'UNE BASES DE DONNÉES

- La méthode **onUpgrade()** contiendra les instructions de modification de la structure de la base de données :
- Exemple :

```
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    String requeteModification = String.format("alter table %s ADD %s int not null",
                                    TauxContract.TABLE_NAME, "NOUVELLE_COLONNE");
    db.execSQL(requeteModification);
}
```

INSERTION DES DONNÉES

- Les opérations d'**insertion**, de **mise à jour** et de **suppression** sont réalisées à l'aide de méthodes de la classe SQLiteDatabase db = dbUtil.getWritableDatabase();
- Exemple :

```
DbUtil dbUtil = new DbUtil(this); //this est le contexte (exemple: l'activité)
SQLiteDatabase db = dbUtil.getWritableDatabase();
ContentValues donnees = new ContentValues();
donnees.put(TauxContract.Colonnes.ID,1);
donnees.put(TauxContract.Colonnes.MONNAIE,"Euro");
donnees.put(TauxContract.Colonnes.VALEUR,1.6);
db.insert(TauxContract.TABLE_NAME, null, donnees);

//Ou

db.insertWithOnConflict(TauxContract.TABLE_NAME, null, donnees, SQLiteDatabase.CONFLICT_IGNORE);
```

INSERTION DES DONNÉES

- La méthode **insert**():
  - est utilisée pour insérer des données dans une table
  - Si les données que vous essayez d'insérer violent une contrainte d'unicité (comme une contrainte de clé primaire ou d'index unique), une exception **SQLiteConstraintException** sera levée et l'insertion échouera.
  - ne permet pas de spécifier explicitement le comportement en cas de conflit avec des données existantes.

- La méthode **insertWithOnConflict**() offre un contrôle supplémentaire sur la gestion des conflits. Elle permet de spécifier explicitement le comportement à adopter en cas de conflit avec des données existantes.

- Les deux méthodes renvoient l'ID de la ligne nouvellement insérée, ou -1 si une erreur s'est produite

CONSULTATION DES DONNÉES

- Les opérations de consultation de données utilisent des méthodes **query()** de SQLiteDatabase. Ces méthodes reçoivent les critères de consultation et retournent un objet de type **Cursor** qui encapsule les données :

- Exemple :

```
SQLiteDatabase db = dbUtil.getReadableDatabase();
String[] colonnesDesirees = {
                            TauxContract.Colonnes.ID,
                            TauxContract.Colonnes.MONNAIE,
                            TauxContract.Colonnes.VALEUR
                            };
//Méthode 1 (requête sur 1 seule table) :
Cursor curseur = db.query(TauxContract.TABLE_NAME, colonnesDesirees, selection,
                                selectionArgs,groupBy,having,orderBy);
//Méthode 2 :
SQLiteQueryBuilder sqb = new SQLiteQueryBuilder();
sqb.setTables(TauxContract.TABLE_NAME);
Cursor curseur = sqb.query(db, colonnesDesirees, selection,
                                selectionArgs,groupBy,having,orderBy);
```

CURSEURS

- Les curseurs encapsulent un ensemble de données comme des enregistrements (lignes et colonnes) et fournissent les méthodes pour y accéder;
- La tête de lecture du curseur est initialement avant le premier enregistrement et il faut appeler la méthode **moveToFirst()** pour accéder au premier enregistrement;
- **Principales méthodes :**

Navigation :

- moveToFirst()
- moveToLast()
- moveToNext()
- moveToPrevious()
- moveToPosition(int position)

Métadonnées :

- getColumnCount()
- getCount()
- getColumnName(indiceColonne)
- getColumnIndex(nomColonne)
- etc ...

Accès à l'état :

- isAfterLast()
- isBeforeFirst()
- isClosed()
- isNull()
- etc ...

Accès à l'enregistrement courant :

- getString(indiceColonne)
- getInt(indiceColonne)
- getDouble(indiceColonne)
- getBlob(indiceColonne)
- etc ...

ADAPTATEURS DE CURSEURS

- Pratique d'afficher les données d'un curseur dans un composant d'affichage (*ListView*, *Spinner*, …);
- Un adaptateur de curseur alimente le composant d'affichage à partir d'un curseur;
- On peut définir notre adaptateur en étendant la classe abstraite **CursorAdapter**;
- Classe fournie pour les cas courants : **SimpleCursorAdapter**;

```
String[] colonnesDesireesDuCurseur = …;
int[] idComposantsDuLayoutDeLItem = …;

SimpleCursorAdapter sca = new SimpleCursorAdapter(this,R.layout.layout_nom_monnaie,
        curseur,
        colonnesDesireesDuCurseur,
        idComposantsDuLayoutDeLItem,
        CursorAdapter.FLAG_REGISTER_CONTENT_OBSERVER);

unListView.setAdapter(sca);
```

MODIFICATION DES DONNÉES

- Exemple :

```
DbUtil dbUtil = new DbUtil(this); //this est le contexte (exemple: l'activité)
SQLiteDatabase db = dbUtil.getWritableDatabase();

ContentValues donnees = new ContentValues();
donnees.put(TauxContract.Colonnes.MONNAIE, "US Dollar");

String selection = TauxContract.Colonnes.ID + " = ?";
String [] selectionArgs = {"1"};

db.update(TauxContract.TABLE_NAME, donnees, selection, selectionArgs);
```

- update() retourne le nombre de lignes modifiées dans la table.

