JDBC

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

En fait cette API est une spécification de ce que doit implanter un constructeur de BD pour que celle ci soit interrogeable par JDBC. De ce fait dans la programmation JDBC on utilise essentiellement des références d'interface (`Connection`, `Statement`, `ResultSet`, ...).

Sun et les constructeurs de BD se sont charger de fournir (vendre ou donner) des classes qui implémentent les interfaces précitées qui permettent de soumettre des requêtes SQL et de récupérer le résultat.

Par exemple Oracle fournit une classe qui, lorsqu'on écrit
```
Statement stmt =
conn.createStatement();
```
retourne un objet concret (de classe `class OracleStatement implements Statement` par exemple) qui est repéré par la référence `stmt` de l'interface `Statement`.

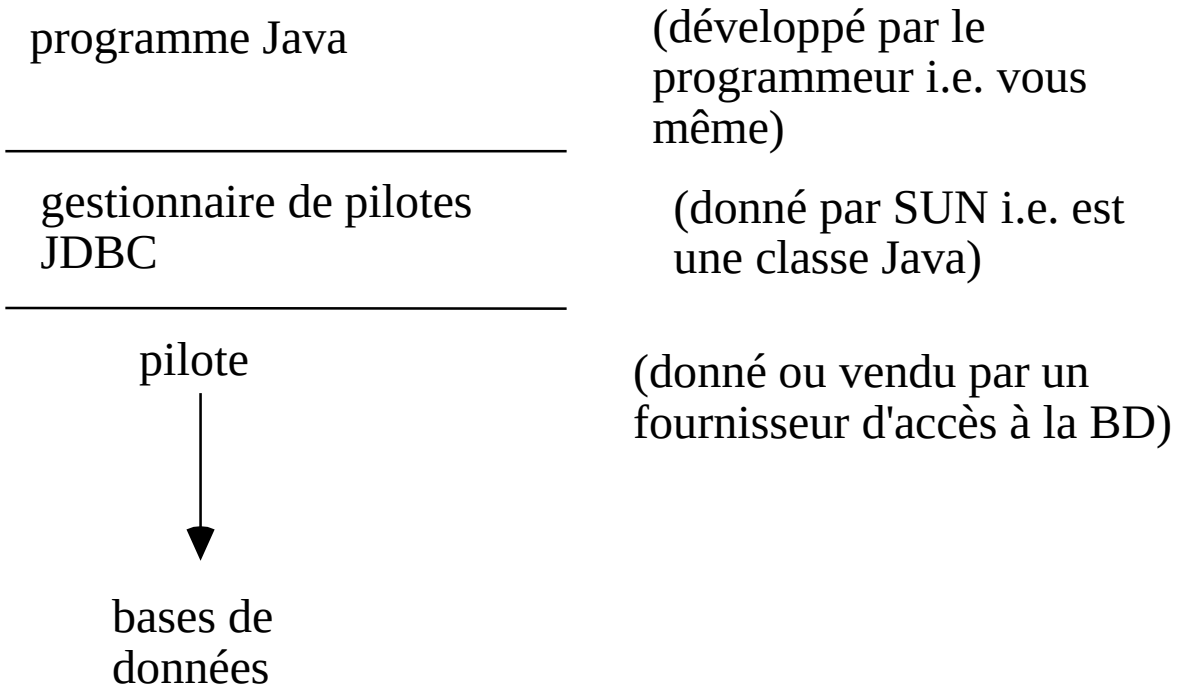
Pilotes (drivers)

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

Parmi les interfaces, l'une d'entre elles, l'interface `Driver`, décrit ce que doit faire tout objet d'une classe qui implémente l'essai de connexion à une base de données. Entre autre, un tel objet doit obligatoirement s'enregistrer auprès du `DriverManager` et retourner en cas de succès un objet d'une classe qui implémente l'interface `Connection`.

