Aplicații Integrate pentru Întreprinderi

Laborator 3

21.10.2013

Gestiunea informațiilor dintr-o bază de date MySQL prin JDBC

Scopul laboratorului îl reprezintă înțelegerea mecanismului prin care pot fi manipulate informațiile stocate într-o bază de date MySQL din cadrul unei aplicații Java prin funcționalitățile oferite Java Database Connectivity (JDBC).

1. JDBC – funcționalitatea, componentele şi arhitectura sa
2. Ce este un *„driver”* de conectare la un sistem de gestiune pentru baze de date ?
3. Configurare Connector/J
4. Arhitectura JDBC
5. Conectarea la sistemul de gestiune al bazei de date
6. Interogarea bazei de date conform specificației JDBC
7. Utilizarea tranzacțiilor în cazul accesului concurent la date
8. Gestiunea informațiilor din dicționarul de date
9. Tratarea excepțiilor de tip `SQLException`
10. Alternative la manipularea informațiilor din sursele de date

1. JDBC – funcționalitatea, componentele şi arhitectura sa

JDBC (Java Database Connectivity) este o interfață de programare Java prin intermediul căreia pot fi manipulate informațiile dintr-o sursă de date.

Operațiile pe care le pune la dispoziție acest API sunt:

➊ conectarea (respectiv, deconectarea) la o sursa de date, cel mai frecvent o bază de date;

➋ transmiterea de interogări către sursa de date respectivă (de tip `SELECT`, `INSERT`, `UPDATE`, `DELETE` dar şi referitoare la informațiile din dicționarul de date), obținerea rezultatelor aferente comenzilor realizate şi procesarea lor, inclusiv propagarea modificărilor realizate.

Componentele pe care le include JDBC sunt:

➊ **interfața de programare propriu-zisă** (JDBC API 4.1) care oferă acces la informațiile din baza de date folosind limbajul Java[1]. Este conținută de pachetele `java.sql` şi `javax.sql`, incluse atât în platforma standard (Java SE) cât şi în platforma pentru implementarea aplicațiilor de întreprinderi (Java EE).

➋ **modulul pentru gestiunea driver-elor** (JDBC Driver Manager), reprezentat de clasa `DriverManager` în care sunt definite obiectele ce pot conecta aplicațiile Java la un „driver” JDBC. De asemenea, pachetele `javax.naming` şi `javax.sql` oferă posibilitatea realizării unei conexiuni către o sursă de date (obiect de tip `DataSource`) înregistrată de către serviciul de nume *Java Naming and Directory Interface* (JNDI).

➌ **suita de teste JDBC** oferă o serie de utilitare care verifică dacă „driverele” JDBC sunt compatibile cu o aplicație Java;

➍ **puntea ODBC-JDBC** pentru realizarea de conexiuni JDBC prin „drivere” ODBC care vor trebui încărcate pe fiecare maşină ce le utilizează.

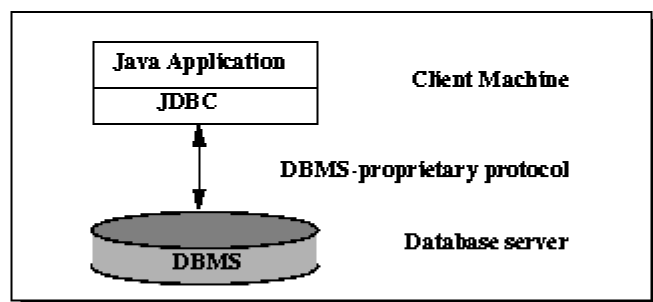
[1] De asemenea, într-un mediu distribuit, există posibilitatea de interacțiune cu mai multe surse de date simultan.

Acestea sunt utilizate atunci când nu există alte soluții de conectare (native). Puntea este ea însăşi un tip de driver bazat pe tehnologia JDBC, fiind conținut de clasa `sun.jdbc.odbc.JdbcOdbcDriver`, definind subprotocolul `odbc`. Deoarece aceasta este o soluție temporară (până la elaborarea unor „drivere” native), componenta va fi eliminată începând cu Java 8.

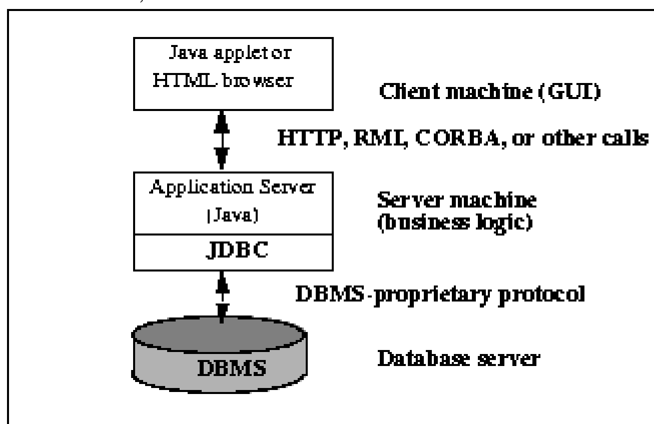
Arhitectura JDBC defineşte două modele de procesare pentru accesul la informațiile din baza de date:

➊ în **modelul pe două niveluri** aplicația Java comunică în mod direct cu sursa de date, necesitând un „driver” JDBC specific acesteia care să poată accesa informațiile. Instrucțiunile utilizatorului sunt transmise sursei de date care întoarce la rândul ei rezultatele. Cele două componente rulează de obicei pe maşini diferite conectate prin intermediul unei rețele de calculatoare (intranet/Internet), modelul fiind cunoscut şi sub numele de client-server.

➋ în **modelul pe trei niveluri** comenzile sunt transmise prin intermediul unor servicii puse la dispoziție într-un nivel intermediar, care au acces la sursa de date. Ca atare, informațiile vor trece – în ambele direcții – prin acest nivel. Conexiunea la nivelul intermediar se poate face prin HTTP sau alte metode pentru acces la distanță (RMI, CORBA, servicii web). Câteva avantaje pe care le oferă acest model sunt controlul centralizat al accesului la date, simplificarea procesului de dezvoltare al aplicațiilor, performanța.



Modelul pe două niveluri (client-server)



Modelul pe trei niveluri

JDBC începe să fie adoptat pe scară largă datorită suportului pentru gestiunea paralelă a conexiunilor, tranzacții distribuite precum şi posibilităților de procesare a informațiilor deconectate de la sursa de date corespunzătoare.

2. Ce este un *„driver”* de conectare la un sistem de gestiune pentru baze de date ?

Un *„driver”* de conectare la un sistem de gestiune al bazei de date reprezintă o bibliotecă prin care sunt transformate apelurile JDBC (din limbajul de programare Java) într-un format suportat de protocolul de rețea folosit de sistemul de gestiune al bazei de date, permițând programatorilor să acceseze datele din medii eterogene.

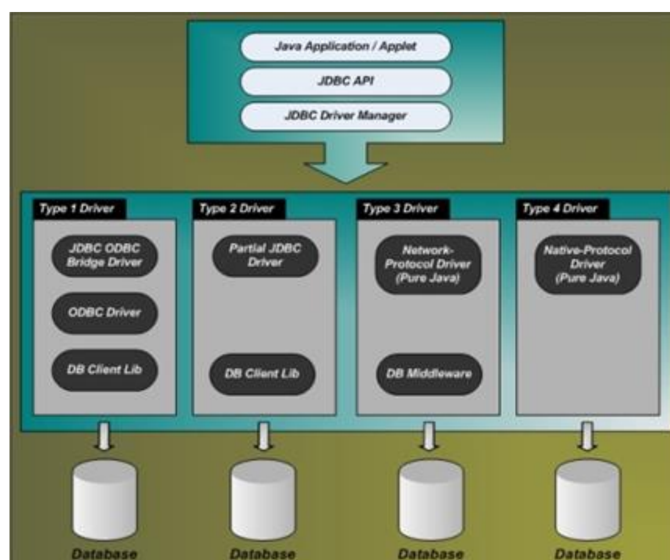
Prin urmare, „driver-ul” pentru sistemul de gestiune al bazei de date realizează legătura între nivelul de logică a aplicației[2] şi nivelul de date (reprezentat prin baza de date propriu-zisă).

[2] În cadrul nivelului de logică a aplicației, poate fi descris un subnivel pentru acces la date care accesează în mod explicit „driver-ul”.

Există patru implementări pentru „drivere” JDBC:

- **tipul 1**: drivere ce implementează API-ul JDBC ca punte peste ODBC (*eng.* Open DataBase Connectivity) care accesează datele propriu-zise; portabilitatea lor este relativ redusă, fiind dependente de o bibliotecă scrisă în cod nativ (şi nu implementate complet în Java); de asemenea, viteza de execuție este destul de redusă datorită transformărilor ce trebuie realizate atât la transmiterea interogărilor cât şi a rezultatelor; implică instalarea de utilitare suplimentare pe client ceea ce le poate face incompatibile cu anumite tipuri de aplicații; această soluție este tranzițională şi ar trebui folosită în cazul în care sistemul de gestiune pentru baze de date respectiv nu oferă un driver JDBC scris doar Java; Oracle nu implementează acest tip de drivere; totuşi, întrucât există „drivere” ODBC pentru toate bazele de date existente, o astfel de soluție oferă acces către orice tip de date;
- **tipul 2**: drivere care sunt scrise parțial în Java şi parțial în cod nativ, folosind o bibliotecă specifică pentru sursele de date la care se conectează, ceea ce le reduce portabilitatea[3] şi posibilitatea utilizării în contextul rețelelor de calculatoare; de asemenea, nu toți producătorii oferă astfel de biblioteci care trebuie instalate pe client.
- **tipul 3**: drivere dezvoltate exclusiv în Java care comunică cu middleware-ul printr-un protocol independent de baza de date, comenzile fiind transformate la acest nivel în instrucțiuni specifice bazei de date care pot fi utilizate apoi pentru accesarea sursei de date; avantajele acestei soluții constau în facilitățile oferite de middleware cum ar fi controlul încărcării, memorarea conexiunilor, rezultatelor interogărilor într-o zonă tampon, opțiuni de administrare sistemului (autentificare, analiza performanțelor); pe lângă portabilitate acest tip de „driver” este performant (cel mai eficient între toate) şi scalabil (se pot accesa mai multe tipuri de baze de date); nu trebuie încărcate pe client produse specifice producătorilor, ceea ce îl face adecvat utilizării în Internet; protocolul independent de baza de date poate determina ca încărcarea driverului să se facă rapid; dezavantajul consta în faptul că operațiile specifice bazei de date trebuie realizate în cadrul nivelului intermediar;
- **tipul 4**: drivere scrise în Java care implementează un protocol de rețea specific sistemului de gestiune pentru baze de date, spre a se conecta la sursa de date în mod direct[4]; se asigură astfel independența de platformă cât şi eficiența întrucât nu sunt necesare niveluri suplimentare pentru translatarea codului dintr-un format într-altul; totodată, nu este necesară instalarea de utilitare suplimentare pe client şi pe server, ceea ce face ca abordarea să fie compatibilă cu utilizarea peste o rețea de calculatoare; nu pot fi procesate mai multe baze de date în paralel, fiind necesar un driver pentru fiecare astfel de conexiune.

