

**--Tema 3--**

**Tehnici de programare fundamentale**

**Warehouse Orders**

**Matei Andrei-George**

**Grupa 30225**

**An academic: 2018-2019**

1. **Objective**

Consider an application OrderManagement for processing customer orders for a warehouse. Relational databases are used to store the products, the clients and the orders.

a. Analyze the application domain, determine the structure and behavior of its classes and draw an extended UML class diagram.

b. Implement the application classes. Use javadoc for documenting classes.

c. Use reflection techniques to create a method createTablethat receives a list of objects and generates the header of the table by extracting through reflection the object properties and then populates the table with the values of the elements from the list:JTable createTable(List<Object> objects);

d. Implement a system of utility programs for reporting such as: under-stock, totals, filters, etc.

**2. Problem analysis, modeling, scenarios, use cases**

We need to create a database in SQL in order to have the best scenario possible for this application. Obviously we need a data base connection in java which uses a JDBC driver.

The database connection class contains the name of the driver (initialized through reflection), the database location (DBURL), and the user and the password for accessing the MySQL Server.

The class contains methods for creating a connection, getting an active connection and closing a connection, a Statement or a ResultSet.

In order to extract elements from the DB table, a special class (named entity) must be created. This class MUST have the fields exactly the same type as the columns from the corresponding table. The class must have also constructors, getters and setters.

In order for the Java application to interact with the DB, a special .jarlibrary must be added to the application.

It can be added either as an external jar file dependency or as a maven dependency, in case of a Maven project

The Java application uses this external library to communicate with the MySQL server

It sends queries to the server using Statements and it receives the results of the queries as ResultSet.

When you are finished using a Statement, call the method Statement.close to immediately release the resources it is using. When you call this method, its ResultSet objects are closed.

One of the use case of this application would be:

* The user clicks on a button from the interface, which can do a lot of operations with Clients and Products
* The user can add/edit/delete clients or products and see all the clients and products in a JTable
* The user can place an order with the desired quantity by choosing the client Id and the product Id necessary.

A particular scenario would be if there are no entries in the database which would then make it impossible to place an order, therefore something needs to be added in the data base.

**3. Design**

The app was designed by using the MCV pattern for the G U I and the model classes: Client, Product, Order. There are also specific classes which implement operations such as adding in the client table of the database, or deleting an entry, or searching for a client’s name.

Also there are several other views beside the main one, each one opens another window by pressing a button and does what the button says.

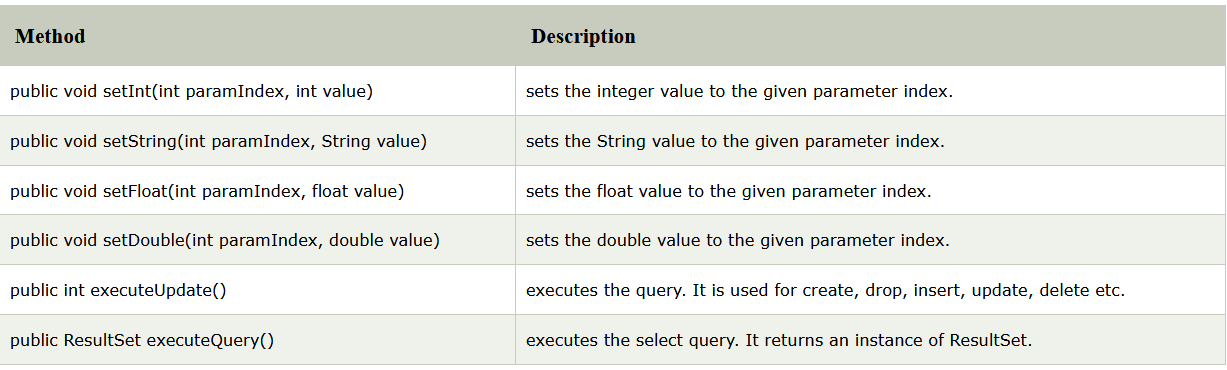
Creating and executing a query:

* Define a string with the query
* Create a connection to the DB
* Initialize the query
* Add the parameters to the query
* Execute the query

The PreparedStatement interface is a subinterface of Statement. It is used to execute parameterized query.

