Университет ИТМО

Кафедра ВТ

Программирование

Лабораторная работа №8

Группа P3110

Вариант 782227

Нгу Фыонг Ань

Проверил:

Писмак Алексей Евгеньевич

2018 год

Доработать программу из [лабораторной работы №7](https://se.ifmo.ru/courses/programming" \l "lab7) следующим образом:

1. В класс, объекты которого хранятся в коллекции, добавить поле типа java.time.LocalDateTime, в котором должны храниться дата и время создания объекта.
2. Графический интерфейс клиентской части должен поддерживать **русский**, **турецкий**, **хорватский** и **английский (Австралия)** языки / локали. Должно обеспечиваться корректное отображение чисел, даты и времени в соответстии с локалью. Переключение языков должно происходить без перезапуска приложения. Локализованные ресурсы должны храниться в **файле свойств**. Сортировку и фильтрацию значений столбцов реализовать с помощью Streams API.
3. Сервер должен обеспечивать простейшую реализацию объектно-реляционного отображения с использованием рефлексии, в том числе создание таблицы базы данных в соответствии с полями объекта и CRUD-запросы.

**Вопросы к защите лабораторной работы:**

1. Интернационализация. Локализация. Хранение локализованных ресурсов.
2. Форматирование локализованных числовых данных, текста, даты и времени.
3. Классы для представления даты и времени из пакета java.time. Преобразование темпоральных величин.
4. Взаимодействие с базами данных. Протокол JDBC.
5. Рефлексия.
6. Объектно-реляционное отображение (для получения повышенных баллов)

**Исходный код:**

**#Server**

**#ORMUtils**

public class ORMUtils {

static Set<String> modifys = new HashSet<>();

//connect to the database

public static Connection getConn() throws Exception {

String url = "jdbc:postgresql://localhost:5432/postgres";//The MySQL connection URL, do not explain

String user = "postgres";//MySQL users

String password = "becung";//MySQL password

return DriverManager.getConnection(url, user, password);//Get connected

}

//INSERT

public static void doInsert(Humandb data) throws Exception {

List<Field> fieldList = new LinkedList<>();

getAllFeild(fieldList, data.getClass());

StringBuffer keys = new StringBuffer();

StringBuffer values = new StringBuffer();

for (Field field : fieldList) {

if (keys.length() > 0) {

keys.append(",");

values.append(",");

}

keys.append(field.getName());

values.append("?");

}

StringBuffer sql = new StringBuffer();

sql.append("INSERT INTO ");

//Here is the table name, I called the name of the table;

sql.append(data.getClass().getSimpleName());

sql.append(" ( ").append(keys).append(") VALUES (").append(values.toString()).append(");");

// System.out.println(sql);

try (Connection conn = getConn(); PreparedStatement statement = conn.prepareStatement(sql.toString())) {

int index = 1;

for (Field field : fieldList) {

// System.out.println(index + " " + data + " " + field );

setStatement(statement, index, data, field);

index++;

}

System.out.println(statement);

statement.executeUpdate();

}

}

public static List<Humandb> doRead(Humandb h) throws Exception {

List<Field> fieldList = new LinkedList<>();

getAllFeild(fieldList, h.getClass());

LinkedList<Humandb> ll = new LinkedList<>();

String sql = "SELECT \* FROM " + h.getClass().getSimpleName() + ";";

System.out.println(sql);

try (Connection conn = getConn();

PreparedStatement statement = conn.prepareStatement(sql);

ResultSet result = statement.executeQuery()) {

while (result.next()) {

Humandb data = (Humandb) h.getClass().newInstance();

for (Field field : fieldList) {

setValue(result, data, field);

}

ll.add(data);

}

}

return (List<Humandb>) ll;

}

//UPDATE

public static void doUpdate(Humandb data) throws Exception {

List<Field> fieldList = new LinkedList<>();

getAllFeild(fieldList, data.getClass());

Field key = null;

