JDBC – Java DataBase Connectivity

Ecole Centrale Paris
Cours électif IS1230

Nicolas Travers
Equipe Vertigo - Laboratoire CEDRIC
Conservatoire National des Arts & Métiers, Paris, France

JDBC

- *Java DataBase Connectivity* (JDBC) : API de bas niveau permettant de travailler avec une ou plusieurs bases de données depuis un programme Java
- Objectif : interface uniforme assurant l'indépendance du SGBDR cible
- Réalité : indépendance relative du SGBDR, l'interface étant assurée par un pilote (*driver*) fourni par l'éditeur du SGBDR ou par un tiers…
- Versions :
  - JDBC 1 : SDK 1.1 (java.sql)
  - JDBC 2 : SDK 1.2 (java.sql, javax.sql)
  - JDBC 3 : SDK 1.4

Taxonomie des pilotes (JavaSoft)

- Type 1 : JDBC-ODBC *Bridge*
  - Utilisation comme intermédiaire de *Open DataBase Connectivity* (ODBC) de Microsoft
  - JDK : **`sun.jdbc.odbc.JdbcOdbcDriver`**
  - Exige l'exécution par le client (dit « épais ») de code non Java, ne peut donc pas être utilisé par les *applets*
- Type 2 : *Native-API Partly-Java Driver*
  - Appelle des fonctions natives (non Java) de l'API du SGBDR
  - Fournis par les éditeurs des SGBDR ou par des tiers
  - Exige l'exécution par le client (dit « épais ») de code non Java, ne peut donc pas être utilisé par les *applets*

Taxonomie des pilotes (JavaSoft)

- Type 3 : *Net-protocol, all Java driver*
  - Utilise, à travers une API réseau générique, un serveur *middleware* comme intermédiaire avec le SGBDR
  - Client « léger » 100% Java, peut donc être employé par les *applets* à condition que l'adresse du *middleware* soit la même que celle du serveur Web
- Type 4 : *Native protocol, all Java driver*
  - Interagit avec le SGBDR directement à travers des *sockets*
  - Fournis par les éditeurs des SGBDR ou par des tiers
  - Client « léger » 100% Java, peut donc être employé par les *applets* à condition que l'adresse du SGBDR soit la même que celle du serveur Web

Structure de l'application Java

1. Importation de paquetages
2. Enregistrement du pilote
3. Établissement des connexions
4. Préparation des instructions SQL
5. Exécution des instructions SQL
6. Traitement des données retournées
7. Fermeture

Importation de paquetages

- Importation du paquetage de base JDBC (obligatoire) :

```
import java.sql.*;
```

- Importation de paquetages spécifiques :
  - Additions au paquetage de base :

```
import javax.sql.*;
```

  - Paquetage permettant d'utiliser des types spécifiques Oracle :

```
import oracle.sql.*;
```

  - Paquetage contenant le pilote Oracle (obligatoire) :

```
import oracle.jdbc.driver.*;
```

- `CLASSPATH` doit inclure le paquetage à employer (en fonction du pilote choisi, par exemple `ojdbc14.jar`)

Enregistrement du pilote

- Chargement de la classe du pilote, qui crée une instance et s'enregistre auprès du `DriverManager` (pour tous les types de pilotes) :

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

ou

- Création explicite d'une instance et enregistrement explicite (pilotes de type 2, 3 ou 4) :

```
DriverManager.registerDriver(new
    oracle.jdbc.driver.OracleDriver());
```


Établissement de connexion

- Connexion = instance de la classe qui implémente l'interface `Connection`
- Appel de la méthode `getConnection`(URLconnexion, *login*, *password*) de la classe `DriverManager`, avec URLconnexion = "`jdbc:`sousProtocole`:`identifiantBase"
  - `jdbc` : protocole
  - sousProtocole : `odbc` pour pilote type 1, `oracle:oci` pour pilote Oracle de type 2, `oracle:thin` pour pilote Oracle de type 4, etc.
  - identifiantBase : dépend du pilote utilisé ; exemple pour pilote de type 4 : nom de la machine (ou adresse IP) + numéro de port + nom de la base

