TEHNOLOGII JAVA
PENTRU DEZVOLTAREA APLICAȚIILOR
LUCRARE DE LABORATOR 19

JDBC – Accesul la baze de date

I. SCOPUL LUCRĂRII

Lucrarea de față are rolul de a prezenta şi familiariza studentul cu modul de accesare si interogare a bazelor de date, lucrând în cadrul aplicațiilor Java.

La sfârşitul acestei lucrări, studentul va avea posibilitatea să scrie programe Java în care să utilizeze noțiunile învățate.

II. NOȚIUNI TEORETICE

1. Drivere JDBC

JDBC (Java DataBase Connectivity) reprezintă API-ul oferit de Sun Microsystems pentru dezvoltarea de aplicații Java care accesează bazele de date gestionate de diverse DBMS-uri (Sisteme de gestiune a bazelor de date). O funcție foarte importantă a JDBC este faptul că lucrează cu instrucțiuni SQL. Pentru a utiliza API-ul JDBC cu un DBMS particular este nevoie de un driver JDBC care să medieze între tehnologia JDBC şi baza de date. Driverul poate fi scris numai în Java sau într-o combinație de Java şi metode native JNI (Java Native Interface). Ultima specificație JDBC existentă are vers. 3.0 şi conține două pachete:

1. **java.sql**
2. **javax.sql** care oferă capabilități adiționale pe partea de server

Observație:

Aplicațiile care utilizează baze de date trebuie să includă pachetul **java.sql**, ci nu pachetul care conține implementarea driver-ului particular folosit. Totuşi calea spre pachetul conținând driverul trebuie să fie prezentă în CLASSPATH.

Există patru tipuri de drivere JDBC:

1. Puntea JDBC-ODBC. Este reprezentat de clasa **sun.jdbc.odbc.JdbcOdbcDriver**. Puntea traduce metodele JDBC în apeluri de funcții ODBC, şi necesită ca bibliotecile ODBC native şi driverele ODBC să fie instalate şi configurate pentru fiecare client ce utilizează un driver de acest tip. Funcționează doar pentru sitemele de operare Microsoft Windows şi Sun Solaris.
2. Java-API nativ. Folosesc interfața JNI pentru a face apeluri direct la API-ul unei baze de date locale. Sunt mai rapide decît tipul 1. Dar, de asemenea, necesită instalarea şi configurarea bibliotecilor client bază de date native pe maşina client.
3. Java-protocol de rețea. Folosesc un protocol de rețea (de obicei, TCP/IP) pentru a comunica cu aplicația JDBC middleware. Aplicația JDBC middleware traduce cererile JDBC folosind protocolul de rețea în apeluri de funcții specifice bazelor de date. Sunt cele mai flexibile, deoarece nu necesită biblioteci de baze de date native pe client şi se pot conecta la mai multe

baze de date.

4. Java-protocol bază de date. Implementează un protocol de bază de date pentru a comunica direct cu baza de date. Sunt cele mai rapide. De asemenea, nu necesită biblioteci de baze de date native pe client Sunt specifice unui singur DBMS.

Pentru gestiunea driverelor folosite într-o aplicație, JDBC API foloseşte un manager de drivere reprezentat de o instanță a clasei **DriverManager**.

2. Utilizarea punții JDBC -ODBC

În ODBC, bazele de date sunt reprezentate ca surse de date, identificate printr-un nume (DSN – Data Source Name) şi accesibile printr-un driver ODBC corespunzător.

Iată un exemplu, care arată cum se procedează pentru o bază de date creată în Access, sub sistemul de operare Windows 98. Presupunem că baza de date a fost deja creată si are numele **mesaje.mdb**. Pentru a putea face interogări asupra ei, trebuie mai întâi să o introducem ca sursă de date în ODBC. Pentru aceasta, se procedează astfel: din Control Panel alegem **ODBC Data Sources**, ceea ce va determina apariția unei ferestre **ODBC Data Sources Administrator**. În rubrica **User DSN**, folosind butonul **Add** şi wizardul care se declanşează, selectăm pe rând driverul ODBC (spre Access pentru exemplul nostru), respectiv baza de date **mesaje.mdb**, şi punem numele acestei surse de date (Data Source Name) **myMessages.mdb**. Mai departe, în aplicația Java, numele bazei de date utilizat va fi numele sursei de date, adică **myMessages.mdb**.

