PROGRAMARE ORIENTATĂ OBIECT 2

Conf.univ.dr. Ana Cristina DĂSCĂLESCU

Temtică curs 10

- Socket-uri
- API Java Database Connectivity
- Principiu de design DAO (Data Access Object)

Comunicarea prin socket

- Programarea cu socket-uri se referă la posibilitatea de a transmite date între două sau mai multe calculatoare interconectate prin intermediul unei rețele.

- Modelul utilizat pe scară largă în sistemele distribuite este sistemul **Client-Server**, care constă din:

✓o mulțime de procese de tip **server**, fiecare jucând rolul de gestionar de resurse pentru o colecție de resurse de un anumit tip (baze de date, fișiere, servicii Web, imprimantă etc.);

✓o mulțime de procese de tip **client**, fiecare executând activități care necesită acces la resurse hardware/software disponibile, prin partajare pe servere.

Comunicarea prin socket

- Serverele sunt cele care își încep primele activitatea.
- Un client își manifestă dorința de a se conecta și, dacă serverul este gata să accepte conexiunea, aceasta se realizează efectiv.
- În continuare, informațiile (datele) sunt transmise bidirecțional!!!
- Pentru conectarea la un server, clientul trebuie să cunoască adresa serverului și numărul portului dedicat.
- Porturile din intervalul **0...1023** sunt în general rezervate pentru servicii speciale, cum ar fi: 20/21 (FTP), 25 (email), 80 (HTTP), 443(HTTPS) etc.

Comunicarea prin socket

- Cea mai simplă modalitate de comunicare între două calculatoare dintr-o rețea o constituie **socket-urile**, care folosesc protocolul **TCP/IP.**

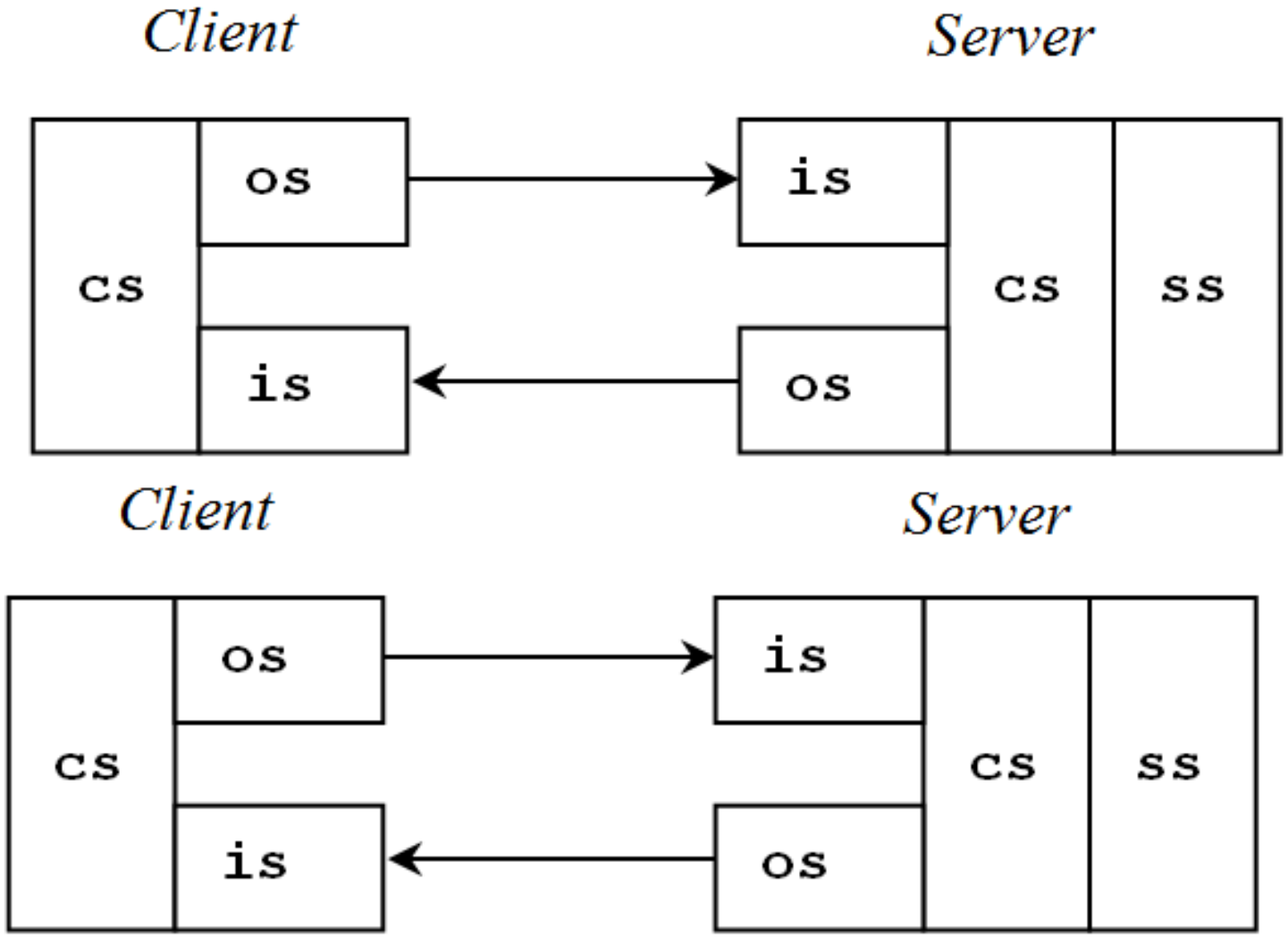
- În pachetul **`java.net`** sunt definite două clase care pot fi utilizate pentru comunicarea bazată pe socket-uri:

✓ **`ServerSocket`** – pentru partea de server;

✓ **`Socket`** – pentru partea de client.

- Oricărui socket îi **sunt atașate două fluxuri**: unul de intrare și unul de ieșire. Astfel, comunicarea folosind socket-uri se reduce la operații de scriere/citire în/din fluxurile atașate!!!

Comunicarea prin socket



Comunicarea prin socket

➢Partea de Server

- Se definește socket de tip server

```
ServerSocket(int port)
```

- Se apelează metoda

```
Socket accept()
```

✓ după ce un client s-a conectat, metoda va întoarce un socket de tip client (Socket), ale cărui fluxuri vor fi folosite pentru comunicarea bidirecțională.

- Fluxurile asociate unui socket se pot prelua folosind următoarele metode:

✓ `InputStream getInputStream(); //DataInputStream/Object`

✓ `OutputStream getOutputStream().`

- Închiderea unui socket se realizează folosind metoda `void close().`

➢Partea de Client

- Se va încerca realizarea unei conexiuni cu un server chiar în momentul creării unui socket de tip client!

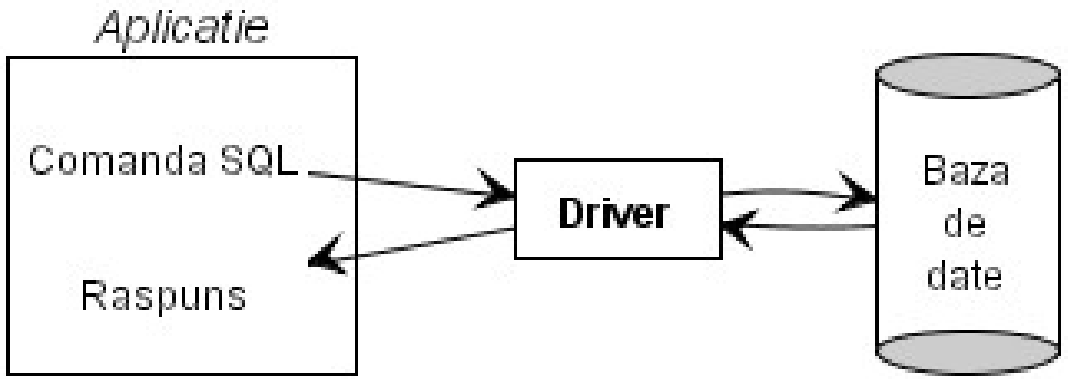
`Socket(String adresa_server, int port)` ->accept()

- În cazul în care conexiunea este realizată, se vor prelua fluxurile asociate socket-ului și se vor utiliza pentru comunicarea bidirecțională

