

Cours de Programmation Orientée Objet JAVA

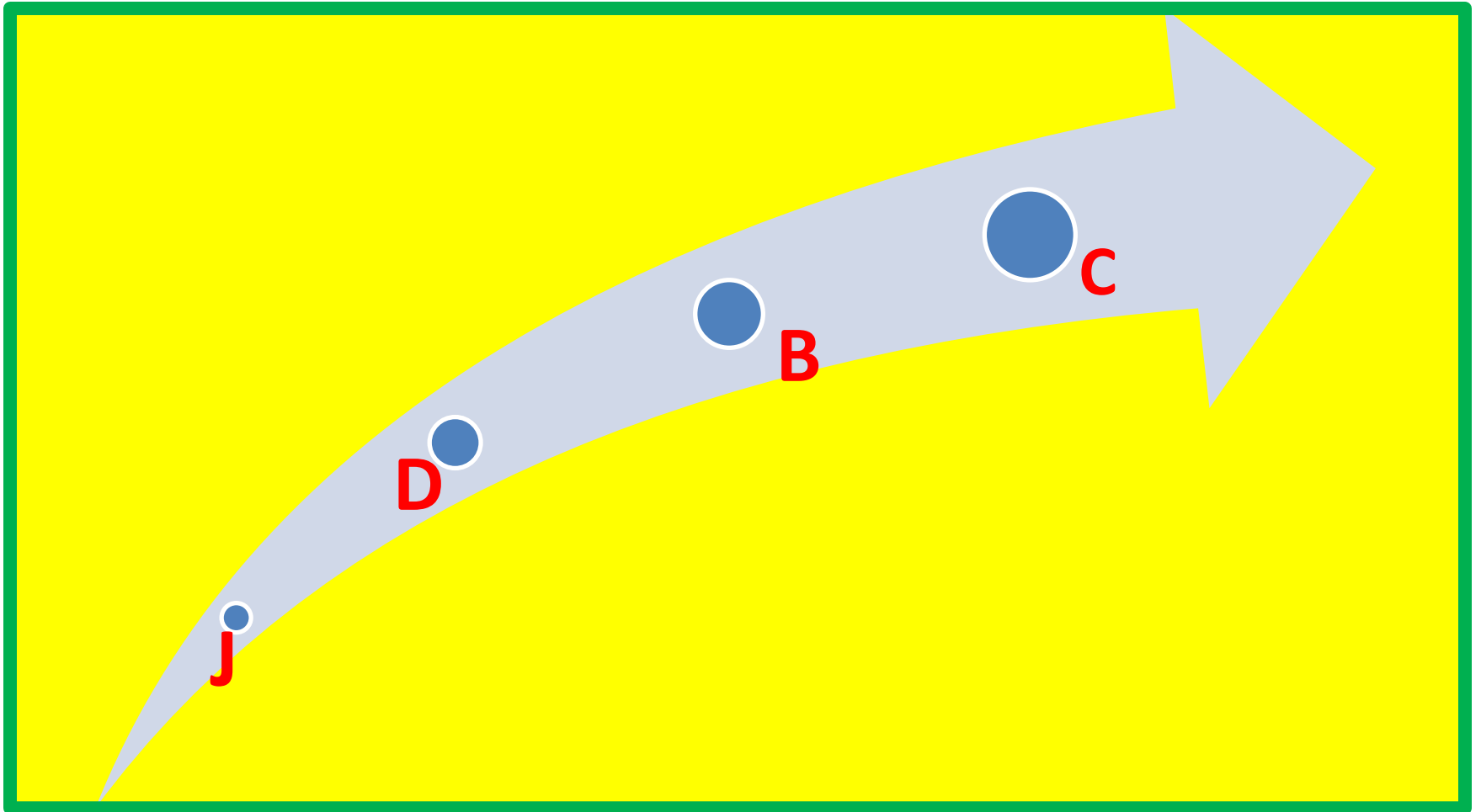
Demba SOW

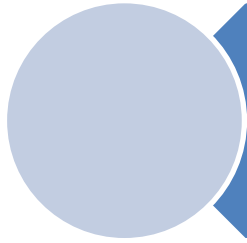
*Docteur en
Mathématiques et
Cryptologie*

L.A.C.G.A.A.

F.S.T. / U.C.A.D.





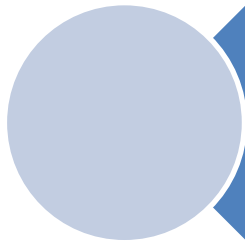


Java DataBase Connectivity: JDBC

JDBC est une API Java (ensemble de classes et d'interfaces défini par SUN et les acteurs du domaine des BD) permettant d'accéder aux bases de données à l'aide du langage Java via des requêtes SQL (langage permettant de dialoguer avec un SGBDR).

Cette API permet d'atteindre de façon quasi transparente des bases Sybase, Oracle, Informix,... avec le même programme Java JDBC.

