Persistance des données en Java

Plan

Introduction

En programmation, la gestion de persistance des données se réfère aux mécanismes responsables de la sauvegarde et la restauration de données, afin que l'état ou les données d'un programme puissent lui survivre dans le temps et dans l'espace.

Introduction

En programmation, la gestion de persistance des données se réfère aux mécanismes responsables de la sauvegarde et la restauration de données, afin que l'état ou les données d'un programme puissent lui survivre dans le temps et dans l'espace.

- La première solution simple de persistance est l'utilisation des fichiers (binaires, textes, plats, XML, ...)

Introduction

En programmation, la gestion de persistance des données se réfère aux mécanismes responsables de la sauvegarde et la restauration de données, afin que l'état ou les données d'un programme puissent lui survivre dans le temps et dans l'espace.

- La première solution simple de persistance est l'utilisation des fichiers (binaires, textes, plats, XML, ...)
- Pour une application répartie, les fichiers ne suffisent plus. Il faut donc une solution pour centraliser toutes les données persistantes.

Introduction

En programmation, la gestion de persistance des données se réfère aux mécanismes responsables de la sauvegarde et la restauration de données, afin que l'état ou les données d'un programme puissent lui survivre dans le temps et dans l'espace.

- La première solution simple de persistance est l'utilisation des fichiers (binaires, textes, plats, XML, ...)
- Pour une application répartie, les fichiers ne suffisent plus. Il faut donc une solution pour centraliser toutes les données persistantes.
- Dans la majorité des cas on souhaite déléguer la gestion des données persistantes à une base de données relationnelle.

Pourquoi une BD relationnelle ?

- Une théorie solide et des normes reconnues.
- Facilité d'accès et généricité du langage de requête SQL.
- Grande efficacité pour effectuer des recherches complexes dans des grandes bases de données.
- Sécurité et intégrité des données.
- Transaction et gestion de la concurrence.
- Faculté de gestion d'énormes volumes de données.
- Technologie largement dominante dans toute l'industrie informatique pour la gestion des données.
- Large offre commerciale répondant à un très large panel de besoins.

Pourquoi une BD relationnelle ?

- Une théorie solide et des normes reconnues.
- Facilité d'accès et généricité du langage de requête SQL.
- Grande efficacité pour effectuer des recherches complexes dans des grandes bases de données.
- Sécurité et intégrité des données.
- Transaction et gestion de la concurrence.
- Faculté de gestion d'énormes volumes de données.
- Technologie largement dominante dans toute l'industrie informatique pour la gestion des données.
- Large offre commerciale répondant à un très large panel de besoins.

La question que l'on va se poser dans ce cours sera donc "comment gérer *proprement* la persistance des données avec un SGBD-R dans une application développée dans un langage orienté objet comme Java ?"

Limites des BD relationnelles

- Les données des applications sont modélisées en objet et non en relationnel.
- Le relationnel est moins riche que le modèle objet : Pas d'héritage, pas de composition ni d'attribut multivalué.
- La modélisation est faite pour des données ayant une structures quasi immuable. Il est très coûteux de modifier le schéma d'une table en production. Le modèle est peu adapté aux données faiblement structurées.
- La situation oligopolistique du marché des gros SGBD-R implique un coût d'accès prohibitif à certaines technologies (BD réparties, Caches, Clusters, ...).

Toutes ces limites ont permis l'émergence récente d'un nouveau type de base de données : les BD "*NoSQL*" (Not Only SQL). Elles sont très en vogue ces derniers temps car elles comblent un besoin marginal mais non satisfait par les BD relationnelles.

Applications réparties : l'architecture 3-tiers

L'architecture 3-tiers, également connue sous le nom de client-serveur distribué, sépare l'application en trois niveaux de services distincts :

- premier niveau : l'affichage et les traitements locaux (contrôles de saisie, mise en forme des données etc.) sont pris en charge par le client ;
- deuxième niveau : les traitements applicatifs globaux relatifs au domaine de l'application (traitements « métiers »). Ils sont pris en charge par le service applicatif ;
- troisième niveau : les services de base de données sont pris en charge par un SGBD.

Généralement les différents niveaux sont nommés respectivement ainsi : la couche présentation, la couche métier et la couche de persistance.

