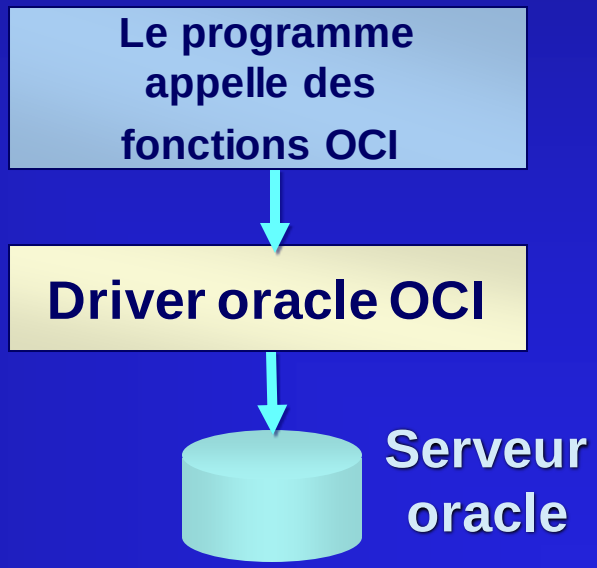
Présentation de l'API JDBC

Pierre Lefebvre

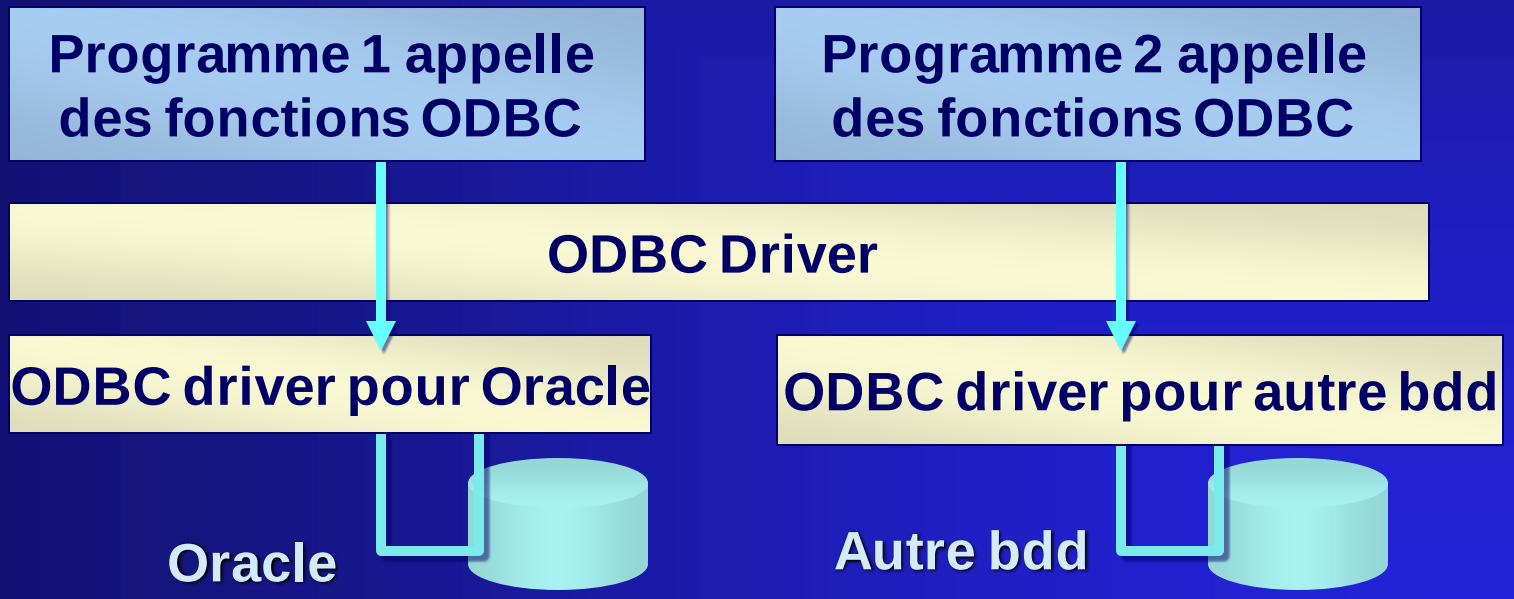
Accéder aux bases de données en dehors de java

- **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)



