

11. JDBC: INTERACTIUNEA CU BAZE DE DATE

11.1. Concepte JDBC.....	2
11.2. Pasi de urmat in lucrul cu JDBC.....	2
11.3. Instalarea si incarcarea driverului.....	2
11.4. Deschiderea unei conexiuni cu baza de date.....	3
11.5. Crearea unui obiect Statement.....	3
11.6. Interrogarea bazei de date.....	4
11.7. Prelucrarea rezultatelor obtinute.....	4
11.7.1. Cazul interogarilor care modifica informatie (INSERT,UPDATE si DELETE).....	4
11.7.1.1. Obtinerea numarului de inregistrari afectate.....	4
11.7.1.2. INSERT: obtinerea valorilor cheilor generate automat.....	5
11.7.2. Cazul interogarilor care extrag informatie (SELECT).....	5
11.7.2.1. Clasa ResultSet.....	5
11.7.2.2. Pozitionarea cursorului si obtinerea pozitiei curente.....	6
11.7.2.3. Obtinerea numarului de randuri ale ResultSet-ului.....	6
11.7.2.4. Extragerea informatiei inregistrarii curente.....	7
11.7.2.5. Modificarea informatiei prin intermediul obiectului ResultSet.....	7
11.7.2.5.1. Conditii pentru reusita.....	7
11.7.2.5.2. Modificarea inregistrarii curente.....	7
11.7.2.5.3. Introducerea unei noi inregistrari.....	8
11.7.2.5.4. Stergerea unei inregistrari.....	8
11.7.3. Interrogari care produc rezultate multiple.....	8
11.8. Extragerea de meta-informatie.....	9
11.9. Tratarea warning-urilor si erorilor.....	10
11.10. Utilizarea obiectelor de tip RowSet.....	11
11.10.1. Principii. Tipuri de RowSet-uri.....	11
11.10.2. Ierarhia de interfete rowset.....	12
11.10.3. Modalitati de creare a unui obiect RowSet.....	12
11.10.4. Configurarea si popularea cu informatie a obiectelor RowSet.....	13
11.10.5. RowSet-uri de tip connected: obiecte JdbcRowSet.....	14
11.10.5.1. Crearea si popularea unui JdbcRowSet.....	14
11.10.5.2. Utilizarea obiectelor de tip JdbcRowSet.....	14
11.10.6. RowSet-uri de tip disconnected.....	15
11.10.6.1. Obiecte de tip CachedRowSet.....	15
11.10.6.1.1. Particularitati.....	15
11.10.6.1.2. Crearea obiectelor de tip CachedRowSet.....	15
11.10.6.1.3. Utilizarea obiectelor de tip CachedRowSet.....	16
11.10.6.2. Interfata WebRowSet.....	17
11.10.7. Filtrarea locala a datelor dintr-un rowset: interfata FilteredRowSet.....	17
11.10.8. Join-uri locale: interfata JoinRowSet.....	18
11.11. BIBLIOGRAFIE.....	19

11.1. Concepte JDBC

JDBC (Java Database Connectivity) este o infrastructura de clase gandita sa ofere acces la baze de date SQL si nu numai. API-ul JDBC poate fi folosit pentru accesarea oricarui fel de informatii tabulare, indiferent de modul in care acestea sunt stocate (sisteme de baze de date relationale (mysql, mSQL, Oracle etc), fisiere, JavaDB). Clasele JDBC se gasesc in pachetele **java.sql**, **javax.sql** si subpachetele acestora.

Independenta de softul de baze de date folosit se realizeaza prin intermediul unor drivere. JDBC dispune de clasa *DriverManager* care administreaza incarcarea driverelor si obtinerea conexiunilor catre softul de baze de date. Odata conexiunea deschisa, JDBC ofera clientului un API ce nu depinde de softul de baze de date folosit, ceea ce deschide calea usoarei migrari de la un DBMS¹ la altul.

11.2. Pasi de urmat in lucrul cu JDBC

Lucrul cu JDBC presupune urmatoarele aspecte:

- Incarcarea driverului corespunzator, pentru ca JDBC sa poata interfata cu DBMS-ul particular folosit de aplicatie
- Deschiderea conexiunii cu baza de date, ce presupune in general furnizarea unei combinatii user/parola si a unui set de optiuni ce guverneaza sesiunea clientului cu serverul de baze de date (criptarea datelor, compresia acestora etc)
- Stabilirea parametrilor rezultatelor pe care le furnizeaza driverul – determina modul in care programatorul va putea interacționa cu datele obtinute de la DBMS
- Formularea de interogari de diferite tipuri
- Prelucrarea rezultatelor obtinute, ce difera in functie de tipul interogarii

Toate aceste aspecte vor fi detaliate in continuare.

11.3. Instalarea si incarcarea driverului

Programatorul lucreaza cu JDBC folosind un API standard; pentru a comunica cu sistemul de stocare a datelor, JDBC foloseste drivere, ce transformaapelurile programatorului in comenzi specifice respectivului protocol/sistem de stocare. Driverele pot fi scrise integral in Java (drivere care implementeaza protocolul de retea pentru anumite tipuri de servere – ex: mysql) sau doar partial, insa in acest al doilea caz nu mai avem independenta de platforma. O lista de drivere existente poate fi gasita aici: <http://developers.sun.com/product/jdbc/drivers>.

Procedura de instalare a driverului poate diferi de la un driver la altul. De exemplu, in cazul driverului Mysql, driverul este o arhiva jar si este suficiente reconfigurarea CLASSPATH-ului sa pointeze si catre aceasta arhiva. Intr-un mediu de dezvoltare, driverul poate fi specificat ca biblioteca atasata proiectului sau poate fi deja disponibil; spre exemplu, versiunile noi de NetBeans contin suport implicit pentru MySQL.

Incarcarea driverului se poate face in doua feluri: prin folosirea lui *DriverManager* sau a unui *DataSource*. Iata prima varianta in cazul driverului mysql:

```
Class.forName("com.mysql.jdbc.Driver");
```

Metoda *Class.forName()* incarca sa localizeze clasa (dat fiind numele acesteia) si sa o incarce in masina virtuala. Clasa *Driver* are un bloc de initializare static care se executa la incarcarea ei, in care este apelata metoda *DriverManager.registerDriver* pasandu-i-se ca argument o instanta a clasei driver. In acest fel driverul este inregistrat automat in *DriverManager* si din acest moment se pot deschide conexiuni cu baza de date MySQL.

1 DBMS – Database Management System. Este softul care depoziteaza datele, le administreaza si ofera clientilor posibilitatea accesarii si modificarii lor. Exemple: MySQL, Oracle, PostgreSQL, MsSQL etc.

Studentul poate utiliza prezentul material si informatiile continute in el exclusiv in scopul asimilarii cunostintelor pe care le include, fara a afecta dreptul de proprietate intelectuala detinut de autor.

11.4. Deschiderea unei conexiuni cu baza de date

Deschiderea unei conexiuni se realizeaza folosind metoda **DriverManager.getConnection(String url)**. URL-ul este de forma **jdbc:subprotocol:resursadorita**. Iata cateva exemple:

```
// pentru MySQL
Connection c1 = DriverManager.getConnection("jdbc:mysql://localhost:3306/numebazadate");

// pentru apache Derby/Java DB
Connection c1 = DriverManager.getConnection("jdbc:derby:numebazadatede");
```

In functie de cel de-al doilea camp al URL-ului primit (cel de subprotocol), *DriverManager* incercă sa identifice driverul corect pentru sistemul de baze de date respectiv, iar in caz de esec va arunca o exceptie (ex: *java.sql.SQLException: No suitable driver found for jdbc:undriver:inexistent*). Driverul implementeaza protocolul de comunicatie cu serverul de baze de date, care difera de la un DBMS la altul.

Cand conectarea la sursa de date presupune autentificare, exista o a doua forma a metodei *getConnection()*, care accepta doi parametri suplimentari – username si parola:

```
Connection c1 = DriverManager.getConnection("jdbc:mysql://localhost/bazadatede", "user1", "pass1");
```

11.5. Crearea unui obiect Statement

Pentru interogarea bazei de date se folosesc obiecte de tip *Statement*. Acestea sunt cele ce dispun de metode pentru interogare si extragerea rezultatelor produse. Optiunile folosite la crearea obiectului *Statement* influenteaza decisiv modul in care programatorul va putea obtine/modifica/prelucra seturile de rezultate produse de interogarile sale.

JDBC suporta trei tipuri de statement, sub forma a 3 interfete:

- **Statement** - interfata ce descrie capabilitatile generale ale oricarui obiect statement. Reprezinta interfata parinte pentru celelalte doua
- **PreparedStatement** - interfata ce descrie capabilitatile obiectelor de tip interogare precompilata. Astfel de obiecte sunt folosite pentru a eficientiza lucrul cu interogari repetitive prin compilarea lor o singura data urmata de executia multipla
- **CallableStatement** - interfata implementata de obiectele utilizate pentru executia rutinelor stocate in baza de date, care pot produce unul sau mai multe seturi de rezultate

Materialul de fata se va concentra pe capabilitatile generale ale statement-urilor cuprinse in interfata *Statement*.