Persistance en Java

La plate-forme Java met à disposition des développeurs plusieurs technologies pour accéder aux données d'un SGBD-R :

Persistance en Java

La plate-forme Java met à disposition des développeurs plusieurs technologies pour accéder aux données d'un SGBD-R :

- Outils fondés sur les concepts relationnels (Relation, Tuple, ...)
  - JDBC

Persistance en Java

La plate-forme Java met à disposition des développeurs plusieurs technologies pour accéder aux données d'un SGBD-R :

- Outils fondés sur les concepts relationnels (Relation, Tuple, ...)
  - JDBC
- Outils fondés sur les concepts objets (Classe, Objet, ...)
  - JDO
  - Toplink
  - MyBatis
  - JPA 2 (Java Persistence API)
    - EclipseLink
    - Hibernate
    - OpenJPA

Persistance en Java

La plate-forme Java met à disposition des développeurs plusieurs technologies pour accéder aux données d'un SGBD-R :

- Outils fondés sur les concepts relationnels (Relation, Tuple, ...)
  - JDBC
- Outils fondés sur les concepts objets (Classe, Objet, ...)
  - JDO
  - Toplink
  - MyBatis
  - JPA 2 (Java Persistence API)
    - EclipseLink
    - Hibernate
    - OpenJPA

Dans ce cours nous allons étudier un produit de chaque famille pour comprendre concrètement leurs avantages et leurs limites respectives.

Plan

Java DataBase Connectivity

JDBC (Java DataBase Connectivity) est une interface de programmation créée par Sun Microsystems, pour les programmes utilisant la plate-forme Java. Il permet aux applications Java d'accéder par le biais d'une interface commune à des sources de données tabulaires (principalement relationnelles) pour lesquelles il existe un pilote JDBC.

Java DataBase Connectivity

JDBC (Java DataBase Connectivity) est une interface de programmation créée par Sun Microsystems, pour les programmes utilisant la plate-forme Java. Il permet aux applications Java d'accéder par le biais d'une interface commune à des sources de données tabulaires (principalement relationnelles) pour lesquelles il existe un pilote JDBC.

Les objectifs de JDBC sont les suivants :

- Fournir une interface uniforme permettant un accès homogène aux SGBD
- Être simple à mettre en œuvre.
- Être indépendant du SGBD cible en n'utilisant que les fonctionnalités fournies par SQL.

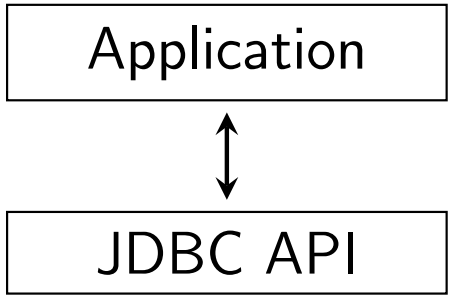
Architecture de JDBC

JDBC est constitué de 3 couches logicielles distinctes :

Architecture de JDBC

JDBC est constitué de 3 couches logicielles distinctes :

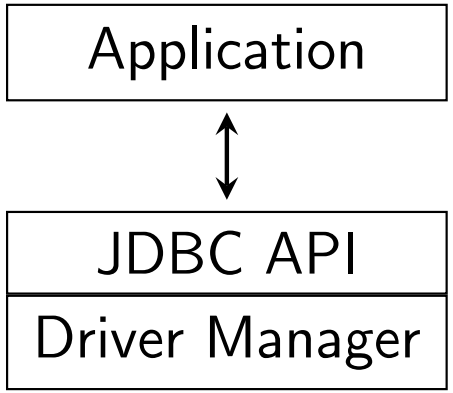
- Une API constituée principalement d'interfaces (au sens Java) qui sera utilisée par le code client.



