# IN01
# Programmation Android

## 04 – Bases de données

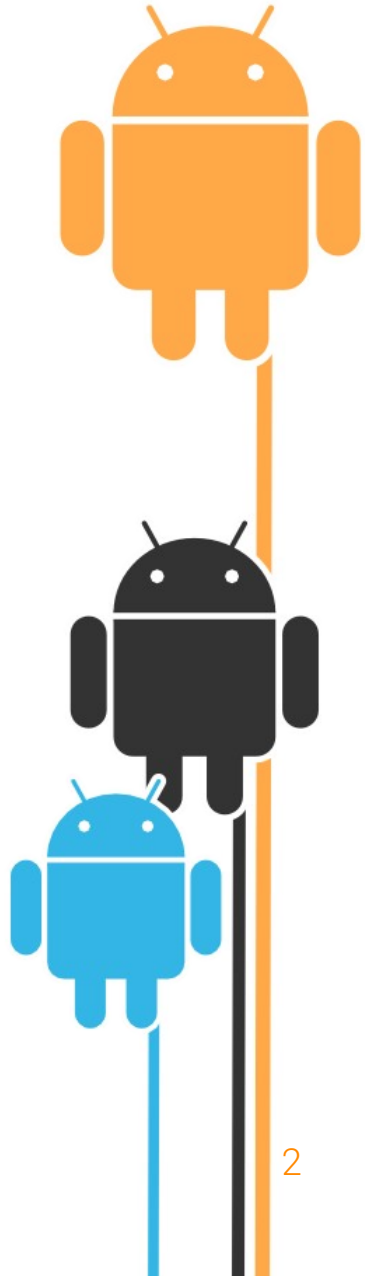

Yann Caron





# Sommaire - Séance 04

- Rappel sur les SGBDs
- Couches logiciels
- SQLite
- Object Relational Mapping
- NoSQL - OODBMS - DB4Object

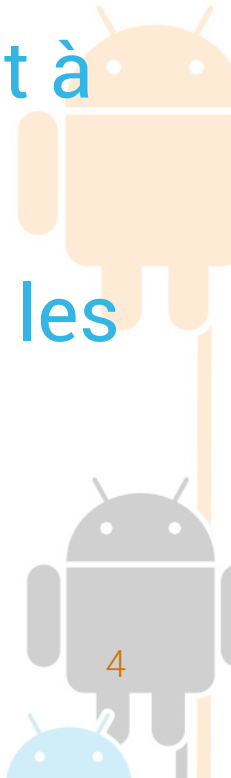


# IN01 – Séance 04

## Rappel sur les SGBDs

# Les bases de données

- Wiki : Une base de données est un magasin de données composé de plusieurs fichiers manipulés exclusivement par le SGBD.
- Ce dernier cache la complexité de manipulation des structures de la base de données en mettant à disposition une vue synthétique du contenu.
- Les SGBD sont les logiciels intermédiaires entre les utilisateurs et les bases de données.

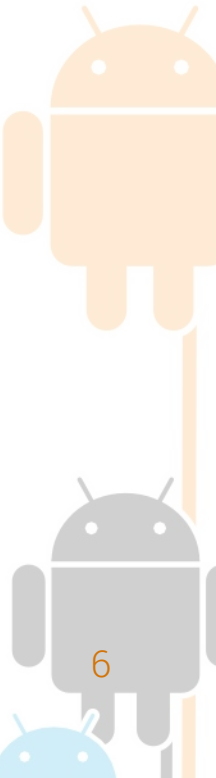


# Exigences d'un SGBD

- Stocker des données, les modifier
- Un processus d'intérrogation et de recherche des données selon un ou plusieurs critères
- Pouvoir créer des sauvegardes et les restaurer si besoin
- Application de règles pour assurer la cohérence des données (selon les relations entre elles)
- Sécurité : un processus d'authentification pour protéger les données

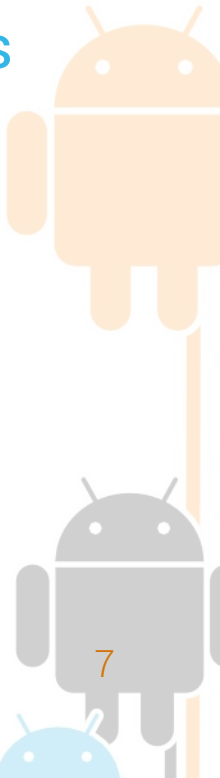
# Exigences d'un SGBD

- Abstraire la couche de stockage et les processus de requêtages
- Logging : track les modifications effectuées par les utilisateurs
- Optimisation : performance tuning



# ACID

- Lors d'une manipulation des données, il faut toujours s'assurer que la base de données conserve son intégrité
- Une transaction (modification bornée) doit être :
  - **Atomique** : la suite des opérations est indivisible (synchronized)
  - **Cohérente** : Le contenu FINAL doit être cohérent, même si les étapes intermédiaires ne le sont pas
  - **Isolée** : Pas d'interaction entre des transactions concurentes
  - **Durable** : Un fois validé, l'état devient permanent et aucun incident technique ne doit alterer le résultat d'une transation.



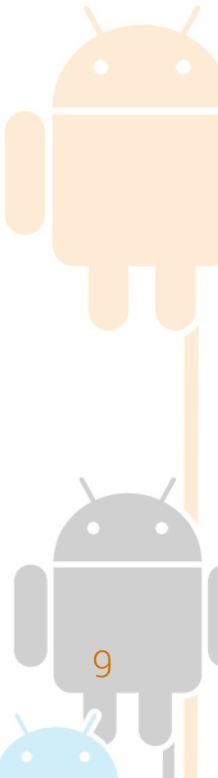
# CRUD

- Opérations de bases dans un SGBD
- Create, Read, Update, Delete

| Operation | SQL | HTTP | DB4O |
|---|---|---|---|
| Create | INSERT | POST | .store |
| Read (Retrieve) | SELECT | GET | .queryByExample<br>.AsQueryable |
| Update | UPDATE | PUT / PATCH | .store |
| Delete | DELETE | DELETE | .delete |

# SGBDR

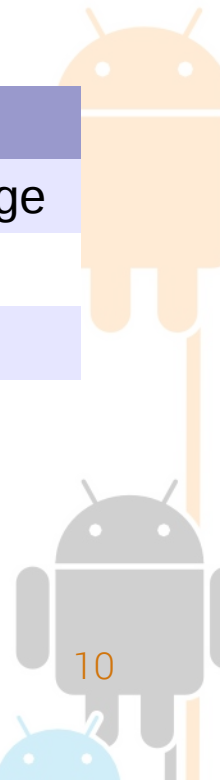
- Systeme de gestion de bases de données Relationnelles
- Relations :
  - One to one
  - One to many
  - Many to many
- Normalisation



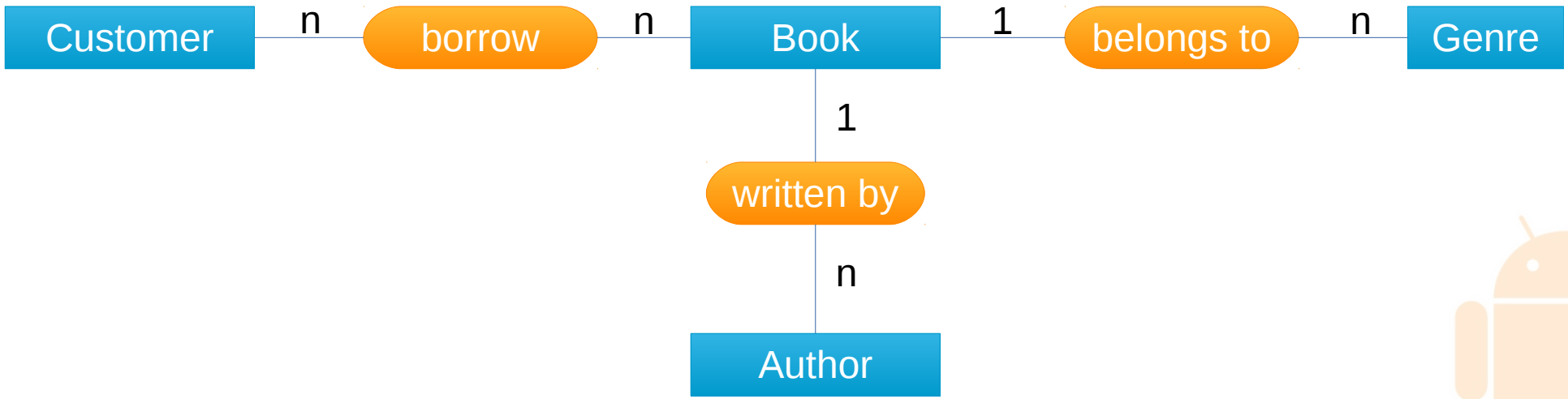
# Bibliothèque

| Book |
| --- |
| title : nvarchar |
| isbn : nvarchar |
| nbPage : int |

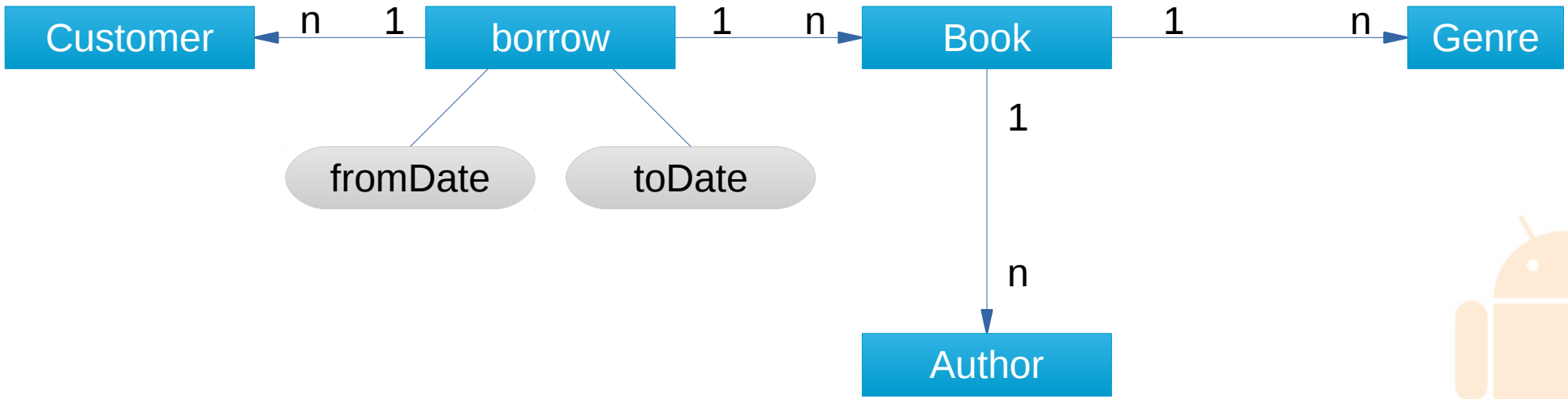
| Book | | |
| --- | --- | --- |
| title | isbn | nbPage |
| Croc blanc | 2010034031 | 248 |
| Vingt mille lieues sous les mers | | 353 |



# Bibliothèque MCD

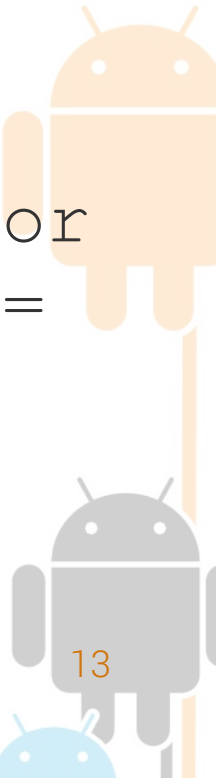


# Bibliothèque MLD



# Queries

- Tous les livres dont le nom commence par la lettre “a”
  - `SELECT * FROM Book WHERE name LIKE 'A%'`
- Tous les livres du même auteur
  - `SELECT * FROM Book b INNER JOIN Author a ON b.idAuthor = a.id WHERE a.name = 'Jack London'`