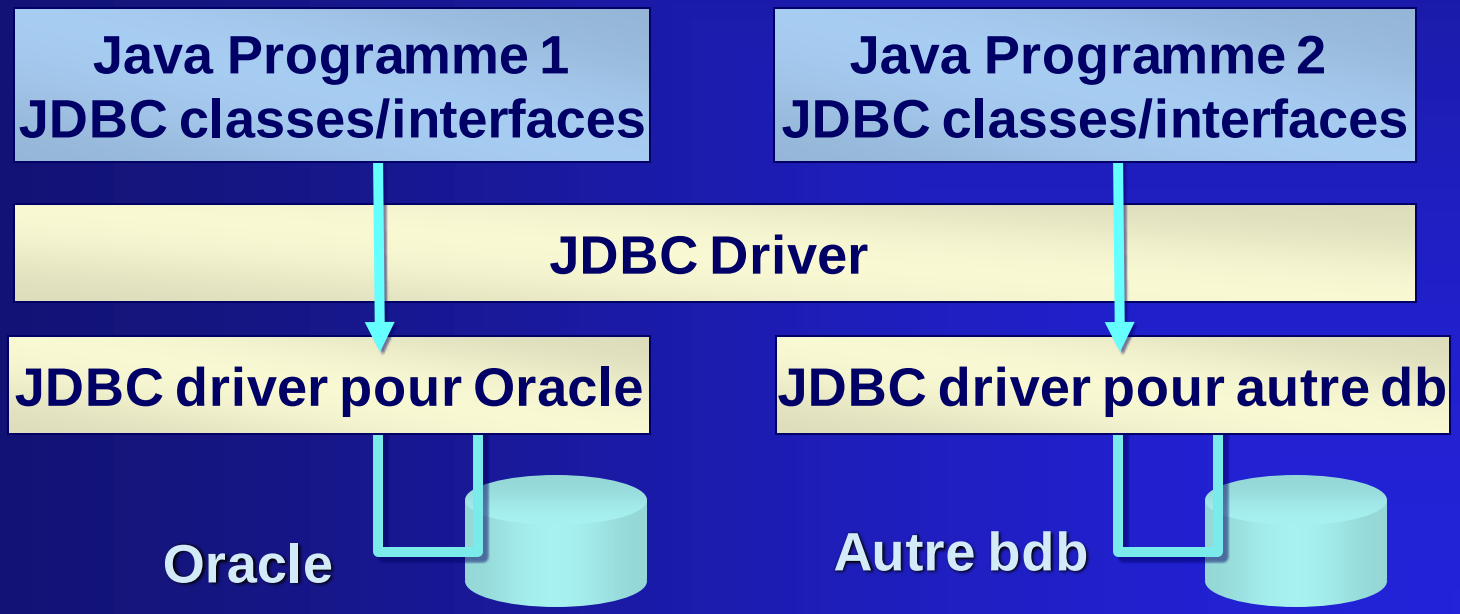
ODBC: une CLI standard

- 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{…}
```

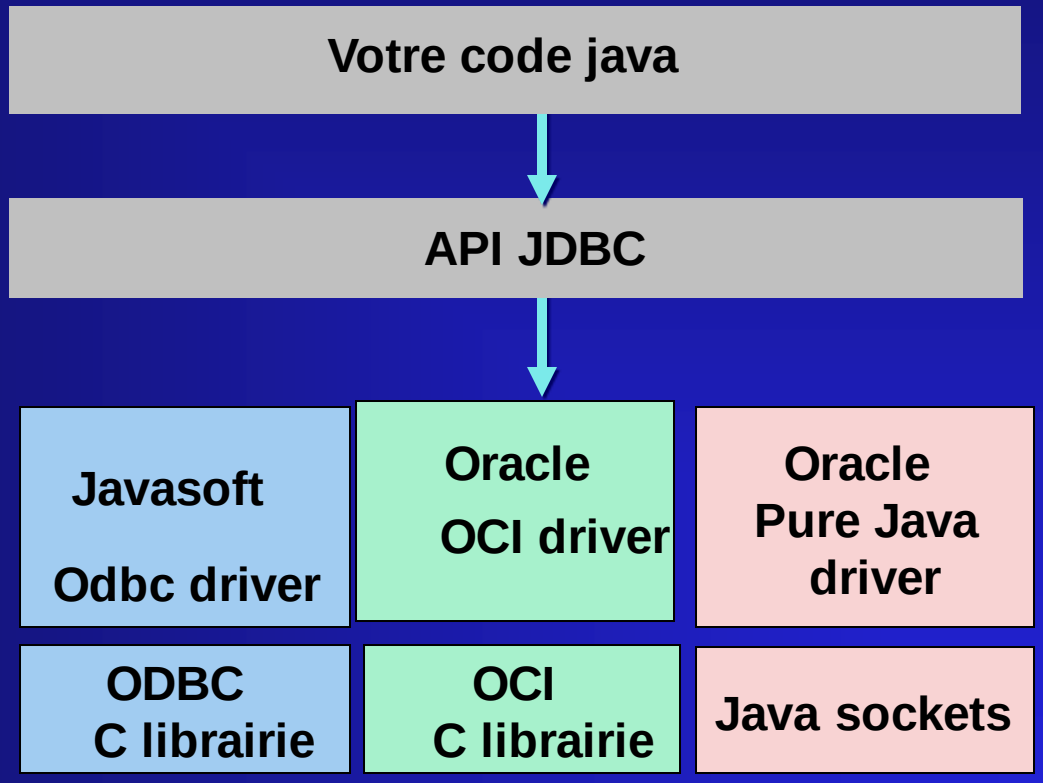
Driver JDBC, comme Oracle