Établissement de connexion (2)

- Exemple :

```
Connection nomConnexion =
    DriverManager.getConnection
    ("jdbc:oracle:thin:@odessa:1521:NFA011",
     "Julien", "monpass");
```

  - Pilote léger de type 4 `oracle:thin`
  - Nom machine `odessa`
  - Numéro de port `1521`
  - Nom de la base `NFA011`
- `DriverManager` essaye tous les drivers, respectant le sous-protocole indiqué, qui se sont enregistrés (dans l'ordre d'enregistrement) et utilise le premier qui accepte la connexion

Options d'une connexion

- Après l'ouverture d'une connexion on peut préciser des options :
  - Lecture seulement ou non : méthode `setReadOnly(boolean)` de `Connection`

```
nomConnexion.setReadOnly(true);
```

  - *Commit* automatique ou non : méthode `setAutoCommit(boolean)` de `Connection` (voir transactions plus loin)

```
nomConnexion.setAutoCommit(false);
```

  - Degré d'isolation : méthode `setTransactionIsolation(...)` de `Connection`

Passage des appels SQL

- Pour transmettre un appel SQL il faut commencer par créer une instance de classe qui implémente l'interface correspondante
- Interfaces utilisables pour les appels SQL :
  - Interface **`Statement`** : pour les instructions SQL simples
  - Interface **`PreparedStatement`** : pour les instructions SQL paramétrées (mais peut servir pour les instructions simples)
  - Interface **`CallableStatement`** : pour les procédures ou fonctions cataloguées (PL/SQL ou autres)

Interface `Statement`

- Création :

```
Statement stmt = connexion.createStatement();
```

- Méthodes :
  - **`ResultSet executeQuery(String)`** : exécute la requête présente dans le **`String`** et retourne un ensemble d'enregistrements (**`ResultSet`**) ; utilisée pour **`SELECT`**
  - **`int executeUpdate(String)`** : exécute la requête présente dans le **`String`** et retourne le nombre d'enregistrements traités (ou 0 pour instructions du LDD) ; utilisée pour **`INSERT`**, **`UPDATE`**, **`DELETE`** (LMD), **`CREATE`**, **`ALTER`**, **`DROP`** (LDD)

Interface `Statement` (2)

- Méthodes (suite) :
  - **`boolean execute(String)`** : exécute la requête présente dans le **`String`**, retourne **`true`** si c'est un **`SELECT`** et **`false`** sinon ; employée dans des cas particuliers
  - **`Connection getConnection()`** : retourne la connexion correspondante
  - **`void setMaxRows(int)`** : borne supérieure sur le nombre d'enregistrements à extraire par toute requête de l'instance
  - **`int getUpdateCount()`** : nombre d'enregistrements affectés par la dernière instruction SQL associée (-1 si **`SELECT`** ou si l'instruction n'a affecté aucun enregistrement)
  - **`void close()`** : fermeture de l'instance

Curseurs statiques

- Le résultat d'une requête est disponible dans une instance de classe qui implémente l'interface **`ResultSet`** (équivalent des curseurs PL/SQL)
- Méthodes de **`ResultSet`** :
  - **`boolean next()`** : positionnement sur l'enregistrement suivant ; retourne **`false`** quand il n'y a plus d'enregistrements
  - **`getXXX(int)`** : retourne la colonne de numéro donné par l'argument **`int`** et de type **`XXX`** de l'enregistrement courant
  - **`updateXXX(int, XXX)`** : dans l'enregistrement courant, donne à la colonne de numéro **`int`** et de type **`XXX`**, une valeur de type **`XXX`**
  - **`void close()`** : fermeture de l'instance
- L'instance est automatiquement fermée quand le **`Statement`** correspondant est fermé ou associé à une autre instruction SQL

Curseurs statiques : exemple

```
…
int delCount;
Statement stmt1 = connexion.createStatement();
Statement stmt2 = connexion.createStatement();
ResultSet rset = stmt1.executeQuery("SELECT Nom
                                     FROM pilote");
while (rset.next())
  System.out.println(rset.getString(1));
rset.close();
stmt1.close();
delCount = stmt2.executeUpdate("DELETE FROM vol
                  WHERE Ville_depart = 'Paris'");
stmt2.close();
…
```


