JDBC

(2e partie)

Bibliographies: Pour cette deuxième partie de ce cours, une revue littéraire a été réalisée à partir des transparents se trouvant la plus part dans les deux documents :

http://deptinfo.unice.fr/~grin/messupports/trjdbcA.pdf
http://www-sop.inria.fr/acacia/personnel/itey/Francais/Cours/cours-bd-fra.html

Le reste est venu des liens et livres suivants :

Site web de SUN sur les JDBC: http://java.sun.com/products/jdbc
Les livres :
«Database Programming with JDBC and Java» de George Reese, O'Reilly. 2nd édition.
«Core Java 2» de C. S. Horstmann, G. Cornell ; volume 2.

Pourquoi JDBC

Pendant des années, les programmeurs codés des outils d'accès à des bases de données en utilisant des outils fournis par les distributeurs (vendeurs, qui sont parfois les concepteurs) de ces bases.

Ces outils prenaient la forme de APIs codées en C/C++, ayant des spécifications propres à elles.

Les distributeurs de bdd fournissaient parfois des préprocesseurs où le programmeur devait inclure ses requêtes à la base de données dans la logique du concepteur de ces préprocesseurs.

Le but recherché …

Permettre aux programmeurs d'écrire un code indépendant de la base de données et du moyen de connectivité utilisé.

Vu les difficultés rencontrées avec d'autres langages à cause des API trop spécifiques à la base de données. Pourquoi pas Java?

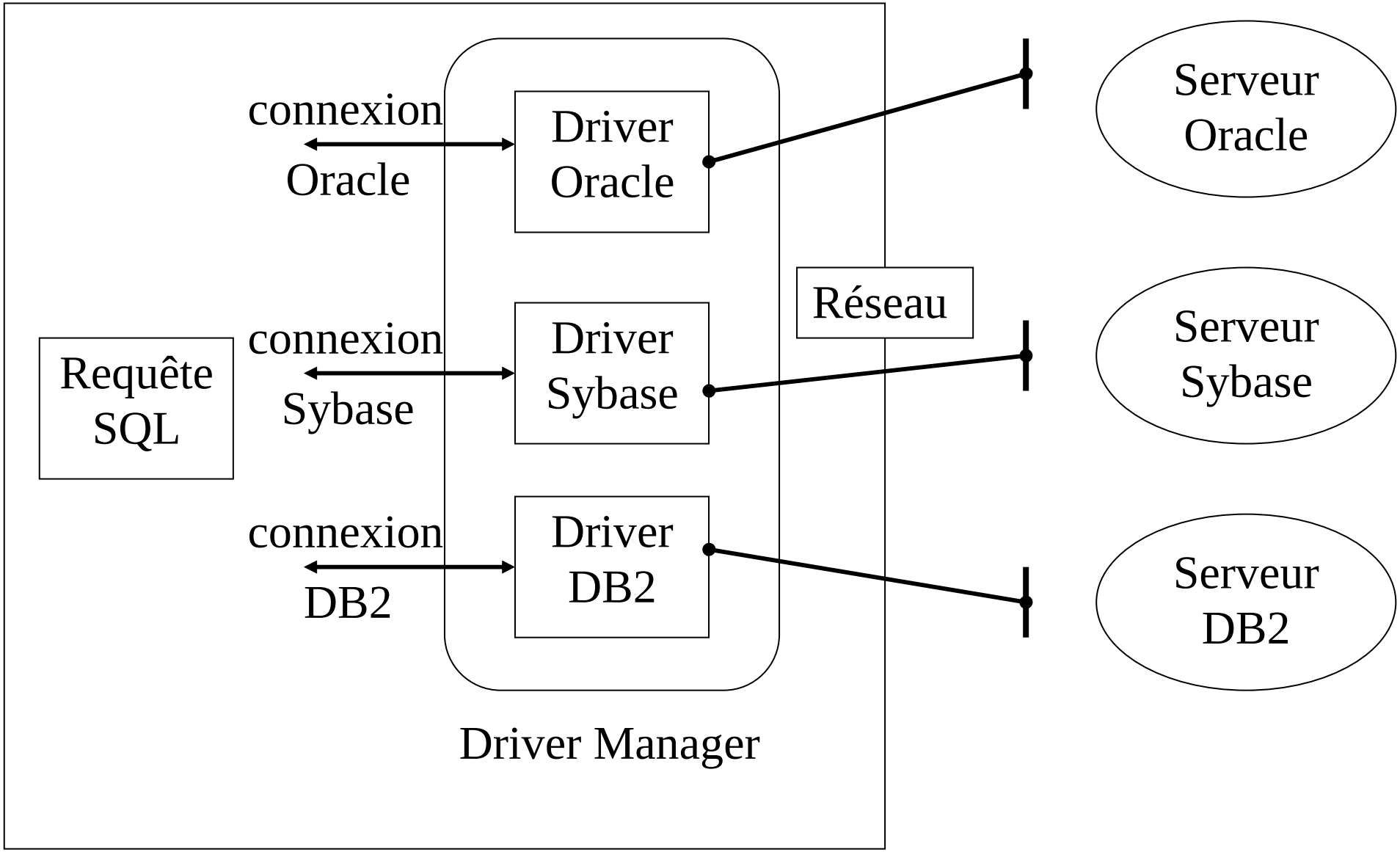
- robuste et sécurisé,
- facile à comprendre,
- automatiquement téléchargeable par le réseau,
- permettant diverses opérations via le net etc.

