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

JDBC

X2003 – Majeure 1 Informatique
2005/06
Emmanuel Waller, LRI

Résumé des épisodes précédents

- But du module : savoir utiliser un SGBD pour résoudre les 12 problèmes BD rencontrés par une application généraliste
- mode interactif
- mode programme
  - PL/SQL : code BD, triggers, procédures stockées

Les 12 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)

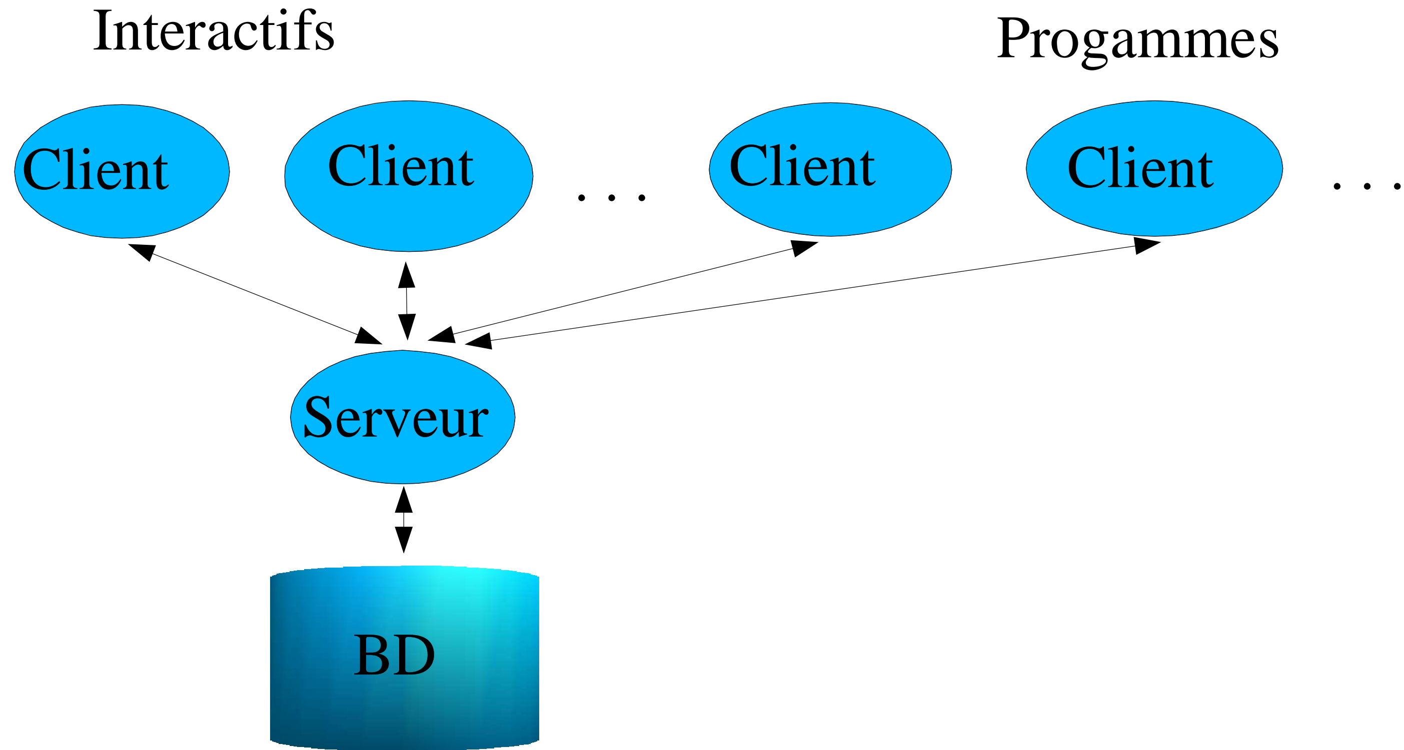
JDBC

- le mode programme
- JDBC : qu'est-ce que c'est ? avantages ?
- exemple
- comment ?
- concrètement
- gestion erreurs BD
- exécution d'ordres SQL
  - principe et déroulement
  - ordres sans paramètre
  - ordres avec paramètres
  - Select

le mode programme

- vue d'ensemble
- qu'est-ce que c'est ?
- pour faire quoi ? pour qui ?
- Comment ?
- PL/SQL et le mode programme

Vue d'ensemble



qu'est-ce que c'est ?

