Aplicatii Integrate pentru Intreprinderi
Semestrul de Toamna 2013

Laborator 3

Gestiunea informațiilor dintr-o bază de date MySQL prin JDBC

Continut

- JDBC – functionalitate, componente, arhitectura
- Ce este un “driver” de conectare la un sistem de gestiune pentru baze de date
- Configurare Connector/J
- Arhitectura JDBC
- Conectarea la sistemul de gestiune pentru baze de date
- Interogarea bazei de date conform specificatiei JDBC
- Utilizarea tranzactiilor in cazul accesului concurent la date
- Gestiunea informatiilor din dictionarul de date
- Tratarea exceptiilor de tip `SQLException`
- Alternative la manipularea informatiilor din sursele de date

JDBC: functionalitate

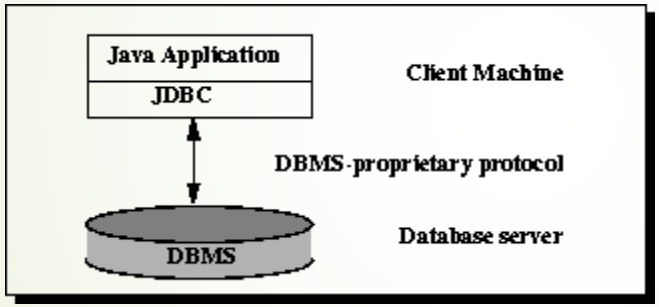
- JDBC = Java DataBase Connectivity
- interfata de programare Java pentru manipularea informatiilor din bazele de date
  - ❶ conectare / deconectare la sursa de date
  - ❷ transmiterea de interogari si recuperarea rezultatelor
    - DDL – interactiune cu dictionarul de date
    - DML – operatii `SELECT`, `INSERT`, `UPDATE`, `DELETE`

JDBC: componente

- ❶ JDBC API 4.1
  - pachetele `java.sql` / `javax.sql`
  - Java SE / Java EE
- ❷ modulul pentru gestiunea driverelor
  - clasa `DriverManager`
  - clasa `DataSource` – conexiune la sursa de date JNDI
- ❸ suita de teste JDBC
- ❹ puntea ODBC-JDBC
  - solutie temporara, pentru SGBD care nu implementeaza driver JDBC
  - continut de clasa `sun.jdbc.odbc.JdbcDriver`

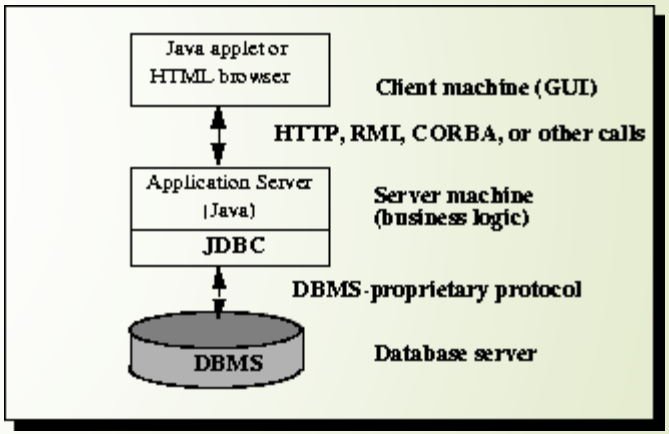
JDBC: arhitectura

modelul pe 2 niveluri



- model client-server
- aplicatia Java comunica direct cu sursa de date printr-un driver

modelul pe 3 niveluri



- comenzile transmise prin serviciile existente in nivelul intermediar (server de aplicatii)
- control centralizat al accesului la date