Architecture de JDBC

JDBC est constitué de 3 couches logicielles distinctes :

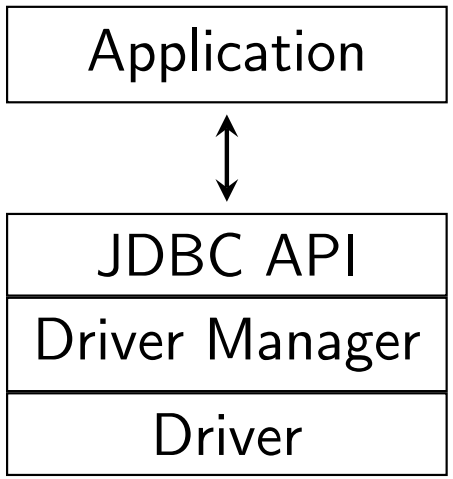
- Une API constituée principalement d'interfaces (au sens Java) qui sera utilisée par le code client.
- Le Driver Manager qui s'occupe de dialoguer avec le pilote propre au SGBD choisi.



Architecture de JDBC

JDBC est constitué de 3 couches logicielles distinctes :

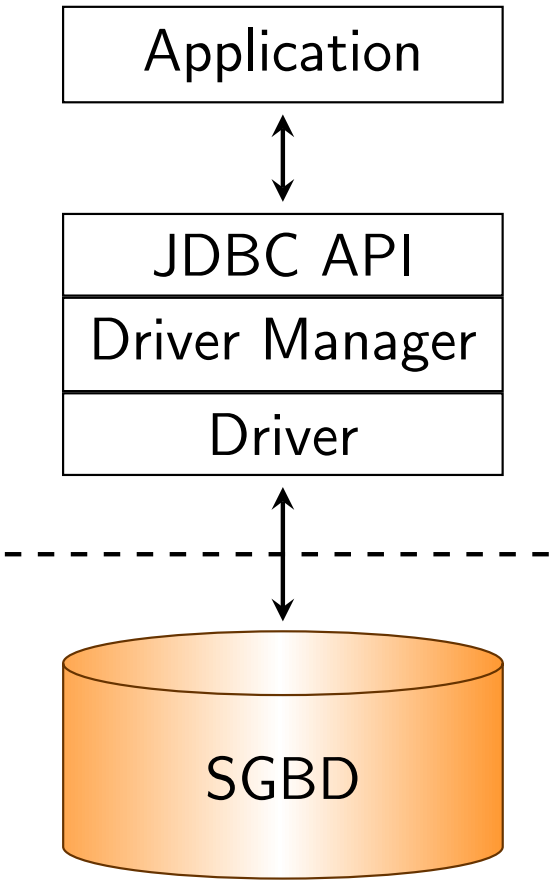
- Une API constituée principalement d'interfaces (au sens Java) qui sera utilisée par le code client.
- Le Driver Manager qui s'occupe de dialoguer avec le pilote propre au SGBD choisi.
- Le Driver (ou pilote) qui a pour rôle de transformer les appels en ordres compréhensibles par le SGBD cible.



Architecture de JDBC

JDBC est constitué de 3 couches logicielles distinctes :

- Une API constituée principalement d'interfaces (au sens Java) qui sera utilisée par le code client.
- Le Driver Manager qui s'occupe de dialoguer avec le pilote propre au SGBD choisi.
- Le Driver (ou pilote) qui a pour rôle de transformer les appels en ordres compréhensibles par le SGBD cible.



API JDBC

L'API JDBC permet principalement de faire 3 types de choses :

- Établir une connexion avec une BD ou toute autre source de données tabulaires.
- Envoyer des ordres écrits en SQL.
- Traiter les résultats.

API JDBC

L'API JDBC permet principalement de faire 3 types de choses :

- Établir une connexion avec une BD ou toute autre source de données tabulaires.
- Envoyer des ordres écrits en SQL.
- Traiter les résultats.

Elle est fournie par les packages `java.sql` et `javax.sql`. Ils contiennent les principales interfaces suivantes :

- `Statement`, `CallableStatement`, `PreparedStatement`
- `DatabaseMetaData`, `ResultSetMetaData`
- `ResultSet`
- `Connection`, `Driver`

API JDBC : interfaces principales

- `Driver` : renvoie une instance de `Connection`
- `Connection` : connexion à une base
- `Statement` : instruction SQL
- `PreparedStatement` : instruction SQL paramétrée
- `CallableStatement` : procédure stockée dans la base
- `ResultSet` : ensemble de tuples retourné par une requête SQL
- `ResultSetMetaData` : description des tuples récupérés
- `DatabaseMetaData` : informations sur la base de données

Traitement d'un ordre SQL

Le scénario pour traiter un ordre SQL avec JDBC est le suivant :

1. Importer l'API.
2. Enregistrer le pilote JDBC (Optionnel depuis JDBC 4.0).
3. Connexion à la base de données.
4. Création d'une instruction SQL.
5. Exécution de la requête.
6. Traitement de l'ensemble des résultats.
7. Libération des ressources et fermeture de la connexion.

