

# Java JDBC

---

***Olivier Perrin***

*IUT Nancy-Charlemagne  
Département Informatique  
Université Nancy 2*

*Olivier.Perrin@loria.fr*

# Présentation

---

- Avant JDBC, il était difficile d'accéder à des bases de données SQL
  - utilisation de bibliothèques C/C++
  - utilisation d'API natives comme ODBC
- Problème majeur
  - dépendance totale avec le SGBD utilisé
- Java est arrivé
  - avantages: portabilité, distribution, couches réseau, GUI

# Objectifs de JDBC

---

- Permettre aux programmeurs Java d'écrire un code indépendant de la base de données et du moyen de connexion utilisé
- API JDBC (*Java DataBase Connectivity*) 3.0
  - interface uniforme permettant un accès homogène aux SGBD
  - simple à mettre en œuvre
  - indépendant du SGBD support
  - supportant les fonctionnalités de base du langage SQL

# Atouts

---

- Liés a Java :
  - portabilité sur de nombreux OS et sur de nombreux SGBDR (Oracle, Informix, Sybase, ..)
  - uniformité du langage de description des applications, des applets et des accès aux bases de données
  - liberté totale vis-à-vis des constructeurs

# C'est quoi JDBC

---

- Un package contenant
  - un ensemble de classes et d'interfaces
  - pour écrire des requêtes destinées aux SGBD (SQL)
- Les interfaces permettent d'utiliser JDBC
- Mais JDBC ne fournit pas les classes qui implémentent les interfaces

# API JDBC

---

- Est fournie par le package `java.sql`
  - permet de formuler et gérer les requêtes aux bases de données relationnelles
  - supporte le standard «SQL-3 Entry Level »
    - bientôt le niveau supérieur : ANSI SQL-4
  - 8 interfaces définissant les objets nécessaires
    - pour la connexion à une base distante
    - pour la création et l'exécution de requêtes SQL
    - pour la récupération et le traitement des résultats

# java.sql

---

- 8 interfaces :
  - Statement
  - CallableStatement, PreparedStatement
  - DatabaseMetaData, ResultSetMetaData
  - ResultSet
  - Connection
  - Driver