# Ensembles

- Sous-ensembles résultant de la jointure

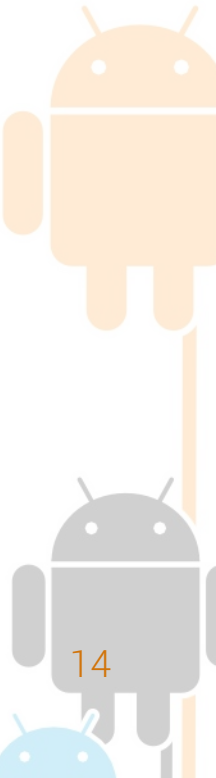
| A |
|---|
| 1 |
| 2 |
| 3 |
| 4 |

FULL JOIN

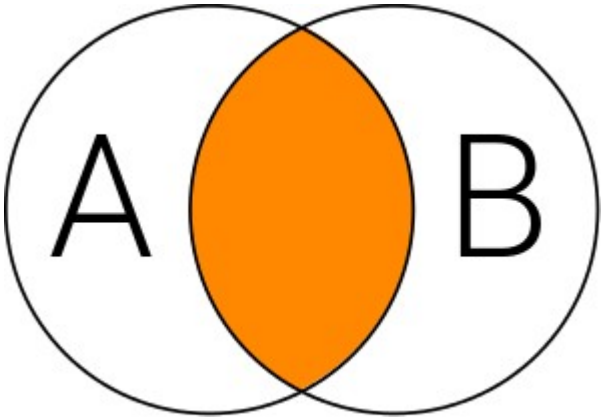
| B |
|---|
| 3 |
| 4 |
| 7 |
| 8 |

=

| A | B |
|---|---|
| 1 | NULL |
| 2 | NULL |
| 3 | 3 |
| 4 | 4 |
| NULL | 7 |
| NULL | 8 |



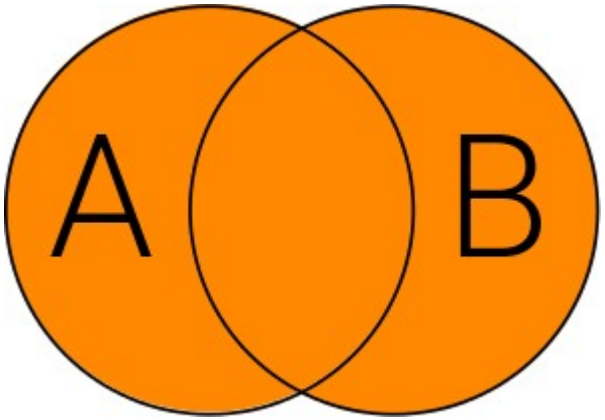
# Ensembles




## Inner Join

```
SELECT * FROM A
INNER JOIN B
ON A.key = B.key
```

| A | B |
|---|---|
| 1 | NULL |
| 2 | NULL |
| 3 | 3 |
| 4 | 4 |
| NULL | 7 |
| NULL | 8 |

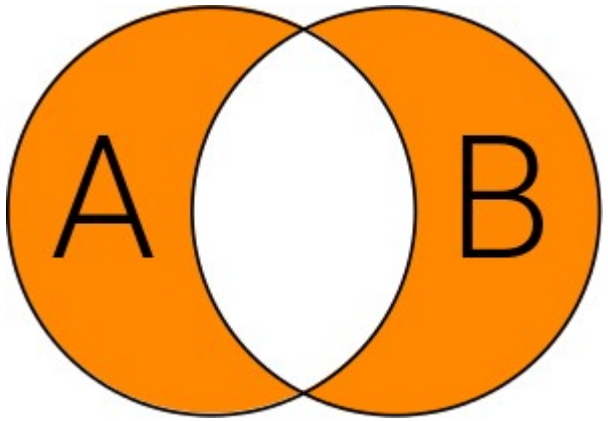



## Full Join

```
SELECT * FROM A
FULL JOIN B
ON A.key = B.key
```

| A | B |
|---|---|
| 1 | NULL |
| 2 | NULL |
| 3 | 3 |
| 4 | 4 |
| NULL | 7 |
| NULL | 8 |

# Ensembles



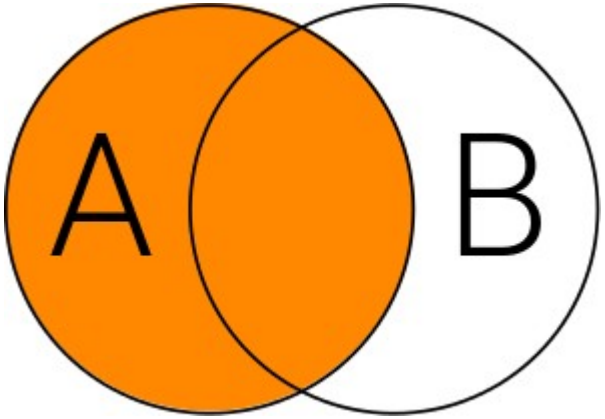

## Full Join

## Sans intersection

```
SELECT * FROM A
FULL JOIN B
ON A.key = B.key
WHERE A.key IS NULL
OR B.key IS NULL
```

| A | B |
|---|---|
| 1 | NULL |
| 2 | NULL |
| 3 | 3 |
| 4 | 4 |
| NULL | 7 |
| NULL | 8 |

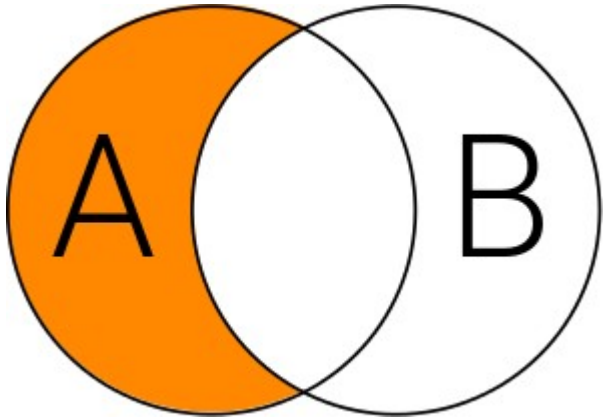
# Ensembles Left




## Left Join

```
SELECT *
FROM A
LEFT JOIN B
ON A.key = B.key
```

| A | B |
|---|---|
| 1 | NULL |
| 2 | NULL |
| 3 | 3 |
| 4 | 4 |
| NULL | 7 |
| NULL | 8 |

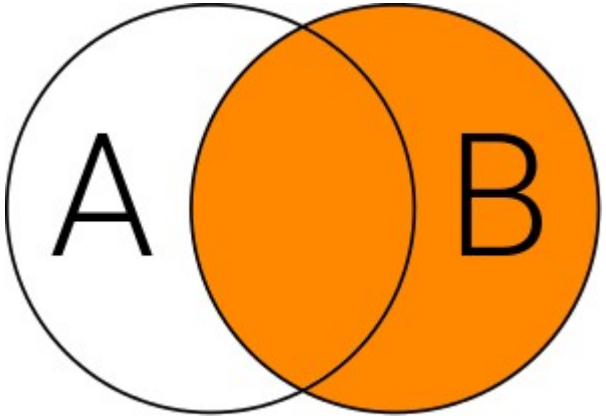



## Left Join sans intersection

```
SELECT *
FROM A
LEFT JOIN B
ON A.key = B.key
WHERE B.key IS NULL
```

| A | B |
|---|---|
| 1 | NULL |
| 2 | NULL |
| 3 | 3 |
| 4 | 4 |
| NULL | 7 |
| NULL | 8 |

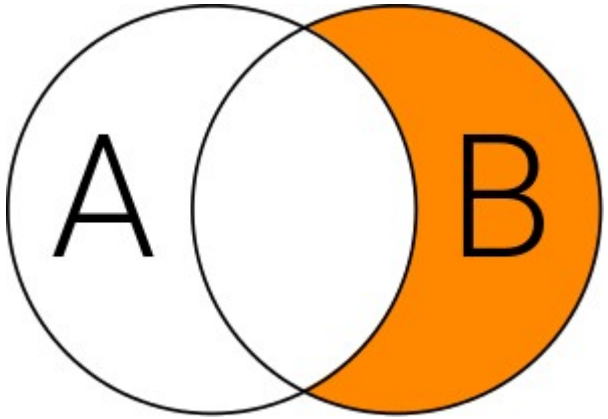
# Ensembles Right




## Right Join

```
SELECT *
FROM A
RIGHT JOIN B
ON A.key = B.key
```

| A | B |
|---|---|
| 1 | NULL |
| 2 | NULL |
| 3 | 3 |
| 4 | 4 |
| NULL | 7 |
| NULL | 8 |




## Right Join sans intersection

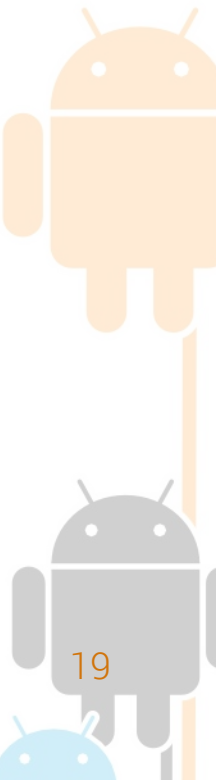
```
SELECT *
FROM A
RIGHT JOIN B
ON A.key = B.key
WHERE A.key IS NULL
```

| A | B |
|---|---|
| 1 | NULL |
| 2 | NULL |
| 3 | 3 |
| 4 | 4 |
| NULL | 7 |
| NULL | 8 |

# Ensembles

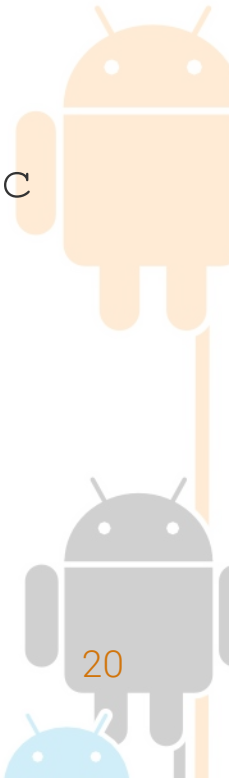
- Ne pas confondre avec le produit cartésien
- Le résultat est l'ensemble de tous les couples (distribution)

```
SELECT * FROM A CROSS JOIN B
```



# CRUD - SQL98

- Create :
  - `INSERT INTO Books (title, isbn, nbPage) VALUES ("Croc blanc", "2010034031", 248)`
- Read :
  - `SELECT * FROM Books WHERE title = "Croc blanc"`
- Update :
  - `UPDATE Books SET nbPage = 248 WHERE title = "Croc blanc"`
- Delete :
  - `DELETE FROM Books WHERE title = "Croc blanc"`

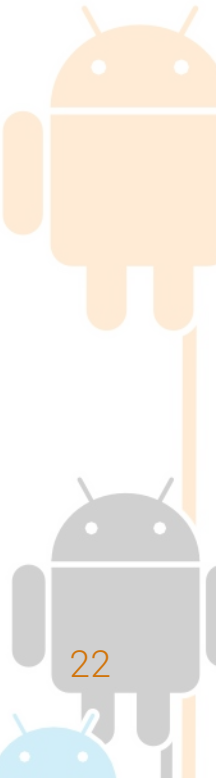


# IN01 – Séance 04

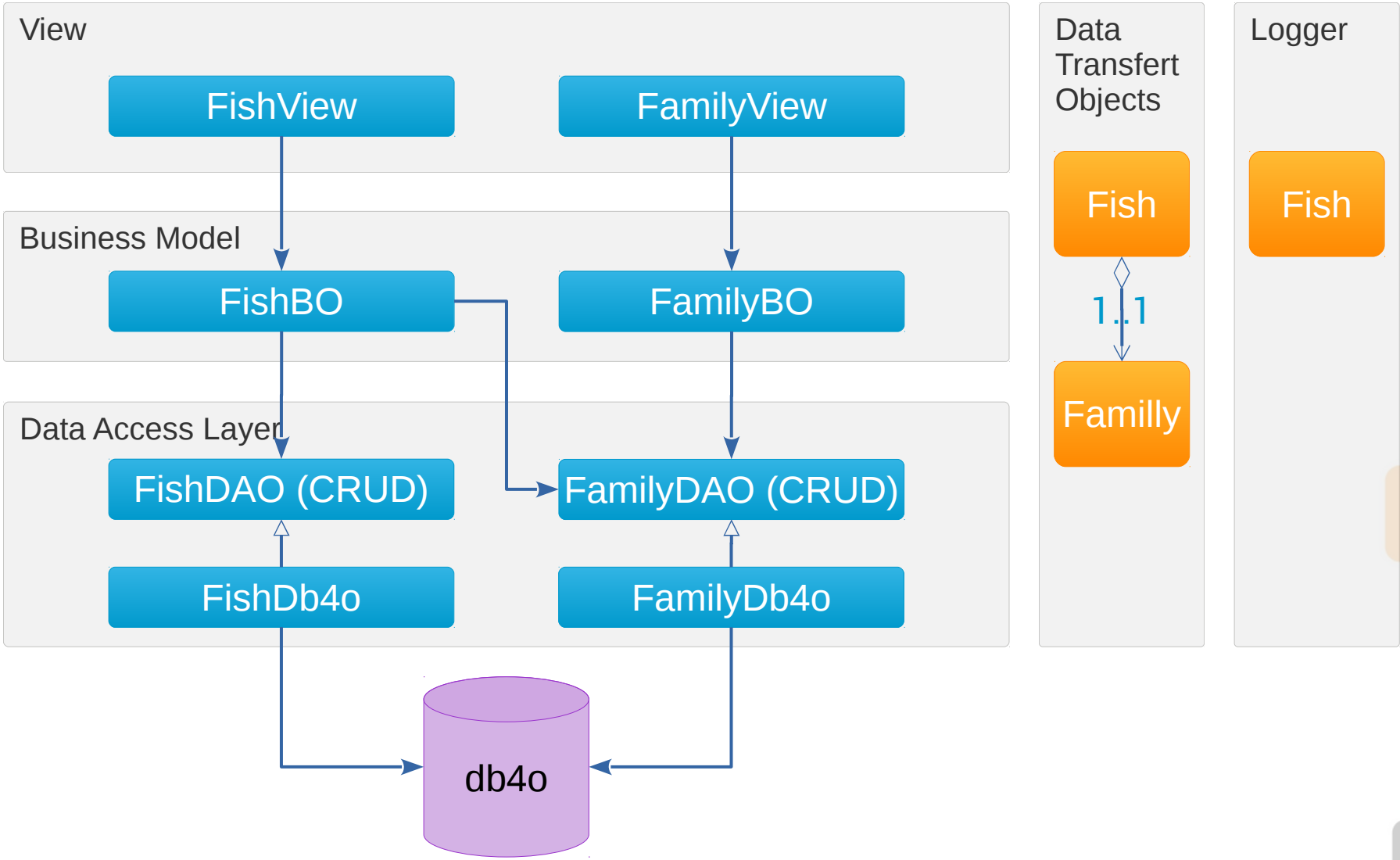
## Couches logiciels

# Patron - Layer

- POSA :: Pattern Oriented Software Architecture
- Architecturer les applications en couches
- Chaque couche à son niveau d'abstraction propre
- Séparation des responsabilités par niveau d'abstraction
- Rend interchangeable les différentes couches
- exemple :: couches OSI (réseau)