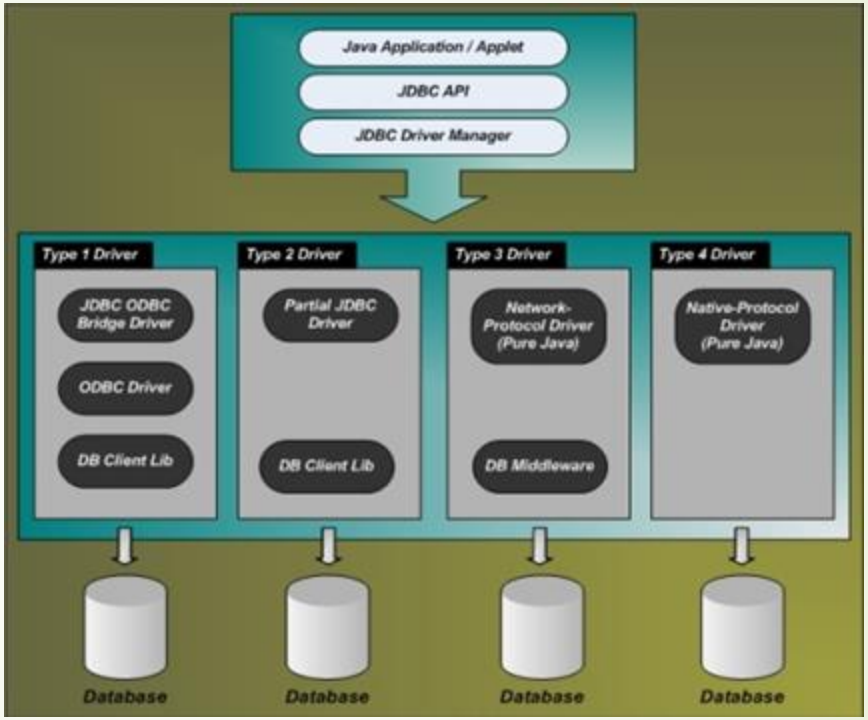
Ce este un “driver” de conectare la un sistem de gestiune pentru baze de date ?

- biblioteca prin care apelurile JDBC (in limbajul Java) sunt transformate intr-un format suportat de protocolul folosit de sistemul de gestiune al bazei de date, permitand accesul la informatii din medii eterogene

- interfata - nivelul de logica a aplicatiei si nivelul de date

- 4 tipuri
  - tipul 1 & 2 – biblioteci scrise in cod native
  - tipul 3 – middleware (protocol independent de SGBD)
  - tipul 4 – drivere scrise in Java folosind un protocol specific

Tipuri de drivere JDBC



Sursa: *JDBC Drivers*, http://www.ustudy.in/node/5475

Tipul 1

- foloseste puntea JDBC-ODBC: apelurile JDBC sunt transformate in apeluri ODBC
- solutie temporara – SGBD care nu implementeaza drivere JDBC

Avantaje

- compatibil cu orice SGBD care are instalat ODBC

Dezavantaje

- nu sunt portabile
- performanta scazuta la transformarea apelurilor
- driver-ul trebuie instalat pe masina client (incompatibilitate cu unele aplicatii Internet)

Tipul 2

- drivere scrise partial in Java, partial in cod nativ (biblioteca specifica pentru sursa de date la care aceasta se conecteaza)
- OCI – Oracle Call Interface

Avantaje

- performante mai bune decat tipul 1

Dezavantaje

- nu sunt portabile
- nu toate SGBD ofera biblioteca specifica
- driver-ul trebuie instalat pe masina client (incompatibilitate cu unele aplicatii Internet)

Tipul 3

- transformarea comenzilor JDBC intr-un protocol independent de baza de date
- foloseste middleware ce transforma acest protocol intr-un format specific sistemului de gestiune petru baze de date

Avantaje

- nu necesita instalarea de biblioteci specifice pe client
- portabilitate / adecvare pentru orice aplicatie Internet
- cel mai efficient tip de driver
- flexibilitate – compatibilitate cu orice tip de baza de date
- functionalitati: memorarea (conexiunilor, seturilor de date), echilibrarea incarcarii, autentificare, analiza performantelor

Dezavantaje

- transformari specifice sistemului de gestiune pentru baza de date realizate in cadrul nivelului intermediar

Tipul 4

- drivere scrise complet in Java
- implementeaza un protocol de retea specific sistemului de gestiune pentru baze de date

Avantaje

- independenta de platform
- adecvate pentru aplicatii Internet
- nu trebuie instalate alte resurse pe client sau server
- drivere incarcate dinamic
- nu exista niveluri intermediare pentru transformarea dintr-un protocol de retea intr-altul

