**Национальный исследовательский университет ИТМО**

**Факультет Систем Управления и Робототехники**

Отчёт по лабораторной работе №5

По дисциплине «Программирование»

Студент: Павлова Дарья

Группа: R3136

Преподаватель:

Санкт-Петербург 2020

**Текст задания.**

Реализовать консольное приложение, которое реализует управление коллекцией объектов в интерактивном режиме. В коллекции необходимо хранить объекты класса MusicBand, описание которого приведено ниже.

**Разработанная программа должна удовлетворять следующим требованиям:**

* Класс, коллекцией экземпляров которого управляет программа, должен реализовывать сортировку по умолчанию.
* Все требования к полям класса (указанные в виде комментариев) должны быть выполнены.
* Для хранения необходимо использовать коллекцию типа java.util.ArrayList
* При запуске приложения коллекция должна автоматически заполняться значениями из файла.
* Имя файла должно передаваться программе с помощью: **аргумент командной строки**.
* Данные должны храниться в файле в формате csv
* Чтение данных из файла необходимо реализовать с помощью класса java.io.BufferedReader
* Запись данных в файл необходимо реализовать с помощью класса java.io.BufferedOutputStream
* Все классы в программе должны быть задокументированы в формате javadoc.
* Программа должна корректно работать с неправильными данными (ошибки пользовательского ввода, отсутсвие прав доступа к файлу и т.п.).

**В интерактивном режиме программа должна поддерживать выполнение следующих команд:**

* help : вывести справку по доступным командам
* info : вывести в стандартный поток вывода информацию о коллекции (тип, дата инициализации, количество элементов и т.д.)
* show : вывести в стандартный поток вывода все элементы коллекции в строковом представлении
* add {element} : добавить новый элемент в коллекцию
* update id {element} : обновить значение элемента коллекции, id которого равен заданному
* remove\_by\_id id : удалить элемент из коллекции по его id
* clear : очистить коллекцию
* save : сохранить коллекцию в файл
* execute\_script file\_name : считать и исполнить скрипт из указанного файла. В скрипте содержатся команды в таком же виде, в котором их вводит пользователь в интерактивном режиме.
* exit : завершить программу (без сохранения в файл)
* remove\_at index : удалить элемент, находящийся в заданной позиции коллекции (index)
* add\_if\_max {element} : добавить новый элемент в коллекцию, если его значение превышает значение наибольшего элемента этой коллекции
* sort : отсортировать коллекцию в естественном порядке
* count\_less\_than\_number\_of\_participants numberOfParticipants : вывести количество элементов, значение поля numberOfParticipants которых меньше заданного
* count\_greater\_than\_front\_man frontMan : вывести количество элементов, значение поля frontMan которых больше заданного
* filter\_by\_number\_of\_participants numberOfParticipants : вывести элементы, значение поля numberOfParticipants которых равно заданному

**Формат ввода команд:**

* Все аргументы команды, являющиеся стандартными типами данных (примитивные типы, классы-оболочки, String, классы для хранения дат), должны вводиться в той же строке, что и имя команды.
* Все составные типы данных (объекты классов, хранящиеся в коллекции) должны вводиться по одному полю в строку.
* При вводе составных типов данных пользователю должно показываться приглашение к вводу, содержащее имя поля (например, "Введите дату рождения:")
* Если поле является enum'ом, то вводится имя одной из его констант (при этом список констант должен быть предварительно выведен).
* При некорректном пользовательском вводе (введена строка, не являющаяся именем константы в enum'е; введена строка вместо числа; введённое число не входит в указанные границы и т.п.) должно быть показано сообщение об ошибке и предложено повторить ввод поля.
* Для ввода значений null использовать пустую строку.
* Поля с комментарием "Значение этого поля должно генерироваться автоматически" не должны вводиться пользователем вручную при добавлении.

**Описание хранимых в коллекции классов:**

public class MusicBand {

private Integer id; //Поле не может быть null, Значение поля должно быть больше 0, Значение этого поля должно быть уникальным, Значение этого поля должно генерироваться автоматически

private String name; //Поле не может быть null, Строка не может быть пустой

private Coordinates coordinates; //Поле не может быть null

private java.time.LocalDate creationDate; //Поле не может быть null, Значение этого поля должно генерироваться автоматически

private int numberOfParticipants; //Значение поля должно быть больше 0

private MusicGenre genre; //Поле не может быть null

private Person frontMan; //Поле может быть null

}

public class Coordinates {

private long x; //Значение поля должно быть больше -785

private Long y; //Значение поля должно быть больше -935, Поле не может быть null

}

public class Person {

private String name; //Поле не может быть null, Строка не может быть пустой

private java.time.LocalDate birthday; //Поле может быть null

private Location location; //Поле может быть null

}

public class Location {

private double x;

private float y;

private int z;

private String name; //Поле может быть null

}

public enum MusicGenre {

SOUL,

POP,

MATH\_ROCK,

POST\_PUNK;

}

**Диаграмма классов разработанной программы.**

Изображение выглядит как монитор, внутренний, компьютер, черный

Автоматически созданное описание

**Исходный код программы.**

*/\*\*  
 \* класс для исполнения программы  
 \*/*public class Main {  
 */\*\*  
 \* метод для исполнения программы  
 \** ***@param*** *args входные значения  
 \*/* public static void main (String[] args) {  
  
 if (args.length < 1) {  
 System.*out*.println("отсутствует обязательный аргумент командной строки");  
 System.*out*.println("имя файла для открытия");  
 System.*exit*(-1);  
 }  
  
 Driver d = new Driver(args[0]);  
 d.run();  
 }  
}

import javafx.util.Pair;  
import java.io.\*;  
import java.util.Scanner;  
  
*/\*\*  
 \*класс для управления коллекцией  
 \*/*public class Driver {  
 */\*\*  
 \*поле комманды для управления коллекцией  
 \*/* private BandList mbList;  
 */\*\*  
 \* поле имя файла для записи введеных команд  
 \*/* private String commandLogFileName;  
 */\*\*  
 \* поле запись введеных комманд  
 \*/* private boolean writeCommandLog;  
 */\*\*  
 \* поле имя файла  
 \*/* private String fileName;  
  
  
 /\*  
 public Driver() {  
  
 IoHelper.in = new Scanner(System.in);  
 fileName = null;  
 mbList = new BandList();  
  
 commandLogFileName = "log.txt";  
 writeCommandLog = true;  
 }  
 \*/  
  
 */\*\*  
 \* метод для считывания файла  
 \** ***@param*** *fileName имя файла  
 \*/* public Driver(String fileName) {  
 IoHelper.*in* = new Scanner(System.*in*);  
 try {  
 mbList = BandList.*loadFile*(fileName);  
 } catch (FileNotFoundException e) {  
 System.*out*.println("указанного файла не существует, создаём пустой BandList");  
 mbList = new BandList();  
 } catch (IOException e) {  
 System.*out*.println("IOException при попытке открытия файла");  
 System.*out*.println("выберите другой файл, программа завершает работу");  
 e.printStackTrace();  
 System.*exit*(-2);  
 }  
  
 this.fileName = fileName;  
 commandLogFileName = "log.txt";  
 writeCommandLog = true;  
 }  
  