Java oui, mais est-ce possible ?

Qu’avons nous besoin réellement ?

- Faire des requêtes SQL. SQL étant le langage permettant de dialoguer avec un SGBDR.

- Prendre cette requête et la transférer telle quelle indépendamment du système et du SGBDR?


connexion
Oracle
Driver
Oracle
Serveur
Oracle
Réseau
connexion
Sybase
Driver
Sybase
Serveur
Sybase
Requête
SQL
connexion
DB2
Driver
DB2
Serveur
DB2
Driver Manager

Le driver et son rôle …

Chaque base de données utilise un pilote (*driver*) qui lui est propre et qui permet de convertir les requêtes dans le langage natif du SGBDR.

Les drivers dépendent du SGBD auquel ils permettent d'accéder

Tous les SGBD importants du marché ont un (et même plusieurs) driver, fourni par l'éditeur du SGBD ou par des éditeurs de logiciels indépendants.

Et la Java la dedans …

JDBC (Java Database Connectivity), est un ensemble de classes Java qui permet de se connecter à une base de données distante sur le réseau, et d'interroger cette base afin d'en extraire des données.

JDBC est fourni par le paquetage java.sql

Ce paquetage permet de formuler et gérer les requêtes aux bases de données relationnelles

Ce paquetage contient un grand nombre d'interfaces et quelques classes.

8 interfaces définissent les objets nécessaires :

— à la connexion à une base éloignée
— et aux création et exécution de requêtes SQL

Les interfaces imposent l'API pour travailler avec JDBC.

JDBC ne fournit pas les classes qui implantent les interfaces

Pour travailler avec un SGBD ; il faut disposer de classes qui implantent les interfaces de JDBC.

Un ensemble de telles classes est désigné sous le nom de *driver* (la classe qui implante l'interface **Driver** de JDBC y joue un rôle important).

Les avantages de JDBC

- JDBC offre une intégration très étroite du client et des modules chargés de l'accès à la base. Cela permet de limiter le trafic réseau.

- JDBC est complètement indépendant de tout SGBD: la même application peut être utilisée pour accéder à une base ORACLE, SYBASE, MySQL, etc. Conséquences : pas besoin d'apprendre une nouvelle API quand on change de système, et réutilisation totale du code.

- Enfin, JDBC est relativement simple, beaucoup plus simple, par exemple, que l'interface C+SQL proposée par les SGBD relationnels.

Types de drivers

Type 1 : pont JDBC-ODBC

Type 2 : driver qui fait appel à des fonctions natives non Java (le plus souvent en langage C) de l'API du SGBD que l'on veut utiliser

Type 3 : driver qui permet l'utilisation d'un serveur middleware (application intermédiaire) d'accès à plusieurs types de sources de données

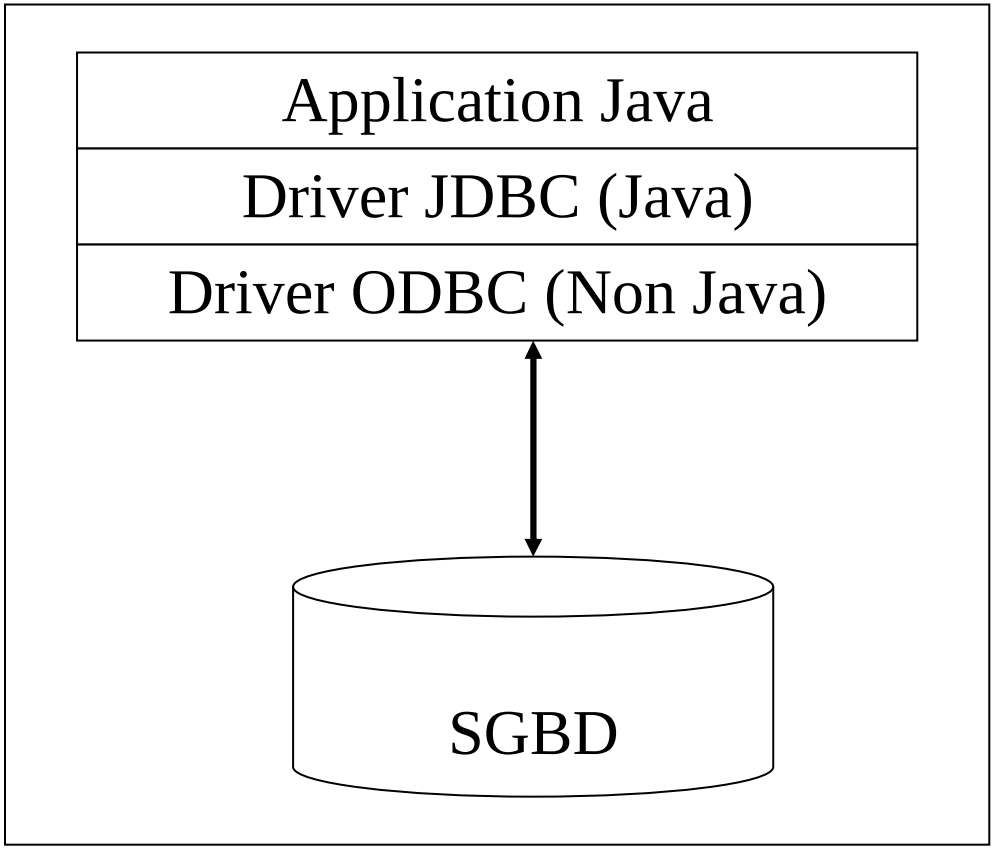
Type 4 : driver écrit entièrement en Java, qui utilise le protocole réseau du SGBD.