Étant donné que chaque étape est susceptible de rencontrer des erreurs, il faut rajouter une étape de gestion des exceptions.

Traitement d'une requête SQL

```
import java.sql.*;// Import de l'API
public class App {
  static String req="SELECT ID, NAME FROM ARTIST";
  public static void main(String[] args){
    try {
      //Connexion a la BD
      Connection conn = DriverManager.getConnection("jdbc:
          oracle:...","user","pwd");
      //Creation d'une instruction
      Statement stmt = conn.createStatement();
      //Execution de la requete
      ResultSet rset = stmt.executeQuery(req);
      //Traitement des resultats
      while (rset.next()){
        System.out.println(rset.getString("NAME"));
      }
      stmt.close();//Liberation des ressources
      conn.close();//Deconnexion de la BD
    } catch (SQLException e) {
      e.printStackTrace();//Arggg!!!
    }
  }
}
```

Interface `Driver`

- La méthode `connect()` de `Driver`
  - Prend en paramètre une URL vers la BD.
  - Retourne un objet `Connection` qui permettra de créer des requêtes.
- URL pour accéder à la base (syntaxe dépend du SGBD cible)
  - `jdbc:<sous-protocole>:<nom-BD>?param=valeur, ...`
  - sous-protocole : `oracle:thin`
  - nom-BD : `@//allegro.iut.univ-aix.fr:1522/orcl`
  - Par exemple :
    - `jdbc:oracle:thin:@//allegro:1522/orcl`
    - `jdbc:derby://localhost:1527/beeDB`
    - `jdbc:mysql://localhost:3306/FloyDB`

Étape 2 : enregistrer le pilote

Cette étape est automatique depuis la version 4.0 de JDBC. Si vous ne disposez pas d'un driver compatible l'enregistrement est manuel.

- La classe `DriverManager` a pour but de gérer les différents drivers (instances de Driver).
- Pour qu'un driver soit disponible, il faut charger sa classe en mémoire. Pour que la JVM accepte de le charger il faut que son archive *jar* soit dans le `CLASSPATH`.

Étape 2 : enregistrer le pilote

Cette étape est automatique depuis la version 4.0 de JDBC. Si vous ne disposez pas d'un driver compatible l'enregistrement est manuel.

- La classe `DriverManager` a pour but de gérer les différents drivers (instances de Driver).
- Pour qu'un driver soit disponible, il faut charger sa classe en mémoire. Pour que la JVM accepte de le charger il faut que son archive *jar* soit dans le `CLASSPATH`.

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

Étape 2 : enregistrer le pilote

Cette étape est automatique depuis la version 4.0 de JDBC. Si vous ne disposez pas d'un driver compatible l'enregistrement est manuel.

- La classe `DriverManager` a pour but de gérer les différents drivers (instances de Driver).
- Pour qu'un driver soit disponible, il faut charger sa classe en mémoire. Pour que la JVM accepte de le charger il faut que son archive *jar* soit dans le `CLASSPATH`.

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

- Cette instruction crée une instance du driver et l'enregistre auprès de la classe `DriverManager`.

Étape 3 : connexion à la base de données

Pour se connecter, on utilise la méthode `getConnection`() de `DriverManager`. Elle admet trois `String` paramètres :

- L'URL de la base de données.
- Le nom d'utilisateur.
- Le mot de passe de l'utilisateur.

