Francisco Rois Siso

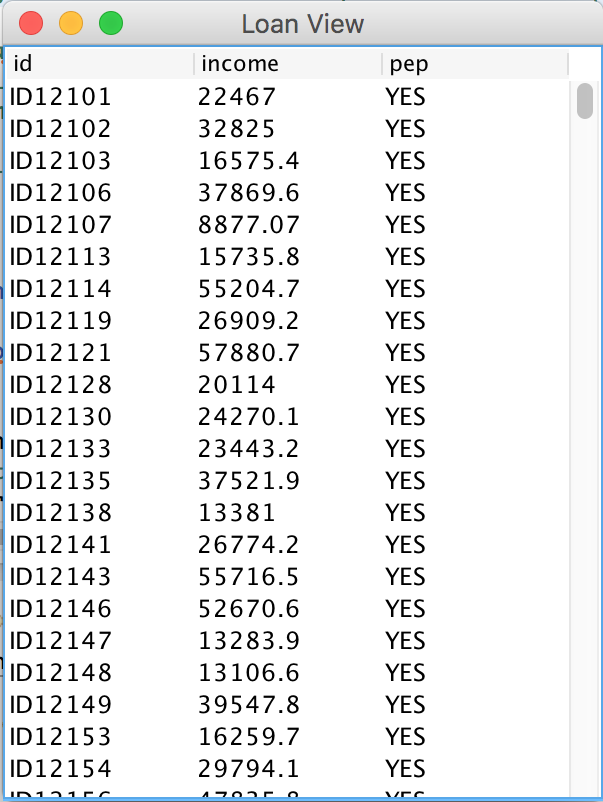
A20385948

03/20/2017

Lab4- ITMD510 Object-Oriented Application Development

**Content**

1. Snapshot of GUI
2. Package bank
   1. BankRecords.java
   2. Client.java
   3. Records.java
   4. RecordsRegionAndAgeComparator.java
   5. RecordsSexAndMortgageComparator.java
   6. RecordsSexRegionCarAndNChildComparator.java
3. Package controllers
   1. LoanConnector.java
   2. LoanController.java
   3. PersistentObject.java
4. Package models
   1. Connector.java
   2. daoModel.java
5. Pack views
   1. LoanView.java
6. **SNAPSHOT OF GUI**

****

*Snapshot of data from bank records displayed graphically.*

1. **Package bank**
   1. **BankRecords.java**

**package** bank;

**import** java.io.BufferedReader;

**import** java.io.FileNotFoundException;

**import** java.io.FileReader;

**import** java.io.IOException;

**import** java.text.SimpleDateFormat;

**import** java.util.ArrayList;

**import** java.util.Arrays;

**import** java.util.Calendar;

**import** java.util.List;

**import** java.io.Serializable;

/\*\*

\* The class BankRecords extends the abstract class Client and implements its methods readData, processData

\* and printData. The purposes of these methods are reading clients' information from a certain csv file, convert

\* each line into an object with the correct parameters and then print information from several of the records.

\*

\* **@author** Francisco Rois Siso

\*

\*/

**public** **class** BankRecords **extends** Client **implements** Serializable{

**private** **static** **final** **long** *serialVersionUID* = -3863252675581079520L;

// default file from which the reading is done if not other file is specified.

**final** **static** String *DEFAULT\_FILE* = "bank-Detail.csv";

// list of objects BankRecords to store all the clients with their information.

**public** List<BankRecords> bankRecords\_list = **new** ArrayList<>();

// list of lists of Strings. Each of the lists is a line (corresponding to a client) and each String is a client's parameter

**static** List<List<String>> *listOfLines* = **new** ArrayList<>();

// list with all the possible values for the field "region" in the Bank Records

List <String> possible\_regions = **new** ArrayList<>();

// parameters of the client

**private** String id;

**private** **int** age;

**private** String sex;

**private** String region;

**private** **double** income;

**private** String married;

**private** **int** children;

**private** String car;

**private** String save\_act;

**private** String current\_act;

**private** String mortgage;

**private** String pep;

/\*\*

\* Constructor without input parameters.

\*/

**public** BankRecords(){}

/\*\*

\* Constructor with all the parameters of the client.

\* **@param** id Identifier as String

\* **@param** age Age as int

\* **@param** sex Sex as String. It can be FEMALE or MALE.

\* **@param** region Region as String. It can be INNER\_CITY, TOWN, RURAL or SUBURBAN.

\* **@param** income Income as double.

\* **@param** married Shows if the client is married or not, as a String. It can be NO or YES

\* **@param** children The number of children that the client has, as an int. It can be 0,1,2 or 3.

