BASE DE DONNÉES AVANCÉES

Adeel Ahmad

DÉFINITION D 'UNE BASE DE DONNÉES

Une Base de Données est une organisation ***cohérente*** *de données* ***permanentes*** *et accessibles par des utilisateurs* ***concurrents.***

3 CARACTÉRISTIQUES PRINCIPALES :

- Cohérence par rapport à des contraintes d 'intégrité et recouvrement de la cohérence même après des pannes
- permanence ou persistance
- concurrence : permettre à plusieurs utilisateurs d'accéder aux données simultanément

BASE DE DONNÉES

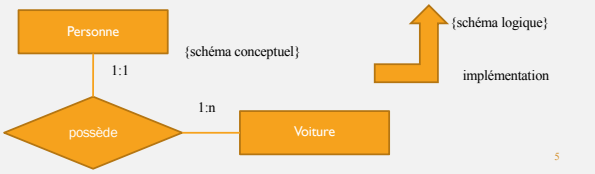
- Une base de données représente
  - un **ensemble de données**
  - mémorisé par un ordinateur
  - accessible à de nombreuses personnes
  - organisé selon un modèle de données

BASE DE DONNÉES RELATIONNEL

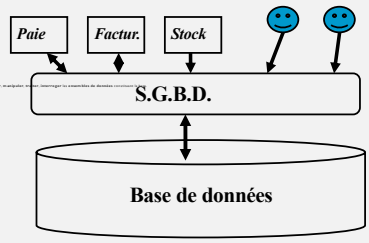

{monde réel}

Personne	Voiture
Nom Prénom	...
DUPONT Charle SUNNA Valérie	...

modélisation conceptuelle



L'IDÉE DES BASES DE DONNÉES



DÉFINITION D 'UN SGBD

- Un SGBD est le système logiciel qui permet l'interaction avec une base de données
- Pour assurer la cohérence des données il faut d'abord les modéliser
  - Un SGBD offre un modèle de données indépendant de la structure physique des données
  - Le modèle intègre aussi l'expression de contraintes d'intégrité

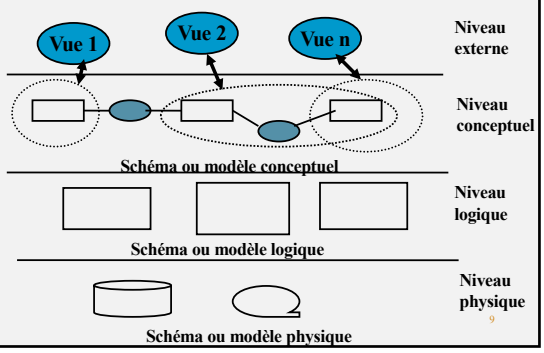
7

DÉFINITION D 'UN SGBD

- Le SGBD offre des langages dédiés pour l 'interrogation de la base de données et la manipulation de ses instances
- Pour assurer l'indépendance des données et des applications les SGBD sont structurés en couches ou niveaux de représentation des données

8

NIVEAUX DE REPRÉSENTATION DES DONNÉES



9

PLAN DU COURS

- Le standard JDBC
- les pilotes JDBC d'Oracle / MySQL, etc.
- SQLJ

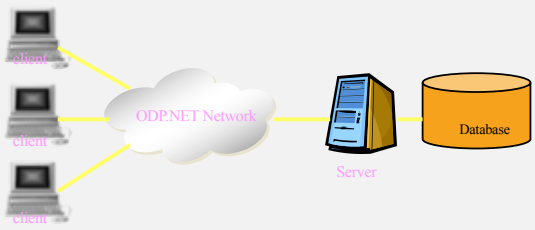
10

MIDDLEWARE ORIENTÉE DES BASES DE DONNÉES

- Middleware orientée de base de données est un middleware qui facilite la communication avec une base de données
- Ce middleware permet aux programmes d'utiliser des requêtes SQL qui auront accès la base de données sans avoir à connaître l'interface de la base de données
- ODBC et JDBC (Open / Java Database Connectivity) sont les deux interfaces standard / libre de programmation (API) pour accéder à une base de données.
- Oracle dispose d'une interface qui est basé sur ODP.NET (Oracle Data Provider/network) protocole
- MySQL dispose d'une interface connector/J

11

BASIC CLIENT – SERVER CONNECTION:



12

PRINCIPES DE JDBC/SQLJ

- Les standards JDBC/SQLJ sont définies pour permettre à une application Java d 'accéder à une base de données :
  - Définir des données
  - Rechercher des données
  - Manipuler des données
- JDBC/SQLJ sont des middlewares destinées aux applications et applets Java