Dezavantaje

- cate un driver separat pentru fiecare sistem de gestiune al bazei de date

Configurare Connector/J

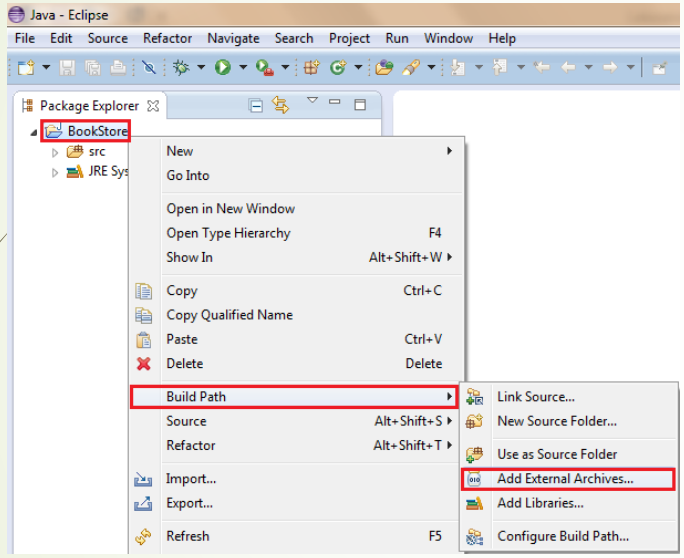
- driver JDBC de tip 4 pentru conectarea la un sistem de gestiune al bazei de date MySQL
- versiunea 5.1.26 disponibila la http://www.mysql.com/downloads/connector/j/
- compilare

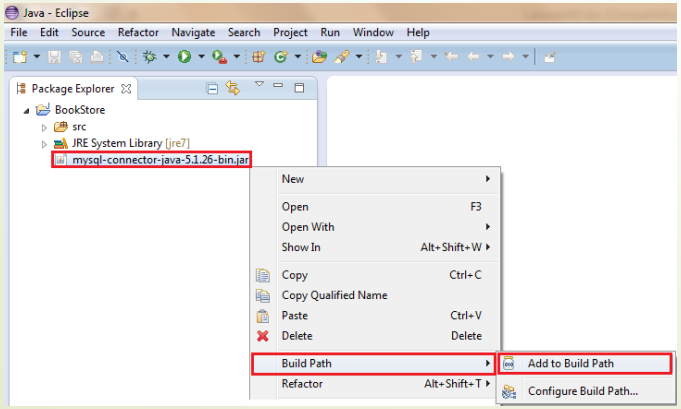
```
>javac -classpath .;mysql-connector-java-5.1.26-bin.jar <nume_fisier>.java
>javac -classpath .:mysql-connector-java-5.1.26-bin.jar <nume_fisier>.java
```