# Principe de fonctionnement

---

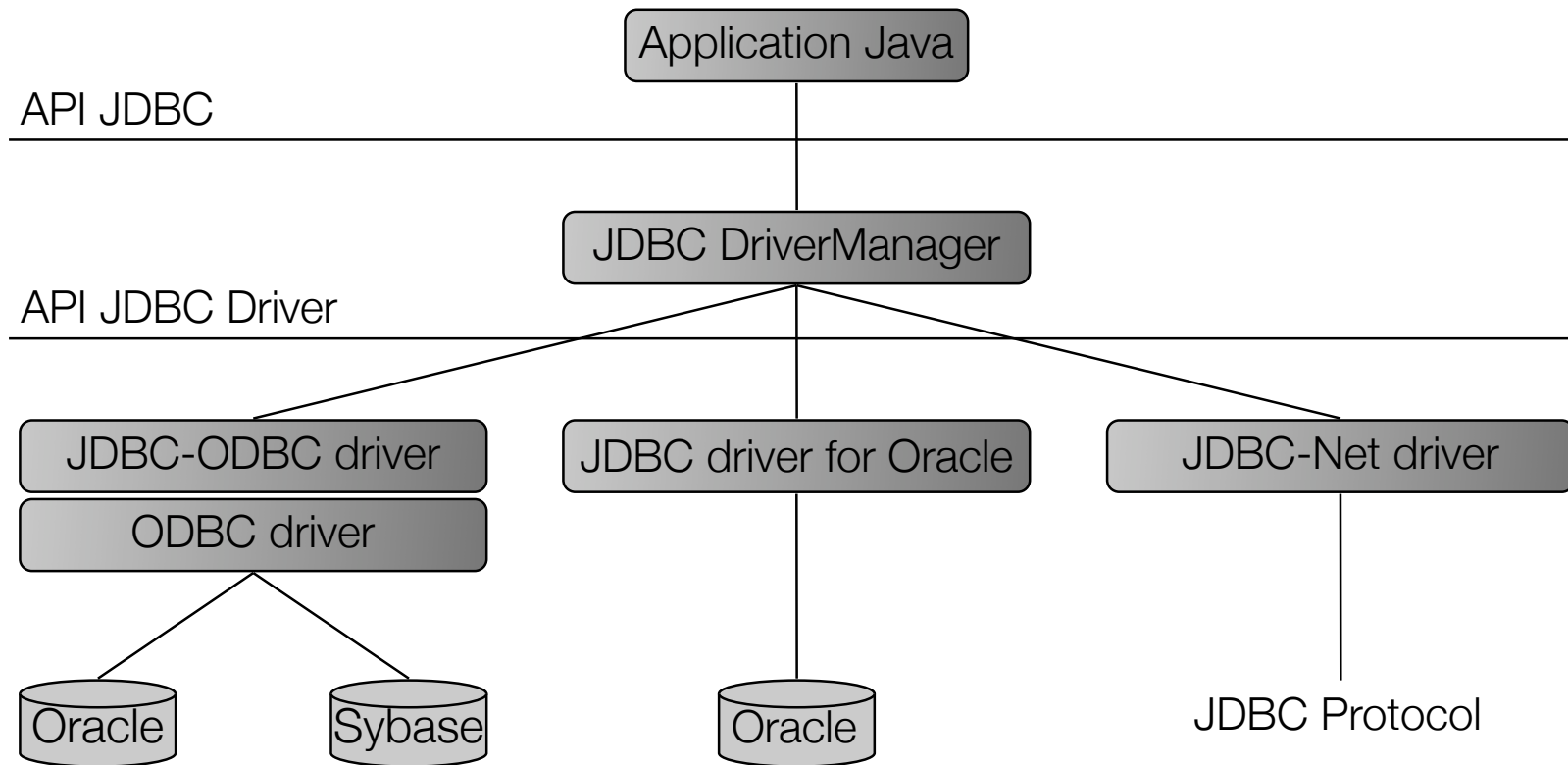
- Drivers
  - chaque SGBD utilise un pilote (driver) qui lui est propre et qui permet de convertir les requêtes JDBC dans le langage natif du SGBD
  - le driver est un ensemble de classes qui implémentent les interfaces de JDBC
  - les drivers font le lien entre le programme Java et le SGBD
  - ces drivers dits JDBC existent pour tous les principaux SGBD: Oracle, Sybase, Informix, DB2, MySQL,...

# Architecture: un modèle à deux niveaux

---

- Premier niveau: API JDBC
  - c'est la couche visible et utile pour développer des applications Java accédant à des SGBD
  - représentée par le package `java.sql`
- Les niveaux inférieurs
  - destinés à faciliter l'implantation de drivers pour des bases de données
  - représentent une interface entre les accès de bas niveau au moteur du SGBD et l'application

# Architecture



# Drivers JDBC

---

- 4 types de drivers (taxonomie de JavaSoft)
  - Type I : *JDBC-ODBC bridge driver*
    - pont JDBC-ODBC
  - Type II : *Native-API, partly-Java driver*
    - driver faisant appel à des fonctions natives non Java de l'API du SGBD
  - Type III : *Net-protocol, all-Java driver*
    - driver qui permet l'utilisation d'un middleware
  - Type IV : *Native-protocol, all-Java driver*
    - driver écrit entièrement en Java qui utilise le protocole réseau du SGBD
- Les drivers disponibles (221 au 29/11/2008)
  - <http://developers.sun.com/product/jdbc/drivers>

# Drivers et sécurité

---

- Applet
  - ne peut pas charger du code natif (non Java)
  - ne peut pas utiliser les drivers de type 1 et 2
- Applet untrusted
  - ne peut pas échanger des données par sockets avec des machines autres que celle d'où elle provient
  - contraintes sur les drivers de type 3 et 4

# Interfaces

---

- **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**: n-uplets récupérés par une instruction SQL
- **ResultSetMetaData**: description des n-uplets récupérés
- **DatabaseMetaData**: informations sur la base de données