3. Accesarea unei baze de date folosind JDBC

Trebuie urmați 5 paşi:

1. înregistrarea driver-ului JDBC folosind gestionarul de drivere **DriverManager**
2. stabilirea unei conexiuni către baza de date
3. execuția unei instrucțiuni SQL
4. procesarea rezultatelor
5. închiderea conexiunii către baza de date

3.1 Înregistrarea driver-ului JDBC

Un driver JDBC este înregistrat automat de managerul de drivere atunci când clasa driver este încărcată dinamic prin meoda **Class.forName()**. Metoda este statică şi permite maşinii virtuale Java să aloce dinamic, să încarce şi să facă o legătură la clasa specificată ca argument al metodei. În cazul în care clasa nu este găsită, se aruncă o excepție **ClassNotFoundException**.

Iată un exemplu pentru înregistrarea driver-ului punte JDBC-ODBC:

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

3.2 Stabilirea conexiunii către baza de date

O conexiune este identificată printr-un URL specific. Sintaxa standard pentru URL-ul unei

baze de date este:
`jdbc<subprotocol>:<nume>`
Prima parte arată că se foloseşte JDBC pentru stabilirea conexiunii, **<subprotocol>** este un nume de driver valid, iar **<nume>** este un nume logic sau alias care corespunde bazei de date fizice. Dacă baza de date este accesată prin Internet, atunci secțiunea **<nume>** va fi de forma `//numeHost:port/numeDB`

Pentru stabilirea unei conexiuni la o bază de date, se foloseşte metoda statică **getConnection()** din clasa **DriverManager**:

```
Connection con=DriverManager.getConnection("jdbc:odbc:myMessages");
```

sau dacă baza de date necesită autentificare:

```
Connection con=DriverManager.getConnection("jdbc:odbc:myMessages",
                                           numeUtilizator,Parola);
```

3.3 Execuția unei instrucțiuni SQL

Pentru execuția unei instrucțiuni SQL neparametrizate, se foloseşte metoda **createStatement()** aplicată unui obiect Connection.

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

Se poate aplica apoi una din următoarele metode:

- **executeQuery()** – pentru interogările care returnează mulțimi rezultat (instanțe ale clasei ResultSet). Este cazul instrucțiunilor SELECT.
- **executeUpdate()** – pentru operațiile de actualizare INSERT, UPDATE, DELETE, cât şi pentru interogările SQL DDL de genul CREATE TABLE, DROP TABLE, ALTER TABLE. Metoda returnează un întreg care reprezintă fie numărul înregistrării afectate, fie este 0.
- **execute()** – se utilizează atunci când se obține mai mult de o mulțime rezultat.

```
ResultSet rs=instructiune.executeQuery("select * from myMessages");
String sql="insert into myMessages values ("Popescu","Ion","Craiova")";
int raspuns=instructiune.executeUpdate(sql);
```

Dacă se doreşte realizarea de apeluri SQL având date variabile drept intrare, se va folosi clasa **PreparedStatement** care moşteneşte clasa Statement.

```
PreparedStatement instructiune=con.prepareStatament("update myMessages
                               set nume=? Where prenume like ?");
instructiune.setString(1,"Popescu");
instructiune.setString(2,"Ion");
instructiune.executeUpdate();
```

Se poate utiliza **PreparedStatement** şi atunci când nu avem parametri.

```
PreparedStatement instructiune=con.prepareStatement("select *
                                                from myMessages");
ResultSet rs=instrucțiune.executeQuery();
```

3.4 Procesarea rezultatelor

Pentru parcurgerea simplă a înregistrărilor unui obiect din clasa **ResultSet** putem folosi metoda **next()**.

```
while(rs.next()) // implicit cursorul pozitionat inainte de prima linie
```

```
{
  System.out.println(rs.getString("nume")+", "+rs.getString("prenume")+
                                    ", "+rs.getString("oras"));
}
```