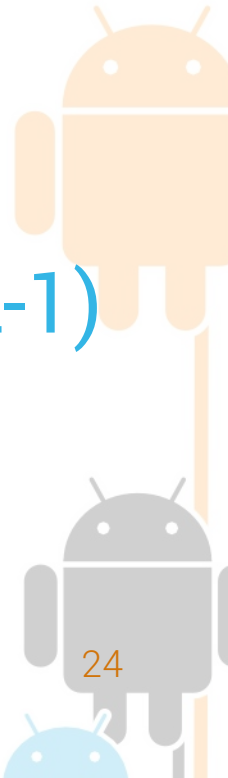


# Patron – Layer – Exemple



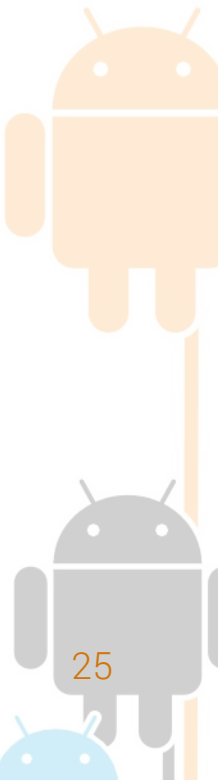
# Patron - Layer

- La couche L
- Elle offre un service à la couche supérieure (L+1)
- Elle délègue les sous tâches à la couche inférieure (L–1)
- Elle collabore qu'avec la couche inférieure (L-1)



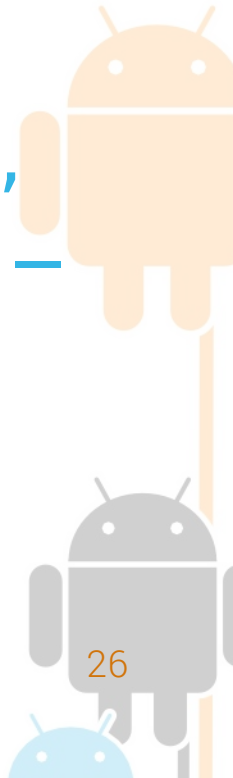
# Patron - Layer

- Communications descendantes directes (création d'objet, appel de méthodes)
  - White boxe, les objets sont accessibles
  - Black boxe, les objets sont utilisés au travers un adapter et / ou des interfaces
- Communications ascendantes découplées (callback, observer / observable, template method – choix pour Algoid)



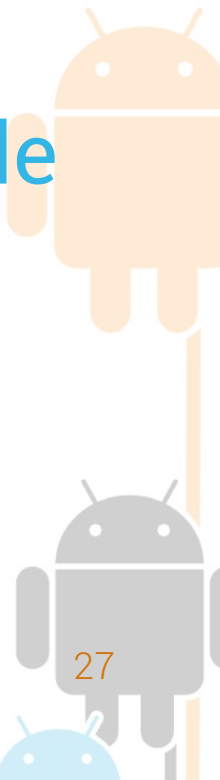
# DTO – Data Transfert Object

- Ensemble de POJOs (Plain Old Java Object) représente le modèle business
- Pour chaque table de la DB = 1 POJO (bean)
- Un bean mutable (attibuts privés, getters, setters, constructeur par défaut – state less, constructeur avec assignation des attributs – state full ....)

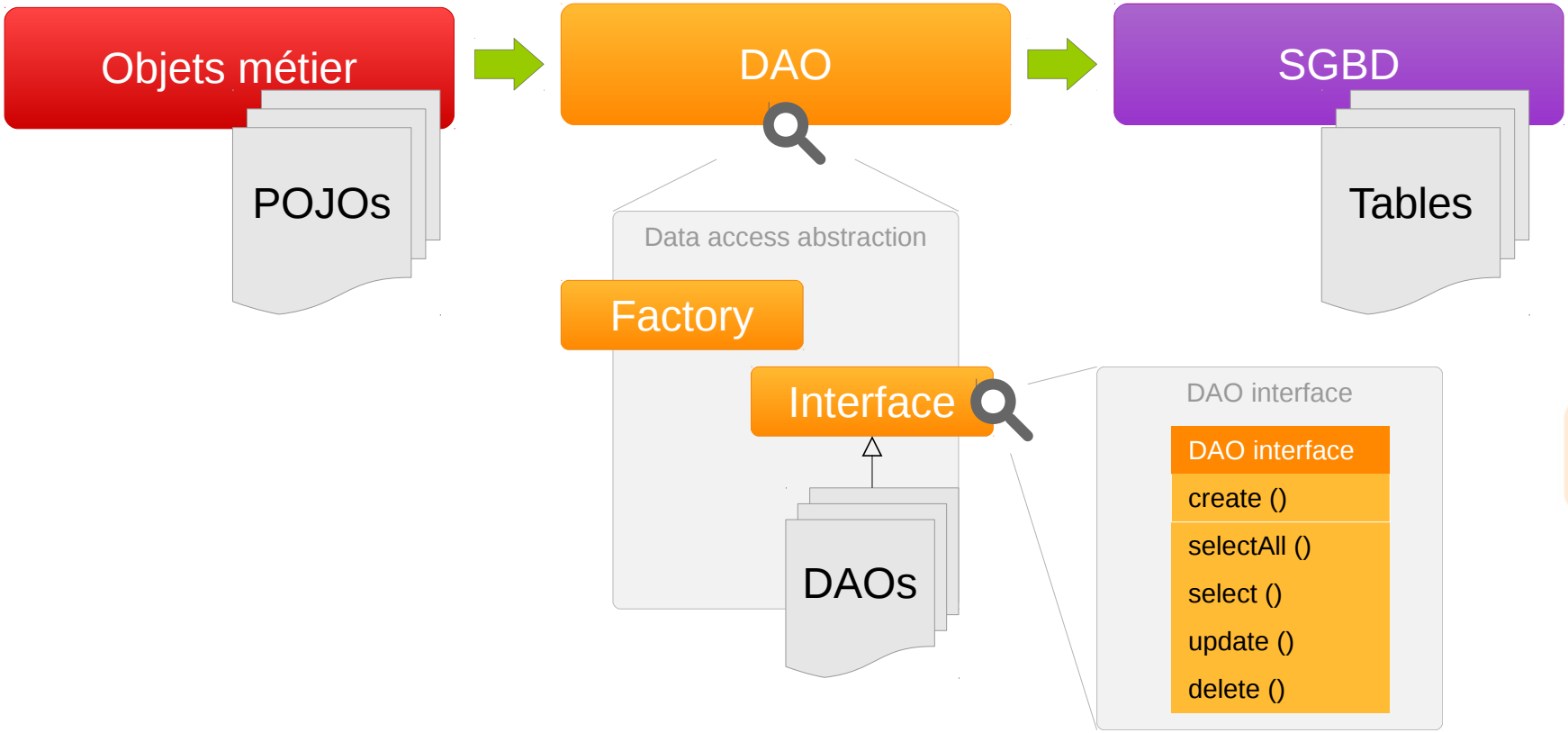


# Un design pattern : DAO

- Data Access Object
- Une interface CRUD : avec les méthodes de création, lecture, modification et suppression
- Permet une séparation des responsabilités entre les objets métiers et l'accès à la base de données.
- 1 table = 1 DAO = 1 DTO (bean)

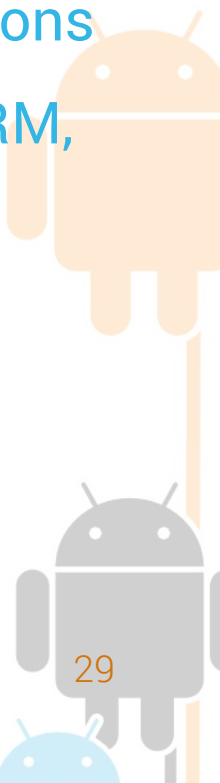


# Un design pattern : DAO



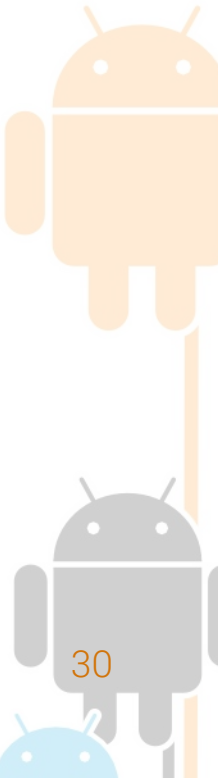
# ORM

- Object Relational Mapping
- Un framework qui le fait pour nous
- Transforme les objets en tables et vice-versa, les attributs en champs.
- Correspondance des types (Java / SQLite)
- Plusieurs modes de configuration : XML, parametrisation, annotations
- Un grand nombre d'ORM sur le marché : Hibernate,  OJB, SimpleORM, **ORMLite**, Nhibernate, Entity Framework, Linq To SQL.
- Une problématique répandue.