[3] Un exemplu de driver JDBC de tip 2 de la Oracle este OCI (Oracle Call Interface).
[4] MySQL Connector/J este un exemplu de driver JDBC de tip 4.



Tipuri de „drivere" JDBC [1]

Aşa cum se poate observa, pentru driverele de tip 1 şi 2 este necesară existența unor biblioteci specifice pentru fiecare tip de bază de date care trebuie puse la dispoziție de producătorii acestora.

Pentru driverul de tip 3 trebuie instalat un server de aplicații care comunică cu sistemul de gestiune pentru baze de date. De regulă acesta este configurat pentru a fi compatibil cu mai multe tipuri de baze de date, iar performanțele sale trebuie să compenseze timpul pentru transferul de informații de la şi către el.

Driverul de tip 4 este cel mai flexibil dintre toate întrucât nu necesită utilitare suplimentare, fiind şi independent de platformă.

3. Configurare Connector/J

Pentru conectarea la baza de date MySQL, se poate folosi **Connector/J**, driver nativ pentru Java dezvoltat de Oracle şi distribuit gratuit utilizatorilor.

Tot ce trebuie făcut pentru utilizarea driver-ului de conectare din limbajul de programare Java împreună cu sistemul de gestiune pentru bazei de date MySQL[5] este să descărcați arhiva care conține Connector/J[6] de pe pagina oficială (http://www.mysql.com/downloads/connector/j/), să o dezarhivați şi să adăugați fişierul .jar din rădăcină la classpath în momentul în care compilați aplicația.

În linie de comandă, acest lucru poate fi realizat astfel:

- compilare:

```
>javac -classpath .;mysql-connector-java-5.1.26-bin.jar <nume_fisier>.java
```

```
>javac -classpath .:mysql-connector-java-5.1.26-bin.jar <nume_fisier>.java
```

- rulare:

```
>java -classpath .;mysql-connector-java-5.1.26-bin.jar <nume_fisier>
```

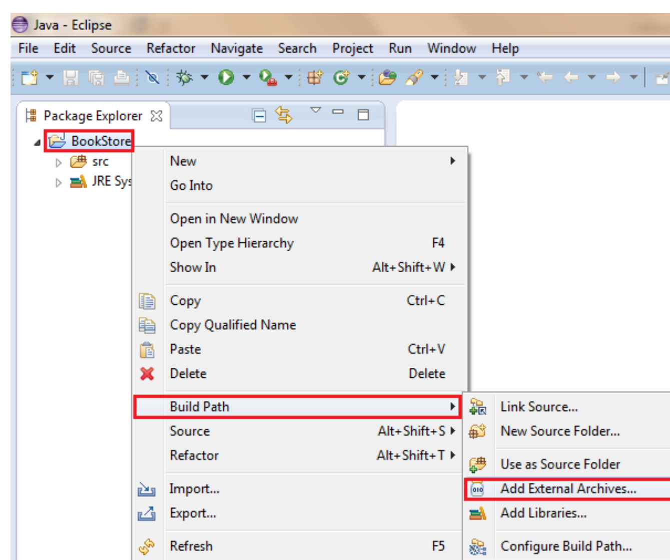
```
>java -classpath .:mysql-connector-java-5.1.26-bin.jar <nume_fisier>
```

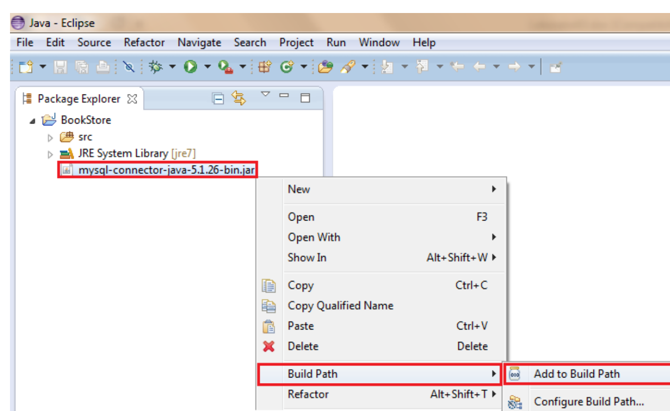
[5] Pentru alte sisteme de gestiune ale bazelor de date (Oracle, IBM DB2), „driver-ele" de conectare sunt de obicei disponibile la adresele de unde pot fi descărcate şi produsele propriu-zise. Java DB este distribuită împreună cu un „driver" de conectare la sistemul de gestiune al bazei de date.

[6] Ultima versiune disponibilă pe pagina oficială este 5.1.26. Pentru Windows este disponibil un fişier de tip .msi (Windows Installer) care va „instala" conectorul respectiv în directorul %System Root%\Program Files [(x86)]\MySQL\Connector J 5.1.26.

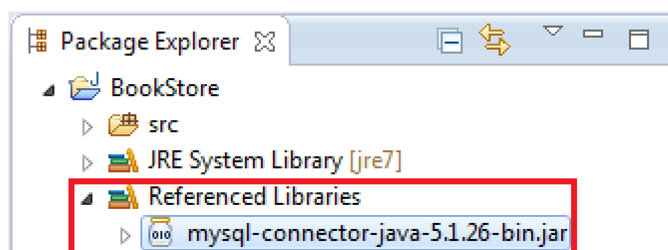
Mai uşor, puteți folosi medii integrate de dezvoltare a aplicațiilor, cum ar fi

- *Eclipse* (a fost testată versiunea Kepler – 4.3.1)
  - clasele conținute în arhiva .jar trebuie adăugate la calea proiectului prin
    - în cazul în care arhiva .jar nu există în structura proiectului: click dreapta pe numele proiectului → Build Path → *„Add External Libraries”*
    - în cazul în care arhiva .jar se găseşte în structura proiectului: click dreapta pe numele bibliotecii → Build Path → *„Add to Build Path”*
  - dacă biblioteca externă a fost adăugată în mod corect, numele ei trebuie să apară în meniul din stânga corespunzător proiectului, secțiunea *„Referenced libraries”*



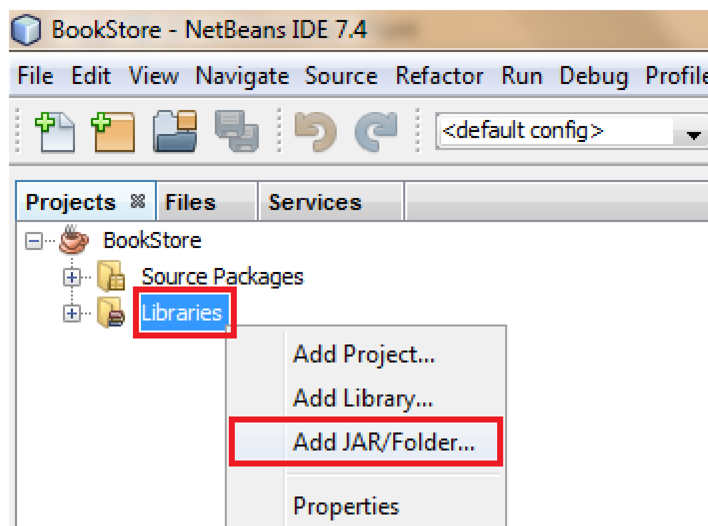


Adăugarea unei biblioteci externe pentru un proiect în *Eclipse Kepler*

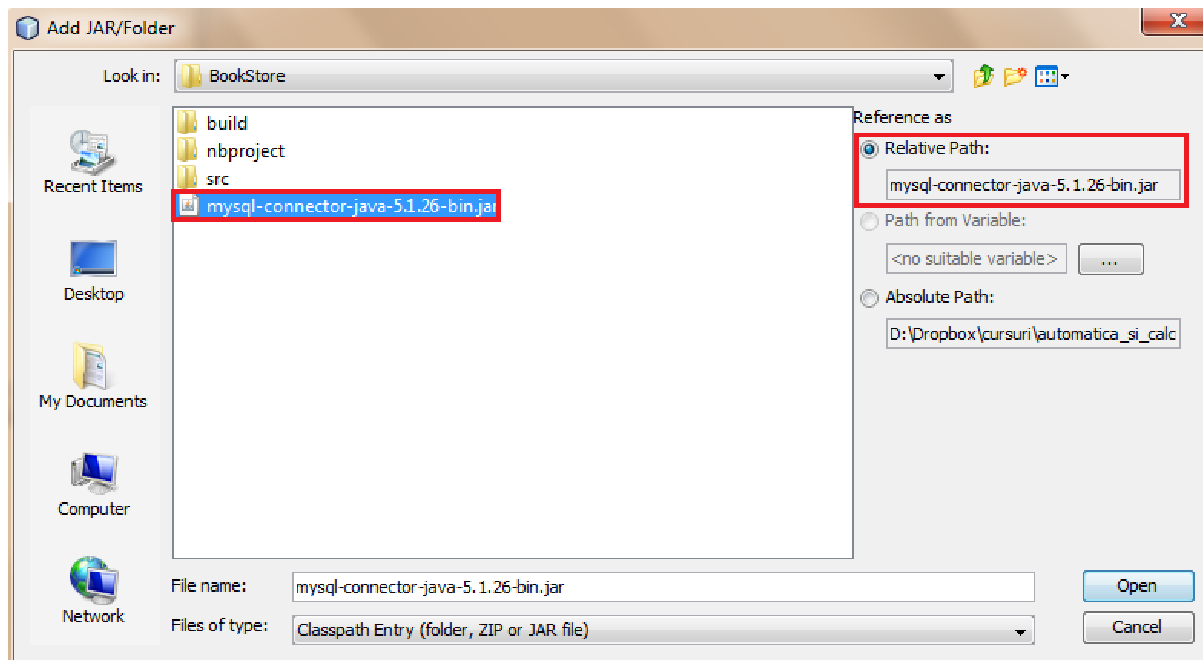


Adăugarea corectă a bibliotecii externe este marcată prin adăugarea ei în meniul *Package Explorer* aferent proiectului, secțiunea *Referenced Libraries*

- *NetBeans* (a fost testată versiunea 7.4)
  - în meniul din stânga corespunzător proiectului, alegeți *Libraries*, apoi click dreapta şi selectați opțiunea *Add JAR/Folder*
  - va apărea o fereastră de dialog în care aveți grijă să selectați referința drept cale relativă (Refernce As: → Relative Path), arhiva găsindu-se în sistemul de fişiere al proiectului