3.5 Închiderea conexiunii la baza de date

Se recomandă ca după procesarea datelor, să se închidă explicit conexiunea către baza de date. Întâi se vor închide obiectele **Statement** şi **ResultSet** folosind metodelor lor **close()**, apoi se va închide obiecul **Connection** prin apelarea metodei **close()** a acestuia:

```
try{
  //…
  rs.close();
  instructiune.close();
}
catch(SQLException e){
  System.out.println("Eroare la inchidere interogare: "+e.toString());
}
finally{
try{
    if(conexiuneBazadate!=null) {
       conexiuneBazadate.close();
    }
}
catch(SQLException e){
  System.out.println("Eroare la inchidere conexiune: "+e.toString());
}
}
```

4. Exemple

Vom crea o bază de date Access numită **StudentiDB**. Se va utiliza apoi puntea JDBC-ODBC pentru această bază de date, aşa cum s-a descris în secţiunea 2 a laboratorului. Sursa de date corespunzătoare acestei baze de date se va numi tot **StudentiDB**.

În continuare vom prezenta câteva programe Java foarte simple care: creează un tabel **studenti** pentru baza noastră de date, listează informaţiile conţinute în acest tabel, inserează o înregistrare în tabel. De asemenea, vom prezenta cum se pot obţine informaţii despre MetaDate (denumirile coloanelor unui tabel, tipurile coloanelor).

```
//CreateStud.java
import java.sql.*;

public class CreateStud {

 public static void main(String args[]) {

  String url = "jdbc:odbc:StudentiDB";
  Connection con;
  String sir;
```

```
  sir = "create table studenti " +
       "(Nume_student varchar(32), " +
       "Nr_matricol int, " +
       "An_stud int, " +
       "Grupa char(10), " +
       "Nota1 int, " +
       "Nota2 int, " +
       "Nota3 int, " +
       "Media double)";
  Statement stmt;

  try {
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

  } catch(java.lang.ClassNotFoundException e) {
   System.err.print("ClassNotFoundException: ");
   System.err.println(e.getMessage());
  }

  try {
   con = DriverManager.getConnection(url,"", "");
   stmt = con.createStatement();
      stmt.executeUpdate(sir);
   stmt.close();
   con.close();

  } catch(SQLException ex) {
   System.err.println("SQLException:" + ex.getMessage());
  }
 }
}

//Interogare.java
import java.sql.*;
public class Interogare {
 public static void main(String args[]) {
  String url = "jdbc:odbc:StudentiDB";
  Connection con;
  Statement stmt;
           String interogare="select Nume_student,Nr_matricol
                                 from Studenti";
  try {
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      }
  catch(java.lang.ClassNotFoundException e) {
   System.err.print("ClassNotFoundException: ");
   System.err.println(e.getMessage());
  }
  try {
     con = DriverManager.getConnection(url,"", "");
               stmt = con.createStatement();
               ResultSet rs = stmt.executeQuery(interogare);
               System.out.println("ListA Studentilor:");
      while (rs.next()) {
    String s = rs.getString("Nume_Student");
```

```
    int f = rs.getInt("Nr_matricol");
    System.out.println(s + "   " + f);
   }

   stmt.close();
   con.close();

  } catch(SQLException ex) {
   System.err.println("SQLException:" + ex.getMessage());
  }
 }
}

//InsertStud.java
import java.sql.*;
public class InsertStud {

 public static void main(String args[]) {

  String url = "jdbc:odbc:StudentiDB";
  Connection con;
  String sir;
  sir = "insert into studenti values
                              ('Gancea Alin',1456,1,123,5,7,6,6)" ;

  Statement stmt;

  try {
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

  } catch(java.lang.ClassNotFoundException e) {
   System.err.print("ClassNotFoundException: ");
   System.err.println(e.getMessage());
  }

  try {
   con = DriverManager.getConnection(url,"", "");
   stmt = con.createStatement();
      stmt.executeUpdate(sir);
   stmt.close();
   con.close();

  } catch(SQLException ex) {
   System.err.println("SQLException:" + ex.getMessage());
  }

 }
}

//ListCol.java
import java.sql.*;

class ListCol  {

public static void main(String args[]) {
```

```
  String url = "jdbc:odbc:StudentiDB";
  Connection con;
  Statement stmt;
                String query = "select * from Studenti";
  try {
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      }
  catch(java.lang.ClassNotFoundException e) {
   System.err.print("ClassNotFoundException: ");
   System.err.println(e.getMessage());
  }

  try {
   con = DriverManager.getConnection(url);
   stmt = con.createStatement();
   ResultSet rs = stmt.executeQuery(query);
   ResultSetMetaData rsmd = rs.getMetaData();

   System.out.println("");

   int numberOfColumns = rsmd.getColumnCount();

   for (int i = 1; i <= numberOfColumns; i++)
                          {
    if (i > 1) System.out.print(",  ");
    String columnName = rsmd.getColumnName(i);
    System.out.print(columnName);
   }
                         for (int i = 1; i <= numberOfColumns; i++)
                         {
    if (i > 1) System.out.print(",  ");
    String columnType = rsmd.getColumnTypeName(i);
    System.out.print(columnType);
   }

   System.out.println("");

   while (rs.next()) {
    for (int i = 1; i <= numberOfColumns; i++)
                                    {
     if (i > 1) System.out.print(",  ");
     String columnValue = rs.getString(i);
     System.out.print(columnValue);
    }
    System.out.println("");
   }

   stmt.close();
   con.close();
  } catch(SQLException ex) {
   System.err.print("SQLException: ");
   System.err.println(ex.getMessage());
  }
 }
}
```

