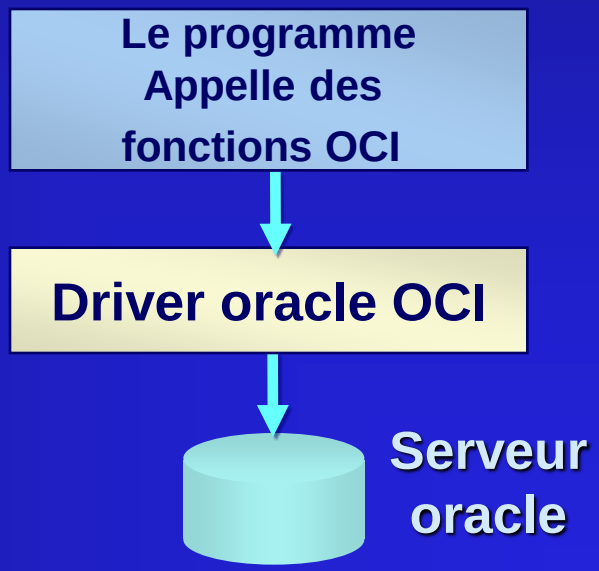
Présentation de l'API JDBC

Pierre Lefebvre

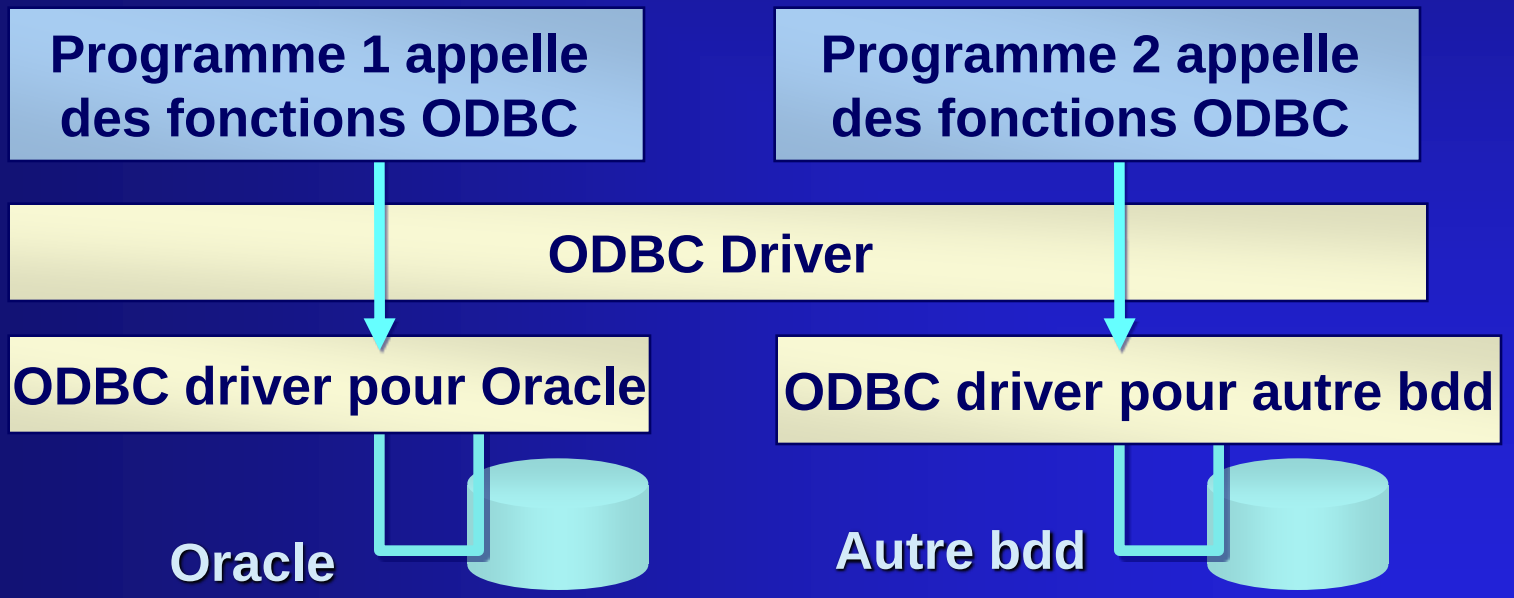
Accéder aux bases de données en dehors de java

- **Les vendeurs de bases de données fournissent une API que les programmeurs peuvent appeler pour accéder à une base de donnée :**
  - Call Level Interface (CLI)
  - Par exemple pour oracle, Call Interface (OCI)



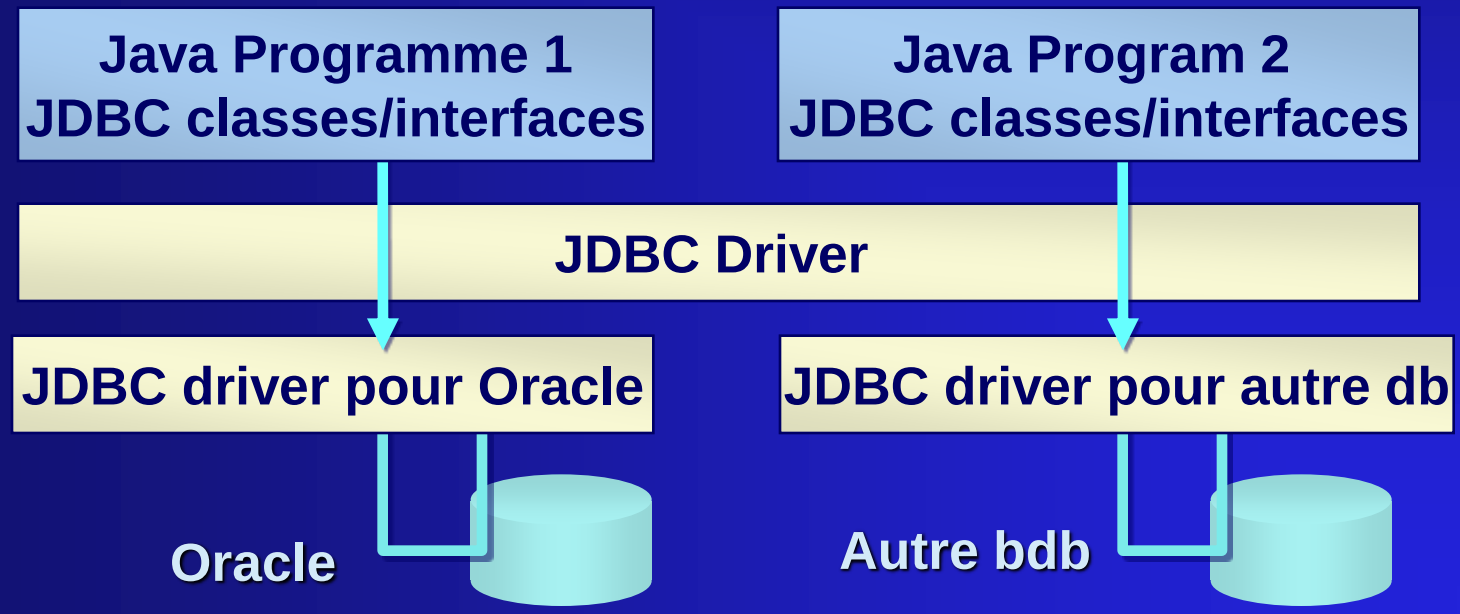
ODBC: une CLI standard

- ODBC fournit une interface standard aux bases de données



D'ODBC à JDBC

- JDBC joue une rôle similaire pour java
- JDBC définit des interfaces standards et des classes que l'on peut utiliser à l'aide de java



JDBC ?

- JDBC définit des interfaces standards
  - Importer le package `java.sql` dans une application java
  - Interfaces implementées par les drivers JDBC

Exemple d'interfaces JDBC

```
interface Driver{…}
```

```
interface Connection{…}
```

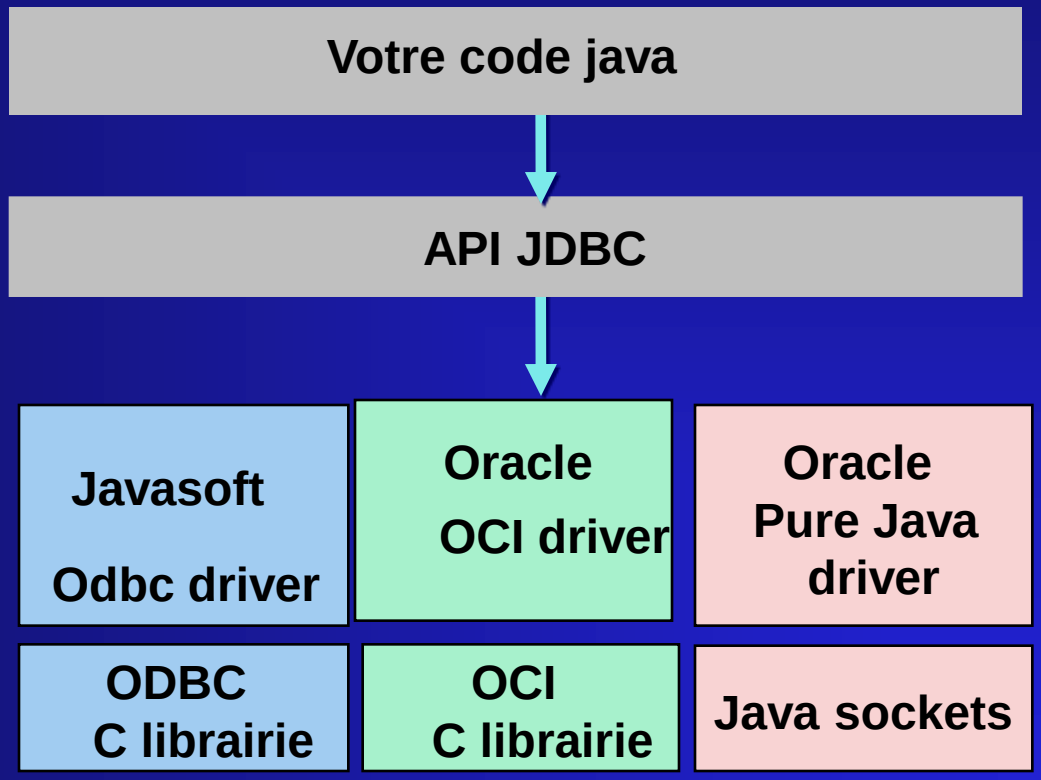
```
interface Statement{…}
```

```
interface ResultSet{…}
```

Driver JDBC, comme Oracle

```
class AAA
 implements Driver{…}

class BBB
 implements Connection{…}

etc…
```

Le modèle en couche de JDBC



Drivers JDBC

- **4 types de drivers (taxonomie de JavaSoft) :**
  - Type 1 : JDBC-ODBC *bridge driver*
  - Type 2 : *Native-API, partly-Java driver*
  - Type 3 : *Net-protocol, all-Java driver*
  - Type 4 : *Native-protocol, all-Java driver*

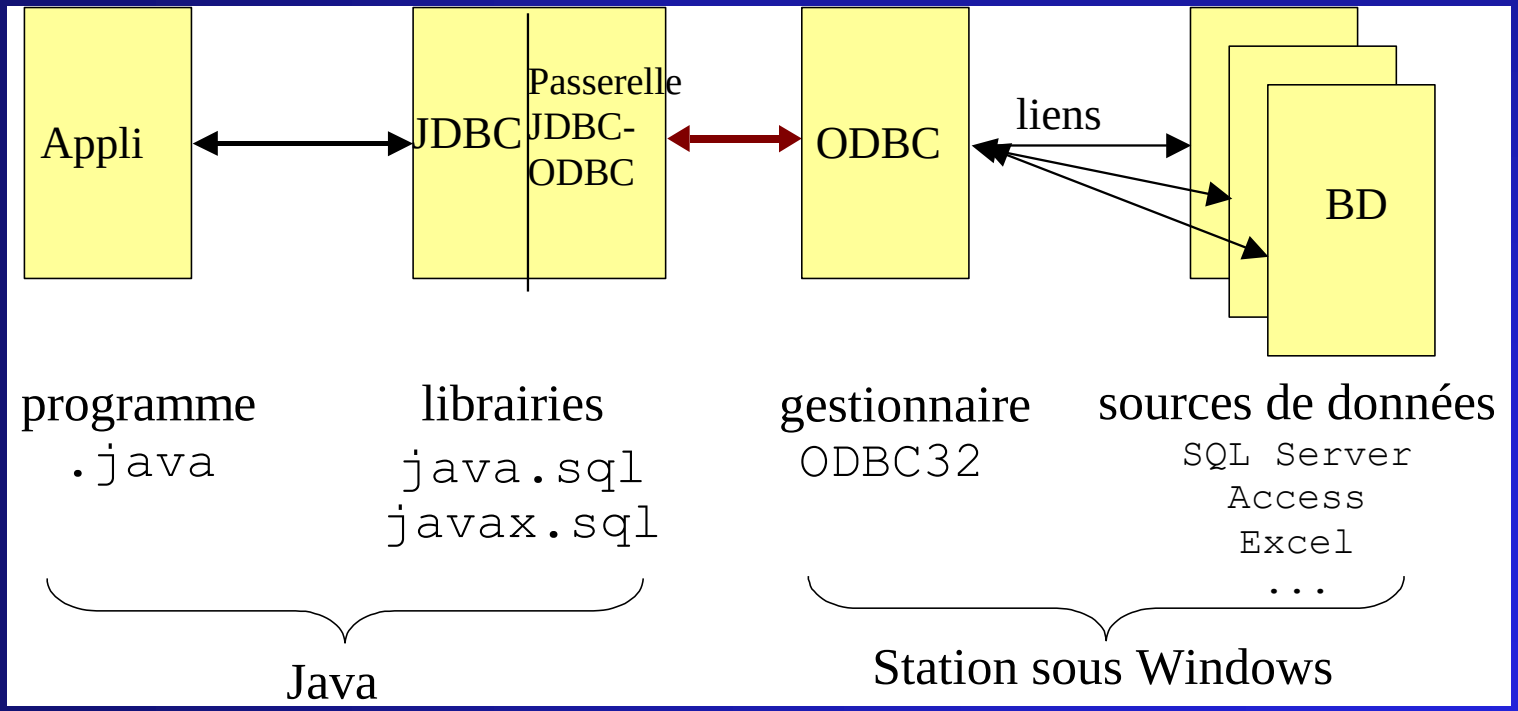
- **Tous les drivers :**
  - http://java.sun.com/products/jdbc/jdbc.drivers.html

Driver de type 1

- **Le driver accède à un SGBDR en passant par les drivers ODBC (standard Microsoft) via un pont JDBC-ODBC :**
  - **les appels JDBC sont traduits en appels ODBC**
    - presque tous les SGBDR sont accessibles (monde Windows)
  - **nécessite l ’emploi d ’une librairie native (code C)**
    - ne peut être utilisé par des *applets* (sécurité)
  - **est fourni par SUN avec le JDK 1.1**
    - `sun.jdbc.odbc.JdbcOdbcDriver`

Accès à une base MS ACCESS

- **Passerelle Jdbc:odbc**
- **gestionnaire de données ODBC32**



Driver de type 2

- **Driver d ’API natif :**
  - **fait appel à des fonctions natives (non Java) de l ’API du SGBDR**
    - gère des appels C/C++ directement avec la base
  - **fourni par les éditeurs de SGBD et généralement payant**
  - **ne convient pas aux *applets* (sécurité)**
    - interdiction de charger du code natif dans la mémoire vive de la plate-forme d ’exécution

