accès aux bases de données avec JDBC

- introduction
- pilotes
- mise en œuvre de l'API JDBC
- Data Access Objects (DAO)

introduction

- trois types de bases de données existent aujourd'hui:
  - les bases hiérarchiques, en voie de disparition
  - les bases relationnelles, les plus utilisées
  - les bases objets, qui font de timides apparitions
- les bases objets sont le prolongement naturel des programmes écrits en Java
- la domination hégémonique des bases relationnelles a rendu nécessaire leur interfaçage avec ces mêmes programmes
- la solution choisie par Sun-Oracle est de fournir à l'utilisateur des méthodes permettant la gestion d'une base de données relationnelle en langage SQL (Structured Query Language)
- l'ensemble de ces méthodes constituent l'api JDBC (Java DataBase Connectivity) du package **`java.sql`**
- Java 7 comporte l'API JDBC 4.1

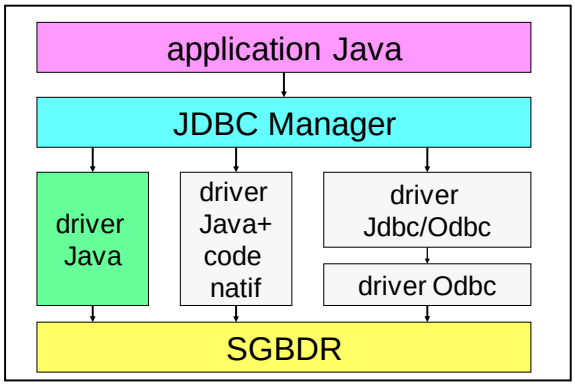
pilotes

- le pilote (driver) est l'élément indispensable pour permettre à un programme Java de se connecter à une base relationnelle et a pour rôle de réaliser l'interfaçage entre Java et une base spécifique
- les pilotes sont de 4 types:
- les pilotes de type 1 nécessitent la présence d'un driver ODBC (Open DataBase Connectivity) pour fonctionner
  - ils jouent en effet de rôle d'un pont de JDBC vers ODBC. Sun-Oracle fournit un tel pont en standard.
  - cela conduit à une solution peu performante mais facile à implémenter, les pilotes ODBC étant très répandus

pilotes

- les pilotes de type 2 sont partiellement écrits en Java, le reste l'étant en code natif, ce qui nécessite l'installation de ce code sur la machine cible
  - le déploiement est moins facile qu'avec les types 3 et 4
- les pilotes de type 3 sont entièrement écrits en Java, et utilisent un protocole indépendant pour communiquer avec un composant serveur, lequel traduit les requêtes en langage spécifique à la base
- les pilotes de type 4 sont entièrement écrits en Java, et traduisent directement des requêtes JDBC dans le protocole spécifique à la base
  - ce sont les pilotes actuellement les plus utilisés car portables

pilotes



mise en œuvre de l'API JDBC

- connexions
- requêtes SQL
- résultats
- gestions des exceptions
- requêtes paramétrées
- procédures stockées
- méta-données
- transactions
- pooling de connexions

connexions

- l'utilisation d'une base de données suppose l'établissement préalable d'une connexion avec elle
  - cette connexion nécessite elle-même l'utilisation du pilote approprié
- depuis JDBC 4.0, le chargement du pilote s'effectue automatiquement dès lors qu'il est présent dans le classpath
  - avant JDBC 4.0, le chargement du pilote dans la VM s'effectuait au moyen de la méthode **`forName`** de la classe **`Class`**:

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

    - le nom de la classe principale du pilote est transmis en paramètre sous la forme d'une chaîne de caractères, facilitant ainsi la mise en œuvre de configurations évolutives
    - exemples de classes de pilotes:
      - `sun.jdbc.odbc.JdbcOdbcDriver`
      - `com.mysql.jdbc.Driver`
      - `COM.cloudscape.core.JDBCDriver`

connexions

- la connexion à une base de données est représentée par un objet de l'interface `java.sql.`**`Connection`**, obtenu par la méthode **`getConnection`** de la classe **`DriverManager`**
  - qui va rechercher, parmi ceux enregistrés, un pilote permettant de se connecter à la base dans le protocole précisé

```
static Connection getConnection(String url);
static Connection getConnection(String url,
  String user, String password) ;
```