\* **@param** car Shows if the client has car or not, as a String. It can be NO or YES.

\* **@param** save\_act

\* **@param** current\_act

\* **@param** mortgage Shows if the client has mortgage or not. It can be NO or YES

\* **@param** pep Shows if the client is a "Politically Exposed Person"

\*/

**public** BankRecords(String id, **int** age, String sex,

String region, **double** income, String married,

**int** children, String car, String save\_act,

String current\_act, String mortgage, String pep){

**this**.id = id;

**this**.age = age;

**this**.sex = sex;

**this**.region = region;

**this**.age = age;

**this**.income = income;

**this**.married = married;

**this**.children = children;

**this**.car = car;

**this**.save\_act = save\_act;

**this**.current\_act = current\_act;

**this**.mortgage = mortgage;

**this**.pep = pep;

}

/\*\*

\* Getters and setters for the parameters of the client.

\*/

**public** String getId() {

**return** id;

}

**public** **void** setId(String id) {

**this**.id = id;

}

**public** **int** getAge() {

**return** age;

}

**public** **void** setAge(**int** age) {

**this**.age = age;

}

**public** String getSex() {

**return** sex;

}

**public** **void** setSex(String sex) {

**this**.sex = sex;

}

**public** String getRegion() {

**return** region;

}

**public** **void** setRegion(String region) {

**this**.region = region;

}

**public** **double** getIncome() {

**return** income;

}

**public** **void** setIncome(**double** income) {

**this**.income = income;

}

**public** String getMarried() {

**return** married;

}

**public** **void** setMarried(String married) {

**this**.married = married;

}

**public** **int** getChildren() {

**return** children;

}

**public** **void** setChildren(**int** children) {

**this**.children = children;

}

**public** String getCar() {

**return** car;

}

**public** **void** setCar(String car) {

**this**.car = car;

}

**public** String getSave\_act() {

**return** save\_act;

}

**public** **void** setSave\_act(String save\_act) {

**this**.save\_act = save\_act;

}

**public** String getCurrent\_act() {

**return** current\_act;

}

**public** **void** setCurrent\_act(String current\_act) {

**this**.current\_act = current\_act;

}

**public** String getMortgage() {

**return** mortgage;

}

**public** **void** setMortgage(String mortgage) {

**this**.mortgage = mortgage;

}

**public** String getPep() {

**return** pep;

}

**public** **void** setPep(String pep) {

**this**.pep = pep;

}

/\*\*

\* Override method toString in order to show the proper String with the parameters of the bank record

\* **@return** String The chain of parameters of the bank record.

\*/

@Override

**public** String toString() {

**return** "BankRecords [id=" + id + ", age=" + age + ", sex=" + sex

+ ", region=" + region + ", income=" + income + ", married="

+ married + ", children=" + children + ", car=" + car

+ ", save\_act=" + save\_act + ", current\_act=" + current\_act

+ ", mortgage=" + mortgage + ", pep=" + pep + "]";

}

/\*\*

\* readData method reads all the data from the file csv indicated and stores it into an ArrayList.

\* Then it calls to the method processData in order to continue the flow of actions.

\* **@param** fileName String indicating the name of the file to read the data from.

\*/

**public** **void** readData(String fileName){

// if the contrary is not specified, the reading is done from the default file.

String file2Read = *DEFAULT\_FILE*;

**if**(!fileName.equals(file2Read)&&!fileName.equals("default")){

file2Read = fileName;

}

// separator used between fields in the document.

**final** String SEPARATOR = ",";

// line that will correspond to a client with its data.

String line = "";

// try to read from file.

**try**(BufferedReader br = **new** BufferedReader(**new** FileReader(file2Read))){

// while there are more lines to read, continue reading.

**while**((line = br.readLine()) != **null**){

// add the new line to the list of lines. The output from the split method is an array,

//therefore it is converted into an ArrayList in order to include the line in the list of lines.

*listOfLines*.add(Arrays.*asList*(line.split(SEPARATOR)));

}

// inform the user when the process is correctly finished.

System.*out*.println("\n--> File correctly read.");

// continue the chain of actions by processing the data.

processData();

} **catch**(FileNotFoundException e){

System.*out*.println("The name of the file specified is not correct.");

}**catch**(IOException e){

System.*out*.println("Please, introduce a correct file name.");

}

}

/\*\*

\* processData method processes the ArrayList from readData and adds the data into each of the bank records objects.

\* It makes use of the setters defined in this class.

\* Then it calls the method printData in order to continue the flow of actions.

\*/

**public** **void** processData() {

// create a new object BankRecords for each line in listOfLines and use setters to add the different features of the clients.