Obiectele de tip *Statement* pot fi create cu metoda supraincarcata *createStatement()* a clasei *Connection*:

```
Statement s = c.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);

// ...sau varianta fara argumente, echivalent cu a specifica TYPE_FORWARD_ONLY si
// CONCUR_READ_ONLY:
Statement s = c.createStatement();
```

In varianta cu argumente, primul parametru al metodei *createStatement()* determina modul in care poate fi parcurs setul de rezultate produs de viitoarele interogari SELECT, sub forma unor constante din clasa *ResultSet*:

- **TYPE_SCROLL_INSENSITIVE** – setul de rezultate produs poate fi parcurs in ambele directii (inainte si inapoi) dar nu reflecta modificarile aduse intre timp de alti clienti ai bazei de date din care au provenit inregistrarile
- **TYPE_SCROLL_SENSITIVE** – acelasi lucru ca mai sus, insa schimbarile produse de alti clienti vor fi reflectate in continutul setului de rezultate pe masura ce acesta este parcurs
- **TYPE_FORWARD_ONLY** – setul de rezultate poate fi parcurs o singura data si intr-o singura directie, de la inceput la sfarsit

Cel de-al doilea argument al metodei *createStatement()* specifica daca seturile de rezultate produse de interogarile SELECT sunt actualizabile sau nu (modificarea lor atragand dupa sine modificararea informatiei de pe serverul de baze de date!). Valori posibile:

- **CONCUR_UPDATABLE** – setul de rezultate este actualizabil. Clientul il poate folosi pentru a produce modificari in baza de date – introducere de noi inregistrari sau operatii de tip UPDATE sau DELETE aplicate inregistrarilor din result set si care vor fi salvate in baza de date de origine

Atentie! Pentru ca setul de rezultate sa fie actualizabil este necesar ca acesta sa provina dintr-o singura tabela si printre coloanele sale sa se gaseasca si cheia primara a tablei!

- **CONCUR_READ_ONLY** – setul de rezultate nu poate fi modificat, ci este o simpla copie a datelor de pe server

Nota: in lipsa oricaror optiuni, seturile de rezultate produse de interogarile SELECT vor fi implicit de tip FORWARD_ONLY si CONCUR_READ_ONLY.

11.6. Interogarea bazei de date

In scopul interogarii, clasa *Statement* ofera metode precum:

- **ResultSet executeQuery(String sql)** – pentru executarea unei instructiuni SQL ce produce un set de rezultate (SELECT). Metoda intoarce un obiect de tip *ResultSet* ce poate fi apoi parcurs si eventual modificat

Atentie! Fiecare Statement poate avea un singur ResultSet deschis la un moment dat. La apelarea oricarei metode de executie a unei interogari, daca exista un ResultSet precedent, acesta va fi inchis.

```
// executarea interogarii si memorarea referintei catre rezultate
ResultSet rs = s.executeQuery("SELECT * FROM firme");
```

- **int executeUpdate(String sql)** – metoda gandita pentru interogari SQL de tipul UPDATE,INSERT, DELETE, sau pentru cele care nu produc rezultate (ex: instructiunile DDL, de manipulare a structurii). Metoda returneaza un intreg ce reprezinta numarul de linii afectate (modificate) in urma operatiunii.

```
// o interogare UPDATE care corecteaza prima litera a unui nume de firma
String sql = "UPDATE firme SET Nume='InfoAcademy' where Nume='infoacademy'";
int randuriModificate = s.executeUpdate(sql);
```

11.7. Prelucrarea rezultatelor obtinute

11.7.1. Cazul interogarilor care modifica informatie (INSERT,UPDATE si DELETE)

11.7.1.1. Obtinerea numarului de inregistrari afectate

Interogarile de acest tip nu produc seturi de inregistrari, insa pe de alta parte afecteaza una sau mai multe inregistrari din baza de date. Numarul de inregistrari afectate nu poate fi intotdeauna cunoscut din momentul formularii interogarii – spre exemplu, dorim sa modificam pretul tuturor produselor dintr-o anumita categorie, sau dorim sa stergem doar produsele a caror data de expirare a trecut. Chiar si in cazul interogarilor INSERT ce adauga o singura inregistrare, numarul de randuri afectate poate fi 0, 1 sau chiar 2! De aceea serverul SQL raporteaza intotdeauna numarul de inregistrari afectate de o interogare INSERT, UPDATE sau DELETE, iar JDBC ofera programatorului modalitati de a accesa aceasta informatie.

Pentru a executa instructiuni ce modifica informatie se foloseste metoda **executeUpdate(String sql)** a clasei *Statement*, care returneaza un intreg reprezentand numarul de randuri afectate:

```
// scumpirea produselor cu pretul sub 100 RON
int n = s.executeUpdate("UPDATE Produse SET Pret=Pret*1.10 WHERE Pret<100");
System.out.println("A fost modificat pretul pentru "+n+" produse");
```

11.7.1.2. INSERT: obtinerea valorilor cheilor generate automat

Tabelele SQL pot contine coloane pentru care valoarea este generata automat de catre serverul de baze de date la introducerea unei noi inregistrari (ex: coloanele AUTO_INCREMENT din MySQL). In astfel de cazuri, programatorul are deseori nevoie sa afle care a fost valoarea alocata de server pentru o putea folosi in interogari ulterioare. In masura in care driverul JDBC o permite, valorile cheilor autogenerate pot fi extrase folosind metode ale clasei *Statement*.

Clasa Statement ofera urmatoarele metode de interes:

- **ResultSet getGeneratedKeys()** - extrage valorile cheilor generate la executia instructiunii anterioare sub forma unui *ResultSet* (vezi mai jos modalitatea de parcursare a unui *ResultSet*). Lista de chei furnizate este aleasa fie de catre programator (vezi metodele urmatoare), fie in mod automat de catre driver
- **execute/executeUpdate(String sql, int autoGeneratedKeys)** – executa o instructiune INSERT indicand driverului ca doreste sau nu obtinerea cheilor autogenerate. In acest sens, valoarea celui de-al doilea parametru poate fi *Statement.RETURN_GENERATED_KEYS* sau *Statement.NO_GENERATED_KEYS*
- **execute/executeUpdate(String sql, String[] numeColoane)** – executa instructiuni INSERT indicand driverului numele coloanelor autogenerate a caror valoare trebuie facuta disponibila programatorului. In acest fel, programatorul poate alege lista exacta de chei de interes
- **execute/executeUpdate(String sql, int[] pozitiiColoane)** – acelasi efect ca mai sus, dar cu specificarea coloanelor de interes sub forma pozitiei lor in cadrul tabelei

11.7.2. Cazul interogarilor care extrag informatie (SELECT)

11.7.2.1. Clasa ResultSet

Spre deosebire de interogarile INSERT, UPDATE si DELETE, interogarea de tip SELECT produce un set de inregistrari. In JDBC, acestea sunt disponibile programatorului sub forma de obiecte ale clasei *ResultSet*. Obiectul mentine un cursor intern, care, in functie de tipul *ResultSet*-ului (scrollable sau forward-only) poate fi deplasat in ambele directii sau numai inainte, permitand accesarea inregistrarilor continute si a informatiilor componente ale acestora.

Modalitatea principală de a obtine un *ResultSet* este de a executa o interogare de tip SELECT folosind metoda *executeQuery()* a unui obiect *Statement*. Exista si o alta modalitate de a obtine *ResultSet*-uri, atunci cand se folosesc interogari multiple (vezi API-ul clasei *Statement*).

Dupa crearea sa, *ResultSet*-ul ramane conectat la baza de date, in sensul in care:

- el poate reflecta modificarile aduse intre timp in baza de date, pe masura ce este parcurs de catre programator. Acest lucru se aplica numai result set-urilor de tip *TYPE_SCROLL_SENSITIVE*
- inregistrarile continute in cadrul sau pot fi modificate, schimbarile fiind salvate inapoi in baza de date prin intermediul metodelor puse la dispozitie de catre insusi *ResultSet*-ul (nu prin interogari separate)
- prin intermediul metodelor lui pot fi introduse inregistrari noi in baza de date sau sterse inregistrari existente

Programatorul poate efectua urmatoarele tipuri de operatii cu un *ResultSet*:

- pozitionarea cursorului intern al *ResultSet*-ului
- obtinerea pozitiei curente
- extragerea si prelucrarea datelor continute
- modificarea inregistrarilor continute (cu trimitera automata a modificarilor inapoi catre baza de date)
- stergerea uneia sau mai multora dintre inregistrarile continute (si, ca urmare, stergerea randurilor corespunzatoare din baza de date)
- introducerea de noi inregistrari
- extragerea setului de caracteristici ale *ResultSet*-ului (meta-informatia acestuia)

11.7.2.2. Pozitionarea cursorului si obtinerea pozitiei curente

Toate metodele *ResultSet*-ului care extrag sau modifica date actioneaza asupra inregistrarii curente, de aceea orice operatie de citire/modificare trebuie precedata de o pozitionare a cursorului pe inregistrarea dorita. Pozitia cursorului intern al unui *ResultSet* poate fi modificata apelând metode ale obiectului în cauză. Pozitii posibile pentru cursor sunt: înaintea primei inregistrări, în dreptul uneia dintre inregistrări, sau după ultima inregistrare (cu alte cuvinte, el nu poate fi plasat *intre* inregistrări, dar se poate afla înaintea sau după întregul ansamblu). Cursorul se află initial înaintea primei inregistrări a *ResultSet*-ului; el poate fi deplasat cu cate o pozitie sau cu mai multe, înainte sau înapoi, în funcție de tipul result-set-ului.