JDBC est fourni par le paquetage `java.sql`



Principe de JDBC

Java DataBase Connectivity (JDBC)

permet à un programme Java d'interagir

- localement ou à distance
- avec une base de données relationnelle

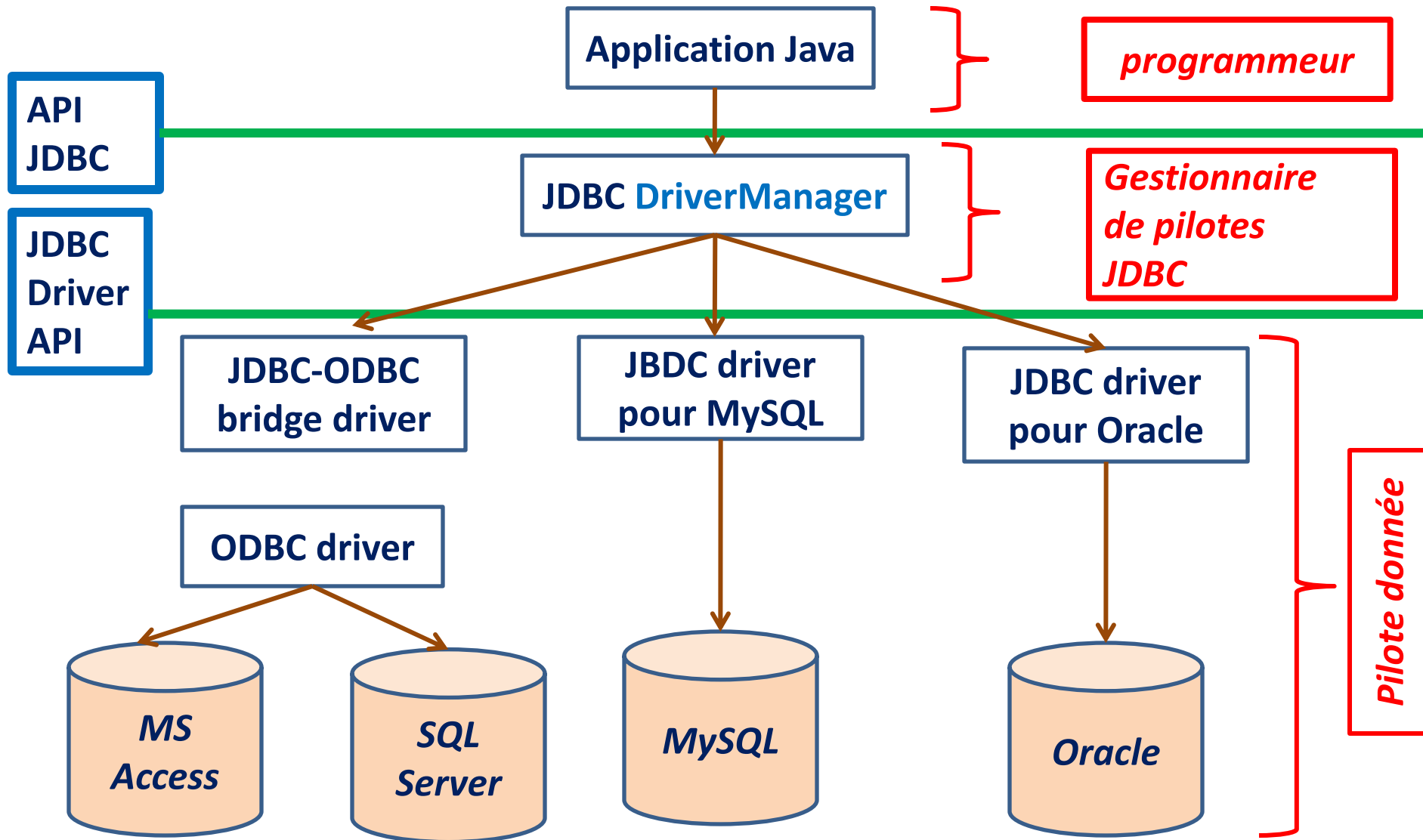
fonctionne selon un principe client/serveur

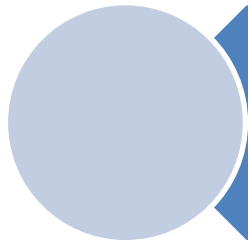
- client = le programme Java
- serveur = la base de données

principe

- le programme Java ouvre une connexion
- il envoie des requêtes SQL
- il récupère les résultats de la requête
- (ici les traitements à faire sur les données recueillies)
- il ferme la connexion une fois les traitements terminés

Architecture JDBC





Pilotes (drivers)

L'ensemble des classes qui implémentent les interfaces spécifiées par JDBC pour un SGBD (Système de Gestion de Bases de Données) particulier est appelé un **pilote JDBC**. Les protocoles d'accès aux bases de données étant propriétaires, il y a donc plusieurs drivers pour atteindre diverses BD.

Chaque BD utilise un pilote 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.

Pour travailler avec un SGBD, il faut disposer de classes (driver) qui implémentent les interfaces de JDBC.

JDBC est totalement indépendant de tout SGBD: la même application peut être utilisée pour accéder à une base Oracle, Sybase, MySQL, etc.