Adăugarea unei biblioteci externe pentru un proiect în *NetBeans 7.4*

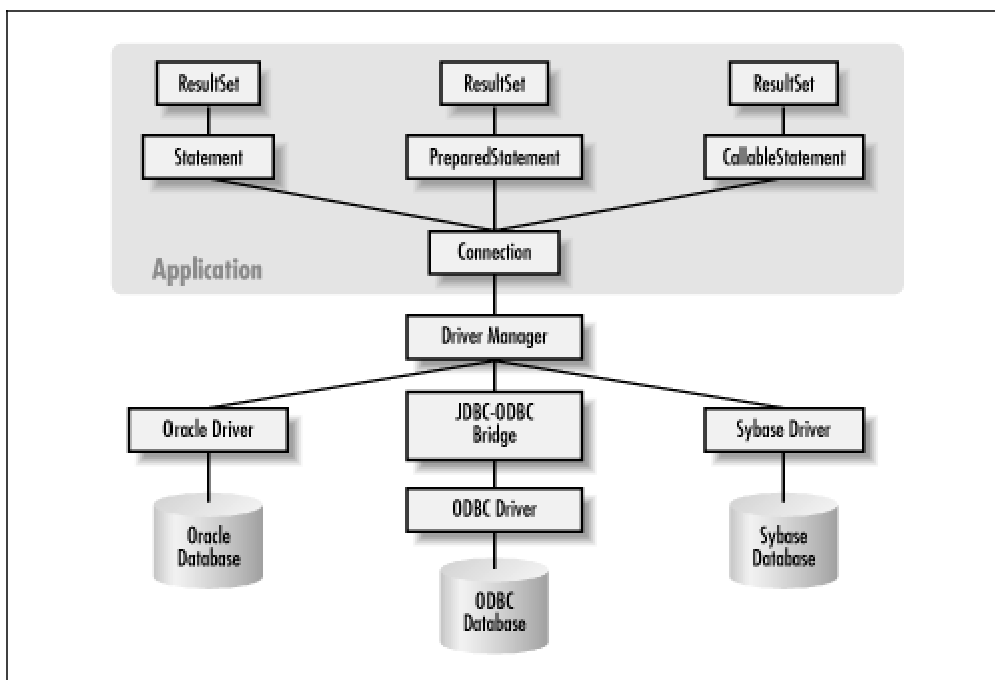


Referirea căii relative către biblioteca externă în condițiile în care este inclusă în sistemul de fişiere al proiectului

4. Arhitectura JDBC

Arhitectura protocolului JDBC este structurată pe două niveluri:

- un API JDBC responsabil de comunicația dintre aplicația Java şi modulul de gestiune al driver-ului;
- un API JDBC Driver care este responsabil de comunicația dintre modulul de gestiune al driver-ului şi baza de date; un astfel de nivel este independent atât în raport cu baza de date la care se conectează precum şi în raport cu limbajul de programare din care este accesat;



Arhitectura protocolului JDBC [2]

O aplicație care se conectează la o bază de date folosind protocolul JDBC trebuie să urmeze următorii paşi:

- [înregistrarea „driver”-ului] – opțional, se poate face în două moduri[7]:
  - `DriverManager.registerDriver (new com.mysql.jdbc.Driver());`
  - `Class.forName ("com.mysql.jdbc.Driver").newInstance();`
- deschiderea conexiunii la baza de date
- realizarea de interogări[8] către baza de date
- procesarea rezultatelor obținute cu propagarea modificărilor realizate înapoi în baza de date
- închiderea conexiunii la baza de date

[7] Metoda `DriverManager.registerDriver` implică existența driver-ului la momentul când se realizează compilarea, în timp ce metoda `Class.forName` verifică acest lucru la execuție, lipsa claselor respective fiind semnalate printr-o excepție `NoClassDefFoundError`. Începând cu JDBC 4.0, se încarcă în mod automat „driver-ul” identificat în classpath, astfel încât metoda `Class.forName` nu mai trebuie apelată explicit.

[8] Interogarea trebuie să fie „construită” anterior execuției sale. În cazul în care unele elemente ale interogării nu se cunosc decât la momentul execuției, aceasta poate fi parametrizată, urmând ca transmiterea valorilor lipsă să se facă în momentul în care sunt cunoscuți, fiind preluați direct de la utilizator sau dintr-un fişier.

5. Conectarea la sistemul de gestiune al bazei de date

Conectarea unei aplicații Java prin intermediul protocolului JDBC la sistemul de gestiune al bazei de date se poate realiza prin două clase:

- `DriverManager` – asigură accesul programului la o sursă de date specificată prin intermediul unui URL; atunci când se încearcă realizarea conexiunii este încărcat în mod automat orice driver JDBC 4.0 pe care îl găseşte în classpath;
- `DataSource` – metodă mai transparentă de acces la informații, un obiect având proprietăți specificate astfel încât să corespundă unor surse de date particulare;

În cadrul laboratorului vom folosi metoda de conectare la baza de date folosind clasa `DriverManager` întrucât este mai intuitivă. Atunci când un client indică un URL pentru a se conecta la o bază de date, clasa `DriverManager` apelează la interfața `Driver` pentru a identifica driver-ul necesar pentru interacțiunea cu sistemul de gestiune al bazei de date.

De obicei, URL-ul respectă următoarea structură:

```
protocol:subprotocol:[nume_baza_de_date][lista_de_proprietati]
```

Câteva exemple de URL-uri specifice anumitor tipuri de baze de date sunt:

```
jdbc:mysql://[host][,failoverhost[9]...][:port]/[database]
[?propertyName1][=propertyValue1][&propertyName2][=propertyValue2]...
jdbc:mysql://localhost:3306/librarie?user=root&password=*****

jdbc:derby:[subprotocol[10]:][databaseName][;attribute=value[11]]*
jdbc:derby:librarie;create=true

jdbc:oracle:[protocol]:@[database_host]:[port]:[instance]
jdbc:oracle:thin:@localhost:1521:orcl

jdbc:odbc:Driver={Microsoft Access Driver (*.mdb)};DBQ=path
jdbc:sqlserver://address\\server:port;database=databaseName;user=usr;password=pwd;
```

Metoda `getConnection`, disponibilă în clasa `DeviceManager` oferă un obiect conexiune (`Connection`) la baza de date, care poate fi folosit ulterior pentru diferite interogări:

```
String URL = "jdbc:mysql://localhost:3306/librarie";
Connection dbConnection = DriverManager.getConnection(URL);
```

În condițiile în care se citesc dintr-o interfață grafică cu utilizatorul informații de tip utilizator şi parolă, stocate, de exemplu în variabilele `usr` şi `pwd`, conectarea se poate face şi sub următoarea formă:

```
Connection dbConnection = DriverManager.getConnection(URL,usr,pwd);
```

```
Connection dbConnection =
      DriverManager.getConnection(URL+"?user="+usr+"&password="+pwd);
```

```
Properties connectionProperties = new Properties();
connectionProperties.put("user",usr);
connectionProperties.put("password",pwd);
Connection dbConnection = DriverManager.getConnection(URL,connectionProperties);
```

[9] Connector/J permite specificarea unei baze de date opționale la care să se încerce conectarea, dacă operația a eşuat în cazul bazei de date primare.
[10] Deşi în general este omis, parametrul `subprotocol` indică locația bazei de date (director din sistemul de fişiere, memorie, classpath, fişier .jar).
[11] În cadrul listei de atribute se poate specifica crearea bazei de date, criptarea acesteia, locația fişierelor în care să se păstreze diferite jurnale, numele de utilizator şi parola pentru conectare.

Atunci când driverele gestionate de interfața `Driver` recunosc URL-ul indicat drept parametru metodei `getConnection`, se stabileşte o legătură cu sistemul de gestiune pentru baza de date, întorcându-se o conexiune deschisă care poate fi utilizată pentru formularea de instrucțiuni JDBC translatate ulterior în interogări către baza de date.

Închiderea unei conexiuni, prin care sunt eliberate toate resursele asociate acesteia, se face prin metoda close():

```
dbConnection.close();
```

6. Interogarea bazei de date conform specificației JDBC

Tipurile de interogări pot fi, conform specificației JDBC, sunt:

- `Statement` – folosite pentru interogări SQL fără parametri;
- `PreparedStatement [extends Statement]` – folosite pentru interogări SQL precompilate care pot conține parametrii de intrare;
- `CallableStatement [extends PreparedStatement]`– folosite pentru a executa rutine stocate care pot conține parametrii de intrare şi de ieşire.

Un obiect de tip interogare (`Statement`) se obține prin metoda `createStatement` aplicabilă unui obiect de tip `Connection`:

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

Începând cu JDBC 4.1 există posibilitatea definirii conexiunii într-un bloc `try`-with-resources, ce eliberează toate resursele alocate în secțiunea respectivă în mod automat, indiferent dacă a fost generată sau nu o excepție `SQLException`:

```
try (Statement stmt = dbConnection.createStatement()) {
    ...
}
```

În continuare, obiectul de tip interogare poate fi utilizat pentru realizarea unei operații cu baza de date şi obținerea unui set de date rezultat în urma executării instrucțiunii. Există mai multe moduri prin care se poate realiza execuția unei interogări SQL:

- metoda `execute`: întoarce true dacă primul obiect al interogării este de tipul `ResultSet`; prin această metodă pot fi obținute unul sau mai multe (sau nici un) obiect(e) de tipul `ResultSet`; obiectele de tip `ResultSet` pot fi accesate apelând metoda `Statement.getResultSet`;

```
String query = "SELECT cnp, nume, prenume FROM clienti";
boolean result = stmt.execute(query);
if (result)
    ResultSet inregistrari = stmt.getResultSet();
```

- metoda `executeQuery`: întoarce **un singur obiect** de tip `ResultSet`;

```
String query = "SELECT COUNT(*) FROM facturi";
ResultSet result = stmt.executeQuery(query);
```

- metoda `executeUpdate`: întoarce un număr întreg având semnificația înregistrărilor afectate de expresia SQL; este folosită de regulă pentru instrucțiuni DML de tip `INSERT`, `UPDATE`, `DELETE`, dar şi pentru instrucțiuni de tip DDL precum `CREATE TABLE`, `ALTER TABLE`, `DROP TABLE`:

```
String query = "CREATE TABLE colectii (
                  id_colectie INT(10) UNSIGNED AUTO_INCREMENT PRIMARY KEY NOT NULL,
                  denumire VARCHAR(30) NOT NULL,
                  descriere VARCHAR(1000)
               )";
int result = stmt.executeUpdate(query);
```

```
String query = "INSERT INTO colectii VALUES ('Arta monumentala', '-');
int result = stmt.executeUpdate(query);
```

Interfața `ResultSet` pune la dispoziția utilizatorului o serie de metode pentru lucrul cu informațiile (seturile de date) obținute în urma interogării bazei de date. Obiectele având tipul `ResultSet` au anumite caracteristici care pot fi modificate între care tipul, gestiunea concurenței şi posibilitatea de deținere a cursorului. Caracteristicile pot fi precizate de utilizator în momentul creării unui obiect de tip interogare (`Statement`).

