# Principes d'utilisation des systèmes de gestion de bases de données

JDBC (1/2)

Cours 7

M1 - Ecole Informatique d'Orsay

2008/09

Emmanuel Waller, LRI, Orsay

# But du cours

Lors écriture application généraliste, savoir utiliser :

- Le SGBD : outils (SQL) pour gérer les 12 problèmes BD rencontrés
- Le mode programme : outils pour gérer les 9 problèmes rencontrés lors accès au serveur BD

# Les problèmes de base de données

- modèle de données, conception, persistance, indépendance des niveaux
- contraintes d'intégrité, confidentialité
- mise à jour, interrogation
- reprise sur panne, contrôle de concurrence
- grandes quantités
- (répartition)

# compétences à acquérir

- concernant les problèmes de base de données :
  - comprendre les mécanismes au coeur de ces problèmes : pourquoi
  - Pour une application donnée :
    - Considérer chacun des 12 problèmes
    - Détecter toutes ses occurrences dans l'application : quand (ex : accès concurrents)
    - Programmer la meilleure solution possible pour chacune des occurrences : comment
  - Répartition : hors programme
- concernant les problèmes du mode programme : même chose

# compétences à acquérir

- pour un code BD donné, savoir dire :
  - exactement ce qu'il fait
  - pourquoi
- pour un cahier des charges donné, savoir :
  - concevoir la meilleure solution
  - écrire le code BD concret correspondant
  - justifier : exhiber une violation du cahier des charges pout tout autre code

# les deux parties du cours

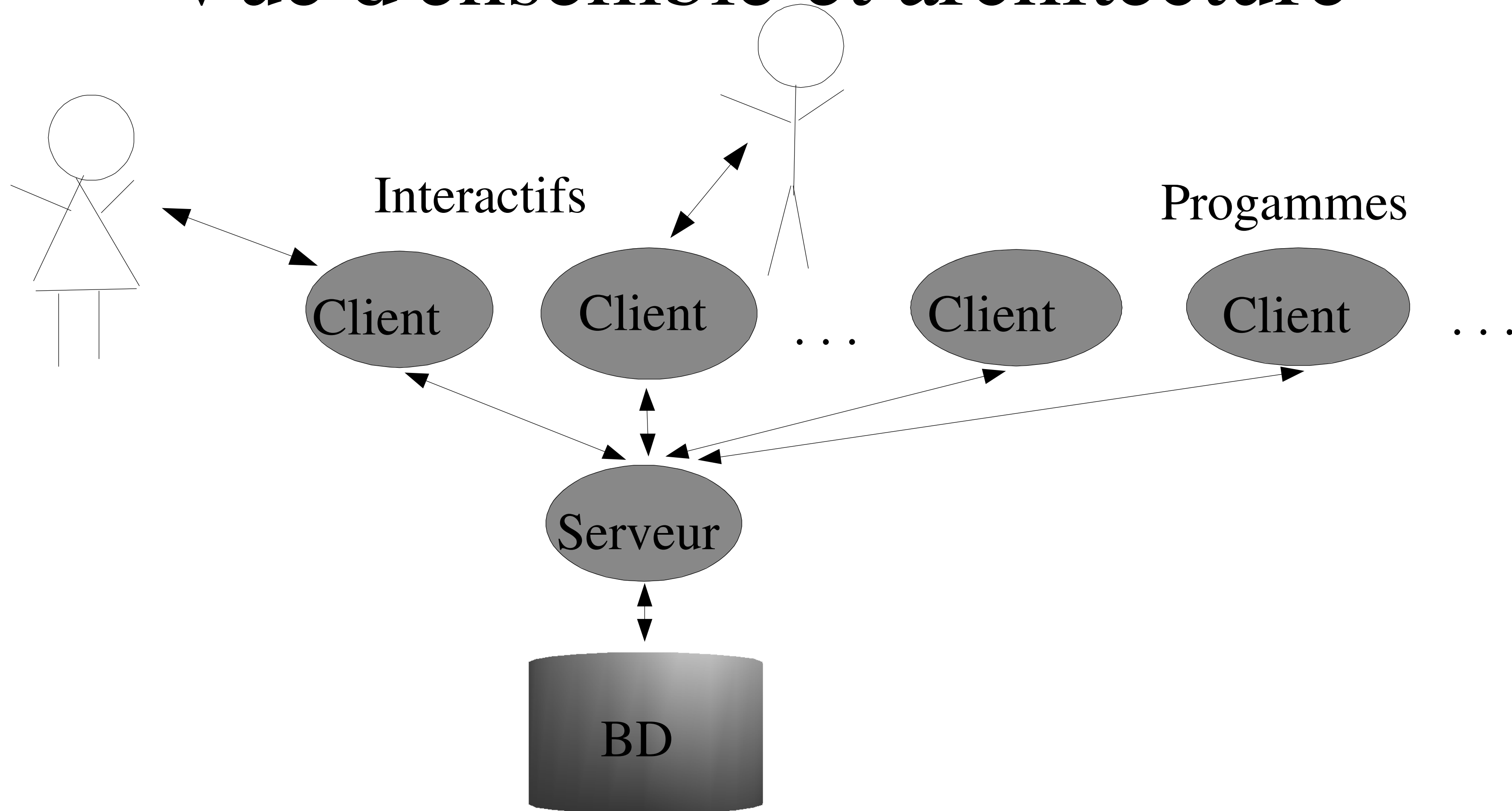
- création et gestion de la base
  - les problèmes de base de données
    - liés à la construction de la base
    - liés à la dynamique de la base
    - liés à l'interrogation de la base