 */\*\*  
 \* метод для вывода справки по доступным командам  
 \*/* public void help() {  
 System.*out*.println("help : вывести справку по доступным командам");  
 System.*out*.println("info : вывести в стандартный поток вывода информацию о коллекции (тип, дата инициализации, количество элементов и т.д.)");  
 System.*out*.println("show : вывести в стандартный поток вывода все элементы коллекции в строковом представлении");  
 System.*out*.println("add {element} : добавить новый элемент в коллекцию");  
 System.*out*.println("update id {element} : обновить значение элемента коллекции, id которого равен заданному");  
 System.*out*.println("remove\_by\_id id : удалить элемент из коллекции по его id");  
 System.*out*.println("clear : очистить коллекцию");  
 System.*out*.println("save : сохранить коллекцию в файл");  
 System.*out*.println("execute\_script file\_name : считать и исполнить скрипт из указанного файла. В скрипте содержатся команды в таком же виде, в котором их вводит пользователь в интерактивном режиме.");  
 System.*out*.println("exit : завершить программу (без сохранения в файл)");  
 System.*out*.println("remove\_at index : удалить элемент, находящийся в заданной позиции коллекции (index)");  
 System.*out*.println("add\_if\_max {element} : добавить новый элемент в коллекцию, если его значение превышает значение наибольшего элемента этой коллекции");  
 System.*out*.println("sort : отсортировать коллекцию в естественном порядке");  
 System.*out*.println("count\_less\_than\_number\_of\_participants numberOfParticipants : вывести количество элементов, значение поля numberOfParticipants которых меньше заданного");  
 System.*out*.println("count\_greater\_than\_front\_man frontMan : вывести количество элементов, значение поля frontMan которых больше заданного");  
 System.*out*.println("filter\_by\_number\_of\_participants numberOfParticipants : вывести элементы, значение поля numberOfParticipants которых равно заданному");  
 }  
  
 */\*\*  
 \* метод для записи введеных команд  
 \** ***@param*** *command введеная команда  
 \*/* public void writeLog(String command) {  
 FileWriter fileWriter = null;  
 try {  
 fileWriter = new FileWriter(commandLogFileName, true);  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 PrintWriter printWriter = new PrintWriter(fileWriter);  
 printWriter.println(command);  
 printWriter.close();  
 }  
  
 */\*\*  
 \*метод для считывания и выполнения скрипта из указанного файла  
 \** ***@param*** *command введеная команда  
 \** ***@return*** *false, если надо выходить (exit)  
 \*/* public boolean executeScript(String[] command) {  
 if (command.length < 2){  
 System.*out*.println("syntax: executeScript <filename>");  
 return true;  
 }  
  
 String fileName = command[1];  
  
 FileReader fr;  
  
 try {  
 fr = new FileReader(fileName);  
 } catch (FileNotFoundException e) {  
 System.*out*.println("не найден файл скрипта");  
 e.printStackTrace();  
 return true;  
 }  
 BufferedReader br = new BufferedReader(fr);  
 String s;  
  
 Scanner temp = IoHelper.*in*;  
 IoHelper.*in* = new Scanner(br);  
  
 try {  
 while (IoHelper.*in*.hasNextLine()) {  
 s = IoHelper.*in*.nextLine();  
 System.*out*.println(">>>" + s);  
 if (!execution(s)) {  
 IoHelper.*in* = temp;  
 return false;  
 }  
 }  
  
 br.close();  
 fr.close();  
 } catch (IOException ex) {  
 System.*out*.println("error reading from file");  
 IoHelper.*in* = temp;  
 return true;  
 }  
  
 IoHelper.*in* = temp;  
 return true;  
 }  
  
 */\*\*  
 \* метод, который удаляет элемент, находящийся в заданной позиции коллекции  
 \** ***@param*** *command введенная команда  
 \*/* public void removeAt(String[] command) {  
 if (command.length < 2){  
 System.*out*.println("syntax: remove\_by\_id <id>");  
 return;  
 }  
  
 int id;  
 try {  
 id = Integer.*parseInt*(command[1]);  
 } catch (NumberFormatException ex) {  
 System.*out*.println("wrong id parameter");  
 return;  
 }  
  
 mbList.removeAt(id);  
 }  
  
 */\*\*  
 \* метод для удаления элемента из коллекции по его индентификационному номеру  
 \** ***@param*** *command введенная команда  
 \*/* public void removeById(String[] command) {  
 if (command.length < 2){  
 System.*out*.println("syntax: remove\_by\_id <id>");  
 return;  
 }  
  
 int id;  
 try {  
 id = Integer.*parseInt*(command[1]);  
 } catch (NumberFormatException ex) {  
 System.*out*.println("wrong id parameter");  
 return;  
 }  
  
 mbList.removeById(id);  
 }  
  
 */\*\*  
 \* метод для считывания и выполнения команд  
 \** ***@param*** *inputCommand введенная комманда  
 \** ***@return*** *true, пока не будет вызвана команда exit  
 \*/* public boolean execution(String inputCommand) {  
 String[] command = inputCommand.split(" ", 2);  
  
 if (writeCommandLog)  
 writeLog(inputCommand);  
  
 switch (command[0]) {  
 case "help":  
 help();  
 break;  
 case "info":  
 mbList.info();  
 break;  
 case "show":  
 mbList.show();  
 break;  
 case "add":  
 add();  
 break;  
 case "update":  
 updateById(command);  
 break;  
 case "remove\_by\_id":  
 removeById(command);  
 break;  
 case "clear":  
 mbList.clear();  
 break;  
 case "save":  
 try {  
 mbList.save(fileName);  
 } catch (IOException e) {  
 System.*out*.println("exception при сохранении файла");  
 e.printStackTrace();  
 }  
 break;  
 case "execute\_script":  
 case "es":  
 if (!executeScript(command)) {  
 return false;  
 }  
 break;  
 case "exit":  
 return false;  
 case "remove\_at":  
 removeAt(command);  
 break;  
 case "add\_if\_max":  
 addIfMax();  
 break;  
 case "sort":  
 mbList.sort();  
 break;  
 case "count\_less\_than\_number\_of\_participants":  
 countLessThanNumberOfParticipants(command);  
 break;  
 case "count\_greater\_than\_front\_man":  
 countGreaterThanFrontMan();  
 break;  
 case "filter\_by\_number\_of\_participants":  
 filterByNumberOfParticipants(command);  
 break;  
 }  
 return true;  
 }  
  
 */\*\*  
 \* метод, котрый выводит количество элементов коллекции, поле frontMan которых больше заданного  
 \*/* private void countGreaterThanFrontMan() {  
 Pair<Boolean, Person> fmp = Person.*input*("front man");  
 if (!fmp.getKey() || fmp.getValue() == null) {  
 System.*out*.println("cancelled");  
 return;  
 }  
 Person frontMan = fmp.getValue();  
 mbList.countGreaterThanFrontMan(frontMan);  
 }  
  
 */\*\*  
 \*метод, котрый выводит элемент коллекции, количество участников которого равно заданному  
 \** ***@param*** *command введенная команда  
 \*/* public void filterByNumberOfParticipants(String[] command) {  
 if (command.length < 2){  
 System.*out*.println("syntax: update <id>");  
 return;  
 }  
  
 int nop;  
 try {  
 nop = Integer.*parseInt*(command[1]);  
 } catch (NumberFormatException ex) {  
 System.*out*.println("wrong id");  
 return;  
 }  
 mbList.filterByNumberOfParticipants(nop);  
 }  
  