**for**(List<String> client\_fields: *listOfLines*){

BankRecords b = **new** BankRecords();

b.setId(client\_fields.get(0));

b.setAge(Integer.*parseInt*(client\_fields.get(1)));

b.setSex(client\_fields.get(2));

b.setRegion(client\_fields.get(3));

b.setIncome(Double.*parseDouble*(client\_fields.get(4)));

b.setMarried(client\_fields.get(5));

b.setChildren(Integer.*parseInt*(client\_fields.get(6)));

b.setCar(client\_fields.get(7));

b.setSave\_act(client\_fields.get(8));

b.setCurrent\_act(client\_fields.get(9));

b.setMortgage(client\_fields.get(10));

b.setPep(client\_fields.get(11));

// once the client's features are correctly set, the client is added to the list of bank records.

bankRecords\_list.add(b);

**if**(!possible\_regions.contains(b.getRegion()))

possible\_regions.add(b.getRegion());

}

// inform the user of successful processing.

System.*out*.println("\n--> Data from file correctly processed.");

// print on screen date and time for lab submission purposes.

//showDateAndTime();

// continue the chain of actions by printing data.

//printData();

}

/\*\*

\* printData method prints the first 25 records for various fields to the console.

\* It makes use of the getters defined in this class.

\* The records printed are ID, AGE, SEX, REGION, INCOME and MORTGAGE.

\*/

**public** **void** printData() {

// Inform the user of the data to be printed on screen.

System.*out*.println("\n--> Here is some data from the first 25 clients in the document:");

// take the first 25 clients from the list of bank records and print information from them

**for**(**int** i = 0;i<25;i++){

BankRecords brec = bankRecords\_list.get(i);

// build String chain in order to print it.

String chain\_client= "\n\t>>>>> CLIENT " + (i+1) + " <<<<<\n"

+ "\t-------------------------" + "\n"

+ "\tID:\t\t" + brec.getId() + "\n"

+ "\tAGE:\t\t" + brec.getAge() + "\n"

+ "\tSEX:\t\t" + brec.getSex() + "\n"

+ "\tREGION:\t\t" + brec.getRegion() + "\n"

+ "\tINCOME:\t\t$" + brec.getIncome() + "\n"

+ "\tMORTGAGE:\t" + brec.getMortgage() + "\n"

+ "\n\n";

// once the chain is built, print it on screen.

System.*out*.print(chain\_client);

}

}

/\*\*

\* showDateAndTime allows to show the current date and time for lab submission purposes

\*/

**static** **void** showDateAndTime(){

String timeStamp = **new** SimpleDateFormat("yyyy/MM/dd HH:mm:ss").format(Calendar.*getInstance*().getTime());

System.*out*.println("\n\nCur dt=" + timeStamp + "\nProgrammed by Francisco Rois Siso\n");

}

}

* 1. **Client.java**

**package** bank;

/\*\*

\* The class Client provides three abstract methods the bank needs in order to process data from a file: readData, processData and printData.

\*

\* **@author** Francisco Rois Siso

\*

\*/

**public** **abstract** **class** Client {

// abstract methods that will be defined in BankRecords.java.

// a String parameter called fileName is included in "readData" in order to be able

// to read information from different files in future implementations.

// The input String "default" triggers the reading from the default file ("bank-Detail.csv")

**public** **abstract** **void** readData(String fileName);

**public** **abstract** **void** processData();

**public** **abstract** **void** printData();

}

* 1. **Records.java**

**package** bank;

**import** java.util.Collections;

**import** java.util.HashMap;

**import** java.util.Map;

/\*\*

\* The class Records has methods for data processing and analysis of the data read and stored in BankRecords.java.

\* The class Records extends BankRecords, therefore it inherits its methods and variables.

\*

\* **@author** Francisco Rois Siso

\*

\*/

**public** **class** Records **extends** BankRecords{

/\*\*

\* Constructor with no input parameters.

\*/

**public** Records(){}

/\*\*

\* method getAverageIncome goes through the list of bank records and calculate the total average income

\* **@return** averageIncome, as a double

\*/

**public** **double** getAverageIncome(){

**double** averageIncome = 0.0;

**for**(BankRecords b:**super**.bankRecords\_list){

averageIncome += b.getIncome();

}

averageIncome /= **super**.bankRecords\_list.size();

**return** averageIncome;

}

/\*\*

\* method getMinAgesPerLocation sorts the list of bank records by region and then creates a map with the minimum age for each of them.