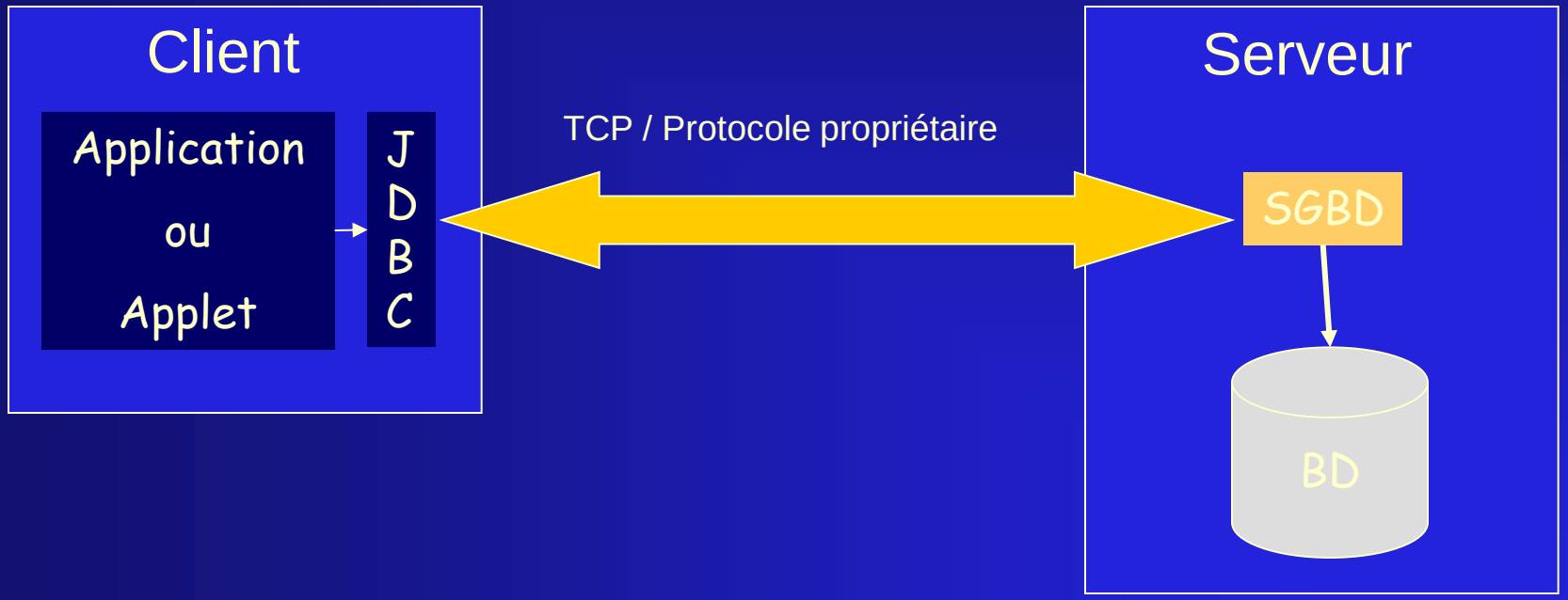
Driver de type 3

- **Pilote « tout Java » ou « 100% Java »**
  - interagit avec une API réseau générique et communique avec une application intermédiaire (*middleware*) sur le serveur
  - le *middleware* accède par un moyen quelconque aux différents SGBDR
  - portable car entièrement écrit en Java
    - pour *applets* et applications

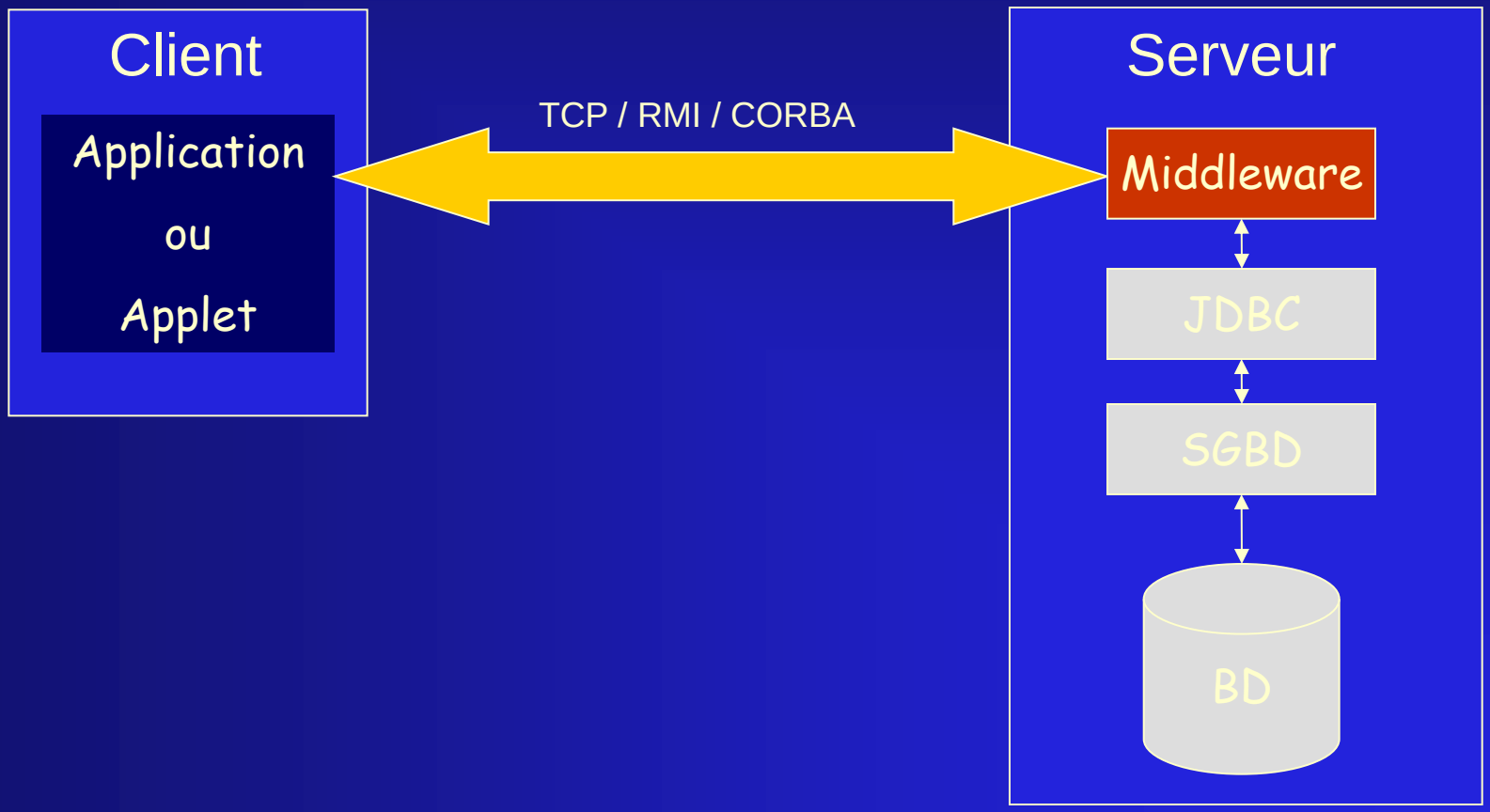
Driver de type 4

- **Driver « 100% Java » mais utilisant le protocole réseau du SGBDR**
  - interagit avec la base de données via des *sockets*
  - généralement fourni par l'éditeur
  - aucun problème d 'exécution pour une *applet* si le SGBDR est installé au même endroit que le serveur Web
    - sécurité pour l 'utilisation des *sockets* : une *applet* ne peut ouvrir une connexion que sur la machine ou elle est hébergée

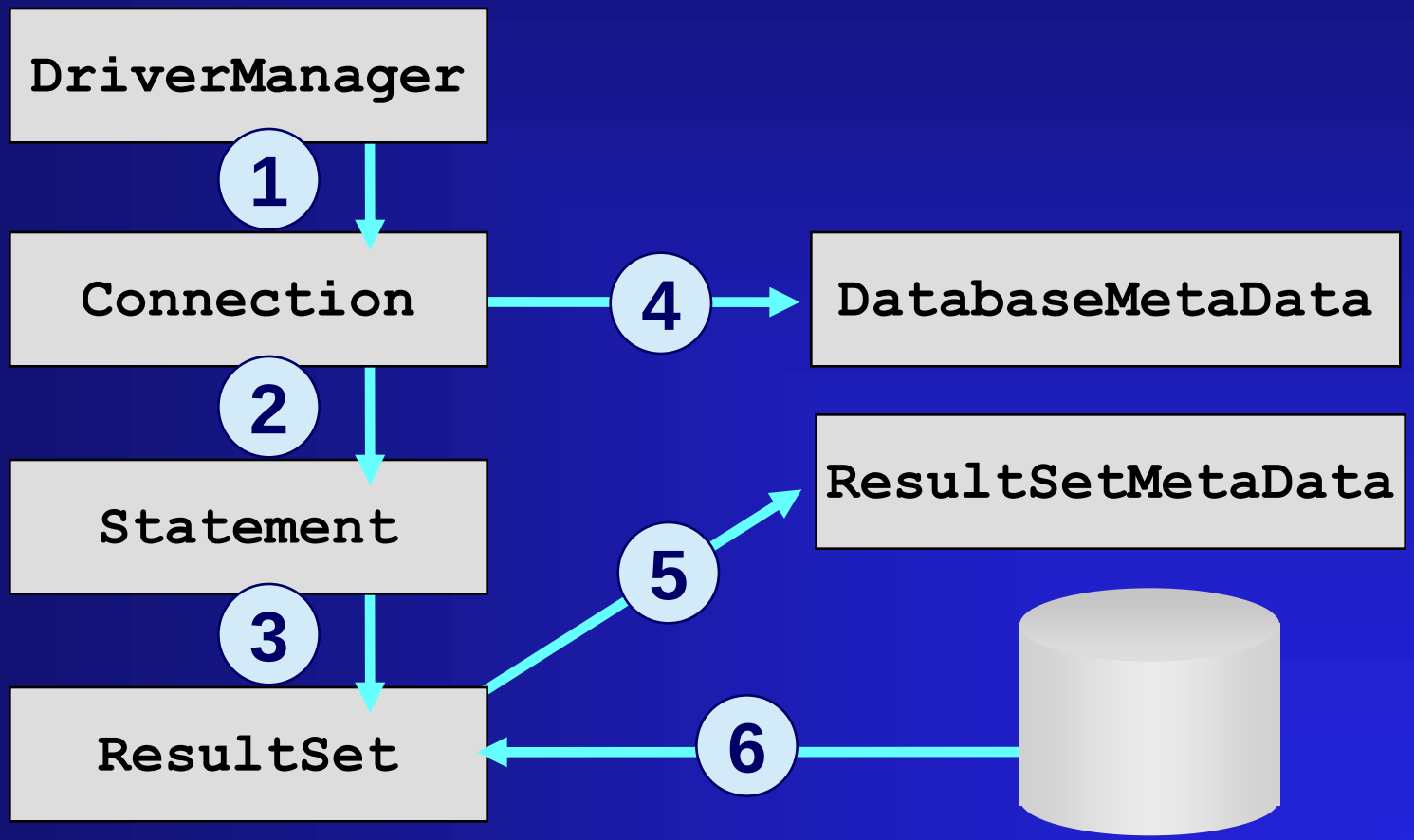
Architecture 2-tiers



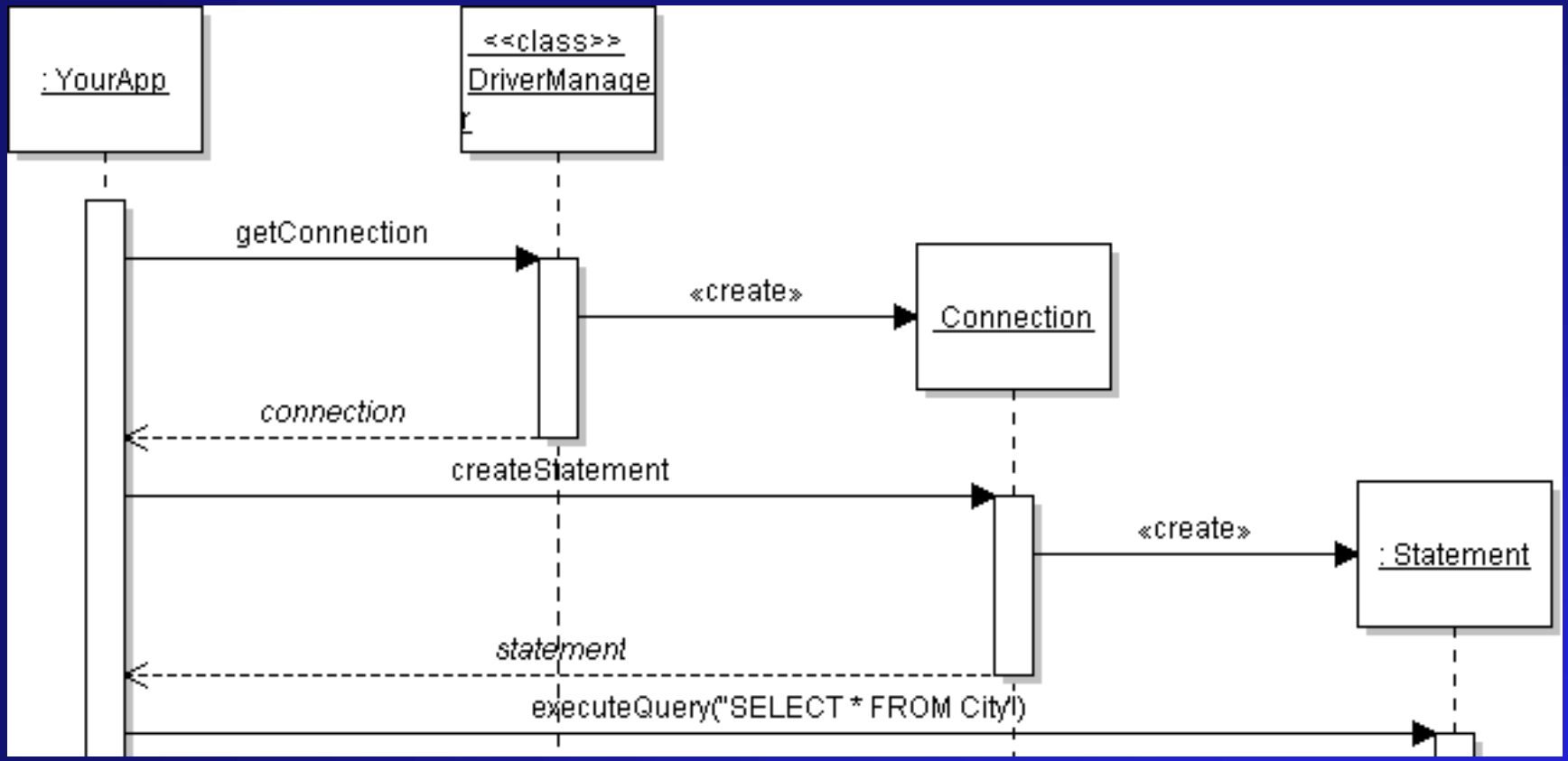
Architecture 3-tiers



Relation entre classes JDBC



Mise en place de JDBC



L'API JDBC 3.0

Classe/interface	Description
`java.sql.Driver` `java.sql.Connection`	Pilotes JDBC pour les connexions aux sources de données SQL.

`java.sql.Statement`
`java.sql.PreparedStatement`
`java.sql.CallableStatement`
	Construction d'ordres SQL.

`java.sql.ResultSet`	Gestion des résultats des requêtes SQL.
`java.sql.DriverManager`	Gestion des pilotes de connexion.
`java.sql.SQLException`	Gestion des erreurs SQL.
`java.sql.DatabaseMetaData` `java.sql.ResultSetetaData`	Gestion des méta-informations (description de la base de données, des tables…).
`java.sql.SavePoint`	Gestion des transactions et des sous-transactions.