Étape 3 : connexion à la base de données

Pour se connecter, on utilise la méthode `getConnection()` de `DriverManager`. Elle admet trois `String` paramètres :

- L'URL de la base de données.
- Le nom d'utilisateur.
- Le mot de passe de l'utilisateur.

```
Connection conn = DriverManager.getConnection(url, user,
pwd);
```

Étape 3 : connexion à la base de données

Pour se connecter, on utilise la méthode `getConnection()` de `DriverManager`. Elle admet trois `String` paramètres :

- L'URL de la base de données.
- Le nom d'utilisateur.
- Le mot de passe de l'utilisateur.

```
Connection conn = DriverManager.getConnection(url, user, pwd);
```

- Le `DriverManager` essaye tous les drivers enregistrés (chargés en mémoire avec `Class.forName()`) jusqu'à ce qu'il trouve un driver qui lui fournisse une instance de `Connection`.
- Les connexions sont des ressources coûteuses et longues à obtenir, il faut donc éviter d'en ouvrir une pour chaque requête à exécuter.

Étape 4 : création d'une instruction SQL

L'interface `Statement` possède les méthodes nécessaires pour réaliser les requêtes sur la base associée à la connexion dont il dépend. Il existe 3 types de `Statement` :

Étape 4 : création d'une instruction SQL

L'interface `Statement` possède les méthodes nécessaires pour réaliser les requêtes sur la base associée à la connexion dont il dépend. Il existe 3 types de `Statement` :

1. Les `Statement` : Ils permettent d'exécuter n'importe quelle requête sans paramètre. La requête est interprétée par le Sgbd au moment de son exécution. Ce type d'ordre est à utiliser principalement pour les requêtes à usage unique.

Étape 4 : création d'une instruction SQL

L'interface `Statement` possède les méthodes nécessaires pour réaliser les requêtes sur la base associée à la connexion dont il dépend. Il existe 3 types de `Statement` :

1. Les `Statement` : Ils permettent d'exécuter n'importe quelle requête sans paramètre. La requête est interprétée par le Sgbd au moment de son exécution. Ce type d'ordre est à utiliser principalement pour les requêtes à usage unique.
2. Les `PreparedStatement` : Ils permettent de précompiler un ordre avant son exécution. Ils sont particulièrement importants pour les ordres destinés à être exécutés plusieurs fois comme par exemple les requêtes paramétrées.

Étape 4 : création d'une instruction SQL

L'interface `Statement` possède les méthodes nécessaires pour réaliser les requêtes sur la base associée à la connexion dont il dépend. Il existe 3 types de `Statement` :

1. Les `Statement` : Ils permettent d'exécuter n'importe quelle requête sans paramètre. La requête est interprétée par le Sgbd au moment de son exécution. Ce type d'ordre est à utiliser principalement pour les requêtes à usage unique.
2. Les `PreparedStatement` : Ils permettent de précompiler un ordre avant son exécution. Ils sont particulièrement importants pour les ordres destinés à être exécutés plusieurs fois comme par exemple les requêtes paramétrées.
3. Les `CallableStatement` : Ils sont destinés à l'appel des procédures stockées.

Étape 4 : création d'une instruction SQL

À partir d'une instance de l'objet `Connection`, on récupère une nouvelle instance de `Statement`.

```
Statement s1=connexion.createStatement();
PreparedStatement s2=connexion.prepareStatement(req);
CallableStatement s3=connexion.prepareCall(req);
```

Étape 5 : Exécution de la requête

Afin d'exécuter une requête, il suffit de faire appel à l'une des méthodes `executeXXXX()` de l'objet `Statement` que l'on vient de créer. Il existe 3 types d'exécution :

- Les interrogation de données : on utilise la méthode `executeQuery()` en lui passant en paramètre une chaîne de caractères (`String`) contenant la requête. Cette méthode retourne un objet du type `ResultSet` contenant l'ensemble des résultats de la requête.
- Les modifications de données (INSERT, UPDATE, DELETE) : exécutés avec la méthode `executeUpdate()` qui retourne un entier correspondant au nombre de lignes impactées par la mise à jour.
- Nature inconnue, plusieurs résultats ou procédures stockées : on utilise la méthode `execute()`.

Étape 6 : Traitement de l'ensemble des résultats

- `executeQuery()` renvoie une instance de `ResultSet`
- `ResultSet` va permettre de parcourir toutes les lignes renvoyées par le SELECT
- Au début, `ResultSet` est positionné avant la première ligne et il faut donc commencer par le faire avancer à la première ligne en appelant la méthode `next()`
- Cette méthode permet de passer à la ligne suivante ; elle renvoie `true` si cette ligne suivante existe et `false` sinon.

```
ResultSet rset = stmt.executeQuery(req);
while (rset.next())
    System.out.println(rset.getString(2));
```

Différents types de `ResultSet`

Il existe quatre types de `ResultSet` :

Scroll-insensitive : Vision figée du résultat de la requête au moment de son évaluation.