- 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>
```

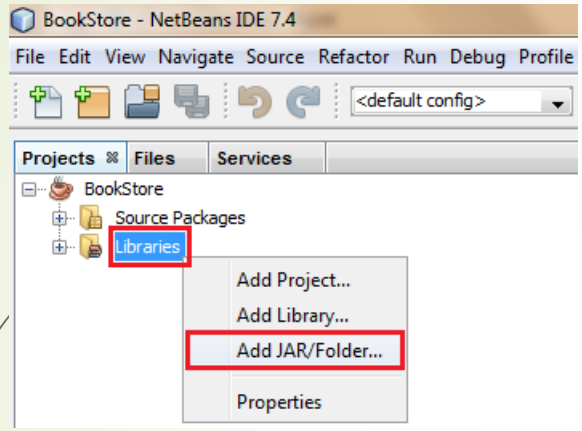
Configurare Connector/J Eclipse

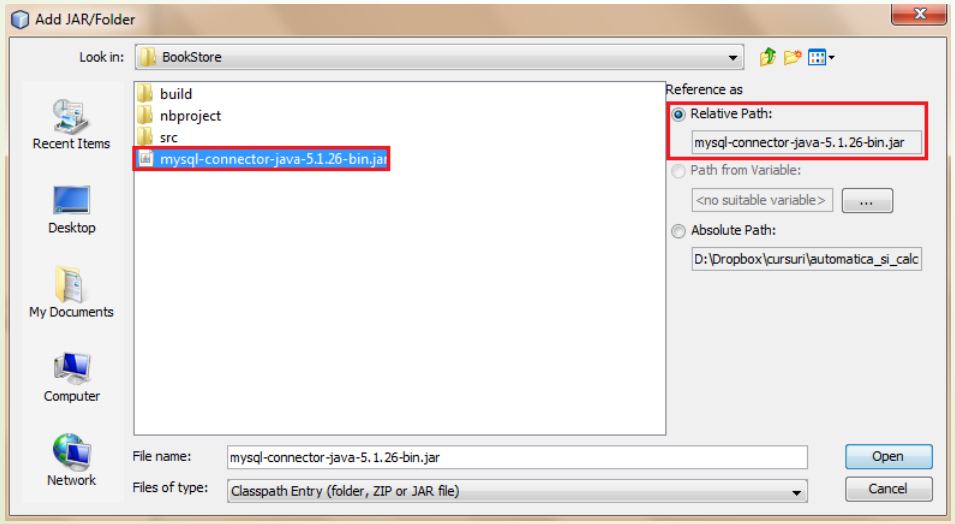




- nume proiect → Build Path → Add External Libraries
- nume biblioteca → Build Path → Add to Build Path
- daca operatia a fost realizata cu succes, numele bibliotecii apare la *Referenced Libraries*

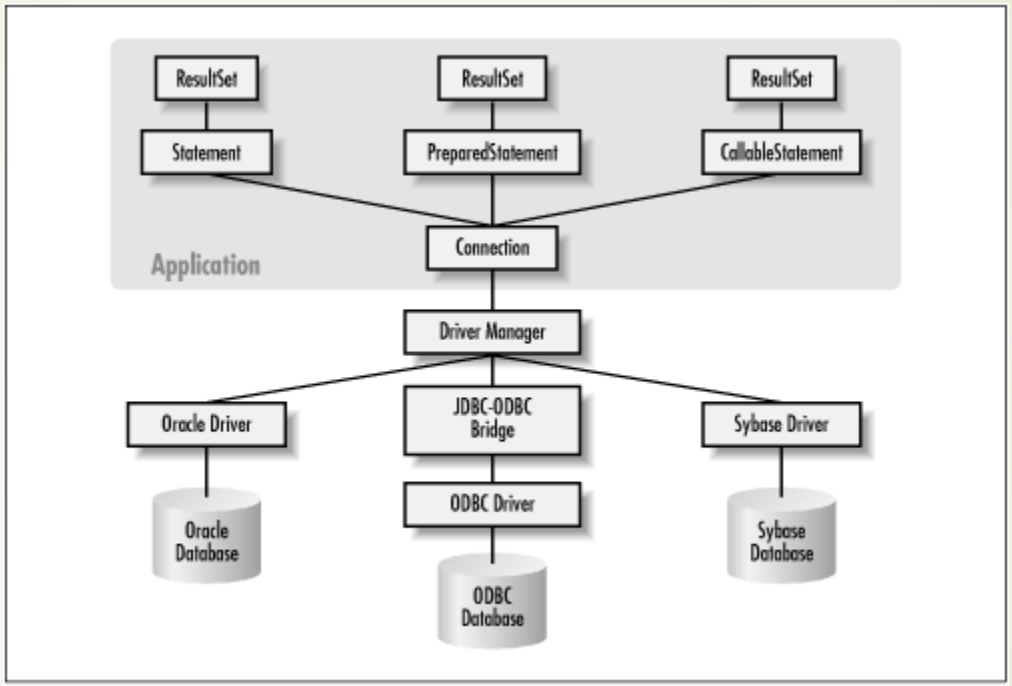
Configurare Connector/J Netbeans





- nume proiect → Libraries → Add JAR/Folder
- referinta specificata drept cale relativa (Reference As → Relative Path)

Arhitectura JDBC



Sursa: Jason HUNTER, William CRAWFORD – *Java Servlet Programming*
http://docstore.mik.ua/orelly/java-ent/servlet/ch09_02.htm

- JDBC API – comunicatia dintre aplicatia Java si modulul de gestiune al driverului
- JDBC Driver API – comunicatia dintre modulul de gestiune al driverului si baza de date

Etapele dezvoltarii unei aplicatii JDBC

- ❶ - pentru versiunile de drivere < JDBC 4.0: inregistrarea driverului