Mise en œuvre de JDBC

- Charger un pilote de base de données
- Ouvrir une connexion de base de données
- Envoyer des instructions SQL à une base de données pour exécution
- Extraire les résultats renvoyés suite à une requête de base de données
- Fermer les objets de connexion et de requête
- Gérer les exceptions et les avertissements

Pilote JDBC

- **Chaque base de données nécessite son driver**
- **Récupérer le pilote de la base de données utilisée**
  - MySQL driver `mysql-connector-java-5.1.7-bin.jar`
  - Derby driver: `derby.jar` ou `derbyclient.jar`
  - HSQLDB driver: `hsqldb.jar`
- Démarche
  - **Ajouter le fichier jar dans votre projet de votre IDE**
  - **Ajouter votre fichier JAR à votre CLASSPATH**
    ```
    CLASSPATH=/my/path/mysql-connector.jar;.
    ```
  - **Ajouter le fichier JAR en utilisant la ligne de commandes:**
    ```
    java -cp /my/path/mysql-connector.jar ...
    ```
  - **Ajouter le fichier JAR dans le répertoire :**
    ```
    C:/java/jre1.6.0/lib/ext/mysql-connector.jar
    ```

Enregistrer un Driver JDBC

- **Quand une classe `Driver` est chargée, elle doit créer une instance d ’elle même et s ’enregistrer auprès du `DriverManager`**

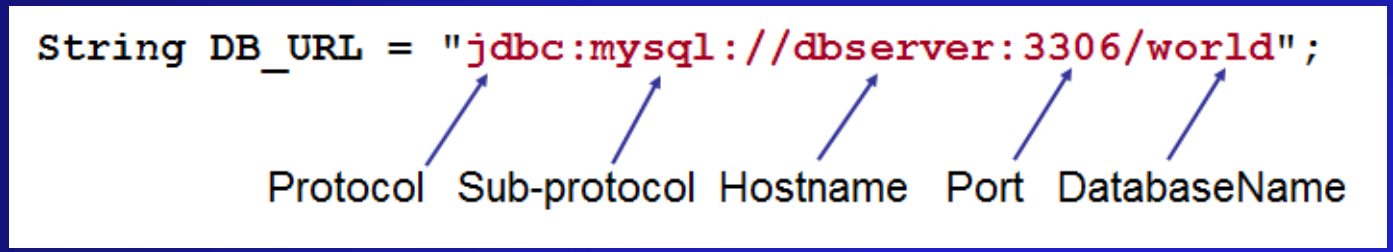
Class.forName(<database-driver>)

```
try {
  Class c = Class.forName(
                "oracle.jdbc.driver.OracleDriver");
}
catch (ClassNotFoundException e) {
  e.printStackTrace();
}
```


);

La connexion à une base de données

- `DriverManager` est utilisé pour ouvrir une connexion à une base de données
- La base est spécifiée en utilisant une URL, qui identifie le driver JDBC

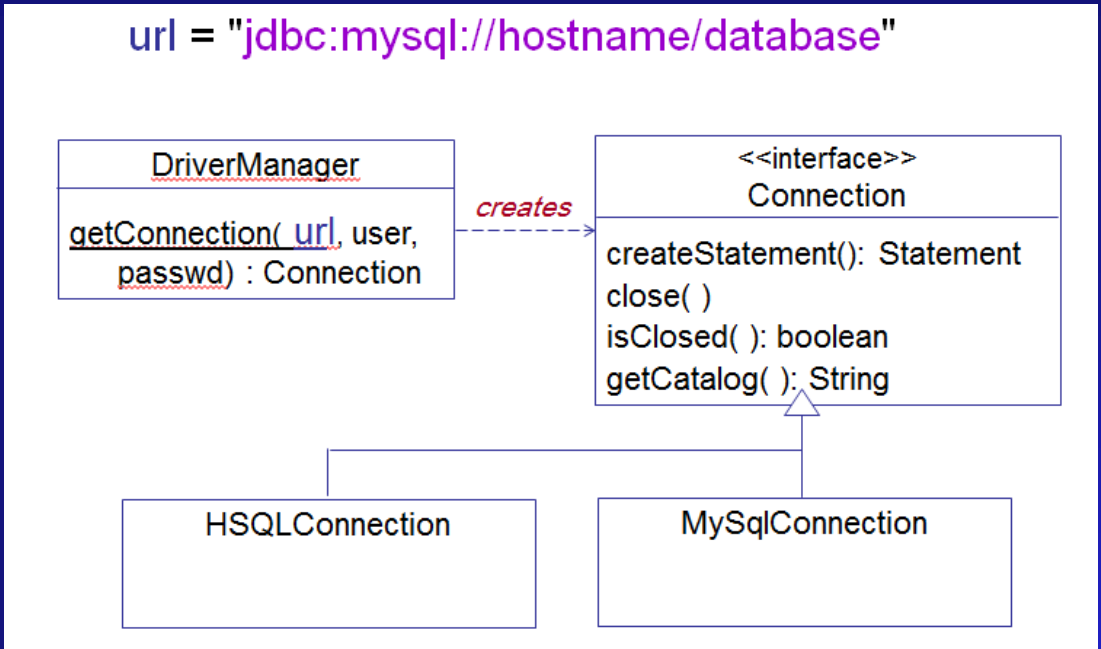


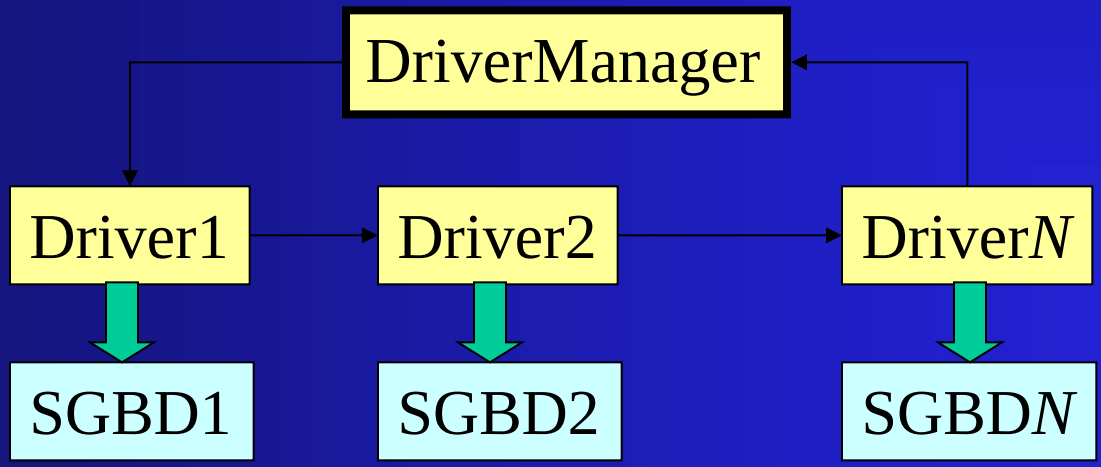
- **Exemples**
  - **Oracle thin driver**
    - oracle:jdbc:oracle:thin:@machinename:1521:dbname
  - **Derby**
    - jdbc:derby ://localhost:1527/sample

Exemple avec Mysql

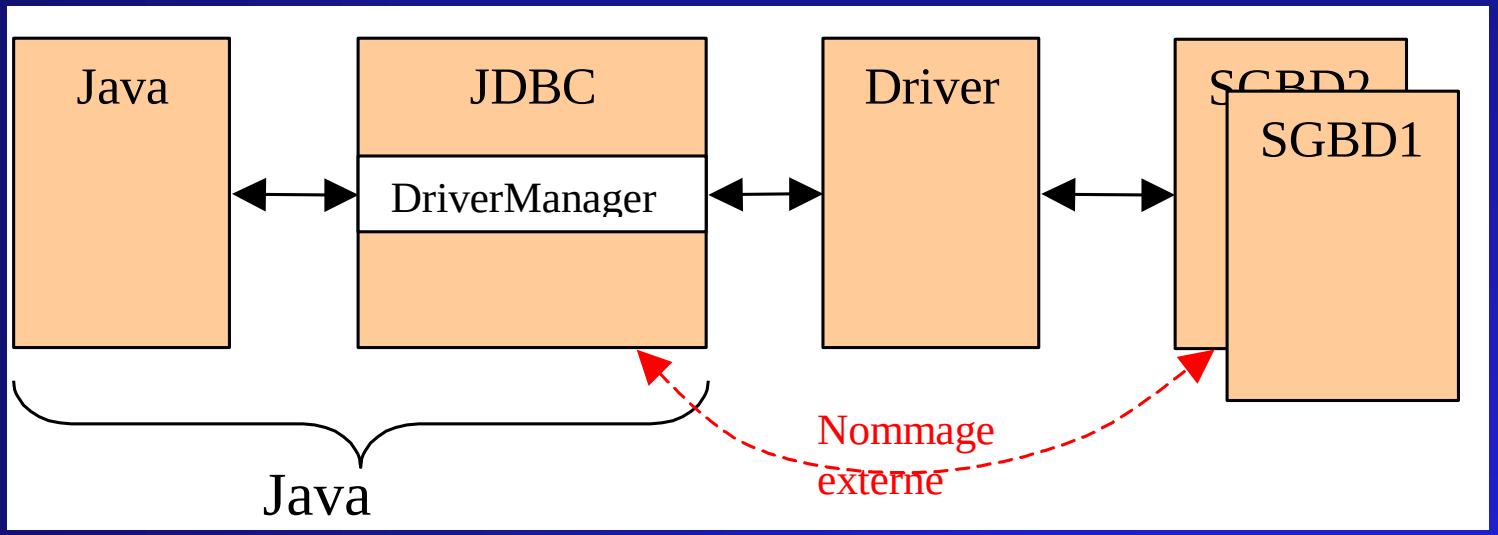
- Le hostname et le port sont optionnels.
- Pour le driver MySQL : les valeurs par défaut sont localhost et le port 3306
- Il y a 4 URL se référant à la même base de données
  - `"jdbc:mysql://localhost:3306/world"`
  - `"jdbc:mysql://localhost/world"`
  - `"jdbc:mysql:///world"`
  - `"jdbc:mysql:/world"`

DriverManager renvoie une connexion





Un même driver peut piloter plusieurs bases



Les principaux drivers

RDBMS	JDBC Driver Name
MySQL	Driver Name com.mysql.jdbc.Driver Database URL format: jdbc:mysql//hostname/databaseName
Oracle	Driver Name: oracle.jdbc.driver.OracleDriver Database URL format: jdbc:oracle:thin@hostname:portnumber:databaseName
DB2	Driver Name: COM.ibm.db2.jdbc.net.DB2Driver Database URL format: jdbc:db2:hostname:portnumber/databaseName
Access	Driver Name: com.jdbc.odbc.JdbcOdbcDriver Database URL format: jdbc:odbc:databaseName

Classe `java.sql.Connection`

- **Demande de connexion**
  - **méthode statique**
    `getConnection(String)` classe `DriverManager`
- **Le driver manager essaye de trouver un driver approprié d'après la chaîne passée en paramètre**
- **Structure de la chaîne décrivant la connexion**

  `jdbc:`*`protocole`*`:`*`URL`*

Exemples

```
jdbc:odbc:epicerie
jdbc:mysql://athens.imaginaire.com:4333/db
jdbc:oracle:thin:@blabla:1715:test
```

Classe `java.sql.Connection`

- **Connexion sans information de sécurité**

```
Connection con = DriverManager.getConnection
   ("jdbc:odbc:epicerie");
```

- **Connexion avec informations de sécurité**

```
Connection con = DriverManager.getConnection
   ("jdbc:odbc:epicerie", user, password);
```

- **Dans tous les cas faut récupérer l'exception `java.sql.SQLException`**

Exemple : se connecter à Oracle

- **Le code suivant se connecte à une base de données Oracle**
  - En utilisant le driver Oracle JDBC Thin
  - 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

```
Connection conn;

try {
  conn = DriverManager.getConnection(
            "jdbc:oracle:thin:@myhost:1521:orcl",
            "theUser", "thePassword");
}
catch (SQLException e) {…}
```

Chargement du pilote MySQL

Code Java	Commentaires

`try`
`{ Class.forName("com.mysql.jdbc.Driver").newInstance();`
`} catch (ClassNotFoundException ex)`
`{ System.out.println`
`("Problème au chargement"+ex.toString()); }`
	Chargement du pilote MySQL.

`try`
`{ Connection cx = DriverManager.getConnection`
`("jdbc:mysql://localhost/bdsoutou?`
`user=soutou&password=iut"); ... }`
	Déclaration d'une connexion.

`catch(SQLException ex)` `{ System.err.println("Erreur : "+ex); }`	Gestion des erreurs.

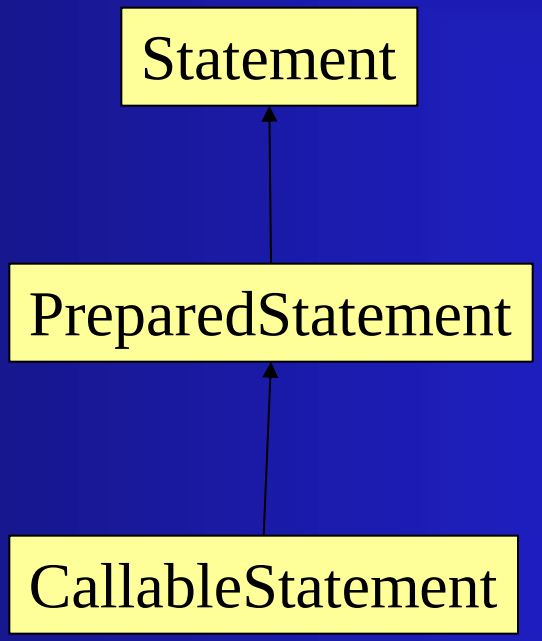
Obtenir des informations sur la base

- **`Connection` peut être utilisé pour obtenir un objet de type `DatabaseMetaData`**
  - Cela offre plusieurs méthodes pour obtenir des informations sur la base

```
Connection conn; …
try {
  DatabaseMetaData dm = conn.getMetaData();
  String  s1 = dm.getURL();
  String  s2 = dm.getSQLKeywords();
  boolean b1 = dm.supportsTransactions();
  boolean b2 = dm.supportsSelectForUpdate();
}
catch (SQLException e) {…}
```

Création d'une requête

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



Création d ’une requête

- **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);
```

Méthodes de l'interface Connection

Méthode	Description
`createStatement()`	Création d'un objet destiné à recevoir un ordre SQL statique, non paramétré.
`prepareStatement(String)`	Précompile un ordre SQL acceptant des paramètres et pouvant être exécuté plusieurs fois.
`prepareCall(String)`	Appel d'une procédure cataloguée (certains pilotes attendent `execute` ou ne reconnaissent pas `prepareCall`).
`void setAutoCommit(boolean)`	Positionne ou non le *commit* automatique.
`void commit()`	Valide la transaction.
`void rollback()`	Invalide la transaction.
`void close()`	Ferme la connexion.

Exécution d'une requête

- **3 types d 'exécution :**
  - `executeQuery()` : pour les requêtes (SELECT) qui retournent un `ResultSet` (tuples résultants)
  - `statement.executeQuery("SELECT ...");`
  - `executeUpdate()` : pour les requêtes (INSERT, UPDATE, DELETE) qui retournent un entier (nombre de tuples traités)
  -
```
int count =
    statement.executeUpdate("UPDATE ...");
```
  - `execute()` : exécuter n'importe quelle requête SQL
    - `statement.execute("DROP TABLE test");`

Méthodes de l'interface Statement

Méthode	Description
`ResultSet executeQuery(String)`	Exécute une requête et retourne un ensemble de lignes (objet `ResultSet`).
`int executeUpdate(String)`	Exécute une instruction SQL et retourne le nombre de lignes traitées (`INSERT`, `UPDATE` ou `DELETE`) ou 0 pour les instructions ne renvoyant aucun résultat (LDD).
`boolean execute(String)`	Exécute une instruction SQL et renvoie `true` si c'est une instruction `SELECT`, `false` sinon (instructions LMD ou plusieurs résultats `ResultSet`).
`Connection getConnection()`	Retourne l'objet de la connexion.
`void setMaxRows(int)`	Positionne la limite du nombre d'enregistrements à extraire par toute requête issue de cet état.
`int getUpdateCount()`	Nombre de lignes traitées par l'instruction SQL (-1 si c'est une requête ou si l'instruction n'affecte aucune ligne).
`void close()`	Ferme l'état.

Statement: deux types d'exécution

- Requètes "`select`" retournant un tableau