\* **@return** map\_mins a map with the regions as key and the minimum age as values, as a Map<String,Integer>

\*/

**public** Map<String,Integer> getMinAgesPerLocation(){

Map<String,Integer> map\_mins = **new** HashMap<String,Integer>();

// sort the BankRecords list by region and age

Collections.*sort*(**super**.bankRecords\_list,**new** RecordsRegionAndAgeComparator());

// sort possible regions for info displaying purposes

Collections.*sort*(**super**.possible\_regions);

**for**(BankRecords b: **super**.bankRecords\_list){

// insert in map the first age value found for each region, which is the minimum

**if**(!map\_mins.containsKey(b.getRegion())){

// insert minimum in map

map\_mins.put(b.getRegion(), b.getAge());

}

}

**return** map\_mins;

}

/\*\*

\* method getMaxAgesPerLocation sorts the list of bank records by region and then creates a map with the maximum age for each of them.

\* **@return** map\_maxs a map with the regions as key and the maximum age as values, as a Map<String,Integer>

\*/

**public** Map<String,Integer> getMaxAgesPerLocation(){

Map<String,Integer> map\_maxs = **new** HashMap<String,Integer>();

// sort the BankRecords list by region and age

Collections.*sort*(**super**.bankRecords\_list,**new** RecordsRegionAndAgeComparator());

// sort possible regions for info displaying purposes

Collections.*sort*(**super**.possible\_regions);

// go through the sorted list, but this time backwards, so the first values found are the maximums

**for**(**int** i = **super**.bankRecords\_list.size()-1 ; i>=0 ; i--){

BankRecords b = **super**.bankRecords\_list.get(i);

**if**(!map\_maxs.containsKey(b.getRegion())){

// insert in map the only the first age value found for each region, which is the maximum

map\_maxs.put(b.getRegion(), b.getAge());

}

}

**return** map\_maxs;

}

/\*\*

\* method getNumberOfFemalesWithMortgages sorts the list of bank records by sex and mortgage. Then it goes through the sorted list

\* and counts the number of females with mortgage.

\* **@return** count the number of females with a mortgage, as an int

\*/

**public** **int** getNumberOfFemalesWithMortgages(){

// sort list by sex and mortgage

Collections.*sort*(**super**.bankRecords\_list,**new** RecordsSexAndMortgageComparator());

// go through the sorted list only until a record not female and without mortgage is found

**int** count = 0;

**int** e = 0;

**while**(**super**.bankRecords\_list.get(e).getSex().equals("FEMALE") && **super**.bankRecords\_list.get(e).getMortgage().equals("YES")){

e++;

count ++;

}

**return** count;

}

/\*\*

\* method getNumberOfMalesWithCarAndOneChildPerLocation first sorts the list of bank records by sex, region, car and number of children.

\* Then the method goes through the collection and counts the number of males with car and one child, per region.

\* **@return** map a map with the regions as key and the number of males with car and 1 child as value for each of the regions

\*/

**public** Map<String,Integer> getNumberOfMalesWithCarAndOneChildPerLocation(){

// sort list by sex, location, car and number of child

Collections.*sort*(**super**.bankRecords\_list, **new** RecordsSexRegionCarAndNChildComparator());

// map to store number of specified elements (value) per region (key)

Map<String,Integer> map = **new** HashMap<String,Integer>();

**int** i = 0;

**int** count = 0;

// while it is a man and he has car

**while**(i < **super**.bankRecords\_list.size() && **super**.bankRecords\_list.get(i).getSex().equals("MALE") && **super**.bankRecords\_list.get(i).getCar().equals("YES")){

// if he has 1 child, increment the count

**if**(**super**.bankRecords\_list.get(i).getChildren() == 1){

count++;

}

// if index in bound

**if**((i+1) < **super**.bankRecords\_list.size()) {

// if i is the last item of the certain region, put the count and the region into the map and restart the count for the next region

**if**(!**super**.bankRecords\_list.get(i).getRegion().equals(**super**.bankRecords\_list.get(i+1).getRegion())){

map.put(**super**.bankRecords\_list.get(i).getRegion(), count);

count = 0;

}

}

// move to the next element in the collection

i++;

}

**return** map;

}

}

* 1. **RecordsRegionAndAgeComparator.java**

**package** bank;

**import** java.util.Comparator;

/\*\*

\* Class used to compare two objects BankRecords by their region and age.

\*

\* **@author** Francisco Rois Siso

\*

\*/

**public** **class** RecordsRegionAndAgeComparator **implements** Comparator<BankRecords>{

@Override

**public** **int** compare(BankRecords rec1, BankRecords rec2) {

// if different region

**if**(!rec1.getRegion().equals(rec2.getRegion())){

**return** rec1.getRegion().compareTo(rec2.getRegion());

}

// if equal region

**else**{

**return** rec1.getAge()-rec2.getAge();

}

}

}

* 1. **RecordsSexAndMortgageComparator.java**

**package** bank;

**import** java.util.Comparator;

/\*\*

\* Class used to compare two objects BankRecords by their sex and mortgage.