Cu privire la modalitatea în care poate fi manipulat cursorul (aspecte ce țin şi de senzitivitatea cursorului), există următoarele constante:

- `TYPE_FORWARD_ONLY` (implicit) – cursorul se poate muta doar înainte, ne-existând posibilitatea parcurgerii în ambele sensuri a setului de date obținut ca rezultat al interogării;
- `TYPE_SCROLL_INSENSITIVE` – cursorul se poate muta înainte şi înapoi, poziționându-se în diferite locații relative față de poziția curentă sau absolute, dar nu este afectat de modificările realizate de alți utilizatori în timp ce este utilizat; conține înregistrările care satisfac condițiile interogării atunci când aceasta este executată sau pe măsură ce sunt obținute entitățile;
- `TYPE_SCROLL_SENSITIVE` – cursorul se poate muta înainte şi înapoi, poziționându-se în diferite locații relative față de poziția curentă sau absolute, şi este afectat de modificările realizate de alți utilizatori.

Nu toate driverele JDBC implementează toate aceste tipuri ale obiectelor `ResultSet`. Se poate utiliza metoda `DatabaseMetaData.supportsResultSetType` pentru a verifica dacă tipul respectiv este suportat sau nu.

Tipul de concurență indică operațiile pe care utilizatorul are permisiunea de a le realiza:

- `CONCUR_READ_ONLY` (implicit) – utilizatorul are doar dreptul de a consulta informațiile, fără a le modifica;
- `CONCUR_UPDATABLE` – utilizatorul poate citi şi poate scrie informațiile reținute în setul rezultat.

Nu toate driverele JDBC implementează concurența obiectelor `ResultSet`. Se poate utiliza metoda `DatabaseMetaData.supportsResultSetConcurrency` pentru a verifica dacă această caracteristică este suportată sau nu.

Deținerea cursorului la realizarea tranzacțiilor[12] se face prin constantele:

- `HOLD_CURSORS_OVER_COMMIT` – cursorul nu este închis în momentul în care este apelată metoda `commit()`; un astfel de comportament este necesar atunci când obiectele de tip `ResultSet` sunt folosite mai mult pentru citire decât pentru scriere;
- `CLOSE_CURSORS_AT_COMMIT` – cursorul este închis după ce este apelată metoda `commit()`; un astfel de comportament poate genera performanțe mai bune pentru unele aplicații.

Comportamentul implicit referitor la deținerea cursorului în cazul tranzacțiilor depinde de sistemul de gestiune pentru baze de date şi poate fi verificat prin metoda `DatabaseMetaData.getResultSetHoldability()`.

[12] Această proprietate specifică comportamentul cursorului în momentul în care se apelează metoda `commit`.

Nu toate driverele JDBC implementează deținerea cursorului în cazul tranzacțiilor pentru obiecte de tipul `ResultSet`. Se poate utiliza metoda `DatabaseMetaData.supportsResultSetHoldability` spre a verifica dacă un anumit comportament este suportat sau nu.

În cazul în care se dorește modificarea ordinii în care sunt parcurse înregistrările, obiectul de tip `ResultSet` dispune de o metodă `setFetchDirection` prin care se sugerează direcția de obținere a tuplurilor corespunzătoare:

- `FETCH_FORWARD` (implicit) – de la prima înregistrare la ultima;
- `FETCH_REVERSE` – de la ultima înregistrare spre prima;
- `FETCH_UNKNOWN` – ordinea de parcurgere este necunoscută.

Un exemplu de creare a unui obiect de tip interogare realizat pentru obținerea unui set de date în care cursorul poate fi mutat în ambele direcții, dar nu poate fi modificat setul de date, menținând cursorul deschis după realizarea unei tranzacții prin metoda `commit` este:

```
Statement stmt = dbConnection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                              ResultSet.CONCUR_READ_ONLY,
                                              ResultSet.HOLD_CURSORS_OVER_COMMIT);
```

Un obiect de tip `ResultSet` conține mai multe (sau nici un) tuplu(ri), în funcție de condițiile interogării, având asociat un cursor care indică la orice moment rândul curent[13]. Câteva dintre metodele care pot fi utilizate pentru a realiza poziționări ale cursorului în cadrul setului de date sunt:

metoda	descriere
next()	mută cursorul pe înregistrarea următoare
previous()	mută cursorul pe înregistrarea precedentă
first()	mută cursorul pe prima înregistrare
last()	mută cursorul pe ultima înregistrare
beforeFirst()	mută cursorul înainte de prima înregistrare
afterLast()	mută cursorul după prima înregistrare
relative(int n)	mută cursorul la n poziții distanță față de poziția curentă
absolute(int n)	mută cursorul la poziția n (absolută) din set

De regulă, metodele întorc rezultate de tip `boolean`, având valoarea `true` dacă s-a reușit poziționarea dorită şi `false` în caz de eşec sau în situația în care setul de date nu conține înregistrări.

În cazul în care tipul cursorului este cel implicit (`TYPE_FORWARD_ONLY`), nu se poate apela decât metoda `next`.

Obținerea informațiilor (valorilor asociate atributelor) se realizează prin metode de tip *getter* (`getString`[14], `getInt`, `getByte`, `getBoolean`, `getBlob`, `getDate`) care pot primi ca parametru fie numele (respectiv aliasul) coloanei[15] fie indexul[16] ei în cadrul tabelei din baza de date.

---

[13] Inițial, cursorul se găseşte deasupra primului rând. Un astfel de cursor este diferit de obiectul de tip cursor definit în cadrul unei rutine stocate pe server.

[14] Metoda poate fi folosită pentru preluarea oricărui tip de informație din baza de date, mai puțin tipul SQL3.

[15] Metoda nu ține cont de capitalizarea şirului de caractere care este oferit drept parametru. Dacă există mai mult de un nume (sau alias) de coloană care are denumirea respectivă, este întoarsă valoarea corespunzătoare primului atribut identificat. Metoda ar trebui folosită în cazul în care numele (sau aliasurile) coloanelor sunt specificate explicit în interogare, nu şi în situația când interogarea are forma `SELECT * FROM ...`.

O rutină de parcurgere a înregistrărilor dintr-o bază de date poate fi:

```
ResultSet result = stmt.executeQuery ("SELECT denumire, cif FROM edituri");
while (result.next()) {
    String denumire = result.getString(1);
    float cif = result.getFloat("cif");
}
```

Procesul de actualizare a informațiilor într-o bază de date printr-un obiect de tip `ResultSet` este realizat în 2 etape:

- modificarea valorilor ce se doresc actualizate, la nivel de coloană, pe rândul unde se găseşte cursorul, prin intermediul metodelor de tip `updateString`, `updateInt`, `updateByte`, `updateBoolean`, `updateBlob`, `updateDate`; la acest moment, nici o modificare nu este realizată la nivelul tabelei;
- actualizarea rândului curent în care au fost marcate spre modificare valorile coloanelor prin intermediul metodei `updateRow()`;

Un exemplu de actualizare a informațiilor în baza de date este:

```
Statement stmt = dbConnection.createStatement
                             (ResultSet.TYPE_SCROLL_SENSITIVE,
                              ResultSet.CONCUR_UPDATABLE);
ResultSet result = stmt.executeQuery ("SELECT data, stare FROM facturi");
GregorianCalendar today = new GregorianCalendar();
today.setTime(new Date());
while (result.next()) {
    GregorianCalendar billDate = result.getDate(data);
    if (billDate.before(today))
        result.updateString(stare,'restanta');
    updateRow();
}
```

În cazul în care se doreşte anularea modificărilor realizate, se poate apela metoda `cancelRowUpdates()`, înainte însă de a apela metoda `updateRow`.

De asemenea, pentru introducerea de informații într-o bază de date folosind un obiect de tip `ResultSet`[17] se pot folosi metodele `moveToInsertRow()` (care mută cursorul la poziția corespunzătoare din setul de date[18]) urmată de specificarea atributelor în acelaşi mod ca pentru o oricare actualizare (folosind metode de tip `update...()`) pentru ca ulterior adăugarea să fie realizată prin metoda `insertRow()`.

```
Statement stmt = dbConnection.createStatement (ResultSet.TYPE_SCROLL_SENSITIVE,
                                               ResultSet.CONCUR_UPDATABLE);
ResultSet result = stmt.executeQuery ("SELECT * FROM scriitori");
result.moveToInsertRow();
result.updateString(1,"Delavrancea");
result.updateString(2,"Barbu");
result.insertRow();
```

După introducerea înregistrării în setul de date, este recomandată mutarea cursorului pe o altă poziție, întrucât realizarea altor operații asupra obiectului `ResultSet` pot avea rezultate imprevizibile în condițiile în care cursorul indică asupra valorii care a fost adăugată în tabelă. Frecvent, este apelată metoda `beforeFirst` astfel încât cursorul să se găsească în aceeaşi stare ca cea ulterioară creării sale.

O înregistrare poate fi ştearsă folosind metoda `deleteRow()`.

---

[16] Această metodă este mai eficientă. Numerotarea coloanelor începe de la 1.
[17] Aceasta reprezintă o alternativă la metoda clasică
[18] Acest tip de rând reprezintă în fapt o zonă de memorie în care poate fi construit un nou rând înainte de a-l adăuga propriu-zis la tabelă.

Atunci când nu toate datele interogării sunt cunoscute la momentul în care este compilată aplicația[19], există posibilitatea ca interogarea să fie generică, urmând a fi completată cu informații (provenite dintr-un fişier sau introduse chiar de către utilizator) atunci când ele sunt disponibile, şi anume la rulare, înainte de execuția interogării asupra bazei de date. În momentul în care acestea sunt create, ele primesc în mod necesar o parte din interogarea propriu-zisă, transmisă sistemului de gestiune al bazei de date care îl precompilează, astfel încât execuția sa va fi mai rapidă[20].

Sunt folosite obiecte de tip `PreparedStatement`, derivate din clasa `Statement`, informațiile necunoscute fiind specificate prin caracterul `?`:

```
String query = "UPDATE utilizatori SET tip = ? WHERE rol = ?";
PreparedStatement pstmt = dbConnection.prepareStatement(query);
```

Înainte de a executa o astfel de interogare, trebuie specificate valorile care corespund atributelor lipsă, lucru care se face prin metode de tip *setter*:

```
pstmt.setString(1, Integer.parseInt(buffer.readLine());
pstmt.setDate(2, Date.valueOf(buffer.readLine());
```

Execuția interogării se face folosind metodele specifice clasei `Statement`.

```
pstmt.executeUpdate();
```

În acest caz însă, metodele nu vor mai primi ca parametru comanda SQL, întrucât aceasta a fost deja asociată în momentul în care a fost creat obiectul de tip interogare parametrizată.