```
Statement stmt = con.createStatement();
String sql = "SELECT * FROM fournisseur";
ResultSet rs = stmt.executeQuery(sql);
```

- Requètes de mise à jour et de création

```
Statement stmt = conn.createStatement();
String sql = "INSERT INTO Customers " +
     "VALUES (1001, 'Simpson', 'Mr.', " +
     "'Springfield', 2001)";
int i = stmt.executeUpdate(sql);
```

executeQuery

- **Pour lancer une requète "SELECT" statique**

```
Statement stmt = con.createStatement();
String s = "select * from employes";
ResultSet rs = stmt.executeQuery(s);
```

- **Un `ResultSet` est un objet qui modélise un tableau à deux dimensions**
  - Lignes (*row*)
  - Colonnes (valeurs d'attributs)

executeUpdate () - 1
INSERT INTO

- Pour lancer une requète `INSERT`

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


String s = "INSERT INTO test (code,val)" +
    "VALUES(" + valCode + ", '" +val +"')";
int i = stmt.executeUpdate(s);
```

- Le résultat est un entier donnant le nombre de lignes créées

executeUpdate () - 2
UPDATE

- **Pour lancer une requète UPDATE**

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


String s = "UPDATE table
   SET column = expression
   WHERE predicates";
int i = stmt.executeUpdate(s);
```

- **Le résultat est un entier donnant le nombre de mises-à-jour**

executeUpdate () - 3
DELETE

- **Pour lancer une requète DELETE**

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


String s = "DELETE FROM table
   WHERE predicates";
int i = stmt.executeUpdate(s);
```

- **Le résultat est un entier donnant le nombre de d'effacements**

Traitement des résultats

- **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}
```

Interprétation du résultat

- **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

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

Interprétation du résultat

```
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");
}
```

Les méthodes du ResultSet

ResultSet	
`next() : boolean`	go to next row of results. "false" if no more.
`previous() : boolean`	go to previous row. "false" if 1st result.
`first() : boolean`	go to first row of results.
`last() : boolean`	go to last row of results.
`absolute( k )`	go to k-th row of results.
`getInt( name: String )`	get int value of field "name"
`getInt( index: int )`	get int value of k-th column in a record
`...`	

Etats simples

Code Java	Commentaires
`Statement etatSimple = cx.createStatement();`	Création de l'état.
`etatSimple.execute("CREATE TABLE IF NOT EXISTS Compagnie(comp VARCHAR(4), nomComp VARCHAR(30), CONSTRAINT pk_Compagnie PRIMARY KEY(comp))");`	Ordre LDD.
`int j = etatSimple.executeUpdate("CREATE TABLE IF NOT EXISTS Avion (immat VARCHAR(6),typeAvion VARCHAR(15), cap SMALLINT, compa VARCHAR(4), CONSTRAINT pk_Avion PRIMARY KEY(immat), CONSTRAINT fk_Avion_comp_Compagnie FOREIGN KEY(compa) REFERENCES Compagnie(comp))");`	Ordre LDD (autre écriture), *j* contient 0 (aucune ligne n'est concernée).
`int k = etatSimple.executeUpdate ("INSERT INTO Compagnie VALUES ('AF','Air France')");`	Ordre LMD, *k* contient 1 (une ligne est concernée).
`etatSimple.execute("INSERT INTO Avion VALUES ('F-WTSS','Concorde',90,'AF')");`	Ordres LMD (autres écritures).
`etatSimple.execute("INSERT INTO Avion VALUES ('F-FGFB','A320',148,'AF')");`	
`etatSimple.setMaxRows(10);`	Pas plus de 10 lignes retournées par les prochaines extractions.
`ResultSet curseurJava = etatSimple.executeQuery("SELECT * FROM Avion");`	Chargement d'un curseur Java.
`etatSimple.execute("DELETE FROM Avion");`	Ordre LMD, *l* contient 2 (avions supprimés).
`int l = etatSimple.getUpdateCount();`	

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

Correspondance 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
….	…..

Les méthodes du ResultSet pour récupérer des données

```
getInt( ), getLong( )     - get Integer field value
getFloat( ), getDouble()  - get floating pt. value
getString( )   - get Char or Varchar field value
getDate( )     - get Date or Timestamp field value
getBoolean( )  - get a Bit field value
getBytes( )    - get Binary data
getBigDecimal( ) - get Decimal field as BigDecimal
getBlob( )     - get Binary Large Object
getObject( )   - get any field value
```

Compatibilité de types

```
int pop1 = rs.getInt( "population" );
long pop2 = rs.getLong( "population" );
// float - int conversion is possible, too
float area = rs.getFloat( "surfacearea" );
// convert char(n) to String
String region = rs.getString( "region" );
```

Exemple complet

Code Java	Commentaires
`import java.sql.*;`	Importation du paquetage.
`public class JDBCTest`	Classe ayant une méthode `main`.
`{public static void main(String[] args)`	
`throws SQLException, Exception`	
`{`	
`try`	
`{ System.out.println`	
`("Initialisation de la connexion");`	
`Class.forName`	Chargement du pilote JDBC
`("com.mysql.jdbc.Driver").newInstance();`	MySQL.
`} catch (ClassNotFoundException ex)`	
`{ System.out.println`	
`("Problème au chargement"+ex.toString()); }`	
`try`	
`{ Connection cx = DriverManager.getConnection`	Création d'une connexion.
`("jdbc:mysql://localhost/bdsoutou?`	
`user=soutou&password=iut") ;`	
`Statement etat = cx.createStatement ();`	Création d'un état de connexion.
`ResultSet rset = etat.executeQuery`	Extraction de la date courante.
`("SELECT SYSDATE()");`	
`while (rset.next ())`	
`System.out.println("Nous sommes le : "+`	
`rset.getString (1));`	Affichage du résultat.
`System.out.println("JDBC correctement`	
`configuré");`	
`}`	
`catch(SQLException ex)`	Gestion des erreurs.
`{ System.err.println("Erreur : "+ex); }`	
`}`	
`}`	

Accès aux méta-données d'un ResultSet

- **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()`

Obtenir des informations sur la composition de la table

- `ResultSet` peut être utilisé pour obtenir un objet `ResultSetMetaData`

```
try {
  ResultSet rset = … ;
  ResultSetMetaData md = rset.getMetaData();

  while (rset.next()) {
    for (int i = 0; i < md.getColumnCount(); i++) {
      String lbl = md.getColumnLabel();
      String typ = md.getColumnTypeName(); …
  }
} catch (SQLException e) {…}
```

Extraction de données

Code Java	Commentaires
`try {…`	
`Statement etatSimple = cx.createStatement();`	Création de l'état.
`ResultSet curseurJava =`	Création et charge-
`  etatSimple.executeQuery("SELECT immat, cap FROM Avion`	ment du curseur.
`WHERE comp = (SELECT comp FROM Compagnie WHERE nom-`	
`Comp='Air France')");`	
`float moyenneCapacité =0;`	
`int nbAvions = 0;`	
`while (curseurJava.next())`	Parcours du curseur.
`{System.out.print("Immat : "+curseurJava.getString(1));`	
`System.out.println("Capacité : "+curseurJava.getInt(2));`	Extraction de colonnes.
`moyenneCapacité += curseurJava.getInt(2);`	
`nbAvions ++; }`	
`moyenneCapacité /= nbAvions;`	
`System.out.println("Capacité moy : "+moyenneCapacité);`	
`curseurJava.close();`	Fermeture du curseur.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Requêtes précompilées

- **Si vous avez besoin d'exécuter une requête plusieurs fois, avec des paramètres variables**
  - Utiliser un objet **`PreparedStatement`**
  - Identifier les variables liées avec un signe `?`

```
try {
  Connection conn = DriverManager.getConnection(…);

  PreparedStatement pstmt =
    conn.prepareStatement("update EMP set SAL = ?");
  …
} catch (SQLException e) {…}
```

Méthodes de l'interface PreparedStatement

Méthode	Description
`ResultSet executeQuery()`	Exécute la requête et retourne un curseur ni navigable, ni modifiable par défaut.
`int executeUpdate()`	Exécute une instruction LMD (`INSERT`, `UPDATE` ou `DELETE`) et retourne le nombre de lignes traitées, ou 0 pour les instructions SQL ne retournant aucun résultat (LDD).
`boolean execute()`	Exécute une instruction SQL et renvoie `true`, si c'est une instruction `SELECT`, `false` sinon.
`void setNull(int, int)`	Affecte la valeur `NULL` au paramètre de numéro et de type (classification `java.sql.Types`) spécifiés.
`void close()`	Ferme l'état.

Lier des variables et executer une requête

`PreparedStatement`

```
try {
  PreparedStatement pstmt =
    conn.prepareStatement("update EMP set SAL = ?");
  …
  pstmt.setBigDecimal(1, new BigDecimal(55000));
  pstmt.executeUpdate();

  pstmt.setBigDecimal(1, new BigDecimal(65000));
  pstmt.executeUpdate();
  …
} catch (SQLException e) {…}
```

Insertion d'un enregistrement par un ordre préparé

Code Java	Commentaires
`try { …`	
`String ordreSQL =`	
`"INSERT INTO Avion VALUES (?, ?, ?, ?)";`	Création d'un état préparé.
`PreparedStatement étatPréparé =`	
`cx.prepareStatement(ordreSQL);`	
`étatPréparé.setString(1, "F-NEW");`	Passage des paramètres.
`étatPréparé.setString(2, "A319");`	
`étatPréparé.setInt(3, 178);`	
`étatPréparé.setString(4, "AF");`	
`System.out.println(étatPréparé.executeUpdate()`	Exécution de l'instruction.
`+ " avion inséré.");`	
`étatPréparé.close();`	Fermeture de l'état.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Effacement par un ordre préparé

Code Java	Commentaires
`try { …`	
`  cx.setAutoCommit(false);`	
`  String ordreSQL =`	
`  "DELETE FROM Avion WHERE immat = ?";`	
`  PreparedStatement étatPréparé =`	
`  cx.prepareStatement(ordreSQL);`	Création d'un état préparé.
`  étatPréparé.setString(1, "F-NEW ");`	Passage du paramètre.
`if (! étatPréparé.execute() )`	Exécution de l'instruction.
`  { System.out.println("Enregistrement sup-primé");`	
`  cx.commit(); }`	
`  étatPréparé.close();`	Fermeture de l'état.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Transactions

- **Les Transactions sont gérées par la propriété `autoCommit dans la classe Connection`**
  - `true` **initiallement,** crée une transaction séparée par instruction SQL

