ACCES DISTANT AUX BASES DE DONNEES

CLIENT/SERVEUR
CLIENT/MULTI-SERVEUR

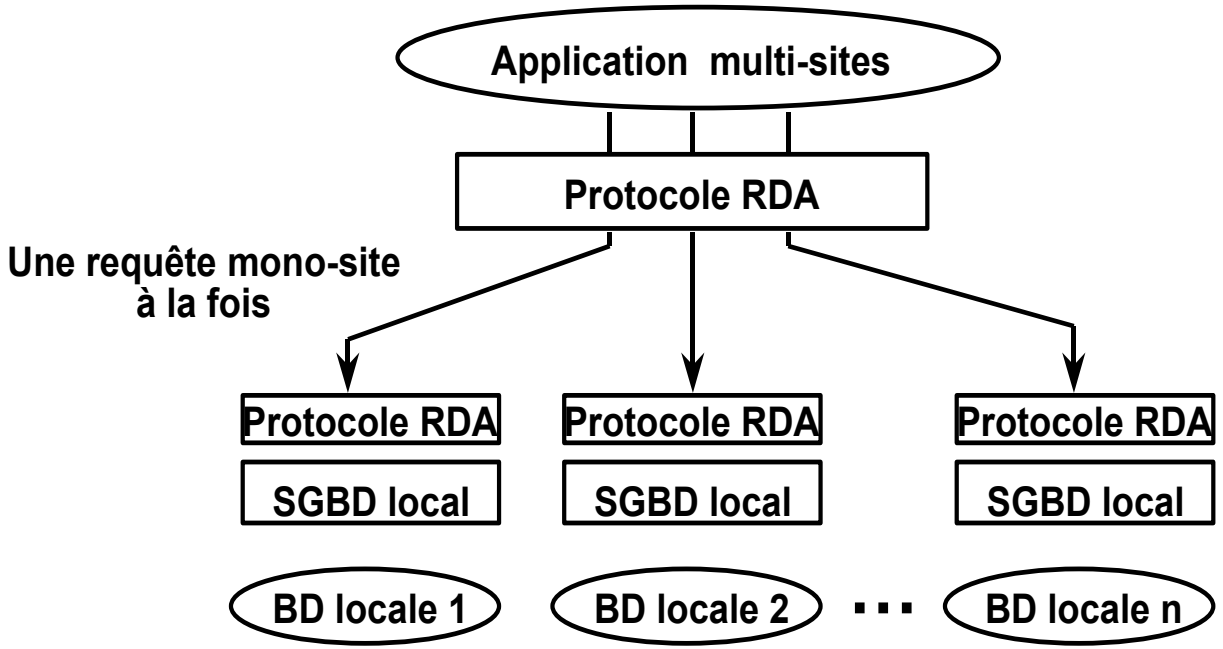
LA SOLUTION JAVA : JDBC

Le client/serveur

- **Les standards:**
  - RDA (Remote Data Access) de l'ISO
  - SQL-CLI (Client Level Interface) de X/Open

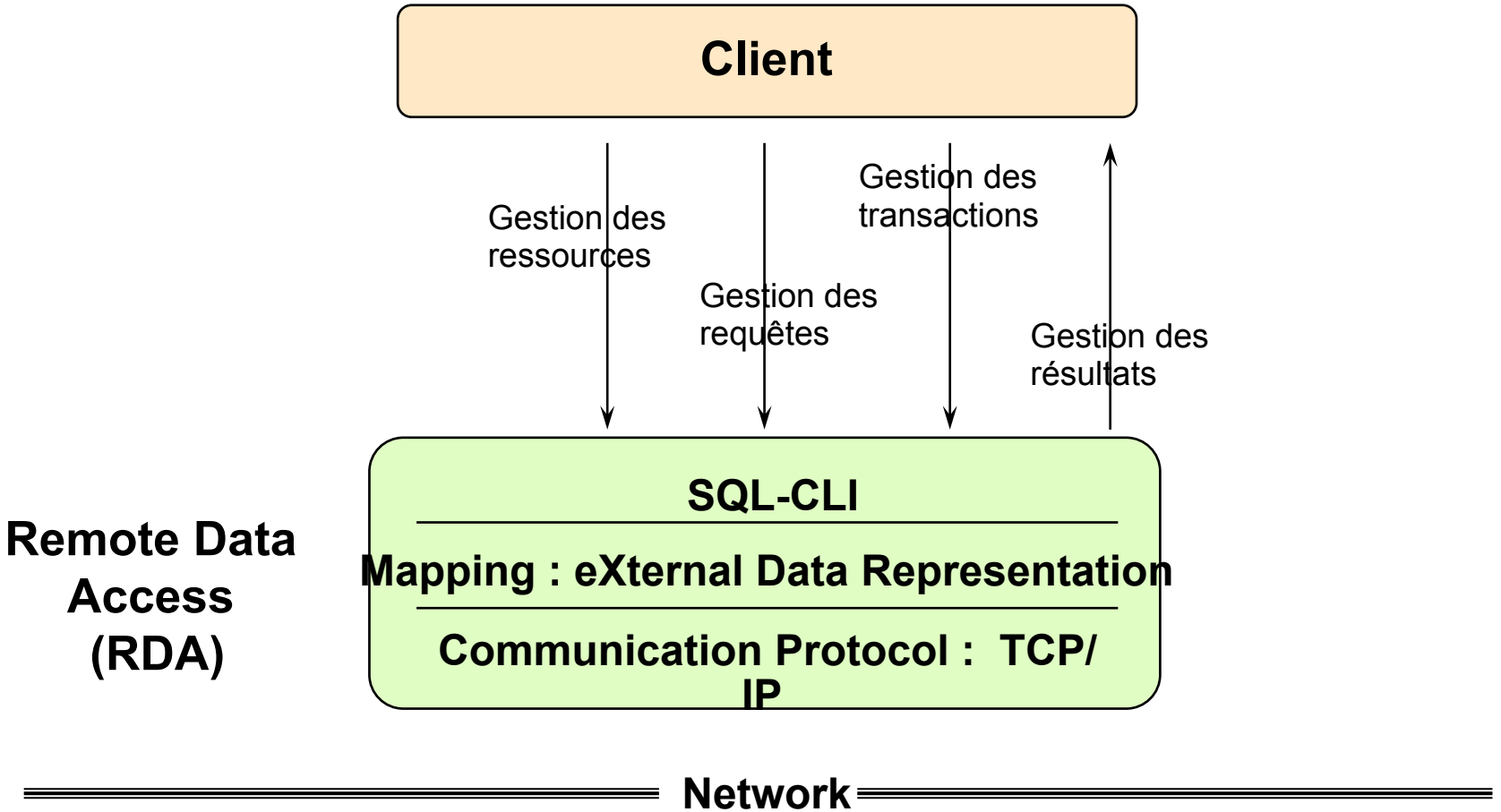
- **Les produits:**
  - ODBC (Open Database Connectivity) de Microsoft
  - IDAPI de Borland, DRDA d'IBM,
  - JDBC (Java DataBase Connectivity) API de SUN...