The prepareStatement() method of Connection interface is used to return the object of PreparedStatement. Syntax:

**public PreparedStatement prepareStatement(String query)throws SQLException{}**

The important methods of this class are:

|  |  |
| --- | --- |
| **Method** | **Description** |
| public void setInt(int paramIndex, int value) | sets the integer value to the given parameter index. |
| public void setString(int paramIndex, String value) | sets the String value to the given parameter index. |
| public void setFloat(int paramIndex, float value) | sets the float value to the given parameter index. |
| public void setDouble(int paramIndex, double value) | sets the double value to the given parameter index. |
| public int executeUpdate() | executes the query. It is used for create, drop, insert, update, delete etc. |
| public ResultSet executeQuery() | executes the select query. It returns an instance of ResultSet. |

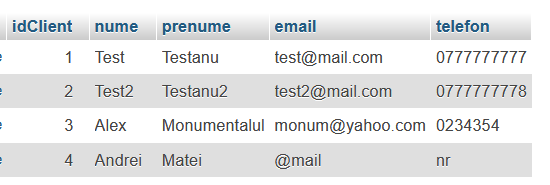
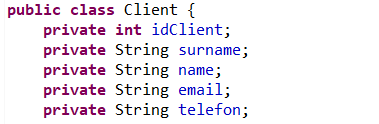
The SQL statements that read data from a database query, return the data in a result set. The SELECT statement is the standard way to select rows from a database and view them in a result set. The *java.sql.ResultSet* interface represents the result set of a database query.

A ResultSet object maintains a cursor that points to the current row in the result set. The term "result set" refers to the row and column data contained in a ResultSet object.

The methods of the ResultSet interface can be broken down into three categories −

* **Navigational methods:** Used to move the cursor around.
* **Get methods:** Used to view the data in the columns of the current row being pointed by the cursor.
* **Update methods:** Used to update the data in the columns of the current row. The updates can then be updated in the underlying database as well.[[1]](#footnote-1)
* **4. Implementation**

The **Client** class has the 5 columns of the table in the database (Clienti) as attributes.



It also has setters and getters for these attributes, in order to be used in the operations( like insert client, delete, and so on ).

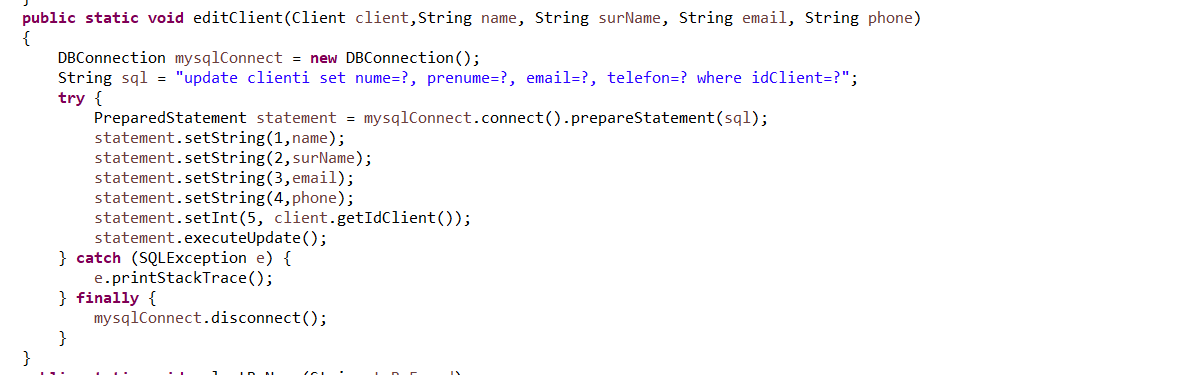
It also has a toString method which was used for testing early in the development.