```
Connection conn = DriverManager.getConnection(…);
conn.setAutoCommit(false);  // No autocommits now
…                           // Issue SQL statements
conn.commit();  … or …      // Commit transaction
conn.rollback();            // Rollback transaction
```

Exemple

Code Java	Commentaires
`try { …`	
`cx.setAutoCommit(false);`	Désactivation de *l'autocommit*.
`String ordreSQL =`	
`"INSERT INTO Avion VALUES (?, ?, ?, ?)";`	
`PreparedStatement étatPréparé =`	
`cx.prepareStatement(ordreSQL);`	
`Savepoint p1 = cx.setSavepoint("P1");`	Création d'un état appelable. Création du point de validation P1.
`étatPréparé.setString(1, "F-NEW2");`	Passage de paramètres et première insertion.
`…`	
`if (! étatPréparé.execute() )`	
`System.out.println("F-NEW2 inséré");`	
`Savepoint p2 = cx.setSavepoint("P2");`	Création du point de validation P2.
`étatPréparé.setString(1, "F-NEW3");`	Passage de paramètres et deuxième insertion.
`…`	
`if (! étatPréparé.execute() )`	
`System.out.println("F-NEW3 inséré");`	
`cx.rollback(p2);`	Annulation de la deuxième partie.
`cx.commit();`	Validation de la première partie.
`cx.close();`	Fermeture de la connexion.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Traitement des exceptions

Méthode	Description
`String getMessage()`	Message décrivant l'erreur.
`String getSQLState()`	Code erreur SQL Standard (XOPEN ou SQL99).
`int getErrorCode()`	Code erreur SQL de la base.
`SQLException getNextException()`	Chaînage à l'exception suivante (si une erreur renvoie plusieurs messages).

Code Java	Commentaires
`import java.sql.*;`	
`public class Exceptions1`	
`{ public static void main (String args []) throws SQLException, Exception`	Classe principale.
`{try`	
`{Class.forName ("com.mysql.jdbc.Driver").newInstance();}`	Chargement du pilote.
`catch (ClassNotFoundException ex) {System.out.println ("Problème au chargement"+ex.toString()); }`	
`try`	Connexion.
`{ Connection cx = DriverManager.getConnection(...`	Instructions...
`cx.close(); }`	
`catch(SQLException ex)`	Gestion des erreurs.
`{System.err.println("Erreur");`	
`while ((ex != null))`	
`{System.err.println("Statut : "+ ex.getSQLState());`	
`System.err.println("Message : "+ ex.getMessage());`	
`System.err.println("Code base : "+ ex.getErrorCode());`	
`ex = ex.getNextException();}`	
`} } }`	

Curseurs modifiables

Code Java	Commentaires

`try {…`
`Statement etatSimple =`
`cx.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,`
`ResultSet.CONCUR_UPDATABLE);`
`cx.setAutoCommit(false);`
	Création de l'état et désactivation de la validation automatique.

`ResultSet curseurModifJava = etatSimple.executeQuery` `("SELECT immat,typeAvion,cap FROM Avion");`	Création du curseur.

`if (curseurModifJava.absolute(3))`
`{ curseurModifJava.deleteRow();`
`cx.commit(); }`
	Accès direct au troisième avion, suppression de l'enregistrement.

`else` `System.out.println("Pas de 3ème avion!");`	
`curseurModifJava.close();`	Fermeture du curseur.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Curseurs modifiables

Code Java	Commentaires
`try {…`	
`Statement etatSimple =`	
`cx.createStatement(ResultSet.TYPE_FORWARD_ONLY,`	
`ResultSet.CONCUR_UPDATABLE);`	Création de l'état et désactivation de la validation automatique.
`cx.setAutoCommit(false);`	
`ResultSet curseurModifJava = etatSimple.executeQuery`	Création du curseur.
`("SELECT immat,typeAvion,cap FROM Avion");`	
`String p_immat = "F-GLFS";`	
`while(curseurModifJava.next())`	Accès à l'enregistrement et suppression.
`{if (curseurModifJava.getString(1).equals(p_immat))`	
`{curseurModifJava.deleteRow();}`	
`}`	
`curseurModifJava.close();`	Fermeture du curseur.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Bonnes pratiques avec JDBC

Généralités

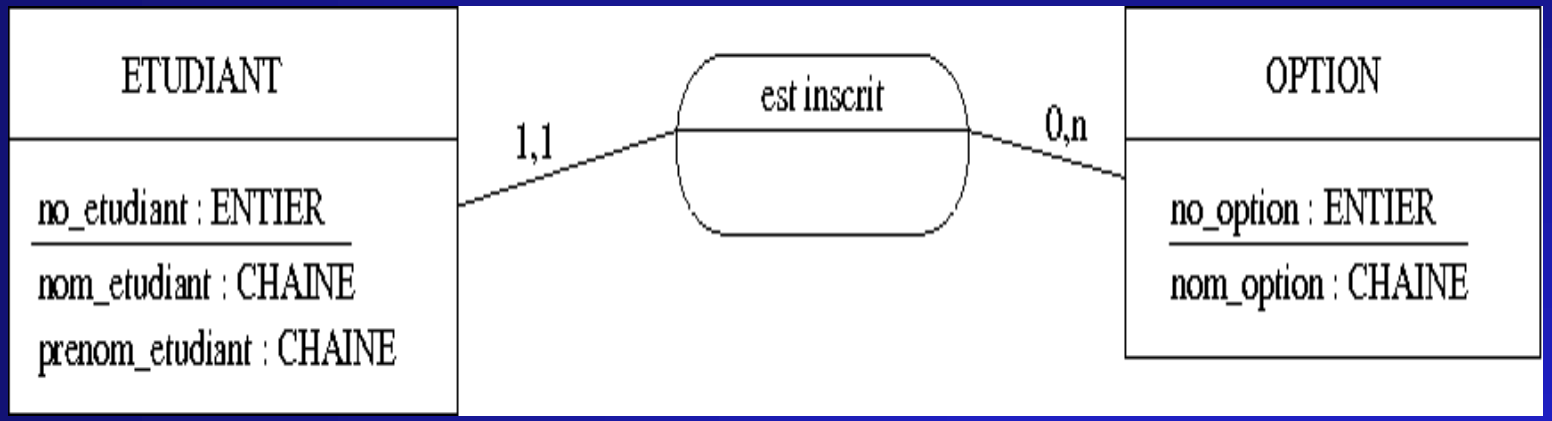
- **Un *design pattern* est un patron de conception : Solution standard, testée et éprouvée par la communauté *objet*, pour répondre à un problème d'architecture ou de conception de logiciel.**
- **Concept issu des travaux du fameux *Gang of Four* (GoF : Erich Gamma, Richard Helm, Ralph Johnson, et John Vlissides).**
- **publiés dans « Design Patterns: Elements of Reusable Object-Oriented Software » édité en 1995 et proposant 23 motifs de conception.**

Pourquoi un patron de conception ?

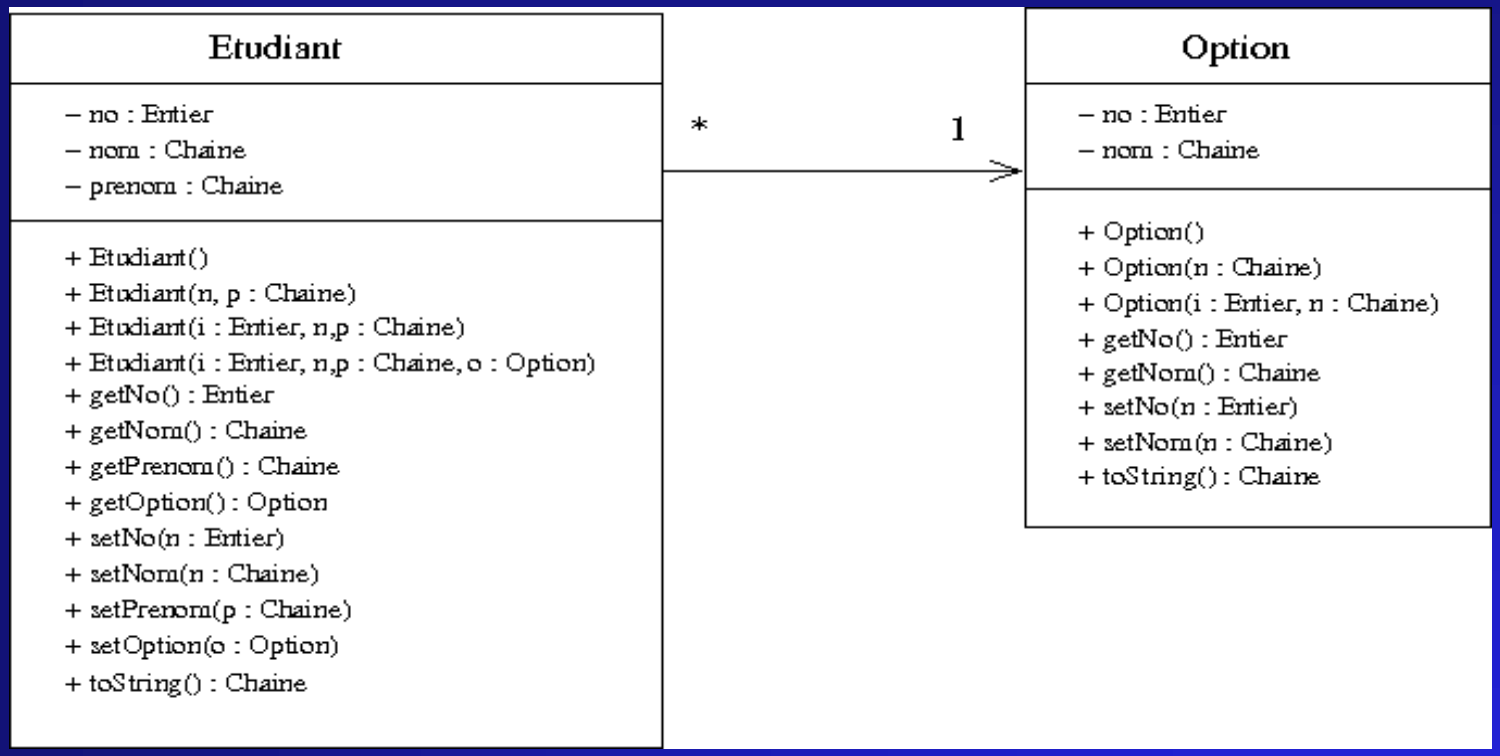
- Quand on accède à une base de données depuis un langage, il y a de nombreuses façons de faire.
- Où mettre le code SQL ?
- Que faire pour simplifier un futur changement de base de données cible ?
- Comment obtenir une application la plus modulaire possible, la plus facilement modifiable et la plus réutilisable ?

→ Le pattern DAO répond à ces questions !

Exemple


ETUDIANT
no_etudiant : ENTIER
nom_etudiant : CHAINE
prenom_etudiant : CHAINE
1,1
est inscrit
0,n
OPTION
no_option : ENTIER
nom_option : CHAINE

Exemple


Etudiant
– no : Entier
– nom : Chaine
– prenom : Chaine
+ Etudiant()
+ Etudiant(n, p : Chaine)
+ Etudiant(i : Entier, n,p : Chaine)
+ Etudiant(i : Entier, n,p : Chaine, o : Option)
+ getNo() : Entier
+ getNom() : Chaine
+ getPrenom() : Chaine
+ getOption() : Option
+ setNo(n : Entier)
+ setNom(n : Chaine)
+ setPrenom(p : Chaine)
+ setOption(o : Option)
+ toString() : Chaine
*
1
Option
– no : Entier
– nom : Chaine
+ Option()
+ Option(n : Chaine)
+ Option(i : Entier, n : Chaine)
+ getNo() : Entier
+ getNom() : Chaine
+ setNo(n : Entier)
+ setNom(n : Chaine)
+ toString() : Chaine

Exemple

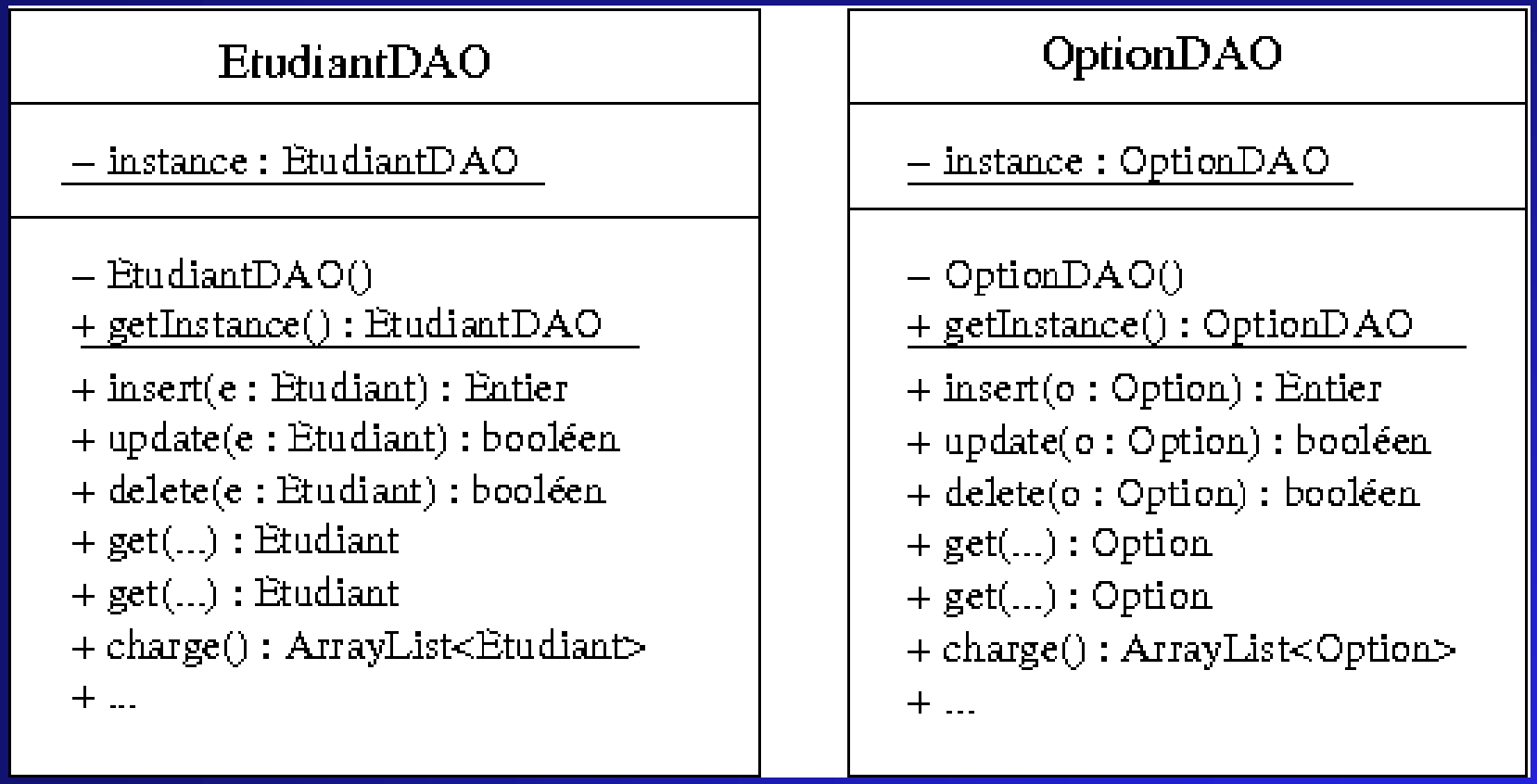
- Se pose maintenant la question du placement du code SQL chargé de créer ces objets depuis la base, ou de mettre à jour la base en fonction de ces objets.
- SQL est un langage normalisé, certes, mais chaque SGBD a ses particularités et son dialecte propre.
- Imaginons que le code SQL est placé dans les objets métier. Lors d'un changement de SGBD cible, ou si on décide d'abandonner la solution SGBD pour une solution XML par exemple, il faudra modifier les objets métiers. Cette solution n'est donc pas la bonne.

Le pattern DAO

- Le patron de conception DAO propose la création d'une classe DAO par classe métier.
- Chaque classe DAO contient les méthodes de liaison avec la base de données, parfois appelées CRUD (pour *Create, Request, Update, Delete*).

- Les méthodes de suppression et de modification renvoient un booléen indiquant le succès de l'opération, la méthode d'insertion renvoie l'identifiant affecté à la nouvelle ligne de la table (utile en cas d'identifiant auto-incrémenté par le SGBD), et plusieurs méthodes get permettent d'obtenie un objet en fonction de différents critères de recherche.

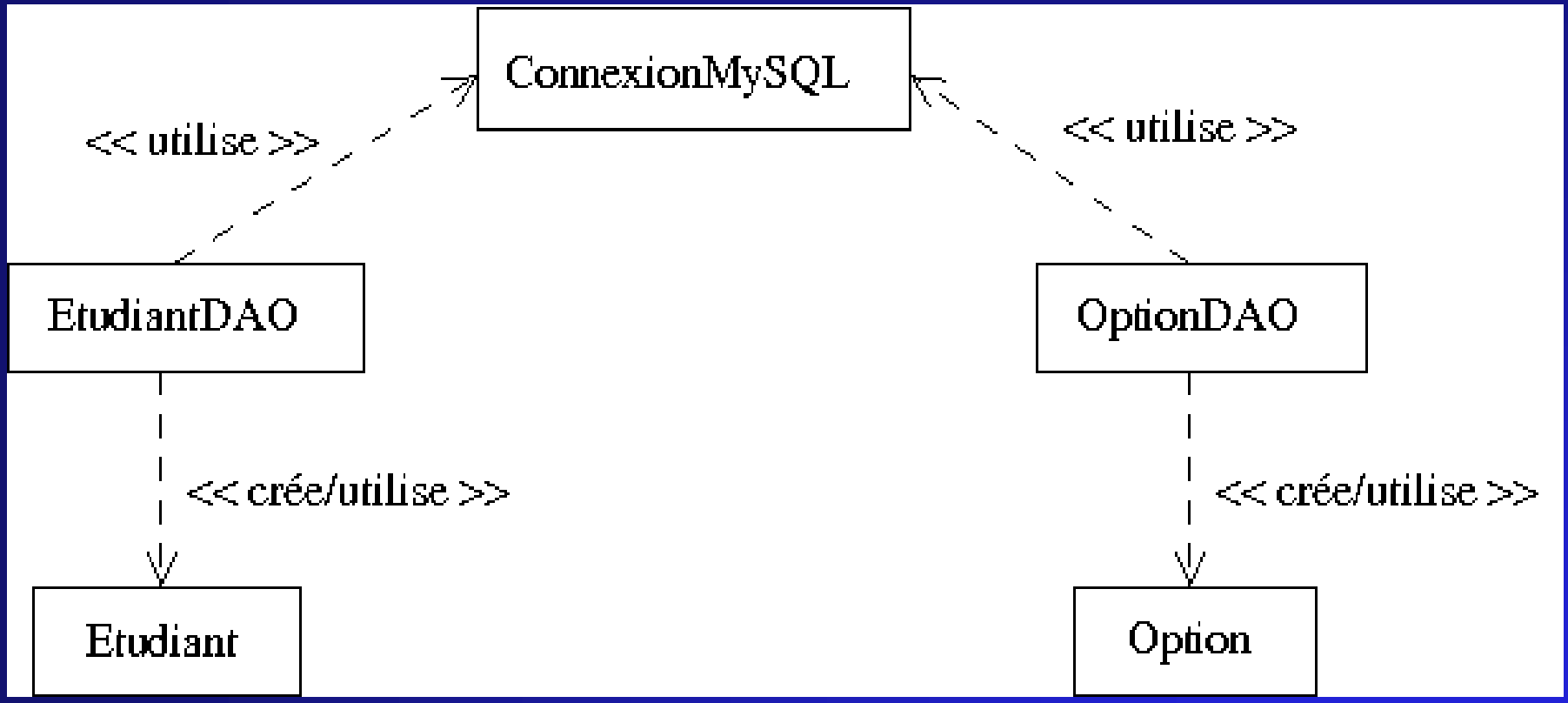
Le pattern DAO



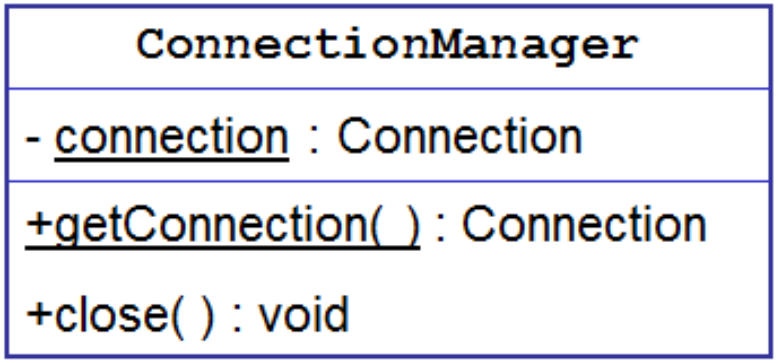
Le pattern Singleton

- Ces objets DAO implémentent le patron de conception Singleton, afin de s'assurer qu'il n'y ait qu'une seule instance de chaque classe.
- Les constructeurs sont donc privés, une méthode de classe publique getInstance permettant d'accéder à l'unique objet instance de chaque classe DAO.
- Le premier appel à cette méthode crée l'objet et le renvoie, les suivants se contentant de renvoyer l'objet préalablement créé.

Diagramme de classes final



Créer une classe pour gérer la connexion à une base de données




