Programare avansata pe obiecte - laborator 11 (231)

Butan Silvia
silvia.butan@endava.com
butan.silvia@gmail.com

Object oriented programming - lab 11

Part I:
Java Sockets

- ➢ Socket programming refers to writing programs that execute across multiple computers in which the devices are all connected to each other using a network.
- ➢ There are two communication protocols that one can use for socket programming: User Datagram Protocol (UDP) and Transfer Control Protocol (TCP).
- ➢ ***UDP is connectionless*** = there is no session between the client and the server
- ➢ ***TCP is connection-oriented*** = an exclusive connection must first be established between client and server for communication to take place

Sockets programming over TCP/IP networks:

- **CLIENT - SERVER**

Java provides a collection of classes and interfaces that take care of low-level communication details between the client and the server. These are mostly contained in the **java.net package**, so we need to make the following import:

```
import java.net.*;
```

For our examples we'll run our client and server programs on the same computer. If we were to execute them on different networked computers, the only thing that would change is the IP address. In our case, we will use localhost on 127.0.0.1.

We will create a basic example involving a client and a server. It's going to be a two-way communication application where the client greets the server and the server responds.

The server application called GreetServer.java:

```
package com.paolabs.lab11;
```

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;

public class GreetServer {

   private ServerSocket serverSocket;
   private Socket clientSocket;
   private PrintWriter printWriter;
   private BufferedReader bufferedReader;

   public void startServer(int port) {
       try {
           serverSocket = new ServerSocket(port);
           clientSocket = serverSocket.accept();
           printWriter = new PrintWriter(clientSocket.getOutputStream(),
true);
           bufferedReader = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));

           String greeting = bufferedReader.readLine();
           if ("hello server".equals(greeting)) {
               printWriter.println("hello client");
           } else {
               printWriter.println("unrecognised greeting");
           }
       } catch (IOException ex) {
           ex.printStackTrace();
       }
   }

   public void stopServer() {
       try {
           printWriter.close();
           bufferedReader.close();
           clientSocket.close();
           serverSocket.close();
       } catch (IOException ex) {
           ex.printStackTrace();
```

```
        }
    }
}
```

```
package com.paolabs.lab11;

public class Main {

    public static void main(String[] args) {
        GreetServer server = new GreetServer();
        server.startServer(8081);
    }
}
```

The client called GreetClient.java:

```
package com.paolabs.lab11;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;

public class GreetClient {

    private Socket clientSocket;
    private PrintWriter printWriter;
    private BufferedReader bufferedReader;

    public void startConnection(String ip, int port) {
        try {
            clientSocket = new Socket(ip, port);
            printWriter = new PrintWriter(clientSocket.getOutputStream(),
true);
            bufferedReader = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
        } catch (IOException e) {
            e.printStackTrace();
        }
```

```
    }

    public String sendMessage(String msg) {
        printWriter.println(msg);
        String resp = null;
        try {
            resp = bufferedReader.readLine();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return resp;
    }

    public void stopConnection() {
        try {
            bufferedReader.close();
            printWriter.close();
            clientSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
package com.paolabs.lab11;

public class Main {

    public static void main(String[] args) {
        GreetClient client = new GreetClient();
        client.startConnection("127.0.0.1", 8081);
        String response = client.sendMessage("hello server");
        System.out.println(response);
    }
}
```

How Sockets Work:

A socket is one endpoint of a two-way communication link between two programs running on different computers on a network. A socket is bound to a port number so that the transport layer

can identify the application that data is destined to be sent to.

The Server runs on a specific computer on the network and has a socket that is bound to a specific port number. In our case, we use the same computer as the client and started the server on port 8081:

```
ServerSocket serverSocket = new ServerSocket(8081);
```

The server just waits, listening to the socket for a client to make a connection request.

```
Socket clientSocket = serverSocket.accept();
```

*When the server code encounters **the accept method,** it blocks until a client makes a connection request to it.*

If everything goes well, the server accepts the connection. The server gets a new socket, clientSocket, bound to the same local port, 8081, and also has its remote endpoint set to the address and port of the client.
At this point, the new Socket object puts the server in direct connection with the client, we can then access the output and input streams to write and receive messages to and from the client respectively.

From here onwards, the server is capable of exchanging messages with the client endlessly until the socket is closed with its streams.

In our example the server can only send a greeting response before it closes the connection, this means that if we send another message from the client code, the connection would be refused:

```
java.net.SocketException: Connection reset
```

To allow continuity in communication, we will have to read from the input stream inside a while loop and only exit when the client sends a termination request.

For every new client, the server needs a new socket returned by the accept call. The serverSocket is used to continue to listen for connection requests while tending to the needs of the connected clients.

The Client must know the hostname or IP of the machine on which the server is running and the port number on which the server is listening.