```
DriverManager.registerDriver (new com.mysql.jdbc.Driver());
Class.forName ("com.mysql.jdbc.Driver").newInstance();
```

- ❶ deschiderea conexiunii la baza de date
- ❷ realizarea de interogari catre baza de date
- ❸ procesarea rezultatelor obtinute cu propagarea modificarilor realizate inapoi in baza de date
- ❹ inchiderea conexiunii la baza de date

Conectarea la sistemul de gestiune al bazei de date

- 2 metode
  - `DriverManager` – accesul la o sursa de date specificata printr-un URL al carei driver (specificat in classpath) este incarcat in mod automat dupa ce este identificat de interfata `Driver`
  - `DataSource` – metoda mai transparenta de acces la date
- structura URL-ului de identificare a bazei de date
  - `protocol:subprotocol:[nume_baza_de_date][lista_de_proprietati]`
  - `jdbc:mysql://[host][,failoverhost ...][:port]/[database] [?propertyName1][=propertyValue1][&propertyName2][=propertyValue2]…`
  - `jdbc:mysql://localhost:3306/librarie?user=root&password=*****`
- `DriverManager.getConnection` – primeste URL precum si alte proprietati (username, password, obiect `Properties`)

Interogarea bazei de date conform specificatiei JDBC

- crearea unui obiect de tip `Statement`
  - `Statement stmt = dbConnection.createStatement();`
  - `try (Statement stmt = dbConnection.createStatement()) {`
    `// …`
    `}`
  - specificarea proprietatilor conexiunii
    - tip: `TYPE_FORWARD_ONLY`, `TYPE_SCROLL_INSENSITIVE`, `TYPE_SCROLL_SENSITIVE`
    - concurenta: `CONCUR_READ_ONLY`, `CONCUR_UPDATABLE`
    - detinerea cursorului: `HOLD_CURSORS_OVER_COMMIT`, `CLOSE_CURSORS_AT_COMMIT`
    - directia de parcurgere – `setFetchDirection` (`FETCH_FORWARD`, `FETCH_REVERSE`, `FETCH_UNKNOWN`)

Metode ale clasei `Statement`

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

- `execute` – pentru interogari care intorc **mai multe obiecte** `ResultSet`

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

- `executeQuery` – interogari care intorc **un singur obiect** `ResultSet`

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

- `executeUpdate` – instructiuni DML si DDL

```
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);
```

Metode ale clasei `ResultSet`

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

- metode de tip getter (`getString`, `getInt`, `getByte`, `getBoolean`, `getBlob`, `getDate`) primind ca parametri
  - numele (aliasul) coloanei
  - indexul coloanei

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

Operatia de adaugare in setul de date

```
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();
```

- ❶ mutarea cursorului la pozitia in care urmeaza sa fie adaugata inregistrarea → metoda `moveToInsertRow`
- ❷ operatia de adaugare propriu-zisa → metoda `insertRow`
- se recomanda repozitionarea cursorului dupa operatia de adaugare !!!

Operatiile de modificare si stergere in setul de date

```
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();
}
```

- ❶ actualizarea atributelor corespunzatoare unei inregistrari → metoda `update...`
- anularea modificarilor realizate asupra atributelor unei inregistrari se face prin metoda `cancelRowUpdates`
- ❷ actualizarea randului ce contine attribute modificate → metoda `updateRow`
- pentru stergerea unei inregistrari se apeleaza metoda `deleteRow` dupa pozitionarea pe randul respectiv

Interogari parametrizate Clasa `PreparedStatement`

- extinde clasa `Statement`
- informatiile necunoscute sunt marcate prin caracterul `?`
- in momentul crearii, interogarea este transmisa SGBD care o precompileaza

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

- inainte de rularea interogarii, trebuie precizate valorile atributelor necunoscute

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

- executia se face prin metoda `executeUpdate` care intoarce numarul de atribute modificate

Interogari pentru apelul rutinelor stocate
Clasa `CallableStatement`

- extinde clasa `PreparedStatement`
- pentru parametrii de intrare (`IN` / `INOUT`) se precizeaza valoarea
- pentru parametrii de iesire (`OUT` / `INOUT`) se precizeaza tipul rezultatului → metoda `registerOutParameter`
- executia rutinei stocate se face cu metoda `execute`

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

Gestiunea tranzactiilor

- set de instructiuni SQL executate atomic

```
void addBatch(String query) throws SQLException
void clearBatch() throws SQLException
```

- executia tranzactiei se face prin metoda `executeBatch` care intoarce un tablou al rezultatelor corespunzatoare fiecarei interogari in parte
- nu accepta instructiuni ce intorc seturi de date
- marcarea modificarilor in baza de date se face apeland metoda commit
  - mecanismul implicit de marcare a modificarilor individuale trebuie schimbat inainte de executia tranzactiei prin metoda `setAutoCommit`

Gestiunea tranzactiilor (cont'd) Exemplu