# Classes

---

- DriverManager
  - gère les drivers, lance les connexions aux bases
- Date
  - date SQL
- Time
  - heures, minutes, secondes SQL
- TimeStamp
  - comme Time, avec une précision à la microseconde
- Types
  - constantes pour désigner les types SQL (conversions)

# Exceptions

---

- `SQLException`
  - erreurs SQL
- `SQLWarning`
  - avertissements SQL
- `DataTruncation`
  - lorsqu'une valeur est tronquée lors d'une conversion SGBD → Java

# Mise en œuvre de JDBC

---

1. Importer le package `java.sql`
2. Enregistrer le driver JDBC
3. Etablir la connexion au SGBD
4. Créer une requête (ou instruction SQL)
5. Exécuter la requête
6. Traiter les données retournées
7. Fermer la connexion

# Interface Driver

---

- La méthode `connect ( )` de `Driver`
  - nécessite un URL vers la base
  - renvoie une instance de l'interface `Connection` (`null` si le driver ne convient pas)
- L'instance de `Connection` obtenue permet de lancer 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: `@charlemagne:1521:infodb`
  - exemples
    - `String url = "jdbc:oracle:thin:@charlemagne:infodb"`
    - `String url = "jdbc:mysql://foo.loria.fr:1114:maBase"`

## Étape 2 : enregistrer le driver

---

- Rôle de la classe `DriverManager`
  - elle gère les différents drivers (instances de `Driver`)
- Pour qu'un driver soit disponible, il faut charger sa classe en mémoire
  - `Class.forName("driverName")`
  - la classe
    - crée une instance d'elle-même
    - enregistre cette instance auprès de la classe `DriverManager`

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

- Certains compilateurs refusent cette notation et demandent plutôt :

```
Class.forName("driverName").newInstance();
```

## Étape 3 : connexion à la base

---

- On utilise la méthode `getConnection()` de `DriverManager`
- 3 arguments :
  - l'URL de la base de données
  - le nom de l'utilisateur de la base
  - son mot de passe

```
Connection connexion =  
    DriverManager.getConnection(url, user, password);
```

- 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 connexion

# Connexions et threads

---

- Les connexions sont des ressources coûteuses et longues à obtenir
- On peut avoir l'idée de les réutiliser dans plusieurs threads différents
- Mais, une connexion ne peut pas être partagée par plusieurs threads
- À la place, il faut utiliser des pools de threads ou des variables de type `java.lang.ThreadLocal`
- Pour plus de détails, voir <http://java.sun.com/developer/onlineTraining/Programming/JDCBook/conpool.html>

## Étape 4: création d'un Statement (1)

---

- L'interface `Statement` possède les méthodes nécessaires pour réaliser les requêtes sur la base associée à la connexion dont il dépend
- 3 types de `Statement` :
  - `Statement` : requêtes statiques simples
  - `PreparedStatement` : requêtes dynamiques précompilées (avec paramètres d'entrée/sortie)
  - `CallableStatement` : procédures stockées

## Étape 4: création d'un Statement (2)

---

- À partir de l'instance de l'objet `Connection`, on récupère le `Statement` associé

```
Statement req1 =  
    connexion.createStatement();  
  
PreparedStatement req2 =  
    connexion.prepareStatement(str);  
  
CallableStatement req3 =  
    connexion.prepareCall(str);
```

## Étape 5: exécution d'une requête (1)

---

- 3 types d'exécution
  - consultation (requêtes de type SELECT)
    - `executeQuery()` : retourne un `ResultSet` (n-uplets résultants)
  - modification (requêtes de type INSERT, UPDATE, DELETE, CREATE TABLE, DROP TABLE)
    - `executeUpdate()` : retourne un entier (nombre de n-uplets traités)
  - nature inconnue, plusieurs résultats ou procédures stockées
    - `execute()`

## Étape 5: exécution d'une requête (2)

---

- `executeQuery()` et `executeUpdate()` de la classe `Statement` prennent comme argument une chaîne (`String`) indiquant la requête SQL à exécuter

```
Statement st = connexion.createStatement();
ResultSet rs = st.executeQuery(
    "SELECT nom, prenom FROM clients " +
    "WHERE nom='perrin' ORDER BY prenom");
int nb = st.executeUpdate("INSERT INTO dept(DEPT) "
    + "VALUES(54)");
```