```
Socket clientSocket = new Socket("127.0.0.1", 8081);
```

The client also needs to identify itself to the server so it binds to a local port number, assigned by the system, that it will use during this connection. We don't deal with this ourselves.

The above constructor only creates a new socket when the server has accepted the connection, otherwise, we will get a connection refused exception. When successfully created we can then obtain input and output streams from it to communicate with the server.

The input stream of the client is connected to the output stream of the server, just like the input stream of the server is connected to the output stream of the client.

Continuous Communication:

Our example server blocks until a client connects to it and then blocks again to listen to a message from the client, after the single message, it closes the connection because we have not dealt with continuity.

Imagine we would like to implement a chat server, continuous back and forth communication between server and client would definitely be required.

We will have to create a while loop to continuously observe the input stream of the server for incoming messages.

Let's modify our server to echo back whatever messages it receives from clients:

```
package com.paolabs.lab11;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;

public class GreetServerContinuous {

   private ServerSocket serverSocket;
   private Socket clientSocket;
   private PrintWriter printWriter;
   private BufferedReader bufferedReader;
```

```
    public void startServer(int port) {
        try {
            serverSocket = new ServerSocket(port);
            clientSocket = serverSocket.accept();
            printWriter = new PrintWriter(clientSocket.getOutputStream(),
true);
            bufferedReader = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));

            String inputLine;
            while ((inputLine = bufferedReader.readLine()) != null) {
                if (".".equals(inputLine)) {
                    printWriter.println("good bye");
                    break;
                }
                printWriter.println("hello client");
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }

    public void stopServer() {
        try {
            printWriter.close();
            bufferedReader.close();
            clientSocket.close();
            serverSocket.close();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

Notice that we have added a termination condition where the while loop exits when we receive a period character.

We will start GreetServerContinuous using the main method just as we did for the GreetServer. GreetServerContinuous is similar to GreetClient, so we can duplicate the code. We are separating them for clarity.

We shall create multiple requests to the GreetServerContinuous to show that they will be served without the server closing the socket. This is true as long as we are sending requests from the same client.

```
GreetClient client = new GreetClient();
client.startConnection("127.0.0.1", 8081);
String response = client.sendMessage("hello server");
System.out.println(response);

String response1 = client.sendMessage("hello server");
System.out.println(response1);

String response2 = client.sendMessage(".");
System.out.println(response2);
```

Output:

```
hello client
hello client
good bye
```

This is an improvement over the initial example, where we would only communicate once before the server closed our connection, now we send a termination signal to tell the server when we're done with the session.

Server With Multiple Clients:

A server must have the capacity to service many clients and many requests simultaneously.

Another feature we will see here is that the same client could disconnect and reconnect again, without getting a connection refused exception or a connection reset on the server. Previously we were not able to do this.

In order to do this we need to create a new socket for every new client and service that client's requests on a different thread. The number of clients being served simultaneously will equal the number of threads running. The main thread will be running a while loop as it listens for new connections.

```
package com.paolabs.lab11.multiserver;

import java.io.BufferedReader;
import java.io.IOException;
```

```
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;

public class GreetClientHandler extends Thread {
   private Socket clientSocket;

   public GreetClientHandler(Socket socket) {
       this.clientSocket = socket;
   }

   public void run() {
       try {
           PrintWriter printWriter = new
PrintWriter(clientSocket.getOutputStream(), true);
           BufferedReader bufferedReader = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));

           String inputLine;
           while ((inputLine = bufferedReader.readLine()) != null) {
               if (".".equals(inputLine)) {
                   printWriter.println("good bye");
                   break;
               }
               printWriter.println("hello client");
           }

           bufferedReader.close();
           printWriter.close();
           clientSocket.close();
       } catch (IOException e) {
           e.printStackTrace();
       }
   }
}
```

```
package com.paolabs.lab11.multiserver;

import java.io.IOException;
import java.net.ServerSocket;
```

```
public class GreetMultiServer {

   private ServerSocket serverSocket;

   public void startServer(int port) {
       try {
           this.serverSocket = new ServerSocket(port);
           while (true) {
               GreetClientHandler clientHandler = new
GreetClientHandler(this.serverSocket.accept());
               clientHandler.start();
           }
       } catch (IOException e) {
           e.printStackTrace();
       }
   }

   public void stopServer() {
       try {
           serverSocket.close();
       } catch (IOException e) {
           e.printStackTrace();
       }
   }
}
```