Solution RDA : Caractéristiques

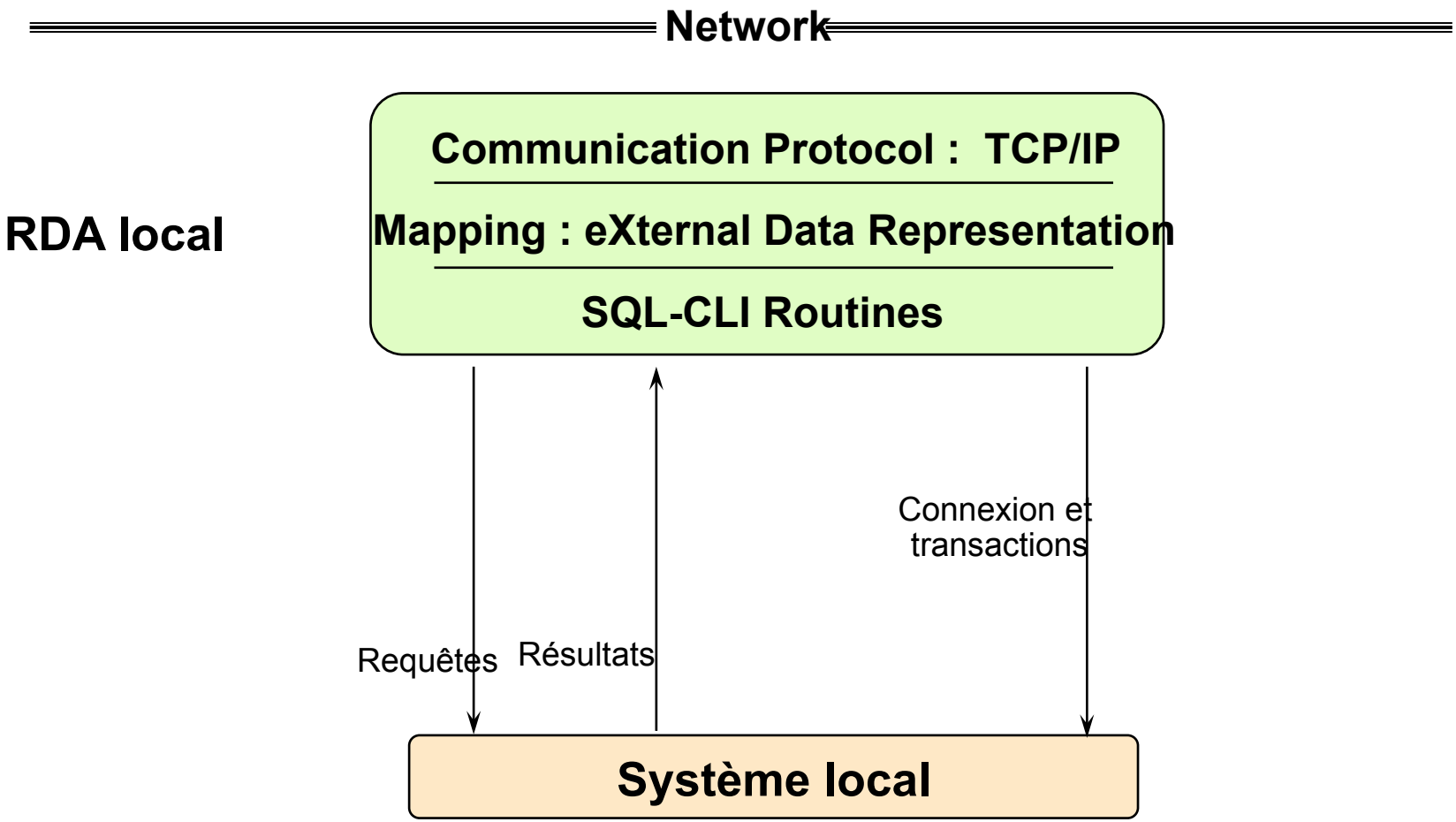


- **Les usagers connaissent la localisation des tables**

La couche de communication



La couche de communication



Scenario

AllocEnv()

AllocConnect()

Connect()

AllocStmt()