Rezultatul metodei `executeUpdate` este o valoare întreagă având semnificația numărului de înregistrări care au fost actualizate. Semnificația unui rezultat nul este acela că interogarea nu a afectat nici o înregistrare din tabelă sau că instrucțiunea a fost de tip DDL.

Pentru apelarea unei rutine stocate, se folosesc obiecte din clasa `CallableStatement`, derivate din `PreparedStatement`:

```
String query = "{? = CALL calculate_bill_value (?)}";
CallableStatement cstmt = dbConnection.prepareCall(query);
cstmt.registerOutParameter(1, java.sql.Types.DECIMAL[21]);
cstmt.setString(2,buffer.readLine());
cstmt.execute();
double result = cstmt.getDouble(1);
cstmt.close();
```

Valorile necunoscute (inclusiv rezultatul rutinei stocate, fie parametru, fie valoare întoarsă) sunt marcate în continuare prin caracterul `?`.

Pentru parametrii procedurilor, în cazul în care au tipul `IN` sau `INOUT`, trebuie specificată valoarea lor pentru ca rutina să poată fi executată. În plus, dacă aceştia au tipul `OUT` sau `INOUT`, trebuie specificat şi tipul de date aşteptat, folosind metoda `registerOutParameter`. Acelaşi comportament trebuie respectat şi pentru parametrii / rezultatele întoarse ale funcțiilor.

Execuția rutinei se face cu metoda `execute`, iar valorile întoarse sunt preluate indexat, prin metodele `get...()` corespunzătoare.

[19] Alte situații în care sunt folosite interogările parametrizabile este reutilizarea foarte frecventă a acestora.
[20] În momentul în care va fi executată o interogare parametrizabilă, ea va fi rulată de către sistemul de gestiune pentru baze de date fără a mai fi compilată.
[21] Tipurile de date din interfața `java.sql.Types` au acelaşi nume ca cele din MySQL.

7. Utilizarea tranzacţiilor în cazul accesului concurent la date

Mai multe comenzi SQL care nu produc un rezultat de tip `ResultSet` pot fi executate împreună în mod atomic[22]:

- `void addBatch(String sql) throws SQLException`
- `void clearBatch() throws SQLException`
- `int[] executeBatch() throws SQLException`

Metoda `executeBatch()` întoarce un vector care conține numărul operațiilor de tip actualizare realizate cu succes.

După metoda `executeBatch()` aplicată unui obiect interogare, se apelează şi metoda `commit()`[23], astfel încât modificările să fie vizibile în cadrul bazei de date.

```
dbConnection.setAutoCommit(false);
Statement stmt = dbConnection.createStatement
                             (ResultSet.TYPE_SCROLL_SENSITIVE,
                              ResultSet.CONCUR_UPDATABLE);
for (ArrayList<String> row:table) {
    String query = "INSERT INTO carti VALUES (";
    for (String column: row)
        query += column+",";
    query+= ")";
    dbConnection.addBatch(query);
}
int[] result = stmt.executeBatch();
dbConnection.commit();
dbConnection.setAutoCommit(true);
```

Utilizarea tranzacţiilor este şi un mecanism prin care este menţinută integritatea datelor, în contextul accesurilor concurente. Astfel, în timpul execuției unei tranzacții, sunt specificate drepturile de acces la nivelul tabelei pentru alți utilizatori care doresc să opereze pe acelaşi set de date. Acestea pot fi specificate prin metoda `setTransactionInsolation` aplicabilă unui obiect de tip `Connection`[24].

Nivel Izolare	Tranzacții	Citiri „murdare”	Citiri ne-repetabile	Citiri fantomă
`TRANSACTION_NONE`	nu	N/A	N/A	N/A
`TRANSACTION_READ_UNCOMMITTED`	da	permise	permise	permise
`TRANSACTION_READ_COMMITTED`	da	prevenite	permise	permise
`TRANSACTION_REPEATABLE_READ`	da	prevenite	prevenite	permise
`TRANSACTION_SERIALIZABLE`	da	prevenite	prevenite	prevenite

De obicei, nu trebuie modificat nivelul de izolare implicit, care este definit pentru fiecare sistem de gestiune pentru baze de date în parte.

---

[22] În caz contrar, excepția `BatchUpdateException` va fi generată. Această excepție va fi generată şi în cazul în care una dintre operațiile tranzacției nu a fost executată cu succes. Astfel, interogările incluse în cadrul unei tranzacții vor fi DDL (`CREATE TABLE`, `ALTER TABLE`, `DROP TABLE`), respectiv DML (`INSERT`, `UPDATE`, `DELETE`).

[23] Este important ca la începutul tranzacțiilor să se apeleze `Connection.setAutoCommit(false)` pentru a nu se produce modificări în baza de date până când acest lucru nu este specificat explicit prin metoda `commit()`. La sfârşitul tranzacțiilor se poate restabili comportamentul implicit (în care fiecare instrucțiune este considerată ca fiind o singură tranzacție), apelându-se `Connection.setAutoCommit(true)`.

[24] Unele drivere JDBC nu implementează toate nivelurile de izolare a unei tranzacții. Se foloseşte metoda `DatabaseMetaData.supportsTransactionIsolationLevel` spre a se verifica dacă este suportat nivelul în cauză.

Se consideră o citire „murdară" (*eng.* dirty read) acele valori ale atributelor care au fost actualizate dar pentru care nu s-a făcut încă commit pentru că există posibilitatea de a se reveni la valorile de dinaintea tranzacției.

O citire ne-repetabilă este aceea în care două tranzacții, A şi B operează asupra aceleiaşi înregistrări (una pentru citire, una pentru scriere) şi în care valorile furnizate sunt diferite.

Similar, o citire fantomă se obține în situația în care obținerea rezultatelor presupune satisfacerea unei condiții ce este îndeplinită ca urmare a actualizării astfel că o nouă interogare va furniza mai multe valori.

timp ↓	Tranzacția A	Tranzacția B
	read()	
		write()
	read()	

În contextul tranzacțiilor, se poate salva starea bazei de date înaintea realizării unor modificări, astfel încât dacă produc efecte nedorite la nivelul informațiilor din tabele, să se poată reveni la informațiile anterioare:

```
SavePoint state = dbConnection.setSavePoint();
...
dbConnection.rollback(state);
```

Metoda `rollback` încheie tranzacția curentă, astfel încât aceasta va fi apelată întotdeauna la sfârşitul tranzacției. De regulă, o astfel de operație trebuie utilizată numai atunci când s-a generat o excepție `SQLException` în tranzacția curentă, astfel încât nu se poate garanta care sunt valorile care au fost

O stare a bazei de date salvată poate fi eliminată din cadrul tranzacției folosind metoda `releaseSavePoint` a obiectului de tip `Statement` corespunzător.

8. Gestiunea informațiilor din dicționarul de date

JDBC permite accesarea informațiilor reținute în dicționarul de date, precum structura bazei de date şi a tabelelor precum şi restricțiile de integritate (chei primare, chei străine). Toate aceste date sunt puse la dispoziție prin clasa `DatabaseMetaData`, care se obține pornind de la obiectul `Connection` aferent bazei de date respective:

```
DatabaseMetaData dbMetaData = dbConnection.getMetaData();
```

Denumirea bazelor de date care pot fi accesate folosind conexiunea respectivă se obține cu metoda `getCatalogs`, care întoarce un obiect de tip `ResultSet` conținând câte o singură înregistrare pentru fiecare rând, şi anume denumirea catalogului.

Pentru fiecare bază de date pot fi aflate descrierile tabelelor componente, acestea putând fi filtrate în funcție de numele schemei sau al tabelei (oferindu-se modele pentru acestea) sau al tipului tabelei[25], prin metoda `getTables`. În cazul când se doreşte obținerea tuturor tabelelor dintr-o bază de date, toate aceste criterii pot fi marcate ca `null`, astfel încât se ignoră orice criteriu care ar fi putut limita rezultatele întoarse.

```
ResultSet getTables (String catalog, String schemaPattern, String tableNamePattern,
String[] types) throws SQLException
```

[25] Tipul tabelei poate fi `TABLE`, `VIEW`, `SYSTEM TABLE`, `GLOBAL TEMPORARY`, `LOCAL TEMPORARY`, `ALIAS`, `SYNONYM`.

Obiectul de tip `ResultSet` întors ca rezultat conține descrierile tabelelor, constând în următoarele informații:

1	**TABLE_CAT**	catalogul tabelei (poate fi `null`)
2	**TABLE_SCHEM**	schema tabelei (poate fi `null`)
3	**TABLE_NAME**	numele tabelei
4	**TABLE_TYPE**	tipul tabelei
5	**REMARKS**	comentariu explicativ asupra tabelei
6	**TYPE_CAT**	catalogul tipurilor (poate fi `null`)
7	**TYPE_SCHEM**	schema tipurilor (poate fi `null`)
8	**TYPE_NAME**	numele tipului (poate fi `null`)
9	**SELF_REFERENCING_COL_NAME**	numele identificatorului desemnat al unei tabele de un anumit tip (poate fi `null`)
10	**REF_GENERATION**	specifică modul în care sunt create valorile din `SELF_REFERENCING_COL_NAME` – `SYSTEM`, `USER`, `DERIVED` (poate fi `null`)

Structura unei tabele se obține prin metoda `getColumns()` în care filtrarea rezultatelor se face după numele schemei, a tabelei şi a coloanelor, omiterea oricăruia dintre criterii făcându-se prin marcarea ca `null` a parametrului aferent:

```
ResultSet getColumns (String catalog, String schemaPattern,
        String tableNamePattern, String columnNamePattern) throws SQLException
```

Obiectul de tip `ResultSet` întors ca rezultat conține descrierile tabelelor, constând în următoarele informații:

1	**TABLE_CAT**	catalogul tabelei (poate fi `null`)
2	**TABLE_SCHEM**	schema tabelei (poate fi `null`)
3	**TABLE_NAME**	numele tabelei
4	**COLUMN_NAME**	numele coloanei
5	**DATA_TYPE**	tipul de dată SQL (din `java.sql.Types`)
6	**TYPE_NAME**	numele tipului de dată (dependent de sursa de date)
7	**COLUMN_SIZE**	dimensiunea coloanei • valori numerice – precizia maximă • şiruri de caractere – lungimea (în caractere) • date calendaristice – lungimea reprezentării ca şir de caractere • reprezentare binară / tipul `ROWID` – dimensiunea (în octeți) • null – N/A
8	**BUFFER_LENGTH**	nu este utilizat
9	**DECIMAL_DIGITS**	numărul de zecimale; `null` dacă nu se aplică
10	**NUM_PREC_RADIX**	baza (de obicei 10 sau 2)
11	**NULLABLE**	indică posibilitatea de a exista valori `null` în coloană • columnNoNulls – ar putea să nu permită `null` • columnNullable – sigur permite `null` • columnNullableUnknown – stare necunoscută
12	**REMARKS**	comentariu ce descrie coloana (poate fi `null`)