 */\*\*  
 \*метод, котрый выводит количество элементов коллекции, количество участников которых меньше заданного  
 \** ***@param*** *command введенная команда  
 \*/* public void countLessThanNumberOfParticipants(String[] command) {  
 if (command.length < 2){  
 System.*out*.println("syntax: update <id>");  
 return;  
 }  
  
 int nop;  
 try {  
 nop = Integer.*parseInt*(command[1]);  
 } catch (NumberFormatException ex) {  
 System.*out*.println("wrong id");  
 return;  
 }  
 mbList.countLessThanNumberOfParticipants(nop);  
 }  
  
 */\*\*  
 \* метод для изменения элемента коллекции, индентификационный номер котрого равен заданному  
 \** ***@param*** *command введенная команда  
 \*/* public void updateById(String[] command) {  
  
 if (command.length < 2) {  
 System.*out*.println("syntax: update <id>");  
 return;  
 }  
  
 int id;  
 try {  
 id = Integer.*parseInt*(command[1]);  
 } catch (NumberFormatException ex) {  
 System.*out*.println("wrong id");  
 return;  
 }  
  
 MusicBand band = mbList.findById(id);  
  
 if (band == null) {  
 System.*out*.println("id does not exist");  
 System.*out*.println("update cancelled");  
 return;  
 }  
  
 System.*out*.println("found band with the requested id");  
 System.*out*.println("proceeding to update");  
 boolean updateRes = band.edit();  
  
 if (updateRes) {  
 System.*out*.println("updated successfully");  
 } else {  
 System.*out*.println("updated cancelled");  
 }  
 }  
  
 */\*\*  
 \* метод для добавления нового элемента  
 \*/* private void add() {  
 System.*out*.println("input band data to be added:");  
 Pair<Boolean, MusicBand> res = MusicBand.*input*();  
 if (res.getKey()) {  
 mbList.add(res.getValue());  
 System.*out*.println("band added successfully");  
 } else {  
 System.*out*.println("adding band has been cancelled");  
 }  
 }  
  
 */\*\*  
 \* метод, который добавляет новый элемент в коллекцию, если он больше наибольшего элемента этой коллекции  
 \*/* private void addIfMax() {  
 System.*out*.println("input band data to be added:");  
 Pair<Boolean, MusicBand> res = MusicBand.*input*();  
 if (res.getKey()) {  
 if (mbList.addIfMax(res.getValue()))  
 System.*out*.println("band added successfully");  
 else  
 System.*out*.println("band is less than the current max, adding cancelled");  
 } else {  
 System.*out*.println("adding band has been cancelled");  
 }  
 }  
  
 */\*\*  
 \* метод для считывания и выполнения введенных команд  
 \*/* public void run() {  
  
 boolean isRun = true;  
 String inputS;  
  
 writeLog("");  
  
 while(isRun) {  
 inputS = IoHelper.*in*.nextLine();  
 isRun = execution(inputS);  
 }  
  
 }  
  
}

import javafx.util.Pair;  
import java.time.LocalDate;  
import java.time.format.DateTimeFormatter;  
import java.time.format.DateTimeParseException;  
import java.util.Scanner;  
  
*/\*\*  
 \* класс для ввода команд  
 \*/*public class IoHelper {  
 */\*\*  
 \* поле Scanner для ввода с клавиатуры  
 \*/* public static Scanner *in*;  
  
 */\*\*  
 \* метод, который спрашивает о продолжении ввода данных  
 \** ***@param*** *message считываемая строка  
 \** ***@return*** *true, если пользователь хочет продолжать ввод, иначе false  
 \*/* public static boolean getConfirmation(String message) {  
 System.*out*.println(message + "(y/n):");  
 String line = *in*.nextLine();  
  
 while (true) {  
 if (line.equals("y"))  
 return true;  
 if (line.equals("n"))  
 return false;  
  
 System.*out*.println(message + "(y/n):");  
 line = *in*.nextLine();  
 }  
 }  
  
 */\*\*  
 \* метод, который проверяет правильность введеного числа типа long  
 \** ***@param*** *prompt считываемая строка  
 \** ***@return*** *результат проверки и число типа long, если результат проверки true; 0, если false  
 \*/* public static Pair<Boolean, Long> getLong(String prompt) {  
 System.*out*.println(prompt + ": ");  
 String line;  
  
 long res;  
  
 line = *in*.nextLine();  
 while (true) {  
 try {  
 res = Long.*parseLong*(line);  
 } catch (NumberFormatException ex) {  
 if (*getConfirmation*("try again")) {  
 System.*out*.println(prompt + ": ");  
 line = *in*.nextLine();  
 continue;  
 } else {  
 return new Pair<Boolean, Long>(false, 0L);  
 }  
 }  
  
 return new Pair<Boolean, Long>(true, res);  
  
 }  
 }  
  
 */\*\*  
 \* метод, который проверяет правильность введеного числа типа int  
 \** ***@param*** *prompt считываемая строка  
 \** ***@return*** *результат проверки и число типа int, если результат проверки true; 0, если false  
 \*/* public static Pair<Boolean, Integer> getInt(String prompt) {  
 System.*out*.println(prompt + ": ");  
 String line;  
  
 int res;  
  
 line = *in*.nextLine();  
 while (true) {  
 try {  
 res = Integer.*parseInt*(line);  
 } catch (NumberFormatException ex) {  
 if (*getConfirmation*("try again")) {  
 System.*out*.println(prompt + ": ");  
 line = *in*.nextLine();  
 continue;  
 } else {  
 return new Pair<Boolean, Integer>(false, 0);  
 }  
 }  
  
 return new Pair<Boolean, Integer>(true, res);  
  
 }  
 }  
  
 */\*\*  
 \* метод, который вызывает метод getDouble(prompt, false), который проверяет правильность введеного числа типа double  
 \** ***@param*** *prompt считываемая строка  
 \** ***@return*** *метод getDouble(prompt, false)  
 \*/* public static Pair<Boolean, Double> getDouble(String prompt) {  
 return *getDouble*(prompt, false);  
 }  
  
 */\*\*  
 \* метод, который проверяет правильность введеного числа типа double  
 \** ***@param*** *prompt считываемая строка  
 \** ***@param*** *allowNull true, если число может быть null, иначе false  
 \** ***@return*** *результат проверки и число типа double, если результат проверки true; 0, если false. Число будет null, если allowNull - true  
 \*/* public static Pair<Boolean, Double> getDouble(String prompt, Boolean allowNull) {  
 System.*out*.println(prompt + ": ");  
 String line;  
  
 double res;  
  
 line = *in*.nextLine();  
 if (allowNull && line.isEmpty())  
 return new Pair<>(true, null);  
  
 while (true) {  
 try {  
 res = Double.*parseDouble*(line);  
 } catch (NumberFormatException ex) {  
 if (*getConfirmation*("try again")) {  
 System.*out*.println(prompt + ": ");  
 line = *in*.nextLine();  
  
 if (allowNull && line.isEmpty())  
 return new Pair<>(true, null);  
  
 continue;  
 } else {  
 return new Pair<Boolean, Double>(false, 0d);  
 }  
 }  
  
 return new Pair<Boolean, Double>(true, res);  
  