JDBC

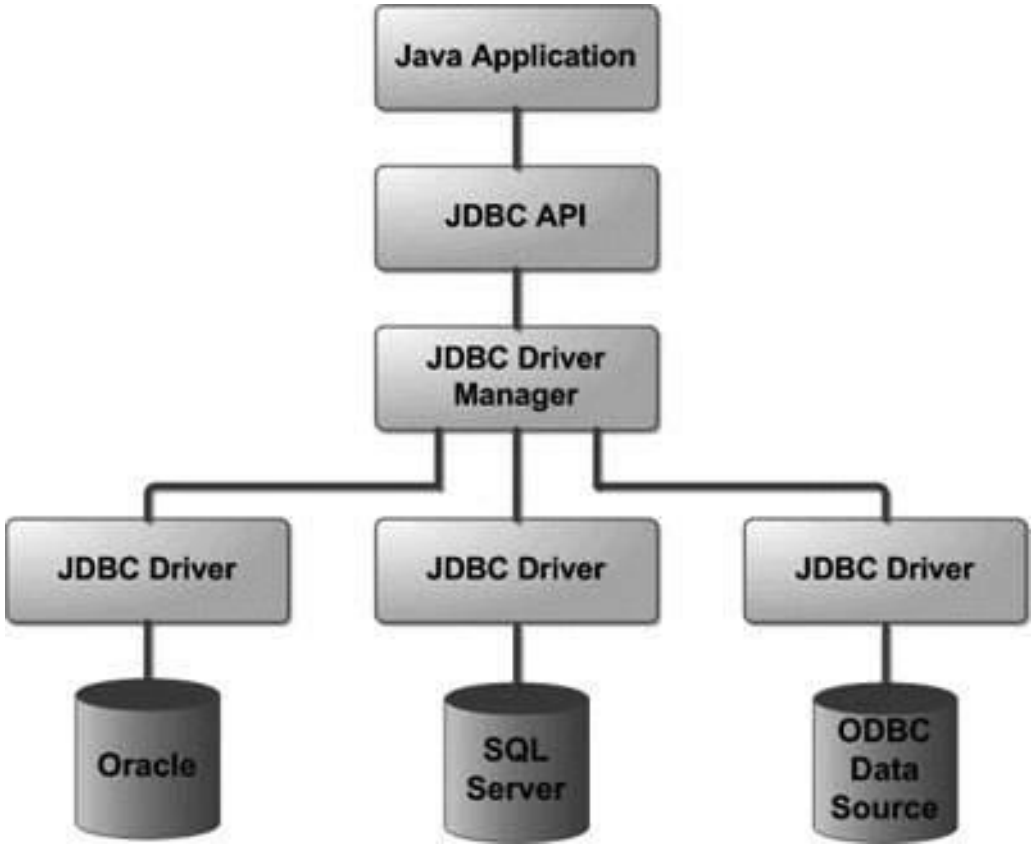
- ***Java DataBase Connectivity* (JDBC)** este un API dedicat accesării bazelor de date din cadrul unei aplicații Java, care permite conectarea la un server de baze de date, precum și executarea unor instrucțiuni SQL

  - Pentru fiecare SGBD există un **driver** dedicat (un program instalat local) care transformă cererile efectuate din cadrul programului Java în instrucțiuni care pot fi înțelese de către SGBD-ul respectiv.



JDBC

- **Procedura de instalare a unui drivere poate fi diferită de la un driver la altul**.
  - în cazul driverului MySQL, driverul este o arhivă de tip **jar**.
  - într-un mediu de dezvoltare, driverul poate fi specificat sub forma unei **biblioteci atașată** proiectului **sau poate fi deja disponibil** (de exemplu, versiunile noi de NetBeans conțin suport implicit pentru MySQL).



➢ Arhitectura JDBC

Nucleul JDBC conține o serie de **clase și interfețe aflate în pachetul** `java.sql`, precum:

- Clasa **`DriverManager`:** gestionează driver-ele JDBC instalate și alege driver-ul potrivit pentru realizarea unei conexiuni la o bază de date;
- Interfața **`Connection`:** gestionează o conexiune cu o bază de date (orice comandă SQL este executată în contextul unei conexiuni);
- Interfețele **`Statement`** / **`PreparedStatement`** / **`CallableStatement`:** sunt utilizate pentru a executa comenzi SQL în SGBD sau pentru a apela proceduri stocate;
- Interfața **`ResultSet`:** stochează sub forma tabelară datele obținute în urma executării unei comenzi SQL;
- Clasa **`SQLException`:** utilizată pentru tratarea erorilor specifice JDBC.

➢ Etapele realizării unei aplicații Java folosind JDBC

1. Stabilirea unei conexiuni cu o bază de date

- O ***conexiune*** **(sesiune)** la o bază de date reprezintă un context prin care sunt trimise secvențe SQL din cadrul aplicației către SGBD și sunt primite înapoi rezultatele obținute.