    (accès à la base en mode interactif)
  - les traitements bas niveau en mode programme : PL/SQL
- accès à la base depuis un programme
  - les problèmes du mode programme étudiés à travers (outre PL/SQL) plusieurs langages généralistes : Java, Web (PHP)
- étude avec SGBD : Oracle10 (possible ex : MySQL5)

# Les problèmes du mode programme

- Interface, portabilité
- Connexion
- Envoi serveur d'ordres BD (SQL, PL/SQL)
  - Curseurs
  - Paramètres
  - SQL dynamique
- Erreurs
  - côté serveur
  - côté programme lors échange données
- Transactions

# Vue d'ensemble et architecture



# les deux parties du cours

- création et gestion de la base
  - les problèmes de base de données
    - liés à la construction de la base
    - liés à la dynamique de la base
    - liés à l'interrogation de la base

  (accès à la base en mode interactif)
  - les traitements bas niveau en mode programme : PL/SQL
- accès à la base depuis un programme
  - les problèmes du mode programme étudiés à travers (outre PL/SQL) plusieurs langages généralistes : Java, Web (PHP)
- étude avec SGBD : Oracle10 (possible ex : MySQL5)

# rappel : PL/SQL

- motivation :
  - factoriser
  - autonomie
  - expressivité
  - performances
  - architecture en couches
  - portabilité (entre environnements différents avec même version d'Oracle)

- le langage :
  - partie programmation :
    - bloc
    - types dérivés
    - tests, boucles, fonctions, exceptions
  - partie BD :
    - types : Oracle, dérivés
    - select into (affectation variable)
    - curseur
    - gestion errerus BD par exception
    - SQL dynamique
- procédures stockées
- triggers

# JDBC

- le mode programme
- JDBC : qu'est-ce que c'est ? avantages ?
- interface, portabilité, connexion
- devant les machines
- gestion erreurs BD
- transactions
- exécution d'ordres BD : curseurs, paramètres, SQL dynamique
- JDBC et l'architecture en couches

# le mode programme : cas général

- qu'est-ce que c'est ?
- pour faire quoi ? pour qui ?
- comment ?
- rappel : PL/SQL et les problèmes du mode programme
- Java et les problèmes du mode programme : JDBC

# qu'est-ce que c'est ?

- lors écriture application :
  - Mêmes ordres SQL qu'en interactif
  - \+ programme autour
  - comme PL/SQL

# pour faire quoi ? pour qui ?

- connu :
  - gérer problèmes BD : ok avec SQL
  - code :
    - dédié à la gestion BD
    - local à la base

  : ok avec PL/SQL
- but : accès BD par application généraliste
  - qui doit faire : calculs, conversationnel, interface, réseau, etc.
- pour : informaticiens (comme PL/SQL)

# comment ?

- programme normal (Java, PHP, C, Cobol, etc.)
- + connexion
  - rappel :
    - Interactif : commande Unix
    - PL/SQL : déjà connecté
  - Transformation du processus prog. en client SGBD
- + n'importe quels ordres SQL
  - avec gestion des problèmes du mode programme

# PL/SQL et les pbs MP (1/5)

- il faut lancer des ordres BD depuis un programme
- l'insertion d'ordres BD dans un programme pose les problèmes suivants