\*

\* **@author** Francisco Rois Siso

\*

\*/

**public** **class** RecordsSexAndMortgageComparator **implements** Comparator<BankRecords>{

@Override

**public** **int** compare(BankRecords rec1, BankRecords rec2) {

// if different sex

**if**(!rec1.getSex().equals(rec2.getSex()))

**return** rec1.getSex().compareTo(rec2.getSex());

// if equal sex

**else**

**return** rec2.getMortgage().compareTo(rec1.getMortgage());

}

}

* 1. **RecordsSexRegionCarAndNChildComparator.java**

**package** bank;

**import** java.util.Comparator;

/\*\*

\* Class used to compare two objects BankRecords by their sex, region, car and number of children.

\*

\* **@author** Francisco Rois Siso

\*

\*/

**public** **class** RecordsSexRegionCarAndNChildComparator **implements** Comparator<BankRecords> {

@Override

**public** **int** compare(BankRecords rec1,BankRecords rec2){

// if different sex

**if**(!rec1.getSex().equals(rec2.getSex())){

**return** rec2.getSex().compareTo(rec1.getSex());

}

**else**{

// if equal sex but different car

**if**(!rec1.getCar().equals(rec2.getCar())){

**return** rec2.getCar().compareTo(rec1.getCar());

}

**else**{

// if equal sex, equal car but different region

**if**(!rec1.getRegion().equals(rec2.getRegion())){

**return** rec1.getRegion().compareTo(rec2.getRegion());

}

// if equal sex, equal car, equal region, then compare by number of children

**else**{

**return** rec1.getChildren()-rec2.getChildren();

}

}

}

}

}

1. **Package controllers**
   1. **LoanConnector.java**

**package** controllers;

/\*\*

\* The class LoanConnector provides connection to the list of bank records, generated in bank.BankRecords and serves for potential future funtionalities.

\* The class LoanConnector extends bankRecords, therefore it inherits its methods and variables.

\*

\* **@author** Francisco Rois Siso

\*

\*/

**public** **class** LoanConnector **extends** bank.BankRecords{

**private** **static** **final** **long** *serialVersionUID* = 4919150415235341463L;

**public** LoanConnector(){

}

}

* 1. **LoanController.java**

**package** controllers;

**import** java.io.FileInputStream;

**import** java.io.FileNotFoundException;

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**import** java.io.ObjectInputStream;

**import** java.io.ObjectOutputStream;

**import** java.sql.Connection;

**import** java.sql.ResultSet;

**import** java.sql.SQLException;

**import** java.util.Date;

**import** java.util.HashMap;

**import** java.util.Map;

**import** bank.BankRecords;

**import** models.\*;

**import** views.\*;

/\*

\* Program to read bank records data from a file, process it, store it in a MySQL database and present it as a GUI.

\* After showing the data on the screen, the application serializes bank records data, sleeps for 5 seconds and deserializes

\* it again, showing in console the time elapsed between both processes.

\*

\* - Programmer: Francisco Rois Siso

\* - Date: 03/20/2017

\* - Source File Name: LoanController.java

\* - Lab 4

\* - ITMD510 Object-Oriented Application Development

\*/

/\*\*

\* The class LoanController contains the main and therefore is the driver file of the program.

\*

\* **@author** Francisco Rois Siso

\*

\*/

**public** **class** LoanController{

// data needed to connect to the database

**final** **static** String *DB\_URL* = "jdbc:mysql://www.papademas.net:3306/510labs?autoReconnect=true&useSSL=false";

**final** **static** String *USERNAME* = "db510";

**final** **static** String *PASSWORD* = "510";

// name of the table to create

**static** String *table\_name* = "f\_rois\_tab";

// map with the bank records read from a file

**static** Map<String,BankRecords> *bankRecords*;

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

// create a new object LoanConnector, which will be used to read data from a file

LoanConnector lc = **new** LoanConnector();

// read data from the default csv file and process it (storage)

lc.readData("default");

// Collect BankRecords into a HashMap

*bankRecords* = **new** HashMap<>();

**for**(BankRecords b: lc.bankRecords\_list){

*bankRecords*.put(b.getId(), b);

}

// create connection to the database using the access data

Connector connector = **new** Connector();

Connection connection = connector.createConnection(*DB\_URL*, *USERNAME*, *PASSWORD*);