- lors écriture application :
  - Mêmes ordres SQL qu'en interactif
  - + programme autour

pour faire quoi ? pour qui ?

- connu :
  - gérer problèmes BD : ok avec SQL
  - ok avec PL/SQL :
    - factoriser dans fonction le code (séquence d'ordres) qui devra se répéter (dans l'application, dans le temps; motivation usuelle informatique)
    - autonomie (expert non présent pour penser et taper)
    - expressivité (requêtes, contraintes)
- but : accès BD par application généraliste
  - 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
  - rappel : PL/SQL : que transactions
- + gestion de la communication
  - erreurs lors ordres SQL (ex : table n'existe pas)
  - transmission valeurs entre prog. et ordres SQL (ex : null, chaîne tronquée)

PL/SQL et le mode programme

- rappel : PL/SQL :
  - factorise séquences d'ordres + contrôle
  - doit gérer erreurs SQL
  - même si pas de problème (différent JDBC, etc.) pour :
    - connexion, déconnexion
    - nature des données échangées entre ordres SQL et procédure stockée (grâce types Oracle + dérivés)
    - ordre SQL est une instruction PL/SQL
    - compilation en connaissant la base
- PL/SQL :
  - est un langage de mode programme
  - à forte intégration avec le SGBD

JDBC

- le mode programme
- JDBC : qu'est-ce que c'est ? avantages ?
- exemple
- comment ?
- concrètement
- gestion erreurs BD
- exécution d'ordres SQL
  - principe et déroulement
  - ordres sans paramètre
  - ordres avec paramètres
  - Select

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)

comment ? principe

- étapes de l'exécution d'un application JDBC (dans cet ordre chronologique)
- importer classes JDBC
- charger driver(s) voulus(s) par le programmeur
- se connecter au serveur BD
- interagir avec serveur : ordres SQL grâce JDBC
- se déconnecter du serveur