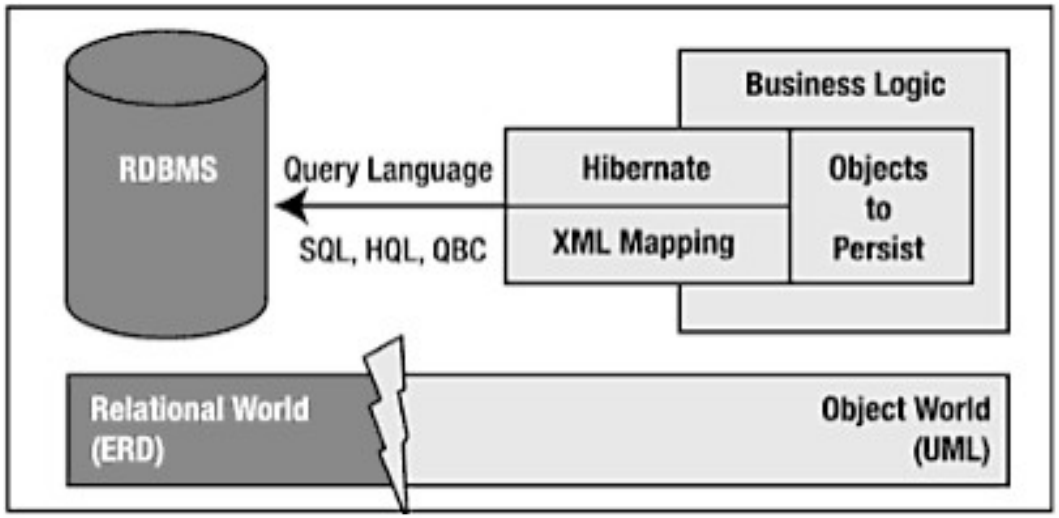


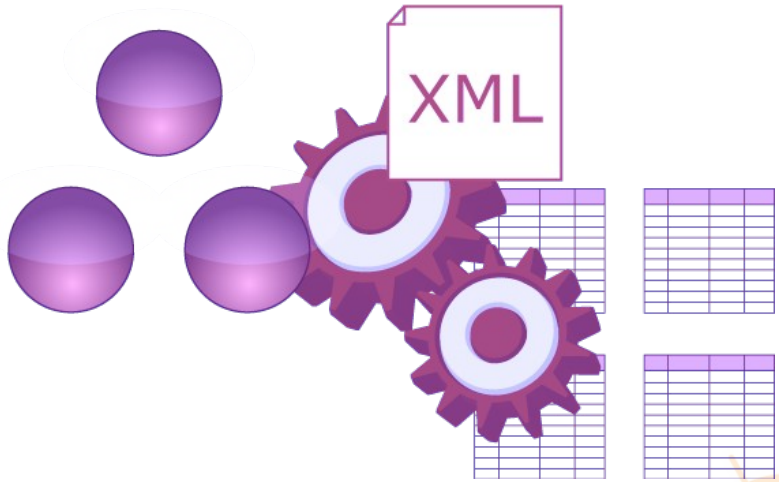
# Alternatives

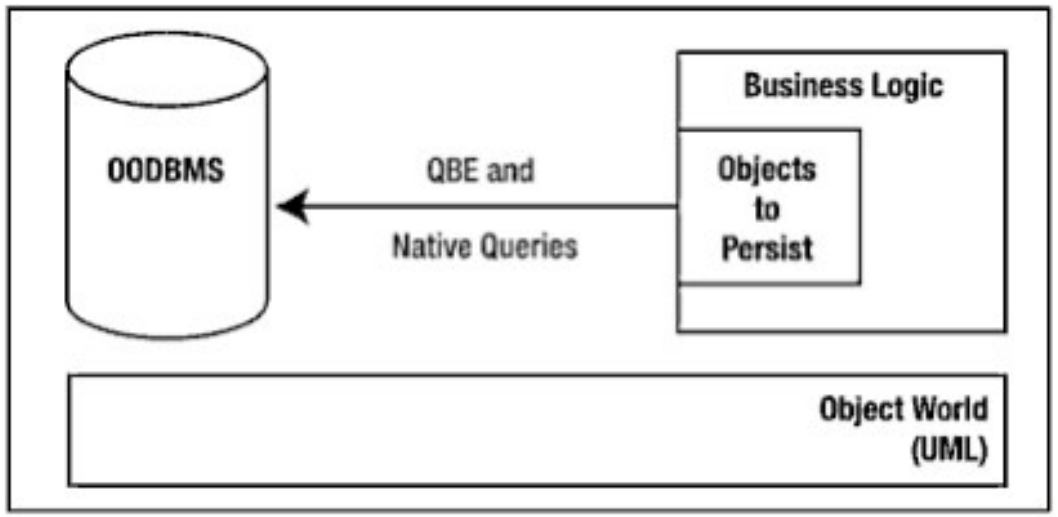
- Systèmes NoSQL (Not Only SQL) :
- Document-oriented database system
  - Basé sur des standards de description de documents : XML, YAML, JSON (JavaScript Object Notation), BSON (Binary JSON)
  - Informix, MongoDB, OrientDB, Cassandra
- Graphs-oriented database system
- OODBMS
  - Object oriented database management system
  - DB4O, une base de données embarquable sur Android

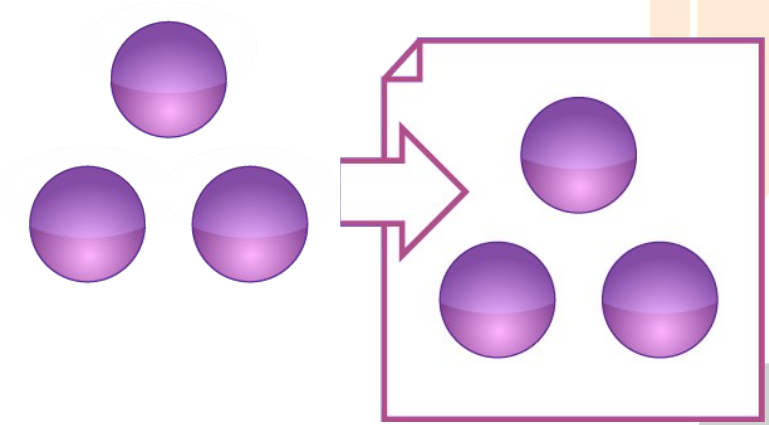


# Différentes approches de la persistence



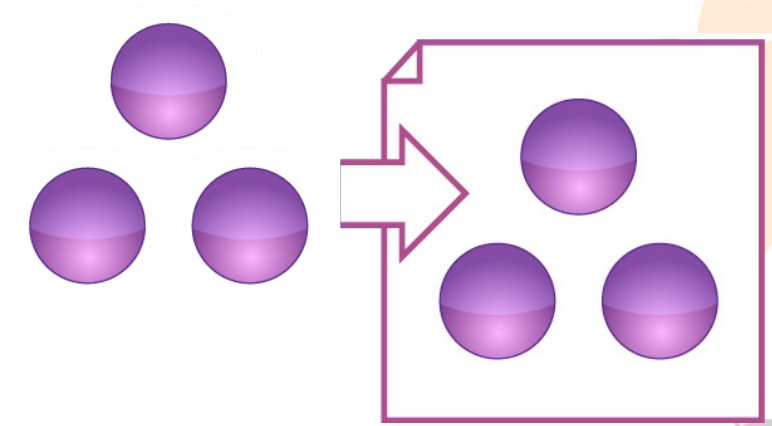



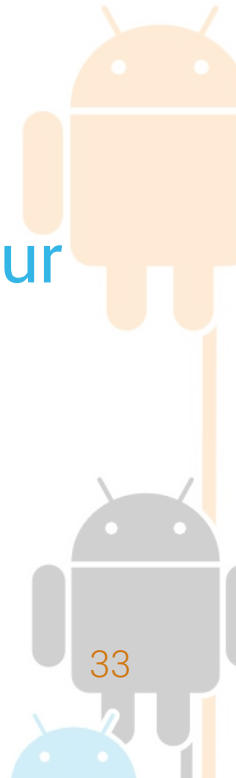
# SGBDOO - Avantages

- Bénéficie de la puissance et de la flexibilité de l'objet (avec les objets on peut tout représenter)
- Pas de mapping
- Relation n / n (many to many) native
- Héritage
- Patron Composite
- Découplage des complexités



# Inconvénients

- Limites des SGBDOO :
  - Méconnues
  - Manque de standardisation malgré l'ODMG (accès multiples)
  - Peu d'interopérabilité avec les BDR et les outils (OLAP, reporting)
  - Toutes n'implémentent pas encore le backup incrémental
  - Coût de migration élevé
- Limitations peu importantes pour une DB embarquée sur Android, n'est-ce-pas ?



# IN01 – Séance 04

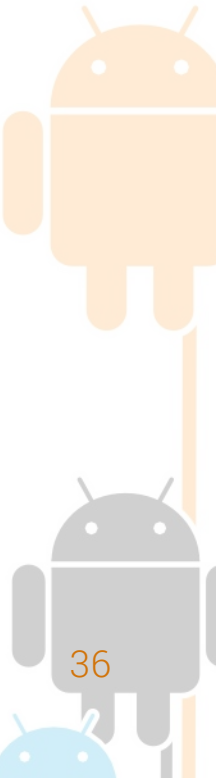
## SQLite

# SQLite

- Moteur de base de données relationnelle
- Crée par D.Richard Hipp
- Ecrit en C (ANSI-C)
- Open source sans restriction
- Multiplate-forme
- Le plus distribué au monde (Firefox, Skype, Google Gears)
- Systèmes embarqués (iPhone, Symbian, Android, mais pas sur Windows Phone)

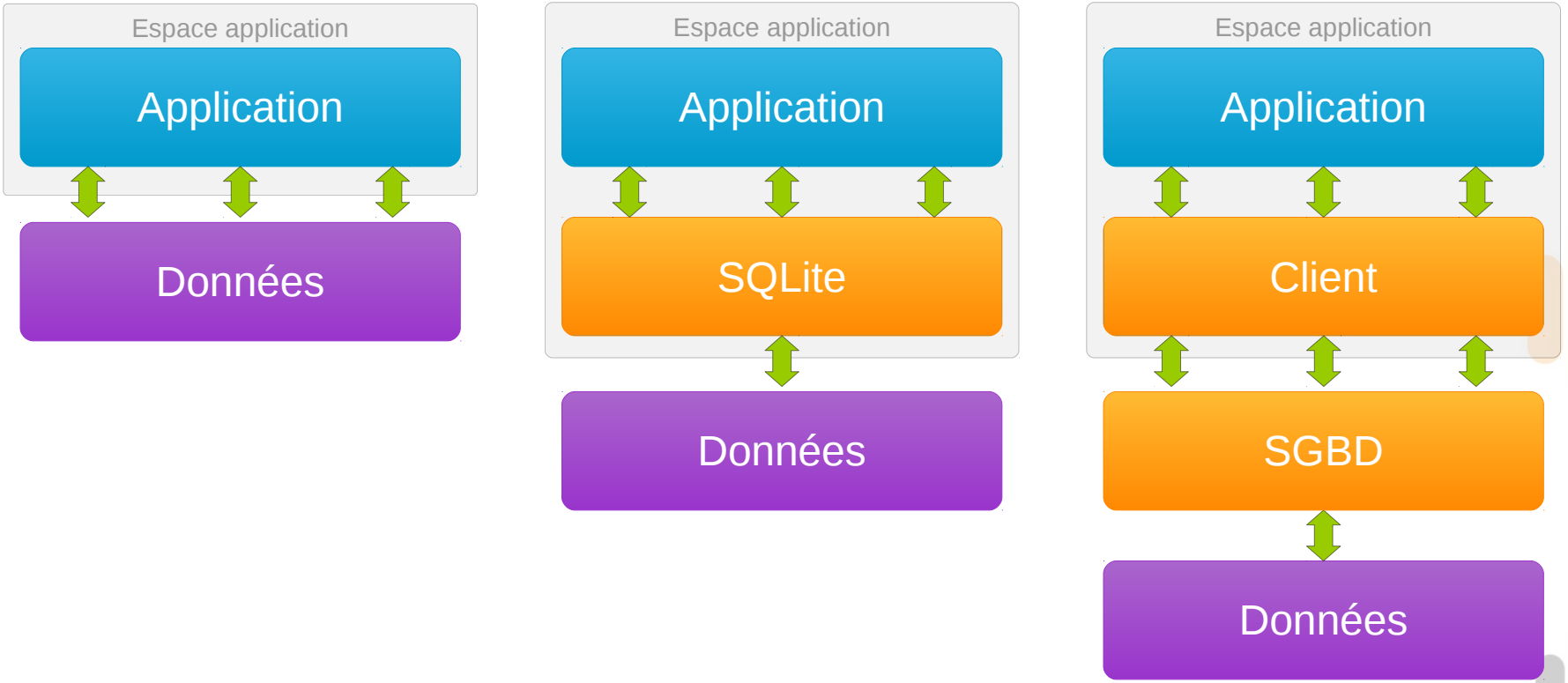
# SQLite

- Implémente le standard SQL-92 (alias SQL2)
- Et les propriétés ACID
- SGBD Embarqué
- Pas de gestion des droits (mono-utilisateur)
- Pas de configuration

