Федеральное государственное автономное образовательное учреждение высшего образования

«Национальный исследовательский университет ИТМО»

Факультет ПИ и КТ

Лабораторная работа №5

по дисциплине: «Программирование»

Вариант 3111002

Выполнил:

Болорболд Аригуун,

группа P3111

Преподаватель:

Письмак Алексей Евгеньевич

Санкт-Петербург

2023

1. **Задание:**

Внимание! У разных вариантов разный текст задания!

Реализовать консольное приложение, которое реализует управление коллекцией объектов в интерактивном режиме. В коллекции необходимо хранить объекты класса `Organization`, 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.Hashtable`
- 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 `json`
- Чтение данных из файла необходимо реализовать с помощью класса `java.io.InputStreamReader`
- Запись данных в файл необходимо реализовать с помощью класса `java.io.OutputStreamWriter`
- 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
- `insert null {element}` : добавить новый элемент с заданным ключом
- `update id {element}` : обновить значение элемента коллекции, id которого равен заданному
- `remove_key null` : удалить элемент из коллекции по его ключу
- `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_lower {element}` : удалить из коллекции все элементы, меньшие, чем заданный
- `history` : вывести последние 9 команд (без их аргументов)
- `remove_lower_key null` : удалить из коллекции все элементы, ключ которых меньше, чем заданный
- `filter_contains_name name` : вывести элементы, значение поля name которых содержит заданную подстроку
- `filter_less_than_type type` : вывести элементы, значение поля type которых меньше заданного
- `print_field_descending_annual_turnover` : вывести значения поля annualTurnover всех элементов в порядке убывания

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 Organization {
    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 double annualTurnover; //Значение поля должно быть больше 0
    private OrganizationType type; //Поле не может быть null
    private Address officialAddress; //Поле может быть null
}
public class Coordinates {
    private Double x; //Поле не может быть null
    private int y; //Максимальное значение поля: 614
}
public class Address {
    private String street; //Поле может быть null
    private String zipCode; //Длина строки должна быть не меньше 9, Поле не может
быть null
}
public enum OrganizationType {
    COMMERCIAL,
    PUBLIC,
    PRIVATE_LIMITED_COMPANY;
}
```

2. Диаграмма:

3. Исходный код:

consoleApp.Main:

```
package consoleApp;

import commands.*;
import utility.*;
import utility.Console;

import java.io.*;
import java.util.Scanner;

/**
 * Main class - runs the console app.
 */
public class Main {
    /**
     * String for CLI indentation.
     */
    public static final String CS1 = "$ ";
    /**
     * String for CLI indentation.
     */
    public static final String CS2 = "> ";
    /**
     * CLI argument passes on to this variable.
     */
```

```
    public static String CLI_ARGUMENT = null;
    /**
     * Main method - launches the console app.
     * @param args main
     */
    public static void main(String[] args) {
        Console.println("Welcome.");
        Scanner userScanner = new Scanner(System.in);
        if (args.length != 0) {
            CLI_ARGUMENT = args[0];
            File file = new File(CLI_ARGUMENT);
            if (!file.exists()) {
                try {
                    if (file.createNewFile()) {
                        OutputStream outputStream = new
FileOutputStream(file);
                        OutputStreamWriter osw = new
OutputStreamWriter(outputStream);
                        osw.write("{}");
                        Console.println("Файл успешно создан.");
                        osw.close();
                    }
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
            FileManager fileManager = new FileManager(CLI_ARGUMENT);
            OrganizationValidator organizationValidator = new
OrganizationValidator(userScanner);
            CollectionManager collectionManager = new
CollectionManager(fileManager);
            CommandManager commandManager = new CommandManager(
                    new clear(collectionManager),
                    new executeScript(),
                    new exit(),
                    new filterContainsName(collectionManager),
                    new filterLessThanType(collectionManager),
                    new help(),
                    new history(),
                    new info(collectionManager),
                    new insert(collectionManager, organizationValidator),
                    new
printFieldDescendingAnnualTurnover(collectionManager),
                    new removeKey(collectionManager),
                    new removeLower(collectionManager),
                    new removeLowerKey(collectionManager),
                    new save(collectionManager),
                    new show(collectionManager),
                    new updateID(collectionManager, organizationValidator)
            );
            Console console = new Console(commandManager, userScanner,
organizationValidator);
            console.InteractiveMode();
            Console.println(collectionManager);
        }
    }
```

utility:

CollectionManager.java:

```java
package utility;

import data.Organization;
import data.OrganizationType;

import java.time.LocalDateTime;
import java.util.*;
import java.util.stream.Collectors;

/**
 * Class for managing collection.
 *
 * @author Ariguun Erkevich Bolorbold
 */
public class CollectionManager {
    /**
     * Field for last initialization time.
     */
    private LocalDateTime lastInitTime;
    /**
     * Field for hashtable initialization.
     */
    private Hashtable<Integer, Organization> OrgCollection = new
Hashtable<>();
    /**
     * Field for last save time.
     */
    private LocalDateTime lastSaveTime;
    /**
     * Instance of file manager class.
     */
    private final FileManager fileManager;

    /**
     * Constructor for utility class CollectionManager.
     * @param fileManager File Manager
     */
    public CollectionManager(FileManager fileManager){
        this.lastInitTime = null;
        this.lastSaveTime = null;
        this.fileManager = fileManager;
        loadCollection();
    }
    /**
     * Getter for hashtable.
     * @return hashtable
     */
    public Hashtable<Integer, Organization> getCollection(){
        return OrgCollection;
    }

    /**
     * Getter for last initialization time.
     * @return last init time
     */
    public LocalDateTime getLastInitTime(){
        return lastInitTime;
    }

    /**
     * Getter for last save time.
```

```
     * @return last save time
     */
    public LocalDateTime getLastSaveTime(){
        return lastSaveTime;
    }

    /**
     * Getter for collection type.
     * @return collection type
     */
    public String getCollectionType(){
        return OrgCollection.getClass().getName();
    }

    /**
     * This method returns the size of the collection.
     * @return collection size
     */
    public int collectionSize(){
        return OrgCollection.size();
    }

    /**
     * Getter for last element of the collection.
     * @return last element
     */
    public Organization getLast(){
        Set<Integer> setOfKeys = OrgCollection.keySet();
        Iterator<Integer> iterator = setOfKeys.iterator();
        if(OrgCollection.isEmpty()) return null;
        if(!iterator.hasNext()){
            Integer key = iterator.next();
            return OrgCollection.get(key);
        } else return null;
    }
    /**
     *
     */
    public TreeMap<Integer, Organization> sortHashtable(Hashtable<Integer,
Organization> OrgCollection){
        TreeMap<Integer, Organization> tm = new TreeMap<>(OrgCollection);
        return tm;
    }
    /**
     * Field for set of keys for hashtable management.
     */
    Set<Integer> setOfKeys = OrgCollection.keySet();
    /**
     * Getter for hashtable element by ID.
     * @param id int
     * @return id
     */
    public Organization getByID(Integer id){
        Organization organization =  OrgCollection.get(id);
        if (organization == null){
            System.err.println("No such organization with given ID");
        }
        return organization;
    }

    /**
```

```
     * This method checks whether a collection element contains a certain
key.
     * @param id int
     * @return boolean value
     */
    public boolean containsKey(Integer id){
        return OrgCollection.containsKey(id);
    }

    /**
     * This method returns a collection element by value.
     * @param orgToFind Organization
     * @return Organization
     */
    public Organization getByValue(Organization orgToFind){
        for(Integer key : setOfKeys){
            if(OrgCollection.get(key).equals(orgToFind)) return
OrgCollection.get(key);
        }
        return null;
    }

    /**
     * This method returns collection elements by given type.
     * @param type Organization
     * @return type of organization
     */
    public Organization getByOrgType(String type){
        Set<Integer> setOfKeys = OrgCollection.keySet();
        for (Integer key : setOfKeys) {
            if
(OrgCollection.get(key).getType().toString().equals(type.toUpperCase(Locale
.ROOT)))
                return OrgCollection.get(key);
        }
        return null;
    }

    /**
     * This method returns annual turnovers by descending order.
     */
    public void printFieldDescendingAnnualTurnover(){
        TreeSet<Organization> copy = new
TreeSet<>(Collections.reverseOrder(Organization::compareToAnnualTurnover));
        ArrayList<Integer> arrayList = new ArrayList<>();
        copy.addAll(OrgCollection.values());
        for(Organization org : copy){
            arrayList.add(org.getAnnualTurnover());
        }
        Console.println(arrayList.toString().trim() + "\n");
    }

    /**
     * This method filters the collection by given type.
     * @param OrgTypeToFilter type
     * @return String type
     */
    public String OrganizationTypeFilteredInfo(OrganizationType
OrgTypeToFilter){
        Set<Integer> setOfKeys = OrgCollection.keySet();
        Iterator<Integer> iterator = setOfKeys.iterator();
```

```
        StringBuilder info = new StringBuilder();
        while(iterator.hasNext()){
            Integer key = iterator.next();
            if(OrgCollection.get(key).getType().ordinal() <
OrgTypeToFilter.ordinal()){
                info.append(OrgCollection.get(key)).append("\n\n");
            }
        }
        return info.toString().trim();
    }

    /**
     * This method inserts into the collection a given element.
     * @param key int
     * @param org Organization
     */
    public void insertToCollection(Integer key, Organization org){
        OrgCollection.put(key, org);
    }

    /**
     * This method removes an element from the collection.
     * @param key int
     */
    public void removeFromCollection(Integer key){
        OrgCollection.remove(key);
    }

    /**
     * This method removes lower value (annual turnover) elements.
     * @param annualTurnover annualTurnover
     */
    public void removeLower(Double annualTurnover) {
        OrgCollection.entrySet().removeIf(organization ->
organization.getValue().getAnnualTurnover() < annualTurnover);
    }

    /**
     * This method removes elements with lower key value.
     * @param key int
     */
    public void removeLowerKey(Integer key){
        OrgCollection.entrySet().removeIf(e -> e.getKey() < key);
    }

    /**
     * This method filters the collection by name.
     * @param name String
     * @return String
     */
    public String filterContainsName(String name){
        Map <Integer, String> result =
OrgCollection.values().stream().filter(organization ->
name.equals(organization.getName())).collect(Collectors.toMap(Organization:
:getId, Organization::getName));
        return result.toString();
    }
    /**
     * This method loads the collection.
     */
    private void loadCollection() {
```

```
    ContentValidator contentValidator = new ContentValidator();
    OrgCollection = contentValidator.validateContent();
    lastInitTime = LocalDateTime.now();
}


    /**
     * This method clears the collection.
     */
    public void clearCollection() {
        OrgCollection.clear();
    }