III. MODUL DE LUCRU

1. Se editează codul sursă al programului Java folosind un editor de text disponibil(de ex., se poate utiliza Notepad).
2. Se salvează fişierul cu extensia **.java**. Fişierul trebuie salvat la următoarea locaţie: **c:\JBulider7\jdk1.3.1\bin**
3. Compilarea aplicaţiei Java se va face din linia de comandă. Se poate proceda astfel. Se deschide o fereastră MS-Dos: Start ->Run, se tipăreşte **command** în căsuţa de text şi se apasă butonul OK. Printr-o schimbare de directoare şi subdirectoare se trece la locaţia: **c:\JBulider7\jdk1.3.1\bin.** Sau, se lansează WindowsCommander. Se trece la locaţia **c:\JBulider7\jdk1.3.1\bin.** Se deschide o fereastră MS-Dos: Commander ->Run Dos.
4. Pentru compilare, se tipăreşte la prompter **javac nume_fişier_sursă.java** şi se apasă Enter. De ex., dacă fişierul se numeşte Test.java, se va scrie **javac Test.java**. În cazul în care programul conţine erori acestea vor fi semnalate şi afişate în fereastra MS-Dos, după care va apare iar prompter-ul. Dacă programul nu conţine erori şi compilarea se face cu succes, atunci va apare numai prompter-ul.
5. Pentru rularea unei aplicaţii Java stand-alone, se lansează interpretorul Java. Se tipăreşte la prompter următoarea comandă **java nume_clasă_care_conţine_main** şi se apasă Enter. De ex., dacă clasa din program care conţine metoda main( ) se numeşte Test, se va scrie **java Test**.
6. Dacă programul Java este un applet, se editează fişierul .**html**. Se salvează în acelaşi director cu fişierul **.class** rezultat în urma compilării cu succes a fişierului sursă java. Apoi pentru rulare se poate utiliza **appletviewer nume.html.**
Alternativ, după compilarea aplicaţiei Java, fişierul **.class** împreună cu fişierul **.html** pot fi mutate în orice alt director (nu trebuie neapărat să fie în **c:\JBulider7\jdk1.3.1\bin**). Se încarcă fişierul **.html** într-un browser Web (ex., Internet Explorer).

IV. TEMĂ

1. Se vor parcurge toate exemplele prezentate în platforma de laborator testându-se practic.
2. Scrieţi un program Java pentru a şterge informaţii din tabelul **studenti** şi pentru a actualiza anumite înregistrări din acest tabel.