The **Product** class, just like the Client one has the same attributes as the columns of the table in the database(produse) and also getters and setters.

The **Order** class encapsulates both Products and Clients, for which foreign keys were used. The idClient and idProdus comes from the clienti table, respectively the produse table from the database. This makes it a relational database by having connections to all three tables.

The operations classes are as follow:

**ClientOperations** class has as attributes the number of clients. Each of the method represents a SQL query, such as: selectClients, insertClient, deleteClient, editClient.

e.g.:

**public** **static** **void** editClient(Client client,String name, String surName, String email, String phone)

{

DBConnection mysqlConnect = **new** DBConnection();

String sql = "update clienti set nume=?, prenume=?, email=?, telefon=? where idClient=?";

**try** {

PreparedStatement statement = mysqlConnect.connect().prepareStatement(sql);

statement.setString(1,name);

statement.setString(2,surName);

statement.setString(3,email);

statement.setString(4,phone);

statement.setInt(5, client.getIdClient());

statement.executeUpdate();

} **catch** (SQLException e) {

e.printStackTrace();

} **finally** {

mysqlConnect.disconnect();

}

}

This class works as follow:

First we connect to the database, and then we have a String which is basically the query, in this case:

"update clienti set nume=?, prenume=?, email=?, telefon=? where idClient=?"

The “?” are being taken from the prepared statament with the methods: .setString, .setInt, .setFloat and so on.

For instance:

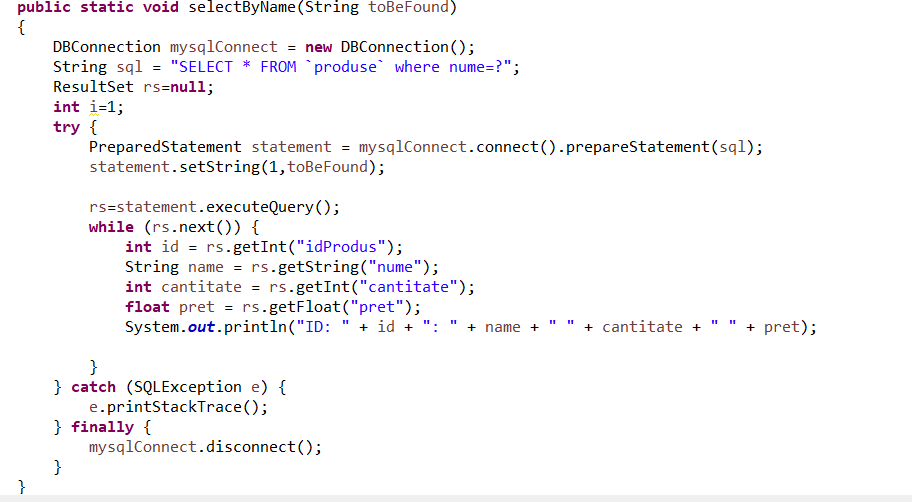
statement.setString(1,name);

corresponds to set nume=**?** because it the first question mark in the query.

This class also has methods for printing the client attributes (separately) and a class to count the number of clients in the database.

**ProductOperations** class has an attribute: noProducts. The methods are similar to the ClientOperations class. The queries are almost the same, but with respect to the table column names and such.

The methods are: selectProduct, editProduct, deleteProduct, selectByName.

For instance, the selectByName class:

**public** **static** **void** selectByName(String toBeFound)

{

DBConnection mysqlConnect = **new** DBConnection();

String sql = "SELECT \* FROM `produse` where nume=?";

ResultSet rs=**null**;

**int** i=1;

**try** {

PreparedStatement statement = mysqlConnect.connect().prepareStatement(sql);

statement.setString(1,toBeFound);

rs=statement.executeQuery();

**while** (rs.next()) {

**int** id = rs.getInt("idProdus");

String name = rs.getString("nume");

**int** cantitate = rs.getInt("cantitate");

**float** pret = rs.getFloat("pret");

System.***out***.println("ID: " + id + ": " + name + " " + cantitate + " " + pret);