```
// example how to use
Statement statement =
    ConnectionManager.getConnection().createStatement( );
```

```
public class ConnectionManager {
 // literal constants in Java code is baaaad code.
 // we will change to a configuration file later.
 private static String driver = "com.mysql.jdbc.Driver";
 private static String url = "jdbc:mysql://hostname/world";
 private static String user = "student";
 private static String password = "student";
 /* a single shared database connection */
 private static Connection connection = null;

 private ConnectionManager() { /* no object creation */ }
```

```
/* the public accessor uses lazy instantiation */
 public static Connection getConnection( ) throws ... {
    if ( connection == null ) connection = makeConnection();
    return connection;
 }
```

```
 private static Connection makeConnection( )
                                throws SQLException {
    try {
       Class.forName( driver );
       // load the database driver class
       connection = DriverManager.getConnection(
                    url, user, password );
    } catch ( FileNotFoundException ex ) {
       logger.error("connection error", ex ); // Logging
       throw new SQLException( ex );
    }
 }

/* the public accessor uses lazy instantiation */
 public static Connection getConnection( ) throws ... {
    if ( connection == null ) connection = makeConnection();
    return connection;
 }
```

Principe d'une classe DAO

```
public class CityDao {
    private static final Logger logger = ...; // log4J
    private static final CountryDao countryDao;
    private static HashMap<Integer,City> cache = ...;

    /** retrieve a city by its id */
    public City findById( Integer id ) {
        if ( cache.containsKey(id) ) return cache.get(id);
        List<City> list = find("WHERE id = "+id);
        return list.get(0);
    }

    /** retrieve a city by name */
    public List<City> findByName( String name ) {
        name = sanitize( name );
        List<City> list = find("WHERE name = '"+name+"'");
        return list;
    }
```

```
/** find cities using a general query, use a
 *  WHERE ..., HAVING ..., or other selection clause */
public List<City> find( String query ) {
    List<City> list = new ArrayList<City>( );
    Statement stmt = ConnectionManager
                     .getConnection( ).createStatement();
    String sqlquery = "SELECT * FROM city c " + query;
    try {
        logger.debug("executing query: " + sqlquery );
        ResultSet rs = stmt .executeQuery( sqlquery );
        while ( rs.next() ) {
            City c = resultSetToCity( rs );
            list.add( c );
        }
    } catch ( SQLException sqle ) {
        logger.error( "error executing: "+sqlquery, sqle);
    } finally {
        if (stmt!=null) try { stmt.close(); }
        catch(SQLException e) { /* forget it */ }
    return list;
}
```

```
/** convert a ResultSet entry to a City object */
private City resultSetToCity(ResultSet rs)
        throws SQLException {
    City city = null;

    Integer id = rs.getInt("id");
    // is this city already in cache? if so, use it
    if ( cache.contains(id) ) city = cache.get(id);
    else city = new City();

    city.setId(id);
    city.setName( rs.getString("Name") );
    city.setDistrict( rs.getString("District") );
    city.setPopulation( rs.getInt("Population") );
    String countrycode = rs.getString("countrycode");
```

```
        // add this city to the cache
        if ( ! cache.containsKey(id) ) cache.put(id, city);

        // now get reference to the country this city refers
        logger.info("get country for city "+city.getName() );
        Country country = countryDao.findById( countrycode );
        city.setCountry( country );

        return city;
    }
```

```
public boolean delete( City city ) {
  if ( city == null || city.getId() == null ) return false;
  Long id = city.getId( );
  Statement statement = ConnectionManager.getStatement( );
  int count = 0;
  if ( statement == null ) return false;
  String query = "DELETE FROM city WHERE id=" + id;
  try {
     count = statement.executeUpdate( query );
  } catch ( SQLException sqle ) {
     logger.error( "error executing: "+query, sqle );
  } finally {
     ConnectionManager.closeStatement( statement );
  }
  // is city in the cache?
  if ( cache.containsKey(id) ) cache.remove( id );
  return count > 0;
}
```

```
/** prompt for a city name and display city info */
private void citySearch( ) {
    out.print("Input name of city: ");
    String name = in.next().trim();

    // run the query
    City city = cityDao.findByName( name );
    if ( city == null ) {
        out.println("Sorry, no match or query error");
    }
    else {
        out.println("Name: "+city.getName( ) );
        out.println("District: "+city.getDistrict( ) );
        out.println("Country: "
                +city.getCountry( ).getName( ) );
        ...
    }
}
```

```
private void countrySearch() {

  out.print("Input name of country: ");
  String name = in.next().trim();
  // perform the query
  List<Country> results = countryDao.findByName( name );
  if ( results == null )  ...              // failed

  for( Country country : results ) {
    out.printf("Name: %s\n",  country.getName() );
    out.printf("Capital: %s\n", country.getCapital() );
    out.printf("Region: %s\n", country.getRegion() );
```

JDBC ET SWING (JTABLE)

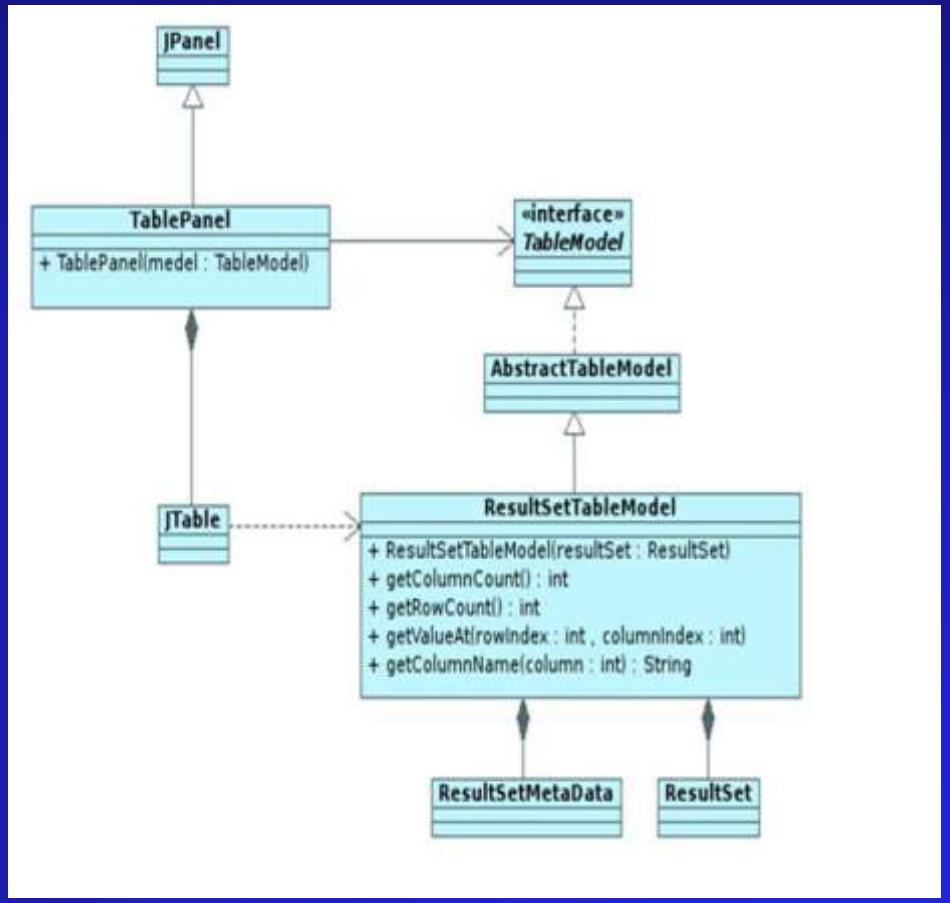
Exemple d'une JTable

```
import javax.swing.*;   import java.awt.*;
class TablePlanetes extends JPanel {
  private Object[][] cellules = {  { "Mercure", new Double(2440), new Integer(0), "non"},
    { "Vénus", new Double(6052), new Integer(0), "non"},
    { "Terre", new Double(6378), new Integer(1), "non"},
    { "Mars", new Double(3397), new Integer(2), "non"},
    { "Jupiter", new Double(71492), new Integer(16), "oui"},
    { "Saturne", new Double(60268), new Integer(18), "oui"}  };
   private String[] columnNames = { "Planète", "Rayon", "Lunes", "Gazeuse"};
   public TablePlanetes() {  setLayout(new BorderLayout());
              JTable table = new JTable(cellules, columnNames);
              add(new JScrollPane(table), BorderLayout.CENTER);
   }
   public static void main(String args[]) {
     TablePlanetes window = new  TablePlanetes();
     JFrame mainFrame = new JFrame( "Affiche table" );
     mainFrame.add( window, BorderLayout.CENTER );
     mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE );
     mainFrame.setSize( 300, 200 );        mainFrame.setVisible( true );
   }   }
```

Planète	Rayon	Lunes	Gazeuse
Mercure	2440.0	0	non
Vénus	6052.0	0	non
Terre	6378.0	1	non
Mars	3397.0	2	non
Jupiter	71492.0	16	oui
Saturne	60268.0	18	oui
Uranus	25559.0	17	oui
Neptune	24766.0	8	oui
Pluton	1137.0	1	non

Afficher le résultat d'une requête dans JTable

- Créer une classe qui dérive de la classe *AbstractTableModel* qui utilise les données du *ResultSet* : le résultat d'une requête SQL.
- Pour afficher le résultat, on intègre le *JTable* dans un *Jpanel (JFrame)*.
- Le diagramme ci-contre, représente les différents composants utilisés :



Afficher le résultat d'une requête dans JTable

- La classe *ResultSetTableModel* est chargée de fournir les données à *JTable*
- en redéfinissant les méthodes suivantes :
  - *public int getColumnCount()* - renvoi le nombre de colonnes
  - *public int getRowCount()* - renvoi le nombre de lignes
  - *public String getColumnName(int column)* - pour utiliser dans *JTable* les noms des champs comme titres de colonnes
  - *public Object getValueAt(int rowIndex, int columnIndex)* - pour afficher dans les cellules *JTable* les valeurs des champs.
  - *ResultSetMetaData rsmd = rs.getMetaData();* - on récupère la structure du résultat de la requête SQL.
  - *NumOfCol = rsmd.getColumnCount();* - retourne le nombre de colonnes dans le *ResultSet*.
  - *columnNames[i] = new String(rsmd.getColumnName(i+1));* - obtient le nom du ième attribut de la table dans le ResultSet.
  - *cellules = new String[rowCount][NumOfCol];* - les valeurs dans le *ResultSet*

- Déterminer le nombre de lignes dans le *ResultSet*
  - *rs.last();* - se positionner à la dernière ligne du *ResultSet*
  - *rowCount = rs.getRow();* - Récupérer le numéro de la ligne
  - *rs.beforeFirst();* - Revenir au début du *ResultSet*
- Obtenir les valeurs des cellules de la table
  - *cellules[j][i] = new String(rs.getString(i+1));*
- Modèle par défaut pour JTable avec les lignes et colonnes:
  - *DefaultTableModel model = new DefaultTableModel(cellules,columnNames);*
- Création de l'objet JTable
  - *JTable table = new JTable(model);*
- La fenêtre, le gestionnaire de placement et les ascenseurs
  - *JFrame mfr = new JFrame("Résultat de la requête SQL");*
  - *mfr.setLayout(new BorderLayout());*
  - *mfr.add(new JScrollPane(table), BorderLayout.CENTER);*

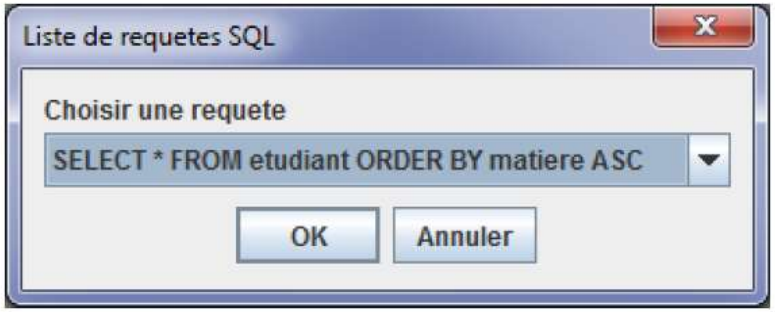
Afficher le résultat d'une requête dans JTable

- Une interface SQL pour faciliter l'interrogation de la BD par une liste
- déroulante de requêtes prédéfinies :

```
import javax.swing.*;
public class interfaceSQL {
  public interfaceSQL() {
     String[] choix = {  "SELECT prenom, nom, age FROM etudiant",
                         "SELECT * FROM etudiant ORDER BY moyenne DESC",
                         "SELECT * FROM etudiant ORDER BY matiere ASC",
                         "SELECT * FROM etudiant ORDER BY age DESC"      };
     String s = (String)JOptionPane.showInputDialog(  null, "Choisir une requete",
        "Liste de requetes SQL",  JOptionPane.PLAIN_MESSAGE, null,  choix,  "SQL" );
     new GetSQLResult(s);
  }
}
```

Afficher le résultat d'une requête dans JTable

- Lancer une requête :



- Résultat de la requête :

Résultat de la requête SQL

id	prenom	nom	age	annee	matiere	moyenne
4	Fred	Gordini	25	2	Com	14.25
5	Jean	Mulini	45	1	Elec	14.63
1	Ivan	Tiradi	25	2	Info	17.99
3	George	Natalon	24	2	Info	15.24
2	Stephane	Marini	21	1	Math	19.99

Premier exemple simple sans ihm

```
// Displaying the contents of the bikes table.
import java.sql.Connection;
import java.sql.Statement;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;

public class DisplayBikes{
  // JDBC driver name and database URL
  static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
  static final String DATABASE_URL = "jdbc:mysql://localhost:3310/bikedb";

  // launch the application
  public static void main( String args[] ) {
    Connection connection = null; // manages connection
    Statement statement = null; // query statement

    // connect to database bikes and query database
    try
    {
      Class.forName( JDBC_DRIVER ); // load database driver class
```

Type 4 JDBC driver (pure Java driver) to connect to MySQL RDBMs

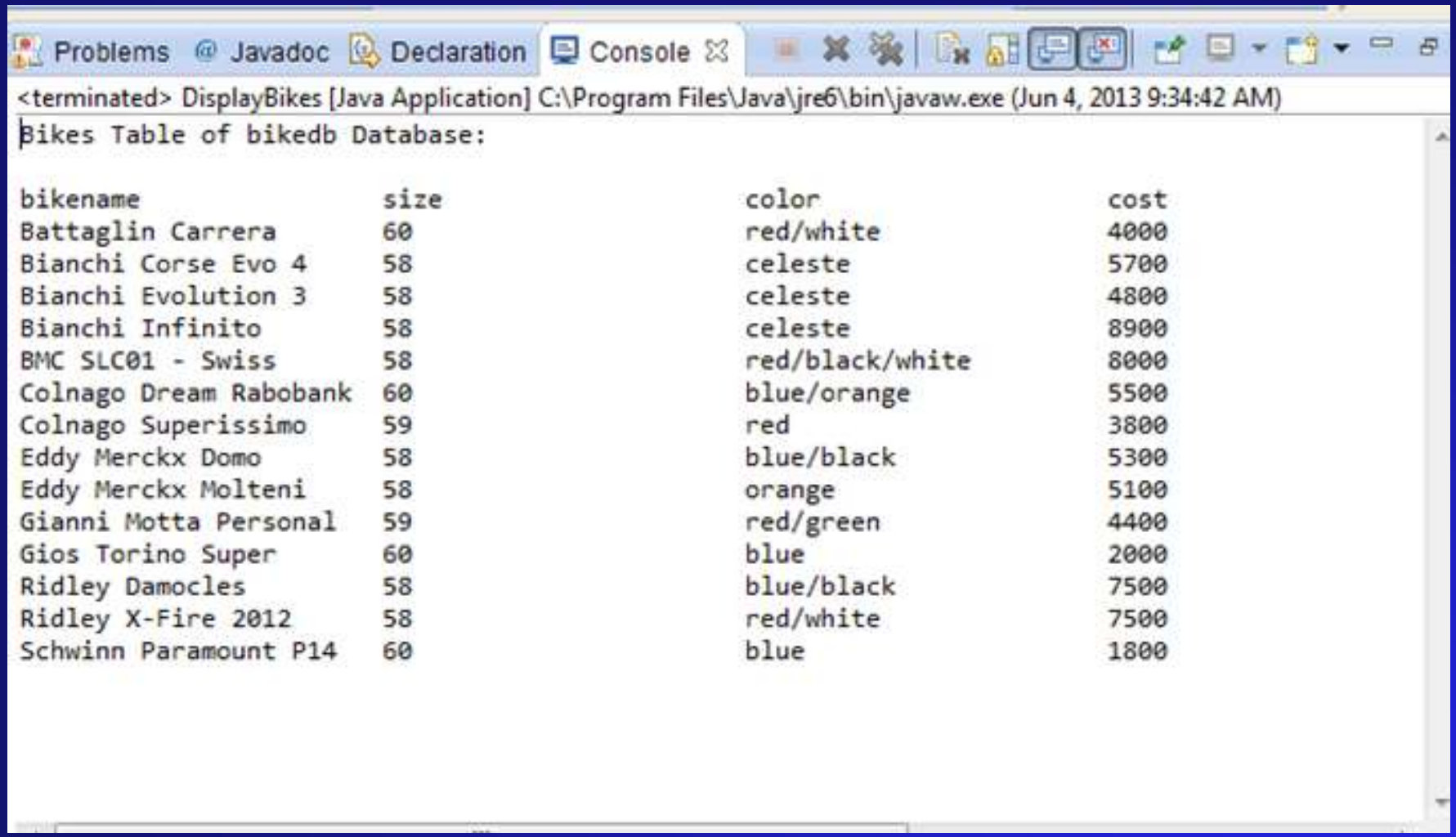
Specify name of database to connect to as well as JDBC driver protocol.