Atentie! În funcție de opțiunile specificate la crearea obiectului *Statement*, *ResultSet*-urile produse de acesta pot fi parcuse intr-o singura direcție (de la început la sfârșit) sau în ambele. De aceea o parte dintre metodele enumerate mai jos vor eşua dacă *ResultSet*-ul este de tip *TYPE_FORWARD_ONLY* sau/si *CONCUR_READ_ONLY*.

Atentie! Randurile și coloanele dintr-un *ResultSet* se numerotează de la 1!

Iată câteva metode utilizate pentru pozitionarea cursorului:

- **first(), last(), next(), previous()** – pozitionează cursorul pe prima/ultima/urmatoarea/precedenta inregistrare. Toate aceste metode returnează *boolean: true* dacă pozitionarea s-a efectuat și *false* dacă s-a ajuns la una dintre extremități (sau dacă *ResultSet*-ul nu conține randuri, ceea ce este echivalent). Apelând *next()* cand cursorul se află pe ultima inregistrare il va plasa după aceasta; analog, cand cursorul se află pe prima pozitie, *previous()* il va plasa înaintea ei. Astfel, un result set poate fi parcurs întotdeauna de la început la sfârșit, indiferent de tipul sau, folosind o buclă *while* după cum urmează:

```
ResultSet r = s.executeQuery("SELECT * FROM produse");
while(r.next()) {
    // prelucrare date
}
```

- **beforeFirst(), afterLast()** – pozitionarea cursorului la extremitatile *ResultSet*-ului (înaintea primei inregistrări sau după ultima), astfel încât acesta să fie pregătit pentru apelarea metodelor *next()*/*previous()*
- **absolute(int pozitie), relative (int nrRanduri)** – pozitionare absolută în cadrul result set-ului sau relativă la randul curent. În cazul pozitionării relative (dacă *ResultSet*-ul o permite), numărul de randuri poate fi negativ, ceea ce permite “întoarcerea” cu un număr de pozitii
- **getRow()** – returnează numărul randului curent

11.7.2.3. Obtinerea numarului de randuri ale *ResultSet*-ului

In mod surprinzător poate, clasa *ResultSet* nu dispune de o metodă care să raporteze numărul de inregistrări continue într-un obiect de acest tip. Acest lucru se întâmplă deoarece datele nu sunt aduse pe client toate odata, ci pe masura ce acesta le accesează. Solutia minimizează atât memoria ocupată pe client, cât și traficul prin rețea atunci când clientul și serverul rulează pe mașini diferite.

Atunci când este totuși necesar numărul de randuri ale unui *ResultSet*, informația poate fi obținută indirect:

- prin efectuarea unei singure interogări, folosind un artificiu, în cazul *ResultSet*-urilor de tip scrollable (care pot fi parcuse în ambele sensuri): se pozitionează cursorul pe ultima inregistrare și se extrage numărul randului curent:

```
ResultSet r = s.executeQuery("SELECT * FROM Produse WHERE Pret<100");
r.last();
int nrRanduri = r.getRow();
```

- prin efectuarea a două interogări separate, în cazul *ResultSet*-urilor de tip *FORWARD_ONLY*. Acestea pot fi parcuse într-un singur sens și o singură dată. Se poate proceda în cel puțin două moduri în privința celor două interogări:

- una produce numarul de inregistrari folosind o functie de agregare (ex: COUNT(*) in MySQL) si alta le extrage
- aceeasi interogare executata de doua ori. La prima executie se parcurge result set-ul si apoi se apeleaza `getRow()`, iar la a doua se parcurge `ResultSet`-ul si se extrag datele

11.7.2.4. Extragerea informatiei inregistrarii curente

Odata pozitionat cursorul pe inregistrarea dorita, datele componente ale acesteia pot fi extrase folosind metode de tip getter. Exista metode getter pentru toate tipurile de date: `getBoolean()`, `getByte()`, `getDate()`, `getDouble()`, `getFloat()`, `getInt()`, `getLong()`, `getShort()`, `getString()` etc. Metodele returneaza informatia dorita, returnand tipul de date specificat in numele metodei (ex: `getInt()` va produce o valoare de tip *int*). Fiecare dintre metode primeste un singur argument care desemneaza coloana a carei informatie se extrage. Coloana dorita poate fi specificata sub forma de:

- pozitie in lista de coloane din `ResultSet`. **Atentie! Coloanele se numeroteaza si ele de la 1!**
- nume al coloanei asa cum apare el in definitia tablei de origine

Exemplu: data fiind tabela Produse cu coloanele Denumire si Pret, putem afisa denumirea primului produs in felul urmator:

```
ResultSet r = s.executeQuery("SELECT Denumire, Pret FROM Produse");
r.first();
System.out.println(r.getString("Denumire"));

// aceeasi valoare putea fi obtinuta referind coloana prin intermediul pozitiei sale:
System.out.println(r.getString(1));
```

11.7.2.5. Modificarea informatiei prin intermediul obiectului `ResultSet`

11.7.2.5.1. Conditii pentru reusita

Pentru ca un result set sa fie actualizabil este necesar sa fie indeplinite urmatoarele conditii:

- driverul JDBC folosit trebuie sa suporte facilitatea de actualizare
- tabela de provenienta a datelor trebuie sa contine cheie primara
- interogarea SELECT care produce result set-ul:
 - trebuie sa extraga date dintr-o singura tabela, cu corespondenta 1 la 1 intre randurile result set-ului si cele ale tablei (aceasta exclude, de exemplu, folosirea functiilor de agregare in interogarea SELECT). Atunci cand datele ar proveni dintr-un join, result set-ul nu ar sti in ce tabele sa produca modificarile
 - trebuie sa selecteze si cheia primara a tablei. Masura este necesara pentru ca result set-ul sa poata referi in mod unic inregistrari din tabela atunci cand doreste sa le modifice/stearga
 - trebuie sa selecteze toate coloanele care, prin definitia lor, sunt obligate sa contin date (cele declarate ca NOT NULL si fara valoare default)
 - datele trebuie sa provina direct din coloanele tablei, fara a fi prelucrate folosind expresii. Aceasta deoarece, in caz contrar, result set-ul ar trebui sa deduca formula inversa pentru a seta valoarea coloanei (exemplu: daca result set-ul a fost produs cu interogarea `SELECT SQRT(Numar) as Radical FROM Numere`, atunci cand programatorul actualizeaza coloana `Radical` ar trebui ca resultset-ul sa ridice numarul oferit de programator la patrat si sa memoreze valoarea rezultata in tabela! si acesta este un caz simplu...) Serverul de baze de date nu poate avea "inteligenta" necesara pentru a deduce inversul unei expresii oarecare
- result set-ul trebuie sa fie de tip CONCUR_UPDATABLE. Chiar daca toate conditiile de mai sus sunt indeplinite, result set-ul nu va putea fi actualizat daca el a fost creat prin intermediul unui `Statement` care genereaza numai rezultate read-only

11.7.2.5.2. Modificarea inregistrarii curente

ResultSet-ul "pastreaza legatura" cu baza de date din care a provenit, in sensul in care datele pot circula si in sens invers: folosind metode ale obiectului `ResultSet`, programatorul poate modifica inregistrari din baza de date. In mod normal, acest lucru ar fi fost posibil numai trimitand interogari de tip UPDATE separate catre serverul de baze de date.

Pentru a modifica campurile unei inregistrari sunt necesari urmatorii pasi:

- pozitionarea pe inregistrarea dorita
- folosirea metodelor **updateInt()**, **updateString()** etc pentru schimbarea valorilor coloanelor. Metodele corespund getterilor enumerati la capitolul anterior si primesc doua argumente: coloana si noua valoare. Si aceste metode sunt supraincarcate, coloana putand fi specificata ca int (pozitia) sau ca String (numele). **Atentie! Simpla apelare a metodelor update...() nu produce modificari in baza de date, ci doar pe client!**
- la incheierea modificarilor coloanelor trebuie apelata metoda **updateRow()** a result set-ului care salveaza modificarile in baza de date

Daca se doreste anularea unui update inceput (s-au apelat deja una sau mai multe metode *update...()* dar nu s-au salvat modificarile folosind *updateRow()*) poate fi folosita metoda **cancelRowUpdates()**. Anularea are loc automat daca cursorul este pozitionat pe o alta inregistrare fara sa se fi apelat *updateRow()*.