Sending Queries

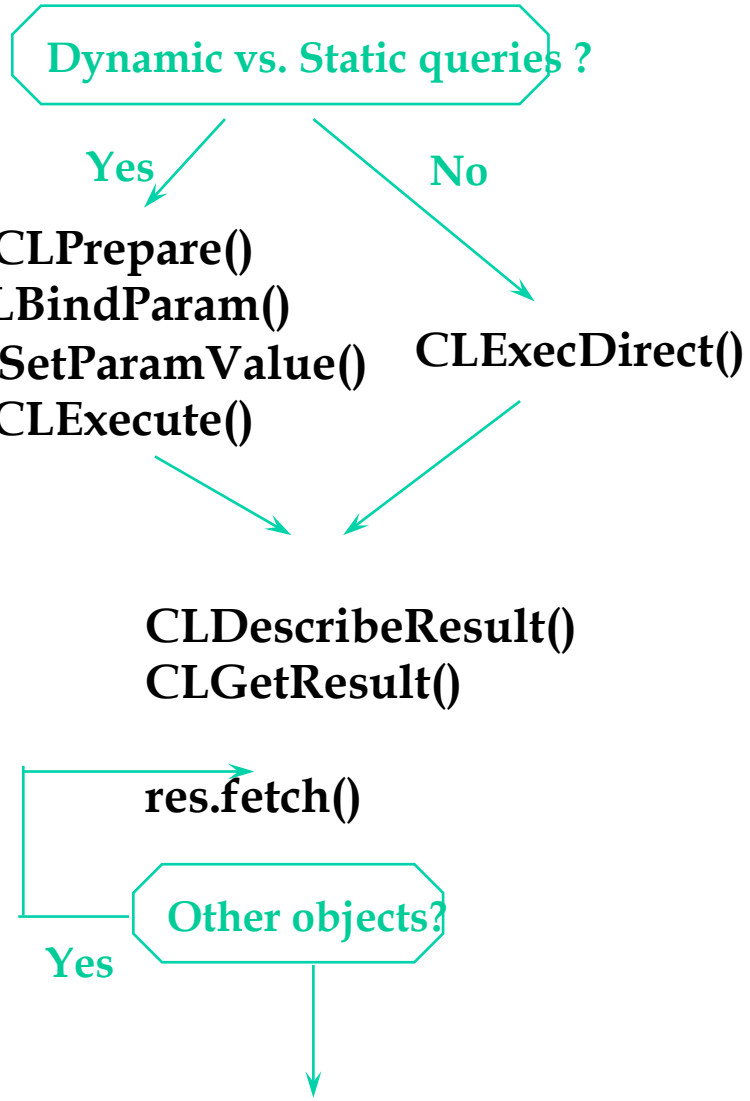
Receiving Results

FreeStmt()

Disconnect()

FreeConnect()

FreeEnv()



JDBC ?

- **Java est un excellent candidat pour le développement d'applications de bases de données :**
  - robuste et sécurisé
  - facile à comprendre
  - automatiquement téléchargeable par le réseau

- **mais avant JDBC, il était difficile d'accéder à des bases de données SQL depuis Java :**
  - obligé d'utiliser des API natives comme ODBC

Objectifs de JDBC

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

- **Réalisé par l'API JDBC :**
  - une interface uniforme permettant un accès homogène aux SGBDs
  - simple à mettre en œuvre
  - indépendante du SGBD cible
  - supportant les fonctionnalités de base du langage SQL

Qu’est ce que JDBC ?

- **Java DataBase Connectivity (Core API 1.1)**
- **API Java adaptée à la connexion avec les bases de données relationnelles (SGBDR)**
- **Fournit un ensemble de classes et d’interfaces permettant l’utilisation sur le réseau d’un ou plusieurs SGBDRs à partir d’un programme Java.**

Avantages

- **Liés à Java :**
  - portabilité sur de nombreux O.S. et sur de nombreux SGBDR (Oracle, Informix, Sybase, ..)
  - uniformité du langage de description des applications, des applets et des accès aux bases de données
  - liberté totale vis a vis des constructeurs

L'API JDBC

- **Est fournie par le package java.sql**
  - permet de formuler et gérer les requêtes aux bases de données relationnelles
  - supporte le standard « SQL-2 Entry Level »
  - 8 interfaces définissant les objets nécessaires :
    - à la connexion à une base éloignée
    - et à la création et exécution de requêtes SQL

java.sql

- **8 interfaces :**
  - Statement
  - CallableStatement, PreparedStatement
  - DatabaseMetaData, ResultSetMetaData
  - ResultSet,
  - Connection
  - Driver

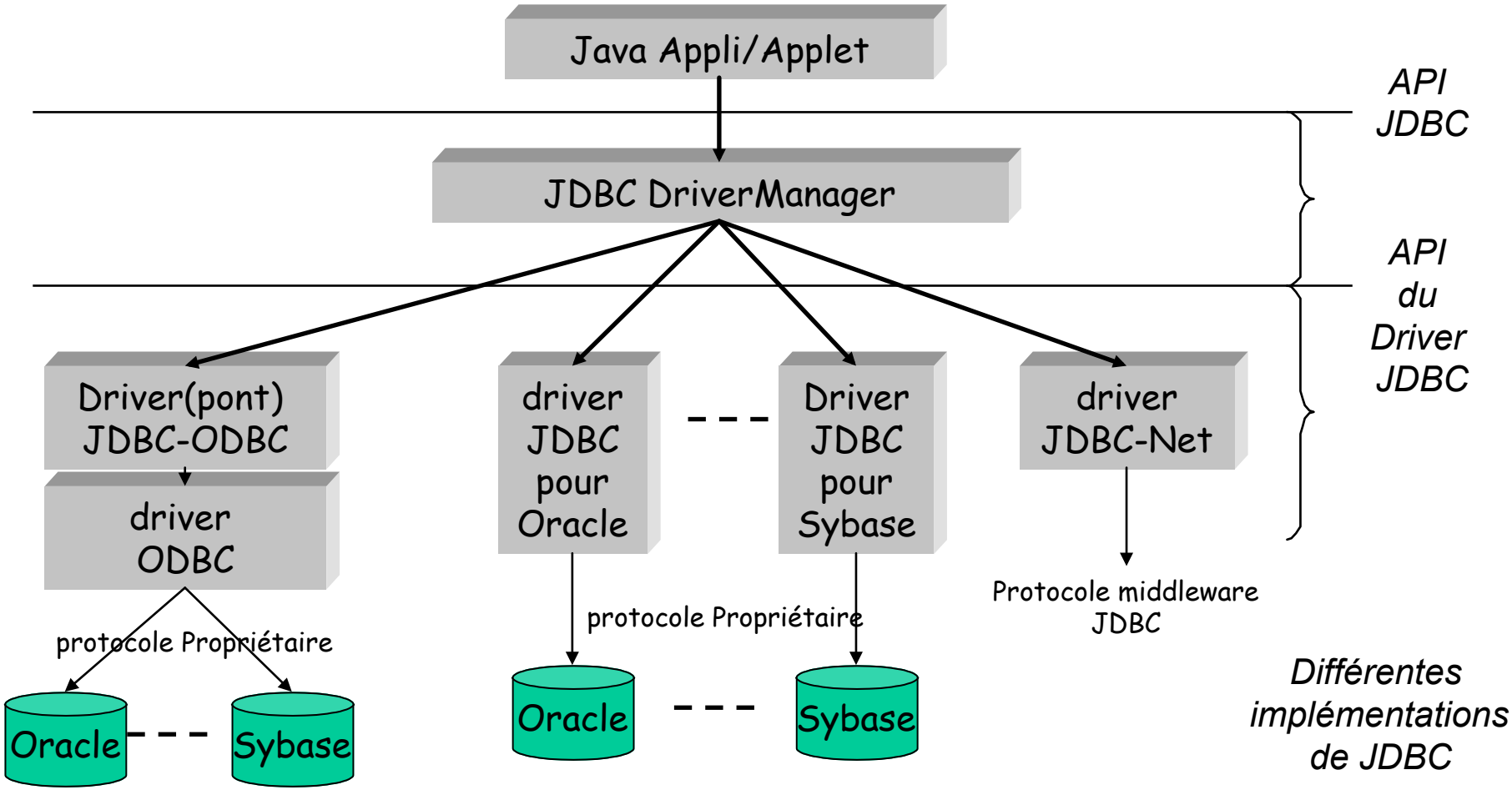
Principe de fonctionnement

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

- **Ces drivers dits JDBC (contenant un ensemble de classes et d'interfaces Java) existent pour tous les constructeurs principaux :**
  - Oracle, Sybase, Informix, DB2, ...