 }  
 }  
  
 */\*\*  
 \* метод, который проверяет правильность введеного числа типа float  
 \** ***@param*** *prompt считываемая строка  
 \** ***@return*** *результат проверки и число типа float, если результат проверки true; 0, если false  
 \*/* public static Pair<Boolean, Float> getFloat(String prompt) {  
 System.*out*.println(prompt + ": ");  
 String line;  
  
 float res;  
  
 line = *in*.nextLine();  
 while (true) {  
 try {  
 res = Float.*parseFloat*(line);  
 } catch (NumberFormatException ex) {  
 if (*getConfirmation*("try again")) {  
 System.*out*.println(prompt + ": ");  
 line = *in*.nextLine();  
 continue;  
 } else {  
 return new Pair<Boolean, Float>(false, 0f);  
 }  
 }  
  
 return new Pair<Boolean, Float>(true, res);  
  
 }  
 }  
  
 */\*\*  
 \* метод, который проверяет правильность введеной даты  
 \** ***@param*** *prompt считываемая строка  
 \** ***@return*** *результат проверки и дату, если результат проверки true; null, если false  
 \*/* public static Pair<Boolean, LocalDate> getLocalDate(String prompt) {  
 System.*out*.println(prompt + ": ");  
  
 String line = *in*.nextLine();  
 while (true) {  
 if (line.isEmpty()) {  
 return new Pair<>(true, null);  
 }  
 LocalDate res;  
 try {  
 res = LocalDate.*parse*(line, DateTimeFormatter.*ofPattern*("dd.MM.yyyy"));  
 } catch (DateTimeParseException ex) {  
 if (*getConfirmation*("try again")) {  
 System.*out*.println(prompt + ": ");  
 line = *in*.nextLine();  
 if (line.isEmpty()) {  
 return new Pair<>(true, null);  
 }  
 continue;  
 } else {  
 return new Pair<>(false, null);  
 }  
 }  
  
 return new Pair<>(true, res);  
 }  
 }  
}

import java.io.\*;  
import java.time.LocalDate;  
import java.time.format.DateTimeFormatter;  
import java.util.ArrayList;  
import java.util.Date;  
  
*/\*\*  
 \* класс, описывающий команды для управления кколлекцией  
 \*/*public class BandList {  
 */\*\*  
 \* поле элемент коллекции  
 \*/* private ArrayList<MusicBand> musicBandList;  
 */\*\*  
 \* поле сгенерированный индентификационный номер  
 \*/* private int generatedId;  
 */\*\*  
 \* поле дата инициализации  
 \*/* private Date initDate;  
  
 */\*\*  
 \* конструктор, который создает объект коллекции с индентификационным номером 1 и дату инициализации  
 \*/* public BandList () {  
 musicBandList = new ArrayList<MusicBand>();  
 generatedId = 1;  
 initDate = new Date();  
 }  
  
 */\*\*  
 \* метод, который находит Music Band с введеным индентификационным номером  
 \** ***@param*** *id индентификационный номер  
 \** ***@return*** *Music Band с введеным индентификационным номером или null  
 \*/* public MusicBand findById(int id) {  
 MusicBand found = null;  
 for (MusicBand b : musicBandList) {  
 if (b.getId() == id) {  
 return b;  
 }  
 }  
 return null;  
 }  
  
 */\*\*  
 \* метод для считывания файла и создания элемента коллекции  
 \** ***@param*** *fileName имя файла  
 \** ***@return*** *элемет коллекции  
 \** ***@throws*** *IOException проверяет, не прервалась ли операция ввода/ввывода  
 \*/* public static BandList loadFile(String fileName) throws IOException {  
  
 BandList res = new BandList();  
  
 FileReader fr = new FileReader(fileName);  
 BufferedReader br = new BufferedReader(fr);  
 String s;  
  
 while ((s = br.readLine()) != null) {  
 String[] fields = s.split(";");  
  
 if (fields.length < 7) {  
 System.*out*.println("в строке csv");  
 System.*out*.println(s);  
 System.*out*.println("меньше 7 обязательных полей");  
 System.*out*.println("пропускаем эту банду");  
 continue;  
 }  
  
 try {  
 int id = Integer.*parseInt*(fields[0]);  
 String name = fields[1];  
 long x = Long.*parseLong*(fields[2]);  
 Long y = Long.*parseLong*(fields[3]);  
 LocalDate date = LocalDate.*parse*(fields[4], DateTimeFormatter.*ofPattern*("dd.MM.yyyy"));  
 int numberOfParticipants = Integer.*parseInt*(fields[5]);  
 MusicGenre genre = MusicGenre.*getMusicGenre*(fields[6]);  
  
 Person fm = null;  
 if (fields.length > 7) {  
 String personName = fields[7];  
 LocalDate personBday = null;  
 Location loc = null;  
 if (fields.length > 8) {  
 personBday = LocalDate.*parse*(fields[8], DateTimeFormatter.*ofPattern*("dd.MM.yyyy"));  
 if (fields.length > 9) {  
 if (fields.length < 12) {  
 System.*out*.println("в строке csv");  
 System.*out*.println(s);  
 System.*out*.println("задано frontman-location, но меньше 3 обязательных полей");  
 System.*out*.println("пропускаем эту банду");  
 continue;  
 }  
 double locX = Double.*parseDouble*(fields[9]);  
 float locY = Float.*parseFloat*(fields[10]);  
 int locZ = Integer.*parseInt*(fields[11]);  
 String locName = null;  
 if (fields.length == 13) {  
 locName = fields[12];  
 }  
 loc = new Location(locX, locY, locZ, locName);  
 }  
 }  
 fm = new Person(personName, personBday, loc);  
 }  
 res.add(new MusicBand(name, new Coordinates(x, y),  
 date, numberOfParticipants, genre, fm), id);  
 } catch (NumberFormatException ex) {  
 System.*out*.println("в строке csv");  
 System.*out*.println(s);  
 System.*out*.println("не получилось распарсить поле");  
 System.*out*.println("пропускаем эту банду");  
 System.*out*.println(ex);  
 continue;  
 }  
 }  
  
 br.close();  
 fr.close();  
  
 return res;  
 }  
  
 */\*\*  
 \* метод для вывода в стандартный поток вывода информации о коллекции  
 \*/* public void info() {  
 System.*out*.printf("Тип: MusicBand\nДата инициализации: %s\nКоличество элементов: %d\n", initDate.toString(), musicBandList.size());  
 }  
  
 */\*\*  
 \* метод для вывода в стандартный поток всех элементов коллекции  
 \*/* public void show() {  
 for (MusicBand m: musicBandList) {  
 System.*out*.println(m);  
 }  
 }  
  
 */\*\*  
 \* метод для добавления нового элемента с генерацией индентификационного номера  
 \** ***@param*** *band добавляемый элемент  
 \*/* public void add(MusicBand band) {  
 band.setId(this.generatedId++);  
 musicBandList.add(band);  
 }  
  
 */\*\*  
 \* метод для добавления нового элемента  
 \** ***@param*** *band добавляемый элемент  
 \** ***@param*** *id индентификационный номер  
 \*/* public void add(MusicBand band, int id) {  
  