Attention aux espaces

## Étape 5: exécution d'une requête (3)

---

- Deux remarques :
- le code SQL n'est pas interprété par Java.
  - c'est le pilote associé à la connexion (et au final par le moteur de la base de données) qui interprète la requête SQL
  - si une requête ne peut s'exécuter ou qu'une erreur de syntaxe SQL a été détectée, l'exception `SQLException` est levée
- le driver JDBC effectue d'abord un accès à la base pour découvrir les types des colonnes impliquées dans la requête puis un 2ème pour l'exécuter.

## Étape 6: traiter les données retournées

---

- Interface `ResultSet`
  - `executeQuery()` renvoie une instance de `ResultSet` qui permet d'accéder aux champs des n-uplets sélectionnés
  - seules les données demandées sont transférées en mémoire par le driver JDBC
  - il faut donc les lire "manuellement" et les stocker dans des variables pour un usage ultérieur

## Étape 6: résultat avec ResultSet (1)

---

- Avec JDBC 1.x
- Parcours itératif ligne par ligne
- Méthode `next ( )`
  - retourne `false` si dernier n-uplet lu, `true` sinon
  - un appel fait avancer le curseur sur le n-uplet suivant
  - au départ, le curseur est positionné avant le premier n-uplet
  - exécuter `next ( )` au moins une fois pour avoir le premier

```
while(rs.next()) { // Traitement de chaque n-uplet }
```

- Impossible de revenir au n-uplet précédent ou de parcourir l'ensemble dans un ordre non séquentiel

## Étape 6: résultat avec ResultSet (2)

---

- Avec JDBC 2.0 :
- on peut parcourir le `ResultSet` d'avant en arrière :
  - `next()` vs. `previous()`
- en déplacement absolu : aller à la n-ième ligne
  - `absolute(int row), first(), last(), ...`
- en déplacement relatif : aller à la n-ième ligne à partir de la position courante du curseur, ... :
  - `relative(int row), afterLast(), beforeFirst(), ...`

## Étape 6: résultat avec ResultSet (3)

---

- Les colonnes sont référencées par leur numéro (commencent à 1) ou par leur nom
- L'accès aux valeurs des colonnes se fait grâce à l'utilisation de méthodes de la forme `getXXX()`
  - lecture du type de données Java XXX dans chaque colonne du n-uplet courant

```
int val = rs.getInt(3) ; // accès au 3e attribut  
String prod = rs.getString("PRODUIT") ;
```

## Étape 6: résultat avec ResultSet (4)

---

- Exemple

```
Statement st = connexion.createStatement();
ResultSet rs = st.executeQuery(
    "SELECT a, b, c, FROM Table1 ");
while(rs.next()) {
    int i = rs.getInt("a");
    String s = rs.getString("b");
    byte[] b = rs.getBytes("c");
}
```

# Types de données JDBC/SQL

---

- Tous les SGBD n'ont pas les mêmes types SQL (même pour les types de base, il peut y avoir des différences importantes)
- Le driver JDBC traduit le type JDBC retourné par le SGBD en un type Java correspondant
  - le `XXX` de `getXXX()` est le nom du type Java correspondant au type JDBC attendu
  - chaque driver a des correspondances entre les types SQL du SGBD et les types JDBC
  - le programmeur est responsable du choix de ces méthodes
    - `SQLException` générée si mauvais choix

# Équivalence de types entre Java et SQL

---

- **Type JDBC/SQL (classe Type)**

CHAR, VARCHAR

LONGVARCHAR

NUMERIC, DECIMAL

BINARY, VARBINARY

LONGVARBINARY

BIT

INTEGER

BIGINT

SMALLINT

TINYINT

REAL

DOUBLE, FLOAT

DATE

TIME

TIME STAMP

- **Méthode Java**

getString()

getAsciiStream()

getBigDecimal()

getBytes()

getBinaryStream()

getBoolean()

getInt()

getLong()

getShort()

getByte()

getFloat()

getDouble()

getDate()

getTime()

getTimeStamp()