Les Types de drivers (1/4)

Il existe *quatre grandes familles de pilotes JDBC en Java*

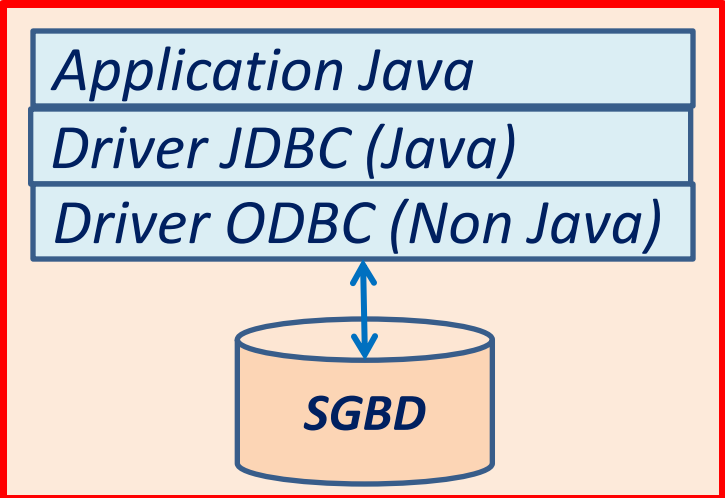
Type I: pont (ou passerelle) JDBC-ODBC

Le driver accède à un SGBDR en passant par les drivers ODBC (standard Microsoft: Open DataBase Connectivity) via un pont JDBC-ODBC:

Les appels JDBC sont traduits en appels ODBC

Ce type de pilote ne peut pas être utilisé par des *applets* puisque il ne permet qu'un accès local.

Il est fourni par Sun: `sun.jdbc.odbc.JdbcOdbcDriver`



The diagram illustrates the architecture of a Type I JDBC driver. It is enclosed in a red rectangular border. At the top, there is a stack of three light blue rectangular boxes. The top box is labeled 'Application Java'. The middle box is labeled 'Driver JDBC (Java)'. The bottom box is labeled 'Driver ODBC (Non Java)'. Below this stack, there is a light orange cylinder representing a database, labeled 'SGBD'. A blue double-headed vertical arrow connects the bottom of the 'Driver ODBC (Non Java)' box to the top of the 'SGBD' cylinder, indicating bidirectional communication between the ODBC driver and the database.

Application Java
Driver JDBC (Java)
Driver ODBC (Non Java)

SGBD

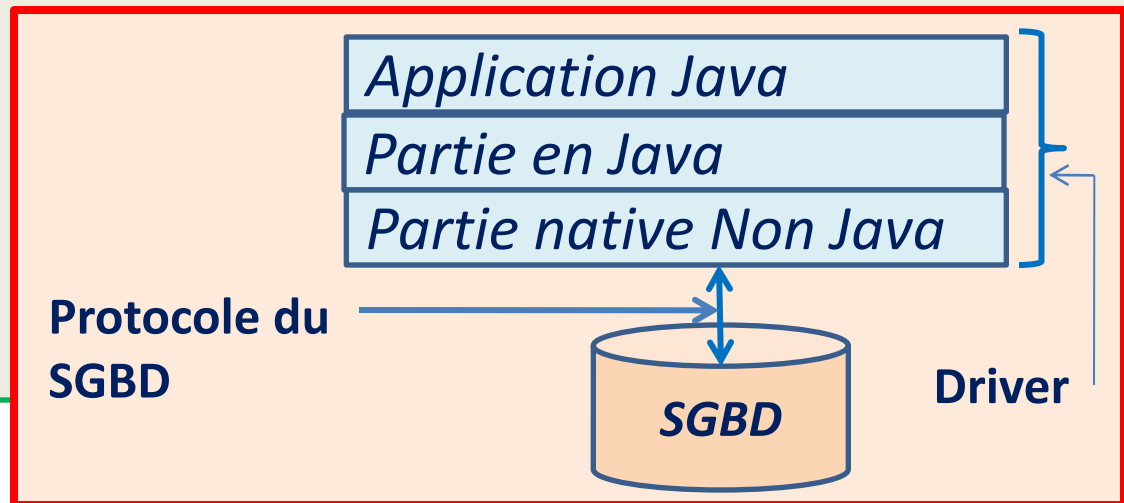
Les Types de drivers (2/4)

Type II: partiellement Java

Ce sont des drivers partiellement écrits en Java et qui reposent sur des librairies propriétaires (des fonctions natives non Java : par ex. C) pour accéder au SGBD. Ils peuvent gérer des appels C/C++ directement avec la base.

Ne convient pas pour les applets (sécurité).

Interdiction de charger du code natif dans la mémoire vive de la plateforme.



Les Types de drivers (3/4)