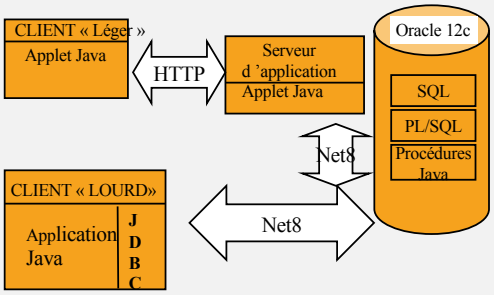
13

PRINCIPE DES PILOTES JDBC

- Il existe 2 types de pilotes JDBC pour Oracle :
  - les pilotes JDBC utilisés par les machines virtuelles Java standard
  - un pilote JDBC spécifique pour les méthodes Java exécutées par Oracle Jserver

14

ARCHITECTURE D'UNE APPLICATION MULTI-TIERS



15

ARCHITECTURE MULTI-TIERS AVEC JDBC

- Déploiement client lourd :
  - L 'application accède à la BD via JDBC par le protocole Net8
  - L 'application nécessite un poste de travail dit client lourd : elle réalise à la fois des opérations sur les données et l 'affichage de ces données.

16

- Déploiement client léger :
  - Le client est un navigateur web qui communique avec un serveur d'applications par le protocole HTTP
  - Le client ne réalise aucun traitement sur les données
  - Le serveur d 'applications recueille les données par une connexion JDBC, puis les traite et transmet le résultat au client léger.

17

JDBC

- JDBC : Java Data Base Connectivity Framework permettant l'accès aux bases de données relationnelles dans un programme Java
- Indépendament du type de la base utilisée (mySQL, Oracle, Postgres ...)
- Seule la phase de connexion au SGBDR change
- Permet de faire tout type de requêtes
- Sélection de données dans des tables
- Création de tables et insertion d'éléments dans les tables
- Gestion des transactions
- Packages : java.sql et javax.sql

18

LES PILOTES JDBC ET LEURS

- JDBC est une bibliothèque de classes Java destinées à faciliter l 'accès aux données contenues dans une BDR.
- JDBC permet :
  - une connexion simultanée à plusieurs BD
  - la gestion des transactions
  - l 'interrogation
  - l 'appel des procédures stockées

19

PRINCIPES GÉNÉRAUX D'ACCÈS À UNE BDD

- Première étape
  - Préciser le type de driver que l'on veut utiliser
  - Driver permet de gérer l'accès à un type particulier de SGBD
- Deuxième étape
  - Récupérer un objet « Connection » en s'identifiant auprès du SGBD et en précisant la base utilisée
- Etapes suivantes
  - A partir de la connexion, créer un « statement » (état) correspondant à une requête particulière
  - Exécuter ce statement au niveau du SGBD
  - Fermer le statement
- Dernière étape
  - Se déconnecter de la base en fermant la connexion

20

CONNEXION JDBC

- Pour l 'ouverture d 'une connexion JDBC les méthodes Java doivent spécifier :
  - le nom du pilote JDBC
  - le nom de l utilisateur et son mot de passe
  - les paramètres pour localiser la base de données.

21

CONNEXION AU SGBD

- Classe java.sql.DriverManager
  - Gestion du contrôle et de la connexion au SGBD
- Méthodes principales

  static void registerDriver(Driver driver)
  - Enregistre le driver (objet driver) pour un type de SGBD particulier
  - Le driver est dépendant du SGBD utilisé
- static Connection getConnection(

  String url, String user, String password)
- Crée une connexion permettant d'utiliser une base
  - url : identification de la base considérée sur le SGBD
  - Format de l'URL est dépendant du SGGB utilisé
  - user : nom de l'utilisateur qui se connecte à la base
  - password : mot de passe de l'utilisateur

22

GESTION DES CONNEXIONS

- Interface java.sql.Connection
- Préparation de l'exécution d'instructions sur la base, 2 types
- Instruction simple : classe Statement
- On exécute directement et une fois l'action sur la base
- Instruction paramétrée : classe PreparedStatement
- L'instruction est générique, des champs sont non remplis
- Permet une pré-compilation de l'instruction optimisant les performances
- Pour chaque exécution, on précise les champs manquants
- Pour ces 2 instructions, 2 types d'ordres possibles
  - Update : mise à jour du contenu de la base
  - Query : consulation (avec un select) des données de la base