 MusicBand found = findById(id);  
 if (found != null) {  
 System.*out*.println(band.getName());  
 System.*out*.println("id банды конфликтует с существующей в списке бандой");  
 System.*out*.println(String.*format*("%s id = %d", found.getName(), found.getId()));  
 System.*out*.println("назначаем автоматически");  
 int newId = this.generatedId++;  
 System.*out*.println(newId);  
 band.setId(newId);  
 } else {  
 band.setId(id);  
 generatedId = Math.*max*(generatedId, band.getId() + 1);  
 }  
  
 musicBandList.add(band);  
 }  
  
 */\*\*  
 \* метод для изменения элемента коллекции, индентификационный номер котрого равен заданному  
 \** ***@param*** *id индентификационный номер  
 \*/* public void update(int id) {  
 MusicBand found = null;  
 for (MusicBand b : musicBandList) {  
 if (b.getId() == id) {  
 found = b;  
 break;  
 }  
 }  
  
 if (found == null) {  
 System.*out*.println("id does not exist");  
 return;  
 }  
 }  
  
 */\*\*  
 \* метод для удаления элемента из коллекции по его индентификационному номеру  
 \** ***@param*** *id индентификационный номер  
 \*/* public void removeById(int id) {  
 musicBandList.removeIf(m -> m.getId() == id);  
 }  
  
 */\*\*  
 \* метод для очистки коллекции  
 \*/* public void clear(){  
 musicBandList.clear();  
 }  
  
 */\*\*  
 \* метод для сохранения коллекции в файл  
 \** ***@param*** *fileName имя файла  
 \** ***@throws*** *IOException проверяет, не прервалась ли операция ввода/вывода  
 \*/* public void save(String fileName) throws IOException {  
 FileOutputStream fos = new FileOutputStream(fileName);  
 BufferedOutputStream bos = new BufferedOutputStream(fos);  
 for (MusicBand m: musicBandList) {  
 bos.write(m.toCSVString().getBytes());  
 bos.write("\n".getBytes());  
 }  
 bos.close();  
 fos.close();  
 }  
  
 */\*\*  
 \* метод, который удаляет элемент, находящийся в заданной позиции коллекции  
 \** ***@param*** *index позиция коллекции  
 \*/* public void removeAt(int index) {  
 if (index < musicBandList.size())  
 musicBandList.remove(index);  
 else {  
 System.*out*.println("no such band");  
 }  
 }  
  
 */\*\*  
 \* метод для нахождения максимального элемента коллекции  
 \** ***@return*** *максимальный элемент коллекции  
 \*/* public MusicBand findMax() {  
 if (musicBandList.size() == 0)  
 return null;  
 MusicBand max = musicBandList.get(0);  
 for (int i = 1; i != musicBandList.size(); ++i)  
 if (musicBandList.get(i).compareTo(max) > 0)  
 max = musicBandList.get(i);  
  
 return max;  
 }  
  
 */\*\*  
 \* метод, который добавляет новый элемент в коллекцию, если он больше наибольшего элемента этой коллекции  
 \** ***@param*** *b элемент коллекции  
 \** ***@return*** *true, если элемент добавлен, иначе false  
 \*/* public boolean addIfMax(MusicBand b) {  
 if (b.compareTo(findMax()) > 0) {  
 add(b);  
 return true;  
 } else  
 return false;  
 }  
  
 */\*\*  
 \* метод для сортировки элементов коллекции  
 \*/* public void sort() {  
 musicBandList.sort(MusicBand::compareTo);  
 }  
  
 */\*\*  
 \*метод, котрый выводит количество элементов коллекции, количество участников которых меньше заданного  
 \** ***@param*** *numberOfParticipants количество участников в музыкальной банде  
 \*/* public void countLessThanNumberOfParticipants(int numberOfParticipants) {  
 System.*out*.println(musicBandList.stream()  
 .mapToInt(MusicBand::getNumberOfParticipants)  
 .filter(x -> x < numberOfParticipants)  
 .count());  
 }  
  
 */\*\*  
 \* метод, котрый выводит количество элементов коллекции, поле frontMan которых больше заданного  
 \** ***@param*** *frontMan человек, состоящий в музыкальной банде  
 \*/* public void countGreaterThanFrontMan(Person frontMan) {  
 System.*out*.println(musicBandList.stream()  
 .filter(x -> x.getFrontMan() != null)  
 .map(MusicBand::getFrontMan)  
 .filter(x -> x.compareTo(frontMan) < 0)  
 .count());  
 }  
  
 */\*\*  
 \*метод, котрый выводит элемент коллекции, количество участников которого равно заданному  
 \** ***@param*** *numberOfParticipants количество участников в музыкальной банде  
 \*/* public void filterByNumberOfParticipants(int numberOfParticipants) {  
 for (MusicBand m: musicBandList) {  
 if (m.getNumberOfParticipants() == numberOfParticipants) {  
 System.*out*.println(m);  
 }  
 }  
 }  
}

import javafx.util.Pair;  
import java.time.LocalDate;  
import java.time.format.DateTimeFormatter;  
  
*/\*\*  
 \* классб описывающий Music Band  
 \*/*public class MusicBand implements Comparable{  
 */\*\*  
 \* поле индентификационный номер музыкальной банды  
 \*/* private Integer id; //Поле не может быть null, Значение поля должно быть больше 0, Значение этого поля должно быть уникальным, Значение этого поля должно генерироваться автоматически  
 */\*\*  
 \* поле название музыкальной банды  
 \*/* private String name; //Поле не может быть null, Строка не может быть пустой  
 */\*\*  
 \* поле координаты музыкальной банды  
 \*/* private Coordinates coordinates; //Поле не может быть null  
 */\*\*  
 \* поле дата основания музыкальной банды  
 \*/* private java.time.LocalDate creationDate; //Поле не может быть null, Значение этого поля должно генерироваться автоматически  
 */\*\*  
 \* поле количество участников музыкальной банды  
 \*/* private int numberOfParticipants; //Значение поля должно быть больше 0  
 */\*\*  
 \* поле жанр, который преподчитает музыкальная банда  
 \*/* private MusicGenre genre; //Поле не может быть null  
 */\*\*  
 \* поле человек, который состоит в музыкальной банде  
 \*/* private Person frontMan; //Поле может быть null  
  
 */\*\*  
 \* конструктор, позволяющий задать имя, координаты, дату основания, число участников, жанр и человека  
 \** ***@param*** *name имя Music Band  
 \** ***@param*** *coordinates координаты Music Band  
 \** ***@param*** *creationDate дата основания Music Band  
 \** ***@param*** *numberOfParticipants количество участников Music Band  
 \** ***@param*** *genre жанр, который преподчитает Music Band  
 \** ***@param*** *frontMan человек, состоящий в Music Band  
 \*/* public MusicBand (String name, Coordinates coordinates,  
 java.time.LocalDate creationDate, int numberOfParticipants,  
 MusicGenre genre, Person frontMan) {  
 this.id = 0;  
 this.name = name;  
 this.coordinates = coordinates;  
 this.creationDate = creationDate;  
 this.numberOfParticipants = numberOfParticipants;  
 this.genre = genre;  
 this.frontMan = frontMan;  
 }  
  
 */\*\*  
 \* метод, который проверяет правильность введенных полей у Music Band  
 \** ***@return*** *результат проверки и Music Band, если результат проверки true; null, если false  
 \*/* public static Pair<Boolean, MusicBand> input() {  
 String name;  
 Coordinates coordinates;  
 java.time.LocalDate creationDate;  
 int numberOfParticipants;  
 MusicGenre genre;  
 Person frontMan;  
  