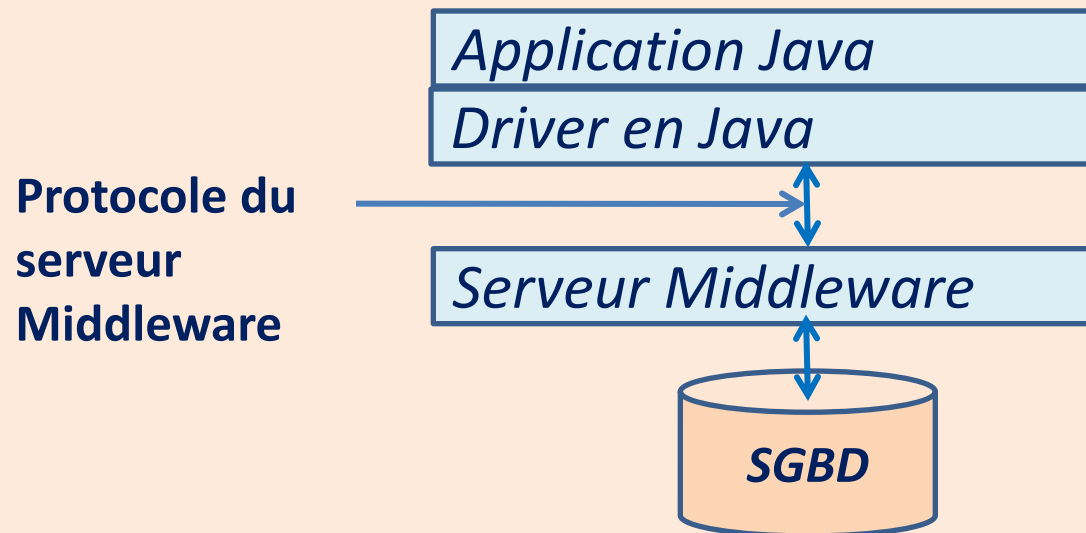
Type III: drivers 100% Java

Ils communiquent localement ou à distance avec le SGBD selon un protocole réseau générique (Sockets). La communication se fait par une application intermédiaire (Middleware) sur le serveur.

Le middleware accède par un moyen quelconque (par exemple JDBC si écrit en Java) aux différents SGBDR.

Ils sont portables car entièrement écrits en Java.

Donc appropriés pour les *Applets et les Applications*.



Les Types de drivers (4/4)

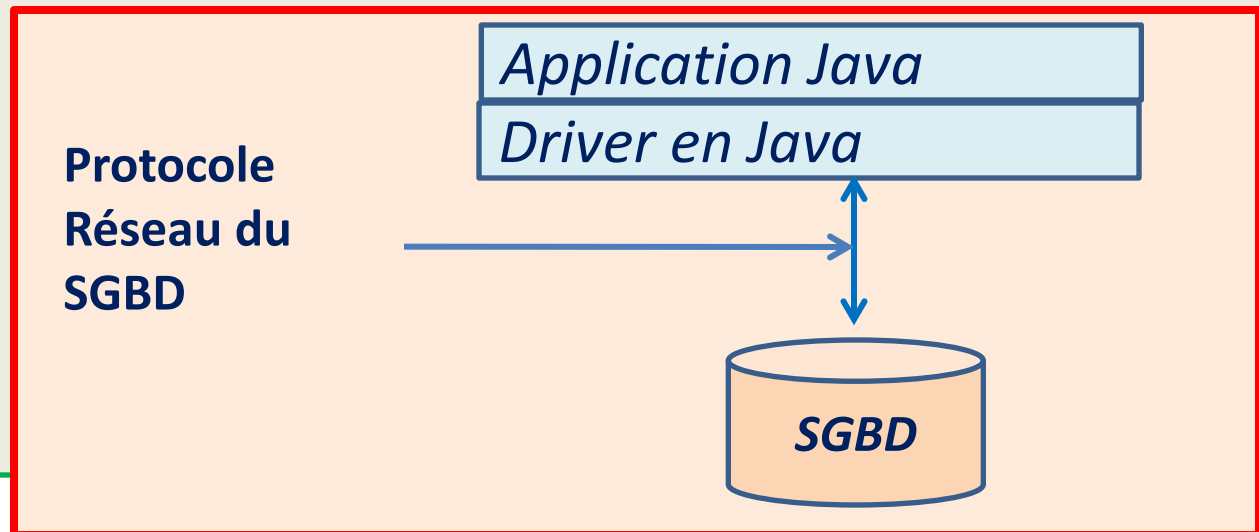
Type IV: drivers 100% Java

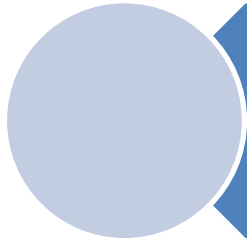
Ces drivers utilisent un protocole réseau propriétaire spécifique au SGBD.

- 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 assurée pour l'utilisation des *sockets*: une *applet* ne peut ouvrir une *connexion* que sur la machine où elle est hébergée.





Chargement du pilote

Un programme JDBC débute toujours par le chargement du pilote approprié pour la BD.

Mais puisque le programme a la possibilité d'accéder à plusieurs types de BD, il peut avoir plusieurs pilotes.

C'est au moment de la connexion à la BD que le **DriverManager** choisit alors le *bon pilote*.

DriverManager = gestionnaire de tous les drivers chargés par un programme java.

Les URLs JDBC

(Localisation et accès à la BD: établissement d'une connexion avec la BD)

Après le chargement du driver, il faut localiser la BD en spécifiant son emplacement.

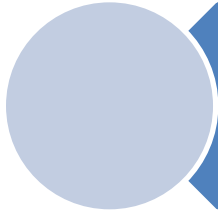
Pour chaque type de driver il existe un schéma de désignation de la base à atteindre avec JDBC:

Type I: ***jdbc:odbc:source*** *source = source de données ODBC.*
 (ex: jdbc: odbc: employe)

Type II: ***jdbc:protocole***
 où protocole = protocole spécifique et utilisant des méthodes natives.