StringBuffer keys = new StringBuffer();

for (Field field : fieldList) {

if (field.isAnnotationPresent(PrimaryKey.class)) {

key = field;

}

if (keys.length() > 0) {

keys.append(",");

}

keys.append(field.getName()).append(" = ?");

}

StringBuffer sql = new StringBuffer();

sql.append("UPDATE ").append(data.getClass().getSimpleName());

sql.append(" SET ").append(keys.toString());

sql.append(" WHERE ").append(key.getName()).append(" LIKE '%").append(data.getId()).append("'").append(";");

System.out.println(sql);

try (Connection conn = getConn();

PreparedStatement statement = conn.prepareStatement(sql.toString())) {

int index = 1;

for (Field field : fieldList) {

setStatement(statement, index, data, field);

index++;

}

statement.executeUpdate();

}

}

//DELETE

public static void doDelete(Humandb data) throws Exception {

doDelete(data.getClass(), data.getId());

}

public static void doDelete(Class<?> clazz, Object Id) throws Exception {

List<Field> fieldList = new LinkedList<>();

getAllFeild(fieldList, clazz);

Field key = fieldList.get(0);

String sql = "DELETE FROM " + clazz.getSimpleName() + " WHERE " + key.getName() + " = " + Id + ";";

System.out.println(sql);

try (Connection conn = getConn();

PreparedStatement statement = conn.prepareStatement(sql)) {

statement.executeUpdate();

}

System.out.println(sql);

}

private static void setStatement(PreparedStatement statement, int index, Humandb data, Field field) throws Exception {

boolean isAccess = field.isAccessible();

try {

field.setAccessible(true);

if (field.getType() == byte.class || field.getType() == Byte.class) {

statement.setByte(index, field.getByte(data));

} else if (field.getType() == boolean.class || field.getType() == Boolean.class) {

statement.setBoolean(index, field.getBoolean(data));

} else if (field.getType() == short.class || field.getType() == Short.class) {

statement.setShort(index, field.getShort(data));

} else if (field.getType() == char.class || field.getType() == Character.class) {

statement.setString(index, String.valueOf(field.getChar(data)));

} else if (field.getType() == int.class || field.getType() == Integer.class) {

statement.setInt(index, field.getInt(data));

} else if (field.getType() == float.class || field.getType() == Float.class) {

statement.setFloat(index, field.getFloat(data));

} else if (field.getType() == long.class || field.getType() == Long.class) {

statement.setLong(index, field.getLong(data));

} else if (field.getType() == double.class || field.getType() == Double.class) {

statement.setDouble(index, field.getDouble(data));

} else if (field.getType() == String.class) {

statement.setString(index, field.get(data).toString());

} else if (field.getType() == LocalDateTime.class) {

LocalDateTime localDateTime = (LocalDateTime) field.get(data);

statement.setTimestamp(index, Timestamp.valueOf(localDateTime));

} else if (field.getType() == Color.class) {

Color color = (Color) field.get(data);

String newColor = "";

if (Color.RED == color) {

newColor = "RED";

} else if (Color.GREEN == color) {

newColor = "GREEN";

} else if (Color.BLACK == color) {

newColor = "BLACK";

} else if (Color.BLUE == color) {

newColor = "BLUE";

} else if (Color.YELLOW == color) {

newColor = "YELLOW";

};

statement.setString(index, newColor);

} else {

throw new RuntimeException("unsupport provided type:" + field.getType());

}

} finally {

field.setAccessible(isAccess);

}

}

private static void setValue(ResultSet result, Humandb data, Field field) throws Exception {

boolean isAccess = field.isAccessible();

try {

field.setAccessible(true);

if (field.getType() == byte.class || field.getType() == Byte.class) {

field.setByte(data, result.getByte(field.getName()));

} else if (field.getType() == boolean.class || field.getType() == Boolean.class) {

field.setBoolean(data, result.getBoolean(field.getName()));

} else if (field.getType() == short.class || field.getType() == Short.class) {

field.setShort(data, result.getShort(field.getName()));