# PL/SQL et les pbs MP (2/5)

- interface : concrètement, comment placer « insert into t values (7, 'ok') » dans un programme ?
  - PL/SQL : ordre SQL est instruction
- portabilité d'un SGBD à l'autre : est-il possible de faire tourner le même programme sur des SGBD différents ? (Oracle, MySQL, etc.)
  - PL/SQL : non : une procédure stockée est une entité Oracle

# PL/SQL et les pbs MP (3/5)

- connexion : comment indiquer qu'on démarre une connexion BD (+ login, base, etc.) ?
  - PL/SQL : procédure stockée (ou trigger) appelés depuis session : déjà connecté
- curseurs : quand une interrogation renvoie plusieurs résultats (ex : tous étudiants ayant A en BD), comment y accéder dans le programme ?
  - PL/SQL : curseur

# PL/SQL et les pbs MP (4/5)

- paramètres : concrètement, comment indiquer dans un ordre BD que les valeurs ne seront connues qu'à l'exécution ; puis les lier alors ?
  - PL/SQL : variables dans ordres BD
- SQL dynamique : de même, comment faire si ce sont des parties entières de l'ordre BD qui sont inconnues ?
  - construction ordre par concaténation de chaînes (possible aussi ordre avec variables, select)
  - PL/SQL : ok

# PL/SQL et les pbs MP (5/5)

- erreurs : comment le programme est-il informé d'une erreur
  - côté serveur (ex : table n'existe pas)
  - côté programme lors échange de données avec un ordre BD (ex : chaîne de caractères tronquée)
  - Oracle lève exception, programme la voit passer et peut la gérer
  - PL/SQL : ok : gestion d'exception
- transactions : problèmes et gestion reprise sur panne et concurrence mêmes qu'en interactif ? oui

# Java et les pbs MP : JDBC

- un programme Java souhaitant accéder à BD doit gérer ces problèmes
- le package JDBC fournit les outils nécessaires pour gérer les problèmes du mode programme

# JDBC

- le mode programme
- JDBC : qu'est-ce que c'est ? avantages ?
- interface, portabilité, connexion
- devant les machines
- gestion erreurs BD
- transactions
- exécution d'ordres BD : curseurs, paramètres, SQL dynamique
- JDBC et l'architecture en couches

# JDBC : qu'est-ce que c'est ? avantages ?

- API = ensemble de classes et interfaces (package) Java
- Java DataBase Connectivity
- permet d'accéder à des SGBD par SQL
- application JDBC = programme Java utilisant JDBC
- indépendant
  - d'un SGBD particulier (grâce drivers)
  - d'une architecture matérielle (grâce Java)

# exemple

- situation :
  - c'est l'anniversaire de Jeanne
  - on veut exécuter :

    ```
    update personne
    set age = age + 1
    where nom = 'Jeanne'
    ```
- regardons intuitivement premier programme Java/JDBC complet qui tourne
- puis nous reviendrons systématiquement sur tous les points

```
import java.sql.*;
class Exemple {
    public static void main(String[] args)
        throws SQLException, ClassNotFoundException {
        Class.forName(«oracle.jdbc.driver.OracleDriver » );
        Connection conn = DriverManager.getConnection(
            « jdbc:oracle:thin:waller/waller@orasrv:1521:orcl»);
        Statement stmt = conn.createStatement();
        stmt.executeUpdate(
            «update personne set age = age + 1 where nom = 'Jeanne' »);
        stmt.close();
        conn.close();
    }
}
```

# pbs MP : interface

- comment exécuter un ordre BD depuis Java ?
- Java appelle des fonctions d'une bibliothèque (JDBC)
- ces fonctions encapsulent les accès au serveur BD
- les ordres BD sont :
  - certaines fonctions de la bibliothèque
  - chaînes de caractères paramètres de fonctions de la bibliothèque

# pbs MP : portabilité

- driver = module logiciel de communication entre un logiciel et
  - du matériel (ex : imprimante)
  - un autre logiciel (ex : entre Java et Oracle)
- = implantation des interfaces de l'API JDBC
- chaque constructeur de SGBD fournit un driver à Java
- conséquence : un programme Java/JDBC qui utilise SQL standard tourne au-dessus de n'importe quel SGBD

# chargement driver(s)

- Class.forName(« oracle.jdbc.driver.OracleDriver ») : recherche cette classe Java
- possible plusieurs dans même programme : interagir avec différents serveurs de différents constructeurs (Oracle, MySQL, Sybase, Informix, etc.)
- (variante : DriverManager.registerDriver(new Oracle.jdbc.driver.OracleDriver()))
- (si applet : « dnlddriver » au lieu « driver »)

```
import java.sql.*;
class Exemple {
    public static void main(String[] args)
        throws SQLException, ClassNotFoundException {
        Class.forName(«oracle.jdbc.driver.OracleDriver » );
        Connection conn = DriverManager.getConnection(
            « jdbc:oracle:thin:waller/waller@orasrv:1521:orcl»);
        Statement stmt = conn.createStatement();
        stmt.executeUpdate(
            «update personne set age = age + 1 where nom = 'Jeanne' »);
        stmt.close();
        conn.close();
    }
}
```