SUPPRESSION DES DONNÉES

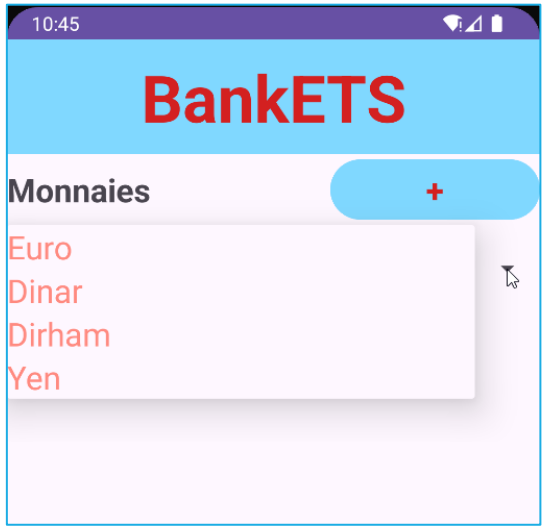
- Exemple :

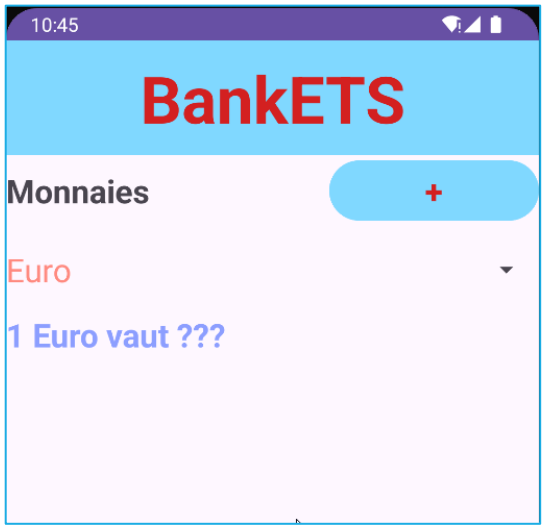
```
DbUtil dbUtil = new DbUtil(this); //this est le contexte (exemple: l'activité)
SQLiteDatabase db = dbUtil.getWritableDatabase();

ContentValues donnees = new ContentValues();
donnees.put(TauxContract.Colonnes.MONNAIE, "US Dollar");

String selection = TauxContract.Colonnes.ID + " = ?";
String [] selectionArgs = {"1"};

db.delete(TauxContract.TABLE_NAME, selection, selectionArgs);
```

- delete() retourne le nombre de lignes supprimées de la table.

EXERCICE DU COURS

- Projet **SQLite_Exemple.** Base de données **MONNAIES.DB** créée à l'installation de l'application;
- Table **taux** contenant des monnaies avec leur valeur en $CAN (initialement vide);
- Écran d'ajout de monnaies;
- Écran d'accueil affichant les monnaies disponibles dans un *Spinner*;

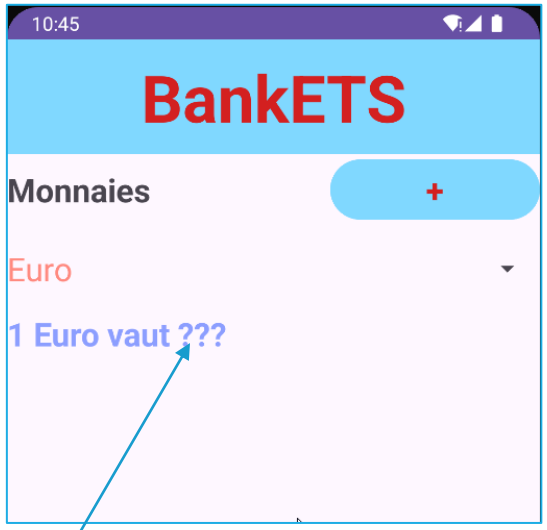




EXERCICE DU COURS

Ajouter au projet **SQLite_Exemple:**

- la fonctionnalité qui récupère la liste des monnaies de la base de données pour les afficher dans le Spinner.
- la fonctionnalité qui récupère la valeur $CAN de la monnaie sélectionnée dans le Spinner. La valeur récupérée est affichée dans le *TextView* de l'écran d'accueil.
- la fonctionnalité qui nous permet d'ajouter une nouvelle monnaie et sa valeur $CAN dans la base de données;



Afficher ici la valeur de 1 euro en $CAN

PROCHAINE SÉANCE

- Sujets divers.