11.7.2.5.3. Introducerea unei noi inregistrari

Obiectul *ResultSet* dispune de un rand special numit "insert row", care, odata populat cu informatii, poate fi adaugat in *ResultSet*, determinand introducerea unei noi inregistrari in tabela din care provine *ResultSet*-ul. Insert row este un rand cu regim special, deoarece cursorul nu poate fi plasat pe el folosind metodele obisnuite de pozitionare.

Pasii de urmat pentru a introduce o noua inregistrare folosind un *ResultSet* sunt urmatorii:

- pozitionarea pe "insert row" - se realizeaza folosind metoda **moveToInsertRow()** a result-set-ului
- popularea coloanelor noii inregistrari, folosind aceleasi metode *update...()* de la sectiunea anterioara
- introducerea noii inregistrari in *ResultSet* si in baza de date prin apelarea metodei **insertRow()**

Ca si in cazul actualizarii inregistrarilor, mutarea cursorului pe alt rand inaintea apelarii lui *insertRow()* nu produce efecte in baza de date (noua inregistrare nu va fi introdusa).

```
// exemplu: introducerea unei noi inregistrari in tabela Produse
ResultSet r = s.executeQuery("SELECT * FROM Produse");
r.moveToInsertRow();
r.updateString("Denumire", "Laptop");
r.updateDouble("Pret", 1899.99);
r.insertRow();
```

11.7.2.5.4. Stergerea unei inregistrari

Clasa *ResultSet* pune la dispozita programatorului metoda **deleteRow()** care sterge randul curent din *ResultSet* si din baza de date:

```
// exemplu: stergerea ultimei inregistrari
ResultSet r = s.executeQuery("SELECT * FROM Produse ORDER BY Denumire");
r.last();
r.deleteRow();
```

11.7.3. Interogari care produc rezultate multiple

Unele interogari SQL pot produce mai multe rezultate. Spre exemplu, o interogare poate apela o procedura memorata in baza de date MySQL, care la randul sau poate produce multiple rezultate prin simpla executare a mai multor instructiuni SQL. In masura in care driverul o permite, JDBC ofera programatorului acces la fiecare dintre aceste rezultate prin intermediul metodelor din clasa *Statement*.

Obiectele de tip *Statement* au la randul lor un cursor intern, care se poate pozitiona pe rand pe rezultatul fiecarei interogari in parte. Lista de rezultate ale unei interogari poate contine elemente de natura diferita - unele pot fi *ResultSet*-uri

(corespunzatoare instructiunilor SELECT) si altele pot indica numarul de randuri afectate (pentru INSERT, DELETE, UPDATE).

Metodele clasei *Statement* folosite pentru a accesa si manipula rezultate multiple sunt:

- **boolean execute(String sql)** – utilizat pentru a executa o interogare de orice fel, care ar putea produce o succesiune de rezultate de diferite nati. Daca primul dintre rezultatele produse este un *ResultSet*, metoda returneaza *true*, in caz contrar *false*. Primul rezultat poate fi extras folosind metodele *getResultSet()* sau *getUpdateCount()*, in functie de natura sa (vezi mai jos descrierea metodelor)
- **boolean getMoreResults()** - pozitioneaza cursorul intern al obiectului *Statement* pe urmatorul rezultat, oricare ar fi natura acestuia, inchizand *ResultSet*-ul anterior daca exista unul. Returneaza *true* daca rezultatul curent este un *ResultSet* si *false* daca nu mai exista *ResultSet*-uri sau daca urmatorul rezultat este un numar de randuri afectate
- **ResultSet getResultSet()** - returneaza rezultatul curent sub forma unui *ResultSet*. Daca nu mai sunt rezultate sau daca rezultatul curent este un numar de randuri afectate, metoda returneaza *null*
- **int getUpdateCount()** - returneaza rezultatul curent sub forma unui numar de randuri afectate. In cazul in care rezultatul curent este un *ResultSet* sau daca nu mai exista alte rezultate, metoda returneaza *-1*

Nota: putem fi siguri ca nu mai exista rezultate numai atunci cand ambele conditii urmatoare sunt indeplinite:

- *getMoreResults()* returneaza *false*
- *getUpdateCount()* returneaza *-1*

Exemplu: presupunand ca a fost executata o interogare care contine o combinatie de instructiuni SELECT, UPDATE etc, tratarea rezultatelor ar putea fi facuta astfel:

```
boolean isRS = s.execute(interrogareComplexa);
int updCount = isRS ? -1 : s.getUpdateCount();
while ((isRS != false) || (updCount != -1)) {
    if (isRS) {
        System.out.println("Rezultatul curent este un result set");
        // ...plus prelucrare date...
    } else {
        System.out.println("Au fost afectate " + updCount + " randuri");
    }
    isRS = s.getMoreResults();
    updCount = s.getUpdateCount();
}
```

11.8. Extragerea de meta-informatie

Numim meta-informatie caracteristicile structurilor informationale care contin informatia insasi. Spre exemplu, meta-informatia unui result set ne poate indica numele coloanelor sale si tipurile lor de date, pe cand meta-informatia unei baze de date ne releva character set-ul sau lista sa de tabele.

Meta-informatia este necesara deoarece nu intotdeauna o aplicatie opereaza pe o structura cunoscuta a bazei de date si tabelelor. Sa luam exemplul unui program care permite crearea si administrarea de baze de date pe un server: utilizatorul este cel care decide numele tuturor elementelor de structura, iar aplicatia nu le poate cunoaste a priori, motiv pentru care este obligata sa “descopere” structura bazei de date si sa faca o afisare a informatiei “in orb”.

In Java programatorul poate obtine doua tipuri de meta-informatie:

- meta-informatia atasata unei surse de date (ex: un server de baze de date). Aceasta presupune determinarea listei de baze de date, de tabele cuprinse in acestea, lista de coloane ale fiecarei tabele impreuna cu tipul de date si clauze suplimentare atasate etc. Informatia de acest fel se determina sub forma unui obiect de tip **DatabaseMetaData** returnat de metoda *getMetaData()* a clasei *java.sql.Connection*
- meta-informatia atasata unui result set. Programatorul poate determina numarul de coloane, numele acestora, tipurile lor de date etc. Informatia de acest fel este memorata in obiecte de tip **ResultSetMetaData** returnate de metoda *getMetaData()* a clasei *ResultSet*

Prezentam cateva dintre metodele de interes ale celor doua clase:

- DatabaseMetaData:
 - **ResultSet getCatalogs()** - produce lista de baze de date prezente pe server sub forma unui result set cu o singura coloana numita TABLE_CAT
 - **ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String[] types)** - intoarce lista de tabele, posibil filtrata. Daca toti parametrii sunt *null* va fi produsa lista de tabele din baza de date curenta; specificand primul argument, tabele vor fi cele din baza de date respectiva. Pentru fiecare tabela sunt raportate informatii precum: (lista nu este completa)
 - TABLE_CAT - baza de date din care face parte tabela
 - TABLE_NAME - numele tabelei
 - TABLE_TYPE - tipul tabelei. Poate fi, printre altele, "TABLE" sau "VIEW"
- **ResultSet getColumns(String catalog, String schemaPattern, String tableNamePattern, String columnNamePattern)** - extrage lista de coloane a unei tabele intr-un mod asemănător cu metoda anterioara
- ResultSetMetaData
 - **int getColumnCount()** - returneaza numarul de coloane prezente in rezultat
 - **String getColumnName(int)** - returneaza numele coloanei in functie de pozitia acesteia in rezultat
 - **int getColumnType(int)** - indica tipul de date al coloanei; valoarea intreaga rezultata este una dintre constantele din clasa *java.sql.Types*: INTEGER, DOUBLE, DATE etc.

Iata un exemplu de metoda care afiseaza in consola sistemului continutul unui *ResultSet* precedat de numele coloanelor sale:

```
public static void showResultSet(ResultSet rs) {
    try {
        ResultSetMetaData md = rs.getMetaData();
        for (int i = 1; i <= md.getColumnCount(); i++) {
            System.out.print(md.getColumnName(i) + "\t");
        }
        System.out.println();
        while (rs.next()) {
            for (int i = 1; i <= md.getColumnCount(); i++) {
                System.out.print(rs.getString(i) + "\t");
            }
            System.out.println();
        }
    } catch (SQLException ex) {
        System.out.println(ex.printStackTrace());
    }
}
```

11.9. Tratarea warning-urilor si erorilor

Majoritatea interogarilor SQL pot produce warning-uri sau erori. Acestea se manifesta in JDBC sub forma unor exceptii de tip *SQLException* sau *SQLWarning* (cea de-a doua fiind o subclasa a primeia). Ambele sunt exceptii checked, asadar vom fi obligati sa le tratam sau sa le declarăm in semnatura metodei.