  - `url`: nom de la base de données, en précisant le protocole utilisé. Par exemple `jdbc:odbc:PCShop` représente la base dont le nom est `PCShop`, dans le protocole `jdbc`, dans le sous-protocole `odbc`
  - `user`: nom de l'utilisateur, ou chaîne vide le cas échéant
  - `password`: mot de passe pour accéder à la base, ou chaîne vide le cas échéant

connexions

- exemples d'url pour une connection à une base appelée vignerons:
  - `jdbc:odbc:vignerons`
  - `jdbc:mysql://localhost:3306/vignerons`
  - `jdbc:cloudscape:vignerons`
  - `jdbc:oracle:thin:@//hote56:1521/vignerons`

exemple:

```
Connection con=DriverManager.getConnection(
  jdbc:mysql://localhost:3306/vignerons,"root","admin");
```

- la fermeture de la connexion par la méthode **`close`** est nécessaire lorsque la connexion n'est plus utilisée (Cf libération des ressources)

requêtes SQL

- l'émission de requêtes SQL s'effectue via un objet de type `java.sql.`**`Statement`** (interface), obtenu par la méthode **`createStatement`** de **`Connection`**:

```
Statement createStatement();
```

exemple:

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

- les ordres SQL peuvent alors être transmis sous la forme de `String` au moyen des méthodes **`executeUpdate`** ou **`executeQuery`** ou encore **`execute`**
  - **executeUpdate** pour les requêtes SQL de type INSERT, UPDATE, DELETE, ou autres ordres de type DDL (CREATE TABLE, DROP TABLE)
  - **executeQuery** pour les requêtes SQL de type SELECT
  - **execute** pour tout type de requête

requêtes SQL

`int` **`executeUpdate`**`(String sql);`

- retourne le nombre de lignes modifiées ou 0

exemple:

```
stmt.executeUpdate("UPDATE ...");
```

`ResultSet` **`executeQuery`**`(String sql);`

- retourne un objet `ResultSet` (interface) permettant de balayer les résultats de la requête

exemple:

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

- la méthode **`close`** doit être invoquée lorsque l'objet **`Statement`** n'est plus utilisé

résultats

- les résultats d'une requête de type SELECT sont exploités au moyen d'un objet **`ResultSet`** (interface)
- les méthodes **`next`** et **`previous`** de cette interface permettent d'explorer les enregistrements obtenus, en déplaçant respectivement un curseur sur l'enregistrement suivant et l'enregistrement précédent

```
boolean next();
boolean previous();
```

- ses méthodes **`getString`** et plus généralement **`get`**`XXX` permettent d'obtenir la valeur d'un champ d'un enregistrement, en fournissant soit son numéro, soit son nom de colonne

```
String getString(int columnIndex);
String getString(String columnName);
```

résultats

- exemple:

```
//Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
try (Connection con=DriverManager.getConnection(
       "jdbc:odbc:PCShop","","");
       Statement stmt=con.createStatement();
       ResultSet rs=stmt.executeQuery(sql)){
       .          .         .
       // affichage des résultats
       while(rs.next()) {
         for(int i=1;i<=n;i++) {
              System.out.println(" "+rs.getString(i));
         }
         System.out.println();
}catch(SQLException e){
   System.out.println(e);
}
```

résultats

- les méthodes **get**XXX permettent de récupérer la valeur d'un champ en fonction de son type: **getByte, getShort, getInt, getLong, getFloat, getDouble, getBigDecimal, getBoolean, getString, getBytes, getData, getTime, getTimestamp, getAsciiStream, getUnicodeStream, getBinaryStream, getObject**
  - elles acceptent le nom de la colonne concernée ou son numéro
    - le numéro fourni en argument de ces méthodes commence à la valeur 1 (première colonne)
  - les méthodes **get**XXX réalisent une conversion des types de données SQL vers les types Java

résultats

- correspondance entre les types SQL et Java:

SQL	Java
INTEGER ou INT	int
SMALLINT	short
NUMERIC, DECIMAL ou DEC	java.sql.Numeric
FLOAT	double
REAL	float
DOUBLE	double
CHARACTER ou CHAR	String
VARCHAR	String
BOOLEAN	boolean
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
BLOB	java.sql.Blob
CLOB	java.sql.Clob
ARRAY	java.sql.Array

résultats

- l'objet `ResultSet` obtenu par défaut ne peut être mis-à-jour et ne permet qu'un parcours ascendant
- l'obtention de `ResultSet` plus élaborés s'effectue au moyen de la méthode `createStatement` de `Statement`:

```
Statement createStatement(int resultSetType,
        int resultSetConcurrency)
```

  - `resultSetType` prend l'une des valeurs:
    - TYPE_FORWARD_ONLY: parcours dans le sens ascendant uniquement
    - TYPE_SCROLL_INSENSITIVE: parcours dans les deux sens, insensible aux changements dans les données de la base
    - TYPE_SCROLL_SENSITIVE: parcours dans les deux sens, sensible aux changements dans les données de la base
  - `resultSetConcurrency` prend l'une des valeurs:
    - CONCUR_READ_ONLY: le `ResultSet` ne peut être mis-à-jour
    - CONCUR_UPDATABLE: le `ResultSet` peut être mis-à-jour

résultats

- certaines applications ont parfois besoin de connaître le nombre de lignes avant le parcours de la table résultat
  - `ResultSet` comporte quelques méthodes utiles (si `Scrollable`):
    - `getRow` retourne le numéro de la ligne courante
    - `last` positionne le curseur sur la dernière ligne
    - `absolute` positionne le curseur sur une ligne donnée
    - `beforeFirst` positionne le curseur avant la première ligne
  - exemple:

```
public int rowCount(ResultSet rs) throws SQLException{
    int rowCount = 0;
    int currRow = rs.getRow();
    if (!rs.last()) return -1; // ResultSet valide ?
    rowCount = rs.getRow();
    if (currRow == 0) rs.beforeFirst(); // rétablit le curseur
    else rs.absolute(currRow);
    return rowCount;
}
```

résultats

- le nombre de lignes résultat obtenues en une fois est déterminé par le pilote
  - pour limiter ou augmenter le nombre de lignes en résultat, appeler la méthode:

```
void setFetchSize(int rows)
```

  - exemple:

```
result.setFetchSize(100);
```

    - les appels à la méthode `next()` retournent la donnée en cache jusqu'à la ligne 101, pour laquelle le pilote récupèrera les 100 lignes suivantes

résultats

- Java 7 a introduit l'API `RowSet`
- un `javax.sql.rowset.`**`RowSetProvider`** permet d'obtenir un `javax.sql.rowset.`**`RowSetFactory`**

  `RowSetFactory newFactory()`
- un `RowSetFactory` permet ensuite d'obtenir différent types de **`RowSet`**

`CachedRowSet`	**`createCachedRowSet`**`()`
`FilteredRowSet`	**`createFilteredRowSet`**`()`
`JdbcRowSet`	**`createJdbcRowSet`**`()`
`JoinRowSet`	**`createJoinRowSet`**`()`
`WebRowSet`	**`createWebRowSet`**`()`

résultats

- **`CachedRowSet`**
  - un conteneur pour des lignes de données mises en cache
- **`FilteredRowSet`**
  - un conteneur permettant le filtrage des données
- **`JdbcRowSet`**
  - une enveloppe autour de `ResultSet` permettant de le considérer comme un JavaBean
- **`JoinRowSet`**
  - un `RowSet` qui fournit des mécanismes permettant de combiner les données de plusieurs `RowSets` (SQL Join)
- **`WebRowSet`**
  - un `RowSet` qui supporte le format XML Document permettant de décrire en XML un `RowSet`

résultats

- exemple: `JdbcRowSet`

```
try (JdbcRowSet jdbcRs = RowSetProvider.newFactory().
            createJdbcRowSet()) {
    jdbcRs.setUrl(url);
    jdbcRs.setUsername(username);
    jdbcRs.setPassword(password);
    jdbcRs.setCommand("SELECT * FROM Employee");
    jdbcRs.execute();
    while (jdbcRs.next()) {
        int empID = jdbcRs.getInt("ID");
        String first = jdbcRs.getString("FirstName");
        String last = jdbcRs.getString("LastName");
        Date birthDate = jdbcRs.getDate("BirthDate");
        float salary = jdbcRs.getFloat("Salary");
    }
    //...}
```