Curseurs navigables

- Les options du curseur sont déclarées comme paramètres de la méthode `createStatement` :
  `createStatement(int typeRSet, int modifRSet)`
- Types possibles (selon paramètre `typeRSet`) :
  - **`ResultSet.TYPE_FORWARD_ONLY`** : non navigable (valeur par défaut)
  - **`ResultSet.TYPE_SCROLL_INSENSITIVE`** : navigable mais insensible aux modifications, c'est à dire ne reflète pas les modifications de la base
  - **`ResultSet.TYPE_SCROLL_SENSITIVE`** : navigable et sensible aux modifications (reflète les modifications de la base)

Curseurs navigables (2)

- Quelques méthodes spécifiques :
  - **`int getType()`** : retourne le type de navigabilité du curseur
  - **`void setFetchDirection(int)`** : définit la direction du parcours
    - valeurs du paramètre : **`ResultSet.FETCH_FORWARD`**, **`ResultSet.FETCH_BACKWARD`**, **`ResultSet.FETCH_UNKNOWN`**
    - est aussi méthode de **`Statement`**, ayant dans ce cas effet sur tous les curseurs associés !
  - **`int getFetchDirection()`** : retourne la direction courante
  - **`boolean isBeforeFirst()`** : indique si le curseur est positionné avant le premier enregistrement (**`true`** après ouverture, sauf si aucun enregistrement)

Curseurs navigables (3)

- Quelques méthodes spécifiques (suite) :
  - **`void beforeFirst()`** : positionne le curseur avant le premier enregistrement
  - **`boolean isFirst()`** : indique si le curseur est positionné sur le premier enregistrement ; si aucun enregistrement : **`false`**
  - **`boolean absolute(int)`** : positionne le curseur sur l'enregistrement de numéro indiqué (depuis début si >0, depuis fin si <0) ; **`false`** si aucun enregistrement n'a ce numéro
  - **`boolean relative(int)`** : positionne le curseur sur le *n*-ième enregistrement en partant de la position courante (>0 ou <0) ; **`false`** si aucun enregistrement n'a cette position

Curseurs navigables : exemple

```
…
Statement stmt = connexion.createStatement
            (ResultSet.TYPE_SCROLL_INSENSITIVE,
             ResultSet.CONCUR_READ_ONLY);
ResultSet rset = stmt.executeQuery("SELECT Nom FROM
      pilote");
if(rset.absolute(5)) {
  System.out.println("5ème pilote :" +
                          rset.getString(1));
  if(rset.relative(2))
      System.out.println("7ème pilote :" +
      rset.getString(1));
  else
      System.out.println("Echec, pas de 7ème
  pilote !");
} else
  System.out.println("Echec, pas de 5ème pilote !");
…
```


Curseurs modifiables

- Permettent de modifier le contenu de la base
- Types possibles (selon paramètre `modifRSet`) :
  - **`ResultSet.CONCUR_READ_ONLY`** : n'autorise pas la modification (valeur par défaut)
  - **`ResultSet.CONCUR_UPDATABLE`** : autorise la modification
- Contraintes d'utilisation :
  - Pas de verrouillage automatique comme avec **`CURSOR … IS SELECT … FOR UPDATE`** dans PL/SQL !
  - Seules les requêtes qui extraient des colonnes sont autorisées à produire des curseurs modifiables (**`SELECT tableau.*`** plutôt que **`SELECT *`**, pas de **`AVG()`**, …) ; aussi, pas de jointure dans la requête !