    /**
     * This method saves the collection.
     */
    public void saveCollection(){
        TreeMap<Integer, Organization> treeMap =
sortHashtable(OrgCollection);
        fileManager.writeCollection(treeMap);
        lastSaveTime = LocalDateTime.now();
    }

    /**
     * This method is an automatic id generator.
     * @return id
     */
    public int generateNextId() {
        return (OrgCollection.isEmpty()) ? 1 :
OrgCollection.values().stream().max(Comparator.comparing(Organization::getI
d)).get().getId()+1;
    }

    /**
     * CollectionManager implementation of general method toString()
     * @return String
     */
    @Override
    public String toString(){
        if(OrgCollection.isEmpty()) return "Empty collection";
        Set<Integer> setOfKeys = OrgCollection.keySet();
        Iterator<Integer> iterator = setOfKeys.iterator();
        StringBuilder info = new StringBuilder();
        while(iterator.hasNext()){
            int key = iterator.next();
            info.append(key).append(". ");
            info.append(OrgCollection.get(key).toString());
            info.append("\n-----------------------------------------------
-----------------------------------\n");
        }
        return info.toString();
    }
}
```

CommandManager.java:

```
package utility;

import commands.Command;
import exceptions.EmptyHistoryException;
```

```java
import java.util.ArrayList;
import java.util.List;

/**
 * Class for managing commands.
 * @author Ariguun Erkevich Bolorbold
 */
public class CommandManager {
    /**
     * Field for size of command history.
     */
    private final int CommandHistorySize = 9;
    /**
     * Field for array initialization of command history.
     */
    private final String[] CommandHistory = new String[CommandHistorySize];
    /**
     * Field for list initialization of used commands.
     */
    private final List<Command> commands = new ArrayList<>();
    /**
     * Field for command clear.
     */
    private final Command clear;
    /**
     * Field for command exit.
     */
    private final Command exit;
    /**
     * Field for command execute_script.
     */
    private final Command executeScript;
    /**
     * Field for command filter_contains_name.
     */
    private final Command filterContainsName;
    /**
     * Field for command filter_less_than_type.
     */
    private final Command filterLessThanType;
    /**
     * Field for command help.
     */
    private final Command help;
    /**
     * Field for command history.
     */
    private final Command history;
    /**
     * Field for command info.
     */
    private final Command info;
    /**
     * Field for command insert.
     */
    private final Command insert;
    /**
     * Field for command print_field_descending_annual_turnover.
     */
    private final Command printFieldDescendingAnnualTurnover;
    /**
```

```
     * Field for command remove_key.
     */
    private final Command removeKey;
    /**
     * Field for command remove_lower.
     */
    private final Command removeLower;
    /**
     * Field for command remove_lower_key.
     */
    private final Command removeLowerKey;
    /**
     * Field for command save.
     */
    private final Command save;
    /**
     * Field for command show.
     */
    private final Command show;
    /**
     * Field for command update.
     */
    private final Command updateID;

    /**
     * Constructor of utility class CommandManager.
     * @param clear clear
     * @param executeScript execute_script
     * @param exit exit
     * @param filterContainsName filter_contains_name
     * @param filterLessThanType filter_less_than_type
     * @param help help
     * @param history history
     * @param info info
     * @param insert insert
     * @param printFieldDescendingAnnualTurnover
print_field_descending_annual_turnover
     * @param removeKey remove_key
     * @param removeLower remove_lower
     * @param removeLowerKey remove_lower_key
     * @param save save
     * @param show show
     * @param updateID update
     */
    public CommandManager(Command clear, Command executeScript, Command
exit, Command filterContainsName, Command filterLessThanType, Command help,
Command history, Command info, Command insert, Command
printFieldDescendingAnnualTurnover, Command removeKey, Command removeLower,
Command removeLowerKey, Command save, Command show, Command updateID){
        this.clear = clear;
        this.executeScript = executeScript;
        this.exit = exit;
        this.filterContainsName = filterContainsName;
        this.filterLessThanType = filterLessThanType;
        this.help = help;
        this.history = history;
        this.info = info;
        this.insert = insert;
        this.printFieldDescendingAnnualTurnover =
printFieldDescendingAnnualTurnover;
        this.removeKey = removeKey;
```

```
        this.removeLower = removeLower;
        this.removeLowerKey = removeLowerKey;
        this.save = save;
        this.show = show;
        this.updateID = updateID;

        commands.add(clear);
        commands.add(executeScript);
        commands.add(exit);
        commands.add(filterContainsName);
        commands.add(filterLessThanType);
        commands.add(help);
        commands.add(history);
        commands.add(info);
        commands.add(insert);
        commands.add(printFieldDescendingAnnualTurnover);
        commands.add(removeKey);
        commands.add(removeLower);
        commands.add(removeLowerKey);
        commands.add(save);
        commands.add(show);
        commands.add(updateID);
    }

    /**
     * Getter of command history.
     * @return commandHistory array string
     */
    public String[] getCommandHistory(){
        return CommandHistory;
    }

    /**
     * Getter of command history as a list object.
     * @return list of commands
     */
    public List<Command> getCommands(){
        return commands;
    }
    /**
     * This method adds commands to history.
     * @param commandRecent String
     * @throws NullPointerException exception
     */
    public void addToHistory(String commandRecent) throws
NullPointerException{
        for (Command command : commands){
            if (command.getName().split(" ")[0].equals(commandRecent)){
                for (int i = CommandHistorySize-1; i>0; i--) {
                    CommandHistory[i] = CommandHistory[i-1];
                }
                CommandHistory[0] = commandRecent;
            }
        }
    }
    /**
     * This method is invoked when the user inputs an unavailable command.
     * @param arg String
     */
    public void noSuchCommand(String arg) {
        Console.println("Command '" + arg + "' not found. Use command
```

```
'help' for advice.");
    }
    /**
     * This method invokes the command execute_script.
     * @param argument String
     * @return boolean value
     */
    public boolean executeScript(String argument){
        return executeScript.apply(argument);
    }

    /**
     * This method invokes the command help.
     * @param argument String
     * @return boolean value
     */
    public boolean help(String argument) {
        if (help.apply(argument)) {
            for (Command command : commands) {
                Console.printTable(command.getName(), command.getSpec());
            }
            return true;
        } else return false;
    }
    /**
     * This method invokes the command info.
     * @param argument String
     * @return boolean value
     */
    public boolean info(String argument) {
        return info.apply(argument);
    }
    /**
     * This method invokes the command show.
     * @param argument String
     * @return boolean value
     */
    public boolean show(String argument) {
        return show.apply(argument);
    }
    /**
     * This method invokes the command exit.
     * @param argument String
     * @return boolean value
     */
    public boolean exit(String argument) {
        return exit.apply(argument);
    }
    /**
     * This method invokes the command insert.
     * @param argument String
     * @return boolean value
     */
    public boolean insert(String argument) {
        return insert.apply(argument);
    }
    /**
     * This method invokes the command
print_field_descending_annual_turnover
     * @param argument String
     * @return boolean value
```

```
     */
    public boolean printFieldDescendingAnnualTurnover(String argument){
        return printFieldDescendingAnnualTurnover.apply(argument);
    }
    /**
     * This method invokes the command history.
     * @param argument String
     * @return boolean value
     */
    public boolean history(String argument){
        if (history.apply(argument)) {
            try {
                if (CommandHistory.length == 0) throw new
EmptyHistoryException("You just started this session, that means the
history is empty", new RuntimeException());
                Console.println("Last used commands:");
                for (String s : CommandHistory) {
                    if (s != null) Console.println(" " + s);
                }
                return true;
            } catch (EmptyHistoryException exception) {
                Console.println("Not a single command was used yet!");
            }
        }
        return false;
    }
    /**
     * This method invokes the command update.
     * @param argument String
     * @return boolean value
     */
    public boolean updateID(String argument){
        return updateID.apply(argument);
    }
    /**
     * This method invokes the command clear.
     * @param argument String
     * @return boolean value
     */
    public boolean clear(String argument){
        return clear.apply(argument);
    }
    /**
     * This method invokes the command filter_contains_name.
     * @param argument String
     * @return boolean value
     */
    public boolean filterContainsName(String argument){
        return filterContainsName.apply(argument);
    }
    /**
     *This method invokes the command filter_less_than_type.
     * @param argument String
     * @return boolean value
     */
    public boolean filterLessThanType(String argument){
        return filterLessThanType.apply(argument);
    }
    /**
     * This method invokes the command remove_key.
     * @param argument String
```

```
     * @return boolean value
     */
    public boolean removeKey(String argument){
        return removeKey.apply(argument);
    }
    /**
     * This method invokes the command remove_lower.
     * @param argument String
     * @return boolean value
     */
    public boolean removeLower(String argument){
        return removeLower.apply(argument);
    }
    /**
     * This method invokes the command remove_lower_key.
     * @param argument String
     * @return boolean value
     */
    public boolean removeLowerKey(String argument){
        return removeLowerKey.apply(argument);
    }
    /**
     * This method invokes the command save.
     * @param argument String
     * @return boolean value
     */
    public boolean save(String argument){
        return save.apply(argument);
    }
    /**
     * CommandManager implementation of general method toString()
     * @return String
     */
    @Override
    public String toString() {
        return "CommandManager (utility class for commands)";
    }
}
```

Console.java:

```
package utility;

import consoleApp.Main;
import exceptions.InvalidElementCountException;
import exceptions.RecursionException;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.*;
import java.util.List;

/**
 * Class for console management.
 * @author Ariguun Erkevich Bolorbold
 */
public class Console {
    /**
     * Field for command manager
     */
    private final CommandManager commandManager;
```

```
    /**
     * Field for Scanner - used for user input.
     */
    private final Scanner userScanner;
    /**
     * Field for organization validator.
     */
    private final OrganizationValidator ov;
    /**
     * Field for list scriptStack - intended for script work.
     */
    private final List<String> scriptStack = new ArrayList<>();
    /**
     * Constructor for utility class Console.
     * @param commandManager command manager
     * @param sc stack
     * @param ov validator
     */
    public Console(CommandManager commandManager, Scanner sc,
OrganizationValidator ov) {
        this.commandManager = commandManager;
        this.userScanner = sc;
        this.ov = ov;
    }

    /**
     * Method for invoking interactive mode.
     */
    public void InteractiveMode(){
        String[] userCommand;
        int commandStatus;
        try {
            do {
                Console.print(Main.CS1);
                userCommand = (userScanner.nextLine().trim() + " ").split("
", 2);
                userCommand[1] = userCommand[1].trim();
                commandManager.addToHistory(userCommand[0]);
                commandStatus = executeCommand(userCommand);
            } while (commandStatus != 2);
        } catch (NoSuchElementException nsee) {
            Console.print("User input not detected");
        }
    }