Notice that we now call accept inside a while loop.
Every time the while loop is executed, it blocks on the accept call until a new client connects, then the handler thread, GreetClientHandler, is created for this client.

What happens inside the thread is what we previously did in the GreetServerContinuous where we handled only a single client. So the GreetMultiServer delegates this work to GreetClientHandler so that it can keep listening for more clients in the while loop.

We will still use GreetClient to test the server, this time we will create multiple clients each sending and receiving multiple messages from the server.

```
GreetClient client = new GreetClient();
client.startConnection("127.0.0.1", 8081);
String response = client.sendMessage("hello server");
```

```
System.out.println(response);

String response1 = client.sendMessage("hello server");
System.out.println(response1);

String response2 = client.sendMessage(".");
System.out.println(response2);
```

Output:

```
hello client
hello client
good bye
```

```
GreetClient client2 = new GreetClient();
client2.startConnection("127.0.0.1", 8081);
String response = client2.sendMessage("hello server");
System.out.println(response);

String response1 = client2.sendMessage("hello server");
System.out.println(response1);

String response2 = client2.sendMessage(".");
System.out.println(response2);
```

Output:

```
hello client
hello client
good bye
```

Part II:

JDBC (Java Database Connectivity)

JDBC (Java Database Connectivity) is an API for connecting and executing queries on a database. JDBC can work with any database as long as proper drivers are provided.

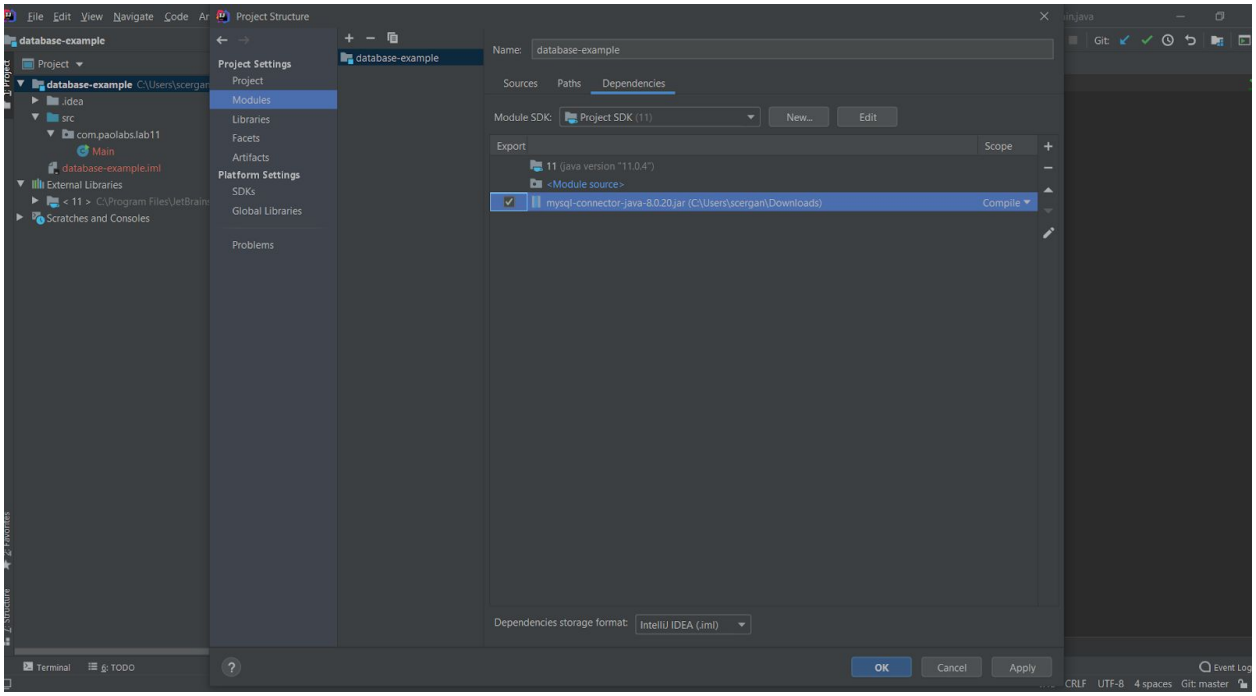
Connecting to a Database:

To connect to a database, we simply have to initialize the driver and open a database connection.