# Équivalence de types entre Java et SQL

---

- **Type JDBC/SQL (classe Type)**

ARRAY

BLOB

CLOB

REF

AUTRE

**Méthode Java**

getArray()

getBlob()

getClob ()

getRef ()

getObject()

# Cas des valeurs NULL

---

- Pour repérer les valeurs NULL de la base :
- utiliser la méthode `wasNull()` de `ResultSet`
  - renvoie `true` si l'on vient de lire un NULL, `false` sinon
- les méthodes `getXXX()` de `ResultSet` convertissent une valeur NULL SQL en une valeur acceptable par le type d'objet demandé :
  - les méthodes retournant un objet (`getString()`, `getDate()`,...) retournent un NULL Java
  - les méthodes numériques (`getByte()`, `getInt()`,...) retournent "0"
  - `getBoolean()` retourne "false"

# Cas des valeurs NULL

---

- Exemple

```
Statement st = conn.createStatement();
ResultSet rs = st.executeQuery(
    "SELECT nomE, comm FROM emp");
while (rs.next()) {
    nom = rs.getString(1);
    commission = rs.getDouble(2);
    if (rs.wasNull())
        System.out.println(nom + ": pas de comm");
    else
        System.out.println(nom + " a " +
            commission + "euros de commission");
}
```

## Étapes 7: fermer les connexions

---

- Pour terminer proprement un traitement, il faut fermer les différents espaces ouverts
  - sinon le ramasse-miettes s'en occupera mais moins efficace
- Chaque objet possède une méthode `close()` :

```
resultset.close();  
statement.close();  
connection.close();
```

```

import java.sql.*;
public class TestJDBC {
    public static void main(String args[]) throws SQLException {
        String url = "jdbc:oracle:thin:@charlemagne:1521:infodb";
        Connection con;
        Statement stmt;
        String query = "select nomski,specialite from skieur";
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            //Class.forName("com.mysql.jdbc.Driver");
        } catch (java.lang.ClassNotFoundException e) {
            System.err.print("ClassNotFoundException (try): ");
            System.err.println(e.getMessage());
        }
        try {
            con = DriverManager.getConnection(url,args[0],args[1]);
            stmt = con.createStatement();
            ResultSet rs = stmt.executeQuery(query);
            System.out.println("Skieurs avec leur specialite:");
            while (rs.next()) {
                String s = rs.getString("nomski");
                String n = rs.getString("specialite");
                System.out.println(s + "    " + n);
            }
            stmt.close();
            con.close();
        } catch (SQLException ex) {
            System.err.println("SQLException: " + ex.getMessage());
        }
    }
}

```

# Accès aux méta-données

---

- 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 elle-même (interface `DatabaseMetaData`)
- Interface `ResultSetMetaData`
  - méthode `getMetaData()` permet d'obtenir des informations sur les types de données du `ResultSet`
  - elle renvoie des instances de `ResultSetMetaData`
  - on peut connaître entre autres :
    - le nombre de colonne : `getColumnCount()`
    - le nom d'une colonne : `columnName(int col)`
    - le nom de la table : `tableName(int col)`
    - si un NULL SQL peut être stocké dans une colonne : `isNullable()`

# ResultSetMetaData

---

```
ResultSet rs = st.executeQuery(
    "SELECT * FROM emp");
ResultSetMetaData rsmd = rs.getMetaData();
int nbColonnes = rsmd.getColumnCount();
for (int i = 1; i <= nbColonnes; i++) {
    String typeColonne = rsmd.getColumnTypeName(i);
    String nomColonne = rsmd.getColumnName(i);
    System.out.println("Colonne " + i
        + " de nom " + nomColonne
        + " de type " + typeColonne);
}
```

# DatabaseMetaData

---

- Interface `DatabaseMetaData`
  - informations sur la base de données
  - méthode `getMetaData()` de l'objet `Connection`
  - dépend du SGBD avec lequel on travaille
  - elle renvoie des instances de `DatabaseMetaData`
  - on peut connaître entre autres :
    - les tables de la base : `getTables()`
    - le nom de l'utilisateur : `getUserName()`
    - ...