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 {
        Class.forName(«oracle.jdbc.driver.OracleDriver » );
        Connection conn = DriverManager.getConnection(
            « jdbc:oracle:thin:waller/waller@miage:1521:dbmiage »);
        Statement stmt = conn.createStatement();
        stmt.executeUpdate(
            «update personne set age = age + 1 where nom = 'Jeanne' »);
        stmt.close();
        conn.close();
    }
}
```

concrètement

- 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 ?
- exemple
- comment ?
- concrètement
- gestion erreurs BD
- exécution d'ordres SQL
  - principe et déroulement
  - ordres sans paramètre
  - ordres avec paramètres
  - Select

gestion des erreurs BD

- un ordre SQL envoyé au serveur peut générer une erreur (ou une demande non SQL : mode confirmation automatique, méta-données, etc.)
- ex : update... : table or view does not exist
- fonction JDBC ayant demandé cet ordre lève automatiquement une exception de la classe SQLException
  - => throws SQLException si pas gérée
- gestion : par rattrapage (catch) de l'exception

La classe SQLException

- Dérive de java.lang.Exception
- Méthodes :
  - String getMessage() : message d'erreur spécifique du SGBD (celui du mode interactif)
  - String getSQLState() : code d'erreur normalisé (spécification X/Open SQL)
  - Int getErrorCode() : code d'erreur spécifique du SGBD (celui du mode interactif)
  - SQLException getNextException() : lien vers exception suivante s'il y en a plusieurs

JDBC

- le mode programme
- JDBC : qu'est-ce que c'est ? avantages ?
- exemple
- comment ?
- concrètement
- gestion erreurs BD
- exécution d'ordres SQL
  - principe et déroulement
  - ordres sans paramètre
  - ordres avec paramètres
  - Select

JDBC

- le mode programme
- JDBC : qu'est-ce que c'est ? avantages ?
- exemple
- comment ?
- concrètement
- gestion erreurs BD
- exécution d'ordres SQL
  - principe et déroulement
  - ordres sans paramètre
  - ordres avec paramètres
  - Select

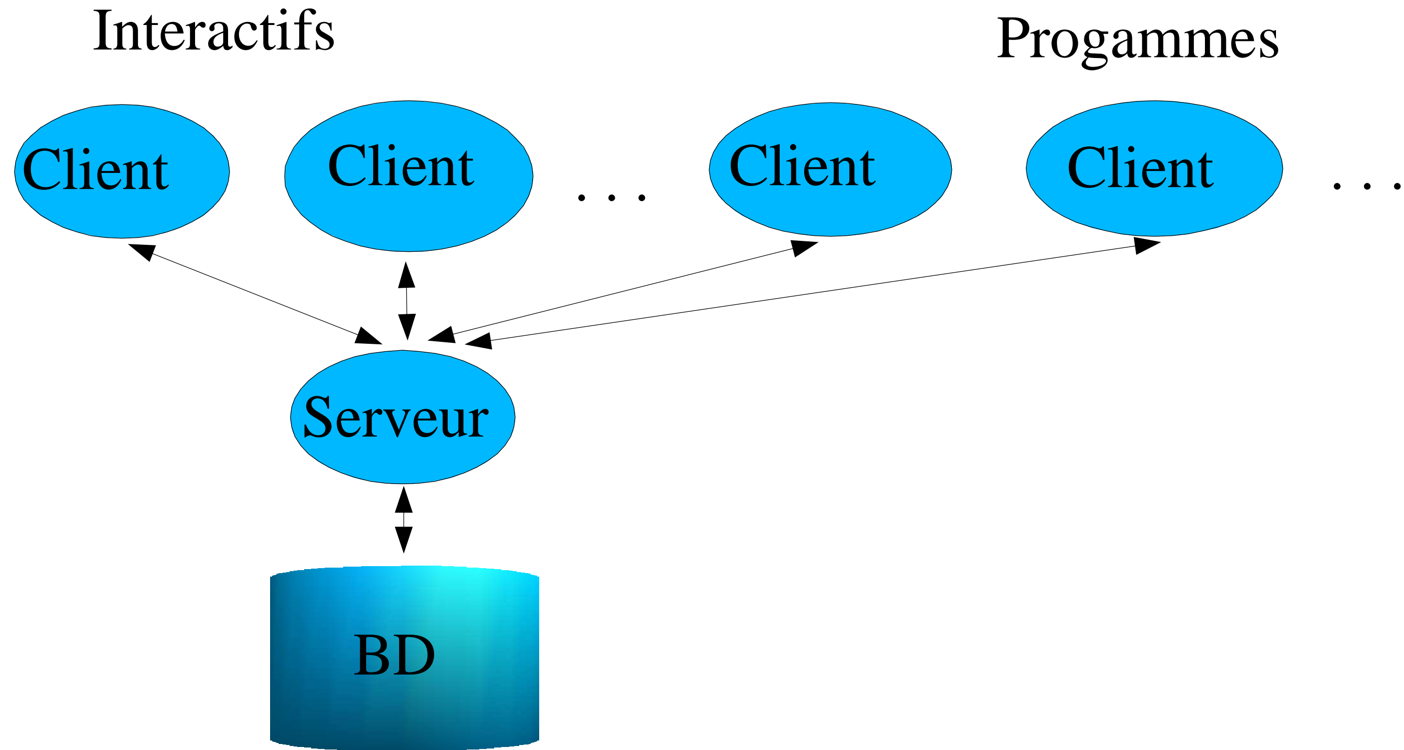
exécution d'ordres SQL

- principe et déroulement
- ordres sans paramètre
- ordres avec paramètres
- appel procédure et fonction stockée

déroulement

- lancement et début de l'exécution d'un programme JDBC : inconnu du serveur
- connexion : devient client
- envoie ordres SQL au serveur
- déconnexion : termine en tant que client
- inconnu du serveur : continue son exécution, puis termine comme programme
- voir sur croquis suivant

Vue d'ensemble



exécution d'ordres SQL

- 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 SQL sans ou avec paramètres
    - paramètre : valeur d'une colonne
  - appel procédure (ou fonction) stockée (PL/SQL)

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 {
        Class.forName(«oracle.jdbc.driver.OracleDriver » );
        Connection conn = DriverManager.getConnection(
            « jdbc:oracle:thin:waller/waller@miage:1521:dbmiage »);
        Statement stmt = conn.createStatement();
        stmt.executeUpdate(
            «update personne set age = age + 1 where nom = 'Jeanne' »);
        stmt.close();
        conn.close();
    }
}
```

ordres SQL avec paramètres

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

    \+ précompilation
  - affectation des valeurs aux paramètres
  - envoi au serveur, etc.