gestion des exceptions

- la plupart des méthodes de l'api JDBC lancent des exceptions de type **`SQLException`** qui peuvent être ainsi récupérées:

```
try {.              .              .
}catch(SQLException esql){
 String err="erreur IHMTable "+esql.getMessage();
 while(esql!=null){
        err+="\nMessage: "+esql.getMessage()+'\n';
        err+="SQLState: "+esql.getSQLState()+'\n';
        err+="Code d'erreur:"+
        esql.getErrorCode()+'\n';
        esql=esql.getNextException();
 }
```

  - **`getNextException`** permet d'atteindre l'objet exception suivant dans la liste
  - **`getSQLState`** fournit le code d'erreur SQL
  - **`getErrorCode`** fournit le code d'erreur du driver

libération des ressources

- la fermeture de l'objet `Connection` entraîne automatiquement celle des objets `Statement`, entraînant à son tour la fermeture des objets `ResultSet`
  - les ressources utilisées par `ResultSet` peuvent ne pas être libérées avant l'intervention du GC
- une bonne pratique consiste à fermer explicitement les objets `Connection`, `Statement` et `ResultSet`
  - exemple:

```
try (Connection con=DriverManager.getConnection(
        "jdbc:odbc:PCShop","","");
      Statement stmt=con.createStatement();
      ResultSet rs=stmt.executeQuery(sql)){
```

    - le compilateur vérifie que les les objets entre les parenthèses de `try` implémentent l'interface `java.lang.AutoClosable`
    - la méthode `close` est automatiquement appelée à la fin du bloc `try` sur chacun des objets dans l'ordre inverse de leur création

requêtes paramétrées

- lorsqu'une requête, ou une transaction SQL est exécutée de nombreuses fois, il peut être préférable d'utiliser une requête paramétrée
  - il s'agit d'une requête SQL généralement incomplète car paramétrée, dont les paramètres seront fournis ultérieurement
- il faut obtenir un objet **`PreparedStatement`** (interface), obtenu par la méthode **`prepareStatement`** de **`Connection`**:

```
PreparedStatement prepareStatement(String sql);
```

- cette méthode reçoit en argument la requête SQL paramétrée à utiliser, dans lequel chacun des paramètres est représenté par le symbole ?

requêtes paramétrées

- les paramètres sont fournis ultérieurement au moyen des méthodes **set**XXX de **PreparedStatement** en précisant l'indice du paramètre en premier argument (à partir de 1) et sa valeur en second

```
void setString(int parameterIndex, String x);
```

- les méthodes **set**XXX correspondant à chacun des types de base: **setByte, setShort, setInt, setLong, setFloat, setDouble, setBigDecimal, setBoolean, setString, setBytes, setData, setTime, setTimestamp, setAsciiStream, setUnicodeStream, setBinaryStream, setObject**
- l'exécution de l'ordre SQL est effectuée par l'une des méthodes **executeUpdate** ou **executeQuery** de la même interface

requêtes paramétrées

- exemple:

```
String sql="SELECT Desig, Prix FROM articles"+
       " WHERE Desig LIKE ? AND Prix < ?";
PreparedStatement pr_stmt=con.prepareStatement(sql);
pr_stmt.setString(1,"carte%");
pr_stmt.setString(2,"100");
ResultSet rs=pr_stmt.executeQuery();
.           .          .
while(rs.next()) {  // affichage des résultats
   for(int i=1;i<=n;i++) {
     System.out.println(" "+rs.getString(i));
   }
   System.out.println();
}
```

procédures stockées

- il s'agit d'un groupe de requêtes SQL paramétrées ou non, identifiées par un nom, et destinées à être exécutées comme une seule requête
- ces groupes appelés procédures sont stockées dans la base de données elle-même
- la syntaxe des procédures stockées varie d'une base de données à une autre, ex:

```
String proc="CREATE PROCEDURE listetout AS "+
      "SELECT * FROM articles";
Statement stmt=con.createStatement();
stmt.executeUpdate(listetout);
```