// daoModel object created in order to execute CRUD functions

daoModel dm = **new** daoModel();

dm.createTable(connection, *table\_name*);

//dm.dropTable(connection, table\_name);

dm.inserts(connection, lc.bankRecords\_list, *table\_name*);

//dm.deleteAllFromTable(connection, table\_name);

// Extract data from result set

**try** {

ResultSet rs = dm.getResultSet(connection, *table\_name*);

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

//Retrieve by column name

String id = rs.getString("id");

**double** income = rs.getDouble("income");

String pep = rs.getString("pep");

//Display values

System.*out*.print("ID: " + id);

System.*out*.print(", Income: " + String.*valueOf*(income));

System.*out*.println(", Pep: " + pep);

// create object LoanView for graphic interface display

LoanView lw = **new** LoanView(rs);

}

} **catch** (SQLException e1) {

System.*out*.println(e1.getMessage());

}

// Close connection to database

**try** {

connection.close();

System.*out*.println("\nConnection to db closed");

} **catch** (SQLException e) {

System.*out*.println(e.getMessage());

}

// Create an instance of PersistentObject with the map of BankRecords and the current timestamp

PersistentObject po = **new** PersistentObject(*bankRecords*,**new** Date());

//Serialize PersistentObject to a file

FileOutputStream outStream = **null**;

ObjectOutputStream objectOutputFile = **null**;

**try** {

outStream = **new** FileOutputStream("bankrecords.ser");

objectOutputFile = **new** ObjectOutputStream(outStream);

objectOutputFile.writeObject(po);

} **catch** (FileNotFoundException e2) {

System.*out*.println(e2.getMessage());

} **catch** (IOException e) {

e.printStackTrace();

System.*out*.println(e.getMessage());

}

// Make application sleep for 5 seconds

**try** {

Thread.*sleep*(5000);

} **catch** (InterruptedException e) {

System.*out*.println(e.getMessage());

}

// De-serialize the persistent object into a date object and a new map object

FileInputStream inStream = **null**;

ObjectInputStream objectInputStream = **null**;

**try** {

inStream = **new** FileInputStream("bankRecords.ser");

objectInputStream = **new** ObjectInputStream(inStream);

} **catch** (FileNotFoundException e1) {

System.*out*.println(e1.getMessage());

} **catch** (IOException e2) {

System.*out*.println(e2.getMessage());

}

// new PersistentObject object, to store deserialized information

PersistentObject new\_po = **null**;

**try** {

new\_po = (PersistentObject) objectInputStream.readObject();

} **catch** (ClassNotFoundException | IOException e) {

System.*out*.println(e.getMessage());

}

// new map to store the bank records obtained from deserialization of file

Map<String,BankRecords> newBankRecords = new\_po.getBankRecords();

// new date object obtained from deserialization of file

Date newDate = new\_po.getDate();

// System.out.println("\nDESERIALIZED MAP");

// for(String key: newBankRecords.keySet()){

// System.out.println(newBankRecords.get(key));

// }

//

// System.out.println("\nDESERIALIZED DATE");

// System.out.println(newDate);

// Display to the console the time difference between serialization and de-serialization

System.*out*.println("\nTime difference between serialization and deserialization: "+(((**new** Date()).getTime()-newDate.getTime())/1000.00) +" seconds");

}

}

* 1. **PersistentObject.java**

**package** controllers;

**import** java.io.Serializable;

**import** java.util.Date;

**import** java.util.HashMap;

**import** java.util.Map;

**import** bank.BankRecords;

/\*\*

\* The class Persistent encapsulates a map of bank records and a date.

\* The class Persistent implements java.io.Serializable.

\*

\* **@author** Francisco Rois Siso

\*

\*/

**public** **class** PersistentObject **implements** Serializable{

**private** **static** **final** **long** *serialVersionUID* = 8711159212803491294L;

**private** Map<String,BankRecords> bankRecords;

**private** Date date;

/\*\*

\* Constructor with no input parameters.

\*/

**public** PersistentObject(){}

/\*\*

\* Constructor with two parameters: bankRecords and date.

\* **@param** bankRecords Map with the bank records stored by id (key)

\* **@param** date

\*/

**public** PersistentObject(Map<String,BankRecords> bankRecords, Date date){

**this**.setBankRecords(bankRecords);

bankRecords = **new** HashMap<>();

**this**.setDate(date);

}

/\*\*

\* Getters and setters for bankRecords and date.