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
sun.jdbc.odbc.JdbcOdbcDriver

Open **D**ata**B**ase **C**onnectivity (ODBC)

permet d'accéder à la plupart des SGBD dans le monde Windows,

définit un format de communication standard entre les clients Windows et les serveurs de bases de données,

est devenu un standard de fait du monde Windows,

tous les constructeurs de SGBD fournissent un driver ODBC.

Avantages :

possibilité d'écrire des applications accédant à des données réparties entre plusieurs sources hétérogènes

— on développe l'application sans se soucier de la source de données

— la base de données utilisée côté serveur peut être interchangée sans aucune modification du développement fait dans la partie cliente.

Inconvénient : Ne convient pas au applet (sécurité).

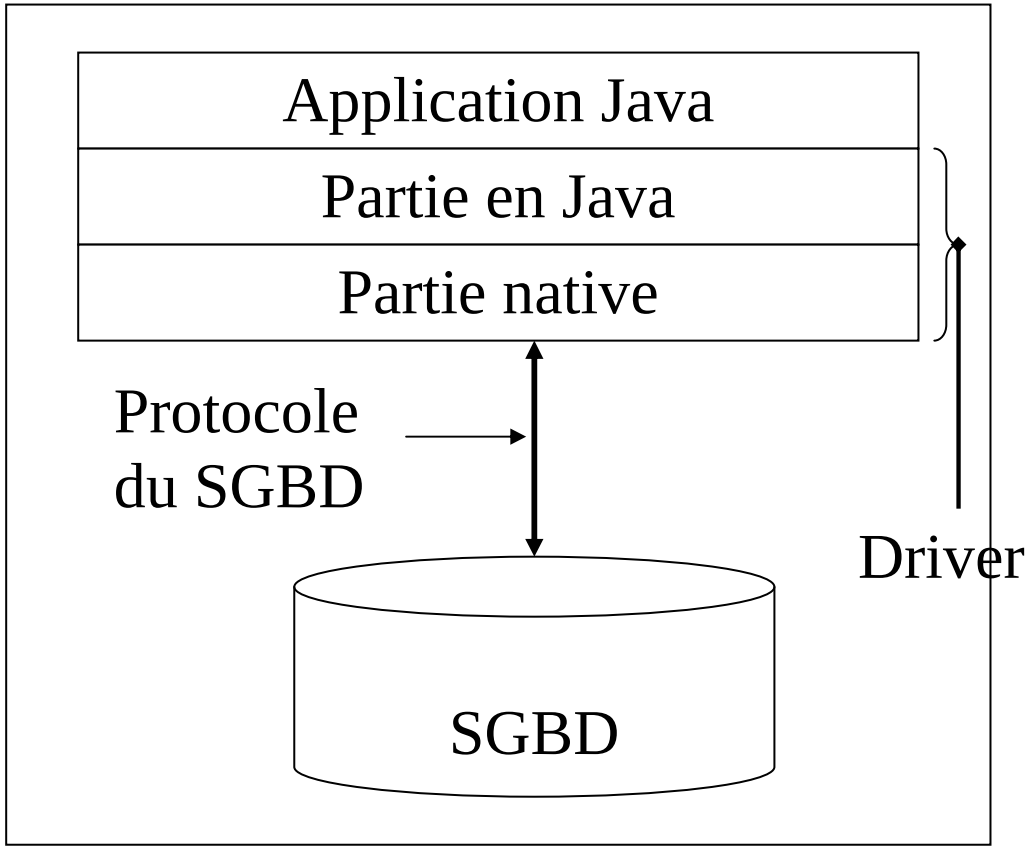
— interdiction de charger du code natif dans la mémoire vive de la plateforme.

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)