}

} **catch** (SQLException e) {

e.printStackTrace();

} **finally** {

mysqlConnect.disconnect();

}

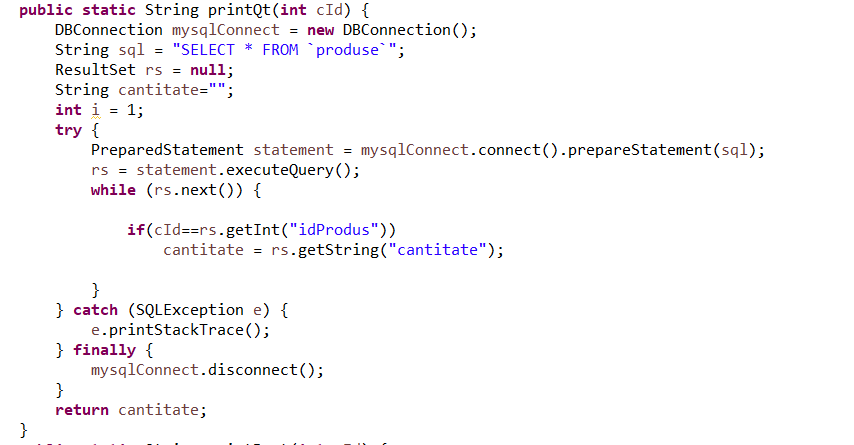
}

The method has an argument, which is the String toBeFound. I chose to do the search after the name of the client, which can be seen in the query:

"SELECT \* FROM `produse` where nume=?"

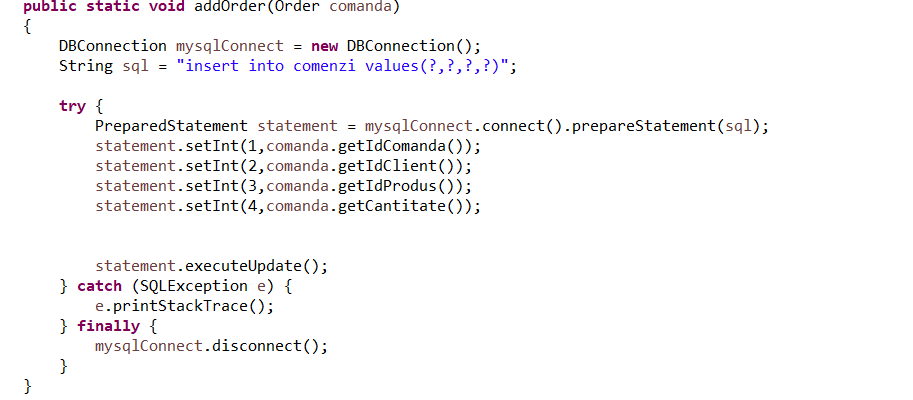
statement.setString(1,toBeFound);

An example of a class for printing the client/product attributes for the JTable: (Print quantity of product)

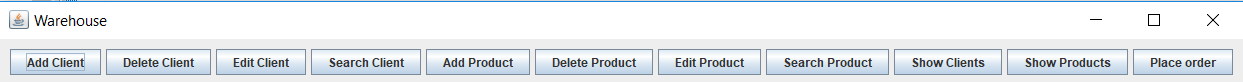


The query is just a select all from the table, but here we return the String in order to be printed in the table. We also check if the client ID is the same as the one we want to be printed. In this case we avoid duplicates and make sure each entry in the JTable is unique.