# devant les machines

- compilation
  - standard (cause import package)
  - + fournir les classes du driver choisi
- concrètement :
  - indiquer au compilateur Java où se trouve le driver utilisé : CLASSPATH Unix
  - javac
  - java

# JDBC

- le mode programme
- JDBC : qu'est-ce que c'est ? avantages ?
- interface, portabilité, connexion
- devant les machines
- gestion erreurs BD
- transactions
- exécution d'ordres BD : curseurs, paramètres, SQL dynamique
- JDBC et l'architecture en couches

# pbs MP : connexion

- Connection x = DriverManager.getConnection(url)
- fonction getConnection :
  - récupère driver indiqué par url (=> déjà chargé)
  - l'utilise pour établir connexion avec le serveur
  - renvoie objet correspondant (classe Connection) : nécessaire toutes interactions avec le serveur

- url : String
  - = jdbc:oracle:drivertype:user/password@database
  - drivertype : oci7, oci8, thin (applet : thin)
  - @database :
    - optionnel, sinon defaut
    - oci : ligne de tnsnames.ora, thin : host:port:sid, SQL*Net

# l'objet de la classe Connection

- « session » =
  1. connexion à un serveur donné
  2. séquence d'ordres SQL
  3. déconnexion
- possible : une application JDBC a plusieurs sessions avec un ou plusieurs serveurs
  - (chaque session est alors un client)

- un objet Connection
  - définit une session
  - utilisé via ses méthodes pour
    - envoi ordres BD au serveur
    - gestion transactions
  - fournit informations sur tables, procédures stockées, etc.
  - créable que par getConnection (Connection: interface)

# déconnexion

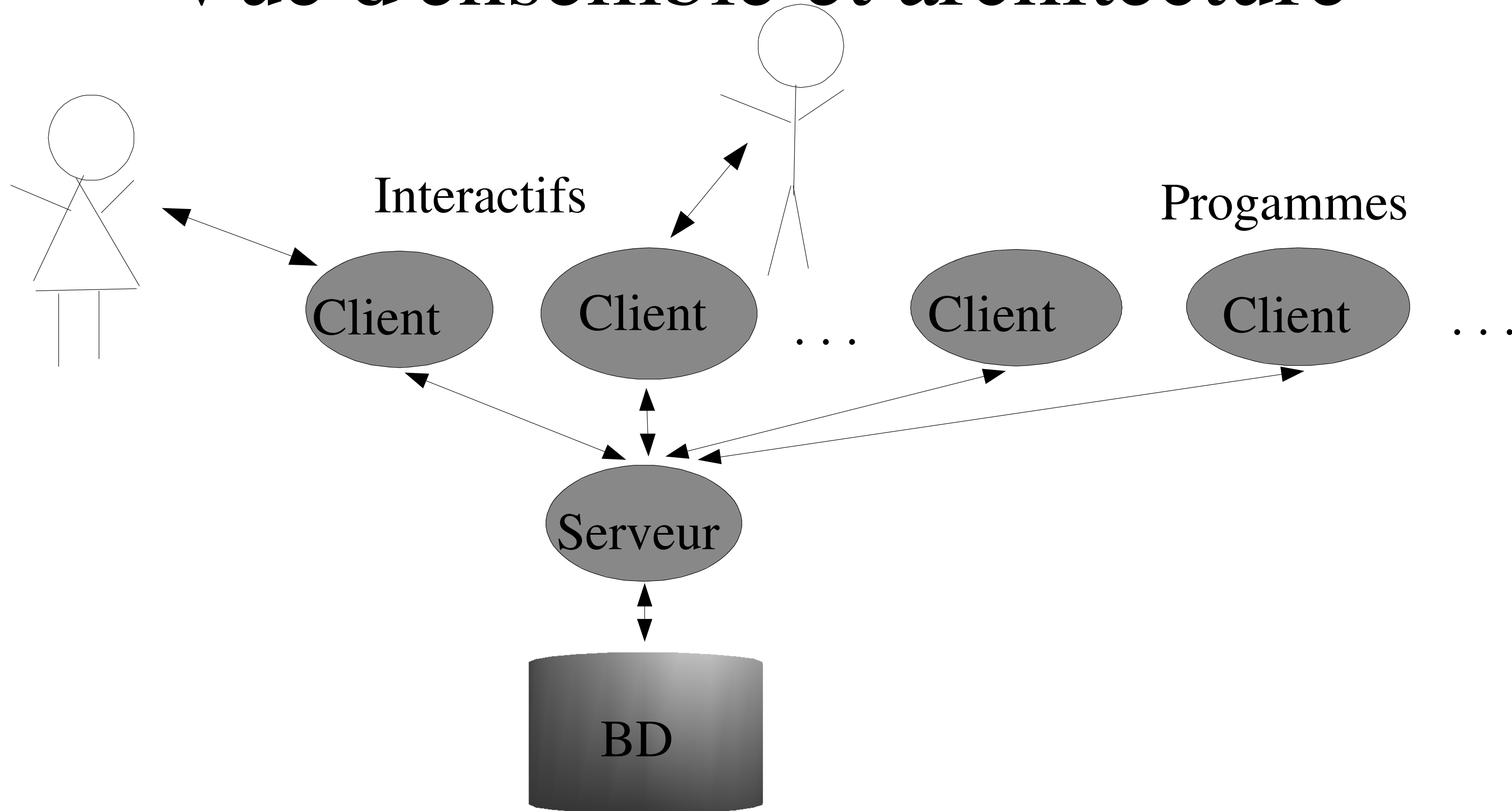
- méthode de Connection :
  - void close() throws SQLException
  - effectue déconnexion normale du serveur qui termine la session
  - utile si on veut déconnexion immédiate au lieu attendre que automatique
- déconnexion implicite automatique lorsque objet Connection libéré par garbage collector

```
import java.sql.*;
class Exemple {
    public static void main(String[] args)
        throws SQLException, ClassNotFoundException {
        Class.forName(«oracle.jdbc.driver.OracleDriver » );
        Connection conn = DriverManager.getConnection(
            « jdbc:oracle:thin:waller/waller@orasrv:1521:orcl»);
        Statement stmt = conn.createStatement();
        stmt.executeUpdate(
            «update personne set age = age + 1 where nom = 'Jeanne' »);
        stmt.close();
        conn.close();
    }
}
```