```
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);
```

Gestiunea tranzactiilor (cont'd) Anomalii in cazul tranzactiilor

- ❶ citire "murdara" – valori ale atributelor modificate dar nemarcate inca in baza de date
- ❷ citire nerepetabila – pentru citiri succesive in tranzactia A se obtin valori diferite datorita modificarilor din tranzactia B

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

- ❸ citire fantoma – rezultate diferite ale unei interogari ca urmare a satisfacerii criteriilor

Gestiunea tranzactiilor (cont'd) Niveluri de izolare

- specificate prin metoda `setTransactionIsolation` aplicabila unui obiect de tip `Connection`

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

- starea bazei de date poate fi retinuta prin metoda `setSavePoint` si restaurarea ei se face prin `rollback`
  - stergerea unei stare se face apeland metoda `releaseSavePoint`

Manipularea informatiilor din dictionarul de date

- informatii precum structura bazei de date si tabelelor, restrictii de integritate

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

- `getCatalogs` – denumirea bazelor de date ce pot fi accesate prin conexiunea respective
- `getFunctionColumns` – descrierea functiilor asociate bazei de date
- `getProcedureColumns` – descrierea procedurilor asociate bazei de date
- `getPrimaryKeys` – obtinerea cheilor primare
  - `getBestRowIdentifier` – cheia primara optima pentru un scop
- `getExportedKeys`, `getImportedKeys` – obtinerea cheilor straine

Obtinerea descrierii unei tabele

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

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)

Obtinerea descrierii atributelor unei tabele

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

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 (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

Continutul exceptiilor de tip `SQLException`

- ❶ descrierea erorii – poate fi obtinuta prin metoda `getMessage`
- ❷ cod reprezentand starea SQL potrivit standardizarii ISO/ANSI si OpenGroup (X/Open)
  - format din 5 caractere alfanumerice
  - intors de metoda `getSQLState`
- ❸ cauza – unul sau mai multe obiecte `Throwable` (prin metoda `getCause`) care se refera unul pe altul

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

- ❹ referinte catre alte exceptii inlantuite, intoarsa de metoda `getNextException`

Gestiunea avertismentelor Clasa `SQLWarning`

- avertismentele nu opresc executia aplicatiei
- raportate pentru obiecte de tip
  - `Connection`
  - `Statement` (`PreparedStatement` / `CallableStatement`)
  - `ResultSet`
  - pot fi obtinute prin metoda `getWarnings`
- metodele puse la dispozitie:
  `getMessage`, `getSQLState`, `getErrorCode`

Obiecte de tip `RowSet`

- alternativa la `ResultSet`
- comportament de componente JavaBeans
  - specificarea unor proprietati
  - mecanismul de notificare
    - schimbarea pozitiei cursorului
    - operatii de adaugare/modificare/stergere a unei inregistrari
    - actualizarea continutului
- clasificare
  - conectate: `JdbcRowSet`
  - neconectate: `CachedRowSet`, `FilteredRowSet`, `JoinRowSet`, `WebRowSet`
- folosite atunci cand driverul JDBC nu ofera functionalitate de parcurgere sau actualizare a obiectelor `ResultSet`

Crearea unui obiect din clasa `JdbcRowSet`

- ❶ folosind un obiect `ResultSet`

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

- ❷ folosind un obiect `Connection`

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

- ❸ folosind constructorul implicit

```
JdbcRowSet jdbcRS = new JdbcRowSetImpl();
jdbcRS.setUrl("jdbc:mysql://localhost:3306/librarie");
jdbcRS.setUsername(usr);
jdbcRS.setPassword(pwd);
// ...
```

- ❹ folosind o instanta a clasei RowSetFactory