Scroll-sensitive : Le `ResultSet` montre l'état courant des données (modifiées/détruites).

Read-only : Pas de modification possible (version par défaut) donc un haut niveau de concurrence.

Updatable : Possibilité de modification donc pose de verrou et faible niveau de concurrence.

```
Statement stmt = con.createStatement(
  ResultSet.TYPE_SCROLL_INSENSITIVE,
  ResultSet.CONCUR_UPDATABLE);
ResultSet rs=stmt.executeQuery("SELECT * FROM ARTIST");
```

Ce `ResultSet` est modifiable mais il ne reflète pas les modifications faites par d'autres transactions.

Déplacement dans un `ResultSet`

Depuis JDBC 3.0, on peut se déplacer à l'intérieur d'un `ResultSet`. Ci dessous sont listées les méthodes de déplacement :

- `rs.first();`
- `rs.beforeFirst();`
- `rs.next();`
- `rs.previous();`
- `rs.afterLast();`
- `rs.absolute( n );`
- `rs.relative( n );`

Les tuples de l'ensemble de résultats n'étant chargés qu'à la demande(avec un préchargement), un parcours aléatoire peut s'avérer très coûteux.

Modification d'un ResultSet

Modification :

```
rs.absolute(100);
rs.updateString("NAME", "RICHARD");
rs.updateRow();
```

Destruction :

```
rs.deleteRow();
```

Insertion de lignes :

```
rs.moveToInsertRow();
rs.updateString("NAME", "ROGER");
rs.insertRow();
rs.first();
```

Étape 6 : Traitement de l'ensemble des résultats

- Quand un `ResultSet` est positionné sur une ligne, les méthodes `getXXX` permettent de récupérer les valeurs des colonnes de la ligne :
  - `getXXX(int numeroColonne)` (position dans le SELECT de la valeur à récupérer)
  - `getXXX(String nomColonne)` (nom simple d'une colonne, pas préfixé par un nom de table ; dans le cas d'une jointure utiliser un alias de colonne)
- `XXX` désigne le type Java de la valeur que l'on va récupérer, par exemple `String`, `int` ou `double`.
- Par exemple, `getInt` renvoie un `int`.

Étape 6 : Traitement de l'ensemble des résultats

- Quand un `ResultSet` est positionné sur une ligne, les méthodes `getXXX` permettent de récupérer les valeurs des colonnes de la ligne :
  - `getXXX(int numeroColonne)` (position dans le SELECT de la valeur à récupérer)
  - `getXXX(String nomColonne)` (nom simple d'une colonne, pas préfixé par un nom de table ; dans le cas d'une jointure utiliser un alias de colonne)
- `XXX` désigne le type Java de la valeur que l'on va récupérer, par exemple `String`, `int` ou `double`.
- Par exemple, `getInt` renvoie un `int`.

```
ResultSet rset = stmt.executeQuery(req);
while (rset.next())
    System.out.println(rset.getInt(1));
```

Correspondance de types entre Java et SQL

- Tous les SGBD n'ont pas les mêmes types SQL ; même les types de base peuvent présenter des différences importantes.
- Pour cacher ces différences, JDBC définit ses propres types SQL dans la classe `Types`.
- Le driver JDBC fait la traduction de ces types dans les types du SGBD.
- Il reste le problème de la correspondance entre les types Java et les types SQL. C'est le rôle de `getXXX` et de `setXXX`.
- Par exemple, `getString` indique que l'on veut récupérer la donnée SQL dans une `String`.
- Si le `Driver` n'arrive pas à effectuer la conversion, il lève une exception.