 String line;  
  
 // name (not null)  
 System.*out*.print("name: ");  
 line = IoHelper.*in*.nextLine();  
 while (line.isEmpty()) {  
 System.*out*.println("name not null");  
 System.*out*.print("name: ");  
 line = IoHelper.*in*.nextLine();  
 }  
  
 name = line;  
  
 System.*out*.println("coordinates: ");  
 Pair<Boolean, Coordinates> cp = Coordinates.*input*();  
 if (!cp.getKey()) {  
 return new Pair<>(false, null);  
 }  
  
 coordinates = cp.getValue();  
  
 creationDate = LocalDate.*now*();  
  
 Pair<Boolean, Integer> nopp = IoHelper.*getInt*("number of participants");  
 if (!nopp.getKey()) {  
 return new Pair<>(false, null);  
 }  
  
 while (nopp.getValue() <= 0) {  
 nopp = IoHelper.*getInt*("number of participants");  
 if (!nopp.getKey()) {  
 return new Pair<>(false, null);  
 }  
 }  
  
 numberOfParticipants = nopp.getValue();  
  
 Pair<Boolean, MusicGenre> mgp = MusicGenre.*input*("genre");  
 if (!mgp.getKey()) {  
 return new Pair<>(false, null);  
 }  
 genre = mgp.getValue();  
  
 Pair<Boolean, Person> fmp = Person.*input*("front man");  
 if (!fmp.getKey()) {  
 return new Pair<>(false, null);  
 }  
 frontMan = fmp.getValue();  
  
  
 return new Pair<Boolean, MusicBand>(true, new MusicBand(name, coordinates, creationDate,  
 numberOfParticipants, genre, frontMan));  
 }  
  
 */\*\*  
 \* метод для редактирования Music Band  
 \** ***@return*** *true, если Music Band был отредактирован  
 \*/* public boolean edit() {  
 String name;  
 Coordinates coordinates;  
 java.time.LocalDate creationDate;  
 int numberOfParticipants;  
 MusicGenre genre;  
 Person frontMan;  
  
 boolean confirm;  
 String line;  
  
  
 // name (not null)  
 System.*out*.println("name: ");  
 System.*out*.println(String.*format*("%s", this.name));  
 confirm = IoHelper.*getConfirmation*("update?");  
 if (confirm) {  
 line = IoHelper.*in*.nextLine();  
 while (line.isEmpty()) {  
 System.*out*.println("name not null");  
 System.*out*.print("name: ");  
 line = IoHelper.*in*.nextLine();  
 }  
  
 name = line;  
 } else {  
 name = this.name;  
 }  
  
 System.*out*.println("coordinates: ");  
 System.*out*.println(String.*format*("%s", this.coordinates.toString()));  
 confirm = IoHelper.*getConfirmation*("update?");  
  
 if (confirm) {  
 Pair<Boolean, Coordinates> cp = Coordinates.*input*();  
 if (!cp.getKey()) {  
 return false;  
 }  
  
 coordinates = cp.getValue();  
 } else {  
 coordinates = this.coordinates;  
 }  
  
 creationDate = LocalDate.*now*();  
  
 System.*out*.println("numberOfParticipants: ");  
 System.*out*.println(String.*format*("%d", this.numberOfParticipants));  
 confirm = IoHelper.*getConfirmation*("update?");  
  
 if (confirm) {  
 Pair<Boolean, Integer> nopp = IoHelper.*getInt*("number of participants");  
 if (!nopp.getKey()) {  
 return false;  
 }  
  
 while (nopp.getValue() <= 0) {  
 nopp = IoHelper.*getInt*("number of participants");  
 if (!nopp.getKey()) {  
 return false;  
 }  
 }  
  
 numberOfParticipants = nopp.getValue();  
 } else {  
 numberOfParticipants = this.numberOfParticipants;  
 }  
  
  
  
 System.*out*.println("genre: ");  
 System.*out*.println(String.*format*("%s", this.genre));  
 confirm = IoHelper.*getConfirmation*("update?");  
  
 if (confirm) {  
 Pair<Boolean, MusicGenre> mgp = MusicGenre.*input*("genre");  
 if (!mgp.getKey()) {  
 return false;  
 }  
 genre = mgp.getValue();  
 } else {  
 genre = this.genre;  
 }  
  
 System.*out*.println("front man: ");  
 System.*out*.println(String.*format*("%s", this.frontMan));  
 confirm = IoHelper.*getConfirmation*("update?");  
  
 if (confirm) {  
 Pair<Boolean, Person> fmp = Person.*input*("front man");  
 if (!fmp.getKey()) {  
 return false;  
 }  
 frontMan = fmp.getValue();  
 } else {  
 frontMan = this.frontMan;  
 }  
  
  
 this.name = name;  
 this.coordinates = coordinates;  
 this.creationDate = creationDate;  
 this.numberOfParticipants = numberOfParticipants;  
 this.genre = genre;  
 this.frontMan = frontMan;  
  
 return true;  
  
 }  
  
 */\*\*  
 \* метод, для получения значения поля индентификационный номер музыкальной банды  
 \** ***@param*** *id индентификационный номер музыкальной банды  
 \*/* public void setId(int id) {this.id = id;}  
 public Integer getId () {  
 return id;  
 }  
  
 */\*\*  
 \* метод, для получения значения поля имя Music Band  
 \** ***@param*** *name имя Music Band  
 \*/* public void setName (String name) {  
 this.name = name;  
 }  
  
 */\*\*  
 \* метод для задания значения поля имя Music Band  
 \** ***@return*** *имя Music Band  
 \*/* public String getName () {  
 return name;  
 }  
  
 */\*\*  
 \* метод для получения значения поля координаты Music Band  
 \** ***@param*** *coordinates координаты Music Band  
 \*/* public void setCoordinates (Coordinates coordinates) {  
 this.coordinates = coordinates;  
 }  
  
 */\*\*  
 \* метод для задания значения поля координаты Music Band  
 \** ***@return*** *координаты Music Band  
 \*/* public Coordinates getCoordinates () {  
 return coordinates;  
 }  
  
 */\*\*  
 \* метод для получения значения поля дата основания Music Band  
 \** ***@param*** *date дата основания Music Band  
 \*/* public void setDate (LocalDate date) {  
 this.creationDate = date;  
 }  
  
 */\*\*  
 \* метод для задания значения поля дата основания Music Band  
 \** ***@return*** *дата основания Music Band  
 \*/* public LocalDate getDate () {  
 return creationDate;  
 }  
  
 */\*\*  
 \* метод для получения значения поля количество участников музыкальной банды  
 \** ***@param*** *numberOfParticipants количество участников музыкальной банды  
 \*/* public void setNumberOfParticipants (int numberOfParticipants) {  
 this.numberOfParticipants = numberOfParticipants;  
 }  
  
 */\*\*  
 \* метод для задания значения поля количество участников музыкальной банды  
 \** ***@return*** *количество участников музыкальной банды  
 \*/* public int getNumberOfParticipants () {  
 return numberOfParticipants;  
 }  
  
 */\*\*  
 \* метод для получения значения поля жанр, который преподчитает музыкальная банда  
 \** ***@param*** *genre жанр, который преподчитает музыкальная банда  
 \*/* public void setGenre (MusicGenre genre) {  
 this.genre = genre;  
 }  
  