```
// establish connection to database
connection =
  DriverManager.getConnection( DATABASE_URL, "root", "root" );

// create Statement for querying database
statement = connection.createStatement();

// query database
ResultSet resultSet = statement.executeQuery(
  "SELECT bikename, cost, mileage FROM bikes" );

// process query results
ResultSetMetaData metaData = resultSet.getMetaData();
int numberOfColumns = metaData.getColumnCount();
System.out.println( "Bikes Table of bikedb Database:" );

for ( int i = 1; i <= numberOfColumns; i++ )
  System.out.printf( "%-20s\t", metaData.getColumnName( i ) );
System.out.println();

while ( resultSet.next() )
{
  for ( int i = 1; i <= numberOfColumns; i++ )
    System.out.printf( "%-20s\t", resultSet.getObject( i ) );
```

The MySQL query to be executed remotely.

Get metadata from the resultSet to be used for the output formatting.

```
        System.out.println();
      } // end while
    }  // end try
    catch ( SQLException sqlException )  {
      sqlException.printStackTrace();
      System.exit( 1 );
    } // end catch
    catch ( ClassNotFoundException classNotFound ) {
      classNotFound.printStackTrace();
      System.exit( 1 );
    } // end catch
    finally  { // ensure statement and connection are closed properly
      try {
          statement.close();
          connection.close();
      } // end try
      catch ( Exception exception ) {
        exception.printStackTrace();
        System.exit( 1 );
      } // end catch
    } // end finally
  } // end main
}  // end class DisplayBikes
```


Problems
Javadoc
Declaration
Console
<terminated> DisplayBikes [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Jun 4, 2013 9:34:42 AM)
Bikes Table of bikedb Database:
bikename size color cost
Battaglin Carrera 60 red/white 4000
Bianchi Corse Evo 4 58 celeste 5700
Bianchi Evolution 3 58 celeste 4800
Bianchi Infinito 58 celeste 8900
BMC SLC01 - Swiss 58 red/black/white 8000
Colnago Dream Rabobank 60 blue/orange 5500
Colnago Superissimo 59 red 3800
Eddy Merckx Domo 58 blue/black 5300
Eddy Merckx Molteni 58 orange 5100
Gianni Motta Personal 59 red/green 4400
Gios Torino Super 60 blue 2000
Ridley Damocles 58 blue/black 7500
Ridley X-Fire 2012 58 red/white 7500
Schwinn Paramount P14 60 blue 1800

JDBC et SWING

```
// A TableModel that supplies ResultSet data to a JTable.
import java.sql.Connection;
import java.sql.Statement;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import javax.swing.table.AbstractTableModel;

// ResultSet rows and columns are counted from 1 and JTable
// rows and columns are counted from 0. When processing
// ResultSet rows or columns for use in a JTable, it is
// necessary to add 1 to the row or column number to manipulate
// the appropriate ResultSet column (i.e., JTable column 0 is
// ResultSet column 1 and JTable row 0 is ResultSet row 1).
public class ResultSetTableModel extends AbstractTableModel {
  private Connection connection;
  private Statement statement;
  private ResultSet resultSet;
  private ResultSetMetaData metaData;
  private int numberOfRows;

  // keep track of database connection status
  private boolean connectedToDatabase = false;
```

This class maintains the connection to the database and handles the results being returned from the database to the application.

```
// constructor initializes resultSet and obtains its meta data object;
  // determines number of rows
  public ResultSetTableModel( String driver, String url,
    String username, String password, String query )
    throws SQLException, ClassNotFoundException
  {
    // load database driver class
    Class.forName( driver );

    // connect to database
    connection = DriverManager.getConnection( url, username, password );

    // create Statement to query database
    statement = connection.createStatement(
      ResultSet.TYPE_SCROLL_INSENSITIVE,
      ResultSet.CONCUR_READ_ONLY );

    // update database connection status
    connectedToDatabase = true;

    // set query and execute it
    setQuery( query );
  } // end constructor ResultSetTableModel
```

```
// get class that represents column type
public Class getColumnClass( int column ) throws IllegalStateException {
   // ensure database connection is available
  if ( !connectedToDatabase )
    throw new IllegalStateException( "Not Connected to Database" );

  // determine Java class of column
  try {
    String className = metaData.getColumnClassName( column + 1 );
    // return Class object that represents className
    return Class.forName( className );
  } // end try
  catch ( Exception exception ) {
     exception.printStackTrace();
  } // end catch

  return Object.class; // if problems occur above, assume type Object
} // end method getColumnClass

// get number of columns in ResultSet
public int getColumnCount() throws IllegalStateException {
   // ensure database connection is available
  if ( !connectedToDatabase )
    throw new IllegalStateException( "Not Connected to Database" );
```

```
   // determine number of columns
   try {
      return metaData.getColumnCount();
   } // end try
   catch ( SQLException sqlException ) {
      sqlException.printStackTrace();
   } // end catch

   return 0; // if problems occur above, return 0 for number of columns
} // end method getColumnCount

// get name of a particular column in ResultSet
public String getColumnName( int column ) throws IllegalStateException {
   // ensure database connection is available
   if ( !connectedToDatabase )
      throw new IllegalStateException( "Not Connected to Database" );

   // determine column name
   try {
      return metaData.getColumnName( column + 1 );
   } // end try
   catch ( SQLException sqlException ) {
      sqlException.printStackTrace();
   } // end catch
```

```
    return ""; // if problems, return empty string for column name
  } // end method getColumnName

  // return number of rows in ResultSet
  public int getRowCount() throws IllegalStateException {
      // ensure database connection is available
    if ( !connectedToDatabase )
      throw new IllegalStateException( "Not Connected to Database" );

    return numberOfRows;
  } // end method getRowCount

  // obtain value in particular row and column
  public Object getValueAt( int row, int column )
    throws IllegalStateException
  {
    // ensure database connection is available
    if ( !connectedToDatabase )
      throw new IllegalStateException( "Not Connected to Database" );

    // obtain a value at specified ResultSet row and column
    try {
      resultSet.absolute( row + 1 );
      return resultSet.getObject( column + 1 );
    } // end try
```

```
 catch ( SQLException  sqlException  ) {
      sqlException.printStackTrace();
  } // end catch

  return ""; // if problems,  return empty  string  object
} // end method  getValueAt

// set new database query  string
public  void  setQuery( String  query )
  throws SQLException,  IllegalStateException
{
  // ensure database connection  is available
  if ( !connectedToDatabase  )
    throw new IllegalStateException(  "Not  Connected  to Database" );

   // specify  query  and execute it
  resultSet  = statement.executeQuery(  query  );

  // obtain  meta data  for ResultSet
  metaData = resultSet.getMetaData();

  // determine  number  of rows in ResultSet
  resultSet.last();                  // move  to last row
  numberOfRows  = resultSet.getRow();   // get row number
```

```
    // notify JTable that model has changed
    fireTableStructureChanged();
  } // end method setQuery

  // close Statement and Connection
  public void disconnectFromDatabase() {
     if ( !connectedToDatabase )
      return;

    // close Statement and Connection
    try {
      statement.close();
      connection.close();
    } // end try
    catch ( SQLException sqlException ) {
      sqlException.printStackTrace();
    } // end catch
    finally // update database connection status
    {
      connectedToDatabase = false;
    } // end finally
  } // end method disconnectFromDatabase
} // end class ResultSetTableModel
```

```
// Display the contents of the bikes table in the bikedb database.
import java.awt.BorderLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.sql.SQLException;
import javax.swing.JFrame;
import javax.swing.JTextArea;
import javax.swing.JScrollPane;
import javax.swing.ScrollPaneConstants;
import javax.swing.JTable;
import javax.swing.JOptionPane;
import javax.swing.JButton;
import javax.swing.Box;

public class DisplayQueryResults extends JFrame
{
  // JDBC driver, database URL, username and password
  static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
  static final String DATABASE_URL = "jdbc:mysql://localhost:3310/bikedb";
  static final String USERNAME= "root";
  static final String PASSWORD= "root";
```

This class extends the JFrame class and creates the user interface (GUI) and creates an instance of the ResultSetTableModel to handle the data displayed in the JTable.

```
// default query retrieves all data from bikes table
static final String DEFAULT_QUERY = "SELECT * FROM bikes";

private ResultSetTableModel tableModel;
private JTextArea queryArea;

// create ResultSetTableModel and GUI
public DisplayQueryResults() {
    super( "Displaying Query Results" );
    // create ResultSetTableModel and display database table
  try {
    // create TableModel for results of query SELECT * FROM bikes
    tableModel = new ResultSetTableModel( JDBC_DRIVER, DATABASE_URL,
      USERNAME, PASSWORD, DEFAULT_QUERY );

    // set up JTextArea in which user types queries
    queryArea = new JTextArea( DEFAULT_QUERY, 3, 100 );
    queryArea.setWrapStyleWord( true );
    queryArea.setLineWrap( true );

    JScrollPane scrollPane = new JScrollPane( queryArea,
      ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED,
      ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER );
```

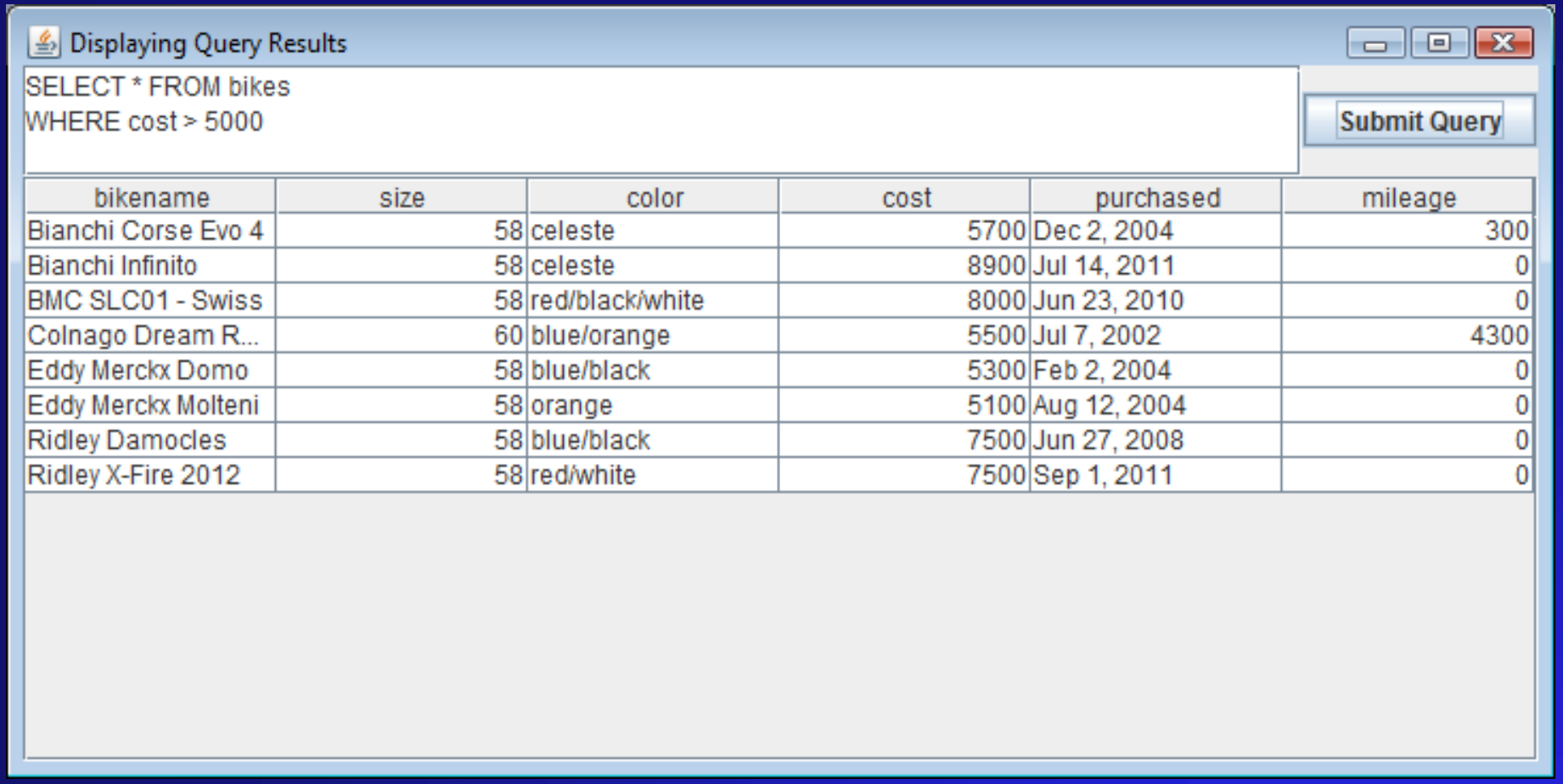
```
// set up JButton for submitting queries
JButton submitButton = new JButton( "Submit Query" );

// create Box to manage placement of queryArea and
// submitButton in GUI
Box box = Box.createHorizontalBox();
box.add( scrollPane );
box.add( submitButton );

// create JTable delegate for tableModel
JTable resultTable = new JTable( tableModel );

// place GUI components on content pane
add( box, BorderLayout.NORTH );
add( new JScrollPane( resultTable ), BorderLayout.CENTER );

// create event listener for submitButton
submitButton.addActionListener(

  new ActionListener() {
    // pass query to table model
    public void actionPerformed( ActionEvent event ) {
      // perform a new query
```

```
            try {
               tableModel.setQuery( queryArea.getText() );
            } // end try
            catch ( SQLException sqlException ) {
               JOptionPane.showMessageDialog( null,
               sqlException.getMessage(), "Database error",
               JOptionPane.ERROR_MESSAGE );
               // try to recover from invalid user query by executing default query
               try {
                  tableModel.setQuery( DEFAULT_QUERY );
                  queryArea.setText( DEFAULT_QUERY );
               } // end try
               catch ( SQLException sqlException2 ) {
                  JOptionPane.showMessageDialog( null,
                  sqlException2.getMessage(), "Database error",
                  JOptionPane.ERROR_MESSAGE );
                  // ensure database connection is closed
                  tableModel.disconnectFromDatabase();
                  System.exit( 1 ); // terminate application
               } // end inner catch
            } // end outer catch
         } // end actionPerformed
      } // end ActionListener inner class
   ); // end call to addActionListener
```