    /**
     * Method for invoking script mode.
     * @param arg String
     * @return int
     */
    public int ScriptMode(String arg) {
        String[] userCommand;
        int commandStatus;
        scriptStack.add(arg);
        try (Scanner scrSc = new Scanner(new File(arg))) {
            if(!scrSc.hasNext()) throw new NoSuchElementException();
            Scanner tmpScanner = ov.getUserScanner();
            ov.setUserScanner(scrSc);
            ov.setFileMode();
            do {
                userCommand = (scrSc.nextLine().trim() + " ").split(" ",
```

```
2);
                userCommand[1] = userCommand[1].trim();
                while (scrSc.hasNextLine() && userCommand[0].isEmpty()) {
                    userCommand = (scrSc.nextLine().trim() + " ").split("
", 2);
                    userCommand[1] = userCommand[1].trim();
                }
                Console.println(Main.CS1 + String.join(" ", userCommand));
                if (userCommand[0].equals("execute_script")) {
                    for (String script : scriptStack) {
                        if (userCommand[1].equals(script))
                            throw new RecursionException("Unchecked
recursion detected", new RuntimeException());
                    }
                }
                commandStatus = executeCommand(userCommand);
            } while(commandStatus == 0 && scrSc.hasNextLine());
            ov.setUserScanner(tmpScanner);
            ov.setUserMode();
            if(commandStatus == 1
&& !(userCommand[0].equals("execute_script")
&& !userCommand[1].isEmpty())){
                Console.printError("EXECUTION ERROR: Please debug your
script");
            return commandStatus;}
            else if(commandStatus == 2 && userCommand[0].equals("exit") &&
userCommand[1].isEmpty()){
                System.exit(0);
            }
        } catch(FileNotFoundException fnfe) {
            Console.printError("Script file not found. If there is one,
then try changing the permission of the file. Maybe chmod 777, idk.");
        } catch(InvalidElementCountException iece) {
            Console.printError("Script file is empty");
        } catch(RecursionException re) {
            Console.printError("CRITICAL ERROR: Recursion detected in
script file");
        } finally {
            scriptStack.remove(scriptStack.size()-1);
        }
        return 1;
    }
    /**
     * Method which executes user-input commands. Added cases just for
banter.
     * @param userCommand String array
     * @return int
     */
    private int executeCommand(String[] userCommand){
        String arg = userCommand[0].toLowerCase();
        switch (arg){
            case "":
                break;
            case "clear", "сдуфк":
                if(!commandManager.clear(userCommand[1])) return 1;
                break;
            case "execute_script", "учусгеу_ыскшзе":
                if(!commandManager.executeScript(userCommand[1])) return 1;
                else return ScriptMode(userCommand[1]);
            case "exit", "учше":
                if(!commandManager.exit(userCommand[1])) return 1;
```

```
                else return 2;
            case "filter_contains_name", "ашдеук_сштефшты_тфьу":
                if(!commandManager.filterContainsName(userCommand[1]))
return 1;
                break;
            case "filter_less_than_type", "ашдеук_дуыы_ерфт_ензу":
                if(!commandManager.filterLessThanType(userCommand[1]))
return 1;
                break;
            case "help", "рудз":
                if(commandManager.help(userCommand[1])) return 1;
                break;
            case "info", "штащ":
                if(!commandManager.info(userCommand[1])) return 1;
                break;
            case "insert", "штыуке":
                if(!commandManager.insert(userCommand[1])) return 1;
                break;
            case "history", "ршыещкн":
                if(!commandManager.history(userCommand[1])) return 1;
                break;
            case "print_field_descending_annual_turnover",
"зкште_ашудв_вуысутвштп_фттгфд_егктщмук", "pfdat":

if(!commandManager.printFieldDescendingAnnualTurnover(userCommand[1]))
return 1;
                break;
            case "remove_key", "куьщму_лун":
                if(!commandManager.removeKey(userCommand[1])) return 1;
                break;
            case "remove_lower", "куьщму_дщцук":
                if(!commandManager.removeLower(userCommand[1])) return 1;
                break;
            case "remove_lower_key", "куьщму_дщцук_лун":
                if(!commandManager.removeLowerKey(userCommand[1])) return
1;
                break;
            case "save", "ыфму":
                if(!commandManager.save(userCommand[1])) return 1;
                break;
            case "show", "ырщц":
                if(!commandManager.show(userCommand[1])) return 1;
                break;
            case "update", "гзвфеу":
                if(!commandManager.updateID(userCommand[1])) return 1;
                break;
            default: commandManager.noSuchCommand(userCommand[0]);
            return 1;
        }
        return 0;
    }
    /**
     * Custom console version of general print method.
     * @param toOut String
     */
    public static void print(Object toOut){
        System.out.print("\033[1;35m" + toOut + "\u001B[0m");
    }
    /**
     * Custom console version of general println method.
     * @param toOut String
```

```
     */
    public static void println(Object toOut){
        System.out.println("\033[4;32m" + toOut + "\u001B[0m");
    }
    /**
     * Custom console version of general err method.
     * @param toOut String
     */
    public static void printError(Object toOut){
        System.out.println("\u001B[41m" + "\u001B[30m" + toOut +
"\u001B[0m");
    }
    /**
     * Custom console version of general printf method (adopted for
printing tables).
     * @param e1 Object
     * @param e2 Object
     */
    public static void printTable(Object e1, Object e2){
        System.out.printf("\u001B[40m" + "-----------------------------------------
-----------------------------------------%n");
        System.out.printf("\u001B[36m" + "| %-38s | %-10s | %n", e1, e2 +
"\u001B[0m");
    }

    /**
     * Console implementation of general method toString()
     * @return String
     */
    @Override
    public String toString(){
        return "Console: CLI class";
    }
}
```

FileManager.java:

```
package utility;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
import com.google.gson.JsonParseException;
import com.google.gson.reflect.TypeToken;
import consoleApp.Main;
import data.Organization;
import java.io.*;
import java.lang.reflect.Type;
import java.nio.charset.StandardCharsets;
import java.util.*;

/**
 * Class for file management.
 * @author Ariguun Erkevich Bolorbold
 */
public class FileManager {
    /**
     * instance of a Gson object - library import of this jar file was
painful.
     */
    public Gson gson = new GsonBuilder().setPrettyPrinting().create();
```

```java
    /**
     * Constructor of utility class FileManager.
     * @param ConsoleArg String
     */
    public FileManager(String ConsoleArg){
        Main.CLI_ARGUMENT = ConsoleArg;
    }
    Hashtable<Integer, Organization> collection;
    /**
     * This method writes collection to a file.
     * @param collection hashtable
     */
    public void writeCollection(TreeMap<Integer, Organization> collection){
        try {
            OutputStream outputStream = new
FileOutputStream(Main.CLI_ARGUMENT);
            Writer writer = new OutputStreamWriter(outputStream,
StandardCharsets.UTF_8);
            String a = gson.toJson(collection);
            writer.write(a);
            writer.close();
        } catch (IOException e) {
            System.err.println("File cannot be opened");
        }
    }
    /**
     * This method reads collection from a file.
     * @return hashtable
     */
    public Hashtable<Integer, Organization> readCollection(){
        if(Main.CLI_ARGUMENT != null){
            File file = new File(Main.CLI_ARGUMENT);
            try(FileInputStream inputStream = new FileInputStream(file)){
                Reader inputStreamReader = new
InputStreamReader(inputStream, StandardCharsets.UTF_8);
                final Type collectionType = new
TypeToken<Hashtable<Integer, Organization>>(){}.getType();
                final BufferedReader reader = new
BufferedReader(inputStreamReader);
                collection = gson.fromJson(reader, collectionType);
                return collection;
            } catch (FileNotFoundException e){
                System.err.println("File not found");
            } catch (NoSuchElementException e){
                System.err.println("The file is emptier than the brain of
the developer");
            } catch (JsonParseException | NullPointerException e){
                System.err.println("No collection detected");
            } catch (IllegalStateException e){
                System.err.println("Unknown error");
                System.exit(0);
            } catch (IOException e) {
                Console.printError("Input/Output operation interrupted");
            }
        } else System.err.println("Not found");
        return new Hashtable<>();
    }

    /**
     * FileManager implementation of general method toString()
     * @return String
```

```
     */
    @Override
    public String toString(){
        return "FileManager (utility class for file management)";
    }
}
```

OrganizationValidator.java:

```
/**
 * Utility package for console app.
 */
package utility;

import main.java.consoleApp.Main;
import data.Address;
import data.Coordinates;
import data.OrganizationType;
import exceptions.InvalidInputException;
import exceptions.InvalidTypeException;
import exceptions.NullValueException;
import exceptions.ValueExceededException;

import java.util.*;

/**
 * Class for validating user input.
 * @author Ariguun Erkevich Bolorbold
 */
public class OrganizationValidator {
    /**
     * Field for y - maximum limit.
     */
    private static final int MAX_Y = 614;
    /**
     * Field for zip code - limit for minimum length of zip code.
     */
    private static final int MIN_ZIP = 9;
    /**
     * Field for annual turnover - limit for minimum value of annual
turnover.
     */
    private static final int MIN_ANNUAL_TURNOVER = 0;
    /**
     * Field for Scanner - reads user input.
     */
    private Scanner userScanner;
    /**
     * Field for boolean fileMode.
     */
    private boolean fileMode;
    /**
     * Constructor of utility class OrganizationValidator.
     * @param userScanner Scanner
     */

    static Set<Integer> IDset = new TreeSet<>();
    static Set<Integer> keySet = new TreeSet<>();
    public OrganizationValidator(Scanner userScanner){
        this.userScanner = userScanner;
        fileMode = false;
    }
```

```
    /**
     * Setter for userScanner.
     * @param userScanner Scanner
     */
    public void setUserScanner(Scanner userScanner){
        this.userScanner = userScanner;
    }
    /**
     * Getter for userScanner.
     * @return userScanner
     */
    public Scanner getUserScanner() {
        return userScanner;
    }
    /**
     * Setter for fileMode.
     */
    public void setFileMode(){
        fileMode = true;
    }

    /**
     * Setter for userMode.
     */
    public void setUserMode(){
        fileMode = false;
    }