} else if (field.getType() == char.class || field.getType() == Character.class) {

field.set(data, result.getString(field.getName()).charAt(0));

} else if (field.getType() == int.class || field.getType() == Integer.class) {

field.setInt(data, result.getInt(field.getName()));

} else if (field.getType() == float.class || field.getType() == Float.class) {

field.setFloat(data, result.getFloat(field.getName()));

} else if (field.getType() == long.class || field.getType() == Long.class) {

field.setLong(data, result.getLong(field.getName()));

} else if (field.getType() == double.class || field.getType() == Double.class) {

field.setDouble(data, result.getDouble(field.getName()));

} else if (field.getType() == String.class) {

field.set(data, result.getString(field.getName()));

} else if (field.getType() == LocalDateTime.class) {

LocalDateTime localDateTime = result.getTimestamp(field.getName()).toLocalDateTime();

field.set(data, localDateTime);

} else if (field.getType() == Color.class) {

String lol = result.getObject(field.getName()).toString();

System.out.println(result.getObject(field.getName()));

Color color;

try {

Field fieldcolor = Class.forName("java.awt.Color").getField(lol);

color = (Color) fieldcolor.get(null);

} catch (Exception e) {

color = null; // Not defined

}

field.set(data, color);

} else {

throw new RuntimeException("unsupport provided type:" + field.getType());

}

} finally {

field.setAccessible(isAccess);

}

}

private static void getAllFeild(List<Field> fieldList, Class<?> clazz) {

if (clazz == Object.class) {

return;

} else {

Field[] fields = clazz.getDeclaredFields();

for (Field field : fields) {

fieldList.add(field);

}

getAllFeild(fieldList, clazz.getSuperclass());

}

}

public static int getNextId() throws Exception {

String sql = "select max(humanid) from Humandb";

try (Connection conn = getConn();

PreparedStatement statement = conn.prepareStatement(sql);

ResultSet result = statement.executeQuery()) {

if (result.next()) {

return result.getInt("max");

}

}

return 0;

}

}

**#Client**

**…**

jComboBox2.addActionListener((ActionEvent e) -> {

String lang = (String) jComboBox2.getSelectedItem();

if (lang.equals("English(Aus)")){

language = "en";

country = "AU";

}

if (lang.equals("Русский")){

language = "ru";

country = "RU";

}

if (lang.equals("Türk")){

language = "tr";

country = "TR";

}

if (lang.equals("Hrvatski")){

language = "hr";

country = "HR";

}

currentLocale = new Locale(language, country);

messages = ResourceBundle.getBundle("resources.Bundle",currentLocale);

jComboBox1.setModel(new DefaultComboBoxModel<>(new String[]{messages.getString("All"),

messages.getString("Blue"), messages.getString("Yellow"), messages.getString("Green"),

messages.getString("Red")}));

button1.setText(messages.getString("Start"));

button2.setText(messages.getString("Stop"));

timerThread.setLocal(currentLocale);

});

button1.addActionListener((ActionEvent e) -> {

messages = ResourceBundle.getBundle("resources.Bundle",new Locale("en","WW"));

String selectedColor = messages.getString((String) jComboBox1.getSelectedItem());

int value = jSlider1.getValue();

Color colorx;

try {

colorx = (Color) Class.forName("java.awt.Color").getField(selectedColor.toLowerCase()).get(null);

} catch (Exception ew) {

colorx = null; // Not defined

}

// System.out.println(colorx);

for (Circle c : circle) {

if (selectedColor.equals("All")) {

if ((c.getRadius() == value)) {

EventQueue.invokeLater(new Runnable() {

@Override

public void run() {

new Dissappear(ExpFrame.this.panel,c, button2);

}

});

}

continue;

}

if ((colorx.getRGB() == c.getColor().getRGB()) && (c.getRadius() == value)) {

EventQueue.invokeLater(new Runnable() {

@Override

public void run() {

new Dissappear(ExpFrame.this.panel,c, button2);

}

});

}

}

});

…