 */\*\*  
 \* метод для задания значения поля жанр, который преподчитает музыкальная банда  
 \** ***@return*** *жанр, который преподчитает музыкальная банда  
 \*/* public MusicGenre getGenre () {  
 return genre;  
 }  
  
 */\*\*  
 \* метод для получения значения поля человек, который состоит в музыкальной банде  
 \** ***@param*** *frontMan человек, который состоит в музыкальной банде  
 \*/* public void setFrontMan (Person frontMan) {  
 this.frontMan = frontMan;  
 }  
  
 */\*\*  
 \* метод для задания значения поля человек, который состоит в музыкальной банде  
 \** ***@return*** *человек, который состоит в музыкальной банде  
 \*/* public Person getFrontMan () {  
 return frontMan;  
 }  
  
 */\*\*  
 \* метод для получения строки с именем, координатами, датой основания, числом участников, жанром и человеком музыкальной банды в формате csv  
 \** ***@return*** *строка с именем, координатами, датой основания, числом участников, жанром и человеком музыкальной банды в формате csv  
 \*/* public String toCSVString(){  
  
 String fmCsv = "";  
 if (frontMan != null)  
 fmCsv = frontMan.toCSVString();  
 return String.*format*("%d;%s;%s;%s;%d;%s;%s", id, name, coordinates.toCSVString(),  
 creationDate.format(DateTimeFormatter.*ofPattern*("dd.MM.yyyy")),  
 numberOfParticipants, genre, fmCsv);  
 }  
 */\*\*  
 \* метод для получения строки с именем, координатами, датой основания, числом участников, жанром и человеком музыкальной банды  
 \** ***@return*** *строка с именем, координатами, датой основания, числом участников, жанром и человеком музыкальной банды  
 \*/* @Override  
 public String toString() {  
 return String.*format*("id: %d\nname: %s\ncoordinates: %s\ncreationDate: %s\nnumberOfParticipants: %d\ngenre: %s\nfrontMan: \n%s",  
 id, name, coordinates.toString(), creationDate.format(DateTimeFormatter.*ofPattern*("dd.MM.yyyy")), numberOfParticipants, genre, frontMan);  
 }  
 */\*\*  
 \* метод для сравнения Music Band  
 \** ***@param*** *o Music Band  
 \** ***@return*** *положительное число, если вызывающий объект больше объекта, переданного в качестве параметра; отрицательное число, если вызывающий объект меньше объекта, переданного в качестве параметра; нуль, если объекты равны  
 \*/* @Override  
 public int compareTo(Object o) {  
 return this.name.compareTo(((MusicBand) o).name);  
 }  
}

import javafx.util.Pair;  
import java.util.Locale;  
  
*/\*\*  
 \* класс, описывающий координаты Music Band  
 \*/*public class Coordinates {  
 */\*\*  
 \* поле координата x Music Band.  
 \*/* private long x; //Значение поля должно быть больше -785  
 */\*\*  
 \* поле координата y Music Band.  
 \*/* private Long y; //Значение поля должно быть больше -935, Поле не может быть null  
 */\*\*  
 \* поле минимальное значение x: -785  
 \*/* private static final long *MINX* = -785;  
 */\*\*  
 \* поле минимальное значение y: -935  
 \*/* private static final long *MINY* = -935;  
  
 */\*\*  
 \* конструктор, позволяющий задать координату x и координату y Music Band  
 \** ***@param*** *x координата x  
 \** ***@param*** *y координата y  
 \*/* public Coordinates (long x, Long y) {  
 this.x = x;  
 this.y = y;  
 }  
  
 */\*\*  
 \* метод для получения строки с координатами Music Band в формате csv  
 \** ***@return*** *строка с координатами Music Band в формате csv  
 \*/* public String toCSVString() {  
 return String.*format*(Locale.*US*, "%d;%d", x, y);  
 }  
 */\*\*  
 \* метод для получения строки с координатами Music Band в формате (x, y)  
 \** ***@return*** *строка с координатами Music Band в формате (x, y)  
 \*/* @Override  
 public String toString() {  
 return String.*format*(Locale.*US*, "(%d, %d)", x, y);  
 }  
  
 */\*\*  
 \* метод, который проверяет правильность введенных координат  
 \** ***@return*** *результат проверки и координаты, если результат проверки true; null, если false  
 \*/* public static Pair<Boolean, Coordinates> input() {  
 long x, y;  
  
 // x (not null)  
  
 Pair<Boolean, Long> res = IoHelper.*getLong*("x");  
 if (!res.getKey()) {  
 System.*out*.println("input cancelled");  
 return new Pair<Boolean, Coordinates>(false, null);  
 }  
  
 while (res.getValue() <= *MINX*) {  
 System.*out*.println(String.*format*("error: x must be greater than %d", *MINX*));  
 res = IoHelper.*getLong*("x");  
 if (!res.getKey()) {  
 System.*out*.println("input cancelled");  
 return new Pair<Boolean, Coordinates>(false, null);  
 }  
 }  
  
 x = res.getValue();  
  
 // y (not null)  
  
 res = IoHelper.*getLong*("y");  
 if (!res.getKey()) {  
 System.*out*.println("input cancelled");  
 return new Pair<Boolean, Coordinates>(false, null);  
 }  
  
 while (res.getValue() <= *MINY*) {  
 System.*out*.println(String.*format*("error: y must be greater than %d", *MINY*));  
 res = IoHelper.*getLong*("y");  
 if (!res.getKey()) {  
 System.*out*.println("input cancelled");  
 return new Pair<Boolean, Coordinates>(false, null);  
 }  
 }  
  
 y = res.getValue();  
  
 return new Pair<Boolean, Coordinates>(true,  
 new Coordinates(x, y));  
 }  
}

import javafx.util.Pair;  
  
*/\*\*  
 \* Набор музыкальных жанров, которые могут преподчитать музыкальные банды  
 \*/*public enum MusicGenre {  
 */\*\*  
 \* контанта soul  
 \*/  
 SOUL*,  
 */\*\*  
 \* контанта pop  
 \*/  
 POP*,  
 */\*\*  
 \* контанта math rock  
 \*/  
 MATH\_ROCK*,  
 */\*\*  
 \* контанта post punk  
 \*/  
 POST\_PUNK*;  
  
 */\*\*  
 \* метод для получения поля жанр, который преподчитает музыкальная группа  
 \** ***@param*** *s считываемая строка  
 \** ***@return*** *музыкальный жанр из набора музыкальных жанров или null  
 \*/* public static MusicGenre getMusicGenre(String s) {  
 switch (s) {  
 case "SOUL":  
 return MusicGenre.*SOUL*;  
 case "POP":  
 return MusicGenre.*POP*;  
 case "MATH\_ROCK":  
 return MusicGenre.*MATH\_ROCK*;  
 case "POST\_PUNK":  
 return MusicGenre.*POST\_PUNK*;  
 }  
 return null;  
 }  
  
 */\*\*  
 \* метод, который проверяет правильность введенного музыкального жанра  
 \** ***@param*** *prompt считываемая строка  
 \** ***@return*** *true и музыкальный жанр, если введенная строка является музыкальным жанром  
 \*/* public static Pair<Boolean, MusicGenre> input(String prompt) {  
  