13	COLUMN_DEF	valoarea implicită a coloanei[26] (poate fi null)
14	SQL_DATA_TYPE	nu este utilizat
15	SQL_DATETIME_SUB	nu este utilizat
16	CHAR_OCTET_LENGTH	pentru şiruri de caractere – numărul maxim de octeți dintr-o coloană
17	ORDINAL_POSITION	indexul coloanei în cadrul tabelei (începând de la 1)
18	IS_NULLABLE	indică posibilitatea de a exista valori null în coloană potrivit regulilor ISO
19	SCOPE_CATALOG	catalogul tabelului spre care indică referința atributului (null dacă DATA_TYPE nu este REF)
20	SCOPE_SCHEMA	schema tabelului spre care indică referința atributului (null dacă DATA_TYPE nu este REF)
21	SCOPE_TABLE	numele tabelului spre care indică referința atributului (null dacă DATA_TYPE nu este REF)
22	SOURCE_DATA_TYPE	sursa tipului de dată pentru un tip distinct sau pentru o referință generată de utilizator (null dacă DATA_TYPE nu este DISTINCT sau referință generată de utilizator)
23	IS_AUTOINCREMENT	indică dacă coloana este auto-incrementală
24	IS_GENERATED_COLUMN	indică dacă coloana este generată

Alte metode importante din clasa DatabaseMetadata sunt cele ce identifică rutinele stocate (funcții şi proceduri): getFunctionColumns şi getProcedureColumns, ambele având definiții similare:

```
ResultSet getFunctionColumns (String catalog, String schemaPattern,
         String functionNamePattern, String columnNamePattern) throws SQLException
```

```
ResultSet getProcedureColumns (String catalog, String schemaPattern,
         String procedureNamePattern, String columnNamePattern) throws SQLException
```

Pentru o rutină stocată se întorc mai multe intrări în ResultSet, corespunzând parametrilor de ieşire şi parametrilor de intrare.

Numele rutinei stocate poate fi obținut de pe poziția a treia, în timp ce următoarele câmpuri descriu parametrul în cauză: numele (4), tipul (5) – IN, OUT, INOUT, valoare întoarsă, tipul de dată din java.sql.Types asociat (6), numele tipului de dată (7), precizia (8), lungimea (9), scala (10), baza (11), proprietatea de a avea valori null (12), comentarii (13), lungimea şirului de caractere exprimată în octeți (14), poziția între parametrii rutinei stocate (15), proprietatea de a lua valori null conform regulilor ISO (16).

Cheile primare ale unei tabele[27] pot fi obținute prin metoda getPrimaryKeys:

```
ResultSet getPrimaryKeys (String catalog, String schema, String table)
          throws SQLException
```

Numele coloanei ce reprezintă cheia primară poate fi obținut de pe poziția a patra, împreună cu poziția pe care o ocupă în cheia primară compusă (într-un astfel de caz) ca şi denumirea pe care o are constrângerea de tip cheie primară.

[26] Pentru şirurile de caractere, aceasta va fi încadrată între caracterele ' şi '.
[27] În cazul în care nu se specifică o tabelă anume (parametrul corespunzător este null), vor fi întoarse toate cheile primare din baza de date respectivă.

De asemenea, se poate genera în mod automat un identificator unic pentru o tabelă, folosind metoda `getBestRowIdentifier`[28].

Constrângerile de tip cheie străină pot fi identificate în ambele sensuri, astfel că sunt definite metode ce identifică coloanele care referă cheia primară pentru o tabelă (`getExportedKeys`) şi metode prin care sunt specificate atributele referite de cheile primare ale altor tabele (`getImportedKeys`).

```
ResultSet getExportedKeys (String catalog, String schema, String table)
          throws SQLException
```

```
ResultSet getImportedKeys (String catalog, String schema, String table)
          throws SQLException
```

Metoda întoarce un set de date conținând descrierea cheii străine şi anume numele tabelei cheii primare referite (3), numele coloanei cheii primare referite (4), numele tabelei cheii străine (7), numele coloanei cheii străine (8), numărul de ordine în cazul cheilor străine compuse (9), regulile[29] în cazul operațiilor de tip `UPDATE` (10) şi `DELETE` (11), numele date constrângerilor de tip cheie străină (12) şi cheie primară (13) – dacă există şi posibilitatea ca evaluarea cheii străine să fie întârziată până la momentul operației `commit` (14).

Clasa `DatabaseMetaData` pune la dispoziție şi alte metode pentru verificarea capabilităților pe care le are driver-ul JDBC utilizat.

9. Tratarea excepțiilor de tip `SQLException`

În momentul când se produc erori în cazul interacțiunii cu o sursă de date este generată o excepție de tip `SQLException` care oferă următoarele informații:

❶ o descriere a erorii care poate fi obținută din metoda `getMessage` a obiectului de tip eroare asociat;

❷ un cod reprezentând starea SQL, potrivit standardizării ISO/ANSI şi OpenGroup (X/Open)[30], format din 5 caractere alfanumerice; acesta poate fi vizualizat ca rezultat al metodei `getSQLState` a obiectului de tip eroare asociat;

❸ o cauză, constând în unul sau mai multe obiecte de tip `Throwable` care au determinat excepția `SQLException`; lanțul cauzal poate fi parcurs recursiv apelând metoda `getCause` până când este returnată o valoare `null`;

```
Throwable t = ex.getCause();
while (t != null) {
    System.out.println("Cauza" : +t);
    t = t.getClause();
}
```

❹ referințe către alte excepții înlănțuite, în cazul în care s-a produs mai mult de o eroare; acestea pot fi obținute prin metoda `getNextException`.

Clasa `SQLException` are mai multe subclase, corespunzând unor excepții care sunt generate în situații particulare, ceea ce face procesul de gestiune a erorilor mult mai portabil.

---

[28] În acest caz, va trebui specificat şi un scop care precizează nivelul la care va fi utilizat identificatorul unic (ale cărui valori pot fi `bestRowTemporary`, `bestRowTransaction`, `bestRowSession`).
[29] Regulile pot avea valorile `importedNoAction` (nu permite realizarea de modificări asupra unei chei primare care este referită), `importedKeyCascade` (propagă modificările asupra cheii primare la nivelul cheii străine), `importedKeySetNull` (valoarea cheii străine se schimbă în `null` dacă valoarea cheii primare referite se schimbă), `importedKeySetDefault` (valoarea cheii străine devine cea implicită în cazul modificării cheii primare) şi `importedKeyRestrict` (la fel cu `importedNoAction`).
[30] Unele coduri au fost rezervate pentru producătorii de baze de date.

Avertismentele, reprezentate de obiecte din clasa `SQLWarning`, nu opresc execuția aplicației, informând totuşi utilizatorul ca una sau mai multe operații nu s-au desfăşurat aşa cum ar fi trebuit. Un avertisment poate fi raportat pentru obiecte de tip `Connection`, `Statement` (`PreparedStatament` / `CallableStatement`) sau `ResultSet`, fiecare dintre acestea dispunând de o metodă `getWarnings` care întoarce un rezultat de tip `SQLWarning`. În cazul în care nu este `null`, acesta dispune de o metoda `getNextWarning` ce indică şi alte avertismente[31].

Metodele pe care le pune la dispoziție clasa `SQLWarning` sunt: `getMessage`, `getSQLState` şi `getErrorCode`.

Cel mai frecvent avertisment este de tip `DataTruncation` ce indică faptul că tipul de date folosit pentru obținerea unui rezultat nu este cel corespunzător[32].

10. Alternative la manipularea informațiilor din surse de date

JDBC permite utilizarea unor obiecte de tip `RowSet`, derivate din `ResultSet`, care oferă programatorilor posibilitatea de a accesa datele mai uşor, având comportament[33] de componente JavaBeans. Astfel de obiecte sunt considerate conectate sau deconectate de la sursa de date, după cum mențin conexiunea (printr-un „driver”) la baza de date pe parcursul ciclului de viață. Un tip de obiect conectat este `JdbcRowSet` (care oferă o funcționalitate asemănătoare cu `ResultSet`) în timp ce tipurile de obiecte deconectate[34] sunt `CachedRowSet`, `WebRowSet`, `JoinRowSet` şi `FilteredRowSet` – acestea se vor conecta la sursa de date doar pentru operații de citire şi de scriere, situație în care vor trebui să verifice şi conflictele care pot apărea în astfel de situații.

Este recomandată folosirea obiectelor de tip `RowSet` atunci când sistemele de gestiune a bazelor de date nu implementează funcționalitatea de parcurgere sau actualizare a obiectelor de tip `ResultSet`, capabilități de care aceste clase dispun în mod implicit.

Obiectele `JdbcRowSet` pot fi create[35] folosind un obiect `ResultSet`, `Connection`, utilizând un constructor implicit sau dintr-o instanță a clasei `RowSetFactory`.

❶
```
Statement stmt = dbConnection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                              ResultSet.CONCUR_UPDATABLE);
ResultSet result = stmt.executeQuery("SELECT * FROM carti");
JdbcRowSet jdbcRS = new JdbcRowSetImpl(result);
```

Obiectul de tip `JdbcRowSet` este echivalent cu cel de tip `ResultSet`, având acelaşi conținut. În cazul când interogarea ar fi fost creată cu parametrii impliciți nici obiectul corespunzător nu ar fi putut fi parcurs, respectiv actualizat.

---

[31] Atunci când se execută o instrucțiune, avertismentele de la instrucțiunea precedentă se pierd în mod automat.

[32] Un astfel de obiect oferă posibilitatea investigării coloanei asupra căreia s-a produs eroarea, dacă aceasta corespunde unei operații de scriere sau de citire, câți octeți ar fi trebuit transferați şi câți au fost transferați efectiv.

[33] Comportamentul unor componente JavaBeans se referă la accesarea atributelor ca proprietăți precum şi la mecanismul de notificare, de fiecare dată când se modifică poziția cursorului, când sunt executate operații de adăugare, modificare, ştergere la nivelul unui rând dar şi atunci când se modifică conținutul obiectului respectiv. Aceste notificări sunt transmise tuturor obiectelor `RowSetListener` care au fost asociate obiectului de tip `RowSet`.