- la méthode **`executeUpdate`** de l'interface **`Statement`** permet de stocker la procédure dans la base

procédures stockées

- il faut ensuite disposer d'un objet **`CallableStatement`** (interface), obtenu par la méthode **`prepareCall`** de **`Connection`**:

```
CallableStatement prepareCall(String sql);
```

exemple:

```
CallableStatement cs=con.prepareCall("{call listetout}");
```

- une fois la procédure stockée dans la base, il suffit de l'invoquer par la méthode **`executeQuery`**, **`executeUpdate`** ou **`execute`** de l'interface **`CallableStatement`**

```
ResultSet rs=cs.executeQuery();
```

méta-données

- diverses informations peuvent être obtenues sur les résultats d'une requête, au moyen d'un objet **`ResultSetMetaData`**, que l'on obtient au moyen de la méthode **`getMetaData`** de l'interface **`ResultSet`**

```
ResultSetMetaData getMetaData();
```

- les méthodes **`getColumnCount`**, **`getColumnType`**, **`getColumnName`**, **`getTableName`** permettent d'obtenir le nombre de colonnes, leur type, leur nom, le nom de leur table

```
ResultSet rs=stmt.executeQuery(sql);
ResultSetMetaData rsm=rs.getMetaData();
int n=rsm.getColumnCount();  // nombre de colonnes en
  résultat
while(rs.next()) {
   for(int i=1;i<=n;i++){…}
}
```

méta-données

- diverses informations sur la base de données elle-même peuvent être obtenues, comme par exemple le nom de ses tables, ou le nom des colonnes de l'une de ses tables
- ces informations, dites méta-données, sont récupérables à partir d'un objet **`DataBaseMetaData`**, que l'on obtient par la méthode **`getMetaData`** de **`Connection`**

```
DatabaseMetaData getMetaData();
```

- les méthodes comme **`getTables`**, **`getColumns`** ou **`getProcedures`** de **`DataBaseMetaData`** permettent respectivement d'obtenir les noms des tables, les noms des colonnes d'une table, les noms des procédures stockées
  - la méthode `supportsANSI92EntrySQL` permet de savoir si le SGBD supporte le standard SQL-92 (portabilité entre SGBD)

méta-données

- exemple:

```
DataBaseMetaData dbm=con.getMetaData();
String []typedetable={"TABLE","VIEW"};
ResultSet rs=dbm.getTables(null,null,"%",typedetable);
ResultSetMetaData rsm=rs.getMetaData();
int n=rsm.getColumnCount(); // nombre de col. résultat
while(rs.next()) {
    for(int i=1;i<=n;i++) {
        System.out.println(" "+rs.getObject(i));
    }
    System.out.println();
}
con.close();
```

transactions

- lorsque l'on souhaite émettre des requêtes SQL groupées, afin de réaliser une opération complexe dans une base, l'échec de l'une d'elles peut entraîner des incohérences dans cette base de données
- de plus, une requête d'un autre utilisateur peut avoir lieu alors que le groupe d'ordres SQL n'est pas encore achevé, pouvant entraîner des incohérences dans les résultats
- il faut alors opérer en mode transactionnel, l'ensemble des requêtes SQL constituant une transaction, vue comme une requête unique
- par défaut, chaque requête SQL émise est automatiquement validée (committée) après son exécution

transactions

- il est possible de modifier ce comportement afin de demander explicitement la validation (donc la réalisation) des ordres SQL émis
- il faut préalablement invalider le mode par défaut par la méthode **`setAutoCommit`** de l'interface **`Connection`**:

```
void setAutoCommit(boolean autoCommit);
```

- une fois le groupe d'ordres SQL émis, il suffit de valider ou annuler la transaction, au moyen respectivement des méthodes **`commit`** et **`rollback`** de cette interface

```
void commit();
void rollback()
```

  - il est prudent de retourner au mode par défaut dès que le mode transactionnel n'est plus nécessaire, de façon à éviter de maintenir des verrous dans la base

transactions

```
Connection con=null;
try {
    Connection con=DriverManager.getConnection(...);
    con.setAutoCommit(false);
    .              .              .
    // ...exécution des requêtes
    con.commit();
    con.setAutoCommit(true);
}catch(SQLException e){
    con.rollback();
    //...
}
```