```
RowSetFactory RSfactory = RowSetProvider.newFactory();
JdbcRowSet jdbcRS = RSfactory.createJdbcRowSet();
// ...
```

Proprietatile unui obiect `JdbcRowSet`

- `type: ResultSet.TYPE_SCROLL_INSENSITIVE`
- `concurrency: ResultSet.CONCUR_UPDATABLE`
- `escapeProcessing: true`
- `maxRows: 0`
- `maxFieldSize: 0`
- `queryTimeout: 0`
- `showDeleted: false`
- `transactionIsolation: Connection.TRANSACTION_READ_COMMITTED`
- `typeMap: null`

Seturi de date deconectate Clasa `CachedRowSet`

- datele sunt retinute intr-o zona de memorie, in loc sa fie accesate din sursa de date
- din ea sunt derivate `FilteredRowSet`, `JoinRowSet`, `WebRowSet`
- creare
  - folosind constructorul implicit `CachedRowSetImpl`
  - folosind o instanta a clasei `RowSetFactory`
- contine implementarea implicita `RIOptimisticProvider` a `SyncProvider`
  - obiecte `RowSetReader`, `RowSetWriter`
- pentru actualizare, trebuie specificata cheia primara

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

Seturi de date deconectate Clasa `CachedRowSet` (2)

- metoda `acceptChanges()` – procesarile sunt vizibile la nivelul sursei de date
- gestiunea conflictelor: clasa `RIOptimisticProvider` – model de concurenta optimist
  - daca nu exista conflicte, noile informatii sunt transferate
  - daca sunt detectate conflicte, actualizarile se ignora

```
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,...);
                }
            }
        }
    }
}
```

Seturi de date deconectate Clasa `CachedRowSet` (3)

- actualizarile pot fi notificate catre obiecte care implementeaza interfata `RowSetListener`
  - `cursorMoved` – schimbarea pozitiei cursorului
  - `rowChanged` – unul sau mai multe atribute dintr-o inregistrare sunt modificate / adaugari, stergeri
  - `rowSetChanged` – populatrea cu informatii
  - adaugarea, stergerea unui obiect ascultator se face prin metodele `addRowListener`, `removeRowListener`

Seturi de date deconectate Clasa `FilteredRowSet`

- limitarea numarului de inregistrari conform unui criteriu fara a preciza vreo conditie in interogare
- conditia indicata intr-o clasa ce implementeaza interfata `Predicate`
  - indexul sau numele coloanei dupa care se face filtrarea
  - limitele intre care se gasesc valorile
  - metoda evaluate
    - valoare atribut, denumire / index coloana
    - `RowSet`
- evaluarea are loc in momentul in care se apeleaza metoda `next`
- operatiile de adaugare, modificare, stergere sunt realizate numai daca nu contravin filtrelor asociate

Seturi de date deconectate Clasa `FilteredRowSet`(2)

```
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;
    }
```

Seturi de date deconectate Clasa `FilteredRowSet`(3)

```
    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;
    }
}
```

Seturi de date deconectate Clasa `JoinRowSet`

- operatii de asociere (`JOIN`) intre obiecte `RowSet` care nu sunt conectate la surse de date
- creare prin constructorul implicit `JoinRowSetImpl`
- metoda `addRowSet`
  - adaugarea de informatii la `JoinRowSet`
  - specifica setul de date (`RowSet`), denumirea / indexul coloanei care indica relatia dintre ele
  - obiectul `RowSet` (ce implementeaza Joinable) poate indica atributele care vor servi la realizarea asocierii prin `setMatchColumn`
- metoda `setJoinType` – specifica tipul de asociere
  - `INNER_JOIN` (implicit)
  - `CROSS_JOIN`, `FULL_JOIN`, `LEFT_OUTER_JOIN`, `RIGHT_OUTER_JOIN`

Seturi de date deconectate Clasa `JoinRowSet` (2)

```
CachedRowSet bills = new CachedRowSet();
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 CachedRowSet();
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);
```

Seturi de date deconectate Clasa `WebRowSet`

- scrierea / citirea in / din documente XML, folosit ca standard in comunicatia intre organizatii
- creare folosind constructorul implicit `WebRowSetImpl`
- implementare `RIXMLProvider` a clasei `SyncProvider` pentru gestiunea conflictelor
- scrierea in fisier

```
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);
```

- citirea din fisier

```
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);
```

Seturi de date deconectate Clasa `WebRowSet` (2)

- structura documentelor XML

```
<properties>
...
</properties>
<metadata>
    <column-count>...</column-count>
    <column-definition>
    ...
    </column-definition>
    ...
</metadata>
<data>
    <currentRow>
        <columnValue>...</columnValue>
        <updateValue>...</updateValue>
        ...
    </currentRow>
    <insertRow> <columnValue>...</columnValue> ... </insertRow>
    <deleteRow> <columnValue>...</columnValue> ... </deleteRow>
</data>
```

- operatiile writeXML si readXML sunt transparente pentru utilizator