Curseurs modifiables (2)

- Quelques méthodes spécifiques :
  - **`int getConcurrency()`** : retourne la valeur du paramètre d'autorisation des modifications
  - **`void deleteRow()`** : suppression enregistrement courant
  - **`void updateRow()`** : propagation à la table des modifications de l'enregistrement courant
  - **`void cancelRowUpdates()`** : annulation des modifications de l'enregistrement courant
  - **`void moveToInsertRow()`** : préparation du curseur pour insertion d'un enregistrement
  - **`void insertRow()`** : propagation à la table de l'insertion
  - **`void moveToCurrentRow()`** : retourne à l'enregistrement courant

Curseurs modifiables : exemple 1

```
// Exemple de suppression
…
connexion.setAutoCommit(false);
Statement stmt = connexion.createStatement
                  (ResultSet.TYPE_FORWARD_ONLY,
                   ResultSet.CONCUR_UPDATABLE);
ResultSet rset = stmt.executeQuery("SELECT Nom FROM
                                        pilote");
String nomsAEffacer = "Philippe";
while(rset.next()) {
  if(rset.getString(1).equals(nomsAEffacer)) {
      rset.deleteRow();
      connexion.commit(); // valide une suppression
  }
}
// connexion.commit(); // regroupe les suppressions
rset.close();
```

Curseurs modifiables : exemple 2

```
// Exemple d'insertion
…
Statement stmt = connexion.createStatement
            (ResultSet.TYPE_SCROLL_INSENSITIVE,
             ResultSet.CONCUR_UPDATABLE);
ResultSet rset = stmt.executeQuery("SELECT
  Matricule,                    Nom, Ville, Age,
  Salaire FROM pilote");
if(rset.absolute(3))
  System.out.println(rset.getString(2));
rset.moveToInsertRow();
rset.updateInt(1,3);
rset.updateString(2,"Philippe");
rset.updateString(3,"Paris");
rset.updateInt(4,36);
rset.updateFloat(5,38000);
rset.insertRow();    // demande l'insertion d'une
  ligne
connexion.commit(); // valide l'insertion
rset.moveToCurrentRow();
…
```


Curseurs modifiables : exemple 3

```
// Exemple de modification
…
Statement stmt = connexion.createStatement
            (ResultSet.TYPE_SCROLL_INSENSITIVE,
             ResultSet.CONCUR_UPDATABLE);
ResultSet rset = stmt.executeQuery("SELECT Nom FROM
                                    pilote");
while(rset.next()) {
  if(rset.getString(1).equals("Philipe")) {
      rset.updateString(1,"Philippe");
      rset.updateRow(); // demande la modification
  }
}
connexion.commit(); // valide la modification
rset.close();
```

Interface `PreparedStatement`

- Pourquoi : meilleure efficacité (analyse + compilation + planification une seule fois, nombreuses exécutions)
- Création :

```
PreparedStatement prepStmt =
    connexion.prepareStatement(String instrSQL);
```

- `PreparedStatement` hérite de `Statement`
- Méthodes :
  - **`ResultSet executeQuery()`** : exécute la requête préparée et retourne un ensemble d'enregistrements (**`ResultSet`**) ; pour **`SELECT`**
  - **`int executeUpdate()`** : exécute la requête préparée et retourne le nombre d'enregistrements traités (ou 0 pour instructions du LDD) ; pour **`INSERT`**, **`UPDATE`**, **`DELETE`** (LMD), **`CREATE`**, **`ALTER`**, **`DROP`** (LDD)

Interface `PreparedStatement` (2)

- Méthodes (suite) :
  - **`boolean execute()`** : exécute la requête préparée, retourne **`true`** si c'est un **`SELECT`** et **`false`** sinon ; employée dans des cas particuliers
  - **`Connection getConnection()`** : retourne la connexion correspondante
  - **`void setMaxRows(int)`** : borne supérieure sur le nombre d'enregistrements à extraire par toute requête de l'instance
  - **`int getUpdateCount()`** : nombre d'enregistrements traités par la dernière instruction SQL associée (-1 si **`SELECT`** ou si l'instruction n'a affecté aucun enregistrement)
  - **`void close()`** : fermeture de l'instance

`PreparedStatement` : exemple 1

```
…
String instrSQL = "SELECT Nom FROM pilote";
PreparedStatement prepStmt1 =

    connexion.prepareStatement(instrSQL);
ResultSet rset = prepStmt1.executeQuery();
while (rset.next())
  System.out.println(rset.getString(1));
rset.close();
prepStmt1.close();
…
```