Lista de warning-uri asociate unei interogari SQL poate fi obtinuta imediat dupa executia interogarii folosind metoda *getWarnings()* prezenta in clasele *Connection*, *Statement* sau *ResultSet*. Metoda returneaza un obiect de tip *SQLWarning* corespunzator primului warning aparut. Warning-urile sunt inlantuite intre ele; clasa *SQLWarning* ofera metoda *getNextWarning()* care returneaza urmatorul warning din lista, sau *null* la epuizarea listei. In plus, obiectele de tip *SQLException* implementeaza interfata *Iterable*, ceea ce inseamna ca pot fi utilizate in cadrul unei structuri *for...each*.

Obiectele de tip *SQLException*, odata prinse, dispun de metode care ofera urmatoarele informatii specifice SQL:

- codul numeric de eroare, folosind metoda **getErrorCode()**. Acesta este un cod local al DBMS-ului si difera de la un soft de baze de date la altul
- codul SQLSTATE, folosind metoda **getSQLState()**. Acesta este un cod standardizat ANSI. Nu toate codurile numerice au coduri SQLSTATE corespunzatoare

Exemplu: conectarea la o baza de date si memorarea informatiilor din doua coloane intr-o colectie de tip *Map*:

```

try{
    Class.forName("com.mysql.jdbc.Driver");
    Connection con = DriverManager.getConnection("jdbc:mysql://localhost/bazadedate");
    Statement s =
    con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_READ_ONLY);
    ResultSet = s.executeQuery("SELECT * FROM judete");
    Map<String, Integer> m = new HashMap<String, Integer>();
    while(rs.next()){
        m.put(rs.getString("j_nume"),rs.getInt("j_nrlocuitori"));
    }
} catch(SQLException e){
    System.out.println("Eroare SQL");
    System.out.println("Cod de eroare: "+e.getErrorCode());
    System.out.println("Cod SQLSTATE: "+e.getSQLState());
}
jList1 = new JList(m.keySet().toArray());

```

11.10. Utilizarea obiectelor de tip RowSet

11.10.1. Principii. Tipuri de RowSet-uri

Un rowset este un obiect ce incapsuleaza un set de rezultate si care prezinta capabilitati suplimentare fata de un *ResultSet*, facandu-l astfel mai usor de utilizat si eficientizand interactiunea cu serverul de baze de date. Obiectele de tip rowset sunt instante de clase care implementeaza interfata **RowSet**, care la randul sau extinde *ResultSet*, asadar modul de manevrare a obiectelor rowset este deja in buna parte cunoscut cititorului.

Un *RowSet* poate prezenta multiple facilitati suplimentare comparativ cu un *ResultSet*:

- poate fi updatable si scrollable chiar si cand driver-ul utilizat pentru interactiunea cu DBMS-ul nu ofera aceste servicii
- este utilizabil pe post de componenta beans, conform standardului *Java Beans*. Aceasta inseamna ca poate fi folosit pentru design “la rece” din cadrul mediilor de dezvoltare avansate, asemanator cu componentele grafice, suportand proprietati si event-uri (listeners etc)
- unele tipuri de rowset permit filtrarea locala a datelor continue (echivalentul adaugarii “pe loc” a unei clauze WHERE)
- exista posibilitatea combinarii datelor din doua sau mai multe rowset-uri, formand join-uri locale, in aplicatie, fara a mai interoga suplimentar serverul de baze de date
- unele tipuri de rowset permit citirea si scrierea datelor continue din/in format XML

Exista doua tipuri de rowset-uri, fiecare cu avantajele si dezavantajele sale:

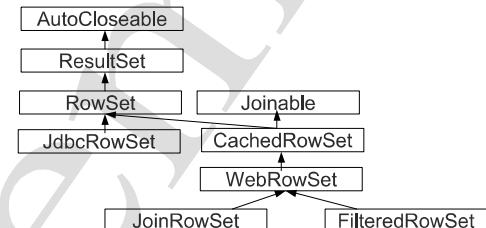
- **rowset-uri de tip connected** – reprezinta obiecte care, odata create, raman conectate la serverul de baze de date. Sunt de fapt obiecte care imbraca functionalitatea unui *ResultSet*, adaugandu-i acestuia posibilitatea de a fi scrollable si updatable
 - avantaje:
 - pot reflecta modificarile din baza de date pe masura ce ele apar
 - permit modificarea datelor din baza de date folosind API-ul obiectului de tip *RowSet*, nemaifiind necesare interogari separate
 - dezavantaje:
 - faptul ca trebuie sa mentina conexiunea cu baza de date le face mai costisitoare din punct de vedere al resurselor consumate
 - nu sunt serializable
- **rowset-uri de tip disconnected** – reprezinta obiecte rowset care se conecteaza la sursa lor de date numai pentru a citi sau modifica informatie; in rest ele sunt deconectate si functioneaza independent de sursa de date
 - avantaje:
 - sunt serializabile, ceea ce le face potrivite pentru transmiterea de informatie inter-aplicatii
 - sunt mai putin costisitoare din punct de vedere al resurselor consumate comparativ cu cele *connected*
 - datele pot fi manipulate si modificate local, in cadrul aplicatiei, si trimise serverului abia la incheierea intregului set de operatiuni, eficientizand astfel lucrul cu serverul

- datele lor pot proveni de pe un server de baze de date, dar si din alte surse (ex: fisiere)
- dezavantaje
 - cantitatea de informatie care poate fi stocata in memorie este limitata; nu se preteaza la informatii de anvergura
 - fiecare operatie de citire/modificare presupune realizarea unei noi conexiuni cu sursa de date; daca aceste operatii survin des, penalitatea introdusa de stabilirea/incheierea conexiunii devine importanta si merita folosite mai degraba variante *connected*

11.10.2. Ierarhia de interfete Rowset

Java SE pune la dispozitie programatorul urmatoarele interfete de tip *RowSet*, grupate in pachetul **javax.sql.rowset**:

- **JdbcRowSet** - un rowset de tip *connected*, care se comporta ca un *ResultSet* de tip *scrollable* si *updatable*
- **CachedRowSet** - un rowset care isi efectueaza o copie a datelor, apoi se deconecteaza de la sursa de date, operand asupra informatiei local. Obiectul rowset se va reconecta la sursa de date ori de cate ori trebuie sa citeasca sau sa modifice date
- **WebRowSet** - un *CachedRowSet* care are in plus capacitatea de a-si scrie continutul din/in format XML
- **FilteredRowSet** - un *CachedRowSet* care are capacitatea de a efectua filtrarea locala a inregistrarilor continue (in cazul interactiunii cu un DBMS, este ca si cum am aplica o clauza WHERE interogarii SELECT care a produs datele, insa local, in aplicatie, fara a mai contacta serverul)
- **JoinRowSet** - un *CachedRowSet* care are capacitatea de a combina date din doua sau mai multe rowset-uri, formand echivalentul unui JOIN SQL insa fara a mai contacta serverul de baze de date



11.10.3. Modalitati de creare a unui obiect RowSet

Programatorul are la dispozitie multiple modalitati de a crea obiecte de tip *RowSet*:

- crearea unei clase care implementeaza una dintre interfetele *RowSet*. Este, bineintele, solutia cea mai flexibila dar si cea mai laborioasa
- folosirea uneia dintre implementarile de referinta, care pot proveni din doua surse:
 - implementari oferite de Sun/Oracle - tehnic nu fac parte din API-ul Java SE, ci sunt clase suplimentare, proprietare, aflate in pachetul *com.sun.rowset*. Acesta este si motivul pentru care compilatorul emite un warning la incercarea de utilizare directa a acestor clase: *warning: com.sun.rowset.CachedRowSetImpl is Sun proprietary API and may be removed in a future release.*

Nota: pentru o explicare mai elaborata a motivului aparitiei warning-ului mai sus amintit consultati urmatorul link: <http://www.oracle.com/technetwork/java/faq-sun-packages-142232.html>.

Numele fiecarei clase este identic cu al interfetei implementate dar cu sufixul *Impl* (ex: *JdbcRowSetImpl*, *WebRowSetImpl* etc). Fiecare clasa ofera constructor fara argumente, unele dispunand si de diversi constructori suplimentari.