Registering the Driver: Class.forName("com.mysql.cj.jdbc.Driver");

We're using a MySQL database, we need the **mysql-connector-java** dependency:

https://search.maven.org/classic/remotecontent?filepath=mysql/mysql-connector-java/8.0.20/mysql-connector-java-8.0.20.jar



Creating the Connection:

We'll assume that we already have a MySQL server installed and running on localhost (default port 3306).

```
Connection connection = DriverManager.getConnection(DB_URL, USER,
PASSWORD);
```

We will have to create the database and a user, and grant the necessary access:

```
CREATE DATABASE lab11;
CREATE USER 'silvia' IDENTIFIED BY 'random-pass';
GRANT ALL on lab11.* TO 'silvia';
```

```
package com.paolabs.lab11.config;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DatabaseConfiguration {

   private static final String DB_URL =
"jdbc:mysql://localhost:3306/lab11";
   private static final String USER = "silvia";
   private static final String PASSWORD = "random-pass";

   private static Connection databaseConnection;

   private DatabaseConfiguration() {
   }

   public static Connection getDatabaseConnection() {
       try {
           if (databaseConnection == null || databaseConnection.isClosed())
{
               Class.forName("com.mysql.cj.jdbc.Driver");
               databaseConnection = DriverManager.getConnection(DB_URL,
USER, PASSWORD);
           }
       } catch (SQLException | ClassNotFoundException e) {
           e.printStackTrace();
       }
       return databaseConnection;
   }
```

```
   public static void closeDatabaseConnection() {
       try {
           if (databaseConnection != null &&
!databaseConnection.isClosed()) {
               databaseConnection.close();
           }
       } catch (SQLException e) {
           e.printStackTrace();
       }
   }
}
```

To send SQL instructions to the database, we can use instances of type **Statement**, **PreparedStatement** or **CallableStatement**. These are obtained using the **Connection** object.

After executing a query, the result is represented by a ResultSet object, with has a structure similar to a table, with lines and columns.

The ResultSet uses the next() method to move to the next line.

1. **Statement:**

The Statement interface contains the essential functions for executing SQL commands.

```
package com.paolabs.lab11.config;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class RepositoryHelper {

   private static RepositoryHelper repositoryHelper = new
RepositoryHelper();

   private RepositoryHelper() {
   }

   public static RepositoryHelper getRepositoryHelper() {
       return repositoryHelper;
```

```
    }

    public void executeSql(Connection connection, String sql) throws
SQLException {
        Statement stmt = connection.createStatement();
        // execute() for updating and select instructions
        stmt.execute(sql);
    }

    public void executeUpdateSql(Connection connection, String sql) throws
SQLException {
        Statement stmt = connection.createStatement();
        // executeUpdate() for updating the data or the database structure
        stmt.executeUpdate(sql);
    }

    public ResultSet executeQuerySql(Connection connection, String sql)
throws SQLException {
        Statement stmt = connection.createStatement();
        // executeQuery() for SELECT instructions
        return stmt.executeQuery(sql);
    }

}
```