Type III et IV: ***jdbc:driver: adresse***
 ex: jdbc:mysql ://elios.lip6.fr/employe
 jdbc:oracle: thin :@elios.lip6.fr:employe

Chaque URL est de la forme: jdbc:sous-protocole:base_de_donnée



Structure d'un programme JDBC (1/4)

(différente étapes lors de l'utilisation de JDBC)

1. **Chargement du driver** (= chargement de la classe du driver dans la JVM)

Class.forName (String driverName);

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

```
Class.forName ("oracle.jdbc.driver.OracleDriver") ;
```

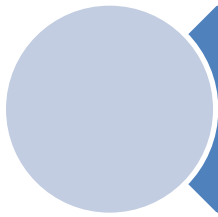
```
Class.forName ("org.gjt.mm.mysql.Driver") ;
```

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

Certains compilateurs refusent la notation précédente et demandent :

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

Cette étape 1 constitue *l'enregistrement du driver JDBC*.



Structure d'un programme JDBC (2/4)

2. Établir (ouverture) une connexion à la base de données

Une fois le driver enregistré, une connexion peut être établie avec la BD.

Pour obtenir une connexion à un SGBD, il faut faire une demande à la classe gestionnaire de drivers:

Demande permise par la méthode `getConnection (...)` de la classe **DriverManager**

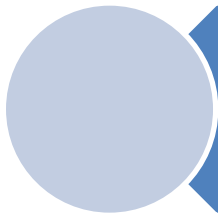
Cette méthode peut prendre 3 arguments au plus:

- *l' URL vers la BD*
- *le nom de l'utilisateur de la base*
- *son mot de passe*

cette méthode renvoie un objet de type **Connection**.

Connection con =

```
DriverManager.getConnection ("jdbc:odbc:employe", "login", "passwd");
```



Structure d'un programme JDBC (3/4)

3. Création de requêtes SQL en utilisant l'objet de type **Connection** (étape 2).

dans cette étape on crée en fait une zone de description de requêtes c'est-à-dire un espace où l'on pourra exécuter des opérations SQL.

Cette zone est un objet de la classe **Statement** que l'on crée par la méthode **createStatement ()**.

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

Il existe trois types de Statement:

Statement: requêtes statiques simples,

PreparedStatement: requêtes dynamiques pré-compilées (avec paramètres d'I/O),

CallableStatement: procédures stockées.

Les deux derniers seront développés plus tard dans ce cours.

Structure d'un programme JDBC (4/4)

4. Envoi et exécution de requêtes.

Il existe trois types d'exécution de requêtes:

- `executeQuery (...)`: pour les requêtes SELECT qui retournent un **ResultSet** (tuples),
- `executeUpdate (...)`: pour les requêtes INSERT, UPDATE, DELETE, CREATE TABLE, DROP TABLE qui retournent un entier (**int**) désignant le nombre de tuples traités.
- `execute ()`: pour les procédures stockées (cas rares).

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

```
Statement st = con.createStatement ( );  
ResultSet rs = st.executeQuery ("select * from Client");
```



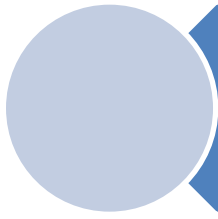

Notes 1

Le code SQL n'est pas interprété par Java.

- c'est le pilote associé à la connexion (et au finish par le moteur de la BD) 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 premier accès à la BD pour découvrir les types des colonnes impliquées dans la requête puis un deuxième pour l'exécuter.



Notes 2: Traitement des données retournées

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

La méthode **next ()** de **ResultSet** permet de parcourir itérativement ligne par ligne l'ensemble des tuples sélectionnés.

Cette méthode :

- retourne **false** si le dernier tuple est 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 tuple par tuple }
```

Impossible de revenir au tuple précédent ou de parcourir l'ensemble dans un ordre quelconque.



Notes 2: Traitement des données retournées

La méthode **next ()** permet de parcourir le **ResultSet** du premier au dernier enregistrement.

Mais il existe d'autres façons aussi de parcourir un **ResultSet**.

On peut parcourir le **ResultSet** ligne par ligne de façon itérative de la fin vers le début:

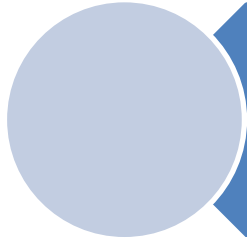
- utilisez pour cela la méthode **previous ()**, [**while (rs.previous ()) { // ...}**]

En déplacement absolu: on peut aller *exactement* à la nième ligne:

- utilisez alors la méthode **absolute (int row)**, [**while (rs.absolute (i)) { // ...}**]
vous pourrez faire usage des méthodes **first ()**, **last ()**,...

En déplacement relatif: on peut aller à la nième ligne à partir de la position courante du curseur:

- utilisez la méthode **relative (int row)** et pour accéder à un enregistrement les méthodes **afterLast ()**, **beforeFirst ()**, ...



Notes 3: Traitement des données retournées

Les colonnes d'une table de la BD sont référencées par leur **numéro** (commençant par 1) ou par leur **nom**.

L'accès aux valeurs des colonnes se fait par des méthodes de la forme **getXXX (...)** permettant la lecture de données du type XXX dans chaque colonne du tuple courant.