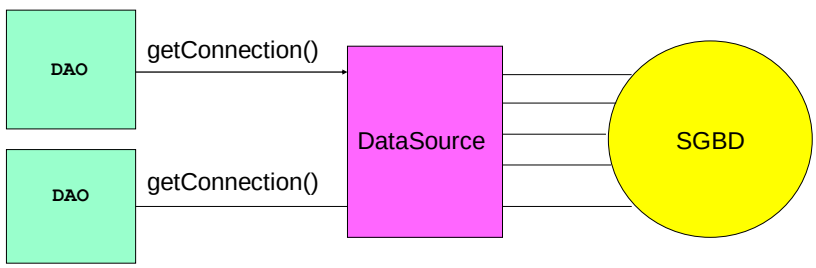
- en cas d'échec dans l'exécution de l'une des requêtes de la transaction, les modifications qui ont pu être partiellement effectuées doivent être annulées par la méthode **`rollback`**

```
void rollback();
```

pooling de connexions

- l'accès aux bases de données depuis une application web nécessite en général de bonnes performances, compte-tenu du nombre de requêtes potentielles
- afin d'éviter des temps de connexion pénalisants, les serveurs d'applications sont tous pourvus de pools (réserves) de connexions aux bases de données
- un pool de connexions contient un certain nombre de connexions physiques ouvertes vers la base de données, l'utilisateur puisant dans ce pool en fonction de ses besoins
- l'obtention d'une connexion (logique) s'effectue en s'adressant au pool, en lieu et place du SGBD, et devient une opération rapide

pooling de connexions



DataSource

- de façon à rendre uniforme l'accès aux pools de connexions, ces objets implémentent l'interface `javax.sql.`**`DataSource`**:

```
public interface DataSource {
    public Connection getConnection();
    public Connection getConnection(String username,
      String password);
    public PrintWriter getLogWriter();
    public void setLogWriter(PrintWriter out);
    public void setLoginTimeout(int seconds);
    public int getLoginTimeout() ;
}
```

DataSource

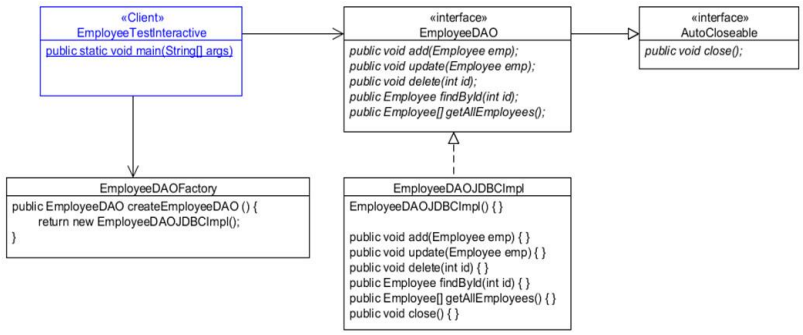
- de façon à rendre le fonctionnement du pool efficace, une connexion doit être libérée dès qu'elle n'est plus utilisée, au moyen de la méthode `close` de l'interface **`Connection`**:

```
close();
```

  - dans le cas contraire, sa libération peut intervenir sur time-out (programmable) géré par le pool
- les instructions d'accès au pool à la base de données et peuvent se trouver dans une servlet, un JavaBean, ou même dans une JSP !!!, mais seront de préférence situés dans un objet spécifique appelé DAO (DataAccess Object)

Data Access Objects (DAO)

- le pattern DAO permet de bien séparer le code d'accès à la base de données (technique) du code client (métier)
  - le code client ne doit pas être dépendant des caractéristiques techniques du SGBD
  - il ne doit pas recevoir d'exceptions SQLException



Data Access Objects (DAO)

- le pattern DAO doit comporter:
  - une interface `EmployeeDAO` qui déclare les méthodes utiles pour manipuler les données d'un objet `Employee`
  - une classe `EmployeeDAOJDBCImpl` qui implémente l'interface `EmployeeDAO`
    - la classe d'implémentation peut être remplacée par une autre sans impact sur l'application cliente
  - une fabrique permettant de créer des instances d'une implémentation de l'interface `EmployeeDAO`
    - une fabrique permet d'isoler le client des détails concernant l'implémentation