exemple

- situation : on veut enregistrer l'anniversaire d'une personne
- 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 chacun
- ex :
  - void setString(int indiceParametre, String x)
    throws SQLException
  - x converti en varchar par Oracle

- exécution par executeUpdate si sql non requête
- utilisable même si aucun paramètre

avantage Prepared Statement 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)

JDBC

- le mode programme
- JDBC : qu'est-ce que c'est ? avantages ?
- exemple
- comment ?
- concrètement
- gestion erreurs BD
- exécution d'ordres SQL
  - principe et déroulement
  - ordres sans paramètre
  - ordres avec paramètres
  - Select

Exécution d'ordres SQL : select

- Exemple 1
- Rappel :
  - exécution d'ordres SQL : principe
  - Curseur : principe
- Select : Fonctionnement
- Exemples
- Le type Oracle ref cursor

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

Rappel : exécution d'ordres SQL : principe

- principe et déroulement :
  - création d'un « objet ordre »
  - envoi de l'ordre au serveur
  - si select : récupération du résultat (curseur)
  - gestion erreurs
- 3 catégories :
  - ordres SQL sans paramètres : Statement
  - ordres SQL avec paramètres : PreparedStatement
  - appel procédure stockée : CallableStatement

```
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 : Curseur PL/SQL

- But : récupérer un résultat de requête faisant plusieurs lignes
- = zone mémoire
- Nommée
- À laquelle est associée une requête
- Peut contenir 0, 1 ou plusieurs lignes
- Taille réglée à l'exécution
- Sert à contenir l'ensemble des lignes résultat de cette requête (table résultat de cette requête)

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 open :
  - pointeur logique positionné avant la 1ère ligne
  - c%found est à vrai
- Fetch c into a :
  1. Avance le pointeur
  2. Lit la ligne pointée par le pointeur
  3. Si pas de ligne pointée (l'avant-dernier fetch avait lu la dernière) : c%found devient 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();
```

Select : fonctionnement (sans paramètre ou avec)

- Exécution d'ordre SQL JDBC + curseur PL/SQL
- Déclaration et nommage :
  - pas vraiment : création « objet ordre »
  - Statement ou PreparedStatement selon si paramètres
  - Vu
- Si paramètres : setXXX (vu)

- 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 déjà vu :
  - 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

Exemple 2

- afficher la première destination par 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

JDBC

- le mode programme
- JDBC : qu'est-ce que c'est ? avantages ?
- exemple
- comment ?
- concrètement
- gestion erreurs BD
- exécution d'ordres SQL
  - principe et déroulement
  - ordres sans paramètre
  - ordres avec paramètres
  - Select

JDBC : aller plus loin

- drivers
- connexion, déconnexion, vue d'ensemble
- SQL dynamique
- transactions
- exécution d'ordres SQL : appels PL/SQL
- Communication programme-serveur : situations de transmission de données

driver

- 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

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, Sybase, Informaix, etc.)
- (variante : DriverManager.registerDriver(new Oracle.jdbc.driver.OracleDriver()))
- (si applet : « dnlddriver » au lieu « driver »)

connexion au serveur

- 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 SQL 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

Rappel : 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 sattique impossible (ex : table n'existe pas)
  - Programmes moins robustes
- SQLJ : permet vérification statique

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

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.registerOutParameters(1, Types.INTEGER);
stmt.execute(); // et non executeUpdate()
int a;
if (stmt.wasNull()) a = -1;
else a = stmt.getInteger(1);
```

- stmt.setNull(3, Types.String)
- il existe
  - variantes syntaxe
  - « bloc anonymes »

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

remarque : transmission de données entre programme et ordres SQL

- serveur vers programme : résultat select
- programme vers serveur : paramètres
  - valeurs pour la base : insert, update set
  - critères de sélection : where (select, update, delete)
- valeurs apparaissant dans ordres :
  - des des constantes
  - dans variables
- ex : PL/SQL
- autres ordres SQL : pas de valeurs

Transmission de données entre « environnement » et ordres SQL

- Interactif : pas de variables
  - Résultats à l'écran (select)
  - Paramètres : dans l'ordre « en dur » (insert, update, where)
  - (en fait variables possibles dans SQL*Plus)
- PL/SQL :
  - Résultats select : into + variable
    - *quid* si variable trop longue ?
  - Paramètres insert, update set et where : variable en pseudo-colonne
- JDBC ?