23

GESTION DES CONNEXIONS

- Méthodes principales de Connection
  - Statement createStatement()
    - Retourne un état permettant de réaliser une instruction simple
  - PreparedStatement prepareStatement(String ordre)
    - Retourne un état permettant de réaliser une instruction paramétrée et pré-compilée pour un ordre ordre
    - Dans l'ordre, les champs libres (au nombre quelconque) sont précisés par des « ? »
      - Ex : "select nom from clients where ville=?"
      - Lors de l'exécution de l'ordre, on précisera la valeur du champ
  - void close()
    - Ferme la connexion avec le SGBD

24

INSTRUCTION SIMPLE

- Classe Statement
  - ResultSet executeQuery(String ordre)
    - Exécute un ordre de type SELECT sur la base
    - Retourne un objet de type ResultSet contenant tous les résultats de la requête
  - int executeUpdate(String ordre)
    - Exécute un ordre de type INSERT, UPDATE, ou DELETE
  - void close()
    - Ferme l'état

25

INSTRUCTION PARAMÉTRÉE

- Classe PreparedStatement
  - Avant d'exécuter l'ordre, on remplit les champs avec
    - void set[Type](int index, [Type] val)
      - Remplit le champ en ième position définie par index avec la valeur val de type [Type]
      - [Type] peut être : String, int, float, long ...
      - Ex : void setString(int index, String val)
  - ResultSet executeQuery()
    - Exécute un ordre de type SELECT sur la base
    - Retourne un objet de type ResultSet contenant tous les résultats de la requête
- int executeUpdate()
  - Exécute un ordre de type INSERT, UPDATE, ou DELETE

26

LECTURE DES RÉSULTATS

- Classe ResultSet
  - Accès aux colonnes/données dans une ligne
  - [type] get[Type](int col)
    - Retourne le contenu de la colonne col dont l'élément est de type [type] avec [type] pouvant être String, int, float, boolean ...
    - Ex : String getString(int col)
- Fermeture du ResultSet
  - void close()

27

EXEMPLE

```
// charger d 'abord les pilotes jdbc
Class.forname (« oracle.jdbc.driver.OracleDriver)
// on peut aussi utiliser
DriverManager.registerDriver(new
   oracle.jdbc.driver.OracleDriver());
// établir la connexion
connexion con
   =DriverManager.getConnection(jdbc:oracle:thin:scott
   /tiger@banane:1521:dptinfo);
```

28

EXEMPLE D'UTILISATION DE JDBC

```
// Exemple de programme JAVA qui utilise le pilote JDBC
// thin d'Oracle pour effectuer un SELECT et itérer sur
// les lignes du résultat
// Il faut importer le paquetage java.sql
// pour utiliser JDBCpackage ExemplesJDBC
import java.sql.*;

class JDBCThinGetDeptScott
{  public static void main (String args [])
   throws SQLException, ClassNotFoundException, java.io.IOException
{
// Charger le pilote JDBC d'Oracle
   Class.forName ("oracle.jdbc.driver.OracleDriver");
 // Connection à une BD à distanceavec un pilote thin
Connection uneConnection   =
    DriverManager.getConnection
("jdbc:oracle:thin:@banane:1521:dptinfo", "scott", "tiger");
```

29

```
// Création d'un énoncé associé à la Connexion
 Statement unEnoncéSQL = uneConnection.createStatement ();
 // Exécution d'un SELECT
  ResultSet résultatSelect = unEnoncéSQL.executeQuery
          ("SELECT deptno, dname "+
                  "FROM DEPT " +
                 "WHERE deptno > 10");
 // Itérer sur les lignes du résultat du SELECT et extraire les valeurs
 // des colonnes dans des variables JAVA
 while (résultatSelect.next ())
{     int deptno = résultatSelect.getInt ("deptno");
      String dname = résultatSelect.getString ("dname");
System.out.println ("Numéro du departement:" + deptno);
System.out.println ("Nom du departement:" + dname);    }
 // Fermeture de l'énoncé et de la connexion
unEnoncéSQL.close();
uneConnection.close();  }}
```

30

EXEMPLE D 'UTILISATION DES OPÉRATION DU LMD

```
import java.sql.*;

class ClientInsertJDBC
{
  public static void main (String args [])
        throws SQLException, ClassNotFoundException, java.io.IOException
  {
    // Charger le pilote JDBC d'Oracle
    Class.forName ("oracle.jdbc.driver.OracleDriver");

    // Connexion à une BD
    Connection uneConnection =
      DriverManager.getConnection ("jdbc:oracle:oci8:@", "godin", "oracle");

    // Création d'un énoncé associé à la Connection
    Statement unEnoncéSQL = uneConnection.createStatement ();

    // Insertion d'une ligne dans la table Client
    int n = unEnoncéSQL.executeUpdate
      ("INSERT INTO CLIENT " +
       "VALUES (100, 'G. Lemoyne-Allaire', '911');
    System.out.println ("Nombre de lignes insérées:" + n);

    // Fermeture de l'énoncé et de la connexion
    unEnoncéSQL.close();
    uneConnection.close();
  }
}
```