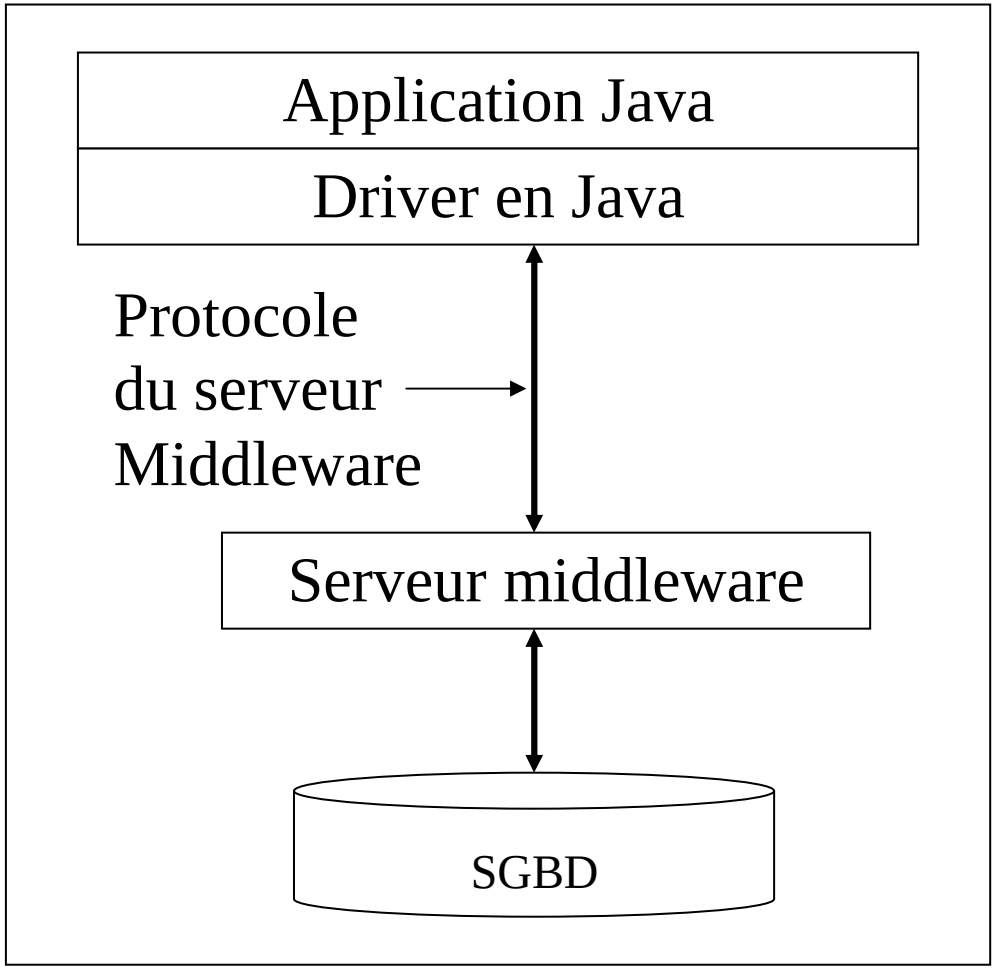
Type 3

Pilote « tout Java » ou « 100% Java »

interagit avec une API réseau générique (Sockets) et communique avec une application intermédiaire (*middleware*) sur le serveur,

le *middleware* accède par un moyen quelconque (par exple JDBC si écrit en Java) aux différents SGBDR

portable car entièrement écrit en Java
— pour *applets* et applications



Type 4

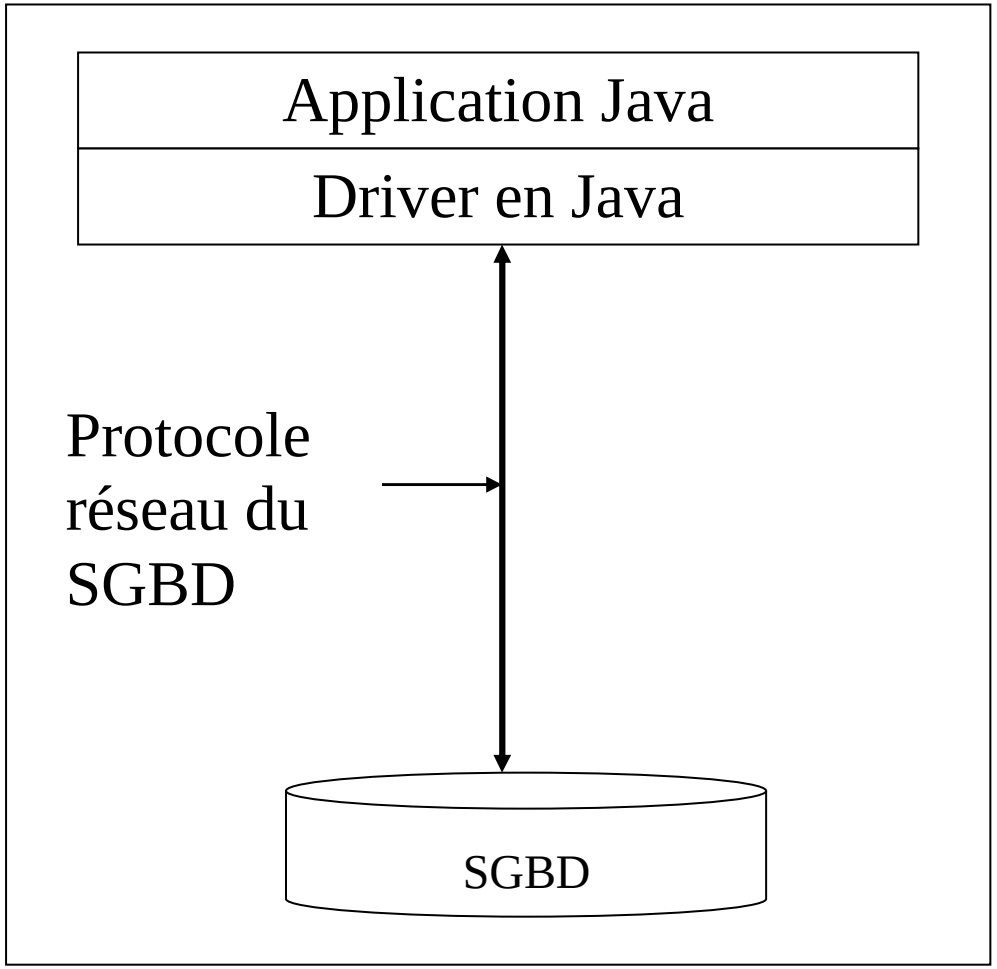
Driver « 100% Java » mais utilisant le protocole réseau du SGBDR