Paramétrage `PreparedStatement`

- Dans la chaîne de caractères qui représente l'instruction SQL on indique par « ? » les champs paramétrés
- Avant l'exécution il faut donner des valeurs aux « paramètres » par des méthodes `setXXX` de `PreparedStatement` : `prepStmt.setXXX(numeroPar, valeurPar)`, où `numeroPar` est la position du « ? » correspondant et `XXX` est le type du « paramètre »
- Donner la valeur `NULL` à un « paramètre » : `prepStmt.setNull(numeroPar, typePar)`

`PreparedStatement` : exemple 2

```
...
int insCount;
String instrSQL = "INSERT INTO avion
                              VALUES(?,?,?,?)";
PreparedStatement prepStmt2 =

     connexion.prepareStatement(instrSQL);
prepStmt2.setInt(1,210);
prepStmt2.setInt(2,570);
prepStmt2.setString(3,"A800");
prepStmt2.setString(4,"Roissy");
insCount = prepStmt2.executeUpdate();
prepStmt2.close();
...
```


Interface `CallableStatement`

- Objectif : appeler des procédures ou fonctions stockées écrites en PL/SQL ou un autre langage
- Création :

```
CallableStatement callStmt =
    connexion.prepareCall(String prepCall);
```

- `CallableStatement` hérite de `PreparedStatement`
- Méthodes :
  - `boolean execute()` : voir `PreparedStatement`
  - `Void registerOutParameter(int,int)` : définit un paramètre de sortie de la procédure appelée ; le premier `int` indique le numéro du paramètre lors de l'appel, le second indique le type du paramètre (suivant `java.sql.Types`)

Interface `CallableStatement` (2)

- Méthodes (suite) :
  - **`void setXXX(int, XXX)`** : donner une valeur à un paramètre ; **`int`** est la position du paramètre et **`XXX`** son type
  - **`XXX getXXX(int)`** : extraire la valeur d'un paramètre de sortie (**`OUT`**) ; **`int`** est la position du paramètre
  - **`boolean wasNull()`** : détermine si la valeur du dernier paramètre de sortie extrait est **`NULL`** ; utilisable après un **`get`**
  - **`void close()`** : fermeture de l'instance

Formats des appels

- Format du **`String`** pour un appel de procédure stockée :
  "`{call nomProcedure(?,...)}`"

- Format du **`String`** pour un appel de fonction stockée :
  "`{? = call nomFonction(?,...)}`"
  - Pour une fonction, la valeur retournée est vue comme paramètre 1 : déclaré avec **`registerOutParameter`**, récupéré avec **`getXXX`**

CallableStatement : exemple 1

- Procédure PL/SQL appelée :

```
PROCEDURE
    accordPrime(villePrime IN pilote.Ville%TYPE,
                valPrime IN NUMBER,
                nbPilotes OUT INTEGER);
```

- Appel en Java :

```
String prepCall = "{call accordPrime(?,?,?)}";
CallableStatement callStmt =
                  connexion.prepareCall(prepCall);
callStmt.setString(1,"Paris");
callStmt.setInt(2,500);
callStmt.registerOutParameter(3,

    java.sql.Types.INTEGER);
callStmt.execute();
nbPilotePrimes = callStmt.getInt(3);
callStmt.close();
```


CallableStatement : exemple 2

- Fonction PL/SQL appelée :

```
FUNCTION moyInt(ageInf IN pilote.Age%TYPE,
                ageSup IN pilote.Age%TYPE)
    RETURN pilote.Salaire%TYPE;
```

- Appel en Java :

```
String prepCall = "{? = call moyInt(?,?)}";
CallableStatement callStmt =
                  connexion.prepareCall(prepCall);
callStmt.registerOutParameter(1,java.sql.Types.NUMBER);
callStmt.setInt(2,38);
callStmt.setInt(3,55);
callStmt.execute();
moyenneSalaires = callStmt.getInt(1); // résultat
callStmt.close();
```