# déroulement d'un programme

- importer classes JDBC (statique)
- lancement et début de l'exécution d'un programme JDBC : inconnu du serveur
- charger driver(s) voulus(s) par le programmeur (=choix du SGBD)
- connexion (JDBC) : devient client
- envoi ordres SQL au serveur (JDBC)
- déconnexion (JDBC) : termine en tant que client
- inconnu du serveur : continue son exécution, puis termine comme programme (voir sur croquis)

# Vue d'ensemble et architecture



# JDBC

- le mode programme
- JDBC : qu'est-ce que c'est ? avantages ?
- interface, portabilité, connexion
- devant les machines
- gestion erreurs BD
- transactions
- exécution d'ordres BD : curseurs, paramètres, SQL dynamique
- JDBC et l'architecture en couches

# pbs MP : gestion des erreurs BD (intuition)

- rappel : un ordre BD envoyé au serveur peut générer une erreur
  - (ou une demande BD non SQL : mode confirmation automatique, méta-données, etc.)
- fonction JDBC ayant demandé cet ordre lève automatiquement une exception de la classe SQLException
  - => throws SQLException si pas gérée

# utilisation

```
try {
    . . . appel JDBC . . .
} catch (SQLException e) {
    . . . e . . .
}
```

- Sinon (Java) :
  - retour au premier catch (Exception)
  - Sinon : arrêt brutal programme
    - BD : Annulation transaction

# pour aller plus loin

- Exceptions JDBC :
  - SQLException
  - SQLWarning
  - DataTruncation
- Méthodes d'accès au compte-rendu

# pbs MP : transactions

- début : comme mode interactif (connexion, etc.)
- fin :
  - comme interactif (déconnexion, panne, etc.) avec en plus :
    - terminaison normale exécution du programme effectue confirmation
    - terminaison anormale (erreur non rattrapée) : est une annulation
- mode confirmation automatique activé par défaut

- méthodes de Connection :
  - void setAutoCommit(boolean) throws SQLException
    - true : mode confirmation automatique activé
    - false : désactivé
    - subtilités sur instant exact de la confirmation (hors module)
  - void commit() throws SQLException : confirmation
  - void rollback() throws SQLException