GESTION DES TRANSACTIONS

- Par défaut : *auto-commit*
- Pour modifier

```
uneConnection.setAutoCommit(false);
```

- Pour un commit explicite :

```
uneConnection.commit();
```

- Pour un rollback :

```
uneConnection.rollback();
```


LES INTERFACES JDBC

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

MISE EN ŒUVRE D 'UNE APPLICATION JAVA-JDBC

- Importer le package **`java.sql`**
- 1. Enregistrer le driver JDBC
- 2. Etablir la connexion à la base de données
- 3. Créer une zone de description de requête
- 4. Exécuter la requête
- 5. Traiter les données retournées
- 6. Fermer les différents espaces de travail

ENREGISTRER LE DRIVER JDBC

- Méthode **forName()** de la classe **Class** :

Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Class.forName("oracle.jdbc.driver.OracleDriver");

- quand une classe **Driver** est chargée, elle doit créer une instance d 'elle même et s 'enregistrer auprès du **DriverManager**
- certains compilateurs refusent cette notation et demande plutôt :

Class.forName("driver_name").newInstance();

URL DE CONNEXION

- **L 'URL est de la forme :**

jdbc:<sousprotocole>:<nomBD>;param=valeur, ...

- l 'utilisation de JDBC
- le driver ou le type de SGBDR
- l 'identification de la base locale ou distante
  - avec des paramètres de configuration éventuels
    - nom utilisateur, mot de passe, ...
- Exemples :

String url = "jdbc:odbc:maBase" ;

String url = "jdbc:oracle: thin:@banane:1521:dptinfo", "scott", "tiger";

CONNEXION À LA BASE

- Méthode **`getConnexion()`** de **`DriverManager`**
  - 3 arguments :
    - l 'URL de la base de données
    - le nom de l 'utilisateur de la base
    - son mot de passe

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

  - le **`DriverManager`** essaye tous les drivers qui se sont enregistrés (chargement en mémoire avec **`Class.forName`**()) jusqu 'à ce qu'il trouve un *driver* qui peut se connecter à la base

37

CRÉATION D 'UN STATEMENT

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

38

CRÉATION D 'UN STATEMENT

- A partir de l 'objet **Connexion**, on récupère le **Statement** associé :

```
Statement req1 = connexion.createStatement();

PreparedStatement req2 =
  connexion.prepareStatement(str);

CallableStatement req3 = connexion.prepareCall(str);
```

39

EXÉCUTION D'UNE REQUÊTE

- 3 types d 'exécution :
  - **executeQuery()** : pour les requêtes (SELECT) qui retournent un **ResultSet** (tuples résultants)
  - **executeUpdate()** : pour les requêtes (INSERT, UPDATE, DELETE, CREATE TABLE, DROP TABLE) qui retournent un entier (nombre de tuples traités)
  - **execute()** : procédures stockées

40

EXÉCUTION D'UNE REQUÊTE

- **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 ename, job FROM emp " +
  "WHERE empno=7188 ORDER BY ename");
int nb = st.executeUpdate("INSERT INTO dept(DEPT) " +
                          "VALUES(06)");
```

41

EXÉCUTION D'UNE REQUÊTE

- 2 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..

42

TRAITEMENT DES DONNÉES DE RETOUR

- L 'objet **ResulSet** (retourné par l 'exécution de **executeQuery()**) permet d 'accéder aux champs des tuples 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

43

RESULTSET

- Il se parcourt itérativement ligne par ligne
  - par la méthode **`next()`**
    - retourne **`false`** si dernier tuple lu, **`true`** sinon
    - chaque appel fait avancer le curseur sur le tuple suivant
    - initialement, le curseur est positionné avant le premier tuple
      - exécuter next() au moins une fois pour avoir le premier

```
while(rs.next()) {// Traitement de chaque
  tuple}
```

44

RESULTSET

- impossible de revenir au tuple précédent ou de parcourir l 'ensemble dans un ordre aléatoire
- Les colonnes sont référencées par leur numéro ou par leur nom
- L 'accès aux valeurs des colonnes se fait par les méthodes de la forme **`getXXX()`**
  - lecture du type de données XXX dans chaque colonne du tuple courant

45

```
int val = rs.getInt(3) ; // accès à la 3e colonne
String prod = rs.getString("PRODUIT") ;