```
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`

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

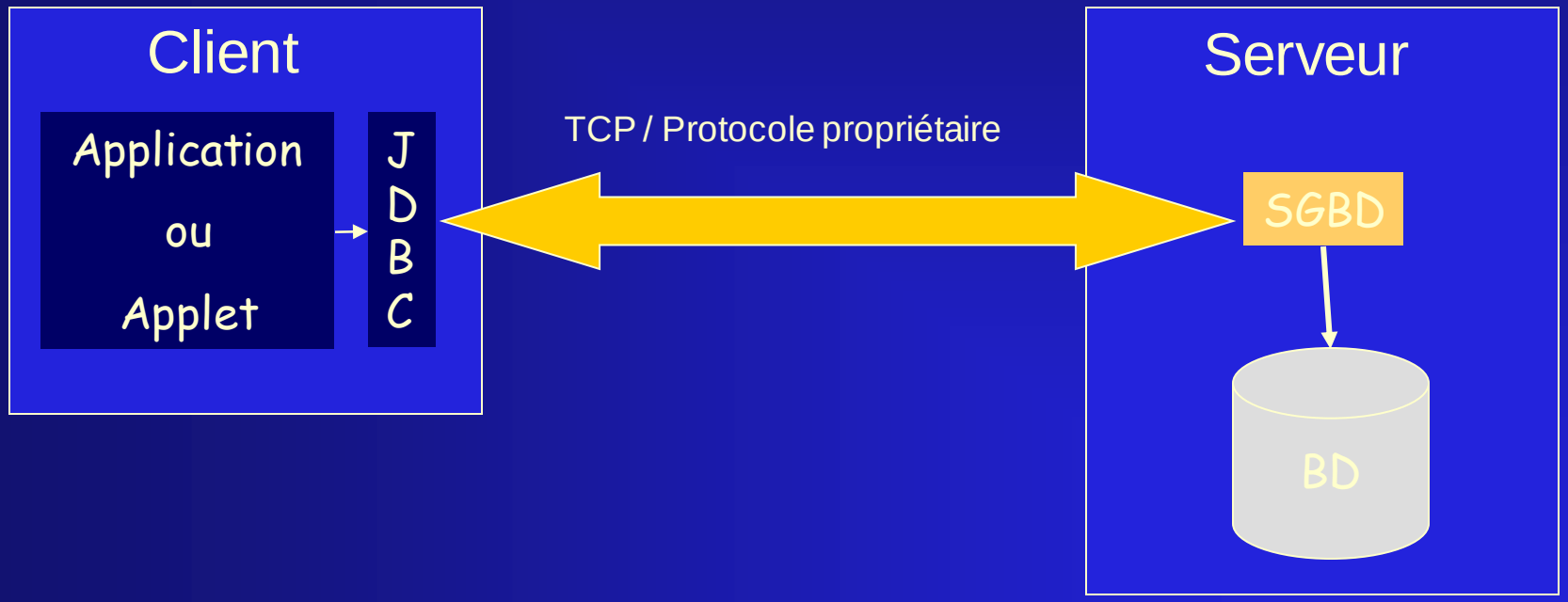
Driver de type 3

- **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

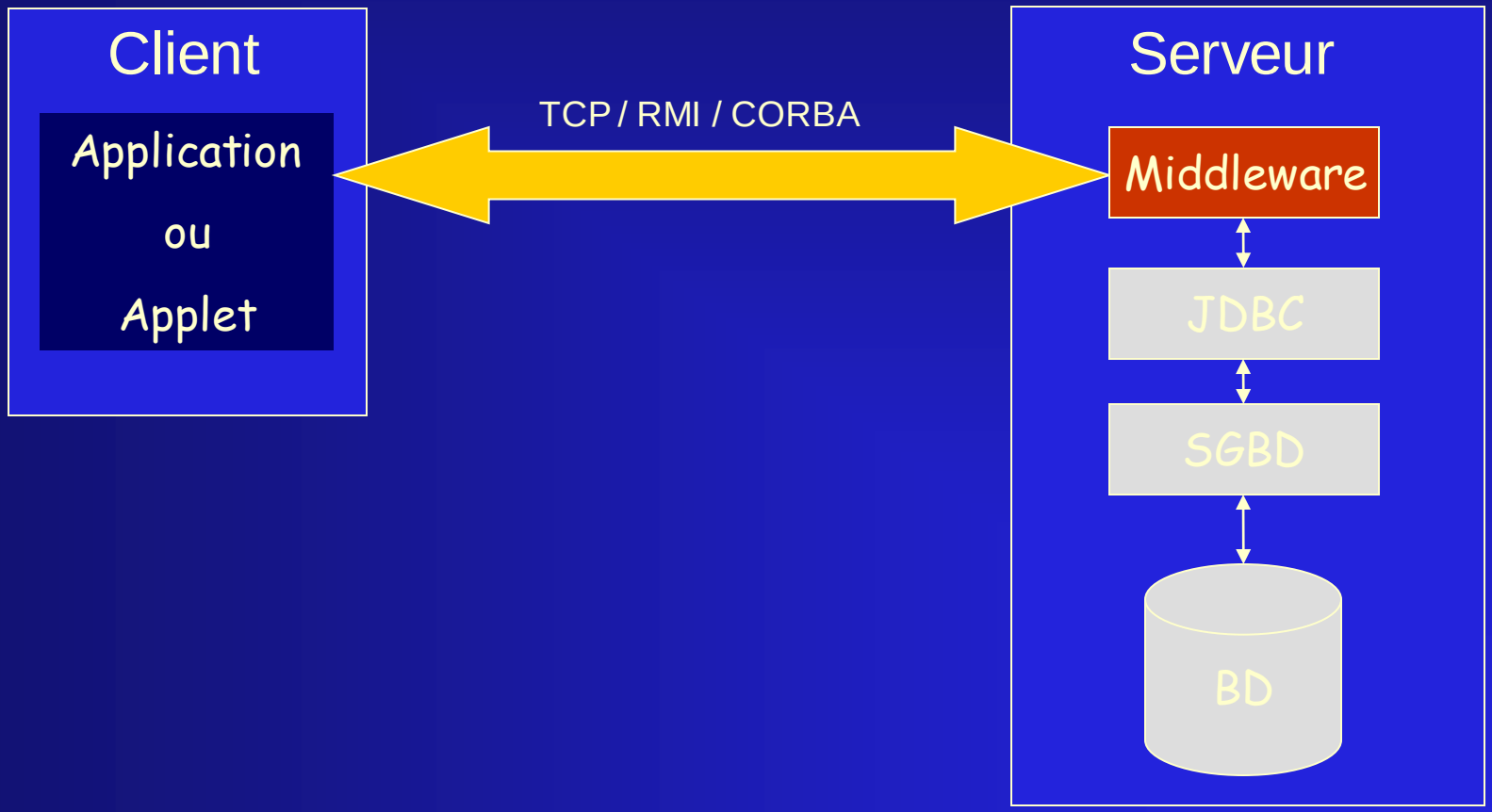
Driver de type 4

- **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

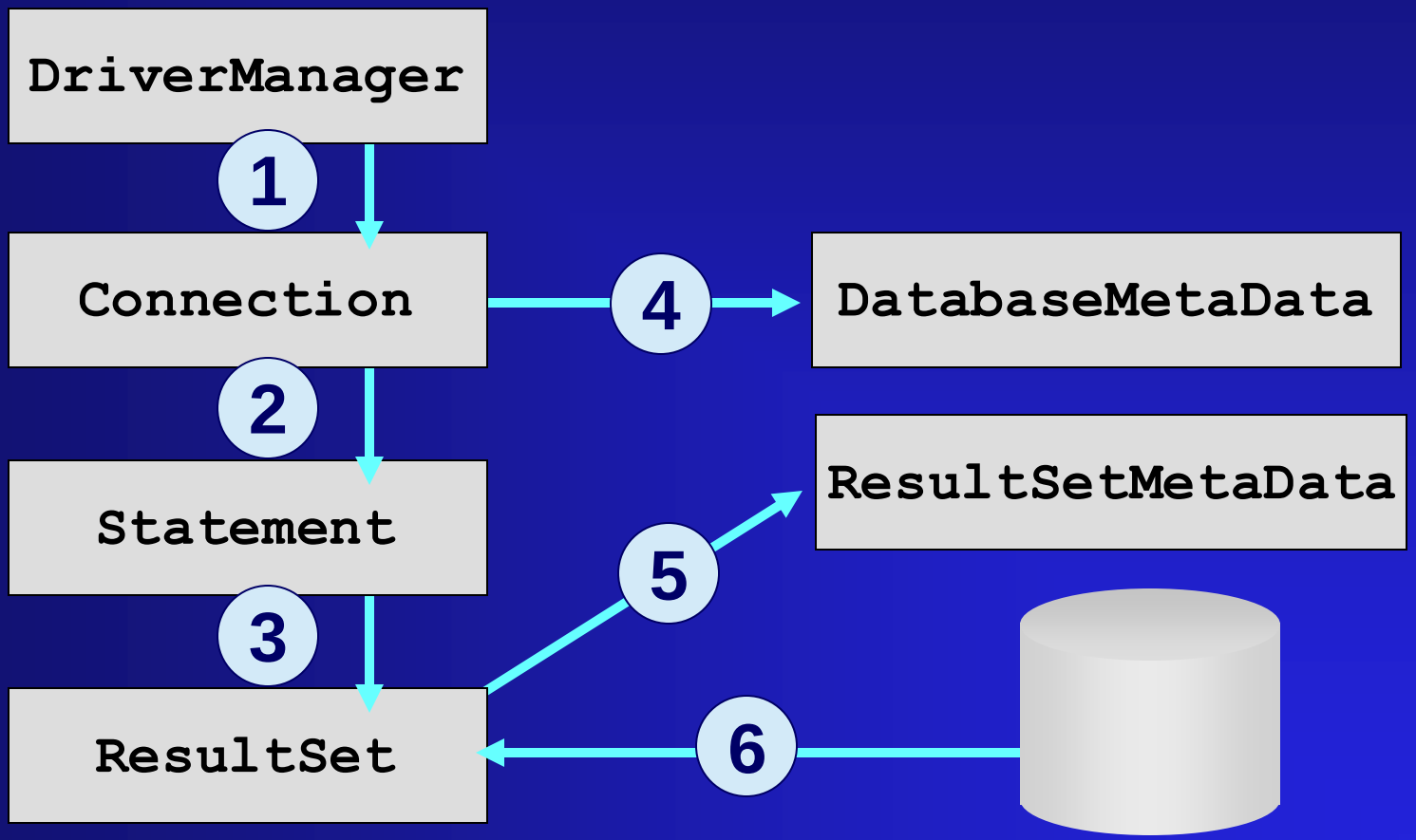
Architecture 2-tiers



Architecture 3-tiers



Relation entre classes JDBC



Mise en 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

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
  - `jdbc:<sous-protocôle>:<sous-nom>`
- **`Exemples`**
  - **Oracle thin driver**
    - oracle:jdbc:oracle:thin:@machinename:1521:dbname
  - **Derby**
    - jdbc:derby ://localhost:1527/sample
  - **Pointbase**
    - jdbc:pointbase :server://localhost/sample

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) {…}
```

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);
```

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

Exemple

- `executeQuery()` exécute une requête SQL, et ramène un `ResultSet`

```
try {
  Statement stmt = conn.createStatement();
  ResultSet rset = stmt.executeQuery
                   ("select ENAME, SAL from EMP");
}
catch (SQLException e) {…}
```

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}
```

      - impossible de revenir au tuple précédent ou de parcourir l ’ensemble dans un ordre aléatoire

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");
}
```

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
….	…..

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()`

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) {…}
```

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) {…}
```

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) {…}
```

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
```

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

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) { }
}
```