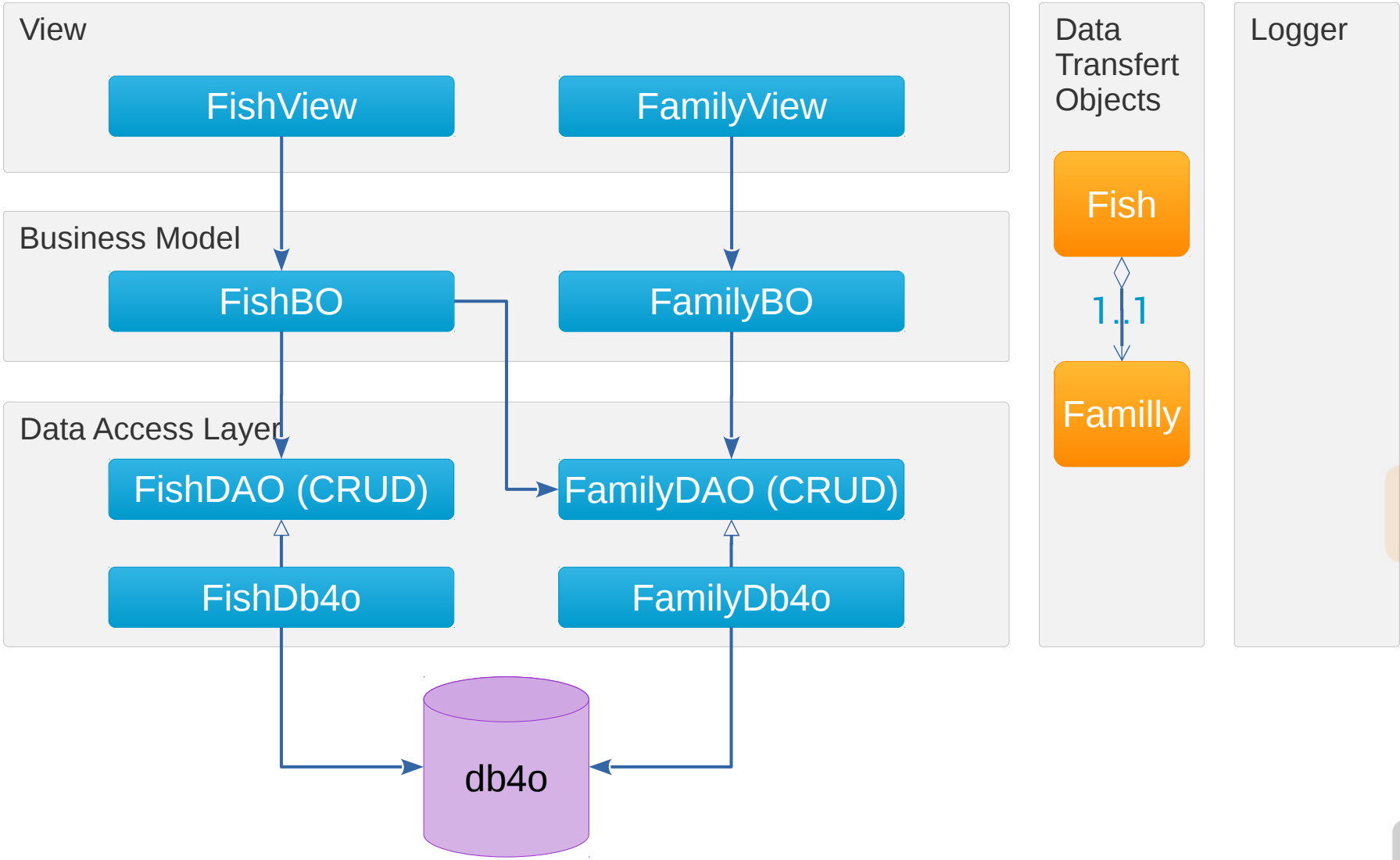


# Embarqué

- Fichiers simples, Modèle embarqué, Modèle client-serveur (n-tiers)

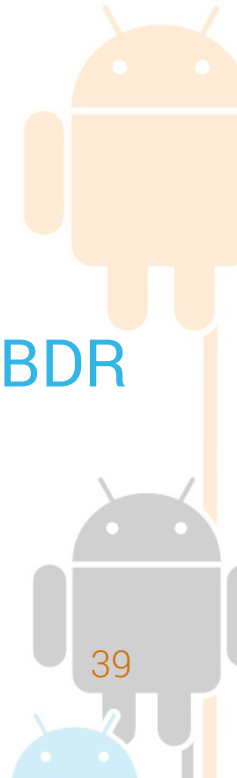


# Patron – Layer – Exemple



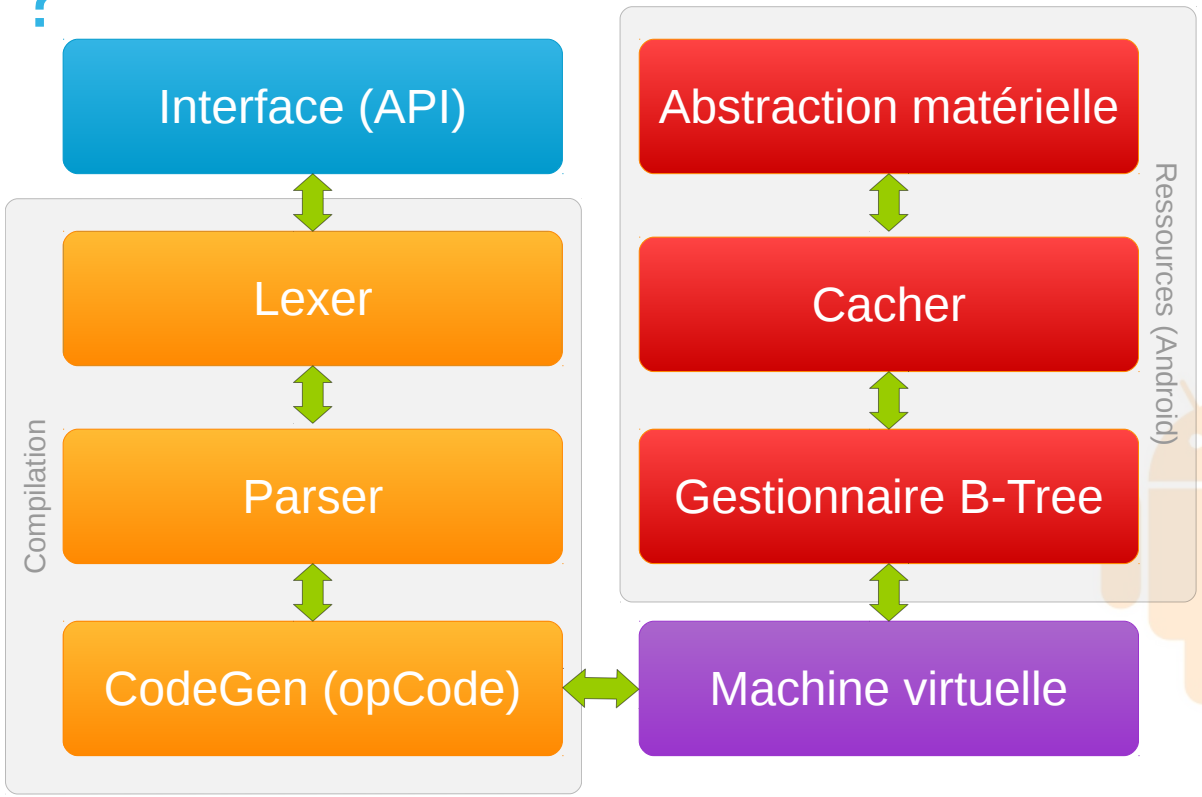
# Mais ....

- Dépendant du système de fichier
- une base de donnée = un fichier
- Attention à la limite sur des DD formatés en FAT 32
- Pas d'extension propre, (“.sqlite”, “.db” sont de bonnes pratiques)
- Possibilité de sauvegarde en mémoire vive (extension “:memory:”)
- Objectif : remplacer le système de fichier, mais pas les SGBDR



# Architecture

- Virtual Machine ?
- OpCode (137 instructions)
- Arbre-B ?
- Abstraction materielle



# Notre bibliothèque avec SQLite

- Le Pojo Book
- Un objet qui représente un record “livre”
- Des attributs, des accesseurs, un constructeur stateless et un statefull

```
public class Book {

  // attributes
  private int id;
  private String title;
  private String isbn;
  private int nbPage;

  // accessors
  public String getTitle() {
    return title;
  }

  public void setTitle(String title) {
     this.title = title;
  }

  // etc....

  // constructor
  public Book(int id) {
    super(id);
  }

  public Book(int id, String title, String isbn, int nbPage) {
    this(id);
    this.title = title;
    this.isbn = isbn;
    this.nbPage = nbPage;
  }

}
```

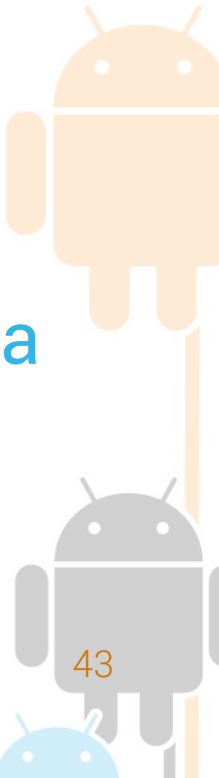
# Création de la base

- Il faut fournir une classe qui a pour responsabilité la création de la base et la mise à jour du schéma

```
public class LibraryDBHelper extends SQLiteOpenHelper {

  @Override
  public void onCreate(SQLiteDatabase db) {
    // TODO Auto-generated method stub
  }

  @Override
  public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    // TODO Auto-generated method stub
  }
}
```

# Creation de la base

- Une classe abstraite : `SQLiteOpenHelper`
- Design pattern : Template method (GoF)
- Un constructeur : données utiles
- Deux méthodes abstraites :
  - `onCreate` : chargée de la création de la DB
  - `onUpgrade` : chargée de la mise à jour du schéma (en général, on efface tout et on recommence)



# Constructeur

- Le context Android
- Le nom du fichier
- Le créateur de curseur (null par défaut)
- La Version
- Un handler en cas de corruption de la base (par défaut)

```
public static final String DB_NAME = "Library.db";
public static final int DB_VERSION = 5;

// constructor
public LibraryDBHelper(Context context) {
  super(context, DB_NAME, null, DB_VERSION);
}
```

# Création et Upgrade

```
public class LibraryDBHelper extends SQLiteOpenHelper {

  public static final String DB_NAME = "Library.db";
  public static final int DB_VERSION = 5;

  public static String getQueryCreate() {
    return "CREATE TABLE Book ("
         + "id Integer PRIMARY KEY AUTOINCREMENT, "
         + "title Text NOT NULL, "
         + "isbn Text NOT NULL, "
         + "nbPage Integer NOT NULL"
         + ");";
  }

  public static String getQueryDrop() {
    return "DROP TABLE IF EXISTS Book;";
  }

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

  @Override
  public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    db.execSQL(getQueryDrop());
    db.execSQL(getQueryCreate());
  }
}
```

Crée les tables

Recrée les tables

# Gestion plus fine de l'upgrade

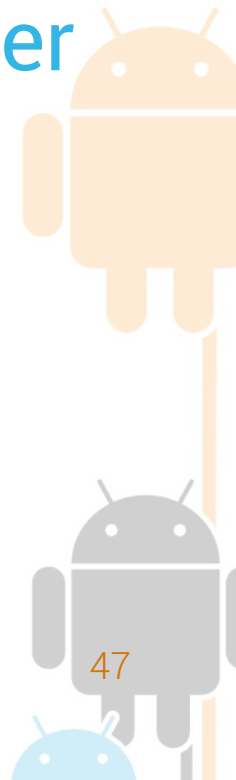
- Des petits pas jusqu'à la version courante

```java
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
  // Management by successive upgrades
  // newVersion is 4 !
  int delta = newVersion - oldVersion;
  if (delta >= 3) {
    // upgrate to version 2
  }

  if (delta >= 2) {
    // upgrate to version 3
  }

  if (delta >= 1) {
    // upgrate to version 4
  }
}
```

Gère les changements étape par étape

# Datasource

- Une classe responsable de l'accès à la base de données
- Responsable de créer le helper
- Ouvrir (`open()`) / Fermer (`close()`) / Donner accès aux données (`getDB()`)
- Factory des différents DAO



# Datasource

- Crée le helper
- Accès à la db
- Ouvre et ferme la connection
- Les factories

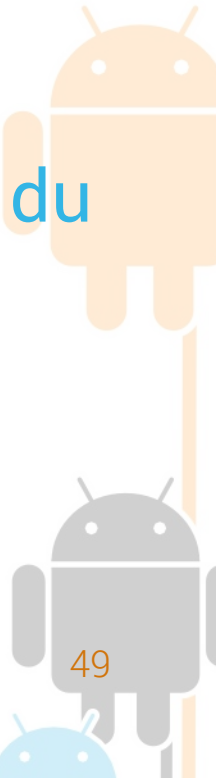
```
public class LibraryDataSource {

  private final LibraryDBHelper helper;
  private SQLiteDatabase db;

  public LibraryDataSource(Context context) {
    helper = new LibraryDBHelper(context);
  }

  public SQLiteDatabase getDB() {
    if (db == null) open(); // lazy initialization
    return db;
  }

  public void open() throws SQLException {
    db = helper.getWritableDatabase();
  }

  public void close() {
    helper.close();
  }

  // factories
  public BookDAO newBookDAO() {
    return new BookDAO(this); // flyweight ?
  }
}
```

Accès à la base

DAO factories

# DAO & CRUD

- Référence la datasource
- `Create()` : crée le record à partir du Pojo, gestion de son nouvel ID
- `Read()` : lit le record selon l'ID et charge ses champs avec les valeurs de la DB
- `Update()` : met à jour les valeurs de la base à partir du Pojo
- `Delete()` : supprime le record selon l'ID du Pojo



# Ecrire la donnée

- Une classe `android.content.ContentValues`
- Tableau associatif (Set)
  - Clé : Nom du champ en base
  - Valeur : valeur de la donnée dans le Pojo
- Une méthode `put(String key, ???? value)` pour créer l'association (overload sur chaque type)
- Des méthodes pour récupérer les valeurs selon les clés (`getAsInteger()`, `getAsDouble()` ….)