# DatabaseMetaData

---

```
private DatabaseMetaData metaData;  
private java.awt.List listTables= new List(10);  
...  
String[] types = { "TABLE", "VIEW" };  
String nomTables;  
...  
metaData = conn.getMetaData();  
ResultSet rs =  
    metaData.getTables(null, null, "%", types);  
while (rs.next()) {  
    nomTable = rs.getString(3);  
    listTables.add(nomTable);  
}
```

# Requêtes pré-compilées

---

- La plupart des SGBD offre la possibilité de n'analyser qu'une seule fois une requête
- JDBC permet de bénéficier de cette fonctionnalité
- L'objet `PreparedStatement` envoie une requête sans paramètres à la base de données pour précompilation et spécifiera le moment voulu la valeur des paramètres
  - plus rapide qu'un `Statement` classique
    - le SGBD n'analyse qu'une seule fois la requête (recherche d'une stratégie d'exécution adéquate)
    - pour de nombreuses exécutions d'une même requête SQL avec des paramètres variables
  - tous les SGBD n'acceptent pas les requêtes précompilées

# Création d'une requête pré-compilée

---

- La méthode `prepareStatement()` de l'objet `Connection` crée un `PreparedStatement` :

```
PreparedStatement ps =  
    c.prepareStatement("SELECT * FROM Clients "+  
                      "WHERE name = ? ");
```

- les paramètres sont spécifiés par un "?"
- ils sont ensuite instanciés par les méthodes `setInt()`, `setString()`, `setDate()`...
- ces méthodes nécessitent 2 arguments (`setInt(n, valeur)`)
  - le premier (int) indique le numéro relatif de l'argument dans la requête
  - le second indique la valeur à positionner
- `setNull(n, type)` positionne le paramètre à NULL (SQL)

# Exécution d'une requête pré-compilée

---

```
PreparedStatement ps =  
    c.prepareStatement("UPDATE emp SET sal = ? "  
    + "WHERE name = ?");  
...  
for(int i = 0; i < 10; i++) {  
    ps.setFloat(1, salary[i]);  
    ps.setString(2, name[i]);  
    int count = ps.executeUpdate();  
}
```

# Avantages des PreparedStatement

---

- Traitement plus rapide si utilisés plusieurs fois avec plusieurs paramètres
- Amélioration de la portabilité car les méthodes `setXXX()` n'ont pas à tenir compte des différences entre SGBD
- Exemple: les SGBD n'utilisent pas tous les mêmes formats de date ('JJ/MM/AA' ou 'AAAA-MM-JJ') ou de chaînes de caractères (pour les caractères d'échappement)

# Les procédures stockées

---

- Rappel
  - procédures permettant de fournir la même fonctionnalité à plusieurs utilisateurs
  - procédure stockée dans la base côté serveur
- JDBC offre une interface dédiée
  - `CallableStatement`
  - dérive de l'interface `PreparedStatement`
  - gère le retour de valeur avec `ResultSet`
- Comment
  - création d'un objet `CallableStatement`
  - en utilisant `Connection.prepareCall(...)`

# Appel des procédures stockées

---

- La requête doit être formatée
  - encadrée par des accolades {...}
  - utilisation du préfixe `call`
- Trois modes pour l'appel
  - la procédure renvoie une valeur  
`{ ? = call nomProcédure(?, ?, ...) }`
  - la procédure ne renvoie aucune valeur  
`{ call nomProcédure(?, ?, ...) }`
  - si on ne lui passe aucun paramètre  
`{ call nomProcédure }`
- Passage de paramètres possible
  - comme pour `PreparedStatement`

# Exemple

---

- Sans valeur de retour

```
CallableStatement testCall;  
testCall = conn.prepareCall(  
    "{ call setSalary(?,?) }");  
testCall.setString(1, "€ EUR");  
testCall.setLong(2, 2000);  
testCall.execute();
```

# Accéder aux résultats

---

- Un `PreparedStatement` peut retourner des données grâce à un `ResultSet`
- Les procédures stockées étendent le modèle
  - appel de la procédure précédé du passage des paramètres in et out grâce aux méthodes `setXXX()`
  - type des paramètres out et in/out grâce à la méthode `registerOutParameter()`
  - exécution de la requête grâce à `executeQuery()`, `executeUpdate()` ou `execute()`
  - récupération des paramètres out et in/out grâce aux méthodes `getXXX()`