Ces pilotes (drivers) sont dits jdbc-baseAAtteindre.
Le premier pilote (développé par SUN) est un pilote jdbc:odbc.
La liste des pilotes disponibles se trouve à :
`http://industry.java.sun.com/products/jdbc/drivers`

Architecture JDBC



Les 4 types de pilotes

Les pilotes sont classés en quatre types :

Ceux dits natifs qui utilisent une partie écrite dans du code spécifique non Java (souvent en langage C) et appelé par ces implantations : les pilotes de classe 1 et 2. Ces pilotes sont rapides mais doivent être présent sur le poste client car ne peuvent pas être téléchargés par le ClassLoader de Java (ce ne sont pas des classes Java mais plutôt des librairies dynamiques). Ils ne peuvent donc pas être utilisés par des applets dans des browsers classiques.

Ceux dits 100% Java qui interrogent le gestionnaire de base de données avec du code uniquement écrits en Java : les pilotes de classe 3 et 4.
Ces pilotes peuvent alors être utilisés par des applets dans des browsers classiques.
Plus précisément :
pilote de classe 1 : pilote jdbc:odbc
pilote de classe 2 : pilote jdbc:protocole spécifique et utilisant des méthodes natives.
pilote de classe 3 : pilote écrit en Java jdbc vers un middleware qui fait l'interface avec la BD
pilote de classe 4 : pilote écrit en Java jdbc:protocole de la BD. Accède directement à l'interface réseau de la BD.

Structure d'un programme JDBC

Un code JDBC est de la forme :
- recherche et chargement du driver approprié à la BD.
- établissement de la connexion à la base de données.
- construction de la requête SQL
- envoi de cette requête et récupération des réponses
- parcours des réponses.

Syntaxe

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection conX = DriverManager.getConnection(...);
Statement stmt = conX.createStatement();
ResultSet rs = stmt.executeQuery("SELECT a, b, c ... FROM ...
WHERE ...");

while (rs.next()) {
...
// traitement
}
```

La programmation avec JDBC

On utilise le paquetage `java.sql`. La plupart des méthodes lèvent l'exception `java.sql.SQLException`.

Les produits JDBC

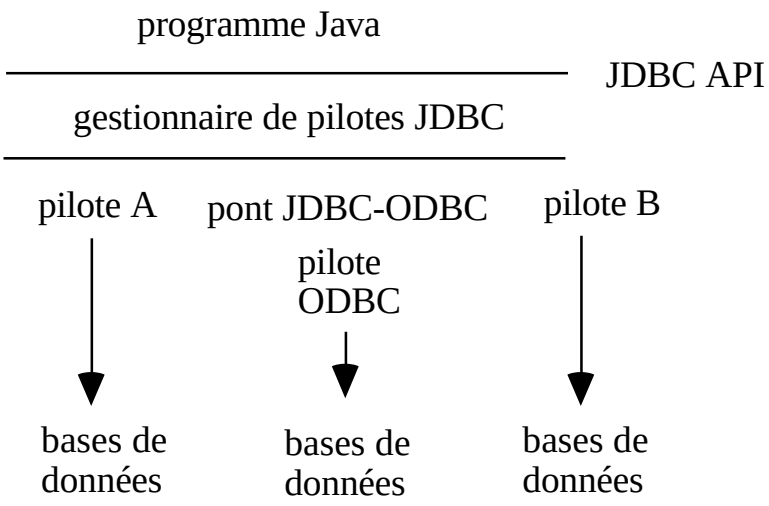
Javasoft fournit 3 produits composants intégré dans le JDK (1.1 et suivant) :

- un gestionnaire de pilotes JDBC. Le gestionnaire de pilotes JDBC permet de connecter les programmes Java au bon pilote JDBC.
- un pont JDBC-ODBC
- un ensemble de tests pour les pilotes JDBC. Voir à `http://www.javasoft.com:80/products/jdbc/download2.html` la partie "The JDBC Driver Test Suite".

Les tests pour les pilotes JDBC vérifient que le pilote implémente la "norme" JDBC COMPLIANT et l'implémentation de toutes les classes et méthodes JDBC ainsi que les fonctionnalités ANSI SQL-2 Entry Level (norme ANSI 1992)

Chargement du pilote

On commence un programme JDBC en chargeant dans le programme, le pilote approprié pour la BD. Comme le programme peut interroger divers types de BD il peut avoir plusieurs pilotes. C'est au moment de la connexion que sera choisi le bon pilote par le `DriverManager`.
On a donc une architecture :



Syntaxe :

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

ou encore

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

Chargement du pilote (suite)

En fait cette instruction construit un objet de la classe `Class` qui modélise la classe dont le nom est l'argument de la méthode statique `forName(...)`. Cette classe est un driver c'est à dire ici une classe qui implémente l'interface `Driver`. Dans les spécifications de JDBC il est dit qu'obligatoirement ces classes doivent, au moment de leur premier chargement :

- créer une instance de cette classe
- enregistrer cette instance au sein du `DriverManager` par la méthode statique `registerDriver(...)` de cette classe.

voir à

`http://java.sun.com/products/jdk/1.1/docs/guide/jdbc/getstart/drivermanager.doc.html#999050`

"All Driver classes should be written with a static section that creates an instance of the class and then registers it with the DriverManager class when it is loaded. Thus, a user would not normally call DriverManager.registerDriver directly; it should be called automatically by a driver when it is loaded. A Driver class is loaded, and therefore automatically registered with the DriverManager"

Chargement du pilote (fin)

Exercice : concrétement comment cela est il écrit ?
réponse :
Quand vous écrivez une classe `MonDriver` implémentant l'interface `Driver` cette classe doit contenir le bloc static suivant