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

Architecture JDBC



Types de drivers et applets

- **Une application Java peut travailler avec tous les types de drivers**

- **Pour une applet (untrusted), code qui peut s'exécuter dans un navigateur Web :**
  - type I ou II : impossible
    - une applet ne peut pas charger à distance du code natif (non Java) sur son poste d 'exécution
  - type III : possible
    - si le serveur middleware se situe au même endroit que le serveur Web (car communication par sockets avec l 'applet)
  - type IV : possible
    - si le SGBDR est installé au même endroit que le serveur Web

Une alternative : 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 exemple)

Modèles de connexion en Java

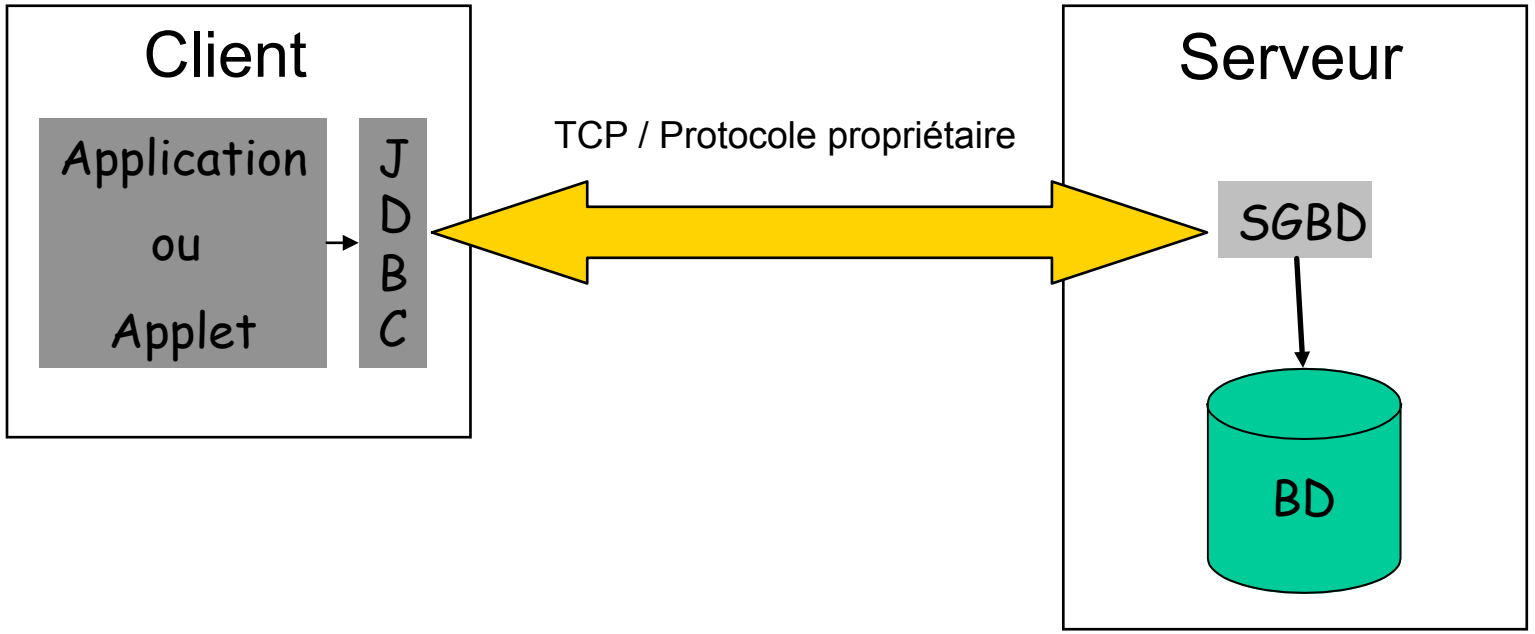
- **Modèle 2-tiers : 2 entités interviennent**
  - 1. une application Java ou une applet
  - 2. le SGBDR

- **Modèle 3-tiers : 3 entités interviennent**
  - 1. une application Java ou une applet
  - 2. un serveur middleware installé sur le réseau
  - 3. le SGBDR

Modèle 2-tiers

- **Principe :**
  - l ’application (ou l ’applet) cliente utilise JDBC pour parler directement avec le SGBD qui gère la base de données
- **Avantages :**
  - simple à mettre en œuvre
  - bon choix pour des applications clientes peu évoluées, à livrer rapidement et n’exigeant que peu de maintenance
- **Inconvénients :**
  - dépendance forte entre le client et la structure du SGBDR
    - modification du client si l ’environnement serveur change
  - tendance à avoir des clients « trop gros »
    - tout le traitement est du côté client