# JDBC

- le mode programme
- JDBC : qu'est-ce que c'est ? avantages ?
- interface, portabilité, connexion
- devant les machines
- gestion erreurs BD
- transactions
- exécution d'ordres BD : curseurs, paramètres, SQL dynamique
- JDBC et l'architecture en couches

# pbs MP : exécution d'ordres BD

- pbs MP :
  - curseurs
  - paramètres
  - SQL dynamique
- principe et déroulement
- ordres sans paramètre
- ordres avec paramètres
- appel procédure stockée
- SQL dynamique
- curseurs

- principe et déroulement :
  - création d'un « objet ordre » générique (ordre SQL non fixé)
  - envoi de l'ordre au serveur par méthode de cet objet
  - si select : récupération du résultat (curseur)
  - gestion erreurs
- 3 catégories :
  - ordres BD sans ou avec paramètres
    - paramètre : valeur d'une colonne
  - appel procédure stockée PL/SQL

```
import java.sql.*;
class Exemple {
    public static void main(String[] args)
        throws SQLException, ClassNotFoundException {
        Class.forName(«oracle.jdbc.driver.OracleDriver » );
        Connection conn = DriverManager.getConnection(
            « jdbc:oracle:thin:waller/waller@orasrv:1521:orcl»);
        Statement stmt = conn.createStatement();
        stmt.executeUpdate(
            «update personne set age = age + 1 where nom = 'Jeanne' »);
        stmt.close();
        conn.close();
    }
}
```

# ordres SQL sans paramètre

- = tout ordre SQL copié-collable sous SQL*Plus
- 2 catégories :
  - select
  - tous les autres
- objet classe Statement
- créé par méthode de Connection :

  Statement createStatement() throws SQLException
- ne pas utiliser si ordre exécuté plusieurs fois avec paramètres différents, car moins efficace

- un seul objet Statement en général (plusieurs possible : hors module)
- envoyer l'ordre au serveur par la méthode de Statement (si ordre non requête) :

  int executeUpdate(String sql) throws SQLException

  - renvoie
    - nombre de lignes traitées pour insert, update, delete
    - 0 sinon
  - String sql : sans le point-virgule de SQL*Plus

```
import java.sql.*;
class Exemple {
    public static void main(String[] args)
        throws SQLException, ClassNotFoundException {
        Class.forName(«oracle.jdbc.driver.OracleDriver » );
        Connection conn = DriverManager.getConnection(
            « jdbc:oracle:thin:waller/waller@orasrv:1521:orcl»);
        Statement stmt = conn.createStatement();
        stmt.executeUpdate(
            «update personne set age = age + 1 where nom = 'Jeanne' »);
        stmt.close();
        conn.close();
    }
}
```

# pbs MP : exécution d'ordres BD

- pbs MP :
  - curseurs
  - paramètres
  - SQL dynamique
- principe et déroulement
- ordres sans paramètre
- ordres avec paramètres
- appel procédure stockée
- SQL dynamique
- curseurs

# ordres SQL avec paramètres

- ordres SQL DML (insert, update, delete, select)
- remarque : même paramètres que dans PL/SQL (rappel : variables table ou colonne impossible)
- déroulement :
  - création « objet ordre » pour
    - un ordre SQL fixé
    - avec des paramètres formels

    + précompilation (« préparation »)
  - affectation des valeurs aux paramètres
  - envoi au serveur, etc.

# exemple

- situation : on veut enregistrer l'anniversaire d'une personne (= age++ dans BD)
- principe : where paramétré par le nom de la personne
- code :

```
PreparedStatement stmt = conn.prepareStatement(
    « update personne set age = age+1 where nom = ? »);
stmt.setString(1, « Jeanne »);
stmt.executeUpdate();
```

# PreparedStatement

- création par méthode de Connection :

  PreparedStatement prepareStatement(String sql)

    throws SQLException

- String sql :
  - contient un symbole « ? » pour chaque paramètre
  - repérés par leur position de gauche à droite (en commençant à 1)

- affectation par setXXX de PreparedStatement
  - où XXX est un type primitif de Java
  - une méthode pour chaque tel type
- ex :
  - void setString(int indiceParametre, String x)

    throws SQLException
  - x converti en varchar par le SGBD

- exécution par :
  - executeUpdate si sql non requête
  - executeQuery si sql requête
- utilisable même si aucun paramètre (mais inutilement coûteux)
- rem : nombre et type des paramètres doivent être connus