# Exemple

---

- On considère la procédure stockée

```
create or replace procedure augmentation
(unDept in integer,
 pourcentage in number,
 cout out number) is
begin
  update emp
    set sal = sal * (1 + pourcentage / 100)
    where dept = unDept;
  select sum(sal) * pourcentage / 100 into cout
    from emp
    where dept = unDept;
end;
```

## Exemple (2)

---

```
CallableStatement csmt = conn.prepareCall(
    "{ call augmentation(?, ?, ?) }");
// 2 chiffres après la virgule pour 3ème paramètre
csmt.registerOutParameter(3, Types.DECIMAL, 2);
// Augmentation de 2,5 % des salaires du dept 10
csmt.setInt(1, 10);
csmt.setDouble(2, 2.5);
csmt.executeQuery();
double cout = csmt.getDouble(3);
System.out.println("Cout total augmentation : "
    + cout);
```

# Procédures stockées et ResultSet

---

- Elles peuvent retourner n'importe quel type de données y compris des `ResultSet`
  - utiliser `executeQuery()` au lieu de `execute()`
  - récupération d'un `ResultSet` normal
    - utilisable comme d'habitude
    - permet de sauvegarder la requête SQL au niveau de la base

```
CallableStatement csmt;  
csmt = myConn.prepareCall("{ call getDetails(?) }");  
ResultSet rs;  
csmt.setLong(1, 1000000);  
rs = csmt.executeQuery();
```

# Procédures stockées

---

- Une procédure stockée peut contenir plusieurs instructions SQL
- Pour accéder à tous les résultats de ces instructions, on utilise la méthode `getMoreResults()` de la classe `Statement`
- Utilisation
  - si une procédure contient 2 instructions `SELECT`
    - `execute()`
    - `getResultSet()`
    - `getMoreResult(), getResultSet()`

## Procédures stockées (2)

---

- Si on ne connaît pas l'ordre des instructions SQL
  - `getMoreResults()` renvoie `true` si le résultat est un `ResultSet`
  - `getUpdateCount()` renvoie le nombre de n-uplets modifiés, ou `-1` s'il n'y a plus de résultats (ou si le résultat est un `ResultSet`)
- Exemple de condition d'arrêt

```
!getMoreResults() && getUpdateCount() == -1
```

# Transactions

---

- Par défaut : mode auto-commit
  - un "commit " est effectué automatiquement après chaque ordre SQL
- Pour gérer soi-même les transactions

```
conn.setAutoCommit(false);
```

- pour valider

```
conn.commit();
```

- pour annuler

```
conn.rollback();
```

## Transactions (2)

---

- Modifier le niveau d'isolation d'une transaction

```
// TRANSACTION_NONE
// TRANSACTION_READ_COMMITTED
// TRANSACTION_READ_UNCOMMITTED
// TRANSACTION_REPEATABLE_READ
// TRANSACTION_SERIALIZABLE

conn.setTransactionIsolation(
    Connection.TRANSACTION_SERIALIZABLE);
```

# Exceptions

---

- `SQLException` est levée dès qu'une connexion ou un ordre SQL ne se déroule pas correctement
  - méthode `getMessage()` pour obtenir le message en clair de l'erreur
  - renvoie également des informations spécifiques au SGBD
    - `SQLState`
    - Code d'erreur SGBD
- `SQLWarning`: avertissement SQL

# ResultSet scrollable

---

- Plusieurs possibilités pour parcourir un `ResultSet`
  - `next()`
  - `previous()`
  - `first()`
  - `last()`
  - `relative(int n)`: déplace le curseur d'un nombre donné de n-uplets
  - `absolute(int n)`: déplace le curseur au n-uplet n
  - `isFirst()`, `isLast()`, `isBeforeFirst()`,  
`isAfterLast()`, `afterLast()`, `beforeFirst()`, `getRow()`
  - `moveToCurrentRow()`: seulement avec un `ResultSet` updatable
  - `moveToInsertRow()`: seulement avec un `ResultSet` updatable