 System.*out*.println(prompt + ": ");  
 for (MusicGenre mg: MusicGenre.*values*()) {  
 System.*out*.println(mg);  
 }  
  
 String line;  
 line = IoHelper.*in*.nextLine();  
  
 for (MusicGenre mg: MusicGenre.*values*()) {  
 if (line.equals(mg.toString()))  
 return new Pair<>(true, mg);  
 }  
  
 while (true) {  
  
 System.*out*.println(prompt + ": ");  
 for (MusicGenre mg: MusicGenre.*values*()) {  
 System.*out*.println(mg);  
 }  
  
 line = IoHelper.*in*.nextLine();  
  
 for (MusicGenre mg: MusicGenre.*values*()) {  
 if (line.equals(mg.toString()))  
 return new Pair<>(true, mg);  
 }  
 }  
 }  
}

import javafx.util.Pair;  
import java.time.LocalDate;  
import java.time.format.DateTimeFormatter;  
import java.util.Locale;  
  
*/\*\*  
 \* класс, описывающий человека из музыкальной банды  
 \*/*public class Person implements Comparable {  
 */\*\*  
 \* поле имя человека  
 \*/* private String name; //Поле не может быть null, Строка не может быть пустой  
 */\*\*  
 \* поле дата рождения человека  
 \*/* private java.time.LocalDate birthday; //Поле может быть null  
 */\*\*  
 \* поле месторасположение человека  
 \*/* private Location location; //Поле может быть null  
  
 */\*\*  
 \* конструктор, позволяющий задать имя, дату рождения и местоположение человека из музыкальной банды  
 \** ***@param*** *name имя человека  
 \** ***@param*** *birthday дата рождения человека  
 \** ***@param*** *location месторасположение человека  
 \*/* public Person (String name, java.time.LocalDate birthday, Location location){  
 this.name = name;  
 this.birthday = birthday;  
 this.location = location;  
 }  
 */\*\*  
 \* метод для получения строки с именем, датой рождения и местоположением человека из Music Band в формате csv  
 \** ***@return*** *строка с именем, датой рождения и местоположением человека из Music Band в формате csv  
 \*/* public String toCSVString() {  
 String bs = "", ls = "";  
 if (birthday != null)  
 bs = birthday.format(DateTimeFormatter.*ofPattern*("dd.MM.yyyy"));  
 if (location != null)  
 ls = location.toCSVString();  
 return String.*format*(Locale.*US*,"%s;%s;%s", name,  
 bs, ls);  
 }  
 */\*\*  
 \* метод для получения строки с именем, датой рождения и местоположением человека из Music Band  
 \** ***@return*** *строка с именем, датой рождения и местоположением человека из Music Band  
 \*/* @Override  
 public String toString() {  
 String result = String.*format*(Locale.*US*, "name: %s\n", name);  
 if (birthday != null)  
 result = result + String.*format*(Locale.*US*, "birthday: %s\n", birthday.format(DateTimeFormatter.*ofPattern*("dd.MM.yyyy")));  
 if (location != null)  
 result = result + String.*format*(Locale.*US*, "location: %s\n", location.toString());  
 return result;  
 }  
  
 */\*\*  
 \* метод для сравнения человека из Music Band  
 \** ***@param*** *o человек из Music Band  
 \** ***@return*** *положительное число, если вызывающий объект больше объекта, переданного в качестве параметра; отрицательное число, если вызывающий объект меньше объекта, переданного в качестве параметра; нуль, если объекты равны  
 \*/* @Override  
 public int compareTo(Object o) {  
 return ((Person) o).name.compareTo(this.name);  
 }  
  
 */\*\*  
 \* метод, который проверяет правильность введенных полей у человека  
 \** ***@param*** *prompt считываемая строка  
 \** ***@return*** *результат проверки и человека, если результат проверки true; null, если false  
 \*/* public static Pair<Boolean, Person> input(String prompt) {  
 System.*out*.println(prompt + ": ");  
  
 String line;  
 System.*out*.println("name: ");  
  
 line = IoHelper.*in*.nextLine();  
 if (line.isEmpty()) {  
 return new Pair<>(true, null);  
 }  
  
 String name = line;  
  
 Pair<Boolean, LocalDate> dp = IoHelper.*getLocalDate*("birthday");  
 if (!dp.getKey())  
 return new Pair<>(false, null);  
  
 Pair<Boolean, Location> lp = Location.*input*("location");  
 if (!lp.getKey())  
 return new Pair<>(false, null);  
  
 return new Pair<>(true, new Person(name, dp.getValue(), lp.getValue()));  
 }  
}

import javafx.util.Pair;  
import java.util.Locale;  
  
*/\*\*  
 \* класс, описывающий месторасположение человека  
 \*/*public class Location {  
 */\*\*  
 \* поле координата x месторасположения человека  
 \*/* private double x;  
 */\*\*  
 \* поле координата y месторасположения человека  
 \*/* private float y;  
 */\*\*  
 \* поле координата z месторасположения человека  
 \*/* private int z;  
 */\*\*  
 \* поле название месторасположения человека  
 \*/* private String name; //Поле может быть null  
  
 */\*\*  
 \* конструтор, позволяющий задать координаты и название месторасположения человека  
 \** ***@param*** *x координата x  
 \** ***@param*** *y координата y  
 \** ***@param*** *z координата z  
 \** ***@param*** *name название месторасположения  
 \*/* public Location (double x, float y, int z, String name) {  
 this.x = x;  
 this.y = y;  
 this.z = z;  
 this.name = name;  
 }  
 */\*\*  
 \* метод для получения строки с координатами и названием месторасположения человека в формате csv  
 \** ***@return*** *строка с координатами и названием месторасположения человека в формате csv  
 \*/* public String toCSVString() {  
 return String.*format*(Locale.*US*, "%f;%f;%d;%s", x, y, z, name);  
 }  
 */\*\*  
 \* метод для получения строки с координатами и названием месторасположения человека  
 \** ***@return*** *строка с координатами и названием месторасположения человека  
 \*/* @Override  
 public String toString() {  
 return String.*format*(Locale.*US*, "(%f, %f, %d)\nname: %s", x, y, z, name);  
 }  
  
 */\*\*  
 \* метод, который проверяет правильность введенных полей у месторасположения человека  
 \** ***@param*** *prompt считываемая строка  
 \** ***@return*** *результат проверки и месторасположение, если результат проверки true; null, если false  
 \*/* public static Pair<Boolean, Location> input(String prompt) {  
 System.*out*.println(prompt + ": ");  
  
 double x;  
 float y;  
 int z;  
 String name;  
  
 Pair<Boolean, Double> px = IoHelper.*getDouble*("x", true);  
 if (!px.getKey())  
 return new Pair<>(false, null);  
  
 if (px.getValue() == null)  
 return new Pair<>(true, null);  
  
 x = px.getValue();  
  
 Pair<Boolean, Float> py = IoHelper.*getFloat*("y");  
 if (!py.getKey())  
 return new Pair<>(false, null);  
 y = py.getValue();  
  
 Pair<Boolean, Integer> pz = IoHelper.*getInt*("z");  
 if (!pz.getKey())  
 return new Pair<>(false, null);  
 z = pz.getValue();  
  
 System.*out*.println("name: ");  
 name = IoHelper.*in*.nextLine();  
  
 return new Pair<>(true, new Location(x, y, z, name));  
  
 }  
}

**Выводы по работе.**