    /**
     * This method keeps user input name in line with limitations given by
the task.
     * @return String name
     * @throws InvalidInputException exception
     */
    public String askName() throws InvalidInputException{
        String name;
        while (true) {
            try {
                Console.println("Insert name: ");
                Console.print(Main.CS2);
                name = userScanner.nextLine().trim();
                if (fileMode) System.out.println(name);
                if (name.equals("")) throw new NullValueException("Field
must not be empty", new RuntimeException());
                break;
            } catch (NoSuchElementException e) {
                System.err.println("Name not found");
            } catch (NullValueException e) {
                System.err.println("Name mustn't be empty");
            } catch (IllegalStateException e) {
                System.err.println("Unknown error");
                System.exit(0);
            }
        }
        return name;
    }
    /**
     * This method keeps user input x in line with limitations given by the
task.
     * @return double x
     * @throws InvalidInputException exception
```

```
     */
    public double askX() throws InvalidInputException {
        String strX;
        double x;
        while (true) {
            try {
                Console.println("Insert coordinate x:");
                Console.print(Main.CS2);
                strX = userScanner.nextLine().trim();
                if (fileMode) Console.println(strX);
                x = Double.parseDouble(strX);
                if (checkX(x)) throw new NullValueException("Coordinate x
is null", new RuntimeException());
                break;
            } catch (NullValueException nve) {
                Console.printError("Coordinate x is null");
                if (fileMode) throw new NullValueException("Coordinate x is
null", new RuntimeException());
            } catch (NoSuchElementException e) {
                Console.printError("Coordinate X not recognized");
                if (fileMode) throw new InvalidInputException("Coordinate X
not recognized", new RuntimeException());
            } catch (NumberFormatException e) {
                Console.printError("Coordinate X must be a number");
                if (fileMode) throw new InvalidInputException("Coordinate X
must be a number", new RuntimeException());
            } catch (NullPointerException | IllegalStateException e) {
                Console.printError("Unknown error");
                System.exit(0);
            }
        }
        return x;
    }
    /**
     * This method keeps user input y in line with limitations given by the
task.
     * @return float y
     * @throws InvalidInputException exception
     */
    public float askY() throws InvalidInputException {
        String strY;
        float y;
        while (true) {
            try {
                Console.println("Insert coordinate Y < " + MAX_Y + ":");
                Console.print(Main.CS2);
                strY = userScanner.nextLine().trim();
                if (fileMode) Console.println(strY);
                y = Float.parseFloat(strY);
                if (checkY(y)) throw new ValueExceededException("The amount
of coordinate y must not exceed" + " " + MAX_Y, new RuntimeException());
                break;
            } catch (NoSuchElementException nsee) {
                Console.printError("Coordinate X not recognized");
                if (fileMode) throw new InvalidInputException("The amount
of coordinate y must not exceed" + " " + MAX_Y, new RuntimeException());
            } catch (ValueExceededException vee) {
                Console.printError("The amount of coordinate y must not
exceed " + MAX_Y + "");
                if (fileMode) throw new InvalidInputException("The amount
of coordinate y must not exceed" + " " + MAX_Y, new RuntimeException());
```

```
            } catch (NumberFormatException nfe) {
                Console.printError("Coordinate Y must be a number");
                if (fileMode) throw new InvalidInputException("The amount
of coordinate y must not exceed 614", new RuntimeException());
            } catch (NullPointerException | IllegalStateException
exception) {
                Console.printError("Unknown error");
                System.exit(0);
            }
        }
        return y;
    }
    /**
     * This method keeps user input zip code in line with limitations given
by the task.
     * @return String zipCode
     * @throws ValueExceededException exception
     */
    public String askZipCode() throws ValueExceededException {
        String ZipCode;
        while (true) {
            try {
                Console.println("Insert zip code: ");
                Console.print(Main.CS2);
                ZipCode = userScanner.nextLine().trim();
                if (fileMode) System.out.println(ZipCode);
                if (checkZipCode(ZipCode)) throw new
ValueExceededException("Zip code must be not null and must exceed 9 by
character length", new RuntimeException());
                break;
            } catch (ValueExceededException vee){
                System.err.println("Zip code must exceed 9 by character
length");
            } catch (NoSuchElementException nsee) {
                System.err.println("Zip code not recognized");
            } catch (NullValueException nve) {
                System.err.println("Field 'name' must not be empty");
            } catch (IllegalStateException ise) {
                System.err.println("Unknown error");
                System.exit(0);
            }
        }
        return ZipCode;
    }

    /**
     * This method initiates the validation of user input coordinates.
     * @return Coordinates
     * @throws InvalidInputException exception
     */
    public Coordinates askCoordinates() throws InvalidInputException {
        double x;
        float y;
        x = askX();
        y = askY();
        return new Coordinates(x, y);
    }

    /**
     * This method keeps user input annual turnover in line with
limitations given by the task.
```

```
     * @return Int annual turnover
     * @throws InvalidInputException exception
     */
    public Double askAnnualTurnover() throws InvalidInputException {
        String strAnnualTurnover;
        double annualTurnover;
        while (true) {
            try {
                Console.println("Insert annual turnover: ");
                Console.print(Main.CS2);
                strAnnualTurnover = userScanner.nextLine().trim();
                if (fileMode) Console.println(strAnnualTurnover);
                annualTurnover = Integer.parseInt(strAnnualTurnover);
                if (checkAnnualTurnover(annualTurnover)) throw new
ValueExceededException("NEGATIVE INCOME?", new RuntimeException());
                break;
            } catch (NoSuchElementException e) {
                Console.printError("Can't recognize");
                if (fileMode) throw new InvalidInputException("User input
not detected", new RuntimeException());
            } catch (ValueExceededException e) {
                Console.printError("Annual turnover must be at least
something");
                if (fileMode) throw new InvalidInputException("Did you
really insert a number?", new RuntimeException());
            } catch (NumberFormatException e) {
                Console.printError("Annual turnover must be a number");
                if (fileMode) throw new InvalidInputException("Unknown
error", new RuntimeException());
            } catch (NullPointerException | IllegalStateException e) {
                Console.printError("Unknown error");
                System.exit(0);
            }
        }
        return annualTurnover;
    }

    /**
     * This method keeps user input organization type in line with
limitations given by the task.
     * @return Type
     */
    public OrganizationType askOrganizationType(){
        String strOrgType;
        OrganizationType organizationType;
        while (true) {
            try {
                System.out.println("\033[1;34m" + "List of organization
types - " + OrganizationType.nameList() + "\u001B[0m");
                Console.println("Insert organization type:");
                Console.print(Main.CS2);
                strOrgType = userScanner.nextLine().trim();
                if (fileMode) Console.println(strOrgType);
                organizationType =
OrganizationType.valueOf(strOrgType.toUpperCase());
                if (checkOrgType(organizationType)) throw new
InvalidInputException("An organization type must not be null and must be
available", new RuntimeException());
                break;
            } catch (InvalidInputException e) {
                Console.printError("An organization type must not be null
```

```
and must be available");
                if (fileMode) throw new InvalidInputException("An
organization type must not be null and must be available", new
RuntimeException());
            } catch (InvalidTypeException exception) {
                Console.printError("Type not recognized");
                if (fileMode) throw new InvalidInputException("Type not
recognized", new RuntimeException());
            } catch (IllegalArgumentException exception) {
                Console.printError("There's no such type");
                if (fileMode) throw new InvalidInputException("There's no
such type", new RuntimeException());
            } catch (IllegalStateException exception) {
                Console.printError("Unknown error");
                System.exit(0);
            }
        }
        return organizationType;
    }

    /**
     * This method keeps user input address in line with limitations given
by the task.
     * @return String address name
     */
    public String askAddressName(){
        String strOfficialAddress;
        while (true) {
            try {
                Console.println("Insert address:");
                Console.print(Main.CS2);
                strOfficialAddress = userScanner.nextLine().trim();
                if (fileMode) Console.println(strOfficialAddress);
                break;
            } catch (NoSuchElementException exception) {
                Console.printError("Address not recognized");
                if (fileMode) throw new InvalidInputException("Address not
recognized", new IllegalArgumentException());
            } catch (IllegalStateException exception) {
                Console.printError("Unknown error");
                System.exit(0);
            }

        }
        return strOfficialAddress;
    }

    /**
     * This method initiates the validation of user input address.
     * @return String Address
     * @throws InvalidInputException exception
     */
    public Address askAddress() throws InvalidInputException{
        String name;
        String ZipCode;
        name = askAddressName();
        ZipCode = askZipCode();
        return new Address(name, ZipCode);
    }
    /**
     * This method is invoked during a data update.
```

```
     * @param question final question
     * @return boolean
     * @throws InvalidInputException exception
     */
    public boolean askQuestion(String question) throws
InvalidInputException {
        String finalQuestion = question + " (+/-):";
        String answer;
        while (true) {
            try {
                Console.println(finalQuestion);
                Console.print(Main.CS2);
                answer = userScanner.nextLine().trim();
                if (fileMode) Console.println(answer);
                if (!answer.equals("+") && !answer.equals("-")) throw new
ValueExceededException("User input must be either '+' or '-'", new
IllegalArgumentException());
                break;
            } catch (NoSuchElementException exception) {
                Console.printError("Answer not recognized");
                if (fileMode) throw new InvalidInputException("Answer not
recognized", new IllegalArgumentException());
            } catch (ValueExceededException exception) {
                Console.printError("Answer must be either '+' or '-'");
                if (fileMode) throw new InvalidInputException("Answer must
be either '+' or '-'", new IllegalArgumentException());
            } catch (IllegalStateException exception) {
                Console.printError("Unknown error");
                System.exit(0);
            }
        }
        return answer.equals("+");
    }

    /**
     * This method checks the ID for required constraints
     * @param ID Integer
     * @return
     */
    protected static boolean checkID(Integer ID) {
        return ID == null || ID < 0;
    }
    protected static boolean checkUniqueID(Integer ID) {
        if(IDset.contains(ID)) {
            return true;
        } else {
            IDset.add(ID);
            return false;
        }
    }
    protected static boolean checkName(String name) {
        return name == null || name.isEmpty();
    }
    protected static boolean checkX(Double x) {
        return x == Float.MIN_VALUE;
    }
    protected static boolean checkY(float y) {
        return y > MAX_Y;
    }
    protected static boolean checkDate(Date creationDate) {
        return creationDate == null || creationDate.toString().equals("");
```

```
    }
    protected static boolean checkAnnualTurnover(Double annualTurnover) {
        return annualTurnover <= MIN_ANNUAL_TURNOVER;
    }
    protected static boolean checkOrgType(OrganizationType type) {
        return type == null || !(type.equals(OrganizationType.COMMERCIAL)
|| type.equals(OrganizationType.PUBLIC) ||
type.equals(OrganizationType.PRIVATE_LIMITED_COMPANY));
    }
    protected static boolean checkZipCode(String zipCode) {
        return zipCode == null || zipCode.length() < MIN_ZIP;
    }
    /**
     * OrganizationValidator implementation of general method toString()
     * @return String
     */
    @Override
    public String toString(){
        return "GroupAsker (utility class for user queries)";
    }
}
}
```

ContentValidator.java:

```
package utility;
import data.*;
import main.java.consoleApp.Main;

import java.io.IOException;
import java.util.*;

public class ContentValidator {
    /**
     * Field for initialization of a FileManager instance.
     */
    FileManager fileManager = new FileManager(Main.CLI_ARGUMENT);
    /**
     * Validates the contents of a file in case of external editing.
     */
    public Hashtable<Integer, Organization> validateContent() {
        Hashtable<Integer, Organization> org =
fileManager.readCollection();
        Set<Integer> keys = org.keySet();
        for (Integer i : keys) {
            Organization orgToCheck = org.get(i);
            Integer ID = orgToCheck.getId();
            String name = orgToCheck.getName();
            Coordinates coordinates = orgToCheck.getCoordinates();
            double x = coordinates.getX();
            float y = coordinates.getY();
            Date date = orgToCheck.getCreationDate();
            Double annualTurnover = orgToCheck.getAnnualTurnover();
            OrganizationType type = orgToCheck.getType();
            Address address = orgToCheck.getOfficialAddress();
            String zip = address.getZipCode();
            if (OrganizationValidator.checkID(ID) ||
OrganizationValidator.checkUniqueID(ID)) {
                Console.printError("WARNING: This element's " + "(" + i +
")" + " ID field was altered externally, and therefore to preserve the
integrity of data constraints, will not be added to the collection");
                Console.printError("The organization ID does not meet
```

```
required constraints: ");
                org.remove(i);
            }
            if (OrganizationValidator.checkName(name)) {
                Console.printError("WARNING: This element's " + "(" + i +
")" + " name value was altered externally, and therefore to preserve the
integrity of data constraints, will not be added to the collection");
                Console.printError("The organization's name is either null
or empty");
                org.remove(i);
            }
            if (OrganizationValidator.checkX(x)) {
                Console.printError("WARNING: This element's " + "(" + i +
")" + " abscissa value was altered externally, and therefore to preserve
the integrity of data constraints, will not be added to the collection");
                Console.printError("The organization's abscissa value must
not be null");
                org.remove(i);
            }
            if (OrganizationValidator.checkY(y)) {
                Console.printError("WARNING: This element's " + "(" + i +
")" + " ordinate value was altered externally, and therefore to preserve
the integrity of data constraints, will not be added to the collection");
                Console.printError("The organization's ordinate value
exceeds the given limit of 614");
                org.remove(i);
            }
            if (OrganizationValidator.checkDate(date)) {
                Console.printError("WARNING: This element's " + "(" + i +
")" + " date value was altered externally, and therefore to preserve the
integrity of data constraints, will not be added to the collection");
                Console.printError("The creation time of this entry must
not be null and is automatically generated");
                org.remove(i);
            }
            if (OrganizationValidator.checkAnnualTurnover(annualTurnover))
{
                Console.printError("WARNING: This element's " + "(" + i +
")" + " annual turnover value was altered externally, and therefore to
preserve the integrity of data constraints, will not be added to the
collection");
                Console.printError("The organization's annual turnover does
not meet required constraints: must not be null and must be a positive
integer");
                org.remove(i);
            }
            if (OrganizationValidator.checkOrgType(type)) {
                Console.printError("WARNING: This element's " + "(" + i +
")" + " type value was altered externally, and therefore to preserve the
integrity of data constraints, will not be added to the collection");
                Console.printError("The organization's type does not match
any of the available ones");
                org.remove(i);
            }
            if (OrganizationValidator.checkZipCode(zip)) {
                Console.printError("WARNING: This element's " + "(" + i +
")" + " zip code value was altered externally, and therefore to preserve
the integrity of data constraints, will not be added to the collection");
                Console.printError("The organization's zip code does not
meet required constraints: must not be null and must not be shorter than 9
characters");
```

```
                org.remove(i);
            }
        }
        return org;
    }
}
```

Data:

Address.java:

```
package data;

/**
 * Custom class for organization address.
 */
public class Address {
    /**
     * Field for street.
     */
    private String street;
    /**
     * Field for zip code.
     */
    private String zipCode;

    /**
     * Constructor for data class Address
     * @param street String
     * @param zipCode String
     */
    public Address(String street, String zipCode){
        this.street = street;
        this.zipCode = zipCode;
    }

    /**
     * @return String Street
     */
    public String getStreet(){
        return street;
    }

    /**
     * Getter method for zip code
     * @return String ZipCode
     */
    public String getZipCode(){
        return zipCode;
    }

    /**
     * Address implementation for general method toString()
     * @return String
     */
    @Override
    public String toString(){
        return "Street: " + street + ", Zip code: " + zipCode;
    }

    /**
```

```
     * Address implementation for general method equals()
     * @param o Object
     * @return boolean value
     */
    @Override
    public boolean equals(Object o){
        if(this == o) return true;
        if(o instanceof Address addr){
            return street.equals(addr.street) &&
zipCode.equals(addr.zipCode);
        }
        return false;
    }
}
```

Coordinates.java:

```
package data;

/**
 * Custom class for organization coordinates.
 * It points to the general location of an organization.
 */
public class Coordinates {
    /**
     * (double) abscissa axis of data class Coordinates
     */
    private Double x; //Field must not be null
    /**
     * (float) ordinate axis of data class Coordinates
     */
    private float y; //Maximum value of y: 614

    /**
     * Constructor for data class Coordinates
     * @param x double
     * @param y float
     */
    public Coordinates(Double x, float y){
        this.x = x;
        this.y = y;
    }

    /**
     * Getter for value x
     * @return x
     */
    public double getX(){
        return x;
    }
    /**
     * Getter for value y
     * @return y
     */
    public float getY(){
        return y;
    }

    /**
     * Coordinates implementation of general method toString()
     * @return String
```

```
     */
    @Override
    public String toString(){
        return "X = " + x + "; " + "Y = " + y;
    }
    /**
     * Coordinates implementation of general method equals()
     * @param o Object
     * @return boolean
     */
    @Override
    public boolean equals(Object o){
        if(this == o) return true;
        if(o instanceof Coordinates coordinates){
            return x.equals(coordinates.x) && (y == coordinates.getY());
        }
        return false;
    }
}
```

Organization.java:

```
package data;

import java.util.Date;

/**
 * Custom class for organizations.
 * The general data class for creating an instance of an organization.
 */
public class Organization implements Comparable<Organization>{
    /**
     * This field represents the unique identifiable of an organization.
Field value must not be null, must be more than 0 and must be automatically
generated.
     */
    private Integer id;
    /**
     * This field represents the official name of an organization. Field
value must not be null and must not be empty.
     */
    private String name;
    /**
     * This field represents the coordinates of an organization. Notice
that this field is represented as its own class. Field must not be null.
     */
    private Coordinates coordinates;
    /**
     * This field represents the creation date of an organization. This
field is generated automatically and MUST DEFINITELY not be null.
     */
    private java.util.Date creationDate;
    /**
     * This field represents the annual turnover of an organization. Field
value must be greater than 0.
     */
    private int annualTurnover;
    /**
     * This field represents the type of organization. Notice that this
field is represented as its own class. Field must not be null.
     */
```

```
    private OrganizationType type;
    /**
     * This field represents the official address of an organization.
Notice that this field is represented as its own class. This field CAN be
null.
     */
    private Address officialAddress;

    /**
     * Constructor of the data class Organization.
     * @param id int
     * @param name String
     * @param coordinates Coordinates
     * @param creationDate Date
     * @param annualTurnover int
     * @param type OrganizationType
     * @param officialAddress Address
     */
    public Organization(Integer id, String name, Coordinates coordinates,
Date creationDate, int annualTurnover, OrganizationType type, Address
officialAddress){
        this.id = id;
        this.name = name;
        this.coordinates = coordinates;
        this.creationDate = creationDate;
        this.annualTurnover = annualTurnover;
        this.type = type;
        this.officialAddress = officialAddress;
    }
    /**
     * Getter for ID
     * @return id
     */
    public Integer getId(){
        return id;
    }
    /**
     * Getter for name
     * @return name
     */
    public String getName(){
        return name;
    }
    /**
     * Getter for coordinates
     * @return coordinates Coordinates
     */
    public Coordinates getCoordinates(){
        return coordinates;
    }
    /**
     * Getter for creation date
     * @return creationDate Date
     */
    public Date getCreationDate(){
        return creationDate;
    }
    /**
     * Getter for annual turnover
     * @return annualTurnover int
     */
```

```
    public int getAnnualTurnover(){
        return annualTurnover;
    }
    /**
     * Getter for organization type
     * @return organization type
     */
    public OrganizationType getType(){
        return type;
    }

    /**
     * Getter for official address
     * @return officialAddress Address
     */
    public Address getOfficialAddress(){
        return officialAddress;
    }
    /**
     * Comparable method: compares an organization object to another.
     * @param organization organization
     * @return int id
     */
    public int compareTo(Organization organization){
        return id.compareTo(organization.getId());
    }
    /**
     * Comparable method: compares an organization object to another by
annual turnover.
     * @param org Organization
     * @return annual turnover
     */
    public int compareToAnnualTurnover(Organization org){
        if(Integer.valueOf(annualTurnover).equals(org.getAnnualTurnover()))
return -1;
        if(annualTurnover > org.getAnnualTurnover()) return 1;
        else return -1;
    }
    /**
     * Organization implementation of general method equals()
     * @return String
     */
    @Override
    public String toString(){
        String output = "";
        output += "Organization №" + id;
        output += " (added " + creationDate + " " + creationDate.getTime()
+ ")";
        output += "\n Name: " + name;
        output += "\n Coordinates: " + coordinates;
        output += "\n Annual turnover: " + annualTurnover;
        output += "\n Organization type: " + type;
        output += "\n Legal address: " + officialAddress;
        return output;
    }
    /**
     * Organization implementation of general method hashCode()
     * @return int hash code
     */
    @Override
    public int hashCode(){
```

```
        return name.hashCode() + coordinates.hashCode() +
creationDate.hashCode() + annualTurnover + type.hashCode() +
officialAddress.hashCode();
    }
    /**
     * Organization implementation of general method equals()
     * @param o Object
     * @return boolean
     */
    @Override
    public boolean equals(Object o){
        if (this == o) return true;
        if (o instanceof Organization org){
            return name.equals(org.getName()) &&
coordinates.equals(org.getCoordinates()) &&
creationDate.equals(org.getCreationDate()) && (annualTurnover ==
org.getAnnualTurnover()) && type.equals(org.getType()) &&
officialAddress.equals(org.getOfficialAddress());
        }
        return false;
    }
}
```

OrganizationType.java:

```
package data;

/**
 * Enumeration class for organization types
 */
public enum OrganizationType {
    /**
     * Organization type #1.
     */
    COMMERCIAL,
    /**
     * Organization type #2.
     */
    PUBLIC,
    /**
     * Organization type #3.
     */
    PRIVATE_LIMITED_COMPANY;
    /**
     * This method returns the list of available types.
     * @return String
     */
    public static String nameList(){
        StringBuilder nameList = new StringBuilder("\n  Types: \n");
        for(OrganizationType organizationType : values()){
            nameList.append(organizationType.name()).append(", ");
        }
        nameList = new StringBuilder(nameList.substring(0,
nameList.length() - 2));

nameList.append("\n=====================================================");
        return nameList.toString();
    }
}
```

Commands:

iCommand.java:

```java
package commands;
/**
 * Primordial interface for all commands.
 */
public interface iCommand {
    /**
     * Abstract command for returning command name.
     * @return String name
     */
    String getName();
    /**
     * Abstract command for returning command specifications.
     * @return String spec
     */
    String getSpec();
    /**
     * Abstract command for command execution.
     * @param arg String
     * @return boolean
     */
    boolean apply(String arg);
}
```

Command.java:

```java
package commands;

import java.util.Objects;

/**
 * Abstract parent class of Command.
 */
public abstract class Command implements iCommand{
    /**
     * Field for command name.
     */
    private final String name;
    /**
     * Field for command specification.
     */
    private final String spec;
    /**
     * Constructor for abstract class Command.
     * @param name name
     * @param spec spec
     */
    public Command(String name, String spec) {
        this.name = name;
        this.spec = spec;
    }
    /**
     * Getter for command name.
     * @return String name
     */
    @Override
    public String getName() {
        return name;
```

```
    }
    /**
     * Getter for command specification.
     * @return String spec
     */
    @Override
    public String getSpec() {
        return spec;
    }
    /**
     * Command implementation of general method hashCode.
     * @return int hash code
     */
    @Override
    public int hashCode(){
        return name.hashCode() + spec.hashCode();
    }
    /**
     * Command implementation of general method equals.
     * @param obj Object
     * @return boolean
     */
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        Command command = (Command) obj;
        return Objects.equals(name, command.name) && Objects.equals(spec,
command.spec);
    }
    /**
     * Abstract method for command logic and execution.
     * @param arg String
     * @return boolean
     */
    public abstract boolean apply(String arg);
}
```

clear.java:

```
package commands;

import exceptions.InvalidElementCountException;
import utility.CollectionManager;
import utility.Console;

/**
 * Class for command clear.
 */
public class clear extends Command {
    /**
     * Initialization of class CollectionManager.
     */
    private final CollectionManager collectionManager;
    /**
     * Constructor for command class clear.
     * @param collectionManager collection manager
     */
    public clear(CollectionManager collectionManager) {
        super("clear", "clear the collection");
        this.collectionManager = collectionManager;
    }
```

```
    /**
     * Command logic - executes the command.
     * @param arg user input
     * @return Command exit status.
     */
    @Override
    public boolean apply(String arg) {
        try {
            if (!arg.isEmpty()) throw new InvalidElementCountException("No
elements in the collection", new RuntimeException());
            collectionManager.clearCollection();
            Console.println("Collection successfully cleared.");
            return true;
        } catch (InvalidElementCountException iece) {
            Console.println("Usage: '" + getName() + "';");
            return false;
        }
    }

}
```

executeScript.java:

```
package commands;
import exceptions.InvalidElementCountException;
import utility.Console;

/**
 * Class for command execute_script.
 */
public class executeScript extends Command{
    /**
     * Constructor for command class execute_script.
     */
    public executeScript(){
        super("execute_script <fileName>", "execute a script from a given
file");
    }
    /**
     * Command logic - executes the command.
     * @param arg user input
     * @return Command exit status.
     */
    @Override
    public boolean apply(String arg){
        try{
            if(arg.isEmpty()) throw new InvalidElementCountException("What
did you want to execute?", new RuntimeException());
            else Console.println("Executing script '" + arg + "', please
wait...");
            return true;
        } catch (InvalidElementCountException iece){
            Console.println("Usage: " + getName());
        }
        return false;
    }
}
```

exit.java:

```java
package commands;

import exceptions.InvalidElementCountException;
import utility.Console;

/**
 * Class for command exit.
 */
public class exit extends Command {
    /**
     * Constructor for command class exit.
     */
    public exit(){
            super("exit", "terminates the console app");
        }
    /**
     * Command logic - executes the command.
     * @param arg user input
     * @return Command exit status.
     */
    @Override
    public boolean apply(String arg){
        try{
            if(!arg.isEmpty()) throw new InvalidElementCountException("What
did you want to execute?", new RuntimeException());
            return true;
        } catch (InvalidElementCountException iece){
            Console.println("Usage: " + getName());
        }
        return false;
    }
}
```

filterContainsName.java:

```java
package commands;

import exceptions.InvalidElementCountException;
import utility.CollectionManager;
import utility.Console;
/**
 * Class for command filter_contains_name.
 */
public class filterContainsName extends Command{
    /**
     * Instance of class CollectionManager.
     */
    private final CollectionManager collectionManager;
    /**
     * Constructor for command class filter_contains_name.
     * @param collectionManager collection manager
     */
    public filterContainsName(CollectionManager collectionManager){
        super("filter_contains_name <name>", "outputs all elements that
contain the given name");
        this.collectionManager = collectionManager;
    }
    /**
     * Command logic - executes the command.
     * @param arg user input
     * @return Command exit status.
     */
```

```
    @Override
    public boolean apply(String arg){
        try {
            if (arg.isEmpty()) throw new
InvalidElementCountException("Inappropriate element count", new
RuntimeException());
            String filteredName =
collectionManager.filterContainsName(arg);
            if (filteredName.equals("{}")) {
                Console.println("Organization with the name of '" + arg +
"' not found.");
            } else {
                Console.println("Organizations with the name of '" + arg +
"': " + filteredName);
            }
            return true;
        } catch (NumberFormatException exception) {
            Console.printError("Organization name must be a String
value!");
        } catch (InvalidElementCountException exception) {
            Console.printError("Invalid argument count!");
            Console.println("Usage: '" + getName() + "'");
        }
        return false;
    }
}
```

filterLessThanType.java:

```
package commands;

import data.OrganizationType;
import exceptions.EmptyCollectionException;
import exceptions.InvalidElementCountException;
import utility.CollectionManager;
import utility.Console;

/**
 * Class for command filter_less_than_type.
 */
public class filterLessThanType extends Command{
    /**
     * Instance of class CollectionManager.
     */
    private final CollectionManager collectionManager;
    /**
     * Constructor for command class filter_less_than_type.
     * @param collectionManager collection manager.
     */
    public filterLessThanType(CollectionManager collectionManager){
        super("filter_less_than_type <type>", "outputs elements which are
less than the given type (by ordinal)");
        this.collectionManager = collectionManager;
    }
    /**
     * Command logic - executes the command.
     * @param arg user input
     * @return Command exit status.
     */
    @Override
    public boolean apply(String arg){
```

```
        try {
            if (arg.isEmpty()) throw new
InvalidElementCountException("Inappropriate element count", new
RuntimeException());
            if (collectionManager.collectionSize() == 0) throw new
EmptyCollectionException("Empty collection", new RuntimeException());
            OrganizationType organizationType =
OrganizationType.valueOf(arg.toUpperCase());
            String filteredInfo =
collectionManager.OrganizationTypeFilteredInfo(organizationType);
            if (!filteredInfo.isEmpty()) {
                Console.println(filteredInfo);
                return true;
            } else Console.println("No organizations with less value than
given type");
        } catch (InvalidElementCountException exception) {
            Console.println("Usage: '" + getName() + "'");
        } catch (EmptyCollectionException exception) {
            Console.printError("Empty collection");
        } catch (IllegalArgumentException exception) {
            Console.printError("No such type among valid organization
types!");
            Console.println("List of types - " +
OrganizationType.nameList());
        }
        return false;
    }
}
```

help.java:

```
package commands;

import exceptions.InvalidElementCountException;
import exceptions.InvalidInputException;
import utility.Console;

/**
 * Class for command help.
 */
public class help extends Command {
    /**
     * Constructor for command class help.
     */
    public help() {
        super("help", "show hint for available commands");
    }
    /**
     * Command logic - executes the command.
     * @param arg user input
     * @return Command exit status.
     */
    @Override
    public boolean apply(String arg) throws InvalidInputException {
        try {
            if (!arg.isEmpty()) throw new
InvalidElementCountException("What did you want to execute?", new
RuntimeException());
            return true;
        } catch (InvalidElementCountException exception) {
            Console.println("Usage:  '" + getName() + "'");
```

```
        }
        return false;
    }
}
```

history.java:

```
package commands;

import exceptions.InvalidElementCountException;
import utility.Console;

/**
 * Class for command history.
 */
public class history extends Command{
    /**
     * Constructor for command class history.
     */
    public history() {
        super("history", "output the history of last 9 user commands");
    }
    /**
     * Command logic - executes the command.
     * @param arg user input
     * @return Command exit status.
     */
    @Override
    public boolean apply(String arg) {
        try {
            if (!arg.isEmpty()) throw new
InvalidElementCountException("What did you want to execute?", new
RuntimeException());
            return true;
        } catch (InvalidElementCountException iece) {
            Console.println("Usage: '" + getName() + "'");
        }
        return false;
    }
}
```

info.java:

```
package commands;

import exceptions.InvalidElementCountException;
import utility.Console;
import utility.CollectionManager;

import java.time.LocalDateTime;

/**
 * Class for command info.
 */
public class info extends Command{
    /**
     * Instance of class CollectionManager.
     */
    private final CollectionManager collectionManager;
    /**
```