\*/

**public** Map<String,BankRecords> getBankRecords() {

**return** bankRecords;

}

**public** **void** setBankRecords(Map<String,BankRecords> bankRecords) {

**this**.bankRecords = bankRecords;

}

**public** Date getDate() {

**return** date;

}

**public** **void** setDate(Date date) {

**this**.date = date;

}

}

1. **Package models**
   1. **Connector.java**

**package** models;

**import** java.sql.\*;

/\*\*

\* The class Connector has a single method (createConnection) to establish a connection to a certain database.

\*

\* **@author** Francisco Rois Siso

\*/

**public** **class** Connector {

/\*\*

\* Constructor with no input parameters

\*/

**public** Connector(){

}

/\*\*

\* Method createConnection establishes a connection to the database indicated as input parameter, by using the data introduced as input parameters as well.

\*

\* **@param** db\_url URL of the database to access

\* **@param** username Username to access the database, as String

\* **@param** password Password to acces the database with the username indicated, as String

\* **@return** connection An object Connection with the connection established to the certain database.

\*/

**public** Connection createConnection(String db\_url, String username, String password){

Connection connection = **null**;

**try**{

// Create connection to the database

connection = DriverManager.*getConnection*(db\_url, username, password);

System.*out*.println("\nConnection to "+db\_url+" created");

}

**catch**(Exception e){

System.*out*.println("ERROR: " + e.getMessage());

}

**return** connection;

}

}

* 1. **daoModel.java**

**package** models;

**import** java.sql.\*;

**import** java.util.List;

**import** bank.BankRecords;

/\*\*

\* The class daoModel allows to process all necessary CRUD operations.

\*

\* **@author** Francisco Rois Siso

\*

\*/

**public** **class** daoModel {

/\*\*

\* Method createTable creates a new table in the database, if it does not exist yet, using the connection already established.

\* **@param** connection Connection already established to a database

\* **@param** table\_name Name of the table to be created, as String

\*/

**public** **void** createTable(Connection connection, String table\_name){

**try**{

// Execute a query to create the table

System.*out*.println("Creating table in database ...");

Statement statement = connection.createStatement();

String sql = "CREATE TABLE "+ table\_name +

"(id VARCHAR(255) not NULL, " +

" income DOUBLE, " +

" pep VARCHAR(255), " +

" PRIMARY KEY ( id ))";

statement.executeUpdate(sql);

System.*out*.println("Created table in given database...");

statement.close();

}

**catch**(SQLException se){

//Handle errors for JDBC

System.*out*.println(se.getMessage());

}**catch**(Exception e){

//Handle errors for Class.forName

System.*out*.println(e.getMessage());

}

}

/\*\*

\* Method dropTable allows to drop a table from a database by using the connection provided.

\* **@param** connection Connection already established to a database.

\* **@param** table\_name Name of the table to be dropped, as String.

\*/

**public** **void** dropTable(Connection connection, String table\_name){

**try**{

// Execute a query to drop the table from the database

System.*out*.println("Droping table in database ...");

Statement statement = connection.createStatement();

String sql = "DROP TABLE "+ table\_name;

statement.executeUpdate(sql);

System.*out*.println("Droped table in given database...");

statement.close();

}

**catch**(SQLException se){

//Handle errors for JDBC

System.*out*.println(se.getMessage());

}**catch**(Exception e){

//Handle errors for Class.forName

System.*out*.println(e.getMessage());

}

}

/\*\*

\* Method deleteAllFromTable allows to delete all the rows of a certain database.

\* **@param** connection Connection already established to a database.

\* **@param** table\_name Name of the table to be cleared, as String.

\*/

**public** **void** deleteAllFromTable(Connection connection, String table\_name){

**try**{

// Execute a query to delete all from the table

System.*out*.println("Deleting all from table in database ...");

Statement statement = connection.createStatement();

String sql = "DELETE FROM "+ table\_name;

statement.executeUpdate(sql);

System.*out*.println("Deleted all rows from table in given database...");

statement.close();

}

**catch**(SQLException se){

//Handle errors for JDBC

System.*out*.println(se.getMessage());

}**catch**(Exception e){

//Handle errors for Class.forName

System.*out*.println(e.getMessage());

}

}

/\*\*

\* Method inserts allows for a list of BankRecords objects to be passed to it and thus insert all record object

\* field data consisting of the id, income and pep from the list into a database.

\* **@param** connection Connection already established to a database.

\* **@param** records List of BankRecords to be stored into the database.

\* **@param** table\_name Name of the table where the items are inserted.