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 où elle est hébergée


Application Java
Driver en Java
Protocole
réseau du
SGBD
SGBD

Types de drivers et applet untrusted

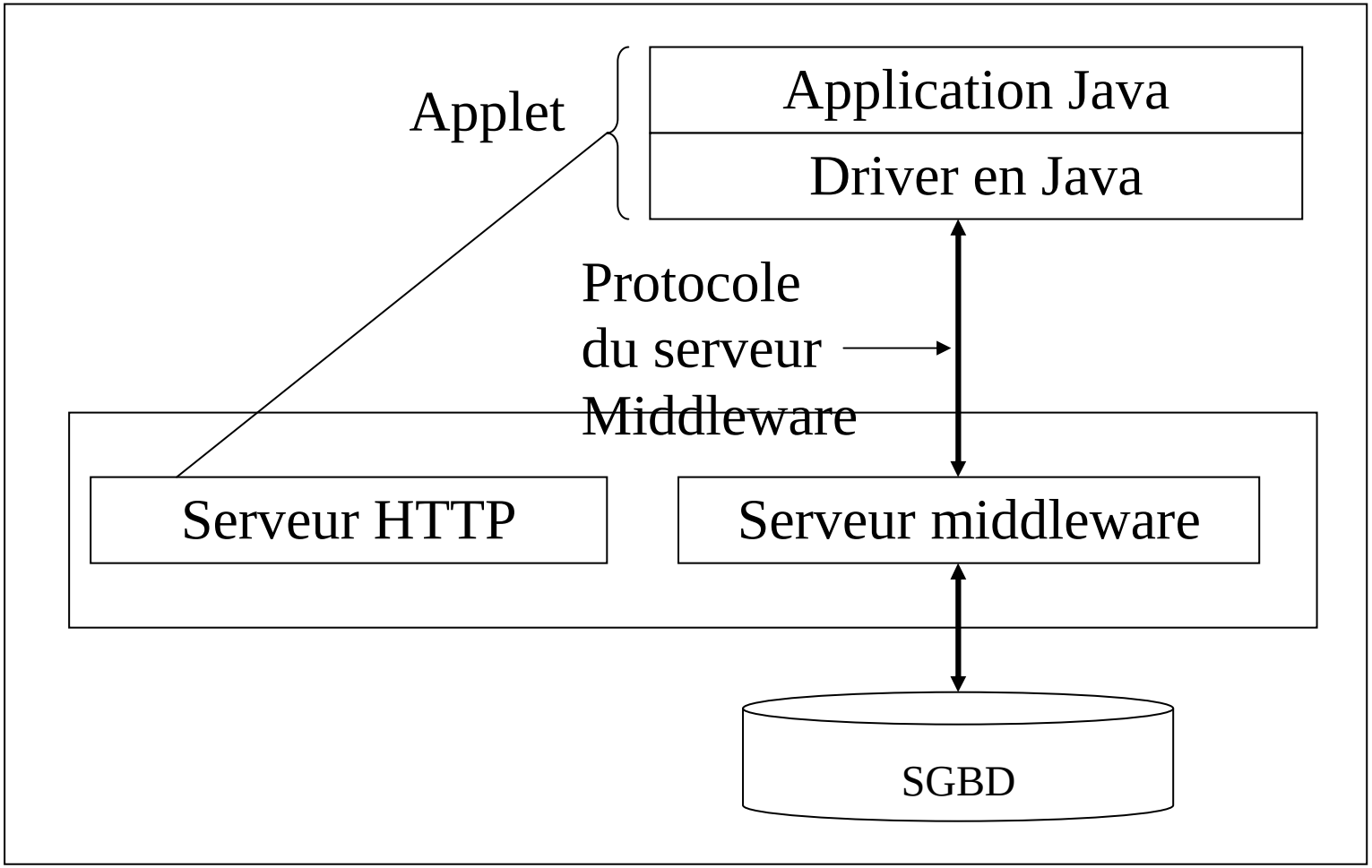
Une application Java peut travailler avec tous les types de drivers

Pour une *applet* (*untrusted*) :