Correspondance de types entre Java et SQL

SQL	Java
CHAR	`String`
VARCHAR	`String`
NUMERIC	`java.math.BigDecimal`
DECIMAL	`java.math.BigDecimal`
BIT	`boolean`
TINYINT	`byte`
SMALLINT	`short`
INTEGER	`int`
BIGINT	`long`
REAL	`float`
FLOAT	`double`
DOUBLE	`double`
BINARY	`byte[]`
VARBINARY	`byte[]`

Correspondance de type entre Java et SQL

SQL	Java	Explication
DATE	`java.sql.Date`	codage de la date
TIME	`java.sql.Time`	codage de l'heure
TIMESTAMP	`java.sql.TimeStamp`	codage de la date et de l'heure

Gestion des valeurs nulles

Comment reconnaître dans Java le cas d'une valeur NULL de SQL ?
Pour les méthodes `getXXX()`, la convention est la suivante :

- Les méthodes `getString()`, `getObject()`, ..., `getDate()` retournent une référence `null`.
- Les méthodes `getInt()`, `getByte()`, ..., `getShort()` retournent la valeur 0.
- La méthode `getBoolean()` renvoie la valeur `false`.

Cette convention implique que l'on ne peut pas distinguer un `NULL` d'un zéro. La méthode `wasNull()` de `ResultSet` permet de vérifier si le dernier `getXXX()` a retourné un `NULL`.

Étape 7 : Libération des ressources

- Toutes les ressources JDBC doivent être fermées dès qu'elles ne sont plus utilisées.
- Le plus souvent, la fermeture doit se faire dans un bloc `finally` pour qu'elle ait lieu quel que soit le déroulement des opérations (avec ou sans erreurs).
- Un `Statement` ou un `ResultSet` fermé ne peut plus être utilisé. La fermeture d'un `Statement` ferme automatiquement les `ResultSet` qui lui sont associés.
- La `Connection` étant une ressource coûteuse, elle doit être impérativement fermée. Si possible utilisez un *pool* de connexion.

```
rset.close();
stmt.close();
conn.close();
```

Étape 8 : Gestion des exceptions

- Une `SQLException` (ou un objet héritant de cette classe) est levée dès qu'une connexion ou un ordre SQL ne se déroule pas correctement.
  - La méthode `getMessage()` permet d'obtenir le message d'erreur en clair.
- Pour distinguer des types d'exception, JDBC a introduit 3 sous-classes de `SQLException` :
  - `SQLNonTransientException` : le problème ne peut être résolu sans une action externe ; inutile de réessayer la même action sans rien faire de spécial.
  - `SQLTransientException` : le problème peut avoir été résolu si on attend un peu avant d'essayer à nouveau.
  - `SQLRecoverableException` : l'application peut résoudre le problème en exécutant une certaine action mais elle devra fermer la connexion actuelle et en ouvrir une nouvelle.

Étape 8 : Gestion des exceptions

- Une requête SQL peut provoquer plusieurs exceptions
- On peut obtenir la prochaine exception par la méthode `getNextException()`
- Une exception peut avoir une cause ; on l'obtient par la méthode `getCause()`
- Toutes ces exceptions peuvent être parcourues par une boucle « for-each » :
  `for (Throwable e : ex) {...}`

Insertion de tuples

Pour insérer un nouveau tuple dans la BD, il suffit d'exécuter l'instruction `INSERT` de la manière suivante :

Insertion de tuples

Pour insérer un nouveau tuple dans la BD, il suffit d'exécuter l'instruction `INSERT` de la manière suivante :

```
Statement st = conn.createStatement();
int nb = st.executeUpdate(
    "INSERT INTO ARTIST(ID, NAME) " +
    "VALUES (" + id + ", '" + nom + "')"
    );
System.out.println(nb + " tuple insere");
st.close();
```

Le principe est identique pour les instructions `UPDATE` et `DELETE`. Si l'on doit exécuter un grand nombre de modifications ce genre de solution est coûteuse et difficile à mettre en œuvre.

Ordre SQL paramétrée

Les paramètres dans un ordre SQL permettent d'économiser des ressources en évitant de recompiler plusieurs fois les mêmes requêtes.