```
int val = rs.getInt (3) ; // acces à la 3e colonne
```

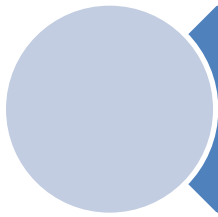
```
String prod = rs.getString ("Produit") ; // accès à la colonne de nom Produit
```



Premier exemple complet d'utilisation de JDBC (1/3)

(exemple avec une source de données ODBC pour MS ACCESS)

```
import java.sql.*;
public class TestJDBC {
public static void main (String args) {
/** chargement du Driver ODBCJDBC*/
try {
    Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
}
catch (Exception e) {
    System.out.println ("Erreur dans le chargement du driver"); System.exit ( 0 );
}
/**Connexion à la base*/
Connection con = null ;
try {
    con = DriverManager.getConnection ("jdbc: odbc: employe", "dba", "sql");
}
catch (SQLException e) {
    System.out.println ("Impossible de se connecter à la BD"); }
```



Premier exemple complet d'utilisation de JDBC (2/3)

(exemple avec une source de données ODBC pour MS ACCESS)

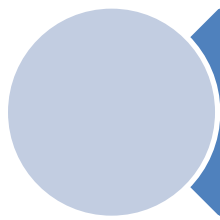
```
/**création d'une zone d'exécution de requêtes SQL*/  
Statement stmt = null;  
try {  
    stmt = con.createStatement ( );  
}  
catch (SQLException e) {  
    System.out.println("Impossible de créer de Statement ");  
}  
/**exécution de requêtes et récupération des données demandées*/  
ResultSet rs = null ;  
try { rs = stmt.executeQuery ("SELECT * FROM CLIENT ");  
}  
catch (SQLException e) {  
    System.out.println(" Erreur de requête SQL ");  
}
```



Premier exemple complet d'utilisation de JDBC (3/3)

(exemple avec une source de données ODBC pour MS ACCESS)

```
/**parcours du résultat: affichage des données lues*/  
while (rs.next ( )) {  
    String prenom = rs.getString ("prenom") ;  
    int age = rs.getInt ("age") ;  
    System.out.println (prenom+" a "+age+" ans");  
}  
/** fermeture de tout lien avec la BD*/  
    rs.close ( );  
    stmt.close ( );  
    con.close ( );  
}  
} // fin de la classe
```

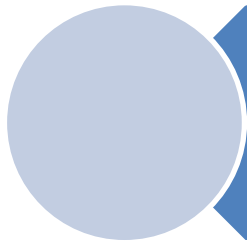


Correspondances Types données SQL/Java

Si vous écrivez une instruction telle que `String prenom = rs.getString ("prenom") ;` cela signifie que vous êtes convaincu que la colonne de nom **PRENOM** a été créée sous SQL avec le type *VARCHAR*. *Autrement cette instruction causerais une erreur de runtime.*

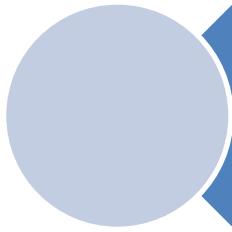
Donc il est bon de savoir la correspondance entre types SQL et types Java pour pouvoir manipuler convenablement les données lues.

Types SQL	Types Java	Méthodes
CHAR/ VARCHAR	String	getString ()
INTEGER	int	getInt ()
TINYINT	byte	getBytes ()
SMALLINT	short	getShort ()



Correspondances Types données SQL/Java

Types SQL	Types Java	Méthodes
BIGINT	long	getLong ()
bit	Boolean	getBoolean ()
REAL	float	getFloat ()
NUMERIC	java.math.BigDecimal	getBigDecimal ()
DECIMAL	java.sql.Date	getDate ()
TIME	java.sql.Time	getTime ()
TIMESTAMP	java.sql.TimeStamp	getTimeStamp ()
FLOAT DOUBLE	double	getDouble ()



Second exemple complet d'utilisation de JDBC (1/3)

(utilisation de JDBC pour MySQL)

Pour travailler avec JDBC et MySQL, il faut commencer par installer le driver pour une base de données MySQL.

Ce driver se nomme **Connector/J**. Vous pouvez le récupérer à l'adresse <http://www.mysql.com/products/connector-j>

Ensuite décompressez le fichier **tar.gz** ou le **zip**.

Et maintenant effectuez l'une des opérations suivantes:

- copier le fichier jar (**mysql-connector-java-3.1.10-bin.jar**) dans l'un des répertoires de votre variable CLASSPATH
- ajouter le répertoire contenant le fichier **jar** à votre CLASSPATH
- copier le fichier jar dans **\$JAVA_HOME/jre /lib/ext**.

Et enfin dans le programme créez une librairie qui encapsule le fichier jar.
Pour tester, vous pouvez créer le programme suivant:



Second exemple complet d'utilisation de JDBC (2/3)

(utilisation de JDBC pour MySQL)

```
import java.sql.*;
public class TestJavaMySQL {
static Connection con;
static Statement st;
static ResultSet rs;
public static void main (String[] args) {
try { /*chargement du driver*/
Class.forName("com.mysql.jdbc.Driver").newInstance ( );
}
catch (Exception e){System.out .println("Erreur driver: "+e.getMessage() ) ;}
```

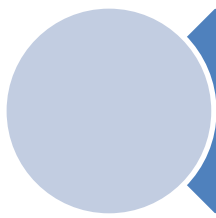
NB: vous pouviez aussi prendre comme driver :
Class.forName("org.gjt.mm.mysql.Driver").newInstance() ;



Second exemple complet d'utilisation de JDBC (3/3)

(utilisation de JDBC pour MySQL)