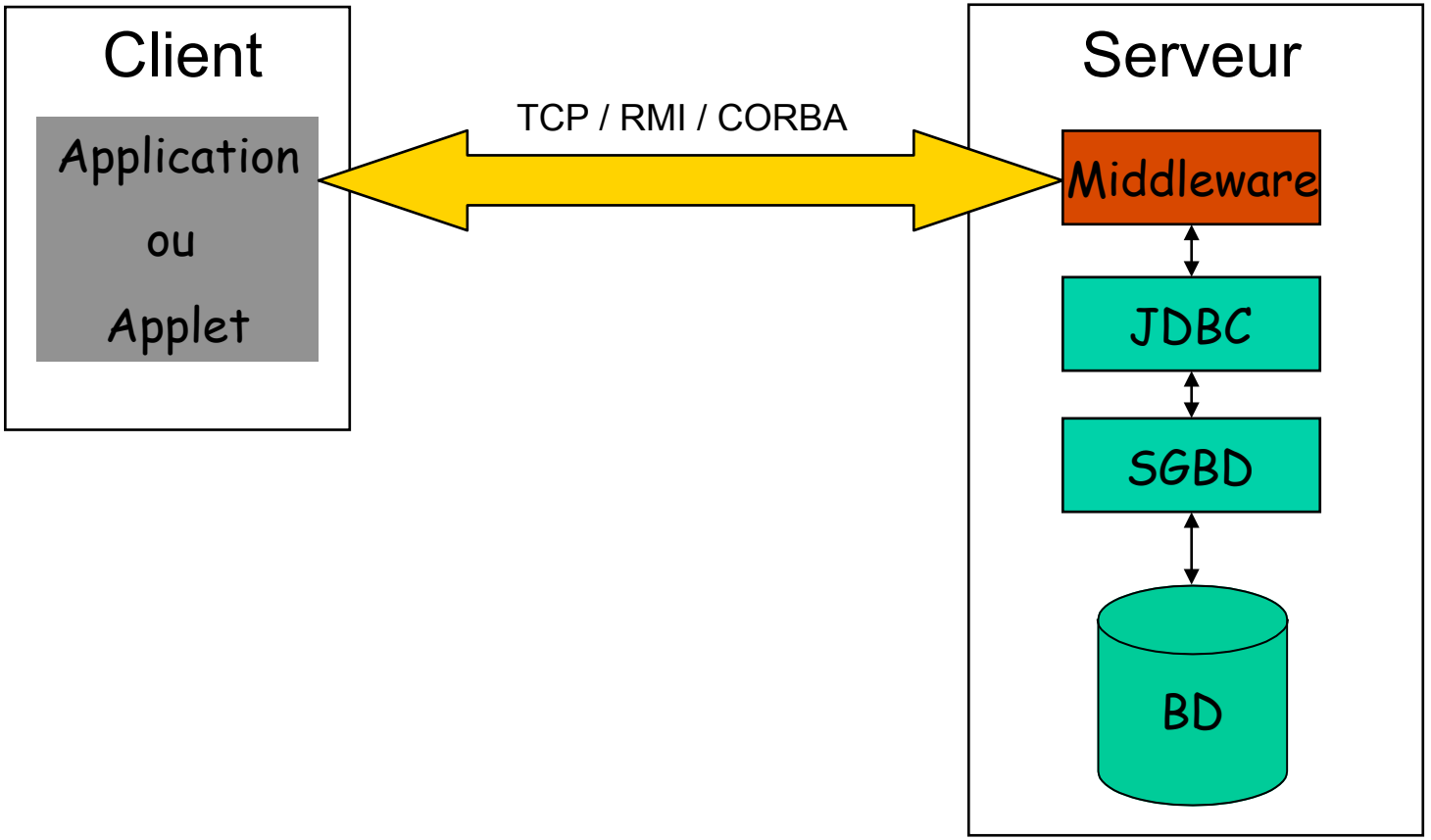
Architecture 2-tiers



Modèle 3-tiers

- **Principes :**
  - le serveur middleware est l ’interlocuteur direct du code Java client; c ’est lui qui échange des données avec le SGBDR
  - pas forcement écrit en Java
  - si c ’est le cas : utilise souvent JDBC pour accéder au SGBDR
- **Avantages:**
  - le middleware peut ajouter un niveau de sécurité
  - plusieurs supports pour les échanges avec le client :
    - sockets, RMI Java, CORBA, …
  - applets : le SGBDR peut se trouver sur une autre machine :
    - mais serveur Web et middleware au même endroit
  - facilite l ’utilisation de clients « légers »

Architecture 3-tiers



Mettre en œuvre JDBC

0. Importer le package java.sql

1. Enregistrer le driver JDBC

2. Etablir la connexion à la base de données

3. Créer une zone de description de requête

4. Exécuter la requête

5. Traiter les données retournées

6. Fermer les différents espaces

Enregistrer le driver JDBC

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

  DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());

- **certains compilateurs refusent cette notation et demande plutôt la méthode forName() de la classe Class :**

  Class.forName("postgresql.Driver");

  Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");

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

Etablir une connexion (1)

- **Accès à la base via une URL de la forme :**

  jdbc:<sous-protocole>:<nom-BD>;param=valeur, ...

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

    ```
    String url = "jdbc:postgresql://visual.gl.epita.fr/qdm";
    String url = "jdbc:msql://basic.gl.fr:1114:BaseClient";
    String url =  "jdbc:odbc:oracle " ;
    String url = " jdbc:oracle:thin:@europe:1521:oracle ";
    ```

Connexion à la base

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

Connection connect =

DriverManager.getConnection(url,user,password);

- le DriverManager essaie 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.

Création d'un Statement (1/2)

- **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 pré-compilées (avec paramètres d'entrée/sortie)
  - CallableStatement : procédures stockées

Création d'un Statement (2/2)

- **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 (1/3)

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

Exécution d ’une requête (2/3)

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

```
Statement st = connexion.createStatement();
ResultSet rs = st.executeQuery(
        "SELECT nom, prenom FROM clients " +
        "WHERE nom = ‘GL’ ORDER BY nom");
int nb = st.executeUpdate("INSERT INTO USERS " +
                          "VALUES(’machin’,’truc’)");
```

Exécution d ’une requête (3/3)

- **2 remarques :**
  - le code SQL n’est pas interprété par Java.
    - c’est le pilote associé à la connexion (et au final par le moteur de la base de données) qui interprète la requête SQL.
    - si une requête ne peut s’exécuter ou qu’une erreur de syntaxe SQL a été détectée, l’exception SQLException est levée.
  - le driver JDBC effectue d’abord un accès à la base pour découvrir les types des colonnes impliquées dans la requête puis un 2ème pour l’exécuter.

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

Le résultat : ResultSet (1/3)

- **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

Le résultat : ResultSet (2/3)

- **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") ;
```

Le résultat : ResultSet (3/3)

```
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
….	…..

Cas des valeurs nulles

- **Pour repérer les valeurs NULL de la base :**
  - utiliser la méthode wasNull() de ResultSet
    - renvoie true si l'on vient de lire un NULL, false sinon
  - les méthodes getXXX() de ResultSet convertissent une valeur NULL SQL en une valeur acceptable par le type d 'objet demandé :
    - les méthodes retournant un objet (getString() , getObject() et getDate()) retournent un "null " Java
    - les méthodes numériques (getByte() , getInt() , etc) retournent "0"
    - getBoolean() retourne " false "

Fermer les différents espaces

- **Pour terminer proprement un traitement, il faut fermer les différents espaces ouverts**
  - sinon le garbage collector s'en occupera mais moins efficacement

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

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

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()

ResultSetMetaData

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

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

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()
    - ...