```
static {
 DriverManager.registerDriver(new MonDriver());
}
```

La connexion

On ouvre une connexion avec une des méthodes `DriverManager.getConnection(...)` qui retourne un objet d'une classe qui implémente l'interface `Connection`. Ces méthodes contiennent comme premier argument une "URL JDBC". Elles recherchent le pilote adapté pour gérer la connexion avec la base repérée par cette URL.

Une URL JDBC doit commencer par `jdbc`. Le second argument est le protocole sous jacent dans lequel le pilote traduit. Le troisième argument est un identificateur de base de données. La syntaxe d'une URL JDBC est donc:

`jdbc:`<*sous-protocole*>`:`<*baseID*>

la partie *baseID* est propre au *sous-protocole*.

Exemples d'URL JDBC :

```
jdbc:odbc:test1
```

ou encore

```
jdbc:odbc:@orsay:1751:test1
jdbc:oracle:thin:@orsay:1751:test1
```

Les arguments suivants de `Driver.getConnection(...)` sont des informations nécessaires à l'ouverture de la connexion souvent un nom de connexion à la base suivi d'un mot de passe.

```
Connection conX = DriverManager.getConnection(URLjdbc,
"dbUser1", "pwuser1");
```

La classe `DriverManager`

Cette classe est une classe qui ne contient que des méthodes statiques.
Elle fournit des méthodes qui sont des utilitaires pour gérer l'accès aux bases de données par Java et les différents drivers JDBC à l'intèrieur d'un programme Java.

Finalement on ne créée ni ne récupère d'objet de cette classe.

Requêtes au bases de données

Sans trop rentrer dans les détails (voir un cours de bases de données pour cela), il existe différentes sortes de requêtes SQL dont :
- les requêtes SELECT qui opèrent sur une ou plusieurs tables et placent le résultat dans une "table" résultante.
- les requêtes d'action qui modifient une ou des colonnes ou lignes de table. On a par exemple comme requête d'action UPDATE, DELETE, APPEND.
- des SQL DDL (Data Definition Language) comme CREATE TABLE, DROP TABLE, ...

SQL est un langage non sensible à la casse (culture IBM-SQL et non pas culture Unix-C). En général les mots clés du langage SQL sont mis en majuscules.

Syntaxiquement en Java on utilise `executeQuery(...)` si la requête est une requête SELECT et `executeUpdate(...)` si a requête est une requête d'action ou une SQL DDL.

Exécution d'instructions SQL

Toutes les instructions ANSI SQL-2 Entry Level sont acceptés puisque le driver `jdbc:protocole SGBD sous-jacent` a été testé pour cela.

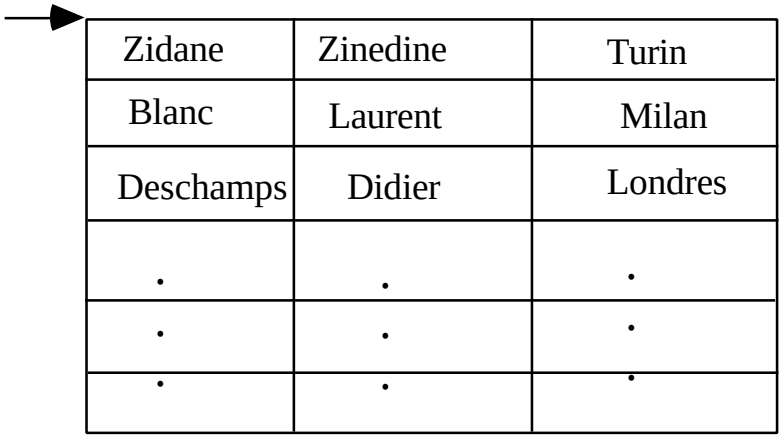
On utilise l'un des trois interfaces `Statement`, `PreparedStatement`, `CallableStatement`. Les instructions `PreparedStatement` sont précompilées. On utilise `CallableStatement` pour lancer une procédure du SGBD.

Avec un `Statement` on a :