```
try {con = DriverManager.getConnection ("jdbc:mysql: //localhost /Employe","root","") ;  
}  
catch (Exception ez ){System.out.println("Erreur de connexion "+ ez.getMessage ( ));}  
try { st = con.createStatement() ;  
}  
catch (SQLException t){System.out.println ("Erreur de Statement "+t.getMessage());}  
try {  
rs = st.executeQuery("select * from client") ;  
while (rs.next() )  
{ System.out .println(rs.getObject (1)+" "+rs.getObject(2)+" "+rs.getObject(3) ) ;  
}  
}  
catch (Exception er) {System.out .println("Erreur ResultSet "+er.getMessage ( ) ) ; }  
try { rs.close ( ) ; st.close ( ) ; con.close ( ) ;  
}  
catch (Exception d) { }  
}}
```



L'interface java.sql.PreparedStatement

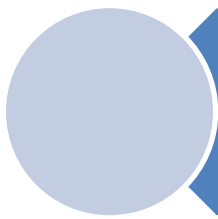
*(pour l'utilisation efficace des requêtes **pré-compilées: SQL DYNAMIQUE**)*

Parfois, il est très utile et plus efficace de faire usage de l'objet **PreparedStatement** pour envoyer une instruction SQL à la base de données. En effet, si vous voulez *exécuter un objet **Statement*** plusieurs fois, le temps d'exécution sera réduit si vous utilisez plutôt on objet **PreparedStatement**.

La fonctionnalité d'un objet **PreparedStatement**, contrairement à l'objet **Statement**, est de lui fournir une instruction SQL dès sa création. **L'avantage est que l'instruction SQL sera directement envoyée au SGBD, où elle sera compilée.**

Ainsi l'objet **PreparedStatement** ne contient plus seulement une instruction SQL, mais bien *une instruction SQL pré compilée*.

Cela signifie donc que quand le **PreparedStatement** est exécuté, le SGBD a juste à lancer l'instruction SQL sans avoir à le compiler à nouveau.



L'interface java.sql.PreparedStatement

Comment créer un objet PreparedStatement ?

Un objet de type **Connection** est nécessaire lors de la création d'un objet **PreparedStatement**. Cet objet appelle la méthode **prepareStatement** de la même classe

```
PreparedStatement pst = objet_conn.prepareStatement ("UPDATE Compte  
SET solde = ? WHERE numC = ? ) ;
```

Tout objet **PreparedStatement** est caractérisé par des paramètres indispensables représentés par les point d'interrogation (?).

La transmission de ces paramètres est réalisée par les méthodes **setXXX (...)** qui permettent de fournir des valeurs qui vont être utilisées à la place des ?.

NB: dans setXXX (...), XXX désigne n'importe quel type Java.

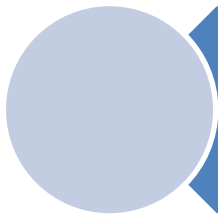
Après création, on peut transmettre effectivement des valeurs à l'objet pst :

```
pst.setDouble (1,20000); // transmission de la valeur 20000 au premier ?
```

```
pst.setInt (2, 703) ; // transmission de la valeur 703 au second ?
```

```
pst.executeUpdate ( ); // exécution maintenant de la requête
```

```
pst.close ( );
```



Exemple complet de PreparedStatement

```
import java.sql.*;