## ResultSet scrollable (2)

---

- Déclaration

```
Stmt = conn.createStatement(  
    int resultSetType,  
    int resultSetConcurrency);
```

- ResultSetType

- `ResultSetType.TYPE_FORWARD_ONLY`: défaut, identique à JDBC 1.0
- `ResultSetType.TYPE_SCROLL_INSENSITIVE`: curseur en avant, en arrière, à un endroit donné. Insensible aux modifications des données
- `ResultSetType.TYPE_SCROLL_SENSITIVE`: curseur en avant, en arrière, à un endroit donné. Les modifications faites sur la base sont visibles dans le `ResultSet`

- ResultSetConcurrency

- `ResultSet.CONCUR_READ_ONLY`: défaut, identique à JDBC 1.0
- `ResultSet.CONCUR_UPDATABLE`: permet de modifier avec le `ResultSet`

## ResultSet scrollable (3)

---

```
stmt = conn.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stmt.executeQuery(
    "select nomski, specialite from skieur");
choix = keyboard.readLine();
while (!choix.equals("q")) {
    if (choix.equals("n")) rs.next();
    else if (choix.equals("p")) rs.previous();
    else if (choix.equals("l")) rs.last();
    else if (choix.equals("f")) rs.first();
    s = rs.getString("nomski");
    n = rs.getString("specialite");
    System.out.println(s + " " + n);
    choix = keyboard.readLine();
}
```

# ResultSet updatable

---

- Permet de parcourir et mettre à jour les lignes d'un curseur (modifier, insérer, supprimer)
- Déclaration

```
Statement stmt = con.createStatement(  
    ResultSet.TYPE_SCROLL_INSENSITIVE,  
    ResultSet.CONCUR_UPDATABLE);  
// CONCUR_UPDATABLE: le curseur est en mode mise à jour
```

- Modification d'un n-uplet

```
// on modifie la colonne colname de la ligne courante  
// en lui affectant la valeur value  
uprs.updateXXX(colname, value)  
//on demande la mise à jour de la ligne courante  
uprs.updateRow()
```

# ResultSet updatable - insert

---

- Insertion de nouvelles lignes et suppression de lignes dans un ResultSet 'updatable'
- Méthodes pour insérer une ligne

```
moveToInsertRow(); // place le curseur sur la ligne d'insertion  
updateXXX(colname,value); // valeur de colname = value  
insertRow(); // insère le contenu du n-uplet dans la base
```

```
uprs.moveToInsertRow();  
uprs.updateString("NOMSKI", "Grange");  
uprs.updateString("SPECIALITE", "Slalom");  
uprs.updateString("NOMSTAT", "Tignes");  
uprs.insertRow();
```

# ResultSet updatable - delete

---

- Supprimer un n-uplet dans un `ResultSet`
  - positionner le curseur sur la ligne à supprimer en utilisant les méthodes de positionnement de curseur (`next()`, `previous()`...)
  - utiliser la méthode `deleteRow()` qui supprime la ligne dans la base

```
Statement uprs = con.createStatement(  
    ResultSet.TYPE_SCROLL_SENSITIVE,  
    ResultSet.CONCUR_UPDATABLE);  
uprs.next();  
uprs.deleteRow();
```

# JDBC et Oracle 9

---

- Environnement (à l'IUT)

```
set CLASSPATH=%CLASSPATH%;p:\SGBD2A\classes12b.zip
```

- Source Java

```
import java.sql
...
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection conniut = DriverManager.getConnection(
    "jdbc:oracle:thin:@charlemagne:1521:infodb",
    "login", "motdepasse");
```

# Synthèse sur JDBC

---

- Interface pour un accès homogène
  - 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
  - Correspondance entre classes Java et relations est un travail de bas niveau

# Et après...

---

- Java DB
  - Open Source Apache Derby DB
  - écrit en Java
- JDO Java Data Objects
  - rend transparente la persistance des objets Java
- Hibernate
  - ORM pour Java
- JPA Java Persistence API
  - le cours de la prochaine fois...