Ordre SQL paramétrée

Les paramètres dans un ordre SQL permettent d'économiser des ressources en évitant de recompiler plusieurs fois les mêmes requêtes.

```
PreparedStatement st = conn.prepareStatement(
    "UPDATE ARTIST SET AGE = ? " +
    "WHERE NAME = ? ");
for( ... ) {
    st.setInt(1, age[i]);
    st.setString(2, nom[i]);
    st.execute();
}
```

On remarquera l'utilisation de la méthode `execute()`

Traitement d'un appel de procédure stockée

Lorsqu'un traitement peut être fait directement par le SGBD, on utilisera les procédures stockées.

Traitement d'un appel de procédure stockée

Lorsqu'un traitement peut être fait directement par le SGBD, on utilisera les procédures stockées.

```
CallableStatement s = conn.prepareCall(
    "{call ma_procedure[(?,?)]}");
// fixer le type de parametre de sortie
s.registerOutParameter(2,java.sql.Types.FLOAT);
//fixer la valeur du parametre
s.setInt(1,valeur);
s.execute();
System.out.println("res = " + s.getFloat(2));
```

Ce type de traitement permet d'économiser des ressources en évitant beaucoup de transferts de données inutiles.

Gestion des transactions

Par défaut, les connections JDBC fonctionnent en mode "Auto Commit". Pour pouvoir utiliser les transactions, il faut au préalable désactiver ce mode pour la connexion courante.

- `conn.setAutoCommit(false);`
- `conn.commit();`
- `conn.rollback();`

Combinée avec une bonne gestion des erreurs, les transactions permettent de garantir l'intégrité des données.

Les Meta Informations

- JDBC permet de récupérer des informations sur le type de données que l'on vient de récupérer par un `SELECT` (interface `ResultSetMetaData`),
- mais aussi sur la base de données elle-même (interface `DatabaseMetaData`).
- Les données que l'on peut récupérer avec `DatabaseMetaData` dépendent du SGBD avec lequel on travaille.

Méta Informations sur les Result Set

```
ResultSetMetaData rsmd = rs.getMetaData();
```

Informations disponibles :

- nombre de colonnes (`getColumnCount()`),
- libellé d'une colonne (`getColumnName(int column)`),
- table d'origine (`getTableName(int column)`),
- type associé à une colonne (`getColumnType(int column)`),
- la colonne est-elle nullable ? (`isNullable(int column)`)
- etc.

Permet d'écrire un code plus générique s'adaptant à un plus grand nombre de requêtes.

Méta Informations sur les Result Set

```
ResultSet rs =
  stmt.executeQuery("SELECT * FROM ARTIST");
ResultSetMetaData rsmd = rs.getMetaData();
int nbCol = rsmd.getColumnCount();
//Affichage de la liste des colonnes
for (int i = 1; i <= nbCol; i++) {
  String typeCol = rsmd.getColumnTypeName(i);
  String nomCol = rsmd.getColumnName(i);
  System.out.println("Colonne " + i
    + " de nom " + nomCol
    + " de type " + typeCol);
}
```

Méta Informations sur la BD

```
DataBaseMetaData metaData = conn.getMetaData();
```

Informations disponibles :

- tables existantes dans la base (`getTables`()),
- nom d'utilisateur (`getUserName`()),
- version du pilote (`getJDBCMajorVersion`() et `getJDBCMinorVersion`()),
- prise en charge des jointure externes ? (`supportsOuterJoins`()),
- etc.

Ces informations permettent d'adapter les traitements JDBC aux capacité du SGBD cible. Le code devient ainsi plus portable.

Méta Informations sur la BD

```
metaData = conn.getMetaData();
String[] types = { "TABLE", "VIEW" };
String nomTables;
//Recuperation de la liste des tables
ResultSet rs =
  metaData.getTables(null, null, "%", types);
//Traitement des resultats
while (rs.next()) {
  nomTable = rs.getString(3);
  System.out.println(nomTable);
}
```

Conclusion Temporaire

- Interface pour un accès homogène aux BD :
  - Le concept de Driver masque au maximum les différences des SGBD.
  - API de bas niveau : il faut connaître SQL.
- Tous les éditeurs proposent un driver JDBC.
- Problèmes :
  - JDBC n'effectue aucun contrôle sur les instructions SQL qui sont entièrement interprétées par le SGBD.
  - Implique l'existence de SQL dispersé dans du code Java (application polyglotte).
  - Correspondance entre classes Java et relations est un travail de bas niveau.

Plan