Usage:

```
package com.paolabs.lab11.config;

import com.paolabs.lab11.util.RepositoryHelper;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;

public class SetUpData {

    public void setUp() {
        String createTableSql = "CREATE TABLE IF NOT EXISTS persons" +
                "(id int PRIMARY KEY AUTO_INCREMENT, name varchar(30)," +
```

```
                "age double)";

        Connection databaseConnection =
DatabaseConfiguration.getDatabaseConnection();
        RepositoryHelper repositoryHelper =
RepositoryHelper.getRepositoryHelper();

        try {
            repositoryHelper.executeSql(databaseConnection, createTableSql);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public void addPerson() {
        String insertPersonSql = "INSERT INTO persons(name, age)
VALUES('john', 20)";

        Connection databaseConnection =
DatabaseConfiguration.getDatabaseConnection();
        RepositoryHelper repositoryHelper =
RepositoryHelper.getRepositoryHelper();

        try {
            repositoryHelper.executeUpdateSql(databaseConnection,
insertPersonSql);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public void displayPerson() {
        String selectSql = "SELECT * FROM persons";

        Connection databaseConnection =
DatabaseConfiguration.getDatabaseConnection();
        RepositoryHelper repositoryHelper =
RepositoryHelper.getRepositoryHelper();

        try {
            ResultSet resultSet =
repositoryHelper.executeQuerySql(databaseConnection, selectSql);
```

```
            while (resultSet.next()) {
                System.out.println("Id:" + resultSet.getString(1));
                System.out.println("Name:" + resultSet.getString(2));
                System.out.println("Age:" + resultSet.getDouble(3));
            }

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

}
```

2. **PreparedStatement**

PreparedStatement objects contain precompiled SQL sequences. They can have one or more parameters denoted by a question mark.

```
package com.paolabs.lab11.repository;

import com.paolabs.lab11.config.DatabaseConfiguration;
import com.paolabs.lab11.model.Person;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class PersonRepository {

    public Person getPersonById(int id) {
        String selectSql = "SELECT * FROM persons WHERE id=?";

        Connection databaseConnection =
DatabaseConfiguration.getDatabaseConnection();
        try {
            PreparedStatement preparedStatement =
databaseConnection.prepareStatement(selectSql);
```

```
            preparedStatement.setInt(1, id);

            ResultSet resultSet = preparedStatement.executeQuery();
            return mapToPerson(resultSet);
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return null;
    }

    public void updatePersonName(String name, int id) {
        String updateNameSql = "UPDATE persons SET name=? WHERE id=?";

        Connection databaseConnection =
DatabaseConfiguration.getDatabaseConnection();
        try {
            PreparedStatement preparedStatement =
databaseConnection.prepareStatement(updateNameSql);
            preparedStatement.setString(1, name);
            preparedStatement.setInt(2, id);

            preparedStatement.executeUpdate();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    private Person mapToPerson(ResultSet resultSet) throws SQLException {
        if (resultSet.next()){
            return new Person(resultSet.getInt(1), resultSet.getString(2),
resultSet.getDouble(3));
        }
        return null;
    }

}
```

3. **CallableStatement**

The CallableStatement interface allows calling stored procedures.

To create a CallableStatement object, we can use the prepareCall() method of Connection:

```
String preparedSql = "{call insertPerson(?,?,?)}";
CallableStatement cstmt = con.prepareCall(preparedSql);
```

Setting input parameter values for the stored procedure is done like in the PreparedStatement interface, using setX() methods:

```
cstmt.setString(2, "silvia");
cstmt.setDouble(3, 26);
```

If the stored procedure has output parameters, we need to add them using the registerOutParameter() method:

```
cstmt.registerOutParameter(1, Types.INTEGER);
```

Then let's execute the statement and retrieve the returned value using a corresponding getX() method:

```
cstmt.execute();
int id = cstmt.getInt(1);
```

For this example to work, we need to create the stored procedure in our MySql database:

```
delimiter //
CREATE PROCEDURE insertPerson(OUT id int,
  IN name varchar(30), IN age double)
BEGIN
INSERT INTO persons(name,age) VALUES (name,age);
SET id = LAST_INSERT_ID();
END //
delimiter ;
```

Example:

```
// CallableStatement
public void insertPerson(Person person) {
   String preparedSql = "{call insertPerson(?,?,?)}";

   Connection databaseConnection =
```

```
DatabaseConfiguration.getDatabaseConnection();
   try {
       CallableStatement cstmt =
databaseConnection.prepareCall(preparedSql);
       cstmt.setString(2, person.getName());
       cstmt.setDouble(3, person.getAge());

       cstmt.registerOutParameter(1, Types.INTEGER);

       cstmt.execute();
       System.out.println("Added user with id:" + cstmt.getInt(1));
   } catch (SQLException e) {
       e.printStackTrace();
   }
}
```

Closing the Connection

When we're no longer using, it's necessary to close the connection to release database resources.

This can be done by using the close() API:

```
databaseConnection.close();
```