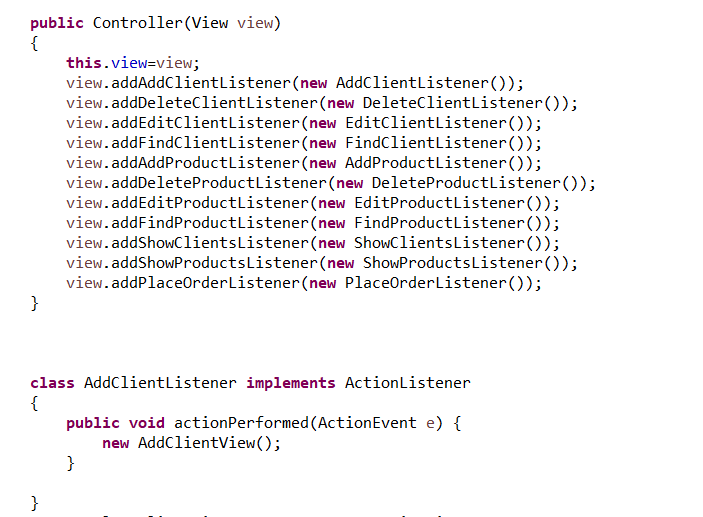
The **OrderOperations** class only has two methods: selectOrders and addOrder.



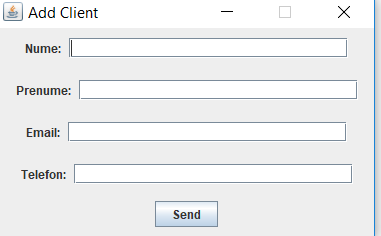
Now, for the interface I chose something as simple as having a few buttons, each with a specific name, that when pressed are going to open a new window for each specific operation.



In the Controller class, I have implemented the action listeners for each button.



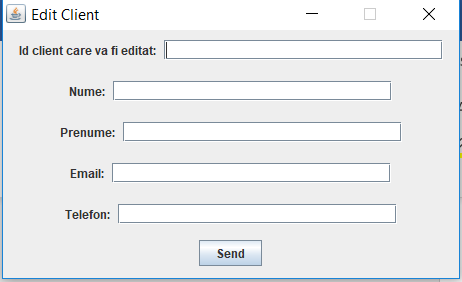
For each listener there is a new interface being opened. For instance the AddClientView looks like this:



After writing the specific attributes and pressing Send, the database updates with the new client entry.

For the Delete interface, the user has to write the client’s id which is going to be deleted and press Send, which will automatically update the database with the deleted entry.

For the Edit Client interface, the user has to write the client’s id which is going to be edited and then write the new attributes for that specific client.



The product operations interfaces are basically the same as the client ones, so I’m not going to explain them with detail.

The last buttons are for “Show Clients” and “Show Products” which both use the JTable class.

The JTable class is a part of Java Swing Package and is generally used to display or edit two-dimensional data that is having both rows and columns. It is similar to a spreadsheet. This arranges data in a tabular form.

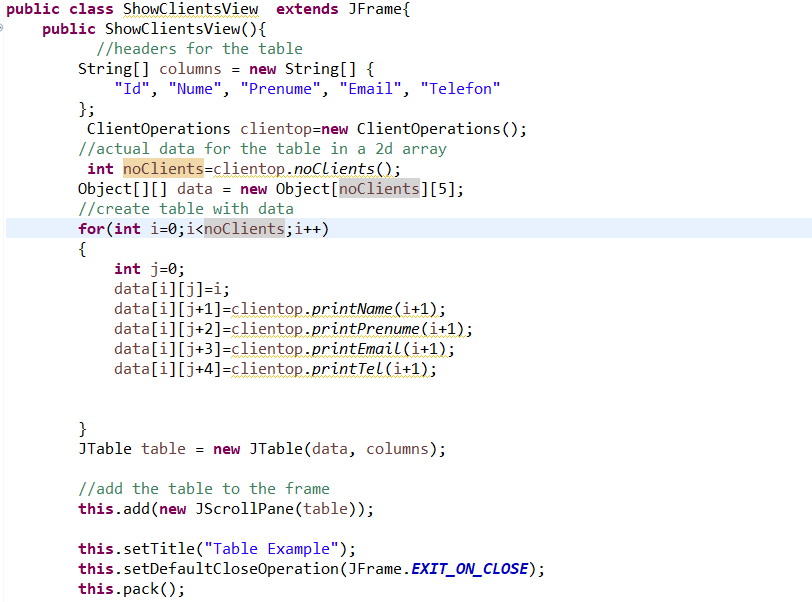
**Constructors in JTable**:

1. **JTable():** A table is created with empty cells.
2. **JTable(int rows, int cols):** Creates a table of size rows \* cols.
3. **JTable(Object[][] data, Object []Column):** A table is created with the specified name where []Column defines the column names.