# avantage PreparedStatement sur Statement

- ex : même situation, mais plusieurs anniversaires
- possible avec Statement :

```
String[] p = {« Jeanne », « Jules »};
Statement stmt = conn.createStatement();
for (int i=0; i<p.length; i++)
    stmt.executeUpdate(
        « update personne set age = age+1 where nom = ' »
        + p[i] + «' » );
```

  - compilation et exécution à chaque fois
  - (rq : il faut les apostrophes si chaîne de car. Oracle)

- PreparedStatement :

```
String[] p = {« Jeanne », « Jules »};
PreparedStatement stmt = conn.prepareStatement(
    « update personne set age = age+1 where nom = ? »);
for (int i=0; i<p.length; i++) {
    stmt.setString(1, p[i]);
    stmt.executeUpdate(); }
```

  - précompilation une seule fois
  - exécution plusieurs fois avec nouvelles valeurs des paramètres
  - (rq : pas d'apostrophe)

# pbs MP : exécution d'ordres BD

- pbs MP :
  - curseurs
  - paramètres
  - SQL dynamique
- principe et déroulement
- ordres sans paramètre
- ordres avec paramètres
- appel procédure stockée
- SQL dynamique
- curseurs

# appel de procédure stockée

- exemple :
  - on a déjà une procédure stockée : anniversaire(n varchar)
  - incrémente de 1 l'âge de la ligne de nom n
  - on veut l'appeler depuis notre programme Java
- code :

```
String n = « Jeanne »;
CallableStatement stmt = conn.prepareCall(
    « {call anniversaire(?)} »);
stmt.setString(1, « Jeanne »); // comme PreparedStatement
stmt.executeUpdate(); // comme PreparedStatement
```

- CallableStatement hérite de PreparedStatement

- lecture de
  - paramètre OUT d'une procédure stockée
  - valeur renvoyée par une fonction stockée

  après exécution :
  - registerOutParameters : « déclarer » le type du paramètre (parmi java.sql.Types)
  - getXXX

- appel de la fonction âge :

```
CallableStatement stmt = conn.prepareCall(
    « {? = call age(?)} »);
stmt.setString(2, « Jeanne »);
stmt.registerOutParameter(1, Types.INTEGER);
stmt.execute();
System.out.println(stmt.getInt(1));
```

- il existe
  - variantes syntaxe
  - « bloc anonymes »

  mais spécifiques à Oracle
- ci-dessus : portable (Sybase, etc.)

# récapitulatif

- ordres SQL sans paramètres (schéma, instance, y compris select) : Statement
- ordres SQL avec paramètres (instance, y compris select) : PreparedStatement
- appel procédure stockée (avec ou sans paramètres) : CallableStatement

# pbs MP : exécution d'ordres BD

- pbs MP :
    - curseurs
    - paramètres
    - SQL dynamique
- principe et déroulement
- ordres sans paramètre
- ordres avec paramètres
- appel procédure stockée
- SQL dynamique
- curseurs

# pbs MP : SQL dynamique

- vu en PL/SQL (ex : table inconnue lors compilation)
- JDBC :
  - immédiat car ordre est une String : concaténation par + (subtilité si nombre/type attributs retour select inconnu)
  - Impossible faire autrement : JDBC ne fait que SQL dynamique
- contrepartie :
  - ordres non connus lors compilation
  - => vérification statique impossible (ex : table n'existe pas)
  - programmes moins robustes
- (vérification statique : SQLJ)

# pbs MP : curseurs

- rappel : curseur
- exemples
- détails

# exemple 1

- Train(client varchar2(10), dest varchar2(10), jour integer)
- afficher la première destination de Cassavetes par ordre alphabétique et le jour du voyage
- On suppose qu'il y en a au moins une
- commençons intuitivement sur un exemple

```
Statement s = c.createStatement();
ResultSet rset =
   s.executeQuery(
    "select dest, jour from train"
    + "where client = 'Cassavetes' order by dest");
rset.next();
System.out.println("Cassavetes va à " + rset.getString(1)
                              + " le jour " + rset.getInt("JOUR"));
r.close();
s.close();
```

# rappel : fonctionnement curseur

1. Déclaration du curseur
2. Remplissage en une seule fois par exécution de la requête
3. Récupération des lignes une par une

   (parcours séquentiel du curseur par un pointeur logique)
4. Libération de la zone : elle devient inaccessible