[34] Obiectele `RowSet` de tip deconectat au şi proprietatea că sunt serializabile ceea ce le face ideale pentru a fi transmise prin intermediul unei rețele.

[35] În toate aceste cazuri se va folosi clasa `JdbcRowSetImpl`.

❷
```
JdbcRowSet jdbcRS = new JdbcRowSetImpl(dbConnection);
jdbcRS.setCommand("SELECT * FROM carti");
jdbcRS.execute();
```

Obiectul de tip `JdbcRowSet` nu conține nici un fel de date până la momentul când nu îi este asociată o instrucțiune SQL prin metoda `setCommand`, apelată prin metoda `execute`. Implicit, un astfel de obiect poate fi parcurs, iar informațiile din el pot fi actualizate. Astfel de comportamente pot fi specificate însă şi explicit.

Metoda `execute` realizează conexiunea cu baza de date folosind parametrii conexiunii respective, execută interogarea aferentă proprietății `command`, citeşte informațiile din obiectul `ResultSet` reținut în obiectul de tip `JdbcRowSet`.

❸
```
JdbcRowSet jdbcRS = new JdbcRowSetImpl();
jdbcRS.setURL("jdbc:mysql://localhost:3306/librarie");
jdbcRS.setUsername(usr);
jdbcRS.setPassword(pwd);
jdbcRS.setCommand("SELECT * FROM carti");
jdbcRS.execute();
```

Pentru fiecare obiect `JdbcRowSet` se pot stabili proprietățile (`url`, `username`, `password`, `dataSourceName`). O interogare se poate specifica folosind metoda `setCommand`, iar execuția se face folosind `execute`, ca şi în cazul `ResultSet`.

❹
```
RowSetFactory RSfactory = RowSetProvider.newFactory();
JdbcRowSet jdbcRS = RSfactory.createJdbcRowSet();
jdbcRS.setURL("jdbc:mysql://localhost:3306/librarie");
jdbcRS.setUsername(usr);
jdbcRS.setPassword(pwd);
jdbcRS.setCommand("SELECT * FROM carti");
jdbcRS.execute();
```

Obiectul de tip `RowSetFactory` utilizează implementarea implicită, însă dacă driverul JDBC pune la dispoziție o implementare proprie, aceasta poate fi utilizată ca parametru transmis metodei `newFactory`. Interfața `RowSetFactory` conține metode spre a crea diferite implementări `RowSet`: `createJdbcRowSet`, `createCachedRowSet`, `createFilteredRowSet`, `createJoinRowSet`, `createWebRowSet`.

Un obiect `JdbcRowSet` creat folosind constructorul implicit va avea următoarele proprietăți:

- `type: ResultSet.TYPE_SCROLL_INSENSITIVE` (poate fi parcurs);
- `concurrency: ResultSet.CONCUR_UPDATABLE` (poate fi actualizat);
- `escapeProcessing: true` (poate fi definită o sintaxă care marchează faptul că există un tip de cod care va fi procesat de baza de date);
- `maxRows: 0` (nu există limită cu privire la numărul de înregistrări);
- `maxFieldSize: 0` (nu există limită cu privire la numărul de octeți pentru memorarea valorii unui atribut – aplicabil doar pentru atribute de tip `BINARY`, `VARBINARY`, `LONGVARBINARY`, `CHAR`, `VARCHAR` şi `LONGVARCHAR`);
- `queyTimeout: 0` (nu există nici o limită impusă referitoare la timpul de execuție al interogării);
- `showDeleted: false` (înregistrările şterse nu sunt vizibile);
- `transactionIsolation: Connection.TRANSACTION_READ_COMMITTED` (pot fi citite numai datele pentru care s-a apelat metoda `commit`);
- `typeMap: null` (tipul asocierii unui obiect `Connection` folosit de acest `RowSet` este `null`).

Modul în care pot fi manipulate datele folosind un obiect de tip `JdbcRowSet` este similar cu cel al obiectelor de tip `ResultSet`, metodele respective având aceeaşi sintaxă.

Interfața `CachedRowSet` desemnează modul de operare deconectat, din ea fiind derivate interfețele `FilteredRowSet`, `JoinRowSet` şi `WebRowSet`. Denumirea sa indică faptul că datele sunt reținute într-o zonă de memorie astfel încât procesările se fac pe acestea în loc de informațiile reținute în baza de date. Crearea unui obiect de tip `CachedRowSet` se face folosind constructorul implicit (`CachedRowSetImpl`) sau folosind o instanță a `RowSetFactory`.

Un obiect `CachedRowSet` conține implementarea implicită a `SyncProvider` ce este un obiect de tip `RIOptimisticProvider`. Acesta oferă un obiect `RowSetReader` şi un obiect `RowSetWriter` care sunt necesare atunci când trebuie citite sau scrise informații din sursa de date. Modul în care operează este transparent.

În cazul în care obiectul va fi folosit pentru actualizarea informațiilor din baza de date şi se doreşte ca modificările să fie operate şi la nivelul acesteia, trebuie specificate coloanele ce identifică în mod unic setul de înregistrări respective, indicând indexul atributelor corespunzătoare

```
int[] keys = {1};
cRS.setKeyColumns(keys);
```

Obiectul `CachedRowSet` nu este populat până nu este apelată metoda `execute`, moment în care obiectul `RowSetReader` asociat realizează coneciunea la baza de date folosind proprietățile corespunzătoare (`url`, `username`, `password`, `dataSourceName`), executând interogarea specificată în proprietatea `command`. După ce sunt obținute înregistrările necesare, conexiunea este închisă.

Pentru actualizarea informațiilor reținute (adăugare, modificare, ştergere) trebuie apelată metoda `acceptChanges()` pentru ca procesările să fie vizibile la nivelul sursei de date. În acest moment, obiectul `RowSetWriter` dechide conexiunea cu baza de date în care realizează operațiile respective după care conexiunea este închisă. În cazul în care este detectat un conflict (informațiile din sursa de date au fost actualizate între timp de un alt proces), este utilizată implementarea `RIOptimisticProvider` a clasei `SyncProvider` care foloseşte un model de concurență optimist, ce presupune faptul că nu vor exista conflicte sau că numărul acestora va fi redus. În situația în care nu sunt detectate conflicte, noile informații sunt transferate către baza de date, în cazul că există, actualizările sunt ignorate. Totuşi, există şi posibilitatea ca în cazul identificării unor conflicte, utilizatorul să poată selecta valorile reținute în baza de date:

```
try {
    cRS.acceptChanges();
} catch (SyncProviderException spe) {
    SyncResolver resolver = spe.getSyncResolver();
    while (resolver.nextConflict()) {
        if (resolver.getStatus() == SyncResolver.UPDATE_ROW_CONFLICT) {
            int conflictedRow = resolver.getRow();
            cRS.absolute(conflictedRow);
            int nbAttributes = cRS.getMetaData().getColumnCount();
            for (int k=1; k <= nbAttributes; k++) {
                if (resolver.getConflictValue(k) != null) {
                    Object cRSValue = cRS.getObject(k);
                    Object resolverValue = resolver.getConflictValue(k);
                    // ...
                    Resolver.setResolvedValue(k,...);
                }
            }
        }
    }
}
```

În cazul în care au fost detectate conflicte, metoda `acceptChanges` generează o excepție de tipul `SyncProviderException` care pune la dispoziție obiectul `SyncResolver`, un iterator pe conflictele identificate. De fapt, este un obiect `RowSet` care conține doar valorile conflictuale ale unei înregistrări, restul atributelor având valoarea `null`. Totodată, dispune de metode precum `getStatus` prin care se verifică tipul conflictului, `getRow` ce identifică indexul înregistrării la care se găseşte conflictul şi `getConflictedValue` ce reține valoarea care a fost actualizată anterior şi marcată ca atare în baza de date.

Actualizările dintr-un obiect de tip `CachedRowSet` pot fi notificate către alte obiecte care implementează interfața `RowSetListener`, ceea ce presupune definirea metodelor:

- `cursorMoved` – defineşte comportamentul obiectului ascultător în cazul când se produc modificări în privința cursorului obiectului `CachedRowSet`;
- `rowChanged` – defineşte comportamentul obiectului ascultător în cazul când unul sau mai multe atribute dintr-o înregistrare sunt modificate, când este adăugată sau ştearsă o înregistrare din obiectul `CachedRowSet`;
- `rowSetChanged` – defineşte comportamentul obiectului ascultător în cazul când obiectul `CachedRowSet` este populat cu informații.

Un obiect ascultător poate fi asociat unui set de date `CachedRowSet` prin metoda `addRowListener`. Oprirea notificărilor se face prin `removeRowListener`.

Obiectele de tip `FilteredRowSet` oferă posibilitatea de a limita numărul de înregistrări vizibile conform unui criteriu şi de a selecta informațiile ce pot fi consultate fără a realiza conexiuni la baza de date şi fără a opera modificări la nivelul interogării asociate.

Criteriul care indică înregistrările dintr-un obiect `FilteredRowSet` care vor fi vizibile este precizat printr-o clasă ce implementează interfața `Predicate`, indicând numele sau indexul coloanei după care se face filtrarea şi limitele între care trebuie să se găsească valorile. Pentru acesta, vor trebui specificate metodele `evaluate` (primind o valoare de comparat şi numele sau indexul coloanei sau un obiect de tip `RowSet`). Asocierea unui filtru (criteriu) pentru un obiect `FilteredRowSet` se face prin metoda `setFilter` care primeşte ca argument clasa care defineşte condițiile respective. Filtrarea propriu-zisă are loc atunci când este apelată metoda `next`, ducând la execuția metodei `evaluate` corespunzătoare. Există posibilitatea apelării mai multor filtre succesive prin apelarea metodei `setFilter` de mai multe ori, după ce anterior s-a produs selecția valorilor dorite prin metoda `next`. De asemenea, eliminarea tuturor filtrelor asociate se face apelând metoda `setFilter` cu parametrul `null`.

```
public class PriceFilter implements Predicate {
    private int lowValue, highValue;
    private String attName = null;
    private int attIndex = -1;
    public PriceFilter(int lowValue, int highValue, String attName) {
        this.lowValue = lowValue;
        this.highValue = highValue;
        this.attName = attName;
    }
    public PriceFilter(int lowValue, int highValue, int attIndex) {
        this.lowValue = lowValue;
        this.highValue = highValue;
        this.attIndex = attIndex;
    }
```

```
    public boolean evaluate (Object value, String attName) {
        boolean result = true;
        if (attName.equalsIgnoreCase(this.attName)) {
            int attValue = ((Integer)value).intValue();
            if (attValue >= this.lowValue && attValue <= this.highValue)
                return true;
            return false;
        }
        return result;
    }
    public boolean evaluate (Object value, int attIndex) {
        boolean result = true;
        if (attIndex == this.attIndex)) {
            int attValue = ((Integer)value).intValue();
            if (attValue >= this.lowValue && attValue <= this.highValue)
                return true;
            return false;
        }
        return result;
    }
    public boolean evaluate (RowSet RS) {
        boolean result = false;
        CachedRowSet cRS = (CachedRowSet)RS;
        int attValue = -1;
        if (this.attName != null)
            attValue = crs.getInt(this.attName);
        else if (this.attIndex > 0))
            attValue = cRS.getInt(this.attIndex);
        else
            return false;
        if (attValue >= this.lowValue && attValue <= this.highValue)
            result = true;
        return result;
    }
}
```

Operațiile de adăugare, modificare sau ştergere sunt permise numai dacă acestea nu contravin filtrelor asociate obiectului de tip `FilteredRowSet`.