- implementari oferite de driverul JDBC - uneori pachetul ce contine driverul JDBC ofera si implementari de referinta ale catorva clase *RowSet*. Spre exemplu, pachetul *oracle.jdbc.rowset* contine implementari proprii pentru toate interfetele *RowSet*
- (Java >=7) folosirea clasei *RowSetProvider* si a interfetei *RowSetFactory*. Clasa *RowSetProvider* dispune de metoda statica supraincarcata *newFactory()* ce produce un obiect de tip *RowSetFactory*, care la randul sau ofera metode pentru crearea diverselor tipuri de rowset-uri: *createCachedRowSet()*, *createFilteredRowSet()*, *createJdbcRowSet()*, *createJoinRowSet()* si *createWebRowSet()*:

```
// creare rowset prin apelarea directa a constructorului unei implementari de referinta
JdbcRowSet r1 = new JdbcRowSetImpl(); // IDE-ul emite warning!
// creare rowset-uri folosind un obiect factory
WebRowSet r2 = RowSetProvider.newFactory().createWebRowSet(); // implementare de referinta Sun
FilteredRowSet r3 = RowSetProvider.newFactory().createFilteredRowSet(); // implementare Sun
```

Metoda *newFactory()* apelata fara argumente produce un obiect de tip *com.sun.rowset.RowSetFactoryImpl* si care creeaza implicit instante ale claselor din pachetul *com.sun.rowset*. Clasa *RowSetProvider* dispune de o a doua forma a metodei *newFactory()* care accepta ca argumente numele clasei de tip *RowSetFactory* si class loader-ul ce trebuie folosit in scopul incarcarii ei. In acest fel, obiectul *RowSetFactory* poate fi unul oferit ca parte a unei implementari de driver si care implicit produce instante ale implementarilor de referinta incluse impreuna cu driverul:

```
RowSetFactory rsf = RowSetProvider.newFactory("com.infoacad.RowSetFactoryImplementation", null);
WebRowSet wrs = rsf.createWebRowSet();
```

11.10.4. Configurarea si popularea cu informatie a obiectelor RowSet

Un rowset este un obiect care trebuie "dotat" cu parametrii necesari conectarii la sursa sa de date, astfel incat sa-si poata obtine/actualiza informatia folosindu-se de aceste setari initiale. In functie de natura sa, el fie ramane in legatura cu sursa de date (connected), fie se conecteaza la ea doar in caz de nevoie (disconnected). Principalii parametri necesari unui rowset - cei care ii asigura conectivitatea cu sursa sa de date si obtinerea informatiei - sunt urmatorii:

- **URL** – reprezinta URL-ul catre baza de date, in acelasi format prezentat mai sus in cazul deschiderii unei conexiuni folosind clasa *DriverManager* (ex: *jdbc:mysql://localhost/test*). Parametrul este manipulabil cu metodele *setUrl(String)* si *String getUrl()*
- **username** – reprezinta username-ul pentru conectarea la sursa de date. Este necesar in general la interactiunea cu servere SQL. Metode relevante din interfata RowSet: *void setUsername(String)* si *String getUsername()*
- **parola** – impreuna cu username-ul permite autentificarea la sursa de informatie (in general, un server SQL). Metode: *void setPassword(String)* si *String getPassword()*
- **command** – reprezinta comanda transmisa sursei de date prin care se populeaza rowset-ul. In cazul interactiunii cu un server SQL aceasta este deseori o instructiune SELECT (ex: *SELECT * FROM tabela*). Metode: *void setCommand(String)* si *String getCommand()*

Dintre parametrii suplimentari ai unui rowset amintim:

- **type** – reprezinta modul de manifestare al rowset-ului (forward only sau scrollable). Metode: *void setType(int)* si *int getType()*, unde valoarea intreaga va fi specificata sub forma constantelor din clasa *ResultSet* discutate anterior: TYPE_FORWARD_ONLY, TYPE_SCROLL_INSENSITIVE, TYPE_SCROLL_SENSITIVE
- **concurrency** – indica posibilitatea de modificare a datelor de la sursa prin intermediul rowset-ului. Metode: *void setConcurrency(int)* si *int getConcurrency()*, unde intregul este o constanta din clasa *ResultSet*: CONCUR_READ_ONLY sau CONCUR_UPDATABLE

Nota: pentru unele versiuni de MySQL este necesara setarea unei optiuni suplimentare in cazul lucrului cu rowset-uri de tip disconnected, folosind parametri additionali in URL: *jdbc:mysql://localhost/test?relaxAutoCommit=true*

Odata obiectul rowset creat si configurat, popularea sa cu informatie se realizeaza apelandu-i metoda *execute()*:

```
RowSet r = RowSetProvider.newFactory().createJdbcRowSet();
r.setUsername("dbuser");
r.setPassword("dbpass");
r.setUrl("jdbc:mysql://localhost/produse");
r.execute();
```

Ulterior apelului, un rowset de tip connected ramane conectat la sursa sa de date, pe cand unul disconnected intrerupe conexiunea si o reface numai atunci cand programatorul solicita explicit salvarea modificarilor efectuate intre timp.

Nota: implicit, un rowset este de tip TYPE_SCROLL_INSENSITIVE si CONCUR_UPDATABLE.

11.10.5. RowSet-uri de tip connected: obiecte JdbcRowSet

11.10.5.1. Crearea si popularea unui JdbcRowSet

In afara de modalitatile de creare comune tuturor obiectelor rowset, utilizarea directa a clasei *com.sun.rowset.JdbcRowSetImpl* ofera urmatoarele posibilitati suplimentare sub forma unor constructori particulari:

- *JdbcRowSetImpl()* - constructor care creeaza un rowset gol. Ulterior crearii este necesara setarea parametrilor obiectului si apoi apelarea metodei *execute()*:

```
JdbcRowSetImpl r = new JdbcRowSetImpl();
r.setUsername("dbuser");
r.setPassword("dbpass");
r.setUrl("jdbc:mysql://localhost/produse");
r.execute();
```

- *JdbcRowSetImpl(ResultSet)* – obiectul rowset va fi prepopulat cu datele din resultset-ul primit ca argument, nemaifiind nevoie de apelarea metodei *execute()*:

```
ResultSet rs = statement.executeQuery("SELECT * FROM products");
JdbcRowSetImpl r = new JdbcRowSetImpl(rs);
```

Nota: daca obiectul ResultSet pasat ca argument nu este actualizabil (updatable), rowset-ul nu este nici el actualizabil!

- *JdbcRowSetImpl(Connection)* – obiectul rowset utilizeaza conexiunea specificata pentru a-si citi datele; este necesara apoi specificarea comenzii ce produce datele si apelarea metodei *execute()*:

```
Connection c = DriverManager.getConnection("jdbc:mysql://localhost/store", "user", "pass");
JdbcRowSetImpl r = new JdbcRowSetImpl(c);
r.setCommand("SELECT * FROM products");
r.execute();
```

- *JdbcRowSetImpl(String URL, String username, String parola)* – crearea unui obiect rowset initializat cu parametrii necesari pentru conectarea la sursa de date; este necesara apoi stabilirea comenzii ce produce datele si apelarea metodei *execute()*:

```
JdbcRowSetImpl r = new JdbcRowSetImpl("jdbc:mysql://localhost/store", "user", "pass");
r.setCommand("SELECT * FROM products");
r.execute();
```

11.10.5.2. Utilizarea obiectelor de tip JdbcRowSet

Interfata *RowSet* deriva din *ResultSet*, ceea ce inseamna ca modul de lucru cu un rowset (si mai ales cu unul de tip connected) este identic cu manipularea unui *ResultSet*. De fapt, un *JdbcRowSet* este un wrapper pentru un *ResultSet* (un obiect construit in jurul lui).

```
// consideram o tabela SQL numita Produse care are doua coloane: Denumire si Pret
JdbcRowSet r = RowSetProvider.newFactory().createJdbcRowSet();
r.setUrl("jdbc:mysql://localhost/Produse"); r.setUsername("user1"); r.setPassword("pass1");
r.execute();
while(r.next()){
    System.out.println(r.getString(1)); // afisam continutul primei coloane (denumirea)
}
r.moveToInsertRow();
r.updateString("mere"); r.updateFloat(3.5);
r.insertRow(); // rowset-ul fiind connected, apelul efectueaza si adaugarea in tabela
// stergerea ultimului rand
r.last(); r.deleteRow();
```

11.10.6. RowSet-uri de tip disconnected

11.10.6.1. Obiecte de tip CachedRowSet

11.10.6.1.1. Particularitati

Un obiect de tip *CachedRowSet* contine un ansamblu de inregistrari copiate de la o sursa de date si are proprietatea de a fi *scrollable*, *updatable* si *Serializable*. Rowset-ul se conecteaza la sursa sa de date numai atunci cand are nevoie sa citeasca sau sa modifice date, in majoritatea timpului el fiind deconectat de aceasta. Intre doua conectari programatorul poate opera modificarile asupra continutului rowset-ului, care vor putea fi trimise inapoi catre sursa de date la urmatoarea conectare. Faptul ca rowset-ul este serializabil ofera o solutie convenabila de a schimba date intre aplicatii care comunica in retea.

Nota: un *CachedRowSet* poate fi scrollable si updatable chiar si atunci cand este construit pe baza unui *ResultSet* care nu are aceste caracteristici! Iata deci o modalitate simpla de a-i adauga ambele facilitati unui astfel de result set.