Méta-données

- Objectif : retrouver les propriétés d'une base de données et de ses tables
- Interface `DatabaseMetaData` : retrouver l'identification de l'éditeur et de la version, la description des tables présentes, utilisateurs connectés, etc.
- Interface `ResultSetMetaData` : pour les tables auxquelles des `Statement` ou `PreparedStatement` ont accédé, retrouver les nombres, noms, types et autre caractéristiques des colonnes

Méthodes de `DatabaseMetaData`

- `String getDatabaseProductName()` : retourne le nom de l'éditeur du SGBD qui a servi à créer la base
- `String getDatabaseProductVersion()` : retourne le numéro de version du SGBD
- `ResultSet getTables(String, String, String, String[])` : retourne une description de toutes les tables de la base
- `String getUserName()` : retourne le nom de l'utilisateur connecté
- `boolean supportsSavepoints()` : retourne `true` si la base supporte les points de validation pour les transactions

Méthodes de `ResultSetMetaData`

- `int getColumnCount()` : retourne le nombre de colonnes de la table
- `String getColumnName(int)` : retourne le nom de la colonne `int`
- `int getColumnType(int)` : retourne le type de la colonne `int` (suivant `java.sql.Types`)
- `String getColumnTypeName(int)` : retourne le nom du type de la colonne `int`
- `int isNullable(int)` : indique si la colonne `int` accepte des valeurs `NULL`
- `int getPrecision(int)` : indique le nombre de chiffres après la virgule pour la colonne `int`

Exceptions

- Classe `SQLException` qui hérite de la classe Java `Exception`
- Méthodes de `SQLException` :
  - `String getMessage()` : retourne le message décrivant l'erreur
  - `String getSQLState()` : retourne le code d'erreur SQL standard
  - `int getErrorCode()` : retourne le code d'erreur SQL du SGBD
  - `SQLException getNextException()` : retourne l'exception suivante si plusieurs ont été levées

Exceptions : exemple

```
...
try {
  Connection connexion =
          DriverManager.getConnection(...);
  String instrSQL = "INSERT INTO avion
                     VALUES(?,?,?,?)";
  PreparedStatement prepStmt =
     connexion.prepareStatement(instrSQL);
  ...
  prepStmt.executeUpdate();
  prepStmt.close();
} catch (SQLException excSQL) {
  while (excSQL != NULL) {
     System.err.println(excSQL.getMessage());
     excSQL = excSQL.getNextException();
  }
} ...
...
```


Transactions

- Gestion des transactions avec JDBC :
  - Par défaut : chaque instruction SQL exécutée constitue une transaction (**`commit`** implicite) ; ce mode peut être désactivé avec **`nomConnexion.setAutoCommit(false)`**; quand ce mode est désactivé, l'exécution d'une instruction du LDD ou la fermeture d'une connexion valident implicitement la transaction
  - Explicite : les méthodes **`commit`** et **`rollback`** de **`Connection`** doivent être utilisées

Transactions : points de validation

- Objectif : rendre possible l'annulation d'une partie des opérations (à partir de JDBC 3.0)
- Méthodes correspondantes de `Connection` :
  - `Savepoint setSavepoint("NomPoint")` : insertion point de validation intermédiaire ; si anonyme, "`NomPoint`" est absent
  - `void releaseSavepoint("NomPoint")` : supprime le point de validation intermédiaire
  - `void rollback(nomPoint)` : retour à l'état d'avant `nomPoint`
- Méthodes de l'interface `Savepoint` :
  - `int getSavepointId(Savepoint)` : retourne l'identifiant (entier) du point (pour les points anonymes)
  - `String getSavepointName(Savepoint)` : retourne le nom du point (vide si le point est anonyme)

Transactions : exemple

```
…
connexion.setAutoCommit(false);
int insCount;
String instrSQL = "INSERT INTO avion VALUES(?, 250,
                  "A400", "Garches")";
PreparedStatement prepStmt =
           connexion.prepareStatement(instrSQL);
prepStmt.setInt(1,210);
insCount = prepStmt.executeUpdate();
prepStmt.setInt(1,211);
insCount = prepStmt.executeUpdate();
connexion.commit();
prepStmt.close();
…
```