```
     * Constructor for command class info.
     * @param collectionManager collection manager
     */
    public info(CollectionManager collectionManager){
        super("info", "give information about the collection");
        this.collectionManager = collectionManager;
    }
    /**
     * Command logic - executes the command.
     * @param arg user input
     * @return Command exit status.
     */
    @Override
    public boolean apply(String arg){
        try{
            if(!arg.isEmpty()) throw new InvalidElementCountException("You
can't execute that", new RuntimeException());
            LocalDateTime lastInitTime =
collectionManager.getLastInitTime();
            String strLastInitTime = (lastInitTime == null) ?
"Initialization in this session has not yet happened" :
                    lastInitTime.toLocalDate().toString() + " " +
lastInitTime.toLocalTime().toString();
            LocalDateTime lastSaveTime =
collectionManager.getLastSaveTime();
            String lastSaveTimeString = (lastSaveTime == null) ? "Saving in
this session has not yet happened" :
                    lastSaveTime.toLocalDate().toString() + " " +
lastSaveTime.toLocalTime().toString();
            Console.println("Information about the collection:");
            Console.println(" Type: " +
collectionManager.getCollectionType());
            Console.println(" Element count: " +
collectionManager.collectionSize());
            Console.println(" Last save time: " + lastSaveTimeString);
            Console.println(" Last init time: " + strLastInitTime);
            return true;
        } catch (InvalidElementCountException iece){
            Console.println("Usage: " + getName());
        }
        return false;
    }
}
```

insert.java:

```
package commands;

import data.Organization;
import exceptions.IllegalKeyException;
import exceptions.InvalidInputException;
import utility.CollectionManager;
import utility.Console;
import utility.OrganizationValidator;

import java.time.Instant;

/**
 * Class for command insert.
 */
public class insert extends Command {
```

```
    /**
     * Instance of class CollectionManager.
     */
    private final CollectionManager collectionManager;
    /**
     * Instance of class organizationValidator.
     */
    private final OrganizationValidator organizationValidator;
    /**
     * Constructor for command class insert
     * @param collectionManager collection manager
     * @param organizationValidator organization validator
     */
    public insert(CollectionManager collectionManager,
OrganizationValidator organizationValidator){
        super("insert null {element}", "inserts new elements with the given
key");
        this.collectionManager = collectionManager;
        this.organizationValidator = organizationValidator;
    }
    /**
     * Command logic - executes the command.
     * @param arg user input
     * @return Command exit status.
     */
    @Override
    public boolean apply(String arg){
        try {
            if (arg.isEmpty()) throw new IllegalArgumentException("There
must be a key value", new RuntimeException());
            Integer intKey = Integer.valueOf(arg);
            if (intKey <= 0) throw new InvalidInputException("Element ID
(key) must be a positive integer", new RuntimeException());
            if (collectionManager.getCollection().containsKey(intKey)) {
                throw new IllegalKeyException("Element with this ID already
exists", new RuntimeException());
            }
            collectionManager.insertToCollection(intKey, new Organization(
                    collectionManager.generateNextId(),
                    organizationValidator.askName(),
                    organizationValidator.askCoordinates(),
                    java.util.Date.from(Instant.now()),
                    organizationValidator.askAnnualTurnover(),
                    organizationValidator.askOrganizationType(),
                    organizationValidator.askAddress()
            ));
            Console.println("Organization successfully added");
            return true;
        } catch (IllegalArgumentException exception) {
            Console.println("Usage: '" + getName() + "'");
        } catch (InvalidInputException exception) {
            Console.printError("Invalid user input; key value must be a
natural number");
        } catch (IllegalKeyException ike){
            Console.printError("Element with this ID already exists");
        }
        return false;
    }
}
```

printFieldDescendingAnnualTurnover.java:

```java
package commands;

import exceptions.EmptyCollectionException;
import exceptions.InvalidElementCountException;
import utility.CollectionManager;
import utility.Console;

/**
 * Class for command print_field_descending_annual_turnover.
 */
public class printFieldDescendingAnnualTurnover extends Command{
    /**
     * Instance of class CollectionManager.
     */
    private final CollectionManager collectionManager;
    /**
     * Constructor for command class print_field_descending_annual_turnover
     * @param collectionManager collection manager
     */
    public printFieldDescendingAnnualTurnover(CollectionManager
collectionManager){
        super("print_field_descending_annual_turnover", "output all
collections by descending order of the field annualTurnover");
        this.collectionManager = collectionManager;
    }
    /**
     * Command logic - executes the command.
     * @param arg user input
     * @return Command exit status.
     */
    @Override
    public boolean apply(String arg) {
        try {
            if (!arg.isEmpty()) throw new
InvalidElementCountException("Inappropriate element count", new
RuntimeException());
            collectionManager.printFieldDescendingAnnualTurnover();
            if (collectionManager.collectionSize() == 0) throw new
EmptyCollectionException("Empty collection", new RuntimeException());
            return true;
        } catch (InvalidElementCountException exception) {
            Console.println("Usage: '" + getName() + "'");
        } catch (EmptyCollectionException exception) {
            Console.printError("Empty collection");
        }
        return false;
    }
}
```

removeKey.java:

```java
package commands;

import data.Organization;
import exceptions.EmptyCollectionException;
import exceptions.InvalidElementCountException;
import exceptions.InvalidInputException;
import exceptions.NullOrganizationException;
import utility.CollectionManager;
import utility.Console;

/**
```

```
 * Class for command remove_key.
 */
public class removeKey extends Command {
    /**
     * Instance of class CollectionManager.
     */
    private final CollectionManager collectionManager;
    /**
     * Constructor for command class remove_key.
     * @param collectionManager collection manager
     */
    public removeKey(CollectionManager collectionManager) {
        super("remove_key <int>", "remove an element of the collection
through it's key value");
        this.collectionManager = collectionManager;
    }
    /**
     * Command logic - executes the command.
     * @param arg user input
     * @return Command exit status.
    */
 @Override
    public boolean apply(String arg) {
        try {
            if (arg.isEmpty()) throw new
InvalidElementCountException("Inappropriate element count", new
RuntimeException());
            if (collectionManager.collectionSize() == 0) throw new
EmptyCollectionException("Empty collection", new RuntimeException());
            Organization orgToFind =
collectionManager.getByID(Integer.valueOf(arg));
            if(orgToFind == null) throw new
NullOrganizationException("There's no such organization", new
RuntimeException());
            collectionManager.removeFromCollection(Integer.valueOf(arg));
            Console.println("Organization successfully deleted");
            return true;
        } catch (InvalidElementCountException exception) {
            Console.println("Usage:  " + getName() + "<int>");
        } catch (EmptyCollectionException exception) {
            Console.printError("Empty collection");
        } catch (NullOrganizationException exception) {
            Console.printError("No group with given key value");
        } catch (InvalidInputException exception) {
            Console.printError("Invalid user input");
        }
        return false;
    }
}
```

removeLower.java:

```
package commands;

import exceptions.EmptyCollectionException;
import exceptions.InvalidElementCountException;
import utility.CollectionManager;
import utility.Console;

/**
 * Class for command remove_lower.
 */
```

```
public class removeLower extends Command{
    /**
     * Instance of class CollectionManager.
     */
    private final CollectionManager collectionManager;
    /**
     * Constructor for command class remove_lower.
     * @param collectionManager collection manager
     */
    public removeLower(CollectionManager collectionManager){
        super("remove_lower <int>", "remove  all elements lesser than given
from the collection (in this case it's compared by annual turnover)");
        this.collectionManager = collectionManager;
    }
    /**
     * Command logic - executes the command.
     * @param arg user input
     * @return Command exit status.
     */
    @Override
    public boolean apply(String arg){
        try {
            if (arg.isEmpty()) throw new
InvalidElementCountException("Inappropriate element count", new
IllegalArgumentException());
            if (collectionManager.collectionSize() == 0) throw new
EmptyCollectionException("Empty collection", new RuntimeException());
            collectionManager.removeLower(Double.parseDouble(arg));
            Console.println("Organizations successfully removed!");
            return true;
        } catch (InvalidElementCountException exception) {
            Console.println("Usage: " + getName());
        } catch (IllegalArgumentException e){
            Console.println("Argument must be a double-precision float
number");
        } catch (EmptyCollectionException exception) {
            Console.printError("Empty collection");
        }
        return false;
    }
}
```

removeLowerKey.java:

```
package commands;

import data.Organization;
import exceptions.EmptyCollectionException;
import exceptions.InvalidElementCountException;
import exceptions.NullOrganizationException;
import utility.CollectionManager;
import utility.Console;

/**
 * Class for command remove_lower_key.
 */
public class removeLowerKey extends Command{
    /**
     * Instance of class CollectionManager.
     */
    private final CollectionManager collectionManager;
```

```
    /**
     * Constructor for command class remove_lower_key.
     * @param collectionManager collection manager
     */
    public removeLowerKey(CollectionManager collectionManager){
        super("remove_lower_key <int>", "remove from the collection all
elements with lower key value than given");
        this.collectionManager = collectionManager;
    }
    /**
     * Command logic - executes the command.
     * @param arg user input
     * @return Command exit status.
     */
    @Override
    public boolean apply(String arg) {
        try {
            if (arg.isEmpty()) throw new
InvalidElementCountException("Inappropriate element count", new
RuntimeException());
            if (collectionManager.collectionSize() == 0) throw new
EmptyCollectionException("Empty collection", new RuntimeException());
            int id = Integer.parseInt(arg);
            Organization groupToRemove = collectionManager.getByID(id);
            if (groupToRemove == null) throw new
NullOrganizationException("No such organization", new RuntimeException());
            collectionManager.removeLowerKey(id);
            Console.println("Organization successfully removed!");
            return true;
        } catch (InvalidElementCountException exception) {
            Console.println("Usage:  " + getName() + "<int>");
        } catch (EmptyCollectionException exception) {
            Console.printError("Empty collection");
        } catch (NumberFormatException exception) {
            Console.printError("ID must be an integer");
        } catch (NullOrganizationException exception) {
            Console.printError("There's no organization with key value less
than given");
        }
        return false;
    }
}
```

save.java:

```
package commands;

import data.Organization;
import utility.Console;
import utility.CollectionManager;
import exceptions.InvalidElementCountException;

import java.util.Hashtable;
import java.util.TreeMap;

/**
 * Class for command save.
 */
public class save extends Command{
    /**
```

```
     * Instance of class CollectionManager.
     */
    private final CollectionManager collectionManager;
    /**
     * Constructor for command class save.
     * @param collectionManager collection manager
     */
    public save(CollectionManager collectionManager){
        super("save", "saves a collection into a file");
        this.collectionManager = collectionManager;
    }

    /**
     * Command logic - executes the command.
     * @param arg user input
     * @return Command exit status.
     */
    @Override
    public boolean apply(String arg){
        try {
            if (!arg.isEmpty()) throw new
InvalidElementCountException("What did you want to execute?", new
RuntimeException());
            collectionManager.saveCollection();
            Console.println("File successfully saved");
            return true;
        } catch (InvalidElementCountException iece) {
            Console.println("Usage: '" + getName() + "'");
        }
        return false;
    }
}
```

show.java:

```
package commands;
import utility.Console;
import utility.CollectionManager;
import exceptions.InvalidElementCountException;