Obiectele de tip `JoinRowSet` permit realizarea operației de asociere (`JOIN`) între obiecte `RowSet` care nu sunt conectate la sursa de date, astfel încât sunt economisite resursele necesare realizării uneia sau mai multor conexiuni.

Crearea unui obiect de tip `JoinRowSet` se face prin constructorul implicit[36] `JoinRowSetImpl`. Acesta nu va conține nici un fel de date până când nu sunt adăugate obiecte `RowSet`, specificându-se totodată şi atributul care serveşte drept legătură (cheie străină) în setul de date respectiv. Acest lucru se face prin metoda `addRowSet` care primeşte ca parametru un obiect `RowSet` şi indexul sau denumirea coloanei care indică relația între tabele. De asemenea, trebuie specificat şi tipul de asociere (`JOIN`) care se va realiza între tabele. Implicit, acesta este `INNER_JOIN`, însă metoda `setJoinType` poate primi drept parametrii şi următoarele tipuri: `CROSS_JOIN`, `FULL_JOIN`, `LEFT_OUTER_JOIN`, `RIGHT_OUTER_JOIN`. Alternativ, la crearea unui obiect `RowSet` care implementează interfața `Joinable`, pot fi precizate atributele care vor fi utilizate la realizarea asocierii prin metoda `setMatchColumn`, astfel încât atunci când sunt adăugate la `JoinRowSet` nu mai este necesară şi specificarea acestei proprietăți. Asocierea obținută va conține toate atributele seturilor de date din care este formată, astfel încât selectarea anumitor coloane se face „manual”, parcurgând obiectul `JoinRowSet` cu afişarea valorilor dorite.

[36] Există şi implementări specifice anumitor drivere JDBC, însă este posibil ca acestea să aibă denumiri şi comportamente diferite față de standard.

Pentru exemplul folosit, dacă se doreşte vizualizarea facturilor precum şi a conținutului acestora, se poate folosi un obiect `JoinRowSet`:

```
CachedRowSet bills = new CachedRowSetImpl();
bills.setURL("jdbc:mysql://localhost:3306/librarie");
bills.setUsername(usr);
bills.setPassword(pwd);
bills.setCommand("SELECT * FROM facturi");
bills.setMatchColumn("id_factura");
bills.execute();
CachedRowSet bill_details = new CachedRowSetImpl();
bill_details.setURL("jdbc:mysql://localhost:3306/librarie");
bill_details.setUsername(usr);
bill_details.setPassword(pwd);
bill_details.setCommand("SELECT * FROM detalii_factura");
bill_details.setMatchColumn("id_factura");
bill_details.execute();
JoinRowSet jRS = new JoinRowSetImpl();
jRS.addRowSet(bills);
jRS.addRowSet(bill_details);
```

Un obiect `WebRowSet` are capabilitatea de a fi reținut ca document XML şi totodată de a fi obținut din acest format. Întrucât limbajul XML este folosit ca standard, mai ales în comunicațiile între organizații, folosind servicii web, obiectul `WebRowSet` răspunde unor necesități reale.

Crearea unui obiect `WebRowSet` se face folosind constructorul implicit `WebRowSetImpl`. Acesta va dispune de un obiect `SyncProvider` care, spre diferență de implementarea standard, va avea asociat un obiect `RIXMLProvider` pentru a defini comportamentul în cazul unui conflict.

Reținerea unui obiect `WebRowSet` ca document XML se face fie folosind un obiect `OutputStream` (caz în care scrierea se face la nivel de octeți, suportând mai multe tipuri de date) sau un obiect `Writer` (caz în care scrierea se face la nivel de caractere).

```
java.io.FileOutputStream foStream = new java.io.FileOutputStream ("domains.xml");
domains.writeXml(fostream);
```

```
java.io.FileWriter fWriter = new java.io.FileWriter("domains.xml");
domains.writeXml(fWriter);
```

De asemenea, există posibilitatea populării dintr-un obiect `ResultSet` înainte de reținerea ca fişier XML:

```
domains.writeXml(rs, foStream);
domains.writeXml(rs, fWriter);
```

Similar, încărcarea conținutului unui document XML într-un obiect `WebRowSet` se face fie folosind un obiect `InputStream`, fie folosind un obiect `Reader`:

```
java.io.FileInputStream fiStream = new java.io.FileOutputStream ("domains.xml");
domains.readXml(fistream);
```

```
java.io.FileReader fReader = new java.io.FileReader("domains.xml");
domains.readXml(fReader);
```

Documentele XML asociate obiectelor de tip `WebRowSet` conțin, pe lângă datele propriu-zise[37], proprietăți (specificate în secțiunea `<properties> ... </properties>`) şi metadate (specificate în secțiunea `<metadata> ... </metadata>`) care conțin structura tabelei (`<column-count>` indică numărul atributelor, urmat de un număr corespunzător de secțiuni `<column-definition>` cu proprietățile lor).

[37] În cazul datelor se rețin atât valorile originale (cele care corespund celei mai recente consultări a bazei de date), cât şi valorile actualizate, astfel încât conflictele să poată fi detectate cu uşurință

Secțiunea `<data> ... </data>` conține, pentru fiecare înregistrare preluată din baza de date o secțiune `<currentRow> ... </currentRow>` cu un număr adecvat de elemente `<columnValue> ... </columnValue>`. În cazul în care valoarea este modificată, ea este urmată de o secțiune `<updateValue>`. Înregistrările adăugate, respectiv şterse sunt marcate prin secțiuni `<insertRow>`, respectiv `<deleteRow>`[38].

Activitate de Laborator

Se doreşte proiectarea unei aplicații care exploatează o bază de date ce urmează să fie integrată în cadrul unui sistem ERP pentru o librărie care comercializează doar cărți. Vom porni de la schema conceptuală, respectiv de la structura bazei de date construite în cadrul laboratorului anterior.

[0p] 1. În MySQL Workbench, să se execute script-ul `Laborator3.sql` unde se crează structura şi diferite obiecte ale bazei de date `libarie`.

În clasa Constants din pachetul general, să se modifice valoarea constantei `DATABASE_PASSWORD` cu cea corespunzătoare sistemului de gestiune pentru baze de date instalat pe maşina dumneavoastră.

[1p] 2. În clasa `DataBaseConnection` din pachetul `dataaccess`, să se implementeze metoda `getTableNumberOfRows` care determină numărul de înregistrări al unei tabele identificata prin nume (primit ca parametru).

```
public static int getTableNumberOfRows(String tableName)
```

Folosind metoda `getTableNumberOfRows`, să se determine numărul de înregistrări stocate în baza de date librărie.

[2p] 3. Folosind metoda `getTableContent`, să se creeze un fişier `books.txt` în care să se creeze lista tuturor cărților disponibile în librărie (existente în stoc). Pentru fiecare carte în parte se va preciza numele şi prenumele autorilor, titlul, editura precum şi anul apariției.

[1p] 4. Să se adauge în tabela `utilizatori` o înregistrare ale cărei atribute sunt introduse de la tastatură.

Pentru a citi o valoare de la tastatură în Java, se poate folosi următorul cod sursă:

```
BufferedReader buffer = new BufferedReader(new InputStreamReader(System.in));
value = buffer.readLine();
```

[1p] 5. Pe baza metodei `updateRecordsIntoTable`, să se modifice toate comenzile de aprovizionare către editura Aramis (`id_editura = 73, CIF = '247764320'`) astfel încât cantitățile să fie crescute cu 20%.

Modificările nu vor fi operate decât asupra comenzilor care nu au fost încă onorate (`stare='plasată'`).

[2p] 6. În clasa `DataBaseConnection` din pachetul `dataaccess`, să se implementeze metoda `deleteRecordsFromTable` care elimină dintr-o tabelă identificată prin nume (primit ca parametru) acele înregistrări care au anumite valori corespunzătoare unor atribute sau care respectă o anumită condiție.

```
public static void deleteRecordsFromTable(String tableName,
ArrayList<String> attributes, ArrayList<String> values, String whereClause)
throws Exception
```

[38] Structura documentului XML nu contează pentru utilizator, căci metodele `writeXml` / `readXml` operează în mod transparent.

Dacă `attributes = (attribute1, ..., attributen)`, respectiv `values = (value1, ..., valuen)`, atunci vor fi şterse înregistrările pentru care `attribute1=value1 AND ... AND attributen=valuen` sau pentru care este îndeplinită condiția `whereClause`. În cazul în care sunt precizate attributes/values, parametrul whereClause va fi ignorat.

Folosind metoda `deleteRecordsFromTable` astfel implementată să se şteargă acele edituri care nu au cărți comercializate de librarie.

[1p] 7. Folosind procedura stocată `calculate_user_total_bill_value` să se determine lista utilizatorilor împreună cu vânzările asociate fiecăruia.

[1p] 8. Folosind funcția stocată `calculate_supply_order_value` să se determine lista editurilor către care s-au efectuat cele mai mari plăți.

[1p] 9. În clasa `DataBaseConnection` din pachetul `dataaccess`, să se implementeze metoda `getReferrencedTables` ce determină pentru o tabelă dată prin nume (primit ca parametru) care sunt tabelele pe care le referă, precum şi atributele care fac obiectul constrângerii de tip `FOREIGN KEY`.

[1p] 10. Folosind un obiect de tip `RowSet` deconectat, să se afişeze cărțile comercializate de editură (precizându-se titlul şi prețul lor), numele editurii care le-a publicat, numele colecției şi numele domeniului din care fac parte.

[1p] 11. Să se filtreze colecția de date anterioară astfel încât să conțină numai cărțile având prețul cuprins între 1000 şi 5000 RON.

Bibliografie

[1] *JDBC Drivers* - http://www.ustudy.in/node/5475
[2] Jason HUNTER, William CRAWFORD – *Java Servlet Programming*
http://docstore.mik.ua/orelly/java-ent/servlet/ch09_02.htm
[3] Java Tutorial – JDBC™ Database Access
http://docs.oracle.com/javase/tutorial/jdbc/index.html