# Méthode create() du DAO

- Crée le tableau associatif avec les valeurs du Pojo
- Exécute la requête d'insertion
- Met à jour l'ID
- Retourne le Pojo mis à jour

```
Public synchronized Book create(Book pojo) {
  // create associative array
  ContentValues values = new ContentValues();
  values.put(COL_TITLE, pojo.getTitle());
  values.put(COL_ISBN, pojo.getIsbn());
  values.put(COL_NBPAGE, pojo.getNbPage());

  // insert query
  int id = (int) getDB().insert(TABLE_NAME, null, values);

  // update id into pojo
  pojo.setId(id);

  return pojo;
}
```

Tableau associatif

API level

# Méthode update() du DAO

- Le même principe que l'insert
- La clause en plus

```
Public synchronized Book update(Book pojo) {
  // create associative array
  ContentValues values = new ContentValues();
  values.put(COL_TITLE, pojo.getTitle());
  values.put(COL_ISBN, pojo.getIsbn());
  values.put(COL_NBPAGE, pojo.getNbPage());

  // where clause
  String clause = COL_ID + " = ?";
  String[] clauseArgs = new String[]{String.valueOf(pojo.getId())};

  // update
  getDB().update(TABLE_NAME, values, clause, clauseArgs);

  // return the pojo
  return pojo;
}
```

Expression de recherche

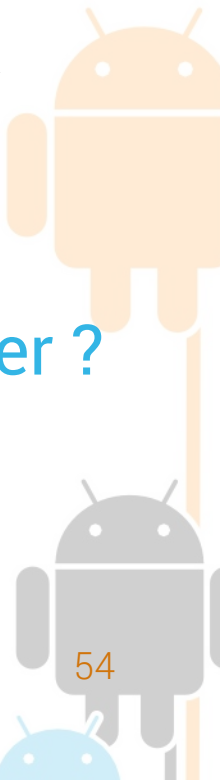
# Méthode delete() du DAO

- Que la clause where, cette fois-ci !

```
Public synchronized void delete(Book pojo) {
  // where clause
  String clause = COL_ID + " = ?";
  String[] clauseArgs = new String[]{String.valueOf(pojo.getId())};

  // delete
  getDB().delete(TABLE_NAME, clause, clauseArgs);
}
```

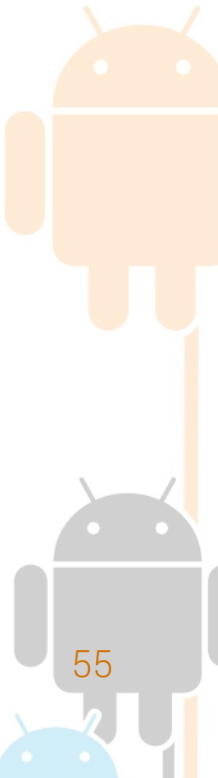
# Lire la donnée

- Deux méthodes :
  - `rawQuery (String query, String[] args)`
    - Remplace les caractères “?” par les valeurs passés en argument
    - ex : rawQuery(“SELECT * FROM book WHERE id = ?”, 1);
  - `query (String tableName, String[] columns, String clause, String[] args, String groupBy, String having, String orderBy)`
  - Idée, le moins de SQL possible dans le code
- Pourquoi pas un design pattern de Builder ou Interpreter ?



# Cursor

- La classe `android.database.Cursor`
- Un design pattern Iterator
- Sur une collection d'éléments typés (mais pas nommés)



# Méthode read() du DAO

```
public Book read(Book pojo) {
  // columns
  String[] allColumns = new String[]{COL_ID, COL_TITLE, COL_ISBN,   COL_NBPAGE};

  // clause
  String clause = COL_ID + " = ?";
  String[] clauseArgs = new String[]{String.valueOf(pojo.getId())};

  // select query
  Cursor cursor = getDB().query(TABLE_NAME, allColumns, "ID = ?",  clauseArgs, null, null, null);

  // read cursor
  cursor.moveToFirst();
  pojo.setTitle(cursor.getString(1));
  pojo.setIsbn(cursor.getString(2));
  pojo.setNbPage(cursor.getInt(3));
  cursor.close();

  return pojo;
}
```

Expression de recherche

Requête

Chargement du Pojo

# Méthode readAll() du DAO

```
public List<Book> readAll() {
  // columns
  String[] allColumns = new String[]{COL_ID, COL_TITLE, COL_ISBN, COL_NBPAGE};

  // select query
  Cursor cursor = getDB().query(TABLE_NAME, allColumns, null, null, null, null, null);

  // iterate on cursor and retreive result
  List<Book> books = new ArrayList<Book>();

  cursor.moveToFirst();
  while (!cursor.isAfterLast()) {

    books.add(new Book(cursor.getInt(0),    cursor.getString(1),
                       cursor.getString(2), cursor.getInt(3)));

    cursor.moveToNext();
  }

  cursor.close();

  return books;
}
```

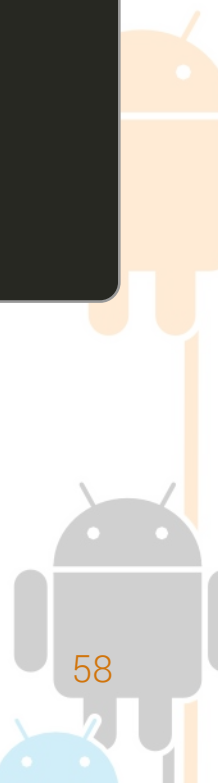
Itération sur le recordset

# Un peu de refactoring

- Une classe abstraite commune à tous les Pojos
- Une interface pour les datasources

```
public interface DataSource {

  SQLiteDatabase getDB();
  void open();
  void close();

}
```

```
public abstract class Pojo {

  protected int id;

  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public Pojo(int id) {
    this.id = id;
  }
}
```



# Un peu de refactoring

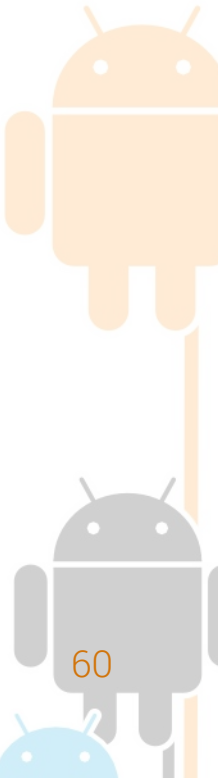
- Une classe abstraite pour tous les DAO
- Des interfaces pour rajouter des spécificités
- Plus très loin d'un ORM !

```
public abstract class DataAccessObject<P extends Pojo> {

  private final DataSource datasource;

  public DataAccessObject(DataSource datasource) {
    this.datasource = datasource;
  }

  public SQLiteDatabase getDB() {
    return datasource.getDB();
  }

  // CRUD
  public abstract P create(P pojo);

  public abstract P read(P pojo);

  public abstract P update(P pojo);

  public abstract void delete(P pojo);
}
```

```
public interface AllReadeable<P extends Pojo> {

  List<P> readAll();

}
```

# Avantages / Inconvénients ?

- Avantages :
  - Séparation des responsabilités
  - Facile à maintenir ?
- Inconvénients
  - Pas vraiment typés (String[] args)
  - Encore un peu de SQL par endroits (“id = ?”)
  - Mapping fastidieux, beaucoup de code à maintenir
  - BLOB (objet binaire) limité à 1Mb

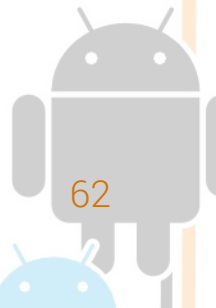


# IN01 – Séance 04

## Object Relational Mapping

# ORMLite

- Object Relational Mapping
- Basé sur des annotations (plus simple qu'Hibernate)
  - Facile à mettre en oeuvre
  - Lisible
  - Contre : intrusif, ne peut pas être appliqué à un objet dont on ne possède pas les sources
- Téléchargement : http://ormlite.com/releases/
- Libraries : ormlite-core et ormlite-android

# Pojo

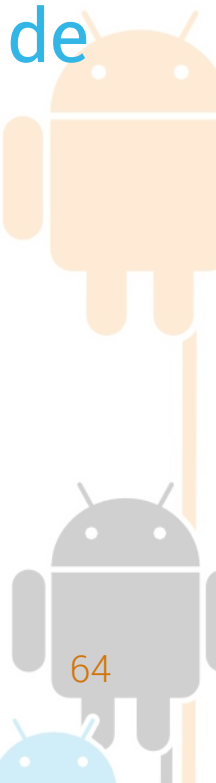
- A quoi ressemble notre nouveau Pojo ?
- Quelques annotations
  `@DatabaseTable`
- `@DatabaseField`
- Un constructeur par défaut obligatoire

```
public class Book extends Pojo {

  // fields
  @DatabaseField(generatedId = true)
  protected int id;
  @DatabaseField(index = true)
  private String title;
  @DatabaseField()
  private String isbn;
  @DatabaseField
  private int nbPage;

  // etx ….

  // accessor
  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }
}
```

Annotation de mapping

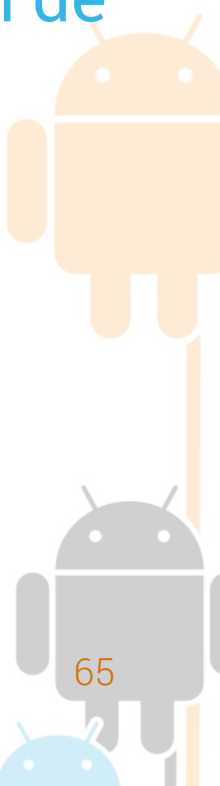
# @DatabaseTable

- Assure la correspondance entre le Pojo, la table et le DAO
- Des paramètres à l'annotation :
  - `tableName` : spécifie le nom de la table (le nom de la classe est utilisé par défaut)
  - `daoClass` : le nom de la classe du DAO



# @DatabaseField

- Permet de spécifier quels attributs doivent être persistés
- Des paramètres à l'annotation :
  - `columnName` : le nom du champ dans la base (le nom de l'attribut est utilisé par défaut)
  - `index` : si le champ est indexé
  - `generatedId` : ID auto incrémental
  - `canBeNull, foreign, unique` ….



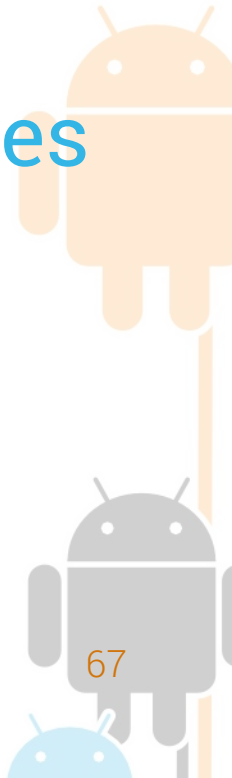
# Le DBHelper

```
public class LibraryDBHelper extends OrmLiteSqliteOpenHelper {

  public static final String DB_NAME = "Library.db";
  public static final int DB_VERSION = 4;

  public LibraryDBHelper(Context context) {
    super(context, DB_NAME, null, DB_VERSION);
  }

  @Override
  public void onCreate(SQLiteDatabase sqld, ConnectionSource cs) {
    try {
      TableUtils.createTable(connectionSource, Book.class);
    } catch (SQLException ex) {
      throw new RuntimeException(ex);
    }
  }

  @Override
  public void onUpgrade(SQLiteDatabase sqld, ConnectionSource cs, int i, int i1) {
    try {
      TableUtils.dropTable(connectionSource, Book.class, true);
      TableUtils.createTable(connectionSource, Book.class);
    } catch (SQLException ex) {
      throw new RuntimeException(ex);
    }
  }
}
```

Factory automatique, lien sur la classe

# OrmLiteSqliteOpenHelper

- Doit hériter de la classe `OrmLiteSqliteOpenHelper`
- Même principe que pour SQLite, en charge de créer ou d'upgrader le schéma de la base
- Mais s'appuie sur les Pojo pour la création des Tables : `TableUtils.createTable`
- Et le drop : `TableUtils.dropTable`



# Mise en oeuvre

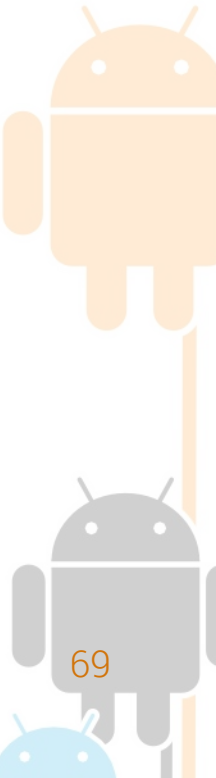
```
public class MainActivity extends OrmLiteBaseActivity<LibraryDBHelper> {

  private void doSomeDBStuff() {
    RuntimeExceptionDao<Book, Integer> bookDAO =   getHelper().getDao(Book.class);

    Book book1 = new Book(-1, "Croc Blanc", "2010034031", 248);
    Book book2 = new Book(-1, "L'appel de la forêt", "2253039861", 158);

    bookDAO.create(book1);
    bookDAO.create(book2);

    List<Book> books = bookDAO.queryForAll();
    for (Book book : books) {
      Log.w("READ ALL", book.toString());
    }

    Log.e("READ", bookDAO.queryForId(1).toString());
    }
  }
}
```