Requêtes pré-compilées

- **L'objet PreparedStatement envoie une requête sans paramètres à la base de données pour pré-compilation et spécifiera le moment voulu la valeur des paramètres**

- **plus rapide qu'un Statement classique**
  - le SGBD analyse qu'une seule fois la requête (recherche d'une stratégie d'exécution adéquate)
  - pour de nombreuses exécutions d'une même requête SQL avec des paramètres variables
- tous les SGBD n'acceptent pas les requêtes pré-compilées

Création d ’une requête pré-compilée

La méthode prepareStatement() de l ’objet Connection crée un PreparedStatement :

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

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

Exécution d'une requête pré-compilée

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

Validation de transaction : Commit

- Utiliser pour valider tout un groupe de transactions à la fois
- Par défaut : mode auto-commit
  - un "commit " est effectué automatiquement après chaque ordre SQL
- Pour repasser en mode manuel :

  connection.setAutoCommit(false);

- L'application doit alors envoyer à la base un "commit" pour rendre permanent tous les changements occasionnés par la transaction :

  connection.commit();

Annulation de transaction : Rollback

- **De même, pour annuler une transaction (ensemble de requêtes SQL), l'application peut envoyer à la base un "rollback" par :**

```
connection.rollback();
```

  - restauration de l'état de la base après le dernier "commit"

Exceptions

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

JDBC et Oracle 8

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

import java.sql.*;