Statement st = connection.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");
}
```

46

TYPES DE DONNÉES JDBC

- 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

47

CORRESPONDANCES DES TYPES

Type JDBC	Type Java
CHAR, VARCHAR , LONGVARCHAR	String
NUMERIC, DECIMAL	java.math.BigDecimal
BINARY, VARBINARY, LONGVARBINARY	byte[]
BIT	boolean
INTEGER	int
BIGINT	long
REAL	float
DOUBLE, FLOAT	double
DATE	java.sql.Date
TIME	java.sql.Time
….	…..

48

CAS DES VALEURS NULLES

- 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

49

CAS DES VALEURS NULLES

- 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()`** , **`getObject()`** et **`getDate()`**) retournent un "**`null`** " Java
  - les méthodes numériques (**`getByte()`** , **`getInt()`** , etc) retournent "0"
  - **`getBoolean()`** retourne " **`false`** "

50

FERMETURE DES ESPACES

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

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

51

ACCÈS AUX MÉTA-DONNÉES

- La méthode **`getMetaData`**() permet d 'obtenir des informations sur les types de données du **`ResultSet`**
  - elle renvoie des **`ResultSetMetaData`**
  - on peut connaître entre autres :
    - le nombre de colonne : **`getColumnCount()`**
    - le nom d 'une colonne : **`getColumnName(int col)`**
    - le type d 'une colonne : **`getColumnType(int col)`**
    - le nom de la table : **`getTableName(int col)`**
    - si un NULL SQL peut être stocké dans une colonne : **`isNullable()`**

52

RESULTSETMETADATA

```
Resultset rs = stmt.executeQuery("SELECT * FROM
  emp");
ResultSetMetaData rsmd = rs.getMetatData();

int nbColonnes = rsmd.getColumnCount();
for(int i = 1; i <= nbColonnes; i++) {
  // colonnes numerotees a partir de 1 (et non 0)
  String typeCol = rsmd.getColumnTypeName(i);
  String nomCol = rsmd.getColumnName(i);
}
```

- Exercice : Retrouver le schéma d 'une table dont le nom est lu au clavier

53

DATABASEMETADATA

- Pour récupérer des informations sur la base de données elle-même, utiliser la méthode **`getMetaData()`** de l 'objet **`Connection`**
  - dépend du SGBD avec lequel on travaille
  - elle renvoie des **`DataBaseMetaData`**
  - on peut connaître entre autres :
    - les tables de la base : **`getTables()`**
    - le nom de l 'utilisateur : **`getUserName()`**
    - ...

54

REQUÊTES PRÉCOMPILÉES

- L 'objet **PreparedStatement** envoie une requête sans paramètres à la base de données pour pré-compilation et spécifiera le moment voulu la valeur des paramètres
  - plus rapide qu 'un **Statement** classique
    - le SGBD 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 pré-compilées

55

REQUÊTES PRÉCOMPILÉES

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

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

- les arguments dynamiques sont spécifiés par un "?"
- ils sont ensuite positionnés par les méthodes **setInt()** , **setString()** , **setDate()** , … de **PreparedStatement**
- **setNull()** positionne le paramètre à NULL (SQL)
- ces méthodes nécessitent 2 arguments :
  - le premier (int) indique le numéro relatif de l 'argument dans la requête
  - le second indique la valeur à positionner

56

EXÉCUTION D 'UNE REQUÊTE PRÉCOMPILÉE

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

57

EXCEPTIONS

- **SQLException** est levée dès qu 'une connexion ou un ordre SQL ne se passe pas correctement
  - la méthode **getMessage()** donne le message en clair de l 'erreur
  - renvoie aussi des informations spécifiques au gestionnaire de la base comme :
    - **SQLState**
    - code d 'erreur fabricant
- **SQLWarning** : avertissements SQL

58

JDBC ET ORACLE 12C

```
ORACLE_HOME = …
CLASSPATH=$CLASSPATH:$ORACLE_HOME/jdbc/lib/classes111.
  zip

import java.sql.*;

Class.forName("oracle.jdbc.driver.OracleDriver");
static final url =
  "jdbc:oracle:thin:@banane:1521:dptinfo";
conn =
  DiverManager.getConnection(url,   "scott",  "tiger"
  );
```

59

PERFORMANCES DE JDBC

- Quelques limitations :
  - **ResultSet.next()** fait un accès à la base pour chaque ligne retournée
    - impossible de ne faire qu'un accès à la base pour obtenir l'ensemble des lignes résultats
  - impossible de revenir en arrière dans le **ResultSet**
    - pénalisant si l'utilisateur veut naviguer dans les lignes
  - JDBC effectue 2 accès à la base par défaut :
    - pour déterminer le type des valeurs de retour
    - puis pour récupérer les valeurs

60