- Conexiunea este definită prin **JDBC URL** având formatul

```
jdbc:sub-protocol:identificator
```

✓ câmpul **`sub-protocol`** specifică tipul de driver care va fi utilizat (de exemplu sqlserver, mysql, postgresql etc.);

✓ câmpul **`identificator`** specifică **adresa** unei mașini gazdă (inclusiv un număr de port), **numele bazei de date** și, eventual, **numele utilizatorului și parola sa**.

➢ Exemple

- pentru o conexiune cu o bază de date denumită **BD**, care este stocată local folosind SGBD-ul MySQL

```
jdbc:mysql://localhost:3306/BD
```

- o conexiune cu o bază de date denumită **BD**, care este stocată local folosind SGBD-ul Oracle

```
jdbc:oracle:thin:@localhost:1521:BD
```

➢ **Deschiderea unei conexiuni** se realizează prin intermediul metodelor statice din clasa **DriverManager**

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

- **Exemplu:**

✓ Deschiderea unei conexiuni la baza de date `Firma`, găzduită local utilizând `MySQL`:

```
Connection
con=DriverManager.getConnection("jdbc:mysql://localhost:3306/Firma");

Connection
con=DriverManager.getConnection ("jdbc:mysql://localhost:3306/Firma",
                                 "popescuion","12345");
```

2. Crearea unui obiect de tip `Statement`

- Obiectele de tip **`Statement`** sunt utilizate pentru a executa instrucțiuni SQL (interogări, actualizări ale datelor sau modificări ale structurii) în cadrul unei conexiuni, precum și pentru prelucrarea datelor.

- JDBC pune la dispoziția programatorului 3 tipuri de statement-uri, sub forma a 3 interfețe:

✓`Statement` – pentru comenzi SQL simple, fără parametri;

✓`PreparedStatement` – pentru comenzi SQL parametrizate;

✓`CallableStatement` – pentru apelarea funcțiilor sau procedurilor stocate.

2. Crearea unui obiect de tip `Statement`

- Crearea unui obiect `Statement` se realizează apelând metoda

`Statement createStatement()` pentru un obiect de tip **`Connection`**

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

- Executarea unei secvențe SQL poate fi realizată prin intermediul următoarelor metode:

a) metoda `ResultSet executeQuery(String sql)` – este folosită pentru executarea interogărilor de tip **`SELECT`** și returnează un obiect de tip `ResultSet`

➢Exemplu

- Extragerea datelor din tabele tabela `Angajati` din baza de date:

```
String sql = "SELECT * FROM Angajati";
ResultSet rs = stmt.executeQuery(sql);
```

- Un obiect de tip `ResultSet` va conține rezultatul interogării sub o formă tabelară, precum și meta-datele interogării (de exemplu, denumirile coloanelor selectate, numărul lor etc.).

Interfața Statement

- **Parcurgerea înregistrărilor** se realizează cu ajutorul unui cursor, poziționat inițial înaintea primei linii. În clasa `ResultSet` sunt definite mai multe metode pentru a muta cursorul:

✓`boolean first()`,
✓`boolean last()`,
✓`boolean next()`,
✓ `boolean previous()`

- Pentru a extrage informațiile de pe fiecare linie se utilizează metode de forma

✓`TipData getTipData(int coloană)`
✓`TipData getTipData(String coloană)`

Interfața Statement

b) metoda **`int executeUpdate(String sql)`** – este folosită pentru executarea unor interogări SQL de tipul:

- **Data Manipulation Language** (DML), care permit actualizări ale datelor de tipul **`UPDATE/INSERT/DELETE`**,
- **Data Definition Language** (DDL) care permit manipularea structurii bazei de date, de exemplu**, `CREATE/ALTER/DROP TABLE`**).
- Metoda returnează **numărul de linii modificate** în urma efectuării unor interogări de tip DML sau 0 în cazul interogărilor de tip DDL.

```
String qrySQL = "INSERT INTO Angajati VALUES('1234567890999',
                                    › 'Albu Ioan',3210.10)";

int n = stmt.executeUpdate(qrySQL);
```

Interfața PreperedStatement

- Este utilizată pentru **comenzi SQL parametrizate**.
- Fiecare parametru este specificat prin intermediul unui semn de întrebare (?).
- Crearea unui obiect de tip PreparedStatement se realizează apelând metoda

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

pentru un obiect de tip `Connection`.