Class.forName("oracle.jdbc.driver.OracleDriver");
static final url = "jdbc:oracle:thin:@local:1521:QDM";
con = DiverManager.getConnection(url,"michel","christophe");
```

JDBC et les mécanismes de sécurité Java

- **Rappel : avec le JDK 1.0, une applet ne peut pas charger un driver natif (I ou II) pour accéder à une base de données distante**
  - pour y remédier: drivers III ou IV et modèle 3-tiers

- **Avec le JDK 1.1 : API de sécurité**
  - une applet peut, sous certaines conditions de signature, accéder à un driver natif
  - et se connecter directement au serveur du SGBD

Performances du JDBC standard

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

Conclusions (1/2)

- **Conclusions sur l'API JDBC :**
  - jeu unique d'interfaces pour un accès homogène
    - cache au maximum les diverses syntaxes SQL des SGBD
  - API de bas niveau
    - nécessaire de connaître la syntaxe SQL
  - 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

Conclusions (2/2)

- 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

Les dernières versions de JDBC

- **Actuellement :**
  - API JDBC 2.0 (inclus dans la version Java 2)
    - software :
      - http://java.sun.com/products/jdbc/jdbcse2.html
    - documentation :
      - http://java.sun.com/products/jdbc/
    - sur les drivers JDBC :