/**
 * Class for command show.
 */
public class show extends Command{
    /**
     * Instance of class CollectionManager
     */
    private final CollectionManager collectionManager;
    /**
     * Constructor for command class show.
     * @param collectionManager collection manager
     */
    public show(CollectionManager collectionManager){
        super("show", "output all elements of the collection");
        this.collectionManager = collectionManager;
    }
    /**
     * Command logic - executes the command.
     * @param arg user input
     * @return Command exit status.
     */
    @Override
```

```
    public boolean apply(String arg) {
        try {
            if (!arg.isEmpty()) throw new
InvalidElementCountException("What did you want to execute??", new
RuntimeException());
            Console.println(collectionManager);
            return true;
        } catch (InvalidElementCountException iece) {
            Console.println("Usage: '" + getName() + "'");
        }
        return false;
    }
}
```

updateID.java:

```
package commands;

import data.Address;
import data.Coordinates;
import data.Organization;
import data.OrganizationType;
import exceptions.EmptyCollectionException;
import exceptions.InvalidElementCountException;
import exceptions.InvalidInputException;
import exceptions.NullOrganizationException;
import utility.CollectionManager;
import utility.Console;
import utility.OrganizationValidator;
import java.util.Date;

/**
 * Class for command update id element
 */
public class updateID extends Command{
    /**
     * Instance of class CollectionManager.
     */
    private final CollectionManager collectionManager;
    /**
     * Instance of class OrganizationValidator.
     */
    private final OrganizationValidator organizationValidator;
    /**
     * Constructor for command class update.
     * @param collectionManager collection manager
     * @param organizationValidator organization validator
     */
    public updateID(CollectionManager collectionManager,
OrganizationValidator organizationValidator) {
        super("update <ID> {element}", "update element field through it's
ID");
        this.collectionManager = collectionManager;
        this.organizationValidator = organizationValidator;
    }
    /**
     * Command logic - executes the command.
     * @param arg user input
     * @return Command exit status.
     */
    @Override
```

```
    public boolean apply(String arg) {
        try {
            if (arg.isEmpty()) throw new InvalidInputException("User input
not detected", new RuntimeException());
            if (collectionManager.collectionSize() == 0) throw new
EmptyCollectionException("Empty collection", new RuntimeException());
            Integer id = Integer.valueOf(arg);
            if (!collectionManager.containsKey(id)) {
                throw new NullOrganizationException("There's no such
organization", new RuntimeException());
            } else {
                Organization org = collectionManager.getByID(id);
                String name = org.getName();
                Coordinates coordinates = org.getCoordinates();
                Date creationDate = org.getCreationDate();
                Integer annualTurnover = org.getAnnualTurnover();
                OrganizationType organizationType = org.getType();
                Address officialAddress = org.getOfficialAddress();
                collectionManager.removeFromCollection(id);
            if (organizationValidator.askQuestion("Do you want to change
the name of the organization?")) name = organizationValidator.askName();
            if (organizationValidator.askQuestion("Do you want to change
the coordinates of the organization?")) coordinates =
organizationValidator.askCoordinates();
            if (organizationValidator.askQuestion("Do you want to change
the annual turnover value of the organization?")) annualTurnover =
organizationValidator.askAnnualTurnover();
            if (organizationValidator.askQuestion("Do you want to change
the type of the organization?")) organizationType =
organizationValidator.askOrganizationType();
            if (organizationValidator.askQuestion("Do you want to change
the address of the organization")) officialAddress =
organizationValidator.askAddress();

            collectionManager.insertToCollection(id, new Organization(
                    id,
                    name,
                    coordinates,
                    creationDate,
                    annualTurnover,
                    organizationType,
                    officialAddress
            ));
            }
            Console.println("Organization successfully updated!");
            return true;
        } catch (InvalidElementCountException exception) {
            Console.println("Usage: '" + getName() + "'");
        } catch (EmptyCollectionException exception) {
            Console.printError("Empty collection!");
        } catch (NumberFormatException exception) {
            Console.printError("ID must be an integer");
        } catch (NullOrganizationException exception) {
            Console.printError("No such organization with given ID");
        } catch (InvalidInputException exception) {
            Console.printError("Invalid user input");
        }
        return false;
    }
}
```

Exceptions:

EmptyCollectionException.java:

```
package exceptions;

/**
 * Exception class for empty collections.
 */
public class EmptyCollectionException extends RuntimeException{
    /**
     * Exception message
     */
    private final String message;
    /**
     * Constructor for exception EmptyCollectionException
     * @param message String
     * @param cause String
     */
    public EmptyCollectionException(String message, Throwable cause) {
        super(message, cause);
        this.message = message;
    }
    /**
     * EmptyCollectionException implementation of general method
getMessage()
     * @return String message
     */
    @Override
    public String getMessage() {
        return message;
    }
}
```

EmptyHistoryException.java

```
package exceptions;

/**
 * Exception class for empty command history.
 */
public class EmptyHistoryException extends RuntimeException{
    /**
     * Exception message
     */
    private final String message;
    /**
     * Constructor for exception EmptyHistoryException
     * @param message String
     * @param cause String
     */
    public EmptyHistoryException(String message, Throwable cause) {
        super(message, cause);
        this.message = message;
    }
    /**
     * EmptyHistoryException implementation of general method getMessage()
     * @return String message
     */
    @Override
    public String getMessage() {
        return message;
```

```
    }
}
```

IllegalKeyException.java:

```
package exceptions;

/**
 * Exception class for illegal key value (they must be unique)
 */
public class IllegalKeyException extends RuntimeException{
    /**
     * exception message
     */
        private final String message;
    /**
     * Constructor for exception IllegalKeyException
     * @param message String
     * @param cause String
     */
        public IllegalKeyException(String message, Throwable cause) {
            super(message, cause);
            this.message = message;
        }
    /**
     * IllegalKeyException implementation of general method getMessage()
     * @return String message
     */
    @Override
        public String getMessage() {
            return message;
        }
}
```

IllegalStateException.java:

```
package exceptions;

/**
 * Exception class for disallowed program state.
 */
public class IllegalStateException extends Exception{
    /**
     * Exception message
     */
    private final String message;
    /**
     * Constructor for exception IllegalStateException
     * @param message String
     * @param cause String
     */
    public IllegalStateException(String message, Throwable cause) {
        super(message, cause);
        this.message = message;
    }
    /**
     * IllegalStateException implementation of general method getMessage()
     * @return String message
     */
    @Override
    public String getMessage() {
        return message;
```

```
    }
}
```

InvalidElementCountException.java:

```
package exceptions;

/**
 * Exception for invalid element count in collection.
 */
public class InvalidElementCountException extends IllegalArgumentException
{
    /**
     * Exception message
     */
    private final String message;
    /**
     * Constructor for exception InvalidElementCountException
     * @param message String
     * @param cause String
     */
    public InvalidElementCountException(String message, Throwable cause) {
        super(message, cause);
        this.message = message;
    }
    /**
     * InvalidElementCountException implementation of general method
getMessage()
     * @return String message
     */
    @Override
    public String getMessage() {
        return message;
    }
}
```

InvalidInputException.java:

```
package exceptions;

/**
 * Exception for invalid user input.
 */
public class InvalidInputException extends RuntimeException {
    /**
     * Exception message
     */
    private final String message;
    /**
     * Constructor for exception InvalidInputException
     * @param message String
     * @param cause String
     */
    public InvalidInputException(String message, Throwable cause) {
        super(message, cause);
        this.message = message;
    }
    /**
     * InvalidInputException implementation of general method getMessage()
     * @return String message
     */
    @Override
```

```
    public String getMessage() {
        return message;
    }
}
```

InvalidTypeException.java:

```
package exceptions;

/**
 * Exception for invalid organization type.
 */
public class InvalidTypeException extends RuntimeException{
    /**
     * Exception message
     */
    private final String message;
    /**
     * Constructor for exception InvalidTypeException
     * @param message String
     * @param cause String
     */
    public InvalidTypeException(String message, Throwable cause) {
        super(message, cause);
        this.message = message;
    }
    /**
     * InvalidTypeException implementation of general method getMessage()
     * @return String message
     */
    @Override
    public String getMessage() {
        return message;
    }
}
```

NullOrganizationException.java:

```
package exceptions;

/**
 * Exception for non-existing organization in the collection.
 */
public class NullOrganizationException extends Exception {
    /**
     * Exception message
     */
    private final String message;
    /**
     * Constructor for exception NullOrganizationException
     * @param message String
     * @param cause String
     */
    public NullOrganizationException(String message, Throwable cause) {
        super(message, cause);
        this.message = message;
    }
    /**
     * NullOrganizationException implementation of general method
getMessage()
     * @return String message
     */
```

```
    @Override
    public String getMessage() {
        return message;
    }
}
```

NullValueException.java:

```
package exceptions;

/**
 * Exception for null value in not null fields
 */
public class NullValueException extends IllegalArgumentException {
    /**
     * Exception message
     */
    private final String message;
    /**
     * Constructor for exception NullValueException
     * @param message String
     * @param cause String
     */
    public NullValueException(String message, Throwable cause) {
        super(message, cause);
        this.message = message;
    }
    /**
     * NullValueException implementation of general method getMessage()
     * @return String message
     */
    @Override
    public String getMessage() {
        return message;
    }
}
```

RecursionException.java:

```
package exceptions;

/**
 * Exception for recursion in scripts.
 */
public class RecursionException extends RuntimeException {
    /**
     * Exception message
     */
    private final String message;
    /**
     * Constructor for exception RecursionException
     * @param message String
     * @param cause String
     */
    public RecursionException(String message, Throwable cause) {
        super(message, cause);
        this.message = message;
    }
    /**
     * RecursionException implementation of general method getMessage()
     * @return String message
     */
```

```
    @Override
    public String getMessage() {
        return message;
    }
}
```

ValueExceededException.java:

```
package exceptions;

/**
 * Exception for user input values that exceed limits.
 */
public class ValueExceededException extends IllegalArgumentException{
    /**
     * Exception message
     */
    private final String message;
    /**
     * Constructor for exception ValueExceededException
     * @param message String
     * @param cause String
     */
    public ValueExceededException(String message, Throwable cause){
        super(message, cause);
        this.message = message;
    }
    /**
     * ValueExceededException implementation of general method getMessage()
     * @return String message
     */
    @Override
    public String getMessage(){
        return message;
    }
}
```

Вывод:

Несмотря на некоторые спорные моменты с оформлением задания, всё обошлось почти идеально. Программа умеет улавливать исключения и обработать их, умеет улавливать рекурсии. Лично научился работать с новыми вещами, как коллекции, потоки (не те), истинное значение за словами (String[] args). За 6 лабой!