DAO factory

Itération sur la liste de Pojos

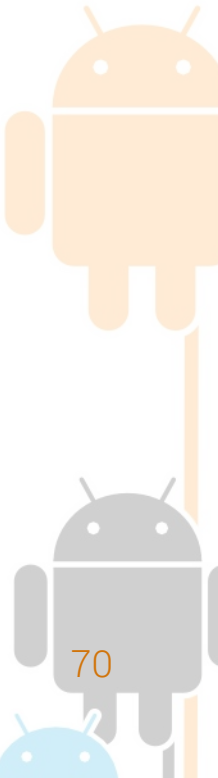
# DAO

- Un nouveau type d'activity
  `OrmLiteBaseActivity`
- Même principe qu'avec les DAO
- Le DAO est récupéré auprés du helper
  - Meilleure gestion des appels multi-threads
  - Utilise la méthode `getDao(Class<T> clazz)`



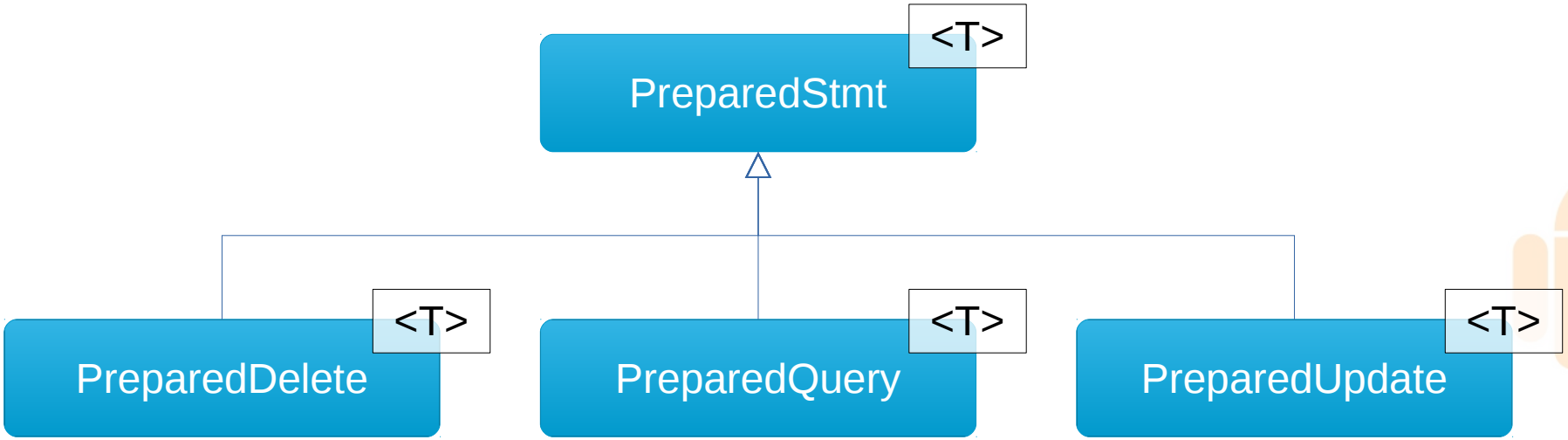
# Méthodes du DAO

- Les DAOs ORMLite définissent plusieurs méthodes :
  - create (T data) : C
  - queryForID (int id) : (R) requête sur l'identifiant
  - queryForEq (String fieldName, Object value) : (R) requête sur une valeur
  - update (T data) : U
  - delete (T data) : D



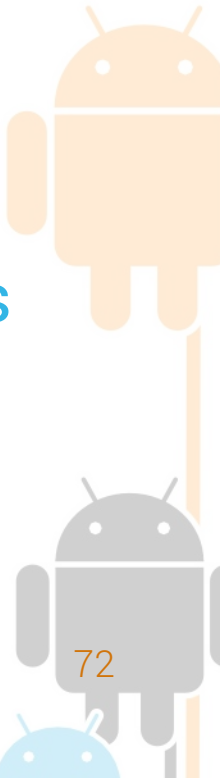
# PreparedStmt

- Requêtes plus complexes
- Utilisation de requête préparées



# QueryBuilder

- Les requêtes préparées sont construites à partir de fabriques
  - Classes `QueryBuilder, UpdateBuilder, DeleteBuilder`
  - méthodes de configuration de la requête : `where(), orderBy(), and(), or()` ….
  - Une méthode `prepare()` permet de construire la requête configurée
- Enfin un design pattern Builder (GoF) pour construire les requêtes



# QueryBuilder

```
private void doOtherDBStuff() {
  RuntimeExceptionDao<Book, Integer> bookDAO = getHelper().getBookDataDao();

  QueryBuilder<Book, Integer> queryBuilder = bookDAO.queryBuilder();
  Where<Book, Integer> where = queryBuilder.where();

  where.and(
    where.eq("title", "Croc Blanc"),
    where.or(
      where.eq("nbPage", 248),
      where.eq("nbPage", 317)
    )
  );

  List<Book> books = bookDAO.query(queryBuilder.prepare());
  for (Book book : books) {
    Log.w("READ BY QUERY", book.toString());
  }
}
```

Builder

Prepare la requête

# Avantages / Inconvénients ?

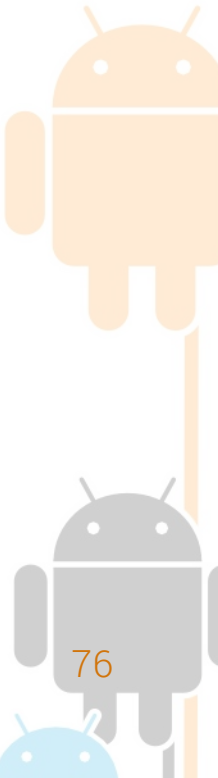
- Avantages :
  - Séparation des responsabilités
  - Moins de code que précédemment (les Pojos et un helper)
  - Un builder pour créer les queries (plus de SQL)
- Inconvénients
  - Pas vraiment typés non plus (where.eq)
  - Mapping Relationel / Objet = complexité supplémentaire

# IN01 – Séance 04

## NoSQL - OODBMS - DB4Object

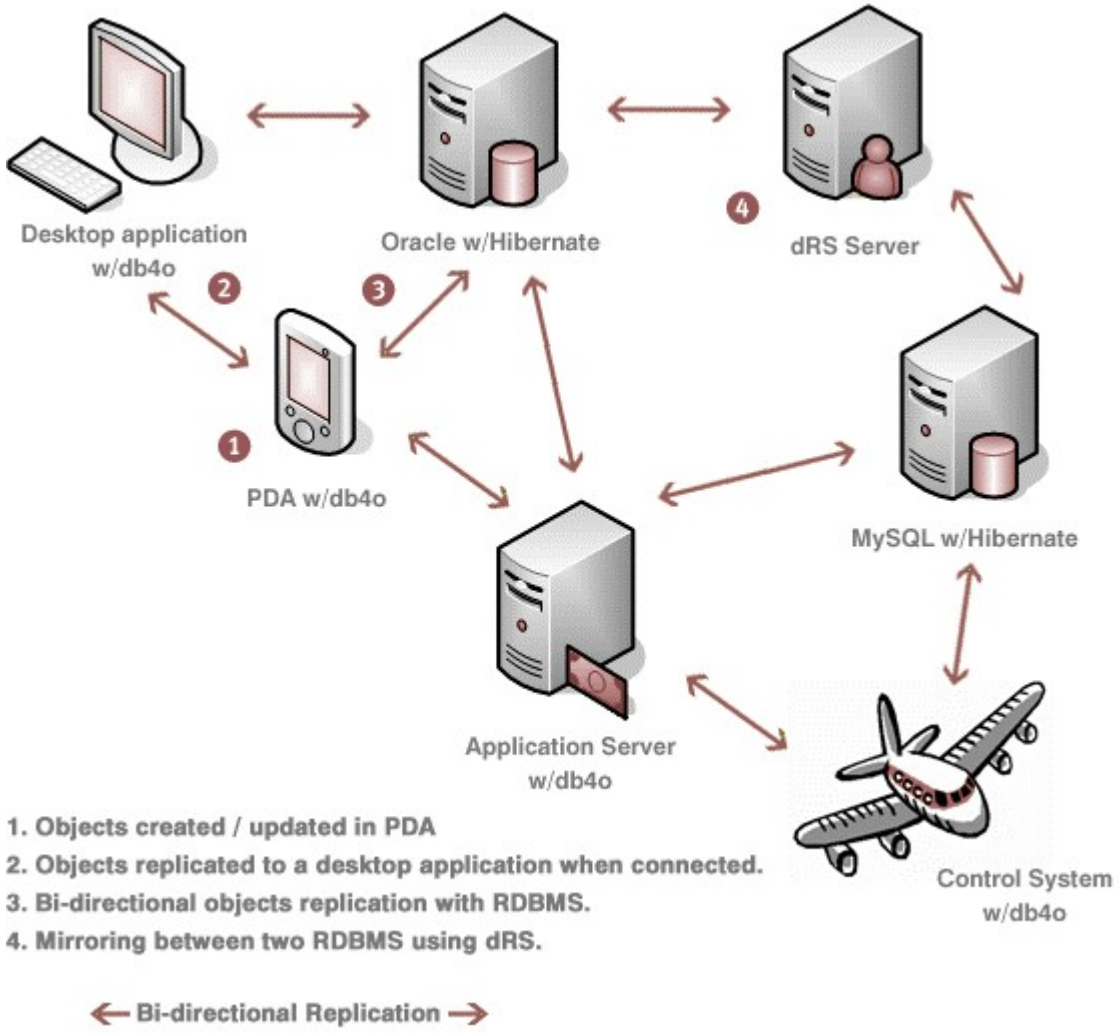
# DB4Object

- Crée par Versant en 2000
- Double license : OpenSource GPL / Commerciale
- Compatible Java et .net
- Divers secteurs d'activités (Mobile, Navigation, SCADA, Devices and equipments, Domotique)
- Embarquable sur Android
- NoSQL / OODBMS
- Des outils de management (plugins Eclipse ou VisualStudio)
- Replication objet ou relationnelle



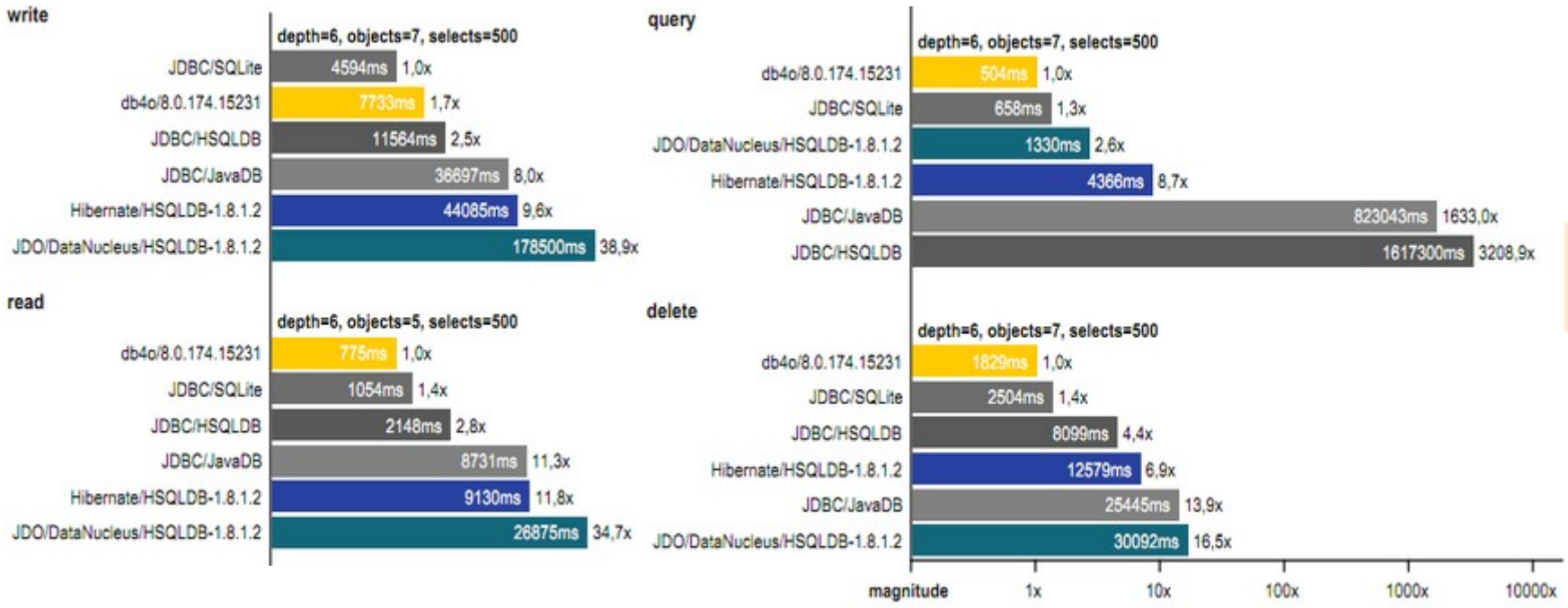
# Replication

- Réplication bi-directionnelle
- Avec des SGBDR via Hibernate



# Performances

- Réalisé par Versant, sur un graph complexe d'objet et un héritage de 5 niveaux



# Nos Pojos

- L'ID n'est plus nécéssaire : relations basées sur les pointeurs
- Réduit à sa plus simple expression

```
public class Book {

  // attributes
  private String title;
  private String isbn;
  private int nbPage;

  // accessors
  public String getTitle() {
    return title;
  }

  public void setTitle(String title) {
    this.title = title;
  }

  // etc ….

  // constructor ….

  // toString ….
```