      - http://java.sun.com/products/jdbc/jdbc.drivers.html

Evolutions prévues

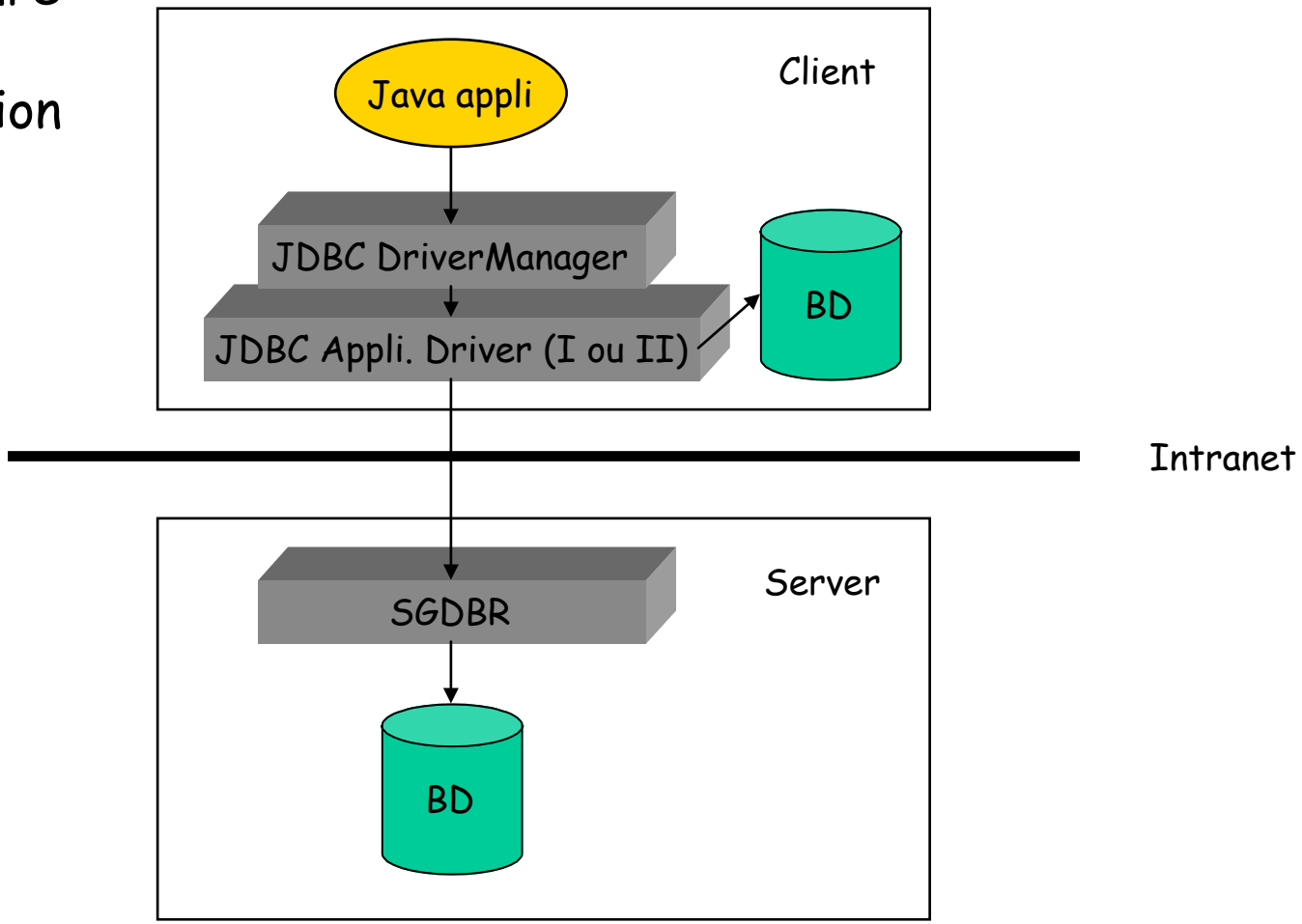
- **3 évolutions importantes prévues :**
  - spécification de J/SQL par Oracle, Tandem, IBM et JavaSoft destinée à :
    - faciliter l'accès au schémas des BDs, augmenter les performances et améliorer les développements
  - spécification de Java Binding ODMG (accès aux BDs objets) par l 'OMG (Object Management Group)
  - JavaSoft prépare Jblend :
    - ensemble d'outils pour effectuer un mapping bidirectionnel entre objet et base relationnelle

Scénarios d'utilisation

- **Scénario 1 :**
  - architecture 2-tiers avec une application Java
- **Scénario 2 :**
  - architecture 2-tiers avec une applet Java
- **Scénario 3 :**
  - architecture 2-tiers Plus avec une applet Java
- **Scénario 4 :**
  - architecture 3-tiers et applet/application Java

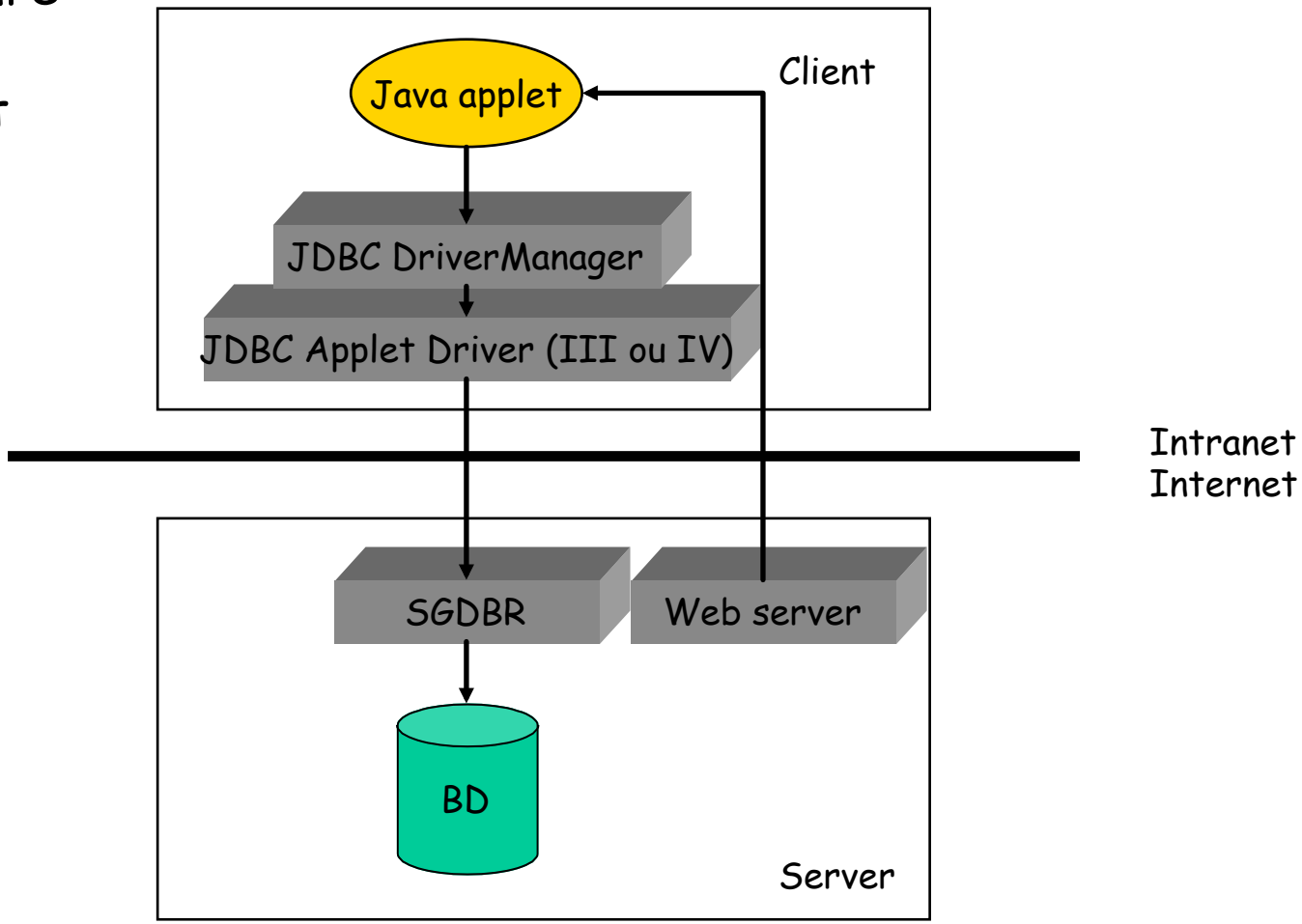
Scénario 1

Architecture
2-tiers
et application



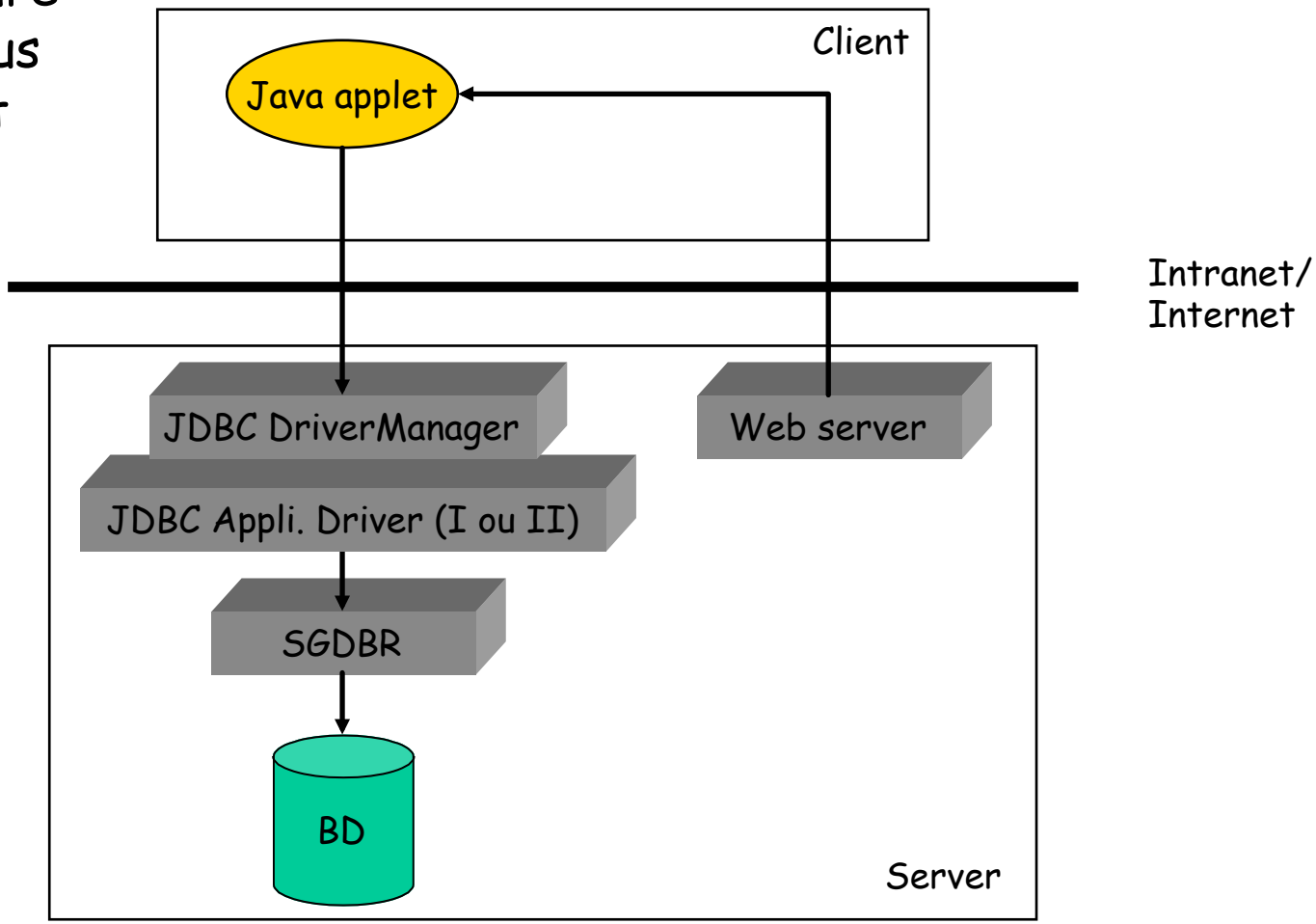
Scénario 2

Architecture
2-tiers
et applet

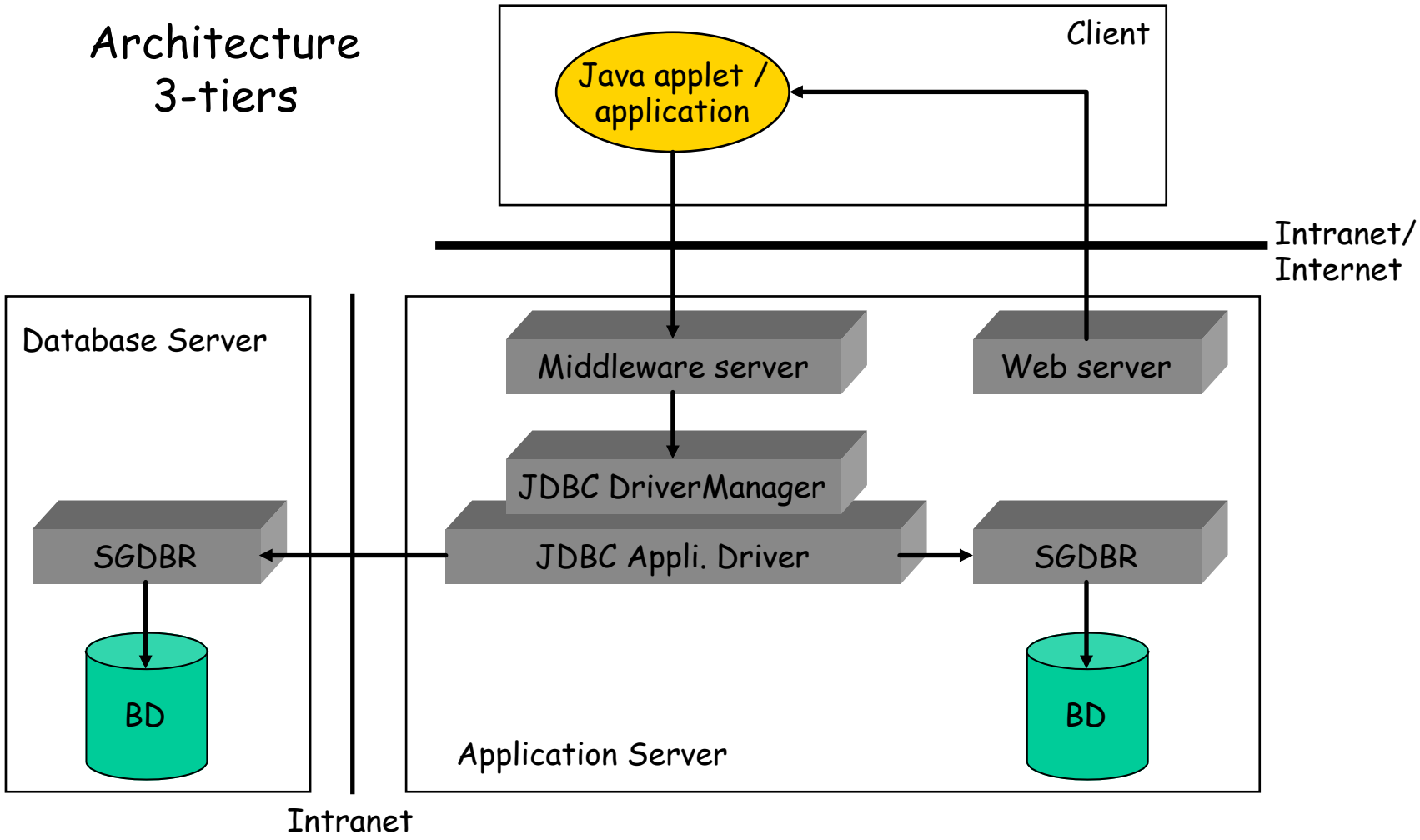


Scénario 3

Architecture
2-tiers Plus
et applet



Scénario 4


Architecture
3-tiers
Client
Java applet /
application
Intranet/
Internet
Database Server
Middleware server
Web server
JDBC DriverManager
SGDBR
JDBC Appli. Driver
SGDBR
BD
BD
Application Server
Intranet