```
Statement smt = conX.createStatement();
ResultSet rs = smt.executeQuery( "SELECT * FROM Livres" );
```

Récupération des résultats (suite)

Les résultats des requêtes SELECT ont été mis dans un `ResultSet`. Cet objet récupéré modélise le résultat qui peut être vu comme une table. Par exemple
SELECT nom, prenom, adresse FROM Personnes
retourne un `ResultSet` qui modélise :



Zidane	Zinedine	Turin
Blanc	Laurent	Milan
Deschamps	Didier	Londres
.	.	.
.	.	.
.	.	.

Un pointeur géré par Java jdbc permet de parcourir tout le `ResultSet` et est initialisé : il est automatiquement positionné avant la première ligne de résultat.

On parcourt alors tout le `ResultSet` pour avoir l'ensemble des réponses à la requête. La boucle de parcours est :

```
while( rs.next() )  {
// traitement
}
```

Récupération des résultats (fin)

Les colonnes demandées par la requête sont numérotées à partir de 1 (culture IBM-SQL et non pas culture Unix-C).

Remarque
La numérotation est relative à l'ordre des champs de la requête et non pas l'ordre des champs de la table interrogée (évidemment).

De plus les colonnes sont typées en type SQL. Pour récupérer une valeur de colonne il faut donc indiquer le numéro de la colonne ou son nom et faire la conversion de type approprié. Par exemple :

```
Statement stmt = ...;
ResultSet rs = stmt.executeQuery(SELECT nom, prenom, age,
date FROM LaTable");
String leNom = rs.getString( 1 ) );
String lePrenom = rs.getString( 2 );
int lage =  rs.getInt ( 3 );
Date laDate = rs.getDate( 4);
```

On peut aussi désigner les colonnes par leur nom dans la BD (c'est moins rapide mais plus lisible) et réécrire :

```
ResultSet rs = stmt.executeQuery("SELECT nom, prenom,
age, date FROM LaTable");
String leNom = rs.getString( "nom") );
String lePrenom = rs.getString( "prenom" );
int lage =  rs.getInt ( "age" );
Date laDate = rs.getDate( "date ");
```

Correspondance type SQL-type Java

Les correspondances entre type SQL et type Java sont données par les spécifications JDBC. En voici certaines :

SQL type	Java Type
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

Conclusion : finalement comment cela fonctionne ?

Après l'instruction

```
Connection conX = DriveManager.getConnection(URLjdbc,
"dbUser1", "pwuser1");
```

un pilote à la base de données a été trouvé par le programme (et la classe `DriverManager`) parmi les nombreux pilotes qui ont pu être chargés à la suite d'instructions

```
Class.forName("piloteToBD");
```

Ce pilote a été capable de faire une connexion à la base et de retourner un objet d'une classe qui implémente l'interface `Connection`. La classe est peut-être `OracleConnection` ou `SUN_JDBC_ODBC_Connection`, peu importe l'essentiel est que cette classe implémente cette interface et qu'on puisse alors repéré cet objet par une référence d'interface `Connection` (polymorphisme !!). Voila pourquoi le code que nous écrivons est universel et permet d'accéder à tout type de base.

Le fonctionnement (suite)

Par la suite cet objet repéré par `conX` va retourner, à la suite de l'instruction,