public class TestPreStat {
    public static void main (String[] args) {
        try { Class.forName ("com.mysql.jdbc.Driver").newInstance() ;
            Connection con=
            DriverManager.getConnection ("jdbc:mysql ://127.0.0.1/Employee","root","") ;
            PreparedStatement p =
            con.prepareStatement ("UPDATE compte SET solde =? where numC =?") ;
            p.setDouble (1,200000) ;
            p.setInt (2,702) ;
            p.setDouble (1,500000) ;
            p.setInt (2,703) ;
            p.executeUpdate ( ) ;
            p.close ( ) ;
            con.close ();
        }
        catch (Exception er) { er.printStackTrace ( ) ; } }
```



Notes sur le PreparedStatement

Si vous réalisez des opérations de **mise à jour** (ex UPDATE) sur l'objet *PreparedStatement*, utilisez la méthode `executeUpdate()` pour effectuer la transaction dans la base.

Si vous réalisez des opérations de **sélection** (SELECT) , utilisez la méthode `executeQuery ()` pour effectuer la transaction dans la base.

Quelques fois, le type de la requête est indéfinie à priori (MàJ ou select): utilisez dans ce cas la méthode `execute ()` qui renvoie un booléen (**true** si c'est une requête SELECT, **false** dans les cas de MàJ).

ATTENTION: les méthodes `executeUpdate ()` et `executeQuery ()` ne prennent aucun paramètre. C'est évident puisque la requête est déjà transmise à l'objet *PreparedStatement*.

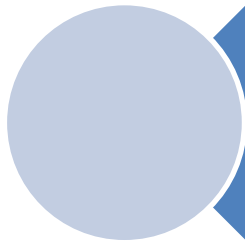


Les mises à jour de masse (Batch Updates)

*JDBC 2.0 permet de réaliser des **mises à jour** de masse en regroupant plusieurs traitements pour les envoyer en une seule fois au SGBD. Ceci permet d'améliorer les performances surtout si le nombre de traitements est important.*

Cette fonctionnalité n'est pas obligatoirement supportée par le pilote. La méthode `supportsBatchUpdate ()` de la classe **DatabaseMetaData** permet de savoir si elle est utilisable avec le pilote. Elle renvoie un *boolean*.

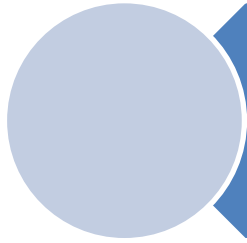
Plusieurs méthodes ont été ajoutées à l'interface **Statement** pour pouvoir utiliser les mises à jour de masse :



Les mises à jour de masse (Batch Updates)

Méthodes	Rôle
<code>void addBatch (String)</code>	permet d'ajouter une chaîne contenant une requête SQL
<code>int [] executeBatch()</code>	permet d'exécuter toutes les requêtes. Elle renvoie un tableau d'entier qui contient pour chaque requête, le nombre de mises à jour effectuées.
<code>void clearBatch ()</code>	supprime toutes les requêtes stockées

Lors de l'utilisation de **batch update**, il est préférable de positionner l'attribut **autocommit** à **false** afin de faciliter la gestion des transactions et le traitement d'une erreur dans l'exécution d'un ou plusieurs traitements.

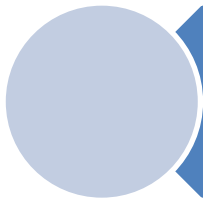


Les mises à jour de masse (Batch Updates) Gestion des exceptions

Une exception particulière peut être levée en plus de l'exception **SQLException** lors de l'exécution d'une mise à jour de masse. L'exception **SQLException** est levée si une requête SQL d'interrogation doit être exécutée (requête de type SELECT).

L'exception **BatchUpdateException** est levée si une des requêtes de mise à jour échoue.

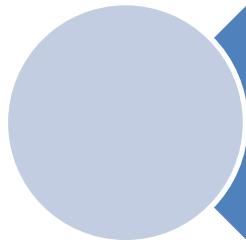
L'exception **BatchUpdateException** possède une méthode **getUpdateCounts ()** qui renvoie un tableau d'entier qui contient *le nombre d'occurrences impactées par chaque requête réussie*.



Les mises à jour de masse (Batch Updates):Exemple

Touts les comptes dont le decouvert est supérieur à 30000 sont mis à jour et le solde devient 500000.

```
public class TestBatchUpdate {
public static void main(String [ ] args) {
try { Class.forName ("org.gjt.mm.mysql.Driver").newInstance ( ) ;
}
catch (Exception e) { }
try {
Connection con = DriverManager.getConnection("jdbc: mysql://localhost
/banque","root","");
con.setAutoCommit (false);
Statement st = con.createStatement ( );
st.addBatch ("UPDATE compte SET solde = 500000 where decouvert >= 30000 " );
st.executeBatch ( );
st.close ( ) ;
con.close ( ) ;
}
catch(Exception er){ er.printStackTrace ( ) ;}
}}
```



Les MétaDonnées

On peut avoir besoin quelque fois des informations **sur la structure et le schéma** de la base de données elle-même.

Ces informations sont appelées des Métadonnées.

Pour obtenir ces renseignements, il faut utiliser un objet de la classe

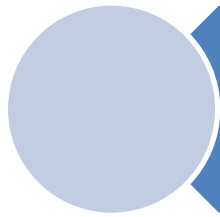
DatabaseMetaData à partir de l'objet **Connection**. La méthode **getMetaData ()** de la classe **Connection** permet de créer cet objet.

```
DatabaseMetaData donneesBD = objet_con.getMetaData ( ) ;
```

A partir de cet objet, cette classe fournit beaucoup de renseignements sur la BD.

Par exemple:

```
DatabaseMetaData d = con.getMetaData ( ) ;  
String nomSGBD = d.getDatabaseProductName ( ) ; // nom du SGBD  
String versionDriver = d.getDriverVersion ( ) ; // version du driver  
String nomDriver = d.getDriverName ( ) ; //nom du driver
```



Les MétaDonnées

On peut aussi avoir des informations sur les objets **ResultSet** générés:

- nombre de colonnes contenues dans le ResultSet,
- noms des colonnes
- types des colonnes,
- etc.

Pour récupérer ces Métadonnées, il faut créer un objet de la classe **ResultSetMetaData** grâce à la méthode **getMetaData ()** de la classe **ResultSet**.

```
ResultSetMetaData rsmd = objet_resultset.getMetaData ( );
```

```
int columnCount = rsmd.getColumnCount ( ); //nombres de colonnes
```

```
String columLabel = rsmd.getColumnLabel ( i ); // nom de la colonne i
```

```
String columnType = rsmd.getColumnTypeName ( i ); // classe de la colonne i
```

NB: ces informations sont très utiles surtout lors de la création de modèles de table pour le chargement d'objets **JTable**.