Un *CachedRowSet* are capabilitatea de a-si prelua datele fie de pe un server de baze de date, fie din alte surse care prezinta date in format tabular (ex: fisiere text TAB-delimited sau CSV, documente de tip spreadsheet etc). In scopul decuplarii rowset-ului de aspectele low-level ale citirii si modificarii datelor, un *CachedRowSet* se foloseste de un obiect de tip *SyncProvider*, care il doteaza cu alte doua obiecte ajutatoare:

- un *RowSetReader* - reprezinta obiectul care implementeaza logica de citire a datelor din sursa de date in rowset
- un *RowSetWriter* - obiectul care implementeaza logica de salvare a datelor inapoi pe sursa de date

Pentru a avea sub control memoria ocupata de catre un cached rowset si felul in care acesta isi gestioneaza traficul cu sursa sa de date, programatorul poate stabili doua caracteristici importante ale rowset-ului:

- numarul maxim de randuri. In orice moment al existentei sale rowset-ul nu va putea contine mai multe inregistrari, indiferent de cat de multe produce interogarea care il genereaza. Daca numarul maxim de randuri nu este setat sau este 0, nu va fi impusa nicio limita
- numarul maxim de inregistrari per pagina (“page size”). Rowset-ul nu citeste toata informatia de la sursa de date de la bun inceput, deoarece memoria disponibila lui este limitata (depinde de sistemul de operare, platforma hardware etc). Informatia este citita in calupuri (pagini), fiecare dintre ele avand un numar de inregistrari egal cu page size-ul. Pentru a aduce urmatorul calup de inregistrari este apelata metoda *nextPage()* a rowset-ului; calupul urmator il inlocuieste pe cel precedent, astfel incat inregistrarile anterioare nu se mai gasesc in rowset. Dupa cum era de asteptat, nu este permis ca page size sa fie mai mare decat numarul maxim de randuri ale rowset-ului. Numarul de pagini al unui rowset se obtine impartind numarul maxim de randuri la dimensiunea unei pagini.

Pentru clarificare, iata un exemplu: setand numarul maxim de randuri la 27 si page size la 10, la prima populare a rowset-ului se vor citi primele 10 inregistrari, la a doua (dupa apelarea lui *nextPage()*) urmatoarele 10 care le vor inlocui pe primele, iar la cea de-a treia ultimele 7 care le vor inlocui pe precedentele 10.

11.10.6.1.2. Crearea obiectelor de tip CachedRowSet

Un cached rowset gol poate fi creat in oricare dintre modalitatile prezentate in sectiunea 11.10.5.1; ulterior crearii el poate fi populat in doua moduri:

- prin citirea datelor de la sursa de date, stabilind mai intai parametrii necesari (URL, username, parola, comanda) si apeland apoi metoda *execute()*
- pe baza unui *ResultSet* deja existent, apeland una dintre metodele *populate()* din interfata *CachedRowSet* (aceasta este cea mai simpla modalitate de a adauga capacitatea de acces aleator si actualizare pentru un result set care nu le are). Metoda *populate()* are doua forme:
 - **populate(ResultSet)** - “importa” toate datele din *ResultSet* in obiectul *CachedRowSet*
 - **populate(ResultSet,int randDeIncep)** - incarca datele din *ResultSet* insa incepand cu randul specificat (inregistrarile precedente din result set nu sunt importate)

```
ResultSet rs = statement.executeQuery("SELECT Name FROM people");
CachedRowSet r = RowSetProvider.newFactory().createCachedRowSet();
r.populate(rs);
r.populate(rs,51); // sunt preluate inregistrarile de la 51 incolo, "sarind" primele 50
```

Atentie! Popularea rowset-ului cu date din result set nu seteaza parametrii necesari conectarii la sursa de date!
Pentru a putea salva schimbarile efectuate asupra datelor este necesara configurarea corecta a parametrilor uzuali! (url, command etc)

11.10.6.1.3. Utilizarea obiectelor de tip CachedRowSet

Pe langa metodele deja cunoscute care tin de interfata *ResultSet* si care permit parcurgerea oricarui rowset, un cached rowset prezinta urmatoarele particularitati:

- modificarile sunt operate implicit doar asupra datelor locale (cele copiate de la sursa de date) la apelarea metodelor *insertRow()*, *updateRow()* sau *deleteRow()*; pentru actualizarea lor la sursa este necesara apelarea metodei *acceptChanges()* a rowset-lui:

```
// introducerea unei inregistrari noi in tabela Produse care are coloanele Denumire si Pret
rowset.moveToInsertRow();
rowset.updateString("Denumire", "mere");
rowset.updateDouble("Pret", 3.5);
rowset.insertRow(); // are ca efect introducerea unei noi inregistrari doar in rowset
rowset.moveToCurrentRow();
rowset.acceptChanges(); // modificarile sunt salvate pe server
```

Fiecare apelare a metodei *acceptChanges()* realizeaza o conexiune cu sursa si ii transmite acesteia modificarile efectuate intre timp. De aceea este mai eficient sa se efectueze intregi succesiuni de operatii in modul disconnected, iar metoda *acceptChanges()* sa fie apelata doar la finalul fiecarui calup de modificari.

Nota: observam mutarea cursorului inapoi pe randul curent din result set inainte de a salva modificarile; omiterea acestei operatii genereaza o exceptie.

- in functie de felul in care a fost configurat rowset-ul (numar maxim de randuri, page size), este posibil - chiar foarte probabil - ca el sa nu contine toate datele produse de interogarea ce constituie comanda atasata acestuia. In aceste conditii, metodele *first()* si *last()* vor plasa cursorul pe prima/ultima inregistrare disponibila in rowset, nu pe prima/ultima furnizata de interogare. Iata un exemplu de rowset care are *page size* de 10 si ale carui inregistrari provin dintr-o tabela cu o singura coloana care memoreaza numerele de la 1 la 18:

```
CachedRowSet r = RowSetProvider.newFactory().createCachedRowSet();
r.setUrl("jdbc:mysql://localhost/test?relaxAutoCommit=true");
r.setUsername("root"); r.setCommand("SELECT * FROM t");
r.setMaxRows(15); // numar maxim de randuri: 15
r.setPageSize(10); // se citesc 10 inregistrari odata
r.execute();
r.last(); System.out.println(r.getString(1)); // 10 (ultima disponibila in acest moment)
r.nextPage(); // aduce inregistrarile 11-15, desi per total sunt 18!
r.first(); System.out.println(r.getString(1)); // 11 (nu mai sunt disponibile primele inregistrari)
r.last(); System.out.println(r.getString(1)); // 15 (din cauza numarului maxim de randuri)
```

Metoda *nextPage()* returneaza *true* cat timp pagina curenta (inainte ca *nextPage()* sa-si faca efectul!) nu este ultima, comutand pe *false* daca rowset-ul se afla deja pe ultima pagina in momentul apelarii lui *nextPage()*. De aceea, modalitatea de parcurgere a tuturor datelor furnizate de o interogare (in limita numarului maxim de randuri al rowset-ului) este urmatoarea:

```
// r este o referinta la un obiect de tip CachedRowSet
r.execute(); // rowset-ul este populat cu prima pagina
do{
    while(r.next()){
        // afisam prima coloana pentru fiecare inregistrare
        System.out.println(r.getString(1));
    }
}while(r.nextPage()); // odata incheiate inregistrarile unei pagini o aducem pe urmatoarea
```

11.10.6.2. Interfata WebRowSet

Un obiect de tip *WebRowSet* ofera in plus fata de un *CachedRowSet* posibilitatea de a-si citi sau salva datele in format XML folosind una dintre metodele urmatoare:

- **readXml(InputStream)** sau **readXml(Reader)** - populeaza obiectul *WebRowSet* pe baza continutului de tip XML citit dintr-un stream
- **writeXml(OutputStream)** sau **writeXml(Writer)** - salveaza continutul curent al rowset-ului in format XML in stream-ul specificat
- **writeXml(ResultSet, OutputStream)** sau **writeXml(ResultSet, Writer)** - populeaza rowset-ul cu datele din result set-ul primit ca argument si, in plus, salveaza informatia in format XML in stream-ul specificat

11.10.7. Filtrarea locala a datelor dintr-un rowset: interfata FilteredRowSet

Un obiect de tip *FilteredRowSet* este un *CachedRowSet* care dispune de capacitatea suplimentara de filtrare locala a inregistrarilor sale. Efectul este echivalent aplicarii unei clauze WHERE interogarii ce a produs datele (proprietaea *command* a rowset-ului). Odata aplicat filtrul, rowset-ul va expune utilizatorului sau numai randurile ce corespund criteriului cerut; cat timp filtrul este aplicat, celelalte randuri nu sunt vizibile sau actualizabile. Filtrul poate fi schimbat sau anulat oricand.