```
Statement smt = conX.createStatement();
```

un objet d'une classe qui implémente l'interface `Statement` et qui est propre à la base (par exemple `OracleStatement` ou `SUN_JDBC_ODBC_Statement`, peu importe l'essentiel est que cette classe implémente l'interface `Statement` et qu'on puisse alors repérer cet objet par une référence d'interface `Statement` air connu).

Enfin l'instruction

```
ResultSet rs = smt.executeQuery( "SELECT * FROM Livres" );
```

demande le lancement de la méthode `executeQuery(...)` sur un objet d'une classe qui implémente `Statement` et qui est propre à la base et cela fonctionne et est universel (pour la troisième fois ;-)).

PreparedStatement

Lors de l'envoi d'une requête pour exécution 4 étapes doivent être faites :
- analyse de la requête
- compilation de la requête
- optimisation de la requête
- exécution de la requête

et ceci même si cette requête est la même que la précédente !! Or les 3 premières étapes ont déja été effectuées dans ce cas.

Les bases de données définissent la notion de requête préparée, requête où les 3 premieres étapes ne sont effectuées qu'une seule fois. JDBC propose l'interface `PreparedStatement` pour modéliser cette notion. Cette interface dérive de l'interface `Statement`.
De plus on ne peut pas avec un `Statement` construire des requêtes où un des arguments est une variable i.e. requêtes paramétrées. Il faut pour cela utiliser un `PreparedStatement`.

`PreparedStatement` : Syntaxe

Même sans paramêtres, les `PreparedStatement` ne s'utilisent pas comme des `Statement`.
Au lieu d'écrire :

```
Statement smt = conX.createStatement();
ResultSet rs = smt.executeQuery( "SELECT * FROM Livres" );
```

on écrit :

```
PreparedStatement pSmt = conX.prepareStatement("SELECT
* FROM Livres" );
ResultSet rs = pSmt.executeQuery();
```

à savoir la requête est décrite au moment de la "construction" pas lors de l'exécution qui est lancée par `executeQuery()` sans argument.

Requêtes paramétrées

On utilise donc les `PreparedStatement` et on écrit des requêtes de la forme :

```
SELECT nom FROM Personnes WHERE age > ?
AND adresse = ?
```

Puis on utilise les méthodes
`set`*`Type`*`(numéroDeLArgument, valeur)`
pour positionner chacun des arguments.
Les numéros commencent à 1 dans l'ordre d'apparition dans la requête.
On a donc un code comme :

```
PreparedStatement pSmt = conX.prepareStatement("SELECT
nom FROM Personnes WHERE age > ? AND adresse = ?" );
pSmt .setInt(1, 22);
pSmt .setString(2, "Turin");
ResultSet rs = pSmt.executeQuery();
```

CallableStatement

Certaines requêtes pour une base de données sont si courantes qu'elles sont intégrées dans la base de données. On les appelle des procédures stockées et elles ont modélisées en JDBC par l'interface `CallableStatement` qui dérive de `PreparedStatement`. Elle peut donc avoir des paramétres.

On lance l'éxécution d'une procédure stockée à l'aide de la syntaxe :

```
{call nom_Procedure[(?, ?, ...)]}
```

([ ] signifiant optionnel)

Pour une procédure retournant un résultat on a la syntaxe :

```
{? = call nom_Procedure[(?, ?, ...)]}
```

La syntaxe pour une procédure stockée sans paramêtres est :

```
{call nom_Procedure}
```

Utilisation :

```
CallableStatement cStmt =
 conX.prepareCall("{call sp_setDomicile(?, ?) }");
cStmt.setString(1, "Djorkaeff");
cStmt.setString(2, "KaisersLautern");
cSmt.executeUpdate();
```

les MetaData

On peut avoir sur une BD, des informations sur la BD elle même. Ces informations sont appelées des métadatas. On obtient un objet de la classe `DatabaseMetaData` à partir de la connexion par :