**Functions in JTable**:

1. **addColumn(TableColumn []column) :** adds a column at the end of the JTable.
2. **clearSelection() :** Selects all the selected rows and columns.
3. **editCellAt(int row, int col) :** edits the intersecting cell of the column number col and row number row programmatically, if the given indices are valid and the corresponding cell is editable.
4. **setValueAt(Object value, int row, int col) :** Sets the cell value as ‘value’ for the position row, col in the JTable.[[2]](#footnote-2)

The class looks like this:



**public** **class** ShowClientsView **extends** JFrame{

**public** ShowClientsView(){

//headers for the table

String[] columns = **new** String[] {

"Id", "Nume", "Prenume", "Email", "Telefon"

};

ClientOperations clientop=**new** ClientOperations();

//actual data for the table in a 2d array

**int** noClients=clientop.*noClients*();

Object[][] data = **new** Object[noClients][5];

//create table with data

**for**(**int** i=0;i<noClients;i++)

{

**int** j=0;

data[i][j]=i;

data[i][j+1]=clientop.*printName*(i+1);

data[i][j+2]=clientop.*printPrenume*(i+1);

data[i][j+3]=clientop.*printEmail*(i+1);

data[i][j+4]=clientop.*printTel*(i+1);

}

JTable table = **new** JTable(data, columns);

//add the table to the frame

**this**.add(**new** JScrollPane(table));

**this**.setTitle("Table Example");

**this**.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

**this**.pack();

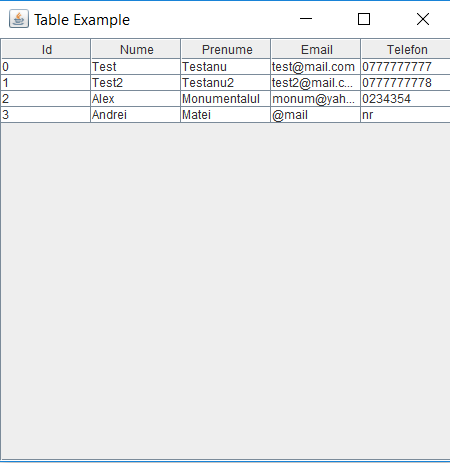
**this**.setVisible(**true**);

}

}

We have a data Object[][] which is used to store the number of rows with the information of each item.

We have an iteration to the number of clients, and we use the print methods from the ClientOperations class to print the specific attributes in the table.

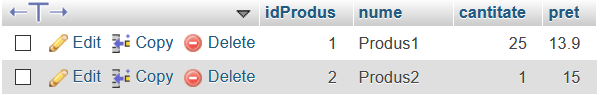


**5. Results**

As expected, the database always updates after adding, deleting or editing an item. Also, in order to place an order you have to introduce the client id and the product id, and also the quantity needed.

For viewing the database I used WAMP and phpmyadmin, so the results were clear and correct. After adding a client, for instance, and refreshing the phpmyadmin web interface you can instantly see the updates to the table.

**Before:**



**After:**



**6. Conclusions**

In conclusion, the application works and can be upgraded to do more queries, and maybe even a more complex search feature.

It is easy to implement and requires a bit more attention than other applications, because you can get lost in the classes and in the queries from the operation classes.

1. https://docs.oracle.com/javase/tutorial/jdbc/basics/processingsqlstatements.html [↑](#footnote-ref-1)
2. https://www.geeksforgeeks.org/java-swing-jtable/ [↑](#footnote-ref-2)