Sigurele metode suplimentare pe care *FilteredRowSet* le introduce comparativ cu *CachedRowSet* sunt cele legate de stabilirea sau obtinerea filtrului atasat:

- **void setFilter(Predicate p)** - stabileste filtrul dorit sub forma unui obiect care implementeaza interfata *Predicate* si care impune conditiile de filtrare. Valoarea *null* anuleaza filtrul curent, facand vizibile toate inregistrarile
- **Predicate getFilter()** - returneaza filtrul curent, sau *null* daca acesta nu a fost setat

Interfata *Predicate* trebuie implementata de catre obiectele ce constituie conditii de filtrare. Ea dispune de 3 metode care ii sunt apelate obiectului predcat de catre filtru in urmatoarele situatii:

- la introducerea unei noi inregistrari
 - **boolean evaluate(Object valoare, int coloana)** - filtrul ii apeleaza aceasta metoda predicatului cate o data pentru fiecare coloana a noii inregistrari; de fiecare data, primul argument contine valoarea noii coloane, iar cel de-al doilea indexul (pozitia) coloanei, numerotat de la 1. Cel de-al doilea argument este necesar pentru ca predicatul sa stie daca sa impuna conditii asupra coloanei sau nu (vezi exemplul mai jos)
 - **boolean evaluate(Object valoare, String numeColoana)** - analog, insa folosita in cazul in care introducerea noii inregistrari se efectueaza utilizand numele coloanelor, nu pozitia lor
- la parcurgerea rowset-ului
 - **boolean evaluate(RowSet)** - filtrul ii apeleaza aceasta metoda predicatului pentru fiecare operatie de mutare a cursorului, deoarece filtrarea este realizata de fapt prin modificarea implementarii interne a manipularii cursorului. Predicatul va raporta daca inregistrarea curenta este sau nu valida conform conditiilor de filtrare

Iata un exemplu: fie tabela *Produse* cu coloanele *Nume* si *Pret* in care incercam sa introducem o noua inregistrare cu valorile componente *Bomboane*, respectiv 20.

```
r.moveToInsertRow();
r.updateString(1, "Bomboane");
r.updateInt(2, 20);
r.insertRow();
```

Sa presupunem ca filtrul atasat rowset-ului doreste sa pastreze numai produsele mai ieftine de 100 RON; metoda `evaluate(valoare,coloana)` va fi apelata de doua ori: o data pentru coloana Denumire - `evaluate("Bomboane",1)` - si inca o data pentru coloana Pret - `evaluate(20,2)`. Filtrul nostru trebuie sa impuna restrictii numai pentru cea de-a doua coloana, si pentru aceasta va tine cont de valoarea celui de-al doilea argument pasat metodei `evaluate()`, actionand numai cand valoarea acestuia este 2:

```
public boolean evaluate(Object valoare, int pozitieColoana) {
    if(pozitieColoana !=2) return true;           // nu impunem restrictii asupra celorlalte coloane
    return ((Integer)valoare<=100);              // pentru coloana 2 sunt permise numai valori <=100
}
```

Nota: daca programatorul ar fi folosit apelurile `r.updateString("Denumire","Bomboane")` si `r.updateInt("Pret",20)` *filtrul ar fi apelat implicit cealalta forma a metodei evaluate() a predicatului: evaluate(Object valoare, String numeColoana)*.

```
class FiltruNumerePare implements Predicate {
    private int coloana;
    private String numeColoana;

    public FiltruNumerePare(int coloana, String numeColoana) {
        this.coloana = coloana;
        this.numeColoana = numeColoana;
    }

    public boolean evaluate(RowSet r) {
        try {
            return !r.isBeforeFirst() && !r.isAfterLast() && r.getInt(coloana)%2==0;
        } catch (SQLException ex) {
            ex.printStackTrace();
            return false;
        }
    }
    public boolean evaluate(Object valoare, int col) throws SQLException {
        return col==coloana && ((Integer)valoare%2==0);
    }
    public boolean evaluate(Object valoare, String numeCol) throws SQLException {
        return numeCol.equals(numeColoana) && ((Integer)valoare%2==0);
    }
}

// utilizarea clasei FiltruNumerePare pentru filtrarea datelor dintr-o tabela SQL care
// contine o coloana nr populata cu numerele de la 1 la 10
FilteredRowSet r = RowSetProvider.newFactory().createFilteredRowSet();
// ...configurare parametri conectare, apoi:
r.setFilter(new FiltruNumerePare(1,"nr"));      // nr este numele coloanei din tabela SQL
r.execute();                                      // populare rowset
r.next(); System.out.println(r.getRow());          // pozitia 2 (desi este prima valoare disponibila!)
r.next(); System.out.println(r.getRow()); // pozitia 4, desi este a doua valoare vizibila
```

Atentie! Filtrul ascunde anumite randuri insa nu modifica pozitiile lor din cadrul rowset-ului! (vezi ultimele doua linii din exemplu)

Filtrul modifica de fapt felul in care cursorul intern al result set-ului se deplaseaza de la o inregistrare la alta; sunt sarite cele care nu respecta conditiile impuse in predicat. Astfel, metoda `next()` poate sari mai multe inregistrari, sau poate chiar plasa cursorul dupa ultima inregistrare ("after last") in cazul in care niciuna dintre inregistrarile ramase nu corespunde conditiilor dorite. In acelasi fel, celelalte metode de pozitionare a cursorului vor proceda de asa natura incat programatorul nu va putea citi sau modifica inregistrari care nu corespund conditiilor predicatorului.

11.10.8. Join-uri locale: interfata JoinRowSet

Pentru a degreva serverul de baze de date de operatiile costisitoare pe care le implica un join intre doua sau mai multe tabele, JDBC ii ofera programatorului posibilitatea de a efectua operatii de acest fel intre rowset-uri deja copiate de pe server. Un obiect de tip `JoinRowSet` este un fel de colectie de rowset-uri (in general de tip disconnected) in care

programatorul indica coloanele ce constituie legatura intre tabelele implicate. Rowset-ul realizeaza join-ul si ii prezinta utilizatorului sau inregistrarile complete (combinante), ce pot fi parcurse folosind deja bine-cunoscutele metode din clasa *ResultSet*.

Dintre noile metode introduse de interfata *JoinRowSet* suplimentar fata de interfetele prezentate anterior evideniem:

- **addRowSet(RowSet, int pozitieColoana)** si **addRowSet(RowSet, String numeColoana)** - adauga in obiectul de tip *JoinRowSet* rowset-ul specificat ca prim argument, desemnand si coloana de legatura cu celelalte rowset-uri din obiect (de obicei, cheia primara sau una externa)
- **setJoinType(int) si int getJoinType()** - stabileste/indica tipul de join dorit, prin intermediul unor constante din interfata *JoinRowSet*: INNER_JOIN, FULL_JOIN, LEFT_OUTER_JOIN etc

Exemplu: fie doua tabele SQL - una cu produse si alta cu categorii. Fiecare produs face parte din cate o categorie, insa o categorie poate contine mai multe produse (relatie 1:N). Tabela *Categorii* are coloanele *idC* (cheia primara) si *NumeCategoria*; tabela *Produse* are coloanele *NumeProdus* si *idCat* (cheia externa). Continutul lor este cel din tabelul alaturat.

NumeProdus	idCat	idC	NumeCategoria
mere	1	1	fructe
pere	1	2	legume
rosii	2	3	mezeluri
parizer	3		

Iata un exemplu de join care combina inregistrarile din cele doua tabele, permitandu-ne sa afisam pentru fiecare produs categoria din care face parte:

```
Connection c = DriverManager.getConnection("jdbc:mysql://localhost/rowseturi","root","");
Statement s = c.createStatement();
// creare rowset-uri individuale
CachedRowSet categorii = RowSetProvider.newFactory().createCachedRowSet();
CachedRowSet produse = RowSetProvider.newFactory().createCachedRowSet();
categorias.populate(s.executeQuery("SELECT * FROM categorii"));
produse.populate(s.executeQuery("SELECT * FROM produse"));
// creare join rowset
JoinRowSet join = RowSetProvider.newFactory().createJoinRowSet();
join.addRowSet(categorias,"idC");
join.addRowSet(produse,"idCat");
// parcurgerea rezultatului operatiei de join; sunt disponibile toate coloanele din ambele tabele
while(join.next()){
    for (int i = 1; i <= join.getMetaData().getColumnCount(); i++) {
        System.out.print(join.getString(i)+"\t");
    }
    System.out.println();
}
/* rezultat:
3      mezeluri      parizer
2      legume         rosii
1      fructe         pere
1      fructe         mere
*/
```

11.11. BIBLIOGRAFIE

- JDBC Tutorial: <http://download.oracle.com/javase/tutorial/jdbc/index.html>
- Pachete JDBC:
 - java.sql: <http://docs.oracle.com/javase/7/docs/api/java/sql/package-summary.html>
 - pachetul javax.sql: <http://docs.oracle.com/javase/7/docs/api/javax/sql/package-summary.html>
 - pachetul javax.sql.rowset: <http://docs.oracle.com/javase/7/docs/api/javax/sql/rowset/package-summary.html>
 - API implementari de referinta Sun (pachetul com.sun.rowset):
http://docs.oracle.com/cd/E17824_01/dsc_docs/docs/jscreator/apis/rowset/com/sun/rowset/package-summary.html