# Mise en oeuvre

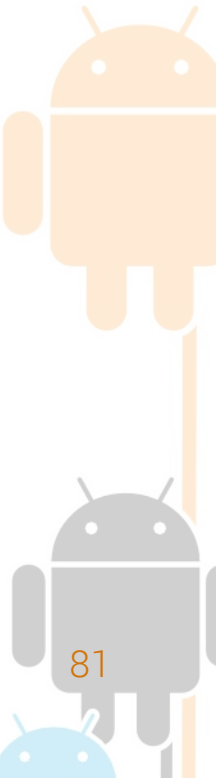
- user-permission `WRITE_EXTERNAL_STORAGE`

```
public static final String DB_FILE = root + "/db.db4o";

  private void doSomeBookStuff(ObjectContainer db) {
    Book book1 = new Book("Croc Blanc", "2010034031", 248);
    Book book2 = new Book("L'appel de la forêt", "2253039861", 158);

    db.store(book1);
    db.store(book2);

    ObjectSet<Book> result = db.queryByExample(Book.class);
    for (Book p : result) {
      textview.append("DB4O SELECT ALL Book " + p + " loaded !\n");
    }
  }
}
```

Méthode store()

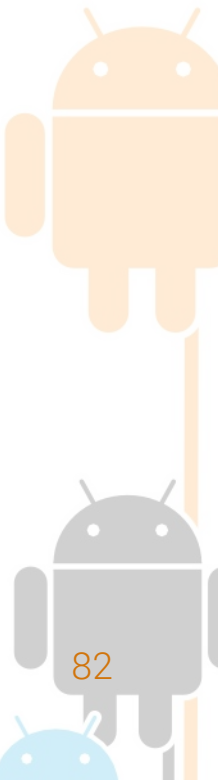
# Mise en oeuvre

- La gestion du modèle devient inutile, DB4O ce charge de tout
  - ➔ Ou presque : veiller à toujours utiliser les mêmes noms de classes, sinon prévoir de configurer la transition
- Insert ou Update automatique



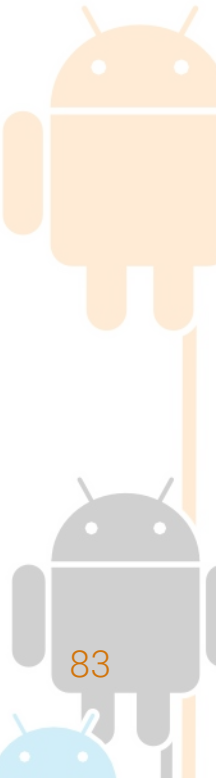
# Héritage

```
private void doSomeHierarchicalStuff(ObjectContainer db) {
  Department root = new Department("T");
  Department td = new Department("D");
  Department to = new Department("O");

  root.addChild(td);
  root.addChild(to);

  db.store(root);

  db.store(new Person("Johanna", "Caron", 26));
  db.store(new Employe(td, "Yann", "Caron", 34));

  ObjectSet<Person> result = db.queryByExample(Person.class);
  for (Person p : result) {
    textview.append("DB4O HIERARCHICAL Object " + p + " loaded !\n");
  }
}
```



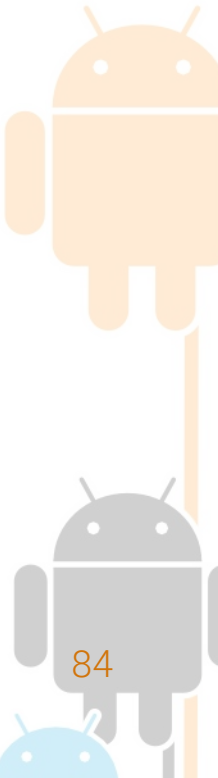
# Héritage

- Un employé est une personne avec un département rattaché
- Les départements sont organisés de façon hiérarchique (Design pattern composite)
- Les relations sont établies sur les pointeurs entre objets
- Aucun mapping !



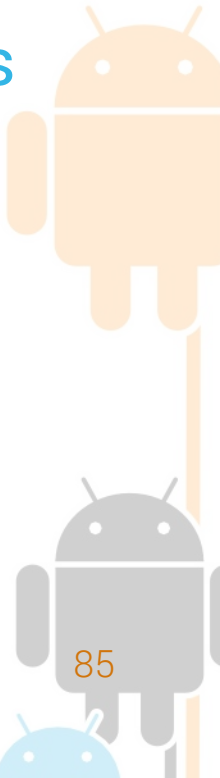
# Mise en oeuvre

- La gestion du modèle devient innutile, DB4O ce charge de tout
  - ➔ Ou presque : veiller à toujours utiliser les même noms de classes, sinon prévoir de configurer la transition
- Insert ou Update automatique



# Requêtes

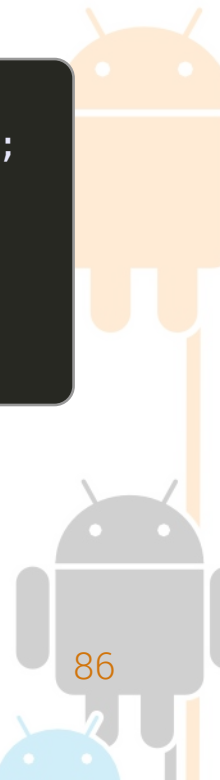
- Trois façons d'effectuer des requêtes :
  - QBE, Query By Example : le plus simple, on donne un objet en exemple, et DB4O retourne tout ce qui y ressemble
  - NQ, Native Query : ressemble à du fonctionnel
  - SODA : Builder de bas niveau, base de construction des deux précédentes
- Vivement une implémentation pour le JDK8 (Map, Filter, Reduce)



# Query By Example

- Principe : créer un objet vide
- Renseigner que les champs qui sont l'objet de la recherche

```
private void doQBEDBStuff(ObjectContainer db) {
  ObjectSet<Book> result = db.queryByExample(new Book("Croc Blanc", null, 0));

  for (Book p : result) {
    textview.append("DB4O QBE Object " + p + " loaded !\n");
  }
}
```



# Native Query

- Predicat ! Jointure ?

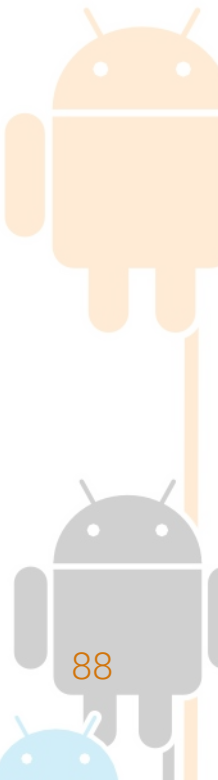
```
private void doNQDBStuff(ObjectContainer db) {
  ObjectSet<Book> result = db.query(new Predicate<Book>() {
    @Override
    public boolean match(Book book) {
      return "Croc Blanc".equals(book.getTitle());
    }
  });

  for (Book p : result) {
    textview.append("DB4O NQ1 Object " + p + " loaded !\n");
  }

  ObjectSet<Employe> result2 = db.query(new Predicate<Employe>() {

    @Override
    public boolean match(Employe person) {
      textview.append("NQ2 PREDICATE departement = " + person.getDepartement().getName() + "\n");
      return "TD".equals(person.getDepartement().getName());
    }
  });
  for (Employe e : result2) {
    textview.append("DB4O NQ2 Object " + e + " loaded !\n");
  }
}
```

Un prédicat

# SODA

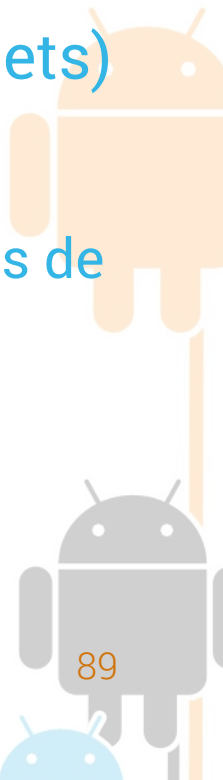
- Ressemble au QueryBuilder vu précédemment
- Plus puissant que les deux précédentes méthodes

```
private void doSODADBStuff(ObjectContainer db) {
  Query query = db.query();
  query.constrain(Book.class);
  query.descend("title").constrain("Croc Blanc").equal();

  ObjectSet<Book> result = query.execute();

  for (Book p : result) {
    textview.append("DB4O SODA Object " + p + " loaded !\n");
  }
}
```



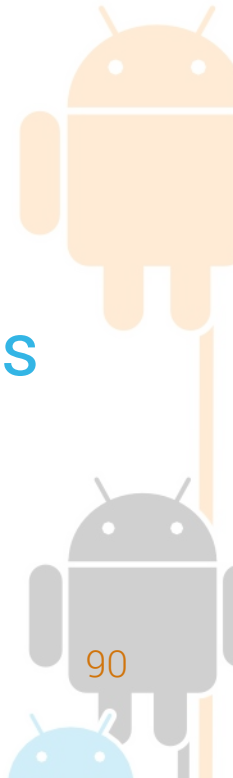
# Constat

- Modèle complexe qui ne pose aucun problème de persistence
- Le code est minimal
- La mise à jour du modèle est automatique
- La gestion create / update également
- Types complexes (tout ce que l'on peut imaginer avec les objets)
  - Composite
  - Interpreter (persister des comportements à interpreter, des formules de calculs par exemple)
  - Réseaux (réseaux de neurones ?)



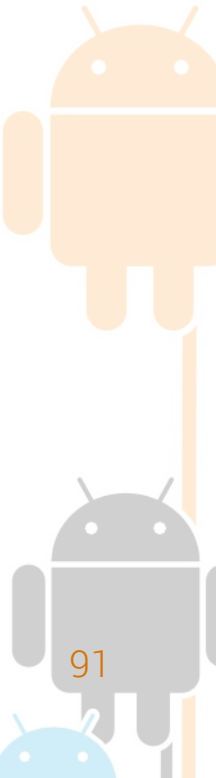
# Et bien plus encore

- De bonnes performances annoncées
- Un “foot print” de 1MB sur le disque
- Une synchronisation prévue avec les SGBDR
- Couche réseau optionnelle
  - Mode client / server
  - Publish / Subscribe pour synchroniser les clients



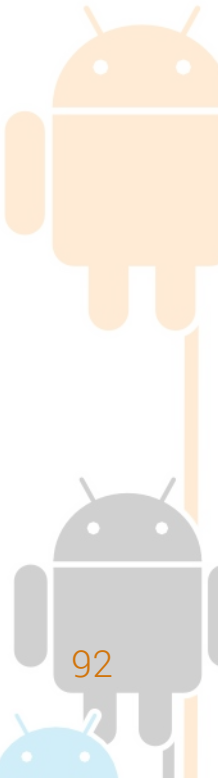
# Pour aller plus loin !

- Intégration avec les lambdas de Java 8 ?
- Qu'en est il des bases de données NoSQL de type document :
  - JasDB
  - CouchDB → CouchBaseLite (plus complexe de mise en oeuvre que db4o)
  - SnappyDB : Clé / Valeurs, c'est tout !?!?



# Bibliographie

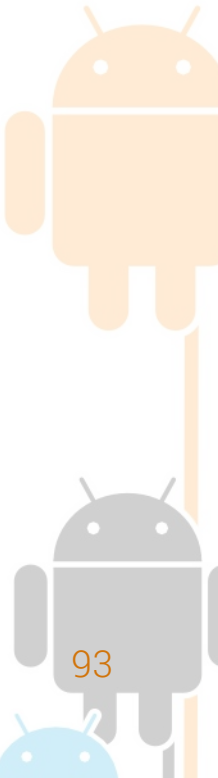
- SQLite : http://www.sqlite.org/
- SQLite sur Developpez.com : http://a-renouard.developpez.com/tutoriels/android/sqlite/
- SQLite-Sync : http://sqlite-sync.com/
- db4objects : http://www.db4o.com/



# Sources de la présentation

- Trouvez toutes les sources de la présentation sur Bitbucket

  https://bitbucket.org/yann_caron/in01/src/



# Fin

- Merci de votre attention
- Des questions ?