```
conX = DriverManager.getConnection(dbURL, "u1db1",
"u1db1");
DatabaseMetaData dmd = conX.getMetaData();
```

À partir de là, cette classe fournit beaucoup de renseignements sur la base par exemple :

```
String nomSGBD = dmd.getDatabaseProductName();
```

De même on peut avoir des métadatas sur un `ResultSet` :

```
//Affichage des colonnes de la table
rs = smt.executeQuery( "SELECT * FROM Livres " );

ResultSetMetaData rsmd = rs.getMetaData();

int numCols = rsmd.getColumnCount();
for ( int i = 1; i <= numCols; i++ )
 System.out.println( "\t" + i + " : " +
        rsmd.getColumnLabel( i ) );
```

JDBC 2.0

Des ajouts ont été faits aux spécifications JDBC 1.0 et beaucoup d'interfaces et de classes de java.sql ont été enrichis. De plus certains ajouts sont implantés dans le paquetage `javax.sql`.
Les ajouts de JDBC 2.0 concernent :
- des nouvelles possibilités de traiter les `ResultSet` (autrement que séquenciellement, ...)
- le traitement par batch
- des types de données avancés
- la notion d'ensemble de lignes manipulables même sans connexion maintenue ouverte : les `RowSet`.
- l'utilisation du service de nommage universel de Java, JNDI
- l'utilisation du service de transaction de Java JTS.

Traitement avancé des `ResultSet`

Essentiellement JDBC 2.0 propose sur les `ResultSet` des déplacements (scrolling) et des mises à jour (updatability). Les déplacements peuvent être relatifs ou absolus.

Les `ResultSet` sont désormais de 3 "types" : - forward-only
- scroll-insensitive. Le `ResultSet` admet les déplacements et n'est pas averti des changements dans la base le concernant.
- scroll-sensitive. Le `ResultSet` admet les déplacements et n'est pas averti des changements dans la base le concernant.

Ils peuvent aussi avoir des "types de concurrence" : read-only ou updatable.

Utilisation :

```
Statement stmt = conX.createStatement(
        ResultSet.TYPE_SCROLL_SENSITIVE,
        ResultSet.CONCUR_UPDATABLE);
    stmt.setFetchSize(25);

    ResultSet rs = stmt.executeQuery(
        "SELECT emp_no, salary FROM employees");
```

Traitement avancé des `ResultSet` (suite)

Un `ResultSet` updatable (de type `CONCUR_UPDATABLE`) peut mettre à jour, insérer, détruire certaines de ses lignes. Ces modifications sont faites sur le `ResultSet` pas dans la base. Elles seront faites dans la base à l'aide de la méthode `updateRow()`
Exemple :

```
rs.first();
rs.updateString(1, "100020");
rs.updateFloat("salary", 10000.0f);
rs.updateRow();
```

voir à :
`http://java.sun.com/products/jdk/1.3/docs/guide/jdbc/spec2/jdbc2.1.frame5.html`

Traitement avancé des `ResultSet` (suite)

Mouvement avancé du curseur.

JDBC 2.0 permet de lancer sur un `ResultSet` les méthodes :

```
public boolean absolute(int row) throws
SQLException
public boolean relative(int rows)
throws SQLException
public boolean previous() throws
SQLException

public void afterLast() throws
SQLException
public void beforeFirst() throws
SQLException
```

Bibliographie

Livre JDBC et Java Guide du programmeur, George Reese edition O'Reilly,
ISBN 2-84177-042-7

Les spécifications JDBC sont à :
http://java.sun.com/products/jdk/1.3/docs/guide/jdbc/getstart/GettingStartedTOC.fm.html

La page de départ de la technologie JDBC
http://java.sun.com/products/jdk/1.3/docs/guide/jdbc/index.html

le tutorial de SUN sur JDBC
http://www.javasoft.com/docs/books/tutorial/jdbc/index.html

Bases de données MS-Access, Conception et programmation, Steven Roman traduit en français; ed O'Reilly ISBN 2-84177-054-0

Merci à Henri ANDRÉ pour ses remarques et suggestions.