```
   setSize( 500, 250 ); // set window size
   setVisible( true ); // display window
} // end try
catch ( ClassNotFoundException classNotFound ) {
      JOptionPane.showMessageDialog( null,
      "MySQL driver not found", "Driver not found",
      JOptionPane.ERROR_MESSAGE );

   System.exit( 1 ); // terminate application
} // end catch
catch ( SQLException sqlException ) {
      JOptionPane.showMessageDialog( null, sqlException.getMessage(),
      "Database error", JOptionPane.ERROR_MESSAGE );
   // ensure database connection is closed
   tableModel.disconnectFromDatabase();
  System.exit( 1 );   // terminate application
} // end catch
 // dispose of window when user quits application (this overrides
// the default of HIDE_ON_CLOSE)
setDefaultCloseOperation( DISPOSE_ON_CLOSE );
 // ensure database connection is closed when user quits application
addWindowListener(
```

```
      new WindowAdapter()
      {
        // disconnect from database and exit when window has closed
        public void windowClosed( WindowEvent event )
        {
          tableModel.disconnectFromDatabase();
          System.exit( 0 );
        } // end method windowClosed
      } // end WindowAdapter inner class
    ); // end call to addWindowListener
  } // end DisplayQueryResults constructor

  // execute application
  public static void main( String args[] )
  {
    new DisplayQueryResults();
  } // end main
} // end class DisplayQueryResults
```

bikename	size	color	cost	purchased	mileage
Battaglin Carrera	60	red/white	4000	Mar 10, 2001	11200
Bianchi Corse Evo 4	58	celeste	5700	Dec 2, 2004	300
Bianchi Evolution 3	58	celeste	4800	Nov 12, 2003	2000
Bianchi Infinito	58	celeste	8900	Jul 14, 2011	0
BMC SLC01 - Swiss	58	red/black/white	8000	Jun 23, 2010	0
Colnago Dream Rabobank	60	blue/orange	5500	Jul 7, 2002	4300
Colnago Superissimo	59	red	3800	Mar 1, 1996	13000
Eddy Merckx Domo	58	blue/black	5300	Feb 2, 2004	0
Eddy Merckx Molteni	58	orange	5100	Aug 12, 2004	0
Gianni Motta Personal	59	red/green	4400	May 1, 2000	8700
Gios Torino Super	60	blue	2000	Nov 8, 1998	9000
Ridley Damocles	58	blue/black	7500	Jun 27, 2008	0
Ridley X-Fire 2012	58	red/white	7500	Sep 1, 2011	0
Schwinn Paramount P14	60	blue	1800	Mar 1, 1992	200



bikename	size	color	cost	purchased	mileage
Bianchi Corse Evo 4	58	celeste	5700	Dec 2, 2004	300
Bianchi Infinito	58	celeste	8900	Jul 14, 2011	0
BMC SLC01 - Swiss	58	red/black/white	8000	Jun 23, 2010	0
Colnago Dream R...	60	blue/orange	5500	Jul 7, 2002	4300
Eddy Merckx Domo	58	blue/black	5300	Feb 2, 2004	0
Eddy Merckx Molteni	58	orange	5100	Aug 12, 2004	0
Ridley Damocles	58	blue/black	7500	Jun 27, 2008	0
Ridley X-Fire 2012	58	red/white	7500	Sep 1, 2011	0

Exemples
(site www.coreservlets.com)

Sample Database

- Table name
  - employees
- Column names
  - id (int). The employee ID.
  - firstname (varchar/String). Employee's given name.
  - lastname (varchar/String). Employee's family name.
  - position (varchar/String). Corporate position (eg, "ceo").
  - salary (int). Yearly base salary.
- Database name
  - myDatabase

- Note
  - See "Prepared Statements" section for code that created DB

Example: Printing Employee Info

The URL and the driver are the only parts that are specific to Derby. So, if you switch to MySql, Oracle, etc., you have to change those two lines (or just one line in Java 6 with JDBC 4 driver, since the driver no longer needs to be declared in that situation). The rest of the code is database independent.

```
package coreservlets;
import java.sql.*;
import java.util.*;
public class ShowEmployees {
  public static void main(String[] args) {
    String url = "jdbc:derby:myDatabase";
    Properties userInfo = new Properties();
    userInfo.put("user", "someuser");
    userInfo.put("password", "somepassword");
    String driver =
     "org.apache.derby.jdbc.EmbeddedDriver";
    showSalaries(url, userInfo, driver);
  }
```

Example: Printing Employee Info (Connecting to Database)

```
  public static void showSalaries(String url,
                                  Properties userInfo,
                                  String driverClass) {
Class.forName(driverClass);
 try {
Connection connection =
      DriverManager.getConnection(url, userInfo);
     System.out.println("Employees\n==========");
     // Create a statement for executing queries.
     Statement statement = connection.createStatement();
     String query =
       "SELECT * FROM employees ORDER BY salary";
     // Send query to database and store results.
     ResultSet resultSet = statement.executeQuery(query);
```

Example: Printing Employee Info (Processing Results)

```
while(resultSet.next()) {
      int id = resultSet.getInt("id");
      String firstName = resultSet.getString("firstname");
      String lastName = resultSet.getString("lastname");
      String position = resultSet.getString("position");
      int salary = resultSet.getInt("salary");
      System.out.printf
          ("%s %s (%s, id=%s) earns $%,d per year.%n",
           firstName, lastName, position, id, salary);
    }
    connection.close();
  } catch(Exception e) {
    System.err.println("Error with connection: " + e);
  }
```

Example: Printing Employee Info (Output)

```
Employees
==========
Gary Grunt (Gofer, id=12) earns $7,777 per year.
Gabby Grunt (Gofer, id=13) earns $8,888 per year.
Cathy Coder (Peon, id=11) earns $18,944 per year.
Cody Coder (Peon, id=10) earns $19,842 per year.
Danielle Developer (Peon, id=9) earns $21,333 per year.
David Developer (Peon, id=8) earns $21,555 per year.
Joe Hacker (Peon, id=6) earns $23,456 per year.
Jane Hacker (Peon, id=7) earns $32,654 per year.
Keith Block (VP, id=4) earns $1,234,567 per year.
Thomas Kurian (VP, id=5) earns $2,431,765 per year.
Charles Phillips (President, id=2) earns $23,456,789 per year.
Safra Catz (President, id=3) earns $32,654,987 per year.
Larry Ellison (CEO, id=1) earns $1,234,567,890 per year.
```

Using JDBC from Web Apps

Employee Info Servlet

```
public class EmployeeServlet1 extends
   HttpServlet {
  //private final String driver =
  //   "org.apache.derby.jdbc.EmbeddedDriver";
  protected final String url =
   "jdbc:derby:myDatabase";
  protected final String tableName =
   "employees";
  protected final String username = "someuser";
  protected final String password =
   "somepassword";
```

The URL and the driver are the only parts that are specific to Derby. So, if you switch to MySql, Oracle, etc., you have to change those two lines (or just one line in Java 6 with JDBC 4 driver, since the driver no longer needs to be declared in that situation). The rest of the code is database independent.

Employee Info Servlet (Continued)

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException {
  response.setContentType("text/html");
  PrintWriter out = response.getWriter();
  String docType =
    "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
    "Transitional//EN\"\n";
  String title = "Company Employees";
  out.print(docType +
      "<HTML>\n" +
      "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
      "<LINK REL='STYLESHEET' HREF='./css/styles.css'\n" +
      "      TYPE='text/css'>" +
      "<BODY><CENTER>\n" +
      "<TABLE CLASS='TITLE' BORDER='5'>" +
      "  <TR><TH>" + title + "</TABLE><P>");
  showTable(out);
  out.println("</CENTER></BODY></HTML>");      }
```

Employee Info Servlet (Continued)

```
protected void showTable(PrintWriter out) {
   try {
     Connection connection = getConnection();
     Statement statement = connection.createStatement();
     String query = "SELECT * FROM " + tableName;
     ResultSet resultSet = statement.executeQuery(query);
     printTableTop(connection, resultSet, out);
     printTableBody(resultSet, out);
     connection.close();
   } catch(Exception e) {
     System.err.println("Error: " + e);
   }
 }
```

Employee Info Servlet (Continued)

```
protected Connection getConnection()
    throws Exception {
  Class.forName(driver);

  // Establish network connection to database.
  Properties userInfo = new Properties();
  userInfo.put("user", username);
  userInfo.put("password", password);
  Connection connection =
    DriverManager.getConnection(url, userInfo);
  return(connection);
}
```

Employee Info Servlet (Continued)

```
protected void printTableTop(Connection connection,
                             ResultSet resultSet,
                             PrintWriter out)
    throws SQLException {
  out.println("<TABLE BORDER='1'>");
  // Print headings from explicit heading names
  String[] headingNames =
    { "ID", "First Name", "Last Name",
      "Position", "Salary" };
  out.print("<TR>");
  for (String headingName : headingNames) {
    out.printf("<TH>%s", headingName);
  }
  out.println();
}
```

Employee Info Servlet (Continued)

```
 protected void printTableBody(ResultSet resultSet,
                               PrintWriter out)
      throws SQLException {
while(resultSet.next()) {
      out.println("<TR ALIGN='RIGHT'>");
      out.printf("  <TD>%d", resultSet.getInt("id"));
      out.printf("  <TD>%s", resultSet.getString("firstname"));
      out.printf("  <TD>%s", resultSet.getString("lastname"));
      out.printf("  <TD>%s", resultSet.getString("position"));
      out.printf("  <TD>$%,d%n", resultSet.getInt("salary"));
    }
    out.println("</TABLE>");
  }
}
```

Employee Info Servlet (Results)

ID	First Name	Last Name	Position	Salary
1	Larry	Ellison	CEO	$1,234,567,890
2	Charles	Phillips	President	$23,456,789
3	Safra	Catz	President	$32,654,987
4	Keith	Block	VP	$1,234,567
5	Thomas	Kurian	VP	$2,431,765
6	Joe	Hacker	Peon	$23,456
7	Jane	Hacker	Peon	$32,654
8	David	Developer	Peon	$21,555
9	Danielle	Developer	Peon	$21,333
10	Cody	Coder	Peon	$19,842
11	Cathy	Coder	Peon	$18,944
12	Gary	Grunt	Gofer	$7,777
13	Gabby	Grunt	Gofer	$8,888

Using MetaData

Using MetaData: Example

```
public class EmployeeServlet2 extends EmployeeServlet1 {
  protected void printTableTop(Connection connection,
                               ResultSet resultSet,
                               PrintWriter out)
      throws SQLException {
    // Look up info about the database as a whole.
    DatabaseMetaData dbMetaData = connection.getMetaData();
    String productName =
      dbMetaData.getDatabaseProductName();
    String productVersion =
      dbMetaData.getDatabaseProductVersion();
    out.println("<UL>\n" +
                "  <LI><B>Database:</B>\n" + productName +
                "  <LI><B>Version:</B>\n" + productVersion +
                "</UL>");
```

Using MetaData: Example (Continued)

```
    out.println("<TABLE BORDER='1'>");
    // Discover and print headings
    ResultSetMetaData resultSetMetaData =
      resultSet.getMetaData();
    int columnCount = resultSetMetaData.getColumnCount();
    out.println("<TR>");
    // Column index starts at 1 (a la SQL), not 0 (a la
  Java).
    for(int i=1; i <= columnCount; i++) {
      out.printf("<TH>%s",
  resultSetMetaData.getColumnName(i));
    }
    out.println();
  }
}
```

Using MetaData: Results

ID	FIRSTNAME	LASTNAME	POSITION	SALARY
1	Larry	Ellison	CEO	$1,234,567,890
2	Charles	Phillips	President	$23,456,789
3	Safra	Catz	President	$32,654,987
4	Keith	Block	VP	$1,234,567
5	Thomas	Kurian	VP	$2,431,765
6	Joe	Hacker	Peon	$23,456
7	Jane	Hacker	Peon	$32,654
8	David	Developer	Peon	$21,555
9	Danielle	Developer	Peon	$21,333
10	Cody	Coder	Peon	$19,842
11	Cathy	Coder	Peon	$18,944
12	Gary	Grunt	Gofer	$7,777
13	Gabby	Grunt	Gofer	$8,888

Using Prepared Statements (Parameterized Commands)

Creating Sample Database

```
public class EmbeddedDbCreator {
  // Driver class not needed in JDBC 4.0 (Java SE 6)
  // private String driver =
  //   "org.apache.derby.jdbc.EmbeddedDriver";
  private String protocol = "jdbc:derby:";
  private String username = "someuser";
  private String password = "somepassword";
  private String dbName = "myDatabase";
  private String tableName = "employees";
  private Properties userInfo;
    public EmbeddedDbCreator() {
    userInfo = new Properties();
    userInfo.put("user", username);
    userInfo.put("password", password);
  }
```

Creating Sample Database (Continued)

```
public void createDatabase() {
  Employee[] employees = {
    new Employee(1, "Larry", "Ellison", "CEO",
                 1234567890),
    new Employee(2, "Charles", "Phillips",
 "President",
                 23456789),
    new Employee(3, "Safra", "Catz",
 "President",
                 32654987),
    …
  };
```

Creating Sample Database (Continued)

```
try {
     String dbUrl = protocol + dbName + ";create=true";
     Connection connection =
       DriverManager.getConnection(dbUrl, userInfo);
     Statement statement = connection.createStatement();
     String format = "VARCHAR(20)";
     String tableDescription =
       String.format
         ("CREATE TABLE %s" +
             "(id INT, firstname %s, lastname %s, " +
                 "position %s, salary INT)",
          tableName, format, format, format);
     statement.execute(tableDescription);
```

Creating Sample Database (Continued)

```
  String template =
  String.format("INSERT INTO %s VALUES(?, ?, ?, ?, ?)",
                tableName);
PreparedStatement inserter =
  connection.prepareStatement(template);
for(Employee e: employees) {
  inserter.setInt(1, e.getEmployeeID());
  inserter.setString(2, e.getFirstName());
  inserter.setString(3, e.getLastName());
  inserter.setString(4, e.getPosition());
  inserter.setInt(5, e.getSalary());
  inserter.executeUpdate();
  System.out.printf("Inserted %s %s.%n",
                    e.getFirstName(),
                    e.getLastName());
}
inserter.close();
connection.close();
```

Triggering Database Creation: Listener

```
package coreservlets;

import javax.servlet.*;


public class DatabaseInitializer
       implements ServletContextListener {
  public void contextInitialized(ServletContextEvent
   event) {
    new EmbeddedDbCreator().createDatabase();
  }


  public void contextDestroyed(ServletContextEvent
   event) {}
}
```

Triggering Database Creation: web.xml

```
...
  <listener>
    <listener-class>
      coreservlets.DatabaseInitializer
    </listener-class>
  </listener>
```

Transactions: Example

```
Connection connection =
  DriverManager.getConnection(url, userProperties);
connection.setAutoCommit(false);
try {
  statement.executeUpdate(...);
  statement.executeUpdate(...);
  ...
  connection.commit();
} catch (Exception e) {
  try {
    connection.rollback();
  } catch (SQLException sqle) {
     // report problem
  }
} finally {
  try {
   connection.close();
  } catch (SQLException sqle) { }
}
```

Conclusions

- **Conclusions sur l ’API JDBC :**
  - jeu unique d ’interfaces pour un accès homogène
    - cache au maximum les diverses syntaxes SQL des SGBD
  - le principe des *drivers* permet au développeur d’ignorer les détails techniques liés aux différents moyens d ’accès aux BDs
    - une convention de nommage basée sur les URL est utilisée pour localiser le bon pilote et lui passer des informations
  - Tous les grands éditeurs de bases de données et les sociétés spécialisées proposent un *driver* JDBC pour leurs produits
  - Le succès de JDBC se voit par le nombre croissant d ’outils de développement graphiques permettant le développement RAD d ’applications client-serveur en Java