\*/

**public** **void** inserts(Connection connection, List<BankRecords> records, String table\_name){

String sql = "";

**try**{

// Execute a query

System.*out*.println("Inserting records into the table...");

Statement statement = connection.createStatement();

**for**(**int** i = 0; i<records.size(); i++){

String id = records.get(i).getId();

**double** income = records.get(i).getIncome();

String pep = records.get(i).getPep();

sql = "INSERT INTO "+table\_name+" " +

"VALUES ("+"'"+id+"'"+", "+"'"+String.*valueOf*(income)+"'"+", "+"'"+pep+"'"+")";

//System.out.println("sql: "+sql);

statement.executeUpdate(sql);

}

System.*out*.println("Inserted records into the table...");

statement.close();

}**catch**(SQLException se){

//Handle errors for JDBC

System.*out*.println(se.getMessage());

}**catch**(Exception e){

//Handle errors for Class.forName

System.*out*.println(e.getMessage());

}

}

/\*\*

\* Method getResultSet returns a ResultSet object containing the record data (id, income and pep).

\* The result is sorted in descending order by pep, in order to show first the premium users.

\* **@param** connection Connection already established to a database.

\* **@param** table\_name Table from which the data is retrieved to make the ResultSet.

\* **@return**

\*/

// sorted descending by pep

**public** ResultSet getResultSet(Connection connection, String table\_name){

ResultSet rs = **null**;

**try**{

// Execute a query

Statement statement = connection.createStatement();

// Select by id, income and pep and sort by pep in descending order in order to show first the premium users (pep = YES)

String sql = "SELECT id, income, pep FROM "+table\_name+

" ORDER BY pep DESC";

rs = statement.executeQuery(sql);

}**catch**(SQLException se){

//Handle errors for JDBC

System.*out*.println(se.getMessage());

}**catch**(Exception e){

//Handle errors for Class.forName

System.*out*.println(e.getMessage());

}

**return** rs;

}

}

1. **Pack views**
   1. **LoanView.java**

**package** views;

**import** java.awt.BorderLayout;

**import** java.sql.ResultSet;

**import** java.sql.ResultSetMetaData;

**import** java.sql.SQLException;

**import** javax.swing.JFrame;

**import** javax.swing.JScrollPane;

**import** javax.swing.JTable;

/\*\*

\* The class LoanView allows to create a graphical user interface to display information about the bank records.

\* The class LoanView extends JFrame.

\*

\* **@author** Francisco Rois Siso

\*

\*/

**public** **class** LoanView **extends** JFrame{

**static** **final** **long** *serialVersionUID* = 2158941944719428789L;

**private** String[] columnNames;

**private** String[][] tableData;

/\*\*

\* Constructor with no input parameters.

\*/

**public** LoanView(){}

/\*\*

\* Constructor with one input parameter: a ResultSet with data to be graphically displayed.

\* **@param** rs ResultSet with data to be graphically displayed.

\*/

**public** LoanView(ResultSet rs){

JFrame frame = **new** JFrame();

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

frame.setTitle("Loan View");

// Object rowData[][] = { { "Row1-Column1", "Row1-Column2", "Row1-Column3" },

// { "Row2-Column1", "Row2-Column2", "Row2-Column3" } };

// Object columnNames[] = { "Column One", "Column Two", "Column Three" };

// Get a metadata object for the result set

ResultSetMetaData metadata;

**try** {

// Get the number of rows.

rs.last(); // Move to last row

**int** numRows = rs.getRow(); // Get row number

rs.first(); // Move to first row

//System.out.println("Number of rows: "+numRows);

metadata = rs.getMetaData();

// Create an array of Strings for the column names

columnNames = **new** String[metadata.getColumnCount()];

// Get the names of the columns

//System.out.println("\nColumn names: ");

**for** (**int** i = 0; i<metadata.getColumnCount(); i++){

columnNames[i] = metadata.getColumnName(i+1);

//System.out.println("\t"+columnNames[i]);

}

// Create 2D String array for the data from the table

tableData = **new** String[numRows][metadata.getColumnCount()];

//System.out.println("\nTable Data: ");

// Get data from columns in the tableData array

**for** (**int** row = 0; row < numRows; row++){

**for** (**int** col = 0; col < metadata.getColumnCount(); col++){

tableData[row][col] = rs.getString(col + 1);

//System.out.println("\t"+tableData[row][col]);

}

// Go to the next row in the ResultSet.

rs.next();

}

// create JTable and JScrollPane with the given rows and columns

JTable table = **new** JTable(tableData, columnNames);

JScrollPane scrollPane = **new** JScrollPane(table);

frame.add(scrollPane, BorderLayout.*CENTER*);

} **catch** (SQLException e) {

System.*out*.println(e.getMessage());

} **catch** (Exception ex){

System.*out*.println(ex.getMessage());

}

// Set other graphical features

frame.setSize(300, 400);

frame.setLocationRelativeTo(**null**);

frame.setVisible(**true**);

}

}