✓ **`sql`** este o comanda SQL cu unul sau mai mulți patametri

Interfața PreperedStatement

➢ **Exemplu**

```
String sql = "UPDATE persoane SET nume=? WHERE cod=?";
PreparedStatement pstmt = con.prepareStatement(sql);
```

- Setarea valorilor parametrilor se realizează prin metode de tip

```
void setTipData(int index, TipData valoare),
```

✓ unde **TipData** este tipul de date corespunzător parametrului respectiv

✓ prin argumentele metodei se specifică **indexul** parametrului (începând de la 1) și `valoarea` utilizată pentru atribuire.

Interfața CallableStatement

➢Este utilizată pentru executarea subprogramelor atașate unei baze de date, respectiv **funcții și proceduri stocate**

- procedurile sunt folosite pentru a efectua prelucrări în baza de date , în timp ce funcțiile sunt folosite pentru a efectua diferite calcule

- procedurile nu returnează nicio valoare prin numele lor (dar pot returna mai multe valori prin parametrii de intrare-ieșire, în timp ce funcțiile returnează o singură valoare.

- procedurile pot avea parametrii de intrare, de ieșire și de intrare-ieșire, în timp ce funcțiile pot avea doar parametrii de intrare;

Interfața CallableStatement

➢ **Apelarea unei funcții stocate** necesită efectuarea următorilor pași:

1. se creează un obiect de tip `CallableStatement` folosind un obiect de tip `Connection`:

```
CallableStatement sfunc = conn.prepareCall("{?=call
denumireFunctie(?)}");
```

- Primul ? reprezintă valoarea returnată de funcție (“parametrul de ieșire”),
- Pentru fiecare parametru de intrare se specifică câte un simbol **?**.

Interfața CallableStatement

2. Apelarea unei funcții stocate necesită efectuarea următorilor pași:

✓ se specifică tipul rezultatului returnat

```
sfunc.registerOutParameter(1, Types.DOUBLE);
```

- Valoarea 1 identifică primul ? din apelul metodei `prepareCall`

✓se setează valorile parametrilor de intrare, folosind metode de tipul `setTipData`

```
sfunc.setString(2, "B");
```

Interfața CallableStatement

✓se preia rezultatul returnat de funcția stocată, folosind metode de tipul `getTipData(int index_parametru_de_intrare):`

```
double total = sfunc.getDouble(1);
```

✓se execută funcția stocată:

```
sfunc.execute();
```

✓se preia rezultatul întors de funcția stocată, folosind metode de tipul `getTipData(int index_parametru_de_intrare):`

```
double total = sfunc.getDouble(1);
```

- Valoarea parametrului este 1 deoarece rezultatul întors de funcție se identifică prin numărul de ordine 1!

Interfața CallableStatement

➢**Apelarea unei proceduri stocate** necesită efectuarea următorilor pași:

1. se creează un obiect de tip `CallableStatement` folosind un obiect de tip `Connection`:

```
CallableStatement sfunc = conn.prepareCall("{call
inserareAngajat(?,?,?,?)}");
```

- Pentru fiecare parametru de intrare se specifică câte un simbol **?**.

- Pentru fiecare parametru de intrare se specifică câte un simbol **?**.

Interfața CallableStatement

- ✓ se specifică tipul parametrului de ieșire

```
sproc.registerOutParameter(4, Types.DOUBLE);
```

- ✓ se setează valorile parametrilor de intrare, folosind metode de tipul `setTipData`

```
sproc.setString(1, "1234567890999");
```

- ✓ se execută procedura stocată:

```
sfunc.execute();
```

- ✓ se preiau eventualele rezultatele întoarse de procedura stocată, folosind metode de tipul `getTipData(int index_parametru_de_ieșire)`:

```
double rezultat = sproc.getInt(4);
```

DATA ACCES OBJECT

- **DAO (Data Access Object)** este un design pattern utilizat în dezvoltarea de software pentru a separa logica de acces la date fața de logica de implementare.

- Acesta oferă un nivel abstract de acces la date, permițând astfel o mai mare flexibilitate și modularitate a codului.

1. Definirea modelului de date

- definirea clasei modelului de date ce va reprezenta înregistrările din baza de date (coloanele devin date membre private).

2. Definirea interfeței DAO

- Cuprinde operațiile CRUD (create, read, update, delete) pentru modelul de date
- Acestă interfața va fi implementată de către clasa DAO ce permite conectarea la baza de date, precum și efectuarea operațiilor CRUD

3. Implementarea Singleton pentru DAO

- Asigură faptul că există o singură instanța DAO

4. Gestionarea excepțiilor

- Tratarea excepțiilor care pot sa apară la nivel de conexiune, la nivel de model sau la nivel de bază de date, comenzi SQL etc