# rappel : parcours du curseur

- Analogue à parcours de fichier séquentiel
- Notion de pointeur logique
- Après le remplissage : pointeur logique positionné avant la 1ère ligne
- une étape :
  1. Avance le pointeur
  2. Lit la ligne pointée par le pointeur
  3. Si pas de ligne pointée (l'avant-dernière étape avait lu la dernière) : indicateur de ligne trouvée passe à faux

```
Statement s = c.createStatement();
ResultSet rset =
   s.executeQuery(
    "select dest, jour from train"
    + "where client = 'Cassavetes' order by dest");
rset.next();
System.out.println("Cassavetes va à " + rset.getString(1)
                          + " le jour " + rset.getInt("JOUR"));
r.close();
s.close();
```

# curseur JDBC

- sans paramètre ou avec
- déclaration et nommage :
  - pas vraiment : création « objet ordre »
  - Statement ou PreparedStatement selon si paramètres

- Remplissage : executeQuery
  - Renvoie objet de la classe ResultSet
  - = « la zone curseur », la table
- Avancement du pointeur et indicateur de fin de curseur
  - Méthode de la classe ResultSet
  - Boolean next() throws SQLException
  - True ssi nouvelle ligne trouvée

- Récupération de la ligne :
  - Colonne par colonne
  - ResultSet getXXX throws SQLException
    - Où XXX tout type primitif Java
  - Driver JDBC convertit donnée du curseur en type Java XXX
  - Colonne désignée au choix par :
    - Position :
      - Commence à 1
      - Plus efficace
    - Nom (majuscules)

- Exemple : String
  - String getString (int indiceColonne)
  - String getString (String nomColonne)
- Cf ci-dessus :
  - PreparedStatement.setXXX
  - CallableStatement.getXXX

- Libération :
  - void ResultSet close() throws SQLException
  - Sinon fait implicitement lors :
    - Fermeture (ou réexécution) de l'ordre qui l'a généré
    - GC

```
Statement s = c.createStatement();
ResultSet rset =
   s.executeQuery(
    "select dest, jour from train"
    + "where client = 'Cassavetes' order by dest");
rset.next();
System.out.println("Cassavetes va à " + rset.getString(1)
                        + " le jour " + rset.getInt("JOUR"));
r.close();
s.close();
```

# Exemple 2

- afficher la première destination p ar ordre alphabétique de Cassavetes et le jour du voyage, et la deuxième si elle existe
- On suppose au moins une

```
Statement s = c.createStatement();
ResultSet rset =
   s.executeQuery(
    "select dest, jour from train"
    + "where client = 'Cassavetes' order by dest");
rset.next();
System.out.println("Cassavetes va à " + rset.getString(1)
                            + " le jour " + rset.getInt("JOUR"));
if (rset.next())
    System.out.println("Cassavetes va à " + rset.getString(1)
                            + " le jour " + rset.getInt("JOUR"));
r.close();
s.close();
```

# Exemple 3

- Les afficher toutes
- On ne suppose rien
- s.executeQuery(. . .);

  while (rset.next())

  . . . getString(1) . . .
- Si aucun : on n'entre pas dans la boucle
- Si un ou plusieurs :
  - on se positionne sur la prochaine ligne, qu'on affiche
  - Après la dernière : on teste rset.next et on ne rentre pas dans la boucle

# pbs MP : exécution d'ordres BD

- pbs MP :
    - curseurs
    - paramètres
    - SQL dynamique
- principe et déroulement
- ordres sans paramètre
- ordres avec paramètres
- appel procédure stockée
- SQL dynamique
- curseurs

# cours 5 : JDBC

- le mode programme
- JDBC : qu'est-ce que c'est ? avantages ?
- interface, portabilité, connexion
- devant les machines
- gestion erreurs BD
- transactions
- exécution d'ordres BD : curseurs, paramètres, SQL dynamique
- JDBC et l'architecture en couches

# JDBC et l'architecture en couches

- même principe qu'en objet :
  - les données sont encapsulées
  - accès par procédures stockées (sauf rares cas)
- Java :
  - interagit peu avec la base :
    - appelle les procédures stockées
    - gère commit et rollback
    - si nécessaire : curseur pour affichage
  - est plutôt dédié à la partie non BD de l'application

# JDBC

- le mode programme
- JDBC : qu'est-ce que c'est ? avantages ?
- interface, portabilité, connexion
- devant les machines
- gestion erreurs BD
- transactions
- exécution d'ordres BD : curseurs, paramètres, SQL dynamique
- JDBC et l'architecture en couches

# Exercices

- SQL dynamique : fonction qui lit au clavier un nom de table et la supprime (drop)
- appel fonction stockée : fonction qui exécute une fonction stockée (ci-jointe)

# Délégués ?