Procédures et fonctions stockées Java

- Procédure/fonction stockée Java = méthode compilée (*byte-code*), stockée avec la base et exécutée par la JVM
- Étapes :
  1. Programmation de la classe qui contient la méthode visée :

```
public class NomClasse {
…
  public static TypeJava nomMethode(paramètres) {
          …
  }
  …
}
```

  2. Compilation de la classe (vérifier d'abord CLASSPATH)

```
javac NomClasse.java
```

  3. Chargement dans la base de la ressource Java contenant la classe :

```
loadjava –user nom/motdepasse NomClasse.class
```

(on peut charger aussi des archives `.jar`, des fichiers sources `.java`)

Procédures et fonctions stockées Java

- Étapes (suite) :
  4. Publication de la méthode Java comme une procédure ou fonction stockée PL/SQL :

```
CREATE [OR REPLACE]
{ FUNCTION nomFonction (paramètres) RETURN TypeSQL
| PROCEDURE nomProcedure (paramètres) }
{ IS | AS } LANGUAGE JAVA
NAME 'NomClasse.nomMethode(paramètres) [return
TypeJava]';
```

  5. Appel de la méthode :
     - À partir de l'interface SQL*Plus :

```
VARIABLE nomVariableGlobale TypeSQL;
SET SERVEROUTPUT ON SIZE 10000
CALL DBMS_JAVA.SET_OUTPUT(10000);
CALL nomFonction(paramètres)
INTO :nomVariableGlobale;
CALL nomProcedure(paramètres);
```

Procédures et fonctions stockées Java

- À partir de SQL (pour les fonctions) :

```
SELECT ... FROM ...
        WHERE nomColonne = nomFonction(paramètres);
```

- Comme un déclencheur :

```
CREATE [OR REPLACE] TRIGGER nomDeclencheur
{BEFORE | AFTER | INSTEAD OF}
{DELETE | INSERT | UPDATE [OF colonne 1, ...] [OR
...]}
ON {nomTable | nomVue} [FOR EACH ROW]
BEGIN
        nomProcedure(paramètres);
END;
```

- À partir d'un programme PL/SQL : comme toute fonction ou procédure cataloguée
- À partir d'un programme Java : avec `CallableStatement` (comme toute fonction ou procédure cataloguée)

Procédures et fonctions stockées Java

- Si un paramètre de la procédure/fonction Java stockée est déclaré `OUT` ou `IN OUT`, il doit correspondre en Java à un tableau à 1 élément (exemple : `float[] table;`) et la valeur transmise est celle d'indice 0 (`table[0]`)
- Communication de la procédure/fonction Java stockée avec la base :
  - Par défaut, pilote JDBC « interne » :

```
Connection connexion =
                DriverManager.getConnection
                        ("jdbc:default:connection:");
```

  - D'autres connexions peuvent être établies avec un autre schéma que celui de l'utilisateur appelant la procédure (ou avec une autre base), en utilisant explicitement d'autres pilotes

Procédures et fonctions externes Java

- Procédure/fonction externe Java = méthode compilée (*byte-code*), exécutée par la JVM, mais non stockée avec la base
- Étapes :
  1. Programmation de la classe qui contient la méthode visée
  2. Compilation de la classe, le résultat étant placé dans un répertoire **`repertoireClasse`** (en général externe aux répertoires Oracle)
  3. Création d'une librairie :

```
CREATE DIRECTORY repProcExternes AS
'repertoireClasse';
```

  4. Chargement de la classe :

```
CREATE JAVA CLASS USING BFILE(repProcExternes,
                                        'NomClasse.c
lass');
```

  5. Publication : comme pour une procédure/fonction stockée Java
  6. Appel : comme pour une procédure/fonction stockée Java

Insuffisances de JDBC

- API de bas niveau, qui exige une bonne connaissance de SQL et de la base avec laquelle il faut travailler
- Aucun contrôle avant exécution pour
  - La validité syntaxique des instructions SQL transmises
  - La bonne correspondance entre ces instructions et la structure des tables