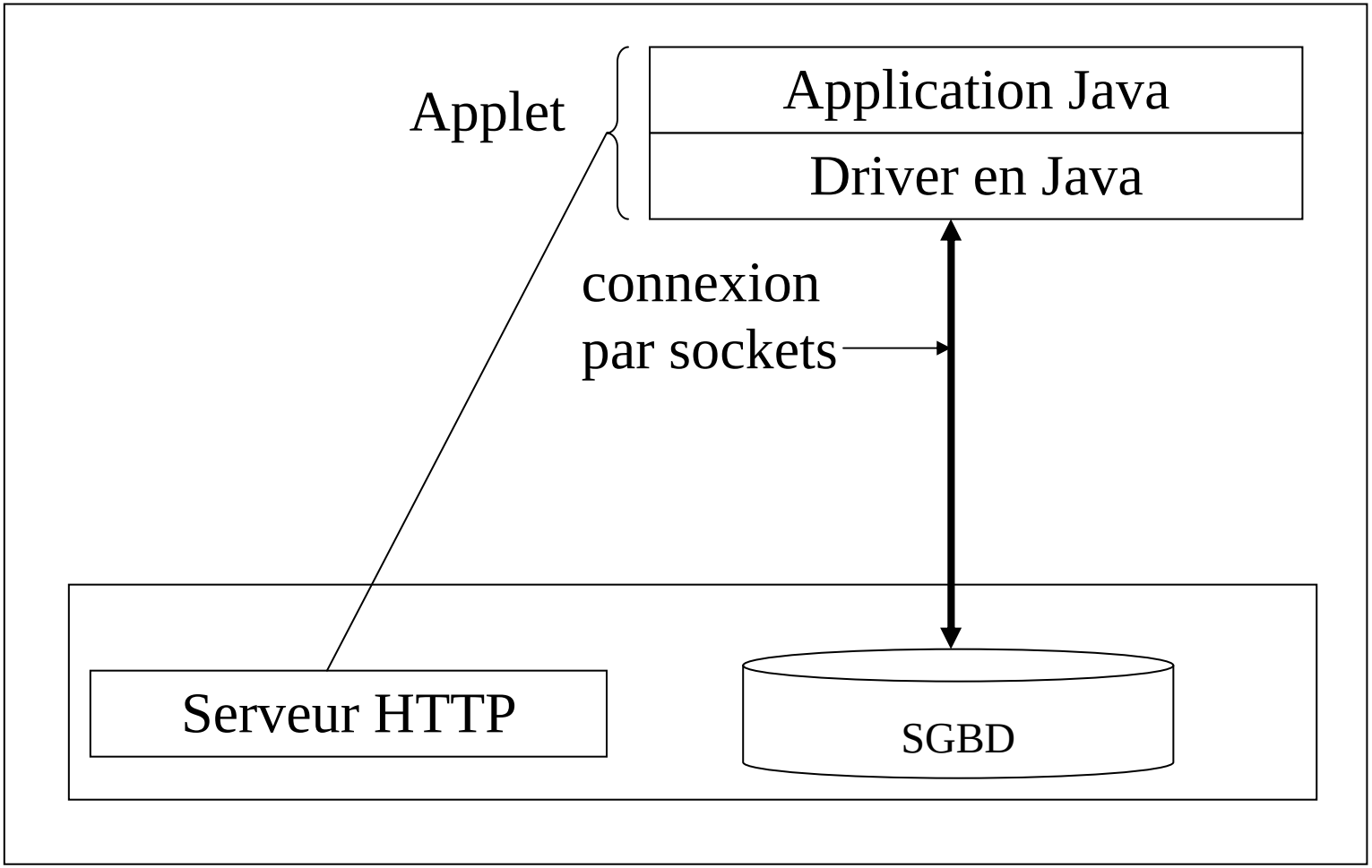
Une applet ne peut pas charger à distance du code natif (non Java) ; elle ne peut donc pas utiliser les drivers de type 1 et 2

Pour des raisons de sécurité, une applet *untrusted* ne peut échanger des données par sockets qu'avec la machine d'où elle provient, ce qui implique des contraintes avec les drivers de type 3 et 4

Avec le driver du type 3



Avec le driver du type 4



Utilisez les servlets …

Constitue une autre solution pour accéder à une base de données à travers le Web

Les *servlets* sont le pendant des *applets* côté serveur :

programmes Java travaillant directement avec le serveur Web

pas de contraintes de sécurité comme les *applets*

peuvent générer des pages HTML contenant les données récupérées grâce à JDBC (par exple)

La pratique des JDBC

0. Importer le package java.sql et configuration des paths
1. Enregistrer le driver JDBC
2. Établir la connexion à la base de données
3. Créer une zone de description de requête
4. Exécuter la requête
5. Traiter les données retournées
6. Fermer les différents espaces

Les classes et interfaces de JDBC

Liste des interfaces principales

Driver : renvoie une instance de **Connection**
Connection : connexion à une base
Statement : ordre SQL
PreparedStatement : ordre SQL paramétré
CallableStatement : procédure stockée sur le SGBD
ResultSet : lignes récupérées par un ordre SELECT
ResultSetMetaData : description des lignes récupérées par un SELECT
DatabaseMetaData : informations sur la base de données

Liste des classes principales

DriverManager : gère les drivers, lance les connexions aux bases

Date : date SQL

Time : heures, minutes, secondes SQL

TimeStamp : comme Time avec une précision à la microseconde

Types : constantes pour désigner les types SQL (pour les conversions avec les types Java)

Liste des exceptions

SQLException : erreurs SQL

est levée dès qu'une connexion ou un ordre SQL ne se passe pas correctement

la méthode **getMessage()** donne le message en clair de l'erreur

SQLWarning : avertissements SQL

DataTruncation : avertit quand une valeur est tronquée lors d'un transfert entre Java et le SGBD

0. Importer le package java.sql et configuration des paths

Mysql/Oracle puis ODBC, pas de DB2!

MySql

paquetage www.mysql.org

suivre les indications d'installation, la configuration des chemins, et le téléchargement du driver mm.mysql.jdb-xxx

Oracle

voir la démo#11, pour la partie sqlplus et démo#12 pour la partie JDBC.

inclure oracle

echo $CLASSPATH

CLASSPATH=/ora0/app/oracle/product/8.1.5/jdbc/lib/classes12.zip

import java.sql.*;

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

/**
 * Example 3.2. Livre Oreilly :
 * Database programming with JDBC and Java ; page 37-38
 */

public class SimpleConnection {

  static public void main(String args[]) {
    Connection connection = null;

    // Vérification des paramètres se trouvant
    // sur la ligne de commande
    if( args.length != 4 ) {
      System.out.println( "Syntax: java SimpleConnection "
              + "DRIVER URL UID PASSWORD");
      return;
    }
```

```
try { // chargement du driver
  Class.forName(args[0]).newInstance();
}

catch( Exception e ) { // prb. charg. du driver
                       // class inexistante?
  e.printStackTrace();
  return;
}
try {
  connection =
      DriverManager.getConnection(
                          args[1], args[2], args[3]);
  System.out.println("Connection successful!");
  // blablabla les requêtes à faire!!!!
}
catch( SQLException e ) {
  e.printStackTrace();
}
```

```
    finally {
      if( connection != null ) {
          try { connection.close(); }
          catch( SQLException e ) {
            e.printStackTrace();
          }
      }
    }
  }
}
```

1. Enregistrer le driver JDBC

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

Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Class.forName("oracle.jdbc.driver.OracleDriver");
Class.forName("org.gjt.mm.mysql.Driver");

quand une classe **Driver** est chargée, elle doit créer une instance d'elle même et s'enregistrer auprès du **DriverManager**

certains compilateurs refusent cette notation et demandent plutôt :
Class.forName("driver_name").newInstance();

2. Établir la connexion à la base de données

Une fois le driver enregistré, une connexion peut-être établie. Afin d'établir une connexion, il est nécessaire d'avoir un url pour la base de données. Cet url est construit suivant la méthodologie suivante :

Un URL pour une base de données est de la forme :

jdbc:*sous-protocole*:*base de donnée*

Par exemple, pour Oracle :

jdbc:oracle:thin:@titan:1521:a99

oracle: thin est le sous -protocole (driver "*thin*" ; Oracle fournit

aussi un autre type de driver)

@ titan:1521:a99 désigne la base de données a99 située sur la machine titan (le serveur écoute sur le port 1521)

La forme exacte des parties *sous-protocole* et *base de données* dépend des SGBD

MySql

jdbc:mysql://[hostname][:port]/dbname[?param1=value1]…

ODBC

jdbc:mysql:dbname

Obtenir une connexion …

Pour obtenir une connexion à un SGBD, on demande cette connexion à la classe gestionnaire de drivers :

Méthode **getConnection()** de **DriverManager**

3 arguments :

- l'URL de la base de données
- le nom de l'utilisateur de la base
- son mot de passe

Connection conn = **DriverManager.getConnection**(url,user,password) ;

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

```
static final String url =
        "jdbc:oracle:thin:@titan:1521:a99";

Connection conn =
    DriverManager.getConnection(url, "toto", "mdp");
```

Gestion des transactions (ensemble de requêtes SQL)

Par défaut la connexion est en "*auto-commit*" : un commit est automatiquement lancé après chaque ordre SQL qui modifie la base

On peut enlever l'auto-commit par :

conn.setAutoCommit(false)

conn.commit() valide la transaction.

conn.rollback() annule la transaction.

3. Créer une zone de description de requête

L'objet **Statement** possède les méthodes nécessaires pour réaliser les requêtes sur la base associée à la connexion dont il dépend

3 types de **Statement** :

Statement : requêtes statiques simples

PreparedStatement : requêtes dynamiques précompilées (avec paramètres d'entrée/sortie)

CallableStatement : procédures stockées

A partir de l'instance de l'objet **Connection**, on récupère le **Statement** associé :

Statement req1 = conn.**createStatement() ;**

PreparedStatement req2 =
conn.**prepareStatement(str)** ;

CallableStatement req3 =
conn.**prepareCall(str)** ;

4. Exécuter la 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 (cas rares)

executeQuery() et **executeUpdate()** de la classe **Statement** prennent comme argument une chaîne (**String**) indiquant la requête SQL à exécuter :

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

ResultSet rset = stmt.executeQuery ("select * from user_catalog");
```

le code SQL n'est pas interprété par Java.

— c'est le pilote associé à la connexion (et au final par le moteur de la base de données) qui interprète la requête SQL

— si une requête ne peut s'exécuter ou qu'une erreur de syntaxe SQL a été détectée, l'exception **SQLException** est levée

le driver JDBC effectue d'abord un accès à la base pour découvrir les types des colonnes impliquées dans la requête puis un 2ème pour l'exécuter.