  ⇒ mise au point difficile des programmes…
- Indépendance relative du SGBDR utilisé

SQLJ

- Principe : API au-dessus de JDBC, qui permet l'inclusion directe d'instructions SQL dans le code Java et, grâce à un pré-compilateur, les traduit en appels à des méthodes JDBC
- Le pré-compilateur assure également la vérification de la validité des instructions SQL (par rapport à un SGBDR particulier, chaque SGBDR aura donc son pré-compilateur SQLJ… mais le code SQLJ sera plus portable !)
- Contrainte : les instructions SQL utilisées doivent être connues lors de l'écriture du programme (alors que JDBC permet de les construire dynamiquement) ; Oracle propose une solution propriétaire qui évite cette contrainte

Environnement et connexions

- Environnement : `CLASSPATH` doit inclure
  - Pré-compilateur : **`Oracle_home/sqlj/lib/translator.jar`** (ou **`.zip`**)
  - JDK : **`Oracle_home/sqlj/lib/classes11.jar`** (ou **`.zip`**), **`Oracle_home/sqlj/lib/classes12.jar`** (ou **`.zip`**)
  - Pilote JDBC : **`Oracle_home/sqlj/lib/runtime11.jar`** (ou **`.zip`**), **`Oracle_home/sqlj/lib/runtime12.jar`** (ou **`.zip`**)
- Connexion : fichier `connect.properties`

```
sqlj.url = jdbc:oracle:thin:@odessa:1521:NFA011
sqlj.user = Julien
sqlj.password = monpass
```

Introduction de SQL dans Java

- Introduction d'instructions SQL :

```
#sql{CREATE TABLE avion (Numav INTEGER, Capacite
     INTEGER, Type VARCHAR2, Entrepot VARCHAR2)};
#sql{INSERT INTO avion VALUES (14, 25, "A400",
                                "Garches")};
```

- Introduction de blocs PL/SQL :

```
#sql{
    [DECLARE ...]
    BEGIN
          ...
    [EXCEPTION ...]
    END;
};
```


Affectation et extraction

- Affectation d'une valeur à une variable Java (traitée dans SQL comme variable hôte) :

```
#sql{SET :variableJava = expression};
```

Exemple :

```
#sql{SET :dateJava = SYSDATE};
```

- Extraction d'un seul enregistrement :

```
#sql{SELECT col1,... INTO :var1Java,...
      FROM listeTables WHERE condition};
```

Exemple :

```
#sql{SELECT Ville_arrivee INTO :villeArrivee
      FROM vol WHERE Numvol = :numVolAller};
```

Extraire plusieurs enregistrements

- Avec une instance de `ResultSet` comme variable hôte :

```
ResultSet rset;
#sql{ BEGIN
          OPEN :OUT rset FOR SELECT Ville_arrivee
                             FROM vol;
      END; };
while(rset.next())...
```

- Avec un itérateur SQLJ (exploitable directement en SQL ou en Java à travers son `ResultSet`) :

```
#sql iterator nomTypeIterateur (type1 col1,...);
nomTypeIterateur nomIterateur;
#sql nomIterateur = {SELECT ...};
...
nomIterateur.close();
```


Appels de sous-programmes

- Appel de fonction stockée :

```
#sql :variableJava =
    {VALUES(nomFonction(parametres))};
```

- Appel de procédure stockée :

```
#sql{CALL nomProcedure(parametres)};
```

- Les paramètres effectifs sont des expressions qui peuvent inclure des variables Java comme variables hôtes :

```
... :IN numVolAller, :OUT villeArrivee, ...
```

Autres API Java ↔ SGBDR

- JDO (*Java Data Objects*) : possibilité de rendre persistants (de façon transparente) des objets Java ; le support de la persistance est assuré par un SGBDR, un SGBDO, des fichiers ordinaires, etc.
- JavaBlend : correspondance automatique relationnel ↔ objet ; par exemple, classe Java ↔ table et instance de classe ↔ enregistrement
- Serveurs d'application EJB (*Enterprise Java Beans*)