5. Traiter les données retournées

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

seules les données demandées sont transférées en mémoire par le driver JDBC

— il faut donc les lire "manuellement" et les stocker dans des variables pour un usage ultérieur

JDBC 1.x : 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

Avec le JDBC 2.0 :

on peut parcourir le **ResultSet** d'avant en arrière :

— next() vs. previous()

en déplacement absolu : aller à la n-ième ligne

— absolute(int row), first(), last(), ...

en déplacement relatif : aller à la n-ième ligne à partir de la position courante du curseur, … :

— relative(int row), afterLast(), beforeFirst(), ...

Les colonnes sont référencées par leur numéro (commencent à 1) 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") ;**

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

```
/*
 * Cet exemple montre comment obtenir la liste des usagers
 * disposant d'un compte dans la base. (de la doc. de Oracle)
 */

import java.sql.*;

class Employee
{
  public static void main (String args [])
       throws SQLException
  {
    // Chrgement du driver JDBC-Oracle
    DriverManager.registerDriver(
            new oracle.jdbc.driver.OracleDriver());
```

```
    // Connexion à la base de données
    // syntaxe <host>:<port>:<sid>.
    Connection conn =
      DriverManager.getConnection (
                "jdbc:oracle:thin:@titan:1521:a99",
                "momo", "toto");
    // Créer un descripteur de requêtes
    Statement stmt = conn.createStatement ();

    // Sélectionner les noms de la table
    ResultSet rset = stmt.executeQuery (
                "select username from dba_users");

    // Itérer afin d'obtenir la liste de tous
    // les noms !
    while (rset.next ())
      System.out.println (rset.getString (1));
  }
}
```

Accès au 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 nom de la table : **getTableName(int col)**
- si un NULL SQL peut être stocké dans une colonne : **isNullable()**

```
ResultSet rs =
            stmt.executeQuery("SELECT * FROM emp") ;

ResultSetMetaData rsmd = rs.getMetaData() ;

int nbColonnes = rsmd.getColumnCount() ;

for (int i = 1 ; i<=nbColonnes ; i++) {
   // à partir de 1 et non pas 0
   String nomCol = rsmd.getColumnName(i) ;
}
```

DatabaseMetaData

Pour récupérer des informations sur la base de données elle-même, utiliser la méthode **getMetaData()** de l ’objet **Connection**

dépend du SGBD avec lequel on travaille

elle renvoie des **DatabaseMetaData**

on peut connaître entre autres :

- les tables de la base : **getTables()**
- le nom de l’utilisateur : **getUserName()**

Type de données traitées

La classe ResultSet fournit des méthodes pour récupérer dans le code Java les valeurs des colonnes des lignes renvoyées par le SELECT:

getXXX(int numéroColonne)
getXXX(String nomColonne)

XXX désigne le type Java de la valeur que l'on va récupérer, par exemple String, Int ou Double

Tous les SGBD n'ont pas les mêmes types SQL ; même pour les types de base on peut avoir des différences importantes

JDBC définit ses propres types SQL dans la classe **Types** sous forme de constantes nommées

Dans un programme JDBC, les méthodes getXXX, setXXX et les types de la classe **Types** servent à préciser le type des données que l'on insère dans la base ou que l'on récupère

C'est le rôle du driver particulier à chaque SGBD de traduire les données Java dans le bon type du SGBD

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

Presque tous les types SQL peuvent être retrouvés par getString()

Cependant, voici les méthodes recommandées :

CHAR et VARCHAR : getString, LONGVARCHAR : getAsciiString et getCharacterString

BINARY et VARBINARY : getBytes,

LONGVARBINARY : getBinaryStream

BIT : getBoolean, TINYINT : getByte, SMALLINT : getShort, INTEGER : getInt, BIGINT: getLong

REAL : getFloat, DOUBLE et FLOAT : getDouble

DECIMAL et NUMERIC : getBigDecimal

etc.

cas des valeurs nulles

Pour repérer les valeurs NULL de la base :

utilise la méthode **wasNull()** de **ResultSet**

— renvoie **true** si l'on vient de lire un NULL, **false** sinon

les méthodes **getXXX()** de **ResultSet** convertissent une valeur NULL SQL en une valeur acceptable par le type d'objet demandé :

— les méthodes retournant un objet (**getString()**, **getDate(),...**) retournent un "**null** " Java

— les méthodes numériques (**getByte()** , **getInt()** , etc) retournent "**0**"

— **getBoolean()** retourne " **false** "

```
Statement stmt1 = conn.createStatement();

ResultSet rset =
   stmt1.executeQuery("SELECT nome, comm FROM emp");

Float commission;

while (rset.next()) {
         nom = rset.getString(1);
         commission = rset.getFloat(2);
         if (rset.wasNull())
            System.out.println(nom +
                     " n'a pas de commission");
         else
            System.out.println(nom + " a " +
             commission + "F de commission");
}
```

6. Fermeture des différents espaces

À la fin du programme, il faut fermer toutes les connexions ouvertes au début du programme, sinon le *garbage collector* s'en occupera mais moins efficace.

Chaque objet possède une méthode **close()** :

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